AMD ROCm™ Programming Guide

Release 7.2.3

Advanced Micro Devices, Inc.

May 04, 2026

CONTENTS

AMD ROCm is a software stack, composed primarily of open-source software, that provides the tools for programming AMD Graphics Processing Units (GPUs), from low-level kernels to high-level end-user applications.

The AMD ROCm Programming Guide introduces the core concepts, APIs, and best practices for programming with ROCm and the HIP programming language. It provides hands-on guidance for writing GPU kernels, managing memory, optimizing performance, and integrating HIP with the broader AMD ROCm ecosystem of tools and libraries.

For details on changes and version information, see *Release notes*.

Getting started

- *Introduction*
- *What is ROCm?*
- *What is HIP?*
- *Introduction to the HIP programming model*
- *Hardware implementation*
- *SAXPY Tutorial - Hello HIP*

Core HIP programming

- *Introduction to core HIP programming*
- *HIP kernel programming*
- *Using HIP runtime API*
  - *Initialization*
  - *Memory management*
  - *Error handling*
  - *Asynchronous concurrent execution*
  - *Cooperative groups*
  - *HIP graphs*
  - *Call stack*
  - *OpenGL interoperability*
  - *External resource interoperability*
- *GPU programming patterns*
  - *Two-dimensional kernels: matrix multiplication tutorial*
  - *Atomic operations: histogram tutorial*
  - *CPU-GPU cooperative computing: K-means clustering tutorial*
  - *Stencil operations: image convolution tutorial*
  - *Multi-kernel programming: breadth-first search tutorial*
- *HIP compilers*

Performance optimization techniques

- *Understanding GPU performance*
- *Performance guidelines*
- *Optimizing performance*

Known issues are listed and can be reported on the on the AMD ROCm Programming 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 Programming Guide.

Version	Release date
7.2.3	May 4, 2026
7.2.2	April 14, 2026
7.2.1	March 25, 2026
7.2.0	January 21, 2026
7.1.1	November 26, 2025

1.1 ROCm 7.2.3

No changes were made to the AMD ROCm Programming Guide.

1.2 ROCm 7.2.2

No changes were made to the AMD ROCm Programming Guide.

1.3 ROCm 7.2.1

1.3.1 New content

Added *HIP compilers* page with description of HIP compilers.

1.3.2 Updated content

Enhanced the following topics:

- *SAXPY Tutorial - Hello HIP*
- *Multi-GPU programming*
- *Performance guidelines*

1.4 ROCm 7.2.0

1.4.1 New content

Added a page listing third-party software compatible with ROCm: *Third-party tools*.

1.4.2 Updated content

- Minor command line argument update on reference/rocm_in_data_centers/kubernetes/configuration page.
- Enhanced *Hardware implementation* page.
- Enhanced *Performance guidelines* page.

1.5 ROCm 7.1.1

Initial release of the AMD ROCm Programming Guide.

CHAPTER

TWO

INTRODUCTION

The rapid growth of high-performance computing (HPC) and artificial intelligence (AI) has driven an equally rapid evolution in the software ecosystems that power these workloads. Users increasingly require an open, flexible, and high-performance GPU compute platform capable of scaling from exploratory prototyping to mission-critical production workloads. AMD's ROCm open-source software platform emerged to meet these needs by enabling developers to fully harness the computational power of AMD Instinct™ GPUs, AMD Radeon™ GPUs, and supported Ryzen™ APUs, while maintaining compatibility with widely adopted industry frameworks.

2.1 Key drivers of ROCm

Several major trends underpin the adoption and growth of ROCm:

- **Open ecosystem demand**: Organizations—from academic institutions to large-scale HPC centers—value transparent, community-driven platforms. ROCm's open-source foundation gives developers insight, flexibility, and long-term sustainability for their GPU computing stack.
- **HPC and AI convergence**: Modern workloads increasingly blend simulation, data analytics, and machine learning. ROCm is designed to accelerate this convergence, offering a unified environment that supports scientific computing, deep learning, training and inference, large-scale simulations, and data-driven modeling.
- **Performance and efficiency**: ROCm is optimized to extract maximum performance from AMD GPUs, enabling fine-grained optimizations across on-premise clusters and cloud deployments.

ROCm users include HPC researchers, AI practitioners, system integrators, cloud providers expanding GPU compute offerings, and developers building high-performance workloads on AMD hardware—from workstation users with desktop Radeon GPUs to large-scale supercomputing installations.

2.2 Key features of ROCm tools and the HIP runtime

At the heart of the ROCm ecosystem is the HIP runtime and programming model, a C++ API introduced by AMD in 2016. HIP provides a familiar, NVIDIA CUDA-like interface for high-performance development on AMD GPUs. Its design balances performance with maintainability, allowing developers to write quality parallel code.

Key features of the ROCm tools and HIP runtime include:

- **Rich language interoperability**: ROCm supports multiple programming languages and interfaces—including HIP, OpenMP, and OpenCL—offering flexibility for different application domains.
- **Optimized math, AI, and HPC libraries**: The platform includes a robust set of tuned libraries for BLAS, FFT, sparse linear algebra, random number generation, graph analytics, and deep learning primitives.
- **Comprehensive development tools**: ROCm provides profilers, debuggers, compilers, and analysis utilities that give developers control over performance tuning and hardware-aware optimization.

- **Scalable performance across AMD GPU families**: Applications can seamlessly run on AMD Instinct accelerators for datacenters as well as Radeon GPUs for workstations and development environments.
- **Integration with industry frameworks**: ROCm supports PyTorch, TensorFlow, JAX, and numerous HPC frameworks, allowing developers to adopt GPU acceleration without needing to rewrite entire codebases.

Together, HIP and the ROCm toolchain form a cohesive, high-performance environment for parallel programming, whether targeting deep learning pipelines or large-scale scientific simulation.

2.3 About AMD ROCm programming guide

This guide serves as a comprehensive programming guide for developers working with HIP and the broader ROCm ecosystem. It introduces HIP fundamentals, demonstrates best practices for GPU development, and provides practical examples that run on any AMD GPU supported by ROCm. While many examples reference AMD Instinct accelerators such as the MI100, the material remains portable and applicable across ROCm-supported hardware.

Importantly, this guide is derived from the official ROCm docs portal, reorganized for PDF export and offline usage. By consolidating and refining the online documentation into a unified volume, this guide provides developers with a clear, accessible, and convenient reference.

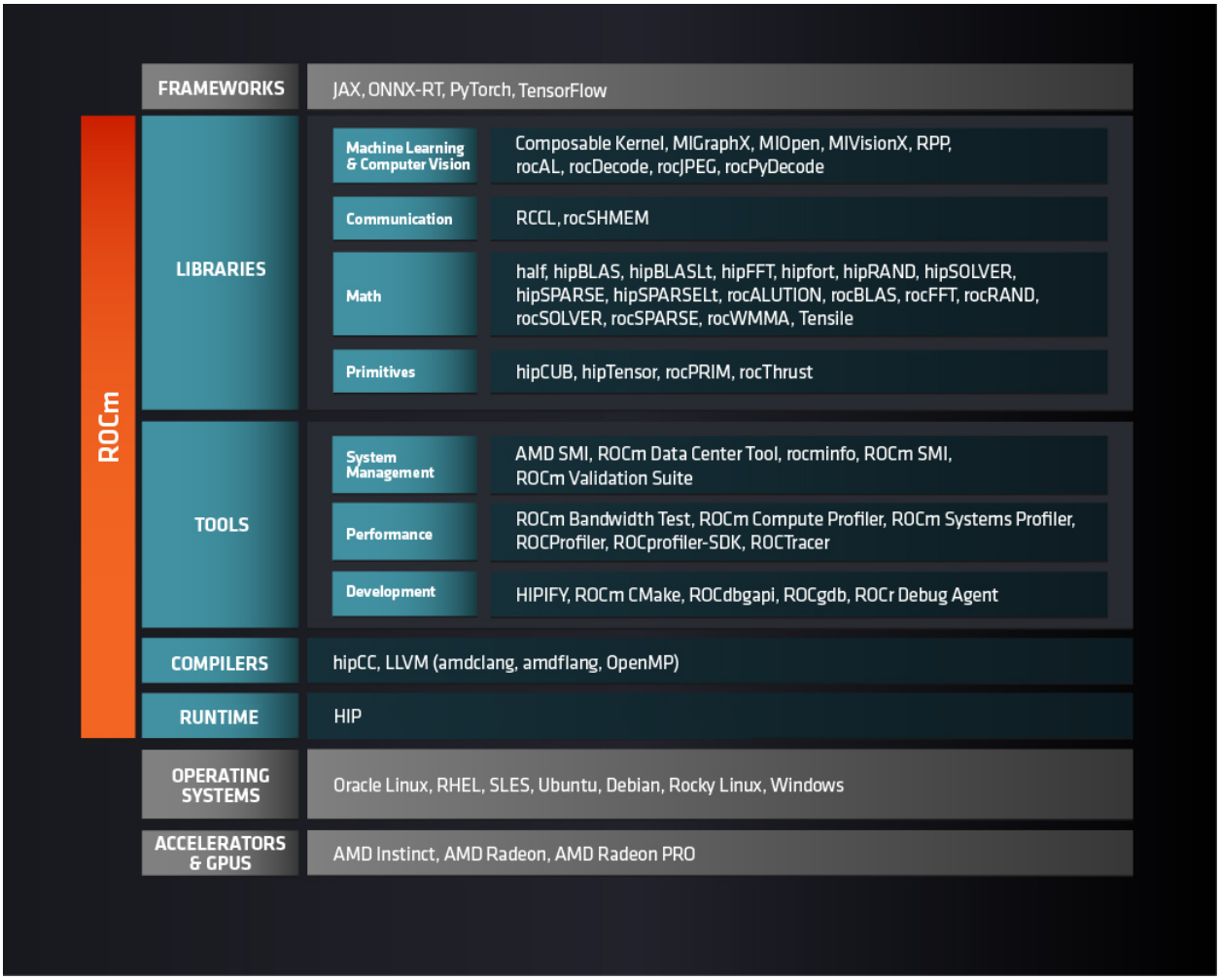
The chapters that follow introduce the principles of parallel programming on AMD GPUs, explore the features and ecosystem of HIP, and provide practical guidance for building efficient, portable, high-performance GPU applications.

CHAPTER

THREE

WHAT IS ROCM?

ROCm is a software stack, composed primarily of open-source software, that provides the tools for programming AMD Graphics Processing Units (GPUs), from low-level kernels to high-level end-user applications.



Specifically, ROCm provides the tools for HIP, OpenCL and OpenMP. These include compilers, libraries for high-level functions, debuggers, profilers and runtimes.

3.1 ROCm components

ROCm consists of the following components. For information on the license associated with each component, see ROCm licensing.

3.1.1 Libraries

3.1.1.1 Machine Learning & Computer Vision

Component	Description
Composable Kernel	Provides a programming model for writing performance critical kernels for machine learning workloads across multiple architectures
MIGraphX	Graph inference engine that accelerates machine learning model inference
MIOpen	An open source deep-learning library
MIVisionX	Set of comprehensive computer vision and machine learning libraries, utilities, and applications
ROCm Performance Primitives (RPP)	Comprehensive high-performance computer vision library for AMD processors with HIP/OpenCL/CPU back-ends
rocAL	An augmentation library designed to decode and process images and videos
rocDecode	High-performance SDK for access to video decoding features on AMD GPUs
rocJPEG	Library for decoding JPG images on AMD GPUs
rocPyDecode	Provides access to rocDecode APIs in both Python and C/C++ languages

Note

rocCV is an efficient GPU-accelerated library for image pre- and post-processing. rocCV is in an early access state. Using it on production workloads is not recommended.

3.1.1.2 Communication

Component	Description
RCCL	Standalone library that provides multi-GPU and multi-node collective communication primitives
rocSHMEM	An intra-kernel networking library that provides GPU-centric networking through an OpenSHMEM-like interface

3.1.1.3 Math

Component	Description
half	C++ header-only library that provides an IEEE 754 conformant, 16-bit half-precision floating-point type, along with corresponding arithmetic operators, type conversions, and common mathematical functions
hipBLAS	BLAS-marshaling library that supports rocBLAS and cuBLAS backends
hipBLASLt	Provides general matrix-matrix operations with a flexible API and extends functionalities beyond traditional BLAS library
hipFFT	Fast Fourier transforms (FFT)-marshalling library that supports rocFFT or cuFFT backends
hipfort	Fortran interface library for accessing GPU Kernels
hipRAND	Ports CUDA applications that use the cuRAND library into the HIP layer
hipSOLVER	An LAPACK-marshalling library that supports rocSOLVER and cuSOLVER backends
hipSPARSE	SPARSE-marshalling library that supports rocSPARSE and cuSPARSE backends
hipSPARSELt	SPARSE-marshalling library with multiple supported backends
rocALUTION	Sparse linear algebra library for exploring fine-grained parallelism on ROCm runtime and toolchains
rocBLAS	BLAS implementation (in the HIP programming language) on the ROCm runtime and toolchains
rocFFT	Software library for computing fast Fourier transforms (FFTs) written in HIP
rocRAND	Provides functions that generate pseudorandom and quasirandom numbers
rocSOLVER	An implementation of LAPACK routines on ROCm software, implemented in the HIP programming language and optimized for AMD's latest discrete GPUs
rocSPARSE	Exposes a common interface that provides BLAS for sparse computation implemented on ROCm runtime and toolchains (in the HIP programming language)
rocWMMA	C++ library for accelerating mixed-precision matrix multiply-accumulate (MMA) operations
Tensile	Creates benchmark-driven backend libraries for GEMMs, GEMM-like problems, and general N-dimensional tensor contractions

3.1.1.4 Primitives

Component	Description
hipCUB	Thin header-only wrapper library on top of rocPRIM or CUB that allows project porting using the CUB library to the HIP layer
hipTensor	AMD's C++ library for accelerating tensor primitives based on the composable kernel library
rocPRIM	Header-only library for HIP parallel primitives
rocThrust	Parallel algorithm library

3.1.2 Tools

3.1.2.1 System Management

Component	Description
AMD SMI	System management interface to control AMD GPU settings, monitor performance, and retrieve device and process information
ROCm Data Center Tool	Simplifies administration and addresses key infrastructure challenges in AMD GPUs in cluster and data-center environments
rocminfo	Reports system information
ROCm SMI	C library for Linux that provides a user space interface for applications to monitor and control GPU applications
ROCm Validation Suite	Detects and troubleshoots common problems affecting AMD GPUs running in a high-performance computing environment

3.1.2.2 Performance

Component	Description
ROCm Bandwidth Test	Captures the performance characteristics of buffer copying and kernel read/write operations
ROCm Compute Profiler	Kernel-level profiling for machine learning and high performance computing (HPC) workloads
ROCm Systems Profiler	Comprehensive profiling and tracing of applications running on the CPU or the CPU and GPU
ROCProfiler	Profiling tool for HIP applications
ROCprofiler-SDK	Toolkit for developing analysis tools for profiling and tracing GPU compute applications. This toolkit is in beta and subject to change
ROCTracer	Intercepts runtime API calls and traces asynchronous activity

Note

ROCprof Compute Viewer is a tool for visualizing and analyzing GPU thread trace data collected using rocprofv3. Note that ROCprof Compute Viewer is in an early access state. Running production workloads is not recommended.

3.1.2.3 Development

Component	Description
HIPIFY	Translates CUDA source code into portable HIP C++
ROCm CMake	Collection of CMake modules for common build and development tasks
ROCdbgapi	ROCm debugger API library
ROCm Debugger (ROCgdb)	Source-level debugger for Linux, based on the GNU Debugger (GDB)
ROCr Debug Agent	Prints the state of all AMD GPU wavefronts that caused a queue error by sending a SIGQUIT signal to the process while the program is running

3.1.3 Compilers

Component	Description
HIPCC	Compiler driver utility that calls Clang and passes the appropriate include and library options for the target compiler and HIP infrastructure
ROCm compilers	ROCm LLVM compiler infrastructure
FLANG	An out-of-tree Fortran compiler targeting LLVM

3.1.4 Runtime API

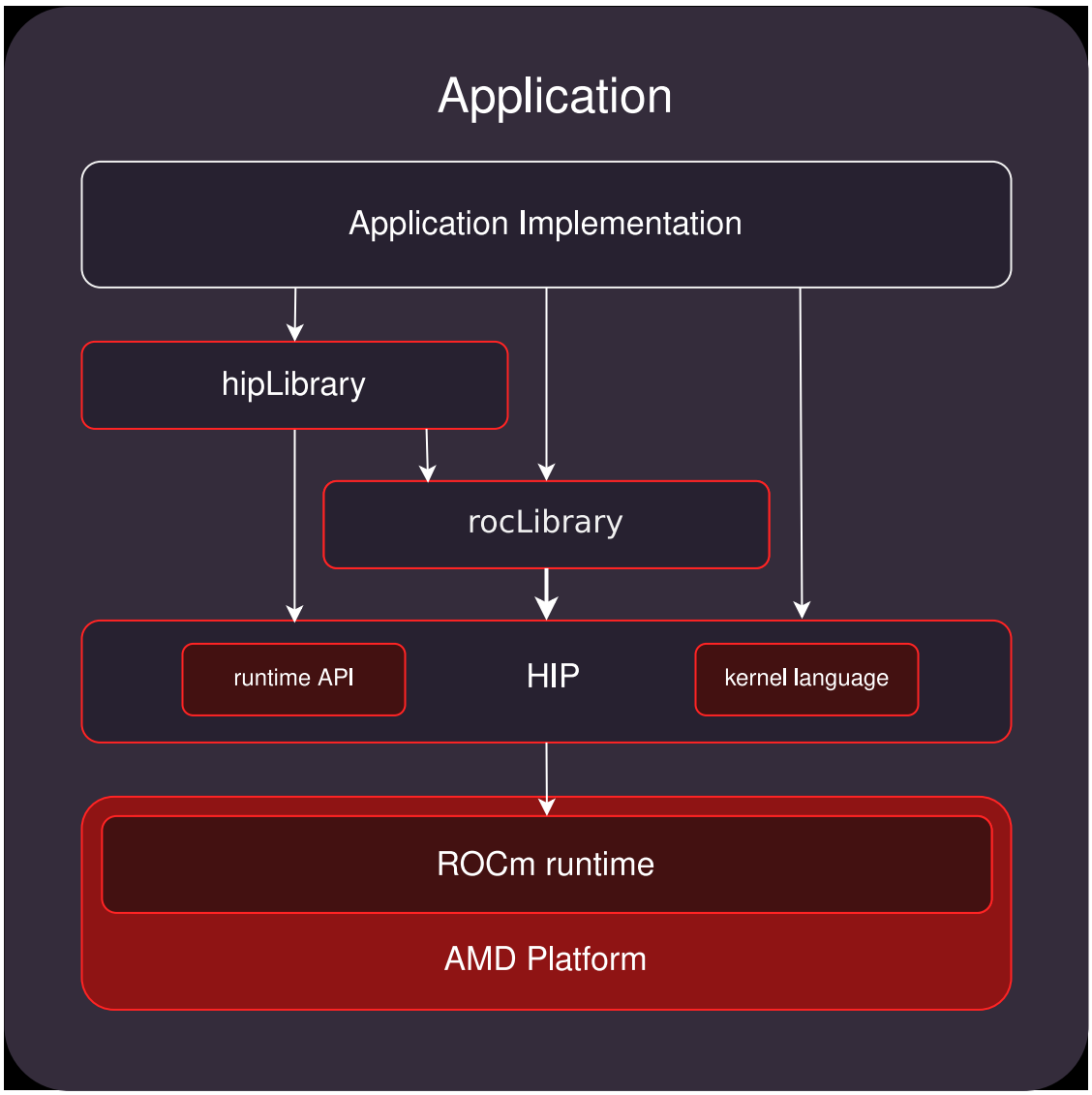
Component	Description
HIP	HIP is a C++ runtime API and kernel language for AMD GPUs

CHAPTER

FOUR

WHAT IS HIP?

HIP is a C++ runtime API and kernel language for AMD GPUs. It is part of AMD's ROCm platform and lets developers create applications that run on heterogeneous systems, using CPUs and AMD GPUs from a single source code base.



- HIP is a thin API with little or no performance impact over coding directly in AMD *ROCm*.

- HIP enables coding in a single-source C++ programming language, including features such as templates, C++11 lambdas, classes, namespaces, and more.
- Developers can tune for performance or handle tricky cases via HIP.

ROCm offers compilers (`clang`, `hipcc`), code profilers (`rocprofv3`), debugging tools (`rocgdb`), libraries and HIP with the runtime API and kernel language, to create heterogeneous applications running on both CPUs and GPUs. ROCm provides libraries like hipFFT and hipBLAS that provide API compatibility with equivalent NVIDIA CUDA libraries, making it easier to port existing NVIDIA CUDA applications. These libraries provide pointer-based memory interfaces and can be easily integrated into your applications.

GPU programmers with NVIDIA CUDA experience will find the HIP API straightforward. You can quickly port NVIDIA CUDA applications to run on AMD GPUs. The HIPify tools, based on the clang front-end and Perl language, can convert NVIDIA CUDA API calls into the corresponding HIP API calls. However, HIP is not intended to be a drop-in replacement for NVIDIA CUDA, and developers should expect to do some manual coding and performance tuning work to port existing projects to AMD GPUs as described in the HIP porting guide.

HIP provides two components: those that run on the CPU, also known as host system, and those that run on GPUs, also referred to as device. The host-based code is used to create device buffers, move data between the host application and a device, launch the device code (also known as kernel), manage streams and events, and perform synchronization. The kernel language provides a way to develop massively parallel programs that run on GPUs, and provides access to GPU specific hardware capabilities.

In summary, HIP simplifies porting NVIDIA CUDA applications to AMD GPUs, maintains performance, and provides a familiar C++ experience for GPU programming.

4.1 HIP components

HIP consists of the following components. For information on the license associated with each component, see HIP licensing.

4.1.1 C++ runtime API

HIP provides headers and a runtime library built on top of HIP-Clang compiler in the repository Compute Language Runtime (CLR). The HIP runtime implements HIP streams, events, and memory APIs, and is an object library that is linked with the application. The source code for all headers and the library implementation is available on GitHub.

For further details, check HIP Runtime API Reference.

4.1.2 Kernel language

HIP provides a C++ syntax that is suitable for compiling most code that commonly appears in compute kernels (classes, namespaces, operator overloading, and templates). HIP also defines other language features that are designed to target accelerators, such as:

- Short-vector headers that can serve on a host or device
- Math functions that resemble those in `math.h`, which is included with standard C++ compilers
- Built-in functions for accessing specific GPU hardware capabilities

For further details, check HIP C++ language extensions and Kernel language C++ support.

CHAPTER

FIVE

INTRODUCTION TO THE HIP PROGRAMMING MODEL

The HIP programming model enables mapping data-parallel C/C++ algorithms to massively parallel SIMD (Single Instruction, Multiple Data) architectures like GPUs. HIP supports many imperative languages, such as Python via PyHIP, but this document focuses on the original C/C++ API of HIP.

While GPUs may be capable of running applications written for CPUs if properly ported and compiled, it would not be an efficient use of GPU resources. GPUs fundamentally differ from CPUs and should be used accordingly to achieve optimum performance. A basic understanding of the underlying device architecture helps you make efficient use of HIP and general purpose graphics processing unit (GPGPU) programming in general. The following topics introduce you to the key concepts of GPU-based programming and the HIP programming model.

5.1 Hardware differences: CPU vs GPU

CPUs and GPUs have been designed for different purposes. CPUs quickly execute a single thread, decreasing the time for a single operation while increasing the number of sequential instructions that can be executed. This includes fetching data and reducing pipeline stalls where the ALU has to wait for previous instructions to finish.

With CPUs, the goal is to quickly process operations. CPUs provide low-latency processing for serial instructions. On the other hand, GPUs have been designed to execute many similar commands, or threads, in parallel, achieving higher throughput. Latency is the time between starting an operation and receiving its result, such as 2 ns, while throughput is the rate of completed operations, for example, operations per second.

For the GPU, the objective is to process as many operations in parallel, rather than to finish a single instruction quickly. GPUs in general are made up of basic building blocks called *compute units (CUs)*, that execute the threads of a kernel. As described in Hardware implementation, these CUs provide the necessary resources for the threads: the Arithmetic Logical Units (ALUs), register files, caches and shared memory for efficient communication between the threads.

The following describes a few hardware differences between CPUs and GPUs:

- CPU:
  - Optimized for sequential processing with a few powerful cores (4-64 typically)
  - High clock speeds (3-5 GHz)
  - One register file per thread. On modern CPUs you have at most 2 register files per core, called hyperthreading.
  - One ALU executing the thread.
    - Designed to quickly execute instructions of the same thread.
    - Complex branch prediction.
  - Large L1/L2 cache per core, shared by fewer threads (maximum of 2 when hyperthreading is available).
  - A disadvantage is switching execution from one thread to another (or context switching) takes a considerable amount of time: the ALU pipeline needs to be emptied, the register file has to be written to memory to free the register for another thread.

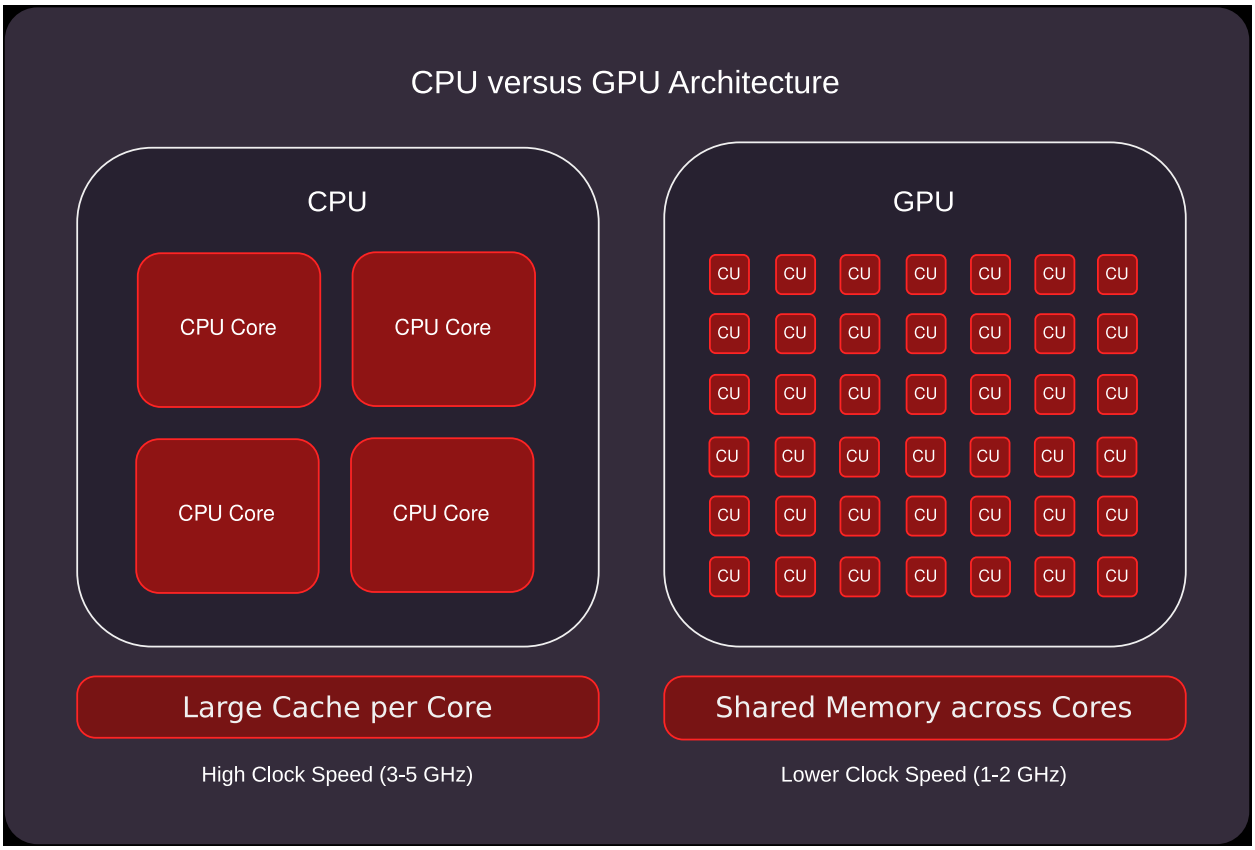


Fig. 1: Differences in CPUs and GPUs

- GPU:
  - Designed for parallel processing with many simpler cores (hundreds/thousands)
  - Lower clock speeds (1-2 GHz)
  - Streamlined control logic
  - Small caches, more registers
  - Register files are shared among threads. The number of threads that can be run in parallel depends on the registers needed per thread.
  - Multiple ALUs execute a collection of threads having the same operations, also known as a *warp* or wavefront. This is called single-instruction, multiple threads (SIMT) operation as described in *Single instruction multiple threads (SIMT)*.
    * The collection of ALUs is called SIMD. SIMDs are an extension to the hardware architecture that allows a *single instruction* to concurrently operate on *multiple data* inputs.
    * For branching threads where conditional instructions lead to thread divergence, ALUs still process the full warp, but the result for divergent threads is masked out. This leads to wasted ALU cycles and should be a consideration in your programming. Keep instructions consistent and leave conditionals out of threads.
  - The advantage for GPUs is that context switching is easy. All threads that run on a core/compute unit have their registers on the compute unit, so they don't need to be stored to global memory, and each cycle one instruction from any wavefront that resides on the compute unit can be issued.

When programming for a heterogeneous system, which incorporates CPUs and GPUs, you must write your program to take advantage of the strengths of the available hardware. Use the CPU for tasks that require complex logic with conditional branching, to reduce the time to reach a decision. Use the GPU for parallel operations of the same instruction across large datasets, with little branching, where the volume of operations is the key.

5.2 Heterogeneous programming

The HIP programming model has two execution contexts. The main application starts on the CPU, or the *host* processor, and compute kernels are launched on the *device* such as Instinct accelerators or AMD GPUs. The host execution is defined by the C++ abstract machine, while device execution follows the *SIMT model* of HIP. These two execution contexts are signified by the `__host__` and `__global__` (or `__device__`) decorators in HIP program code. There are a few key differences between the two contexts:

- The C++ abstract machine assumes a unified memory address space, meaning that one can always access any given address in memory (assuming the absence of data races). HIP however introduces several memory namespaces, an address from one means nothing in another. Moreover, not all address spaces are accessible from all contexts.

  Every CU has an instance of storage backing the namespace `__shared__`. Even if the host were to have access to these regions of memory, the performance benefits of the segmented memory subsystem are supported by the inability of asynchronous access from the host.

- Not all C++ language features map cleanly to typical GPU device architectures. Some C++ features have poor latency when implemented on GPU devices, therefore they are forbidden in device contexts to avoid using features that unexpectedly decimate the program's performance. Offload devices targeted by HIP aren't general purpose devices, at least not in the sense that a CPU is. HIP focuses on data parallel computations and as such caters to throughput optimized architectures, such as GPUs or accelerators derived from GPU architectures.

- Asynchronicity is at the forefront of the HIP API. Computations launched on the device execute asynchronously with respect to the host, and it is the user's responsibility to synchronize their data dispatch/fetch with computations on the device.

> **Note**
>
> HIP performs implicit synchronization on occasions, unlike some APIs where the responsibility for synchronization is left to the user.

5.2.1 Host programming

In heterogeneous programming, the CPU is available for processing operations but the host application has the additional task of managing data and computation exchanges between the CPU (host) and GPU (device). The host acts as the application manager, coordinating the overall workflow and directing operations to the appropriate context, handles data preparation and data transfers, and manages GPU tasks and synchronization. Here is a typical sequence of operations:

1. Initialize the HIP runtime and select the GPU: As described in *Initialization*, refers to identifying and selecting a target GPU, setting up a context to let the CPU interact with the GPU.
2. Data preparation: As discussed in *Memory management*, this includes allocating the required memory on the host and device, preparing input data and transferring it from the host to the device. The data is both transferred to the device, and passed as an input parameter when launching the kernel.
3. Configure and launch the kernel on the GPU: As described in *Device programming*, this defines kernel configurations and arguments, launches kernel to run on the GPU device using the triple chevron syntax or appropriate API call (for example `hipLaunchKernelGGL`). On the GPU, multiple kernels can run on streams, with a queue of operations. Within the same stream, operations run in the order they were issued, but on multiple streams operations are independent and can execute concurrently. In the HIP runtime, kernels run on the default stream when one is not specified, but specifying a stream for the kernel lets you increase concurrency in task scheduling and resource utilization, and launch and manage multiple kernels from the host program.
4. Synchronization: As described in *Asynchronous concurrent execution*, kernel execution occurs in the context of device streams, specifically the default (*0*) stream. You can use streams and events to manage task dependencies, overlap computation with data transfers, and manage asynchronous processes to ensure proper sequencing of operations. Wait for events or streams to finish execution and transfer results from the GPU back to the host.
5. Error handling: As described in *Error handling*, you should catch and handle potential errors from API calls, kernel launches, or memory operations. For example, use `hipGetErrorString` to retrieve error messages.
6. Cleanup and resource management: Validate results, clean up GPU contexts and resources, and free allocated memory on the host and devices.

This structure allows for efficient use of GPU resources and facilitates the acceleration of compute-intensive tasks while keeping the host CPU available for other tasks.

5.2.1.1 HIP Runtime API

The HIP Runtime API provides a high-level C-style interface that abstracts the lower-level ROCr Runtime, simplifying GPU programming on AMD hardware. It offers functions for memory management, kernel launches, synchronization, and device control, enabling developers to write GPU-accelerated applications in a portable, C++-friendly way.

The Runtime API serves the same role within ROCm as the CUDA Runtime API does in NVIDIA's stack. It simplifies GPU programming by abstracting explicit device and context management handled by the HSA/ROCr driver layer. Developers interact with familiar, high-level constructs such as `hipMalloc()`, `hipMemcpy()`, and `hipLaunchKernelGGL`, while the runtime handles resource allocation, queue management, and synchronization transparently.

The Runtime API can be linked either statically or dynamically, with the shared object typically named `libamdhip64.so` on Linux systems. It is open source and maintained as part of the ROCm ecosystem, allowing inspection, extension, and integration into custom build environments.

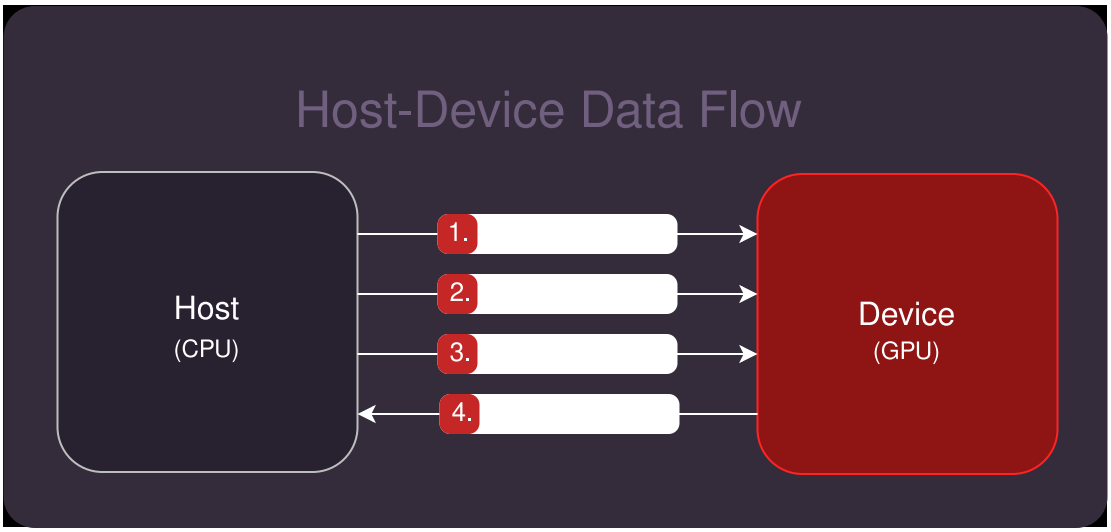


Fig. 2: Interaction of Host and Device in a GPU application

5.2.2 Device programming

The device or kernel program acts as a worker on the GPU application, distributing operations to be handled quickly and efficiently. Launching a kernel in the host application starts the kernel program running on the GPU, defining the parallel operations that repeat the same instructions across many datasets. Understanding how the kernel works and the processes involved is essential to writing efficient GPU applications. Threads, blocks, and grids provide a hierarchical approach to parallel operations. Understanding the thread hierarchy is critical to distributing work across the available CUs, managing parallel operations, and optimizing memory access. The general flow of the kernel program looks like this:

1. Thread Grouping: As described in *Hierarchical thread model*, threads are organized into a hierarchy consisting of threads, which are individual instances of parallel operations, blocks that group the threads, and grids that group blocks into the kernel. Each thread runs an instance of the kernel in parallel with other threads in the block.
2. Indexing: The kernel computes the unique index for each thread to access the relevant data to be processed by the thread.
3. Data Fetch: Threads fetch input data from memory previously transferred from the host to the device. As described in *Memory model*, the hierarchy of threads is influenced by the memory subsystem of GPUs. The memory hierarchy includes local memory per-thread with very fast access, shared memory for the block of threads which also supports quick access, and larger amounts of global memory visible to the whole kernel,but accesses are expensive due to high latency. Understanding the memory model is a key concept for kernel programming.
4. Computation: Threads perform the required computations on the input data, and generate any needed output. Each thread of the kernel runs the same instruction simultaneously on the different datasets. This sometimes require multiple iterations when the number of operations exceeds the resources of the CU.
5. Synchronization: When needed, threads synchronize within their block to ensure correct results when working with shared memory.

Kernels are parallel programs that execute the same instruction set across multiple threads, organized in *warps*, as described below and as demonstrated in the Hello World tutorial or ../tutorial/saxpy. However, heterogeneous GPU applications can also become quite complex, managing hundreds, thousands, or hundreds of thousands of operations with repeated data transfers between host and device to support massive parallelization, using multiple streams to manage concurrent asynchronous operations, using rich libraries of functions optimized for GPU hardware as described in the ROCm documentation.

5.3 Single instruction multiple threads (SIMT)

The HIP kernel code, written as a series of scalar instructions for multiple threads with different thread indices, gets mapped to the SIMD units of the GPUs. Every single instruction, which is executed for every participating thread of a kernel, gets mapped to the SIMD.

This is done by grouping threads into warps, which contain as many threads as there are physical lanes in a SIMD, and issuing that instruction to the SIMD for every warp of a kernel. Ideally, the SIMD is always fully utilized. However, if the number of threads can't be evenly divided by the warpSize, then the unused lanes are masked out from the corresponding SIMD execution.

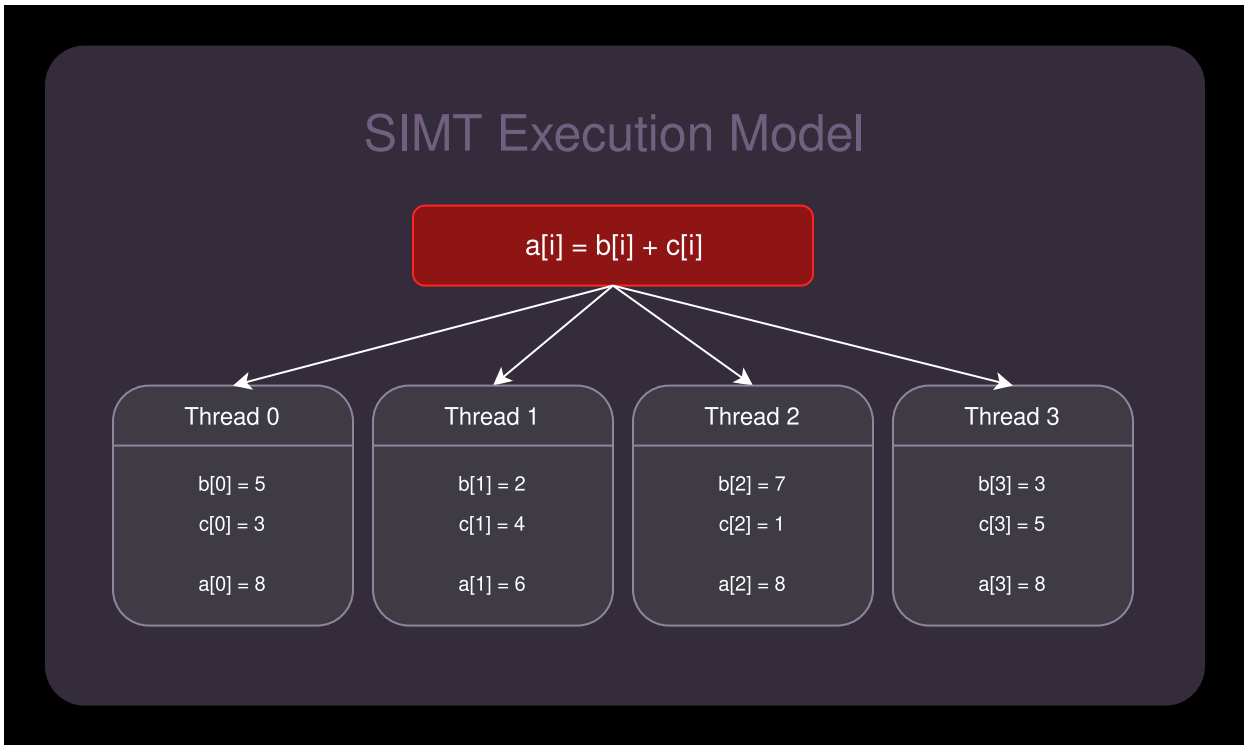


Fig. 3: Instruction flow of a sample SIMT program

A kernel follows the same C++ rules as the functions on the host, but it has a special `__global__` label to mark it for execution on the device, as shown in the following example:

```
__global__ void AddKernel(float* a, const float* b)
{
  int global_idx = threadIdx.x + blockIdx.x * blockDim.x;

  a[global_idx] += b[global_idx];
}
```

One of the first things you might notice is the usage of the special `threadIdx`, `blockIdx` and `blockDim` variables. Unlike normal C++ host functions, a kernel is not launched once, but as often as specified by the user. Each of these instances is a separate thread, with its own values for `threadIdx`, `blockIdx` and `blockDim`.

The kernel program is launched from the host application using a language extension called the triple chevron syntax, which looks like the following:

```
AddKernel<<<number_of_blocks, threads_per_block>>>(a, b);
```

Inside the angle brackets, provide the following:

- The number of blocks to launch, which defines the grid size (relating to blockDim).
- The number of threads in a block, which defines the block size (relating to blockIdx).
- The amount of shared memory to allocate by the host, not specified above.
- The device stream to enqueue the operation on, not specified above so the default stream is used.

> **Note**
>
> The kernel can also be launched through other methods, such as the `hipLaunchKernel()` function.

Here, the total number of threads launched for the `AddKernel` program is defined by `number_of_blocks * threads_per_block`. You define these values when launching the kernel program to address the problem to be solved with the available resources within the system. In other words, the thread configuration is customized to the needs of the operations and the available hardware.

For comparison, the `AddKernel` program could be written in plain C++ as a `FOR` loop:

```
for(int i = 0; i < (number_of_blocks * threads_per_block); ++i){
  a[i] += b[i];
}
```

In HIP, lanes of the SIMD architecture are fed by mapping threads of a SIMT execution, one thread down each lane of an SIMD engine. Execution parallelism usually isn't exploited from the width of the built-in vector types, but across multiple threads via the thread ID constants `threadIdx.x`, `blockIdx.x`, etc.

5.3.1 Hierarchical thread model

The thread hierarchy defines how parallel work is structured and executed across AMD GPUs, from single threads up to the full device. All threads of a kernel are uniquely identified by a set of integral values called thread IDs. The hierarchy consists of three levels: threads, blocks, and grids.

- Threads are single instances of kernel operations, running concurrently
- Blocks group threads together and enable cooperation and shared memory
- Grids define the number of thread blocks for a single kernel launch
- Blocks and grids can be defined in three dimensions (`x`, `y`, `z`)
- By default, the Y and Z dimensions are set to 1

This hierarchy maps directly onto AMD hardware:

- Threads execute on SIMD lanes
- *Work-groups* occupy *compute units*
- *Grids* utilize all available CUs across the GPU

The combined values represent the thread index, and relate to the sequence that the threads execute. The thread hierarchy is integral to how AMD GPUs operate, and is depicted in the following figure.

Thread (Work-item)
The smallest unit of execution in the HIP programming model is a thread, also called a work-item in lower-level

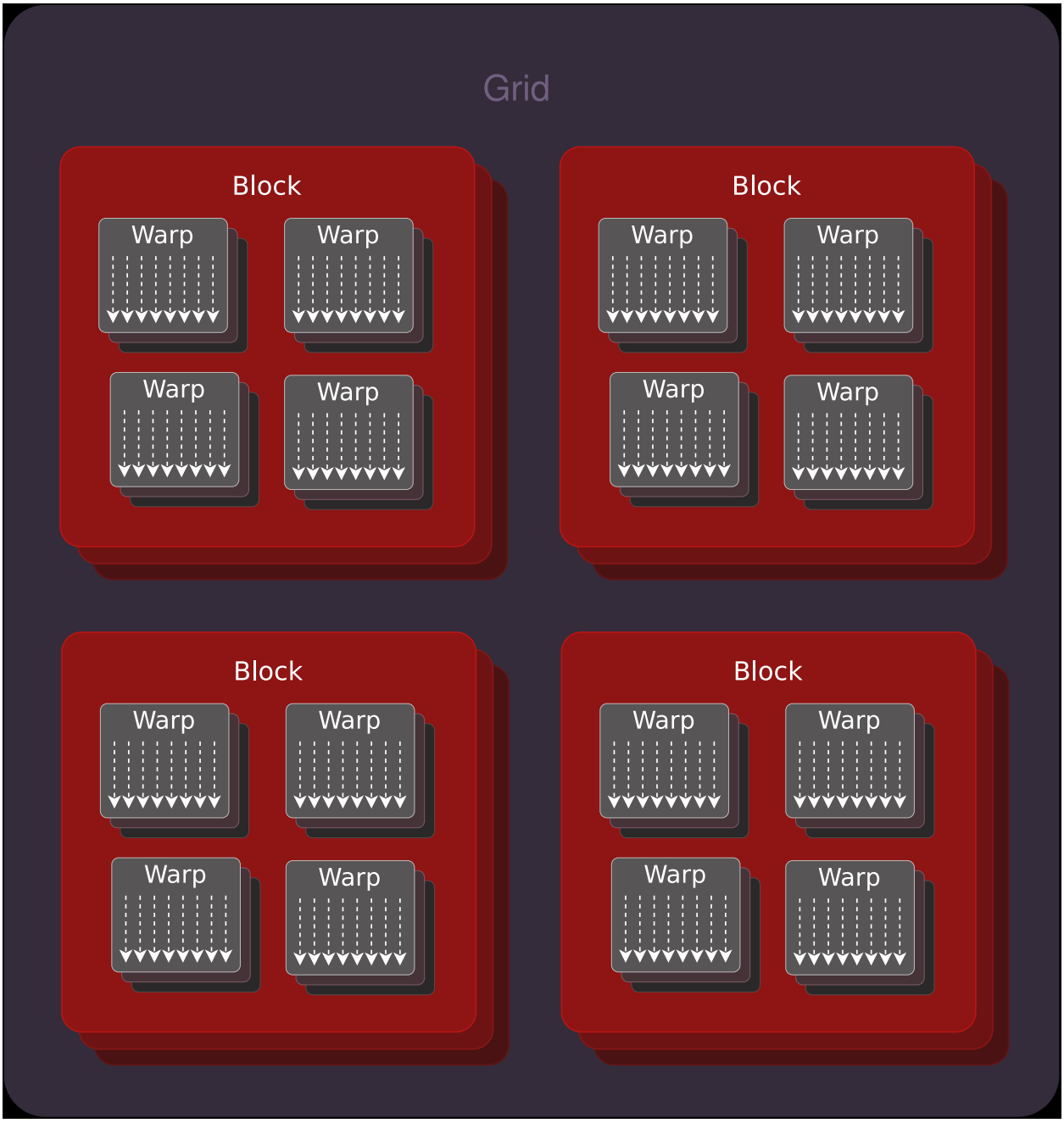


Fig. 4: Hierarchy of thread groups.

documentation such as AMDGPU ISA manuals and HSA specifications. Each thread represents an independent control flow with its own registers and program counter.

Threads execute the same kernel function independently, using identifiers such as `threadIdx.x` and `blockIdx.x` that determine which portion of data the thread operates on within its work group and grid. For example:

```
int tid = threadIdx.x;
int bid = blockIdx.x;
int global_idx = tid + bid * blockDim.x;
```

Each thread maintains its own register state, including vector general-purpose registers (VGPRs), and a private program counter. Threads have private storage for local variables, spilled registers, and function call stacks, which reside in global memory when on-chip register limits are exceeded. High-performance kernels minimize this usage by keeping data in registers or *Local Data Share (LDS)*.

A single SIMD lane executes the instructions for one thread at a time. Because each CU contains multiple SIMD units, thousands of threads can execute concurrently across the GPU. From a programmer's perspective, a thread is the fundamental unit of GPU computation, analogous to a CPU thread but scaled to tens of thousands of instances operating in parallel.

Warp (or Wavefront)

The innermost grouping of threads is called a warp (HIP terminology) or wavefront (AMD ISA terminology). A warp is the most tightly coupled group of threads, both physically and logically. Threads within a warp are executed in lockstep, with each thread executing the same instruction simultaneously on different data elements.

A warp represents the fundamental execution unit of AMD GPUs. Each warp consists of multiple parallel threads that execute the same instruction simultaneously across the SIMD pipelines of a *compute unit*. Threads in a warp are also called lanes, and the value identifying them is the lane ID.

> **Tip**
>
> Lane IDs aren't queried like other thread IDs, but are user-calculated. As a consequence, they are only as multidimensional as the user interprets the calculated values to be.

The size of a warp is architecture dependent and always fixed:

- **64 threads** for CDNA architectures
- **32 threads** for RDNA architectures

Warps are signified by the set of communication primitives at their disposal, as discussed in *Warp cross-lane functions*. On modern AMD datacenter GPUs, the number of resident warps per CU is limited by architectural wave slots and available resources (such as registers and LDS). For example, if a CU supports 64 resident warps, each containing 64 threads, this would correspond to 4,096 threads in flight; actual limits are architecture-specific and described in the hardware implementation documentation.

When a warp issues an instruction whose operands are not yet ready, for instance, a global memory load from *HBM*, it becomes stalled. Rather than idling, the warp scheduler selects another ready warp to execute. This rapid context switching hides memory and instruction latency and is key to achieving high utilization.

To keep the compute units busy, you should maximize occupancy, the number of resident warps per CU, ensuring there is always at least one eligible warp ready to issue instructions. The ratio of active issue cycles to total cycles is known as issue efficiency.

Block (Work-group)

The next level of the thread hierarchy is called a thread block (or work-group in OpenCL terminology). A block is a collection of warps that can synchronize and share local data share (LDS) memory. The defining feature of a

block is that all threads in the block have shared memory that they can use to share data or synchronize with one another, as described in *Memory model*.

All warps of a block execute on the same CU, ensuring they can access the same LDS and synchronize efficiently. This locality is crucial for performance when threads need to cooperate on shared data.

Threads within a work-group can coordinate with one another through barriers and the shared *Local Data Share (LDS)*. Because LDS resides in the CU's L1 cache subsystem, this cooperation is extremely fast. Threads in different work-groups cannot synchronize directly and must communicate through global memory using atomics or fences.

The size of a block, or the block dimension, is the user-configurable number of threads per block, but is limited by the queryable capabilities of the executing hardware. Common limits include:

- Maximum threads per block: typically 1024
- Maximum block dimensions: 1024 x 1024 x 64 (x, y, z)
- Limited by available resources (registers, LDS, warp slots)

The unique ID of the thread within a block can be 1, 2, or 3-dimensional as provided by the HIP API. You can configure the thread block to best represent the data associated with the kernel instruction set.

> **Note**
>
> When linearizing thread IDs within a block, assume the *fast index* is the `x` dimension, followed by the `y` and `z` dimensions.

Grid

The top-most level of the thread hierarchy is a grid. A grid represents the total collection of blocks (work-groups) launched for a single kernel execution. It defines the overall problem size and how work is distributed across the GPU.

When a kernel is launched, the host specifies the grid's dimensions, the number of work-groups, and the number of threads per work-group. Each thread within the grid is assigned a unique index derived from its position within both its work-group and the global grid, allowing threads to cooperatively process large datasets.

All work-groups in a grid execute the same kernel code, but on different portions of input data. The scheduling of work-groups is handled dynamically by the GPU driver and hardware scheduler, distributing them across available compute units. Execution order is non-deterministic, work-groups may execute concurrently or in arbitrary order depending on hardware availability and resource usage.

Work-groups within a grid cannot synchronize directly through barriers, since they may execute on different CUs. Instead, they coordinate via global memory, often using atomic operations or device-wide synchronization mechanisms. The kernel itself returns only after all work-groups in the grid have finished execution.

This grid-level abstraction allows a single kernel launch to scale transparently with GPU size: larger GPUs simply execute more work-groups concurrently, while smaller GPUs schedule them in more waves.

The grid is specified when launching a kernel and determines:

- Total number of threads: `grid_size` × `block_size`
- Distribution of work across compute units
- Overall parallelism of the computation

The unique ID of each block within a grid can be 1, 2, or 3-dimensional, as provided by the API, and is accessible to every thread in the block through the `blockIdx` built-in variable.

Grid dimensions are limited by queryable hardware capabilities.

The three-dimensional thread hierarchy available to a kernel program lends itself to solutions that align closely to the computational problem. The following are some examples:

- 1-dimensional: array processing, linear data structures, or sequential data transformation
- 2-dimensional: Image processing, matrix operations, 2 dimensional simulations
- 3-dimensional: Volume rendering, 3D scientific simulations, spatial algorithms

5.3.2 Cooperative groups thread model

The Cooperative groups API introduces new functions to launch, group, subdivide, synchronize and identify threads, as well as some predefined group-collective algorithms. Cooperative groups let you define your own set of thread groups which may fit your use-cases better than those defined by the hardware. It relaxes some restrictions of the *Hierarchical thread model* imposed by the strict 1:1 mapping of architectural details to the programming model.

> **Note**
>
> The implicit groups defined by kernel launch parameters are still available when working with cooperative groups.

For further information, see Cooperative groups.

5.4 Memory model

The GPU memory architecture is designed to support parallel execution across the thread hierarchy. Understanding the following memory spaces and their relationships to thread groupings is crucial for efficient GPU programming. The choice of memory type and access patterns significantly impacts kernel performance.

HIP's memory hierarchy consists of three main levels that mirror the GPU's physical structure:

- **Private registers**: Per-thread storage with the fastest access
- **Local Data Share (LDS)**: Per-work-group shared memory with low latency
- **High-Bandwidth Memory (HBM)**: Device-wide global memory with high capacity

This hierarchy gives you precise control over data locality and synchronization, allowing compute-bound kernels to efficiently exploit the massively parallel architecture. The following figure summarizes the memory namespaces and how they relate to the various levels of the threading model.

Fig. 5: Memory hierarchy.

Registers or per-thread memory

Read-write storage only visible to the threads defining the given variables, also called per-thread memory. This is the default memory namespace.

Each thread uses registers to store temporary variables, operands, and intermediate results during execution. These registers physically reside in the register file of a *compute unit* and are implemented using fast on-chip SRAM, typically the fastest accessible memory in the GPU.

When a thread requires more storage than the available registers, some values may spill into global memory, which incurs a significant performance penalty due to much higher latency. Registers are managed automatically by the LLVM-based ROCm compiler toolchain during kernel compilation, optimizing register allocation to minimize spills and maximize the number of threads that can run concurrently on a CU.

The size of the blocks for a given kernel, and thereby the number of concurrent warps, are limited by register usage. This relates to the *occupancy* of the CU as described in Compute Units, an important concept in resource usage and performance optimization.

Use registers when the data is specific to a thread. This includes temporary variables, loop counters, and intermediate results that don't need to be shared across threads. Registers provide the lowest latency access for thread-specific data.

Shared memory

Read-write storage visible to all the threads in a given block.

Shared memory corresponds to the *Local Data Share (LDS)*, a fast, on-chip SRAM region within each compute unit's L1 cache subsystem. LDS provides low-latency, programmer-managed shared memory that enables efficient coordination and reuse of data.

A typical high-performance kernel follows this pattern:

- Load data from global memory into LDS
- Perform arithmetic operations using SIMD or *Matrix Cores (MFMA units)*
- Synchronize threads within the work-group via barrier instructions
- Write results back to global memory, optionally using atomics for inter-work-group coordination

Each CU provides a fixed amount of LDS (the size depending on the architecture), shared among all resident work-groups. How much LDS a kernel consumes directly influences occupancy, since larger allocations reduce the number of active work-groups per CU.

Use shared memory when the data is reused within a thread block, when cross-thread communication is needed, or to minimize global memory transactions by using on-chip memory whenever possible.

Global

Read-write storage visible to all threads in a given grid. There are specialized versions of global memory with different usage semantics which are typically backed by the same hardware storing global.

Global memory serves as the main data store for input tensors, intermediate results, and kernel outputs. On AMD GPUs, global memory is physically implemented in *High-Bandwidth Memory (HBM)* or GDDR memory, depending on the product class.

Access to global memory is explicit and programmer-managed, with synchronization possible through atomic operations or memory fences. Data transfers between the host and device are managed using HIP runtime APIs such as `hipMalloc()`, `hipMemcpy()`, and `hipFree()`. Performance is largely determined by memory coalescing, alignment, and reuse.

Use global memory when you have large datasets, are transferring memory between the host and the device, and when you are sharing data between thread blocks.

Constant

Read-only storage visible to all threads in a given grid. It is a limited segment of global with queryable size. Use constant memory for read-only data that is shared across multiple threads, and that has a small data size.

Texture

Read-only storage visible to all threads in a given grid and accessible through additional APIs.

Surface

A read-write version of texture memory.

5.4.1 Memory optimizations and best practices

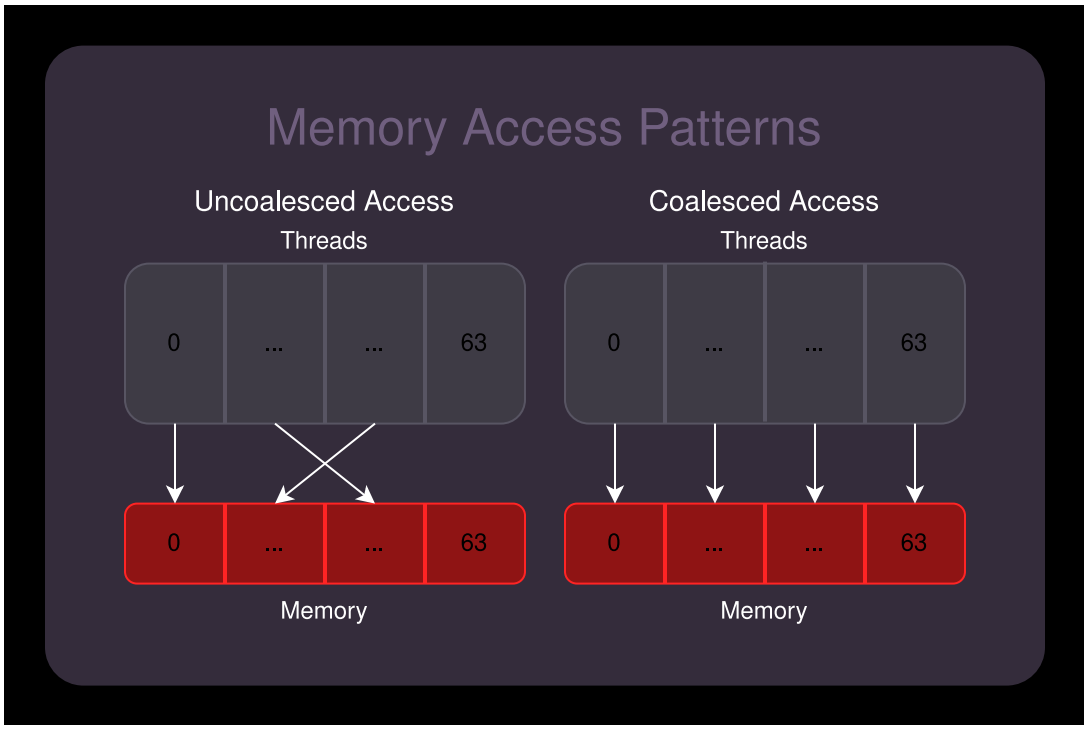


Fig. 6: Coalesced memory accesses

The following are a few memory access patterns and best practices to improve performance. You can find additional information in *Memory management* and ../how-to/performance_guidelines.

- **Global memory**: Coalescing reduces the number of memory transactions.

  Coalesced memory access in HIP refers to the optimization of memory transactions to maximize throughput when accessing global memory. When a kernel accesses global memory, the memory transactions typically occur in chunks of 32, 64, or 128 bytes, which must be naturally aligned. Coalescing memory accesses means aligning and organizing these accesses so that multiple threads in a warp can combine their memory requests into the fewest possible transactions. If threads access memory in a coalesced manner, meaning consecutive threads read or write consecutive memory locations, the memory controller can merge these accesses into a single transaction. This is crucial because global memory bandwidth is relatively low compared to on-chip bandwidths, and non-optimal memory accesses can significantly impact performance. If all the threads in a warp can access consecutive memory locations, memory access is fully coalesced.

  To achieve coalesced memory access in HIP, you should:

  1. *Align Data*: Use data types that are naturally aligned and ensure that structures and arrays are aligned properly.
  2. *Optimize Access Patterns*: Arrange memory accesses so that consecutive threads in a warp access consecutive memory locations. For example, if threads access a 2D array, the array and thread block widths should be multiples of the warp size.
  3. *Avoid strided access*: For example array[i * stride] can lead to memory bank conflicts and inefficient access.
  4. *Pad Data*: If necessary, pad data structures to ensure alignment and coalescing.

- **Shared memory**: Avoiding bank conflicts reduces the serialization of memory transactions.

  Shared memory is a small, fast memory region inside the CU. Unlike global memory, shared memory accesses do not require coalescing, but they can suffer from bank conflicts, which are another form of inefficient memory

access. Shared memory is divided into multiple memory banks (usually 32 banks on modern GPUs). If multiple threads within a warp try to access different addresses that map to the same memory bank, accesses get serialized, leading to poor performance. To optimize shared memory usage, ensure that consecutive threads access different memory banks. Use padding if necessary to avoid conflicts.

- **Texture memory**: Spatial locality improves caching performance.

  Texture memory is read-only memory optimized for spatial locality and caching rather than coalescing. Texture memory is cached, unlike standard global memory, and it provides optimized access patterns for 2D and spatially local data. Accessing neighboring values results in cache hits, improving performance. Therefore, instead of worrying about coalescing, optimal memory access patterns involve ensuring that threads access spatially adjacent texture elements, and the memory layout aligns well with the 2D caching mechanism.

- **Unified memory**: Structured access reduces the overhead of page migrations.

  Unified memory allows the CPU and GPU to share memory seamlessly, but performance depends on access patterns. Unified memory enables automatic page migration between CPU and GPU memory. However, if different threads access different pages, it can lead to expensive page migrations and slow throughput performance. Accessing unified memory in a structured, warp-friendly manner reduces unnecessary page transfers. Ensure threads access memory in a structured, consecutive manner, minimizing page faults. Prefetch data to the GPU before computation by using `hipMemPrefetchAsync()`. In addition, using small batch transfers as described below, can reduce unexpected page migrations when using unified memory.

- **Small batch transfers**: Enable pipelining and improve PCIe bandwidth use.

  Memory transfers between the host and the device can become a major bottleneck if not optimized. One method is to use small batch memory transfers where data is transferred in smaller chunks instead of dealing with large datasets to avoid long blocking operations. Small batch transfers offer better PCIe bandwidth utilization over large data transfers. Small batch transfers offer performance improvement by offering reduced latency with small batches that run asynchronously using `hipMemcpyAsync()` as described in *Asynchronous concurrent execution*, pipelining data transfers and kernel execution using separate streams. Finally, using pinned memory with small batch transfers enables faster DMA transfers without CPU involvement, greatly improving memory transfer performance.

5.5 Execution model

As previously discussed in *Heterogeneous programming*, HIP programs consist of two distinct scopes:

- The host-side API running on the host processor.
- The device-side kernels running on GPUs.

Both the host and the device-side APIs have synchronous and asynchronous functions.

5.5.1 Host-side execution

The host-side API dealing with device management and their queries are synchronous. All asynchronous APIs, such as kernel execution, data movement and potentially data allocation/freeing all happen in the context of device streams, as described in Managing streams.

Streams are FIFO buffers of commands to execute relating to a given device. Operations that enqueue tasks on a stream all return promptly, and the command is executed asynchronously. All side effects of a command on a stream are visible to all subsequent commands on the same stream. Multiple streams may point to the same device and those streams may be fed from multiple concurrent host-side threads. Execution on multiple streams may be concurrent but isn't required to be.

Asynchronous APIs involving a stream all return a stream event, which can be used to synchronize the execution of multiple streams. A user may enqueue a barrier onto a stream referencing an event. The barrier will block activity on the stream until the operation related to the event completes. After the event completes, all side effects of the operation will be visible to subsequent commands even if those side effects manifest on different devices.

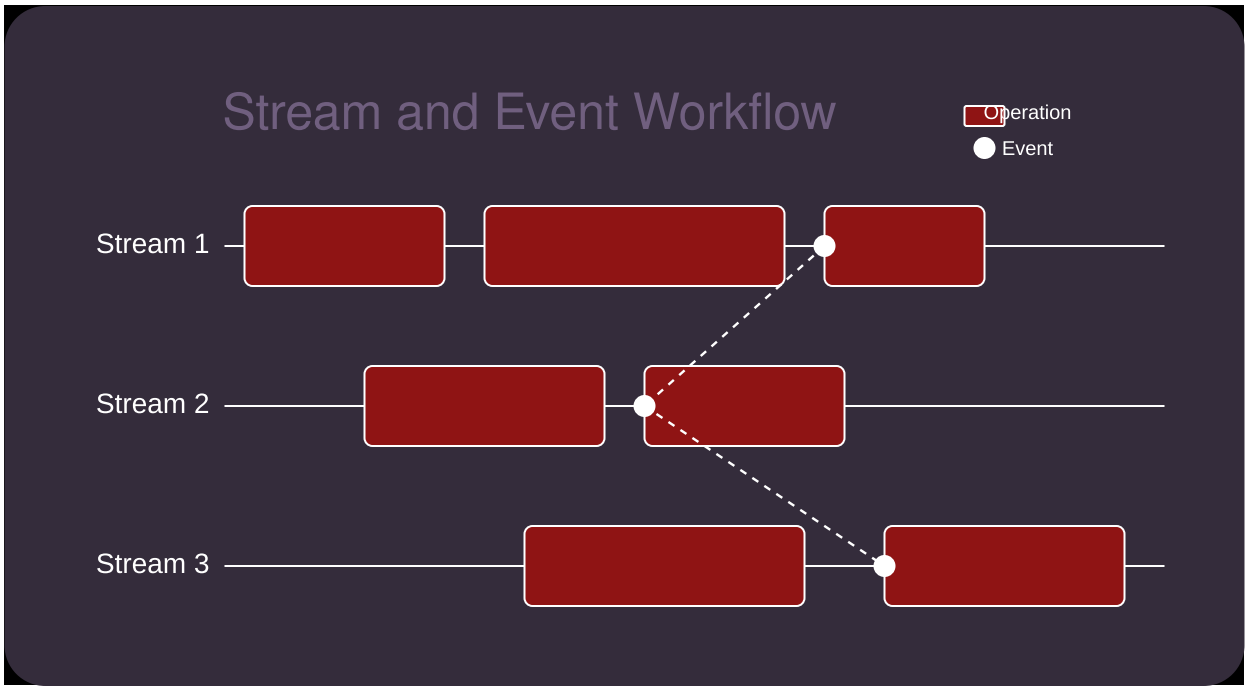


Fig. 7: Multiple stream workflow

Streams also support executing user-defined functions as callbacks on the host. The stream will not launch subsequent commands until the callback completes.

5.5.2 Device-side execution

Kernels may be launched in multiple ways, all with different syntaxes and intended use cases.

- Using the triple-chevron `<<<...>>>` operator on a `__global__` annotated function.
- Using `hipLaunchKernelGGL()` on a `__global__` annotated function.

> **Tip**
>
> This name, by default, is a macro expanding to the triple-chevron syntax. In cases where language syntax extensions are undesirable, or where launching templated and/or overloaded kernel functions define the `HIP_TEMPLATE_KERNEL_LAUNCH` preprocessor macro before including the HIP headers to turn it into a templated function.

5.5.3 Asynchronous execution

Asynchronous operations between the host and the kernel provide a variety of opportunities, or challenges, for managing synchronization, as described in *Asynchronous concurrent execution*. For instance, a basic model would be to launch an asynchronous operation on a kernel in a stream, create an event to track the operation, continue operations in the host program, and when the event shows that the asynchronous operation is complete, synchronize the kernel to return the results.

However, one of the opportunities of asynchronous operation is the pipelining of operations between launching kernels and transferring memory. In this case, you would be working with multiple streams running concurrently, or at least overlapping in some regard, and managing any dependencies between the streams in the host application. The producer-consumer paradigm can be used to convert a sequential program into parallel operations to improve performance. This

process can employ multiple streams to kick off asynchronous kernels, provide data to the kernels, perform operations, and return the results for further processing in the host application.

These asynchronous activities call for stream management strategies. In the case of the single stream, the only management would be the stream synchronization when the work was complete. However, with multiple streams you have overlapping execution of operations and synchronization becomes more complex, as shown in the variations of the example in Programmatic dependent launch and synchronization. You need to manage each stream's activities, evaluate the availability of results, evaluate the critical path of the tasks, allocate resources on the hardware, and manage the execution order.

5.5.4 Multi-GPU and load balancing

For applications requiring additional computational power beyond a single device, HIP supports utilizing multiple GPUs within a system. Large-scale applications that need more compute power can use multiple GPUs in the system. This enables the runtime to distribute workloads across multiple GPUs to balance the load and prevent some GPUs from being over-utilized while others are idle.

For more information, see multi-device.

5.6 Domain-specific programming models

Beyond the low-level kernel-based programming model described in previous sections, ROCm provides domain-specific library abstractions that offer alternative ways to express GPU computation. These libraries encapsulate common computational patterns, providing higher-level programming models optimized for specific problem domains.

Rather than writing explicit kernels with manual memory management and optimization, developers can express algorithms using domain-appropriate operations. The libraries handle kernel selection, memory layout, and performance tuning automatically, adapting to different GPU architectures and problem sizes at runtime.

This section presents two representative examples: rocBLAS for linear algebra and MIOpen for deep learning. These libraries illustrate how domain-specific abstractions can improve productivity while maintaining performance portability across AMD GPU architectures.

5.6.1 Linear algebra with rocBLAS

rocBLAS implements the BLAS standard, providing a high-performance library for fundamental dense linear algebra operations. It represents an alternative programming model where computation is expressed as matrix and vector operations rather than explicit GPU kernels.

5.6.1.1 Programming model characteristics

The rocBLAS programming model differs from kernel-level HIP in several key aspects:

- **Abstraction level**: Developers call functions like `rocblas_sgemm()` for matrix multiplication instead of writing tiled kernels manually
- **Memory layout**: Matrices are conceptualized as mathematical objects with rows and columns, not raw pointers and index arithmetic
- **Automatic optimization**: The library selects optimal kernel implementations based on problem dimensions, data types (FP64, FP32, FP16, BF16, INT8), and target GPU architecture (for example, `gfx90a`, `gfx942`)
- **Performance portability**: The same API call adapts to CDNA, RDNA, and future architectures without code changes

Instead of managing thread blocks, shared memory tiles, and register allocation, the developer specifies the mathematical operation. rocBLAS handles all low-level optimizations, including memory coalescing, use of *Matrix Cores (MFMA units)*, and exploitation of on-chip *LDS*.

5.6.1.2 Column-major versus row-major layouts

One practical consideration when using rocBLAS is matrix layout. To maintain compatibility with the original Fortran-based BLAS specification, rocBLAS expects matrices in column-major order. Most C and C++ programs, including HIP kernels, use row-major order by default.

This mismatch can be resolved mathematically without physically transposing data. Given the identity:

$$\text{If } \boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}, \text{ then } \boldsymbol{C}^T = \boldsymbol{B}^T \times \boldsymbol{A}^T$$

By swapping the operand order and dimensions when calling rocBLAS, and interpreting the result as a row-major matrix, the correct result is obtained without memory rearrangement:

```
#include <rocblas/rocblas.h>

// Matrix multiply C = alpha * A @ B + beta * C
// for row-major matrices using rocBLAS
void sgemm_row_major(rocblas_handle handle, int M, int N, int K,
                     const float* alpha,
                     const float* A, const float* B,
                     const float* beta, float* C) {
    // Swap A and B, swap M and N for row-major layout
    rocblas_sgemm(handle,
                  rocblas_operation_none, rocblas_operation_none,
                  N, M, K,
                  alpha,
                  B, N,  // leading dimension of B^T
                  A, K,  // leading dimension of A^T
                  beta,
                  C, N); // leading dimension of C^T
}
```

This technique preserves both correctness and performance. The `rocblas_operation_none` flag indicates matrices should be used as provided, avoiding additional device-side transposes that would harm performance.

5.6.1.3 Integration with HIP applications

rocBLAS integrates with HIP applications through a handle-based API. The library manages its own stream-based execution model, coordinating with HIP's asynchronous operations:

```
rocblas_handle handle;
rocblas_create_handle(&handle);

// Optional: associate with HIP stream
rocblas_set_stream(handle, hipStream);

// Perform computation
rocblas_sgemm(handle, ...);

// Synchronize if needed
hipStreamSynchronize(hipStream);

rocblas_destroy_handle(handle);
```

Applications link against `librocblas.so` and include `rocblas.h`. The library serves as the foundation for deep learning frameworks such as PyTorch and TensorFlow when running on AMD GPUs, handling dense matrix operations in fully connected and transformer layers.

5.6.2 Deep learning with MIOpen

MIOpen provides GPU-accelerated primitives for deep neural networks, offering a programming model centered on neural network operations rather than explicit kernels. It occupies the same software layer as rocBLAS but is specialized for deep learning workloads.

5.6.2.1 Programming model characteristics

MIOpen abstracts GPU programming to the level of neural network layers and operations:

- **Domain expressiveness**: Computations are expressed as convolutions, normalizations, activations, and attention mechanisms, not thread hierarchies
- **Declarative fusion**: Operations can be combined (Convolution → Bias → ReLU) into fused kernels automatically
- **Automatic algorithm selection**: The library chooses between direct, FFT-based, Winograd, and GEMM-based implementations at runtime
- **Memory optimization**: Fused operations keep intermediate results in *LDS*, minimizing *global memory* traffic

Rather than manually implementing convolutional kernels with careful attention to memory coalescing and *MFMA unit* utilization, developers describe the neural network architecture. MIOpen handles low-level optimization, auto-tuning, and performance-critical implementation details.

5.6.2.2 Graph and Fusion APIs

Modern MIOpen workflows use Graph and Fusion APIs, which allow declarative specification of operation sequences:

```
// Declarative fusion example (conceptual)
auto conv = createConvolution(input, weights, ...);
auto bias = createBiasAdd(conv, bias_vector);
auto relu = createActivation(bias, ACTIVATION_RELU);

// Library fuses into single optimized kernel
auto fused_op = fuseOperations({conv, bias, relu});
executeGraph(fused_op);
```

Operation fusion significantly improves performance by reducing memory bandwidth pressure. Instead of three separate kernel launches with intermediate global memory writes and reads, a single fused kernel keeps data on-chip throughout the computation pipeline.

This is particularly important for operations like batch normalization followed by activation, where intermediate tensors would otherwise require expensive round-trips through *HBM*. Keeping these values in LDS reduces memory traffic and improves energy efficiency.

5.6.2.3 Runtime optimization and tuning

MIOpen maintains multiple implementations for each operation. For convolutions, this includes:

- **Direct convolution**: Explicit loops over kernel dimensions
- **GEMM-based**: Reformulates convolution as matrix multiplication
- **Winograd**: Reduces arithmetic complexity for small kernels
- **FFT-based**: Frequency-domain convolution for large kernels

The library uses runtime heuristics and pre-tuned configuration databases to select the optimal algorithm based on:

- Input tensor dimensions and layout
- Filter sizes and stride patterns

- Target GPU architecture (CDNA versus RDNA, specific `gfx` version)
- Available memory and compute resources
- Data types (FP32, FP16, BF16, INT8)

This auto-tuning abstracts performance optimization from the application developer. The same MIOpen call automatically adapts when moving from a datacenter MI300X (`gfx942`) to a consumer RDNA3 GPU (`gfx1200`), selecting architecture-appropriate implementations without code changes.

5.6.2.4 Framework integration

Deep learning frameworks like PyTorch and TensorFlow use MIOpen as their GPU backend on AMD hardware. When a PyTorch model calls `torch.nn.Conv2d` or `torch.nn.BatchNorm2d`, these operations dispatch to MIOpen primitives.

The framework provides the high-level neural network API, while MIOpen handles the GPU-specific implementation. This separation allows framework developers to focus on model architectures and training algorithms, while library developers optimize GPU kernels for specific hardware generations.

For applications requiring both general linear algebra and specialized deep learning operations, rocBLAS and MIOpen work together. MIOpen uses rocBLAS (via hipBLASLt and Tensile backends) for the matrix-multiply-intensive portions of operations like fully connected layers, while providing specialized kernels for convolutions, pooling, and normalization.

5.6.3 Comparison of programming models

The following table summarizes how different programming models trade control for abstraction:

Table 1: Programming Model Comparison

Aspect	HIP Kernels	Domain-Specific Libraries
Programming unit	Thread, block, grid	Domain primitives (matrices, layers, transforms)
Memory management	Explicit (`hipMalloc`, `hipMemcpy`)	Library-managed, handle-based
Optimization approach	Manual tuning required	Auto-tuned, architecture-adaptive
Thread hierarchy	Explicit configuration	Abstracted by API
Performance portability	Requires retuning per architecture	Automatic adaptation across architectures
Development effort	High (low-level control)	Low (high-level API calls)

The examples of rocBLAS and MIOpen illustrate a common pattern across the ROCm software stack: domain-specific libraries that abstract low-level GPU programming into higher-level operations tailored to specific computational domains. The ROCm ecosystem provides many additional libraries following this pattern, including rocFFT for Fast Fourier Transforms, rocRAND for random number generation, rocSPARSE for sparse linear algebra, and RCCL for collective communication across multiple GPUs.

Choosing the appropriate abstraction level depends on the application domain. For novel algorithms not covered by existing libraries, kernel-level HIP provides full control. For well-established operations within specific domains, the ROCm library ecosystem offers productivity and performance portability.

Many applications combine multiple programming models: HIP kernels for custom algorithms, domain-specific libraries for standard operations, and framework integration for high-level workflows. The ROCm software stack supports this layered approach, allowing developers to select the appropriate abstraction for each component of their application.

For a complete list of ROCm libraries and their capabilities, see the ROCm documentation.

CHAPTER

SIX

HARDWARE IMPLEMENTATION

This topic describes the hardware architecture of AMD GPUs supported by HIP, focusing on the internal organization and operation of GPU hardware components. Understanding these hardware details helps you optimize GPU applications and achieve maximum performance.

6.1 Overall GPU architecture

AMD GPUs consist of interconnected blocks of digital circuits that work together to execute complex parallel computing tasks. Unlike central processing units (CPUs), which dedicate significant silicon area to instruction flow control, branch prediction, and complex caching hierarchies, GPUs allocate the majority of their die area to arithmetic pipelines. This design choice enables extreme throughput density for data-parallel workloads. The architecture is organized hierarchically to enable massive parallelism while efficiently managing resources.

6.1.1 Command processor and control

The command processor (CP) serves as the primary interface between the CPU and GPU, receiving and distributing commands for execution. The CP consists of two main components:

- **Command processor fetcher (CPF)**: Fetches commands from memory and passes them to the command processor packet processor (CPC) for processing.
- **Command processor packet processor (CPC)**: A microcontroller that decodes the fetched commands and dispatches kernels to the workgroup processors for scheduling.

The command processor handles several types of operations:

- Kernel launches, which are forwarded to asynchronous compute engines (ACEs)
- Memory transfers, which are delegated to direct memory access (DMA) engines
- Synchronization operations and memory fences

DMA engines handle memory transfers between CPU and GPU memory without CPU involvement after initialization. Most GPUs contain two DMA engines, enabling concurrent bidirectional transfers to better utilize PCIe bandwidth. The DMA engines fetch data in small chunks and can process transfers in parallel but cannot handle multiple copy commands on the same engine simultaneously.

Asynchronous compute engines (ACEs) break down kernels into workgroups for distribution to shader processor input (SPI) blocks. Multiple ACEs enable concurrent kernel execution, with each ACE capable of dispatching one kernel at a time. ACEs process commands from different queues asynchronously, enabling overlap between different kernel executions and memory operations.

6.1.2 Hierarchical organization

The GPU organizes compute resources in a three-level hierarchy that enables modular design and resource sharing:

1. **Shader engines (SE)**: Top-level organizational units containing multiple shader arrays and shared resources
2. **Shader arrays**: Groups of compute units (CUs) sharing instruction and scalar caches
3. **Compute units (CU)**: Basic execution units containing the arithmetic logic units (ALUs) and registers for thread execution

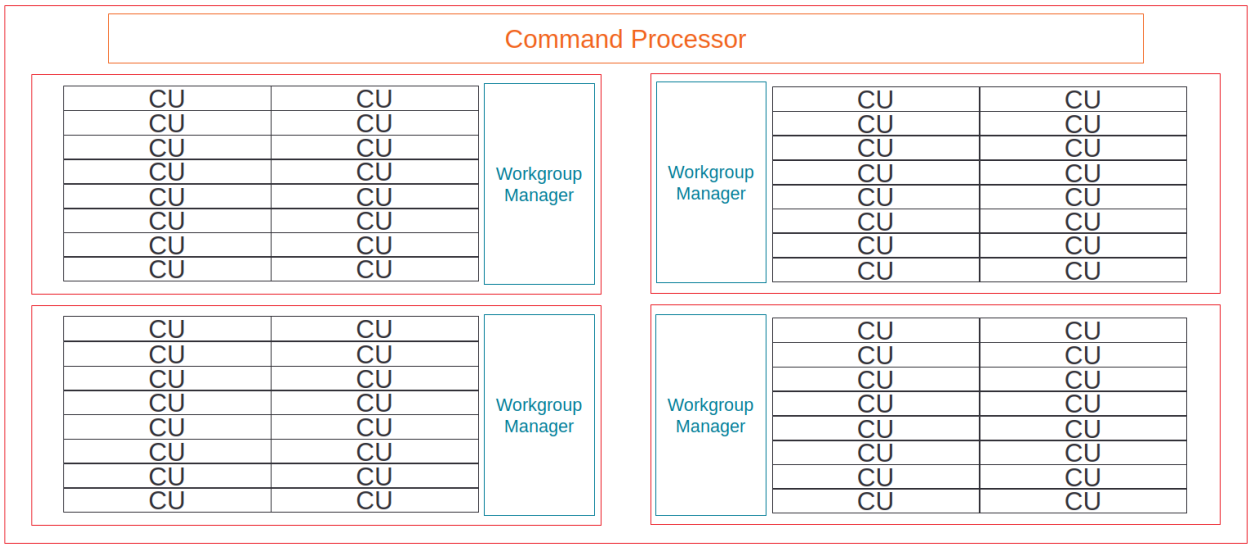


Fig. 1: Hierarchical organization of compute units into shader engines

This hierarchical design allows different GPU configurations using the same underlying architecture.

6.2 Shader engine components

Shader engines group multiple compute units together, sharing resources to improve efficiency and reduce redundancy. Each shader engine contains several key components shared across its compute units.

6.2.1 Workgroup manager (SPI)

The workgroup manager, also called the shader processor input (SPI), bridges the command processor and compute units. After the CP processes a kernel dispatch, the SPI:

- Receives workgroups from the ACEs
- Schedules workgroups onto available compute units
- Initializes registers with kernel parameters
- Ensures all warps of a workgroup execute on the same CU for synchronization
- Monitors resource availability and queues workgroups when resources are exhausted

The SPI tracks four critical resources that limit concurrent execution:

- warp slots (execution contexts)
- Vector general-purpose registers (VGPRs)
- Scalar general-purpose registers (SGPRs)

- Local data share (LDS) memory

Workgroup-to-CU mapping is non-deterministic and based on available resources. You should not assume any specific mapping pattern, as the same kernel launched multiple times can have different workgroup distributions.

6.2.2 Scalar L1 data cache (sL1D)

The scalar L1 data cache (sL1D) serves scalar memory operations from multiple CUs within a shader array. The sL1D is shared between CUs and caches data that is uniform across a warp, including:

- Kernel arguments and pointers
- Grid and block dimensions
- Constants accessed uniformly across threads
- Data from `__constant__` memory when accessed uniformly

Unlike the vector L1 cache, the sL1D doesn't use a "hit-on-miss" approach, meaning subsequent requests to the same pending cache line count as duplicated misses rather than hits.

6.2.3 L1 instruction cache (L1I)

The L1 instruction cache (L1I) is a read-only cache shared between multiple CUs in a shader array. Like the sL1D, it's backed by the L2 cache and doesn't use the "hit-on-miss" approach. The L1I stores kernel instructions fetched by the compute units, reducing instruction fetch latency and L2 cache pressure.

6.3 Compute unit architecture

The compute unit (CU) is the fundamental execution block of AMD GPUs, serving as the atomic building block for massive parallelism. Each CU is responsible for executing kernels through its various specialized components and pipelines. Data flows into these pipelines, undergoes arithmetic transformation, and exits as results, to maximize the number of such transformations per clock cycle.

CUs enable latency hiding through massive hardware multithreading. A single CU can manage thousands of concurrent threads organized as a number of warps, each containing 32 (RDNA) or 64 (CDNA) threads. This massive concurrency allows the hardware to hide memory access latency by executing other warps while some wait for data.

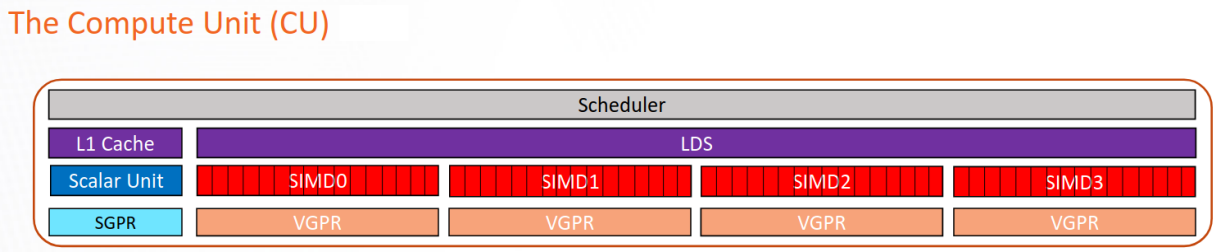


Fig. 2: Internal architecture of an AMD CDNA compute unit

6.3.1 Sequencer and scheduling

The instruction sequencer (SQ) serves as the control center of each compute unit, managing instruction flow through the execution pipelines. The sequencer maintains warp state and coordinates instruction execution across different functional units.

Warp organization: The sequencer organizes active warps into four pools, each containing slots for up to ten warps (eight on the CDNA2 MI200 Series). Each slot includes:

- Warp-level registers (program counter, execution mask, and others)
- Instruction buffer for prefetched instructions
- State information for scheduling decisions

This organization theoretically allows up to 40 concurrent warps per CU, though actual occupancy is typically limited by register and LDS usage.

Instruction fetching: The fetch arbiter selects one warp per cycle to fetch instructions from memory, prioritizing the oldest warps. Each CU can fetch up to 32 bytes (4-8 instructions) per cycle.

Instruction issuing: The issue arbiter determines which instructions execute each cycle, selecting warps from one pool per cycle in round-robin fashion. The arbiter can issue multiple instructions per cycle to different execution units, with a theoretical maximum of five instructions per cycle:

- One VALU instruction
- One vector memory operation
- One SALU and/or scalar memory operation
- One LDS operation
- One branch operation

Instructions always issue at warp granularity, with all threads in the warp executing the same instruction in lockstep. The hardware can perform single-cycle context switching between warps with zero overhead, as all warp contexts remain resident on the CU. This enables efficient latency hiding, allowing the CU to switch to another warp immediately when the current warp encounters a stall condition such as a memory access.

6.3.2 Execution pipelines

Each CU contains multiple specialized execution pipelines that process different types of instructions in parallel, enabling efficient utilization of the hardware resources.

6.3.2.1 Vector arithmetic logic unit (VALU)

The VALU executes vector instructions across entire warps, with each thread potentially operating on different data. For CDNA architectures, the VALU consists of:

- **Four SIMD processors**: Each containing 16 single-precision ALUs (or equivalent), for 64 total ALUs per CU. In CDNA3, these are SIMD64 pipelines that can execute 256 operations per cycle per CU.
- **Vector register files**: 256-512 KiB of VGPR storage split across the four SIMDs. VGPRs are organized as 32-bit lanes, providing flexibility for mixed-precision computations.
- **Instruction buffers**: Storage for up to 8-10 warps per SIMD

On architectures with 64-thread warps and 16-instruction wide SIMD units, executing one instruction takes four cycles (one cycle per 16 threads). The four SIMD design ensures full utilization when sufficient warps are available, as a new instruction can issue to each SIMD every cycle.

The VALU serves as the primary arithmetic engine, executing the majority of computation in GPU kernels. Data flows into these pipelines, undergoes arithmetic transformation, and exits as results, with the goal of maximizing the number of such transformations per clock cycle.

For CDNA architectures with matrix operations, the VALU also dispatches matrix fused multiply-add (MFMA) instructions to specialized matrix units.

6.3.2.2 Register pressure and occupancy

Register usage directly impacts CU occupancy. Each warp requires a portion of the finite VGPR and SGPR pools. Higher register usage per thread reduces the maximum number of concurrent warps, potentially limiting the CU's ability to hide latency. Mixed-precision workloads can optimize register usage by storing lower-precision values in fewer registers.

6.3.2.3 Scalar arithmetic logic unit (SALU)

The SALU executes instructions uniformly across all threads in a warp, handling operations such as:

- Control flow (branches, loops)
- Address calculations
- Loading kernel arguments and constants
- Managing warp-uniform values

The SALU includes:

- A scalar processor for arithmetic and logic operations
- 12.5 KiB of SGPR storage per CU
- A scalar memory (SMEM) unit for memory operations

Scalar operations reduce pressure on vector units and registers by handling uniform computations efficiently.

6.3.2.4 Vector memory unit (VMEM)

The VMEM unit handles all vector memory operations, including loads, stores, and atomic operations. Each thread supplies its own address and data, though the hardware optimizes access through memory coalescing when threads access nearby addresses. The VMEM unit connects to the vector L1 cache and implements both address generation and coalescing logic.

6.3.2.5 Branch unit

The branch unit executes jumps and branches for control flow changes affecting entire warps. Note that the branch unit handles warp-level control flow, not execution mask updates for thread divergence, which are handled through predication.

6.3.2.6 Special function unit (SFU)

The special function units accelerate certain arithmetic operations that are too complex and/or costly to implement purely within the standard vector ALUs.

SFUs are responsible for executing transcendental and reciprocal mathematical functions, operations such as `exp`, `log`, `sin`, `cos`, `rcp` (reciprocal), and `rsqrt` (reciprocal square root). These are heavily used in scientific, physics, and machine learning workloads, particularly in activation functions such as GELU, sigmoid, and/or softmax.

Each CU includes a set of specialized pipelines and/or transcendental function units (TFUs) that handle these operations with dedicated hardware. While their throughput is lower than that of the primary SIMD pipelines, they enable these functions to execute efficiently without consuming general ALU bandwidth.

From the compiler's perspective, these operations map to specific AMDGPU ISA instructions, such as:

- `v_exp_f32` - compute exponential base e
- `v_log_f32` - compute natural logarithm
- `v_sin_f32`, `v_cos_f32` - compute sine and/or cosine
- `v_rsq_f32`, `v_rcp_f32` - compute reciprocal and/or reciprocal square root

In CDNA3-based GPUs (such as MI300), SFU throughput and latency have been tuned for deep learning primitives. For instance, exponentiation (`exp`) and logarithm (`log`) functions are now pipelined to complete in a few cycles per lane, allowing vectorized activation functions in large-scale matrix workloads to execute without significant stalls.

For programmers targeting ROCm and/or HIP, these SFU-accelerated operations are typically accessed through math intrinsics such as `__expf`, `__logf`, and/or `__sinf`, which the compiler lowers to the corresponding AMDGPU ISA instructions at compile time.

6.3.2.7 Load/store unit (LSU)

The load/store units handle the transfer of data between the compute units and the GPU's memory subsystems. They are responsible for issuing, tracking, and retiring memory operations, including loads from and stores to global memory, local shared memory, and caches, for thousands of concurrent threads.

Each CU includes a set of LSUs tightly integrated with its vector and scalar pipelines. These units handle memory instructions generated by active warps, such as `buffer_load`, `buffer_store`, and `flat_load_dword`, and route them through the GPU's hierarchical memory system.

The LSU's responsibilities include:

- Managing vector memory accesses for SIMD instructions
- Coordinating local data share (LDS) reads and writes
- Accessing the L0 and/or L1 caches and forwarding requests to the L2 cache and high-bandwidth memory (HBM)
- Handling synchronization and atomic operations between threads and workgroups

LSUs manage thousands of outstanding memory requests per GPU, dynamically scheduling them to hide memory latency. While arithmetic pipelines continue executing other warps, the LSUs maintain queues of pending transactions and reorder responses as data returns from memory.

6.3.2.8 Matrix fused multiply-add (MFMA)

CDNA architectures (MI100 and newer) include specialized matrix acceleration units for high-throughput matrix operations. These units execute independently from other VALU operations, allowing overlap between matrix and vector computations. MFMA units support various data types including `INT8`, `FP16`, `BF16`, and `FP32`, with different throughput characteristics for each.

Matrix cores are GPU execution units that perform large-scale matrix operations in a single instruction. In AMD architectures, these units are formally known as MFMA (matrix fused multiply-add) units, the core hardware blocks responsible for accelerating deep learning, high-performance computing (HPC), and dense linear-algebra workloads on modern Instinct GPUs.

Operating on entire tiles of matrices per instruction allows MFMA units to deliver far greater arithmetic throughput and energy efficiency than scalar and/or vector ALUs. Rather than fetching and decoding thousands of per-element multiply-add instructions, each MFMA instruction processes an entire matrix fragment, drastically reducing power per operation and increasing overall throughput. The MFMA units implement a mini-systolic array design that efficiently processes matrix tiles.

An example MFMA instruction from the AMDGPU ISA is:

```
v_mfma_f32_16x16x4f16 v[0:15], v[16:31], v[32:47], v[0:15]
```

This instruction performs a matrix multiplication and accumulation $\boldsymbol{D} = \boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{C}$, where the fragments $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are stored in VGPRs. The suffix `16x16x4f16` indicates a tile size of 16×16, with an inner dimension of 4, operating on half-precision (FP16) inputs and accumulating into 32-bit floating-point outputs.

Programmers can access MFMA functionality at multiple levels: through optimized libraries, compiler intrinsics, and/or inline assembly, providing flexibility for different use cases.

The MFMA units use both standard VGPRs and additional accumulation VGPRs (AGPRs) on supported architectures, providing up to 512 KiB of combined register storage per CU.

6.3.2.9 Data movement engine (CDNA 3 / CDNA 4)

CDNA 3 and CDNA 4 architectures include specialized Data Movement Engine (DME) hardware units designed to accelerate access to multi-dimensional tensor data in GPU memory. DMEs perform high-throughput, low-overhead copies between global memory (HBM) and the on-chip memory hierarchy, particularly the Local Data Share (LDS) and L0 and/or L1 caches, without consuming compute resources.

A DME issues bulk memory transactions for contiguous and/or affine data regions, such as tensors laid out as multi-dimensional arrays in global memory. The hardware computes memory addresses for large block transfers in parallel, offloading this work from the SIMD pipelines and reducing pressure on both the register file and the instruction scheduler. This enables higher sustained bandwidth and lower latency for operations involving tiled matrix and/or tensor data.

In practice, DMEs handle transfers of the form addr $=$ stride $\times$ index $+$ offset across many threads and dimensions simultaneously. By performing these affine address calculations directly in hardware, the DME avoids the need for per-thread address arithmetic, freeing up scalar ALUs and registers for computation.

The DME design provides two key advantages:

- **Resource decoupling**: By removing large tensor copies from the main execution pipelines, the CU can continue executing arithmetic instructions while data movement occurs in the background.
- **Asynchronous execution model**: A single warp can issue a DME copy command, immediately resume computation, and later synchronize only when the transfer has completed. This enables producer-consumer parallelism.

Programmers can access this functionality through asynchronous copy intrinsics in ROCm, such as `__builtin_amdgcn_async_work_group_copy`, which map directly to hardware-level DME operations. These intrinsics allow explicit control over data transfer overlap, synchronization, and cache placement.

6.3.3 Local data share (LDS)

The local data share provides fast on-CU scratchpad memory for communication between threads in a workgroup.

Organization: The LDS contains 32 (CDNA, CDNA 2, and CDNA 3) or 64 (CDNA 4 and RDNA 2, RDNA 3, and RDNA 4) banks, each 4-bytes wide, providing 128 (CDNA, CDNA 2, and CDNA 3) or 256 (CDNA 4 and RDNA 2, RDNA 3, and RDNA 4) bytes per cycle total bandwidth. Each bank can be accessed independently every cycle for reads, writes, and/or atomic operations. The SIMDs connect to the LDS in pairs, with each pair sharing a 64-byte bidirectional port.

Access patterns: A single warp can achieve up to 64 bytes per cycle throughput (16 lanes per cycle). The actual bandwidth depends on data size and access patterns:

- 4-byte values: 8 cycles for 64 threads (50% peak bandwidth)
- 16-byte values: 20 cycles for 64 threads (80% peak bandwidth)

Conflict resolution: The LDS includes hardware to detect and resolve bank conflicts when multiple threads access different addresses in the same bank. Conflicts are resolved by serializing accesses across multiple cycles. Address conflicts (multiple threads atomically updating the same address) are similarly serialized. Broadcasting from the same address to multiple threads is handled efficiently without conflicts.

6.3.4 Vector L1 cache

Each CU contains a dedicated vector L1 data cache (vL1D) serving vector memory operations. Key characteristics include:

- Write-through design (writes go directly to L2)
- Optimization for high-bandwidth streaming access patterns

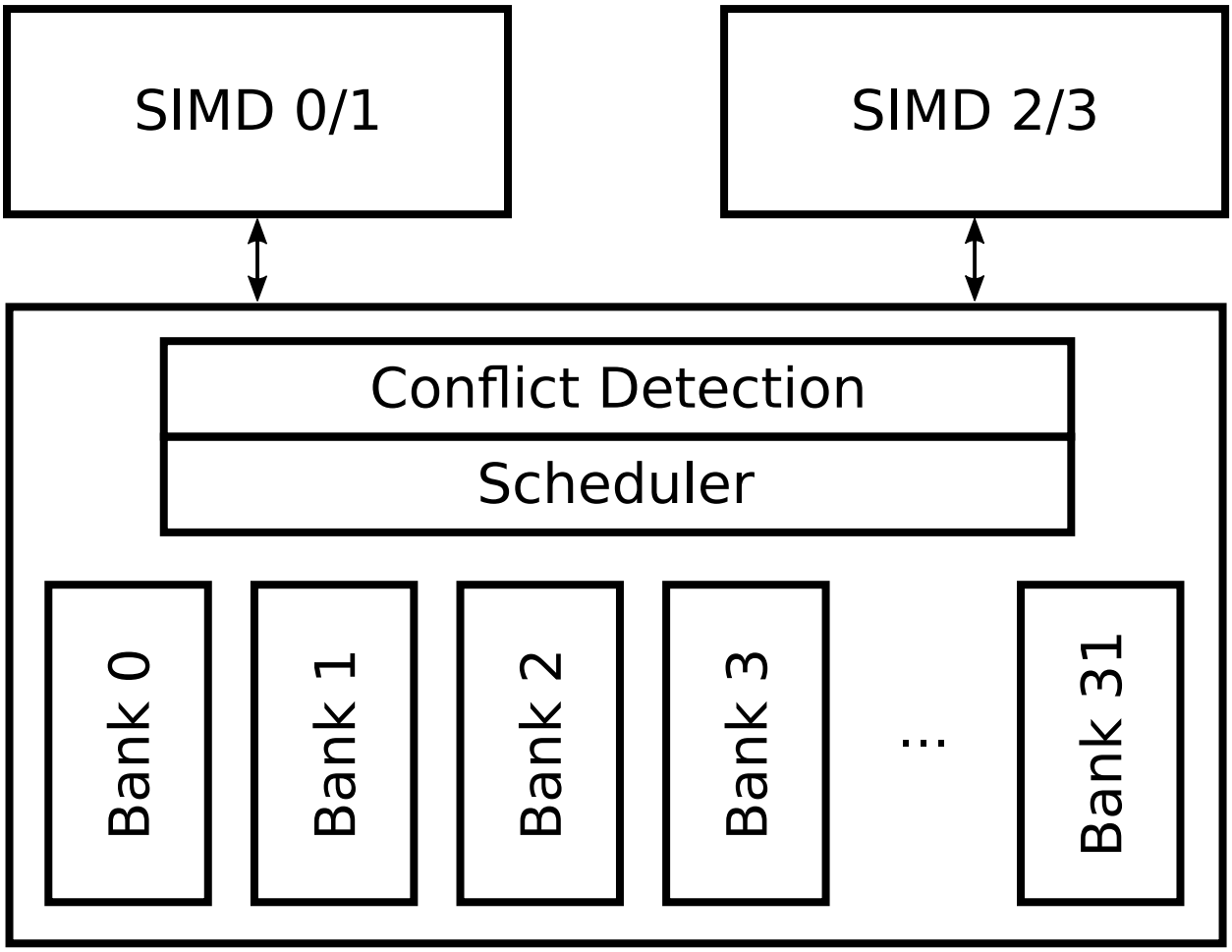


Fig. 3: Local data share organization and SIMD connections

- Coherent with other CUs through software management
- Typical size of 16 KB per CU

The vector cache tags are checked for all vector memory operations, with misses forwarded to the L2 cache. The write-through design simplifies coherence at the cost of write bandwidth.

6.4 Memory hierarchy and system

The GPU memory system provides the bandwidth and capacity needed for massive parallel computation while managing data coherence and access efficiency.

6.4.1 Memory organization

AMD GPUs typically use high-bandwidth memory (HBM) for data-intensive workloads, providing significantly higher bandwidth than traditional GDDR memory at the cost of slightly higher latency. HBM achieves this through vertical stacking of memory dies and wide memory buses, enabling massive parallel access to memory channels.

The memory system includes:

- **Memory channels**: Multiple independent memory controllers (typically 8-16)
- **L2 cache banks**: Distributed cache banks serving as the coherence point
- **Infinity Fabric**: High-speed interconnect for data routing

6.4.2 L2 cache architecture

The L2 cache serves as the coherence point for all GPU memory accesses and is shared by all compute units. The L2 consists of multiple independent channels (32 on CDNA GPUs at 256-byte interleaving) that operate in parallel.

Key characteristics:

- **Channel organization**: Each channel handles a portion of the address space, with addresses interleaved across channels for load balancing.
- **Hit-on-miss behavior**: If a request arrives for a pending cache line fill, it counts as a hit, improving the effective hit rate.
- **Write coalescing**: Multiple writes to the same cache line are combined.
- **Atomic operation support**: Atomics execute directly in the L2 cache for coherence.

L2-Fabric interface: Requests missing in L2 are routed through Infinity Fabric to the appropriate memory location, which could be:

- Local HBM on the same GPU
- Remote GPU memory (in multi-GPU systems)
- System memory (CPU DRAM)

The interface categorizes requests by type (read and/or write), size (32B and/or 64B), and destination for optimal routing.

6.4.3 Memory coherence

GPU memory coherence differs significantly from CPU designs to optimize for throughput over latency:

Write-through L1 caches: All writes update both L1 and L2, ensuring L2 always has the latest data. This eliminates the need for complex coherence protocols between L1 caches but requires higher write bandwidth.

Software-managed coherence: Coherence between CUs requires explicit synchronization through:

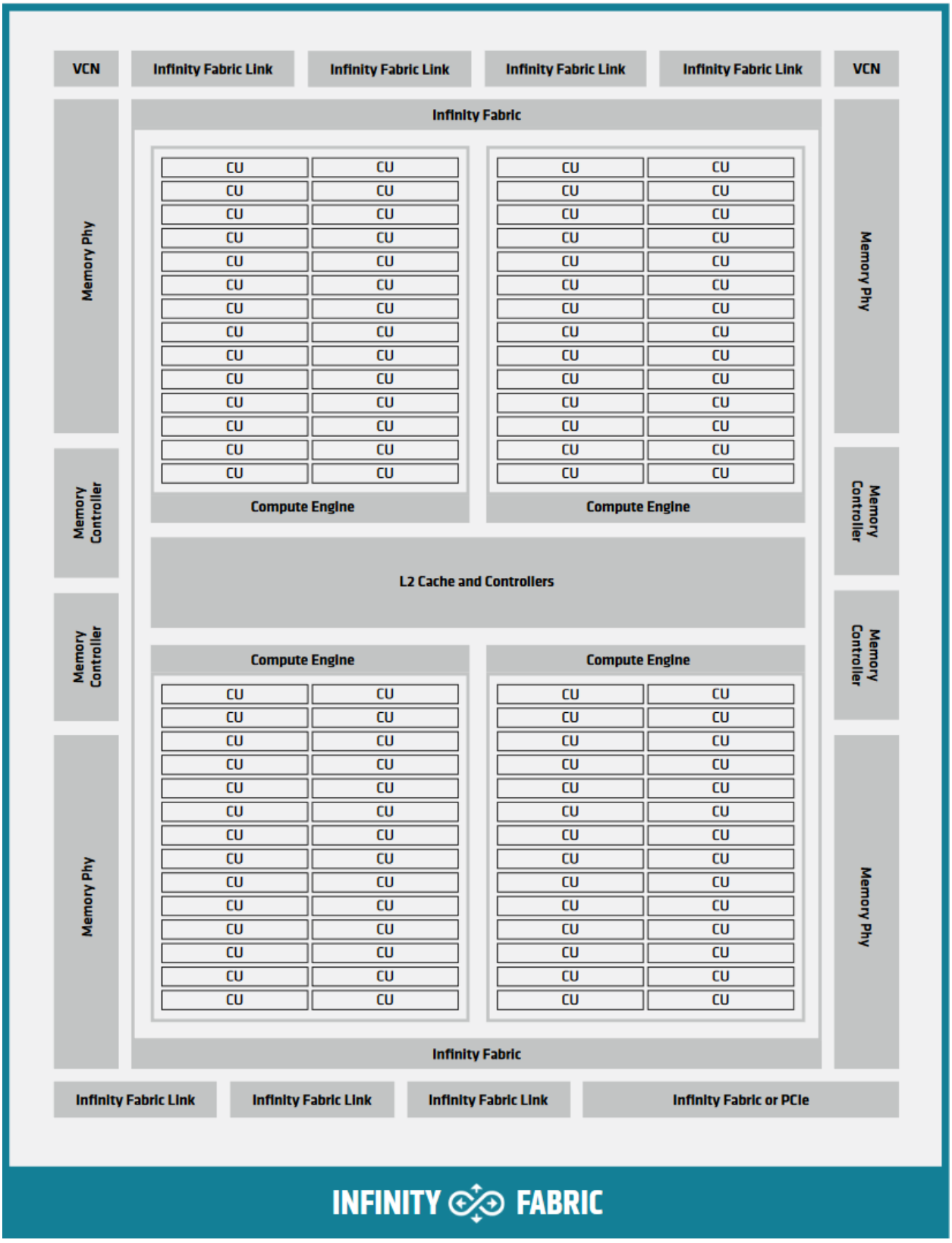


Fig. 4: CDNA2 Graphics Compute Die organization showing memory subsystem

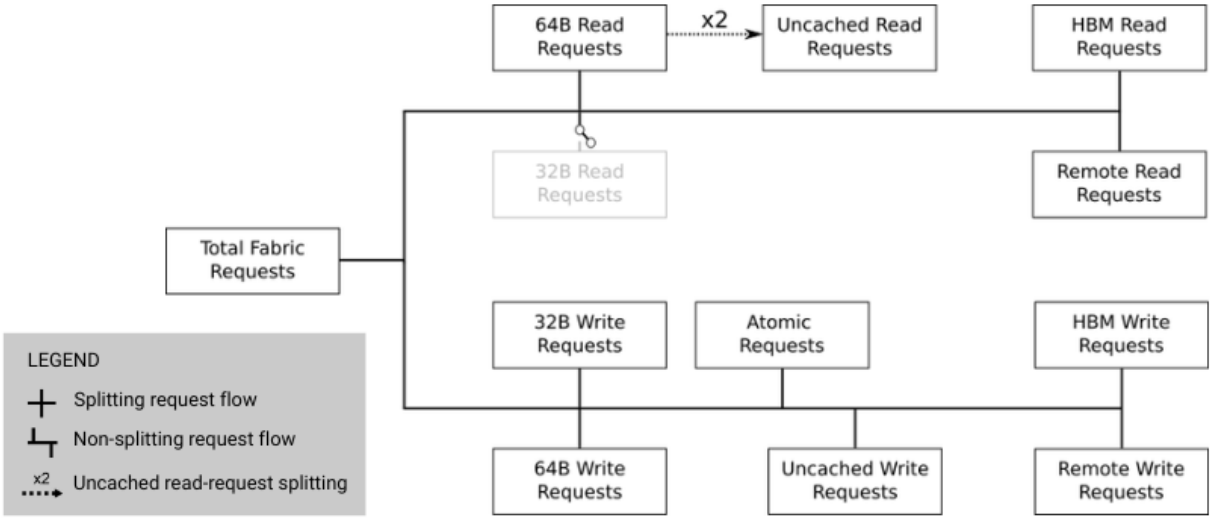


Fig. 5: L2 cache to Infinity Fabric transaction flow

- Memory fences for ordering
- Cache invalidation instructions
- Atomic operations (executed at L2 level)
- Kernel boundaries (implicit synchronization)

Write combining: To handle partial cache line updates from different CUs, the GPU uses write masks indicating which bytes to update. This prevents false sharing issues while maintaining correctness.

6.4.4 Memory coalescing

Memory coalescing combines memory accesses from multiple threads into fewer transactions, significantly improving bandwidth utilization. The coalescing hardware in the VMEM unit analyzes addresses from all threads in a warp and groups them into the minimum number of cache line requests.

Coalesced access pattern: When consecutive threads access consecutive memory addresses, the hardware can combine all 64 thread requests into as few as 4-8 cache line requests (depending on data size and alignment).

Non-coalesced access pattern: When threads access widely separated addresses, each thread can generate a separate memory transaction, reducing effective bandwidth by up to 16x or more.

To achieve optimal memory performance:

- Ensure consecutive threads access consecutive memory addresses
- Align data structures to cache line boundaries (64B and/or 128B)
- Use structure-of-arrays rather than array-of-structures layouts
- Consider padding to avoid bank conflicts

6.5 Architecture variants

AMD supports multiple GPU architecture families optimized for different use cases while maintaining HIP compatibility.

6.5.1 CDNA architecture

CDNA (Compute DNA) specializes in high-performance computing and machine learning workloads. Key features include:

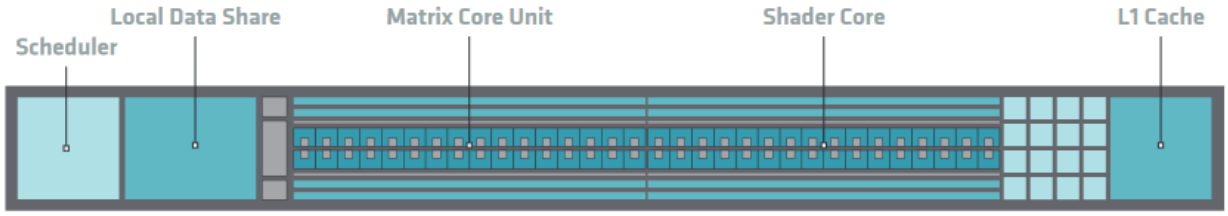


Fig. 6: CDNA3 compute unit with matrix acceleration

Matrix Core Unit: Specialized hardware for matrix multiply-accumulate operations, providing significantly more throughput than vector units for supported operations. Matrix cores support multiple precisions (`INT8`, `FP16`, `BF16`, `FP32`) with varying performance characteristics.

Accumulation VGPRs (AGPRs): Additional register file space (up to 256 KB) dedicated to matrix accumulation, doubling the available register storage for matrix operations. Data movement between VGPRs and AGPRs uses specialized instructions (`v_accvgpr_*`).

Enhanced memory bandwidth: CDNA GPUs typically use HBM2, HBM2e, and/or HBM3 memory technology.

Multi-die designs: CDNA2 (MI250) and CDNA3 (MI300) use chiplet architectures with multiple dies connected through high-speed links, scaling to higher compute and memory capacities.

6.5.2 RDNA architecture

RDNA optimizes for graphics and lower-latency compute workloads through fundamental architectural changes:

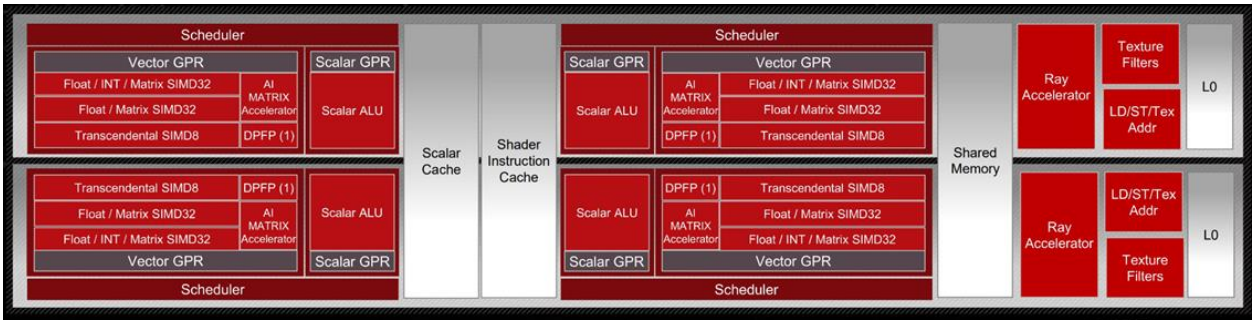


Fig. 7: RDNA3 work group processor architecture

Wave32 execution: Primary execution mode uses 32-thread warps, reducing divergence penalties and register pressure.

Dual compute units: The work group processor (WGP) replaces standalone CUs, containing two closely coupled compute units sharing resources:

- Each CU has two 32-wide SIMD units
- Warps execute in a single cycle on 32-wide SIMDs

- Reduced instruction latency improves responsiveness

Three-level cache hierarchy:

- **L0 cache**: Per-CU cache
- **L1 cache**: Shared between CUs in a WGP (new intermediate level)
- **L2 cache**: Global cache shared across all WGPs

128-byte cache lines: Aligning with Wave32 access patterns (32 threads × 4 bytes = 128 bytes).

These RDNA optimizations target gaming workloads where latency matters more than pure throughput, though the architecture remains capable for general compute tasks.

6.6 Performance considerations

Understanding hardware characteristics helps you optimize GPU applications for maximum performance.

6.6.1 Occupancy and resource limits

Occupancy measures the ratio of active warps to maximum possible warps on a CU. Higher occupancy generally improves latency hiding but is limited by:

- **Register usage**: Each warp requires VGPRs and SGPRs from finite pools
- **LDS allocation**: Shared memory used per workgroup
- **warp slots**: Fixed number of execution contexts per CU
- **Workgroup size**: Smaller workgroups can waste resources

Balancing these resources is critical for achieving optimal occupancy. Tools such as `rocprofv3` can help analyze occupancy and identify limiting factors.

6.6.2 Latency hiding through multithreading

GPUs hide memory and instruction latency through massive hardware multithreading rather than complex CPU techniques such as out-of-order execution and/or speculation. With sufficient warps:

- Memory latency is hidden by executing other warps during waits
- Pipeline latencies are covered by round-robin warp scheduling
- No context switch overhead as all contexts remain resident

The hardware can switch between warps every cycle, maintaining high ALU utilization even with long-latency operations in flight.

6.6.3 Memory bandwidth utilization

Effective memory bandwidth depends on access patterns:

- **Coalesced access**: Can achieve 70-90% of peak bandwidth
- **Random access**: Might achieve only 5-15% of peak bandwidth
- **Bank conflicts**: Can serialize LDS access, reducing throughput

Memory-bound kernels should focus on:

- Maximizing coalescing through proper data layout
- Prefetching and data reuse in LDS

- Balancing computation with memory access
- Using appropriate cache policies

6.6.4 Hardware-specific optimizations

Different AMD GPU architectures benefit from tailored optimizations:

For CDNA:

- Optimize for 64-thread warp granularity
- Leverage matrix cores for applicable algorithms
- Consider AGPR usage for register spilling

For RDNA:

- Design for 32-thread warp execution
- Utilize improved divergence handling
- Take advantage of additional cache level

Architecture-agnostic:

- Minimize divergent control flow
- Ensure memory access coalescing
- Balance resource usage for occupancy
- Overlap computation with memory access

6.7 Summary

AMD GPU hardware architecture provides massive parallelism through hierarchical organization of compute resources, specialized execution units, and a sophisticated memory system. Understanding these hardware details, from the command processor through shader engines to individual compute units and the memory hierarchy, enables you to write more efficient GPU applications.

Key hardware concepts for optimization include:

- Workgroup scheduling and resource management by the SPI
- Instruction scheduling and warp execution in compute units
- Memory coalescing and cache behavior
- Architecture-specific features (matrix cores, Wave32 and/or Wave64 modes)
- Resource limits affecting occupancy

For details on mapping parallel algorithms to this hardware, see the Introduction to the HIP programming model chapter. For specific optimization techniques, consult the performance optimization guides in the ROCm documentation.

CHAPTER

SEVEN

SAXPY TUTORIAL - HELLO HIP

To follow this tutorial, you'll need installed drivers and a HIP compiler toolchain to compile your code. Because HIP supports compiling and running on Linux and Windows with AMD GPUs, the install instructions are more than worth covering as part of this tutorial. For more information about installing HIP development packages, see *ROCm install*.

7.1 Heterogeneous programming

Heterogeneous programming and *offloading APIs* are often mentioned together. Heterogeneous programming deals with devices of varying capabilities simultaneously. Offloading focuses on the "remote" and asynchronous aspects of computation. HIP encompasses both. It exposes GPGPU (general-purpose GPU) programming much like ordinary host-side CPU programming and lets you move data across various devices.

When programming in HIP (and other heterogenous APIs for that matter), remember that target devices are built for a specific purpose. They are designed with different tradeoffs than traditional CPUs and therefore have very different performance characteristics. Even subtle changes in code might adversely affect execution time.

7.2 Your first lines of HIP code

First, let's do the "Hello, World!" of GPGPU: SAXPY. Single-precision A times X Plus Y (*SAXPY*) is a mathematical acronym; a vector equation $a \cdot x + y = z$ where $a \in \mathbb{R}$ is a scalar and $x, y, z \in \mathbb{V}$ are vector quantities of some large dimensionality. This vector space is defined over the set of reals. Practically speaking, you can compute this using a single `for` loop over three arrays.

```
for (int i = 0 ; i < N ; ++i)
    z[i] = a * x[i] + y[i];
```

In linear algebra libraries, such as BLAS (Basic Linear Algebra Subsystem) this operation is defined as AXPY "A times X Plus Y". The "S" comes from *single-precision*, meaning that array element is `float` -s (IEEE 754 binary32 representation).

To quickly get started, use the set of HIP samples from GitHub. With Git configured on your machine, open a command-line and navigate to your desired working directory, then run:

```
git clone https://github.com/amd/rocm-examples.git
```

A simple implementation of SAXPY resides in the `HIP-Basic/saxpy/main.hip` file in this repository. The HIP code here mostly deals with where data has to be and when, and how devices transform this data. The first HIP calls deal with allocating device-side memory and copying data from host-side memory to device side in a C runtime-like fashion.

```
// Allocate and copy vectors to device memory.
float* d_x{};
float* d_y{};
```

(continues on next page)

(continued from previous page)

```
HIP_CHECK(hipMalloc(&d_x, size_bytes));
HIP_CHECK(hipMalloc(&d_y, size_bytes));
HIP_CHECK(hipMemcpy(d_x, x.data(), size_bytes, hipMemcpyHostToDevice));
HIP_CHECK(hipMemcpy(d_y, y.data(), size_bytes, hipMemcpyHostToDevice));
```

`HIP_CHECK` is a custom macro borrowed from the examples utilities which checks the error code returned by API functions for errors and reports them to the console. It is not essential to the API, but it is a good practice to check the error codes of the HIP APIs in case you pass on incorrect values to the API, or the API might be out of resources.

The code selects the device to allocate to and to copy to. Commands are issued to the HIP runtime per thread, and every thread has a device set as the target of commands. The default device is `0`, which is equivalent to calling `hipSetDevice(0)`.

Launch the calculation on the device after the input data has been prepared.

```
__global__ void saxpy_kernel(const float a, const float* d_x, float* d_y, const␣
↪unsigned int size)
{
    // ...
}

int main()
{
    // ...

    // Launch the kernel on the default stream.
    saxpy_kernel<<<dim3(grid_size), dim3(block_size), 0, hipStreamDefault>>>(a, d_x,␣
↪d_y, size);
}
```

Analyze at the signature of the offloaded function:

- `__global__` instructs the compiler to generate code for this function as an entrypoint to a device program, such that it can be launched from the host.
- The function does not return anything, because there is no trivial way to construct a return channel of a parallel invocation. Device-side entrypoints may not return a value, their results should be communicated using output parameters.
- Device-side functions are typically called compute kernels, or just kernels for short. This is to distinguish them from non-graphics-related graphics shaders, or just shaders for short.
- Arguments are taken by value and all arguments shall be TriviallyCopyable, meaning they should be *memcpy*-friendly. (Imagine if they had custom copy constructors. Where would that logic execute? On the host? On the device?) Pointer arguments are pointers to device memory, one typically backed by VRAM.
- We said that we'll be computing $a \cdot x + y = z$, however we only pass two pointers to the function. We'll be canonically reusing one of the inputs as outputs.

This function is launched from the host using a language extension often called the triple chevron syntax. Inside the angle brackets, provide the following.

- The number of *blocks* to launch (our *grid* size)
- The number of threads in a *block* (our *block* size)
- The amount of shared memory to allocate by the host
- The device stream to enqueue the operation on

The *block* size and shared memory become important later in *Reduction*. For now, a hardcoded `256` is a safe default for simple kernels such as this. Following the triple chevron is ordinary function argument passing.

Look at how the kernel is implemented.

```
__global__ void saxpy_kernel(const float a, const float* d_x, float* d_y, const␣
↪unsigned int size)
{
    // Compute the current thread's index in the grid.
    const unsigned int global_idx = blockIdx.x * blockDim.x + threadIdx.x;

    // The grid can be larger than the number of items in the vectors. Avoid out-of-
↪bounds addressing.
    if(global_idx < size)
    {
        d_y[global_idx] = a * d_x[global_idx] + d_y[global_idx];
    }
}
```

- The unique linear index identifying the thread is computed from the *block* ID the thread is a member of, the *block*'s size and the ID of the thread within the *block*.
- A check is made to avoid overindexing the input.
- The useful part of the computation is carried out.

Retrieval of the result from the device is done much like input data copy. In this current step the results copied from device to host. The opposite direction of the input data copy:

```
HIP_CHECK(hipMemcpy(y.data(), d_y, size_bytes, hipMemcpyDeviceToHost));
```

7.3 Compiling on the command line

7.3.1 Setting up the command line

Strictly speaking there's no such thing as "setting up the command-line for compilation" on Linux. To make invocations more terse, Linux and Windows example follow.

Linux

While distro maintainers might package ROCm so that it installs to system-default locations, AMD's packages aren't installed that way. They need to be added to the PATH by the user.

```
export PATH=/opt/rocm/bin:${PATH}
```

You should be able to call the compiler on the command line now:

```
amdclang++ --version
```

> **Note**
>
> Docker images distributed by AMD, such as rocm-terminal already have */opt/rocm/bin* on the Path for convenience. This subtly affects CMake package detection logic of ROCm libraries.

Windows

Windows compilers and command line tooling have traditionally relied on extra environmental variables and PATH entries to function correctly. Visual Studio refers to command lines with this setup as "Developer Command Prompt" or "Developer PowerShell" for `cmd.exe` and PowerShell respectively.

The HIP SDK on Windows doesn't include a complete toolchain. You will also need:

- The Microsoft Windows SDK. It provides the import libs to crucial system libraries that all executables must link to and some auxiliary compiler tooling.
- A Standard Template Library (STL). Installed as part of the Microsoft Visual C++ compiler (MSVC) or with Visual Studio.

If you don't have a version of Visual Studio 2022 installed, for a minimal command line experience, install the Build Tools for Visual Studio 2022 with the Desktop Developemnt Workload. Under Individual Components select:

- A version of the Windows SDK
- "MSVC v143 - VS 2022 C++ x64/x86 build tools (Latest)"
- "C++ CMake tools for Windows" (optional)

> **Note**
>
> The "C++ CMake tools for Windows" individual component is a convenience which puts both `cmake.exe` and `ninja.exe` onto the PATH inside developer command prompts. You can install these manually, but then you must manage them manually.

Visual Studio 2017 and later are detectable as COM object instances via WMI. To setup a command line from any shell for the latest Visual Studio's default Visual C++ toolset issue:

```
$InstallationPath = Get-CimInstance MSFT_VSInstance | Sort-Object -Property Version -
↪Descending | Select-Object -First 1 -ExpandProperty InstallLocation
Import-Module $InstallationPath\Common7\Tools\Microsoft.VisualStudio.DevShell.dll
Enter-VsDevShell -InstallPath $InstallationPath -SkipAutomaticLocation -Arch amd64 -
↪HostArch amd64 -DevCmdArguments '-no_logo'
$env:PATH = "${env:HIP_PATH}bin;${env:PATH}"
```

You should be able to call the compiler on the command line now:

```
clang++ --version
```

7.3.2 Invoking the compiler manually

To compile and link a single-file application, use the following commands:

Linux

```
amdclang++ ./HIP-Basic/saxpy/main.hip -o saxpy -I ./Common -lamdhip64 -L /opt/rocm/
↪lib -O2
```

Windows

```
clang++ .\HIP-Basic\saxpy\main.hip -o saxpy.exe -I .\Common -lamdhip64 -L ${env:HIP_
↪PATH}lib -O2
```

Depending on your computer, the resulting binary might or might not run. If not, it typically complains about "Invalid device function". That error (corresponding to the `hipErrorInvalidDeviceFunction` entry of `hipError_t`) means that the runtime could not find a device program binary of the appropriate flavor embedded into the executable.

So far, the discussion has covered how data makes it from the host to the device and back. It has also discussed the device code as source, with the HIP runtime arguing that the correct binary to dispatch for execution. How can you find out what device binary flavors are embedded into the executable?

Linux

The utilities included with ROCm help significantly to inspect binary artifacts on disk. Add the ROCmCC installation folder to your PATH if you want to use these utilities (the utilities expect them to be on the PATH).

You can list embedded program binaries using `llvm-objdump` with `--offloading` option.

```
llvm-objdump --offloading ./saxpy
```

It should return something like:

```
./saxpy:        file format elf64-x86-64
Extracting offload bundle: ./saxpy.0.host-x86_64-unknown-linux-gnu-
Extracting offload bundle: ./saxpy.0.hipv4-amdgcn-amd-amdhsa--gfx942
```

The compiler embeds a version 4 code object (more on code object versions) and used the LLVM target triple `amdgcn-amd-amdhsa--gfx942` (more on target triples). You can extract that program object in a disassembled fashion for human consumption via `llvm-objdump`.

```
llvm-objdump --disassemble saxpy.0.hipv4-amdgcn-amd-amdhsa--gfx942 > saxpy.s
```

This creates a file on the disk called `saxpy.s` Opening this file or dumping it to the console using `cat` lets find the disassembled binary of the SAXPY compute kernel, something similar to:

```
saxpy.0.hipv4-amdgcn-amd-amdhsa--gfx942:        file format elf64-amdgpu

Disassembly of section .text:

0000000000001900 <_Z12saxpy_kernelfPKfPfj>:
  s_load_dword s3, s[0:1], 0x2c                                 // 000000001900:␣
↪C00200C0 0000002C
  s_load_dword s4, s[0:1], 0x18                                 // 000000001908:␣
↪C0020100 00000018
  s_waitcnt lgkmcnt(0)                                          // 000000001910: BF8CC07F
  s_and_b32 s3, s3, 0xffff                                      // 000000001914:␣
↪8603FF03 0000FFFF
  s_mul_i32 s2, s2, s3                                          // 00000000191C: 92020302
  v_add_u32_e32 v0, s2, v0                                      // 000000001920: 68000002
  v_cmp_gt_u32_e32 vcc, s4, v0                                  // 000000001924: 7D980004
  s_and_saveexec_b64 s[2:3], vcc                                // 000000001928: BE82206A
  s_cbranch_execz 20                                            // 00000000192C:␣
↪BF880014 <_Z12saxpy_kernelfPKfPfj+0x80>
```

(continues on next page)

(continued from previous page)

```
  s_load_dwordx4 s[4:7], s[0:1], 0x8                          // 000000001930:␣
↪C00A0100 00000008
  v_mov_b32_e32 v1, 0                                         // 000000001938: 7E020280
  v_lshlrev_b64 v[0:1], 2, v[0:1]                             // 00000000193C:␣
↪D28F0000 00020082
  s_load_dword s0, s[0:1], 0x0                                // 000000001944:␣
↪C0020000 00000000
  s_waitcnt lgkmcnt(0)                                        // 00000000194C: BF8CC07F
  v_lshl_add_u64 v[2:3], s[4:5], 0, v[0:1]                    // 000000001950:␣
↪D2080002 04010004
  v_lshl_add_u64 v[0:1], s[6:7], 0, v[0:1]                    // 000000001958:␣
↪D2080000 04010006
  global_load_dword v4, v[2:3], off                           // 000000001960:␣
↪DC508000 047F0002
  global_load_dword v5, v[0:1], off                           // 000000001968:␣
↪DC508000 057F0000
  s_waitcnt vmcnt(0)                                          // 000000001970: BF8C0F70
  v_fmac_f32_e32 v5, s0, v4                                   // 000000001974: 760A0800
  global_store_dword v[0:1], v5, off                          // 000000001978:␣
↪DC708000 007F0500
  s_endpgm                                                    // 000000001980: BF810000
  s_nop 0                                                     // 000000001984: BF800000
```

Alternatively, call the compiler with `--save-temps` to dump all device binary to disk in separate files.

```
amdclang++ ./HIP-Basic/saxpy/main.hip -o saxpy -I ./Common -lamdhip64 -L /opt/rocm/
↪lib -O2 --save-temps --offload-arch=gfx942
```

List all the temporaries created while compiling `main.hip` with:

```
ls main-hip-amdgcn-amd-amdhsa-*
main-hip-amdgcn-amd-amdhsa-gfx942.bc
main-hip-amdgcn-amd-amdhsa-gfx942.o
main-hip-amdgcn-amd-amdhsa-gfx942.out.resolution.txt
main-hip-amdgcn-amd-amdhsa-gfx942.hipi
main-hip-amdgcn-amd-amdhsa-gfx942.out
main-hip-amdgcn-amd-amdhsa-gfx942.s
```

Files with the `.s` extension hold the disassembled contents of the binary. The filename notes the graphics IPs used by the compiler. The contents of this file are similar to the **.s* file created with `llvm-objdump` earlier.

Windows

The HIP SDK for Windows don't yet sport the `roc-*` set of utilities to work with binary artifacts. To find out what binary formats are embedded into an executable, one may use `dumpbin` tool from the Windows SDK to obtain the raw data of the `.hip_fat` section of an executable. (This binary payload is what gets parsed by the `roc-*` set of utilities on Linux.) Skipping over the reported header, the rendered raw data as ASCII has ~3 lines per entries. Depending on how many binaries are embedded, you may need to alter the number of rendered lines. An invocation such as:

```
dumpbin.exe /nologo /section:.hip_fat /rawdata:8 .\saxpy.exe | select -Skip 20 -First␣
↪12
```

The output may look like:

```
000000014004C000: 5F474E414C435F5F 5F44414F4C46464F   __CLANG_OFFLOAD_
000000014004C010: 5F5F454C444E5542 0000000000000002   BUNDLE__........
000000014004C020: 0000000000001000 0000000000000000   ................
000000014004C030: 0000000000000019 3638782D74736F68   ........host-x86
000000014004C040: 6E6B6E752D34365F 756E696C2D6E776F   _64-unknown-linu
000000014004C050: 0000000000100078 00000000000D9800   x...............
000000014004C060: 0000000000001F00 612D347670696800   .........hipv4-a
000000014004C070: 6D612D6E6367646D 617368646D612D64   mdgcn-amd-amdhsa
000000014004C080: 3630397866672D2D 0000000000000000   --gfx906........
000000014004C090: 0000000000000000 0000000000000000   ................
000000014004C0A0: 0000000000000000 0000000000000000   ................
000000014004C0B0: 0000000000000000 0000000000000000   ................
```

We can see that the compiler embedded a version 4 code object (more on code object versions) and used the LLVM target triple `amdgcn-amd-amdhsa--gfx906` (more on target triples). Don't be alarmed about linux showing up as a binary format, AMDGPU binaries uploaded to the GPU for execution are proper linux ELF binaries in their format.

Alternatively we can call the compiler with `--save-temps` to dump all device binary to disk in separate files.

```
clang++ .\HIP-Basic\saxpy\main.hip -o saxpy.exe -I .\Common -lamdhip64 -L ${env:HIP_
↪PATH}lib -O2 --save-temps
```

Now we can list all the temporaries created while compiling `main.hip` via

```
Get-ChildItem -Filter main-hip-* | select -Property Name

Name
----
main-hip-amdgcn-amd-amdhsa-gfx906.bc
main-hip-amdgcn-amd-amdhsa-gfx906.hipi
main-hip-amdgcn-amd-amdhsa-gfx906.o
main-hip-amdgcn-amd-amdhsa-gfx906.out
main-hip-amdgcn-amd-amdhsa-gfx906.out.resolution.txt
main-hip-amdgcn-amd-amdhsa-gfx906.s
```

Files with the `.s` extension hold the disassembled contents of the binary and the filename directly informs us of the graphics IPs used by the compiler.

```
Get-ChildItem main-hip-*.s | Get-Content
        .text
        .amdgcn_target "amdgcn-amd-amdhsa--gfx906"
        .protected      _Z12saxpy_kernelfPKfPfj ; -- Begin function _Z12saxpy_
↪kernelfPKfPfj
        .globl  _Z12saxpy_kernelfPKfPfj
        .p2align        8
        .type   _Z12saxpy_kernelfPKfPfj,@function
_Z12saxpy_kernelfPKfPfj:                ; @_Z12saxpy_kernelfPKfPfj
; %bb.0:
        s_load_dword s0, s[4:5], 0x4
        s_load_dword s1, s[6:7], 0x18
        s_waitcnt lgkmcnt(0)
        s_and_b32 s0, s0, 0xffff
        s_mul_i32 s8, s8, s0
        v_add_u32_e32 v0, s8, v0
```

(continues on next page)

(continued from previous page)

```
        v_cmp_gt_u32_e32 vcc, s1, v0
        s_and_saveexec_b64 s[0:1], vcc
        s_cbranch_execz .LBB0_2
; %bb.1:
        s_load_dwordx4 s[0:3], s[6:7], 0x8
        v_mov_b32_e32 v1, 0
        v_lshlrev_b64 v[0:1], 2, v[0:1]
        s_waitcnt lgkmcnt(0)
        v_mov_b32_e32 v3, s1
        v_add_co_u32_e32 v2, vcc, s0, v0
        v_addc_co_u32_e32 v3, vcc, v3, v1, vcc
        global_load_dword v2, v[2:3], off
        v_mov_b32_e32 v3, s3
        v_add_co_u32_e32 v0, vcc, s2, v0
        v_addc_co_u32_e32 v1, vcc, v3, v1, vcc
        global_load_dword v3, v[0:1], off
        s_load_dword s0, s[6:7], 0x0
        s_waitcnt vmcnt(0) lgkmcnt(0)
        v_fmac_f32_e32 v3, s0, v2
        global_store_dword v[0:1], v3, off
.LBB0_2:
        s_endpgm
        ...
```

Now that you've found what binary got embedded into the executable, find which format our available devices use.

Linux

On Linux a utility called `rocminfo` helps us list all the properties of the devices available on the system, including which version of graphics IP (`gfxXYZ`) they employ. You can filter the output to have only these lines:

```
/opt/rocm/bin/rocminfo | grep gfx
  Name:                    gfx906
      Name:                    amdgcn-amd-amdhsa--gfx906:sramecc+:xnack-
```

Now that you know which graphics IPs our devices use, recompile your program with the appropriate parameters.

```
amdclang++ ./HIP-Basic/saxpy/main.hip -o saxpy -I ./Common -lamdhip64 -L /opt/rocm/
↪lib -O2 --offload-arch=gfx906:sramecc+:xnack-
```

Now the sample will run.

```
./saxpy
Calculating y[i] = a * x[i] + y[i] over 1000000 elements.
First 10 elements of the results: [ 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 ]
```

Windows

On Windows, a utility called `hipInfo.exe` helps us list all the properties of the devices available on the system, including which version of graphics IP (`gfxXYZ`) they employ. Filter the output to have only these lines:

```
& ${env:HIP_PATH}bin\hipInfo.exe | Select-String gfx

```

(continues on next page)

(continued from previous page)

```
gcnArchName:                    gfx1032
gcnArchName:                    gfx1035
```

Now that you know which graphics IPs our devices use, recompile your program with the appropriate parameters.

```
clang++ .\HIP-Basic\saxpy\main.hip -o saxpy.exe -I .\Common -lamdhip64 -L ${env:HIP_
↪PATH}lib -O2 --offload-arch=gfx1032 --offload-arch=gfx1035
```

Now the sample will run.

```
.\saxpy.exe
Calculating y[i] = a * x[i] + y[i] over 1000000 elements.
First 10 elements of the results: [ 3, 5, 7, 9, 11, 13, 15, 17, 19, 21 ]
```

CHAPTER

EIGHT

INTRODUCTION TO CORE HIP PROGRAMMING

HIP is AMD's GPU programming interface within the ROCm ecosystem. It provides a C++ runtime API and kernel language for writing GPU applications. HIP is designed to expose GPU hardware capabilities directly, enabling developers to implement fine-grained parallelism for scientific computing, simulations, and AI workloads.

As described in *Introduction to the HIP programming model*, the HIP API uses the following features for GPU programming:

- Kernels: Functions executed in parallel across many GPU threads.
- Thread hierarchy: Threads are organized into blocks, and blocks form a grid.
- Memory spaces: HIP supports global, shared, and constant memory, each optimized for different access patterns.
- Runtime API functions: Device management, memory allocation, kernel launches, synchronization, and error handling.

Note

HIP syntax is similar to NVIDIA CUDA to facilitate porting applications from CUDA to HIP. However, its real strength lies in enabling direct, efficient programming for AMD GPUs.

These elements are configured and accessed through the HIP runtime API as described in this section. Beyond the runtime API, there are specific GPU programming patterns that help to deliver parallelism and performance optimization for your GPU applications. These programming patterns provide:

- Massive concurrency: Patterns map naturally to thousands of GPU threads.
- Memory efficiency: Shared memory and tiling reduce costly global memory accesses.
- Synchronization tools: HIP provides barriers and atomics to coordinate threads.
- Scalability: Patterns generalize across problem sizes, making them ideal for HPC and AI workloads.

The HIP runtime API gives developers direct control over GPU execution while tutorials on GPU programming patterns provide practical strategies for implementing parallelism. Together, they form a foundation for building scalable, efficient algorithms in HPC and AI.

CHAPTER

NINE

HIP KERNEL PROGRAMMING

As described in the *Introduction to the HIP programming model*, there are two execution contexts in HIP application code: host and device. These execution contexts are signified by the use of `__host__` and `__global__` (or `__device__`) qualifiers in the code. The following text describes the various qualifiers that can be defined in your HIP code, and the circumstances for using these qualifiers. Additional details are provided related to the use of built-in constants, HIP vector types, and built-in device functions in your code.

9.1 HIP qualifiers

9.1.1 Function-type qualifiers

HIP introduces three different function qualifiers to mark functions for execution on the device or the host, and also adds new qualifiers to control inlining of functions.

9.1.1.1 __host__

The `__host__` qualifier is used to specify functions for execution on the host. This qualifier is implicitly defined for any function where no `__host__`, `__device__` or `__global__` qualifier is added, in order to not break compatibility with existing C++ functions.

You can't combine `__host__` with `__global__`.

9.1.1.2 __device__

The `__device__` qualifier is used to specify functions for execution on the device. They can only be called from other `__device__` functions or from `__global__` functions.

You can combine it with the `__host__` qualifier and mark functions `__host__ __device__`. In this case, the function is compiled for the host and the device. Note that these functions can't use the HIP built-ins (e.g., *threadIdx.x* or *warpSize*), as they are not available on the host. If you need to use HIP grid coordinate functions, you can pass the necessary coordinate information as an argument.

9.1.1.3 __global__

Functions marked `__global__` are executed on the device and are referred to as kernels. Their return type must be `void`. Kernels have a special launch mechanism, and have to be launched from the host.

There are some restrictions on the parameters of kernels. Kernels can't:

- have a parameter of type `std::initializer_list` or `va_list`
- have a variable number of arguments
- use references as parameters

- use parameters having different sizes in host and device code, e.g. long double arguments, or structs containing long double members.
- use struct-type arguments which have different layouts in host and device code.

Kernels can have variadic template parameters, but only one parameter pack, which must be the last item in the template parameter list.

> **Note**
>
> Unlike CUDA, HIP does not support dynamic parallelism, meaning that kernels can not be called from the device.

9.1.1.3.1 Calling __global__ functions

The launch mechanism for kernels differs from standard function calls, as they need an additional configuration, that specifies the grid and block dimensions (i.e. the amount of threads to be launched), as well as specifying the amount of shared memory per block and which stream to execute the kernel on.

Kernels are called using the triple chevron `<<<>>>` syntax known from CUDA, but HIP also supports the `hipLaunchKernelGGL` macro.

When using `hipLaunchKernelGGL`, the first five configuration parameters must be:

- `symbol kernelName`: The name of the kernel you want to launch. To support template kernels that contain several template parameters separated by use the `HIP_KERNEL_NAME` macro to wrap the template instantiation (HIPIFY inserts this automatically).
- `dim3 gridDim`: 3D-grid dimensions that specifies the number of blocks to launch.
- `dim3 blockDim`: 3D-block dimensions that specifies the number of threads in each block.
- `size_t dynamicShared`: The amount of additional shared dynamic memory to allocate per block.
- `hipStream_t`: The stream on which to run the kernel. A value of `0` corresponds to the default stream.

The kernel arguments are listed after the configuration parameters.

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

#include <iostream>

#define HIP_CHECK(expression)                                  \
{                                                              \
    const hipError_t err = expression;                         \
    if(err != hipSuccess)                                      \
    {                                                          \
        std::cerr << "HIP error: " << hipGetErrorString(err) \
            << " at " << __LINE__ << "\n";                     \
    }                                                          \
}

// Performs a simple initialization of an array with the thread's index variables.
// This function is only available in device code.
__device__ void init_array(float * const a, const unsigned int arraySize)
{
    // globalIdx uniquely identifies a thread in a 1D launch configuration.
    const int globalIdx = threadIdx.x + blockIdx.x * blockDim.x;
```

(continues on next page)

(continued from previous page)

```
    // Each thread initializes a single element of the array.
    if(globalIdx < arraySize)
    {
        a[globalIdx] = globalIdx;
    }
}

// Rounds a value up to the next multiple.
// This function is available in host and device code.
__host__ __device__ constexpr int round_up_to_nearest_multiple(int number, int␣
↪multiple)
{
    return (number + multiple - 1)/multiple;
}

__global__ void example_kernel(float * const a, const unsigned int N)
{
    // Initialize array.
    init_array(a, N);
    // Perform additional work:
    // - work with the array
    // - use the array in a different kernel
    // - ...
}

int main()
{
    constexpr int N = 100000000; // problem size
    constexpr int blockSize = 256; //configurable block size

    //needed number of blocks for the given problem size
    constexpr int gridSize = round_up_to_nearest_multiple(N, blockSize);

    float *a;
    // allocate memory on the GPU
    HIP_CHECK(hipMalloc(&a, sizeof(*a) * N));

    std::cout << "Launching kernel." << std::endl;
    example_kernel<<<dim3(gridSize), dim3(blockSize), 0/*example doesn't use shared␣
↪memory*/, 0/*default stream*/>>>(a, N);
    // make sure kernel execution is finished by synchronizing. The CPU can also
    // execute other instructions during that time
    HIP_CHECK(hipDeviceSynchronize());
    std::cout << "Kernel execution finished." << std::endl;

    HIP_CHECK(hipFree(a));
}
```

9.1.1.4 Inline qualifiers

HIP adds the `__noinline__` and `__forceinline__` function qualifiers.

`__noinline__` is a hint to the compiler to not inline the function, whereas `__forceinline__` forces the compiler to inline the function. These qualifiers can be applied to both `__host__` and `__device__` functions.

`__noinline__` and `__forceinline__` can not be used in combination.

9.1.1.5 __launch_bounds__

GPU multiprocessors have a fixed pool of resources (primarily registers and shared memory) which are shared by the actively running warps. Using more resources per thread can increase executed instructions per cycle but reduces the resources available for other warps and may therefore limit the occupancy, i.e. the number of warps that can be executed simultaneously. Thus GPUs have to balance resource usage between instruction- and thread-level parallelism.

`__launch_bounds__` allows the application to provide hints that influence the resource (primarily registers) usage of the generated code. It is a function attribute that must be attached to a __global__ function:

```
__global__ void __launch_bounds__(MAX_THREADS_PER_BLOCK, MIN_WARPS_PER_EXECUTION_UNIT)
kernel_name(/*args*/);
```

The `__launch_bounds__` parameters are explained in the following sections:

9.1.1.5.1 MAX_THREADS_PER_BLOCK

This parameter is a guarantee from the programmer, that kernel will not be launched with more threads than `MAX_THREADS_PER_BLOCK`.

If no `__launch_bounds__` are specified, `MAX_THREADS_PER_BLOCK` is the maximum block size supported by the device (see ../reference/hardware_features). Reducing `MAX_THREADS_PER_BLOCK` allows the compiler to use more resources per thread than an unconstrained compilation. This might however reduce the amount of blocks that can run concurrently on a CU, thereby reducing occupancy and trading thread-level parallelism for instruction-level parallelism.

`MAX_THREADS_PER_BLOCK` is particularly useful in cases, where the compiler is constrained by register usage in order to meet requirements of large block sizes that are never used at launch time.

The compiler can only use the hints to manage register usage, and does not automatically reduce shared memory usage. The compilation fails, if the compiler can not generate code that satisfies the launch bounds.

On NVCC this parameter maps to the `.maxntid` PTX directive.

When launching kernels HIP will validate the launch configuration to make sure the requested block size is not larger than `MAX_THREADS_PER_BLOCK` and return an error if it is exceeded.

If AMD_LOG_LEVEL is set, detailed information will be shown in the error log message, including the launch configuration of the kernel and the specified `__launch_bounds__`.

9.1.1.5.2 MIN_WARPS_PER_EXECUTION_UNIT

This parameter specifies the minimum number of warps that must be able to run concurrently on an execution unit. `MIN_WARPS_PER_EXECUTION_UNIT` is optional and defaults to 1 if not specified. Since active warps compete for the same fixed pool of resources, the compiler must constrain the resource usage of the warps. This option gives a lower bound to the occupancy of the kernel.

From this parameter, the compiler derives a maximum number of registers that can be used in the kernel. The amount of registers that can be used at most is $\frac{\text{available registers}}{\text{MIN\_WARPS\_PER\_EXECUTION\_UNIT}}$, but it might also have other, architecture specific, restrictions.

The available registers per Compute Unit are listed in GPU hardware specifications. Beware that these values are per Compute Unit, not per Execution Unit. On AMD GPUs a Compute Unit consists of 4 Execution Units, also known as SIMDs, each with their own register file. For more information see ../understand/hardware_implementation. `hipDeviceProp_t` also has a field `executionUnitsPerMultiprocessor`.

9.1.2 Memory space qualifiers

HIP adds qualifiers to specify the memory space in which the variables are located.

Generally, variables allocated in host memory are not directly accessible within device code, while variables allocated in device memory are not directly accessible from the host code. More details on this can be found in *Unified memory management*.

9.1.2.1 __device__

Variables marked with `__device__` reside in device memory. It can be combined together with one of the following qualifiers, however these qualifiers also imply the `__device__` qualifier.

By default it can only be accessed from the threads on the device. In order to access it from the host, its address and size need to be queried using `hipGetSymbolAddress()` and `hipGetSymbolSize()` and copied with `hipMemcpyToSymbol()` or `hipMemcpyFromSymbol()`.

9.1.2.2 __constant__

Variables marked with `__constant__` reside in device memory. Variables in that address space are routed through the constant cache, but that address space has a limited logical size. This memory space is read-only from within kernels and can only be set by the host before kernel execution.

To get the best performance benefit, these variables need a special access pattern to benefit from the constant cache - the access has to be uniform within a warp, otherwise the accesses are serialized.

The constant cache reduces the pressure on the other caches and may enable higher throughput and lower latency accesses.

To set the `__constant__` variables the host must copy the data to the device using `hipMemcpyToSymbol()`, for example:

```
__constant__ int const_array[8];

void set_constant_memory(){
  int host_data[8] {1,2,3,4,5,6,7,8};

  hipMemcpyToSymbol(const_array, host_data, sizeof(int) * 8);

  // call kernel that accesses const_array
}
```

9.1.2.3 __shared__

Variables marked with `__shared__` are only accessible by threads within the same block and have the lifetime of that block. It is usually backed by on-chip shared memory, providing fast access to all threads within a block, which makes it perfectly suited for sharing variables.

Shared memory can be allocated statically within the kernel, but the size of it has to be known at compile time.

In order to dynamically allocate shared memory during runtime, but before the kernel is launched, the variable has to be declared `extern`, and the kernel launch has to specify the needed amount of `extern` shared memory in the launch configuration. The statically allocated shared memory is allocated without this parameter.

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

#include <cstdlib>
#include <iostream>

extern __shared__ int shared_array[];

__global__ void kernel()
{
    // initialize shared memory
    shared_array[threadIdx.x] = threadIdx.x;
    // use shared memory - synchronize to make sure, that all threads of the
    // block see all changes to shared memory
    __syncthreads();
}

int main()
{
    //shared memory in this case depends on the configurable block size
    constexpr int blockSize = 256;
    constexpr int sharedMemSize = blockSize * sizeof(int);
    constexpr int gridSize = 2;

    kernel<<<dim3(gridSize), dim3(blockSize), sharedMemSize, 0>>>();
    if(auto err = hipDeviceSynchronize(); err != hipSuccess)
        std::cerr << "HIP error " << err << ": " << hipGetErrorString(err) <<␣
↪std::endl;

    return EXIT_SUCCESS;
}
```

9.1.2.4 __managed__

Managed memory is a special qualifier, that makes the marked memory available on the device and on the host. For more details see *Unified memory management*.

9.1.2.5 __restrict__

The `__restrict__` keyword tells the compiler that the associated memory pointer does not alias with any other pointer in the function. This can help the compiler perform better optimizations. For best results, every pointer passed to a function should use this keyword.

9.2 Built-in constants

HIP defines some special built-in constants for use in device code.

These built-ins are not implicitly defined by the compiler, the `hip_runtime.h` header has to be included instead.

9.2.1 Index built-ins

Kernel code can use these identifiers to distinguish between the different threads and blocks within a kernel.

These built-ins are of type dim3, and are constant for each thread, but differ between the threads or blocks, and are initialized at kernel launch.

9.2.1.1 blockDim and gridDim

`blockDim` and `gridDim` contain the sizes specified at kernel launch. `blockDim` contains the amount of threads in the x-, y- and z-dimensions of the block of threads. Similarly `gridDim` contains the amount of blocks in the grid.

9.2.1.2 threadIdx and blockIdx

`threadIdx` and `blockIdx` can be used to identify the threads and blocks within the kernel.

`threadIdx` identifies the thread within a block, meaning its values are within `0` and `blockDim.{x,y,z} - 1`. Likewise `blockIdx` identifies the block within the grid, and the values are within `0` and `gridDim.{} - 1`.

A global unique identifier of a three-dimensional grid can be calculated using the following code:

```
(threadIdx.x + blockIdx.x * blockDim.x) +
(threadIdx.y + blockIdx.y * blockDim.y) * blockDim.x +
(threadIdx.z + blockIdx.z * blockDim.z) * blockDim.x * blockDim.y
```

9.2.2 warpSize

The `warpSize` constant contains the number of threads per warp for the given target device. On AMD hardware, this is referred to as `wavefront size`, which may vary depending on the architecture. For more details, see the hardware features.

Since `warpSize` can differ between devices, it can not be assumed to be a compile-time constant on the host. It has to be queried using `hipDeviceGetAttribute()` or `hipDeviceGetProperties()`, e.g.:

```
int warpSizeHost;
hipDeviceGetAttribute(&warpSizeHost, hipDeviceAttributeWarpSize, deviceId);
```

Note

`warpSize` should not be assumed to be a specific value in portable HIP applications. NVIDIA devices return 32 for this variable; AMD devices return 64 for gfx9 and 32 for gfx10 and above. HIP doesn't support `warpSize` of 64 on gfx10 and above. While code that assumes a `warpSize` of 32 can run on devices with a `warpSize` of 64, it only utilizes half of the compute resources.

Prior to ROCm 7.0, the warpSize parameter was a compile-time constant. Starting with ROCm 7.0, it is early folded by the compiler, allowing it to be used in loop bounds and enabling loop unrolling in a manner similar to a compile-time constant warp size.

If compile time warp size is required, for example to select the correct mask type or code path at compile time, the recommended approach is to determine the warp size of the GPU on host side and setup the kernel accordingly, as shown in the following block reduce example.

The `block_reduce` kernel has a template parameter for warp size and performs a reduction operation in two main phases:

- Shared memory reduction: Reduction is performed iteratively, halving the number of active threads each step until only a warp remains (32 or 64 threads, depending on the device).

- Warp-level reduction: Once the shared memory reduction completes, the remaining threads use warp-level shuffling to sum the remaining values. This is done efficiently with the `__shfl_down` intrinsic, which allows threads within the warp to exchange values without explicit synchronization.

WarpSize template parameter

```
template<std::uint32_t WarpSize>
using lane_mask_t = typename std::conditional<WarpSize == 32, std::uint32_t,␣
↪std::uint64_t>::type;

template<std::uint32_t WarpSize>
__global__ void block_reduce(int* input, lane_mask_t<WarpSize>* mask, int* output,␣
↪size_t size)
{
    extern __shared__ int shared[];

    // Read of input with bounds check
    auto read_global_safe = [&](const std::uint32_t i, const std::uint32_t lane_id,␣
↪const std::uint32_t mask_id)
    {
        lane_mask_t<WarpSize> warp_mask = lane_mask_t<WarpSize>(1) << lane_id;
        return (i < size) && (mask[mask_id] & warp_mask) ? input[i] : 0;
    };

    const std::uint32_t tid = threadIdx.x,
                        lid = threadIdx.x % WarpSize,
                        wid = threadIdx.x / WarpSize,
                        bid = blockIdx.x,
                        gid = bid * blockDim.x + tid;

    // Read input buffer to shared
    shared[tid] = read_global_safe(gid, lid, bid * (blockDim.x / WarpSize) + wid);
    __syncthreads();

    // Shared reduction
    for (std::uint32_t i = blockDim.x / 2; i >= WarpSize; i /= 2)
    {
        if (tid < i)
        shared[tid] = shared[tid] + shared[tid + i];
        __syncthreads();
    }

    // Use local variable in warp reduction
    int result =  shared[tid];
    __syncthreads();

    // This loop would be unrolled the same with the runtime warpSize.
    #pragma unroll
    for (std::uint32_t i = WarpSize/2; i >= 1; i /= 2)
    {
        result = result + __shfl_down(result, i);
    }
```

(continues on next page)

(continued from previous page)

```
    // Write result to output buffer
    if (tid == 0)
        output[bid] = result;
}
```

HIP warpSize

```
__global__ void block_reduce(int* input, std::uint64_t* mask, int* output, std::size_
↪t size)
{
    extern __shared__ int shared[];
    // Read of input with bounds check
    auto read_global_safe = [&](const std::uint32_t i, const std::uint32_t lane_id,␣
↪const std::uint32_t mask_id)
    {
        std::uint64_t warp_mask = 1ull << lane_id;
        return (i < size) && (mask[mask_id] & warp_mask) ? input[i] : 0;
    };

    const std::uint32_t tid = threadIdx.x,
                        lid = threadIdx.x % warpSize,
                        wid = threadIdx.x / warpSize,
                        bid = blockIdx.x,
                        gid = bid * blockDim.x + tid;

    // Read input buffer to shared
    shared[tid] = read_global_safe(gid, lid, bid * (blockDim.x / warpSize) + wid);
    __syncthreads();

    // Shared reduction
    for (std::uint32_t i = blockDim.x / 2; i >= warpSize; i /= 2)
    {
        if (tid < i)
        shared[tid] = shared[tid] + shared[tid + i];
        __syncthreads();
    }

    // Use local variable in warp reduction
    int result =  shared[tid];
    __syncthreads();

    // This loop would be unrolled the same with the compile-time WarpSize.
    #pragma unroll
    for (std::uint32_t i = warpSize/2; i >= 1; i /= 2) {
        result = result + __shfl_down(result, i);
    }

    // Write result to output buffer
    if (tid == 0)
        output[bid] = result;
}
```

The host code with the main function:

- Retrieves the warp size of the GPU (`warpSizeHost`) to determine the optimal kernel configuration.
- Allocates device memory (`d_data` for input, `d_results` for block-wise output) and initializes the input vector to 1.
- Generates the mask variables for every warp and copies them to the device.

WarpSize template parameter

```
template<std::uint32_t WarpSize>
void generate_and_copy_mask(
    void *d_mask,
    std::vector<int>& vectorExpected,
    int numOfBlocks,
    int numberOfWarp,
    int mask_size,
    int mask_element_size)
{
    std::random_device rd;
    std::mt19937_64 eng(rd());

    // Host side mask vector
    std::vector<lane_mask_t<WarpSize>> mask(mask_size);
    // Define uniform unsigned int distribution
    std::uniform_int_distribution<lane_mask_t<WarpSize>> distr;
    // Fill up the mask
    for(int i=0; i < numOfBlocks; i++)
    {
        int count = 0;
        for(int j=0; j < numberOfWarp; j++)
        {
            int mask_index = i * numberOfWarp + j;
            mask[mask_index] = distr(eng);
            if constexpr(WarpSize == 32)
                count += popcount(static_cast<std::uint32_t>(mask[mask_index]));
            else
                count += popcount(mask[mask_index]);
        }
        vectorExpected[i]= count;
    }

    // Copy the mask array
    HIP_CHECK(hipMemcpy(d_mask, mask.data(), mask_size * mask_element_size,␣
↪hipMemcpyHostToDevice));
}
```

HIP warpSize

```
void generate_and_copy_mask(
    std::uint64_t *d_mask,
    std::vector<int>& vectorExpected,
    int warpSizeHost,
```

(continues on next page)

(continued from previous page)

```
    int numOfBlocks,
    int numberOfWarp,
    int mask_size,
    int mask_element_size)
{
    std::random_device rd;
    std::mt19937_64 eng(rd());

    // Host side mask vector
    std::vector<std::uint64_t> mask(mask_size);
    // Define uniform unsigned int distribution
    std::uniform_int_distribution<std::uint64_t> distr;
    // Fill up the mask
    for(int i=0; i < numOfBlocks; i++)
    {
        int count = 0;
        for(int j=0; j < numberOfWarp; j++)
        {
            int mask_index = i * numberOfWarp + j;
            mask[mask_index] = distr(eng);
            if(warpSizeHost == 32)
                count += popcount(static_cast<std::uint32_t>(mask[mask_index]));
            else
                count += popcount(mask[mask_index]);
        }
        vectorExpected[i]= count;
    }
    // Copy the mask array
    HIP_CHECK(hipMemcpy(d_mask, mask.data(), mask_size * mask_element_size,␣
↪hipMemcpyHostToDevice));
}
```

- Selects the appropriate kernel specialization based on the warp size (either 32 or 64) and launches the kernel.

WarpSize template parameter

```
    // Fill up the mask variable, copy to device and select the right kernel.
    if(warpSizeHost == 32)
    {
        // Generate and copy mask arrays
        generate_and_copy_mask<32>(d_mask, vectorExpected, numOfBlocks,␣
↪numberOfWarp, mask_size, mask_element_size);

        // Start the kernel
        block_reduce<32><<<dim3(numOfBlocks), dim3(threadsPerBlock),␣
↪threadsPerBlock * sizeof(*d_data)>>>(
            d_data,
            static_cast<std::uint32_t*>(d_mask),
            d_results,
            arraySize);
    }
    else if(warpSizeHost == 64)
```

(continues on next page)

(continued from previous page)

```
    {
        // Generate and copy mask arrays
        generate_and_copy_mask<64>(d_mask, vectorExpected, numOfBlocks,␣
↪numberOfWarp, mask_size, mask_element_size);

        // Start the kernel
        block_reduce<64><<<dim3(numOfBlocks), dim3(threadsPerBlock),␣
↪threadsPerBlock * sizeof(*d_data)>>>(
            d_data,
            static_cast<std::uint64_t*>(d_mask),
            d_results,
            arraySize);
    }
    else
    {
        std::cerr << "Unsupported warp size." << std::endl;
        return EXIT_FAILURE;
    }
```

HIP warpSize

```
    // Generate and copy mask arrays
    generate_and_copy_mask(
        d_mask,
        vectorExpected,
        warpSizeHost,
        numOfBlocks,
        numberOfWarp,
        mask_size,
        mask_element_size);

    // Start the kernel
    block_reduce<<<dim3(numOfBlocks), dim3(threadsPerBlock), threadsPerBlock *␣
↪sizeof(*d_data)>>>(
        d_data,
        d_mask,
        d_results,
        arraySize);
```

- Synchronizes the device and copies the results back to the host.
- Checks that each block's sum is equal with the expected mask bit count, verifying the reduction's correctness.
- Frees the device memory to prevent memory leaks.

Note

The `warpSize` runtime example code is also provided for comparison purposes and the full example codes are located in the tools folder.

The variable `warpSize` can be used for loop bounds and supports loop unrolling similarly to the template parameter `WarpSize`.

For users who still require a compile-time constant warp size as a macro on the device side, it can be defined manually based on the target device architecture, as shown in the following example.

```
#if defined(__GFX8__) || defined(__GFX9__)
  #define WarpSize 64
#else
  #define WarpSize 32
#endif
```

Note

`mwavefrontsize64` compiler option is not supported by HIP runtime, that's why the architecture based compile time selector is an acceptable approach.

9.3 Vector types

These types are not automatically provided by the compiler. The `hip_vector_types.h` header, which is also included by `hip_runtime.h` has to be included to use these types.

9.3.1 Fundamental vector types

Fundamental vector types derive from the fundamental C++ integral and floating-point types. These types are defined in `hip_vector_types.h`, which is included by `hip_runtime.h`.

All vector types can be created with `1`, `2`, `3` or `4` elements, the corresponding type is `<fundamental_type>i`, where `i` is the number of elements.

All vector types support a constructor function of the form `make_<type_name>()`. For example, `float3 make_float3(float x, float y, float z)` creates a vector of type `float3` with value `(x,y,z)`. The elements of the vectors can be accessed using their members `x`, `y`, `z`, and `w`.

```
double2 d2_vec = make_double2(2.0, 4.0);
double first_elem = d2_vec.x;
```

HIP supports vectors created from the following fundamental types:

Integral Types	
char	uchar
short	ushort
int	uint
long	ulong
longlong	ulonglong
Floating-Point Types	
float	
double	

9.3.2 dim3

`dim3` is a special three-dimensional unsigned integer vector type that is commonly used to specify grid and group dimensions for kernel launch configurations.

Its constructor accepts up to three arguments. The unspecified dimensions are initialized to 1.

9.4 Built-in device functions

9.4.1 Memory fence instructions

HIP does not enforce strict ordering on memory operations, meaning, that the order in which memory accesses are executed, is not necessarily the order in which other threads observe these changes. So it can not be assumed, that data written by one thread is visible by another thread without synchronization.

Memory fences are a way to enforce a sequentially consistent order on the memory operations. This means, that all writes to memory made before a memory fence are observed by all threads after the fence. The scope of these fences depends on what specific memory fence is called.

HIP supports `__threadfence()`, `__threadfence_block()` and `__threadfence_system()`:

- `__threadfence_block()` orders memory accesses for all threads within a thread block.
- `__threadfence()` orders memory accesses for all threads on a device.
- `__threadfence_system()` orders memory accesses for all threads in the system, making writes to memory visible to other devices and the host

9.4.2 Synchronization functions

Synchronization functions cause all threads in a group to wait at this synchronization point until all threads reached it. These functions implicitly include a *threadfence*, thereby ensuring visibility of memory accesses for the threads in the group.

The `__syncthreads()` function comes in different versions.

`void __syncthreads()` simply synchronizes the threads of a block. The other versions additionally evaluate a predicate:

`int __syncthreads_count(int predicate)` returns the number of threads for which the predicate evaluates to non-zero.

`int __syncthreads_and(int predicate)` returns non-zero if the predicate evaluates to non-zero for all threads.

`int __syncthreads_or(int predicate)` returns non-zero if any of the predicates evaluates to non-zero.

The Cooperative Groups API offers options to synchronize threads on a developer defined set of thread groups. For further information, check the Cooperative Groups API reference or the *Cooperative Groups section in the programming guide*.

9.4.3 Math functions

HIP-Clang supports a set of math operations that are callable from the device. HIP supports most of the device functions supported by CUDA. These are described on Math API page.

9.4.4 Texture functions

The supported texture functions are listed in `texture_fetch_functions.h` and `texture_indirect_functions.h` header files in the HIP-AMD backend repository.

Texture functions are not supported on some devices. To determine if texture functions are supported on your device, use `Macro __HIP_NO_IMAGE_SUPPORT == 1`. You can query the attribute `hipDeviceAttributeImageSupport` to check if texture functions are supported in the host runtime code.

9.4.5 Surface functions

The supported surface functions are located on Surface object reference page.

9.4.6 Timer functions

HIP provides device functions to read a high-resolution timer from within the kernel.

The following functions count the cycles on the device, where the rate varies with the actual frequency.

```
clock_t clock()
long long int clock64()
```

> **Note**
>
> `clock()` and `clock64()` do not work properly on AMD RDNA3 (GFX11) graphic processors.

The difference between the returned values represents the cycles used.

```
__global__ void kernel()
{
  long long int start = clock64();
  // kernel code
  long long int stop = clock64();
  long long int cycles = stop - start;
}
```

`long long int wall_clock64()` returns the wall clock time on the device, with a constant, fixed frequency. The frequency is device dependent and can be queried using:

```
    int wallClkRate = 0; //in kilohertz
    HIP_CHECK(hipDeviceGetAttribute(&wallClkRate, hipDeviceAttributeWallClockRate,␣
↪deviceId));
```

9.4.7 Atomic functions

Atomic functions are read-modify-write (RMW) operations, whose result is visible to all other threads on the scope of the atomic operation, once the operation completes.

If multiple instructions from different devices or threads target the same memory location, the instructions are serialized in an undefined order.

Atomic operations in kernels can operate on block scope (i.e. shared memory), device scope (global memory), or system scope (system memory), depending on hardware support.

The listed functions are also available with the `_system` (e.g. `atomicAdd_system`) suffix, operating on system scope, which includes host memory and other GPUs' memory. The functions without suffix operate on shared or global memory on the executing device, depending on the memory space of the variable.

HIP supports the following atomic operations, where `TYPE` is one of `int`, `unsigned int`, `unsigned long`, `unsigned long long`, `float` or `double`, while `INTEGER` is `int`, `unsigned int`, `unsigned long`, `unsigned long long`:

Table 1: Atomic operations

```
TYPE atomicAdd(TYPE* address, TYPE val)
TYPE atomicSub(TYPE* address, TYPE val)
TYPE atomicMin(TYPE* address, TYPE val)
long long atomicMin(long long* address, long long val)
TYPE atomicMax(TYPE* address, TYPE val)
long long atomicMax(long long* address, long long val)
TYPE atomicExch(TYPE* address, TYPE val)
TYPE atomicCAS(TYPE* address, TYPE compare, TYPE val)
INTEGER atomicAnd(INTEGER* address, INTEGER val)
INTEGER atomicOr(INTEGER* address, INTEGER val)
INTEGER atomicXor(INTEGER* address, INTEGER val)
unsigned int atomicInc(unsigned int* address)
unsigned int atomicDec(unsigned int* address)
```

9.4.7.1 Unsafe floating-point atomic operations

Some HIP devices support fast atomic operations on floating-point values. For example, `atomicAdd` on single- or double-precision floating-point values may generate a hardware instruction that is faster than emulating the atomic operation using an atomic compare-and-swap (CAS) loop.

On some devices, fast atomic instructions can produce results that differ from the version implemented with atomic CAS loops. For example, some devices will use different rounding or denormal modes, and some devices produce incorrect answers if fast floating-point atomic instructions target fine-grained memory allocations.

The HIP-Clang compiler offers compile-time options to control the generation of unsafe atomic instructions. By default the compiler does not generate unsafe instructions. This is the same behaviour as with the `-mno-unsafe-fp-atomics` compilation flag. The `-munsafe-fp-atomics` flag indicates to the compiler that all floating-point atomic function calls are allowed to use an unsafe version, if one exists. For example, on some devices, this flag indicates to the compiler that no floating-point `atomicAdd` function can target fine-grained memory. These options are applied globally for the entire compilation.

HIP provides special functions that override the global compiler option for safe or unsafe atomic functions.

The `safe` prefix always generates safe atomic operations, even when `-munsafe-fp-atomics` is used, whereas `unsafe` always generates fast atomic instructions, even when `-mno-unsafe-fp-atomics`. The following table lists the safe and unsafe atomic functions, where `FLOAT_TYPE` is either `float` or `double`.

Table 2: AMD specific atomic operations

```
FLOAT_TYPE unsafeAtomicAdd(FLOAT_TYPE* address, FLOAT_TYPE val)
FLOAT_TYPE safeAtomicAdd(FLOAT_TYPE* address, FLOAT_TYPE val)
```

9.4.8 Warp cross-lane functions

Threads in a warp are referred to as `lanes` and are numbered from `0` to `warpSize - 1`. Warp cross-lane functions cooperate across all lanes in a warp. AMD GPUs guarantee, that all warp lanes are executed in lockstep, whereas NVIDIA GPUs that support Independent Thread Scheduling might require additional synchronization, or the use of the `__sync` variants.

Note that different devices can have different warp sizes. You should query the *warpSize* in portable code and not assume a fixed warp size.

All mask values returned or accepted by these built-ins are 64-bit unsigned integer values, even when compiled for a device with 32 threads per warp. On such devices the higher bits are unused. CUDA code ported to HIP requires changes to ensure that the correct type is used.

Note that the __sync variants are available in ROCm 7.0 (and enabled by default, unlike in previous versions). They can be disabled by setting the preprocessor macro HIP_DISABLE_WARP_SYNC_BUILTINS.

The _sync variants require a 64-bit unsigned integer mask argument that specifies the lanes of the warp that will participate. Each participating thread must have its own bit set in its mask argument, and all active threads specified in any mask argument must execute the same call with the same mask, otherwise the result is undefined. The implementation includes a static assert to check that the program source uses the correct type for the mask.

9.4.8.1 Warp vote and ballot functions

```
int __all(int predicate)
int __any(int predicate)
unsigned long long __ballot(int predicate)
unsigned long long __activemask()

int __all_sync(unsigned long long mask, int predicate)
int __any_sync(unsigned long long mask, int predicate)
unsigned long long __ballot_sync(unsigned long long mask, int predicate)
```

You can use __any and __all to get a summary view of the predicates evaluated by the participating lanes.

- __any(): Returns 1 if the predicate is non-zero for any participating lane, otherwise it returns 0.
- __all(): Returns 1 if the predicate is non-zero for all participating lanes, otherwise it returns 0.

To determine if the target platform supports the any/all instruction, you can query the hasWarpVote device property on the host or use the HIP_ARCH_HAS_WARP_VOTE compiler definition in device code.

__ballot returns a bit mask containing the 1-bit predicate value from each lane. The nth bit of the result contains the bit contributed by the nth lane.

__activemask() returns a bit mask of currently active warp lanes. The nth bit of the result is 1 if the nth lane is active.

Note that the __ballot and __activemask built-ins in HIP have a 64-bit return value (unlike the 32-bit value returned by the CUDA built-ins). Code ported from CUDA should be adapted to support the larger warp sizes that the HIP version requires.

Applications can test whether the target platform supports the __ballot or __activemask instructions using the hasWarpBallot device property in host code or the HIP_ARCH_HAS_WARP_BALLOT macro defined by the compiler for device code.

9.4.8.2 Warp match functions

```
unsigned long long __match_any(T value)
unsigned long long __match_all(T value, int *pred)

unsigned long long __match_any_sync(unsigned long long mask, T value)
unsigned long long __match_all_sync(unsigned long long mask, T value, int *pred)
```

T can be a 32-bit integer type, 64-bit integer type or a single precision or double precision floating point type.

__match_any returns a bit mask where the n-th bit is set to 1 if the n-th lane has the same value as the current lane, and 0 otherwise.

__match_all returns a bit mask with the bits of the participating lanes are set to 1 if all lanes have the same value, and 0 otherwise. The predicate pred is set to true if all participating threads have the same value, and false otherwise.

9.4.8.3 Warp shuffle functions

```
T __shfl      (T var, int srcLane, int width=warpSize);
T __shfl_up   (T var, unsigned int delta, int width=warpSize);
T __shfl_down (T var, unsigned int delta, int width=warpSize);
T __shfl_xor  (T var, int laneMask, int width=warpSize);

T __shfl_sync      (unsigned long long mask, T var, int srcLane, int width=warpSize);
T __shfl_up_sync   (unsigned long long mask, T var, unsigned int delta, int␣
↪width=warpSize);
T __shfl_down_sync (unsigned long long mask, T var, unsigned int delta, int␣
↪width=warpSize);
T __shfl_xor_sync  (unsigned long long mask, T var, int laneMask, int width=warpSize);
```

`T` can be a 32-bit integer type, 64-bit integer type or a single precision or double precision floating point type.

The warp shuffle functions exchange values between threads within a warp.

The optional `width` argument specifies subgroups, in which the warp can be divided to share the variables. It has to be a power of two smaller than or equal to `warpSize`. If it is smaller than `warpSize`, the warp is grouped into separate groups, that are each indexed from 0 to width as if it was its own entity, and only the lanes within that subgroup participate in the shuffle. The lane indices in the subgroup are given by `laneIdx % width`.

The different shuffle functions behave as following:

__shfl
: The thread reads the value from the lane specified in `srcLane`.

__shfl_up
: The thread reads `var` from lane `laneIdx - delta`, thereby "shuffling" the values of the lanes of the warp "up". If the resulting source lane is out of range, the thread returns its own `var`.

__shfl_down
: The thread reads `var` from lane `laneIdx + delta`, thereby "shuffling" the values of the lanes of the warp "down". If the resulting source lane is out of range, the thread returns its own `var`.

__shfl_xor
: The thread reads `var` from lane `laneIdx xor lane_mask`. If `width` is smaller than `warpSize`, the threads can read values from subgroups before the current subgroup. If it tries to read values from later subgroups, the function returns the `var` of the calling thread.

9.4.8.4 Warp reduction functions

Arithmetic reduces:

```
T __reduce_add_sync (unsigned long long mask, T var);
T __reduce_min_sync (unsigned long long mask, T var);
T __reduce_max_sync (unsigned long long mask, T var);
```

`T` can be:

- On Nvidia platform: `int` or `unsigned int`
- On AMD platform: `int` or `unsigned int`; if the user defines the macro `HIP_ENABLE_EXTRA_WARP_SYNC_TYPES`, then: `unsigned long long`, `long long`, `half`/`single`/`double` precision floating

point types are also be supported.

Returns the aggregated result of the arithmetic operation, where each of the participating threads (i.e. the ones mentioned on the mask) contribute `var`.

NOTE: for type `half`, these intrinsics are not available in environments where the arithmetic operators are not available for that type.

Logical reduces:

```
T __reduce_and_sync (unsigned long long mask, T var);
T __reduce_or_sync  (unsigned long long mask, T var);
T __reduce_xor_sync (unsigned long long mask, T var);
```

`T` can be:

- On Nvidia platform: `unsigned int`
- On AMD platform: `unsigned int`, and if the user defines the macro `HIP_ENABLE_EXTRA_WARP_SYNC_TYPES`, then `int`, `unsigned long long` or `long long` are also supported

Returns the result of the aggregated logical AND/OR/XOR operation where each of the participating threads (i.e. the ones mentioned on the mask) contribute `var`.

The mask argument is a 64-bit unsigned integer that specifies the lanes in the warp that participate in cross-lane communication with the calling lane. Each participating thread must have its own bit set in its mask argument, and all active threads specified in any mask argument must execute the same call with the same mask, otherwise the result is undefined.

Informational note: On the AMD platform, **masks that start from lane zero and have no "holes" use faster cross-lane operations and exhibit better performance** than masks with "holes" (example of mask with no holes: 0xFF and with holes: 0xFB; the reduction with 0xFF is faster).

These functions do not provide a memory barrier on any platform.

9.4.8.5 Warp matrix functions

Warp matrix functions allow a warp to cooperatively operate on small matrices that have elements spread over lanes in an unspecified manner.

HIP does not support warp matrix types or functions.

9.4.9 Cooperative groups functions

You can use cooperative groups to synchronize groups of threads across thread blocks. It also provide a way of communicating between these groups.

For further information, check the Cooperative Groups API reference or the *Cooperative Groups programming guide*.

CHAPTER

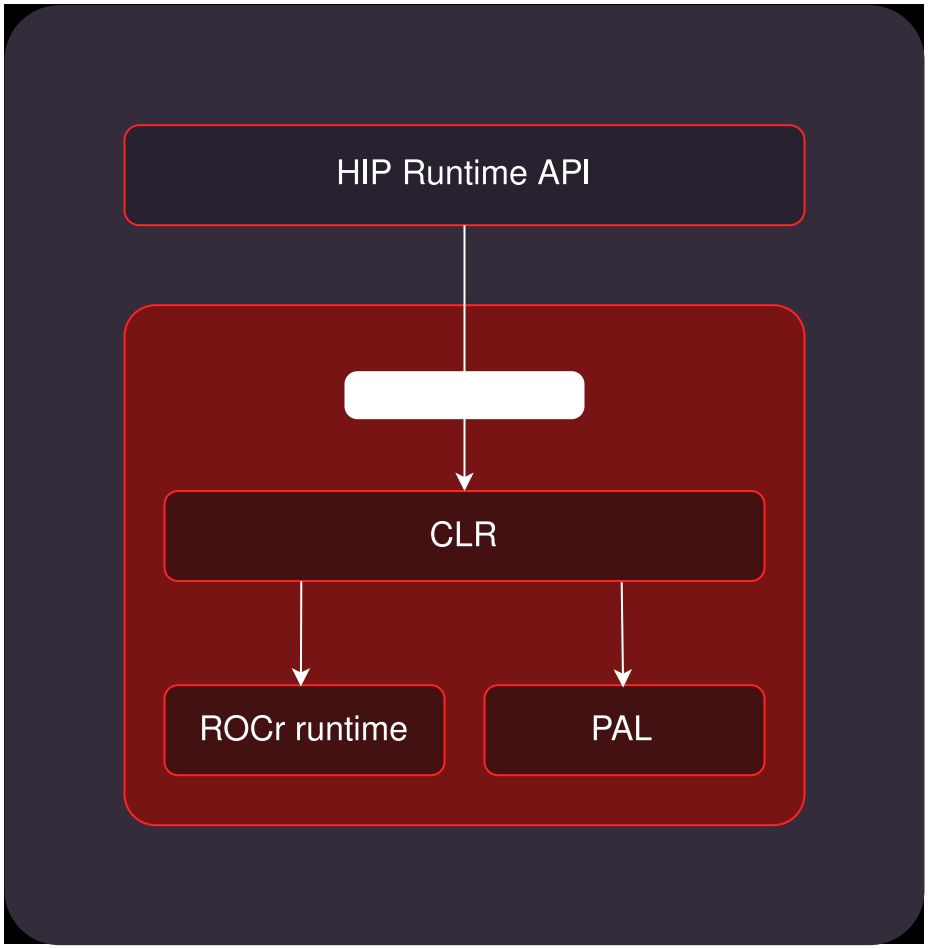
TEN

USING HIP RUNTIME API

The HIP runtime API provides C and C++ functionalities to manage event, stream, and memory on GPUs. The HIP runtime uses Compute Language Runtime (CLR).

CLR contains source code for the AMD ROCm compute language runtimes: HIP and OpenCL™`. CLR includes the HIP implementation on the AMD ROCm platform: hipamd and the ROCm Compute Language Runtime (`rocclr`). `rocclr` is a virtual device interface that enables the HIP runtime to interact with different backends, such as ROCr on Linux or PAL on Microsoft Windows. CLR also includes the OpenCL runtime implementation.

The HIP runtime API backends are summarized in the following illustration:



The following is a list of HIP runtime API high-level functions:

- *Initialization*
- *Memory management*
- *Error handling*
- *Asynchronous concurrent execution*
- *Cooperative groups*
- *HIP graphs*
- *Call stack*
- *OpenGL interoperability*
- *External resource interoperability*

10.1 Initialization

The initialization involves setting up the environment and resources needed for using GPUs. The following steps are covered with the initialization:

- Setting up the HIP runtime

  This includes reading the environment variables set during init, setting up the active or visible devices, loading necessary libraries, setting up internal buffers for memory copies or cooperative launches, initialize the compiler as well as HSA runtime and checks any errors due to lack of resources or no active devices.

- Querying and setting GPUs

  Identifying and querying the available GPU devices on the system.

- Setting up contexts

  Creating contexts for each GPU device, which are essential for managing resources and executing kernels. For further details, check the context section.

10.1.1 Initialize the HIP runtime

The HIP runtime is initialized automatically when the first HIP API call is made. However, you can explicitly initialize it using `hipInit()`, to be able to control the timing of the initialization. The manual initialization can be useful to ensure that the GPU is initialized and ready, or to isolate GPU initialization time from other parts of your program.

> **Note**
>
> You can use `hipDeviceReset()` to delete all streams created, memory allocated, kernels running and events created by the current process. Any new HIP API call initializes the HIP runtime again.

10.1.2 Querying and setting GPUs

If multiple GPUs are available in the system, you can query and select the desired GPU(s) to use based on device properties, such as size of global memory, size shared memory per block, support of cooperative launch and support of managed memory.

10.1.2.1 Querying GPUs

The properties of a GPU can be queried using `hipGetDeviceProperties()`, which returns a struct of `hipDeviceProp_t`. The properties in the struct can be used to identify a device or give an overview of hardware characteristics, that might make one GPU better suited for the task than others.

The `hipGetDeviceCount()` function returns the number of available GPUs, which can be used to loop over the available GPUs.

Example code of querying GPUs:

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

int main()
{
    int deviceCount;
    if (hipGetDeviceCount(&deviceCount) == hipSuccess)
    {
        for (int i = 0; i < deviceCount; ++i)
        {
            hipDeviceProp_t prop;
            if (hipGetDeviceProperties(&prop, i) == hipSuccess)
                std::cout << "Device" << i << prop.name << std::endl;
        }
    }

    return 0;
}
```

10.1.2.2 Setting the GPU

`hipSetDevice()` function select the GPU to be used for subsequent HIP operations. This function performs several key tasks:

- Context Binding

  Binds the current thread to the context of the specified GPU device. This ensures that all subsequent operations are executed on the selected device.

- Resource Allocation

  Prepares the device for resource allocation, such as memory allocation and stream creation.

- Check device availability

  Checks for errors in device selection and returns error if the specified device is not available or not capable of executing HIP operations.

10.2 Memory management

Memory management is an important part of the HIP runtime API, when creating high-performance applications. Both allocating and copying memory can result in bottlenecks, which can significantly impact performance.

The programming model is based on a system with a host and a device, each having its own distinct memory. Kernels operate on *Device memory*, while host functions operate on *Host memory*.

The runtime offers functions for allocating, freeing, and copying device memory, along with transferring data between host and device memory.

Here are the various memory management techniques:

- *Coherence control*
- *Unified memory management*
- *Virtual memory management*
- *Stream-ordered memory allocator*

10.2.1 Memory allocation

The API calls and the resulting allocations are listed here:

Table 1: Memory coherence control

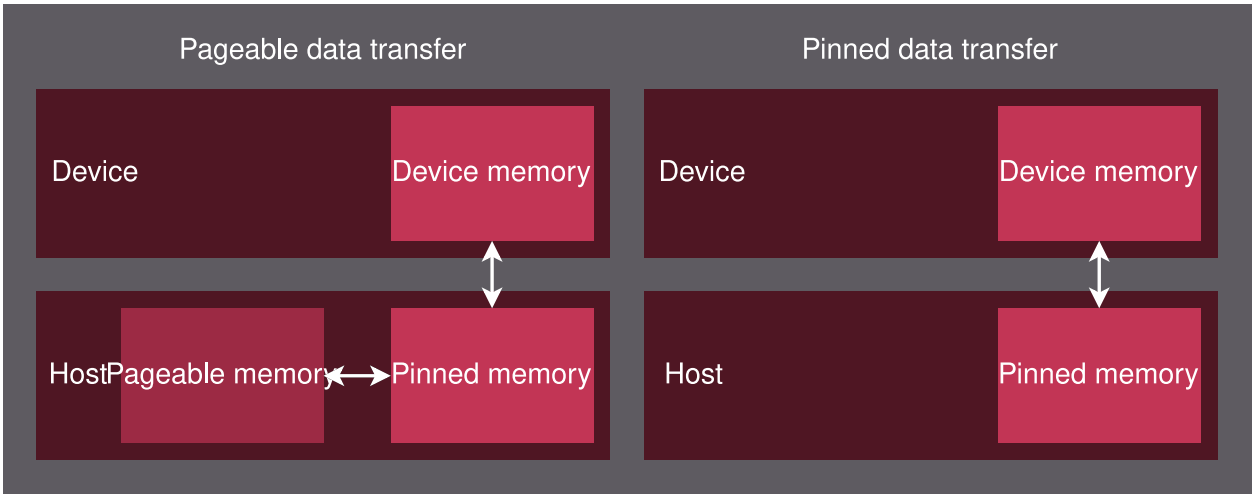
API	Data location	Allocation
System allocated	Host	*Pageable*
`hipMallocManaged()`	Host	*Managed*
`hipHostMalloc()`	Host	*Pinned*
`hipMalloc()`	Device	Pinned

10.2.2 Host memory

Host memory is the "normal" memory residing in the host RAM and allocated by C or C++. Host memory can be allocated in two different ways:

- Pageable memory
- Pinned memory

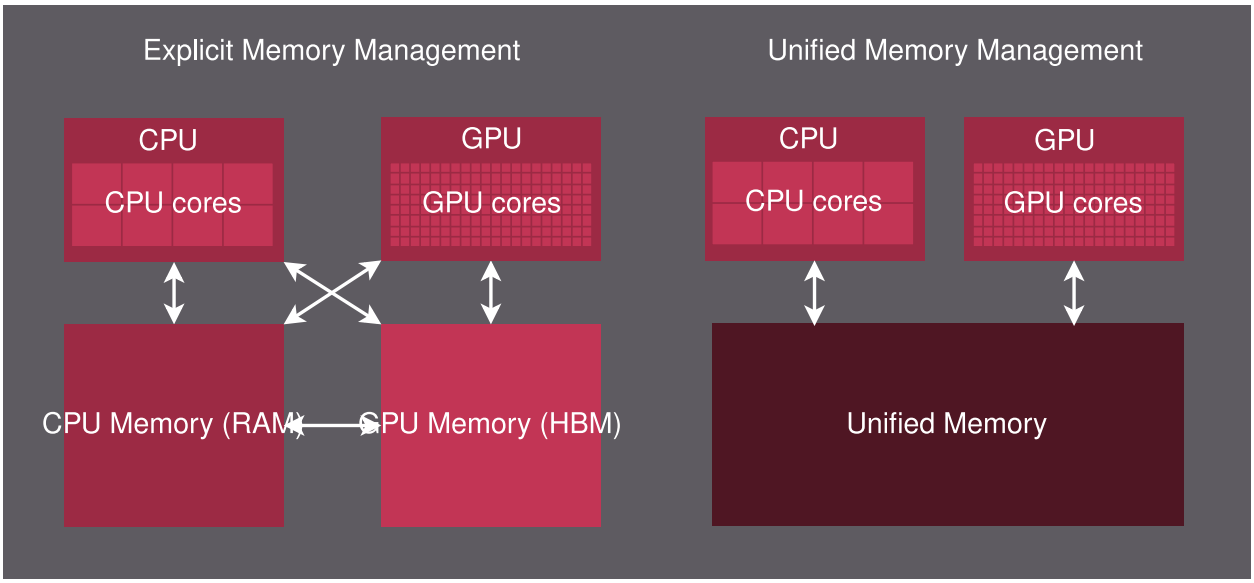
The following figure explains how data is transferred in pageable and pinned memory.



The pageable and pinned memory allow you to exercise direct control over memory operations, which is known as explicit memory management. When using the unified memory, you get a simplified memory model with less control over low level memory operations.

The difference in memory transfers between explicit and unified memory management is highlighted in the following figure:

For more details on unified memory management, see *Unified memory management*.



10.2.2.1 Pageable memory

Pageable memory exists on memory blocks known as "pages" that can be migrated to other memory storage. For example, migrating memory between CPU sockets on a motherboard or in a system whose RAM runs out of space and starts dumping pages into the swap partition of the hard drive.

Pageable memory is usually allocated with a call to `malloc` or `new` in a C++ application.

Example: Using pageable host memory in HIP

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

#include <cstring>
#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;    \
    }                                          \
}

int main()
{
    const int element_number = 100;

    int *host_input, *host_output;
    // Host allocation
    host_input  = new int[element_number];
```

(continues on next page)

(continued from previous page)

```
    host_output = new int[element_number];

    // Host data preparation
    for (int i = 0; i < element_number; i++) {
        host_input[i] = i;
    }
    std::memset(host_output, 0, element_number * sizeof(int));

    int *device_input, *device_output;

    // Device allocation
    HIP_CHECK(hipMalloc((int **)&device_input,  element_number * sizeof(int)));
    HIP_CHECK(hipMalloc((int **)&device_output, element_number * sizeof(int)));

    // Device data preparation
    HIP_CHECK(hipMemcpy(device_input, host_input, element_number * sizeof(int),␣
↪hipMemcpyHostToDevice));
    HIP_CHECK(hipMemset(device_output, 0, element_number * sizeof(int)));

    // Run the kernel
    // ...

    HIP_CHECK(hipMemcpy(device_input, host_input, element_number * sizeof(int),␣
↪hipMemcpyHostToDevice));

    // Free host memory
    delete[] host_input;
    delete[] host_output;

    // Free device memory
    HIP_CHECK(hipFree(device_input));
    HIP_CHECK(hipFree(device_output));
}
```

Note

`hipMalloc()` and `hipFree()` are blocking calls. However, HIP also provides non-blocking versions `hipMallocAsync()` and `hipFreeAsync()`, which require a stream as an additional argument. For asynchronous memory allocations made with `hipMallocAsync` and `hipMallocFromPoolAsync` `hipFree` does not implicitly wait for synchronization, to match the behavior of `cudaFree`.

10.2.2.2 Pinned memory

Pinned memory or page-locked memory is stored in pages that are locked in specific sectors in RAM and can't be migrated. The pointer can be used on both host and device. Accessing host-resident pinned memory in device kernels is generally not recommended for performance, as it can force the data to traverse the host-device interconnect such as PCIe, which is much slower than the on-device bandwidth.

The advantage of pinned memory is the improved transfer time between host and device. For transfer operations, such as `hipMemcpy()` or `hipMemcpyAsync()`, using pinned memory instead of pageable memory on the host can lead to a three times improvement in bandwidth.

The disadvantage of pinned memory is the reduced availability of RAM for other processes, which can negatively impact the overall performance of the host.

Example: Using pinned memory in HIP

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

#include <cstring>
#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;    \
    }                                          \
}

int main()
{
    const int element_number = 100;

    int *host_input, *host_output;
    // Host allocation
    HIP_CHECK(hipHostMalloc(&host_input, element_number * sizeof(int)));
    HIP_CHECK(hipHostMalloc(&host_output, element_number * sizeof(int)));

    // Host data preparation
    for (int i = 0; i < element_number; i++)
    {
        host_input[i] = i;
    }
    std::memset(host_output, 0, element_number * sizeof(int));

    int *device_input, *device_output;

    // Device allocation
    HIP_CHECK(hipMalloc(&device_input,  element_number * sizeof(int)));
    HIP_CHECK(hipMalloc(&device_output, element_number * sizeof(int)));

    // Device data preparation
    HIP_CHECK(hipMemcpy(device_input, host_input, element_number * sizeof(int),␣
↪hipMemcpyHostToDevice));
    HIP_CHECK(hipMemset(device_output, 0, element_number * sizeof(int)));

    // Run the kernel
    // ...

    HIP_CHECK(hipMemcpy(device_input, host_input, element_number * sizeof(int),␣
```

(continues on next page)

(continued from previous page)

```
↪hipMemcpyHostToDevice));

    // Free host memory
    HIP_CHECK(hipFreeHost(host_input));
    HIP_CHECK(hipFreeHost(host_output));

    // Free device memory
    HIP_CHECK(hipFree(device_input));
    HIP_CHECK(hipFree(device_output));
}
```

10.2.2.2.1 Memory allocation flags for pinned memory

The memory allocation for pinned memory can be controlled using `hipHostMalloc` flags:

- `hipHostMallocPortable`: The memory allocation is not restricted to the context making the allocation.
- `hipHostMallocMapped`: The memory is allocated into the address space for the current device and the device pointer can be obtained with `hipHostGetDevicePointer()`.
- `hipHostMallocNumaUser`: The host memory allocation follows Numa policy specified by the user. Target of Numa policy is to select a CPU that is closest to each GPU. Numa distance is the distance between GPU and CPU devices.
- `hipHostMallocWriteCombined`: The memory is allocated as write-combined. Although lacking read efficiency by most CPUs, write-combined allocation might be transferred faster across the PCIe bus on some system configurations. It's a good option for data transfer from host to device via mapped pinned memory.
- `hipHostMallocCoherent`: Fine-grained memory is allocated. Overrides `HIP_HOST_COHERENT` environment variable for specific allocation. For details, see *Coherence control*.
- `hipHostMallocNonCoherent`: Coarse-grained memory is allocated. Overrides `HIP_HOST_COHERENT` environment variable for specific allocation. For details, see *Coherence control*.

All allocation flags are independent and can be set in any combination. The only exception is setting `hipHostMallocCoherent` and `hipHostMallocNonCoherent` together, which leads to an illegal state. An example of a valid flag combination is calling `hipHostMalloc()` with both `hipHostMallocPortable` and `hipHostMallocMapped` flags set. Both the flags use the same model and differentiate only between how the surrounding code uses the host memory.

Note

By default, each GPU selects a Numa CPU node with the least Numa distance between them. This implies that the host memory is automatically allocated on the closest memory pool of the current GPU device's Numa node. Using `hipSetDevice()` API to set a different GPU increases the Numa distance but still allows you to access the host allocation.

Numa policy is implemented on Linux and is under development on Microsoft Windows.

10.2.3 Device memory

Device memory is random access memory that is physically located on a GPU. In general it is memory with a bandwidth that is an order of magnitude higher compared to RAM available to the host. That high bandwidth is only available to on-device accesses, accesses from the host or other devices have to go over a special interface which is considerably slower, usually the PCIe bus or the AMD Infinity Fabric.

On certain architectures like APUs, the GPU and CPU share the same physical memory.

There is also a special local data share on-chip directly accessible to the compute units, that can be used for shared memory.

The physical device memory can be used to back up several different memory spaces in HIP, as described in the following.

10.2.3.1 Global memory

Global memory is the general read-write accessible memory visible to all threads on a given device. Since variables located in global memory have to be marked with the `__device__` qualifier, this memory space is also referred to as device memory.

Without explicitly copying it, it can only be accessed by the threads within a kernel operating on the device, however *Unified memory management* can be used to let the runtime manage this, if desired.

10.2.3.1.1 Allocating global memory

This memory needs to be explicitly allocated.

It can be allocated from the host via the HIP runtime memory management functions like `hipMalloc()`, or can be defined using the `__device__` qualifier on variables.

It can also be allocated within a kernel using `malloc` or `new`. The specified amount of memory is allocated by each thread that executes the instructions. The recommended way to allocate the memory depends on the use case. If the memory is intended to be shared between the threads of a block, it is generally beneficial to allocate one large block of memory, due to the way the memory is accessed.

> **Note**
>
> Memory allocated within a kernel can only be freed in kernels, not by the HIP runtime on the host, like `hipFree()`. It is also not possible to free device memory allocated on the host, with `hipMalloc()` for example, in a kernel.

An example for how to share memory allocated within a kernel by only one thread is given in the following example. In case the device memory is only needed for communication between the threads in a single block, *Shared memory* is the better option, but is also limited in size.

```
__global__ void kernel_memory_allocation(TYPE* pointer){
  // The pointer is stored in shared memory, so that all
  // threads of the block can access the pointer
  __shared__ int *memory;

  size_t blockSize = blockDim.x;
  constexpr size_t elementsPerThread = 1024;
  if(threadIdx.x == 0){
    // allocate memory in one contiguous block
    memory = new int[blockDim.x * elementsPerThread];
  }
  __syncthreads();

  // load pointer into thread-local variable to avoid
  // unnecessary accesses to shared memory
  int *localPtr = memory;

  // work with allocated memory, e.g. initialization
  for(int i = 0; i < elementsPerThread; ++i){
    // access in a contiguous way
```

(continues on next page)

(continued from previous page)

```
    localPtr[i * blockSize + threadIdx.x] = i;
  }

  // synchronize to make sure no thread is accessing the memory before freeing
  __syncthreads();
  if(threadIdx.x == 0){
    delete[] memory;
  }
}
```

10.2.3.1.2 Copying between device and host

When not using *Unified memory management*, memory has to be explicitly copied between the device and the host, using the HIP runtime API.

```
size_t elements = 1 << 20;
size_t size_bytes = elements * sizeof(int);

// allocate host and device memory
int *host_pointer = new int[elements];
int *device_input, *device_result;
HIP_CHECK(hipMalloc(&device_input, size_bytes));
HIP_CHECK(hipMalloc(&device_result, size_bytes));

// copy from host to the device
HIP_CHECK(hipMemcpy(device_input, host_pointer, size_bytes, hipMemcpyHostToDevice));

// Use memory on the device, i.e. execute kernels

// copy from device to host, to e.g. get results from the kernel
HIP_CHECK(hipMemcpy(host_pointer, device_result, size_bytes, hipMemcpyDeviceToHost));

// free memory when not needed any more
HIP_CHECK(hipFree(device_result));
HIP_CHECK(hipFree(device_input));
delete[] host_pointer;
```

10.2.3.2 Constant memory

Constant memory is read-only storage visible to all threads on a given device. It is a limited segment backed by device memory, that takes a different caching route than normal device memory accesses. It needs to be set by the host before kernel execution.

In order to get the highest bandwidth from the constant memory, all threads of a warp have to access the same memory address. If they access different addresses, the accesses get serialized and the bandwidth is therefore reduced.

10.2.3.2.1 Using constant memory

Constant memory can not be dynamically allocated, and the size has to be specified during compile time. If the values can not be specified during compile time, they have to be set by the host before the kernel, that accesses the constant memory, is called.

```
constexpr size_t const_array_size = 32;
__constant__ double const_array[const_array_size];

void set_constant_memory(double* values){
  hipMemcpyToSymbol(const_array, values, const_array_size * sizeof(double));
}

__global__ void kernel_using_const_memory(double* array){

  int warpIdx = threadIdx.x / warpSize;
  // uniform access of warps to const_array for best performance
  array[blockDim.x] *= const_array[warpIdx];
}
```

10.2.3.3 Texture memory

Texture memory is special read-only memory visible to all threads on a given device and accessible through additional APIs. Its origins come from graphics APIs, and provides performance benefits when accessing memory in a pattern where the addresses are close to each other in a 2D or 3D representation of the memory. It also provides additional features like filtering and addressing for out-of-bounds accesses, which are further explained in *Texture fetching*.

The original use of the texture cache was also to take pressure off the global memory and other caches, however on modern GPUs, that support textures, the L1 cache and texture cache are combined, so the main purpose is to make use of the texture specific features.

To find out whether textures are supported on a device, query `hipDeviceAttributeImageSupport`.

10.2.3.3.1 Using texture memory

Textures are more complex than just a region of memory, so their layout has to be specified. They are represented by `hipTextureObject_t` and created using `hipCreateTextureObject()`.

The underlying memory is a 1D, 2D or 3D `hipArray_t`, that needs to be allocated using `hipMallocArray()`.

On the device side, texture objects are accessed using the `tex1D/2D/3D` functions.

The texture management functions can be found in the Texture management API reference

A full example for how to use textures can be found in the ROCm texture management example

10.2.3.4 Surface memory

A read-write version of texture memory. It is created in the same way as a texture, but with `hipCreateSurfaceObject()`.

Since surfaces are also cached in the read-only texture cache, the changes written back to the surface can't be observed in the same kernel. A new kernel has to be launched in order to see the updated surface.

The corresponding functions are listed in the Surface object API reference.

10.2.3.5 Shared memory

Shared memory is read-write memory, that is only visible to the threads within a block. It is allocated per thread block, and needs to be either statically allocated at compile time, or can be dynamically allocated when launching the kernel, but not during kernel execution. Its general use-case is to share variables between the threads within a block, but can also be used as scratch pad memory.

Shared memory is not backed by the same physical memory as the other address spaces. It is on-chip memory local to the compute units, providing low-latency, high-bandwidth access, comparable to the L1 cache. It is however limited in size, and as it is allocated per block, can restrict how many blocks can be scheduled to a compute unit concurrently, thereby potentially reducing occupancy.

An overview of the size of the local data share (LDS), that backs up shared memory, is given in the GPU hardware specifications.

10.2.3.5.1 Allocate shared memory

Memory can be dynamically allocated by declaring an `extern __shared__` array, whose size can be set during kernel launch, which can then be accessed in the kernel.

```
extern __shared__ int dynamic_shared[];
__global__ void kernel(int array1SizeX, int array1SizeY, int array2Size){
  // at least (array1SizeX * array1SizeY + array2Size) * sizeof(int) bytes
  // dynamic shared memory need to be allocated when the kernel is launched
  int* array1 = dynamic_shared;
  // array1 is interpreted as 2D of size:
  int array1Size = array1SizeX * array1SizeY;

  int* array2 = &(array1[array1Size]);

  if(threadIdx.x < array1SizeX && threadIdx.y < array1SizeY){
    // access array1 with threadIdx.x + threadIdx.y * array1SizeX
  }
  if(threadIdx.x < array2Size){
    // access array2 threadIdx.x
  }
}
```

A more in-depth example on dynamically allocated shared memory can be found in the ROCm dynamic shared example.

To statically allocate shared memory, just declare it in the kernel. The memory is allocated per block, not per thread. If the kernel requires more shared memory than is available to the architecture, the compilation fails.

```
__global__ void kernel(){
  __shared__ int array[128];
  __shared__ double result;
}
```

A more in-depth example on statically allocated shared memory can be found in the ROCm shared memory example.

10.2.3.6 Texture fetching

Textures give access to specialized hardware on GPUs that is usually used in graphics processing. In particular, textures use a different way of accessing their underlying device memory. Memory accesses to textures are routed through a special read-only texture cache, that is optimized for logical spatial locality, e.g. locality in 2D grids. This can also benefit certain algorithms used in GPGPU computing, when the access pattern is the same as used when accessing normal textures.

Additionally, textures can be indexed using floating-point values. This is used in graphics applications to interpolate between neighboring values of a texture. Depending on the interpolation mode the index can be in the range of `0` to `size - 1` or `0` to `1`. Textures also have a way of handling out-of-bounds accesses.

Depending on the value of the index, *texture filtering* or *texture addressing* is performed.

Here is the example texture used in this document for demonstration purposes. It is 2x2 texels and indexed in the [0 to 1] range.

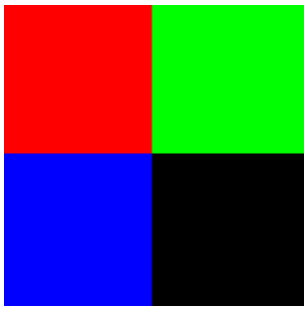


Fig. 1: Texture used as example

In HIP textures objects are of type `hipTextureObject_t` and created using `hipCreateTextureObject()`.

For a full list of available texture functions see the HIP texture API reference.

A code example for how to use textures can be found in the ROCm texture management example

10.2.3.6.1 Texture filtering

Texture filtering handles the usage of fractional indices. When the index is a fraction, the queried value lies between two or more texels (texture elements), depending on the dimensionality of the texture. The filtering method defines how to interpolate between these values.

The filter modes are specified in `hipTextureFilterMode`.

The various texture filtering methods are discussed in the following sections.

10.2.3.6.1.1 Nearest point filtering

This filter mode corresponds to `hipFilterModePoint`.

In this method, the modulo of index is calculated as:

```
tex(x) = T[floor(x)]
```

This is also applicable for 2D and 3D variants.

This doesn't interpolate between neighboring values, which results in a pixelated look.

The following image shows a texture stretched to a 4x4 pixel quad but still indexed in the [0 to 1] range. The in-between values are the same as the values of the nearest texel.

10.2.3.6.1.2 Linear filtering

This filter mode corresponds to `hipFilterModeLinear`.

The linear filtering method does a linear interpolation between values. Linear interpolation is used to create a linear transition between two values. The formula used is `(1-t)P1 + tP2` where `P1` and `P2` are the values and `t` is within the [0 to 1] range.

In the case of linear texture filtering the following formulas are used:

- For one dimensional textures: `tex(x) = (1-α)T[i] + αT[i+1]`
- For two dimensional textures: `tex(x,y) = (1-α)(1-β)T[i,j] + α(1-β)T[i+1,j] + (1-α)βT[i,j+1] + αβT[i+1,j+1]`

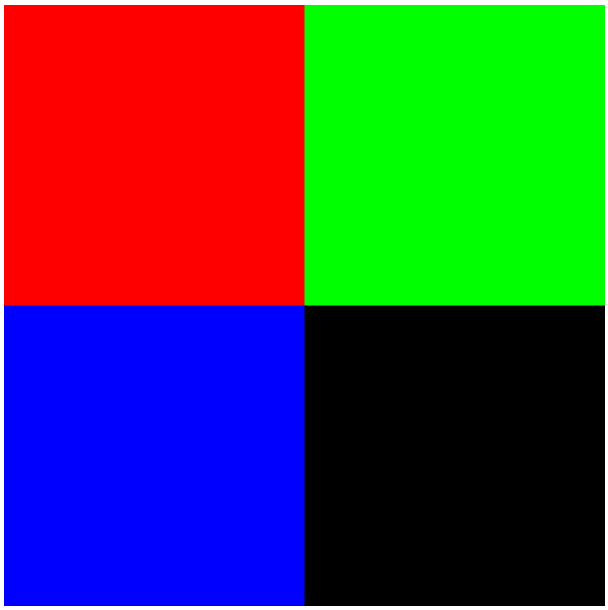


Fig. 2: Texture upscaled with nearest point filtering

- For three dimensional textures: `tex(x,y,z) = (1−α)(1−β)(1−γ)T[i,j,k] + α(1−β)(1−γ)T[i+1,j,k] + (1−α)β(1−γ)T[i,j+1,k] + αβ(1−γ)T[i+1,j+1,k] + (1−α)(1−β)γT[i,j,k+1] + α(1−β)γT[i+1,j,k+1] + (1−α)βγT[i,j+1,k+1] + αβγT[i+1,j+1,k+1]`

Where x, y, and, z are the floating-point indices. i, j, and, k are the integer indices and, α, β, and, γ values represent how far along the sampled point is on the three axes. These values are calculated by these formulas: `i = floor(x'), α = frac(x'), x' = x − 0.5, j = floor(y'), β = frac(y'), y' = y − 0.5, k = floor(z'), γ = frac(z')` and `z' = z − 0.5`

The following image shows a texture stretched out to a 4x4 pixel quad, but still indexed in the [0 to 1] range. The in-between values are interpolated between the neighboring texels.

10.2.3.6.2 Texture addressing

The texture addressing modes are specified in `hipTextureAddressMode`.

The texture addressing mode handles out-of-bounds accesses to the texture. This can be used in graphics applications to e.g. repeat a texture on a surface multiple times in various ways or create visible signs of out-of-bounds indexing.

The following sections describe the various texture addressing methods.

10.2.3.6.2.1 Address mode border

This addressing mode is set using `hipAddressModeBorder`.

This addressing mode returns a border value when indexing out of bounds. The border value must be set before texture fetching.

The following image shows the texture on a 4x4 pixel quad, indexed in the [0 to 3] range. The out-of-bounds values are the border color, which is yellow.

The purple lines are not part of the texture. They only denote the edge, where the addressing begins.

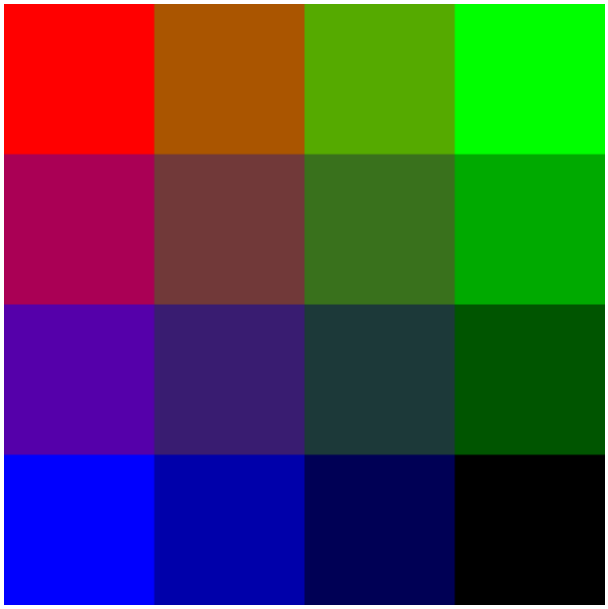


Fig. 3: Texture upscaled with linear filtering

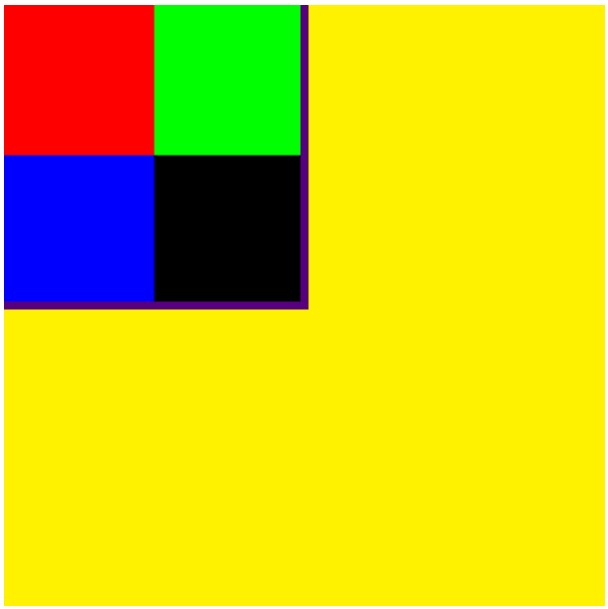


Fig. 4: Texture with yellow border color.

10.2.3.6.2.2 Address mode clamp

This addressing mode is set using `hipAddressModeClamp`.

This mode clamps the index between [0 to size-1]. Due to this, when indexing out-of-bounds, the values on the edge of the texture repeat. The clamp mode is the default addressing mode.

The following image shows the texture on a 4x4 pixel quad, indexed in the [0 to 3] range. The out-of-bounds values are repeating the values at the edge of the texture.

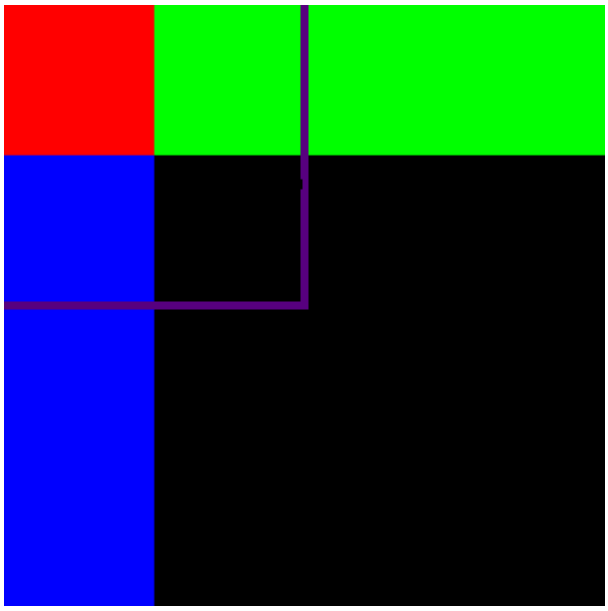


Fig. 5: Texture with clamp addressing

The purple lines are not part of the texture. They only denote the edge, where the addressing begins.

10.2.3.6.2.3 Address mode wrap

This addressing mode is set using `hipAddressModeWrap`.

Wrap mode addressing is only available for normalized texture coordinates. In this addressing mode, the fractional part of the index is used:

`tex(frac(x))`

This creates a repeating image effect.

The following image shows the texture on a 4x4 pixel quad, indexed in the [0 to 3] range. The out-of-bounds values are repeating the original texture.

The purple lines are not part of the texture. They only denote the edge, where the addressing begins.

10.2.3.6.2.4 Address mode mirror

This addressing mode is set using `hipAddressModeMirror`.

Similar to the wrap mode the mirror mode is only available for normalized texture coordinates and also creates a repeating image, but mirroring the neighboring instances.

The formula is the following:

`tex(frac(x))`, if `floor(x)` is even,

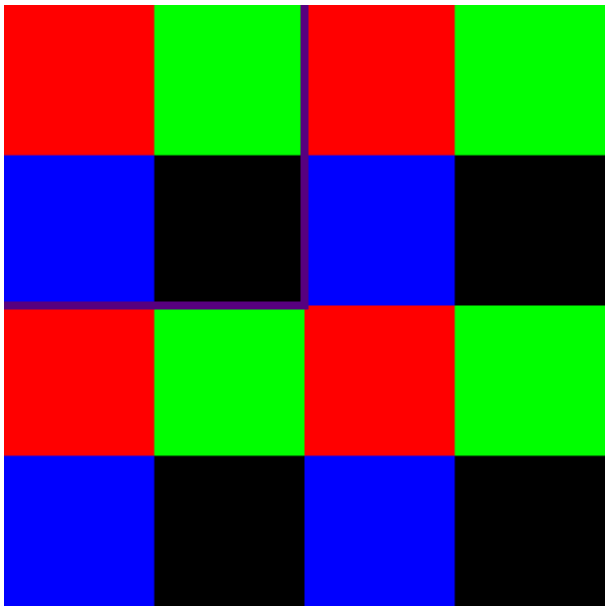


Fig. 6: Texture with wrap addressing.

`tex(1 - frac(x))`, if `floor(x)` is odd.

The following image shows the texture on a 4x4 pixel quad, indexed in The [0 to 3] range. The out-of-bounds values are repeating the original texture, but mirrored.

The purple lines are not part of the texture. They only denote the edge, where the addressing begins.

10.2.4 Coherence control

Memory coherence describes how memory of a specific part of the system is visible to the other parts of the system. For example, how GPU memory is visible to the CPU and vice versa. In HIP, host and device memory can be allocated with two different types of coherence:

- **Coarse-grained coherence:** The memory is considered up-to-date only after synchronization performed using `hipDeviceSynchronize()`, `hipStreamSynchronize()`, or any blocking operation that acts on the null stream such as `hipMemcpy()`. To avoid the cache from being accessed by a part of the system while simultaneously being written by another, the memory is made visible only after the caches have been flushed.
- **Fine-grained coherence:** The memory is coherent even while being modified by a part of the system. Fine-grained coherence ensures that up-to-date data is visible to others regardless of kernel boundaries. This can be useful if both host and device operate on the same data.

> **Note**
>
> To achieve fine-grained coherence, many AMD GPUs use a limited cache policy, such as leaving these allocations uncached by the GPU or making them read-only.

Mi200 accelerator's hardware based floating point instructions work on coarse-grained memory regions. Coarse-grained coherence is typically useful in reducing host-device interconnect communication.

To check the availability of fine- and coarse-grained memory pools, use `rocminfo`:

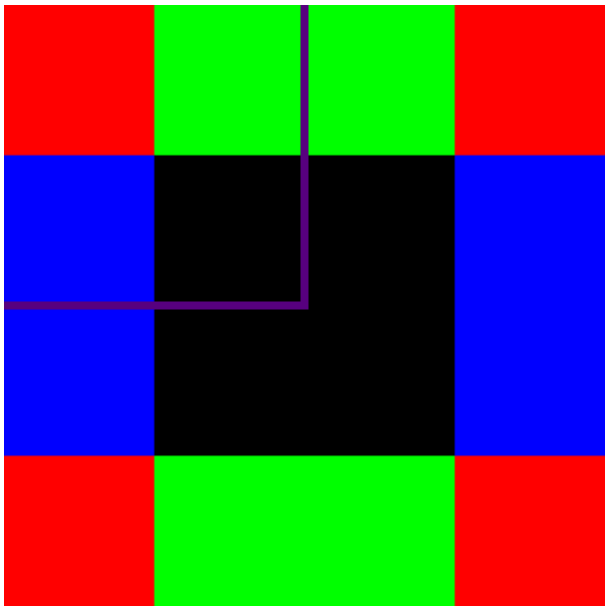


Fig. 7: Texture with mirror addressing

```
$ rocminfo
...
*******
Agent 1
*******
Name:                    AMD EPYC 7742 64-Core Processor
...
Pool Info:
Pool 1
Segment:                 GLOBAL; FLAGS: FINE GRAINED
...
Pool 3
Segment:                 GLOBAL; FLAGS: COARSE GRAINED
...
*******
Agent 9
*******
Name:                    gfx90a
...
Pool Info:
Pool 1
Segment:                 GLOBAL; FLAGS: COARSE GRAINED
...
```

The APIs, flags and respective memory coherence control are listed in the following table:

Table 2: Memory coherence control

API	Flag	hipMemAdvise() call with argument	Coherence
hipHostMalloc [1]	hipHostMallocDefault		Fine-grained
hipHostMalloc [1]	hipHostMallocNonCoherent		Coarse-grained
hipExtMallocWithFlags	hipDeviceMallocDefault		Coarse-grained
hipExtMallocWithFlags	hipDeviceMallocFinegrained		Fine-grained
hipMallocManaged			Fine-grained
hipMallocManaged		hipMemAdviseSetCoarseGrain	Coarse-grained
malloc			Fine-grained
malloc		hipMemAdviseSetCoarseGrain	Coarse-grained

[1] The hipHostMalloc() memory allocation coherence mode can be affected by the HIP_HOST_COHERENT environment variable, if the hipHostMallocCoherent, hipHostMallocNonCoherent, and hipHostMallocMapped are unset. If neither these flags nor the HIP_HOST_COHERENT environment variable is set, or set as 0, the host memory allocation is coarse-grained.

Note

- When hipHostMallocMapped flag is set, the allocated host memory is fine-grained and the hipHostMallocNonCoherent flag is ignored.
- Setting both the hipHostMallocCoherent and hipHostMallocNonCoherent flags leads to an illegal state.

10.2.4.1 Visibility of synchronization functions

The fine-grained coherence memory is visible at the synchronization points, however the visibility of coarse-grained memory depends on the synchronization function used. The effect and visibility of various synchronization functions on fine- and coarse-grained memory types are listed here:

Table 3: HIP synchronize functions effect and visibility

HIP API	hipStreamSynchronize()	hipDeviceSynchronize()	hipEventSynchronize()	hipStreamWaitEvent()
Synchronization effect	Host waits for all commands in the specified stream to complete	Host waits for all commands in all streams on the specified device to complete	Host waits for the specified event to complete	Stream waits for the specified event to complete
Fence	System-scope release	System-scope release	System-scope release	None
Fine-grained host memory visibility	Yes	Yes	Yes	Yes
Coarse-grained host memory visibility	Yes	Yes	Depends on the used event.	No

You can control the release scope for hipEvents. By default, the GPU performs a device-scope acquire and release

operation with each recorded event. This makes the host and device memory visible to other commands executing on the same device.

`hipEventCreateWithFlags()`: You can specify a stronger system-level fence by creating the event with `hipEventCreateWithFlags`:

- `hipEventReleaseToSystem`: Performs a system-scope release operation when the event is recorded. This makes both fine-grained and coarse-grained host memory visible to other agents in the system, which might also involve heavyweight operations such as cache flushing. Fine-grained memory typically uses lighter-weight in-kernel synchronization mechanisms such as an atomic operation and thus doesn't need to use `hipEventReleaseToSystem`.
- `hipEventDisableTiming`: Events created with this flag don't record profiling data, which significantly improves synchronization performance.

10.2.5 Unified memory management

This document covers unified memory management in HIP, which encompasses several approaches that provide a single address space accessible from both CPU and GPU. **Unified memory** refers to the overall architectural concept of this shared address space, while **managed memory** is one specific implementation that provides automatic page migration between devices. Other unified memory allocators like `hipMalloc()` and `hipHostMalloc()` provide different access patterns within the same unified address space concept.

In conventional architectures CPUs and attached devices have their own memory space and dedicated physical memory backing it up, e.g. normal RAM for CPUs and VRAM on GPUs. This way each device can have physical memory optimized for its use case. GPUs usually have specialized memory whose bandwidth is a magnitude higher than the RAM attached to CPUs.

While providing exceptional performance, this setup typically requires explicit memory management, as memory needs to be allocated, copied and freed on the used devices and on the host. Additionally, this makes using more than the physically available memory on the devices complicated.

Modern GPUs circumvent the problem of having to explicitly manage the memory, while still keeping the benefits of the dedicated physical memories, by supporting the concept of unified memory. This enables the CPU and the GPUs in the system to access host and other GPUs' memory without explicit memory management.

10.2.5.1 Unified memory

Unified Memory is a single memory address space accessible from any processor within a system. This setup simplifies memory management and enables applications to allocate data that can be read or written on both CPUs and GPUs without explicitly copying it to the specific CPU or GPU. The Unified memory model is shown in the following figure.

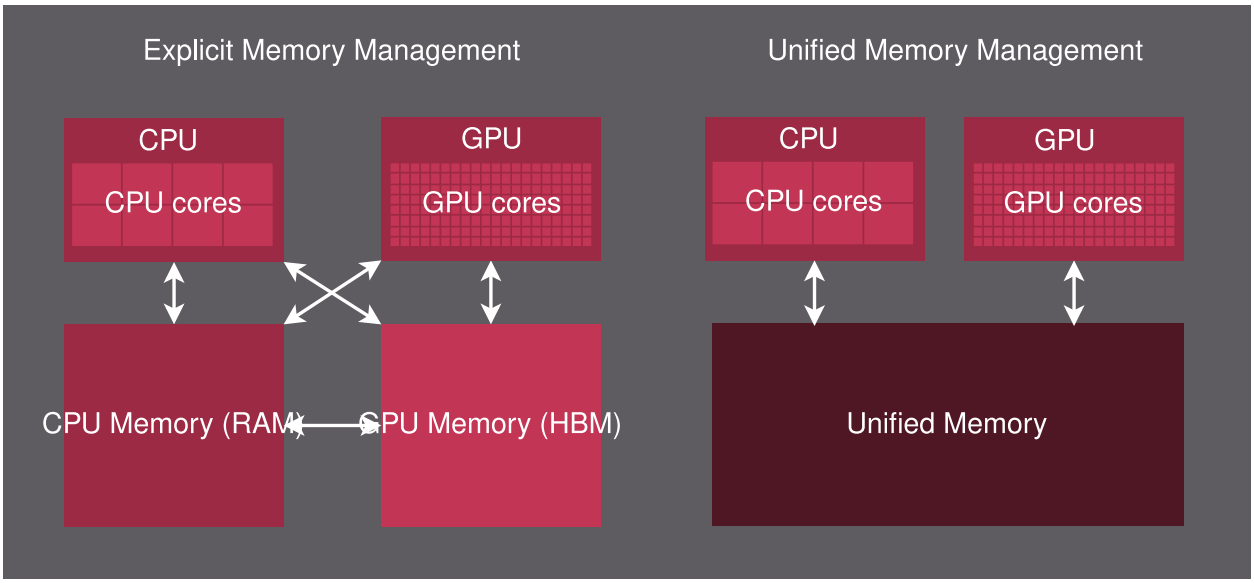
Unified memory enables the access to memory located on other devices via several methods, depending on whether hardware support is available or has to be managed by the driver. CPUs can access memory allocated via `hipMalloc()`, providing bidirectional memory accessibility within the unified address space.

10.2.5.2 Managed memory

Managed Memory is an extension of the unified memory architecture in which HIP monitors memory access and intelligently migrates data between device and system memories, thereby improving performance and resource efficiency.

When a kernel on the device tries to access a managed memory address that is not in its local device memory, a page-fault is triggered. The GPU then in turn requests the page from the host or other device on which the memory is located. The page is unmapped from the source, sent to the device and mapped to the device's memory. The requested memory is then available locally to the processes running on the device, which improves performance as local memory access outperforms remote memory access.

Managed memory also expands the memory capacity available to a GPU kernel. When migrating memory into the device on page-fault, if the device's memory is already at capacity, a page is unmapped from the device's memory first and sent



and mapped to host memory. This enables more memory to be allocated and used for a GPU than the GPU itself has physically available. This level of support can be very beneficial, for example, for sparse accesses to an array that is not often used on the device.

10.2.5.2.1 System requirements for managed memory

Some AMD GPUs do not support page-faults, and thus do not support on-demand page-fault driven migration. On these architectures, if the programmer prefers all GPU memory accesses to be local, all pages have to migrated before the kernel is dispatched, as the driver cannot know beforehand which parts of a dataset are going to be accessed. This can lead to significant delays on the first run of a kernel, and, in the example of a sparsely accessed array, can also lead to copying more memory than is actually accessed by the kernel.

Note that on systems which do not support page-faults, managed memory APIs are still accessible to the programmer, but managed memory operates in a degraded fashion due to the lack of demand-driven migration. Furthermore, on these systems it is still possible to use unified memory allocators that do not provide managed memory features; see *Memory allocation approaches in unified memory* for more details.

Managed memory is supported on Linux by all modern AMD GPUs from the Vega series onward, as shown in the following table. Managed memory can be explicitly allocated using `hipMallocManaged()` or marking variables with the `__managed__` attribute. For the latest GPUs, with a Linux kernel that supports Heterogeneous Memory Management (HMM), the normal system allocators (e.g., `new`, `malloc()`) can be used.

> **Note**
>
> To ensure the proper functioning of managed memory on supported GPUs, it is **essential** to set the environment variable `HSA_XNACK=1` and use a GPU kernel mode driver that supports HMM. Without this configuration, access-driven memory migration will be disabled, and the behavior will be similar to that of systems without HMM support.

Table 4: Managed Memory Support by GPU Architecture

Architecture	`hipMallocManaged()`, `__managed__`	`new`, `malloc()`, `allocate()`
CDNA4	✅	✅ [1]
CDNA3	✅	✅ [1]
CDNA2	✅	✅ [1]
CDNA1	✅	❌
RDNA1	✅	❌
GCN5	✅	❌

✅: **Supported**

❌: **Unsupported**

[1] Works only with `HSA_XNACK=1` and kernels with HMM support. First GPU access causes recoverable page-fault.

10.2.5.3 Memory allocation approaches in unified memory

While managed memory provides automatic migration, unified memory encompasses several allocation methods, each with different access patterns and migration behaviors. The following section covers all available unified memory allocation approaches, including but not limited to managed memory APIs.

Support for the different unified memory allocators depends on the GPU architecture and on the system. For more information, see *System requirements for managed memory* and *Checking unified memory support*.

- **HIP allocated managed memory and variables**

  `hipMallocManaged()` is a dynamic memory allocator available on all GPUs with unified memory support. For more details, visit Managed memory.

  The `__managed__` declaration specifier, which serves as its counterpart, can be utilized for static allocation.

- **System allocated unified memory**

  Starting with CDNA2, the `new`, `malloc()`, and `allocate()` (Fortran) system allocators allow you to reserve unified memory. The system allocator is more versatile and offers an easy transition for code written for CPUs to HIP code as the same system allocation API is used. Memory allocated by these allocators can be registered to be accessible on device using `hipHostRegister()`.

- **HIP allocated non-managed memory**

  `hipMalloc()` and `hipHostMalloc()` are dynamic memory allocators available on all GPUs with unified memory support. Memory allocated by these allocators is not migrated between device and host memory.

The table below illustrates the expected behavior of managed and unified memory functions on ROCm and CUDA, both with and without HMM support.

ROCm allocation behaviour

Table 5: Comparison of expected behavior of managed and unified memory functions in ROCm

call	Allocation origin without HMM or HSA_XNACK=0	Access outside the origin without HMM or HSA_XNACK=0	Allocation origin with HMM and HSA_XNACK=1	Access outside the origin with HMM and HSA_XNACK=1
new, malloc(), allocate()	host	not accessible on device	first touch	page-fault migration
hipMalloc()	device	zero copy [zc]	device	zero copy [zc]
hipMallocManaged(), __managed__	pinned host	zero copy [zc]	first touch	page-fault migration
hipHostRegister()	pinned host	zero copy [zc]	pinned host	zero copy [zc]
hipHostMalloc()	pinned host	zero copy [zc]	pinned host	zero copy [zc]

CUDA allocation behaviour

Table 6: Comparison of expected behavior of managed and unified memory functions in CUDA

call	Allocation origin without HMM	Access outside the origin without HMM	Allocation origin with HMM	Access outside the origin with HMM
new, malloc()	host	not accessible on device	first touch	page-fault migration
cudaMalloc()	device	not accessible on host	device	page-fault migration
cudaMallocManaged(), __managed__	host	page-fault migration	first touch	page-fault migration
cudaHostRegister()	host	page-fault migration	host	page-fault migration
cudaMallocHost()	pinned host	zero copy [zc]	pinned host	zero copy [zc]

10.2.5.3.1 Checking unified memory support

The following device attributes can offer information about which *Memory allocation approaches in unified memory* are supported. The attribute value is 1 if the functionality is supported, and 0 if it is not supported.

Table 7: Device attributes for unified memory management

Attribute	Description
hipDeviceAttributeManagedMemory	Device supports allocating managed memory on this system
hipDeviceAttributePageableMemoryAccess	Device supports coherently accessing pageable memory without calling hipHostRegister() on it.
hipDeviceAttributeConcurrentManagedAccess	Full unified memory support. Device can coherently access managed memory concurrently with the CPU

For details on how to get the attributes of a specific device see hipDeviceGetAttribute().

10.2.5.3.2 Example for unified memory management

The following example shows how to use unified memory with hipMallocManaged() for dynamic allocation, the __managed__ attribute for static allocation and the standard new allocation. For comparison, the explicit memory management example is presented in the last tab.

hipMallocManaged()

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

#include <iostream>

#define HIP_CHECK(expression)                \
{                                            \
    const hipError_t err = expression;       \
    if(err != hipSuccess)                    \
    {                                        \
        std::cerr << "HIP error: "           \
            << hipGetErrorString(err)        \
            << " at " << __LINE__ << "\n";   \
    }                                        \
}

// Addition of two values.
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}

int main()
{
    int *a, *b, *c;

    // Allocate memory for a, b and c that is accessible to both device and host␣
↪codes.
    HIP_CHECK(hipMallocManaged(&a, sizeof(*a)));
    HIP_CHECK(hipMallocManaged(&b, sizeof(*b)));
    HIP_CHECK(hipMallocManaged(&c, sizeof(*c)));

    // Setup input values.
    *a = 1;
    *b = 2;

    // Launch add() kernel on GPU.
    add<<<1, 1>>>(a, b, c);

    // Wait for GPU to finish before accessing on host.
    HIP_CHECK(hipDeviceSynchronize());

    // Print the result.
    std::cout << *a << " + " << *b << " = " << *c << std::endl;

    // Cleanup allocated memory.
    HIP_CHECK(hipFree(a));
    HIP_CHECK(hipFree(b));
    HIP_CHECK(hipFree(c));

    return 0;
}
```

__managed__

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

#include <iostream>

#define HIP_CHECK(expression)                \
{                                            \
    const hipError_t err = expression;       \
    if(err != hipSuccess)                    \
    {                                        \
        std::cerr << "HIP error: "           \
            << hipGetErrorString(err)        \
            << " at " << __LINE__ << "\n";   \
    }                                        \
}

// Addition of two values.
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}

// Declare a, b and c as static variables.
__managed__ int a, b, c;

int main()
{
    // Setup input values.
    a = 1;
    b = 2;

    // Launch add() kernel on GPU.
    add<<<1, 1>>>(&a, &b, &c);

    // Wait for GPU to finish before accessing on host.
    HIP_CHECK(hipDeviceSynchronize());

    // Print the result.
    std::cout << a << " + " << b << " = " << c << std::endl;

    return 0;
}
```

new

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

#include <iostream>

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

(continues on next page)

(continued from previous page)

```
    if(err != hipSuccess)                    \
    {                                        \
        std::cerr << "HIP error: "           \
            << hipGetErrorString(err)        \
            << " at " << __LINE__ << "\n";   \
    }                                        \
}

// Addition of two values.
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}

// This example requires HMM support and the environment variable HSA_XNACK needs to␣
↪be set to 1
int main()
{
    // Allocate memory for a, b, and c.
    int *a = new int[1];
    int *b = new int[1];
    int *c = new int[1];

    // Setup input values.
    *a = 1;
    *b = 2;

    // Launch add() kernel on GPU.
    add<<<1, 1>>>(a, b, c);

    // Wait for GPU to finish before accessing on host.
    HIP_CHECK(hipDeviceSynchronize());

    // Print the result.
    std::cout << *a << " + " << *b << " = " << *c << std::endl;

    // Cleanup allocated memory.
    delete[] c;
    delete[] b;
    delete[] a;

    return 0;
}
```

Explicit Memory Management

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

#include <iostream>

#define HIP_CHECK(expression)                \
```

(continues on next page)

(continued from previous page)

```
{                                                   \
    const hipError_t err = expression;              \
    if(err != hipSuccess)                           \
    {                                               \
        std::cerr << "HIP error: "                  \
            << hipGetErrorString(err)               \
            << " at " << __LINE__ << "\n";          \
    }                                               \
}

// Addition of two values.
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}

int main()
{
    int a, b, c;
    int *d_a, *d_b, *d_c;

    // Setup input values.
    a = 1;
    b = 2;

    // Allocate device copies of a, b and c.
    HIP_CHECK(hipMalloc(&d_a, sizeof(*d_a)));
    HIP_CHECK(hipMalloc(&d_b, sizeof(*d_b)));
    HIP_CHECK(hipMalloc(&d_c, sizeof(*d_c)));

    // Copy input values to device.
    HIP_CHECK(hipMemcpy(d_a, &a, sizeof(*d_a), hipMemcpyHostToDevice));
    HIP_CHECK(hipMemcpy(d_b, &b, sizeof(*d_b), hipMemcpyHostToDevice));

    // Launch add() kernel on GPU.
    add<<<1, 1>>>(d_a, d_b, d_c);

    // Copy the result back to the host.
    HIP_CHECK(hipMemcpy(&c, d_c, sizeof(*d_c), hipMemcpyDeviceToHost));

    // Cleanup allocated memory.
    HIP_CHECK(hipFree(d_a));
    HIP_CHECK(hipFree(d_b));
    HIP_CHECK(hipFree(d_c));

    // Prints the result.
    std::cout << a << " + " << b << " = " << c << std::endl;

    return 0;
}
```

10.2.5.4 Using unified memory

Unified memory can simplify the complexities of memory management in GPU computing, by not requiring explicit copies between the host and the devices. It can be particularly useful in use cases with sparse memory accesses from both the CPU and the GPU, as only the parts of the memory region that are actually accessed need to be transferred to the corresponding processor, not the whole memory region. This reduces the amount of memory sent over the PCIe bus or other interfaces.

In HIP, pinned memory allocations are coherent by default. Pinned memory is host memory mapped into the address space of all GPUs, meaning that the pointer can be used on both host and device. Additionally, using pinned memory instead of pageable memory on the host can improve bandwidth for transfers between the host and the GPUs.

While unified memory can provide numerous benefits, it's important to be aware of the potential performance overhead associated with unified memory. You must thoroughly test and profile your code to ensure it's the most suitable choice for your use case.

10.2.5.5 Performance optimizations for unified memory

There are several ways, in which the developer can guide the runtime to reduce copies between devices, in order to improve performance.

With `numactl --membind` bindings, developers can control where physical allocation occurs by restricting memory allocation to specific NUMA nodes. This approach can reduce or eliminate the need for explicit data prefetching since memory is allocated in the desired location from the start.

10.2.5.5.1 Data prefetching

> **Warning**
>
> Data prefetching is not always an optimization and can slow down execution, as the API takes time to execute. If the memory is already in the right place, prefetching will waste time. Users should profile their code to verify whether prefetching is beneficial for their specific use case.

When prefetching is beneficial, developers can consider setting different default locations for different devices and using prefetch between them, which can help eliminate IPC communication overhead when memory moves between devices.

Data prefetching is a technique used to improve the performance of your application by moving data to the desired device before it's actually needed. `hipCpuDeviceId` is a special constant to specify the CPU as target.

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

#include <iostream>

#define HIP_CHECK(expression)               \
{                                           \
    const hipError_t err = expression;      \
    if(err != hipSuccess)                   \
    {                                       \
        std::cerr << "HIP error: "          \
            << hipGetErrorString(err)       \
            << " at " << __LINE__ << "\n";  \
    }                                       \
}
```

(continues on next page)

(continued from previous page)

```
// Addition of two values.
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}

int main()
{
    int *a, *b, *c;
    int deviceId;
    HIP_CHECK(hipGetDevice(&deviceId)); // Get the current device ID

    // Allocate memory for a, b and c that is accessible to both device and host␣
↪codes.
    HIP_CHECK(hipMallocManaged(&a, sizeof(*a)));
    HIP_CHECK(hipMallocManaged(&b, sizeof(*b)));
    HIP_CHECK(hipMallocManaged(&c, sizeof(*c)));

    // Setup input values.
    *a = 1;
    *b = 2;

    // Prefetch the data to the GPU device.
    HIP_CHECK(hipMemPrefetchAsync(a, sizeof(*a), deviceId, 0));
    HIP_CHECK(hipMemPrefetchAsync(b, sizeof(*b), deviceId, 0));
    HIP_CHECK(hipMemPrefetchAsync(c, sizeof(*c), deviceId, 0));

    // Launch add() kernel on GPU.
    add<<<1, 1>>>(a, b, c);

    // Prefetch the result back to the CPU.
    HIP_CHECK(hipMemPrefetchAsync(c, sizeof(*c), hipCpuDeviceId, 0));

    // Wait for the prefetch operations to complete.
    HIP_CHECK(hipDeviceSynchronize());

    // Prints the result.
    std::cout << *a << " + " << *b << " = " << *c << std::endl;

    // Cleanup allocated memory.
    HIP_CHECK(hipFree(a));
    HIP_CHECK(hipFree(b));
    HIP_CHECK(hipFree(c));

    return 0;
}
```

10.2.5.5.2 Memory advice

Unified memory runtime hints can be set with `hipMemAdvise()` to help improve the performance of your code if you know the memory usage pattern. There are several different types of hints as specified in the enum `hipMemoryAdvise`, for example, whether a certain device mostly reads the memory region, where it should ideally be located, and even

whether that specific memory region is accessed by a specific device.

For the best performance, profile your application to optimize the utilization of HIP runtime hints.

The effectiveness of `hipMemAdvise()` comes from its ability to inform the runtime of the developer's intentions regarding memory usage. When the runtime has knowledge of the expected memory access patterns, it can make better decisions about data placement, leading to less transfers via the interconnect and thereby reduced latency and bandwidth requirements. However, the actual impact on performance can vary based on the specific use case and the system.

The following is the updated version of the example above with memory advice instead of prefetching.

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

#define HIP_CHECK(expression)              \
{                                          \
    const hipError_t err = expression;     \
    if(err != hipSuccess)                  \
    {                                      \
        std::cerr << "HIP error: "         \
            << hipGetErrorString(err)      \
            << " at " << __LINE__ << "\n"; \
    }                                      \
}

// Addition of two values.
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}

int main()
{
    int deviceId;
    HIP_CHECK(hipGetDevice(&deviceId));
    int *a, *b, *c;

    // Allocate memory for a, b, and c accessible to both device and host codes.
    HIP_CHECK(hipMallocManaged(&a, sizeof(*a)));
    HIP_CHECK(hipMallocManaged(&b, sizeof(*b)));
    HIP_CHECK(hipMallocManaged(&c, sizeof(*c)));

    // Set memory advice for a and b to be read, located on and accessed by the GPU.
    HIP_CHECK(hipMemAdvise(a, sizeof(*a), hipMemAdviseSetPreferredLocation,␣
↪deviceId));
    HIP_CHECK(hipMemAdvise(a, sizeof(*a), hipMemAdviseSetAccessedBy, deviceId));
    HIP_CHECK(hipMemAdvise(a, sizeof(*a), hipMemAdviseSetReadMostly, deviceId));

    HIP_CHECK(hipMemAdvise(b, sizeof(*b), hipMemAdviseSetPreferredLocation,␣
↪deviceId));
    HIP_CHECK(hipMemAdvise(b, sizeof(*b), hipMemAdviseSetAccessedBy, deviceId));
    HIP_CHECK(hipMemAdvise(b, sizeof(*b), hipMemAdviseSetReadMostly, deviceId));

    // Set memory advice for c to be read, located on and accessed by the CPU.
    HIP_CHECK(hipMemAdvise(c, sizeof(*c), hipMemAdviseSetPreferredLocation,␣
```

(continues on next page)

(continued from previous page)

```
↪hipCpuDeviceId));
    HIP_CHECK(hipMemAdvise(c, sizeof(*c), hipMemAdviseSetAccessedBy, hipCpuDeviceId));
    HIP_CHECK(hipMemAdvise(c, sizeof(*c), hipMemAdviseSetReadMostly, hipCpuDeviceId));

    // Setup input values.
    *a = 1;
    *b = 2;

    // Launch add() kernel on GPU.
    add<<<1, 1>>>(a, b, c);

    // Wait for GPU to finish before accessing on host.
    HIP_CHECK(hipDeviceSynchronize());

    // Prints the result.
    std::cout << *a << " + " << *b << " = " << *c << std::endl;

    // Cleanup allocated memory.
    HIP_CHECK(hipFree(a));
    HIP_CHECK(hipFree(b));
    HIP_CHECK(hipFree(c));

    return 0;
}
```

10.2.5.5.3 Memory range attributes

`hipMemRangeGetAttribute()` allows you to query attributes of a given memory range. The attributes are given in `hipMemRangeAttribute`.

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

#include <cstddef>
#include <iostream>

#define HIP_CHECK(expression)                \
{                                            \
    const hipError_t err = expression;       \
    if(err != hipSuccess)                    \
    {                                        \
        std::cerr << "HIP error: "           \
            << hipGetErrorString(err)        \
            << " at " << __LINE__ << "\n";   \
    }                                        \
}

// Addition of two values.
__global__ void add(int *a, int *b, int *c)
{
    *c = *a + *b;
}
```

(continues on next page)

(continued from previous page)

```
int main()
{
    int *a, *b, *c;
    unsigned int attributeValue;
    constexpr std::size_t attributeSize = sizeof(attributeValue);

    int deviceId;
    HIP_CHECK(hipGetDevice(&deviceId));

    // Allocate memory for a, b and c that is accessible to both device and host␣
↪codes.
    HIP_CHECK(hipMallocManaged(&a, sizeof(*a)));
    HIP_CHECK(hipMallocManaged(&b, sizeof(*b)));
    HIP_CHECK(hipMallocManaged(&c, sizeof(*c)));

    // Setup input values.
    *a = 1;
    *b = 2;

    HIP_CHECK(hipMemAdvise(a, sizeof(*a), hipMemAdviseSetReadMostly, deviceId));

    // Launch add() kernel on GPU.
    add<<<1, 1>>>(a, b, c);

    // Wait for GPU to finish before accessing on host.
    HIP_CHECK(hipDeviceSynchronize());

    // Query an attribute of the memory range.
    HIP_CHECK(hipMemRangeGetAttribute(&attributeValue,
                              attributeSize,
                              hipMemRangeAttributeReadMostly,
                              a,
                              sizeof(*a)));

    // Prints the result.
    std::cout << *a << " + " << *b << " = " << *c << std::endl;
    std::cout << "The array a is" << (attributeValue == 1 ? "" : " NOT") << " set to␣
↪hipMemRangeAttributeReadMostly" << std::endl;

    // Cleanup allocated memory.
    HIP_CHECK(hipFree(a));
    HIP_CHECK(hipFree(b));
    HIP_CHECK(hipFree(c));

    return 0;
}
```

10.2.5.5.4 Asynchronously attach memory to a stream

The `hipStreamAttachMemAsync()` function attaches memory to a stream, which can reduce the amount of memory transferred, when managed memory is used. When the memory is attached to a stream using this function, it only gets transferred between devices, when a kernel that is launched on this stream needs access to the memory.

10.2.6 Virtual memory management

Memory management is important when creating high-performance applications in the HIP ecosystem. Both allocating and copying memory can result in bottlenecks, which can significantly impact performance.

Global memory allocation in HIP uses the C language style allocation function. This works fine for simple cases but can cause problems if your memory needs change. If you need to increase the size of your memory, you must allocate a second larger buffer and copy the data to it before you can free the original buffer. This increases overall memory usage and causes unnecessary `memcpy` calls. Another solution is to allocate a larger buffer than you initially need. However, this isn't an efficient way to handle resources and doesn't solve the issue of reallocation when the extra buffer runs out.

Virtual memory management solves these memory management problems. It helps to reduce memory usage and unnecessary `memcpy` calls.

HIP virtual memory management is built on top of HSA, which provides low-level access to AMD GPU memory. For more details on the underlying HSA runtime, see ROCr documentation

10.2.6.1 Memory allocation

Standard memory allocation uses the `hipMalloc()` function to allocate a block of memory on the device. However, when using virtual memory, this process is separated into multiple steps using the `hipMemCreate()`, `hipMemAddressReserve()`, `hipMemMap()`, and `hipMemSetAccess()` functions. This guide explains what these functions do and how you can use them for virtual memory management.

10.2.6.1.1 Virtual memory management support

The first step is to check if the targeted device or GPU supports virtual memory management. Use the `hipDeviceGetAttribute()` function to get the `hipDeviceAttributeVirtualMemoryManagementSupported` attribute for a specific GPU, as shown in the following example.

```
int vmm = 0, currentDev = 0;
hipDeviceGetAttribute(
    &vmm, hipDeviceAttributeVirtualMemoryManagementSupported, currentDev
);

if (vmm == 0) {
    std::cout << "GPU " << currentDev << " doesn't support virtual memory management.
↪" << std::endl;
} else {
    std::cout << "GPU " << currentDev << " support virtual memory management." <<␣
↪std::endl;
}
```

10.2.6.1.2 Allocate physical memory

The next step is to allocate the physical memory using the `hipMemCreate()` function. This function accepts the size of the buffer, an `unsigned long long` variable for the flags, and a `hipMemAllocationProp` variable. `hipMemAllocationProp` contains the properties of the memory to be allocated, such as where the memory is physically located and what kind of shareable handles are available. If the allocation is successful, the function returns a value of `hipSuccess`, with `hipMemGenericAllocationHandle_t` representing a valid physical memory allocation.

The allocated memory must be aligned with the appropriate granularity. The granularity value can be queried with `hipMemGetAllocationGranularity()`, and its value depends on the target device hardware and the type of memory allocation. If the allocation size is not aligned, meaning it is not cleanly divisible by the minimum granularity value, `hipMemCreate()` will return an out-of-memory error.

```
size_t granularity = 0;
hipMemGenericAllocationHandle_t allocHandle;
hipMemAllocationProp prop = {};
// The pinned allocation type cannot be migrated from its current location
// while the application is actively using it.
prop.type = hipMemAllocationTypePinned;
// Set the location type to device, currently there are no other valid option.
prop.location.type = hipMemLocationTypeDevice;
// Set the device id, where the memory will be allocated.
prop.location.id = currentDev;
hipMemGetAllocationGranularity(&granularity, &prop,␣
↪hipMemAllocationGranularityMinimum);
padded_size = ROUND_UP(size, granularity);
hipMemCreate(&allocHandle, padded_size, &prop, 0);
```

10.2.6.1.3 Reserve virtual address range

After you have acquired an allocation of physical memory, you must map it to a virtual address before you can use it. Mapping means the physical memory allocation is available from the virtual address range it is mapped to. To reserve a virtual memory range, use the `hipMemAddressReserve()` function. The size of the virtual memory must match the amount of physical memory previously allocated. You can then map the physical memory allocation to the newly-acquired virtual memory address range using the `hipMemMap()` function.

```
hipMemAddressReserve(&ptr, padded_size, 0, 0, 0);
hipMemMap(ptr, padded_size, 0, allocHandle, 0);
```

10.2.6.1.4 Set memory access

Finally, use the `hipMemSetAccess()` function to enable memory access. It accepts the pointer to the virtual memory, the size, and a `hipMemAccessDesc` descriptor as parameters. In a multi-GPU environment, you can map the device memory of one GPU to another. This feature also works with the traditional memory management system, but isn't as scalable as with virtual memory. When memory is allocated with `hipMalloc()`, `hipDeviceEnablePeerAccess()` is used to enable peer access. This function enables access between two devices, but it means that every call to `hipMalloc()` takes more time to perform the checks and the mapping between the devices. When using virtual memory management, peer access is enabled by `hipMemSetAccess()`, which provides a finer level of control over what is shared. This has no performance impact on memory allocation and gives you more control over what memory buffers are shared with which devices.

```
hipMemAccessDesc accessDesc = {};
accessDesc.location.type = hipMemLocationTypeDevice;
accessDesc.location.id = currentDev;
accessDesc.flags = hipMemAccessFlagsProtReadwrite;
hipMemSetAccess(ptr, padded_size, &accessDesc, 1);
```

At this point the memory is allocated, mapped, and ready for use. You can read and write to it, just like you would a C style memory allocation.

10.2.6.1.5 Dynamically increase allocation size

To increase the amount of pre-allocated memory, use `hipMemAddressReserve()`, which accepts the starting address, and the size of the reservation in bytes. This allows you to have a continuous virtual address space without worrying about the underlying physical allocation.

```
hipMemAddressReserve(&new_ptr, (new_size - padded_size), 0, ptr + padded_size, 0);
hipMemMap(new_ptr, (new_size - padded_size), 0, newAllocHandle, 0);
hipMemSetAccess(new_ptr, (new_size - padded_size), &accessDesc, 1);
```

The code sample above assumes that `hipMemAddressReserve()` was able to reserve the memory address at the specified location. However, this isn't guaranteed to be true, so you should validate that `new_ptr` points to a specific virtual address before using it.

10.2.6.1.6 Free virtual memory

To free the memory allocated in this manner, use the corresponding free functions. To unmap the memory, use `hipMemUnmap()`. To release the virtual address range, use `hipMemAddressFree()`. Finally, to release the physical memory, use `hipMemRelease()`. A side effect of these functions is the lack of synchronization when memory is released. If you call `hipFree()` when you have multiple streams running in parallel, it synchronizes the device. This causes worse resource usage and performance.

```
hipMemUnmap(ptr, size);
hipMemRelease(allocHandle);
hipMemAddressFree(ptr, size);
```

10.2.6.2 Example code

The virtual memory management example follows these steps:

1. Check virtual memory management *support*: The `hipDeviceGetAttribute()` function is used to check the virtual memory management support of the GPU with ID 0.
2. Physical memory *allocation*: Physical memory is allocated using `hipMemCreate()` with pinned memory on the device.
3. Virtual memory *reservation*: Virtual address range is reserved using `hipMemAddressReserve()`.
4. Mapping virtual address to physical memory: The physical memory is mapped to a virtual address (`virtualPointer`) using `hipMemMap()`.
5. Memory *access permissions*: Permission is set for pointer to allow read and write access using `hipMemSetAccess()`.
6. Memory operation: Data is written to the memory via `virtualPointer`.
7. Launch kernels: The `zeroAddr` and `fillAddr` kernels are launched using the virtual memory pointer.
8. *Cleanup*: The mappings, physical memory, and virtual address are released at the end to avoid memory leaks.

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

#define ROUND_UP(SIZE,GRANULARITY) ((1 + SIZE / GRANULARITY) * GRANULARITY)

#define HIP_CHECK(expression)                \
{                                            \
    const hipError_t err = expression;       \
    if(err != hipSuccess){                   \
        std::cerr << "HIP error: "           \
            << hipGetErrorString(err)        \
            << " at " << __LINE__ << "\n";   \
    }                                        \
```

(continues on next page)

(continued from previous page)

```
}

__global__ void zeroAddr(int* pointer) {
    *pointer = 0;
}

__global__ void fillAddr(int* pointer) {
    *pointer = 42;
}


int main() {

    int currentDev = 0;

    // Step 1: Check virtual memory management support on device 0
    int vmm = 0;
    HIP_CHECK(
        hipDeviceGetAttribute(
            &vmm, hipDeviceAttributeVirtualMemoryManagementSupported, currentDev
        )
    );

    std::cout << "Virtual memory management support value: " << vmm << std::endl;

    if (vmm == 0) {
        std::cout << "GPU 0 doesn't support virtual memory management.";
        return 0;
    }

    // Size of memory to allocate
    size_t size = 4 * 1024;

    // Step 2: Allocate physical memory
    hipMemGenericAllocationHandle_t allocHandle;
    hipMemAllocationProp prop = {};
    prop.type = hipMemAllocationTypePinned;
    prop.location.type = hipMemLocationTypeDevice;
    prop.location.id = currentDev;
    size_t granularity = 0;
    HIP_CHECK(
        hipMemGetAllocationGranularity(
            &granularity,
            &prop,
            hipMemAllocationGranularityMinimum));
    size_t padded_size = ROUND_UP(size, granularity);
    HIP_CHECK(hipMemCreate(&allocHandle, padded_size * 2, &prop, 0));

    // Step 3: Reserve a virtual memory address range
    void* virtualPointer = nullptr;
    HIP_CHECK(hipMemAddressReserve(&virtualPointer, padded_size, granularity, nullptr,
↪ 0));
```

(continues on next page)

(continued from previous page)

```
    // Step 4: Map the physical memory to the virtual address range
    HIP_CHECK(hipMemMap(virtualPointer, padded_size, 0, allocHandle, 0));

    // Step 5: Set memory access permission for pointer
    hipMemAccessDesc accessDesc = {};
    accessDesc.location.type = hipMemLocationTypeDevice;
    accessDesc.location.id = currentDev;
    accessDesc.flags = hipMemAccessFlagsProtReadWrite;

    HIP_CHECK(hipMemSetAccess(virtualPointer, padded_size, &accessDesc, 1));

    // Step 6: Perform memory operation
    int value = 42;
    HIP_CHECK(hipMemcpy(virtualPointer, &value, sizeof(int), hipMemcpyHostToDevice));

    int result = 1;
    HIP_CHECK(hipMemcpy(&result, virtualPointer, sizeof(int), hipMemcpyDeviceToHost));
    if( result == 42) {
        std::cout << "Success. Value: " << result << std::endl;
    } else {
        std::cout << "Failure. Value: " << result << std::endl;
    }

    // Step 7: Launch kernels
    // Launch zeroAddr kernel
    zeroAddr<<<1, 1>>>((int*)virtualPointer);
    HIP_CHECK(hipDeviceSynchronize());

    // Check zeroAddr kernel result
    result = 1;
    HIP_CHECK(hipMemcpy(&result, virtualPointer, sizeof(int), hipMemcpyDeviceToHost));
    if( result == 0) {
        std::cout << "Success. zeroAddr kernel: " << result << std::endl;
    } else {
        std::cout << "Failure. zeroAddr kernel: " << result << std::endl;
    }

    // Launch fillAddr kernel
    fillAddr<<<1, 1>>>((int*)virtualPointer);
    HIP_CHECK(hipDeviceSynchronize());

    // Check fillAddr kernel result
    result = 1;
    HIP_CHECK(hipMemcpy(&result, virtualPointer, sizeof(int), hipMemcpyDeviceToHost));
    if( result == 42) {
        std::cout << "Success. fillAddr kernel: " << result << std::endl;
    } else {
        std::cout << "Failure. fillAddr kernel: " << result << std::endl;
    }

    // Step 8: Cleanup
```

(continues on next page)

(continued from previous page)

```
    HIP_CHECK(hipMemUnmap(virtualPointer, padded_size));
    HIP_CHECK(hipMemRelease(allocHandle));
    HIP_CHECK(hipMemAddressFree(virtualPointer, padded_size));

    return 0;
}
```

10.2.6.3 Virtual aliases

Virtual aliases are multiple virtual memory addresses mapping to the same physical memory on the GPU. When this occurs, different threads, processes, or memory allocations to access shared physical memory through different virtual addresses on different devices.

Multiple virtual memory mappings can be created using multiple calls to `hipMemMap()` on the same memory allocation.

> **Note**
>
> RDNA cards may not produce correct results, if users access two different virtual addresses that map to the same physical address. In this case, the L1 data caches will be incoherent due to the virtual-to-physical aliasing. These GPUs will produce correct results if users access virtual-to-physical aliases using volatile pointers.
>
> NVIDIA GPUs require special fences to produce correct results when using virtual aliases.

In the following code block, the kernels input device pointers are virtual aliases of the same memory allocation:

```
__global__ void updateBoth(int* pointerA, int* pointerB) {
    // May produce incorrect results on RDNA and NVIDIA cards.
    *pointerA = 0;
    *pointerB = 42;
}

__global__ void updateBoth_v2(volatile int* pointerA, volatile int* pointerB) {
    // May produce incorrect results on NVIDIA cards.
    *pointerA = 0;
    *pointerB = 42;
}
```

10.2.7 Stream-ordered memory allocator

The Stream-ordered memory allocator (SOMA) is part of the HIP runtime API. SOMA provides an asynchronous memory allocation mechanism with stream-ordering semantics. You can use SOMA to allocate and free memory in stream order, which ensures that all asynchronous accesses occur between the stream executions of allocation and deallocation. Compliance with stream order prevents use-before-allocation or use-after-free errors, which helps to avoid an undefined behavior.

Advantages of SOMA:

- Efficient reuse: Enables efficient memory reuse across streams, which reduces unnecessary allocation overhead.
- Fine-grained control: Allows you to set attributes and control caching behavior for memory pools.
- Inter-process sharing: Enables secure sharing of allocations between processes.
- Optimizations: Allows driver to optimize based on its awareness of SOMA and other stream management APIs.

Disadvantages of SOMA:

- Temporal constraints: Requires you to adhere strictly to stream order to avoid errors.
- Complexity: Involves memory management in stream order, which can be intricate.
- Learning curve: Requires you to put additional efforts to understand and utilize SOMA effectively.

10.2.7.1 Using SOMA

You can allocate memory using `hipMallocAsync()` with stream-ordered semantics. This restricts the asynchronous access to the memory between the stream executions of the allocation and deallocation. Accessing memory if the compliant memory accesses won't overlap temporally. `hipFreeAsync()` frees memory from the pool with stream-ordered semantics.

Here is how to use stream-ordered memory allocation:

Stream-ordered memory allocation

```
#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);                       \
    }                                                  \
}

// Kernel to perform some computation on allocated memory.
__global__ void myKernel(int* data, std::size_t numElements)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < numElements)
    {
        data[tid] = tid * 2;
    }
}

int main()
{
    // Stream 0.
    constexpr hipStream_t streamId = 0;

    // Allocate memory with stream ordered semantics.
    constexpr std::size_t numElements = 1024;
```

(continues on next page)

(continued from previous page)

```
    int* devData;
    HIP_CHECK(hipMallocAsync(reinterpret_cast<void**>(&devData), numElements *␣
↪sizeof(*devData), streamId));

    // Launch the kernel to perform computation.
    dim3 blockSize(256);
    dim3 gridSize((numElements + blockSize.x - 1) / blockSize.x);
    myKernel<<<gridSize, blockSize>>>(devData, numElements);

    // Copy data back to host.
    int* hostData = new int[numElements];
    HIP_CHECK(hipMemcpy(hostData, devData, numElements * sizeof(*devData),␣
↪hipMemcpyDeviceToHost));

    // Print the array.
    for (std::size_t i = 0; i < numElements; ++i)
        std::cout << "Element " << i << ": " << hostData[i] << std::endl;

    // Free memory with stream ordered semantics.
    HIP_CHECK(hipFreeAsync(devData, streamId));
    delete[] hostData;

    // Synchronize to ensure completion.
    HIP_CHECK(hipDeviceSynchronize());

    return EXIT_SUCCESS;
}
```

Ordinary allocation

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

#include <cstddef>
#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);                       \
    }                                                  \
}

// Kernel to perform some computation on allocated memory.
__global__ void myKernel(int* data, std::size_t numElements)
{
```

(continues on next page)

(continued from previous page)

```
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < numElements)
    {
        data[tid] = tid * 2;
    }
}

int main()
{
    // Allocate memory.
    constexpr std::size_t numElements = 1024;
    int* devData;
    HIP_CHECK(hipMalloc(&devData, numElements * sizeof(*devData)));

    // Launch the kernel to perform computation.
    dim3 blockSize(256);
    dim3 gridSize((numElements + blockSize.x - 1) / blockSize.x);
    myKernel<<<gridSize, blockSize>>>(devData, numElements);

    // Copy data back to host.
    int* hostData = new int[numElements];
    HIP_CHECK(hipMemcpy(hostData, devData, numElements * sizeof(*devData),␣
↪hipMemcpyDeviceToHost));

    // Print the array.
    for (std::size_t i = 0; i < numElements; ++i)
        std::cout << "Element " << i << ": " << hostData[i] << std::endl;

    // Free memory.
    HIP_CHECK(hipFree(devData));
    delete[] hostData;

    // Synchronize to ensure completion.
    HIP_CHECK(hipDeviceSynchronize());

    return EXIT_SUCCESS;
}
```

For more details, see Stream ordered memory allocator.

10.2.7.2 Memory pools

Memory pools provide a way to manage memory with stream-ordered behavior while ensuring proper synchronization and avoiding memory access errors. Division of a single memory system into separate pools facilitates querying the access path properties for each partition. Memory pools are used for host memory, device memory, and unified memory.

10.2.7.2.1 Set pools

The `hipMallocAsync()` function uses the current memory pool and also provides the opportunity to create and access different pools using `hipMemPoolCreate()` and `hipMallocFromPoolAsync()` functions respectively.

Unlike NVIDIA CUDA, where stream-ordered memory allocation can be implicit, ROCm HIP is explicit. This requires managing memory allocation for each stream in HIP while ensuring precise control over memory usage and synchroniza-

tion.

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

#include <cstddef>
#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);                       \
    }                                                  \
}

// Kernel to perform some computation on allocated memory.
__global__ void myKernel(int* data, std::size_t numElements)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < numElements)
    {
        data[tid] = tid * 2;
    }
}

int main()
{
    // Create a stream.
    hipStream_t stream;
    HIP_CHECK(hipStreamCreate(&stream));

    // Create a memory pool with default properties.
    hipMemPoolProps poolProps = {};
    poolProps.allocType = hipMemAllocationTypePinned;
    poolProps.handleTypes = hipMemHandleTypePosixFileDescriptor;
    poolProps.location.type = hipMemLocationTypeDevice;
    poolProps.location.id = 0; // Assuming device 0.

    hipMemPool_t memPool;
    HIP_CHECK(hipMemPoolCreate(&memPool, &poolProps));

    // Allocate memory from the pool asynchronously.
    constexpr std::size_t numElements = 1024;
    int* devData = nullptr;
    HIP_CHECK(hipMallocFromPoolAsync(reinterpret_cast<void**>(&devData),
                                     numElements * sizeof(*devData),
                                     memPool,
                                     stream));
```

(continues on next page)

(continued from previous page)

```
    // Define grid and block sizes.
    dim3 blockSize(256);
    dim3 gridSize((numElements + blockSize.x - 1) / blockSize.x);

    // Launch the kernel to perform computation.
    myKernel<<<gridSize, blockSize, 0, stream>>>(devData, numElements);

    // Synchronize the stream.
    HIP_CHECK(hipStreamSynchronize(stream));

    // Copy data back to host.
    int* hostData = new int[numElements];
    HIP_CHECK(hipMemcpy(hostData, devData, numElements * sizeof(*devData),␣
↪hipMemcpyDeviceToHost));

    // Print the array.
    for (std::size_t i = 0; i < numElements; ++i)
        std::cout << "Element " << i << ": " << hostData[i] << std::endl;

    // Free the allocated memory.
    HIP_CHECK(hipFreeAsync(devData, stream));

    // Synchronize the stream again to ensure all operations are complete.
    HIP_CHECK(hipStreamSynchronize(stream));

    // Destroy the memory pool and stream.
    HIP_CHECK(hipMemPoolDestroy(memPool));
    HIP_CHECK(hipStreamDestroy(stream));

    // Free host memory.
    delete[] hostData;

    return EXIT_SUCCESS;
}
```

10.2.7.2.2 Trim pools

The memory allocator allows you to allocate and free memory in stream order. To control memory usage, set the release threshold attribute using `hipMemPoolAttrReleaseThreshold`. This threshold specifies the amount of reserved memory in bytes to hold onto.

```
    std::uint64_t threshold = std::numeric_limits<std::uint64_t>::max();
    HIP_CHECK(hipMemPoolSetAttribute(memPool, hipMemPoolAttrReleaseThreshold, &
↪threshold));
```

When the amount of memory held in the memory pool exceeds the threshold, the allocator tries to release memory back to the operating system during the next call to stream, event, or context synchronization.

To improve performance, it is a good practice to adjust the memory pool size using `hipMemPoolTrimTo()`. It helps to reclaim memory from an excessive memory pool, which optimizes memory usage for your application.

```
#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);                       \
    }                                                  \
}

int main()
{
    hipMemPool_t memPool;
    hipDevice_t device = 0; // Specify the device index.

    // Initialize the device.
    HIP_CHECK(hipSetDevice(device));

    // Get the default memory pool for the device.
    HIP_CHECK(hipDeviceGetDefaultMemPool(&memPool, device));

    // Allocate memory from the pool (e.g., 1 MB).
    std::size_t allocSize = 1 * 1024 * 1024;
    void* ptr;
    HIP_CHECK(hipMalloc(&ptr, allocSize));

    // Free the allocated memory.
    HIP_CHECK(hipFree(ptr));

    // Trim the memory pool to a specific size (e.g., 512 KB).
    std::size_t newSize = 512 * 1024;
    HIP_CHECK(hipMemPoolTrimTo(memPool, newSize));

    std::cout << "Memory pool trimmed to " << newSize << " bytes." << std::endl;
    return EXIT_SUCCESS;
}
```

10.2.7.2.3 Resource usage statistics

Resource usage statistics help in optimization. Here is the list of pool attributes used to query memory usage:

- `hipMemPoolAttrReservedMemCurrent`: Returns the total physical GPU memory currently held in the pool.
- `hipMemPoolAttrUsedMemCurrent`: Returns the total size of all the memory allocated from the pool.

- `hipMemPoolAttrReservedMemHigh`: Returns the total physical GPU memory held in the pool since the last reset.
- `hipMemPoolAttrUsedMemHigh`: Returns the total size of all the memory allocated from the pool since the last reset.

To reset these attributes to the current value, use `hipMemPoolSetAttribute()`.

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

#include <cstddef>
#include <cstdint>
#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);                       \
    }                                                  \
}

// Sample helper functions for getting the usage statistics in bulk.
struct usageStatistics
{
    std::uint64_t reservedMemCurrent;
    std::uint64_t reservedMemHigh;
    std::uint64_t usedMemCurrent;
    std::uint64_t usedMemHigh;
};

void getUsageStatistics(hipMemPool_t memPool, struct usageStatistics *statistics)
{
    HIP_CHECK(hipMemPoolGetAttribute(memPool, hipMemPoolAttrReservedMemCurrent, &
↪statistics->reservedMemCurrent));
    HIP_CHECK(hipMemPoolGetAttribute(memPool, hipMemPoolAttrReservedMemHigh, &
↪statistics->reservedMemHigh));
    HIP_CHECK(hipMemPoolGetAttribute(memPool, hipMemPoolAttrUsedMemCurrent, &
↪statistics->usedMemCurrent));
    HIP_CHECK(hipMemPoolGetAttribute(memPool, hipMemPoolAttrUsedMemHigh, &statistics->
↪usedMemHigh));
}

// Resetting the watermarks resets them to the current value.
void resetStatistics(hipMemPool_t memPool)
{
    std::uint64_t value = 0;
    HIP_CHECK(hipMemPoolSetAttribute(memPool, hipMemPoolAttrReservedMemHigh, &value));
```

(continues on next page)

(continued from previous page)

```
    HIP_CHECK(hipMemPoolSetAttribute(memPool, hipMemPoolAttrUsedMemHigh, &value));
}

int main()
{
    hipMemPool_t memPool;
    hipDevice_t device = 0; // Specify the device index.

    // Initialize the device.
    HIP_CHECK(hipSetDevice(device));

    // Get the default memory pool for the device.
    HIP_CHECK(hipDeviceGetDefaultMemPool(&memPool, device));

    // Allocate memory from the pool (e.g., 1 MB).
    std::size_t allocSize = 1 * 1024 * 1024;
    void* ptr;
    HIP_CHECK(hipMalloc(&ptr, allocSize));

    // Free the allocated memory.
    HIP_CHECK(hipFree(ptr));

    // Trim the memory pool to a specific size (e.g., 512 KB).
    std::size_t newSize = 512 * 1024;
    HIP_CHECK(hipMemPoolTrimTo(memPool, newSize));

    // Get and print usage statistics before resetting.
    usageStatistics statsBefore;
    getUsageStatistics(memPool, &statsBefore);
    std::cout << "Before resetting statistics:" << std::endl;
    std::cout << "Reserved Memory Current: " << statsBefore.reservedMemCurrent << "␣
↪bytes" << std::endl;
    std::cout << "Reserved Memory High: " << statsBefore.reservedMemHigh << " bytes" <
↪< std::endl;
    std::cout << "Used Memory Current: " << statsBefore.usedMemCurrent << " bytes" <<␣
↪std::endl;
    std::cout << "Used Memory High: " << statsBefore.usedMemHigh << " bytes" <<␣
↪std::endl;

    // Reset the statistics.
    resetStatistics(memPool);

    // Get and print usage statistics after resetting.
    usageStatistics statsAfter;
    getUsageStatistics(memPool, &statsAfter);
    std::cout << "After resetting statistics:" << std::endl;
    std::cout << "Reserved Memory Current: " << statsAfter.reservedMemCurrent << "␣
↪bytes" << std::endl;
    std::cout << "Reserved Memory High: " << statsAfter.reservedMemHigh << " bytes" <
↪< std::endl;
    std::cout << "Used Memory Current: " << statsAfter.usedMemCurrent << " bytes" <<␣
↪std::endl;
```

(continues on next page)

(continued from previous page)

```
    std::cout << "Used Memory High: " << statsAfter.usedMemHigh << " bytes" <<␣
↪std::endl;

    return EXIT_SUCCESS;
}
```

10.2.7.2.4 Memory reuse policies

The allocator might reallocate memory as long as the compliant memory accesses will not to overlap temporally. To optimize the memory usage, disable or enable the following memory pool reuse policy attribute flags:

- `hipMemPoolReuseFollowEventDependencies`: Checks event dependencies before allocating additional GPU memory.
- `hipMemPoolReuseAllowOpportunistic`: Checks freed allocations to determine if the stream order semantic indicated by the free operation has been met.
- `hipMemPoolReuseAllowInternalDependencies`: Manages reuse based on internal dependencies in runtime. If the driver fails to allocate and map additional physical memory, it searches for memory waiting for another stream's progress and reuses it.

10.2.7.2.5 Device accessibility for multi-GPU support

Allocations are initially accessible from the device where they reside.

10.2.7.3 Interprocess memory handling

> **Note**
>
> IPC API calls are only supported on systems with an active `amdgpu-dkms` driver. For more information, see AMD GPU Driver (amdgpu).

Interprocess capable (IPC) memory pools facilitate efficient and secure sharing of GPU memory between processes.

To achieve interprocess memory sharing, you can use either *device pointer* or *shareable handle*. Both provide allocator (export) and consumer (import) interfaces.

10.2.7.3.1 Device pointer

To export data to share a memory pool pointer directly between processes, use `hipMemPoolExportPointer()`. It allows you to share a memory allocation with another process.

```
#include <iostream>
#include <fstream>
#include <hip/hip_runtime.h>
#include <sys/stat.h>

int main() {
    // Allocate memory.
    void* devPtr;
    hipMalloc(&devPtr, sizeof(int));
```

(continues on next page)

(continued from previous page)

```
    // Export the memory pool pointer.
    hipMemPoolPtrExportData exportData;
    hipError_t result = hipMemPoolExportPointer(&exportData, devPtr);
    if (result != hipSuccess) {
        std::cerr << "Error exporting memory pool pointer: " <<␣
↪hipGetErrorString(result) << std::endl;
        return 1;
    }

    // Create a named pipe (FIFO).
    const char* fifoPath = "/tmp/myfifo"; // Change this to a unique path.
    mkfifo(fifoPath, 0666);

    // Write the exported data to the named pipe.
    std::ofstream fifoStream(fifoPath, std::ios::out | std::ios::binary);
    fifoStream.write(reinterpret_cast<char*>(&exportData),␣
↪sizeof(hipMemPoolPtrExportData));
    fifoStream.close();

    // Clean up.
    hipFree(devPtr);

    return 0;
}
```

To import a memory pool pointer directly from another process, use `hipMemPoolImportPointer()`.

Here is how to read the pool exported in the preceding example:

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

int main() {
    // Considering that you have exported the memory pool pointer already.
    // Now, let's simulate reading the exported data from a named pipe (FIFO).
    const char* fifoPath = "/tmp/myfifo"; // Change this to a unique path.
    std::ifstream fifoStream(fifoPath, std::ios::in | std::ios::binary);

    if (!fifoStream.is_open()) {
        std::cerr << "Error opening FIFO file: " << fifoPath << std::endl;
        return 1;
    }

    // Read the exported data.
    hipMemPoolPtrExportData importData;
    fifoStream.read(reinterpret_cast<char*>(&importData),␣
↪sizeof(hipMemPoolPtrExportData));
    fifoStream.close();

    if (fifoStream.fail()) {
        std::cerr << "Error reading from FIFO file." << std::endl;
        return 1;
```

(continues on next page)

(continued from previous page)

```
    }

    // Create a memory pool with default properties.
    hipMemPoolProps poolProps = {};
    poolProps.allocType = hipMemAllocationTypePinned;
    poolProps.handleTypes = hipMemHandleTypePosixFileDescriptor;
    poolProps.location.type = hipMemLocationTypeDevice;
    poolProps.location.id = 0; // Assuming device 0.

    hipMemPool_t memPool;
    hipMemPoolCreate(&memPool, &poolProps);

    // Import the memory pool pointer.
    void* importedDevPtr;
    hipError_t result = hipMemPoolImportPointer(&importedDevPtr, memPool, &
↪importData);
    if (result != hipSuccess) {
        std::cerr << "Error imported memory pool pointer: " <<␣
↪hipGetErrorString(result) << std::endl;
        return 1;
    }

    // Now you can use the importedDevPtr for your computations.

    // Clean up (free the memory).
    hipFree(importedDevPtr);

    return 0;
}
```

10.2.7.3.2 Shareable handle

To export a memory pool pointer to a shareable handle, use `hipMemPoolExportToSharedHandle()`. This handle could be a file descriptor or a handle obtained from another process. The exported handle contains information about the memory pool, such as size, location, and other relevant details.

```
#include <iostream>
#include <fstream>
#include <hip/hip_runtime.h>
#include <sys/stat.h>

int main() {
    // Create a memory pool with default properties.
    hipMemPoolProps poolProps = {};
    poolProps.allocType = hipMemAllocationTypePinned;
    poolProps.handleTypes = hipMemHandleTypePosixFileDescriptor;
    poolProps.location.type = hipMemLocationTypeDevice;
    poolProps.location.id = 0; // Assuming device 0.

    hipMemPool_t memPool;
    hipError_t poolResult = hipMemPoolCreate(&memPool, &poolProps);
    if (poolResult != hipSuccess) {
```

(continues on next page)

(continued from previous page)

```
        std::cerr << "Error creating memory pool: " << hipGetErrorString(poolResult) <
↪< std::endl;
        return 1;
    }

    // Allocate memory from the memory pool.
    void* devPtr;
    hipMallocFromPoolAsync(&devPtr, sizeof(int), memPool, 0);

    // Export the memory pool pointer.
    int descriptor;
    hipError_t result = hipMemPoolExportToShareableHandle(&descriptor, memPool,␣
↪hipMemHandleTypePosixFileDescriptor, 0);
    if (result != hipSuccess) {
        std::cerr << "Error exporting memory pool pointer: " <<␣
↪hipGetErrorString(result) << std::endl;
        return 1;
    }

    // Create a named pipe (FIFO).
    const char* fifoPath = "/tmp/myfifo"; // Change this to a unique path.
    mkfifo(fifoPath, 0666);

    // Write the exported data to the named pipe.
    std::ofstream fifoStream(fifoPath, std::ios::out | std::ios::binary);
    fifoStream.write(reinterpret_cast<char*>(&descriptor), sizeof(int));
    fifoStream.close();

    // Clean up.
    hipFree(devPtr);
    hipMemPoolDestroy(memPool);

    return 0;
}
```

To import and restore a memory pool pointer from a shareable handle, which could be a file descriptor or a handle obtained from another process, use `hipMemPoolImportFromShareableHandle()`. The exported shareable handle data contains information about the memory pool, including its size, location, and other relevant details. Importing the handle provides a valid memory pointer to the same memory, which allows you to share memory across different contexts.

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

int main() {
    // Considering that you have exported the memory pool pointer already.
    // Now, let's simulate reading the exported data from a named pipe (FIFO).
    const char* fifoPath = "/tmp/myfifo"; // Change this to a unique path
    std::ifstream fifoStream(fifoPath, std::ios::in | std::ios::binary);

    if (!fifoStream.is_open()) {
        std::cerr << "Error opening FIFO file: " << fifoPath << std::endl;
```

(continues on next page)

(continued from previous page)

```
        return 1;
    }

    // Read the exported data.
    int descriptor;
    fifoStream.read(reinterpret_cast<char*>(&descriptor), sizeof(int));
    fifoStream.close();

    if (fifoStream.fail()) {
        std::cerr << "Error reading from FIFO file." << std::endl;
        return 1;
    }

    // Import the memory pool.
    hipMemPool_t memPool;
    hipError_t result = hipMemPoolImportFromShareableHandle(&memPool, &descriptor,␣
↪hipMemHandleTypePosixFileDescriptor, 0);
    if (result != hipSuccess) {
        std::cerr << "Error importing memory pool: " << hipGetErrorString(result) <<␣
↪std::endl;
        return 1;
    }

    // Allocate memory from the imported memory pool.
    void* importedDevPtr;
    hipMallocFromPoolAsync(&importedDevPtr, sizeof(int), memPool, 0);

    // Now you can use the importedDevPtr for your computations.

    // Clean up (free the memory).
    hipFree(importedDevPtr);
    hipMemPoolDestroy(memPool);

    return 0;
}
```

10.3 Error handling

HIP provides functionality to detect, report, and manage errors that occur during the execution of HIP runtime functions or when launching kernels. Every HIP runtime function, apart from launching kernels, has `hipError_t` as return type. `hipGetLastError()` and `hipPeekAtLastError()` can be used for catching errors from kernel launches, as kernel launches don't return an error directly. HIP maintains an internal state, that includes the last error code. `hipGetLastError()` returns and resets that error to `hipSuccess`, while `hipPeekAtLastError()` just returns the error without changing it. To get a human readable version of the errors, `hipGetErrorString()` and `hipGetErrorName()` can be used.

Note

`hipGetLastError()` returns the last actual HIP API error caught in the current thread during the application execution. Prior to ROCm 7.0, `hipGetLastError` might also return `hipSuccess` or `hipErrorNotReady` from the

last HIP runtime API call, which are not errors.

Best practices of HIP error handling:

1. Check errors after each API call - Avoid error propagation.
2. Use macros for error checking - Check *HIP check macros*.
3. Handle errors gracefully - Free resources and provide meaningful error messages to the user.

For more details on the error handling functions, see error handling functions reference page.

For a list of all error codes, see HIP error codes.

10.3.1 HIP check macros

HIP uses check macros to simplify error checking and reduce code duplication. The `HIP_CHECK` macros are mainly used to detect and report errors. It can also exit from application with `exit(1);` function call after the error print. The `HIP_CHECK` macro example:

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

10.3.2 Complete example

A complete example to demonstrate the error handling with a simple addition of two values kernel:

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

#include <algorithm>
#include <cstddef>
#include <cstdlib>
#include <iostream>
#include <vector>

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

(continues on next page)

(continued from previous page)

```
}

// Addition of two values.
__global__ void add(int *a, int *b, int *c, std::size_t size)
{
    const std::size_t index = threadIdx.x + blockDim.x * blockIdx.x;
    if(index < size)
    {
        c[index] += a[index] + b[index];
    }
}

int main()
{
    constexpr int numOfBlocks = 256;
    constexpr int threadsPerBlock = 256;
    constexpr std::size_t arraySize = 1U << 16;

    std::vector<int> a(arraySize), b(arraySize), c(arraySize);
    int *d_a, *d_b, *d_c;

    // Setup input values.
    std::fill(a.begin(), a.end(), 1);
    std::fill(b.begin(), b.end(), 2);

    // Allocate device copies of a, b and c.
    HIP_CHECK(hipMalloc(&d_a, arraySize * sizeof(int)));
    HIP_CHECK(hipMalloc(&d_b, arraySize * sizeof(int)));
    HIP_CHECK(hipMalloc(&d_c, arraySize * sizeof(int)));

    // Copy input values to device.
    HIP_CHECK(hipMemcpy(d_a, a.data(), arraySize * sizeof(int),␣
↪hipMemcpyHostToDevice));
    HIP_CHECK(hipMemcpy(d_b, b.data(), arraySize * sizeof(int),␣
↪hipMemcpyHostToDevice));

    // Launch add() kernel on GPU.
    add<<<numOfBlocks, threadsPerBlock>>>(d_a, d_b, d_c, arraySize);
    // Check the kernel launch
    HIP_CHECK(hipGetLastError());
    // Check for kernel execution error
    HIP_CHECK(hipDeviceSynchronize());

    // Copy the result back to the host.
    HIP_CHECK(hipMemcpy(c.data(), d_c, arraySize * sizeof(int),␣
↪hipMemcpyDeviceToHost));

    // Cleanup allocated memory.
    HIP_CHECK(hipFree(d_a));
    HIP_CHECK(hipFree(d_b));
    HIP_CHECK(hipFree(d_c));
```

(continues on next page)

(continued from previous page)

```
    // Print the result.
    std::cout << a[0] << " + " << b[0] << " = " << c[0] << std::endl;

    return EXIT_SUCCESS;
}
```

10.4 Asynchronous concurrent execution

Asynchronous concurrent execution is important for efficient parallelism and resource utilization, with techniques such as overlapping computation and data transfer, managing concurrent kernel execution with streams on single or multiple devices, or using HIP graphs.

10.4.1 Streams and concurrent execution

All asynchronous APIs, such as kernel execution, data movement and potentially data allocation/freeing all happen in the context of device streams.

Streams are FIFO buffers of commands to execute in order on a given device. Commands which enqueue tasks on a stream all return promptly and the task is executed asynchronously. Multiple streams can point to the same device and those streams might be fed from multiple concurrent host-side threads. Multiple streams tied to the same device are not guaranteed to execute their commands in order.

10.4.1.1 Managing streams

Streams enable the overlap of computation and data transfer, ensuring continuous GPU activity. By enabling tasks to run concurrently within the same GPU or across different GPUs, streams improve performance and throughput in high-performance computing (HPC).

To create a stream, the following functions are used, each defining a handle to the newly created stream:

- `hipStreamCreate()`: Creates a stream with default settings.
- `hipStreamCreateWithFlags()`: Creates a stream, with specific flags, listed below, enabling more control over stream behavior:
  - `hipStreamDefault`: creates a default stream suitable for most operations. The default stream is a blocking operation.
  - `hipStreamNonBlocking`: creates a non-blocking stream, allowing concurrent execution of operations. It ensures that tasks can run simultaneously without waiting for each other to complete, thus improving overall performance.
- `hipStreamCreateWithPriority()`: Allows creating a stream with a specified priority, enabling prioritization of certain tasks.

The `hipStreamSynchronize()` function is used to block the calling host thread until all previously submitted tasks in a specified HIP stream have completed. It ensures that all operations in the given stream, such as kernel executions or memory transfers, are finished before the host thread proceeds.

> **Note**
>
> If the `hipStreamSynchronize()` function input stream is 0 (or the default stream), it waits for all operations in the default stream to complete.

10.4.1.2 Concurrent execution between host and device

Concurrent execution between the host (CPU) and device (GPU) allows the CPU to perform other tasks while the GPU is executing kernels. Kernels are launched asynchronously using `hipLaunchKernelGGL` or using the triple chevron with a stream, enabling the CPU to continue executing other code while the GPU processes the kernel. Similarly, memory operations like `hipMemcpyAsync()` are performed asynchronously, allowing data transfers between the host and device without blocking the CPU.

10.4.1.3 Concurrent kernel execution

Concurrent execution of multiple kernels on the GPU allows different kernels to run simultaneously to maximize GPU resource usage. Managing dependencies between kernels is crucial for ensuring correct execution order. This can be achieved using `hipStreamWaitEvent()`, which allows a kernel to wait for a specific event before starting execution.

Independent kernels can only run concurrently if there are enough registers and shared memory for the kernels. To enable concurrent kernel executions, the developer may have to reduce the block size of the kernels. The kernel runtimes can be misleading for concurrent kernel runs, that is why during optimization it is a good practice to check the trace files, to see if one kernel is blocking another kernel, while they are running in parallel. For more information about application tracing, see Using rocprof.

When running kernels in parallel, the execution time can increase due to contention for shared resources. This is because multiple kernels may attempt to access the same GPU resources simultaneously, leading to delays.

Multiple kernels executing concurrently is only beneficial under specific conditions. It is most effective when the kernels do not fully utilize the GPU's resources. In such cases, overlapping kernel execution can improve overall throughput and efficiency by keeping the GPU busy without exceeding its capacity.

10.4.2 Overlap of data transfer and kernel execution

One of the primary benefits of asynchronous operations and multiple streams is the ability to overlap data transfer with kernel execution, leading to better resource utilization and improved performance.

Asynchronous execution is particularly advantageous in iterative processes. For instance, if a kernel is initiated, it can be efficient to prepare the input data simultaneously, provided that this preparation does not depend on the kernel's execution. Such iterative data transfer and kernel execution overlap can be find in the *Example*.

10.4.2.1 Querying device capabilities

Some AMD HIP-enabled devices can perform asynchronous memory copy operations to or from the GPU concurrently with kernel execution. Applications can query this capability by checking the `asyncEngineCount` device property. Devices with an `asyncEngineCount` greater than zero support concurrent data transfers. Additionally, if host memory is involved in the copy, it should be page-locked to ensure optimal performance. Page-locking (or pinning) host memory increases the bandwidth between the host and the device, reducing the overhead associated with data transfers. For more details, visit *Host memory* page.

10.4.2.2 Asynchronous memory operations

Asynchronous memory operations do not block the host while copying data and, when used with multiple streams, allow data to be transferred between the host and device while kernels are executed on the same GPU. Using operations like `hipMemcpyAsync()` or `hipMemcpyPeerAsync()`, developers can initiate data transfers without waiting for the previous operation to complete. This overlap of computation and data transfer ensures that the GPU is not idle while waiting for data. `hipMemcpyPeerAsync()` enables data transfers between different GPUs, facilitating multi-GPU communication.

Example include launching kernels in one stream while performing data transfers in another. This technique is especially useful in applications with large data sets that need to be processed quickly.

10.4.2.3 Concurrent data transfers with intra-device copies

Devices that support the `concurrentKernels` property can perform intra-device copies concurrently with kernel execution. Additionally, devices that support the `asyncEngineCount` property can perform data transfers to or from the GPU simultaneously with kernel execution. Intra-device copies can be initiated using standard memory copy functions with destination and source addresses residing on the same device.

10.4.3 Synchronization, event management and synchronous calls

Synchronization and event management are important for coordinating tasks and ensuring correct execution order, and synchronous calls are necessary for maintaining data consistency.

10.4.3.1 Synchronous calls

Synchronous calls ensure task completion before moving to the next operation. For example, `hipMemcpy()` for data transfers waits for completion before returning control to the host. Similarly, synchronous kernel launches are used when immediate completion is required. When a synchronous function is called, control is not returned to the host thread before the device has completed the requested task. The behavior of the host thread—whether to yield, block, or spin—can be specified using `hipSetDeviceFlags()` with appropriate flags. Understanding when to use synchronous calls is important for managing execution flow and avoiding data races.

10.4.3.2 Events for synchronization

By creating an event with `hipEventCreate()` and recording it with `hipEventRecord()`, developers can synchronize operations across streams, ensuring correct task execution order. `hipEventSynchronize()` lets the application wait for an event to complete before proceeding with the next operation.

10.4.3.3 Programmatic dependent launch and synchronization

While CUDA supports programmatic dependent launches allowing a secondary kernel to start before the primary kernel finishes, HIP achieves similar functionality using streams and events. By employing `hipStreamWaitEvent()`, it is possible to manage the execution order without explicit hardware support. This mechanism allows a secondary kernel to launch as soon as the necessary conditions are met, even if the primary kernel is still running.

10.4.3.4 Example

The examples shows the difference between sequential, asynchronous calls and asynchronous calls with `hipEvents`.

The example codes

Sequential

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

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

#define HIP_CHECK(expression)                \
{                                            \
    const hipError_t status = expression;    \
    if(status != hipSuccess)                 \
    {                                        \
            std::cerr << "HIP error "        \
```

(continues on next page)


ntial calls:
ult stream
H2D data
H2D data
kernel data1
kernel data2
D2H data
D2H data2
onous calls:
tream1
H2D data
kernel data1
D2H data1
tream2
H2D data2
kernel data2
D2H data2
calls with hipEvent:
event
eventA
eventB
tream1
H2D data
kernel data1
D2H data1
tream2
H2D data2
kernel data2
D2H data2
time

(continued from previous page)

```
                << status << ": "            \
                << hipGetErrorString(status) \
                << " at " << __FILE__ << ":" \
                << __LINE__ << std::endl;    \
    }                                        \
}

// GPU Kernels
__global__ void kernelA(double* arrayA, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayA[x] += 1.0;
    }
}

__global__ void kernelB(double* arrayA, double* arrayB, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayB[x] += arrayA[x] + 3.0;
    }
}

int main()
{
    constexpr int numOfBlocks = 1 << 20;
    constexpr int threadsPerBlock = 1024;
    constexpr int numberOfIterations = 50;
    // The array size smaller to avoid the relatively short kernel launch compared to␣
↪memory copies
    constexpr std::size_t arraySize = 1U << 25;
    double *d_dataA;
    double *d_dataB;

    double initValueA = 0.0;
    double initValueB = 2.0;

    std::vector<double> vectorA(arraySize, initValueA);
    std::vector<double> vectorB(arraySize, initValueB);
    // Allocate device memory
    HIP_CHECK(hipMalloc(&d_dataA, arraySize * sizeof(*d_dataA)));
    HIP_CHECK(hipMalloc(&d_dataB, arraySize * sizeof(*d_dataB)));
    for(int iteration = 0; iteration < numberOfIterations; iteration++)
    {
        // Host to Device copies
        HIP_CHECK(hipMemcpy(d_dataA, vectorA.data(), arraySize * sizeof(*d_dataA),␣
↪hipMemcpyHostToDevice));
        HIP_CHECK(hipMemcpy(d_dataB, vectorB.data(), arraySize * sizeof(*d_dataB),␣
↪hipMemcpyHostToDevice));
```

(continues on next page)

(continued from previous page)

```
        // Launch the GPU kernels
        kernelA<<<numOfBlocks, threadsPerBlock>>>(d_dataA, arraySize);
        kernelB<<<numOfBlocks, threadsPerBlock>>>(d_dataA, d_dataB, arraySize);
        // Device to Host copies
        HIP_CHECK(hipMemcpy(vectorA.data(), d_dataA, arraySize * sizeof(*vectorA.
↪data()), hipMemcpyDeviceToHost));
        HIP_CHECK(hipMemcpy(vectorB.data(), d_dataB, arraySize * sizeof(*vectorB.
↪data()), hipMemcpyDeviceToHost));
    }
    // Wait for all operations to complete
    HIP_CHECK(hipDeviceSynchronize());

    // Verify results
    const double expectedA = (double)numberOfIterations;
    const double expectedB = initValueB + (3.0 * numberOfIterations) + (expectedA *␣
↪(expectedA + 1.0)) / 2.0;
    bool passed = true;
    for(std::size_t i = 0; i < arraySize; ++i)
    {
        if(vectorA[i] != expectedA)
        {
            passed = false;
            std::cerr << "Validation failed! Expected " << expectedA << " got " <<␣
↪vectorA[i] << " at index: " << i << std::endl;
            break;
        }
        if(vectorB[i] != expectedB)
        {
            passed = false;
            std::cerr << "Validation failed! Expected " << expectedB << " got " << ␣
↪vectorB[i] << " at index: " << i << std::endl;
            break;
        }
    }

    if(passed)
    {
        std::cout << "Sequential execution completed successfully." << std::endl;
    }
    else
    {
        std::cerr << "Sequential execution failed." << std::endl;
    }

    // Cleanup
    HIP_CHECK(hipFree(d_dataA));
    HIP_CHECK(hipFree(d_dataB));

    return EXIT_SUCCESS;
}
```

Asynchronous

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

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

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

// GPU Kernels
__global__ void kernelA(double* arrayA, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayA[x] += 1.0;
    }
}

__global__ void kernelB(double* arrayA, double* arrayB, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayB[x] += arrayA[x] + 3.0;
    }
}

int main()
{
    constexpr int numOfBlocks = 1 << 20;
    constexpr int threadsPerBlock = 1024;
    constexpr int numberOfIterations = 50;
    // The array size smaller to avoid the relatively short kernel launch compared to␣
↪memory copies
    constexpr std::size_t arraySize = 1U << 25;
    double *d_dataA;
    double *d_dataB;

    double initValueA = 0.0;
```

(continues on next page)

(continued from previous page)

```
    double initValueB = 2.0;

    std::vector<double> vectorA(arraySize, initValueA);
    std::vector<double> vectorB(arraySize, initValueB);
    // Allocate device memory
    HIP_CHECK(hipMalloc(&d_dataA, arraySize * sizeof(*d_dataA)));
    HIP_CHECK(hipMalloc(&d_dataB, arraySize * sizeof(*d_dataB)));
    // Create streams
    hipStream_t streamA, streamB;
    HIP_CHECK(hipStreamCreate(&streamA));
    HIP_CHECK(hipStreamCreate(&streamB));
    for(unsigned int iteration = 0; iteration < numberOfIterations; iteration++)
    {
        // Stream 1: Host to Device 1
        HIP_CHECK(hipMemcpyAsync(d_dataA, vectorA.data(), arraySize * sizeof(*d_
↪dataA), hipMemcpyHostToDevice, streamA));
        // Stream 2: Host to Device 2
        HIP_CHECK(hipMemcpyAsync(d_dataB, vectorB.data(), arraySize * sizeof(*d_
↪dataB), hipMemcpyHostToDevice, streamB));
        // Stream 1: Kernel 1
        kernelA<<<numOfBlocks, threadsPerBlock, 0, streamA>>>(d_dataA, arraySize);
        // Wait for streamA finish
        HIP_CHECK(hipStreamSynchronize(streamA));
        // Stream 2: Kernel 2
        kernelB<<<numOfBlocks, threadsPerBlock, 0, streamB>>>(d_dataA, d_dataB,␣
↪arraySize);
        // Stream 1: Device to Host 2 (after Kernel 1)
        HIP_CHECK(hipMemcpyAsync(vectorA.data(), d_dataA, arraySize * sizeof(*vectorA.
↪data()), hipMemcpyDeviceToHost, streamA));
        // Stream 2: Device to Host 2 (after Kernel 2)
        HIP_CHECK(hipMemcpyAsync(vectorB.data(), d_dataB, arraySize * sizeof(*vectorB.
↪data()), hipMemcpyDeviceToHost, streamB));
    }
    // Wait for all operations in both streams to complete
    HIP_CHECK(hipStreamSynchronize(streamA));
    HIP_CHECK(hipStreamSynchronize(streamB));
    // Verify results
    double expectedA = (double)numberOfIterations;
    double expectedB = initValueB + (3.0 * numberOfIterations) + (expectedA *␣
↪(expectedA + 1.0)) / 2.0;
    bool passed = true;
    for(std::size_t i = 0; i < arraySize; ++i)
    {
        if(vectorA[i] != expectedA)
        {
            passed = false;
            std::cerr << "Validation failed! Expected " << expectedA << " got " <<␣
↪vectorA[i] << " at index: " << i << std::endl;
            break;
        }
        if(vectorB[i] != expectedB)
        {
```

(continues on next page)

(continued from previous page)

```
            passed = false;
            std::cerr << "Validation failed! Expected " << expectedB << " got " << ␣
↪vectorB[i] << " at index: " << i << std::endl;
            break;
        }
    }

    if(passed)
    {
        std::cout << "Asynchronous execution completed successfully." << std::endl;
    }
    else
    {
        std::cerr << "Asynchronous execution failed." << std::endl;
    }

    // Cleanup
    HIP_CHECK(hipStreamDestroy(streamA));
    HIP_CHECK(hipStreamDestroy(streamB));
    HIP_CHECK(hipFree(d_dataA));
    HIP_CHECK(hipFree(d_dataB));

    return EXIT_SUCCESS;
}
```

hipStreamWaitEvent

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

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

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

// GPU Kernels
__global__ void kernelA(double* arrayA, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
```

(continues on next page)

(continued from previous page)

```
    if(x < size)
    {
        arrayA[x] += 1.0;
    }
}

__global__ void kernelB(double* arrayA, double* arrayB, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayB[x] += arrayA[x] + 3.0;
    }
}

int main()
{
    constexpr int numOfBlocks = 1 << 20;
    constexpr int threadsPerBlock = 1024;
    constexpr int numberOfIterations = 50;
    // The array size smaller to avoid the relatively short kernel launch compared to␣
↪memory copies
    constexpr std::size_t arraySize = 1U << 25;
    double *d_dataA;
    double *d_dataB;
    double initValueA = 0.0;
    double initValueB = 2.0;

    std::vector<double> vectorA(arraySize, initValueA);
    std::vector<double> vectorB(arraySize, initValueB);
    // Allocate device memory
    HIP_CHECK(hipMalloc(&d_dataA, arraySize * sizeof(*d_dataA)));
    HIP_CHECK(hipMalloc(&d_dataB, arraySize * sizeof(*d_dataB)));
    // Create streams
    hipStream_t streamA, streamB;
    HIP_CHECK(hipStreamCreate(&streamA));
    HIP_CHECK(hipStreamCreate(&streamB));
    // Create events
    hipEvent_t event, eventA, eventB;
    HIP_CHECK(hipEventCreate(&event));
    HIP_CHECK(hipEventCreate(&eventA));
    HIP_CHECK(hipEventCreate(&eventB));
    for(unsigned int iteration = 0; iteration < numberOfIterations; iteration++)
    {
        // Stream 1: Host to Device 1
        HIP_CHECK(hipMemcpyAsync(d_dataA, vectorA.data(), arraySize * sizeof(*d_
↪dataA), hipMemcpyHostToDevice, streamA));
        // Stream 2: Host to Device 2
        HIP_CHECK(hipMemcpyAsync(d_dataB, vectorB.data(), arraySize * sizeof(*d_
↪dataB), hipMemcpyHostToDevice, streamB));
        // Stream 1: Kernel 1
        kernelA<<<numOfBlocks, threadsPerBlock, 0, streamA>>>(d_dataA, arraySize);
```

(continues on next page)

(continued from previous page)

```
        // Record event after the GPU kernel in Stream 1
        HIP_CHECK(hipEventRecord(event, streamA));
        // Stream 2: Wait for event before starting Kernel 2
        HIP_CHECK(hipStreamWaitEvent(streamB, event, 0));
        // Stream 2: Kernel 2
        kernelB<<<numOfBlocks, threadsPerBlock, 0, streamB>>>(d_dataA, d_dataB,␣
↪arraySize);
        // Stream 1: Device to Host 2 (after Kernel 1)
        HIP_CHECK(hipMemcpyAsync(vectorA.data(), d_dataA, arraySize * sizeof(*vectorA.
↪data()), hipMemcpyDeviceToHost, streamA));
        // Stream 2: Device to Host 2 (after Kernel 2)
        HIP_CHECK(hipMemcpyAsync(vectorB.data(), d_dataB, arraySize * sizeof(*vectorB.
↪data()), hipMemcpyDeviceToHost, streamB));
        // Wait for all operations in both streams to complete
        HIP_CHECK(hipEventRecord(eventA, streamA));
        HIP_CHECK(hipEventRecord(eventB, streamB));
        HIP_CHECK(hipStreamWaitEvent(streamA, eventA, 0));
        HIP_CHECK(hipStreamWaitEvent(streamB, eventB, 0));
    }
    // Verify results
    double expectedA = (double)numberOfIterations;
    double expectedB = initValueB + (3.0 * numberOfIterations) + (expectedA *␣
↪(expectedA + 1.0)) / 2.0;
    bool passed = true;
    for(std::size_t i = 0; i < arraySize; ++i)
    {
        if(vectorA[i] != expectedA)
        {
            passed = false;
            std::cerr << "Validation failed! Expected " << expectedA << " got " <<␣
↪vectorA[i] << std::endl;
            break;
        }
        if(vectorB[i] != expectedB)
        {
            passed = false;
            std::cerr << "Validation failed! Expected " << expectedB << " got " << ␣
↪vectorB[i] << std::endl;
            break;
        }
    }

    if(passed)
    {
        std::cout << "Asynchronous execution with events completed successfully." <<␣
↪std::endl;
    }
    else
    {
        std::cerr << "Asynchronous execution with events failed." << std::endl;
    }
```

(continues on next page)

(continued from previous page)

```
    // Cleanup
    HIP_CHECK(hipEventDestroy(event));
    HIP_CHECK(hipEventDestroy(eventA));
    HIP_CHECK(hipEventDestroy(eventB));
    HIP_CHECK(hipStreamDestroy(streamA));
    HIP_CHECK(hipStreamDestroy(streamB));
    HIP_CHECK(hipFree(d_dataA));
    HIP_CHECK(hipFree(d_dataB));

    return 0;
}
```

10.4.4 HIP Graphs

HIP graphs offer an efficient alternative to the standard method of launching GPU tasks via streams. Comprising nodes for operations and edges for dependencies, HIP graphs reduce kernel launch overhead and provide a high-level abstraction for managing dependencies and synchronization. By representing sequences of kernels and memory operations as a single graph, they simplify complex workflows and enhance performance, particularly for applications with intricate dependencies and multiple execution stages. For more details, see the *HIP graphs* documentation.

10.5 Cooperative groups

The cooperative groups API is an extension to the HIP programming model, which provides developers with a flexible, dynamic grouping mechanism for the communicating threads. Cooperative groups let you define your own set of thread groups which may fit your use-cases better than those defined by the hardware. This lets you specify the level of granularity for thread communication which can lead to more efficient parallel decompositions.

The API is accessible in the `cooperative_groups` namespace after the `hip_cooperative_groups.h` header is included. The header contains the following elements:

- Static functions to create groups and subgroups.
- Hardware-accelerated operations over the whole group, like shuffles.
- Data types of cooperative groups.
- Synchronize member function of the groups.
- Get group properties member functions.

10.5.1 Cooperative groups thread model

The thread hierarchy abstractions of cooperative groups are depicted in the following figures: *grid hierarchy* and *block hierarchy*.

The **multi grid** is an abstraction of potentially multiple simultaneous launches of the same kernel over multiple devices. The **grid** in cooperative groups is a single dispatch of kernels for execution like the original grid.

Note

- The ability to synchronize over a grid or multi grid requires the kernel to be launched using the specific cooperative groups API.
- Multi grid deprecated since ROCm 5.0.

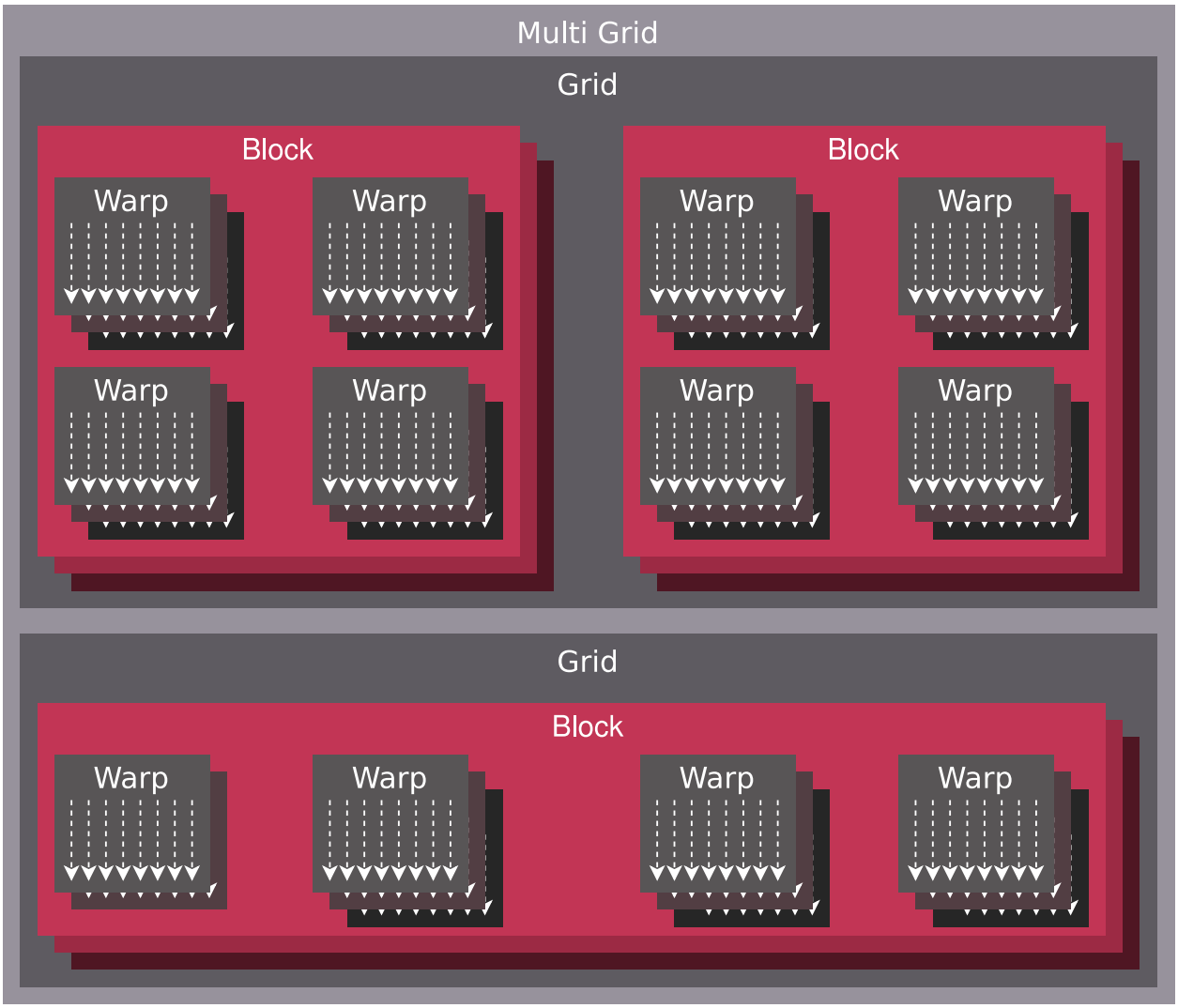


Fig. 8: Cooperative group thread hierarchy in grids.

The **block** is the same as the *Hierarchical thread model* block entity.

> **Note**
>
> Explicit warp-level thread handling is absent from the Cooperative groups API. In order to exploit the known hardware SIMD width on which built-in functionality translates to simpler logic, you can use the group partitioning part of the API, such as `tiled_partition`.

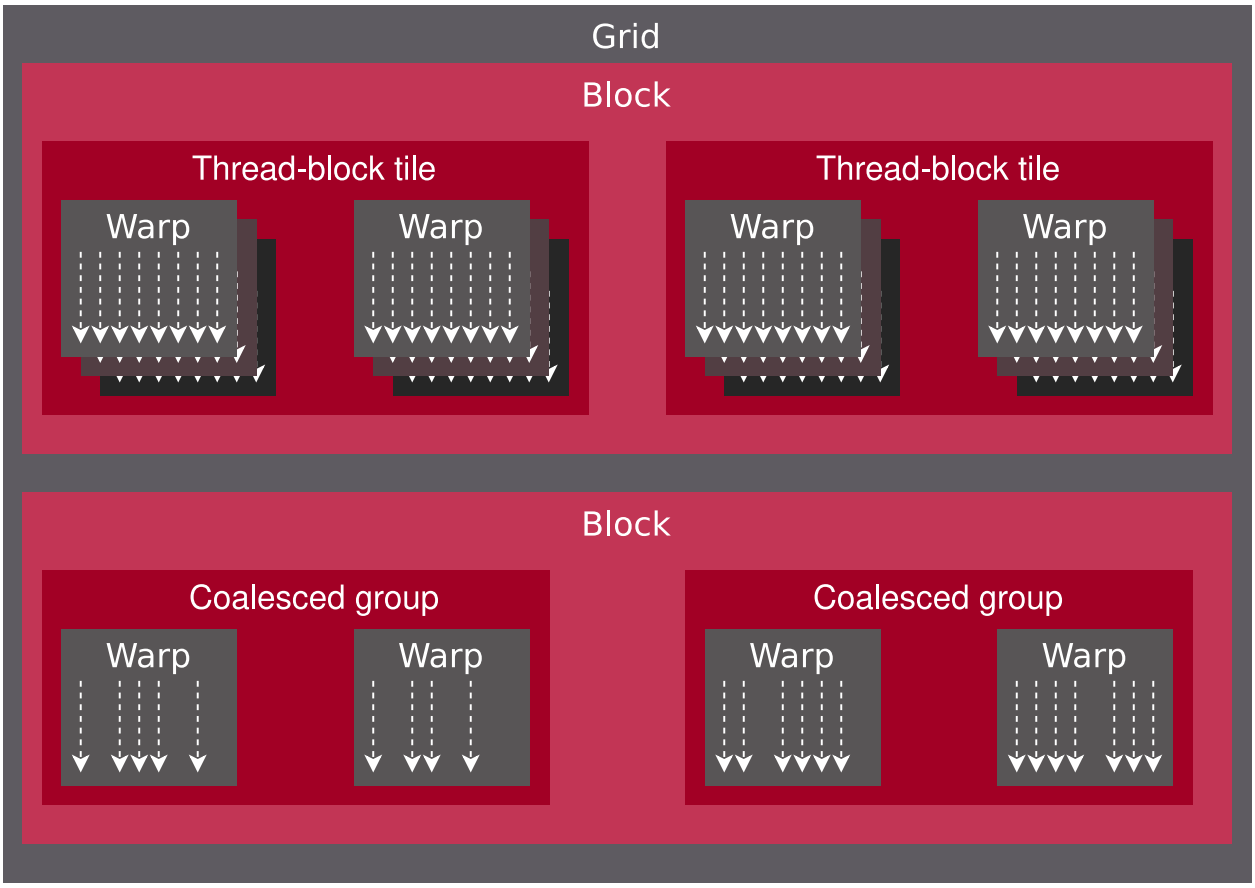


Fig. 9: Cooperative group thread hierarchy in blocks.

The cooperative groups API introduce a new level between block thread and threads. The *thread-block tile* give the opportunity to have tiles in the thread block, while the *coalesced group* holds the active threads of the parent group. These groups further discussed in the *groups types* section.

For details on memory model, check the *memory model description*.

10.5.2 Group types

Group types are based on the levels of synchronization and data sharing among threads.

10.5.2.1 Thread-block group

Represents an intra-block cooperative groups type where the participating threads within the group are the same threads that participated in the currently executing `block`.

```
class thread_block;
```

Constructed via:

```
thread_block g = this_thread_block();
```

The `group_index()` , `thread_index()` , `thread_rank()` , `size()`, `cg_type()`, `is_valid()` , `sync()` and `group_dim()` member functions are public of the thread_block class. For further details, check the thread_block references .

10.5.2.2 Grid group

Represents an inter-block cooperative groups type where the group's participating threads span multiple blocks running the same kernel on the same device. Use the cooperative launch API to enable synchronization across the grid group.

```
class grid_group;
```

Constructed via:

```
grid_group g = this_grid();
```

The `thread_rank()` , `size()`, `cg_type()`, `is_valid()` and `sync()` member functions are public of the `grid_group` class. For further details, check the grid_group references.

10.5.2.3 Multi-grid group

Represents an inter-device cooperative groups type where the participating threads within the group span multiple devices that run the same kernel on the devices. Use the cooperative launch API to enable synchronization across the multi-grid group.

```
class multi_grid_group;
```

Constructed via:

```
// Kernel must be launched with the cooperative multi-device API
multi_grid_group g = this_multi_grid();
```

The `num_grids()` , `grid_rank()` , `thread_rank()`, `size()`, `cg_type()`, `is_valid()` , and `sync()` member functions are public of the `multi_grid_group` class. For further details check the multi_grid_group references .

10.5.2.4 Thread-block tile

This constructs a templated class derived from `thread_group`. The template defines the tile size of the new thread group at compile time. This group type also supports sub-wave level intrinsics.

```
template <unsigned int Size, typename ParentT = void>
class thread_block_tile;
```

Constructed via:

```
template <unsigned int Size, typename ParentT>
_CG_QUALIFIER thread_block_tile<Size, ParentT> tiled_partition(const ParentT& g)
```

Note

- Size must be a power of 2 and not larger than warp (wavefront) size.
- `shfl()` functions support integer or float type.

The `thread_rank()`, `size()`, `cg_type()`, `is_valid()`, `sync()`, `meta_group_rank()`, `meta_group_size()`, `shfl()`, `shfl_down()`, `shfl_up()`, `shfl_xor()`, `ballot()`, `any()`, `all()`, `match_any()` and `match_all()` member functions are public of the `thread_block_tile` class. For further details, check the thread_block_tile references .

10.5.2.5 Coalesced groups

Threads (64 threads on CDNA and 32 threads on RDNA) in a warp cannot execute different instructions simultaneously, so conditional branches are executed serially within the warp. When threads encounter a conditional branch, they can diverge, resulting in some threads being disabled if they do not meet the condition to execute that branch. The active threads are referred to as coalesced, and coalesced group represents an active thread group within a warp.

Note

The NVIDIA GPU's independent thread scheduling presents the appearance that threads on different branches execute concurrently.

Warning

AMD GPUs do not support independent thread scheduling. Some CUDA application can rely on this feature and the ported HIP version on AMD GPUs can deadlock, when they try to make use of independent thread scheduling.

This group type also supports sub-wave level intrinsics.

```
class coalesced_group;
```

Constructed via:

```
coalesced_group active = coalesced_threads();
```

Note

`shfl()` functions support integer or float type.

The `thread_rank()`, `size()`, `cg_type()`, `is_valid()`, `sync()`, `meta_group_rank()`, `meta_group_size()`, `shfl()`, `shfl_down()`, `shfl_up()`, `ballot()`, `any()`, `all()`, `match_any()` and `match_all()` member functions are public of the `coalesced_group` class. For more information, see coalesced_group references .

10.5.3 Cooperative groups simple example

The difference to the original block model in the `reduce_sum` device function is the following.

Original Block

```
__device__ int reduce_sum(int *shared, int val) {

    // Thread ID
    const unsigned int thread_id = threadIdx.x;

    // Every iteration the number of active threads
    // halves, until we processed all values
    for(unsigned int i = blockDim.x / 2; i > 0; i /= 2) {
        // Store value in shared memory with thread ID
        shared[thread_id] = val;

        // Synchronize all threads
        __syncthreads();

        // Active thread sum up
        if(thread_id < i)
            val += shared[thread_id + i];

        // Synchronize all threads in the group
        __syncthreads();
    }

    // ...
}
```

Cooperative groups

```
__device__ int reduce_sum(thread_group g,
                          int *shared,
                          int val) {

    // Thread ID
    const unsigned int group_thread_id = g.thread_rank();

    // Every iteration the number of active threads
    // halves, until we processed all values
    for(unsigned int i = g.size() / 2; i > 0; i /= 2) {
        // Store value in shared memroy with thread ID
        shared[group_thread_id] = val;

        // Synchronize all threads in the group
        g.sync();

        // Active thread sum up
        if(group_thread_id < i)
            val += shared[group_thread_id + i];
```

(continues on next page)

(continued from previous page)

```
        // Synchronize all threads in the group
        g.sync();
    }

    // ...
}
```

The `reduce_sum()` function call and input data initialization difference to the original block model is the following.

Original Block

```
__global__ void sum_kernel(...) {

    // ...

    // Workspace array in shared memory
    __shared__ unsigned int workspace[2048];

    // ...

    // Perform reduction
    output = reduce_sum(workspace, input);

    // ...
}
```

Cooperative groups

```
__global__ void sum_kernel(...) {

    // ...

    // Workspace array in shared memory
    __shared__ unsigned int workspace[2048];

    // ...

    // Initialize the thread_block
    thread_block thread_block_group = this_thread_block();
    // Perform reduction
    output = reduce_sum(thread_block_group, workspace, input);

    // ...
}
```

At the device function, the input group type is the `thread_group`, which is the parent class of all the cooperative groups type. With this, you can write generic functions, which can work with any type of cooperative groups.

10.5.4 Synchronization

With each group type, the synchronization requires using the correct cooperative groups launch API.

Check the kernel launch capability

Thread-block

Do not need kernel launch validation.

Grid

Confirm the cooperative launch capability on the single AMD GPU:

```
int device               = 0;
int supports_coop_launch = 0;
// Check support
// Use hipDeviceAttributeCooperativeMultiDeviceLaunch when launching across multiple␣
↪devices
HIP_CHECK(hipGetDevice(&device));
HIP_CHECK(
    hipDeviceGetAttribute(&supports_coop_launch, hipDeviceAttributeCooperativeLaunch,␣
↪device));
if(!supports_coop_launch)
{
    std::cout << "Skipping, device " << device << " does not support cooperative␣
↪groups"
              << std::endl;
    return 0;
}
```

Multi-grid

Confirm the cooperative launch capability over multiple GPUs:

```
// Check support of cooperative groups
std::vector<int> deviceIDs;
for(int deviceID = 0; deviceID < device_count; deviceID++) {
#ifdef __HIP_PLATFORM_AMD__
    int supports_coop_launch = 0;
    HIP_CHECK(
        hipDeviceGetAttribute(
            &supports_coop_launch,
            hipDeviceAttributeCooperativeMultiDeviceLaunch,
            deviceID));
    if(!supports_coop_launch) {
        std::cout << "Skipping, device " << deviceID << " does not support␣
↪cooperative groups"
                  << std::endl;
    }
    else
#endif
    {
        std::cout << deviceID << std::endl;
        // Collect valid deviceIDs.
```

(continues on next page)

(continued from previous page)

```
        deviceIDs.push_back(deviceID);
    }
}
```

Kernel launch

Thread-block

You can access the new block representation using the original kernel launch methods.

```
void* params[] = {&d_vector, &d_block_reduced, &d_partition_reduced};
// Launching kernel from host.
HIP_CHECK(hipLaunchKernelGGL(vector_reduce_kernel<partition_size>,
                             dim3(num_blocks),
                             dim3(threads_per_block),
                             0,
                             hipStreamDefault,
                             &d_vector,
                             &d_block_reduced,
                             &d_partition_reduced));
```

Grid

Launch the cooperative kernel on a single GPU:

```
void* params[] = {};
// Launching kernel from host.
HIP_CHECK(hipLaunchCooperativeKernel(vector_reduce_kernel<partition_size>,
                                     dim3(num_blocks),
                                     dim3(threads_per_block),
                                     0,
                                     0,
                                     hipStreamDefault));
```

Multi-grid

Launch the cooperative kernel over multiple GPUs:

```
hipLaunchParams *launchParamsList = (hipLaunchParams*)malloc(sizeof(hipLaunchParams)␣
↪* deviceIDs.size());
for(int deviceID : deviceIDs) {

    // Set device
    HIP_CHECK(hipSetDevice(deviceID));

    // Create stream
    hipStream_t stream;
    HIP_CHECK(hipStreamCreate(&stream));

    // Parameters
    void* params[] = {&(d_vector[deviceID]), &(d_block_reduced[deviceID]), &(d_
↪partition_reduced[deviceID])};
```

(continues on next page)

(continued from previous page)

```
    // Set launchParams
    launchParamsList[deviceID].func = (void*)vector_reduce_kernel<partition_size>;
    launchParamsList[deviceID].gridDim = dim3(1);
    launchParamsList[deviceID].blockDim = dim3(threads_per_block);
    launchParamsList[deviceID].sharedMem = 0;
    launchParamsList[deviceID].stream = stream;
    launchParamsList[deviceID].args = params;
}

HIP_CHECK(hipLaunchCooperativeKernelMultiDevice(launchParamsList,
                                                (int)deviceIDs.size(),
                                                ␣
↪hipCooperativeLaunchMultiDeviceNoPreSync));
```

Device side synchronization

Thread-block

The device side code of the thread_block synchronization over single GPUs:

```
thread_block g = this_thread_block();
g.sync();
```

Grid

The device side code of the grid synchronization over single GPUs:

```
grid_group grid = this_grid();
grid.sync();
```

Multi-grid

The device side code of the multi-grid synchronization over multiple GPUs:

```
multi_grid_group multi_grid = this_multi_grid();
multi_grid.sync();
```

10.5.5 Unsupported NVIDIA CUDA features

HIP doesn't support the following NVIDIA CUDA optional headers:

- `cooperative_groups/memcpy_async.h`
- `cooperative_groups/reduce.h`
- `cooperative_groups/scan.h`

HIP doesn't support the following CUDA class in `cooperative_groups` namespace:

- `cluster_group`

HIP doesn't support the following CUDA functions/operators in `cooperative_groups` namespace:

- `synchronize`
- `memcpy_async`

- `wait` and `wait_prior`
- `barrier_arrive` and `barrier_wait`
- `invoke_one` and `invoke_one_broadcast`
- `reduce`
- `reduce_update_async` and `reduce_store_async`
- Reduce operators `plus` , `less` , `greater` , `bit_and` , `bit_xor` and `bit_or`
- `inclusive_scan` and `exclusive_scan`

10.6 HIP graphs

HIP graphs are an alternative way of executing tasks on a GPU that can provide performance benefits over launching kernels using the standard method via streams. A HIP graph is made up of nodes and edges. The nodes of a HIP graph represent the operations performed, while the edges mark dependencies between those operations.

Hint

The HIP Graph API tutorial demonstrates how to use HIP graphs in a real-world application.

The nodes can be one of the following:

- empty nodes
- nested graphs
- kernel launches
- host-side function calls
- HIP memory functions (copy, memset, …)
- HIP events
- signalling or waiting on external semaphores

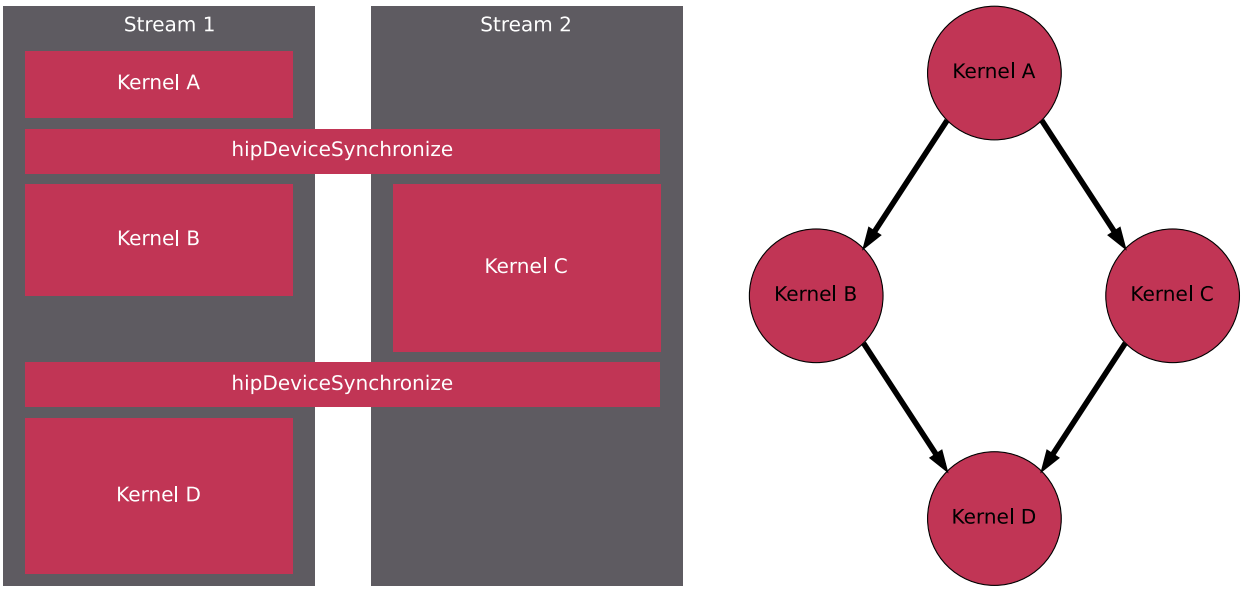
Note

The available node types are specified by `hipGraphNodeType`.

The following figure visualizes the concept of graphs, compared to using streams.

The standard method of launching kernels incurs a small overhead for each iteration of the operation involved. That overhead is negligible, when the kernel is launched directly with the HIP C/C++ API, but depending on the framework used, there can be several levels of redirection, until the actual kernel is launched by the HIP runtime, leading to significant overhead. Especially for some AI frameworks, a GPU kernel might run faster than the time it takes for the framework to set up and launch the kernel, and so the overhead of repeatedly launching kernels can have a significant impact on performance.

HIP graphs are designed to address this issue, by predefining the HIP API calls and their dependencies with a graph, and performing most of the initialization beforehand. Launching a graph only requires a single call, after which the HIP runtime takes care of executing the operations within the graph. Graphs can provide additional performance benefits, by enabling optimizations that are only possible when knowing the dependencies between the operations.



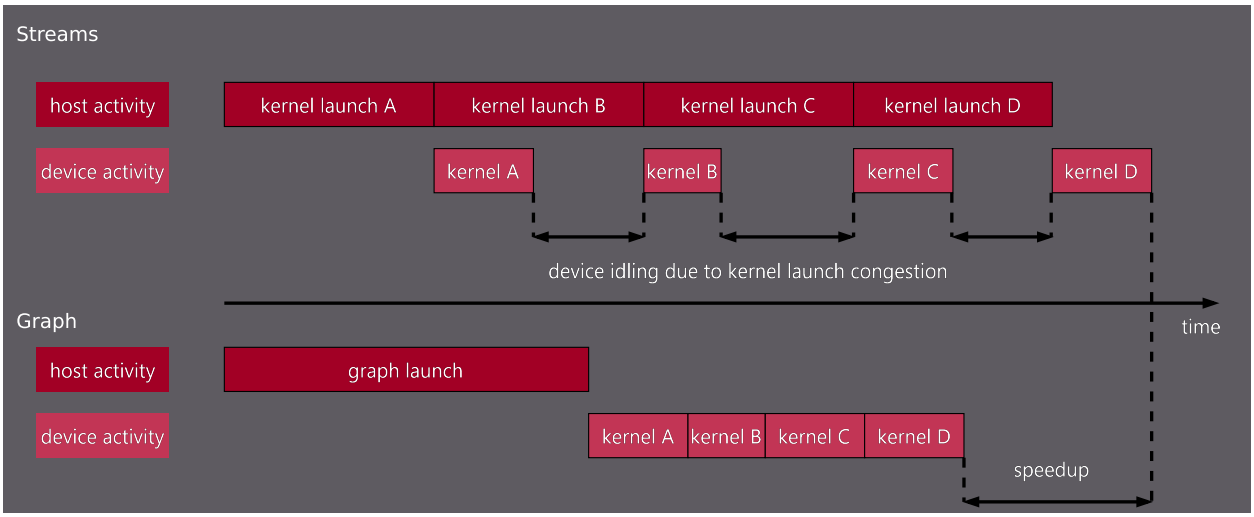


Fig. 10: Qualitative presentation of the execution time of many short-running kernels when launched using HIP stream versus HIP graph. This does not include the time needed to set up the graph.

10.6.1 Using HIP graphs

There are two different ways of creating graphs: Capturing kernel launches from a stream, or explicitly creating graphs. The difference between the two approaches is explained later in this chapter.

The general flow for using HIP graphs includes the following steps.

1. Create a `hipGraph_t` graph template using one of the two approaches described in this chapter
2. Create a `hipGraphExec_t` executable instance of the graph template using `hipGraphInstantiate()`
3. Use `hipGraphLaunch()` to launch the executable graph to a stream
4. After execution completes free and destroy graph resources

The first two steps are the initial setup and only need to be executed once. First step is the definition of the operations (nodes) and the dependencies (edges) between them. The second step is the instantiation of the graph. This takes care of validating and initializing the graph, to reduce the overhead when executing the graph. The third step is the execution of the graph, which takes care of launching all the kernels and executing the operations while respecting their dependencies and necessary synchronizations as specified.

Because HIP graphs require some setup and initialization overhead before their first execution, graphs only provide a benefit for workloads that require many iterations to complete.

In both methods the `hipGraph_t` template for a graph is used to define the graph. In order to actually launch a graph, the template needs to be instantiated using `hipGraphInstantiate()`, which results in an executable graph of type `hipGraphExec_t`. This executable graph can then be launched with `hipGraphLaunch()`, replaying the operations within the graph. Note, that launching graphs is fundamentally no different to executing other HIP functions on a stream, except for the fact, that scheduling the operations within the graph encompasses less overhead and can enable some optimizations, but they still need to be associated with a stream for execution.

10.6.1.1 Memory management

Memory that is used by operations in graphs can either be pre-allocated or managed within the graph. Graphs can contain nodes that take care of allocating memory on the device or copying memory between the host and the device. Whether you want to pre-allocate the memory or manage it within the graph depends on the use-case. If the graph is executed in a tight loop the performance is usually better when the memory is preallocated, so that it does not need to be reallocated in every iteration.

The same rules as for normal memory allocations apply for memory allocated and freed by nodes, meaning that the nodes that access memory allocated in a graph must be ordered after allocation and before freeing.

Memory management within the graph enables the runtime to take care of memory reuse and optimizations. The lifetime of memory managed in a graph begins when the execution reaches the node allocating the memory, and ends when either reaching the corresponding free node within the graph, or after graph execution when a corresponding `hipFreeAsync()` or `hipFree()` call is reached. The memory can also be freed with a free node in a different graph that is associated with the same memory address.

Unlike device memory that is not associated with a graph, this does not necessarily mean that the freed memory is returned back to the operating system immediately. Graphs can retain a memory pool for quickly reusing memory within the graph. This can be especially useful when memory is freed and reallocated later on within a graph, as that memory doesn't have to be requested from the operating system. It also potentially reduces the total memory footprint of the graph, by reusing the same memory.

The amount of memory allocated for graph memory pools on a specific device can be queried using `hipDeviceGetGraphMemAttribute()`. In order to return the freed memory `hipDeviceGraphMemTrim()` can be used. This will return any memory that is not in active use by graphs.

These memory allocations can also be set up to allow access from multiple GPUs, just like normal allocations. HIP then takes care of allocating and mapping the memory to the GPUs. When capturing a graph from a stream, the node sets the accessibility according to `hipMemPoolSetAccess()` at the time of capturing.

10.6.2 Capture graphs from a stream

The easy way to integrate HIP graphs into already existing code is to use `hipStreamBeginCapture()` and `hipStreamEndCapture()` to obtain a `hipGraph_t` graph template that includes the captured operations.

When starting to capture operations for a graph using `hipStreamBeginCapture()`, the operations assigned to the stream are captured into a graph instead of being executed. The associated graph is returned when calling `hipStreamEndCapture()`, which also stops capturing operations. In order to capture to an already existing graph use `hipStreamBeginCaptureToGraph()`.

The functions assigned to the capturing stream are not executed, but instead are captured and defined as nodes in the graph, to be run when the instantiated graph is launched.

Functions must be associated with a stream in order to be captured. This means that non-HIP API functions are not captured by default, but are executed as standard functions when encountered and not added to the graph. In order to assign host functions to a stream use `hipLaunchHostFunc()`, as shown in the following code example. They will then be captured and defined as a host node in the resulting graph, and won't be executed when encountered.

Synchronous HIP API calls that are implicitly assigned to the default stream are not permitted while capturing a stream and will return an error. This is because they implicitly synchronize and cause a dependency that can not be captured within the stream. This includes functions like `hipMalloc()`, `hipMemcpy()` and `hipFree()`. In order to capture these to the stream, replace them with the corresponding asynchronous calls like `hipMallocAsync()`, `hipMemcpyAsync()` or `hipFreeAsync()`.

The general flow for using stream capture to create a graph template is:

1. Create a stream from which to capture the operations
2. Call `hipStreamBeginCapture()` before the first operation to be captured
3. Call `hipStreamEndCapture()` after the last operation to be captured
    1. Define a `hipGraph_t` graph template to which `hipStreamEndCapture()` passes the captured graph

The following code is an example of how to use the HIP graph API to capture a graph from a stream.

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

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

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

__global__ void kernelA(double* arrayA, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
```

(continues on next page)

(continued from previous page)

```
    if(x < size)
    {
        arrayA[x] *= 2.0;
    }
}

__global__ void kernelB(int* arrayB, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayB[x] = 3;
    }
}

__global__ void kernelC(double* arrayA, const int* arrayB, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayA[x] += arrayB[x];
    }
}

struct set_vector_args
{
    std::vector<double>& h_array;
    double value;
};

void set_vector(void* args)
{
    set_vector_args h_args{*(reinterpret_cast<set_vector_args*>(args))};

    std::vector<double>& vec{h_args.h_array};
    vec.assign(vec.size(), h_args.value);
}

int main()
{
    constexpr int numOfBlocks = 1024;
    constexpr int threadsPerBlock = 1024;
    constexpr std::size_t arraySize = 1U << 20;

    // This example assumes that kernelA operates on data that needs to be␣
↪initialized on
    // and copied from the host, while kernelB initializes the array that is passed␣
↪to it.
    // Both arrays are then used as input to kernelC, where arrayA is also used as
   //  output, that is copied back to the host, while arrayB is only read from and␣
↪not modified.
```

(continues on next page)

(continued from previous page)

```
    double* d_arrayA;
    int* d_arrayB;
    std::vector<double> h_array(arraySize);
    constexpr double initValue = 2.0;

    hipStream_t captureStream;
    HIP_CHECK(hipStreamCreate(&captureStream));

    // Start capturing the operations assigned to the stream
    HIP_CHECK(hipStreamBeginCapture(captureStream, hipStreamCaptureModeGlobal));

    // hipMallocAsync and hipMemcpyAsync are needed, to be able to assign it to a␣
↪stream
    HIP_CHECK(hipMallocAsync(reinterpret_cast<void**>(&d_arrayA),␣
↪arraySize*sizeof(double), captureStream));
    HIP_CHECK(hipMallocAsync(reinterpret_cast<void**>(&d_arrayB),␣
↪arraySize*sizeof(int), captureStream));

    // Assign host function to the stream
    // Needs a custom struct to pass the arguments
    set_vector_args args{h_array, initValue};
    HIP_CHECK(hipLaunchHostFunc(captureStream, set_vector, &args));

    HIP_CHECK(hipMemcpyAsync(d_arrayA, h_array.data(), arraySize*sizeof(double),␣
↪hipMemcpyHostToDevice, captureStream));

    kernelA<<<numOfBlocks, threadsPerBlock, 0, captureStream>>>(d_arrayA, arraySize);
    kernelB<<<numOfBlocks, threadsPerBlock, 0, captureStream>>>(d_arrayB, arraySize);
    kernelC<<<numOfBlocks, threadsPerBlock, 0, captureStream>>>(d_arrayA, d_arrayB,␣
↪arraySize);

    HIP_CHECK(hipMemcpyAsync(h_array.data(), d_arrayA, arraySize*sizeof(*d_arrayA),␣
↪hipMemcpyDeviceToHost, captureStream));

    HIP_CHECK(hipFreeAsync(d_arrayA, captureStream));
    HIP_CHECK(hipFreeAsync(d_arrayB, captureStream));

    // Stop capturing
    hipGraph_t graph;
    HIP_CHECK(hipStreamEndCapture(captureStream, &graph));

    // Create an executable graph from the captured graph
    hipGraphExec_t graphExec;
    HIP_CHECK(hipGraphInstantiate(&graphExec, graph, nullptr, nullptr, 0));

    // The graph template can be deleted after the instantiation if it's not needed␣
↪for later use
    HIP_CHECK(hipGraphDestroy(graph));

    // Actually launch the graph. The stream does not have
    // to be the same as the one used for capturing.
    HIP_CHECK(hipGraphLaunch(graphExec, captureStream));
```

(continues on next page)

(continued from previous page)

```
    HIP_CHECK(hipStreamSynchronize(captureStream));

    // Verify results
    constexpr double expected = initValue * 2.0 + 3;
    bool passed = true;
    for(std::size_t i = 0; i < arraySize; ++i)
    {
        if(h_array[i] != expected)
        {
            passed = false;
            std::cerr << "Validation failed! Expected " << expected << " got " << h_
↪array[0] << std::endl;
            break;
        }
    }

    if(passed)
    {
        std::cerr << "Validation passed." << std::endl;
    }

    // Free graph and stream resources after usage
    HIP_CHECK(hipGraphExecDestroy(graphExec));
    HIP_CHECK(hipStreamDestroy(captureStream));

    return EXIT_SUCCESS;
}
```

10.6.3 Explicit graph creation

Graphs can also be created directly using the HIP graph API, giving more fine-grained control over the graph. In this case, the graph nodes are created explicitly, together with their parameters and dependencies, which specify the edges of the graph, thereby forming the graph structure.

The nodes are represented by the generic `hipGraphNode_t` type. The actual node type is implicitly defined by the specific function used to add the node to the graph, for example `hipGraphAddKernelNode()` See the HIP graph API documentation for the available functions, they are of type `hipGraphAdd{Type}Node`. Each type of node also has a predefined set of parameters depending on the operation, for example `hipKernelNodeParams` for a kernel launch. See the documentation for the general hipGraphNodeParams type for a list of available parameter types and their members.

The general flow for explicitly creating a graph is usually:

1. Create a graph `hipGraph_t`
2. Create the nodes and their parameters and add them to the graph
    1. Define a `hipGraphNode_t`
    2. Define the parameter struct for the desired operation, by explicitly setting the appropriate struct's members.
    3. Use the appropriate `hipGraphAdd{Type}Node` function to add the node to the graph.
        1. The dependencies can be defined when adding the node to the graph, or afterwards by using `hipGraphAddDependencies()`

The following code example demonstrates how to explicitly create nodes in order to create a graph.

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

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

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

__global__ void kernelA(double* arrayA, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayA[x] *= 2.0;
    }
}

__global__ void kernelB(int* arrayB, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayB[x] = 3;
    }
}

__global__ void kernelC(double* arrayA, const int* arrayB, std::size_t size)
{
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;
    if(x < size)
    {
        arrayA[x] += arrayB[x];
    }
}

struct set_vector_args
{
    std::vector<double>& h_array;
    double value;
};
```

(continues on next page)

(continued from previous page)

```
void set_vector(void* args)
{
    set_vector_args h_args{*(reinterpret_cast<set_vector_args*>(args))};

    std::vector<double>& vec{h_args.h_array};
    vec.assign(vec.size(), h_args.value);
}

int main()
{
    constexpr int numOfBlocks = 1024;
    constexpr int threadsPerBlock = 1024;
    std::size_t arraySize = 1U << 20;

    // The pointers to the device memory don't need to be declared here,
    // they are contained within the hipMemAllocNodeParams as the dptr member
    std::vector<double> h_array(arraySize);
    constexpr double initValue = 2.0;

    // Create graph an empty graph
    hipGraph_t graph;
    HIP_CHECK(hipGraphCreate(&graph, 0));

    // Parameters to allocate arrays
    hipMemAllocNodeParams allocArrayAParams{};
    allocArrayAParams.poolProps.allocType = hipMemAllocationTypePinned;
    allocArrayAParams.poolProps.location.type = hipMemLocationTypeDevice;
    allocArrayAParams.poolProps.location.id = 0; // GPU on which memory resides
    allocArrayAParams.bytesize = arraySize * sizeof(double);

    hipMemAllocNodeParams allocArrayBParams{};
    allocArrayBParams.poolProps.allocType = hipMemAllocationTypePinned;
    allocArrayBParams.poolProps.location.type = hipMemLocationTypeDevice;
    allocArrayBParams.poolProps.location.id = 0; // GPU on which memory resides
    allocArrayBParams.bytesize = arraySize * sizeof(int);

    // Add the allocation nodes to the graph. They don't have any dependencies
    hipGraphNode_t allocNodeA, allocNodeB;
    HIP_CHECK(hipGraphAddMemAllocNode(&allocNodeA, graph, nullptr, 0, &
↪allocArrayAParams));
    HIP_CHECK(hipGraphAddMemAllocNode(&allocNodeB, graph, nullptr, 0, &
↪allocArrayBParams));

    // Parameters for the host function
    // Needs custom struct to pass the arguments
    set_vector_args args{h_array, initValue};
    hipHostNodeParams hostParams{};
    hostParams.fn = set_vector;
    hostParams.userData = static_cast<void*>(&args);

    // Add the host node that initializes the host array. It also doesn't have any␣
↪dependencies
```

(continues on next page)

(continued from previous page)

```
    hipGraphNode_t hostNode;
    HIP_CHECK(hipGraphAddHostNode(&hostNode, graph, nullptr, 0, &hostParams));

    // Add memory copy node, that copies the initialized host array to the device.
    // It has to wait for the host array to be initialized and the device memory to␣
↪be allocated
    hipGraphNode_t cpyNodeDependencies[] = {allocNodeA, hostNode};
    hipGraphNode_t cpyToDevNode;
    HIP_CHECK(hipGraphAddMemcpyNode1D(&cpyToDevNode, graph, cpyNodeDependencies, 2,␣
↪allocArrayAParams.dptr, h_array.data(), arraySize * sizeof(double),␣
↪hipMemcpyHostToDevice));

    // Parameters for kernelA
    hipKernelNodeParams kernelAParams;
    void* kernelAArgs[] = {&allocArrayAParams.dptr, static_cast<void*>(&arraySize)};
    kernelAParams.func = reinterpret_cast<void*>(kernelA);
    kernelAParams.gridDim = numOfBlocks;
    kernelAParams.blockDim = threadsPerBlock;
    kernelAParams.sharedMemBytes = 0;
    kernelAParams.kernelParams = kernelAArgs;
    kernelAParams.extra = nullptr;

    // Add the node for kernelA. It has to wait for the memory copy to finish, as it␣
↪depends on the values from the host array.
    hipGraphNode_t kernelANode;
    HIP_CHECK(hipGraphAddKernelNode(&kernelANode, graph, &cpyToDevNode, 1, &
↪kernelAParams));

    // Parameters for kernelB
    hipKernelNodeParams kernelBParams;
    void* kernelBArgs[] = {&allocArrayBParams.dptr, static_cast<void*>(&arraySize)};
    kernelBParams.func = reinterpret_cast<void*>(kernelB);
    kernelBParams.gridDim = numOfBlocks;
    kernelBParams.blockDim = threadsPerBlock;
    kernelBParams.sharedMemBytes = 0;
    kernelBParams.kernelParams = kernelBArgs;
    kernelBParams.extra = nullptr;

    //  Add the node for kernelB. It only has to wait for the memory to be allocated,␣
↪as it initializes the array.
    hipGraphNode_t kernelBNode;
    HIP_CHECK(hipGraphAddKernelNode(&kernelBNode, graph, &allocNodeB, 1, &
↪kernelBParams));

    // Parameters for kernelC
    hipKernelNodeParams kernelCParams;
    void* kernelCArgs[] = {&allocArrayAParams.dptr, &allocArrayBParams.dptr, static_
↪cast<void*>(&arraySize)};
    kernelCParams.func = reinterpret_cast<void*>(kernelC);
    kernelCParams.gridDim = numOfBlocks;
    kernelCParams.blockDim = threadsPerBlock;
    kernelCParams.sharedMemBytes = 0;
```

(continues on next page)

(continued from previous page)

```
    kernelCParams.kernelParams = kernelCArgs;
    kernelCParams.extra = nullptr;

    // Add the node for kernelC. It has to wait on both kernelA and kernelB to finish,
↪ as it depends on their results.
    hipGraphNode_t kernelCNode;
    hipGraphNode_t kernelCDependencies[] = {kernelANode, kernelBNode};
    HIP_CHECK(hipGraphAddKernelNode(&kernelCNode, graph, kernelCDependencies, 2, &
↪kernelCParams));

    // Copy the results back to the host. Has to wait for kernelC to finish.
    hipGraphNode_t cpyToHostNode;
    HIP_CHECK(hipGraphAddMemcpyNode1D(&cpyToHostNode, graph, &kernelCNode, 1, h_array.
↪data(), allocArrayAParams.dptr, arraySize * sizeof(double), hipMemcpyDeviceToHost));

    // Free array of allocNodeA. It needs to wait for the copy to finish, as kernelC␣
↪stores its results in it.
    hipGraphNode_t freeNodeA;
    HIP_CHECK(hipGraphAddMemFreeNode(&freeNodeA, graph, &cpyToHostNode, 1,␣
↪allocArrayAParams.dptr));
    // Free array of allocNodeB. It only needs to wait for kernelC to finish, as it␣
↪is not written back to the host.
    hipGraphNode_t freeNodeB;
    HIP_CHECK(hipGraphAddMemFreeNode(&freeNodeB, graph, &kernelCNode, 1,␣
↪allocArrayBParams.dptr));

    // Instantiate the graph in order to execute it
    hipGraphExec_t graphExec;
    HIP_CHECK(hipGraphInstantiate(&graphExec, graph, nullptr, nullptr, 0));

    // The graph can be freed after the instantiation if it's not needed for other␣
↪purposes
    HIP_CHECK(hipGraphDestroy(graph));

    // Actually launch the graph
    hipStream_t graphStream;
    HIP_CHECK(hipStreamCreate(&graphStream));
    HIP_CHECK(hipGraphLaunch(graphExec, graphStream));

    HIP_CHECK(hipStreamSynchronize(graphStream));

    // Verify results
    constexpr double expected = initValue * 2.0 + 3;
    bool passed = true;
    for(std::size_t i = 0; i < arraySize; ++i)
    {
        if(h_array[i] != expected)
        {
            passed = false;
            std::cerr << "Validation failed! Expected " << expected << " got " << h_
↪array[0] << std::endl;
            break;
```

(continues on next page)

(continued from previous page)

```
        }
    }

    if(passed)
    {
        std::cerr << "Validation passed." << std::endl;
    }

    HIP_CHECK(hipGraphExecDestroy(graphExec));
    HIP_CHECK(hipStreamDestroy(graphStream));

    return EXIT_SUCCESS;
}
```

10.7 Call stack

The call stack is a data structure for managing function calls, by saving the state of the current function. Each time a function is called, a new call frame is added to the top of the stack, containing information such as local variables, return addresses and function parameters. When the function execution completes, the frame is removed from the stack and loaded back into the corresponding registers. This concept allows the program to return to the calling function and continue execution from where it left off.

The call stack for each thread must track its function calls, local variables, and return addresses. However, in GPU programming, the memory required to store the call stack increases due to the parallelism inherent to the GPUs. NVIDIA and AMD GPUs use different approaches. NVIDIA GPUs have the independent thread scheduling feature where each thread has its own call stack and effective program counter. On AMD GPUs threads are grouped; each warp has its own call stack and program counter. Warps are described and explained in the *Hierarchical thread model*

If a thread or warp exceeds its stack size, a stack overflow occurs, causing kernel failure. This can be detected using debuggers.

10.7.1 Call stack management with HIP

You can adjust the call stack size as shown in the following example, allowing fine-tuning based on specific kernel requirements. This helps prevent stack overflow errors by ensuring sufficient stack memory is allocated.

```
#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;    \
```

(continues on next page)

(continued from previous page)

```
    }                                           \
}

int main()
{
    std::size_t stackSize;
    HIP_CHECK(hipDeviceGetLimit(&stackSize, hipLimitStackSize));
    std::cout << "Default stack size: " << stackSize << " bytes" << std::endl;

    // Set a new stack size
    std::size_t newStackSize = 1024 * 8; // 8 KiB
    HIP_CHECK(hipDeviceSetLimit(hipLimitStackSize, newStackSize));

    HIP_CHECK(hipDeviceGetLimit(&stackSize, hipLimitStackSize));
    std::cout << "Updated stack size: " << stackSize << " bytes" << std::endl;

    return EXIT_SUCCESS;
}
```

Depending on the GPU model, at full occupancy, it can consume a significant amount of memory. For instance, an MI300X with 304 compute units (CU) and up to 2048 threads per CU could use $304 \cdot 2048 \cdot 1024$ bytes = 608 MiB for the call stack by default.

10.7.1.1 Handling recursion and deep function calls

Similar to CPU programming, recursive functions and deeply nested function calls are supported. However, developers must ensure that these functions do not exceed the available stack memory, considering the huge amount of memory needed for the call stack due to the GPUs inherent parallelism. This can be achieved by increasing stack size or optimizing code to reduce stack usage. To detect stack overflow add proper error handling or use debugging tools.

```
#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;      \
    }                                          \
}

__device__ unsigned long long fibonacci(unsigned long long n)
{
    if (n == 0 || n == 1)
```

(continues on next page)

(continued from previous page)

```
    {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

__global__ void kernel(unsigned long long n)
{
    unsigned long long result = fibonacci(n);
    const std::size_t x = threadIdx.x + blockDim.x * blockIdx.x;

    if (x == 0)
        printf("%llu! = %llu \n", n, result);
}

int main()
{
    kernel<<<1, 1>>>(10);
    HIP_CHECK(hipDeviceSynchronize());

    // With -O0 optimization option hit the stack limit
    // kernel<<<1, 256>>>(2048);
    // HIP_CHECK(hipDeviceSynchronize());

    return EXIT_SUCCESS;
}
```

10.8 OpenGL interoperability

The HIP–OpenGL interoperation involves mapping OpenGL resources, such as buffers and textures, for HIP to interact with OpenGL. This mapping process enables HIP to utilize these resources directly, bypassing the need for costly data transfers between the CPU and GPU. This capability is useful in applications that require both intensive GPU computation and real-time visualization.

The graphics resources must be registered using functions like `hipGraphicsGLRegisterBuffer()` or `hipGraphicsGLRegisterImage()` then they can be mapped to HIP with `hipGraphicsMapResources()` function.

After mapping, the `hipGraphicsResourceGetMappedPointer()` or `hipGraphicsSubResourceGetMappedArray()` functions used to retrieve a device pointer to the mapped resource, which can then be used in HIP kernels.

Unmapping resources with `hipGraphicsUnmapResources()` after computations ensure proper resource management.

10.8.1 Example

ROCm examples have a HIP–OpenGL interoperation example, where a simple HIP kernel is used to simulate a sine wave and rendered to a window as a grid of triangles using OpenGL. For a working example, there are multiple initialization steps needed like creating and opening a window, initializing OpenGL or selecting the OpenGL-capable device. After the initialization in the example, the kernel simulates the sinewave and updates the window's framebuffer in a cycle until the window is closed.

Note

The more recent OpenGL functions are loaded with OpenGL loader, as these are not loaded by default on all platforms. The use of a custom loader is shown in the following example

```
    // Make GLFW use a custom loader - we need this for the more recent OpenGL␣
↪functions,
    // as these are not loaded by default on all platforms.
    if(!gladLoadGLLoader(reinterpret_cast<GLADloadproc>(glfwGetProcAddress)))
    {
        std::cerr << "Failed to load OpenGL function pointers" << std::endl;
        return error_exit_code;
    }
```

The OpenGL buffer is imported to HIP in the following way:

```
        // Import the OpenGL height buffer into a HIP graphics resource.
        HIP_CHECK(hipGraphicsGLRegisterBuffer(
            &this->hip_height_buffer,
            renderer.height_buffer,
            // We are going to write to this buffer from HIP,
            // but we do not need to read from it.
            // As an optimization we can pass hipGraphicsRegisterFlagsWriteDiscard,
            // so that the driver knows that we do not need the old values of
            // the buffer.
            hipGraphicsRegisterFlagsWriteDiscard));

        // After importing the OpenGL height buffer into HIP, map it into HIP memory␣
↪so that we can use it.
        HIP_CHECK(hipGraphicsMapResources(1, &this->hip_height_buffer, this->hip_
↪stream));

        // Fetch the device pointer that points to the OpenGL buffer's memory.
        // This function also fetches the size of the buffer. We already know it, but␣
↪we still need to pass
        // a valid pointer to hipGraphicsResourceGetMappedPointer.
        size_t size;
        HIP_CHECK(
            hipGraphicsResourceGetMappedPointer(reinterpret_cast<void**>(&this->hip_
↪height_ptr),
                                                &size,
                                                this->hip_height_buffer));
```

The imported pointer is manipulated in the sinewave kernel as shown in the following example:

```
/// \brief The main HIP kernel for this example - computes a simple sine wave over a
/// 2-dimensional grid of points.
/// \param height_map - the grid of points to compute a sine wave for. It is expected␣
↪to be
///   a \p grid_width by \p grid_height array packed into memory.(y on the inner␣
↪axis).
/// \param time - The current time relative to the start of the program.
__global__ void sinewave_kernel(float* height_map, const float time)
```

(continues on next page)

(continued from previous page)

```
{
    const float        freq = 10.f;
    const unsigned int x    = blockIdx.x * blockDim.x + threadIdx.x;
    const unsigned int y    = blockIdx.y * blockDim.y + threadIdx.y;
    const float        u    = (2.f * x) / grid_width - 1.f;
    const float        v    = (2.f * y) / grid_height - 1.f;

    if(x < grid_width && y < grid_height)
    {
        height_map[x * grid_width + y] = sinf(u * freq + time) * cosf(v * freq +␣
↪time);
    }
}
```

```
        // The tile size to be used for each block of the computation. A tile is
        // tile_size by tile_size threads in this case, since we are invoking the
        // computation over a 2D-grid.
        constexpr size_t tile_size = 8;

        // Launch the HIP kernel to advance the simulation.
        sinewave_kernel<<<dim3(ceiling_div(grid_width, tile_size),
                               ceiling_div(grid_height, tile_size)),
                          dim3(tile_size, tile_size),
                          0,
                          this->hip_stream>>>(this->hip_height_ptr, time);

        // Check that no errors occured while launching the kernel.
        HIP_CHECK(hipGetLastError());
```

The HIP graphics resource that is imported from the OpenGL buffer and is not needed anymore should be unmapped and unregistered as shown in the following way:

```
        HIP_CHECK(hipStreamSynchronize(this->hip_stream));
        HIP_CHECK(hipGraphicsUnmapResources(1, &this->hip_height_buffer, this->hip_
↪stream));
        HIP_CHECK(hipGraphicsUnregisterResource(this->hip_height_buffer));
        HIP_CHECK(hipStreamDestroy(this->hip_stream));
```

10.9 External resource interoperability

This feature allows HIP to work with resources – like memory and semaphores – created by other APIs. This means resources can be used from APIs like CUDA, OpenCL and Vulkan within HIP, making it easier to integrate HIP into existing projects.

To use external resources in HIP, you typically follow these steps:

- Import resources from other APIs using HIP provided functions
- Use external resources as if they were created in HIP
- Destroy the HIP resource object to clean up

10.9.1 Semaphore functions

Semaphore functions are essential for synchronization in parallel computing. These functions facilitate communication and coordination between different parts of a program or between different programs. By managing semaphores, tasks are executed in the correct order, and resources are utilized effectively. Semaphore functions ensure smooth operation, preventing conflicts and maintaining the integrity of processes; upholding the integrity and performance of concurrent processes.

External semaphore functions can be used in HIP as described in External resource interoperability.

10.9.2 Memory functions

HIP external memory functions focus on the efficient sharing and management of memory resources. These functions enable importing memory created by external systems, enabling the HIP program to use this memory seamlessly. Memory functions include mapping memory for effective use and ensuring proper cleanup to prevent resource leaks. This is critical for performance, particularly in applications handling large datasets or complex structures such as textures in graphics. Proper memory management ensures stability and efficient resource utilization.

10.9.3 Example

ROCm examples include a HIP–Vulkan interoperation example demonstrates how to perform interoperation between HIP and Vulkan.

In this example, a simple HIP kernel is used to compute a sine wave, which is then rendered to a window as a graphical output using Vulkan. The process requires several initialization steps, such as setting up a HIP context, creating a Vulkan instance, and configuring the GPU device and queue. After these initial steps, the kernel executes the sine wave computation, and Vulkan continuously updates the window framebuffer to display the computed data until the window is closed.

The following code converts a Vulkan memory handle to its equivalent HIP handle. The input `VkDeviceMemory` and the created HIP memory represents the same physical area of GPU memory, through the handles of each respective API. Writing to the buffer in one API will allow us to read the results through the other. Note that access to the buffer should be synchronized between the APIs, for example using queue syncs or semaphores.

```
    // Prepare the HIP external semaphore descriptor with the platform-specific
    // handle type that we wish to import. This value should correspond to the
    // handleTypes field set in VkExportMemoryAllocateInfoKHR while creating the
    // Vulkan buffer.
    hipExternalMemoryHandleDesc desc = {};
    desc.size                        = size;

    // Export the Vulkan buffer handle to a platform-specific native handle, depending
    // on the current platform: On Windows the buffer is converted to a HANDLE, and␣
↪on Linux
    // to a file descriptor representing the driver's GPU handle to the memory.
    // This native handle is then passed to the HIP external memory descriptor so␣
↪that it
    // may be imported.
#ifdef _WIN64
    desc.type = hipExternalMemoryHandleTypeOpaqueWin32Kmt;

    VkMemoryGetWin32HandleInfoKHR get_handle_info = {};
    get_handle_info.sType      = VK_STRUCTURE_TYPE_MEMORY_GET_WIN32_HANDLE_INFO_KHR;
    get_handle_info.memory     = memory;
    get_handle_info.handleType = VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT_
```

(continues on next page)

(continued from previous page)

```
↪KHR;

    VK_CHECK(
        ctx.vkd->get_memory_win32_handle(ctx.dev, &get_handle_info, &desc.handle.
↪win32.handle));
#else
    desc.type = hipExternalMemoryHandleTypeOpaqueFd;

    VkMemoryGetFdInfoKHR get_fd_info = {};
    get_fd_info.sType                = VK_STRUCTURE_TYPE_MEMORY_GET_FD_INFO_KHR;
    get_fd_info.memory               = memory;
    get_fd_info.handleType           = VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT_
↪KHR;

    VK_CHECK(ctx.vkd->get_memory_fd(ctx.dev, &get_fd_info, &desc.handle.fd));
#endif

    // Import the native memory handle to HIP to create an external memory.
    hipExternalMemory_t hip_memory;
    HIP_CHECK(hipImportExternalMemory(&hip_memory, &desc));
    return hip_memory;
```

The Vulkan semaphore is converted to HIP semaphore shown in the following example. Signaling on the semaphore in one API will allow the other API to wait on it, which is how we can guarantee synchronized access to resources in a cross-API manner.

```
    // Prepare the HIP external semaphore descriptor with the platform-specific␣
↪handle type
    // that we wish to import. This value should correspond to the handleTypes field␣
↪set in
    // the VkExportSemaphoreCreateInfoKHR structure that was passed to Vulkan when␣
↪creating
    // the semaphore.
    hipExternalSemaphoreHandleDesc desc = {};

    // Export the Vulkan semaphore to a platform-specific handle depending on the␣
↪current
    // platform: On Windows, we convert the semaphore into a HANDLE, and on Linux it␣
↪is
    // converted to a file descriptor.
    // This native handle is then passed to the HIP external semaphore descriptor.
#ifdef _WIN64
    desc.type = hipExternalSemaphoreHandleTypeOpaqueWin32;

    VkSemaphoreGetWin32HandleInfoKHR get_handle_info = {};
    get_handle_info.sType      = VK_STRUCTURE_TYPE_SEMAPHORE_GET_WIN32_HANDLE_INFO_
↪KHR;
    get_handle_info.semaphore  = sema;
    get_handle_info.handleType = VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT_
↪KHR;

    VK_CHECK(
```

(continues on next page)

(continued from previous page)

```
        ctx.vkd->get_semaphore_win32_handle(ctx.dev, &get_handle_info, &desc.handle.
↪win32.handle));

#else
    desc.type = hipExternalSemaphoreHandleTypeOpaqueFd;

    VkSemaphoreGetFdInfoKHR get_fd_info = {};
    get_fd_info.sType                   = VK_STRUCTURE_TYPE_SEMAPHORE_GET_FD_INFO_KHR;
    get_fd_info.semaphore               = sema;
    get_fd_info.handleType              = VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_
↪BIT_KHR;

    VK_CHECK(ctx.vkd->get_semaphore_fd(ctx.dev, &get_fd_info, &desc.handle.fd));
#endif

    // Import the native semaphore to HIP to create a HIP external semaphore.
    hipExternalSemaphore_t hip_sema;
    HIP_CHECK(hipImportExternalSemaphore(&hip_sema, &desc));
```

When the HIP external memory is exported from Vulkan and imported to HIP, it is not yet ready for use. The Vulkan handle is shared, allowing for memory sharing rather than copying during the export process. To actually use the memory, we need to map it to a pointer so that we may pass it to the kernel so that it can be read from and written to. The external memory map to HIP in the following example:

```
    hipExternalMemoryBufferDesc desc = {};
    desc.offset                      = 0;
    desc.size                        = size;
    desc.flags                       = 0;

    void* ptr;
    HIP_CHECK(hipExternalMemoryGetMappedBuffer(&ptr, mem, &desc));
```

Wait for buffer is ready and not under modification at Vulkan side:

```
            hipExternalSemaphoreWaitParams wait_params = {};
            HIP_CHECK(hipWaitExternalSemaphoresAsync(&this->hip_buffer_ready,
                                                     &wait_params,
                                                     1,
                                                     this->hip_stream));
```

The sinewave kernel implementation:

```
/// \brief The main HIP kernel for this example - computes a simple sine wave over a
/// 2-dimensional grid of points.
/// \param height_map - the grid of points to compute a sine wave for. It is expected␣
↪to be
///   a \p grid_width by \p grid_height array packed into memory.(y on the inner␣
↪axis).
/// \param time - The current time relative to the start of the program.
__global__ void sinewave_kernel(float* height_map, const float time)
{
    const float        freq = 10.f;
    const unsigned int x    = blockIdx.x * blockDim.x + threadIdx.x;
```

(continues on next page)

(continued from previous page)

```
    const unsigned int y    = blockIdx.y * blockDim.y + threadIdx.y;
    const float        u    = (2.f * x) / grid_width - 1.f;
    const float        v    = (2.f * y) / grid_height - 1.f;

    if(x < grid_width && y < grid_height)
    {
        height_map[x * grid_width + y] = sinf(u * freq + time) * cosf(v * freq +␣
↪time);
    }
}
```

Signal to Vulkan that we are done with the buffer and that it can proceed with rendering:

```
        hipExternalSemaphoreSignalParams signal_params = {};
        HIP_CHECK(hipSignalExternalSemaphoresAsync(&this->hip_simulation_finished,
                                                   &signal_params,
                                                   1,
                                                   this->hip_stream));
```

CHAPTER

ELEVEN

GPU PROGRAMMING PATTERNS

GPU programming patterns are fundamental algorithmic structures that enable efficient parallel computation on GPUs. Understanding these patterns is essential for developers looking to effectively harness the massive parallel processing capabilities of modern GPUs for scientific computing, machine learning, image processing, and other computationally intensive applications.

These tutorials describe core programming patterns demonstrating how to efficiently implement common parallel algorithms using the HIP runtime API and kernel extensions. Each pattern addresses a specific computational challenge and provides practical implementations with detailed explanations.

11.1 Common GPU programming challenges

GPU programming introduces unique challenges not present in traditional CPU programming:

- **Memory coherence**: GPUs lack robust cache coherence mechanisms, requiring careful coordination when multiple threads access shared memory.
- **Race conditions**: Concurrent memory access requires atomic operations or careful algorithm design.
- **Irregular parallelism**: Real-world algorithms often have varying amounts of parallel work across iterations.
- **CPU-GPU communication**: Data transfer overhead between host and device must be minimized.

11.2 Tutorial overview

This collection provides comprehensive tutorials on essential GPU programming patterns:

- *Two-dimensional kernels*: Processing grid-structured data such as matrices and images.
- *Stencil operations*: Updating array elements based on neighboring values.
- *Atomic operations*: Ensuring data integrity during concurrent memory access.
- *Multi-kernel applications*: Coordinating multiple GPU kernels to solve complex problems.
- *CPU-GPU cooperation*: Strategic work distribution between CPU and GPU.

11.2.1 Prerequisites

To get the most from these tutorials, you should have:

- Basic understanding of C/C++ programming.
- Familiarity with parallel programming concepts.
- HIP runtime environment installed (see *ROCm install*).
- Basic knowledge of GPU architecture (recommended).

11.2.2 Getting started

Each tutorial is self-contained and can be studied independently, though we recommend following the order presented for a comprehensive understanding:

1. **Start with Two-dimensional kernels** to understand basic GPU thread organization and memory access patterns.
2. **Progress to stencil operations** to learn about neighborhood dependencies.
3. **Study atomic operations** to understand concurrent memory access.
4. **Explore multi-kernel programming** for complex algorithmic patterns.
5. **Check CPU-GPU cooperation** to handle mixed-parallelism workloads.

11.3 Two-dimensional kernels: matrix multiplication tutorial

GPUs provide a massively parallel architecture consisting of thousands of cores, making them exceptionally well-suited for data-parallel computations. Two-dimensional kernel patterns are commonly data-parallel, enabling us to leverage GPU capabilities to exploit this inherent parallelism.

Tasks involving large matrices, which are common in image processing and machine learning applications, can be significantly accelerated by distributing computations across GPU cores. This tutorial explores how to implement matrix multiplication using two-dimensional GPU kernels with HIP.

11.3.1 Prerequisites

To follow this tutorial, you'll need installed drivers and a HIP compiler toolchain to compile your code. HIP supports compiling and running on Linux and Windows with AMD GPUs, the combination of install instructions is more than worth covering as part of this tutorial. For more information about installing HIP development packages, see *ROCm install*.

11.3.2 Characteristics of 2D computational problems

- Spatial locality and data dependencies: Adjacent elements in the grid often exhibit strong spatial correlations, making memory access patterns and cache utilization critical for performance.
- Natural 2D data representation: Many datasets—such as images, numerical matrices, and discretized physical fields—map directly onto a two-dimensional coordinate space.
- Prevalence in simulation and modeling: Numerous scientific and engineering workloads such as finite difference methods, fluid dynamics, heat transfer, and image processing, are inherently two-dimensional.

Modern GPU architectures are engineered to exploit the parallelism of 2D computational grids. By designing kernels that operate on two-dimensional thread blocks and memory layouts, developers can optimize global memory access, minimize latency, and maximize throughput. Leveraging 2D kernel configurations not only aligns the computation with the GPU's hardware topology but also enables substantial performance improvements for domain-specific applications.

11.3.3 Matrix multiplication

Let **A** and **B** be two matrices defined as follows, $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times k}$. The matrix product $C = A \cdot B$ is defined only when the number of columns of A equals the number of rows of B. The resulting matrix $C \in \mathbb{R}^{m \times k}$ has elements given by:

$$C_{ij} = \sum_{r=1}^{n} A_{ir} B_{rj}$$

for all $i = 1, \ldots, m$ and $j = 1, \ldots, k$.

In other words, each element C_{ij} is computed as the dot product of the i-th row vector of A and the j-th column vector of B. This operation is repeated for all valid pairs of (i, j) to construct the complete matrix C.

For two square matrices of size $N \times N$, the computational cost of classical matrix multiplication is $O(N^3)$. As an example, consider $N = 32$:

- **Multiplication operations**: $32^3 = 32{,}768$
- **Addition operations**: $32^2 \times (32 - 1) = 31{,}744$

Each element in the resulting matrix C is computed independently of the others, since it depends only on a single row of A and a single column of B. This property makes matrix multiplication **highly parallelizable** and well-suited for execution on GPUs, multi-core CPUs, or distributed computing architectures.

11.3.4 CPU implementation

A baseline CPU implementation provides a clear understanding of the classical matrix multiplication algorithm before exploring parallel GPU execution.

```
#include <iostream>
#include <cstdlib>
#define N 32

void cpu_matrix_multiplication(float *a, float *b, float *c, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float sum = 0.0f;
            for (int k = 0; k < n; ++k) {
                sum += a[i * n + k] * b[k * n + j];
            }
            c[i * n + j] = sum;
        }
    }
}

int main() {
    float *a, *b, *c;
    a = (float*)malloc(sizeof(float) * N * N);
    b = (float*)malloc(sizeof(float) * N * N);
    c = (float*)malloc(sizeof(float) * N * N);

    // Initialize matrix A
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            a[i * N + j] = static_cast<float>(rand()) / RAND_MAX;
        }
    }

    // Initialize matrix B
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            b[i * N + j] = static_cast<float>(rand()) / RAND_MAX;
        }
    }
```

(continues on next page)

(continued from previous page)

```
    cpu_matrix_multiplication(a, b, c, N);
    free(a);
    free(b);
    free(c);
    return 0;
}
```

The `cpu_matrix_multiplication` function performs the classical $O(N^3)$ matrix multiplication algorithm using three nested loops. The implementation proceeds as follows:

- **Input parameters:** Three pointers to contiguous memory blocks representing matrices A, B, and the output matrix C.
- **Outer and middle loops:** The indices i and j iterate over the rows and columns of the output matrix C, respectively.
- **Innermost loop:** For each element C_{ij}, the loop over k performs a dot product between the i-th row of A and the j-th column of B:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} \cdot B_{kj}$$

- **Temporary accumulation:** A local scalar `sum` accumulates the intermediate sum before being written to `c[i * n + j]`.

This implementation has a computational complexity of $O(N^3)$ and poor cache locality for large matrices, but it serves as a reference for understanding sequential computation before introducing GPU parallelization.

11.3.5 GPU implementation

The following example demonstrates a complete HIP implementation of matrix multiplication, including host and device memory management, kernel implementation, configuration, and synchronization.

11.3.5.1 GPU kernel

The core computation is performed by the GPU kernel, where each thread computes one element of the output matrix. For comparison, the CPU implementation is also provided.

GPU version

```
__global__ void gpu_matrix_multiplication(float *a, float *b, float *c, int n) {

    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    float sum = 0.0f;

    if (row < n && col < n) {
        for (int k = 0; k < n; ++k) {
            sum += a[row * n + k] * b[k * n + col];
        }
        c[row * n + col] = sum;
    }
}
```

CPU version

```
void cpu_matrix_multiplication(float *a, float *b, float *c, int n) {

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            float sum = 0.0f;
            for (int k = 0; k < n; ++k) {
                sum += a[i * n + k] * b[k * n + j];
            }
            c[i * n + j] = sum;
        }
    }
}
```

The outer and middle loops of the CPU implementation are replaced by the parallel execution of the GPU implementation. Each GPU thread computes the **single element** C_{ij} of the output matrix corresponding to one dot product between a row of A and a column of B. This decomposition exposes massive parallelism, as all elements of C can be computed independently and concurrently.

11.3.5.1.1 Thread and block identification

Each thread's global position within the grid is determined by:

```
int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
```

Here:

- `threadIdx.(x|y)`: Local thread indices within a block.
- `blockIdx.(x|y)`: Block indices within the grid.
- `blockDim.(x|y)`: Dimensions of each block (used to scale offsets).

11.3.5.1.2 Boundary checking

Since the total number of threads launched may exceed N^2, boundary checking ensures that threads outside the matrix domain do not perform invalid memory accesses:

```
if (row < n && col < n) {
    // Safe computation region
}
```

11.3.5.1.3 Dot product computation

Within the valid region, each thread executes a dot product over k:

- Loads one element from row `row` of matrix A.
- Loads one element from column `col` of matrix B.
- Multiplies and accumulates these values into `sum`.
- Writes the final scalar result to `c[row * n + col]`.

This kernel performs the same $O(N^3)$ arithmetic operations as the CPU version but distributes them across thousands of concurrent GPU threads, achieving significant acceleration through parallel execution and memory throughput optimization.

11.3.5.2 Step 1: Host memory allocation and initialization

Host memory is allocated for matrices and initialized with random floating-point values.

```
int main() {
    float *h_a, *h_b, *h_c;
    h_a = (float*)malloc(sizeof(float) * N * N);
    h_b = (float*)malloc(sizeof(float) * N * N);
    h_c = (float*)malloc(sizeof(float) * N * N);

    // Initialize matrix A
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            h_a[i * N + j] = static_cast<float>(rand()) / RAND_MAX;
        }
    }

    // Initialize matrix B
    for (int i = 0; i < N; ++i) {
        for (int j = 0; j < N; ++j) {
            h_b[i * N + j] = static_cast<float>(rand()) / RAND_MAX;
        }
    }
```

Description:

- Declare host (CPU) pointers `h_a`, `h_b`, and `h_c`.
- Allocate contiguous memory for each $N \times N$ matrix.
- Initialize input matrices A and B with pseudo-random floating-point values in the range [0, 1).

11.3.5.3 Step 2: Device memory allocation and data transfer

Memory is allocated on the GPU, and input matrices are transferred from host to device.

```
float *d_a, *d_b, *d_c;
hipMalloc((void**)&d_a, sizeof(float) * N * N);
hipMalloc((void**)&d_b, sizeof(float) * N * N);
hipMalloc((void**)&d_c, sizeof(float) * N * N);

hipMemcpy(d_a, h_a, sizeof(float) * N * N, hipMemcpyHostToDevice);
hipMemcpy(d_b, h_b, sizeof(float) * N * N, hipMemcpyHostToDevice);
```

Operations:

1. Allocate GPU (device) memory for matrices `d_a`, `d_b`, and `d_c`.
2. Transfer data from host to device using `hipMemcpy()` with direction `hipMemcpyHostToDevice`.

11.3.5.4 Step 3: Configure and launch kernel

Kernel launch parameters define how threads are organized across blocks and the grid.

```
dim3 threadsPerBlock(BLOCK_SIZE, BLOCK_SIZE);
int n_blocks = static_cast<int>(ceil(static_cast<float>(N) / BLOCK_SIZE));
dim3 blocksPerGrid(n_blocks, n_blocks);

hipLaunchKernelGGL(gpu_matrix_multiplication,
                   blocksPerGrid,
                   threadsPerBlock,
                   0, 0,
                   d_a, d_b, d_c, N);

hipDeviceSynchronize();
```

11.3.5.4.1 Configuration details

The `dim3` type defines thread and block dimensions:

- `threadsPerBlock`: Number of threads per block. A 16×16 block (256 threads total).
- `n_blocks`: Number of blocks per dimension. Computed as $\lceil N/\text{BLOCK\_SIZE} \rceil$.
- `blocksPerGrid`: A grid of blocks covering the entire $N \times N$ matrix.

For example $N = 256$ and `BLOCK_SIZE = 16`:

- `n_blocks`: $\lceil 256/16 \rceil = 16$
- `blocksPerGrid`: $16 \times 16 = 256$
- Total threads: 256 blocks $\times$ 256 threads/block $= 65{,}536$ threads.

Rounding up ensures full coverage of the matrix even when N is not an exact multiple of `BLOCK_SIZE`. The boundary check in the kernel

```
if (row < n && col < n)
```

prevents out-of-bounds memory access for extra threads.

11.3.5.4.2 Synchronization

The call to `hipDeviceSynchronize()` ensures that all GPU computations complete before the CPU accesses results or proceeds to subsequent operations. This is essential for correctness and debugging.

11.3.5.5 Step 4: Copy results back and cleanup

After the kernel execution, results are transferred back to host memory, and all allocated resources are released.

```
hipMemcpy(h_c, d_c, sizeof(float) * N * N, hipMemcpyDeviceToHost);

hipFree(d_a);
hipFree(d_b);
hipFree(d_c);
free(h_a);
free(h_b);
free(h_c);
```

(continues on next page)

(continued from previous page)

```
return 0;
}
```

Summary of final steps:

1. Copy the result matrix from device to host using `hipMemcpy`.
2. Free all device memory allocations with `hipFree`.
3. Release host memory with `free`.
4. Return control to the operating system, indicating successful program termination.

11.3.6 Parallelization benefits

For a 256 × 256 matrix multiplication:

- **Sequential CPU version:** Computes 65,536 output elements serially.
- **Parallel GPU version:** Executes up to 65,536 independent threads concurrently.

This results in a theoretical performance gain proportional to the number of active GPU threads and the device's compute throughput.

Each output element C_{ij} is computed independently from others, since it depends solely on row i of matrix A and column j of matrix B. This independence allows full utilization of GPU streaming multiprocessors and makes the algorithm highly scalable.

11.3.7 Best practices

1. **Choose optimal block sizes**

   Powers of two (e.g., 16 or 32) often yield better occupancy and memory alignment.

2. **Handle boundary conditions**

   Always include thread boundary checks.

3. **Synchronize appropriately**

   Use `hipDeviceSynchronize()` after kernel launches to ensure data consistency.

4. **Memory coalescing**

   Arrange data access patterns so consecutive threads access contiguous memory locations, maximizing bandwidth utilization.

5. **Use shared memory**

   Use shared memory to cache sub-blocks of matrices, significantly reducing global memory latency.

6. **Profile and tune**

   Use tools such as rocprofv3 or ROCm compute profiler to identify bottlenecks and fine-tune kernel launch configurations.

11.3.8 Conclusion

Two-dimensional GPU kernels provide an efficient mechanism to accelerate dense linear algebra computations such as matrix multiplication by exploiting fine-grained data parallelism. This example demonstrates:

- Structuring GPU kernels for 2D problems.

- Managing memory transfers between host and device.
- Configuring thread and block hierarchies.
- Achieving substantial speedups via massive parallel execution.

Understanding these concepts enables developers to implement optimized GPU solutions for computationally intensive workloads, including scientific simulations, numerical linear algebra, and machine learning. Because each output element is computed independently, matrix multiplication serves as an ideal introductory example for mastering GPU programming paradigms applicable to a wide range of data-parallel applications.

11.4 Atomic operations: histogram tutorial

In GPU programming, a core design principle is to **avoid simultaneous writes to the same memory address by multiple threads**. When multiple threads write to the same location without proper synchronization, this creates a **race condition**, where the final result depends on unpredictable thread execution order.

Unlike CPUs, GPUs are designed for high-throughput parallel execution with relaxed memory consistency models and limited cache coherence mechanisms. This architectural choice maximizes bandwidth and scalability but introduces challenges when multiple threads need to safely update shared state.

This tutorial demonstrates how to safely handle **concurrent memory updates** using **atomic operations**, illustrated through the practical example of computing an image brightness histogram on the GPU.

11.4.1 Prerequisites

To follow this tutorial, you'll need installed drivers and a HIP compiler toolchain to compile your code. HIP supports compiling and running on Linux and Windows with AMD GPUs, the combination of install instructions is more than worth covering as part of this tutorial. For more information about installing HIP development packages, see *ROCm install*.

11.4.2 Race condition

A **race condition** occurs when two or more threads attempt to read-modify-write the same memory location concurrently without proper synchronization. Because GPU threads execute asynchronously across multiple cores (compute units), concurrent writes can interleave unpredictably, leading to incorrect results.

For example, if two threads simultaneously attempt:

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

both may read the same old value before either writes back, resulting in only one increment being reflected. This results in **lost updates** and **nondeterministic output**, which must be avoided.

11.4.3 Histogram

A **histogram** partitions continuous data into discrete intervals called **bins** and counts how many data points fall into each bin. In image processing, a histogram typically represents the **distribution of pixel intensities** for example brightness or color channel values.

The histogram algorithm can be expressed as:

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

where $f(x_i)$ maps each data value to its corresponding bin index b, and $\delta()$ is 1 when the value belongs to bin b and 0 otherwise.

The basic computational steps are:

1. Iterate through all pixels (or data points).
2. Determine the appropriate bin for each value.
3. Increment that bin's count.

In a serial CPU program, this is straightforward. On a GPU, thousands of threads may attempt to increment the same bin concurrently, leading to **race conditions** unless atomic synchronization is used.

11.4.3.1 The challenge in parallel context

When multiple threads attempt to increment the same bin:

- One thread's update can overwrite another's pending increment.
- Memory coherence cannot guarantee ordered visibility across thread blocks.
- The final result may be inconsistent or incorrect.

This necessitates synchronization mechanisms to ensure that updates occur in a **mutually exclusive** manner without introducing high overhead.

11.4.4 Atomic operations

An **atomic operation** ensures that a compound operation — typically a read-modify-write sequence — executes as an **indivisible unit**. From the programmer's perspective, atomicity guarantees that no other thread can observe a partially completed operation.

Formally, an operation $O(x)$ on shared variable x is **atomic** if its execution satisfies:

$$\forall T_i, T_j, \text{ the effects of } O(x) \text{ appear serializable.}$$

That is, all threads observe results as if operations occurred in a single, sequential order.

11.4.4.1 Mechanics

Atomic operations on GPUs are implemented in hardware through a **memory arbitration unit** that locks a cache line, performs the modification, and releases the lock. This ensures correctness even under massive parallelism.

When a thread performs an atomic operation:

1. The target memory location is temporarily locked.
2. The value is fetched and updated.
3. The update is written back, and the lock is released.

No other thread can modify the same memory location during this sequence.

11.4.4.2 Atomic functions

HIP provides a wide set of atomic primitives to synchronize updates to shared memory or global memory locations:

Operation	Description
`atomicAdd`	Atomically adds a value to a memory location and returns the old value.
`atomicSub`	Atomically subtracts a value.
`atomicExch`	Atomically exchanges values between a register and memory.
`atomicCAS`	Performs an atomic compare-and-swap; fundamental for implementing locks.
`atomicMax`/`atomicMin`	Updates to the maximum or minimum of two values.
`atomicInc`/`atomicDec`	Atomically increments or decrements a counter, wrapping at a boundary.

Atomic operations in kernels can operate on block scope (shared memory), device scope (global memory), or system scope (system memory), depending on hardware support.

For more information, please check *atomic functions*.

11.4.5 Image brightness histogram

We will compute a histogram that captures the **distribution of pixel brightness** in an RGB image. The algorithm:

1. Reads image data in **channel-height-width** format.
2. Converts RGB values to grayscale brightness.
3. Maps brightness to a histogram bin.
4. Atomically increments the corresponding bin counter.

11.4.5.1 Kernel implementation

```
__global__ void calculateHistogram(float* imageData, int* histogram,
                                   int width, int height,
                                   int channels, int numBins)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= width || y >= height)
        return;

    int idx = (y * width + x) * channels;
    float brightness = 0.0f;

    for (int c = 0; c < channels; ++c)
        brightness += imageData[idx + c];

    brightness /= channels; // Normalize to [0, 1]
    int bin = static_cast<int>(brightness * numBins);

    // Atomic increment to avoid race conditions
    atomicAdd(&histogram[bin], 1);
}
```

11.4.5.1.1 Thread identification

Each thread computes one pixel's contribution using its 2D thread and block indices:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

This mapping provides a 1:1 correspondence between threads and pixels, making the computation naturally parallel.

11.4.5.1.2 Brightness computation

Each pixel's brightness is computed as the arithmetic mean of its RGB channels:

$$I'(x, y) = \frac{R + G + B}{3}$$

This value is then normalized to [0, 1] and mapped to one of *numBins* histogram intervals.

11.4.5.1.3 Safe histogram update

The key step is:

```
atomicAdd(&histogram[bin], 1);
```

This ensures that even if thousands of threads map to the same bin, each increment is serialized correctly, maintaining an accurate bin count.

11.4.6 Performance characteristics

11.4.6.1 Benefits

- **Correctness under parallel updates:** Ensures race-free accumulation.
- **Simplified synchronization:** No explicit locks or barriers needed.
- **Hardware-level efficiency:** Implemented directly in the GPU memory subsystem.

11.4.6.2 Limitations

While atomic operations guarantee correctness, they can **serialize execution** when multiple threads target the same memory address. This causes contention and reduces effective parallelism.

Typical performance degradation sources include:

- **Hot bins:** When many pixels fall into a small subset of bins.
- **Global memory atomics:** Global memory atomics are slower than shared memory atomics due to higher access latency.
- **Warp serialization:** Threads within a warp waiting for the same atomic target serialize.

11.4.7 Best practices

1. **Apply atomic operations only where necessary**

   Atomic instructions serialize access to a memory location and use can diminish SIMT parallel efficiency and increase warp stalls. Restrict atomic usage to code paths where data races cannot be eliminated through algorithmic restructuring.

2. **Minimize contention**

   High contention on a single address or a small set of addresses leads to serialization. Distribute writes across independent memory locations.

3. **Leverage shared memory**

   Use fast, low-latency shared memory to aggregate partial results within a block before issuing a single atomic update to global memory.

4. **Validate correctness**

   Validate the numerical and logical correctness of GPU kernels by comparing against single-threaded or deterministic multi-threaded CPU baselines.

5. **Profile regularly**

   GPU performance is highly sensitive to thread divergence, memory-access patterns, and workload distribution. Regularly use profiling tools such as rocprofv3 or ROCm compute profiler to examine warp-level execution efficiency, memory-coalescing behavior, occupancy, and atomic throughput bottlenecks.

11.4.8 Conclusion

Atomic operations provide a low-level synchronization mechanism that allows correct and deterministic parallel updates to shared data structures. In the histogram example, `atomicAdd()` ensures that all threads safely contribute to their corresponding bins, preventing race conditions.

While atomics incur some serialization overhead, they are indispensable for algorithms that require concurrent accumulation or counting. By applying techniques like privatization and reduction, developers can achieve both **correctness** and **high performance** on modern GPUs.

Atomic operations form the foundation for more advanced synchronization patterns, including parallel reductions, prefix sums, and graph traversal, and are essential for developing scalable, data-parallel GPU algorithms.

11.5 CPU-GPU cooperative computing: K-means clustering tutorial

Modern heterogeneous systems combine CPUs and GPUs to maximize computational throughput. GPUs provide massive data-parallel performance, whereas CPUs excel at complex control logic, and latency-sensitive serial tasks. Many real-world algorithms—including unsupervised clustering—contain both parallelizable and inherently sequential components.

This tutorial demonstrates a hybrid CPU–GPU cooperative execution model using the K-means clustering algorithm, showcasing partitioned workload distribution, memory management, and performance optimization strategies in heterogeneous architectures.

11.5.1 Prerequisites

To follow this tutorial, you'll need installed drivers and a HIP compiler toolchain to compile your code. HIP supports compiling and running on Linux and Windows with AMD GPUs, the combination of install instructions is more than worth covering as part of this tutorial. For more information about installing HIP development packages, see *ROCm install*.

11.5.2 CPU–GPU cooperative computing

Modern computing platforms integrate **central processing units (CPUs)** and **graphics processing units (GPUs)** into unified heterogeneous systems. Each processor type offers distinct architectural strengths:

- **CPUs** feature a small number of complex cores optimized for sequential execution, branching, and latency-sensitive control flow.

- **GPUs** consist of thousands of simpler cores optimized for throughput and massive data-level parallelism.

Cooperative computing refers to a programming paradigm that combines these capabilities in a coordinated execution model, where both CPU and GPU perform complementary portions of a workload. Rather than treating the GPU as a passive accelerator, this approach distributes computation dynamically between devices to maximize total system utilization.

11.5.3 K-means clustering

K-means is an unsupervised machine learning algorithm that partitions a dataset into k clusters (groups of similar data points) by minimizing intra-cluster variance. It iteratively refines cluster assignments and centroid (center point of a cluster) locations until convergence.

The optimization objective is defined as:

$$\min_{C_1,\ldots,C_k} \sum_{i=1}^{k} \sum_{x_j \in C_i} \|x_j - \mu_i\|^2$$

where μ_i is the centroid of cluster C_i.

K-means is frequently used in:

- **Customer segmentation**: Grouping customers by behavior patterns
- **Image compression**: Reducing color palette by clustering similar colors
- **Anomaly detection**: Identifying outliers that don't fit clusters
- **Document clustering**: Organizing similar documents together
- **Feature engineering**: Creating new features based on cluster membership

11.5.4 Algorithm

The K-means algorithm iteratively refines a partition of a dataset into k clusters by alternating between two primary computational phases: **assignment** and **update**. Each iteration minimizes the total within-cluster variance, driving the system toward convergence where centroid movement or membership changes fall below a defined threshold.

11.5.4.1 Iterative procedure

1. **Initialization**: Select initial centroids (random or via K-means++).
2. **Assignment**: Assign each data point to the nearest centroid.
3. **Update**: Recalculate each centroid as the mean of its assigned members.
4. **Convergence**: Repeat until centroids stabilize or a maximum iteration limit is reached.

These phases alternate until the solution converges or a maximum iteration count is reached.

11.5.4.1.1 Initialization

The algorithm begins by selecting initial centroid positions. This step significantly influences both convergence rate and clustering quality.

- **Random initialization**: centroids are randomly chosen from the dataset.
- **K-means++**: probabilistic seeding to spread centroids across data space.
- **Domain-specific heuristics**: custom initializations for structured data.

Initial centroid diversity reduces the likelihood of poor local minima and improves overall stability.

11.5.4.1.2 Assignment

For each data point x_i, the algorithm computes the Euclidean distance to each centroid μ_j and assigns the point to the nearest cluster:

$$C_i = \arg\min_j \|x_i - \mu_j\|^2$$

- Each point–centroid distance calculation is independent.
- Ideal for SIMD/SIMT architectures (GPU execution).
- Dominated by dense floating-point arithmetic and memory bandwidth utilization.

This phase represents the **embarrassingly parallel** portion of the algorithm and provides the largest opportunity for GPU acceleration.

11.5.4.1.3 Update

Once all data points are assigned, new centroid positions are computed as the mean of their corresponding cluster members:

$$\mu_j = \frac{1}{|C_j|} \sum_{x_i \in C_j} x_i$$

- Requires aggregation and division per cluster.
- Involves variable membership counts across clusters.
- Reduction-heavy and branch-divergent.

Although reduction can be implemented on GPUs, small k values and irregular membership sizes typically make the CPU more efficient for this phase, given its superior cache hierarchy and flexible control flow.

11.5.4.1.4 Convergence

After the update phase, convergence is evaluated using one of the following criteria:

- No change in point memberships.
- Centroid displacement below a user-defined threshold.
- Iteration count exceeds maximum limit.

If the convergence condition is not met, the assignment and update phases repeat.

11.5.4.2 Summary of cooperative execution

1. **GPU:** performs parallel distance evaluations and membership assignments.
2. **CPU:** executes centroid averaging and convergence checks.
3. **Synchronization:** only essential data (centroids and membership arrays) is exchanged per iteration.

This division of labor minimizes data movement and maximizes hardware utilization. The resulting hybrid implementation combines GPU throughput for massive data-parallel operations with CPU efficiency for aggregation and control tasks, enabling scalable clustering performance on heterogeneous systems.

11.5.5 Implementation

This implementation follows a hybrid CPU/GPU design to accelerate the most computationally expensive phase of K-means: assigning each data point to its nearest centroid. The GPU performs distance calculations in parallel, while the CPU handles the centroid recomputation, where sequential reductions are efficient and data sizes are small. The algorithm iterates between GPU-based membership updates and CPU-based centroid averaging until convergence or a maximum iteration count is reached.

11.5.5.1 Data structures

The implementation stores all data in simple, contiguous arrays for efficient memory access on both CPU and GPU. Data points and centroids are represented as flattened `std::vector<float>` arrays, while cluster assignments are stored as `std::vector<int>`.

```
// length: Number of data points to cluster
// dimension: Number of features per data point
// k: Number of clusters

std::vector<float> data;           // Data points (length * dimension)
std::vector<float> centroids;      // Centroid positions (k * dimension)
std::vector<int> memberships;      // Cluster assignments (length)
```

11.5.5.2 Main loop

The core K-means iteration:

```
// length is an integer for the number of entries to be clustered.
// dimension is an integer for the number of properties of each entry.
// k is an integer that determines the number of clusters.

std::vector<float> centroids = initializeCentroids(length * dimension, k);
std::vector<int> memberships(length, 0);

for (int iteration = 0; iteration < maxIterations; ++iteration) {
    // Determine the cluster that each entry belongs to.
    // The function returns how many entries changed membership.
    int membershipChanges = updateMembership(data, centroids, memberships);

    // Converge checking.
    if (membershipChanges == 0) {
        break;
    }

    // Calculate new centroids.
    std::vector<Point> newCentroids = centroids;
    updateCentroid(data, newCentroids, memberships);
    centroids = newCentroids;
}
```

Loop structure:

1. Update memberships using GPU
2. Check for convergence (no changes)
3. Update centroids using CPU

4. Repeat until convergence or max iterations

11.5.5.3 GPU membership update function

The `updateMembership` function serves as the interface between CPU and GPU:

```
int updateMembership(float* data, float* centroids, int* membership,
                     int dataSize, int dimension, int k) {
    // gpuData is allocated and copied to the GPU earlier.

    float *gpuCentroids, *gpuMembership;

    // Allocate GPU memory
    hipMalloc(&gpuCentroids, k * dimension * sizeof(float));
    hipMalloc(&gpuMembership, dataSize * sizeof(int));

    // Copy data from CPU to GPU
    hipMemcpy(gpuCentroids, centroids, k * dimension * sizeof(float),
              hipMemcpyHostToDevice);

    // Calculate the sizes for the kernel launch
    int localSize = 256;
    int globalSize = (dataSize + localSize - 1) / localSize;

    // Launch the kernel
    updateMembershipGPU<<<globalSize, localSize>>>(
        gpuData, gpuCentroids, gpuMembership, dataSize, dimension, k);
    hipDeviceSynchronize();

    // Create CPU Data to hold the results
    std::vector<int> cpuNewMembership(dataSize);

    // Copy GPU data back to CPU
    hipMemcpy(cpuNewMembership.data(), gpuMembership,
              dataSize * sizeof(int), hipMemcpyDeviceToHost);

    // Count membership updates
    int membershipUpdate = 0;
    for (int i = 0; i < dataSize; ++i) {
        if (membership[i] != cpuNewMembership[i]) {
            membershipUpdate++;
            membership[i] = cpuNewMembership[i]; // Update the original
        }
    }

    // Free GPU memory
    hipFree(gpuCentroids);
    hipFree(gpuMembership);

    return membershipUpdate;
}
```

11.5.5.3.1 Step 1: GPU memory allocation

```
float *gpuCentroids, *gpuMembership;
hipMalloc(&gpuCentroids, k * dimension * sizeof(float));
hipMalloc(&gpuMembership, dataSize * sizeof(int));
```

Allocate space on the GPU for:

- **Centroids**: k centroids, each with dimension features
- **Membership assignments**: One cluster ID per data point

11.5.5.3.2 Step 2: data transfer to GPU

```
hipMemcpy(gpuCentroids, centroids, k * dimension * sizeof(float),
          hipMemcpyHostToDevice);
```

Transfer the current centroid positions from CPU to GPU. Note that the data points (`gpuData`) are already on the GPU from earlier initialization.

11.5.5.3.3 Step 3: kernel configuration

```
int localSize = 256;
int globalSize = (dataSize + localSize - 1) / localSize;
```

- `localSize`: 256 threads per block (common choice)
- `globalSize`: Number of blocks needed to cover all data points
- Rounding up ensures we process all data points

11.5.5.3.4 Step 4: kernel launch

```
updateMembershipGPU<<<globalSize, localSize>>>(
    gpuData, gpuCentroids, gpuMembership, dataSize, dimension, k);
hipDeviceSynchronize();
```

Launch the GPU kernel to compute cluster assignments in parallel, then wait for completion.

11.5.5.3.5 Step 5: retrieve results

```
std::vector<int> cpuNewMembership(dataSize);
hipMemcpy(cpuNewMembership.data(), gpuMembership,
          dataSize * sizeof(int), hipMemcpyDeviceToHost);
```

Copy the new membership assignments back from GPU to CPU.

11.5.5.3.6 Step 6: count changes

```
int membershipUpdate = 0;
for (int i = 0; i < dataSize; ++i) {
    if (membership[i] != cpuNewMembership[i]) {
        membershipUpdate++;
```

(continues on next page)

(continued from previous page)

```
        membership[i] = cpuNewMembership[i];
    }
}
```

Count how many data points changed clusters. This value determines if the algorithm has converged.

11.5.5.3.7 Step 7: cleanup

```
hipFree(gpuCentroids);
hipFree(gpuMembership);
return membershipUpdate;
```

Free temporary GPU memory and return the change count.

11.5.5.3.8 Kernel implementation

```
__global__ void updateMembershipGPU(
    float* data, float* centroids, int* membership,
    int dataSize, int dimension, int k)
{
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid < dataSize) {
        float minDistance = INFINITY;
        int bestCluster = 0;

        // Find nearest centroid
        for (int cluster = 0; cluster < k; ++cluster) {
            float distance = 0.0f;

            // Calculate Euclidean distance
            for (int d = 0; d < dimension; ++d) {
                float diff = data[tid * dimension + d] -
                             centroids[cluster * dimension + d];
                distance += diff * diff;
            }

            if (distance < minDistance) {
                minDistance = distance;
                bestCluster = cluster;
            }
        }

        membership[tid] = bestCluster;
    }
}
```

Kernel operation:

1. Each thread processes one data point
2. Calculates distance to all k centroids
3. Assigns point to nearest centroid

4. Stores result in membership array

11.5.5.4 CPU centroid update

The CPU handles the averaging operation:

```
void updateCentroid(float* data, float* centroids, int* membership,
                    int dataSize, int dimension, int k)
{
    std::vector<int> counts(k, 0);
    std::vector<float> sums(k * dimension, 0.0f);

    // Accumulate sums for each cluster
    for (int i = 0; i < dataSize; ++i) {
        int cluster = membership[i];
        counts[cluster]++;

        for (int d = 0; d < dimension; ++d) {
            sums[cluster * dimension + d] += data[i * dimension + d];
        }
    }

    // Calculate averages (new centroids)
    for (int cluster = 0; cluster < k; ++cluster) {
        if (counts[cluster] > 0) {
            for (int d = 0; d < dimension; ++d) {
                centroids[cluster * dimension + d] =
                    sums[cluster * dimension + d] / counts[cluster];
            }
        }
    }
}
```

The centroid update requires:

1. **Reduction operation**: Sum all points in each cluster.
2. **Variable-sized groups**: Clusters have different numbers of points.
3. **Division**: Calculate average (not easily parallelized).

While GPUs can perform reductions, for this workload:

- The number of clusters (k) is typically small (< 100).
- The CPU can efficiently handle this sequential aggregation.
- Avoiding GPU complexity keeps code simpler.

11.5.5.4.1 Data transfer considerations

Data already on CPU:

- Membership array was just copied back.
- Data points can be kept on CPU for small datasets.
- Centroids are small (k × dimension values).

Transfer costs:

- Only centroids need to copy back to GPU.
- Much smaller than the full dataset.
- Overhead justified by parallel speedup in assignment phase.

11.5.6 Best practices

This section outlines recommended practices for implementing an efficient GPU-accelerated Breadth-First Search (BFS). It highlights design principles, memory-management strategies, and debugging techniques that help ensure correctness, maintainability, and high performance when mapping BFS onto modern GPU architectures.

11.5.6.1 Design principles

1. **Identify parallelism**

   Decompose the K-means workflow into independent, data-parallel kernels such as distance computation. Minimize sequential dependencies in assignment and reduction steps.

2. **Minimize host–device data movement**

   Maintain dataset and intermediate buffers resident on the GPU across iterations. Transfer only convergence metrics or centroids when absolutely necessary.

3. **Batch and fuse operations**

   Combine smaller kernels such as distance computation and assignment, or process multiple mini-batches per kernel launch to reduce kernel invocation and PCIe transfer overhead.

4. **Overlap communication with computation**

   Utilize asynchronous HIP streams to overlap host–device memory transfers with GPU kernel execution. Employ pinned (page-locked) memory for faster DMA transfers.

11.5.6.2 Memory strategy

1. **Persistent device allocations**

   Allocate GPU memory once before iterative processing. Reuse buffers such as those for centroids, assignments, and temporary reductions to avoid the overhead of repeated `hipMalloc()` and `hipFree()` calls.

2. **Data locality optimization**

   Co-locate data structures according to access frequency:

   - Store high-throughput arrays (points, centroids) in global memory with coalesced access.
   - Cache small, frequently reused values in shared memory or registers.

3. **Transfer scheduling**

   Schedule host–device transfers asynchronously while the GPU executes kernels. Use HIP events or streams to synchronize only when necessary.

4. **Memory pooling and reuse**

   Use memory pools such as `hipMallocAsync()`, `hipMallocFromPoolAsync()`, or custom allocators to mitigate fragmentation and reduce allocation latency, especially in iterative or batched workloads.

11.5.6.3 Performance considerations

Aspect	Recommendation
Large datasets	GPU acceleration scales linearly with dataset size. Prioritize keeping data on GPU to avoid PCIe bottlenecks.
Many iterations	Use persistent buffers and asynchronous reduction to minimize per-iteration synchronization overhead.
Small number of clusters (k)	Kernel execution may be underutilized. CPU execution or hybrid scheduling may be more efficient.
High-dimensional data	Distance computation dominates cost. Leverage GPU shared memory for centroid caching and ensure memory coalescing for point vectors.

11.5.7 When to use CPU-GPU cooperation

- **Hybrid computational structure**

  The algorithm exhibits a clear division between compute-intensive, data-parallel kernels such as distance computation in K-means and lightweight, control-heavy serial stages such as centroid updates or convergence checks.

- **High arithmetic intensity in parallel regions**

  The GPU-executed portion performs substantial arithmetic operations per memory access, ensuring efficient utilization of GPU cores and minimizing the impact of memory latency.

- **Low-complexity reduction or synchronization on CPU**

  The CPU phase primarily aggregates results or performs decision logic that does not justify GPU kernel execution overhead.

- **Large-scale data processing**

  Dataset size exceeds the threshold where PCIe transfer costs can be amortized, enabling sustained GPU throughput and effective memory reuse.

- **Iterative or streaming workloads**

  Algorithms involving multiple iterations benefit from persistent GPU memory allocations and overlapped data transfer, significantly reducing per-iteration latency.

11.5.8 When to avoid CPU-GPU cooperation

- **Fully parallelizable workloads**

  Algorithms with uniform, independent computations across all data points are better suited for GPU-only execution without CPU coordination overhead.

- **Low computational density per element**

  If each data element requires few operations relative to transfer cost, CPU–GPU communication latency will dominate, leading to suboptimal performance.

- **High inter-phase data dependency**

  Frequent synchronization or data exchange between CPU and GPU phases prevents effective pipeline overlap and leads to idle compute units.

- **Small problem sizes**

  When datasets fit comfortably in CPU cache or system memory, the GPU launch overhead and transfer latency outweigh any computational gains from offloading.

11.5.9 Conclusion

The K-means implementation illustrates **heterogeneous workload partitioning** between CPU and GPU. By mapping compute-intensive, data-parallel operations to the GPU and reduction-heavy serial logic to the CPU, total runtime is minimized while code complexity remains manageable.

This CPU–GPU cooperative execution paradigm generalizes to many algorithms combining reduction, aggregation, and distance-based computation—enabling scalable, efficient utilization of modern heterogeneous hardware.

11.6 Stencil operations: image convolution tutorial

Stencil operations represent an important class of **embarrassingly parallel** algorithms that are ideally suited for GPU acceleration. A stencil algorithm iteratively updates data in an array based on a data item's adjacent cells, making it a fundamental technique in various computational domains.

11.6.1 Prerequisites

To follow this tutorial, you'll need installed drivers and a HIP compiler toolchain to compile your code. HIP supports compiling and running on Linux and Windows with AMD GPUs, the combination of install instructions is more than worth covering as part of this tutorial. For more information about installing HIP development packages, see *ROCm install*.

11.6.2 Applications of stencil operations

Stencil algorithms are commonly used in:

- **Physics simulations**: Modeling heat transfer, fluid dynamics, and wave propagation
- **Partial differential equations**: Numerical solutions to scientific computing problems
- **Image processing**: Convolutional operations for smoothing, sharpening, and edge detection
- **Convolutional Neural Networks**: A major building block in modern deep learning

By applying different image convolution kernels (not to be confused with GPU kernels), these algorithms can smooth and sharpen image features and detect edges effectively.

11.6.3 Image convolution

An image convolution applies a small matrix (the **mask** or **filter kernel**) to an input image. For each pixel (x, y), the output is computed as:

$$I'(x,y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} M(i,j) \cdot I(x+i, y+j)$$

where $M(i,j)$ is the mask coefficient, and r is half the mask width (assuming a square kernel).

Step by step description of the equation:

1. Center the mask over the current pixel
2. Multiply each mask value by the corresponding image pixel
3. Sum all the products
4. Store the result as the new pixel value

11.6.3.1 Dimensionality of stencils

Stencil computations extend beyond 2D image grids:

- **1D:** Signal filtering, time series processing
- **2D:** Image processing, texture analysis
- **3D:** Volume data, fluid flow, and physical field simulation

This tutorial focuses on **2D image convolution**, the most common stencil operation in visual and scientific computing.

11.6.3.2 The smoothing operation

The tutorial implements a **box blur** (uniform smoothing filter). Each pixel's new value is the average of its local neighborhood. This operation:

- Reduces noise by averaging local intensity variations.
- Acts as a low-pass filter, attenuating high-frequency components.
- Provides an ideal example of a stencil computation with uniform weights.

11.6.4 Two-dimensional grid architecture

The tutorial uses a two-dimensional grid that maps to the shape of the image, significantly simplifying the implementation. This approach:

- Maps threads directly to pixel positions
- Simplifies coordinate calculations
- Enables intuitive spatial reasoning
- Aligns with the natural structure of images

11.6.4.1 Grid configuration

Rather than using a single integer to represent the size of the grid, the tutorial uses a `dim3` object containing three values to represent the number of block-based work items per dimension:

- **x dimension**: Width of the image
- **y dimension**: Height of the image
- **z dimension**: Set to 1 for 2D problems

11.6.5 Complete implementation

This section provides the complete source code for the 2D convolution implementation.

11.6.5.1 Header and setup

```
#include <hip/hip_runtime.h>
#include <vector>
#include "image.h"
```

11.6.5.2 The convolution kernel

Here's the complete 2D convolution kernel for image smoothing:

```
__global__ void conv2d(uint8_t *image, float *mask, int image_width,
                       int image_height, int mask_width, int mask_height)
{
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x >= image_width || y >= image_height) {
        return;
    }

    float sum = 0;
    for (int i = 0; i < mask_width; i++) {
        for (int j = 0; j < mask_height; j++) {
            // Calculate the coordinate of the pixel to read.
            int image_x = x + i - mask_width / 2;
            int image_y = y + j - mask_height / 2;

            // Do not read outside the image.
            if (image_x < 0 || image_x >= image_width ||
                image_y < 0 || image_y >= image_height) {
                continue;
            }

            // Accumulate the value of the pixel.
            int image_index = image_y * image_width + image_x;
            int mask_index = j * mask_width + i;
            sum += image[image_index] / 255.0f * mask[mask_index];
        }
    }

    int image_index = y * image_width + x;
    image[image_index] = sum * 255;
}
```

11.6.5.2.1 Thread identification in 2D

To obtain thread IDs in both x and y dimensions:

```
int x = blockIdx.x * blockDim.x + threadIdx.x;
int y = blockIdx.y * blockDim.y + threadIdx.y;
```

This calculation combines:

- `threadIdx.x` and `threadIdx.y`: Local thread coordinates within a block
- `blockIdx.x` and `blockIdx.y`: Block coordinates within the grid
- `blockDim.x` and `blockDim.y`: Block dimensions

11.6.5.2.2 Boundary checking

```
if (x >= image_width || y >= image_height) {
    return;
}
```

This ensures threads don't process pixels outside the image bounds. Threads that exceed the image dimensions simply return without doing work.

11.6.5.2.3 Mask application loop

The nested loops iterate over the mask dimensions:

```
for (int i = 0; i < mask_width; i++) {
    for (int j = 0; j < mask_height; j++) {
        // Process each mask element
    }
}
```

11.6.5.2.4 Coordinate calculation

For each position in the mask, calculate the corresponding image coordinate:

```
int image_x = x + i - mask_width / 2;
int image_y = y + j - mask_height / 2;
```

The `- mask_width / 2` and `- mask_height / 2` center the mask over the current pixel.

11.6.5.2.5 Edge handling

```
if (image_x < 0 || image_x >= image_width ||
    image_y < 0 || image_y >= image_height) {
    continue;
}
```

This prevents reading outside the image boundaries. When the mask extends beyond the image edge, the code simply skips those pixels (continue to the next iteration).

11.6.5.2.6 Accumulation

```
int image_index = image_y * image_width + image_x;
int mask_index = j * mask_width + i;
sum += image[image_index] / 255.0f * mask[mask_index];
```

For each mask position:

1. Calculate the flattened array index for the image pixel
2. Calculate the flattened array index for the mask value
3. Normalize the pixel value (divide by 255 to get 0-1 range)
4. Multiply by the mask weight and accumulate

11.6.5.2.7 Writing the result

```
int image_index = y * image_width + x;
image[image_index] = sum * 255;
```

After processing all mask positions, write the accumulated result back to the output image, scaling back to the 0-255 range.

11.6.5.3 Host code implementation

The host code manages memory allocation, data transfer, and kernel launch.

11.6.5.3.1 Main function setup

```
int main() {
    int width, height, channels;
    static const int maskWidth = 200;
    static const int maskHeight = 200;
    std::vector<float> mask(maskWidth * maskHeight * channels);

    // Initialize mask with uniform averaging weights
    for (int i = 0; i < maskWidth * maskHeight; ++i) {
        mask[i] = 1.0f / maskWidth / maskHeight / channels;
    }

    // Load an image from disk (implementation not shown)
```

11.6.5.3.2 Mask initialization

The mask is initialized with uniform weights that sum to 1.0:

- Each element is `1.0 / (maskWidth * maskHeight * channels)`
- This creates an averaging filter
- When applied, it produces a smoothing (blurring) effect

11.6.5.3.3 Memory allocation and data transfer

```
// Allocate GPU memory and copy data to the GPU.
uint8_t *d_image;
float *d_mask;
hipMalloc(&d_image, width * height * channels * sizeof(uint8_t));
hipMalloc(&d_mask, maskWidth * maskHeight * channels * sizeof(float));

hipMemcpy(d_image, image, width * height * channels * sizeof(uint8_t),
          hipMemcpyHostToDevice);
hipMemcpy(d_mask, mask.data(),
          maskWidth * maskHeight * channels * sizeof(float),
          hipMemcpyHostToDevice);
```

11.6.5.3.4 Grid configuration and kernel launch

```
// Calculate grid size and launch the kernel.
dim3 block_size = {16, 16, 1};
dim3 grid_size = {(width + block_size.x - 1) / block_size.x,
                  (height + block_size.y - 1) / block_size.y, 1};

conv2d<<<grid_size, block_size>>>(d_image, d_mask, width, height,
                                  maskWidth, maskHeight);
hipDeviceSynchronize();
```

Grid size calculation:

- `block_size`: 16 × 16 threads per block (256 threads total).
- `grid_size`: Calculated to cover the entire image.
- The `(width + block_size.x - 1) / block_size.x` formula ensures there are enough blocks to cover all pixels, rounding up.

11.6.5.3.5 Retrieving results and cleanup

```
    // Copy the data back to the host.
    hipMemcpy(image, d_image, width * height * channels * sizeof(uint8_t),
              hipMemcpyDeviceToHost);

    // Store the image to disk (implementation not shown)

    hipFree(d_image);
    hipFree(d_mask);

    return 0;
}
```

11.6.6 Performance considerations

For high-resolution images for example 4096×4096 pixels, the GPU acceleration provides:

- **Massive parallelism:** Tens of thousands of concurrent threads.
- **High throughput:** Leveraging GPU memory bandwidth and computational density.
- **Scalability:** Linear speedup with increased SM occupancy.

Typical speedups over CPU implementations range from **10× to 100×**, depending on image size, mask complexity, and GPU architecture.

11.6.6.1 Memory access patterns

Image convolution requires repeated access to neighboring pixels, leading to non-coalesced memory transactions. Optimizing memory access is essential:

- Non-coalesced memory accesses (not all accesses are contiguous).
- Repeated reads of the same pixels by adjacent threads.
- Potential for optimization using shared memory (advanced technique).

11.6.7 Best practices

1. **Handle boundaries carefully**

   Conditional branches in edge regions can cause wavefronts or warps to execute serially. Prefer approaches that avoid per-thread branching, including:

   - Pre-clamping coordinates to valid index ranges
   - Padding or haloing input images so all threads operate on valid data
   - Using hardware boundary modes (such as texture sampling modes on RDNA) to offload boundary handling

   These techniques help maintain high SIMD lane utilization across the entire grid.

2. **Center your stencil properly**

   Compute the stencil origin using mask_width / 2 (or the equivalent for rectangular masks) to ensure correct alignment between the input data and the mask coefficients. This prevents off-by-one misalignment that can propagate as spatial artifacts.

3. **Select mask sizes based on compute–memory tradeoffs**

   Larger kernels increase arithmetic intensity but also expand the set of neighbor loads per output element. Balance mask dimensions with available bandwidth, register pressure, and shared-memory capacity, particularly when implementing separable or multi-pass stencils.

4. **Normalize properly**

   Ensure mask weights sum to the intended normalization constant, commonly 1.0 for averaging operations. When using integer or half-precision paths, verify scaling behavior to avoid overflow or unintended bias.

5. **Consider edge strategies**

   Adopt a clear policy for pixels whose neighborhoods extend outside the valid domain. Options include skipping output generation, clamping to the nearest valid coordinate, wrapping coordinates, or mirroring.

11.6.8 Conclusion

Stencil operations are a fundamental pattern in GPU computing that enables efficient parallel processing of spatially-dependent data. The 2D convolution example demonstrates:

- How to structure kernels for stencil patterns
- Proper boundary handling for neighborhood operations
- Effective use of 2D thread grids that map naturally to image structure
- Memory access patterns for adjacent data elements

By understanding stencil operations, developers can implement a wide range of image processing algorithms, scientific simulations, and deep learning operations on GPUs. The patterns demonstrated here extend beyond image processing to any computational problem involving spatial relationships in multi-dimensional data.

The key to successful stencil implementations is carefully managing boundary conditions, ensuring correct coordinate calculations, and leveraging the GPU's parallel architecture to process many independent stencil operations simultaneously.

11.7 Multi-kernel programming: breadth-first search tutorial

Many real-world GPU workloads involve multiple kernels cooperating to solve a single problem. This tutorial explores **multi-kernel GPU programming** using the breadth-first search (BFS) algorithm, a foundational graph traversal method widely used in networking, path-finding, and social network analysis.

The implementation is adapted from the **Rodinia benchmark suite**, a well-known collection of heterogeneous computing workloads that demonstrate different parallel programming strategies.

11.7.1 Prerequisites

To follow this tutorial, you'll need installed drivers and a HIP compiler toolchain to compile your code. HIP supports compiling and running on Linux and Windows with AMD GPUs, the combination of install instructions is more than worth covering as part of this tutorial. For more information about installing HIP development packages, see *ROCm install*.

11.7.2 Multi-kernel GPU programming

In GPU computing, some algorithms cannot be efficiently expressed using a single kernel due to synchronization or dependency constraints. Instead, they are decomposed into multiple kernels that execute sequentially, with each kernel responsible for a specific computation phase.

This approach, called **multi-kernel programming**, is essential when:

- Results from one kernel determine the input for the next.
- Global synchronization between thread blocks is required.
- Control flow depends on runtime conditions.
- The algorithm involves iterative or level-wise processing.

11.7.3 Breadth-first search (BFS)

Breadth-first search (BFS) is a **layered graph traversal algorithm** that explores nodes level by level, starting from a root node. It guarantees finding the shortest path (in edge count) to all reachable nodes in an unweighted graph.

Applications of BFS include:

- **Path-finding**: Finding shortest paths between nodes.
- **Peer-to-peer networking**: Network topology discovery.
- **GPS navigation**: Route planning and optimization.
- **Social networks**: Friend recommendations and connection analysis.
- **Web crawling**: Systematic website exploration.

11.7.3.1 Algorithm characteristics

BFS is structured as a level-synchronous algorithm:

- Nodes in the same graph level are processed concurrently.
- A queue (or "frontier") tracks which nodes to explore next.
- Each node is visited once to prevent redundant processing.

Sequential BFS is straightforward but inherently serial due to the level-by-level dependency between nodes. GPU parallelization requires restructuring the traversal to exploit data parallelism across nodes within the same frontier.

11.7.4 Sequential BFS algorithm

Let's first understand how BFS works sequentially before parallelizing it.

11.7.4.1 Example graph

Consider a simple graph with four nodes:

```
  R (root)
 / \
A   B
 \ /
  C
```

11.7.4.2 Step-by-step execution

Step 1: Start at the root node `R`

- Mark `R` as visited
- Enqueue `R`
- Queue: [R]

Step 2: Process `R`

- Dequeue `R`
- Discover neighbors: `A` and `B`
- Enqueue both, mark as visited
- Queue: [A, B]

Step 3: Process `A`

- Dequeue `A`
- Neighbors: `R` (visited) and `C` (new)
- Enqueue `C`
- Queue: [B, C]

Step 4: Process `B`

- Dequeue `B`
- Neighbors: `R` (visited) and `C` (visited)
- Queue: [C]

Step 5: Process `C`

- Dequeue `C`
- All neighbors visited
- Queue becomes empty — traversal complete

11.7.5 Parallel BFS on GPU

Unlike dense linear algebra, BFS is an **irregular** algorithm. The amount of work per node varies, and the connectivity pattern of the graph drives execution. The main challenges are:

1. **Data dependencies**: nodes in the next level depend on the previous level.
2. **Irregular parallelism**: each frontier may contain a very different number of nodes.
3. **Dynamic workload**: the size of the next frontier is unknown at runtime.

4. **Synchronization**: all nodes in one frontier must complete before the next begins.

The **frontier** is the set of nodes being processed at a given BFS level. Parallel BFS executes all frontier nodes simultaneously, using one thread per node to discover new neighbors and mark them for the next iteration.

11.7.5.1 Implementation strategy

The GPU implementation performs BFS using **two cooperating kernels**:

1. **Kernel 1**: processes all nodes in the current frontier.
2. **Kernel 2**: updates the next frontier and checks if work remains.

This design provides **implicit synchronization** between levels while avoiding race conditions. The host (CPU) manages the iterative control loop, launching kernels repeatedly until no more frontier nodes exist.

11.7.6 Data structures

The graph is represented using adjacency lists stored in arrays:

```
struct Node {
    int starting;       // starting index in the edge list
    int no_of_edges;    // number of outgoing edges
};
```

Main arrays:

- `g_graph_nodes`: node array storing offsets into the edge list.
- `g_graph_edges`: flattened list of edge destinations.
- `g_graph_mask`: boolean array indicating active frontier nodes.
- `g_updating_graph_mask`: marks nodes to be added to the next frontier.
- `g_graph_visited`: tracks which nodes were visited.
- `g_graph_cost`: stores the distance (edge count) from the source node.

Control flow flags:

- `g_over`: device-side flag indicating whether another iteration is needed.
- The host resets this flag each iteration and checks it after kernel execution.

11.7.7 The two-kernel approach

The two-kernel structure ensures correctness and efficient synchronization:

- **Exploration kernel (Kernel 1)** discovers new nodes.
- **Update kernel (Kernel 2)** finalizes state for the next iteration.

This separation:

- Avoids race conditions between threads of different levels.
- Provides synchronization between BFS levels.
- Keeps control logic simple on the host side.

11.7.7.1 Kernel 1: process current frontier

Each thread processes one node from the current frontier, examining all of its outgoing edges:

```
__global__ void Kernel1(
    Node* g_graph_nodes,
    int* g_graph_edges,
    bool* g_graph_mask,
    bool* g_updating_graph_mask,
    bool* g_graph_visited,
    int* g_graph_cost,
    int no_of_nodes)
{
    int tid = hipBlockIdx_x * MAX_THREADS_PER_BLOCK + hipThreadIdx_x;

    if (tid < no_of_nodes && g_graph_mask[tid]) {
        g_graph_mask[tid] = false;

        for (int i = g_graph_nodes[tid].starting;
             i < g_graph_nodes[tid].starting + g_graph_nodes[tid].no_of_edges;
             i++) {
            int id = g_graph_edges[i];
            if (!g_graph_visited[id]) {
                g_graph_cost[id] = g_graph_cost[tid] + 1;
                g_updating_graph_mask[id] = true;
            }
        }
    }
}
```

Kernel 1 responsibilities:

- Clear the node's mask (mark processed).
- Explore all edges.
- For each unvisited neighbor:
  - Compute cost (distance).
  - Add to the next frontier.

11.7.7.2 Kernel 2: update frontier

This kernel finalizes the next frontier:

```
__global__ void Kernel2(
    bool* g_graph_mask,
    bool* g_updating_graph_mask,
    bool* g_graph_visited,
    bool* g_over,
    int no_of_nodes)
{
    int tid = hipBlockIdx_x * MAX_THREADS_PER_BLOCK + hipThreadIdx_x;

    if (tid < no_of_nodes && g_updating_graph_mask[tid]) {
        g_graph_mask[tid] = true;
```

(continues on next page)

(continued from previous page)

```
        g_graph_visited[tid] = true;
        *g_over = true;
        g_updating_graph_mask[tid] = false;
    }
}
```

Kernel 2 responsibilities:

- Move newly discovered nodes into the active frontier.
- Mark them as visited.
- Signal continuation via `*g_over`.

11.7.8 Host-side control loop

```
do {
    h_over = false;
    hipMemcpy(d_over, &h_over, sizeof(bool), hipMemcpyHostToDevice);

    Kernel1<<<num_blocks, MAX_THREADS_PER_BLOCK>>>(
        d_graph_nodes, d_graph_edges, d_graph_mask,
        d_graph_updating_graph_mask, d_graph_visited,
        d_graph_cost, no_of_nodes);
    hipDeviceSynchronize();

    Kernel2<<<num_blocks, MAX_THREADS_PER_BLOCK>>>(
        d_graph_mask, d_graph_updating_graph_mask,
        d_graph_visited, d_over, no_of_nodes);
    hipDeviceSynchronize();

    hipMemcpy(&h_over, d_over, sizeof(bool), hipMemcpyDeviceToHost);
} while (h_over);
```

The loop exits when no new nodes are discovered. `g_over` or `h_over` on host side remains `false` after one full iteration.

11.7.9 Performance characteristics

This section examines the performance characteristics of the parallel BFS implementation.

11.7.9.1 Parallelism patterns

Within each iteration:

- High parallelism: All frontier nodes processed simultaneously
- Work distribution: One thread per node

Across iterations:

- Sequential: Must complete one level before starting the next
- Variable parallelism: Different levels may have different numbers of nodes

11.7.9.2 Workload characteristics

Characteristic	Description
Irregular	Frontier size varies dramatically across levels
Data-dependent	Graph structure determines parallel work available
Dynamic	Cannot predict workload statically
Memory-bound	Many memory accesses per computation

11.7.10 Best practices

This section outlines recommended practices for implementing an efficient GPU-accelerated breadth-first search (BFS). It highlights design principles, memory-management strategies, and debugging techniques that help ensure correctness, maintainability, and high performance when mapping BFS onto modern GPU architectures.

11.7.10.1 Design principles

1. **Define clear kernel roles**

   Decompose BFS into well-defined GPU kernels, each responsible for a specific phase of computation. For example:

   - **Kernel 1**: frontier expansion (discovering new nodes)
   - **Kernel 2**: frontier update (marking next-level nodes)

   This separation simplifies synchronization and ensures that each kernel operates on independent data regions.

2. **Minimize host–device communication**

   Keep graph data structures (nodes, edges, masks) resident on the GPU across iterations. Only transfer lightweight control flags such as `g_over` to the host each loop iteration to check termination conditions.

3. **Kernel boundaries as synchronization points**

   Kernel launch boundaries on the same stream naturally enforce global synchronization across all threads on the GPU. Each kernel invocation completes before the next begins, ensuring that:

   - All nodes in the current frontier are fully processed before updating the next frontier.
   - Memory updates to arrays like `g_graph_cost` or `g_graph_mask` are visible to all threads in subsequent kernels.

   This avoids the need for costly device-wide barriers or explicit synchronization primitives within a single kernel. Leverage kernel sequencing to structure iterative algorithms cleanly—each kernel represents one computation phase per BFS level.

4. **Flag-based control**

   Use device-side flags for dynamic termination and conditional control flow. In BFS, the Boolean flag `g_over` serves as a device-to-host signal indicating whether new nodes were discovered during the current iteration.

   - Initialize `g_over` to `false` on the host at the start of each iteration.
   - Allow GPU threads in **Kernel 2** to set `*g_over = true` when adding new nodes to the next frontier.
   - After kernel completion, copy the flag back to the host using `hipMemcpy()`. If `g_over` remains false, the traversal is complete.

   This mechanism avoids repeated host intervention and enables a tight CPU–GPU control loop that dynamically adapts to workload size without transferring large data structures.

11.7.10.2 Memory strategy

1. **Persistent device allocations**

   Allocate all required device buffers once prior to traversal. Reuse these allocations across multiple BFS runs or multiple source nodes to minimize the overhead of repeated `hipMalloc()` and `hipFree()` calls.

2. **Minimize host–device communication**

   Keep graph data structures (nodes, edges, masks) resident on the GPU across iterations. Only transfer lightweight control flags such as `g_over` to the host each loop iteration to check termination conditions.

3. **Use pinned host memory for control flags**

   When copying `g_over` or other control signals between host and device, allocate host memory using pinned (page-locked) buffers to accelerate DMA transfers.

11.7.10.3 Debugging and validation

1. **Frontier validation**

   After each iteration, verify the number of nodes marked in `g_graph_mask`. Unexpected empty or overfull frontiers often indicate incorrect synchronization or uninitialized masks.

2. **Termination condition check**

   Confirm that the host-side loop terminates when `g_over` remains false for one iteration. If the loop never ends, ensure `g_over` is reset on the host before each kernel launch.

3. **Result verification**

   Compare computed distances in `g_graph_cost` against a CPU reference implementation for small graphs to validate correctness.

4. **Profiling and bottleneck detection**

   Use tools such as rocprofv3 or ROCm compute profiler to measure per-kernel execution times, memory throughput, and synchronization overhead.

5. **Logging and debug builds**

   Enable optional logging for iteration counts, frontier sizes, and synchronization states during development. Disable logging in production builds to avoid performance impact.

11.7.11 Conclusion

Multi-kernel GPU programming is essential for complex algorithms that require:

- Multiple phases of computation.
- Data dependencies between phases.
- Dynamic control flow based on intermediate results.

The BFS example demonstrates:

- How to decompose algorithms into multiple cooperating kernels.
- Techniques for managing frontiers and iterative processing.
- Strategies for handling irregular and dynamic parallelism.
- Proper synchronization between kernel launches.

Key takeaways:

1. **Kernel boundaries provide synchronization**: Use them strategically to ensure correctness.

2. **Separate exploration from update**: Prevents race conditions in level-based algorithms.
3. **Host controls iteration**: CPU manages the overall loop while GPU does heavy lifting.
4. **Flags enable dynamic control**: Device-side flags allow work-dependent termination.

Understanding multi-kernel patterns enables developers to implement sophisticated algorithms like graph processing, dynamic programming, and iterative refinement methods efficiently on GPUs.

CHAPTER

TWELVE

HIP COMPILERS

AMD GPUs. The compilers set up the default libraries, and include paths for the HIP and ROCm libraries, along with required environment variables. For more information, see the ROCm compiler reference.

12.1 Compilation workflow

HIP provides a flexible compilation workflow that supports both offline compilation and runtime or just-in-time (JIT) compilation. Each approach has advantages depending on the use case, target architecture, and performance needs.

The offline compilation is ideal for production environments, where the performance is critical, and the target GPU architecture is known in advance.

The runtime compilation is useful in development environments or when distributing software that must run on a wide range of hardware without the knowledge of the GPU in advance. It provides flexibility at the cost of some The runtime compilation is useful in development environments or when distributing software that must run on a wide range of hardware without prior knowledge of the GPU. It provides flexibility at the cost of some performance overhead.

12.1.1 Offline compilation

Offline compilation is performed in two steps: host and device code compilation.

- Host-code compilation: On the host side, `amdclang++` or `hipcc` can compile the host code in one step without other C++ compilers.
- Device-code compilation: The compiled device code is embedded into the host object file. Depending on the platform, the device code can be compiled into assembly or binary.

For an example on how to compile HIP from the command line, see *SAXPY tutorial* .

12.1.2 Runtime compilation

HIP allows you to compile kernels at runtime using the `hiprtc*` API. Kernels are stored as a text string, which is passed to HIPRTC alongside options to guide the compilation.

For more information, see HIP runtime compiler.

12.2 Target GPU architectures (GFX IP)

Instructions in the AMDGPU instruction set architecture (ISA) are compatible only with certain physical GPU architectures. The versioning system that abstracts hardware details from compilers and software is called the GFX IP version. Each AMD GPU family, such as RDNA or CDNA, defines its own GFX IP identifiers. These identifiers specify which instruction formats, memory models, and compute features are supported by that generation of hardware.

A GFX version is expressed as a short string, for example:

- `gfx90a` — CDNA2 (MI250 series)
- `gfx942` — CDNA3 (MI300 series)
- `gfx1100` — RDNA3 (RX 7900 series)

Most GFX IP versions are composed of three numerical fields, which act roughly like a major-minor-subminor versioning system:

- The major version corresponds to the architectural family (for example, CDNA3 versus RDNA3).
- The minor version encodes core feature updates within that family (for example, new *MFMA* or *LDS* capabilities).
- The subminor version may specify packaging or stepping differences (for example, MI300A versus MI300X).

Each new generation introduces additional hardware features, for example, mixed-precision MFMA instructions, asynchronous data movement engines, and updated cache hierarchies, while maintaining broad forward compatibility for code compiled at the intermediate representation (IR) level. When compiling GPU programs with `amdclang++`, the target GFX architecture determines the ISA and hardware features available to the kernel.

For example, compiling for the MI300X would use:

```
amdclang++ --offload-arch=gfx942 kernel.cpp -o kernel.out
```

The compiler uses this flag to emit optimized machine code for that GPU's *compute units*, *matrix cores*, and memory subsystem. The GFX IP acts as a virtual hardware target, decoupling high-level programming models (HIP, OpenMP, OpenCL) from the underlying physical GPU design.

While AMD does not explicitly name its compatibility model as "onion-layered," the GFX IP system follows a similar principle: newer architectures generally extend, rather than break, existing instruction sets.

Thus, code compiled for an older architecture (for example, `gfx90a`) can often be re-optimized or recompiled for a newer one (`gfx942`) with minimal modification. However, compatibility is not guaranteed across major GFX families, since those may introduce fundamentally new pipelines or execution semantics.

12.3 AMDGPU assembly

AMDGPU assembly, sometimes called GFX ISA (Instruction Set Architecture), is the low-level assembly format for programs running directly on AMD GPUs. It is the lowest-level human-readable representation of GPU instructions, typically generated by the ROCm compiler toolchain and converted into binary microcode for execution on the device.

Each AMD GPU architecture family, such as RDNA or CDNA, defines its own GFX ISA version. For example:

- `gfx90a` corresponds to CDNA2 (MI250 series)
- `gfx942` corresponds to CDNA3 (MI300 series)
- `gfx1100` corresponds to RDNA3 consumer GPUs

Each version defines its unique instruction encodings, supported data types, and pipeline behavior. Compiled GPU kernels must target a specific *GFX architecture* to ensure instruction compatibility and optimal scheduling.

AMDGPU assembly is rich and expressive, exposing every level of GPU behavior: scalar operations, vector math, memory access, synchronization, and matrix-multiply instructions. A few examples of instructions from the `gfx942` architecture include:

- `v_add_f32 v0, v1, v2` — add 32-bit floats in vector registers
- `s_mov_b32 s4, 0x3f800000` — move immediate value (1.0f) into a scalar register
- `v_mfma_f32_16x16x4f16 v[0:15], v[16:31], v[32:47], v[0:15]` — perform a 16×16×4 matrix fused multiply-add

- `s_barrier` — synchronize all warps in the work-group

In this syntax:

- `v_` instructions operate on vector registers (VGPRs) per thread.
- `s_` instructions operate on scalar registers (SGPRs) shared by the warp.
- Specialized matrix instructions (`v_mfma_*`) invoke the *Matrix Core (MFMA) hardware units*.

While it is possible to write AMDGPU assembly by hand, this practice is rare. In most cases, developers review compiler-generated assembly to optimize kernels, analyze register pressure, or refine memory-access behavior.

The ROCm disassembler (`llvm-objdump`) and ROCm profiler (`rocprofv3`) allow inspection of generated GFX ISA alongside high-level HIP or OpenMP source code. Because the instruction semantics and binary encodings are publicly documented by AMD, the GFX ISA is a fully open, compiler-targetable standard.

This openness allows tool developers, performance engineers, and researchers to reason about GPU behavior at the instruction level, bridging the gap between hardware and high-level kernel code.

12.4 AMDGPU intermediate representation

AMDGPU intermediate representation (AMDGPU IR) is an intermediate representation for code that runs on AMD GPUs and other parallel processors. It is one of the key outputs of the ROCm compiler toolchain, produced by `amd-clang++` before being translated into architecture-specific GPU assembly (*GFX ISA*).

AMD documentation refers to this layer as both a virtual instruction set and a target-specific dialect of LLVM IR. From a programmer's perspective, AMDGPU IR defines a virtual machine model for parallel thread execution: compilers and optimizers emit this IR with the expectation that it will execute with consistent semantics across multiple generations of AMD hardware, including future architectures not yet released.

This makes AMDGPU IR a "narrow waist" between software and hardware, abstracting the details of the physical *compute units*, *warp* schedulers, and memory hierarchies, while still providing explicit control over threads, registers, and memory spaces.

Unlike traditional CPU ISAs, AMDGPU IR is not executed directly. Instead, it is further compiled (either just-in-time or ahead-of-time) into GFX ISA, the device-specific binary that runs on a given GPU architecture (for example, `gfx942` for CDNA3 MI300X or `gfx1200` for RDNA3). This multi-stage compilation allows forward compatibility: kernels compiled for one generation can often be re-optimized and executed efficiently on newer hardware through updated ROCm runtimes and drivers. Some exemplary fragments of AMDGPU IR:

```
; declare usage of 64 vector registers
!amdgpu.num_vgpr = !{i32 64}

; perform fused multiply-add
%f5 = call float @llvm.fma.f32(float %f3, float %f4, float 1.5)

; read thread and workgroup indices
%tid  = call i32 @llvm.amdgcn.workitem.id.x()
%wgid = call i32 @llvm.amdgcn.workgroup.id.x()
```

These instructions represent operations in the virtual machine model:

- Registers are virtual VGPRs/SGPRs dynamically mapped to physical hardware registers by the compiler.
- Arithmetic and memory intrinsics (`llvm.fma`, `llvm.amdgcn.buffer.load`) map one-to-one to GPU instructions.
- Built-in functions like `llvm.amdgcn.workitem.id.x()` access special per-thread state, such as the current thread or *block* index.

The AMDGPU IR machine model reflects the hardware reality of AMD GPUs: a single instruction stream drives a *wavefront* of 64 threads that execute in lockstep on the SIMD pipelines, each maintaining its own register state while sharing program flow and *local memory*. Warps cooperate through the *Local Data Share (LDS)* and synchronize via barriers, the same abstractions exposed in the high-level ROCm programming model.

Since AMDGPU IR is integrated with the open-source LLVM compiler infrastructure, its syntax and semantics are well-documented and publicly accessible. Developers can inspect, modify, or emit AMDGPU IR directly for performance analysis, research, or custom toolchains.

12.5 ROCm binary utilities

The ROCm binary utilities are a collection of command-line tools for examining and manipulating GPU binaries produced by `amdclang++` or other ROCm build tools. These utilities allow developers to inspect, disassemble, and analyze AMDGPU code objects, the compiled GPU kernels embedded in host executables or distributed as standalone `.hsaco` files. Previous ROCm versions shipped with the `roc-obj-ls`, `roc-obj-extract`, and `roc-obj utilities`. These have been deprecated in favor of enhanced capabilities now available in standard LLVM object tools, such as `llvm-objdump` and `llvm-readobj`. For more information, see llvm-objdump — LLVM's object file dumper at.

The `llvm-objdump` utility provides multiple capabilities for analyzing GPU binaries. With the `--offloading` flag, it can list and extract information from the contents of ROCm binaries, including code object metadata, kernel symbols, target architectures (for example, `gfx90a`, `gfx1100`), and linkage details. It supports both standalone `.hsaco` files and "fat binaries" stored as embedded objects within host executables. In current releases, the tool additionally extracts HIP fat-binary bundle entries into standalone code object files. With the `--triple=amdgcn` option, it can disassemble GPU kernels into readable AMDGPU ISA, allowing detailed examination of instruction streams, register usage, and control-flow structure. Such capabilities are critical for performance tuning, correctness verification, and low-level kernel analysis, including tasks such as optimizing *MFMA* instructions or assessing compiler-generated transformations.

The `llvm-readobj` tool, used with the `--offloading` flag, provides a complete listing of HIP fat-binary offload bundle entries, superseding the earlier `roc-obj-ls` utility.

Together, these utilities provide developers with insight into how HIP C++ code is compiled, optimized, and mapped to GPU hardware. They complement profiling tools like `rocprofv3` by exposing the static structure of compiled GPU binaries.

12.6 Static libraries

`amdclang++` supports generating two types of static libraries.

- The first type of static library only exports and launches host functions within the same library and not the device functions. This library type offers the ability to link with another compiler such as `gcc`. Additionally, this library type contains host objects with device code embedded as fat binaries. This library type is generated using the flag `--emit-static-lib`:

```
amdclang++ hipOptLibrary.cpp --emit-static-lib -fPIC -o libHipOptLibrary.a
gcc test.cpp -L. -lhipOptLibrary -L/path/to/hip/lib -lamdhip64 -o test.out
```

- The second type of static library exports device functions to be linked by other code objects by using `amdclang++` as the linker. This library type contains relocatable device objects and is generated using `ar`:

```
amdclang++ hipDevice.cpp -c -fgpu-rdc -o hipDevice.o
ar rcsD libHipDevice.a hipDevice.o
amdclang++ libHipDevice.a test.cpp -fgpu-rdc -o test.out
```

Examples of this can be found in rocm-examples under static host libraries or static device libraries.

CHAPTER

THIRTEEN

UNDERSTANDING GPU PERFORMANCE

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*.

13.1 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.

13.2 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.

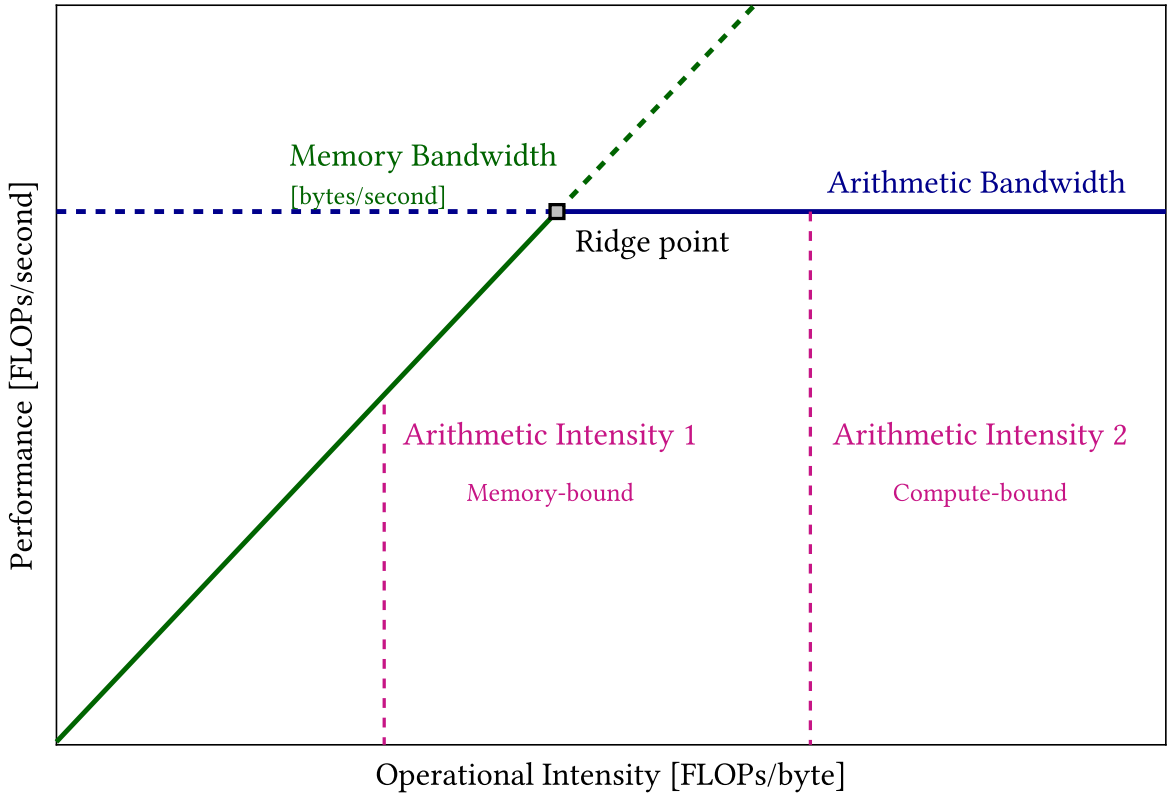


Fig. 1: Roofline model showing the relationship between arithmetic intensity and achievable performance. The memory bandwidth ceiling represents the GPU's memory bandwidth limit, while the compute ceiling shows the maximum achievable TFLOPs. Kernels falling into the area to the left of the ridge point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

13.3 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))

13.4 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

13.5 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

13.5.1 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

13.6 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 *warps* keeping overall throughput high even when individual instructions take many cycles. When one warp stalls on a slow *global-memory* access, the *scheduler* immediately issues instructions from another eligible warp.

How latency hiding works:

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

This keeps the *compute units* busy: while one warp 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 warps with independent work. The number of instructions required from other warps to hide latency depends on the GPU's specific memory latency and instruction throughput characteristics.

13.6.1 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, *warp* switching by the warp schedulers is the primary latency-hiding mechanism.

Little's Law determines how many independent memory transactions or instructions must be outstanding across active warps 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 warp granularity (64 threads per warp on CDNA). When latency hiding is successful, the CU maintains enough active warps 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 warps)
- Independent instructions to overlap
- Balanced resource usage
- Minimal divergence

13.7 Warp (Wavefront) execution states

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

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

An **eligible** warp is an active warp ready to issue its next instruction. For a warp 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 warps are the immediate candidates for issue. A lack of eligible warps often indicates dependency or memory stalls—a key target in performance tuning.

A **stalled** warp 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 warp. When waiting on *LDS* or ALU results, a warp 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 warps (thread-level parallelism) rather than within them (instruction-level parallelism).

A **selected** warp is an eligible one chosen by the warp 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 warps 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 warps resident so that the schedulers always have work to select, ensuring the CU pipelines remain fully utilized.

13.8 Occupancy theory

Occupancy measures the ratio of active *warp* to the maximum possible warps on a *compute unit*.

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

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 warps 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 warps.

Why occupancy matters:

- Higher occupancy improves *latency hiding*
- More concurrent warps 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
- **Warp slots**: Hardware limit on concurrent warps
- **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 warps 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 warp—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 warps are needed to fully saturate the MFMA and memory pipelines.

13.9 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

13.9.1 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 *warp* access addresses that fall into the same burst, those logical accesses can be coalesced into that single transaction.

This fits naturally with the warp execution model: in normal execution, all threads in a warp execute the same instruction at the same time. If each lane in the warp loads or stores a value from a contiguous region (e.g., lane *i* accesses `base + i`), the memory system can typically serve the entire warp'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.

13.9.1.1 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 warp, 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 warp 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 warp's loads spread across more bursts
- The number of physical transactions per logical access increases
- Effective bandwidth drops

13.9.2 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 *warp* 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)

13.9.2.1 Example: conflict-free access

If you access sequential elements of a float array in LDS, different lanes in a warp 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.

13.9.2.2 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 warp 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.

13.9.2.3 When conflicts don't happen

If all threads in a warp 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.

13.10 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.

13.10.1 How register pressure arises

Each thread in a *warp* 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 warps
- Fewer registers per thread → higher occupancy but may need memory spills
- Optimal balance depends on kernel memory access patterns

13.11 Performance metrics explained

Understanding performance metrics is essential for analyzing GPU behavior:

13.11.1 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.

13.11.2 Utilization metrics

13.11.2.1 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 *warps* and how close the kernel is to saturating the hardware's arithmetic or memory throughput.

13.11.2.2 Issue efficiency

Issue efficiency measures how effectively the warp scheduler on each compute unit keeps the execution pipelines busy by issuing instructions from eligible warps. 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 warp was eligible and an instruction was successfully issued. Lower values indicate that during some cycles, all active warps 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 warp 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.

13.11.2.3 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 warps.

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 warps. High CU utilization indicates good parallel workload distribution, while low CU utilization may point to poor occupancy, launch configuration limits, or insufficient concurrency.

13.11.2.4 Branch efficiency

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

It quantifies control-flow uniformity—that is, how often all lanes in a *warp* 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 warp 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 warps consist entirely of threads that either satisfy `idx < n` or not—except perhaps for the last partial warp, 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 warp 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.

13.12 Theoretical performance limits

Understanding theoretical limits helps set realistic performance expectations.

13.12.1 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)

13.12.2 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.

13.13 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 warp switching
- **Occupancy**: How warp 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

FOURTEEN

PERFORMANCE GUIDELINES

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*.

14.1 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.

14.2 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.

14.2.1 rocprofv3

The tool rocprofv3 provides command-line-driven profiling for detailed performance analysis. It collects metrics on kernel execution time, memory bandwidth, warp 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
- Warp occupancy metrics
- Compute unit utilization
- Instruction-level performance counters
- API call tracing
- Hardware event collection

14.2.1.1 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

14.2.2 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

14.2.3 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
```

14.2.4 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.

14.3 Parallel execution

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

14.3.1 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)

14.3.2 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)

14.3.3 Multiprocessor level

Maximize parallel execution within each *compute unit*:

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

14.4 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.

14.4.1 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);
}
```

14.4.2 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 warp 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 warp 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 warp-aligned
```

Coalesce memory accesses

When consecutive threads in a warp 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 *warp* 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);
```

14.5 Instruction throughput optimization

14.5.1 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
```

14.5.2 Control flow optimization

Minimize divergence

When threads in a warp 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-warp execute
}

// Bad: divergence within warp
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 warps

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
```

14.5.3 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);
```

14.6 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 warps 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*.

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

```
// Use multiples of warp size
dim3 block(64);    // Good for AMD Instinct GPUs (warp=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 *warps* to maximum possible warps 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*.

14.8 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);
```

14.9 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 *warps* coherent

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

CHAPTER

FIFTEEN

OPTIMIZING PERFORMANCE

Performance optimization is essential for unlocking the full potential of AMD GPUs in HIP applications. While writing correct GPU code is the first step, achieving optimal performance requires understanding GPU architecture, memory hierarchies, and execution patterns. Modern AMD GPUs offer massive parallel processing capabilities with thousands of computing cores and high-bandwidth memory systems. However, realizing this potential requires careful attention to how kernels are structured, how memory is accessed, and how computational resources are utilized.

This collection of tutorials demonstrates proven optimization techniques that can dramatically improve application performance. The optimization techniques presented here address common performance bottlenecks and provide practical strategies for improvement.

15.1 Performance optimization challenges

Optimizing GPU applications presents unique challenges that differ from traditional CPU optimization:

- **Performance portability**: Optimization techniques that work well on one GPU architecture may not yield similar results on another due to architectural differences.
- **Memory bandwidth limitations**: GPU performance is often constrained by memory bandwidth rather than computational throughput, requiring careful attention to memory access patterns.
- **Thread organization**: The way threads are organized into blocks and grids significantly impacts performance, with optimal configurations varying by workload and hardware.
- **Resource utilization**: Achieving high GPU utilization requires balancing multiple factors including occupancy, memory coalescing, and instruction throughput.
- **Profiling complexity**: Understanding performance bottlenecks requires systematic measurement and analysis using specialized profiling tools.

15.2 Optimization principles

Effective GPU optimization follows several key principles:

Start with correctness: Always verify that your code produces correct results before optimizing. Performance improvements are meaningless if the output is incorrect.

Measure before optimizing: Use profiling tools to identify actual bottlenecks rather than optimizing based on assumptions. The most time-consuming portions of your code deserve the most attention.

Optimize iteratively: Apply one optimization technique at a time and measure its impact. This approach helps you understand which techniques are most effective for your specific workload.

Consider the target architecture: Different AMD GPU architectures (CDNA, RDNA) have different characteristics. Optimization strategies should account for the specific hardware you're targeting.

Balance multiple factors: GPU performance depends on the interplay of occupancy, memory bandwidth, instruction throughput, and other factors. Optimizing one aspect may negatively impact another, requiring careful balance.

15.2.1 Performance analysis workflow

Effective performance optimization requires systematic measurement and analysis. The recommended workflow for optimizing HIP applications includes:

1. **Profile your application** using tools like rocprofv3, the ROCm Compute Profiler, or the ROCm Systems Profiler to identify performance bottlenecks and collect execution traces.
2. **Analyze the profiling data** to understand kernel execution times, memory transfer overhead, and GPU utilization patterns.
3. **Apply optimization techniques** based on your analysis, focusing on the most impactful bottlenecks first.
4. **Measure and validate** improvements by re-profiling your optimized code to ensure changes have the desired effect.

15.2.2 Prerequisites

To get the most from these tutorials, you should have:

- Understanding of HIP programming fundamentals (see *SAXPY Tutorial - Hello HIP*).
- Familiarity with GPU architecture concepts (see Hardware implementation).
- HIP runtime environment installed (see *ROCm install*).
- Basic knowledge of performance profiling concepts (recommended).

15.3 Optimization tutorials

This collection provides comprehensive tutorials on essential HIP performance optimization techniques:

- *Highly parallel workloads*: Optimizing embarrassingly parallel algorithms like image processing.
- *Fixed-sized kernels*: Reducing thread dispatch overhead through fixed kernel dimensions.
- *Reduction operations*: Efficient parallel reduction algorithms using shared memory.
- *Tiling and reuse*: Leveraging local data share memory to improve matrix multiplication performance.
- *Tiling and coalescing*: Converting non-coalesced memory access patterns to coalesced ones for better bandwidth utilization.

15.3.1 Getting started

Each tutorial builds upon concepts from previous ones. Follow the order presented for optimal learning:

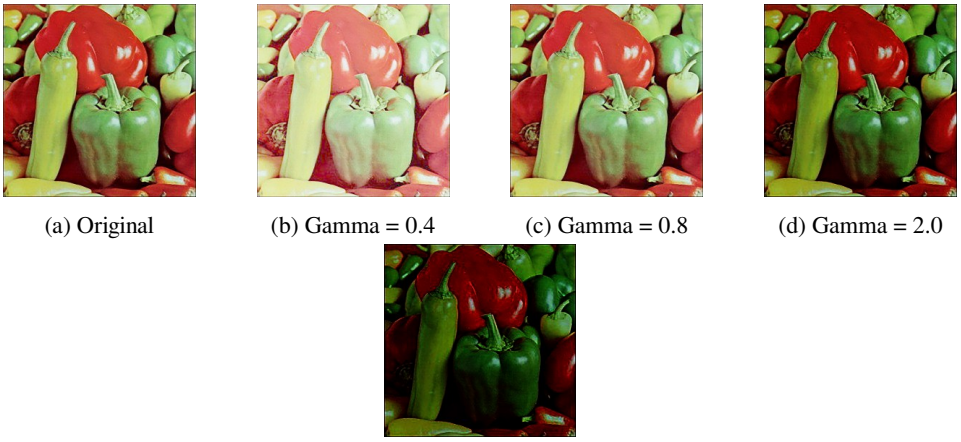
1. **Start with highly parallel workloads** to understand block size selection and thread organization fundamentals.
2. **Progress to fixed-sized kernels** to learn techniques for reducing thread dispatch overhead.
3. **Study reduction operations** to understand efficient parallel aggregation patterns.
4. **Explore tiling and data reuse** to leverage local data share memory for improved performance.
5. **Master memory coalescing** to optimize memory bandwidth utilization and avoid non-coalesced access patterns.

Each tutorial includes complete code examples, performance measurements, and detailed explanations of the optimization techniques applied. By working through these tutorials systematically, you'll develop the skills needed to write high-performance HIP applications for AMD GPUs.

15.4 Highly parallel workload: image gamma correction

A GPU typically has thousands of computing cores that run in parallel. These cores are best utilized when each calculates one partition of the output results so that they do not need to communicate with other cores. Such algorithms are often classified as "embarrassingly parallel," as the required programming effort is minimal. In the earlier `vector_add` example (see *SAXPY Tutorial - Hello HIP*), you identified an embarrassingly parallel workload example. In this section, you will explore another one: image gamma correction.

15.4.1 Computational challenges in pixel-wise operations



(a) Original (b) Gamma = 0.4 (c) Gamma = 0.8 (d) Gamma = 2.0

(e) Gamma = 4.0

Fig. 1: Image gamma correction algorithm. A gamma value smaller than one renders the image more brightly, whereas a value greater than one renders a darker image.

Image gamma correction is a common image processing algorithm that adjusts the brightness of an image without changing the content. As shown in the figure above, when you apply a gamma value of less than one, the image becomes brighter, whereas a value greater than one darkens it.

Pictures stored on a computer are typically encoded using pixels (see figure below), which are square regions of an image that are each rendered with a single color. For example, the images used in the figures above and below have 267 (width) × 267 (height) pixels total. If the image is in color, three values representing red, green, and blue (RGB) would be required for each pixel to represent the brightness. Brightness is coded using an 8-bit integer [0–255], where zero represents the darkest shade, and 255 represents the brightest. The other key method uses a floating-point number between zero and one. The red brightness values of all pixels comprise the "red channel". A colored image typically has three channels (RGB).

$$V_{out} = V_{in}^{\gamma}$$

Image gamma correction is an element-wise operation that applies simple floating-point manipulation to all pixels and their channels. As shown in the equation above, the gamma value is applied as a power to the brightness value. A greater-than-one gamma decreases the brightness value. As suggested by the equation, the number of output values matches the number of input values, and the calculation of each output value is independent of the calculation of neighboring values. The image gamma correction algorithm is embarrassingly parallel and is therefore suitable for GPUs.

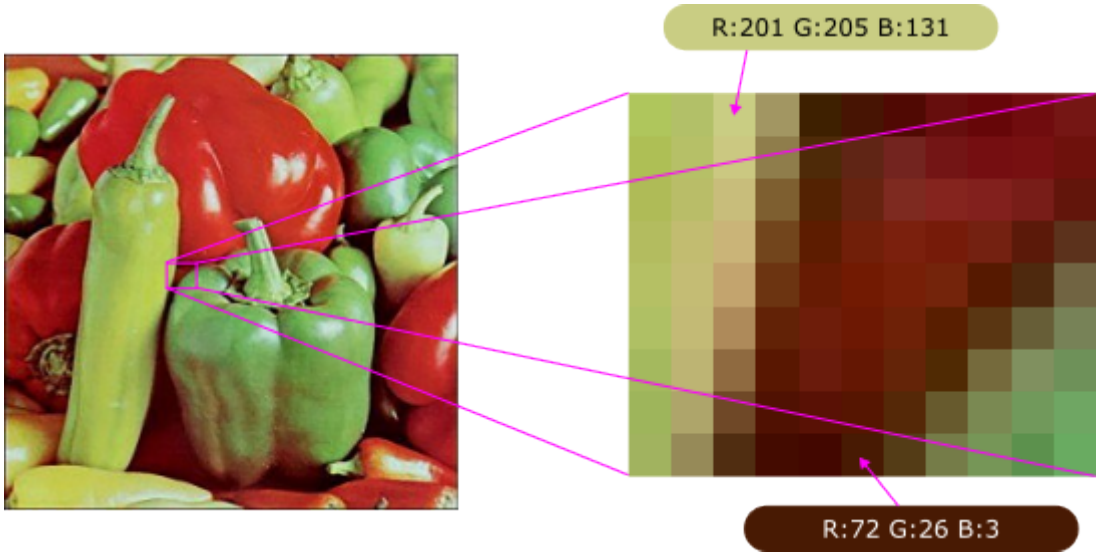


Fig. 2: Computerized image organized using pixels. A color image usually requires RGB brightness values to represent the final pixel color.

15.4.2 GPU kernel design and optimization strategies

The following listing presents a simple implementation of the gamma correction algorithm using HIP.

Listing 1: Image gamma correction application as an embarrassingly parallel algorithm. Each thread is responsible for processing a value in the array of image pixels. The grid size reflects the number of pixels.

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

#include <cstdint>
#include <cstdlib>

__global__ void image_gamma(std::uint8_t* d_image, float gamma, int num_values)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < num_values)
    {
        d_image[idx] = powf(d_image[idx] / 255.f, gamma) * 255.f;
    }
}

int main()
{
    int width, height, channels, num_values;
    std::uint8_t* data;

    // Load an image from file.

    int blockSize = 256;
    int gridSize = (num_values + blockSize - 1) / blockSize;

    float gamma = 4.f;
```

(continues on next page)

(continued from previous page)

```
26      image_gamma<<<gridSize, blockSize>>>(d_image, gamma, num_values);
27
28      hipMemcpy(
29          data, d_image, num_values * sizeof(std::uint8_t), hipMemcpyDeviceToHost
30      );
31
32      // Save the image to an output file.
33
34      hipFree(d_image);
35      return EXIT_SUCCESS;
36  }
```

To address this GPU programming problem, first consider how to map the threads to the input and output elements. For embarrassingly parallel algorithms, the mapping is straightforward because each thread is responsible for a part of the output channel (a pixel). Therefore, the total number of threads will equal the height × width × number of channels = `num_values`. To calculate the grid size, divide `num_values` by the block size. If the block size is not a multiple of `num_values`, you can round up to the next integer, as shown in line 23 of the code sample above.

The block size can be between 1 and 1,024, depending on the device limitation. Note that selecting a suitable block size will improve the overall performance. According to the figure below, the performance is best when the block size is a multiple of 32, because RDNA GPUs organize groups of 32 threads in a wavefront, which is the smallest resource allocation unit.

> **Note**
>
> On CDNA GPUs, the wavefront size is 64.

Otherwise, the CUs will be forced to allocate more resources for each block, resulting in performance degradation. The selection of the block size as a multiple of the wavefront size is likely to optimize embarrassingly parallel applications with AMD GPUs.

15.4.3 Performance benefits and application patterns

The GPU implementation of image gamma correction demonstrates significant performance advantages over CPU-based approaches. By leveraging thousands of parallel threads, you can process millions of pixels simultaneously, making this technique particularly effective for high-resolution images and real-time video processing applications.

The embarrassingly parallel nature of gamma correction makes it an ideal candidate for GPU acceleration. Similar patterns appear in many other image processing operations, including:

- Color space conversions (RGB to grayscale, HSV transformations)
- Brightness and contrast adjustments
- Image filtering with separable kernels
- Histogram equalization
- Pixel-wise arithmetic operations

When you encounter algorithms where each output element depends only on a corresponding input element (or a small, fixed neighborhood), consider applying the same parallelization strategy demonstrated here. The key characteristics that make a workload suitable for this approach include:

- Independent calculations for each output element
- Minimal or no inter-thread communication requirements

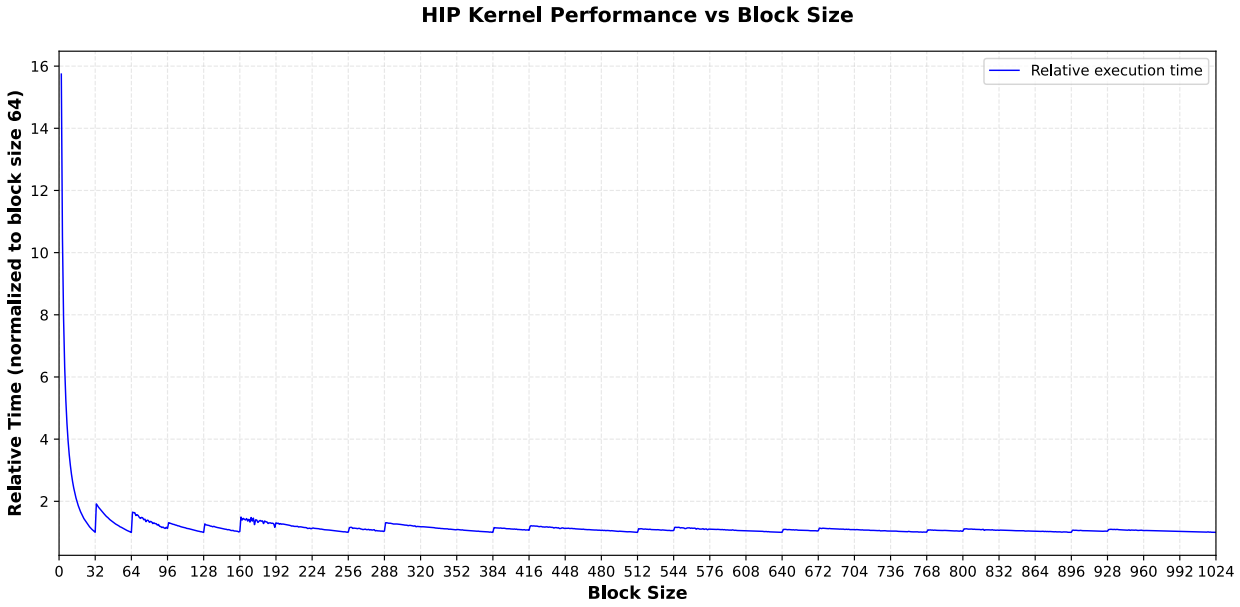


Fig. 3: Impact of block size on program performance. The image gamma correction execution time reflects a 16,384 × 16,384 RGB image on a Radeon PRO W7800 GPU.

- Regular memory access patterns
- Sufficient computational work per thread to offset kernel launch overhead

By understanding these patterns and optimization techniques, you can effectively accelerate a wide range of image processing and data-parallel workloads on AMD GPUs using HIP.

15.5 Fixed-size kernels: image gamma correction

In most of the examples presented thus far, each thread calculates a single output element. This approach, while straightforward, introduces performance bottlenecks that can be addressed through more efficient kernel design patterns.

15.5.1 Addressing kernel dispatch overhead

Upon closer observation of one-thread-per-element implementations, you can see that the overhead is high because for each output element, a thread must be dispatched to the compute unit to calculate the global ID and check the boundary conditions. This overhead becomes particularly significant in memory-bound kernels where the actual computation per element is minimal compared to the thread management costs.

To reduce this overhead, assign more tasks to each thread to amortize the cost of launching threads on the GPU. Rather than changing the grid size to grow with the input and output image, you can fix the number of workgroups in the kernel and change the amount of work per thread according to the image size. This approach, known as a *grid-stride loop* pattern, provides several advantages:

- Reduced thread dispatch overhead through work amortization
- Consistent kernel launch configuration across different problem sizes
- Better resource utilization on the GPU
- Improved scalability for varying input dimensions

15.5.2 Grid-stride loop implementation

This code sample reimplements the *image gamma correction example* using a fixed-sized kernel with the grid-stride loop pattern.

Listing 2: GPU implementation of the image gamma correction using fixed-sized kernel patterns. The kernel is launched with a predefined size, and each thread is responsible for multiple pixels in the image.

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

#include <cstdint>
#include <cstdlib>

__global__ void image_gamma(std::uint8_t* d_image, float gamma, int num_values)
{
    int global_size = blockDim.x * gridDim.x;
    int idx = threadIdx.x + blockIdx.x * blockDim.x;

    for (; idx < num_values; idx += global_size)
    {
        float value = d_image[idx] / 255.f;
        value = powf(value, gamma);
        d_image[idx] = (std::uint8_t)(value * 255.f);
    }
}

int main(int argc, char* argv[])
{
    int width, height, channels;
    std::uint8_t* data;

    // Load an image from file.

    int blockSize = 256;
    int gridSize = [GridSize];

    float gamma = 4.f;
    image_gamma<<<gridSize, blockSize>>>(d_image, gamma, num_values);

    hipMemcpy(
        data, d_image, num_values * sizeof(std::uint8_t), hipMemcpyDeviceToHost
    );

    // Save the image to disk.

    return EXIT_SUCCESS;
}
```

This new implementation approximates the previous one but with two minor modifications. First, it moves the boundary checking `if` statements in the kernel into a `for` loop. This allows each thread to process multiple values and increment the index by the number of threads in each iteration. For example, if you have 640 threads, Thread 0 will process values with indices 0, 640, 1,280, and so on, and Thread 1 will use indices 1, 641, 1,281, and so on. From this change, you can set the number of threads, regardless of the problem size.

15.5.3 Performance characteristics and best practices

Next, identify the best grid size to use. Examine the following figure:

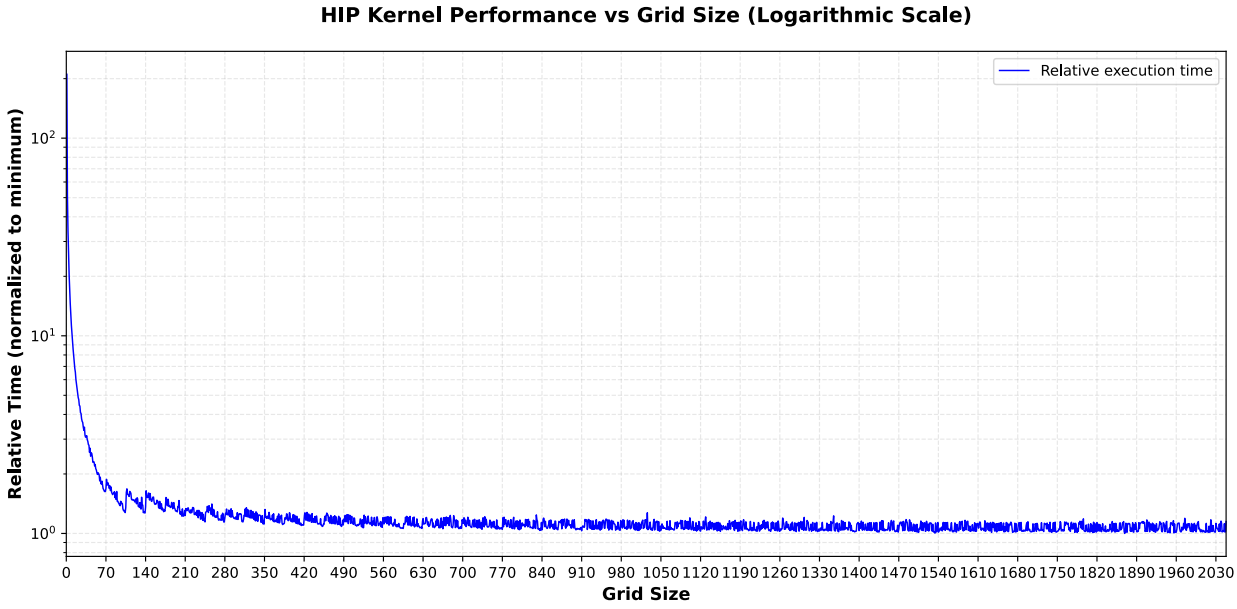


Fig. 4: Impact of grid size on image gamma program performance. The execution time is measured based on a 16,384 × 16,384 RGB image on a Radeon PRO W7800 GPU.

These measurements were taken on a Radeon™ PRO W7800 GPU using a 16,384 × 16,384 RGB image. The GPU has 35 work-group processors (WGPs), and each WGP can execute up to 32 wavefronts concurrently.

At a high level, you observe three major trends:

- The performance improves as the number of workgroups increases because when there is a small number of workgroups, only a fraction of the compute units are used.
- The performance saturates around 1,400 workgroups, which provides approximately 10× oversubscription of the Radeon PRO W7800 GPU's capacity of 1,120 maximum concurrent wavefronts (35 WGPs × 32 wavefronts). This oversubscription is necessary for effective latency hiding in this memory-bound kernel.
- A zig-zag pattern of execution times appears in the figure above. Sudden increases occur after multiples of 35. Given 35 workgroups, each WGP executes one workgroup in parallel; hence, they will likely finish at the same time. A few extra workgroups would not fully overlap with the first 35, but they would significantly increase the kernel execution time. This happens because although the last set of remaining workgroups might not be able to entirely utilize the GPU, they still have to do the same amount of work as the first set of workgroups. This effect, which is known as the *tail-effect*, is more pronounced when you have small grid sizes and should be avoided, if possible.

15.5.3.1 Key takeaways for optimal grid sizing

The experiment demonstrates that a fixed-sized kernel can effectively optimize embarrassingly parallel implementations. When selecting grid sizes for your kernels, consider the following best practices:

- Choose the smallest grid size that can fill all WGPs (RDNA GPUs in WGP mode) or compute units (RDNA GPUs in CU mode and CDNA GPUs) of the GPU.
- Always use a grid size that is a multiple of the WGP or CU count to avoid the tail-effect.

- Aim for sufficient oversubscription (approximately 10× in this example) to enable effective latency hiding in memory-bound kernels.
- Test different grid sizes empirically, as optimal values may vary based on the specific GPU architecture and workload characteristics.

15.6 Reduction

Reduction is a common algorithmic operation used in parallel programming to reduce an array of elements into a shorter array of elements or a single value. This document exploits reduction to introduce some key considerations while designing and optimizing GPU algorithms.

This document is a rejuvenation and extension of the invaluable work of Mark Harris. While the author approaches the topic with a less naive approach, reviewing some original material is valuable to see how much the underlying hardware has changed. This document provides a greater insight to demonstrate progress.

15.6.1 The algorithm

Reduction has many names depending on the domain; in functional programming it's referred to as fold, in C++, it's called `std::accumulate` and in C++17, as `std::reduce`. A reduction takes a range of inputs and "reduces" the given range with a binary operation to a singular or scalar output. Canonically, a reduction requires a "zero" element that bootstraps the algorithm and serves as one of the initial operands to the binary operation. The "zero" element is generally called identity or neutral element in the group theory, which implies that it is an operand that doesn't change the result. Some typical use cases are: calculating a sum or normalizing a dataset and finding the maximum value in the dataset. The latter use case is discussed further in this tutorial.

There are multiple variations of reduction that allow parallel processing. The approach taken by `std::reduce` requires the user-provided binary operator to operate on any combination of identity and input range elements, or even exclusively on any of them. This allows you to insert any number of identities to facilitate parallel processing and then combine the partial results of parallel execution.

15.6.2 Reduction on GPUs

Implementing reductions on GPUs requires a basic understanding of the *Introduction to the HIP programming model*. The document explores aspects of low-level optimization best discussed through the *Hierarchical thread model*, and refrains from using cooperative groups.

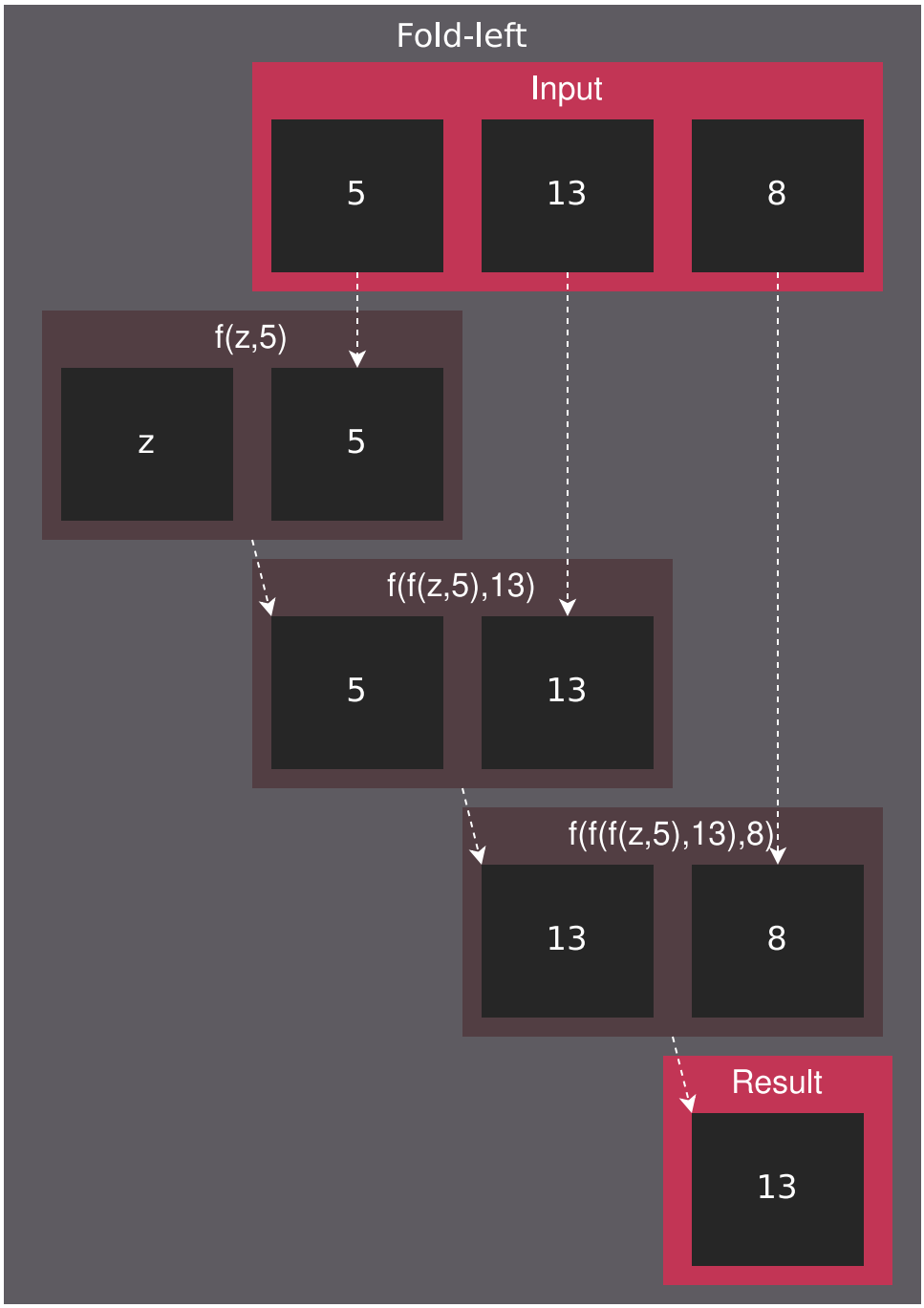
Synchronizing parallel threads of execution across a GPU is crucial for correctness as the partial results can't be synchronized before they manifest. Synchronizing all the threads running on a GPU at any given time is possible, however, it is a costly and intricate operation. If synchronization is not absolutely necessary, map the parallel algorithm so that multiprocessors and blocks can make independent progress and need not sync frequently.

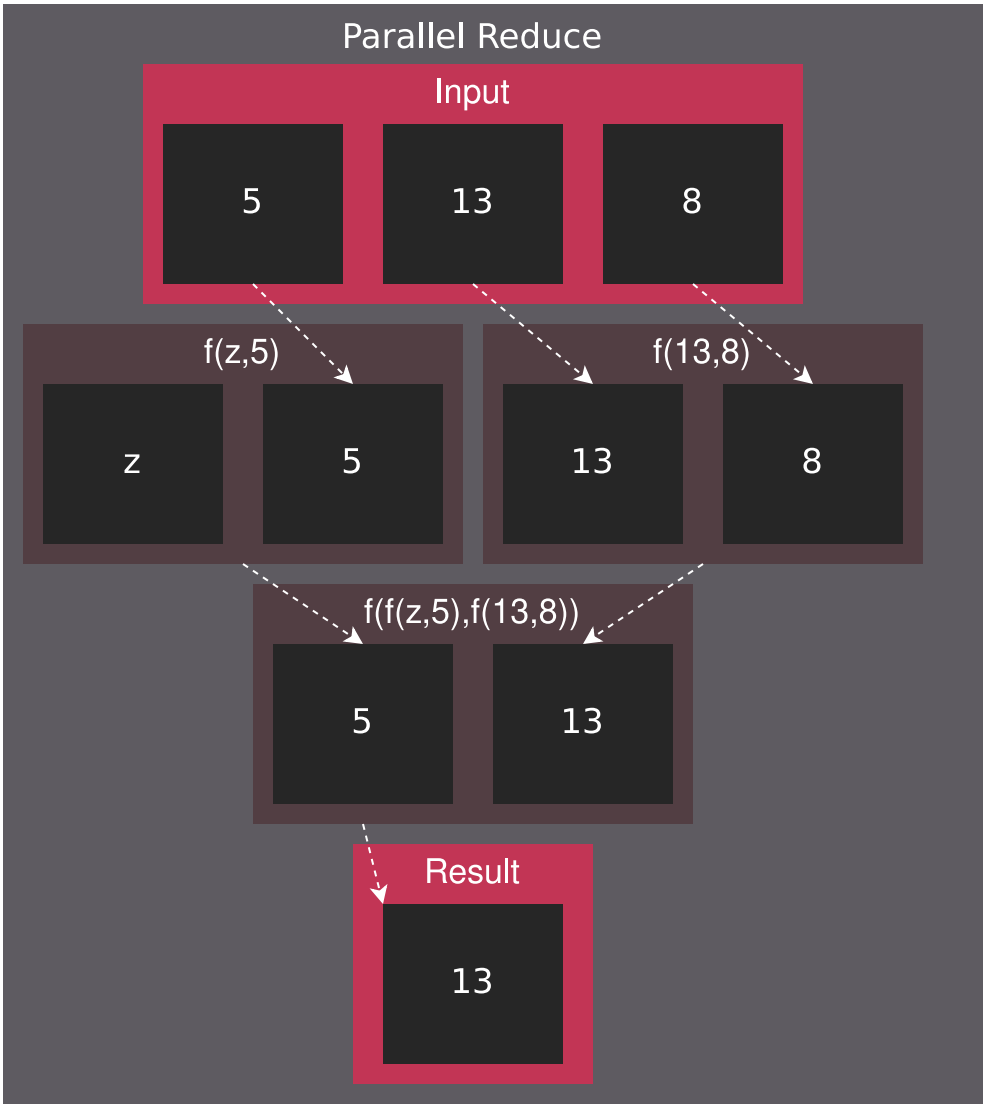
There are ten reduction implementations in the rocm-examples, which are described in the following sections.

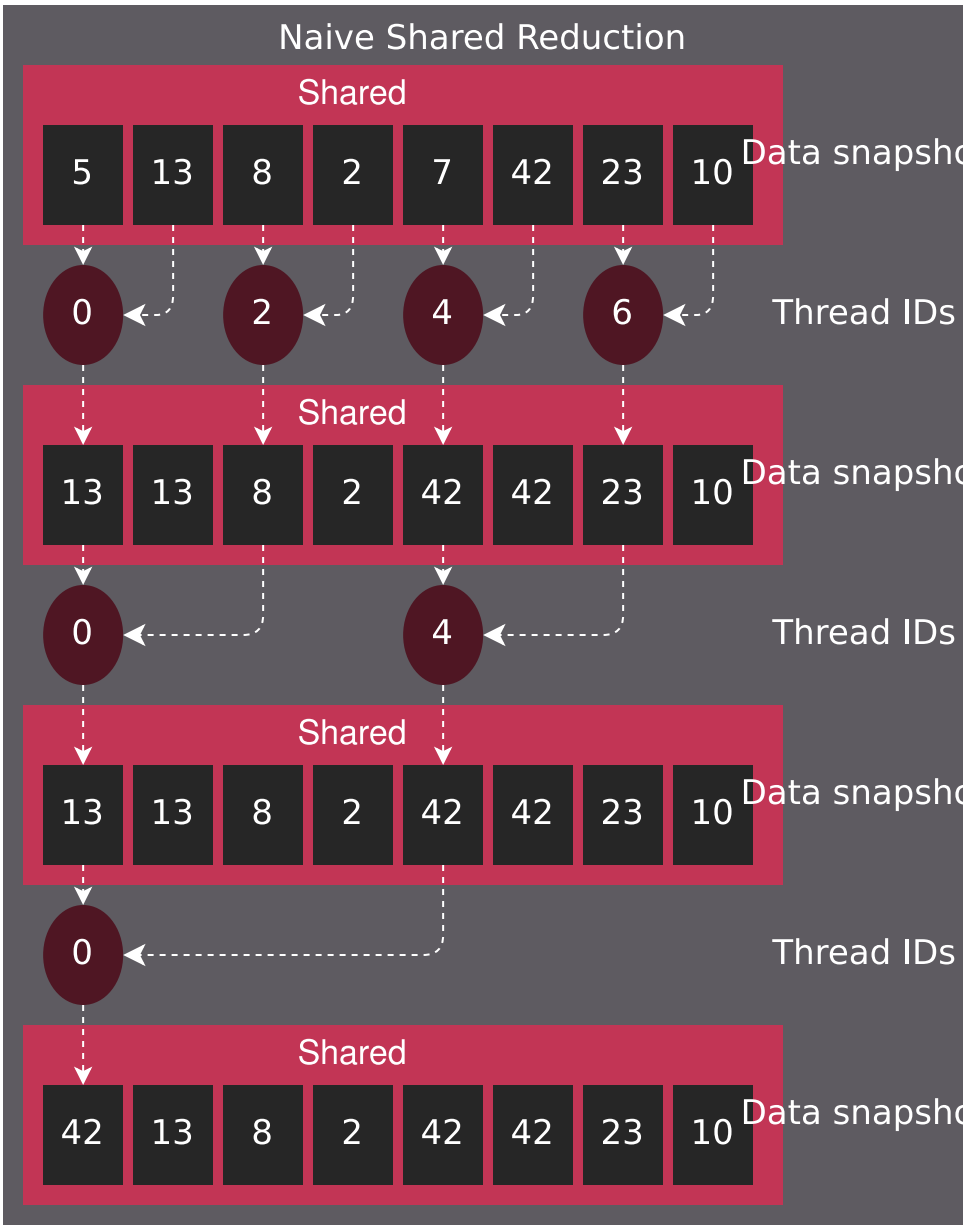
15.6.2.1 Naive shared reduction

The naive algorithm takes a tree-like shape, where the computational domain is purposefully distributed among blocks. In all blocks, all threads participate in loading data from persistent (from the kernel's perspective) global memory into the shared memory. This helps to perform tree-like reduction for a single thread by writing the partial result to global, in a location unique to the block, which allows the block to make independent progress. The partial results are combined in subsequent launches of the same kernel until a scalar result is reached.

This approach requires temporary storage based on the number of blocks launched, as each block outputs a scalar partial result. Depending on the need to store or destroy the input, a second temporary storage might be needed, which could be large enough to store the results of the second kernel launch. Alternatively, you can reuse the storage of the larger than necessary original input. These implementations differ so slightly that the document only considers the use case where the input could be destroyed.


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


Naive Shared Reduction
Shared
5 13 8 2 7 42 23 10
Data snapsho
0 2 4 6
Thread IDs
Shared
13 13 8 2 42 42 23 10
Data snapsho
0 4
Thread IDs
Shared
13 13 8 2 42 42 23 10
Data snapsho
0
Thread IDs
Shared
42 13 8 2 42 42 23 10
Data snapsho

```
std::size_t factor = block_size; // block_size from hipGetDeviceProperties()
auto new_size = [factor](const std::size_t actual)
{
    // Every pass reduces input length by 'factor'. If actual size is not divisible␣
↪by factor,
    // an extra output element is produced using some number of zero_elem inputs.
    return actual / factor + (actual % factor == 0 ? 0 : 1);
};
```

For threads that don't have unique inputs, feed `zero_elem` instances to threads. The backing of double-buffering is allocated as such:

```
// Initialize host-side storage
std::vector<unsigned> input(input_count);
std::iota(input.begin(), input.end(), 0);

// Initialize device-side storage
unsigned *front,
         *back;
hipMalloc((void**)&front, sizeof(unsigned) * input_count);
hipMalloc((void**)&back,  sizeof(unsigned) * new_size(input_count));

hipMemcpy(front, input.data(), input.size() * sizeof(unsigned),␣
↪hipMemcpyHostToDevice);
```

Data is initialized on the host and dispatched to the device followed by the commencement of device-side reduction. The swapping of the double-buffer on the last iteration is omitted, therefore the result is in the back-buffer irrespective of the input size.

```
for (uint32_t curr = input_count; curr > 1;)
{
    hipLaunchKernelGGL(
        kernel,
        dim3(new_size(curr)),
        dim3(block_size),
        factor * sizeof(unsigned),
        hipStreamDefault,
        front,
        back,
        kernel_op,
        zero_elem,
        curr);

    curr = new_size(curr);
    if (curr > 1)
        std::swap(front, back);
}
```

This structure persists in the kernel throughout all the variations of reduction with slight modifications to `factor` and shared memory allocation:

```
template<typename T, typename F>
__global__ void kernel(
```

(continues on next page)

(continued from previous page)

```
    T* front,
    T* back,
    F op,
    T zero_elem,
    uint32_t front_size)
{
    extern __shared__ T shared[];

    // Overindex-safe read of input
    auto read_global_safe = [&](const uint32_t i)
    {
        return i < front_size ? front[i] : zero_elem;
    };

    const uint32_t tid = threadIdx.x,
                   bid = blockIdx.x,
                   gid = bid * blockDim.x + tid;

    // Read input from front buffer to shared
    shared[tid] = read_global_safe(gid);
    __syncthreads();

    // Shared reduction
    for (uint32_t i = 1; i < blockDim.x; i *= 2)
    {
        if (tid % (2 * i) == 0)
            shared[tid] = op(shared[tid], shared[tid + i]);
        __syncthreads();
    }

    // Write result from shared to back buffer
    if (tid == 0)
        back[bid] = shared[0];
}
```

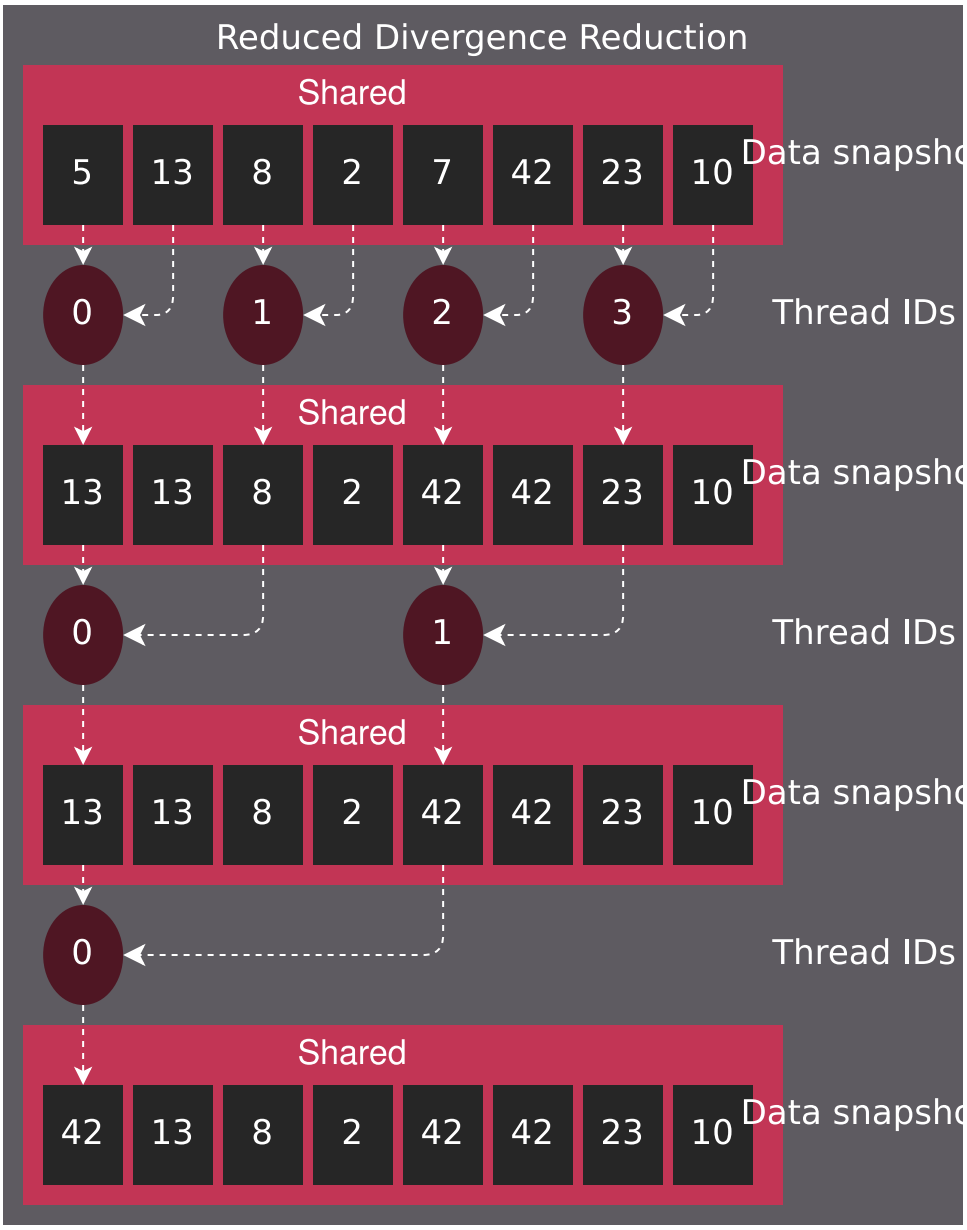
While the `tid % (2 * i) == 0` indexing scheme yields correct results, it also leads to high thread divergence. Thread divergence indicates the event when the threads in a warp diverge, which implies that the threads have to execute different instructions in a given clock cycle. This is easily manifested using `if-else` statements as shown here, but can also be manifested as `for` loop dependent on thread ID lengths. Even though the number of active threads participating in the reduction reduces, warps remain active longer than necessary, as at least one lane in a warp hits the `if` statement.

15.6.2.2 Reducing thread divergence

You can reduce divergence by keeping dataflow between memory addresses identical but reassigning the thread ids.

```
// Shared reduction
for (uint32_t i = 1; i < blockDim.x; i *= 2)
{
-    if (tid % (2 * i) == 0)
-        shared[tid] = op(shared[tid], shared[tid + i]);
+    if (uint32_t j = 2 * i * tid; j < blockDim.x)
+        shared[j] = op(shared[j], shared[j + i]);
```

(continues on next page)


Reduced Divergence Reduction
Shared
5 13 8 2 7 42 23 10
Data snapsho
0 1 2 3
Thread IDs
Shared
13 13 8 2 42 42 23 10
Data snapsho
0 1
Thread IDs
Shared
13 13 8 2 42 42 23 10
Data snapsho
0
Thread IDs
Shared
42 13 8 2 42 42 23 10
Data snapsho

(continued from previous page)

```
    __syncthreads();
}
```

This way inactive threads start accumulating uniformly towards the higher thread ID index range and might uniformly skip to `__syncthreads()`. However, this introduces a bank conflicts issue.

15.6.2.3 Resolving bank conflicts

Both AMD and NVIDIA implement shared memory in the hardware by organizing storage into banks of various sizes. This hardware element is known as Local Data Share (LDS) on AMD hardware. On NVIDIA hardware, it's implemented using the same silicon as the L1 data cache. You can think of shared memory as a striped 2-dimensional range of memory. Shared memory bank's count, width, and depth depend on the architecture. A bank conflict occurs when different threads in a warp access the same bank during the same operation. In this case, the hardware prevents the attempted concurrent accesses to the same bank by converting them into serial accesses.

- "AMD Instinct MI200" Instruction Set Architecture, Chapter 11.1
- "RDNA 2" Instruction Set Architecture, Chapter 10.1

A notable exception is when the shared read uniformly broadcasts to the same address across the entire warp. A better implementation of the naive algorithm is to form continuous ranges of the threads activities and their memory accesses.

```
// Shared reduction
-for (uint32_t i = 1; i < blockDim.x; i *= 2)
-{
-    if (tid % (2 * i) == 0)
+for (uint32_t i = blockDim.x / 2; i != 0; i /= 2)
+{
+    if (tid < i)
        shared[tid] = op(shared[tid], shared[tid + i]);
    __syncthreads();
}
```

> **Note**
>
> To avoid bank conflicts, read shared memory in a coalesced manner, which implies that reads/writes of each lane in a warp evaluate to consecutive locations. Analyzing the read/write patterns could help you to understand the cause of bank conflicts. For more details, check CDNA3 ISA or RDNA3 ISA data share operations chapter.

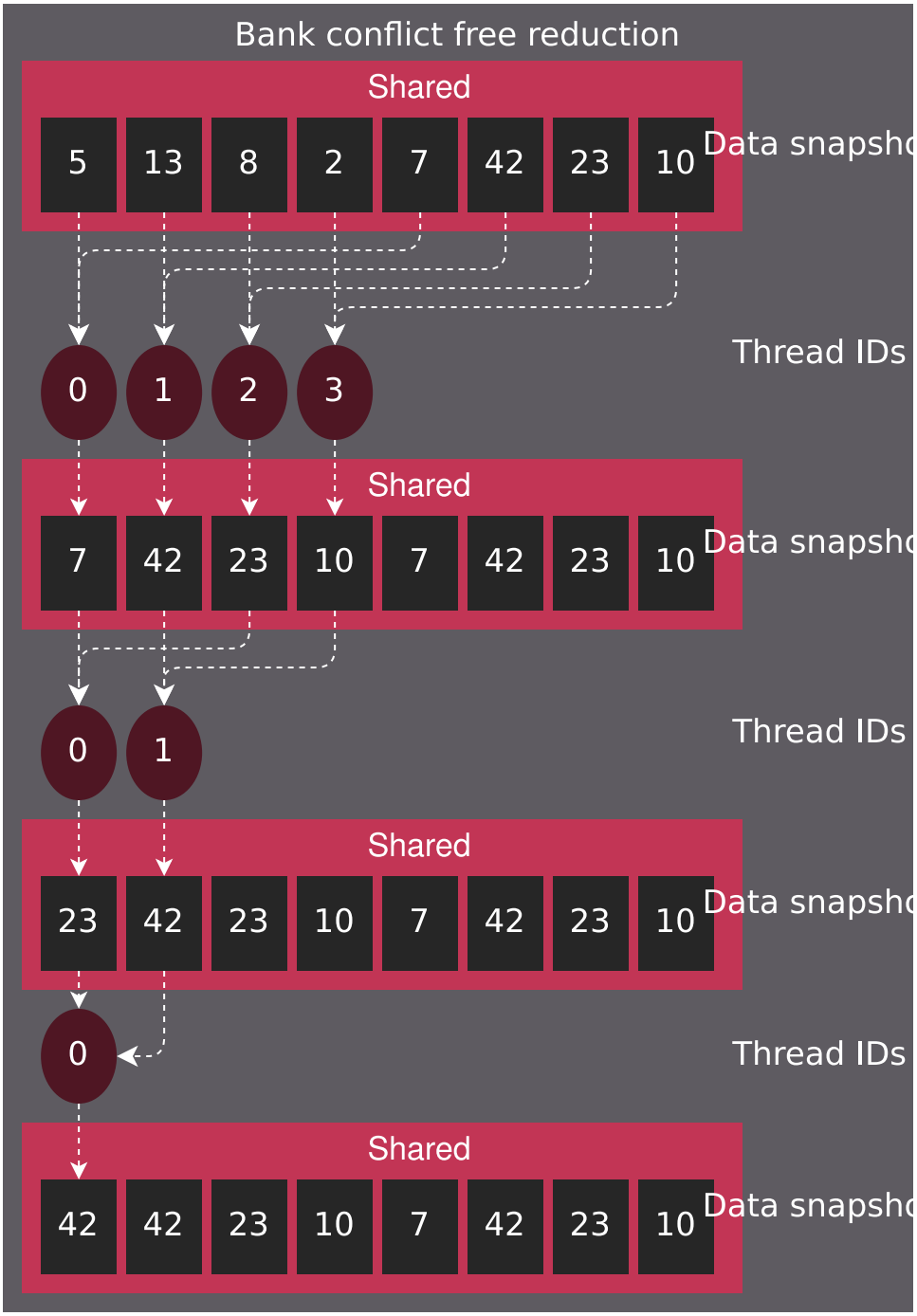
15.6.2.4 Utilize upper half of the block

The preceding implementation is free of low-level GPU-specific anti-patterns. However, it still exhibits some common shortcomings. The loop performing the reduction in the shared memory starts from `i = blockDim.x / 2` and the first predicate `if (tid < i)` immediately disables half of the block, which only helps load the data into the shared memory. You can change the kernel along with the calculation of `factor` on the host, as shown here:

```
const uint32_t tid = threadIdx.x,
               bid = blockIdx.x,
-              gid = bid * blockDim.x + tid;
+              gid = bid * (blockDim.x * 2) + tid;

// Read input from front buffer to shared
-shared[tid] = read_global_safe(gid);
```

(continues on next page)


Bank conflict free reduction
Shared
5 13 8 2 7 42 23 10
Data snapsho
Thread IDs
0 1 2 3
Shared
7 42 23 10 7 42 23 10
Data snapsho
Thread IDs
0 1
Shared
23 42 23 10 7 42 23 10
Data snapsho
Thread IDs
0
Shared
42 42 23 10 7 42 23 10
Data snapsho

(continued from previous page)

```
+shared[tid] = op(read_global_safe(gid), read_global_safe(gid + blockDim.x));
__syncthreads();
```

By eliminating half of the threads and giving meaningful work to all the threads by unconditionally performing a binary `op`, you can prevent the wastage of half of the threads.

Even though global memory is read in a coalesced fashion, as preferred by the memory controller, optimal performance is still limited by the instruction throughput. Omit superfluous synchronization ——————————————

Warps are known to execute in a strict lockstep fashion. Therefore, once shared reduction reaches a point where only a single warp participates meaningfully, you can cut short the loop and let the rest of the warps terminate. Moreover, you can also unroll the loop without syncing the entire block.

The `tmp` namespace used beyond this point in this document holds a handful of template meta-programmed utilities to facilitate writing flexible and optimal code.

`tmp::static_for` is not just a constant folding within the optimizer but a variation of the language `for` loop, where the running index is a compile-time constant and is eligible for use in compile-time evaluated contexts.

Consider the following code:

```
constexpr int size = 4;
for (int i = 0 ; i < size ; ++i)
{
    printf("%d", i);
}
```

This compiles to the following binaries:

LLVM Block

```
main:
    push    rbx
    lea     rbx, [rip + .L.str]
    mov     rdi, rbx
    xor     esi, esi
    xor     eax, eax
    call    printf@PLT
    mov     rdi, rbx
    mov     esi, 1
    xor     eax, eax
    call    printf@PLT
    mov     rdi, rbx
    mov     esi, 2
    xor     eax, eax
    call    printf@PLT
    mov     rdi, rbx
    mov     esi, 3
    xor     eax, eax
    call    printf@PLT
    xor     eax, eax
    pop     rbx
    ret
.L.str:
    .asciz  "%d"
```

GCC

```
.LC0:
    .string "%d"
main:
    push    rbx
    xor     ebx, ebx
.L2:
    mov     esi, ebx
    mov     edi, OFFSET FLAT:.LC0
    xor     eax, eax
    add     ebx, 1
    call    printf
    cmp     ebx, 4
    jne     .L2
    xor     eax, eax
    pop     rbx
    ret
```

MSVC

```
main    PROC
    $LN12:
    push    rbx
    sub     rsp, 32
    xor     ebx, ebx
    npad    8
$LL4@main:
    mov     edx, ebx
    lea     rcx, OFFSET FLAT:'string'
    call    printf
    inc     ebx
    cmp     ebx, 4
    jl      SHORT $LL4@main
    xor     eax, eax
    add     rsp, 32
    pop     rbx
    ret     0
main    ENDP
```

LLVM unrolls the loop and compiles to a flat series of `printf` invocations, while both GCC and MSVC keep the loop intact, as visible from the compare (`cmp`) and the jump (`jne`, `jl`) instructions. LLVM code generation is identical to manually writing the unrolled loop:

```
printf("%d", 0);
printf("%d", 1);
printf("%d", 2);
printf("%d", 3);
```

While various non-standard pragmas are available to hint or force the compiler to unroll the loop, we instead use template meta-programming to force feed the compiler the unrolled loop.

```
constexpr int size = 4;
```

(continues on next page)

(continued from previous page)

```
// Maybe unrolled loop
for (int i = 0 ; i < size ; ++i)
{
    printf("%d", i);
}

// Force unrolled loop
using namespace tmp;
static_for<0, less_than<size>, increment<1>>([]<int i>()
{
    printf("%d", i);
});
```

The most notable structural difference is that in the language `for` loop, the loop variable is given a name in the beginning, while in the `static_for` utility, the loop variable is given a name in the end. An important bonus is that in the loop's body, you can use the running index `i` in contexts requiring constant expressions such as template arguments or inside `if constexpr`.

`tmp::static_switch` takes runtime value and runtime dispatches to a range of set of tabulated functions, where said value is a compile-time constant and is eligible for use in compile-time evaluated contexts.

Consider the following code:

```
int warp_size = device_props.warpSize;
switch (warp_size)
{
case 32:
    hipLaunchKernelGGL(kernel<32>, ...);
    break;
case 64:
    hipLaunchKernelGGL(kernel<64>, ...);
    break;
}
```

In the preceding code, note the code repetition for all possible values of `warp_size`, the code is prepared to handle. To avoid this, use `tmp::static_switch`, as shown:

```
tmp::static_switch<std::array{32, 64}>(warp_size, [&]<int WarpSize>()
{
    hipLaunchKernelGGL(kernel<WarpSize>, ...);
});
```

```
-template<typename T, typename F>
+template<uint32_t WarpSize, typename T, typename F>
__global__ void kernel(
    ...
)
{
    ...
// Shared reduction
-for (uint32_t i = blockDim.x / 2; i != 0; i /= 2)
+for (uint32_t i = blockDim.x / 2; i > WarpSize; i /= 2)
{
```

(continues on next page)

(continued from previous page)

```
    if (tid < i)
        shared[tid] = op(shared[tid], shared[tid + i]);
    __syncthreads();
}
+// Warp reduction
+tmp::static_for<WarpSize, tmp::not_equal<0>, tmp::divide<2>>([&]<int I>()
+{
+    if (tid < I)
+        shared[tid] = op(shared[tid], shared[tid + I]);
+#ifdef __HIP_PLATFORM_NVIDIA__
+    __syncwarp(0xffffffff >> (WarpSize - I));
+#endif
+});
```

Because HIP typically targets hardware with warp sizes of 32 (RDNA AMD GPUs) and 64 (CDNA AMD GPUs), portable HIP code must handle both. That is why instead of assuming a warp size of 32, make the warp size a template argument of the kernel. This allows you to unroll the final loop using `tmp::static_for` in a parametric way but still having the code read much like an ordinary loop.

Promoting the warp size to being a compile-time constant also requires you to handle it similarly on the host-side. You can sandwich the kernel launch with `tmp::static_switch`, promoting the snake-case run-time `warp_size` variable to a camel-case compile-time constant `WarpSize`.

```
// Device-side reduction
for (uint32_t curr = input_count; curr > 1;)
{
+    tmp::static_range_switch<std::array{32, 64}>(warp_size, [&]<int WarpSize>()␣
↪noexcept
+    {
        hipLaunchKernelGGL(
-            kernel,
+            kernel<WarpSize>,
            dim3(new_size(curr)),
            dim3(block_size),
            factor * sizeof(unsigned),
            hipStreamDefault,
            front,
            back,
            kernel_op,
            zero_elem,
            curr);
+    });
    ...
}
```

Note

Neither RDNA- nor CDNA-based AMD hardware provides guaranteed independent progress to lanes of the same warp. When targeting NVIDIA hardware, lanes of a warp might execute somewhat independently as long as the programmer assists the compiler using dedicated built-in functions. This feature is called Independent Thread Scheduling. The HIP headers don't expose the necessary warp primitives and their overloads.

Portable applications can still tap into this feature with carefully `#ifdef` -ed code, but at this particular optimization

level, it's a requirement. The code implicitly relies on the lockstep behavior of an ROCm wavefront, but CUDA warps don't share this property. You must synchronize all the active lanes of a warp to avoid a data race with some lanes progressing faster than others in the same warp.

15.6.2.5 Unroll all loops

While the previous step primarily aims to remove unnecessary syncing, it also unrolls the end of the loop. However, you could also force unrolling the first part of the loop. This saves a few scalar registers (values the compiler can prove to be uniform across warps).

```
-template<uint32_t WarpSize, typename T, typename F>
-__global__ void kernel(
+template<uint32_t BlockSize, uint32_t WarpSize, typename T, typename F>
+__global__ __launch_bounds__(BlockSize) void kernel(
    T* front,
    T* back,
    F op,
    T zero_elem,
    uint32_t front_size)
{
-    extern __shared__ T shared[];
+    __shared__ T shared[BlockSize];

    ...

    // Shared reduction
-    for (uint32_t i = blockDim.x / 2; i > WarpSize; i /= 2)
+    tmp::static_for<BlockSize / 2, tmp::greater_than<WarpSize>, tmp::divide<2>>([&]
↪<int I>()
    {
-        if (tid < i)
-            shared[tid] = op(shared[tid], shared[tid + i]);
+        if (tid < I)
+            shared[tid] = op(shared[tid], shared[tid + I]);
        __syncthreads();
    }
+    );
```

Introducing yet another template argument for the kernel and moving from `for` to `tmp::static_for` leads to the following two notable improvements:

- Introducing new attribute `__launch_bounds__(BlockSize)` to the kernel instructs the compiler that the kernel will only be launched using the designated block size. This implies that the launches of differing block sizes will fail. This allows the optimizer to enroll the `blockDim.x` variable in constant folding as well as get information about register usage.
- Turning the block size into a compile-time constant allows you to statically allocate the shared memory.

15.6.2.6 Communicate using warp-collective functions

Shared memory provides a fast communication path within a block, however when performing reduction within the last warp, you can use faster means of communication, which is warp-collective or cross-lane functions. Instead of using the hardware-backed shared memory, you can directly copy between the local memory (registers) of each lane in a warp. This can be achieve using the shuffle functions.

See how to use `__shfl_down()`, which is one of the most restrictive but also the most structured communication schemes.

```
// Warp reduction
if (tid < WarpSize)
{
    T res = op(shared[tid], shared[tid + WarpSize]);
    tmp::static_for<WarpSize / 2, tmp::not_equal<0>, tmp::divide<2>>([&]<int Delta>()
    {
        res = op(res, __shfl_down(res, Delta));
    });

    // Write result from shared to back buffer
    if (tid == 0)
        back[bid] = res;
}
```

Using warp-collective functions for communication requires the control flow to be uniform across warps, as the name warp-collective implies. Therefore, you can see that the thread ID is being checked outside the loop, but the result is written inside due to variable scoping.

15.6.2.7 Prefer warp communication over shared

As mentioned in the previous step, communication between local memory is faster than shared memory. Instead of relying on the local memory only at the end of the tree-like reduction, a better approach is to turn the tree reduction inside out and perform multiple warp reductions in parallel on all active threads, thus communicating only their partial results through the shared memory.

The kernel versions differ significantly enough to be described using a diff; use afresh instead.

```
template<uint32_t BlockSize, uint32_t WarpSize, typename T, typename F>
__global__ __launch_bounds__(BlockSize) void kernel(
    T* front,
    T* back,
    F op,
    T zero_elem,
    uint32_t front_size)
{
    // ...
}
```

The kernel signature and the reduction factor are the same as in previous cases; only the implementation differs.

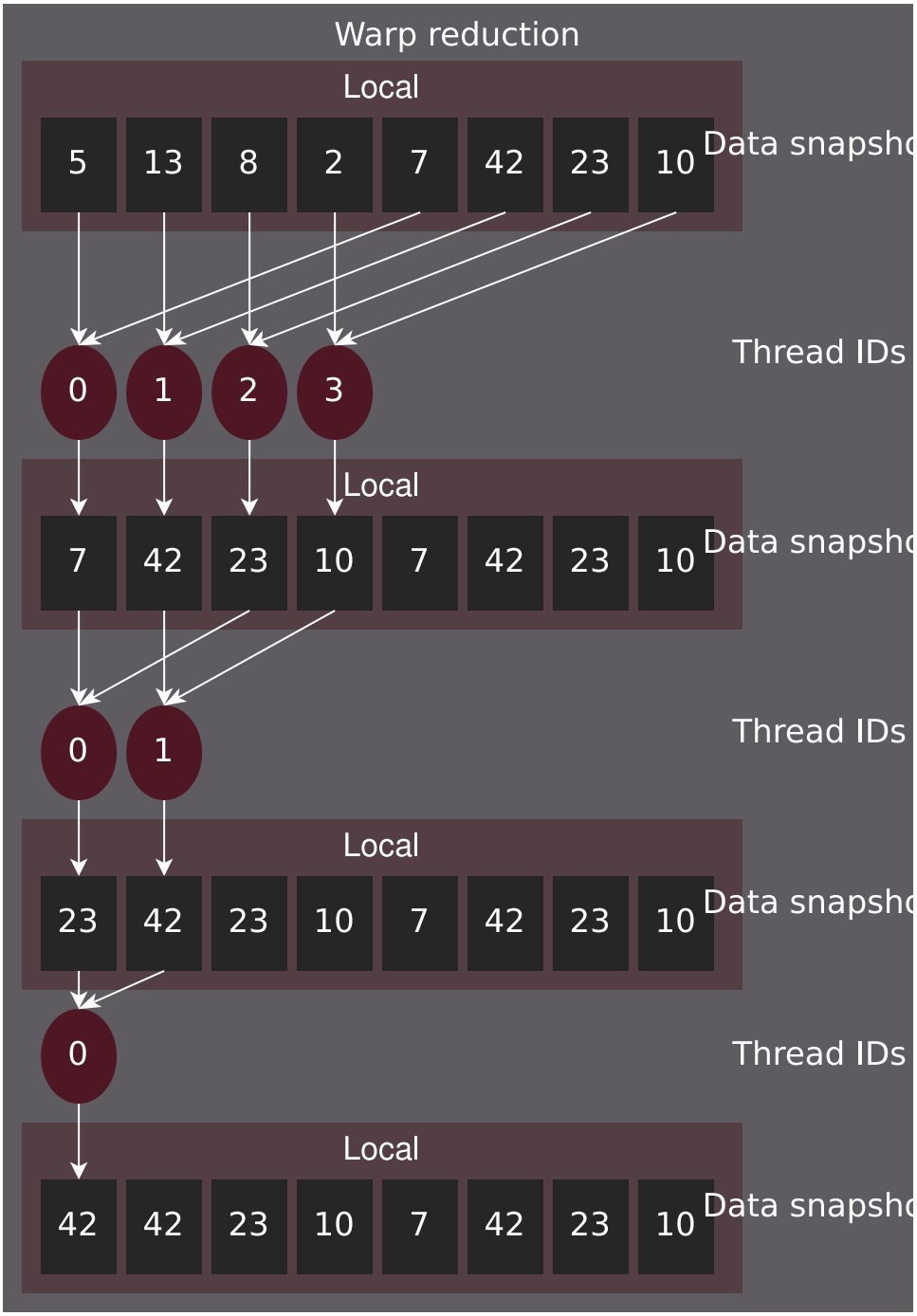
```
static constexpr uint32_t WarpCount = BlockSize / WarpSize;

__shared__ T shared[WarpCount];

auto read_global_safe =
    [&](const uint32_t i) { return i < front_size ? front[i] : zero_elem; };
auto read_shared_safe =
    [&](const uint32_t i) { return i < WarpCount ? shared[i] : zero_elem; };

const uint32_t tid = threadIdx.x,
               bid = blockIdx.x,
               gid = bid * (blockDim.x * 2) + tid,
```

(continues on next page)


Warp reduction
Local
5 13 8 2 7 42 23 10
Data snapsho
Thread IDs
0 1 2 3
Local
7 42 23 10 7 42 23 10
Data snapsho
Thread IDs
0 1
Local
23 42 23 10 7 42 23 10
Data snapsho
Thread IDs
0
Local
42 42 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
7 42 23 10 7 42 23 10
Local
23 42 23 10 7 42 23 10
Local
42 42 23 10 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
42 50
Local
42 50
Local
92 50

(continued from previous page)

```
            wid = tid / WarpSize,
            lid = tid % WarpSize;

// Read input from front buffer to local
T res = op(read_global_safe(gid), read_global_safe(gid + blockDim.x));
```

As we communicate the results of warps through shared memory, the same number of elements are required in the shared memory as warps within the block. Similar to how you can only launch kernels at block granularity, you can only warp reduce with `WarpSize` granularity due to the collective nature of the cross-lane builtins. To address this, you can use `read_shared_safe` to pad overindexing by reading `zero_elem`. Reading from global remains unaffected.

```
// Perform warp reductions and communicate results via shared
// for (uint32_t ActiveWarps = WarpCount;
//      ActiveWarps != 0;
//      ActiveWarps = ActiveWarps != 1 ?
//          divide_ceil(ActiveWarps, WarpSize) :
//          ActiveWarps = 0)
tmp::static_for<
    WarpCount,
    tmp::not_equal<0>,
    tmp::select<
        tmp::not_equal<1>,
        tmp::divide_ceil<WarpSize>,
        tmp::constant<0>>>([&]<uint32_t ActiveWarps>()
{
    if(wid < ActiveWarps)
    {
        // Warp reduction
        tmp::static_for<WarpSize / 2, tmp::not_equal<0>, tmp::divide<2>>([&]<int␣
↪Delta>()
        {
            res = op(res, __shfl_down(res, Delta));
        });

        // Write warp result from local to shared
        if(lid == 0)
            shared[wid] = res;
    }
    __syncthreads();

    // Read warp result from shared to local
    res = read_shared_safe(tid);
});

// Write result from local to back buffer
if(tid == 0)
    back[bid] = res;
```

`ActiveWarps` iterates from `WarpCount` until it reaches `0`. Every iteration of `ActiveWarps` reduces the `WarpSize`. In cases where the partial result count isn't a divisor of `ActiveWarps` and you need to launch an extra warp, use `tmp::divide_ceil`, which always rounds to positive infinity. The tertiary `tmp::select` is required because such division never reaches `0`, so you must terminate the loop after the last warp concludes.

In each iteration, if the warp is active, which means it has at least a single valid input, it carries out a pass of warp reduction and writes output based on warp ID. Reading is carried out based on thread ID. Global output continues to be based on block ID.

15.6.2.8 Amortize bookkeeping variable overhead

The previous sections explained how to reduce register usage to improve occupancy. This allows more blocks to execute in parallel on all multiprocessors, leading to more global store/load latency to be hidden. Reducing the number of kernels in flight while still carrying out the same workload reduces the wastage of registers while loading and maintaining bookkeeping variables such as kernel indices.

An example of this optimization is performing one binary `op` while loading input from global. Even though the operation is said to be carried out "in flight", the two values are loaded into local memory (registers) before `op` is called.

A more general form of this optimization is wrapping most kernel logic in loops that carry out the workload of multiple kernel instances but require storing only a single instance of most of the bookkeeping logic. In code, this multiplicity factor is referred to via the `ItemsPerThread` compile-time constant, which is supplied by a template argument to allow for loop unrolling.

This kernel variant utilizes another generally applicable utility known as `hip::static_array`, which is a more restrictive wrapper over the builtin array than `std::array`, as it allows indexing only compile-time constants using the usual tuple-like `template <size_t I> auto get<I>(...)` interface.

Note

On a GPU, there is no stack, and the local memory is provisioned from the register file. This provisioning takes place statically. To paraphrase, the address range of a thread's local memory is determined at compile-time. When an array is defined and used in the local storage, the compiler can only maintain its storage in the register file as long as all accesses to the array are computable by the compiler at compile-time. It doesn't need to be a compile-time constant as long as the compiler can resolve the addresses of the accesses through constant folding or some other means. If the compiler fails to do so, the array will be backed by global memory, which is indicated by allocating a non-zero number of spill registers observable using static analysis tools. However, this is slower by the magnitude of multiple order. `hip::static_array` via its `hip::get<>` interface ensures that no such spills occur.

```
template<uint32_t BlockSize, uint32_t WarpSize, uint32_t ItemsPerThread>
__global__ static __launch_bounds__(BlockSize) void kernel(...)
```

The kernel now has three compile-time configurable parameters. The only part of the kernel that changes depends on how you load data from global and perform the binary operation on those loaded values. So, the following step to read input from front buffer to global is now split into two steps: *Reading ItemsPerThread* and *Processing ItemsPerThread* .

```
// Read input from front buffer to local
T res = op(read_global_safe(gid), read_global_safe(gid + blockDim.x));
```

15.6.2.8.1 Reading ItemsPerThread

The change to reading happens inside `read_global_safe`:

```
auto read_global_safe = [&](const int32_t i) -> hip::static_array<T, ItemsPerThread>
{
    return [&]<int32_t... I>(std::integer_sequence<int32_t, I...>)
    {
        if(i + ItemsPerThread < front_size)
```

(continues on next page)

(continued from previous page)

```
            return hip::static_array<T, ItemsPerThread>{
                front[i + I]...
            };
        else
            return hip::static_array<T, ItemsPerThread>{
                (i + I < front_size ? front[i + I] : zero_elem)...
            };
    }(std::make_integer_sequence<int32_t, ItemsPerThread>());
};
```

Note that each array element is being loaded consecutively without the flexibility of a configurable `ItemsPerThread` property. This is morally equivalent to:

```
T arr[4] = {
    front[gid + 0],
    front[gid + 1],
    front[gid + 2],
    front[gid + 3]
}
```

This is exactly what's happening in the `front[i + I]...` fold-expression. However, this can only be issued if the entire read operates on real input without padding using `zero_elem`. If some reads over-index the input, the read turns into:

```
T arr[4] = {
    i + 0 < front_size ? front[i + 0] : zero_elem,
    i + 1 < front_size ? front[i + 1] : zero_elem,
    i + 2 < front_size ? front[i + 2] : zero_elem,
    i + 3 < front_size ? front[i + 3] : zero_elem
}
```

This makes it easier for the compiler to recognize vector loads from global. As the performance at large is dominated by how you move the data, it's only natural to utilize dedicated instructions to move more data with less binary. This is evident by the huge performance improvement when loading two values per thread. For more information, see the compiler explorer to learn how loading for AMD (both RDNA and CDNA) compiles to `global_load_dwordx4`, where `x4` denotes the 4-vector variant of the instruction.

Note

Note that `read_global_safe`, which used to take an `uint32_t` as the index type, now takes a signed integer. When indexing an array with unsigned integers, the compiler has to handle integer overflows, as the C/C++ standards defined them. It might happen that some part of the vector load indices overflow, thus resulting in a non-contiguous read. If you change the previously linked code to use an unsigned integer as the thread ID, the compiler won't emit a vector load. Signed integer overflow is an undefined behavior, and hence, unknown to the optimizer. To convey the absence of overflow to the compiler with unsigned indices, add `__builtin_assume(gid + 4 > gid)`, or the more portable `[[assume]](gid + 4 > gid)`, once `amdclang++` supports it.

`read_global_safe` implementation is an Immediately Invoked Lambda Expression (IILE), because `ItemsPerThread` is an integer value, while you need a compile-time `iota`-like sequence of integers as a pack for the fold-expressions to expand on. This can only occur as part of template argument deduction on the IILE.

15.6.2.8.2 Processing `ItemsPerThread`

Once the kernel reads `ItemsPerThread` number of inputs to local, it immediately reduces them to a scalar. There is no reason to propagate the input element multiplicity to the warp reduction phase.

```
T res = [&]()
{
    // Read input from front buffer to local
    hip::static_array<T, ItemsPerThread> arr = read_global_safe(gid);

    // Reduce ItemsPerThread to scalar
    tmp::static_for<1, tmp::less_than<ItemsPerThread>, tmp::increment<1>>([&]<int I>()
    {
        get<0>(arr) = op(get<0>(arr), get<I>(arr));
    });

    return get<0>(arr);
}();
```

15.6.2.9 Two-pass reduction

Alter kernel launch and input fetching such that no more blocks are launched than what a subsequent kernel launch's single block can conveniently reduce, while performing multiple passes of input reading from global and combining their results before engaging in the end game tree-like reduction.

With this method, you can save at least one to two kernel launches for large inputs.

15.6.2.10 Global data share

Warning

This modification can only be executed on AMD hardware.

Perform the first step of the two-pass reduction, but in the end, instead of writing to global and reading it back in a subsequent kernel, write the partial results to the Global Data Share (GDS). This is an `N+1` th shared memory that is accessed by all multiprocessors and is also on-chip memory.

Note

The API doesn't guarantee the order in which blocks are scheduled even though all GPUs schedule them in the same monotonically increasing order of block ids. Relying on this implicitly, the last block of a grid is in the optimal position to observe the side effects of all other blocks (using spinlocks or other methods) without occupying a multiprocessor for longer than necessary.

Without launching a second kernel, you can make the last block collect the results of all other blocks from GDS by implicitly exploiting the scheduling behavior or relying on another AMD-specific feature called Global Wave Sync (GWS) to merge them for a final tree-like reduction.

Note

GDS and GWS are reserved runtime features that the HIP API doesn't cover. Invoking these functionalities requires

inline AMDGCN assembly. Moreover, the fact that the runtime doesn't virtualize the GDS, imposes further restrictions on concurrent scheduling of other kernels.

15.6.3 Conclusion

Optimizing code on GPUs, like on any other architecture, requires careful consideration and balancing of resources and costs of various operations to obtain optimal performance. This document explored optimizing reductions much beyond the territory of diminishing returns. This approach introduced multiple optimization techniques and discussed opportunities.

The document focused on reductions when an entire device participates in it. Still, the choice of optimal compile-time constants or even the algorithm itself might not be optimal when its multiple blocks participate in multiple parallel reductions or when each thread performs its reduction. However, when multiple devices participate in the same reduction, other aspects must be considered.

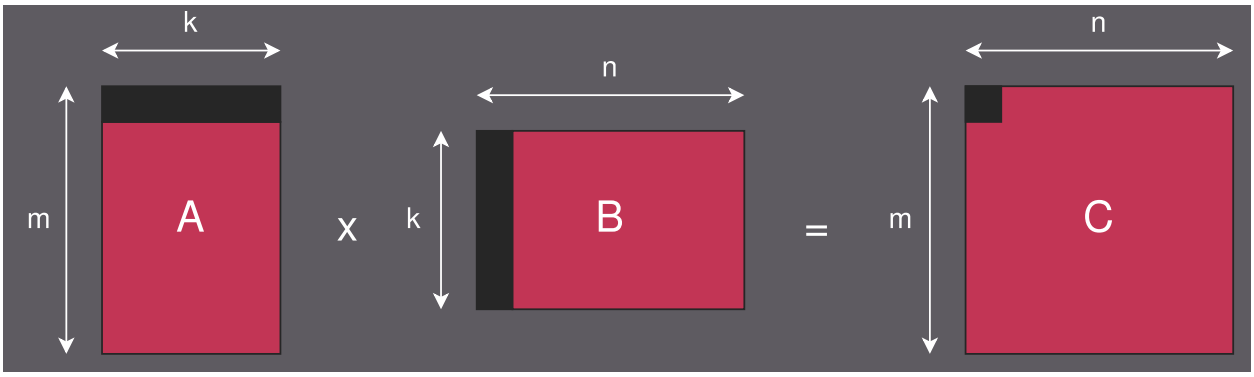
Most solutions, including the ones covered in this document, are given to the end users in a turnkey fashion via algorithm primitive libraries. These solutions might not be the fastest in all cases, but they are close to being the gold standard for carrying out certain operations as reasonably as possible.

15.7 Tiling and reuse: matrix multiplication

Matrix multiplication is one of the most important GPU workloads as it supports many computational problems. With the increased popularity of deep neural networks (DNNs), matrix multiplication has become vital and is found in the most computationally-intensive layers (i.e., fully connected and convolutional) of neural networks. In some cases, you might want to convert a problem to use matrix multiplications because GPUs are optimized to efficiently execute matrix multiplication, and to outperform an iterative implementation of the original problem.

15.7.1 Matrix multiplication fundamentals

A matrix multiplication operation involves three matrices, A, B, and C. Matrices A and B are input matrices, and matrix C is the output matrix. You can denote matrix A as having m rows and k columns and matrix B as having k rows and n columns. Matrix multiplication operations require the number of columns in A to equal the number of rows in B. Output matrix C then inherits m rows and n columns. Each element in C is the sum of the k multiplications of each element in the corresponding row of A and column of B, as formally represented in the figure below:



An iterative CPU implementation will apply three nested loops that traverse through m, n, and k. Each iteration will perform single multiplication and accumulation operations. Therefore, theoretically, a total of $2 \times m \times n \times k$ operations are required.

A GPU implementation by comparison is embarrassingly parallel. The GPU kernel parallelizes the m and n loops by calculating each element in the output matrix using a thread. Thus, you need a total of $m \times n$ threads, with each traversing

a loop of k iterations.

15.7.2 Naive implementation and memory access patterns

Although the naive implementation produces the correct result, the resulting performance is not ideal. Profiling the kernel execution shows that it reads from the GPU's Dynamic Random Access Memory (DRAM) several times more often than the combined size of the input matrices. To further explore this issue, you must examine how the threads read the input matrices.

In the naive implementation, each thread in the kernel reads an entire row in matrix A and a column in matrix B. The same data item must be accessed by multiple threads from different wavefronts. In the naive implementation, all memory reads are handled by the main memory system, and although there are repeated memory references made to the same location, these memory accesses should instead be serviced by the cache. However, given the limited cache space and the large number of threads involved, this places inordinate memory pressure on the caches. Then, as you increase the matrix size, the caches will become incapable of accommodating the data and memory reads, which will quickly push the data to the main memory, causing significant delays.

Ideally, you want to have more control over caches. When data are loaded into the cache, it should be reused by as many threads as possible before being released. In CPU programming, caches are fully transparent to the programmer and are typically not under their control. In contrast, GPUs provide Local Data Store (LDS) memory, which is effectively a programmable L1 cache.

```
// Naive matrix multiplication kernel
__global__ void matrix_multiply_naive(float *a, float *b, float *out, int m, int n,␣
↪int k)
{
    int gid_x = blockDim.x * blockIdx.x + threadIdx.x;
    int gid_y = blockDim.y * blockIdx.y + threadIdx.y;

    if (gid_x < n && gid_y < m)
    {
        float sum = 0.0f;
        for (int i = 0; i < k; ++i)
        {
            sum += a[gid_y * k + i] * b[i * n + gid_x];
        }
        out[gid_y * n + gid_x] = sum;
    }
}
```

15.7.3 LDS tiling optimization

To improve matrix multiplication performance, and make better use of LDS, you apply a common GPU programming technique called "tiling." Tiling methods use all threads from a workgroup to collectively process parts of the data before moving to the next portion of the problem. The first workgroup loads the data of matrices A and B in a tile to the LDS. Then, matrix multiplication is applied to the loaded tile. Next, the wavefronts move to the next tile, continuing to accumulate multiplication results.

The LDS tiling-based kernel implementation uses LDS memory to dramatically reduce global memory access. The kernel's interface is identical to the native implementation. The only special requirement is that the tile size parameter must be defined to determine the number of elements in each of the two dimensions to group into a single tile. The workgroup size must match the tile size to help ensure the correctness of the output.

```
// LDS tiling parameters
constexpr int base_tile_size = 16; // LDS tile dimension
```

(continues on next page)

(continued from previous page)

```
3
4  // LDS tiling kernel
5  __global__ void matrix_multiply_lds_tiling(float *a, float *b, float *out, int m, int␣
   ↪n, int k)
6  {
7      __shared__ float tilea[base_tile_size][base_tile_size];
8      __shared__ float tileb[base_tile_size][base_tile_size];
9
10     int tx = threadIdx.x;
11     int ty = threadIdx.y;
12
13     int row = blockIdx.y * base_tile_size + ty;
14     int col = blockIdx.x * base_tile_size + tx;
15
16     float sum = 0.0f;
17
18     int numtiles = (k + base_tile_size - 1) / base_tile_size;
19     for (int t = 0; t < numtiles; t++)
20     {
21         int acol = t * base_tile_size + tx;
22         if (row < m && acol < k)
23         {
24             tilea[ty][tx] = a[row * k + acol];
25         }
26         else
27         {
28             tilea[ty][tx] = 0.0f;
29         }
30
31         int brow = t * base_tile_size + ty;
32         if (brow < k && col < n)
33         {
34             tileb[ty][tx] = b[brow * n + col];
35         }
36         else
37         {
38             tileb[ty][tx] = 0.0f;
39         }
40
41         __syncthreads();
42
43         for (int i = 0; i < base_tile_size; i++)
44         {
45             sum += tilea[ty][i] * tileb[i][tx];
46         }
47
48         __syncthreads();
49     }
50
51     if (row < m && col < n)
52     {
53         out[row * n + col] = sum;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

The kernel implementation must be modified to use two loops, rather than just one. The outer loop addresses memory across the tile, and the inner loop accumulates multiplications. Before the loop starts, you allocate two buffers for the tiles for matrices A and B. In each iteration of the outer loop, you load the data from the main memory to the LDS. Here, the complexity of the implementation mainly comes from the index calculation and boundary checking. Next, you use the inner loop in a way similar to the naive implementation to perform multiplication and accumulation. Note that you need barriers before and after the inner loop to guarantee that all data are loaded and used, respectively. Finally, you store the final result in the output matrix.

LDS tiling is an effective way to reduce DRAM access and improve performance. Performance improvements with tiling depend on the specific GPU architecture and matrix dimensions, but significant speedups over the naive implementation are commonly observed on AMD GPUs.

15.7.4 Register tiling: The next optimization step

After implementing LDS tiling, the next optimization step is register tiling. While LDS tiling reduces global memory traffic by caching tiles in shared memory, register tiling further reduces memory traffic by keeping frequently accessed data in thread-local registers. This approach eliminates the need for many shared memory reads within the inner computation loop.

Register tiling works by having each thread compute multiple output elements and load the necessary input data into registers. The computation then proceeds using only register-to-register operations, with minimal shared memory access. This technique is particularly effective because register access is significantly faster than shared memory access, each thread has private registers eliminating bank conflicts, synchronization overhead between threads is reduced, and better instruction-level parallelism can be achieved.

The register tiling kernel uses a more complex thread-to-data mapping where each thread computes a small tile of output elements (typically 4×4) and loads the required input data into registers before performing the multiplication.

```
// Register tiling parameters
constexpr int thread_tile_m = 4; // Each thread computes 4x4 output
constexpr int thread_tile_n = 4;
constexpr int warp_threads_n = thread_tile_n; // 4
constexpr int block_warps_m = 2;
constexpr int block_warps_n = 2;
constexpr int k_tile_size = 16; // K-dimension tile size

// Register tiling kernel
__global__ void matrix_multiply_register_tiling(float *a, float *b, float *out, int m,
↪ int n, int k)
{
    constexpr int warp_threads_m = warp_size / warp_threads_n;
    constexpr int warp_tile_m = warp_threads_m * thread_tile_m;
    constexpr int warp_tile_n = warp_threads_n * thread_tile_n;
    constexpr int block_threads = warp_size * block_warps_m * block_warps_n;
    constexpr int block_tile_m = block_warps_m * warp_tile_m;
    constexpr int block_tile_n = block_warps_n * warp_tile_n;

    __shared__ float tilea[block_tile_m][k_tile_size + 4];
    __shared__ float tileb[k_tile_size][block_tile_n];

    int tid = threadIdx.y * blockDim.x + threadIdx.x;
```

(continues on next page)

(continued from previous page)

```
    int warp_id = tid / warp_size;
    int lane_id = tid % warp_size;

    int warp_row = warp_id / block_warps_n;
    int warp_col = warp_id % block_warps_n;

    int lane_row = lane_id / warp_threads_n;
    int lane_col = lane_id % warp_threads_n;

    int thread_row_in_block = warp_row * warp_tile_m + lane_row * thread_tile_m;
    int thread_col_in_block = warp_col * warp_tile_n + lane_col * thread_tile_n;

    int block_row_start = blockIdx.y * block_tile_m;
    int block_col_start = blockIdx.x * block_tile_n;

    float regc[thread_tile_m][thread_tile_n] = {0.0f};

    int numtiles = (k + k_tile_size - 1) / k_tile_size;
    for (int t = 0; t < numtiles; t++)
    {
        int elements_to_load_a = (block_tile_m * k_tile_size) / block_threads;
        for (int i = 0; i < elements_to_load_a; i++)
        {
            int idx = tid + i * block_threads;
            int tile_m = idx / k_tile_size;
            int tile_n = idx % k_tile_size;

            int global_m = block_row_start + tile_m;
            int global_n = t * k_tile_size + tile_n;

            if (global_m < m && global_n < k && tile_m < block_tile_m)
            {
                tilea[tile_m][tile_n] = a[global_m * k + global_n];
            }
            else if (tile_m < block_tile_m)
            {
                tilea[tile_m][tile_n] = 0.0f;
            }
        }

        int elements_to_load_b = (k_tile_size * block_tile_n) / block_threads;
        for (int i = 0; i < elements_to_load_b; i++)
        {
            int idx = tid + i * block_threads;
            int tile_m = idx / block_tile_n;
            int tile_n = idx % block_tile_n;

            int global_m = t * k_tile_size + tile_m;
            int global_n = block_col_start + tile_n;

            if (global_m < k && global_n < n && tile_m < k_tile_size)
            {
```

(continues on next page)

(continued from previous page)

```
                tileb[tile_m][tile_n] = b[global_m * n + global_n];
            }
            else if (tile_m < k_tile_size)
            {
                tileb[tile_m][tile_n] = 0.0f;
            }
        }

        __syncthreads();

        for (int kk = 0; kk < k_tile_size; kk++)
        {
            float rega[thread_tile_m];
            for (int i = 0; i < thread_tile_m; i++)
            {
                rega[i] = tilea[thread_row_in_block + i][kk];
            }

            float regb[thread_tile_n];
            for (int j = 0; j < thread_tile_n; j++)
            {
                regb[j] = tileb[kk][thread_col_in_block + j];
            }

            for (int i = 0; i < thread_tile_m; i++)
            {
                for (int j = 0; j < thread_tile_n; j++)
                {
                    regc[i][j] += rega[i] * regb[j];
                }
            }
        }

        __syncthreads();
    }

    for (int i = 0; i < thread_tile_m; i++)
    {
        for (int j = 0; j < thread_tile_n; j++)
        {
            int global_row = block_row_start + thread_row_in_block + i;
            int global_col = block_col_start + thread_col_in_block + j;

            if (global_row < m && global_col < n)
            {
                out[global_row * n + global_col] = regc[i][j];
            }
        }
    }
}
```

The register tiling implementation introduces thread tile computation, where each thread computes multiple output elements (defined by `thread_tile_m` and `thread_tile_n` parameters), and register-based accumulation, where inter-

mediate results are stored in registers instead of shared memory. Threads also use efficient data loading, loading data cooperatively into shared memory and then copying to registers for computation.

This approach typically provides additional performance improvements beyond LDS tiling. The actual speedup will vary based on GPU architecture, matrix dimensions, and other factors.

15.7.5 Practical implementation considerations

When implementing these optimizations for your specific application, consider the following factors:

- **Matrix dimensions**: The effectiveness of tiling depends on how well the matrix dimensions align with tile sizes and GPU compute unit resources
- **Memory requirements**: Larger tiles may not fit in available LDS space, requiring careful tuning of tile dimensions
- **GPU architecture**: Different AMD GPU architectures have varying LDS capacities and register availability, affecting optimal tile sizes
- **Workload characteristics**: Some applications may benefit more from LDS tiling alone, while others justify the added complexity of register tiling

Choosing the right optimization approach depends on your specific performance requirements, target hardware, and implementation complexity constraints.

15.8 Tiling and coalescing: matrix transpose

Matrix transpose is a fundamental linear algebra operation that serves as a building block for many tasks. At a high level, it does not involve calculations; it involves only memory movement. Because GPUs typically have significantly higher memory bandwidth than CPUs, they have a unique advantage when performing matrix transpose operations.

Like the previous examples, the matrix transpose operation can benefit from a significantly parallel implementation, in which each thread is responsible for moving a matrix element.

```
// Simple matrix transpose kernel
__global__ void transpose_naive_kernel(float* in, float* out, int width, int height)
{
    int x_index = blockIdx.x * tile_dim + threadIdx.x;
    int y_index = blockIdx.y * tile_dim + threadIdx.y;

    int in_index  = y_index * width + x_index;
    int out_index = x_index * height + y_index;

    out[out_index] = in[in_index];
}
```

15.8.1 Memory access patterns

Matrix transpose operations present a unique challenge in terms of memory access patterns. Theoretically, the number of reads and writes should be approximately equal, as you write all the data that you read from the input matrix. However, in practice, the actual memory traffic can be significantly higher due to the nature of how threads access memory.

To understand this behavior, you need to examine the memory access pattern carefully. Unlike matrix multiplication where the same data might be accessed multiple times, in matrix transpose each element is typically accessed only once. The issue instead lies in the address pattern of the read and write operations.

In a typical matrix transpose implementation, adjacent threads read data horizontally (row-wise) and write data vertically (column-wise). Since matrices are stored in row-major order, adjacent threads read data from contiguous memory

addresses, which allows the GPU to coalesce these read operations into fewer memory transactions. However, when writing data vertically, the threads access memory addresses with large strides (the distance between consecutive elements in a column). This prevents write coalescing, as each thread's write operation typically requires a separate memory transaction.

As a result, the read operations are coalescable and efficient, while the write operations are non-coalescable and generate significantly more memory traffic. This fundamental characteristic of matrix transpose operations makes them an excellent example for understanding the importance of memory access patterns and coalescing in GPU programming.

15.8.2 Memory coalescing with LDS

It is possible to convert non-coalescable memory access to coalescable ones using the Local Data Store (LDS). The optimization approach splits the matrix transpose process into two steps. First, you move the data from the input matrix to LDS memory. For this, you read the data horizontally, ensuring that the memory access is coalesced. When the LDS buffer is filled, you then move the data from the LDS to the main memory. In this step, you read the data vertically and write them horizontally. Since LDS memory has a significantly higher bandwidth than global memory, reading them horizontally and vertically does not impact overall performance. However, you write the data horizontally so that the writes to the main memory can also be coalesced. Thus, you use the LDS as an intermediate data transfer buffer between the input and output matrices, allowing you to make both read and write operations coalescable.

The LDS-based kernel implementation uses the same interface as the naive implementation. The kernel uses a shared memory tile to temporarily store data before writing it back to the output matrix. This approach ensures that both the read from global memory and the write to global memory are coalescable, which significantly improves performance.

```
// LDS-based matrix transpose kernel for better memory coalescing
__global__ void transpose_lds_kernel(float* in, float* out, int width, int height)
{
    __shared__ float tile[tile_dim][tile_dim];

    int x_tile_index = blockIdx.x * tile_dim;
    int y_tile_index = blockIdx.y * tile_dim;

    int in_index  = (y_tile_index + threadIdx.y) * width + (x_tile_index + threadIdx.
↪x);
    int out_index = (x_tile_index + threadIdx.y) * height + (y_tile_index + threadIdx.
↪x);

    tile[threadIdx.y][threadIdx.x] = in[in_index];

    __syncthreads();

    out[out_index] = tile[threadIdx.x][threadIdx.y];
}
```

In the LDS implementation, each threadblock loads a tile of the input matrix into shared memory in a coalesced fashion. The threads then transpose the data within the shared memory tile and write it back to the output matrix in a coalesced manner. This approach effectively solves the non-coalescable write problem of the naive implementation.

The LDS-based approach provides significant performance benefits because both read and write operations to global memory are now coalescable, reducing the total number of memory transactions. By using LDS as an intermediate buffer, you reduce the pressure on the GPU's memory system and make better use of the available bandwidth. The tile-based approach improves cache hit rates by reusing data within the shared memory before writing it back to global memory.

This optimization technique is particularly important for memory-bound kernels like matrix transpose, where the performance is limited by memory bandwidth rather than computational throughput.

15.8.3 Implementation guidelines and performance considerations

When implementing LDS-based matrix transpose optimization, consider the following practical aspects:

- **Tile size selection**: Choose tile dimensions that balance LDS memory usage and coalescing benefits. Common sizes include 16 × 16 or 32 × 32 tiles, but the optimal size depends on your specific GPU architecture
- **Bank conflict avoidance**: Structure your data access patterns to minimize memory bank conflicts in the LDS, which can reduce the effectiveness of the optimization
- **Memory alignment**: Ensure proper memory alignment for both global memory accesses to maximize coalescing benefits
- **Edge case handling**: Account for matrix dimensions that are not perfectly divisible by your chosen tile size, implementing proper boundary checks

The performance impact of LDS-based optimization varies depending on matrix dimensions, GPU architecture, and memory subsystem characteristics. For larger matrices, the benefits are typically more pronounced as memory bandwidth becomes the dominant performance factor.

CHAPTER

SIXTEEN

MULTI-GPU PROGRAMMING

The multi-GPU Programming section explains how to leverage multiple 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 utilize 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.

16.1 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;
}
```

16.2 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;
}
```

16.3 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;
}
```

16.4 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

SEVENTEEN

ROCM INSTALL

If you're new to ROCm™, we recommend using the *Quick start installation guide*.

Note

- If you're using ROCm with AMD Radeon GPUs or Ryzen APUs for graphics workloads, see the Use ROCm on Radeon and Ryzen.

To install ROCm, you can use the package manager. You can also opt for single-version or multi-version installation. These topics are described in detail in the following sections.

17.1 Installation methods

- *Package manager*
- *Multi-version installation*
- *ROCm Runfile Installer*

17.1.1 Package manager

The distribution's package manager lets the user install, upgrade and uninstall using familiar commands and workflows. Third party ecosystem support is the same as your OS package manager. See *Installation via native package manager* for instructions based on the operating system.

17.1.2 Multi-version installation

A multi-version ROCm installation handles situations where users need multiple versions of ROCm on the same machine for compatibility with different applications and hardware, testing, and other use cases. For instructions, see Installing multiple ROCm versions.

17.1.3 ROCm Runfile Installer

The ROCm Runfile Installer lets you install ROCm without using a native Linux package management system. It can be used with or without network or internet access. See ROCm Runfile Installer for instructions.

17.2 Installation prerequisites

Before installing ROCm, complete the following prerequisites.

1. Confirm the system has a supported Linux version.

- To obtain the Linux distribution information, use the following command:

```
uname -m && cat /etc/*release
```

- Confirm that your Linux distribution matches a supported distribution.

  Example: Running the preceding command on an Ubuntu system produces the following output:

```
x86_64
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=24.04
DISTRIB_CODENAME=noble
DISTRIB_DESCRIPTION="Ubuntu 24.04.3 LTS"
```

2. Verify the kernel version.

   - To check the kernel version of your Linux system, type the following command:

```
uname -srmv
```

   Example: The preceding command lists the kernel version in the following format:

```
Linux 6.8.0-50-generic #51-Ubuntu SMP PREEMPT_DYNAMIC Sat Nov  9 17:58:29 UTC␣
↪2024 x86_64
```

   - Confirm that your kernel version matches the system requirements, as listed in Supported operating systems.

17.2.1 Register your Enterprise Linux

If you're using Red Hat Enterprise Linux (RHEL) or SUSE Linux Enterprise Server (SLES), register your operating system to ensure you're able to download and install packages.

Ubuntu

There is no registration required for Ubuntu.

Debian

There is no registration required for Debian.

Red Hat Enterprise Linux

10.1

Typically you can register by following the step-by-step user interface. If you need to register by command line, use the following commands:

```
subscription-manager register --username <username> --password <password>
```

More details about registering for RHEL

10.0

Typically you can register by following the step-by-step user interface. If you need to register by command line, use the following commands:

```
subscription-manager register --username <username> --password <password>
```

More details about registering for RHEL

9.7

Typically you can register by following the step-by-step user interface. If you need to register by command line, use the following commands:

```
subscription-manager register --username <username> --password <password>
subscription-manager attach --auto
```

More details about registering for RHEL

9.6

Typically you can register by following the step-by-step user interface. If you need to register by command line, use the following commands:

```
subscription-manager register --username <username> --password <password>
subscription-manager attach --auto
```

More details about registering for RHEL

9.4

Typically you can register by following the step-by-step user interface. If you need to register by command line, use the following commands:

```
subscription-manager register --username <username> --password <password>
subscription-manager attach --auto
```

More details about registering for RHEL

8.10

Typically you can register by following the step-by-step user interface. If you need to register by command line, use the following commands:

```
subscription-manager register --username <username> --password <password>
subscription-manager attach --auto
```

More details about registering for RHEL

Oracle Linux

There is no registration required for Oracle Linux.

SUSE Linux Enterprise Server

Typically you can register by following the step-by-step user interface. If you need to register by command line, use the following commands:

```
sudo SUSEConnect -r <REGCODE>
```

More details about registering for SLES

Rocky Linux

There is no registration required for Rocky Linux.

17.2.2 Update your Enterprise Linux

If you are using Red Hat Enterprise Linux (RHEL), SUSE Linux Enterprise Servers (SLES), or Oracle Linux (OL), it is recommended that you update your operating system to the latest packages from the Linux distribution. This is a requirement for newer hardware on older versions of RHEL, SLES, or OL.

Ubuntu

There is no update required for Ubuntu.

Debian

There is no update required for Debian.

Red Hat Enterprise Linux

10.1

```
sudo dnf update redhat-release
sudo dnf update --releasever=10.1 --exclude=\*release\*
```

10.0

```
sudo dnf update --releasever=10.0 --exclude=\*release\*
```

9.7

```
sudo dnf update --releasever=9.7 --exclude=\*release\*
```

9.6

```
sudo dnf update --releasever=9.6 --exclude=\*release\*
```

9.4

```
sudo dnf update --releasever=9.4 --exclude=\*release\*
```

8.10

```
sudo dnf update --releasever=8.10 --exclude=\*release\*
```

Oracle Linux

10.1

```
sudo dnf update --releasever=10.1 --exclude=\*release\*
```

9.7

```
sudo dnf update --releasever=9.7 --exclude=\*release\*
```

8.10

```
sudo dnf update --releasever=8.10 --exclude=\*release\*
```

SUSE Linux Enterprise Server

```
sudo zypper update
```

Rocky Linux

There is no update required for Rocky Linux.

Important

To apply all settings, reboot your system.

17.2.3 Additional package repositories

For some distributions, the ROCm installation packages depend on packages that aren't included in the default package repositories. These external repositories need to be sourced before installation. Use the following instructions specific to your distribution to add the necessary repositories.

Ubuntu

All ROCm installation packages are available in the default Ubuntu repositories.

Debian

All ROCm installation packages are available in the default Debian repositories.

Red Hat Enterprise Linux

1. Add the EPEL repository.

10.1

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-10.noarch.rpm
sudo rpm -ivh epel-release-latest-10.noarch.rpm
```

10.0

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-10.noarch.rpm
sudo rpm -ivh epel-release-latest-10.noarch.rpm
```

9.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
```

9.6

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
```

9.4

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
```

8.10

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
sudo rpm -ivh epel-release-latest-8.noarch.rpm
```

2. Enable the CodeReady Linux Builder (CRB) repository.

10.1

```
sudo dnf config-manager --enable codeready-builder-for-rhel-10-x86_64-rpms
```

10.0

```
sudo dnf config-manager --enable codeready-builder-for-rhel-10-x86_64-rpms
```

9.7

```
sudo dnf config-manager --enable codeready-builder-for-rhel-9-x86_64-rpms
```

9.6

```
sudo dnf config-manager --enable codeready-builder-for-rhel-9-x86_64-rpms
```

9.4

```
sudo dnf config-manager --enable codeready-builder-for-rhel-9-x86_64-rpms
```

8.10

```
sudo dnf config-manager --enable codeready-builder-for-rhel-8-x86_64-rpms
```

Oracle Linux

1. Add the EPEL repository.

10.1

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-10.noarch.rpm
sudo rpm -ivh epel-release-latest-10.noarch.rpm
```

9.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
```

8.10

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
sudo rpm -ivh epel-release-latest-8.noarch.rpm
```

2. Enable the CodeReady Linux Builder (CRB) repository.

```
sudo crb enable
```

SUSE Linux Enterprise Server

Add a few modules with SUSEConnect and the science repository.

15.7

```
sudo SUSEConnect -p sle-module-desktop-applications/15.7/x86_64
sudo SUSEConnect -p sle-module-development-tools/15.7/x86_64
sudo SUSEConnect -p PackageHub/15.7/x86_64
sudo zypper install zypper
sudo zypper addrepo https://download.opensuse.org/repositories/science/SLE_15_SP5/
↪science.repo
```

Rocky Linux

1. Add the EPEL repository.

9.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
```

2. Enable the CodeReady Linux Builder (CRB) repository.

   In order to enable CRB, you may need to install `dnf-plugin-config-manager` first.

```
sudo dnf install dnf-plugin-config-manager
sudo crb enable
```

17.2.4 Additional development packages

ROCm installation requires additional packages for operation and development.

To install the required packages, use the following instructions specific to your distribution:

Ubuntu

```
sudo apt install python3-setuptools python3-wheel
```

Debian

```
sudo apt install python3-setuptools python3-wheel
```

Red Hat Enterprise Linux

```
sudo dnf install python3-setuptools python3-wheel
```

Oracle Linux

```
sudo dnf install python3-setuptools python3-wheel
```

SUSE Linux Enterprise Server

```
sudo zypper install python3-setuptools python3-wheel
```

Rocky Linux

```
sudo dnf install python3-setuptools python3-wheel
```

Optionally, if configuring the *post-ROCm installation* using `environment-modules`, install the following:

Ubuntu

```
sudo apt install environment-modules
```

Debian

```
sudo apt install environment-modules
```

Red Hat Enterprise Linux

```
sudo dnf install environment-modules
```

Oracle Linux

```
sudo dnf install environment-modules
```

SUSE Linux Enterprise Server

```
sudo zypper install environment-modules

# Create a link for installed modules version
version=$(rpm -qa | grep '^Modules-' | awk -F'-' '{print $2}')
sudo ln -s /usr/share/Modules/$version/modulefiles /usr/share/Modules/modulefiles
```

Rocky Linux

```
sudo dnf install environment-modules
```

17.2.5 Configuring permissions for GPU access

There are two primary methods to configure GPU access for ROCm: group membership or udev rules. Each method has its own advantages, and the choice depends on your specific requirements and system management preferences.

17.2.5.1 1. Using group membership

By default, GPU access is managed through membership in the `video` and `render` groups. The `video` and `render` groups are system groups in Linux used to manage access to graphics hardware and related functionality. Traditionally, the `video` group is used to control access to video devices, including graphics cards and video capture devices. The `render` group is more recent and specifically controls access to GPU rendering capabilities through Direct Rendering Manager (DRM) render nodes.

1. To check the groups in your system, issue the following command:

   ```
   groups
   ```

2. Add yourself to the `video` and `render` groups:

   ```
   sudo usermod -a -G video,render $LOGNAME
   ```

3. Optionally, add other users to the `video` and `render` groups:

   ```
   sudo usermod -a -G video,render user1
   sudo usermod -a -G video,render user2
   ```

4. To add all future users to the render and video groups by default, run the following commands:

   ```
   echo 'ADD_EXTRA_GROUPS=1' | sudo tee -a /etc/adduser.conf
   echo 'EXTRA_GROUPS=video' | sudo tee -a /etc/adduser.conf
   echo 'EXTRA_GROUPS=render' | sudo tee -a /etc/adduser.conf
   ```

17.2.5.2 2. Using udev rules

A flexible way to manage device permissions is to use udev rules. They apply system-wide, can be easily deployed via configuration management tools, and eliminate the need for user group management. This method provides more granular control over GPU access.

GPU access may be granted to either all users or a custom group:

17.2.5.2.1 a. Grant GPU access to all users on the system

To set up udev rules, install the package using the following instructions specific to your distribution:

Ubuntu

24.04

```
sudo apt update
wget https://repo.radeon.com/amdgpu/30.30.3/ubuntu/pool/main/a/amdgpu-insecure-
↪instinct-udev-rules/amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.24.04_all.
↪deb
sudo apt install ./amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.24.04_all.deb
```

22.04

```
sudo apt update
wget https://repo.radeon.com/amdgpu/30.30.3/ubuntu/pool/main/a/amdgpu-insecure-
↪instinct-udev-rules/amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.22.04_all.
↪deb
sudo apt install ./amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.22.04_all.deb
```

Debian

13

```
sudo apt update
wget https://repo.radeon.com/amdgpu/30.30.3/ubuntu/pool/main/a/amdgpu-insecure-
↪instinct-udev-rules/amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.24.04_all.
↪deb
sudo apt install ./amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.24.04_all.deb
```

12

```
sudo apt update
wget https://repo.radeon.com/amdgpu/30.30.3/ubuntu/pool/main/a/amdgpu-insecure-
↪instinct-udev-rules/amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.22.04_all.
↪deb
sudo apt install ./amdgpu-insecure-instinct-udev-rules_30.30.3.0-2327507.22.04_all.deb
```

Red Hat Enterprise Linux

10.1

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/10/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el10.noarch.rpm
```

10.0

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/10/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el10.noarch.rpm
```

9.7

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/9.7/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el9.noarch.rpm
```

9.6

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/9.6/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el9.noarch.rpm
```

9.4

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/9.4/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el9.noarch.rpm
```

8.10

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/8/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el8.noarch.rpm
```

Oracle Linux

10.1

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/10/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el10.noarch.rpm
```

9.7

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/9.7/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el9.noarch.rpm
```

8.10

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/8/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el8.noarch.rpm
```

SUSE Linux Enterprise Server

15.7

```
sudo zypper --no-gpg-checks install https://repo.radeon.com/amdgpu/30.30.3/sle/15.7/
↪main/x86_64/amdgpu-insecure-instinct-udev-rules-30.30.3.0-2327507.noarch.rpm
```

Rocky Linux

9.7

```
sudo dnf install https://repo.radeon.com/amdgpu/30.30.3/el/9.7/main/x86_64/amdgpu-
↪insecure-instinct-udev-rules-30.30.3.0-2327507.el9.noarch.rpm
```

17.2.5.2.2 b. Grant GPU access to a custom group

1. Create a new group (e.g., `devteam`):

```
sudo groupadd devteam
```

2. Add users to the new group:

```
sudo usermod -a -G devteam dev1
sudo usermod -a -G devteam dev2
```

3. Create udev rules to assign GPU devices to this group:

   Create a file `/etc/udev/rules.d/70-amdgpu.rules` with:

```
KERNEL=="kfd", GROUP="devteam", MODE="0660"
SUBSYSTEM=="drm", KERNEL=="renderD*", GROUP="devteam", MODE="0660"
```

4. Reload the udev rules:

```
sudo udevadm control --reload-rules && sudo udevadm trigger
```

This configuration grants all users in the `devteam` group read and write access to AMD GPU resources, including the AMD Kernel-mode GPU Driver (KMD) and Direct Rendering Manager (DRM) devices.

17.2.6 Disable integrated graphics (IGP)

ROCm doesn't currently support integrated graphics. If your system has an AMD IGP installed, disable it in the BIOS prior to using ROCm. If the driver can enumerate the IGP, the ROCm runtime might crash the system, even when omission was specified via HIP_VISIBLE_DEVICES.

17.2.7 Secure Boot

When installing the AMDGPU driver with Secure Boot enabled, you must sign `amdgpu-dkms` to prevent potential system loading issues. For more information, see Secure Boot Support. If you prefer not to sign the AMDGPU driver, you can disable Secure Boot from the BIOS settings instead.

17.3 Quick start installation guide

> **Note**
>
> See Use ROCm on Radeon and Ryzen for instructions on installing ROCm on systems with AMD Radeon™ GPUs or Ryzen™ APUs for graphics workloads.

Before proceeding, ensure your kernel meets the ROCm system requirements. Then select your operating system and version, and run the provided commands to install the AMD GPU and ROCm.

For detailed guidance, see Installation via native package manager for AMD GPU installation and Detailed install for ROCm installation.

17.3.1 Installing

17.3.1.1 Register repositories

Ubuntu

24.04

```
wget https://repo.radeon.com/amdgpu-install/7.2.3/ubuntu/noble/amdgpu-install_7.2.3.
↪70203-1_all.deb
sudo apt install ./amdgpu-install_7.2.3.70203-1_all.deb
sudo apt update
```

22.04

```
wget https://repo.radeon.com/amdgpu-install/7.2.3/ubuntu/jammy/amdgpu-install_7.2.3.
↪70203-1_all.deb
sudo apt install ./amdgpu-install_7.2.3.70203-1_all.deb
sudo apt update
```

Debian

13

```
wget https://repo.radeon.com/amdgpu-install/7.2.3/ubuntu/noble/amdgpu-install_7.2.3.
↪70203-1_all.deb
sudo apt install ./amdgpu-install_7.2.3.70203-1_all.deb
sudo apt update
```

12

```
wget https://repo.radeon.com/amdgpu-install/7.2.3/ubuntu/jammy/amdgpu-install_7.2.3.
↪70203-1_all.deb
sudo apt install ./amdgpu-install_7.2.3.70203-1_all.deb
sudo apt update
```

Red Hat Enterprise Linux

10.1

Before installing ROCm on RHEL, *register and update your Enterprise Linux*.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/rhel/10/amdgpu-install-
↪7.2.3.70203-1.el10.noarch.rpm
sudo dnf clean all
```

10.0

Before installing ROCm on RHEL, *register and update your Enterprise Linux*.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/rhel/10/amdgpu-install-
↪7.2.3.70203-1.el10.noarch.rpm
sudo dnf clean all
```

9.7

Before installing ROCm on RHEL, *register and update your Enterprise Linux*.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/rhel/9.7/amdgpu-install-
↪7.2.3.70203-1.el9.noarch.rpm
sudo dnf clean all
```

9.6

Before installing ROCm on RHEL, *register and update your Enterprise Linux*.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/rhel/9.6/amdgpu-install-
↪7.2.3.70203-1.el9.noarch.rpm
sudo dnf clean all
```

9.4

Before installing ROCm on RHEL, *register and update your Enterprise Linux*.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/rhel/9.4/amdgpu-install-
↪7.2.3.70203-1.el9.noarch.rpm
sudo dnf clean all
```

8.10

Before installing ROCm on RHEL, *register and update your Enterprise Linux*.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/rhel/8/amdgpu-install-7.
↪2.3.70203-1.el8.noarch.rpm
sudo dnf clean all
```

Oracle Linux

10.1

Before installing ROCm on OL, update your Enterprise Linux.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/el/10/amdgpu-install-7.
↪2.3.70203-1.el10.noarch.rpm
sudo dnf clean all
```

9.7

Before installing ROCm on OL, update your Enterprise Linux.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/el/9.7/amdgpu-install-7.
↪2.3.70203-1.el9.noarch.rpm
sudo dnf clean all
```

8.10

Before installing ROCm on OL, update your Enterprise Linux.

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/el/8/amdgpu-install-7.2.
↪3.70203-1.el8.noarch.rpm
sudo dnf clean all
```

SUSE Linux Enterprise Server

15.7

Before installing ROCm on SLES, register and update your Enterprise Linux.

```
sudo SUSEConnect -p sle-module-desktop-applications/15.7/x86_64
sudo SUSEConnect -p sle-module-development-tools/15.7/x86_64
sudo SUSEConnect -p PackageHub/15.7/x86_64
sudo zypper install zypper
sudo zypper --no-gpg-checks install https://repo.radeon.com/amdgpu-install/7.2.3/sle/
↪15.7/amdgpu-install-7.2.3.70203-1.noarch.rpm
sudo zypper --gpg-auto-import-keys refresh
```

Rocky Linux

9.7

```
sudo dnf install https://repo.radeon.com/amdgpu-install/7.2.3/el/9.7/amdgpu-install-7.
↪2.3.70203-1.el9.noarch.rpm
sudo dnf clean all
```

17.3.1.2 Install kernel driver

Ubuntu

24.04

> **Caution**
>
> Remove any AMD GPU driver from a previous installation by following the uninstall steps in *Uninstall kernel driver*.

```
sudo apt install "linux-headers-$(uname -r)" "linux-modules-extra-$(uname -r)"
sudo apt install amdgpu-dkms
```

22.04

> **Caution**
>
> Remove any AMD GPU driver from a previous installation by following the uninstall steps in *Uninstall kernel driver*.

```
sudo apt install "linux-headers-$(uname -r)" "linux-modules-extra-$(uname -r)"
sudo apt install amdgpu-dkms
```

Debian

13

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo apt install "linux-headers-$(uname -r)"
sudo apt install amdgpu-dkms
```

12

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo apt install "linux-headers-$(uname -r)"
sudo apt install amdgpu-dkms
```

Red Hat Enterprise Linux

10.1

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-headers-$(uname -r)" "kernel-devel-$(uname -r)" "kernel-
↪devel-matched-$(uname -r)"
sudo dnf install amdgpu-dkms
```

10.0

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-headers-$(uname -r)" "kernel-devel-$(uname -r)" "kernel-
↪devel-matched-$(uname -r)"
sudo dnf install amdgpu-dkms
```

9.7

> **Caution**
>
> Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-headers-$(uname -r)" "kernel-devel-$(uname -r)" "kernel-
↪devel-matched-$(uname -r)"
sudo dnf install amdgpu-dkms
```

9.6

> **Caution**
>
> Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-headers-$(uname -r)" "kernel-devel-$(uname -r)" "kernel-
↪devel-matched-$(uname -r)"
sudo dnf install amdgpu-dkms
```

9.4

> **Caution**
>
> Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-headers-$(uname -r)" "kernel-devel-$(uname -r)" "kernel-
↪devel-matched-$(uname -r)"
sudo dnf install amdgpu-dkms
```

8.10

> **Caution**
>
> Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-headers-$(uname -r)" "kernel-devel-$(uname -r)"
sudo dnf install amdgpu-dkms
```

Oracle Linux

10.1

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-uek-devel-$(uname -r)"
sudo dnf install amdgpu-dkms
```

9.7

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-uek-devel-$(uname -r)"
sudo dnf install amdgpu-dkms
```

8.10

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-uek-devel-$(uname -r)"
sudo dnf install amdgpu-dkms
```

SUSE Linux Enterprise Server

15.7

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo zypper install kernel-default-devel
sudo zypper install amdgpu-dkms
```

Rocky Linux

9.7

Caution

Remove any AMD GPU driver from a previous installation by following uninstallation steps in *Uninstall kernel driver*.

```
sudo dnf install "kernel-headers" "kernel-devel" "kernel-devel-matched"
sudo dnf install amdgpu-dkms
```

Important

To apply all settings, reboot your system.

17.3.1.3 Install ROCm

Ubuntu

24.04

```
sudo apt install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo apt install rocm
```

22.04

```
sudo apt install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo apt install rocm
```

Debian

13

```
sudo apt install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo apt install rocm
```

12

```
sudo apt install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo apt install rocm
```

Red Hat Enterprise Linux

10.1

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-10.noarch.rpm
sudo rpm -ivh epel-release-latest-10.noarch.rpm
sudo dnf config-manager --enable codeready-builder-for-rhel-10-x86_64-rpms
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
```

(continues on next page)

(continued from previous page)

```
↪video groups
sudo dnf install rocm
```

10.0

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-10.noarch.rpm
sudo rpm -ivh epel-release-latest-10.noarch.rpm
sudo dnf config-manager --enable codeready-builder-for-rhel-10-x86_64-rpms
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

9.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
sudo dnf config-manager --enable codeready-builder-for-rhel-9-x86_64-rpms
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

9.6

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
sudo dnf config-manager --enable codeready-builder-for-rhel-9-x86_64-rpms
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

9.4

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
sudo dnf config-manager --enable codeready-builder-for-rhel-9-x86_64-rpms
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

8.10

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
sudo rpm -ivh epel-release-latest-8.noarch.rpm
sudo dnf config-manager --enable codeready-builder-for-rhel-8-x86_64-rpms
sudo dnf install python3-setuptools python3-wheel
```

(continues on next page)

(continued from previous page)

```
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

Oracle Linux

10.1

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-10.noarch.rpm
sudo rpm -ivh epel-release-latest-10.noarch.rpm
sudo crb enable
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

9.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
sudo crb enable
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

8.10

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm
sudo rpm -ivh epel-release-latest-8.noarch.rpm
sudo crb enable
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

SUSE Linux Enterprise Server

15.7

```
sudo zypper addrepo https://download.opensuse.org/repositories/science/SLE_15_SP5/
↪science.repo
sudo zypper install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo zypper install rocm
```

Rocky Linux

9.7

```
wget https://dl.fedoraproject.org/pub/epel/epel-release-latest-9.noarch.rpm
sudo rpm -ivh epel-release-latest-9.noarch.rpm
sudo dnf install dnf-plugin-config-manager
sudo crb enable
sudo dnf install python3-setuptools python3-wheel
sudo usermod -a -G render,video $LOGNAME # Add the current user to the render and␣
↪video groups
sudo dnf install rocm
```

Important

To apply all settings, reboot your system.

Note

Quick Start enables GPU access for the current user only. To grant GPU access to all users, see Configuring permissions for GPU access.

After completing the installation, review the *Post-installation instructions*. If you have issues with your installation, see Troubleshooting.

17.3.2 Uninstalling

17.3.2.1 Uninstall ROCm

Ubuntu

24.04

```
sudo apt autoremove rocm
sudo apt autoremove rocm-core
```

22.04

```
sudo apt autoremove rocm
sudo apt autoremove rocm-core
```

Debian

13

```
sudo apt autoremove rocm
sudo apt autoremove rocm-core
```

12

```
sudo apt autoremove rocm
sudo apt autoremove rocm-core
```

Red Hat Enterprise Linux

10.1

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

10.0

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

9.7

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

9.6

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

9.4

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

8.10

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

Oracle Linux

10.1

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

9.7

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

8.10

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

SUSE Linux Enterprise Server

15.7

```
sudo zypper remove rocm
sudo zypper remove rocm-core amdgpu-core
```

Rocky Linux

9.7

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

17.3.2.2 Uninstall kernel driver

Ubuntu

24.04

```
sudo apt autoremove amdgpu-dkms
```

22.04

```
sudo apt autoremove amdgpu-dkms
```

Debian

13

```
sudo apt autoremove amdgpu-dkms
```

12

```
sudo apt autoremove amdgpu-dkms
```

Red Hat Enterprise Linux

10.1

```
sudo dnf remove amdgpu-dkms
```

10.0

```
sudo dnf remove amdgpu-dkms
```

9.7

```
sudo dnf remove amdgpu-dkms
```

9.6

```
sudo dnf remove amdgpu-dkms
```

9.4

```
sudo dnf remove amdgpu-dkms
```

8.10

```
sudo dnf remove amdgpu-dkms
```

Oracle Linux

10.1

```
sudo dnf remove amdgpu-dkms
```

9.7

```
sudo dnf remove amdgpu-dkms
```

8.10

```
sudo dnf remove amdgpu-dkms
```

SUSE Linux Enterprise Server

15.7

```
sudo zypper remove amdgpu-dkms amdgpu-dkms-firmware
```

Rocky Linux

9.7

```
sudo dnf remove amdgpu-dkms
```

Important

To apply all settings, reboot your system.

17.3.2.3 Remove repositories

Ubuntu

24.04

```
sudo apt purge amdgpu-install
sudo apt autoremove

# Clear the cache and clean the system
sudo rm -rf /var/cache/apt/*
sudo apt clean all
sudo apt update
```

22.04

```
sudo apt purge amdgpu-install
sudo apt autoremove

# Clear the cache and clean the system
sudo rm -rf /var/cache/apt/*
sudo apt clean all
sudo apt update
```

Debian

13

```
sudo apt purge amdgpu-install
sudo apt autoremove

# Clear the cache and clean the system
sudo rm -rf /var/cache/apt/*
sudo apt clean all
sudo apt update
```

12

```
sudo apt purge amdgpu-install
sudo apt autoremove

# Clear the cache and clean the system
sudo rm -rf /var/cache/apt/*
sudo apt clean all
sudo apt update
```

Red Hat Enterprise Linux

10.1

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
```

(continues on next page)

(continued from previous page)

```
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

10.0

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

9.7

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

9.6

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

9.4

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

8.10

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

Oracle Linux

10.1

```
sudo dnf remove amdgpu-install

```

(continues on next page)

(continued from previous page)

```
# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

9.7

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

8.10

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

SUSE Linux Enterprise Server

15.7

```
sudo zypper remove amdgpu-install

# Clear the cache and clean the system
sudo zypper clean --all
sudo zypper refresh
```

Rocky Linux

9.7

```
sudo dnf remove amdgpu-install

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

17.4 Installation via native package manager

Select the install instructions for your operating system

Install

- *Ubuntu*
- *Debian*
- *Red Hat Enterprise Linux*
- *Oracle Linux*

- *Rocky Linux*
- *SUSE Linux Enterprise Server*

Uninstall

- *Ubuntu*
- *Debian*
- *Red Hat Enterprise Linux*
- *Oracle Linux*
- *Rocky Linux*
- *SUSE Linux Enterprise Server*

See also: System requirements (Linux). If you encounter install issues, you can refer to the troubleshooting page.

17.4.1 Ubuntu native installation

Caution

Ensure that the *Installation prerequisites* are met.

Note

The following installation steps also apply when upgrading from a previous ROCm version.

17.4.1.1 Registering ROCm repositories

17.4.1.1.1 Package signing key

Download and convert the package signing key.

```
# Make the directory if it doesn't exist yet.
# This location is recommended by the distribution maintainers.
sudo mkdir --parents --mode=0755 /etc/apt/keyrings

# Download the key, convert the signing-key to a full
# keyring required by apt and store in the keyring directory
wget https://repo.radeon.com/rocm/rocm.gpg.key -O - | \
    gpg --dearmor | sudo tee /etc/apt/keyrings/rocm.gpg > /dev/null
```

Note

The GPG key may change; ensure it is updated when installing a new release. If the key signature verification fails while updating, re-add the key from the ROCm to the apt repository as mentioned above.

17.4.1.1.2 Register packages

Ubuntu 24.04

```
sudo tee /etc/apt/sources.list.d/rocm.list << EOF
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/rocm/
↪apt/7.2.3 noble main
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/
↪graphics/7.2.3/ubuntu noble main
EOF

sudo tee /etc/apt/preferences.d/rocm-pin-600 << EOF
Package: *
Pin: release o=repo.radeon.com
Pin-Priority: 600
EOF

sudo apt update
```

Ubuntu 22.04

```
sudo tee /etc/apt/sources.list.d/rocm.list << EOF
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/rocm/
↪apt/7.2.3 jammy main
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/
↪graphics/7.2.3/ubuntu jammy main
EOF

sudo tee /etc/apt/preferences.d/rocm-pin-600 << EOF
Package: *
Pin: release o=repo.radeon.com
Pin-Priority: 600
EOF

sudo apt update
```

17.4.1.2 Installing

17.4.1.2.1 Install kernel driver

For information about the AMDGPU driver installation, see the Ubuntu native installation in the AMD Instinct Data Center GPU Documentation.

For information about driver compatibility, see User and AMD GPU Driver (amdgpu) support matrix.

17.4.1.2.2 Install ROCm

```
sudo apt install rocm
```

17.4.1.3 Post-installation

Complete the *Post-installation instructions*.

17.4.1.4 Uninstalling

17.4.1.4.1 Uninstall ROCm meta packages

```
sudo apt autoremove rocm
sudo apt autoremove rocm-core
```

17.4.1.4.2 Remove ROCm repositories

```
# Remove the repositories
sudo rm /etc/apt/sources.list.d/rocm.list

# Clear the cache and clean the system
sudo rm -rf /var/cache/apt/*
sudo apt clean all
sudo apt update
```

Important

To apply all settings, reboot your system.

17.4.2 Debian native installation

Caution

Ensure that the *Installation prerequisites* are met.

Note

The following installation steps also apply when upgrading from a previous ROCm version.

17.4.2.1 Registering ROCm repositories

17.4.2.1.1 Package signing key

Download and convert the package signing key.

```
# Make the directory if it doesn't exist yet.
# This location is recommended by the distribution maintainers.
sudo mkdir --parents --mode=0755 /etc/apt/keyrings
```

(continues on next page)

(continued from previous page)

```
# Download the key, convert the signing-key to a full
# keyring required by apt and store in the keyring directory
wget https://repo.radeon.com/rocm/rocm.gpg.key -O - | \
    gpg --dearmor | sudo tee /etc/apt/keyrings/rocm.gpg > /dev/null
```

Note

The GPG key may change; ensure it is updated when installing a new release. If the key signature verification fails while updating, re-add the key from the ROCm to the apt repository as mentioned above.

17.4.2.1.2 Register packages

Debian 13

```
# Register ROCm packages
sudo tee /etc/apt/sources.list.d/rocm.list << EOF
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/rocm/
↪apt/7.2.3 noble main
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/
↪graphics/7.2.3/ubuntu noble main
EOF

sudo tee /etc/apt/preferences.d/rocm-pin-600 << EOF
Package: *
Pin: release o=repo.radeon.com
Pin-Priority: 600
EOF

sudo apt update
```

Debian 12

```
# Register ROCm packages
sudo tee /etc/apt/sources.list.d/rocm.list << EOF
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/rocm/
↪apt/7.2.3 jammy main
deb [arch=amd64 signed-by=/etc/apt/keyrings/rocm.gpg] https://repo.radeon.com/
↪graphics/7.2.3/ubuntu jammy main
EOF

sudo tee /etc/apt/preferences.d/rocm-pin-600 << EOF
Package: *
Pin: release o=repo.radeon.com
Pin-Priority: 600
EOF

sudo apt update
```

17.4.2.2 Installing

17.4.2.2.1 Install kernel driver

For information about the AMDGPU driver installation, see the Debian native installation in the AMD Instinct Data Center GPU Documentation.

For information about driver compatibility, see User and AMD GPU Driver (amdgpu) support matrix.

17.4.2.2.2 Install ROCm

```
sudo apt install rocm
```

17.4.2.3 Post-installation

Complete the *Post-installation instructions*.

17.4.2.4 Uninstalling

17.4.2.4.1 Uninstall ROCm meta packages

```
sudo apt autoremove rocm
sudo apt autoremove rocm-core
```

17.4.2.4.2 Remove ROCm repositories

```
# Remove the repositories
sudo rm /etc/apt/sources.list.d/rocm.list

# Clear the cache and clean the system
sudo rm -rf /var/cache/apt/*
sudo apt clean all
sudo apt update
```

Important

To apply all settings, reboot your system.

17.4.3 Red Hat Enterprise Linux native installation

Caution

Ensure that the *Installation prerequisites* are met.

Note

The following installation steps also apply when upgrading from a previous ROCm version.

17.4.3.1 Registering ROCm repositories

RHEL 10.1

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el10/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/10/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

RHEL 10.0

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el10/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/10/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

RHEL 9.7

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el9/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
```

(continues on next page)

(continued from previous page)

```
[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/9.7/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

RHEL 9.6

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el9/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/9.6/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

RHEL 9.4

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el9/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/9.4/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
```

(continues on next page)

(continued from previous page)

```
sudo dnf clean all
```

RHEL 8.10

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el8/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/8/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

17.4.3.2 Installing

17.4.3.2.1 Install kernel driver

For information about the AMDGPU driver installation, see the Red Hat Enterprise Linux native installation in the AMD Instinct Data Center GPU Documentation.

For information about driver compatibility, see User and AMD GPU Driver (amdgpu) support matrix.

17.4.3.2.2 Install ROCm

```
sudo dnf install rocm
```

17.4.3.3 Post-installation

Complete the *Post-installation instructions*.

17.4.3.4 Uninstalling

17.4.3.4.1 Uninstall ROCm meta packages

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

17.4.3.4.2 Remove ROCm repositories

```
# Remove the repositories
sudo rm /etc/yum.repos.d/rocm.repo*

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

Important

To apply all settings, reboot your system.

17.4.4 Oracle Linux native installation

Caution

Ensure that the *Installation prerequisites* are met.

Note

The following installation steps also apply when upgrading from a previous ROCm version.

17.4.4.1 Register ROCm repositories

OL 10.1

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el10/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/10/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

OL 9.7

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el9/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/9.7/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

OL 8.10

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el8/7.2.3/main
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/8/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

17.4.4.2 Installing

17.4.4.2.1 Install kernel driver

For information about the AMDGPU driver installation, see the Oracle Linux native installation in the AMD Instinct Data Center GPU Documentation.

For information about driver compatibility, see User and AMD GPU Driver (amdgpu) support matrix.

17.4.4.2.2 Install ROCm

```
sudo dnf install rocm
```

17.4.4.3 Post-installation

Complete the *Post-installation instructions*.

17.4.4.4 Uninstalling

17.4.4.4.1 Uninstall ROCm meta packages

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

17.4.4.4.2 Remove ROCm repositories

```
# Remove the repositories
sudo rm /etc/yum.repos.d/rocm.repo*

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

Important

To apply all settings, reboot your system.

17.4.5 Rocky Linux native installation

Caution

Ensure that the *Installation prerequisites* are met.

Note

The following installation steps also apply when upgrading from a previous ROCm version.

17.4.5.1 Registering ROCm repositories

Rocky 9.7

```
sudo tee /etc/yum.repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/el9/7.2.3/main
enabled=1
priority=50
```

(continues on next page)

(continued from previous page)

```
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/el/9.7/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo dnf clean all
```

17.4.5.2 Installing

17.4.5.2.1 Install kernel driver

For information about the AMDGPU driver installation, see the Rocky Linux native installation in the AMD Instinct Data Center GPU Documentation.

For information about driver compatibility, see User and AMD GPU Driver (amdgpu) support matrix.

17.4.5.2.2 Install ROCm

```
sudo dnf install rocm
```

17.4.5.3 Post-installation

Complete the *Post-installation instructions*.

17.4.5.4 Uninstalling

17.4.5.4.1 Uninstall ROCm meta packages

```
sudo dnf remove rocm
sudo dnf remove rocm-core amdgpu-core
```

17.4.5.4.2 Remove ROCm repositories

```
# Remove the repositories
sudo rm /etc/yum.repos.d/rocm.repo*

# Clear the cache and clean the system
sudo rm -rf /var/cache/dnf
sudo dnf clean all
```

Important

To apply all settings, reboot your system.

17.4.6 SUSE Linux Enterprise Server native installation

Caution

Ensure that the *Installation prerequisites* are met.

Note

The following installation steps also apply when upgrading from a previous ROCm version.

17.4.6.1 Registering ROCm repositories

SLES 15.7

```
sudo tee /etc/zypp/repos.d/rocm.repo <<EOF
[rocm]
name=ROCm 7.2.3 repository
baseurl=https://repo.radeon.com/rocm/zyp/7.2.3/main
enabled=1
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key

[amdgraphics]
name=AMD Graphics 7.2.3 repository
baseurl=https://repo.radeon.com/graphics/7.2.3/sle/15.7/main/x86_64/
enabled=1
priority=50
gpgcheck=1
gpgkey=https://repo.radeon.com/rocm/rocm.gpg.key
EOF
sudo zypper refresh
```

17.4.6.2 Installing

17.4.6.2.1 Install kernel driver

For information about the AMDGPU driver installation, see the SUSE Linux Enterprise Server native installation in the AMD Instinct Data Center GPU Documentation.

For information about driver compatibility, see User and AMD GPU Driver (amdgpu) support matrix.

17.4.6.2.2 Install ROCm

```
sudo zypper --gpg-auto-import-keys install rocm
```

17.4.6.3 Post-installation

Complete the *Post-installation instructions*.

17.4.6.4 Uninstalling

17.4.6.4.1 Uninstall ROCm meta packages

```
sudo zypper remove rocm
sudo zypper remove rocm-core amdgpu-core
```

17.4.6.4.2 Remove ROCm repositories

```
# Remove ROCm repositories
sudo zypper removerepo "rocm"
sudo zypper removerepo "amdgraphics"

# Clear cache and clean system
sudo zypper clean --all
sudo zypper refresh
```

Important

To apply all settings, reboot your system.

17.5 Post-installation instructions

After installing ROCm, follow these steps to finalize and validate the installation.

17.5.1 Environment Configuration

17.5.1.1 1. Configure ROCm shared objects

Configure the system linker by specifying where to find the shared objects (`.so` files) for ROCm applications.

```
sudo tee --append /etc/ld.so.conf.d/rocm.conf <<EOF
/opt/rocm/lib
/opt/rocm/lib64
EOF
sudo ldconfig
```

17.5.1.2 2. Configure ROCm `PATH`

Configure the path to the ROCm binary using one of the following Linux utilities or manually update the `PATH` variable. The ROCm installation process adds the ROCm executables to these systems, provided they are installed on the system.

Option A: `update-alternatives`

The `update-alternatives` utility is available on most Linux distributions. It helps manage multiple versions of a command or program. For more information about `update-alternatives`, see Linux man.

To use `update-alternatives`, follow these steps:

1. Display a list of all ROCm versions available:

   ```
   sudo update-alternatives --display rocm
   ```

2. If multiple ROCm versions are installed, switch between them using this command and selecting the ROCm version:

```
sudo update-alternatives --config rocm
```

Option B: `environment-modules`

The `environment-modules` tool simplifies shell initialization. It lets you modify your session environment using module files. For more information, see Environment Modules.

> **Note**
>
> The `environment-modules` package should be installed on the system before ROCm can be configured using modules. For more information, see *prerequisites*.

To use `environment-modules`, follow these instructions:

1. Enable `environment-modules`:

   ```
   source /etc/profile.d/modules.sh
   ```

2. Display a list of all available modules (including ROCm modules):

   ```
   module avail
   ```

3. Display a list of all currently loaded modules:

   ```
   module list
   ```

4. If multiple ROCm versions are installed, and no other ROCm modules are currently loaded, set ROCm by version:

   ```
   module load rocm/7.2.3
   ```

5. If multiple ROCm versions are installed with a current ROCm module in use, switch to another ROCm version as follows:

   ```
   module switch rocm/7.2.3
   ```

> **Note**
>
> If modules are used for ROCm, any `update-alternatives` ROCm setting will be overwritten for the terminal session.

Option C: `PATH`

If `update-alternatives` or `environment-modules` are not available on the system, configure the ROCm path by setting the `PATH` variable to `/opt/rocm-<version>/bin`.

```
export PATH=$PATH:/opt/rocm-7.2.3/bin
```

17.5.1.3 3. Configure `LD_LIBRARY_PATH`

> **Important**
>
> This step is required for version-specific or multi-version installations.

```
export LD_LIBRARY_PATH=/opt/rocm-7.2.3/lib
```

17.5.2 Install verification

Once ROCm has been configured, validate the installation.

17.5.2.1 1. Verify the package installation

Use the package manager to validate the list of ROCm component packages installed on the system. If package installation was successful, the list will contain `rocm*` and `hip*` packages currently installed on the system.

Ubuntu

```
apt list --installed
```

Debian

```
apt list --installed
```

RHEL

```
dnf list installed
```

OL

```
dnf list installed
```

Rocky

```
dnf list installed
```

SLES

```
zypper search --installed-only
```

17.5.2.2 2. Verify the ROCm installation

Use the following ROCm tools to verify that installation was successful:

rocminfo

```
rocminfo | grep -i "Marketing Name:"
```

Example output:

```
Marketing Name:          AMD EPYC 9654 96-Core Processor
Marketing Name:          AMD EPYC 9654 96-Core Processor
Marketing Name:          AMD Instinct MI300X
```

clinfo

```
clinfo | grep -i "Board name:"
```

Example output:

```
Board name:                                     AMD Instinct MI300X
```

amd-smi

```
amd-smi version
```

Example output:

```
AMDSMI Tool: 26.2.2+c2d9476115 | AMDSMI Library version: 26.2.2 | ROCm version: 7.2.3␣
↪| amdgpu version: 6.16.13 | hsmp version: N/A
```

17.5.3 Troubleshooting

1. **What if environment-modules is not installed before installing ROCm?**

   You can still install `environment-modules` package after installing ROCm. However, no ROCm modules will be listed for the `module avail` command. As an alternative to the standard method of loading the ROCm modules, you can load each version-specific module directly from the `/opt/rocm-<version>` directory:

   ```
   module load /opt/rocm-7.2.3/lib/rocmmod
   ```

2. **Will the ROCm path configuration persist once I set it?**

   - If you are using update-alternatives to configure ROCm, then yes, the currently set configuration will persist even after a system reboot.
   - If you are using environment-modules to configure ROCm, then no, the current set configuration will only last for the current terminal session.

CHAPTER

EIGHTEEN

THIRD-PARTY TOOLS

Many third-party tools can support your HIP and ROCm development. This section provides a brief overview of some of these tools and how you can use them with AMD GPUs. For detailed, up-to-date information, refer to the official documentation of each tool, as features and compatibility change frequently.

18.1 Performance monitoring and profiling

You can use several tools to monitor performance counters and profile your GPU-accelerated applications.

18.1.1 PAPI

The Performance Application Programming Interface (PAPI) provides access to hardware performance counters on CPUs, GPUs, and other system components. PAPI 6.0.0 and later includes components for ROCm that enable monitoring of AMD GPU events through the ROCprofiler library. You can track events such as L1 and L2 cache activity, vector and scalar arithmetic logic unit operations, and memory transactions. The AMD SMI component adds power management support, enabling you to monitor and cap power usage on AMD GPUs.

For more information, see the official PAPI documentation.

18.1.2 HPCToolkit

HPCToolkit from Rice University measures and analyzes GPU-accelerated applications by recording call-path profiles and traces CPU and GPU activity. It helps you understand how your application uses GPU operations, and attributes the contributions of the calling context in which GPU operations are invoked. HPCToolkit supports multiple programming models, including HIP, and it uses hardware performance counters to measure GPU operations in detail.

For more information, see the HPCToolkit project website.

18.1.3 TAU Performance System

The TAU Performance System provides a parallel performance evaluation toolkit that supports profiling and tracing modes of measurement. TAU can profile HIP programs using the ROCprofiler and ROCTracer APIs to gather timestamp information of executing kernels on the GPU and data-transfer information. It supports various parallel programming models including, MPI, OpenMP, and Kokkos, and can generate traces in multiple formats for visualization.

For more information, see the TAU project website.

18.2 Tracing and visualization

Tools in this category help you trace application execution and visualize performance data to identify bottlenecks and optimize your code.

18.2.1 Score-P and trace visualization

Score-P is a highly scalable measurement infrastructure for profiling and event-tracing HPC applications. It uses the ROCTracer library to record HIP API functions, memory transfers between host and device, kernel launches, and other runtime behaviors. Vampir is a commercial tool that visualizes Score-P event logs as timelines and statistical charts, helping you detect performance problems that change over your application's runtime.

For an open-source approach to visualizing Score-P traces, you can use Scalasca and Cube. Scalasca is a performance analysis toolset for HPC applications, and Cube provides a graphical display for presenting performance metrics and timeline views from Score-P generated OTF2 trace files. These tools are freely available and integrate with Score-P for analyzing MPI, OpenMP, and GPU-accelerated applications.

For more information, see the Score-P documentation, the Vampir website, the Cube download page, and the Scalasca download page.

18.2.2 Trace Compass and Theia

Trace Compass provides visualization and analysis for several trace formats, including those generated by ROCm applications. It models the system's state over time, enabling analysis of process threads, GPU compute kernels, and system events. Theia is an open, extensible integrated development environment platform that integrates with Trace Compass via a trace extension, allowing you to analyze traces directly from within your development environment.

For more information, see the Trace Compass project and the Theia IDE project.

18.3 Debugging tools

Debuggers help you identify and fix errors in your HIP applications running on AMD GPUs.

18.3.1 TotalView

TotalView is a feature-rich debugger that supports HIP applications running on AMD GPUs. It enables you to debug heterogeneous applications that mix processor architectures, with separate address spaces for CPU processes and GPU agents. You can launch, attach to, and detach from processes, display GPU registers and disassembled machine instructions, create breakpoints, and trace code at both source and instruction levels. TotalView provides a unified view of source code and breakpoints across all image files, including dynamically loaded AMD GPU ELF images.

For more information, see the TotalView documentation.

18.3.2 Linaro Forge (DDT and MAP)

Linaro Forge combines Linaro DDT for parallel debugging and Linaro MAP for performance profiling. DDT is a graphical debugger suitable for heterogeneous software, including GPU programs, while MAP helps you identify the most time-consuming lines of code in your application. Both tools support AMD ROCm programs and can provide insights into GPU kernel execution, memory usage, and performance metrics.

For more information, see the Linaro Forge documentation.

18.4 Development environments

Container-based environments provide consistent development and deployment platforms for ROCm applications.

18.4.1 E4S (Extreme Scale Scientific Software Stack)

E4S is a curated collection of software products based on the Spack package manager, available as container images supporting Docker and Singularity runtimes. The E4S images include ROCm with compilers, and Python-based artificial intelligence and machine learning tools such as PyTorch for ROCm and TensorFlow for ROCm. These containers enable you to leverage ready-to-use environments for HPC and AI or ML development with AMD GPUs, providing consistent environments from workstations to large-scale datacenter deployments.

For more information, see the E4S project website.

BIBLIOGRAPHY

[zc] Zero copy is a feature, where the memory is pinned to either the device or the host, and won't be transferred when accessed by another device or the host. Instead only the requested memory is transferred, without making an explicit copy, like a normal memory access, hence the term "zero copy".