AMD ROCm™ Optimization Guide

Release 1.0.0

Advanced Micro Devices, Inc.

Sep 29, 2026

CONTENTS

1 Release notes **3**
- 1.1 1.0.0 . . . 3

2 Introduction to optimizing AMD GPU performance **5**
- 2.1 Why GPU performance optimization matters . . . 5
- 2.2 Who should read this guide . . . 5
- 2.3 How this guide is organized . . . 6
- 2.4 What you will learn . . . 6
- 2.5 Prerequisites . . . 6

3 Understanding GPU performance **7**
- 3.1 Performance bottlenecks . . . 7
- 3.2 Roofline model . . . 7
- 3.3 Compute-bound performance . . . 9
- 3.4 Memory-bound performance . . . 9
- 3.5 Arithmetic intensity . . . 10
  - 3.5.1 Algorithmic complexity and intensity scaling . . . 10
- 3.6 Latency hiding mechanisms . . . 11
  - 3.6.1 Little's Law . . . 11
- 3.7 Wavefront execution states . . . 12
- 3.8 Occupancy theory . . . 12
- 3.9 Memory hierarchy impact on performance . . . 13
  - 3.9.1 Memory coalescing theory . . . 13
    - 3.9.1.1 Example: strided access pattern . . . 14
  - 3.9.2 Bank conflict theory . . . 15
    - 3.9.2.1 Example: conflict-free access . . . 15
    - 3.9.2.2 Example: pathological strided access (conflict-heavy) . . . 16
    - 3.9.2.3 When conflicts don't happen . . . 16
- 3.10 Register pressure theory . . . 16
  - 3.10.1 How register pressure arises . . . 16
- 3.11 Performance metrics explained . . . 17
  - 3.11.1 Peak rate . . . 17
  - 3.11.2 Utilization metrics . . . 17
    - 3.11.2.1 Pipe utilization . . . 17
    - 3.11.2.2 Issue efficiency . . . 18
    - 3.11.2.3 CU utilization . . . 18
    - 3.11.2.4 Branch efficiency . . . 19
- 3.12 Theoretical performance limits . . . 19
  - 3.12.1 Peak performance bounds . . . 19
  - 3.12.2 Achievable performance . . . 20

3.13 Summary . . . 20

4 Performance guidelines **21**
4.1 Optimization workflow . . . 21
4.2 Profiling and analysis tools . . . 21
4.2.1 rocprofv3 . . . 22
4.2.1.1 Trace visualization with Perfetto . . . 22
4.2.2 ROCprof Compute Viewer . . . 22
4.2.3 AMD System Management Interface . . . 23
4.2.4 Typical workflow . . . 23
4.3 Parallel execution . . . 24
4.3.1 Application level . . . 24
4.3.2 Device level . . . 24
4.3.3 Multiprocessor level . . . 24
4.4 Memory throughput optimization . . . 24
4.4.1 Data transfer optimization . . . 24
4.4.2 Device memory access . . . 25
4.5 Instruction throughput optimization . . . 27
4.5.1 Arithmetic instructions . . . 27
4.5.2 Control flow optimization . . . 28
4.5.3 Synchronization . . . 29
4.6 Managing register pressure . . . 29
4.7 Improving occupancy . . . 30
4.8 Minimizing memory thrashing . . . 31
4.9 Summary . . . 32

5 Multi-GPU programming **33**
5.1 Device enumeration . . . 33
5.2 Device selection . . . 34
5.3 Stream and event behavior . . . 36
5.4 Peer-to-peer memory access . . . 38

6 Introduction to optimization patterns **43**
6.1 What makes a pattern . . . 43
6.2 Patterns covered . . . 43
6.3 Learning approach . . . 43
6.4 What you will learn . . . 44
6.5 Prerequisites . . . 44

7 Optimizing reduction in HIP **45**
7.1 Prerequisites . . . 45
7.2 Reduction fundamentals . . . 45
7.3 Naive kernel . . . 48
7.3.1 What to observe . . . 48
7.4 Reducing thread divergence . . . 50
7.5 Resolving bank conflicts . . . 51
7.5.1 What to observe . . . 51
7.6 Wavefront reduction . . . 53
7.6.1 WGP mode and CU mode on RDNA GPUs . . . 56
7.6.2 What to observe . . . 56
7.7 Vectorized loads . . . 56
7.7.1 What to observe . . . 58
7.8 Device-level reduction . . . 58
7.9 Further reading . . . 59

8 Optimizing histogram in HIP **61**
8.1 Prerequisites . . . 61
8.2 Histogram fundamentals . . . 61
8.3 Race conditions . . . 62
8.4 Atomic operations . . . 62
8.5 Naive kernel . . . 63
8.5.1 What to observe . . . 64
8.6 Shared memory histogram . . . 64
8.6.1 What to observe . . . 65
8.7 Partial histograms . . . 65
8.7.1 What to observe . . . 68
8.8 Further reading . . . 68

9 Optimizing GEMM in HIP **69**
9.1 Prerequisites . . . 69
9.2 GEMM fundamentals . . . 70
9.3 Background: the GEMM arithmetic intensity . . . 70
9.4 Step 1: Naive kernel . . . 71
9.5 Step 2: LDS tiling . . . 72
9.5.1 LDS bank conflict analysis . . . 73
9.5.2 What to observe after this step . . . 74
9.6 Step 3: Register tiling . . . 74
9.6.1 What to observe . . . 77
9.6.2 Tile parameter tuning guidance . . . 77
9.7 Step 4: Double buffering . . . 78
9.7.1 What to observe . . . 86
9.8 Step 5: Vectorized loads . . . 86
9.8.1 Vectorized loads and smaller data types . . . 88
9.8.2 What to observe . . . 89
9.9 Step 6: Register pressure and occupancy . . . 89
9.9.1 Finding optimal values with rocprofv3 . . . 90
9.9.2 What to observe . . . 91
9.10 Step 7: Generic kernel . . . 92
9.10.1 Policy interfaces . . . 92
9.10.1.1 Data type scope: policy coverage . . . 93
9.10.2 Compile-time validation . . . 93
9.10.2.1 C++17 versus C++20: concept syntax . . . 95
9.10.3 Concrete policies . . . 95
9.10.4 Generic kernel template . . . 100
9.10.5 Architecture dispatch . . . 102
9.10.6 Builtin drop-in: `DirectLoadTilePolicy` . . . 103
9.11 Further reading . . . 107

10 Introduction to Compiler builtins **109**
10.1 Why Compiler builtins matter . . . 109
10.2 How this chapter is organized . . . 109
10.3 When to use builtins . . . 109
10.4 Connection to optimization patterns . . . 110
10.5 What you will learn . . . 110
10.6 Prerequisites . . . 111

11 Arithmetic and packing builtins for AMD GPUs **113**
11.1 Architecture availability . . . 113
11.2 Sum of absolute differences . . . 113

11.2.1 Masked SAD with `msad_u8` . . . 113
11.2.2 Dual-SAD packing with `sad_u8` and `sad_hi_u8` . . . 114
11.2.3 Overlapping SAD with `qsad_pk_u16_u8` . . . 115
11.3 Integer dot products . . . 115
11.3.1 Unsigned 4-element dot product with `udot4` . . . 115
11.4 FP16 dot product . . . 116
11.4.1 FP16-to-FP32 accumulation with `fdot2` . . . 116
11.5 Conversion and packing . . . 117
11.5.1 Byte packing with `cvt_pk_u8_f32` . . . 117
11.6 Naming convention . . . 118
11.7 Arithmetic and packing builtin reference . . . 118
11.8 Sum of absolute differences builtins . . . 118
11.8.1 Common parameters . . . 118
11.8.2 Architecture availability . . . 119
11.8.3 Builtin reference . . . 119
11.8.3.1 `__builtin_amdgcn_msad_u8` . . . 119
11.8.3.2 `__builtin_amdgcn_sad_u8` . . . 120
11.8.3.3 `__builtin_amdgcn_sad_hi_u8` . . . 120
11.8.3.4 `__builtin_amdgcn_sad_u16` . . . 120
11.8.3.5 `__builtin_amdgcn_qsad_pk_u16_u8` . . . 121
11.8.3.6 `__builtin_amdgcn_mqsad_pk_u16_u8` . . . 121
11.8.3.7 `__builtin_amdgcn_mqsad_u32_u8` . . . 122
11.9 Integer dot product builtins . . . 122
11.9.1 Common parameters . . . 122
11.9.2 Architecture availability . . . 122
11.9.3 Builtin reference . . . 123
11.9.3.1 `__builtin_amdgcn_udot4` . . . 123
11.9.3.2 `__builtin_amdgcn_udot8` . . . 123
11.9.3.3 `__builtin_amdgcn_sdot4` . . . 124
11.9.3.4 `__builtin_amdgcn_sdot8` . . . 124
11.9.3.5 `__builtin_amdgcn_sdot2` . . . 124
11.9.3.6 `__builtin_amdgcn_udot2` . . . 125
11.9.3.7 `__builtin_amdgcn_sudot4` . . . 125
11.9.3.8 `__builtin_amdgcn_sudot8` . . . 126
11.10 Floating-point dot product builtins . . . 126
11.10.1 Common parameters . . . 126
11.10.2 Architecture availability . . . 127
11.10.3 Builtin reference . . . 127
11.10.3.1 `__builtin_amdgcn_fdot2` . . . 127
11.10.3.2 `__builtin_amdgcn_fdot2_f16_f16` . . . 127
11.10.3.3 `__builtin_amdgcn_fdot2_bf16_bf16` . . . 128
11.10.3.4 `__builtin_amdgcn_fdot2_f32_bf16` . . . 128
11.10.3.5 `__builtin_amdgcn_dot4_f32_fp8_fp8` . . . 129
11.10.3.6 `__builtin_amdgcn_dot4_f32_bf8_bf8` . . . 129
11.10.3.7 `__builtin_amdgcn_dot4_f32_fp8_bf8` . . . 129
11.10.3.8 `__builtin_amdgcn_dot4_f32_bf8_fp8` . . . 130
11.10.3.9 `__builtin_amdgcn_fdot2c_f32_bf16` . . . 130
11.11 Conversion and packing builtins . . . 130
11.11.1 Common parameters . . . 131
11.11.2 Architecture availability . . . 131
11.11.3 Builtin reference . . . 131
11.11.3.1 FP32 to packed integer . . . 131
11.11.3.1.1 `__builtin_amdgcn_cvt_pk_u8_f32` . . . 131
11.11.3.2 FP32 to packed FP16 . . . 132

11.11.3.2.1 __builtin_amdgcn_cvt_pkrtz . . . 132
11.11.3.3 Integer packing . . . 132
11.11.3.3.1 __builtin_amdgcn_cvt_pk_i16 . . . 132
11.11.3.3.2 __builtin_amdgcn_cvt_pk_u16 . . . 133
11.11.3.4 Fixed-point conversion . . . 133
11.11.3.4.1 __builtin_amdgcn_cvt_off_f32_i4 . . . 133
11.11.3.5 Normalized packing . . . 133
11.11.3.5.1 __builtin_amdgcn_cvt_pknorm_i16 . . . 133
11.11.3.5.2 __builtin_amdgcn_cvt_pknorm_u16 . . . 134
11.11.3.6 FP8 and BF8 pack and unpack . . . 134
11.11.3.6.1 __builtin_amdgcn_cvt_pk_fp8_f32 . . . 134
11.11.3.6.2 __builtin_amdgcn_cvt_pk_bf8_f32 . . . 135
11.11.3.6.3 __builtin_amdgcn_cvt_pk_f32_fp8 . . . 135
11.11.3.6.4 __builtin_amdgcn_cvt_pk_f32_bf8 . . . 135
11.11.3.6.5 __builtin_amdgcn_cvt_f32_fp8 . . . 136
11.11.3.6.6 __builtin_amdgcn_cvt_f32_bf8 . . . 136
11.11.3.7 Stochastic rounding conversions . . . 136
11.11.3.7.1 __builtin_amdgcn_cvt_sr_fp8_f32 . . . 136
11.11.3.7.2 __builtin_amdgcn_cvt_sr_bf8_f32 . . . 137
11.11.3.7.3 __builtin_amdgcn_cvt_sr_bf16_f32 . . . 137
11.11.3.7.4 __builtin_amdgcn_cvt_sr_f16_f32 . . . 138

12 Global-to-LDS builtins for AMD GPUs **139**
12.1 Architecture availability . . . 139
12.2 Flat-addressed builtins . . . 139
12.2.1 __builtin_amdgcn_global_load_lds . . . 139
12.2.2 __builtin_amdgcn_load_to_lds . . . 140
12.3 Buffer-addressed builtins . . . 140
12.3.1 __builtin_amdgcn_raw_ptr_buffer_load_lds . . . 141
12.3.2 __builtin_amdgcn_struct_ptr_buffer_load_lds . . . 141
12.4 Cache policy (aux parameter) . . . 142
12.4.1 CDNA3 and CDNA4 . . . 143
12.4.2 CDNA and CDNA2 . . . 143
12.4.3 RDNA2 . . . 144

13 Wavefront-level builtins for AMD GPUs **145**
13.1 Architecture availability . . . 145
13.2 Lane operations . . . 145
13.2.1 Broadcasting from a specific lane with readlane . . . 145
13.2.2 Wavefront rotation using mov_dpp and ds_bpermute . . . 146
13.2.3 Wavefront rotation using __shfl . . . 147
13.2.4 Lane swap using ds_swizzle . . . 148
13.3 Wavefront reductions . . . 149
13.3.1 Reduction using __shfl_down . . . 149
13.3.2 Reduction using mov_dpp . . . 150
13.3.3 Reduction using wave_reduce_add_u32 . . . 152
13.4 Wavefront voting . . . 153
13.4.1 Compaction index using ballot and mbcnt . . . 153
13.5 Naming convention . . . 154
13.6 Wavefront builtin reference . . . 155
13.7 Shuffle and lane access builtins . . . 155
13.7.1 Common parameters . . . 156
13.7.2 Architecture availability . . . 156
13.7.3 Builtin reference . . . 156

13.7.3.1 HIP shuffles . . . 156
13.7.3.1.1 __shfl . . . 156
13.7.3.1.2 __shfl_up . . . 157
13.7.3.1.3 __shfl_down . . . 157
13.7.3.1.4 __shfl_xor . . . 157
13.7.3.2 Hardware lane access . . . 158
13.7.3.2.1 __builtin_amdgcn_readfirstlane . . . 158
13.7.3.2.2 __builtin_amdgcn_readlane . . . 158
13.7.3.2.3 __builtin_amdgcn_writelane . . . 158
13.8 DPP and data-share permutation builtins . . . 159
13.8.1 Common parameters . . . 159
13.8.2 Architecture availability . . . 159
13.8.3 Builtin reference . . . 160
13.8.3.1 DPP move operations . . . 160
13.8.3.1.1 __builtin_amdgcn_mov_dpp . . . 160
13.8.3.1.2 __builtin_amdgcn_update_dpp . . . 160
13.8.3.1.3 __builtin_amdgcn_mov_dpp8 . . . 161
13.8.3.2 Data-share permutations . . . 161
13.8.3.2.1 __builtin_amdgcn_ds_swizzle . . . 161
13.8.3.2.2 __builtin_amdgcn_ds_permute . . . 162
13.8.3.2.3 __builtin_amdgcn_ds_bpermute . . . 162
13.9 Cross-row permutation builtins . . . 163
13.9.1 Common parameters . . . 163
13.9.2 Architecture availability . . . 163
13.9.3 Builtin reference . . . 163
13.9.3.1 __builtin_amdgcn_permlane16 . . . 163
13.9.3.2 __builtin_amdgcn_permlanex16 . . . 164
13.9.3.3 __builtin_amdgcn_permlane64 . . . 164
13.9.3.4 __builtin_amdgcn_permlane16_var . . . 165
13.9.3.5 __builtin_amdgcn_permlanex16_var . . . 165
13.9.3.6 __builtin_amdgcn_permlane16_swap . . . 166
13.9.3.7 __builtin_amdgcn_permlane32_swap . . . 166
13.10 Wavefront reduction builtins . . . 166
13.10.1 Common parameters . . . 167
13.10.2 Architecture availability . . . 167
13.10.3 Builtin reference . . . 167
13.10.3.1 Addition . . . 167
13.10.3.1.1 __builtin_amdgcn_wave_reduce_add_u32 . . . 167
13.10.3.1.2 __builtin_amdgcn_wave_reduce_add_u64 . . . 167
13.10.3.2 Subtraction . . . 168
13.10.3.2.1 __builtin_amdgcn_wave_reduce_sub_u32 . . . 168
13.10.3.2.2 __builtin_amdgcn_wave_reduce_sub_u64 . . . 168
13.10.3.3 Minimum . . . 168
13.10.3.3.1 __builtin_amdgcn_wave_reduce_min_i32 . . . 168
13.10.3.3.2 __builtin_amdgcn_wave_reduce_min_u32 . . . 168
13.10.3.3.3 __builtin_amdgcn_wave_reduce_min_i64 . . . 169
13.10.3.3.4 __builtin_amdgcn_wave_reduce_min_u64 . . . 169
13.10.3.4 Maximum . . . 169
13.10.3.4.1 __builtin_amdgcn_wave_reduce_max_i32 . . . 169
13.10.3.4.2 __builtin_amdgcn_wave_reduce_max_u32 . . . 169
13.10.3.4.3 __builtin_amdgcn_wave_reduce_max_i64 . . . 170
13.10.3.4.4 __builtin_amdgcn_wave_reduce_max_u64 . . . 170
13.10.3.5 Bitwise AND . . . 170
13.10.3.5.1 __builtin_amdgcn_wave_reduce_and_b32 . . . 170

13.10.3.5.2 __builtin_amdgcn_wave_reduce_and_b64 . . . 170
13.10.3.6 Bitwise OR . . . 170
13.10.3.6.1 __builtin_amdgcn_wave_reduce_or_b32 . . . 171
13.10.3.6.2 __builtin_amdgcn_wave_reduce_or_b64 . . . 171
13.10.3.7 Bitwise XOR . . . 171
13.10.3.7.1 __builtin_amdgcn_wave_reduce_xor_b32 . . . 171
13.10.3.7.2 __builtin_amdgcn_wave_reduce_xor_b64 . . . 171
13.11 Wavefront voting and synchronization builtins . . . 171
13.11.1 Common parameters . . . 172
13.11.2 Architecture availability . . . 172
13.11.3 Builtin reference . . . 172
13.11.3.1 Ballot . . . 172
13.11.3.1.1 __builtin_amdgcn_ballot_w64 . . . 172
13.11.3.1.2 __builtin_amdgcn_ballot_w32 . . . 173
13.11.3.1.3 __builtin_amdgcn_inverse_ballot_w64 . . . 173
13.11.3.1.4 __builtin_amdgcn_inverse_ballot_w32 . . . 173
13.11.3.2 Masked bit count . . . 173
13.11.3.2.1 __builtin_amdgcn_mbcnt_lo . . . 174
13.11.3.2.2 __builtin_amdgcn_mbcnt_hi . . . 174
13.11.3.3 Synchronization . . . 174
13.11.3.3.1 __builtin_amdgcn_wave_barrier . . . 174
13.11.3.3.2 __builtin_amdgcn_wave_id . . . 175

14 MFMA builtins **177**
14.1 Dense MFMA builtins . . . 177
14.1.1 CDNA dense MFMA builtins . . . 177
14.1.1.1 Architecture availability . . . 178
14.1.1.2 Naming convention . . . 178
14.1.1.3 Accumulator layout . . . 178
14.1.1.3.1 32×32 layout . . . 179
14.1.1.3.2 16×16 layout . . . 180
14.1.1.3.3 4×4 layout . . . 182
14.1.1.4 Register types used in this reference . . . 182
14.1.1.5 Common parameters . . . 182
14.1.1.6 Using MFMA builtins as a compute policy . . . 183
14.1.1.7 Instruction throughput . . . 187
14.1.1.8 Builtin reference . . . 188
14.1.1.8.1 FP32-accumulate builtins . . . 188
14.1.1.8.1.1 FP32 matrix inputs . . . 188
14.1.1.8.1.2 __builtin_amdgcn_mfma_f32_32x32x1f32 . . . 188
14.1.1.8.1.3 __builtin_amdgcn_mfma_f32_16x16x1f32 . . . 189
14.1.1.8.1.4 __builtin_amdgcn_mfma_f32_4x4x1f32 . . . 189
14.1.1.8.1.5 __builtin_amdgcn_mfma_f32_32x32x2f32 . . . 190
14.1.1.8.1.6 __builtin_amdgcn_mfma_f32_16x16x4f32 . . . 190
14.1.1.8.1.7 FP16 matrix inputs . . . 191
14.1.1.8.1.8 __builtin_amdgcn_mfma_f32_32x32x4f16 . . . 191
14.1.1.8.1.9 __builtin_amdgcn_mfma_f32_16x16x4f16 . . . 191
14.1.1.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16 . . . 192
14.1.1.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16 . . . 192
14.1.1.8.1.12 __builtin_amdgcn_mfma_f32_16x16x16f16 . . . 193
14.1.1.8.1.13 BF16 matrix inputs . . . 194
14.1.1.8.1.14 __builtin_amdgcn_mfma_f32_32x32x2bf16 . . . 194
14.1.1.8.1.15 __builtin_amdgcn_mfma_f32_16x16x2bf16 . . . 194
14.1.1.8.1.16 __builtin_amdgcn_mfma_f32_4x4x2bf16 . . . 195

14.1.1.8.1.17 __builtin_amdgcn_mfma_f32_32x32x4bf16 . . . 195
14.1.1.8.1.18 __builtin_amdgcn_mfma_f32_16x16x8bf16 . . . 196
14.1.1.8.2 INT32-accumulate builtins . . . 196
14.1.1.8.2.1 INT8 matrix inputs . . . 196
14.1.1.8.2.2 __builtin_amdgcn_mfma_i32_32x32x4i8 . . . 196
14.1.1.8.2.3 __builtin_amdgcn_mfma_i32_16x16x4i8 . . . 197
14.1.1.8.2.4 __builtin_amdgcn_mfma_i32_4x4x4i8 . . . 198
14.1.1.8.2.5 __builtin_amdgcn_mfma_i32_32x32x8i8 . . . 198
14.1.1.8.2.6 __builtin_amdgcn_mfma_i32_16x16x16i8 . . . 199
14.1.2 CDNA2 dense MFMA builtins . . . 199
14.1.2.1 Architecture availability . . . 199
14.1.2.2 Naming convention . . . 199
14.1.2.3 Accumulator layout . . . 200
14.1.2.3.1 32×32 layout . . . 201
14.1.2.3.2 16×16 layout . . . 202
14.1.2.3.3 4×4 layout . . . 203
14.1.2.3.4 16×16 FP64 layout . . . 204
14.1.2.3.5 4×4 FP64 layout . . . 205
14.1.2.4 Register types used in this reference . . . 205
14.1.2.5 Common parameters . . . 206
14.1.2.6 Using MFMA builtins as a compute policy . . . 206
14.1.2.7 Instruction throughput . . . 211
14.1.2.8 Builtin reference . . . 211
14.1.2.8.1 FP32-accumulate builtins . . . 211
14.1.2.8.1.1 FP32 matrix inputs . . . 211
14.1.2.8.1.2 __builtin_amdgcn_mfma_f32_32x32x1f32 . . . 212
14.1.2.8.1.3 __builtin_amdgcn_mfma_f32_16x16x1f32 . . . 212
14.1.2.8.1.4 __builtin_amdgcn_mfma_f32_4x4x1f32 . . . 213
14.1.2.8.1.5 __builtin_amdgcn_mfma_f32_32x32x2f32 . . . 213
14.1.2.8.1.6 __builtin_amdgcn_mfma_f32_16x16x4f32 . . . 214
14.1.2.8.1.7 FP16 matrix inputs . . . 214
14.1.2.8.1.8 __builtin_amdgcn_mfma_f32_32x32x4f16 . . . 214
14.1.2.8.1.9 __builtin_amdgcn_mfma_f32_16x16x4f16 . . . 215
14.1.2.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16 . . . 215
14.1.2.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16 . . . 216
14.1.2.8.1.12 __builtin_amdgcn_mfma_f32_16x16x16f16 . . . 216
14.1.2.8.1.13 BF16 matrix inputs . . . 217
14.1.2.8.1.14 __builtin_amdgcn_mfma_f32_32x32x2bf16 . . . 217
14.1.2.8.1.15 __builtin_amdgcn_mfma_f32_16x16x2bf16 . . . 217
14.1.2.8.1.16 __builtin_amdgcn_mfma_f32_4x4x2bf16 . . . 218
14.1.2.8.1.17 __builtin_amdgcn_mfma_f32_32x32x4bf16 . . . 218
14.1.2.8.1.18 __builtin_amdgcn_mfma_f32_16x16x8bf16 . . . 219
14.1.2.8.1.19 __builtin_amdgcn_mfma_f32_32x32x4bf16_1k . . . 220
14.1.2.8.1.20 __builtin_amdgcn_mfma_f32_16x16x4bf16_1k . . . 220
14.1.2.8.1.21 __builtin_amdgcn_mfma_f32_4x4x4bf16_1k . . . 221
14.1.2.8.1.22 __builtin_amdgcn_mfma_f32_32x32x8bf16_1k . . . 221
14.1.2.8.1.23 __builtin_amdgcn_mfma_f32_16x16x16bf16_1k . . . 222
14.1.2.8.2 FP64-accumulate builtins . . . 222
14.1.2.8.2.1 FP64 matrix inputs . . . 222
14.1.2.8.2.2 __builtin_amdgcn_mfma_f64_16x16x4f64 . . . 222
14.1.2.8.2.3 __builtin_amdgcn_mfma_f64_4x4x4f64 . . . 223
14.1.2.8.3 INT32-accumulate builtins . . . 224
14.1.2.8.3.1 INT8 matrix inputs . . . 224
14.1.2.8.3.2 __builtin_amdgcn_mfma_i32_32x32x4i8 . . . 224

14.1.2.8.3.3 __builtin_amdgcn_mfma_i32_16x16x4i8 . . . 224
14.1.2.8.3.4 __builtin_amdgcn_mfma_i32_4x4x4i8 . . . 225
14.1.2.8.3.5 __builtin_amdgcn_mfma_i32_32x32x8i8 . . . 225
14.1.2.8.3.6 __builtin_amdgcn_mfma_i32_16x16x16i8 . . . 226
14.1.3 CDNA3 dense MFMA builtins . . . 227
14.1.3.1 Architecture availability . . . 227
14.1.3.2 Naming convention . . . 227
14.1.3.3 Accumulator layout . . . 228
14.1.3.3.1 32×32 layout . . . 228
14.1.3.3.2 16×16 layout . . . 230
14.1.3.3.3 4×4 layout . . . 231
14.1.3.3.4 16×16 FP64 layout . . . 232
14.1.3.3.5 4×4 FP64 layout . . . 233
14.1.3.4 Register types used in this reference . . . 233
14.1.3.5 Common parameters . . . 234
14.1.3.6 Using MFMA builtins as a compute policy . . . 234
14.1.3.7 Instruction throughput . . . 239
14.1.3.8 Builtin reference . . . 240
14.1.3.8.1 FP32-accumulate builtins . . . 240
14.1.3.8.1.1 FP32 matrix inputs . . . 240
14.1.3.8.1.2 __builtin_amdgcn_mfma_f32_32x32x1f32 . . . 240
14.1.3.8.1.3 __builtin_amdgcn_mfma_f32_16x16x1f32 . . . 240
14.1.3.8.1.4 __builtin_amdgcn_mfma_f32_4x4x1f32 . . . 241
14.1.3.8.1.5 __builtin_amdgcn_mfma_f32_32x32x2f32 . . . 241
14.1.3.8.1.6 __builtin_amdgcn_mfma_f32_16x16x4f32 . . . 242
14.1.3.8.1.7 FP16 matrix inputs . . . 243
14.1.3.8.1.8 __builtin_amdgcn_mfma_f32_32x32x4f16 . . . 243
14.1.3.8.1.9 __builtin_amdgcn_mfma_f32_16x16x4f16 . . . 243
14.1.3.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16 . . . 244
14.1.3.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16 . . . 244
14.1.3.8.1.12 __builtin_amdgcn_mfma_f32_16x16x16f16 . . . 245
14.1.3.8.1.13 BF16 matrix inputs . . . 245
14.1.3.8.1.14 __builtin_amdgcn_mfma_f32_32x32x4bf16_1k . . . 245
14.1.3.8.1.15 __builtin_amdgcn_mfma_f32_16x16x4bf16_1k . . . 246
14.1.3.8.1.16 __builtin_amdgcn_mfma_f32_4x4x4bf16_1k . . . 246
14.1.3.8.1.17 __builtin_amdgcn_mfma_f32_32x32x8bf16_1k . . . 247
14.1.3.8.1.18 __builtin_amdgcn_mfma_f32_16x16x16bf16_1k . . . 248
14.1.3.8.1.19 XF32 matrix inputs . . . 248
14.1.3.8.1.20 __builtin_amdgcn_mfma_f32_32x32x4_xf32 . . . 248
14.1.3.8.1.21 __builtin_amdgcn_mfma_f32_16x16x8_xf32 . . . 249
14.1.3.8.1.22 FP8 and BF8 matrix inputs . . . 249
14.1.3.8.1.23 __builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8 . . . 250
14.1.3.8.1.24 __builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8 . . . 250
14.1.3.8.1.25 __builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8 . . . 251
14.1.3.8.1.26 __builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8 . . . 251
14.1.3.8.1.27 __builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8 . . . 252
14.1.3.8.1.28 __builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8 . . . 252
14.1.3.8.1.29 __builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8 . . . 253
14.1.3.8.1.30 __builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8 . . . 254
14.1.3.8.2 FP64-accumulate builtins . . . 254
14.1.3.8.2.1 FP64 matrix inputs . . . 254
14.1.3.8.2.2 __builtin_amdgcn_mfma_f64_16x16x4f64 . . . 254
14.1.3.8.2.3 __builtin_amdgcn_mfma_f64_4x4x4f64 . . . 255
14.1.3.8.3 INT32-accumulate builtins . . . 256

14.1.3.8.3.1 INT8 matrix inputs . . . 256
14.1.3.8.3.2 __builtin_amdgcn_mfma_i32_32x32x4i8 . . . 256
14.1.3.8.3.3 __builtin_amdgcn_mfma_i32_16x16x4i8 . . . 257
14.1.3.8.3.4 __builtin_amdgcn_mfma_i32_4x4x4i8 . . . 257
14.1.3.8.3.5 __builtin_amdgcn_mfma_i32_32x32x16_i8 . . . 258
14.1.3.8.3.6 __builtin_amdgcn_mfma_i32_16x16x32_i8 . . . 258
14.1.4 CDNA4 dense MFMA builtins . . . 259
14.1.4.1 Architecture availability . . . 259
14.1.4.2 Naming convention . . . 259
14.1.4.3 Accumulator layout . . . 260
14.1.4.3.1 32×32 layout . . . 261
14.1.4.3.2 16×16 layout . . . 262
14.1.4.3.3 4×4 layout . . . 263
14.1.4.3.4 16×16 FP64 layout . . . 264
14.1.4.3.5 4×4 FP64 layout . . . 265
14.1.4.4 Register types used in this reference . . . 265
14.1.4.5 Common parameters . . . 266
14.1.4.6 Using MFMA builtins as a compute policy . . . 266
14.1.4.7 Instruction throughput . . . 272
14.1.4.8 Builtin reference . . . 273
14.1.4.8.1 FP32-accumulate builtins . . . 273
14.1.4.8.1.1 FP32 matrix inputs . . . 273
14.1.4.8.1.2 __builtin_amdgcn_mfma_f32_32x32x1f32 . . . 273
14.1.4.8.1.3 __builtin_amdgcn_mfma_f32_16x16x1f32 . . . 274
14.1.4.8.1.4 __builtin_amdgcn_mfma_f32_4x4x1f32 . . . 274
14.1.4.8.1.5 __builtin_amdgcn_mfma_f32_32x32x2f32 . . . 275
14.1.4.8.1.6 __builtin_amdgcn_mfma_f32_16x16x4f32 . . . 275
14.1.4.8.1.7 FP16 matrix inputs . . . 276
14.1.4.8.1.8 __builtin_amdgcn_mfma_f32_32x32x4f16 . . . 276
14.1.4.8.1.9 __builtin_amdgcn_mfma_f32_16x16x4f16 . . . 276
14.1.4.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16 . . . 277
14.1.4.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16 . . . 277
14.1.4.8.1.12 __builtin_amdgcn_mfma_f32_16x16x16f16 . . . 278
14.1.4.8.1.13 __builtin_amdgcn_mfma_f32_32x32x16_f16 . . . 279
14.1.4.8.1.14 __builtin_amdgcn_mfma_f32_16x16x32_f16 . . . 279
14.1.4.8.1.15 BF16 matrix inputs . . . 280
14.1.4.8.1.16 __builtin_amdgcn_mfma_f32_32x32x4bf16_1k . . . 280
14.1.4.8.1.17 __builtin_amdgcn_mfma_f32_16x16x4bf16_1k . . . 281
14.1.4.8.1.18 __builtin_amdgcn_mfma_f32_4x4x4bf16_1k . . . 281
14.1.4.8.1.19 __builtin_amdgcn_mfma_f32_32x32x8bf16_1k . . . 282
14.1.4.8.1.20 __builtin_amdgcn_mfma_f32_16x16x16bf16_1k . . . 282
14.1.4.8.1.21 __builtin_amdgcn_mfma_f32_32x32x16_bf16 . . . 283
14.1.4.8.1.22 __builtin_amdgcn_mfma_f32_16x16x32_bf16 . . . 283
14.1.4.8.1.23 FP8 and BF8 matrix inputs . . . 284
14.1.4.8.1.24 __builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8 . . . 284
14.1.4.8.1.25 __builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8 . . . 285
14.1.4.8.1.26 __builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8 . . . 285
14.1.4.8.1.27 __builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8 . . . 286
14.1.4.8.1.28 __builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8 . . . 286
14.1.4.8.1.29 __builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8 . . . 287
14.1.4.8.1.30 __builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8 . . . 287
14.1.4.8.1.31 __builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8 . . . 288
14.1.4.8.2 FP64-accumulate builtins . . . 289
14.1.4.8.2.1 FP64 matrix inputs . . . 289

14.1.4.8.2.2 __builtin_amdgcn_mfma_f64_16x16x4f64 . . . 289
14.1.4.8.2.3 __builtin_amdgcn_mfma_f64_4x4x4f64 . . . 290
14.1.4.8.3 INT32-accumulate builtins . . . 290
14.1.4.8.3.1 INT8 matrix inputs . . . 290
14.1.4.8.3.2 __builtin_amdgcn_mfma_i32_32x32x4i8 . . . 291
14.1.4.8.3.3 __builtin_amdgcn_mfma_i32_16x16x4i8 . . . 291
14.1.4.8.3.4 __builtin_amdgcn_mfma_i32_4x4x4i8 . . . 292
14.1.4.8.3.5 __builtin_amdgcn_mfma_i32_32x32x16_i8 . . . 292
14.1.4.8.3.6 __builtin_amdgcn_mfma_i32_16x16x32_i8 . . . 293
14.1.4.8.3.7 __builtin_amdgcn_mfma_i32_32x32x32_i8 . . . 293
14.1.4.8.3.8 __builtin_amdgcn_mfma_i32_16x16x64_i8 . . . 294
14.1.4.8.4 Mixed-precision with scaling . . . 295
14.1.4.8.4.1 FP8, FP6, and FP4 matrix inputs . . . 295
14.1.4.8.4.2 __builtin_amdgcn_mfma_scale_f32_32x32x64_f8f6f4 . . 295
14.1.4.8.4.3 __builtin_amdgcn_mfma_scale_f32_16x16x128_f8f6f4 . 296
14.2 Sparse MFMA builtins . . . 298
14.2.1 CDNA and CDNA2 sparse MFMA builtins . . . 298
14.2.1.1 Architecture availability . . . 298
14.2.1.2 Naming convention . . . 298
14.2.1.3 Structured sparsity (4:2 pattern) . . . 299
14.2.1.4 Accumulator layout . . . 299
14.2.1.4.1 16×16 layout . . . 299
14.2.1.4.2 32×32 layout . . . 303
14.2.1.5 Register types used in this reference . . . 303
14.2.1.6 Common parameters . . . 303
14.2.1.7 Using sparse MFMA builtins as a compute policy . . . 306
14.2.1.7.1 FP16 16×16 sparse kernel . . . 306
14.2.1.7.2 FP16 32×32 sparse kernel . . . 309
14.2.1.8 Instruction throughput . . . 310
14.2.1.9 Builtin reference . . . 310
14.2.1.9.1 FP32-accumulate builtins . . . 310
14.2.1.9.1.1 FP16 matrix inputs . . . 311
14.2.1.9.1.2 __builtin_amdgcn_smfmac_f32_16x16x32_f16 . . . 311
14.2.1.9.1.3 __builtin_amdgcn_smfmac_f32_32x32x16_f16 . . . 311
14.2.1.9.1.4 BF16 matrix inputs . . . 312
14.2.1.9.1.5 __builtin_amdgcn_smfmac_f32_16x16x32_bf16 . . . 312
14.2.1.9.1.6 __builtin_amdgcn_smfmac_f32_32x32x16_bf16 . . . 313
14.2.1.9.2 INT32-accumulate builtins . . . 313
14.2.1.9.2.1 INT8 matrix inputs . . . 313
14.2.1.9.2.2 __builtin_amdgcn_smfmac_i32_16x16x64_i8 . . . 313
14.2.1.9.2.3 __builtin_amdgcn_smfmac_i32_32x32x32_i8 . . . 314
14.2.2 CDNA3 sparse MFMA builtins . . . 314
14.2.2.1 Architecture availability . . . 315
14.2.2.2 Naming convention . . . 315
14.2.2.3 Structured sparsity (4:2 pattern) . . . 315
14.2.2.4 Accumulator layout . . . 316
14.2.2.4.1 16×16 layout . . . 316
14.2.2.4.2 32×32 layout . . . 319
14.2.2.5 Register types used in this reference . . . 319
14.2.2.6 Common parameters . . . 324
14.2.2.7 Using sparse MFMA builtins as a compute policy . . . 324
14.2.2.7.1 FP8 16×16 sparse kernel . . . 324
14.2.2.8 Instruction throughput . . . 326
14.2.2.9 Builtin reference . . . 326

14.2.2.9.1 FP32-accumulate builtins . . . 326
14.2.2.9.1.1 FP8 and BF8 matrix inputs . . . 326
14.2.2.9.1.2 __builtin_amdgcn_smfmac_f32_16x16x64_fp8_fp8 . . . 326
14.2.2.9.1.3 __builtin_amdgcn_smfmac_f32_16x16x64_fp8_bf8 . . . 327
14.2.2.9.1.4 __builtin_amdgcn_smfmac_f32_16x16x64_bf8_fp8 . . . 327
14.2.2.9.1.5 __builtin_amdgcn_smfmac_f32_16x16x64_bf8_bf8 . . . 328
14.2.2.9.1.6 __builtin_amdgcn_smfmac_f32_32x32x32_fp8_fp8 . . . 329
14.2.2.9.1.7 __builtin_amdgcn_smfmac_f32_32x32x32_fp8_bf8 . . . 329
14.2.2.9.1.8 __builtin_amdgcn_smfmac_f32_32x32x32_bf8_fp8 . . . 330
14.2.2.9.1.9 __builtin_amdgcn_smfmac_f32_32x32x32_bf8_bf8 . . . 330
14.2.3 CDNA4 sparse MFMA builtins . . . 331
14.2.3.1 Architecture availability . . . 331
14.2.3.2 Naming convention . . . 331
14.2.3.3 Structured sparsity (4:2 pattern) . . . 332
14.2.3.4 Accumulator layout . . . 332
14.2.3.4.1 16×16 layout . . . 332
14.2.3.4.2 32×32 layout . . . 333
14.2.3.5 Register types used in this reference . . . 333
14.2.3.6 Common parameters . . . 334
14.2.3.7 Using sparse MFMA builtins as a compute policy . . . 334
14.2.3.7.1 FP8 16×16 sparse kernel (K=128) . . . 334
14.2.3.8 Instruction throughput . . . 336
14.2.3.9 Builtin reference . . . 336
14.2.3.9.1 FP32-accumulate builtins . . . 336
14.2.3.9.1.1 FP16 matrix inputs . . . 336
14.2.3.9.1.2 __builtin_amdgcn_smfmac_f32_16x16x64_f16 . . . 336
14.2.3.9.1.3 __builtin_amdgcn_smfmac_f32_32x32x32_f16 . . . 337
14.2.3.9.1.4 BF16 matrix inputs . . . 338
14.2.3.9.1.5 __builtin_amdgcn_smfmac_f32_16x16x64_bf16 . . . 338
14.2.3.9.1.6 __builtin_amdgcn_smfmac_f32_32x32x32_bf16 . . . 338
14.2.3.9.1.7 FP8 and BF8 matrix inputs . . . 339
14.2.3.9.1.8 __builtin_amdgcn_smfmac_f32_16x16x128_fp8_fp8 . . . 339
14.2.3.9.1.9 __builtin_amdgcn_smfmac_f32_16x16x128_fp8_bf8 . . . 340
14.2.3.9.1.10 __builtin_amdgcn_smfmac_f32_16x16x128_bf8_fp8 . . . 340
14.2.3.9.1.11 __builtin_amdgcn_smfmac_f32_16x16x128_bf8_bf8 . . . 341
14.2.3.9.1.12 __builtin_amdgcn_smfmac_f32_32x32x64_fp8_fp8 . . . 341
14.2.3.9.1.13 __builtin_amdgcn_smfmac_f32_32x32x64_fp8_bf8 . . . 342
14.2.3.9.1.14 __builtin_amdgcn_smfmac_f32_32x32x64_bf8_fp8 . . . 343
14.2.3.9.1.15 __builtin_amdgcn_smfmac_f32_32x32x64_bf8_bf8 . . . 343
14.2.3.9.2 INT32-accumulate builtins . . . 344
14.2.3.9.2.1 INT8 matrix inputs . . . 344
14.2.3.9.2.2 __builtin_amdgcn_smfmac_i32_16x16x128_i8 . . . 344
14.2.3.9.2.3 __builtin_amdgcn_smfmac_i32_32x32x64_i8 . . . 345
14.3 CDNA4 MFMA transpose load builtins . . . 345
14.4 Architecture availability . . . 346
14.5 Naming convention . . . 346
14.6 Register types used in this reference . . . 346
14.7 Builtin reference . . . 346
14.7.1 Hardware constraints . . . 346
14.7.2 Sub-byte operand loads . . . 347
14.7.2.1 __builtin_amdgcn_ds_read_tr4_b64_v2i32 . . . 347
14.7.2.2 __builtin_amdgcn_ds_read_tr6_b96_v3i32 . . . 347
14.7.3 8-bit operand loads . . . 348
14.7.3.1 __builtin_amdgcn_ds_read_tr8_b64_v2i32 . . . 348

14.7.3.2 __builtin_amdgcn_ds_read_tr16_b64_v4i16 . . . 348
14.7.4 16-bit operand loads . . . 348
14.7.4.1 __builtin_amdgcn_ds_read_tr16_b64_v4f16 . . . 348
14.7.4.2 __builtin_amdgcn_ds_read_tr16_b64_v4bf16 . . . 349
14.8 FP16 GEMM example . . . 349
14.8.1 Step 1: store B in row-major order in LDS . . . 349
14.8.2 Step 2: load B with hardware-transposed calls . . . 351
14.9 Common MFMA parameters . . . 354
14.9.1 Shared parameters . . . 354
14.9.2 Dense MFMA parameters . . . 355
14.9.3 Sparse MFMA parameters . . . 356

15 WMMA builtins **357**
15.1 Dense WMMA builtins . . . 357
15.1.1 RDNA3 dense WMMA builtins . . . 357
15.1.1.1 Architecture availability . . . 358
15.1.1.2 Naming convention . . . 358
15.1.1.3 Fragment layouts . . . 358
15.1.1.3.1 Accumulator layout . . . 358
15.1.1.3.2 srcA and srcB (FP16 and BF16) . . . 359
15.1.1.4 Using WMMA builtins as a compute policy . . . 360
15.1.1.5 Register types used in this reference . . . 365
15.1.1.6 Common parameters . . . 366
15.1.1.7 Instruction throughput . . . 366
15.1.1.8 Builtin reference . . . 367
15.1.1.8.1 FP32-accumulate builtins . . . 367
15.1.1.8.1.1 FP16 inputs . . . 367
15.1.1.8.1.2 __builtin_amdgcn_wmma_f32_16x16x16_f16_w32 . . . 367
15.1.1.8.1.3 BF16 inputs . . . 367
15.1.1.8.1.4 __builtin_amdgcn_wmma_f32_16x16x16_bf16_w32 . . . 367
15.1.1.8.2 FP16-accumulate builtins . . . 368
15.1.1.8.2.1 FP16 inputs . . . 368
15.1.1.8.2.2 __builtin_amdgcn_wmma_f16_16x16x16_f16_w32 . . . 368
15.1.1.8.2.3 __builtin_amdgcn_wmma_f16_16x16x16_f16_tied_w32 . . . 368
15.1.1.8.3 BF16-accumulate builtins . . . 369
15.1.1.8.3.1 BF16 inputs . . . 369
15.1.1.8.3.2 __builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32 . . . 369
15.1.1.8.3.3 __builtin_amdgcn_wmma_bf16_16x16x16_bf16_tied_w32 370
15.1.1.8.4 INT32-accumulate builtins . . . 370
15.1.1.8.4.1 INT8 and UINT8 inputs . . . 370
15.1.1.8.4.2 __builtin_amdgcn_wmma_i32_16x16x16_iu8_w32 . . . 370
15.1.1.8.4.3 INT4 and UINT4 inputs . . . 371
15.1.1.8.4.4 __builtin_amdgcn_wmma_i32_16x16x16_iu4_w32 . . . 371
15.1.2 RDNA4 dense WMMA builtins . . . 371
15.1.2.1 Architecture availability . . . 372
15.1.2.2 Naming convention . . . 372
15.1.2.3 Fragment layouts . . . 372
15.1.2.3.1 Accumulator layout . . . 373
15.1.2.3.2 srcA and srcB (FP16 and BF16) . . . 374
15.1.2.4 Using WMMA builtins as a compute policy . . . 374
15.1.2.4.1 Baseline policy . . . 375
15.1.2.5 Register types used in this reference . . . 380
15.1.2.6 Common parameters . . . 381
15.1.2.7 Instruction throughput . . . 381

15.1.2.8 Builtin reference . . . 381
15.1.2.8.1 FP32-accumulate builtins . . . 381
15.1.2.8.1.1 FP16 inputs . . . 381
15.1.2.8.1.2 __builtin_amdgcn_wmma_f32_16x16x16_f16_w32_gfx12 . . . 381
15.1.2.8.1.3 BF16 inputs . . . 382
15.1.2.8.1.4 __builtin_amdgcn_wmma_f32_16x16x16_bf16_w32_gfx12 . . . 382
15.1.2.8.1.5 FP8 and BF8 inputs . . . 382
15.1.2.8.1.6 __builtin_amdgcn_wmma_f32_16x16x16_fp8_fp8_w32_gfx12 . . . 382
15.1.2.8.1.7 __builtin_amdgcn_wmma_f32_16x16x16_fp8_bf8_w32_gfx12 . . . 383
15.1.2.8.1.8 __builtin_amdgcn_wmma_f32_16x16x16_bf8_fp8_w32_gfx12 . . . 383
15.1.2.8.1.9 __builtin_amdgcn_wmma_f32_16x16x16_bf8_bf8_w32_gfx12 . . . 384
15.1.2.8.2 FP16-accumulate builtins . . . 384
15.1.2.8.2.1 FP16 inputs . . . 384
15.1.2.8.2.2 __builtin_amdgcn_wmma_f16_16x16x16_f16_w32_gfx12 . . . 384
15.1.2.8.3 BF16-accumulate builtins . . . 385
15.1.2.8.3.1 BF16 inputs . . . 385
15.1.2.8.3.2 __builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32_gfx12 . . . 385
15.1.2.8.4 INT32-accumulate builtins . . . 385
15.1.2.8.4.1 INT8 and UINT8 inputs (16x16x16) . . . 385
15.1.2.8.4.2 __builtin_amdgcn_wmma_i32_16x16x16_iu8_w32_gfx12 . . . 385
15.1.2.8.4.3 INT4 and UINT4 inputs (16x16x16) . . . 386
15.1.2.8.4.4 __builtin_amdgcn_wmma_i32_16x16x16_iu4_w32_gfx12 . . . 386
15.1.2.8.4.5 INT4 and UINT4 inputs (16x16x32) . . . 386
15.1.2.8.4.6 __builtin_amdgcn_wmma_i32_16x16x32_iu4_w32_gfx12 . . . 387
15.2 RDNA4 sparse WMMA builtins . . . 387
15.2.1 Architecture availability . . . 388
15.2.2 Naming convention . . . 388
15.2.3 Structured sparsity (2:4 pattern) . . . 388
15.2.4 Fragment layouts . . . 388
15.2.4.1 Accumulator layout . . . 389
15.2.4.2 srcA (sparse, FP16 and BF16) . . . 390
15.2.4.3 srcB (dense, FP16 and BF16) . . . 390
15.2.5 Register types used in this reference . . . 392
15.2.6 Common parameters . . . 392
15.2.7 Using sparse WMMA builtins as a compute policy . . . 393
15.2.8 Instruction throughput . . . 398
15.2.9 Builtin reference . . . 398
15.2.9.1 FP32-accumulate builtins . . . 398
15.2.9.1.1 FP16 inputs . . . 398
15.2.9.1.1.1 __builtin_amdgcn_swmmac_f32_16x16x32_f16_w32 . . . 398
15.2.9.1.2 BF16 inputs . . . 399
15.2.9.1.2.1 __builtin_amdgcn_swmmac_f32_16x16x32_bf16_w32 . . . 399
15.2.9.1.3 FP8 and BF8 inputs . . . 399
15.2.9.1.3.1 __builtin_amdgcn_swmmac_f32_16x16x32_fp8_fp8_w32 . . . 399
15.2.9.1.3.2 __builtin_amdgcn_swmmac_f32_16x16x32_fp8_bf8_w32 . . . 400
15.2.9.1.3.3 __builtin_amdgcn_swmmac_f32_16x16x32_bf8_fp8_w32 . . . 400
15.2.9.1.3.4 __builtin_amdgcn_swmmac_f32_16x16x32_bf8_bf8_w32 . . . 401
15.2.9.2 FP16-accumulate builtins . . . 401
15.2.9.2.1 FP16 inputs . . . 401
15.2.9.2.1.1 __builtin_amdgcn_swmmac_f16_16x16x32_f16_w32 . . . 401
15.2.9.3 BF16-accumulate builtins . . . 402
15.2.9.3.1 BF16 inputs . . . 402
15.2.9.3.1.1 __builtin_amdgcn_swmmac_bf16_16x16x32_bf16_w32 . . . 402
15.2.9.4 INT32-accumulate builtins . . . 402

15.2.9.4.1 INT8 and UINT8 inputs (16x16x32) . . . 403
15.2.9.4.1.1 __builtin_amdgcn_swmmac_i32_16x16x32_iu8_w32 . . . 403
15.2.9.4.2 INT4 and UINT4 inputs (16x16x32) . . . 403
15.2.9.4.2.1 __builtin_amdgcn_swmmac_i32_16x16x32_iu4_w32 . . . 403
15.2.9.4.3 INT4 and UINT4 inputs (16x16x64) . . . 404
15.2.9.4.3.1 __builtin_amdgcn_swmmac_i32_16x16x64_iu4_w32 . . . 404
15.3 RDNA4 WMMA transpose load builtins . . . 405
15.3.1 Architecture availability . . . 405
15.3.2 Naming convention . . . 405
15.3.3 Register types used in this reference . . . 405
15.3.4 Hardware constraints . . . 406
15.3.5 Per-lane fragment layout . . . 406
15.3.6 Builtin reference . . . 406
15.3.6.1 __builtin_amdgcn_global_load_tr_b64_v2i32 . . . 406
15.3.6.2 __builtin_amdgcn_global_load_tr_b128_v8i16 . . . 407
15.3.6.3 __builtin_amdgcn_global_load_tr_b128_v8f16 . . . 407
15.3.6.4 __builtin_amdgcn_global_load_tr_b128_v8bf16 . . . 407
15.3.7 Using a transpose load as a compute policy . . . 408
15.3.8 Compiling and running . . . 412

The AMD ROCm Optimization Guide focuses on performance optimization techniques for AMD GPUs using the HIP programming language. This guide provides comprehensive patterns and best practices for maximizing GPU performance, covering essential topics such as parallel workload optimization, reduction operations, memory coalescing, and multi-GPU programming.

Getting started

- *Introduction to optimizing AMD GPU performance*
- *Understanding GPU performance*
- *Performance guidelines*
- *Multi-GPU programming*

Performance optimization patterns

- *Introduction to optimization patterns*
- *Optimizing reduction in HIP*
- *Optimizing histogram in HIP*
- *Optimizing GEMM in HIP*

Compiler builtins

- *Introduction to Compiler builtins*
- *Arithmetic and packing builtins for AMD GPUs*
  - *Sum of absolute differences builtins*
  - *Integer dot product builtins*
  - *Floating-point dot product builtins*
  - *Conversion and packing builtins*
- *Global-to-LDS builtins for AMD GPUs*
- *Wavefront-level builtins for AMD GPUs*
  - *Shuffle and lane access builtins*
  - *DPP and data-share permutation builtins*
  - *Cross-row permutation builtins*
  - *Wavefront reduction builtins*
  - *Wavefront voting and synchronization builtins*
- *MFMA builtins*
  - *Dense MFMA builtins*
    - *CDNA dense MFMA builtins*
    - *CDNA2 dense MFMA builtins*
    - *CDNA3 dense MFMA builtins*
    - *CDNA4 dense MFMA builtins*
  - *Sparse MFMA builtins*
    - *CDNA and CDNA2 sparse MFMA builtins*
    - *CDNA3 sparse MFMA builtins*

        * *CDNA4 sparse MFMA builtins*
    - *CDNA4 MFMA transpose load builtins*
    - *Common MFMA parameters*
- *WMMA builtins*
    - *Dense WMMA builtins*
        * *RDNA3 dense WMMA builtins*
        * *RDNA4 dense WMMA builtins*
    - *RDNA4 sparse WMMA builtins*
    - *RDNA4 WMMA transpose load builtins*

Known issues are listed and can be reported on the AMD ROCm Optimization Guide GitHub repository.

To contribute to the documentation, see Contributing to ROCm docs for contribution guidelines.

You can find licensing information on the Licensing page.

CHAPTER

ONE

RELEASE NOTES

This page tracks releases of the AMD ROCm Optimization Guide.

Version	Release date
1.0.0	October 5, 2026

1.1 1.0.0

Initial release of the AMD ROCm Optimization Guide.

CHAPTER

TWO

INTRODUCTION TO OPTIMIZING AMD GPU PERFORMANCE

Achieving optimal GPU performance requires more than writing correct code. Understanding the underlying hardware architecture, memory hierarchies, and execution patterns will help you improve AMD GPU performance. These ROCm-supported GPUs offer massive parallel processing capabilities, but realizing their full potential demands careful attention to how you structure your code and access data.

This guide focuses on performance optimization techniques for AMD GPUs using the HIP programming language. Whether you are optimizing existing applications or designing new high-performance systems, the material here will help you extract maximum performance from AMD hardware.

Performance optimization is an iterative process that requires measurement, analysis, and refinement. The most effective approach is to profile your application first to identify actual bottlenecks, then apply targeted optimizations to address those specific issues. Begin with the foundational chapters on understanding GPU performance and general optimization guidelines. Then work through the optimization patterns to see practical techniques in action. Finally, explore Compiler builtins to use architecture-specific features for maximum performance.

2.1 Why GPU performance optimization matters

GPU acceleration offers tremendous performance potential, but achieving that potential requires deliberate optimization. The same algorithm can exhibit dramatically different performance characteristics depending on how you implement it. Small changes in memory access patterns or thread organization can yield order-of-magnitude performance improvements.

AMD GPUs have sophisticated memory hierarchies, execution pipelines, and resource constraints. Understanding these architectural details enables you to make informed optimization decisions. Additionally, performance that scales well on one problem size or architecture might not scale to others, making it essential to understand fundamental optimization principles rather than architecture-specific tricks.

2.2 Who should read this guide

This guide is intended for developers with a basic familiarity with HIP programming who want to deepen their understanding of GPU performance optimization. You will benefit most from this material if you:

- Have existing HIP or NVIDIA CUDA code that you want to optimize
- Are designing GPU-accelerated applications with performance requirements
- Need to understand performance profiling and analysis techniques
- Want to learn practical optimization patterns and recommended practices

Performance engineers and GPU programmers will find practical techniques for identifying and resolving performance bottlenecks. HPC and AI practitioners will discover strategies for scaling their workloads efficiently across AMD GPU architectures.

2.3 How this guide is organized

This guide takes a structured approach to GPU optimization, progressing from foundational concepts to practical patterns to hardware-specific techniques.

The guide begins with fundamental performance concepts and general optimization guidelines that apply across most GPU workloads. You will examine performance metrics, identify bottlenecks, and apply proven optimization strategies for memory access and thread organization.

Next, the guide explores optimization patterns through complete, working examples. You will implement and optimize reduction algorithms, histogram computations, and matrix multiplication using standard HIP constructs. Each pattern demonstrates a specific optimization journey from naive implementation to production-quality code.

Finally, the guide introduces Compiler builtins and specialized instructions that provide direct access to GPU features that compilers cannot automatically generate. You will learn to use architecture-specific capabilities for maximum performance on CDNA and RDNA GPUs.

2.4 What you will learn

Working through this guide will help you:

- Understand fundamental GPU performance concepts and identify bottlenecks
- Apply memory access optimizations and efficient thread organization
- Implement efficient parallel algorithms using proven patterns
- Use Compiler builtins for architecture-specific acceleration
- Measure optimization impact and make data-driven decisions

Each chapter includes complete, runnable examples with performance measurements, helping you understand which techniques matter most and how to adapt them to your own applications.

2.5 Prerequisites

Before working through this guide, you should have:

- Basic familiarity with HIP programming, including kernel launches, thread indexing, and memory management
- Understanding of C++ programming fundamentals
- Knowledge of basic parallel programming concepts
- Access to AMD GPU hardware and ROCm software stack for running examples

CHAPTER

THREE

UNDERSTANDING GPU PERFORMANCE

This topic explains the theoretical foundations of GPU performance on AMD hardware. Understanding these concepts helps you analyze performance characteristics, identify bottlenecks, and make informed optimization decisions.

For practical optimization techniques and step-by-step guidance, see *Performance guidelines*.

3.1 Performance bottlenecks

The neck of a bottle limits the rate at which liquid can be poured. A performance bottleneck in a computing system similarly limits the rate at which work can be completed.

A performance bottleneck is the limiting factor that prevents a GPU kernel from achieving higher performance. Understanding which bottleneck applies helps identify the appropriate optimization approach.

Performance bottlenecks for GPU kernels fall into three main categories:

- **Compute-bound**: The kernel is limited by arithmetic throughput (the arithmetic bandwidth of compute units)
- **Memory-bound**: The kernel is limited by memory bandwidth (how quickly data can move between High Bandwidth Memory (HBM) and on-chip caches or Local Data Share (LDS))
- **Overhead-bound**: The kernel is limited by latency (host-side scheduling, kernel launch overhead, or small array operations)

This categorization aligns with the textbook approach to optimization: determine the bottleneck, elevate the bottleneck until it is no longer limiting, and repeat on the new bottleneck.

Roofline model analysis helps quickly identify whether a kernel's performance is bottlenecked by compute throughput or memory bandwidth.

3.2 Roofline model

The roofline model is a simplified, visual model of performance used to quickly determine whether a program is limited by memory bandwidth or arithmetic bandwidth.

In the roofline model, two hardware-derived "roofs" place upper bounds—or ceilings—on achievable performance:

- **Compute roof**: The peak arithmetic rate of the target hardware (vector Arithmetic Logic Units (ALUs) or Matrix Fused Multiply-Add (MFMA) units), sometimes referred to as its arithmetic bandwidth
- **Memory roof**: The peak data transfer rate of the memory subsystem, or memory bandwidth

These are plotted on a plane with arithmetic intensity (operations per byte) on the x-axis and performance (operations per second) on the y-axis.

The compute roof is a horizontal line at a height equal to the hardware's maximum arithmetic throughput. The memory roof is a slanted line whose slope equals the memory bandwidth (in bytes per second). Because slope is "rise over run," its units correspond to throughput per intensity.

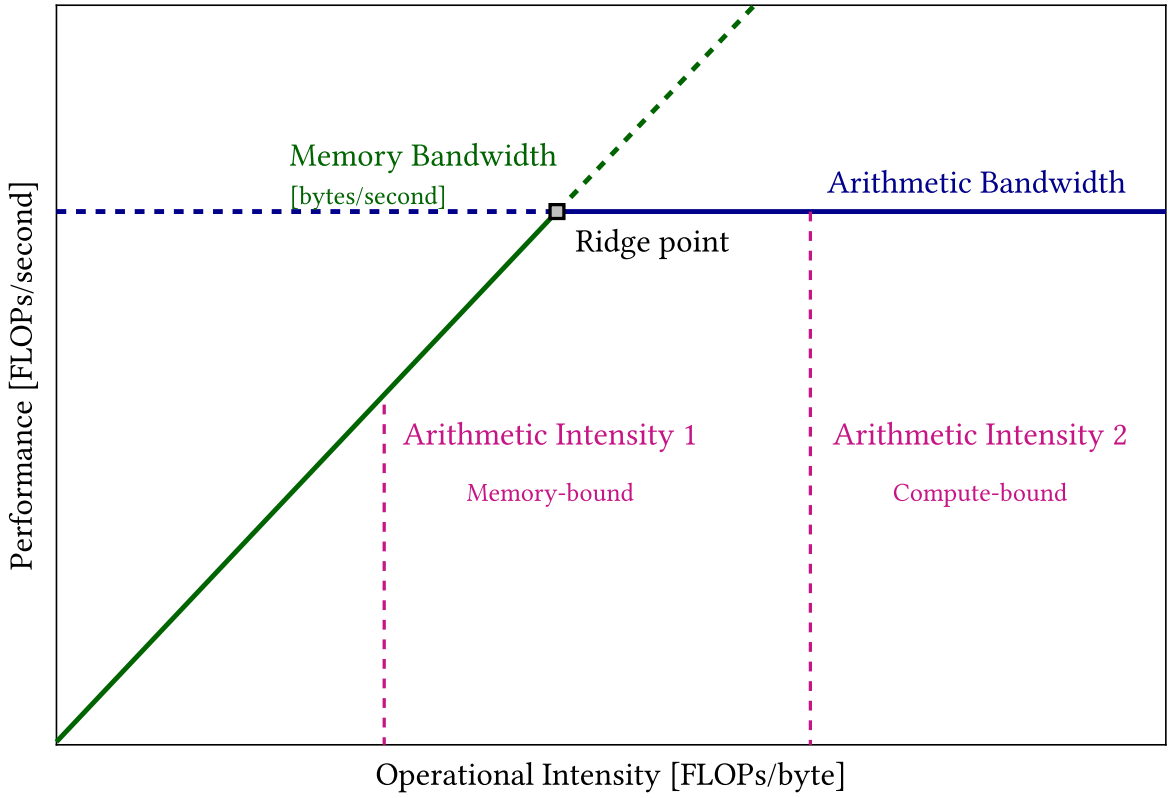


Fig. 1: Roofline model showing the relationship between arithmetic intensity and achievable performance. The memory bandwidth ceiling represents the GPU's memory bandwidth limit, while the compute ceiling shows the maximum achievable TFLOPs. Kernels falling into the area to the left of the ridge point are memory-bound, while they are compute-bound if they fall into the right area.

A kernel's position on the x-axis indicates whether it is fundamentally compute-bound (beneath the flat roof) or memory-bound (beneath the slanted roof). In practice, few kernels ever fully reach either roof due to overhead, latency, and control-divergence effects.

The point where the diagonal and horizontal roofs intersect is called the ridge point. Its x-coordinate gives the minimum arithmetic intensity required to escape the memory bottleneck. Systems with ridge points farther to the left are easier to saturate, but over time, improvements in compute throughput have far outpaced memory growth—pushing ridge points steadily to the right.

On AMD platforms, roofline analysis is built into ROCm's profiling and performance visualization ecosystem. rocprofv3 can be used to gather achieved FLOPs, memory transactions, and operational intensity.

The roofline model's elegance lies in its simplicity—but also in its deliberate omissions. It ignores latency entirely, focusing only on sustained throughput limits. Understanding those assumptions—and when they hold—is essential to applying the model correctly.

3.3 Compute-bound performance

A kernel is compute-bound when its performance is limited by the GPU's arithmetic throughput rather than memory bandwidth. These kernels have high *arithmetic intensity*, spending most cycles executing arithmetic operations.

Kernels that are compute-bound are limited by the arithmetic bandwidth of the GPU's Compute Units (CUs)—on AMD architectures, this means the vector ALUs and MFMA units within each CU or Single Instruction Multiple Data (SIMD) unit.

Characteristics of compute-bound kernels:

- High ratio of arithmetic operations to memory accesses (high arithmetic intensity)
- Performance scales with GPU compute capacity
- Limited benefit from memory bandwidth optimization
- Can often achieve a high percentage of peak theoretical FLOPS
- The limiting factor is utilization of arithmetic pipelines: the number of concurrent floating-point or integer operations the GPU can sustain per clock

The theoretical maximum is determined by:

- Number of compute units and SIMD lanes
- Clock frequency
- Instruction throughput per cycle
- Specialized unit capabilities (matrix cores, Special Function Units (SFUs))

3.4 Memory-bound performance

A kernel is memory-bound when its performance is limited by memory bandwidth rather than compute capacity. These kernels have low *arithmetic intensity* and spend significant time waiting for memory operations.

Kernels that are memory-bound are limited by the memory bandwidth of the GPU—that is, by how quickly data can move between HBM and the on-chip caches or LDS of the GPU's compute units.

Memory-bound kernels are limited by the bandwidth between GPU RAM and local caches because the working sets of most real-world GPU workloads are far larger than any higher level of the memory hierarchy. When data reuse is low and arithmetic operations per byte are few, the speed of computation is dominated by how quickly memory can feed operands to the arithmetic units.

Characteristics of memory-bound kernels:

- Low ratio of arithmetic operations to memory accesses (low arithmetic intensity)
- Performance scales with memory bandwidth
- Sensitive to memory access patterns
- Typically achieve lower percentage of peak FLOPS
- Fall to the left of the ridge point on the roofline diagram

The theoretical maximum is determined by:

- HBM bandwidth capacity
- Memory controller efficiency
- Cache hierarchy effectiveness

- Memory access pattern efficiency

3.5 Arithmetic intensity

Arithmetic intensity is the ratio of arithmetic operations to memory operations in a kernel. It is the ratio of floating-point operations (FLOPs) to memory traffic (bytes) for a given kernel or algorithm.

$$\text{Arithmetic Intensity} = \frac{\text{FLOPs}}{\text{Bytes Transferred}}$$

This metric determines whether a kernel is compute-bound or memory-bound.

Key points:

- Higher arithmetic intensity indicates more computation per byte transferred
- A high arithmetic intensity indicates that a kernel performs many arithmetic operations per byte loaded
- The balance point (ridge point) depends on the GPU's compute-to-bandwidth ratio
- It can be calculated theoretically or measured empirically
- Different precision types affect both FLOPs and bytes

For modern AMD GPUs:

- The compute-to-bandwidth ratio varies by GPU generation
- Higher-end models have higher ratios
- Kernels above the GPU's specific ratio (ridge point) are compute-bound
- Because modern GPUs deliver far more arithmetic throughput than memory bandwidth, the most efficient kernels are those with high arithmetic intensity

3.5.1 Algorithmic complexity and intensity scaling

Because algorithms have different operational and memory complexities, they scale differently in arithmetic intensity:

- An algorithm with $\mathcal{O}(1)$ operations and $\mathcal{O}(N)$ memory has $\mathcal{O}(\frac{1}{N})$ intensity (decreasing with size)
- One with $\mathcal{O}(N)$ operations and $\mathcal{O}(1)$ memory has $\mathcal{O}(N)$ intensity (increasing with size)

Examples of kernel complexity scaling:

- **SAXPY** ($y = ax + y$): $2N$ FLOPs, $8N$ bytes → intensity $\frac{1}{4}$ → $\mathcal{O}(1)$ scaling
- **Single-Precision Real Fast Fourier Transform (FFT)**: $\frac{5}{2}N \log N$ FLOPs, $16N$ bytes → intensity $\frac{5}{32} \log N$ → $\mathcal{O}(\log N)$ scaling
- **SGEMM** (matrix multiplication): $2N^3$ FLOPs, $6N^2$ bytes → $\mathcal{O}(N)$ scaling

Matrix multiplication scales linearly in arithmetic intensity— $\mathcal{O}(N^3)$ operations versus $\mathcal{O}(N^2)$ memory—making it an ideal match for high-throughput architectures. This favorable scaling is a key reason why many machine-learning algorithms built around dense linear algebra achieve high GPU utilization.

Techniques that shift memory transfers to additional compute operations reduce memory traffic but increase arithmetic load, thereby raising the arithmetic intensity. For example:

- Compressing data in global memory reduces bytes transferred but adds decompression arithmetic—raising arithmetic intensity
- In training and inference of neural networks, techniques like gradient checkpointing reduce memory storage of activations (fewer bytes stored and loaded) but add recompute work—again increasing arithmetic intensity

3.6 Latency hiding mechanisms

GPUs hide memory and instruction latency through massive hardware multithreading rather than complex CPU techniques like out-of-order execution.

Latency hiding is the strategy of masking long-latency operations by running them concurrently. On AMD GPUs, performant kernels interleave the execution of many threads across wavefronts keeping overall throughput high even when individual instructions take many cycles. When one wavefront stalls on a slow global-memory access, the scheduler immediately issues instructions from another eligible wavefront.

How latency hiding works:

- **wavefront switching**: Context switches occur every cycle with zero overhead
- **Multiple wavefronts per CU**: Many concurrent wavefronts supported
- **Instruction-level parallelism**: Multiple independent instructions in flight

This keeps the compute units busy: while one wavefront drives MFMA matrix ops, another runs scalar and vector ALU work (e.g., quantize and dequantize), and a third issues loads and stores through the memory pipeline (LDS, L1, and L2 ↔ HBM).

The hardware can completely hide memory latency if there are enough active wavefronts with independent work. The number of instructions required from other wavefronts to hide latency depends on the GPU's specific memory latency and instruction throughput characteristics.

3.6.1 Little's Law

Little's Law relates concurrency (how much work is in flight) to latency and throughput:

$$\text{concurrency (ops)} = \text{latency (s)} \times \text{throughput (ops/s)}$$

In GPU terms, it tells you how much independent work you must have in flight to hide latency via fine-grained scheduling. On AMD GPUs, wavefront switching by the wavefront schedulers is the primary latency-hiding mechanism.

Little's Law determines how many independent memory transactions or instructions must be outstanding across active wavefronts to keep the compute units busy.

Concretely, consider a simple sequence in AMD CDNA terms:

```
global_load_dword     v1, v[2:3], off   ; long-latency global load (hundreds of cycles)
v_mul_lo_u32          v2, v1, 0xBEEF    ; integer multiply
v_add_u32             v4, v2, 0xAFFE    ; integer add
v_mul_lo_u32          v6, v4, 0x1337    ; integer multiply
```

Executed strictly serially, the total time is dominated by the global load. Using Little's Law, if your effective issue rate is ~1 inst/cycle and the load takes hundreds of cycles, you need hundreds of independent in-flight operations to finish, on average, one such sequence per cycle—hiding the memory latency from consumers.

Issuance occurs at the wavefront granularity (64 threads per wavefront on CDNA). When latency hiding is successful, the CU maintains enough active wavefronts and rapidly context-switches among them whenever one blocks, so execution units don't sit idle waiting on memory or other long-latency events.

Requirements for effective latency hiding:

- Sufficient occupancy (active wavefronts)
- Independent instructions to overlap
- Balanced resource usage
- Minimal divergence

3.7 Wavefront execution states

The state of the wavefronts executing a kernel on an AMD GPU can be described using several non-exclusive terms—active, stalled, eligible, and selected.

A wavefront is considered **active** from the time its threads begin executing until all threads in that wavefront have completed the kernel. The wavefront schedulers select wavefronts from the active pool each cycle; the selected wavefronts then issue their instructions.

An **eligible** wavefront is an active wavefront ready to issue its next instruction. For a wavefront to be eligible:

- Its next instruction has been fetched
- The required pipeline (vector ALU, MFMA, or memory) is available
- All data dependencies have been resolved
- No synchronization barriers (for example, `s_barrier`) are pending

Eligible wavefronts are the immediate candidates for issue. A lack of eligible wavefronts often indicates dependency or memory stalls—a key target in performance tuning.

A **stalled** wavefront is active but unable to issue its next instruction due to resource or data hazards. Common causes include:

- Execution dependencies: waiting for results from previous ALU or MFMA operations
- Memory dependencies: waiting for global or LDS memory fetches
- Pipeline conflicts: required execution units are occupied

AMD hardware uses a scoreboard mechanism to track outstanding dependencies per wavefront. When waiting on LDS or ALU results, a wavefront is said to be on the short scoreboard; when waiting on off-chip HBM accesses, it is on the long scoreboard. This scoreboarding approach—originally from the CDC 6600 supercomputer—allows dynamic scheduling across wavefronts (thread-level parallelism) rather than within them (instruction-level parallelism).

A **selected** wavefront is an eligible one chosen by the wavefront scheduler to issue an instruction in the current cycle. Each CU typically has multiple schedulers that can each issue one instruction per cycle from their eligible pool.

Understanding these states helps explain GPU utilization metrics:

- **Active cycles**: Percentage of cycles with at least one instruction executing
- **Stall cycles**: Percentage of cycles waiting for resources
- **Idle cycles**: No wavefronts available to execute

Maximizing active cycles while minimizing stall and idle cycles improves performance. Effective latency hiding on AMD hardware relies on keeping enough active and eligible wavefronts resident so that the schedulers always have work to select, ensuring the CU pipelines remain fully utilized.

3.8 Occupancy theory

Occupancy measures the ratio of active wavefront to the maximum possible wavefronts on a compute unit.

$$\text{Occupancy} = \frac{\text{Active wavefronts}}{\text{Max wavefronts per CU}}$$

There are two common ways to measure it:

- **Theoretical occupancy**: The upper limit determined by the kernel's launch configuration (threads per block, register use, LDS use) and the hardware limits of the CU
- **Achieved occupancy**: The actual number of wavefronts active during kernel execution, i.e., on active cycles

As part of the AMD execution model, all threads in a block are scheduled to the same CU. Each CU has finite resources—Vector General-Purpose Registers (VGPRs), Scalar General-Purpose Registers (SGPRs), LDS (shared memory), and wave slots—that must be shared among all resident blocks. These constraints jointly determine the maximum number of active wavefronts.

Why occupancy matters:

- Higher occupancy improves *latency hiding*
- More concurrent wavefronts mask memory and instruction latency
- Enables better utilization of execution units

Limiting factors:

- **Register usage**: VGPRs and SGPRs per thread
- **Shared memory (LDS)**: Allocation per block
- **Wavefront slots**: Hardware limit on concurrent wavefronts
- **Block size**: Small blocks may waste resources

Trade-offs:

- Higher occupancy improves latency hiding but reduces resources per thread
- Lower occupancy allows more resources per thread but may expose latency
- Optimal occupancy depends on kernel characteristics
- Memory-bound kernels benefit more from high occupancy

Low occupancy often reduces performance when there aren't enough eligible wavefronts to hide memory or arithmetic latency, causing low issue efficiency and underutilized pipelines. However, once occupancy is sufficient for latency hiding, increasing it further can hurt performance by reducing the number of available registers or LDS per wavefront—both of which can limit *arithmetic intensity*.

In short, occupancy measures how fully a CU is loaded, not how efficiently it is utilized. High-performance kernels (for example, MFMA-based GEMMs on CDNA) often operate at low occupancy because only a few wavefronts are needed to fully saturate the MFMA and memory pipelines.

3.9 Memory hierarchy impact on performance

The GPU memory hierarchy has different bandwidths and latencies:

Memory types by speed:

1. **Registers**: Fastest, lowest latency (per-thread storage)
2. **LDS (shared memory)**: Very fast, on-chip (per-block storage)
3. **L1 cache**: Fast, on-chip (per-CU cache)
4. **L2 cache**: Moderate, on-chip (shared across CUs)
5. **HBM (global memory)**: Slower, off-chip but high bandwidth

3.9.1 Memory coalescing theory

Memory coalescing is a hardware technique that improves effective memory bandwidth by servicing many logical loads or stores with a small number of physical memory transactions.

Memory coalescing combines memory accesses from multiple threads into fewer transactions. When consecutive threads access consecutive memory addresses, the hardware can merge requests into efficient cache line accesses.

Memory coalescing is relevant when accessing global memory (HBM and GDDR attached to the GPU). For efficient access to LDS and shared memory, see the discussion of bank conflicts below.

On AMD GPUs, global memory is backed by HBM (in data-center parts) or GDDR (on many client parts). These Dynamic Random-Access Memory (DRAM) technologies provide very high bandwidth but have relatively long access latency. If each thread's load or store were always turned into its own physical DRAM transaction, the GPU would leave a large fraction of that raw bandwidth unused.

Coalescing takes advantage of how DRAM is organized internally. When a DRAM address is accessed, the hardware actually fetches or writes a burst: a run of consecutive addresses fetched together in a single transaction. If multiple threads in a wavefront access addresses that fall into the same burst, those logical accesses can be coalesced into that single transaction.

This fits naturally with the wavefront execution model: in normal execution, all threads in a wavefront execute the same instruction at the same time. If each lane in the wavefront loads or stores a value from a contiguous region (e.g., lane *i* accesses `base + i`), the memory system can typically serve the entire wavefront's request with a small number of large, aligned bursts. When addresses are scattered (e.g., large strides or irregular indexing), more bursts are needed, and effective bandwidth drops.

Why coalescing matters:

- Reduces number of memory transactions
- Improves memory bandwidth utilization
- Decreases memory access latency

Coalesced pattern: Consecutive threads accessing consecutive addresses achieve high bandwidth utilization.

Non-coalesced pattern: Random or strided addresses result in many separate transactions and low bandwidth utilization.

3.9.1.1 Example: strided access pattern

Consider this kernel that reads from an input array with a configurable stride (distance between consecutive elements accessed by each thread):

```
__global__ void strided_read_kernel(const float* __restrict__ in,
                                    float* __restrict__ out,
                                    std::size_t N, std::size_t stride)
{
    const std::size_t t  = blockIdx.x * blockDim.x + threadIdx.x;
    const std::size_t T  = gridDim.x * blockDim.x;

    float acc = 0.f;

    for (std::size_t j = t * stride; j < N; j += T * stride)
    {
        // across a wavefront, addresses differ by (stride * sizeof(float))
        float v = in[j]; // perfectly coalesced for stride == 1
        acc = acc * 1.000000119f + v;  // force compiler to keep the load
    }

    // one write per thread (negligible vs reads)
    if (t < N) out[t] = acc;
}
```

When `stride == 1`, threads in the same wavefront access consecutive 4-byte elements (`in[j]`, `in[j+1]`, `in[j+2]`, …). These accesses fall into a small number of aligned DRAM bursts, so the memory system can coalesce them efficiently and deliver high bandwidth.

As stride increases:

- The addresses accessed by neighboring threads move farther apart
- Each wavefront's loads spread across more bursts
- The number of physical transactions per logical access increases
- Effective bandwidth drops

3.9.2 Bank conflict theory

Shared memory (LDS) is organized into banks that can be accessed independently. Bank conflicts occur when multiple threads access different addresses in the same bank.

A bank conflict occurs when multiple threads in a wavefront simultaneously access different addresses that reside in the same LDS bank. When that happens, the accesses to that bank must be serialized, reducing effective LDS throughput by an integer factor and preventing full utilization of the on-chip memory bandwidth.

On AMD GPUs, the on-chip LDS (HIP's "shared memory") inside each compute unit is physically organized into banks. These banks can be accessed in parallel, which is how LDS achieves very high bandwidth. On CDNA and RDNA architectures, LDS is divided into 32 (CDNA, CDNA2 and CDNA3) or 64 (CDNA4+ and RDNA2+) banks, each 4 bytes wide. Conceptually, addresses map to banks like this (low bits only, in bytes):

```
Address: 0x00 0x04 0x08 0x0C 0x10 0x14 0x18 0x1C ... 0x7C
Bank:       0    1    2    3    4    5    6    7 ...   31

Address: 0x80 0x84 0x88 0x8C 0x90 0x94 0x98 0x9C ... 0xFC
Bank:       0    1    2    3    4    5    6    7 ...   31
```

Any two addresses that differ by $32 \times 4 = 128$bytes map to the same bank.

Why bank conflicts matter:

- Conflicts serialize accesses, reducing throughput
- LDS bandwidth drops proportionally to conflict degree
- Can turn parallel operations into sequential ones

Common patterns:

- **No conflict**: Each thread accesses a different bank (full bandwidth)
- **Broadcast**: Multiple threads read the same address (no conflict)
- **N-way conflict**: N threads access the same bank (1/N bandwidth)

3.9.2.1 Example: conflict-free access

If you access sequential elements of a float array in LDS, different lanes in a wavefront naturally land in different banks:

```
__shared__ float data[1024];  // in HIP, __shared__ maps to LDS

int tid = threadIdx.x;
float value = data[tid];  // addresses: 0x00, 0x04, 0x08, ...
```

For 32 consecutive float elements, this maps cleanly: each 4-byte word goes to a different bank (0–31). All these accesses can be serviced in one LDS transaction, so there are no bank conflicts. This is the "good" pattern.

3.9.2.2 Example: pathological strided access (conflict-heavy)

Now consider a pattern that walks down a column of a row-major LDS array where each row has 32 floats:

```
__shared__ float data[32 * 32];  // 32 columns per row

int tid = threadIdx.x;
float value = data[tid * 32];    // addresses: 0x00, 0x80, 0x100, ...
// recall: sizeof(float) == 4 bytes
```

Here, each successive access is offset by $32 \times 4 = 128$bytes —exactly one full bank span. So:

- `data[0]` → bank 0
- `data[32]` → bank 0
- `data[64]` → bank 0

Every lane in the wavefront is hitting bank 0, but at different addresses, all in the same cycle. These accesses must be serialized by the LDS hardware, which can turn a fast LDS access into a slow operation.

3.9.2.3 When conflicts don't happen

If all threads in a wavefront access the exact same address in a bank (e.g., all lanes reading the same control value from LDS), hardware can often broadcast that value. In that case, the request is not treated as a conflict—it's one read, fanned out to many lanes.

3.10 Register pressure theory

Register pressure refers to a situation where the register file becomes a performance bottleneck due to excessive demand for registers by active threads.

Register pressure occurs when a kernel requires more registers than optimal for the target occupancy.

In GPU programming, registers are the fastest level of the memory hierarchy, holding per-thread variables for arithmetic and address computation. However, while the compiler (*amdclang++* for ROCm) works with an unbounded set of virtual registers, the hardware has only a finite number of physical registers per compute unit.

3.10.1 How register pressure arises

Each thread in a wavefront consumes a number of registers as determined by the compiled Instruction Set Architecture (ISA) code (AMD CDNA or RDNA assembly). All threads in a work-group share the same CU, so the total register file usage per work-group depends on both:

- The number of threads per work-group (launch configuration), and
- The number of registers required per thread (kernel complexity)

As the register footprint per work-group increases, fewer work-groups can be resident on a CU at once. This directly reduces occupancy, which in turn limits the ability of the GPU to hide latency through thread-level parallelism. In extreme cases, the compiler may even "spill" register values into global memory—orders of magnitude slower than register access.

Why register pressure matters:

- Reduces maximum occupancy
- May cause register spilling to memory
- Decreases ability to hide latency
- Lowers overall throughput

The relationship between registers and occupancy:

- More registers per thread → fewer concurrent wavefronts
- Fewer registers per thread → higher occupancy but may need memory spills
- Optimal balance depends on kernel memory access patterns

3.11 Performance metrics explained

Understanding performance metrics is essential for analyzing GPU behavior:

3.11.1 Peak rate

Peak rate is the theoretical maximum throughput a hardware system can achieve. It represents the absolute upper bound of GPU performance—the architecture's effective 'speed of light.'

Peak rate assumes ideal conditions: every compute unit is fully active, all execution pipelines are perfectly fed, and no constraints (e.g., register pressure, memory stalls, synchronization, or bandwidth limits) impede progress.

The theoretical maximum performance of a GPU:

- **Peak FLOPS**: Maximum floating-point operations per second
- **Peak bandwidth**: Maximum memory throughput
- **Peak instruction rate**: Maximum instructions per cycle

In performance analysis, peak rate serves several roles:

- It defines the compute-bound "roof" in a roofline model
- It forms the denominator for utilization metrics such as pipe utilization or CU utilization
- It provides the theoretical yardstick against which achieved performance (measured FLOPs per second) is compared

Actual performance is always below peak due to various inefficiencies.

3.11.2 Utilization metrics

3.11.2.1 Pipe utilization

Pipe utilization measures how effectively a kernel uses the execution pipelines within each compute unit.

Each CU contains multiple independent execution pipes, each specialized for a different class of operations—for example:

- VALU pipes handle general vector arithmetic (add, multiply, Fused Multiply-Add (FMA))
- MFMA pipes handle matrix operations on matrix-fused multiply–accumulate units
- SALU pipes handle scalar operations
- Load and Store (LDS and VMEM) units handle memory access instructions
- Branch and Control units handle program flow

Pipe utilization quantifies the percentage of each pipeline's peak theoretical throughput that is being achieved, averaged over all active CUs and cycles in which the pipeline is active.

Pipe utilization: The percentage of execution cycles where the pipeline is actively processing instructions. Low utilization indicates stalls or insufficient work.

Before analyzing performance at the level of pipe utilization, you should first examine kernel utilization (how often CUs are busy) and CU utilization (how evenly work is distributed across CUs). Once those are sufficient, per-pipe metrics reveal whether the performance limit is arithmetic, memory, or control-bound.

On AMD GPUs, these measurements are exposed through ROCm profiling tools such as `rocprofv3`.

Relevant counters include:

- `SQ_ACCUM_PREV_HI_BUSY` (VALU pipeline busy percentage)
- `SQ_ACCUM_MFMA_BUSY` (MFMA utilization)
- `SQ_ACCUM_LDS_BUSY` and `SQ_ACCUM_VMEM_BUSY` (memory pipelines)
- `SQ_ACCUM_SALU_BUSY` (scalar ALU activity)

Together, these form the pipe utilization profile—showing how well each instruction pipeline is being fed with eligible wavefronts and how close the kernel is to saturating the hardware's arithmetic or memory throughput.

3.11.2.2 Issue efficiency

Issue efficiency measures how effectively the wavefront scheduler on each compute unit keeps the execution pipelines busy by issuing instructions from eligible wavefronts. In a perfectly efficient kernel, the scheduler issues one instruction every cycle for every active CU.

An issue efficiency of 100% means that on every active cycle, at least one wavefront was eligible and an instruction was successfully issued. Lower values indicate that during some cycles, all active wavefronts were stalled—waiting on memory, dependencies, or resources—and the scheduler was idle, reducing total instruction throughput.

Issue efficiency: The ratio of issued instructions to the maximum possible. Low efficiency can indicate instruction cache misses, scheduling inefficiencies, or resource conflicts.

On AMD GPUs, issue efficiency can be measured using hardware performance counters exposed through ROCProfiler or Omnitrace, such as:

- `SQ_WAVES_BUSY` — percentage of cycles where any wavefront was actively executing
- `SQ_WAVES_ISSUED` — number of issued waves per cycle
- `SQ_ACCUM_INSTS_ISSUED` — total instructions issued per CU
- `SQ_ACCUM_CYCLES_BUSY` — number of cycles the CU was active

By combining these metrics, you can estimate how efficiently the scheduler keeps the CU's pipelines fed. Low issue efficiency typically signals insufficient concurrency (low occupancy) or high memory latency, both of which prevent the hardware from issuing instructions continuously.

3.11.2.3 CU utilization

CU utilization measures the percentage of time that compute units on an AMD GPU are actively executing instructions.

Instead of reporting the fraction of time a kernel is executing somewhere on the GPU, CU utilization reports the fraction of time all CUs spend executing wavefronts.

CU utilization: The percentage of compute units actively executing work. Low utilization suggests insufficient parallelism, load imbalance, or synchronization overhead.

As with GPU utilization, high CU utilization is generally desirable. It indicates that most CUs are busy executing instructions across the device. However, high CU utilization alone does not guarantee full performance.

If CU utilization is high but throughput remains low, the kernel may not be effectively using the functional pipelines within each CU—such as vector ALUs, MFMA tensor cores, or load and store units. In that case, you should examine pipe utilization, which measures how fully those individual execution paths are being used.

CU utilization can be observed with AMD's profiling and monitoring tools:

- `amd-smi metric --usage` — reports overall GPU activity percentage

- `rocprofv3` — provides performance counters like `SQ_WAVE_CYCLES`, `SQ_BUSY_CYCLES`, and per-pipe instruction metrics
- `rocminfo` — shows the number of CUs available per GPU

In summary, CU utilization captures how actively the GPU's compute units are engaged in running wavefronts. High CU utilization indicates good parallel workload distribution, while low CU utilization may point to poor occupancy, launch configuration limits, or insufficient concurrency.

3.11.2.4 Branch efficiency

Branch efficiency measures how often all threads within a wavefront take the same execution path when encountering conditional statements.

It quantifies control-flow uniformity—that is, how often all lanes in a wavefront evaluate a conditional identically. It is calculated as the ratio of uniform branch decisions to total branch instructions executed. High branch efficiency indicates little to no wavefront divergence, while low branch efficiency means many lanes are masked off due to diverging control flow.

Branch efficiency: The ratio of non-divergent to total branches. Low efficiency indicates significant divergence overhead.

Not all conditionals reduce branch efficiency. The common "bounds check" pattern found in most GPU kernels, for instance:

```
int idx = blockIdx.x * blockDim.x + threadIdx.x;
if (idx < n)
```

usually has very high branch efficiency, since nearly all wavefronts consist entirely of threads that either satisfy `idx < n` or not—except perhaps for the last partial wavefront, which straddles the boundary of `n`.

While CPUs also optimize branch behavior, they focus on temporal uniformity—predicting whether the same branch will be taken or not over repeated iterations. GPUs, on the other hand, care about spatial uniformity: whether all lanes in the wavefront take the same branch at the same time.

On AMD architectures, this spatial uniformity is tracked via the EXEC mask. Divergence forces EXEC to toggle individual bits to deactivate lanes following a different control path. High branch efficiency implies minimal EXEC manipulation, meaning nearly all lanes execute the same instruction stream simultaneously—maximizing SIMD efficiency and overall throughput.

3.12 Theoretical performance limits

Understanding theoretical limits helps set realistic performance expectations.

3.12.1 Peak performance bounds

Every GPU has theoretical maximum performance determined by:

- Clock frequency and number of compute units
- Instruction throughput per clock cycle
- Memory bandwidth capacity
- Specialized unit capabilities (matrix cores, SFUs)

3.12.2 Achievable performance

Real applications typically achieve a fraction of theoretical peak due to:

- Imperfect resource utilization
- Memory access inefficiencies
- Control flow divergence
- Synchronization overhead
- Launch and scheduling costs

The gap between theoretical and achieved performance reveals optimization opportunities. The roofline model provides a framework for understanding these limits and identifying which factor (compute or memory) constrains performance.

3.13 Summary

Understanding GPU performance requires knowledge of several interconnected concepts:

- **Performance bottlenecks**: Whether compute, memory, or overhead limits performance
- **Roofline model**: Visual framework for analyzing performance limits based on arithmetic intensity
- **Arithmetic intensity**: The compute-to-memory ratio of algorithms
- **Latency hiding**: How concurrent execution masks delays through wavefront switching
- **Occupancy**: How wavefront concurrency affects resource utilization
- **Memory hierarchy**: How different memory types affect bandwidth and the importance of coalescing
- **Performance metrics**: Quantitative measures for analysis including pipe utilization, issue efficiency, CU utilization, and branch efficiency

These theoretical foundations inform practical optimization decisions. For step-by-step optimization techniques and practical guidance, see *Performance guidelines*.

CHAPTER

FOUR

PERFORMANCE GUIDELINES

The AMD HIP performance guidelines provide practical, actionable techniques for optimizing application performance on AMD GPUs. This guide focuses on step-by-step instructions and best practices for improving performance.

For theoretical foundations and performance concepts, see *Understanding GPU performance*.

4.1 Optimization workflow

Follow this systematic approach to optimize GPU performance:

1. **Profile and measure baseline**

   Use `rocprofv3` to identify bottlenecks:

   ```
   rocprofv3 --stats --<tracing_option> -- <application_path>
   ```

   Collect metrics on kernel execution time, memory bandwidth, occupancy, and CU utilization. For more details on using `rocprofv3` for application tracing and profiling, see rocprofv3 documentation.

2. **Analyze metrics to identify bottlenecks**

   Determine if kernels are compute-bound or memory-bound. Check arithmetic intensity, memory bandwidth achieved vs peak, and compute throughput.

   For understanding the roofline model, see *Roofline model*.

3. **Apply targeted optimizations**

   Based on identified bottlenecks, apply techniques from this guide.

4. **Verify improvements**

   Re-profile to confirm performance gains.

5. **Iterate**

   Repeat until performance goals are met.

4.2 Profiling and analysis tools

ROCm provides a comprehensive suite of profiling and analysis tools that help developers understand and optimize GPU performance. These tools are essential for identifying bottlenecks and evaluating the effectiveness of performance optimizations.

4.2.1 rocprofv3

The tool rocprofv3 provides command-line-driven profiling for detailed performance analysis. It collects metrics on kernel execution time, memory bandwidth, wavefront occupancy, VALU utilization, and instruction-level counters.

`rocprofv3` integrates with the rocProfiler-SDK framework to collect hardware traces and API-level timing data. The collected data can be exported in JSON and CSV formats for further analysis or visualization.

Key capabilities:

- Kernel execution profiling
- Memory bandwidth analysis
- Wavefront occupancy metrics
- Compute unit utilization
- Instruction-level performance counters
- API call tracing
- Hardware event collection

4.2.1.1 Trace visualization with Perfetto

For application-level tracing, `rocprofv3` can generate traces compatible with Perfetto, a third-party, open-source trace viewer. This enables visualization of the complete timeline of application execution, including the temporal relationships among host operations, kernel launches, memory transfers, and synchronization events.

Using Perfetto with traces generated by `rocprofv3` can help identify performance issues caused by API overhead, inefficient synchronization, or insufficient overlap between computation and data movement.

The following trace data is available for visualization:

- Application execution timelines
- HIP API call tracking and timing
- ROCm library call analysis
- Host-device synchronization events
- Memory transfer operations

4.2.2 ROCprof Compute Viewer

ROCprof Compute Viewer provides a GUI-based environment for analyzing GPU kernel performance data. It delivers detailed kernel-level insights, including counter correlation and hierarchical performance breakdowns, helping developers interpret execution patterns and identify optimization opportunities.

Key features:

- Kernel performance counter analysis
- Counter correlation and visualization
- Hierarchical kernel performance breakdown
- Interactive performance data exploration
- Kernel-level optimization insights

4.2.3 AMD System Management Interface

The AMD System Management Interface (AMD SMI) is a command-line utility for querying, monitoring, and managing AMD GPUs. It provides system administrators and developers with detailed, real-time information about GPU hardware, utilization, and power metrics.

`amd-smi` reports the following categories of information:

- GPU identity information, such as the card name, device ID, and PCI bus location
- Live utilization metrics, including GPU activity, memory usage, clock speeds, and active processes
- Power and thermal readings, such as temperature, fan speed, voltage, and power draw
- Performance states and limits, such as available frequency levels, clock throttling, and voltage controls

These metrics are retrieved through the AMD SMI C API, which exposes a stable, scriptable interface for system and performance monitoring tools.

`amd-smi` also supports management operations, including:

- Setting power caps and performance profiles
- Adjusting clock frequencies
- Performing GPU resets and controlling persistence mode
- Reporting or controlling ECC status on supported data center GPUs

Output can be formatted as human-readable text or JSON (`--json`), commonly used for integration into automated monitoring pipelines.

Basic usage examples:

```
# Display GPU information
amd-smi static

# Monitor GPU utilization and metrics
amd-smi metric --usage

# Show detailed information in JSON format
amd-smi static --json
```

4.2.4 Typical workflow

For comprehensive ROCm performance analysis:

1. Use `rocprofv3` to collect profiling data and generate traces
2. Use ROCprof Compute Viewer for detailed kernel performance analysis and counter correlation
3. Use third-party tools like Perfetto to visualize application traces, API calls, and timeline behavior
4. Use `amd-smi` to monitor GPU utilization and system health
5. Iterate between profiling and optimization, verifying improvements with each change

This multi-tool approach provides kernel-level metrics, application-level tracing, system monitoring, and visual context to understand overall performance behavior.

4.3 Parallel execution

For optimal use and to keep all system components busy, the application must reveal and efficiently provide as much parallelism as possible.

4.3.1 Application level

To enable parallel execution across the host and devices:

- Use asynchronous calls and streams
- Assign serial workloads to the host
- Assign parallel workloads to the devices

For parallel workloads:

- Use `__syncthreads()` (see Synchronization functions) for intra-block synchronization
- Use global memory with separate kernel invocations for inter-block synchronization (has overhead, minimize when possible)

4.3.2 Device level

Maximize parallel execution across multiprocessors:

- Execute multiple kernels concurrently on a device
- Use streams to overlap computation and data transfers
- Keep all multiprocessors busy with enough concurrent kernels
- Avoid launching too many kernels (causes resource contention)

4.3.3 Multiprocessor level

Maximize parallel execution within each compute unit:

- Ensure sufficient resident wavefronts for every clock cycle
- Exploit instruction-level parallelism within wavefronts
- Exploit thread-level parallelism across wavefronts
- Balance resource usage for optimal *occupancy*

4.4 Memory throughput optimization

The first step in maximizing memory throughput is to minimize low-bandwidth data transfers between the host and the device.

Additionally, maximize the use of on-chip memory (shared memory and caches) and minimize transfers with global memory.

4.4.1 Data transfer optimization

Minimize host-device transfers

- Move computations from host to device when possible
- Create, use, and discard intermediate data structures on device

- Avoid unnecessary copies to host memory

Batch small transfers

Each memory transfer incurs a fixed overhead from driver calls and PCIe transaction setup. Consolidating many small transfers into a single large transfer amortizes this overhead across more data, resulting in much higher effective bandwidth.

```
// Instead of many small transfers
for (int i = 0; i < n; i++) {
    hipMemcpy(&d_data[i], &h_data[i], sizeof(float), ...);
}

// Use a single large transfer
hipMemcpy(d_data, h_data, n * sizeof(float), ...);
```

Use page-locked memory for transfers

Page-locked (pinned) memory cannot be swapped to disk by the operating system, allowing the GPU to access it directly via DMA without CPU involvement. This eliminates an extra copy through a staging buffer and achieves higher bandwidth.

```
float* h_pinned;
hipHostMalloc(&h_pinned, size);
// Faster transfers than pageable memory
hipMemcpy(d_data, h_pinned, size, hipMemcpyHostToDevice);
```

Use mapped memory on integrated systems

On integrated GPUs (APUs), the CPU and GPU share the same physical memory. Mapped page-locked memory allows zero-copy access, where the GPU reads directly from host memory without requiring an explicit transfer, eliminating redundant copies.

```
int integrated;
hipDeviceGetAttribute(&integrated, hipDeviceAttributeIntegrated, device);
if (integrated) {
    // Use mapped page-locked memory - no explicit copy needed
    hipHostMalloc(&ptr, size, hipHostMallocMapped);
}
```

4.4.2 Device memory access

Ensure proper alignment

Memory hardware loads data in aligned chunks (typically 128 bytes). Using naturally aligned data types ensures each access maps to a single memory transaction, maximizing bandwidth and avoiding split transactions.

```
// Use naturally aligned types
float4 data;  // 16-byte aligned
float2 data;  // 8-byte aligned

// Ensure structure alignment
struct __align__(16) MyStruct {
    float4 data;
};
```

Optimize 2D array access

Padding 2D arrays to multiples of the wavefront size ensures each row starts at an aligned memory boundary. This allows consecutive threads accessing the same row to generate coalesced memory transactions, thereby maximizing bandwidth.

```
// Ensure array width is multiple of wavefront size
int width = ((actual_width + warpSize - 1) / warpSize) * warpSize;
hipMalloc(&array, width * height * sizeof(float));

// Access pattern
int idx = x + width * y;  // width should be wavefront-aligned
```

Coalesce memory accesses

When consecutive threads in a wavefront access consecutive memory addresses, the hardware combines these into a single wide transaction. Non-coalesced patterns require multiple transactions, reducing effective bandwidth.

```
// Good: consecutive threads access consecutive addresses
int idx = threadIdx.x + blockIdx.x * blockDim.x;
data[idx] = value;

// Bad: strided access
int idx = threadIdx.x * stride;  // Non-coalesced if stride > 1
data[idx] = value;
```

For understanding memory coalescing theory, see *Memory hierarchy impact on performance*.

Use shared memory for data reuse

Shared memory (LDS) provides fast on-CU scratchpad memory for communication between threads in a block. Loading data into shared memory once and reusing it many times reduces global memory traffic, particularly effective for tiled algorithms such as matrix multiplication.

```
__global__ void optimized_kernel(float* input, float* output) {
    __shared__ float tile[TILE_SIZE][TILE_SIZE];

    // Load data into shared memory
    tile[threadIdx.y][threadIdx.x] = input[...];
    __syncthreads();

    // Reuse data from fast shared memory
    float result = 0;
    for (int i = 0; i < TILE_SIZE; i++) {
        result += tile[threadIdx.y][i] * tile[i][threadIdx.x];
    }
    __syncthreads();

    output[...] = result;
}
```

Avoid bank conflicts in shared memory

Shared memory is organized into banks, each capable of servicing one request per cycle. When multiple threads in a wavefront access the same bank simultaneously, the requests are serialized, reducing throughput. Padding arrays by one element shifts addresses to avoid systematic conflicts.

```
// Bad: power-of-2 stride causes conflicts
__shared__ float data[32][32];
float value = data[threadIdx.x][threadIdx.y];
```

(continues on next page)

(continued from previous page)

```
// Good: padding avoids conflicts
__shared__ float data[32][33];  // Extra column
float value = data[threadIdx.x][threadIdx.y];
```

For bank conflict theory, see *Bank conflict theory*.

Use texture memory for 2D spatial access

Texture memory provides hardware-accelerated 2D filtering and caching optimized for spatial locality. It automatically handles boundary conditions and can interpolate values, making it ideal for image processing and nearby-neighbor access patterns.

```
// Create texture object
hipTextureObject_t texObj;
hipCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);

// Access in kernel
float value = tex2D<float>(texObj, x, y);
```

4.5 Instruction throughput optimization

4.5.1 Arithmetic instructions

Use efficient operations

Division requires many more hardware cycles than multiplication. Similarly, bitwise operations (shifts, AND, OR) are single-cycle instructions on integer units, making them far more efficient than equivalent arithmetic for power-of-two calculations.

```
// Prefer multiplication over division
float result = value * 0.5f;     // Fast
float result = value / 2.0f;     // Slower

// Use bitwise operations for powers of 2
int index = threadIdx.x << 2;    // Multiply by 4
int mask = (1 << n) - 1;         // Create bit mask
```

Use single-precision when possible

AMD GPUs have significantly higher throughput for single-precision (FP32) operations compared to double-precision (FP64). Using single-precision math functions can deliver substantial performance gains when FP64 accuracy is not required.

```
// Single-precision (faster)
float result = sinf(x);
float result = expf(x);

// Double-precision (slower, use only when necessary)
double result = sin(x);
double result = exp(x);
```

Leverage fast math intrinsics

Hardware-specific intrinsics bypass certain accuracy checks and use lookup tables or polynomial approximations, trading slight precision loss for significantly higher throughput. These should be used when the application can tolerate reduced

precision.

```
// Fast intrinsic versions
float ex = __expf(x);             // Fast exponential
float lg = __logf(x);             // Fast logarithm
float sq = __fsqrt_rn(x);         // Fast square root
float rc = __frcp_rn(x);          // Fast reciprocal
```

4.5.2 Control flow optimization

Minimize divergence

When threads in a wavefront take different execution paths, the hardware serializes both branches, executing each path with only the relevant threads active. This reduces effective parallelism and wastes cycles on inactive threads.

```
// Good: no divergence (condition depends on threadIdx)
if (threadIdx.x < 32) {
    // All threads in first half-wavefront execute
}

// Bad: divergence within wavefront
if (data[threadIdx.x] > threshold) {
    // Some threads execute, others don't
}
```

Use branch hints for predictable conditions

Providing hints about branch likelihood helps the compiler generate better instruction ordering and can improve the branch predictor's accuracy, reducing pipeline stalls when the prediction proves correct.

```
if (__builtin_expect(rare_condition, 0)) {
    // Unlikely branch
}

// C++20 attribute
if (common_condition) [[likely]] {
    // Likely branch
}
```

Avoid divergent wavefronts

When divergence is unavoidable, restructure the code to separate divergent paths into different kernel launches or use predication (branchless programming) to keep all threads active, though computing unnecessary values may be acceptable if it avoids the serialization penalty.

```
// Instead of:
if (threadIdx.x % 2 == 0) {
    result = compute_even();
} else {
    result = compute_odd();
}

// Consider separating into different kernels or using predication
```

4.5.3 Synchronization

Use minimal synchronization

Each synchronization point stalls all threads in a block until the slowest one reaches the barrier. Minimize synchronizations by carefully analyzing data dependencies—only synchronize when threads genuinely need to exchange data through shared memory.

```
__global__ void kernel() {
    __shared__ float data[256];

    // Load phase
    data[threadIdx.x] = input[...];
    __syncthreads();  // Necessary sync

    // Compute phase - no sync needed if threads are independent
    float result = compute(data[...]);

    // Store phase - sync only if needed
    output[...] = result;
}
```

Use streams for async execution

Streams enable concurrent execution of independent operations. Commands in different streams can overlap in time, allowing kernel execution and memory transfers to run simultaneously. This maximizes GPU utilization by keeping multiple execution engines busy concurrently.

```
hipStream_t stream1, stream2;
hipStreamCreate(&stream1);
hipStreamCreate(&stream2);

// Overlap independent operations
kernel1<<<grid, block, 0, stream1>>>(...);
kernel2<<<grid, block, 0, stream2>>>(...);

hipStreamSynchronize(stream1);
hipStreamSynchronize(stream2);
```

4.6 Managing register pressure

High register usage can limit *occupancy*. Follow these steps:

Minimize live variables

The compiler allocates registers for every variable that must remain accessible. Reducing the number of simultaneously live variables frees registers, allowing more wavefronts to fit on each CU. Chaining function calls trades some redundant computation for lower register usage.

```
// Instead of storing all intermediate results
float a = compute_a();
float b = compute_b();
float c = compute_c();
float result = combine(a, b, c);
```

(continues on next page)

(continued from previous page)

```
// Recompute or chain operations
float result = combine(compute_a(), compute_b(), compute_c());
```

Use shared memory for temporary storage

Per-thread arrays stored in registers consume valuable register space, limiting *occupancy*. Moving temporary storage to shared memory trades register usage for shared memory usage, often allowing higher occupancy since shared memory limits are typically less restrictive.

```
// Instead of per-thread arrays (uses registers)
float temp[100];

// Use shared memory
__shared__ float temp[blockDim.x][100];
float* my_temp = temp[threadIdx.x];
```

Adjust launch bounds

The `__launch_bounds__` attribute provides hints to the compiler about expected thread block size and minimum blocks per CU. This guides register allocation decisions, potentially trading per-thread register count for higher occupancy.

```
__global__ void
__launch_bounds__(256, 4)  // 256 threads, 4 blocks per CU
my_kernel() {
    // Kernel code
}
```

Check register usage during compilation

The compiler can report per-kernel register usage statistics. Monitoring this output helps identify kernels consuming excessive registers, guiding optimization efforts toward reducing register pressure in the most impactful areas.

```
hipcc --resource-usage kernel.hip
```

For register pressure theory, see *Register pressure theory*.

4.7 Improving occupancy

Higher *occupancy* helps hide latency. Follow these steps:

Reduce register usage per thread

Use techniques from "Managing register pressure" above.

Reduce shared memory usage per block

Each CU has limited shared memory that must be divided among resident blocks. Reducing per-block shared memory usage allows more blocks to reside simultaneously, increasing *occupancy* and improving latency hiding through greater thread-level parallelism.

```
// Allocate only what's needed
__shared__ float tile[TILE_SIZE][TILE_SIZE];

// Or use dynamic allocation
extern __shared__ float dynamic_shared[];
```

Optimize block size

AMD Instinct GPUs execute threads in wavefronts of 64, while AMD Radeon GPUs execute threads in wavefronts of 32. Choosing block sizes as multiples of 64 or 32 prevents partial wavefronts that waste execution slots. Larger blocks (128-256 threads) typically achieve better *occupancy* and resource utilization.

```
// Use multiples of wavefront size
dim3 block(64);    // Good for AMD Instinct GPUs (wavefront=64)
dim3 block(128);   // Common choice
dim3 block(256);   // Good for high-occupancy kernels

// Avoid very small blocks
dim3 block(32);    // May waste resources on Instinct GPUs
```

Profile occupancy

Profiling tools report the ratio of active wavefronts to maximum possible wavefronts per CU. Low *occupancy* suggests resource constraints (registers or shared memory) are limiting parallelism and may indicate opportunities for optimization.

```
rocprofv3 --occupancy ./your_application
```

For occupancy theory, see *Occupancy theory*.

4.8 Minimizing memory thrashing

Applications frequently allocating and freeing memory might experience slower allocation calls over time. To optimize:

Allocate early, deallocate late

Frequent allocation and deallocation causes memory fragmentation and increases allocator overhead. Reusing allocations across iterations amortizes the cost of memory management and maintains better memory locality.

```
// Bad: frequent allocation in loop
for (int i = 0; i < iterations; i++) {
    float* temp;
    hipMalloc(&temp, size);
    // Use temp
    hipFree(temp);
}

// Good: allocate once
float* temp;
hipMalloc(&temp, size);
for (int i = 0; i < iterations; i++) {
    // Reuse temp
}
hipFree(temp);
```

Avoid allocating all available memory

Reserving some memory headroom prevents allocation failures and system instability. The driver and runtime need workspace for internal operations, and leaving a safety margin ensures stable operation without unexpected out-of-memory errors.

```
std::size_t free, total;
hipMemGetInfo(&free, &total);
```

(continues on next page)

(continued from previous page)

```
// Don't allocate all free memory
std::size_t safe_size = free * 0.9;  // Leave some margin
```

Use managed memory for oversubscription

Managed memory automatically migrates data between host and device on demand, allowing allocations larger than physical GPU memory. Prefetching hints help the runtime optimize page placement, reducing migration overhead during kernel execution.

```
// Allows exceeding physical memory
float* data;
hipMallocManaged(&data, large_size);

// Optionally prefetch to device
hipMemPrefetchAsync(data, size, device, stream);
```

4.9 Summary

Key optimization techniques:

- **Profile first**: Use `rocprofv3` to identify actual bottlenecks
- **Parallelize effectively**: Maximize work at all levels (application, device, CU)
- **Optimize memory**: Minimize transfers, maximize coalescing, use LDS
- **Manage resources**: Balance registers, shared memory, and occupancy
- **Minimize divergence**: Structure control flow to keep wavefronts coherent

For understanding the theory behind these techniques, refer to *Understanding GPU performance* and Hardware implementation.

CHAPTER

FIVE

MULTI-GPU PROGRAMMING

Multi-GPU programming uses the available GPUs within a single system or across nodes to accelerate compute workloads. It covers techniques for multi-device management, including device selection, synchronization, and memory handling. These topics help developers scale HIP applications to fully use the computational power of multi-GPU and distributed environments.

Note

The ROCm Communication Collectives Library (RCCL) is a stand-alone library that also provides multi-GPU and multi-node collective communication primitives optimized for AMD GPUs. It uses PCIe and xGMI high-speed interconnects. For more information, see RCCL documentation.

5.1 Device enumeration

Device enumeration involves identifying all the available GPUs connected to the host system. A single host machine can have multiple GPUs, each with its own unique identifier. By listing these devices, you can decide which GPU to use for computation. The host queries the system to count and list all connected GPUs that support the chosen `HIP_PLATFORM`, ensuring that the application can leverage the full computational power available. Typically, applications list devices and their properties for deployment planning, and also make dynamic selections during runtime to ensure optimal performance.

If the application does not define a specific GPU, device 0 is selected.

```
#include <hip/hip_runtime.h>

#include <cstdlib>
#include <iostream>

#define HIP_CHECK(expression)                          \
{                                                      \
    const hipError_t status = expression;              \
    if (status != hipSuccess)                          \
    {                                                  \
        std::cerr << "HIP error " << status            \
                << ": " << hipGetErrorString(status)  \
                << " at " << __FILE__ << ":"           \
                << __LINE__ << std::endl;              \
        std::exit(EXIT_FAILURE);                       \
    }                                                  \
}
```

(continues on next page)

(continued from previous page)

```
int main()
{
    int deviceCount;
    HIP_CHECK(hipGetDeviceCount(&deviceCount));
    std::cout << "Number of devices: " << deviceCount << std::endl;

    for (int deviceId = 0; deviceId < deviceCount; ++deviceId)
    {
        hipDeviceProp_t deviceProp;
        HIP_CHECK(hipGetDeviceProperties(&deviceProp, deviceId));
        std::cout << "Device " << deviceId << std::endl << " Properties:" <<␣
↪std::endl;
        std::cout << "  Name: " << deviceProp.name << std::endl;
        std::cout << "  Total Global Memory: " << deviceProp.totalGlobalMem / (1024 *␣
↪1024) << " MiB" << std::endl;
        std::cout << "  Shared Memory per Block: " << deviceProp.sharedMemPerBlock /␣
↪1024 << " KiB" << std::endl;
        std::cout << "  Registers per Block: " << deviceProp.regsPerBlock <<␣
↪std::endl;
        std::cout << "  Warp Size: " << deviceProp.warpSize << std::endl;
        std::cout << "  Max Threads per Block: " << deviceProp.maxThreadsPerBlock <<␣
↪std::endl;
        std::cout << "  Max Threads per Multiprocessor: " << deviceProp.
↪maxThreadsPerMultiProcessor << std::endl;
        std::cout << "  Number of Multiprocessors: " << deviceProp.
↪multiProcessorCount << std::endl;
        std::cout << "  Max Threads Dimensions: ["
                << deviceProp.maxThreadsDim[0] << ", "
                << deviceProp.maxThreadsDim[1] << ", "
                << deviceProp.maxThreadsDim[2] << "]" << std::endl;
        std::cout << "  Max Grid Size: ["
                << deviceProp.maxGridSize[0] << ", "
                << deviceProp.maxGridSize[1] << ", "
                << deviceProp.maxGridSize[2] << "]" << std::endl;
        std::cout << std::endl;
    }

    return EXIT_SUCCESS;
}
```

5.2 Device selection

Once you have enumerated the available GPUs, the next step is to select a specific device for computation. This involves setting the active GPU that will execute subsequent operations. This step is crucial in multi-GPU systems where different GPUs might have different capabilities or workloads. By selecting the appropriate device, you ensure that the computational tasks are directed to the correct GPU, optimizing performance and resource utilization.

```
#include <hip/hip_runtime.h>

#include <cstddef>
#include <cstdlib>
```

(continues on next page)

(continued from previous page)

```
#include <iostream>

#define HIP_CHECK(expression)                          \
{                                                      \
    const hipError_t status = expression;              \
    if (status != hipSuccess)                          \
    {                                                  \
        std::cerr << "HIP error " << status            \
                << ": " << hipGetErrorString(status)  \
                << " at " << __FILE__ << ":"           \
                << __LINE__ << std::endl;              \
        std::exit(EXIT_FAILURE);                       \
    }                                                  \
}

__global__ void simpleKernel(double *data, std::size_t elems)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < elems)
        data[idx] = idx * 2.0;
}

int main()
{
    int deviceCount;
    HIP_CHECK(hipGetDeviceCount(&deviceCount));
    if(deviceCount < 2)
    {
        std::cout << "This example requires at least two HIP devices." << std::endl;
        return EXIT_SUCCESS;
    }

    double* deviceData0;
    double* deviceData1;
    constexpr std::size_t elems = 1024;
    constexpr std::size_t size = elems * sizeof(double);

    int deviceId0 = 0;
    int deviceId1 = 1;

    // Set device 0 and perform operations
    HIP_CHECK(hipSetDevice(deviceId0)); // Set device 0 as current
    HIP_CHECK(hipMalloc(&deviceData0, size)); // Allocate memory on device 0
    simpleKernel<<<8, 128>>>(deviceData0, elems); // Launch kernel on device 0
    HIP_CHECK(hipDeviceSynchronize());

    // Set device 1 and perform operations
    HIP_CHECK(hipSetDevice(deviceId1)); // Set device 1 as current
    HIP_CHECK(hipMalloc(&deviceData1, size)); // Allocate memory on device 1
    simpleKernel<<<8, 128>>>(deviceData1, elems); // Launch kernel on device 1
    HIP_CHECK(hipDeviceSynchronize());
```

(continues on next page)

(continued from previous page)

```
    // Copy result from device 0
    double hostData0[elems];
    HIP_CHECK(hipSetDevice(deviceId0));
    HIP_CHECK(hipMemcpy(hostData0, deviceData0, size, hipMemcpyDeviceToHost));

    // Copy result from device 1
    double hostData1[elems];
    HIP_CHECK(hipSetDevice(deviceId1));
    HIP_CHECK(hipMemcpy(hostData1, deviceData1, size, hipMemcpyDeviceToHost));

    // Display results from both devices
    std::cout << "Device 0 data: " << hostData0[0] << std::endl;
    std::cout << "Device 1 data: " << hostData1[0] << std::endl;

    // Free device memory
    HIP_CHECK(hipFree(deviceData0));
    HIP_CHECK(hipFree(deviceData1));

    return EXIT_SUCCESS;
}
```

5.3 Stream and event behavior

In a multi-device system, streams and events are essential for efficient parallel computation and synchronization. Streams enable asynchronous task execution, allowing multiple devices to process data concurrently without blocking one another. Events provide a mechanism for synchronizing operations across streams and devices, ensuring that tasks on one device are completed before dependent tasks on another device begin. This coordination prevents race conditions and optimizes data flow in multi-GPU systems. Together, streams and events maximize performance by enabling parallel execution, load balancing, and effective resource utilization across heterogeneous hardware.

```
#include <hip/hip_runtime.h>

#include <cstddef>
#include <cstdlib>
#include <iostream>

#define HIP_CHECK(expression)                          \
{                                                      \
    const hipError_t status = expression;              \
    if (status != hipSuccess)                          \
    {                                                  \
        std::cerr << "HIP error " << status            \
                << ": " << hipGetErrorString(status)   \
                << " at " << __FILE__ << ":"           \
                << __LINE__ << std::endl;              \
        std::exit(EXIT_FAILURE);                       \
    }                                                  \
}

__global__ void simpleKernel(double *data, std::size_t elems)
{
```

(continues on next page)

(continued from previous page)

```
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < elems)
        data[idx] = idx * 2.0;
}

int main()
{
    int numDevices;
    HIP_CHECK(hipGetDeviceCount(&numDevices));

    if (numDevices < 2)
    {
        std::cout << "This example requires at least two HIP devices." << std::endl;
        return EXIT_SUCCESS;
    }

    double *deviceData0;
    double *deviceData1;
    constexpr std::size_t elems = 1024;
    constexpr std::size_t size = elems * sizeof(double);

    // Create streams and events for each device
    hipStream_t stream0, stream1;
    hipEvent_t startEvent0, stopEvent0, startEvent1, stopEvent1;

    // Initialize device 0
    HIP_CHECK(hipSetDevice(0));
    HIP_CHECK(hipStreamCreate(&stream0));
    HIP_CHECK(hipEventCreate(&startEvent0));
    HIP_CHECK(hipEventCreate(&stopEvent0));
    HIP_CHECK(hipMalloc(&deviceData0, size));

    // Initialize device 1
    HIP_CHECK(hipSetDevice(1));
    HIP_CHECK(hipStreamCreate(&stream1));
    HIP_CHECK(hipEventCreate(&startEvent1));
    HIP_CHECK(hipEventCreate(&stopEvent1));
    HIP_CHECK(hipMalloc(&deviceData1, size));

    // Record the start event on device 0
    HIP_CHECK(hipSetDevice(0));
    HIP_CHECK(hipEventRecord(startEvent0, stream0));

    // Launch the kernel asynchronously on device 0
    simpleKernel<<<8, 128, 0, stream0>>>(deviceData0, elems);

    // Record the stop event on device 0
    HIP_CHECK(hipEventRecord(stopEvent0, stream0));

    // Wait for the stop event on device 0 to complete
    HIP_CHECK(hipEventSynchronize(stopEvent0));
```

(continues on next page)

(continued from previous page)

```
    // Record the start event on device 1
    HIP_CHECK(hipSetDevice(1));
    HIP_CHECK(hipEventRecord(startEvent1, stream1));

    // Launch the kernel asynchronously on device 1
    simpleKernel<<<8, 128, 0, stream1>>>(deviceData1, elems);

    // Record the stop event on device 1
    HIP_CHECK(hipEventRecord(stopEvent1, stream1));

    // Wait for the stop event on device 1 to complete
    HIP_CHECK(hipEventSynchronize(stopEvent1));

    // Calculate elapsed time between the events for both devices
    float milliseconds0 = 0, milliseconds1 = 0;
    HIP_CHECK(hipEventElapsedTime(&milliseconds0, startEvent0, stopEvent0));
    HIP_CHECK(hipEventElapsedTime(&milliseconds1, startEvent1, stopEvent1));

    std::cout << "Elapsed time on GPU 0: " << milliseconds0 << " ms" << std::endl;
    std::cout << "Elapsed time on GPU 1: " << milliseconds1 << " ms" << std::endl;

    // Cleanup for device 0
    HIP_CHECK(hipSetDevice(0));
    HIP_CHECK(hipEventDestroy(startEvent0));
    HIP_CHECK(hipEventDestroy(stopEvent0));
    HIP_CHECK(hipStreamSynchronize(stream0));
    HIP_CHECK(hipStreamDestroy(stream0));
    HIP_CHECK(hipFree(deviceData0));

    // Cleanup for device 1
    HIP_CHECK(hipSetDevice(1));
    HIP_CHECK(hipEventDestroy(startEvent1));
    HIP_CHECK(hipEventDestroy(stopEvent1));
    HIP_CHECK(hipStreamSynchronize(stream1));
    HIP_CHECK(hipStreamDestroy(stream1));
    HIP_CHECK(hipFree(deviceData1));

    return EXIT_SUCCESS;
}
```

5.4 Peer-to-peer memory access

In multi-GPU systems, peer-to-peer memory access enables one GPU to directly read or write to the memory of another GPU. This capability reduces data transfer times by allowing GPUs to communicate directly without involving the host. Enabling peer-to-peer access can significantly improve the performance of applications that require frequent data exchange between GPUs, as it eliminates the need to transfer data through the host memory.

By adding peer-to-peer access to the example referenced in *Device selection*, data can be efficiently copied between devices. If peer-to-peer access is not activated, the call to `hipMemcpy()` still works but internally uses a staging buffer in host memory, which incurs a performance penalty.

with peer-to-peer

```
#include <hip/hip_runtime.h>

#include <cstddef>
#include <cstdlib>
#include <iostream>

#define HIP_CHECK(expression)                          \
{                                                      \
    const hipError_t status = expression;              \
    if (status != hipSuccess)                          \
    {                                                  \
        std::cerr << "HIP error " << status            \
                << ": " << hipGetErrorString(status)   \
                << " at " << __FILE__ << ":"           \
                << __LINE__ << std::endl;              \
        std::exit(EXIT_FAILURE);                       \
    }                                                  \
}

__global__ void simpleKernel(double *data, std::size_t elems)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < elems)
        data[idx] = idx * 2.0;
}

int main()
{
    int deviceCount;
    HIP_CHECK(hipGetDeviceCount(&deviceCount));
    if(deviceCount < 2)
    {
        std::cout << "This example requires at least two HIP devices." << std::endl;
        return EXIT_SUCCESS;
    }

    double* deviceData0;
    double* deviceData1;
    constexpr std::size_t elems = 1024;
    constexpr std::size_t size = elems * sizeof(double);

    int deviceId0 = 0;
    int deviceId1 = 1;

    // Enable peer access to the memory (allocated and future) on the peer device.
    // Ensure the device is active before enabling peer access.
    HIP_CHECK(hipSetDevice(deviceId0));
    HIP_CHECK(hipDeviceEnablePeerAccess(deviceId1, 0));

    HIP_CHECK(hipSetDevice(deviceId1));
    HIP_CHECK(hipDeviceEnablePeerAccess(deviceId0, 0));
```

(continues on next page)

(continued from previous page)

```
    // Set device 0 and perform operations
    HIP_CHECK(hipSetDevice(deviceId0)); // Set device 0 as current
    HIP_CHECK(hipMalloc(&deviceData0, size)); // Allocate memory on device 0
    simpleKernel<<<8, 128>>>(deviceData0, elems); // Launch kernel on device 0
    HIP_CHECK(hipDeviceSynchronize());

    // Set device 1 and perform operations
    HIP_CHECK(hipSetDevice(deviceId1)); // Set device 1 as current
    HIP_CHECK(hipMalloc(&deviceData1, size)); // Allocate memory on device 1
    simpleKernel<<<8, 128>>>(deviceData1, elems); // Launch kernel on device 1
    HIP_CHECK(hipDeviceSynchronize());

    // Use peer-to-peer access
    HIP_CHECK(hipSetDevice(deviceId0));

    // Now device 0 can access memory allocated on device 1
    HIP_CHECK(hipMemcpy(deviceData0, deviceData1, size, hipMemcpyDeviceToDevice));

    // Copy result from device 0
    double hostData0[elems];
    HIP_CHECK(hipSetDevice(deviceId0));
    HIP_CHECK(hipMemcpy(hostData0, deviceData0, size, hipMemcpyDeviceToHost));

    // Copy result from device 1
    double hostData1[elems];
    HIP_CHECK(hipSetDevice(deviceId1));
    HIP_CHECK(hipMemcpy(hostData1, deviceData1, size, hipMemcpyDeviceToHost));

    // Display results from both devices
    std::cout << "Device 0 data: " << hostData0[0] << std::endl;
    std::cout << "Device 1 data: " << hostData1[0] << std::endl;

    // Free device memory
    HIP_CHECK(hipFree(deviceData0));
    HIP_CHECK(hipFree(deviceData1));

    return EXIT_SUCCESS;
}
```

without peer-to-peer

```
#include <hip/hip_runtime.h>

#include <cstddef>
#include <cstdlib>
#include <iostream>

#define HIP_CHECK(expression)                          \
{                                                      \
    const hipError_t status = expression;              \
```

(continues on next page)

(continued from previous page)

```
    if (status != hipSuccess)                           \
    {                                                   \
        std::cerr << "HIP error " << status             \
                << ": " << hipGetErrorString(status)    \
                << " at " << __FILE__ << ":"            \
                << __LINE__ << std::endl;               \
        std::exit(EXIT_FAILURE);                        \
    }                                                   \
}

__global__ void simpleKernel(double *data, std::size_t elems)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if(idx < elems)
        data[idx] = idx * 2.0;
}

int main()
{
    int deviceCount;
    HIP_CHECK(hipGetDeviceCount(&deviceCount));
    if(deviceCount < 2)
    {
        std::cout << "This example requires at least two HIP devices." << std::endl;
        return EXIT_SUCCESS;
    }

    double* deviceData0;
    double* deviceData1;
    constexpr std::size_t elems = 1024;
    constexpr std::size_t size = elems * sizeof(double);

    int deviceId0 = 0;
    int deviceId1 = 1;

    // Set device 0 and perform operations
    HIP_CHECK(hipSetDevice(deviceId0)); // Set device 0 as current
    HIP_CHECK(hipMalloc(&deviceData0, size)); // Allocate memory on device 0
    simpleKernel<<<8, 128>>>(deviceData0, elems); // Launch kernel on device 0
    HIP_CHECK(hipDeviceSynchronize());

    // Set device 1 and perform operations
    HIP_CHECK(hipSetDevice(deviceId1)); // Set device 1 as current
    HIP_CHECK(hipMalloc(&deviceData1, size)); // Allocate memory on device 1
    simpleKernel<<<8, 128>>>(deviceData1, elems); // Launch kernel on device 1
    HIP_CHECK(hipDeviceSynchronize());

    // Use deviceData0 on device 1. This works but incurs a performance penalty.
    HIP_CHECK(hipSetDevice(deviceId1));
    HIP_CHECK(hipMemcpy(deviceData1, deviceData0, size, hipMemcpyDeviceToDevice));

    // Copy result from device 0
```

(continues on next page)

(continued from previous page)

```
    double hostData0[elems];
    HIP_CHECK(hipSetDevice(deviceId0));
    HIP_CHECK(hipMemcpy(hostData0, deviceData0, size, hipMemcpyDeviceToHost));

    // Copy result from device 1
    double hostData1[elems];
    HIP_CHECK(hipSetDevice(deviceId1));
    HIP_CHECK(hipMemcpy(hostData1, deviceData1, size, hipMemcpyDeviceToHost));

    // Display results from both devices
    std::cout << "Device 0 data: " << hostData0[0] << std::endl;
    std::cout << "Device 1 data: " << hostData1[0] << std::endl;

    // Free device memory
    HIP_CHECK(hipFree(deviceData0));
    HIP_CHECK(hipFree(deviceData1));

    return EXIT_SUCCESS;
}
```

CHAPTER

SIX

INTRODUCTION TO OPTIMIZATION PATTERNS

This chapter takes a patterns-based approach to GPU optimization. Rather than providing isolated tips or theoretical advice, each section demonstrates a complete optimization journey for a fundamental algorithmic pattern. You will start with straightforward implementations, then apply progressively sophisticated optimizations while measuring the impact of each change.

6.1 What makes a pattern

An optimization pattern is a proven strategy for solving a particular class of performance problems on GPU architectures. Each pattern addresses a frequently encountered operation like reduction, histogram computation, or matrix multiplication. The optimization techniques you learn apply broadly to similar computational structures in your own code.

Patterns in this chapter use standard HIP programming constructs, teaching principles that work across GPU architectures. You will learn to use the memory hierarchy, organize threads efficiently, and structure computations for optimal performance without relying on architecture-specific features.

6.2 Patterns covered

This chapter explores three fundamental patterns: reduction operations that efficiently aggregate values, histogram computations that manage concurrent memory updates, and matrix multiplication that maximizes compute throughput. These patterns form the foundation for many GPU computing applications.

The patterns are ordered to build your optimization skills progressively. Reduction introduces fundamental concepts like shared memory usage and bank conflict avoidance. Histogram builds on these concepts while adding complexity around concurrent updates. Matrix multiplication synthesizes multiple techniques into a comprehensive optimization example.

6.3 Learning approach

Each pattern tutorial follows a consistent structure. You begin with a naive implementation that works but performs poorly, establishing a baseline. Then you apply optimizations one at a time, measuring each change's impact. This approach helps you understand why optimizations work, not just what to do.

Performance measurements accompany each optimization step, helping you develop intuition about which techniques provide the greatest benefit. You will learn to identify optimization opportunities in your own code and select the most effective strategies for your specific workloads.

6.4 What you will learn

Working through these patterns will help you:

- Recognize common computational patterns in GPU applications
- Apply appropriate optimization strategies for different pattern types
- Implement efficient parallel algorithms for reduction, histogram, and matrix operations
- Use shared memory effectively and avoid performance pitfalls
- Measure and analyze optimization impact
- Transfer these techniques to your own applications

6.5 Prerequisites

Before working through this chapter, you should be familiar with:

- Basic HIP programming concepts, including kernel launches, thread indexing, and memory management
- GPU architecture fundamentals: threads, wavefronts (i.e., CUDA warps), workgroups, and memory hierarchy
- C++ programming and basic parallel programming concepts

CHAPTER

SEVEN

OPTIMIZING REDUCTION IN HIP

Reduction is a fundamental operation that uses a parallel pattern to combine a range of input values into a single scalar using a binary operation such as sum, maximum, or product. It appears throughout GPU workloads and is a practical vehicle for studying shared memory, thread divergence, bank conflicts, and memory bandwidth optimization.

This tutorial walks through a series of HIP kernels for summing a large array of floating-point values, with each step addressing a specific performance bottleneck. Starting from a naive interleaved-addressing kernel, each version eliminates a specific bottleneck: thread divergence, shared-memory bank conflicts, redundant synchronization barriers, and finally memory bandwidth.

The complete source file for all kernels is available at:

- `Reduction kernels`

Note

All kernels target a large array of single-precision floating-point values and are validated against a sequential reference sum. They compile with `amdclang++ -O3 -std=c++17` and run on any ROCm-supported GPU architecture.

7.1 Prerequisites

Before starting this tutorial, ensure the following are in place.

- ROCm installed and `amdclang++` available on `PATH`.
- Familiarity with the HIP execution model (grids, blocks, wavefronts) and its mapping to AMD GPU hardware (dispatches, workgroups, wavefronts).
- Application tracing and profiling using rocprofv3 installed for performance analysis.

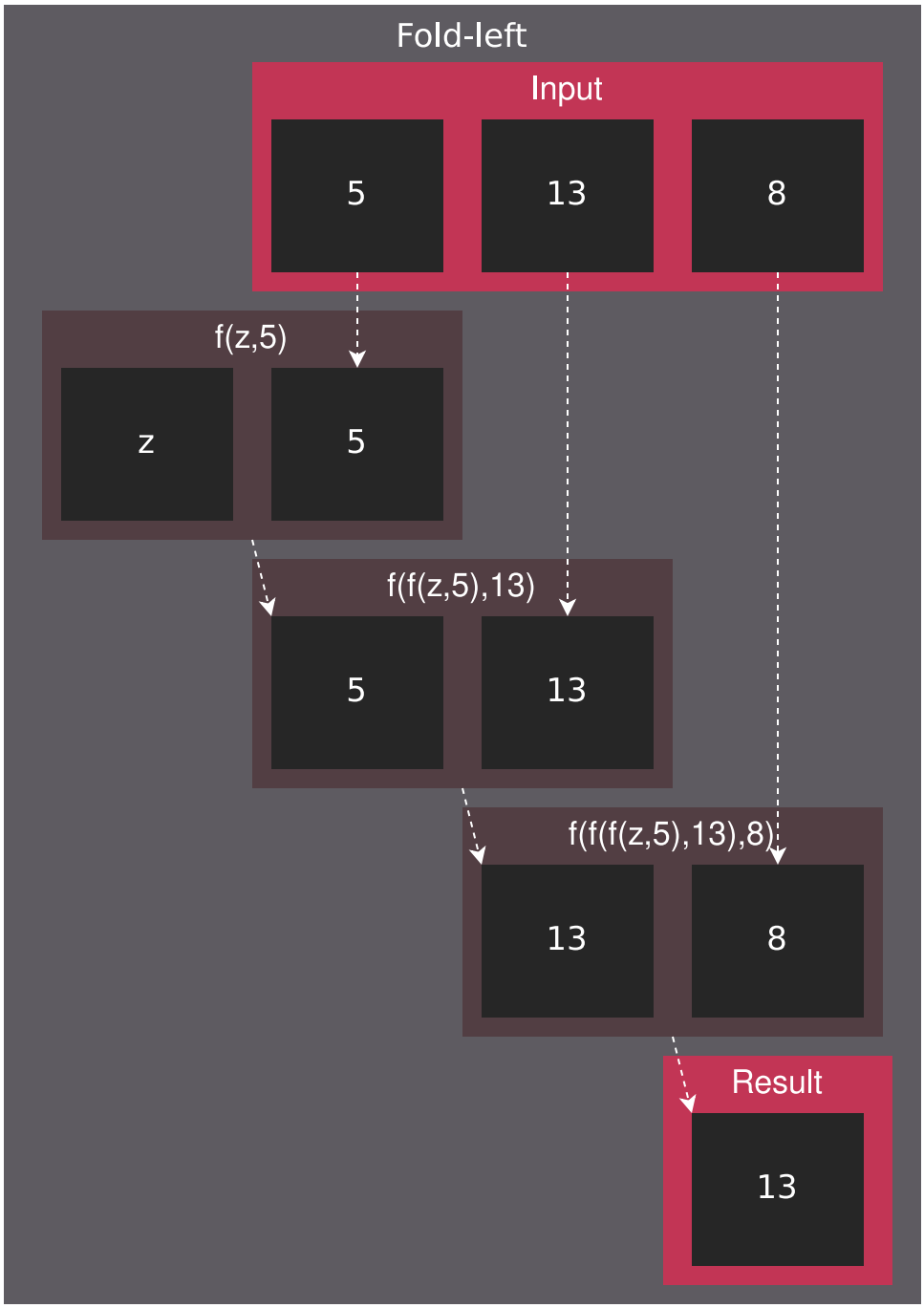
7.2 Reduction fundamentals

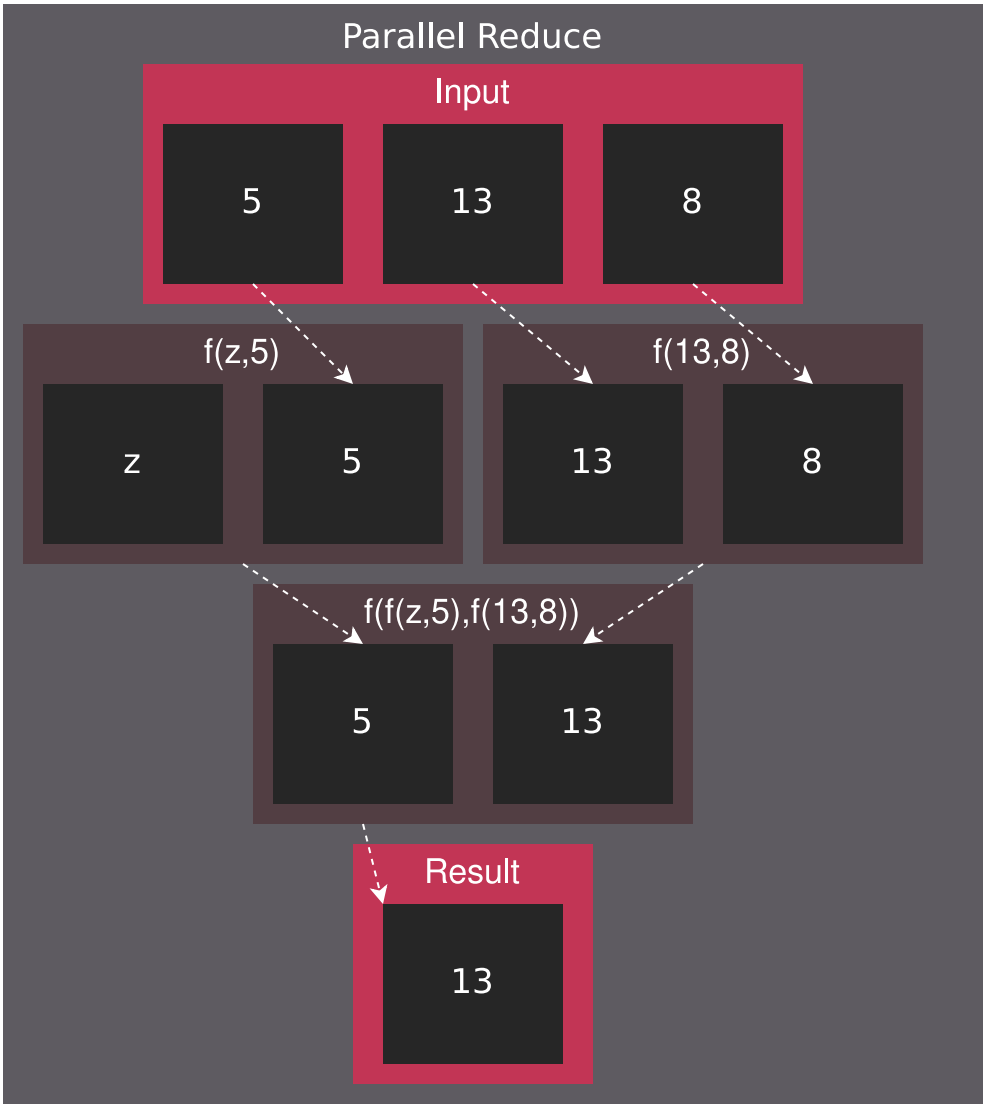
In functional programming, reduction is known as fold. In the C++ standard library, it appears as `std::accumulate` and, since C++17, as `std::reduce`. A reduction takes a range of inputs, an identity element, and a binary operation, and repeatedly applies the operation until a single value remains. For addition, the identity is 0.

On a GPU, a parallel reduction takes a tree-shaped form. In each round, threads pair up and evaluate two values, reducing them to one, halving the number of live values until one remains.

Each block independently reduces its portion of the input to a single partial result. Repeating this process on successive output arrays until only one element remains completes the device-wide reduction without requiring global synchronization within a single kernel launch.

Compile and run:


Fold-left
Input
5
13
8
f(z,5)
z
5
f(f(z,5),13)
5
13
f(f(f(z,5),13),8)
13
8
Result
13


Parallel Reduce
Input
5
13
8
f(z,5)
z
5
f(13,8)
13
8
f(f(z,5),f(13,8))
5
13
Result
13

```
amdclang++ -O3 -std=c++17 reduction.hip -o reduction
./reduction
```

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./reduction
```

The `--kernel-trace` CSV reports `End_Timestamp - Start_Timestamp` (both in nanoseconds) for each kernel dispatch. Compare the duration column across kernel variants to measure the impact of each optimization step.

7.3 Naive kernel

The naive kernel loads one element per thread into shared memory and applies a tree-shaped reduction using an interleaved addressing pattern. At each step, only threads whose index is a multiple of `2 * stride` are active and participate in the reduction.

The following kernel implements this pattern:

This causes thread divergence. Within a wavefront, all lanes must execute the same instruction at the same time. When some lanes take a branch, and others do not, the hardware must execute both paths serially with the inactive lanes masked off. In the interleaved pattern, at least one lane in each wavefront hits the `if` statement at every level of the tree, so wavefronts remain active long after the majority of their lanes have stopped doing useful work.

```
__global__ void reduction_naive(const float *input, float *output, int n)
{
    extern __shared__ float sdata[];

    int tid = threadIdx.x;
    int gid = blockIdx.x * blockDim.x + tid;

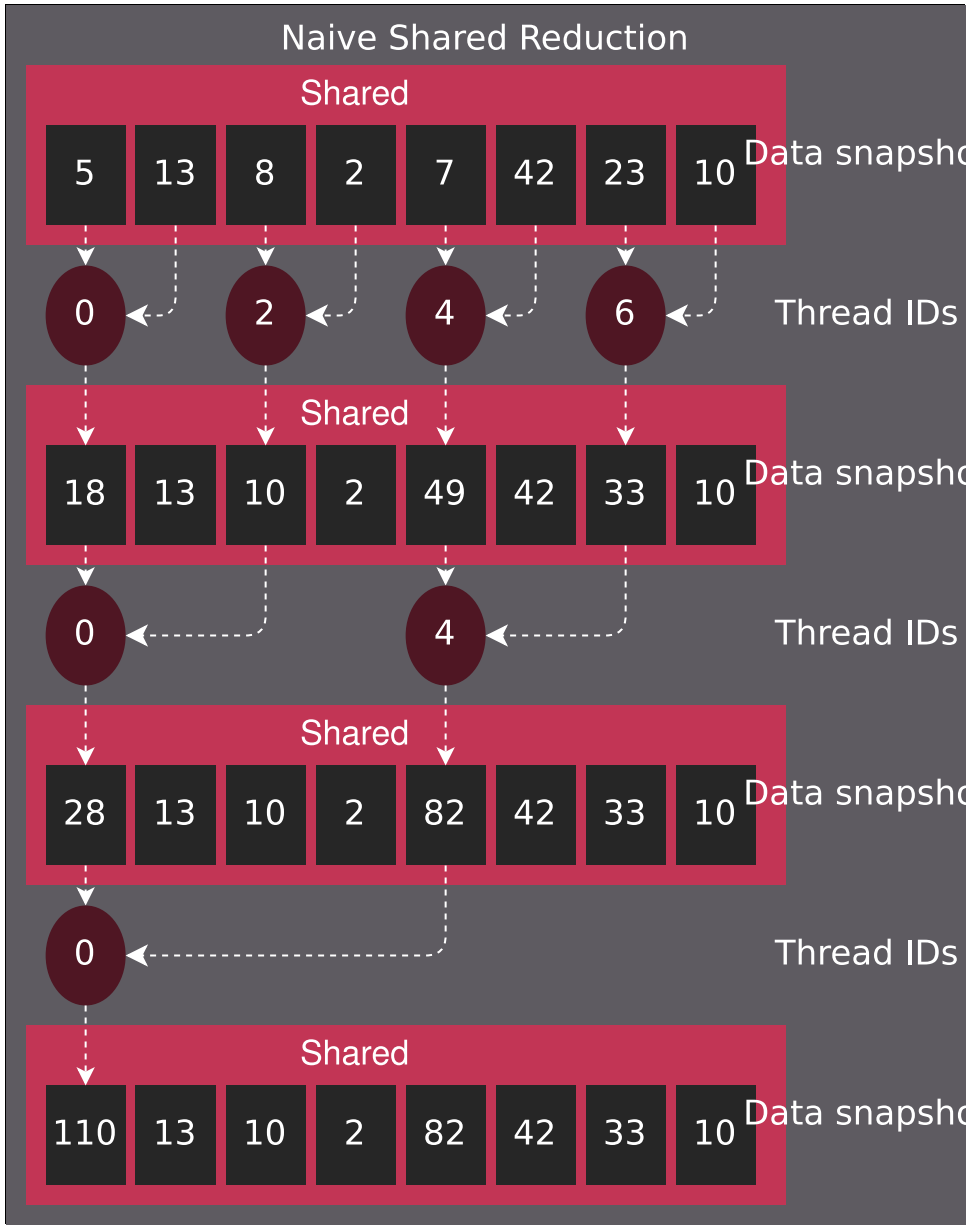
    sdata[tid] = (gid < n) ? input[gid] : 0.0f;
    __syncthreads();

    for (int stride = 1; stride < blockDim.x; stride *= 2)
    {
        if (tid % (2 * stride) == 0)
        {
            sdata[tid] += sdata[tid + stride];
        }
        __syncthreads();
    }

    if (tid == 0)
    {
        output[blockIdx.x] = sdata[0];
    }
}
```

7.3.1 What to observe

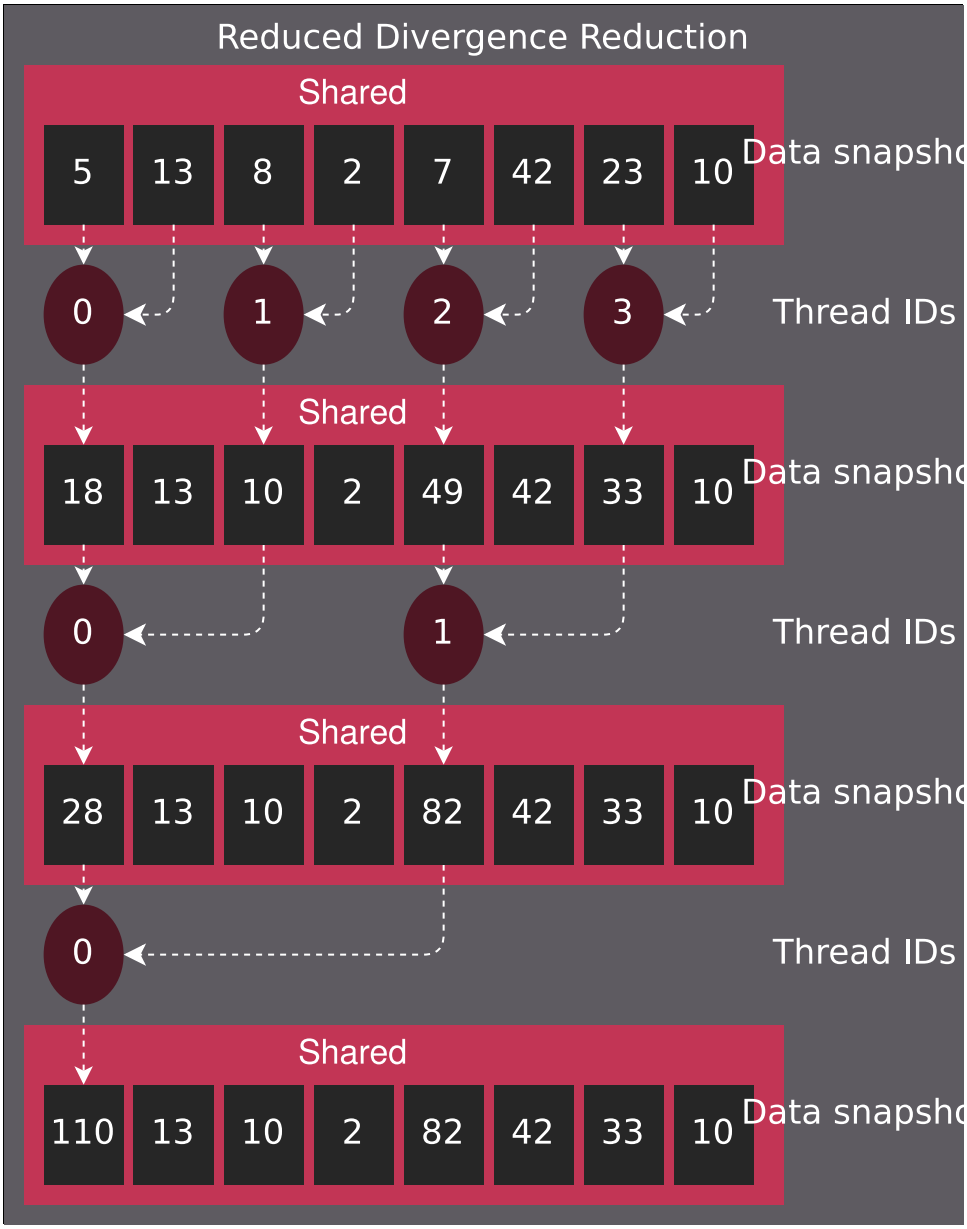
Profile the naive kernel and record the following counter.


Naive Shared Reduction
Shared
5 13 8 2 7 42 23 10
Data snapsho
0 2 4 6
Thread IDs
Shared
18 13 10 2 49 42 33 10
Data snapsho
0 4
Thread IDs
Shared
28 13 10 2 82 42 33 10
Data snapsho
0
Thread IDs
Shared
110 13 10 2 82 42 33 10
Data snapsho

Counter	What to look for
Kernel duration (kernel-trace CSV)	`End_Timestamp - Start_Timestamp` establishes the baseline. This is the slowest variant because interleaved addressing keeps wavefronts partially active at every tree level.

7.4 Reducing thread divergence

Reduce divergence by reassigning which threads are active so that inactive threads accumulate uniformly toward the upper end of the thread index range. Once an entire wavefront is inactive, it can skip directly to `__syncthreads()`, rather than executing the branch with all lanes masked.



This pattern, however, introduces a new problem: bank conflicts.

7.5 Resolving bank conflicts

With AMD GPUs, shared memory (Local Data Share, or LDS) is organized into banks of 4 bytes each. On CDNA GPUs each Compute Unit has 32 banks. On RDNA GPUs the Work Group Processor has 64 banks, sub-divided into two sets of 32 banks each affiliated with a pair of SIMD32 units; a wavefront executes on one SIMD32 and maps its accesses to the affiliated 32-bank set. A bank conflict occurs when two or more threads in the same wavefront access different addresses that map to the same bank, causing those accesses to be serialized.

The reduced-divergence pattern still causes conflicts because the stride between active threads' memory accesses doesn't align with the bank layout. The fix is to set the stride to half the block size and halve each step, so that active threads always occupy the lower half of the remaining live range and access consecutive banks.

Note

To avoid bank conflicts, read and write shared memory in a coalesced manner, where each lane in a wavefront accesses a consecutive location. For more details, see the data share operations chapter of the CDNA3 ISA or RDNA3 ISA.

```
__global__ void reduction_sequential(const float *input, float *output, int n)
{
    extern __shared__ float sdata[];

    int tid = threadIdx.x;
    int gid = blockIdx.x * blockDim.x + tid;

    sdata[tid] = (gid < n) ? input[gid] : 0.0f;
    __syncthreads();

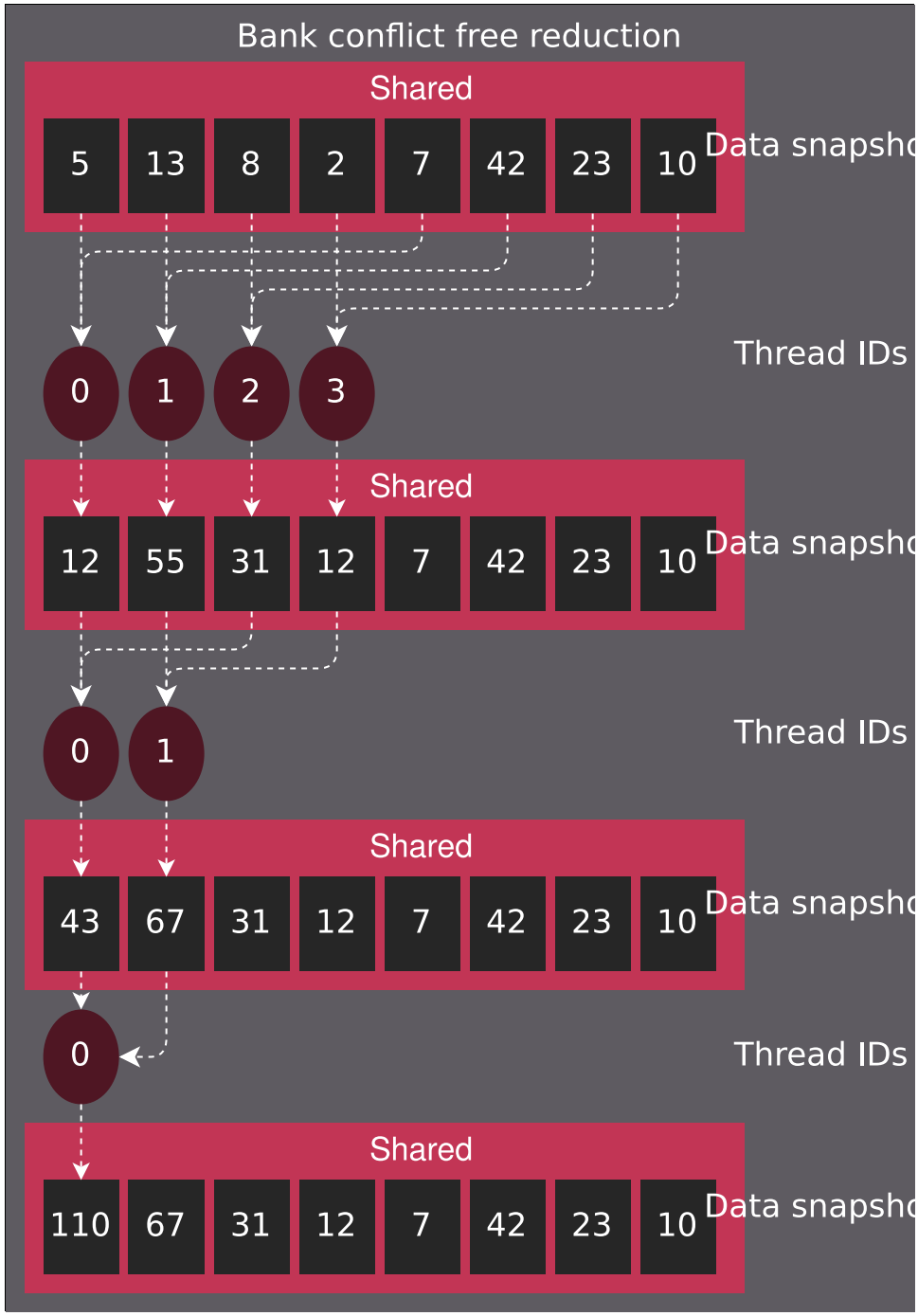
    for (int stride = blockDim.x / 2; stride > 0; stride >>= 1)
    {
        if (tid < stride)
        {
            sdata[tid] += sdata[tid + stride];
        }
        __syncthreads();
    }

    if (tid == 0)
    {
        output[blockIdx.x] = sdata[0];
    }
}
```

7.5.1 What to observe

Compare the following counters against the naive kernel baseline.

Counter	What to look for
Kernel duration (kernel-trace CSV)	`End_Timestamp - Start_Timestamp` should drop versus the naive kernel. Sequential addressing eliminates both thread divergence and bank conflicts.
`LDSBankConflict`	Should be at or near zero, confirming that consecutive active threads access consecutive LDS banks.


Bank conflict free reduction
Shared
5 13 8 2 7 42 23 10
Data snapsho
Thread IDs
0 1 2 3
Shared
12 55 31 12 7 42 23 10
Data snapsho
Thread IDs
0 1
Shared
43 67 31 12 7 42 23 10
Data snapsho
Thread IDs
0
Shared
110 67 31 12 7 42 23 10
Data snapsho

7.6 Wavefront reduction

Every `__syncthreads()` operation in the reduction loop is necessary while threads from different wavefronts are cooperating. Within a single wavefront, however, threads execute in lockstep: they all advance through instructions together, so a write by one lane is immediately visible to all other lanes in the same wavefront without a barrier. Once the active thread count drops to one wavefront, the remaining barriers are unnecessary overhead.

The wavefront reduction kernel exploits this by restructuring the algorithm into two phases. First, each wavefront independently reduces its own slice of shared memory without any barriers.

Lane 0 of each wavefront then writes its partial result into a compact staging area, and a single `__syncthreads()` makes all partial results visible. The first wavefront then reduces the staging area, again without barriers.

```
template<int WarpSize>
__device__ __forceinline__ void warp_reduce(float *sdata, int tid, int lane)
{
    if (WarpSize == 64) { if (lane < 32) { sdata[tid] += sdata[tid + 32]; } }
                          if (lane < 16) { sdata[tid] += sdata[tid + 16]; }
                          if (lane <  8) { sdata[tid] += sdata[tid +  8]; }
                          if (lane <  4) { sdata[tid] += sdata[tid +  4]; }
                          if (lane <  2) { sdata[tid] += sdata[tid +  2]; }
                          if (lane <  1) { sdata[tid] += sdata[tid +  1]; }
}
```

```
template<int WarpSize>
__global__ __launch_bounds__(block_size)
void reduction_warp(const float *input, float *output, int n)
{
    extern __shared__ float sdata[];

    int tid    = threadIdx.x;
    int gid    = blockIdx.x * blockDim.x + tid;
    int lane   = tid % WarpSize;
    int warpid = tid / WarpSize;

    sdata[tid] = (gid < n) ? input[gid] : 0.0f;

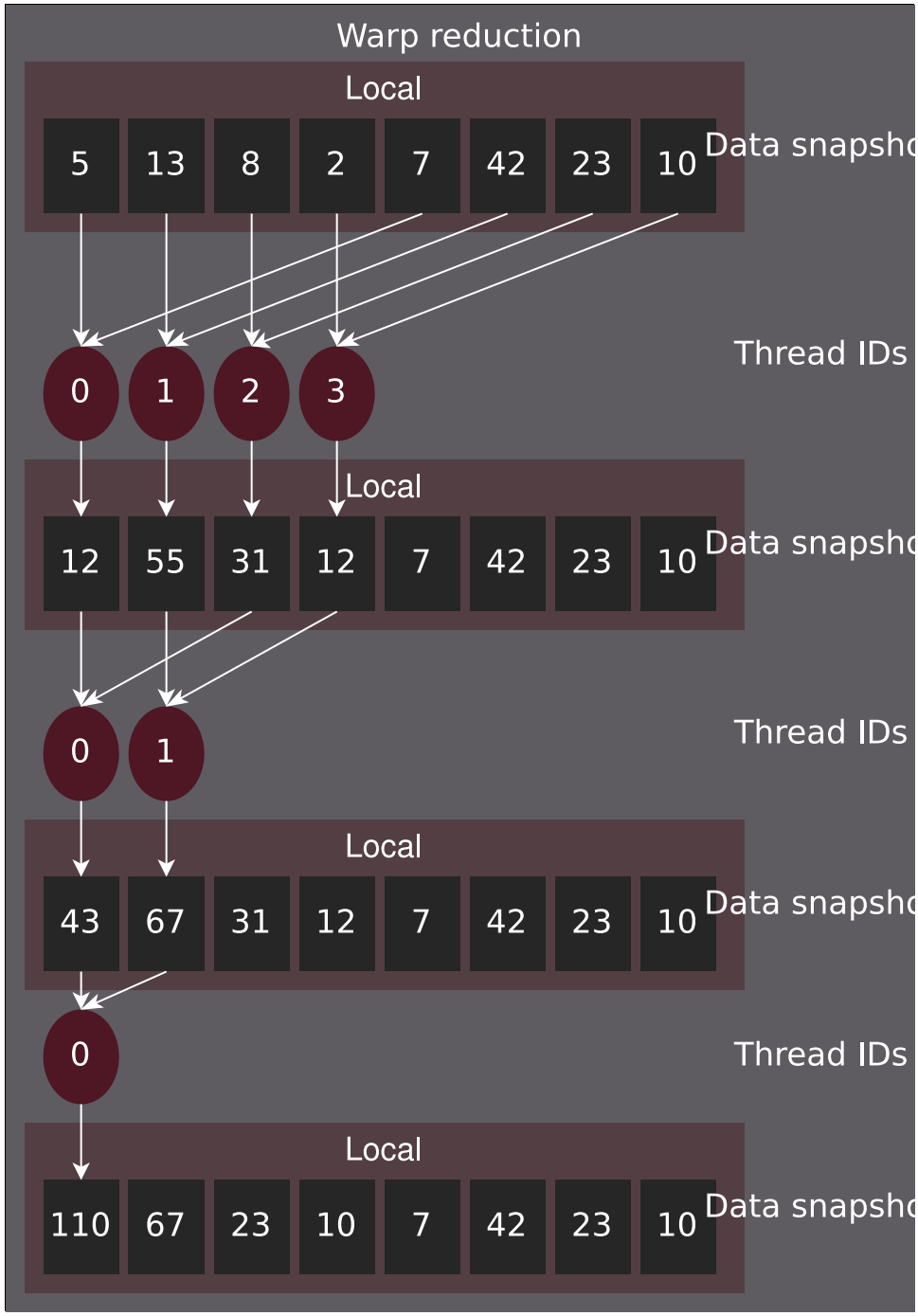
    warp_reduce<WarpSize>(sdata, tid, lane);

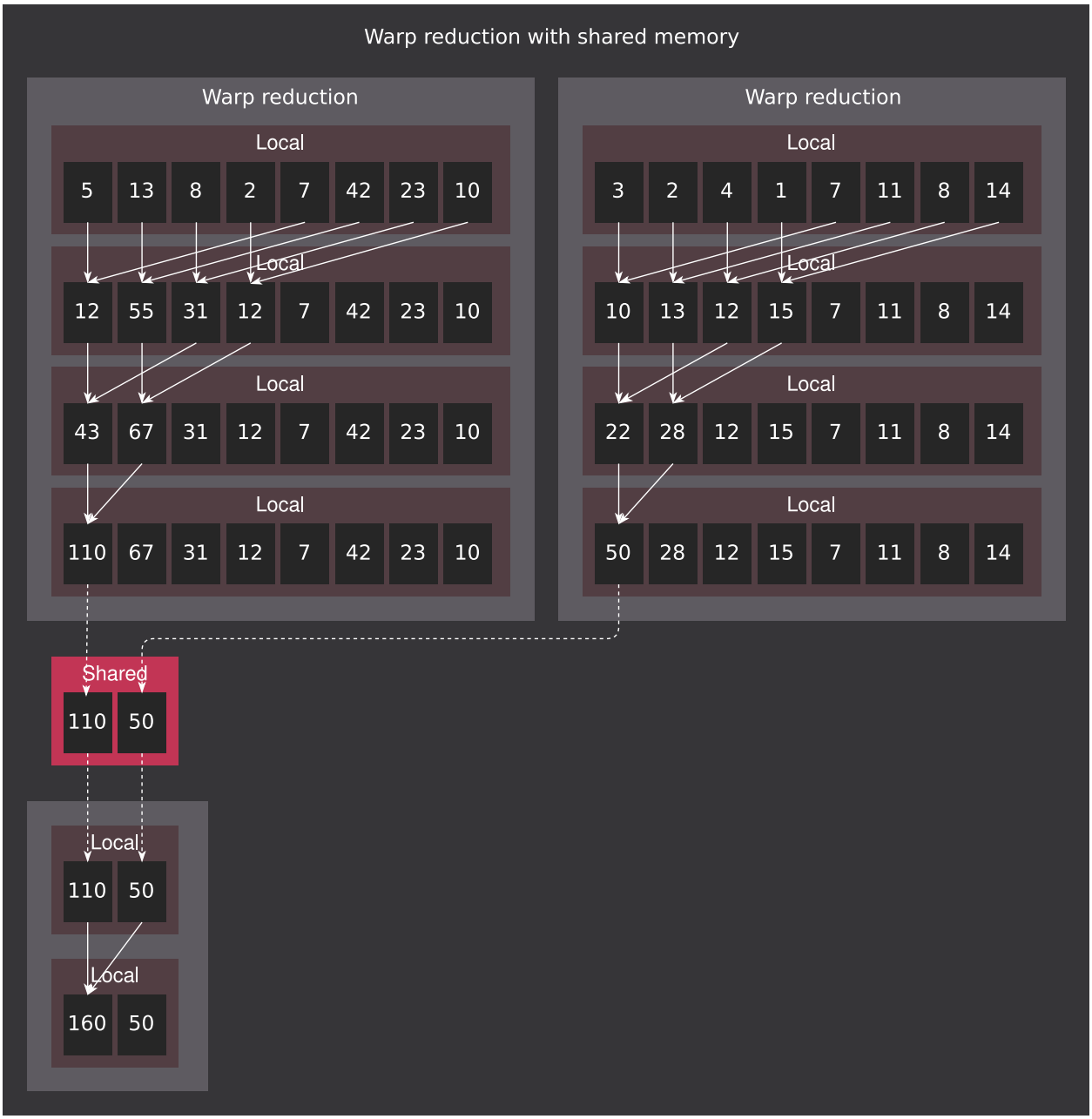
    int num_warps = blockDim.x / WarpSize;
    if (lane == 0)
    {
        sdata[blockDim.x + warpid] = sdata[tid];
    }
    __syncthreads();

    if (warpid == 0)
    {
        sdata[tid] = (lane < num_warps) ? sdata[blockDim.x + lane] : 0.0f;
        warp_reduce<WarpSize>(sdata, tid, lane);
    }

    if (tid == 0)
    {
```

(continues on next page)


Warp reduction
Local
5 13 8 2 7 42 23 10
Data snapsho
Thread IDs
0 1 2 3
Local
12 55 31 12 7 42 23 10
Data snapsho
Thread IDs
0 1
Local
43 67 31 12 7 42 23 10
Data snapsho
Thread IDs
0
Local
110 67 23 10 7 42 23 10
Data snapsho


Warp reduction with shared memory
Warp reduction
Local
5 13 8 2 7 42 23 10
Local
12 55 31 12 7 42 23 10
Local
43 67 31 12 7 42 23 10
Local
110 67 31 12 7 42 23 10
Warp reduction
Local
3 2 4 1 7 11 8 14
Local
10 13 12 15 7 11 8 14
Local
22 28 12 15 7 11 8 14
Local
50 28 12 15 7 11 8 14
Shared
110 50
Local
110 50
Local
160 50

(continued from previous page)

```
        output[blockIdx.x] = sdata[0];
    }
}
```

> **Note**
>
> The wavefront-level reduction shown here uses shared memory. On AMD GPUs, the same result can be achieved without shared memory traffic using shuffle instructions or Data Parallel Primitives (DPP), which exchange values between lanes entirely in registers. These techniques are covered in the *Introduction to Compiler builtins* chapter.

7.6.1 WGP mode and CU mode on RDNA GPUs

For a full list of supported GPUs, see the ROCm system requirements.

On CDNA GPUs, the array is organized as a set of Compute Unit (CU) pipelines. Each CU contains four SIMD64 units and its own Local Data Share (LDS), which threads from wavefronts running on that CU can access. CDNA does not offer Work Group Processor mode as RDNA does, so the following information does not apply.

On RDNA GPUs, the array is organized as a set of Work Group Processor (WGP) pipelines. Each WGP contains two CUs, each with two SIMD32 units. The LDS is attached to the WGP, so threads from different wavefronts can access the same LDS if they run on CUs within the same WGP.

Wavefronts are dispatched in one of two modes. These control whether wavefronts are distributed across two SIMD32s within a single CU (CU mode) or across all four SIMD32s within a WGP (WGP mode).

CU mode executes two wavefronts per block on a CU and provides only half the LDS to each wavefront. Independence between CUs can improve performance for workloads that avoid inter-wavefront communication.

WGP mode executes four wavefronts per block on a WGP with a shared LDS. It can increase occupancy and improve performance for workloads without heavy inter-wavefront communication, but it can degrade performance for programs that rely on atomics or extensive inter-wavefront communication through shared memory.

The wavefront reduction kernel communicates partial results between wavefronts through shared memory, making it sensitive to this distinction. The inter-wavefront staging step benefits from CU mode because all wavefronts in the block are guaranteed to share the same LDS instance. Compile with `-mcumode` to enable CU mode on RDNA GPUs. Memory-bandwidth-bound kernels such as the vectorized loads kernel are unaffected by this setting.

7.6.2 What to observe

Compare the following counters against the sequential-addressing kernel.

Counter	What to look for
Kernel duration (kernel-trace CSV)	`End_Timestamp - Start_Timestamp` should drop versus the sequential kernel. Eliminating unnecessary `__syncthreads()` barriers removes synchronization overhead.
`SQ_WAIT_INST_LD`	Reduction in LDS stall cycles (fewer barriers mean less time waiting for shared memory to become consistent).

7.7 Vectorized loads

All kernels so far issue one 32-bit load per thread. The GPU memory system can issue 128-bit loads at the same cost, so replacing four scalar loads with a single `float4` read quadruples the data moved per instruction. Each thread loads four consecutive elements, reduces them to a scalar in registers, and then enters the same wavefront reduction as before.

```
template<int WarpSize>
__global__ __launch_bounds__(block_size)
void reduction_float4(const float *input, float *output, int n)
{
    extern __shared__ float sdata[];

    int tid    = threadIdx.x;
    int gid    = blockIdx.x * (blockDim.x * 4) + tid;
    int lane   = tid % WarpSize;
    int warpid = tid / WarpSize;

    float val = 0.0f;
    if (gid + 3 * blockDim.x < n)
    {
        float4 v = reinterpret_cast<const float4 *>(input)[blockIdx.x * blockDim.x + tid];
        val = v.x + v.y + v.z + v.w;
    }
    else
    {
        if (gid                   < n) { val += input[gid]; }
        if (gid +     blockDim.x < n) { val += input[gid +     blockDim.x]; }
        if (gid + 2 * blockDim.x < n) { val += input[gid + 2 * blockDim.x]; }
        if (gid + 3 * blockDim.x < n) { val += input[gid + 3 * blockDim.x]; }
    }

    int num_warps = blockDim.x / WarpSize;
    sdata[tid] = val;
    warp_reduce<WarpSize>(sdata, tid, lane);

    if (lane == 0)
    {
        sdata[blockDim.x + warpid] = sdata[tid];
    }
    __syncthreads();

    if (warpid == 0)
    {
        sdata[tid] = (lane < num_warps) ? sdata[blockDim.x + lane] : 0.0f;
        warp_reduce<WarpSize>(sdata, tid, lane);
    }

    if (tid == 0)
    {
        output[blockIdx.x] = sdata[0];
    }
}
```

Note

The `float4` reinterpret cast requires the input pointer to be 16-byte aligned. Allocations from `hipMalloc` satisfy this requirement.

7.7.1 What to observe

Compare the following counters against the wavefront reduction kernel.

Counter	What to look for
Kernel duration (kernel-trace CSV)	`End_Timestamp - Start_Timestamp` should drop versus the wavefront reduction kernel. Each thread moves four times as much data per instruction.
`SQ_INST_CYCLES_VMEM` (RDNA) / `SQ_INST_CYCLES_VMEM_RD` (CDNA)	Reduction in VMEM instruction cycles (fewer instructions for the same total data moved).

7.8 Device-level reduction

The block kernels above each produce one partial result per block. To reduce an entire array to a scalar, apply the block kernel repeatedly on successively smaller arrays, alternating between two scratch buffers, until only one value remains. For an input of N elements processed B elements per block, this completes in $\log_B(N)$ passes.

```
    {
        float *d_partial_a = nullptr;
        float *d_partial_b = nullptr;
        HIP_CHECK(hipMalloc(&d_partial_a, sizeof(float) * nb1));
        HIP_CHECK(hipMalloc(&d_partial_b, sizeof(float) * nb1));

        int          current = input_size;
        const float *src     = d_input;
        float       *dst     = d_partial_a;

        while (current > 1)
        {
            int nb = (current + block_size * 4 - 1) / (block_size * 4);
            if (warpsizehost == 64)
            {
                reduction_float4<64><<<nb, block_size, smem_warp>>>(src, dst,␣
↪current);
            }
            else
            {
                reduction_float4<32><<<nb, block_size, smem_warp>>>(src, dst,␣
↪current);
            }
            HIP_CHECK(hipGetLastError());
            current = nb;
            if (current > 1)
            {
                src = dst;
                dst = (dst == d_partial_a) ? d_partial_b : d_partial_a;
            }
        }

        HIP_CHECK(hipDeviceSynchronize());
        float result = 0.0f;
        HIP_CHECK(hipMemcpy(&result, dst, sizeof(float), hipMemcpyDeviceToHost));
```

(continues on next page)

(continued from previous page)

```
34
35          // Compute the scalar reference by reducing the per-block CPU partials.
36          float ref_scalar = 0.0f;
37          for (float v : ref_bs4)
38          {
39              ref_scalar += v;
40          }
41          float rel_err = std::fabs(result - ref_scalar) / (std::fabs(ref_scalar) + 1.
    ↪0f);
42
43          std::cout << "  Device reduction: ";
44          std::cout << (rel_err < 1e-3f ? "PASSED" : "FAILED") << std::endl;
45
46          HIP_CHECK(hipFree(d_partial_a));
47          HIP_CHECK(hipFree(d_partial_b));
48      }
```

7.9 Further reading

The following resources provide deeper coverage of the tools and hardware referenced in this tutorial.

- rocPRIM - production-quality reduction primitives that handle edge cases and automatically apply architecture-specific tuning.
- rocprofv3 documentation - detailed guide to timeline and counter profiling.
- AMD GPU architecture guides (ISA references) - VGPR budgets, LDS bank geometry, and wavefront scheduling details for each architecture family.

CHAPTER

EIGHT

OPTIMIZING HISTOGRAM IN HIP

Histogram is an operation that counts how often each value (or range of values) appears in an input dataset. It appears throughout GPU workloads, including image processing, radix sort, database operations, and machine learning. The challenge on GPUs is that the output location of each write is determined by the input value at runtime, meaning multiple threads can attempt to update the same output bin simultaneously. Efficiently managing this concurrent access is the central optimization problem.

This tutorial walks through a series of HIP kernels for computing a 256-bin histogram over a large array of unsigned integers. Starting from a naive kernel that issues one global atomic per input element, each step reduces global atomic traffic: first by moving accumulation into shared memory, then by having each thread process more elements so fewer blocks are launched. The final approach eliminates global atomics at the merge step entirely by having each block write its local histogram to a private slice of a temporary buffer, then summing those slices in a separate reduction kernel.

The complete source file for all kernels is available at:

- `Histogram kernels`

Note

All kernels compute a 256-bin histogram over a large array of unsigned integers and are validated against a sequential reference implementation. They compile with `amdclang++ -O3 -std=c++17` and run on any ROCm-supported GPU architecture.

8.1 Prerequisites

Before starting this tutorial, ensure the following are in place.

- ROCm installed and `amdclang++` available on `PATH`.
- Familiarity with the HIP execution model (grids, blocks, wavefronts) and its mapping to AMD GPU hardware (dispatches, workgroups, wavefronts).
- Application tracing and profiling using rocprofv3 installed for performance analysis.

8.2 Histogram fundamentals

A histogram maps each element of an input array to one of `num_bins` output counters, called bins, and increments that counter. Formally, for an input sequence $x_1, \ldots, x_N$ and a bin mapping function f, the count for bin b is:

$$H[b] = \sum_{i=1}^{N} \delta\big(b - \lfloor f(x_i) \rfloor\big)$$

where $\delta()$ is 1 when its argument is zero and 0 otherwise. For a simple integer input, $f(x_i) = x_i \bmod B$ maps each value to one of B bins by remainder.

The algorithm consists of three steps:

1. Read each input element.
2. Determine its bin.
3. Increment that bin's counter.

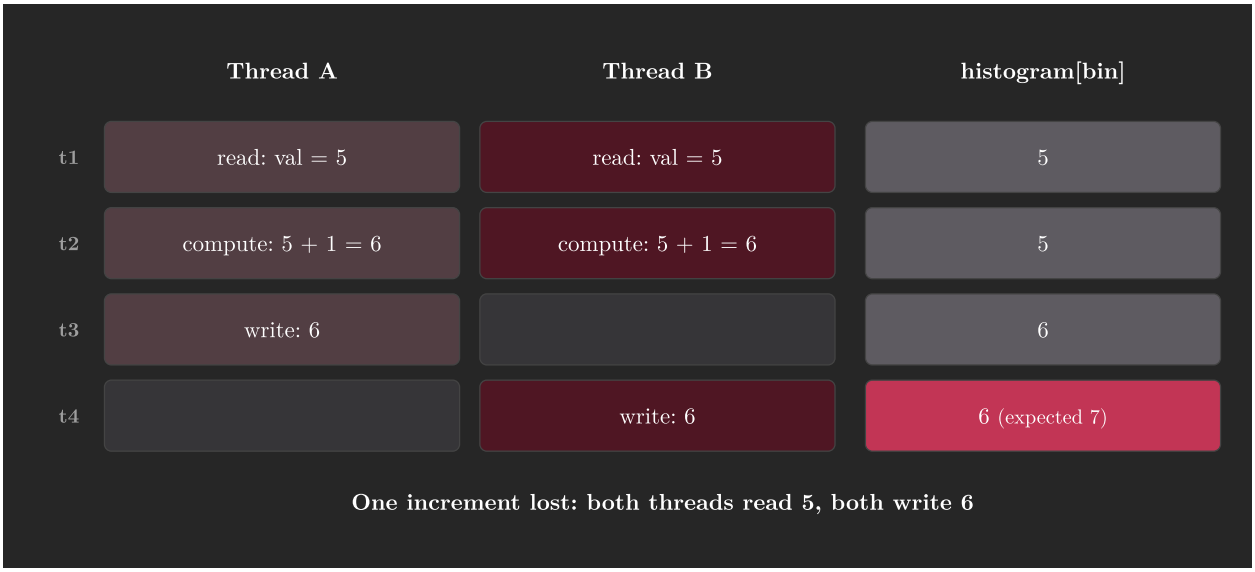
On a CPU, this is straightforward. On a GPU, thousands of threads execute these steps simultaneously, and many might land in the same bin at the same time.

8.3 Race conditions

A race condition occurs when two or more threads concurrently attempt to read-modify-write the same memory location. Consider two threads that both want to increment `histogram[bin]`:

```
histogram[bin] = histogram[bin] + 1;
```

If both threads read the value before either writes back, they both compute the same result, and the second write overwrites the first, so one increment is lost. This is a natural consequence of parallel execution: with many threads running concurrently across compute units, some will read the same value before any of them writes back.



When multiple threads map to the same bin, this kind of overlap is expected. Atomic operations, covered in the next section, are the standard solution.

8.4 Atomic operations

An atomic operation executes a read-modify-write sequence as an indivisible unit. No other thread can observe a partially completed operation or interleave its own update between the read and write. From the hardware's perspective, the memory arbitration unit locks the relevant cache line, performs the update, and releases the lock. All competing threads observe results as if the operations occurred in a single sequential order.

HIP provides a set of atomic primitives for both global and shared memory:

Operation	Description
`atomicAdd`	Adds a value to a memory location and returns the old value.
`atomicSub`	Subtracts a value from a memory location and returns the old value.
`atomicExch`	Exchanges a register value with a memory location.
`atomicCAS`	Compares a memory location to an expected value and, if equal, replaces it with a new value. The fundamental building block for custom atomic operations.
`atomicMax`/`atomicMin`	Updates a memory location to the maximum or minimum of its current value and a given value.
`atomicInc`/`atomicDec`	Atomically increments or decrements a counter, wrapping at a boundary.

Atomic operations can target shared memory (block scope), global memory (device scope), or system memory, depending on hardware support. For more information, see the GPU atomics operations reference.

Compile and run:

```
amdclang++ -O3 -std=c++17 histogram.hip -o histogram
./histogram
```

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./histogram
```

The `--kernel-trace` CSV reports `End_Timestamp - Start_Timestamp` (both in nanoseconds) for each kernel dispatch. Compare the duration column across kernel variants to measure the impact of each optimization step.

8.5 Naive kernel

The naive kernel issues one global `atomicAdd` per thread:

```
__global__ void histogram_naive(const unsigned int *input, unsigned int *histogram,␣
↪int n)
{
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid < n)
    {
        unsigned int bin = input[gid] % num_bins;
        atomicAdd(&histogram[bin], 1u);
    }
}
```

Global memory atomics must traverse the full memory hierarchy. When many threads target the same bin, the hardware serializes their updates. That is, only one can proceed at a time while the others stall. This is called **atomic contention**, and it limits throughput in proportion to the number of threads competing for the same address.

Two factors make contention worse in practice:

- **Hot bins:** When the input distribution is skewed, a small number of bins receive a disproportionate share of increments. Every thread targeting a hot bin serializes against every other thread.
- **Wavefront serialization:** Within a wavefront, if multiple lanes map to the same bin, the hardware issues their atomic operations one at a time, stalling the whole wavefront until each completes.

The example code uses a skewed input where every fourth element is fixed to bin 1, to reflect a realistic distribution in which one bin is significantly busier than the others.

8.5.1 What to observe

Profile the naive kernel and record the following counter.

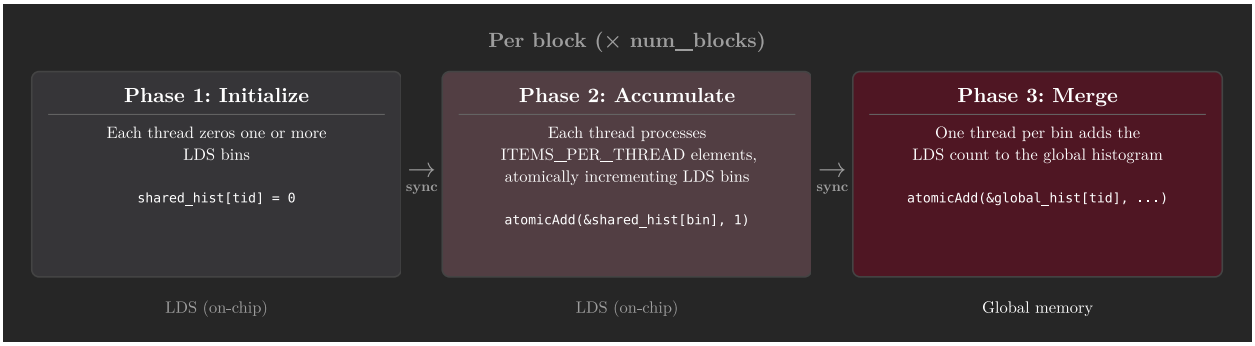
Counter	What to look for
Kernel duration (kernel-trace CSV)	`End_Timestamp - Start_Timestamp` establishes the baseline. Global atomic contention on hot bins makes this the slowest variant.

8.6 Shared memory histogram

Atomic operations in shared memory (the Local Data Share, or LDS, on AMD GPUs) are an order of magnitude faster than global memory atomics because the LDS is on-chip and directly connected to the compute units. The shared memory kernel reduces global atomic traffic in two independent ways: moving accumulation into LDS, and having each thread process multiple elements so fewer blocks are launched overall.

The kernel has three phases:

1. Initialize the per-block histogram in LDS to zero.
2. Each thread processes `ITEMS_PER_THREAD` input elements (default: 16), atomically incrementing the appropriate LDS bin for each.
3. One thread per bin merges the LDS histogram into the global histogram with a single global atomic.




```
__global__ void histogram_shared(const unsigned int *input,
                                 unsigned int *histogram,
                                 int n)
{
    __shared__ unsigned int shared_hist[num_bins];

    int tid = threadIdx.x;
    int base = (blockIdx.x * blockDim.x * ITEMS_PER_THREAD) + tid;

    // 1. Initialize shared histogram
    for (int i = tid; i < num_bins; i += blockDim.x)
    {
        shared_hist[i] = 0;
    }
    __syncthreads();

    // 2. Process multiple items per thread (grid-stride style)
    #pragma unroll
```

(continues on next page)

(continued from previous page)

```
    for (int i = 0; i < ITEMS_PER_THREAD; i++)
    {
        int idx = base + i * blockDim.x;

        if (idx < n)
        {
            unsigned int bin = input[idx] % num_bins;
            atomicAdd(&shared_hist[bin], 1u);
        }
    }

    __syncthreads();

    // 3. Merge into global histogram
    for (int i = tid; i < num_bins; i += blockDim.x)
    {
        atomicAdd(&histogram[i], shared_hist[i]);
    }
}
```

The inner loop uses the stride `i * blockDim.x`, so consecutive threads in a wavefront always read consecutive memory addresses in each iteration, so the access pattern remains coalesced throughout. Because `ITEMS_PER_THREAD` is a compile-time constant, `#pragma unroll` allows the compiler to eliminate the loop counter and branch overhead.

With `block_size = 256` and `ITEMS_PER_THREAD = 16`, each block covers 4,096 input elements. For a 16 M-element input, this launches 4,096 blocks, compared to 65,536 for the naive kernel. The global merge step issues at most `num_bins` atomics per block: 256 per block × 4,096 blocks = ~1 M global atomics total, versus 16 M for the naive kernel.

> **Note**
>
> The LDS histogram occupies `num_bins * sizeof(unsigned int)` bytes. For 256 bins, this is 1 KB, well within the 64 KB of LDS available per Compute Unit on CDNA GPUs and per Work Group Processor on RDNA GPUs.

8.6.1 What to observe

Compare the following counter against the naive kernel baseline.

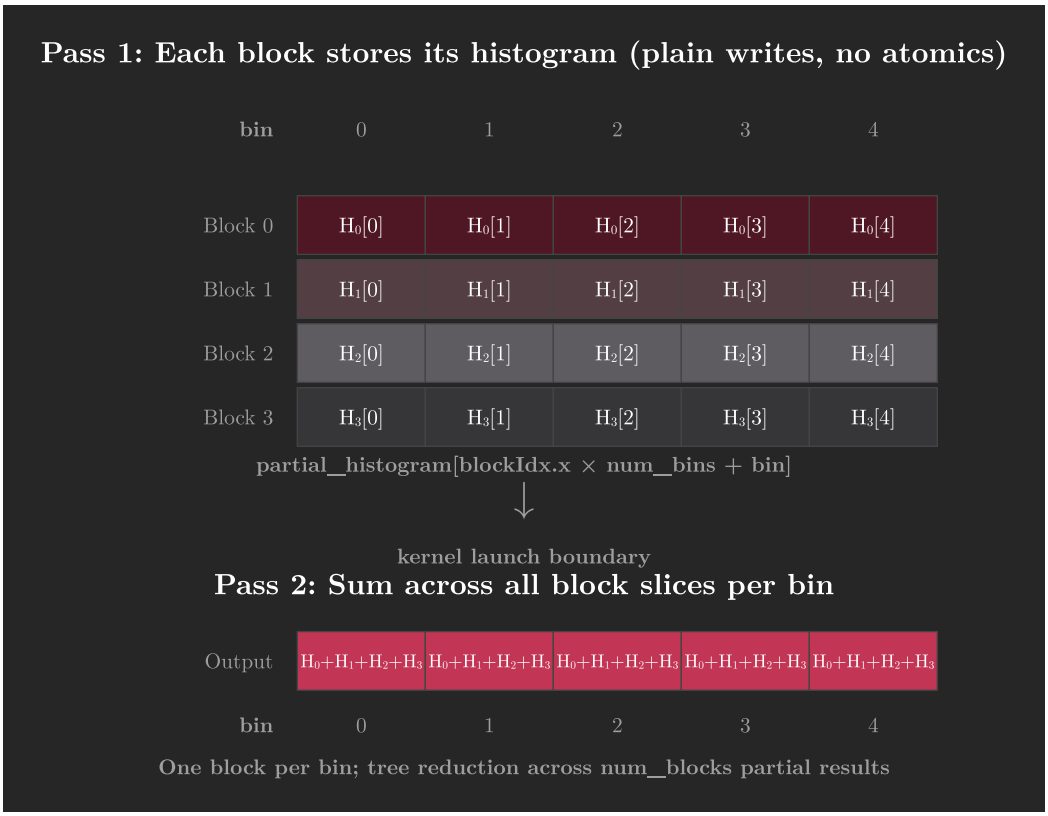
Counter	What to look for
Kernel duration (kernel-trace CSV)	`End_Timestamp - Start_Timestamp` should drop significantly versus the naive kernel. LDS atomics are an order of magnitude faster than global atomics, and processing multiple elements per thread reduces the total block count and global merge traffic.

8.7 Partial histograms

The shared memory kernel still issues up to `num_bins` global atomics per block during the merge phase. For 4,096 blocks and 256 bins, that is roughly one million global atomic operations.

A partial histogram avoids this by deferring the merge entirely. Instead of each block atomically adding its local counts into a single shared output array, each block writes its `num_bins` counts to its own reserved slice of a temporary `partial_histogram` buffer - using plain stores, with no contention. The global output is then computed in a second kernel

that sums across all the per-block slices for each bin. Because the two passes are separated, neither requires global atomics: the first pass uses conflict-free stores, and the second pass is a straightforward parallel reduction (see *Optimizing reduction in HIP*).



The first kernel is identical to the shared memory kernel except for the merge step, which becomes a plain store rather than a global atomic:

```
__global__ void histogram_partial(const unsigned int *input,
                                  unsigned int *partial_histogram,
                                  int n)
{
    __shared__ unsigned int shared_hist[num_bins];

    int tid  = threadIdx.x;
    int base = (blockIdx.x * blockDim.x * ITEMS_PER_THREAD) + tid;

    for (int i = tid; i < num_bins; i += blockDim.x)
    {
        shared_hist[i] = 0;
    }
    __syncthreads();

    #pragma unroll
    for (int i = 0; i < ITEMS_PER_THREAD; i++)
    {
        int idx = base + i * blockDim.x;
        if (idx < n)
```

(continues on next page)

(continued from previous page)

```
        {
            unsigned int bin = input[idx] % num_bins;
            atomicAdd(&shared_hist[bin], 1u);
        }
    }
    __syncthreads();

    unsigned int *block_hist = partial_histogram + blockIdx.x * num_bins;
    for (int i = tid; i < num_bins; i += blockDim.x)
    {
        block_hist[i] = shared_hist[i];
    }
}
```

The `partial_histogram` array has `num_blocks * num_bins` elements, laid out as `partial_histogram[blockIdx.x * num_bins + bin]`. Writing a full row of `num_bins` consecutive values per block keeps the stores coalesced.

The second kernel assigns one block to each bin. Threads within each block accumulate their share of the partial results with a stride loop, then reduce to a single bin count using a shared memory tree reduction:

```
__global__ void histogram_reduce(const unsigned int *partial_histogram,
                                 unsigned int *histogram,
                                 int num_partial)
{
    int bin = blockIdx.x;
    int tid = threadIdx.x;

    if (bin >= num_bins)
        return;

    unsigned int sum = 0;

    // Each thread accumulates a chunk
    for (int i = tid; i < num_partial; i += blockDim.x)
    {
        sum += partial_histogram[i * num_bins + bin];
    }

    // Shared memory reduction
    __shared__ unsigned int sdata[block_size];
    sdata[tid] = sum;
    __syncthreads();

    // Tree reduction
    for (int s = blockDim.x / 2; s > 0; s >>= 1)
    {
        if (tid < s)
            sdata[tid] += sdata[tid + s];
        __syncthreads();
    }

    // Write result
```

(continues on next page)

(continued from previous page)

```
    if (tid == 0)
        histogram[bin] = sdata[0];
}
```

The kernel launches a number of blocks equal to `num_bins`, with each block consisting of `block_size` threads. Every thread accumulates data from every `blockDim.x`-th partial block into a register accumulator. It then writes the result to shared memory and participates in the tree reduction process. The access pattern `partial_histogram[i * num_bins + bin]` strides by `num_bins` between iterations, so consecutive threads in a wavefront read consecutive addresses so the reads are coalesced.

Note

The `partial_histogram` buffer requires `num_blocks * num_bins * sizeof(unsigned int)` bytes of device memory. With `ITEMS_PER_THREAD = 16`, `block_size = 256`, and a 16 M-element input, this is 4,096 blocks × 256 bins × 4 bytes = 4 MB.

8.7.1 What to observe

Compare the following counters against the shared memory kernel.

Counter	What to look for
Kernel duration (kernel-trace CSV)	Compare `End_Timestamp - Start_Timestamp` for both passes (the partial histogram kernel and the reduction kernel) against the single-pass shared memory kernel. The two-pass approach eliminates global atomics at the merge step but introduces overhead from the additional kernel launch and the tree reduction.
`VGPR_Count` (kernel-trace CSV)	From `--kernel-trace` CSV: compare register usage between the shared-memory kernel and the partial histogram kernel to confirm similar resource consumption.

Tip

Increasing `ITEMS_PER_THREAD` reduces the block count, which reduces merge overhead in all variants. Performance improves until the kernel becomes compute-bound within each block and the benefit plateaus. Use `rocprofv3 --kernel-trace` to find the crossover point for your target GPU.

8.8 Further reading

The following resources provide deeper coverage of the tools and hardware referenced in this tutorial.

- rocPRIM - production-quality histogram primitives that handle edge cases and automatically apply architecture-specific tuning.
- rocprofv3 documentation - detailed guide to timeline and counter profiling.
- AMD GPU architecture guides (ISA references) - VGPR budgets, LDS bank geometry, and wavefront scheduling details for each architecture family.

CHAPTER

NINE

OPTIMIZING GEMM IN HIP

Matrix multiplication is one of the most fundamental GPU workloads. It underlies the compute-intensive layers of deep neural networks - fully connected layers, convolutional layers expressed as implicit GEMMs, and attention mechanisms - and is central to scientific computing, computer vision, and recommendation systems. GPUs are heavily optimized for matrix multiplication, and understanding how to write an efficient GEMM kernel is an effective way to learn how GPU hardware resources interact.

This tutorial walks through seven progressive optimization steps applied to a general-purpose single-precision (FP32) matrix multiplication kernel (General Matrix Multiply, or GEMM: $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$). Each step builds on the previous one, introducing a specific technique and explaining how to measure its effect with the ROCm performance analysis stack.

The complete source files for all steps are available at:

- `Step 1 - Naive`
- `Step 2 - LDS tiling`
- `Step 3 - Register tiling`
- `Step 4 - Double buffering`
- `Step 5 - Vectorized loads`
- `Step 6 - Register pressure`
- `Step 7 - Generic kernel`

Note

All examples target 4096×4096 matrices in FP32 row-major layout and are validated against the identity-matrix test ($B = I \implies C = A$). They compile with `amdclang++ -O3 -std=c++17` and run on any ROCm-supported GPU architecture.

9.1 Prerequisites

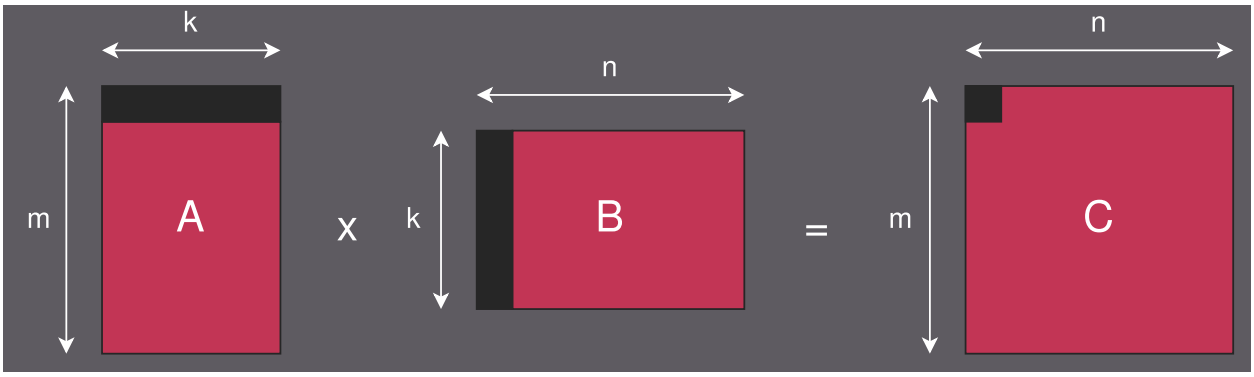
Before starting this tutorial, ensure the following are in place.

- ROCm installed and `amdclang++` available on `PATH`.
- Familiarity with the HIP execution model (grids, blocks, wavefronts) and its mapping to AMD GPU hardware (dispatches, workgroups, wavefronts).
- Application tracing and profiling using rocprofv3 installed for performance analysis.

> **Tip**
>
> Hardware performance counter availability varies by ROCm version, Linux kernel version, and system permissions. A counter that appears in `rocprofv3 --list-avail` might still return only zero values on a given system. If that happens, the counter is unavailable. Kernel duration (`End_Timestamp - Start_Timestamp` from `rocprofv3 --kernel-trace`) is always available and provides a reliable baseline across all steps and architectures.

9.2 GEMM fundamentals

A GEMM multiplies an $M \times K$ matrix $\boldsymbol{A}$ by a $K \times N$ matrix $\boldsymbol{B}$ to produce an $M \times N$ output matrix $\boldsymbol{C}$. Each element C_{ij} is the inner product of the *i*-th row of $\boldsymbol{A}$ and the *j*-th column of $\boldsymbol{B}$:



A CPU implementation applies three nested loops over *m*, *n*, and *k*, performing one multiply-accumulate per iteration - 2 × M × N × K scalar operations in total.

A GPU implementation is embarrassingly parallel across the output elements. A HIP kernel assigns one thread (or a small tile of threads) to each output element, eliminating the *m* and *n* loops entirely and leaving only the *k* reduction loop inside each thread. With $M \times N$ output elements and a modern AMD GPU fielding tens of thousands of concurrent threads, the full output matrix can be computed in a single dispatch - provided data can be supplied fast enough to keep the Compute Units (CUs) busy.

9.3 Background: the GEMM arithmetic intensity

For an $M \times K \times N$ GEMM the arithmetic intensity - floating-point operations per byte of DRAM traffic - is:

$$I = \frac{2 \cdot M \cdot N \cdot K}{\text{sizeof(float)} \cdot (M \cdot K + K \cdot N + M \cdot N)}$$

For $M = N = K = n$ this simplifies to $\frac{n}{6}$. With $n = 4096$ that gives roughly **683 FLOPs/byte**, far above the roofline ridge point of any current AMD GPU. GEMM is therefore **compute-bound** in principle - but only if data is supplied fast enough to keep the compute units busy. The naive kernel falls well below the roofline because it is *memory-bound in practice*: global memory latency stalls dominate (see *Roofline model* for background on roofline analysis and *performance bottlenecks*).

The optimization steps that follow progressively close the gap between actual and theoretical throughput by improving data reuse and instruction-level efficiency.

9.4 Step 1: Naive kernel

The naive kernel assigns one thread per output element. Each thread reads a full row of $\boldsymbol{A}$ and a full column of $\boldsymbol{B}$ directly from global memory.

While this maps naturally onto the GPU's parallel execution model, it produces severe **cache thrashing**. Consider that any element $A[i][k]$ is needed by *all N threads* that compute a different output column in the same row, and $B[k][j]$ is needed by all *M threads* that compute a different output row in the same column. With tens of thousands of threads in flight simultaneously, the working set far exceeds the L2 cache, so data that should be reused is evicted before the next thread requests it. The result is that total DRAM traffic is a large multiple of the minimum required bandwidth (`sizeof(float) * (M*K + K*N + M*N)`), and the kernel is firmly memory-bound despite GEMM's high arithmetic intensity in principle.

```
__global__ void matrix_multiply_naive(const float* __restrict__ A,
                                      const float* __restrict__ B,
                                      float* __restrict__ C,
                                      int m,
                                      int n,
                                      int k)
{
    const int col = blockIdx.x * blockDim.x + threadIdx.x;
    const int row = blockIdx.y * blockDim.y + threadIdx.y;

    if(row < m && col < n)
    {
        float sum = 0.0f;
        for(int i = 0; i < k; ++i)
        {
            sum += A[row * k + i] * B[i * n + col];
        }
        C[row * n + col] = sum;
    }
}
```

Launch configuration:

```
const dim3 block(BLOCK_SIZE, BLOCK_SIZE);
const dim3 grid((N + BLOCK_SIZE - 1) / BLOCK_SIZE,
                (M + BLOCK_SIZE - 1) / BLOCK_SIZE);
```

Compile and run:

```
amdclang++ -O3 -std=c++17 matrix_multiply_naive.hip -o mm_naive
./mm_naive
```

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./mm_naive
```

Kernel duration is `End_Timestamp - Start_Timestamp` (both in nanoseconds). The trace also captures `VGPR_Count` and `Scratch_Size` for each dispatch.

The `--kernel-trace` CSV is sufficient to establish the baseline: the naive kernel's `End_Timestamp - Start_Timestamp` will be the slowest of all steps because every global memory access is a cache miss.

9.5 Step 2: LDS tiling

The root cause of the naive kernel's cache thrashing is that all threads share a single transparent L2 cache with no way to guarantee that a loaded value stays resident until every thread that needs it has read it. AMD GPUs expose **Local Data Share (LDS)** - a low-latency, high-bandwidth on-chip memory that is explicitly managed by the programmer, functioning as a programmable L1 cache. Unlike CPU hardware caches, data placed in LDS stays there until the kernel explicitly overwrites or discards it.

The key insight is that every element of $\boldsymbol{A}$ is used by N threads (one per output column) and every element of $\boldsymbol{B}$ is used by M threads. Caching a `TILE_SIZE * TILE_SIZE` strip of $\boldsymbol{A}$ and $\boldsymbol{B}$ in LDS lets all `TILE_SIZE`^{2} threads in a block reuse that data without touching global memory again.

```
constexpr int TILE_SIZE = 16;
```

Shared memory allocation:

```
// Allocate two LDS tiles: one strip of A (row-major) and one of B (row-major).
// Each tile holds TILE_SIZE x TILE_SIZE float elements = 1 KiB.
__shared__ float tile_a[TILE_SIZE][TILE_SIZE];
__shared__ float tile_b[TILE_SIZE][TILE_SIZE];
```

Load phase (cooperative, one element per thread):

```
// Cooperative load: thread (tx, ty) loads one element of each tile.
//
// tile_a ← A[row][t*TILE_SIZE + tx]
//   Row comes from the block's output row; column strides across K.
const int a_col = t * TILE_SIZE + tx;
tile_a[ty][tx]  = (row < m && a_col < k) ? A[row * k + a_col] : 0.0f;

// tile_b ← B[t*TILE_SIZE + ty][col]
//   Row strides across K; column comes from the block's output column.
const int b_row = t * TILE_SIZE + ty;
tile_b[ty][tx]  = (b_row < k && col < n) ? B[b_row * n + col] : 0.0f;

// Ensure all threads have written their element before any thread reads.
__syncthreads();
```

Compute phase (inner product from LDS):

```
// Accumulate the partial dot product for this tile from LDS.
// Both tile_a and tile_b reside in LDS; no global memory is touched here.
for(int i = 0; i < TILE_SIZE; ++i)
{
    sum += tile_a[ty][i] * tile_b[i][tx];
}

// Ensure all threads have finished reading before the next iteration
// overwrites the tiles.
__syncthreads();
```

9.5.1 LDS bank conflict analysis

Moving data into LDS is only half the battle - *how* threads access that data determines whether the LDS delivers its full bandwidth. LDS is divided into independently addressable **banks**. When threads in the same cycle access different addresses that map to the same bank, the hardware must serialize those accesses. This is called a **bank conflict**, and it directly reduces the effective LDS bandwidth by the degree of the conflict (a *k*-way conflict takes *k* cycles instead of one).

Bank mapping is straightforward: consecutive 4-byte words are assigned to consecutive banks in round-robin order. For a 32-bank LDS, word at byte address `a` maps to bank `(a / 4) % 32`. Two threads accessing the *same* address are not in conflict - the hardware broadcasts the value to both.

The number of LDS banks varies across AMD GPU architectures:

Architecture	Banks	Entries per bank (4 bytes each)
CDNA	32	512
CDNA2	32	512
CDNA3	32	512
CDNA4	64	640
RDNA2	64	512
RDNA3	64	512
RDNA3.5	64	512
RDNA4	64	512

> **Note**
>
> On RDNA GPUs the 64 banks are sub-divided into two sets of 32 banks, each affiliated with a pair of SIMD32 units within the WGP. A wavefront executes on one SIMD32 and maps its accesses to the affiliated 32-bank set. In CU mode (`-mcumode`), a workgroup runs on a single CU and accesses one set of 32 banks; in WGP mode (the default) the workgroup spans the full WGP.

For this kernel's compute phase - an inner product over the K-strip:

```
sum += tile_a[ty][i] * tile_b[i][tx];
```

- the access pattern with `float` data is **inherently conflict-free

regardless of tile size**:

- `tile_a[ty][i]`: all threads in a wavefront that share the same `ty` read the *same address*. The hardware broadcasts the value - no conflict.
- `tile_b[i][tx]`: each thread has a unique `tx`, and because each `float` is exactly 4 bytes (= one bank slot), consecutive `tx` values always map to consecutive banks. No two threads in the same cycle can hit the same bank at a different address, regardless of how large `TILE_SIZE` is. The `i` loop iterates sequentially within each thread, so accesses to different rows of `tile_b` are never concurrent.

In other words, with FP32 data and the simple inner-product pattern of this step, bank conflicts are a non-issue. The bank mechanism is worth understanding now, however, because it becomes a real concern later.

Compile and run:

```
amdclang++ -O3 -std=c++17 matrix_multiply_lds.hip -o mm_lds
./mm_lds
```

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./mm_lds
```

Compare `End_Timestamp - Start_Timestamp` against the Step 1 baseline. To measure L2-to-High Bandwidth Memory (HBM) read traffic on CDNA GPUs:

```
# CDNA only
rocprofv3 --pmc TCP_TCC_READ_REQ_sum --output-format csv -- ./mm_lds
```

9.5.2 What to observe after this step

Compare the following counters against the Step 1 naive kernel baseline.

Counter	What to look for
Kernel duration (kernel-trace CSV)	`End_Timestamp - Start_Timestamp` should drop significantly versus the naive kernel, confirming that LDS data reuse is reducing global memory traffic.
`TCP_TCC_READ_REQ_` (CDNA)	CDNA only: reduction proportional to `TILE_SIZE` confirms fewer L2-to-HBM read requests.
`LDSBankConflict`	Should remain at or near zero for `TILE_SIZE=16` with FP32 data (confirms no bank conflicts).

9.6 Step 3: Register tiling

In the LDS kernel each thread computes exactly one output element, reading `TILE_SIZE` values from `tile_a` and `TILE_SIZE` values from `tile_b` for every K-strip. If instead each thread computes a `THREAD_TILE_M` × `THREAD_TILE_N` sub-tile in registers, it amortizes the LDS load cost across `THREAD_TILE_M` × `THREAD_TILE_N` outputs.

The outer product of a length-`THREAD_TILE_M` column fragment of A and a length-`THREAD_TILE_N` row fragment of B produces a full `THREAD_TILE_M` × `THREAD_TILE_N` block of C contributions using only `THREAD_TILE_M` + `THREAD_TILE_N` LDS reads instead of `THREAD_TILE_M` × `THREAD_TILE_N`.

Tile parameters:

```
constexpr int BLOCK_TILE_M  = 128; // Rows of C computed by one block
constexpr int BLOCK_TILE_N  = 128; // Columns of C computed by one block
constexpr int K_TILE_SIZE   = 16;  // K-strip width loaded into LDS each iteration
constexpr int THREAD_TILE_M = 8;   // Rows of C computed by one thread
constexpr int THREAD_TILE_N = 8;   // Columns of C computed by one thread
```

Transposed tile_b layout:

`tile_b` is stored transposed in LDS as `tile_b_T[BLOCK_TILE_N][K_TILE_SIZE]` (column-major for B). This layout choice is motivated by two concerns:

1. **Stride-1 reads during the outer-product compute phase**: In the outer-product pattern, each thread loads a fragment of `THREAD_TILE_N` consecutive elements from `tile_b_T` along the `ki` dimension. Because the inner dimension of `tile_b_T` is `K_TILE_SIZE`, consecutive `ki` values are adjacent in memory - stride-1 access. Without transposition (`tile_b[K_TILE_SIZE][BLOCK_TILE_N]`), the fragment load would stride across the large `BLOCK_TILE_N` dimension, producing scattered LDS reads.
2. **Bank-conflict safety for sub-4-byte types**: In Step 2's simple inner-product loop, each `float` occupies exactly one 4-byte LDS bank slot, so bank conflicts cannot arise regardless of tile dimensions. This property does not hold for smaller data types. When architecture-specific builtins introduce half-precision (FP16, 2 bytes) or quarter-precision (FP8, 1 byte) data in follow-up sections, multiple elements pack into a single 4-byte bank slot. If the row

stride of `tile_b` equals or is a multiple of the bank count, different threads can address different sub-word elements within the same bank slot, producing conflicts that are impossible with FP32. The transposed layout decouples the fragment access stride (`K_TILE_SIZE`) from the tile's outer dimension (`BLOCK_TILE_N`), avoiding this class of conflict regardless of element size or bank count.

LDS allocation:

```
// tile_a: row-major strip of A, shape [BLOCK_TILE_M][K_TILE_SIZE]
__shared__ float tile_a[BLOCK_TILE_M][K_TILE_SIZE];
// tile_b_T: column-major strip of B (transposed in LDS), shape [BLOCK_TILE_N][K_TILE_
↪SIZE]
// Storing transposed eliminates the bank-conflict risk that arises when
// TILE_SIZE equals the LDS bank count, and produces stride-1 reads during
// both the load phase and the compute phase.
__shared__ float tile_b_T[BLOCK_TILE_N][K_TILE_SIZE];
```

Cooperative tile load:

```
// --- Cooperative load of tile_a (BLOCK_TILE_M x K_TILE_SIZE) ---
// Each thread loads one or more elements by striding through the flat
// tile index space with step = num_threads.
{
    const int tile_elems = BLOCK_TILE_M * K_TILE_SIZE;
    for(int idx = tid; idx < tile_elems; idx += num_threads)
    {
        const int tile_row  = idx / K_TILE_SIZE;
        const int tile_col  = idx % K_TILE_SIZE;
        const int global_row = block_row_start + tile_row;
        const int global_col = t * K_TILE_SIZE + tile_col;
        tile_a[tile_row][tile_col] =
            (global_row < m && global_col < k)
                ? A[global_row * k + global_col]
                : 0.0f;
    }
}

// --- Cooperative load of tile_b_T (BLOCK_TILE_N x K_TILE_SIZE, transposed) ---
// We load B[b_row][b_col] into tile_b_T[b_col - block_col_start][b_row - t*K].
// Equivalently: thread loads the element at (col_within_tile, k_within_tile)
// of the transposed tile, reading B in row-major order.
{
    const int tile_elems = BLOCK_TILE_N * K_TILE_SIZE;
    for(int idx = tid; idx < tile_elems; idx += num_threads)
    {
        const int tile_col_idx = idx / K_TILE_SIZE; // column within block tile
        const int tile_k_idx   = idx % K_TILE_SIZE; // k index within strip
        const int global_row   = t * K_TILE_SIZE + tile_k_idx;
        const int global_col   = block_col_start + tile_col_idx;
        tile_b_T[tile_col_idx][tile_k_idx] =
            (global_row < k && global_col < n)
                ? B[global_row * n + global_col]
                : 0.0f;
    }
}
```

(continues on next page)

(continued from previous page)

```

__syncthreads();
```

Outer-product accumulation:

```
// Outer-product accumulation entirely in registers.
// For each k-index within the strip:
//   - Load a column fragment of tile_a into registers (a_frag).
//   - Load a row fragment of tile_b_T into registers (b_frag).
//   - Accumulate the TM x TN outer product into acc.
for(int ki = 0; ki < K_TILE_SIZE; ++ki)
{
    float a_frag[THREAD_TILE_M];
    #pragma unroll
    for(int i = 0; i < THREAD_TILE_M; ++i)
    {
        a_frag[i] = tile_a[thread_row + i][ki];
    }

    float b_frag[THREAD_TILE_N];
    #pragma unroll
    for(int j = 0; j < THREAD_TILE_N; ++j)
    {
        b_frag[j] = tile_b_T[thread_col + j][ki];
    }

    #pragma unroll
    for(int i = 0; i < THREAD_TILE_M; ++i)
    {
        #pragma unroll
        for(int j = 0; j < THREAD_TILE_N; ++j)
        {
            acc[i][j] += a_frag[i] * b_frag[j];
        }
    }
}

__syncthreads();
```

Write-back:

```
// Write the register accumulator to global memory.
// Each store is a coalesced write: threads with consecutive tx values write
// to consecutive columns (thread_col strides by THREAD_TILE_N = 8, but the
// inner j loop produces 8 consecutive output columns per thread, so adjacent
// threads in x cover adjacent 8-wide strips → effectively coalesced).
#pragma unroll
for(int i = 0; i < THREAD_TILE_M; ++i)
{
    #pragma unroll
    for(int j = 0; j < THREAD_TILE_N; ++j)
    {
```

(continues on next page)

(continued from previous page)

```
        const int out_row = block_row_start + thread_row + i;
        const int out_col = block_col_start + thread_col + j;
        if(out_row < m && out_col < n)
        {
            C[out_row * n + out_col] = acc[i][j];
        }
    }
}
```

Compile and run:

```
amdclang++ -O3 -std=c++17 matrix_multiply_register_tiling.hip -o mm_register_tiling
./mm_register_tiling
```

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./mm_register_tiling
```

For LDS and arithmetic instruction counts:

```
rocprofv3 --pmc SQ_INSTS_LDS --output-format csv -- ./mm_register_tiling
```

9.6.1 What to observe

Compare the following counters against the LDS tiling kernel from Step 2.

Counter	What to look for
`VALUInsts` (RDNA, CDNA, CDNA2) / `SQ_INSTS_VALU` (CDNA3, CDNA4)	Increase in VALU instructions per wave (more Fused Multiply-Accumulate (FMA) operations per LDS read).
`SQ_INSTS_LDS`	Reduction in LDS instructions per wave (`THREAD_TILE_M + THREAD_TILE_N` instead of `2 × THREAD_TILE_M × THREAD_TILE_N`).
`SQ_WAIT_INST_LDS`	Reduction in LDS stall cycles (register reuse hides LDS latency).

9.6.2 Tile parameter tuning guidance

The tile dimensions (`BLOCK_TILE_M`, `BLOCK_TILE_N`, `K_TILE_SIZE`, `THREAD_TILE_M`, `THREAD_TILE_N`) are not one-size-fits-all. Optimal values depend on several interacting constraints:

Matrix dimensions
: Tile sizes should evenly divide the matrix dimensions to avoid boundary handling overhead. Padding the matrices to a multiple of the tile size is common in production GEMM libraries.

LDS capacity
: Each workgroup allocates `BLOCK_TILE_M × K_TILE_SIZE` and `K_TILE_SIZE × BLOCK_TILE_N` floats in LDS. LDS capacity varies across AMD GPU architectures (typically 64–160 KiB per compute unit). Exceeding the LDS budget reduces occupancy by limiting how many workgroups can be resident simultaneously.

Register file
: Each thread holds a `THREAD_TILE_M × THREAD_TILE_N` accumulator array plus fragment temporaries. Larger thread tiles increase arithmetic intensity but consume more Vector General-Purpose Registers (VGPRs), reducing occupancy. Step 6 addresses this tradeoff directly.

Architecture-to-architecture variation

LDS bank counts, wavefront widths, VGPR file size, and L1/L2 cache line sizes differ across CDNA and RDNA families. Tile sizes that are optimal on a CDNA3 GPU might not be optimal on an RDNA4 GPU. Use `rocprofv3 --pmc` to measure LDS efficiency, VGPR usage, and occupancy on each target, and re-tune accordingly.

9.7 Step 4: Double buffering

Every iteration of the K-strip loop stalls at `__syncthreads()` waiting for LDS tile loads to complete before compute can begin. Software double buffering hides this latency by maintaining two pairs of LDS buffers - one pair for **A** tiles and one for **B** tiles - each with a *ping* and a *pong* slot. The next tile is loaded into the background slot while the current slot is being consumed.

Both buffering strategies are hidden behind a `TilePolicy` interface so that the kernel body is identical regardless of the chosen approach.

Policy interface overview:

Method	Responsibility
`prologue`	Load tile 0 into buffer 0 and synchronize (double-buffer only; no-op for single).
`prefetch`	Issue the load for the next tile into the background buffer.
`acquire`	Synchronize before compute (single-buffer: `__syncthreads()`; double: no-op).
`release`	Synchronize after compute (both: `__syncthreads()`).
`buf_idx`	Return which buffer to read for the current iteration.

Single-buffer policy (baseline - same logic as Step 3):

```
struct SingleBufferTilePolicy
{
    static constexpr int num_buffers = 1;

    // LDS holds one pair of tiles.
    struct SharedStorage
    {
        float tile_a[BLOCK_TILE_M][K_TILE_SIZE];
        float tile_b_T[BLOCK_TILE_N][K_TILE_SIZE];
    };

    // No preloading: the first prefetch() call loads tile 0.
    static __device__ void prologue(SharedStorage& /*smem*/,
                                    const float* /*A*/,
                                    const float* /*B*/,
                                    int /*block_row_start*/,
                                    int /*block_col_start*/,
                                    int /*m*/,
                                    int /*n*/,
                                    int /*k*/,
                                    int /*tid*/,
                                    int /*num_threads*/,
                                    int /*num_tiles*/)
    {}

    // Load tile `tile_idx - 1` (the CURRENT tile t) into the single buffer.
```

(continues on next page)

(continued from previous page)

```
    // The kernel calls prefetch(t + 1), so tile_idx = t + 1.  We load
    // tile t = tile_idx - 1 here; acquire() then __syncthreads() to ensure
    // it is visible before compute begins.
    static __device__ void prefetch(SharedStorage&  smem,
                                    const float*    A,
                                    const float*    B,
                                    int             tile_idx,
                                    int             block_row_start,
                                    int             block_col_start,
                                    int             m,
                                    int             n,
                                    int             k,
                                    int             tid,
                                    int             num_threads,
                                    int             /*num_tiles*/)
    {
        // tile_idx = t + 1, so the tile to load is t = tile_idx - 1.
        // The loop runs t in [0, num_tiles), so tile_idx in [1, num_tiles],
        // and load_idx in [0, num_tiles - 1] - always in range.
        const int load_idx = tile_idx - 1;

        const int tile_elems_a = BLOCK_TILE_M * K_TILE_SIZE;
        for(int idx = tid; idx < tile_elems_a; idx += num_threads)
        {
            const int tile_row   = idx / K_TILE_SIZE;
            const int tile_col   = idx % K_TILE_SIZE;
            const int global_row = block_row_start + tile_row;
            const int global_col = load_idx * K_TILE_SIZE + tile_col;
            smem.tile_a[tile_row][tile_col] =
                (global_row < m && global_col < k)
                    ? A[global_row * k + global_col]
                    : 0.0f;
        }

        const int tile_elems_b = BLOCK_TILE_N * K_TILE_SIZE;
        for(int idx = tid; idx < tile_elems_b; idx += num_threads)
        {
            const int tile_col_idx = idx / K_TILE_SIZE;
            const int tile_k_idx   = idx % K_TILE_SIZE;
            const int global_row   = load_idx * K_TILE_SIZE + tile_k_idx;
            const int global_col   = block_col_start + tile_col_idx;
            smem.tile_b_T[tile_col_idx][tile_k_idx] =
                (global_row < k && global_col < n)
                    ? B[global_row * n + global_col]
                    : 0.0f;
        }
    }

    // Wait for the prefetch just issued to complete.
    static __device__ void acquire(int /*tile_idx*/)
    {
        __syncthreads();
```

(continues on next page)

(continued from previous page)

```
    }

    // Protect the buffer: no thread may start the next prefetch until all
    // threads have finished reading the current tile.
    static __device__ void release(int /*tile_idx*/)
    {
        __syncthreads();
    }

    // Always return buffer index 0.
    static __device__ int buf_idx(int /*tile_idx*/)
    {
        return 0;
    }

    static __device__ const float* get_tile_a(const SharedStorage& smem, int /*buf*/)
    {
        return &smem.tile_a[0][0];
    }

    static __device__ const float* get_tile_b_T(const SharedStorage& smem, int /*buf*/
)
    {
        return &smem.tile_b_T[0][0];
    }
};
```

Software double-buffer policy:

```
struct SoftwareDoubleBufferTilePolicy
{
    static constexpr int num_buffers = 2;

    // LDS holds TWO pairs of tiles, indexed by [buf][...].
    struct SharedStorage
    {
        float tile_a[2][BLOCK_TILE_M][K_TILE_SIZE];
        float tile_b_T[2][BLOCK_TILE_N][K_TILE_SIZE];
    };

    // Load tile 0 into buf[0] and synchronise so that the first iteration
    // can call acquire() as a no-op.
    static __device__ void prologue(SharedStorage&  smem,
                                    const float*    A,
                                    const float*    B,
                                    int             block_row_start,
                                    int             block_col_start,
                                    int             m,
                                    int             n,
                                    int             k,
                                    int             tid,
                                    int             num_threads,
```

(continues on next page)

(continued from previous page)

```
                                    int             /*num_tiles*/)
{
    const int tile_elems_a = BLOCK_TILE_M * K_TILE_SIZE;
    for(int idx = tid; idx < tile_elems_a; idx += num_threads)
    {
        const int tile_row   = idx / K_TILE_SIZE;
        const int tile_col   = idx % K_TILE_SIZE;
        const int global_row = block_row_start + tile_row;
        const int global_col = tile_col; // tile 0: K offset = 0
        smem.tile_a[0][tile_row][tile_col] =
            (global_row < m && global_col < k)
                ? A[global_row * k + global_col]
                : 0.0f;
    }

    const int tile_elems_b = BLOCK_TILE_N * K_TILE_SIZE;
    for(int idx = tid; idx < tile_elems_b; idx += num_threads)
    {
        const int tile_col_idx = idx / K_TILE_SIZE;
        const int tile_k_idx   = idx % K_TILE_SIZE;
        const int global_row   = tile_k_idx; // tile 0: K offset = 0
        const int global_col   = block_col_start + tile_col_idx;
        smem.tile_b_T[0][tile_col_idx][tile_k_idx] =
            (global_row < k && global_col < n)
                ? B[global_row * n + global_col]
                : 0.0f;
    }

    // Ensure tile 0 is fully in LDS before the main loop begins.
    __syncthreads();
}

// Issue the load for tile `tile_idx` into the alternate buffer.
// This load runs concurrently with the compute on the current buffer.
static __device__ void prefetch(SharedStorage&  smem,
                                const float*    A,
                                const float*    B,
                                int             tile_idx,
                                int             block_row_start,
                                int             block_col_start,
                                int             m,
                                int             n,
                                int             k,
                                int             tid,
                                int             num_threads,
                                int             num_tiles)
{
    if(tile_idx >= num_tiles)
    {
        return;
    }

```

(continues on next page)

(continued from previous page)

```
        // Write into the buffer that is NOT currently being read.
        const int dst_buf = tile_idx % 2;

        const int tile_elems_a = BLOCK_TILE_M * K_TILE_SIZE;
        for(int idx = tid; idx < tile_elems_a; idx += num_threads)
        {
            const int tile_row   = idx / K_TILE_SIZE;
            const int tile_col   = idx % K_TILE_SIZE;
            const int global_row = block_row_start + tile_row;
            const int global_col = tile_idx * K_TILE_SIZE + tile_col;
            smem.tile_a[dst_buf][tile_row][tile_col] =
                (global_row < m && global_col < k)
                    ? A[global_row * k + global_col]
                    : 0.0f;
        }

        const int tile_elems_b = BLOCK_TILE_N * K_TILE_SIZE;
        for(int idx = tid; idx < tile_elems_b; idx += num_threads)
        {
            const int tile_col_idx = idx / K_TILE_SIZE;
            const int tile_k_idx   = idx % K_TILE_SIZE;
            const int global_row   = tile_idx * K_TILE_SIZE + tile_k_idx;
            const int global_col   = block_col_start + tile_col_idx;
            smem.tile_b_T[dst_buf][tile_col_idx][tile_k_idx] =
                (global_row < k && global_col < n)
                    ? B[global_row * n + global_col]
                    : 0.0f;
        }
        // No sync here: the load runs concurrently with compute on the other buffer.
    }

    // The current buffer is already filled (by prologue or the previous release).
    // No synchronisation is needed here.
    static __device__ void acquire(int /*tile_idx*/) {}

    // Single __syncthreads() serves dual purpose:
    //   - Ensures every thread has finished reading buf[tile_idx % 2]
    //     (so the next prefetch may safely overwrite it).
    //   - Ensures the prefetch issued at the top of this iteration
    //     (loading into buf[(tile_idx+1) % 2]) has completed.
    static __device__ void release(int /*tile_idx*/)
    {
        __syncthreads();
    }

    static __device__ int buf_idx(int tile_idx)
    {
        return tile_idx % 2;
    }

    static __device__ const float* get_tile_a(const SharedStorage& smem, int buf)
    {
```

(continues on next page)

(continued from previous page)

```
        return &smem.tile_a[buf][0][0];
    }

    static __device__ const float* get_tile_b_T(const SharedStorage& smem, int buf)
    {
        return &smem.tile_b_T[buf][0][0];
    }
};
```

Compile-time policy validation:

```
template<typename TilePolicy>
struct TilePolicyTraits
{
    static_assert(TilePolicy::num_buffers == 1 || TilePolicy::num_buffers == 2,
                  "TilePolicy::num_buffers must be 1 (single buffer) or 2 (double␣
↪buffer)");

    // SharedStorage must be a non-empty, trivially-destructible aggregate
    // that the compiler can place in __shared__ memory.
    static_assert(sizeof(typename TilePolicy::SharedStorage) > 0,
                  "TilePolicy::SharedStorage must have non-zero size");
    static_assert(std::is_trivially_destructible_v<typename TilePolicy::SharedStorage>
↪,
                  "TilePolicy::SharedStorage must be trivially destructible "
                  "(required for __shared__ variables)");
};
```

Unified kernel template:

```
template<typename TilePolicy>
__global__ void matrix_multiply_buffered(const float* __restrict__ A,
                                         const float* __restrict__ B,
                                         float* __restrict__ C,
                                         int m,
                                         int n,
                                         int k)
{
    // Instantiate the trait checker; all static_asserts fire at compile time.
    (void)TilePolicyTraits<TilePolicy>{};

    __shared__ typename TilePolicy::SharedStorage smem;

    const int tx          = threadIdx.x;
    const int ty          = threadIdx.y;
    const int tid         = ty * blockDim.x + tx;
    const int num_threads = blockDim.x * blockDim.y;

    const int thread_row      = ty * THREAD_TILE_M;
    const int thread_col      = tx * THREAD_TILE_N;
    const int block_row_start = blockIdx.y * BLOCK_TILE_M;
    const int block_col_start = blockIdx.x * BLOCK_TILE_N;
```

(continues on next page)

(continued from previous page)

```
    float acc[THREAD_TILE_M][THREAD_TILE_N] = {};

    const int num_tiles = (k + K_TILE_SIZE - 1) / K_TILE_SIZE;

    // [Sphinx prologue start]
    // Prologue: SingleBuffer → no-op; DoubleBuffer → preload tile 0 + sync.
    TilePolicy::prologue(smem, A, B, block_row_start, block_col_start,
                         m, n, k, tid, num_threads, num_tiles);
    // [Sphinx prologue end]

    for(int t = 0; t < num_tiles; ++t)
    {
        // [Sphinx prefetch acquire start]
        // Issue the load for the NEXT tile into the alternate buffer (or the
        // single buffer for SingleBufferTilePolicy).
        TilePolicy::prefetch(smem, A, B, t + 1, block_row_start, block_col_start,
                             m, n, k, tid, num_threads, num_tiles);

        // Ensure the CURRENT tile (tile t) is fully in LDS before reading it.
        // SingleBuffer: __syncthreads() after the load just issued above.
        // DoubleBuffer: no-op (tile t was already in LDS from the previous release).
        TilePolicy::acquire(t);
        // [Sphinx prefetch acquire end]

        // Outer-product accumulation from LDS (register tiling, same as Step 3).
        const float* ta   = TilePolicy::get_tile_a(smem, TilePolicy::buf_idx(t));
        const float* tb_T = TilePolicy::get_tile_b_T(smem, TilePolicy::buf_idx(t));

        for(int ki = 0; ki < K_TILE_SIZE; ++ki)
        {
            float a_frag[THREAD_TILE_M];
            #pragma unroll
            for(int i = 0; i < THREAD_TILE_M; ++i)
            {
                // tile_a is laid out as [BLOCK_TILE_M][K_TILE_SIZE]
                a_frag[i] = ta[(thread_row + i) * K_TILE_SIZE + ki];
            }

            float b_frag[THREAD_TILE_N];
            #pragma unroll
            for(int j = 0; j < THREAD_TILE_N; ++j)
            {
                // tile_b_T is laid out as [BLOCK_TILE_N][K_TILE_SIZE]
                b_frag[j] = tb_T[(thread_col + j) * K_TILE_SIZE + ki];
            }

            #pragma unroll
            for(int i = 0; i < THREAD_TILE_M; ++i)
            {
                #pragma unroll
                for(int j = 0; j < THREAD_TILE_N; ++j)
```

(continues on next page)

(continued from previous page)

```
                {
                    acc[i][j] += a_frag[i] * b_frag[j];
                }
            }
        }

        // [Sphinx release start]
        // SingleBuffer: second __syncthreads() - protects LDS before prefetch
        //               overwrites it in the next iteration.
        // DoubleBuffer: single __syncthreads() - ensures:
        //   (a) compute has finished reading buf[t%2], so the next prefetch
        //       may safely write into it.
        //   (b) the prefetch issued above (for tile t+1 into buf[(t+1)%2])
        //       has completed before the next iteration's acquire() returns.
        TilePolicy::release(t);
        // [Sphinx release end]
    }

    // Write accumulator to global C
    #pragma unroll
    for(int i = 0; i < THREAD_TILE_M; ++i)
    {
        #pragma unroll
        for(int j = 0; j < THREAD_TILE_N; ++j)
        {
            const int out_row = block_row_start + thread_row + i;
            const int out_col = block_col_start + thread_col + j;
            if(out_row < m && out_col < n)
            {
                C[out_row * n + out_col] = acc[i][j];
            }
        }
    }
}
```

Note

The LDS footprint doubles with software double buffering: two copies of the A and B tile buffers are needed instead of one. For the parameters in this example (`BLOCK_TILE_M = BLOCK_TILE_N = 128`, `K_TILE_SIZE = 16`):

- Single-buffer LDS: 128 × 16 × 4 × 2 = 16 KiB
- Double-buffer LDS: 16 KiB × 2 = 32 KiB

This is within the 64–160 KiB LDS budget on all supported architectures, but leaves less headroom for occupancy. Use `rocprofv3 --pmc MeanOccupancyPerCU` to verify occupancy does not drop when switching from single- to double-buffered policy.

Compile and run:

```
amdclang++ -O3 -std=c++17 matrix_multiply_double_buffer.hip -o mm_double_buffer
./mm_double_buffer
```

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./mm_double_buffer
```

To measure LDS stall cycles (RDNA3 and all CDNA):

```
# RDNA3 and all CDNA
rocprofv3 --pmc SQ_WAIT_INST_LDS --output-format csv -- ./mm_double_buffer
```

9.7.1 What to observe

Compare the following counters against the register tiling kernel from Step 3.

Counter	What to look for
`SQ_WAIT_INST_LDS`	Reduction in LDS stall cycles (load latency hidden by prefetch).
Kernel duration (kernel-trace CSV)	Should remain similar to Step 3 (prefetch hides latency but does not reduce total data fetched).
`TCP_TCC_READ_REQ_sum`	CDNA only: roughly constant vs Step 3 (same number of L2-to-HBM reads; only latency is hidden, not traffic).

9.8 Step 5: Vectorized loads

Each global memory load instruction in the tile-loading loop fetches one `float` per thread. Replacing it with a `float2` or `float4` load fetches 2 or 4 `float` values per instruction - the same total data moves through the cache hierarchy, but in fewer instructions. This reduces pressure on the VMEM instruction-issue pipeline and can improve overall throughput when instruction issue is the bottleneck rather than memory bandwidth.

To put this in context, consider how a single coalesced scalar load of a `float` maps to cache lines on each architecture family:

Architecture	Wavefront width	L1/L0 cache line size	Cache lines per coalesced scalar load
CDNA	64 lanes	64 bytes	64 × 4 B / 64 B = **4**
CDNA2	64 lanes	64 bytes	64 × 4 B / 64 B = **4**
CDNA3	64 lanes	128 bytes	64 × 4 B / 128 B = **2**
CDNA4	64 lanes	128 bytes	64 × 4 B / 128 B = **2**
RDNA2	32 lanes	128 bytes	32 × 4 B / 128 B = **1**
RDNA3	32 lanes	128 bytes	32 × 4 B / 128 B = **1**
RDNA3.5	32 lanes	128 bytes	32 × 4 B / 128 B = **1**
RDNA4	32 lanes	128 bytes	32 × 4 B / 128 B = **1**

On RDNA GPUs, a coalesced scalar `float` load already fills exactly one cache line - wider vector loads do not reduce cache traffic. On CDNA and CDNA2 a scalar load spans 4 cache lines. On CDNA3 and CDNA4, it spans 2. In all cases, vector loads do not change the number of cache lines accessed; they reduce the number of **instructions** the wavefront must issue to move the same amount of data.

Two vector widths are shown alongside the scalar baseline:

Width	HIP type	Alignment	Instruction
1	`float`	4 bytes	`buffer_load_dword` (one 4 B element per thread)
2	`float2`	8 bytes	`buffer_load_dwordx2` (two 4 B elements per thread)
4	`float4`	16 bytes	`buffer_load_dwordx4` (four 4 B elements per thread)

Vector type helper:

```
template<int Width>
struct VectorType;

template<>
struct VectorType<1>
{
    using type                      = float;
    static constexpr int width      = 1;
    static constexpr int byte_align = alignof(float); // 4 bytes
};

template<>
struct VectorType<2>
{
    using type                      = float2;
    static constexpr int width      = 2;
    static constexpr int byte_align = 8; // float2 must be 8-byte aligned
};

template<>
struct VectorType<4>
{
    using type                      = float4;
    static constexpr int width      = 4;
    static constexpr int byte_align = 16; // float4 must be 16-byte aligned
};
```

Vectorized load function:

```
template<int Width>
__device__ void vectorized_load(float* __restrict__       dst,
                                const float* __restrict__ src,
                                int                       tile_elems,
                                int                       tid,
                                int                       num_threads)
{
    using Vec = typename VectorType<Width>::type;

    static_assert(sizeof(Vec) == Width * sizeof(float),
                  "Vector type size mismatch");

    // Number of complete vector elements in the tile.
    // The static_asserts in the kernel guarantee tile_elems % Width == 0.
    const int vec_elems = tile_elems / Width;

    const Vec* src_vec = reinterpret_cast<const Vec*>(src);
    Vec*       dst_vec = reinterpret_cast<Vec*>(dst);

    for(int idx = tid; idx < vec_elems; idx += num_threads)
    {
        dst_vec[idx] = src_vec[idx];
    }
```

(continues on next page)

(continued from previous page)

```
}
```

Alignment requirements:

The `reinterpret_cast` in `vectorized_load` is only safe when the source pointer is aligned to `sizeof(Vec)` bytes. Two alignment guarantees apply:

1. `hipMalloc` returns a pointer aligned to at least 256 bytes, satisfying `float4` (16 bytes) for the base of any matrix.
2. Each row of A begins at offset `row × K × sizeof(float)`. For `float4` loads the row length `K` must be a multiple of 4. A compile-time `static_assert` enforces this for the tile parameters.

A runtime check is included in the example to catch misaligned user-supplied pointers:

```
void check_alignment(const void* ptr, int required_bytes, const char* name)
{
    const uintptr_t addr = reinterpret_cast<uintptr_t>(ptr);
    if(addr % static_cast<uintptr_t>(required_bytes) != 0)
    {
        std::cerr << "WARNING: " << name << " pointer " << ptr
                  << " is not " << required_bytes << "-byte aligned. "
                  << "Vectorised loads of width " << required_bytes
                  << " bytes will produce undefined behaviour.\n";
    }
}
```

Note

Only the A-tile load is vectorized because its elements are contiguous in global memory (row-major, stride 1). The B-tile is loaded column-by-column (stride N in global memory), which is not amenable to simple vector loads. Architecture-specific builtins (MFMA and WMMA), covered in follow-up sections, address this asymmetry.

9.8.1 Vectorized loads and smaller data types

For FP32, vectorized loads are a moderate optimization: they reduce instruction count, but a scalar load already fills cache lines well (see table above). The picture changes significantly for smaller data types. With FP16 (2 bytes) or FP8 (1 byte), a scalar load per thread no longer fills a full cache line:

Element type	Size	RDNA scalar load (32 lanes)	Cache line fill (128 B line)
FP32	4 bytes	32 × 4 = 128 B	100% (1 full line)
FP16	2 bytes	32 × 2 = 64 B	**50%** (half a line wasted)
FP8	1 byte	32 × 1 = 32 B	**25%** (three quarters wasted)

The wasted portion of each cache line is fetched from DRAM but never used - this is pure bandwidth overhead. A 2-wide vector load for FP16 or a 4-wide vector load for FP8 restores the full cache line fill. On CDNA3/CDNA4 (128 B cache line, 64-wide wavefronts), FP8 scalar loads similarly fill only half a line.

This is why the `TilePolicy` parameterizes the vector load width: the optimal width depends on both the element type and the target architecture. For the FP32 case in this tutorial, the benefit is modest, but for the low-precision builtins introduced in follow-up sections, vectorized loads become essential to avoid wasting memory bandwidth.

Compile and run:

```
amdclang++ -O3 -std=c++17 matrix_multiply_vectorized.hip -o mm_vectorized
./mm_vectorized
```

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./mm_vectorized
```

To compare VMEM instruction cycles across scalar and vector variants:

```
# RDNA (combined VMEM counter)
rocprofv3 --pmc SQ_INST_CYCLES_VMEM --output-format csv -- ./mm_vectorized

# CDNA (read counter only; GEMM tile loads are reads)
rocprofv3 --pmc SQ_INST_CYCLES_VMEM_RD --output-format csv -- ./mm_vectorized
```

9.8.2 What to observe

Compare the following counters against the double-buffered kernel from Step 4.

Counter	What to look for
`SQ_INST_CYCLES_VMEM` (all RDNA GPUs) / `SQ_INST_CYCLES_VMEM_RD` (all CDNA)	Reduction in VMEM instruction cycles (fewer instructions for the same total data). CDNA GPUs split this into `SQ_INST_CYCLES_VMEM_RD` (reads) and `SQ_INST_CYCLES_VMEM_WR` (writes); use the read counter for GEMM tile loads.
`TCP_TOTAL_CACHE_ACCESSES`	CDNA only: total L2 cache accesses should remain roughly constant (cache-line traffic does not change – only the instruction count does).

9.9 Step 6: Register pressure and occupancy

GPU occupancy - the ratio of active wavefronts to the hardware maximum - is set by the most constrained resource. For register-tiled GEMM kernels that resource is typically the **VGPR file**: each thread holds `THREAD_TILE_M` × `THREAD_TILE_N` accumulator registers plus fragment arrays, and the compiler might allocate additional temporaries.

Higher register usage means fewer concurrent wavefronts per CU, which reduces the GPU's ability to hide memory latency through wavefront switching. Conversely, forcibly reducing register usage can increase register spilling to scratch memory (a slow VGPR-to-DRAM path), degrading performance.

Three kernel variants illustrate the tradeoff:

No annotation (compiler decides freely):

```
__global__ void matrix_multiply_no_hint(const float* __restrict__ A,
                                        const float* __restrict__ B,
                                        float* __restrict__ C,
                                        int m,
                                        int n,
                                        int k)
{
    __shared__ float tile_a[BLOCK_TILE_M][K_TILE_SIZE];
    __shared__ float tile_b_T[BLOCK_TILE_N][K_TILE_SIZE];
    gemm_body(A, B, C, m, n, k, tile_a, tile_b_T);
}
```

``__launch_bounds__``:

```
constexpr int MIN_WAVES_PER_EU = 2; // tune this with rocprofv3

__global__
__launch_bounds__(BLOCK_THREADS, MIN_WAVES_PER_EU)
void matrix_multiply_launch_bounds(const float* __restrict__ A,
                                   const float* __restrict__ B,
                                   float* __restrict__ C,
                                   int m,
                                   int n,
                                   int k)
{
    __shared__ float tile_a[BLOCK_TILE_M][K_TILE_SIZE];
    __shared__ float tile_b_T[BLOCK_TILE_N][K_TILE_SIZE];
    gemm_body(A, B, C, m, n, k, tile_a, tile_b_T);
}
```

The two-argument form `__launch_bounds__(max_threads, min_waves_per_eu)` tells the compiler to allocate VGPRs such that at least `min_waves_per_eu` wavefronts can be resident per EU simultaneously, given `max_threads / warpSize` wavefronts per block.

``[[clang::amdgpu_waves_per_eu]]``:

```
// tune WAVES_PER_EU_MIN and WAVES_PER_EU_MAX with rocprofv3 --kernel-trace
// (compare VGPR_Count across variants).
constexpr int WAVES_PER_EU_MIN = 2;
constexpr int WAVES_PER_EU_MAX = 4;

__global__ void matrix_multiply_waves_per_eu
[[clang::amdgpu_waves_per_eu(WAVES_PER_EU_MIN, WAVES_PER_EU_MAX)]]
(const float* __restrict__ A,
 const float* __restrict__ B,
 float* __restrict__ C,
 int m,
 int n,
 int k)
{
    __shared__ float tile_a[BLOCK_TILE_M][K_TILE_SIZE];
    __shared__ float tile_b_T[BLOCK_TILE_N][K_TILE_SIZE];
    gemm_body(A, B, C, m, n, k, tile_a, tile_b_T);
}
```

This Clang attribute directly instructs the backend to target a wavefront occupancy in the range `[min, max]` per EU.

Compile and run:

```
amdclang++ -O3 -std=c++17 matrix_multiply_launch_bounds.hip -o mm_launch_bounds
./mm_launch_bounds
```

9.9.1 Finding optimal values with rocprofv3

1. Collect static kernel metadata for all three variants:

```
rocprofv3 --kernel-trace --output-format csv -- ./mm_launch_bounds
```

In the resulting CSV, compare the `VGPR_Count` and `Scratch_Size` columns across the three kernels.

2. Collect dynamic occupancy counters:

```
rocprofv3 --pmc SQ_WAVES_sum MeanOccupancyPerCU SQ_WAIT_INST_LDS --output-format␣
↪csv -- ./mm_launch_bounds

# if SQ_WAIT_INST_LDS not available
rocprofv3 --pmc SQ_WAVES_sum MeanOccupancyPerCU --output-format csv -- ./mm_
↪launch_bounds

# if MeanOccupancyPerCU and SQ_WAVES_sum not available
rocprofv3 --pmc SQ_LEVEL_WAVES SQ_WAIT_INST_LDS --output-format csv -- ./mm_
↪launch_bounds
```

3. Open both CSVs and note the values of:
   - **VGPR_Count** (vector registers allocated per thread, from the kernel-trace CSV)
   - **Scratch_Size** (>0 means VGPRs are spilling to DRAM - avoid this)
   - **MeanOccupancyPerCU** or **SQ_LEVEL_WAVES**: mean active wavefronts per CU, from the PMC CSV
4. Calculate the theoretical maximum occupancy from `VGPR_Count` using the formula for your architecture (available in the AMD ISA reference).
5. If the compiler allocated more VGPRs than necessary and occupancy is below the target, add `__launch_bounds__` with a `min_waves_per_eu` that reflects the desired occupancy.
6. Re-profile and re-check `Scratch_Size` - if it increases significantly, the compiler was forced to spill and the constraint is too aggressive.

Note

The numeric values `MIN_WAVES_PER_EU = 2` and `WAVES_PER_EU_MAX = 4` in the example are illustrative. Optimal values depend on the target GPU and the exact kernel register usage shown in the `rocprofv3 --kernel-trace` CSV. Always re-profile with `rocprofv3` after applying the annotation to confirm that occupancy improves without introducing scratch-memory spilling.

9.9.2 What to observe

Compare the following counters across the three kernel variants (no annotation, `__launch_bounds__`, and `[[clang::amdgpu_waves_per_eu]]`).

Counter / column	What to look for
`VGPR_Count` (kernel-trace CSV)	From `--kernel-trace` CSV: should decrease after adding the annotation.
`Scratch_Size` (kernel-trace CSV)	From `--kernel-trace` CSV: must remain zero (non-zero means register spilling to DRAM).
`MeanOccupancyPerCU` + `SQ_WAVES_sum` / `SQ_LEVEL_WAVES`	From `--pmc` CSV: mean active wavefronts per CU; should rise with fewer VGPRs.
`SQ_WAIT_INST_LDS`	LDS stall cycles (fall when more wavefronts are resident).

9.10 Step 7: Generic kernel

By this point the kernel is already a strong general-purpose implementation: it uses LDS tiling to exploit data reuse, register tiling to maximize arithmetic intensity, software double buffering to hide load latency, and vectorized loads to reduce memory transaction overhead. For many workloads this is sufficient.

Squeezing out the last few percent of throughput, however, requires architecture-specific matrix-multiply instructions: Matrix Fused Multiply-Accumulate (MFMA) on CDNA GPUs and Wave Matrix Multiply-Accumulate (WMMA) on RDNA3 and RDNA4. These instructions perform a small matrix multiply directly in hardware and deliver substantially higher FLOP/s than an equivalent sequence of scalar FMAs.

Steps 1–6 produced a well-optimized scalar GEMM kernel, but repeating the same work for each architecture-specific instruction set - MFMA on CDNA, WMMA on RDNA3, the relaxed WMMA variant on RDNA4 - would mean maintaining several near-identical copies of the kernel with only the inner computation swapped out. Any future improvement (a new tiling strategy, a wider vector load, a different buffering depth) would have to be applied to every copy independently.

The goal of this step is to factor the kernel so that the **optimized orchestration is written once** and architecture-specific builtins can be **dropped in later as a policy**, without touching any kernel code.

The insight from the preceding steps is that all the work decomposes into exactly two independent concerns:

1. **Data movement** (`TilePolicy`) - tile shape, LDS layout, vector load width, and buffering strategy. These optimizations are identical regardless of which instruction is used to compute the output; a `float4` load into a double-buffered LDS tile is just as beneficial whether the inner loop uses scalar FMAs or MFMA.
2. **Arithmetic** (`ComputePolicy`) - how a thread's register fragment is loaded from LDS and how the output accumulator is updated. This is the only part that differs between scalar code and architecture-specific builtins.

Following the principle of *lifting* an algorithm into its most general form, these two responsibilities are encapsulated in two orthogonal *policy classes*: `TilePolicy` and `ComputePolicy`. A single kernel template `matrix_multiply_generic<TilePolicy, ComputePolicy>` orchestrates the common control flow while delegating all architecture-specific details to the policies.

The kernel presented here uses `ScalarFMAPolicy` as the `ComputePolicy`. It already incorporates all the optimizations from the preceding steps: LDS tiling, register tiling, software double buffering, and vectorized loads. When an MFMA or WMMA `ComputePolicy` is provided in one of the architecture-specific builtins chapters, the same data-movement infrastructure and the same kernel orchestration are reused unchanged - only the inner arithmetic changes.

9.10.1 Policy interfaces

Both interfaces are documented in full in `matrix_multiply_generic.hip`. The key design decisions are:

TilePolicy interface summary:

Requirement	Rationale
`num_buffers = 1 | 2`	Governs LDS layout (one or two buffer pairs) and sync strategy.
`block_tile_m`, `block_tile_n`, `k_tile_size`	Tile shape needed by the kernel to compute grid dimensions.
`SharedStorage` (trivially destructible)	Placed in `__shared__`; must not require a destructor call.
`prologue`, `prefetch`, `acquire`, `release`	Data movement hooks; see below.

ComputePolicy interface summary:

Requirement	Rationale
`thread_tile_m`, `thread_tile_n`	Output sub-tile per thread; kernel derives block dimensions from these.
`effective_lanes`	Unique output lanes per wavefront.
`thread_tile_offset`, `tid`, `lane_id`, `*row`, `*col`	Architecture-aware thread-to-output mapping.
`elem_a`, `elem_b`	Element types of the register fragments (for example, `float` or `__half`); the kernel loop is fully templated on these.
`k_step`	Number of k-indices consumed per `mma()` call (1 for scalar FMA; higher values for builtins that process multiple k-indices per call); the kernel loop advances `ki` by this amount.
`load_a`, `load_b`, `mma`, `store_c`	Fragment load, multiply-accumulate, and write-back.

9.10.1.1 Data type scope: policy coverage

`elem_a` and `elem_b` parameterize the *register fragment* type and are already fully wired through the kernel loop. A future `ComputePolicy` can set `elem_a = __half` and the float-to-half conversion happens entirely inside `load_a` / `load_b` - the kernel body is untouched.

However, two things are not yet parameterized and are hardcoded to `float` in this file:

- The **LDS storage type** - `SharedStorage` in every `TilePolicy` holds `float` arrays, and the cooperative load helpers write `float` into LDS.
- The **global memory pointer type** - the kernel signature takes `const float* A`, `const float* B`, `float* C`.

This means two distinct cases arise when introducing builtins in follow-up sections:

Case	Example	What needs to change
FP32 in memory, low-precision fragments	Global `float` → LDS `float` → `__half` fragment for MFMA	Only `ComputePolicy::load_a` and `load_b` (conversion in registers). `TilePolicy` and the kernel signature are unchanged.
Low-precision in memory	Global `__half` → LDS `__half` → `__half` fragment	`TilePolicy` needs an `InputT` template parameter so `SharedStorage` and the cooperative load helpers use `InputT` instead of `float`. The kernel signature changes from `const float*` to `const InputT*`.

The global memory and LDS remain `float` throughout this tutorial, so only Case 1 applies. The `DirectLoadTilePolicy` *below* is a Case 1 example at the `TilePolicy` level; a `ComputePolicy` that converts to FP16 in its `load_a` / `load_b` would be a Case 1 example at the compute level. Case 2 is left as an extension for architecture-specific follow-up sections that operate on native FP16 or FP8 input matrices.

9.10.2 Compile-time validation

Both policy interfaces are validated with C++17 `static_assert` traits:

TilePolicy traits:

```
template<typename TilePolicy>
struct TilePolicyTraits
{
```

(continues on next page)

(continued from previous page)

```
    // num_buffers must be 1 or 2
    static_assert(TilePolicy::num_buffers == 1 || TilePolicy::num_buffers == 2,
                  "TilePolicy::num_buffers must be 1 (single) or 2 (double-buffer)");

    // Tile dimensions must be positive multiples of k_tile_size
    static_assert(TilePolicy::block_tile_m > 0,
                  "TilePolicy::block_tile_m must be a positive integer");
    static_assert(TilePolicy::block_tile_n > 0,
                  "TilePolicy::block_tile_n must be a positive integer");
    static_assert(TilePolicy::k_tile_size > 0,
                  "TilePolicy::k_tile_size must be a positive integer");

    // SharedStorage must be trivially destructible (no LDS destructor call)
    static_assert(std::is_trivially_destructible_v<typename TilePolicy::SharedStorage>
↪,
                  "TilePolicy::SharedStorage must be trivially destructible");

    // SharedStorage must actually occupy LDS
    static_assert(sizeof(typename TilePolicy::SharedStorage) > 0,
                  "TilePolicy::SharedStorage must have non-zero size");
};
```

ComputePolicy traits:

```
template<typename ComputePolicy>
struct ComputePolicyTraits
{
    static_assert(ComputePolicy::thread_tile_m > 0,
                  "ComputePolicy::thread_tile_m must be a positive integer");
    static_assert(ComputePolicy::thread_tile_n > 0,
                  "ComputePolicy::thread_tile_n must be a positive integer");
    static_assert(ComputePolicy::effective_lanes > 0,
                  "ComputePolicy::effective_lanes must be a positive integer");
    static_assert(ComputePolicy::k_step > 0,
                  "ComputePolicy::k_step must be a positive integer");

    // Accumulator must be trivially constructible (no runtime init in registers)
    static_assert(std::is_trivially_constructible_v<typename␣
↪ComputePolicy::Accumulator>,
                  "ComputePolicy::Accumulator must be trivially constructible");

    // Fragment element types must have a well-defined size (covers both
    // scalar arithmetic types like float and vector types like half2).
    static_assert(sizeof(typename ComputePolicy::elem_a) > 0,
                  "ComputePolicy::elem_a must have non-zero size");
    static_assert(sizeof(typename ComputePolicy::elem_b) > 0,
                  "ComputePolicy::elem_b must have non-zero size");
};
```

9.10.2.1 C++17 versus C++20: concept syntax

The `static_assert` approach above is portable to C++17 but requires the traits struct to be instantiated explicitly, and error messages appear at the trait instantiation site. C++20 `requires` clauses provide a more ergonomic alternative:

```
template<typename TilePolicy>
concept TilePolicyConcept = requires
{
    requires TilePolicy::num_buffers == 1 || TilePolicy::num_buffers == 2;
    requires TilePolicy::block_tile_m > 0;
    requires TilePolicy::block_tile_n > 0;
    requires TilePolicy::k_tile_size > 0;
    requires std::is_trivially_destructible_v<typename TilePolicy::SharedStorage>;
    requires sizeof(typename TilePolicy::SharedStorage) > 0;
};

// The kernel declaration then becomes self-documenting:
template<TilePolicyConcept TilePolicy, typename ComputePolicy>
__global__ void matrix_multiply_generic(/* … */);
//
// Constraint violations are reported at the point of the template
// instantiation with a clear "constraint not satisfied" diagnostic rather than
// a cryptic static_assert failure inside a trait struct.
```

With C++20, the constraint is expressed directly in the function signature and violations are reported at the point of the invalid template instantiation with a clear “constraint not satisfied” diagnostic.

9.10.3 Concrete policies

ScalarFMAPolicy - portable scalar FP32 outer-product (no builtins):

```
template<int ThreadTileM_, int ThreadTileN_, int BlockDimX_>
struct ScalarFMAPolicy
{
    static constexpr int thread_tile_m  = ThreadTileM_;
    static constexpr int thread_tile_n  = ThreadTileN_;
    // All lanes in a wavefront compute unique output elements (no mirroring).
    static constexpr int effective_lanes = 64; // AMD wavefront size
    static constexpr int k_step         = 1;  // one scalar FMA per ki iteration

    using elem_a = float;
    using elem_b = float;

    struct Accumulator
    {
        float data[ThreadTileM_][ThreadTileN_];
    };

    __device__ static void zero(Accumulator& acc)
    {
        #pragma unroll
        for(int i = 0; i < ThreadTileM_; ++i)
            #pragma unroll
            for(int j = 0; j < ThreadTileN_; ++j)
```

(continues on next page)

(continued from previous page)

```
                acc.data[i][j] = 0.0f;
    }

    // Map thread id to the top-left corner of its output sub-tile.
    // lane_id is accepted but unused (interface forward-compatibility).
    __device__ static void thread_tile_offset(int tid,
                                              int /*lane_id*/,
                                              int* thread_row,
                                              int* thread_col)
    {
        const int tx = tid % BlockDimX_;
        const int ty = tid / BlockDimX_;
        *thread_row  = ty * ThreadTileM_;
        *thread_col  = tx * ThreadTileN_;
    }

    // Load a column fragment of tile_a into registers.
    // tile_a is stored row-major: tile_a[row][ki] = ptr[row * k_tile_size + ki]
    __device__ static void load_a(const float* tile_a_ptr,
                                  int          tile_a_row,
                                  int          ki,
                                  int          k_tile_size,
                                  int          /*lane_id*/,
                                  elem_a (&frag)[ThreadTileM_])
    {
        #pragma unroll
        for(int i = 0; i < ThreadTileM_; ++i)
            frag[i] = tile_a_ptr[(tile_a_row + i) * k_tile_size + ki];
    }

    // Load a row fragment of tile_b_T into registers.
    // tile_b_T is stored column-major: tile_b_T[col][ki] = ptr[col * k_tile_size +
↪ki]
    __device__ static void load_b(const float* tile_b_T_ptr,
                                  int          tile_b_col,
                                  int          ki,
                                  int          k_tile_size,
                                  int          /*lane_id*/,
                                  elem_b (&frag)[ThreadTileN_])
    {
        #pragma unroll
        for(int j = 0; j < ThreadTileN_; ++j)
            frag[j] = tile_b_T_ptr[(tile_b_col + j) * k_tile_size + ki];
    }

    // Outer-product FMA: acc += a_frag ⊗ b_frag
    __device__ static void mma(Accumulator&       acc,
                               const elem_a (&a_frag)[ThreadTileM_],
                               const elem_b (&b_frag)[ThreadTileN_])
    {
        #pragma unroll
        for(int i = 0; i < ThreadTileM_; ++i)
```

(continues on next page)

(continued from previous page)

```
            #pragma unroll
            for(int j = 0; j < ThreadTileN_; ++j)
                acc.data[i][j] += a_frag[i] * b_frag[j];
    }

    // Write accumulator to global memory.
    // lane_id accepted but unused; all lanes write their unique sub-tile.
    __device__ static void store_c(const Accumulator& acc,
                                   float*             C,
                                   int                out_row_base,
                                   int                out_col_base,
                                   int                m,
                                   int                n,
                                   int                /*lane_id*/)
    {
        #pragma unroll
        for(int i = 0; i < ThreadTileM_; ++i)
            #pragma unroll
            for(int j = 0; j < ThreadTileN_; ++j)
            {
                const int r = out_row_base + i;
                const int c = out_col_base + j;
                if(r < m && c < n)
                    C[r * n + c] = acc.data[i][j];
            }
    }
};
```

SingleBufferTilePolicy - single LDS buffer pair, equivalent to Step 3:

```
template<int BlockTileM_, int BlockTileN_, int KTileSize_>
struct SingleBufferTilePolicy
{
    static constexpr int num_buffers  = 1;
    static constexpr int block_tile_m = BlockTileM_;
    static constexpr int block_tile_n = BlockTileN_;
    static constexpr int k_tile_size  = KTileSize_;

    struct SharedStorage
    {
        float tile_a[BlockTileM_][KTileSize_];
        float tile_b_T[BlockTileN_][KTileSize_];
    };

    __device__ static void prologue(SharedStorage& /*smem*/,
                                    const float* /*A*/, const float* /*B*/,
                                    int /*block_row_start*/, int /*block_col_start*/,
                                    int /*m*/, int /*n*/, int /*k*/,
                                    int /*tid*/, int /*num_threads*/)
    {
        // No prologue needed for single-buffer; tile is loaded at the top of
        // each loop iteration in prefetch().
```

(continues on next page)

(continued from previous page)

```
    }

    __device__ static void prefetch(SharedStorage& smem,
                                    const float* A, const float* B,
                                    int          t_next,
                                    int block_row_start, int block_col_start,
                                    int m, int n, int k,
                                    int tid, int num_threads)
    {
        // For single-buffer we load the *current* tile (t_next is actually t
        // from the caller's perspective - see kernel loop).
        // Rename for clarity: t_next here is the index we want to load.
        const int t = t_next;
        cooperative_load_tile_a(&smem.tile_a[0][0], A, block_row_start,
                                block_tile_m, t, k_tile_size, m, k, tid, num_threads);
        cooperative_load_tile_b_T(&smem.tile_b_T[0][0], B, block_col_start,
                                  block_tile_n, t, k_tile_size, k, n, tid, num_
↪threads);
    }

    // After prefetch: wait for the load to complete before compute.
    __device__ static void acquire(int /*t*/) { __syncthreads(); }

    // After compute: wait for all threads to finish reading before the next
    // iteration overwrites the tile.
    __device__ static void release(int /*t*/) { __syncthreads(); }

    __device__ static int buf_idx(int /*t*/) { return 0; }

    __device__ static float* get_tile_a(SharedStorage& smem, int /*buf*/)
    {
        return &smem.tile_a[0][0];
    }
    __device__ static float* get_tile_b_T(SharedStorage& smem, int /*buf*/)
    {
        return &smem.tile_b_T[0][0];
    }
};
```

SoftwareDoubleBufferTilePolicy - ping-pong LDS buffers, equivalent to Step 4:

```
template<int BlockTileM_, int BlockTileN_, int KTileSize_>
struct SoftwareDoubleBufferTilePolicy
{
    static constexpr int num_buffers  = 2;
    static constexpr int block_tile_m = BlockTileM_;
    static constexpr int block_tile_n = BlockTileN_;
    static constexpr int k_tile_size  = KTileSize_;

    struct SharedStorage
    {
        float tile_a[2][BlockTileM_][KTileSize_];
```

(continues on next page)

(continued from previous page)

```
        float tile_b_T[2][BlockTileN_][KTileSize_];
    };

    // Prologue: load tile 0 into buffer 0 and synchronise so compute can
    // begin immediately at the start of the first iteration.
    __device__ static void prologue(SharedStorage& smem,
                                    const float* A, const float* B,
                                    int block_row_start, int block_col_start,
                                    int m, int n, int k,
                                    int tid, int num_threads)
    {
        cooperative_load_tile_a(&smem.tile_a[0][0][0], A, block_row_start,
                                block_tile_m, 0, k_tile_size, m, k, tid, num_threads);
        cooperative_load_tile_b_T(&smem.tile_b_T[0][0][0], B, block_col_start,
                                  block_tile_n, 0, k_tile_size, k, n, tid, num_
↪threads);
        __syncthreads(); // tile 0 visible to all threads before loop entry
    }

    // Prefetch tile t_next into the background buffer (1 - current) while
    // the current buffer is in use for compute.  No sync here - the sync that
    // ends the previous compute phase (release) also covers this load because
    // the background buffer is not touched until two iterations later.
    __device__ static void prefetch(SharedStorage& smem,
                                    const float* A, const float* B,
                                    int          t_next,
                                    int block_row_start, int block_col_start,
                                    int m, int n, int k,
                                    int tid, int num_threads)
    {
        if(t_next * k_tile_size < k) // guard: do not load beyond K
        {
            const int buf = t_next % 2; // background buffer index
            cooperative_load_tile_a(&smem.tile_a[buf][0][0], A, block_row_start,
                                    block_tile_m, t_next, k_tile_size, m, k, tid, num_
↪threads);
            cooperative_load_tile_b_T(&smem.tile_b_T[buf][0][0], B, block_col_start,
                                      block_tile_n, t_next, k_tile_size, k, n, tid,␣
↪num_threads);
        }
    }

    // acquire: the prologue already placed a barrier; nothing needed here.
    // The previous iteration's release() synced both the completed compute
    // and the in-flight prefetch into the background buffer.
    __device__ static void acquire(int /*t*/) {}

    // release: one __syncthreads covers two things:
    //   (a) all threads finished computing from the current buffer, and
    //   (b) all threads finished writing the prefetched tile into the
    //       background buffer (the load issued in prefetch() above).
    // Both must be visible before the next iteration swaps buffers.
```

(continues on next page)

(continued from previous page)

```
    __device__ static void release(int /*t*/) { __syncthreads(); }

    __device__ static int buf_idx(int t) { return t % 2; }

    __device__ static float* get_tile_a(SharedStorage& smem, int buf)
    {
        return &smem.tile_a[buf][0][0];
    }
    __device__ static float* get_tile_b_T(SharedStorage& smem, int buf)
    {
        return &smem.tile_b_T[buf][0][0];
    }
};
```

9.10.4 Generic kernel template

```
template<typename TilePolicy, typename ComputePolicy>
__global__ void
    __launch_bounds__(TilePolicy::block_tile_m / ComputePolicy::thread_tile_m
                      * TilePolicy::block_tile_n / ComputePolicy::thread_tile_n)
    matrix_multiply_generic(const float* __restrict__ A,
                            const float* __restrict__ B,
                            float* __restrict__       C,
                            int                       m,
                            int                       n,
                            int                       k)
{
    // Instantiate TilePolicyTraits/ComputePolicyTraits to trigger static_assert
    // checks at the point where this kernel is instantiated.
    (void)TilePolicyTraits<TilePolicy>{};
    (void)ComputePolicyTraits<ComputePolicy>{};

    // [Sphinx gemm kernel smem start]
    __shared__ typename TilePolicy::SharedStorage smem;
    // [Sphinx gemm kernel smem end]

    const int tid         = threadIdx.y * blockDim.x + threadIdx.x;
    const int lane_id     = tid % 64; // wavefront lane index (AMD: 64-wide)
    const int num_threads = blockDim.x * blockDim.y;

    // Top-left corner of the block tile in global C
    const int block_row_start = blockIdx.y * TilePolicy::block_tile_m;
    const int block_col_start = blockIdx.x * TilePolicy::block_tile_n;

    // Thread's output sub-tile offset within the block tile
    int thread_row = 0, thread_col = 0;
    ComputePolicy::thread_tile_offset(tid, lane_id, &thread_row, &thread_col);

    // Initialise accumulator to zero
    typename ComputePolicy::Accumulator acc;
    ComputePolicy::zero(acc);
```

(continues on next page)

(continued from previous page)

```
    const int num_tiles = (k + TilePolicy::k_tile_size - 1) / TilePolicy::k_tile_size;

    // [Sphinx gemm kernel prologue start]
    // Double-buffer prologue: load tile 0 for policies that support it.
    // For SingleBufferTilePolicy this is a no-op.
    TilePolicy::prologue(smem, A, B,
                         block_row_start, block_col_start,
                         m, n, k, tid, num_threads);
    // [Sphinx gemm kernel prologue end]

    // [Sphinx gemm kernel loop start]
    for(int t = 0; t < num_tiles; ++t)
    {
        // --- Prefetch next tile (or load current tile for single-buffer) ---
        // For SoftwareDoubleBufferTilePolicy: issues the load for tile (t+1)
        //   into the background buffer while we compute from the foreground.
        // For SingleBufferTilePolicy: loads tile t (t_next == t here).
        TilePolicy::prefetch(smem, A, B, t + (TilePolicy::num_buffers == 1 ? 0 : 1),
                             block_row_start, block_col_start,
                             m, n, k, tid, num_threads);

        // --- Acquire: synchronise before compute ---
        // Single-buffer:  __syncthreads() after load
        // Double-buffer:  no-op (previous release() already synced)
        TilePolicy::acquire(t);

        // --- Compute: outer-product accumulation from the active buffer ---
        const int buf = TilePolicy::buf_idx(t);
        const float* tile_a_ptr   = TilePolicy::get_tile_a(smem, buf);
        const float* tile_b_T_ptr = TilePolicy::get_tile_b_T(smem, buf);

        // k_step: number of k-indices consumed per mma() call.
        // ScalarFMAPolicy uses k_step=1 (one FMA per call); intrinsic-based
        // policies can use higher values (e.g. k_step=2 for dot-product ops).
        static_assert(TilePolicy::k_tile_size % ComputePolicy::k_step == 0,
                      "k_tile_size must be divisible by ComputePolicy::k_step");

        for(int ki = 0; ki < TilePolicy::k_tile_size; ki += ComputePolicy::k_step)
        {
            typename ComputePolicy::elem_a a_frag[ComputePolicy::thread_tile_m];
            typename ComputePolicy::elem_b b_frag[ComputePolicy::thread_tile_n];

            ComputePolicy::load_a(tile_a_ptr, thread_row, ki,
                                  TilePolicy::k_tile_size, lane_id, a_frag);
            ComputePolicy::load_b(tile_b_T_ptr, thread_col, ki,
                                  TilePolicy::k_tile_size, lane_id, b_frag);
            ComputePolicy::mma(acc, a_frag, b_frag);
        }

        // --- Release: synchronise after compute ---
        // Single-buffer:  __syncthreads() before next iteration overwrites tile
```

(continues on next page)

(continued from previous page)

```
        // Double-buffer:  __syncthreads() covers both compute-done and prefetch-done
        TilePolicy::release(t);
    }
    // [Sphinx gemm kernel loop end]

    // [Sphinx gemm kernel store start]
    // Write the accumulator to global memory.
    ComputePolicy::store_c(acc, C,
                           block_row_start + thread_row,
                           block_col_start + thread_col,
                           m, n, lane_id);
    // [Sphinx gemm kernel store end]
}
```

9.10.5 Architecture dispatch

A compile-time dispatch block selects the appropriate `ComputePolicy` based on the target Instruction Set Architecture (ISA). The architecture-specific MFMA and WMMA policies are stubs to be filled by follow-up architecture-specific sections:

```
//
// Selecting a ComputePolicy at compile time based on device ISA:
//
//   gfx90a / gfx942   →  MFMAPolicy<float,float>   (CDNA2/3)
//   gfx1100 / gfx1101                  →  WMMAPolicy<__half,float>  (RDNA3)
//   gfx1200 / gfx1201                  →  RDNA4WMMAPolicy (relaxed lane rules)
//   fallback (all others)              →  ScalarFMAPolicy
//
// Usage pattern (not yet compiled; shown for documentation):
//
// #if defined(__gfx90a__) || defined(__gfx942__)
//   using DefaultComputePolicy = MFMAPolicy</* … */>;
// #elif defined(__gfx1100__) || defined(__gfx1101__) || defined(__gfx1102__)
//   using DefaultComputePolicy = WMMAPolicy</* … */>;
// #elif defined(__gfx1200__) || defined(__gfx1201__)
//   using DefaultComputePolicy = RDNA4WMMAPolicy</* … */>;
// #else
//   using DefaultComputePolicy = ScalarFMAPolicy<8, 8, 16>;
// #endif
//
// The kernel template (GemmKernel) is identical for all policies; only the
// type alias changes.
```

Policy aliases used in this example:

```
// Single-buffer variant (baseline - equivalent to register-tiled step 3)
using SingleBufPolicy = SingleBufferTilePolicy<BLOCK_TILE_M, BLOCK_TILE_N, K_TILE_
↪SIZE>;

// Double-buffer variant (software ping-pong, same tile/compute parameters)
using DoubleBufPolicy = SoftwareDoubleBufferTilePolicy<BLOCK_TILE_M, BLOCK_TILE_N, K_
↪TILE_SIZE>;
```

(continues on next page)

(continued from previous page)

```

// Portable scalar outer-product compute policy
using ScalarPolicy = ScalarFMAPolicy<THREAD_TILE_M, THREAD_TILE_N, BLOCK_DIM_X>;

// Direct global→LDS tile policy (CDNA3/CDNA4, bypasses VGPRs during tile load)
#if defined(__gfx942__) || defined(__gfx950__)
using DirectLoadPolicy = DirectLoadTilePolicy<BLOCK_TILE_M, BLOCK_TILE_N, K_TILE_SIZE>
↪;
#endif
```

Compile and run:

```
amdclang++ -O3 -std=c++17 matrix_multiply_generic.hip -o mm_generic
./mm_generic
```

The program launches three variants:

1. `GemmKernel<SingleBufPolicy, ScalarPolicy>` - single-buffer, scalar FP32
2. `GemmKernel<DoubleBufPolicy, ScalarPolicy>` - double-buffer, scalar FP32
3. `GemmKernel<DirectLoadPolicy, ScalarPolicy>` - direct global-to-LDS loads (CDNA3 and CDNA4 only)

The first two differ only in their `TilePolicy`; the third replaces the standard cooperative load with a hardware-specific builtin. All three share the same kernel template and `ComputePolicy`.

Profile wall-clock time with rocprofv3:

```
rocprofv3 --kernel-trace --output-format csv -- ./mm_generic
```

The `--kernel-trace` CSV shows `VGPR_Count` and `LDS_Block_Size` for each of the three dispatched kernels. Compare these columns across the `SingleBufPolicy`, `DoubleBufPolicy`, and `DirectLoadPolicy` variants to confirm the expected differences in register and LDS usage.

9.10.6 Builtin drop-in: `DirectLoadTilePolicy`

The `DirectLoadTilePolicy` demonstrates how an architecture-specific builtin can be used as a drop-in `TilePolicy` without touching the kernel template or the `ComputePolicy`.

On CDNA3 and CDNA4, the `__builtin_amdgcn_global_load_lds` intrinsic transfers data from global memory directly into LDS without staging in vector registers:

```
// Gather 64 floats from global memory into 64 contiguous LDS locations.
// Each lane provides its own global source address (per-lane VADDR).
// The hardware writes lane k's value to dst_chunk + k * sizeof(float).
__builtin_amdgcn_global_load_lds(
    src_lane,      // per-lane global address (VADDR)
    dst_chunk,     // wave-uniform LDS base (-> M0)
    4,             // size per lane in bytes (immediate)
    0,             // offset (immediate)
    0);            // cache policy (immediate)
```

There are three distinct benefits:

1. **Instruction count reduction** - each `global_load_lds_dword` replaces a `global_load_dword` into a VGPR followed by a `ds_write_b32` from that VGPR into LDS. A single instruction does the work of two, halving the

total instruction count for the tile-load phase. For the tile parameters in this example (`BLOCK_TILE_M = 128`, `K_TILE_SIZE = 16`, 4 wavefronts) each wavefront issues 32 `global_load_lds_dword` instructions instead of 32 `global_load_dword` + 32 `ds_write_b32` = 64 instructions. This directly frees instruction-issue bandwidth for the FMA compute phase.

2. **VGPR file bandwidth** - the VGPR file is no longer used as a staging area for tile data during the load phase. Its read/write bandwidth is fully available to the outer-product FMA loop, reducing contention between the load and compute phases.
3. **VGPR count** - the loaded data never occupies vector registers. Fewer VGPRs allocated means more wavefronts can be resident per CU simultaneously (see Step 6), which improves the hardware's ability to hide memory latency through wavefront switching.

On throughput-sensitive workloads the first effect dominates: if the scalar tile-load sequence is instruction-issue-bound, halving its instruction count can produce speedups larger than VGPR savings alone would suggest.

At the ISA level, `global_load_lds_dword` is a **wavefront-wide gather**:

- `VADDR` (a vector register) provides each lane's global source address - the global pointer need not be wave-uniform.
- `M0` (a scalar register) provides the LDS base address - wave-uniform.
- The LDS write destination is implicitly offset per lane: lane *k* writes to `M0 + offset + k * 4` (for `size <= 4`), or `M0 + offset + k * 16` (for `size > 4`).

A single `global_load_lds_dword` instruction therefore gathers **64 floats** (256 bytes) from 64 potentially different global addresses into 64 contiguous LDS locations - all without touching VGPRs.

The `DirectLoadTilePolicy` treats the tile as a flat array of elements and processes it in chunks of 64 (one wavefront width per instruction). For `BLOCK_TILE_M = 128`, `K_TILE_SIZE = 16`: 2048 elements / 64 per instruction = 32 chunks. With 4 wavefronts: **8 instructions per wavefront** to fill the entire tile.

```
template<int BlockTileM_, int BlockTileN_, int KTileSize_>
struct DirectLoadTilePolicy
{
    static constexpr int num_buffers  = 1;
    static constexpr int block_tile_m = BlockTileM_;
    static constexpr int block_tile_n = BlockTileN_;
    static constexpr int k_tile_size  = KTileSize_;

    static_assert((BlockTileM_ * KTileSize_) % 64 == 0,
                  "tile_a element count must be a multiple of 64 (wavefront width)");
    static_assert((BlockTileN_ * KTileSize_) % 64 == 0,
                  "tile_b_T element count must be a multiple of 64 (wavefront width)
↪");

    struct SharedStorage
    {
        float tile_a[BlockTileM_][KTileSize_];
        float tile_b_T[BlockTileN_][KTileSize_];
    };

    // Helper: gather-load a flat tile from global memory to LDS.
    //
    // Elements are processed in chunks of 64 (one per wavefront-wide
    // instruction).  Each lane computes its own global source address;
    // the hardware writes lane k's value to M0 + k * sizeof(float).
    //
```

(continues on next page)

(continued from previous page)

```
    // Parameters:
    //   lds_dest       - base of the flat LDS tile array
    //   global_src     - base of the global matrix
    //   num_rows/cols  - tile dimensions (LDS layout is [num_rows][num_cols])
    //   global_row/col - top-left corner of this tile in the global matrix
    //   global_stride  - row stride of the global matrix
    //   transpose      - if true, swap row/col in the global read pattern
    //                    (used for tile_b_T: LDS[col][ki], global B[ki][col])
    __device__ static void direct_load_flat(float*       lds_dest,
                                            const float* global_src,
                                            int          num_rows,
                                            int          num_cols,
                                            int          global_row_start,
                                            int          global_col_start,
                                            int          global_stride,
                                            bool         transpose,
                                            int          tid,
                                            int          num_threads)
    {
        constexpr int warp_size    = 64;
        const int     wave_id      = tid / warp_size;
        const int     lane         = tid % warp_size;
        const int     num_waves    = num_threads / warp_size;
        const int     total        = num_rows * num_cols;
        const int     total_chunks = total / warp_size;

        for(int chunk = wave_id; chunk < total_chunks; chunk += num_waves)
        {
            const int chunk_start = chunk * warp_size;

            // --- Per-lane global source address ---
            // Flat index of the element this lane handles.
            const int flat_idx = chunk_start + lane;
            const int lr       = flat_idx / num_cols; // row in LDS tile
            const int lc       = flat_idx % num_cols; // col in LDS tile

            // Map to global coordinates (swap row/col for transposed tiles).
            const int gr = global_row_start + (transpose ? lc : lr);
            const int gc = global_col_start + (transpose ? lr : lc);

            // Each lane computes its own global address (VADDR, per-lane).
            const float* src_lane = &global_src[gr * global_stride + gc];

            // --- Wave-uniform LDS destination ---
            // M0 points to the start of this 64-element chunk.
            // The hardware adds lane * sizeof(float) implicitly, so
            // lane k's value lands at lds_dest[chunk_start + k].
            float* dst_chunk = &lds_dest[chunk_start];

            __builtin_amdgcn_global_load_lds(
                const_cast<float*>(src_lane), // per-lane global address
                dst_chunk,                    // wave-uniform LDS base (→ M0)
```

(continues on next page)

(continued from previous page)

```
                4,                              // size: 4 bytes per lane
                0,                              // offset: immediate, 0
                0);                             // aux: wave scope, temporal
        }
    }

    __device__ static void prologue(SharedStorage& /*smem*/,
                                    const float* /*A*/, const float* /*B*/,
                                    int /*block_row_start*/, int /*block_col_start*/,
                                    int /*m*/, int /*n*/, int /*k*/,
                                    int /*tid*/, int /*num_threads*/)
    {
    }

    __device__ static void prefetch(SharedStorage& smem,
                                    const float*   A,
                                    const float*   B,
                                    int            t_next,
                                    int block_row_start, int block_col_start,
                                    int m, int n, int k,
                                    int tid, int num_threads)
    {
        const int t = t_next; // single-buffer: t_next == t

        // tile_a: row-major gather [block_tile_m × k_tile_size]
        // LDS flat layout matches global row-major order, so transpose=false.
        direct_load_flat(&smem.tile_a[0][0], A,
                         block_tile_m, k_tile_size,
                         block_row_start, t * k_tile_size,
                         k,
                         /*transpose=*/false,
                         tid, num_threads);

        // tile_b_T: transposed gather [block_tile_n × k_tile_size]
        //   LDS layout: tile_b_T[col_within_tile][k_idx]
        //   Global read: B[k_idx_global][col_global]
        //   transpose=true swaps lr↔lc in the global coordinate mapping.
        direct_load_flat(&smem.tile_b_T[0][0], B,
                         block_tile_n, k_tile_size,
                         t * k_tile_size, block_col_start,
                         n,
                         /*transpose=*/true,
                         tid, num_threads);
    }

    __device__ static void acquire(int /*t*/) { __syncthreads(); }
    __device__ static void release(int /*t*/) { __syncthreads(); }
    __device__ static int  buf_idx(int /*t*/) { return 0; }

    __device__ static float* get_tile_a(SharedStorage& smem, int /*buf*/)
    {
        return &smem.tile_a[0][0];
```

(continues on next page)

(continued from previous page)

```
    }
    __device__ static float* get_tile_b_T(SharedStorage& smem, int /*buf*/)
    {
        return &smem.tile_b_T[0][0];
    }
};
```

Note

Because the arithmetic is unchanged (full FP32 scalar outer-product), the `DirectLoadPolicy` variant produces results identical to the other two. The only difference is the data path during the tile load phase.

For the full builtin reference - signatures, parameter tables, address calculation formulas, and cache policy encoding - see *Global-to-LDS builtins for AMD GPUs*.

What to observe:

Counter	What to look for
Kernel duration (kernel-trace CSV)	Should be similar to or better than `SingleBufPolicy` (same tile parameters mean same DRAM traffic).
`VGPR_Count` (kernel-trace CSV)	From `--kernel-trace` CSV: should be lower for `DirectLoadPolicy` than for `SingleBufPolicy` (tile data bypasses VGPRs during load).
`LDS_Block_Size` (kernel-trace CSV)	From `--kernel-trace` CSV: same as `SingleBufPolicy` (both use one buffer pair).

9.11 Further reading

The following resources provide deeper coverage of the tools and hardware referenced in this tutorial.

- rocprofv3 documentation - detailed guide to timeline and counter profiling.
- AMD GPU architecture guides (ISA references) - VGPR budgets, LDS bank geometry, and wavefront scheduling details for each architecture family.

CHAPTER

TEN

INTRODUCTION TO COMPILER BUILTINS

In the previous chapters, you learned optimization patterns using standard HIP programming constructs. You implemented reduction algorithms, histogram computations, and matrix multiplications using shared memory, efficient memory access patterns, and workgroup-level parallelism. These patterns teach fundamental optimization principles that apply across GPU architectures.

This chapter introduces Compiler builtins that can significantly accelerate the patterns you have already learned. Compiler builtins provide direct access to specialized GPU units and instructions that compilers cannot automatically generate. Think of this chapter as your hardware acceleration toolkit, organized by architecture for quick reference when optimizing specific patterns.

10.1 Why Compiler builtins matter

Modern compilers are sophisticated, but they cannot always use the full range of hardware capabilities available on AMD GPUs. The compiler rarely auto-vectorizes code to use dedicated dot product units, even when the pattern is evident. Wave-level reduction hardware requires explicit builtin calls rather than relying on compiler optimization. Matrix multiply-accumulate units require direct invocation to achieve peak performance.

Builtins bridge this gap by giving you explicit control over specialized operations. When used appropriately, they can provide orders-of-magnitude performance improvements for compute-bound workloads.

10.2 How this chapter is organized

This chapter organizes builtins by their availability across AMD GPU architectures. Cross-architecture builtins work on both CDNA and RDNA architectures, providing arithmetic and packing operations (dot products, SAD, type conversion), wavefront-level operations (shuffle, DPP, reductions, permlane, vote), and direct-to-Local Data Share (LDS) memory transfers. CDNA-specific builtins cover matrix operations (MFMA) with both dense and sparse variants. RDNA-specific builtins cover wave-matrix multiply-accumulate (WMMA) operations, including sparse variants on RDNA4.

Some builtins require specific hardware features beyond basic architecture support, which are noted with feature flags where applicable.

10.3 When to use builtins

Compiler builtins are most beneficial when:

Compiler optimization is insufficient
: The compiler cannot automatically vectorize your code to use specialized units, even when the pattern is obvious.

Performance-critical loops dominate runtime
: Operations in tight loops that account for most execution time can benefit significantly from hardware acceleration.

Algorithmic patterns map to hardware features
Operations like reductions, matrix multiplications, and histograms have dedicated hardware support that builtins expose.

You are targeting a specific GPU architecture
When you know your target architecture (CDNA or RDNA), you can use its specialized capabilities to maximize performance.

However, builtins introduce trade-offs that you should weigh against the performance benefits:

Reduced portability across architectures
Builtins tie your code to specific GPU architectures or architecture families. Code using MFMA builtins will not compile for RDNA targets, and WMMA builtins will not compile for CDNA targets. If your application must run across multiple architectures, you need separate code paths or an abstraction layer.

Increased maintenance burden
Builtin interfaces can change between architecture generations. New GPU families may introduce different builtins for the same operation, requiring you to update and test multiple code paths. Standard HIP code, by contrast, benefits from compiler improvements automatically.

Lower code readability
Builtin calls are harder to read and reason about than equivalent high-level code. Operations expressed as `__builtin_amdgcn_mfma_f32_32x32x8f16` are less self-documenting than a straightforward nested loop. This increases the cost of bringing new team members up to speed and reviewing changes.

Harder debugging and validation
Debugging builtin-heavy code requires architecture-specific knowledge of register layouts, lane assignments, and hardware execution semantics. Standard debugging techniques such as printf-based inspection become more difficult when data is distributed across wavefront lanes in hardware-defined patterns.

Premature optimization risk
Using builtins before profiling can waste development effort on code paths that are not performance bottlenecks. Always profile first to confirm that a particular operation is the dominant cost, then apply builtins to that specific bottleneck rather than rewriting an entire application.

Start with standard HIP constructs and the optimization patterns in the preceding chapters. Introduce builtins selectively where profiling shows clear opportunities, and isolate builtin code behind abstraction boundaries to limit the impact on portability and maintenance.

10.4 Connection to optimization patterns

The builtins in this chapter directly accelerate the patterns you learned previously. Wave reduction builtins replace manual shuffle-based reductions with single-instruction operations. Direct-to-LDS builtins bypass register staging to reduce register pressure in memory-bound kernels. Matrix multiply-accumulate builtins leverage dedicated matrix cores to achieve orders-of-magnitude higher throughput than scalar operations.

Each builtin section includes notes on which patterns benefit from its use, helping you identify optimization opportunities in your code.

10.5 What you will learn

Working through the Compiler builtins sections will help you:

- Identify which builtins are available on your target GPU architecture
- Apply dot product acceleration and wave-level operations
- Leverage matrix operations (MFMA on CDNA, WMMA on RDNA) for compute-bound applications

- Use direct-to-LDS memory transfers to reduce register pressure
- Apply architecture-specific optimizations for CDNA or RDNA GPUs
- Integrate builtins into the optimization patterns you learned previously

10.6 Prerequisites

Before working through this chapter, you should be familiar with:

- The optimization patterns covered in previous chapters (reduction, histogram, matrix multiplication)
- HIP programming fundamentals, including kernel launch, memory hierarchy, and workgroup organization
- GPU architecture concepts such as waves, wavefronts, and LDS
- Profiling and performance analysis tools to measure optimization benefits

CHAPTER

ELEVEN

ARITHMETIC AND PACKING BUILTINS FOR AMD GPUS

Arithmetic and packing builtins give you direct access to dedicated units for the sum of absolute differences, integer dot products, FP16 (16-bit half-precision) dot products, and type conversion and packing. For a complete listing of all builtins with their signatures and supported architectures, see *Arithmetic and packing builtin reference*.

The complete source file is available for download:

- `builtins_cross_arch_arithmetic.hip`

11.1 Architecture availability

Most arithmetic and packing builtins are available on all AMD Instinct (CDNA) and AMD Radeon (RDNA) architectures. Some builtins — particularly mixed-sign dot products, FP8/BF8 conversions, and stochastic rounding — are limited to specific generations. For per-builtin availability, see the architecture tables in the *reference pages* below.

11.2 Sum of absolute differences

The sum of absolute differences (SAD) family computes the L1 distance between two byte vectors, producing a scalar accumulator that measures the similarity between the two blocks of data. The fundamental use case is block matching in video encoding and image processing. Each candidate reference block is compared against the current block by accumulating the per-byte absolute differences, and the candidate with the lowest SAD is the best match. The three builtins discussed below cover single 4-byte SAD, dual-SAD packing, and four overlapping 8-byte SADs, giving you progressively higher throughput as your data layout permits.

11.2.1 Masked SAD with `msad_u8`

The `__builtin_amdgcn_msad_u8` intrinsic computes the SAD of four packed bytes, excluding byte positions where the reference byte (`src1`) is zero from the accumulation. This masking behavior is designed for padded reference windows, where the encoder marks positions outside the frame boundary with zero rather than valid pixel data. Accumulating those positions would artificially inflate the SAD and bias the match; skipping them keeps the comparison honest regardless of the padding layout.

```
// msad_u8: SAD of four packed bytes, skipping positions where the src1 byte is zero.
// Useful for block matching over padded reference windows where padding is coded as
↪zero.
__global__ __launch_bounds__(block_size)
void msad_u8(const unsigned int *src0,
             const unsigned int *src1,
             unsigned int       *output,
             int                 n)
{
```

(continues on next page)

(continued from previous page)

```
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid >= n)
    {
        return;
    }
    output[gid] = __builtin_amdgcn_msad_u8(src0[gid], src1[gid], 0u);
}
```

The third argument is the initial accumulator value. Passing `0u` gives a fresh SAD for each element; passing the result of a previous call accumulates across multiple 32-bit values.

For the full signature and parameter details, see *msad_u8* in the SAD builtins reference.

11.2.2 Dual-SAD packing with `sad_u8` and `sad_hi_u8`

`__builtin_amdgcn_sad_u8` computes the SAD of four packed bytes and accumulates the result into bits [15:0] of the destination 32-bit value. `__builtin_amdgcn_sad_hi_u8` computes the same byte SAD but shifts each per-byte absolute difference left by 16 bits before accumulating, placing its result in bits [31:16]. Because the two results occupy different halves of the same 32-bit value, one call to each packs two independent 4-byte SADs into a single 32-bit value with no additional packing instructions, storing two SAD results in one register.

```
// sad_u8 accumulates into bits [15:0]; sad_hi_u8 shifts each byte difference
// left 16 before accumulating, placing the result in bits [31:16]. One call
// to each packs two independent 4-byte SADs into one register.
__global__ __launch_bounds__(block_size)
void sad_variants(const unsigned int *src0_lo,
                  const unsigned int *src0_hi,
                  const unsigned int *src1_lo,
                  const unsigned int *src1_hi,
                  unsigned int       *output,
                  int                 n)
{
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid >= n)
    {
        return;
    }
    unsigned int packed = __builtin_amdgcn_sad_u8(src0_lo[gid], src1_lo[gid], 0u);
    packed              = __builtin_amdgcn_sad_hi_u8(src0_hi[gid], src1_hi[gid],␣
↪packed);
    output[gid]         = packed;
}
```

The low SAD accumulates from `src0_lo` against `src1_lo`; the high SAD accumulates from `src0_hi` against `src1_hi` into the same 32-bit output value. The maximum value of a 4-byte SAD is $4 \times 255 = 1020$, which fits in 10 bits, so neither half overflows into the other regardless of the input values.

For the full signatures and parameter details, see *sad_u8* and *sad_hi_u8* in the SAD builtins reference.

11.2.3 Overlapping SAD with `qsad_pk_u16_u8`

`__builtin_amdgcn_qsad_pk_u16_u8` computes four overlapping 4-byte SADs from a single 8-byte reference window in one instruction. The 8-byte source (`src0`) is treated as a sliding window: the four SADs compare bytes [0:3], [1:4], [2:5], and [3:6] of the window against the same 4-byte query (`src1`). The four 16-bit results are packed into the returned 64-bit value, making this the natural tool for a full search over a scan line where the reference window advances one byte at a time.

```
// qsad_pk_u16_u8: four overlapping 8-byte SADs in one instruction.
// src0 is a 64-bit sliding reference window; src1 is the 4-byte query.
// The four results are packed as uint16 values in the returned uint64.
__global__ __launch_bounds__(block_size)
void qsad_pk_u16_u8(const unsigned long long *src0,
                    const unsigned int        *src1,
                    unsigned long long        *output,
                    int                        n)
{
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    if (gid >= n)
    {
        return;
    }
    output[gid] = __builtin_amdgcn_qsad_pk_u16_u8(src0[gid], src1[gid], 0ull);
}
```

Each of the four packed uint16 results can be extracted with a shift and mask: `(result >> (i * 16)) & 0xffff` gives the SAD for window offset `i`. The third argument is the initial accumulator for all four channels simultaneously.

For the full signature and parameter details, see *qsad_pk_u16_u8* in the SAD builtins reference.

11.3 Integer dot products

The dot product builtins compute the inner product of two short integer vectors in a single instruction, accumulating the result into a wider integer. Using the builtin guarantees the corresponding hardware instruction is emitted regardless of compiler heuristics, and makes the intent explicit in the source code.

11.3.1 Unsigned 4-element dot product with `udot4`

`__builtin_amdgcn_udot4` computes the dot product of two vectors of four unsigned bytes, accumulating into a 32-bit unsigned integer. The builtin takes its inputs as packed 32-bit values. The kernel is responsible for packing the individual bytes before the call. The fourth argument is a Boolean that negates the result of the dot product before accumulation. Passing `false` gives a plain accumulate.

```
// udot4: dot product of four unsigned byte pairs accumulated into uint32 (dot7-
↪insts).
// Bytes are packed inline per thread before calling the intrinsic.
__global__ __launch_bounds__(block_size)
void udot4(const unsigned char *a, const unsigned char *b, unsigned int *output, int␣
↪n)
{
    int gid  = blockIdx.x * blockDim.x + threadIdx.x;
    int base = gid * 4;
    if (base + 3 >= n)
    {
```

(continues on next page)

(continued from previous page)

```
        return;
    }
    unsigned int a_word = (unsigned int)a[base + 0]
                        | ((unsigned int)a[base + 1] << 8)
                        | ((unsigned int)a[base + 2] << 16)
                        | ((unsigned int)a[base + 3] << 24);
    unsigned int b_word = (unsigned int)b[base + 0]
                        | ((unsigned int)b[base + 1] << 8)
                        | ((unsigned int)b[base + 2] << 16)
                        | ((unsigned int)b[base + 3] << 24);
    output[gid] = __builtin_amdgcn_udot4(a_word, b_word, 0u, false);
}
```

Each thread packs four consecutive bytes from each input array into a 32-bit value by using shifts and bitwise OR, then calls the builtin with an accumulator of 0. The packing is the only per-thread overhead; the dot product itself is one instruction. For signed integer dot products, see *sdot4* in the integer dot product reference.

For the full signature and parameter details, see *udot4* in the integer dot product reference.

11.4 FP16 dot product

The FP16 dot product builtin computes the inner product of two pairs of half-precision values and accumulates into a single-precision float. The primary motivation is throughput: FP16 multiply-add has higher theoretical throughput than FP32 (32-bit single-precision) on hardware with dedicated mixed-precision units, and accumulating into FP32 preserves the dynamic range needed for deep learning and signal processing workloads.

11.4.1 FP16-to-FP32 accumulation with `fdot2`

`__builtin_amdgcn_fdot2` computes `a[0] * b[0] + a[1] * b[1]` in FP16 precision and accumulates the result into a FP32 accumulator. Inputs are passed as `half2` two-element vectors. The fourth argument negates the dot product before accumulation; passing `false` gives a plain accumulate.

```
// fdot2: dot product of two FP16 pairs accumulated into FP32 (dot10-insts).
// Each thread loads one pair of half2 vectors from consecutive elements.
__global__ __launch_bounds__(block_size)
void fdot2(const __fp16 *a, const __fp16 *b, float *output, int n)
{
    int gid  = blockIdx.x * blockDim.x + threadIdx.x;
    int base = gid * 2;
    if (base + 1 >= n)
    {
        return;
    }
    using half2 = __attribute__((vector_size(4))) __fp16;
    half2 va    = *reinterpret_cast<const half2 *>(&a[base]);
    half2 vb    = *reinterpret_cast<const half2 *>(&b[base]);
    output[gid] = __builtin_amdgcn_fdot2(va, vb, 0.0f, false);
}
```

The kernel casts two consecutive `__fp16` array elements to a `half2` vector via `reinterpret_cast` before the call. The builtin is most valuable when the surrounding code is complex enough that the compiler does not automatically generate the equivalent hardware instruction.

For the full signature and parameter details, see *fdot2* in the floating-point dot product reference.

11.5 Conversion and packing

The SAD and dot product builtins expect their inputs packed as byte fields within 32-bit values. When your source data is stored as floats, converting and packing manually would require several steps. `__builtin_amdgcn_cvt_pk_u8_f32` handles both in one instruction: it clamps the float to [0, 255], converts it to an unsigned byte, and inserts it into one of the four byte positions of an existing destination 32-bit value, leaving the other three bytes unchanged.

11.5.1 Byte packing with `cvt_pk_u8_f32`

`__builtin_amdgcn_cvt_pk_u8_f32` takes the source float, a compile-time byte selector (0–3), and the current destination 32-bit value. Four calls, advancing the selector from 0 to 3 and threading the output of each call into the next, fill all four byte slots of the output 32-bit value in four instructions with no intermediate registers.

```
// cvt_pk_u8_f32: clamp-and-convert one FP32 to uint8 and insert it at byte position
↪i of dst.
// Four calls pack a full word suitable for use with sad_u8, udot4, or similar.
__global__ __launch_bounds__(block_size)
void cvt_pk_u8_f32(const float *input, unsigned int *output, int n)
{
    int gid  = blockIdx.x * blockDim.x + threadIdx.x;
    int base = gid * 4;
    if (base + 3 >= n)
    {
        return;
    }
    unsigned int word = 0u;
    word = __builtin_amdgcn_cvt_pk_u8_f32(input[base + 0], 0, word);
    word = __builtin_amdgcn_cvt_pk_u8_f32(input[base + 1], 1, word);
    word = __builtin_amdgcn_cvt_pk_u8_f32(input[base + 2], 2, word);
    word = __builtin_amdgcn_cvt_pk_u8_f32(input[base + 3], 3, word);
    output[gid] = word;
}
```

The packed 32-bit value is directly usable as the `src0` argument to `sad_u8` or as a component of the packed inputs to `udot4`, making this builtin the natural bridge between float pipelines and the byte-oriented arithmetic builtins in this chapter.

For the full signature and parameter details, see *cvt_pk_u8_f32* in the conversion and packing reference.

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx942 \
    builtins_cross_arch_arithmetic.hip -o arithmetic_builtins
./arithmetic_builtins
```

> **Note**
>
> The example above targets CDNA3 (`gfx942`). Replace `--offload-arch` with the appropriate target for your GPU — for example, `gfx90a` for CDNA2 or `gfx1200` for RDNA4. All builtins used in the examples are available on every supported architecture.

11.6 Naming convention

All hardware arithmetic and packing builtins use the prefix `__builtin_amdgcn_` followed by an operation name and type suffix:

```
__builtin_amdgcn_<operation>[_<qualifier>]_<type>
```

`operation`
The core operation (`sad`, `msad`, `qsad`, `udot`, `sdot`, `sudot`, `fdot`, `cvt`).

`qualifier` (optional)
A modifier that refines the operation. `hi` places the result in the upper half (`sad_hi_u8`). `pk` indicates a pack/unpack conversion (`cvt_pk_u8_f32`). `sr` indicates stochastic rounding (`cvt_sr_fp8_f32`).

`type`
The element type. `u8` = unsigned 8-bit, `u16` = unsigned 16-bit, `f32` = FP32, `f16` = FP16, `bf16` = BF16, `fp8` = FP8 (E4M3), `bf8` = BF8 (E5M2), `i4` = 4-bit fixed-point. When two types appear, the first is the input and the second is the output (`cvt_pk_u8_f32`: FP32 input, uint8 output).

The trailing digit on dot-product names indicates the number of element pairs per call: `udot4` computes a 4-element dot product, `udot8` an 8-element dot product, `fdot2` a 2-element dot product.

11.7 Arithmetic and packing builtin reference

Each reference page documents the full signature, parameter details, and architecture support for every builtin in that family.

Family	Description
Sum of absolute differences	L1 distance between packed byte vectors (`msad_u8`, `sad_u8`, `sad_hi_u8`, `sad_u16`, `qsad_pk_u16_u8`, `mqsad_pk_u16_u8`, `mqsad_u32_u8`)
Integer dot products	Inner products of packed integer vectors (`udot4`, `udot8`, `sdot4`, `sdot8`, `sdot2`, `udot2`, `sudot4`, `sudot8`)
Floating-point dot products	Inner products of packed FP16, BF16, FP8, and BF8 vectors (`fdot2`, `fdot2_f16_f16`, `fdot2_bf16_bf16`, `fdot2_f32_bf16`, `dot4_f32_fp8_*`, `fdot2c_f32_bf16`)
Conversion and packing	Float-to-byte, byte-to-float, and pack/unpack conversions (`cvt_pk_*`, `cvt_pkrtz`, `cvt_pknorm_*`, `cvt_sr_*`, `cvt_f32_*`)

11.8 Sum of absolute differences builtins

The SAD builtins compute the L1 distance between packed byte vectors in a single instruction. They are used in motion estimation, template matching, and image comparison kernels where many small absolute differences must be accumulated quickly.

11.8.1 Common parameters

All SAD builtins share the same three-operand pattern:

`src0` (reference block)
Four packed unsigned bytes (or eight bytes for the `qsad` and `mqsad` variants). This is the data being compared against a query.

`src1` (query block)
Four packed unsigned bytes representing the query. For `msad_u8` and `mqsad_pk_u16_u8`, byte positions where

`src1` is zero are excluded from the accumulation — a masking behavior designed for padded reference windows where positions outside the frame boundary are marked with zero.

`src2` (accumulator)
Initial accumulator value. Pass `0` for a fresh SAD. Pass the result of a previous call to chain multiple SAD operations into a single running total without additional add instructions.

11.8.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
`msad_u8`, `sad_u8`, `sad_hi_u8`, `sad_u16`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`qsad_pk_u16_u8`, `mqsad_pk_u16_u8`, `mqsad_u32_u8`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

11.8.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

11.8.3.1 `__builtin_amdgcn_msad_u8`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_msad_u8(
    unsigned int src0,
    unsigned int src1,
    unsigned int src2);
```

Computes the sum of absolute differences of four packed unsigned bytes, skipping byte positions where the corresponding `src1` byte is zero. This masking behavior is designed for padded reference windows, where positions outside the frame boundary are marked with zero rather than valid pixel data.

Parameter	Type	Description
`src0`	unsigned int	Four packed unsigned bytes: the source data to compare.
`src1`	unsigned int	Four packed unsigned bytes: the reference data. Byte positions where `src1` is zero are excluded from the accumulation.
`src2`	unsigned int	Initial accumulator value. Pass `0u` for a fresh SAD; pass the result of a previous call to accumulate across multiple 32-bit values.

Returns `unsigned int` – accumulated SAD ($\text{src2} + \sum_{k=0}^{3} |\text{src0}[k] - \text{src1}[k]|$ for positions where `src1[k] != 0`).

11.8.3.2 `__builtin_amdgcn_sad_u8`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_sad_u8(
    unsigned int src0,
    unsigned int src1,
    unsigned int src2);
```

Computes the sum of absolute differences of four packed unsigned bytes and accumulates the result into bits [15:0] of the destination. Pair with `sad_hi_u8` to pack two independent 4-byte SADs into a single 32-bit value.

Parameter	Type	Description
`src0`	unsigned int	Four packed unsigned bytes: the source data.
`src1`	unsigned int	Four packed unsigned bytes: the reference data.
`src2`	unsigned int	Initial accumulator value (bits [15:0] are used).

Returns `unsigned int` – accumulated SAD in bits [15:0] ($\text{src2}[15{:}0] + \sum_{k=0}^{3} |\text{src0}[k] - \text{src1}[k]|$).

11.8.3.3 `__builtin_amdgcn_sad_hi_u8`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_sad_hi_u8(
    unsigned int src0,
    unsigned int src1,
    unsigned int src2);
```

Computes the sum of absolute differences of four packed unsigned bytes, shifts each per-byte absolute difference left by 16 bits before accumulating, and places the result in bits [31:16] of the destination. Pair with `sad_u8` to pack two independent 4-byte SADs into a single 32-bit value without additional packing instructions.

Parameter	Type	Description
`src0`	unsigned int	Four packed unsigned bytes: the source data.
`src1`	unsigned int	Four packed unsigned bytes: the reference data.
`src2`	unsigned int	Initial accumulator value (bits [31:16] are used).

Returns `unsigned int` – accumulated SAD in bits [31:16].

11.8.3.4 `__builtin_amdgcn_sad_u16`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_sad_u16(
    unsigned int src0,
    unsigned int src1,
    unsigned int src2);
```

Computes the sum of absolute differences of two packed unsigned 16-bit values and accumulates the result into `src2`.

Parameter	Type	Description
`src0`	unsigned int	Two packed unsigned 16-bit values: the source data.
`src1`	unsigned int	Two packed unsigned 16-bit values: the reference data.
`src2`	unsigned int	Initial accumulator value.

Returns `unsigned int` – accumulated SAD (src2 + $\sum_{k=0}^{1} |\text{src0}[k] - \text{src1}[k]|$).

11.8.3.5 `__builtin_amdgcn_qsad_pk_u16_u8`

Signature and parameters for this builtin.

```
uint64_t __builtin_amdgcn_qsad_pk_u16_u8(
    uint64_t      src0,
    unsigned int src1,
    uint64_t      src2);
```

Computes four overlapping 4-byte SADs from a single 8-byte reference window in one instruction. The 8-byte source (`src0`) is treated as a sliding window: the four SADs compare bytes [0:3], [1:4], [2:5], and [3:6] against the same 4-byte query (`src1`). The four 16-bit results are packed into the returned 64-bit value.

Parameter	Type	Description
`src0`	uint64_t	Eight packed unsigned bytes: the sliding reference window.
`src1`	unsigned int	Four packed unsigned bytes: the query to compare against each window position.
`src2`	uint64_t	Initial accumulator for all four channels simultaneously.

Returns `uint64_t` – four 16-bit SAD results packed into a 64-bit value. Extract individual results with `(result >> (i * 16)) & 0xffff` for window offset `i`.

11.8.3.6 `__builtin_amdgcn_mqsad_pk_u16_u8`

Signature and parameters for this builtin.

```
uint64_t __builtin_amdgcn_mqsad_pk_u16_u8(
    uint64_t      src0,
    unsigned int src1,
    uint64_t      src2);
```

Masked variant of `qsad_pk_u16_u8`. Computes four overlapping 4-byte SADs from an 8-byte sliding window, but skips byte positions where the corresponding `src1` byte is zero. Provides the same sliding-window behavior as `qsad_pk_u16_u8` with the same masking semantics as `msad_u8`.

Parameter	Type	Description
`src0`	uint64_t	Eight packed unsigned bytes: the sliding reference window.
`src1`	unsigned int	Four packed unsigned bytes: the query. Byte positions where `src1` is zero are excluded from the accumulation.
`src2`	uint64_t	Initial accumulator for all four channels simultaneously.

Returns `uint64_t` – four 16-bit masked SAD results packed into a 64-bit value.

11.8.3.7 `__builtin_amdgcn_mqsad_u32_u8`

Signature and parameters for this builtin.

```
uint4 __builtin_amdgcn_mqsad_u32_u8(
    uint64_t      src0,
    unsigned int src1,
    uint4         src2);
```

Computes four overlapping 4-byte SADs from an 8-byte sliding window, returning four full 32-bit results instead of the 16-bit packed format of the `qsad`/`mqsad` variants. Use this when the accumulated SAD values may exceed the 16-bit range (65535).

Parameter	Type	Description
`src0`	uint64_t	Eight packed unsigned bytes: the sliding reference window.
`src1`	unsigned int	Four packed unsigned bytes: the query.
`src2`	uint4	Four-element uint32 accumulator vector.

Returns `uint4` – four 32-bit SAD results.

11.9 Integer dot product builtins

The integer dot product builtins compute inner products of packed integer vectors in a single instruction, accumulating into a 32-bit result. They are the foundation for INT8 and INT4 quantized inference, where weights and activations are packed as four bytes or eight nibbles per register.

11.9.1 Common parameters

All integer dot product builtins share the following parameters:

`src0`, `src1` (packed input vectors)
: Packed integer vectors. The element width and sign interpretation depend on the variant: `udot4`/`sdot4` pack four 8-bit elements, `udot8`/`sdot8` pack eight 4-bit elements, and `sdot2`/`udot2` pack two 16-bit elements. The caller is responsible for packing individual values into the 32-bit register before the call.

`src2` (accumulator)
: Initial accumulator value. The dot product result is added to (or subtracted from, if `neg` is true) this value. Pass `0` for a fresh dot product; pass the result of a previous call to chain multiple dot products across a K-dimension loop.

`neg` (negate before accumulation)
: When `true`, negates the dot product result before adding to `src2`. Pass `false` for a plain accumulate. Present on all variants.

The `sudot4`/`sudot8` variants add per-operand sign control:

`src0_neg`, `src1_neg` (sign selectors)
: When `true`, the corresponding operand is treated as signed; when `false`, it is treated as unsigned. This allows mixed-sign dot products (e.g., signed weights times unsigned activations) without separate builtins for every sign combination.

11.9.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
`udot4`, `udot8`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`sdot4`, `sdot8`, `sdot2`, `udot2`	Yes	Yes	Yes	Yes	No	No	No	No
`sudot4`, `sudot8`	No	No	No	No	Yes	Yes	Yes	Yes

11.9.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

11.9.3.1 `__builtin_amdgcn_udot4`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_udot4(
    unsigned int src0,
    unsigned int src1,
    unsigned int src2,
    bool         neg);
```

Computes the dot product of two vectors of four unsigned bytes, accumulating into a 32-bit unsigned integer.

Parameter	Type	Description
`src0`	unsigned int	Four packed unsigned bytes: the first input vector.
`src1`	unsigned int	Four packed unsigned bytes: the second input vector.
`src2`	unsigned int	Accumulator value.
`neg`	bool	When `true`, negates the dot product before accumulation.

Returns `unsigned int` – updated accumulator $(\text{src2} + (-1)^{\text{neg}} \sum_{k=0}^{3} \text{src0}[k] \times \text{src1}[k])$.

11.9.3.2 `__builtin_amdgcn_udot8`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_udot8(
    unsigned int src0,
    unsigned int src1,
    unsigned int src2,
    bool         neg);
```

Dot product of eight unsigned nibble (4-bit) pairs, accumulating into a 32-bit unsigned integer.

Parameter	Type	Description
`src0`	unsigned int	Eight packed unsigned 4-bit values: the first input vector.
`src1`	unsigned int	Eight packed unsigned 4-bit values: the second input vector.
`src2`	unsigned int	Accumulator value.
`neg`	bool	When `true`, negates the dot product before accumulation.

Returns `unsigned int` – updated accumulator $(\text{src2} + (-1)^{\text{neg}} \sum_{k=0}^{7} \text{src0}[k] \times \text{src1}[k])$.

11.9.3.3 __builtin_amdgcn_sdot4

Signature and parameters for this builtin.

```
int __builtin_amdgcn_sdot4(
    int  src0,
    int  src1,
    int  src2,
    bool neg);
```

Dot product of four signed byte pairs, accumulating into a 32-bit signed integer.

Parameter	Type	Description
src0	int	Four packed signed bytes: the first input vector.
src1	int	Four packed signed bytes: the second input vector.
src2	int	Accumulator value.
neg	bool	When true, negates the dot product before accumulation.

Returns int – updated accumulator.

11.9.3.4 __builtin_amdgcn_sdot8

Signature and parameters for this builtin.

```
int __builtin_amdgcn_sdot8(
    int  src0,
    int  src1,
    int  src2,
    bool neg);
```

Dot product of eight signed nibble (4-bit) pairs, accumulating into a 32-bit signed integer.

Parameter	Type	Description
src0	int	Eight packed signed 4-bit values: the first input vector.
src1	int	Eight packed signed 4-bit values: the second input vector.
src2	int	Accumulator value.
neg	bool	When true, negates the dot product before accumulation.

Returns int – updated accumulator.

11.9.3.5 __builtin_amdgcn_sdot2

Signature and parameters for this builtin.

```
int __builtin_amdgcn_sdot2(
    short2 src0,
    short2 src1,
    int    src2,
    bool   neg);
```

Dot product of two signed 16-bit pairs, accumulating into a 32-bit signed integer.

Parameter	Type	Description
src0	short2	Two-element signed int16 vector: the first input.
src1	short2	Two-element signed int16 vector: the second input.
src2	int	Accumulator value.
neg	bool	When true, negates the dot product before accumulation.

Returns int – updated accumulator.

11.9.3.6 __builtin_amdgcn_udot2

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_udot2(
    ushort2      src0,
    ushort2      src1,
    unsigned int src2,
    bool         neg);
```

Dot product of two unsigned 16-bit pairs, accumulating into a 32-bit unsigned integer.

Parameter	Type	Description
src0	ushort2	Two-element unsigned int16 vector: the first input.
src1	ushort2	Two-element unsigned int16 vector: the second input.
src2	unsigned int	Accumulator value.
neg	bool	When true, negates the dot product before accumulation.

Returns unsigned int – updated accumulator.

11.9.3.7 __builtin_amdgcn_sudot4

Signature and parameters for this builtin.

```
int __builtin_amdgcn_sudot4(
    bool src0_neg,
    int  src0,
    bool src1_neg,
    int  src1,
    int  src2,
    bool neg);
```

Dot product of four byte pairs with per-operand sign control. src0_neg and src1_neg independently select whether each operand is interpreted as signed or unsigned, enabling mixed-sign dot products without separate builtins.

Parameter	Type	Description
src0_neg	bool	When true, src0 bytes are treated as signed (int8); when false, unsigned (uint8).
src0	int	Four packed bytes: the first input vector.
src1_neg	bool	When true, src1 bytes are treated as signed; when false, unsigned.
src1	int	Four packed bytes: the second input vector.
src2	int	Accumulator value.
neg	bool	When true, negates the dot product before accumulation.

Returns `int` – updated accumulator.

11.9.3.8 `__builtin_amdgcn_sudot8`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_sudot8(
    bool src0_neg,
    int  src0,
    bool src1_neg,
    int  src1,
    int  src2,
    bool neg);
```

Dot product of eight nibble pairs with per-operand sign control. Same mixed-sign semantics as `sudot4` but operating on 4-bit elements.

Parameter	Type	Description
`src0_neg`	bool	When `true`, `src0` nibbles are treated as signed; when `false`, unsigned.
`src0`	int	Eight packed 4-bit values: the first input vector.
`src1_neg`	bool	When `true`, `src1` nibbles are treated as signed; when `false`, unsigned.
`src1`	int	Eight packed 4-bit values: the second input vector.
`src2`	int	Accumulator value.
`neg`	bool	When `true`, negates the dot product before accumulation.

Returns `int` – updated accumulator.

11.10 Floating-point dot product builtins

The floating-point dot product builtins compute inner products of packed floating-point vectors, accumulating into FP16, BF16, or FP32. They provide higher throughput than scalar multiply-add sequences on hardware with dedicated mixed-precision units, and are used in deep learning inference, signal processing, and any workload that benefits from reduced-precision computation with full-precision accumulation.

11.10.1 Common parameters

Most floating-point dot product builtins share the following parameters:

`src0`, `src1` (packed input vectors)
: Packed floating-point vectors. The element type depends on the variant: `half2` for FP16, `short2` for BF16 (storage type), or `unsigned int` for FP8/BF8 (four packed 8-bit floats).

`src2` (accumulator)
: Initial accumulator value. The dot product result is added to (or subtracted from, if `neg` is present and true) this value. The accumulator type matches the output type: `float` for FP32-accumulate variants, `__half` for FP16-accumulate, `short` for BF16-accumulate.

`neg` (negate before accumulation)
: When `true`, negates the dot product result before adding to `src2`. Pass `false` for a plain accumulate. Present on `fdot2`, `fdot2_f32_bf16`, and `fdot2c_f32_bf16`. Not present on the same-type-accumulate variants (`fdot2_f16_f16`, `fdot2_bf16_bf16`) or the FP8/BF8 variants.

11.10.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDN	CDN/	CDN/	CDN/	RDN/	RDN/	RDNA:	RDNA4
`fdot2`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`fdot2_f16_f16`, `fdot2_bf16_bf16`	No	No	No	No	No	Yes	Yes	Yes
`fdot2_f32_bf16`	No	No	No	No	No	Yes	Yes	Yes
`dot4_f32_fp8_fp8`, `dot4_f32_bf8_bf8`, `dot4_f32_fp8_bf8`, `dot4_f32_bf8_fp8`	No	No	No	No	No	No	No	Yes
`fdot2c_f32_bf16`	No	No	No	Yes	No	No	No	No

11.10.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

11.10.3.1 `__builtin_amdgcn_fdot2`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_fdot2(
    half2 src0,
    half2 src1,
    float src2,
    bool  neg);
```

Computes the dot product of two pairs of FP16 values and accumulates the result into a single-precision FP32 accumulator. The FP16 multiply-add has higher throughput than FP32 on hardware with dedicated mixed-precision units, while accumulating into FP32 preserves the dynamic range needed for deep learning and signal processing workloads.

Parameter	Type	Description
`src0`	half2	Two-element FP16 vector: the first input.
`src1`	half2	Two-element FP16 vector: the second input.
`src2`	float	FP32 accumulator value.
`neg`	bool	When `true`, negates the dot product before accumulation.

Returns `float` – updated accumulator $(\text{src2} + (-1)^{\text{neg}}(\text{src0}[0] \times \text{src1}[0] + \text{src0}[1] \times \text{src1}[1]))$.

11.10.3.2 `__builtin_amdgcn_fdot2_f16_f16`

Signature and parameters for this builtin.

```
__half __builtin_amdgcn_fdot2_f16_f16(
    half2  src0,
    half2  src1,
    __half src2);
```

Dot product of two FP16 pairs accumulated into FP16. Both computation and accumulation use FP16 precision, avoiding the cost of FP32 accumulation when the reduced dynamic range is acceptable.

Parameter	Type	Description
src0	half2	Two-element FP16 vector: the first input.
src1	half2	Two-element FP16 vector: the second input.
src2	__half	FP16 accumulator value.

Returns `__half` – updated FP16 accumulator.

11.10.3.3 `__builtin_amdgcn_fdot2_bf16_bf16`

Signature and parameters for this builtin.

```
short __builtin_amdgcn_fdot2_bf16_bf16(
    short2 src0,
    short2 src1,
    short  src2);
```

Dot product of two BF16 pairs accumulated into BF16. Uses `short`/`short2` as the storage type for BF16 values.

Parameter	Type	Description
src0	short2	Two-element BF16 vector (storage type): the first input.
src1	short2	Two-element BF16 vector (storage type): the second input.
src2	short	BF16 accumulator value (storage type).

Returns `short` – updated BF16 accumulator (storage type).

11.10.3.4 `__builtin_amdgcn_fdot2_f32_bf16`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_fdot2_f32_bf16(
    short2 src0,
    short2 src1,
    float  src2,
    bool   neg);
```

Dot product of two BF16 pairs accumulated into FP32. Combines the throughput advantage of BF16 inputs with the dynamic range of FP32 accumulation.

Parameter	Type	Description
src0	short2	Two-element BF16 vector (storage type): the first input.
src1	short2	Two-element BF16 vector (storage type): the second input.
src2	float	FP32 accumulator value.
neg	bool	When `true`, negates the dot product before accumulation.

Returns `float` – updated FP32 accumulator.

11.10.3.5 `__builtin_amdgcn_dot4_f32_fp8_fp8`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_dot4_f32_fp8_fp8(
    unsigned int src0,
    unsigned int src1,
    float        src2);
```

Dot product of four FP8 (E4M3) pairs accumulated into FP32. Each 32-bit input packs four 8-bit E4M3 floating-point values.

Parameter	Type	Description
`src0`	unsigned int	Four packed FP8 (E4M3) values: the first input vector.
`src1`	unsigned int	Four packed FP8 (E4M3) values: the second input vector.
`src2`	float	FP32 accumulator value.

Returns `float` – updated FP32 accumulator.

11.10.3.6 `__builtin_amdgcn_dot4_f32_bf8_bf8`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_dot4_f32_bf8_bf8(
    unsigned int src0,
    unsigned int src1,
    float        src2);
```

Dot product of four BF8 (E5M2) pairs accumulated into FP32.

Parameter	Type	Description
`src0`	unsigned int	Four packed BF8 (E5M2) values: the first input vector.
`src1`	unsigned int	Four packed BF8 (E5M2) values: the second input vector.
`src2`	float	FP32 accumulator value.

Returns `float` – updated FP32 accumulator.

11.10.3.7 `__builtin_amdgcn_dot4_f32_fp8_bf8`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_dot4_f32_fp8_bf8(
    unsigned int src0,
    unsigned int src1,
    float        src2);
```

Dot product of four pairs with FP8 (E4M3) `src0` and BF8 (E5M2) `src1`, accumulated into FP32.

Parameter	Type	Description
`src0`	unsigned int	Four packed FP8 (E4M3) values: the first input vector.
`src1`	unsigned int	Four packed BF8 (E5M2) values: the second input vector.
`src2`	float	FP32 accumulator value.

Returns `float` – updated FP32 accumulator.

11.10.3.8 `__builtin_amdgcn_dot4_f32_bf8_fp8`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_dot4_f32_bf8_fp8(
    unsigned int src0,
    unsigned int src1,
    float        src2);
```

Dot product of four pairs with BF8 (E5M2) `src0` and FP8 (E4M3) `src1`, accumulated into FP32.

Parameter	Type	Description
`src0`	unsigned int	Four packed BF8 (E5M2) values: the first input vector.
`src1`	unsigned int	Four packed FP8 (E4M3) values: the second input vector.
`src2`	float	FP32 accumulator value.

Returns `float` – updated FP32 accumulator.

11.10.3.9 `__builtin_amdgcn_fdot2c_f32_bf16`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_fdot2c_f32_bf16(
    __hip_bfloat162 src0,
    __hip_bfloat162 src1,
    float           src2,
    bool            neg);
```

Dot product of two BF16 pairs accumulated into FP32, using the HIP `__hip_bfloat162` type for inputs. This variant uses the native BF16 type rather than the `short2` storage type used by `fdot2_f32_bf16`.

Parameter	Type	Description
`src0`	__hip_bfloat162	Two-element BF16 vector: the first input.
`src1`	__hip_bfloat162	Two-element BF16 vector: the second input.
`src2`	float	FP32 accumulator value.
`neg`	bool	When `true`, negates the dot product before accumulation.

Returns `float` – updated FP32 accumulator.

11.11 Conversion and packing builtins

The conversion and packing builtins convert between floating-point and integer types and pack multiple values into a single 32-bit register. They bridge float pipelines with the byte-oriented arithmetic builtins (SAD, dot products) by converting FP32 results to packed bytes, or unpacking reduced- precision inputs to FP32 for further computation.

11.11.1 Common parameters

Several parameters appear across multiple conversion builtins:

`byte_sel` (byte position selector)
An integer in the range 0–3 that selects which byte of a 32-bit destination to overwrite. Byte 0 is the least significant. Used by `cvt_pk_u8_f32` and the single-element FP8/BF8 conversion builtins (`cvt_sr_fp8_f32`, `cvt_sr_bf8_f32`).

`sel` (half selector)
A boolean that selects the upper or lower 16-bit half of the destination. When `false`, the lower 16 bits are written; when `true`, the upper 16 bits. Used by the pack-two variants (`cvt_pk_fp8_f32`, `cvt_pk_bf8_f32`) and the stochastic-rounding BF16/FP16 variants (`cvt_sr_bf16_f32`, `cvt_sr_f16_f32`).

`dst` / `src0` (destination register for in-place insertion)
The current value of the destination register. The selected byte or half is overwritten; the remaining bits are preserved. This allows multiple calls to fill a packed register incrementally without extra move instructions.

`rand` (stochastic rounding seed)
A 32-bit random value used for stochastic rounding. The builtin uses this value to randomly round up or down to the nearest representable value in the target format, providing an unbiased estimator that improves convergence in training workloads compared to deterministic rounding. Used by `cvt_sr_fp8_f32`, `cvt_sr_bf8_f32`, `cvt_sr_bf16_f32`, and `cvt_sr_f16_f32`.

11.11.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDN	CDN	CDN	CDN	RDN	RDN	RDN	RDNA4
`cvt_pk_u8_f32`, `cvt_pkrtz`, `cvt_pk_i16`, `cvt_pk_u16`, `cvt_off_f32_i4`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`cvt_pknorm_i16`, `cvt_pknorm_u16`	No	No	No	No	Yes	Yes	Yes	Yes
`cvt_pk_fp8_f32`, `cvt_pk_bf8_f32`, `cvt_pk_f32_fp8`, `cvt_pk_f32_bf8`, `cvt_f32_fp8`, `cvt_f32_bf8`, `cvt_sr_fp8_f32`, `cvt_sr_bf8_f32`	No	No	Yes	Yes	No	No	No	Yes
`cvt_sr_bf16_f32`, `cvt_sr_f16_f32`	No	No	No	Yes	No	No	No	No

11.11.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

11.11.3.1 FP32 to packed integer

These builtins convert FP32 values to packed unsigned byte representations.

11.11.3.1.1 `__builtin_amdgcn_cvt_pk_u8_f32`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_cvt_pk_u8_f32(
    float        src,
    unsigned int byte_sel,
    unsigned int dst);
```

Clamps a float to [0, 255], converts it to an unsigned byte, and inserts it at one of the four byte positions of an existing 32-bit value, leaving the other three bytes unchanged. Four calls, advancing `byte_sel` from 0 to 3 and threading the output of each call into the next, fill all four byte slots in four instructions with no intermediate registers.

Parameter	Type	Description
`src`	float	Source float value. Clamped to [0, 255] before conversion.
`byte_sel`	unsigned int	Byte position selector (0–3). Selects which byte of `dst` to overwrite.
`dst`	unsigned int	Current destination value. The selected byte is overwritten; the other three bytes are preserved.

Returns `unsigned int` – `dst` with byte position `byte_sel` replaced by the clamped and converted `src` value.

11.11.3.2 FP32 to packed FP16

These builtins convert pairs of FP32 values to packed FP16 vectors.

11.11.3.2.1 `__builtin_amdgcn_cvt_pkrtz`

Signature and parameters for this builtin.

```
__half2 __builtin_amdgcn_cvt_pkrtz(
    float src0,
    float src1);
```

Converts two FP32 values to FP16 using round-toward-zero and packs them into a two-element vector. Round-toward-zero is faster than round-to-nearest on some hardware and is acceptable when the conversion is followed by further reduced-precision computation.

Parameter	Type	Description
`src0`	float	First FP32 value to convert (placed in element 0).
`src1`	float	Second FP32 value to convert (placed in element 1).

Returns `__half2` – two-element FP16 vector containing the converted values.

11.11.3.3 Integer packing

These builtins pack pairs of 32-bit integer values into 16-bit vector types.

11.11.3.3.1 `__builtin_amdgcn_cvt_pk_i16`

Signature and parameters for this builtin.

```
short2 __builtin_amdgcn_cvt_pk_i16(
    int src0,
    int src1);
```

Packs two int32 values into a two-element int16 vector by taking the lower 16 bits of each input.

Parameter	Type	Description
`src0`	int	First int32 value (lower 16 bits used).
`src1`	int	Second int32 value (lower 16 bits used).

Returns `short2` – two-element int16 vector.

11.11.3.3.2 `__builtin_amdgcn_cvt_pk_u16`

Signature and parameters for this builtin.

```
ushort2 __builtin_amdgcn_cvt_pk_u16(
    unsigned int src0,
    unsigned int src1);
```

Packs two uint32 values into a two-element uint16 vector by taking the lower 16 bits of each input.

Parameter	Type	Description
`src0`	unsigned int	First uint32 value (lower 16 bits used).
`src1`	unsigned int	Second uint32 value (lower 16 bits used).

Returns `ushort2` – two-element uint16 vector.

11.11.3.4 Fixed-point conversion

These builtins convert between fixed-point nibble values and FP32.

11.11.3.4.1 `__builtin_amdgcn_cvt_off_f32_i4`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_cvt_off_f32_i4(
    int src);
```

Converts bits [3:0] of `src` as a 4-bit fixed-point fraction to FP32, mapping nibble value `n` to `n / 16.0`. Used internally by some dithering and noise-generation patterns.

Parameter	Type	Description
`src`	int	Source value; only bits [3:0] are used.

Returns `float` – the 4-bit nibble interpreted as a fixed-point fraction in the range [0, 15/16].

11.11.3.5 Normalized packing

These builtins convert FP32 values to normalized 16-bit integers and pack them into vectors.

11.11.3.5.1 `__builtin_amdgcn_cvt_pknorm_i16`

Signature and parameters for this builtin.

```
short2 __builtin_amdgcn_cvt_pknorm_i16(
    float src0,
    float src1);
```

Converts two FP32 values to normalized int16 and packs them into a two-element vector. The FP32 inputs are clamped to [-1.0, 1.0] and scaled to the int16 range [-32768, 32767].

Parameter	Type	Description
`src0`	float	First FP32 value to convert.
`src1`	float	Second FP32 value to convert.

Returns `short2` – two-element normalized int16 vector.

11.11.3.5.2 `__builtin_amdgcn_cvt_pknorm_u16`

Signature and parameters for this builtin.

```
ushort2 __builtin_amdgcn_cvt_pknorm_u16(
    float src0,
    float src1);
```

Converts two FP32 values to normalized uint16 and packs them into a two-element vector. The FP32 inputs are clamped to [0.0, 1.0] and scaled to the uint16 range [0, 65535].

Parameter	Type	Description
`src0`	float	First FP32 value to convert.
`src1`	float	Second FP32 value to convert.

Returns `ushort2` – two-element normalized uint16 vector.

11.11.3.6 FP8 and BF8 pack and unpack

These builtins convert between FP32 and 8-bit floating-point formats (FP8 E4M3 and BF8 E5M2).

11.11.3.6.1 `__builtin_amdgcn_cvt_pk_fp8_f32`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_cvt_pk_fp8_f32(
    float src0,
    float src1,
    int   dst,
    bool  sel);
```

Converts two FP32 values to FP8 (E4M3) and packs them into the upper or lower 16 bits of `dst` depending on `sel`.

Parameter	Type	Description
`src0`	float	First FP32 value to convert.
`src1`	float	Second FP32 value to convert.
`dst`	int	Current destination value. The unselected 16 bits are preserved.
`sel`	bool	When `false`, writes to the lower 16 bits; when `true`, the upper 16 bits.

Returns `int` – `dst` with the selected half replaced by the two packed FP8 values.

11.11.3.6.2 `__builtin_amdgcn_cvt_pk_bf8_f32`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_cvt_pk_bf8_f32(
    float src0,
    float src1,
    int   dst,
    bool  sel);
```

Converts two FP32 values to BF8 (E5M2) and packs them into the upper or lower 16 bits of `dst` depending on `sel`.

Parameter	Type	Description
`src0`	float	First FP32 value to convert.
`src1`	float	Second FP32 value to convert.
`dst`	int	Current destination value. The unselected 16 bits are preserved.
`sel`	bool	When `false`, writes to the lower 16 bits; when `true`, the upper 16 bits.

Returns `int` – `dst` with the selected half replaced by the two packed BF8 values.

11.11.3.6.3 `__builtin_amdgcn_cvt_pk_f32_fp8`

Signature and parameters for this builtin.

```
float2 __builtin_amdgcn_cvt_pk_f32_fp8(
    int  src,
    bool sel);
```

Unpacks two FP8 (E4M3) bytes from the upper or lower 16 bits of `src` and converts them to two FP32 values.

Parameter	Type	Description
`src`	int	Packed source containing four FP8 values.
`sel`	bool	When `false`, reads from the lower 16 bits; when `true`, the upper 16 bits.

Returns `float2` – two FP32 values converted from the selected FP8 pair.

11.11.3.6.4 `__builtin_amdgcn_cvt_pk_f32_bf8`

Signature and parameters for this builtin.

```
float2 __builtin_amdgcn_cvt_pk_f32_bf8(
    int  src,
    bool sel);
```

Unpacks two BF8 (E5M2) bytes from the upper or lower 16 bits of `src` and converts them to two FP32 values.

Parameter	Type	Description
`src`	int	Packed source containing four BF8 values.
`sel`	bool	When `false`, reads from the lower 16 bits; when `true`, the upper 16 bits.

Returns `float2` – two FP32 values converted from the selected BF8 pair.

11.11.3.6.5 `__builtin_amdgcn_cvt_f32_fp8`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_cvt_f32_fp8(
    int src,
    int byte_sel);
```

Unpacks one FP8 (E4M3) byte at position `byte_sel` from `src` and converts it to FP32.

Parameter	Type	Description
`src`	int	Packed source containing four FP8 values.
`byte_sel`	int	Byte position selector (0–3).

Returns `float` – the selected FP8 value converted to FP32.

11.11.3.6.6 `__builtin_amdgcn_cvt_f32_bf8`

Signature and parameters for this builtin.

```
float __builtin_amdgcn_cvt_f32_bf8(
    int src,
    int byte_sel);
```

Unpacks one BF8 (E5M2) byte at position `byte_sel` from `src` and converts it to FP32.

Parameter	Type	Description
`src`	int	Packed source containing four BF8 values.
`byte_sel`	int	Byte position selector (0–3).

Returns `float` – the selected BF8 value converted to FP32.

11.11.3.7 Stochastic rounding conversions

These builtins convert FP32 values to reduced-precision formats using stochastic rounding.

11.11.3.7.1 `__builtin_amdgcn_cvt_sr_fp8_f32`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_cvt_sr_fp8_f32(
    float src,
    int   rand,
    int   dst,
    int   byte_sel);
```

Converts FP32 to FP8 (E4M3) with stochastic rounding and inserts the result at byte position `byte_sel` of `dst`.

Parameter	Type	Description
src	float	FP32 value to convert.
rand	int	Random seed for stochastic rounding.
dst	int	Current destination value. The selected byte is overwritten.
byte_sel	int	Byte position selector (0–3).

Returns int – dst with byte position byte_sel replaced by the FP8 value produced by stochastic rounding.

11.11.3.7.2 __builtin_amdgcn_cvt_sr_bf8_f32

Signature and parameters for this builtin.

```
int __builtin_amdgcn_cvt_sr_bf8_f32(
    float src,
    int   rand,
    int   dst,
    int   byte_sel);
```

Converts FP32 to BF8 (E5M2) with stochastic rounding and inserts the result at byte position byte_sel of dst.

Parameter	Type	Description
src	float	FP32 value to convert.
rand	int	Random seed for stochastic rounding.
dst	int	Current destination value. The selected byte is overwritten.
byte_sel	int	Byte position selector (0–3).

Returns int – dst with byte position byte_sel replaced by the BF8 value produced by stochastic rounding.

11.11.3.7.3 __builtin_amdgcn_cvt_sr_bf16_f32

Signature and parameters for this builtin.

```
__hip_bfloat162 __builtin_amdgcn_cvt_sr_bf16_f32(
    __hip_bfloat162 src0,
    float           src1,
    unsigned int    rand,
    bool            sel);
```

Converts FP32 to BF16 with stochastic rounding and inserts the result into the lane of src0 selected by sel.

Parameter	Type	Description
src0	__hip_bfloat162	Current two-element BF16 vector. The unselected lane is preserved.
src1	float	FP32 value to convert.
rand	unsigned int	Random seed for stochastic rounding.
sel	bool	When false, writes to element 0; when true, element 1.

Returns __hip_bfloat162 – src0 with the selected element replaced by the BF16 value produced by stochastic rounding.

11.11.3.7.4 `__builtin_amdgcn_cvt_sr_f16_f32`

Signature and parameters for this builtin.

```
half2 __builtin_amdgcn_cvt_sr_f16_f32(
    half2        src0,
    float        src1,
    unsigned int rand,
    bool         sel);
```

Converts FP32 to FP16 with stochastic rounding and inserts the result into the lane of `src0` selected by `sel`.

Parameter	Type	Description
`src0`	half2	Current two-element FP16 vector. The unselected lane is preserved.
`src1`	float	FP32 value to convert.
`rand`	unsigned int	Random seed for stochastic rounding.
`sel`	bool	When `false`, writes to element 0; when `true`, element 1.

Returns `half2` – `src0` with the selected element replaced by the FP16 value produced by stochastic rounding.

CHAPTER

TWELVE

GLOBAL-TO-LDS BUILTINS FOR AMD GPUS

AMD GPUs provide a family of builtins that load data from global or buffer memory directly into Local Data Share (LDS) without staging the values in Vector General-Purpose Registers (VGPRs). Because the loaded data never occupies a VGPR, these builtins reduce register pressure and can enable higher occupancy in VGPR-constrained kernels.

All builtins in this family are wavefront-wide operations: each lane provides its own source address, and the hardware writes lane *k*'s value to the LDS base address plus an implicit per-lane stride.

12.1 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
`__builtin_amdgcn_global_load_lds`	No	No	Yes	Yes	No	No	No	No
`__builtin_amdgcn_load_to_lds`	No	No	Yes	Yes	No	No	No	No
`__builtin_amdgcn_raw_ptr_buffer_`	No	No	Yes	Yes	Yes	No	No	No
`__builtin_amdgcn_struct_ptr_buff`	No	No	Yes	Yes	Yes	No	No	No

12.2 Flat-addressed builtins

These builtins take a pointer directly as the source address.

12.2.1 `__builtin_amdgcn_global_load_lds`

Signature and parameters for this builtin.

```
void __builtin_amdgcn_global_load_lds(const void* src,
                                      __shared__ void* dst_base,
                                      std::int32_t size,
                                      std::int32_t offset,
                                      std::int32_t aux);
```

Wavefront-wide gather from global memory to LDS, bypassing VGPRs. Each lane loads `size` bytes from its own global address; the hardware writes lane *k*'s value to `dst_base + offset + k * stride` where `stride = 4` for `size <= 4` and (on CDNA4 GPUs) `16` for `size > 4`.

Parameter	Description
`src`	Pointer to global address space. Each lane can hold a different address.
`dst_ba`	Pointer to LDS address space base. Must be wave-uniform. The pointer is implicitly offset by `lane_id * stride`. `stride` is `4` for `size <= 4` and (on CDNA4 GPUs) `16` for `size > 4`.
`size`	Transfer size per lane in bytes. Must be a compile-time constant. CDNA3 supports 1, 2, and 4. CDNA4 additionally supports 12 and 16.
`offset`	Signed byte offset applied to **both** `src` and `dst_base`. Must be a compile-time constant. Encoded as a 13-bit signed immediate (valid range: `-4096` to `4095`).
`aux`	Cache policy bits. Must be a compile-time constant. See *Cache policy (aux parameter)*.

12.2.2 `__builtin_amdgcn_load_to_lds`

Signature and parameters for this builtin.

```
void __builtin_amdgcn_load_to_lds(const void* src,
                                  __shared__ void* dst_base,
                                  std::int32_t size,
                                  std::int32_t offset,
                                  std::int32_t aux);
```

Wavefront-wide gather from generic, global, or buffer memory to LDS, bypassing VGPRs. Each lane loads `size` bytes from its own address; the hardware writes lane *k*'s value to `dst_base + offset + k * stride` where `stride = 4` for `size <= 4` and (on CDNA4 GPUs) `16` for `size > 4`.

Parameter	Description
`src`	Generic, global, or buffer address pointer. Each lane can hold a different address.
`dst_ba`	Pointer to LDS address space base. Must be wave-uniform. The pointer is implicitly offset by `lane_id * stride`. `stride` is `4` for `size <= 4` and (on CDNA4 GPUs) `16` for `size > 4`.
`size`	Transfer size per lane in bytes. Must be a compile-time constant. CDNA3 supports 1, 2, and 4. CDNA4 additionally supports 12 and 16.
`offset`	Signed byte offset applied to **both** `src` and `dst_base`. Must be a compile-time constant. Encoded as a 13-bit signed immediate (valid range: `-4096` to `4095`).
`aux`	Cache policy bits. Must be a compile-time constant. See *Cache policy (aux parameter)*.

12.3 Buffer-addressed builtins

These builtins take a buffer resource descriptor as the source, created with `__builtin_amdgcn_make_buffer_rsrc`. The three offset parameters (per-lane, wave-uniform, and compile-time) are combined to form the final source address.

> **Note**
>
> The buffer-addressed builtins are available on CDNA, CDNA2, CDNA3, CDNA4, and RDNA2. They're not supported on RDNA3, RDNA3.5, or RDNA4.

12.3.1 __builtin_amdgcn_raw_ptr_buffer_load_lds

Signature and parameters for this builtin.

```
void __builtin_amdgcn_raw_ptr_buffer_load_lds(
    __amdgpu_buffer_rsrc_t src,
    __shared__ void*       dst_base,
    std::int32_t           size,
    std::int32_t           voffset,
    std::int32_t           soffset,
    std::int32_t           offset,
    std::int32_t           aux);
```

For each lane in a wavefront, the **load address** (global source) is:

```
src_addr = src_base + soffset + offset + voffset + stride * lane_id
```

`src_base` and `stride` come from the buffer resource descriptor. The `lane_id` term is active only if the corresponding flag was set when creating the resource descriptor. Otherwise, it evaluates to zero.

For each lane in a wavefront, the **store address** (LDS destination) is:

```
dst_addr = dst_base + offset + lane_id * 4    // size <= 4
dst_addr = dst_base + offset + lane_id * 16   // size > 4 (CDNA4 only)
```

The `lane_id` term in the LDS address is always active.

Parameter	Description
`src`	Buffer resource descriptor (`__amdgpu_buffer_rsrc_t`). Created with `__builtin_amdgcn_make_buffer_rsrc`. The descriptor wraps a base pointer, stride, extent, and flags.
`dst_ba`	Pointer to LDS address space base. Must be wave-uniform. The hardware adds `lane_id * stride` implicitly.
`size`	Transfer size per lane in bytes. Must be a compile-time constant. CDNA, CDNA2, CDNA3, and RDNA2 support 1, 2, and 4. CDNA4 additionally supports 12 and 16.
`voffset`	Per-lane byte offset into the buffer. Each lane can provide a different value, enabling gather-style reads from the buffer.
`soffset`	Wave-uniform byte offset into the buffer.
`offset`	Unsigned compile-time byte offset. Encoded as a 12-bit unsigned immediate (valid range: `0` to `4095`).
`aux`	Cache policy bits (compile-time constant). Bit 3 enables buffer swizzling on all supported architectures. The remaining bits differ by architecture. See *Cache policy (aux parameter)*.

12.3.2 __builtin_amdgcn_struct_ptr_buffer_load_lds

Signature and parameters for this builtin.

```
void __builtin_amdgcn_struct_ptr_buffer_load_lds(
    __amdgpu_buffer_rsrc_t src,
    __shared__ void*       dst_base,
    std::int32_t           size,
```

(continues on next page)

(continued from previous page)

```
    std::int32_t            vindex,
    std::int32_t            voffset,
    std::int32_t            soffset,
    std::int32_t            offset,
    std::int32_t            aux);
```

Structured-buffer direct-to-LDS load. The source is described by a buffer resource descriptor created with `__builtin_amdgcn_make_buffer_rsrc`. The `vindex` parameter provides a per-lane index that is combined with `lane_id` in the source address calculation, enabling structured access patterns over the buffer.

For each lane in a wavefront, the **load address** (global source) is:

```
src_addr = src_base + soffset + offset + voffset + stride * (vindex + lane_id)
```

`src_base` and `stride` come from the buffer resource descriptor. The `lane_id` term is active only if the corresponding flag was set when creating the resource descriptor. Otherwise, it evaluates to zero.

For each lane in a wavefront, the **store address** (LDS destination) is:

```
dst_addr = dst_base + offset + lane_id * 4    // size <= 4
dst_addr = dst_base + offset + lane_id * 16   // size > 4 (CDNA4 only)
```

The `lane_id` term in the LDS address is always active.

Parameter	Description
`src`	Buffer resource descriptor (`__amdgpu_buffer_rsrc_t`). Created with `__builtin_amdgcn_make_buffer_rsrc`. The descriptor wraps a base pointer, stride, extent, and flags.
`dst_ba`	Pointer to LDS address space base. Must be wave-uniform. The hardware adds `lane_id * stride` implicitly.
`size`	Transfer size per lane in bytes. Must be a compile-time constant. CDNA, CDNA2, CDNA3, and RDNA2 support 1, 2, and 4. CDNA4 additionally supports 12 and 16.
`vindex`	Per-lane index value.
`voffset`	Per-lane byte offset into the buffer. Each lane can provide a different value, enabling gather-style reads from the buffer.
`soffset`	Wave-uniform byte offset into the buffer.
`offset`	Unsigned compile-time byte offset. Encoded as a 12-bit unsigned immediate (valid range: `0` to `4095`).
`aux`	Cache policy bits (compile-time constant). Bit 3 enables buffer swizzling on all supported architectures. The remaining bits differ by architecture. See *Cache policy (aux parameter)*.

12.4 Cache policy (`aux` parameter)

All direct-to-LDS builtins accept an `aux` parameter whose individual bits control cache scope and temporal reuse hints. The bit layout differs by architecture generation. For the buffer-addressed builtins, bit 3 enables buffer swizzling on all supported architectures.

For background on AMD GPU cache hierarchy and coherence scopes, see the CDNA3 Instruction Set Architecture white paper and the Instruction Set Architecture (ISA) reference guides published on GPUOpen.

In a single-GPU kernel, wave scope with temporal reuse (`aux = 0`) is the typical choice.

12.4.1 CDNA3 and CDNA4

Three control bits are relevant:

- **SC0** (Scope Control 0, bit 0) and **SC1** (Scope Control 1, bit 4): together define the coherence scope.
- **NT** (Non-Temporal, bit 1): `0` = expect temporal reuse, `1` = do not expect temporal reuse.

SC1	SC0	NT	L1 cache behavior	L2 cache behavior	Last-level cache behavior
0	0	0	Hit LRU	Hit LRU	Hit LRU
0	0	1	Miss Evict	Hit Stream	Hit Evict
0	1	0	Hit LRU / Miss LRU[1]	Hit LRU	Hit Evict / Hit LRU[1]
0	1	1	Miss Evict	Hit Stream	Hit Evict
1	0	0	Miss Evict	Hit LRU (1 L2) or coherent cache bypass (>1 L2)	Hit LRU
1	0	1	Miss Evict	Hit Stream (1 L2) or coherent cache bypass (>1 L2)	Hit Evict
1	1	0	Miss Evict	Coherent cache bypass	Hit LRU
1	1	1	Miss Evict	Coherent cache bypass	Hit Evict

The following terms describe cache line behavior in the table above:

- **Hit LRU** – cache the data and replace the least recently used line when the cache is full. This is the normal temporal caching behavior.
- **Miss Evict** – bypass the cache level; fetch data directly from the next level.
- **Hit Stream** – on a cache hit, serve the data and evict the line; on a cache miss, fetch the data into the cache, use it once, then discard it.
- **Hit Evict** – on a cache hit, immediately discard the line after serving the request.
- **Coherent cache bypass** – when reading a cache line: if the line is modified, write it back to memory and reissue the read; if not modified, discard it and fetch a new line.

12.4.2 CDNA and CDNA2

One control bit is relevant:

- **GLC** (Global Level Coherent, bit 0): controls the first-level cache (L1).

GLC	L1 cache behavior	Notes
0	The load can read data from the L1 cache.	
1	The load intentionally misses the L1 cache and reads from L2. If a matching line exists in L1, it is invalidated and L2 is re-read.	Depending on alignment, L2 might not be re-read for every lane in the same wavefront. If the address is aligned correctly, the first lane brings in the line from L2 (or beyond), and all other lanes in the wavefront read from the same L1 cache line.

[1] When compiled with `tgsplit+`.

12.4.3 RDNA2

Three control bits are relevant:

- **GLC** (Global Level Coherent, bit 0): controls the first-level cache (L0).
- **SLC** (System Level Coherent, bit 1) and **DLC** (Device Level Coherent, bit 2): together control the behavior of L1 and L2.

GLC behavior:

GLC	L0 cache behavior	Notes
0	The load can read data from the L0 cache.	
1	The load intentionally misses the L0 cache and reads from L2. If a matching line exists in L0, it is invalidated and L2 is re-read.	Depending on alignment, L2 might not be re-read for every lane in the same wavefront. If the address is aligned correctly, the first lane brings in the line from L2 (or beyond), and all other lanes in the wavefront read from the same L0 cache line.

SLC and DLC behavior:

SLC	DLC	L2 cache	L1 cache
0	0	Hit LRU	Hit LRU
0	1	Hit LRU	Miss Evict
1	0	Hit Stream	Hit LRU
1	1	Hit No Allocate	Miss Evict

The following terms describe cache line behavior in the table above:

- **Hit LRU** – cache the data and replace the least recently used line when the cache is full. This is the normal temporal caching behavior.
- **Miss Evict** – bypass the cache level; fetch data directly from the next level.
- **Hit Stream** – on a cache hit, serve the data and evict the line; on a cache miss, fetch the data into the cache, use it once, then discard it.
- **Hit No Allocate** – on a cache hit, serve the data and evict the line; on a cache miss, fetch the data directly from the next memory level without allocating a cache line.

Tip

When choosing between *Hit Stream* and *Hit No Allocate*, consider the spatial locality of the access pattern. For contiguous or regularly-strided reads, prefer *Hit Stream*: even though each cache line is discarded after one use, a single memory fetch serves multiple lanes that share the same line, so the fetch cost is amortized. For scattered or random reads, prefer *Hit No Allocate*: each fetch is unlikely to benefit more than one lane, so skipping cache allocation avoids polluting L2 with lines that will never be reused.

CHAPTER

THIRTEEN

WAVEFRONT-LEVEL BUILTINS FOR AMD GPUS

Wavefront-level operations give you direct access to the hardware mechanisms that move and aggregate data within a wavefront. The builtins in this topic work across both CDNA (AMD Instinct) and RDNA (AMD Radeon) architectures, and cover three areas: lane operations, wavefront reductions, and wavefront voting.

13.1 Architecture availability

Most wavefront-level builtins are available on all AMD Instinct (CDNA) and AMD Radeon (RDNA) architectures. Some builtins — particularly Data Parallel Primitives (DPP) DPP8, cross-row permutations, and ballot width variants — are limited to specific generations. For per-builtin availability, see the architecture tables in the *reference pages* below.

The complete source file is available for download:

- `wavefront_builtins.hip`

13.2 Lane operations

Within a wavefront, lanes execute the same instruction simultaneously, but each holds its own register values. Many algorithms require lanes to exchange or replicate those values - to share a computed result, apply a cyclic shift, or reorganize data before the next computation step. The builtins in this topic cover the distinct ways of expressing that communication: reading from a named lane, moving data in a cyclic pattern, applying a compile-time-fixed permutation, and broadcasting within a fixed-size group. Each fills a different niche, and choosing the right one affects both correctness and performance.

13.2.1 Broadcasting from a specific lane with `readlane`

When a single lane holds a value that every other lane needs - a wavefront-wide maximum, a shared configuration parameter, a count computed by one thread - the most direct way to share it is `__builtin_amdgcn_readlane`. Unlike a shuffle, which requires every lane to participate with a relative offset, `readlane` reads from one named lane regardless of who is calling, making it the natural choice when the source is fixed rather than relative. The example here uses it to broadcast a wavefront maximum after a `__shfl_down` reduction, which is a representative case: one lane holds the answer, and all others need it.

`__builtin_amdgcn_readlane(val, lane)` returns the value of `val` held by the lane specified by `lane`. Unlike `readfirstlane`, which always reads the first active lane, `readlane` accepts a runtime lane index. The lane index must be uniform — the same value across all lanes in the wavefront at the point of the call. If the index is derived from per-lane data, use `readfirstlane` to promote it to a uniform value first. `readfirstlane` reads from the first active lane of the wavefront, which is a deterministic choice, so every lane in the wavefront receives the same index value and the uniformity requirement is satisfied.

```
__global__ void broadcast_readlane(const int *input, int *output, int n)
{
```

(continues on next page)

(continued from previous page)

```
    int gid  = blockIdx.x * blockDim.x + threadIdx.x;
    int lane = threadIdx.x % warpSize;
    int val  = (gid < n) ? input[gid] : 0;

    // Find the warp maximum via __shfl_down reduction.
    int max_val = val;
    for (int offset = warpSize / 2; offset > 0; offset >>= 1)
    {
        max_val = max(max_val, __shfl_down(max_val, offset));
    }

    // Broadcast the maximum from lane 0 to all lanes.
    // The lane index must be uniform; use readfirstlane to promote a
    // runtime-derived index to a uniform value.
    int broadcast = __builtin_amdgcn_readlane(max_val, 0);

    if (gid < n)
    {
        output[gid] = broadcast;
    }

    (void)lane;
}
```

After a `__shfl_down` reduction, lane 0 holds the wavefront maximum. Passing the literal `0` as the lane index satisfies the uniformity requirement and broadcasts that value to every lane in the wavefront.

For the full signatures and parameter details, see *readlane* and *readfirstlane* in the shuffle and lane access reference.

13.2.2 Wavefront rotation using `mov_dpp` and `ds_bpermute`

Rotation shifts every lane's value one position around the wavefront, so lane `i` receives what lane `i - 1` held, and lane 0 wraps around to receive what the last lane held. This cyclic movement is a building block for algorithms that scan or pipeline values across lanes, such as prefix operations or producer-consumer patterns within a wavefront. Data Parallel Primitives (DPP) is the hardware-native way to express rotation on AMD GPUs, executing in a dedicated unit without consuming the general instruction pipeline. On CDNA-based GPUs, a single `wave_ror:1` instruction covers the entire wave64 atomically. On RDNA-based GPUs, where `wave_ror` is not available on gfx11xx and later, the same result requires two steps: a `row_ror:1` rotates within each 16-lane row, and `ds_bpermute` supplies the wrap-around value at the boundary of each row.

```
// wave_ror:1 (CDNA): rotates the entire warp right by one lane.
// row_ror:1 (RDNA): rotates within each row of 16; ds_bpermute fixes wrap-around.
#ifdef HAS_DPP_BROADCAST
constexpr int dpp_wave_ror1 = 0x134;
#else
constexpr int dpp_row_ror1  = 0x121;
#endif
__device__ int warp_rotate_right(int val)
{
#ifdef HAS_DPP_BROADCAST
    // CDNA: single wave_ror:1 rotates all lanes atomically.
    return __builtin_amdgcn_mov_dpp(val, dpp_wave_ror1, 0xf, 0xf, false);
#else
```

(continues on next page)

(continued from previous page)

```
    // RDNA: row_ror:1 rotates within each row of 16.
    int rotated = __builtin_amdgcn_mov_dpp(val, dpp_row_ror1, 0xf, 0xf, false);

    // ds_bpermute (byte addressing: lane k -> addr k*4) fixes up the
    // wrap-around value at lane 0 of each row.
    int lane    = threadIdx.x % warpSize;
    int addr    = (lane == 0) ? 124 : (lane == 16) ? 60 : lane * 4;
    int wrapped = __builtin_amdgcn_ds_bpermute(addr, val);

    return (lane == 0 || lane == 16) ? wrapped : rotated;
#endif
}

__global__ void rotate_warp(const int *input, int *output, int n)
{
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int val = (gid < n) ? input[gid] : 0;

    // Rotate right warpSize times, XORing each step with the step index.
    for (int i = 0; i < warpSize; ++i)
    {
        val = warp_rotate_right(val);
        val ^= i;
    }

    if (gid < n)
    {
        output[gid] = val;
    }
}
```

`ds_bpermute` uses byte addressing: to read from lane `k`, pass `k * 4` as the index. Lane 0 of the first row reads from lane 31 (address 124), and lane 0 of the second row reads from lane 15 (address 60). All other lanes keep the `row_ror` result unchanged.

For the full signatures and parameter details, see *mov_dpp* and *ds_bpermute* in the DPP and data-share permutation reference.

13.2.3 Wavefront rotation using `__shfl`

The DPP rotation above is the preferred approach on AMD hardware, but the same pattern can be expressed using `__shfl`. This is useful when the performance difference is not significant for your workload. Each lane computes its source index as `(lane - 1 + warpSize) % warpSize` and because `__shfl` accepts any uniform lane index, the modular arithmetic produces the wrap-around without the architecture-specific workaround that the DPP path requires.

```
__device__ int warp_rotate_right_shfl(int val)
{
    int lane = threadIdx.x % warpSize;
    // Rotate right: each lane reads from (lane - 1 + warpSize) % warpSize.
    return __shfl(val, (lane - 1 + warpSize) % warpSize);
}

__global__ void rotate_warp_shfl(const int *input, int *output, int n)
```

(continues on next page)

(continued from previous page)

```
{
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int val = (gid < n) ? input[gid] : 0;

    // Rotate right warpSize times, XORing each step with the step index.
    // Produces the same result as rotate_warp.
    for (int i = 0; i < warpSize; ++i)
    {
        val = warp_rotate_right_shfl(val);
        val ^= i;
    }

    if (gid < n)
    {
        output[gid] = val;
    }
}
```

The DPP and `__shfl` kernels produce identical output and can be verified against the same CPU reference.

The two paths compile to different Instruction Set Architecture (ISA) on CDNA-based GPUs: the DPP path emits `v_mov_b32_dpp wave_rol:1`, a pure register move that executes in the dedicated DPP unit without touching memory, while the `__shfl` path emits `ds_bpermute_b32`, which routes through the data-share unit. On RDNA-based GPUs, where `wave_ror` is unavailable, the DPP path uses `v_mov_b32_dpp row_ror:1` for the within-row lanes and `ds_bpermute_b32` for the cross-row fixup; the `__shfl` path uses only `ds_bpermute_b32`. The ISA output for both can be inspected by compiling with `-save-temps` and examining the generated `.s` file.

13.2.4 Lane swap using `ds_swizzle`

While `readlane` and rotation address individual lanes or shift the whole wavefront by one, some algorithms need a fixed, symmetric rearrangement of groups - for example, swapping pairs of lanes, or exchanging two halves of a tile (a fixed-size contiguous sub-group of lanes within a wavefront). `__builtin_amdgcn_ds_swizzle` is designed exactly for this: the permutation is encoded entirely in a compile-time mask, so that the hardware can apply it in a single instruction with no per-lane index computation. The mask specifies a bitwise transformation of each lane index to its source, making it well-suited to power-of-two group swaps that appear in butterfly networks, data rearrangement before matrix operations, or paired-lane exchanges. For the full signature, see *ds_swizzle* in the DPP and data-share permutation reference.

The 16-bit mask encodes the permutation: bits [14:10] hold `and_mask`, bits [9:5] hold `or_mask`, and bits [4:0] hold `xor_mask`. The source lane is computed as `(lane & and_mask) | or_mask ^ xor_mask`. Setting `and_mask = 0x1F` and `or_mask = 0`, and placing a single set bit in `xor_mask`, swaps neighboring groups whose size is determined by the position of that bit. The example demonstrates the swap pattern; the mask value and how it determines group size are explained in the prose below.

```
// ds_swizzle mask layout: bits [14:10]=and_mask, [9:5]=or_mask, [4:0]=xor_mask.
// Source lane = (lane & and_mask) | or_mask ^ xor_mask.
// 0x101F: and_mask=0x1F, xor_mask=0x04 - swaps neighboring groups of 4.
constexpr int swizzle_swap_groups4 = 0x101F;
__global__ void swizzle_swap_groups(const int *input, int *output, int n)
{
    int gid = blockIdx.x * blockDim.x + threadIdx.x;
    int val = (gid < n) ? input[gid] : 0;

    int swapped = __builtin_amdgcn_ds_swizzle(val, swizzle_swap_groups4);
```

(continues on next page)

(continued from previous page)

```

    if (gid < n)
    {
        output[gid] = swapped;
    }
}
```

The mask `0x101F` sets `xor_mask = 0x04`, which applies a bitwise XOR between each lane index and 4. Lanes 0–3 read from lanes 4–7 and vice versa, lanes 8–11 read from lanes 12–15, and so on across the wavefront. Shifting the set bit changes the group size: `0x041F` swaps neighboring pairs, `0x081F` swaps groups of two, `0x201F` swaps groups of eight, and `0x401F` swaps groups of sixteen.

For the full signature and parameter details, see *ds_swizzle* in the DPP and data-share permutation reference.

13.3 Wavefront reductions

A wavefront reduction combines a value held by each lane into a single scalar result. This pattern appears constantly in GPU kernels - summing partial products, finding a maximum across a tile, counting active threads - and the efficiency of the reduction matters because it often sits in the critical path of a kernel. The three implementations below all express the same butterfly pattern, where lanes exchange values with increasingly distant partners until one lane holds the aggregate. Still, they differ significantly in how directly they map to hardware. The first two reduce floating-point values and show the progression from shuffle-based code to hardware-native DPP. The third uses an unsigned integer reduction to demonstrate `wave_reduce_add_u32`, the single-instruction form currently available in the compiler. Understanding all three lets you choose the right level of abstraction for your target and performance requirements.

13.3.1 Reduction using `__shfl_down`

`__shfl_down` is the most readable starting point for wavefront reductions. It reads from a lane a fixed offset ahead of the current lane, which maps directly onto the butterfly pattern when called in a loop that halves the offset on each step. The code structure closely mirrors the algorithm's logical structure, making it straightforward to reason about and maintain. The kernel is templated on `WarpSize`, a compile-time constant passed at the call site, which allows the compiler to fully unroll the reduction loop and makes the code adapt to wave32 and wave64 without modification. This makes `__shfl_down` the right choice when clarity and simplicity matter, or as a reference implementation against which a more hardware-specific version can be verified.

```
template<int WarpSize>
__device__ float shfl_down_reduce(float val)
{
    // Halve the active offset each step until all lanes have contributed.
    // WarpSize is a compile-time constant, allowing the compiler to fully
    // unroll this loop.
    for (int offset = WarpSize / 2; offset > 0; offset >>= 1)
    {
        val += __shfl_down(val, offset);
    }
    return val;
}

template<int WarpSize>
__global__ __launch_bounds__(block_size)
void reduce_shfl_down(const float *input, float *output, int n)
{
```

(continues on next page)

(continued from previous page)

```
    int gid    = blockIdx.x * blockDim.x + threadIdx.x;
    int lane   = threadIdx.x % WarpSize;
    int warpid = threadIdx.x / WarpSize;

    float val = (gid < n) ? input[gid] : 0.0f;
    val = shfl_down_reduce<WarpSize>(val);

    // Collect one partial result per warp into shared memory, then reduce
    // those with the first warp.
    extern __shared__ float sdata[];
    int num_warps = blockDim.x / WarpSize;
    if (lane == 0)
    {
        sdata[warpid] = val;
    }
    __syncthreads();

    if (warpid == 0)
    {
        val = (lane < num_warps) ? sdata[lane] : 0.0f;
        val = shfl_down_reduce<WarpSize>(val);
    }

    if (threadIdx.x == 0)
    {
        output[blockIdx.x] = val;
    }
}
```

After the loop, lane 0 holds the sum of all lanes in the wavefront. The kernel uses a two-phase structure: each wavefront reduces its slice independently, lane 0 writes its partial result to shared memory, and the first wavefront reduces those partials to the block result.

13.3.2 Reduction using `mov_dpp`

When targeting supported AMD GPUs specifically, Data Parallel Primitives offer a faster path. `__builtin_amdgcn_mov_dpp` moves data between lanes according to a DPP control word, executing in a dedicated hardware unit rather than the general instruction pipeline used by shuffle operations. The same butterfly pattern as above can be expressed with DPP control words, and the result is a reduction that makes better use of the hardware's data movement capabilities. The trade-off is that DPP instructions expose architectural differences directly: broadcast instructions are available on gfx9xx targets (CDNA), but not on gfx10xx and later (RDNA, wave32), so the `HAS_DPP_BROADCAST` macro selects the appropriate path at compile time.

```
// DPP control words for butterfly reduction steps.
// quad_perm and row_ror steps work on both CDNA and RDNA.
// row_bcast steps are CDNA only; ds_swizzle mask 0x1e0 is the RDNA fallback.
constexpr int dpp_quad_perm_1032 = 0xb1;
constexpr int dpp_quad_perm_2301 = 0x4e;
constexpr int dpp_row_ror4       = 0x124;
constexpr int dpp_row_ror8       = 0x128;
#ifdef HAS_DPP_BROADCAST
constexpr int dpp_row_bcast15    = 0x142;
```

(continues on next page)

(continued from previous page)

```
constexpr int dpp_row_bcast31    = 0x143;
#else
constexpr int swizzle_bcast15    = 0x1e0;
#endif
template<int WarpSize>
__device__ float dpp_warp_reduce(float val)
{
    // Reinterpret float bits as int for DPP and swizzle intrinsics.
    auto as_int   = [](float f) { return __builtin_bit_cast(int, f); };
    auto as_float = [](int i)   { return __builtin_bit_cast(float, i); };

    // Steps 1-2: combine adjacent pairs within groups of 4.
    val += as_float(__builtin_amdgcn_mov_dpp(as_int(val), dpp_quad_perm_1032, 0xf, 0xf, false));
    val += as_float(__builtin_amdgcn_mov_dpp(as_int(val), dpp_quad_perm_2301, 0xf, 0xf, false));
    // Steps 3-4: combine groups of 4 into a row of 16.
    val += as_float(__builtin_amdgcn_mov_dpp(as_int(val), dpp_row_ror4, 0xf, 0xf, false));
    val += as_float(__builtin_amdgcn_mov_dpp(as_int(val), dpp_row_ror8, 0xf, 0xf, false));

#ifdef HAS_DPP_BROADCAST
    // Steps 5-6 (CDNA): combine two rows of 16 into the full wave64 result.
    val += as_float(__builtin_amdgcn_mov_dpp(as_int(val), dpp_row_bcast15, 0xa, 0xf, false));
    val += as_float(__builtin_amdgcn_mov_dpp(as_int(val), dpp_row_bcast31, 0xc, 0xf, false));
#else
    // Step 5 (RDNA): ds_swizzle combines the two rows of the wave32 warp.
    val += as_float(__builtin_amdgcn_ds_swizzle(as_int(val), swizzle_bcast15));
#endif

    return val;
}

template<int WarpSize>
__global__ __launch_bounds__(block_size)
void reduce_dpp(const float *input, float *output, int n)
{
    int gid    = blockIdx.x * blockDim.x + threadIdx.x;
    int lane   = threadIdx.x % WarpSize;
    int warpid = threadIdx.x / WarpSize;

    float val = (gid < n) ? input[gid] : 0.0f;
    val = dpp_warp_reduce<WarpSize>(val);

    extern __shared__ float sdata[];
    int num_warps = blockDim.x / WarpSize;
    if (lane == WarpSize - 1)
    {
        sdata[warpid] = val;
```

(continues on next page)

(continued from previous page)

```
    }
    __syncthreads();

    if (warpid == 0)
    {
        val = (lane < num_warps) ? sdata[lane] : 0.0f;
        val = dpp_warp_reduce<WarpSize>(val);
    }

    if (threadIdx.x == WarpSize - 1)
    {
        output[blockIdx.x] = val;
    }
}
```

The first two steps use `quad_perm` control words, which exchange lanes within groups of four. Steps three and four use `row_ror` rather than `row_shr`: rotation wraps lanes within the row boundary, whereas a shift leaves destination registers undefined when the source is out of range. On RDNA, where DPP broadcasts are unavailable, `ds_swizzle` with mask `0x1e0` replicates the value held by lane 15 across lanes 16–31, combining the two rows of the wave32 wavefront in a single instruction. The control word constants used in the example (`0xb1`, `0x4e`, `0x124`, and so on) are user-defined values derived from the DPP control word encoding; they are not predefined by HIP or LLVM. For the full DPP control word encoding and available patterns, refer to the Data Parallel Primitives chapter of the CDNA3 ISA or RDNA3 ISA.

13.3.3 Reduction using `wave_reduce_add_u32`

Both implementations above express the butterfly pattern explicitly in code, requiring you to select control words, handle the architecture split, and manage the reduction steps yourself. `__builtin_amdgcn_wave_reduce_add_u32` encapsulates all of that: you supply the value and a strategy hint, and the builtin handles the reduction without you needing to implement or maintain the underlying pattern. This represents the most hardware-specific end of the spectrum - the implementation details are handled by the compiler and runtime rather than by your code. Note that the wave reduce builtins currently cover integer and bitwise operations; for float workloads, the DPP path above remains the recommended approach.

```
template<int WarpSize>
__global__ __launch_bounds__(block_size)
void reduce_wave(const unsigned int *input, unsigned int *output, int n)
{
    int gid    = blockIdx.x * blockDim.x + threadIdx.x;
    int lane   = threadIdx.x % WarpSize;
    int warpid = threadIdx.x / WarpSize;

    unsigned int val = (gid < n) ? input[gid] : 0u;

    // Reduces val across all active lanes in a single instruction.
    // The result is broadcast to every lane.
    unsigned int sum = __builtin_amdgcn_wave_reduce_add_u32(val, 0);

    extern __shared__ unsigned int sdataui[];
    int num_warps = blockDim.x / WarpSize;
    if (lane == 0)
    {
        sdataui[warpid] = sum;
```

(continues on next page)

(continued from previous page)

```
    }
    __syncthreads();

    if (warpid == 0)
    {
        sum = (lane < num_warps) ? sdataui[lane] : 0u;
        sum = __builtin_amdgcn_wave_reduce_add_u32(sum, 0);
    }

    if (threadIdx.x == 0)
    {
        output[blockIdx.x] = sum;
    }
}
```

The second argument is a strategy hint: `0` lets the compiler choose, `1` requests the iterative strategy, and `2` requests the DPP-based strategy.

For the full signature and parameter details, see *wave_reduce_add_u32* in the wavefront reduction reference.

13.4 Wavefront voting

Lane operations move data between lanes, and reductions aggregate it. Wavefront voting answers a different question: which lanes satisfy a condition, and what can the wavefront do with that information collectively? The `ballot` builtin captures the answer as a bitmask - one bit per lane - which can then be inspected, counted, or used to coordinate writes. The `mbcnt` builtin builds on this by counting the number of lanes before the current one that have their bit set, giving each qualifying lane a unique sequential index. Together, ballot and mbcnt is the standard mechanism for stream compaction within a wavefront: filtering a set of values to only those that pass a predicate, and writing them contiguously to an output buffer without gaps or collisions, all without shared memory or barriers.

13.4.1 Compaction index using `ballot` and `mbcnt`

```
__global__ void compaction_index(const int *input, int *output, int n)
{
    int gid    = blockIdx.x * blockDim.x + threadIdx.x;
    int lane   = threadIdx.x % warpSize;
    int warpid = threadIdx.x / warpSize;
    int val    = (gid < n) ? input[gid] : 0;

    bool keep  = (gid < n) && (val > 0);
    int  index = 0;

#ifdef HAS_DPP_BROADCAST
    // wave64 (CDNA): chain mbcnt_lo and mbcnt_hi over the 64-bit mask.
    unsigned long long mask = __builtin_amdgcn_ballot_w64(keep);
    unsigned int lo         = static_cast<unsigned int>(mask);
    unsigned int hi         = static_cast<unsigned int>(mask >> 32);
    index                   = __builtin_amdgcn_mbcnt_lo(lo, 0);
    index                   = __builtin_amdgcn_mbcnt_hi(hi, index);
#else
    // wave32 (RDNA): mbcnt_lo over the 32-bit mask suffices.
```

(continues on next page)

(continued from previous page)

```
    unsigned int mask = __builtin_amdgcn_ballot_w32(keep);
    index             = __builtin_amdgcn_mbcnt_lo(mask, 0);
#endif

    // Compute per-warp base offsets for contiguous compacted output.
    extern __shared__ int sdatai[];
    int num_warps  = blockDim.x / warpSize;
    int warp_count = __builtin_amdgcn_readlane(index + (keep ? 1 : 0), warpSize - 1);
    if (lane == 0)
    {
        sdatai[warpid] = warp_count;
    }
    __syncthreads();

    int warp_base = 0;
    for (int w = 0; w < warpid; ++w)
    {
        warp_base += sdatai[w];
    }

    if (keep)
    {
        output[blockIdx.x * blockDim.x + warp_base + index] = val;
    }

    (void)num_warps;
}
```

`mbcnt_hi` takes the upper 32 bits of the mask and the result of `mbcnt_lo` as its accumulator, so the final index is the count of passing lanes strictly below the current lane across the full 64-bit mask. Lanes where the predicate is false receive an index, too, but should not write to the output.

For the full signatures and parameter details, see *ballot_w64* and *mbcnt_lo* / *mbcnt_hi* in the wavefront voting reference.

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx942 \
    wavefront_builtins.hip -o wavefront_builtins
./wavefront_builtins
```

> **Note**
>
> The example above targets CDNA3 (`gfx942`, wave64). Replace `--offload-arch` with the appropriate target for your GPU — for example, `gfx90a` for CDNA2 or `gfx1200` for RDNA4 (wave32). The `__shfl*` builtins work on all architectures; the DPP-specific code paths are guarded by architecture macros in the example file.

13.5 Naming convention

Wavefront-level builtins come in two families:

HIP wrappers use the `__shfl` prefix:

```
__shfl[_up|_down|_xor](val, offset_or_lane, width)
```

The suffix indicates the direction or mode: `_up` reads from a lower lane, `_down` from a higher lane, `_xor` reads from a lane whose index is the bitwise XOR of the current lane index and a mask, and no suffix reads from an absolute lane index.

Compiler builtins use the `__builtin_amdgcn_` prefix and map directly to ISA instructions:

```
__builtin_amdgcn_<operation>(args...)
```

Key operation names:

- `readlane` / `readfirstlane` / `writelane` – named lane access
- `mov_dpp` / `update_dpp` / `mov_dpp8` – Data Parallel Primitives
- `ds_swizzle` / `ds_permute` / `ds_bpermute` – data-share permutations
- `permlane16` / `permlanex16` / `permlane64` – cross-row permutations
- `ballot_w64` / `ballot_w32` – wavefront voting
- `mbcnt_lo` / `mbcnt_hi` – masked bit count (compaction index)
- `wave_reduce_<op>_<type>` – single-instruction wavefront reductions, where `<op>` is `add`, `sub`, `min`, `max`, `and`, `or`, or `xor`, and `<type>` is `u32`, `u64`, `i32`, `i64`, `b32`, or `b64`

Use the `__shfl*` family for standard shuffle operations. Use the `__builtin_amdgcn_*` intrinsics when you need a specific hardware feature (DPP, `ds_swizzle`, wavefront voting) or when the ISA instruction gives measurable performance benefit.

13.6 Wavefront builtin reference

Each reference page documents the full signature, parameter details, and architecture support for every builtin in that family.

Family	Description
Shuffle and lane access	`__shfl*` wrappers and hardware `readlane`, `readfirstlane`, `writelane`
DPP and data-share permutations	`mov_dpp`, `update_dpp`, `mov_dpp8`, `ds_swizzle`, `ds_permute`, `ds_bpermute`
Cross-row permutations	`permlane16`, `permlanex16`, `permlane64`, and runtime/swap variants
Wavefront reductions	`wave_reduce_<op>_<type>` for add, sub, min, max, and, or, xor
Wavefront voting and synchronization	`ballot`, `inverse_ballot`, `mbcnt`, `wave_barrier`, `wave_id`

13.7 Shuffle and lane access builtins

Shuffle builtins copy a value from one lane to another within a wavefront. The `__shfl*` family provides a higher-level interface, while the `__builtin_amdgcn_*` Compiler builtins map directly to AMD Instruction Set Architecture (ISA) instructions and offer lower-level control.

13.7.1 Common parameters

`val` (source value)
The value each lane provides. One lane's value is selected and returned to the requesting lane(s) based on the shuffle mode.

`width` (sub-wavefront partition size)
Divides the wavefront into independent partitions of `width` lanes. Shuffles only move data within a partition, not across partitions. Must be a power of two. Defaults to `warpSize` (the full wavefront).

`offset` / `src_lane` (source selector)
Identifies which lane to read from. The meaning depends on the shuffle variant: `__shfl` reads from an absolute lane index, `__shfl_up` and `__shfl_down` read from a relative offset, and `__shfl_xor` computes the source lane by applying a bitwise XOR between the current lane index and a mask.

`lane` (Compiler builtins)
A uniform runtime lane index. Must hold the same value across all active lanes at the point of the call.

13.7.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
`__shfl`, `__shfl_up`, `__shfl_down`, `__shfl_xor`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`readfirstlane`, `readlane`, `write-lane`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

13.7.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

13.7.3.1 HIP shuffles

These builtins provide a higher-level interface for copying values between lanes within a wavefront.

13.7.3.1.1 `__shfl`

Signature and parameters for this builtin.

```
T __shfl(T val, int src_lane, int width = warpSize);
```

Copies `val` from an absolute lane index within the current partition.

Parameter	Type	Description
`val`	T	Value to read from the source lane.
`src_lane`	int	Absolute lane index within the partition. If `src_lane >= width`, behavior is undefined.
`width`	int	Partition size (power of two). Defaults to `warpSize`.

Returns `T` – `val` from lane `src_lane` within the partition.

13.7.3.1.2 `__shfl_up`

Signature and parameters for this builtin.

```
T __shfl_up(T val, unsigned int offset, int width = warpSize);
```

Copies `val` from a lane `offset` positions behind the current lane within the partition. If the source lane would be negative (below the partition start), the current lane's own value is returned.

Parameter	Type	Description
`val`	T	Value to read.
`offset`	unsigned int	Number of lanes behind the current lane to read from.
`width`	int	Partition size. Defaults to `warpSize`.

Returns `T` – `val` from lane `current_lane - offset`, or the current lane's `val` if the source is out of range.

13.7.3.1.3 `__shfl_down`

Signature and parameters for this builtin.

```
T __shfl_down(T val, unsigned int offset, int width = warpSize);
```

Copies `val` from a lane `offset` positions ahead of the current lane within the partition. If the source lane would be beyond the partition end, the current lane's own value is returned.

Parameter	Type	Description
`val`	T	Value to read.
`offset`	unsigned int	Number of lanes ahead of the current lane to read from.
`width`	int	Partition size. Defaults to `warpSize`.

Returns `T` – `val` from lane `current_lane + offset`, or the current lane's `val` if the source is out of range.

13.7.3.1.4 `__shfl_xor`

Signature and parameters for this builtin.

```
T __shfl_xor(T val, int mask, int width = warpSize);
```

Copies `val` from the lane whose index is the bitwise XOR of the current lane index and `mask`. This is the building block for butterfly reductions.

Parameter	Type	Description
`val`	T	Value to read.
`mask`	int	XOR mask applied to the current lane index to compute the source lane.
`width`	int	Partition size. Defaults to `warpSize`.

Returns `T` – `val` from lane `current_lane ^ mask`.

13.7.3.2 Hardware lane access

These builtins map directly to AMD ISA instructions for reading and writing individual lane values.

13.7.3.2.1 `__builtin_amdgcn_readfirstlane`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_readfirstlane(
    unsigned int val);
```

Returns the value of `val` held by the first active lane of the wavefront. The result is uniform (the same value across all lanes), which makes this builtin useful for promoting a per-lane value to a uniform value for use as an index argument to `readlane` or as a scalar operand.

Parameter	Type	Description
`val`	unsigned int	The value to read from the first active lane.

Returns `unsigned int` – the value of `val` in the first active lane, broadcast to all lanes.

13.7.3.2.2 `__builtin_amdgcn_readlane`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_readlane(
    unsigned int val,
    unsigned int lane);
```

Returns the value of `val` held by the specified lane. Unlike `readfirstlane`, which always reads the first active lane, `readlane` accepts a runtime lane index. The lane index must be uniform — the same value across all lanes in the wavefront at the point of the call.

Parameter	Type	Description
`val`	unsigned int	The value to read from the specified lane.
`lane`	unsigned int	Uniform runtime lane index. Must hold the same value across all active lanes. If derived from per-lane data, use `readfirstlane` to promote it to a uniform value first.

Returns `unsigned int` – the value of `val` in the specified lane, broadcast to all lanes.

13.7.3.2.3 `__builtin_amdgcn_writelane`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_writelane(
    unsigned int val,
    unsigned int lane,
    unsigned int old);
```

Writes `val` to a single lane; all other lanes return `old`. This is the inverse of `readlane`: instead of broadcasting one lane's value to all lanes, it injects a value into one specific lane while leaving the rest unchanged.

Parameter	Type	Description
`val`	unsigned int	The value to write into the target lane.
`lane`	unsigned int	Uniform runtime lane index of the target lane.
`old`	unsigned int	The value returned by all non-target lanes.

Returns `unsigned int` – `val` in the target lane, `old` in all other lanes.

13.8 DPP and data-share permutation builtins

Data Parallel Primitives (DPP) move data between lanes using a hardware control word that encodes fixed permutation patterns (quad permute, row shift, row rotate, row broadcast, etc.). DPP instructions execute in a dedicated hardware unit, offering lower latency than general-purpose shuffle operations. The data-share permutation builtins (`ds_swizzle`, `ds_permute`, `ds_bpermute`) provide complementary patterns that operate through the Local Data Share (LDS) unit without reading or writing shared memory.

13.8.1 Common parameters

DPP parameters (used by `mov_dpp` and `update_dpp`):

`dpp_ctrl` (control word)
: Encodes the permutation pattern. User-defined value derived from the DPP control word encoding in the Instruction Set Architecture (ISA) specification. Common patterns include `quad_perm` (arbitrary permutation within groups of 4), `row_shr` (shift right within a row of 16), `row_ror` (rotate right within a row), `row_bcast` (broadcast from one lane to the row, CDNA/gfx9xx only), and `wave_shr` and `wave_ror` (shift and rotate across the full wave). See the *Data Parallel Primitives* chapter of the CDNA3 ISA or RDNA3 ISA for the full encoding.

`row_mask` (row enable mask)
: 4-bit mask selecting which rows of 16 lanes participate. Each bit enables one row: bit 0 = lanes 0–15, bit 1 = lanes 16–31, bit 2 = lanes 32–47, bit 3 = lanes 48–63. Disabled rows return their own value unchanged. Pass `0xf` to enable all rows.

`bank_mask` (bank enable mask)
: 4-bit mask selecting which banks of 4 lanes participate within each row. Each bit enables one bank within the 16-lane row. Disabled banks return their own value unchanged. Pass `0xf` to enable all banks.

`bound_ctrl` (boundary control)
: Controls the value for out-of-range source lanes. When `true`, out-of-range lanes return zero. When `false`, out-of-range lanes return their own value (identity behavior).

13.8.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
`mov_dpp`, `update_dpp`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`mov_dpp8`	No	No	No	No	Yes	Yes	Yes	Yes
`ds_swizzle`, `ds_permute`, `ds_bpermute`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

13.8.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

13.8.3.1 DPP move operations

These builtins move data between lanes using DPP control words that encode fixed permutation patterns.

13.8.3.1.1 `__builtin_amdgcn_mov_dpp`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_mov_dpp(
    int  val,
    int  dpp_ctrl,
    int  row_mask,
    int  bank_mask,
   bool bound_ctrl);
```

Moves data between lanes using a DPP control word. The control word encodes the permutation pattern; the row and bank masks filter which lanes participate.

Parameter	Type	Description
`val`	int	The value each lane provides. The DPP control word determines which lane's value each destination lane receives.
`dpp_ctrl`	int	DPP control word encoding the permutation pattern.
`row_mask`	int	4-bit row enable mask. Pass `0xf` for all rows.
`bank_mask`	int	4-bit bank enable mask. Pass `0xf` for all banks.
`bound_ctrl`	bool	When `true`, out-of-range lanes return zero; when `false`, they return their own `val`.

Returns `int` – the value from the source lane determined by `dpp_ctrl`, or zero / identity for out-of-range lanes depending on `bound_ctrl`.

13.8.3.1.2 `__builtin_amdgcn_update_dpp`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_update_dpp(
    int  old,
    int  val,
    int  dpp_ctrl,
    int  row_mask,
    int  bank_mask,
   bool bound_ctrl);
```

DPP move with a fallback value for out-of-range or masked-off lanes. Lanes that are disabled by `row_mask`/`bank_mask` or whose source is out-of-range receive `old` instead of zero or their own value.

Parameter	Type	Description
`old`	int	Fallback value for disabled or out-of-range lanes.
`val`	int	The value each lane provides for the DPP operation.
`dpp_ctrl`	int	DPP control word encoding the permutation pattern.
`row_mask`	int	4-bit row enable mask. Pass `0xf` for all rows.
`bank_mask`	int	4-bit bank enable mask. Pass `0xf` for all banks.
`bound_ctrl`	bool	When `true`, out-of-range lanes return zero; when `false`, they return `old`.

Returns `int` – the DPP result for participating lanes, or `old` for disabled/out-of-range lanes.

13.8.3.1.3 `__builtin_amdgcn_mov_dpp8`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_mov_dpp8(
    unsigned int val,
    unsigned int sel);
```

DPP move using an 8-element compile-time permutation within groups of 8 lanes. The `sel` parameter encodes a 3-bit lane index for each of the 8 positions, packed into a 24-bit value.

Parameter	Type	Description
`val`	unsigned int	The value each lane provides.
`sel`	unsigned int	Compile-time permutation selector. Each group of 3 bits encodes the source lane index (0–7) within the 8-lane group for the corresponding destination lane.

Returns `unsigned int` – the value from the source lane within the 8-lane group as determined by `sel`.

13.8.3.2 Data-share permutations

These builtins permute data between lanes through the LDS unit without reading or writing shared memory.

13.8.3.2.1 `__builtin_amdgcn_ds_swizzle`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_ds_swizzle(
    int val,
    int mask);
```

Applies a fixed bitmask permutation to lane indices, returning the value held by the computed source lane. The permutation is encoded entirely in a compile-time mask, so the hardware applies it in a single instruction with no per-lane index computation.

Parameter	Type	Description
`val`	int	The value each lane provides.
`mask`	int	16-bit permutation mask. Bits [14:10] = `and_mask`, bits [9:5] = `or_mask`, bits [4:0] = `xor_mask`. The source lane for each lane `L` is: `(L & and_mask) | or_mask ^ xor_mask`.

Returns `int` – the value from the source lane computed by the bitmask permutation.

13.8.3.2.2 `__builtin_amdgcn_ds_permute`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_ds_permute(
    int index,
    int val);
```

Forward permutation: each lane writes its value to the lane given by `index`. Uses byte addressing: to write to lane `k`, pass `k * 4` as the index.

Parameter	Type	Description
`index`	int	Byte address of the destination lane. Lane `k` is addressed by passing `k * 4`.
`val`	int	The value to send to the destination lane.

Returns `int` – the value received from whichever lane targeted the current lane.

13.8.3.2.3 `__builtin_amdgcn_ds_bpermute`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_ds_bpermute(
    int index,
    int val);
```

Backward permutation: each lane reads the value from the lane specified by `index`. Uses byte addressing: to read from lane `k`, pass `k * 4` as the index.

Parameter	Type	Description
`index`	int	Byte address of the source lane. Lane `k`'s value is accessed by passing `k * 4`. Each lane can specify a different source.
`val`	int	The value each lane provides for others to read.

Returns `int` – the value of `val` from the lane specified by `index / 4`.

13.9 Cross-row permutation builtins

The permlane builtins move data across 16-lane row boundaries, complementing the Data Parallel Primitives (DPP) instructions which operate within rows. They enable cross-row communication patterns (such as combining results from different rows in a reduction, or redistributing data across the full wavefront) without going through shared memory.

13.9.1 Common parameters

`old` (fallback value)
: The value returned by lanes that are inactive or not targeted by the permutation. Present on all permlane variants except `permlane64`.

`val` (source value)
: The per-lane value to permute.

`fi` (fetch-inactive)
: When `true`, inactive lanes are included in the permutation (their last-written value is used as the source). When `false`, inactive lanes are excluded and the corresponding destination receives `old`.

`bc` (bound-control / broadcast)
: When `true`, the lane selector wraps at the row boundary, effectively broadcasting. When `false`, out-of-range selectors produce `old`.

`lanesel_lo`, `lanesel_hi` (lane selectors)
: Two 4-bit values that select which lane within each 16-lane group provides the source data for the lower and upper halves of the wavefront respectively. Only present on the compile-time selector variants (`permlane16`, `permlanex16`).

`lanesel` (runtime lane selector)
: A per-lane selector value replacing the compile-time `lanesel_lo`/ `lanesel_hi` pair. Only present on the `_var` variants.

13.9.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
`permlane16`, `permlanex16`	No	No	No	No	Yes	Yes	Yes	Yes
`permlane64`	No	No	No	No	No	Yes	Yes	No
`permlane16_var`, `permlanex16_var`	No	No	No	No	No	No	No	Yes
`permlane16_swap`, `permlane32_swap`	No	No	No	Yes	No	No	No	No

13.9.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

13.9.3.1 `__builtin_amdgcn_permlane16`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_permlane16(
    int  old,
```

(continues on next page)

(continued from previous page)

```
    int  val,
    int  lanesel_lo,
    int  lanesel_hi,
    bool fi,
    bool bc);
```

Permutes within each 16-lane group using two 4-bit selectors. Each lane in the lower 16-lane group reads from the lane at `lanesel_lo`; each lane in the upper group reads from `lanesel_hi`.

Parameter	Type	Description
`old`	int	Fallback value for inactive or out-of-range lanes.
`val`	int	Per-lane value to permute.
`lanesel_lo`	int	4-bit source lane selector for the lower 16-lane group.
`lanesel_hi`	int	4-bit source lane selector for the upper 16-lane group.
`fi`	bool	Fetch-inactive: include inactive lanes as sources.
`bc`	bool	Bound-control: wrap at row boundary.

Returns `int` – permuted value.

13.9.3.2 `__builtin_amdgcn_permlanex16`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_permlanex16(
    int  old,
    int  val,
    int  lanesel_lo,
    int  lanesel_hi,
    bool fi,
    bool bc);
```

Same as `permlane16` but crosses the 16-lane boundary: each lane reads from the *other* 16-lane group.

Parameter	Type	Description
`old`	int	Fallback value for inactive or out-of-range lanes.
`val`	int	Per-lane value to permute.
`lanesel_lo`	int	4-bit source lane selector for the lower 16-lane group.
`lanesel_hi`	int	4-bit source lane selector for the upper 16-lane group.
`fi`	bool	Fetch-inactive: include inactive lanes as sources.
`bc`	bool	Bound-control: wrap at row boundary.

Returns `int` – permuted value from the opposite 16-lane group.

13.9.3.3 `__builtin_amdgcn_permlane64`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_permlane64(
    int val);
```

Exchanges data between the two 32-lane halves of a wave64 without a selector. Lane `k` in the lower half reads from lane `k + 32` in the upper half, and vice versa.

Parameter	Type	Description
`val`	int	Per-lane value to exchange.

Returns `int` – the value from the corresponding lane in the other 32-lane half.

13.9.3.4 `__builtin_amdgcn_permlane16_var`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_permlane16_var(
    unsigned int old,
    unsigned int val,
    unsigned int lanesel,
    bool         fi,
    bool         bc);
```

`permlane16` with a runtime lane selector. Each lane independently selects its source lane via `lanesel`, rather than using a single compile-time selector for the entire group.

Parameter	Type	Description
`old`	unsigned int	Fallback value for inactive or out-of-range lanes.
`val`	unsigned int	Per-lane value to permute.
`lanesel`	unsigned int	Per-lane runtime source lane selector.
`fi`	bool	Fetch-inactive: include inactive lanes as sources.
`bc`	bool	Bound-control: wrap at row boundary.

Returns `unsigned int` – permuted value.

13.9.3.5 `__builtin_amdgcn_permlanex16_var`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_permlanex16_var(
    unsigned int old,
    unsigned int val,
    unsigned int lanesel,
    bool         fi,
    bool         bc);
```

`permlanex16` with a runtime lane selector. Same cross-row behavior as `permlanex16` but with per-lane source selection.

Parameter	Type	Description
`old`	unsigned int	Fallback value for inactive or out-of-range lanes.
`val`	unsigned int	Per-lane value to permute.
`lanesel`	unsigned int	Per-lane runtime source lane selector.
`fi`	bool	Fetch-inactive: include inactive lanes as sources.
`bc`	bool	Bound-control: wrap at row boundary.

Returns `unsigned int` – permuted value from the opposite 16-lane group.

13.9.3.6 `__builtin_amdgcn_permlane16_swap`

Signature and parameters for this builtin.

```
_Vector<2, unsigned int> __builtin_amdgcn_permlane16_swap(
    unsigned int src0,
    unsigned int src1,
    bool         fi,
    bool         bc);
```

Swaps odd and even 16-lane rows between two operands. Returns a two-element vector where element 0 is the new value for `src0` and element 1 is the new value for `src1`.

Parameter	Type	Description
`src0`	unsigned int	First operand.
`src1`	unsigned int	Second operand.
`fi`	bool	Fetch-inactive: include inactive lanes as sources.
`bc`	bool	Bound-control: wrap at row boundary.

Returns `_Vector<2, unsigned int>` – two-element vector with the swapped values.

13.9.3.7 `__builtin_amdgcn_permlane32_swap`

Signature and parameters for this builtin.

```
_Vector<2, unsigned int> __builtin_amdgcn_permlane32_swap(
    unsigned int src0,
    unsigned int src1,
    bool         fi,
    bool         bc);
```

Swaps the upper and lower 32-lane halves between two operands.

Parameter	Type	Description
`src0`	unsigned int	First operand.
`src1`	unsigned int	Second operand.
`fi`	bool	Fetch-inactive: include inactive lanes as sources.
`bc`	bool	Bound-control: wrap at row boundary.

Returns `_Vector<2, unsigned int>` – two-element vector with the swapped values.

13.10 Wavefront reduction builtins

The wavefront reduction builtins compute a single scalar result from per-lane values across all active lanes in a single builtin call. They encapsulate the butterfly reduction pattern — exchanging values with increasingly distant partners — without requiring the caller to manage Data Parallel Primitives (DPP) control words or architecture-specific code paths.

13.10.1 Common parameters

All wavefront reduction builtins share the same two-parameter interface:

`src` (per-lane input)
The value contributed by each lane. The type (`unsigned int`, `unsigned long`, `int`, or `long`) matches the builtin's type suffix.

`strategy` (reduction strategy hint)
Controls how the compiler implements the reduction:

- `0` – let the compiler choose the best strategy for the target.
- `1` – request the iterative (loop-based) strategy.
- `2` – request the DPP-based strategy.

The compiler might ignore the hint if the requested strategy is not available on the target architecture. Use `0` unless profiling shows a measurable difference.

13.10.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
All `wave_reduce_*` variants	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

13.10.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

13.10.3.1 Addition

These builtins compute the sum of per-lane values across all active lanes.

13.10.3.1.1 `__builtin_amdgcn_wave_reduce_add_u32`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_wave_reduce_add_u32(
    unsigned int src,
    int          strategy);
```

Computes the sum of `src` across all active lanes.

Returns `unsigned int` – the sum, broadcast to every lane.

13.10.3.1.2 `__builtin_amdgcn_wave_reduce_add_u64`

Signature and parameters for this builtin.

```
unsigned long __builtin_amdgcn_wave_reduce_add_u64(
    unsigned long src,
    int           strategy);
```

64-bit unsigned addition across all active lanes.

Returns `unsigned long` – the sum, broadcast to every lane.

13.10.3.2 Subtraction

These builtins compute a subtraction reduction across all active lanes.

13.10.3.2.1 `__builtin_amdgcn_wave_reduce_sub_u32`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_wave_reduce_sub_u32(
    unsigned int src,
    int          strategy);
```

Unsigned 32-bit subtraction reduction across all active lanes.

Returns `unsigned int` – the result, broadcast to every lane.

13.10.3.2.2 `__builtin_amdgcn_wave_reduce_sub_u64`

Signature and parameters for this builtin.

```
unsigned long __builtin_amdgcn_wave_reduce_sub_u64(
    unsigned long src,
    int           strategy);
```

Unsigned 64-bit subtraction reduction across all active lanes.

Returns `unsigned long` – the result, broadcast to every lane.

13.10.3.3 Minimum

These builtins find the minimum value across all active lanes.

13.10.3.3.1 `__builtin_amdgcn_wave_reduce_min_i32`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_min_i32(
    int src,
    int strategy);
```

Signed 32-bit minimum across all active lanes.

Returns `int` – the minimum value, broadcast to every lane.

13.10.3.3.2 `__builtin_amdgcn_wave_reduce_min_u32`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_wave_reduce_min_u32(
    unsigned int src,
    int          strategy);
```

Unsigned 32-bit minimum across all active lanes.

Returns `unsigned int` – the minimum value, broadcast to every lane.

13.10.3.3.3 `__builtin_amdgcn_wave_reduce_min_i64`

Signature and parameters for this builtin.

```
long __builtin_amdgcn_wave_reduce_min_i64(
    long src,
    int  strategy);
```

Signed 64-bit minimum across all active lanes.

Returns `long` – the minimum value, broadcast to every lane.

13.10.3.3.4 `__builtin_amdgcn_wave_reduce_min_u64`

Signature and parameters for this builtin.

```
unsigned long __builtin_amdgcn_wave_reduce_min_u64(
    unsigned long src,
    int           strategy);
```

Unsigned 64-bit minimum across all active lanes.

Returns `unsigned long` – the minimum value, broadcast to every lane.

13.10.3.4 Maximum

These builtins find the maximum value across all active lanes.

13.10.3.4.1 `__builtin_amdgcn_wave_reduce_max_i32`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_max_i32(
    int src,
    int strategy);
```

Signed 32-bit maximum across all active lanes.

Returns `int` – the maximum value, broadcast to every lane.

13.10.3.4.2 `__builtin_amdgcn_wave_reduce_max_u32`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_wave_reduce_max_u32(
    unsigned int src,
    int          strategy);
```

Unsigned 32-bit maximum across all active lanes.

Returns `unsigned int` – the maximum value, broadcast to every lane.

13.10.3.4.3 __builtin_amdgcn_wave_reduce_max_i64

Signature and parameters for this builtin.

```
long __builtin_amdgcn_wave_reduce_max_i64(
    long src,
    int  strategy);
```

Signed 64-bit maximum across all active lanes.

Returns `long` – the maximum value, broadcast to every lane.

13.10.3.4.4 __builtin_amdgcn_wave_reduce_max_u64

Signature and parameters for this builtin.

```
unsigned long __builtin_amdgcn_wave_reduce_max_u64(
    unsigned long src,
    int           strategy);
```

Unsigned 64-bit maximum across all active lanes.

Returns `unsigned long` – the maximum value, broadcast to every lane.

13.10.3.5 Bitwise AND

These builtins compute the bitwise AND of all active lanes.

13.10.3.5.1 __builtin_amdgcn_wave_reduce_and_b32

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_and_b32(
    int src,
    int strategy);
```

Bitwise AND of all active lanes (32-bit).

Returns `int` – the AND of all lane values, broadcast to every lane.

13.10.3.5.2 __builtin_amdgcn_wave_reduce_and_b64

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_and_b64(
    int src,
    int strategy);
```

Bitwise AND of all active lanes (64-bit).

Returns `int` – the AND of all lane values, broadcast to every lane.

13.10.3.6 Bitwise OR

These builtins compute the bitwise OR of all active lanes.

13.10.3.6.1 `__builtin_amdgcn_wave_reduce_or_b32`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_or_b32(
    int src,
    int strategy);
```

Bitwise OR of all active lanes (32-bit).

Returns `int` – the OR of all lane values, broadcast to every lane.

13.10.3.6.2 `__builtin_amdgcn_wave_reduce_or_b64`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_or_b64(
    int src,
    int strategy);
```

Bitwise OR of all active lanes (64-bit).

Returns `int` – the OR of all lane values, broadcast to every lane.

13.10.3.7 Bitwise XOR

These builtins compute the bitwise XOR of all active lanes.

13.10.3.7.1 `__builtin_amdgcn_wave_reduce_xor_b32`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_xor_b32(
    int src,
    int strategy);
```

Bitwise XOR of all active lanes (32-bit).

Returns `int` – the XOR of all lane values, broadcast to every lane.

13.10.3.7.2 `__builtin_amdgcn_wave_reduce_xor_b64`

Signature and parameters for this builtin.

```
int __builtin_amdgcn_wave_reduce_xor_b64(
    int src,
    int strategy);
```

Bitwise XOR of all active lanes (64-bit).

Returns `int` – the XOR of all lane values, broadcast to every lane.

13.11 Wavefront voting and synchronization builtins

Wavefront voting builtins answer the question “which lanes satisfy a condition?” by converting per-lane predicates into bitmasks, and vice versa. The `ballot` builtin captures a predicate as a bitmask; `mbcnt` counts the number of set bits

below the current lane to produce a compaction index. Together, ballot and mbcnt form the standard mechanism for stream compaction within a wavefront: filtering values, computing unique sequential indices, and writing contiguously to an output buffer without shared memory or barriers.

13.11.1 Common parameters

`pred` (per-lane predicate)
A boolean value evaluated independently by each lane. Lanes where `pred` is true have their corresponding bit set in the ballot mask.

`mask` (ballot mask half)
A 32-bit value representing half of a 64-bit ballot mask. `mbcnt_lo` takes the lower 32 bits; `mbcnt_hi` takes the upper 32 bits.

`base` (initial count)
An initial value added to the popcount result. For `mbcnt_lo`, pass `0`. For `mbcnt_hi`, pass the result of `mbcnt_lo` to accumulate the count across both halves of the 64-bit mask.

13.11.2 Architecture availability

The following table summarizes architecture support for each builtin.

Builtin	CDNA	CDNA2	CDNA3	CDNA4	RDNA2	RDNA3	RDNA3.5	RDNA4
`ballot_w64`, `inverse_ballot_w64`	Yes	Yes	Yes	Yes	No	No	No	No
`ballot_w32`, `inverse_ballot_w32`	No	No	No	No	Yes	Yes	Yes	Yes
`mbcnt_lo`, `mbcnt_hi`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
`wave_barrier`, `wave_id`	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

13.11.3 Builtin reference

Each builtin's full signature, parameters, and return value are documented below.

13.11.3.1 Ballot

These builtins convert per-lane predicates into bitmasks and vice versa.

13.11.3.1.1 `__builtin_amdgcn_ballot_w64`

Signature and parameters for this builtin.

```
unsigned long long __builtin_amdgcn_ballot_w64(
    bool pred);
```

Returns a 64-bit bitmask in which bit `k` is set if and only if lane `k` is active and its `pred` argument is true. This is the wave64 variant; use `ballot_w32` on wave32 targets.

Parameter	Type	Description
`pred`	bool	Per-lane predicate.

Returns `unsigned long long` – 64-bit mask of active lanes where `pred` is true.

13.11.3.1.2 `__builtin_amdgcn_ballot_w32`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_ballot_w32(
    bool pred);
```

Returns a 32-bit bitmask in which bit `k` is set if and only if lane `k` is active and its `pred` argument is true. This is the wave32 variant.

Parameter	Type	Description
`pred`	bool	Per-lane predicate.

Returns `unsigned int` – 32-bit mask of active lanes where `pred` is true.

13.11.3.1.3 `__builtin_amdgcn_inverse_ballot_w64`

Signature and parameters for this builtin.

```
bool __builtin_amdgcn_inverse_ballot_w64(
    uint64_t mask);
```

Returns `true` if the current lane's bit is set in `mask`. The inverse of `ballot_w64`: converts a bitmask back to a per-lane predicate.

Parameter	Type	Description
`mask`	uint64_t	64-bit lane mask.

Returns `bool` – `true` if the current lane's bit is set in `mask`.

13.11.3.1.4 `__builtin_amdgcn_inverse_ballot_w32`

Signature and parameters for this builtin.

```
bool __builtin_amdgcn_inverse_ballot_w32(
    uint32_t mask);
```

Returns `true` if the current lane's bit is set in `mask`. The inverse of `ballot_w32`.

Parameter	Type	Description
`mask`	uint32_t	32-bit lane mask.

Returns `bool` – `true` if the current lane's bit is set in `mask`.

13.11.3.2 Masked bit count

These builtins count set bits in a ballot mask below the current lane to produce compaction indices.

13.11.3.2.1 `__builtin_amdgcn_mbcnt_lo`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_mbcnt_lo(
    unsigned int mask,
    unsigned int base);
```

Counts the number of set bits in the lower 32 bits of a ballot mask that correspond to lanes below the current lane. Used together with `mbcnt_hi` to compute a compaction index across the full 64-bit ballot mask.

Parameter	Type	Description
`mask`	unsigned int	Lower 32 bits of the ballot mask.
`base`	unsigned int	Initial count value. Pass `0` when used as the first step.

Returns `unsigned int` – `base` plus the number of set bits in `mask` below the current lane index.

13.11.3.2.2 `__builtin_amdgcn_mbcnt_hi`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_mbcnt_hi(
    unsigned int mask,
    unsigned int base);
```

Counts the number of set bits in the upper 32 bits of a ballot mask that correspond to lanes below the current lane. Pass the result of `mbcnt_lo` as `base` to compute the full compaction index across a 64-bit ballot mask.

Parameter	Type	Description
`mask`	unsigned int	Upper 32 bits of the ballot mask.
`base`	unsigned int	Result of `mbcnt_lo` (count from the lower 32-bit half).

Returns `unsigned int` – `base` plus the number of set bits in `mask` below the current lane index, giving the total compaction index across the full 64-bit mask.

13.11.3.3 Synchronization

These builtins provide wavefront-level synchronization and identification.

13.11.3.3.1 `__builtin_amdgcn_wave_barrier`

Signature and parameters for this builtin.

```
void __builtin_amdgcn_wave_barrier();
```

Synchronizes all lanes within the wavefront. Guarantees that all lanes have reached this point before any lane proceeds past it. This is a compiler scheduling barrier, not a memory fence.

Returns `void`

13.11.3.3.2 `__builtin_amdgcn_wave_id`

Signature and parameters for this builtin.

```
unsigned int __builtin_amdgcn_wave_id();
```

Returns the wavefront's index within the workgroup.

Returns `unsigned int` – the wavefront index.

CHAPTER

FOURTEEN

MFMA BUILTINS

Matrix Fused Multiply-Add (MFMA) builtins let you issue hardware matrix multiply-accumulate operations directly from HIP device code. This reference covers both dense and sparse MFMA variants on AMD CDNA architectures (AMD Instinct GPUs). Coverage is organized first by operand density – dense or sparse – and then by CDNA generation. The CDNA4 MFMA LDS transpose load builtins are covered on a separate page.

- *Dense MFMA builtins*
  - *CDNA dense MFMA builtins*
  - *CDNA2 dense MFMA builtins*
  - *CDNA3 dense MFMA builtins*
  - *CDNA4 dense MFMA builtins*
- *Sparse MFMA builtins*
  - *CDNA and CDNA2 sparse MFMA builtins*
  - *CDNA3 sparse MFMA builtins*
  - *CDNA4 sparse MFMA builtins*
- *CDNA4 MFMA LDS transpose load builtins*
- *Common MFMA parameters*

14.1 Dense MFMA builtins

Dense Matrix Fused Multiply-Add (MFMA) builtins multiply full A and B tiles and accumulate the result into a C tile. All three operands are fully populated with data. Select a CDNA generation below for the corresponding builtin reference. For trailing parameter documentation, see *Common MFMA parameters*.

- *CDNA dense MFMA builtins*
- *CDNA2 dense MFMA builtins*
- *CDNA3 dense MFMA builtins*
- *CDNA4 dense MFMA builtins*

14.1.1 CDNA dense MFMA builtins

Matrix Fused Multiply-Add (MFMA) builtins let you issue hardware matrix multiply-accumulate operations directly from HIP device code on first-generation CDNA GPUs (`gfx908`, MI100). Each MFMA instruction multiplies a small $\boldsymbol{A}$ fragment by a small $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single wavefront of 64 lanes. The hardware delivers significantly higher throughput than an equivalent sequence of scalar fused multiply-add (FMA) instructions.

14.1.1.1 Architecture availability

The builtins on this page target CDNA (`gfx908`, MI100) exclusively. Equivalent builtins for later CDNA generations are documented on their own reference pages:

- *CDNA2 dense MFMA builtins* – CDNA2 (`gfx90a`, MI200 series)
- *CDNA3 dense MFMA builtins* – CDNA3 (`gfx942`, MI300 series)
- *CDNA4 dense MFMA builtins* – CDNA4 (`gfx950`, MI350 series)

14.1.1.2 Naming convention

All MFMA builtins follow the pattern:

```
__builtin_amdgcn_mfma_<out_type>_<M>x<N>x<K><in_type>
```

`out_type`
: Accumulator element type (`f32` or `i32`).

`M`, `N`, `K`
: Tile dimensions in elements. The instruction computes the contribution of a K-wide panel of $\boldsymbol{A}$ ($M \times K$) and a K-wide panel of $\boldsymbol{B}$ ($K \times N$) to an $M \times N$ output tile. Each instruction processes one K step; the caller loops over K to accumulate a full matrix product.

`in_type`
: Input element type (`f32`, `f16`, `bf16`, or `i8`).

14.1.1.3 Accumulator layout

Each MFMA instruction computes one or more independent $M \times N$ output tiles simultaneously across the 64 lanes of a wavefront. The number of independent tiles is called the *block count*. It depends on how the $\boldsymbol{A}$ matrix rows are distributed across lane groups:

- **Scalar-input variants** (FP32 $\boldsymbol{A}$ and $\boldsymbol{B}$): each K position occupies a separate group of M lanes, so blocks $= \frac{\text{wavefront size}}{M \times K}$.
- **Packed-input variants** (FP16, BF16, INT8 $\boldsymbol{A}$ and $\boldsymbol{B}$): all K positions are packed into the register bits of the *same* lane group, so blocks $= \frac{\text{wavefront size}}{M}$ regardless of K.

Each block is independent: the $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$/$\boldsymbol{D}$ operands of different blocks occupy distinct lane and accVGPR positions and compute separate outer products.

The total number of accVGPRs per lane scales with the block count:

$$\text{accVGPRs per lane} = \text{blocks} \times \frac{M \times N}{\text{wavefront size}}$$

> **Note**
>
> On CDNA GPUs, the $\boldsymbol{C}$ and $\boldsymbol{D}$ matrix operands must reside in *accumulation VGPRs* (accVGPRs). Unlike $\boldsymbol{A}$ and $\boldsymbol{B}$, they cannot use standard architecture VGPRs (archVGPRs). In HIP device code the transfer to archVGPRs happens automatically when assigning the builtin return value to a local variable of the appropriate vector type.
>
> Each of the 64 wavefront lanes maintains its own private accVGPR file. An accVGPR index always refers to a register within one specific lane's file; the same index in two different lanes denotes two distinct physical registers.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \leq i < M$

- j – zero-based column index within the tile, $0 \leq j < N$
- b – block index, $0 \leq b <$ blocks
- **lane** – wavefront lane that holds the element, $0 \leq$ lane < 64
- **accVGPR** – zero-based index into that lane's *private* accumulator register file; the same index in two different lanes refers to two distinct physical registers

14.1.1.3.1 32×32 layout

The 32×32 tile shape uses two blocks. All 32 accVGPRs per lane are active: accVGPRs 0—15 belong to block 0, accVGPRs 16—31 to block 1.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0 lanes 0–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=4 lanes 32–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=8 lanes 0–31	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=12 lanes 32–63	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=16 lanes 0–31	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=20 lanes 32–63	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=24 lanes 0–31	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
i=28 lanes 32–63	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Fig. 1: 32×32 **accumulator layout – 2 blocks.** Each cell shows the accVGPR index that holds output element (i, j) of block 0; block 1 adds 16. Teal cells (rows where $\lfloor \frac{i}{4} \rfloor$ is even) belong to lanes 0—31; grey cells to lanes 32—63. Column j gives the lane offset within the group. Applies to all 32×32 FP32 and INT32 MFMA builtins. See the formulas and row-to-lane table below.

Given output element (i, j) in block b:

$$\text{lane} = \left(32 \cdot \lfloor \frac{i}{4} \rfloor\right) \bmod 64 + j$$

$$\text{accVGPR} = 16b + 4\lfloor \frac{i}{8} \rfloor + (i \bmod 4)$$

Conversely, given a lane L and accVGPR index G:

$$i = \left(8 \cdot \lfloor \frac{G}{4} \rfloor\right) \bmod 32 + 4\lfloor \frac{L}{32} \rfloor + (G \bmod 4)$$

$$j = L \bmod 32$$

$$b = \lfloor \frac{G}{16} \rfloor$$

The row-to-lane mapping groups rows in bands of four. Within block 0:

Rows	Lanes	accVGPRs
0—3	0—31	0—3
4—7	32—63	0—3
8—11	0—31	4—7
12—15	32—63	4—7
16—19	0—31	8—11
20—23	32—63	8—11
24—27	0—31	12—15
28—31	32—63	12—15

Block 1 uses the same lane pattern with accVGPRs 16—31.

Builtins with a **1-block** 32×32 result (`v16float` / `v16int`, 16 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

14.1.1.3.2 16×16 layout

The 16×16 tile shape uses four blocks. The 16 accVGPRs per lane are partitioned by block: accVGPRs 0—3 for block 0, 4—7 for block 1, 8—11 for block 2, and 12—15 for block 3.

Given output element (i, j) in block b:

$$\text{lane} = 16\lfloor \frac{i}{4} \rfloor + j$$

$$\text{accVGPR} = 4b + (i \bmod 4)$$

Conversely, given lane L and accVGPR index G:

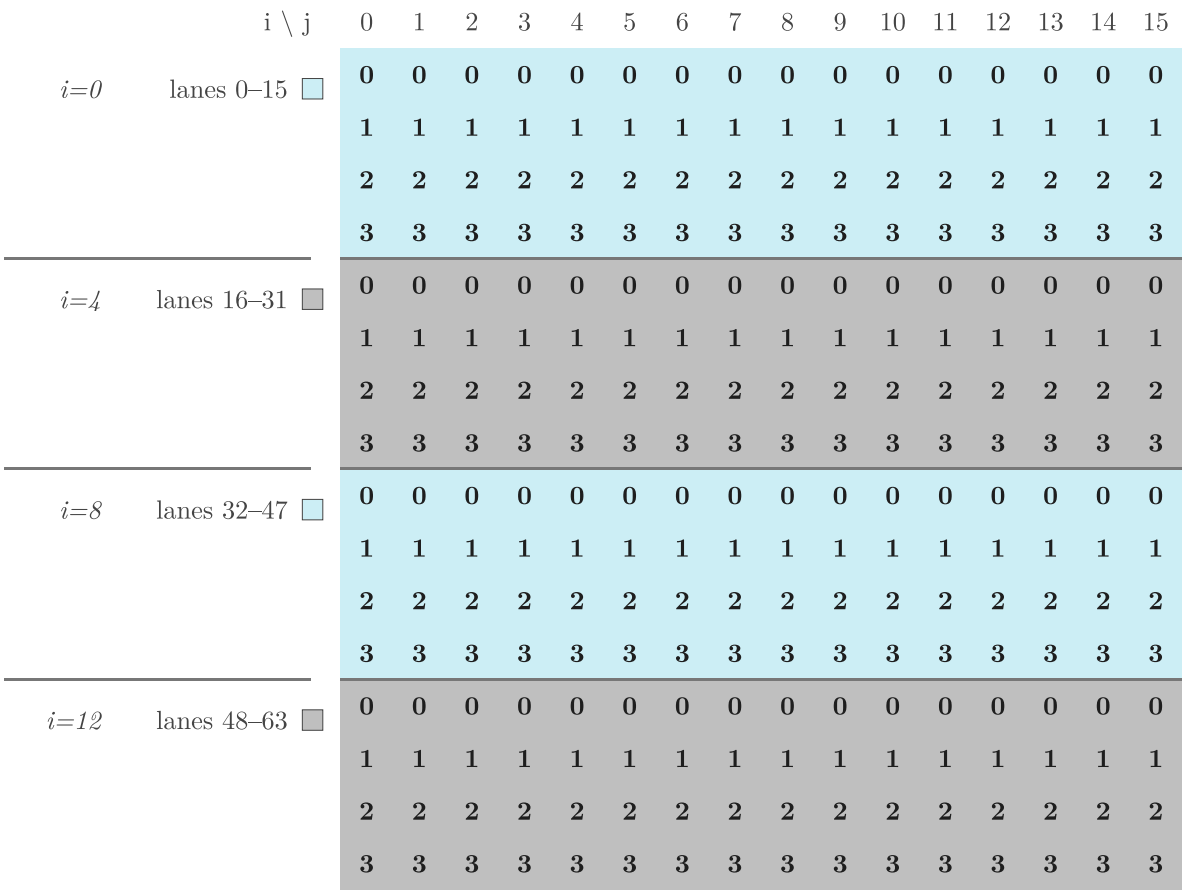
$$i = 4\lfloor \frac{L}{16} \rfloor + (G \bmod 4)$$

$$j = L \bmod 16$$

$$b = \lfloor \frac{G}{4} \rfloor$$

The row-to-lane mapping (identical for every block):

Rows	Lanes
0—3	0—15
4—7	16—31
8—11	32—47
12—15	48—63



	i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=4	lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=8	lanes 32–47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=12	lanes 48–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Fig. 2: 16×16 **accumulator layout – 4 blocks.** Each cell shows the accVGPR index for block 0; block k adds $4k$. Rows 0—3 (teal, lanes 0—15), rows 4—7 (grey, lanes 16—31), rows 8—11 (teal, lanes 32—47), rows 12—15 (grey, lanes 48—63). Column j gives the lane offset within the group. Applies to all 16×16 FP32 and INT32 MFMA builtins. 1-block variants use only block 0. See the formulas and row-to-lane table below.

Builtins with a **1-block** 16×16 result (`v4float` / `v4int`, 4 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

14.1.1.3.3 4×4 layout

The 4×4 tile shape uses 16 blocks. A single wavefront simultaneously computes 16 independent 4×4 outer products. Each group of 4 consecutive lanes (lanes $4b$ through $4b + 3$) holds all 16 output elements of block b across 4 accVGPRs.

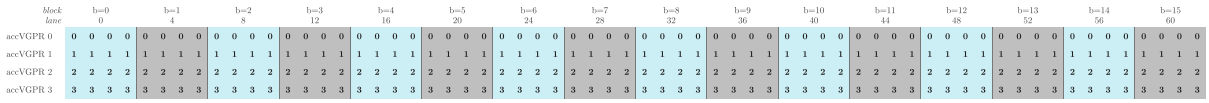


Fig. 3: 4×4 **accumulator layout – 16 blocks.** Columns are lanes 0—63; each group of 4 consecutive lanes holds one complete 4×4 output tile in accVGPRs 0—3 (rows). Teal groups are even-numbered blocks, grey groups odd-numbered blocks. Applies to all 4×4 FP32 and INT32 MFMA builtins. See the formulas below.

Given output element (i, j) in block b:

$$\begin{aligned} \text{lane} &= 4b + j \\ \text{accVGPR} &= i \end{aligned}$$

Conversely, given lane L and accVGPR index G:

$$\begin{aligned} i &= G \bmod 4 \\ j &= L \bmod 4 \\ b &= \lfloor \frac{L}{4} \rfloor \end{aligned}$$

14.1.1.4 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v4float   = float [[clang::ext_vector_type(4)]];
using v16float  = float [[clang::ext_vector_type(16)]];
using v32float  = float [[clang::ext_vector_type(32)]];
using v4half    = _Float16 [[clang::ext_vector_type(4)]];
using v4int     = int [[clang::ext_vector_type(4)]];
using v16int    = int [[clang::ext_vector_type(16)]];
using v32int    = int [[clang::ext_vector_type(32)]];
using v2bfloat  = short [[clang::ext_vector_type(2)]]; // bf16 storage
```

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane; the total VGPR count equals the element count multiplied by the element size in 32-bit words.

14.1.1.5 Common parameters

See *Common MFMA parameters* for a complete description of the `cbsz`, `abid`, and `blgp` modifiers shared by all MFMA builtins.

14.1.1.6 Using MFMA builtins as a compute policy

The matrix multiplication tutorial in *Optimizing GEMM in HIP* uses a `ComputePolicy` type parameter to separate the multiply-accumulate logic from the rest of the kernel. You can drop an MFMA-based policy into that framework without changing the outer kernel.

The example below implements `MfmaCdnaPolicy` using `v_mfma_f32_32x32x1f32` – a 32×32 FP32 builtin available on all CDNA generations. Each wavefront computes a single 32×32 output tile per `mma()` call; two blocks are active, giving 32 accVGPRs per lane.

The complete source file is available for download:

- `matrix_multiply_cdna_mfma.hip`

Policy constants

`v_mfma_f32_32x32x1f32` takes one FP32 scalar from A and one from B per call (K=1), so `k_step = 1`. The entire 32×32 tile is owned by a single wavefront, so `thread_tile_m = thread_tile_n = 32` and `effective_lanes = 64`.

Accumulator layout

The builtin returns a `v32float` holding 32 accVGPR values – 16 for block 0 and 16 for block 1. The `Accumulator` struct wraps this directly. The layout formulas from the *Accumulator layout* section translate `(lane, accVGPR)` back to (i, j) coordinates during the `store_c()` pass.

Fragment loading

Because K=1, each `load_a` call reads a single scalar: the element at row `tile_a_row + (lane mod 32)` (since 32 A-rows map to the lower 32 lanes). In practice the kernel inner loop increments `ki` in steps of `k_step = 1`, so each `load_a`/`load_b` call loads one element.

```
// MfmaCdnaPolicy is defined for all compilation passes so that the host and
// device agree on the template instantiation.  The intrinsic body inside mma()
// is guarded by __gfx908__, which is defined only during the device pass.
struct MfmaCdnaPolicy
{
    // -- ComputePolicy constants ------------------------------------------
    // One wavefront owns a 32×32 output tile.
    // All 64 lanes contribute unique output elements (no lane mirroring).
    static constexpr int thread_tile_m  = 32;
    static constexpr int thread_tile_n  = 32;
    static constexpr int effective_lanes = 64;
    // v_mfma_f32_32x32x1f32 consumes one K-index per call (K=1 per instruction).
    static constexpr int k_step          = 1;
    // MFMA takes one scalar per lane.
    static constexpr int frag_size_m     = 1;
    static constexpr int frag_size_n     = 1;

    using elem_a = float;
    using elem_b = float;

    // -- Accumulator ------------------------------------------------------
    // v32float holds 32 accVGPRs per lane:
    //   elements [0,15]  → block 0 accVGPRs 0-15
    //   elements [16,31] → block 1 accVGPRs 16-31
```

(continues on next page)

(continued from previous page)

```
using v32float = float [[clang::ext_vector_type(32)]];

struct Accumulator
{
    v32float regs;
};

__device__ static void zero(Accumulator& acc)
{
    acc.regs = v32float{};
}

// -- thread_tile_offset ----------------------------------------------------
// The entire 32×32 tile belongs to one wavefront; multiple wavefronts
// in a block cover different 32×32 sub-tiles.  Each wavefront's top-left
// corner is determined by its wavefront index within the block.
//
//   wavefront id within block: wid = tid / 64
//   wid_x = wid % (block_tile_n / 32)
//   wid_y = wid / (block_tile_n / 32)
//   *thread_row = wid_y * 32
//   *thread_col = wid_x * 32
__device__ static void thread_tile_offset(int  tid,
                                          int  /*lane_id*/,
                                          int* thread_row,
                                          int* thread_col)
{
    // block_tile_n / 32 = number of wavefronts along the N dimension.
    // For a 64×64 block tile this is 64/32 = 2.
    constexpr int waves_n = 2; // block_tile_n / thread_tile_n
    const int wid   = tid / 64;
    const int wid_x = wid % waves_n;
    const int wid_y = wid / waves_n;
    *thread_row = wid_y * 32;
    *thread_col = wid_x * 32;
}

// -- Fragment loads --------------------------------------------------------
// load_a: each lane reads its own A element from LDS.
//
// The 32×32 MFMA A-operand distributes 32 rows across the lower 32 lanes:
//   lane L in [0,31]:  provides A-row (tile_a_row + L)
//   lane L in [32,63]: mirrors the same 32 rows for the second block
// Indexing: tile_a[tile_a_row + (lane_id mod 32)][ki].
__device__ static void load_a(const float* tile_a_ptr,
                              int          tile_a_row,
                              int          ki,
                              int          k_tile_size,
                              int          lane_id,
                              elem_a     (&frag)[1])
{
    const int row = tile_a_row + (lane_id % 32);
```

(continues on next page)

(continued from previous page)

```
        frag[0] = tile_a_ptr[row * k_tile_size + ki];
    }

    // load_b: each lane reads its own B element from LDS.
    //
    // The 32×32 MFMA B-operand distributes 32 cols across the lower 32 lanes:
    //   lane L in [0,31]:  provides B-col (tile_b_col + L)
    //   lane L in [32,63]: mirrors the same 32 cols for the second block
    // tile_b_T is column-major: tile_b_T[(tile_b_col + (lane_id mod 32))][ki].
    __device__ static void load_b(const float* tile_b_T_ptr,
                                  int          tile_b_col,
                                  int          ki,
                                  int          k_tile_size,
                                  int          lane_id,
                                  elem_b     (&frag)[1])
    {
        const int col = tile_b_col + (lane_id % 32);
        frag[0] = tile_b_T_ptr[col * k_tile_size + ki];
    }

    // -- mma ---------------------------------------------------------------
    // Issue one v_mfma_f32_32x32x1f32 instruction.
    // cbsz=0: use both blocks (no A-broadcast).
    // abid=0: select block 0 as A source.
    // blgp=0: no B lane-group permutation.
    __device__ static void mma(Accumulator&       acc,
                               const elem_a     (&a_frag)[1],
                               const elem_b     (&b_frag)[1])
    {
#if defined(__gfx908__)
        acc.regs = __builtin_amdgcn_mfma_f32_32x32x1f32(
            a_frag[0], b_frag[0], acc.regs,
            /*cbsz=*/0, /*abid=*/0, /*blgp=*/0);
#endif
    }

    // -- store_c -----------------------------------------------------------
    // Scatter the 32 accVGPR values to their global-memory positions.
    //
    // Inverse layout (lane L, accVGPR index G in [0,31]):
    //   g_local = G mod 16        -- accVGPR within its block
    //   i = 8*(g_local/4) + 4*(L/32) + (g_local mod 4)
    //   j = L mod 32
    __device__ static void store_c(const Accumulator& acc,
                                   float*             C,
                                   int                out_row_base,
                                   int                out_col_base,
                                   int                m,
                                   int                n,
                                   int                lane_id)
    {
        const int L = lane_id;
```

(continues on next page)

(continued from previous page)

```
        const int j = L % 32;

        #pragma unroll
        for(int G = 0; G < 32; ++G)
        {
            const int g_local = G % 16;
            const int i = 8 * (g_local / 4)
                        + 4 * (L / 32)
                        + (g_local % 4);

            const int r = out_row_base + i;
            const int c = out_col_base + j;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[G];
        }
    }
};
```

Instantiating the kernel

With `MfmaCdnaPolicy` in hand, plug it into the generic kernel alongside any `TilePolicy` whose `block_tile_m` and `block_tile_n` are multiples of 32 and whose `k_tile_size` is a multiple of `k_step = 1`. The policy aliases and launch configuration from the example file are:

```
// TilePolicy: 64x64 block tile, 16-element K-strip, single-buffered.
using MfmaTilePolicy = SingleBufferTilePolicy<64, 64, 16>;
```

```
// Block tile dimensions shared by both dispatch paths.
constexpr int BLOCK_TILE_M = 64;
constexpr int BLOCK_TILE_N = 64;
```

```
    // Grid: one block per 64×64 output tile.
    const dim3 grid((N + BLOCK_TILE_N - 1) / BLOCK_TILE_N,
                    (M + BLOCK_TILE_M - 1) / BLOCK_TILE_M);

    float       ms;
    bool        ok;
    const char* policy_label;

    if(is_gfx908)
    {
        // CDNA (MI100): 2×2 wavefronts of 64 lanes = 256 threads per block.
        auto launch = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicy, MfmaCdnaPolicy>
                <<<grid, dim3(256)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<MfmaTilePolicy, MfmaCdnaPolicy> [gfx908]";
    }
```

(continues on next page)

(continued from previous page)

```
    else
    {
        // Other targets: scalar fallback, 8×8 thread tiles = 64 threads per block.
        auto launch = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicy, ScalarFMAPolicy<8, 8, 8>>
                <<<grid, dim3(64)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<MfmaTilePolicy, ScalarFMAPolicy> [fallback]";
    }
```

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx908 \
    matrix_multiply_cdna_mfma.hip -o mm_cdna_mfma
./mm_cdna_mfma
```

> **Note**
>
> `MfmaCdnaPolicy` requires a CDNA GPU (`gfx908`). Compile with `--offload-arch=gfx908` to select the correct architecture. On other targets the `#if defined(__gfx908__)` guard selects `ScalarFMAPolicy` automatically, so the file compiles without modification.

14.1.1.7 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency. Instructions that support vector ALU (VALU) co-execution allow the compiler to overlap matrix and vector work; the VALU co-execution cycle count gives the number of VALU cycles available during the MFMA latency window. A value of 0 means VALU co-execution is not supported.

Builtin	Ops	Cycle count	VALU co-execution cycles
`__builtin_amdgcn_mfma_f32_32x32x1f32`	4096	64	56
`__builtin_amdgcn_mfma_f32_16x16x1f32`	2048	32	24
`__builtin_amdgcn_mfma_f32_4x4x1f32`	512	8	0
`__builtin_amdgcn_mfma_f32_32x32x2f32`	4096	64	56
`__builtin_amdgcn_mfma_f32_16x16x4f32`	2048	32	24
`__builtin_amdgcn_mfma_f32_32x32x4f16`	16384	64	56
`__builtin_amdgcn_mfma_f32_16x16x4f16`	8192	32	24
`__builtin_amdgcn_mfma_f32_4x4x4f16`	2048	8	0
`__builtin_amdgcn_mfma_f32_32x32x8f16`	16384	64	56
`__builtin_amdgcn_mfma_f32_16x16x16f16`	8192	32	24
`__builtin_amdgcn_mfma_f32_32x32x2bf16`	8192	64	56
`__builtin_amdgcn_mfma_f32_16x16x2bf16`	4096	32	24
`__builtin_amdgcn_mfma_f32_4x4x2bf16`	1024	8	0
`__builtin_amdgcn_mfma_f32_32x32x4bf16`	8192	64	56
`__builtin_amdgcn_mfma_f32_16x16x8bf16`	4096	32	24
`__builtin_amdgcn_mfma_i32_32x32x4i8`	16384	64	56
`__builtin_amdgcn_mfma_i32_16x16x4i8`	8192	32	24
`__builtin_amdgcn_mfma_i32_4x4x4i8`	2048	8	0
`__builtin_amdgcn_mfma_i32_32x32x8i8`	16384	64	56
`__builtin_amdgcn_mfma_i32_16x16x16i8`	8192	32	24

14.1.1.8 Builtin reference

14.1.1.8.1 FP32-accumulate builtins

These builtins accumulate into single-precision (FP32) output fragments. They differ in the data type of the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrix inputs.

14.1.1.8.1.1 FP32 matrix inputs

The following builtins accept one FP32 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP32 output fragments.

14.1.1.8.1.2 `__builtin_amdgcn_mfma_f32_32x32x1f32`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x1f32(
    float    srcA,
    float    srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v32float	Accumulator input: 32 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v32float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.3 __builtin_amdgcn_mfma_f32_16x16x1f32

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x1f32(
    float   srcA,
    float   srcB,
    v16float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.4 __builtin_amdgcn_mfma_f32_4x4x1f32

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x1f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: 4 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.5 __builtin_amdgcn_mfma_f32_32x32x2f32

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x2f32(
    float   srcA,
    float   srcB,
    v16float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 2$ FP32 column elements of $\boldsymbol{A}$ and $K = 2$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be 0.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be 0.
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.6 __builtin_amdgcn_mfma_f32_16x16x4f32

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x4f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP32 column elements of $\boldsymbol{A}$ and $K = 4$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.7 FP16 matrix inputs

The following builtins accept four FP16 elements per lane packed into a `v4half` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into FP32 output fragments.

14.1.1.8.1.8 `__builtin_amdgcn_mfma_f32_32x32x4f16`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x4f16(
    v4half   srcA,
    v4half   srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.9 `__builtin_amdgcn_mfma_f32_16x16x4f16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x4f16(
    v4half   srcA,
    v4half   srcB,
    v16float srcC,
    int      cbsz,
```

(continues on next page)

(continued from previous page)

```
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x4f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across four accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: Four FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x8f16(
    v4half   srcA,
```

(continues on next page)

(continued from previous page)

```
    v4half   srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 8$ FP16 column elements of $\boldsymbol{A}$ and $K = 8$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.12 `__builtin_amdgcn_mfma_f32_16x16x16f16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x16f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP16 column elements of $\boldsymbol{A}$ and $K = 16$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across four accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: Four FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.13 BF16 matrix inputs

The following builtins accept two BF16 elements per lane packed into a `v2bfloat` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into FP32 output fragments.

14.1.1.8.1.14 `__builtin_amdgcn_mfma_f32_32x32x2bf16`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x2bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 2$ BF16 column elements of $\boldsymbol{A}$ and $K = 2$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
`srcB`	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.15 `__builtin_amdgcn_mfma_f32_16x16x2bf16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x2bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 2$ BF16 column elements of $\boldsymbol{A}$ and $K = 2$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
srcB	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.16 `__builtin_amdgcn_mfma_f32_4x4x2bf16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x2bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 2$ BF16 column elements of $\boldsymbol{A}$ and $K = 2$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
srcB	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: 4 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.17 `__builtin_amdgcn_mfma_f32_32x32x4bf16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x4bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
`srcB`	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.1.18 `__builtin_amdgcn_mfma_f32_16x16x8bf16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x8bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 8$ BF16 elements of $\boldsymbol{A}$ and $K = 8$ BF16 elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
`srcB`	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.2 INT32-accumulate builtins

These builtins accept four signed 8-bit integer elements per lane packed into a single `int` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into INT32 output fragments.

14.1.1.8.2.1 INT8 matrix inputs

The following builtins use INT8 matrix inputs.

14.1.1.8.2.2 `__builtin_amdgcn_mfma_i32_32x32x4i8`

Signature and parameters for this builtin.

```
v32int __builtin_amdgcn_mfma_i32_32x32x4i8(
    int    srcA,
    int    srcB,
    v32int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 32×32 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v32int	Accumulator input: 32 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.2.3 `__builtin_amdgcn_mfma_i32_16x16x4i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_16x16x4i8(
    int    srcA,
    int    srcB,
    v16int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.2.4 `__builtin_amdgcn_mfma_i32_4x4x4i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_4x4x4i8(
    int   srcA,
    int   srcB,
    v4int srcC,
    int   cbsz,
    int   abid,
    int   blgp);
```

Computes one step of a 4×4 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 4×4 INT32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.2.5 `__builtin_amdgcn_mfma_i32_32x32x8i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_32x32x8i8(
    int    srcA,
    int    srcB,
    v16int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 32×32 INT32 outer-product accumulation. Each instruction processes $K = 8$ INT8 column elements of $\boldsymbol{A}$ and $K = 8$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.1.8.2.6 `__builtin_amdgcn_mfma_i32_16x16x16i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_16x16x16i8(
    int   srcA,
    int   srcB,
    v4int srcC,
    int   cbsz,
    int   abid,
    int   blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 16$ INT8 column elements of $\boldsymbol{A}$ and $K = 16$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2 CDNA2 dense MFMA builtins

Matrix Fused Multiply-Add (MFMA) builtins let you issue hardware matrix multiply-accumulate operations directly from HIP device code on CDNA2 GPUs (`gfx90a`, MI200 series). Each MFMA instruction multiplies a small $\boldsymbol{A}$ fragment by a small $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single wavefront of 64 lanes. The hardware delivers significantly higher throughput than an equivalent sequence of scalar fused multiply-add (FMA) instructions.

CDNA2 supports the same FP32, FP16, BF16, and INT8 shapes as the first CDNA generation and adds native FP64 accumulation and high-precision BF16 variants.

14.1.2.1 Architecture availability

The builtins on this page target CDNA2 (`gfx90a`, MI200 series) exclusively. Equivalent builtins for other CDNA generations are documented on their own reference pages:

- *CDNA dense MFMA builtins* – CDNA (`gfx908`, MI100 series)
- *CDNA3 dense MFMA builtins* – CDNA3 (`gfx942`, MI300 series)
- *CDNA4 dense MFMA builtins* – CDNA4 (`gfx950`, MI350 series)

14.1.2.2 Naming convention

All MFMA builtins follow the pattern:

```
__builtin_amdgcn_mfma_<out_type>_<M>x<N>x<K><in_type>[_1k]
```

`out_type`
: Accumulator element type (`f32`, `f64`, or `i32`).

M, N, K
: Tile dimensions in elements. The instruction computes the contribution of a K-wide panel of $\boldsymbol{A}$ ($M \times K$) and a K-wide panel of $\boldsymbol{B}$ ($K \times N$) to an $M \times N$ output tile. Each instruction processes one K step; the caller loops over K to accumulate a full matrix product.

in_type
: Input element type (`f32`, `f64`, `f16`, `bf16`, or `i8`).

_1k (optional)
: Suffix present on BF16 variants that deliver 1024 Flops per cycle per compute unit (CU) peak throughput, twice the 512 Flops per cycle per CU of the non-`_1k` BF16 variants.

14.1.2.3 Accumulator layout

Each MFMA instruction computes one or more independent $M \times N$ output tiles simultaneously across the 64 lanes of a wavefront. The number of independent tiles is called the *block count*. It depends on how the $\boldsymbol{A}$ matrix rows are distributed across lane groups:

- **Scalar-input variants** (FP32 or FP64 $\boldsymbol{A}$ and $\boldsymbol{B}$): each K position occupies a separate group of M lanes, so blocks $= \frac{\text{wavefront size}}{M \times K}$.
- **Packed-input variants** (FP16, BF16, INT8 $\boldsymbol{A}$ and $\boldsymbol{B}$): all K positions are packed into the register bits of the *same* lane group, so blocks $= \frac{\text{wavefront size}}{M}$ regardless of K.

Each block is independent: the $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$/$\boldsymbol{D}$ operands of different blocks occupy distinct lane and accVGPR positions and compute separate outer products.

The total number of accVGPRs per lane scales with the block count:

$$\text{accVGPRs per lane} = \text{blocks} \times \frac{M \times N}{\text{wavefront size}}$$

Note

On CDNA2, all four operands ($\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$) can reside in either accumulation VGPRs (accVGPRs) or standard architecture VGPRs (archVGPRs). In HIP device code the compiler selects the appropriate register class automatically.

Each of the 64 wavefront lanes maintains its own private accVGPR file. An accVGPR index always refers to a register within one specific lane's file; the same index in two different lanes denotes two distinct physical registers. The layout tables and formulas in the following sections use *accVGPR index* as a logical position label; the actual register class (accVGPR or archVGPR) is chosen by the compiler and does not affect the layout.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \le i < M$
- j – zero-based column index within the tile, $0 \le j < N$
- b – block index, $0 \le b <$ blocks
- **lane** – wavefront lane that holds the element, $0 \le$ lane < 64
- **accVGPR** – zero-based index into that lane's *private* accumulator register file; the same index in two different lanes refers to two distinct physical registers

14.1.2.3.1 32×32 layout

The 32×32 tile shape uses two blocks. All 32 accVGPRs per lane are active: accVGPRs 0–15 belong to block 0, accVGPRs 16–31 to block 1.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0 lanes 0–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=4 lanes 32–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=8 lanes 0–31	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=12 lanes 32–63	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=16 lanes 0–31	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=20 lanes 32–63	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=24 lanes 0–31	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
i=28 lanes 32–63	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Fig. 4: 32×32 **accumulator layout – 2 blocks.** Each cell shows the accVGPR index that holds output element (i, j) of block 0; block 1 adds 16. Teal cells (rows where $\lfloor i/4 \rfloor$ is even) belong to lanes 0–31; grey cells to lanes 32–63. Column j gives the lane offset within the group.

Given output element (i, j) in block b:

$$\text{lane} = \left(32 \cdot \lfloor \frac{i}{4} \rfloor\right) \bmod 64 + j$$

$$\text{accVGPR} = 16b + 4\lfloor \frac{i}{8} \rfloor + (i \bmod 4)$$

Conversely, given a lane L and accVGPR index G:

$$
\begin{aligned}
i &= \left(8 \cdot \lfloor \frac{G}{4} \rfloor\right) \bmod 32 + 4\lfloor \frac{L}{32} \rfloor + (G \bmod 4) \\
j &= L \bmod 32 \\
b &= \lfloor \frac{G}{16} \rfloor
\end{aligned}
$$

The row-to-lane mapping groups rows in bands of four. Within block 0:

Rows	Lanes	accVGPRs
0—3	0—31	0—3
4—7	32—63	0—3
8—11	0—31	4—7
12—15	32—63	4—7
16—19	0—31	8—11
20—23	32—63	8—11
24—27	0—31	12—15
28—31	32—63	12—15

Block 1 uses the same lane pattern with accVGPRs 16–31.

Builtins with a **1-block** 32×32 result (`v16float` / `v16int`, 16 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

14.1.2.3.2 16×16 layout

The 16×16 tile shape uses four blocks. The 16 accVGPRs per lane are partitioned by block: accVGPRs 0—3 for block 0, 4—7 for block 1, 8—11 for block 2, and 12—15 for block 3.

Given output element (i, j) in block b:

$$
\begin{aligned}
\text{lane} &= 16\lfloor \frac{i}{4} \rfloor + j \\
\text{accVGPR} &= 4b + (i \bmod 4)
\end{aligned}
$$

Conversely, given lane L and accVGPR index G:

$$
\begin{aligned}
i &= 4\lfloor \frac{L}{16} \rfloor + (G \bmod 4) \\
j &= L \bmod 16 \\
b &= \lfloor \frac{G}{4} \rfloor
\end{aligned}
$$

The row-to-lane mapping (identical for every block):

Rows	Lanes
0—3	0—15
4—7	16—31
8—11	32—47
12—15	48—63

Builtins with a **1-block** 16×16 result (`v4float` / `v4int`, 4 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

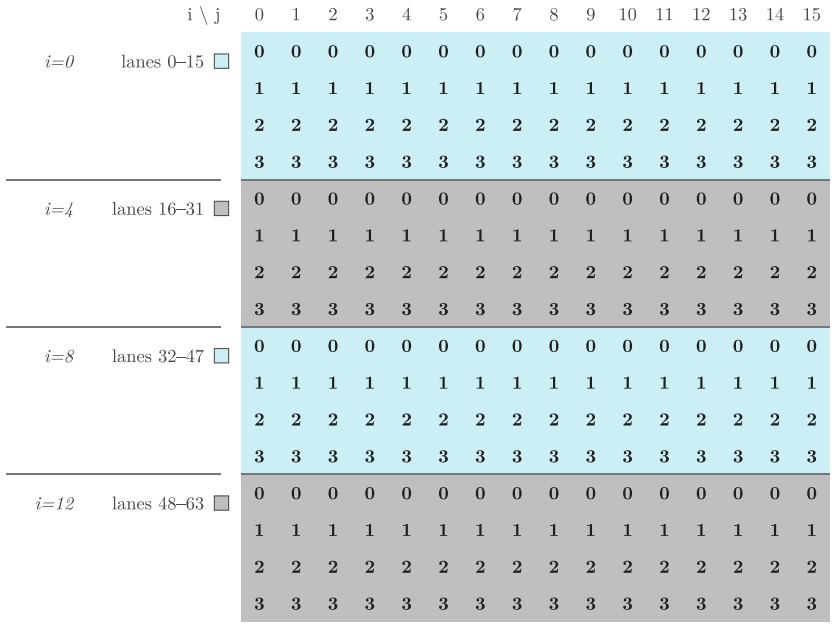


Fig. 5: 16×16 **accumulator layout – 4 blocks.** Each cell shows the accVGPR index for block 0; block k adds $4k$. Rows 0—3 (teal, lanes 0—15), rows 4—7 (grey, lanes 16—31), rows 8—11 (teal, lanes 32—47), rows 12—15 (grey, lanes 48—63). Column j gives the lane offset within the group.

14.1.2.3.3 4×4 layout

The 4×4 tile shape uses 16 blocks. A single wavefront simultaneously computes 16 independent 4×4 outer products. Each group of 4 consecutive lanes (lanes $4b$ through $4b + 3$) holds all 16 output elements of block b across 4 accVGPRs.

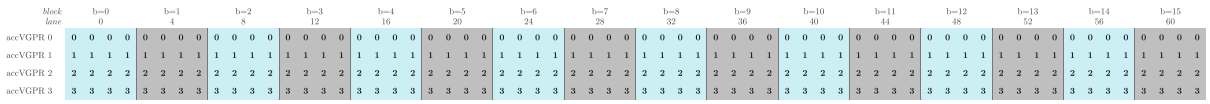


Fig. 6: 4×4 **accumulator layout – 16 blocks.** Columns are lanes 0–63; each group of 4 consecutive lanes holds one complete 4×4 output tile in accVGPRs 0–3 (rows). Teal groups are even-numbered blocks, grey groups odd-numbered blocks.

Given output element (i, j) in block b:

$$\text{lane} = 4b + j$$
$$\text{accVGPR} = i$$

Conversely, given lane L and accVGPR index G:

$$i = G \bmod 4$$
$$j = L \bmod 4$$
$$b = \lfloor \frac{L}{4} \rfloor$$

14.1.2.3.4 16×16 FP64 layout

The 16×16 FP64 tile shape uses one block. Each lane holds four FP64 output elements across 4 accVGPR pairs (8 physical accVGPRs, since each FP64 value occupies two 32-bit registers).

> **Note**
>
> FP64 accVGPR indices count *pairs* of physical registers. accVGPR pair k corresponds to physical registers `v[2k+1:2k]`.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0 lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=1 lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=2 lanes 32–47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=3 lanes 48–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=4 lanes 0–15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=5 lanes 16–31	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=6 lanes 32–47	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=7 lanes 48–63	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=8 lanes 0–15	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=9 lanes 16–31	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=10 lanes 32–47	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=11 lanes 48–63	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=12 lanes 0–15	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=13 lanes 16–31	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=14 lanes 32–47	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=15 lanes 48–63	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Fig. 7: 16×16 **FP64 accumulator layout – 1 block.** Each cell shows the accVGPR pair index k that holds output element (i, j); physical registers are `v[2k+1:2k]`. Each group of 16 consecutive lanes (column offset j) covers all four rows within one row-modulo-4 band.

Given output element (i, j):

$$\text{lane} = 16 \cdot (i \bmod 4) + j$$

$$\text{accVGPR pair } k = \lfloor \frac{i}{4} \rfloor$$

Conversely, given lane L and accVGPR pair index k:

$$i = 4k + \lfloor \frac{L}{16} \rfloor$$

$$j = L \bmod 16$$

The row-to-lane mapping. Each group of 16 consecutive lanes covers one row modulo 4; the accVGPR pair selects the row group:

Rows	Lanes	accVGPR pair (physical regs)
0, 4, 8, 12	0—15	0, 1, 2, 3 (`v[1:0]`, `v[3:2]`, `v[5:4]`, `v[7:6]`)
1, 5, 9, 13	16—31	0, 1, 2, 3
2, 6, 10, 14	32—47	0, 1, 2, 3
3, 7, 11, 15	48—63	0, 1, 2, 3

14.1.2.3.5 4×4 FP64 layout

The 4×4 FP64 tile shape uses four blocks. Each lane holds one FP64 output element in a single accVGPR pair (2 physical accVGPRs, `v[1:0]`).

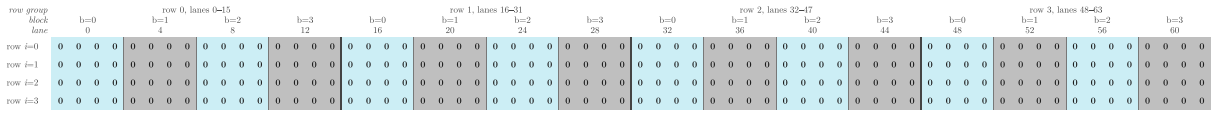


Fig. 8: 4×4 **FP64 accumulator layout – 4 blocks.** Columns are lanes 0–63; rows are matrix rows 0–3. Within each 16-lane row group, four sub-groups of 4 lanes hold blocks 0–3. Every cell holds one FP64 value in accVGPR pair 0 (`v[1:0]`). Teal sub-groups are even-numbered blocks, grey sub-groups are odd-numbered blocks.

Given output element (i, j) in block b:

$$\text{lane} = 16i + 4b + j$$

Conversely, given lane L:

$$i = \lfloor \frac{L}{16} \rfloor$$
$$j = L \bmod 4$$
$$b = \lfloor \frac{L \bmod 16}{4} \rfloor$$

The row-to-lane mapping. Lanes 0—15 hold all four blocks of row 0; the pattern repeats for rows 1—3 at lane offsets of 16, 32, and 48:

Row	Block	Lanes	accVGPR pair
0	0	0—3	0 (`v[1:0]`)
0	1	4—7	0 (`v[1:0]`)
0	2	8—11	0 (`v[1:0]`)
0	3	12—15	0 (`v[1:0]`)
1	0—3	16—31	0 (`v[1:0]`)
2	0—3	32—47	0 (`v[1:0]`)
3	0—3	48—63	0 (`v[1:0]`)

14.1.2.4 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v4float   = float [[clang::ext_vector_type(4)]];
using v16float  = float [[clang::ext_vector_type(16)]];
```

(continues on next page)

(continued from previous page)

```
using v32float  = float [[clang::ext_vector_type(32)]];
using v4half    = _Float16 [[clang::ext_vector_type(4)]];
using v4int     = int [[clang::ext_vector_type(4)]];
using v16int    = int [[clang::ext_vector_type(16)]];
using v32int    = int [[clang::ext_vector_type(32)]];
using v2bfloat  = short [[clang::ext_vector_type(2)]];  // bf16 storage
using v4bfloat  = short [[clang::ext_vector_type(4)]];  // bf16 storage
using v4double  = double [[clang::ext_vector_type(4)]];
```

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane; the total VGPR count equals the element count multiplied by the element size in 32-bit words.

14.1.2.5 Common parameters

See *Common MFMA parameters* for a complete description of the `cbsz`, `abid`, and `blgp` modifiers shared by all MFMA builtins.

14.1.2.6 Using MFMA builtins as a compute policy

The matrix multiplication tutorial in *Optimizing GEMM in HIP* uses a `ComputePolicy` type parameter to separate the multiply-accumulate logic from the rest of the kernel. You can drop an MFMA-based policy into that framework without changing the outer kernel.

The example below implements `MfmaCdna2F64Policy` using `v_mfma_f64_16x16x4f64` – a 16×16 FP64 builtin available on CDNA2 (`gfx90a`, MI200 series). Each wavefront computes a single 16×16 output tile; one block is active, giving 4 FP64 accVGPR pairs (8 physical accVGPRs) per lane.

The complete source file is available for download:

- `matrix_multiply_cdna2_mfma.hip`

Policy constants

`v_mfma_f64_16x16x4f64` consumes four K-positions per call (K=4), so `k_step = 4`. The 64 lanes are partitioned into four groups of 16, each group providing one of the four K-position inputs for both $\boldsymbol{A}$ and $\boldsymbol{B}$. The entire 16×16 tile belongs to one wavefront, so `thread_tile_m = thread_tile_n = 16` and `effective_lanes = 64`.

Accumulator layout

The builtin returns a `v4double` holding 4 FP64 values – one for each accVGPR pair. The `Accumulator` struct wraps this directly. The layout formulas from the *Accumulator layout* section translate `(lane, accVGPR pair)` back to (i, j) coordinates during the `store_c()` pass.

Fragment loading

Because K=4, each `load_a` call reads a single scalar double. The 64 lanes split into four groups of 16 by `lane_id / 16`; group g provides data for K-position `ki + g`. Within each group, the lane offset `lane_id mod 16` selects one of the 16 $\boldsymbol{A}$-rows (or $\boldsymbol{B}$-columns).

> **Note**
>
> The all-modifier-zero constraint means `cbsz`, `abid`, and `blgp` must all be passed as `0`. `v_mfma_f64_16x16x4f64` does not support the `cbsz`/`abid` broadcast mechanism or the `blgp` lane-group pattern modifier. Additionally, the instruction cannot co-execute with vector ALU (VALU) instructions in the same wavefront.

```
struct MfmaCdna2F64Policy
{
    // -- ComputePolicy constants ------------------------------------------
    // One wavefront owns a 16×16 output tile.
    // All 64 lanes hold unique output elements (4 elements per lane).
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int frag_size_m     = 1;
    static constexpr int frag_size_n     = 1;
    static constexpr int effective_lanes = 64;
    // v_mfma_f64_16x16x4f64 consumes 4 K-positions per call (k_step=4).
    // A single call with cbsz=0 uses all 64 lanes as 4 groups of 16,
    // each contributing one double for one of the 4 K-positions.
    static constexpr int k_step          = 4;

    using elem_a = double;
    using elem_b = double;

    // -- Accumulator ------------------------------------------------------
    // v4double holds 4 FP64 accVGPR pairs per lane.
    // accVGPR pair k = floor(i/4), physical regs v[2k+1:2k].
    using v4double = double [[clang::ext_vector_type(4)]];

    struct Accumulator
    {
        v4double regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
        acc.regs = v4double{};
    }

    // -- thread_tile_offset -----------------------------------------------
    // The entire 16×16 tile belongs to one wavefront.  Multiple wavefronts
    // in a block cover different 16×16 sub-tiles.
    //
    //   wavefront id within block: wid = tid / 64
    //   waves_n = block_tile_n / thread_tile_n = 32 / 16 = 2
    //   wid_x = wid % waves_n
    //   wid_y = wid / waves_n
    //   *thread_row = wid_y * 16
    //   *thread_col = wid_x * 16
    __device__ static void thread_tile_offset(int  tid,
                                              int  /*lane_id*/,
                                              int* thread_row,
                                              int* thread_col)
    {
        constexpr int waves_n = 2; // block_tile_n / thread_tile_n = 32 / 16
        const int wid   = tid / 64;
        const int wid_x = wid % waves_n;
        const int wid_y = wid / waves_n;
        *thread_row = wid_y * 16;
```

(continues on next page)

(continued from previous page)

```
    *thread_col = wid_x * 16;
}

// -- Fragment loads ------------------------------------------------------
// load_a: each lane reads one double from LDS.
//
// v_mfma_f64_16x16x4f64 (cbsz=0, abid=0, blgp=0) interprets the 64
// lanes as 4 groups of 16 (indexed by g = lane_id / 16).  Each group
// covers one of the 4 K-positions: group g provides data for ki + g.
// Within group g, lane offset (lane_id mod 16) selects one of the 16
// A-rows.
//
//   A-row index: tile_a_row + (lane_id mod 16)
//   K-position : ki + (lane_id / 16)
__device__ static void load_a(const double* tile_a_ptr,
                              int           tile_a_row,
                              int           ki,
                              int           k_tile_size,
                              int           lane_id,
                              elem_a      (&frag)[1])
{
    const int row    = tile_a_row + (lane_id % 16);
    const int k_off  = ki + (lane_id / 16);
    frag[0] = tile_a_ptr[row * k_tile_size + k_off];
}

// load_b: each lane reads one double from LDS (symmetric to load_a).
//
// tile_b_T is column-major: tile_b_T[(tile_b_col + col)][k_off].
//   B-col index: tile_b_col + (lane_id mod 16)
//   K-position : ki + (lane_id / 16)
__device__ static void load_b(const double* tile_b_T_ptr,
                              int           tile_b_col,
                              int           ki,
                              int           k_tile_size,
                              int           lane_id,
                              elem_b      (&frag)[1])
{
    const int col    = tile_b_col + (lane_id % 16);
    const int k_off = ki + (lane_id / 16);
    frag[0] = tile_b_T_ptr[col * k_tile_size + k_off];
}

// -- mma -----------------------------------------------------------------
// Issue one v_mfma_f64_16x16x4f64 instruction.
// All modifier parameters must be 0 for this instruction.
// cbsz=0, abid=0: standard 64-lane A-operand, no broadcast.
// blgp=0: no B lane-group permutation.
__device__ static void mma(Accumulator&       acc,
                           const elem_a     (&a_frag)[1],
                           const elem_b     (&b_frag)[1])
{
```

(continues on next page)

(continued from previous page)

```
#if defined(__gfx90a__)
        acc.regs = __builtin_amdgcn_mfma_f64_16x16x4f64(
            a_frag[0], b_frag[0], acc.regs,
            /*cbsz=*/0, /*abid=*/0, /*blgp=*/0);
#endif
    }

    // -- store_c ----------------------------------------------------------------
    // Scatter the 4 accVGPR pair values to their global-memory positions.
    //
    // Inverse layout (lane L, accVGPR pair index k in [0,3]):
    //   i = 4k + floor(L / 16)
    //   j = L mod 16
    __device__ static void store_c(const Accumulator& acc,
                                   double*            C,
                                   int                out_row_base,
                                   int                out_col_base,
                                   int                m,
                                   int                n,
                                   int                lane_id)
    {
        const int L = lane_id;
        const int j = L % 16;

        #pragma unroll
        for(int k = 0; k < 4; ++k)
        {
            const int i = 4 * k + (L / 16);
            const int r = out_row_base + i;
            const int c = out_col_base + j;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[k];
        }
    }
};
```

Instantiating the kernel

With `MfmaCdna2F64Policy` in hand, plug it into the generic kernel alongside a `TilePolicy` whose `block_tile_m` and `block_tile_n` are multiples of 16 and whose `k_tile_size` is a multiple of `k_step = 4`. The policy aliases and launch configuration from the example file are:

```
// TilePolicy: 32x32 block tile, 4-element K-strip, single-buffered.
using MfmaTilePolicy = SingleBufferTilePolicyD<32, 32, 4>;
```

```
// Launch parameters:
//
// CDNA2 (gfx90a): 2x2 wavefronts of 64 lanes = 256 threads per block.
//   block_tile_m = 32, thread_tile_m = 16  → 2 wavefronts along M
//   block_tile_n = 32, thread_tile_n = 16  → 2 wavefronts along N
//
// Fallback (scalar): 8x8 thread tiles in a 32x32 block → 16 threads per block,
```

(continues on next page)

(continued from previous page)

```
//   padded to one full wavefront of 64.
//
// The kernel to launch and its thread count are selected at runtime based on
// the device architecture string returned by hipGetDeviceProperties.
//
// Grid: one block per 32x32 output tile.
constexpr int BLOCK_TILE_M = 32;
constexpr int BLOCK_TILE_N = 32;
```

```
    // Grid: one block per 32×32 output tile.
    const dim3 grid((N + BLOCK_TILE_N - 1) / BLOCK_TILE_N,
                    (M + BLOCK_TILE_M - 1) / BLOCK_TILE_M);

    float       ms = 0.0f;
    const char* policy_label = nullptr;

    if(is_gfx90a)
    {
        auto launch = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicy, MfmaCdna2F64Policy>
                <<<grid, dim3(256)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<MfmaTilePolicy, MfmaCdna2F64Policy> [gfx90a]";
    }
    else
    {
        auto launch = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicy, ScalarFMADPolicy<8, 8, 8>>
                <<<grid, dim3(64)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<MfmaTilePolicy, ScalarFMADPolicy> [fallback]";
    }
```

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx90a \
    matrix_multiply_cdna2_mfma.hip -o mm_cdna2_mfma
./mm_cdna2_mfma
```

Note

`MfmaCdna2F64Policy` requires a CDNA2 GPU (`gfx90a`). Compile with `--offload-arch=gfx90a` to select the correct architecture. On other targets the `#if defined(__gfx90a__)` guard selects `ScalarFMADPolicy` automatically, so the file compiles without modification.

14.1.2.7 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency. Instructions that support VALU co-execution allow the compiler to overlap matrix and vector work; the VALU co-execution cycle count gives the number of VALU cycles available during the MFMA latency window. A value of 0 means VALU co-execution is not supported.

Builtin	Ops	Cycle count	VALU co-execution cycles
`__builtin_amdgcn_mfma_f32_32x32x1f32`	4096	64	60
`__builtin_amdgcn_mfma_f32_16x16x1f32`	2048	32	28
`__builtin_amdgcn_mfma_f32_4x4x1f32`	512	8	4
`__builtin_amdgcn_mfma_f32_32x32x2f32`	4096	64	60
`__builtin_amdgcn_mfma_f32_16x16x4f32`	2048	32	28
`__builtin_amdgcn_mfma_f32_32x32x4f16`	16384	64	60
`__builtin_amdgcn_mfma_f32_16x16x4f16`	8192	32	28
`__builtin_amdgcn_mfma_f32_4x4x4f16`	2048	8	4
`__builtin_amdgcn_mfma_f32_32x32x8f16`	16384	64	60
`__builtin_amdgcn_mfma_f32_16x16x16f16`	8192	32	28
`__builtin_amdgcn_mfma_f32_32x32x2bf16`	8192	64	60
`__builtin_amdgcn_mfma_f32_16x16x2bf16`	4096	32	28
`__builtin_amdgcn_mfma_f32_4x4x2bf16`	1024	8	4
`__builtin_amdgcn_mfma_f32_32x32x4bf16`	8192	64	60
`__builtin_amdgcn_mfma_f32_16x16x8bf16`	4096	32	28
`__builtin_amdgcn_mfma_f32_32x32x4bf16_1k`	16384	64	60
`__builtin_amdgcn_mfma_f32_16x16x4bf16_1k`	8192	32	28
`__builtin_amdgcn_mfma_f32_4x4x4bf16_1k`	2048	8	4
`__builtin_amdgcn_mfma_f32_32x32x8bf16_1k`	16384	64	60
`__builtin_amdgcn_mfma_f32_16x16x16bf16_1k`	8192	32	28
`__builtin_amdgcn_mfma_f64_16x16x4f64`	2048	32	0
`__builtin_amdgcn_mfma_f64_4x4x4f64`	512	16	0
`__builtin_amdgcn_mfma_i32_32x32x4i8`	16384	64	60
`__builtin_amdgcn_mfma_i32_16x16x4i8`	8192	32	28
`__builtin_amdgcn_mfma_i32_4x4x4i8`	2048	8	4
`__builtin_amdgcn_mfma_i32_32x32x8i8`	16384	64	60
`__builtin_amdgcn_mfma_i32_16x16x16i8`	8192	32	28

14.1.2.8 Builtin reference

14.1.2.8.1 FP32-accumulate builtins

These builtins accumulate into single-precision (FP32) output fragments. They differ in the data type of the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrix inputs.

14.1.2.8.1.1 FP32 matrix inputs

The following builtins accept one FP32 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP32 output fragments.

14.1.2.8.1.2 `__builtin_amdgcn_mfma_f32_32x32x1f32`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x1f32(
    float    srcA,
    float    srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.3 `__builtin_amdgcn_mfma_f32_16x16x1f32`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x1f32(
    float    srcA,
    float    srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.4 `__builtin_amdgcn_mfma_f32_4x4x1f32`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x1f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.5 `__builtin_amdgcn_mfma_f32_32x32x2f32`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x2f32(
    float    srcA,
    float    srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 2$ FP32 column elements of $\boldsymbol{A}$ and $K = 2$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.6 `__builtin_amdgcn_mfma_f32_16x16x4f32`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x4f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP32 column elements of $\boldsymbol{A}$ and $K = 4$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.7 FP16 matrix inputs

The following builtins accept four FP16 elements per lane packed into a `v4half` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into FP32 output fragments.

14.1.2.8.1.8 `__builtin_amdgcn_mfma_f32_32x32x4f16`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x4f16(
    v4half   srcA,
    v4half   srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v32float	Accumulator input: 32 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v32float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.9 __builtin_amdgcn_mfma_f32_16x16x4f16

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x4f16(
    v4half   srcA,
    v4half   srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x4f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across four accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: Four FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x8f16(
    v4half   srcA,
    v4half   srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 8$ FP16 column elements of $\boldsymbol{A}$ and $K = 8$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be 0.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be 0.
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.12 __builtin_amdgcn_mfma_f32_16x16x16f16

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x16f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP16 column elements of $\boldsymbol{A}$ and $K = 16$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across four accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: Four FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.13 BF16 matrix inputs

The following builtins accept two BF16 elements per lane packed into a `v2bfloat` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into FP32 output fragments.

14.1.2.8.1.14 `__builtin_amdgcn_mfma_f32_32x32x2bf16`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x2bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 2$ BF16 column elements of $\boldsymbol{A}$ and $K = 2$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
`srcB`	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.15 `__builtin_amdgcn_mfma_f32_16x16x2bf16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x2bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v16float srcC,
    int      cbsz,
```

(continues on next page)

(continued from previous page)

```
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 2$ BF16 column elements of $\boldsymbol{A}$ and $K = 2$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
srcB	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.16 __builtin_amdgcn_mfma_f32_4x4x2bf16

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x2bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 2$ BF16 column elements of $\boldsymbol{A}$ and $K = 2$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
srcB	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: 4 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.17 __builtin_amdgcn_mfma_f32_32x32x4bf16

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x4bf16(
    v2bfloat srcA,
```

(continues on next page)

(continued from previous page)

```
    v2bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
`srcB`	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.18 `__builtin_amdgcn_mfma_f32_16x16x8bf16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x8bf16(
    v2bfloat srcA,
    v2bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 8$ BF16 elements of $\boldsymbol{A}$ and $K = 8$ BF16 elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v2bfloat	Two elements of $\boldsymbol{A}$ per lane.
`srcB`	v2bfloat	Two elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

The following variants carry the `_1k` suffix.

14.1.2.8.1.19 `__builtin_amdgcn_mfma_f32_32x32x4bf16_1k`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.20 `__builtin_amdgcn_mfma_f32_16x16x4bf16_1k`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.21 `__builtin_amdgcn_mfma_f32_4x4x4bf16_1k`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.22 `__builtin_amdgcn_mfma_f32_32x32x8bf16_1k`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x8bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 8$ BF16 column elements of $\boldsymbol{A}$ and $K = 8$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.1.23 `__builtin_amdgcn_mfma_f32_16x16x16bf16_1k`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x16bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 16$ BF16 column elements of $\boldsymbol{A}$ and $K = 16$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.2 FP64-accumulate builtins

CDNA2 introduces native double-precision (FP64) matrix accumulation. These builtins accept one FP64 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP64 output fragments held in `v4double` registers. FP64 MFMA operations use IEEE round-to-nearest-even rather than round-to-zero for partial products.

> **Note**
>
> FP64 MFMA builtins do not support the `blgp` lane-group pattern modifier. Pass `blgp = 0` for all FP64 variants; any other value is reserved. Additionally, FP64 MFMA instructions cannot co-execute with VALU instructions. Schedule non-VALU instructions (loads, stores, or branches) around FP64 MFMA operations to hide execution latency.

14.1.2.8.2.1 FP64 matrix inputs

The following builtins accept one FP64 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP64 output fragments.

14.1.2.8.2.2 `__builtin_amdgcn_mfma_f64_16x16x4f64`

Signature and parameters for this builtin.

```
v4double __builtin_amdgcn_mfma_f64_16x16x4f64(
    double   srcA,
    double   srcB,
    v4double srcC,
```

(continues on next page)

(continued from previous page)

```
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP64 outer-product accumulation. Each instruction processes $K = 4$ FP64 column elements of $\boldsymbol{A}$ and $K = 4$ FP64 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP64 tile held across 4 accVGPRs per lane (8 physical accVGPRs, since each FP64 value occupies two 32-bit registers). This builtin operates on a single input block. Partial products use IEEE round-to-nearest-even rather than round-to-zero.

Parameter	Type	Description
`srcA`	double	One FP64 element of $\boldsymbol{A}$ per lane.
`srcB`	double	One FP64 element of $\boldsymbol{B}$ per lane.
`srcC`	v4double	Accumulator input: 4 FP64 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Reserved. Must be `0`.

Returns `v4double` – updated accumulator (srcA $\times$ srcB + srcC).

Note

Unlike FP32, BF16, and INT8 MFMA instructions, FP64 MFMA instructions cannot co-execute with VALU instructions. To hide execution latency, schedule non-VALU instructions (loads, stores, or branches) around FP64 MFMA operations.

14.1.2.8.2.3 `__builtin_amdgcn_mfma_f64_4x4x4f64`

Signature and parameters for this builtin.

```
double __builtin_amdgcn_mfma_f64_4x4x4f64(
    double srcA,
    double srcB,
    double srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 4×4 FP64 outer-product accumulation. Each instruction processes $K = 4$ FP64 column elements of $\boldsymbol{A}$ and $K = 4$ FP64 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP64 tile. Each lane holds one FP64 output element across four input blocks. Partial products use IEEE round-to-nearest-even rather than round-to-zero.

Parameter	Type	Description
`srcA`	double	One FP64 element of $\boldsymbol{A}$ per lane.
`srcB`	double	One FP64 element of $\boldsymbol{B}$ per lane.
`srcC`	double	Accumulator input: 1 FP64 element per lane.
`cbsz`	int	Reserved. Must be `0`.
`abid`	int	Reserved. Must be `0`.
`blgp`	int	Reserved. Must be `0`.

Returns `double` – updated accumulator (srcA × srcB + srcC).

> **Note**
>
> Unlike FP32, BF16, and INT8 MFMA instructions, FP64 MFMA instructions cannot co-execute with VALU instructions. To hide execution latency, schedule non-VALU instructions (loads, stores, or branches) around FP64 MFMA operations.

14.1.2.8.3 INT32-accumulate builtins

These builtins accept four signed 8-bit integer elements per lane packed into a single `int` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into INT32 output fragments.

14.1.2.8.3.1 INT8 matrix inputs

The following builtins use INT8 matrix inputs.

14.1.2.8.3.2 `__builtin_amdgcn_mfma_i32_32x32x4i8`

Signature and parameters for this builtin.

```
v32int __builtin_amdgcn_mfma_i32_32x32x4i8(
    int    srcA,
    int    srcB,
    v32int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 32 × 32 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32 × 32 INT32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v32int	Accumulator input: 32 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32int` – updated accumulator (srcA × srcB + srcC).

14.1.2.8.3.3 `__builtin_amdgcn_mfma_i32_16x16x4i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_16x16x4i8(
    int    srcA,
    int    srcB,
    v16int srcC,
```

(continues on next page)

(continued from previous page)

```
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.3.4 `__builtin_amdgcn_mfma_i32_4x4x4i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_4x4x4i8(
    int    srcA,
    int    srcB,
    v4int  srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 4×4 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 4×4 INT32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.3.5 `__builtin_amdgcn_mfma_i32_32x32x8i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_32x32x8i8(
    int    srcA,
    int    srcB,
    v16int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 32×32 INT32 outer-product accumulation. Each instruction processes $K = 8$ INT8 column elements of $\boldsymbol{A}$ and $K = 8$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.2.8.3.6 `__builtin_amdgcn_mfma_i32_16x16x16i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_16x16x16i8(
    int   srcA,
    int   srcB,
    v4int srcC,
    int   cbsz,
    int   abid,
    int   blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 16$ INT8 column elements of $\boldsymbol{A}$ and $K = 16$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3 CDNA3 dense MFMA builtins

Matrix Fused Multiply-Add (MFMA) builtins let you issue hardware matrix multiply-accumulate operations directly from HIP device code on CDNA3 GPUs (`gfx942`, MI300 series). Each MFMA instruction multiplies a small $\boldsymbol{A}$ fragment by a small $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single wavefront of 64 lanes. The hardware delivers significantly higher throughput than an equivalent sequence of scalar fused multiply-add (FMA) instructions.

CDNA3 supports the same FP32, FP16, INT8, and FP64 shapes as CDNA2 and adds eXtended Float32 (XF32), FP8, and BF8 input formats. The BF16 and narrower-K INT8 builtins from CDNA1 and CDNA2 are not available on CDNA3; the underlying instruction set architecture (ISA) instructions have been replaced with K-doubled variants that do not yet have corresponding HIP builtins.

14.1.3.1 Architecture availability

The builtins on this page target CDNA3 (`gfx942`, MI300 series) exclusively. Equivalent builtins for other CDNA generations are documented on their own reference pages:

- *CDNA dense MFMA builtins* – CDNA (`gfx908`, MI100 series)
- *CDNA2 dense MFMA builtins* – CDNA2 (`gfx90a`, MI200 series)
- *CDNA4 dense MFMA builtins* – CDNA4 (`gfx950`, MI350 series)

14.1.3.2 Naming convention

All MFMA builtins follow the pattern:

```
__builtin_amdgcn_mfma_<out_type>_<M>x<N>x<K><in_type>
```

For FP8 and BF8 builtins, where the $\boldsymbol{A}$ and $\boldsymbol{B}$ input types can differ, the pattern is:

```
__builtin_amdgcn_mfma_<out_type>_<M>x<N>x<K>_<typeA>_<typeB>
```

`out_type`
: Accumulator element type (`f32`, `f64`, or `i32`).

`M`, `N`, `K`
: Tile dimensions in elements. The instruction computes the contribution of a K-wide panel of $\boldsymbol{A}$ ($M \times K$) and a K-wide panel of $\boldsymbol{B}$ ($K \times N$) to an $M \times N$ output tile. Each instruction processes one K step; the caller loops over K to accumulate a full matrix product.

`in_type`
: Input element type (`f32`, `f64`, `f16`, `bf16`, `i8`, or `xf32`).

`typeA`, `typeB`
: For FP8 and BF8 builtins: the element type of the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices respectively. Each is one of `fp8` (E4M3 format) or `bf8` (E5M2 format).

> **Note**
>
> CDNA3 renames several underlying ISA instructions to include an explicit block count (for example, `v_mfma_f32_32x32x1f32` becomes `v_mfma_f32_32x32x1_2b_f32` in the CDNA3 ISA). The HIP builtin names (`__builtin_amdgcn_mfma_*`) are unchanged; the rename is transparent to HIP device code.

14.1.3.3 Accumulator layout

Each MFMA instruction computes one or more independent $M \times N$ output tiles simultaneously across the 64 lanes of a wavefront. The number of independent tiles is called the *block count*. It depends on how the $\boldsymbol{A}$ matrix rows are distributed across lane groups:

- **Scalar-input variants** (FP32 or FP64 $\boldsymbol{A}$ and $\boldsymbol{B}$): each K position occupies a separate group of M lanes, so blocks $= \frac{\text{wavefront size}}{M \times K}$.
- **Packed-input variants** (FP16, BF16, XF32, INT8, FP8, BF8 $\boldsymbol{A}$ and $\boldsymbol{B}$): all K positions are packed into the register bits of the *same* lane group, so blocks $= \frac{\text{wavefront size}}{M}$ regardless of K.

Each block is independent: the $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}/\boldsymbol{D}$ operands of different blocks occupy distinct lane and accVGPR positions and compute separate outer products.

The total number of accVGPRs per lane scales with the block count:

$$\text{accVGPRs per lane} = \text{blocks} \times \frac{M \times N}{\text{wavefront size}}$$

Note

On CDNA3, all four operands ($\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$) can reside in either accumulation VGPRs (accVGPRs) or standard architecture VGPRs (archVGPRs). In HIP device code the compiler selects the appropriate register class automatically.

Each of the 64 wavefront lanes maintains its own private accVGPR file. An accVGPR index always refers to a register within one specific lane's file; the same index in two different lanes denotes two distinct physical registers. The layout tables and formulas in the following sections use *accVGPR index* as a logical position label; the actual register class (accVGPR or archVGPR) is chosen by the compiler and does not affect the layout.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \le i < M$
- j – zero-based column index within the tile, $0 \le j < N$
- b – block index, $0 \le b <$ blocks
- **lane** – wavefront lane that holds the element, $0 \le \text{lane} < 64$
- **accVGPR** – zero-based index into that lane's *private* accumulator register file; the same index in two different lanes refers to two distinct physical registers

14.1.3.3.1 32×32 layout

The 32×32 tile shape uses two blocks. All 32 accVGPRs per lane are active: accVGPRs 0–15 belong to block 0, accVGPRs 16–31 to block 1.

Given output element (i, j) in block b:

$$\text{lane} = \left(32 \cdot \lfloor \frac{i}{4} \rfloor\right) \bmod 64 + j$$

$$\text{accVGPR} = 16b + 4\lfloor \frac{i}{8} \rfloor + (i \bmod 4)$$

i \ j		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0	lanes 0–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=4	lanes 32–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=8	lanes 0–31	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
		5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
		6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
		7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=12	lanes 32–63	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
		5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
		6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
		7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=16	lanes 0–31	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
		9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
		10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
		11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=20	lanes 32–63	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
		9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
		10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
		11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=24	lanes 0–31	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
		13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
		14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
		15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
i=28	lanes 32–63	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
		13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
		14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
		15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Fig. 9: 32×32 **accumulator layout – 2 blocks.** Each cell shows the accVGPR index that holds output element (i, j) of block 0; block 1 adds 16. Teal cells (rows where $\lfloor i/4 \rfloor$ is even) belong to lanes 0–31; grey cells to lanes 32–63. Column j gives the lane offset within the group.

Conversely, given a lane L and accVGPR index G:

$$i = \left(8 \cdot \lfloor \frac{G}{4} \rfloor\right) \bmod 32 + 4\lfloor \frac{L}{32} \rfloor + (G \bmod 4)$$
$$j = L \bmod 32$$
$$b = \lfloor \frac{G}{16} \rfloor$$

The row-to-lane mapping groups rows in bands of four. Within block 0:

Rows	Lanes	accVGPRs
0—3	0—31	0—3
4—7	32—63	0—3
8—11	0—31	4—7
12—15	32—63	4—7
16—19	0—31	8—11
20—23	32—63	8—11
24—27	0—31	12—15
28—31	32—63	12—15

Block 1 uses the same lane pattern with accVGPRs 16–31.

Builtins with a **1-block** 32×32 result (`v16float` / `v16int`, 16 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

14.1.3.3.2 16×16 layout

The 16×16 tile shape uses four blocks. The 16 accVGPRs per lane are partitioned by block: accVGPRs 0—3 for block 0, 4—7 for block 1, 8—11 for block 2, and 12—15 for block 3.

Given output element (i, j) in block b:

$$\text{lane} = 16\lfloor \frac{i}{4} \rfloor + j$$
$$\text{accVGPR} = 4b + (i \bmod 4)$$

Conversely, given lane L and accVGPR index G:

$$i = 4\lfloor \frac{L}{16} \rfloor + (G \bmod 4)$$
$$j = L \bmod 16$$
$$b = \lfloor \frac{G}{4} \rfloor$$

The row-to-lane mapping (identical for every block):

Rows	Lanes
0—3	0—15
4—7	16—31
8—11	32—47
12—15	48—63

Builtins with a **1-block** 16×16 result (`v4float` / `v4int`, 4 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

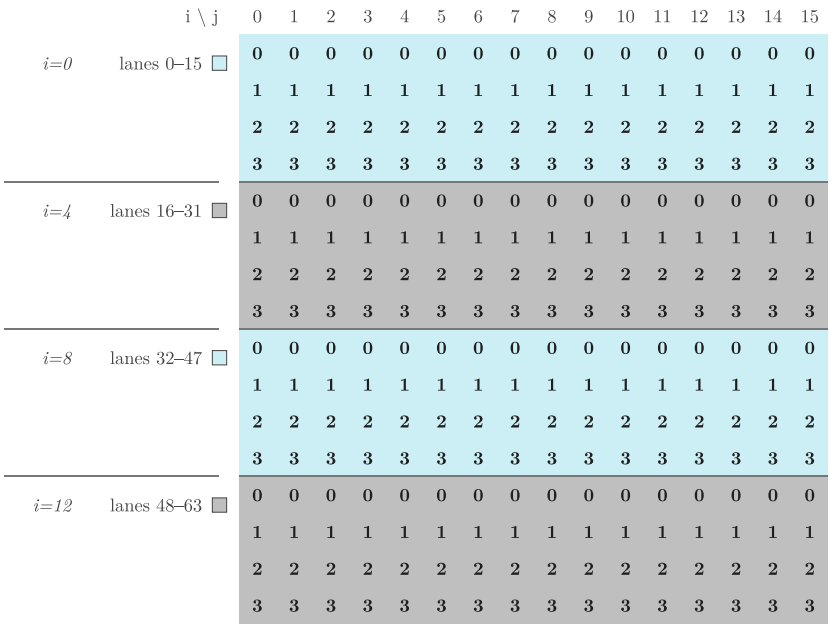


Fig. 10: 16×16 **accumulator layout – 4 blocks.** Each cell shows the accVGPR index for block 0; block k adds $4k$. Rows 0—3 (teal, lanes 0—15), rows 4—7 (grey, lanes 16—31), rows 8—11 (teal, lanes 32—47), rows 12—15 (grey, lanes 48—63). Column j gives the lane offset within the group.

14.1.3.3.3 4×4 layout

The 4×4 tile shape uses 16 blocks. A single wavefront simultaneously computes 16 independent 4×4 outer products. Each group of 4 consecutive lanes (lanes $4b$ through $4b + 3$) holds all 16 output elements of block b across 4 accVGPRs.

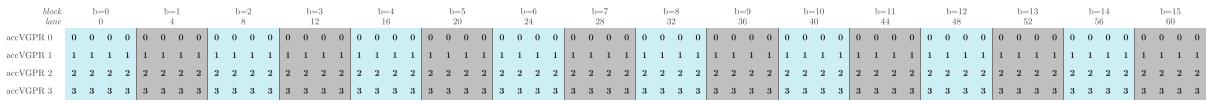


Fig. 11: 4×4 **accumulator layout – 16 blocks.** Columns are lanes 0–63; each group of 4 consecutive lanes holds one complete 4×4 output tile in accVGPRs 0–3 (rows). Teal groups are even-numbered blocks, grey groups odd-numbered blocks.

Given output element (i, j) in block b:

$$\text{lane} = 4b + j$$
$$\text{accVGPR} = i$$

Conversely, given lane L and accVGPR index G:

$$i = G \bmod 4$$
$$j = L \bmod 4$$
$$b = \lfloor \frac{L}{4} \rfloor$$

14.1.3.3.4 16×16 FP64 layout

The 16×16 FP64 tile shape uses one block. Each lane holds four FP64 output elements across 4 accVGPR pairs (8 physical accVGPRs, since each FP64 value occupies two 32-bit registers).

> **Note**
>
> FP64 accVGPR indices count *pairs* of physical registers. accVGPR pair k corresponds to physical registers `v[2k+1:2k]`.

i \ j		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=1	lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=2	lanes 32–47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=3	lanes 48–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=4	lanes 0–15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=5	lanes 16–31	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=6	lanes 32–47	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=7	lanes 48–63	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=8	lanes 0–15	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=9	lanes 16–31	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=10	lanes 32–47	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=11	lanes 48–63	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=12	lanes 0–15	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=13	lanes 16–31	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=14	lanes 32–47	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=15	lanes 48–63	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Fig. 12: 16×16 **FP64 accumulator layout – 1 block.** Each cell shows the accVGPR pair index k that holds output element (i, j); physical registers are `v[2k+1:2k]`. Each group of 16 consecutive lanes (column offset j) covers all four rows within one row-modulo-4 band.

Given output element (i, j):

$$\text{lane} = 16 \cdot (i \bmod 4) + j$$

$$\text{accVGPR pair } k = \lfloor \frac{i}{4} \rfloor$$

Conversely, given lane L and accVGPR pair index k:

$$i = 4k + \lfloor \frac{L}{16} \rfloor$$

$$j = L \bmod 16$$

The row-to-lane mapping. Each group of 16 consecutive lanes covers one row modulo 4; the accVGPR pair selects the row group:

Rows	Lanes	accVGPR pair (physical regs)
0, 4, 8, 12	0—15	0, 1, 2, 3 (`v[1:0]`, `v[3:2]`, `v[5:4]`, `v[7:6]`)
1, 5, 9, 13	16—31	0, 1, 2, 3
2, 6, 10, 14	32—47	0, 1, 2, 3
3, 7, 11, 15	48—63	0, 1, 2, 3

14.1.3.3.5 4×4 FP64 layout

The 4×4 FP64 tile shape uses four blocks. Each lane holds one FP64 output element in a single accVGPR pair (2 physical accVGPRs, `v[1:0]`).

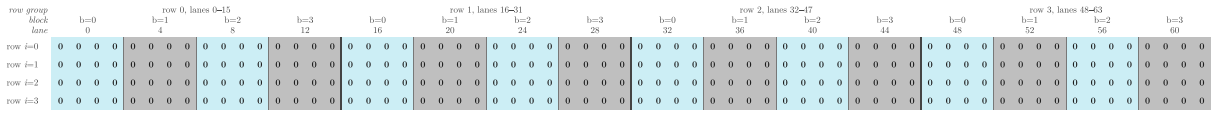


Fig. 13: 4×4 **FP64 accumulator layout – 4 blocks.** Columns are lanes 0–63; rows are matrix rows 0–3. Within each 16-lane row group, four sub-groups of 4 lanes hold blocks 0–3. Every cell holds one FP64 value in accVGPR pair 0 (`v[1:0]`). Teal sub-groups are even-numbered blocks, grey sub-groups are odd-numbered blocks.

Given output element (i, j) in block b:

$$\text{lane} = 16i + 4b + j$$

Conversely, given lane L:

$$i = \lfloor \frac{L}{16} \rfloor$$
$$j = L \bmod 4$$
$$b = \lfloor \frac{L \bmod 16}{4} \rfloor$$

The row-to-lane mapping. Lanes 0—15 hold all four blocks of row 0; the pattern repeats for rows 1—3 at lane offsets of 16, 32, and 48:

Row	Block	Lanes	accVGPR pair
0	0	0—3	0 (`v[1:0]`)
0	1	4—7	0 (`v[1:0]`)
0	2	8—11	0 (`v[1:0]`)
0	3	12—15	0 (`v[1:0]`)
1	0—3	16—31	0 (`v[1:0]`)
2	0—3	32—47	0 (`v[1:0]`)
3	0—3	48—63	0 (`v[1:0]`)

14.1.3.4 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v2float   = float [[clang::ext_vector_type(2)]];   // xf32 storage
using v4float   = float [[clang::ext_vector_type(4)]];
```

(continues on next page)

(continued from previous page)

```
using v16float  = float [[clang::ext_vector_type(16)]];
using v32float  = float [[clang::ext_vector_type(32)]];
using v4half    = _Float16 [[clang::ext_vector_type(4)]];
using v4int     = int [[clang::ext_vector_type(4)]];
using v16int    = int [[clang::ext_vector_type(16)]];
using v32int    = int [[clang::ext_vector_type(32)]];
using v2bfloat  = short [[clang::ext_vector_type(2)]];   // bf16 storage
using v4bfloat  = short [[clang::ext_vector_type(4)]];   // bf16 storage
using v4double  = double [[clang::ext_vector_type(4)]];
```

FP8 and BF8 input operands, and the wider-K INT8 operands, are passed as a 64-bit integer (`long long`) that packs eight 8-bit values per lane.

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane; the total VGPR count equals the element count multiplied by the element size in 32-bit words.

14.1.3.5 Common parameters

See *Common MFMA parameters* for a complete description of the `cbsz`, `abid`, and `blgp` modifiers shared by all MFMA builtins, including the CDNA3-specific `blgp` behaviour for FP64 builtins.

14.1.3.6 Using MFMA builtins as a compute policy

The matrix multiplication tutorial in *Optimizing GEMM in HIP* uses a `ComputePolicy` type parameter to separate the multiply-accumulate logic from the rest of the kernel. You can drop an MFMA-based policy into that framework without changing the outer kernel.

The example below implements `MfmaCdna3XF32Policy` using `v_mfma_f32_16x16x8_xf32` – a 16×16 XF32 builtin available on CDNA3 (`gfx942`, MI300 series). Each wavefront computes a single 16×16 output tile; one block is active, giving 4 FP32 accVGPRs per lane.

The complete source file is available for download:

- `matrix_multiply_cdna3_mfma.hip`

Policy constants

`v_mfma_f32_16x16x8_xf32` consumes eight K-positions per call (K=8), so `k_step = 8`. Each lane provides two FP32 values (`v2float`) for $\boldsymbol{A}$ and two for $\boldsymbol{B}$; the 64 lanes partition into four groups of 16, each group covering two K-positions. The entire 16×16 tile belongs to one wavefront, so `thread_tile_m = thread_tile_n = 16` and `effective_lanes = 64`.

Accumulator layout

The builtin returns a `v4float` holding 4 FP32 values – one for each accVGPR. The `Accumulator` struct wraps this directly. The layout formulas from the *Accumulator layout* section translate `(lane, accVGPR)` back to (i, j) coordinates during the `store_c()` pass.

Fragment loading

Because each lane provides two FP32 values for K=8, each `load_a` call reads two consecutive scalars. Lane group $g = \lfloor \text{lane_id}/16 \rfloor$ covers K-positions `ki + 2g` and `ki + 2g + 1`; within the group, lane offset `lane_id mod 16` selects one of the 16 $\boldsymbol{A}$-rows (or $\boldsymbol{B}$-columns). The `mma()` method packs the two values into a `v2float` before issuing the instruction.

> **Note**
>
> The all-modifier-zero constraint means `cbsz`, `abid`, and `blgp` must all be passed as `0`. `v_mfma_f32_16x16x8_xf32` is single-block only and does not support the `cbsz`/`abid` broadcast mechanism or the `blgp` lane-group pattern modifier.

```
struct MfmaCdna3XF32Policy
{
    // -- ComputePolicy constants ------------------------------------------
    // One wavefront owns a 16×16 output tile.
    // All 64 lanes hold unique output elements (4 elements per lane).
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int frag_size_m     = 2;
    static constexpr int frag_size_n     = 2;
    static constexpr int effective_lanes = 64;
    // v_mfma_f32_16x16x8_xf32 consumes 8 K-positions per call (k_step=8).
    // Each lane provides two FP32 values (v2float) for A and two for B.
    // The 64 lanes split into 4 groups of 16; each group covers 2 K-positions.
    static constexpr int k_step          = 8;

    using elem_a = float;
    using elem_b = float;

    // -- Accumulator ------------------------------------------------------
    // v4float holds 4 FP32 accVGPRs per lane (accVGPRs 0-3).
    using v4float = float [[clang::ext_vector_type(4)]];

    struct Accumulator
    {
        v4float regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
        acc.regs = v4float{};
    }

    // -- thread_tile_offset -----------------------------------------------
    // The entire 16×16 tile belongs to one wavefront.  Multiple wavefronts
    // in a block cover different 16×16 sub-tiles.
    //
    //   wavefront id within block: wid = tid / 64
    //   waves_n = block_tile_n / thread_tile_n = 32 / 16 = 2
    //   wid_x = wid % waves_n
    //   wid_y = wid / waves_n
    //   *thread_row = wid_y * 16
    //   *thread_col = wid_x * 16
    __device__ static void thread_tile_offset(int  tid,
                                              int  /*lane_id*/,
                                              int* thread_row,
                                              int* thread_col)
```

(continues on next page)

(continued from previous page)

```
    {
        constexpr int waves_n = 2; // block_tile_n / thread_tile_n = 32 / 16
        const int wid   = tid / 64;
        const int wid_x = wid % waves_n;
        const int wid_y = wid / waves_n;
        *thread_row = wid_y * 16;
        *thread_col = wid_x * 16;
    }

    // -- Fragment loads ------------------------------------------------------
    // load_a: each lane reads two floats from LDS into a v2float.
    //
    // v_mfma_f32_16x16x8_xf32 (1-block) interprets the 64 lanes as 4 groups
    // of 16 (indexed by g = lane_id / 16).  Each group covers 2 consecutive
    // K-positions: group g provides data for ki + 2*g and ki + 2*g + 1.
    // Within group g, lane offset (lane_id mod 16) selects one of the 16
    // A-rows.
    //
    //   A-row index  : tile_a_row + (lane_id mod 16)
    //   K-position 0 : ki + 2*(lane_id / 16)
    //   K-position 1 : ki + 2*(lane_id / 16) + 1
    __device__ static void load_a(const float* tile_a_ptr,
                                  int          tile_a_row,
                                  int          ki,
                                  int          k_tile_size,
                                  int          lane_id,
                                  elem_a     (&frag)[2])
    {
        const int row   = tile_a_row + (lane_id % 16);
        const int k_off = ki + 2 * (lane_id / 16);
        frag[0] = tile_a_ptr[row * k_tile_size + k_off];
        frag[1] = tile_a_ptr[row * k_tile_size + k_off + 1];
    }

    // load_b: each lane reads two floats from LDS (symmetric to load_a).
    //
    // tile_b_T is column-major: tile_b_T[(tile_b_col + col)][k_off].
    //   B-col index  : tile_b_col + (lane_id mod 16)
    //   K-position 0 : ki + 2*(lane_id / 16)
    //   K-position 1 : ki + 2*(lane_id / 16) + 1
    __device__ static void load_b(const float* tile_b_T_ptr,
                                  int          tile_b_col,
                                  int          ki,
                                  int          k_tile_size,
                                  int          lane_id,
                                  elem_b     (&frag)[2])
    {
        const int col   = tile_b_col + (lane_id % 16);
        const int k_off = ki + 2 * (lane_id / 16);
        frag[0] = tile_b_T_ptr[col * k_tile_size + k_off];
        frag[1] = tile_b_T_ptr[col * k_tile_size + k_off + 1];
    }
```

(continues on next page)

(continued from previous page)

```
    // -- mma ---------------------------------------------------------------
    // Issue one v_mfma_f32_16x16x8_xf32 instruction.
    // This intrinsic is single-block only; all modifier parameters must be 0.
    // cbsz=0, abid=0: single-block A-operand, no broadcast.
    // blgp=0: not supported; must be 0.
    __device__ static void mma(Accumulator&       acc,
                               const elem_a     (&a_frag)[2],
                               const elem_b     (&b_frag)[2])
    {
#if defined(__gfx942__)
        using v2float = float [[clang::ext_vector_type(2)]];
        const v2float a_vec = {a_frag[0], a_frag[1]};
        const v2float b_vec = {b_frag[0], b_frag[1]};
        acc.regs = __builtin_amdgcn_mfma_f32_16x16x8_xf32(
            a_vec, b_vec, acc.regs,
            /*cbsz=*/0, /*abid=*/0, /*blgp=*/0);
#endif
    }

    // -- store_c -----------------------------------------------------------
    // Scatter the 4 accVGPR values to their global-memory positions.
    //
    // Inverse layout (lane L, accVGPR index G in [0,3]):
    //   i = 4 * floor(L / 16) + (G mod 4)
    //   j = L mod 16
    __device__ static void store_c(const Accumulator& acc,
                                   float*              C,
                                   int                 out_row_base,
                                   int                 out_col_base,
                                   int                 m,
                                   int                 n,
                                   int                 lane_id)
    {
        const int L = lane_id;
        const int j = L % 16;

        #pragma unroll
        for(int G = 0; G < 4; ++G)
        {
            const int i = 4 * (L / 16) + (G % 4);
            const int r = out_row_base + i;
            const int c = out_col_base + j;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[G];
        }
    }
};
```

Instantiating the kernel

With `MfmaCdna3XF32Policy` in hand, plug it into the generic kernel alongside a `TilePolicy` whose `block_tile_m` and `block_tile_n` are multiples of 16 and whose `k_tile_size` is a multiple of `k_step = 8`. The policy aliases and launch configuration from the example file are:

```
// TilePolicy: 32x32 block tile, 8-element K-strip, single-buffered.
using MfmaTilePolicy = SingleBufferTilePolicyF<32, 32, 8>;
```

```
// Launch parameters:
//
// CDNA3 (gfx942): 2x2 wavefronts of 64 lanes = 256 threads per block.
//   block_tile_m = 32, thread_tile_m = 16  → 2 wavefronts along M
//   block_tile_n = 32, thread_tile_n = 16  → 2 wavefronts along N
//
// Fallback (scalar): 8x8 thread tiles in a 32x32 block → 16 threads per block,
//   padded to one full wavefront of 64.
//
// The kernel to launch and its thread count are selected at runtime based on
// the device architecture string returned by hipGetDeviceProperties.
//
// Grid: one block per 32x32 output tile.
constexpr int BLOCK_TILE_M = 32;
constexpr int BLOCK_TILE_N = 32;
```

```
    // Grid: one block per 32×32 output tile.
    const dim3 grid((N + BLOCK_TILE_N - 1) / BLOCK_TILE_N,
                    (M + BLOCK_TILE_M - 1) / BLOCK_TILE_M);

    float       ms = 0.0f;
    const char* policy_label = nullptr;

    if(is_gfx942)
    {
        auto launch = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicy, MfmaCdna3XF32Policy>
                <<<grid, dim3(256)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<MfmaTilePolicy, MfmaCdna3XF32Policy> [gfx942]";
    }
    else
    {
        auto launch = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicy, ScalarFMAPolicy<8, 8, 8>>
                <<<grid, dim3(64)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<MfmaTilePolicy, ScalarFMAPolicy> [fallback]";
    }
```

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx942 \
    matrix_multiply_cdna3_mfma.hip -o mm_cdna3_mfma
./mm_cdna3_mfma
```

> **Note**
>
> `MfmaCdna3XF32Policy` requires a CDNA3 GPU (`gfx942`). Compile with `--offload-arch=gfx942` to select the correct architecture. On other targets the `#if defined(__gfx942__)` guard selects `ScalarFMAPolicy` automatically, so the file compiles without modification.

14.1.3.7 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency. Instructions that support vector ALU (VALU) co-execution allow the compiler to overlap matrix and vector work; the VALU co-execution cycle count gives the number of VALU cycles available during the MFMA latency window. A value of 0 means VALU co-execution is not supported.

Builtin	Ops	Cycle count	VALU co-execution cycles
`__builtin_amdgcn_mfma_f32_32x32x1f32`	4096	64	0
`__builtin_amdgcn_mfma_f32_16x16x1f32`	2048	32	0
`__builtin_amdgcn_mfma_f32_4x4x1f32`	512	8	0
`__builtin_amdgcn_mfma_f32_32x32x2f32`	4096	64	0
`__builtin_amdgcn_mfma_f32_16x16x4f32`	2048	32	0
`__builtin_amdgcn_mfma_f32_32x32x4f16`	16384	64	60
`__builtin_amdgcn_mfma_f32_16x16x4f16`	8192	32	28
`__builtin_amdgcn_mfma_f32_4x4x4f16`	2048	8	4
`__builtin_amdgcn_mfma_f32_32x32x8f16`	16384	32	28
`__builtin_amdgcn_mfma_f32_16x16x16f16`	8192	16	12
`__builtin_amdgcn_mfma_f32_32x32x4bf16_1k`	16384	64	60
`__builtin_amdgcn_mfma_f32_16x16x4bf16_1k`	8192	32	28
`__builtin_amdgcn_mfma_f32_4x4x4bf16_1k`	2048	8	4
`__builtin_amdgcn_mfma_f32_32x32x8bf16_1k`	16384	32	28
`__builtin_amdgcn_mfma_f32_16x16x16bf16_1k`	8192	16	12
`__builtin_amdgcn_mfma_f32_32x32x4_xf32`	8192	32	28
`__builtin_amdgcn_mfma_f32_16x16x8_xf32`	4096	16	12
`__builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8`	32768	32	28
`__builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8`	32768	32	28
`__builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8`	32768	32	28
`__builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8`	32768	32	28
`__builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8`	16384	16	12
`__builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8`	16384	16	12
`__builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8`	16384	16	12
`__builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8`	16384	16	12
`__builtin_amdgcn_mfma_f64_16x16x4f64`	2048	32	0
`__builtin_amdgcn_mfma_f64_4x4x4f64`	512	16	0
`__builtin_amdgcn_mfma_i32_32x32x4i8`	16384	64	60
`__builtin_amdgcn_mfma_i32_16x16x4i8`	8192	32	28
`__builtin_amdgcn_mfma_i32_4x4x4i8`	2048	8	4
`__builtin_amdgcn_mfma_i32_32x32x16_i8`	32768	32	28

continues on next page

Table 1 – continued from previous page

Builtin	Ops	Cycle count	VALU co-execution cycles
`__builtin_amdgcn_mfma_i32_16x16x32_i8`	16384	16	12

14.1.3.8 Builtin reference

14.1.3.8.1 FP32-accumulate builtins

These builtins accumulate into single-precision (FP32) output fragments. They differ in the data type of the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrix inputs.

14.1.3.8.1.1 FP32 matrix inputs

The following builtins accept one FP32 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP32 output fragments.

14.1.3.8.1.2 `__builtin_amdgcn_mfma_f32_32x32x1f32`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x1f32(
    float    srcA,
    float    srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.3 `__builtin_amdgcn_mfma_f32_16x16x1f32`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x1f32(
    float    srcA,
    float    srcB,
    v16float srcC,
    int      cbsz,
```

(continues on next page)

(continued from previous page)

```
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.4 __builtin_amdgcn_mfma_f32_4x4x1f32

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x1f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: 4 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.5 __builtin_amdgcn_mfma_f32_32x32x2f32

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x2f32(
    float   srcA,
```

(continues on next page)

(continued from previous page)

```
    float    srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 2$ FP32 column elements of $\boldsymbol{A}$ and $K = 2$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.6 `__builtin_amdgcn_mfma_f32_16x16x4f32`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x4f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP32 column elements of $\boldsymbol{A}$ and $K = 4$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.7 FP16 matrix inputs

The following builtins accept four FP16 elements per lane packed into a `v4half` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into FP32 output fragments.

14.1.3.8.1.8 `__builtin_amdgcn_mfma_f32_32x32x4f16`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x4f16(
    v4half   srcA,
    v4half   srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.9 `__builtin_amdgcn_mfma_f32_16x16x4f16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x4f16(
    v4half   srcA,
    v4half   srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x4f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across four accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: Four FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x8f16(
    v4half   srcA,
    v4half   srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 8$ FP16 column elements of $\boldsymbol{A}$ and $K = 8$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be 0.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be 0.
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.12 __builtin_amdgcn_mfma_f32_16x16x16f16

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x16f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP16 column elements of $\boldsymbol{A}$ and $K = 16$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across four accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: Four FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be 0.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be 0.
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.13 BF16 matrix inputs

CDNA3 supports only the _1k BF16 variants. These builtins accept four BF16 elements per lane packed into a v4bfloat register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into FP32 output fragments.

14.1.3.8.1.14 __builtin_amdgcn_mfma_f32_32x32x4bf16_1k

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v32float srcC,
    int      cbsz,
```

(continues on next page)

(continued from previous page)

```
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
srcA	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
srcB	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
srcC	v32float	Accumulator input: 32 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.15 `__builtin_amdgcn_mfma_f32_16x16x4bf16_1k`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
srcB	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.16 `__builtin_amdgcn_mfma_f32_4x4x4bf16_1k`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x4bf16_1k(
    v4bfloat srcA,
```

(continues on next page)

(continued from previous page)

```
    v4bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.17 `__builtin_amdgcn_mfma_f32_32x32x8bf16_1k`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x8bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 8$ BF16 column elements of $\boldsymbol{A}$ and $K = 8$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.18 `__builtin_amdgcn_mfma_f32_16x16x16bf16_1k`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x16bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 16$ BF16 column elements of $\boldsymbol{A}$ and $K = 16$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.19 XF32 matrix inputs

eXtended Float32 (XF32) accepts standard FP32 (IEEE binary32) inputs but rounds each mantissa to 10 bits before multiplication, providing higher throughput than full FP32 matrix multiplication. Each lane passes two values packed into a `v2float` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs.

14.1.3.8.1.20 `__builtin_amdgcn_mfma_f32_32x32x4_xf32`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x4_xf32(
    v2float  srcA,
    v2float  srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ XF32 column elements of $\boldsymbol{A}$ and $K = 4$ XF32 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

eXtended Float32 (XF32) accepts standard FP32 (IEEE binary32) inputs but rounds each mantissa to 10 bits before multiplication, providing higher throughput than full FP32 matrix multiplication. Each lane provides two values packed into a `v2float` register.

Parameter	Type	Description
`srcA`	v2float	Two XF32 elements of $\boldsymbol{A}$ per lane.
`srcB`	v2float	Two XF32 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.21 `__builtin_amdgcn_mfma_f32_16x16x8_xf32`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x8_xf32(
    v2float srcA,
    v2float srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 8$ XF32 column elements of $\boldsymbol{A}$ and $K = 8$ XF32 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

eXtended Float32 (XF32) accepts standard FP32 (IEEE binary32) inputs but rounds each mantissa to 10 bits before multiplication, providing higher throughput than full FP32 matrix multiplication. Each lane provides two values packed into a `v2float` register.

Parameter	Type	Description
`srcA`	v2float	Two XF32 elements of $\boldsymbol{A}$ per lane.
`srcB`	v2float	Two XF32 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.22 FP8 and BF8 matrix inputs

The following builtins accept eight 8-bit floating-point values per lane packed into a `long long` register. Two 8-bit formats are supported:

- **FP8** (`fp8`): E4M3 format (1 sign, 4 exponent, 3 mantissa bits).
- **BF8** (`bf8`): E5M2 format (1 sign, 5 exponent, 2 mantissa bits).

The $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices can use different formats, giving four combinations per tile shape.

14.1.3.8.1.23 `__builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP8 column elements of $\boldsymbol{A}$ and $K = 16$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the FP8 (E4M3) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.24 `__builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP8 column elements of $\boldsymbol{A}$ and $K = 16$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the FP8 (E4M3) format and $\boldsymbol{B}$ uses the BF8 (E5M2) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.25 `__builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ BF8 column elements of $\boldsymbol{A}$ and $K = 16$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the BF8 (E5M2) format and $\boldsymbol{B}$ uses the FP8 (E4M3) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.26 `__builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ BF8 column elements of $\boldsymbol{A}$ and $K = 16$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the BF8 (E5M2) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.27 `__builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8(
    long long srcA,
    long long srcB,
    v4float   srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ FP8 column elements of $\boldsymbol{A}$ and $K = 32$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the FP8 (E4M3) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.28 `__builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8(
    long long srcA,
    long long srcB,
    v4float   srcC,
```

(continues on next page)

(continued from previous page)

```
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ FP8 column elements of $\boldsymbol{A}$ and $K = 32$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the FP8 (E4M3) format and $\boldsymbol{B}$ uses the BF8 (E5M2) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.29 `__builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8(
    long long srcA,
    long long srcB,
    v4float   srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ BF8 column elements of $\boldsymbol{A}$ and $K = 32$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the BF8 (E5M2) format and $\boldsymbol{B}$ uses the FP8 (E4M3) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.1.30 `__builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8(
    long long srcA,
    long long srcB,
    v4float   srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ BF8 column elements of $\boldsymbol{A}$ and $K = 32$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the BF8 (E5M2) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.2 FP64-accumulate builtins

CDNA3 retains native double-precision (FP64) matrix accumulation from CDNA2. These builtins accept one FP64 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP64 output fragments held in `v4double` registers.

> **Note**
>
> On CDNA3, the `blgp` parameter of FP64 builtins is repurposed as a 3-bit source negation modifier rather than a lane-group pattern selector. See the *Common parameters* section for details.

14.1.3.8.2.1 FP64 matrix inputs

The following builtins accept one FP64 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP64 output fragments.

14.1.3.8.2.2 `__builtin_amdgcn_mfma_f64_16x16x4f64`

Signature and parameters for this builtin.

```
v4double __builtin_amdgcn_mfma_f64_16x16x4f64(
    double   srcA,
    double   srcB,
    v4double srcC,
    int      cbsz,
```

(continues on next page)

(continued from previous page)

```
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP64 outer-product accumulation. Each instruction processes $K = 4$ FP64 column elements of $\boldsymbol{A}$ and $K = 4$ FP64 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP64 tile held across 4 accVGPRs per lane (8 physical accVGPRs, since each FP64 value occupies two 32-bit registers). This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	double	One FP64 element of $\boldsymbol{A}$ per lane.
`srcB`	double	One FP64 element of $\boldsymbol{B}$ per lane.
`srcC`	v4doub	Accumulator input: 4 FP64 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, repurposed as a negation modifier (3-bit field). Bit 0 negates `srcA`, bit 1 negates `srcB`, bit 2 negates `srcC`.

Returns `v4double` – updated accumulator (srcA $\times$ srcB + srcC).

Note

Unlike most FP32, BF16, and INT8 MFMA instructions, FP64 MFMA instructions cannot co-execute with VALU instructions. To hide execution latency, schedule non-VALU instructions (loads, stores, or branches) around FP64 MFMA operations.

14.1.3.8.2.3 `__builtin_amdgcn_mfma_f64_4x4x4f64`

Signature and parameters for this builtin.

```
double __builtin_amdgcn_mfma_f64_4x4x4f64(
    double srcA,
    double srcB,
    double srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 4×4 FP64 outer-product accumulation. Each instruction processes $K = 4$ FP64 column elements of $\boldsymbol{A}$ and $K = 4$ FP64 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP64 tile. Each lane holds one FP64 output element across four input blocks.

Parameter	Type	Description
`srcA`	double	One FP64 element of $\boldsymbol{A}$ per lane.
`srcB`	double	One FP64 element of $\boldsymbol{B}$ per lane.
`srcC`	double	Accumulator input: 1 FP64 element per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, repurposed as a negation modifier (3-bit field). Bit 0 negates `srcA`, bit 1 negates `srcB`, bit 2 negates `srcC`.

Returns `double` – updated accumulator (srcA $\times$ srcB $+$ srcC).

Note

Unlike most FP32, BF16, and INT8 MFMA instructions, FP64 MFMA instructions cannot co-execute with VALU instructions. To hide execution latency, schedule non-VALU instructions (loads, stores, or branches) around FP64 MFMA operations.

14.1.3.8.3 INT32-accumulate builtins

These builtins accumulate into signed 32-bit integer (INT32) output fragments.

14.1.3.8.3.1 INT8 matrix inputs

The following builtins accept four signed 8-bit integer elements per lane packed into a single `int` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs.

14.1.3.8.3.2 `__builtin_amdgcn_mfma_i32_32x32x4i8`

Signature and parameters for this builtin.

```
v32int __builtin_amdgcn_mfma_i32_32x32x4i8(
    int    srcA,
    int    srcB,
    v32int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 32×32 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v32int	Accumulator input: 32 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.3.3 `__builtin_amdgcn_mfma_i32_16x16x4i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_16x16x4i8(
    int    srcA,
    int    srcB,
    v16int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.3.4 `__builtin_amdgcn_mfma_i32_4x4x4i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_4x4x4i8(
    int   srcA,
    int   srcB,
    v4int srcC,
    int   cbsz,
    int   abid,
    int   blgp);
```

Computes one step of a 4×4 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 4×4 INT32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

The following wider-K variants accept eight signed 8-bit integer elements per lane packed into a `long long` register.

14.1.3.8.3.5 `__builtin_amdgcn_mfma_i32_32x32x16_i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_32x32x16_i8(
    long long srcA,
    long long srcB,
    v16int    srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 INT32 outer-product accumulation. Each instruction processes $K = 16$ INT8 column elements of $\boldsymbol{A}$ and $K = 16$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Each lane provides eight signed 8-bit integer values packed into a `long long` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$.

Parameter	Type	Description
`srcA`	long long	Eight INT8 elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight INT8 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.3.8.3.6 `__builtin_amdgcn_mfma_i32_16x16x32_i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_16x16x32_i8(
    long long srcA,
    long long srcB,
    v4int     srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 32$ INT8 column elements of $\boldsymbol{A}$ and $K = 32$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Each lane provides eight signed 8-bit integer values packed into a `long long` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$.

Parameter	Type	Description
`srcA`	long long	Eight INT8 elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight INT8 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4 CDNA4 dense MFMA builtins

Matrix Fused Multiply-Add (MFMA) builtins let you issue hardware matrix multiply-accumulate operations directly from HIP device code on CDNA4 GPUs (`gfx950`, MI350 series). Each MFMA instruction multiplies a small $\boldsymbol{A}$ fragment by a small $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single wavefront of 64 lanes. The hardware delivers significantly higher throughput than an equivalent sequence of scalar fused multiply-add (FMA) instructions.

CDNA4 retains FP32, FP16, BF16, INT8, FP64, FP8, and BF8 shapes from CDNA3 and adds wider-K single-block variants for FP16, BF16, and INT8, plus a new class of scaled sub-byte builtins (`mfma_scale_*`) that support FP8, BF8, FP6 (E2M3 and E3M2), and FP4 (E2M1) input formats with per-block scaling. The eXtended Float32 (XF32) input format available on CDNA3 is not supported on CDNA4. See the builtin reference below for the complete list of supported shapes and formats.

14.1.4.1 Architecture availability

The builtins on this page target CDNA4 (`gfx950`, MI350 series) exclusively. Equivalent builtins for other CDNA generations are documented on their own reference pages:

- *CDNA dense MFMA builtins* – CDNA (`gfx908`, MI100 series)
- *CDNA2 dense MFMA builtins* – CDNA2 (`gfx90a`, MI200 series)
- *CDNA3 dense MFMA builtins* – CDNA3 (`gfx942`, MI300 series)

14.1.4.2 Naming convention

All MFMA builtins follow the pattern:

```
__builtin_amdgcn_mfma_<out_type>_<M>x<N>x<K><in_type>
```

For FP8 and BF8 builtins, where the $\boldsymbol{A}$ and $\boldsymbol{B}$ input types can differ, the pattern is:

```
__builtin_amdgcn_mfma_<out_type>_<M>x<N>x<K>_<typeA>_<typeB>
```

`out_type`
: Accumulator element type (`f32`, `f64`, or `i32`).

`M, N, K`
: Tile dimensions in elements. The instruction computes the contribution of a K-wide panel of $\boldsymbol{A}$ ($M \times K$) and a K-wide panel of $\boldsymbol{B}$ ($K \times N$) to an $M \times N$ output tile. Each instruction processes one K step; the caller loops over K to accumulate a full matrix product.

in_type
: Input element type (`f32`, `f64`, `f16`, `bf16`, `i8`, or `xf32`).

typeA, typeB
: For FP8 and BF8 builtins: the element type of the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices respectively. Each is one of `fp8` (E4M3 format) or `bf8` (E5M2 format).

> **Note**
>
> CDNA4 renames several underlying instruction set architecture (ISA) instructions to include an explicit block count (for example, `v_mfma_f32_32x32x1f32` becomes `v_mfma_f32_32x32x1_2b_f32` in the CDNA4 ISA). The HIP builtin names (`__builtin_amdgcn_mfma_*`) are unchanged; the rename is transparent to HIP device code.

14.1.4.3 Accumulator layout

Each MFMA instruction computes one or more independent $M \times N$ output tiles simultaneously across the 64 lanes of a wavefront. The number of independent tiles is called the *block count*. It depends on how the $\boldsymbol{A}$ matrix rows are distributed across lane groups:

- **Scalar-input variants** (FP32 or FP64 $\boldsymbol{A}$ and $\boldsymbol{B}$): each K position occupies a separate group of M lanes, so blocks $=$ wavefront_size$/(M \times K)$.
- **Packed-input variants** (FP16, BF16, XF32, INT8, FP8, BF8 $\boldsymbol{A}$ and $\boldsymbol{B}$): all K positions are packed into the register bits of the *same* lane group, so blocks $=$ wavefront_size$/M$ regardless of K.

Each block is independent: the $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$/$\boldsymbol{D}$ operands of different blocks occupy distinct lane and accVGPR positions and compute separate outer products.

The total number of accVGPRs per lane scales with the block count:

$$\text{accVGPRs per lane} = \text{blocks} \times \frac{M \times N}{\text{wavefront_size}}$$

> **Note**
>
> On CDNA4, all four operands ($\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$) can reside in either accumulation VGPRs (accVGPRs) or standard architecture VGPRs (archVGPRs). In HIP device code the compiler selects the appropriate register class automatically.
>
> Each of the 64 wavefront lanes maintains its own private accVGPR file. An accVGPR index always refers to a register within one specific lane's file; the same index in two different lanes denotes two distinct physical registers. The layout tables and formulas in the following sections use *accVGPR index* as a logical position label; the actual register class (accVGPR or ArchVGPR) is chosen by the compiler and does not affect the layout.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \le i < M$
- j – zero-based column index within the tile, $0 \le j < N$
- b – block index, $0 \le b <$ blocks
- **lane** – wavefront lane that holds the element, $0 \le$ lane < 64
- **accVGPR** – zero-based index into that lane's *private* accumulator register file; the same index in two different lanes refers to two distinct physical registers

14.1.4.3.1 32×32 layout

The 32×32 tile shape uses two blocks. All 32 accVGPRs per lane are active: accVGPRs 0–15 belong to block 0, accVGPRs 16–31 to block 1.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0 lanes 0–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=4 lanes 32–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=8 lanes 0–31	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=12 lanes 32–63	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=16 lanes 0–31	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=20 lanes 32–63	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
i=24 lanes 0–31	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
i=28 lanes 32–63	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15

Fig. 14: 32×32 **accumulator layout – 2 blocks.** Each cell shows the accVGPR index that holds output element (i, j) of block 0; block 1 adds 16. Teal cells (rows where $\lfloor i/4 \rfloor$ is even) belong to lanes 0–31; grey cells to lanes 32–63. Column j gives the lane offset within the group.

Given output element (i, j) in block b:

$$\text{lane} = \left(32 \cdot \lfloor \frac{i}{4} \rfloor\right) \bmod 64 + j$$

$$\text{accVGPR} = 16b + 4\lfloor \frac{i}{8} \rfloor + (i \bmod 4)$$

Conversely, given a lane L and accVGPR index G:

$$i = \left(8 \cdot \lfloor \frac{G}{4} \rfloor\right) \bmod 32 + 4\lfloor \frac{L}{32} \rfloor + (G \bmod 4)$$
$$j = L \bmod 32$$
$$b = \lfloor \frac{G}{16} \rfloor$$

The row-to-lane mapping groups rows in bands of four. Within block 0:

Rows	Lanes	accVGPRs
0—3	0—31	0—3
4—7	32—63	0—3
8—11	0—31	4—7
12—15	32—63	4—7
16—19	0—31	8—11
20—23	32—63	8—11
24—27	0—31	12—15
28—31	32—63	12—15

Block 1 uses the same lane pattern with accVGPRs 16–31.

Builtins with a **1-block** 32×32 result (`v16float` / `v16int`, 16 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

14.1.4.3.2 16×16 layout

The 16×16 tile shape uses four blocks. The 16 accVGPRs per lane are partitioned by block: accVGPRs 0—3 for block 0, 4—7 for block 1, 8—11 for block 2, and 12—15 for block 3.

Given output element (i, j) in block b:

$$\text{lane} = 16\lfloor \frac{i}{4} \rfloor + j$$
$$\text{accVGPR} = 4b + (i \bmod 4)$$

Conversely, given lane L and accVGPR index G:

$$i = 4\lfloor \frac{L}{16} \rfloor + (G \bmod 4)$$
$$j = L \bmod 16$$
$$b = \lfloor \frac{G}{4} \rfloor$$

The row-to-lane mapping (identical for every block):

Rows	Lanes
0—3	0—15
4—7	16—31
8—11	32—47
12—15	48—63

Builtins with a **1-block** 16×16 result (`v4float` / `v4int`, 4 accVGPRs per lane) use only block 0 of the layout above. These variants do not support the `cbsz` and `abid` modifiers.

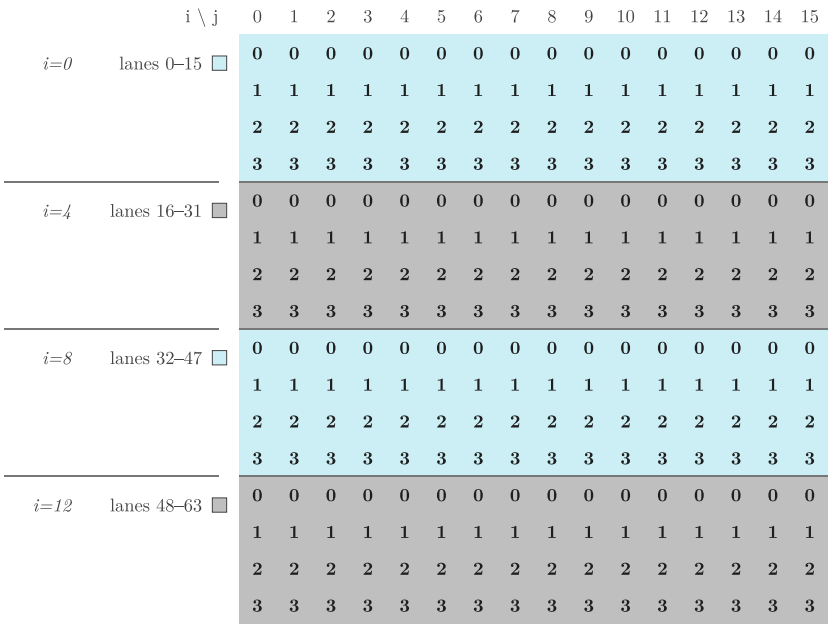


Fig. 15: 16×16 **accumulator layout – 4 blocks.** Each cell shows the accVGPR index for block 0; block k adds $4k$. Rows 0—3 (teal, lanes 0—15), rows 4—7 (grey, lanes 16—31), rows 8—11 (teal, lanes 32—47), rows 12—15 (grey, lanes 48—63). Column j gives the lane offset within the group.

14.1.4.3.3 4×4 layout

The 4×4 tile shape uses 16 blocks. A single wavefront simultaneously computes 16 independent 4×4 outer products. Each group of 4 consecutive lanes (lanes $4b$ through $4b + 3$) holds all 16 output elements of block b across 4 accVGPRs.

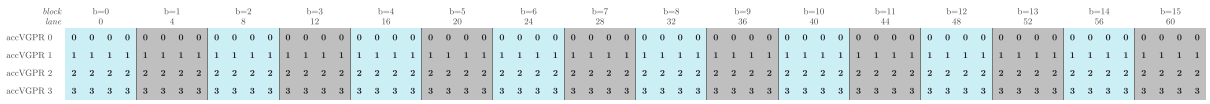


Fig. 16: 4×4 **accumulator layout – 16 blocks.** Columns are lanes 0–63; each group of 4 consecutive lanes holds one complete 4×4 output tile in accVGPRs 0–3 (rows). Teal groups are even-numbered blocks, grey groups odd-numbered blocks.

Given output element (i, j) in block b:

$$\text{lane} = 4b + j$$
$$\text{accVGPR} = i$$

Conversely, given lane L and accVGPR index G:

$$i = G \bmod 4$$
$$j = L \bmod 4$$
$$b = \lfloor \frac{L}{4} \rfloor$$

14.1.4.3.4 16×16 FP64 layout

The 16×16 FP64 tile shape uses one block. Each lane holds four FP64 output elements across 4 accVGPR pairs (8 physical accVGPRs, since each FP64 value occupies two 32-bit registers).

> **Note**
>
> FP64 accVGPR indices count *pairs* of physical registers. accVGPR pair k corresponds to physical registers `v[2k+1:2k]`.

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0 lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=1 lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=2 lanes 32–47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=3 lanes 48–63	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=4 lanes 0–15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=5 lanes 16–31	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=6 lanes 32–47	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=7 lanes 48–63	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=8 lanes 0–15	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=9 lanes 16–31	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=10 lanes 32–47	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=11 lanes 48–63	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=12 lanes 0–15	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=13 lanes 16–31	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=14 lanes 32–47	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=15 lanes 48–63	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

Fig. 17: 16×16 **FP64 accumulator layout – 1 block.** Each cell shows the accVGPR pair index k that holds output element (i, j); physical registers are `v[2k+1:2k]`. Each group of 16 consecutive lanes (column offset j) covers all four rows within one row-modulo-4 band.

Given output element (i, j):

$$\text{lane} = 16 \cdot (i \bmod 4) + j$$

$$\text{accVGPR pair } k = \lfloor \frac{i}{4} \rfloor$$

Conversely, given lane L and accVGPR pair index k:

$$i = 4k + \lfloor \frac{L}{16} \rfloor$$

$$j = L \bmod 16$$

The row-to-lane mapping. Each group of 16 consecutive lanes covers one row modulo 4; the accVGPR pair selects the row group:

Rows	Lanes	accVGPR pair (physical regs)
0, 4, 8, 12	0—15	0, 1, 2, 3 (`v[1:0]`, `v[3:2]`, `v[5:4]`, `v[7:6]`)
1, 5, 9, 13	16—31	0, 1, 2, 3
2, 6, 10, 14	32—47	0, 1, 2, 3
3, 7, 11, 15	48—63	0, 1, 2, 3

14.1.4.3.5 4×4 FP64 layout

The 4×4 FP64 tile shape uses four blocks. Each lane holds one FP64 output element in a single accVGPR pair (2 physical accVGPRs, `v[1:0]`).

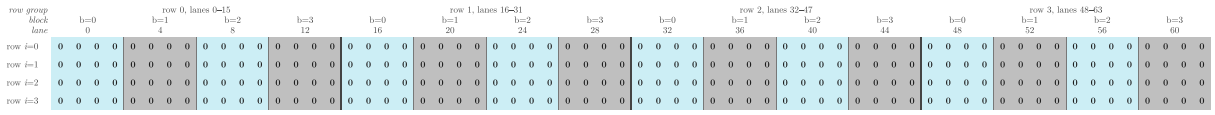


Fig. 18: 4×4 **FP64 accumulator layout – 4 blocks.** Columns are lanes 0–63; rows are matrix rows 0–3. Within each 16-lane row group, four sub-groups of 4 lanes hold blocks 0–3. Every cell holds one FP64 value in accVGPR pair 0 (`v[1:0]`). Teal sub-groups are even-numbered blocks, grey sub-groups are odd-numbered blocks.

Given output element (i, j) in block b:

$$\text{lane} = 16i + 4b + j$$

Conversely, given lane L:

$$i = \lfloor \frac{L}{16} \rfloor$$
$$j = L \bmod 4$$
$$b = \lfloor \frac{L \bmod 16}{4} \rfloor$$

The row-to-lane mapping. Lanes 0—15 hold all four blocks of row 0; the pattern repeats for rows 1—3 at lane offsets of 16, 32, and 48:

Row	Block	Lanes	accVGPR pair
0	0	0—3	0 (`v[1:0]`)
0	1	4—7	0 (`v[1:0]`)
0	2	8—11	0 (`v[1:0]`)
0	3	12—15	0 (`v[1:0]`)
1	0—3	16—31	0 (`v[1:0]`)
2	0—3	32—47	0 (`v[1:0]`)
3	0—3	48—63	0 (`v[1:0]`)

14.1.4.4 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v4float   = float [[clang::ext_vector_type(4)]];
using v16float  = float [[clang::ext_vector_type(16)]];
```

(continues on next page)

(continued from previous page)

```
using v32float  = float [[clang::ext_vector_type(32)]];
using v4half    = _Float16 [[clang::ext_vector_type(4)]];
using v8half    = _Float16 [[clang::ext_vector_type(8)]];
using v4int     = int [[clang::ext_vector_type(4)]];
using v6int     = int [[clang::ext_vector_type(6)]];        // FP6 scale inputs
using v8int     = int [[clang::ext_vector_type(8)]];        // FP8/BF8 scale inputs
using v16int    = int [[clang::ext_vector_type(16)]];
using v32int    = int [[clang::ext_vector_type(32)]];
using v2bfloat  = short [[clang::ext_vector_type(2)]];      // bf16 storage
using v4bfloat  = short [[clang::ext_vector_type(4)]];      // bf16 storage
using v8bfloat  = short [[clang::ext_vector_type(8)]];      // wider-K BF16
using v4double  = double [[clang::ext_vector_type(4)]];
```

FP8 and BF8 operands passed to the standard (non-scale) MFMA builtins use a 64-bit integer (`long long`) that packs eight 8-bit values per lane. The scaled sub-byte builtins (`mfma_scale_*`) use `v4int`, `v6int`, or `v8int` depending on the selected format; see the per-builtin reference for details.

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane; the total VGPR count equals the element count multiplied by the element size in 32-bit words.

14.1.4.5 Common parameters

See *Common MFMA parameters* for a complete description of the `cbsz`, `abid`, and `blgp` modifiers shared by all MFMA builtins.

14.1.4.6 Using MFMA builtins as a compute policy

The matrix multiplication tutorial in *Optimizing GEMM in HIP* uses a `ComputePolicy` type parameter to separate the multiply-accumulate logic from the rest of the kernel. You can drop an MFMA-based policy into that framework without changing the outer kernel.

The example below implements `MfmaCdna4F16Policy` using `v_mfma_f32_16x16x32_f16` – a 16×16 FP16 builtin available on CDNA4 (`gfx950`, MI350 series). Each wavefront computes a single 16×16 output tile; one block is active, giving 4 FP32 accVGPRs per lane.

The complete source file is available for download:

- `matrix_multiply_cdna4_mfma.hip`

Policy constants

`v_mfma_f32_16x16x32_f16` consumes 32 K-positions per call (K=32), so `k_step = 32`. Each lane provides eight FP16 values (`v8half`) for $\boldsymbol{A}$ and eight for $\boldsymbol{B}$; the 64 lanes partition into four groups of 16, each group covering eight K-positions. The entire 16×16 tile belongs to one wavefront, so `thread_tile_m = thread_tile_n = 16` and `effective_lanes = 64`.

Accumulator layout

The builtin returns a `v4float` holding 4 FP32 values – one for each accVGPR. The `Accumulator` struct wraps this directly. The layout formulas from the *Accumulator layout* section translate `(lane, accVGPR)` back to (i, j) coordinates during the `store_c()` pass.

Fragment loading

Because each lane provides eight FP16 values for K=32, each `load_a` call reads eight consecutive scalars. Lane group $g = \lfloor \text{lane_id}/16 \rfloor$ covers K-positions `ki + 8g` through `ki + 8g + 7`; within the group, lane offset `lane_id mod 16` selects one of the 16 $\boldsymbol{A}$-rows (or $\boldsymbol{B}$-columns). The `mma()` method packs the eight values into a `v8half` before issuing the instruction.

> **Note**
>
> The all-modifier-zero constraint means `cbsz`, `abid`, and `blgp` must all be passed as `0`. `v_mfma_f32_16x16x32_f16` is single-block only and does not support the `cbsz`/`abid` broadcast mechanism or the `blgp` lane-group pattern modifier.

```
struct MfmaCdna4F16Policy
{
    // -- ComputePolicy constants ------------------------------------------
    // One wavefront owns a 16×16 output tile.
    // All 64 lanes hold unique output elements (4 elements per lane).
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int frag_size_m     = 8;
    static constexpr int frag_size_n     = 8;
    static constexpr int effective_lanes = 64;
    // v_mfma_f32_16x16x32_f16 consumes 32 K-positions per call (k_step=32).
    // Each lane provides eight FP16 values (v8half) for A and eight for B.
    // The 64 lanes split into 4 groups of 16; each group covers 8 K-positions.
    static constexpr int k_step          = 32;

    using elem_a = _Float16;
    using elem_b = _Float16;

    // -- Accumulator ------------------------------------------------------
    // v4float holds 4 FP32 accVGPRs per lane (accVGPRs 0--3).
    using v4float = float [[clang::ext_vector_type(4)]];

    struct Accumulator
    {
        v4float regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
        acc.regs = v4float{};
    }

    // -- thread_tile_offset -----------------------------------------------
    // The entire 16×16 tile belongs to one wavefront.  Multiple wavefronts
    // in a block cover different 16×16 sub-tiles.
    //
    //   wavefront id within block: wid = tid / 64
    //   waves_n = block_tile_n / thread_tile_n = 32 / 16 = 2
    //   wid_x = wid % waves_n
```

(continues on next page)

(continued from previous page)

```
//   wid_y = wid / waves_n
//   *thread_row = wid_y * 16
//   *thread_col = wid_x * 16
__device__ static void thread_tile_offset(int  tid,
                                          int  /*lane_id*/,
                                          int* thread_row,
                                          int* thread_col)
{
    constexpr int waves_n = 2; // block_tile_n / thread_tile_n = 32 / 16
    const int wid   = tid / 64;
    const int wid_x = wid % waves_n;
    const int wid_y = wid / waves_n;
    *thread_row = wid_y * 16;
    *thread_col = wid_x * 16;
}

// -- Fragment loads ------------------------------------------------------
// load_a: each lane reads eight FP16 values from LDS into a v8half.
//
// v_mfma_f32_16x16x32_f16 (1-block) interprets the 64 lanes as 4 groups
// of 16 (indexed by g = lane_id / 16).  Each group covers 8 consecutive
// K-positions: group g provides data for ki + 8*g through ki + 8*g + 7.
// Within group g, lane offset (lane_id mod 16) selects one of the 16
// A-rows.
//
//   A-row index : tile_a_row + (lane_id mod 16)
//   K-position e: ki + 8*(lane_id / 16) + e   (e = 0..7)
__device__ static void load_a(const _Float16* tile_a_ptr,
                              int             tile_a_row,
                              int             ki,
                              int             k_tile_size,
                              int             lane_id,
                              elem_a        (&frag)[8])
{
    const int row   = tile_a_row + (lane_id % 16);
    const int k_off = ki + 8 * (lane_id / 16);
    #pragma unroll
    for(int e = 0; e < 8; ++e)
        frag[e] = tile_a_ptr[row * k_tile_size + k_off + e];
}

// load_b: each lane reads eight FP16 values from LDS (symmetric to load_a).
//
// tile_b_T is stored transposed: tile_b_T[(tile_b_col + col)][k_off].
//   B-col index : tile_b_col + (lane_id mod 16)
//   K-position e: ki + 8*(lane_id / 16) + e   (e = 0..7)
__device__ static void load_b(const _Float16* tile_b_T_ptr,
                              int             tile_b_col,
                              int             ki,
                              int             k_tile_size,
                              int             lane_id,
                              elem_b        (&frag)[8])
```

(continues on next page)

(continued from previous page)

```
    {
        const int col   = tile_b_col + (lane_id % 16);
        const int k_off = ki + 8 * (lane_id / 16);
        #pragma unroll
        for(int e = 0; e < 8; ++e)
            frag[e] = tile_b_T_ptr[col * k_tile_size + k_off + e];
    }

    // -- mma ---------------------------------------------------------------
    // Issue one v_mfma_f32_16x16x32_f16 instruction.
    // This intrinsic is single-block only; all modifier parameters must be 0.
    // cbsz=0, abid=0: single-block A-operand, no broadcast.
    // blgp=0: not supported; must be 0.
    __device__ static void mma(Accumulator&       acc,
                               const elem_a     (&a_frag)[8],
                               const elem_b     (&b_frag)[8])
    {
#if defined(__gfx950__)
        using v8half = _Float16 [[clang::ext_vector_type(8)]];
        const v8half a_vec = {a_frag[0], a_frag[1], a_frag[2], a_frag[3],
                              a_frag[4], a_frag[5], a_frag[6], a_frag[7]};
        const v8half b_vec = {b_frag[0], b_frag[1], b_frag[2], b_frag[3],
                              b_frag[4], b_frag[5], b_frag[6], b_frag[7]};
        acc.regs = __builtin_amdgcn_mfma_f32_16x16x32_f16(
            a_vec, b_vec, acc.regs,
            /*cbsz=*/0, /*abid=*/0, /*blgp=*/0);
#endif
    }

    // -- store_c -----------------------------------------------------------
    // Scatter the 4 accVGPR values to their global-memory positions.
    //
    // Inverse layout (lane L, accVGPR index G in [0,3]):
    //   i = 4 * floor(L / 16) + (G mod 4)
    //   j = L mod 16
    __device__ static void store_c(const Accumulator& acc,
                                   float*             C,
                                   int                out_row_base,
                                   int                out_col_base,
                                   int                m,
                                   int                n,
                                   int                lane_id)
    {
        const int L = lane_id;
        const int j = L % 16;

        #pragma unroll
        for(int G = 0; G < 4; ++G)
        {
            const int i = 4 * (L / 16) + (G % 4);
            const int r = out_row_base + i;
            const int c = out_col_base + j;
```

(continues on next page)

(continued from previous page)

```
            if(r < m && c < n)
                C[r * n + c] = acc.regs[G];
        }
    }
};
```

Instantiating the kernel

With `MfmaCdna4F16Policy` in hand, plug it into the generic kernel alongside a `TilePolicy` whose `block_tile_m` and `block_tile_n` are multiples of 16 and whose `k_tile_size` is a multiple of `k_step = 32`. The policy aliases and launch configuration from the example file are:

```
// TilePolicy for the standard FP16 path (transposed B in LDS).
using MfmaTilePolicy    = SingleBufferTilePolicyF16<32, 32, 32>;
// TilePolicy for the ds_read_tr path (row-major B in LDS).
using MfmaTilePolicyTrB = SingleBufferTilePolicyF16RowB<32, 32, 32>;
```

```
// Launch parameters:
//
// CDNA4 (gfx950): 2×2 wavefronts of 64 lanes = 256 threads per block.
//   block_tile_m = 32, thread_tile_m = 16  → 2 wavefronts along M
//   block_tile_n = 32, thread_tile_n = 16  → 2 wavefronts along N
//
// Fallback (scalar): 8×8 thread tiles in a 32×32 block → 16 threads per
//   block, padded to one full wavefront of 64.
//
// The kernel to launch and its thread count are selected at runtime based
// on the device architecture string returned by hipGetDeviceProperties.
//
// Grid: one block per 32×32 output tile.
constexpr int BLOCK_TILE_M = 32;
constexpr int BLOCK_TILE_N = 32;
```

```
    // Grid: one block per 32×32 output tile.
    const dim3 grid((N + BLOCK_TILE_N - 1) / BLOCK_TILE_N,
                    (M + BLOCK_TILE_M - 1) / BLOCK_TILE_M);

    float       ms = 0.0f;
    const char* policy_label = nullptr;

    if(is_gfx950)
    {
        // Standard path: scalar B load from transposed LDS tile.
        auto launch = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicy,
                                    MfmaCdna4F16Policy,
                                    _Float16>
                <<<grid, dim3(256)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
```

(continues on next page)

(continued from previous page)

```
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<MfmaTilePolicy, MfmaCdna4F16Policy> [gfx950]";

        // ds_read_tr path: hardware-transposed B load from row-major LDS tile.
        // Re-use the same device buffers; d_B was already uploaded.
        HIP_CHECK(hipMemset(d_C, 0, sizeof(float) * M * N));
        auto launch_tr = [&]()
        {
            matrix_multiply_generic<MfmaTilePolicyTrB,
                                    MfmaCdna4F16TrPolicy,
                                    _Float16>
                <<<grid, dim3(256)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        const float ms_tr = time_kernel_ms(launch_tr, WARMUP_RUNS, TIMING_RUNS);
        HIP_CHECK(hipMemcpy(h_C.data(), d_C, sizeof(float) * M * N,
                            hipMemcpyDeviceToHost));
        const bool ok_tr = verify_result(h_C, h_A, M, N, K);
        print_metrics(
            "GemmKernel<MfmaTilePolicyTrB, MfmaCdna4F16TrPolicy> [gfx950]",
            ok_tr, ms_tr, sizeof(_Float16), M, N, K);
    }
    else
    {
        // Scalar FP32 fallback: upload B as FP32.
        std::vector<float> h_B_f32(K * N, 0.0f);
        for(int i = 0; i < std::min(K, N); ++i)
            h_B_f32[i * N + i] = 1.0f;
        float* d_B_f32;
        HIP_CHECK(hipMalloc(&d_B_f32, sizeof(float) * K * N));
        HIP_CHECK(hipMemcpy(d_B_f32, h_B_f32.data(), sizeof(float) * K * N,
                            hipMemcpyHostToDevice));
        float* d_A_f32;
        HIP_CHECK(hipMalloc(&d_A_f32, sizeof(float) * M * K));
        std::vector<float> h_A_f32(M * K);
        for(int i = 0; i < M * K; ++i)
            h_A_f32[i] = static_cast<float>(h_A[i]);
        HIP_CHECK(hipMemcpy(d_A_f32, h_A_f32.data(), sizeof(float) * M * K,
                            hipMemcpyHostToDevice));

        auto launch = [&]()
        {
            matrix_multiply_generic<SingleBufferTilePolicyF<32, 32, 8>,
                                    ScalarFMAPolicy<8, 8, 8>,
                                    float>
                <<<grid, dim3(64)>>>(d_A_f32, d_B_f32, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<SingleBufferTilePolicyF, ScalarFMAPolicy>␣
↪[fallback]";
        HIP_CHECK(hipFree(d_A_f32));
```

(continues on next page)

(continued from previous page)

```
        HIP_CHECK(hipFree(d_B_f32));
    }
```

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx950 \
    matrix_multiply_cdna4_mfma.hip -o mm_cdna4_mfma
./mm_cdna4_mfma
```

Note

`MfmaCdna4F16Policy` requires a CDNA4 GPU (`gfx950`). Compile with `--offload-arch=gfx950` to select the correct architecture. On other targets the `#if defined(__gfx950__)` guard selects `ScalarFMAPolicy` automatically, so the file compiles without modification.

14.1.4.7 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency.

Builtin	Ops	Cycle count
`__builtin_amdgcn_mfma_f32_32x32x1f32`	4096	64
`__builtin_amdgcn_mfma_f32_16x16x1f32`	2048	32
`__builtin_amdgcn_mfma_f32_4x4x1f32`	512	8
`__builtin_amdgcn_mfma_f32_32x32x2f32`	4096	64
`__builtin_amdgcn_mfma_f32_16x16x4f32`	2048	32
`__builtin_amdgcn_mfma_f32_32x32x4f16`	16384	64
`__builtin_amdgcn_mfma_f32_16x16x4f16`	8192	32
`__builtin_amdgcn_mfma_f32_4x4x4f16`	2048	8
`__builtin_amdgcn_mfma_f32_32x32x8f16`	16384	32
`__builtin_amdgcn_mfma_f32_16x16x16f16`	8192	16
`__builtin_amdgcn_mfma_f32_32x32x16_f16`	32768	32
`__builtin_amdgcn_mfma_f32_16x16x32_f16`	16384	16
`__builtin_amdgcn_mfma_f32_32x32x4bf16_1k`	16384	64
`__builtin_amdgcn_mfma_f32_16x16x4bf16_1k`	8192	32
`__builtin_amdgcn_mfma_f32_4x4x4bf16_1k`	2048	8
`__builtin_amdgcn_mfma_f32_32x32x8bf16_1k`	16384	32
`__builtin_amdgcn_mfma_f32_16x16x16bf16_1k`	8192	16
`__builtin_amdgcn_mfma_f32_32x32x16_bf16`	32768	32
`__builtin_amdgcn_mfma_f32_16x16x32_bf16`	16384	16
`__builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8`	32768	32
`__builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8`	32768	32
`__builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8`	32768	32
`__builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8`	32768	32
`__builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8`	16384	16
`__builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8`	16384	16
`__builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8`	16384	16
`__builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8`	16384	16
`__builtin_amdgcn_mfma_scale_f32_32x32x64_f8f6f4`	131072	32 (FP4/FP6), 64 (FP8/BF8)

continues on next page

Table 2 – continued from previous page

Builtin	Ops	Cycle count
`__builtin_amdgcn_mfma_scale_f32_16x16x128_f8f6f4`	65536	16 (FP4/FP6), 32 (FP8/BF8)
`__builtin_amdgcn_mfma_f64_16x16x4f64`	2048	64
`__builtin_amdgcn_mfma_f64_4x4x4f64`	512	32
`__builtin_amdgcn_mfma_i32_32x32x4i8`	16384	64
`__builtin_amdgcn_mfma_i32_16x16x4i8`	8192	32
`__builtin_amdgcn_mfma_i32_4x4x4i8`	2048	8
`__builtin_amdgcn_mfma_i32_32x32x16_i8`	32768	32
`__builtin_amdgcn_mfma_i32_16x16x32_i8`	16384	16
`__builtin_amdgcn_mfma_i32_32x32x32_i8`	65536	32
`__builtin_amdgcn_mfma_i32_16x16x64_i8`	32768	16

See also

CDNA4 MFMA transpose load builtins – Transpose load builtins that load operands directly from LDS into the per-lane fragment layout expected by the builtins below, without a software shuffle step.

14.1.4.8 Builtin reference

14.1.4.8.1 FP32-accumulate builtins

These builtins accumulate into single-precision (FP32) output fragments. They differ in the data type of the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrix inputs.

14.1.4.8.1.1 FP32 matrix inputs

The following builtins accept one FP32 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP32 output fragments.

14.1.4.8.1.2 `__builtin_amdgcn_mfma_f32_32x32x1f32`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x1f32(
    float    srcA,
    float    srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v32float	Accumulator input: 32 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v32float – updated accumulator (srcA × srcB + srcC).

14.1.4.8.1.3 __builtin_amdgcn_mfma_f32_16x16x1f32

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x1f32(
    float   srcA,
    float   srcB,
    v16float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA × srcB + srcC).

14.1.4.8.1.4 __builtin_amdgcn_mfma_f32_4x4x1f32

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x1f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 1$ FP32 column element of $\boldsymbol{A}$ and $K = 1$ FP32 row element of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: 4 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.5 __builtin_amdgcn_mfma_f32_32x32x2f32

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x2f32(
    float   srcA,
    float   srcB,
    v16float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 32 $\times$ 32 FP32 outer-product accumulation. Each instruction processes $K = 2$ FP32 column elements of $\boldsymbol{A}$ and $K = 2$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 32 $\times$ 32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
srcA	float	One element of $\boldsymbol{A}$ per lane.
srcB	float	One element of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be 0.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be 0.
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.6 __builtin_amdgcn_mfma_f32_16x16x4f32

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x4f32(
    float   srcA,
    float   srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16 $\times$ 16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP32 column elements of $\boldsymbol{A}$ and $K = 4$ FP32 row elements of $\boldsymbol{B}$, accumulating into a 16 $\times$ 16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	float	One element of $\boldsymbol{A}$ per lane.
`srcB`	float	One element of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.7 FP16 matrix inputs

CDNA4 supports two FP16 builtin families that differ in K step and block count.

The narrow-K multi-block variants accept four FP16 elements per lane packed into a `v4half` register:

14.1.4.8.1.8 `__builtin_amdgcn_mfma_f32_32x32x4f16`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x4f16(
    v4half   srcA,
    v4half   srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.9 `__builtin_amdgcn_mfma_f32_16x16x4f16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x4f16(
    v4half   srcA,
    v4half   srcB,
    v16float srcC,
    int      cbsz,
```

(continues on next page)

(continued from previous page)

```
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v16float	Accumulator input: 16 FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v16float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.10 __builtin_amdgcn_mfma_f32_4x4x4f16

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x4f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 4$ FP16 column elements of $\boldsymbol{A}$ and $K = 4$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across four accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
srcA	v4half	Four elements of $\boldsymbol{A}$ per lane.
srcB	v4half	Four elements of $\boldsymbol{B}$ per lane.
srcC	v4float	Accumulator input: Four FP32 elements per lane.
cbsz	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
blgp	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns v4float – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.11 __builtin_amdgcn_mfma_f32_32x32x8f16

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x8f16(
    v4half   srcA,
```

(continues on next page)

(continued from previous page)

```
    v4half   srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 8$ FP16 column elements of $\boldsymbol{A}$ and $K = 8$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.12 `__builtin_amdgcn_mfma_f32_16x16x16f16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x16f16(
    v4half  srcA,
    v4half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP16 column elements of $\boldsymbol{A}$ and $K = 16$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across four accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4half	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	v4half	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: Four FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

The wider-K single-block variants accept eight FP16 elements per lane packed into a `v8half` register:

14.1.4.8.1.13 `__builtin_amdgcn_mfma_f32_32x32x16_f16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_f16(
    v8half   srcA,
    v8half   srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP16→FP32 matrix multiply-accumulate. Each instruction processes $K = 16$ FP16 column elements of $\boldsymbol{A}$ and $K = 16$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile. Each lane holds eight FP16 values packed into one `v8half` register for each of $\boldsymbol{A}$ and $\boldsymbol{B}$.

This is a **single-block** variant. The result is a `v16float` (16 FP32 elements per lane) using accVGPRs 0–15 in the 32×32 layout.

Parameter	Type	Description
`srcA`	v8half	Eight FP16 elements of $\boldsymbol{A}$ per lane, packed into a 128-bit register.
`srcB`	v8half	Eight FP16 elements of $\boldsymbol{B}$ per lane, packed into a 128-bit register.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Reserved. Must be `0` (single-block variant; broadcast modifier not supported).
`abid`	int	Reserved. Must be `0` (single-block variant; block ID modifier not supported).
`blgp`	int	Lane-group pattern modifier. See *Common MFMA parameters*.

Returns `v16float` – updated accumulator.

> **Note**
>
> This single-block variant does not support the `cbsz` and `abid` broadcast modifiers; both must be `0`.

14.1.4.8.1.14 `__builtin_amdgcn_mfma_f32_16x16x32_f16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_f16(
    v8half  srcA,
    v8half  srcB,
    v4float srcC,
    int     cbsz,
    int     abid,
    int     blgp);
```

Computes one step of a 16×16 FP16→FP32 matrix multiply-accumulate. Each instruction processes $K = 32$ FP16 column elements of $\boldsymbol{A}$ and $K = 32$ FP16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile. Each lane holds eight FP16 values packed into one `v8half` register for each of $\boldsymbol{A}$ and $\boldsymbol{B}$.

This is a **single-block** variant. The result is a `v4float` (4 FP32 elements per lane) using accVGPRs 0–3 in the 16×16 layout.

Parameter	Type	Description
`srcA`	v8half	Eight FP16 elements of $\boldsymbol{A}$ per lane, packed into a 128-bit register.
`srcB`	v8half	Eight FP16 elements of $\boldsymbol{B}$ per lane, packed into a 128-bit register.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Reserved. Must be `0` (single-block variant; broadcast modifier not supported).
`abid`	int	Reserved. Must be `0` (single-block variant; block ID modifier not supported).
`blgp`	int	Lane-group pattern modifier. See *Common MFMA parameters*.

Returns `v4float` – updated accumulator.

Note

This single-block variant does not support the `cbsz` and `abid` broadcast modifiers; both must be `0`.

14.1.4.8.1.15 BF16 matrix inputs

CDNA4 supports two BF16 builtin families that differ in K step and block count.

The narrow-K `_1k` multi-block variants, inherited from CDNA3, accept four BF16 elements per lane packed into a `v4bfloat` register:

14.1.4.8.1.16 `__builtin_amdgcn_mfma_f32_32x32x4bf16_1k`

Signature and parameters for this builtin.

```
v32float __builtin_amdgcn_mfma_f32_32x32x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v32float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v32float	Accumulator input: 32 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.17 `__builtin_amdgcn_mfma_f32_16x16x4bf16_1k`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_16x16x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.18 `__builtin_amdgcn_mfma_f32_4x4x4bf16_1k`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_4x4x4bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 4×4 FP32 outer-product accumulation. Each instruction processes $K = 4$ BF16 column elements of $\boldsymbol{A}$ and $K = 4$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.19 `__builtin_amdgcn_mfma_f32_32x32x8bf16_1k`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x8bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 8$ BF16 column elements of $\boldsymbol{A}$ and $K = 8$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.20 `__builtin_amdgcn_mfma_f32_16x16x16bf16_1k`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x16bf16_1k(
    v4bfloat srcA,
    v4bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 16$ BF16 column elements of $\boldsymbol{A}$ and $K = 16$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	v4bfloat	Four BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v4bfloat	Four BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

The wider-K single-block variants accept eight BF16 elements per lane packed into a `v8bfloat` register:

14.1.4.8.1.21 `__builtin_amdgcn_mfma_f32_32x32x16_bf16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_bf16(
    v8bfloat srcA,
    v8bfloat srcB,
    v16float srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 32×32 BF16→FP32 matrix multiply-accumulate. Each instruction processes $K = 16$ BF16 column elements of $\boldsymbol{A}$ and $K = 16$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile. Each lane holds eight BF16 values packed into one `v8bfloat` register (stored as `short`) for each of $\boldsymbol{A}$ and $\boldsymbol{B}$.

This is a **single-block** variant. The result is a `v16float` (16 FP32 elements per lane) using accVGPRs 0–15 in the 32×32 layout.

Parameter	Type	Description
`srcA`	v8bfloat	Eight BF16 elements of $\boldsymbol{A}$ per lane, stored as `short`, packed into a 128-bit register.
`srcB`	v8bfloat	Eight BF16 elements of $\boldsymbol{B}$ per lane, stored as `short`, packed into a 128-bit register.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Reserved. Must be `0` (single-block variant; broadcast modifier not supported).
`abid`	int	Reserved. Must be `0` (single-block variant; block ID modifier not supported).
`blgp`	int	Lane-group pattern modifier. See *Common MFMA parameters*.

Returns `v16float` – updated accumulator.

> **Note**
>
> This single-block variant does not support the `cbsz` and `abid` broadcast modifiers; both must be `0`.

14.1.4.8.1.22 `__builtin_amdgcn_mfma_f32_16x16x32_bf16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_bf16(
    v8bfloat srcA,
    v8bfloat srcB,
    v4float  srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 BF16→FP32 matrix multiply-accumulate. Each instruction processes $K = 32$ BF16 column elements of $\boldsymbol{A}$ and $K = 32$ BF16 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile. Each lane holds eight BF16 values packed into one `v8bfloat` register (stored as `short`) for each of $\boldsymbol{A}$ and $\boldsymbol{B}$.

This is a **single-block** variant. The result is a `v4float` (4 FP32 elements per lane) using accVGPRs 0–3 in the 16×16 layout.

Parameter	Type	Description
`srcA`	v8bfloat	Eight BF16 elements of $\boldsymbol{A}$ per lane, stored as `short`, packed into a 128-bit register.
`srcB`	v8bfloat	Eight BF16 elements of $\boldsymbol{B}$ per lane, stored as `short`, packed into a 128-bit register.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Reserved. Must be `0` (single-block variant; broadcast modifier not supported).
`abid`	int	Reserved. Must be `0` (single-block variant; block ID modifier not supported).
`blgp`	int	Lane-group pattern modifier. See *Common MFMA parameters*.

Returns `v4float` – updated accumulator.

Note

This single-block variant does not support the `cbsz` and `abid` broadcast modifiers; both must be `0`.

14.1.4.8.1.23 FP8 and BF8 matrix inputs

The following builtins accept eight 8-bit floating-point values per lane packed into a `long long` register. Two 8-bit formats are supported:

- **FP8** (`fp8`): E4M3 format (1 sign, 4 exponent, 3 mantissa bits).
- **BF8** (`bf8`): E5M2 format (1 sign, 5 exponent, 2 mantissa bits).

The $\boldsymbol{A}$ and $\boldsymbol{B}$ matrices can use different formats, giving four combinations per tile shape.

14.1.4.8.1.24 `__builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_fp8_fp8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP8 column elements of $\boldsymbol{A}$ and $K = 16$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the FP8 (E4M3) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.25 `__builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_fp8_bf8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ FP8 column elements of $\boldsymbol{A}$ and $K = 16$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the FP8 (E4M3) format and $\boldsymbol{B}$ uses the BF8 (E5M2) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.26 `__builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_bf8_fp8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ BF8 column elements of $\boldsymbol{A}$ and $K = 16$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the BF8 (E5M2) format and $\boldsymbol{B}$ uses the FP8 (E4M3) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.27 `__builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_f32_32x32x16_bf8_bf8(
    long long srcA,
    long long srcB,
    v16float  srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 FP32 outer-product accumulation. Each instruction processes $K = 16$ BF8 column elements of $\boldsymbol{A}$ and $K = 16$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the BF8 (E5M2) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.28 `__builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_fp8_fp8(
    long long srcA,
    long long srcB,
    v4float   srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ FP8 column elements of $\boldsymbol{A}$ and $K = 32$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the FP8 (E4M3) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.29 `__builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_fp8_bf8(
    long long srcA,
    long long srcB,
    v4float   srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ FP8 column elements of $\boldsymbol{A}$ and $K = 32$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the FP8 (E4M3) format and $\boldsymbol{B}$ uses the BF8 (E5M2) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.30 `__builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_bf8_fp8(
    long long srcA,
    long long srcB,
```

(continues on next page)

(continued from previous page)

```
    v4float   srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ BF8 column elements of $\boldsymbol{A}$ and $K = 32$ FP8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

$\boldsymbol{A}$ uses the BF8 (E5M2) format and $\boldsymbol{B}$ uses the FP8 (E4M3) format; eight 8-bit values are packed into a `long long` register per lane for each operand.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight FP8 (E4M3) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.1.31 `__builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_f32_16x16x32_bf8_bf8(
    long long srcA,
    long long srcB,
    v4float   srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 FP32 outer-product accumulation. Each instruction processes $K = 32$ BF8 column elements of $\boldsymbol{A}$ and $K = 32$ BF8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Both $\boldsymbol{A}$ and $\boldsymbol{B}$ use the BF8 (E5M2) format: eight 8-bit values packed into a `long long` register per lane.

Parameter	Type	Description
`srcA`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight BF8 (E5M2) elements of $\boldsymbol{B}$ per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.2 FP64-accumulate builtins

CDNA4 retains native double-precision (FP64) matrix accumulation from CDNA3. These builtins accept one FP64 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP64 output fragments held in `v4double` registers.

Note

On CDNA4, the `blgp` parameter of FP64 builtins is repurposed as a 3-bit source negation modifier rather than a lane-group pattern selector. See the *Common parameters* section for details.

14.1.4.8.2.1 FP64 matrix inputs

The following builtins accept one FP64 element per lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs and accumulate into FP64 output fragments.

14.1.4.8.2.2 `__builtin_amdgcn_mfma_f64_16x16x4f64`

Signature and parameters for this builtin.

```
v4double __builtin_amdgcn_mfma_f64_16x16x4f64(
    double   srcA,
    double   srcB,
    v4double srcC,
    int      cbsz,
    int      abid,
    int      blgp);
```

Computes one step of a 16×16 FP64 outer-product accumulation. Each instruction processes $K = 4$ FP64 column elements of $\boldsymbol{A}$ and $K = 4$ FP64 row elements of $\boldsymbol{B}$, accumulating into a 16×16 FP64 tile held across 4 accVGPRs per lane (8 physical accVGPRs, since each FP64 value occupies two 32-bit registers). This builtin operates on a single input block.

Parameter	Type	Description
`srcA`	double	One FP64 element of $\boldsymbol{A}$ per lane.
`srcB`	double	One FP64 element of $\boldsymbol{B}$ per lane.
`srcC`	v4doub	Accumulator input: 4 FP64 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, repurposed as a negation modifier (3-bit field). Bit 0 negates `srcA`, bit 1 negates `srcB`, bit 2 negates `srcC`.

Returns `v4double` – updated accumulator (srcA $\times$ srcB + srcC).

Note

Unlike most FP32, BF16, and INT8 MFMA instructions, FP64 MFMA instructions cannot co-execute with VALU instructions. To hide execution latency, schedule non-VALU instructions (loads, stores, or branches) around FP64

MFMA operations.

14.1.4.8.2.3 `__builtin_amdgcn_mfma_f64_4x4x4f64`

Signature and parameters for this builtin.

```
double __builtin_amdgcn_mfma_f64_4x4x4f64(
    double srcA,
    double srcB,
    double srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 4×4 FP64 outer-product accumulation. Each instruction processes $K = 4$ FP64 column elements of $\boldsymbol{A}$ and $K = 4$ FP64 row elements of $\boldsymbol{B}$, accumulating into a 4×4 FP64 tile. Each lane holds one FP64 output element across four input blocks.

Parameter	Type	Description
`srcA`	double	One FP64 element of $\boldsymbol{A}$ per lane.
`srcB`	double	One FP64 element of $\boldsymbol{B}$ per lane.
`srcC`	double	Accumulator input: 1 FP64 element per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, repurposed as a negation modifier (3-bit field). Bit 0 negates `srcA`, bit 1 negates `srcB`, bit 2 negates `srcC`.

Returns `double` – updated accumulator (srcA $\times$ srcB $+$ srcC).

Note

Unlike most FP32, BF16, and INT8 MFMA instructions, FP64 MFMA instructions cannot co-execute with VALU instructions. To hide execution latency, schedule non-VALU instructions (loads, stores, or branches) around FP64 MFMA operations.

14.1.4.8.3 INT32-accumulate builtins

These builtins accumulate into signed 32-bit integer (INT32) output fragments.

14.1.4.8.3.1 INT8 matrix inputs

The following builtins accept four signed 8-bit integer elements per lane packed into a single `int` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs.

14.1.4.8.3.2 `__builtin_amdgcn_mfma_i32_32x32x4i8`

Signature and parameters for this builtin.

```
v32int __builtin_amdgcn_mfma_i32_32x32x4i8(
    int    srcA,
    int    srcB,
    v32int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 32×32 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 32 accVGPRs per lane. This builtin operates on two input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v32int	Accumulator input: 32 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..1
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..1
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v32int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.3.3 `__builtin_amdgcn_mfma_i32_16x16x4i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_16x16x4i8(
    int    srcA,
    int    srcB,
    v16int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 16 accVGPRs per lane. This builtin operates on four input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..2
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..3
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.1.4.8.3.4 `__builtin_amdgcn_mfma_i32_4x4x4i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_4x4x4i8(
    int   srcA,
    int   srcB,
    v4int srcC,
    int   cbsz,
    int   abid,
    int   blgp);
```

Computes one step of a 4×4 INT32 outer-product accumulation. Each instruction processes $K = 4$ INT8 column elements of $\boldsymbol{A}$ and $K = 4$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 4×4 INT32 tile held across 4 accVGPRs per lane. This builtin operates on 16 input blocks.

Parameter	Type	Description
`srcA`	int	Four elements of $\boldsymbol{A}$ per lane.
`srcB`	int	Four elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Legal range: 0..4
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Legal range: 0..15
`blgp`	int	$\boldsymbol{B}$-matrix Lane Group Pattern modifier, see *Common MFMA parameters*.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

The following wider-K variants accept eight signed 8-bit integer elements per lane packed into a `long long` register:

14.1.4.8.3.5 `__builtin_amdgcn_mfma_i32_32x32x16_i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_32x32x16_i8(
    long long srcA,
    long long srcB,
    v16int    srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 32×32 INT32 outer-product accumulation. Each instruction processes $K = 16$ INT8 column elements of $\boldsymbol{A}$ and $K = 16$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 16 accVGPRs per lane. This builtin operates on a single input block.

Each lane provides eight signed 8-bit integer values packed into a `long long` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$.

Parameter	Type	Description
`srcA`	long long	Eight INT8 elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight INT8 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v16int` – updated accumulator (srcA × srcB + srcC).

14.1.4.8.3.6 `__builtin_amdgcn_mfma_i32_16x16x32_i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_16x16x32_i8(
    long long srcA,
    long long srcB,
    v4int     srcC,
    int       cbsz,
    int       abid,
    int       blgp);
```

Computes one step of a 16×16 INT32 outer-product accumulation. Each instruction processes $K = 32$ INT8 column elements of $\boldsymbol{A}$ and $K = 32$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 4 accVGPRs per lane. This builtin operates on a single input block.

Each lane provides eight signed 8-bit integer values packed into a `long long` register for both $\boldsymbol{A}$ and $\boldsymbol{B}$.

Parameter	Type	Description
`srcA`	long long	Eight INT8 elements of $\boldsymbol{A}$ per lane.
`srcB`	long long	Eight INT8 elements of $\boldsymbol{B}$ per lane.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Control Broadcast Size modifier, see *Common MFMA parameters*. Must be `0`.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Common MFMA parameters*. Must be `0`.
`blgp`	int	Must be `0`.

Returns `v4int` – updated accumulator (srcA × srcB + srcC).

The following CDNA4 wider-K single-block variants accept 16 signed 8-bit integer elements per lane packed into a `v4int` (four 32-bit words) register:

14.1.4.8.3.7 `__builtin_amdgcn_mfma_i32_32x32x32_i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_mfma_i32_32x32x32_i8(
    v4int  srcA,
    v4int  srcB,
    v16int srcC,
    int    cbsz,
    int    abid,
    int    blgp);
```

Computes one step of a 32×32 INT8→INT32 matrix multiply-accumulate. Each instruction processes $K = 32$ INT8 column elements of $\boldsymbol{A}$ and $K = 32$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile. Each lane holds 16 signed 8-bit integers packed into a `v4int` (four 32-bit words) for each of $\boldsymbol{A}$ and $\boldsymbol{B}$.

This is a **single-block** variant. The result is a `v16int` (16 INT32 elements per lane) using accVGPRs 0–15 in the 32×32 layout.

Parameter	Type	Description
`srcA`	v4int	Sixteen INT8 elements of $\boldsymbol{A}$ per lane, packed as four 32-bit words.
`srcB`	v4int	Sixteen INT8 elements of $\boldsymbol{B}$ per lane, packed as four 32-bit words.
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`cbsz`	int	Reserved. Must be `0` (single-block variant; broadcast modifier not supported).
`abid`	int	Reserved. Must be `0` (single-block variant; block ID modifier not supported).
`blgp`	int	Lane-group pattern modifier. See *Common MFMA parameters*.

Returns `v16int` – updated accumulator.

Note

This single-block variant does not support the `cbsz` and `abid` broadcast modifiers; both must be `0`.

14.1.4.8.3.8 `__builtin_amdgcn_mfma_i32_16x16x64_i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_mfma_i32_16x16x64_i8(
    v4int srcA,
    v4int srcB,
    v4int srcC,
    int   cbsz,
    int   abid,
    int   blgp);
```

Computes one step of a 16×16 INT8→INT32 matrix multiply-accumulate. Each instruction processes $K = 64$ INT8 column elements of $\boldsymbol{A}$ and $K = 64$ INT8 row elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile. Each lane holds 16 signed 8-bit integers packed into a `v4int` (four 32-bit words) for each of $\boldsymbol{A}$ and $\boldsymbol{B}$.

This is a **single-block** variant. The result is a `v4int` (4 INT32 elements per lane) using accVGPRs 0–3 in the 16×16 layout.

Parameter	Type	Description
`srcA`	v4int	Sixteen INT8 elements of $\boldsymbol{A}$ per lane, packed as four 32-bit words.
`srcB`	v4int	Sixteen INT8 elements of $\boldsymbol{B}$ per lane, packed as four 32-bit words.
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`cbsz`	int	Reserved. Must be `0` (single-block variant; broadcast modifier not supported).
`abid`	int	Reserved. Must be `0` (single-block variant; block ID modifier not supported).
`blgp`	int	Lane-group pattern modifier. See *Common MFMA parameters*.

Returns `v4int` – updated accumulator.

Note

This single-block variant does not support the `cbsz` and `abid` broadcast modifiers; both must be `0`.

14.1.4.8.4 Mixed-precision with scaling

CDNA4 introduces scaled sub-byte matrix builtins (`mfma_scale_*`) that support FP8 (E4M3), BF8 (E5M2), FP6 (E2M3), FP6 (E3M2), and FP4 (E2M1) input formats with per-block scaling. Unlike the standard MFMA builtins, the `cbsz` and `blgp` parameters select the **input format** for $\boldsymbol{A}$ and $\boldsymbol{B}$ respectively rather than acting as broadcast or lane-group-pattern modifiers. The vector type of `srcA` and `srcB` must match the selected format: `v8int` for 8-bit formats, `v6int` for 6-bit formats, and `v4int` for 4-bit formats.

Each instruction also accepts a pair of per-block scale values loaded with `v_mfma_ld_scale_b32`. The 2-bit `op_sel_a` and `op_sel_b` immediates select which quarter of the scale register applies to each output block.

14.1.4.8.4.1 FP8, FP6, and FP4 matrix inputs

The following builtins accept scaled sub-byte matrix inputs with per-block scaling.

14.1.4.8.4.2 `__builtin_amdgcn_mfma_scale_f32_32x32x64_f8f6f4`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_mfma_scale_f32_32x32x64_f8f6f4(
    <vector>  srcA,
    <vector>  srcB,
    v16float  srcC,
    int       cbsz,
    int       blgp,
    int       op_sel_a,
    int       scale_a,
    int       op_sel_b,
    int       scale_b);
```

Computes one step of a 32×32 scaled sub-byte matrix multiply-accumulate. Each instruction processes $K = 64$ elements drawn from a sub-byte floating-point format and accumulates into a 32×32 FP32 tile. The $\boldsymbol{A}$ and $\boldsymbol{B}$ input formats are selected independently at compile time by `cbsz` and `blgp`, and each block of output elements is scaled by a per-block scale factor loaded with `v_mfma_ld_scale_b32`.

This is a **single-block** variant. The result is a `v16float` (16 FP32 elements per lane) using accVGPRs 0–15 in the 32×32 layout.

The vector type of `srcA` and `srcB` depends on the format specified by `cbsz` and `blgp` respectively:

Format code (`cbsz` / `blgp`)	Format	Vector type	Bits per element
0	FP8 (E4M3)	`v8int` (8 × `int`)	8
1	BF8 (E5M2)	`v8int` (8 × `int`)	8
2	FP6 (E2M3)	`v6int` (6 × `int`)	6
3	FP6 (E3M2)	`v6int` (6 × `int`)	6
4	FP4 (E2M1)	`v4int` (4 × `int`)	4

The `srcA` vector type must be consistent with the format selected by `cbsz`, and the `srcB` vector type must be consistent with the format selected by `blgp`. Mixing a vector type with an incompatible format code is undefined behavior.

Parameter	Type	Description
`srcA`	`v8int` / `v6int` / `v4int`	$\boldsymbol{A}$ matrix fragment per lane; element type and packing determined by `cbsz`. Pass as `int [[clang::ext_vector_type(N)]]` where N is 8, 6, or 4.
`srcB`	`v8int` / `v6int` / `v4int`	$\boldsymbol{B}$ matrix fragment per lane; element type and packing determined by `blgp`.
`srcC`	v16float	Accumulator input: 16 FP32 elements per lane.
`cbsz`	int (immediate)	Format selector for $\boldsymbol{A}$. Must be a compile-time constant in the range 0–4. See the format table above.
`blgp`	int (immediate)	Format selector for $\boldsymbol{B}$. Must be a compile-time constant in the range 0–4. See the format table above.
`op_sel`	int (immediate)	2-bit selector that picks which quarter of `scale_a` applies to each output block. Must be a compile-time constant.
`scale_`	int	Per-block scale register for $\boldsymbol{A}$. Applied to all $\boldsymbol{A}$ input values before accumulation.
`op_sel`	int (immediate)	2-bit selector that picks which quarter of `scale_b` applies to each output block. Must be a compile-time constant.
`scale_`	int	Per-block scale register for $\boldsymbol{B}$. Applied to all $\boldsymbol{B}$ input values before accumulation.

Returns `v16float` – updated accumulator.

Note

Unlike the standard `cbsz` and `blgp` modifiers described in *Common MFMA parameters*, in this builtin `cbsz` and `blgp` are **format selectors**, not broadcast and lane-group-pattern modifiers.

Note

This instruction uses round-toward-nearest-even semantics.

14.1.4.8.4.3 `__builtin_amdgcn_mfma_scale_f32_16x16x128_f8f6f4`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_mfma_scale_f32_16x16x128_f8f6f4(
    <vector>  srcA,
    <vector>  srcB,
    v4float   srcC,
    int       cbsz,
    int       blgp,
    int       op_sel_a,
    int       scale_a,
    int       op_sel_b,
    int       scale_b);
```

Computes one step of a 16×16 scaled sub-byte matrix multiply-accumulate. Each instruction processes $K = 128$ elements drawn from a sub-byte floating-point format and accumulates into a 16×16 FP32 tile. The $\boldsymbol{A}$ and $\boldsymbol{B}$ input

formats are selected independently at compile time by `cbsz` and `blgp`, and each block of output elements is scaled by a per-block scale factor loaded with `v_mfma_ld_scale_b32`.

This is a **single-block** variant. The result is a `v4float` (4 FP32 elements per lane) using accVGPRs 0–3 in the 16×16 layout.

The vector type of `srcA` and `srcB` depends on the format specified by `cbsz` and `blgp` respectively:

Format code (`cbsz` / `blgp`)	Format	Vector type	Bits per element
0	FP8 (E4M3)	`v8int` (8 × `int`)	8
1	BF8 (E5M2)	`v8int` (8 × `int`)	8
2	FP6 (E2M3)	`v6int` (6 × `int`)	6
3	FP6 (E3M2)	`v6int` (6 × `int`)	6
4	FP4 (E2M1)	`v4int` (4 × `int`)	4

The `srcA` vector type must be consistent with the format selected by `cbsz`, and the `srcB` vector type must be consistent with the format selected by `blgp`. Mixing a vector type with an incompatible format code is undefined behavior.

Parameter	Type	Description
`srcA`	`v8int` / `v6int` / `v4int`	$\boldsymbol{A}$ matrix fragment per lane; element type and packing determined by `cbsz`. Pass as `int [[clang::ext_vector_type(N)]]` where N is 8, 6, or 4.
`srcB`	`v8int` / `v6int` / `v4int`	$\boldsymbol{B}$ matrix fragment per lane; element type and packing determined by `blgp`.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`cbsz`	int (immediate)	Format selector for $\boldsymbol{A}$. Must be a compile-time constant in the range 0–4. See the format table above.
`blgp`	int (immediate)	Format selector for $\boldsymbol{B}$. Must be a compile-time constant in the range 0–4. See the format table above.
`op_sel`	int (immediate)	2-bit selector that picks which quarter of `scale_a` applies to each output block. Must be a compile-time constant.
`scale_`	int	Per-block scale register for $\boldsymbol{A}$, loaded with `v_mfma_ld_scale_b32`. Applied to all $\boldsymbol{A}$ input values before accumulation.
`op_sel`	int (immediate)	2-bit selector that picks which quarter of `scale_b` applies to each output block. Must be a compile-time constant.
`scale_`	int	Per-block scale register for $\boldsymbol{B}$, loaded with `v_mfma_ld_scale_b32`. Applied to all $\boldsymbol{B}$ input values before accumulation.

Returns `v4float` – updated accumulator.

Note

Unlike the standard `cbsz` and `blgp` modifiers described in *Common MFMA parameters*, in this builtin `cbsz` and `blgp` are **format selectors**, not broadcast and lane-group-pattern modifiers.

Note

This builtin uses round-toward-nearest-even semantics.

14.2 Sparse MFMA builtins

Sparse Matrix Fused Multiply-Accumulate (SMFMAC) builtins multiply a compressed $\boldsymbol{A}$ tile by a dense $\boldsymbol{B}$ tile and accumulate the result into a $\boldsymbol{C}$ tile, exploiting 4:2 structured sparsity to halve the storage and bandwidth required for $\boldsymbol{A}$. Select a CDNA generation below for the corresponding builtin reference. For trailing parameter documentation, see *Common MFMA parameters*.

- *CDNA and CDNA2 sparse MFMA builtins*
- *CDNA3 sparse MFMA builtins*
- *CDNA4 sparse MFMA builtins*

14.2.1 CDNA and CDNA2 sparse MFMA builtins

Sparse Matrix Fused Multiply-Accumulate (SMFMAC) builtins let you issue hardware matrix multiply-accumulate operations that exploit 4:2 structured sparsity directly from HIP device code on CDNA GPUs (`gfx908`, MI100) and CDNA2 GPUs (`gfx90a`, MI200 series). Each SMFMAC instruction multiplies a compressed $\boldsymbol{A}$ fragment by a dense $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single wavefront of 64 lanes. Because the $\boldsymbol{A}$ operand is stored in compressed form, these builtins halve the storage and memory bandwidth required for $\boldsymbol{A}$ compared to their dense MFMA counterparts, while the hardware uses a sparsity index to reconstruct the original element positions during the multiply.

14.2.1.1 Architecture availability

The builtins on this page target CDNA (`gfx908`, MI100) and CDNA2 (`gfx90a`, MI200 series). Equivalent builtins for later CDNA generations are documented on their own reference pages:

- *CDNA3 sparse MFMA builtins* – CDNA3 (`gfx942`, MI300 series)
- *CDNA4 sparse MFMA builtins* – CDNA4 (`gfx950`, MI350 series)

14.2.1.2 Naming convention

All SMFMAC builtins follow the pattern:

```
__builtin_amdgcn_smfmac_<out_type>_<M>x<N>x<K>_<in_type_a>[_<in_type_b>]
```

`out_type`
: Accumulator element type (`f32` or `i32`).

`M`, `N`, `K`
: Tile dimensions in elements. The instruction computes the contribution of a K-wide compressed panel of $\boldsymbol{A}$ and a K-wide dense panel of $\boldsymbol{B}$ to an $M \times N$ output tile. Each instruction processes one K step; the caller loops over K to accumulate a full matrix product.

`in_type_a`
: Input element type of the $\boldsymbol{A}$ matrix (`f16`, `bf16`, or `i8`).

`in_type_b` (FP8/BF8 variants only)
: Input element type of the $\boldsymbol{B}$ matrix. Always present for FP8 and BF8 variants to disambiguate the four possible A×B type combinations (`fp8_fp8`, `fp8_bf8`, `bf8_fp8`, `bf8_bf8`). Omitted for FP16, BF16, and INT8 variants where $\boldsymbol{A}$ and $\boldsymbol{B}$ always share the same type.

For example:

- `__builtin_amdgcn_smfmac_f32_16x16x32_f16` – a 16×16 sparse MMA with $K = 32$ that multiplies FP16 $\boldsymbol{A}$ by FP16 $\boldsymbol{B}$ and accumulates into FP32.

14.2.1.3 Structured sparsity (4:2 pattern)

The SMFMAC instructions require the $\boldsymbol{A}$ matrix to obey 4:2 structured sparsity: in every contiguous group of four elements along the K dimension, exactly two are non-zero and the other two are zero. This constraint allows the $\boldsymbol{A}$ operand to be stored in a compressed representation that contains only the non-zero values, reducing the storage to half the original K dimension.

Along with the compressed non-zero values, the hardware requires a sparsity index that encodes which two of the four positions in each group hold the non-zeros. This index is passed as the `idx` argument to every SMFMAC builtin. The hardware uses it at execution time to align the compressed $\boldsymbol{A}$ elements against the correct rows of the $\boldsymbol{B}$ operand before accumulating the products.

Host-side compression is straightforward: walk the K dimension in groups of four, extract the two non-zero positions, and pack them consecutively into the output buffer. The example kernels in this topic adopt a fixed pattern that keeps positions 0 and 2 of every group, producing a compressed buffer of half the original K length.

14.2.1.4 Accumulator layout

Every SMFMAC instruction computes a single independent $M \times N$ output tile (block count = 1). The accumulator ($\boldsymbol{C}$ / $\boldsymbol{D}$) layout across wavefront lanes and VGPRs is identical to the 1-block layout of the corresponding dense MFMA tile shape.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \leq i < M$
- j – zero-based column index within the tile, $0 \leq j < N$
- **lane** – wavefront lane that holds the element, $0 \leq \text{lane} < 64$
- **VGPR** – zero-based index into that lane's accumulator register file

14.2.1.4.1 16×16 layout

The 16×16 output tile occupies 4 VGPRs per lane (`v4float` or `v4int`).

Given output element (i, j):

$$\text{lane} = 16\lfloor\frac{i}{4}\rfloor + j$$
$$\text{VGPR} = i \bmod 4$$

Conversely, given lane L and VGPR index G:

$$i = 4\lfloor\frac{L}{16}\rfloor + (G \bmod 4)$$
$$j = L \bmod 16$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0—3	0—15	0—3
4—7	16—31	0—3
8—11	32—47	0—3
12—15	48—63	0—3

A (sparse, 2:4)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=2	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=3	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=4	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=5	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=6	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=7	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=8	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=9	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=10	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=11	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=12	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=13	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=14	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=15	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1

A (sparse): 2 VGPRs (v0–v1), 2:4 structured sparsity along the k-dimension; VGPR = $\lfloor k/4 \rfloor \bmod 2$, lane = $16 \cdot \lfloor k/8 \rfloor + i$. Colour groups of 8 k-columns alternate teal/grey.

Fig. 19: 16×16, **K=32 A (sparse, 2:4) operand layout.**

Compression index (v0)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=1	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=2	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=3	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=4	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=5	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=6	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=7	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=8	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=9	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=10	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=11	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=12	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=13	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=14	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=15	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]

Compression index: 1 VGPR (v0), nibble [3:0] for k∈{0..3,8..11,...}, nibble [7:4] for k∈{4..7,12..15,...}; same lane assignment as A. Colour groups of 8 k-columns alternate teal/grey.

Fig. 20: 16×16, **K=32 compression index layout.**

B (dense)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k=0	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l
k=1	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h
k=2	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l
k=3	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h
k=4	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l
k=5	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h
k=6	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l
k=7	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h
k=8	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l
k=9	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h
k=10	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l
k=11	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h
k=12	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l
k=13	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h
k=14	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l
k=15	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h
k=16	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l
k=17	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h
k=18	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l
k=19	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h
k=20	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l
k=21	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h
k=22	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l
k=23	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h
k=24	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l	v0l
k=25	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h	v0h
k=26	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l	v1l
k=27	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h	v1h
k=28	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l	v2l
k=29	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h	v2h
k=30	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l	v3l
k=31	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h	v3h

B (dense): 4 VGPRs (v0–v3), lo/hi 16-bit half per k-step; VGPR = $\lfloor k/2 \rfloor \bmod 4$, half = $k \bmod 2$, lane = $16 \cdot \lfloor k/16 \rfloor + j$. Colour groups of 8 k-rows alternate teal/grey.

Fig. 21: 16 × 16, **K=32 B (dense) operand layout.**

C / D (input / output)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=4	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=5	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=6	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=7	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=8	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=9	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=10	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=11	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=12	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=13	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=14	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=15	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3

C/D: 4 VGPRs (v0–v3); VGPR = i mod 4, lane = $16 \cdot \lfloor i/4 \rfloor$ + j. Colour groups of 4 i-rows alternate teal/grey.

Fig. 22: 16×16, **K=32 accumulator layout.** Each cell shows the VGPR index that holds output element (i, j). Rows 0—3 (teal, lanes 0—15), rows 4—7 (grey, lanes 16—31), rows 8—11 (teal, lanes 32—47), rows 12—15 (grey, lanes 48—63). Column j gives the lane offset within the group.

14.2.1.4.2 32×32 layout

The 32×32 output tile occupies 16 VGPRs per lane (`v16float` or `v16int`).

Given output element (i, j):

$$\text{lane} = \left(32 \cdot \lfloor \frac{i}{4} \rfloor\right) \bmod 64 + j$$

$$\text{VGPR} = 4\lfloor \frac{i}{8} \rfloor + (i \bmod 4)$$

Conversely, given lane L and VGPR index G:

$$i = \left(8 \cdot \lfloor \frac{G}{4} \rfloor\right) \bmod 32 + 4\lfloor \frac{L}{32} \rfloor + (G \bmod 4)$$

$$j = L \bmod 32$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0—3	0—31	0—3
4—7	32—63	0—3
8—11	0—31	4—7
12—15	32—63	4—7
16—19	0—31	8—11
20—23	32—63	8—11
24—27	0—31	12—15
28—31	32—63	12—15

14.2.1.5 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v4float  = float    [[clang::ext_vector_type(4)]];
using v16float = float    [[clang::ext_vector_type(16)]];
using v4half   = _Float16 [[clang::ext_vector_type(4)]];
using v8half   = _Float16 [[clang::ext_vector_type(8)]];
using v4short  = short    [[clang::ext_vector_type(4)]];   // BF16 storage
using v8short  = short    [[clang::ext_vector_type(8)]];   // BF16 storage
using v2int    = int      [[clang::ext_vector_type(2)]];
using v4int    = int      [[clang::ext_vector_type(4)]];
using v16int   = int      [[clang::ext_vector_type(16)]];
```

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane; the total VGPR count equals the element count multiplied by the element size in 32-bit words.

14.2.1.6 Common parameters

The SMFMAC builtins do not use the `cbsz`, `abid`, or `blgp` modifiers from the dense MFMA family. See *Common MFMA parameters* for a description of those modifiers in the dense context.

A (sparse, 2:4)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=1	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=2	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=3	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=4	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=5	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=6	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=7	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=8	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=9	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=10	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=11	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=12	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=13	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=14	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=15	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=16	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=17	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=18	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=19	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=20	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=21	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=22	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=23	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=24	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=25	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=26	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=27	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=28	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=29	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=30	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1
i=31	v0	v0	v0	v0	v1	v1	v1	v1	v0	v0	v0	v0	v1	v1	v1	v1

A (sparse): 2 VGPRs (v0–v1), 2:4 structured sparsity along the k-dimension; VGPR = $\lfloor k/4 \rfloor \bmod 2$, lane = $i + 32 \cdot \lfloor k/8 \rfloor$. Colour groups of 8 k-columns alternate teal/grey.

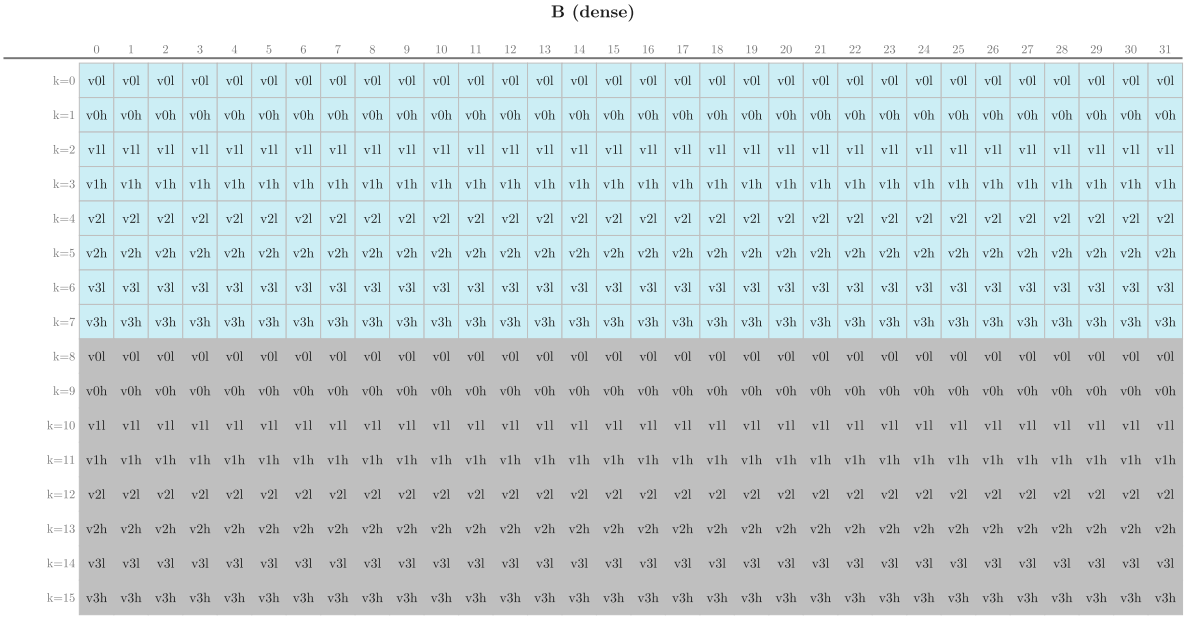
Fig. 23: 32×32, **K=16 A (sparse, 2:4) operand layout.**

Compression index (v0)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=1	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=2	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=3	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=4	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=5	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=6	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=7	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=8	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=9	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=10	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=11	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=12	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=13	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=14	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=15	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=16	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=17	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=18	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=19	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=20	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=21	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=22	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=23	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=24	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=25	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=26	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=27	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=28	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=29	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=30	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]
i=31	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]	[3:0]	[3:0]	[3:0]	[3:0]	[7:4]	[7:4]	[7:4]	[7:4]

Compression index: 1 VGPR (v0), nibble [3:0] for k∈{0..3,8..11}, nibble [7:4] for k∈{4..7,12..15}; same lane assignment as A. Colour groups of 8 k-columns alternate teal/grey.

Fig. 24: 32 × 32, **K=16 compression index layout.**



B (dense): 4 VGPRs (v0–v3), lo/hi 16-bit half per k-step; VGPR = ⌊k/2⌋ mod 4, half = k mod 2, lane = 32 · ⌊k/8⌋ + j. Colour groups of 8 k-rows alternate teal/grey.

Fig. 25: 32 × 32, **K=16 B (dense) operand layout.**

14.2.1.7 Using sparse MFMA builtins as a compute policy

The following example kernels demonstrate SMFMAC builtins in the context of a tiled matrix multiplication. Each kernel loads tiles of the compressed $\boldsymbol{A}$ matrix and the dense $\boldsymbol{B}$ matrix into LDS, then calls the SMFMAC builtin to replace the inner-product loop of a conventional scalar kernel.

The complete source file is available for download:

- `matrix_multiply_cdna_sparse_mfma.hip`

14.2.1.7.1 FP16 16×16 sparse kernel

This kernel uses `__builtin_amdgcn_smfmac_f32_16x16x32_f16` to compute a 16×16 sparse matrix multiply-accumulate with $K = 32$ per instruction. The compressed $\boldsymbol{A}$ operand is loaded as a `v4half` vector, the dense $\boldsymbol{B}$ as a `v8half` vector, and the accumulator is a `v4float` vector.

```
struct SmfmacCdna16x16F16Policy
{
    static constexpr int thread_tile_m = 16;
    static constexpr int thread_tile_n = 16;

    using v4float = float [[clang::ext_vector_type(4)]];
    using Accumulator = v4float;

    using v4half = _Float16 [[clang::ext_vector_type(4)]];
    using v8half = _Float16 [[clang::ext_vector_type(8)]];
    using AFrag = v4half;
    using BFrag = v8half;

```

(continues on next page)

C / D (input / output)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=4	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=5	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=6	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=7	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=8	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4
i=9	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5
i=10	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6
i=11	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7
i=12	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4	v4
i=13	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5	v5
i=14	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6	v6
i=15	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7	v7
i=16	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8
i=17	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9
i=18	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10
i=19	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11
i=20	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8	v8
i=21	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9	v9
i=22	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10	v10
i=23	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11	v11
i=24	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12
i=25	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13
i=26	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14
i=27	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15
i=28	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12	v12
i=29	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13	v13
i=30	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14	v14
i=31	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15	v15

C/D: 16 VGPRs (v0–v15); VGPR = $4 \cdot \lfloor i/8 \rfloor + (i \bmod 4)$, lane = $(32 \cdot \lfloor i/4 \rfloor) \bmod 64 + j$. Colour groups of 8 i-rows alternate teal/grey.

Fig. 26: 32×32, **K=16 accumulator layout.** Each cell shows the VGPR index that holds output element (i, j). Teal cells (rows where $\lfloor i/4 \rfloor$ is even) belong to lanes 0—31; grey cells to lanes 32—63. Column j gives the lane offset within the group.

(continued from previous page)

```
    __device__ static void zero(Accumulator &d) { d = {0.0f, 0.0f, 0.0f, 0.0f}; }

    __device__ static void load_a(const _Float16 *sA, int warp_m, int lane,
                                  int tk_comp, AFrag &a)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        int a_off = (warp_m * 16 + n) * tk_comp + g * 4;
        for (int i = 0; i < 4; ++i)
        {
            a[i] = sA[a_off + i];
        }
    }

    __device__ static void load_b(const _Float16 *sB, int warp_n, int lane,
                                  int cta_n, BFrag &b)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        int b_col = warp_n * 16 + n;
        for (int v = 0; v < 4; ++v)
        {
            int kk = g * 8 + v * 2;
            b[v * 2 + 0] = sB[kk * cta_n + b_col];
            b[v * 2 + 1] = sB[(kk + 1) * cta_n + b_col];
        }
    }

    __device__ static void mma(Accumulator &d, const AFrag &a, const BFrag &b)
    {
#if defined(__gfx908__) || defined(__gfx90a__) || defined(__gfx940__) || \
    defined(__gfx942__) || defined(__gfx950__)
        d = __builtin_amdgcn_smfmac_f32_16x16x32_f16(a, b, d, 0x88, 0, 0);
#endif
    }

    __device__ static void store_c(const Accumulator &d, float *D,
                                   int sub_m, int sub_n, int N, int lane)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        for (int r = 0; r < 4; ++r)
        {
            D[(sub_m + g * 4 + r) * N + sub_n + n] = d[r];
        }
    }
};
```

Each lane loads its $\boldsymbol{A}$ fragment from the compressed LDS tile, where the K dimension is halved. The $\boldsymbol{B}$ fragment is loaded from the full dense $\boldsymbol{B}$ tile in LDS. The builtin call replaces the entire inner-product loop of a baseline kernel with a single instruction. The 4-element FP32 result vector maps to four rows of the 16×16 output sub-tile, with the lane index selecting the column.

14.2.1.7.2 FP16 32×32 sparse kernel

This kernel uses `__builtin_amdgcn_smfmac_f32_32x32x16_f16` to compute a 32×32 sparse matrix multiply-accumulate with $K = 16$ per instruction. The Cooperative Thread Array (CTA) tile grows to 64×64 to accommodate the larger 32×32 wavefront tiles. The accumulator is a 16-element FP32 vector.

```
struct SmfmacCdna32x32F16Policy
{
    static constexpr int thread_tile_m = 32;
    static constexpr int thread_tile_n = 32;

    using v16float = float [[clang::ext_vector_type(16)]];
    using Accumulator = v16float;

    using v4half = _Float16 [[clang::ext_vector_type(4)]];
    using v8half = _Float16 [[clang::ext_vector_type(8)]];
    using AFrag = v4half;
    using BFrag = v8half;

    __device__ static void zero(Accumulator &d) { d = {}; }

    __device__ static void load_a(const _Float16 *sA, int warp_m, int lane,
                                  int tk_comp, AFrag &a)
    {
        const int g = lane / 16;
        int a_row = warp_m * 32 + (g % 2) * 16 + (lane % 16);
        int a_col = (g / 2) * 4;
        for (int i = 0; i < 4; ++i)
        {
            a[i] = sA[a_row * tk_comp + a_col + i];
        }
    }

    __device__ static void load_b(const _Float16 *sB, int warp_n, int lane,
                                  int cta_n, BFrag &b)
    {
        const int g = lane / 16;
        int b_col = warp_n * 32 + (g % 2) * 16 + (lane % 16);
        int k_off = (g / 2) * 8;
        for (int v = 0; v < 4; ++v)
        {
            int kk = k_off + v * 2;
            b[v * 2 + 0] = sB[kk * cta_n + b_col];
            b[v * 2 + 1] = sB[(kk + 1) * cta_n + b_col];
        }
    }

    __device__ static void mma(Accumulator &d, const AFrag &a, const BFrag &b)
    {
#if defined(__gfx908__) || defined(__gfx90a__) || defined(__gfx940__) || \
    defined(__gfx942__) || defined(__gfx950__)
        d = __builtin_amdgcn_smfmac_f32_32x32x16_f16(a, b, d, 0x88, 0, 0);
#endif
    }
```

(continues on next page)

(continued from previous page)

```
    __device__ static void store_c(const Accumulator &d, float *D,
                                   int sub_m, int sub_n, int N, int lane)
    {
        int j_base = lane % 32;
        int grp    = lane / 32;
        for (int r = 0; r < 16; ++r)
        {
            int i_out = (r / 4) * 8 + grp * 4 + (r % 4);
            D[(sub_m + i_out) * N + sub_n + j_base] = d[r];
        }
    }
};
```

The lane-to-element mapping for the 16-element accumulator differs from the 16×16 variant: each lane's 16 result elements are distributed across four groups of four consecutive rows, with the group stride determined by the lane's position within the wavefront.

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx908 \
    matrix_multiply_cdna_sparse_mfma.hip -o mm_cdna_sparse_mfma
./mm_cdna_sparse_mfma
```

> **Note**
>
> This example requires a CDNA or CDNA2 GPU (`gfx908` or `gfx90a`). Compile with `--offload-arch=gfx908` or `--offload-arch=gfx90a` to select the correct architecture.

14.2.1.8 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency. All SMFMAC instructions support vector ALU (VALU) co-execution; the VALU co-execution cycle count gives the number of VALU cycles available during the SMFMAC latency window.

Builtin	Ops	Cycle count	VALU co-execution cycles
`__builtin_amdgcn_smfmac_f32_16x16x32_f16`	16384	16	8
`__builtin_amdgcn_smfmac_f32_32x32x16_f16`	32768	32	24
`__builtin_amdgcn_smfmac_f32_16x16x32_bf16`	16384	16	8
`__builtin_amdgcn_smfmac_f32_32x32x16_bf16`	32768	32	24
`__builtin_amdgcn_smfmac_i32_16x16x64_i8`	32768	16	8
`__builtin_amdgcn_smfmac_i32_32x32x32_i8`	65536	32	24

14.2.1.9 Builtin reference

14.2.1.9.1 FP32-accumulate builtins

These builtins accumulate into single-precision (FP32) output fragments. They differ in the data type of the A and B matrix inputs.

14.2.1.9.1.1 FP16 matrix inputs

The following builtins accept compressed FP16 elements for $\boldsymbol{A}$ and dense FP16 elements for $\boldsymbol{B}$, accumulating into FP32 output fragments.

14.2.1.9.1.2 `__builtin_amdgcn_smfmac_f32_16x16x32_f16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x32_f16(
    v4half  srcA,
    v8half  srcB,
    v4float srcC,
    int     idx,
    int     cbsz,
    int     abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed FP16 elements of $\boldsymbol{A}$ and $K = 32$ dense FP16 elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane.

Parameter	Type	Description
`srcA`	v4half	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse).
`srcB`	v8half	Dense $\boldsymbol{B}$ elements per lane.
`srcC`	v4floa	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.1.9.1.3 `__builtin_amdgcn_smfmac_f32_32x32x16_f16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x16_f16(
    v4half   srcA,
    v8half   srcB,
    v16float srcC,
    int      idx,
    int      cbsz,
    int      abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 16$ compressed FP16 elements of $\boldsymbol{A}$ and $K = 16$ dense FP16 elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane.

Parameter	Type	Description
`srcA`	v4half	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse).
`srcB`	v8half	Dense $\boldsymbol{B}$ elements per lane.
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.1.9.1.4 BF16 matrix inputs

The following builtins accept compressed BF16 elements for $\boldsymbol{A}$ and dense BF16 elements for $\boldsymbol{B}$, accumulating into FP32 output fragments. BF16 values are stored as `short` in register.

14.2.1.9.1.5 `__builtin_amdgcn_smfmac_f32_16x16x32_bf16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x32_bf16(
    v4short  srcA,
    v8short  srcB,
    v4float  srcC,
    int      idx,
    int      cbsz,
    int      abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed BF16 elements of $\boldsymbol{A}$ and $K = 32$ dense BF16 elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane.

Parameter	Type	Description
`srcA`	v4shor	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, BF16 stored as `short`).
`srcB`	v8shor	Dense $\boldsymbol{B}$ elements per lane (BF16 stored as `short`).
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.1.9.1.6 `__builtin_amdgcn_smfmac_f32_32x32x16_bf16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x16_bf16(
    v4short   srcA,
    v8short   srcB,
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 16$ compressed BF16 elements of $\boldsymbol{A}$ and $K = 16$ dense BF16 elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane.

Parameter	Type	Description
`srcA`	v4short	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, BF16 stored as `short`).
`srcB`	v8short	Dense $\boldsymbol{B}$ elements per lane (BF16 stored as `short`).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.1.9.2 INT32-accumulate builtins

These builtins accept four signed 8-bit integer elements per 32-bit register lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into INT32 output fragments.

14.2.1.9.2.1 INT8 matrix inputs

The following builtins use INT8 matrix inputs.

14.2.1.9.2.2 `__builtin_amdgcn_smfmac_i32_16x16x64_i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_smfmac_i32_16x16x64_i8(
    v2int  srcA,
    v4int  srcB,
    v4int  srcC,
    int    idx,
    int    cbsz,
    int    abid);
```

Computes one step of a 16×16 sparse INT32 outer-product accumulation. Each instruction processes $K = 64$ compressed INT8 elements of $\boldsymbol{A}$ and $K = 64$ dense INT8 elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four INT8 elements.

Parameter	Type	Description
`srcA`	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four INT8 values packed per 32-bit lane).
`srcB`	v4int	Dense $\boldsymbol{B}$ elements per lane (four INT8 values packed per 32-bit lane).
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4int` – updated accumulator (srcA × srcB + srcC).

14.2.1.9.2.3 `__builtin_amdgcn_smfmac_i32_32x32x32_i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_smfmac_i32_32x32x32_i8(
    v2int   srcA,
    v4int   srcB,
    v16int  srcC,
    int     idx,
    int     cbsz,
    int     abid);
```

Computes one step of a 32×32 sparse INT32 outer-product accumulation. Each instruction processes $K = 32$ compressed INT8 elements of $\boldsymbol{A}$ and $K = 32$ dense INT8 elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four INT8 elements.

Parameter	Type	Description
`srcA`	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four INT8 values packed per 32-bit lane).
`srcB`	v4int	Dense $\boldsymbol{B}$ elements per lane (four INT8 values packed per 32-bit lane).
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16int` – updated accumulator (srcA × srcB + srcC).

14.2.2 CDNA3 sparse MFMA builtins

Sparse Matrix Fused Multiply-Accumulate (SMFMAC) builtins let you issue hardware matrix multiply-accumulate operations that exploit 4:2 structured sparsity directly from HIP device code on CDNA3 GPUs (`gfx942`, MI300 series). Each SMFMAC instruction multiplies a compressed $\boldsymbol{A}$ fragment by a dense $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single wavefront of 64 lanes. Because the $\boldsymbol{A}$ operand is stored in compressed form, these builtins halve the storage and memory bandwidth required for $\boldsymbol{A}$ compared to their dense MFMA counterparts, while the hardware uses a sparsity index to reconstruct the original element positions during the multiply.

CDNA3 adds SMFMAC builtins with FP8 (E4M3) and BF8 (E5M2) inputs in all four A×B type combinations.

14.2.2.1 Architecture availability

The builtins on this page target CDNA3 (`gfx942`, MI300 series). Equivalent builtins for other CDNA generations are documented on their own reference pages:

- *CDNA and CDNA2 sparse MFMA builtins* – CDNA (`gfx908`, MI100) and CDNA2 (`gfx90a`, MI200 series)
- *CDNA4 sparse MFMA builtins* – CDNA4 (`gfx950`, MI350 series)

> **Note**
>
> On CDNA3, all four operands ($\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$) can reside in either accumulation VGPRs (accVGPRs) or standard architecture VGPRs (archVGPRs). In HIP device code the compiler selects the appropriate register class automatically.

14.2.2.2 Naming convention

All SMFMAC builtins follow the pattern:

```
__builtin_amdgcn_smfmac_<out_type>_<M>x<N>x<K>_<in_type_a>[_<in_type_b>]
```

`out_type`
: Accumulator element type (`f32` or `i32`).

`M`, `N`, `K`
: Tile dimensions in elements. The instruction computes the contribution of a K-wide compressed panel of $\boldsymbol{A}$ and a K-wide dense panel of $\boldsymbol{B}$ to an $M \times N$ output tile. Each instruction processes one K step; the caller loops over K to accumulate a full matrix product.

`in_type_a`
: Input element type of the $\boldsymbol{A}$ matrix (`fp8` or `bf8`).

`in_type_b`
: Input element type of the $\boldsymbol{B}$ matrix. Always present for FP8 and BF8 variants to disambiguate the four possible A×B type combinations (`fp8_fp8`, `fp8_bf8`, `bf8_fp8`, `bf8_bf8`).

For example:

- `__builtin_amdgcn_smfmac_f32_16x16x64_fp8_bf8` – a 16×16 sparse MMA with $K = 64$ that multiplies FP8 $\boldsymbol{A}$ by BF8 $\boldsymbol{B}$ and accumulates into FP32.

14.2.2.3 Structured sparsity (4:2 pattern)

The SMFMAC instructions require the $\boldsymbol{A}$ matrix to obey 4:2 structured sparsity: in every contiguous group of four elements along the K dimension, exactly two are non-zero and the other two are zero. This constraint allows the $\boldsymbol{A}$ operand to be stored in a compressed representation that contains only the non-zero values, reducing the storage to half the original K dimension.

Along with the compressed non-zero values, the hardware requires a sparsity index that encodes which two of the four positions in each group hold the non-zeros. This index is passed as the `idx` argument to every SMFMAC builtin. The hardware uses it at execution time to align the compressed $\boldsymbol{A}$ elements against the correct rows of the $\boldsymbol{B}$ operand before accumulating the products.

Host-side compression is straightforward: walk the K dimension in groups of four, extract the two non-zero positions, and pack them consecutively into the output buffer. The example kernels in this topic adopt a fixed pattern that keeps positions 0 and 2 of every group, producing a compressed buffer of half the original K length.

14.2.2.4 Accumulator layout

Every SMFMAC instruction computes a single independent $M \times N$ output tile (block count = 1). The accumulator (***C*** / ***D***) layout across wavefront lanes and VGPRs is identical to the 1-block layout of the corresponding dense MFMA tile shape.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \leq i < M$
- j – zero-based column index within the tile, $0 \leq j < N$
- **lane** – wavefront lane that holds the element, $0 \leq \text{lane} < 64$
- **VGPR** – zero-based index into that lane's accumulator register file

14.2.2.4.1 16×16 layout

The 16×16 output tile occupies 4 VGPRs per lane (`v4float` or `v4int`).

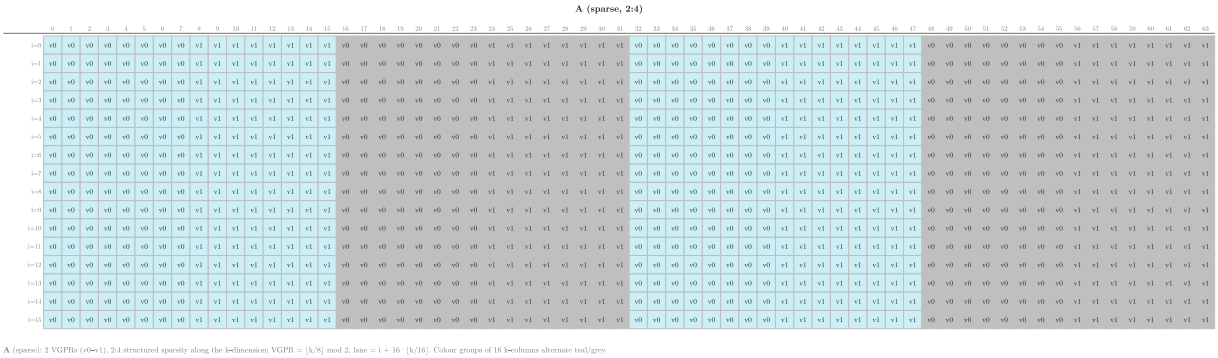


Fig. 27: 16×16, **K=64 A (sparse, 2:4) operand layout.**

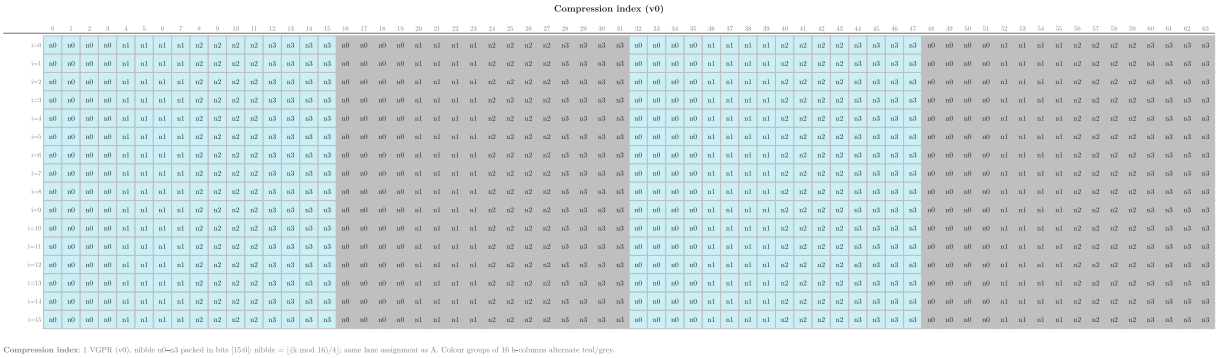
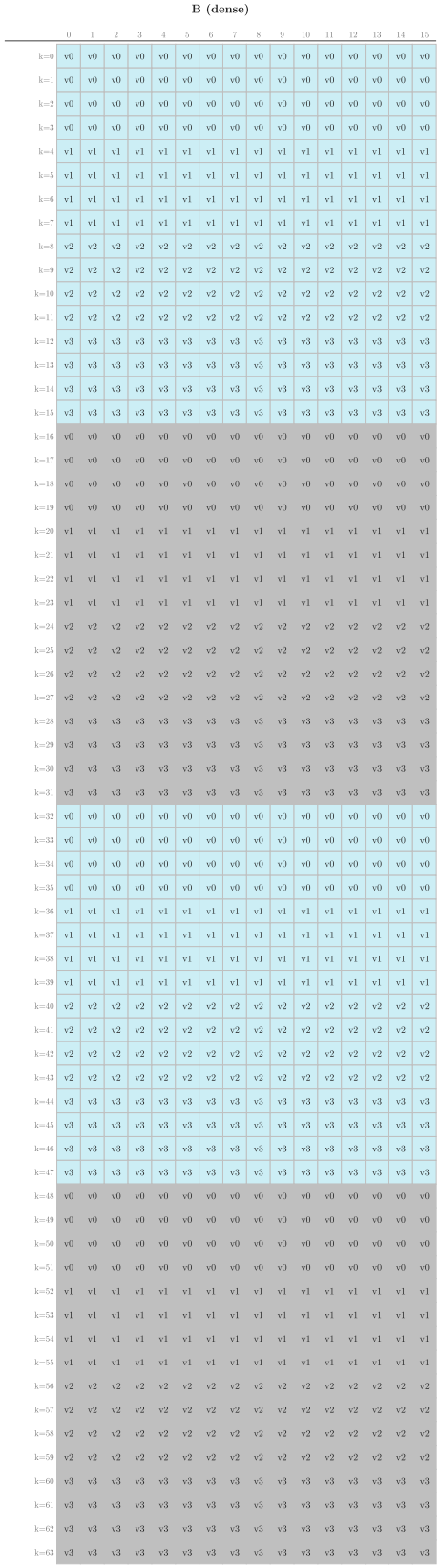


Fig. 28: 16×16, **K=64 compression index layout.**

Given output element (i, j):

$$\text{lane} = 16\lfloor \frac{i}{4} \rfloor + j$$

$$\text{VGPR} = i \bmod 4$$



B (dense): 4 VGPRs (v0–v3), one byte per k-step; VGPR = ⌊(k mod 16)/4⌋, lane = j + 16 · ⌊k/16⌋. Colour groups of 16 k-rows alternate teal/grey.

Fig. 29: 16 × 16, **K=64 B (dense) operand layout.**

C / D (input / output)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=4	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=5	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=6	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=7	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=8	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=9	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=10	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=11	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
i=12	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
i=13	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
i=14	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
i=15	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3

C/D: 4 VGPRs (v0–v3); VGPR = i mod 4, lane = $16 \cdot \lfloor i/4 \rfloor + j$. Colour groups of 4 i-rows alternate teal/grey.

Fig. 30: 16 × 16, **K=64 accumulator layout.** The output layout is identical to K=32; only the ***A*** and ***B*** input fragment sizes differ.

Conversely, given lane L and VGPR index G:

$$i = 4\lfloor\frac{L}{16}\rfloor + (G \bmod 4)$$
$$j = L \bmod 16$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0—3	0—15	0—3
4—7	16—31	0—3
8—11	32—47	0—3
12—15	48—63	0—3

14.2.2.4.2 32×32 layout

The 32×32 output tile occupies 16 VGPRs per lane (`v16float` or `v16int`).

Given output element (i, j):

$$\text{lane} = \left(32 \cdot \lfloor\frac{i}{4}\rfloor\right) \bmod 64 + j$$
$$\text{VGPR} = 4\lfloor\frac{i}{8}\rfloor + (i \bmod 4)$$

Conversely, given lane L and VGPR index G:

$$i = \left(8 \cdot \lfloor\frac{G}{4}\rfloor\right) \bmod 32 + 4\lfloor\frac{L}{32}\rfloor + (G \bmod 4)$$
$$j = L \bmod 32$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0—3	0—31	0—3
4—7	32—63	0—3
8—11	0—31	4—7
12—15	32—63	4—7
16—19	0—31	8—11
20—23	32—63	8—11
24—27	0—31	12—15
28—31	32—63	12—15

14.2.2.5 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v4float  = float    [[clang::ext_vector_type(4)]];
using v16float = float    [[clang::ext_vector_type(16)]];
using v2int    = int      [[clang::ext_vector_type(2)]];
using v4int    = int      [[clang::ext_vector_type(4)]];
using v8int    = int      [[clang::ext_vector_type(8)]];
using v16int   = int      [[clang::ext_vector_type(16)]];
```

A (sparse, 2:4)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=2	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=3	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=4	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=5	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=6	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=7	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=8	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=9	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=10	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=11	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=12	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=13	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=14	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=15	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=16	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=17	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=18	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=19	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=20	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=21	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=22	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=23	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=24	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=25	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=26	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=27	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=28	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=29	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=30	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1
i=31	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1	v0	v0	v0	v0	v0	v0	v0	v0	v1	v1	v1	v1	v1	v1	v1	v1

A (sparse): 2 VGPRs (v0–v1), 2:4 structured sparsity along the k-dimension; VGPR = $\lfloor k/8 \rfloor \bmod 2$, lane = $i + 32 \cdot \lfloor k/16 \rfloor$. Colour groups of 16 k-columns alternate teal/grey.

Fig. 31: 32×32, **K=32 A (sparse, 2:4) operand layout.**

Compression index (v0)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
i=0	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=1	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=2	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=4	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=5	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=6	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=7	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=8	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=9	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=10	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=11	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=12	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=13	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=14	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=15	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=16	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=17	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=18	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=19	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=20	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=21	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=22	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=23	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=24	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=25	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=26	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=27	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=28	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=29	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=30	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3
i=31	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3	n0	n0	n0	n0	n1	n1	n1	n1	n2	n2	n2	n2	n3	n3	n3	n3

Compression index: 1 VGPR (v0), nibble n0–n3 packed in bits [15:0]; nibble = ⌊(k mod 16)/4⌋; same lane assignment as A. Colour groups of 16 k-columns alternate teal/grey.

Fig. 32: 32 × 32, **K=32 compression index layout.**

B (dense)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
k=0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=1	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=2	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=3	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=4	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=5	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=6	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=7	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=8	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=9	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=10	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=11	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=12	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
k=13	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
k=14	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
k=15	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
k=16	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=17	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=18	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=19	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0	v0
k=20	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=21	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=22	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=23	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1	v1
k=24	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=25	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=26	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=27	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2	v2
k=28	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
k=29	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
k=30	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3
k=31	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3	v3

B (dense): 4 VGPRs (v0–v3), one byte per k-step; VGPR = $\lfloor k/4 \rfloor \bmod 4$, lane = $j + 32 \cdot \lfloor k/16 \rfloor$. Colour groups of 16 k-rows alternate teal/grey.

Fig. 33: 32×32, **K=32 B (dense) operand layout.**

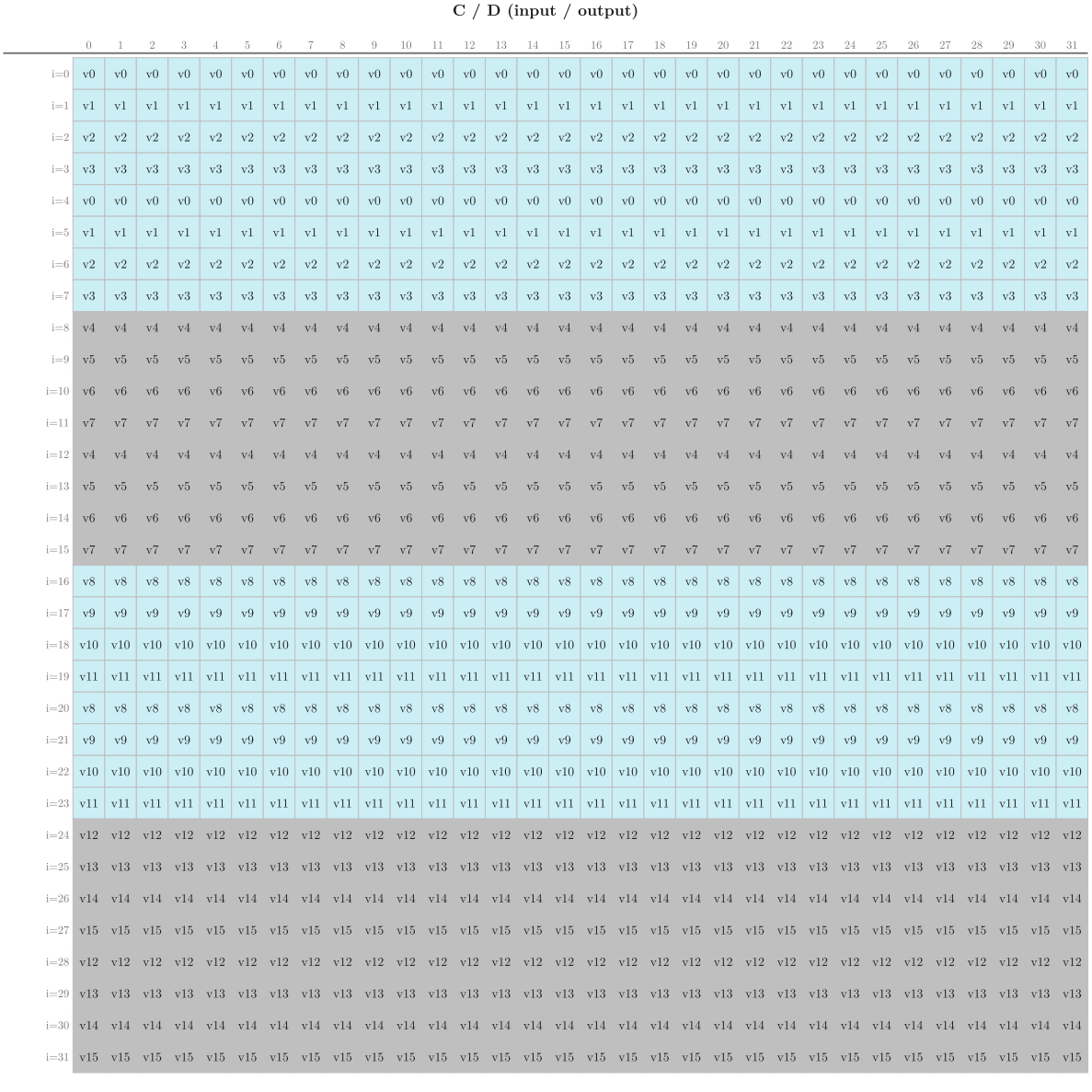


Fig. 34: 32 × 32, **K=32 accumulator layout.** The output layout is identical to K=16; only the ***A*** and ***B*** input fragment sizes differ.

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane; the total VGPR count equals the element count multiplied by the element size in 32-bit words.

14.2.2.6 Common parameters

The SMFMAC builtins do not use the `cbsz`, `abid`, or `blgp` modifiers from the dense MFMA family. See *Common MFMA parameters* for a description of those modifiers in the dense context.

14.2.2.7 Using sparse MFMA builtins as a compute policy

The following example kernel demonstrates the FP8 SMFMAC builtin in the context of a tiled matrix multiplication. The kernel loads tiles of the compressed $\boldsymbol{A}$ matrix and the dense $\boldsymbol{B}$ matrix into LDS, then calls the SMFMAC builtin to replace the inner-product loop of a conventional scalar kernel.

The complete source file is available for download:

- `matrix_multiply_cdna3_sparse_mfma.hip`

14.2.2.7.1 FP8 16×16 sparse kernel

This kernel uses `__builtin_amdgcn_smfmac_f32_16x16x64_fp8_fp8` to compute a 16×16 sparse matrix multiply-accumulate with $K = 64$ per instruction using FP8 (E4M3) inputs. Each 32-bit register lane packs four FP8 bytes; the kernel constructs the packed `v2int` and `v4int` operands from individual bytes loaded from LDS.

```
struct SmfmacCdna3FP8Policy
{
    static constexpr int thread_tile_m = 16;
    static constexpr int thread_tile_n = 16;

    using v4float = float [[clang::ext_vector_type(4)]];
    using Accumulator = v4float;

    using v2int = int [[clang::ext_vector_type(2)]];
    using v4int = int [[clang::ext_vector_type(4)]];
    using AFrag = v2int;
    using BFrag = v4int;

    __device__ static void zero(Accumulator &d) { d = {0.0f, 0.0f, 0.0f, 0.0f}; }

    __device__ static void load_a(const uint8_t *sA, int warp_m, int lane,
                                  int tk_comp, AFrag &a)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        int a_off = (warp_m * 16 + n) * tk_comp + g * 8;
        for (int j = 0; j < 2; ++j)
        {
            a[j] = static_cast<int>(
                uint32_t(sA[a_off + j*4 + 0])       | (uint32_t(sA[a_off + j*4 + 1]) <
↪< 8) |
                (uint32_t(sA[a_off + j*4 + 2]) << 16) | (uint32_t(sA[a_off + j*4 +␣
↪3]) << 24));
        }
    }
```

(continues on next page)

(continued from previous page)

```

    __device__ static void load_b(const uint8_t *sB, int warp_n, int lane,
                                  int cta_n, BFrag &b)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        int b_col = warp_n * 16 + n;
        for (int v = 0; v < 4; ++v)
        {
            int kk = g * 16 + v * 4;
            b[v] = static_cast<int>(
                uint32_t(sB[kk * cta_n + b_col])       | (uint32_t(sB[(kk+1) * cta_n + b_col]) << 8) |
                (uint32_t(sB[(kk+2) * cta_n + b_col]) << 16) | (uint32_t(sB[(kk+3) * cta_n + b_col]) << 24));
        }
    }

    __device__ static void mma(Accumulator &d, const AFrag &a, const BFrag &b)
    {
#if defined(__gfx942__) || defined(__gfx950__)
        d = __builtin_amdgcn_smfmac_f32_16x16x64_fp8_fp8(a, b, d, 0x8888, 0, 0);
#endif
    }

    __device__ static void store_c(const Accumulator &d, float *D,
                                   int sub_m, int sub_n, int N, int lane)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        for (int r = 0; r < 4; ++r)
        {
            D[(sub_m + g * 4 + r) * N + sub_n + n] = d[r];
        }
    }
};
```

The kernel packs four consecutive FP8 bytes into each 32-bit lane using shifts and bitwise OR before passing the vectors to the builtin. The sparsity index is wider than the FP16 variant (`0x8888` vs `0x88`) because the larger K dimension requires more index bits to cover all element groups.

> **Note**
>
> The FP8 encoding used by `gfx942` is FNUZ (Finite, No Unsigned Zero), while later architectures use the standard OCP (Open Compute Project) E4M3 encoding. The example code selects the correct interpretation at runtime based on the device architecture.

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx942 \
    matrix_multiply_cdna3_sparse_mfma.hip -o mm_cdna3_sparse_mfma
./mm_cdna3_sparse_mfma
```

> **Note**
>
> This example requires a CDNA3 GPU (`gfx942`). Compile with `--offload-arch=gfx942` to select the correct architecture.

14.2.2.8 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency. All SMFMAC instructions support vector ALU (VALU) co-execution; the VALU co-execution cycle count gives the number of VALU cycles available during the SMFMAC latency window.

Builtin	Ops	Cycle count	VALU co-execution cycles
`__builtin_amdgcn_smfmac_f32_16x16x64_fp8_fp8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_16x16x64_fp8_bf8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_16x16x64_bf8_fp8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_16x16x64_bf8_bf8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_32x32x32_fp8_fp8`	65536	32	24
`__builtin_amdgcn_smfmac_f32_32x32x32_fp8_bf8`	65536	32	24
`__builtin_amdgcn_smfmac_f32_32x32x32_bf8_fp8`	65536	32	24
`__builtin_amdgcn_smfmac_f32_32x32x32_bf8_bf8`	65536	32	24

14.2.2.9 Builtin reference

14.2.2.9.1 FP32-accumulate builtins

These builtins accumulate into single-precision (FP32) output fragments.

14.2.2.9.1.1 FP8 and BF8 matrix inputs

The following builtins accept compressed FP8 (E4M3) or BF8 (E5M2) elements for $\boldsymbol{A}$ and dense FP8 or BF8 elements for $\boldsymbol{B}$, accumulating into FP32 output fragments. All four combinations of FP8 and BF8 input types are supported for each tile size.

14.2.2.9.1.2 `__builtin_amdgcn_smfmac_f32_16x16x64_fp8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x64_fp8_fp8(
    v2int    srcA,
    v4int    srcB,
    v4float  srcC,
    int      idx,
    int      cbsz,
    int      abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 64$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
srcA	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
srcB	v4int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
srcC	v4floa	Accumulator input: 4 FP32 elements per lane.
idx	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
cbsz	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.2.9.1.3 `__builtin_amdgcn_smfmac_f32_16x16x64_fp8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x64_fp8_bf8(
    v2int    srcA,
    v4int    srcB,
    v4float  srcC,
    int      idx,
    int      cbsz,
    int      abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 64$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
srcA	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
srcB	v4int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
srcC	v4floa	Accumulator input: 4 FP32 elements per lane.
idx	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
cbsz	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.2.9.1.4 `__builtin_amdgcn_smfmac_f32_16x16x64_bf8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x64_bf8_fp8(
    v2int    srcA,
    v4int    srcB,
    v4float  srcC,
```

(continues on next page)

(continued from previous page)

```
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 64$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
`srcA`	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
`srcB`	v4int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
`srcC`	v4floa	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.2.9.1.5 `__builtin_amdgcn_smfmac_f32_16x16x64_bf8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x64_bf8_bf8(
    v2int     srcA,
    v4int     srcB,
    v4float   srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 64$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
`srcA`	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
`srcB`	v4int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
`srcC`	v4floa	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.2.9.1.6 `__builtin_amdgcn_smfmac_f32_32x32x32_fp8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x32_fp8_fp8(
    v2int     srcA,
    v4int     srcB,
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 32$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
`srcA`	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
`srcB`	v4int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.2.9.1.7 `__builtin_amdgcn_smfmac_f32_32x32x32_fp8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x32_fp8_bf8(
    v2int     srcA,
    v4int     srcB,
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 32$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
srcA	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
srcB	v4int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
srcC	v16floa	Accumulator input: 16 FP32 elements per lane.
idx	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
cbsz	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.2.9.1.8 `__builtin_amdgcn_smfmac_f32_32x32x32_bf8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x32_bf8_fp8(
    v2int     srcA,
    v4int     srcB,
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 32$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
srcA	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
srcB	v4int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
srcC	v16floa	Accumulator input: 16 FP32 elements per lane.
idx	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
cbsz	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
abid	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.2.9.1.9 `__builtin_amdgcn_smfmac_f32_32x32x32_bf8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x32_bf8_bf8(
    v2int     srcA,
    v4int     srcB,
    v16float  srcC,
```

(continues on next page)

(continued from previous page)

```
    int         idx,
    int         cbsz,
    int         abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 32$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements.

Parameter	Type	Description
`srcA`	v2int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
`srcB`	v4int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB $+$ srcC).

14.2.3 CDNA4 sparse MFMA builtins

Sparse Matrix Fused Multiply-Accumulate (SMFMAC) builtins let you issue hardware matrix multiply-accumulate operations that exploit 4:2 structured sparsity directly from HIP device code on CDNA4 GPUs (`gfx950`, MI350 series). Each SMFMAC instruction multiplies a compressed $\boldsymbol{A}$ fragment by a dense $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single wavefront of 64 lanes. Because the $\boldsymbol{A}$ operand is stored in compressed form, these builtins halve the storage and memory bandwidth required for $\boldsymbol{A}$ compared to their dense MFMA counterparts, while the hardware uses a sparsity index to reconstruct the original element positions during the multiply.

CDNA4 adds doubled-K SMFMAC variants in all data types (FP16, BF16, INT8, FP8, and BF8), doubling the K depth processed per instruction compared to earlier generations.

14.2.3.1 Architecture availability

The builtins on this page target CDNA4 (gfx950 architecture also known as AMD Instinct MI350 series) exclusively. Equivalent builtins for other CDNA generations are documented on their own reference pages:

- *CDNA and CDNA2 sparse MFMA builtins* – CDNA (`gfx908`, MI100) and CDNA2 (`gfx90a`, MI200 series)
- *CDNA3 sparse MFMA builtins* – CDNA3 (`gfx942`, MI300 series)

14.2.3.2 Naming convention

All SMFMAC builtins follow the pattern:

```
__builtin_amdgcn_smfmac_<out_type>_<M>x<N>x<K>_<in_type_a>[_<in_type_b>]
```

`out_type`
: Accumulator element type (`f32` or `i32`).

`M`, `N`, `K`
: Tile dimensions in elements. The instruction computes the contribution of a K-wide compressed panel of $\boldsymbol{A}$ and a

K-wide dense panel of $\boldsymbol{B}$ to an $M \times N$ output tile. Each instruction processes one K step; the caller loops over K to accumulate a full matrix product.

in_type_a
Input element type of the $\boldsymbol{A}$ matrix (`f16`, `bf16`, `i8`, `fp8`, or `bf8`).

in_type_b (FP8/BF8 variants only)
Input element type of the $\boldsymbol{B}$ matrix. Always present for FP8 and BF8 variants to disambiguate the four possible A×B type combinations (`fp8_fp8`, `fp8_bf8`, `bf8_fp8`, `bf8_bf8`). Omitted for FP16, BF16, and INT8 variants where $\boldsymbol{A}$ and $\boldsymbol{B}$ always share the same type.

For example:

- `__builtin_amdgcn_smfmac_f32_16x16x128_fp8_fp8` – a 16×16 sparse MMA with $K = 128$ that multiplies FP8 $\boldsymbol{A}$ by FP8 $\boldsymbol{B}$ and accumulates into FP32.

14.2.3.3 Structured sparsity (4:2 pattern)

The SMFMAC instructions require the $\boldsymbol{A}$ matrix to obey 4:2 structured sparsity: in every contiguous group of four elements along the K dimension, exactly two are non-zero and the other two are zero. This constraint allows the $\boldsymbol{A}$ operand to be stored in a compressed representation that contains only the non-zero values, reducing the storage to half the original K dimension.

Along with the compressed non-zero values, the hardware requires a sparsity index that encodes which two of the four positions in each group hold the non-zeros. This index is passed as the `idx` argument to every SMFMAC builtin. The hardware uses it at execution time to align the compressed $\boldsymbol{A}$ elements against the correct rows of the $\boldsymbol{B}$ operand before accumulating the products.

Host-side compression is straightforward: walk the K dimension in groups of four, extract the two non-zero positions, and pack them consecutively into the output buffer. The example kernels in this topic adopt a fixed pattern that keeps positions 0 and 2 of every group, producing a compressed buffer of half the original K length.

14.2.3.4 Accumulator layout

Every SMFMAC instruction computes a single independent $M \times N$ output tile (block count = 1). The accumulator ($\boldsymbol{C}$ / $\boldsymbol{D}$) layout across wavefront lanes and VGPRs is identical to the 1-block layout of the corresponding dense MFMA tile shape.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \leq i < M$
- j – zero-based column index within the tile, $0 \leq j < N$
- **lane** – wavefront lane that holds the element, $0 \leq \text{lane} < 64$
- **VGPR** – zero-based index into that lane's accumulator register file

14.2.3.4.1 16×16 layout

The 16×16 output tile occupies 4 VGPRs per lane (`v4float` or `v4int`).

Given output element (i, j):

$$\text{lane} = 16\lfloor\frac{i}{4}\rfloor + j$$
$$\text{VGPR} = i \bmod 4$$

Conversely, given lane L and VGPR index G:

$$i = 4\lfloor\frac{L}{16}\rfloor + (G \bmod 4)$$
$$j = L \bmod 16$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0—3	0—15	0—3
4—7	16—31	0—3
8—11	32—47	0—3
12—15	48—63	0—3

14.2.3.4.2 32×32 layout

The 32×32 output tile occupies 16 VGPRs per lane (`v16float` or `v16int`).

Given output element (i, j):

$$\text{lane} = \left(32 \cdot \lfloor\frac{i}{4}\rfloor\right) \bmod 64 + j$$
$$\text{VGPR} = 4\lfloor\frac{i}{8}\rfloor + (i \bmod 4)$$

Conversely, given lane L and VGPR index G:

$$i = \left(8 \cdot \lfloor\frac{G}{4}\rfloor\right) \bmod 32 + 4\lfloor\frac{L}{32}\rfloor + (G \bmod 4)$$
$$j = L \bmod 32$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0—3	0—31	0—3
4—7	32—63	0—3
8—11	0—31	4—7
12—15	32—63	4—7
16—19	0—31	8—11
20—23	32—63	8—11
24—27	0—31	12—15
28—31	32—63	12—15

14.2.3.5 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v4float  = float    [[clang::ext_vector_type(4)]];
using v16float = float    [[clang::ext_vector_type(16)]];
using v4half   = _Float16 [[clang::ext_vector_type(4)]];
using v8half   = _Float16 [[clang::ext_vector_type(8)]];
using v16half  = _Float16 [[clang::ext_vector_type(16)]];
```

(continues on next page)

(continued from previous page)

```
using v8bf16   = __bf16   [[clang::ext_vector_type(8)]];
using v16bf16  = __bf16   [[clang::ext_vector_type(16)]];
using v2int    = int      [[clang::ext_vector_type(2)]];
using v4int    = int      [[clang::ext_vector_type(4)]];
using v8int    = int      [[clang::ext_vector_type(8)]];
using v16int   = int      [[clang::ext_vector_type(16)]];
```

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane; the total VGPR count equals the element count multiplied by the element size in 32-bit words.

14.2.3.6 Common parameters

The SMFMAC builtins do not use the `cbsz`, `abid`, or `blgp` modifiers from the dense MFMA family. See *Common MFMA parameters* for a description of those modifiers in the dense context.

14.2.3.7 Using sparse MFMA builtins as a compute policy

The following example kernel demonstrates the doubled-K FP8 SMFMAC builtin in the context of a tiled matrix multiplication. The kernel loads tiles of the compressed $\boldsymbol{A}$ matrix and the dense $\boldsymbol{B}$ matrix into LDS, then calls the SMFMAC builtin to replace the inner-product loop of a conventional scalar kernel.

The complete source file is available for download:

- `matrix_multiply_cdna4_sparse_mfma.hip`

14.2.3.7.1 FP8 16×16 sparse kernel (K=128)

This kernel uses `__builtin_amdgcn_smfmac_f32_16x16x128_fp8_fp8` to compute a 16×16 sparse matrix multiply-accumulate with $K = 128$ per instruction, doubling the K depth of the `gfx942` variant. The compressed $\boldsymbol{A}$ operand grows to a `v4int` vector and the dense $\boldsymbol{B}$ to a `v8int` vector.

```
struct SmfmacCdna4FP8Policy
{
    static constexpr int thread_tile_m = 16;
    static constexpr int thread_tile_n = 16;

    using v4float = float [[clang::ext_vector_type(4)]];
    using Accumulator = v4float;

    using v4int = int [[clang::ext_vector_type(4)]];
    using v8int = int [[clang::ext_vector_type(8)]];
    using AFrag = v4int;
    using BFrag = v8int;

    __device__ static void zero(Accumulator &d) { d = {0.0f, 0.0f, 0.0f, 0.0f}; }

    __device__ static void load_a(const uint8_t *sA, int warp_m, int lane,
                                  int tk_comp, AFrag &a)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        int a_off = (warp_m * 16 + n) * tk_comp + g * 16;
        for (int j = 0; j < 4; ++j)
```

(continues on next page)

(continued from previous page)

```
        {
            a[j] = static_cast<int>(
                uint32_t(sA[a_off + j*4 + 0])       | (uint32_t(sA[a_off + j*4 + 1]) <
↪< 8) |
                (uint32_t(sA[a_off + j*4 + 2]) << 16) | (uint32_t(sA[a_off + j*4 +␣
↪3]) << 24));
        }
    }

    __device__ static void load_b(const uint8_t *sB, int warp_n, int lane,
                                  int cta_n, BFrag &b)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        int b_col = warp_n * 16 + n;
        for (int v = 0; v < 8; ++v)
        {
            int kk = (v / 4) * 64 + g * 16 + (v % 4) * 4;
            b[v] = static_cast<int>(
                uint32_t(sB[kk * cta_n + b_col])       | (uint32_t(sB[(kk+1) * cta_n␣
↪+ b_col]) << 8) |
                (uint32_t(sB[(kk+2) * cta_n + b_col]) << 16) | (uint32_t(sB[(kk+3) *␣
↪cta_n + b_col]) << 24));
        }
    }

    __device__ static void mma(Accumulator &d, const AFrag &a, const BFrag &b)
    {
#if defined(__gfx950__)
        d = __builtin_amdgcn_smfmac_f32_16x16x128_fp8_fp8(
                a, b, d, static_cast<int>(0x88888888u), 0, 0);
#endif
    }

    __device__ static void store_c(const Accumulator &d, float *D,
                                   int sub_m, int sub_n, int N, int lane)
    {
        const int g = lane / 16;
        const int n = lane % 16;
        for (int r = 0; r < 4; ++r)
        {
            D[(sub_m + g * 4 + r) * N + sub_n + n] = d[r];
        }
    }
};
```

The byte-packing pattern is the same as the K=64 variant, but each lane now loads twice as many bytes. The sparsity index expands to a full 32-bit value (`0x88888888`) to cover the 128-element K dimension.

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx950 \
    matrix_multiply_cdna4_sparse_mfma.hip -o mm_cdna4_sparse_mfma
```

(continues on next page)

(continued from previous page)

```
./mm_cdna4_sparse_mfma
```

> **Note**
>
> This example requires a CDNA4 GPU (`gfx950`). Compile with `--offload-arch=gfx950` to select the correct architecture.

14.2.3.8 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency. All SMFMAC instructions support vector ALU (VALU) co-execution; the VALU co-execution cycle count gives the number of VALU cycles available during the SMFMAC latency window.

Builtin	Ops	Cycle count	VALU co-execution cycles
`__builtin_amdgcn_smfmac_f32_16x16x64_f16`	16384	16	8
`__builtin_amdgcn_smfmac_f32_32x32x32_f16`	32768	32	24
`__builtin_amdgcn_smfmac_f32_16x16x64_bf16`	16384	16	8
`__builtin_amdgcn_smfmac_f32_32x32x32_bf16`	32768	32	24
`__builtin_amdgcn_smfmac_i32_16x16x128_i8`	32768	16	8
`__builtin_amdgcn_smfmac_i32_32x32x64_i8`	65536	32	24
`__builtin_amdgcn_smfmac_f32_16x16x128_fp8_fp8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_16x16x128_fp8_bf8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_16x16x128_bf8_fp8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_16x16x128_bf8_bf8`	32768	16	8
`__builtin_amdgcn_smfmac_f32_32x32x64_fp8_fp8`	65536	32	24
`__builtin_amdgcn_smfmac_f32_32x32x64_fp8_bf8`	65536	32	24
`__builtin_amdgcn_smfmac_f32_32x32x64_bf8_fp8`	65536	32	24
`__builtin_amdgcn_smfmac_f32_32x32x64_bf8_bf8`	65536	32	24

14.2.3.9 Builtin reference

14.2.3.9.1 FP32-accumulate builtins

These builtins accumulate into single-precision (FP32) output fragments. They differ in the data type of the $\boldsymbol{A}$ and $\boldsymbol{B}$ matrix inputs.

14.2.3.9.1.1 FP16 matrix inputs

The following doubled-K FP16 variants are available on `gfx950` only.

14.2.3.9.1.2 `__builtin_amdgcn_smfmac_f32_16x16x64_f16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x64_f16(
    v8half  srcA,
    v16half srcB,
    v4float srcC,
    int     idx,
```

(continues on next page)

(continued from previous page)

```
    int     cbsz,
    int     abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed FP16 elements of $\boldsymbol{A}$ and $K = 64$ dense FP16 elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v8half	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse).
`srcB`	v16hal	Dense $\boldsymbol{B}$ elements per lane.
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.3 `__builtin_amdgcn_smfmac_f32_32x32x32_f16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x32_f16(
    v8half   srcA,
    v16half  srcB,
    v16float srcC,
    int      idx,
    int      cbsz,
    int      abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed FP16 elements of $\boldsymbol{A}$ and $K = 32$ dense FP16 elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v8half	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse).
`srcB`	v16hal	Dense $\boldsymbol{B}$ elements per lane.
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.4 BF16 matrix inputs

The following doubled-K BF16 variants are available on `gfx950` only. These use native `__bf16` register types rather than `short` storage.

14.2.3.9.1.5 `__builtin_amdgcn_smfmac_f32_16x16x64_bf16`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x64_bf16(
    v8bf16  srcA,
    v16bf16 srcB,
    v4float srcC,
    int     idx,
    int     cbsz,
    int     abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed BF16 elements of $\boldsymbol{A}$ and $K = 64$ dense BF16 elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v8bf16	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, BF16 stored as `__bf16`).
`srcB`	v16bf1	Dense $\boldsymbol{B}$ elements per lane (BF16 stored as `__bf16`).
`srcC`	v4float	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.6 `__builtin_amdgcn_smfmac_f32_32x32x32_bf16`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x32_bf16(
    v8bf16   srcA,
    v16bf16  srcB,
    v16float srcC,
    int      idx,
    int      cbsz,
    int      abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 32$ compressed BF16 elements of $\boldsymbol{A}$ and $K = 32$ dense BF16 elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v8bf16	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, BF16 stored as `__bf16`).
`srcB`	v16bf1	Dense $\boldsymbol{B}$ elements per lane (BF16 stored as `__bf16`).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.7 FP8 and BF8 matrix inputs

The following doubled-K FP8 and BF8 variants are available on `gfx950` only.

14.2.3.9.1.8 `__builtin_amdgcn_smfmac_f32_16x16x128_fp8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x128_fp8_fp8(
    v4int     srcA,
    v8int     srcB,
    v4float   srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 128$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 128$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
`srcC`	v4floa	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.9 `__builtin_amdgcn_smfmac_f32_16x16x128_fp8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x128_fp8_bf8(
    v4int     srcA,
    v8int     srcB,
    v4float   srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 128$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 128$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
`srcC`	v4floa	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.10 `__builtin_amdgcn_smfmac_f32_16x16x128_bf8_fp8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x128_bf8_fp8(
    v4int     srcA,
    v8int     srcB,
    v4float   srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 128$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 128$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
`srcC`	v4floa	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.11 `__builtin_amdgcn_smfmac_f32_16x16x128_bf8_bf8`

Signature and parameters for this builtin.

```
v4float __builtin_amdgcn_smfmac_f32_16x16x128_bf8_bf8(
    v4int     srcA,
    v8int     srcB,
    v4float   srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 16×16 sparse FP32 outer-product accumulation. Each instruction processes $K = 128$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 128$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 16×16 FP32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
`srcC`	v4floa	Accumulator input: 4 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.1.12 `__builtin_amdgcn_smfmac_f32_32x32x64_fp8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x64_fp8_fp8(
    v4int     srcA,
    v8int     srcB,
```

(continues on next page)

(continued from previous page)

```
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 64$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB $+$ srcC).

14.2.3.9.1.13 `__builtin_amdgcn_smfmac_f32_32x32x64_fp8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x64_fp8_bf8(
    v4int     srcA,
    v8int     srcB,
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed FP8 (E4M3) elements of $\boldsymbol{A}$ and $K = 64$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four FP8 (E4M3) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA × srcB + srcC).

14.2.3.9.1.14 `__builtin_amdgcn_smfmac_f32_32x32x64_bf8_fp8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x64_bf8_fp8(
    v4int     srcA,
    v8int     srcB,
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 64$ dense FP8 (E4M3) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four FP8 (E4M3) values packed per 32-bit lane).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA × srcB + srcC).

14.2.3.9.1.15 `__builtin_amdgcn_smfmac_f32_32x32x64_bf8_bf8`

Signature and parameters for this builtin.

```
v16float __builtin_amdgcn_smfmac_f32_32x32x64_bf8_bf8(
    v4int     srcA,
    v8int     srcB,
    v16float  srcC,
    int       idx,
    int       cbsz,
    int       abid);
```

Computes one step of a 32×32 sparse FP32 outer-product accumulation. Each instruction processes $K = 64$ compressed BF8 (E5M2) elements of $\boldsymbol{A}$ and $K = 64$ dense BF8 (E5M2) elements of $\boldsymbol{B}$, accumulating into a 32×32 FP32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four 8-bit elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four BF8 (E5M2) values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four BF8 (E5M2) values packed per 32-bit lane).
`srcC`	v16floa	Accumulator input: 16 FP32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16float` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.2 INT32-accumulate builtins

These builtins accept four signed 8-bit integer elements per 32-bit register lane for both $\boldsymbol{A}$ and $\boldsymbol{B}$ inputs, and accumulate into INT32 output fragments.

14.2.3.9.2.1 INT8 matrix inputs

The following doubled-K INT8 variants are available on `gfx950` only.

14.2.3.9.2.2 `__builtin_amdgcn_smfmac_i32_16x16x128_i8`

Signature and parameters for this builtin.

```
v4int __builtin_amdgcn_smfmac_i32_16x16x128_i8(
    v4int  srcA,
    v8int  srcB,
    v4int  srcC,
    int    idx,
    int    cbsz,
    int    abid);
```

Computes one step of a 16×16 sparse INT32 outer-product accumulation. Each instruction processes $K = 128$ compressed INT8 elements of $\boldsymbol{A}$ and $K = 128$ dense INT8 elements of $\boldsymbol{B}$, accumulating into a 16×16 INT32 tile held across 4 accVGPRs per lane. Each 32-bit register lane packs four INT8 elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four INT8 values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four INT8 values packed per 32-bit lane).
`srcC`	v4int	Accumulator input: 4 INT32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v4int` – updated accumulator (srcA $\times$ srcB + srcC).

14.2.3.9.2.3 `__builtin_amdgcn_smfmac_i32_32x32x64_i8`

Signature and parameters for this builtin.

```
v16int __builtin_amdgcn_smfmac_i32_32x32x64_i8(
    v4int   srcA,
    v8int   srcB,
    v16int  srcC,
    int     idx,
    int     cbsz,
    int     abid);
```

Computes one step of a 32×32 sparse INT32 outer-product accumulation. Each instruction processes $K = 64$ compressed INT8 elements of $\boldsymbol{A}$ and $K = 64$ dense INT8 elements of $\boldsymbol{B}$, accumulating into a 32×32 INT32 tile held across 16 accVGPRs per lane. Each 32-bit register lane packs four INT8 elements. This variant is available on `gfx950` only.

Parameter	Type	Description
`srcA`	v4int	Compressed $\boldsymbol{A}$ elements per lane (4:2 sparse, four INT8 values packed per 32-bit lane).
`srcB`	v8int	Dense $\boldsymbol{B}$ elements per lane (four INT8 values packed per 32-bit lane).
`srcC`	v16int	Accumulator input: 16 INT32 elements per lane.
`idx`	int	Sparsity index encoding the positions of non-zero elements within each group of four along the K dimension, see *Sparse MFMA parameters*.
`cbsz`	int	Control Broadcast Size modifier, see *Sparse MFMA parameters*.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier, see *Sparse MFMA parameters*.

Returns `v16int` – updated accumulator (srcA $\times$ srcB + srcC).

14.3 CDNA4 MFMA transpose load builtins

The CDNA4 (`gfx950`, MI350 series) architecture introduces a set of LDS (Local Data Share) load builtins that perform a hardware-assisted transpose as part of the load. These builtins let you store matrix $\boldsymbol{A}$ tiles in LDS in column-major order or matrix $\boldsymbol{B}$ tiles in row-major order, then load them directly into the per-lane fragment layout expected by the *CDNA4 dense MFMA builtins*, without a software shuffle step.

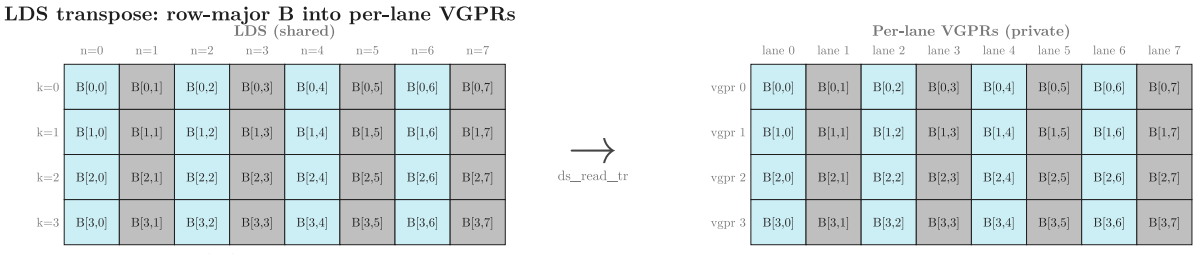


Fig. 35: `ds_read_tr` loads B from LDS in row-major order and distributes each column to the corresponding lane's VGPRs. The simplified example uses $K = 4$ and $N = 8$; colors identify each N-column.

14.4 Architecture availability

The builtins on this page target CDNA4 (`gfx950`, MI350 series) exclusively. For the MFMA compute builtins they are designed to feed, see *CDNA4 dense MFMA builtins*.

14.5 Naming convention

All transpose load builtins follow the pattern:

```
__builtin_amdgcn_ds_read_tr<N>_b<M>_v<K><type>
```

N
: Transpose group width: the number of elements exchanged between adjacent lanes during the transpose operation.

M
: Bits loaded per lane from LDS.

K
: Number of elements in the return vector.

type
: Element type of the return vector: `i32`, `i16`, `f16`, or `bf16`.

14.6 Register types used in this reference

The signatures below use the following type aliases, which you can declare with Clang vector attributes in any HIP translation unit:

```
using v2int    = int   [[clang::ext_vector_type(2)]];     // FP4 / FP8 / BF8
using v3int    = int   [[clang::ext_vector_type(3)]];     // FP6
using v4short  = short [[clang::ext_vector_type(4)]];     // INT8 (2 × INT8 per short)
using v4half   = _Float16 [[clang::ext_vector_type(4)]];  // FP16
using v4bfloat = short [[clang::ext_vector_type(4)]];     // BF16 (1 × BF16 per short)
```

14.7 Builtin reference

The following builtins are available on CDNA4. Each takes a single pointer argument in LDS address space (`__shared__`) and returns a vector holding the lane's share of the transposed tile. All builtins are `const` and have no side effects.

14.7.1 Hardware constraints

The following constraints apply to every builtin in this family:

- **Pair of calls required.** One call loads only half of the matrix tile's K positions. A second call with a different `ptr` value and a different destination register pair completes the tile. The K-position split varies by element size and is summarized in the table below.
- **LDS address alignment.** The `ptr` address must be aligned to the load data size: 8 bytes for `b64` loads and 12 bytes for `b96` loads.
- **VGPR alignment.** All `b64` loads write to an even-aligned VGPR pair. The `b96` load (`ds_read_tr6_b96_v3i32`) does not require even-VGPR alignment.

Table 3: K-position split by builtin

Builtin	Call 1 K positions	Call 2 K positions
`ds_read_tr4_b64_v2i32`	0—15, 32—47	16—31, 48—63
`ds_read_tr6_b96_v3i32`	0—15, 32—47	16—31, 48—63
`ds_read_tr8_b64_v2i32`	0—7, 16—23, 32—39, 48—55	8—15, 24—31, 40—47, 56—63
`ds_read_tr16_b64_v4i16`	0—3, 8—11	4—7, 12—15
`ds_read_tr16_b64_v4f16`	0—3, 8—11	4—7, 12—15
`ds_read_tr16_b64_v4bf16`	0—3, 8—11	4—7, 12—15

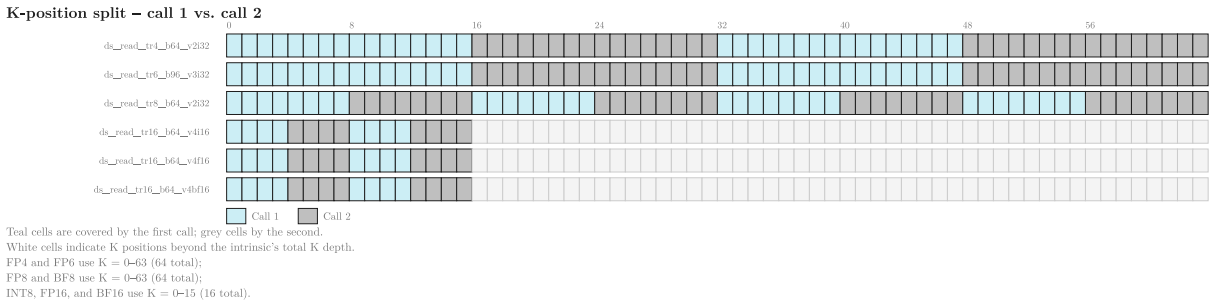


Fig. 36: K-position split for each `ds_read_tr` builtin. Teal cells are covered by the first call; grey cells by the second. White cells indicate K positions beyond the builtin's total K depth.

14.7.2 Sub-byte operand loads

These builtins load and transpose operands for the scaled sub-byte MFMA family (`__builtin_amdgcn_mfma_scale_f32_*`).

14.7.2.1 `__builtin_amdgcn_ds_read_tr4_b64_v2i32`

```
v2int __builtin_amdgcn_ds_read_tr4_b64_v2i32(__shared__ v2int* ptr);
```

Loads 64 bits per lane from LDS with a 4-bit element transpose and returns the result as a `v2int` (two 32-bit words).

Parameter	Type	Description
`ptr`	`__shared__ v2int*`	Pointer to the lane's portion of the FP4 tile in LDS.

Returns `v2int` – two 32-bit words holding the lane's FP4 fragment after transposition.

14.7.2.2 `__builtin_amdgcn_ds_read_tr6_b96_v3i32`

```
v3int __builtin_amdgcn_ds_read_tr6_b96_v3i32(__shared__ v3int* ptr);
```

Loads 96 bits per lane from LDS with a 6-bit element transpose and returns the result as a `v3int` (three 32-bit words).

Parameter	Type	Description
`ptr`	`__shared__ v3int*`	Pointer to the lane's portion of the FP6 tile in LDS.

Returns `v3int` – three 32-bit words holding the lane's FP6 fragment after transposition.

14.7.3 8-bit operand loads

These builtins load and transpose operands for the FP8, BF8, and INT8 MFMA families.

14.7.3.1 `__builtin_amdgcn_ds_read_tr8_b64_v2i32`

```
v2int __builtin_amdgcn_ds_read_tr8_b64_v2i32(__shared__ v2int* ptr);
```

Loads 64 bits per lane from LDS with an 8-bit element transpose and returns the result as a `v2int` (two 32-bit words).

Parameter	Type	Description
`ptr`	`__shared__ v2int*`	Pointer to the lane's portion of the FP8 or BF8 tile in LDS.

Returns `v2int` – two 32-bit words holding the lane's FP8 or BF8 fragment after transposition.

14.7.3.2 `__builtin_amdgcn_ds_read_tr16_b64_v4i16`

```
v4short __builtin_amdgcn_ds_read_tr16_b64_v4i16(__shared__ v4short* ptr);
```

Loads 64 bits per lane from LDS with a 16-bit element transpose and returns the result as a `v4short` (four 16-bit words, each holding two packed INT8 values). Each lane receives four consecutive values along the M or N dimension of the matrix.

Parameter	Type	Description
`ptr`	`__shared__ v4short*`	Pointer to the lane's portion of the INT8 tile in LDS.

Returns `v4short` – four 16-bit words (eight packed INT8 values) holding the lane's INT8 fragment after transposition.

14.7.4 16-bit operand loads

These builtins load and transpose operands for the FP16 and BF16 MFMA families.

14.7.4.1 `__builtin_amdgcn_ds_read_tr16_b64_v4f16`

```
v4half __builtin_amdgcn_ds_read_tr16_b64_v4f16(__shared__ v4half* ptr);
```

Loads 64 bits per lane from LDS with a 16-bit element transpose and returns the result as a `v4half` (four FP16 values). Each lane receives four consecutive values along the M or N dimension of the matrix.

Parameter	Type	Description
`ptr`	`__shared__ v4half*`	Pointer to the lane's portion of the FP16 tile in LDS.

Returns `v4half` – four FP16 values holding the lane's FP16 fragment after transposition.

> **Note**
>
> This builtin requires the `__fp16` type (GCC-style half-precision), not `_Float16` (ISO C half-precision). When calling the builtin, cast pointers and vector types to `__fp16`-based variants. The standard MFMA compute builtins accept `_Float16`, so a cast is needed only at the `ds_read_tr` call site.

14.7.4.2 `__builtin_amdgcn_ds_read_tr16_b64_v4bf16`

```
v4bfloat __builtin_amdgcn_ds_read_tr16_b64_v4bf16(__shared__ v4bfloat* ptr);
```

Loads 64 bits per lane from LDS with a 16-bit element transpose and returns the result as a `v4bfloat` (four BF16 values). Each lane receives four consecutive values along the M or N dimension of the matrix.

Parameter	Type	Description
`ptr`	`__shared__ v4bfloat*`	Pointer to the lane's portion of the BF16 tile in LDS.

Returns `v4bfloat` – four BF16 values holding the lane's BF16 fragment after transposition.

14.8 FP16 GEMM example

The following excerpts from `docs/tools/example_codes/matrix_multiply_cdna4_mfma.hip` show how to use `__builtin_amdgcn_ds_read_tr16_b64_v4f16` in a complete FP16 GEMM kernel. Two cooperating structures are required: a tile policy that stores the B tile in LDS in row-major order, and a compute policy that issues the transpose load calls in place of the scalar load loop.

14.8.1 Step 1: store B in row-major order in LDS

The standard FP16 tile policy stores B transposed (`tile_b_T[N][K]`) so that scalar lane indexing can stride along the K dimension. The `ds_read_tr` path instead requires B in row-major order (`tile_b[K][N]`) so the hardware can scatter each N-column to its lane.

Listing 1: SingleBufferTilePolicyF16RowB shared-memory layout (matrix_multiply_cdna4_mfma.hip)

```
template<int BlockTileM_, int BlockTileN_, int KTileSize_>
struct SingleBufferTilePolicyF16RowB
{
    static constexpr int num_buffers  = 1;
    static constexpr int block_tile_m = BlockTileM_;
    static constexpr int block_tile_n = BlockTileN_;
    static constexpr int k_tile_size  = KTileSize_;

    struct SharedStorage
    {
        _Float16 tile_a[BlockTileM_][KTileSize_];
        // B stored row-major: tile_b[k][n], so that ds_read_tr can transpose
        // K-rows into per-lane N-column fragments in hardware.
        _Float16 tile_b[KTileSize_][BlockTileN_];
    };

    __device__ static void prologue(SharedStorage& /*smem*/,
```

(continues on next page)

(continued from previous page)

```
                                    const _Float16* /*A*/, const _Float16* /*B*/,
                                    int /*block_row_start*/, int /*block_col_start*/,
                                    int /*m*/, int /*n*/, int /*k*/,
                                    int /*tid*/, int /*num_threads*/) {}

__device__ static void prefetch(SharedStorage&  smem,
                                    const _Float16* A,
                                    const _Float16* B,
                                    int             t_next,
                                    int block_row_start, int block_col_start,
                                    int m, int n, int k,
                                    int tid, int num_threads)
{
    const int t = t_next;

    // Load A tile: row-major, M rows × K cols.
    const int tile_a_elems = block_tile_m * k_tile_size;
    for(int idx = tid; idx < tile_a_elems; idx += num_threads)
    {
        const int tile_row   = idx / k_tile_size;
        const int tile_col   = idx % k_tile_size;
        const int global_row = block_row_start + tile_row;
        const int global_col = t * k_tile_size + tile_col;
        smem.tile_a[tile_row][tile_col] =
            (global_row < m && global_col < k)
                ? A[global_row * k + global_col]
                : static_cast<_Float16>(0.0f);
    }

    // Load B tile: row-major K × N layout in LDS.
    const int tile_b_elems = k_tile_size * block_tile_n;
    for(int idx = tid; idx < tile_b_elems; idx += num_threads)
    {
        const int tile_k_idx   = idx / block_tile_n;
        const int tile_col_idx = idx % block_tile_n;
        const int global_row   = t * k_tile_size + tile_k_idx;
        const int global_col   = block_col_start + tile_col_idx;
        smem.tile_b[tile_k_idx][tile_col_idx] =
            (global_row < k && global_col < n)
                ? B[global_row * n + global_col]
                : static_cast<_Float16>(0.0f);
    }
}

__device__ static void acquire(int /*t*/) { __syncthreads(); }
__device__ static void release(int /*t*/) { __syncthreads(); }
__device__ static int  buf_idx(int /*t*/) { return 0; }

__device__ static _Float16* get_tile_a(SharedStorage& smem, int /*buf*/)
{
    return &smem.tile_a[0][0];
}
```

(continues on next page)

(continued from previous page)

```
    // Returns the row-major B tile base pointer; the ds_read_tr policy will
    // compute per-lane pointers into this region.
    __device__ static _Float16* get_tile_b_T(SharedStorage& smem, int /*buf*/)
    {
        return &smem.tile_b[0][0];
    }
};
```

14.8.2 Step 2: load B with hardware-transposed calls

`MfmaCdna4F16TrPolicy::load_b` replaces the scalar load loop with two `ds_read_tr16_b64_v4f16` calls. Each call loads 64 bits (four FP16 values) per lane and performs the 16-element transpose in hardware.

`v_mfma_f32_16x16x32_f16` consumes 32 K-positions per call (`k_step = 32`). The 64 lanes form four groups of 16 (`g = lane_id / 16`), each group covering 8 consecutive K-positions starting at `ki + 8 * g`. Each `ds_read_tr16_b64_v4f16` call fills half of a group's 8 K-positions (see the K-position split table above), so two calls per `load_b` invocation produce the full 8-element B fragment:

- Call 1 (`r0`): each group loads the first 4 of its 8 K-positions. Wavefront-wide this covers K positions `ki + 0` to `ki + 3`, `ki + 8` to `ki + 11`, `ki + 16` to `ki + 19`, and `ki + 24` to `ki + 27`.
- Call 2 (`r1`): each group loads the remaining 4 K-positions. Wavefront-wide: `ki + 4` to `ki + 7`, `ki + 12` to `ki + 15`, `ki + 20` to `ki + 23`, and `ki + 24` to `ki + 31`.

Each lane addresses its own N-column: `column = tile_b_col + (lane_id mod 16)`. The pointer stride between the two calls is `4 * BlockTileN * sizeof(__fp16)` bytes (four K-rows of the B tile in LDS).

Listing 2: MfmaCdna4F16TrPolicy with ds_read_tr B loads (matrix_multiply_cdna4_mfma.hip)

```
struct MfmaCdna4F16TrPolicy
{
    // -- ComputePolicy constants ------------------------------------------------
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int frag_size_m     = 8;
    static constexpr int frag_size_n     = 8;
    static constexpr int effective_lanes = 64;
    // k_step=32: one v_mfma_f32_16x16x32_f16 call consumes 32 K-positions,
    // filled by two ds_read_tr16_b64_v4f16 calls.
    static constexpr int k_step          = 32;

    using elem_a = _Float16;
    using elem_b = _Float16;

    // Accumulator and zero() are identical to MfmaCdna4F16Policy.
    using v4float = float [[clang::ext_vector_type(4)]];

    struct Accumulator
    {
        v4float regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
```

(continues on next page)

(continued from previous page)

```
    acc.regs = v4float{};
}

// thread_tile_offset is identical to MfmaCdna4F16Policy.
__device__ static void thread_tile_offset(int  tid,
                                          int  /*lane_id*/,
                                          int* thread_row,
                                          int* thread_col)
{
    constexpr int waves_n = 2;
    const int wid   = tid / 64;
    const int wid_x = wid % waves_n;
    const int wid_y = wid / waves_n;
    *thread_row = wid_y * 16;
    *thread_col = wid_x * 16;
}

// -- load_a ------------------------------------------------------------
// Scalar load, identical to MfmaCdna4F16Policy::load_a.
__device__ static void load_a(const _Float16* tile_a_ptr,
                              int             tile_a_row,
                              int             ki,
                              int             k_tile_size,
                              int             lane_id,
                              elem_a        (&frag)[8])
{
    const int row   = tile_a_row + (lane_id % 16);
    const int k_off = ki + 8 * (lane_id / 16);
    #pragma unroll
    for(int e = 0; e < 8; ++e)
        frag[e] = tile_a_ptr[row * k_tile_size + k_off + e];
}

// -- load_b ------------------------------------------------------------
// Hardware-transposed load using ds_read_tr16_b64_v4f16.
//
// tile_b_ptr points to the base of the row-major B tile in LDS:
//   tile_b[k][n], dimensions KTileSize × BlockTileN.
//
// Each lane belongs to one of four 16-lane groups (g = lane_id / 16),
// handling K-positions ki + 8*g through ki + 8*g + 7.  Two ds_read_tr
// calls fill the eight-element B fragment per lane:
//
//   local_tx = lane_id % 16
//   k_base   = ki + 8 * (lane_id / 16)
//
//   Call 1: ptr = &tile_b[k_base + local_tx/4][tile_b_col + (local_tx%4)*4]
//           -> frag[0..3]
//   Call 2: ptr = &tile_b[k_base + 4 + local_tx/4][tile_b_col + (local_tx%4)*4]
//           -> frag[4..7]
//
// The stride between the two calls is 4 rows of the B tile (4 * BlockTileN
```

(continues on next page)

(continued from previous page)

```
    // elements).  The hardware transpose gathers strided K elements across
    // adjacent lanes so that each lane receives its own column's data.
    __device__ static void load_b(const _Float16* tile_b_ptr,
                                  int             tile_b_col,
                                  int             ki,
                                  int             block_tile_n,
                                  int             lane_id,
                                  elem_b        (&frag)[8])
    {
#if defined(__gfx950__)
        using v4half = __fp16 [[clang::ext_vector_type(4)]];

        const int local_tx = lane_id % 16;
        const int k_base   = ki + 8 * (lane_id / 16);

        __fp16* base0 =
            const_cast<__fp16*>(reinterpret_cast<const __fp16*>(tile_b_ptr))
            + (k_base + local_tx / 4) * block_tile_n
            + tile_b_col + (local_tx % 4) * 4;

        __fp16* base1 =
            const_cast<__fp16*>(reinterpret_cast<const __fp16*>(tile_b_ptr))
            + (k_base + 4 + local_tx / 4) * block_tile_n
            + tile_b_col + (local_tx % 4) * 4;

        const v4half r0 = __builtin_amdgcn_ds_read_tr16_b64_v4f16(
            (__attribute__((address_space(3))) v4half*)(base0));
        const v4half r1 = __builtin_amdgcn_ds_read_tr16_b64_v4f16(
            (__attribute__((address_space(3))) v4half*)(base1));

        frag[0] = r0[0]; frag[1] = r0[1]; frag[2] = r0[2]; frag[3] = r0[3];
        frag[4] = r1[0]; frag[5] = r1[1]; frag[6] = r1[2]; frag[7] = r1[3];
#endif
    }

    // mma and store_c are identical to MfmaCdna4F16Policy.
    __device__ static void mma(Accumulator&       acc,
                               const elem_a     (&a_frag)[8],
                               const elem_b     (&b_frag)[8])
    {
#if defined(__gfx950__)
        using v8half = _Float16 [[clang::ext_vector_type(8)]];
        const v8half a_vec = {a_frag[0], a_frag[1], a_frag[2], a_frag[3],
                              a_frag[4], a_frag[5], a_frag[6], a_frag[7]};
        const v8half b_vec = {b_frag[0], b_frag[1], b_frag[2], b_frag[3],
                              b_frag[4], b_frag[5], b_frag[6], b_frag[7]};
        acc.regs = __builtin_amdgcn_mfma_f32_16x16x32_f16(
            a_vec, b_vec, acc.regs,
            /*cbsz=*/0, /*abid=*/0, /*blgp=*/0);
#endif
    }
```

(continues on next page)

(continued from previous page)

```
    __device__ static void store_c(const Accumulator& acc,
                                   float*              C,
                                   int                 out_row_base,
                                   int                 out_col_base,
                                   int                 m,
                                   int                 n,
                                   int                 lane_id)
    {
        const int L = lane_id;
        const int j = L % 16;
        #pragma unroll
        for(int G = 0; G < 4; ++G)
        {
            const int i = 4 * (L / 16) + (G % 4);
            const int r = out_row_base + i;
            const int c = out_col_base + j;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[G];
        }
    }
};
```

14.9 Common MFMA parameters

Every MFMA builtin across all CDNA generations shares trailing integer parameters that must be compile-time integer constants. Some parameters apply to all MFMA variants; others are specific to dense or sparse forms.

14.9.1 Shared parameters

The following parameters are present on every MFMA builtin.

Parameter	Type	Description
`cbsz`	int	Control Broadcast Size modifier. Supported by MFMA builtins operating on multiple input blocks for $\boldsymbol{A}$. Legal range 0..4, but must not exceed $\log_2(\text{input blocks})$. Setting `cbsz` informs the instruction to broadcast values of one chosen input block to $2^{cbsz} - 1$ neighboring blocks in $\boldsymbol{A}$. The input block is chosen by setting `abid`. For example, for a 16-block $\boldsymbol{A}$ matrix, setting `cbsz` to `1` results in blocks 0 and 1 receiving the same input values, blocks 2 and 3 receiving the same input values, blocks 4 and 5 receiving the same input values, etc. Setting `cbsz` to `0` results in no broadcast.
`abid`	int	$\boldsymbol{A}$-matrix Broadcast Identifier. Supported by MFMA builtins operating on multiple input blocks for $\boldsymbol{A}$. Used together with `cbsz`; indicates which input block is selected for broadcast to neighboring blocks in $\boldsymbol{A}$. For example, for a 16-block $\boldsymbol{A}$ matrix, setting `cbsz` to `2` and `abid` to `1` broadcasts block 1's values to blocks 0, 2, and 3; block 5's values to blocks 4, 6, and 7; etc.

14.9.2 Dense MFMA parameters

The following parameter applies to dense MFMA builtins only.

Parameter	Type	Description
`blgp`	int	B-matrix Lane Group Pattern modifier. Allows a constrained set of swizzling operations on B data between lanes. Supported values: • `0`: No swizzling; normal matrix layout for B. • `1`: Data from lanes 0—31 is broadcast into lanes 32—63. • `2`: Data from lanes 32—63 is broadcast into lanes 0—31. • `3`: Data from all lanes is rotated down by 16 positions: lane 0's data goes to lane 48, lane 16's data goes to lane 0, etc. • `4`: Data from lanes 0—15 is broadcast into lanes 16—31, 32—47, and 48—63. • `5`: Data from lanes 16—31 is broadcast into lanes 0—15, 32—47, and 48—63. • `6`: Data from lanes 32—47 is broadcast into lanes 0—15, 16—31, and 48—63. • `7`: Data from lanes 48—63 is broadcast into lanes 0—15, 16—31, and 32—47. **Note** **CDNA3 FP64 exception:** for FP64 MFMA builtins on CDNA3 (`gfx942`), `blgp` is repurposed as a 3-bit NEG modifier. Each bit negates the corresponding source operand: bit 0 negates A, bit 1 negates B, and bit 2 negates C. Lane-group pattern swizzling is not available for FP64 on CDNA3.

14.9.3 Sparse MFMA parameters

Sparse MFMA (SMFMAC) builtins share the `cbsz` and `abid` parameters described above with the same semantics as their dense counterparts. In addition, SMFMAC builtins accept the following parameter:

Parameter	Type	Description
`idx`	int	Sparsity index. Encodes which two of the four positions in each group of four elements along the K dimension hold the non-zeros in the compressed $\boldsymbol{A}$ matrix. The bit width of the index scales with K: a 2-bit field per group of four is needed, so K=16 requires 8 bits, K=32 requires 16 bits, and K=64 requires 32 bits. This is a runtime value, not a compile-time constant.

Note

SMFMAC builtins do not support the `blgp` ($\boldsymbol{B}$-matrix Lane Group Pattern) modifier available on dense MFMA instructions.

CHAPTER

FIFTEEN

WMMA BUILTINS

Wave-Matrix Multiply-Accumulate (WMMA) builtins let you issue hardware matrix multiply-accumulate operations directly from HIP device code. This reference covers both dense and sparse WMMA variants on AMD RDNA architectures (AMD Radeon GPUs). Coverage is organized first by operand density – dense or sparse – and then by RDNA generation.

- *Dense WMMA builtins*
  - *RDNA3 dense WMMA builtins*
  - *RDNA4 dense WMMA builtins*
- Sparse WMMA builtins
  - *RDNA4 sparse WMMA builtins*
- *RDNA4 WMMA transpose load builtins*

15.1 Dense WMMA builtins

Dense Wave-Matrix Multiply-Accumulate (WMMA) builtins multiply full A and B fragments and accumulate the result into a C fragment. All operands are fully populated with data. Select a generation below for the corresponding builtin reference.

- *RDNA3 dense WMMA builtins*
- *RDNA4 dense WMMA builtins*

15.1.1 RDNA3 dense WMMA builtins

Wave-Matrix Multiply-Accumulate (WMMA) builtins let you issue hardware dense matrix multiply-accumulate operations directly from HIP device code on RDNA3 and RDNA3.5 GPUs (`gfx1100`–`gfx1103`, `gfx1150`–`gfx1153`). Each WMMA instruction multiplies a dense $\boldsymbol{A}$ fragment by a dense $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single 32-wide wavefront.

WMMA is the dense variant of the RDNA wave-matrix instruction family. RDNA4 extends the family with sparse WMMA variants documented on the *RDNA4 sparse WMMA builtins* page. CDNA (Instinct) GPUs provide a comparable dense operation through *Matrix Fused Multiply-Accumulate (MFMA)*. The two differ in wavefront size (wave32 for WMMA, wave64 for MFMA), accumulator storage (ordinary Vector General-Purpose Registers (VGPRs) for WMMA, dedicated accVGPRs for MFMA on CDNA and CDNA2), and tile shapes.

> **Note**
>
> RDNA3 and RDNA3.5 GPUs run all shader programs in `wave32` mode by default. The `_w32` suffix in each builtin name reflects this: all WMMA builtins on this page require `wavefrontsize32`.

15.1.1.1 Architecture availability

The builtins on this page target RDNA3 and RDNA3.5 GPUs. To automatically enable them, pass the Low Level Virtual Machine (LLVM) target architecture flag at compile time:

```
amdclang++ --offload-arch=gfx1100 ...
amdclang++ --offload-arch=gfx1150 ...
```

Equivalent dense WMMA builtins for RDNA4 are documented on the *RDNA4 dense WMMA builtins* page.

15.1.1.2 Naming convention

All dense WMMA builtins on this page follow the pattern:

```
__builtin_amdgcn_wmma_<out_type>_<M>x<N>x<K>_<in_type>[_<in_type_b>]_w32
```

out_type
: Accumulator element type (`f32`, `f16`, `bf16`, or `i32`).

M, N, K
: Tile dimensions in elements. The instruction computes the contribution of a K-wide panel of $\boldsymbol{A}$ ($M \times K$) and a K-wide panel of $\boldsymbol{B}$ ($K \times N$) to an $M \times N$ output tile.

in_type
: Input element type of $\boldsymbol{A}$ (and $\boldsymbol{B}$ when both share the same type): `f16`, `bf16`, `iu8`, or `iu4`. The `iu` prefix means the builtin accepts either signed or unsigned integers, controlled by the `a_neg` and `b_neg` parameters.

_w32
: Wavefront size suffix. All RDNA3 WMMA builtins use wave32.

15.1.1.3 Fragment layouts

All WMMA builtins on this page use a 16×16 output tile computed by one wave32 wavefront. The 32 lanes split into two groups of 16. Unlike RDNA4's contiguous-block layout, RDNA3 interleaves the groups across rows: even-numbered rows are held by lanes 0–15, odd-numbered rows by lanes 16–31. This is the *matrix replication* pattern: lanes 0–15 and lanes 16–31 carry identical copies of the A and B input fragments, and the hardware fuses their contributions into alternating output rows. The diagrams below show the mapping between matrix elements and lane or VGPR positions for each operand.

15.1.1.3.1 Accumulator layout

Each lane holds 8 output elements across VGPRs 0–7.

Given lane L and VGPR index g:

$$i = g \cdot 2 + \lfloor \frac{L}{16} \rfloor$$
$$j = L \bmod 16$$

Conversely, given output element (i, j):

$$\text{lane} = (i \bmod 2) \cdot 16 + j$$
$$\text{VGPR} = \lfloor \frac{i}{2} \rfloor$$

The row-to-lane mapping:

16×16 WMMA accumulator layout - wave32 (RDNA3)

i \ j		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=2	lanes 0–15	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=4	lanes 0–15	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=6	lanes 0–15	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=8	lanes 0–15	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
i=10	lanes 0–15	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
i=12	lanes 0–15	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
i=14	lanes 0–15	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=1	lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
i=3	lanes 16–31	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
i=5	lanes 16–31	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
i=7	lanes 16–31	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
i=9	lanes 16–31	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
i=11	lanes 16–31	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
i=13	lanes 16–31	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
i=15	lanes 16–31	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Accumulator for all 16×16 WMMA intrinsics on RDNA3/3.5.
Cell value = VGPR index g (0–7). Column j = matrix column = lane % 16.
Even rows → lanes 0–15; odd rows → lanes 16–31 (matrix replication).
Formulas: lane = $(i \bmod 2) \cdot 16 + j$; VGPR = $\lfloor i/2 \rfloor$.

Rows	Lanes	VGPRs
0	0–15	0
1	16–31	0
2	0–15	1
3	16–31	1
4	0–15	2
5	16–31	2
6	0–15	3
7	16–31	3
8	0–15	4
9	16–31	4
10	0–15	5
11	16–31	5
12	0–15	6
13	16–31	6
14	0–15	7
15	16–31	7

15.1.1.3.2 srcA and srcB (FP16 and BF16)

Each lane holds 16 input elements of $\boldsymbol{A}$ (`v16half` for FP16 or `v16short` for BF16; 8 VGPRs × 2 elements), covering one row of the A fragment; similarly, each lane holds 16 elements of $\boldsymbol{B}$ covering one column of the B fragment.

Due to matrix replication, **lanes 0–15 and 16–31 must load identical copies** of every row and column. Only `lane %`

`16` determines which row or column a lane covers – the upper lane-group bit is ignored. This means your `load_a` and `load_b` code must supply the same data to both halves of the wavefront.

Lane L covers:

- srcA row $= L \bmod 16$
- srcB column $= L \bmod 16$

The row and column-to-lane mapping:

Row (srcA) or Column (srcB)	Lanes	VGPRs
0	0, 16	0–7
1	1, 17	0–7
2	2, 18	0–7
3	3, 19	0–7
4	4, 20	0–7
5	5, 21	0–7
6	6, 22	0–7
7	7, 23	0–7
8	8, 24	0–7
9	9, 25	0–7
10	10, 26	0–7
11	11, 27	0–7
12	12, 28	0–7
13	13, 29	0–7
14	14, 30	0–7
15	15, 31	0–7

The 16 FP16 elements are packed contiguously across 8 VGPRs (2 FP16 values per VGPR):

VGPR	K positions
0	K {0, 1}
1	K {2, 3}
2	K {4, 5}
3	K {6, 7}
4	K {8, 9}
5	K {10, 11}
6	K {12, 13}
7	K {14, 15}

15.1.1.4 Using WMMA builtins as a compute policy

Using MFMA builtins as a compute policy explains the `ComputePolicy` pattern used to separate the multiply-accumulate logic from the rest of a kernel. The example below implements `WmmaRdna3F16Policy` using `__builtin_amdgcn_wmma_f32_16x16x16_f16_w32`.

Each wavefront computes a single 16×16 output tile. Due to matrix replication, lanes 0–15 and 16–31 load identical A and B fragments (`lane_id % 16` selects the row or column), and each lane supplies 16 FP16 values for both operands.

The complete source file is available for download:

- `matrix_multiply_rdna3_wmma.hip`

Policy constants

`v_wmma_f32_16x16x16_f16` consumes 16 FP16 K positions per call, so `k_step = 16`. The wavefront holds the entire 16×16 tile: `thread_tile_m = thread_tile_n = 16` and `effective_lanes = 32`.

Accumulator layout

The builtin returns a `v8float` holding 8 VGPR values per lane. The `store_c()` pass maps `(lane, VGPR index)` back to (i, j) coordinates using the RDNA3 interleaved accumulator layout.

```
struct WmmaRdna3F16Policy
{
    // -- ComputePolicy constants ------------------------------------------
    // One wavefront (wave32) owns a 16x16 output tile.
    // All 32 lanes hold unique output elements (8 elements per lane).
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int effective_lanes = 32;
    // v_wmma_f32_16x16x16_f16 consumes 16 K-positions per call (k_step=16).
    // Each lane provides 16 FP16 values (v16half) for A and 16 for B.
    // Lanes 0--15 and 16--31 mirror each other (matrix replication).
    static constexpr int k_step          = 16;

    // -- Vector types -----------------------------------------------------
    using v16half = _Float16 [[clang::ext_vector_type(16)]];
    using v8float = float    [[clang::ext_vector_type(8)]];

    using elem_a = _Float16;
    using elem_b = _Float16;

    // -- Accumulator ------------------------------------------------------
    // v8float holds 8 FP32 VGPRs per lane (VGPRs 0--7).
    struct Accumulator
    {
        v8float regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
        acc.regs = v8float{};
    }

    // -- thread_tile_offset -----------------------------------------------
    // The entire 16x16 tile belongs to one wavefront (wave32).  Multiple
    // wavefronts in a block cover different 16x16 sub-tiles.
    //
    //   wavefront id within block: wid = tid / 32
    //   waves_n = block_tile_n / thread_tile_n = 32 / 16 = 2
    //   wid_x = wid % waves_n
    //   wid_y = wid / waves_n
    //   *thread_row = wid_y * 16
    //   *thread_col = wid_x * 16
    __device__ static void thread_tile_offset(int  tid,
                                              int  /*lane_id*/,
```

(continues on next page)

(continued from previous page)

```
                                          int* thread_row,
                                          int* thread_col)
{
    constexpr int waves_n = 2; // block_tile_n / thread_tile_n = 32 / 16
    const int wid   = tid / 32;
    const int wid_x = wid % waves_n;
    const int wid_y = wid / waves_n;
    *thread_row = wid_y * 16;
    *thread_col = wid_x * 16;
}

// -- Fragment loads ------------------------------------------------------
// load_a: each lane reads 16 FP16 values from LDS into the fragment
// array.  The 16-element _Float16 array has the same layout as v16half
// and is reinterpreted as such in mma().
//
// v_wmma_f32_16x16x16_f16 (wave32) uses matrix replication: lanes 0--15
// and 16--31 index identically via (lane_id mod 16).  Each lane loads
// one full row of the 16x16 A-tile.
//
//   A-row index : tile_a_row + (lane_id mod 16)
//   K-position e: ki + e   (e = 0..15)
__device__ static void load_a(const _Float16* tile_a_ptr,
                              int             tile_a_row,
                              int             ki,
                              int             k_tile_size,
                              int             lane_id,
                              elem_a        (&frag)[16])
{
    const int row = tile_a_row + (lane_id % 16);
    #pragma unroll
    for(int e = 0; e < 16; ++e)
        frag[e] = tile_a_ptr[row * k_tile_size + ki + e];
}

// load_b: each lane reads 16 FP16 values from LDS (symmetric to load_a).
//
// tile_b_T is stored transposed: tile_b_T[(tile_b_col + col)][k_off].
//   B-col index : tile_b_col + (lane_id mod 16)
//   K-position e: ki + e   (e = 0..15)
__device__ static void load_b(const _Float16* tile_b_T_ptr,
                              int             tile_b_col,
                              int             ki,
                              int             k_tile_size,
                              int             lane_id,
                              elem_b        (&frag)[16])
{
    const int col = tile_b_col + (lane_id % 16);
    #pragma unroll
    for(int e = 0; e < 16; ++e)
        frag[e] = tile_b_T_ptr[col * k_tile_size + ki + e];
}
```

(continues on next page)

(continued from previous page)

```
    // -- mma ---------------------------------------------------------------
    // Issue one v_wmma_f32_16x16x16_f16 instruction.
    //
    // The _Float16[16] fragment arrays share the same in-register layout as
    // v16half; a reinterpret_cast avoids any per-element copy.
    __device__ static void mma(Accumulator&       acc,
                               const elem_a     (&a_frag)[16],
                               const elem_b     (&b_frag)[16])
    {
#if defined(__gfx1100__) || defined(__gfx1101__) || defined(__gfx1102__) || \
    defined(__gfx1103__) || defined(__gfx1150__) || defined(__gfx1151__) || \
    defined(__gfx1152__) || defined(__gfx1153__)
        acc.regs = __builtin_amdgcn_wmma_f32_16x16x16_f16_w32(
            reinterpret_cast<const v16half&>(a_frag),
            reinterpret_cast<const v16half&>(b_frag),
            acc.regs);
#endif
    }

    // -- store_c -----------------------------------------------------------
    // Scatter the 8 VGPR values to their global-memory positions.
    //
    // Inverse layout (lane L, VGPR index G in [0,7]):
    //   row = G * 2 + (L / 16)
    //   col = L mod 16
    __device__ static void store_c(const Accumulator& acc,
                                   float*              C,
                                   int                 out_row_base,
                                   int                 out_col_base,
                                   int                 m,
                                   int                 n,
                                   int                 lane_id)
    {
        const int L = lane_id;
        const int j = L % 16;

        #pragma unroll
        for(int G = 0; G < 8; ++G)
        {
            const int i = G * 2 + (L / 16);
            const int r = out_row_base + i;
            const int c = out_col_base + j;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[G];
        }
    }
};
```

Instantiating the kernel

With `WmmaRdna3F16Policy` in hand, plug it into the generic kernel alongside any `TilePolicy` whose `block_tile_m` and `block_tile_n` are multiples of 16 and whose `k_tile_size` is a multiple of `k_step = 16`.

```
// TilePolicy for the FP16 WMMA path (transposed B in LDS).
using WmmaTilePolicy = SingleBufferTilePolicyF16<32, 32, 16>;
```

```
// Launch parameters:
//
// RDNA 3/3.5 (gfx11): 2x2 wavefronts of 32 lanes = 128 threads per block.
//   block_tile_m = 32, thread_tile_m = 16  → 2 wavefronts along M
//   block_tile_n = 32, thread_tile_n = 16  → 2 wavefronts along N
//
// Fallback (scalar): 8x8 thread tiles in a 32x32 block → 16 threads per
//   block, padded to one full wavefront of 32.
//
// The kernel to launch and its thread count are selected at runtime based
// on the device architecture string returned by hipGetDeviceProperties.
//
// Grid: one block per 32x32 output tile.
constexpr int BLOCK_TILE_M = 32;
constexpr int BLOCK_TILE_N = 32;
```

```
    // Grid: one block per 32x32 output tile.
    const dim3 grid((N + BLOCK_TILE_N - 1) / BLOCK_TILE_N,
                    (M + BLOCK_TILE_M - 1) / BLOCK_TILE_M);

    float       ms = 0.0f;
    const char* policy_label = nullptr;

    if(is_gfx11)
    {
        auto launch = [&]()
        {
            matrix_multiply_generic<WmmaTilePolicy,
                                    WmmaRdna3F16Policy,
                                    _Float16>
                <<<grid, dim3(128)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<WmmaTilePolicy, WmmaRdna3F16Policy> [gfx11]";
    }
    else
    {
        // Scalar FP32 fallback: upload A and B as FP32.
        std::vector<float> h_A_f32(M * K);
        for(int i = 0; i < M * K; ++i)
            h_A_f32[i] = static_cast<float>(h_A[i]);
        std::vector<float> h_B_f32(K * N, 0.0f);
        for(int i = 0; i < std::min(K, N); ++i)
            h_B_f32[i * N + i] = 1.0f;
```

(continues on next page)

(continued from previous page)

```
        float* d_A_f32;
        float* d_B_f32;
        HIP_CHECK(hipMalloc(&d_A_f32, sizeof(float) * M * K));
        HIP_CHECK(hipMalloc(&d_B_f32, sizeof(float) * K * N));
        HIP_CHECK(hipMemcpy(d_A_f32, h_A_f32.data(), sizeof(float) * M * K,
                            hipMemcpyHostToDevice));
        HIP_CHECK(hipMemcpy(d_B_f32, h_B_f32.data(), sizeof(float) * K * N,
                            hipMemcpyHostToDevice));

        auto launch = [&]()
        {
            matrix_multiply_generic<SingleBufferTilePolicyF<32, 32, 8>,
                                    ScalarFMAPolicy<8, 8, 4>,
                                    float>
                <<<grid, dim3(32)>>>(d_A_f32, d_B_f32, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<SingleBufferTilePolicyF, ScalarFMAPolicy>␣
↪[fallback]";
        HIP_CHECK(hipFree(d_A_f32));
        HIP_CHECK(hipFree(d_B_f32));
    }
```

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx1100 \
    matrix_multiply_rdna3_wmma.hip -o mm_rdna3_wmma
./mm_rdna3_wmma
```

> **Note**
>
> `WmmaRdna3F16Policy` requires an RDNA3 or RDNA3.5 GPU (`gfx1100` or later gfx11 target). The `#if defined(__gfx1100__) || ...` guard in the example file falls back to `ScalarFMAPolicy` on other targets, so the file compiles without modification.

15.1.1.5 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v16half  = _Float16 [[clang::ext_vector_type(16)]];
using v16short = short [[clang::ext_vector_type(16)]];
using v8float  = float [[clang::ext_vector_type(8)]];
using v8half   = _Float16 [[clang::ext_vector_type(8)]];
using v8short  = short [[clang::ext_vector_type(8)]];
using v8int    = int [[clang::ext_vector_type(8)]];
using v4int    = int [[clang::ext_vector_type(4)]];
```

15.1.1.6 Common parameters

The following parameters are shared across multiple WMMA builtins on this page.

Parameter	Type	Description
opse	bool (compile time constant)	FP16 and BF16 accumulate variants only. Selects which half of the 16-element accumulator register holds the result. When false, the 8 output elements occupy the low half of each VGPR pair (even-indexed elements); when true, the high half (odd-indexed elements). The _tied variants additionally constrain srcC and the return value to the same physical register so that the non-selected half is preserved without an extra copy.
a_ne	bool (compile time constant)	Integer variants only. When true, the $\boldsymbol{A}$ elements are treated as signed integers; when false, as unsigned.
b_ne	bool (compile time constant)	Integer variants only. Same as a_neg but for $\boldsymbol{B}$.
clam	bool (compile time constant)	Integer variants only. When true, the INT32 accumulator output is clamped to the representable range of the input type on overflow.

15.1.1.7 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency.

Builtin	Ops	Cycle count
__builtin_amdgcn_wmma_f32_16x16x16_f16_w32	8192	32
__builtin_amdgcn_wmma_f32_16x16x16_bf16_w32	8192	32
__builtin_amdgcn_wmma_f16_16x16x16_f16_w32	8192	32
__builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32	8192	32
__builtin_amdgcn_wmma_i32_16x16x16_iu8_w32	8192	32
__builtin_amdgcn_wmma_i32_16x16x16_iu4_w32	8192	16

15.1.1.8 Builtin reference

The following sections list every dense WMMA builtin available on RDNA3 and RDNA3.5, grouped by accumulator type.

15.1.1.8.1 FP32-accumulate builtins

These builtins accumulate into FP32 and accept FP16 or BF16 matrix inputs.

15.1.1.8.1.1 FP16 inputs

The following builtins use FP16 matrix inputs.

15.1.1.8.1.2 `__builtin_amdgcn_wmma_f32_16x16x16_f16_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_f16_w32(
    v16half srcA,
    v16half srcB,
    v8float srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with FP16 inputs. Each lane supplies 16 FP16 values for both $\boldsymbol{A}$ and $\boldsymbol{B}$. Due to matrix replication, the 16-element fragment is indexed by `lane_id % 16`, so lanes 0–15 and 16–31 carry identical copies.

Parameter	Type	Description
`srcA`	v16half	Sixteen FP16 elements of $\boldsymbol{A}$ per lane (one full row of the 16×16 A tile, replicated across both lane groups).
`srcB`	v16half	Sixteen FP16 elements of $\boldsymbol{B}$ per lane (one full column of the 16×16 B tile, replicated across both lane groups).
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.1.8.1.3 BF16 inputs

The following builtins use BF16 matrix inputs.

15.1.1.8.1.4 `__builtin_amdgcn_wmma_f32_16x16x16_bf16_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_bf16_w32(
    v16short srcA,
    v16short srcB,
    v8float  srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with BF16 inputs. BF16 values are passed as `short` (16-bit storage). Otherwise identical in structure to the FP16 variant.

Parameter	Type	Description
`srcA`	v16short	Sixteen BF16 elements of $\boldsymbol{A}$ per lane (BF16 stored as `short`).
`srcB`	v16short	Sixteen BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA × srcB + srcC).

15.1.1.8.2 FP16-accumulate builtins

These builtins accumulate into FP16 with FP16 inputs. Both accept an `opsel` parameter that selects which half of the 16-element accumulator register is written (see *Common parameters*). The `_tied` variant constrains `srcC` and the return value to the same physical register, preserving the non-selected half in place.

15.1.1.8.2.1 FP16 inputs

The following builtins use FP16 matrix inputs.

15.1.1.8.2.2 `__builtin_amdgcn_wmma_f16_16x16x16_f16_w32`

Signature and parameters for this builtin.

```
v16half __builtin_amdgcn_wmma_f16_16x16x16_f16_w32(
    v16half srcA,
    v16half srcB,
    v16half srcC,
    bool    opsel);
```

Computes one step of a dense 16 × 16 FP16 accumulation. Both inputs and the accumulator are FP16. The `opsel` parameter selects the storage half of the accumulator register (see *Common parameters*).

The accumulator is a 16-element vector (`v16half`, 8 VGPRs × 2 FP16); the 8 result elements occupy either the low (`opsel = false`) or high (`opsel = true`) half of each VGPR pair.

Parameter	Type	Description
`srcA`	v16half	Sixteen FP16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v16half	Sixteen FP16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16half	Accumulator input: sixteen FP16 elements per lane (8 VGPRs). Only the half selected by `opsel` is read.
`opsel`	`bool` (compile-time constant)	Selects accumulator half. `false` = low half of each VGPR pair; `true` = high half.

Returns `v16half` – updated accumulator (srcA × srcB + srcC).

15.1.1.8.2.3 `__builtin_amdgcn_wmma_f16_16x16x16_f16_tied_w32`

Signature and parameters for this builtin.

```
v16half __builtin_amdgcn_wmma_f16_16x16x16_f16_tied_w32(
    v16half srcA,
    v16half srcB,
```

(continues on next page)

(continued from previous page)

```
    v16half srcC,
    bool    opsel);
```

Register-tied variant of `__builtin_amdgcn_wmma_f16_16x16x16_f16_w32`. The compiler constrains `srcC` and the return value to the same physical register, guaranteeing that the non-selected half of the accumulator (the half not written by this `opsel` setting) is preserved in place.

Use this variant when you chain two WMMA calls with opposite `opsel` values into the same `v16half` accumulator so that both halves remain live without an extra register copy.

Parameters, types, and return value are identical to the non-tied variant above.

15.1.1.8.3 BF16-accumulate builtins

These builtins accumulate into BF16 with BF16 inputs. Both accept an `opsel` parameter and the `_tied` variant provides register tying, as described in the FP16 section above.

15.1.1.8.3.1 BF16 inputs

The following builtins use BF16 matrix inputs.

15.1.1.8.3.2 `__builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32`

Signature and parameters for this builtin.

```
v16short __builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32(
    v16short srcA,
    v16short srcB,
    v16short srcC,
    bool     opsel);
```

Computes one step of a dense 16×16 BF16 accumulation. Both inputs and the accumulator are BF16 (stored as `short`). The `opsel` parameter selects the storage half of the accumulator register (see *Common parameters*).

The accumulator is a 16-element vector (`v16short`, 8 VGPRs × 2 BF16); the 8 result elements occupy either the low (`opsel = false`) or high (`opsel = true`) half of each VGPR pair.

Parameter	Type	Description
`srcA`	v16short	Sixteen BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v16short	Sixteen BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v16short	Accumulator input: sixteen BF16 elements per lane (8 VGPRs, stored as `short`). Only the half selected by `opsel` is read.
`opsel`	`bool` (compile-time constant)	Selects accumulator half. `false` = low half of each VGPR pair; `true` = high half.

Returns `v16short` – updated accumulator (BF16 stored as `short`) (srcA × srcB + srcC).

15.1.1.8.3.3 `__builtin_amdgcn_wmma_bf16_16x16x16_bf16_tied_w32`

Signature and parameters for this builtin.

```
v16short __builtin_amdgcn_wmma_bf16_16x16x16_bf16_tied_w32(
    v16short srcA,
    v16short srcB,
    v16short srcC,
    bool     opsel);
```

Register-tied variant of `__builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32`. The compiler constrains `srcC` and the return value to the same physical register, guaranteeing that the non-selected half of the accumulator (the half not written by this `opsel` setting) is preserved in place.

Use this variant when you chain two WMMA calls with opposite `opsel` values into the same `v16short` accumulator so that both halves remain live without an extra register copy.

Parameters, types, and return value are identical to the non-tied variant above.

15.1.1.8.4 INT32-accumulate builtins

Integer WMMA builtins accept either signed or unsigned 8-bit or 4-bit integer inputs, controlled by the `a_neg` and `b_neg` compile-time constants.

15.1.1.8.4.1 INT8 and UINT8 inputs

The following builtins use INT8 and UINT8 matrix inputs.

15.1.1.8.4.2 `__builtin_amdgcn_wmma_i32_16x16x16_iu8_w32`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_wmma_i32_16x16x16_iu8_w32(
    bool  a_neg,
    v4int srcA,
    bool  b_neg,
    v4int srcB,
    v8int srcC,
    bool  clamp);
```

Computes one step of a dense 16×16 INT32 accumulation with 8-bit integer inputs. Sixteen INT8 elements of $\boldsymbol{A}$ are packed into four `int` registers per lane; likewise for $\boldsymbol{B}$.

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT8, `false` for unsigned UINT8. Compile-time constant, see *Common parameters*.
`srcA`	v4int	Sixteen 8-bit elements of $\boldsymbol{A}$ per lane, packed into four 32-bit registers (four 8-bit values per register).
`b_neg`	bool	`true` for signed INT8, `false` for unsigned UINT8 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	v4int	Sixteen 8-bit elements of $\boldsymbol{B}$ per lane, packed into four 32-bit registers.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator (srcA × srcB + srcC).

15.1.1.8.4.3 INT4 and UINT4 inputs

The following builtins use INT4 and UINT4 matrix inputs.

15.1.1.8.4.4 `__builtin_amdgcn_wmma_i32_16x16x16_iu4_w32`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_wmma_i32_16x16x16_iu4_w32(
    bool  a_neg,
    v2int srcA,
    bool  b_neg,
    v2int srcB,
    v8int srcC,
    bool  clamp);
```

Computes one step of a dense 16 × 16 INT32 accumulation with 4-bit integer inputs. Sixteen INT4 elements of $\boldsymbol{A}$ are packed into two `int` registers per lane; likewise for $\boldsymbol{B}$.

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{A}$. Compile-time constant, see *Common parameters*.
`srcA`	v2int	Sixteen 4-bit elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (eight 4-bit values per register).
`b_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	v2int	Sixteen 4-bit elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator (srcA × srcB + srcC).

15.1.2 RDNA4 dense WMMA builtins

Wave-Matrix Multiply-Accumulate (WMMA) builtins let you issue hardware dense matrix multiply-accumulate operations directly from HIP device code on RDNA4 GPUs (`gfx1200`, `gfx1201`). Each WMMA instruction multiplies a dense $\boldsymbol{A}$ fragment by a dense $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single 32-wide wavefront.

RDNA4 supports the same FP16, BF16, INT8, and INT4 WMMA shapes as RDNA3 and adds FP8 (E4M3), BF8 (E5M2) input formats and a deeper-K INT4 variant (16x16x32). RDNA4 also provides sparse *sparse WMMA* variants that halve the $\boldsymbol{A}$ bandwidth using 2:4 structured sparsity.

CDNA (Instinct) GPUs provide a comparable dense operation through *Matrix Fused Multiply-Accumulate (MFMA)*. The two differ in wavefront size (wave32 for WMMA, wave64 for MFMA) and accumulator storage.

> **Note**
>
> RDNA4 WMMA builtins use a `_gfx12` suffix (for example, `__builtin_amdgcn_wmma_f32_16x16x16_f16_w32_gfx12`) to distinguish them from the RDNA3 builtins, which lack this suffix. The two generations have different fragment

sizes and accumulator layouts despite sharing the same tile dimensions.

Note

RDNA4 GPUs run all shader programs in `wave32` mode by default. The `_w32` suffix in each builtin name reflects this: all WMMA builtins on this page require `wavefrontsize32`.

15.1.2.1 Architecture availability

The builtins on this page target RDNA4 GPUs. To automatically enable them, pass the Low Level Virtual Machine (LLVM) target architecture flag at compile time:

```
amdclang++ --offload-arch=gfx1200 ...
amdclang++ --offload-arch=gfx1201 ...
```

Equivalent dense WMMA builtins for the previous generation are documented on the *RDNA3 dense WMMA builtins* page.

15.1.2.2 Naming convention

All dense WMMA builtins on this page follow the pattern:

```
__builtin_amdgcn_wmma_<out_type>_<M>x<N>x<K>_<in_type>[_<in_type_b>]_w32_gfx12
```

`out_type`
: Accumulator element type (`f32`, `f16`, `bf16`, or `i32`).

`M`, `N`, `K`
: Tile dimensions in elements. The instruction computes the contribution of a K-wide panel of $\boldsymbol{A}$ ($M \times K$) and a K-wide panel of $\boldsymbol{B}$ ($K \times N$) to an $M \times N$ output tile.

`in_type`
: Input element type of $\boldsymbol{A}$ (and $\boldsymbol{B}$ when both share the same type): `f16`, `bf16`, `iu8`, `iu4`, `fp8`, or `bf8`. The `iu` prefix means the builtin accepts either signed or unsigned integers, controlled by the `a_neg` and `b_neg` parameters.

`in_type_b` (optional)
: Input element type of $\boldsymbol{B}$ when it differs from $\boldsymbol{A}$. Used only for mixed FP8 and BF8 variants.

`_w32`
: Wavefront size suffix. All RDNA4 WMMA builtins use wave32.

`_gfx12`
: Architecture suffix distinguishing RDNA4 builtins from the RDNA3 variants.

15.1.2.3 Fragment layouts

All WMMA builtins on this page use a 16×16 output tile computed by one wave32 wavefront. The 32 lanes split into two groups of 16; each group owns a contiguous block of 8 output rows. This is a *lane-group-split* layout. Unlike RDNA3's matrix replication, RDNA4 splits the K dimension across lane groups: the two groups cover different K positions in the A and B fragments, interleaved in groups of 4 consecutive K values. The diagram and tables below show the mapping between matrix elements and lane or Vector General-Purpose Register (VGPR) positions for each operand.

15.1.2.3.1 Accumulator layout

Each lane holds 8 output elements across VGPRs 0–7. This layout is identical to the *SWMMAC accumulator* on the same hardware.

16×16 WMMA accumulator layout - wave32 (RDNA4)

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0–7 lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=8–15 lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Accumulator for all 16×16 WMMA intrinsics on RDNA4.
Cell value = VGPR index g (0–7). Column j = matrix column = lane % 16.
Formulas: lane = $\lfloor i/8 \rfloor \cdot 16 + j$; VGPR = $i \bmod 8$.

Given lane L and VGPR index g:

$$i = \lfloor \frac{L}{16} \rfloor \cdot 8 + g$$
$$j = L \bmod 16$$

Conversely, given output element (i, j):

$$\text{lane} = \lfloor \frac{i}{8} \rfloor \cdot 16 + j$$
$$\text{VGPR} = i \bmod 8$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0–7	0–15	0–7
8–15	16–31	0–7

> **Note**
>
> RDNA3 uses an interleaved row assignment (even rows in lanes 0–15, odd rows in lanes 16–31), while RDNA4 uses contiguous 8-row blocks. See *Fragment layouts* for the RDNA3 layout.

15.1.2.3.2 srcA and srcB (FP16 and BF16)

Each lane holds 8 input elements of $\boldsymbol{A}$ (`v8half`, 4 VGPRs × 2 FP16), covering one row of the 16×16 A fragment; similarly, each lane holds 8 elements of $\boldsymbol{B}$ covering one column of the 16×16 B fragment. For srcA, `lane % 16` gives the matrix row; for srcB, `lane % 16` gives the matrix column. The VGPR-to-K mapping is the same for both operands (transposed orientation).

Unlike RDNA3, where both lane groups carry identical copies of A and B (matrix replication), RDNA4 splits the K dimension across lane groups. The two groups cover different K positions in an interleaved pattern: groups of 4 consecutive K values alternate between lane groups. **Lane group 0 (lanes 0–15) covers K {0–3, 8–11}; lane group 1 (lanes 16–31) covers K {4–7, 12–15}**. Both groups cover all 16 rows (srcA) or columns (srcB).

The row and column-to-lane mapping:

Row (srcA) or Column (srcB)	Lanes	VGPRs
0	0, 16	0–3
1	1, 17	0–3
2	2, 18	0–3
3	3, 19	0–3
4	4, 20	0–3
5	5, 21	0–3
6	6, 22	0–3
7	7, 23	0–3
8	8, 24	0–3
9	9, 25	0–3
10	10, 26	0–3
11	11, 27	0–3
12	12, 28	0–3
13	13, 29	0–3
14	14, 30	0–3
15	15, 31	0–3

The 8 FP16 elements per lane are distributed across VGPRs:

Lane group	VGPR 0	VGPR 1	VGPR 2	VGPR 3
0 (lanes 0–15)	K {0, 1}	K {2, 3}	K {8, 9}	K {10, 11}
1 (lanes 16–31)	K {4, 5}	K {6, 7}	K {12, 13}	K {14, 15}

15.1.2.4 Using WMMA builtins as a compute policy

Using MFMA builtins as a compute policy explains the `ComputePolicy` pattern used to separate the multiply-accumulate logic from the rest of a kernel. A baseline policy is provided for RDNA4; a second, hardware-transpose policy that loads B via *a transpose-load builtin* instead of a scalar loop is also defined in the same example file and covered on that page.

The complete source file is available for download:

- `matrix_multiply_rdna4_wmma.hip`

15.1.2.4.1 Baseline policy

`WmmaRdna4F16Policy` uses scalar Local Data Share (LDS) loads for both A and B fragments. B is cooperatively transposed during the tile load (in the `TilePolicy`).

Each wavefront computes a single 16×16 output tile. The 32 lanes form 2 groups of 16 (`laneGroup = lane_id / 16`); each group covers 8 consecutive K-positions, so each lane supplies 8 FP16 values for both A and B.

Policy constants

`v_wmma_f32_16x16x16_f16` consumes 16 FP16 K positions per call, so `k_step = 16`. The wavefront holds the entire 16×16 tile: `thread_tile_m = thread_tile_n = 16` and `effective_lanes = 32`.

Accumulator layout

The builtin returns a `v8float` holding 8 VGPR values per lane. The `store_c()` pass maps `(lane, VGPR index)` back to (i, j) coordinates using the RDNA4 lane-group-split layout.

```
struct WmmaRdna4F16Policy
{
    // -- ComputePolicy constants ---------------------------------------------
    // One wave32 wavefront owns a 16×16 output tile.
    // All 32 lanes hold unique output elements (8 elements per lane).
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int frag_size_m     = 8;
    static constexpr int frag_size_n     = 8;
    // RDNA4 uses wave32: 32 lanes per wavefront (vs 64 on CDNA4).
    static constexpr int effective_lanes = 32;
    // wmma_f32_16x16x16_f16_w32 consumes 16 K-positions per call (k_step=16).
    // Each lane provides eight FP16 values (half8) for A and eight for B.
    // The 32 lanes split into 2 groups of 16; each group covers 8 K-positions.
    static constexpr int k_step          = 16;

    using elem_a = _Float16;
    using elem_b = _Float16;

    // -- Accumulator ---------------------------------------------------------
    // wmma_f8 holds 8 FP32 values per lane (8 output elements per lane in
    // the 16×16 tile, vs 4 per lane in CDNA4's MFMA with wave64).
    using wmma_f8 = float [[clang::ext_vector_type(8)]];

    struct Accumulator
    {
        wmma_f8 regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
        acc.regs = wmma_f8{};
    }

    // -- thread_tile_offset --------------------------------------------------
    // The entire 16×16 tile belongs to one wave32 wavefront.  Multiple
    // wavefronts in a block cover different 16×16 sub-tiles.
```

(continues on next page)

(continued from previous page)

```
//
//   wavefront id within block: wid = tid / 32  (wave32, not 64!)
//   waves_n = block_tile_n / thread_tile_n = 32 / 16 = 2
//   wid_x = wid % waves_n
//   wid_y = wid / waves_n
//   *thread_row = wid_y * 16
//   *thread_col = wid_x * 16
__device__ static void thread_tile_offset(int  tid,
                                          int  /*lane_id*/,
                                          int* thread_row,
                                          int* thread_col)
{
    constexpr int waves_n = 2; // block_tile_n / thread_tile_n = 32 / 16
    const int wid   = tid / 32; // wave32: divide by 32, not 64
    const int wid_x = wid % waves_n;
    const int wid_y = wid / waves_n;
    *thread_row = wid_y * 16;
    *thread_col = wid_x * 16;
}

// -- Fragment loads ------------------------------------------------------
// load_a: each lane reads eight FP16 values from LDS into a half8.
//
// wmma_f32_16x16x16_f16_w32 interprets the 32 lanes as 2 groups of 16
// (indexed by laneGroup = lane_id / 16).  Each group covers 8 consecutive
// K-positions: group g provides data for ki + 8*g through ki + 8*g + 7.
// Within a group, lane offset (lane_id % 16) selects one of the 16
// A-rows.
//
//   A-row index : tile_a_row + (lane_id mod 16)
//   K-position e: ki + 8*(lane_id / 16) + e   (e = 0..7)
//
// Note: the LDS tile is indexed as tile_a[k_local][m_local] (K rows,
// BlockTileM cols), so tile_a_ptr[k * k_tile_size + row] becomes
// tile_a_ptr[row * k_tile_size + k] when stored as tile_a[M][K].
// We use the M×K (row-major A) layout: tile_a[m_local][k_local].
__device__ static void load_a(const _Float16* tile_a_ptr,
                              int             tile_a_row,
                              int             ki,
                              int             k_tile_size,
                              int             lane_id,
                              elem_a        (&frag)[8])
{
    const int row   = tile_a_row + (lane_id % 16);
    const int k_off = ki + 8 * (lane_id / 16);
    #pragma unroll
    for(int e = 0; e < 8; ++e)
        frag[e] = tile_a_ptr[row * k_tile_size + k_off + e];
}

// load_b: each lane reads eight FP16 values from transposed LDS tile.
//
```

(continues on next page)

(continued from previous page)

```
    // tile_b_T is stored transposed: tile_b_T[n_local][k_local].
    //   B-col index : tile_b_col + (lane_id mod 16)
    //   K-position e: ki + 8*(lane_id / 16) + e   (e = 0..7)
    __device__ static void load_b(const _Float16* tile_b_T_ptr,
                                  int             tile_b_col,
                                  int             ki,
                                  int             k_tile_size,
                                  int             lane_id,
                                  elem_b        (&frag)[8])
    {
        const int col   = tile_b_col + (lane_id % 16);
        const int k_off = ki + 8 * (lane_id / 16);
        #pragma unroll
        for(int e = 0; e < 8; ++e)
            frag[e] = tile_b_T_ptr[col * k_tile_size + k_off + e];
    }

    // -- mma ---------------------------------------------------------------
    // Issue one wmma_f32_16x16x16_f16_w32_gfx12 instruction.
    __device__ static void mma(Accumulator&       acc,
                               const elem_a     (&a_frag)[8],
                               const elem_b     (&b_frag)[8])
    {
#if defined(__gfx1200__) || defined(__gfx1201__)
        using half8 = __attribute__((ext_vector_type(8))) __fp16;
        const half8 a_vec = {a_frag[0], a_frag[1], a_frag[2], a_frag[3],
                             a_frag[4], a_frag[5], a_frag[6], a_frag[7]};
        const half8 b_vec = {b_frag[0], b_frag[1], b_frag[2], b_frag[3],
                             b_frag[4], b_frag[5], b_frag[6], b_frag[7]};
        acc.regs = __builtin_amdgcn_wmma_f32_16x16x16_f16_w32_gfx12(
            a_vec, b_vec, acc.regs);
#endif
    }

    // -- store_c -----------------------------------------------------------
    // Scatter the 8 accumulator values to their global-memory positions.
    //
    // Inverse layout (lane L, accumulator index G in [0,7]):
    //   i = laneGroup * 8 + G    (laneGroup = L / 16)
    //   j = laneWrapped          (laneWrapped = L % 16)
    __device__ static void store_c(const Accumulator& acc,
                                   float*             C,
                                   int                out_row_base,
                                   int                out_col_base,
                                   int                m,
                                   int                n,
                                   int                lane_id)
    {
        const int laneWrapped = lane_id % 16;
        const int laneGroup   = lane_id / 16;

        #pragma unroll
```

(continues on next page)

(continued from previous page)

```
        for(int G = 0; G < 8; ++G)
        {
            const int i = laneGroup * 8 + G;
            const int j = laneWrapped;
            const int r = out_row_base + i;
            const int c = out_col_base + j;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[G];
        }
    }
};
```

Instantiating the kernel

Plug this policy into the generic kernel alongside any `TilePolicy` whose `block_tile_m` and `block_tile_n` are multiples of 16 and whose `k_tile_size` is a multiple of `k_step = 16`. The launch code below also instantiates `WmmaTilePolicyTrB`, the hardware-transpose tile policy explained on *RDNA4 WMMA transpose load builtins*, and runs both back to back for comparison.

```
// TilePolicy for the baseline FP16 path (row-major B, scalar-transposed into LDS).
using WmmaTilePolicy       = SingleBufferTilePolicyF16<32, 32, 16>;
// TilePolicy for the global_load_tr path (row-major B, hardware-transposed into LDS).
using WmmaTilePolicyTrB    = SingleBufferTilePolicyF16TrB<32, 32, 16>;
```

```
// Launch parameters:
//
// RDNA4 (gfx12xx): 2×2 wave32 wavefronts of 32 lanes = 128 threads per block.
//   block_tile_m = 32, thread_tile_m = 16  → 2 wavefronts along M
//   block_tile_n = 32, thread_tile_n = 16  → 2 wavefronts along N
//
// Fallback (scalar): 8×8 thread tiles in a 32×32 block → 16 threads per
//   block, padded to one full wave32 of 32.
//
// The kernel to launch and its thread count are selected at runtime based
// on the device architecture string returned by hipGetDeviceProperties.
//
// Grid: one block per 32×32 output tile.
constexpr int BLOCK_TILE_M = 32;
constexpr int BLOCK_TILE_N = 32;
```

```
    // Grid: one block per 32×32 output tile.
    const dim3 grid((N + BLOCK_TILE_N - 1) / BLOCK_TILE_N,
                    (M + BLOCK_TILE_M - 1) / BLOCK_TILE_M);

    float       ms = 0.0f;
    const char* policy_label = nullptr;

    if(is_gfx12)
    {
        // Baseline path: row-major B, scalar transpose into LDS.
        auto launch = [&]()
```

(continues on next page)

(continued from previous page)

```
        {
            matrix_multiply_generic<WmmaTilePolicy,
                                    WmmaRdna4F16Policy,
                                    _Float16>
                <<<grid, dim3(128)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<WmmaTilePolicy, WmmaRdna4F16Policy> [gfx12]";

        HIP_CHECK(hipMemcpy(h_C.data(), d_C, sizeof(float) * M * N,
                            hipMemcpyDeviceToHost));
        const bool ok_baseline = verify_result(h_C, h_A, M, N, K);
        print_metrics(policy_label, ok_baseline, ms, sizeof(_Float16), M, N, K);

        // global_load_tr path: row-major B, hardware transpose into LDS.
        HIP_CHECK(hipMemset(d_C, 0, sizeof(float) * M * N));
        auto launch_tr = [&]()
        {
            matrix_multiply_generic<WmmaTilePolicyTrB,
                                    WmmaRdna4F16TrPolicy,
                                    _Float16>
                <<<grid, dim3(128)>>>(d_A, d_B, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        const float ms_tr = time_kernel_ms(launch_tr, WARMUP_RUNS, TIMING_RUNS);
        HIP_CHECK(hipMemcpy(h_C.data(), d_C, sizeof(float) * M * N,
                            hipMemcpyDeviceToHost));
        const bool ok_tr = verify_result(h_C, h_A, M, N, K);
        print_metrics(
            "GemmKernel<WmmaTilePolicyTrB, WmmaRdna4F16TrPolicy> [gfx12]",
            ok_tr, ms_tr, sizeof(_Float16), M, N, K);
    }
    else
    {
        // Scalar FP32 fallback: upload A and B as FP32.
        std::vector<float> h_A_f32(M * K);
        for(int i = 0; i < M * K; ++i)
            h_A_f32[i] = static_cast<float>(h_A[i]);

        std::vector<float> h_B_f32(K * N, 0.0f);
        for(int i = 0; i < std::min(K, N); ++i)
            h_B_f32[i * N + i] = 1.0f;

        float* d_A_f32;
        float* d_B_f32;
        HIP_CHECK(hipMalloc(&d_A_f32, sizeof(float) * M * K));
        HIP_CHECK(hipMalloc(&d_B_f32, sizeof(float) * K * N));
        HIP_CHECK(hipMemcpy(d_A_f32, h_A_f32.data(), sizeof(float) * M * K,
                            hipMemcpyHostToDevice));
        HIP_CHECK(hipMemcpy(d_B_f32, h_B_f32.data(), sizeof(float) * K * N,
                            hipMemcpyHostToDevice));
```

(continues on next page)

(continued from previous page)

```
        auto launch = [&]()
        {
            matrix_multiply_generic<SingleBufferTilePolicyF<32, 32, 8>,
                                    ScalarFMAPolicy<8, 8, 4>,
                                    float>
                <<<grid, dim3(32)>>>(d_A_f32, d_B_f32, d_C, M, N, K);
            HIP_CHECK(hipGetLastError());
        };
        ms           = time_kernel_ms(launch, WARMUP_RUNS, TIMING_RUNS);
        policy_label = "GemmKernel<SingleBufferTilePolicyF, ScalarFMAPolicy>␣
↪[fallback]";

        HIP_CHECK(hipMemcpy(h_C.data(), d_C, sizeof(float) * M * N,
                            hipMemcpyDeviceToHost));
        const bool ok = verify_result(h_C, h_A, M, N, K);
        print_metrics(policy_label, ok, ms, sizeof(_Float16), M, N, K);

        HIP_CHECK(hipFree(d_A_f32));
        HIP_CHECK(hipFree(d_B_f32));
    }
```

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx1200 \
    matrix_multiply_rdna4_wmma.hip -o mm_rdna4_wmma
./mm_rdna4_wmma

amdclang++ -O3 -std=c++17 --offload-arch=gfx1201 \
    matrix_multiply_rdna4_wmma.hip -o mm_rdna4_wmma
./mm_rdna4_wmma
```

> **Note**
>
> `WmmaRdna4F16Policy` and `WmmaRdna4F16TrPolicy` require an RDNA4 GPU (`gfx1200` or `gfx1201`). The `#if defined(__gfx1200__) || defined(__gfx1201__)` guard in the example file falls back to `ScalarFMAPolicy` on other targets, so the file compiles without modification.

15.1.2.5 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v8half   = _Float16 [[clang::ext_vector_type(8)]];
using v8float  = float [[clang::ext_vector_type(8)]];
using v8short  = short [[clang::ext_vector_type(8)]];
using v8int    = int [[clang::ext_vector_type(8)]];
using v4int    = int [[clang::ext_vector_type(4)]];
using v2int    = int [[clang::ext_vector_type(2)]];
```

15.1.2.6 Common parameters

The `a_neg`, `b_neg`, and `clamp` parameters appear only on integer WMMA variants.

Parameter	Type	Description
`a_neg`	`bool` (compile-time constant)	Integer variants only. When `true`, the $\boldsymbol{A}$ elements are treated as signed integers; when `false`, as unsigned.
`b_neg`	`bool` (compile-time constant)	Integer variants only. Same as `a_neg` but for $\boldsymbol{B}$.
`clamp`	`bool` (compile-time constant)	Integer variants only. When `true`, the INT32 accumulator output is clamped to the representable range of the input type on overflow.

15.1.2.7 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency.

Builtin	Ops	Cycle count
`__builtin_amdgcn_wmma_f32_16x16x16_f16_w32_gfx12`	8192	16
`__builtin_amdgcn_wmma_f32_16x16x16_bf16_w32_gfx12`	8192	16
`__builtin_amdgcn_wmma_f16_16x16x16_f16_w32_gfx12`	8192	16
`__builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32_gfx12`	8192	16
`__builtin_amdgcn_wmma_i32_16x16x16_iu8_w32_gfx12`	8192	8
`__builtin_amdgcn_wmma_i32_16x16x16_iu4_w32_gfx12`	8192	8
`__builtin_amdgcn_wmma_i32_16x16x32_iu4_w32_gfx12`	16384	8
`__builtin_amdgcn_wmma_f32_16x16x16_fp8_fp8_w32_gfx12`	8192	8
`__builtin_amdgcn_wmma_f32_16x16x16_fp8_bf8_w32_gfx12`	8192	8
`__builtin_amdgcn_wmma_f32_16x16x16_bf8_fp8_w32_gfx12`	8192	8
`__builtin_amdgcn_wmma_f32_16x16x16_bf8_bf8_w32_gfx12`	8192	8

15.1.2.8 Builtin reference

The following sections list every dense WMMA builtin available on RDNA4, grouped by accumulator type.

15.1.2.8.1 FP32-accumulate builtins

These builtins accumulate into FP32 and accept FP16, BF16, FP8, or BF8 matrix inputs.

15.1.2.8.1.1 FP16 inputs

The following builtins use FP16 matrix inputs.

15.1.2.8.1.2 `__builtin_amdgcn_wmma_f32_16x16x16_f16_w32_gfx12`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_f16_w32_gfx12(
    v8half  srcA,
    v8half  srcB,
    v8float srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with FP16 inputs. Each lane supplies 8 FP16 values for both $\boldsymbol{A}$ and $\boldsymbol{B}$. The 32 lanes form two groups of 16; each group covers 8 of the 16 K-positions.

Parameter	Type	Description
`srcA`	v8half	Eight FP16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v8half	Eight FP16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.1.3 BF16 inputs

The following builtins use BF16 matrix inputs.

15.1.2.8.1.4 `__builtin_amdgcn_wmma_f32_16x16x16_bf16_w32_gfx12`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_bf16_w32_gfx12(
    v8short srcA,
    v8short srcB,
    v8float srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with BF16 inputs. BF16 values are passed as `short` (16-bit storage). Otherwise identical in structure to the FP16 variant.

Parameter	Type	Description
`srcA`	v8short	Eight BF16 elements of $\boldsymbol{A}$ per lane (BF16 stored as `short`).
`srcB`	v8short	Eight BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.1.5 FP8 and BF8 inputs

The following builtins use FP8 and BF8 matrix inputs.

15.1.2.8.1.6 `__builtin_amdgcn_wmma_f32_16x16x16_fp8_fp8_w32_gfx12`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_fp8_fp8_w32_gfx12(
    v2int   srcA,
    v2int   srcB,
    v8float srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with FP8 (E4M3) inputs for both $\boldsymbol{A}$ and $\boldsymbol{B}$. Eight FP8 values are packed into two 32-bit registers per lane (four 8-bit values per register).

Parameter	Type	Description
`srcA`	v2int	Eight FP8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v2int	Eight FP8 elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.1.7 `__builtin_amdgcn_wmma_f32_16x16x16_fp8_bf8_w32_gfx12`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_fp8_bf8_w32_gfx12(
    v2int   srcA,
    v2int   srcB,
    v8float srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with mixed 8-bit floating-point inputs: FP8 (E4M3) for $\boldsymbol{A}$ and BF8 (E5M2) for $\boldsymbol{B}$. Eight elements per lane are packed into two 32-bit registers.

Parameter	Type	Description
`srcA`	v2int	Eight FP8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v2int	Eight BF8 elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.1.8 `__builtin_amdgcn_wmma_f32_16x16x16_bf8_fp8_w32_gfx12`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_bf8_fp8_w32_gfx12(
    v2int   srcA,
    v2int   srcB,
    v8float srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with mixed 8-bit floating-point inputs: BF8 (E5M2) for $\boldsymbol{A}$ and FP8 (E4M3) for $\boldsymbol{B}$. Eight elements per lane are packed into two 32-bit registers.

Parameter	Type	Description
`srcA`	v2int	Eight BF8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v2int	Eight FP8 elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.1.9 `__builtin_amdgcn_wmma_f32_16x16x16_bf8_bf8_w32_gfx12`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_wmma_f32_16x16x16_bf8_bf8_w32_gfx12(
    v2int   srcA,
    v2int   srcB,
    v8float srcC);
```

Computes one step of a dense 16×16 FP32 accumulation with BF8 (E5M2) inputs for both $\boldsymbol{A}$ and $\boldsymbol{B}$. Eight BF8 values are packed into two 32-bit registers per lane (four 8-bit values per register).

Parameter	Type	Description
`srcA`	v2int	Eight BF8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v2int	Eight BF8 elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.

Returns `v8float` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.2 FP16-accumulate builtins

This builtin accumulates into FP16 with FP16 inputs.

15.1.2.8.2.1 FP16 inputs

The following builtin uses FP16 matrix inputs.

15.1.2.8.2.2 `__builtin_amdgcn_wmma_f16_16x16x16_f16_w32_gfx12`

Signature and parameters for this builtin.

```
v8half __builtin_amdgcn_wmma_f16_16x16x16_f16_w32_gfx12(
    v8half srcA,
    v8half srcB,
    v8half srcC);
```

Computes one step of a dense 16×16 FP16 accumulation. Both inputs and the accumulator are FP16.

Parameter	Type	Description
`srcA`	v8half	Eight FP16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v8half	Eight FP16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8half	Accumulator input: eight FP16 elements per lane.

Returns `v8half` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.3 BF16-accumulate builtins

This builtin accumulates into BF16 with BF16 inputs.

15.1.2.8.3.1 BF16 inputs

The following builtin uses BF16 matrix inputs.

15.1.2.8.3.2 `__builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32_gfx12`

Signature and parameters for this builtin.

```
v8short __builtin_amdgcn_wmma_bf16_16x16x16_bf16_w32_gfx12(
    v8short srcA,
    v8short srcB,
    v8short srcC);
```

Computes one step of a dense 16×16 BF16 accumulation. Both inputs and the accumulator are BF16 (stored as `short`).

Parameter	Type	Description
`srcA`	v8short	Eight BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v8short	Eight BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8short	Accumulator input: eight BF16 elements per lane (stored as `short`).

Returns `v8short` – updated accumulator (BF16 stored as `short`) (srcA $\times$ srcB $+$ srcC).

15.1.2.8.4 INT32-accumulate builtins

Integer WMMA builtins accept either signed or unsigned 8-bit or 4-bit integer inputs, controlled by the `a_neg` and `b_neg` compile-time constants.

15.1.2.8.4.1 INT8 and UINT8 inputs (16x16x16)

The following builtins use INT8 and UINT8 matrix inputs.

15.1.2.8.4.2 `__builtin_amdgcn_wmma_i32_16x16x16_iu8_w32_gfx12`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_wmma_i32_16x16x16_iu8_w32_gfx12(
    bool  a_neg,
    v2int srcA,
    bool  b_neg,
    v2int srcB,
    v8int srcC,
    bool  clamp);
```

Computes one step of a dense 16×16 INT32 accumulation with 8-bit integer inputs. Sixteen INT8 elements of $\boldsymbol{A}$ are packed into two `int` registers per lane (four 8-bit values per register); likewise for $\boldsymbol{B}$.

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT8, `false` for unsigned UINT8. Compile-time constant, see *Common parameters*.
`srcA`	v2int	Sixteen 8-bit elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers.
`b_neg`	bool	`true` for signed INT8, `false` for unsigned UINT8 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	v2int	Sixteen 8-bit elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.4.3 INT4 and UINT4 inputs (16x16x16)

The following builtins use INT4 and UINT4 matrix inputs.

15.1.2.8.4.4 `__builtin_amdgcn_wmma_i32_16x16x16_iu4_w32_gfx12`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_wmma_i32_16x16x16_iu4_w32_gfx12(
    bool  a_neg,
    int   srcA,
    bool  b_neg,
    int   srcB,
    v8int srcC,
    bool  clamp);
```

Computes one step of a dense 16 $\times$ 16 INT32 accumulation with 4-bit integer inputs. Sixteen INT4 elements of $\boldsymbol{A}$ fit into a single `int` per lane; likewise for $\boldsymbol{B}$.

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{A}$. Compile-time constant, see *Common parameters*.
`srcA`	int	Sixteen 4-bit elements of $\boldsymbol{A}$ per lane, packed into one 32-bit register.
`b_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	int	Sixteen 4-bit elements of $\boldsymbol{B}$ per lane, packed into one 32-bit register.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator (srcA $\times$ srcB + srcC).

15.1.2.8.4.5 INT4 and UINT4 inputs (16x16x32)

The following builtins use INT4 and UINT4 matrix inputs.

15.1.2.8.4.6 `__builtin_amdgcn_wmma_i32_16x16x32_iu4_w32_gfx12`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_wmma_i32_16x16x32_iu4_w32_gfx12(
    bool  a_neg,
    v2int srcA,
    bool  b_neg,
    v2int srcB,
    v8int srcC,
    bool  clamp);
```

Computes one step of a dense 16×16 INT32 accumulation with 4-bit integer inputs over a deeper K=32 strip. Thirty-two INT4 elements of $\boldsymbol{A}$ are packed into two `int` registers per lane; likewise for $\boldsymbol{B}$.

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{A}$. Compile-time constant, see *Common parameters*.
`srcA`	v2int	Thirty-two 4-bit elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (sixteen 4-bit values per register).
`b_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	v2int	Thirty-two 4-bit elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator (srcA $\times$ srcB $+$ srcC).

15.2 RDNA4 sparse WMMA builtins

Sparse Wave-Matrix Multiply-Accumulate (SWMMAC) builtins let you issue hardware sparse matrix multiply-accumulate operations directly from HIP device code on RDNA4 GPUs (`gfx1200`, `gfx1201`). Each SWMMAC instruction multiplies a compressed sparse $\boldsymbol{A}$ fragment by a dense $\boldsymbol{B}$ fragment and accumulates the result into a $\boldsymbol{C}$ fragment, all within a single 32-wide wavefront. Because the $\boldsymbol{A}$ operand is stored in compressed form, SWMMAC halves the storage and bandwidth required for $\boldsymbol{A}$ relative to a dense multiply of the same tile size.

SWMMAC is the sparse variant of the WMMA instruction family, which is used on RDNA (consumer) GPUs. CDNA (Instinct) GPUs provide a comparable operation through *Sparse Matrix Fused Multiply-Accumulate (SMFMAC)*, the sparse variant of Matrix Fused Multiply-Accumulate (MFMA). The two differ in wavefront size (wave32 for SWMMAC, wave64 for SMFMAC) and accumulator storage (ordinary Vector General-Purpose Registers (VGPRs) for SWMMAC, dedicated accVGPRs for SMFMAC). The underlying sparsity model (2:4 structured sparsity on $\boldsymbol{A}$) is the same on both architectures.

> **Note**
>
> RDNA4 GPUs run all shader programs in `wave32` mode by default. The `_w32` suffix in each builtin name reflects this: all SWMMAC builtins on this page require `wavefrontsize32`. `wavefrontsize64` is not supported for HIP code.

15.2.1 Architecture availability

The builtins on this page target RDNA4 GPUs. To automatically enable them, pass the Low Level Virtual Machine (LLVM) target architecture flag at compile time:

```
amdclang++ --offload-arch=gfx1201 ...
amdclang++ --offload-arch=gfx1200 ...
```

15.2.2 Naming convention

All SWMMAC builtins follow the pattern:

```
__builtin_amdgcn_swmmac_<out_type>_<M>x<N>x<K>_<in_type>[_<in_type_b>]_w32
```

`out_type`
Accumulator element type (`f32`, `f16`, `bf16`, or `i32`).

`M`, `N`, `K`
Tile dimensions. The K dimension refers to the *compressed* K of the sparse $\boldsymbol{A}$ operand. The corresponding dense K is twice as large due to 2:4 sparsity.

`in_type`
Input element type of $\boldsymbol{A}$ (and $\boldsymbol{B}$ when both share the same type): `f16`, `bf16`, `fp8`, `bf8`, `iu8`, or `iu4`. The `iu` prefix means the builtin accepts either signed or unsigned integers, controlled by the `a_neg` and `b_neg` parameters.

`in_type_b` (optional)
Input element type of $\boldsymbol{B}$ when it differs from $\boldsymbol{A}$. Used only for mixed FP8 and BF8 variants.

`_w32`
Wavefront size suffix. All RDNA4 SWMMAC builtins use wave32.

15.2.3 Structured sparsity (2:4 pattern)

SWMMAC instructions require the $\boldsymbol{A}$ matrix to obey 2:4 structured sparsity: in every contiguous group of four elements along the K dimension, exactly two are non-zero and the other two are zero. This constraint allows the $\boldsymbol{A}$ operand to be stored in a compressed representation containing only the non-zero values, reducing the storage to half the original K dimension.

Along with the compressed non-zero values, the hardware requires a sparsity index that encodes which two of the four positions in each group hold the non-zeros. This index is passed as the `index` argument to every SWMMAC builtin. The hardware uses it at execution time to align the compressed $\boldsymbol{A}$ elements against the correct rows of the $\boldsymbol{B}$ operand before accumulating the products.

Host-side compression is straightforward: walk the K dimension in groups of four, extract the two non-zero positions, and pack them consecutively into the output buffer. The example kernel in this topic uses a fixed pattern that keeps positions 0 and 2 of every group, producing a compressed buffer of half the original K length.

15.2.4 Fragment layouts

All SWMMAC builtins on this page use a 16×16 output tile computed by one wave32 wavefront. The 32 lanes split into two groups of 16; each group owns half the output rows. The diagrams below show the mapping between matrix elements and lane or VGPR positions for each operand.

The formulas in the subsections below use the following notation:

- i – zero-based row index within the tile, $0 \leq i < 16$
- j – zero-based column index within the tile, $0 \leq j < 16$

- **lane** – wavefront lane, $0 \leq \text{lane} < 32$
- **VGPR** – zero-based index into that lane's register vector

15.2.4.1 Accumulator layout

Each lane holds 8 FP32 output elements across VGPRs 0–7.

16×16 SWMMAC accumulator layout - wave32

i \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0–7 lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
i=8–15 lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

Accumulator for all 16×16 SWMMAC intrinsics on RDNA4.
Cell value = VGPR index g (0–7). Column j = matrix column = lane % 16.
Formulas: lane = $\lfloor i/8 \rfloor \cdot 16 + j$; VGPR = $i \bmod 8$.

Given output element (i, j):

$$\text{lane} = \lfloor \frac{i}{8} \rfloor \cdot 16 + j$$
$$\text{VGPR} = i \bmod 8$$

Conversely, given lane L and VGPR index g:

$$i = \lfloor \frac{L}{16} \rfloor \cdot 8 + g$$
$$j = L \bmod 16$$

The row-to-lane mapping:

Rows	Lanes	VGPRs
0–7	0–15	0–7
8–15	16–31	0–7

15.2.4.2 srcA (sparse, FP16 and BF16)

Each lane holds 8 compressed FP16 values (`v8fp16`, 4 VGPRs × 2 FP16) covering one row of the sparse $\boldsymbol{A}$ matrix. The compressed-K positions are non-contiguous across the two lane groups.

16×16×32 SWMMAC srcA layout - wave32 (compressed K, FP16)

lane grp	lanes 0–15				lanes 16–31				lanes 0–15				lanes 16–31			
c	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i=0	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
i=4	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
i=8	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
i=12	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3
	0	0	1	1	0	0	1	1	2	2	3	3	2	2	3	3

srcA sparse fragment for 16×16×32 SWMMAC (FP16/BF16 inputs), RDNA4.
Cell value = VGPR index (0–3); each VGPR holds 2 packed FP16 values.
Teal columns = lane group 0 (lanes 0–15); grey = lane group 1 (lanes 16–31).
Both groups cover all 16 rows (row = lane % 16). Column c = compressed-K index.

Lane L covers matrix row L mod 16. The 8 compressed elements are distributed across VGPRs as follows:

Lane group	VGPR 0	VGPR 1	VGPR 2	VGPR 3
0 (lanes 0–15)	compressed K {0, 1}	compressed K {2, 3}	compressed K {8, 9}	compressed K {10, 11}
1 (lanes 16–31)	compressed K {4, 5}	compressed K {6, 7}	compressed K {12, 13}	compressed K {14, 15}

15.2.4.3 srcB (dense, FP16 and BF16)

Each lane holds 16 dense FP16 values (`v16fp16`, 8 VGPRs × 2 FP16) covering one column of the dense $\boldsymbol{B}$ matrix. The K-row positions are non-contiguous across the two lane groups.

Lane L covers matrix column L mod 16. The 16 dense K rows are distributed across VGPRs as follows:

Lane group	VGPRs 0–3	VGPRs 4–7
0 (lanes 0–15)	K rows 0–7	K rows 16–23
1 (lanes 16–31)	K rows 8–15	K rows 24–31

16×16×32 SWMMAC srcB layout - wave32 (dense K, FP16)

k \ j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
k=0–7 V0–3 lanes 0–15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
k=8–15 V0–3 lanes 16–31	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
k=16–23 V4–7 lanes 0–15	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
k=24–31 V4–7 lanes 16–31	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

srcB dense fragment for 16×16×32 SWMMAC (FP16/BF16 inputs), RDNA4.
Cell value = VGPR index (0–7); each VGPR holds 2 packed FP16 values.
Teal rows = lane group 0 (lanes 0–15); grey = lane group 1 (lanes 16–31).
Both groups cover all 16 columns (col = lane % 16). Row k = dense-K index.

15.2.5 Register types used in this reference

The signatures below use the following type aliases, which you can declare with C++ attributes in any HIP translation unit:

```
using v2int    = int    [[clang::ext_vector_type(2)]];
using v4int    = int    [[clang::ext_vector_type(4)]];
using v8int    = int    [[clang::ext_vector_type(8)]];
using v8float  = float  [[clang::ext_vector_type(8)]];
using v8short  = short  [[clang::ext_vector_type(8)]];   // BF16 storage
using v16short = short  [[clang::ext_vector_type(16)]];  // BF16 storage
using v8fp16   = __fp16 [[clang::ext_vector_type(8)]];
using v16fp16  = __fp16 [[clang::ext_vector_type(16)]];
```

Each type alias maps one-to-one to the corresponding LLVM vector type used in the builtin definition. The number in the name is the element count per lane.

Note

`__fp16` and `_Float16` are distinct Clang types. `__fp16` is a storage-only type: arithmetic on `__fp16` values promotes to `float` before the operation. `_Float16` is the C standard FP16 arithmetic type that supports native half-precision operations without promotion. RDNA4 SWMMAC builtins use `__fp16`; CDNA MFMA builtins use `_Float16`. The two types are not implicitly convertible, so cast explicitly when sharing FP16 data between code paths that target different architectures.

Note

BF16 matrix inputs use `short` as the storage type for RDNA4 SWMMAC (not `__bf16`). Reinterpret your BF16 data with `__builtin_bit_cast` or a union before passing it to the builtin.

15.2.6 Common parameters

The `index` parameter is shared by all SWMMAC builtins. The `a_neg`, `b_neg`, and `clamp` parameters appear only on integer variants.

Parameter	Type	Description
index	int	Sparsity index register. Each pair of bits encodes the position (0–3) of one non-zero element within its block of four consecutive $\boldsymbol{A}$ elements along K. The 32-bit int holds indices for all elements owned by one lane. Must satisfy the 2:4 constraint: exactly two of the four elements in each block must be selected.
a_ne	bool (compile-time constant)	Integer variants only. When true, the $\boldsymbol{A}$ elements are treated as signed integers; when false, as unsigned.
b_ne	bool (compile-time constant)	Integer variants only. Same as a_neg but for $\boldsymbol{B}$.
clam	bool (compile-time constant)	Integer variants only. When true, the INT32 accumulator output is clamped to the representable range of the input type on overflow.

15.2.7 Using sparse WMMA builtins as a compute policy

Using MFMA builtins as a compute policy explains the ComputePolicy pattern used to separate the multiply-accumulate logic from the rest of a kernel. The example below implements SwmmacRdna4F16Policy using __builtin_amdgcn_swmmac_f32_16x16x32_f16_w32.

Each wavefront computes a single 16×16 output tile. The $\boldsymbol{A}$ operand is pre-sparsified: half the K positions are zero and omitted from storage, so the compressed K dimension is 16 (representing 32 dense K positions).

The complete source file is available for download:

- matrix_multiply_rdna4_swmmac.hip

Policy constants

swmmac_f32_16x16x32_f16_w32 consumes 32 dense K positions per call (compressed to K=16 in $\boldsymbol{A}$), so k_step = 32. The wavefront holds the entire 16×16 tile: thread_tile_m = thread_tile_n = 16 and effective_lanes = 32.

Sparsity index

Each lane's index register encodes two bits per compressed K position, identifying which of the four elements in each 2:4 block is non-zero. The example constructs an even-column sparsity pattern (elements at positions 0 and 2 in each block of four) at the host and passes it to the device.

Accumulator layout

The builtin returns a v8float holding 8 FP32 values per lane. The store_c() pass maps (lane, VGPR index) back to (i, j) coordinates using the SWMMAC accumulator layout.

```
// Vector type aliases required by the intrinsic.
// Use the __attribute__ form; [[clang::ext_vector_type]] is not supported
// for __fp16 in HIP device code.
```

(continues on next page)

(continued from previous page)

```
typedef __fp16 v8fp16  __attribute__((ext_vector_type(8)));
typedef __fp16 v16fp16 __attribute__((ext_vector_type(16)));
typedef float  v8float __attribute__((ext_vector_type(8)));

struct SwmmacRdna4F16Policy
{
    // -- ComputePolicy constants ------------------------------------------
    // One wave32 wavefront owns a 16×16 output tile.
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int effective_lanes = 32;
    // swmmac_f32_16x16x32_f16_w32 processes 32 dense K-positions per call.
    static constexpr int k_step          = 32;

    using elem_a = __fp16;
    using elem_b = __fp16;

    // -- Accumulator ------------------------------------------------------
    // v8float holds 8 FP32 output elements per lane.
    struct Accumulator
    {
        v8float regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
        acc.regs = v8float{};
    }

    // -- thread_tile_offset -----------------------------------------------
    // The entire 16×16 tile belongs to one wavefront. Multiple wavefronts in
    // a block cover different 16×16 sub-tiles.
    //
    //   wavefront id within block: wid = tid / 32
    //   waves_n = block_tile_n / thread_tile_n
    //   wid_x = wid % waves_n
    //   wid_y = wid / waves_n
    __device__ static void thread_tile_offset(int  tid,
                                              int  /*lane_id*/,
                                              int* thread_row,
                                              int* thread_col)
    {
        constexpr int waves_n = 2; // block_tile_n / thread_tile_n = 32 / 16
        const int wid   = tid / 32;
        const int wid_x = wid % waves_n;
        const int wid_y = wid / waves_n;
        *thread_row = wid_y * 16;
        *thread_col = wid_x * 16;
    }

    // -- Fragment loads ---------------------------------------------------
    // load_a: each lane reads 8 compressed FP16 values and one index int from
```

(continues on next page)

(continued from previous page)

```
    // LDS.
    //
    // The SWMMAC instruction requires srcA as v8fp16 (8 compressed elements)
    // and srcB as v16fp16 (16 dense elements). The index register is a scalar
    // int containing 16 two-bit position codes (bits[15:0] used).
    //
    // Lane L covers A-row = tile_a_row + L % 16. The 8 compressed elements
    // are loaded from non-contiguous positions within the row (AMD Matrix
    // Instruction Calculator layout for v_swmmac_f32_16x16x32_f16_w32):
    //   Lanes 0-15:  elements[0-3] = compressed k {0-3}   (dense k  0- 7)
    //                elements[4-7] = compressed k {8-11}  (dense k 16-23)
    //   Lanes 16-31: elements[0-3] = compressed k {4-7}   (dense k  8-15)
    //                elements[4-7] = compressed k {12-15} (dense k 24-31)

    // Compressed A fragment: 8 FP16 elements per lane.
    // Stores both the FP16 data and the matching index int.
    struct AFragment
    {
        v8fp16 data;
        int    index;
    };

    __device__ static AFragment load_a_frag(const __fp16* tile_a_ptr,
                                            const int*    tile_idx_ptr,
                                            int           tile_a_row,
                                            int           k_tile_sparse,
                                            int           lane_id)
    {
        AFragment frag;
        const int row       = tile_a_row + (lane_id % 16);
        const int lane_grp  = lane_id / 16; // 0 for lanes 0-15, 1 for lanes 16-31
        // Lower 4 compressed elements: lanes 0-15 use k {0-3}, lanes 16-31 use k {4-
↪7}.
        const int k_lo      = lane_grp * 4;
        // Upper 4 compressed elements: lanes 0-15 use k {8-11}, lanes 16-31 use k
↪{12-15}.
        const int k_hi      = 8 + lane_grp * 4;
        const __fp16* row_ptr = tile_a_ptr + row * k_tile_sparse;
        #pragma unroll
        for(int e = 0; e < 4; ++e)
            frag.data[e]     = row_ptr[k_lo + e];
        #pragma unroll
        for(int e = 0; e < 4; ++e)
            frag.data[4 + e] = row_ptr[k_hi + e];
        // One int per row per K-tile (bits[15:0] = 4 × 4-bit groups).
        // All lanes in the same output row load the same int; hardware uses
        // lane_id / 16 to select the appropriate 8-bit half.
        frag.index = tile_idx_ptr[row * (k_tile_sparse / 16)];
        return frag;
    }

    // Dense B fragment: 16 FP16 elements per lane.
```

(continues on next page)

(continued from previous page)

```
// tile_b_T is stored N-major: tile_b_T[n][k], stride = k_tile_dense (=32).
//
// SWMMAC srcB layout (wave32, 16x16x32, AMD Matrix Instruction Calculator):
//   Lane L covers one B-column: n = tile_b_col + L % 16.
//   The 16 k-positions are non-contiguous within that column:
//     Lanes 0-15:  elements[0-7] = B[0..7][n]   (k  0- 7)
//                  elements[8-15]= B[16..23][n]  (k 16-23)
//     Lanes 16-31: elements[0-7] = B[8..15][n]  (k  8-15)
//                  elements[8-15]= B[24..31][n]  (k 24-31)
__device__ static v16fp16 load_b_frag(const __fp16* tile_b_T_ptr,
                                      int           tile_b_col,
                                      int           k_tile_dense,
                                      int           lane_id)
{
    v16fp16 frag;
    const int n        = tile_b_col + (lane_id % 16);
    const int lane_grp = lane_id / 16; // 0 for lanes 0-15, 1 for lanes 16-31
    // Lower 8 k-positions: lanes 0-15 use k {0-7}, lanes 16-31 use k {8-15}.
    const int k_lo     = lane_grp * 8;
    // Upper 8 k-positions: lanes 0-15 use k {16-23}, lanes 16-31 use k {24-31}.
    const int k_hi     = 16 + lane_grp * 8;
    const __fp16* col_ptr = tile_b_T_ptr + n * k_tile_dense;
    #pragma unroll
    for(int e = 0; e < 8; ++e)
        frag[e]     = col_ptr[k_lo + e];
    #pragma unroll
    for(int e = 0; e < 8; ++e)
        frag[8 + e] = col_ptr[k_hi + e];
    return frag;
}

// -- mma ---------------------------------------------------------------------
// Issue one swmmac_f32_16x16x32_f16_w32 instruction.
__device__ static void mma(Accumulator&      acc,
                           const AFragment&  a_frag,
                           const v16fp16&    b_frag)
{
    acc.regs = __builtin_amdgcn_swmmac_f32_16x16x32_f16_w32(
        a_frag.data, b_frag, acc.regs, a_frag.index);
}

// -- store_c -----------------------------------------------------------------
// Scatter the 8 accumulator values to their global-memory positions.
//
// SWMMAC 16x16x32 wave32 output layout (AMD Matrix Instruction Calculator):
//   Lane L, VGPR g (0-7):
//     row = (L / 16) * 8 + g
//     col = L % 16
__device__ static void store_c(const Accumulator& acc,
                               float*             C,
                               int                out_row_base,
                               int                out_col_base,
```

(continues on next page)

(continued from previous page)

```
                                int                 m,
                                int                 n,
                                int                 lane_id)
    {
        const int col = lane_id % 16;

        #pragma unroll
        for(int g = 0; g < 8; ++g)
        {
            const int r = out_row_base + (lane_id / 16) * 8 + g;
            const int c = out_col_base + col;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[g];
        }
    }
};
```

Instantiating the kernel

With `SwmmacRdna4F16Policy` in place, plug it into the generic kernel alongside a `TilePolicy` (see *Using MFMA builtins as a compute policy*) whose `block_tile_m` and `block_tile_n` are multiples of 16 and whose `k_tile_size` is a multiple of `k_step = 32`.

```
// TilePolicy: 32×32 block tile, 16-element sparse K-strip (= 32 dense).
// SwmmacRdna4F16Policy: one 16×16 tile per wave32 wavefront.
using SwmmacTilePolicy = SingleBufferTilePolicyH<32, 32, 16>;
```

```
// Block tile: 32×32 (2×2 wavefronts, each computing a 16×16 SWMMAC tile).
// Threads per block: 2 × 2 wavefronts × 32 lanes = 128.
constexpr int BLOCK_TILE_M      = 32;
constexpr int BLOCK_TILE_N      = 32;
constexpr int THREADS_PER_BLOCK = 128; // 4 wave32 wavefronts × 32 lanes
```

```
    // Grid: one block per 32×32 output tile.
    const dim3 grid((N + BLOCK_TILE_N - 1) / BLOCK_TILE_N,
                    (M + BLOCK_TILE_M - 1) / BLOCK_TILE_M);
    const dim3 block(THREADS_PER_BLOCK);

    auto launch = [&]()
    {
        matrix_multiply_swmmac<SwmmacTilePolicy>
            <<<grid, block>>>(d_A_sparse, reinterpret_cast<const int*>(d_A_index),
                              d_B, d_C, M, N, K_SPARSE);
        HIP_CHECK(hipGetLastError());
    };
```

Compile and run:

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx1201 \
    matrix_multiply_rdna4_swmmac.hip -o mm_rdna4_swmmac
./mm_rdna4_swmmac
```

(continues on next page)

(continued from previous page)

```
amdclang++ -O3 -std=c++17 --offload-arch=gfx1200 \
    matrix_multiply_rdna4_swmmac.hip -o mm_rdna4_swmmac
./mm_rdna4_swmmac
```

Note

`SwmmacRdna4F16Policy` requires an RDNA4 GPU (`gfx1200` or `gfx1201`). The `#if defined(__gfx1200__) || defined(__gfx1201__)` guard in the example file falls back to `ScalarFMASPolicy` on other targets, so the file compiles without modification.

The example uses pre-sparsified input data with a fixed 2:4 pattern. In a production kernel, apply a sparsity pruning pass to the weight matrix offline and store the compressed values and index tensor separately.

15.2.8 Instruction throughput

The cycle count below is the value used to compute theoretical peak throughput: peak throughput $= \frac{\text{ops per instruction}}{\text{cycle count}} \times$ clock frequency.

Builtin	Ops	Cycle count
`__builtin_amdgcn_swmmac_f32_16x16x32_f16_w32`	16384	16
`__builtin_amdgcn_swmmac_f32_16x16x32_bf16_w32`	16384	16
`__builtin_amdgcn_swmmac_f16_16x16x32_f16_w32`	16384	16
`__builtin_amdgcn_swmmac_bf16_16x16x32_bf16_w32`	16384	16
`__builtin_amdgcn_swmmac_i32_16x16x32_iu8_w32`	16384	8
`__builtin_amdgcn_swmmac_i32_16x16x32_iu4_w32`	16384	8
`__builtin_amdgcn_swmmac_i32_16x16x64_iu4_w32`	32768	8
`__builtin_amdgcn_swmmac_f32_16x16x32_fp8_fp8_w32`	16384	8
`__builtin_amdgcn_swmmac_f32_16x16x32_fp8_bf8_w32`	16384	8
`__builtin_amdgcn_swmmac_f32_16x16x32_bf8_fp8_w32`	16384	8
`__builtin_amdgcn_swmmac_f32_16x16x32_bf8_bf8_w32`	16384	8

15.2.9 Builtin reference

The following sections list every SWMMAC builtin available on RDNA4, grouped by accumulator type.

15.2.9.1 FP32-accumulate builtins

These builtins accumulate into FP32 and accept FP16, BF16, FP8, or BF8 matrix inputs.

15.2.9.1.1 FP16 inputs

The following builtins use FP16 matrix inputs.

15.2.9.1.1.1 `__builtin_amdgcn_swmmac_f32_16x16x32_f16_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_swmmac_f32_16x16x32_f16_w32(
    v8fp16  srcA,
    v16fp16 srcB,
```

(continues on next page)

(continued from previous page)

```
    v8float srcC,
    int     index);
```

Computes one step of a sparse 16×16 FP32 accumulation with FP16 inputs. $\boldsymbol{A}$ is a compressed sparse fragment (8 FP16 values per lane representing 16 K positions after 2:4 expansion). $\boldsymbol{B}$ is a dense fragment (16 FP16 values per lane). The `index` register identifies the non-zero positions in $\boldsymbol{A}$.

Parameter	Type	Description
`srcA`	v8fp16	Eight compressed FP16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v16fp16	Sixteen dense FP16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8float` – updated accumulator (expand(srcA, index) $\times$ srcB + srcC).

15.2.9.1.2 BF16 inputs

The following builtins use BF16 matrix inputs.

15.2.9.1.2.1 `__builtin_amdgcn_swmmac_f32_16x16x32_bf16_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_swmmac_f32_16x16x32_bf16_w32(
    v8short  srcA,
    v16short srcB,
    v8float  srcC,
    int      index);
```

Computes one step of a sparse 16×16 FP32 accumulation with BF16 inputs. BF16 values are passed as `short` (16-bit storage). Otherwise identical in structure to the FP16 variant.

Parameter	Type	Description
`srcA`	v8short	Eight compressed BF16 elements of $\boldsymbol{A}$ per lane (BF16 stored as `short`).
`srcB`	v16short	Sixteen dense BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8float` – updated accumulator (expand(srcA, index) $\times$ srcB + srcC).

15.2.9.1.3 FP8 and BF8 inputs

The following builtins use FP8 and BF8 matrix inputs.

15.2.9.1.3.1 `__builtin_amdgcn_swmmac_f32_16x16x32_fp8_fp8_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_swmmac_f32_16x16x32_fp8_fp8_w32(
    v2int   srcA,
    v4int   srcB,
    v8float srcC,
    int     index);
```

Computes one step of a sparse 16×16 FP32 accumulation with FP8 (E4M3) inputs for both $\boldsymbol{A}$ and $\boldsymbol{B}$. Eight compressed FP8 values per lane are packed into two 32-bit registers; sixteen dense FP8 values into four registers.

Parameter	Type	Description
`srcA`	v2int	Eight compressed FP8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v4int	Sixteen dense FP8 elements of $\boldsymbol{B}$ per lane, packed into four 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8float` – updated accumulator $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.2.9.1.3.2 `__builtin_amdgcn_swmmac_f32_16x16x32_fp8_bf8_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_swmmac_f32_16x16x32_fp8_bf8_w32(
    v2int   srcA,
    v4int   srcB,
    v8float srcC,
    int     index);
```

Computes one step of a sparse 16×16 FP32 accumulation with mixed 8-bit floating-point inputs: FP8 (E4M3) for $\boldsymbol{A}$ and BF8 (E5M2) for $\boldsymbol{B}$.

Parameter	Type	Description
`srcA`	v2int	Eight compressed FP8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v4int	Sixteen dense BF8 elements of $\boldsymbol{B}$ per lane, packed into four 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8float` – updated accumulator $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.2.9.1.3.3 `__builtin_amdgcn_swmmac_f32_16x16x32_bf8_fp8_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_swmmac_f32_16x16x32_bf8_fp8_w32(
    v2int   srcA,
    v4int   srcB,
    v8float srcC,
    int     index);
```

Computes one step of a sparse 16×16 FP32 accumulation with mixed 8-bit floating-point inputs: BF8 (E5M2) for $\boldsymbol{A}$ and FP8 (E4M3) for $\boldsymbol{B}$.

Parameter	Type	Description
`srcA`	v2int	Eight compressed BF8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v4int	Sixteen dense FP8 elements of $\boldsymbol{B}$ per lane, packed into four 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8float` – updated accumulator $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.2.9.1.3.4 `__builtin_amdgcn_swmmac_f32_16x16x32_bf8_bf8_w32`

Signature and parameters for this builtin.

```
v8float __builtin_amdgcn_swmmac_f32_16x16x32_bf8_bf8_w32(
    v2int   srcA,
    v4int   srcB,
    v8float srcC,
    int     index);
```

Computes one step of a sparse 16×16 FP32 accumulation with BF8 (E5M2) inputs for both $\boldsymbol{A}$ and $\boldsymbol{B}$. Eight compressed BF8 values per lane are packed into two 32-bit registers; sixteen dense BF8 values into four registers.

Parameter	Type	Description
`srcA`	v2int	Eight compressed BF8 elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers (four 8-bit values per register).
`srcB`	v4int	Sixteen dense BF8 elements of $\boldsymbol{B}$ per lane, packed into four 32-bit registers.
`srcC`	v8float	Accumulator input: eight FP32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8float` – updated accumulator $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.2.9.2 FP16-accumulate builtins

This builtin accumulates into FP16 with FP16 inputs.

15.2.9.2.1 FP16 inputs

The following builtin uses FP16 matrix inputs.

15.2.9.2.1.1 `__builtin_amdgcn_swmmac_f16_16x16x32_f16_w32`

Signature and parameters for this builtin.

```
v8fp16 __builtin_amdgcn_swmmac_f16_16x16x32_f16_w32(
    v8fp16  srcA,
    v16fp16 srcB,
```

(continues on next page)

(continued from previous page)

```
    v8fp16  srcC,
    int     index);
```

Computes one step of a sparse 16×16 FP16 accumulation. Both inputs and the accumulator are FP16. Operand sizes and the sparsity index are identical to the FP32-accumulate FP16 variant.

Parameter	Type	Description
`srcA`	v8fp16	Eight compressed FP16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v16fp16	Sixteen dense FP16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8fp16	Accumulator input: eight FP16 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8fp16` – updated accumulator $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.2.9.3 BF16-accumulate builtins

This builtin accumulates into BF16 with BF16 inputs.

15.2.9.3.1 BF16 inputs

The following builtin uses BF16 matrix inputs.

15.2.9.3.1.1 `__builtin_amdgcn_swmmac_bf16_16x16x32_bf16_w32`

Signature and parameters for this builtin.

```
v8short __builtin_amdgcn_swmmac_bf16_16x16x32_bf16_w32(
    v8short  srcA,
    v16short srcB,
    v8short  srcC,
    int      index);
```

Computes one step of a sparse 16×16 BF16 accumulation. Both inputs and the accumulator are BF16 (stored as `short`).

Parameter	Type	Description
`srcA`	v8short	Eight compressed BF16 elements of $\boldsymbol{A}$ per lane.
`srcB`	v16short	Sixteen dense BF16 elements of $\boldsymbol{B}$ per lane.
`srcC`	v8short	Accumulator input: eight BF16 elements per lane (stored as `short`).
`index`	int	Sparsity index, see *Common parameters*.

Returns `v8short` – updated accumulator (BF16 stored as `short`) $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.2.9.4 INT32-accumulate builtins

Integer SWMMAC builtins accept either signed or unsigned 8-bit or 4-bit integer inputs, controlled by the `a_neg` and `b_neg` compile-time constants.

15.2.9.4.1 INT8 and UINT8 inputs (16x16x32)

The following builtins use INT8 and UINT8 matrix inputs.

15.2.9.4.1.1 `__builtin_amdgcn_swmmac_i32_16x16x32_iu8_w32`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_swmmac_i32_16x16x32_iu8_w32(
    bool  a_neg,
    v2int srcA,
    bool  b_neg,
    v4int srcB,
    v8int srcC,
    int   index,
    bool  clamp);
```

Computes one step of a sparse 16×16 INT32 accumulation with 8-bit integer inputs. $\boldsymbol{A}$ is packed as two `int` registers per lane (8 bytes = 8 INT8 elements after 2:4 expansion to 16 K positions). $\boldsymbol{B}$ is packed as four `int` registers per lane (16 bytes = 16 INT8 elements).

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT8, `false` for unsigned UINT8. Compile-time constant, see *Common parameters*.
`srcA`	v2int	Eight compressed 8-bit elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers.
`b_neg`	bool	`true` for signed INT8, `false` for unsigned UINT8 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	v4int	Sixteen dense 8-bit elements of $\boldsymbol{B}$ per lane, packed into four 32-bit registers.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator (expand(srcA, index) $\times$ srcB + srcC).

15.2.9.4.2 INT4 and UINT4 inputs (16x16x32)

The following builtins use INT4 and UINT4 matrix inputs.

15.2.9.4.2.1 `__builtin_amdgcn_swmmac_i32_16x16x32_iu4_w32`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_swmmac_i32_16x16x32_iu4_w32(
    bool  a_neg,
    int   srcA,
    bool  b_neg,
    v2int srcB,
    v8int srcC,
    int   index,
    bool  clamp);
```

Computes one step of a sparse 16×16 INT32 accumulation with 4-bit integer inputs. Eight INT4 elements of $\boldsymbol{A}$ fit into a single `int` per lane; sixteen INT4 elements of $\boldsymbol{B}$ fit into two `int` registers.

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{A}$. Compile-time constant, see *Common parameters*.
`srcA`	int	Eight compressed 4-bit elements of $\boldsymbol{A}$ per lane, packed into one 32-bit register.
`b_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	v2int	Sixteen dense 4-bit elements of $\boldsymbol{B}$ per lane, packed into two 32-bit registers.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.2.9.4.3 INT4 and UINT4 inputs (16x16x64)

The following builtins use INT4 and UINT4 matrix inputs.

15.2.9.4.3.1 `__builtin_amdgcn_swmmac_i32_16x16x64_iu4_w32`

Signature and parameters for this builtin.

```
v8int __builtin_amdgcn_swmmac_i32_16x16x64_iu4_w32(
    bool  a_neg,
    v2int srcA,
    bool  b_neg,
    v4int srcB,
    v8int srcC,
    int   index,
    bool  clamp);
```

Computes one step of a sparse 16×16 INT32 accumulation with 4-bit integer inputs over a deeper K=64 strip. Sixteen INT4 elements of $\boldsymbol{A}$ are packed into two `int` registers per lane; thirty-two INT4 elements of $\boldsymbol{B}$ into four `int` registers.

Parameter	Type	Description
`a_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{A}$. Compile-time constant, see *Common parameters*.
`srcA`	v2int	Sixteen compressed 4-bit elements of $\boldsymbol{A}$ per lane, packed into two 32-bit registers.
`b_neg`	bool	`true` for signed INT4, `false` for unsigned UINT4 in $\boldsymbol{B}$. Compile-time constant, see *Common parameters*.
`srcB`	v4int	Thirty-two dense 4-bit elements of $\boldsymbol{B}$ per lane, packed into four 32-bit registers.
`srcC`	v8int	Accumulator input: eight INT32 elements per lane.
`index`	int	Sparsity index, see *Common parameters*.
`clamp`	bool	Clamp output to input type range on overflow. Compile-time constant, see *Common parameters*.

Returns `v8int` – updated accumulator $(\text{expand}(\text{srcA}, \text{index}) \times \text{srcB} + \text{srcC})$.

15.3 RDNA4 WMMA transpose load builtins

RDNA4 (`gfx1200`, `gfx1201`) provides hardware instructions that load a matrix sub-tile into per-lane VGPRs while transposing row-major and column-major storage order in the same operation. When a matrix's storage order in memory doesn't match the fragment layout the *RDNA4 dense WMMA builtins* expect, these builtins remove the need for a scalar transpose loop.

> **Note**
>
> Like the WMMA builtins they feed, these load-transpose instructions are supported only for **wave32** on RDNA4. RDNA4 also has `wavefrontsize64` transpose-load variants, but since RDNA4 WMMA itself is wave32-only, those aren't tied to feeding WMMA and aren't covered on this page.

15.3.1 Architecture availability

The builtins on this page target RDNA4 GPUs. To automatically enable them, pass the Low Level Virtual Machine (LLVM) target architecture flag at compile time:

```
amdclang++ --offload-arch=gfx1200 ...
amdclang++ --offload-arch=gfx1201 ...
```

15.3.2 Naming convention

All transpose-load builtins on this page follow the pattern:

```
__builtin_amdgcn_global_load_tr_b<M>_v<K><type>
```

Unlike CDNA5's transpose-load builtins, RDNA4 names carry no bit-width digit (no `tr16`/`tr8`) – RDNA4 has only one transpose-load instruction family per load size, so no extra digit is needed to disambiguate it.

M
: Bits loaded per lane (`128` or `64`).

K
: Number of elements in the return vector.

type
: Element type of the return vector: `i32`, `i16`, `f16`, or `bf16`.

15.3.3 Register types used in this reference

The signatures below use the following type aliases, which you can declare with Clang vector attributes in any HIP translation unit:

```
using v2int   = int      [[clang::ext_vector_type(2)]];
using v8short = short    [[clang::ext_vector_type(8)]];
using v8half  = _Float16 [[clang::ext_vector_type(8)]];
using v8bf16  = __bf16   [[clang::ext_vector_type(8)]];
```

> **Note**
>
> The `f16` builtin requires the `__fp16` type (GCC-style half-precision), not `_Float16` (ISO C half-precision). Cast pointers and vector types to `__fp16`-based variants at the call site only – the standard WMMA builtins accept

`_Float16` as usual. The worked example below does exactly this with its own `half8` alias.

15.3.4 Hardware constraints

- **Sub-tile per call, not a full 16x16 tile.** `global_load_tr_b128` covers an 8-row x 32-column sub-tile per call (see *Per-lane fragment layout* below) – two calls, the second offset by 8 rows, are needed to fill a full 16-row K-strip. This differs from CDNA5's `tr16_b128`/`tr8_b64`, which each load a complete 16×16 tile in a single call.
- **Wave32 only.**
- All builtins are `const` and have no side effects.

15.3.5 Per-lane fragment layout

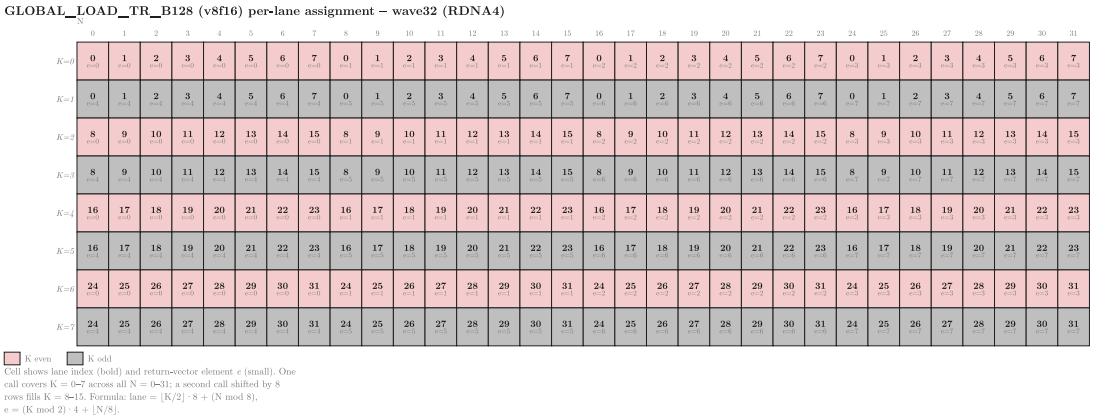


Fig. 1: Per-lane assignment for `global_load_tr_b128_v8f16`. Each cell shows the lane index (bold) and the return-vector element index e (small) that produced it.

Given a call whose base K-row is `k_tile` and base N-column is `block_n`, lane `tx` (0–31) reads from:

```
ptr(tx) = &input[(k_tile + tx/4) * cols + block_n + (tx%4)*8]
```

and the returned vector's element `e` (0–7) lands at:

$$\text{row} = k_tile + \lfloor tx/8 \rfloor \cdot 2 + \lfloor e/4 \rfloor$$
$$\text{col} = block_n + (e \bmod 4) \cdot 8 + (tx \bmod 8)$$

A second call with `ptr` shifted by 8 rows (`k_tile + 8`) fills the upper half of a 16-row K-strip; together the two calls cover 16×32 elements – exactly the B operand a `wmma_f32_16x16x16_f16_w32_gfx12` K-step needs from two adjacent wave32 blocks. `global_load_tr_b128_v8i16` and `_v8bf16` use the same 128-bit data path with a different element-type interpretation, so they follow this same mapping. `global_load_tr_b64_v2i32` uses a different (64-bit) data path; no per-lane mapping is documented for it below.

15.3.6 Builtin reference

15.3.6.1 `__builtin_amdgcn_global_load_tr_b64_v2i32`

```
v2int __builtin_amdgcn_global_load_tr_b64_v2i32(v2int* ptr);
```

Loads and transposes 64 bits per lane of 8-bit data (INT8, FP8, or BF8), returning two packed 32-bit words rather than eight discrete 8-bit elements. This builtin uses a different data path from `global_load_tr_b128`; see *Per-lane fragment layout*.

Parameter	Type	Description
`ptr`	`v2int*`	Pointer to global address space.

Returns `v2int` – two 32-bit words holding the lane's 8-bit fragment after transposition.

15.3.6.2 `__builtin_amdgcn_global_load_tr_b128_v8i16`

```
v8short __builtin_amdgcn_global_load_tr_b128_v8i16(v8short* ptr);
```

Loads and transposes 128 bits per lane of INT16 data. Expected to share `v8f16`'s permutation – see *Per-lane fragment layout*.

Parameter	Type	Description
`ptr`	`v8short*`	Pointer to global address space.

Returns `v8short` – eight INT16 elements holding the lane's fragment after transposition.

15.3.6.3 `__builtin_amdgcn_global_load_tr_b128_v8f16`

```
v8half __builtin_amdgcn_global_load_tr_b128_v8f16(v8half* ptr);
```

Loads and transposes 128 bits per lane of FP16 data. See the `__fp16` note above – cast `ptr` and the return value to a `__fp16`-based `v8half` at the call site. Its per-lane mapping is detailed in *Per-lane fragment layout*.

Parameter	Type	Description
`ptr`	`v8half*`	Pointer to global address space.

Returns `v8half` – eight FP16 elements holding the lane's fragment after transposition.

15.3.6.4 `__builtin_amdgcn_global_load_tr_b128_v8bf16`

```
v8bf16 __builtin_amdgcn_global_load_tr_b128_v8bf16(v8bf16* ptr);
```

Loads and transposes 128 bits per lane of BF16 data. Expected to share `v8f16`'s permutation – see *Per-lane fragment layout*.

Parameter	Type	Description
`ptr`	`v8bf16*`	Pointer to global address space.

Returns `v8bf16` – eight BF16 elements holding the lane's fragment after transposition.

15.3.7 Using a transpose load as a compute policy

Using MFMA builtins as a compute policy explains the `ComputePolicy`/`TilePolicy` pattern used to separate tile staging from the compute loop. This section extends the RDNA4 WMMA FP16 example from *Using WMMA builtins as a compute policy* with a transpose-load tile policy.

The complete source file is available for download:

- `matrix_multiply_rdna4_wmma.hip`

`WmmaRdna4F16TrPolicy` replaces the scalar B transpose with a hardware- accelerated `GLOBAL_LOAD_TR_B128` instruction (`__builtin_amdgcn_global_load_tr_b128_v8f16`). This intrinsic loads 128 bits per lane from global memory and transposes the data into LDS in a single operation, avoiding the per-element scalar transpose loop. The per-lane addressing follows exactly the formula in *Per-lane fragment layout*.

`SingleBufferTilePolicyF16TrB` fills B's shared-memory tile using this instruction instead of a scalar loop:

Listing 1: SingleBufferTilePolicyF16TrB (matrix_multiply_rdna4_wmma.hip)

```
template<int BlockTileM_, int BlockTileN_, int KTileSize_>
struct SingleBufferTilePolicyF16TrB
{
    static constexpr int num_buffers  = 1;
    static constexpr int block_tile_m = BlockTileM_;
    static constexpr int block_tile_n = BlockTileN_;
    static constexpr int k_tile_size  = KTileSize_;

    struct SharedStorage
    {
        _Float16 tile_a[BlockTileM_][KTileSize_];
        // B stored transposed: tile_b_T[n][k], same layout as baseline.
        // Filled by global_load_tr hardware scatter from row-major global B.
        _Float16 tile_b_T[BlockTileN_][KTileSize_];
    };

    __device__ static void prologue(SharedStorage& /*smem*/,
                                    const _Float16* /*A*/, const _Float16* /*B*/,
                                    int /*block_row_start*/, int /*block_col_start*/,
                                    int /*m*/, int /*n*/, int /*k*/,
                                    int /*tid*/, int /*num_threads*/) {}

    __device__ static void prefetch(SharedStorage&  smem,
                                    const _Float16* A,
                                    const _Float16* B_row,
                                    int             t_next,
                                    int block_row_start, int block_col_start,
                                    int m, int n, int k,
                                    int tid, int num_threads)
    {
        const int t = t_next;

        cooperative_load_tile_a_f16(
            &smem.tile_a[0][0], A, block_row_start,
            block_tile_m, t, k_tile_size, m, k, tid, num_threads);

#if defined(__gfx1200__) || defined(__gfx1201__)
```

(continues on next page)

(continued from previous page)

```
        using half8 = __attribute__((ext_vector_type(8))) __fp16;

        constexpr int wave_size = 32;
        const int lane_id       = tid % wave_size;
        const int k_tile        = t * k_tile_size;
        const int col_offset    = lane_id % 8;
        const int row_pair_base = (lane_id / 8) * 2;

        // Call 1: lower K-half (k_local = 0..7).
        {
            const __fp16* ptr_b =
                reinterpret_cast<const __fp16*>(
                    &B_row[(k_tile + lane_id / 4) * n
                           + block_col_start + (lane_id % 4) * 8]);
            const half8 tr_lo =
                __builtin_amdgcn_global_load_tr_b128_v8f16(
                    (__attribute__((address_space(1))) half8*)ptr_b);
            #pragma unroll
            for(int e = 0; e < 8; ++e)
            {
                const int n_local = (e % 4) * 8 + col_offset;
                const int k_local = row_pair_base + e / 4;
                smem.tile_b_T[n_local][k_local] = tr_lo[e];
            }
        }

        // Call 2: upper K-half (k_local = 8..15).
        {
            const __fp16* ptr_b =
                reinterpret_cast<const __fp16*>(
                    &B_row[(k_tile + (k_tile_size / 2) + lane_id / 4) * n
                           + block_col_start + (lane_id % 4) * 8]);
            const half8 tr_hi =
                __builtin_amdgcn_global_load_tr_b128_v8f16(
                    (__attribute__((address_space(1))) half8*)ptr_b);
            #pragma unroll
            for(int e = 0; e < 8; ++e)
            {
                const int n_local = (e % 4) * 8 + col_offset;
                const int k_local = (k_tile_size / 2) + row_pair_base + e / 4;
                smem.tile_b_T[n_local][k_local] = tr_hi[e];
            }
        }
#endif
    }

    __device__ static void acquire(int /*t*/) { __syncthreads(); }
    __device__ static void release(int /*t*/) { __syncthreads(); }
    __device__ static int  buf_idx(int /*t*/) { return 0; }

    __device__ static _Float16* get_tile_a(SharedStorage& smem, int /*buf*/)
    {
```

(continues on next page)

(continued from previous page)

```
        return &smem.tile_a[0][0];
    }
    __device__ static _Float16* get_tile_b_T(SharedStorage& smem, int /*buf*/)
    {
        return &smem.tile_b_T[0][0];
    }
};
```

The compute path (`load_a`, `load_b`, `mma`, `store_c`) is identical to the baseline `WmmaRdna4F16Policy` – only the tile policy's `prefetch()` stage changes:

Listing 2: WmmaRdna4F16TrPolicy (matrix_multiply_rdna4_wmma.hip)

```
struct WmmaRdna4F16TrPolicy
{
    // -- ComputePolicy constants ------------------------------------------------
    static constexpr int thread_tile_m   = 16;
    static constexpr int thread_tile_n   = 16;
    static constexpr int frag_size_m     = 8;
    static constexpr int frag_size_n     = 8;
    // RDNA4: wave32, 32 lanes per wavefront.
    static constexpr int effective_lanes = 32;
    // k_step=16: one wmma_f32_16x16x16_f16_w32 call consumes 16 K-positions,
    // filled by two GLOBAL_LOAD_TR_B128 calls (one per 8-K half).
    static constexpr int k_step          = 16;

    using elem_a = _Float16;
    using elem_b = _Float16;

    // Accumulator and zero() are identical to WmmaRdna4F16Policy.
    using wmma_f8 = float [[clang::ext_vector_type(8)]];

    struct Accumulator
    {
        wmma_f8 regs;
    };

    __device__ static void zero(Accumulator& acc)
    {
        acc.regs = wmma_f8{};
    }

    // thread_tile_offset is identical to WmmaRdna4F16Policy.
    __device__ static void thread_tile_offset(int  tid,
                                              int  /*lane_id*/,
                                              int* thread_row,
                                              int* thread_col)
    {
        constexpr int waves_n = 2;
        const int wid   = tid / 32; // wave32
        const int wid_x = wid % waves_n;
        const int wid_y = wid / waves_n;
```

(continues on next page)

(continued from previous page)

```
        *thread_row = wid_y * 16;
        *thread_col = wid_x * 16;
    }

    // -- load_a ------------------------------------------------------------
    // Scalar load, identical to WmmaRdna4F16Policy::load_a.
    __device__ static void load_a(const _Float16* tile_a_ptr,
                                  int             tile_a_row,
                                  int             ki,
                                  int             k_tile_size,
                                  int             lane_id,
                                  elem_a        (&frag)[8])
    {
        const int row   = tile_a_row + (lane_id % 16);
        const int k_off = ki + 8 * (lane_id / 16);
        #pragma unroll
        for(int e = 0; e < 8; ++e)
            frag[e] = tile_a_ptr[row * k_tile_size + k_off + e];
    }

    // -- load_b ------------------------------------------------------------
    // Scalar read from tile_b_T[n][k] in LDS - identical to baseline.
    // The global_load_tr hardware transpose already happened in
    // SingleBufferTilePolicyF16TrB::prefetch().
    __device__ static void load_b(const _Float16* tile_b_T_ptr,
                                  int             tile_b_col,
                                  int             ki,
                                  int             k_tile_size,
                                  int             lane_id,
                                  elem_b        (&frag)[8])
    {
        const int col   = tile_b_col + (lane_id % 16);
        const int k_off = ki + 8 * (lane_id / 16);
        #pragma unroll
        for(int e = 0; e < 8; ++e)
            frag[e] = tile_b_T_ptr[col * k_tile_size + k_off + e];
    }

    // mma and store_c are identical to WmmaRdna4F16Policy.
    __device__ static void mma(Accumulator&       acc,
                               const elem_a     (&a_frag)[8],
                               const elem_b     (&b_frag)[8])
    {
#if defined(__gfx1200__) || defined(__gfx1201__)
        using half8 = __attribute__((ext_vector_type(8))) __fp16;
        const half8 a_vec = {a_frag[0], a_frag[1], a_frag[2], a_frag[3],
                             a_frag[4], a_frag[5], a_frag[6], a_frag[7]};
        const half8 b_vec = {b_frag[0], b_frag[1], b_frag[2], b_frag[3],
                             b_frag[4], b_frag[5], b_frag[6], b_frag[7]};
        acc.regs = __builtin_amdgcn_wmma_f32_16x16x16_f16_w32_gfx12(
            a_vec, b_vec, acc.regs);
#endif
```

(continues on next page)

(continued from previous page)

```
    }

    __device__ static void store_c(const Accumulator& acc,
                                   float*              C,
                                   int                 out_row_base,
                                   int                 out_col_base,
                                   int                 m,
                                   int                 n,
                                   int                 lane_id)
    {
        const int laneWrapped = lane_id % 16;
        const int laneGroup   = lane_id / 16;

        #pragma unroll
        for(int G = 0; G < 8; ++G)
        {
            const int i = laneGroup * 8 + G;
            const int j = laneWrapped;
            const int r = out_row_base + i;
            const int c = out_col_base + j;
            if(r < m && c < n)
                C[r * n + c] = acc.regs[G];
        }
    }
};
```

Instantiating the kernel

With either policy, plug it into the generic kernel alongside any `TilePolicy` whose `block_tile_m` and `block_tile_n` are multiples of 16 and whose `k_tile_size` is a multiple of `k_step = 16`. See *Using WMMA builtins as a compute policy* for the shared policy aliases, launch configuration, and kernel launch code – the same `matrix_multiply_rdna4_wmma.hip` file wires up both `WmmaTilePolicy` (baseline) and `WmmaTilePolicyTrB` (this page) and runs both back to back for comparison.

15.3.8 Compiling and running

```
amdclang++ -O3 --offload-arch=gfx1200 \
    matrix_multiply_rdna4_wmma.hip -o mm_rdna4_wmma
./mm_rdna4_wmma

amdclang++ -O3 --offload-arch=gfx1201 \
    matrix_multiply_rdna4_wmma.hip -o mm_rdna4_wmma
./mm_rdna4_wmma
```

Note

`WmmaRdna4F16TrPolicy` requires an RDNA4 GPU (`gfx1200` or `gfx1201`). The `#if defined(__gfx1200__) || defined(__gfx1201__)` guard in the example file falls back to `ScalarFMAPolicy` on other targets, so the file compiles without modification.