CUDA Programming Guide

Release 13.1

NVIDIA Corporation

Dec 12, 2025

Contents

CUDA and the CUDA Programming Guide

CUDA is a parallel computing platform and programming model developed by NVIDIA that enables dramatic increases in computing performance by harnessing the power of the GPU. It allows developers to accelerate compute-intensive applications and is widely used in fields such as deep learning, scientific computing, and high-performance computing (HPC).

This CUDA Programming Guide is the official, comprehensive resource on the CUDA programming model and how to write code that executes on the GPU using the CUDA platform. This guide covers everything from the the CUDA programming model and the CUDA platform to the details of language extensions and covers how to make use of specific hardware and software features. This guide provides a pathway for developers to learn CUDA if they are new, and also provides an essential resource for developers as they build applications using CUDA.

Organization of This Guide

Even for developers who primarily use libraries, frameworks, or DSLs, an understanding of the CUDA programming model and how GPUs execute code is valuable in knowing what is happening behind the layers of abstraction. This guide starts with a chapter on the *CUDA programming model* outside of any specific programming language which is applicable to anyone interested in understanding how CUDA works, even non-developers.

The guide is broken down into five primary parts:

- Part 1: Introduction and Programming Model Abstract
  - A language agnostic overview of the CUDA programming model as well as a brief tour of the CUDA platform.
  - This section is meant to be read by anyone wanting to understand GPUs and the concepts of executing code on GPUs, even if they are not developers.
- Part 2: Programming GPUs in CUDA
  - The basics of programming GPUs using CUDA C++.
  - This section is meant to be read by anyone wanting to get started in GPU programming.
  - This section is meant to be instructional, not complete, and teaches the most important and common parts of CUDA programming, including some common performance considerations.
- Part 3: Advanced CUDA
  - Introduces some more advance features of CUDA that enable both fine-grained control and more opportunities to maximize performance, including the use of multiple GPUs in a single application.
  - This section concludes with a *tour of the features covered in part 4* with a brief introduction to the purpose and function of each, sorted by when and why a developer may find each feature useful.
- Part 4: CUDA Features
  - This section contains complete coverage of specific CUDA features such as CUDA graphs, dynamic parallelism, interoperability with graphics APIs, and unified memory.
  - This section should be consulted when knowing the complete picture of a specific CUDA feature is needed. Where possible, care has been taken to introduce and motivate the features covered in this section in earlier sections.
- Part 5: Technical Appendices

- The technical appendices provide some reference documentation on CUDA's C++ high-level language support, hardware-specific specifications, and other technical specifications.
- This section is meant as technical reference for specific description of syntax, semantics, and technical behavior of elements of CUDA.

Parts 1-3 provide a guided learning experience for developers new to CUDA, though they also provide insight and updated information useful for CUDA developers of any experience level.

Parts 4 and 5 provide a wealth of information about specific features and detailed topics, and are intended to provide a curated, well-organized reference for developers needing to know more details as they write CUDA applications.

Chapter 1. Introduction to CUDA

1.1. Introduction

1.1.1. The Graphics Processing Unit

Born as a special-purpose processor for 3D graphics, the *Graphics Processing Unit* (GPU) started out as fixed-function hardware to accelerate parallel operations in real-time 3D rendering. Over successive generations, GPUs became more programmable. By 2003, some stages of the graphics pipeline became fully programmable, running custom code in parallel for each component of a 3D scene or an image.

In 2006, NVIDIA introduced the *Compute Unified Device Architecture* (CUDA) to enable any computational workload to use the throughput capability of GPUs independent of graphics APIs.

Since then, CUDA and GPU computing have been used to accelerate computational workloads of nearly every type, from scientific simulations such as fluid dynamics or energy transport to business applications like databases and analytics. Moreover, the capability and programmability of GPUs has been foundational to the advancement of new algorithms and technologies ranging from image classification to generative artificial intelligence such as diffusion or large language models.

1.1.2. The Benefits of Using GPUs

A GPU provides much higher instruction throughput and memory bandwidth than a CPU within a similar price and power envelope. Many applications leverage these capabilities to run significantly faster on the GPU than on the CPU (see GPU Applications). Other computing devices, like FPGAs, are also very energy efficient, but offer much less programming flexibility than GPUs.

GPUs and CPUs are designed with different goals in mind. While a CPU is designed to excel at executing a serial sequence of operations (called a thread) as fast as possible and can execute a few tens of these threads in parallel, a GPU is designed to excel at executing thousands of threads in parallel, trading off lower single-thread performance to achieve much greater total throughput.

GPUs are specialized for highly parallel computations and devote more transistors to data processing units, while CPUs dedicate more transistors to data caching and flow control. Figure 1 shows an example distribution of chip resources for a CPU versus a GPU.

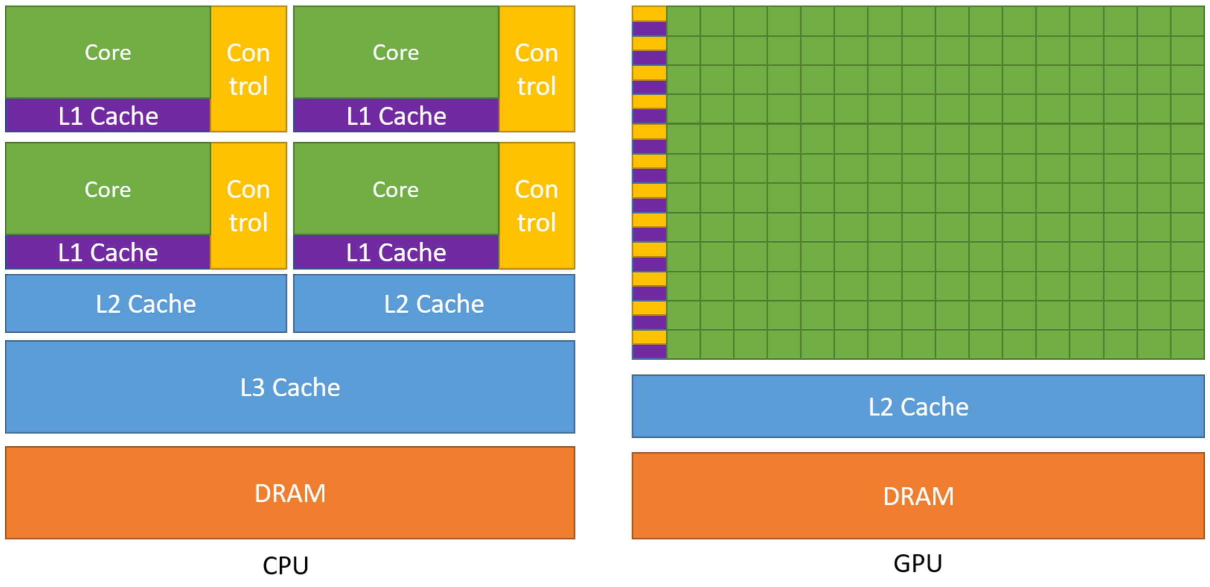


Figure 1: The GPU Devotes More Transistors to Data Processing

1.1.3. Getting Started Quickly

There are many ways to leverage the compute power provided by GPUs. This guide covers programming for the CUDA GPU platform in high-level languages such as C++. However, there are many ways to utilize GPUs in applications that do not require directly writing GPU code.

An ever-growing collection of algorithms and routines from a variety of domains is available through specialized libraries. When a library has already been implemented—especially those provided by NVIDIA—using it is often more productive and performant than reimplementing algorithms from scratch. Libraries like cuBLAS, cuFFT, cuDNN, and CUTLASS are just a few examples of libraries that help developers avoid reimplementing well-established algorithms. These libraries have the added benefit of being optimized for each GPU architecture, providing an ideal mix of productivity, performance, and portability.

There are also frameworks, particularly those used for artificial intelligence, that provide GPU-accelerated building blocks. Many of these frameworks achieve their acceleration by leveraging the GPU-accelerated libraries mentioned above.

Additionally, domain-specific languages (DSLs) such as NVIDIA's Warp or OpenAI's Triton compile to run directly on the CUDA platform. This provides an even higher-level method of programming GPUs than the high-level languages covered in this guide.

The NVIDIA Accelerated Computing Hub contains resources, examples, and tutorials to teach GPU and CUDA computing.

1.2. Programming Model

This chapter introduces the CUDA programming model at a high level and separate from any language. The terminology and concepts introduced here apply to CUDA in any supported programming language. Later chapters will illustrate these concepts in C++.

1.2.1. Heterogeneous Systems

The CUDA programming model assumes a heterogeneous computing system, which means a system that includes both GPUs and CPUs. The CPU and the memory directly connected to it are called the *host* and *host memory*, respectively. A GPU and the memory directly connected to it are referred to as the *device* and *device memory*, respectively. In some system-on-chip (SoC) systems, these may be part of a single package. In larger systems, there may be multiple CPUs or GPUs.

CUDA applications execute some part of their code on the GPU, but applications always start execution on the CPU. The host code, which is the code that runs on the CPU, can use CUDA APIs to copy data between the host memory and device memory, start code executing on the GPU, and wait for data copies or GPU code to complete. The CPU and GPU can both be executing code simultaneously, and best performance is usually found by maximizing utilization of both CPUs and GPUs.

The code an application executes on the GPU is referred to as *device code*, and a function that is invoked for execution on the GPU is, for historical reasons, called a *kernel*. The act of starting a kernel running is called *launching* the kernel. A kernel launch can be thought of as starting many threads executing the kernel code in parallel on the GPU. GPU threads operate similarly to threads on CPUs, though there are some differences important to both correctness and performance that will be covered in later sections (see Section 3.2.2.1.1).

1.2.2. GPU Hardware Model

Like any programming model, CUDA relies on a conceptual model of the underlying hardware. For the purposes of CUDA programming, the GPU can be considered to be a collection of *Streaming Multiprocessors* (SMs) which are organized into groups called *Graphics Processing Clusters* (GPCs). Each SM contains a local register file, a unified data cache, and a number of functional units that perform computations. The unified data cache provides the physical resources for *shared memory* and L1 cache. The allocation of the unified data cache to L1 and shared memory can be configured at runtime. The sizes of different types of memory and the number of functional units within an SM can vary across GPU architectures.

Note

The actual hardware layout of a GPU or the way it physically carries out the execution of the programming model may vary. These differences do not affect correctness of software written using the CUDA programming model.

1.2.2.1 Thread Blocks and Grids

When an application launches a kernel, it does so with many threads, often millions of threads. These threads are organized into blocks. A block of threads is referred to, perhaps unsurprisingly, as a *thread block*. Thread blocks are organized into a *grid*. All the thread blocks in a grid have the same size and dimensions. Figure 3 shows an illustration of a grid of thread blocks.

Thread blocks and grids may be 1, 2, or 3 dimensional. These dimensions can simplify mapping of individual threads to units of work or data items.

When a kernel is launched, it is launched using a specific *execution configuration* which specifies the grid and thread block dimensions. The execution configuration may also include optional parameters such as cluster size, stream, and SM configuration settings, which will be introduced in later sections.

Using built-in variables, each thread executing the kernel can determine its location within its containing block and the location of its block within the containing grid. A thread can also use these

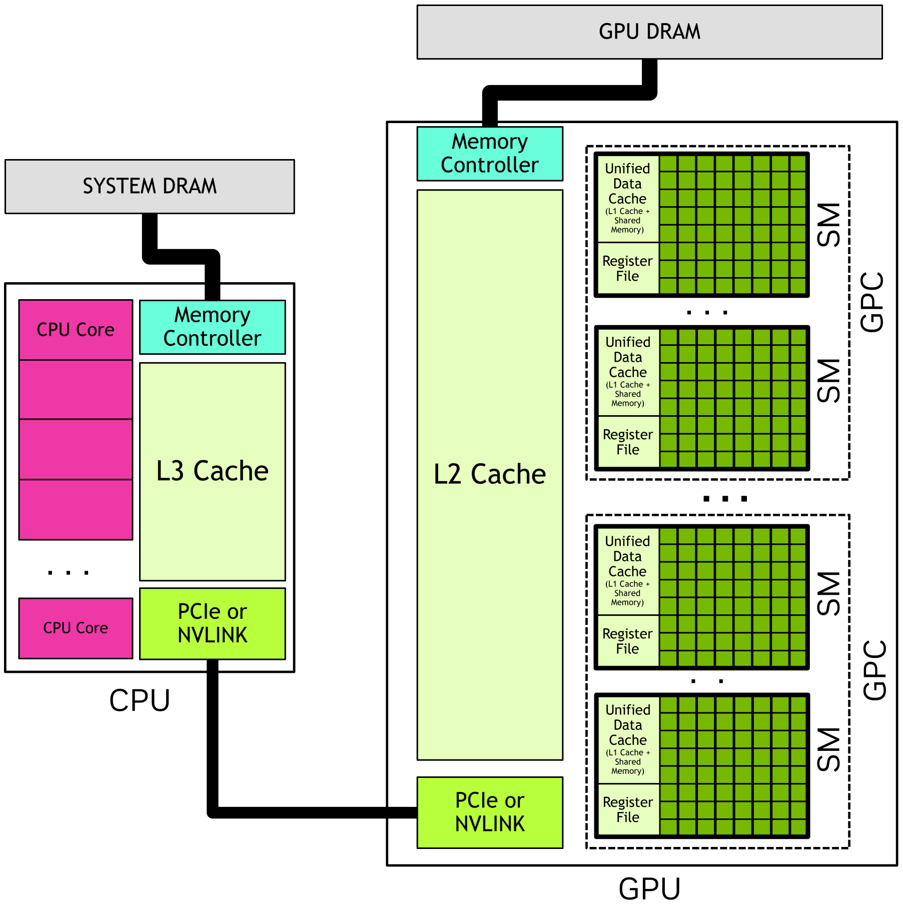


Figure 2: A GPU has many streaming multiprocessors (SMs), each of which contains many functional units. Graphics processing clusters (GPCs) are collections of SMs. A GPU is a set of GPCs connected to the GPU memory. A CPU typically has several cores and a memory controller which connects to the system memory. A CPU and a GPU are connected by an interconnect such as PCIe or NVLINK.

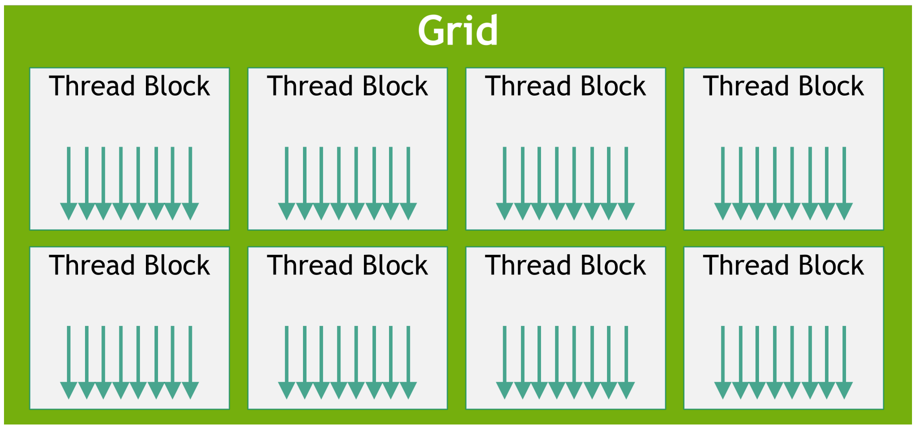


Figure 3: Grid of Thread Blocks. Each arrow represents a thread (the number of arrows is not representative of actual number of threads).

built-in variables to determine the dimensions of the thread blocks and the grid on which the kernel was launched. This gives each thread a unique identity among all the threads running the kernel. This identity is frequently used to determine what data or operations a thread is responsible for.

All threads of a thread block are executed in a single SM. This allows threads within a thread block to communicate and synchronize with each other efficiently. Threads within a thread block all have access to the on-chip shared memory, which can be used for exchanging information between threads of a thread block.

A grid may consist of millions of thread blocks, while the GPU executing the grid may have only tens or hundreds of SMs. All threads of a thread block are executed by a single SM and, in most cases[1], run to completion on that SM. There is no guarantee of scheduling between thread blocks, so a thread block cannot rely on results from other thread blocks, as they may not be able to be scheduled until that thread block has completed. Figure 4 shows an example of how thread blocks from a grid are assigned to an SM.

The CUDA programming model enables arbitrarily large grids to run on GPUs of any size, whether it has only one SM or thousands of SMs. To achieve this, the CUDA programming model, with some exceptions, requires that there be no data dependencies between threads in different thread blocks. That is, a thread should not depend on results from or synchronize with a thread in a different thread block of the same grid. All the threads within a thread block run on the same SM at the same time. Different thread blocks within the grid are scheduled among the available SMs and may be executed in any order. In short, the CUDA programming model requires that it be possible to execute thread blocks in any order, in parallel or in series.

1.2.2.1.1 Thread Block Clusters

In addition to thread blocks, GPUs with compute capability 9.0 and higher have an optional level of grouping called *clusters*. Clusters are a group of thread blocks which, like thread blocks and grids, can be laid out in 1, 2, or 3 dimensions. Figure 5 illustrates a grid of thread blocks that is also organized into clusters. Specifying clusters does not change the grid dimensions or the indices of a thread block within a grid.

Specifying clusters groups adjacent thread blocks into clusters and provides some additional opportunities for synchronization and communication at the cluster level. Specifically, all thread blocks in a cluster are executed in a single GPC. Figure 6 shows how thread blocks are scheduled to SMs in a GPC when clusters are specified. Because the thread blocks are scheduled simultaneously and within a single GPC, threads in different blocks but within the same cluster can communicate and synchronize with each other using software interfaces provided by *Cooperative Groups*. Threads in clusters can access the shared memory of all blocks in the cluster, which is referred to as *distributed shared memory*.The maximum size of a cluster is hardware dependent and varies between devices.

Figure 6 illustrates the how thread blocks within a cluster are scheduled simultaneously on SMs within a GPC. Thread blocks within a cluster are always adjacent to each other within the grid.

1.2.2.2 Warps and SIMT

Within a thread block, threads are organized into groups of 32 threads called *warps*. A warp executes the kernel code in a *Single-Instruction Multiple-Threads* (SIMT) paradigm. In SIMT, all threads in the warp are executing the same kernel code, but each thread may follow different branches through the code. That is, though all threads of the program execute the same code, threads do not need to follow the same execution path.

[1] In certain situations when using features such as *CUDA Dynamic Parallelism*, a thread block may be suspended to memory. This means the state of the SM is stored to a system-managed area of GPU memory and the SM is freed to execute other thread blocks. This is similar to context swapping on CPUs. This is not common.

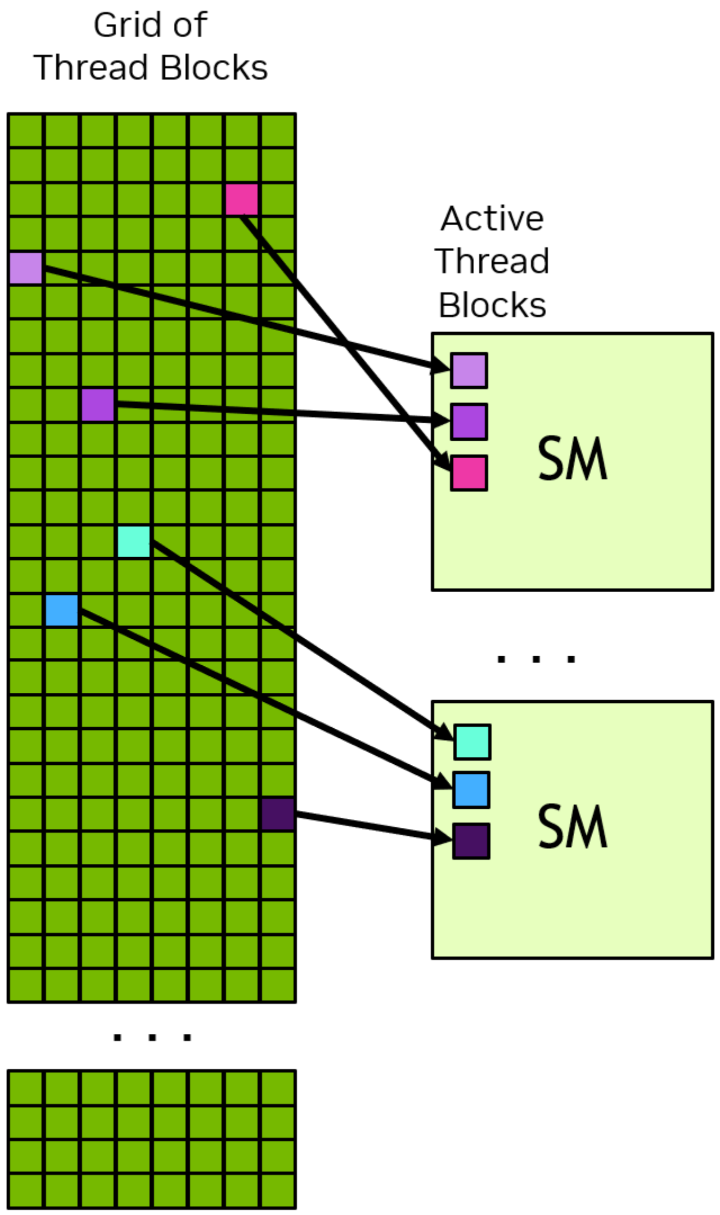


Figure 4: Each SM has one or more active thread blocks. In this example, each SM has three thread blocks scheduled simultaneously. There are no guarantees about the order in which thread blocks from a grid are assigned to SMs.

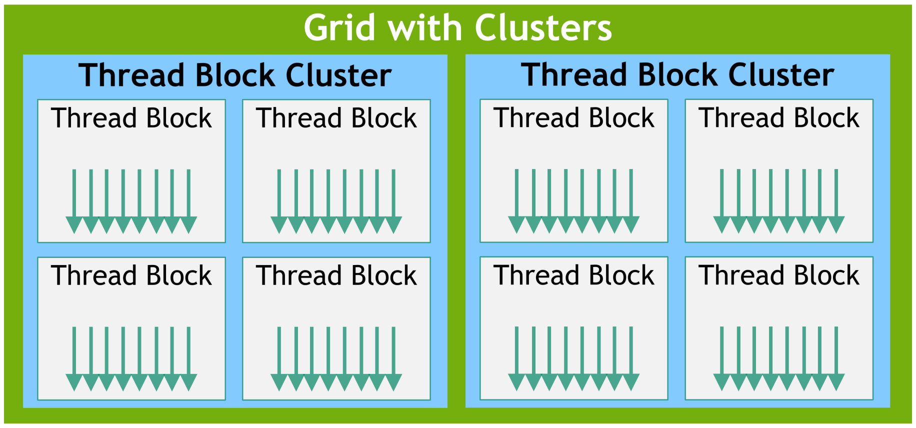


Figure 5: When clusters are specified, thread blocks are in the same location in the grid but also have a position within the containing cluster.

When threads are executed by a warp, they are assigned a warp lane. Warp lanes are numbered 0 to 31 and threads from a thread block are assigned to warps in a predictable fashion detailed in *Hardware Multithreading*.

All threads in the warp execute the same instruction simultaneously. If some threads within a warp follow a control flow branch in execution while others do not, the threads which do not follow the branch will be masked off while the threads which follow the branch are executed. For example, if a conditional is only true for half the threads in a warp, the other half of the warp would be masked off while the active threads execute those instructions. This situation is illustrated in Figure 7. When different threads in a warp follow different code paths, this is sometimes called warp divergence. It follows that utilization of the GPU is maximized when threads within a warp follow the same control flow path.

In the SIMT model, all threads in a warp progress through the kernel in lock step. Hardware execution may differ. See the sections on *Independent Thread Execution* for more information on where this distinction is important. Exploiting knowledge of how warp execution is actually mapped to real hardware is discouraged. The CUDA programming model and SIMT say that all threads in a warp progress through the code together. Hardware may optimize masked lanes in ways that are transparent to the program so long as the programming model is followed. If the program violates this model, this can result in undefined behavior that can be different in different GPU hardware.

While it is not necessary to consider warps when writing CUDA code, understanding the warp execution model is helpful in understanding concepts such as *global memory coalescing* and *shared memory bank access patterns*. Some advanced programming techniques use specialization of warps within a thread block to limit thread divergence and maximize utilization. This and other optimizations make use of the knowledge that threads are grouped into warps when executing.

One implication of warp execution is that thread blocks are best specified to have a total number of threads which is a multiple of 32. It is legal to use any number of threads, but when the total is not a multiple of 32, the last warp of the thread block will have some lanes that are unused throughout execution. This will likely lead to suboptimal functional units utilization and memory access for that warp.

SIMT is often compared to Single Instruction Multiple Data (SIMD) parallelism, but there are some important differences. In SIMD, execution follows a single control flow path, while in SIMT, each thread is allowed to follow its own control flow path. Because of this, SIMT does not have a fixed data-width like SIMD. A more detailed discussion of SIMT can be found in *SIMT Execution Model*.

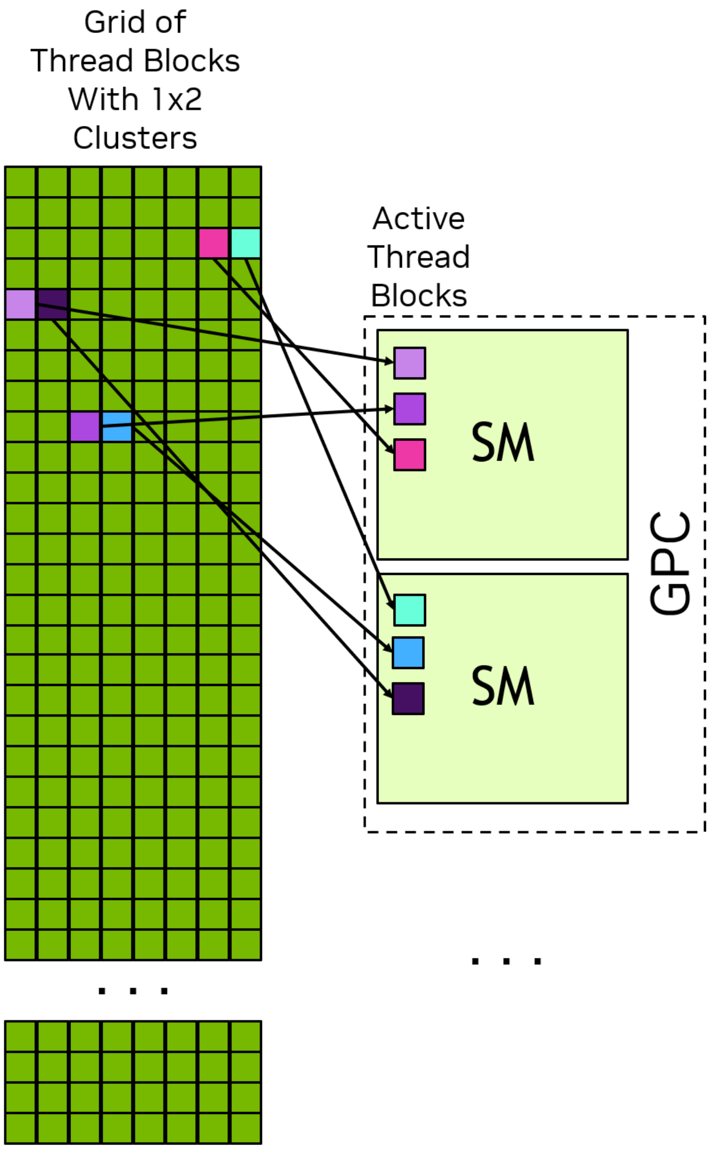


Figure 6: When clusters are specified, the thread blocks in a cluster are arranged in their cluster shape within the grid. The thread blocks of a cluster are scheduled simultaneously on the SMs of a single GPC.

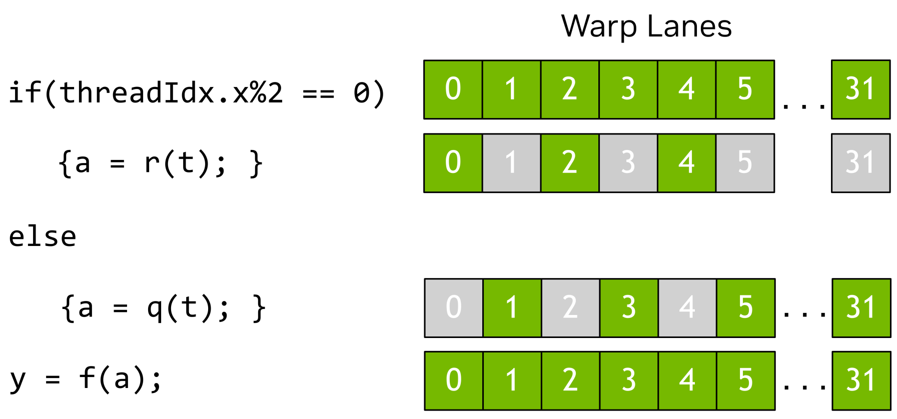


Figure 7: In this example, only threads with even thread index execute the body of the if statement, the others are masked off while the body is executed.

1.2.3. GPU Memory

In modern computing systems, efficiently utilizing memory is just as important as maximizing the use of functional units performing computations. Heterogeneous systems have multiple memory spaces, and GPUs contain various types of programmable on-chip memory in addition to caches. The following sections introduce these memory spaces in more details.

1.2.3.1 DRAM Memory in Heterogeneous Systems

GPUs and CPUs both have directly attached DRAM chips. In systems with more than one GPU, each GPU has its own memory. From the perspective of device code, the DRAM attached to the GPU is called *global memory*, because it is accessible to all SMs in the GPU. This terminology does not mean it is necessarily accessible everywhere within the system. The DRAM attached to the CPU(s) is called *system memory* or *host memory*.

Like CPUs, GPUs use virtual memory addressing. On all currently-supported systems, the CPU and GPU use a single unified virtual memory space. This means that the virtual memory address range for each GPU in the system is unique and distinct from the CPU and every other GPU in the system. For a given virtual memory address, it is possible to determine whether that address is in GPU memory or system memory and, on systems with multiple GPUs, which GPU memory contains that address.

There are CUDA APIs to allocate GPU memory, CPU memory, and to copy between allocations on the CPU and GPU, within a GPU, or between GPUs in multi-GPU systems. The locality of data can be explicitly controlled when desired. *Unified Memory*, discussed below, allows the placement of memory to be handled automatically by the CUDA runtime or system hardware.

1.2.3.2 On-Chip Memory in GPUs

In addition to the global memory, each GPU has some on-chip memory. Each SM has its own register file and shared memory. These memories are part of the SM and can be accessed extremely quickly from threads executing within the SM, but they are not accessible to threads running in other SMs.

The register file stores thread local variables which are usually allocated by the compiler. The shared memory is accessible by all threads within a thread block or cluster. Shared memory can be used for exchanging data between threads of a thread block or cluster.

The register file and unified data cache in an SM have finite sizes. The size of an SM's register file, unified data cache, and how the unified data cache can be configured for L1 and shared memory balance can be found in *Memory Information per Compute Capability*. The register file, shared memory space, and L1 cache are shared among all threads in a thread block.

To schedule a thread block to an SM, the total number of registers needed for each thread multiplied by the number of threads in the thread block must be less than or equal to the available registers in the SM. If the number of registers required for a thread block exceeds the size of the register file, the kernel is not launchable and the number of threads in the thread block must be decreased to make the thread block launchable.

Shared memory allocations are done at the thread block level. That is, unlike register allocations which are per thread, allocations of shared memory are common to the entire thread block.

1.2.3.2.1 Caches

In addition to programmable memories, GPUs have both L1 and L2 caches. Each SM has an L1 cache which is part of the unified data cache. A larger L2 cache is shared by all SMs within a GPU. This can be seen in the GPU block diagram in Figure 2. Each SM also has a separate *constant cache*, which is used to cache values in global memory that have been declared to be constant over the life of a kernel. The compiler may place kernel parameters into constant memory as well. This can improve kernel performance by allowing kernel parameters to be cached in the SM separately from the L1 data cache.

1.2.3.3 Unified Memory

When an application allocates memory explicitly on the GPU or CPU, that memory is only accessible to code running on that device. That is, CPU memory can only be accessed from CPU code, and GPU memory can only be accessed from kernels running on the GPU[2] . CUDA APIs for copying memory between the CPU and GPU are used to explicitly copy data to the correct memory at the right time.

A CUDA feature called *unified memory* allows applications to make memory allocations which can be accessed from CPU or GPU. The CUDA runtime or underlying hardware enables access or relocates the data to the correct place when needed. Even with unified memory, optimal performance is attained by keeping the migration of memory to a minimum and accessing data from the processor directly attached to the memory where it resides as much as possible.

The hardware features of the system determine how access and exchange of data between memory spaces is achieved. Section *Unified Memory* introduces the different categories of unified memory systems. Section *Unified Memory* contains many more details about use and behavior of unified memory in all situations.

1.3. The CUDA platform

The NVIDIA CUDA platform consists of many pieces of software and hardware and many important technologies developed to enable computing on heterogeneous systems. This chapter serves to introduce some of the fundamental concepts and components of the CUDA platform that are important for application developers to understand. This chapter, like *Programming Model*, is not specific to any programming language, but applies to everything that uses the CUDA platform.

[2] An exception to this is *mapped memory*, which is CPU memory allocated with properties that enable it to be directly accessed from the GPU. However, mapped access occurs over the PCIe or NVLINK connection. The GPU is unable to hide the higher latency and lower bandwidth behind parallelism, so mapped memory is not a performant replacement to unified memory or placing data in the appropriate memory space.

1.3.1. Compute Capability and Streaming Multiprocessor Versions

Every NVIDIA GPU has a *Compute Capability* (CC) number, which indicates what features are supported by that GPU and specifies some hardware parameters for that GPU. These specifications are documented in the Section 5.1 appendix. A list of all NVIDIA GPUs and their compute capabilities is maintained on the CUDA GPU Compute Capability page.

Compute capability is denoted as a major and minor version number in the format X.Y where X is the major version number and Y is the minor version number. For example, CC 12.0 has a major version of 12 and a minor version of 0. The compute capability directly corresponds to the version number of the SM. For example, the SMs within a GPU of CC 12.0 have SM version *sm_120*. This version is used to label binaries.

Section 5.1.1 shows how to query and determine the compute capability of the GPU(s) in a system.

1.3.2. CUDA Toolkit and NVIDIA Driver

The *NVIDIA Driver* can be thought of as the operating system of the GPU. The NVIDIA Driver is a software component which must be installed on the host system's operating system and is necessary for all GPU uses, including display and graphical functionality. The NVIDIA Driver is foundational to the CUDA platform. In addition to CUDA, the NVIDIA Driver provides all other methods of using the GPU, for example Vulkan and Direct3D. The NVIDIA Driver has version numbers such as r580.

The *CUDA Toolkit* is a set of libraries, headers, and tools for writing, building, and analyzing software which utilizes GPU computing. The CUDA Toolkit is a separate software product from the NVIDIA driver

The *CUDA runtime* is a special case of one of the libraries provided by the CUDA Toolkit. The CUDA runtime provides both an API and some language extensions to handle common tasks such as allocating memory, copying data between GPUs and other GPUs or CPUs, and launching kernels. The API components of the CUDA runtime are referred to as the CUDA runtime API.

The CUDA Compatibility document provides full details of compatibility between different GPUs, NVIDIA Drivers, and CUDA Toolkit versions.

1.3.2.1 CUDA Runtime API and CUDA Driver API

The CUDA runtime API is implemented on top of a lower-level API called the *CUDA driver API*, which is an API exposed by the NVIDIA Driver. This guide focuses on the APIs exposed by the CUDA runtime API. All the same functionality can be achieved using only the driver API if desired. Some features are only available using the driver API. Applications may use either API or both interoperably. Section *The CUDA Driver API* covers interoperation between the runtime and driver APIs.

The full API reference for the CUDA runtime API functions can be found in the CUDA Runtime API Documentation .

The full API reference for the CUDA driver API can be found in the CUDA Driver API Documentation .

1.3.3. Parallel Thread Execution (PTX)

A fundamental but sometimes invisible layer of the CUDA platform is the *Parallel Thread Execution* (PTX) virtual instruction set architecture (ISA). PTX is a high-level assembly language for NVIDIA GPUs. PTX provides an abstraction layer over the physical ISA of real GPU hardware. Like other platforms, applications can be written directly in this assembly language, though doing so can add unnecessary complexity and difficulty to software development.

Domain-specific languages and compilers for high-level languages can generate PTX code as an intermediate representation (IR) and then use NVIDIA's offline or just-in-time (JIT) compilation tools to produce executable binary GPU code. This enables the CUDA platform to be programmable from languages other than just those supported by NVIDIA-provided tools such as *NVCC: The NVIDIA CUDA Compiler*.

Since GPU capabilities change and grow over time, the PTX virtual ISA specification is versioned. PTX versions, like SM versions, correspond to a compute capability. For example, PTX which supports all the features of compute capability 8.0 is called *compute_80*.

Full documentation on PTX can be found in the PTX ISA .

1.3.4. Cubins and Fatbins

CUDA applications and libraries are usually written in a higher-level language like C++. That higher-level language is compiled to PTX, and then the PTX is compiled into real binary for a physical GPU, called a *CUDA binary*, or *cubin* for short. A cubin has a specific binary format for a specific SM version, such as *sm_120*.

Executables and library binaries that use GPU computing contain both CPU and GPU code. The GPU code is stored within a container called a *fatbin*. Fatbins can contain cubins and PTX for multiple different targets. For example, an application could be built with binaries for multiple different GPU architectures, that is, different SM versions. When an application is run, its GPU code is loaded onto a specific GPU and the best binary for that GPU from the fatbin is used.

Fatbins can also contain one or more PTX versions of GPU code, the use for which is described in *PTX Compatibility*. Figure 8 shows an example of an application or library binary which contains multiple cubin versions of GPU code as well as one version of PTX code.

1.3.4.1 Binary Compatibility

NVIDIA GPUs guarantee binary compatibility in certain circumstances. Specifically, within a major version of compute capability, GPUs with minor compute capability greater than or equal to the targeted version of cubin can load and execute that cubin. For example, if an application contains a cubin with code compiled for compute capability 8.6, that cubin can be loaded and executed on GPUs with compute capability 8.6 or 8.9. It cannot, however, be loaded on GPUs with compute capability 8.0, because the GPU's CC minor version, 0, is lower than the code's minor version, 6.

NVIDIA GPUs are not binary compatible between major compute capability versions. That is, cubin code compiled for compute capability 8.6 will not load on GPUs of compute capability 9.0.

When discussing binary code, the binary code is often referred to as having a version such as *sm_86* in the above example. This is the same as saying the binary was built for compute capability 8.6. This shorthand is often used because it is how a developer specifies this binary build target to the NVIDIA CUDA compiler, *nvcc*.

Note

Binary compatibility is promised only for binaries created by NVIDIA tools such as nvcc. Manual editing or generating binary code for NVIDIA GPUs is not supported. Compatibility promises are invalidated if binaries are modified in any way.

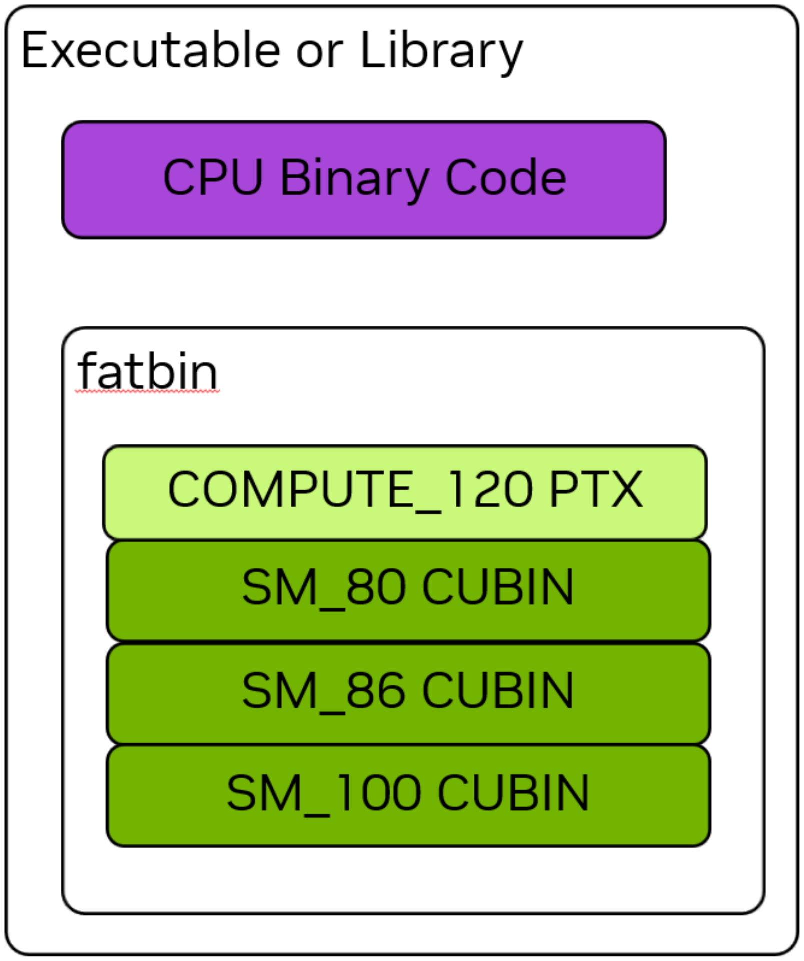


Figure 8: The binary for an executable or library contains both CPU binary code and a fatbin container for GPU code. A fatbin can contain both cubin GPU binary code and PTX virtual ISA code. PTX code can be JIT compiled for future targets.

1.3.4.2 PTX Compatibility

GPU code can be stored in executables in binary or PTX form, which is covered in *Cubins and Fatbins*. When an application stores the PTX version of GPU code, that PTX can be JIT compiled at application runtime for any compute capability equal or higher to the compute capability of the PTX code. For example, if an application contains PTX for *compute_80*, that PTX code can be JIT compiled to later SM versions, such as *sm_120* at application runtime. This enables forward compatibility with future GPUs without the need to rebuild applications or libraries.

1.3.4.3 Just-in-Time Compilation

PTX code loaded by an application at runtime is compiled to binary code by the device driver. This is called just-in-time (JIT) compilation. Just-in-time compilation increases application load time, but allows the application to benefit from any new compiler improvements coming with each new device driver. It also enables applications to run on devices that did not exist at the time the application was compiled.

When the device driver just-in-time compiles PTX code for an application, it automatically caches a copy of the generated binary code in order to avoid repeating the compilation in subsequent invocations of the application. The cache - called the compute cache - is automatically invalidated when the device driver is upgraded, so that applications can benefit from the improvements in the new just-in-time compiler built into the device driver.

How and when PTX is JIT compiled at runtime has been relaxed since the earliest versions of CUDA, allowing more flexibility for when and if to JIT compile some or all kernels. The section *Lazy Loading* describes the available options and how to control JIT behavior. There are also a few environment variables which control just-in-time compilation behavior, as described in *CUDA Environment Variables*.

As an alternative to using nvcc to compile CUDA C++ device code, NVRTC can be used to compile CUDA C++ device code to PTX at runtime. NVRTC is a runtime compilation library for CUDA C++; more information can be found in the NVRTC User guide.

Chapter 2. Programming GPUs in CUDA

2.1. Intro to CUDA C++

This chapter introduces some of the basic concepts of the CUDA programming model by illustrating how they are exposed in C++.

This programming guide focuses on the CUDA runtime API. The CUDA runtime API is the most commonly used way of using CUDA in C++ and is built on top of the lower level CUDA driver API.

CUDA Runtime API and CUDA Driver API discusses the difference between the APIs and *CUDA driver API* discusses writing code that mixes the APIs.

This guide assumes the CUDA Toolkit and NVIDIA Driver are installed and that a supported NVIDIA GPU is present. See The CUDA Quickstart Guide for instructions on installing the necessary CUDA components.

2.1.1. Compilation with NVCC

GPU code written in C++ is compiled using the NVIDIA Cuda Compiler, `nvcc`. `nvcc` is a compiler driver that simplifies the process of compiling C++ or PTX code: It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages.

This guide will show `nvcc` command lines which can be used on any Linux system with the CUDA Toolkit installed, at a Windows command line or power shell, or on Windows Subsystem for Linux with the CUDA Toolkit. The *nvcc chapter* of this guide covers common use cases of `nvcc`, and complete documentation is provided by the nvcc user manual.

2.1.2. Kernels

As mentioned in the introduction to the *CUDA Programming Model*, functions which execute on the GPU which can be invoked from the host are called kernels. Kernels are written to be run by many parallel threads simultaneously.

2.1.2.1 Specifying Kernels

The code for a kernel is specified using the `__global__` declaration specifier. This indicates to the compiler that this function will be compiled for the GPU in a way that allows it to be invoked from a kernel launch. A kernel launch is an operation which starts a kernel running, usually from the CPU. Kernels are functions with a `void` return type.

```
// Kernel definition
__global__ void vecAdd(float* A, float* B, float* C)
{

}
```

2.1.2.2 Launching Kernels

The number of threads that will execute the kernel in parallel is specified as part of the kernel launch. This is called the execution configuration. Different invocations of the same kernel may use different execution configurations, such as a different number of threads or thread blocks.

There are two ways of launching kernels from CPU code, *triple chevron notation* and `cudaLaunchKernelEx`. Triple chevron notation, the most common way of launching kernels, is introduced here. An example of launching a kernel using `cudaLaunchKernelEx` is shown and discussed in detail in in section Section 3.1.1.

2.1.2.2.1 Triple Chevron Notation

Triple chevron notation is a *CUDA C++ Language Extension* which is used to launch kernels. It is called triple chevron because it uses three chevron characters to encapsulate the execution configuration for the kernel launch, i.e. `<<< >>>`. Execution configuration parameters are specified as a comma separated list inside the chevrons, similar to parameters to a function call. The syntax for a kernel launch of the `vecAdd` kernel is shown below.

```
 __global__ void vecAdd(float* A, float* B, float* C)
 {

 }

int main()
{
    ...
    // Kernel invocation
    vecAdd<<<1, 256>>>(A, B, C);
    ...
}
```

The first two parameters to the triple chevron notation are the grid dimensions and the thread block dimensions, respectively. When using 1-dimensional thread blocks or grids, integers can be used to specify dimensions.

The above code launches a single thread block containing 256 threads. Each thread will execute the exact same kernel code. In *Thread and Grid Index Intrinsics*, we'll show how each thread can use its index within the thread block and grid to change the data it operates on.

There is a limit to the number of threads per block, since all threads of a block reside on the same streaming multiprocessor(SM) and must share the resources of the SM. On current GPUs, a thread block may contain up to 1024 threads. If resources allow, more than one thread block can be scheduled on an SM simultaneously.

Kernel launches are asynchronous with respect to the host thread. That is, the kernel will be setup for execution on the GPU, but the host code will not wait for the kernel to complete (or even start) executing on the GPU before proceeding. Some form of synchronization between the GPU and CPU must be used to determine that the kernel has completed. The most basic version, completely synchronizing the entire GPU, is shown in *Synchronizing CPU and GPU*. More sophisticated methods of synchronization are covered in *Asynchronous Execution*.

When using 2 or 3-dimensional grids or thread blocks, the CUDA type `dim3` is used as the grid and thread block dimension parameters. The code fragment below shows a kernel launch of a `MatAdd` kernel using 16 by 16 grid of thread blocks, each thread block is 8 by 8.

```
int main()
{
    ...
    dim3 grid(16,16);
    dim3 block(8,8);
    MatAdd<<<grid, block>>>(A, B, C);
    ...
}
```

2.1.2.3 Thread and Grid Index Intrinsics

Within kernel code, CUDA provides intrinsics to access parameters of the execution configuration and the index of a thread or block.

- `threadIdx` gives the index of a thread within its thread block. Each thread in a thread block will have a different index.
- `blockDim` gives the dimensions of the thread block, which was specified in the execution configuration of the kernel launch.
- `blockIdx` gives the index of a thread block within the grid. Each thread block will have a different index.
- `gridDim` gives the dimensions of the grid, which was specified in the execution configuration when the kernel was launched.

Each of these intrinsics is a 3-component vector with a `.x`, `.y`, and `.z` member. Dimensions not specified by a launch configuration will default to 1. `threadIdx` and `blockIdx` are zero indexed. That is, `threadIdx.x` will take on values from 0 up to and including `blockDim.x-1`. `.y` and `.z` operate the same in their respective dimensions.

Similarly, `blockIdx.x` will have values from 0 up to and including `gridDim.x-1`, and the same for `.y` and `.z` dimensions, respectively.

These allow an individual thread to identify what work it should carry out. Returning to the `vecAdd` kernel, the kernel takes three parameters, each is a vector of floats. The kernel performs an element-wise addition of A and B and stores the result in C. The kernel is parallelized such that each thread will perform one addition. Which element it computes is determined by its thread and grid index.

```
__global__ void vecAdd(float* A, float* B, float* C)
{
   // calculate which element this thread is responsible for computing
   int workIndex = threadIdx.x + blockDim.x * blockIdx.x

   // Perform computation
   C[workIndex] = A[workIndex] + B[workIndex];
}

int main()
```

(continues on next page)

(continued from previous page)

```
{
    ...
    // A, B, and C are vectors of 1024 elements
    vecAdd<<<4, 256>>>(A, B, C);
    ...
}
```

In this example, 4 thread blocks of 256 threads are used to add a vector of 1024 elements. In the first thread block, `blockIdx.x` will be zero, and so each thread's workIndex will simply be its `threadIdx.x`. In the second thread block, `blockIdx.x` will be 1, so `blockDim.x * blockIdx.x` will be the same as `blockDim.x`, which is 256 in this case. The `workIndex` for each thread in the second thread block will be its `threadIdx.x + 256`. In the third thread block `workIndex` will be `threadIdx.x + 512`.

This computation of `workIndex` is very common for 1-dimensional parallelizations. Expanding to two or three dimensions often follows the same pattern in each of those dimensions.

2.1.2.3.1 Bounds Checking

The example given above assumes that the length of the vector is a multiple of the thread block size, 256 threads in this case. To make the kernel handle any vector length, we can add checks that the memory access is not exceeding the bounds of the arrays as shown below, and then launch one thread block which will have some inactive threads.

```
__global__ void vecAdd(float* A, float* B, float* C, int vectorLength)
{
     // calculate which element this thread is responsible for computing
     int workIndex = threadIdx.x + blockDim.x * blockIdx.x

     if(workIndex < vectorLength)
     {
         // Perform computation
         C[workIndex] = A[workIndex] + B[workIndex];
     }
}
```

With the above kernel code, more threads than needed can be launched without causing out-of-bounds accesses to the arrays. When `workIndex` exceeds `vectorLength`, threads exit and do not do any work. Launching extra threads in a block that do no work does not incur a large overhead cost, however launching thread blocks in which no threads do work should be avoided. This kernel can now handle vector lengths which are not a multiple of the block size.

The number of thread blocks which are needed can be calculated as the ceiling of the number of threads needed, the vector length in this case, divided by the number of threads per block. That is, the integer division of the number of threads needed by the number of threads per block, rounded up. A common way of expressing this as a single integer division is given below. By adding `threads - 1` before the integer division, this behaves like a ceiling function, adding another thread block only if the vector length is not divisible by the number of threads per block.

```
// vectorLength is an integer storing number of elements in the vector
int threads = 256;
int blocks = (vectorLength + threads-1)/threads;
vecAdd<<<blocks, threads>>>(devA, devB, devC, vectorLength);
```

The CUDA Core Compute Library (CCCL) provides a convenient utility, `cuda::ceil_div`, for doing this ceiling divide to calculate the number of blocks needed for a kernel launch. This utility is available by including the header `<cuda/cmath>`.

```
// vectorLength is an integer storing number of elements in the vector
int threads = 256;
int blocks = cuda::ceil_div(vectorLength, threads);
vecAdd<<<blocks, threads>>>(devA, devB, devC, vectorLength);
```

The choice of 256 threads per block here is arbitrary, but this is quite often a good value to start with.

2.1.3. Memory in GPU Computing

In order to use the `vecAdd` kernel shown above, the arrays `A`, `B`, and `C` must be in memory accessible to the GPU. There are several different ways to do this, two of which will be illustrated here. Other methods will be covered in later sections on *unified memory*. The memory spaces available to code running on the GPU were introduced in *GPU Memory* and are covered in more detail in *GPU Device Memory Spaces*.

2.1.3.1 Unified Memory

Unified memory is a feature of the CUDA runtime which lets the NVIDIA Driver manage movement of data between host and device(s). Memory is allocated using the `cudaMallocManaged` API or by declaring a variable with the `__managed__` specifier. The NVIDIA Driver will make sure that the memory is accessible to the GPU or CPU whenever either tries to access it.

The code below shows a complete function to launch the `vecAdd` kernel which uses unified memory for the input and output vectors that will be used on the GPU. `cudaMallocManaged` allocates buffers which can be accessed from either the CPU or the GPU. These buffers are released using `cudaFree`.

```
void unifiedMemExample(int vectorLength)
{
    // Pointers to memory vectors
    float* A = nullptr;
    float* B = nullptr;
    float* C = nullptr;
    float* comparisonResult = (float*)malloc(vectorLength*sizeof(float));

    // Use unified memory to allocate buffers
    cudaMallocManaged(&A, vectorLength*sizeof(float));
    cudaMallocManaged(&B, vectorLength*sizeof(float));
    cudaMallocManaged(&C, vectorLength*sizeof(float));

    // Initialize vectors on the host
    initArray(A, vectorLength);
    initArray(B, vectorLength);

    // Launch the kernel. Unified memory will make sure A, B, and C are
    // accessible to the GPU
    int threads = 256;
    int blocks = cuda::ceil_div(vectorLength, threads);
    vecAdd<<<blocks, threads>>>(A, B, C, vectorLength);
    // Wait for the kernel to complete execution
    cudaDeviceSynchronize();
```

(continues on next page)

(continued from previous page)

```
    // Perform computation serially on CPU for comparison
    serialVecAdd(A, B, comparisonResult, vectorLength);

    // Confirm that CPU and GPU got the same answer
    if(vectorApproximatelyEqual(C, comparisonResult, vectorLength))
    {
        printf("Unified Memory: CPU and GPU answers match\n");
    }
    else
    {
        printf("Unified Memory: Error - CPU and GPU answers do not match\n");
    }

    // Clean Up
    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
    free(comparisonResult);

}
```

Unified memory is supported on all operating systems and GPUs supported by CUDA, though the underlying mechanism and performance may differ based on system architecture. *Unified Memory* provides more details. On some Linux systems, (e.g. those with *address translation services* or *heterogeneous memory management*) all system memory is automatically unified memory, and there is no need to use cudaMallocManaged or the __managed__ specifier.

2.1.3.2 Explicit Memory Management

Explicitly managing memory allocation and data migration between memory spaces can help improve application performance, though it does make for more verbose code. The code below explicitly allocates memory on the GPU using cudaMalloc. Memory on the GPU is freed using the same cudaFree API as was used for unified memory in the previous example.

```
void explicitMemExample(int vectorLength)
{
    // Pointers for host memory
    float* A = nullptr;
    float* B = nullptr;
    float* C = nullptr;
    float* comparisonResult = (float*)malloc(vectorLength*sizeof(float));

    // Pointers for device memory
    float* devA = nullptr;
    float* devB = nullptr;
    float* devC = nullptr;

    //Allocate Host Memory using cudaMallocHost API. This is best practice
    // when buffers will be used for copies between CPU and GPU memory
    cudaMallocHost(&A, vectorLength*sizeof(float));
    cudaMallocHost(&B, vectorLength*sizeof(float));
```

(continues on next page)

(continued from previous page)

```
    cudaMallocHost(&C, vectorLength*sizeof(float));

    // Initialize vectors on the host
    initArray(A, vectorLength);
    initArray(B, vectorLength);

    // start-allocate-and-copy
    // Allocate memory on the GPU
    cudaMalloc(&devA, vectorLength*sizeof(float));
    cudaMalloc(&devB, vectorLength*sizeof(float));
    cudaMalloc(&devC, vectorLength*sizeof(float));

    // Copy data to the GPU
    cudaMemcpy(devA, A, vectorLength*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(devB, B, vectorLength*sizeof(float), cudaMemcpyDefault);
    cudaMemset(devC, 0, vectorLength*sizeof(float));
    // end-allocate-and-copy

    // Launch the kernel
    int threads = 256;
    int blocks = cuda::ceil_div(vectorLength, threads);
    vecAdd<<<blocks, threads>>>(devA, devB, devC);
    // wait for kernel execution to complete
    cudaDeviceSynchronize();

    // Copy results back to host
    cudaMemcpy(C, devC, vectorLength*sizeof(float), cudaMemcpyDefault);

    // Perform computation serially on CPU for comparison
    serialVecAdd(A, B, comparisonResult, vectorLength);

    // Confirm that CPU and GPU got the same answer
    if(vectorApproximatelyEqual(C, comparisonResult, vectorLength))
    {
        printf("Explicit Memory: CPU and GPU answers match\n");
    }
    else
    {
        printf("Explicit Memory: Error - CPU and GPU answers to not match\n");
    }

    // clean up
    cudaFree(devA);
    cudaFree(devB);
    cudaFree(devC);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);
    free(comparisonResult);
}
```

The CUDA API `cudaMemcpy` is used to copy data from a buffer residing on the CPU to a buffer residing on the GPU. Along with the destination pointer, source pointer, and size in bytes, the final parameter of `cudaMemcpy` is a `cudaMemcpyKind_t`. This can have values such as `cudaMemcpyHostToDevice` for copies from the CPU to a GPU, `cudaMemcpyDeviceToHost` for copies from the CPU to the GPU, or `cudaMemcpyDeviceToDevice` for copies within a GPU or between GPUs.

In this example, `cudaMemcpyDefault` is passed as the last argument to `cudaMemcpy`. This causes CUDA to use the value of the source and destination pointers to determine the type of copy to perform.

The `cudaMemcpy` API is synchronous. That is, it does not return until the copy has completed. Asynchronous copies are introduced in *Launching Memory Transfers in CUDA Streams*.

The code uses `cudaMallocHost` to allocate memory on the CPU. This allocates *page-locked memory* on the host, which can improve copy performance and is necessary for *asynchronous* memory transfers. In general, it is good practice to use page-locked memory for CPU buffers that will be used in data transfers to and from GPUs. Performance can degrade on some systems if too much host memory is page-locked. Best practice is to page-lock only buffers which will be used for sending or receiving data from the GPU.

2.1.3.3 Memory Management and Application Performance

As can be seen in the above example, explicit memory management is more verbose, requiring the programmer to specify copies between the host and device. This is the advantage and disadvantage of explicit memory management: it affords more control of when data is copied between host and devices, where memory is resident, and exactly what memory is allocated where. Explicit memory management can provide performance opportunities controlling memory transfers and overlapping them with other computations.

When using unified memory, there are CUDA APIs (which will be covered in *Memory Advise and Prefetch*), which provide hints to the NVIDIA driver managing the memory, which can enable some of the performance benefits of using explicit memory management when using unified memory.

2.1.4. Synchronizing CPU and GPU

As mentioned in *Launching Kernels*, kernel launches are asynchronous with respect to the CPU thread which called them. This means the control flow of the CPU thread will continue executing before the kernel has completed, and possibly even before it has launched. In order to guarantee that a kernel has completed execution before proceeding in host code, some synchronization mechanism is necessary.

The simplest way to synchronize the GPU and a host thread is with the use of `cudaDeviceSynchronize`, which blocks the host thread until all previously issued work on the GPU has completed. In the examples of this chapter this is sufficient because only single operations are being executed on the GPU. In larger applications, there may be multiple *streams* executing work on the GPU and `cudaDeviceSynchronize` will wait for work in all streams to complete. In these applications, using *Stream Synchronization* APIs to synchronize only with a specific stream or *CUDA Events* is recommended. These will be covered in detail in the *Asynchronous Execution* chapter.

2.1.5. Putting it All Together

The following listings show the entire code for the simple vector addition kernel introduced in this chapter along with all host code and utility functions for checking to verify that the answer obtained is correct. These examples default to using a vector length of 1024, but accept a different vector length as a command line argument to the executable.

Unified Memory

```
#include <cuda_runtime_api.h>
#include <memory.h>
#include <cstdlib>
#include <ctime>
#include <stdio.h>
#include <cuda/cmath>

__global__ void vecAdd(float* A, float* B, float* C, int vectorLength)
{
    int workIndex = threadIdx.x + blockIdx.x*blockDim.x;
    if(workIndex < vectorLength)
    {
        C[workIndex] = A[workIndex] + B[workIndex];
    }
}

void initArray(float* A, int length)
{
     std::srand(std::time({}));
    for(int i=0; i<length; i++)
    {
        A[i] = rand() / (float)RAND_MAX;
    }
}

void serialVecAdd(float* A, float* B, float* C,  int length)
{
    for(int i=0; i<length; i++)
    {
        C[i] = A[i] + B[i];
    }
}

bool vectorApproximatelyEqual(float* A, float* B, int length, float epsilon=0.
↪00001)
{
    for(int i=0; i<length; i++)
    {
        if(fabs(A[i] -B[i]) > epsilon)
        {
            printf("Index %d mismatch: %f != %f", i, A[i], B[i]);
            return false;
        }
    }
    return true;
}

//unified-memory-begin
void unifiedMemExample(int vectorLength)
{
    // Pointers to memory vectors
```

(continues on next page)

(continued from previous page)

```
    float* A = nullptr;
    float* B = nullptr;
    float* C = nullptr;
    float* comparisonResult = (float*)malloc(vectorLength*sizeof(float));

    // Use unified memory to allocate buffers
    cudaMallocManaged(&A, vectorLength*sizeof(float));
    cudaMallocManaged(&B, vectorLength*sizeof(float));
    cudaMallocManaged(&C, vectorLength*sizeof(float));

    // Initialize vectors on the host
    initArray(A, vectorLength);
    initArray(B, vectorLength);

    // Launch the kernel. Unified memory will make sure A, B, and C are
    // accessible to the GPU
    int threads = 256;
    int blocks = cuda::ceil_div(vectorLength, threads);
    vecAdd<<<blocks, threads>>>(A, B, C, vectorLength);
    // Wait for the kernel to complete execution
    cudaDeviceSynchronize();

    // Perform computation serially on CPU for comparison
    serialVecAdd(A, B, comparisonResult, vectorLength);

    // Confirm that CPU and GPU got the same answer
    if(vectorApproximatelyEqual(C, comparisonResult, vectorLength))
    {
        printf("Unified Memory: CPU and GPU answers match\n");
    }
    else
    {
        printf("Unified Memory: Error - CPU and GPU answers do not match\n");
    }

    // Clean Up
    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
    free(comparisonResult);

}
//unified-memory-end


int main(int argc, char** argv)
{
    int vectorLength = 1024;
    if(argc >=2)
    {
        vectorLength = std::atoi(argv[1]);
    }
```

(continues on next page)

(continued from previous page)

```
    unifiedMemExample(vectorLength);
    return 0;
}
```

Explicit Memory Management

```
#include <cuda_runtime_api.h>
#include <memory.h>
#include <cstdlib>
#include <ctime>
#include <stdio.h>
#include <cuda/cmath>

__global__ void vecAdd(float* A, float* B, float* C, int vectorLength)
{
    int workIndex = threadIdx.x + blockIdx.x*blockDim.x;
    if(workIndex < vectorLength)
    {
        C[workIndex] = A[workIndex] + B[workIndex];
    }
}

void initArray(float* A, int length)
{
     std::srand(std::time({}));
    for(int i=0; i<length; i++)
    {
        A[i] = rand() / (float)RAND_MAX;
    }
}

void serialVecAdd(float* A, float* B, float* C,  int length)
{
    for(int i=0; i<length; i++)
    {
        C[i] = A[i] + B[i];
    }
}

bool vectorApproximatelyEqual(float* A, float* B, int length, float epsilon=0.
↪00001)
{
    for(int i=0; i<length; i++)
    {
        if(fabs(A[i] -B[i]) > epsilon)
        {
            printf("Index %d mismatch: %f != %f", i, A[i], B[i]);
            return false;
        }
    }
    return true;
```

(continues on next page)

(continued from previous page)

```
}

//explicit-memory-begin
void explicitMemExample(int vectorLength)
{
    // Pointers for host memory
    float* A = nullptr;
    float* B = nullptr;
    float* C = nullptr;
    float* comparisonResult = (float*)malloc(vectorLength*sizeof(float));

    // Pointers for device memory
    float* devA = nullptr;
    float* devB = nullptr;
    float* devC = nullptr;

    //Allocate Host Memory using cudaMallocHost API. This is best practice
    // when buffers will be used for copies between CPU and GPU memory
    cudaMallocHost(&A, vectorLength*sizeof(float));
    cudaMallocHost(&B, vectorLength*sizeof(float));
    cudaMallocHost(&C, vectorLength*sizeof(float));

    // Initialize vectors on the host
    initArray(A, vectorLength);
    initArray(B, vectorLength);

    // start-allocate-and-copy
    // Allocate memory on the GPU
    cudaMalloc(&devA, vectorLength*sizeof(float));
    cudaMalloc(&devB, vectorLength*sizeof(float));
    cudaMalloc(&devC, vectorLength*sizeof(float));

    // Copy data to the GPU
    cudaMemcpy(devA, A, vectorLength*sizeof(float), cudaMemcpyDefault);
    cudaMemcpy(devB, B, vectorLength*sizeof(float), cudaMemcpyDefault);
    cudaMemset(devC, 0, vectorLength*sizeof(float));
    // end-allocate-and-copy

    // Launch the kernel
    int threads = 256;
    int blocks = cuda::ceil_div(vectorLength, threads);
    vecAdd<<<blocks, threads>>>(devA, devB, devC);
    // wait for kernel execution to complete
    cudaDeviceSynchronize();

    // Copy results back to host
    cudaMemcpy(C, devC, vectorLength*sizeof(float), cudaMemcpyDefault);

    // Perform computation serially on CPU for comparison
    serialVecAdd(A, B, comparisonResult, vectorLength);

    // Confirm that CPU and GPU got the same answer
```

(continues on next page)

(continued from previous page)

```
    if(vectorApproximatelyEqual(C, comparisonResult, vectorLength))
    {
        printf("Explicit Memory: CPU and GPU answers match\n");
    }
    else
    {
        printf("Explicit Memory: Error - CPU and GPU answers to not match\n");
    }

    // clean up
    cudaFree(devA);
    cudaFree(devB);
    cudaFree(devC);
    cudaFreeHost(A);
    cudaFreeHost(B);
    cudaFreeHost(C);
    free(comparisonResult);
}
//explicit-memory-end


int main(int argc, char** argv)
{
    int vectorLength = 1024;
    if(argc >=2)
    {
        vectorLength = std::atoi(argv[1]);
    }
    explicitMemExample(vectorLength);
    return 0;
}
```

These can be built and run using *nvcc* as follows:

```
$ nvcc vecAdd_unifiedMemory.cu -o vecAdd_unifiedMemory
$ ./vecAdd_unifiedMemory
Unified Memory: CPU and GPU answers match
$ ./vecAdd_unifiedMemory 4096
Unified Memory: CPU and GPU answers match
```

```
$ nvcc vecAdd_explicitMemory.cu -o vecAdd_explicitMemory
$ ./vecAdd_explicitMemory
Explicit Memory: CPU and GPU answers match
$ ./vecAdd_explicitMemory 4096
Explicit Memory: CPU and GPU answers match
```

In these examples, all threads are doing independent work and do not need to coordinate or synchronize with each other. Frequently, threads will need to cooperate and communicate with other threads to carry out their work. Threads within a block can share data through *shared memory* and synchronize to coordinate memory accesses.

The most basic mechanism for synchronization at the block level is the `__syncthreads()` intrinsic, which acts as a barrier at which all threads in the block must wait before any threads are allowed to proceed. *Shared Memory* gives an example of using shared memory.

For efficient cooperation, shared memory is expected to be a low-latency memory near each processor core (much like an L1 cache) and `__syncthreads()` is expected to be lightweight. `__syncthreads()` only synchronizes the threads within a single thread block. Synchronization between blocks is not supported by the CUDA programming model. *Cooperative Groups* provides mechanism to set synchronization domains other than a single thread block.

Best performance is usually achieved when synchronization is kept within a thread block. Thread blocks can still work on common results using *atomic memory functions*, which will be covered in coming sections.

Section Section 3.2.4 covers CUDA synchronization primitives that provide very fine-grained control for maximizing performance and resource utilization.

2.1.6. Runtime Initialization

The CUDA runtime creates a *CUDA context* for each device in the system. This context is the primary context for this device and is initialized at the first runtime function which requires an active context on this device. The context is shared among all the host threads of the application. As part of context creation, the device code is *just-in-time compiled* if necessary and loaded into device memory. This all happens transparently. The primary context created by the CUDA runtime can be accessed from the driver API for interoperability as described in *Interoperability between Runtime and Driver APIs*.

As of CUDA 12.0, the `cudaInitDevice` and `cudaSetDevice` calls initialize the runtime and the primary *context* associated with the specified device. The runtime will implicitly use device 0 and self-initialize as needed to process runtime API requests if they occur before these calls. This is important when timing runtime function calls and when interpreting the error code from the first call into the runtime. Prior to CUDA 12.0, `cudaSetDevice` would not initialize the runtime.

`cudaDeviceReset` destroys the primary context of the current device. If CUDA runtime APIs are called after the primary context has been destroyed, a new primary context for that device will be created.

Note

The CUDA interfaces use global state that is initialized during host program initiation and destroyed during host program termination. Using any of these interfaces (implicitly or explicitly) during program initiation or termination after main will result in undefined behavior.

As of CUDA 12.0, `cudaSetDevice` explicitly initializes the runtime, if it has not already been initialized, after changing the current device for the host thread. Previous versions of CUDA delayed runtime initialization on the new device until the first runtime call was made after `cudaSetDevice`. Because of this, it is very important to check the return value of `cudaSetDevice` for initialization errors.

The runtime functions from the error handling and version management sections of the reference manual do not initialize the runtime.

2.1.7. Error Checking in CUDA

Every CUDA API returns a value of an enumerated type, `cudaError_t`. In example code these errors are often not checked. In production applications, it is best practice to always check and manage the return value of every CUDA API call. When there are no errors, the value returned is `cudaSuccess`. Many applications choose to implement a utility macro such as the one shown below

```
#define CUDA_CHECK(expr_to_check) do {            \
    cudaError_t result  = expr_to_check;          \
    if(result != cudaSuccess)                     \
    {                                             \
        fprintf(stderr,                           \
                "CUDA Runtime Error: %s:%i:%d = %s\n", \
                __FILE__,                         \
                __LINE__,                         \
                result,\
                cudaGetErrorString(result));      \
    }                                             \
} while(0)
```

This macro uses the `cudaGetErrorString` API, which returns a human readable string describing the meaning of a specific `cudaError_t` value. Using the above macro, an application would call CUDA runtime API calls within a `CUDA_CHECK(expression)` macro, as shown below:

```
    CUDA_CHECK(cudaMalloc(&devA, vectorLength*sizeof(float)));
    CUDA_CHECK(cudaMalloc(&devB, vectorLength*sizeof(float)));
    CUDA_CHECK(cudaMalloc(&devC, vectorLength*sizeof(float)));
```

If any of these calls detect an error, it will be printed to `stderr` using this macro. This macro is common for smaller projects, but can be adapted to a logging system or other error handling mechanism in larger applications.

Note

It is important to note that the error state returned from any CUDA API call can also indicate an error from a previously issued asynchronous operation. Section *Asynchronous Error Handling* covers this in more detail.

2.1.7.1 Error State

The CUDA runtime maintains a `cudaError_t` state for each host thread. The value defaults to `cudaSuccess` and is overwritten whenever an error occurs. `cudaGetLastError` returns current error state and then resets it to `cudaSuccess`. Alternatively, `cudaPeekLastError` returns error state without resetting it.

Kernel launches using *triple chevron notation* do not return a `cudaError_t`. It is good practice to check the error state immediately after kernel launches to detect immediate errors in the kernel launch or *asynchronous errors* prior to the kernel launch. A value of `cudaSuccess` when checking the error state immediately after a kernel launch does not mean the kernel has executed successfully or even started execution. It only verifies that the kernel launch parameters and execution configuration passed to the runtime did not trigger any errors and that the error state is not a previous or asynchronous error before the kernel started.

2.1.7.2 Asynchronous Errors

CUDA kernel launches and many runtime APIs are asynchronous. Asynchronous CUDA runtime APIs will be discussed in detail in *Asynchronous Execution*. The CUDA error state is set and overwritten whenever an error occurs. This means that errors which occur during the execution of asynchronous operations will only be reported when the error state is examined next. As noted, this may be a call to `cudaGetLastError`, `cudaPeekLastError`, or it could be any CUDA API which returns `cudaError_t`.

When errors are returned by CUDA runtime API functions, the error state is not cleared. This means that error code from an asynchronous error, such as an invalid memory access by a kernel, will be returned by every CUDA runtime API until the error state has been cleared by calling `cudaGetLastError`.

```
    vecAdd<<<blocks, threads>>>(devA, devB, devC);
    // check error state after kernel launch
    CUDA_CHECK(cudaGetLastError());
    // wait for kernel execution to complete
    // The CUDA_CHECK will report errors that occurred during execution of the
↪kernel
    CUDA_CHECK(cudaDeviceSynchronize());
```

Note

The `cudaError_t` value `cudaErrorNotReady`, which may be returned by `cudaStreamQuery` and `cudaEventQuery`, is not considered an error and is not reported by `cudaPeekAtLastError` or `cudaGetLastError`.

2.1.7.3 CUDA_LOG_FILE

Another good way to identify CUDA errors is with the `CUDA_LOG_FILE` environment variable. When this environment variable is set, the CUDA driver will write error messages encountered out to a file whose path is specified in the environment variable. For example, take the following incorrect CUDA code, which attemtps to launch a thread block which is larger than the maximum supported by any architecture.

```
__global__ void k()
{ }

int main()
{
        k<<<8192, 4096>>>(); // Invalid block size
        CUDA_CHECK(cudaGetLastError());
        return 0;
}
```

Building and running this, the check after the kernel launch detects and reports the error using the macros illustrated in Section 2.1.7.

```
$ nvcc errorLogIllustration.cu -o errlog
$ ./errlog
CUDA Runtime Error: /home/cuda/intro-cpp/errorLogIllustration.cu:24:1 =
↪invalid argument
```

However, when the application is run with `CUDA_LOG_FILE` set to a text file, that file contains a bit more information about the error.

```
$ env CUDA_LOG_FILE=cudaLog.txt ./errlog
CUDA Runtime Error: /home/cuda/intro-cpp/errorLogIllustration.cu:24:1 =
↪invalid argument
$ cat cudaLog.txt
```

(continues on next page)

(continued from previous page)

```
[12:46:23.854][137216133754880][CUDA][E] One or more of block dimensions of
↪(4096,1,1) exceeds corresponding maximum value of (1024,1024,64)
[12:46:23.854][137216133754880][CUDA][E] Returning 1 (CUDA_ERROR_INVALID_
↪VALUE) from cuLaunchKernel
```

Setting `CUDA_LOG_FILE` to `stdout` or `stderr` will print to standard out and standard error, respectively. Using the `CUDA_LOG_FILE` environment variable, it is possible to capture and identify CUDA errors, even if the application does not implement proper error checking on CUDA return values. This approach can be extremely powerful for debugging, but the environment variable alone does not allow an application to handle and recover from CUDA errors at runtime. The *error log management* feature of CUDA also allows a callback function to be registered with the driver which will be called whenever an error is detected. This can be used to capture and handle errors at runtime, and also to integrate CUDA error logging seamlessly into an application's existing logging system.

Section 4.8 shows more examples of the error log management feature of CUDA. Error log management and `CUDA_LOG_FILE` are available with NVIDIA Driver version r570 and later.

2.1.8. Device and Host Functions

The `__global__` specifier is used to indicate the entry point for a kernel. That is, a function which will be invoked for parallel execution on the GPU. Most often, kernels are launched from the host, however it is possible to launch a kernel from within another kernel using *dynamic parallelism*.

The specifier `__device__` indicates that a function should be compiled for the GPU and be callable from other `__device__` or `__global__` functions. A function, including class member functions, functors, and lambdas, can be specified as both `__device__` and `__host__` as in the example below.

2.1.9. Variable Specifiers

CUDA specifiers can be used on static variable declarations to control placement.

- `__device__` specifies that a variable is stored in *Global Memory*
- `__constant__` specifies that a variable is stored in *Constant Memory*
- `__managed__` specifies that a variable is stored as *Unified Memory*
- `__shared__` specifies that a variable is store in *Shared Memory*

When a variable is declared with no specifier inside a `__device__` or `__global__` function, it is allocated to registers when possible, and *local memory* when necessary. Any variable declared with no specifier outside a `__device__` or `__global__` function will be allocated in system memory.

2.1.9.1 Detecting Device Compilation

When a function is specified with `__host__ __device__`, the compiler is instructed to generate both a GPU and a CPU code for this function. In such functions, it may be desirable to use the preprocessor to specify code only for the GPU or the CPU copy of the function. Checking whether `__CUDA_ARCH_` is defined is the most common way of doing this, as illustrated in the example below.

2.1.10. Thread Block Clusters

From compute capability 9.0 onward, the CUDA programming model includes an optional level of hierarchy called thread block clusters that are made up of thread blocks. Similar to how threads in a thread block are guaranteed to be co-scheduled on a streaming multiprocessor, thread blocks in a cluster are also guaranteed to be co-scheduled on a GPU Processing Cluster (GPC) in the GPU.

Similar to thread blocks, clusters are also organized into a one-dimension, two-dimension, or three-dimension grid of thread block clusters as illustrated by Figure 5.

The number of thread blocks in a cluster can be user-defined, and a maximum of 8 thread blocks in a cluster is supported as a portable cluster size in CUDA. Note that on GPU hardware or MIG configurations which are too small to support 8 multiprocessors the maximum cluster size will be reduced accordingly. Identification of these smaller configurations, as well as of larger configurations supporting a thread block cluster size beyond 8, is architecture-specific and can be queried using the `cudaOccupancyMaxPotentialClusterSize` API.

All the thread blocks in the cluster are guaranteed to be co-scheduled to execute simultaneously on a single GPU Processing Cluster (GPC) and allow thread blocks in the cluster to perform hardware-supported synchronization using the *cooperative groups* API `cluster.sync()`. Cluster group also provides member functions to query cluster group size in terms of number of threads or number of blocks using `num_threads()` and `num_blocks()` API respectively. The rank of a thread or block in the cluster group can be queried using `dim_threads()` and `dim_blocks()` API respectively.

Thread blocks that belong to a cluster have access to the *distributed shared memory*, which is the combined shared memory of all thread blocks in the cluster. Thread blocks in a cluster have the ability to read, write, and perform atomics to any address in the distributed shared memory. *Distributed Shared Memory* gives an example of performing histograms in distributed shared memory.

Note

In a kernel launched using cluster support, the gridDim variable still denotes the size in terms of number of thread blocks, for compatibility purposes. The rank of a block in a cluster can be found using the *Cooperative Groups* API.

2.1.10.1 Launching with Clusters in Triple Chevron Notation

A thread block cluster can be enabled in a kernel either using a compile-time kernel attribute using `__cluster_dims__(X,Y,Z)` or using the CUDA kernel launch API `cudaLaunchKernelEx`. The example below shows how to launch a cluster using a compile-time kernel attribute. The cluster size using kernel attribute is fixed at compile time and then the kernel can be launched using the classical `<<< , >>>`. If a kernel uses compile-time cluster size, the cluster size cannot be modified when launching the kernel.

```
// Kernel definition
// Compile time cluster size 2 in X-dimension and 1 in Y and Z dimension
__global__ void __cluster_dims__(2, 1, 1) cluster_kernel(float *input, float*
↪output)
{

}

int main()
{
```

(continues on next page)

(continued from previous page)

```
    float *input, *output;
    // Kernel invocation with compile time cluster size
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    // The grid dimension is not affected by cluster launch, and is still
↪enumerated
    // using number of blocks.
    // The grid dimension must be a multiple of cluster size.
    cluster_kernel<<<numBlocks, threadsPerBlock>>>(input, output);
}
```

2.2. Writing CUDA SIMT Kernels

CUDA C++ kernels can largely be written in the same way that traditional CPU code would be written for a given problem. However, there are some unique features of the GPU that can be used to improve performance. Additionally, some understanding of how threads on the GPU are scheduled, how they access memory, and how their execution proceeds can help developers write kernels that maximize utilization of the available computing resources.

2.2.1. Basics of SIMT

From the developer's perspective, the CUDA thread is the fundamental unit of parallelism. *Warps and SIMT* describes the basic SIMT model of GPU execution and *SIMT Execution Model* provides additional details of the SIMT model. The SIMT model allows each thread to maintain its own state and control flow. From a functional perspective, each thread can execute a separate code path. However, substantial performance improvements can be realized by taking care that kernel code minimizes the situations where threads in the same warp take divergent code paths.

2.2.2. Thread Hierarchy

Threads are organized into thread blocks, which are then organized into a grid. Grids may be 1, 2, or 3 dimensional and the size of the grid can be queried inside a kernel with the `gridDim` built-in variable. Thread blocks may also be 1, 2, or 3 dimensional. The size of the thread block can be queried inside a kernel with the `blockDim` built-in variable. The index of the thread block can be queried with the `blockIdx` built-in variable. Within a thread block, the index of the thread is obtained using the `threadIdx` built-in variable. These built-in variables are used to compute a unique global thread index for each thread, thereby enabling each thread to load/store specific data from global memory and execute a unique code path as needed.

- `gridDim.[x|y|z]`: Size of the grid in the x, y and z dimension respectively. These values are set at kernel launch.
- `blockDim.[x|y|z]`: Size of the block in the x, y and z dimension respectively. These values are set at kernel launch.
- `blockIdx.[x|y|z]`: Index of the block in the x, y and z dimension respectively. These values change depending on which block is executing.

- `threadIdx.[x|y|z]`: Index of the thread in the x, y and z dimension respectively. These values change depending on which thread is executing.

The use of multi-dimensional thread blocks and grids is for convenience only and does not affect performance. The threads of a block are linearized predictably: the first index x moves the fastest, followed by y and then z. This means that in the linearization of a thread indices, consecutive values of `threadIdx.x` indicate consecutive threads, `threadIdx.y` has a stride of `blockDim.x`, and `threadIdx.z` has a stride of `blockDim.x * blockDim.y`. This affects how threads are assigned to warps, as detailed in *Hardware Multithreading*.

Figure 9 shows a simple example of a 2D grid, with 1D thread blocks.

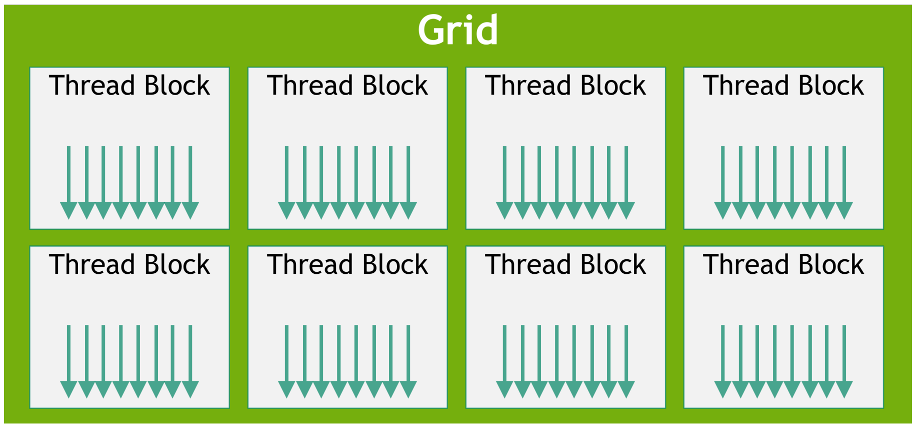


Figure 9: Grid of Thread Blocks

2.2.3. GPU Device Memory Spaces

CUDA devices have several memory spaces that can be accessed by CUDA threads within kernels. Table 1 shows a summary of the common memory types, their thread scopes, and their lifetimes. The following sections explain each of these memory types in more detail.

Table 1: Memory Types, Scopes and Lifetimes

Memory Type	Scope	Lifetime	Location
Global	Grid	Application	Device
Constant	Grid	Application	Device
Shared	Block	Kernel	SM
Local	Thread	Kernel	Device
Register	Thread	Kernel	SM

2.2.3.1 Global Memory

Global memory (also called device memory) is the primary memory space for storing data that is accessible by all threads in a kernel. It is similar to RAM in a CPU system. Kernels running on the GPU have direct access to global memory in the same way code running on the CPU has access to system memory.

Global memory is persistent. That is, an allocation made in global memory and the data stored in it persist until the allocation is freed or until the application is terminated. `cudaDeviceReset` also frees all allocations.

Global memory is allocated with CUDA API calls such as `cudaMalloc` and `cudaMallocManaged`. Data can be copied into global memory from CPU memory using CUDA runtime API calls such as `cudaMemcpy`. Global memory allocations made with CUDA APIs are freed using `cudaFree`.

Prior to a kernel launch, global memory is allocated and initialized by CUDA API calls. During kernel execution, data from global memory can be read by the CUDA threads, and the result from operations carried out by CUDA threads can be written back to global memory. Once a kernel has completed execution, the results it wrote to global memory can be copied back to the host or used by other kernels on the GPU.

Because global memory is accessible by all threads in a grid, care must be taken to avoid data races between threads. Since CUDA kernels launched from the host have the return type `void`, the only way for numerical results computed by a kernel to be returned to the host is by writing those results to global memory.

A simple example illustrating the use of global memory is the `vecAdd` kernel below, where the three arrays A, B, and C are in global memory and are being accessed by this vector add kernel.

```
__global__ void vecAdd(float* A, float* B, float* C, int vectorLength)
{
    int workIndex = threadIdx.x + blockIdx.x*blockDim.x;
    if(workIndex < vectorLength)
    {
        C[workIndex] = A[workIndex] + B[workIndex];
```

2.2.3.2 Shared Memory

Shared memory is a memory space that is accessible by all threads in a thread block. It is physically located on each SM and uses the same physical resource as the L1 cache, the unified data cache. The data in shared memory persists throughout the kernel execution. Shared memory can be considered a user-managed scratchpad for use during kernel execution. While small in size compared to global memory, because shared memory is located on each SM, the bandwidth is higher and the latency is lower than accessing global memory.

Since shared memory is accessible by all threads in a thread block, care must be taken to avoid data races between threads in the same thread block. Synchronization between threads in the same thread block can be achieved using the `__syncthreads()` function. This function blocks all threads in the thread block until all threads have reached the call to `__syncthreads()`.

```
// assuming blockDim.x is 128
__global__ void example_syncthreads(int* input_data, int* output_data) {
    __shared__ int shared_data[128];
    // Every thread writes to a distinct element of 'shared_data':
    shared_data[threadIdx.x] = input_data[threadIdx.x];

    // All threads synchronize, guaranteeing all writes to 'shared_data' are
↪ordered
    // before any thread is unblocked from '__syncthreads()':
    __syncthreads();

    // A single thread safely reads 'shared_data':
    if (threadIdx.x == 0) {
```

(continues on next page)

(continued from previous page)

```
        int sum = 0;
        for (int i = 0; i < blockDim.x; ++i) {
            sum += shared_data[i];
        }
        output_data[blockIdx.x] = sum;
    }
}
```

The size of shared memory varies depending on the GPU architecture being used. Because shared memory and L1 cache share the same physical space, using shared memory reduces the size of the usable L1 cache for a kernel. Additionally, if no shared memory is used by the kernel, the entire physical space will be utilized by L1 cache. The CUDA runtime API provides functions to query the shared memory size on a per SM basis and a per thread block basis, using the `cudaGetDeviceProperties` function and investigating the `cudaDeviceProp.sharedMemPerMultiprocessor` and `cudaDeviceProp.sharedMemPerBlock` device properties.

The CUDA runtime API provides a function `cudaFuncSetCacheConfig` to tell the runtime whether to allocate more space to shared memory, or more space to L1 cache. This function specifies a preference to the runtime, but is not guaranteed to be honored. The runtime is free to make decisions based on the available resources and the needs of the kernel.

Shared memory can be allocated both statically and dynamically.

2.2.3.2.1 Static Allocation of Shared Memory

To allocate shared memory statically, the programmer must declare a variable inside the kernel using the `__shared__` specifier. The variable will be allocated in shared memory and will persist for the duration of the kernel execution. The size of the shared memory declared in this way must be specified at compile time. For example, the following code snippet, located in the body of the kernel, declares a shared memory array of type `float` with 1024 elements.

```
__shared__ float sharedArray[1024];
```

After this declaration, all the threads in the thread block will have access to this shared memory array. Care must be taken to avoid data races between threads in the same thread block, typically with the use of `__syncthreads()`.

2.2.3.2.2 Dynamic Allocation of Shared Memory

To allocate shared memory dynamically, the programmer can specify the desired amount of shared memory per thread block in bytes as the third (and optional) argument to the kernel launch in the triple chevron notation like this `functionName<<<grid, block, sharedMemoryBytes>>>()`.

Then, inside the kernel, the programmer can use the `extern __shared__` specifier to declare a variable that will be allocated dynamically at kernel launch.

```
extern __shared__ float sharedArray[];
```

One caveat is that if one wants multiple dynamically allocated shared memory arrays, the single `extern __shared__` must be partitioned manually using pointer arithmetic. For example, if one wants the equivalent of the following,

```
short array0[128];
float array1[64];
int   array2[256];
```

in dynamically allocated shared memory, one could declare and initialize the arrays in the following way:

```
extern __shared__ float array[];

short* array0 = (short*)array;
float* array1 = (float*)&array0[128];
int*   array2 =   (int*)&array1[64];
```

Note that pointers need to be aligned to the type they point to, so the following code, for example, does not work since `array1` is not aligned to 4 bytes.

```
extern __shared__ float array[];
short* array0 = (short*)array;
float* array1 = (float*)&array0[127];
```

2.2.3.3 Registers

Registers are located on the SM and have thread local scope. Register usage is managed by the compiler and registers are used for thread local storage during the execution of a kernel. The number of registers per SM and the number of registers per thread block can be queried using the `regsPerMultiprocessor` and `regsPerBlock` device properties of the GPU.

NVCC allows the developer to specify a maximum number of registers to be used by a kernel via the `-maxrregcount` option. Using this option to reduce the number of registers a kernel can use may result in more thread blocks being scheduled on the SM concurrently, but may also result in more register spilling.

2.2.3.4 Local Memory

Local memory is thread local storage similar to registers and managed by NVCC, but the physical location of local memory is in the global memory space. The 'local' label refers to its logical scope, not its physical location. Local memory is used for thread local storage during the execution of a kernel. Automatic variables that the compiler is likely to place in local memory are:

- Arrays for which it cannot determine that they are indexed with constant quantities,
- Large structures or arrays that would consume too much register space,
- Any variable if the kernel uses more registers than available, that is register spilling.

Because the local memory space resides in device memory, local memory accesses have the same latency and bandwidth as global memory accesses and are subject to the same requirements for memory coalescing as described in *Coalesced Global Memory Access*. Local memory is however organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address, such as the same index in an array variable or the same member in a structure variable.

2.2.3.5 Constant Memory

Constant memory has a grid scope and is accessible for the lifetime of the application. The constant memory resides on the device and is read-only to the kernel. As such, it must be declared and initialized on the host with the `__constant__` specifier, outside any function.

The `__constant__` memory space specifier declares a variable that:

- Resides in constant memory space,
- Has the lifetime of the CUDA context in which it is created,
- Has a distinct object per device,
- Is accessible from all the threads within the grid and from the host through the runtime library (`cudaGetSymbolAddress()` / `cudaGetSymbolSize()` / `cudaMemcpyToSymbol()` / `cudaMemcpyFromSymbol()`).

The total amount of constant memory can be queried with the `totalConstMem` device property element.

Constant memory is useful for small amounts of data that each thread will use in a read-only fashion. Constant memory is small relative to other memories, typically 64KB per device.

An example snippet of declaring and using constant memory follows.

```
// In your .cu file
__constant__ float coeffs[4];

__global__ void compute(float *out) {
    int idx = threadIdx.x;
    out[idx] = coeffs[0] * idx + coeffs[1];
}

// In your host code
float h_coeffs[4] = {1.0f, 2.0f, 3.0f, 4.0f};
cudaMemcpyToSymbol(coeffs, h_coeffs, sizeof(h_coeffs));
compute<<<1, 10>>>(device_out);
```

2.2.3.6 Caches

GPU devices have a multi-level cache structure which includes L2 and L1 caches.

The L2 cache is located on the device and is shared among all the SMs. The size of the L2 cache can be queried with the `l2CacheSize` device property element from the function `cudaGetDeviceProperties`.

As described above in *Shared Memory*, L1 cache is physically located on each SM and is the same physical space used by shared memory. If no shared memory is utilized by a kernel, the entire physical space will be utilized by the L1 cache.

The L2 and L1 caches can be controlled via functions that allow the developer to specify various caching behaviors. The details of these functions are found in *Configuring L1/Shared Memory Balance*, *L2 Cache Control*, and *Low-Level Load and Store Functions*.

If these hints are not used, the compiler and runtime will do their best to utilize the caches efficiently.

2.2.3.7 Texture and Surface Memory

Note

Some older CUDA code may use texture memory because, in older NVIDIA GPUs, doing so provided performance benefits in some scenarios. On all currently supported GPUs, these scenarios may be handled using direct load and store instructions, and use of texture and surface memory instructions no longer provides any performance benefit.

A GPU may have specialized instructions for loading data from an image to be used as textures in 3D rendering. CUDA exposes these instructions and the machinery to use them in the texture object API and the surface object API.

Texture and Surface memory are not discussed further in this guide as there is no advantage to using them in CUDA on any currently supported NVIDIA GPU. CUDA developers should feel free to ignore these APIs. For developers working on existing code bases which still use them, explanations of these APIs can still be found in the legacy CUDA C++ Programming Guide.

2.2.3.8 Distributed Shared Memory

Thread Block Clusters introduced in compute capability 9.0 and facilitated by *Cooperative Groups*, provide the ability for threads in a thread block cluster to access shared memory of all the participating thread blocks in that cluster. This partitioned shared memory is called *Distributed Shared Memory*, and the corresponding address space is called Distributed Shared Memory address space. Threads that belong to a thread block cluster can read, write or perform atomics in the distributed address space, regardless whether the address belongs to the local thread block or a remote thread block. Whether a kernel uses distributed shared memory or not, the shared memory size specifications, static or dynamic is still per thread block. The size of distributed shared memory is just the number of thread blocks per cluster multiplied by the size of shared memory per thread block.

Accessing data in distributed shared memory requires all the thread blocks to exist. A user can guarantee that all thread blocks have started executing using `cluster.sync()` from *class cluster_group*. The user also needs to ensure that all distributed shared memory operations happen before the exit of a thread block, e.g., if a remote thread block is trying to read a given thread block's shared memory, the program needs to ensure that the shared memory read by the remote thread block is completed before it can exit.

Let's look at a simple histogram computation and how to optimize it on the GPU using thread block cluster. A standard way of computing histograms is to perform the computation in the shared memory of each thread block and then perform global memory atomics. A limitation of this approach is the shared memory capacity. Once the histogram bins no longer fit in the shared memory, a user needs to directly compute histograms and hence the atomics in the global memory. With distributed shared memory, CUDA provides an intermediate step, where depending on the histogram bins size, the histogram can be computed in shared memory, distributed shared memory or global memory directly.

The CUDA kernel example below shows how to compute histograms in shared memory or distributed shared memory, depending on the number of histogram bins.

```
#include <cooperative_groups.h>

// Distributed Shared memory histogram kernel
__global__ void clusterHist_kernel(int *bins, const int nbins, const int bins_
↪per_block, const int *__restrict__ input,
                                   size_t array_size)
```

(continues on next page)

(continued from previous page)

```
{
  extern __shared__ int smem[];
  namespace cg = cooperative_groups;
  int tid = cg::this_grid().thread_rank();

  // Cluster initialization, size and calculating local bin offsets.
  cg::cluster_group cluster = cg::this_cluster();
  unsigned int clusterBlockRank = cluster.block_rank();
  int cluster_size = cluster.dim_blocks().x;

  for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x)
  {
    smem[i] = 0; //Initialize shared memory histogram to zeros
  }

  // cluster synchronization ensures that shared memory is initialized to zero
↪in
  // all thread blocks in the cluster. It also ensures that all thread blocks
  // have started executing and they exist concurrently.
  cluster.sync();

  for (int i = tid; i < array_size; i += blockDim.x * gridDim.x)
  {
    int ldata = input[i];

    //Find the right histogram bin.
    int binid = ldata;
    if (ldata < 0)
      binid = 0;
    else if (ldata >= nbins)
      binid = nbins - 1;

    //Find destination block rank and offset for computing
    //distributed shared memory histogram
    int dst_block_rank = (int)(binid / bins_per_block);
    int dst_offset = binid % bins_per_block;

    //Pointer to target block shared memory
    int *dst_smem = cluster.map_shared_rank(smem, dst_block_rank);

    //Perform atomic update of the histogram bin
    atomicAdd(dst_smem + dst_offset, 1);
  }

  // cluster synchronization is required to ensure all distributed shared
  // memory operations are completed and no thread block exits while
  // other thread blocks are still accessing distributed shared memory
  cluster.sync();

  // Perform global memory histogram, using the local distributed memory
↪histogram
  int *lbins = bins + cluster.block_rank() * bins_per_block;
```

(continues on next page)

(continued from previous page)

```
  for (int i = threadIdx.x; i < bins_per_block; i += blockDim.x)
  {
    atomicAdd(&lbins[i], smem[i]);
  }
}
```

The above kernel can be launched at runtime with a cluster size depending on the amount of distributed shared memory required. If the histogram is small enough to fit in shared memory of just one block, the user can launch the kernel with cluster size 1. The code snippet below shows how to launch a cluster kernel dynamically based on shared memory requirements.

```
// Launch via extensible launch
{
  cudaLaunchConfig_t config = {0};
  config.gridDim = array_size / threads_per_block;
  config.blockDim = threads_per_block;

  // cluster_size depends on the histogram size.
  // ( cluster_size == 1 ) implies no distributed shared memory, just thread
↪block local shared memory
  int cluster_size = 2; // size 2 is an example here
  int nbins_per_block = nbins / cluster_size;

  //dynamic shared memory size is per block.
  //Distributed shared memory size =  cluster_size * nbins_per_block *
↪sizeof(int)
  config.dynamicSmemBytes = nbins_per_block * sizeof(int);

  CUDA_CHECK(::cudaFuncSetAttribute((void *)clusterHist_kernel,
↪cudaFuncAttributeMaxDynamicSharedMemorySize, config.dynamicSmemBytes));

  cudaLaunchAttribute attribute[1];
  attribute[0].id = cudaLaunchAttributeClusterDimension;
  attribute[0].val.clusterDim.x = cluster_size;
  attribute[0].val.clusterDim.y = 1;
  attribute[0].val.clusterDim.z = 1;

  config.numAttrs = 1;
  config.attrs = attribute;

  cudaLaunchKernelEx(&config, clusterHist_kernel, bins, nbins, nbins_per_
↪block, input, array_size);
}
```

2.2.4. Memory Performance

Ensuring proper memory usage is key to achieving high performance in CUDA kernels. This section discusses some general principles and examples for achieving high memory throughput in CUDA kernels.

2.2.4.1 Coalesced Global Memory Access

Global memory is accessed via 32-byte memory transactions. When a CUDA thread requests a word of data from global memory, the relevant warp coalesces the memory requests from all the threads in that warp into the number of memory transactions necessary to satisfy the request, depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. For example, if a thread requests a 4-byte word, the actual memory transaction the warp will generate to global memory will be 32 bytes in total. To use the memory system most efficiently, the warp should use all the memory that is fetched in a single memory transaction. That is, if a thread is requesting a 4-byte word from global memory, and the transaction size is 32 bytes, if other threads in that warp can use other 4-byte words of data from that 32-byte request, this will result in the most efficient use of the memory system.

As a simple example, if consecutive threads in the warp request consecutive 4-byte words in memory, then the warp will request 128 bytes of memory total, and this 128 bytes required will be fetched in four 32-byte memory transactions. This results in 100% utilization of the memory system. That is, 100% of the memory traffic is utilized by the warp. Figure 10 illustrates this example of perfectly coalesced memory access.

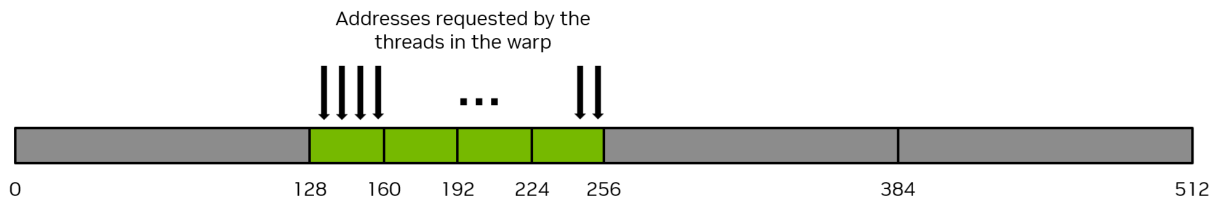


Figure 10: Coalesced memory access

Conversely, the pathologically worst case scenario is when consecutive threads access data elements that are 32 bytes or more apart from each other in memory. In this case, the warp will be forced to issue a 32-byte memory transaction for each thread, and the total number of bytes of memory traffic will be 32 bytes times 32 threads/warp = 1024 bytes. However, the amount of memory used will be 128 bytes only (4 bytes for each thread in the warp), so the memory utilization will only be 128 / 1024 = 12.5%. This is a very inefficient use of the memory system. Figure 11 illustrates this example of uncoalesced memory access.

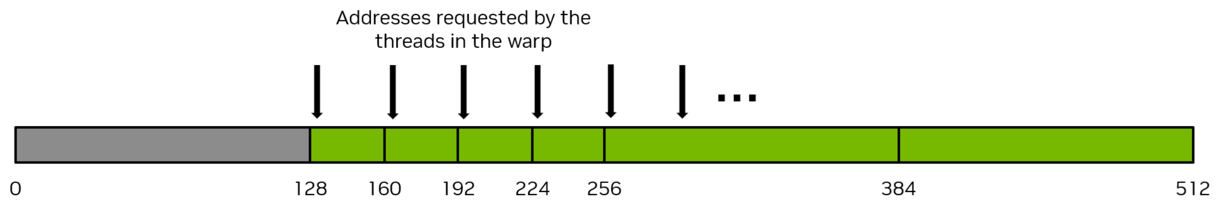


Figure 11: Uncoalesced memory access

The most straightforward way to achieve coalesced memory access is for consecutive threads to access consecutive elements in memory. For example, for a kernel launched with 1d thread blocks, the following `VecAdd` kernel will achieve coalesced memory access. Notice how thread `workIndex` accesses the three arrays, and consecutive threads (indicated by consecutive values of `workIndex`) access consecutive elements in the arrays.

```
__global__ void vecAdd(float* A, float* B, float* C, int vectorLength)
{
    int workIndex = threadIdx.x + blockIdx.x*blockDim.x;
    if(workIndex < vectorLength)
    {
        C[workIndex] = A[workIndex] + B[workIndex];
```

There is no requirement that consecutive threads access consecutive elements of memory to achieve coalesced memory access, it is merely the typical way coalescing is achieved. Coalesced memory access occurs provided all the threads in the warp access elements from the same 32-byte segments of memory in some linear or permuted way. Stated another way, the best way to achieve coalesced memory access is to maximize the ratio of bytes used to bytes transferred.

Note

Ensuring proper coalescing of global memory accesses is one of the most important performance considerations for writing performant CUDA kernels. It is imperative that applications use the memory system as efficiently as possible.

2.2.4.1.1 Matrix Transpose Example Using Global Memory

As a simple example, consider an out-of-place matrix transpose kernel that transposes a 32 bit float square matrix of size N x N, from matrix a to matrix c. This example uses a 2d grid, and assumes a launch of 2d thread blocks of size 32 x 32 threads, that is, `blockDim.x = 32` and `blockDim.y = 32`, so each 2d thread block will operate on a 32 x 32 tile of the matrix. Each thread operates on a unique element of the matrix, so no explicit synchronization of threads is necessary. Figure 12 illustrates this matrix transpose operation. The kernel source code follows the figure.

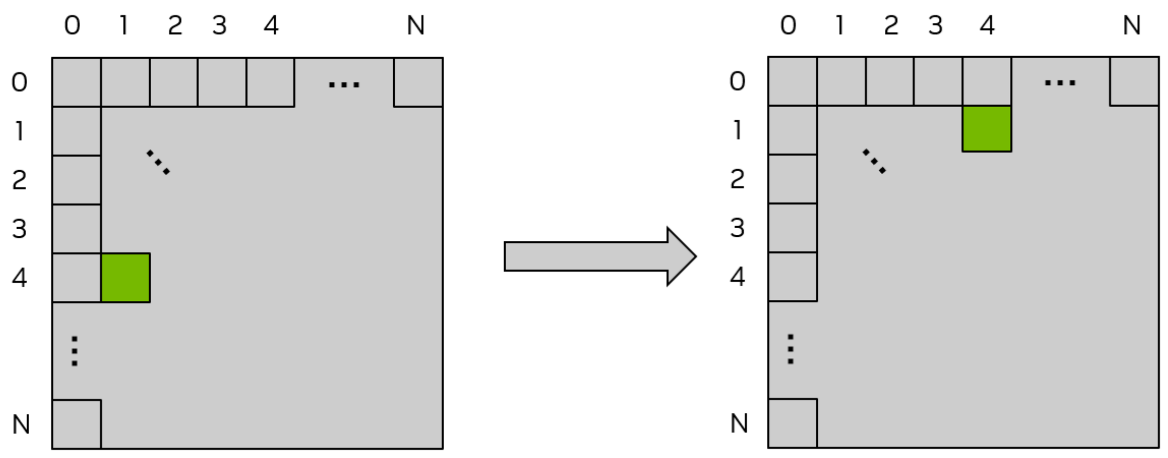


Figure 12: Matrix Transpose using Global memory

The labels on the top and left of each matrix are the 2d thread block indices and also can be considered the tile indices, where each small square indicates a tile of the matrix that will be operated on by a 2d thread block. In this example, the tile size is 32 x 32 elements, so each of the small squares represents a 32 x 32 tile of the matrix. The green shaded square shows the location of an example tile before and after the transpose operation.

```
/* macro to index a 1D memory array with 2D indices in row-major order */
/* ld is the leading dimension, i.e. the number of columns in the matrix     */

#define INDX( row, col, ld ) ( ( (row) * (ld) ) + (col) )

/* CUDA kernel for naive matrix transpose */

__global__ void naive_cuda_transpose(int m, float *a, float *c )
{
    int myCol = blockDim.x * blockIdx.x + threadIdx.x;
    int myRow = blockDim.y * blockIdx.y + threadIdx.y;

    if( myRow < m && myCol < m )
    {
        c[INDX( myCol, myRow, m )] = a[INDX( myRow, myCol, m )];
    } /* end if */
    return;
} /* end naive_cuda_transpose */
```

To determine whether this kernel is achieving coalesced memory access one needs to determine whether consecutive threads are accessing consecutive elements of memory. In a 2d thread block, the x index moves the fastest, so consecutive values of `threadIdx.x` should be accessing consecutive elements of memory. `threadIdx.x` appears in `myCol`, and one can observe that when `myCol` is the second argument to the `INDX` macro, consecutive threads are reading consecutive values of a, so the read of a is perfectly coalesced.

However, the writing of c is not coalesced, because consecutive values of `threadIdx.x` (again examine `myCol`) are writing elements to c that are `ld` (leading dimension) elements apart from each other. This is observed because now `myCol` is the first argument to the `INDX` macro, and as the first argument to `INDX` increments by 1, the memory location changes by `ld`. When `ld` is larger than 32 (which occurs whenever the matrix sizes are larger than 32), this is equivalent to the pathological case shown in Figure 11.

To alleviate these uncoalesced writes, the use of shared memory can be employed, which will be described in the next section.

2.2.4.2 Shared Memory Access Patterns

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle.

When multiple threads in the same warp attempt to access different elements in the same bank, a bank conflict occurs. In this case, the access to the data in that bank will be serialized until the data in that bank has been obtained by all the threads that have requested it. This serialization of access results in a performance penalty.

The two exceptions to this scenario happen when multiple threads in the same warp are accessing (either reading or writing) the same shared memory location. For read accesses, the word is broadcast to the requesting threads. For write accesses, each shared memory address is written by only one of the threads (which thread performs the write is undefined).

Figure 13 shows some examples of strided access. The red box inside the bank indicates a unique location in shared memory.

Figure 14 shows some examples of memory read accesses that involve the broadcast mechanism. The red box inside the bank indicates a unique location in shared memory. If multiple arrows point to the

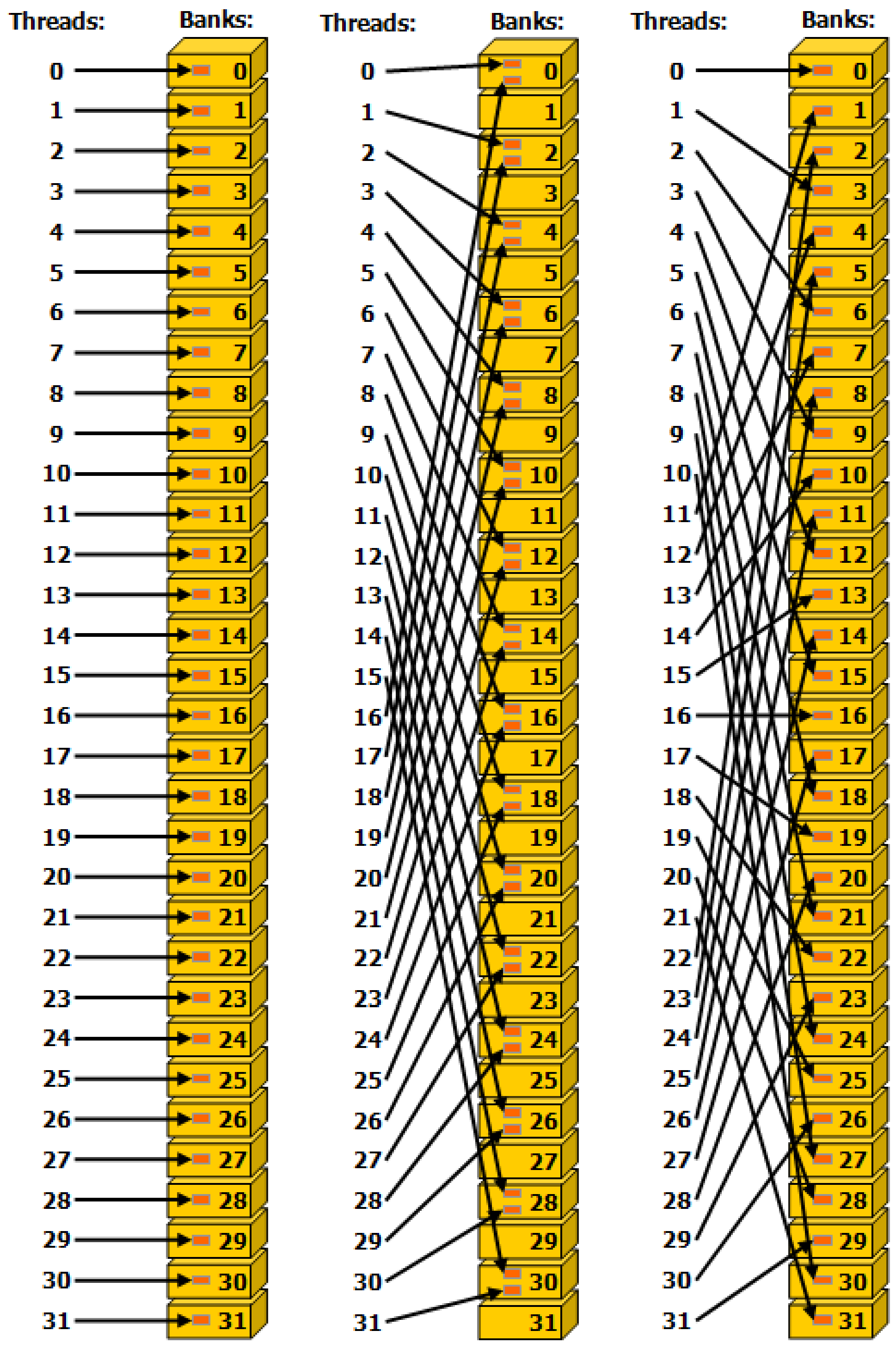


Figure 13: Strided Shared Memory Accesses in 32 bit bank size mode.

Left

Linear addressing with a stride of one 32-bit word (no bank conflict).

Middle

Linear addressing with a stride of two 32-bit words (two-way bank conflict).

Right

Linear addressing with a stride of three 32-bit words (no bank conflict).

same location, the data is broadcast to all threads that requested it.

Note

Avoiding bank conflicts is an important performance consideration for writing performant CUDA kernels that use shared memory.

2.2.4.2.1 Matrix Transpose Example Using Shared Memory

In the previous example *Matrix Transpose Example Using Global Memory*, a naive implementation of matrix transpose was illustrated that was functionally correct, but not optimized for efficient use of global memory because the write of the c matrix was not coalesced properly. In this example, shared memory will be treated as a user-managed cache to stage loads and stores from global memory, resulting in coalesced global memory access of both reads and writes.

Example

```
/* definitions of thread block size in X and Y directions */

#define THREADS_PER_BLOCK_X 32
#define THREADS_PER_BLOCK_Y 32

/* macro to index a 1D memory array with 2D indices in row-major order */
/* ld is the leading dimension, i.e. the number of columns in the matrix     */

#define INDX( row, col, ld ) ( ( (row) * (ld) ) + (col) )

/* CUDA kernel for shared memory matrix transpose */

__global__ void smem_cuda_transpose(int m, float *a, float *c )
{

    /* declare a statically allocated shared memory array */

    __shared__ float smemArray[THREADS_PER_BLOCK_X][THREADS_PER_BLOCK_Y];

    /* determine my row tile and column tile index */

    const int tileCol = blockDim.x * blockIdx.x;
    const int tileRow = blockDim.y * blockIdx.y;

    /* read from global memory into shared memory array */
    smemArray[threadIdx.x][threadIdx.y] = a[INDX( tileRow + threadIdx.y,
↪tileCol + threadIdx.x, m )];

    /* synchronize the threads in the thread block */
    __syncthreads();

    /* write the result from shared memory to global memory */
    c[INDX( tileCol + threadIdx.y, tileRow + threadIdx.x, m )] =
↪smemArray[threadIdx.y][threadIdx.x];
    return;
```

(continues on next page)

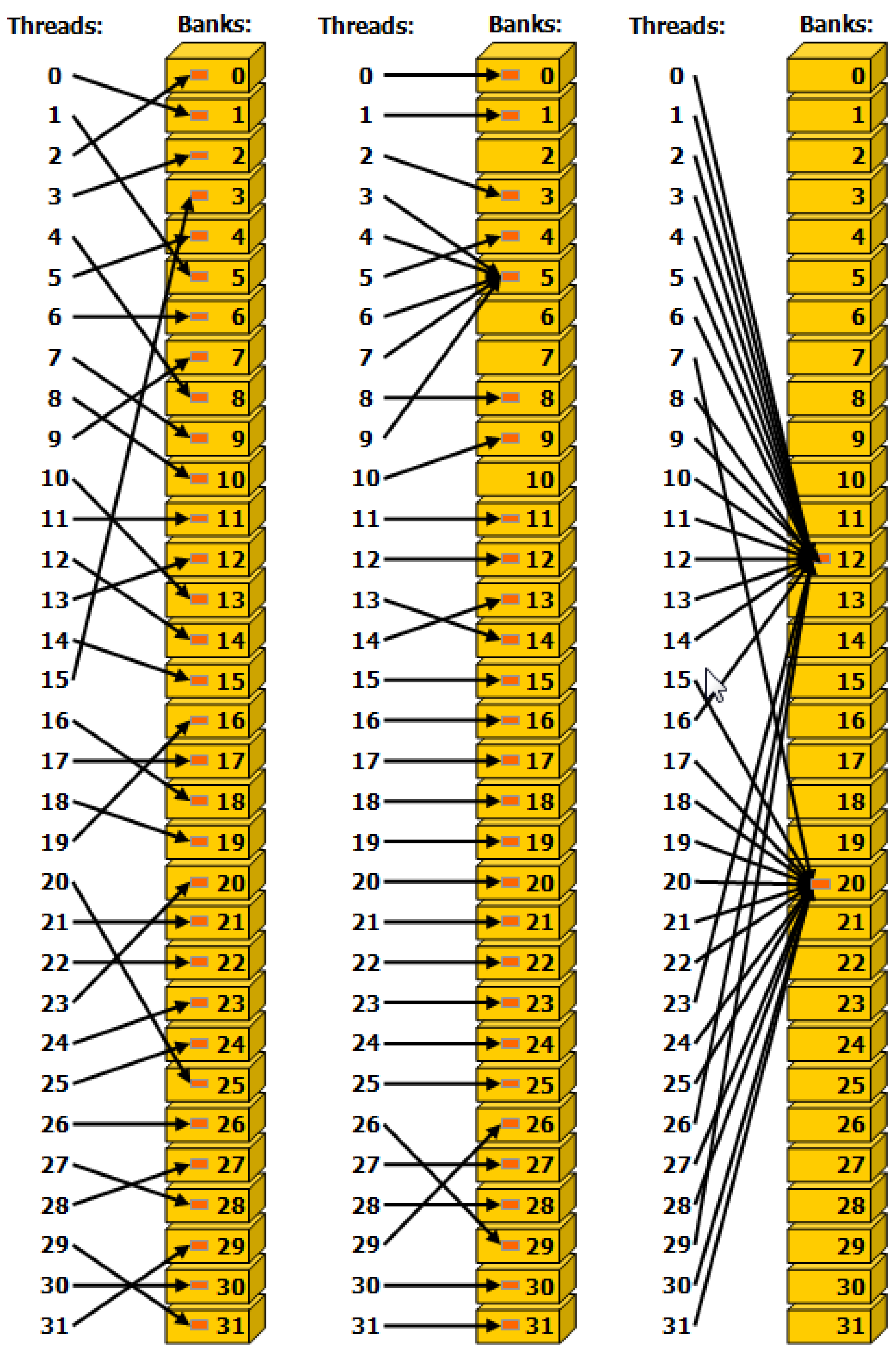


Figure 14: Irregular Shared Memory Accesses.

Left

Conflict-free access via random permutation.

Middle

Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.

Right

Conflict-free broadcast access (threads access the same word within a bank).

(continued from previous page)

```

} /* end smem_cuda_transpose */
```

Example with array checks

```
/* definitions of thread block size in X and Y directions */

#define THREADS_PER_BLOCK_X 32
#define THREADS_PER_BLOCK_Y 32

/* macro to index a 1D memory array with 2D indices in column-major order */
/* ld is the leading dimension, i.e. the number of rows in the matrix     */

#define INDX( row, col, ld ) ( ( (col) * (ld) ) + (row) )

/* CUDA kernel for shared memory matrix transpose */

__global__ void smem_cuda_transpose(int m,
                                    float *a,
                                    float *c )
{

    /* declare a statically allocated shared memory array */

    __shared__ float smemArray[THREADS_PER_BLOCK_X][THREADS_PER_BLOCK_Y];

    /* determine my row and column indices for the error checking code */

    const int myRow = blockDim.x * blockIdx.x + threadIdx.x;
    const int myCol = blockDim.y * blockIdx.y + threadIdx.y;

    /* determine my row tile and column tile index */

    const int tileX = blockDim.x * blockIdx.x;
    const int tileY = blockDim.y * blockIdx.y;

    if( myRow < m && myCol < m )
    {
        /* read from global memory into shared memory array */
        smemArray[threadIdx.x][threadIdx.y] = a[INDX( tileX + threadIdx.x,
↪tileY + threadIdx.y, m )];
    } /* end if */

    /* synchronize the threads in the thread block */
    __syncthreads();

    if( myRow < m && myCol < m )
    {
        /* write the result from shared memory to global memory */
        c[INDX( tileY + threadIdx.x, tileX + threadIdx.y, m )] =
↪smemArray[threadIdx.y][threadIdx.x];
```

(continues on next page)

(continued from previous page)

```
    } /* end if */
    return;

} /* end smem_cuda_transpose */
```

The fundamental performance optimization illustrated in this example is to ensure that when accessing global memory, the memory accesses are coalesced properly. Prior to the execution of the copy, each thread computes its `tileRow` and `tileCol` indices. These are the indices for the specific tile that will be operated on, and these tile indices are based on which thread block is executing. Each thread in the same thread block has the same `tileRow` and `tileCol` values, so it can be thought of as the starting position of the tile that this specific thread block will operate on.

The kernel then proceeds with each thread block copying a 32 x 32 tile of the matrix from global memory to shared memory with the following statement. Since the size of a warp is 32 threads, this copy operation will be executed by 32 warps, with no guaranteed order between the warps.

```
smemArray[threadIdx.x][threadIdx.y] = a[INDX( tileRow + threadIdx.y, tileCol
↪+ threadIdx.x, m )];
```

Note that because `threadIdx.x` appears in the second argument to `INDX`, consecutive threads are accessing consecutive elements in memory, and the read of a is perfectly coalesced.

The next step in the kernel is the call to the `__syncthreads()` function. This ensures that all threads in the thread block have completed their execution of the previous code before proceeding and therefore that the write of a into shared memory is completed before the next step. This is critically important because the next step will involve threads reading from shared memory. Without the `__syncthreads()` call, the read of a into shared memory would not be guaranteed to be completed by all the warps in the thread block before some warps advance further in the code.

At this point in the kernel, for each thread block, the `smemArray` has a 32 x 32 tile of the matrix, arranged in the same order as the original matrix. To ensure that the elements within the tile are transposed properly, `threadIdx.x` and `threadIdx.y` are swapped when they read `smemArray`. To ensure that the overall tile is placed in the correct place in c, the `tileRow` and `tileCol` indices are also swapped when they write to c. To ensure proper coalescing, `threadIdx.x` is used in the second argument to `INDX`, as shown by the statement below.

```
c[INDX( tileCol + threadIdx.y, tileRow + threadIdx.x, m )] =
↪smemArray[threadIdx.y][threadIdx.x];
```

This kernel illustrates two common uses of shared memory.

- Shared memory is used to stage data from global memory to ensure that reads from and writes to global memory are both coalesced properly.
- Shared memory is used to allow threads in the same thread block to share data among themselves.

2.2.4.2.2 Shared Memory Bank Conflicts

In Section 2.2.4.2, the bank structure of shared memory was described. In the previous matrix transpose example, the proper coalesced memory access to/from global memory was achieved, but no consideration was given to whether shared memory bank conflicts were present. Consider the following 2d shared memory declaration,

```
__shared__ float smemArray[32][32];
```

Since a warp is 32 threads, each thread in the same warp will have a fixed value for `threadIdx.y` and will have `0 <= threadIdx.x < 32`.

The left panel of Figure 15 illustrates the situation when the threads in a warp access the data in a column of `smemArray`. Warp 0 is accessing memory locations `smemArray[0][0]` through `smemArray[31][0]`. In C++ multi-dimensional array ordering, the last index moves the fastest, so consecutive threads in warp 0 are accessing memory locations that are 32 elements apart. As illustrated in the figure, the colors denote the banks, and this access down the entire column by warp 0 results in a 32-way bank conflict.

The right panel of Figure 15 illustrates the situation when the threads in a warp access the data across a row of `smemArray`. Warp 0 is accessing memory locations `smemArray[0][0]` through `smemArray[0][31]`. In this case, consecutive threads in warp 0 are accessing memory locations that are adjacent. As illustrated in the figure, the colors denote the banks, and this access across the entire row by warp 0 results in no bank conflicts. The ideal scenario is for each thread in a warp to access a shared memory location with a different color.

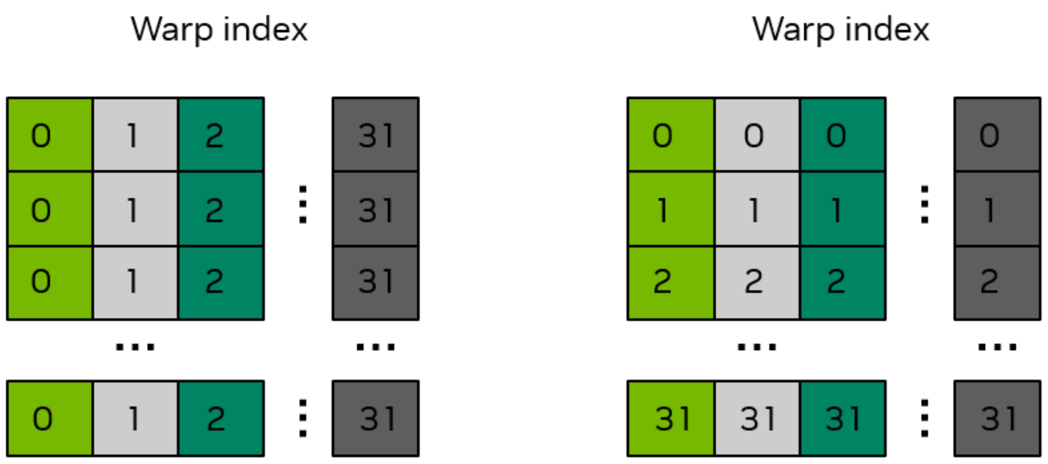


Figure 15: Bank structure in a 32 x 32 shared memory array.
The numbers in the boxes indicate the warp index. The colors indicate which bank is associated with that shared memory location.

Returning to the example from Section 2.2.4.2.1, one can examine the usage of shared memory to determine whether bank conflicts are present. The first usage of shared memory is when data from global memory is stored to shared memory:

```
smemArray[threadIdx.x][threadIdx.y] = a[INDX( tileRow + threadIdx.y, tileCol
↪+ threadIdx.x, m )];
```

Because C++ arrays are stored in row-major order, consecutive threads in the same warp, as indicated by consecutive values of `threadIdx.x`, will access `smemArray` with a stride of 32 elements, because `threadIdx.x` is the first index into the array. This results in a 32-way bank conflict and is illustrated by the left panel of Figure 15.

The second usage of shared memory is when data from shared memory is written back to global memory:

```
c[INDX( tileCol + threadIdx.y, tileRow + threadIdx.x, m )] =
↪smemArray[threadIdx.y][threadIdx.x];
```

In this case, because `threadIdx.x` is the second index into the `smemArray` array, consecutive threads in the same warp will access `smemArray` with a stride of 1 element. This results in no bank conflicts and is illustrated by the right panel of Figure 15.

The matrix transpose kernel as illustrated in Section 2.2.4.2.1 has one access of shared memory that has no bank conflicts and one access that has a 32-way bank conflict. A common fix to avoid bank conflicts is to pad the shared memory by adding one to the column dimension of the array as follows:

```
__shared__ float smemArray[THREADS_PER_BLOCK_X][THREADS_PER_BLOCK_Y+1];
```

This minor adjustment to the declaration of `smemArray` will eliminate the bank conflicts. To illustrate this, consider Figure 16 where the shared memory array has been declared with a size of 32 x 33. One observes that whether the threads in the same warp access the shared memory array down an entire column or across an entire row, the bank conflicts have been eliminated, i.e., the threads in the same warp access locations with different colors.

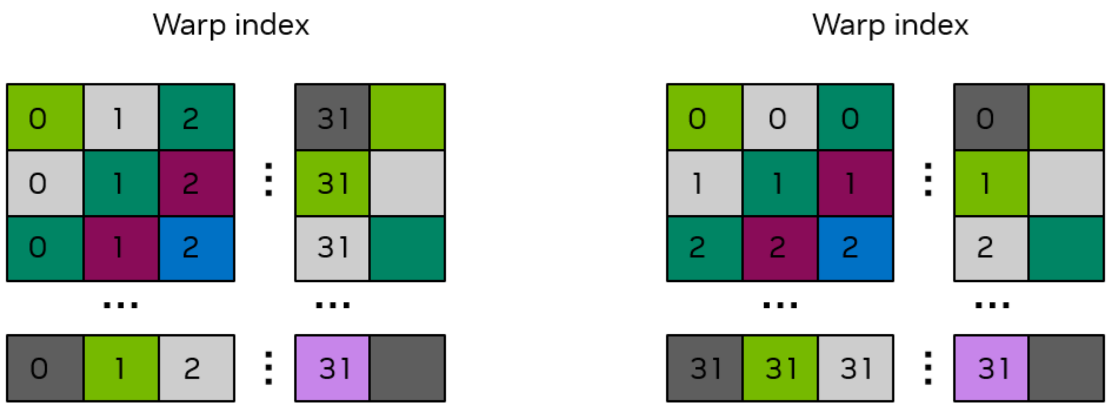


Figure 16: Bank structure in a 32 x 33 shared memory array.
The numbers in the boxes indicate the warp index. The colors indicate which bank is associated with that shared memory location.

2.2.5. Atomics

Performant CUDA kernels rely on expressing as much algorithmic parallelism as possible. The asynchronous nature of GPU kernel execution requires that threads operate as independently as possible. It's not always possible to have complete independence of threads and as we saw in *Shared Memory*, there exists a mechanism for threads in the same thread block to exchange data and synchronize.

On the level of an entire grid there is no such mechanism to synchronize all threads in a grid. There is however a mechanism to provide synchronous access to global memory locations via the use of atomic functions. Atomic functions allow a thread to obtain a lock on a global memory location and perform a read-modify-write operation on that location. No other thread can access the same location while the lock is held. CUDA provides atomics with the same behavior as the C++ standard library atomics as `cuda::std::atomic` and `cuda::std::atomic_ref`. CUDA also provides extended C++

atomics `cuda::atomic` and `cuda::atomic_ref` which allow the user to specify the *thread scope* of the atomic operation. The details of atomic functions are covered in *Atomic Functions*.

An example usage of `cuda::atomic_ref` to perform a device-wide atomic addition is as follows, where `array` is an array of floats, and `result` is a float pointer to a location in global memory which is the location where the sum of the array will be stored.

```
__global__ void sumReduction(int n, float *array, float *result) {
   ...
   tid = threadIdx.x + blockIdx.x * blockDim.x;

   cuda::atomic_ref<float, cuda::thread_scope_device> result_ref(result);
   result_ref.fetch_add(array[tid]);
   ...
}
```

Atomic functions should be used sparingly as they enforce thread synchronization that can impact performance.

2.2.6. Cooperative Groups

Cooperative groups is a software tool available in CUDA C++ that allows applications to define groups of threads which can synchronize with each other, even if that group of threads spans multiple thread blocks, multiple grids on a single GPU, or even across multiple GPUs. The CUDA programming model in general allows threads within a thread block or thread block cluster to synchronize efficiently, but does not provide a mechanism for specifying thread groups smaller than a thread block or cluster. Similarly, the CUDA programming model does not provide mechanisms or guarantees that enable synchronization across thread blocks.

Cooperative groups provide both of these capabilities through software. Cooperative groups allows the application to create thread groups that cross the boundary of thread blocks and clusters, though doing so comes with some semantic limitations and performance implications which are described in detail in the *feature section covering cooperative groups*.

2.2.7. Kernel Launch and Occupancy

When a CUDA kernel is launched, CUDA threads are grouped into thread blocks and a grid based on the execution configuration specified at kernel launch. Once the kernel is launched, the scheduler assigns thread blocks to SMs. The details of which thread blocks are scheduled to execute on which SMs cannot be controlled or queried by the application and no ordering guarantees are made by the scheduler, so programs cannot not rely on a specific scheduling order or scheme for correct execution.

The number of blocks that can be scheduled on an SM depends on the hardware resources a given thread block requires, and the hardware resources available on the SM. When a kernel is first launched, the scheduler begins assigning thread blocks to SMs. As long as SMs have sufficient hardware resources unoccupied by other thread blocks, the scheduler will continue assigning thread blocks to SMs. If at some point no SM has the capacity to accept another thread block, the scheduler will wait until the SMs complete previously assigned thread blocks. Once this happens, SMs are free to accept more work, and the scheduler assigns thread blocks to them. This process continues until all thread blocks have been scheduled and executed.

The `cudaGetDeviceProperties` function allows an application to query the limits of each SM via device properties. Note that there are limits per SM and per thread block.

- `maxBlocksPerMultiProcessor`: The maximum number of resident blocks per SM.

- `sharedMemPerMultiprocessor`: The amount of shared memory available per SM in bytes.
- `regsPerMultiprocessor`: The number of 32-bit registers available per SM.
- `maxThreadsPerMultiProcessor`: The maximum number of resident threads per SM.
- `sharedMemPerBlock`: The maximum amount of shared memory that can be allocated by a thread block in bytes.
- `regsPerBlock`: The maximum number of 32-bit registers that can be allocated by a thread block.
- `maxThreadsPerBlock`: The maximum number of threads per thread block.

The occupancy of a CUDA kernel is the ratio of the number of active warps to the maximum number of active warps supported by the SM. In general, it's a good practice to have occupancy as high as possible which hides latency and increases performance.

To calculate occupancy, one needs to know the resource limits of the SM, which were just described, and one needs to know what resources are required by the CUDA kernel in question. To determine resource usage on a per kernel basis, during program compilation one can use the `--resource-usage` option to `nvcc`, which will show the number of registers and shared memory required by the kernel.

To illustrate, consider a device such as compute capability 10.0 with the device properties enumerated in Table 2.

Table 2: SM Resource Example

Resource	Value
`maxBlocksPerMultiProcessor`	32
`sharedMemPerMultiprocessor`	233472
`regsPerMultiprocessor`	65536
`maxThreadsPerMultiProcessor`	2048
`sharedMemPerBlock`	49152
`regsPerBlock`	65536
`maxThreadsPerBlock`	1024

If a kernel was launched as `testKernel<<<512, 768>>>()`, i.e., 768 threads per block, each SM would only be able to execute 2 thread blocks at a time. The scheduler cannot assign more than 2 thread blocks per SM because the `maxThreadsPerMultiProcessor` is 2048. So the occupancy would be (768 * 2) / 2048, or 75%.

If a kernel was launched as `testKernel<<<512, 32>>>()`, i.e., 32 threads per block, each SM would not run into a limit on `maxThreadsPerMultiProcessor`, but since the `maxBlocksPerMultiProcessor` is 32, the scheduler would only be able to assign 32 thread blocks to each SM. Since the number of threads in the block is 32, the total number of threads resident on the SM would be 32 blocks * 32 threads per block, or 1024 total threads. Since a compute capability 10.0 SM has a maximum value of 2048 resident threads per SM, the occupancy in this case is 1024 / 2048, or 50%.

The same analysis can be done with shared memory. If a kernel uses 100KB of shared memory, for example, the scheduler would only be able to assign 2 thread blocks to each SM, because the third thread block on that SM would require another 100KB of shared memory for a total of 300KB, which is more than the 233472 bytes available per SM.

Threads per block and shared memory usage per block are explicitly controlled by the programmer and can be adjusted to achieve the desired occupancy. The programmer has limited control over register usage as the compiler and runtime will attempt to optimize register usage. However the programmer can specify a maximum number of registers per thread block via the `--maxrregcount` option to `nvcc`. If the kernel needs more registers than this specified amount, the kernel is likely to spill to local memory, which will change the performance characteristics of the kernel. In some cases even though spilling occurs, limiting registers allows more thread blocks to be scheduled which in turn increases occupancy and may result in a net increase in performance.

2.3. Asynchronous Execution

2.3.1. What is Asynchronous Concurrent Execution?

CUDA allows concurrent, or overlapping, execution of multiple tasks, specifically:

- computation on the host
- computation on the device
- memory transfers from the host to the device
- memory transfers from the device to the host
- memory transfers within the memory of a given device
- memory transfers among devices

The concurrency is expressed via an asynchronous interface, where a dispatching function call or kernel launch returns immediately. Asynchronous calls usually return before the dispatched operation has completed and may return before the asynchronous operation has started. The application is then free to perform other tasks at the same time as the originally dispatched operation. When the final results of the initially dispatched operation are needed, the application must perform some form of synchronization to ensure that the operation in question has completed. A typical example of a concurrent execution pattern is the overlapping of host and device memory transfers with computation and thus reducing or eliminating their overhead.

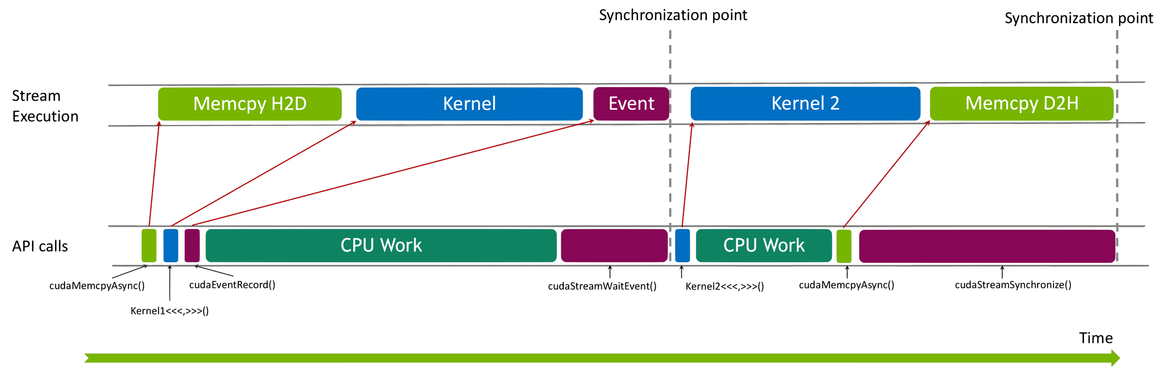


Figure 17: Asynchronous COncurrent Execution with CUDA streams

In general, asynchronous interfaces typically provide three main ways to synchronize with the dispatched operation

- a **blocking approach**, where the application calls a function that blocks, or waits until the operation has completed

- a **non-blocking approach**, or polling approach where the application calls a function that returns immediately and supplies information about the status of the operation
- a **callback approach**, where a pre-registered function is executed when the operation has completed.

While the programming interfaces are asynchronous, the actual ability to carry out various operations concurrently will depend on the version of CUDA and the compute capability of the hardware being used – these details will be left to a later section of this guide (see *Compute Capabilities*).

In *Synchronizing CPU and GPU*, the CUDA runtime function `cudaDeviceSynchronize()` was introduced, which is a blocking call which waits for all previously issued work to complete. The reason the `cudaDeviceSynchronize()` call was needed is because the kernel launch is asynchronous and returns immediately. CUDA provides an API for both blocking and non-blocking approaches to synchronization and even supports the use of host-side callback functions.

The core API components for asynchronous execution in CUDA are **CUDA Streams** and **CUDA Events**. In the rest of this section we will explain how these elements can be used to express asynchronous execution in CUDA.

A related topic is that of **CUDA Graphs**, which allow a graph of asynchronous operations to be defined up front, which can then be executed repeatedly with minimal overhead. We cover CUDA Graphs in a very introductory level in section *2.4.9.2 Introduction to CUDA Graphs with Stream Capture*, and a more comprehensive discussion is provided in section *4.1 CUDA Graphs*.

2.3.2. CUDA Streams

At the most basic level, a CUDA stream is an abstraction which allows the programmer to express a sequence of operations. A stream operates like a work-queue into which programs can add operations, such as memory copies or kernel launches, to be executed in order. Operations at the front of the queue for a given stream are executed and then dequeued allowing the next queued operation to come to the front and to be considered for execution. The order of execution of operations in a stream is sequential and the operations are executed in the order they are enqueued into the stream.

An application may use multiple streams simultaneously. In such cases, the runtime will select a task to execute from the streams that have work available depending on the state of the GPU resources. Streams may be assigned a priority which acts as a hint to the runtime to influence the scheduling, but does not guarantee a specific order of execution.

The API function calls and kernel-launches operating in a stream are asynchronous with respect to the host thread. Applications can synchronize with a stream by waiting for it to be empty of tasks, or they can also synchronize at the device level.

CUDA has a default stream, and operations and kernel launches without a specific stream are queued into this default stream. Code examples which do not specify a stream are using this default stream implicitly. The default stream has some specific semantics which are discussed in subsection *Blocking and non-blocking streams and the default stream*.

2.3.2.1 Creating and Destroying CUDA Streams

CUDA streams can be created using the `cudaStreamCreate()` function. The function call initializes the stream handle which can be used to identify the stream in subsequent function calls.

```
cudaStream_t stream;        // Stream handle
cudaStreamCreate(&stream);  // Create a new stream
```

(continues on next page)

(continued from previous page)

```
// stream based operations ...

cudaStreamDestroy(stream);  // Destroy the stream
```

If the the device is still doing work in stream `stream` when the application calls `cudaStreamDestroy()`, the stream will complete all the work in the stream before being destroyed.

2.3.2.2 Launching Kernels in CUDA Streams

The usual triple-chevron syntax for launching a kernel can also be used to launch a kernel into a specific stream. The stream is specified as an extra parameter to the kernel launch. In the following example the kernel named `kernel` is launched into the stream with handle `stream`, which is of type `cudaStream_t` and has been assumed to have been created previously:

```
kernel<<<grid, block, shared_mem_size, stream>>>(...);
```

The kernel launch is asynchronous and the function call returns immediately. Assuming that the kernel launch is successful, the kernel will execute in the stream `stream` and the application is free to perform other tasks on the CPU or in other streams on the GPU while the kernel is executing.

2.3.2.3 Launching Memory Transfers in CUDA Streams

To launch a memory transfer into a stream, we can use the function `cudaMemcpyAsync()`. This function is similar to the `cudaMemcpy()` function, but it takes an additional parameter specifying the stream to use for the memory transfer. The function call in the code block below copies `size` bytes from the host memory pointed to by `src` to the device memory pointed to by `dst` in the stream `stream`.

```
// Copy `size` bytes from `src` to `dst` in stream `stream`
cudaMemcpyAsync(dst, src, size, cudaMemcpyHostToDevice, stream);
```

Like other asynchronous function calls, this function call returns immediately, whereas the `cudaMemcpy()` function blocks until the memory transfer is complete. In order to access the results of the transfer safely, the application must determine that the operation has completed using some form of synchronization.

Other CUDA memory transfer functions such as `cudaMemcpy2D()` also have asynchronous variants.

Note

In order for memory copies involving CPU memory to be carried out asynchronously, the host buffers must be pinned and page-locked. `cudaMemcpyAsync()` will function correctly if host memory which is not pinned and page-locked is used, but it will revert to a synchronous behavior which will not overlap with other work. This can inhibit the performance benefits of using asynchronous memory transfers. It is recommended programs use `cudaMallocHost()` to allocate buffers which will be used to send or receive data from GPUs.

2.3.2.4 Stream Synchronization

The simplest way to synchronize with a stream is to wait for the stream to be empty of tasks. This can be done in two ways, using the `cudaStreamSynchronize()` function or the `cudaStreamQuery()` function.

The `cudaStreamSynchronize()` function will block until all the work in the stream has completed.

```
// Wait for the stream to be empty of tasks
cudaStreamSynchronize(stream);

// At this point the stream is done
// and we can access the results of stream operations safely
```

If we prefer not to block, but just need a quick check to see if the steam is empty we can use the `cudaStreamQuery()` function.

```
// Have a peek at the stream
// returns cudaSuccess if the stream is empty
// returns cudaErrorNotReady if the stream is not empty
cudaError_t status = cudaStreamQuery(stream);

switch (status) {
    case cudaSuccess:
        // The stream is empty
        std::cout << "The stream is empty" << std::endl;
        break;
    case cudaErrorNotReady:
        // The stream is not empty
        std::cout << "The stream is not empty" << std::endl;
        break;
    default:
        // An error occurred - we should handle this
        break;
};
```

2.3.3. CUDA Events

CUDA events are a mechanism for inserting markers into a CUDA stream. They are essentially like tracer particles that can be used to track the progress of tasks in a stream. Imagine launching two kernels into a stream. Without such tracking events, we would only be able to determine whether the stream is empty or not. If we had an operation that was dependent on the output of the first kernel, we would not be able to start that operation safely until we knew the stream was empty by which time both kernels would have completed.

Using CUDA Events we can do better. By enqueuing an event into a stream directly after the first kernel, but before the second kernel, we can wait for this event to come to the front of the stream. Then, we can safely start our dependent operation knowing that the first kernel has completed, but before the second kernel has started. Using CUDA events in this way can build up a graph of dependencies between operations and streams. This graph analogy translates directly into the later discussion of *CUDA graphs*.

CUDA streams also keep time information which can be used to time kernel launches and memory transfers.

2.3.3.1 Creating and Destroying CUDA Events

CUDA Events can be created and destroyed using the `cudaEventCreate()` and `cudaEventDestroy()` functions.

```
cudaEvent_t event;

// Create the event
cudaEventCreate(&event);

// do some work involving the event

// Once the work is done and the event is no longer needed
// we can destroy the event
cudaEventDestroy(event);
```

The application is responsible for destroying events when they are no longer needed.

2.3.3.2 Inserting Events into CUDA Streams

CUDA Events can be inserted into a stream using the `cudaEventRecord()` function.

```
cudaEvent_t event;
cudaStream_t stream;

// Create the event
cudaEventCreate(&event);

// Insert the event into the stream
cudaEventRecord(event, stream);
```

2.3.3.3 Timing Operations in CUDA Streams

CUDA events can be used to time the execution of various stream operations including kernels. When an event reaches the front of a stream it records a timestamp. By surrounding a kernel in a stream with two events we can get an accurate timing of the duration of the kernel execution as is shown in the code snippet below:

```
cudaStream_t stream;
cudaStreamCreate(&stream);

cudaEvent_t start;
cudaEvent_t stop;

// create the events
cudaEventCreate(&start);
cudaEventCreate(&stop);

 // record the start event
cudaEventRecord(start, stream);

// launch the kernel
kernel<<<grid, block, 0, stream>>>(...);
```

(continues on next page)

(continued from previous page)

```
// record the stop event
cudaEventRecord(stop, stream);

// wait for the stream to complete
// both events will have been triggered
cudaStreamSynchronize(stream);

// get the timing
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
std::cout << "Kernel execution time: " << elapsedTime << " ms" << std::endl;

// clean up
cudaEventDestroy(start);
cudaEventDestroy(stop);
cudaStreamDestroy(stream);
```

2.3.3.4 Checking the Status of CUDA Events

Like in the case of checking the status of streams, we can check the status of events in either a blocking or a non-blocking way.

The `cudaEventSynchronize()` function will block until the event has completed. In the code snippet below we launch a kernel into a stream, followed by an event and then by a second kernel. We can use the `cudaEventSynchronize()` function to wait for the event after the first kernel to complete and in principle launch a dependent task immediately, potentially before *kernel2* finishes.

```
cudaEvent_t event;
cudaStream_t stream;

// create the stream
cudaStreamCreate(&stream);

// create the event
cudaEventCreate(&event);

// launch a kernel into the stream
kernel<<<grid, block, 0, stream>>>(...);

// Record the event
cudaEventRecord(event, stream);

// launch a kernel into the stream
kernel2<<<grid, block, 0, stream>>>(...);

// Wait for the event to complete
// Kernel 1 will be  guaranteed to have completed
// and we can launch the dependent task.
cudaEventSynchronize(event);
dependentCPUtask();

// Wait for the stream to be empty
```

(continues on next page)

(continued from previous page)

```
// Kernel 2 is guaranteed to have completed
cudaStreamSynchronize(stream);

// destroy the event
cudaEventDestroy(event);

// destroy the stream
cudaStreamDestroy(stream);
```

CUDA Events can be checked for completion in a non-blocking way using the `cudaEventQuery()` function. In the example below we launch 2 kernels into a stream. The first kernel, *kernel1* generates some data which we would like to copy to the host, however we also have some CPU side work to do. In the code below, we enqueue *kernel1* followed by an event (*event*) and then *kernel2* into stream *stream1*. We then go into a CPU work loop, but occasionally take a peek to see if the event has completed indicating that *kernel1* is done. If so, we launch a host to device copy into stream *stream2*. This approach allows the overlap of the CPU work with the GPU kernel execution and the device to host copy.

```
cudaEvent_t event;
cudaStream_t stream1;
cudaStream_t stream2;

size_t size = LARGE_NUMBER;
float *d_data;

// Create some data
cudaMalloc(&d_data, size);
float *h_data = (float *)malloc(size);

// create the streams
cudaStreamCreate(&stream1);   // Processing stream
cudaStreamCreate(&stream2);   // Copying stream
bool copyStarted = false;

//  create the event
cudaEventCreate(&event);

// launch kernel1 into the stream
kernel1<<<grid, block, 0, stream1>>>(d_data, size);
// enqueue an event following kernel1
cudaEventRecord(event, stream1);

// launch kernel2 into the stream
kernel2<<<grid, block, 0, stream1>>>();

// while the kernels are running do some work on the CPU
// but check if kernel1 has completed because then we will start
// a device to host copy in stream2
while ( not allCPUWorkDone() || not copyStarted ) {
    doNextChunkOfCPUWork();

    // peek to see if kernel 1 has completed
```

(continues on next page)

(continued from previous page)

```
    // if so enqueue a non-blocking copy into stream2
    if ( not copyStarted ) {
        if( cudaEventQuery(event) == cudaSuccess ) {
            cudaMemcpyAsync(h_data, d_data, size, cudaMemcpyDeviceToHost,
↪stream2);
            copyStarted = true;
        }
    }
}

// wait for both streams to be done
cudaStreamSynchronize(stream1);
cudaStreamSynchronize(stream2);

// destroy the event
cudaEventDestroy(event);

// destroy the streams and free the data
cudaStreamDestroy(stream1);
cudaStreamDestroy(stream2);
cudaFree(d_data);
free(h_data);
```

2.3.4. Callback Functions from Streams

CUDA provides a mechanism for launching functions on the host from within a stream. There are currently two functions available for this purpose: `cudaLaunchHostFunc()` and `cudaAddCallback()`. However, `cudaAddCallback()` is slated for deprecation, so applications should use `cudaLaunchHostFunc()`.

Using `cudaLaunchHostFunc()`

The signature of the `cudaLaunchHostFunc()` function is as follows:

```
cudaError_t cudaLaunchHostFunc(cudaStream_t stream, void (*func)(void *),
↪void *data);
```

where

- `stream`: The stream to launch the callback function into.
- `func`: The callback function to launch.
- `data`: A pointer to the data to pass to the callback function.

The host function itself is a simple C function with the signature:

```
void hostFunction(void *data);
```

with the `data` parameter pointing to a user defined data structure which the function can interpret. There are some caveats to keep in mind when using callback functions like this. In particular, the host function may not call any CUDA APIs.

For the purposes of being used with unified memory, the following execution guarantees are provided: - The stream is considered idle for the duration of the function's execution. Thus, for example, the function may always use memory attached to the stream it was enqueued in. - The start of execution of the function has the same effect as synchronizing an event recorded in the same stream immediately prior to the function. It thus synchronizes streams which have been "joined" prior to the function. - Adding device work to any stream does not have the effect of making the stream active until all preceding host functions and stream callbacks have executed. Thus, for example, a function might use global attached memory even if work has been added to another stream, if the work has been ordered behind the function call with an event. - Completion of the function does not cause a stream to become active except as described above. The stream will remain idle if no device work follows the function, and will remain idle across consecutive host functions or stream callbacks without device work in between. Thus, for example, stream synchronization can be done by signaling from a host function at the end of the stream.

2.3.4.1 Using `cudaStreamAddCallback()`

Note

The `cudaStreamAddCallback()` function is slated for deprecation and removal and is discussed here for completeness and because it may still appear in existing code. Applications should use or switch to using `cudaLaunchHostFunc()`.

The signature of the `cudaStreamAddCallback()` function is as follows:

```
cudaError_t cudaStreamAddCallback(cudaStream_t stream, cudaStreamCallback_t
↪callback, void* userData, unsigned int flags);
```

where

- `stream`: The stream to launch the callback function into.
- `callback`: The callback function to launch.
- `userData`: A pointer to the data to pass to the callback function.
- `flags`: Currently, this parameter must be 0 for future compatibility.

The signature of the `callback` function is a little different from the case when we used the `cudaLaunchHostFunc()` function. In this case the callback function is a C function with the signature:

```
void callbackFunction(cudaStream_t stream, cudaError_t status, void
↪*userData);
```

where the function is now passed

- `stream`: The stream handle from which the callback function was launched.
- `status`: The status of the stream operation that triggered the callback.
- `userData`: A pointer to the data that was passed to the callback function.

In particular the `status` parameter will contain the current error status of the stream, which may have been set by previous operations. Similarly to the `cudaLaunchHostFunc()` func case, the stream will not be active and advance to tasks until the host-function has completed, and no CUDA functions may be called from within the callback function.

2.3.4.2 Asynchronous Error Handling

In a cuda stream, errors may originate from any operation in the stream, including for kernel launches and memory transfers. These errors may not be propagated back to the user at run-time until the stream is synchronized, for example, by waiting for an event or calling `cudaStreamSynchronize()`. There are two ways to find out about errors which may have occurred in a stream.

- Using the function `cudaGetLastError()` - this function returns and clears the last error encountered in any stream in the current context. An immediate second call to cudaGetLastError() would return `cudaSuccess` if no other error occurred between the two calls.
- Using the function `cudaPeekAtLastError()` - this function returns the last error in the current context, but does not clear it.

Both of these functions return the error as a value of type `cudaError_t`. Printable names names of the errors can be generated using the functions *cudaGetErrorName()* and *cudaGetErrorString()*.

An example of using these functions is shown below:

Listing 1: Example of using cudaGetLastError() and cudaPeekAtLastError()

```
// Some work occurs in streams.
cudaStreamSynchronize(stream);

// Look at the last error but do not clear it
cudaError_t err = cudaPeekAtLastError();
if (err != cudaSuccess) {
    printf("Error with name: %s\n", cudaGetErrorName(err));
    printf("Error description: %s\n", cudaGetErrorString(err));
}

// Look at the last error and clear it
cudaError_t err2 = cudaGetLastError();
if (err2 != cudaSuccess) {
    printf("Error with name: %s\n", cudaGetErrorName(err2));
    printf("Error description: %s\n", cudaGetErrorString(err2));
}

if (err2 != err) {
    printf("As expected, cudaPeekAtLastError() did not clear the error\n");
}

// Check again
cudaError_t err3 = cudaGetLastError();
if (err3 == cudaSuccess) {
    printf("As expected, cudaGetLastError() cleared the error\n");
}
```

Tip

When an error appears at a synchronization, especially in a stream with many operations, it is often difficult to pinpoint exactly where in the stream the error may have occurred. To debug such a situation a useful trick may be to set the environment variable `CUDA_LAUNCH_BLOCKING=1` and then run the application. The effect of this environment variable is to synchronize after every single kernel launch. This can aid in tracking down which kernel, or transfer caused the error. Synchronization can be expensive; applications may run substantially slower when this environment variable is set.

2.3.5. CUDA Stream Ordering

Now that we have discussed the basic mechanisms of streams, events and callback functions it is important to consider the ordering semantics of asynchronous operations in a stream. These semantics are to allow application programmers to think about the ordering of operations in a stream in a safe way. There are some special cases where these semantics may be relaxed for purposes of performance optimization such as in the case of a *Programmatic Dependent Kernel Launch* scenario, which allows the overlap of two kernels through the use of special attributes and kernel launch mechanisms, or in the case of batching memory transfers using the `cudaMemcpyBatchAsync()` function when the runtime can perform non-overlapping batch copies concurrently. We will discuss these optimizations later on *link needed*.

Most importantly CUDA streams are what are known as in-order streams. This means that the order of execution of the operations in a stream is the same as the order in which those operations were enqueued. An operation in a stream cannot leap-frog other operations. Memory operations (such as copies) are tracked by the runtime and will always complete before the next operation in order to allow dependent kernels safe access to the data being transferred.

2.3.6. Blocking and non-blocking streams and the default stream

In CUDA there are two types of streams: blocking and non-blocking. The name can be a little misleading as the blocking and non-blocking semantics refer only to how the streams synchronize with the default stream. By default, streams created with `cudaStreamCreate()` are blocking streams. In order to create a non-blocking stream, the `cudaStreamCreateWithFlags()` function must be used with the `cudaStreamNonBlocking` flag:

```
cudaStream_t stream;
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
```

and non-blocking streams can be destroyed in the usual way with `cudaStreamDestroy()`.

2.3.6.1 Legacy Default Stream

The key difference between the blocking and non-blocking streams is how they synchronize with the **default stream**. CUDA provides a legacy default stream (also known as the NULL stream or the stream with stream ID 0) which is used when no stream is specified in kernel launches or in blocking `cudaMemcpy()` calls. This default stream, which was shared amongst all host threads, is a blocking stream. When an operation is launched into this default stream, it will synchronize with all other blocking streams, in other words it will wait for all other blocking streams to complete before it can execute.

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

kernel1<<<grid, block, 0, stream1>>>(...);
kernel2<<<grid, block>>>(...);
```

(continues on next page)

(continued from previous page)

```
kernel3<<<grid, block, 0, stream2>>>(...);

cudaDeviceSynchronize();
```

The default stream behavior means that in the above code snippet above, *kernel2* will wait for *kernel1* to complete, and *kernel3* will wait for *kernel2* to complete, even if in principle all three kernels could execute concurrently. By creating a non-blocking stream we can avoid this synchronization behavior. In the code snippet below we create two non-blocking streams. The default stream will no longer synchronize with these streams and in principle all three kernels could execute concurrently. As such we cannot assume any ordering of execution of the kernels and should perform explicit synchronization (such as with the rather heavy handed `cudaDeviceSynchronize()` call) in order to ensure that the kernels have completed.

```
cudaStream_t stream1, stream2;
cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);
cudaStreamCreateWithFlags(&stream2, cudaStreamNonBlocking);

kernel1<<<grid, block, 0, stream1>>>(...);
kernel2<<<grid, block>>>(...);
kernel3<<<grid, block, 0, stream2>>>(...);

cudaDeviceSynchronize();
```

2.3.6.2 Per-thread Default Stream

Starting in CUDA-7, CUDA allows for each host thread to have its own independent default stream, rather than the shared legacy default stream. In order to enable this behavior one must either use the *nvcc* compiler option `--default-stream per-thread` or define the `CUDA_API_PER_THREAD_DEFAULT_STREAM` preprocessor macro. When this behavior is enabled, each host thread will have its own independent default stream which will not synchronize with other streams in the same way the legacy default stream does. In such a situation the *legacy default stream example* will now exhibit the same synchronization behavior as the *non-blocking stream example*.

2.3.7. Explicit Synchronization

There are various ways to explicitly synchronize streams with each other.

`cudaDeviceSynchronize()` waits until all preceding commands in all streams of all host threads have completed.

`cudaStreamSynchronize()`takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

`cudaStreamWaitEvent()`takes a stream and an event as parameters (see *CUDA Events* for a description of events)and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()`delay their execution until the given event has completed.

`cudaStreamQuery()`provides applications with a way to know if all preceding commands in a stream have completed.

2.3.8. Implicit Synchronization

Two operations from different streams cannot run concurrently if any CUDA operation on the NULL stream is submitted in-between them, unless the streams are non-blocking streams (created with the `cudaStreamNonBlocking` flag).

Applications should follow these guidelines to improve their potential for concurrent kernel execution:

- All independent operations should be issued before dependent operations,
- Synchronization of any kind should be delayed as long as possible.

2.3.9. Miscellaneous and Advanced topics

2.3.9.1 Stream Prioritization

As mentioned previously, developers can assign priorities to CUDA streams. Prioritized streams need to be created using the `cudaStreamCreateWithPriority()` function. The function takes two parameters: the stream handle and the priority level. The general scheme is that lower numbers correspond to higher priorities. The given priority range for a given device and context can be queried using the `cudaDeviceGetStreamPriorityRange()` function. The default priority of a stream is 0.

```
int minPriority, maxPriority;

// Query the priority range for the device
cudaDeviceGetStreamPriorityRange(&minPriority, &maxPriority);

// Create two streams with different priorities
// cudaStreamDefault indicates the stream should be created with default flags
// in other words they will be blocking streams with respect to the legacy
↪default stream
// One could also use the option `cudaStreamNonBlocking` here to create a non-
↪blocking streams
cudaStream_t stream1, stream2;
cudaStreamCreateWithPriority(&stream1, cudaStreamDefault, minPriority);  //
↪Lowest priority
cudaStreamCreateWithPriority(&stream2, cudaStreamDefault, maxPriority);  //
↪Highest priority
```

We should note that a priority of a stream is only a hint to the runtime and generally applies primarily to kernel launches, and may not be respected for memory transfers. Stream priorities will not preempt already executing work, or guarantee any specific execution order.

2.3.9.2 Introduction to CUDA Graphs with Stream Capture

CUDA streams allow programs to specify a sequence of operations, kernels or memory copies, in order. Using multiple streams and cross-stream dependencies with `cudaStreamWaitEvent`, an application can specify a full directed acyclic graph (DAG) of operations. Some applications may have a sequence or DAG of operations that needs to be run many times throughout execution.

For this situation, CUDA provides a feature known as CUDA graphs. This section introduces CUDA graphs and one mechanism of creating them called *stream capture*. A more detailed discussion of CUDA graphs is presented in *CUDA Graphs*. Capturing or creating a graph can help reduce latency and CPU overhead of repeatedly invoking the same chain of API calls from the host thread. Instead, the

APIs to specify the graph operations can be called once, and then the resulting graph executed many times.

CUDA Graphs work in the following way:

i) The graph is *captured* by the application. This step is done once the first time the graph is executed. The graph can also be manually composed using the CUDA graph API.

ii) The graph is *instantiated*. This step is done one time, after the graph is captured. This step can set up all the various runtime structures needed to execute the graph, in order to make launching its components as fast as possible.

iii) In the remaining steps, the pre-instantiated graph is executed as many times as required. Since all the runtime structures needed to execute the graph operations are already in place, the CPU overheads of the graph execution are minimized.

Listing 2: The stages of capturing, instantiating and executing a simple linear graph using CUDA Graphs (from CUDA Developer Technical Blog, A. Gray, 2019)

```
#define N 500000 // tuned such that kernel takes a few microseconds

// A very lightweight kernel
__global__ void shortKernel(float * out_d, float * in_d){
    int idx=blockIdx.x*blockDim.x+threadIdx.x;
    if(idx<N) out_d[idx]=1.23*in_d[idx];
}

bool graphCreated=false;
cudaGraph_t graph;
cudaGraphExec_t instance;

// The graph will be executed NSTEP times
for(int istep=0; istep<NSTEP; istep++){
    if(!graphCreated){
        // Capture the graph
        cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);

        // Launch NKERNEL kernels
        for(int ikrnl=0; ikrnl<NKERNEL; ikrnl++){
            shortKernel<<<blocks, threads, 0, stream>>>(out_d, in_d);
        }

        // End the capture
        cudaStreamEndCapture(stream, &graph);

        // Instantiate the graph
        cudaGraphInstantiate(&instance, graph, NULL, NULL, 0);
        graphCreated=true;
    }

    // Launch the graph
    cudaGraphLaunch(instance, stream);

    // Synchronize the stream
```

(continues on next page)

(continued from previous page)

```
    cudaStreamSynchronize(stream);
}
```

Much more detail on CUDA graph is provided in *CUDA Graphs*.

2.3.10. Summary of Asynchronous Execution

The key points of this section are:

- Asynchronous APIs allow us to express concurrent execution of tasks providing the way to express overlapping of various operations. The actual concurrency achieved is dependent on available hardware resources and compute-capabilities.
- The key abstractions in CUDA for asynchronous execution are streams, events and callback functions.
- Synchronization is possible at the event, stream and device level
- The default stream is a blocking stream which synchronizes with all other blocking streams, but does not synchronize with non-blocking streams
- The default stream behavior can be avoided using per-thread default streams via the `--default-stream per-thread` compiler option or the CUDA_API_PER_THREAD_DEFAULT_STREAM preprocessor macro.
- Streams can be created with different priorities, which are hints to the runtime and may not be respected for memory transfers.
- CUDA provides API functions to reduce, or overlap overheads of kernel launches and memory transfers such as CUDA Graphs, Batched Memory Transfers and Programmatic Dependent Kernel Launch.

2.4. Unified and System Memory

Heterogeneous systems have multiple physical memories where data can be stored. The host CPU has attached DRAM, and every GPU in a system has its own attached DRAM. Performance is best when data is resident in the memory of the processor accessing it. CUDA provides APIs to *explicitly manage memory placement*, but this can be verbose and complicate software design. CUDA provides features and capabilities aimed at easing allocation, placement, and migration of data between different physical memories.

The purpose of this chapter is to introduce and explain these features and what they mean to application developers for both functionality and performance. Unified memory has several different manifestations which depend upon the OS, driver version, and GPU used. This chapter will show how to determine which unified memory paradigm applies and how the features of unified memory behave in each. The later *chapter on unified memory* explains unified memory in more detail.

The following concepts will be defined and explained in this chapter:

- *Unified Virtual Address Space* - CPU memory and each GPU's memory have a distinct range within a single virtual address space
- *Unified Memory* - A CUDA feature that enables managed memory which can be automatically migrated between CPU and GPUs
  - *Limited Unified Memory* - A unified memory paradigm with some limitations

  - *Full Unified Memory* - Full support for unified memory features
  - *Full Unified Memory with Hardware Coherency* - Full support for unified memory using hardware capabilities
  - *Unified memory hints* - APIs to guide unified memory behavior for specific allocations
- *Page-locked Host Memory* - Non-pageable system memory, which is necessary for some CUDA operations
  - *Mapped memory* - A mechanism (different from unified memory) for accessing host memory directly from a kernel

Additionally, the following terms used when discussing unified and system memory are introduced here:

- *Heterogeneous Managed Memory* (HMM) - A feature of the Linux kernel that enables software coherency for full unified memory
- *Address Translation Services* (ATS) - A hardware feature, available when GPUs are connected to the CPU by the NVLink Chip-to-Chip (C2C) interconnect, which provides hardware coherency for full unified memory

2.4.1. Unified Virtual Address Space

A single virtual address space is used for all host memory and all global memory on all GPUs in the system within a single OS process. All memory allocations on the host and on all devices lie in this virtual address space. This is true whether allocations are made with CUDA APIs (e.g. `cudaMalloc`, `cudaMallocHost`) or with system allocation APIs (e.g. `new`, `malloc`, `mmap`). The CPU and each GPU has a unique range within the unified virtual address space.

This means:

- The location of any memory (that is, CPU or which GPU's memory it lies in) can be determined from the value of a pointer using `cudaPointerGetAttributes()`
- The `cudaMemcpyKind` parameter of `cudaMemcpy*()` can be set to `cudaMemcpyDefault` to automatically determine the copy type from the pointers

2.4.2. Unified Memory

Unified memory is a CUDA memory feature which allows memory allocations called *managed memory* to be accessed from code running on either the CPU or the GPU. Unified memory was shown in *the intro to CUDA in C++*. Unified memory is available on all systems supported by CUDA.

On some systems, managed memory must be explicitly allocated. Managed memory can be explicitly allocated in CUDA in a few different ways:

- The CUDA API `cudaMallocManaged`
- The CUDA API `cudaMallocFromPoolAsync` with a pool created with `allocType` set to `cudaMemAllocationTypeManaged`
- Global variables with the `__managed__` specifier (see *Memory Space Specifiers*)

On systems with *HMM* or *ATS*, all system memory is implicitly managed memory, regardless of how it is allocated. No special allocation is needed.

2.4.2.1 Unified Memory Paradigms

The features and behavior of unified memory vary between operating systems, kernel versions on Linux, GPU hardware, and the GPU-CPU interconnect. The form of unified memory available can be determined by using `cudaDeviceGetAttribute` to query a few attributes:

- `cudaDevAttrConcurrentManagedAccess` - 1 for full unified memory support, 0 for limited support
- `cudaDevAttrPageableMemoryAccess` - 1 means all system memory is fully-supported unified memory, 0 means only memory explicitly allocated as managed memory is fully-supported unified memory
- `cudaDevAttrPageableMemoryAccessUsesHostPageTables` - Indicates the mechanism of CPU/GPU coherence: 1 is hardware, 0 is software.

Figure 18 illustrates how to determine the unified memory paradigm visually and is followed by a *code sample* implementing the same logic.

There are four paradigms of unified memory operation:

- *Full support for explicit managed memory allocations*
- *Full support for all allocations with software coherence*
- *Full support for all allocations with hardware coherence*
- *Limited unified memory support*

When full support is available, it can either require explicit allocations, or all system memory may implicitly be unified memory. When all memory is implicitly unified, the coherence mechanism can either be software or hardware. Windows and some Tegra devices have limited support for unified memory.

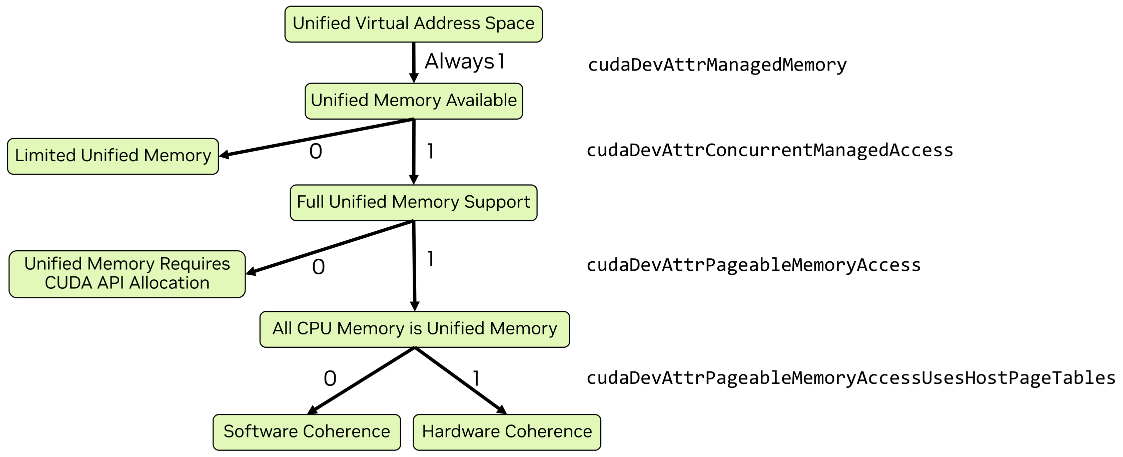

Unified Virtual Address Space
Always 1
cudaDevAttrManagedMemory
Unified Memory Available
Limited Unified Memory
0
1
cudaDevAttrConcurrentManagedAccess
Full Unified Memory Support
Unified Memory Requires CUDA API Allocation
0
1
cudaDevAttrPageableMemoryAccess
All CPU Memory is Unified Memory
0
1
cudaDevAttrPageableMemoryAccessUsesHostPageTables
Software Coherence
Hardware Coherence

Figure 18: All current GPUs use a unified virtual address space and have unified memory available. When `cudaDevAttrConcurrentManagedAccess` is 1, full unified memory support is available, otherwise only limited support is available. When full support is available, if `cudaDevAttrPageableMemoryAccess` is also 1, then all system memory is unified memory. Otherwise, only memory allocated with CUDA APIs (such as `cudaMallocManaged`) is unified memory. When all system memory is unified, `cudaDevAttrPageableMemoryAccessUsesHostPageTables` indicates whether coherence is provided by hardware (when value is 1) or software (when value is 0).

Table 3 shows the same information as Figure 18 as a table with links to the relevant sections of this chapter and more complete documentation in a later section of this guide.

Table 3: Overview of Unified Memory Paradigms

Unified Memory Paradigm	Device Attributes	Full Documentation
Limited unified memory support	`cudaDevAttrConcurrentMana` is 0	*Unified Memory on Windows, WSL, and Tegra* CUDA for Tegra Memory Management Unified memory on Tegra
Full support for explicit managed memory allocations	`cudaDevAttrPageableMemory` is 0 and `cudaDevAttrConcurrentManagedAccess` is 1	*Unified Memory on Devices with only CUDA Managed Memory Support*
Full support for all allocations with software coherence	`cudaDevAttrPageableMemory` is 0 and `cudaDevAttrPageableMemoryAccess` is 1 and `cudaDevAttrConcurrentManagedAccess` is 1	*Unified Memory on Devices with Full CUDA Unified Memory Support*
Full support for all allocations with hardware coherence	`cudaDevAttrPageableMemory` is 1 and `cudaDevAttrPageableMemoryAccess` is 1 and `cudaDevAttrConcurrentManagedAccess` is 1	*Unified Memory on Devices with Full CUDA Unified Memory Support*

2.4.2.1.1 Unified Memory Paradigm: Code Example

The following code example demonstrates querying the device attributes and determining the unified memory paradigm, following the logic of Figure 18, for each GPU in a system.

```
void queryDevices()
{
    int numDevices = 0;
    cudaGetDeviceCount(&numDevices);
    for(int i=0; i<numDevices; i++)
    {
        cudaSetDevice(i);
```

(continues on next page)

(continued from previous page)

```
        cudaInitDevice(0, 0, 0);
        int deviceId = i;

        int concurrentManagedAccess = -1;
        cudaDeviceGetAttribute (&concurrentManagedAccess,
↪cudaDevAttrConcurrentManagedAccess, deviceId);
        int pageableMemoryAccess = -1;
        cudaDeviceGetAttribute (&pageableMemoryAccess,
↪cudaDevAttrPageableMemoryAccess, deviceId);
        int pageableMemoryAccessUsesHostPageTables = -1;
        cudaDeviceGetAttribute (&pageableMemoryAccessUsesHostPageTables,
↪cudaDevAttrPageableMemoryAccessUsesHostPageTables, deviceId);

        printf("Device %d has ", deviceId);
        if(concurrentManagedAccess){
            if(pageableMemoryAccess){
                printf("full unified memory support");
                if( pageableMemoryAccessUsesHostPageTables)
                    { printf(" with hardware coherency\n");  }
                else
                    { printf(" with software coherency\n"); }
            }
            else
                { printf("full unified memory support for CUDA-made managed
↪allocations\n"); }
        }
        else
        {   printf("limited unified memory support: Windows, WSL, or Tegra\n
↪");  }
    }
}
```

2.4.2.2 Full Unified Memory Feature Support

Most Linux systems have full unified memory support. If device attribute `cudaDevAttrPageableMemoryAccess` is 1, then all system memory, whether allocated by CUDA APIs or system APIs, operates as unified memory with full feature support. This includes file-backed memory allocations created with `mmap`.

If `cudaDevAttrPageableMemoryAccess` is 0, then only memory allocated as managed memory by CUDA behaves as unified memory. Memory allocated with system APIs is not managed and is not necessarily accessible from GPU kernels.

In general, for unified allocations with full support:

- Managed memory is usually allocated in the memory space of the processor where it is first touched
- Managed memory is usually migrated when it is used by a processor other than the processor where it currently resides
- Managed memory is migrated or accessed at the granularity of memory pages (software coherence) or cache lines (hardware coherence)

- Oversubscription is allowed: an application may allocate more managed memory than is physically available on the GPU

Allocation and migration behavior can deviate from the above. This can by influenced the programmer using *hints and prefetches*. Full coverage of full unified memory support can be found in *Unified Memory on Devices with Full CUDA Unified Memory Support*.

2.4.2.2.1 Full Unified Memory with Hardware Coherency

On hardware such as Grace Hopper and Grace Blackwell, where an NVIDIA CPU is used and the interconnect between the CPU and GPU is NVLink Chip-to-Chip (C2C), address translation services (ATS) are available. `cudaDevAttrPageableMemoryAccessUsesHostPageTables` is 1 when ATS is available.

With ATS, in addition to full unified memory support for all host allocations:

- GPU allocations (e.g. `cudaMalloc`) can be accessed from the CPU (`cudaDevAttrDirectManagedMemAccessFromHost` will be 1)
- The link between CPU and GPU supports native atomics (`cudaDevAttrHostNativeAtomicSupported` will be 1)
- Hardware support for coherence can improve performance compared to software coherence

ATS provides all capabilities of *HMM*. When ATS is available, HMM is automatically disabled. Further discussion of hardware vs. software coherency is found in *CPU and GPU Page Tables: Hardware Coherency vs. Software Coherency*.

2.4.2.2.2 HMM - Full Unified Memory with Software Coherency

Heterogeneous Memory Management (HMM) is a feature available on Linux operating systems (with appropriate kernel versions) which enables software-coherent *full unified memory support*. Heterogeneous memory management brings some of the capabilities and convenience provided by ATS to PCIe-connected GPUs.

On Linux with at least Linux Kernel 6.1.24, 6.2.11, or 6.3 or later, heterogeneous memory management (HMM) may be available. The following command can be used to find if the addressing mode is `HMM`.

```
$ nvidia-smi -q | grep Addressing
Addressing Mode : HMM
```

When HMM is available, *full unified memory* is supported and all system allocations are implicitly unified memory. If a system also has *ATS*, HMM is disabled and ATS is used, since ATS provides all the capabilities of HMM and more.

2.4.2.3 Limited Unified Memory Support

On Windows, including Windows Subsystem for Linux (WSL), and on some Tegra systems, a limited subset of unified memory functionality is available. On these systems, managed memory is available, but migration between CPU and GPUs behaves differently.

- Managed memory is first allocated in the CPU’s physical memory
- Managed memory is migrated in larger granularity than virtual memory pages
- Managed memory is migrated to the GPU when the GPU begins executing
- The CPU must not access managed memory while the GPU is active
- Managed memory is migrated back to the CPU when the GPU is synchronized

- Oversubscription of GPU memory is not allowed
- Only memory explicitly allocated by CUDA as managed memory is unified

Full coverage of this paradigm can be found in *Unified Memory on Windows, WSL, and Tegra*.

2.4.2.4 Memory Advise and Prefetch

The programmer can provide hints to the NVIDIA Driver managing unified memory to help it maximize application performance. The CUDA API `cudaMemAdvise` allows the programmer to specify properties of allocations that affect where they are placed and whether or not the memory is migrated when accessed from another device.

`cudaMemPrefetchAsync` allows the programmer to suggest an asynchronous migration of a specific allocation to a different location be started. A common use is starting the transfer of data a kernel will use before the kernel is launched. This enables the copy of data to occur while other GPU kernels are executing.

The section on *Performance Hints* covers the different hints that can be passed to `cudaMemAdvise` and shows examples of using `cudaMemPrefetchAsync`.

2.4.3. Page-Locked Host Memory

In *introductory code examples*, `cudaMallocHost` was used to allocate memory on the CPU. This allocates *page-locked* memory (also known as *pinned* memory) on the host. Host allocations made through traditional allocation mechanisms like `malloc`, `new`, or `mmap` are not page-locked, which means they may be swapped to disk or physically relocated by the operating system.

Page-locked host memory is required for *asynchronous copies between the CPU and GPU*. Page-locked host memory also improves performance of synchronous copies. Page-locked memory can be *mapped* to the GPU for direct access from GPU kernels.

The CUDA runtime provides APIs to allocate page-locked host memory or to page-lock existing allocations:

- `cudaMallocHost` allocates page-locked host memory
- `cudaHostAlloc` defaults to the same behavior as `cudaMallocHost`, but also takes flags to specify other memory parameters
- `cudaFreeHost` frees memory allocated with `cudaMallocHost` or `cudaHostAlloc`
- `cudaHostRegister` page-locks a range of existing memory allocated outside the CUDA API, such as with `malloc` or `mmap`

`cudaHostRegister` enables host memory allocated by 3rd party libraries or other code outside of a developer's control to be page-locked so that it can be used in asynchronous copies or mapped.

Note

Page-locked host memory can be used for asynchronous copies and mapped-memory by all GPUs in the system.

Page-locked host memory is not cached on non I/O coherent Tegra devices. Also, `cudaHostRegister()` is not supported on non I/O coherent Tegra devices.

2.4.3.1 Mapped Memory

On systems with *HMM* or *ATS*, all host memory is directly accessible from the GPU using the host pointers. When ATS or HMM are not available, host allocations can be made accessible to the GPU by *mapping* the memory into the GPU's memory space. Mapped memory is always page-locked.

The code examples which follow will illustrate the following array copy kernel operating directly on mapped host memory.

```
__global__ void copyKernel(float* a, float* b)
{
        int idx = threadIdx.x + blockDim.x * blockIdx.x;
        a[idx] = b[idx];
}
```

While mapped memory may be useful in some cases where certain data which is not copied to the GPU needs to be accessed from a kernel, accessing mapped memory in a kernel requires transactions across the CPU-GPU interconnect, PCIe, or NVLink C2C. These operations have higher latency and lower bandwidth compared to accessing device memory. Mapped memory should not be considered a performant alternative to *unified memory* or *explicit memory management* for the majority of a kernel's memory needs.

2.4.3.1.1 cudaMallocHost and cudaHostAlloc

Host memory allocated with `cudaHostMalloc` or `cudaHostAlloc` is automatically mapped. The pointers returned by these APIs can be directly used in kernel code to access the memory on the host. The host memory is accessed over the CPU-GPU interconnect.

cudaMallocHost

```
void usingMallocHost() {
  float* a = nullptr;
  float* b = nullptr;

  CUDA_CHECK(cudaMallocHost(&a, vLen*sizeof(float)));
  CUDA_CHECK(cudaMallocHost(&b, vLen*sizeof(float)));

  initVector(b, vLen);
  memset(a, 0, vLen*sizeof(float));

  int threads = 256;
  int blocks = vLen/threads;
  copyKernel<<<blocks, threads>>>(a, b);
  CUDA_CHECK(cudaGetLastError());
  CUDA_CHECK(cudaDeviceSynchronize());

  printf("Using cudaMallocHost: ");
  checkAnswer(a,b);
}
```

cudaAllocHost

```
void usingCudaHostAlloc() {
  float* a = nullptr;
  float* b = nullptr;

  CUDA_CHECK(cudaHostAlloc(&a, vLen*sizeof(float), cudaHostAllocMapped));
  CUDA_CHECK(cudaHostAlloc(&b, vLen*sizeof(float), cudaHostAllocMapped));

  initVector(b, vLen);
  memset(a, 0, vLen*sizeof(float));

  int threads = 256;
  int blocks = vLen/threads;
  copyKernel<<<blocks, threads>>>(a, b);
  CUDA_CHECK(cudaGetLastError());
  CUDA_CHECK(cudaDeviceSynchronize());

  printf("Using cudaAllocHost: ");
  checkAnswer(a, b);
}
```

2.4.3.1.2 cudaHostRegister

When ATS and HMM are not available, allocations made by system allocators can still be mapped for access directly from GPU kernels using `cudaHostRegister`. Unlike memory created with CUDA APIs, however, the memory cannot be accessed from the kernel using the host pointer. A pointer in the device's memory region must be obtained using `cudaHostGetDevicePointer()`, and that pointer must be used for accesses in kernel code.

```
void usingRegister() {
  float* a = nullptr;
  float* b = nullptr;
  float* devA = nullptr;
  float* devB = nullptr;

  a = (float*)malloc(vLen*sizeof(float));
  b = (float*)malloc(vLen*sizeof(float));
  CUDA_CHECK(cudaHostRegister(a, vLen*sizeof(float), 0 ));
  CUDA_CHECK(cudaHostRegister(b, vLen*sizeof(float), 0  ));

  CUDA_CHECK(cudaHostGetDevicePointer((void**)&devA, (void*)a, 0));
  CUDA_CHECK(cudaHostGetDevicePointer((void**)&devB, (void*)b, 0));

  initVector(b, vLen);
  memset(a, 0, vLen*sizeof(float));

  int threads = 256;
  int blocks = vLen/threads;
  copyKernel<<<blocks, threads>>>(devA, devB);
  CUDA_CHECK(cudaGetLastError());
  CUDA_CHECK(cudaDeviceSynchronize());
```

(continues on next page)

(continued from previous page)

```
  printf("Using cudaHostRegister: ");
  checkAnswer(a, b);
}
```

2.4.3.1.3 Comparing Unified Memory and Mapped Memory

Mapped memory makes CPU memory accessible from the GPU, but does not guarantee that all types of access, for example atomics, are supported on all systems. Unified memory guarantees that all access types are supported.

Mapped memory remains in CPU memory, which means all GPU accesses must go through the connection between the CPU and GPU: PCIe or NVLink. Latency of accesses made across these links are significantly higher than access to GPU memory, and total available bandwidth is lower. As such, using mapped memory for all kernel memory accesses is unlikely to fully utilize GPU computing resources.

Unified memory is most often migrated to the physical memory of the processor accessing it. After the first migration, repeated access to the same memory page or cache line by a kernel can utilize the full GPU memory bandwidth.

Note

Mapped memory has also been referred to as *zero-copy* memory in previous documents.

Prior to all CUDA applications using a *unified virtual address space*, additional APIs were needed to enable memory mapping (`cudaSetDeviceFlags` with `cudaDeviceMapHost`). These APIs are no longer needed.

Atomic functions (see *Atomic Functions*) operating on mapped host memory are not atomic from the point of view of the host or other GPUs.

CUDA runtime requires that 1-byte, 2-byte, 4-byte, 8-byte, and 16-byte naturally aligned loads and stores to host memory initiated from the device are preserved as single accesses from the point of view of the host and other devices. On some platforms, atomics to memory may be broken by the hardware into separate load and store operations. These component load and store operations have the same requirements on preservation of naturally aligned accesses. The CUDA runtime does not support a PCI Express bus topology where a PCI Express bridge splits 8-byte naturally aligned operations and NVIDIA is not aware of any topology that splits 16-byte naturally aligned operations.

2.4.4. Summary

- On Linux platforms with heterogeneous memory management (HMM) or address translation services (ATS), all system-allocated memory is managed memory
- On Linux platforms without HMM or ATS, on Tegra processors, and on all Windows platforms, managed memory must be allocated using CUDA:
  - `cudaMallocManaged` or
  - `cudaMallocFromPoolAsync` with a pool created with `allocType=cudaMemAllocationTypeManaged`
  - Global variables with `__managed__` specifier
- On Windows and Tegra processors, unified memory has limitations

- On NVLINK C2C connected systems with ATS, device memory allocated with `cudaMalloc` can be directly accessed from the CPU or other GPUs

2.5. NVCC: The NVIDIA CUDA Compiler

The NVIDIA CUDA Compiler `nvcc` is a toolchain from NVIDIA for compiling CUDA C/C++ as well as PTX code. The toolchain is part of the CUDA Toolkit and consists of several tools, including the compiler, linker, and the PTX and *Cubin* assemblers. The top-level `nvcc` tool coordinates the compilation process, invoking the appropriate tool for each stage of compilation.

`nvcc` drives offline compilation of CUDA code, in contrast to online or Just-in-Time (JIT) compilation driven by the CUDA runtime compiler nvrtc.

This chapter covers the most common uses and details of `nvcc` needed for building applications. Full coverage of `nvcc` is found in the nvcc documentation.

2.5.1. CUDA Source Files and Headers

Source files compiled with `nvcc` may contain a combination of host code, which executes on the CPU, and device code that executes on the GPU. `nvcc` accepts the common C/C++ source file extensions `.c`, `.cpp`, `.cc`, `.cxx` for host-only code and `.cu` for files that contain device code or a mix of host and device code. Headers containing device code typically adopt the `.cuh` extension to distinguish them from host-only code headers `.h`, `.hpp`, `.hh`, `.hxx`, etc.

File Extension	Description	Content
`.c`	C source file	Host-only code
`.cpp`, `.cc`, `.cxx`	C++ source file	Host-only code
`.h`, `.hpp`, `.hh`, `.hxx`	C/C++ header file	Device code, host code, mix of host/device code
`.cu`	CUDA source file	Device code, host code, mix of host/device code
`.cuh`	CUDA header file	Device code, host code, mix of host/device code

2.5.2. NVCC Compilation Workflow

In the initial phase, `nvcc` separates the device code from the host code and dispatches their compilation to the GPU and the host compilers, respectively.

To compile the host code, the CUDA compiler `nvcc` requires a compatible host compiler to be available. The CUDA Toolkit defines the host compiler support policy for Linux and Windows platforms.
Files containing only host code can be built using either `nvcc` or the host compiler directly. The resulting object files can be combined with object files from `nvcc` which contain GPU code at link-time.

The GPU compiler compiles the C/C++ device code to PTX assembly code. The GPU compiler is run for each virtual machine instruction set architecture (e.g. `compute_90`) specified in the compilation command line.

Individual PTX code is then passed to the `ptxas` tool, which generates *Cubin* for the target hardware ISAs. The hardware ISA is identified by its *SM version*.

It is possible to embed multiple PTX and Cubin targets into a single binary *Fatbin* container within an application or library so that a single binary can support multiple virtual and target hardware ISAs.

The invocation and coordination of the tools described above are done automatically by `nvcc`. The `-v` option can be used to display the full compilation workflow and tool invocation. The `-keep` option can be used to save the intermediate files generated during the compilation in the current directory or in the directory specified by `--keep-dir` instead.

The following example illustrates the compilation workflow for a CUDA source file `example.cu`:

```
// ----- example.cu -----
#include <stdio.h>
__global__ void kernel() {
    printf("Hello from kernel\n");
}

void kernel_launcher() {
    kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}

int main() {
    kernel_launcher();
    return 0;
}
```

nvcc basic compilation workflow:

nvcc compilation workflow with multiple PTX and Cubin architectures:

A more detailed description of the `nvcc` compilation workflow can be found in the compiler documentation.

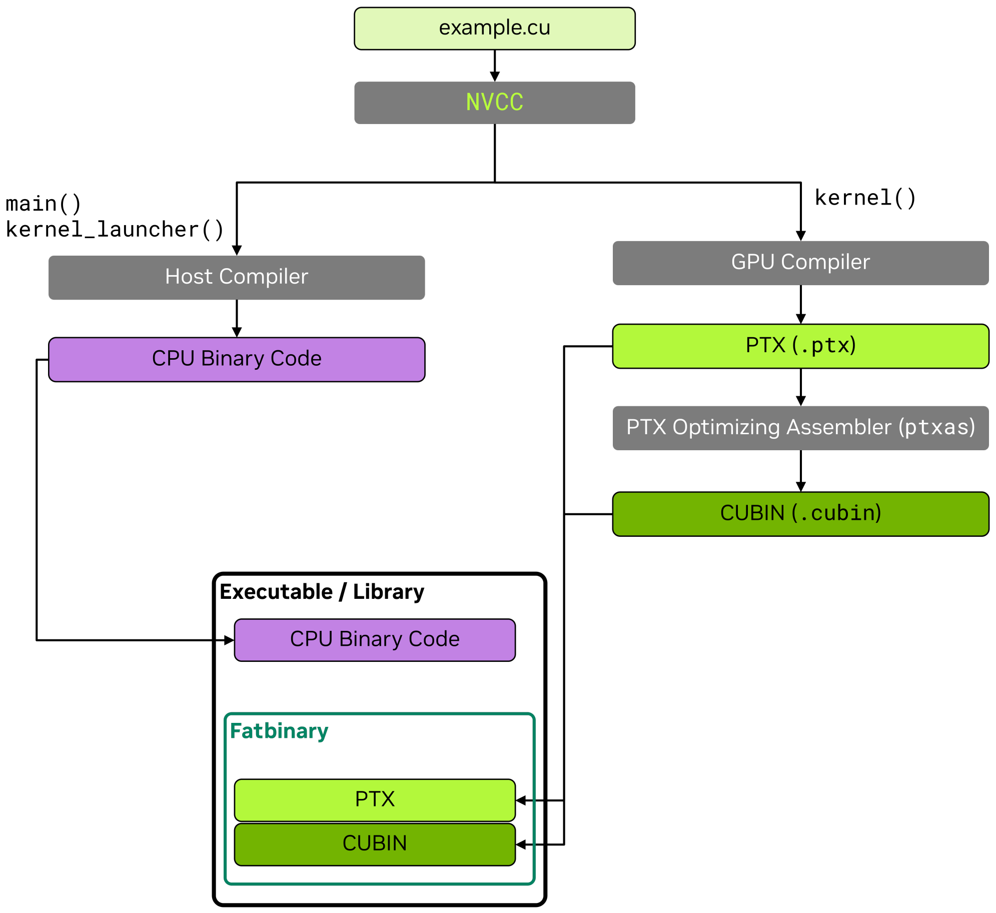
2.5.3. NVCC Basic Usage

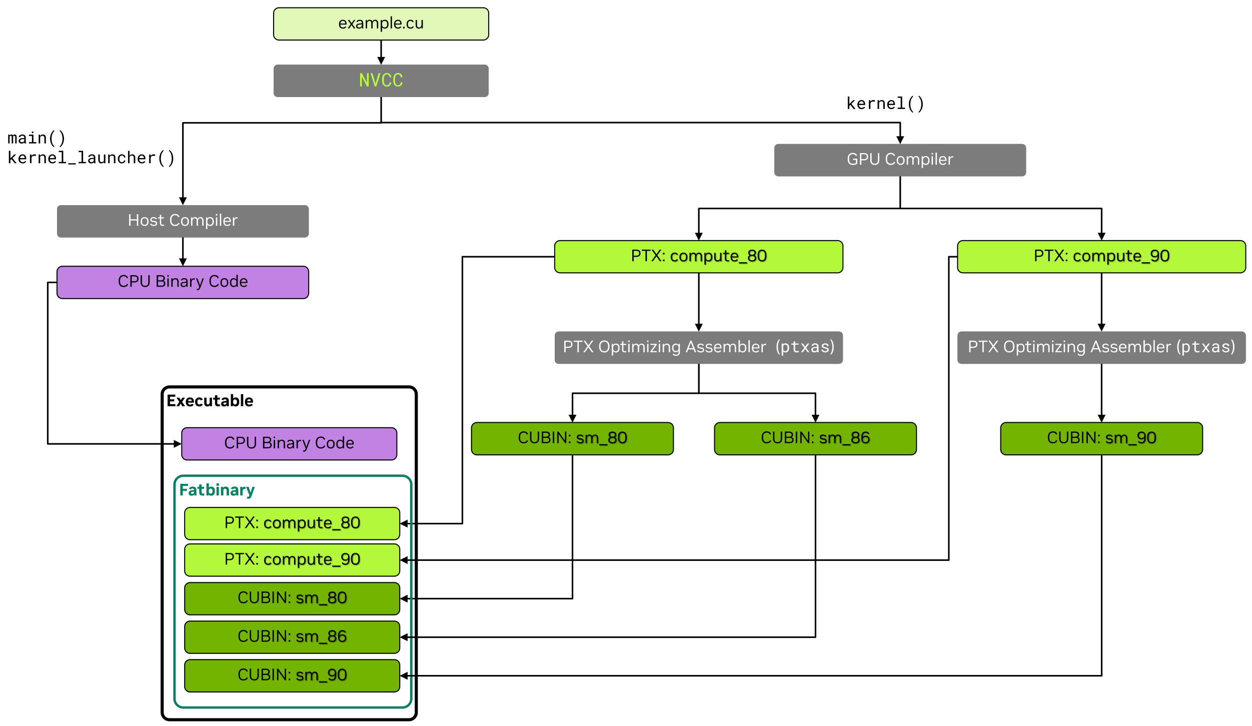
The basic command to compile a CUDA source file with `nvcc` is:

```
nvcc <source_file>.cu -o <output_file>
```

`nvcc` accepts common compiler flags used for specifying include directories `-I <path>` and library paths `-L <path>`, linking against other libraries `-l<library>`, and defining macros `-D<macro>=<value>`.

```
nvcc example.cu -I path_to_include/ -L path_to_library/ -lcublas -o <output_
↪file>
```


example.cu
NVCC
main()
kernel_launcher()
kernel()
Host Compiler
GPU Compiler
CPU Binary Code
PTX (.ptx)
PTX Optimizing Assembler (ptxas)
CUBIN (.cubin)
Executable / Library
CPU Binary Code
Fatbinary
PTX
CUBIN


example.cu
NVCC
main()
kernel_launcher()
kernel()
Host Compiler
GPU Compiler
CPU Binary Code
PTX: compute_80
PTX: compute_90
PTX Optimizing Assembler (ptxas)
PTX Optimizing Assembler (ptxas)
CUBIN: sm_80
CUBIN: sm_86
CUBIN: sm_90
Executable
CPU Binary Code
Fatbinary
PTX: compute_80
PTX: compute_90
CUBIN: sm_80
CUBIN: sm_86
CUBIN: sm_90

2.5.3.1 NVCC PTX and Cubin Generation

By default, nvcc generates PTX and Cubin for the earliest GPU architecture (lowest `compute_XY` and `sm_XY` version) supported by the CUDA Toolkit to maximize compatibility.

- The `-arch` option can be used to generate PTX and Cubin for a specific GPU architecture.
- The `-gencode` option can be used to generate PTX and Cubin for multiple GPU architectures.

The complete list of supported virtual and real GPU architectures can be obtained by passing the `--list-gpu-code` and `--list-gpu-arch` flags respectively, or by referring to the Virtual Architecture List and the GPU Architecture List sections within the nvcc documentation.

```
nvcc --list-gpu-code # list all supported real GPU architectures
nvcc --list-gpu-arch # list all supported virtual GPU architectures
```

```
nvcc example.cu -arch=compute_<XY> # e.g. -arch=compute_80 for NVIDIA Ampere
↪GPUs and later
                                   # PTX-only, GPU forward compatible

nvcc example.cu -arch=sm_<XY>      # e.g. -arch=sm_80 for NVIDIA Ampere GPUs
↪and later
                                   # PTX and Cubin, GPU forward compatible

nvcc example.cu -arch=native       # automatically detects and generates Cubin
↪for the current GPU
                                   # no PTX, no GPU forward compatibility

nvcc example.cu -arch=all          # generate Cubin for all supported GPU
↪architectures
                                   # also includes the latest PTX for GPU
↪forward compatibility

nvcc example.cu -arch=all-major    # generate Cubin for all major supported
↪GPU architectures, e.g. sm_80, sm_90,
                                   # also includes the latest PTX for GPU
↪forward compatibility
```

More advanced usage allows PTX and Cubin targets to be specified individually:

```
# generate PTX for virtual architecture compute_80 and compile it to Cubin for
↪real architecture sm_86, keep compute_80 PTX
nvcc example.cu -arch=compute_80 -gpu-code=sm_86,compute_80 # (PTX and Cubin)

# generate PTX for virtual architecture compute_80 and compile it to Cubin for
↪real architecture sm_86, sm_89
nvcc example.cu -arch=compute_80 -gpu-code=sm_86,sm_89    # (no PTX)
nvcc example.cu -gencode=arch=compute_80,code=sm_86,sm_89 # same as above

# (1) generate PTX for virtual architecture compute_80 and compile it to Cubin
↪for real architecture sm_86, sm_89
# (2) generate PTX for virtual architecture compute_90 and compile it to Cubin
↪for real architecture sm_90
nvcc example.cu -gencode=arch=compute_80,code=sm_86,sm_89 -
↪gencode=arch=compute_90,code=sm_90
```

The full reference of nvcc command-line options for steering GPU code generation can be found in the nvcc documentation.

2.5.3.2 Host Code Compilation Notes

Compilation units, namely a source file and its headers, that do not contain device code or symbols can be compiled directly with a host compiler. If any compilation unit uses CUDA runtime API functions, the application must be linked with the CUDA runtime library. The CUDA runtime is available as both a static and a shared library, `libcudart_static` and `libcudart`, respectively. By default, `nvcc` links against the static CUDA runtime library. To use the shared library version of the CUDA runtime, pass the flag `--cudart=shared` to `nvcc` on the compile or link command.

`nvcc` allows the host compiler used for host functions to be specified via the `-ccbin <compiler>` argument. The environment variable `NVCC_CCBIN` can also be defined to specify the host compiler used by `nvcc`. The `-Xcompiler` argument to `nvcc` passes through arguments to the host compiler. For example, in the example below, the `-O3` argument is passed to the host compiler by `nvcc`.

```
nvcc example.cu -ccbin=clang++

export NVCC_CCBIN='gcc'
nvcc example.cu -Xcompiler=-O3
```

2.5.3.3 Separate Compilation of GPU Code

`nvcc` defaults to *whole-program compilation*, which expects all GPU code and symbols to be present in the compilation unit that uses them. CUDA device functions may call device functions or access device variables defined in other compilation units, but either the `-rdc=true` or its alias the `-dc` flag must be specified on the `nvcc` command line to enable linking of device code from different compilation units. The ability to link device code and symbols from different compilation units is called *separate compilation*.

Separate compilation allows more flexible code organization, can improve compile time, and can lead to smaller binaries. Separate compilation may involve some build-time complexity compared to whole-program compilation. Performance can be affected by the use of device code linking, which is why it is not used by default. *Link-Time Optimization (LTO)* can help reduce the performance overhead of separate compilation.

Separate compilation requires the following conditions:

- Non-`const` device variables defined in one compilation unit must be referred to with the `extern` keyword in other compilation units.
- All `const` device variables must be defined and referred to with the `extern` keyword.
- All CUDA source files `.cu` must be compiled with the `-dc` or `-rdc=true` flags.

Host and device functions have external linkage by default and do not require the `extern` keyword. Note that starting from CUDA 13, `__global__` functions and `__managed__`/`__device__`/`__constant__` variables have internal linkage by default.

In the following example, `definition.cu` defines a variable and a function, while `example.cu` refers to them. Both files are compiled separately and linked into the final binary.

```
// ----- definition.cu -----
extern __device__ int device_variable = 5;
__device__        int device_function() { return 10; }
```

```
// ----- example.cu -----
extern __device__ int  device_variable;
__device__        int device_function();

__global__ void kernel(int* ptr) {
    device_variable = 0;
    *ptr            = device_function();
}
```

```
nvcc -dc definition.cu -o definition.o
nvcc -dc example.cu    -o example.o
nvcc definition.o example.o -o program
```

2.5.4. Common Compiler Options

This section presents the most relevant compiler options that can be used with nvcc, covering language features, optimization, debugging, profiling, and build aspects. The full description of all options can be found in the nvcc documentation.

2.5.4.1 Language Features

nvcc supports the C++ core language features, from C++03 to C++20. The `-std` flag can be used to specify the language standard to use:

- `--std={c++03|c++11|c++14|c++17|c++20}`

In addition, nvcc supports the following language extensions:

- `-restrict`: Assert that all kernel pointer parameters are *restrict* pointers.
- `-extended-lambda`: Allow `__host__`, `__device__` annotations in lambda declarations.
- `-expt-relaxed-constexpr`: (Experimental flag) Allow host code to invoke `__device__ constexpr` functions, and device code to invoke `__host__ constexpr` functions.

More detail on these features can be found in the *extended lambda* and *constexpr* sections.

2.5.4.2 Debugging Options

nvcc supports the following options to generate debug information:

- `-g`: Generate debug information for host code. `gdb/lldb` and similar tools rely on such information for host code debugging.
- `-G`: Generate debug information for device code. cuda-gdb relies on such information for device-code debugging. The flag also defines the `__CUDACC_DEBUG__` macro.
- `-lineinfo`: Generate line-number information for device code. This option does not affect execution performance and is useful in conjunction with the compute-sanitizer tool to trace the kernel execution.

nvcc uses the highest optimization level `-O3` for GPU code by default. The debug flag `-G` prevents some compiler optimizations, and so debug code is expected to have lower performance than non-debug code. The `-DNDEBUG` flag can be defined to disable runtime assertions, as these can also slow down execution.

2.5.4.3 Optimization Options

nvcc provides many options for optimizing performance. This section aims to provide a brief survey of some of the options available that developers may find useful, as well as links to further information. Complete coverage can be found in the nvcc documentation.

- `-Xptxas` passes arguments to the PTX assembler tool `ptxas`. The `nvcc` documentation provides a list of useful arguments for `ptxas`. For example, `-Xptxas=-maxrregcount=N` specifies the maximum number of registers to use, per thread.
- `-extra-device-vectorization`: Enables more aggressive device code vectorization.
- Additional flags which provide fine-grained control over floating point behavior are covered in the *Floating-Point Computation* section and in the nvcc documentation.

The following flags get output from the compiler which can be useful in more advanced code optimization:

- `-res-usage`: Print resource usage report after compilation. It includes the number of registers, shared memory, constant memory, and local memory allocated for each kernel function.
- `-opt-info=inline`: Print information about inlined functions.
- `-Xptxas=-warn-lmem-usage`: Warn if local memory is used.
- `-Xptxas=-warn-spills`: Warn if registers are spilled to local memory.

2.5.4.4 Link-Time Optimization (LTO)

Separate compilation can result in lower performance than whole-program compilation due to limited cross-file optimization opportunities. Link-Time Optimization (LTO) addresses this by performing optimizations across separately compiled files at link time, at the cost of increased compilation time. LTO can recover much of the performance of whole-program compilation while maintaining the flexibility of separate compilation.

nvcc requires the `-dlto` flag or `lto_<SM version>` link-time optimization targets to enable LTO:

```
nvcc -dc -dlto -arch=sm_100 definition.cu -o definition.o
nvcc -dc -dlto -arch=sm_100 example.cu    -o example.o
nvcc -dlto definition.o example.o -o program
```

```
nvcc -dc -arch=lto_100 definition.cu -o definition.o
nvcc -dc -arch=lto_100 example.cu    -o example.o
nvcc -dlto definition.o example.o -o program
```

2.5.4.5 Profiling Options

It is possible to directly profile a CUDA application using the Nsight Compute and Nsight Systems tools without the need for additional flags during the compilation process. However, additional information which can be generated by nvcc can assist profiling by correlating source files with the generated code:

- `-lineinfo`: Generate line-number information for device code; this allows viewing the source code in the profiling tools. Profiling tools require the original source code to be available in the same location where the code was compiled.
- `-src-in-ptx`: Keep the original source code in the PTX, avoiding the limitations of `-lineinfo` mentioned above. Requires `-lineinfo`.

2.5.4.6 Fatbin Compression

`nvcc` compresses the *fatbins* stored in application or library binaries by default. Fatbin compression can be controlled using the following options:

- `-no-compress`: Disable the compression of the fatbin.
- `--compress-mode={default|size|speed|balance|none}`: Set the compression mode. `speed` focuses on fast decompression time, while `size` aims at reducing the fatbin size. `balance` provides a trade-off between speed and size. The default mode is `speed`. `none` disables compression.

2.5.4.7 Compiler Performance Controls

`nvcc` provides options to analyze and accelerate the compilation process itself:

- `-t <N>`: The number of CPU threads used to parallelize the compilation of a single compilation unit for multiple GPU architectures.
- `-split-compile <N>`: The number of CPU threads used to parallelize the optimization phase.
- `-split-compile-extended <N>`: More aggressive form of split compilation. Requires link-time optimization.
- `-Ofc <N>`: Level of device code compilation speed.
- `-time <filename>`: Generate a comma-separated value (CSV) table with the time taken by each compilation phase.
- `-fdevice-time-trace`: Generate a time trace for device code compilation.

Chapter 3. Advanced CUDA

3.1. Advanced CUDA APIs and Features

This section will cover use of more advanced CUDA APIs and features. These topics cover techniques or features that do not usually require CUDA kernel modifications, but can still influence, from the host-side, application-level behavior, both in terms of GPU work execution and performance as well as CPU-side performance.

3.1.1. cudaLaunchKernelEx

When the *triple chevron notation* was introduced in first versions of, the *Kernel Configuration* of a kernel had only four programmable parameters: - thread block dimensions - grid dimensions - dynamic shared-memory (optional, 0 if unspecified) - stream (default stream used if unspecified)

Some CUDA features can benefit from additional attributes and hints provided with a kernel launch. The `cudaLaunchKernelEx` enables a program to set the above mentioned execution configuration parameters via the `cudaLaunchConfig_t` structure. In addition, the `cudaLaunchConfig_t` structure allows the program to pass in zero or more `cudaLaunchAttributes` to control or suggest other parameters for the kernel launch. For example, the `cudaLaunchAttributePreferredSharedMemoryCarveout` discussed later in this chapter (see *Configuring L1/Shared Memory Balance*) is specified using `cudaLaunchKernelEx`. The `cudaLaunchAttributeClusterDimension` attribute, discussed later in this chapter, is used to specify the desired cluster size for the kernel launch.

The complete list of supported attributes and their meaning is captured in the CUDA Runtime API Reference Documentation.

3.1.2. Launching Clusters:

Thread block clusters, introduced in previous sections, are an optional level of thread block organization available in compute capability 9.0 and higher which enable applications to guarantee that thread blocks of a cluster are simultaneously executed on single GPC. This enables larger groups of threads than those that fit in a single SM to exchange data and synchronize with each other.

Section Section 2.1.10.1 showed how a kernel which uses clusters can be specified and launched using triple chevron notation. In this section, the `__cluster_dims__` annotation was used to specify the dimensions of the cluster which must be used to launch the kernel. When using triple chevron notation, the size of the clusters is determined implicitly.

3.1.2.1 Launching with Clusters using cudaLaunchKernelEx

Unlike *launching kernels using clusters with triple chevron notation*, the size of the thread block cluster can be configured on a per-launch basis. The code example below shows how to launch a cluster kernel using `cudaLaunchKernelEx`.

```
// Kernel definition
// No compile time attribute attached to the kernel
__global__ void cluster_kernel(float *input, float* output)
{

}

int main()
{
    float *input, *output;
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);

    // Kernel invocation with runtime cluster size
    {
        cudaLaunchConfig_t config = {0};
        // The grid dimension is not affected by cluster launch, and is still
↪enumerated
        // using number of blocks.
        // The grid dimension should be a multiple of cluster size.
        config.gridDim = numBlocks;
        config.blockDim = threadsPerBlock;

        cudaLaunchAttribute attribute[1];
        attribute[0].id = cudaLaunchAttributeClusterDimension;
        attribute[0].val.clusterDim.x = 2; // Cluster size in X-dimension
        attribute[0].val.clusterDim.y = 1;
        attribute[0].val.clusterDim.z = 1;
        config.attrs = attribute;
        config.numAttrs = 1;

        cudaLaunchKernelEx(&config, cluster_kernel, input, output);
    }
}
```

There are two `cudaLaunchAttribute` types which are relevant to thread block clusters clusters: `cudaLaunchAttributeClusterDimension` and `cudaLaunchAttributePreferredClusterDimension`.

The attribute id `cudaLaunchAttributeClusterDimension` specifies the required dimensions with which to execute the cluster. The value for this attribute, `clusterDim`, is a 3-dimensional value. The corresponding dimensions of the grid (x, y, and z) must be divisible by the respective dimensions of the specified cluster dimension. Setting this is similar to using the `__cluster_dims__` attribute on the kernel definition at compile time as shown in *Launching with Clusters in Triple Chevron Notation*, but can be changed at runtime for different launches of the same kernel.

On GPUs with compute capability of 10.0 and higher, another attribute id `cudaLaunchAttributePreferredClusterDimension` allows the application to additionally specify a preferred dimension for the cluster. The preferred dimension must be an integer multiple of the minimum cluster dimensions specified by the `__cluster_dims__` attribute on the kernel or the `cudaLaunchAt-`

tributeClusterDimension attribute to cudaLaunchKernelEx. That is, a minimum cluster dimension must be specified in addition to the preferred cluster dimension. The corresponding dimensions of the grid (x, y, and z) must be divisible by the respective dimension of the specified preferred cluster dimension.

All thread blocks will execute in clusters of at least the minimum cluster dimension. Where possible, clusters of the preferred dimension will be used, but not all clusters are guaranteed to execute with the preferred dimensions. All thread blocks will execute in clusters with either the minimum or preferred cluster dimension. Kernels which use a preferred cluster dimension must be written to operate correctly in either the minimum or the preferred cluster dimension.

3.1.2.2 Blocks as Clusters

When a kernel is defined with the __cluster_dims__ annotation, the number of clusters in the grid is implicit and can be calculated from the size of the grid divided into the specified cluster size.

```
__cluster_dims__((2, 2, 2)) __global__ void foo();

// 8x8x8 clusters each with 2x2x2 thread blocks.
foo<<<dim3(16, 16, 16), dim3(1024, 1, 1)>>>();
```

In the above example, the kernel is launched as a grid of 16x16x16 thread blocks, which means a grid of of 8x8x8 clusters is used.

A kernel can alternatively use the __block_size__ annotation, which specifies both the required block size and cluster size at the time the kernel is defined. When this annotation is used, the triple chevron launch becomes the grid dimension in terms of clusters rather than thread blocks, as shown below.

```
// Implementation detail of how many threads per block and blocks per cluster
// is handled as an attribute of the kernel.
__block_size__((1024, 1, 1), (2, 2, 2)) __global__ void foo();

// 8x8x8 clusters.
foo<<<dim3(8, 8, 8)>>>();
```

__block_size__ requires two fields each being a tuple of 3 elements. The first tuple denotes block dimension and second cluster size. The second tuple is assumed to be (1,1,1) if it's not passed. To specify the stream, one must pass 1 and 0 as the second and third arguments within <<<>>> and lastly the stream. Passing other values would lead to undefined behavior.

Note that it is illegal for the second tuple of __block_size__ and __cluster_dims__ to be specified at the same time. It's also illegal to use __block_size__ with an empty __cluster_dims__. When the second tuple of __block_size__ is specified, it implies the "Blocks as Clusters" being enabled and the compiler would recognize the first argument inside <<<>>> as the number of clusters instead of thread blocks.

3.1.3. More on Streams and Events

CUDA Streams introduced the basics of CUDA streams. By default, operations submitted on a given CUDA stream are serialized: one cannot start executing until the previous one has completed. The only exception is the recently added *Programmatic Dependent Launch and Synchronization* feature. Having multiple CUDA streams is a way to enable concurrent execution; another way is using *CUDA Graphs*. The two approaches can also be combined.

Work submitted on different CUDA streams *may* execute concurrently under specific circumstances, e.g., if there are no event dependencies, if there is no implicit synchronization, if there are sufficient resources, etc.

Independent operations from different CUDA streams cannot run concurrently if any CUDA operation on the NULL stream is submitted in between them, unless the streams are non-blocking CUDA streams. These are streams created with `cudaStreamCreateWithFlags()` runtime API with the `cudaStreamNonBlocking` flag. To improve potential for concurrent GPU work execution, it is recommended that the user creates non-blocking CUDA streams.

It is also recommended that the user selects the least general synchronization option that is sufficient for their problem. For example, if the requirement is for the CPU to wait (block) for all work on a specific CUDA stream to complete, using `cudaStreamSynchronize()` for that stream would be preferable to `cudaDeviceSynchronize()`, as the latter would unnecessarily wait for GPU work on all CUDA streams of the device to complete. And if the requirement is for the CPU to wait without blocking, then using `cudaStreamQuery()` and checking its return value, in a polling loop, may be preferable.

A similar synchronization effect can also be achieved with CUDA events (*CUDA Events*), e.g., by recording an event on that stream and calling `cudaEventSynchronize()` to wait, in a blocking manner, for the work captured in that event to complete. Again, this would be preferable and more focused than using `cudaDeviceSynchronize()`. Calling `cudaEventQuery()` and checking its return value, e.g., in a polling loop, would be a non-blocking alternative.

The choice of the explicit synchronization method is particularly important if this operation happens in the application's critical path. Table 4 provides a high-level summary of various synchronization options with the host.

Table 4: Summary of explicit synchronization options with the host

	Wait for specific stream	**Wait for specific event**	**Wait for everything on the device**
Non-blocking (would need a polling loop)	cudaStream-Query()	cudaEvent-Query()	N/A
Blocking	cudaStreamSyn-chronize()	cudaEventSyn-chronize()	cudaDeviceSynchronize()

For synchronization, i.e., to express dependencies, between CUDA streams, use of non-timing CUDA events is recommended, as described in *CUDA Events*. A user can call `cudaStreamWaitEvent()` to force future submitted operations on a specific stream to wait for the completion of a previously recorded event (e.g., on another stream). Note that for any CUDA API waiting or querying an event, it is the responsibility of the user to ensure the cudaEventRecord API has been already called, as a non-recorded event will always return success.

CUDA events carry, by default, timing information, as they can be used in `cudaEventElapsedTime()` API calls. However, a CUDA event that is solely used to express dependencies across streams does not need timing information. For such cases, it is recommended to create events with timing information disabled for improved performance. This is possible using `cudaEventCreateWithFlags()` API with the `cudaEventDisableTiming` flag.

3.1.3.1 Stream Priorities

The relative priorities of streams can be specified at creation time using `cudaStreamCreateWithPriority()`. The range of allowable priorities, ordered as [greatest priority, least priority] can be obtained using the `cudaDeviceGetStreamPriorityRange()` function. At runtime, the GPU scheduler utilizes stream priorities to determine task execution order, but these priorities serve as hints rather than guarantees. When selecting work to launch, pending tasks in higher-priority streams take precedence over those in lower-priority streams. Higher-priority tasks do not preempt already running lower-priority tasks. The GPU does not reassess work queues during task execution, and increasing a stream's priority will not interrupt ongoing work. Stream priorities influence task execution without enforcing strict ordering, so users can leverage stream priorities to influence task execution without relying on strict ordering guarantees.

The following code sample obtains the allowable range of priorities for the current device, and creates two non-blocking CUDA streams with the highest and lowest available priorities.

```
// get the range of stream priorities for this device
int leastPriority, greatestPriority;
cudaDeviceGetStreamPriorityRange(&leastPriority, &greatestPriority);

// create streams with highest and lowest available priorities
cudaStream_t st_high, st_low;
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking,
→greatestPriority));
cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking, leastPriority);
```

3.1.3.2 Explicit Synchronization

As previously outlined, there are a number of ways that streams can synchronize with other streams. The following provides common methods at different levels of granularity: - `cudaDeviceSynchronize()` waits until all preceding commands in all streams of all host threads have completed. - `cudaStreamSynchronize()`takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device. - `cudaStreamWaitEvent()`takes a stream and an event as parameters (see *CUDA Events* for a description of events) and makes all the commands added to the given stream after the call to `cudaStreamWaitEvent()`delay their execution until the given event has completed. - `cudaStreamQuery()`provides applications with a way to know if all preceding commands in a stream have completed.

3.1.3.3 Implicit Synchronization

Two commands from different streams cannot run concurrently if any one of the following operations is issued in-between them by the host thread:

- a page-locked host memory allocation
- a device memory allocation
- a device memory set
- a memory copy between two addresses to the same device memory
- any CUDA command to the NULL stream
- a switch between the L1/shared memory configurations

Operations that require a dependency check include any other commands within the same stream as the launch being checked and any call to `cudaStreamQuery()` on that stream. Therefore, applications should follow these guidelines to improve their potential for concurrent kernel execution:

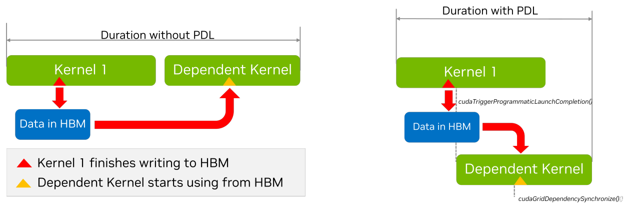
- All independent operations should be issued before dependent operations,
- Synchronization of any kind should be delayed as long as possible.

3.1.4. Programmatic Dependent Kernel Launch

As we have discussed earlier, the semantics of CUDA Streams are such that kernels execute in order. This is so that if we have two successive kernels, where the second kernel depends on results from the first one, the programmer can be safe in the knowledge that by the time the second kernel starts executing the dependent data will be available. However, it may be the case that the first kernel can have the data on which a subsequent kernel depends already written to global memory and it still has more work to do. Likewise the dependent second kernel may have some independent work before it needs the data from the first kernel. In such a situation it is possible to partially overlap the execution of the two kernels (assuming that hardware resources are available). The overlapping can also overlap the launch overheads of the second kernel too. Other than the availability of hardware resources,the degree of overlap which can be achieved is dependent on the specific structure of the kernels, such as

- when in its execution does the first kernel finish the work on which the second kernel depends?
- when in its execution does the second kernel start working on the data from the first kernel?

since this is very much dependent on the specific kernels in question it is difficult to automate completely and hence CUDA provides a mechanism to allow the application developer to specify the synchronization point between the two kernels. This is done via a technique known as Programmatic Dependent Kernel Launch. The situation is depicted in the figure below.


Duration without PDL
Kernel 1
Dependent Kernel
Data in HBM
Kernel 1 finishes writing to HBM
Dependent Kernel starts using from HBM
Duration with PDL
Kernel 1
cudaTriggerProgrammaticLaunchCompletion()
Data in HBM
Dependent Kernel
cudaGridDependencySynchronize()

PDL has three main components.

i) The first kernel (the so called *primary kernel*) needs to call a special function to indicate that it is done with the everything that the subsequent dependent kernels (also called *secondary kernel*) will need. This is done by calling the function `cudaTriggerProgrammaticLaunchCompletion()`.

ii) In turn, the dependent secondary kernel needs to indicate that it has reached the portion of the its work which is independent of the primary kernel and that it is now waiting on the primary kernel to finish the work on which it depends. This is done with the function `cudaGridDependencySynchronize()`.

iii) THe second kernel needs to be launched with a special attribute *cudaLaunchAttributeProgrammaticStreamSerialization* with its *programmaticStreamSerializationAllowed* field set to '1'.

The following code snippet shows an example of how this can be done.

Listing 3: Example of Programmatic Dependent Kernel Launch with two Kernels

```
__global__ void primary_kernel() {
    // Initial work that should finish before starting secondary kernel

    // Trigger the secondary kernel
    cudaTriggerProgrammaticLaunchCompletion();

    // Work that can coincide with the secondary kernel
}

__global__ void secondary_kernel()
{
    // Initialization, Independent work, etc.

    // Will block until all primary kernels the secondary kernel is dependent
↪on have
    // completed and flushed results to global memory
    cudaGridDependencySynchronize();

    // Dependent work
}

// Launch the secondary kernel with the special attribute

// Set Up the attribute
cudaLaunchAttribute attribute[1];
attribute[0].id = cudaLaunchAttributeProgrammaticStreamSerialization;
attribute[0].val.programmaticStreamSerializationAllowed = 1;

// Set the attribute in a kernel launch configuration
 cudaLaunchConfig_t config = {0};

// Base launch configuration
config.gridDim = grid_dim;
config.blockDim = block_dim;
config.dynamicSmemBytes= 0;
config.stream = stream;

// Add special attribute for PDL
config.attrs = attribute;
config.numAttrs = 1;

// Launch primary kernel
primary_kernel<<<grid_dim, block_dim, 0, stream>>>();

// Launch secondary (dependent) kernel using the configuration with
// the attribute
cudaLaunchKernelEx(&config, secondary_kernel);
```

3.1.5. Batched Memory Transfers

A common pattern in CUDA development is to use a technique of batching. By batching we loosely mean that we have several (typically small)tasks grouped together into a single (typically bigger) operation. The components of the batch do not necessarily all have to be identical although they often are. An example of this idea is the batch matrix multiplication operation provided by cuBLAS.

Generally as with CUDA Graphs, and PDL, the purpose of batching is to reduce overheads associated with dispatching the individual batch tasks separately. In terms of memory transfers launching a memory transfer can incur some CPU and driver overheads. Further, the regular `cudaMemcpyAsync()` function in its current form does not necessarily provide enough information for the driver to optimize the transfer, for example, in terms of hints about the source and destination. On Tegra platforms one has the choice of using SMs or Copy Engines (CEs)o perform transfers. The choice of which is currently specified by a heuristic in the driver. This can be important because using the SMs may result in a faster transfer, however it ties down some of the available compute power. On the other hand, using the CEs may result in a slower transfer but overall higher application performance, since it leaves the SMs free to perform other work.

These considerations motivated the design of the `cudaMemcpyBatchAsync()` function (and its relative `cudaMemcpyBatch3DAsync()`). These functions allow batched memory transfers to be optimized. Apart from the lists of source and destination pointers, the API uses memory copy attributes to specify expectations of orderings, with hints for source and destination locations, as well as for whether one prefers to overlap the transfer with compute (something that is currently only supported on Tegra platforms with CEs).

Let us first consider the simplest case of a simple batch transfer of data from pinned host memory to pinned device memory

Listing 4: Example of Homogeneous Batched Memory Transfer from Pinned Host Memory to Pinned Device Memory

```
std::vector<void *> srcs(batch_size);
std::vector<void *> dsts(batch_size);
std::vector<void *> sizes(batch_size);

// Allocate the source and destination buffers
// initialize with the stream number
for (size_t i = 0; i < batch_size; i++) {
    cudaMallocHost(&srcs[i], sizes[i]);
    cudaMalloc(&dsts[i], sizes[i]);
    cudaMemsetAsync(srcs[i], sizes[i], stream);
}

// Setup attributes for this batch of copies
cudaMemcpyAttributes attrs = {};
attrs.srcAccessOrder = cudaMemcpySrcAccessOrderStream;

// All copies in the batch have same copy attributes.
size_t attrsIdxs = 0;  // Index of the attributes

// Launch the batched memory transfer
cudaMemcpyBatchAsync(&dsts[0], &srcs[0], &sizes[0], batch_size,
    &attrs, &attrsIdxs, 1 /*numAttrs*/, nullptr /*failIdx*/, stream);
```

The first few parameters to the `cudaMemcpyBatchAsync()` function seem immediately sensible. The are comprised of arrays containing the source and destination pointers, as well as the transfer sizes.

Each array has to have``batch_size`` elements. The new information comes from the attributes. The function needs a pointer to an array of attributes, and a corresponding array of attribute indices. In principle it is also possible to pass an array of `size_t` and in this array the indices of an failed transfers can be recorded, however it is safe to pass a `nullptr` here, in this case the indices of failures will simply not be recorded.

Turning to the attributes, in this instance the transfers are homogeneous. So we use only one attribute, which will apply to all the transfers. This is controlled by the *attrIndex* parameter. In principle this can be an array. Element *i* of the array contains the index of the first transfer to which the *i*-th element of the attribute array applies. In this case, *attrIndex* is treated as a single element array, with the value '0' meaning that `attribute[0]` will apply to all transfers with index 0 and up, in other words all the transfers.

Finally, we note that we have set the the `srcAccessOrder` attribute to `cudaMemcpySrcAccessOrderStream`. This means that the source data will be accessed in regular stream order. In other words, the memcpy will block until previous kernels dealing with the data from any of these source and destination pointers are completed.

In the next example we will consider a more complex case of a heterogeneous batch transfer.

Listing 5: Example of Heterogeneous Batched Memory Transfer using some Ephemeral Host Memory to Pinned Device Memory

```
std::vector<void *> srcs(batch_size);
std::vector<void *> dsts(batch_size);
std::vector<void *> sizes(batch_size);

// Allocate the src and dst buffers
for (size_t i = 0; i < batch_size - 10; i++) {
    cudaMallocHost(&srcs[i], sizes[i]);
    cudaMalloc(&dsts[i], sizes[i]);
}

int buffer[10];

for (size_t i = batch_size - 10; i < batch_size; i++) {
    srcs[i] = &buffer[10 - (batch_size - i];
    cudaMalloc(&dsts[i], sizes[i]);
}

// Setup attributes for this batch of copies
cudaMemcpyAttributes attrs[2] = {};
attrs[0].srcAccessOrder = cudaMemcpySrcAccessOrderStream;
attrs[1].srcAccessOrder = cudaMemcpySrcAccessOrderDuringApiCall;

size_t attrsIdxs[2];
attrsIdxs[0] = 0;
attrsIdxs[1] = batch_size - 10;

// Launch the batched memory transfer
cudaMemcpyBatchAsync(&dsts[0], &srcs[0], &sizes[0], batch_size,
    &attrs, &attrsIdxs, 2 /*numAttrs*/, nullptr /*failIdx*/, stream);
```

Here we have two kinds of transfers: `batch_size-10` transfer from pinned host memory to pinned device memory, and 10 transfers from a host array to pinned device memory. Further, the *buffer* array is not only on the host but is only in existence in the current scope – its address is what is known as an *ephemeral pointer*. This pointer may not be valid after the API call completes (it is asynchronous). To perform the copies with such ephemeral pointers, the *srcAccessOrder* in the attribute must be set to *cudaMemcpySrcAccessOrderDuringApiCall*.

We now have two attributes, the first one applies to all transfers with indices starting at 0, and less than `batch_size-10`. The second one applies to all transfers with indices starting at `batch_size-10` and less than `batch_size`.

If instead of allocating the *buffer* array from the stack, we had allocated it from the heap using *malloc* the data would not be ephemeral any more. It would be valid until the pointer was explicitly freed. In such a case the best option for how to stage the copies would depend on whether the system had hardware managed memory or coherent GPU access to host memory via address translation in which case it would be best to use stream ordering, or whether it did not in which case staging the transfers immediately would make most sense. In this situation, one should use the value `cudaMemcpyAccessOrderAny` for the `srcAccessOrder` of the attribute.

The `cudaMemcpyBatchAsync` function also allows the programmer to provide hints about the source and destination locations. This is done by setting the `srcLocation` and `dstLocation` fields of the `cudaMemcpyAttributes` structure. The `srcLocation``and ``dstLocation` fields are both of type `cudaMemLocation` which is a structure that contains the type of the location and the ID of the location. This is the same `cudaMemLocation` structure that can be used to give prefetching hints to the runtime when using `cudaMemPrefetchAsync()`. We illustrate how to set up the hints for a transfer from the device, to a specific NUMA node of the host in the code example below:

Listing 6: Example of Setting Source and Destination Location Hints

```
// Allocate the source and destination buffers
std::vector<void *> srcs(batch_size);
std::vector<void *> dsts(batch_size);
std::vector<void *> sizes(batch_size);

// cudaMemLocation structures we will use tp provide location hints
// Device device_id
cudaMemLocation srcLoc = {cudaMemLocationTypeDevice, dev_id};

// Host with numa Node numa_id
cudaMemLocation dstLoc = {cudaMemLocationTypeHostNuma, numa_id};

// Allocate the src and dst buffers
for (size_t i = 0; i < batch_size; i++) {
    cudaMallocManaged(&srcs[i], sizes[i]);
    cudaMallocManaged(&dsts[i], sizes[i]);

    cudaMemPrefetchAsync(srcs[i], sizes[i], srcLoc, 0, stream);
    cudaMemPrefetchAsync(dsts[i], sizes[i], dstLoc, 0, stream);
    cudaMemsetAsync(srcs[i], sizes[i], stream);
}

// Setup attributes for this batch of copies
cudaMemcpyAttributes attrs = {};

// These are managed memory pointers so Stream Order is appropriate
attrs.srcAccessOrder = cudaMemcpySrcAccessOrderStream;
```

(continues on next page)

(continued from previous page)

```
// Now we can specify the location hints here.
attrs.srcLocHint = srcLoc;
attrs.dstlocHint = dstLoc;

// All copies in the batch have same copy attributes.
size_t attrsIdxs = 0;

// Launch the batched memory transfer
cudaMemcpyBatchAsync(&dsts[0], &srcs[0], &sizes[0], batch_size,
    &attrs, &attrsIdxs, 1 /*numAttrs*/, nullptr /*failIdx*/, stream);
```

THe last thing to cover is the flag for hinting whether we want to use SM's or CEs for the transfers. The field for this is the `cudaMemcpyAttributesflags::flags` and the possible values are:

- `cudaMemcpyFlagDefault` – default behavior
- `cudaMemcpyFlagPreferOverlapWithCompute` – this hints that the system should prefer to use CEs for the transfers overlapping the transfer with computations. This flag is ignored on non-Tegra platforms

In summary, the main points regarding "cudaMemcpyBatchAsync" are as follows:

- The `cudaMemcpyBatchAsync` function (and its 3D variant) allows the programmer to specify a batch of memory transfers, allowing the amortization of transfer setup overheads.
- Other than the source and destination pointers and the transfer sizes, the function can take one or more memory copy attributes providing information about the kind of memory being transferred and the corresponding stream ordering behavior of the source pointers, hints about the source and destination locations, and hints as to whether to prefer to overlap the transfer with compute (if possible) or whether to use SMs for the transfer.
- Given the above information the runtime can attempt to optimize the transfer to the maximum degree possible..

3.1.6. Environment Variables

CUDA provides various environment variables (see Section 5.2), which can affect execution and performance. If they are not explicitly set, CUDA uses reasonable default values for them, but special handling may be required on a per-case basis, e.g., for debugging purposes or to get improved performance.

For example, increasing the value of the `CUDA_DEVICE_MAX_CONNECTIONS` environment variable may be necessary to reduce the possibility that independent work from different CUDA streams gets serialized due to false dependencies. Such false dependencies may be introduced when the same underlying resource(s) are used. It is recommended to start by using the default value and only explore the impact of this environment variable in case of performance issues (e.g., unexpected serialization of independent work across CUDA streams that cannot be attributed to other factors like lack of available SM resources). Worth noting that this environment variable has a different (lower) default value in case of MPS.

Similarly, setting the `CUDA_MODULE_LOADING` environment variable to `EAGER` may be preferable for latency-sensitive applications, in order to move all overhead due to module loading to the application initialization phase and outside its critical phase. The current default mode is lazy module loading. In this default mode, a similar effect to eager module loading could be achieved by adding "warm-up" calls of the various kernels during the application's initialization phase, to force module loading to happen sooner.

Please refer to *CUDA Environment Variables* for more details about the various CUDA environment variables. It is recommended that you set the environment variables to new values *before* you launch the application; attempting to set them within your application may have no effect.

3.2. Advanced Kernel Programming

This chapter will first take a deeper dive into the hardware model of NVIDIA GPUs, and then introduce some of the more advanced features available in CUDA kernel code aimed at improving kernel performance. This chapter will introduce some concepts related to thread scopes, asynchronous execution, and the associated synchronization primitives. These conceptual discussions provide a necessary foundation for some of the advanced performance features available within kernel code.

Detailed descriptions for some of these features are contained in chapters dedicated to the features in the next part of this programming guide.

- *Advanced synchronization primitives* introduced in this chapter, are covered completely in Section 4.9 and Section 4.10.
- *Asynchronous data copies*, including the tensor memory accelerator (TMA), are introduced in this chapter and covered completely in Section 4.11.

3.2.1. Using PTX

Parallel Thread Execution (PTX), the virtual machine instruction set architecture (ISA) that CUDA uses to abstract hardware ISAs, was introduced in Section 1.3.3. Writing code in PTX directly is a highly advanced optimization technique that is not necessary for most developers and should be considered a tool of last resort. Nevertheless, there are situations where the fine-grained control enabled by writing PTX directly enables performance improvements in specific applications. These situations are typically in very performance-sensitive portions of an application where every fraction of a percent of performance improvement has significant benefits. All of the available PTX instructions are in the PTX ISA document.

`cuda::ptx` namespace

One way to use PTX directly in your code is to use the `cuda::ptx` namespace from libcu++. This namespace provides C++ functions that map directly to PTX instructions, simplifying their use within a C++ application. For more information, please refer to the cuda::ptx namespace documentation.

Inline PTX

Another way to include PTX in your code is to use inline PTX. This method is described in detail in the corresponding documentation. This is very similar to writing assembly code on a CPU.

3.2.2. Hardware Implementation

A streaming multiprocessor or SM (see *GPU Hardware Model*) is designed to execute hundreds of threads concurrently. To manage such a large number of threads, it employs a unique parallel computing model called *Single-Instruction, Multiple-Thread*, or *SIMT*, that is described in *SIMT Execution Model*. The instructions are pipelined, leveraging instruction-level parallelism within a single thread, as well as extensive thread-level parallelism through simultaneous hardware multithreading as detailed in *Hardware Multithreading*. Unlike CPU cores, SMs issue instructions in order and do not perform branch prediction or speculative execution.

Sections *SIMT Execution Model* and *Hardware Multithreading* describe the architectural features of the SM that are common to all devices. Section *Compute Capabilities* provides the specifics for devices of different compute capabilities.

The NVIDIA GPU architecture uses a little-endian representation.

3.2.2.1 SIMT Execution Model

Each SM creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term *warp* originates from weaving, the first parallel thread technology. A *half-warp* is either the first or second half of a warp. A *quarter-warp* is either the first, second, third, or fourth quarter of a warp.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp executes each branch path taken, disabling threads that are not on that path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines: the cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

3.2.2.1.1 Independent Thread Scheduling

On GPUs with compute capability lower than 7.0, warps used a single program counter shared amongst all 32 threads in the warp together with an active mask specifying the active threads of the warp. As a result, threads from the same warp in divergent regions or different states of execution cannot signal each other or exchange data, and algorithms requiring fine-grained sharing of data guarded by locks or mutexes can lead to deadlock, depending on which warp the contending threads come from.

In GPUs of compute capability 7.0 and later, *independent thread scheduling* allows full concurrency between threads, regardless of warp. With independent thread scheduling, the GPU maintains execution state per thread, including a program counter and call stack, and can yield execution at a per-thread granularity, either to make better use of execution resources or to allow one thread to wait for data to be produced by another. A schedule optimizer determines how to group active threads from the same warp together into SIMT units. This retains the high throughput of SIMT execution as in prior NVIDIA GPUs, but with much more flexibility: threads can now diverge and reconverge at sub-warp granularity.

Independent thread scheduling can break code that relies on implicit warp-synchronous behavior from previous GPU architectures. *Warp-synchronous* code assumes that threads in the same warp execute in lockstep at every instruction, but the ability for threads to diverge and reconverge at sub-warp granularity makes such assumptions invalid. This can lead to a different set of threads participating in the executed code than intended. Any warp-synchronous code developed for GPUs prior to CC 7.0 (such as synchronization-free intra-warp reductions) should be revisited to ensure compatibility. Developers should explicitly synchronize such code using `__syncwarp()` to ensure correct behavior across all GPU generations.

Note

The threads of a warp that are participating in the current instruction are called the *active* threads, whereas threads not on the current instruction are *inactive* (disabled). Threads can be inactive for a variety of reasons including having exited earlier than other threads of their warp, having taken a different branch path than the branch path currently executed by the warp, or being the last threads of a block whose number of threads is not a multiple of the warp size.

If a non-atomic instruction executed by a warp writes to the same location in global or shared memory from more than one of the threads of the warp, the number of serialized writes that occur to that location may vary depending on the compute capability of the device. However, for all compute capabilities, which thread performs the final write is undefined.

If an *atomic* instruction executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read/modify/write to that location occurs and they are all serialized, but the order in which they occur is undefined.

3.2.2.2 Hardware Multithreading

When an SM is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled for execution by a *warp scheduler*. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. *Thread Hierarchy* describes how thread IDs relate to thread indices in the block.

The total number of warps in a block is defined as follows:

ceil $\left(\frac{T}{W_{size}}, 1\right)$

- *T* is the number of threads per block,
- *Wsize* is the warp size, which is equal to 32,
- ceil(x, y) is equal to x rounded up to the nearest multiple of y.

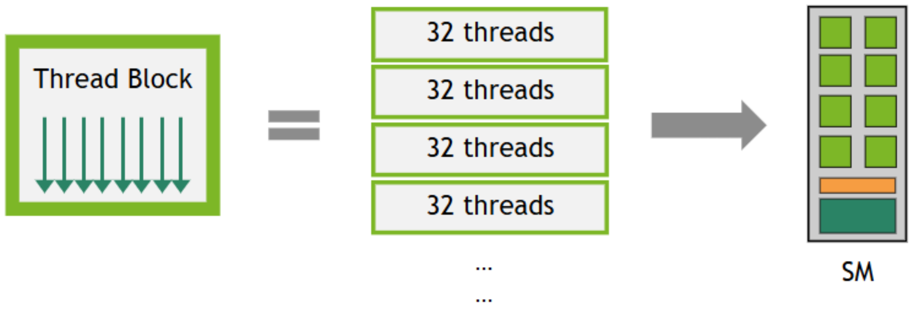


Figure 19: A thread block is partitioned into warps of 32 threads.

The execution context (program counters, registers, etc.) for each warp processed by an SM is maintained on-chip throughout the warp's lifetime. Therefore, switching between warps incurs no cost. At each instruction issue cycle, a warp scheduler selects a warp with threads ready to execute its next instruction (the *active threads* of the warp) and issues the instruction to those threads.

Each SM has a set of 32-bit registers that are partitioned among the warps, and a *shared memory* that is partitioned among the thread blocks. The number of blocks and warps that can reside and be processed concurrently on the SM for a given kernel depends on the amount of registers and shared memory used by the kernel, as well as the amount of registers and shared memory available on the SM. There are also a maximum number of resident blocks and warps per SM. These limits, as well the amount of registers and shared memory available on the SM, depend on the compute capability of the device and are specified in *Compute Capabilities*. If there are not enough resources available per SM to process at least one block, the kernel will fail to launch. The total number of registers and shared memory allocated for a block can be determined in several ways documented in the *Occupancy* section.

3.2.2.3 Asynchronous Execution Features

Recent NVIDIA GPU generations have included asynchronous execution capabilities to allow more overlap of data movement, computation, and synchronization within the GPU. These capabilities enable certain operations invoked from GPU code to execute asynchronously to other GPU code in the same thread block. This asynchronous execution should not be confused with asynchronous CUDA APIs discussed in Section 2.3, which enable GPU kernel launches or memory operations to operate asynchronously to each other or to the CPU.

Compute capability 8.0 (The NVIDIA Ampere GPU Architecture) introduced hardware-accelerated asynchronous data copies from global to shared memory and asynchronous barriers (see NVIDIA A100 Tensor Core GPU Architecture).

Compute capability 9.0 (The NVIDIA Hopper GPU architecture) extended the asynchronous execution features with the *Tensor Memory Accelerator (TMA)* unit, which can transfer large blocks of data and multidimensional tensors from global memory to shared memory and vice versa, asynchronous transaction barriers, and asynchronous matrix multiply-accumulate operations (see Hopper Architecture in Depth blog post for details.).

CUDA provides APIs which can be called by threads from device code to use these features. The asynchronous programming model defines the behavior of asynchronous operations with respect to CUDA threads.

An asynchronous operation is an operation initiated by a CUDA thread, but executed asynchronously as if by another thread, which we will refer to as an *async thread*. In a well-formed program, one or more CUDA threads synchronize with the asynchronous operation. The CUDA thread that initiated the asynchronous operation is not required to be among the synchronizing threads. The async thread is always associated with the CUDA thread that initiated the operation.

An asynchronous operation uses a synchronization object to signal its completion, which could be a barrier or a pipeline. These synchronization objects are explained in detail in *Advanced Synchronization Primitives*, and their role in performing asynchronous memory operations is demonstrated in *Asynchronous Data Copies*.

3.2.2.3.1 Async Thread and Async Proxy

Asynchronous operations may access memory differently than regular operations. To distinguish between these different memory access methods, CUDA introduces the concepts of an *async thread*, a *generic proxy*, and an *async proxy*. Normal operations (loads and stores) go through the generic proxy. Some asynchronous instructions, such as *LDGSTS* and *STAS/REDAS*, are modeled using an async thread operating in the generic proxy. Other asynchronous instructions, such as bulk-asynchronous copies with TMA and some tensor core operations (tcgen05.*, wgmma.mma_async.*), are modeled using an async thread operating in the async proxy.

Async thread operating in generic proxy. When an asynchronous operation is initiated, it is associated with an async thread, which is different from the CUDA thread that initiated the operation. *Preceding* generic proxy (normal) loads and stores to the same address are guaranteed to be ordered before the asynchronous operation. However, *subsequent* normal loads and stores to the same address are not guaranteed to maintain their ordering, potentially incurring a race condition until the async thread completes.

Async thread operating in async proxy. When an asynchronous operation is initiated, it is associated with an async thread, which is different from the CUDA thread that initiated the operation. *Prior and subsequent* normal loads and stores to the same address are not guaranteed to maintain their ordering. A proxy fence is required to synchronize them across the different proxies to ensure proper memory ordering. Section *Using the Tensor Memory Accelerator (TMA)* demonstrates use of proxy fences to ensure correctness when performing asynchronous copies with TMA.

For more details on these concepts, see the PTX ISA documentation.

3.2.3. Thread Scopes

CUDA threads form a *Thread Hierarchy*, and using this hierarchy is essential for writing both correct and performant CUDA kernels. Within this hierarchy, the visibility and synchronization scope of memory operations can vary. To account for this non-uniformity, the CUDA programming model introduces the concept of *thread scopes*. A thread scope defines which threads can observe a thread's loads and stores and specifies which threads can synchronize with each other using synchronization primitives such as atomic operations and barriers. Each scope has an associated point of coherency in the memory hierarchy.

Thread scopes are exposed in CUDA PTX and are also available as extensions in the libcu++ library. The following table defines the thread scopes available:

CUDA C++ Thread Scope	**CUDA PTX Thread Scope**	**Description**	**Point of Coherency in Memory Hierarchy**
`cuda::thread_`		Memory operations are visible only to the local thread.	–
`cuda::thread_`	`.cta`	Memory operations are visible to other threads in the same thread block.	L1
	`.cluster`	Memory operations are visible to other threads in the same thread block cluster.	L2
`cuda::thread_`	`.gpu`	Memory operations are visible to other threads in the same GPU device.	L2
`cuda::thread_`	`.sys`	Memory operations are visible to other threads in the same system (CPU, other GPUs).	L2 + connected caches

Sections *Advanced Synchronization Primitives* and *Asynchronous Data Copies* demonstrate use of thread scopes.

3.2.4. Advanced Synchronization Primitives

This section introduces three families of synchronization primitives:

- *Scoped Atomics*, which pair C++ memory ordering with CUDA thread scopes to safely communicate across threads at block, cluster, device, or system scope (see *Thread Scopes*).
- *Asynchronous Barriers*, which split synchronization into arrival and wait phases, and can be used to track the progress of asynchronous operations.
- *Pipelines*, which stage work and coordinate multi-buffer producer–consumer patterns, commonly used to overlap compute with *asynchronous data copies*.

3.2.4.1 Scoped Atomics

Section 5.4.5 gives an overview of atomic functions available in CUDA. In this section, we will focus on *scoped* atomics that support C++ standard atomic memory semantics, available through the libcu++ library or through compiler built-in functions. Scoped atomics provide the tools for efficient synchronization at the appropriate level of the CUDA thread hierarchy, enabling both correctness and performance in complex parallel algorithms.

3.2.4.1.1 Thread Scope and Memory Ordering

Scoped atomics combine two key concepts:

- **Thread Scope**: defines which threads can observe the effect of the atomic operation (see *Thread Scopes*).
- **Memory Ordering**: defines the ordering constraints relative to other memory operations (see C++ standard atomic memory semantics).

CUDA C++ `cuda::atomic`

```
#include <cuda/atomic>

__global__ void block_scoped_counter() {
    // Shared atomic counter visible only within this block
    __shared__ cuda::atomic<int, cuda::thread_scope_block> counter;

    // Initialize counter (only one thread should do this)
    if (threadIdx.x == 0) {
        counter.store(0, cuda::memory_order_relaxed);
    }
    __syncthreads();

    // All threads in block atomically increment
    int old_value = counter.fetch_add(1, cuda::memory_order_relaxed);

    // Use old_value...
}
```

Built-in Atomic Functions

```
__global__ void block_scoped_counter() {
    // Shared counter visible only within this block
    __shared__ int counter;

    // Initialize counter (only one thread should do this)
    if (threadIdx.x == 0) {
        __nv_atomic_store_n(&counter, 0,
                            __NV_ATOMIC_RELAXED,
                            __NV_THREAD_SCOPE_BLOCK);
    }
    __syncthreads();

    // All threads in block atomically increment
    int old_value = __nv_atomic_fetch_add(&counter, 1,
                                          __NV_ATOMIC_RELAXED,
                                          __NV_THREAD_SCOPE_BLOCK);

    // Use old_value...
}
```

This example implements a *block-scoped atomic counter* that demonstrates the fundamental concepts of scoped atomics:

- **Shared Variable**: a single counter is shared among all threads in the block using `__shared__` memory.
- **Atomic Type Declaration**: `cuda::atomic<int, cuda::thread_scope_block>` creates an atomic integer with block-level visibility.
- **Single Initialization**: only thread 0 initializes the counter to prevent race conditions during setup.
- **Block Synchronization**: `__syncthreads()` ensures all threads see the initialized counter before proceeding.
- **Atomic Increment**: each thread atomically increments the counter and receives the previous value.

`cuda::memory_order_relaxed` is chosen here because we only need atomicity (indivisible read-modify-write) without ordering constraints between different memory locations. Since this is a straightforward counting operation, the order of increments doesn't matter for correctness.

For producer-consumer patterns, acquire-release semantics ensure proper ordering:

CUDA C++ `cuda::atomic`

```
__global__ void producer_consumer() {
    __shared__ int data;
    __shared__ cuda::atomic<bool, cuda::thread_scope_block> ready;

    if (threadIdx.x == 0) {
        // Producer: write data then signal ready
        data = 42;
        ready.store(true, cuda::memory_order_release);  // Release ensures
↪data write is visible
```

(continues on next page)

(continued from previous page)

```
    } else {
        // Consumer: wait for ready signal then read data
        while (!ready.load(cuda::memory_order_acquire)) {  // Acquire ensures
↪data read sees the write
            // spin wait
        }
        int value = data;
        // Process value...
    }
}
```

Built-in Atomic Functions

```
__global__ void producer_consumer() {
    __shared__ int data;
    __shared__ bool ready; // Only ready flag needs atomic operations

    if (threadIdx.x == 0) {
        // Producer: write data then signal ready
        data = 42;
        __nv_atomic_store_n(&ready, true,
                            __NV_ATOMIC_RELEASE,
                            __NV_THREAD_SCOPE_BLOCK);  // Release ensures
↪data write is visible
    } else {
        // Consumer: wait for ready signal then read data
        while (!__nv_atomic_load_n(&ready,
                                   __NV_ATOMIC_ACQUIRE,
                                   __NV_THREAD_SCOPE_BLOCK)) {  // Acquire
↪ensures data read sees the write
            // spin wait
        }
        int value = data;
        // Process value...
    }
}
```

3.2.4.1.2 Performance Considerations

- *Use the narrowest scope possible*: block-scoped atomics are much faster than system-scoped atomics.
- *Prefer weaker orderings*: use stronger orderings only when necessary for correctness.
- *Consider memory location*: shared memory atomics are faster than global memory atomics.

3.2.4.2 Asynchronous Barriers

An asynchronous barrier differs from a typical single-stage barrier (`__syncthreads()`) in that the notification by a thread that it has reached the barrier (the "arrival") is separated from the operation of waiting for other threads to arrive at the barrier (the "wait"). This separation increases execution efficiency by allowing a thread to perform additional operations unrelated to the barrier, making more efficient use of the wait time. Asynchronous barriers can be used to implement producer-consumer patterns with CUDA threads or enable asynchronous data copies within the memory hierarchy by having the copy operation signal ("arrive on") a barrier upon completion.

Asynchronous barriers are available on devices of compute capability 7.0 or higher. Devices of compute capability 8.0 or higher provide hardware acceleration for asynchronous barriers in shared-memory and a significant advancement in synchronization granularity, by allowing hardware-accelerated synchronization of any subset of CUDA threads within the block. Previous architectures only accelerate synchronization at a whole-warp (`__syncwarp()`) or whole-block (`__syncthreads()`) level.

The CUDA programming model provides asynchronous barriers via `cuda::std::barrier`, an ISO C++-conforming barrier available in the libcu++ library. In addition to implementing std::barrier, the library offers CUDA-specific extensions to select a barrier's thread scope to improve performance and exposes a lower-level cuda::ptx API. A `cuda::barrier` can interoperate with `cuda::ptx` by using the `friend` function `cuda::device::barrier_native_handle()` to retrieve the barrier's native handle and pass it to `cuda::ptx` functions. CUDA also provides a *primitives API* for asynchronous barriers in shared memory at thread-block scope.

The following table gives an overview of asynchronous barriers available for synchronizing at different thread scopes.

Thread Scope	Memory Location	Arrive on Barrier	Wait on Barrier	Hardware-accelerated	CUDA APIs
block	local shared memory	allowed	allowed	yes (8.0+)	`cuda::barrier`, `cuda::ptx`, primitives
cluster	local shared memory	allowed	allowed	yes (9.0+)	`cuda::barrier`, `cuda::ptx`
cluster	remote shared memory	allowed	not allowed	yes (9.0+)	`cuda::barrier`, `cuda::ptx`
device	global memory	allowed	allowed	no	`cuda::barrier`
system	global/unified memory	allowed	allowed	no	`cuda::barrier`

Temporal Splitting of Synchronization

Without the asynchronous arrive-wait barriers, synchronization within a thread block is achieved using `__syncthreads()` or `block.sync()` when using *Cooperative Groups*.

```
#include <cooperative_groups.h>

__global__ void simple_sync(int iteration_count) {
    auto block = cooperative_groups::this_thread_block();

    for (int i = 0; i < iteration_count; ++i) {
        /* code before arrive */

         // Wait for all threads to arrive here.
```

(continues on next page)

(continued from previous page)

```
        block.sync();

        /* code after wait */
    }
}
```

Threads are blocked at the synchronization point (`block.sync()`) until all threads have reached the synchronization point. In addition, memory updates that happened before the synchronization point are guaranteed to be visible to all threads in the block after the synchronization point.

This pattern has three stages:

- Code **before** the sync performs memory updates that will be read **after** the sync.
- Synchronization point.
- Code **after** the sync, with visibility of memory updates that happened **before** the sync.

Using asynchronous barriers instead, the temporally-split synchronization pattern is as follows.

CUDA C++ cuda::barrier

```
#include <cuda/barrier>
#include <cooperative_groups.h>

__device__ void compute(float *data, int iteration);

__global__ void split_arrive_wait(int iteration_count, float *data)
{
  using barrier_t = cuda::barrier<cuda::thread_scope_block>;
  __shared__ barrier_t bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // Initialize barrier with expected arrival count.
    init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < iteration_count; ++i)
  {
    /* code before arrive */

    // This thread arrives. Arrival does not block a thread.
    barrier_t::arrival_token token = bar.arrive();

    compute(data, i);

    // Wait for all threads participating in the barrier to complete bar.
↪arrive().
    bar.wait(std::move(token));

    /* code after wait */
  }
}
```

CUDA C++ cuda::ptx

```
#include <cuda/ptx>
#include <cooperative_groups.h>

__device__ void compute(float *data, int iteration);

__global__ void split_arrive_wait(int iteration_count, float *data)
{
  __shared__ uint64_t bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // Initialize barrier with expected arrival count.
    cuda::ptx::mbarrier_init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < iteration_count; ++i)
  {
    /* code before arrive */

    // This thread arrives. Arrival does not block a thread.
    uint64_t token = cuda::ptx::mbarrier_arrive(&bar);

    compute(data, i);

    // Wait for all threads participating in the barrier to complete mbarrier_
↪arrive().
    while(!cuda::ptx::mbarrier_try_wait(&bar, token)) {}

    /* code after wait */
  }
}
```

CUDA C primitives

```
#include <cuda_awbarrier_primitives.h>
#include <cooperative_groups.h>

__device__ void compute(float *data, int iteration);

__global__ void split_arrive_wait(int iteration_count, float *data)
{
  __shared__ __mbarrier_t bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // Initialize barrier with expected arrival count.
    __mbarrier_init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < iteration_count; ++i)
  {
    /* code before arrive */

    // This thread arrives. Arrival does not block a thread.
    __mbarrier_token_t token = __mbarrier_arrive(&bar);

    compute(data, i);

    // Wait for all threads participating in the barrier to complete __
↪mbarrier_arrive().
    while(!__mbarrier_try_wait(&bar, token, 1000)) {}

    /* code after wait */
  }
}
```

In this pattern, the synchronization point is split into an arrive point (`bar.arrive()`) and a wait point (`bar.wait(std::move(token))`). A thread begins participating in a `cuda::barrier` with its first call to `bar.arrive()`. When a thread calls `bar.wait(std::move(token))` it will be blocked until participating threads have completed `bar.arrive()` the expected number of times, which is the expected arrival count argument passed to `init()`. Memory updates that happen before participating threads' call to `bar.arrive()` are guaranteed to be visible to participating threads after their call to `bar.wait(std::move(token))`. Note that the call to `bar.arrive()` does not block a thread, it can proceed with other work that does not depend upon memory updates that happen before other participating threads' call to `bar.arrive()`.

The *arrive and wait* pattern has five stages:

- Code **before** the arrive performs memory updates that will be read **after** the wait.
- Arrive point with implicit memory fence (i.e., equivalent to `cuda::atomic_thread_fence(cuda::memory_orde` `cuda::thread_scope_block)`).

- Code **between** arrive and wait.
- Wait point.
- Code **after** the wait, with visibility of updates that were performed **before** the arrive.

For a comprehensive guide on how to use asynchronous barriers, see *Asynchronous Barriers*.

3.2.4.3 Pipelines

The CUDA programming model provides the pipeline synchronization object as a coordination mechanism to sequence asynchronous memory copies into multiple stages, facilitating the implementation of double- or multi-buffering producer-consumer patterns. A pipeline is a double-ended queue with a *head* and a *tail* that processes work in a first-in first-out (FIFO) order. Producer threads commit work to the pipeline's head, while consumer threads pull work from the pipeline's tail.

Pipelines are exposed through the `cuda::pipeline` API in the libcu++ library, as well as through a *primitives API*. The following tables describe the main functionality of the two APIs.

`cuda::pipeline` API	Description
`producer_acquire`	Acquires an available stage in the pipeline's internal queue.
`producer_commit`	Commits the asynchronous operations issued after the `producer_acquire` call on the currently acquired stage of the pipeline.
`consumer_wait`	Waits for completion of asynchronous operations in the oldest stage of the pipeline.
`consumer_release`	Releases the oldest stage of the pipeline to the pipeline object for reuse. The released stage can be then acquired by a producer.

Primitives API	Description
`__pipeline_memcpy_async`	Request a memory copy from global to shared memory to be submitted for asynchronous evaluation.
`__pipeline_commit`	Commits the asynchronous operations issued before the call on the current stage of the pipeline.
`__pipeline_wait_prior(N`	Waits for completion of asynchronous operations in all but the last N commits to the pipeline.

The `cuda::pipeline` API has a richer interface with less restrictions, while the primitives API only supports tracking asynchronous copies from global memory to shared memory with specific size and alignment requirements. The primitives API provides equivalent functionality to a `cuda::pipeline` object with `cuda::thread_scope_thread`.

For detailed usage patterns and examples, see *Pipelines*.

3.2.5. Asynchronous Data Copies

Efficient data movement within the memory hierarchy is fundamental to achieving high performance in GPU computing. Traditional synchronous memory operations force threads to wait idle during data transfers. GPUs inherently hide memory latency through parallelism. That is, the SM switches to execute another warp while memory operations complete. Even with this latency hiding through parallelism, it is still possible for memory latency to be a bottleneck on both memory bandwidth utilization and compute resource efficiency. To address these bottlenecks, modern GPU architectures provide hardware-accelerated asynchronous data copy mechanisms that allow memory transfers to proceed independently while threads continue executing other work.

Asynchronous data copies enable overlapping of computation with data movement, by decoupling the initiation of a memory transfer from waiting for its completion. This way, threads can perform useful work during memory latency periods, leading to improved overall throughput and resource utilization.

Note

While concepts and principles underlying this section are similar to those discussed in the earlier chapter on *Asynchronous Execution*, that chapter covered asynchronous execution of kernels and memory transfers such as those invoked by `cudaMemcpyAsync`. That can be considered asynchrony of different components of the application.

The asynchrony described in this section refers to enabling transfer of data between the GPU's DRAM, i.e. global memory, and on-SM memory such as shared memory or tensor memory without blocking the GPU threads. This is an asynchrony within the execution of a single kernel launch.

To understand how asynchronous copies can improve performance, it is helpful to examine a common GPU computing pattern. CUDA applications often employ a *copy and compute* pattern that:

- fetches data from global memory,
- stores data to shared memory, and
- performs computations on shared memory data, and potentially writes results back to global memory.

The *copy* phase of this pattern is typically expressed as `shared[local_idx] = global[global_idx]`. This global to shared memory copy is expanded by the compiler to a read from global memory into a register followed by a write to shared memory from the register.

When this pattern occurs within an iterative algorithm, each thread block needs to synchronize after the `shared[local_idx] = global[global_idx]` assignment, to ensure all writes to shared memory have completed before the compute phase can begin. The thread block also needs to synchronize again after the compute phase, to prevent overwriting shared memory before all threads have completed their computations. This pattern is illustrated in the following code snippet.

```
#include <cooperative_groups.h>

__device__ void compute(int* global_out, int const* shared_in) {
    // Computes using all values of current batch from shared memory.
    // Stores this thread's result back to global memory.
}

__global__ void without_async_copy(int* global_out, int const* global_in,
↪size_t size, size_t batch_sz) {
  auto grid = cooperative_groups::this_grid();
  auto block = cooperative_groups::this_thread_block();
  assert(size == batch_sz * grid.size()); // Exposition: input size fits batch_
↪sz * grid_size
```

(continues on next page)

(continued from previous page)

```
  extern __shared__ int shared[]; // block.size() * sizeof(int) bytes

  size_t local_idx = block.thread_rank();

  for (size_t batch = 0; batch < batch_sz; ++batch) {
    // Compute the index of the current batch for this block in global memory.
    size_t block_batch_idx = block.group_index().x * block.size() + grid.
↪size() * batch;
    size_t global_idx = block_batch_idx + threadIdx.x;
    shared[local_idx] = global_in[global_idx];

    // Wait for all copies to complete.
    block.sync();

    // Compute and write result to global memory.
    compute(global_out + block_batch_idx, shared);

    // Wait for compute using shared memory to finish.
    block.sync();
  }
}
```

With asynchronous data copies, data movement from global memory to shared memory can be done asynchronously to enable more efficient use of the SM while waiting for data to arrive.

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

__device__ void compute(int* global_out, int const* shared_in) {
    // Computes using all values of current batch from shared memory.
    // Stores this thread's result back to global memory.
}

__global__ void with_async_copy(int* global_out, int const* global_in, size_t
↪size, size_t batch_sz) {
  auto grid = cooperative_groups::this_grid();
  auto block = cooperative_groups::this_thread_block();
  assert(size == batch_sz * grid.size()); // Exposition: input size fits batch_
↪sz * grid_size

  extern __shared__ int shared[]; // block.size() * sizeof(int) bytes

  size_t local_idx = block.thread_rank();

  for (size_t batch = 0; batch < batch_sz; ++batch) {
    // Compute the index of the current batch for this block in global memory.
    size_t block_batch_idx = block.group_index().x * block.size() + grid.
↪size() * batch;

    // Whole thread-group cooperatively copies whole batch to shared memory.
    cooperative_groups::memcpy_async(block, shared, global_in + block_batch_
↪idx, block.size());
```

(continues on next page)

(continued from previous page)

```
    // Compute on different data while waiting.

    // Wait for all copies to complete.
    cooperative_groups::wait(block);

    // Compute and write result to global memory.
    compute(global_out + block_batch_idx, shared);

    // Wait for compute using shared memory to finish.
    block.sync();
  }
}
```

The *cooperative_groups::memcpy_async* function copies `block.size()` elements from global memory to the `shared` data. This operation happens as-if performed by another thread, which synchronizes with the current thread's call to *cooperative_groups::wait* after the copy has completed. Until the copy operation completes, modifying the global data or reading or writing the shared data introduces a data race.

This example illustrates the fundamental concept behind all asynchronous copy operations: they decouple memory transfer initiation from completion, allowing threads to perform other work while data moves in the background. The CUDA programming model provides several APIs to access these capabilities, including `memcpy_async` functions available in *Cooperative Groups* and the libcu++ library, as well as lower-level `cuda::ptx` and primitives APIs. These APIs share similar semantics: they copy objects from source to destination as-if performed by another thread which, on completion of the copy, can be synchronized using different completion mechanisms.

Modern GPU architectures provide multiple hardware mechanisms for asynchronous data movement.

- LDGSTS (compute capability 8.0+) allows for efficient small-scale asynchronous transfers from global to shared memory.
- The tensor memory accelerator (TMA, compute capability 9.0+) extends these capabilities, providing bulk-asynchronous copy operations optimized for large multi-dimensional data transfers
- STAS instructions (compute capability 9.0+) enable small-scale asynchronous transfers from registers to distributed shared memory within a cluster.

These mechanisms support different data paths, transfer sizes, and alignment requirements, allowing developers to choose the most appropriate approach for their specific data access patterns. The following table gives an overview of the supported data paths for asynchronous copies within the GPU.

Table 5: Asynchronous copies with possible source and destination memory spaces. An empty cell indicates that a source-destination pair is not supported.

Direction		Copy Mechanism	
Source	**Destination**	**Asynchronous Copy**	**Bulk-Asynchronous Copy**
global	global		
shared::cta	global		supported (TMA, 9.0+)
global	shared::cta	supported (LDGSTS, 8.0+)	supported (TMA, 9.0+)
global	shared::cluster		supported (TMA, 9.0+)
shared::cluster	shared::cta		supported (TMA, 9.0+)
shared::cta	shared::cta		
registers	shared::cluster	supported (STAS, 9.0+)	

Sections *Using LDGSTS*, *Using the Tensor Memory Accelerator (TMA)* and *Using STAS* will go into more details about each mechanism.

3.2.6. Configuring L1/Shared Memory Balance

As mentioned in *L1 data cache*, the L1 and shared memory on an SM use the same physical resource, known as the unified data cache. On most architectures, if a kernel uses little or no shared memory, the unified data cache can be configured to provide the maximal amount of L1 cache allowed by the architecture.

The unified data cache reserved for shared memory is configurable on a per-kernel basis. An application can set the `carveout`, or preferred shared memory capacity, with the cudaFuncSetAttribute function called before the kernel is launched.

```
cudaFuncSetAttribute(kernel_name,
↪cudaFuncAttributePreferredSharedMemoryCarveout, carveout);
```

The application can set the `carveout` as an integer percentage of the maximum supported shared memory capacity of that architecture. In addition to an integer percentage, three convenience enums are provided as carveout values.

- `cudaSharedmemCarveoutDefault`
- `cudaSharedmemCarveoutMaxL1`
- `cudaSharedmemCarveoutMaxShared`

The maximum supported shared memory and the supported carveout sizes vary by architecture; see *Shared Memory Capacity per Compute Capability* for details.

Where a chosen integer percentage carveout does not map exactly to a supported shared memory capacity, the next larger capacity is used. For example, for devices of compute capability 12.0, which have a maximum shared memory capacity of 100KB, setting the carveout to 50% will result in 64KB of shared memory, not 50KB, because devices of compute capability 12.0 support shared memory sizes of 0, 8, 16, 32, 64, and 100.

The function passed to cudaFuncSetAttribute must be declared with the __global__ specifier. cudaFuncSetAttribute is interpreted by the driver as a hint, and the driver may choose a different carveout size if required to execute the kernel.

Note

Another CUDA API, cudaFuncSetCacheConfig, also allows an application to adjust the balance between L1 and shared memory for a kernel. However, this API set a hard requirements on shared/L1 balance for kernel launch. As a result, interleaving kernels with different shared memory configurations would needlessly *serialize launches* behind shared memory reconfigurations. cudaFuncSetAttribute is preferred because driver may choose a different configuration if required to execute the function or to avoid thrashing.

Kernels relying on shared memory allocations over 48 KB per block are architecture-specific. As such they must use *dynamic shared memory* rather than statically-sized arrays and require an explicit opt-in using cudaFuncSetAttribute as follows.

```
// Device code
__global__ void MyKernel(...)
{
  extern __shared__ float buffer[];
  ...
}

// Host code
int maxbytes = 98304; // 96 KB
cudaFuncSetAttribute(MyKernel, cudaFuncAttributeMaxDynamicSharedMemorySize,
↪maxbytes);
MyKernel <<<gridDim, blockDim, maxbytes>>>(...);
```

3.3. The CUDA Driver API

Previous sections of this guide have covered the CUDA runtime. As mentioned in *CUDA Runtime API and CUDA Driver API*, the CUDA runtime is written on top of the lower level CUDA driver API. This section covers some of the differences between the CUDA runtime and the driver APIs, as well has how to intermix them. Most applications can operate at full performance without ever needing to interact with the CUDA driver API. However, new interfaces are sometimes available in the driver API earlier than the runtime API, and some advanced interfaces, such as *Virtual Memory Management*, are only exposed in the driver API.

The driver API is implemented in the cuda dynamic library (cuda.dll or cuda.so) which is copied on the system during the installation of the device driver. All its entry points are prefixed with cu.

It is a handle-based, imperative API: Most objects are referenced by opaque handles that may be specified to functions to manipulate the objects.

The objects available in the driver API are summarized in Table 6.

Table 6: Objects Available in the CUDA Driver API

Object	Handle	Description
Device	CUdevice	CUDA-enabled device
Context	CUcontext	Roughly equivalent to a CPU process
Module	CUmodule	Roughly equivalent to a dynamic library
Function	CUfunction	Kernel
Heap memory	CUdeviceptr	Pointer to device memory
CUDA array	CUarray	Opaque container for one-dimensional or two-dimensional data on the device, readable via texture or surface references
Texture object	CUtexref	Object that describes how to interpret texture memory data
Surface reference	CUsurfref	Object that describes how to read or write CUDA arrays
Stream	CUstream	Object that describes a CUDA stream
Event	CUevent	Object that describes a CUDA event

The driver API must be initialized with `cuInit()` before any function from the driver API is called. A CUDA context must then be created that is attached to a specific device and made current to the calling host thread as detailed in *Context*.

Within a CUDA context, kernels are explicitly loaded as PTX or binary objects by the host code as described in *Module*. Kernels written in C++ must therefore be compiled separately into *PTX* or binary objects. Kernels are launched using API entry points as described in *Kernel Execution*.

Any application that wants to run on future device architectures must load *PTX*, not binary code. This is because binary code is architecture-specific and therefore incompatible with future architectures, whereas *PTX* code is compiled to binary code at load time by the device driver.

Here is the host code of the sample from *Kernels* written using the driver API:

```
int main()
{
    int N = ...;
    size_t size = N * sizeof(float);

    // Allocate input vectors h_A and h_B in host memory
    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);

    // Initialize input vectors
    ...
```

(continues on next page)

(continued from previous page)

```
    // Initialize
    cuInit(0);

    // Get number of devices supporting CUDA
    int deviceCount = 0;
    cuDeviceGetCount(&deviceCount);
    if (deviceCount == 0) {
        printf("There is no device supporting CUDA.\n");
        exit (0);
    }

    // Get handle for device 0
    CUdevice cuDevice;
    cuDeviceGet(&cuDevice, 0);

    // Create context
    CUcontext cuContext;
    cuCtxCreate(&cuContext, 0, cuDevice);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, "VecAdd.ptx");

    // Allocate vectors in device memory
    CUdeviceptr d_A;
    cuMemAlloc(&d_A, size);
    CUdeviceptr d_B;
    cuMemAlloc(&d_B, size);
    CUdeviceptr d_C;
    cuMemAlloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cuMemcpyHtoD(d_A, h_A, size);
    cuMemcpyHtoD(d_B, h_B, size);

    // Get function handle from module
    CUfunction vecAdd;
    cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    void* args[] = { &d_A, &d_B, &d_C, &N };
    cuLaunchKernel(vecAdd,
                   blocksPerGrid, 1, 1, threadsPerBlock, 1, 1,
                   0, 0, args, 0);

    ...
}
```

Full code can be found in the `vectorAddDrv` CUDA sample.

3.3.1. Context

A CUDA context is analogous to a CPU process. All resources and actions performed within the driver API are encapsulated inside a CUDA context, and the system automatically cleans up these resources when the context is destroyed. Besides objects such as modules and texture or surface references, each context has its own distinct address space. As a result, `CUdeviceptr` values from different contexts reference different memory locations.

A host thread may have only one device context current at a time. When a context is created with `cuCtxCreate()`, it is made current to the calling host thread. CUDA functions that operate in a context (most functions that do not involve device enumeration or context management) will return `CUDA_ERROR_INVALID_CONTEXT` if a valid context is not current to the thread.

Each host thread has a stack of current contexts. `cuCtxCreate()` pushes the new context onto the top of the stack. `cuCtxPopCurrent()` may be called to detach the context from the host thread. The context is then "floating" and may be pushed as the current context for any host thread. `cuCtxPopCurrent()` also restores the previous current context, if any.

A usage count is also maintained for each context. `cuCtxCreate()` creates a context with a usage count of 1. `cuCtxAttach()` increments the usage count and `cuCtxDetach()` decrements it. A context is destroyed when the usage count goes to 0 when calling `cuCtxDetach()` or `cuCtxDestroy()`.

The driver API is interoperable with the runtime and it is possible to access the primary context (see *Runtime Initialization*) managed by the runtime from the driver API via `cuDevicePrimaryCtxRetain()`.

Usage count facilitates interoperability between third party authored code operating in the same context. For example, if three libraries are loaded to use the same context, each library would call `cuCtxAttach()` to increment the usage count and `cuCtxDetach()` to decrement the usage count when the library is done using the context. For most libraries, it is expected that the application will have created a context before loading or initializing the library; that way, the application can create the context using its own heuristics, and the library simply operates on the context handed to it. Libraries that wish to create their own contexts - unbeknownst to their API clients who may or may not have created contexts of their own - would use `cuCtxPushCurrent()` and `cuCtxPopCurrent()` as illustrated in the following figure.

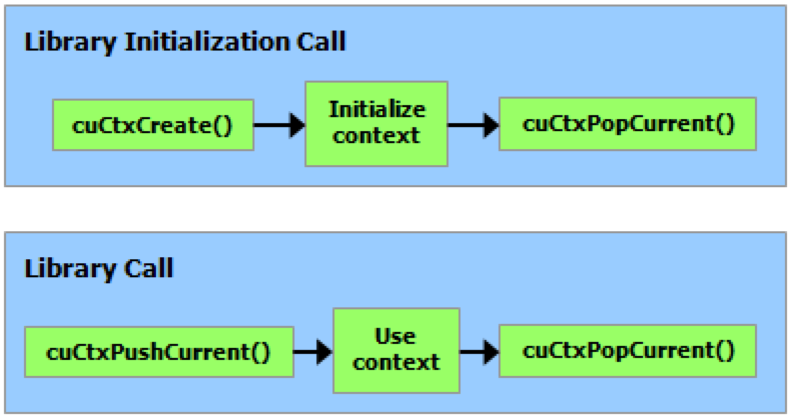


Figure 20: Library Context Management

3.3.2. Module

Modules are dynamically loadable packages of device code and data, akin to DLLs in Windows, that are output by nvcc (see *Compilation with NVCC*). The names for all symbols, including functions, global variables, and texture or surface references, are maintained at module scope so that modules written by independent third parties may interoperate in the same CUDA context.

This code sample loads a module and retrieves a handle to some kernel:

```
CUmodule cuModule;
cuModuleLoad(&cuModule, "myModule.ptx");
CUfunction myKernel;
cuModuleGetFunction(&myKernel, cuModule, "MyKernel");
```

This code sample compiles and loads a new module from PTX code and parses compilation errors:

```
#define BUFFER_SIZE 8192
CUmodule cuModule;
CUjit_option options[3];
void* values[3];
char* PTXCode = "some PTX code";
char error_log[BUFFER_SIZE];
int err;
options[0] = CU_JIT_ERROR_LOG_BUFFER;
values[0]  = (void*)error_log;
options[1] = CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES;
values[1]  = (void*)BUFFER_SIZE;
options[2] = CU_JIT_TARGET_FROM_CUCONTEXT;
values[2]  = 0;
err = cuModuleLoadDataEx(&cuModule, PTXCode, 3, options, values);
if (err != CUDA_SUCCESS)
    printf("Link error:\n%s\n", error_log);
```

This code sample compiles, links, and loads a new module from multiple PTX codes and parses link and compilation errors:

```
#define BUFFER_SIZE 8192
CUmodule cuModule;
CUjit_option options[6];
void* values[6];
float walltime;
char error_log[BUFFER_SIZE], info_log[BUFFER_SIZE];
char* PTXCode0 = "some PTX code";
char* PTXCode1 = "some other PTX code";
CUlinkState linkState;
int err;
void* cubin;
size_t cubinSize;
options[0] = CU_JIT_WALL_TIME;
values[0] = (void*)&walltime;
options[1] = CU_JIT_INFO_LOG_BUFFER;
values[1] = (void*)info_log;
options[2] = CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES;
values[2] = (void*)BUFFER_SIZE;
```

(continues on next page)

(continued from previous page)

```
options[3] = CU_JIT_ERROR_LOG_BUFFER;
values[3] = (void*)error_log;
options[4] = CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES;
values[4] = (void*)BUFFER_SIZE;
options[5] = CU_JIT_LOG_VERBOSE;
values[5] = (void*)1;
cuLinkCreate(6, options, values, &linkState);
err = cuLinkAddData(linkState, CU_JIT_INPUT_PTX,
                    (void*)PTXCode0, strlen(PTXCode0) + 1, 0, 0, 0, 0);
if (err != CUDA_SUCCESS)
    printf("Link error:\n%s\n", error_log);
err = cuLinkAddData(linkState, CU_JIT_INPUT_PTX,
                    (void*)PTXCode1, strlen(PTXCode1) + 1, 0, 0, 0, 0);
if (err != CUDA_SUCCESS)
    printf("Link error:\n%s\n", error_log);
cuLinkComplete(linkState, &cubin, &cubinSize);
printf("Link completed in %fms. Linker Output:\n%s\n", walltime, info_log);
cuModuleLoadData(cuModule, cubin);
cuLinkDestroy(linkState);
```

It's possible to accelerate some parts of the module linking/loading process by using multiple threads, including when loading a cubin. This code sample uses `CU_JIT_BINARY_LOADER_THREAD_COUNT` to speed up module loading.

```
#define BUFFER_SIZE 8192
CUmodule cuModule;
CUjit_option options[3];
void* values[3];
char* cubinCode = "some cubin code";
char error_log[BUFFER_SIZE];
int err;
options[0] = CU_JIT_ERROR_LOG_BUFFER;
values[0]  = (void*)error_log;
options[1] = CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES;
values[1]  = (void*)BUFFER_SIZE;
options[2] = CU_JIT_BINARY_LOADER_THREAD_COUNT;
values[2]  = 0; // Use as many threads as CPUs on the machine
err = cuModuleLoadDataEx(&cuModule, cubinCode, 3, options, values);
if (err != CUDA_SUCCESS)
    printf("Link error:\n%s\n", error_log);
```

Full code can be found in the `ptxjit` CUDA sample.

3.3.3. Kernel Execution

`cuLaunchKernel()` launches a kernel with a given execution configuration.

Parameters are passed either as an array of pointers (next to last parameter of `cuLaunchKernel()`) where the nth pointer corresponds to the nth parameter and points to a region of memory from which the parameter is copied, or as one of the extra options (last parameter of `cuLaunchKernel()`).

When parameters are passed as an extra option (the `CU_LAUNCH_PARAM_BUFFER_POINTER` option), they are passed as a pointer to a single buffer where parameters are assumed to be properly offset with respect to each other by matching the alignment requirement for each parameter type in device code.

Alignment requirements in device code for the built-in vector types are listed in Table 42. For all other basic types, the alignment requirement in device code matches the alignment requirement in host code and can therefore be obtained using `__alignof()`. The only exception is when the host compiler aligns `double` and `long long` (and `long` on a 64-bit system) on a one-word boundary instead of a two-word boundary (for example, using `gcc`'s compilation flag `-mno-align-double`) since in device code these types are always aligned on a two-word boundary.

`CUdeviceptr` is an integer, but represents a pointer, so its alignment requirement is `__alignof(void*)`.

The following code sample uses a macro (`ALIGN_UP()`) to adjust the offset of each parameter to meet its alignment requirement and another macro (`ADD_TO_PARAM_BUFFER()`) to add each parameter to the parameter buffer passed to the `CU_LAUNCH_PARAM_BUFFER_POINTER` option.

```
#define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)

char paramBuffer[1024];
size_t paramBufferSize = 0;

#define ADD_TO_PARAM_BUFFER(value, alignment)                   \
    do {                                                        \
        paramBufferSize = ALIGN_UP(paramBufferSize, alignment); \
        memcpy(paramBuffer + paramBufferSize,                   \
               &(value), sizeof(value));                        \
        paramBufferSize += sizeof(value);                       \
    } while (0)

int i;
ADD_TO_PARAM_BUFFER(i, __alignof(i));
float4 f4;
ADD_TO_PARAM_BUFFER(f4, 16); // float4's alignment is 16
char c;
ADD_TO_PARAM_BUFFER(c, __alignof(c));
float f;
ADD_TO_PARAM_BUFFER(f, __alignof(f));
CUdeviceptr devPtr;
ADD_TO_PARAM_BUFFER(devPtr, __alignof(devPtr));
float2 f2;
ADD_TO_PARAM_BUFFER(f2, 8); // float2's alignment is 8

void* extra[] = {
    CU_LAUNCH_PARAM_BUFFER_POINTER, paramBuffer,
    CU_LAUNCH_PARAM_BUFFER_SIZE,    &paramBufferSize,
    CU_LAUNCH_PARAM_END
};
cuLaunchKernel(cuFunction,
               blockWidth, blockHeight, blockDepth,
               gridWidth, gridHeight, gridDepth,
               0, 0, 0, extra);
```

The alignment requirement of a structure is equal to the maximum of the alignment requirements of its fields. The alignment requirement of a structure that contains built-in vector types, `CUdeviceptr`, or non-aligned `double` and `long long`, might therefore differ between device code and host code. Such a structure might also be padded differently. The following structure, for example, is not padded at all in host code, but it is padded in device code with 12 bytes after field `f` since the alignment requirement for field `f4` is 16.

```
typedef struct {
    float  f;
    float4 f4;
} myStruct;
```

3.3.4. Interoperability between Runtime and Driver APIs

An application can mix runtime API code with driver API code.

If a context is created and made current via the driver API, subsequent runtime calls will use this context instead of creating a new one.

If the runtime is initialized, `cuCtxGetCurrent()` can be used to retrieve the context created during initialization. This context can be used by subsequent driver API calls.

The implicitly created context from the runtime is called the primary context (see *Runtime Initialization*). It can be managed from the driver API with the Primary Context Management functions.

Device memory can be allocated and freed using either API. `CUdeviceptr` can be cast to regular pointers and vice-versa:

```
CUdeviceptr devPtr;
float* d_data;

// Allocation using driver API
cuMemAlloc(&devPtr, size);
d_data = (float*)devPtr;

// Allocation using runtime API
cudaMalloc(&d_data, size);
devPtr = (CUdeviceptr)d_data;
```

In particular, this means that applications written using the driver API can invoke libraries written using the runtime API (such as cuFFT, cuBLAS, …).

All functions from the device and version management sections of the reference manual can be used interchangeably.

3.4. Programming Systems with Multiple GPUs

Multi-GPU programming allows an application to address problem sizes and achieve performance levels beyond what is possible with a single GPU by exploiting the larger aggregate arithmetic performance, memory capacity, and memory bandwidth provided by multi-GPU systems.

CUDA enables multi-GPU programming through host APIs, driver infrastructure, and supporting GPU hardware technologies:

- Host thread CUDA context management
- Unified memory addressing for all processors in the system

- Peer-to-peer bulk memory transfers between GPUs
- Fine-grained peer-to-peer GPU load/store memory access
- Higher level abstractions and supporting system software such as CUDA interprocess communication, parallel reductions using NCCL, and communication using NVLink and/or GPU-Direct RDMA with APIs such as NVSHMEM and MPI

At the most basic level, multi-GPU programming requires the application to manage multiple active CUDA contexts concurrently, distribute data to the GPUs, launch kernels on the GPUs to complete their work, and to communicate or collect the results so that they can be acted upon by the application. The details of how this is done differ depending on the most effective mapping of an application's algorithms, available parallelism, and existing code structure to a suitable multi-GPU programming approach. Some of the most common multi-GPU programming approaches include:

- A single host thread driving multiple GPUs
- Multiple host threads, each driving their own GPU
- Multiple single-threaded host processes, each driving their own GPU
- Multiple host processes containing multiple threads, each driving their own GPU
- Multi-node NVLink-connected clusters, with GPUs driven by threads and processes running within multiple operating system instances across the cluster nodes

GPUs can communicate with each other through memory transfers and peer accesses between device memories, covering each of the multi-device work distribution approaches listed above. High performance, low-latency GPU communications are supported by querying for and enabling the use of peer-to-peer GPU memory access, and leveraging NVLink to achieve high bandwidth transfers and finer-grained load/store operations between devices.

CUDA unified virtual addressing permits communication between multiple GPUs within the same host process with minimal additional steps to query and enable the use of high performance peer-to-peer memory access and transfers, e.g., via NVLink.

Communication between multiple GPUs managed by different host processes is supported through the use of interprocess communication (IPC) and Virtual memory Management (VMM) APIs. An introduction to high level IPC concepts and intra-node CUDA IPC APIs are discussed in the *Interprocess Communication* section. AdvancedVirtual Memory Management (VMM) APIs support both intra-node and multi-node IPC, are usable on both Linux and Windows operating systems, and allow per-allocation granularity control over IPC sharing of memory buffers as described in *Virtual Memory Management*.

CUDA itself provides the APIs needed to implement collective operations within a group of GPUs, potentially including the host, but it does not provide high level multi-GPU collective APIs itself. Multi-GPU collectives are provided by higher abstraction CUDA communication libraries such as NCCL and NVSHMEM.

3.4.1. Multi-Device Context and Execution Management

The first steps that are required to for an application to use multiple GPUs are to enumerate the available GPU devices, select among the available devices as appropriate based on their hardware properties, CPU affinity, and connectivity to peers, and to create CUDA contexts for each device that the application will use.

3.4.1.1 Device Enumeration

The following code sample shows how to query number of CUDA-enabled devices, enumerate each of the devices, and query their properties.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    printf("Device %d has compute capability %d.%d.\n",
           device, deviceProp.major, deviceProp.minor);
}
```

3.4.1.2 Device Selection

A host thread can set the device it is currently operating on at any time by calling `cudaSetDevice()`. Device memory allocations and kernel launches are made on the current device; streams and events are created in association with the currently set device. Until a call to `cudaSetDevice()` is made by the host thread, the current device defaults to device 0.

The following code sample illustrates how setting the current device affects subsequent memory allocation and kernel execution operations.

```
size_t size = 1024 * sizeof(float);
cudaSetDevice(0);            // Set device 0 as current
float* p0;
cudaMalloc(&p0, size);       // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0); // Launch kernel on device 0

cudaSetDevice(1);            // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);       // Allocate memory on device 1
MyKernel<<<1000, 128>>>(p1); // Launch kernel on device 1
```

3.4.1.3 Multi-Device Stream, Event, and Memory Copy Behavior

A kernel launch will fail if it is issued to a stream that is not associated to the current device as illustrated in the following code sample.

```
cudaSetDevice(0);               // Set device 0 as current
cudaStream_t s0;
cudaStreamCreate(&s0);          // Create stream s0 on device 0
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 0 in s0

cudaSetDevice(1);               // Set device 1 as current
cudaStream_t s1;
cudaStreamCreate(&s1);          // Create stream s1 on device 1
MyKernel<<<100, 64, 0, s1>>>(); // Launch kernel on device 1 in s1

// This kernel launch will fail, since stream s0 is not associated to device 1:
MyKernel<<<100, 64, 0, s0>>>(); // Launch kernel on device 1 in s0
```

A memory copy will succeed even if it is issued to a stream that is not associated to the current device.

cudaEventRecord() will fail if the input event and input stream are associated to different devices.

cudaEventElapsedTime() will fail if the two input events are associated to different devices.

cudaEventSynchronize() and cudaEventQuery() will succeed even if the input event is associated to a device that is different from the current device.

cudaStreamWaitEvent() will succeed even if the input stream and input event are associated to different devices. cudaStreamWaitEvent() can therefore be used to synchronize multiple devices with each other.

Each device has its own *default stream*, so commands issued to the default stream of a device may execute out of order or concurrently with respect to commands issued to the default stream of any other device.

3.4.2. Multi-Device Peer-to-Peer Transfers and Memory Access

3.4.2.1 Peer-to-Peer Memory Transfers

CUDA can perform memory transfers between devices and will take advantage of dedicated copy engines and NVLink hardware to maximize performance when peer-to-peer memory access is possible.

cudaMemcpy can be used with the copy type cudaMemcpyDeviceToDevice or cudaMemcpyDefault.

Otherwise, copies must be performed using cudaMemcpyPeer(), cudaMemcpyPeerAsync(), cudaMemcpy3DPeer(), or cudaMemcpy3DPeerAsync() as illustrated in the following code sample.

```
cudaSetDevice(0);                    // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);               // Allocate memory on device 0

cudaSetDevice(1);                    // Set device 1 as current
float* p1;
cudaMalloc(&p1, size);               // Allocate memory on device 1

cudaSetDevice(0);                    // Set device 0 as current
MyKernel<<<1000, 128>>>(p0);         // Launch kernel on device 0

cudaSetDevice(1);                    // Set device 1 as current
cudaMemcpyPeer(p1, 1, p0, 0, size); // Copy p0 to p1
MyKernel<<<1000, 128>>>(p1);         // Launch kernel on device 1
```

A copy (in the implicit *NULL* stream) between the memories of two different devices:

- does not start until all commands previously issued to either device have completed and
- runs to completion before any commands (see *Asynchronous Execution*) issued after the copy to either device can start.

Consistent with the normal behavior of streams, an asynchronous copy between the memories of two devices may overlap with copies or kernels in another stream.

If peer-to-peer access is enabled between two devices, e.g., as described in *Peer-to-Peer Memory Access*, peer-to-peer memory copies between these two devices no longer need to be staged through the host and are therefore faster.

3.4.2.2 Peer-to-Peer Memory Access

Depending on the system properties, specifically the PCIe and/or NVLink topology, devices are able to address each other's memory (i.e., a kernel executing on one device can dereference a pointer to the memory of the other device). Peer-to-peer memory access is supported between two devices if `cudaDeviceCanAccessPeer()` returns true for the specified devices.

Peer-to-peer memory access must be enabled between two devices by calling `cudaDeviceEnablePeerAccess()` as illustrated in the following code sample. On non-NVSwitch enabled systems, each device can support a system-wide maximum of eight peer connections.

A unified virtual address space is used for both devices (see *Unified Virtual Address Space*), so the same pointer can be used to address memory from both devices as shown in the code sample below.

```
cudaSetDevice(0);                     // Set device 0 as current
float* p0;
size_t size = 1024 * sizeof(float);
cudaMalloc(&p0, size);                // Allocate memory on device 0
MyKernel<<<1000, 128>>>(p0);          // Launch kernel on device 0

cudaSetDevice(1);                     // Set device 1 as current
cudaDeviceEnablePeerAccess(0, 0);     // Enable peer-to-peer access
                                      // with device 0

// Launch kernel on device 1
// This kernel launch can access memory on device 0 at address p0
MyKernel<<<1000, 128>>>(p0);
```

Note

The use of `cudaDeviceEnablePeerAccess()` to enable peer memory access operates globally on all previous and subsequent GPU memory allocations on the peer device. Enabling peer access to a device via `cudaDeviceEnablePeerAccess()` adds runtime cost to device memory allocation operations on that peer due to the need make the allocations immediately accessible to the current device and any other peers that also have access, adding multiplicative overhead that scales with the number of peer devices.

A more scalable alternative to enabling peer memory access for all device memory allocations is to make use of CUDA Virtual Memory Management APIs to explicitly allocate peer-accessible memory regions only as-needed, at allocation time. By requesting peer-accessibility explicitly during memory allocation, the runtime cost of memory allocations are unharmed for allocations not accessible to peers, and peer-accessible data structures are correctly scoped for improved software debugging and reliability (see ref::*virtual-memory-management*).

3.4.2.3 Peer-to-Peer Memory Consistency

Synchronization operations must be used to enforce the ordering and correctness of memory accesses by concurrently executing threads in grids distributed across multiple devices. Threads synchronizing across devices operate at the `thread_scope_system` synchronization scope. Similarly, memory operations fall within the `thread_scope_system` memory synchronization domain.

CUDA ref::*atomic-functions* can perform read-modify-write operations on an object in peer device memory when only a single GPU is accessing that object. The requirements and limitations for peer atomicity are described in the CUDA memory model atomicity requirements discussion.

3.4.2.4 Multi-Device Managed Memory

Managed memory can be used on multi-GPU systems with peer-to-peer support. The detailed requirements for concurrent multi-device managed memory access and APIs for GPU-exclusive access to managed memory are described in *Multi-GPU*.

3.4.2.5 Host IOMMU Hardware, PCI Access Control Services, and VMs

On Linux specifically, CUDA and the display driver do not support IOMMU-enabled bare-metal PCIe peer-to-peer memory transfer. However, CUDA and the display driver do support IOMMU via virtual machine pass through. The IOMMU must be disabled when running Linux on a bare metal system to prevent silent device memory corruption. Conversely, the IOMMU should be enabled and the VFIO driver be used for PCIe pass through for virtual machines.

On Windows the IOMMU limitation above does not exist.

See also Allocating DMA Buffers on 64-bit Platforms.

Additionally, PCI Access Control Services (ACS) can be enabled on systems that support IOMMU. The PCI ACS feature redirects all PCI point-to-point traffic through the CPU root complex, which can cause significant performance loss due to the reduction in overall bisection bandwidth.

3.5. A Tour of CUDA Features

Sections 1-3 of this programming guide have introduced CUDA and GPU programming, covering foundational topics both conceptually and in simple code examples. The sections describing specific CUDA features in part 4 of this guide assume knowledge of the concepts covered in sections 1-3 of this guide.

CUDA has many features which apply to different problems. Not all of them will be applicable to every use case. This chapter serves to introduce each of these features and describe its intended use and the problems it may help solve. Features are coarsely sorted into categories by the type of problem they are intended to solve. Some features, such as CUDA graphs, could fit into more than one category.

Section 4 covers these CUDA features in more complete detail.

3.5.1. Improving Kernel Performance

The features outlined in this section are all intended to aid kernel developers to maximize the performance of their kernels.

3.5.1.1 Asynchronous Barriers

Asynchronous barriers were introduced in Section 3.2.4.2 and allow for more nuanced control over synchronization between threads. Asynchronous barriers separate the arrival and the wait of a barrier. This allows applications to perform work that does not depend on the barrier while waiting for other threads to arrive. Asynchronous barriers can be specified for different *thread scopes*. Full details of asynchronous barriers are found in Section 4.9.

3.5.1.2 Asynchronous Data Copies and the Tensor Memory Accelerator (TMA)

Asynchronous data copies in the context of CUDA kernel code refers to the ability to move data between shared memory and GPU DRAM while still carrying out computations. This should not be confused with asynchronous memory copies between the CPU and GPU. This feature makes used of asynchronous barriers. Section 4.11 covers the use of asynchronous copies in detail.

3.5.1.3 Pipelines

Pipelines are a mechanism for staging work and coordinating multi-buffer producer–consumer patterns, commonly used to overlap compute with *asynchronous data copies*. Section 4.10 has details and examples of using pipelines in CUDA.

3.5.1.4 Work Stealing with Cluster Launch Control

Work stealing is a technique for maintaining utilization in uneven workloads where workers that have completed their work can 'steal' tasks from other workers. Cluster launch control, a feature introduced in compute capability 10.0 (Blackwell), gives kernels direct control over in-flight block scheduling so they can adapt to uneven workloads in real time. A thread block can cancel the launch of another thread block or cluster that has not yet started, claim its index, and immediately begin executing the stolen work. This work-stealing flow keeps SMs busy and cuts idle time under irregular data or runtime variation—delivering finer-grained load balancing without relying on the hardware scheduler alone.

Section 4.12 provides details on how to use this feature.

3.5.2. Improving Latencies

The features outlined in this section share a common theme of aiming to reduce some type of latency, though the type of latency being addressed differs between the different features. By and large they are focused on latencies at the kernel launch level or higher. GPU memory access latency within a kernel is not one of the latencies considered here.

3.5.2.1 Green Contexts

Green contexts, also called *execution contexts*, is the name given to a CUDA feature which enables a program to create *CUDA contexts* which will execute work only on a subset of the SMs of a GPU. By default, the thread blocks of a kernel launch are dispatched to any SM within the GPU which can fulfill the resource requirements of the kernel. There are a large number of factors which can affect which SMs can execute a thread block, including but not necessarily limited to: shared memory use, register use, use of clusters, and total number of threads in the thread block.

Execution contexts allow a kernel to be launched into a specially created context which further limits the number of SMs available to execute the kernel. Importantly, when a program creates a green context which uses some set of SMs, other contexts on the GPU will not schedule thread blocks onto the SMs allocated to the green context. This includes the primary context, which is the default context used by the CUDA runtime. This allows these SMs to be reserved for workloads which are high priority or latency-sensitive.

Section 4.6 gives full details on the use of green contexts. Green contexts are available in the CUDA runtime in CUDA 13.1 and later.

3.5.2.2 Stream-Ordered Memory Allocation

The *stream-ordered memory allocator* allows programs to sequence allocation and freeing of GPU memory into a *CUDA stream*. Unlike `cudaMalloc` and `cudaFree` which execute immediately, `cudaMallocAsync` and `cudaFreeAsync` inserts a memory allocation or free operation into a CUDA stream. Section 4.3 covers all the details of these APIs.

3.5.2.3 CUDA Graphs

CUDA graphs enable an application to specify a sequence of CUDA operations such as kernel launches or memory copies and the dependencies between these operations so that they can be executed efficiently on the GPU. Similar behavior can be attained by using *CUDA streams*, and indeed one of the mechanisms for creating a graph is called *stream capture*, which enables the operations on a stream to be recorded into a CUDA graph. Graphs can also be created using the *CUDA graphs API*.

Once a graph has been created, it can be instantiated and executed many times. This is useful for specifying workloads that will be repeated. Graphs offer some performance benefits in reducing CPU launch costs associated with invoking CUDA operations as well as enabling optimizations only available when the whole workload is specified in advance.

Section 4.2 describes and demonstrates how to use CUDA graphs.

3.5.2.4 Programmatic Dependent Launch

Programmatic dependent launch is a CUDA feature which allows a dependent kernel, i.e. a kernel which depends on the output of a prior kernel, to begin execution before the primary kernel on which it depends has completed. The dependent kernel can execute setup code and unrelated work up until it requires data from the primary kernel and block there. The primary kernel can signal when the data required by the dependent kernel is ready, which will release the dependent kernel to continue executing. This enables some overlap between the kernels which can help keep GPU utilization high while minimizing the latency of the critical data path. Section 4.5 covers programmatic dependent launch.

3.5.2.5 Lazy Loading

Lazy loading is a feature which allows control over how the JIT compiler operates at application startup. Applications which have many kernels which need to be JIT compiled from PTX to cubin may experience long startup times if all kernels are JIT compiled as part of application startup. The default behavior is that modules are not compiled until they are needed. This can be changed by the use of *environment variables*, as detailed in Section 4.7.

3.5.3. Functionality Features

The features described here share a common trait that they are meant to enable additional capabilities or functionality.

3.5.3.1 Extended GPU Memory

Extended GPU memory is a feature available in NVLink-C2C connected systems that enables efficient access to all memory within the system from within a GPU. EGM is covered in detail in Section 4.17.

3.5.3.2 Dynamic Parallelism

CUDA applications most commonly launch kernels from code running on the CPU. It is also possible to create new kernel invocations from a kernel running on the GPU. This feature is referred to as *CUDA dynamic parallelism*. Section 4.18 covers the details of creating new GPU kernel launches from code running on the GPU.

3.5.4. CUDA Interoperability

3.5.4.1 CUDA Interoperability with other APIs

There are other mechanisms than CUDA for running code on GPUs. The application GPUs were originally built to accelerate, computer graphics, uses its own set of APIs such as Direct3D and Vulkan. Applications may wish to use one of the graphics APIs for 3D rendering while performing computations in CUDA. CUDA provides mechanisms for exchanging data stored on the GPU between the CUDA contexts and the GPU contexts used by the 3D APIs. For example, an application may perform a simulation using CUDA, and then use a 3D API to create visualizations of the results. This is achieved by making some buffers readable and/or writeable from both CUDA and the graphics API.

The same mechanisms which allow sharing of buffers with graphics APIs are also used to share buffers with communications mechanisms which can enable rapid, direct GPU-to-GPU communication within multi-node environments.

Section 4.19 describes how CUDA interoperates with other GPU APIs and how to share data between CUDA and other APIs, providing specific examples for a number of different APIs.

3.5.4.2 Interprocess Communication

For very large computations, it is common to use multiple GPUs together to make use of more memory and more compute resources working together on a problem. Within a single system, or node in cluster computing terminology, multiple GPUs can be used in a single host process. This is described in Section 3.4.

It is also common to use separate host processes spanning either a single computer or multiple computers. When multiple processes are working together, communication between them is known as interprocess communication. CUDA interprocess communication (CUDA IPC) provides mechanisms to share GPU buffers between different processes. Section 4.15 explains and demonstrates how CUDA IPC can be used to coordinate and communicate between different host processes.

3.5.5. Fine-Grained Control

3.5.5.1 Virtual Memory Management

As mentioned in Section 2.4.1, all GPUs in a system, along with the CPU memory, share a single unified virtual address space. Most applications can use the default memory management provided by CUDA without the need to change its behavior. However, *the CUDA driver API* provides advanced and detailed controls over the layout of this virtual memory space for those that need it. This is mostly applicable for controlling the behavior of buffers when sharing between GPUs both within and across multiple systems.

Section 4.16 covers the controls offered by the CUDA driver API, how they work and when a developer may find them advantageous.

3.5.5.2 Driver Entry Point Access

Driver entry point access refers to the ability, starting in CUDA 11.3, to retrieve function pointers to the CUDA Driver and CUDA Runtime APIs. It also allows developers to retrieve function pointers for specific variants of driver functions, and to access driver functions from drivers newer than those available in the CUDA toolkit. Section 4.20 covers driver entry point access.

3.5.5.3 Error Log Management

Error log management provides utilities for handling and logging errors from CUDA APIs. Setting a single environment variable `CUDA_LOG_FILE` enables capturing CUDA errors directly to stderr, stdout, or a file. Error log management also enables applications to register a callback which is triggered when CUDA encounters an error. Section 4.8 provides more details on error log management.

Chapter 4. CUDA Features

4.1. Unified Memory

This section explains the detailed behavior and use of each of the different paradigms of unified memory available. *The earlier section on unified memory* showed how to determine which unified memory paradigm applies and briefly introduced each.

As discussed previously there are four paradigms of unified memory programming:

- *Full support for explicit managed memory allocations*
- *Full support for all allocations with software coherence*
- *Full support for all allocations with hardware coherence*
- *Limited unified memory support*

The first three paradigms involving full unified memory support have very similar behavior and programming model and are covered in *Unified Memory on Devices with Full CUDA Unified Memory Support* with any differences highlighted.

The last paradigm, where unified memory support is limited, is discussed in detail in *Unified Memory on Windows, WSL, and Tegra*.

4.1.1. Unified Memory on Devices with Full CUDA Unified Memory Support

These systems include hardware-coherent memory systems, such as NVIDIA Grace Hopper and modern Linux systems with Heterogeneous Memory Management (HMM) enabled. HMM is a software-based memory management system, providing the same programming model as hardware-coherent memory systems.

Linux HMM requires Linux kernel version 6.1.24+, 6.2.11+ or 6.3+, devices with compute capability 7.5 or higher and a CUDA driver version 535+ installed with Open Kernel Modules.

Note

We refer to systems with a combined page table for both CPUs and GPUs as *hardware coherent* systems. Systems with separate page tables for CPUs and GPUs are referred to as *software-coherent*.

Hardware-coherent systems such as NVIDIA Grace Hopper offer a logically combined page table for both CPUs and GPUs, see *CPU and GPU Page Tables: Hardware Coherency vs. Software Coherency*. The following section only applies to hardware-coherent systems:

▶ *Access Counter Migration*

4.1.1.1 Unified Memory: In-Depth Examples

Systems with full CUDA unified memory support, see table *Overview of Unified Memory Paradigms*, allow the device to access any memory owned by the host process interacting with the device.

This section shows a few advanced use-cases, using a kernel that simply prints the first 8 characters of an input character array to the standard output stream:

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

The following tabs show various ways of how this kernel may be called with system-allocated memory:

Malloc

```
void test_malloc() {
  const char test_string[] = "Hello World";
  char* heap_data = (char*)malloc(sizeof(test_string));
  strncpy(heap_data, test_string, sizeof(test_string));
  kernel<<<1, 1>>>("malloc", heap_data);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
  free(heap_data);
}
```

Managed

```
void test_managed() {
  const char test_string[] = "Hello World";
  char* data;
  cudaMallocManaged(&data, sizeof(test_string));
  strncpy(data, test_string, sizeof(test_string));
  kernel<<<1, 1>>>("managed", data);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
  cudaFree(data);
}
```

Stack variable

```
void test_stack() {
  const char test_string[] = "Hello World";
  kernel<<<1, 1>>>("stack", test_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
```

(continues on next page)

(continued from previous page)

```
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

File-scope static variable

```
void test_static() {
  static const char test_string[] = "Hello World";
  kernel<<<1, 1>>>("static", test_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

Global-scope variable

```
const char global_string[] = "Hello World";

void test_global() {
  kernel<<<1, 1>>>("global", global_string);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

Global-scope extern variable

```
// declared in separate file, see below
extern char* ext_data;

void test_extern() {
  kernel<<<1, 1>>>("extern", ext_data);
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
}
```

```
/** This may be a non-CUDA file */
char* ext_data;
static const char global_string[] = "Hello World";

void __attribute__ ((constructor)) setup(void) {
  ext_data = (char*)malloc(sizeof(global_string));
  strncpy(ext_data, global_string, sizeof(global_string));
}

void __attribute__ ((destructor)) tear_down(void) {
  free(ext_data);
}
```

Note that for the extern variable, it could be declared and its memory owned and managed by a third-party library, which does not interact with CUDA at all.

Also note that stack variables as well as file-scope and global-scope variables can only be accessed through a pointer by the GPU. In this specific example, this is convenient because the character array is already declared as a pointer: `const char*`. However, consider the following example with a global-scope integer:

```
// this variable is declared at global scope
int global_variable;

__global__ void kernel_uncompilable() {
  // this causes a compilation error: global (__host__) variables must not
  // be accessed from __device__ / __global__ code
  printf("%d\n", global_variable);
}

// On systems with pageableMemoryAccess set to 1, we can access the address
// of a global variable. The below kernel takes that address as an argument
__global__ void kernel(int* global_variable_addr) {
  printf("%d\n", *global_variable_addr);
}
int main() {
  kernel<<<1, 1>>>(&global_variable);
  ...
  return 0;
}
```

In the example above, we need to ensure to pass a *pointer* to the global variable to the kernel instead of directly accessing the global variable in the kernel. This is because global variables without the `__managed__` specifier are declared as `__host__`-only by default, thus most compilers won't allow using these variables directly in device code as of now.

4.1.1.1.1 File-backed Unified Memory

Since systems with full CUDA unified memory support allow the device to access any memory owned by the host process, they can directly access file-backed memory.

Here, we show a modified version of the initial example shown in the previous section to use file-backed memory in order to print a string from the GPU, read directly from an input file. In the following example, the memory is backed by a physical file, but the example applies to memory-backed files too.

```
__global__ void kernel(const char* type, const char* data) {
  static const int n_char = 8;
  printf("%s - first %d characters: '", type, n_char);
  for (int i = 0; i < n_char; ++i) printf("%c", data[i]);
  printf("'\n");
}
```

```
void test_file_backed() {
  int fd = open(INPUT_FILE_NAME, O_RDONLY);
  ASSERT(fd >= 0, "Invalid file handle");
  struct stat file_stat;
  int status = fstat(fd, &file_stat);
  ASSERT(status >= 0, "Invalid file stats");
  char* mapped = (char*)mmap(0, file_stat.st_size, PROT_READ, MAP_PRIVATE, fd,
↪ 0);
  ASSERT(mapped != MAP_FAILED, "Cannot map file into memory");
  kernel<<<1, 1>>>("file-backed", mapped);
```

(continues on next page)

(continued from previous page)

```
  ASSERT(cudaDeviceSynchronize() == cudaSuccess,
    "CUDA failed with '%s'", cudaGetErrorString(cudaGetLastError()));
  ASSERT(munmap(mapped, file_stat.st_size) == 0, "Cannot unmap file");
  ASSERT(close(fd) == 0, "Cannot close file");
}
```

Note that on systems without the `hostNativeAtomicSupported` property (see *Host Native Atomics*) including systems with Linux HMM enabled, atomic accesses to file-backed memory are not supported.

4.1.1.1.2 Inter-Process Communication (IPC) with Unified Memory

Note

As of now, using IPC with unified memory can have significant performance implications.

Many applications prefer to manage one GPU per process, but still need to use unified memory, for example for over-subscription, and access it from multiple GPUs.

CUDA IPC (see *Interprocess Communication*) does not support managed memory: handles to this type of memory may not be shared through any of the mechanisms discussed in this section. On systems with full CUDA unified memory support, system-allocated memory is IPC capable. Once access to system-allocated memory has been shared with other processes, the same programming model applies, similar to *File-backed Unified Memory*.

See the following references for more information on various ways of creating IPC-capable system-allocated memory under Linux:

- mmap with MAP_SHARED
- POSIX IPC APIs
- Linux memfd_create .

Note that it is not possible to share memory between different hosts and their devices using this technique.

4.1.1.2 Performance Tuning

In order to achieve good performance with unified memory, it is important to:

- understand how paging works on your system, and how to avoid unnecessary page faults
- understand the various mechanisms allowing you to keep data local to the accessing processor
- consider tuning your application for the granularity of memory transfers of your system.

As general advice, performance hints (see *Performance Hints*) might provide improved performance, but using them incorrectly might degrade performance compared to the default behavior. Also note that any hint has a performance cost associated with it on the host, thus useful hints must at the very least improve performance enough to overcome this cost.

4.1.1.2.1 Memory Paging and Page Sizes

To better understand the performance implication of unified memory, it is important to understand virtual addressing, memory pages and page sizes. This sub-section attempts to define all necessary terms and explain why paging matters for performance.

All currently supported systems for unified memory use a virtual address space: this means that memory addresses used by an application represent a *virtual* location which might be *mapped* to a physical location where the memory actually resides.

All currently supported processors, including both CPUs and GPUs, additionally use memory *paging*. Because all systems use a virtual address space, there are two types of memory pages:

- Virtual pages: This represents a fixed-size contiguous chunk of virtual memory per process tracked by the operating system, which can be *mapped* into physical memory. Note that the virtual page is linked to the *mapping*: for example, a single virtual address might be mapped into physical memory using different page sizes.
- Physical pages: This represents a fixed-size contiguous chunk of memory the processor's main Memory Management Unit (MMU) supports and into which a virtual page can be mapped.

Currently, all x86_64 CPUs use a default physical page size of 4KiB. Arm CPUs support multiple physical page sizes - 4KiB, 16KiB, 32KiB and 64KiB - depending on the exact CPU. Finally, NVIDIA GPUs support multiple physical page sizes, but prefer 2MiB physical pages or larger. Note that these sizes are subject to change in future hardware.

The default page size of virtual pages usually corresponds to the physical page size, but an application may use different page sizes as long as they are supported by the operating system and the hardware. Typically, supported virtual page sizes must be powers of 2 and multiples of the physical page size.

The logical entity tracking the mapping of virtual pages into physical pages will be referred to as a *page table*, and each mapping of a given virtual page with a given virtual size to physical pages is called a *Page Table Entry (PTE)*. All supported processors provide specific caches for the page table to speed up the translation of virtual addresses to physical addresses. These caches are called *Translation Lookaside Buffers (TLBs)*.

There are two important aspects for performance tuning of applications:

- the choice of virtual page size,
- whether the system offers a combined page table used by both CPUs and GPUs, or separate page tables for each CPU and GPU individually.

4.1.1.2.1.1 Choosing the Right Page Size

In general, small page sizes lead to less (virtual) memory fragmentation but more TLB misses, whereas larger page sizes lead to more memory fragmentation but less TLB misses. Additionally, memory migration is generally more expensive with larger page sizes compared to smaller page sizes, because we typically migrate full memory pages. This can cause larger latency spikes in an application using large page sizes. See also the next section for more details on page faults.

One important aspect for performance tuning is that TLB misses are generally significantly more expensive on the GPU compared to the CPU. This means that if a GPU thread frequently accesses random locations of unified memory mapped using a small enough page size, it might be significantly slower compared to the same accesses to unified memory mapped using a large enough page size. While a similar effect might occur for a CPU thread randomly accessing a large area of memory mapped using a small page size, the slowdown is less pronounced, meaning that the application might want to trade-off this slowdown with having less memory fragmentation.

Note that in general, applications should not tune their performance to the physical page size of a given processor, since physical page sizes are subject to change depending on the hardware. The advice above only applies to virtual page sizes.

4.1.1.2.1.2 CPU and GPU Page Tables: Hardware Coherency vs. Software Coherency

Hardware-coherent systems such as NVIDIA Grace Hopper offer a logically combined page table for both CPUs and GPUs. This is important because in order to access system-allocated memory from the GPU, the GPU uses whichever page table entry was created by the CPU for the requested memory. If that page table entry uses the default CPU page size of 4KiB or 64KiB, accesses to large virtual memory areas will cause significant TLB misses, thus significant slowdowns.

On the other hand, on software-coherent systems where the CPUs and GPUs each have their own logical page table, different performance tuning aspects should be considered: in order to guarantee coherency, these systems usually use *page faults* in case a processor accesses a memory address mapped into the physical memory of a different processor. Such a page fault means that:

- It needs to be ensured that the currently owning processor (where the physical page currently resides) cannot access this page anymore, either by deleting the page table entry or updating it.
- It needs to be ensured that the processor requesting access can access this page, either by creating a new page table entry or updating and existing entry, such that it becomes valid/active.
- The physical page backing this virtual page must be moved/migrated to the processor requesting access: this can be an expensive operation, and the amount of work is proportional to the page size.

Overall, hardware-coherent systems provide significant performance benefits compared to software-coherent systems in cases where frequent concurrent accesses to the same memory page are made by both CPU and GPU threads:

- less page-faults: these systems do not need to use page-faults for emulating coherency or migrating memory,
- less contention: these systems are coherent at cache-line granularity instead of page-size granularity, that is, when there is contention from multiple processors within a cache line, only the cache line is exchanged which is much smaller than the smallest page-size, and when the different processors access different cache-lines within a page, then there is no contention.

This impacts the performance of the following scenarios:

- atomic updates to the same address concurrently from both CPUs and GPUs
- signaling a GPU thread from a CPU thread or vice-versa.

4.1.1.2.2 Direct Unified Memory Access from the Host

Some devices have hardware support for coherent reads, stores and atomic accesses from the host on GPU-resident unified memory. These devices have the attribute `cudaDevAttrDirectManagedMemAccessFromHost` set to 1. Note that all hardware-coherent systems have this attribute set for NVLink-connected devices. On these systems, the host has direct access to GPU-resident memory without page faults and data migration. Note that with CUDA managed memory, the `cudaMemAdviseSetAccessedBy` hint with location type `cudaMemLocationTypeHost` is necessary to enable this direct access without page faults, see example below.

System Allocator

```
__global__ void write(int *ret, int a, int b) {
  ret[threadIdx.x] = a + b + threadIdx.x;
}

__global__ void append(int *ret, int a, int b) {
  ret[threadIdx.x] += a + b + threadIdx.x;
}

void test_malloc() {
  int *ret = (int*)malloc(1000 * sizeof(int));
  // for shared page table systems, the following hint is not necesary
  cudaMemLocation location = {.type = cudaMemLocationTypeHost};
  cudaMemAdvise(ret, 1000 * sizeof(int), cudaMemAdviseSetAccessedBy,
↪location);

  write<<< 1, 1000 >>>(ret, 10, 100);            // pages populated in GPU
↪memory
  cudaDeviceSynchronize();
  for(int i = 0; i < 1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);        //
↪directManagedMemAccessFromHost=1: CPU accesses GPU memory directly without
↪migrations
                                                  //
↪directManagedMemAccessFromHost=0: CPU faults and triggers device-to-host
↪migrations
  append<<< 1, 1000 >>>(ret, 10, 100);            //
↪directManagedMemAccessFromHost=1: GPU accesses GPU memory without migrations
  cudaDeviceSynchronize();                        //
↪directManagedMemAccessFromHost=0: GPU faults and triggers host-to-device
↪migrations
  free(ret);
}
```

Managed

```
__global__ void write(int *ret, int a, int b) {
  ret[threadIdx.x] = a + b + threadIdx.x;
}

__global__ void append(int *ret, int a, int b) {
  ret[threadIdx.x] += a + b + threadIdx.x;
}

void test_managed() {
  int *ret;
  cudaMallocManaged(&ret, 1000 * sizeof(int));
  cudaMemLocation location = {.type = cudaMemLocationTypeHost};
  cudaMemAdvise(ret, 1000 * sizeof(int), cudaMemAdviseSetAccessedBy,
↪location);  // set direct access hint

  write<<< 1, 1000 >>>(ret, 10, 100);            // pages populated in GPU
```

(continues on next page)

(continued from previous page)

```
↪memory
  cudaDeviceSynchronize();
  for(int i = 0; i < 1000; i++)
      printf("%d: A+B = %d\n", i, ret[i]);        //
↪directManagedMemAccessFromHost=1: CPU accesses GPU memory directly without
↪migrations
                                                  //
↪directManagedMemAccessFromHost=0: CPU faults and triggers device-to-host
↪migrations
  append<<< 1, 1000 >>>(ret, 10, 100);            //
↪directManagedMemAccessFromHost=1: GPU accesses GPU memory without migrations
  cudaDeviceSynchronize();                        //
↪directManagedMemAccessFromHost=0: GPU faults and triggers host-to-device
↪migrations
  cudaFree(ret);
```

After `write` kernel is completed, `ret` will be created and initialized in GPU memory. Next, the CPU will access `ret` followed by `append` kernel using the same `ret` memory again. This code will show different behavior depending on the system architecture and support of hardware coherency:

- on systems with `directManagedMemAccessFromHost=1`: CPU accesses to the managed buffer will not trigger any migrations; the data will remain resident in GPU memory and any subsequent GPU kernels can continue to access it directly without inflicting faults or migrations
- on systems with `directManagedMemAccessFromHost=0`: CPU accesses to the managed buffer will page fault and initiate data migration; any GPU kernel trying to access the same data first time will page fault and migrate pages back to GPU memory.

4.1.1.2.3 Host Native Atomics

Some devices, including NVLink-connected devices of hardware-coherent systems, support hardware-accelerated atomic accesses to CPU-resident memory. This implies that atomic accesses to host memory do not have to be emulated with a page fault. For these devices, the attribute `cudaDevAttrHostNativeAtomicSupported` is set to 1.

4.1.1.2.4 Atomic Accesses and Synchronization Primitives

CUDA unified memory supports all atomic operations available to host and device threads, enabling all threads to cooperate by concurrently accessing the same shared memory location. The libcu++ library provides many heterogeneous synchronization primitives tuned for concurrent use between host and device threads, including `cuda::atomic`, `cuda::atomic_ref`, `cuda::barrier`, `cuda::semaphore`, among many others.

On software-coherent systems, atomic accesses from the device to file-backed host memory are not supported. The following example code is valid on hardware-coherent systems but exhibits undefined behavior on other systems:

```
#include <cuda/atomic>

#include <cstdio>
#include <fcntl.h>
#include <sys/mman.h>
```

(continues on next page)

(continued from previous page)

```
#define ERR(msg, ...) { fprintf(stderr, msg, ##__VA_ARGS__); return EXIT_
↪FAILURE; }

__global__ void kernel(int* ptr) {
  cuda::atomic_ref{*ptr}.store(2);
}

int main() {
  // this will be closed/deleted by default on exit
  FILE* tmp_file = tmpfile64();
  // need to allocate space in the file, we do this with posix_fallocate here
  int status = posix_fallocate(fileno(tmp_file), 0, 4096);
  if (status != 0) ERR("Failed to allocate space in temp file\n");
  int* ptr = (int*)mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_PRIVATE,
↪fileno(tmp_file), 0);
  if (ptr == MAP_FAILED) ERR("Failed to map temp file\n");

  // initialize the value in our file-backed memory
  *ptr = 1;
  printf("Atom value: %d\n", *ptr);

  // device and host thread access ptr concurrently, using cuda::atomic_ref
  kernel<<<1, 1>>>(ptr);
  while (cuda::atomic_ref{*ptr}.load() != 2);
  // this will always be 2
  printf("Atom value: %d\n", *ptr);

  return EXIT_SUCCESS;
}
```

On software-coherent systems, atomic accesses to unified memory may incur page faults which can lead to significant latencies. Note that this is not the case for all GPU atomics to CPU memory on these systems: operations listed by `nvidia-smi -q | grep "Atomic Caps Outbound"` may avoid page faults.

On hardware-coherent systems, atomics between host and device do not require page faults, but may still fault for other reasons that can cause any memory access to fault.

4.1.1.2.5 Memcpy()/Memset() Behavior With Unified Memory

`cudaMemcpy*()` and `cudaMemset*()` accept any unified memory pointer as arguments.

For `cudaMemcpy*()`, the direction specified as `cudaMemcpyKind` is a performance hint, which can have a higher performance impact if any of the arguments is a unified memory pointer.

Thus, it is recommended to follow the following performance advice:

- When the physical location of unified memory is known, use an accurate `cudaMemcpyKind` hint.
- Prefer `cudaMemcpyDefault` over an inaccurate `cudaMemcpyKind` hint.
- Always use populated (initialized) buffers: avoid using these APIs to initialize memory.
- Avoid using `cudaMemcpy*()` if both pointers point to system-allocated memory: launch a kernel or use a CPU memory copy algorithm such as `std::memcpy` instead.

4.1.1.2.6 Overview of Memory Allocators for Unified Memory

For systems with full CUDA unified memory support various different allocators may be used to allocate unified memory. The following table shows an overview of a selection of allocators with their respective features. Note that all information in this section is subject to change in future CUDA versions.

Table 7: Overview of unified memory support of different allocators

API	**Placement Policy**	**Accessible From**	**Migrate Based On Access**[2]	**Page Sizes**[4][5]
`malloc`, `new`, `mmap`	First touch/hi	CPU, GPU	Yes[3]	System or huge page size[6]
`cudaMallocManaged`	First touch/hi	CPU, GPU	Yes	CPU resident: system page size GPU resident: 2MB
`cudaMalloc`	GPU	GPU	No	GPU page size: 2MB
`cudaMallocHost`, `cudaHostAlloc`, `cudaHostRegister`	CPU	CPU, GPU	No	Mapped by CPU: system page size Mapped by GPU: 2MB
Memory pools, location type host: `cuMemCreate`, `cudaMemPoolCreate`	CPU	CPU, GPU	No	Mapped by CPU: system page size Mapped by GPU: 2MB
Memory pools, location type device: `cuMemCreate`, `cudaMemPoolCreate`, `cudaMallocAsync`	GPU	GPU	No	2MB

The table *Overview of unified memory support of different allocators* shows the difference in semantics of several allocators that may be considered to allocate data accessible from multiple processors at a time, including host and device. For additional details about `cudaMemPoolCreate`, see the *Memory Pools* section, for additional details about `cuMemCreate`, see the *Virtual Memory Management* section.

On hardware-coherent systems where device memory is exposed as a NUMA domain to the system, special allocators such as `numa_alloc_on_node` may be used to pin memory to the given NUMA node, either host or device. This memory is accessible from both host and device and does not migrate. Similarly, `mbind` can be used to pin memory to the given NUMA node(s), and can cause file-backed memory to be placed on the given NUMA node(s) before it is first accessed.

The following applies to allocators of memory that is shared:

[2] This feature can be overridden with `cudaMemAdvise`. Even if access-based migrations are disabled, if the backing memory space is full, memory might migrate.

[4] The default system page size is 4KiB or 64KiB on most systems, unless huge page size was explicitly specified (for example, with `mmap MAP_HUGETLB` / `MAP_HUGE_SHIFT`). In this case, any huge page size configured on the system is supported.

[5] Page-sizes for GPU-resident memory may evolve in future CUDA versions.

[1] For `mmap`, file-backed memory is placed on the CPU by default, unless specified otherwise through `cudaMemAdviseSetPreferredLocation` (or `mbind`, see bullet points below).

[3] File-backed memory will not migrate based on access.

[6] Currently huge page sizes may not be kept when migrating memory to the GPU or placing it through first-touch on the GPU.

- System allocators such as mmap allow sharing the memory between processes using the MAP_SHARED flag. This is supported in CUDA and can be used to share memory between different devices connected to the same host. However, this is currently not supported for sharing memory between multiple hosts as well as multiple devices. See *Inter-Process Communication (IPC) with Unified Memory* for details.
- For access to unified memory or other CUDA memory through a network on multiple hosts, consult the documentation of the communication library used, for example NCCL, NVSHMEM, OpenMPI, UCX, etc.

4.1.1.2.7 Access Counter Migration

On hardware-coherent systems, the access counters feature keeps track of the frequency of access that a GPU makes to memory located on other processors. This is needed to ensure memory pages are moved to the physical memory of the processor that is accessing the pages most frequently. It can guide migrations between CPU and GPU, as well as between peer GPUs, a process called access counter migration.

Starting with CUDA 12.4, access counters are supported system-allocated memory. Note that file-backed memory does not migrate based on access. For system-allocated memory, access counters migration can be switched on by using the cudaMemAdviseSetAccessedBy hint to a device with the corresponding device id. If access counters are on, one can use cudaMemAdviseSetPreferredLocation set to host to prevent migrations. Per default cudaMallocManaged migrates based on a fault-and-migrate mechanism.[7]

The driver may also use access counters for more efficient thrashing mitigation or memory oversubscription scenarios.

4.1.1.2.8 Avoid Frequent Writes to GPU-Resident Memory from the CPU

If the host accesses unified memory, cache misses may introduce more traffic than expected between host and device. Many CPU architectures require all memory operations to go through the cache hierarchy, including writes. If system memory is resident on the GPU, this means that frequent writes by the CPU to this memory can cause cache misses, thus transferring the data first from the GPU to CPU before writing the actual value into the requested memory range. On software-coherent systems, this may introduce additional page faults, while on hardware-coherent systems, it may cause higher latencies between CPU operations. Thus, in order to share data produced by the host with the device, consider writing to CPU-resident memory and reading the values directly from the device. The code below shows how to achieve this with unified memory.

System Allocator

```
size_t data_size = sizeof(int);
int* data = (int*)malloc(data_size);
// ensure that data stays local to the host and avoid faults
cudaMemLocation location = {.type = cudaMemLocationTypeHost};
cudaMemAdvise(data, data_size, cudaMemAdviseSetPreferredLocation, location);
cudaMemAdvise(data, data_size, cudaMemAdviseSetAccessedBy, location);

// frequent exchanges of small data: if the CPU writes to CPU-resident
↪memory,
// and GPU directly accesses that data, we can avoid the CPU caches re-
```

(continues on next page)

[7] Current systems allow the use of access-counter migration with managed memory when the accessed-by device hint is set. This is an implementation detail and should not be relied on for future compatibility.

(continued from previous page)

```
↪loading
  // data if it was evicted in between writes
  for (int i = 0; i < 10; ++i) {
    *data = 42 + i;
    kernel<<<1, 1>>>(data);
    cudaDeviceSynchronize();
    // CPU cache potentially evicted data here
  }
  free(data);
```

Managed

```
  int* data;
  size_t data_size = sizeof(int);
  cudaMallocManaged(&data, data_size);
  // ensure that data stays local to the host and avoid faults
  cudaMemLocation location = {.type = cudaMemLocationTypeHost};
  cudaMemAdvise(data, data_size, cudaMemAdviseSetPreferredLocation, location);
  cudaMemAdvise(data, data_size, cudaMemAdviseSetAccessedBy, location);

  // frequent exchanges of small data: if the CPU writes to CPU-resident
↪memory,
  // and GPU directly accesses that data, we can avoid the CPU caches re-
↪loading
  // data if it was evicted in between writes
  for (int i = 0; i < 10; ++i) {
    *data = 42 + i;
    kernel<<<1, 1>>>(data);
    cudaDeviceSynchronize();
    // CPU cache potentially evicted data here
  }
  cudaFree(data);
```

4.1.1.2.9 Exploiting Asynchronous Access to System Memory

If an application needs to share results from work on the device with the host, there are several possible options:

1. The device writes its result to GPU-resident memory, the result is transferred using `cudaMemcpy*`, and the host reads the transferred data.
2. The device directly writes its result to CPU-resident memory, and the host reads that data.
3. The device writes to GPU-resident memory, and the host directly accesses that data.

If independent work can be scheduled on the device while the result is transferred/accessed by the host, options 1 or 3 are preferred. If the device is starved until the host has accessed the result, option 2 might be preferred. This is because the device can generally write at a higher bandwidth than the host can read, unless many host threads are used to read the data.

1. **Explicit Copy**

```
void exchange_explicit_copy(cudaStream_t stream) {
  int* data, *host_data;
  size_t n_bytes = sizeof(int) * 16;
  // allocate receiving buffer
  host_data = (int*)malloc(n_bytes);
  // allocate, since we touch on the device first, will be GPU-resident
  cudaMallocManaged(&data, n_bytes);
  kernel<<<1, 16, 0, stream>>>(data);
  // launch independent work on the device
  // other_kernel<<<1024, 256, 0, stream>>>(other_data, ...);
  // transfer to host
  cudaMemcpyAsync(host_data, data, n_bytes, cudaMemcpyDeviceToHost, stream);
  // sync stream to ensure data has been transferred
  cudaStreamSynchronize(stream);
  // read transferred data
  printf("Got values %d - %d from GPU\n", host_data[0], host_data[15]);
  cudaFree(data);
  free(host_data);
}
```

2. **Device Direct Write**

```
void exchange_device_direct_write(cudaStream_t stream) {
  int* data;
  size_t n_bytes = sizeof(int) * 16;
  // allocate receiving buffer
  cudaMallocManaged(&data, n_bytes);
  // ensure that data is mapped and resident on the host
  cudaMemLocation location = {.type = cudaMemLocationTypeHost};
  cudaMemAdvise(data, n_bytes, cudaMemAdviseSetPreferredLocation, location);
  cudaMemAdvise(data, n_bytes, cudaMemAdviseSetAccessedBy, location);
  kernel<<<1, 16, 0, stream>>>(data);
  // sync stream to ensure data has been transferred
  cudaStreamSynchronize(stream);
  // read transferred data
  printf("Got values %d - %d from GPU\n", data[0], data[15]);
  cudaFree(data);
}
```

3. **Host Direct Read**

```
void exchange_host_direct_read(cudaStream_t stream) {
  int* data;
  size_t n_bytes = sizeof(int) * 16;
  // allocate receiving buffer
  cudaMallocManaged(&data, n_bytes);
  // ensure that data is mapped and resident on the device
  cudaMemLocation device_loc = {};
  cudaGetDevice(&device_loc.id);
  device_loc.type = cudaMemLocationTypeDevice;
```

(continues on next page)

(continued from previous page)

```
cudaMemAdvise(data, n_bytes, cudaMemAdviseSetPreferredLocation, device_loc);
cudaMemAdvise(data, n_bytes, cudaMemAdviseSetAccessedBy, device_loc);
kernel<<<1, 16, 0, stream>>>(data);
// launch independent work on the GPU
// other_kernel<<<1024, 256, 0, stream>>>(other_data, ...);
// sync stream to ensure data may be accessed (has been written by device)
cudaStreamSynchronize(stream);
// read data directly from host
printf("Got values %d - %d from GPU\n", data[0], data[15]);
cudaFree(data);
```

Finally, in the Explicit Copy example above, instead of using `cudaMemcpy*` to transfer data, one could use a host or device kernel to perform this transfer explicitly. For contiguous data, using the CUDA copy-engines is preferred because operations performed by copy-engines can be overlapped with work on both the host and device. Copy-engines might be used in `cudaMemcpy*` and `cudaMemPrefetchAsync` APIs, but there is no guarantee. that copy-engines are used with `cudaMemcpy*` API calls. For the same reason, explicitly copy is preferred over direct host read for large enough data: if both host and device perform work that does not saturate their respective memory systems, the transfer can be performed by the copy-engines concurrently with the work performed by both host and device.

Copy-engines are generally used for both transfers between host and device as well as between peer devices within an NVLink-connected system. Due to the limited total number of copy-engines, some systems may have a lower bandwidth of `cudaMemcpy*` compared to using the device to explicitly perform the transfer. In such a case, if the transfer is in the critical path of the application, it may be preferred to use an explicit device-based transfer.

4.1.2. Unified Memory on Devices with only CUDA Managed Memory Support

For devices with compute capability 6.x or higher but without pageable memory access, see table *Overview of Unified Memory Paradigms*, CUDA managed memory is fully supported and coherent but the GPU cannot access system-allocated memory. The programming model and performance tuning of unified memory is largely similar to the model as described in the section, *Unified Memory on Devices with Full CUDA Unified Memory Support*, with the notable exception that system allocators cannot be used to allocate memory. Thus, the following list of sub-sections do not apply:

- *Unified Memory: In-Depth Examples*
- *CPU and GPU Page Tables: Hardware Coherency vs. Software Coherency*
- *Atomic Accesses and Synchronization Primitives*
- *Access Counter Migration*
- *Avoid Frequent Writes to GPU-Resident Memory from the CPU*
- *Exploiting Asynchronous Access to System Memory*

4.1.3. Unified Memory on Windows, WSL, and Tegra

Note

This section is only looking at devices with compute capability lower than 6.0 or Windows platforms, devices with `concurrentManagedAccess` property set to 0.

Devices with compute capability lower than 6.0 or Windows platforms, devices with `concurrentManagedAccess` property set to 0, see *Overview of Unified Memory Paradigms*, support CUDA managed memory with the following limitations:

- **Data Migration and Coherency**: Fine-grained movement of the managed data to GPU on-demand is not supported. Whenever a GPU kernel is launched all managed memory generally has to be transferred to GPU memory to avoid faulting on memory access. Page faulting is only supported from the CPU side.
- **GPU Memory Oversubscription**: They cannot allocate more managed memory than the physical size of GPU memory.
- **Coherency and Concurrency**: Simultaneous access to managed memory is not possible, because coherence could not be guaranteed if the CPU accessed a unified memory allocation while a GPU kernel is active because of the missing GPU page faulting mechanism.

4.1.3.1 Multi-GPU

On systems with devices of compute capabilities lower than 6.0 or Windows platforms managed allocations are automatically visible to all GPUs in a system via the peer-to-peer capabilities of the GPUs. Managed memory allocations behave similar to unmanaged memory allocated using `cudaMalloc()`: the current active device is the home for the physical allocation but other GPUs in the system will access the memory at reduced bandwidth over the PCIe bus.

On Linux the managed memory is allocated in GPU memory as long as all GPUs that are actively being used by a program have the peer-to-peer support. If at any time the application starts using a GPU that doesn't have peer-to-peer support with any of the other GPUs that have managed allocations on them, then the driver will migrate all managed allocations to system memory. In this case, all GPUs experience PCIe bandwidth restrictions.

On Windows, if peer mappings are not available (for example, between GPUs of different architectures), then the system will automatically fall back to using mapped memory, regardless of whether both GPUs are actually used by a program. If only one GPU is actually going to be used, it is necessary to set the `CUDA_VISIBLE_DEVICES` environment variable before launching the program. This constrains which GPUs are visible and allows managed memory to be allocated in GPU memory.

Alternatively, on Windows users can also set `CUDA_MANAGED_FORCE_DEVICE_ALLOC` to a non-zero value to force the driver to always use device memory for physical storage. When this environment variable is set to a non-zero value, all devices used in that process that support managed memory have to be peer-to-peer compatible with each other. The error `::cudaErrorInvalidDevice` will be returned if a device that supports managed memory is used and it is not peer-to-peer compatible with any of the other managed memory supporting devices that were previously used in that process, even if `::cudaDeviceReset` has been called on those devices. These environment variables are described in *CUDA Environment Variables*.

4.1.3.2 Coherency and Concurrency

To ensure coherency the unified memory programming model puts constraints on data accesses while both the CPU and GPU are executing concurrently. In effect, the GPU has exclusive access to all managed data and the CPU is not permitted to access it, while any kernel operation is executing, regardless of whether the specific kernel is actively using the data. Concurrent CPU/GPU accesses, even to different managed memory allocations, will cause a segmentation fault because the page is considered inaccessible to the CPU.

For example the following code runs successfully on devices of compute capability 6.x due to the GPU page faulting capability which lifts all restrictions on simultaneous access but fails on on pre-6.x architectures and Windows platforms because the GPU program kernel is still active when the CPU touches y:

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}
int main() {
    kernel<<< 1, 1 >>>();
    y = 20;            // Error on GPUs not supporting concurrent access

    cudaDeviceSynchronize();
    return  0;
}
```

The program must explicitly synchronize with the GPU before accessing y (regardless of whether the GPU kernel actually touches y (or any managed data at all):

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}
int main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
    y = 20;            //  Success on GPUs not supporting concurrent access
    return  0;
}
```

Note that any function call that logically guarantees the GPU completes its work is valid to ensure logically that the GPU work is completed, see *Explicit Synchronization*.

Note that if memory is dynamically allocated with `cudaMallocManaged()` or `cuMemAllocManaged()` while the GPU is active, the behavior of the memory is unspecified until additional work is launched or the GPU is synchronized. Attempting to access the memory on the CPU during this time may or may not cause a segmentation fault. This does not apply to memory allocated using the flag `cudaMemAttachHost` or `CU_MEM_ATTACH_HOST`.

4.1.3.3 Stream Associated Unified Memory

The CUDA programming model provides streams as a mechanism for programs to indicate dependence and independence among kernel launches. Kernels launched into the same stream are guaranteed to execute consecutively, while kernels launched into different streams are permitted to execute concurrently. See section *CUDA Streams*.

4.1.3.3.1 Stream Callbacks

It is legal for the CPU to access managed data from within a stream callback, provided no other stream that could potentially be accessing managed data is active on the GPU. In addition, a callback that is not followed by any device work can be used for synchronization: for example, by signaling a condition variable from inside the callback; otherwise, CPU access is valid only for the duration of the callback(s). There are several important points of note:

1. It is always permitted for the CPU to access non-managed mapped memory data while the GPU is active.
2. The GPU is considered active when it is running any kernel, even if that kernel does not make use of managed data. If a kernel might use data, then access is forbidden
3. There are no constraints on concurrent inter-GPU access of managed memory, other than those that apply to multi-GPU access of non-managed memory.
4. There are no constraints on concurrent GPU kernels accessing managed data.

Note how the last point allows for races between GPU kernels, as is currently the case for non-managed GPU memory. In the perspective of the GPU, managed memory functions are identical to non-managed memory. The following code example illustrates these points:

```
int main() {
    cudaStream_t stream1, stream2;
    cudaStreamCreate(&stream1);
    cudaStreamCreate(&stream2);
    int *non_managed, *managed, *also_managed;
    cudaMallocHost(&non_managed, 4);    // Non-managed, CPU-accessible memory
    cudaMallocManaged(&managed, 4);
    cudaMallocManaged(&also_managed, 4);
    // Point 1: CPU can access non-managed data.
    kernel<<< 1, 1, 0, stream1 >>>(managed);
    *non_managed = 1;
    // Point 2: CPU cannot access any managed data while GPU is busy,
    //          unless concurrentManagedAccess = 1
    // Note we have not yet synchronized, so "kernel" is still active.
    *also_managed = 2;      // Will issue segmentation fault
    // Point 3: Concurrent GPU kernels can access the same data.
    kernel<<< 1, 1, 0, stream2 >>>(managed);
    // Point 4: Multi-GPU concurrent access is also permitted.
    cudaSetDevice(1);
    kernel<<< 1, 1 >>>(managed);
    return  0;
}
```

4.1.3.3.2 Managed memory associated to streams allows for finer-grained control

Unified memory builds upon the stream-independence model by allowing a CUDA program to explicitly associate managed allocations with a CUDA stream. In this way, the programmer indicates the use of data by kernels based on whether they are launched into a specified stream or not. This enables opportunities for concurrency based on program-specific data access patterns. The function to control this behavior is:

```
cudaError_t cudaStreamAttachMemAsync(cudaStream_t stream,
                                     void *ptr,
```

(continues on next page)

(continued from previous page)

```
                                                        size_t length=0,
                                                        unsigned int flags=0);
```

The `cudaStreamAttachMemAsync()` function associates length bytes of memory starting from ptr with the specified stream. This allows CPU access to that memory region so long as all operations in stream have completed, regardless of whether other streams are active. In effect, this constrains exclusive ownership of the managed memory region by an active GPU to per-stream activity instead of whole-GPU activity. Most importantly, if an allocation is not associated with a specific stream, it is visible to all running kernels regardless of their stream. This is the default visibility for a `cudaMallocManaged()` allocation or a `__managed__` variable; hence, the simple-case rule that the CPU may not touch the data while any kernel is running.

Note

By associating an allocation with a specific stream, the program makes a guarantee that only kernels launched into that stream will touch that data. No error checking is performed by the unified memory system.

Note

In addition to allowing greater concurrency, the use of `cudaStreamAttachMemAsync()` can enable data transfer optimizations within the unified memory system that may affect latencies and other overhead.

The following example shows how to explicitly associate y with host accessibility, thus enabling access at all times from the CPU. (Note the absence of `cudaDeviceSynchronize()` after the kernel call.) Accesses to y by the GPU running kernel will now produce undefined results.

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}
int main() {
    cudaStream_t stream1;
    cudaStreamCreate(&stream1);
    cudaStreamAttachMemAsync(stream1, &y, 0, cudaMemAttachHost);
    cudaDeviceSynchronize();          // Wait for Host attachment to occur.
    kernel<<< 1, 1, 0, stream1 >>>(); // Note: Launches into stream1.
    y = 20;                           // Success – a kernel is running but “y”
                                      // has been associated with no stream.

    return  0;
}
```

4.1.3.3.3 A more elaborate example on multithreaded host programs

The primary use for `cudaStreamAttachMemAsync()` is to enable independent task parallelism using CPU threads. Typically in such a program, a CPU thread creates its own stream for all work that it generates because using CUDA’s NULL stream would cause dependencies between threads. The default global visibility of managed data to any GPU stream can make it difficult to avoid interactions between CPU threads in a multi-threaded program. Function `cudaStreamAttachMemAsync()`

is therefore used to associate a thread's managed allocations with that thread's own stream, and the association is typically not changed for the life of the thread. Such a program would simply add a single call to cudaStreamAttachMemAsync() to use unified memory for its data accesses:

```
// This function performs some task, in its own , in its own private stream and
↪can be run in parallel
void run_task(int *in, int *out, int length) {
    // Create a stream for us to use.
    cudaStream_t stream;
    cudaStreamCreate(&stream);
    // Allocate some managed data and associate with our stream.
    // Note the use of the host-attach flag to cudaMallocManaged();
    // we then associate the allocation with our stream so that
    // our GPU kernel launches can access it.
    int *data;
    cudaMallocManaged((void **)&data, length, cudaMemAttachHost);
    cudaStreamAttachMemAsync(stream, data);
    cudaStreamSynchronize(stream);
    // Iterate on the data in some way, using both Host & Device.
    for(int i=0; i<N; i++) {
        transform<<< 100, 256, 0, stream >>>(in, data, length);
        cudaStreamSynchronize(stream);
        host_process(data, length);    // CPU uses managed data.
        convert<<< 100, 256, 0, stream >>>(out, data, length);
    }
    cudaStreamSynchronize(stream);
    cudaStreamDestroy(stream);
    cudaFree(data);
}
```

In this example, the allocation-stream association is established just once, and then data is used repeatedly by both the host and device. The result is much simpler code than occurs with explicitly copying data between host and device, although the result is the same.

The function cudaMallocManaged() specifies the cudaMemAttachHost flag, which creates an allocation that is initially invisible to device-side execution. (The default allocation would be visible to all GPU kernels on all streams.) This ensures that there is no accidental interaction with another thread's execution in the interval between the data allocation and when the data is acquired for a specific stream.

Without this flag, a new allocation would be considered in-use on the GPU if a kernel launched by another thread happens to be running. This might impact the thread's ability to access the newly allocated data from the CPU before it is able to explicitly attach it to a private stream. To enable safe independence between threads, therefore, allocations should be made specifying this flag.

An alternative would be to place a process-wide barrier across all threads after the allocation has been attached to the stream. This would ensure that all threads complete their data/stream associations before any kernels are launched, avoiding the hazard. A second barrier would be needed before the stream is destroyed because stream destruction causes allocations to revert to their default visibility. The cudaMemAttachHost flag exists both to simplify this process, and because it is not always possible to insert global barriers where required.

4.1.3.3.4 Data Movement of Stream Associated Unified Memory

Memcpy()/Memset() with stream associated unified memory behaves different on devices where `concurrentManagedAccess` is not set, the following rules apply:

If `cudaMemcpyHostTo*` is specified and the source data is unified memory, then it will be accessed from the host if it is coherently accessible from the host in the copy stream *(1)*; otherwise it will be accessed from the device. Similar rules apply to the destination when `cudaMemcpy*ToHost` is specified and the destination is unified memory.

If `cudaMemcpyDeviceTo*` is specified and the source data is unified memory, then it will be accessed from the device. The source must be coherently accessible from the device in the copy stream *(2)*; otherwise, an error is returned. Similar rules apply to the destination when `cudaMemcpy*ToDevice` is specified and the destination is unified memory.

If `cudaMemcpyDefault` is specified, then unified memory will be accessed from the host either if it cannot be coherently accessed from the device in the copy stream *(2)* or if the preferred location for the data is `cudaCpuDeviceId` and it can be coherently accessed from the host in the copy stream *(1)*; otherwise, it will be accessed from the device.

When using `cudaMemset*()` with unified memory, the data must be coherently accessible from the device in the stream being used for the `cudaMemset()` operation *(2)*; otherwise, an error is returned.

When data is accessed from the device either by `cudaMemcpy*` or `cudaMemset*`, the stream of operation is considered to be active on the GPU. During this time, any CPU access of data that is associated with that stream or data that has global visibility, will result in a segmentation fault if the GPU has a zero value for the device attribute `concurrentManagedAccess`. The program must synchronize appropriately to ensure the operation has completed before accessing any associated data from the CPU.

1. Coherently accessible from the host in a given stream means that the memory neither has global visibility nor is it associated with the given stream.
2. Coherently accessible from the device in a given stream means that the memory either has global visibility or is associated with the given stream.

4.1.4. Performance Hints

Performance hints allow programmers to provide CUDA with more information about unified memory usage. CUDA uses performance hints to managed memory more efficiently and improve application performance. Performance hints never impact the correctness of an application. Performance hints only affect performance.

Note

Applications should only use unified memory performance hints if they improve performance.

Performance hints may be used on any unified memory allocation, including CUDA managed memory. On systems with full CUDA unified memory support, performance hints can be applied to all system-allocated memory.

4.1.4.1 Data Prefetching

The cudaMemPrefetchAsync API is an asynchronous stream-ordered API that may migrate data to reside closer to the specified processor. The data may be accessed while it is being prefetched. The migration does not begin until all prior operations in the stream have completed, and completes before any subsequent operation in the stream.

```
cudaError_t cudaMemPrefetchAsync(const void *devPtr,
                                 size_t count,
                                 struct cudaMemLocation location,
                                 unsigned int flags,
                                 cudaStream_t stream=0);
```

A memory region containing [devPtr, devPtr + count) may be migrated to the destination device location.id if location.type is cudaMemLocationTypeDevice, or CPU if location.type is cudaMemLocationTypeHost, when the prefetch task is executed in the given stream. For details on flags, see the current CUDA Runtime API documentation.

Consider the simple code example below:

System Allocator

```
void test_prefetch_sam(const cudaStream_t& s) {
  // initialize data on CPU
  char *data = (char*)malloc(dataSizeBytes);
  init_data(data, dataSizeBytes);
  cudaMemLocation location = {.type = cudaMemLocationTypeDevice, .id =
↪myGpuId};

  // encourage data to move to GPU before use
  const unsigned int flags = 0;
  cudaMemPrefetchAsync(data, dataSizeBytes, location, flags, s);

  // use data on GPU
  const unsigned num_blocks = (dataSizeBytes + threadsPerBlock - 1) /
↪threadsPerBlock;
  mykernel<<<num_blocks, threadsPerBlock, 0, s>>>(data, dataSizeBytes);

  // encourage data to move back to CPU
  location = {.type = cudaMemLocationTypeHost};
  cudaMemPrefetchAsync(data, dataSizeBytes, location, flags, s);

  cudaStreamSynchronize(s);

  // use data on CPU
  use_data(data, dataSizeBytes);
  free(data);
}
```

Managed

```
void test_prefetch_managed(const cudaStream_t& s) {
  // initialize data on CPU
```

(continues on next page)

(continued from previous page)

```
    char *data;
    cudaMallocManaged(&data, dataSizeBytes);
    init_data(data, dataSizeBytes);
    cudaMemLocation location = {.type = cudaMemLocationTypeDevice, .id =
↪myGpuId};

    // encourage data to move to GPU before use
    const unsigned int flags = 0;
    cudaMemPrefetchAsync(data, dataSizeBytes, location, flags, s);

    // use data on GPU
    const uinsigned num_blocks = (dataSizeBytes + threadsPerBlock - 1) /
↪threadsPerBlock;
    mykernel<<<num_blocks, threadsPerBlock, 0, s>>>(data, dataSizeBytes);

    // encourage data to move back to CPU
    location = {.type = cudaMemLocationTypeHost};
    cudaMemPrefetchAsync(data, dataSizeBytes, location, flags, s);

    cudaStreamSynchronize(s);

    // use data on CPU
    use_data(data, dataSizeBytes);
    cudaFree(data);
}
```

4.1.4.2 Data Usage Hints

When multiple processors simultaneously access the same data, cudaMemAdvise may be used to hint how the data at [devPtr, devPtr + count) will be accessed:

```
cudaError_t cudaMemAdvise(const void *devPtr,
                          size_t count,
                          enum cudaMemoryAdvise advice,
                          struct cudaMemLocation location);
```

The example shows how to use cudaMemAdvise:

```
    init_data(data, dataSizeBytes);
    cudaMemLocation location = {.type = cudaMemLocationTypeDevice, .id =
↪myGpuId};

    // encourage data to move to GPU before use
    const unsigned int flags = 0;
    cudaMemPrefetchAsync(data, dataSizeBytes, location, flags, s);

    // use data on GPU
    const uinsigned num_blocks = (dataSizeBytes + threadsPerBlock - 1) /
↪threadsPerBlock;
    mykernel<<<num_blocks, threadsPerBlock, 0, s>>>(data, dataSizeBytes);

    // encourage data to move back to CPU
```

(continues on next page)

(continued from previous page)

```
  location = {.type = cudaMemLocationTypeHost};
  cudaMemPrefetchAsync(data, dataSizeBytes, location, flags, s);

  cudaStreamSynchronize(s);

  // use data on CPU
  use_data(data, dataSizeBytes);
  cudaFree(data);
}
// test-prefetch-managed-end

static const int maxDevices = 1;
static const int maxOuterLoopIter = 3;
static const int maxInnerLoopIter = 4;

// test-advise-managed-begin
void test_advise_managed(cudaStream_t stream) {
  char *dataPtr;
  size_t dataSize = 64 * threadsPerBlock;  // 16 KiB
```

Where advice may take the following values:

- **cudaMemAdviseSetReadMostly:**
  This implies that the data is mostly going to be read from and only occasionally written to. In general, it allows trading off read bandwidth for write bandwidth on this region.
- **cudaMemAdviseSetPreferredLocation:**
  This hint sets the preferred location for the data to be the specified device's physical memory. This hint encourages the system to keep the data at the preferred location, but does not guarantee it. Passing in a value of cudaMemLocationTypeHost for location.type sets the preferred location as CPU memory. Other hints, like cudaMemPrefetchAsync, may override this hint and allow the memory to migrate away from its preferred location.
- **cudaMemAdviseSetAccessedBy:**
  In some systems, it may be beneficial for performance to establish a mapping into memory before accessing the data from a given processor. This hint tells the system that the data will be frequently accessed by location.id when location.type is cudaMemLocationTypeDevice, enabling the system to assume that creating these mappings pays off. This hint does not imply where the data should reside, but it can be combined with cudaMemAdviseSetPreferredLocation to specify that. On hardware-coherent systems, this hint switches on access counter migration, see *Access Counter Migration*.

Each advice can be also unset by using one of the following values: cudaMemAdviseUnsetReadMostly, cudaMemAdviseUnsetPreferredLocation and cudaMemAdviseUnsetAccessedBy.

The example shows how to use cudaMemAdvise:

System Allocator

```
void test_advise_sam(cudaStream_t stream) {
  char *dataPtr;
  size_t dataSize = 64 * threadsPerBlock;  // 16 KiB

  // Allocate memory using malloc or cudaMallocManaged
```

(continues on next page)

(continued from previous page)

```
  dataPtr = (char*)malloc(dataSize);

  // Set the advice on the memory region
  cudaMemLocation loc = {.type = cudaMemLocationTypeDevice, .id = myGpuId};
  cudaMemAdvise(dataPtr, dataSize, cudaMemAdviseSetReadMostly, loc);

  int outerLoopIter = 0;
  while (outerLoopIter < maxOuterLoopIter) {
    // The data is written by the CPU each outer loop iteration
    init_data(dataPtr, dataSize);

    // The data is made available to all GPUs by prefetching.
    // Prefetching here causes read duplication of data instead
    // of data migration
    cudaMemLocation location;
    location.type = cudaMemLocationTypeDevice;
    for (int device = 0; device < maxDevices; device++) {
      location.id = device;
      const unsigned int flags = 0;
      cudaMemPrefetchAsync(dataPtr, dataSize, location, flags, stream);
    }

    // The kernel only reads this data in the inner loop
    int innerLoopIter = 0;
    while (innerLoopIter < maxInnerLoopIter) {
      mykernel<<<32, threadsPerBlock, 0, stream>>>((const char *)dataPtr,
↪dataSize);
      innerLoopIter++;
    }
    outerLoopIter++;
  }

  free(dataPtr);
}
```

Managed

```
void test_advise_managed(cudaStream_t stream) {
  char *dataPtr;
  size_t dataSize = 64 * threadsPerBlock;  // 16 KiB

  // Allocate memory using cudaMallocManaged
  // (malloc may be used on systems with full CUDA Unified memory support)
  cudaMallocManaged(&dataPtr, dataSize);

  // Set the advice on the memory region
  cudaMemLocation loc = {.type = cudaMemLocationTypeDevice, .id = myGpuId};
  cudaMemAdvise(dataPtr, dataSize, cudaMemAdviseSetReadMostly, loc);

  int outerLoopIter = 0;
  while (outerLoopIter < maxOuterLoopIter) {
```

(continues on next page)

(continued from previous page)

```
    // The data is written by the CPU each outer loop iteration
    init_data(dataPtr, dataSize);

    // The data is made available to all GPUs by prefetching.
    // Prefetching here causes read duplication of data instead
    // of data migration
    cudaMemLocation location;
    location.type = cudaMemLocationTypeDevice;
    for (int device = 0; device < maxDevices; device++) {
      location.id = device;
      const unsigned int flags = 0;
      cudaMemPrefetchAsync(dataPtr, dataSize, location, flags, stream);
    }

    // The kernel only reads this data in the inner loop
    int innerLoopIter = 0;
    while (innerLoopIter < maxInnerLoopIter) {
      mykernel<<<32, threadsPerBlock, 0, stream>>>((const char *)dataPtr,
↪dataSize);
      innerLoopIter++;
    }
    outerLoopIter++;
  }

  cudaFree(dataPtr);
}
```

4.1.4.3 Querying Data Usage Attributes on Managed Memory

A program can query memory range attributes assigned through `cudaMemAdvise` or `cudaMem-PrefetchAsync` on CUDA managed memory by using the following API:

```
cudaMemRangeGetAttribute(void *data,
                         size_t dataSize,
                         enum cudaMemRangeAttribute attribute,
                         const void *devPtr,
                         size_t count);
```

This function queries an attribute of the memory range starting at `devPtr` with a size of `count` bytes. The memory range must refer to managed memory allocated via `cudaMallocManaged` or declared via `__managed__` variables. It is possible to query the following attributes:

- `cudaMemRangeAttributeReadMostly`: returns 1 if the entire memory range has the `cud-aMemAdviseSetReadMostly` attribute set, or 0 otherwise.
- `cudaMemRangeAttributePreferredLocation`: the result returned will be a GPU device id or `cudaCpuDeviceId` if the entire memory range has the corresponding processor as preferred location, otherwise `cudaInvalidDeviceId` will be returned. An application can use this query API to make decision about staging data through CPU or GPU depending on the preferred location attribute of the managed pointer. Note that the actual location of the memory range at the time of the query may be different from the preferred location.
- `cudaMemRangeAttributeAccessedBy`: will return the list of devices that have that advise set for that memory range.

- `cudaMemRangeAttributeLastPrefetchLocation`: will return the last location to which the memory range was prefetched explicitly using `cudaMemPrefetchAsync`. Note that this simply returns the last location that the application requested to prefetch the memory range to. It gives no indication as to whether the prefetch operation to that location has completed or even begun.
- `cudaMemRangeAttributePreferredLocationType`: it returns the location type of the preferred location with the following values:
  - `cudaMemLocationTypeDevice`: if all pages in the memory range have the same GPU as their preferred location,
  - `cudaMemLocationTypeHost`: if all pages in the memory range have the CPU as their preferred location,
  - `cudaMemLocationTypeHostNuma`: if all the pages in the memory range have the same host NUMA node ID as their preferred location,
  - `cudaMemLocationTypeInvalid`: if either all the pages don't have the same preferred location or some of the pages don't have a preferred location at all.
- `cudaMemRangeAttributePreferredLocationId`: returns the device ordinal if the `cudaMemRangeAttributePreferredLocationType` query for the same address range returns `cudaMemLocationTypeDevice`. If the preferred location type is a host NUMA node, it returns the host NUMA node ID. Otherwise, the id should be ignored.
- `cudaMemRangeAttributeLastPrefetchLocationType`: returns the last location type to which all pages in the memory range were prefetched explicitly via `cudaMemPrefetchAsync`. The following values are returned:
  - `cudaMemLocationTypeDevice`: if all pages in the memory range were prefetched to the same GPU,
  - `cudaMemLocationTypeHost`: if all pages in the memory range were prefetched to the CPU,
  - `cudaMemLocationTypeHostNuma`: if all the pages in the memory range were prefetched to the same host NUMA node ID,
  - `cudaMemLocationTypeInvalid`: if either all the pages were not prefetched to the same location or some of the pages were never prefetched at all.
- `cudaMemRangeAttributeLastPrefetchLocationId`: if the `cudaMemRangeAttributeLastPrefetchLocationType` query for the same address range returns `cudaMemLocationTypeDevice`, it will be a valid device ordinal or if it returns `cudaMemLocationTypeHostNuma`, it will be a valid host NUMA node ID. Otherwise, the id should be ignored.

Additionally, multiple attributes can be queried by using corresponding `cudaMemRangeGetAttributes` function.

4.1.4.4 GPU Memory Oversubscription

Unified memory enables applications to *oversubscribe* the memory of any individual processor: in other words they can allocate and share arrays larger than the memory capacity of any individual processor in the system, enabling among others out-of-core processing of datasets that do not fit within a single GPU, without adding significant complexity to the programming model.

Additionally, multiple attributes can be queried by using corresponding `cudaMemRangeGetAttributes` function.

4.2. CUDA Graphs

CUDA Graphs present another model for work submission in CUDA. A graph is a series of operations such as kernel launches, data movement, etc., connected by dependencies, which is defined separately from its execution. This allows a graph to be defined once and then launched repeatedly. Separating out the definition of a graph from its execution enables a number of optimizations: first, CPU launch costs are reduced compared to streams, because much of the setup is done in advance; second, presenting the whole workflow to CUDA enables optimizations which might not be possible with the piecewise work submission mechanism of streams.

To see the optimizations possible with graphs, consider what happens in a stream: when you place a kernel into a stream, the host driver performs a sequence of operations in preparation for the execution of the kernel on the GPU. These operations, necessary for setting up and launching the kernel, are an overhead cost which must be paid for each kernel that is issued. For a GPU kernel with a short execution time, this overhead cost can be a significant fraction of the overall end-to-end execution time. By creating a CUDA graph that encompasses a workflow that will be launched many times, these overhead costs can be paid once for the entire graph during instantiation, and the graph itself can then be launched repeatedly with very little overhead.

4.2.1. Graph Structure

An operation forms a node in a graph. The dependencies between the operations are the edges. These dependencies constrain the execution sequence of the operations.

An operation may be scheduled at any time once the nodes on which it depends are complete. Scheduling is left up to the CUDA system.

4.2.1.1 Node Types

A graph node can be one of:

- kernel
- CPU function call
- memory copy
- memset
- empty node
- waiting on a *CUDA Event*
- recording a *CUDA Event*
- signalling an *external semaphore*
- waiting on an *external semaphore*
- *conditional node*
- *memory node*
- child graph: To execute a separate nested graph, as shown in the following figure.

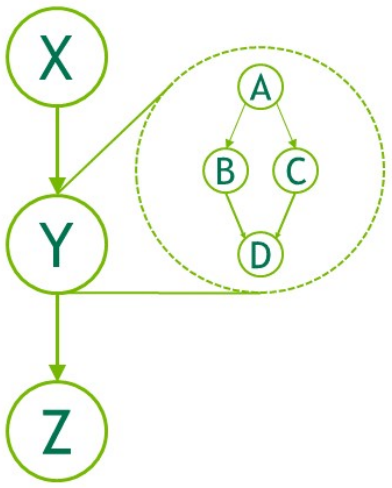


Figure 21: Child Graph Example

4.2.1.2 Edge Data

CUDA 12.3 introduced edge data on CUDA Graphs. At this time, the only use for non-default edge data is enabling *Programmatic Dependent Launch*.

Generally speaking, edge data modifies a dependency specified by an edge and consists of three parts: an outgoing port, an incoming port, and a type. An outgoing port specifies when an associated edge is triggered. An incoming port specifies what portion of a node is dependent on an associated edge. A type modifies the relation between the endpoints.

Port values are specific to node type and direction, and edge types may be restricted to specific node types. In all cases, zero-initialized edge data represents default behavior. Outgoing port 0 waits on an entire task, incoming port 0 blocks an entire task, and edge type 0 is associated with a full dependency with memory synchronizing behavior.

Edge data is optionally specified in various graph APIs via a parallel array to the associated nodes. If it is omitted as an input parameter, zero-initialized data is used. If it is omitted as an output (query) parameter, the API accepts this if the edge data being ignored is all zero-initialized, and returns `cudaErrorLossyQuery` if the call would discard information.

Edge data is also available in some stream capture APIs: `cudaStreamBeginCaptureToGraph()`, `cudaStreamGetCaptureInfo()`, and `cudaStreamUpdateCaptureDependencies()`. In these cases, there is not yet a downstream node. The data is associated with a dangling edge (half edge) which will either be connected to a future captured node or discarded at termination of stream capture. Note that some edge types do not wait on full completion of the upstream node. These edges are ignored when considering if a stream capture has been fully rejoined to the origin stream, and cannot be discarded at the end of capture. See *Stream Capture*.

No node types define additional incoming ports, and only kernel nodes define additional outgoing ports. There is one non-default dependency type, `cudaGraphDependencyTypeProgrammatic`, which is used to enable *Programmatic Dependent Launch* between two kernel nodes.

4.2.2. Building and Running Graphs

Work submission using graphs is separated into three distinct stages: definition, instantiation, and execution.

- During the **definition** or **creation** phase, a program creates a description of the operations in the graph along with the dependencies between them.
- **Instantiation** takes a snapshot of the graph template, validates it, and performs much of the setup and initialization of work with the aim of minimizing what needs to be done at launch. The resulting instance is known as an *executable graph.*
- An **executable** graph may be launched into a stream, similar to any other CUDA work. It may be launched any number of times without repeating the instantiation.

4.2.2.1 Graph Creation

Graphs can be created via two mechanisms: using the explicit Graph API and via stream capture.

4.2.2.1.1 Graph APIs

The following is an example (omitting declarations and other boilerplate code) of creating the below graph. Note the use of `cudaGraphCreate()` to create the graph and `cudaGraphAddNode()` to add the kernel nodes and their dependencies. The CUDA Runtime API documentation lists all the functions available for adding nodes and dependencies.

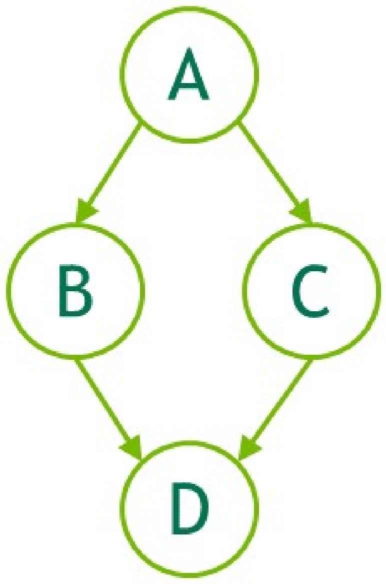


Figure 22: Creating a Graph Using Graph APIs Example

```
// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

// Create the nodes and their dependencies
cudaGraphNode_t nodes[4];
cudaGraphNodeParams kParams = { cudaGraphNodeTypeKernel };
```

(continues on next page)

(continued from previous page)

```
kParams.kernel.func         = (void *)kernelName;
kParams.kernel.gridDim.x    = kParams.kernel.gridDim.y  = kParams.kernel.
↪gridDim.z  = 1;
kParams.kernel.blockDim.x   = kParams.kernel.blockDim.y = kParams.kernel.
↪blockDim.z = 1;

cudaGraphAddNode(&nodes[0], graph, NULL, NULL, 0, &kParams);
cudaGraphAddNode(&nodes[1], graph, &nodes[0], NULL, 1, &kParams);
cudaGraphAddNode(&nodes[2], graph, &nodes[0], NULL, 1, &kParams);
cudaGraphAddNode(&nodes[3], graph, &nodes[1], NULL, 2, &kParams);
```

The example above shows four kernel nodes with dependencies between them to illustrate the creation of a very simple graph. In a typical user application there would also need to be nodes added for memory operations, such as `cudaGraphAddMemcpyNode()` and the like. For full reference of all graph API functions to add nodes, see the The CUDA Runtime API documentation .

4.2.2.1.2 Stream Capture

Stream capture provides a mechanism to create a graph from existing stream-based APIs. A section of code which launches work into streams, including existing code, can be bracketed with calls to `cudaStreamBeginCapture()` and `cudaStreamEndCapture()`. See below.

```
cudaGraph_t graph;

cudaStreamBeginCapture(stream);

kernel_A<<< ..., stream >>>(...);
kernel_B<<< ..., stream >>>(...);
libraryCall(stream);
kernel_C<<< ..., stream >>>(...);

cudaStreamEndCapture(stream, &graph);
```

A call to `cudaStreamBeginCapture()` places a stream in capture mode. When a stream is being captured, work launched into the stream is not enqueued for execution. It is instead appended to an internal graph that is progressively being built up. This graph is then returned by calling `cudaStreamEndCapture()`, which also ends capture mode for the stream. A graph which is actively being constructed by stream capture is referred to as a *capture graph.*

Stream capture can be used on any CUDA stream except `cudaStreamLegacy` (the “NULL stream”). Note that it *can* be used on `cudaStreamPerThread`. If a program is using the legacy stream, it may be possible to redefine stream 0 to be the per-thread stream with no functional change. See *Blocking and non-blocking streams and the default stream*.

Whether a stream is being captured can be queried with `cudaStreamIsCapturing()`.

Work can be captured to an existing graph using `cudaStreamBeginCaptureToGraph()`. Instead of capturing to an internal graph, work is captured to a graph provided by the user.

4.2.2.1.2.1 Cross-stream Dependencies and Events

Stream capture can handle cross-stream dependencies expressed with `cudaEventRecord()` and `cudaStreamWaitEvent()`, provided the event being waited upon was recorded into the same capture graph.

When an event is recorded in a stream that is in capture mode, it results in a *captured event*. A captured event represents a set of nodes in a capture graph.

When a captured event is waited on by a stream, it places the stream in capture mode if it is not already, and the next item in the stream will have additional dependencies on the nodes in the captured event. The two streams are then being captured to the same capture graph.

When cross-stream dependencies are present in stream capture, `cudaStreamEndCapture()` must still be called in the same stream where `cudaStreamBeginCapture()` was called; this is the *origin stream*. Any other streams which are being captured to the same capture graph, due to event-based dependencies, must also be joined back to the origin stream. This is illustrated below. All streams being captured to the same capture graph are taken out of capture mode upon `cudaStreamEndCapture()`. Failure to rejoin to the origin stream will result in failure of the overall capture operation.

```
// stream1 is the origin stream
cudaStreamBeginCapture(stream1);

kernel_A<<< ..., stream1 >>>(...);

// Fork into stream2
cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1);

kernel_B<<< ..., stream1 >>>(...);
kernel_C<<< ..., stream2 >>>(...);

// Join stream2 back to origin stream (stream1)
cudaEventRecord(event2, stream2);
cudaStreamWaitEvent(stream1, event2);

kernel_D<<< ..., stream1 >>>(...);

// End capture in the origin stream
cudaStreamEndCapture(stream1, &graph);

// stream1 and stream2 no longer in capture mode
```

The graph returned by the above code is shown in Figure 22.

Note

When a stream is taken out of capture mode, the next non-captured item in the stream (if any) will still have a dependency on the most recent prior non-captured item, despite intermediate items having been removed.

4.2.2.1.2.2 Prohibited and Unhandled Operations

It is invalid to synchronize or query the execution status of a stream which is being captured or a captured event, because they do not represent items scheduled for execution. It is also invalid to query the execution status of or synchronize a broader handle which encompasses an active stream capture, such as a device or context handle when any associated stream is in capture mode.

When any stream in the same context is being captured, and it was not created with `cudaStreamNonBlocking`, any attempted use of the legacy stream is invalid. This is because the legacy stream handle at all times encompasses these other streams; enqueueing to the legacy stream would create a dependency on the streams being captured, and querying it or synchronizing it would query or synchronize the streams being captured.

It is therefore also invalid to call synchronous APIs in this case. One example of a synchronous APIs is `cudaMemcpy()` which enqueues work to the legacy stream and synchronizes on it before returning.

Note

As a general rule, when a dependency relation would connect something that is captured with something that was not captured and instead enqueued for execution, CUDA prefers to return an error rather than ignore the dependency. An exception is made for placing a stream into or out of capture mode; this severs a dependency relation between items added to the stream immediately before and after the mode transition.

It is invalid to merge two separate capture graphs by waiting on a captured event from a stream which is being captured and is associated with a different capture graph than the event. It is invalid to wait on a non-captured event from a stream which is being captured without specifying the `cudaEventWaitExternal` flag.

A small number of APIs that enqueue asynchronous operations into streams are not currently supported in graphs and will return an error if called with a stream which is being captured, such as `cudaStreamAttachMemAsync()`.

4.2.2.1.2.3 Invalidation

When an invalid operation is attempted during stream capture, any associated capture graphs are *invalidated*. When a capture graph is invalidated, further use of any streams which are being captured or captured events associated with the graph is invalid and will return an error, until stream capture is ended with `cudaStreamEndCapture()`. This call will take the associated streams out of capture mode, but will also return an error value and a NULL graph.

4.2.2.1.2.4 Capture Introspection

Active stream capture operations can be inspected using `cudaStreamGetCaptureInfo()`. This allows the user to obtain the status of the capture, a unique(per-process) ID for the capture, the underlying graph object, and dependencies/edge data for the next node to be captured in the stream. This dependency information can be used to obtain a handle to the node(s) which were last captured in the stream.

4.2.2.1.3 Putting It All Together

The example in Figure 22 is a simplistic example intended to show a small graph conceptually. In an application that utilizes CUDA graphs, there is more complexity to using either the graph API or stream capture. The following code snippet shows a side by side comparison of the Graph API and Stream Capture to create a CUDA graph to execute a simple two stage reduction algorithm.

Figure 23 is an illustration of this CUDA graph and was generated using the `cudaGraphDebugDotPrint` function applied to the code below, with small adjustments for readability, and then rendered using Graphviz.

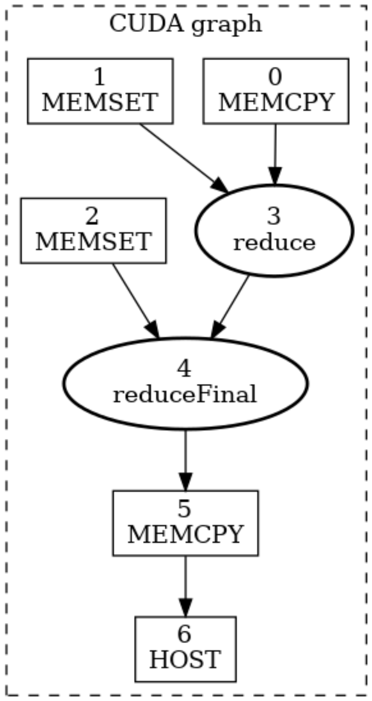


Figure 23: CUDA graph example using two stage reduction kernel

Graph API

```
void cudaGraphsManual(float  *inputVec_h,
                      float  *inputVec_d,
                      double *outputVec_d,
                      double *result_d,
                      size_t  inputSize,
                      size_t  numOfBlocks)
{
   cudaStream_t                 streamForGraph;
   cudaGraph_t                  graph;
   std::vector<cudaGraphNode_t> nodeDependencies;
   cudaGraphNode_t              memcpyNode, kernelNode, memsetNode;
   double                       result_h = 0.0;

   cudaStreamCreate(&streamForGraph));

   cudaKernelNodeParams kernelNodeParams = {0};
   cudaMemcpy3DParms    memcpyParams     = {0};
   cudaMemsetParams     memsetParams     = {0};
```

(continues on next page)

(continued from previous page)

```
    memcpyParams.srcArray = NULL;
    memcpyParams.srcPos   = make_cudaPos(0, 0, 0);
    memcpyParams.srcPtr   = make_cudaPitchedPtr(inputVec_h, sizeof(float) *
↪inputSize, inputSize, 1);
    memcpyParams.dstArray = NULL;
    memcpyParams.dstPos   = make_cudaPos(0, 0, 0);
    memcpyParams.dstPtr   = make_cudaPitchedPtr(inputVec_d, sizeof(float) *
↪inputSize, inputSize, 1);
    memcpyParams.extent   = make_cudaExtent(sizeof(float) * inputSize, 1, 1);
    memcpyParams.kind     = cudaMemcpyHostToDevice;

    memsetParams.dst         = (void *)outputVec_d;
    memsetParams.value       = 0;
    memsetParams.pitch       = 0;
    memsetParams.elementSize = sizeof(float); // elementSize can be max 4 bytes
    memsetParams.width       = numOfBlocks * 2;
    memsetParams.height      = 1;

    cudaGraphCreate(&graph, 0);
    cudaGraphAddMemcpyNode(&memcpyNode, graph, NULL, 0, &memcpyParams);
    cudaGraphAddMemsetNode(&memsetNode, graph, NULL, 0, &memsetParams);

    nodeDependencies.push_back(memsetNode);
    nodeDependencies.push_back(memcpyNode);

    void *kernelArgs[4] = {(void *)&inputVec_d, (void *)&outputVec_d, &
↪inputSize, &numOfBlocks};

    kernelNodeParams.func            = (void *)reduce;
    kernelNodeParams.gridDim         = dim3(numOfBlocks, 1, 1);
    kernelNodeParams.blockDim        = dim3(THREADS_PER_BLOCK, 1, 1);
    kernelNodeParams.sharedMemBytes = 0;
    kernelNodeParams.kernelParams    = (void **)kernelArgs;
    kernelNodeParams.extra           = NULL;

    cudaGraphAddKernelNode(
       &kernelNode, graph, nodeDependencies.data(), nodeDependencies.size(), &
↪kernelNodeParams);

    nodeDependencies.clear();
    nodeDependencies.push_back(kernelNode);

    memset(&memsetParams, 0, sizeof(memsetParams));
    memsetParams.dst         = result_d;
    memsetParams.value       = 0;
    memsetParams.elementSize = sizeof(float);
    memsetParams.width       = 2;
    memsetParams.height      = 1;
    cudaGraphAddMemsetNode(&memsetNode, graph, NULL, 0, &memsetParams);

    nodeDependencies.push_back(memsetNode);
```

(continues on next page)

(continued from previous page)

```
   memset(&kernelNodeParams, 0, sizeof(kernelNodeParams));
   kernelNodeParams.func             = (void *)reduceFinal;
   kernelNodeParams.gridDim          = dim3(1, 1, 1);
   kernelNodeParams.blockDim         = dim3(THREADS_PER_BLOCK, 1, 1);
   kernelNodeParams.sharedMemBytes = 0;
   void *kernelArgs2[3]              = {(void *)&outputVec_d, (void *)&result_d,
↪ &numOfBlocks};
   kernelNodeParams.kernelParams   = kernelArgs2;
   kernelNodeParams.extra            = NULL;

   cudaGraphAddKernelNode(
      &kernelNode, graph, nodeDependencies.data(), nodeDependencies.size(), &
↪kernelNodeParams);

   nodeDependencies.clear();
   nodeDependencies.push_back(kernelNode);

   memset(&memcpyParams, 0, sizeof(memcpyParams));

   memcpyParams.srcArray = NULL;
   memcpyParams.srcPos   = make_cudaPos(0, 0, 0);
   memcpyParams.srcPtr   = make_cudaPitchedPtr(result_d, sizeof(double), 1,
↪1);
   memcpyParams.dstArray = NULL;
   memcpyParams.dstPos   = make_cudaPos(0, 0, 0);
   memcpyParams.dstPtr   = make_cudaPitchedPtr(&result_h, sizeof(double), 1,
↪1);
   memcpyParams.extent   = make_cudaExtent(sizeof(double), 1, 1);
   memcpyParams.kind     = cudaMemcpyDeviceToHost;

   cudaGraphAddMemcpyNode(&memcpyNode, graph, nodeDependencies.data(),
↪nodeDependencies.size(), &memcpyParams);
   nodeDependencies.clear();
   nodeDependencies.push_back(memcpyNode);

   cudaGraphNode_t    hostNode;
   cudaHostNodeParams hostParams = {0};
   hostParams.fn                 = myHostNodeCallback;
   callBackData_t hostFnData;
   hostFnData.data      = &result_h;
   hostFnData.fn_name   = "cudaGraphsManual";
   hostParams.userData = &hostFnData;

   cudaGraphAddHostNode(&hostNode, graph, nodeDependencies.data(),
↪nodeDependencies.size(), &hostParams);
}
```

Stream Capture

```
void cudaGraphsUsingStreamCapture(float  *inputVec_h,
                                  float  *inputVec_d,
                                  double *outputVec_d,
                                  double *result_d,
                                  size_t  inputSize,
                                  size_t  numOfBlocks)
{
   cudaStream_t stream1, stream2, stream3, streamForGraph;
   cudaEvent_t  forkStreamEvent, memsetEvent1, memsetEvent2;
   cudaGraph_t  graph;
   double       result_h = 0.0;

   cudaStreamCreate(&stream1);
   cudaStreamCreate(&stream2);
   cudaStreamCreate(&stream3);
   cudaStreamCreate(&streamForGraph);

   cudaEventCreate(&forkStreamEvent);
   cudaEventCreate(&memsetEvent1);
   cudaEventCreate(&memsetEvent2);

   cudaStreamBeginCapture(stream1, cudaStreamCaptureModeGlobal);

   cudaEventRecord(forkStreamEvent, stream1);
   cudaStreamWaitEvent(stream2, forkStreamEvent, 0);
   cudaStreamWaitEvent(stream3, forkStreamEvent, 0);

   cudaMemcpyAsync(inputVec_d, inputVec_h, sizeof(float) * inputSize,
↪cudaMemcpyDefault, stream1);

   cudaMemsetAsync(outputVec_d, 0, sizeof(double) * numOfBlocks, stream2);

   cudaEventRecord(memsetEvent1, stream2);

   cudaMemsetAsync(result_d, 0, sizeof(double), stream3);
   cudaEventRecord(memsetEvent2, stream3);

   cudaStreamWaitEvent(stream1, memsetEvent1, 0);

   reduce<<<numOfBlocks, THREADS_PER_BLOCK, 0, stream1>>>(inputVec_d,
↪outputVec_d, inputSize, numOfBlocks);

   cudaStreamWaitEvent(stream1, memsetEvent2, 0);

   reduceFinal<<<1, THREADS_PER_BLOCK, 0, stream1>>>(outputVec_d, result_d,
↪numOfBlocks);
   cudaMemcpyAsync(&result_h, result_d, sizeof(double), cudaMemcpyDefault,
↪stream1);

   callBackData_t hostFnData = {0};
   hostFnData.data           = &result_h;
```

(continues on next page)

(continued from previous page)

```
    hostFnData.fn_name          = "cudaGraphsUsingStreamCapture";
    cudaHostFn_t fn             = myHostNodeCallback;
    cudaLaunchHostFunc(stream1, fn, &hostFnData);
    cudaStreamEndCapture(stream1, &graph);
}
```

4.2.2.2 Graph Instantiation

Once a graph has been created, either by the use of the graph API or stream capture, the graph must be instantiated to create an executable graph, which can then be launched. Assuming the `cudaGraph_t graph` has been created successfully, the following code will instantiate the graph and create the executable graph `cudaGraphExec_t graphExec`:

```
cudaGraphExec_t graphExec;
cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
```

4.2.2.3 Graph Execution

After a graph has been created and instantiated to create an executable graph, it can be launched. Assuming the `cudaGraphExec_t graphExec` has been created successfully, the following code snippet will launch the graph into the specified stream:

```
cudaGraphLaunch(graphExec, stream);
```

Pulling it all together and using the stream capture example from Section 4.2.2.1.2, the following code snippet will create a graph, instantiate it, and launch it:

```
cudaGraph_t graph;

cudaStreamBeginCapture(stream);

kernel_A<<< ..., stream >>>(...);
kernel_B<<< ..., stream >>>(...);
libraryCall(stream);
kernel_C<<< ..., stream >>>(...);

cudaStreamEndCapture(stream, &graph);

cudaGraphExec_t graphExec;
cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
cudaGraphLaunch(graphExec, stream);
```

4.2.3. Updating Instantiated Graphs

When a workflow changes, the graph becomes out of date and must be modified. Major changes to graph structure (such as topology or node types) require re-instantiation because topology-related optimizations must be re-applied. However, it is common for only node parameters (such as kernel parameters and memory addresses) to change while the graph topology remains the same. For this case, CUDA provides a lightweight “Graph Update” mechanism that allows certain node parameters to be modified in-place without rebuilding the entire graph, which is much more efficient than re-instantiation.

Updates take effect the next time the graph is launched, so they do not impact previous graph launches, even if they are running at the time of the update. A graph may be updated and relaunched repeatedly, so multiple updates/launches can be queued on a stream.

CUDA provides two mechanisms for updating instantiated graph parameters, whole graph update and individual node update. Whole graph update allows the user to supply a topologically identical `cudaGraph_t` object whose nodes contain updated parameters. Individual node update allows the user to explicitly update the parameters of individual nodes. Using an updated `cudaGraph_t` is more convenient when a large number of nodes are being updated, or when the graph topology is unknown to the caller (i.e., The graph resulted from stream capture of a library call). Using individual node update is preferred when the number of changes is small and the user has the handles to the nodes requiring updates. Individual node update skips the topology checks and comparisons for unchanged nodes, so it can be more efficient in many cases.

CUDA also provides a mechanism for enabling and disabling individual nodes without affecting their current parameters.

The following sections explain each approach in more detail.

4.2.3.1 Whole Graph Update

`cudaGraphExecUpdate()` allows an instantiated graph (the "original graph") to be updated with the parameters from a topologically identical graph (the "updating" graph). The topology of the updating graph must be identical to the original graph used to instantiate the `cudaGraphExec_t`. In addition, the order in which the dependencies are specified must match. Finally, CUDA needs to consistently order the sink nodes (nodes with no dependencies). CUDA relies on the order of specific api calls to achieve consistent sink node ordering.

More explicitly, following the following rules will cause `cudaGraphExecUpdate()` to pair the nodes in the original graph and the updating graph deterministically:

1. For any capturing stream, the API calls operating on that stream must be made in the same order, including event wait and other api calls not directly corresponding to node creation.
2. The API calls which directly manipulate a given graph node's incoming edges (including captured stream APIs, node add APIs, and edge addition / removal APIs) must be made in the same order. Moreover, when dependencies are specified in arrays to these APIs, the order in which the dependencies are specified inside those arrays must match.
3. Sink nodes must be consistently ordered. Sink nodes are nodes without dependent nodes / outgoing edges in the final graph at the time of the `cudaGraphExecUpdate()` invocation. The following operations affect sink node ordering (if present) and must (as a combined set) be made in the same order:
   - Node add APIs resulting in a sink node.
   - Edge removal resulting in a node becoming a sink node.
   - `cudaStreamUpdateCaptureDependencies()`, if it removes a sink node from a capturing stream's dependency set.
   - `cudaStreamEndCapture()`.

The following example shows how the API could be used to update an instantiated graph:

```
cudaGraphExec_t graphExec = NULL;

for (int i = 0; i < 10; i++) {
    cudaGraph_t graph;
```

(continues on next page)

(continued from previous page)

```
    cudaGraphExecUpdateResult updateResult;
    cudaGraphNode_t errorNode;

    // In this example we use stream capture to create the graph.
    // You can also use the Graph API to produce a graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);

    // Call a user-defined, stream based workload, for example
    do_cuda_work(stream);

    cudaStreamEndCapture(stream, &graph);

    // If we've already instantiated the graph, try to update it directly
    // and avoid the instantiation overhead
    if (graphExec != NULL) {
        // If the graph fails to update, errorNode will be set to the
        // node causing the failure and updateResult will be set to a
        // reason code.
        cudaGraphExecUpdate(graphExec, graph, &errorNode, &updateResult);
    }

    // Instantiate during the first iteration or whenever the update
    // fails for any reason
    if (graphExec == NULL || updateResult != cudaGraphExecUpdateSuccess) {

        // If a previous update failed, destroy the cudaGraphExec_t
        // before re-instantiating it
        if (graphExec != NULL) {
            cudaGraphExecDestroy(graphExec);
        }
        // Instantiate graphExec from graph. The error node and
        // error message parameters are unused here.
        cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    }

    cudaGraphDestroy(graph);
    cudaGraphLaunch(graphExec, stream);
    cudaStreamSynchronize(stream);
}
```

A typical workflow is to create the initial `cudaGraph_t` using either the stream capture or graph API. The `cudaGraph_t` is then instantiated and launched as normal. After the initial launch, a new `cudaGraph_t` is created using the same method as the initial graph and `cudaGraphExecUpdate()` is called. If the graph update is successful, indicated by the `updateResult` parameter in the above example, the updated `cudaGraphExec_t` is launched. If the update fails for any reason, the `cudaGraphExecDestroy()` and `cudaGraphInstantiate()` are called to destroy the original `cudaGraphExec_t` and instantiate a new one.

It is also possible to update the `cudaGraph_t` nodes directly (i.e., Using `cudaGraphKernelNodeSetParams()`) and subsequently update the `cudaGraphExec_t`, however it is more efficient to use the explicit node update APIs covered in the next section.

Conditional handle flags and default values are updated as part of the graph update.

Please see the Graph API for more information on usage and current limitations.

4.2.3.2 Individual Node Update

Instantiated graph node parameters can be updated directly. This eliminates the overhead of instantiation as well as the overhead of creating a new `cudaGraph_t`. If the number of nodes requiring update is small relative to the total number of nodes in the graph, it is better to update the nodes individually. The following methods are available for updating `cudaGraphExec_t` nodes:

Table 8: Individual Node Update APIs

API	Node Type
`cudaGraphExecKernelNodeSetParams()`	Kernel node
`cudaGraphExecMemcpyNodeSetParams()`	Memory copy node
`cudaGraphExecMemsetNodeSetParams()`	Memory set node
`cudaGraphExecHostNodeSetParams()`	Host node
`cudaGraphExecChildGraphNodeSetParams()`	Child graph node
`cudaGraphExecEventRecordNodeSetEvent()`	Event record node
`cudaGraphExecEventWaitNodeSetEvent()`	Event wait node
`cudaGraphExecExternalSemaphoresSignalNodeSetParams()`	External semaphore signal node
`cudaGraphExecExternalSemaphoresWaitNodeSetParams()`	External semaphore wait node

Please see the Graph API for more information on usage and current limitations.

4.2.3.3 Individual Node Enable

Kernel, memset and memcpy nodes in an instantiated graph can be enabled or disabled using the `cudaGraphNodeSetEnabled()` API. This allows the creation of a graph which contains a superset of the desired functionality which can be customized for each launch. The enable state of a node can be queried using the `cudaGraphNodeGetEnabled()` API.

A disabled node is functionally equivalent to empty node until it is re-enabled. Node parameters are not affected by enabling/disabling a node. Enable state is unaffected by individual node update or whole graph update with `cudaGraphExecUpdate()`. Parameter updates while the node is disabled will take effect when the node is re-enabled.

Refer to the Graph API for more information on usage and current limitations.

4.2.3.4 Graph Update Limitations

Kernel nodes:

- The owning context of the function cannot change.
- A node whose function originally did not use CUDA dynamic parallelism cannot be updated to a function which uses CUDA dynamic parallelism.

`cudaMemset` and `cudaMemcpy` nodes:

- The CUDA device(s) to which the operand(s) was allocated/mapped cannot change.

- The source/destination memory must be allocated from the same context as the original source/destination memory.
- Only 1D `cudaMemset`/`cudaMemcpy` nodes can be changed.

Additional memcpy node restrictions:

- Changing either the source or destination memory type (i.e., `cudaPitchedPtr`, `cudaArray_t`, etc.), or the type of transfer (i.e., `cudaMemcpyKind`) is not supported.

External semaphore wait nodes and record nodes:

- Changing the number of semaphores is not supported.

Conditional nodes:

- The order of handle creation and assignment must match between the graphs.
- Changing node parameters is not supported (i.e. number of graphs in the conditional, node context, etc).
- Changing parameters of nodes within the conditional body graph is subject to the rules above.

Memory nodes:

- It is not possible to update a `cudaGraphExec_t` with a `cudaGraph_t` if the `cudaGraph_t` is currently instantiated as a different `cudaGraphExec_t`.

There are no restrictions on updates to host nodes, event record nodes, or event wait nodes.

4.2.4. Conditional Graph Nodes

Conditional nodes allow conditional execution and looping of a graph contained within the conditional node. This allows dynamic and iterative workflows to be represented completely within a graph and frees up the host CPU to perform other work in parallel.

Evaluation of the condition value is performed on the device when the dependencies of the conditional node have been met. Conditional nodes can be one of the following types:

- Conditional *IF nodes* execute their body graph once if the condition value is non-zero when the node is executed. An optional second body graph can be provided and this will be executed once if the condition value is zero when the node is executed.
- Conditional *WHILE nodes* execute their body graph if the condition value is non-zero when the node is executed and will continue to execute their body graph until the condition value is zero.
- Conditional *SWITCH nodes* execute the zero-indexed nth body graph once if the condition value is equal to n. If the condition value does not correspond to a body graph, no body graph is launched.

A condition value is accessed by a *conditional handle* , which must be created before the node. The condition value can be set by device code using `cudaGraphSetConditional()`. A default value, applied on each graph launch, can also be specified when the handle is created.

When the conditional node is created, an empty graph is created and the handle is returned to the user so that the graph can be populated. This conditional body graph can be populated using either the *graph APIs* or *cudaStreamBeginCaptureToGraph()*.

Conditional nodes can be nested.

4.2.4.1 Conditional Handles

A condition value is represented by `cudaGraphConditionalHandle` and is created by `cudaGraphConditionalHandleCreate()`.

The handle must be associated with a single conditional node. Handles cannot be destroyed and as such there is no need to keep track of them.

If `cudaGraphCondAssignDefault` is specified when the handle is created, the condition value will be initialized to the specified default at the beginning of each graph execution. If this flag is not provided, the condition value is undefined at the start of each graph execution and code should not assume that the condition value persists across executions.

The default value and flags associated with a handle will be updated during *whole graph update*.

4.2.4.2 Conditional Node Body Graph Requirements

General requirements:

- The graph's nodes must all reside on a single device.
- The graph can only contain kernel nodes, empty nodes, memcpy nodes, memset nodes, child graph nodes, and conditional nodes.

Kernel nodes:

- Use of CUDA Dynamic Parallelism or Device Graph Launch by kernels in the graph is not permitted.
- Cooperative launches are permitted so long as MPS is not in use.

Memcpy/Memset nodes:

- Only copies/memsets involving device memory and/or pinned device-mapped host memory are permitted.
- Copies/memsets involving CUDA arrays are not permitted.
- Both operands must be accessible from the current device at time of instantiation. Note that the copy operation will be performed from the device on which the graph resides, even if it is targeting memory on another device.

4.2.4.3 Conditional IF Nodes

The body graph of an IF node will be executed once if the condition is non-zero when the node is executed. The following diagram depicts a 3 node graph where the middle node, B, is a conditional node:

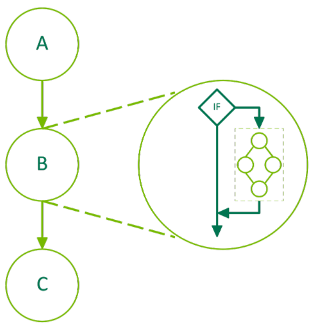


Figure 24: Conditional IF Node

The following code illustrates the creation of a graph containing an IF conditional node. The default value of the condition is set using an upstream kernel. The body of the conditional is populated using the *graph API*.

```
__global__ void setHandle(cudaGraphConditionalHandle handle, int value)
{
    ...
    // Set the condition value to the value passed to the kernel
    cudaGraphSetConditional(handle, value);
    ...
}

void graphSetup() {
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
    cudaGraphNode_t node;
    void *kernelArgs[2];
    int value = 1;

    // Create the graph
    cudaGraphCreate(&graph, 0);

    // Create the conditional handle; because no default value is provided,
↪the condition value is undefined at the start of each graph execution
    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.
↪z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.
↪blockDim.z = 1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    kernelArgs[1] = &value;
    cudaGraphAddNode(&node, graph, NULL, 0, &params);

    // Create and add the conditional node
    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeIf;
    cParams.conditional.size   = 1; // There is only an "if" body graph
    cudaGraphAddNode(&node, graph, &node, 1, &cParams);

    // Get the body graph of the conditional node
    cudaGraph_t bodyGraph = cParams.conditional.phGraph_out[0];

    // Populate the body graph of the IF conditional node
    ...
    cudaGraphAddNode(&node, bodyGraph, NULL, 0, &params);
```

(continues on next page)

(continued from previous page)

```
    // Instantiate and launch the graph
    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();

    // Clean up
    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
}
```

IF nodes can also have an optional second body graph which is executed once when the node is executed if the condition value is zero.

```
void graphSetup() {
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
    cudaGraphNode_t node;
    void *kernelArgs[2];
    int value = 1;

    // Create the graph
    cudaGraphCreate(&graph, 0);

    // Create the conditional handle; because no default value is provided,
↪the condition value is undefined at the start of each graph execution
    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.
↪z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.
↪blockDim.z = 1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    kernelArgs[1] = &value;
    cudaGraphAddNode(&node, graph, NULL, 0, &params);

    // Create and add the IF conditional node
    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeIf;
    cParams.conditional.size   = 2; // There is both an "if" and an "else"
↪body graph
    cudaGraphAddNode(&node, graph, &node, 1, &cParams);

    // Get the body graphs of the conditional node
    cudaGraph_t ifBodyGraph = cParams.conditional.phGraph_out[0];
    cudaGraph_t elseBodyGraph = cParams.conditional.phGraph_out[1];
```

(continues on next page)

(continued from previous page)

```
    // Populate the body graphs of the IF conditional node
    ...
    cudaGraphAddNode(&node, ifBodyGraph, NULL, 0, &params);
    ...
    cudaGraphAddNode(&node, elseBodyGraph, NULL, 0, &params);

    // Instantiate and launch the graph
    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();

    // Clean up
    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
}
```

4.2.4.4 Conditional WHILE Nodes

The body graph of a WHILE node will be executed as long as the condition is non-zero. The condition will be evaluated when the node is executed and after completion of the body graph. The following diagram depicts a 3 node graph where the middle node, B, is a conditional node:

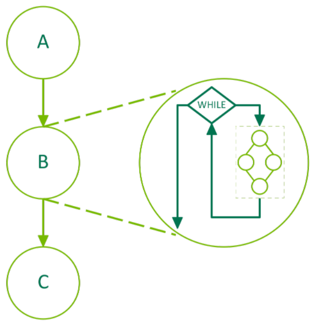


Figure 25: Conditional WHILE Node

The following code illustrates the creation of a graph containing a WHILE conditional node. The handle is created using *cudaGraphCondAssignDefault* to avoid the need for an upstream kernel. The body of the conditional is populated using the *graph API*.

```
__global__ void loopKernel(cudaGraphConditionalHandle handle, char *dPtr)
{
   // Decrement the value of dPtr and set the condition value to 0 once dPtr
↪is 0
   if (--(*dPtr) == 0) {
      cudaGraphSetConditional(handle, 0);
   }
}

void graphSetup() {
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
```

(continues on next page)

(continued from previous page)

```
    cudaGraphNode_t node;
    void *kernelArgs[2];

    // Allocate a byte of device memory to use as input
    char *dPtr;
    cudaMalloc((void **)&dPtr, 1);

    // Create the graph
    cudaGraphCreate(&graph, 0);

    // Create the conditional handle with a default value of 1
    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph, 1,
↪cudaGraphCondAssignDefault);

    // Create and add the WHILE conditional node
    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeWhile;
    cParams.conditional.size   = 1;
    cudaGraphAddNode(&node, graph, NULL, 0, &cParams);

    // Get the body graph of the conditional node
    cudaGraph_t bodyGraph = cParams.conditional.phGraph_out[0];

    // Populate the body graph of the conditional node
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)loopKernel;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.
↪z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.
↪blockDim.z = 1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    kernelArgs[1] = &dPtr;
    cudaGraphAddNode(&node, bodyGraph, NULL, 0, &params);

    // Initialize device memory, instantiate, and launch the graph
    cudaMemset(dPtr, 10, 1); // Set dPtr to 10; the loop will run until dPtr
↪is 0
    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();

    // Clean up
    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
    cudaFree(dPtr);
}
```

4.2.4.5 Conditional SWITCH Nodes

The zero-indexed nth body graph of a SWITCH node will be executed once if the condition is equal to n when the node is executed. The following diagram depicts a 3 node graph where the middle node, B, is a conditional node:

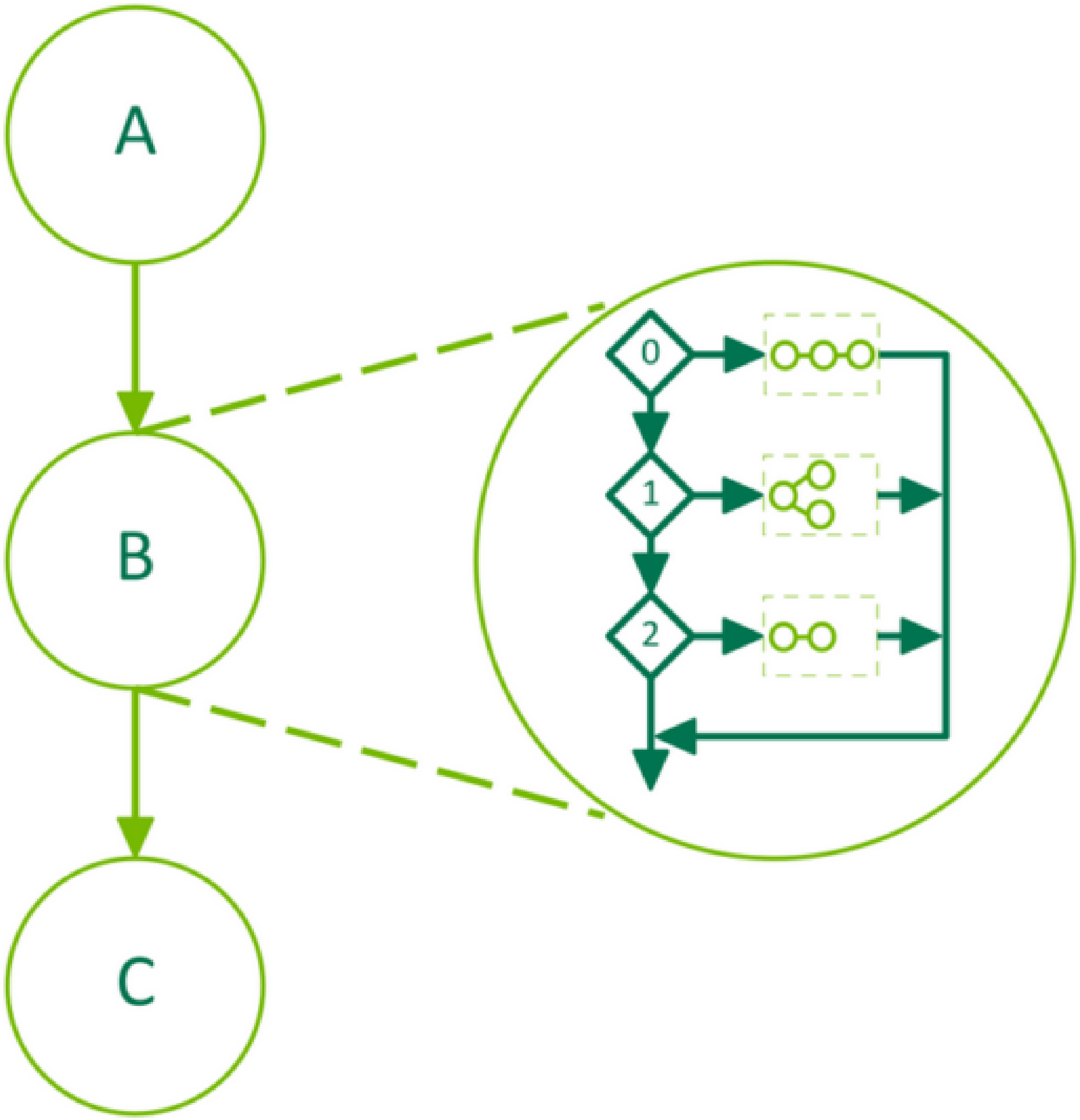


Figure 26: Conditional SWITCH Node

The following code illustrates the creation of a graph containing a SWITCH conditional node. The value of the condition is set using an upstream kernel. The bodies of the conditional are populated using the *graph API*.

```
__global__ void setHandle(cudaGraphConditionalHandle handle, int value)
{
    ...
    // Set the condition value to the value passed to the kernel
```

(continues on next page)

(continued from previous page)

```
    cudaGraphSetConditional(handle, value);
    ...
}

void graphSetup() {
    cudaGraph_t graph;
    cudaGraphExec_t graphExec;
    cudaGraphNode_t node;
    void *kernelArgs[2];
    int value = 1;

    // Create the graph
    cudaGraphCreate(&graph, 0);

    // Create the conditional handle; because no default value is provided,
↪the condition value is undefined at the start of each graph execution
    cudaGraphConditionalHandle handle;
    cudaGraphConditionalHandleCreate(&handle, graph);

    // Use a kernel upstream of the conditional to set the handle value
    cudaGraphNodeParams params = { cudaGraphNodeTypeKernel };
    params.kernel.func = (void *)setHandle;
    params.kernel.gridDim.x = params.kernel.gridDim.y = params.kernel.gridDim.
↪z = 1;
    params.kernel.blockDim.x = params.kernel.blockDim.y = params.kernel.
↪blockDim.z = 1;
    params.kernel.kernelParams = kernelArgs;
    kernelArgs[0] = &handle;
    kernelArgs[1] = &value;
    cudaGraphAddNode(&node, graph, NULL, 0, &params);

    // Create and add the conditional SWITCH node
    cudaGraphNodeParams cParams = { cudaGraphNodeTypeConditional };
    cParams.conditional.handle = handle;
    cParams.conditional.type   = cudaGraphCondTypeSwitch;
    cParams.conditional.size   = 5;
    cudaGraphAddNode(&node, graph, &node, 1, &cParams);

    // Get the body graphs of the conditional node
    cudaGraph_t *bodyGraphs = cParams.conditional.phGraph_out;

    // Populate the body graphs of the SWITCH conditional node
    ...
    cudaGraphAddNode(&node, bodyGraphs[0], NULL, 0, &params);
    ...
    cudaGraphAddNode(&node, bodyGraphs[4], NULL, 0, &params);

    // Instantiate and launch the graph
    cudaGraphInstantiate(&graphExec, graph, NULL, NULL, 0);
    cudaGraphLaunch(graphExec, 0);
    cudaDeviceSynchronize();
```

(continues on next page)

(continued from previous page)

```
    // Clean up
    cudaGraphExecDestroy(graphExec);
    cudaGraphDestroy(graph);
}
```

4.2.5. Graph Memory Nodes

4.2.5.1 Introduction

Graph memory nodes allow graphs to create and own memory allocations. Graph memory nodes have GPU ordered lifetime semantics, which dictate when memory is allowed to be accessed on the device. These GPU ordered lifetime semantics enable driver-managed memory reuse, and match those of the stream ordered allocation APIs `cudaMallocAsync` and `cudaFreeAsync`, which may be captured when creating a graph.

Graph allocations have fixed addresses over the life of a graph including repeated instantiations and launches. This allows the memory to be directly referenced by other operations within the graph without the need of a graph update, even when CUDA changes the backing physical memory. Within a graph, allocations whose graph ordered lifetimes do not overlap may use the same underlying physical memory.

CUDA may reuse the same physical memory for allocations across multiple graphs, aliasing virtual address mappings according to the GPU ordered lifetime semantics. For example when different graphs are launched into the same stream, CUDA may virtually alias the same physical memory to satisfy the needs of allocations which have single-graph lifetimes.

4.2.5.2 API Fundamentals

Graph memory nodes are graph nodes representing either memory allocation or free actions. As a shorthand, nodes that allocate memory are called allocation nodes. Likewise, nodes that free memory are called free nodes. Allocations created by allocation nodes are called graph allocations. CUDA assigns virtual addresses for the graph allocation at node creation time. While these virtual addresses are fixed for the lifetime of the allocation node, the allocation contents are not persistent past the freeing operation and may be overwritten by accesses referring to a different allocation.

Graph allocations are considered recreated every time a graph runs. A graph allocation's lifetime, which differs from the node's lifetime, begins when GPU execution reaches the allocating graph node and ends when one of the following occurs:

- GPU execution reaches the freeing graph node
- GPU execution reaches the freeing `cudaFreeAsync()` stream call
- immediately upon the freeing call to `cudaFree()`

Note

Graph destruction does not automatically free any live graph-allocated memory, even though it ends the lifetime of the allocation node. The allocation must subsequently be freed in another graph, or using `cudaFreeAsync()`/`cudaFree()`.

Just like other *graph-structure*, graph memory nodes are ordered within a graph by dependency edges. A program must guarantee that operations accessing graph memory:

- are ordered after the allocation node
- are ordered before the operation freeing the memory

Graph allocation lifetimes begin and usually end according to GPU execution (as opposed to API invocation). GPU ordering is the order that work runs on the GPU as opposed to the order that the work is enqueued or described. Thus, graph allocations are considered 'GPU ordered.'

4.2.5.2.1 Graph Node APIs

Graph memory nodes may be explicitly created with the node creation API, `cudaGraphAddNode`. The address allocated when adding a cudaGraphNodeTypeMemAlloc node is returned to the user in the `alloc::dptr` field of the passed `cudaGraphNodeParams` structure. All operations using graph allocations inside the allocating graph must be ordered after the allocating node. Similarly, any free nodes must be ordered after all uses of the allocation within the graph. Free nodes are created using `cudaGraphAddNode` and a node type of cudaGraphNodeTypeMemFree.

In the following figure, there is an example graph with an alloc and a free node. Kernel nodes **a**, **b**, and **c** are ordered after the allocation node and before the free node such that the kernels can access the allocation. Kernel node **e** is not ordered after the alloc node and therefore cannot safely access the memory. Kernel node **d** is not ordered before the free node, therefore it cannot safely access the memory.

The following code snippet establishes the graph in this figure:

```
// Create the graph - it starts out empty
cudaGraphCreate(&graph, 0);

// parameters for a basic allocation
cudaGraphNodeParams params = { cudaGraphNodeTypeMemAlloc };
params.alloc.poolProps.allocType = cudaMemAllocationTypePinned;
params.alloc.poolProps.location.type = cudaMemLocationTypeDevice;
// specify device 0 as the resident device
params.alloc.poolProps.location.id = 0;
params.alloc.bytesize = size;

cudaGraphAddNode(&allocNode, graph, NULL, NULL, 0, &params);

// create a kernel node that uses the graph allocation
cudaGraphNodeParams nodeParams = { cudaGraphNodeTypeKernel };
nodeParams.kernel.kernelParams[0] = params.alloc.dptr;
// ...set other kernel node parameters...

// add the kernel node to the graph
cudaGraphAddNode(&a, graph, &allocNode, 1, NULL, &nodeParams);
cudaGraphAddNode(&b, graph, &a, 1, NULL, &nodeParams);
cudaGraphAddNode(&c, graph, &a, 1, NULL, &nodeParams);
cudaGraphNode_t dependencies[2];
// kernel nodes b and c are using the graph allocation, so the freeing node
↪must depend on them.  Since the dependency of node b on node a establishes
↪an indirect dependency, the free node does not need to explicitly depend on
↪node a.
dependencies[0] = b;
dependencies[1] = c;
cudaGraphNodeParams freeNodeParams = { cudaGraphNodeTypeMemFree };
```

(continues on next page)

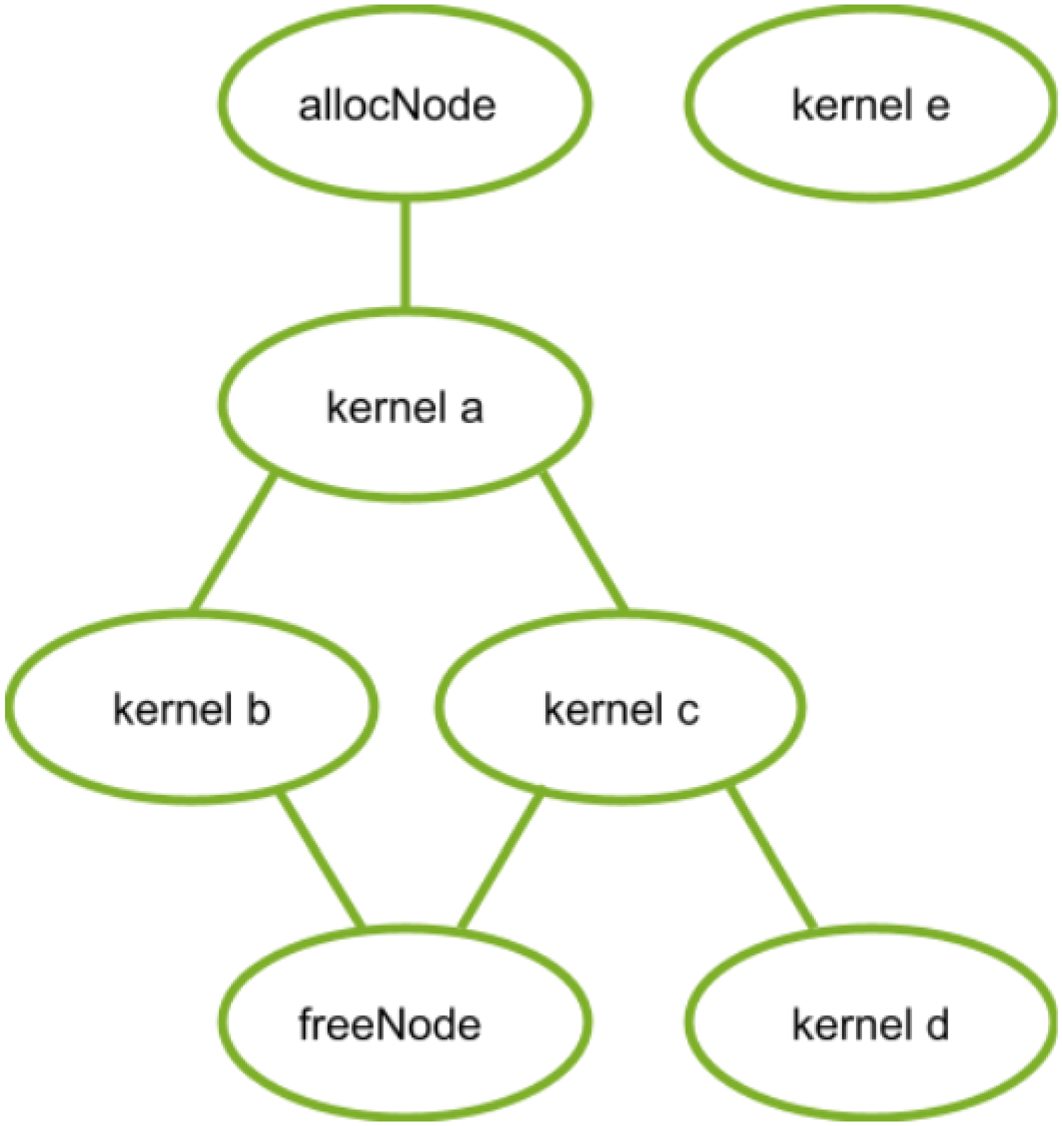


Figure 27: Kernel Nodes

(continued from previous page)

```
freeNodeParams.free.dptr = params.alloc.dptr;
cudaGraphAddNode(&freeNode, graph, dependencies, NULL, 2, freeNodeParams);
// free node does not depend on kernel node d, so it must not access the freed
↪graph allocation.
cudaGraphAddNode(&d, graph, &c, NULL, 1, &nodeParams);

// node e does not depend on the allocation node, so it must not access the
↪allocation.  This would be true even if the freeNode depended on kernel node
↪e.
cudaGraphAddNode(&e, graph, NULL, NULL, 0, &nodeParams);
```

4.2.5.2.2 Stream Capture

Graph memory nodes can be created by capturing the corresponding stream ordered allocation and free calls cudaMallocAsync and cudaFreeAsync. In this case, the virtual addresses returned by the captured allocation API can be used by other operations inside the graph. Since the stream ordered dependencies will be captured into the graph, the ordering requirements of the stream ordered allocation APIs guarantee that the graph memory nodes will be properly ordered with respect to the captured stream operations (for correctly written stream code).

Ignoring kernel nodes **d** and **e**, for clarity, the following code snippet shows how to use stream capture to create the graph from the previous figure:

```
cudaMallocAsync(&dptr, size, stream1);
kernel_A<<< ..., stream1 >>>(dptr, ...);

// Fork into stream2
cudaEventRecord(event1, stream1);
cudaStreamWaitEvent(stream2, event1);

kernel_B<<< ..., stream1 >>>(dptr, ...);
// event dependencies translated into graph dependencies, so the kernel node
↪created by the capture of kernel C will depend on the allocation node
↪created by capturing the cudaMallocAsync call.
kernel_C<<< ..., stream2 >>>(dptr, ...);

// Join stream2 back to origin stream (stream1)
cudaEventRecord(event2, stream2);
cudaStreamWaitEvent(stream1, event2);

// Free depends on all work accessing the memory.
cudaFreeAsync(dptr, stream1);

// End capture in the origin stream
cudaStreamEndCapture(stream1, &graph);
```

4.2.5.2.3 Accessing and Freeing Graph Memory Outside of the Allocating Graph

Graph allocations do not have to be freed by the allocating graph. When a graph does not free an allocation, that allocation persists beyond the execution of the graph and can be accessed by subsequent CUDA operations. These allocations may be accessed in another graph or directly using a stream operation as long as the accessing operation is ordered after the allocation through CUDA events and other stream ordering mechanisms. An allocation may subsequently be freed by regular calls to cudaFree, cudaFreeAsync, or by the launch of another graph with a corresponding free node, or a subsequent launch of the allocating graph (if it was instantiated with the *graph-memory-nodes-cudagraphinstantiateflagautofreeonlaunch* flag). It is illegal to access memory after it has been freed - the free operation must be ordered after all operations accessing the memory using graph dependencies, CUDA events, and other stream ordering mechanisms.

Note

Since graph allocations may share underlying physical memory, free operations must be ordered after all device operations complete. Out-of-band synchronization (such as memory-based synchronization within a compute kernel) is insufficient for ordering between memory writes and free operations. For more information, see the *Virtual Aliasing Support* rules relating to consistency and coherency.

The three following code snippets demonstrate accessing graph allocations outside of the allocating graph with ordering properly established by: using a single stream, using events between streams, and using events baked into the allocating and freeing graph.

First, ordering established by using a single stream:

```
// Contents of allocating graph
void *dptr;
cudaGraphNodeParams params = { cudaGraphNodeTypeMemAlloc };
params.alloc.poolProps.allocType = cudaMemAllocationTypePinned;
params.alloc.poolProps.location.type = cudaMemLocationTypeDevice;
params.alloc.bytesize = size;
cudaGraphAddNode(&allocNode, allocGraph, NULL, NULL, 0, &params);
dptr = params.alloc.dptr;

cudaGraphInstantiate(&allocGraphExec, allocGraph, NULL, NULL, 0);

cudaGraphLaunch(allocGraphExec, stream);
kernel<<< ..., stream >>>(dptr, ...);
cudaFreeAsync(dptr, stream);
```

Second, ordering established by recording and waiting on CUDA events:

```
// Contents of allocating graph
void *dptr;

// Contents of allocating graph
cudaGraphAddNode(&allocNode, allocGraph, NULL, NULL, 0, &allocNodeParams);
dptr = allocNodeParams.alloc.dptr;

// contents of consuming/freeing graph
kernelNodeParams.kernel.kernelParams[0] = allocNodeParams.alloc.dptr;
cudaGraphAddNode(&freeNode, freeGraph, NULL, NULL, 1, dptr);

cudaGraphInstantiate(&allocGraphExec, allocGraph, NULL, NULL, 0);
cudaGraphInstantiate(&freeGraphExec, freeGraph, NULL, NULL, 0);

cudaGraphLaunch(allocGraphExec, allocStream);
```

(continues on next page)

(continued from previous page)

```
// establish the dependency of stream2 on the allocation node
// note: the dependency could also have been established with a stream
↪synchronize operation
cudaEventRecord(allocEvent, allocStream);
cudaStreamWaitEvent(stream2, allocEvent);

kernel<<< ..., stream2 >>> (dptr, ...);

// establish the dependency between the stream 3 and the allocation use
cudaStreamRecordEvent(streamUseDoneEvent, stream2);
cudaStreamWaitEvent(stream3, streamUseDoneEvent);

// it is now safe to launch the freeing graph, which may also access the memory
cudaGraphLaunch(freeGraphExec, stream3);
```

Third, ordering established by using graph external event nodes:

```
// Contents of allocating graph
void *dptr;
cudaEvent_t allocEvent; // event indicating when the allocation will be ready
↪for use.
cudaEvent_t streamUseDoneEvent; // event indicating when the stream operations
↪are done with the allocation.

// Contents of allocating graph with event record node
cudaGraphAddNode(&allocNode, allocGraph, NULL, NULL, 0, &allocNodeParams);
dptr = allocNodeParams.alloc.dptr;
// note: this event record node depends on the alloc node

cudaGraphNodeParams allocEventNodeParams = { cudaGraphNodeTypeEventRecord };
allocEventParams.eventRecord.event = allocEvent;
cudaGraphAddNode(&recordNode, allocGraph, &allocNode, NULL, 1,
↪allocEventNodeParams);
cudaGraphInstantiate(&allocGraphExec, allocGraph, NULL, NULL, 0);

// contents of consuming/freeing graph with event wait nodes
cudaGraphNodeParams streamWaitEventNodeParams = { cudaGraphNodeTypeEventWait }
↪;
streamWaitEventNodeParams.eventWait.event = streamUseDoneEvent;
cudaGraphAddNode(&streamUseDoneEventNode, waitAndFreeGraph, NULL, NULL, 0,
↪streamWaitEventNodeParams);

cudaGraphNodeParams allocWaitEventNodeParams = { cudaGraphNodeTypeEventWait };
allocWaitEventNodeParams.eventWait.event = allocEvent;
cudaGraphAddNode(&allocReadyEventNode, waitAndFreeGraph, NULL, NULL, 0,
↪allocWaitEventNodeParams);

kernelNodeParams->kernelParams[0] = allocNodeParams.alloc.dptr;

// The allocReadyEventNode provides ordering with the alloc node for use in a
↪consuming graph.
```

(continues on next page)

(continued from previous page)

```
cudaGraphAddNode(&kernelNode, waitAndFreeGraph, &allocReadyEventNode, NULL, 1,
↪ &kernelNodeParams);

// The free node has to be ordered after both external and internal users.
// Thus the node must depend on both the kernelNode and the
↪streamUseDoneEventNode.
dependencies[0] = kernelNode;
dependencies[1] = streamUseDoneEventNode;

cudaGraphNodeParams freeNodeParams = { cudaGraphNodeTypeMemFree };
freeNodeParams.free.dptr = dptr;
cudaGraphAddNode(&freeNode, waitAndFreeGraph, &dependencies, NULL, 2,
↪freeNodeParams);
cudaGraphInstantiate(&waitAndFreeGraphExec, waitAndFreeGraph, NULL, NULL, 0);

cudaGraphLaunch(allocGraphExec, allocStream);

// establish the dependency of stream2 on the event node satisfies the ordering
↪requirement
cudaStreamWaitEvent(stream2, allocEvent);
kernel<<< ..., stream2 >>> (dptr, ...);
cudaStreamRecordEvent(streamUseDoneEvent, stream2);

// the event wait node in the waitAndFreeGraphExec establishes the dependency
↪on the "readyForFreeEvent" that is needed to prevent the kernel running in
↪stream two from accessing the allocation after the free node in execution
↪order.
cudaGraphLaunch(waitAndFreeGraphExec, stream3);
```

4.2.5.2.4 cudaGraphInstantiateFlagAutoFreeOnLaunch

Under normal circumstances, CUDA will prevent a graph from being relaunched if it has unfreed memory allocations because multiple allocations at the same address will leak memory. Instantiating a graph with the `cudaGraphInstantiateFlagAutoFreeOnLaunch` flag allows the graph to be relaunched while it still has unfreed allocations. In this case, the launch automatically inserts an asynchronous free of the unfreed allocations.

Auto free on launch is useful for single-producer multiple-consumer algorithms. At each iteration, a producer graph creates several allocations, and, depending on runtime conditions, a varying set of consumers accesses those allocations. This type of variable execution sequence means that consumers cannot free the allocations because a subsequent consumer may require access. Auto free on launch means that the launch loop does not need to track the producer's allocations - instead, that information remains isolated to the producer's creation and destruction logic. In general, auto free on launch simplifies an algorithm which would otherwise need to free all the allocations owned by a graph before each relaunch.

Note

The `cudaGraphInstantiateFlagAutoFreeOnLaunch` flag does not change the behavior of graph destruction. The application must explicitly free the unfreed memory in order to avoid memory leaks, even for graphs instantiated with the flag. The following code shows the use of `cudaGraphInstantiateFlagAutoFreeOnLaunch` to simplify a single-producer / multiple-consumer algorithm:

```
// Create producer graph which allocates memory and populates it with data
cudaStreamBeginCapture(cudaStreamPerThread, cudaStreamCaptureModeGlobal);
cudaMallocAsync(&data1, blocks * threads, cudaStreamPerThread);
cudaMallocAsync(&data2, blocks * threads, cudaStreamPerThread);
produce<<<blocks, threads, 0, cudaStreamPerThread>>>(data1, data2);
...
cudaStreamEndCapture(cudaStreamPerThread, &graph);
cudaGraphInstantiateWithFlags(&producer,
                              graph,
                              cudaGraphInstantiateFlagAutoFreeOnLaunch);
cudaGraphDestroy(graph);

// Create first consumer graph by capturing an asynchronous library call
cudaStreamBeginCapture(cudaStreamPerThread, cudaStreamCaptureModeGlobal);
consumerFromLibrary(data1, cudaStreamPerThread);
cudaStreamEndCapture(cudaStreamPerThread, &graph);
cudaGraphInstantiateWithFlags(&consumer1, graph, 0); //regular instantiation
cudaGraphDestroy(graph);

// Create second consumer graph
cudaStreamBeginCapture(cudaStreamPerThread, cudaStreamCaptureModeGlobal);
consume2<<<blocks, threads, 0, cudaStreamPerThread>>>(data2);
...
cudaStreamEndCapture(cudaStreamPerThread, &graph);
cudaGraphInstantiateWithFlags(&consumer2, graph, 0);
cudaGraphDestroy(graph);

// Launch in a loop
bool launchConsumer2 = false;
do {
    cudaGraphLaunch(producer, myStream);
    cudaGraphLaunch(consumer1, myStream);
    if (launchConsumer2) {
        cudaGraphLaunch(consumer2, myStream);
    }
} while (determineAction(&launchConsumer2));

cudaFreeAsync(data1, myStream);
cudaFreeAsync(data2, myStream);

cudaGraphExecDestroy(producer);
cudaGraphExecDestroy(consumer1);
cudaGraphExecDestroy(consumer2);
```

4.2.5.2.5 Memory Nodes in Child Graphs

CUDA 12.9 introduces the ability to move child graph ownership to a parent graph. Child graphs which are moved to the parent are allowed to contain memory allocation and free nodes. This allows a child graph containing allocation or free nodes to be independently constructed prior to its addition in a parent graph.

The following restrictions apply to child graphs after they have been moved:

- Cannot be independently instantiated or destroyed.
- Cannot be added as a child graph of a separate parent graph.
- Cannot be used as an argument to cuGraphExecUpdate.
- Cannot have additional memory allocation or free nodes added.

```
// Create the child graph
cudaGraphCreate(&child, 0);

// parameters for a basic allocation
cudaGraphNodeParams allocNodeParams = { cudaGraphNodeTypeMemAlloc };
allocNodeParams.alloc.poolProps.allocType = cudaMemAllocationTypePinned;
allocNodeParams.alloc.poolProps.location.type = cudaMemLocationTypeDevice;
// specify device 0 as the resident device
allocNodeParams.alloc.poolProps.location.id = 0;
allocNodeParams.alloc.bytesize = size;

cudaGraphAddNode(&allocNode, graph, NULL, NULL, 0, &allocNodeParams);
// Additional nodes using the allocation could be added here
cudaGraphNodeParams freeNodeParams = { cudaGraphNodeTypeMemFree };
freeNodeParams.free.dptr = allocNodeParams.alloc.dptr;
cudaGraphAddNode(&freeNode, graph, &allocNode, NULL, 1, freeNodeParams);

// Create the parent graph
cudaGraphCreate(&parent, 0);

// Move the child graph to the parent graph
cudaGraphNodeParams childNodeParams = { cudaGraphNodeTypeGraph };
childNodeParams.graph.graph = child;
childNodeParams.graph.ownership = cudaGraphChildGraphOwnershipMove;
cudaGraphAddNode(&parentNode, parent, NULL, NULL, 0, &childNodeParams);
```

4.2.5.3 Optimized Memory Reuse

CUDA reuses memory in two ways:

- Virtual and physical memory reuse within a graph is based on virtual address assignment, like in the stream ordered allocator.
- Physical memory reuse between graphs is done with virtual aliasing: different graphs can map the same physical memory to their unique virtual addresses.

4.2.5.3.1 Address Reuse within a Graph

CUDA may reuse memory within a graph by assigning the same virtual address ranges to different allocations whose lifetimes do not overlap. Since virtual addresses may be reused, pointers to different allocations with disjoint lifetimes are not guaranteed to be unique.

The following figure shows adding a new allocation node (2) that can reuse the address freed by a dependent node (1).

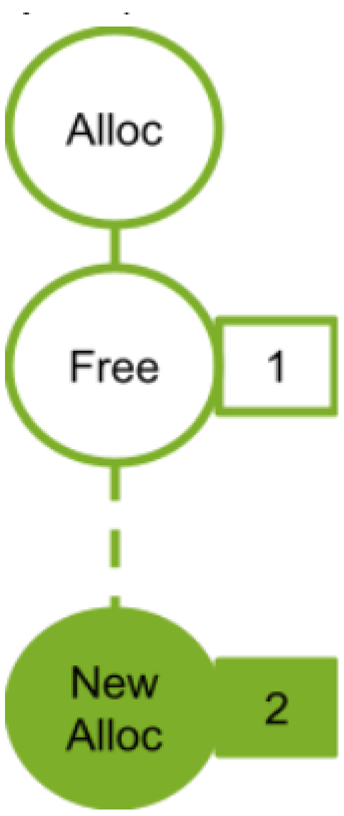


Figure 28: Adding New Alloc Node 2
The following figure shows adding a new alloc node (4). The new alloc node is not dependent on the free node (2) so cannot reuse the address from the associated alloc node (2). If the alloc node (2) used the address freed by free node (1), the new alloc node 3 would need a new address.

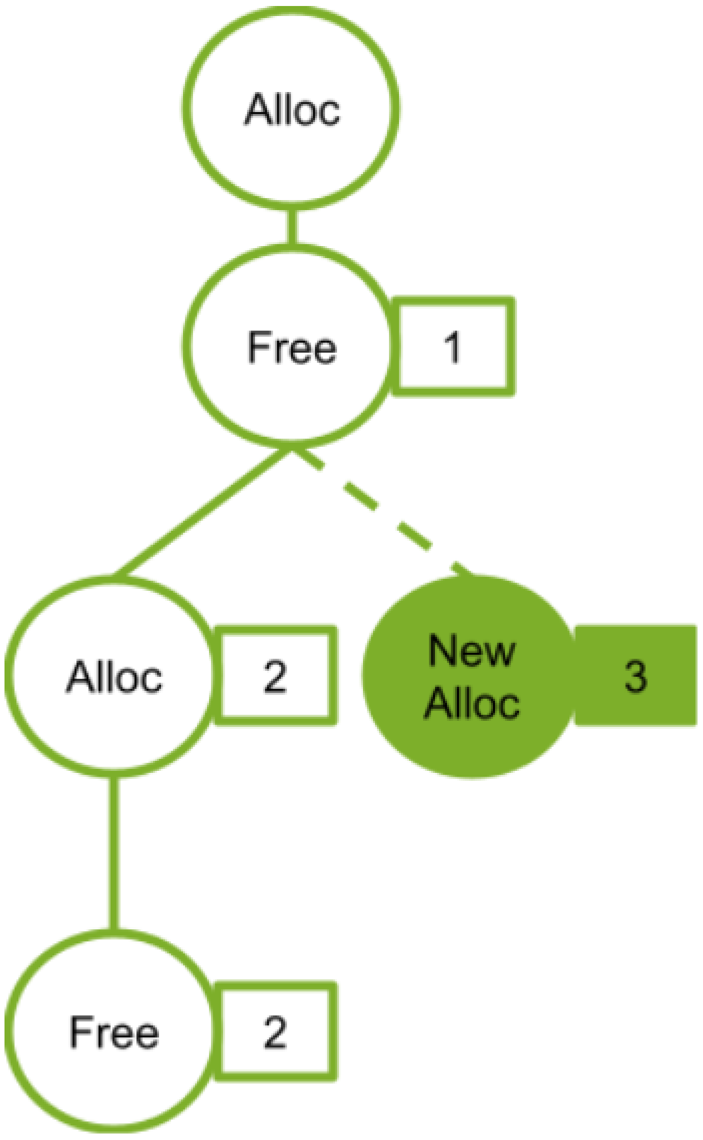


Figure 29: Adding New Alloc Node 3

4.2.5.3.2 Physical Memory Management and Sharing

CUDA is responsible for mapping physical memory to the virtual address before the allocating node is reached in GPU order. As an optimization for memory footprint and mapping overhead, multiple graphs may use the same physical memory for distinct allocations if they will not run simultaneously; however, physical pages cannot be reused if they are bound to more than one executing graph at the same time, or to a graph allocation which remains unfreed.

CUDA may update physical memory mappings at any time during graph instantiation, launch, or execution. CUDA may also introduce synchronization between future graph launches in order to prevent live graph allocations from referring to the same physical memory. As for any allocate-free-allocate pattern, if a program accesses a pointer outside of an allocation's lifetime, the erroneous access may silently read or write live data owned by another allocation (even if the virtual address of the allocation is unique). Use of compute sanitizer tools can catch this error.

The following figure shows graphs sequentially launched in the same stream. In this example, each graph frees all the memory it allocates. Since the graphs in the same stream never run concurrently, CUDA can and should use the same physical memory to satisfy all the allocations.

4.2.5.4 Performance Considerations

When multiple graphs are launched into the same stream, CUDA attempts to allocate the same physical memory to them because the execution of these graphs cannot overlap. Physical mappings for a graph are retained between launches as an optimization to avoid the cost of remapping. If, at a later time, one of the graphs is launched such that its execution may overlap with the others (for example if it is launched into a different stream) then CUDA must perform some remapping because concurrent graphs require distinct memory to avoid data corruption.

In general, remapping of graph memory in CUDA is likely caused by these operations:

- Changing the stream into which a graph is launched
- A trim operation on the graph memory pool, which explicitly frees unused memory (discussed in *graph-memory-nodes-physical-memory-footprint*)
- Relaunching a graph while an unfreed allocation from another graph is mapped to the same memory will cause a remap of memory before relaunch

Remapping must happen in execution order, but after any previous execution of that graph is complete (otherwise memory that is still in use could be unmapped). Due to this ordering dependency, as well as because mapping operations are OS calls, mapping operations can be relatively expensive. Applications can avoid this cost by launching graphs containing allocation memory nodes consistently into the same stream.

4.2.5.4.1 First Launch / cudaGraphUpload

Physical memory cannot be allocated or mapped during graph instantiation because the stream in which the graph will execute is unknown. Mapping is done instead during graph launch. Calling `cudaGraphUpload` can separate out the cost of allocation from the launch by performing all mappings for that graph immediately and associating the graph with the upload stream. If the graph is then launched into the same stream, it will launch without any additional remapping.

Using different streams for graph upload and graph launch behaves similarly to switching streams, likely resulting in remap operations. In addition, unrelated memory pool management is permitted to pull memory from an idle stream, which could negate the impact of the uploads.

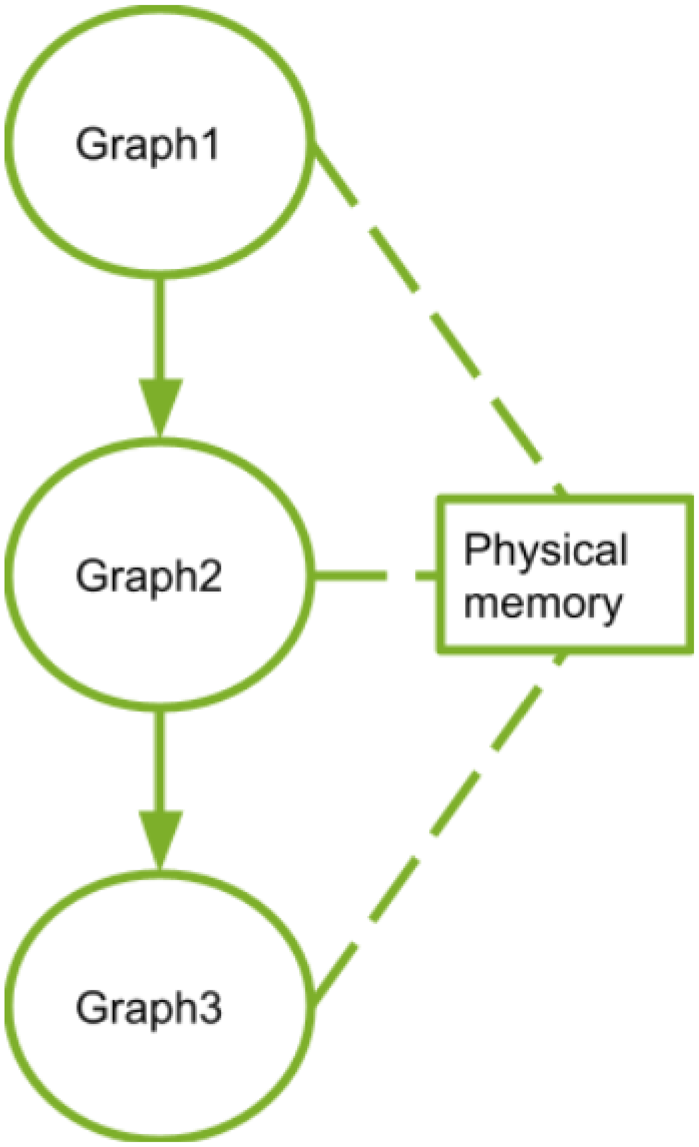


Figure 30: Sequentially Launched Graphs

4.2.5.5 Physical Memory Footprint

The pool-management behavior of asynchronous allocation means that destroying a graph which contains memory nodes (even if their allocations are free) will not immediately return physical memory to the OS for use by other processes. To explicitly release memory back to the OS, an application should use the `cudaDeviceGraphMemTrim` API.

`cudaDeviceGraphMemTrim` will unmap and release any physical memory reserved by graph memory nodes that is not actively in use. Allocations that have not been freed and graphs that are scheduled or running are considered to be actively using the physical memory and will not be impacted. Use of the trim API will make physical memory available to other allocation APIs and other applications or processes, but will cause CUDA to reallocate and remap memory when the trimmed graphs are next launched. Note that `cudaDeviceGraphMemTrim` operates on a different pool from `cudaMemPoolTrimTo()`. The graph memory pool is not exposed to the steam ordered memory allocator. CUDA allows applications to query their graph memory footprint through the `cudaDeviceGetGraphMemAttribute` API. Querying the attribute `cudaGraphMemAttrReservedMemCurrent` returns the amount of physical memory reserved by the driver for graph allocations in the current process. Querying `cudaGraphMemAttrUsedMemCurrent` returns the amount of physical memory currently mapped by at least one graph. Either of these attributes can be used to track when new physical memory is acquired by CUDA for the sake of an allocating graph. Both of these attributes are useful for examining how much memory is saved by the sharing mechanism.

4.2.5.6 Peer Access

Graph allocations can be configured for access from multiple GPUs, in which case CUDA will map the allocations onto the peer GPUs as required. CUDA allows graph allocations requiring different mappings to reuse the same virtual address. When this occurs, the address range is mapped onto all GPUs required by the different allocations. This means an allocation may sometimes allow more peer access than was requested during its creation; however, relying on these extra mappings is still an error.

4.2.5.6.1 Peer Access with Graph Node APIs

The `cudaGraphAddNode` API accepts mapping requests in the `accessDescs` array field of the alloc node parameters structures. The `poolProps.location` embedded structure specifies the resident device for the allocation. Access from the allocating GPU is assumed to be needed, thus the application does not need to specify an entry for the resident device in the `accessDescs` array.

```
cudaGraphNodeParams allocNodeParams = { cudaGraphNodeTypeMemAlloc };
allocNodeParams.alloc.poolProps.allocType = cudaMemAllocationTypePinned;
allocNodeParams.alloc.poolProps.location.type = cudaMemLocationTypeDevice;
// specify device 1 as the resident device
allocNodeParams.alloc.poolProps.location.id = 1;
allocNodeParams.alloc.bytesize = size;

// allocate an allocation resident on device 1 accessible from device 1
cudaGraphAddNode(&allocNode, graph, NULL, NULL, 0, &allocNodeParams);

accessDescs[2];
// boilerplate for the access descs (only ReadWrite and Device access supported
↪by the add node api)
accessDescs[0].flags = cudaMemAccessFlagsProtReadWrite;
accessDescs[0].location.type = cudaMemLocationTypeDevice;
accessDescs[1].flags = cudaMemAccessFlagsProtReadWrite;
accessDescs[1].location.type = cudaMemLocationTypeDevice;
```

(continues on next page)

(continued from previous page)

```
// access being requested for device 0 & 2.  Device 1 access requirement left
↪implicit.
accessDescs[0].location.id = 0;
accessDescs[1].location.id = 2;

// access request array has 2 entries.
allocNodeParams.accessDescCount = 2;
allocNodeParams.accessDescs = accessDescs;

// allocate an allocation resident on device 1 accessible from devices 0, 1 and
↪2. (0 & 2 from the descriptors, 1 from it being the resident device).
cudaGraphAddNode(&allocNode, graph, NULL, NULL, 0, &allocNodeParams);
```

4.2.5.6.2 Peer Access with Stream Capture

For stream capture, the allocation node records the peer accessibility of the allocating pool at the time of the capture. Altering the peer accessibility of the allocating pool after a `cudaMallocFromPoolAsync` call is captured does not affect the mappings that the graph will make for the allocation.

```
// boilerplate for the access descs (only ReadWrite and Device access supported
↪by the add node api)
accessDesc.flags = cudaMemAccessFlagsProtReadWrite;
accessDesc.location.type = cudaMemLocationTypeDevice;
accessDesc.location.id = 1;

// let memPool be resident and accessible on device 0

cudaStreamBeginCapture(stream);
cudaMallocAsync(&dptr1, size, memPool, stream);
cudaStreamEndCapture(stream, &graph1);

cudaMemPoolSetAccess(memPool, &accessDesc, 1);

cudaStreamBeginCapture(stream);
cudaMallocAsync(&dptr2, size, memPool, stream);
cudaStreamEndCapture(stream, &graph2);

//The graph node allocating dptr1 would only have the device 0 accessibility
↪even though memPool now has device 1 accessibility.
//The graph node allocating dptr2 will have device 0 and device 1 accessibility,
↪ since that was the pool accessibility at the time of the cudaMallocAsync
↪call.
```

4.2.6. Device Graph Launch

There are many workflows which need to make data-dependent decisions during runtime and execute different operations depending on those decisions. Rather than offloading this decision-making process to the host, which may require a round-trip from the device, users may prefer to perform it on the device. To that end, CUDA provides a mechanism to launch graphs from the device.

Device graph launch provides a convenient way to perform dynamic control flow from the device, be it something as simple as a loop or as complex as a device-side work scheduler.

Graphs which can be launched from the device will henceforth be referred to as device graphs, and graphs which cannot be launched from the device will be referred to as host graphs.

Device graphs can be launched from both the host and device, whereas host graphs can only be launched from the host. Unlike host launches, launching a device graph from the device while a previous launch of the graph is running will result in an error, returning `cudaErrorInvalidValue`; therefore, a device graph cannot be launched twice from the device at the same time. Launching a device graph from the host and device simultaneously will result in undefined behavior.

4.2.6.1 Device Graph Creation

In order for a graph to be launched from the device, it must be instantiated explicitly for device launch. This is achieved by passing the `cudaGraphInstantiateFlagDeviceLaunch` flag to the `cudaGraphInstantiate()` call. As is the case for host graphs, device graph structure is fixed at time of instantiation and cannot be updated without re-instantiation, and instantiation can only be performed on the host. In order for a graph to be able to be instantiated for device launch, it must adhere to various requirements.

4.2.6.1.1 Device Graph Requirements

General requirements:

- The graph's nodes must all reside on a single device.
- The graph can only contain kernel nodes, memcpy nodes, memset nodes, and child graph nodes.

Kernel nodes:

- Use of CUDA Dynamic Parallelism by kernels in the graph is not permitted.
- Cooperative launches are permitted so long as MPS is not in use.

Memcpy nodes:

- Only copies involving device memory and/or pinned device-mapped host memory are permitted.
- Copies involving CUDA arrays are not permitted.
- Both operands must be accessible from the current device at time of instantiation. Note that the copy operation will be performed from the device on which the graph resides, even if it is targeting memory on another device.

4.2.6.1.2 Device Graph Upload

In order to launch a graph on the device, it must first be uploaded to the device to populate the necessary device resources. This can be achieved in one of two ways.

Firstly, the graph can be uploaded explicitly, either via `cudaGraphUpload()` or by requesting an upload as part of instantiation via `cudaGraphInstantiateWithParams()`.

Alternatively, the graph can first be launched from the host, which will perform this upload step implicitly as part of the launch.

Examples of all three methods can be seen below:

```
// Explicit upload after instantiation
cudaGraphInstantiate(&deviceGraphExec1, deviceGraph1,
↪cudaGraphInstantiateFlagDeviceLaunch);
cudaGraphUpload(deviceGraphExec1, stream);

// Explicit upload as part of instantiation
cudaGraphInstantiateParams instantiateParams = {0};
instantiateParams.flags = cudaGraphInstantiateFlagDeviceLaunch |
↪cudaGraphInstantiateFlagUpload;
instantiateParams.uploadStream = stream;
cudaGraphInstantiateWithParams(&deviceGraphExec2, deviceGraph2, &
↪instantiateParams);

// Implicit upload via host launch
cudaGraphInstantiate(&deviceGraphExec3, deviceGraph3,
↪cudaGraphInstantiateFlagDeviceLaunch);
cudaGraphLaunch(deviceGraphExec3, stream);
```

4.2.6.1.3 Device Graph Update

Device graphs can only be updated from the host, and must be re-uploaded to the device upon executable graph update in order for the changes to take effect. This can be achieved using the same methods outlined in Section *device-graph-upload*. Unlike host graphs, launching a device graph from the device while an update is being applied will result in undefined behavior.

4.2.6.2 Device Launch

Device graphs can be launched from both the host and the device via `cudaGraphLaunch()`, which has the same signature on the device as on the host. Device graphs are launched via the same handle on the host and the device. Device graphs must be launched from another graph when launched from the device.

Device-side graph launch is per-thread and multiple launches may occur from different threads at the same time, so the user will need to select a single thread from which to launch a given graph.

Unlike host launch, device graphs cannot be launched into regular CUDA streams, and can only be launched into distinct named streams, which each denote a specific launch mode. The following table lists the available launch modes.

Table 9: Device-only Graph Launch Streams

Stream	Launch Mode
`cudaStreamGraphFireAndForget`	Fire and forget launch
`cudaStreamGraphTailLaunch`	Tail launch
`cudaStreamGraphFireAndForgetAsSibling`	Sibling launch

4.2.6.2.1 Fire and Forget Launch

As the name suggests, a fire and forget launch is submitted to the GPU immediately, and it runs independently of the launching graph. In a fire-and-forget scenario, the launching graph is the parent, and the launched graph is the child.

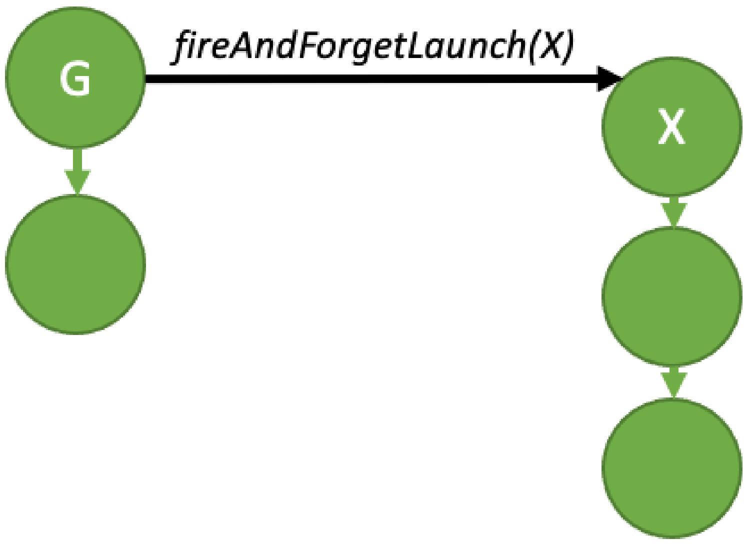


Figure 31: Fire and forget launch

The above diagram can be generated by the sample code below:

```
__global__ void launchFireAndForgetGraph(cudaGraphExec_t graph) {
    cudaGraphLaunch(graph, cudaStreamGraphFireAndForget);
}

void graphSetup() {
    cudaGraphExec_t gExec1, gExec2;
    cudaGraph_t g1, g2;

    // Create, instantiate, and upload the device graph.
    create_graph(&g2);
    cudaGraphInstantiate(&gExec2, g2, cudaGraphInstantiateFlagDeviceLaunch);
    cudaGraphUpload(gExec2, stream);

    // Create and instantiate the launching graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    launchFireAndForgetGraph<<<1, 1, 0, stream>>>(gExec2);
    cudaStreamEndCapture(stream, &g1);
    cudaGraphInstantiate(&gExec1, g1);

    // Launch the host graph, which will in turn launch the device graph.
    cudaGraphLaunch(gExec1, stream);
}
```

A graph can have up to 120 total fire-and-forget graphs during the course of its execution. This total resets between launches of the same parent graph.

4.2.6.2.1.1 Graph Execution Environments

In order to fully understand the device-side synchronization model, it is first necessary to understand the concept of an execution environment.

When a graph is launched from the device, it is launched into its own execution environment. The execution environment of a given graph encapsulates all work in the graph as well as all generated fire and forget work. The graph can be considered complete when it has completed execution and when all generated child work is complete.

The below diagram shows the environment encapsulation that would be generated by the fire-and-forget sample code in the previous section.

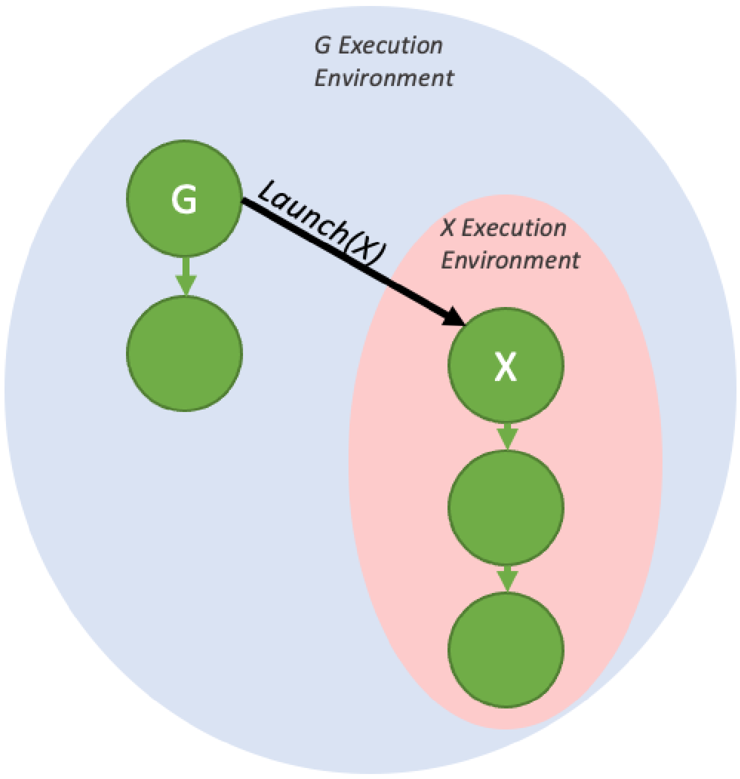


Figure 32: Fire and forget launch, with execution environments

These environments are also hierarchical, so a graph environment can include multiple levels of child-environments from fire and forget launches.

When a graph is launched from the host, there exists a stream environment that parents the execution environment of the launched graph. The stream environment encapsulates all work generated as part of the overall launch. The stream launch is complete (i.e. downstream dependent work may now run) when the overall stream environment is marked as complete.

4.2.6.2.2 Tail Launch

Unlike on the host, it is not possible to synchronize with device graphs from the GPU via traditional methods such as `cudaDeviceSynchronize()` or `cudaStreamSynchronize()`. Rather, in order to enable serial work dependencies, a different launch mode - tail launch - is offered, to provide similar functionality.

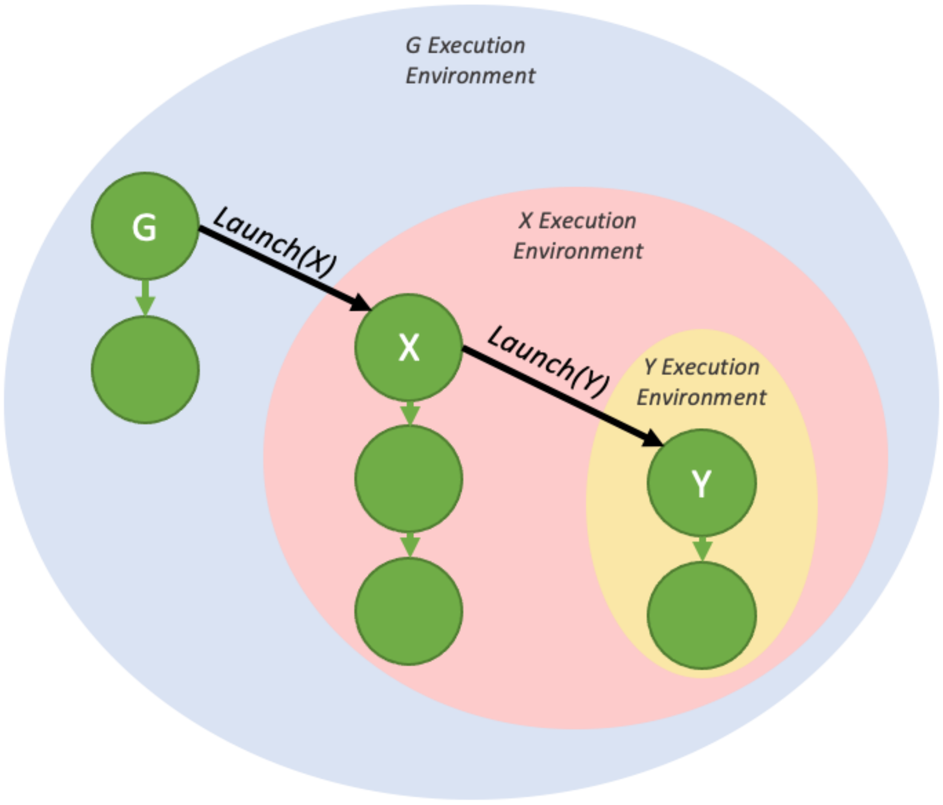


Figure 33: Nested fire and forget environments

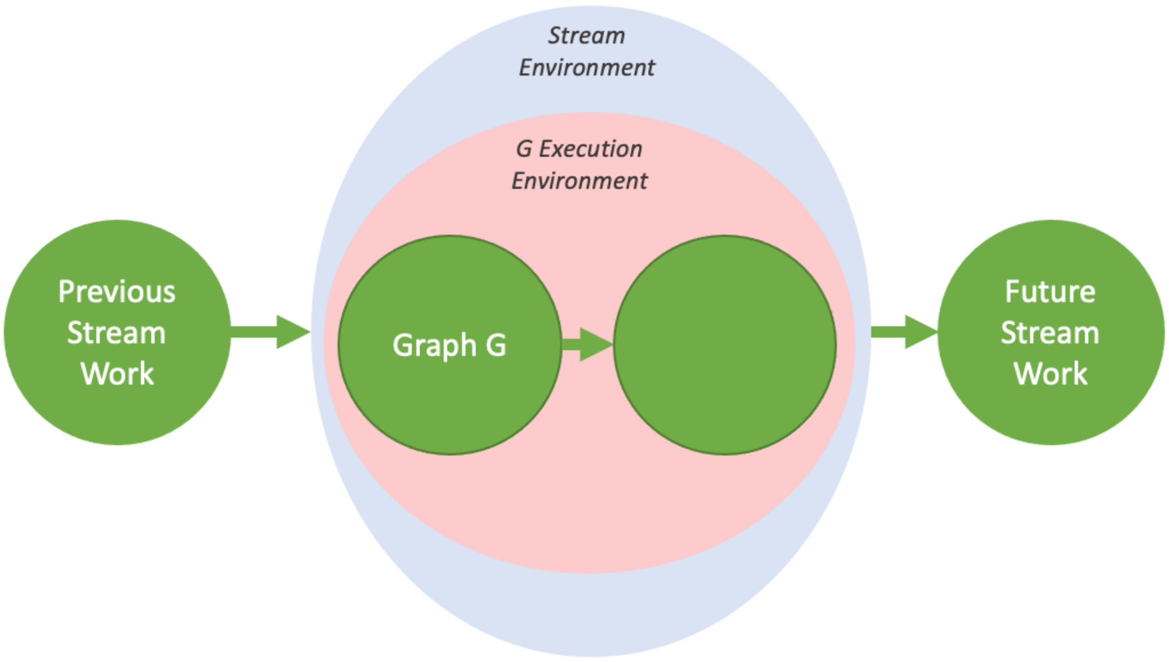


Figure 34: The stream environment, visualized

A tail launch executes when a graph's environment is considered complete - ie, when the graph and all its children are complete. When a graph completes, the environment of the next graph in the tail launch list will replace the completed environment as a child of the parent environment. Like fire-and-forget launches, a graph can have multiple graphs enqueued for tail launch.

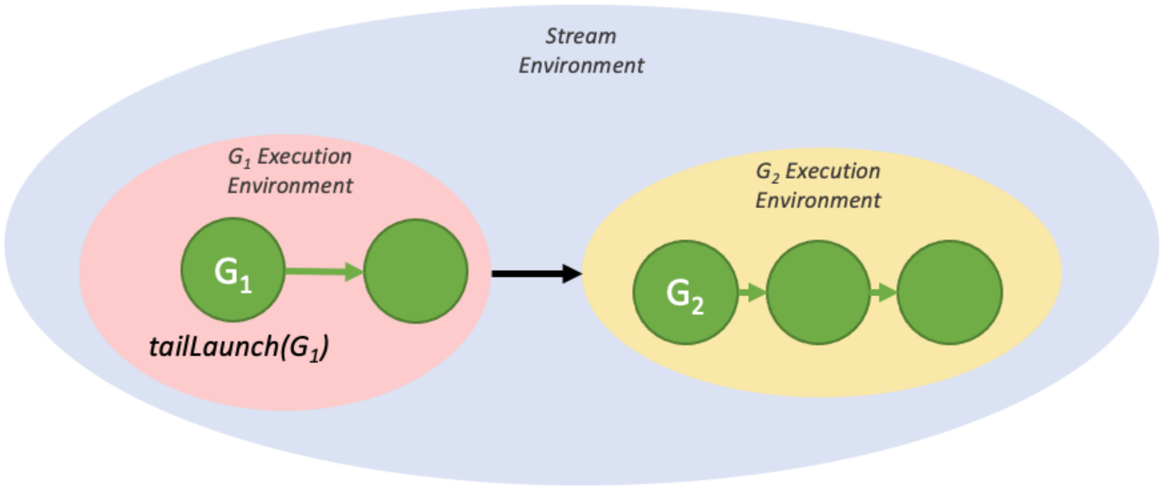


Figure 35: A simple tail launch

The above execution flow can be generated by the code below:

```
__global__ void launchTailGraph(cudaGraphExec_t graph) {
    cudaGraphLaunch(graph, cudaStreamGraphTailLaunch);
}

void graphSetup() {
    cudaGraphExec_t gExec1, gExec2;
    cudaGraph_t g1, g2;

    // Create, instantiate, and upload the device graph.
    create_graph(&g2);
    cudaGraphInstantiate(&gExec2, g2, cudaGraphInstantiateFlagDeviceLaunch);
    cudaGraphUpload(gExec2, stream);

    // Create and instantiate the launching graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    launchTailGraph<<<1, 1, 0, stream>>>(gExec2);
    cudaStreamEndCapture(stream, &g1);
    cudaGraphInstantiate(&gExec1, g1);

    // Launch the host graph, which will in turn launch the device graph.
    cudaGraphLaunch(gExec1, stream);
}
```

Tail launches enqueued by a given graph will execute one at a time, in order of when they were enqueued. So the first enqueued graph will run first, and then the second, and so on.

Tail launches enqueued by a tail graph will execute before tail launches enqueued by previous graphs in the tail launch list. These new tail launches will execute in the order they are enqueued.

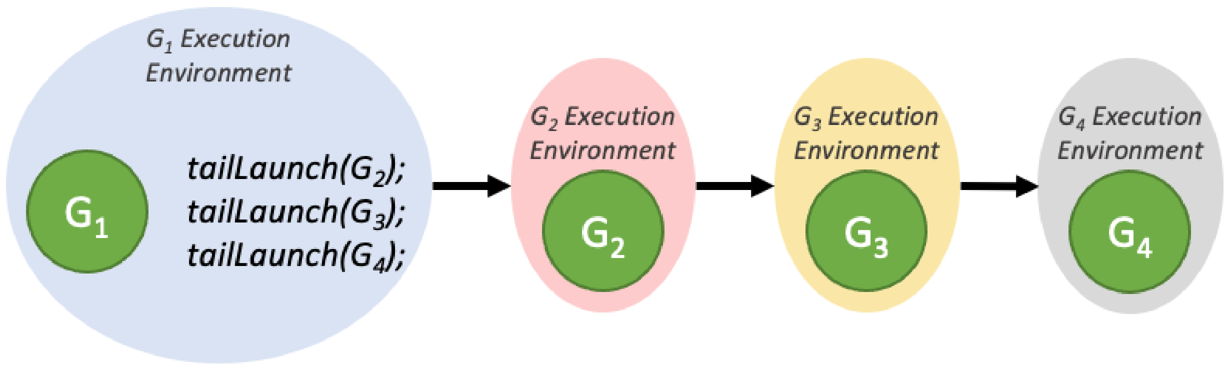

G_1 Execution Environment
G_1
tailLaunch(G_2);
tailLaunch(G_3);
tailLaunch(G_4);
G_2 Execution Environment
G_2
G_3 Execution Environment
G_3
G_4 Execution Environment
G_4

Figure 36: Tail launch ordering

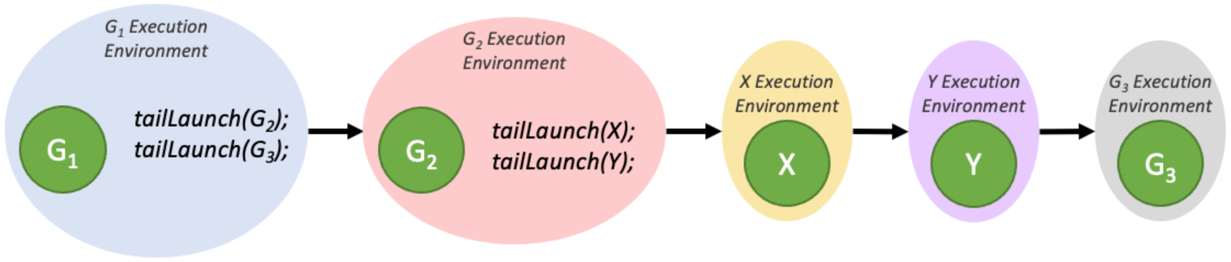


Figure 37: Tail launch ordering when enqueued from multiple graphs

A graph can have up to 255 pending tail launches.

4.2.6.2.2.1 Tail Self-launch

It is possible for a device graph to enqueue itself for a tail launch, although a given graph can only have one self-launch enqueued at a time. In order to query the currently running device graph so that it can be relaunched, a new device-side function is added:

```
cudaGraphExec_t cudaGetCurrentGraphExec();
```

This function returns the handle of the currently running graph if it is a device graph. If the currently executing kernel is not a node within a device graph, this function will return NULL.

Below is sample code showing usage of this function for a relaunch loop:

```
__device__ int relaunchCount = 0;

__global__ void relaunchSelf() {
    int relaunchMax = 100;

    if (threadIdx.x == 0) {
        if (relaunchCount < relaunchMax) {
            cudaGraphLaunch(cudaGetCurrentGraphExec(),
↪cudaStreamGraphTailLaunch);
        }
```

(continues on next page)

(continued from previous page)

```
        relaunchCount++;
    }
}
```

4.2.6.2.3 Sibling Launch

Sibling launch is a variation of fire-and-forget launch in which the graph is launched not as a child of the launching graph's execution environment, but rather as a child of the launching graph's parent environment. Sibling launch is equivalent to a fire-and-forget launch from the launching graph's parent environment.

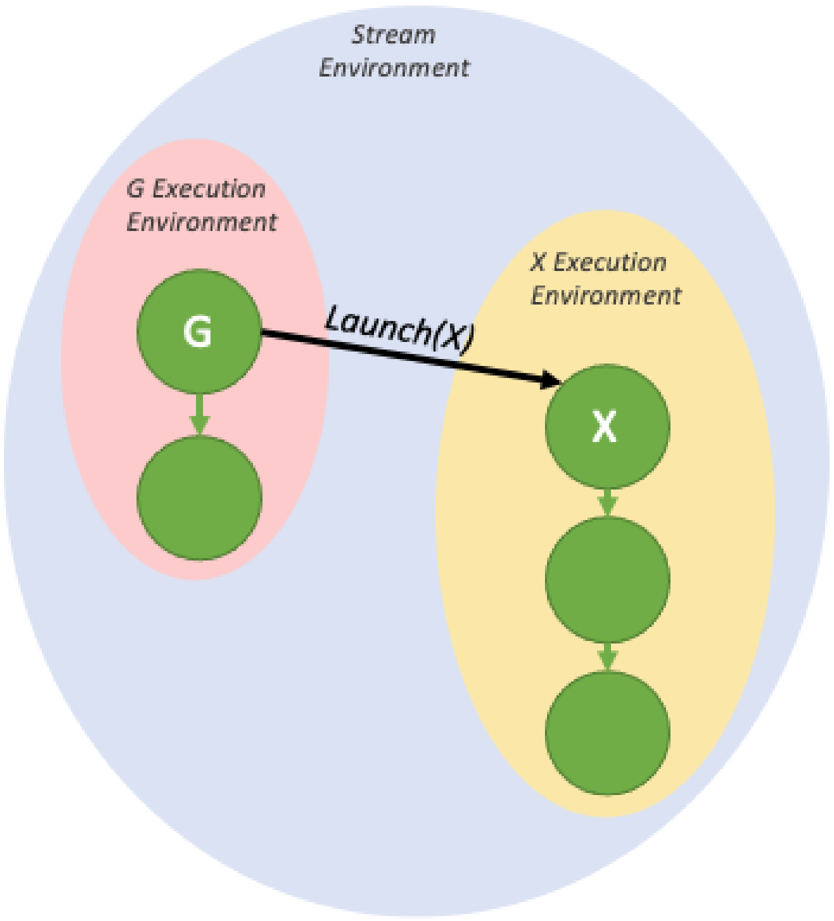


Figure 38: A simple sibling launch

The above diagram can be generated by the sample code below:

```
__global__ void launchSiblingGraph(cudaGraphExec_t graph) {
    cudaGraphLaunch(graph, cudaStreamGraphFireAndForgetAsSibling);
}

void graphSetup() {
```

(continues on next page)

(continued from previous page)

```
    cudaGraphExec_t gExec1, gExec2;
    cudaGraph_t g1, g2;

    // Create, instantiate, and upload the device graph.
    create_graph(&g2);
    cudaGraphInstantiate(&gExec2, g2, cudaGraphInstantiateFlagDeviceLaunch);
    cudaGraphUpload(gExec2, stream);

    // Create and instantiate the launching graph.
    cudaStreamBeginCapture(stream, cudaStreamCaptureModeGlobal);
    launchSiblingGraph<<<1, 1, 0, stream>>>(gExec2);
    cudaStreamEndCapture(stream, &g1);
    cudaGraphInstantiate(&gExec1, g1);

    // Launch the host graph, which will in turn launch the device graph.
    cudaGraphLaunch(gExec1, stream);
}
```

Since sibling launches are not launched into the launching graph's execution environment, they will not gate tail launches enqueued by the launching graph.

4.2.7. Using Graph APIs

`cudaGraph_t` objects are not thread-safe. It is the responsibility of the user to ensure that multiple threads do not concurrently access the same `cudaGraph_t`.

A `cudaGraphExec_t` cannot run concurrently with itself. A launch of a `cudaGraphExec_t` will be ordered after previous launches of the same executable graph.

Graph execution is done in streams for ordering with other asynchronous work. However, the stream is for ordering only; it does not constrain the internal parallelism of the graph, nor does it affect where graph nodes execute.

See Graph API.

4.2.8. CUDA User Objects

CUDA User Objects can be used to help manage the lifetime of resources used by asynchronous work in CUDA. In particular, this feature is useful for *cuda-graphs* and *stream capture*.

Various resource management schemes are not compatible with CUDA graphs. Consider for example an event-based pool or a synchronous-create, asynchronous-destroy scheme.

```
// Library API with pool allocation
void libraryWork(cudaStream_t stream) {
    auto &resource = pool.claimTemporaryResource();
    resource.waitOnReadyEventInStream(stream);
    launchWork(stream, resource);
    resource.recordReadyEvent(stream);
}
```

```
// Library API with asynchronous resource deletion
void libraryWork(cudaStream_t stream) {
    Resource *resource = new Resource(...);
    launchWork(stream, resource);
    cudaLaunchHostFunc(
        stream,
        [](void *resource) {
            delete static_cast<Resource *>(resource);
        },
        resource,
        0);
    // Error handling considerations not shown
}
```

These schemes are difficult with CUDA graphs because of the non-fixed pointer or handle for the resource which requires indirection or graph update, and the synchronous CPU code needed each time the work is submitted. They also do not work with stream capture if these considerations are hidden from the caller of the library, and because of use of disallowed APIs during capture. Various solutions exist such as exposing the resource to the caller. CUDA user objects present another approach.

A CUDA user object associates a user-specified destructor callback with an internal refcount, similar to C++ `shared_ptr`. References may be owned by user code on the CPU and by CUDA graphs. Note that for user-owned references, unlike C++ smart pointers, there is no object representing the reference; users must track user-owned references manually. A typical use case would be to immediately move the sole user-owned reference to a CUDA graph after the user object is created.

When a reference is associated to a CUDA graph, CUDA will manage the graph operations automatically. A cloned `cudaGraph_t` retains a copy of every reference owned by the source `cudaGraph_t`, with the same multiplicity. An instantiated `cudaGraphExec_t` retains a copy of every reference in the source `cudaGraph_t`. When a `cudaGraphExec_t` is destroyed without being synchronized, the references are retained until the execution is completed.

Here is an example use.

```
cudaGraph_t graph;  // Preexisting graph

Object *object = new Object;  // C++ object with possibly nontrivial destructor
cudaUserObject_t cuObject;
cudaUserObjectCreate(
    &cuObject,
    object,  // Here we use a CUDA-provided template wrapper for this API,
             // which supplies a callback to delete the C++ object pointer
    1,  // Initial refcount
    cudaUserObjectNoDestructorSync  // Acknowledge that the callback cannot be
                                    // waited on via CUDA
);
cudaGraphRetainUserObject(
    graph,
    cuObject,
    1,  // Number of references
    cudaGraphUserObjectMove  // Transfer a reference owned by the caller (do
                             // not modify the total reference count)
);
// No more references owned by this thread; no need to call release API
```

(continues on next page)

(continued from previous page)

```
cudaGraphExec_t graphExec;
cudaGraphInstantiate(&graphExec, graph, nullptr, nullptr, 0);  // Will retain
↪a
                                                               // new
↪reference
cudaGraphDestroy(graph);  // graphExec still owns a reference
cudaGraphLaunch(graphExec, 0);  // Async launch has access to the user objects
cudaGraphExecDestroy(graphExec);  // Launch is not synchronized; the release
                                  // will be deferred if needed
cudaStreamSynchronize(0);  // After the launch is synchronized, the remaining
                           // reference is released and the destructor will
                           // execute. Note this happens asynchronously.
// If the destructor callback had signaled a synchronization object, it would
// be safe to wait on it at this point.
```

References owned by graphs in child graph nodes are associated to the child graphs, not the parents. If a child graph is updated or deleted, the references change accordingly. If an executable graph or child graph is updated with `cudaGraphExecUpdate` or `cudaGraphExecChildGraphNodeSetParams`, the references in the new source graph are cloned and replace the references in the target graph. In either case, if previous launches are not synchronized, any references which would be released are held until the launches have finished executing.

There is not currently a mechanism to wait on user object destructors via a CUDA API. Users may signal a synchronization object manually from the destructor code. In addition, it is not legal to call CUDA APIs from the destructor, similar to the restriction on `cudaLaunchHostFunc`. This is to avoid blocking a CUDA internal shared thread and preventing forward progress. It is legal to signal another thread to perform an API call, if the dependency is one way and the thread doing the call cannot block forward progress of CUDA work.

User objects are created with `cudaUserObjectCreate`, which is a good starting point to browse related APIs.

4.3. Stream-Ordered Memory Allocator

4.3.1. Introduction

Managing memory allocations using `cudaMalloc` and `cudaFree` causes the GPU to synchronize across all executing CUDA streams. The stream-ordered memory allocator enables applications to order memory allocation and deallocation with other work launched into a CUDA stream such as kernel launches and asynchronous copies. This improves application memory use by taking advantage of stream-ordering semantics to reuse memory allocations. The allocator also allows applications to control the allocator's memory caching behavior. When set up with an appropriate release threshold, the caching behavior allows the allocator to avoid expensive calls into the OS when the application indicates it is willing to accept a bigger memory footprint. The allocator also supports easy and secure allocation sharing between processes.

Stream-Ordered Memory Allocator:

- Reduces the need for custom memory management abstractions, and makes it easier to create high-performance custom memory management for applications that need it.
- Enables multiple libraries to share a common memory pool managed by the driver. This can reduce excess memory consumption.

- Allows, the driver to perform optimizations based on its awareness of the allocator and other stream management APIs.

Note

Nsight Compute and the Next-Gen CUDA debugger is aware of the allocator since CUDA 11.3.

4.3.2. Memory Management

`cudaMallocAsync` and `cudaFreeAsync` are the APIs which enable stream-ordered memory management. `cudaMallocAsync` returns an allocation and `cudaFreeAsync` frees an allocation. Both APIs accept stream arguments to define when the allocation will become and stop being available for use. These functions allow memory operations to be tied to specific CUDA streams, enabling them to occur without blocking the host or other streams. Application performance can be improved by avoiding potentially costly synchronization of `cudaMalloc` and `cudaFree`.

These APIs can be used for further performance optimization through memory pools, which manage and reuse large blocks of memory for more efficient allocation and deallocation. Memory pools help reduce overhead and prevent fragmentation, improving performance in scenarios with frequent memory allocation operations.

4.3.2.1 Allocating Memory

The `cudaMallocAsync` function triggers asynchronous memory allocation on the GPU, linked to a specific CUDA stream. `cudaMallocAsync` allows memory allocation to occur without hindering the host or other streams, eliminating the need for expensive synchronization.

Note

`cudaMallocAsync` ignores the current device/context when determining where the allocation will reside. Instead, `cudaMallocAsync` determines the appropriate device based on the specified memory pool or the supplied stream.

The listing below illustrates a fundamental use pattern: the memory is allocated, used, and then freed back into the same stream.

```
void *ptr;
size_t size = 512;
cudaMallocAsync(&ptr, size, cudaStreamPerThread);
// do work using the allocation
kernel<<<..., cudaStreamPerThread>>>(ptr, ...);
// An asynchronous free can be specified without synchronizing the cpu and GPU
cudaFreeAsync(ptr, cudaStreamPerThread);
```

Note

When accessing allocation from a stream other than the stream that made the allocation, the user must guarantee that the access occurs after the allocation operation, otherwise the behavior is undefined.

4.3.2.2 Freeing Memory

cudaFreeAsync() asynchronously frees device memory in a stream-ordered fashion, meaning the memory deallocation is assigned to a specific CUDA stream and does not block the host or other streams.

The user must guarantee that the free operation happens after the allocation operation and any uses of the allocation. Any use of the allocation after the free operation starts results in undefined behavior.

Events and/or stream synchronizing operations should be used to guarantee any access to the allocation from other streams is complete before the free operation begins, as illustrated in the following example.

```
cudaMallocAsync(&ptr, size, stream1);
cudaEventRecord(event1, stream1);
//stream2 must wait for the allocation to be ready before accessing
cudaStreamWaitEvent(stream2, event1);
kernel<<<..., stream2>>>(ptr, ...);
cudaEventRecord(event2, stream2);
// stream3 must wait for stream2 to finish accessing the allocation before
// freeing the allocation
cudaStreamWaitEvent(stream3, event2);
cudaFreeAsync(ptr, stream3);
```

Memory allocated with cudaMalloc() can be freed with with cudaFreeAsync(). As above, all accesses to the memory must be complete before the free operation begins.

```
cudaMalloc(&ptr, size);
kernel<<<..., stream>>>(ptr, ...);
cudaFreeAsync(ptr, stream);
```

Likewise, memory allocated with cudaMallocAsync can be freed with cudaFree(). When freeing such allocations through the cudaFree() API, the driver assumes that all accesses to the allocation are complete and performs no further synchronization. The user can use cudaStreamQuery / cudaStreamSynchronize / cudaEventQuery / cudaEventSynchronize / cudaDeviceSynchronize to guarantee that the appropriate asynchronous work is complete and that the GPU will not try to access the allocation.

```
cudaMallocAsync(&ptr, size,stream);
kernel<<<..., stream>>>(ptr, ...);
// synchronize is needed to avoid prematurely freeing the memory
cudaStreamSynchronize(stream);
cudaFree(ptr);
```

4.3.3. Memory Pools

Memory pools encapsulate virtual address and physical memory resources that are allocated and managed according to the pools attributes and properties. The primary aspect of a memory pool is the kind and location of memory it manages.

All calls to cudaMallocAsync use resources from memory pool. If a memory pool is not specified, cudaMallocAsync uses the current memory pool of the supplied stream's device. The current memory pool for a device may be set with cudaDeviceSetMempool and queried with cudaDeviceGetMempool. Each device has a default memory pool, which is active if cudaDeviceSetMempool has not been called.

The API cudaMallocFromPoolAsync and c++ overloads of cudaMallocAsync allow a user to specify the pool to be used for an allocation without setting it as the current pool. The APIs cudaDeviceGetDefaultMempool and cudaMemPoolCreate return handles to memory pools. cudaMemPoolSetAttribute and cudaMemPoolGetAttribute control the attributes of memory pools.

Note

The mempool current to a device will be local to that device. So allocating without specifying a memory pool will always yield an allocation local to the stream's device.

4.3.3.1 Default/Implicit Pools

The default memory pool of a device can be retrieved by calling cudaDeviceGetDefaultMempool. Allocations from the default memory pool of a device are non-migratable device allocation located on that device. These allocations will always be accessible from that device. The accessibility of the default memory pool can be modified with cudaMemPoolSetAccess and queried with cudaMemPoolGetAccess. Since the default pools do not need to be explicitly created, they are sometimes referred to as implicit pools. The default memory pool of a device does not support IPC.

4.3.3.2 Explicit Pools

cudaMemPoolCreate creates an explicit pool. This allows applications to request properties for their allocation beyond what is provided by the default/implicit pools. These include properties such as IPC capability, maximum pool size, allocations resident on a specific CPU NUMA node on supported platforms etc.

```
// create a pool similar to the implicit pool on device 0
int device = 0;
cudaMemPoolProps poolProps = { };
poolProps.allocType = cudaMemAllocationTypePinned;
poolProps.location.id = device;
poolProps.location.type = cudaMemLocationTypeDevice;

cudaMemPoolCreate(&memPool, &poolProps));
```

The following code snippet illustrates an example of creating an IPC capable memory pool on a valid CPU NUMA node.

```
// create a pool resident on a CPU NUMA node that is capable of IPC sharing
↪(via a file descriptor).
int cpu_numa_id = 0;
cudaMemPoolProps poolProps = { };
poolProps.allocType = cudaMemAllocationTypePinned;
poolProps.location.id = cpu_numa_id;
poolProps.location.type = cudaMemLocationTypeHostNuma;
poolProps.handleType = cudaMemHandleTypePosixFileDescriptor;

cudaMemPoolCreate(&ipcMemPool, &poolProps));
```

4.3.3.3 Device Accessibility for Multi-GPU Support

Like allocation accessibility controlled through the virtual memory management APIs, memory pool allocation accessibility does not follow cudaDeviceEnablePeerAccess or cuCtxEnablePeerAccess. For memory pools, the API cudaMemPoolSetAccess modifies what devices can access allocations from a pool. By default, allocations are accessible only from the device where the allocations are located. This access cannot be revoked. To enable access from other devices, the accessing device must be peer capable with the memory pool's device. This can be verified with cudaDeviceCanAccessPeer. If the peer capability is not checked, the set access may fail with cudaErrorInvalidDevice. However, if no allocations had been made from the pool, the cudaMemPoolSetAccess call may succeed even when the devices are not peer capable. In this case, the next allocation from the pool will fail.

It is worth noting that cudaMemPoolSetAccess affects all allocations from the memory pool, not just future ones. Likewise, the accessibility reported by cudaMemPoolGetAccess applies to all allocations from the pool, not just future ones. Changing the accessibility settings of a pool for a given GPU frequently is not recommended. That is, once a pool is made accessible from a given GPU, it should remain accessible from that GPU for the lifetime of the pool.

```
// snippet showing usage of cudaMemPoolSetAccess:
cudaError_t setAccessOnDevice(cudaMemPool_t memPool, int residentDevice,
              int accessingDevice) {
    cudaMemAccessDesc accessDesc = {};
    accessDesc.location.type = cudaMemLocationTypeDevice;
    accessDesc.location.id = accessingDevice;
    accessDesc.flags = cudaMemAccessFlagsProtReadWrite;

    int canAccess = 0;
    cudaError_t error = cudaDeviceCanAccessPeer(&canAccess, accessingDevice,
              residentDevice);
    if (error != cudaSuccess) {
        return error;
    } else if (canAccess == 0) {
        return cudaErrorPeerAccessUnsupported;
    }

    // Make the address accessible
    return cudaMemPoolSetAccess(memPool, &accessDesc, 1);
}
```

4.3.3.4 Enabling Memory Pools for IPC

Memory pools can be enabled for interprocess communication (IPC) to allow easy, efficient and secure sharing of GPU memory between processes. CUDA's IPC memory pools provide the same security benefits as CUDA's virtual memory management APIs.

There are two steps to sharing memory between processes with memory pools: the processes first needs to share access to the pool, then share specific allocations from that pool. The first step establishes and enforces security. The second step coordinates what virtual addresses are used in each process and when mappings need to be valid in the importing process.

4.3.3.4.1 Creating and Sharing IPC Memory Pools

Sharing access to a pool involves retrieving an OS-native handle to the pool with `cudaMemPoolExportToShareableHandle()`, transferring the handle to the importing process using OS-native IPC mechanisms, and then creating an imported memory pool with the `cudaMemPoolImportFromShareableHandle()` API. For `cudaMemPoolExportToShareableHandle` to succeed, the memory pool must have been created with the requested handle type specified in the pool properties structure.

Please reference samples for the appropriate IPC mechanisms to transfer the OS-native handle between processes. The rest of the procedure can be found in the following code snippets.

```
// in exporting process
// create an exportable IPC capable pool on device 0
cudaMemPoolProps poolProps = { };
poolProps.allocType = cudaMemAllocationTypePinned;
poolProps.location.id = 0;
poolProps.location.type = cudaMemLocationTypeDevice;

// Setting handleTypes to a non zero value will make the pool exportable (IPC
↪capable)
poolProps.handleTypes = CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR;

cudaMemPoolCreate(&memPool, &poolProps));

// FD based handles are integer types
int fdHandle = 0;


// Retrieve an OS native handle to the pool.
// Note that a pointer to the handle memory is passed in here.
cudaMemPoolExportToShareableHandle(&fdHandle,
             memPool,
             CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR,
             0);

// The handle must be sent to the importing process with the appropriate
// OS-specific APIs.
```

```
// in importing process
 int fdHandle;
// The handle needs to be retrieved from the exporting process with the
// appropriate OS-specific APIs.
// Create an imported pool from the shareable handle.
// Note that the handle is passed by value here.
cudaMemPoolImportFromShareableHandle(&importedMemPool,
          (void*)fdHandle,
          CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR,
          0);
```

4.3.3.4.2 Set Access in the Importing Process

Imported memory pools are initially only accessible from their resident device. The imported memory pool does not inherit any accessibility set by the exporting process. The importing process needs to enable access with `cudaMemPoolSetAccess` from any GPU it plans to access the memory from.

If the imported memory pool belongs to a device that is not visible to importing process, the user must use the `cudaMemPoolSetAccess` API to enable access from the GPUs the allocations will be used on. (See *Device Accessibility for Multi-GPU Support*)

4.3.3.4.3 Creating and Sharing Allocations from an Exported Pool

Once the pool has been shared, allocations made with `cudaMallocAsync()` from the pool in the exporting process can be shared with processes that have imported the pool. Since the pool's security policy is established and verified at the pool level, the OS does not need extra bookkeeping to provide security for specific pool allocations. In other words, the opaque `cudaMemPoolPtrExportData` required to import a pool allocation may be sent to the importing process using any mechanism.

While allocations may be exported and imported without synchronizing with the allocating stream in any way, the importing process must follow the same rules as the exporting process when accessing the allocation. Specifically, access to the allocation must happen after the allocation operation in the allocating stream executes. The two following code snippets show `cudaMemPoolExportPointer()` and `cudaMemPoolImportPointer()` sharing the allocation with an IPC event used to guarantee that the allocation isn't accessed in the importing process before the allocation is ready.

```
// preparing an allocation in the exporting process
cudaMemPoolPtrExportData exportData;
cudaEvent_t readyIpcEvent;
cudaIpcEventHandle_t readyIpcEventHandle;

// ipc event for coordinating between processes
// cudaEventInterprocess flag makes the event an ipc event
// cudaEventDisableTiming  is set for performance reasons

cudaEventCreate(&readyIpcEvent, cudaEventDisableTiming |
↪cudaEventInterprocess)

// allocate from the exporting mem pool
cudaMallocAsync(&ptr, size,exportMemPool, stream);

// event for sharing when the allocation is ready.
cudaEventRecord(readyIpcEvent, stream);
cudaMemPoolExportPointer(&exportData, ptr);
cudaIpcGetEventHandle(&readyIpcEventHandle, readyIpcEvent);

// Share IPC event and pointer export data with the importing process using
//  any mechanism. Here we copy the data into shared memory
shmem->ptrData = exportData;
shmem->readyIpcEventHandle = readyIpcEventHandle;
// signal consumers data is ready
```

```
// Importing an allocation
cudaMemPoolPtrExportData *importData = &shmem->prtData;
cudaEvent_t readyIpcEvent;
```

(continues on next page)

(continued from previous page)

```
cudaIpcEventHandle_t *readyIpcEventHandle = &shmem->readyIpcEventHandle;

// Need to retrieve the ipc event handle and the export data from the
// exporting process using any mechanism.  Here we are using shmem and just
// need synchronization to make sure the shared memory is filled in.

cudaIpcOpenEventHandle(&readyIpcEvent, readyIpcEventHandle);

// import the allocation. The operation does not block on the allocation being
↪ready.
cudaMemPoolImportPointer(&ptr, importedMemPool, importData);

// Wait for the prior stream operations in the allocating stream to complete
↪before
// using the allocation in the importing process.
cudaStreamWaitEvent(stream, readyIpcEvent);
kernel<<<..., stream>>>(ptr, ...);
```

When freeing the allocation, the allocation must be freed in the importing process before it is freed in the exporting process. The following code snippet demonstrates the use of CUDA IPC events to provide the required synchronization between the cudaFreeAsync operations in both processes. Access to the allocation from the importing process is obviously restricted by the free operation in the importing process side. It is worth noting that cudaFree can be used to free the allocation in both processes and that other stream synchronization APIs may be used instead of CUDA IPC events.

```
// The free must happen in importing process before the exporting process
kernel<<<..., stream>>>(ptr, ...);

// Last access in importing process
cudaFreeAsync(ptr, stream);

// Access not allowed in the importing process after the free
cudaIpcEventRecord(finishedIpcEvent, stream);
```

```
// Exporting process
// The exporting process needs to coordinate its free with the stream order
// of the importing process's free.
cudaStreamWaitEvent(stream, finishedIpcEvent);
kernel<<<..., stream>>>(ptrInExportingProcess, ...);

// The free in the importing process doesn't stop the exporting process
// from using the allocation.
cudFreeAsync(ptrInExportingProcess,stream);
```

4.3.3.4.4 IPC Export Pool Limitations

IPC pools currently do not support releasing physical blocks back to the OS. As a result the cudaMemPoolTrimTo API has no effect and the cudaMemPoolAttrReleaseThreshold is effectively ignored. This behavior is controlled by the driver, not the runtime and may change in a future driver update.

4.3.3.4.5 IPC Import Pool Limitations

Allocating from an import pool is not allowed; specifically, import pools cannot be set current and cannot be used in the `cudaMallocFromPoolAsync` API. As such, the allocation reuse policy attributes do not have meaning for these pools.

IPC Import pools, like IPC export pools, currently do not support releasing physical blocks back to the OS.

The resource usage stat attribute queries only reflect the allocations imported into the process and the associated physical memory.

4.3.4. Best Practices and Tuning

4.3.4.1 Query for Support

An application can determine whether or not a device supports the stream-ordered memory allocator by calling `cudaDeviceGetAttribute()` (see developer blog) with the device attribute `cudaDevAttrMemoryPoolsSupported`.

IPC memory pool support can be queried with the `cudaDevAttrMemoryPoolSupportedHandleTypes` device attribute. This attribute was added in CUDA 11.3, and older drivers will return `cudaErrorInvalidValue` when this attribute is queried.

```
int driverVersion = 0;
int deviceSupportsMemoryPools = 0;
int poolSupportedHandleTypes = 0;
cudaDriverGetVersion(&driverVersion);
if (driverVersion >= 11020) {
    cudaDeviceGetAttribute(&deviceSupportsMemoryPools,
                           cudaDevAttrMemoryPoolsSupported, device);
}
if (deviceSupportsMemoryPools != 0) {
    // `device` supports the Stream-Ordered Memory Allocator
}

if (driverVersion >= 11030) {
    cudaDeviceGetAttribute(&poolSupportedHandleTypes,
              cudaDevAttrMemoryPoolSupportedHandleTypes, device);
}
if (poolSupportedHandleTypes & cudaMemHandleTypePosixFileDescriptor) {
   // Pools on the specified device can be created with posix file descriptor-
↪based IPC
}
```

Performing the driver version check before the query avoids hitting a `cudaErrorInvalidValue` error on drivers where the attribute was not yet defined. One can use `cudaGetLastError` to clear the error instead of avoiding it.

4.3.4.2 Physical Page Caching Behavior

By default, the allocator tries to minimize the physical memory owned by a pool. To minimize the OS calls to allocate and free physical memory, applications must configure a memory footprint for each pool. Applications can do this with the release threshold attribute (`cudaMemPoolAttrReleaseThreshold`).

The release threshold is the amount of memory in bytes a pool should hold onto before trying to release memory back to the OS. When more than the release threshold bytes of memory are held by the memory pool, the allocator will try to release memory back to the OS on the next call to stream, event or device synchronize. Setting the release threshold to UINT64_MAX will prevent the driver from attempting to shrink the pool after every synchronization.

```
Cuuint64_t setVal = UINT64_MAX;
cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrReleaseThreshold, &setVal);
```

Applications that set `cudaMemPoolAttrReleaseThreshold` high enough to effectively disable memory pool shrinking may wish to explicitly shrink a memory pool's memory footprint. `cudaMemPoolTrimTo` allows applications to do so. When trimming a memory pool's footprint, the `minBytesToKeep` parameter allows an application to hold onto a specified amount of memory, for example the amount it expects to need in a subsequent phase of execution.

```
Cuuint64_t setVal = UINT64_MAX;
cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrReleaseThreshold, &setVal);

// application phase needing a lot of memory from the stream-ordered allocator
for (i=0; i<10; i++) {
    for (j=0; j<10; j++) {
        cudaMallocAsync(&ptrs[j],size[j], stream);
    }
    kernel<<<...,stream>>>(ptrs,...);
    for (j=0; j<10; j++) {
        cudaFreeAsync(ptrs[j], stream);
    }
}

// Process does not need as much memory for the next phase.
// Synchronize so that the trim operation will know that the allocations are no
// longer in use.
cudaStreamSynchronize(stream);
cudaMemPoolTrimTo(mempool, 0);

// Some other process/allocation mechanism can now use the physical memory
// released by the trimming operation.
```

4.3.4.3 Resource Usage Statistics

Querying the `cudaMemPoolAttrReservedMemCurrent` attribute of a pool reports the current total physical GPU memory consumed by the pool. Querying the `cudaMemPoolAttrUsedMemCurrent` of a pool returns the total size of all of the memory allocated from the pool and not available for reuse.

The`cudaMemPoolAttr*MemHigh` attributes are watermarks recording the max value achieved by the respective `cudaMemPoolAttr*MemCurrent` attribute since last reset. They can be reset to the current value by using the `cudaMemPoolSetAttribute` API.

```
// sample helper functions for getting the usage statistics in bulk
struct usageStatistics {
    cuuint64_t reserved;
    cuuint64_t reservedHigh;
    cuuint64_t used;
    cuuint64_t usedHigh;
```

(continues on next page)

(continued from previous page)

```
};

void getUsageStatistics(cudaMemoryPool_t memPool, struct usageStatistics
↪*statistics)
{
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrReservedMemCurrent,
↪statistics->reserved);
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrReservedMemHigh,
↪statistics->reservedHigh);
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrUsedMemCurrent,
↪statistics->used);
    cudaMemPoolGetAttribute(memPool, cudaMemPoolAttrUsedMemHigh, statistics->
↪usedHigh);
}


// resetting the watermarks will make them take on the current value.
void resetStatistics(cudaMemoryPool_t memPool)
{
    cuuint64_t value = 0;
    cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrReservedMemHigh, &value);
    cudaMemPoolSetAttribute(memPool, cudaMemPoolAttrUsedMemHigh, &value);
}
```

4.3.4.4 Memory Reuse Policies

In order to service an allocation request, the driver attempts to reuse memory that was previously freed via `cudaFreeAsync()` before attempting to allocate more memory from the OS. For example, memory freed in a stream can be reused immediately in a subsequent allocation request on the same stream. When a stream is synchronized with the CPU, the memory that was previously freed in that stream becomes available for reuse for an allocation in any stream. Reuse policies can be applied to both default and explicit memory pools.

The stream-ordered allocator has a few controllable allocation policies. The pool attributes `cudaMemPoolReuseFollowEventDependencies`, `cudaMemPoolReuseAllowOpportunistic`, and `cudaMemPoolReuseAllowInternalDependencies` control these policies and are detailed below. These policies can be enabled or disabled through a call to `cudaMemPoolSetAttribute`. Upgrading to a newer CUDA driver may change, enhance, augment and/or reorder the enumeration of the reuse policies.

4.3.4.4.1 cudaMemPoolReuseFollowEventDependencies

Before allocating more physical GPU memory, the allocator examines dependency information established by CUDA events and tries to allocate from memory freed in another stream.

```
cudaMallocAsync(&ptr, size, originalStream);
kernel<<<..., originalStream>>>(ptr, ...);
cudaFreeAsync(ptr, originalStream);
cudaEventRecord(event,originalStream);

// waiting on the event that captures the free in another stream
// allows the allocator to reuse the memory to satisfy
```

(continues on next page)

(continued from previous page)

```
// a new allocation request in the other stream when
// cudaMemPoolReuseFollowEventDependencies is enabled.
cudaStreamWaitEvent(otherStream, event);
cudaMallocAsync(&ptr2, size, otherStream);
```

4.3.4.4.2 cudaMemPoolReuseAllowOpportunistic

When the `cudaMemPoolReuseAllowOpportunistic` policy is enabled, the allocator examines freed allocations to see if the free operations stream order semantic has been met, for example the stream has passed the point of execution indicated by the free operation. When this policy is disabled, the allocator will still reuse memory made available when a stream is synchronized with the CPU. Disabling this policy does not stop the `cudaMemPoolReuseFollowEventDependencies` from applying.

```
cudaMallocAsync(&ptr, size, originalStream);
kernel<<<..., originalStream>>>(ptr, ...);
cudaFreeAsync(ptr, originalStream);


// after some time, the kernel finishes running
wait(10);

// When cudaMemPoolReuseAllowOpportunistic is enabled this allocation request
// can be fulfilled with the prior allocation based on the progress of
↪originalStream.
cudaMallocAsync(&ptr2, size, otherStream);
```

4.3.4.4.3 cudaMemPoolReuseAllowInternalDependencies

Failing to allocate and map more physical memory from the OS, the driver will look for memory whose availability depends on another stream's pending progress. If such memory is found, the driver will insert the required dependency into the allocating stream and reuse the memory.

```
cudaMallocAsync(&ptr, size, originalStream);
kernel<<<..., originalStream>>>(ptr, ...);
cudaFreeAsync(ptr, originalStream);

// When cudaMemPoolReuseAllowInternalDependencies is enabled
// and the driver fails to allocate more physical memory, the driver may
// effectively perform a cudaStreamWaitEvent in the allocating stream
// to make sure that future work in 'otherStream' happens after the work
// in the original stream that would be allowed to access the original
↪allocation.
cudaMallocAsync(&ptr2, size, otherStream);
```

4.3.4.4.4 Disabling Reuse Policies

While the controllable reuse policies improve memory reuse, users may want to disable them. Allowing opportunistic reuse (such as `cudaMemPoolReuseAllowOpportunistic`) introduces run to run variance in allocation patterns based on the interleaving of CPU and GPU execution. Internal dependency insertion (such as `cudaMemPoolReuseAllowInternalDependencies`) can serialize work in unexpected and potentially non-deterministic ways when the user would rather explicitly synchronize an event or stream on allocation failure.

4.3.4.5 Synchronization API Actions

One of the optimizations that comes with the allocator being part of the CUDA driver is integration with the synchronize APIs. When the user requests that the CUDA driver synchronize, the driver waits for asynchronous work to complete. Before returning, the driver will determine what frees the synchronization guaranteed to be completed. These allocations are made available for allocation regardless of specified stream or disabled allocation policies. The driver also checks `cudaMemPoolAttrReleaseThreshold` here and releases any excess physical memory that it can.

4.3.5. Addendums

4.3.5.1 cudaMemcpyAsync Current Context/Device Sensitivity

In the current CUDA driver, any async `memcpy` involving memory from `cudaMallocAsync` should be done using the specified stream's context as the calling thread's current context. This is not necessary for `cudaMemcpyPeerAsync`, as the device primary contexts specified in the API are referenced instead of the current context.

4.3.5.2 cudaPointerGetAttributes Query

Invoking `cudaPointerGetAttributes` on an allocation after invoking `cudaFreeAsync` on it results in undefined behavior. Specifically, it does not matter if an allocation is still accessible from a given stream: the behavior is still undefined.

4.3.5.3 cudaGraphAddMemsetNode

`cudaGraphAddMemsetNode` does not work with memory allocated via the stream ordered allocator. However, memsets of the allocations can be stream captured.

4.3.5.4 Pointer Attributes

The `cudaPointerGetAttributes` query works on stream-ordered allocations. Since stream-ordered allocations are not context associated, querying `CU_POINTER_ATTRIBUTE_CONTEXT` will succeed but return NULL in `*data`. The attribute `CU_POINTER_ATTRIBUTE_DEVICE_ORDINAL` can be used to determine the location of the allocation: this can be useful when selecting a context for making p2h2p copies using `cudaMemcpyPeerAsync`. The attribute `CU_POINTER_ATTRIBUTE_MEMPOOL_HANDLE` was added in CUDA 11.3 and can be useful for debugging and for confirming which pool an allocation comes from before doing IPC.

4.3.5.5 CPU Virtual Memory

When using CUDA stream-ordered memory allocator APIs, avoid setting VRAM limitations with "ulimit -v" as this is not supported.

4.4. Cooperative Groups

4.4.1. Introduction

Cooperative Groups are an extension to the CUDA programming model for organizing groups of collaborating threads. Cooperative Groups allow developers to control the granularity at which threads are collaborating, helping them to express richer, more efficient parallel decompositions. Cooperative Groups also provide implementations of common parallel primitives like scan and parallel reduce.

Historically, the CUDA programming model has provided a single, simple construct for synchronizing cooperating threads: a barrier across all threads of a thread block, as implemented with the `__syncthreads()` intrinsic function. In an effort to express broader patterns of parallel interaction, many performance-oriented programmers have resorted to writing their own ad hoc and unsafe primitives for synchronizing threads within a single warp, or across sets of thread blocks running on a single GPU. Whilst the performance improvements achieved have often been valuable, this has resulted in an ever-growing collection of brittle code that is expensive to write, tune, and maintain over time and across GPU generations. Cooperative Groups provides a safe and future-proof mechanism for writing performant code.

The full Cooperative Groups API is available in the *Cooperative Groups API*.

4.4.2. Cooperative Group Handle & Member Functions

Cooperative Groups are managed via a Cooperative Group Handle. The Cooperative Group handle allows participating threads to learn their position in the group, the group size, and other group information. Select group member functions are shown in the following table.

Table 10: Select Member Functions

Accessor	Returns
`thread_rank()`	The rank of the calling thread.
`num_threads()`	The total number of threads in the group .
`thread_index()`	A 3-Dimensional index of the thread within the launched block.
`dim_threads()`	The 3D dimensions of the launched block in units of threads.

A complete list pf member functions is available in the *Cooperative Groups API*.

4.4.3. Default Behavior / Groupless Execution

Groups representing the grid and thread blocks are implicitly created based on the kernel launch configuration. These "implicit" groups provide a starting point that developers can explicitly decompose into finer grained groups. Implicit groups can be accessed using the following methods:

Table 11: Cooperative Groups Implicitly Created by CUDA Runtime

Accessor	Group Scope
`this_thread_block()`	Returns the handle to a group containing all threads in current thread block.
`this_grid()`	Returns the handle to a group containing all threads in the grid.
`coalesced_threads()`[1]	Returns the handle to a group of currently active threads in a warp.
`this_cluster()`[2]	Returns the handle to a group of threads in the current cluster.

More information is available in the *Cooperative Groups API*.

4.4.3.1 Create Implicit Group Handles As Early As Possible

For best performance it is recommended that you create a handle for the implicit group upfront (as early as possible, before any branching has occurred) and use that handle throughout the kernel.

4.4.3.2 Only Pass Group Handles by Reference

It is recommended that you pass group handles by reference to functions when passing a group handle into a function. Group handles must be initialized at declaration time, as there is no default constructor. Copy-constructing group handles is discouraged.

4.4.4. Creating Cooperative Groups

Groups are created by partitioning a parent group into subgroups. When a group is partitioned, a group handle is created to manage the resulting subgroup. The following partitioning operations are available to developers:

Table 12: Cooperative Group Partitioning Operations

Partition Type	Description
tiled_partition	Divides parent group into a series of fixed-size subgroups arranged in a one-dimensional, row-major format.
stride_partition	Divides parent group into equally-sized subgroups where threads are assigned to subgroups in a round-robin manner.
labeled_partition	Divides parent group into one-dimensional subgroups based on a conditional label, which can be any integral type.
binary_partition	Specialized form of labeled partitioning where label can only be “0” or “1”.

The following example shows how a tiled partition is created:

[1] The `coalesced_threads()` operator returns the set of active threads at that point in time, and makes no guarantee about which threads are returned (as long as they are active) or that they will stay coalesced throughout execution.

[2] The `this_cluster()` assumes a 1x1x1 cluster when a non-cluster grid is launched. Requires Compute Capability 9.0 or greater.

```
namespace cg = cooperative_groups;
// Obtain the current thread's cooperative group
cg::thread_block my_group = cg::this_thread_block();

// Partition the cooperative group into tiles of size 8
cg::thread_block_tile<8> my_subgroup = cg::tiled_partition<8>(cta);

// do work as my_subgroup
```

The best partitioning strategy to use depends on the context. More information is available in the *Cooperative Groups API*.

4.4.4.1 Avoiding Group Creation Hazards

Partitioning a group is a collective operation and all threads in the group must participate. If the group was created in a conditional branch that not all threads reach, this can lead to deadlocks or data corruption.

4.4.5. Synchronization

Prior to the introduction of Cooperative Groups, the CUDA programming model only allowed synchronization between thread blocks at a kernel completion boundary. Cooperative groups allows developers to synchronize groups of cooperating threads at different granularities.

4.4.5.1 Sync

You can synchronize a group by calling the collective `sync()` function. Like `__syncthreads()`, the `sync()` function makes the following guarantees: - All memory accesses (e.g., reads and writes) made by threads in the group before the synchronization point are visible to all threads in the group after the synchronization point. - All threads in the group reach the synchronization point before any thread is allowed to proceed beyond it.

The following example shows a `cooperative_groups::sync()` that is equivalent to `__syncthreads()`.

```
namespace cg = cooperative_groups;

cg::thread_block my_group = cg::this_thread_block();

// Synchronize threads in the block
cg::sync(my_group);
```

Cooperative groups can be used to synchronize the entire grid. As of CUDA 13, cooperative groups can no longer be used for multi-device synchronization. For details see the *Large Scale Groups* section.

More information about synchronization is available in the *Cooperative Groups API*.

4.4.5.2 Barriers

Cooperative Groups provides a barrier API similar to `cuda::barrier` that can be used for more advanced synchronization. Cooperative Groups barrier API differs from `cuda::barrier` in a few key ways: - Cooperative Groups barriers are automatically initialized - All threads in the group must arrive and wait at the barrier once per phase. - `barrier_arrive` returns an `arrival_token` object that must be passed into the corresponding `barrier_wait`, where it is consumed and cannot be used again.

Programmers must take care to avoid hazards when using Cooperative Groups barriers: - No collective operations can be used by a group between after calling `barrier_arrive` and before calling `barrier_wait`. - `barrier_wait` only guarantees that all threads in the group have called `barrier_arrive`. `barrier_wait` does NOT guarantee that all threads have called `barrier_wait`.

```
namespace cg = cooperative_groups;

cg::thread_block my_group = this_block();

auto token = cluster.barrier_arrive();

// Optional: Do some local processing to hide the synchronization latency
local_processing(block);

// Make sure all other blocks in the cluster are running and initialized shared
↪data before accessing dsmem
cluster.barrier_wait(std::move(token));
```

4.4.6. Collective Operations

Cooperative Groups includes a set of collective operations that can be performed by a group of threads. These operations require participation of all threads in the specified group in order to complete the operation.

All threads in the group must pass the same values for corresponding arguments to each collective call, unless different values are explicitly allowed in the *Cooperative Groups API*. Otherwise the behavior of the call is undefined.

4.4.6.1 Reduce

The `reduce` function is used to perform a parallel reduction on the data provided by each thread in the specified group. The type of reduction must be specified by providing one of the operators shown in the following table.

Table 13: Cooperative Groups Reduction Operators

Operator	Returns
plus	Sum of all values in group
less	Minimum value
greater	Maximum value
bit_and	Bitwise AND reduction
bit_or	Bitwise OR reduction
bit_xor	Bitwise XOR reduction

Hardware acceleration is used for reductions when available (requires Compute Capability 8.0 or greater). A software fallback is available for older hardware where hardware acceleration is not available. Only 4B types are accelerated by hardware.

More information about reductions is available in the *Cooperative Groups API*.

The following example shows how to use `cooperative_groups::reduce()` to perform a block-wide sum reduction.

```
namespace cg = cooperative_groups;

cg::thread_block my_group = cg::this_thread_block();

int val = data[threadIdx.x];

int sum = cg::reduce(cta, val, cg::plus<int>());

// Store the result from the reduction
if (my_group.thread_rank() == 0) {
   result[blockIdx.x] = sum;
}
```

4.4.6.2 Scans

Cooperative Groups includes implementations of `inclusive_scan` and `exclusive_scan` that can be used on arbitrary group sizes. The functions perform a scan operation on the data provided by each thread named in the specified group.

Programmers can optionally specify a reduction operator, as listed in *Reduction Operators Table* above.

```
namespace cg = cooperative_groups;

cg::thread_block my_group = cg::this_thread_block();

int val = data[my_group.thread_rank()];

int exclusive_sum = cg::exclusive_scan(my_group, val, cg::plus<int>());

result[my_group.thread_rank()] = exclusive_sum;
```

More information about scans is available in the *Cooperative Groups Scan API*.

4.4.6.3 Invoke One

Cooperative Groups provides an `invoke_one` function for use when a single thread must perform a serial portion of work on behalf of a group. - `invoke_one` selects a single arbitrary thread from the calling group and uses that thread to call the supplied invocable function using the supplied arguments. - `invoke_one_broadcast` is the same as `invoke_one` except the result of the call is also broadcast to all threads in the group.

The thread selection mechanism is not guaranteed to be deterministic.

The following example shows basic `invoke_one` utilization.

```
namespace cg = cooperative_groups;
cg::thread_block my_group = cg::this_thread_block();

```

(continues on next page)

(continued from previous page)

```
// Ensure only one thread in the thread block prints the message
cg::invoke_one(my_group, []() {
   printf("Hello from one thread in the block!");
});

// Synchronize to make sure all threads wait until the message is printed
cg::sync(my_group);
```

Communication or synchronization within the calling group is not allowed inside the invocable function. Communication with threads outside of the calling group is allowed.

4.4.7. Asynchronous Data Movement

Cooperative Groups `memcpy_async` functionality in CUDA provides a way to perform asynchronous memory copies between global memory and shared memory. `memcpy_async` is particularly useful for optimizing memory transfers and overlapping computation with data transfer to improve performance.

The `memcpy_async` function is used to start an asynchronous load from global memory to shared memory. `memcpy_async` is intended to be used like a "prefetch" where data is loaded before it is needed.

The `wait` function forces all threads in a group to wait until the asynchronous memory transfer is completed. `wait` must be called by all threads in the group before the data can be accessed in shared memory.

The following example shows how to use `memcpy_async` and `wait` to prefetch data.

```
namespace cg = cooperative_groups;

cg::thread_group my_group = cg::this_thread_block();

__shared__ int shared_data[];

// Perform an asynchronous copy from global memory to shared memory
cg::memcpy_async(my_group, shared_data + my_group.rank(), input + my_group.
↪rank(), sizeof(int));

// Hide latency by doing work here. Cannot use shared_data

// Wait for the asynchronous copy to complete
cg::wait(my_group);

// Prefetched data is now available
```

See the *Cooperative Groups API* for more information.

4.4.7.1 Memcpy Async Alignment Requirements

`memcpy_async` is only asynchronous if the source is global memory and the destination is shared memory and both are at least 4-byte aligned. For achieving best performance: an alignment of 16 bytes for both shared memory and global memory is recommended.

4.4.8. Large Scale Groups

Cooperative Groups allows for large groups that span the entire grid. All Cooperative Group functionality described previously is available to these large groups, with one notable exception: synchronizing the entire grid requires using the `cudaLaunchCooperativeKernel` runtime launch API.

Multi-device launch APIs and related references for Cooperative Groups have been removed as of CUDA 13.

4.4.8.1 When to use `cudaLaunchCooperativeKernel`

`cudaLaunchCooperativeKernel` is a CUDA runtime API function used to launch a single-device kernel that employs cooperative groups, specifically designed for executing kernels that require inter-block synchronization. This function ensures that all threads in the kernel can synchronize and cooperate across the entire grid, which is not possible with traditional CUDA kernels that only allow synchronization within individual thread blocks. `cudaLaunchCooperativeKernel` ensures that the kernel launch is atomic, i.e. if the API call succeeds, then the provided number of thread blocks will launch on the specified device.

It is good practice to first ensure the device supports cooperative launches by querying the device attribute `cudaDevAttrCooperativeLaunch`:

```
int dev = 0;
int supportsCoopLaunch = 0;
cudaDeviceGetAttribute(&supportsCoopLaunch, cudaDevAttrCooperativeLaunch,
↪dev);
```

which will set `supportsCoopLaunch` to 1 if the property is supported on device 0. Only devices with compute capability of 6.0 and higher are supported. In addition, you need to be running on either of these:

- The Linux platform without MPS
- The Linux platform with MPS and on a device with compute capability 7.0 or higher
- The latest Windows platform

4.5. Programmatic Dependent Launch and Synchronization

The *Programmatic Dependent Launch* mechanism allows for a dependent *secondary* kernel to launch before the *primary* kernel it depends on in the same CUDA stream has finished executing. Available starting with devices of compute capability 9.0, this technique can provide performance benefits when the *secondary* kernel can complete significant work that does not depend on the results of the *primary* kernel.

4.5.1. Background

A CUDA application utilizes the GPU by launching and executing multiple kernels on it. A typical GPU activity timeline is shown in Figure 39.

Here, `secondary_kernel` is launched after `primary_kernel` finishes its execution. Serialized execution is usually necessary because `secondary_kernel` depends on result data produced by `pri-`

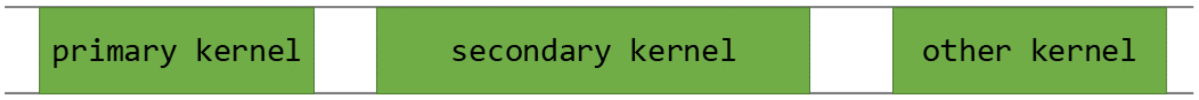


Figure 39: GPU activity timeline

mary_kernel. If secondary_kernel has no dependency on primary_kernel, both of them can be launched concurrently by using *CUDA Streams*. Even if secondary_kernel is dependent on primary_kernel, there is some potential for concurrent execution. For example, almost all the kernels have some sort of *preamble* section during which tasks such as zeroing buffers or loading constant values are performed.

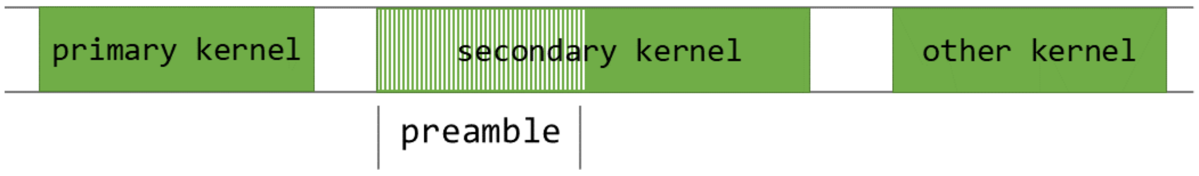


Figure 40: Preamble section of secondary_kernel

Figure 40 demonstrates the portion of secondary_kernel that could be executed concurrently without impacting the application. Note that concurrent launch also allows us to hide the launch latency of secondary_kernel behind the execution of primary_kernel.

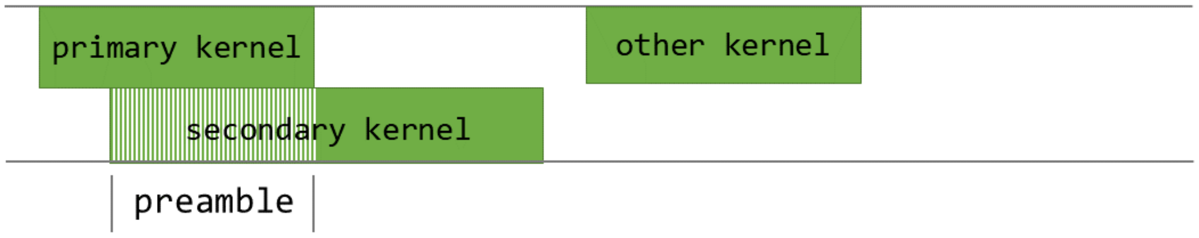


Figure 41: Concurrent execution of primary_kernel and secondary_kernel

The concurrent launch and execution of secondary_kernel shown in Figure 41 is achievable using *Programmatic Dependent Launch*.

Programmatic Dependent Launch introduces changes to the CUDA kernel launch APIs as explained in following section. These APIs require at least compute capability 9.0 to provide overlapping execution.

4.5.2. API Description

In Programmatic Dependent Launch, a primary and a secondary kernel are launched in the same CUDA stream. The primary kernel should execute `cudaTriggerProgrammaticLaunchCompletion` with all thread blocks when it's ready for the secondary kernel to launch. The secondary kernel must be launched using the extensible launch API as shown.

```
__global__ void primary_kernel() {
   // Initial work that should finish before starting secondary kernel

   // Trigger the secondary kernel
   cudaTriggerProgrammaticLaunchCompletion();

   // Work that can coincide with the secondary kernel
}

__global__ void secondary_kernel()
{
   // Independent work

   // Will block until all primary kernels the secondary kernel is dependent
↪on have completed and flushed results to global memory
   cudaGridDependencySynchronize();

   // Dependent work
}

cudaLaunchAttribute attribute[1];
attribute[0].id = cudaLaunchAttributeProgrammaticStreamSerialization;
attribute[0].val.programmaticStreamSerializationAllowed = 1;
configSecondary.attrs = attribute;
configSecondary.numAttrs = 1;

primary_kernel<<<grid_dim, block_dim, 0, stream>>>();
cudaLaunchKernelEx(&configSecondary, secondary_kernel);
```

When the secondary kernel is launched using the `cudaLaunchAttributeProgrammaticStreamSerialization` attribute, the CUDA driver is safe to launch the secondary kernel early and not wait on the completion and memory flush of the primary before launching the secondary.

The CUDA driver can launch the secondary kernel when all primary thread blocks have launched and executed `cudaTriggerProgrammaticLaunchCompletion`. If the primary kernel doesn't execute the trigger, it implicitly occurs after all thread blocks in the primary kernel exit.

In either case, the secondary thread blocks might launch before data written by the primary kernel is visible. As such, when the secondary kernel is configured with *Programmatic Dependent Launch*, it must always use `cudaGridDependencySynchronize` or other means to verify that the result data from the primary is available.

Please note that these methods provide the opportunity for the primary and secondary kernels to execute concurrently, however this behavior is opportunistic and not guaranteed to lead to concurrent kernel execution. Reliance on concurrent execution in this manner is unsafe and can lead to deadlock.

4.5.3. Use in CUDA Graphs

Programmatic Dependent Launch can be used in *CUDA Graphs* via *stream capture* or directly via *edge data*. To program this feature in a CUDA Graph with edge data, use a `cudaGraphDependencyType` value of `cudaGraphDependencyTypeProgrammatic` on an edge connecting two kernel nodes. This edge type makes the upstream kernel visible to a `cudaGridDependencySynchronize()` in the downstream kernel. This type must be used with an outgoing port of either `cudaGraphKernelNodePortLaunchCompletion` or `cudaGraphKernelNodePortProgrammatic`.

The resulting graph equivalents for stream capture are as follows:

Stream code (abbreviated)	Resulting graph edge

`cudaLaunchAttribute attribute;`
`attribute.id =`
`↪cudaLaunchAttributeProgrammaticStrea`
`↪`
`attribute.val.`
`↪programmaticStreamSerializationAllow`
`↪= 1;`
	`cudaGraphEdgeData edgeData;`
	`edgeData.type =`
	`↪cudaGraphDependencyTypeProgrammatic;`
	`↪`
	`edgeData.from_port =`
	`↪cudaGraphKernelNodePortProgrammatic;`
	`↪`

`cudaLaunchAttribute attribute;`
`attribute.id =`
`↪cudaLaunchAttributeProgrammaticEvent`
`↪`
`attribute.val.programmaticEvent.`
`↪triggerAtBlockStart = 0;`
	`cudaGraphEdgeData edgeData;`
	`edgeData.type =`
	`↪cudaGraphDependencyTypeProgrammatic;`
	`↪`
	`edgeData.from_port =`
	`↪cudaGraphKernelNodePortProgrammatic;`
	`↪`

`cudaLaunchAttribute attribute;`
`attribute.id =`
`↪cudaLaunchAttributeProgrammaticEvent`
`↪`
`attribute.val.programmaticEvent.`
`↪triggerAtBlockStart = 1;`
	`cudaGraphEdgeData edgeData;`
	`edgeData.type =`
	`↪cudaGraphDependencyTypeProgrammatic;`
	`↪`
	`edgeData.from_port =`
	`↪cudaGraphKernelNodePortLaunchCompletion;`
	`↪`

4.6. Green Contexts

A green context (GC) is a lightweight context associated, from its creation, with a set of specific GPU resources. Users can partition GPU resources, currently streaming multiprocessors (SMs) and work queues (WQs), during green context creation, so that GPU work targeting a green context can only use its provisioned SMs and work queues. Doing so can be beneficial in reducing, or better controlling, interference due to use of common resources. An application can have multiple green contexts.

Using green contexts does not require any GPU code (kernel) changes, just small host-side changes (e.g., green context creation and stream creation for this green context). The green context functionality can be useful in various scenarios. For example, it can help ensure some SMs are always available for a latency-sensitive kernel to start executing, assuming no other constraints, or provide a quick way to test the effect of using fewer SMs without any kernel modifications.

Green context support first became available via the CUDA Driver API. Starting from CUDA 13.1, contexts are exposed in the CUDA runtime via the execution context (EC) abstraction. Currently, an execution context can correspond to either the primary context (the context runtime API users have always implicitly interacted with) or a green context. This section will use the terms *execution context* and *green context* interchangeably when referring to a green context.

With the runtime exposure of green contexts, using the CUDA runtime API directly is strongly recommended. This section will also solely use the CUDA runtime API.

The remaining of this section is organized as follows: Section 4.6.1 provides a motivating example, Section 4.6.2 highlights ease of use, and Section 4.6.3 presents the device resource and resource descriptor structs. Section 4.6.4 explains how to create a green context, Section 4.6.5 how to launch work that targets it, and Section 4.6.6 highlights some additional green context APIs. Finally, Section 4.6.7 wraps up with an example.

4.6.1. Motivation / When to Use

When launching a CUDA kernel, the user has no direct control over the number of SMs that kernel will execute on. One can only indirectly influence this by changing the kernel's launch geometry or anything that can affect the kernel's maximum number of active thread blocks per SM. Additionally, when multiple kernels execute in parallel on the GPU (kernels running on different CUDA streams or as part of a CUDA graph), they may also contend for the same SM resources.

There are, however, use cases where the user needs to ensure there are always GPU resources available for latency-sensitive work to start, and thus complete, as soon as possible. Green contexts provide a way towards that by partitioning SM resources, so a given green context can only use specific SMs (the ones provisioned during its creation).

Figure 42 illustrates such an example. Assume an application where two independent kernels A and B run on two different non-blocking CUDA streams. Kernel A is launched first and starts executing occupying all available SM resources. When, later in time, latency-sensitive kernel B is launched, no SM resources are available. As a result, kernel B can only start executing once kernel A ramps down, i.e., once thread blocks from kernel A finish executing. The first graph illustrates this scenario where critical work B gets delayed. The y-axis shows the percentage of SMs occupied and x-axis depicts time.

Using green contexts, one could partition the GPU's SMs, so that green context A, targeted by kernel A, has access to some SMs of the GPU, while green context B, targeted by kernel B, has access to the remaining SMs. In this setting, kernel A can only use the SMs provisioned for green context A, irrespective of its launch configuration. As a result, when critical kernel B gets launched, it is guaranteed that there will be available SMs for it to start executing immediately, barring any other resource constraints. As the second graph in Figure 42 illustrates, even though the duration of kernel A may increase, latency-sensitive work B will no longer be delayed due to unavailable SMs. The figure shows that green context A is provisioned with an SM count equivalent to 80% SMs of the GPU for illustration purposes.

This behavior can be achieved without any code modifications to kernels A and B. One simply needs to ensure they are launched on CUDA streams belonging to the appropriate green contexts. The number of SMs each green context will have access to should be decided by the user during green context creation on a per case basis.

Work Queues:

Streaming multiprocessors are one resource type that can be provisioned for a green context. Another

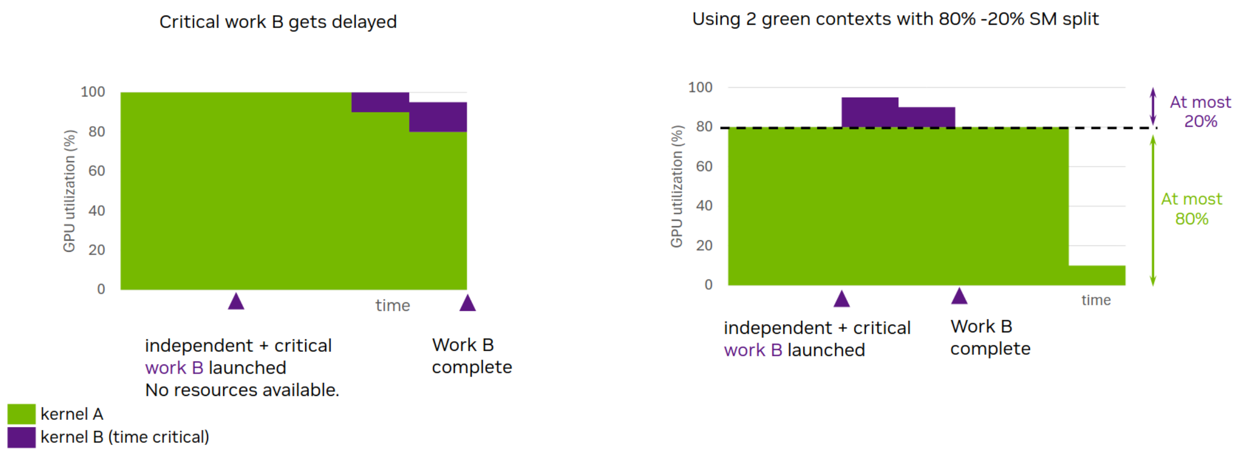


Figure 42: Motivation: GCs' static resource partitioning enables latency-sensitive work B to start and complete sooner

resource type is work queues. Think of a workqueue as a black-box resource abstraction, which can also influence GPU work execution concurrency, along with other factors. If independent GPU work tasks (e.g., kernels submitted on different CUDA streams) map to the same workqueue, a false dependence between these tasks may be introduced, which can lead to their serialized execution. The user can influence the upper limit of work queues on the GPU via the `CUDA_DEVICE_MAX_CONNECTIONS` environment variable (see Section 5.2, Section 3.1).

Building on top of the previous example, assume work B maps to the same workqueue as work A. In that case, even if SM resources are available (green contexts case), work B may still need to wait for work A to complete in its entirety. Similar to SMs, the user has no direct control over the specific work queues that may be used under the hood. But green contexts allow the user to express the maximum concurrency they would expect in terms of expected number of concurrent stream-ordered workloads. The driver can then use this value as a hint to try to prevent work from different execution contexts from using the same workqueue(s), thus preventing unwanted interference across execution contexts.

Attention

Even when different SM resources and work queues are provisioned per green context, concurrent execution of independent GPU work is not guaranteed. It is best to think of all the techniques described under the *Green Contexts* section as removing factors which can prevent concurrent execution (i.e., reducing potential interference).

Green Contexts versus MIG or MPS

For completeness, this section briefly compares green contexts with two other resource partitioning mechanisms: MIG (Multi-Instance GPU) and MPS (Multi-Process Service).

MIG statically partitions a MIG-supported GPU into multiple MIG instances ("smaller GPUs"). This partitioning has to happen before the launch of an application, and different applications can use different MIG instances. Using MIG can be beneficial for users whose applications consistently underutilize the available GPU resources; an issue more pronounced as GPUs get bigger. With MIG, users can run these different applications on different MIG instances, thus improving GPU utilization. MIG can be attractive for cloud service providers (CSPs) not only for the increased GPU utilization for such applications, but also for the quality of service (QoS) and isolation it can provide across clients running on different MIG instances. Please refer to the MIG documentation linked above for more details.

But using MIG cannot address the problematic scenario described earlier, where critical work B is delayed because all SM resources are occupied by other GPU work from the same application. This issue can still exist for an application running on a single MIG instance. To address it, one can use green contexts alongside MIG. In that case, the SM resources available for partitioning would be the resources of the given MIG instance.

MPS primarily targets different processes (e.g., MPI programs), allowing them to run on the GPU at the same time without time-slicing. It requires an MPS daemon to be running before the application is launched. By default, MPS clients will contend for all available SM resources of the GPU or the MIG instance they are running on. In this multiple client processes setting, MPS can support dynamic partitioning of SM resources, using the *active thread percentage* option, which places an upper limit on the percentage of SMs an MPS client process can use. Unlike green contexts, the active thread percentage partitioning happens with MPS at the process level, and the percentage is typically specified by an environment variable before the application is launched. The MPS active thread percentage signifies that a given client application cannot use more than *x%* of a GPU's SMs, let that be N SMs. However, these SMs can be *any* N SMs of the GPU, which can also vary over time. On the other hand, a green context provisioned with N SMs during its creation can only use these specific N SMs.

Starting with CUDA 13.1, MPS also supports static partitioning, if it is explicitly enabled when starting the MPS control daemon. With static partitioning, the user has to specify the static partition an MPS client process can use, when the application is launched. Dynamic sharing with active thread percentage is no longer applicable in that case. A key difference between MPS in static partitioning mode and green contexts is that MPS targets different processes, while green contexts is applicable within a single process too. Also, contrary to green contexts, MPS with static partitioning does not allow oversubscription of SM resources.

With MPS, programmatic partitioning of SM resources is also possible for a CUDA context created via the `cuCtxCreate` driver API, with execution affinity. This programmatic partitioning allows different client CUDA contexts from one or more processes to each use up to a specified number of SMs. As with the active thread percentage partitioning, these SMs can be *any* SMs of the GPU and can vary over time, unlike the green contexts case. This option is possible even under the presence of static MPS partitioning. Please note that creating a green context is much more lightweight in comparison to an MPS context, as many underlying structures are owned by the primary context and thus shared.

4.6.2. Green Contexts: Ease of use

To highlight how easy it is to use green contexts, assume you have the following code snippet that creates two CUDA streams and then calls a function that launches kernels via <<<>>> on these CUDA streams. As discussed earlier, other than changing the kernels' launch geometries, one cannot influence how many SMs these kernels can use.

```
int gpu_device_index = 0; // GPU ordinal
CUDA_CHECK(cudaSetDevice(gpu_device_index));

cudaStream_t strm1, strm2;
CUDA_CHECK(cudaStreamCreateWithFlags(&strm1, cudaStreamNonBlocking));
CUDA_CHECK(cudaStreamCreateWithFlags(&strm2, cudaStreamNonBlocking));

// No control over how many SMs kernel(s) running on each stream can use
code_that_launches_kernels_on_streams(strm1, strm2); // what is abstracted in
↪this function + the kernels is the vast majority of your code

// cleanup code not shown
```

Starting with CUDA 13.1, one can control the number of SMs a given kernel can have access to, using green contexts. The code snippet below shows how easy it is to do that. With a few extra lines and without any kernel modifications, you can control the SMs resources kernel(s) launched on these different streams can use.

```
int gpu_device_index = 0; // GPU ordinal
CUDA_CHECK(cudaSetDevice(gpu_device_index));

/* ------------------ Code required to create green contexts -------------------
↪--------- */


// Get all available GPU SM resources
cudaDevResource initial_GPU_SM_resources {};
CUDA_CHECK(cudaDeviceGetDevResource(gpu_device_index, &initial_GPU_SM_
↪resources, cudaDevResourceTypeSm));

// Split SM resources. This example creates one group with 16 SMs and one with
↪8. Assuming your GPU has >= 24 SMs
cudaDevSmResource result[2] {{}, {}};
cudaDevSmResourceGroupParams group_params[2] =  {
        {.smCount=16, .coscheduledSmCount=0, .preferredCoscheduledSmCount=0, .
↪flags=0},
        {.smCount=8,  .coscheduledSmCount=0, .preferredCoscheduledSmCount=0, .
↪flags=0}};
CUDA_CHECK(cudaDevSmResourceSplit(&result[0], 2, &initial_GPU_SM_resources,
↪nullptr, 0, &group_params[0]));

// Generate resource descriptors for each resource
cudaDevResourceDesc_t resource_desc1 {};
cudaDevResourceDesc_t resource_desc2 {};
CUDA_CHECK(cudaDevResourceGenerateDesc(&resource_desc1, &result[0], 1));
CUDA_CHECK(cudaDevResourceGenerateDesc(&resource_desc2, &result[1], 1));

// Create green contexts
cudaExecutionContext_t my_green_ctx1 {};
cudaExecutionContext_t my_green_ctx2 {};
CUDA_CHECK(cudaGreenCtxCreate(&my_green_ctx1, resource_desc1, gpu_device_
↪index, 0));
CUDA_CHECK(cudaGreenCtxCreate(&my_green_ctx2, resource_desc2, gpu_device_
↪index, 0));

/* ------------------ Modified code --------------------------- */

// You just need to use a different CUDA API to create the streams
cudaStream_t strm1, strm2;
CUDA_CHECK(cudaExecutionCtxStreamCreate(&strm1, my_green_ctx1,
↪cudaStreamDefault, 0));
CUDA_CHECK(cudaExecutionCtxStreamCreate(&strm2, my_green_ctx2,
↪cudaStreamDefault, 0));

/* ------------------ Unchanged code --------------------------- */
```

(continues on next page)

(continued from previous page)

```
// No need to modify any code in this function or in your kernel(s).
// Reminder: what is abstracted in this function + kernels is the vast majority
↪of your code
// Now kernel(s) running on stream strm1 will use at most 16 SMs and kernel(s)
↪on strm2 at most 8 SMs.
code_that_launches_kernels_on_streams(strm1, strm2);

// cleanup code not shown
```

Various execution context APIs, some of which were shown in the previous example, take an explicit `cudaExecutionContext_t` handle and thus ignore the context that is current to the calling thread. Until now, CUDA runtime users who did not use the driver API would by default only interact with the primary context that is implicitly set as current to a thread via `cudaSetDevice()`. This shift to explicit context-based programming provides easier to understand semantics and can have additional benefits compared to the previous implicit context-based programming that relied on thread-local state (TLS).

The following sections will explain all the steps shown in the previous code snippet in detail.

4.6.3. Green Contexts: Device Resource and Resource Descriptor

At the heart of a green context is a device resource (`cudaDevResource`) tied to a specific GPU device. Resources can be combined and encapsulated into a descriptor (`cudaDevResourceDesc_t`). A green context only has access to the resources encapsulated into the descriptor used for its creation.

Currently the `cudaDevResource` data structure is defined as:

```
struct {
     enum cudaDevResourceType type;
     union {
         struct cudaDevSmResource sm;
         struct cudaDevWorkqueueConfigResource wqConfig;
         struct cudaDevWorkqueueResource wq;
     };
 };
```

The supported valid resource types are `cudaDevResourceTypeSm`, `cudaDevResourceTypeWorkqueueConfig` and `cudaDevResourceTypeWorkqueue`, while `cudaDevResourceTypeInvalid` identifies an invalid resource type.

A valid device resource can be associated with:

- a specific set of streaming multiprocessors (SMs) (resource type `cudaDevResourceTypeSm`),
- a specific workqueue configuration (resource type `cudaDevResourceTypeWorkqueueConfig`) or
- a pre-existing workqueue resource (resource type `cudaDevResourceTypeWorkqueue`).

One can query if a given execution context or CUDA stream is associated with a `cudaDevResource` resource of a given type, using the `cudaExecutionCtxGetDevResource` and `cudaStreamGetDevResource` APIs respectively. Being associated with different types of device resources (e.g., SMs and work queues) is also possible for an execution context, while a stream can only be associated with an SM-type resource.

A given GPU device has, by default, all three device resource types: an SM-type resource encompassing all the SMs of the GPU, a workqueue configuration resource encompassing all available work queues and its corresponding workqueue resource. These resources can be retrieved via the `cudaDeviceGetDevResource` API.

Overview of relevant device resource structs

The different resource type structs have fields that are set either explicitly by the user or by a relevant CUDA API call. It is recommended to zero-initialize all device resource structs.

- An SM-type device resource (`cudaDevSmResource`) has the following relevant fields:
  - `unsigned int smCount`: number of SMs available in this resource
  - `unsigned int minSmPartitionSize`: minimum SM count required to partition this resource
  - `unsigned int smCoscheduledAlignment`: number of SMs in the resource guaranteed to be co-scheduled on the same GPU processing cluster, which is relevant for thread block clusters. `smCount` is a multiple of this value when `flags` is zero.
  - `unsigned int flags`: supported flags are 0 (default) and `cudaDevSmResourceGroupBackfill` (see `cudaDevSmResourceGroup` flags).

  The above fields will be set via either the appropriate split API (`cudaDevSmResourceSplitByCount` or `cudaDevSmResourceSplit`) used to create this SM-type resource or will be populated by the `cudaDeviceGetDevResource` API which retrieves the SM resources of a given GPU device. These fields should never be set directly by the user. See next section for more details.
- A workqueue configuration device resource (`cudaDevWorkqueueConfigResource`) has the following relevant fields:
  - `int device`: the device on which the workqueue resources are available
  - `unsigned int wqConcurrencyLimit`: the number of stream-ordered workloads expected to avoid false dependencies
  - `enum cudaDevWorkqueueConfigScope sharingScope`: the sharing scope for the workqueue resources. Supported values are: `cudaDevWorkqueueConfigScopeDeviceCtx` (default) and `cudaDevWorkqueueConfigScopeGreenCtxBalanced`. With the default option, all workqueue resources are shared across all contexts, while with the balanced option the driver tries to use non-overlapping workqueue resources across green contexts wherever possible, using the user-specified `wqConcurrencyLimit` as a hint.

  These fields need to be set by the user. There is no CUDA API similar to the split APIs that generates a workqueue configuration resource, with the exception of the workqueue configuration resource populated by the `cudaDeviceGetDevResource` API. That API can retrieve the workqueue configuration resources of a given GPU device.
- Finally, a pre-existing workqueue resource (`cudaDevResourceTypeWorkqueue`) has no fields that can be set by the user. As with the other resource types, `cudaDevGetDevResource` can retrieve the pre-existing workqueue resource of a given GPU device.

4.6.4. Green Context Creation Example

There are four main steps involved in green context creation:

- Step 1: Start with an initial set of resources, e.g., by fetching the available resources of the GPU

- Step 2: Partition the SM resources into one or more partitions (using one of the available split APIs).
- Step 3: Create a resource descriptor combining, if needed, different resources
- Step 4: Create a green context from the descriptor, provisioning its resources

After the green context has been created, you can create CUDA streams belonging to that green context. GPU work subsequently launched on such a stream, such as a kernel launched via `<<< >>>`, will only have access to this green context's provisioned resources. Libraries can also easily leverage green contexts, as long as the user passes a stream belonging to a green context to them. See *Green Contexts - Launching work* for more details.

4.6.4.1 Step 1: Get available GPU resources

The first step in green context creation is to get the available device resources and populate the `cudaDevResource` struct(s). There are currently three possible starting points: a device, an execution context or a CUDA stream.

The relevant CUDA runtime API function signatures are listed below:

- For a **device**: `cudaError_t cudaDeviceGetDevResource(int device, cudaDevResource* resource, cudaDevResourceType type)`
- For an **execution context**: `cudaError_t cudaExecutionCtxGetDevResource(cudaExecutionContext_t ctx, cudaDevResource* resource, cudaDevResourceType type)`
- For a **stream**: `cudaError_t cudaStreamGetDevResource(cudaStream_t hStream, cudaDevResource* resource, cudaDevResourceType type)`

All valid `cudaDevResourceType` types are permitted for each of these APIs, with the exception of `cudaStreamGetDevResource` which only supports an SM-type resource.

Usually, the starting point will be a GPU device. The code snippet below shows how to get the available SM resources of a given GPU device. After a successful `cudaDeviceGetDevResource` call, the user can review the number of SMs available in this resource.

```
int current_device = 0; // assume device ordinal of 0
CUDA_CHECK(cudaSetDevice(current_device));

cudaDevResource initial_SM_resources = {};
CUDA_CHECK(cudaDeviceGetDevResource(current_device /* GPU device */,
                                    &initial_SM_resources /* device resource
↪to populate */,
                                    cudaDevResourceTypeSm /* resource type*/));

std::cout << "Initial SM resources: " << initial_SM_resources.sm.smCount << "
↪SMs" << std::endl; // number of available SMs

// Special fields relevant for partitioning (see Step 3 below)
std::cout << "Min. SM partition size: " <<  initial_SM_resources.sm.
↪minSmPartitionSize << " SMs" << std::endl;
std::cout << "SM co-scheduled alignment: " <<  initial_SM_resources.sm.
↪smCoscheduledAlignment << " SMs" << std::endl;
```

One can also get the available workqueue config. resources, as shown in the code snippet below.

```
int current_device = 0; // assume device ordinal of 0
CUDA_CHECK(cudaSetDevice(current_device));

cudaDevResource initial_WQ_config_resources = {};
CUDA_CHECK(cudaDeviceGetDevResource(current_device /* GPU device */,
                                    &initial_WQ_config_resources /* device
↪resource to populate */,
                                    cudaDevResourceTypeWorkqueueConfig /*
↪resource type*/));

std::cout << "Initial WQ config. resources: " << std::endl;
std::cout << "  - WQ concurrency limit: " << initial_WQ_config_resources.
↪wqConfig.wqConcurrencyLimit << std::endl;
std::cout << "  - WQ sharing scope: " << initial_WQ_config_resources.wqConfig.
↪sharingScope << std::endl;
```

After a successful cudaDeviceGetDevResource call, the user can review the wqConcurrencyLimit for this resource. When the starting point is a GPU device, the wqConcurrencyLimit will match the value of CUDA_DEVICE_MAX_CONNECTIONS environment variable or its default value.

4.6.4.2 Step 2: Partition SM resources

The second step in green context creation is to statically split the available cudaDevResource SM resources into one or more partitions, with potentially some SMs left over in a *remaining* partition. This partitioning is possible using the cudaDevSmResourceSplitByCount() or the cudaDevSmResourceSplit() API. The cudaDevSmResourceSplitByCount() API can only create one or more *homogeneous* partitions, plus a potential *remaining* partition, while the cudaDevSmResourceSplit() API can also create *heterogeneous* partitions, plus the potential *remaining* one. The subsequent sections describe the functionality of both APIs in detail. Both APIs are only applicable to SM-type device resources.

cudaDevSmResourceSplitByCount API

The cudaDevSmResourceSplitByCount runtime API signature is:

cudaError_t cudaDevSmResourceSplitByCount(cudaDevResource* result, unsigned int* nbGroups, const cudaDevResource* input, cudaDevResource* remaining, unsigned int useFlags, unsigned int minCount)

As Figure 43 highlights, the user requests to split the input SM-type device resource into *nbGroups homogeneous groups with minCount SMs each. However, the end result will contain a potentially updated *nbGroups number of homogeneous groups with N SMs each. The potentially updated *nbGroups will be less than or equal to the originally requested group number, while N will be equal to or greater than minCount. These adjustments may occur due to some granularity and alignment requirements, which are architecture specific.

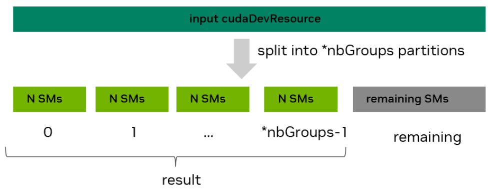


Figure 43: SM resource split using the cudaDevSmResourceSplitByCount API

Table 30 lists the minimum SM partition size and the SM co-scheduled alignment for all the currently supported compute capabilities, for the default `useFlags=0` case. One can also retrieve these values via the `minSmPartitionSize` and `smCoscheduledAlignment` fields of `cudaDevSmResource`, as shown in *Step 1: Get available GPU resources*. Some of these requirements can be lowered via a different `useFlags` value. Table 14 provides some relevant examples highlighting the difference between what is requested and the final result, along with an explanation. The table focuses on compute capability (CC 9.0), where the minimum number of SMs per partition is 8 and the SM count has to be a multiple of 8, if `useFlags` is zero.

Table 14: Split functionality

Re-queste			Actual (for GH200 with 132 SMs)		
*nbGro	min-Count	useFlags	*nbGroups with N SMs	Remaining SMs	Reason
2	72	0	1 group of 72 SMs	60	cannot exceed 132 SMs
6	11	0	6 groups of 16 SMs	36	multiple of 8 requirement
6	11	CU_DEV_SM_RESOURCE_SPLIT_IG	6 groups with 12 SMs each	60	lowered to multiple of 2 req.
2	1	0	2 groups with 8 SMs each	116	min. 8 SMs requirement

Here is a code snippet requesting to split the available SM resources into *five groups* of *8 SMs* each:

```
cudaDevResource avail_resources = {};
// Code that has populated avail_resources not shown

unsigned int min_SM_count = 8;
unsigned int actual_split_groups = 5; // may be updated

cudaDevResource actual_split_result[5] = {{}, {}, {}, {}, {}};
cudaDevResource remaining_partition = {};

CUDA_CHECK(cudaDevSmResourceSplitByCount(&actual_split_result[0],
                                          &actual_split_groups,
                                          &avail_resources,
                                          &remaining_partition,
                                          0 /*useFlags */,
                                          min_SM_count));

std::cout << "Split " << avail_resources.sm.smCount << " SMs into " << actual_
↪split_groups << " groups " \
          << "with " << actual_split_result[0].sm.smCount << " each " \
          << "and a remaining group with " << remaining_partition.sm.smCount <
↪< " SMs" << std::endl;
```

Be aware that:

- one could use `result=nullptr` to query the number of groups that would be created

- one could set `remaining=nullptr`, if one does not care for the SMs of the remaining partition
- the *remaining* (leftover) partition does not have the same functional or performance guarantees as the homogeneous groups in *result*.
- `useFlags` is expected to be 0 in the default case, but values of `cudaDevSmResourceSplitIgnoreSmCoscheduling` and `cudaDevSmResourceSplitMaxPotentialClusterSize` are also supported
- any resulting `cudaDevResource` cannot be repartitioned without first creating a resource descriptor and a green context from it (i.e., steps 3 and 4 below)

Please refer to cudaDevSmResourceSplitByCount runtime API reference for more details.

cudaDevSmResourceSplit API

As mentioned earlier, a single `cudaDevSmResourceSplitByCount` API call can only create homogeneous partitions, i.e., partitions with the same number of SMs, plus the remaining partition. This can be limiting for heterogeneous workloads, where work running on different green contexts has different SM count requirements. To achieve heterogeneous partitions with the split-by-count API, one would usually need to re-partition an existing resource by repeating Steps 1-4 (multiple times). Or, in some cases, one may be able to create homogeneous partitions each with SM count equal to the GCD (greatest common divisor) of all the heterogeneous partitions as part of step-2 and then merge the required number of them together as part of step-3. This last approach however is not recommended, as the CUDA driver may be able to create better partitions if larger sizes were requested up front.

The `cudaDevSmResourceSplit` API aims to address these limitations by allowing the user to create non-overlapping heterogeneous partitions in a single call. The `cudaDevSmResourceSplit` runtime API signature is:

`cudaError_t cudaDevSmResourceSplit(cudaDevResource* result, unsigned int nbGroups, const cudaDevResource* input, cudaDevResource* remainder, unsigned int flags, cudaDevSmResourceGroupParams* groupParams)`

This API will attempt to partition the `input` SM-type resource into `nbGroups` valid device resources (groups) placed in the `result` array based on the requirements specified for each one in the `groupParams` array. An optional remaining partition may also be created. In a successful split, as shown in Figure 44, each resource in the `result` can have a different number of SMs, but never zero SMs.

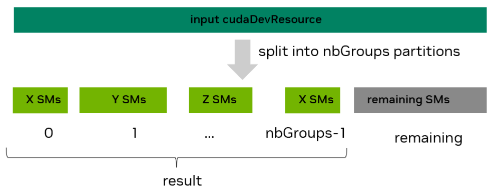


Figure 44: SM resource split using the cudaDevSmResourceSplit API

When requesting a heterogeneous split, one needs to specify the SM count (`smCount` field of relevant `groupParams` entry) for each resource in `result`. This SM count should always be a multiple of two. For the scenario in the previous image, `groupParams[0].smCount` would be X, `groupParams[1].smCount` Y, etc. However, just specifying the SM count is not sufficient, if an application uses *Thread Block Clusters*. Since all the thread blocks of a cluster are guaranteed to be co-scheduled, the user also needs to specify the maximum supported cluster size, if any, a given resource group should support. This is possible via the `coscheduledSmCount` field of the relevant `groupParams` entry. For GPUs with compute capability 10.0 and on (CC 10.0+), clusters can also have a preferred dimension, which is a multiple of their default cluster dimension. During a single kernel launch on supported systems, this larger preferred cluster dimension is used as much as possible, if at all, and the smaller default cluster dimension is used otherwise. The user can express this preferred cluster dimension hint via the `preferredCoscheduledSmCount` field of the relevant `groupParams` entry. Finally, there may be cases where the user may want to loosen the SM count requirements and pull in more available SMs in a given group; the user can express this backfill option by setting the `flags` field of the relevant `groupParams` entry to its non-default flag value.

To provide more flexibility, the `cudaDevSmResourceSplit` API also has a **discovery** mode, to be used when the exact SM count, for one or more groups, is not known ahead of time. For example, a user may want to create a device resource that has as many SMs as possible, while meeting some co-scheduling requirements (e.g., allowing clusters of size four). To exercise this discovery mode, the user can set the `smCount` field of the relevant `groupParams` entry (or entries) to zero. After a successful `cudaDevSmResourceSplit` call, the `smCount` field of the `groupParams` will have been populated with a valid non-zero value; we refer to this as the **actual** `smCount` value. If `result` was not null (so this was not a dry run), then the relevant group of `result` will also have its `smCount` set to the same value. The order the `nbGroups` `groupParams` entries are specified matters, as they are evaluated from left (index 0) to right (index nbGroups-1).

Table 15 provides a high level view of the supported arguments for the `cudaDevSmResourceSplit` API.

Table 15: Overview of cudaDevSmResourceSplit split API

					groupParams array; showing entry i with i [0, nbGroups)			
result	**nbGr**	**input**	**remainder**	**flags**	**smCount**	**coscheduledSmCount**	**preferredCoscheduledSmCount**	**flags**
nullptr for explorative dry run; not null ptr otherwise	number of group	resource to split into nbGroups groups	nullptr if you do not want a remainder group	0	0 for discovery mode or other valid smCount	0 (default) or valid coscheduled SM count	0 (default) or valid preferred coscheduled SM count (hint)	0 (default) or cudaDevSmResourceGroupBackfill

Notes:

1) `cudaDevSmResourceSplit` API's return value depends on `result`:

- `result != nullptr`: the API will return `cudaSuccess` only when the split is successful and `nbGroups` valid `cudaDevResource` groups, meeting the specified requirements were created; otherwise, it will return an error. As different types of errors may return the same error code (e.g., `CUDA_ERROR_INVALID_RESOURCE_CONFIGURATION`), it is recommended to use the `CUDA_LOG_FILE` environment variable to get more informative error descriptions during development.
- `result == nullptr`: the API may return `cudaSuccess` even if the resulting `smCount` of a group is zero, a case which would have returned an error with a non-nullptr `result`. Think of this mode as a dry-run test you can use while exploring what is supported, especially in *discovery* mode.

2) On a successful call with result != nullptr, the resulting `result[i]` device resource with i in `[0, nbGroups)` will be of type `cudaDevResourceTypeSm` and have a `result[i].sm.smCount` that will either be the non-zero user-specified `groupParams[i].smCount` value or the discovered one. In both cases, the `result[i].sm.smCount` will meet all the following constraints:

- be a `multiple of 2` and
- be in the `[2, input.sm.smCount]` range and
- `(flags == 0) ? (multiple of actual group_params[i].coscheduledSmCount) : (>= groups_params[i].coscheduledSmCount)`

3) Specifying zero for any of the `coscheduledSmCount` and `preferredCoscheduledSmCount` fields indicates that the default values for these fields should be used; these can vary per GPU. These default values are both equal to the `smCoscheduledAlignment` of the SM resource retrieved via the `cudaDeviceGetDevResource` API for the given device (and not any SM resource). To review these default values, one can examine their updated values in the relevant `groupParams` entry after a successful `cudaDevSmResourceSplit` call with them initially set to 0; see below.

```
int gpu_device_index = 0;
cudaDevResource initial_GPU_SM_resources {};
CUDA_CHECK(cudaDeviceGetDevResource(gpu_device_index, &initial_GPU_
↪SM_resources, cudaDevResourceTypeSm));
std::cout << "Default value will be equal to " << initial_GPU_SM_
↪resources.sm.smCoscheduledAlignment << std::endl;

int default_split_flags = 0;
cudaDevSmResourceGroupParams group_params_tmp = {.smCount=0, .
↪coscheduledSmCount=0, .preferredCoscheduledSmCount=0, .flags=0};
CUDA_CHECK(cudaDevSmResourceSplit(nullptr, 1, &initial_GPU_SM_
↪resources, nullptr /*remainder*/, default_split_flags, &group_
↪params_tmp));
std::cout << "coscheduledSmcount default value: " << group_params.
↪coscheduledSmCount << std::endl;
std::cout << "preferredCoscheduledSmcount default value: " << group_
↪params.preferredCoscheduledSmCount << std::endl;
```

4) The remainder group, if present, will not have any constraints on its SM count or co-scheduling requirements. It will be up to the user to explore that.

Before providing more detailed information for the various `cudaDevSmResourceGroupParams` fields, Table 16 shows what these values could be for some example use cases. Assume an `initial_GPU_SM_resources` device resource has already been populated, as in the previous code snippet, and is the resource that will be split. Every row in the table will have that same starting point. For simplicity the table will only show the `nbGroups` value and the `groupParams` fields per use case that can be used in a code snippet like the one below.

```
int nbGroups = 2; // update as needed
unsigned int default_split_flags = 0;
cudaDevResource remainder {}; // update as needed
cudaDevResource result_use_case[2] = {{}, {}}; // Update depending on number
↪of groups planned. Increase size if you plan to also use a workqueue resource
cudaDevSmResourceGroupParams group_params_use_case[2] = {{.smCount = X, .
↪coscheduledSmCount=0, .preferredCoscheduledSmCount = 0, .flags = 0},
                                                          {.smCount = Y, .
↪coscheduledSmCount=0, .preferredCoscheduledSmCount = 0, .flags = 0}}
CUDA_CHECK(cudaDevSmResourceSplit(&result_use_case[0], nbGroups, &initial_GPU_
↪SM_resources, remainder, default_split_flags, &group_params_use_case[0]));
```

Table 16: split API use cases

#	Goal/Use Cases	nbGr	remainder	groupParams[i] fields (i shown in ascending order; see last column)				i
				smCount	coscheduledSmCount	preferredCoscheduledSmCount	flags	
1	Resource with 16 SMs. Do not care for remaining SMs. May use clusters.	1	nullp	16	0	0	0	0
2a	One resource with 16 SMs and one with everything else. Will not use clusters. (Note: showing two options: in option (2a),the 2nd resource is the remainder; in option (2b), it is the result_use_case[1].)	1 (2a)	not nullp	16	2	2	0	0
2b		2 (2b)	nullp	16	2	2	0	0
				0	2	2	cudaDevSmResourceGroupBackfill	1
3	Two resources with 28 and 32 SMs respectively. Will use clusters of size 4.	2	nullp	28	4	4	0	0
				32	4	4	0	1
4	One resource with as many SMs as possible, which can run clusters of size 8, and one remainder.	1	not nullp	0	8	8	0	0
5	One resource with as many SMs as possible, which can run clusters of size 4, and one with 8 SMs. (Note: Order matters! Changing order of entries in groupParams array could mean no SMs left for the 8-SM group)	2	nullp	8	2	2	0	0
				0	4	4	0	1

Detailed information about the various cudaDevSmResourceGroupParams struct fields

`smCount`:

- Controls SM count for the corresponding group in result.
- **Values**: 0 (discovery mode) or valid non-zero value (non-discovery mode)

  - Valid non-zero `smCount` value requirements: `(multiple of 2) and in [2, input->sm.smCount] and ((flags == 0) ? multiple of actual coscheduledSmCount : greater than or equal to coscheduledSmCount)`
- **Use cases**: use discovery mode to explore what's possible when SM count is not known/fixed; use non-discovery mode to request a specific number of SMs.
- Note: in discovery mode, actual SM count, after successful split call with non-nullptr result, will meet valid non-zero value requirements

`coscheduledSmCount`:

- Controls number of SMs grouped together ("co-scheduled") to enable launch of different clusters on compute capability 9.0+. It can thus impact the number of SMs in a resulting group and the cluster sizes they can support.
- **Values**: 0 (default for current architecture) or valid non-zero value
  - Valid non-zero value requirements: `(multiple of 2)` up to max limit
- **Use cases**: Use default or a manually chosen value for clusters, keeping in mind the max. portable cluster size on a given architecture. If your code does not use clusters, you can use the minimum supported value of 2 or the default value.
- Note: when the default value is used, the actual `coscheduledSmCount`, after a successful split call, will also meet valid non-zero value requirements. If flags is not zero, the resulting smCount will be >= coscheduledSmCount. Think of coscheduledSmCount as providing some guaranteed underlying "structure" to valid resulting groups (i.e., that group can run at least a single cluster of coscheduledSmCount size in the worst case). This type of structure guarantee does not apply to the remaining group; there it is up to the user to explore what cluster sizes can be launched.

`preferredCoscheduledSmCount`:

- Acts as a hint to the driver to try to merge groups of actual `coscheduledSmCount` SMs into larger groups of `preferredCoscheduledSmCount` if possible. Doing so can allow code to make use of preferred cluster dimensions feature available on devices with compute capability (CC) 10.0 and on). See cudaLaunchAttributeValue::preferredClusterDim.
- **Values**: 0 (default for current architecture) or valid non-zero value
  - Valid non-zero value requirements: `(multiple of actual coscheduledSmCount)`
- **Use cases**: use a manually chosen value greater than 2 if you use preferred clusters and are on a device of compute capability 10.0 (Blackwell) or later. If you don't use clusters, choose the same value as `coscheduledSmCount`: either select the minimum supported value of 2 or use 0 for both
- Note: when the default value is used, the actual `preferredCoscheduledSmCount`, after a successful split call, will also meet valid non-zero value requirement.

`flags`:

- Controls if the resulting SM count of a group will be a multiple of actual coscheduled SM count (default) or if SMs can be backfilled into this group (backfill). In the backfill case, the resulting SM count (`result[i].sm.smCount`) will be greater than or equal to the specified `groupParams[i].smCount`.
- **Values**: 0 (default) or `cudaDevSmResourceGroupBackfill`
- **Use cases**: Use the zero (default), so the resulting group has the guaranteed flexibility of supporting multiple clusters of coScheduledSmCount size. Use the backfill option, if you want to get as many SMs as possible in the group, with some of these SMs (the backfilled ones), not providing any coscheduling guarantee.

- Note: a group created with the backfill flag can still support clusters (e.g., it is guaranteed to support at least one coscheduledSmCount size).

4.6.4.3 Step 2 (continued): Add workqueue resources

If you also want to specify a workqueue resource, then this needs to be done explicitly. The following example shows how to create a workqueue configuration resource for a specific device with balanced sharing scope and a concurrency limit of four.

```
cudaDevResource split_result[2] = {{}, {}};
// code to populate split_result[0] not shown; used split API with nbGroups=1

// The last resource will be a workqueue resource.
split_result[1].type = cudaDevResourceTypeWorkqueueConfig;
split_result[1].wqConfig.device = 0; // assume device ordinal of 0
split_result[1].wqConfig.sharingScope =
↪cudaDevWorkqueueConfigScopeGreenCtxBalanced;
split_result[1].wqConfig.wqConcurrencyLimit = 4;
```

A workqueue concurrency limit of four hints to the driver that the user expects maximum four concurrent stream-ordered workloads. The driver will assign work queues trying to respect this hint, if possible.

4.6.4.4 Step 3: Create a Resource Descriptor

The next step, after resources have been split, is to generate a resource descriptor, using the `cudaDevResourceGenerateDesc` API, for all the resources expected to be available to a green context.

The relevant CUDA runtime API function signature is:

`cudaError_t cudaDevResourceGenerateDesc(cudaDevResourceDesc_t *phDesc, cudaDevResource *resources, unsigned int nbResources)`

It is possible to combine multiple `cudaDevResource` resources. For example, the code snippet below shows how to generate a resource descriptor that encapsulates three groups of resources. You just need to ensure that these resources are all allocated continuously in the `resources` array.

```
cudaDevResource actual_split_result[5] = {};
// code to populate actual_split_result not shown

// Generate resource desc. to encapsulate 3 resources: actual_split_result[2]
↪to [4]
cudaDevResourceDesc_t resource_desc;
CUDA_CHECK(cudaDevResourceGenerateDesc(&resource_desc, &actual_split_
↪result[2], 3));
```

Combining different types of resources is also supported. For example, one could generate a descriptor with both SM and workqueue resources.

For a `cudaDevResourceGenerateDesc` call to be successful:

- All `nbResources` resources should belong to the same GPU device.
- If multiple SM-type resources are combined, they should be generated from the same split API call and have the same `coscheduledSmCount` values (if not part of remainder)
- Only a single workqueue config or workqueue type resource may be present.

4.6.4.5 Step 4: Create a Green Context

The final step is to create a green context from a resource descriptor using the `cudaGreenCtxCreate` API. That green context will only have access to the resources (e.g., SMs, work queues) encapsulated in the resource descriptor specified during its creation. These resources will be provisioned during this step.

The relevant CUDA runtime API function signature is:

`cudaError_t cudaGreenCtxCreate(cudaExecutionContext_t *phCtx, cudaDevResourceDesc_t desc, int device, unsigned int flags)`

The `flags` parameter should be set to 0. It is also recommended to explicitly initialize the device's primary context before creating a green context via either the `cudaInitDevice` API or the `cudaSetDevice` API, which also sets the primary context as current to the calling thread. Doing so ensures there will be no additional primary context initialization overhead during green context creation.

See code snippet below.

```
int current_device = 0; // assume single GPU
CUDA_CHECK(cudaSetDevice(current_device)); // Or cudaInitDevice

cudaDevResourceDesc_t resource_desc {};
// Code to generate resource_desc not shown

// Create a green_ctx on GPU with current_device ID with access to resources
↪from resource_desc
cudaExecutionContext_t green_ctx {};
CUDA_CHECK(cudaGreenCtxCreate(&green_ctx, resource_desc, current_device, 0));
```

After a successful green context creation, the user can verify its resources by calling `cudaExecutionCtxGetDevResource` on that execution context for each resource type.

Creating Multiple Green Contexts

An application can have more than one green context, in which case some of the steps above should be repeated. For most use cases, these green contexts will each have a separate non-overlapping set of provisioned SMs. For example, for the case of five homogeneous `cudaDevResource` groups (`actual_split_result` array), one green context's descriptor may encapsulate actual_split_result[2] to [4] resources, while the descriptor of another green context may encapsulate actual_split_result[0] to [1]. In this case, a specific SM will be provisioned for only one of the two green contexts of the application.

But SM oversubscription is also possible and may be used in some cases. For example, it may be acceptable to have the second green context's descriptor encapsulate actual_split_result[0] to [2]. In this case, all the SMs of actual_split_resource[2] `cudaDevResource` will be oversubscribed, i.e., provisioned for both green contexts, while resources actual_split_resource[0] to [1] and actual_split_resource[3] to [4] may only be used by one of the two green contexts. SM oversubscription should be judiciously used on a per-case basis.

4.6.5. Green Contexts - Launching work

To launch a kernel targeting a green context created using the prior steps, you first need to create a stream for that green context with the `cudaExecutionCtxStreamCreate` API. Launching a kernel on that stream using `<<< >>>` or the `cudaLaunchKernel` API, will ensure that kernel can only use the resources (SMs, work queues) available to that stream via its execution context. For example:

```
// Create green_ctx_stream CUDA stream for previously created green_ctx green
↪context
cudaStream_t green_ctx_stream;
int priority = 0;
CUDA_CHECK(cudaExecutionCtxStreamCreate(&green_ctx_stream,
                                        green_ctx,
                                        cudaStreamDefault,
                                        priority));

// Kernel my_kernel will only use the resources (SMs, work queues, as
↪applicable) available to green_ctx_stream's execution context
my_kernel<<<grid_dim, block_dim, 0, green_ctx_stream>>>();
CUDA_CHECK(cudaGetLastError());
```

The default stream creation flag passed to the stream creation API above is equivalent to `cudaStreamNonBlocking` given `green_ctx` is a green context.

CUDA graphs

For kernels launched as part of a CUDA graph (see *CUDA Graphs*), there are a few more subtleties. Unlike kernels, the CUDA stream a CUDA graph is launched on does **not** determine the SM resources used, as that stream is solely used for dependency tracking.

The execution context a kernel node (and other applicable node types) will execute on is set during node creation. If the CUDA graph will be created using stream capture, then the execution context(s) of the stream(s) involved in the capture will determine the execution context(s) of the relevant graph nodes. If the graph will be created using the graph APIs, then the user should explicitly set the execution context for each relevant node. For example, to add a kernel node, the user should use the polymorphic `cudaGraphAddNode` API with `cudaGraphNodeTypeKernel` type and explicitly specify the `.ctx` field of the `cudaKernelNodeParamsV2` struct under `.kernel`. The `cudaGraphAddKernelNode` does not allow the user to specify an execution context and should thus be avoided. Please note that it is possible for different graph nodes in a graph to belong to different execution contexts.

For verification purposes, one could use Nsight Systems in node tracing mode (`--cuda-graph-trace node`) to observe the green context(s) specific graph nodes will execute on. Note that in the default *graph* tracing mode, the entire graph will appear under the green context of the stream it was launched on, but, as previously explained, this does not provide any information about the execution context(s) of the various graph nodes.

To verify programmatically, one could potentially use the CUDA driver API `cuGraphKernelNodeGetParams(graph_node, &node_params)` and compare the `node_params.ctx` context handle field with the expected context handle for that graph node. Using the driver API is possible given `CUgraphNode` and `cudaGraphNode_t` can be used interchangeably, but the user would need to include the relevant `cuda.h` header and link with the driver directly (`-lcuda`).

Thread Block Clusters

Kernels with thread block clusters (see Section 1.2.2.1.1) can also be launched on a green context stream, like any other kernel, and thus use that green context's provisioned resources. Section 4.6.4.2 showed how to specify the number of SMs that need to be coscheduled when a device resource is split, to facilitate clusters. But as with any kernel using clusters, the user should make use of the relevant occupancy APIs to determine the max potential cluster size for a kernel (via `cudaOccupancyMaxPotentialClusterSize`) and, if needed, the maximum number of active clusters (`cudaOccupancyMaxActiveClusters`). If the user specifies a green context stream as the `stream` field of the relevant `cudaLaunchConfig`, then these occupancy APIs will take into consideration the SM resources provisioned for that green context. This use case is especially relevant for libraries that may get a green context CUDA stream passed to them by the user, as well as in cases where the green context was created from a remaining device resource.

The code snippet below shows how these APIs can be used.

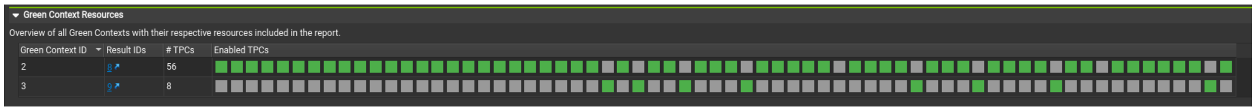
```
// Assume cudaStream_t gc_stream  has already been created and a __global__
↪void cluster_kernel exists.

// Uncomment to support non portable cluster size, if possible
// CUDA_CHECK(cudaFuncSetAttribute(cluster_kernel,
↪cudaFuncAttributeNonPortableClusterSizeAllowed, 1))

cudaLaunchConfig_t config = {0};
config.gridDim          = grid_dim; // has to be a multiple of cluster dim.
config.blockDim         = block_dim;
config.dynamicSmemBytes = expected_dynamic_shared_mem;

cudaLaunchAttribute attribute[1];
attribute[0].id = cudaLaunchAttributeClusterDimension;
attribute[0].val.clusterDim.x = 1;
attribute[0].val.clusterDim.y = 1;
attribute[0].val.clusterDim.z = 1;
config.attrs = attribute;
config.numAttrs = 1;

config.stream=gc_stream; // Need to pass the CUDA stream that will be used for
↪that kernel

int max_potential_cluster_size = 0;
// the next call will ignore cluster dims in launch config
CUDA_CHECK(cudaOccupancyMaxPotentialClusterSize(&max_potential_cluster_size,
↪cluster_kernel, &config));
std::cout << "max potential cluster size is " << max_potential_cluster_size <
↪< " for CUDA stream gc_stream" << std::endl;

// Could choose to update launch config's clusterDim with max_potential_cluster_
↪size.
// Doing so would result in a successful cudaLaunchKernelEx call for the same
↪kernel and launch config.

int num_clusters= 0;
CUDA_CHECK(cudaOccupancyMaxActiveClusters(&num_clusters, cluster_kernel, &
↪config));
std::cout << "Potential max. active clusters count is " << num_clusters <<
↪std::endl;
```

Verify Green Contexts Use

Beyond empirical observations of affected kernel execution times due to green context provisioning, the user can leverage Nsight Systems or Nsight Compute CUDA developer tools to verify, to some extent, correct green contexts use.

For example, kernels launched on CUDA streams belonging to different green contexts will appear under different *Green Context* rows under the CUDA HW timeline section of an *Nsight Systems* report. *Nsight Compute* provides a *Green Context Resources* overview in its Session page as well as updated *# SMs* under the *Launch Statistics* of the *Details* section. The former provides a visual bitmask of provisioned resources. This is particularly useful if an application uses different green contexts, as the user can confirm expected overlap across GCs (no overlap or expected non-zero overlap if SMs are oversubscribed).

Figure 45 depicts these resources for an example with two green contexts provisioned with 112 and 16 SMs respectively, with no SM overlap across them. The provided view can help the user verify the provisioned SM resource count per green context. It also helps confirm that no SMs were oversubscribed, as no box is marked green (provisioned for that GC) across both green contexts.



Overview of all Green Contexts with their respective resources included in the report.

Green Context ID	Result IDs	# TPCs	Enabled TPCs
2	8	56	🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩⬜🟩⬜🟩🟩⬜🟩🟩🟩⬜🟩🟩🟩🟩🟩⬜🟩🟩🟩🟩⬜🟩🟩🟩⬜🟩🟩🟩🟩⬜🟩🟩⬜🟩🟩🟩🟩🟩🟩⬜🟩
3	9	8	⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜🟩⬜🟩⬜⬜🟩⬜⬜⬜🟩⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜🟩⬜⬜⬜🟩⬜⬜⬜⬜🟩⬜⬜⬜⬜⬜⬜⬜⬜⬜🟩⬜

Figure 45: Green contexts resources section from Nsight Compute

The *Launch Statistics* section also explicitly lists the number of SMs provisioned for this green context, which can thus be used by this kernel. Please note that these are the SMs a given kernel can have access to during its execution, and not the actual number of SMs that kernel ran on. The same applies to the resources overview shown earlier. The actual number of SMs used by the kernel can depend on various factors, including the kernel itself (launch geometry, etc.), other work running at the same time on the GPU, etc.

4.6.6. Additional Execution Contexts APIs

This section touches upon some additional green context APIs. For a complete list, please refer to the relevant CUDA runtime API section.

For synchronization using CUDA events, one can leverage the `cudaError_t cudaExecutionCtxRecordEvent(cudaExecutionContext_t ctx, cudaEvent_t event)` and `cudaError_t cudaExecutionCtxWaitEvent(cudaExecutionCtxWaitEvent(cudaExecutionContext_t ctx, cudaEvent_t event)` APIs. `cudaExecutionCtxRecordEvent` records a CUDA event capturing all work/activities of the specified execution context at the time of this call, while `cudaExecutionCtxWaitEvent` makes all future work submitted to the execution context wait for the work captured in the specified event.

Using `cudaExecutionCtxRecordEvent` is more convenient than `cudaEventRecord` if the execution context has multiple CUDA streams. To achieve equivalent behavior without this execution context API, one would need to record a separate CUDA event via `cudaEventRecord` on every execution context stream and then have dependent work wait separately for all these events. Similarly, `cudaExecutionCtxWaitEvent` is more convenient than `cudaStreamWaitEvent`, if one needs all execution context streams to wait for an event to complete. The alternative would be a separate `cudaStreamWaitEvent` for every stream in this execution context.

For blocking synchronization on the CPU side, one can use `cudaError_t cudaExecutionCtxSynchronize(cudaExecutionContext_t ctx)`. This call will block until the specified execution context has completed all its work. If the specified execution context was not created via `cudaGreenCtxCreate`, but was rather obtained via `cudaDeviceGetExecutionCtx`, and is thus the device's primary context, calling that function will also synchronize all green contexts that have been created on the same device.

To retrieve the device a given execution context is associated with, one can use `cudaExecutionCtxGetDevice`. To retrieve the unique identifier of a given execution context, one can use `cudaExecutionCtxGetId`.

Finally, an explicitly created execution context can be destroyed via the `cudaError_t cudaExecutionCtxDestroy(cudaExecutionContext_t ctx)` API.

4.6.7. Green Contexts Example

This section illustrates how green contexts can enable critical work to start and complete sooner. Similar to the scenario used in Section 4.6.1, the application has two kernels that will run on two different non-blocking CUDA streams. The timeline, from the CPU side, is as follows. A long running kernel (*delay_kernel_us*), which takes multiple waves on the full GPU, is launched first on CUDA stream *strm1*. Then after a brief wait time (less than the kernel duration), a shorter but critical kernel (*critical_kernel*) is launched on stream *strm2*. The GPU durations and time from CPU launch to completion for both kernels are measured.

As a proxy for a long running kernel, a delay kernel is used where every thread block runs for a fixed number of microseconds and the number of thread blocks exceeds the GPU's available SMs.

Initially, no green contexts are used, but the critical kernel is launched on a CUDA stream with a higher priority than the long running kernel. Because of its high priority stream, the critical kernel can start executing as soon as some of the thread blocks of the long running kernel complete. However, it will still need to wait for some potentially long-running thread blocks to complete, which will delay its execution start.

Figure 46 shows this scenario in an Nsight Systems report. The long running kernel is launched on stream 13, while the short but critical kernel is launched on stream 14, which has higher stream priority. As highlighted on the image, the critical kernel waits for 0.9ms (in this case) before it can start executing. If the two streams had identical priorities, the critical kernel would execute much later.

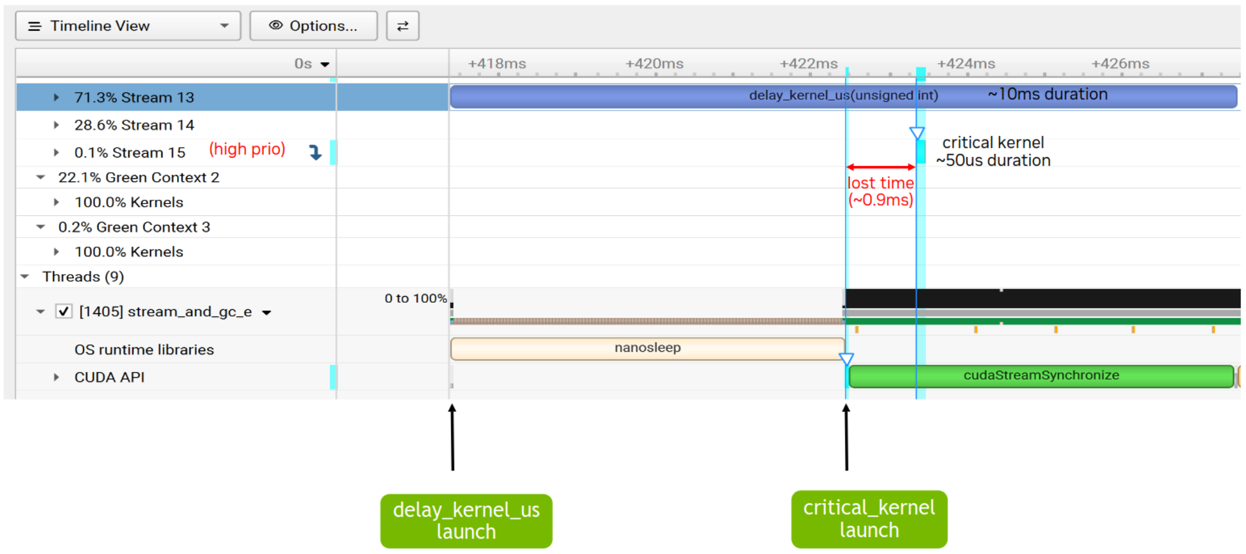


Figure 46: Nsight Systems timeline without green contexts

To leverage the green contexts feature, two green contexts are created, each provisioned with a distinct non-overlapping set of SMs. The exact SM split in this case for an H100 with 132 SMs was chosen, for illustration purposes, as 16 SMs for the critical kernel (Green Context 3) and 112 SMs for the long running kernel (Green Context 2). As Figure 47 shows, the critical kernel can now start almost instantaneously, as there are SMs only Green Context 3 can use.

The duration of the short kernel may increase, compared to its duration when running in isolation, as there is now a limit on the number of SMs it can use. The same is also the case for the long running kernel, which can no longer use all the SMs of the GPU, but is constrained by its green context's provisioned resources. However, the key result is that the critical kernel work can now start and complete significantly sooner than before. That is barring any other limitations, as parallel execution, as mentioned earlier, cannot be guaranteed.

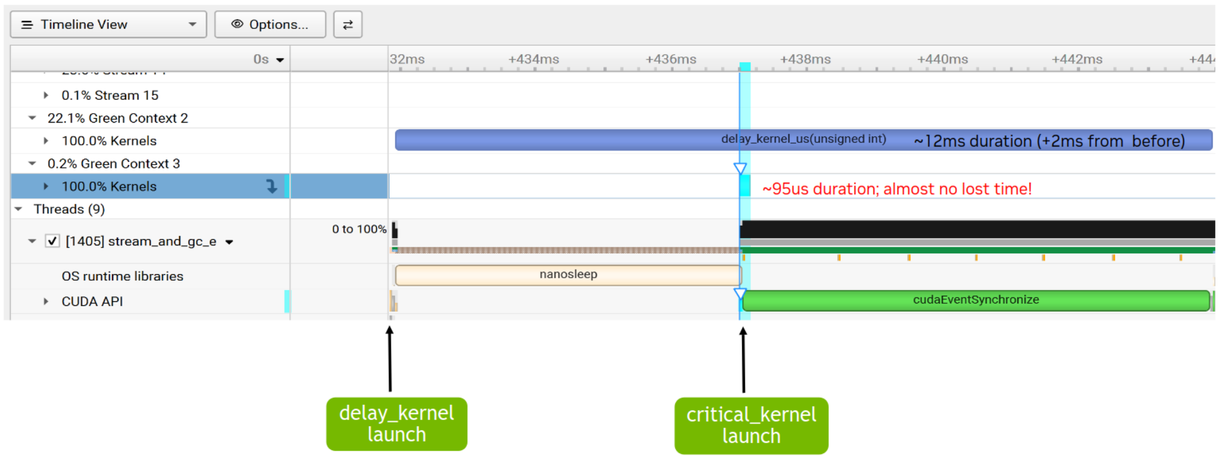


Figure 47: Nsight Systems timeline with green contexts

In all cases, the exact SM split should be decided on a per case basis after experimentation.

4.7. Lazy Loading

4.7.1. Introduction

Lazy loading reduces program initialization time by waiting to load CUDA modules until they are needed. Lazy loading is particularly effective for programs that only use a small number of the kernels they include, as is common when using libraries. Lazy loading is designed to be invisible to the user when the CUDA programming model is followed. *Potential Hazards* explains this in detail. As of CUDA 12.3 lazy Loading is enabled by default on all platforms, but can be controlled via the `CUDA_MODULE_LOADING` environment variable.

4.7.2. Change History

Table 17: Select Lazy Loading Changes by CUDA Version

CUDA Version	Change
12.3	Lazy loading performance improved. Now enabled by default for Windows.
12.2	Lazy loading enabled by default for Linux.
11.7	Lazy loading first introduced, disabled by default.

4.7.3. Requirements for Lazy Loading

Lazy loading is a joint feature of both the CUDA runtime and driver. Lazy loading is only available when the runtime and driver version requirements are satisfied.

4.7.3.1 CUDA Runtime Version Requirement

Lazy loading is available starting in CUDA runtime version 11.7. As CUDA runtime is usually linked statically into programs and libraries, only programs and libraries from or compiled with CUDA 11.7+ toolkit will benefit from lazy loading. Libraries compiled using older CUDA runtime versions will load all modules eagerly.

4.7.3.2 CUDA Driver Version Requirement

Lazy loading requires driver version 515 or newer. Lazy loading is not available for driver versions older than 515, even when using CUDA toolkit 11.7 or newer.

4.7.3.3 Compiler Requirements

Lazy loading does not require any compiler support. Both SASS and PTX compiled with pre-11.7 compilers can be loaded with lazy loading enabled, and will see full benefits of the feature. However, the version 11.7+ CUDA runtime is still required, as described above.

4.7.3.4 Kernel Requirements

Lazy loading does not affect modules containing managed variables, which will still be loaded eagerly.

4.7.4. Usage

4.7.4.1 Enabling & Disabling

Lazy loading is enabled by setting the `CUDA_MODULE_LOADING` environment variable to LAZY. Lazy loading can be disabled by setting the `CUDA_MODULE_LOADING` environment variable to EAGER. As of CUDA 12.3, lazy loading is enabled by default on all platforms.

4.7.4.2 Checking if Lazy Loading is Enabled at Runtime

The `cuModuleGetLoadingMode` API in the CUDA driver API can be used to determine if lazy loading is enabled. Note that CUDA must be initialized before running this function. Sample usage is shown in the snippet below.

```
#include "<cuda.h>"
#include "<assert.h>"
#include "<iostream>"

int main() {
        CUmoduleLoadingMode mode;

        assert(CUDA_SUCCESS == cuInit(0));
        assert(CUDA_SUCCESS == cuModuleGetLoadingMode(&mode));

        std::cout << "CUDA Module Loading Mode is " << ((mode == CU_MODULE_
↪LAZY_LOADING) ? "lazy" : "eager") << std::endl;

        return 0;
}
```

4.7.4.3 Forcing a Module to Load Eagerly at Runtime

Loading kernels and variables happens automatically, without any need for explicit loading. Kernels can be loaded explicitly even without executing them by doing the following:

- The `cuModuleGetFunction()` function will cause a module to be loaded into device memory
- The `cudaFuncGetAttributes()` function will cause a kernel to be loaded into device memory

Note

`cuModuleLoad()` does not guarantee that a module will be loaded immediately.

4.7.5. Potential Hazards

Lazy loading is designed so that it should not require any modifications to applications to use it. That said, there are some caveats, especially when applications are not fully compliant with the CUDA programming model, as described below.

4.7.5.1 Impact on Concurrent Kernel Execution

Some programs incorrectly assume that concurrent kernel execution is guaranteed. A deadlock can occur if cross-kernel synchronization is required, but kernel execution has been serialized. To minimize the impact of lazy loading on concurrent kernel execution, do the following:

- preload all kernels that you hope to execute concurrently prior to launching them or
- run application with `CUDA_MODULE_LOADING = EAGER` to force loading data eagerly without forcing each function to load eagerly

4.7.5.2 Large Memory Allocations

Lazy loading delays memory allocation for CUDA modules from program initialization until closer to execution time. If an application allocates the entire VRAM on startup, CUDA can fail to allocate memory for modules at runtime. Possible solutions:

- use `cudaMallocAsync()` instead of an allocator that allocates the entire VRAM on startup
- add some buffer to compensate for the delayed loading of kernels
- preload all kernels that will be used in the program before trying to initialize the allocator

4.7.5.3 Impact on Performance Measurements

Lazy loading may skew performance measurements by moving CUDA module initialization into the measured execution window. To avoid this:

- do at least one warmup iteration prior to measurement
- preload the benchmarked kernel prior to launching it

4.8. Error Log Management

The *Error Log Management* mechanism allows for CUDA API errors to be reported to developers in a plain-English format that describes the cause of the issue.

4.8.1. Background

Traditionally, the only indication of a failed CUDA API call is the return of a non-zero code. As of CUDA Toolkit 12.9, the CUDA Runtime defines over 100 different return codes for error conditions, but many of them are generic and give the developer no assistance with debugging the cause.

4.8.2. Activation

Set the *CUDA_LOG_FILE* environment variable. Acceptable values are *stdout*, *stderr*, or a valid path on the system to write a file. The log buffer can be dumped via API even if *CUDA_LOG_FILE* was not set before program execution. NOTE: An error-free execution may not print any logs.

4.8.3. Output

Logs are output in the following format:

```
[Time][TID][Source][Severity][API Entry Point] Message
```

The following line is an actual error message that is generated if the developer tries to dump the Error Log Management logs to an unallocated buffer:

```
[22:21:32.099][25642][CUDA][E][cuLogsDumpToMemory] buffer cannot be NULL
```

Where before, all the developer would have gotten is *CUDA_ERROR_INVALID_VALUE* in the return code and possibly "invalid argument" if *cuGetErrorString* is called.

4.8.4. API Description

The CUDA Driver provides APIs in two categories for interacting with the Error Log Management feature.

This feature allows developers to register callback functions to be used whenever an error log is generated, where the callback signature is:

```
void callbackFunc(void *data, CUlogLevel logLevel, char *message, size_t
↪length)
```

Callbacks are registered with this API:

```
CUresult cuLogsRegisterCallback(CUlogsCallback callbackFunc, void *userData,
↪CUlogsCallbackHandle *callback_out)
```

Where *userData* is passed to the callback function without modifications. *callback_out* should be stored by the caller for use in *cuLogsUnregisterCallback*.

```
CUresult cuLogsUnregisterCallback(CUlogsCallbackHandle callback)
```

The other set of API functions are for managing the output of logs. An important concept is the log iterator, which points to the current end of the buffer:

```
CUresult cuLogsCurrent(CUlogIterator *iterator_out, unsigned int flags)
```

The iterator position can be kept by the calling software in situations where a dump of the entire log buffer is not desired. Currently, the flags parameter must be 0, with additional options reserved for future CUDA releases.

At any time, the error log buffer can be dumped to either a file or memory with these functions:

```
CUresult cuLogsDumpToFile(CUlogIterator *iterator, const char *pathToFile,
↪unsigned int flags)
CUresult cuLogsDumpToMemory(CUlogIterator *iterator, char *buffer, size_t
↪*size, unsigned int flags)
```

If *iterator* is NULL, the entire buffer will be dumped, up to the maximum of 100 entries. If *iterator* is not NULL, logs will be dumped starting from that entry and the value of *iterator* will be updated to the current end of the logs, as if *culogsCurrent* had been called. If there have been more than 100 log entries into the buffer, a note will be added at the start of the dump noting this.

The flags parameter must be 0, with additional options reserved for future CUDA releases.

The *cuLogsDumpToMemory* function has additional considerations:

1. The buffer itself will be null-terminated, but each individual log entry will only be separated by a newline (n) character.
2. The maximum size of the buffer is 25600 bytes.
3. If the value provided in *size* is not sufficient to store all desired logs, a note will be added as the first entry and the oldest entries that do not fit will not be dumped.
4. After returning, *size* will contain the actual number of bytes written to the provided buffer.

4.8.5. Limitations and Known Issues

1. The log buffer is limited to 100 entries. After this limit is reached, the oldest entries will be replaced and log dumps will contain a line noting the rollover.
2. Not all CUDA APIs are covered yet. This is an ongoing project to provide better usage error reporting for all APIs.
3. The Error Log Management log location (if given) will not be tested for validity until/unless a log is generated.
4. The Error Log Management APIs are currently only available via the CUDA Driver. Equivalent APIs will be added to the CUDA Runtime in a future release.
5. The log messages are not localized to any language and all provided logs are in US English.

4.9. Asynchronous Barriers

Asynchronous barriers, introduced in *Advanced Synchronization Primitives*, extend CUDA synchronization beyond `__syncthreads()` and `__syncwarp()`, enabling fine-grained, non-blocking coordination and better overlap of communication and computation.

This section provides details on how to use asynchronous barriers mainly via the `cuda::barrier` API (with pointers to `cuda::ptx` and primitives where applicable).

4.9.1. Initialization

Initialization must happen before any thread begins participating in a barrier.

CUDA C++ cuda::barrier

```
#include <cuda/barrier>
#include <cooperative_groups.h>

__global__ void init_barrier()
{
  __shared__ cuda::barrier<cuda::thread_scope_block> bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // A single thread initializes the total expected arrival count.
    init(&bar, block.size());
  }
  block.sync();
}
```

CUDA C++ cuda::ptx

```
#include <cuda/ptx>
#include <cooperative_groups.h>

__global__ void init_barrier()
{
  __shared__ uint64_t bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // A single thread initializes the total expected arrival count.
    cuda::ptx::mbarrier_init(&bar, block.size());
  }
  block.sync();
}
```

CUDA C primitives

```
#include <cuda_awbarrier_primitives.h>
#include <cooperative_groups.h>

__global__ void init_barrier()
{
  __shared__ uint64_t bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // A single thread initializes the total expected arrival count.
    __mbarrier_init(&bar, block.size());
  }
  block.sync();
}
```

Before any thread can participate in a barrier, the barrier must be initialized using the `cuda::barrier::init()` friend function. This must happen before any thread arrives on the barrier. This poses a bootstrapping challenge in that threads must synchronize before participating in the barrier, but threads are creating a barrier in order to synchronize. In this example, threads that will participate are part of a cooperative group and use `block.sync()` to bootstrap initialization. Since a whole thread block is participating in the barrier, `__syncthreads()` could also be used.

The second parameter of `init()` is the *expected arrival count*, i.e., the number of times `bar.arrive()` will be called by participating threads before a participating thread is unblocked from its call to `bar.wait(std::move(token))`. In this and the previous examples, the barrier is initialized with the number of threads in the thread block i.e., `cooperative_groups::this_thread_block().size()`, so that all threads within the thread block can participate in the barrier.

Asynchronous barriers are flexible in specifying *how* threads participate (split arrive/wait) and *which* threads participate. In contrast, `this_thread_block.sync()` or `__syncthreads()` is applicable to the whole thread-block and `__syncwarp(mask)` to a specified subset of a warp. Nonetheless, if the intention of the user is to synchronize a full thread block or a full warp, we recommend using `__syncthreads()` and `__syncwarp()` respectively for better performance.

4.9.2. A Barrier's Phase: Arrival, Countdown, Completion, and Reset

An asynchronous barrier counts down from the expected arrival count to zero as participating threads call `bar.arrive()`. When the countdown reaches zero, the barrier is complete for the current phase. When the last call to `bar.arrive()` causes the countdown to reach zero, the countdown is automatically and atomically reset. The reset assigns the countdown to the expected arrival count, and moves the barrier to the next phase.

A `token` object of class `cuda::barrier::arrival_token`, as returned from `token=bar.arrive()`, is associated with the current phase of the barrier. A call to `bar.wait(std::move(token))` blocks the calling thread while the barrier is in the current phase, i.e., while the phase associated with the token matches the phase of the barrier. If the phase is advanced (because the countdown reaches zero) before the call to `bar.wait(std::move(token))` then the thread does not block; if the phase is advanced while the thread is blocked in `bar.wait(std::move(token))`, the thread is unblocked.

It is essential to know when a reset could or could not occur, especially in non-trivial arrive/wait synchronization patterns.

- A thread's calls to `token=bar.arrive()` and `bar.wait(std::move(token))` must be sequenced such that `token=bar.arrive()` occurs during the barrier's current phase, and `bar.wait(std::move(token))` occurs during the same or next phase.
- A thread's call to `bar.arrive()` must occur when the barrier's counter is non-zero. After barrier initialization, if a thread's call to `bar.arrive()` causes the countdown to reach zero then a call to `bar.wait(std::move(token))` must happen before the barrier can be reused for a subsequent call to `bar.arrive()`.
- `bar.wait()` must only be called using a `token` object of the current phase or the immediately preceding phase. For any other values of the `token` object, the behavior is undefined.

For simple arrive/wait synchronization patterns, compliance with these usage rules is straightforward.

4.9.2.1 Warp Entanglement

Warp-divergence affects the number of times an arrive on operation updates the barrier. If the invoking warp is fully converged, then the barrier is updated once. If the invoking warp is fully diverged, then 32 individual updates are applied to the barrier.

Note

It is recommended that `arrive-on(bar)` invocations are used by converged threads to minimize updates to the barrier object. When code preceding these operations diverges threads, then the warp should be re-converged, via `__syncwarp` before invoking arrive-on operations.

4.9.3. Explicit Phase Tracking

An asynchronous barrier can have multiple phases depending on how many times it is used to synchronize threads and memory operations. Instead of using tokens to track barrier phase flips, we can directly track a phase using the `mbarrier_try_wait_parity()` family of functions available through the `cuda::ptx` and primitives APIs.

In its simplest form, the `cuda::ptx::mbarrier_try_wait_parity(uint64_t* bar, const uint32_t& phaseParity)` function waits for a phase with a particular parity. The `phaseParity` operand is the integer parity of either the current phase or the immediately preceding phase of the barrier object. An even phase has integer parity 0 and an odd phase has integer parity 1. When we initialize a barrier, its phase has parity 0. So the valid values of `phaseParity` are 0 and 1. Explicit phase tracking can be useful when tracking *asynchronous memory operations*, as it allows only a single thread to arrive on the barrier and set the transaction count, while other threads only wait for a parity-based phase flip. This can be more efficient than having all threads arrive on the barrier and use tokens. This functionality is only available for shared-memory barriers at thread-block and cluster scope.

CUDA C++ cuda::barrier

```
#include <cuda/ptx>
#include <cooperative_groups.h>

__device__ void compute(float *data, int iteration);

__global__ void split_arrive_wait(int iteration_count, float *data)
{
  using barrier_t = cuda::barrier<cuda::thread_scope_block>;
  __shared__ barrier_t bar;
  int parity = 0; // Initial phase parity is 0.
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // Initialize barrier with expected arrival count.
    init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < iteration_count; ++i)
  {
    /* code before arrive */

    // This thread arrives. Arrival does not block a thread.
    // Get a handle to the native barrier to use with cuda::ptx API.
    (void)cuda::ptx::mbarrier_arrive(cuda::device::barrier_native_
↪handle(bar));

    compute(data, i);

    // Wait for all threads participating in the barrier to complete mbarrier_
↪arrive().
    // Get a handle to the native barrier to use with cuda::ptx API.
    while (!cuda::ptx::mbarrier_try_wait_parity(cuda::device::barrier_
↪native_handle(bar), parity)) {}
    // Flip parity.
    parity ^= 1;

    /* code after wait */
  }
}
```

CUDA C++ cuda::ptx

```
#include <cuda/ptx>
#include <cooperative_groups.h>

__device__ void compute(float *data, int iteration);

__global__ void split_arrive_wait(int iteration_count, float *data)
{
  __shared__ uint64_t bar;
  int parity = 0; // Initial phase parity is 0.
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // Initialize barrier with expected arrival count.
    cuda::ptx::mbarrier_init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < iteration_count; ++i)
  {
    /* code before arrive */

    // This thread arrives. Arrival does not block a thread.
    (void)cuda::ptx::mbarrier_arrive(&bar);

    compute(data, i);

    // Wait for all threads participating in the barrier to complete mbarrier_
↪arrive().
    while (!cuda::ptx::mbarrier_try_wait_parity(&bar, parity)) {}
    // Flip parity.
    parity ^= 1;

    /* code after wait */
  }
}
```

CUDA C primitives

```
#include <cuda_awbarrier_primitives.h>
#include <cooperative_groups.h>

__device__ void compute(float *data, int iteration);

__global__ void split_arrive_wait(int iteration_count, float *data)
{
  __shared__ __mbarrier_t bar;
  bool parity = false; // Initial phase parity is false.
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    // Initialize barrier with expected arrival count.
    __mbarrier_init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < iteration_count; ++i)
  {
    /* code before arrive */

    // This thread arrives. Arrival does not block a thread.
    (void)__mbarrier_arrive(&bar);

    compute(data, i);

    // Wait for all threads participating in the barrier to complete __
↪mbarrier_arrive().
    while(!__mbarrier_try_wait_parity(&bar, parity, 1000)) {}
    parity ^= 1;

    /* code after wait */
  }
}
```

4.9.4. Early Exit

When a thread that is participating in a sequence of synchronizations must exit early from that sequence, that thread must explicitly drop out of participation before exiting. The remaining participating threads can proceed normally with subsequent arrive and wait operations.

CUDA C++ cuda::barrier

```cpp
#include <cuda/barrier>
#include <cooperative_groups.h>

__device__ bool condition_check();

__global__ void early_exit_kernel(int N)
{
  __shared__ cuda::barrier<cuda::thread_scope_block> bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < N; ++i)
  {
    if (condition_check())
    {
      bar.arrive_and_drop();
      return;
    }
    // Other threads can proceed normally.
    auto token = bar.arrive();

    /* code between arrive and wait */

    // Wait for all threads to arrive.
    bar.wait(std::move(token));

    /* code after wait */
  }
}
```

CUDA C primitives

```
#include <cuda_awbarrier_primitives.h>
#include <cooperative_groups.h>

__device__ bool condition_check();

__global__ void early_exit_kernel(int N)
{
  __shared__ __mbarrier_t bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    __mbarrier_init(&bar, block.size());
  }
  block.sync();

  for (int i = 0; i < N; ++i)
  {
    if (condition_check())
    {
      __mbarrier_token_t token = __mbarrier_arrive_and_drop(&bar);
      return;
    }
    // Other threads can proceed normally.
    __mbarrier_token_t token = __mbarrier_arrive(&bar);

    /* code between arrive and wait */

    // Wait for all threads to arrive.
    while (!__mbarrier_try_wait(&bar, token, 1000)) {}

    /* code after wait */
  }
}
```

The `bar.arrive_and_drop()` operation arrives on the barrier to fulfill the participating thread's obligation to arrive in the **current** phase, and then decrements the expected arrival count for the **next** phase so that this thread is no longer expected to arrive on the barrier.

4.9.5. Completion Function

The `cuda::barrier` API supports an optional completion function. A `CompletionFunction` of `cuda::barrier<Scope, CompletionFunction>` is executed once per phase, after the last thread *arrives* and before any thread is unblocked from the `wait`. Memory operations performed by the threads that arrived at the `barrier` during the phase are visible to the thread executing the `CompletionFunction`, and all memory operations performed within the `CompletionFunction` are visible to all threads waiting at the `barrier` once they are unblocked from the `wait`.

CUDA C++ cuda::barrier

```
#include <cuda/barrier>
#include <cooperative_groups.h>
#include <functional>
namespace cg = cooperative_groups;

__device__ int divergent_compute(int *, int);
__device__ int independent_computation(int *, int);

__global__ void psum(int *data, int n, int *acc)
{
  auto block = cg::this_thread_block();

  constexpr int BlockSize = 128;
  __shared__ int smem[BlockSize];
  assert(BlockSize == block.size());
  assert(n % BlockSize == 0);

  auto completion_fn = [&]
  {
    int sum = 0;
    for (int i = 0; i < BlockSize; ++i)
    {
      sum += smem[i];
    }
    *acc += sum;
  };

  /* Barrier storage.
     Note: the barrier is not default-constructible because
           completion_fn is not default-constructible due
           to the capture. */
  using completion_fn_t = decltype(completion_fn);
  using barrier_t = cuda::barrier<cuda::thread_scope_block,
                                  completion_fn_t>;
  __shared__ std::aligned_storage<sizeof(barrier_t),
                                  alignof(barrier_t)>
      bar_storage;

  // Initialize barrier.
  barrier_t *bar = (barrier_t *)&bar_storage;
  if (block.thread_rank() == 0)
  {
    assert(*acc == 0);
    assert(blockDim.x == blockDim.y == blockDim.y == 1);
    new (bar) barrier_t{block.size(), completion_fn};
    /* equivalent to: init(bar, block.size(), completion_fn); */
  }
  block.sync();

  // Main loop.
  for (int i = 0; i < n; i += block.size())
  {
    smem[block.thread_rank()] = data[i] + *acc;
    auto token = bar->arrive();
    // We can do independent computation here.
    bar->wait(std::move(token));
```

4.9.6. Tracking Asynchronous Memory Operations

Asynchronous barriers can be used to track *asynchronous memory copies*. When an asynchronous copy operation is bound to a barrier, the copy operation automatically increments the expected count of the current barrier phase upon initiation and decrements it upon completion. This mechanism ensures that the barrier's `wait()` operation will block until all associated asynchronous memory copies have completed, providing a convenient way to synchronize multiple concurrent memory operations.

Starting with compute capability 9.0, asynchronous barriers in shared memory with thread-block or cluster scope can **explicitly** track asynchronous memory operations. We refer to these barriers as *asynchronous transaction barriers*. In addition to the expected arrival count, a barrier object can accept a **transaction count**, which can be used for tracking the completion of asynchronous transactions. The transaction count tracks the number of asynchronous transactions that are outstanding and yet to be complete, in units specified by the asynchronous memory operation (typically bytes). The transaction count to be tracked by the current phase can be set on arrival with `cuda::device::barrier_arrive_tx()` or directly with `cuda::device::barrier_expect_tx()`. When a barrier uses a transaction count, it blocks threads at the wait operation until all the producer threads have performed an arrive *and* the sum of all the transaction counts reaches an expected value.

CUDA C++ `cuda::barrier`

```
#include <cuda/barrier>
#include <cooperative_groups.h>

__global__ void track_kernel()
{
  __shared__ cuda::barrier<cuda::thread_scope_block> bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    init(&bar, block.size());
  }
  block.sync();

  auto token = cuda::device::barrier_arrive_tx(bar, 1, 0);

  bar.wait(cuda::std::move(token));
}
```

CUDA C++ cuda::ptx

```
#include <cuda/ptx>
#include <cooperative_groups.h>

__global__ void track_kernel()
{
  __shared__ uint64_t bar;
  auto block = cooperative_groups::this_thread_block();

  if (block.thread_rank() == 0)
  {
    cuda::ptx::mbarrier_init(&bar, block.size());
  }
  block.sync();

  uint64_t token = cuda::ptx::mbarrier_arrive_expect_tx(cuda::ptx::sem_
↪release, cuda::ptx::scope_cluster, cuda::ptx::space_shared, &bar, 1, 0);

  while (!cuda::ptx::mbarrier_try_wait(&bar, token)) {}
}
```

In this example, the `cuda::device::barrier_arrive_tx()` operation constructs an arrival token object associated with the phase synchronization point for the current phase. Then, decrements the arrival count by 1 and increments the expected transaction count by 0. Since the transaction count update is 0, the barrier is not tracking any transactions. The subsequent section on *Using the Tensor Memory Accelerator (TMA)* includes examples of tracking asynchronous memory operations.

4.9.7. Producer-Consumer Pattern Using Barriers

A thread block can be spatially partitioned to allow different threads to perform independent operations. This is most commonly done by assigning threads from different warps within the thread block to specific tasks. This technique is referred to as *warp specialization*.

This section shows an example of spatial partitioning in a producer-consumer pattern, where one subset of threads produces data that is concurrently consumed by the other (disjoint) subset of threads. A producer-consumer spatial partitioning pattern requires two one-sided synchronizations to manage a data buffer between the producer and consumer.

Producer	Consumer
wait for buffer to be ready to be filled	signal buffer is ready to be filled
produce data and fill the buffer	
signal buffer is filled	wait for buffer to be filled
	consume data in filled buffer

Producer threads wait for consumer threads to signal that the buffer is ready to be filled; however, consumer threads do not wait for this signal. Consumer threads wait for producer threads to signal that the buffer is filled; however, producer threads do not wait for this signal. For full producer/consumer concurrency this pattern has (at least) double buffering where each buffer requires two barriers.

CUDA C++ cuda::barrier

```
#include <cuda/barrier>

using barrier_t = cuda::barrier<cuda::thread_scope_block>;

__device__ void produce(barrier_t ready[], barrier_t filled[], float
↪*buffer, int buffer_len, float *in, int N)
{
  for (int i = 0; i < N / buffer_len; ++i)
  {
    ready[i % 2].arrive_and_wait(); /* wait for buffer_(i%2) to be ready to
↪be filled */
    /* produce, i.e., fill in, buffer_(i%2)  */
    barrier_t::arrival_token token = filled[i % 2].arrive(); /* buffer_(i
↪%2) is filled */
  }
}

__device__ void consume(barrier_t ready[], barrier_t filled[], float
↪*buffer, int buffer_len, float *out, int N)
{
  barrier_t::arrival_token token1 = ready[0].arrive(); /* buffer_0 is ready
↪for initial fill */
  barrier_t::arrival_token token2 = ready[1].arrive(); /* buffer_1 is ready
↪for initial fill */
  for (int i = 0; i < N / buffer_len; ++i)
  {
    filled[i % 2].arrive_and_wait(); /* wait for buffer_(i%2) to be filled */
    /* consume buffer_(i%2) */
    barrier_t::arrival_token token3 = ready[i % 2].arrive(); /* buffer_(i
↪%2) is ready to be re-filled */
  }
}

__global__ void producer_consumer_pattern(int N, float *in, float *out, int
↪buffer_len)
{
  constexpr int warpSize = 32;

  /* Shared memory buffer declared below is of size 2 * buffer_len
     so that we can alternatively work between two buffers.
     buffer_0 = buffer and buffer_1 = buffer + buffer_len */
  __shared__ extern float buffer[];

  /* bar[0] and bar[1] track if buffers buffer_0 and buffer_1 are ready to be
↪filled,
     while bar[2] and bar[3] track if buffers buffer_0 and buffer_1 are
↪filled-in respectively */
  #pragma nv_diag_suppress static_var_with_dynamic_init
  __shared__ barrier_t bar[4];

  if (threadIdx.x < 4)
  {
    init(bar + threadIdx.x, blockDim.x);
  }
  __syncthreads();
```

CUDA C++ cuda::ptx

```
#include <cuda/ptx>

__device__ void produce(barrier ready[], barrier filled[], float *buffer,
↪int buffer_len, float *in, int N)
{
  for (int i = 0; i < N / buffer_len; ++i)
  {
    uint64_t token1 = cuda::ptx::mbarrier_arrive(ready[i % 2]);
    while(!cuda::ptx::mbarrier_try_wait(&ready[i % 2], token1)) {} /* wait
↪for buffer_(i%2) to be ready to be filled */
    /* produce, i.e., fill in, buffer_(i%2)  */
    uint64_t token2 = cuda::ptx::mbarrier_arrive(&filled[i % 2]); /* buffer_
↪(i%2) is filled */
  }
}

__device__ void consume(barrier ready[], barrier filled[], float *buffer,
↪buffer_len, float *out, int N)
{
  uint64_t token1 = cuda::ptx::mbarrier_arrive(&ready[0]); /* buffer_0 is
↪ready for initial fill */
  uint64_t token2 = cuda::ptx::mbarrier_arrive(&ready[1]); /* buffer_1 is
↪ready for initial fill */
  for (int i = 0; i < N / buffer_len; ++i)
  {
    uint64_t token3 = cuda::ptx::mbarrier_arrive(&filled[i % 2]);
    while(!cuda::ptx::mbarrier_try_wait(&filled[i % 2], token3x)) {} /*
↪wait for buffer_(i%2) to be filled */
    /* consume buffer_(i%2) */
    uint64_t token4 = cuda::ptx::mbarrier_arrive(&ready[i % 2]); /* buffer_
↪(i%2) is ready to be re-filled */
  }
}

__global__ void producer_consumer_pattern(int N, float *in, float *out, int
↪buffer_len)
{
  constexpr int warpSize = 32;

  /* Shared memory buffer declared below is of size 2 * buffer_len
     so that we can alternatively work between two buffers.
     buffer_0 = buffer and buffer_1 = buffer + buffer_len */
  __shared__ extern float buffer[];

  /* bar[0] and bar[1] track if buffers buffer_0 and buffer_1 are ready to be
↪filled,
     while bar[2] and bar[3] track if buffers buffer_0 and buffer_1 are
↪filled-in respectively */
  #pragma nv_diag_suppress static_var_with_dynamic_init
  __shared__ uint64_t bar[4];

  if (threadIdx.x < 4)
  {
    cuda::ptx::mbarrier_init(bar + block.thread_rank(), block.size());
  }
  __syncthreads();
```

CUDA C primitives

```
#include <cuda_awbarrier_primitives.h>

__device__ void produce(__mbarrier_t ready[], __mbarrier_t filled[], float
↪*buffer, int buffer_len, float *in, int N)
{
  for (int i = 0; i < N / buffer_len; ++i)
  {
    __mbarrier_token_t token1 = __mbarrier_arrive(&ready[i % 2]); /* wait
↪for buffer_(i%2) to be ready to be filled */
    while(!__mbarrier_try_wait(&ready[i % 2], token1, 1000)) {}
    /* produce, i.e., fill in, buffer_(i%2)  */
    __mbarrier_token_t token2 = __mbarrier_arrive(filled[i % 2]);  /*
↪buffer_(i%2) is filled */
  }
}

__device__ void consume(__mbarrier_t ready[], __mbarrier_t filled[], float
↪*buffer, int buffer_len, float *out, int N)
{
  __mbarrier_token_t token1 = __mbarrier_arrive(&ready[0]); /* buffer_0 is
↪ready for initial fill */
  __mbarrier_token_t token2 = __mbarrier_arrive(&ready[1]); /* buffer_1 is
↪ready for initial fill */
  for (int i = 0; i < N / buffer_len; ++i)
  {
    __mbarrier_token_t token3 = __mbarrier_arrive(&filled[i % 2]);
    while(!__mbarrier_try_wait(&filled[i % 2], token3, 1000)) {}
    /* consume buffer_(i%2) */
    __mbarrier_token_t token4 = __mbarrier_arrive(&ready[i % 2]); /* buffer_
↪(i%2) is ready to be re-filled */
  }
}

__global__ void producer_consumer_pattern(int N, float *in, float *out, int
↪buffer_len)
{
  constexpr int warpSize = 32;

  /* Shared memory buffer declared below is of size 2 * buffer_len
     so that we can alternatively work between two buffers.
     buffer_0 = buffer and buffer_1 = buffer + buffer_len */
  __shared__ extern float buffer[];

  /* bar[0] and bar[1] track if buffers buffer_0 and buffer_1 are ready to be
↪filled,
     while bar[2] and bar[3] track if buffers buffer_0 and buffer_1 are
↪filled-in respectively */
  #pragma nv_diag_suppress static_var_with_dynamic_init
  __shared__ __mbarrier_t bar[4];

  if (threadIdx.x < 4)
  {
    __mbarrier_init(bar + threadIdx.x, blockDim.x);
  }
  __syncthreads();
```

In this example, the first warp is specialized as the producer and the remaining warps are specialized as consumers. All producer and consumer threads participate (call `bar.arrive()` or `bar.arrive_and_wait()`) in each of the four barriers so the expected arrival counts are equal to `block.size()`.

A producer thread waits for the consumer threads to signal that the shared memory buffer can be filled. In order to wait for a barrier, a producer thread must first arrive on that `ready[i%2].arrive()` to get a token and then `ready[i%2].wait(token)` with that token. For simplicity, `ready[i%2].arrive_and_wait()` combines these operations.

```
bar.arrive_and_wait();
/* is equivalent to */
bar.wait(bar.arrive());
```

Producer threads compute and fill the ready buffer, they then signal that the buffer is filled by arriving on the filled barrier, `filled[i%2].arrive()`. A producer thread does not wait at this point, instead it waits until the next iteration's buffer (double buffering) is ready to be filled.

A consumer thread begins by signaling that both buffers are ready to be filled. A consumer thread does not wait at this point, instead it waits for this iteration's buffer to be filled, `filled[i%2].arrive_and_wait()`. After the consumer threads consume the buffer they signal that the buffer is ready to be filled again, `ready[i%2].arrive()`, and then wait for the next iteration's buffer to be filled.

4.10. Pipelines

Pipelines, introduced in *Advanced Synchronization Primitives*, are a mechanism for staging work and coordinating multi-buffer producer–consumer patterns, commonly used to overlap compute with *asynchronous data copies*.

This section provides details on how to use pipelines mainly via the `cuda::pipeline` API (with pointers to primitives where applicable).

4.10.1. Initialization

A `cuda::pipeline` can be created at different thread scopes. For a scope other than `cuda::thread_scope_thread`, a `cuda::pipeline_shared_state<scope, count>` object is required to coordinate the participating threads. This state encapsulates the finite resources that allow a pipeline to process up to `count` concurrent stages.

```
// Create a pipeline at thread scope
constexpr auto scope = cuda::thread_scope_thread;
cuda::pipeline<scope> pipeline = cuda::make_pipeline();
```

```
// Create a pipeline at block scope
constexpr auto scope = cuda::thread_scope_block;
constexpr auto stages_count = 2;
__shared__ cuda::pipeline_shared_state<scope, stages_count> shared_state;
auto pipeline = cuda::make_pipeline(group, &shared_state);
```

Pipelines can be either *unified* or *partitioned*. In a unified pipeline, all the participating threads are both producers and consumers. In a partitioned pipeline, each participating thread is either a producer or a consumer and its role cannot change during the lifetime of the pipeline object. A thread-local pipeline cannot be partitioned. To create a partitioned pipeline, we need to provide either the number of producers or the role of the thread to `cuda::make_pipeline()`.

```
// Create a partitioned pipeline at block scope where only thread 0 is a
↪producer
constexpr auto scope = cuda::thread_scope_block;
constexpr auto stages_count = 2;
__shared__ cuda::pipeline_shared_state<scope, stages_count> shared_state;
auto thread_role = (group.thread_rank() == 0) ? cuda::pipeline_role::producer
↪: cuda::pipeline_role::consumer;
auto pipeline = cuda::make_pipeline(group, &shared_state, thread_role);
```

To support partitioning, a shared `cuda::pipeline` incurs additional overheads, including using a set of shared memory barriers per stage for synchronization. These are used even when the pipeline is unified and could use `__syncthreads()` instead. Thus, it is preferable to use thread-local pipelines which avoid these overheads when possible.

4.10.2. Submitting Work

Committing work to a pipeline stage involves:

- Collectively *acquiring* the pipeline *head* from a set of producer threads using `pipeline.producer_acquire()`.
- Submitting asynchronous operations, e.g., `memcpy_async`, to the pipeline head.
- Collectively *committing* (advancing) the pipeline head using `pipeline.producer_commit()`.

If all resources are in use, `pipeline.producer_acquire()` blocks producer threads until the resources of the next pipeline stage are released by consumer threads.

4.10.3. Consuming Work

Consuming work from a previously committed stage involves:

- Collectively waiting for the stage to complete, e.g., using `pipeline.consumer_wait()` to wait on the tail (oldest) stage, from a set of consumer threads.
- Collectively *releasing* the stage using `pipeline.consumer_release()`.

With `cuda::pipeline<cuda:thread_scope_thread>` one can also use the `cuda::pipeline_consumer_wait_prior<N>()` friend function to wait for all except the last N stages to complete, similar to `__pipeline_wait_prior(N)` in the primitives API.

4.10.4. Warp Entanglement

The pipeline mechanism is shared among CUDA threads in the same warp. This sharing causes sequences of submitted operations to be entangled within a warp, which can impact performance under certain circumstances.

Commit. The commit operation is coalesced such that the pipeline's sequence is incremented once for all converged threads that invoke the commit operation and their submitted operations are batched together. If the warp is fully converged, the sequence is incremented by one and all submitted operations will be batched in the same stage of the pipeline; if the warp is fully diverged, the sequence is incremented by 32 and all submitted operations will be spread to different stages.

- Let *PB* be the warp-shared pipeline's *actual* sequence of operations.

  `PB = {BP0, BP1, BP2, …, BPL}`

- Let *TB* be a thread's *perceived* sequence of operations, as if the sequence were only incremented by this thread's invocation of the commit operation.

  `TB = {BT0, BT1, BT2, …, BTL}`

  The `pipeline::producer_commit()` return value is from the thread's *perceived* batch sequence.

- An index in a thread's perceived sequence always aligns to an equal or larger index in the actual warp-shared sequence. The sequences are equal only when all commit operations are invoked from fully converged threads.

  `BTn` `BPm` where `n <= m`

For example, when a warp is fully diverged:

- The warp-shared pipeline's actual sequence would be: `PB = {0, 1, 2, 3, ..., 31}` (`PL=31`).
- The perceived sequence for each thread of this warp would be:
  - Thread 0: `TB = {0}` (`TL=0`)
  - Thread 1: `TB = {0}` (`TL=0`)
  - …
  - Thread 31: `TB = {0}` (`TL=0`)

Wait. A CUDA thread invokes `pipeline::consumer_wait()` or `pipeline_consumer_wait_prior<N>()` to wait for batches in the *perceived* sequence `TB` to complete. Note that `pipeline::consumer_wait()` is equivalent to `pipeline_consumer_wait_prior<N>()`, where `N = PL`.

The *wait prior* variants wait for batches in the *actual* sequence at least up to and including `PL-N`. Since `TL <= PL`, waiting for batch up to and including `PL-N` includes waiting for batch `TL-N`. Thus, when `TL < PL`, the thread will unintentionally wait for additional, more recent batches. In the extreme fully-diverged warp example above, each thread could wait for all 32 batches.

Note

It is recommended that commit invocations are by converged threads to not over-wait, by keeping threads' perceived sequence of batches aligned with the actual sequence.

When code preceding these operations diverges threads, then the warp should be re-converged, via `__syncwarp` before invoking commit operations.

4.10.5. Early Exit

When a thread that is participating in a pipeline must exit early, that thread must explicitly drop out of participation before exiting using `cuda::pipeline::quit()`. The remaining participating threads can proceed normally with subsequent operations.

4.10.6. Tracking Asynchronous Memory Operations

The following example demonstrates how to collectively copy data from global to shared memory with asynchronous memory copies using a pipeline to keep track of the copy operations. Each thread uses its own pipeline to independently submit memory copies and then wait for them to complete and consume the data. For more details on asynchronous data copies, see Section 3.2.5.

CUDA C++ cuda::pipeline

```
#include <cuda/pipeline>

__global__ void example_kernel(const float *in)
{
    constexpr int block_size = 128;
    __shared__ __align__(sizeof(float)) float buffer[4 * block_size];

    // Create a unified pipeline per thread
    cuda::pipeline<cuda::thread_scope_thread> pipeline = cuda::make_
↪pipeline();

    // First stage of memory copies
    pipeline.producer_acquire();
    // Every thread fetches one element of the first block
    cuda::memcpy_async(buffer, in, sizeof(float), pipeline);
    pipeline.producer_commit();

    // Second stage of memory copies
    pipeline.producer_acquire();
    // Every thread fetches one element of the second and third block
    cuda::memcpy_async(buffer + block_size, in + block_size, sizeof(float),
↪pipeline);
    cuda::memcpy_async(buffer + 2 * block_size, in + 2 * block_size,
↪sizeof(float), pipeline);
    pipeline.producer_commit();

    // Third stage of memory copies
    pipeline.producer_acquire();
    // Every thread fetches one element of the last block
    cuda::memcpy_async(buffer + 3 * block_size, in + 3 * block_size,
↪sizeof(float), pipeline);
    pipeline.producer_commit();

    // Wait for the oldest stage (waits for first stage)
    pipeline.consumer_wait();
    pipeline.consumer_release();

    // __syncthreads();
    // Use data from the first stage

    // Wait for the oldest stage (waits for second stage)
    pipeline.consumer_wait();
    pipeline.consumer_release();

    // __syncthreads();
    // Use data from the second stage

    // Wait for the oldest stage (waits for third stage)
    pipeline.consumer_wait();
    pipeline.consumer_release();

    // __syncthreads();
    // Use data from the third stage
}
```

CUDA C primitives

```
#include <cuda_pipeline.h>

__global__ void example_kernel(const float *in)
{
    constexpr int block_size = 128;
    __shared__ __align__(sizeof(float)) float buffer[4 * block_size];

    // First batch of memory copies
    // Every thread fetches one element of the first block
    __pipeline_memcpy_async(buffer, in, sizeof(float));
    __pipeline_commit();

    // Second batch of memory copies
    // Every thread fetches one element of the second and third block
    __pipeline_memcpy_async(buffer + block_size, in + block_size,
↪sizeof(float));
    __pipeline_memcpy_async(buffer + 2 * block_size, in + 2 * block_size,
↪sizeof(float));
    __pipeline_commit();

    // Third batch of memory copies
    // Every thread fetches one element of the last block
    __pipeline_memcpy_async(buffer + 3 * block_size, in + 3 * block_size,
↪sizeof(float));
    __pipeline_commit();

    // Wait for all except the last two batches of memory copies (waits for
↪first batch)
    __pipeline_wait_prior(2);

    // __syncthreads();
    // Use data from the first batch

    // Wait for all except the last batch of memory copies (waits for second
↪batch)
    __pipeline_wait_prior(1);

    // __syncthreads();
    // Use data from the second batch

    // Wait for all batches of memory copies (waits for third batch)
    __pipeline_wait_prior(0);

    // __syncthreads();
    // Use data from the last batch
}
```

4.10.7. Producer-Consumer Pattern using Pipelines

In Section 4.9.7, we showed how a thread block can be spatially partitioned to implement a producer-consumer pattern using *asynchronous barriers*. With `cuda::pipeline`, this can be simplified using a single partitioned pipeline with one stage per data buffer instead of two asynchronous barriers per buffer.

CUDA C++ cuda::pipeline

```
#include <cuda/pipeline>
#include <cooperative_groups.h>

#pragma nv_diag_suppress static_var_with_dynamic_init

using pipeline = cuda::pipeline<cuda::thread_scope_block>;

__device__ void produce(pipeline &pipe, int num_stages, int stage, int num_
↪batches, int batch, float *buffer, int buffer_len, float *in, int N)
{
  if (batch < num_batches)
  {
    pipe.producer_acquire();
    /* copy data from in(batch) to buffer(stage) using asynchronous memory
↪copies */
    pipe.producer_commit();
  }
}

__device__ void consume(pipeline &pipe, int num_stages, int stage, int num_
↪batches, int batch, float *buffer, int buffer_len, float *out, int N)
{
  pipe.consumer_wait();
  /* consume buffer(stage) and update out(batch) */
  pipe.consumer_release();
}

__global__ void producer_consumer_pattern(float *in, float *out, int N, int
↪buffer_len)
{
  auto block = cooperative_groups::this_thread_block();

  /* Shared memory buffer declared below is of size 2 * buffer_len
     so that we can alternatively work between two buffers.
     buffer_0 = buffer and buffer_1 = buffer + buffer_len */
  __shared__ extern float buffer[];

  const int num_batches = N / buffer_len;

  // Create a partitioned pipeline with 2 stages where half the threads are
↪producers and the other half are consumers.
  constexpr auto scope = cuda::thread_scope_block;
  constexpr int num_stages = 2;
  cuda::std::size_t producer_count = block.size() / 2;
  __shared__ cuda::pipeline_shared_state<scope, num_stages> shared_state;
  pipeline pipe = cuda::make_pipeline(block, &shared_state, producer_count);

  // Fill the pipeline
  if (block.thread_rank() < producer_count)
  {
    for (int s = 0; s < num_stages; ++s)
    {
      produce(pipe, num_stages, s, num_batches, s, buffer, buffer_len, in,
↪N);
    }
  }
```

In this example, we use half of the threads in the thread block as producers and the other half as consumers. As a first step, we need to create a `cuda::pipeline` object. Since we want some threads to be producers and some to be consumers, we need to use a **partitioned** pipeline with `cuda::thread_scope_block`. Partitioned pipelines require a `cuda::pipeline_shared_state` to coordinate the participating threads. We initialize the state for a 2-stage pipeline in thread-block scope and then call `cuda::make_pipeline()`. Next, producer threads fill the pipeline by submitting asynchronous copies from `in` to `buffer`. At this point all data copies are in-flight. Finally, in the main loop, we go over all of the batches of data and depending on whether a thread is a producer or consumer, we either submit another asynchronous copy for a future batch or consume the current batch.

4.11. Asynchronous Data Copies

Building on Section 3.2.5, this section provides detailed guidance and examples for asynchronous data movement within the GPU memory hierarchy. It covers LDGSTS for element-wise copies, the Tensor Memory Accelerator (TMA) for bulk (one-dimensional and multi-dimensional) transfers, and STAS for register to distributed shared memory copies, and shows how these mechanisms integrate with *asynchronous barriers* and *pipelines*.

4.11.1. Using LDGSTS

Many CUDA applications require frequent data movement between global and shared memory. Often, this involves copying smaller data elements or performing irregular memory access patterns. The primary goal of LDGSTS (CC 8.0+, see PTX documentation) is to provide an efficient asynchronous data transfer mechanism from global memory to shared memory for smaller, element-wise data transfers while enabling better utilization of compute resources through overlapped execution.

Dimensions. LDGSTS supports copying 4, 8, or 16 bytes. Copying 4 or 8 bytes always happens in the so called L1 ACCESS mode, in which case data is also cached in the L1, while copying 16-bytes enables the L1 BYPASS mode, in which case the L1 is not polluted.

Source and destination. The only direction supported for asynchronous copy operations with LDGSTS is from global to shared memory. The pointers need to be aligned to 4, 8, or 16 bytes depending on the size of the data being copied. Best performance is achieved when the alignment of both shared memory and global memory is 128 bytes.

Asynchronicity. Data transfers using LDGSTS are *asynchronous* and are modeled as async thread operations (see *Async Thread and Async Proxy*). This allows the initiating thread to continue computing while the hardware asynchronously copies the data. *Whether the data transfer occurs asynchronously in practice is up to the hardware implementation and may change in the future*.

LDGSTS must provide a signal when the operation is complete. LDGSTS can use *shared memory barriers* or *pipelines* as mechanisms to provide completion signals. By default, each thread only waits for its own LDGSTS copies. Thus, if you use LDGSTS to prefetch some data that will be shared with other threads, a `__syncthreads()` is necessary after synchronizing with the LDGSTS completion mechanism.

Table 18: Asynchronous copies with possible source and destination memory spaces and completion mechanisms using LDGSTS. An empty cell indicates that a source-destination pair is not supported.

Direction		Asynchronous Copy (LDGSTS, CC 8.0+)	
Source	Destination	Completion Mechanism	API
global	global		
shared::cta	global		
global	shared::cta	shared memory barrier, pipeline	cuda::memcpy_async, *cooperative_groups::memcpy_async*, *__pipeline_memcpy_async*
global	shared::clu		
shared::clu	shared::cta		
shared::cta	shared::cta		

In the following sections, we will demonstrate how to use LDGSTS through examples and explain the differences between the different APIs.

4.11.1.1 Batching Loads in Conditional Code

In this stencil example, the first warp of the thread block is responsible for collectively loading all the required data from the center as well as the left and right halos. With synchronous copies, due to the conditional nature of the code, the compiler may choose to generate a sequence of load-from-global (LDG) store-to-shared (STS) instructions instead of 3 LDGs followed by 3 STSs, which would be the optimal way to load the data to hide the global memory latency.

```
__global__ void stencil_kernel(const float *left, const float *center, const
↪float *right)
{
    // Left halo (8 elements) - center (32 elements) - right halo (8 elements)
    __shared__ float buffer[8 + 32 + 8];
    const int tid = threadIdx.x;

    if (tid < 8) {
        buffer[tid] = left[tid]; // Left halo
    } else if (tid >= 32 - 8) {
        buffer[tid + 16] = right[tid]; // Right halo
    }
    if (tid < 32) {
      buffer[tid + 8] = center[tid]; // Center
    }
    __syncthreads();

    // Compute stencil
}
```

To ensure that the data is loaded in the optimal way, we can replace the synchronous memory copies with asynchronous copies that load data directly from global memory to shared memory. This not only reduces register usage by copying the data directly to shared memory, but also ensures all loads from global memory are in-flight.

CUDA C++ cuda::memcpy_async

```
#include <cooperative_groups.h>
#include <cuda/barrier>

__global__ void stencil_kernel(const float *left, const float *center,
↪const float *right)
{
    auto block = cooperative_groups::this_thread_block();
    auto thread = cooperative_groups::this_thread();
    using barrier_t = cuda::barrier<cuda::thread_scope_block>;
    __shared__ barrier_t barrier;
    __shared__ float buffer[8 + 32 + 8];

    // Initialize synchronization object.
    if (block.thread_rank() == 0) {
        init(&barrier, block.size());
    }
    __syncthreads();

    // Version 1: Issue the copies in individual threads.
    if (tid < 8) {
        cuda::memcpy_async(buffer + tid, left + tid, cuda::aligned_size_t<4>
↪(sizeof(float)), barrier); // Left halo
        // or cuda::memcpy_async(thread, buffer + tid, left + tid,
↪cuda::aligned_size_t<4>(sizeof(float)), barrier);
    } else if (tid >= 32 - 8) {
        cuda::memcpy_async(buffer + tid + 16, right + tid, cuda::aligned_
↪size_t<4>(sizeof(float)), barrier); // Right halo
        // or cuda::memcpy_async(thread, buffer + tid + 16, right + tid,
↪cuda::aligned_size_t<4>(sizeof(float)), barrier);
    }
    if (tid < 32) {
        cuda::memcpy_async(buffer + 40, right + tid, cuda::aligned_size_t<4>
↪(sizeof(float)), barrier); // Center
        // or cuda::memcpy_async(thread, buffer + 40, right + tid,
↪cuda::aligned_size_t<4>(sizeof(float)), barrier);
    }

    // Version 2: Cooperatively issue the copies across all threads.
    cuda::memcpy_async(block, buffer, left, cuda::aligned_size_t<4>(8 *
↪sizeof(float)), barrier); // Left halo
    cuda::memcpy_async(block, buffer + 8, center, cuda::aligned_size_t<4>
↪(32 * sizeof(float)), barrier); // Center
    cuda::memcpy_async(block, buffer + 40, right, cuda::aligned_size_t<4>(8
↪* sizeof(float)), barrier); // Right halo

    // Wait for all copies to complete.
    barrier.arrive_and_wait();
    __syncthreads();

    // Compute stencil
}
```

CUDA C++ cooperative_groups::memcpy_async

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

namespace cg = cooperative_groups;

__global__ void stencil_kernel(const float *left, const float *center,
↪const float *right)
{
    cg::thread_block block = cg::this_thread_block();
    // Left halo (8 elements) - center (32 elements) - right halo (8
↪elements).
    __shared__ float buffer[8 + 32 + 8];

    // Cooperatively issue the copies across all threads.
    cg::memcpy_async(block, buffer, left, 8 * sizeof(float)); // Left halo
    cg::memcpy_async(block, buffer + 8, center, 32 * sizeof(float)); //
↪Center
    cg::memcpy_async(block, buffer + 40, right, 8 * sizeof(float)); //
↪Right halo
    cg::wait(block); // Waits for all copies to complete.
    __syncthreads();

    // Compute stencil.
}
```

CUDA C primitives

```
#include <cuda_pipeline.h>

__global__ void stencil_kernel(const float *left, const float *center,
↪const float *right)
{
    // Left halo (8 elements) - center (32 elements) - right halo (8
↪elements).
    __shared__ float buffer[8 + 32 + 8];
    const int tid = threadIdx.x;

    if (tid < 8) {
        __pipeline_memcpy_async(buffer + tid, left + tid, sizeof(float)); //
↪ Left halo
    } else if (tid >= 32 - 8) {
        __pipeline_memcpy_async(buffer + tid + 16, right + tid,
↪sizeof(float)); // Right halo
    }
    if (tid < 32) {
        __pipeline_memcpy_async(buffer + tid + 8, center + tid,
↪sizeof(float)); // Center
    }
    __pipeline_commit();
    __pipeline_wait_prior(0);
    __syncthreads();

    // Compute stencil.
}
```

The `cuda::memcpy_async` overload for `cuda::barrier` enables synchronizing asynchronous data transfers using an *asynchronous barrier*. This overload executes the copy operation as-if performed by another thread bound to the barrier by incrementing the expected count of the current phase on creation, and decrementing it on completion of the copy operation, such that the phase of the `barrier` will only advance when all threads participating in the barrier have arrived, and all `memcpy_async` bound to the current phase of the barrier have completed. We use a block-wide `barrier`, where all threads in the block participate, and merge the arrival and wait on the barrier with `arrive_and_wait`, since we do not perform any work between the phases.

Note that we can either use thread-level copies (version 1) or collective copies (version 2) to achieve the same result. In version 2, the API will automatically handle how the copies are done under the hood. In both versions, we use `cuda::aligned_size_t<4>()` to inform the compiler that the data is aligned to 4 bytes and the size of the data to copy is a multiple of 4 to enable use of LDGSTS. Note that for interoperability with `cuda::barrier`, `cuda::memcpy_async` from the `cuda/barrier` header is used here.

The *cooperative_groups::memcpy_async* implementation coordinates the memory transfers collectively across all threads in the block, but synchronizes completion with `cg::wait(block)` instead of explicit barrier operations.

The implementation based on the low-level primitives uses `__pipeline_memcpy_async()` to initiate element-wise memory transfers, `__pipeline_commit()` to commit the batch of copies, and `__pipeline_wait_prior(0)` to wait for all operations in the pipeline to complete. This provides the most direct control at the expense of more verbose code compared to the higher-level APIs. It also ensures LDGSTS will be used under the hood, which is not guaranteed with the higher-level APIs.

Note

The `cooperative_groups::memcpy_async` API is less efficient than the other APIs in this example because it automatically commits each copy operation immediately upon launch, preventing the optimization of batching multiple copies before a single commit operation that the other APIs enable.

4.11.1.2 Prefetching Data

In this example, we will demonstrate how to use asynchronous data copies to prefetch data from global memory to shared memory. In an iterative copy and compute pattern, this allows hiding the latency of data transfers of future iterations with computation on the current iteration, potentially increasing bytes-in-flight.

CUDA C++ cuda::memcpy_async

```
#include <cooperative_groups.h>
#include <cuda/pipeline>

template <size_t num_stages = 2 /* Pipeline with num_stages stages */>
__global__ void prefetch_kernel(int* global_out, int const* global_in, size_
↪t size, size_t batch_size) {
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
    auto thread = cooperative_groups::this_thread();
    assert(size == batch_size * grid.size()); // Assume input size fits
↪batch_size * grid_size

    extern __shared__ int shared[]; // num_stages * block.size() *
↪sizeof(int) bytes
    size_t shared_offset[num_stages];
    for (int s = 0; s < num_stages; ++s) shared_offset[s] = s * block.
↪size();

    cuda::pipeline<cuda::thread_scope_thread> pipeline = cuda::make_
↪pipeline();

    auto block_batch = [&](size_t batch) -> int {
        return block.group_index().x * block.size() + grid.size() * batch;
    };

    // Fill the pipeline with the first ``num_stages`` batches.
    for (int s = 0; s < num_stages; ++s) {
        pipeline.producer_acquire();
        cuda::memcpy_async(shared + shared_offset[s] + tid, global_in +
↪block_batch(s) + tid, cuda::aligned_size_t<4>(sizeof(int)), pipeline);
        pipeline.producer_commit();
    }

    int stage = 0;

    // compute_batch: next batch to process
    // fetch_batch:   next batch to fetch from global memory
    for (size_t compute_batch = 0, fetch_batch = num_stages; compute_batch
↪< batch_size; ++compute_batch, ++fetch_batch) {
        // Wait for the first requested stage to complete.
        constexpr size_t pending_batches = num_stages - 1;
        cuda::pipeline_consumer_wait_prior<pending_batches>(pipeline);
        __syncthreads(); // Not required if each thread works on the data it
↪copied.

        // Compute on the current batch
        compute(global_out + block_batch(compute_batch) + tid, shared +
↪shared_offset[stage] + tid);

        // Release the current stage.
        pipeline.consumer_release();
        __syncthreads(); // Not required if each thread works on the data it
↪copied.

        // Load future stage ``num_stages`` ahead of current compute batch.
        pipeline.producer_acquire();
```

CUDA C++ cooperative_groups::memcpy_async

```
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

namespace cg = cooperative_groups;

template <size_t num_stages = 2 /* Pipeline with num_stages stages */>
__global__ void prefetch_kernel(int* global_out, int const* global_in, size_
↪t size, size_t batch_size) {
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
    assert(size == batch_size * grid.size()); // Assume input size fits
↪batch_size * grid_size

    extern __shared__ int shared[]; // num_stages * block.size() *
↪sizeof(int) bytes
    size_t shared_offset[num_stages];
    for (int s = 0; s < num_stages; ++s) shared_offset[s] = s * block.
↪size();

    cuda::pipeline<cuda::thread_scope_thread> pipeline = cuda::make_
↪pipeline();

    auto block_batch = [&](size_t batch) -> int {
        return block.group_index().x * block.size() + grid.size() * batch;
    };

    // Fill the pipeline with the first ``num_stages`` batches.
    for (int s = 0; s < num_stages; ++s) {
        size_t block_batch_idx = block_batch(s);
        cg::memcpy_async(block, shared + shared_offset[s], global_in +
↪block_batch_idx, cuda::aligned_size_t<4>(sizeof(int)));
    }

    int stage = 0;

    // compute_batch: next batch to process
    // fetch_batch:   next batch to fetch from global memory
    for (size_t compute_batch = 0, fetch_batch = num_stages; compute_batch
↪< batch_size; ++compute_batch, ++fetch_batch) {
        // Wait for the first requested stage to complete.
        size_t pending_batches = (fetch_batch < batch_size - num_stages) ?
↪num_stages - 1 : batch_size - fetch_batch - 1;
        cg::wait_prior(pending_batches);
        __syncthreads(); // Not required if each thread works on the data it
↪copied.

        // Compute on the current batch.
        compute(global_out + block_batch(compute_batch) + tid, shared +
↪shared_offset[stage] + tid);

        __syncthreads(); // Not required if each thread works on the data it
↪copied.
```

```
        // Load future stage ``num_stages`` ahead of current compute batch.
        size_t fetch_batch_idx = block_batch(fetch_batch);
        if (fetch_batch < batch_size) {
```

CUDA C primitives

```
#include <cooperative_groups.h>
#include <cuda_awbarrier_primitives.h>

template <size_t num_stages = 2 /* Pipeline with num_stages stages */>
__global__ void prefetch_kernel(int* global_out, int const* global_in, size_
↪t size, size_t batch_size) {
    auto grid = cooperative_groups::this_grid();
    auto block = cooperative_groups::this_thread_block();
    assert(size == batch_size * grid.size()); // Assume input size fits
↪batch_size * grid_size

    extern __shared__ int shared[]; // num_stages * block.size() *
↪sizeof(int) bytes
    size_t shared_offset[num_stages];
    for (int s = 0; s < num_stages; ++s) shared_offset[s] = s * block.
↪size();

    auto block_batch = [&](size_t batch) -> int {
        return block.group_index().x * block.size() + grid.size() * batch;
    };

    // Fill the pipeline with the first ``num_stages`` batches.
    for (int s = 0; s < num_stages; ++s) {
        __pipeline_memcpy_async(shared + shared_offset[s] + tid, global_in
↪+ block_batch(s)+ tid, cuda::aligned_size_t<4>(sizeof(int)));
        __pipeline_commit();
    }

    // compute_batch: next batch to process
    // fetch_batch:   next batch to fetch from global memory
    for (size_t compute_batch = 0, fetch_batch = num_stages; compute_batch
↪< batch_size; ++compute_batch, ++fetch_batch) {
        // Wait for the first requested stage to complete.
        constexpr size_t pending_batches = num_stages - 1;
        __pipeline_wait_prior<pending_batches>();
        __syncthreads(); // Not required if each thread works on the data it
↪copied.

        // Compute on the current batch.
        compute(global_out + block_batch(compute_batch) + tid, shared +
↪shared_offset[stage] + tid);

        __syncthreads(); // Not required if each thread works on the data it
↪copied.

        // Load future stage ``num_stages`` ahead of current compute batch.
        if (fetch_batch < batch_size) {
            __pipeline_memcpy_async(shared + shared_offset[stage] + tid,
↪global_in + block_batch(fetch_batch) + tid, cuda::aligned_size_t<4>
↪(sizeof(int)));
        }
        __pipeline_commit();
        stage = (stage + 1) % num_stages;
    }
}
```

The `cuda::memcpy_async` implementation demonstrates a multi-stage data prefetching using `cuda::pipeline` (see *Pipelines*) with `cuda::memcpy_async`. It:

- Initializes a pipeline that is local to the thread.
- Kick-starts the pipeline by scheduling `num_stages` `memcpy_async` operations.
- Loops over all the batches: it blocks all threads on the completion of the current batch, then performs the computation on the current batch and finally schedules the next `memcpy_async` if there is one.

The `cooperative_groups::memcpy_async` implementation demonstrates multi-stage data prefetching using `cooperative_groups::memcpy_async`. The main difference with the previous implementation is that we do not use a pipeline object, but instead rely on `cooperative_groups::memcpy_async` to schedule the memory transfers in stages under the hood.

The CUDA C primitives implementation demonstrates multi-stage data prefetching using the low-level primitives in a quite similar manner to the first.

An important detail to enable efficient code generation in this example is to keep `num_stages` batches in the pipeline, even if there are no more batches to fetch. This is done by committing to the pipeline even if there are no more batches to fetch (`pipeline.producer_commit()` or `__pipeline_commit()`). Note that this is not possible with the cooperative groups API as we have no access to the internal pipeline.

4.11.1.3 Producer-Consumer Pattern Through Warp Specialization

In this example, we will demonstrate how to implement a producer-consumer pattern where a single warp is specialized as the producer performing asynchronous data copies from global to shared memory, while the remaining warps consume the data from shared memory and perform computations. To enable concurrency between the producer and the consumer threads, we use double-buffering in shared memory. While consumer warps process data in one buffer, the producer warp asynchronously fetches the next batch of data into the other buffer.

CUDA C++ cuda::memcpy_async

```
#include <cooperative_groups.h>
#include <cuda/pipeline>

#pragma nv_diag_suppress static_var_with_dynamic_init

using pipeline = cuda::pipeline<cuda::thread_scope_block>;

__device__ void produce(pipeline &pipe, int num_stages, int stage, int num_
↪batches, int batch, float *buffer, int buffer_len, float *in, int N)
{
  if (batch < num_batches)
  {
    pipe.producer_acquire();
    /* copy data from in(batch) to buffer(stage) using asynchronous memory
↪copies */
    cuda::memcpy_async(buffer + stage * buffer_len + threadIdx.x, in +
↪batch * buffer_len + threadIdx.x, cuda::aligned_size_t<4>(sizeof(float)),
↪pipe);
    pipe.producer_commit();
  }
}

__device__ void consume(pipeline &pipe, int num_stages, int stage, int num_
↪batches, int batch, float *buffer, int buffer_len, float *out, int N)
{
  pipe.consumer_wait();
  /* consume buffer(stage) and update out(batch) */
  pipe.consumer_release();
}

__global__ void producer_consumer_pattern(float *in, float *out, int N, int
↪buffer_len)
{
  auto block = cooperative_groups::this_thread_block();
  constexpr int warpSize = 32;

  /* Shared memory buffer declared below is of size 2 * buffer_len
     so that we can alternatively work between two buffers.
     buffer_0 = buffer and buffer_1 = buffer + buffer_len */
  __shared__ extern float buffer[];

  const int num_batches = N / buffer_len;

  // Create a partitioned pipeline with 2 stages where the first warp is the
↪producer and the other warps are consumers.
  constexpr auto scope = cuda::thread_scope_block;
  constexpr int num_stages = 2;
  cuda::std::size_t producer_count = warpSize;
  __shared__ cuda::pipeline_shared_state<scope, num_stages> shared_state;
  pipeline pipe = cuda::make_pipeline(block, &shared_state, producer_count);

  // Producer fills the pipeline
  if (block.thread_rank() < producer_count)
    for (int s = 0; s < num_stages; ++s)
      produce(pipe, num_stages, s, num_batches, s, buffer, buffer_len, in,
↪N);
```

CUDA C primitives

```
#include <cooperative_groups.h>
#include <cuda_awbarrier_primitives.h>

__device__ void produce(__mbarrier_t ready[], __mbarrier_t filled[], float
↪*buffer, int buffer_len, float *in, int N)
{
  for (int i = 0; i < N / buffer_len; ++i)
  {
    __mbarrier_token_t token = __mbarrier_arrive(&ready[i % 2]); /* wait
↪for buffer_(i%2) to be ready to be filled */
    while(!__mbarrier_try_wait(&ready[i % 2], token, 1000)) {}
    /* produce, i.e., fill in, buffer_(i%2)  */
    __pipeline_memcpy_async(buffer + i * buffer_len + threadIdx.x, in + i *
↪buffer_len + threadIdx.x, cuda::aligned_size_t<4>(sizeof(float)));
    __pipeline_arrive_on(filled[i % 2]);
    __mbarrier_arrive(filled[i % 2]);  /* buffer_(i%2) is filled */
  }
}

__device__ void consume(__mbarrier_t ready[], __mbarrier_t filled[], float
↪*buffer, int buffer_len, float *out, int N)
{
  __mbarrier_arrive(&ready[0]); /* buffer_0 is ready for initial fill */
  __mbarrier_arrive(&ready[1]); /* buffer_1 is ready for initial fill */
  for (int i = 0; i < N / buffer_len; ++i)
  {
    __mbarrier_token_t token = __mbarrier_arrive(&filled[i % 2]);
    while(!__mbarrier_try_wait(&filled[i % 2], token, 1000)) {}
    /* consume buffer_(i%2) */
    __mbarrier_arrive(&ready[i % 2]); /* buffer_(i%2) is ready to be re-
↪filled */
  }
}

__global__ void producer_consumer_pattern(int N, float *in, float *out, int
↪buffer_len)
{

  /* Shared memory buffer declared below is of size 2 * buffer_len
     so that we can alternatively work between two buffers.
     buffer_0 = buffer and buffer_1 = buffer + buffer_len */
  __shared__ extern float buffer[];

  /* bar[0] and bar[1] track if buffers buffer_0 and buffer_1 are ready to be
↪filled,
     while bar[2] and bar[3] track if buffers buffer_0 and buffer_1 are
↪filled-in respectively */
  __shared__ __mbarrier_t bar[4];

  // Initialize the barriers
  auto block = cooperative_groups::this_thread_block();
  if (block.thread_rank() < 4)
    __mbarrier_init(bar + block.thread_rank(), block.size());
  __syncthreads();

  if (block.thread_rank() < warpSize)
```

The `cuda::memcpy_async` implementation demonstrates the API with the highest level of abstraction with `cuda::memcpy_async` and a `cuda::pipeline` with 2 stages. It uses a partitioned pipeline (see *Pipelines*) where the first warp serves as a producer and the remaining warps as consumers. Producers initially fill both pipeline stages. Then, in the main processing loop, while consumers process the current batch, producers fetch data for future batches, maintaining a steady flow of work.

The CUDA C primitives implementation based on primitives combines `__pipeline_memcpy_async()` with *shared memory barriers* as the completion mechanism to coordinate the asynchronous memory transfers. The `__pipeline_arrive_on()` function associates the memory copy with the barrier. It increments the barrier arrival count by one and when all asynchronous operations sequenced before it have completed, the arrival count is automatically decremented by one and hence the net effect on the arrival count is zero. For this reason, we also need to explicitly wait on the barrier with `__mbarrier_arrive()`.

4.11.2. Using the Tensor Memory Accelerator (TMA)

Many applications need to move large amounts of data to and from global memory. Often, the data is laid out in global memory as a multi-dimensional array with non-sequential data access patterns. To reduce global memory accesses, sub-tiles of such arrays are copied to shared memory before use in computations. The loading and storing involves address-calculations that can be error-prone and repetitive. To offload these computations, compute capability 9.0 (Hopper) and later (see PTX documentation) have a *tensor memory accelerator* (TMA). The primary goal of the TMA is to provide an efficient data transfer mechanism from global memory to shared memory for multi-dimensional arrays.

Naming. Tensor memory accelerator (TMA) is a broad term used to refer to the features described in this section. For the purpose of forward-compatibility and to reduce discrepancies with the PTX ISA, the text in this section refers to TMA operations as either *bulk-asynchronous copies* or *bulk-tensor asynchronous copies*, depending on the specific type of copy used. The term "bulk" is used to contrast these operations with the asynchronous memory operations described in the previous section.

Dimensions. TMA supports copying both one-dimensional and multi-dimensional arrays (up to 5-dimensional). The programming model for bulk-asynchronous copies of one-dimensional contiguous arrays is different from the programming model for bulk-tensor asynchronous copies of multi-dimensional arrays. To perform a bulk-tensor asynchronous copy of a multi-dimensional array, the hardware requires a tensor map. This object describes the layout of the multi-dimensional array in global and shared memory. A tensor map is typically created on the host using the cuTensorMapEncode API and then transferred from host to device as a `const` kernel parameter annotated with `__grid_constant__` (see *__grid_constant__ Parameters*). The tensor map is transferred from host to device as a `const` kernel parameter annotated with `__grid_constant__`, and can be used on the device to copy a tile of data between shared and global memory. In contrast, performing a bulk-asynchronous copy of a contiguous one-dimensional array does not require a tensor map: it can be performed on-device with a pointer and size parameter.

Source and destination. The source and destination addresses of TMA operations can be in shared or global memory. The operations can read data from global to shared memory, write data from shared to global memory, and also copy from shared memory to *distributed shared memory* of another block in the same cluster. In addition, when in a cluster, a bulk-asynchronous tensor operation can be specified as being *multicast*. In this case, data can be transferred from global memory to the shared memory of multiple blocks within the cluster. The multicast feature is optimized for target architecture `sm_90a` and may have significantly reduced performance on other targets. Hence, it is advised to be used with compute architecture `sm_90a`.

Asynchronicity. Data transfers using TMA are *asynchronous* and are modeled as async proxy operations (see *Async Thread and Async Proxy*). This allows the initiating thread to continue computing while the hardware asynchronously copies the data. *Whether the data transfer occurs asynchronously in practice is up to the hardware implementation and may change in the future*. There are several completion mechanisms that bulk-asynchronous operations can use to signal that they have completed. When the operation reads from global to shared memory, any thread in the block can wait for the data to be readable in shared memory by waiting on a *shared memory barrier*. When the bulk-asynchronous operation writes data from shared memory to global or distributed shared memory, only the initiating thread can wait for the operation to have completed. This is accomplished using a *bulk async-group* based completion mechanism. A table describing the completion mechanisms can be found below and in the PTX ISA.

Table 19: Asynchronous copies with possible source and destination memory spaces and completion mechanisms using TMA. An empty cell indicates that a source-destination pair is not supported.

Direction		**Asynchronous Copy (TMA, CC 9.0+)**
Source	**Destination**	**Completion Mechanism**
global	global	
shared::cta	global	bulk async-group
global	shared::cta	shared memory barrier
global	shared::cluster	shared memory barrier (multicast)
shared::cta	shared::cluster	shared memory barrier
shared::cta	shared::cta	

4.11.2.1 Using TMA to transfer one-dimensional arrays

The following table summarizes the possible source and destination memory spaces and completion mechanisms for bulk-asynchronous TMA along with the API that exposes it.

Table 20: Asynchronous copies with possible source and destination memory spaces and completion mechanisms using bulk-asynchronous TMA. An empty cell indicates that a source-destination pair is not supported.

Direction		Bulk-Asynchronous Copy (TMA, CC9.0+)	
Source	Destination	Completion Mechanism	API
global	global		
shared::c	global	bulk async-group	cuda::ptx::cp_async_bulk
global	shared::cta	shared memory barrier	cuda::memcpy_async, cuda::device::memcpy_async_tx, cuda::ptx::cp_async_bulk
global	shared::clus	shared memory barrier	cuda::ptx::cp_async_bulk
shared::c	shared::clus	shared memory barrier	cuda::ptx::cp_async_bulk
shared::c	shared::cta		

Some functionality requires inline PTX that is currently made available through the `cuda::ptx` namespace in the CUDA Standard C++ library. The availability of these wrappers can be checked with the following code:

```
#if defined(__CUDA_MINIMUM_ARCH__) && __CUDA_MINIMUM_ARCH__ < 900
static_assert(false, "Device code is being compiled with older architectures
↪that are incompatible with TMA.");
#endif // __CUDA_MINIMUM_ARCH__
```

Note that `cuda::memcpy_async` uses TMA if the source and destination addresses are 16-byte aligned and the size is a multiple of 16 bytes, otherwise it falls back to synchronous copies. On the other hand, `cuda::device::memcpy_async_tx` and `cuda::ptx::cp_async_bulk` always use TMA and will result in undefined behavior if the requirements are not met.

In the following, we demonstrate how to use bulk-asynchronous copies through an example. The example read-modify-writes a one-dimensional array. The kernel goes through the following steps:

1. Initialize a shared memory barrier as a completion mechanism for the bulk-asynchronous copy from global to shared memory.
2. Initiate the copy of a block of memory from global to shared memory.
3. Arrive and wait on the shared memory barrier for completion of the copy.
4. Increment the shared memory buffer values.
5. Use a proxy fence to ensure shared memory writes (generic proxy) become visible to the subsequent bulk-asynchronous copy (async proxy).
6. Initiate a bulk-asynchronous copy of the buffer in shared memory to global memory.
7. Wait for the bulk-asynchronous copy to have finished reading shared memory.

```
#include <cuda/barrier>
#include <cuda/ptx>

using barrier = cuda::barrier<cuda::thread_scope_block>;
namespace ptx = cuda::ptx;

static constexpr size_t buf_len = 1024;

__device__ inline bool is_elected()
{
    unsigned int tid = threadIdx.x;
    unsigned int warp_id = tid / 32;
    unsigned int uniform_warp_id = __shfl_sync(0xFFFFFFFF, warp_id, 0); //
↪Broadcast from lane 0.
    return (uniform_warp_id == 0 && ptx::elect_sync(0xFFFFFFFF)); // Elect a
↪leader thread among warp 0.
}

__global__ void add_one_kernel(int* data, size_t offset)
{
  // Shared memory buffer. The destination shared memory buffer of
  // a bulk operation should be 16 byte aligned.
  __shared__ alignas(16) int smem_data[buf_len];

  // 1. Initialize shared memory barrier with the number of threads
↪participating in the barrier.
  #pragma nv_diag_suppress static_var_with_dynamic_init
  __shared__ barrier bar;
  if (threadIdx.x == 0) {
    init(&bar, blockDim.x);
  }
  __syncthreads();

  // 2. Initiate TMA transfer to copy global to shared memory from a single
↪thread.
  if (is_elected()) {
    // Launch the async copy and communicate how many bytes are expected to
↪come in (the transaction count).

    // Version 1: cuda::memcpy_async
    cuda::memcpy_async(
        smem_data, data + offset,
        cuda::aligned_size_t<16>(sizeof(smem_data)),
        bar);

    // Version 2: cuda::device::memcpy_async_tx
    // cuda::device::memcpy_async_tx(
    //   smem_data, data + offset,
    //   cuda::aligned_size_t<16>(sizeof(smem_data)),
    //   bar);
    // cuda::device::barrier_expect_tx(
    //     cuda::device::barrier_native_handle(bar),
    //     sizeof(smem_data));
```

(continues on next page)

(continued from previous page)

```
    // Version 3: cuda::ptx::cp_async_bulk
    // ptx::cp_async_bulk(
    //     ptx::space_shared, ptx::space_global,
    //     smem_data, data + offset,
    //     sizeof(smem_data),
    //     cuda::device::barrier_native_handle(bar));
    // cuda::device::barrier_expect_tx(
    //     cuda::device::barrier_native_handle(bar),
    //     sizeof(smem_data));
  }

  // 3a. All threads arrive on the barrier.
  barrier::arrival_token token = bar.arrive();

  // 3b. Wait for the data to have arrived.
  bar.wait(std::move(token));

  // 4. Compute saxpy and write back to shared memory.
  for (int i = threadIdx.x; i < buf_len; i += blockDim.x) {
    smem_data[i] += 1;
  }

  // 5. Wait for shared memory writes to be visible to TMA engine.
  ptx::fence_proxy_async(ptx::space_shared);
  __syncthreads();
  // After syncthreads, writes by all threads are visible to TMA engine.

  // 6. Initiate TMA transfer to copy shared memory to global memory.
  if (is_elected()) {
    ptx::cp_async_bulk(
        ptx::space_global, ptx::space_shared,
        data + offset, smem_data, sizeof(smem_data));
    // 7. Wait for TMA transfer to have finished reading shared memory.
    // Create a "bulk async-group" out of the previous bulk copy operation.
    ptx::cp_async_bulk_commit_group();
    // Wait for the group to have completed reading from shared memory.
    ptx::cp_async_bulk_wait_group_read(ptx::n32_t<0>());
  }
}
```

Barrier initialization. The barrier is initialized with the number of threads participating in the block. As a result, the barrier will flip only if all threads have arrived on this barrier. Shared memory barriers are described in more detail in *shared memory barriers*.

TMA read. The bulk-asynchronous copy instruction directs the hardware to copy a large chunk of data into shared memory, and to update the transaction count of the shared memory barrier after completing the read. In general, issuing as few bulk copies with as big a size as possible results in the best performance. Because the copy can be performed asynchronously by the hardware, it is not necessary to split the copy into smaller chunks.

The thread that initiates the bulk-asynchronous copy operation also tells the barrier how many transactions (tx) are expected to arrive. In this case, the transactions are counted in bytes. This is automatically performed by `cuda::memcpy_async`, but not by cuda::device::memcpy_async_tx and cuda::ptx::cp_async_bulk after which we need to explicitly call cuda::ptx::mbarrier_expect_tx. If multiple threads update the transaction count, the expected transaction will be the sum of the updates. The barrier will only flip once all threads have arrived **and** all bytes have arrived. Once the barrier has flipped, the bytes are safe to read from shared memory, both by the threads as well as by subsequent bulk-asynchronous copies. More information about barrier transaction accounting can be found in *Tracking Asynchronous Memory Operations*.

Barrier wait. Waiting for the barrier to flip is done using tokens with bar.wait(). It can be more efficient to use explicit phase tracking of the barrier (see *Explicit Phase Tracking*).

SMEM write and sync. The increment of the buffer values reads and writes to shared memory. To make the writes visible to subsequent bulk-asynchronous copies, the cuda::ptx::fence_proxy_async function is used. This orders the writes to shared memory before subsequent reads from bulk-asynchronous copy operations, which read through the async proxy. So each thread first orders the writes to objects in shared memory in the async proxy via the cuda::ptx::fence_proxy_async, and these operations by all threads are ordered before the async operation performed in thread 0 using __syncthreads().

TMA write and sync. The write from shared to global memory is again initiated by a single thread. The completion of the write is not tracked by a shared memory barrier. Instead, a thread-local mechanism is used. Multiple writes can be batched into a so-called *bulk async-group*. Afterwards, the thread can wait for all operations in this group to have completed reading from shared memory (as in the code above) or to have completed writing to global memory, making the writes visible to the initiating thread. For more information, refer to the PTX ISA documentation of cp.async.bulk.wait_group. Note that the bulk-asynchronous and non-bulk-asynchronous copy instructions have different async-groups: there exist both cp.async.wait_group and cp.async.bulk.wait_group instructions.

Note

It is recommended to initiate TMA operations by a single thread in the block. While using if (threadIdx.x == 0) might seem sufficient, the compiler cannot verify that indeed only one thread is initiating the copy and may insert a peeling loop over all active threads, which results in warp serialization and reduced performance. To prevent this, we define the is_elected() helper function that uses cuda::ptx::elect_sync to select one thread from warp 0 – which is known to the compiler – to execute the copy allowing it to generate more efficient code. Alternatively, the same effect can be achieved with *cooperative_groups::invoke_one*.

The bulk-asynchronous instructions have specific alignment requirements on their source and destination addresses. More information can be found in the table below.

Table 21: Alignment requirements for one-dimensional bulk-asynchronous operations.

Address / Size	**Alignment**
Global memory address	Must be 16 byte aligned.
Shared memory address	Must be 16 byte aligned.
Shared memory barrier address	Must be 8 byte aligned (this is guaranteed by cuda::barrier).
Size of transfer	Must be a multiple of 16 bytes.

4.11.2.1.1 Prefetching Data

In this example, we will demonstrate how to use TMA to prefetch data from global memory to shared memory. In an iterative copy and compute pattern, this allows hiding the latency of data transfers of future iterations with computation on the current iteration, potentially increasing bytes-in-flight.

CUDA C++ cuda::device::memcpy_async_tx

```
#include <cooperative_groups.h>
#include <cuda/barrier>
#include <cuda/ptx>

namespace ptx = cuda::ptx;
namespace cg = cooperative_groups;

__device__ inline bool is_elected()
{
    unsigned int tid = threadIdx.x;
    unsigned int warp_id = tid / 32;
    unsigned int uniform_warp_id = __shfl_sync(0xFFFFFFFF, warp_id, 0); //
↪Broadcast from lane 0.
    return (uniform_warp_id == 0 && ptx::elect_sync(0xFFFFFFFF)); // Elect
↪a leader thread among warp 0.
}

template <int block_size, int num_stages>
__global__ void prefetch_kernel(int* global_out, int const* global_in, size_
↪t size, size_t batch_size) {
    auto grid = cg::this_grid();
    auto block = cg::this_thread_block();
    const int tid = threadIdx.x;
    assert(size == batch_size * grid.size()); // Assume input size fits
↪batch_size * grid_size

    // 1. Initialization Phase
    __shared__ int shared[num_stages * block_size];
    size_t shared_offset[num_stages];
    for (int s = 0; s < num_stages; ++s) shared_offset[s] = s * block.
↪size();

    auto block_batch = [&](size_t batch) -> int {
        return block.group_index().x * block.size() + grid.size() * batch;
    };

    // Initialize shared memory barrier with the number of threads
↪participating in the barrier.
    // We will use explicit phase tracking for the barrier, which allows us
↪to have only one
    // thread arrive on the barrier to set the transaction count and other
↪threads wait for
    // a parity-based phase flip.
    #pragma nv_diag_suppress static_var_with_dynamic_init
    __shared__ cuda::barrier<cuda::thread_scope_block> bar[num_stages];
    if (tid == 0) {
        #pragma unroll num_stages
        for (int i = 0; i < num_stages; i++) {
            init(&bar[i], 1);
        }
    }
    __syncthreads();
```

```
    // Fill the pipeline with the first ``num_stages`` batches.
    if (is_elected()) {
        size_t num_bytes = block_size * sizeof(int);
```

This example implements *multi-stage data prefetching* using `cuda::device::memcpy_async_tx` for the TMA copies and employs shared memory barriers with explicit phase tracking for synchronization of the copies.

1. **Initialization Phase**: Sets up shared memory barriers (one per stage) and pre-loads the first `num_stages` batches into different shared memory sections.
2. **Main Processing Loop**:
   a. **Wait**: Uses `mbarrier_try_wait_parity()` to wait for the current batch to complete copying.
   b. **Compute**: Processes the current batch data.
   c. **Prefetch**: Schedules the next `memcpy_async_tx` operation for future data (staying `num_stages` ahead).
   d. **Stage Management**: Cycles through stages using a rotating buffer approach and tracks barrier parity.

4.11.2.2 Using TMA to transfer multi-dimensional arrays

In this section, we will focus on multi-dimensional TMA copies. The primary difference between the one-dimensional and multi-dimensional case is that a tensor map must be created on the host and passed to the CUDA kernel.

The following table summarizes the possible source and destination memory spaces and completion mechanisms for bulk-tensor asynchronous TMA along with the API that exposes it in device code.

Table 22: Asynchronous copies with possible source and destination memory spaces and completion mechanisms using bulk-tensor asynchronous TMA. An empty cell indicates that a source-destination pair is not supported.

Direction		Bulk-Tensor Asynchronous Copy (TMA, CC9.0+)	
Source	**Destination**	**Completion Mechanism**	**API**
global	global		
shared::cta	global	bulk async-group	cuda::ptx::cp_async_bulk_tensor
global	shared::cta	shared memory barrier	cuda::ptx::cp_async_bulk_tensor
global	shared::cluster	shared memory barrier	cuda::ptx::cp_async_bulk_tensor
shared::cta	shared::cluster	shared memory barrier	cuda::ptx::cp_async_bulk_tensor
shared::cta	shared::cta		

All functionality requires inline PTX that is currently made available through the `cuda::ptx` namespace in the CUDA Standard C++ library.

In the following, we describe how to create a tensor map using the CUDA driver API, how to pass it to the device, and how to use it on the device.

Driver API. A tensor map is created using the cuTensorMapEncodeTiled driver API. This API can be accessed by linking to the driver directly (`-lcuda`) or by using the cudaGetDriverEntryPointByVersion API. Below, we show how to get a pointer to the `cuTensorMapEncodeTiled` API. For more information, refer to *Driver Entry Point Access*.

```
#include <cudaTypedefs.h> // PFN_cuTensorMapEncodeTiled, CUtensorMap

PFN_cuTensorMapEncodeTiled_v12000 get_cuTensorMapEncodeTiled() {
  // Get pointer to cuTensorMapEncodeTiled
  cudaDriverEntryPointQueryResult driver_status;
  void* cuTensorMapEncodeTiled_ptr = nullptr;
  CUDA_CHECK(cudaGetDriverEntryPointByVersion("cuTensorMapEncodeTiled", &
↪cuTensorMapEncodeTiled_ptr, 12000, cudaEnableDefault, &driver_status));
  assert(driver_status == cudaDriverEntryPointSuccess);

  return reinterpret_cast<PFN_cuTensorMapEncodeTiled_v12000>
↪(cuTensorMapEncodeTiled_ptr);
}
```

Creation. Creating a tensor map requires many parameters. Among them are the base pointer to an array in global memory, the size of the array (in number of elements), the stride from one row to the next (in bytes), the size of the shared memory buffer (in number of elements). The code below creates a tensor map to describe a two-dimensional row-major array of size `GMEM_HEIGHT x GMEM_WIDTH`. Note the order of the parameters: the fastest moving dimension comes first.

```
  CUtensorMap tensor_map{};
  // rank is the number of dimensions of the array.
  constexpr uint32_t rank = 2;
  uint64_t size[rank] = {GMEM_WIDTH, GMEM_HEIGHT};
  // The stride is the number of bytes to traverse from the first element of
↪one row to the next.
  // It must be a multiple of 16.
  uint64_t stride[rank - 1] = {GMEM_WIDTH * sizeof(int)};
  // The box_size is the size of the shared memory buffer that is used as the
  // destination of a TMA transfer.
  uint32_t box_size[rank] = {SMEM_WIDTH, SMEM_HEIGHT};
  // The distance between elements in units of sizeof(element). A stride of 2
  // can be used to load only the real component of a complex-valued tensor,
↪for instance.
  uint32_t elem_stride[rank] = {1, 1};

  // Get a function pointer to the cuTensorMapEncodeTiled driver API.
  auto cuTensorMapEncodeTiled = get_cuTensorMapEncodeTiled();

  // Create the tensor descriptor.
  CUresult res = cuTensorMapEncodeTiled(
    &tensor_map,                // CUtensorMap *tensorMap,
    CUtensorMapDataType::CU_TENSOR_MAP_DATA_TYPE_INT32,
    rank,                       // cuuint32_t tensorRank,
    tensor_ptr,                 // void *globalAddress,
    size,                       // const cuuint64_t *globalDim,
    stride,                     // const cuuint64_t *globalStrides,
    box_size,                   // const cuuint32_t *boxDim,
    elem_stride,                // const cuuint32_t *elementStrides,
    // Interleave patterns can be used to accelerate loading of values that
    // are less than 4 bytes long.
    CUtensorMapInterleave::CU_TENSOR_MAP_INTERLEAVE_NONE,
    // Swizzling can be used to avoid shared memory bank conflicts.
```

(continues on next page)

(continued from previous page)

```
    CUtensorMapSwizzle::CU_TENSOR_MAP_SWIZZLE_NONE,
    // L2 Promotion can be used to widen the effect of a cache-policy to a
↪wider
    // set of L2 cache lines.
    CUtensorMapL2promotion::CU_TENSOR_MAP_L2_PROMOTION_NONE,
    // Any element that is outside of bounds will be set to zero by the TMA
↪transfer.
    CUtensorMapFloatOOBfill::CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE
  );
```

Host-to-device transfer. There are three ways to make a tensor map accessible to device code. The recommended approach is to pass the tensor map as a const `__grid_constant__` parameter to a kernel. The other possibilities are copying the tensor map into device `__constant__` memory using `cudaMemcpyToSymbol` or accessing it via global memory. When passing the tensor map as a parameter, some versions of the GCC C++ compiler issue the warning "the ABI for passing parameters with 64-byte alignment has changed in GCC 4.6". This warning can be ignored.

```
#include <cuda.h>

__global__ void kernel(const __grid_constant__ CUtensorMap tensor_map)
{
   // Use tensor_map here.
}
int main() {
  CUtensorMap map;
  // [ ..Initialize map.. ]
  kernel<<<1, 1>>>(map);
}
```

As an alternative to the `__grid_constant__` kernel parameter, a global `__constant__` variable can be used. An example is included below.

```
#include <cuda.h>

__constant__ CUtensorMap global_tensor_map;
__global__ void kernel()
{
  // Use global_tensor_map here.
}
int main() {
  CUtensorMap local_tensor_map;
  // [ ..Initialize map.. ]
  cudaMemcpyToSymbol(global_tensor_map, &local_tensor_map,
↪sizeof(CUtensorMap));
  kernel<<<1, 1>>>();
}
```

Finally, it is possible to copy the tensor map to global memory. Using a pointer to a tensor map in global device memory requires a fence in each thread block before any thread in the block uses the updated tensor map. Further uses of the tensor map by that thread block do not need to be fenced unless the tensor map is modified again. Note that this mechanism may be slower than the two mechanisms described above.

```
#include <cuda.h>
#include <cuda/ptx>
namespace ptx = cuda::ptx;

__device__ CUtensorMap global_tensor_map;
__global__ void kernel(CUtensorMap *tensor_map)
{
  // Fence acquire tensor map:
  ptx::n32_t<128> size_bytes;
  // Since the tensor map was modified from the host using cudaMemcpy,
  // the scope should be .sys.
  ptx::fence_proxy_tensormap_generic(
     ptx::sem_acquire, ptx::scope_sys, tensor_map, size_bytes
 );
 // Safe to use tensor_map after fence inside this thread.
}
int main() {
  CUtensorMap local_tensor_map;
  // [ ..Initialize map.. ]
  cudaMemcpy(&global_tensor_map, &local_tensor_map, sizeof(CUtensorMap),
↪cudaMemcpyHostToDevice);
  kernel<<<1, 1>>>(global_tensor_map);
}
```

Use. The kernel below loads a 2D tile of size SMEM_HEIGHT x SMEM_WIDTH from a larger 2D array. The top-left corner of the tile is indicated by the indices x and y. The tile is loaded into shared memory, modified, and written back to global memory.

```
#include <cuda.h>         // CUtensormap
#include <cuda/barrier>

using barrier = cuda::barrier<cuda::thread_scope_block>;
namespace ptx = cuda::ptx;

__device__ inline bool is_elected()
{
    unsigned int tid = threadIdx.x;
    unsigned int warp_id = tid / 32;
    unsigned int uniform_warp_id = __shfl_sync(0xFFFFFFFF, warp_id, 0); //
↪Broadcast from lane 0.
    return (uniform_warp_id == 0 && ptx::elect_sync(0xFFFFFFFF)); // Elect a
↪leader thread among warp 0.
}

__global__ void kernel(const __grid_constant__ CUtensorMap tensor_map, int x,
↪int y) {
  // The destination shared memory buffer of a bulk tensor operation should be
  // 128 byte aligned.
  __shared__ alignas(128) int smem_buffer[SMEM_HEIGHT][SMEM_WIDTH];

  // Initialize shared memory barrier with the number of threads participating
↪in the barrier.
  #pragma nv_diag_suppress static_var_with_dynamic_init
```

(continues on next page)

(continued from previous page)

```
  __shared__ barrier bar;

  if (threadIdx.x == 0) {
    // Initialize barrier. All `blockDim.x` threads in block participate.
    init(&bar, blockDim.x);
  }
  // Syncthreads so initialized barrier is visible to all threads.
  __syncthreads();

  barrier::arrival_token token;
  if (is_elected()) {
    // Initiate bulk tensor copy.
    int32_t tensor_coords[2] = { x, y };
    ptx::cp_async_bulk_tensor(
      ptx::space_shared, ptx::space_global,
      &smem_buffer, &tensor_map, tensor_coords,
      cuda::device::barrier_native_handle(bar));
    // Arrive on the barrier and tell how many bytes are expected to come in.
    token = cuda::device::barrier_arrive_tx(bar, 1, sizeof(smem_buffer));
  } else {
    // Other threads just arrive.
    token = bar.arrive();
  }
  // Wait for the data to have arrived.
  bar.wait(std::move(token));

  // Symbolically modify a value in shared memory.
  smem_buffer[0][threadIdx.x] += threadIdx.x;

  // Wait for shared memory writes to be visible to TMA engine.
  ptx::fence_proxy_async(ptx::space_shared);
  __syncthreads();
  // After syncthreads, writes by all threads are visible to TMA engine.

  // Initiate TMA transfer to copy shared memory to global memory
  if (is_elected()) {
    int32_t tensor_coords[2] = { x, y };
    ptx::cp_async_bulk_tensor(
      ptx::space_global, ptx::space_shared,
      &tensor_map, tensor_coords, &smem_buffer);
    // Wait for TMA transfer to have finished reading shared memory.
    // Create a "bulk async-group" out of the previous bulk copy operation.
    ptx::cp_async_bulk_commit_group();
    // Wait for the group to have completed reading from shared memory.
    ptx::cp_async_bulk_wait_group_read(ptx::n32_t<0>());
  }

  // Destroy barrier. This invalidates the memory region of the barrier. If
  // further computations were to take place in the kernel, this allows the
  // memory location of the shared memory barrier to be reused.
  if (threadIdx.x == 0) {
    (&bar)->~barrier();
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Negative indices and out of bounds. When part of the tile that is being *read* from global to shared memory is out of bounds, the shared memory that corresponds to the out of bounds area is zero-filled. The top-left corner indices of the tile may also be negative. When *writing* from shared to global memory, parts of the tile may be out of bounds, but the top left corner cannot have any negative indices.

Size and stride. The size of a tensor is the number of elements along one dimension. All sizes must be greater than one. The stride is the number of bytes between elements of the same dimension. For instance, a *4 x 4* matrix of integers has sizes 4 and 4. Since it has 4 bytes per element, the strides are 4 and 16 bytes. Due to alignment requirements, a *4 x 3* row-major matrix of integers must have strides of 4 and 16 bytes as well. Each row is padded with 4 extra bytes to ensure that the start of the next row is aligned to 16 bytes. More information about alignment requirements can be found in the table below.

Table 23: Alignment requirements for multi-dimensional bulk tensor asynchronous copy operations.

Address / Size	**Alignment**
Global memory address	Must be 16 byte aligned.
Global memory sizes	Must be greater than or equal to one. Does not have to be a multiple of 16 bytes.
Global memory strides	Must be multiples of 16 bytes.
Shared memory address	Must be 128 byte aligned.
Shared memory barrier address	Must be 8 byte aligned (this is guaranteed by `cuda::barrier`).
Size of transfer	Must be a multiple of 16 bytes.

4.11.2.2.1 Encoding a Tensor Map on Device

Previous sections have described how to create a tensor map on the host using the CUDA driver API.

This section explains how to encode a tiled-type tensor map on device. This is useful in situations where the typical way of transferring the tensor map (using `const __grid_constant__` kernel parameters) is undesirable, for instance, when processing a batch of tensors of various sizes in a single kernel launch.

The recommended pattern is as follows:

1. Create a tensor map "template", `template_tensor_map`, using the Driver API on the host.
2. In a device kernel, copy the `template_tensor_map`, modify the copy, store in global memory, and appropriately fence.
3. Use the tensor map in a kernel with appropriate fencing.

The high-level code structure is as follows:

```
// Initialize device context:
CUDA_CHECK(cudaDeviceSynchronize());

// Create a tensor map template using the cuTensorMapEncodeTiled driver function
CUtensorMap template_tensor_map = make_tensormap_template();

// Allocate tensor map and tensor in global memory
CUtensorMap* global_tensor_map;
CUDA_CHECK(cudaMalloc(&global_tensor_map, sizeof(CUtensorMap)));
char* global_buf;
CUDA_CHECK(cudaMalloc(&global_buf, 8 * 256));

// Fill global buffer with data.
fill_global_buf<<<1, 1>>>(global_buf);

// Define the parameters of the tensor map that will be created on device.
tensormap_params p{};
p.global_address    = global_buf;
p.rank              = 2;
p.box_dim[0]        = 128; // The box in shared memory has half the width of
↪the full buffer
p.box_dim[1]        = 4;   // The box in shared memory has half the height of
↪the full buffer
p.global_dim[0]     = 256; //
p.global_dim[1]     = 8;   //
p.global_stride[0]  = 256; //
p.element_stride[0] = 1;   //
p.element_stride[1] = 1;   //

// Encode global_tensor_map on device:
encode_tensor_map<<<1, 32>>>(template_tensor_map, p, global_tensor_map);

// Use it from another kernel:
consume_tensor_map<<<1, 1>>>(global_tensor_map);

// Check for errors:
CUDA_CHECK(cudaDeviceSynchronize());
```

The following sections describe the high-level steps. Throughout the examples, the following tensormap_params struct contains the new values of the fields to be updated. It is included here to reference when reading the examples.

```
struct tensormap_params {
  void* global_address;
  int rank;
  uint32_t box_dim[5];
  uint64_t global_dim[5];
  size_t global_stride[4];
  uint32_t element_stride[5];
};
```

4.11.2.2.2 Device-side Encoding and Modification of a Tensor Map

The recommended process of encoding a tensor map in global memory proceeds as follows.

1. Pass an existing tensor map, the `template_tensor_map`, to the kernel. In contrast to kernels that use the tensor map in a `cp.async.bulk.tensor` instruction, this may be done in any way: a pointer to global memory, kernel parameter, a `__const___` variable, and so on.
2. Copy-initialize a tensor map in shared memory with the template_tensor_map value.
3. Modify the tensor map in shared memory using the cuda::ptx::tensormap_replace functions. These functions wrap the tensormap.replace PTX instruction, which can be used to modify any field of a tiled-type tensor map, including the base address, size, stride, and so on.
4. Using the cuda::ptx::tensormap_copy_fenceproxy function, copy the modified tensor map from shared memory to global memory and perform any necessary fencing.

The following code contains a kernel that follows these steps. For completeness, it modifies all the fields of the tensor map. Typically, a kernel will modify just a few fields.

In this kernel, `template_tensor_map` is passed as a kernel parameter. This is the preferred way of moving `template_tensor_map` from the host to the device. If the kernel is intended to update an existing tensor map in device memory, it can take a pointer to the existing tensor map to modify.

Note

The format of the tensor map may change over time. Therefore, the cuda::ptx::tensormap_replace functions and corresponding tensormap.replace.tile PTX instructions are marked as specific to sm_90a. To use them, compile using `nvcc -arch sm_90a ....`.

Tip

On sm_90a, a zero-initialized buffer in shared memory may also be used as the initial tensor map value. This enables encoding a tensor map purely on device, without using the driver API to encode the `template_tensor_map value`.

Note

On-device modification is only supported for tiled-type tensor maps; other tensor map types cannot be modified on device. For more information on the tensor map types, refer to the Driver API reference.

```
#include <cuda/ptx>

namespace ptx = cuda::ptx;

// launch with 1 warp.
__launch_bounds__(32)
__global__ void encode_tensor_map(const __grid_constant__ CUtensorMap
↪template_tensor_map, tensormap_params p, CUtensorMap* out) {
   __shared__ alignas(128) CUtensorMap smem_tmap;
   if (threadIdx.x == 0) {
```

(continues on next page)

(continued from previous page)

```
    // Copy template to shared memory:
    smem_tmap = template_tensor_map;

    const auto space_shared = ptx::space_shared;
    ptx::tensormap_replace_global_address(space_shared, &smem_tmap, p.
↪global_address);
    // For field .rank, the operand new_val must be ones less than the
↪desired
    // tensor rank as this field uses zero-based numbering.
    ptx::tensormap_replace_rank(space_shared, &smem_tmap, p.rank - 1);

    // Set box dimensions:
    if (0 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_
↪tmap, ptx::n32_t<0>{}, p.box_dim[0]); }
    if (1 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_
↪tmap, ptx::n32_t<1>{}, p.box_dim[1]); }
    if (2 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_
↪tmap, ptx::n32_t<2>{}, p.box_dim[2]); }
    if (3 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_
↪tmap, ptx::n32_t<3>{}, p.box_dim[3]); }
    if (4 < p.rank) { ptx::tensormap_replace_box_dim(space_shared, &smem_
↪tmap, ptx::n32_t<4>{}, p.box_dim[4]); }
    // Set global dimensions:
    if (0 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_
↪tmap, ptx::n32_t<0>{}, (uint32_t) p.global_dim[0]); }
    if (1 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_
↪tmap, ptx::n32_t<1>{}, (uint32_t) p.global_dim[1]); }
    if (2 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_
↪tmap, ptx::n32_t<2>{}, (uint32_t) p.global_dim[2]); }
    if (3 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_
↪tmap, ptx::n32_t<3>{}, (uint32_t) p.global_dim[3]); }
    if (4 < p.rank) { ptx::tensormap_replace_global_dim(space_shared, &smem_
↪tmap, ptx::n32_t<4>{}, (uint32_t) p.global_dim[4]); }
    // Set global stride:
    if (1 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &
↪smem_tmap, ptx::n32_t<0>{}, p.global_stride[0]); }
    if (2 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &
↪smem_tmap, ptx::n32_t<1>{}, p.global_stride[1]); }
    if (3 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &
↪smem_tmap, ptx::n32_t<2>{}, p.global_stride[2]); }
    if (4 < p.rank) { ptx::tensormap_replace_global_stride(space_shared, &
↪smem_tmap, ptx::n32_t<3>{}, p.global_stride[3]); }
    // Set element stride:
    if (0 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &
↪smem_tmap, ptx::n32_t<0>{}, p.element_stride[0]); }
    if (1 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &
↪smem_tmap, ptx::n32_t<1>{}, p.element_stride[1]); }
    if (2 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &
↪smem_tmap, ptx::n32_t<2>{}, p.element_stride[2]); }
    if (3 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &
↪smem_tmap, ptx::n32_t<3>{}, p.element_stride[3]); }
    if (4 < p.rank) { ptx::tensormap_replace_element_size(space_shared, &
```

(continues on next page)

(continued from previous page)

```
↪smem_tmap, ptx::n32_t<4>{}, p.element_stride[4]); }

      // These constants are documented in this table:
      // https://docs.nvidia.com/cuda/parallel-thread-execution/index.html
↪#tensormap-new-val-validity
      auto u8_elem_type = ptx::n32_t<0>{};
      ptx::tensormap_replace_elemtype(space_shared, &smem_tmap, u8_elem_type);
      auto no_interleave = ptx::n32_t<0>{};
      ptx::tensormap_replace_interleave_layout(space_shared, &smem_tmap, no_
↪interleave);
      auto no_swizzle = ptx::n32_t<0>{};
      ptx::tensormap_replace_swizzle_mode(space_shared, &smem_tmap, no_
↪swizzle);
      auto zero_fill = ptx::n32_t<0>{};
      ptx::tensormap_replace_fill_mode(space_shared, &smem_tmap, zero_fill);
   }
   // Synchronize the modifications with other threads in warp
   __syncwarp();
   // Copy the tensor map to global memory collectively with threads in the
↪warp.
   // In addition: make the updated tensor map visible to other threads on
↪device that
   // for use with cp.async.bulk.
   ptx::n32_t<128> bytes_128;
   ptx::tensormap_cp_fenceproxy(ptx::sem_release, ptx::scope_gpu, out, &smem_
↪tmap, bytes_128);
}
```

4.11.2.2.3 Usage of a Modified Tensor Map

In contrast to using a tensor map that is passed as a `const __grid_constant__` kernel parameter, using a tensor map in global memory requires explicitly establishing a release-acquire pattern in the tensor map proxy between the threads that modify the tensor map and the threads that use it.

The release part of the pattern was shown in the previous section. It is accomplished using the cuda::ptx::tensormap.cp_fenceproxy function.

The acquire part is accomplished using the cuda::ptx::fence_proxy_tensormap_generic function that wraps the fence.proxy.tensormap::generic.acquire instruction. If the two threads participating in the release-acquire pattern are on the same device, the `.gpu` scope suffices. If the threads are on different devices, the `.sys` scope must be used. Once a tensor map has been acquired by one thread, it can be used by other threads in the block after sufficient synchronization, for example, using `__syncthreads()`. The thread that uses the tensor map and the thread that performs the fence must be in the same block. That is, if the threads are in, for example, two different thread blocks of the same cluster, the same grid, or a different kernel, synchronization APIs such as `cooperative_groups::cluster` or `grid_group::sync()` or stream-order synchronization do not suffice to establish ordering for tensor map updates, that is, threads in these other thread blocks still need to acquire the tensor map proxy at the right scope before using the updated tensor map. If there are no intermediate modifications, the fence does not have to be repeated before each `cp.async.bulk.tensor` instruction.

The `fence` and subsequent use of the tensor map is shown in the following example.

```
// Consumer of tensor map in global memory:
__global__ void consume_tensor_map(CUtensorMap* tensor_map) {
  // Fence acquire tensor map:
  ptx::n32_t<128> size_bytes;
  ptx::fence_proxy_tensormap_generic(ptx::sem_acquire, ptx::scope_sys, tensor_
↪map, size_bytes);
  // Safe to use tensor_map after fence.

  __shared__ uint64_t bar;
  __shared__ alignas(128) char smem_buf[4][128];

  if (threadIdx.x == 0) {
    // Initialize barrier
    ptx::mbarrier_init(&bar, 1);
    // Issue TMA request
    ptx::cp_async_bulk_tensor(ptx::space_cluster, ptx::space_global, smem_buf,
↪ tensor_map, {0, 0}, &bar);
    // Arrive on barrier. Expect 4 * 128 bytes.
    ptx::mbarrier_arrive_expect_tx(ptx::sem_release, ptx::scope_cta,
↪ptx::space_shared, &bar, sizeof(smem_buf));
  }
  const int parity = 0;
  // Wait for load to have completed
  while (!ptx::mbarrier_try_wait_parity(&bar, parity)) {}

  // print items:
  printf("Got:\n\n");
  for (int j = 0; j < 4; ++j) {
    for (int i = 0; i < 128; ++i) {
      printf("%3d ", smem_buf[j][i]);
      if (i % 32 == 31) { printf("\n"); };
    }
    printf("\n");
  }
}
```

4.11.2.2.4 Creating a Template Tensor Map Value Using the Driver API

The following code creates a minimal tiled-type tensor map that can be subsequently modified on device.

```
CUtensorMap make_tensormap_template() {
  CUtensorMap template_tensor_map{};
  auto cuTensorMapEncodeTiled = get_cuTensorMapEncodeTiled();

  uint32_t dims_32         = 16;
  uint64_t dims_strides_64 = 16;
  uint32_t elem_strides    = 1;

  // Create the tensor descriptor.
  CUresult res = cuTensorMapEncodeTiled(
    &template_tensor_map, // CUtensorMap *tensorMap,
```

(continues on next page)

(continued from previous page)

```
    CUtensorMapDataType::CU_TENSOR_MAP_DATA_TYPE_UINT8,
    1,                  // cuuint32_t tensorRank,
    nullptr,            // void *globalAddress,
    &dims_strides_64,  // const cuuint64_t *globalDim,
    &dims_strides_64,  // const cuuint64_t *globalStrides,
    &dims_32,           // const cuuint32_t *boxDim,
    &elem_strides,      // const cuuint32_t *elementStrides,
    CUtensorMapInterleave::CU_TENSOR_MAP_INTERLEAVE_NONE,
    CUtensorMapSwizzle::CU_TENSOR_MAP_SWIZZLE_NONE,
    CUtensorMapL2promotion::CU_TENSOR_MAP_L2_PROMOTION_NONE,
    CUtensorMapFloatOOBfill::CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE);

  CU_CHECK(res);
  return template_tensor_map;
}
```

4.11.2.2.5 Shared-Memory Bank Swizzling

By default, the TMA engine loads data to shared memory in the same order as it is laid out in global memory. However, this layout may not be optimal for certain shared memory access patterns, as it could cause shared memory bank conflicts. To improve performance and reduce bank conflicts, we can change the shared memory layout by applying a 'swizzle pattern'.

Shared memory has 32 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per clock cycle. When loading and storing shared memory, bank conflicts arise if the same bank is used multiple times within a transaction, resulting in reduced bandwidth. See *Shared Memory Access Patterns*.

To ensure that data is laid out in shared memory in such a way that user code can avoid shared memory bank conflicts, the TMA engine can be instructed to 'swizzle' the data before storing it in shared memory and 'unswizzle' it when copying the data back from shared memory to global memory. The tensor map encodes the 'swizzle mode' indicating which swizzle pattern is used.

Example: Matrix Transpose

An example is the transpose of a matrix where data is mapped from row to column first access. The data is stored row major in global memory, but we want to also access it column wise in shared memory, which leads to bank conflicts. However, by using the 128 bytes 'swizzle' mode and new shared memory indices, they are eliminated.

In the example, we load an 8x8 matrix of type `int4`, stored as row major in global memory to shared memory. Then, each set of eight threads loads a row from the shared memory buffer and stores it to a column in a separate transpose shared memory buffer. This results in an eight-way bank conflict when storing. Finally, the transpose buffer is written back to global memory.

To avoid bank conflicts, the `CU_TENSOR_MAP_SWIZZLE_128B` layout can be used. This layout matches the 128 bytes row length and changes the shared memory layout in a way that both the column wise and row wise access don't require the same banks per transaction.

The two tables, Figure 48 and Figure 49, below show the normal and the swizzled shared memory layout of the 8x8 matrix of type `int4` and its transpose matrix. The colors indicate which of the eight groups of four banks the matrix element is mapped to, and the margin row and margin column list the global memory row and column indices. The entries show the shared memory indices of the 16-byte matrix elements.

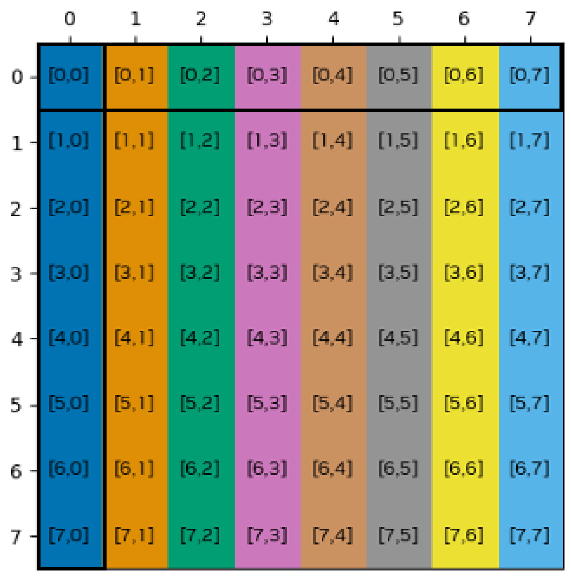


Figure 48: In the shared memory data layout without swizzle, the shared memory indices are equivalent to the global memory indices. Per load instruction, one row is read and stored in a column of the transpose buffer. Since all matrix elements of the column in the transpose fall in the same bank, the store must be serialized, resulting in eight store transactions, giving an eight-way bank conflict per stored column.

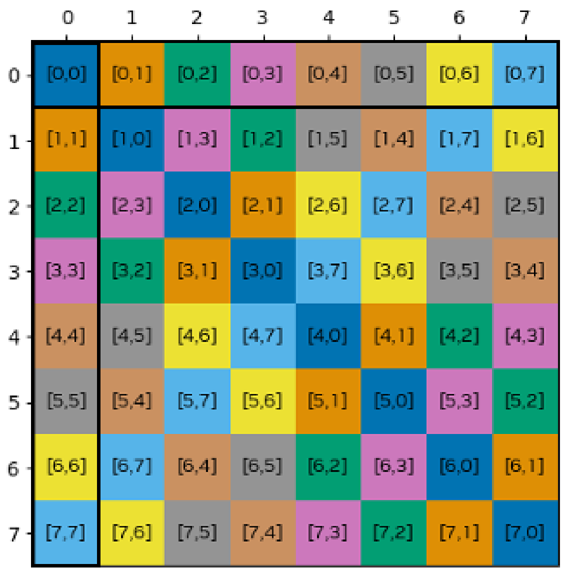


Figure 49: The shared memory data layout with CU_TENSOR_MAP_SWIZZLE_128B swizzle. One row is stored in a column, each matrix element is from a different bank for both the rows and columns, and so without any bank conflicts.

```
__global__ void kernel_tma(const __grid_constant__ CUtensorMap tensor_map) {
   // The destination shared memory buffer of a bulk tensor operation
   // with the 128-byte swizzle mode, it should be 1024 bytes aligned.
   __shared__ alignas(1024) int4 smem_buffer[8][8];
   __shared__ alignas(1024) int4 smem_buffer_tr[8][8];

   // Initialize shared memory barrier
   #pragma nv_diag_suppress static_var_with_dynamic_init
   __shared__ barrier bar;

   if (threadIdx.x == 0) {
     init(&bar, blockDim.x);
   }
   __syncthreads();

   barrier::arrival_token token;
   if (is_elected()) {
     // Initiate bulk tensor copy from global to shared memory,
     // in the same way as without swizzle.
     int32_t tensor_coords[2] = { 0, 0 };
     ptx::cp_async_bulk_tensor(
       ptx::space_shared, ptx::space_global,
       &smem_buffer, &tensor_map, tensor_coords,
       cuda::device::barrier_native_handle(bar));
     token = cuda::device::barrier_arrive_tx(bar, 1, sizeof(smem_buffer));
```

(continues on next page)

(continued from previous page)

```
   } else {
     token = bar.arrive();
   }

   bar.wait(std::move(token));

   /* Matrix transpose
    *  When using the normal shared memory layout, there are eight
    *  8-way shared memory bank conflict when storing to the transpose.
    *  When enabling the 128-byte swizzle pattern and using the according
↪access pattern,
    *  they are eliminated both for load and store. */
   for(int sidx_j =threadIdx.x; sidx_j < 8; sidx_j+= blockDim.x){
      for(int sidx_i = 0; sidx_i < 8; ++sidx_i){
         const int swiz_j_idx = (sidx_i % 8) ^ sidx_j;
         const int swiz_i_idx_tr = (sidx_j % 8) ^ sidx_i;
         smem_buffer_tr[sidx_j][swiz_i_idx_tr] = smem_buffer[sidx_i][swiz_j_
↪idx];
      }
   }

   // Wait for shared memory writes to be visible to TMA engine.
   ptx::fence_proxy_async(ptx::space_shared);
   __syncthreads();

   /* Initiate TMA transfer to copy the transposed shared memory buffer back
↪to global memory,
    * it will 'unswizzle' the data. */
   if (is_elected()) {
       int32_t tensor_coords[2] = { x, y };
       ptx::cp_async_bulk_tensor(
         ptx::space_global, ptx::space_shared,
         &tensor_map, tensor_coords, &smem_buffer_tr);
      ptx::cp_async_bulk_commit_group();
      ptx::cp_async_bulk_wait_group_read(ptx::n32_t<0>());
   }

   // Destroy barrier
   if (threadIdx.x == 0) {
     (&bar)->~barrier();
   }
}

// --------------------------------- main -------------------------------------
↪--

int main(){

...
   void* tensor_ptr = d_data;

   CUtensorMap tensor_map{};
```

(continues on next page)

(continued from previous page)

```
    // rank is the number of dimensions of the array.
    constexpr uint32_t rank = 2;
    // global memory size
    uint64_t size[rank] = {4*8, 8};
    // global memory stride, must be a multiple of 16.
    uint64_t stride[rank - 1] = {8 * sizeof(int4)};
    // The inner shared memory box dimension in bytes, equal to the swizzle
↪span.
    uint32_t box_size[rank] = {4*8, 8};

    uint32_t elem_stride[rank] = {1, 1};

    // Create the tensor descriptor.
    CUresult res = cuTensorMapEncodeTiled(
        &tensor_map,                 // CUtensorMap *tensorMap,
        CUtensorMapDataType::CU_TENSOR_MAP_DATA_TYPE_INT32,
        rank,                        // cuuint32_t tensorRank,
        tensor_ptr,                  // void *globalAddress,
        size,                        // const cuuint64_t *globalDim,
        stride,                      // const cuuint64_t *globalStrides,
        box_size,                    // const cuuint32_t *boxDim,
        elem_stride,                 // const cuuint32_t *elementStrides,
        CUtensorMapInterleave::CU_TENSOR_MAP_INTERLEAVE_NONE,
        // Using a swizzle pattern of 128 bytes.
        CUtensorMapSwizzle::CU_TENSOR_MAP_SWIZZLE_128B,
        CUtensorMapL2promotion::CU_TENSOR_MAP_L2_PROMOTION_NONE,
        CUtensorMapFloatOOBfill::CU_TENSOR_MAP_FLOAT_OOB_FILL_NONE
    );

    kernel_tma<<<1, 8>>>(tensor_map);
  ...
}
```

Remark. This example is supposed to show the use of swizzle and 'as-is' is not performant nor does it scale beyond the given dimensions.

Explanation. During data transfer, the TMA engine shuffles the data according to the swizzle pattern, as described in the following tables. These swizzle patterns define the mapping of the 16-byte chunks along the swizzle width to subgroups of four banks. It is of type `CUtensorMapSwizzle` and has four options: none, 32 bytes, 64 bytes and 128 bytes. Note that the shared memory box's inner dimension must be less or equal to the span of the swizzle pattern.

The Swizzle Modes

As previously mentioned, there are four swizzle modes. The following tables show the different swizzle patterns, including the relation of the new shared memory indices. The tables define the mapping of the 16-byte chunks along the 128 bytes to eight subgroups of four banks.

Considerations. When applying a TMA swizzle pattern, it is crucial to adhere to specific memory requirements:

- **Global memory alignment:** Global memory must be aligned to 128 bytes.
- **Shared memory alignment:** For simplicity shared memory should be aligned according to the number of bytes after which the swizzle pattern repeats. When the shared memory buffer is

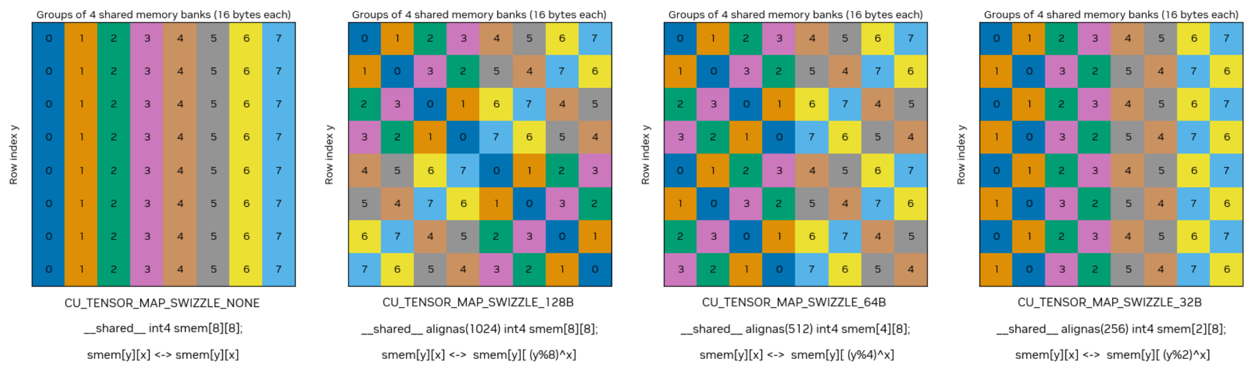

Groups of 4 shared memory banks (16 bytes each)
Row index y
CU_TENSOR_MAP_SWIZZLE_NONE
__shared__ int4 smem[8][8];
smem[y][x] <-> smem[y][x]
CU_TENSOR_MAP_SWIZZLE_128B
__shared__ alignas(1024) int4 smem[8][8];
smem[y][x] <-> smem[y][(y%8)^x]
CU_TENSOR_MAP_SWIZZLE_64B
__shared__ alignas(512) int4 smem[4][8];
smem[y][x] <-> smem[y][(y%4)^x]
CU_TENSOR_MAP_SWIZZLE_32B
__shared__ alignas(256) int4 smem[2][8];
smem[y][x] <-> smem[y][(y%2)^x]

Figure 50: An Overview of TMA Swizzle Patterns

not aligned by the number of bytes by which the swizzle pattern repeats itself, there is an offset between the swizzle pattern and the shared memory. See *comment*, below.

- **Inner dimension:** The inner dimension of the shared memory block must meet the size requirements specified in Table 25. If these requirements are not met, the instruction is considered invalid. Additionally, if the swizzle width exceeds the inner dimension, ensure that the shared memory is allocated to accommodate the full swizzle width.
- **Granularity:** The granularity of swizzle mapping is fixed at 16 bytes. This means that data is organized and accessed in chunks of 16 bytes, which must be considered when planning memory layout and access patterns.

Swizzle Pattern Pointer Offset Computation. Here, we describe how to determine the offset between the swizzle pattern and the shared memory, when the shared memory buffer is not aligned by the number of bytes by which the swizzle pattern repeats itself. When using TMA, the shared memory is required to be aligned to 128 bytes. To find how many times the shared memory buffer relative to the swizzle pattern is shifted by that, apply the corresponding offset formula.

Table 24: Swizzle Pattern Pointer Offset Formula and Index Relation

Swizzle Mode	Offset Formula	Index Relation
CU_TENSOR_MAP_SWI	`(reinterpret_cast <uintptr_t>(smem_ptr)/128)%8`	`smem[y][x] <-> smem[y][((y+offset)%8)^x]`
CU_TENSOR_MAP_SWI	`(reinterpret_cast <uintptr_t>(smem_ptr)/128)%4`	`smem[y][x] <-> smem[y][((y+offset)%4)^x]`
CU_TENSOR_MAP_SWI	`(reinterpret_cast <uintptr_t>(smem_ptr)/128)%2`	`smem[y][x] <-> smem[y][((y+offset)%2)^x]`

In Figure 50, this offset represents the initial row offset, thus, in the swizzle index calculation, it is added to the row index y. The following snippet shows how to access the swizzled shared memory in the CU_TENSOR_MAP_SWIZZLE_128B mode.

```
data_t* smem_ptr = &smem[0][0];
int offset = (reinterpret_cast<uintptr_t>(smem_ptr)/128)%8;
smem[y][((y+offset)%8)^x] = ...
```

Summary. The following Table 25 summarizes the requirements and properties of the different swizzle patterns for Compute Capability 9.

Table 25: Requirements and properties of the different swizzle patterns for Compute Capability 9

Pattern	Swizzle width	Shared box's inner dimension	Repeats after	Shared memory alignment	Global memory alignment
CU_TENSOR_MAP_SWIZZI	128 bytes	<=128 bytes	1024 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZI	64 bytes	<=64 bytes	512 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZI	32 bytes	<=32 bytes	256 bytes	128 bytes	128 bytes
CU_TENSOR_MAP_SWIZZI (default)				128 bytes	16 bytes

4.11.3. Using STAS

CUDA applications using *thread block clusters* may need to move small data elements between thread blocks within the cluster. STAS instructions (CC 9.0+, see PTX documentation) enable asynchronous data copies directly from registers to distributed shared memory. STAS is only exposed through a lower-level `cuda::ptx::st_async` API available in the libcu++ library.

Dimensions. STAS supports copying 4, 8 or 16 bytes.

Source and destination. The only direction supported for asynchronous copy operations with STAS is from registers to distributed shared memory. The destination pointer needs to be aligned to 4, 8, or 16 bytes depending on the size of the data being copied.

Asynchronicity. Data transfers using STAS are *asynchronous* and are modeled as async thread operations (see *Async Thread and Async Proxy*). This allows the initiating thread to continue computing while the hardware asynchronously copies the data. *Whether the data transfer occurs asynchronously in practice is up to the hardware implementation and may change in the future*. The completion mechanisms that STAS operations can use to signal that they have completed are *shared memory barriers*.

In the following example, we show how to use STAS to implement a producer-consumer pattern within a thread-block cluster. This kernel creates a circular communication pipeline where 8 thread blocks are arranged in a ring, and each block simultaneously:

- Produces data for the next block in the sequence.
- Consumes data from the previous block in the sequence.

To implement this pattern, we need 2 shared memory barriers per thread block, one to notify the consumer block that the data has been copied to the shared memory buffer (`filled`) and one to notify the producer block that the buffer on the consumer is ready to be filled (`ready`).

CUDA C++ cuda::ptx

```
#include <cooperative_groups.h>
#include <cuda/barrier>
#include <cuda/ptx>

__global__ __cluster_dims__(8, 1, 1) void producer_consumer_kernel()
{
    using namespace cooperative_groups;
    using namespace cuda::device;
    using namespace cuda::ptx;
    using barrier_t = cuda::barrier<cuda::thread_scope_block>;

    auto cluster = this_cluster();

    #pragma nv_diag_suppress static_var_with_dynamic_init
    __shared__ int buffer[BLOCK_SIZE];
    __shared__ barrier_t filled;
    __shared__ barrier_t ready;

    // Initialize shared memory barriers.
    if (threadIdx.x == 0) {
        init(&filled, 1);
        init(&ready, BLOCK_SIZE);
    }

    // Sync cluster to ensure remote barriers are initialized.
    cluster.sync();

    // Define my own and my neighbor's ranks.
    int rk = cluster.block_rank();
    int rk_next = (rk + 1) % 8;
    int rk_prev = (rk + 7) % 8;

    // Get addresses of remote buffer we are writing to and remote barriers
↪of previous and next blocks.
    auto buffer_next = cluster.map_shared_rank(buffer, rk_next);
    auto bar_next = cluster.map_shared_rank(barrier_native_handle(filled),
↪rk_next);
    auto bar_prev = cluster.map_shared_rank(barrier_native_handle(ready),
↪rk_prev);

    int phase = 0;
    for (int it = 0; it < 1000; ++it) {

        // As producers, send data to our right neighbor.
        st_async(&buffer_next[threadIdx.x], rk, bar_next);

        if (threadIdx.x == 0) {
            // Thread 0 arrives on local barrier and indicates it expects to
↪receive a certain number of bytes.
            mbarrier_arrive_expect_tx(sem_release, scope_cluster, space_
↪shared, barrier_native_handle(filled), sizeof(buffer));
        }
```

```
        // As consumers, wait on local barrier for data from left neighbor to
↪arrive.
        while (!mbarrier_try_wait_parity(barrier_native_handle(filled),
```

- Shared memory barriers are initialized by the first thread of each block. Barrier `filled` is initialized to 1 and barrier `ready` is initialized to the number of threads in the block.
- A cluster-wide synchronization is performed to ensure that all barriers are initialized before any thread starts communication.
- Each thread determines its neighbors' ranks and uses them to map the remote shared memory barriers and the remote shared memory buffer to write data to.
- In each iteration:
  1. As a producer, each thread sends data to its right neighbor.
  2. As a consumer, thread 0 arrives on the local `filled` barrier and indicates it expects to receive a certain number of bytes.
  3. As a consumer, each thread waits on the local `filled` barrier for data from the left neighbor to arrive.
  4. As a consumer, each thread uses the data to do something.
  5. As a consumer, each thread notifies the left neighbor that it is done with the data.
  6. As a producer, each thread waits on the local `ready` barrier until the right neighbor is ready to receive new data.

Note that for each barrier, we need to use the correct space. For mapped remote barriers, we need to use the `space_cluster` space, while for local barriers, we need to use the `space_shared` space.

4.12. Work Stealing with Cluster Launch Control

Dealing with problems of variable data and computation sizes is essential when developing CUDA applications. Traditionally, CUDA developers have used two main approaches to determine the number of kernel thread blocks to launch: *fixed work per thread block* and *fixed number of thread blocks*. Both approaches have their advantages and disadvantages.

Fixed Work per Thread Block: In this approach, the number of thread blocks is determined by the problem size, while the amount of work done by each thread block remains constant.

Key advantages of this approach:

- *Load balancing between SMs*

  When thread block run-times exhibit variability and/or when the number of thread blocks is much larger than what the GPU can execute simultaneously (resulting in a low-tail effect), this approach allows the GPU scheduler to run more thread blocks on some SMs than others.

- *Preemption*

  The GPU scheduler can start executing a *higher-priority kernel*, even if it is launched after a lower-priority kernel has already begun executing, by scheduling its thread blocks as thread blocks of the lower-priority kernel complete. It can then resume execution of the lower-priority kernel once the higher-priority kernel has finished executing.

Fixed Number of Thread Blocks: In this approach, often implemented as a block-stride or grid-stride loop, the number of thread blocks does not depend on the problem size. Instead, the amount of work done by each thread block is a function of the problem size. Typically, the number of thread blocks is based on the number of SMs on the GPU where the kernel is executed and the desired occupancy.

Key advantages of this approach:

- *Reduced thread block overheads*

  This approach not only reduces amortized thread block launch latency but also minimizes the computational overhead associated with shared operations across all thread blocks. These overheads can be significantly higher than launch latency overheads.

  For example, in convolution kernels, a prologue for calculating convolution coefficients – independent of the thread block index – can be computed fewer times due to the fixed number of thread blocks, thus reducing redundant computations.

Cluster Launch Control is a feature introduced in the NVIDIA Blackwell GPU architecture (compute capability 10.0) that aims to combine the benefits of the previous two approaches. It provides developers with more control over thread block scheduling by allowing them to cancel thread blocks or thread block clusters. This mechanism enables work stealing. Work stealing is a dynamic load-balancing technique in parallel computing where idle processors actively "steal" tasks from the work queues of busy processors, rather than wait for work to be assigned.

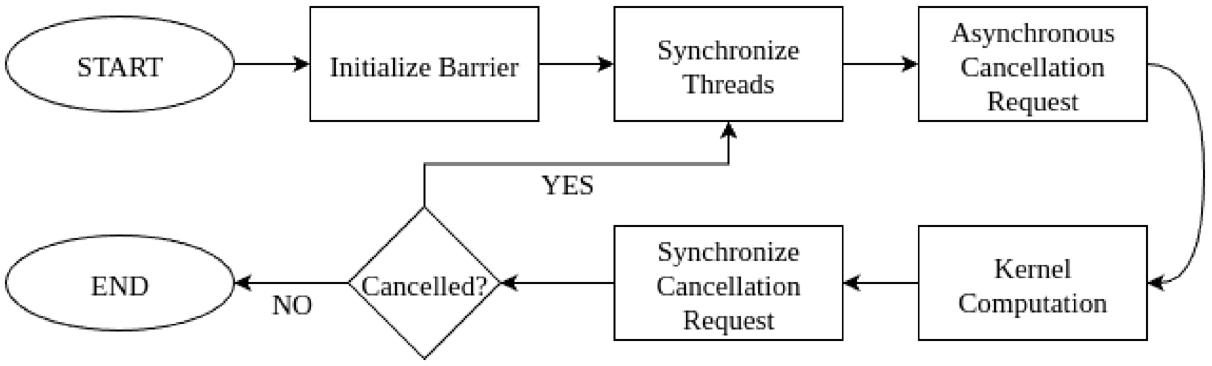


Figure 51: Cluster Launch Control Flow

With cluster launch control, a thread block attempts to cancel the launch of another thread block that has not started executing yet. If the cancellation request succeeds, it "steals" the other thread block's work by using its index to perform the task. The cancellation will fail if there are no more thread block indices available or for other reasons, such as a higher-priority kernel being scheduled. In the latter case, if a thread block exits after a cancellation failure, the scheduler can start executing the higher-priority kernel, after which it will continue scheduling the remaining thread blocks of the current kernel for execution. The *figure* above presents the execution flow of this procedure.

The table below summarizes advantages and disadvantages of the three approaches:

	Fixed Work per Thread Block	Fixed Number of Thread Blocks	Cluster Launch Control
Reduced overheads	X	V	V
Preemption	V	X	V
Load balancing	V	X	V

4.12.1. API Details

Cancelling a thread block via the cluster launch control API is done asynchronously and synchronized using a shared memory barrier, following a programming pattern similar to *asynchronous data copies*.

The API, available through libcu++, provides:

- A request instruction that writes encoded cancellation results to a `__shared__` variable.
- Decoding instructions that extract success/failure status and the cancelled thread block index.

Note that cluster launch control operations are modeled as async proxy operations (see *Async Thread and Async Proxy*).

4.12.1.1 Thread Block Cancellation

The preferred way to use Cluster Launch Control is from a single thread, i.e., one request at a time.

The cancellation process involves five steps:

- **Setup Phase** (Steps 1-2): Declare and initialize cancellation result and synchronization variables.
- **Work-Stealing Loop** (Steps 3-5): Execute repeatedly to request, synchronize, and process cancellation results.

1. Declare variables for thread block cancellation:

```
__shared__ uint4 result; // Request result.
__shared__ uint64_t bar; // Synchronization barrier.
int phase = 0;           // Synchronization barrier phase.
```

2. Initialize shared memory barrier with a single arrival count:

```
if (cg::thread_block::thread_rank() == 0)
    ptx::mbarrier_init(&bar, 1);
__syncthreads();
```

3. Submit asynchronous cancellation request by a single thread and set transaction count:

```
if (cg::thread_block::thread_rank() == 0) {
    cg::invoke_one(cg::coalesced_threads(), [&]()
↪{ptx::clusterlaunchcontrol_try_cancel(&result, &bar);});
    ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_cta,
↪ptx::space_shared, &bar, sizeof(uint4));
}
```

Note

Since thread block cancellation is a uniform instruction, it is recommended to submit it inside *invoke_one* thread selector. This allows the compiler to optimize out the peeling loop.

4. Synchronize (complete) asynchronous cancellation request:

```
while (!ptx::mbarrier_try_wait_parity(&bar, phase))
{}
phase ^= 1;
```

5. Retrieve cancellation status and cancelled thread block index:

```
bool success = ptx::clusterlaunchcontrol_query_cancel_is_canceled(result);
if (success) {
    // Don't need all three for 1D/2D thread blocks:
    int bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_
↪x(result);
    int by = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_
↪y(result);
    int bz = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_
↪z(result);
}
```

6. Ensure visibility of shared memory operations between async and generic proxies, and protect against data races between iterations of the work-stealing loop.

4.12.1.2 Constraints on Thread Block Cancellation

The constraints are related to failed cancellation requests:

- Submitting another cancellation request after **observing** a previously failed request is *undefined behavior*.

  In the two code examples below, assuming the first cancellation request fails, only the first example exhibits undefined behavior. The second example is correct because there is no observation between the cancellation requests:

  Invalid code:

```
// First request:
ptx::clusterlaunchcontrol_try_cancel(&result0, &bar0);

// First request query:
[Synchronize bar0 code here.]
bool success0 = ptx::clusterlaunchcontrol_query_cancel_is_
↪canceled(result0);
assert(!success0); // Observed failure; second cancellation will be
↪invalid.

// Second request - next line is Undefined Behavior:
ptx::clusterlaunchcontrol_try_cancel(&result1, &bar1);
```

  Valid code:

```
// First request:
ptx::clusterlaunchcontrol_try_cancel(&result0, &bar0);

// Second request:
ptx::clusterlaunchcontrol_try_cancel(&result1, &bar1);

// First request query:
[Synchronize bar0 code here.]
bool success0 = ptx::clusterlaunchcontrol_query_cancel_is_
↪canceled(result0);
assert(!success0); // Observed failure; second cancellation was valid.
```

- Retrieving the thread block index of a failed cancellation request is Undefined Behavior.
- Submitting a cancellation request from multiple threads is not recommended. It results in the cancellation of multiple thread blocks and requires careful handling, such as:
  - Each submitting thread must provide a unique `__shared__` result pointer to avoid data races.
  - If the same barrier is used for synchronization, the arrival and transaction counts must be adjusted accordingly.

4.12.2. Example: Vector-Scalar Multiplication

In the following subsections, we demonstrate work stealing through cluster launch control with a vector-scalar multiplication kernel. We show two variants of the same problem: one using thread blocks and one using thread block clusters.

4.12.2.1 Use-case: Thread Blocks

The three kernels below demonstrate the *Fixed Work per Thread Block*, *Fixed Number of Thread Blocks*, and *Cluster Launch Control* approaches for vector-scalar multiplication $\overline{v} := \alpha\overline{v}$.

- Fixed Work per Thread Block:

```
__global__
void kernel_fixed_work (float* data, int n)
{
    // Prologue:
    float alpha = compute_scalar();

    // Computation:
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n)
        data[i] *= alpha;
}

// Launch: kernel_fixed_work<<<1024, (n + 1023) / 1024>>>(data, n);
```

- Fixed Number of Thread Blocks:

```
__global__
void kernel_fixed_blocks (float* data, int n)
{
    // Prologue:
    float alpha = compute_scalar();

    // Computation:
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    while (i < n) {
        data[i] *= alpha;
        i += gridDim.x * blockDim.x;
    }
}

// Launch: kernel_fixed_blocks<<<1024, SM_COUNT>>>(data, n);
```

▶ Cluster Launch Control:

```
#include <cooperative_groups.h>
#include <cuda/ptx>

namespace cg = cooperative_groups;
namespace ptx = cuda::ptx;

__global__
void kernel_cluster_launch_control (float* data, int n)
{
    // Cluster launch control initialization:
    __shared__ uint4 result;
    __shared__ uint64_t bar;
    int phase = 0;

    if (cg::thread_block::thread_rank() == 0)
        ptx::mbarrier_init(&bar, 1);

    // Prologue:
    float alpha = compute_scalar(); // Device function not shown in this
↪code snippet.

    // Work-stealing loop:
    int bx = blockIdx.x; // Assuming 1D x-axis thread blocks.

    while (true) {
        // Protect result from overwrite in the next iteration,
        // (also ensure barrier initialization at 1st iteration):
        __syncthreads();

        // Cancellation request:
        if (cg::thread_block::thread_rank() == 0) {
            // Acquire write of result in the async proxy:
            ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire,
↪ ptx::space_cluster, ptx::scope_cluster);

            cg::invoke_one(cg::coalesced_threads(), [&]()
↪{ptx::clusterlaunchcontrol_try_cancel(&result, &bar);});
            ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_
↪cta, ptx::space_shared, &bar, sizeof(uint4));
        }

        // Computation:
        int i = bx * blockDim.x + threadIdx.x;
        if (i < n)
            data[i] *= alpha;

        // Cancellation request synchronization:
        while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire,
↪ptx::scope_cta, &bar, phase))
        {}
        phase ^= 1;
```

(continues on next page)

(continued from previous page)

```
        // Cancellation request decoding:
        bool success = ptx::clusterlaunchcontrol_query_cancel_is_
↪canceled(result);
        if (!success)
            break;

        bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x<int>
↪(result);

        // Release read of result to the async proxy:
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release,
↪ptx::space_shared, ptx::scope_cluster);
    }
}

// Launch: kernel_cluster_launch_control<<<1024, (n + 1023) / 1024>>>(data,
↪ n);
```

4.12.2.2 Use-case: Thread Block Clusters

In the case of a *thread block clusters*, the thread block cancellation steps are the same as in a non-cluster setting, with minor adjustments. As in the non-cluster case, submitting a cancellation request from multiple threads **within a cluster** is not recommended, as this will attempt to cancel multiple clusters.

- The cancellation is submitted by a single cluster thread.
- The shared memory result of each cluster's thread block will receive the same (encoded) value of the cancelled thread block index (i.e., the result value is multicasted). The result received by all thread blocks corresponds to the local block index `{0, 0, 0}` within a cluster. Therefore, thread blocks within the cluster need to add the local block index.
- Synchronization is performed by each cluster's thread block using a local `__shared__` memory barrier. Barrier operations must be performed with the `ptx::scope_cluster` scope.
- Cancelling in the cluster case requires all the thread blocks to exist. A user can guarantee that all thread blocks are running by using `cg::cluster_group::sync()` from *sync* API.

The kernel below demonstrates the cluster launch control approach using thread block clusters.

```
#include <cooperative_groups.h>
#include <cuda/ptx>

namespace cg = cooperative_groups;
namespace ptx = cuda::ptx;

__global__ __cluster_dims__(2, 1, 1)
void kernel_cluster_launch_control (float* data, int n)
{
    // Cluster launch control initialization:
    __shared__ uint4 result;
    __shared__ uint64_t bar;
    int phase = 0;
```

(continues on next page)

(continued from previous page)

```
    if (cg::thread_block::thread_rank() == 0) {
        ptx::mbarrier_init(&bar, 1);
        ptx::fence_mbarrier_init(ptx::sem_release, ptx::scope_cluster); //
↪CGA-level fence.
    }

    // Prologue:
    float alpha = compute_scalar(); // Device function not shown in this code
↪snippet.

    // Work-stealing loop:
    int bx = blockIdx.x; // Assuming 1D x-axis thread blocks.

    while (true) {
        // Protect result from overwrite in the next iteration,
        // (also ensure all thread blocks have started at 1st iteration):
        cg::cluster_group::sync();

        // Cancellation request by a single cluster thread:
        if (cg::cluster_group::thread_rank() == 0) {
            // Acquire write of result in the async proxy:
            ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_acquire,
↪ptx::space_cluster, ptx::scope_cluster);

            cg::invoke_one(cg::coalesced_threads(), [&]()
↪{ptx::clusterlaunchcontrol_try_cancel_multicast(&result, &bar);});
        }

        // Cancellation completion tracked by each thread block:
        if (cg::thread_block::thread_rank() == 0)
            ptx::mbarrier_arrive_expect_tx(ptx::sem_relaxed, ptx::scope_
↪cluster, ptx::space_shared, &bar, sizeof(uint4));

        // Computation:
        int i = bx * blockDim.x + threadIdx.x;
        if (i < n)
            data[i] *= alpha;

        // Cancellation request synchronization:
        while (!ptx::mbarrier_try_wait_parity(ptx::sem_acquire, ptx::scope_
↪cluster, &bar, phase))
        {}
        phase ^= 1;

        // Cancellation request decoding:
        bool success = ptx::clusterlaunchcontrol_query_cancel_is_
↪canceled(result);
        if (!success)
            break;

        bx = ptx::clusterlaunchcontrol_query_cancel_get_first_ctaid_x<int>
```

(continues on next page)

(continued from previous page)

```
↪(result);
        bx += cg::cluster_group::block_index().x; // Add local offset.

        // Release read of result to the async proxy:
        ptx::fence_proxy_async_generic_sync_restrict(ptx::sem_release,
↪ptx::space_shared, ptx::scope_cluster);
    }
}

// Launch: kernel_cluster_launch_control<<<1024, (n + 1023) / 1024>>>(data, n);
```

4.13. L2 Cache Control

When a CUDA kernel accesses a data region in the global memory repeatedly, such data accesses can be considered to be persisting. On the other hand, if the data is only accessed once, such data accesses can be considered to be streaming.

Devices of compute capability 8.0 and above have the capability to influence persistence of data in the L2 cache, potentially providing higher bandwidth and lower latency accesses to global memory.

This functionality is exposed through two main APIs:

- The CUDA runtime API (starting with CUDA 11.0) provides programmatic control over L2 cache persistence.
- The `cuda::annotated_ptr` API in the libcu++ library (starting with CUDA 11.5) annotates pointers in CUDA kernels with memory access properties to achieve a similar effect..

The following sections focus on the CUDA runtime API. For detailed information about the `cuda::annotated_ptr` approach, please refer to the libcu++ documentation.

4.13.1. L2 Cache Set-Aside for Persisting Accesses

A portion of the L2 cache can be set aside to be used for persisting data accesses to global memory. Persisting accesses have prioritized use of this set-aside portion of L2 cache, whereas normal or streaming accesses to global memory can only utilize this portion of L2 when it is unused by persisting accesses.

The L2 cache set-aside size for persisting accesses may be adjusted, within limits:

```
cudaGetDeviceProperties(&prop, device_id);
size_t size = min(int(prop.l2CacheSize * 0.75), prop.
↪persistingL2CacheMaxSize);
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, size); /* set-aside 3/4 of
↪L2 cache for persisting accesses or the max allowed*/
```

When the GPU is configured in Multi-Instance GPU (MIG) mode, the L2 cache set-aside functionality is disabled.

When using the Multi-Process Service (MPS), the L2 cache set-aside size cannot be changed by `cudaDeviceSetLimit`. Instead, the set-aside size can only be specified at start up of MPS server through the environment variable `CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT`.

4.13.2. L2 Policy for Persisting Accesses

An access policy window specifies a contiguous region of global memory and a persistence property in the L2 cache for accesses within that region.

The code example below shows how to set an L2 persisting access window using a CUDA Stream.

CUDA Stream Example

```
cudaStreamAttrValue stream_attribute;
↪// Stream level attributes data structure
stream_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>(ptr);
↪// Global Memory data pointer
stream_attribute.accessPolicyWindow.num_bytes = num_bytes;
↪// Number of bytes for persistence access.

↪// (Must be less than cudaDeviceProp::accessPolicyMaxWindowSize)
stream_attribute.accessPolicyWindow.hitRatio  = 0.6;
↪// Hint for cache hit ratio
stream_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting;
↪// Type of access property on cache hit
stream_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;
↪// Type of access property on cache miss.

//Set the attributes to a CUDA stream of type cudaStream_t
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_
↪attribute);
```

When a kernel subsequently executes in CUDA `stream`, memory accesses within the global memory extent `[ptr..ptr+num_bytes)` are more likely to persist in the L2 cache than accesses to other global memory locations.

L2 persistence can also be set for a CUDA Graph Kernel Node as shown in the example below:

CUDA GraphKernelNode Example

```
cudaKernelNodeAttrValue node_attribute;                                     //
↪ Kernel level attributes data structure
node_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>(ptr); //
↪ Global Memory data pointer
node_attribute.accessPolicyWindow.num_bytes = num_bytes;                    //
↪ Number of bytes for persistence access.
                                                                            //
↪ (Must be less than cudaDeviceProp::accessPolicyMaxWindowSize)
node_attribute.accessPolicyWindow.hitRatio  = 0.6;                          //
↪ Hint for cache hit ratio
node_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting; //
↪ Type of access property on cache hit
node_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;  //
↪ Type of access property on cache miss.

//Set the attributes to a CUDA Graph Kernel node of type cudaGraphNode_t
cudaGraphKernelNodeSetAttribute(node,
↪cudaKernelNodeAttributeAccessPolicyWindow, &node_attribute);
```

The `hitRatio` parameter can be used to specify the fraction of accesses that receive the `hitProp` property. In both of the examples above, 60% of the memory accesses in the global memory region [ptr..ptr+num_bytes) have the persisting property and 40% of the memory accesses have the streaming property. Which specific memory accesses are classified as persisting (the hitProp) is random with a probability of approximately hitRatio; the probability distribution depends upon the hardware architecture and the memory extent.

For example, if the L2 set-aside cache size is 16KB and the num_bytes in the accessPolicyWindow is 32KB:

- With a hitRatio of 0.5, the hardware will select, at random, 16KB of the 32KB window to be designated as persisting and cached in the set-aside L2 cache area.
- With a hitRatio of 1.0, the hardware will attempt to cache the whole 32KB window in the set-aside L2 cache area. Since the set-aside area is smaller than the window, cache lines will be evicted to keep the most recently used 16KB of the 32KB data in the set-aside portion of the L2 cache.

The hitRatio can therefore be used to avoid thrashing of cache lines and overall reduce the amount of data moved into and out of the L2 cache.

A hitRatio value below 1.0 can be used to manually control the amount of data different accessPolicyWindows from concurrent CUDA streams can cache in L2. For example, let the L2 set-aside cache size be 16KB; two concurrent kernels in two different CUDA streams, each with a 16KB accessPolicyWindow, and both with hitRatio value 1.0, might evict each others' cache lines when competing for the shared L2 resource. However, if both accessPolicyWindows have a hitRatio value of 0.5, they will be less likely to evict their own or each others' persisting cache lines.

4.13.3. L2 Access Properties

Three types of access properties are defined for different global memory data accesses:

1. cudaAccessPropertyStreaming: Memory accesses that occur with the streaming property are less likely to persist in the L2 cache because these accesses are preferentially evicted.
2. cudaAccessPropertyPersisting: Memory accesses that occur with the persisting property are more likely to persist in the L2 cache because these accesses are preferentially retained in the set-aside portion of L2 cache.
3. cudaAccessPropertyNormal: This access property forcibly resets previously applied persisting access property to a normal status. Memory accesses with the persisting property from previous CUDA kernels may be retained in L2 cache long after their intended use. This persistence-after-use reduces the amount of L2 cache available to subsequent kernels that do not use the persisting property. Resetting an access property window with the cudaAccessPropertyNormal property removes the persisting (preferential retention) status of the prior access, as if the prior access had been without an access property.

4.13.4. L2 Persistence Example

The following example shows how to set-aside L2 cache for persistent accesses, use the set-aside L2 cache in CUDA kernels via CUDA Stream and then reset the L2 cache.

```
cudaStream_t stream;
cudaStreamCreate(&stream);
↪                // Create CUDA stream
```

(continues on next page)

(continued from previous page)

```
cudaDeviceProp prop;
↪               // CUDA device properties variable
cudaGetDeviceProperties( &prop, device_id);
↪               // Query GPU properties
size_t size = min( int(prop.l2CacheSize * 0.75) , prop.
↪persistingL2CacheMaxSize );
cudaDeviceSetLimit( cudaLimitPersistingL2CacheSize, size);
↪               // set-aside 3/4 of L2 cache for persisting accesses or the
↪max allowed

size_t window_size = min(prop.accessPolicyMaxWindowSize, num_bytes);
↪               // Select minimum of user defined num_bytes and max window
↪size.

cudaStreamAttrValue stream_attribute;
↪               // Stream level attributes data structure
stream_attribute.accessPolicyWindow.base_ptr  = reinterpret_cast<void*>
↪(data1);                // Global Memory data pointer
stream_attribute.accessPolicyWindow.num_bytes = window_size;
↪               // Number of bytes for persistence access
stream_attribute.accessPolicyWindow.hitRatio  = 0.6;
↪               // Hint for cache hit ratio
stream_attribute.accessPolicyWindow.hitProp   = cudaAccessPropertyPersisting;
↪               // Persistence Property
stream_attribute.accessPolicyWindow.missProp  = cudaAccessPropertyStreaming;
↪               // Type of access property on cache miss

cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_
↪attribute);   // Set the attributes to a CUDA Stream

for(int i = 0; i < 10; i++) {
    cuda_kernelA<<<grid_size,block_size,0,stream>>>(data1);
↪               // This data1 is used by a kernel multiple times
}
↪               // [data1 + num_bytes) benefits from L2 persistence
cuda_kernelB<<<grid_size,block_size,0,stream>>>(data1);
↪               // A different kernel in the same stream can also benefit

↪               // from the persistence of data1

stream_attribute.accessPolicyWindow.num_bytes = 0;
↪               // Setting the window size to 0 disable it
cudaStreamSetAttribute(stream, cudaStreamAttributeAccessPolicyWindow, &stream_
↪attribute);   // Overwrite the access policy attribute to a CUDA Stream
cudaCtxResetPersistingL2Cache();
↪               // Remove any persistent lines in L2

cuda_kernelC<<<grid_size,block_size,0,stream>>>(data2);
↪               // data2 can now benefit from full L2 in normal mode
```

4.13.5. Reset L2 Access to Normal

A persisting L2 cache line from a previous CUDA kernel may persist in L2 long after it has been used. Hence, a reset to normal for L2 cache is important for streaming or normal memory accesses to utilize the L2 cache with normal priority. There are three ways a persisting access can be reset to normal status.

1. Reset a previous persisting memory region with the access property, `cudaAccessPropertyNormal`.
2. Reset all persisting L2 cache lines to normal by calling `cudaCtxResetPersistingL2Cache()`.
3. **Eventually** untouched lines are automatically reset to normal. Reliance on automatic reset is strongly discouraged because of the undetermined length of time required for automatic reset to occur.

4.13.6. Manage Utilization of L2 set-aside cache

Multiple CUDA kernels executing concurrently in different CUDA streams may have a different access policy window assigned to their streams. However, the L2 set-aside cache portion is shared among all these concurrent CUDA kernels. As a result, the net utilization of this set-aside cache portion is the sum of all the concurrent kernels' individual use. The benefits of designating memory accesses as persisting diminish as the volume of persisting accesses exceeds the set-aside L2 cache capacity.

To manage utilization of the set-aside L2 cache portion, an application must consider the following:

- Size of L2 set-aside cache.
- CUDA kernels that may concurrently execute.
- The access policy window for all the CUDA kernels that may concurrently execute.
- When and how L2 reset is required to allow normal or streaming accesses to utilize the previously set-aside L2 cache with equal priority.

4.13.7. Query L2 cache Properties

Properties related to L2 cache are a part of `cudaDeviceProp` struct and can be queried using CUDA runtime API `cudaGetDeviceProperties`

CUDA Device Properties include:

- `l2CacheSize`: The amount of available L2 cache on the GPU.
- `persistingL2CacheMaxSize`: The maximum amount of L2 cache that can be set-aside for persisting memory accesses.
- `accessPolicyMaxWindowSize`: The maximum size of the access policy window.

4.13.8. Control L2 Cache Set-Aside Size for Persisting Memory Access

The L2 set-aside cache size for persisting memory accesses is queried using CUDA runtime API `cudaDeviceGetLimit` and set using CUDA runtime API `cudaDeviceSetLimit` as a `cudaLimit`. The maximum value for setting this limit is `cudaDeviceProp::persistingL2CacheMaxSize`.

```
enum cudaLimit {
    /* other fields not shown */
    cudaLimitPersistingL2CacheSize
};
```

4.14. Memory Synchronization Domains

4.14.1. Memory Fence Interference

Some CUDA applications may see degraded performance due to memory fence/flush operations waiting on more transactions than those necessitated by the CUDA memory consistency model.

```
__managed__ int x = 0;
__device__  cuda::atomic
↪<int, cuda::thread_
↪scope_device> a(0);
__managed__ cuda::atomic
↪<int, cuda::thread_
↪scope_system> b(0);
```

Thread 1 (SM)	Thread 2 (SM)	Thread 3 (CPU)

`x = 1;`
`a = 1;`
	`while (a != 1) ;`
	`assert(x == 1);`
	`b = 1;`
		`while (b != 1) ;`
		`assert(x == 1);`

Consider the example above. The CUDA memory consistency model guarantees that the asserted condition will be true, so the write to x from thread 1 must be visible to thread 3, before the write to b from thread 2.

The memory ordering provided by the release and acquire of a is only sufficient to make x visible to thread 2, not thread 3, as it is a device-scope operation. The system-scope ordering provided by release and acquire of b, therefore, needs to ensure not only writes issued from thread 2 itself are visible to thread 3, but also writes from other threads that are visible to thread 2. This is known as cumulativity. As the GPU cannot know at the time of execution which writes have been guaranteed at the source level to be visible and which are visible only by chance timing, it must cast a conservatively wide net for in-flight memory operations.

This sometimes leads to interference: because the GPU is waiting on memory operations it is not required to at the source level, the fence/flush may take longer than necessary.

Note that fences may occur explicitly as intrinsics or atomics in code, like in the example, or implicitly to implement *synchronizes-with* relationships at task boundaries.

A common example is when a kernel is performing computation in local GPU memory, and a parallel kernel (e.g. from NCCL) is performing communications with a peer. Upon completion, the local kernel will implicitly flush its writes to satisfy any *synchronizes-with* relationships to downstream work. This may unnecessarily wait, fully or partially, on slower nvlink or PCIe writes from the communication kernel.

4.14.2. Isolating Traffic with Domains

Beginning with compute capability 9.0 (Hopper architecture) GPUs and CUDA 12.0, the memory synchronization domains feature provides a way to alleviate such interference. In exchange for explicit assistance from code, the GPU can reduce the net cast by a fence operation. Each kernel launch is given a domain ID. Writes and fences are tagged with the ID, and a fence will only order writes matching the fence's domain. In the concurrent compute vs communication example, the communication kernels can be placed in a different domain.

When using domains, code must abide by the rule that **ordering or synchronization between distinct domains on the same GPU requires system-scope fencing**. Within a domain, device-scope fencing remains sufficient. This is necessary for cumulativity as one kernel's writes will not be encompassed by a fence issued from a kernel in another domain. In essence, cumulativity is satisfied by ensuring that cross-domain traffic is flushed to the system scope ahead of time.

Note that this modifies the definition of `thread_scope_device`. However, because kernels will default to domain 0 as described below, backward compatibility is maintained.

4.14.3. Using Domains in CUDA

Domains are accessible via the new launch attributes `cudaLaunchAttributeMemSyncDomain` and `cudaLaunchAttributeMemSyncDomainMap`. The former selects between logical domains `cudaLaunchMemSyncDomainDefault` and `cudaLaunchMemSyncDomainRemote`, and the latter provides a mapping from logical to physical domains. The remote domain is intended for kernels performing remote memory access in order to isolate their memory traffic from local kernels. Note, however, the selection of a particular domain does not affect what memory access a kernel may legally perform.

The domain count can be queried via device attribute `cudaDevAttrMemSyncDomainCount`. Devices of compute capability 9.0 (Hopper) have 4 domains. To facilitate portable code, domains functionality can be used on all devices and CUDA will report a count of 1 on devices prior to compute capability 9.0.

Having logical domains eases application composition. An individual kernel launch at a low level in the stack, such as from NCCL, can select a semantic logical domain without concern for the surrounding application architecture. Higher levels can steer logical domains using the mapping. The default value for the logical domain if it is not set is the default domain, and the default mapping is to map the default domain to 0 and the remote domain to 1 (on GPUs with more than 1 domain). Specific libraries may tag launches with the remote domain in CUDA 12.0 and later; for example, NCCL 2.16 will do so. Together, this provides a beneficial use pattern for common applications out of the box, with no code changes needed in other components, frameworks, or at application level. An alternative use pattern, for example in an application using NVSHMEM or with no clear separation of kernel types, could be to partition parallel streams. Stream A may map both logical domains to physical domain 0, stream B to 1, and so on.

```
// Example of launching a kernel with the remote logical domain
cudaLaunchAttribute domainAttr;
domainAttr.id = cudaLaunchAttrMemSyncDomain;
domainAttr.val = cudaLaunchMemSyncDomainRemote;
cudaLaunchConfig_t config;
// Fill out other config fields
config.attrs = &domainAttr;
config.numAttrs = 1;
cudaLaunchKernelEx(&config, myKernel, kernelArg1, kernelArg2...);
```

```
// Example of setting a mapping for a stream
// (This mapping is the default for streams starting on compute capability 9.0
↪(Hopper) or later if not
// explicitly set, and provided for illustration)
cudaLaunchAttributeValue mapAttr;
mapAttr.memSyncDomainMap.default_ = 0;
mapAttr.memSyncDomainMap.remote = 1;
cudaStreamSetAttribute(stream, cudaLaunchAttributeMemSyncDomainMap, &mapAttr);
```

```
// Example of mapping different streams to different physical domains, ignoring
// logical domain settings
cudaLaunchAttributeValue mapAttr;
mapAttr.memSyncDomainMap.default_ = 0;
mapAttr.memSyncDomainMap.remote = 0;
cudaStreamSetAttribute(streamA, cudaLaunchAttributeMemSyncDomainMap, &
↪mapAttr);
mapAttr.memSyncDomainMap.default_ = 1;
mapAttr.memSyncDomainMap.remote = 1;
cudaStreamSetAttribute(streamB, cudaLaunchAttributeMemSyncDomainMap, &
↪mapAttr);
```

As with other launch attributes, these are exposed uniformly on CUDA streams, individual launches using `cudaLaunchKernelEx`, and kernel nodes in CUDA graphs. A typical use would set the mapping at stream level and the logical domain at launch level (or bracketing a section of stream use) as described above.

Both attributes are copied to graph nodes during stream capture. Graphs take both attributes from the node itself, essentially an indirect way of specifying a physical domain. Domain-related attributes set on the stream a graph is launched into are not used in execution of the graph.

4.15. Interprocess Communication

Communication between multiple GPUs managed by different host processes is supported through the use of interprocess communication (IPC) APIs and IPC-shareable memory buffers, by creating process-portable handles that are subsequently used to obtain process-local device pointers to the device memory on peer GPUs.

Any device memory pointer or event handle created by a host thread can be directly referenced by any other thread within the same process. However, device pointers or event handles are not valid outside the process that created them, and therefore cannot be directly referenced by threads belonging to a different process. To access device memory and CUDA events across processes, an application must use CUDA Interprocess Communication (IPC) or Virtual Memory Management APIs to create process-portable handles that can be shared with other processes using standard host operating system IPC mechanisms, e.g., interprocess shared memory or files. Once the process-portable handles have been exchanged between processes, process-local device pointers must be obtained from the handles using CUDA IPC or VMM APIs. Process-local device pointers can then be used just as they would within a single process.

The same kind of portable-handle approach used for IPC within a single-node and single operating system instance is also used for peer-to-peer communication among the GPUs in multi-node NVLink-connected clusters. In the multi-node case, communicating GPUs are managed by processes running within independent operating system instances on each cluster node, requiring additional abstraction above the level of operating system instances. Multi-node peer communication is achieved by creating and exchanging so-called “fabric” handles between multi-node GPU peers, and by then obtaining process-local device pointers within the participating processes and operating system instances corresponding to the multi-node ranks.

See below (single-node CUDA IPC) and ref::*virtual-memory-management* for the specific APIs used to establish and exchange process-portable and node and operating system instance-portable handles that are used to obtain process-local device pointers for GPU communication.

Note

There are individual advantages and limitations associated with the use of the CUDA IPC APIs and Virtual Memory Management (VMM) APIs when used for IPC.

The CUDA IPC API is only currently supported on Linux platforms.

The CUDA Virtual Memory Management APIs permit per-allocation control over peer accessibility and sharing at memory allocation time, but require the use of the CUDA Driver API.

4.15.1. IPC using the Legacy Interprocess Communication API

To share device memory pointers and events across processes, an application must use the CUDA Interprocess Communication API, which is described in detail in the reference manual. The IPC API permits an application to get the IPC handle for a given device memory pointer using `cudaIpcGetMemHandle()`. A CUDA IPC handle can be passed to another process using standard host operating system IPC mechanisms, e.g., interprocess shared memory or files. `cudaIpcOpenMemHandle()` uses the IPC handle to retrieve a valid device pointer that can be used within the other process. Event handles can be shared using similar entry points.

An example of using the IPC API is where a single primary process generates a batch of input data, making the data available to multiple secondary processes without requiring regeneration or copying.

Note

The IPC API is only supported on Linux.

Note that the IPC API is not supported for `cudaMallocManaged` allocations.

Applications using CUDA IPC to communicate with each other should be compiled, linked, and run with the same CUDA driver and runtime.

Allocations made by `cudaMalloc()` may be sub-allocated from a larger block of memory for performance reasons. In such case, CUDA IPC APIs will share the entire underlying memory block which may cause other sub-allocations to be shared, which can potentially lead to information disclosure between processes. To prevent this behavior, it is recommended to only share allocations with a 2MiB aligned size.

Only the IPC events-sharing APIs are supported on L4T and embedded Linux Tegra devices with compute capability 7.x and higher. The IPC memory-sharing APIs are not supported on Tegra platforms.

4.15.2. IPC using the Virtual Memory Management API

The CUDA Virtual Memory Management API allows the creation of IPC-shareable memory allocations, and it supports multiple operating systems by virtue of operating-system specific IPC handle data structures.

4.16. Virtual Memory Management

In the CUDA programming model, memory allocation calls (such as `cudaMalloc()`) return a memory address that in GPU memory. The address can be used with any CUDA API or inside a device kernel. Developers can enable peer device access to that memory allocations by using `cudaEnablePeerAccess`. By doing so, kernels on different devices can access the same data. However, all past and future user allocations are also mapped to the target peer device. This can lead to users unintentionally paying a runtime cost for mapping all `cudaMalloc` allocations to peer devices. In most situations, applications communicate by sharing only a few allocations with another device. It is usually not necessary to map all allocations to all devices. In addition, extending this approach to multi-node settings becomes inherently difficult.

CUDA provides a *virtual memory management* (VMM) API to give developers explicit, low-level control over this process.

Virtual memory allocation, a complex process managed by the operating system and the Memory Management Unit (MMU), works in two key stages. First, the OS reserves a contiguous range of virtual addresses for a program without assigning any physical memory. Then, when the program attempts to use that memory for the first time, the OS commits the virtual addresses, assigning physical storage to the virtual pages as needed.

CUDA's VMM API brings a similar concept to GPU memory management by allowing developers to explicitly reserve a virtual address range and then later map it to physical GPU memory. With VMM, applications can specifically choose certain allocations to be accessible by other devices.

The VMM API lets complex applications to manage memory more efficiently across multiple GPUs (and CPU cores). By enabling manual control over memory reservation, mapping, and access permissions, the VMM API enables advanced techniques like fine-grained data sharing, zero-copy transfers, and custom memory allocators. The CUDA VMM API expose fine grained control to the user for managing the GPU memory in applications.

Developers can benefit from the VMM API in several key ways:

- Fine-grained control over virtual and physical memory management, allowing allocation and mapping of non-contiguous physical memory chunks to contiguous virtual address spaces. This helps reduce GPU memory fragmentation and improve memory utilization, especially for large workloads like deep neural network training.
- Efficient memory allocation and deallocation by separating the reservation of virtual address space from the physical memory allocation. Developers can reserve large virtual memory regions and map physical memory on demand without costly memory copies or reallocations, leading to performance improvements in dynamic data structures and variable-sized memory allocations.
- The ability to grow GPU memory allocations dynamically without needing to copy and reallocate all data, similar to how `realloc` or `std::vector` works in CPU memory management. This supports more flexible and efficient GPU memory use patterns.
- Enhancements to developer productivity and application performance by providing low-level APIs that allow building sophisticated memory allocators and cache management systems, such as

dynamically managing key-value caches in large language models, improving throughput and latency.

- The CUDA VMM API is highly valuable in distributed multi-GPU settings as it enables efficient memory sharing and access across multiple GPUs. By decoupling virtual addresses from physical memory, the API allows developers to create a unified virtual address space where data can be dynamically mapped to different GPUs. This optimizes memory usage and reduces data transfer overhead. For instance, NVIDIA's libraries like NCCL, and NVShmem actively uses VMM.

In summary, the CUDA VMM API gives developers advanced tools for fine-tuned, efficient, flexible, and scalable GPU memory management beyond traditional malloc-like abstractions, which is important for high-performance and large-memory applications

Note

The suite of APIs described in this section require a system that supports UVA. See The Virtual Memory Management APIs.

4.16.1. Preliminaries

4.16.1.1 Definitions

Fabric Memory: Fabric memory refers to memory that is accessible over a high-speed interconnect fabric such as NVIDIA's NVLink and NVSwitch. This fabric provides a memory coherence and high-bandwidth communication layer between multiple GPUs or nodes, enabling them to share memory efficiently as if the memory is attached to a unified fabric rather than isolated on individual devices.

CUDA 12.4 and later have a VMM allocation handle type `CU_MEM_HANDLE_TYPE_FABRIC`. On supported platforms and provided the NVIDIA IMEX daemon is running, this allocation handle type enables sharing allocations not only intra-node with any communication mechanism, e.g. MPI, but also inter-node. This allows GPUs in a multi-node NVLink system to map the memory of all other GPUs part of the same NVLink fabric even if they are in different nodes.

Memory Handles: In VMM, handles are opaque identifiers that represent physical memory allocations. These handles are central to managing memory in the low-level CUDA VMM API. They enable flexible control over physical memory objects that can be mapped into virtual address spaces. A handle uniquely identifies a physical memory allocation. Handles serve as an abstract reference to memory resources without exposing direct pointers. Handles allow operations like exporting and importing memory across processes or devices, facilitating memory sharing and virtualization.

IMEX Channels: The name IMEX stands for *internode memory exchange* and is part of NVIDIA's solution for GPU-to-GPU communication across different nodes. IMEX channels are a GPU driver feature that provides user-based memory isolation in multi-user or multi-node environments within an IMEX domain. IMEX channels serve as a security and isolation mechanism.

IMEX channels are directly related to the fabric handle and has to be enabled in multi-node GPU communication. When a GPU allocates memory and wants to make it accessible to a GPU on a different node, it first needs to export a handle to that memory. The IMEX channel is used during this export process to generate a secure fabric handle that can only be imported by a remote process with the correct channel access.

Unicast Memory Access: Unicast memory access in the context of VMM API refers to the controlled, direct mapping and access of physical memory to a unique virtual address range by a specific device or process. Instead of broadcasting access to multiple devices, unicast memory access means that a particular GPU device is granted explicit read/write permissions to a reserved virtual address range that maps to a physical memory allocation.

Multicast Memory Access: Multicast memory access in the context of the VMM API refers to the capability for a single physical memory allocation or region to be mapped simultaneously to multiple devices' virtual address spaces using a multicast mechanism. This allows data to be efficiently shared in a one-to-many fashion across multiple GPUs, reducing redundant data transfers and improving communication efficiency. NVIDIA's CUDA VMM API supports creating a multicast object that binds together physical memory allocations from multiple devices.

4.16.1.2 Query for Support

Applications should query for feature support before attempting to use them, as their availability can vary depending on the GPU architecture, driver version, and specific software libraries being used. The following sections detail how to programmatically check for the necessary support.

VMM Support Before attempting to use VMM APIs, applications must ensure that the devices they want to use support CUDA virtual memory management. The following code sample shows querying for VMM support:

```
int deviceSupportsVmm;
CUresult result = cuDeviceGetAttribute(&deviceSupportsVmm, CU_DEVICE_
↪ATTRIBUTE_VIRTUAL_MEMORY_MANAGEMENT_SUPPORTED, device);
if (deviceSupportsVmm != 0) {
    // `device` supports Virtual Memory Management
}
```

Fabric Memory Support: Before attempting to use fabric memory, applications must ensure that the devices they want to use support fabric memory. The following code sample shows querying for fabric memory support:

```
int deviceSupportsFabricMem;
CUresult result = cuDeviceGetAttribute(&deviceSupportsFabricMem, CU_DEVICE_
↪ATTRIBUTE_HANDLE_TYPE_FABRIC_SUPPORTED, device);
if (deviceSupportsFabricMem != 0) {
    // `device` supports Fabric Memory
}
```

Aside from using `CU_MEM_HANDLE_TYPE_FABRIC` as handle type and not requiring OS native mechanisms for inter-process communication to exchange sharable handles, there is no difference in using fabric memory compared to other allocation handle types.

IMEX Channels Support Within an IMEX domain, IMEX channels enable secure memory sharing in multi-user environments. The NVIDIA driver implements this by creating a character device, `nvidia-caps-imex-channels`. To use fabric handle-based sharing, users should verify two things:

- First, applications must verify that this device exists under */proc/devices*:

```
# cat /proc/devices | grep nvidia
195 nvidia
195 nvidiactl
234 nvidia-caps-imex-channels
509 nvidia-nvswitch

The nvidia-caps-imex-channels device should have a major number (e.g., 234).
```

- Second, for two CUDA processes (an exporter and an importer) to share memory, they must both have access to the same IMEX channel file. These files, such as */dev/nvidia-caps-imex-channels/channel0*, are nodes that represent individual IMEX channels. System administrators must create these files, for example, using the *mknod()* command.

```
# mknod /dev/nvidia-caps-imex-channels/channelN c <major_number> 0

This command creates channelN using the major number obtained from
/proc/devices.
```

Note

By default, the driver can create channel0 if the *NVreg_CreateImexChannel0* module parameter is specified.

Multicast Object Support: Before attempting to use multicast objects, applications must ensure that the devices they want to use support them. The following code sample shows querying for multicast object support:

```
int deviceSupportsMultiCast;
CUresult result = cuDeviceGetAttribute(&deviceSupportsMultiCast, CU_DEVICE_
→ATTRIBUTE_MULTICAST_SUPPORTED, device);
if (deviceSupportsMultiCast != 0) {
    // `device` supports Multicast Objects
}
```

4.16.2. API Overview

The VMM API provides developers with granular control over virtual memory management. VMM, being a very low-level API, requires use of the *CUDA Driver API* directly. This versatile API can be used in both single-node and multi-node environments.

To use VMM effectively, developers must have a solid grasp of a few key concepts in memory management: - Knowledge of the operating system's virtual memory fundamentals, including how it handles pages and address spaces - An understanding of memory hierarchy and hardware characteristics is necessary - Familiarity with inter-process communication (IPC) methods, such as sockets or message passing, - A basic knowledge of security for memory access rights

The VMM API workflow involves a sequence of steps for memory management, with a key focus on sharing memory between different devices or processes. Initially, a developer must allocate physical memory on the source device. To facilitate sharing, the VMM API utilizes handles to convey necessary information to the target device or process. The user must export a handle for sharing, which can be either an OS-specific handle or a fabric-specific handle. OS-specific handles are limited to inter-process communication on a single node, while fabric-specific handles offer greater versatility and can be used in both single-node and multi-node environments. It's important to note that using fabric-specific handles requires the enablement of IMEX channels.

Once the handle is exported, it must be shared with the receiving process or processes using an inter-process communication protocol, with the choice of method left to the developer. The receiving process then uses the VMM API to import the handle. After the handle has been successfully exported, shared, and imported, both the source and target processes must reserve virtual address space where the allocated physical memory will be mapped. The final step is to set the memory access rights for each device, ensuring proper permissions are established. This entire process, including both handle approaches, is further detailed in the accompanying figure.

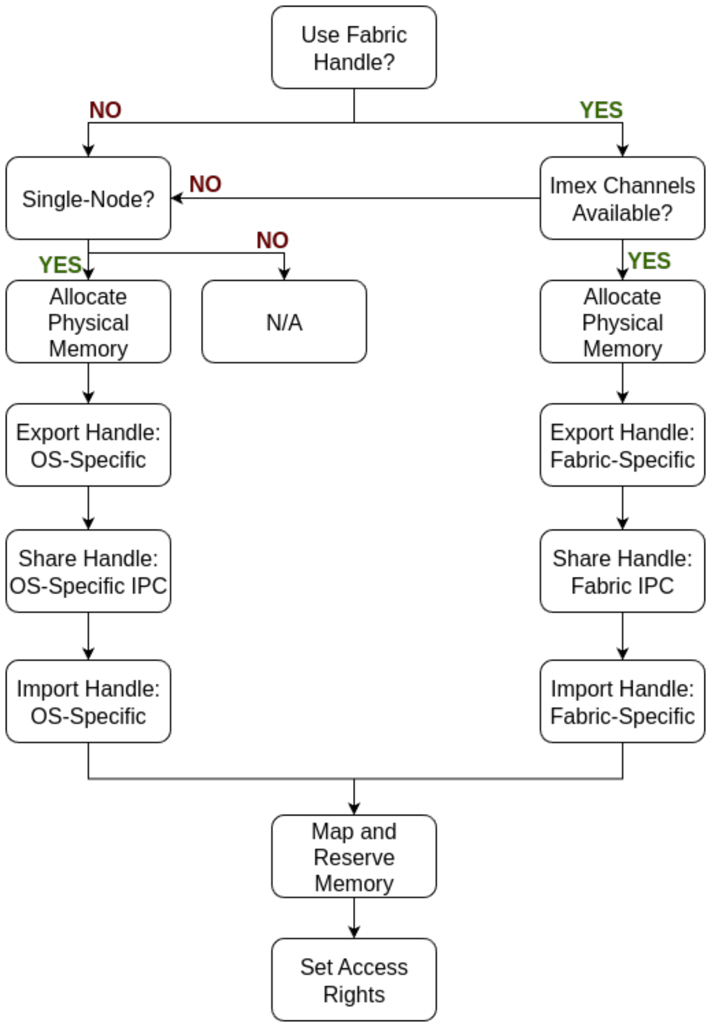


Figure 52: VMM Usage Overview. This diagram outlines the series of steps required for VMM utilization. The process begins by evaluating the environmental setup. Based on this assessment, the user must make a critical initial decision: whether to utilize fabric memory handles or OS-specific handles. A distinct series of subsequent steps must be taken based on the initial handle choice. However, the final memory management operations—specifically mapping, reserving, and setting access rights of the allocated memory—are identical to the type of handle that was selected.

4.16.3. Unicast Memory Sharing

Sharing GPU memory can happen on one machine with multiple GPUs or across a network of machines. The process follows these steps:

- Allocate and Export: A CUDA program on one GPU allocates memory and gets a sharable handle for it.
- Share and Import: The handle is then sent to other programs on the node using IPC, MPI, or NCCL etc. In the receiving GPUs, the CUDA driver imports the handle, creates the necessary memory objects
- Reserve and Map: The driver creates a mapping from the program's Virtual Address (VA) to the GPU's Physical Address (PA) to its network Fabric Address (FA).
- Access Rights: Setting access rights for the allocation.
- Releasing the Memory: Freeing all allocations when program ends its execution.

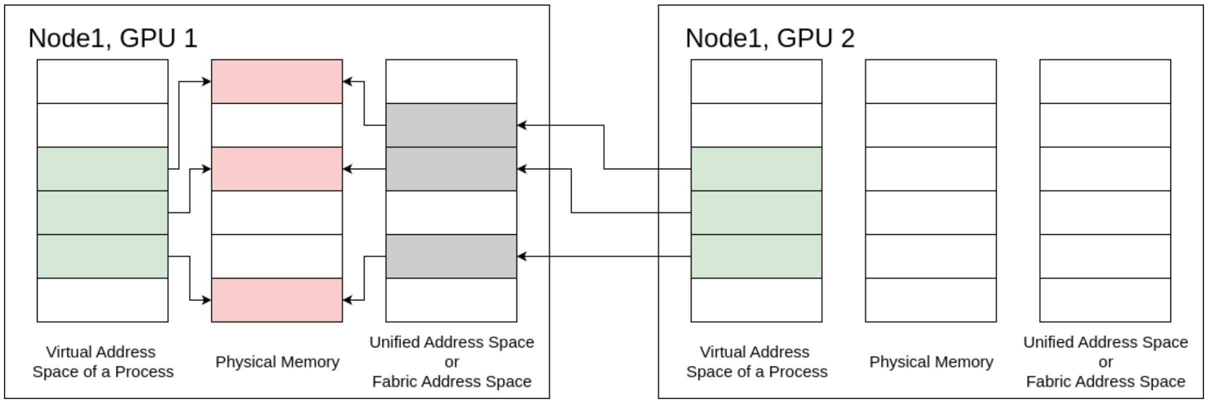

Node1, GPU 1
Node1, GPU 2
Virtual Address Space of a Process
Physical Memory
Unified Address Space or Fabric Address Space
Virtual Address Space of a Process
Physical Memory
Unified Address Space or Fabric Address Space

Figure 53: Unicast Memory Sharing Example

4.16.3.1 Allocate and Export

Allocating Physical Memory The first step in memory allocation using virtual memory management APIs is to create a physical memory chunk that will provide a backing for the allocation. In order to allocate physical memory, applications must use the `cuMemCreate` API. The allocation created by this function does not have any device or host mappings. The function argument `CUmemGenericAllocationHandle` describes the properties of the memory to allocate such as the location of the allocation, if the allocation is going to be shared to another process (or graphics APIs), or the physical attributes of the memory to be allocated. Users must ensure the requested allocation's size is aligned to appropriate granularity. Information regarding an allocation's granularity requirements can be queried using `cuMemGetAllocationGranularity`.

OS-Specific Handle (Linux)

```
CUmemGenericAllocationHandle allocatePhysicalMemory(int device, size_t size) {
    CUmemAllocationHandleType handleType = CU_MEM_HANDLE_TYPE_POSIX_FILE_
↪DESCRIPTOR;
    CUmemAllocationProp prop = {};
    prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
    prop.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
```

(continues on next page)

(continued from previous page)

```
    prop.location.id = device;
    prop.requestedHandleType = handleType;

    size_t granularity = 0;
    cuMemGetAllocationGranularity(&granularity, &prop, CU_MEM_ALLOC_
↪GRANULARITY_MINIMUM);

    // Ensure size matches granularity requirements for the allocation
    size_t padded_size = ROUND_UP(size, granularity);

    // Allocate physical memory
    CUmemGenericAllocationHandle allocHandle;
    cuMemCreate(&allocHandle, padded_size, &prop, 0);

    return allocHandle;
}
```

Fabric Handle

```
CUmemGenericAllocationHandle allocatePhysicalMemory(int device, size_t size) {
    CUmemAllocationHandleType handleType = CU_MEM_HANDLE_TYPE_FABRIC;
    CUmemAllocationProp prop = {};
    prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
    prop.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
    prop.location.id = device;
    prop.requestedHandleType = handleType;

    size_t granularity = 0;
    cuMemGetAllocationGranularity(&granularity, &prop, CU_MEM_ALLOC_
↪GRANULARITY_MINIMUM);

    // Ensure size matches granularity requirements for the allocation
    size_t padded_size = ROUND_UP(size, granularity);

    // Allocate physical memory
    CUmemGenericAllocationHandle allocHandle;
    cuMemCreate(&allocHandle, padded_size, &prop, 0);

    return allocHandle;
}
```

Note

The memory allocated by `cuMemCreate` is referenced by the `CUmemGenericAllocationHandle` it returns. Note that this reference is not a pointer and its memory is not accessible yet.

Note

Properties of the allocation handle can be queried using `cuMemGetAllocationPropertiesFromHandle`.

Exporting Memory Handle The CUDA virtual memory management API expose a new mechanism for interprocess communication using handles to exchange necessary information about the allocation and physical address space. One can export handles for OS-specific IPC or fabric-specific IPC. OS-specific IPC handles can only be used on a single-node setup. Fabric-specific handles can be used on a single or multi node setups.

OS-Specific Handle (Linux)

```
CUmemAllocationHandleType handleType = CU_MEM_HANDLE_TYPE_POSIX_FILE_
↪DESCRIPTOR;
CUmemGenericAllocationHandle handle = allocatePhysicalMemory(0, 1<<21);
int fd;
cuMemExportToShareableHandle(&fd, handle, handleType, 0);
```

Fabric Handle

```
CUmemAllocationHandleType handleType = CU_MEM_HANDLE_TYPE_FABRIC;
CUmemGenericAllocationHandle handle = allocatePhysicalMemory(0, 1<<21);
CUmemFabricHandle fh;
cuMemExportToShareableHandle(&fh, handle, handleType, 0);
```

Note

OS-specific handles require all processes to be part of the same OS.

Note

Fabric-specific handles require IMEX channels to be enabled by sysadmin.

The memMapIpcDrv sample can be used as an example for using IPC with VMM allocations.

4.16.3.2 Share and Import

Sharing Memory Handle Once the handle is exported, it must be shared with the receiving process or processes using an inter-process communication protocol. The developer is free to use any method for sharing the handle. The specific IPC method used depends on the application's design and environment. Common methods include OS-specific inter-process sockets and distributed message passing. Using OS-specific IPC offers high-performance transfer, but is limited to processes on the same machine and not portable. Fabric-specific IPC is simpler and more portable. However, fabric-specific IPC requires system-level support. The chosen method must securely and reliably transfer the handle data to the target process so it can be used to import the memory and establish a valid mapping. The flexibility in choosing the IPC method allows the VMM API to be integrated into a wide range of system architectures, from single-node applications to distributed, multi-node setups. In the following code snippets, we'll provide examples for sharing and receiving handles using both socket programming and MPI.

Send: OS-Specific IPC (Linux)

```
int ipcSendShareableHandle(int socket, int fd, pid_t process) {
    struct msghdr msg;
    struct iovec iov[1];

    union {
        struct cmsghdr cm;
        char* control;
    } control_un;

    size_t sizeof_control = CMSG_SPACE(sizeof(int)) * sizeof(char);
    control_un.control = (char*) malloc(sizeof_control);

    struct cmsghdr *cmptr;
    ssize_t readResult;
    struct sockaddr_un cliaddr;
    socklen_t len = sizeof(cliaddr);

    // Construct client address to send this SHareable handle to
    memset(&cliaddr, 0, sizeof(cliaddr));
    cliaddr.sun_family = AF_UNIX;
    char temp[20];
    sprintf(temp, "%s%u", "/tmp/", process);
    strcpy(cliaddr.sun_path, temp);
    len = sizeof(cliaddr);

    // Send corresponding shareable handle to the client
    int sendfd = fd;

    msg.msg_control = control_un.control;
    msg.msg_controllen = sizeof_control;

    cmptr = CMSG_FIRSTHDR(&msg);
    cmptr->cmsg_len = CMSG_LEN(sizeof(int));
    cmptr->cmsg_level = SOL_SOCKET;
    cmptr->cmsg_type = SCM_RIGHTS;

    memmove(CMSG_DATA(cmptr), &sendfd, sizeof(sendfd));

    msg.msg_name = (void *)&cliaddr;
    msg.msg_namelen = sizeof(struct sockaddr_un);

    iov[0].iov_base = (void *)"";
    iov[0].iov_len = 1;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

    ssize_t sendResult = sendmsg(socket, &msg, 0);
    if (sendResult <= 0) {
        perror("IPC failure: Sending data over socket failed");
        free(control_un.control);
        return -1;
```

(continues on next page)

(continued from previous page)

```
    }

    free(control_un.control);
    return 0;
}
```

Send: OS-Specific IPC (WIN)

```
int ipcSendShareableHandle(HANDLE *handle, HANDLE &shareableHandle, PROCESS_
↪INFORMATION process) {
    HANDLE hProcess = OpenProcess(PROCESS_DUP_HANDLE, FALSE, process.
↪dwProcessId);
    HANDLE hDup = INVALID_HANDLE_VALUE;
    DuplicateHandle(GetCurrentProcess(), shareableHandle, hProcess, &hDup, 0,
↪FALSE, DUPLICATE_SAME_ACCESS);
    DWORD cbWritten;
    WriteFile(handle->hMailslot[i], &hDup, (DWORD)sizeof(hDup), &cbWritten,
↪(LPOVERLAPPED)NULL);
    CloseHandle(hProcess);
    return 0;
}
```

Send: Fabric IPC

```
MPI_Send(&fh, sizeof(CUmemFabricHandle), MPI_BYTE, 1, 0, MPI_COMM_WORLD);
```

Receive: OS-Specific IPC (Linux)

```
int ipcRecvShareableHandle(int socket, int* fd) {
    struct msghdr msg = {0};
    struct iovec iov[1];
    struct cmsghdr cm;

    // Union to guarantee alignment requirements for control array
    union {
        struct cmsghdr cm;
        // This will not work on QNX as QNX CMSG_SPACE calls __cmsg_alignbytes
        // And __cmsg_alignbytes is a runtime function instead of compile-time
↪macros
        // char control[CMSG_SPACE(sizeof(int))]
        char* control;
    } control_un;

    size_t sizeof_control = CMSG_SPACE(sizeof(int)) * sizeof(char);
    control_un.control = (char*) malloc(sizeof_control);
    struct cmsghdr *cmptr;
    ssize_t n;
    int receivedfd;
    char dummy_buffer[1];
    ssize_t sendResult;
```

(continues on next page)

(continued from previous page)

```
    msg.msg_control = control_un.control;
    msg.msg_controllen = sizeof_control;

    iov[0].iov_base = (void *)dummy_buffer;
    iov[0].iov_len = sizeof(dummy_buffer);

    msg.msg_iov = iov;
    msg.msg_iovlen = 1;
    if ((n = recvmsg(socket, &msg, 0)) <= 0) {
        perror("IPC failure: Receiving data over socket failed");
        free(control_un.control);
        return -1;
    }

    if (((cmptr = CMSG_FIRSTHDR(&msg)) != NULL) &&
        (cmptr->cmsg_len == CMSG_LEN(sizeof(int)))) {
        if ((cmptr->cmsg_level != SOL_SOCKET) || (cmptr->cmsg_type != SCM_
↪RIGHTS)) {
        free(control_un.control);
        return -1;
        }

        memmove(&receivedfd, CMSG_DATA(cmptr), sizeof(receivedfd));
        *fd = receivedfd;
    } else {
        free(control_un.control);
        return -1;
    }

    free(control_un.control);
    return 0;
}
```

Receive: OS-Specific IPC (WIN)

```
int ipcRecvShareableHandle(HANDLE &handle, HANDLE *shareableHandle) {
    DWORD cbRead;
    ReadFile(handle, shareableHandle, (DWORD)sizeof(*shareableHandles), &
↪cbRead, NULL);
    return 0;
}
```

Receive: Fabric IPC

```
MPI_Recv(&fh, sizeof(CUmemFabricHandle), MPI_BYTE, 1, 0, MPI_COMM_WORLD);
```

Importing Memory Handle Again, the user can import handles for OS-specific IPC or fabric-specific IPC. OS-specific IPC handles can only be used on a single-node. Fabric-specific handles can be used for single or multi node.

OS-Specific Handle (Linux)

```
CUmemAllocationHandleType handleType = CU_MEM_HANDLE_TYPE_POSIX_FILE_
↪DESCRIPTOR;
cuMemImportFromShareableHandle(handle, (void*) &fd, handleType);
```

Fabric Handle

```
CUmemAllocationHandleType handleType = CU_MEM_HANDLE_TYPE_FABRIC;
cuMemImportFromShareableHandle(handle, (void*) &fh, handleType);
```

4.16.3.3 Reserve and Map

Reserving a Virtual Address Range

Since notions of address and memory are distinct in VMM, applications must carve out an address range that can hold the memory allocations made by `cuMemCreate`. The address range reserved must be at least as large as the sum of the sizes of all the physical memory allocations the user plans to place in them.

Applications can reserve a virtual address range by passing appropriate parameters to `cuMemAddressReserve`. The address range obtained will not have any device or host physical memory associated with it. The reserved virtual address range can be mapped to memory chunks belonging to any device in the system, thus providing the application a continuous VA range backed and mapped by memory belonging to different devices. Applications are expected to return the virtual address range back to CUDA using `cuMemAddressFree`. Users must ensure that the entire VA range is unmapped before calling `cuMemAddressFree`. These functions are conceptually similar to `mmap` and `munmap` on Linux or `VirtualAlloc` AND `VirtualFree` on Windows. The following code snippet illustrates the usage for the function:

```
CUdeviceptr ptr;
// `ptr` holds the returned start of virtual address range reserved.
CUresult result = cuMemAddressReserve(&ptr, size, 0, 0, 0); // alignment = 0
↪for default alignment
```

Mapping Memory

The allocated physical memory and the carved out virtual address space from the previous two sections represent the memory and address distinction introduced by the VMM APIs. For the allocated memory to be useable, the user must map the memory to the address space. The address range obtained from `cuMemAddressReserve` and the physical allocation obtained from `cuMemCreate` or `cuMemImportFromShareableHandle` must be associated with each other by using `cuMemMap`.

Users can associate allocations from multiple devices to reside in contiguous virtual address ranges as long as they have carved out enough address space. To decouple the physical allocation and the address range, users must unmap the address of the mapping with `cuMemUnmap`. Users can map and unmap memory to the same address range as many times as they want, so long as they ensure that they don't attempt to create mappings on VA range reservations that are already mapped. The following code snippet illustrates the usage for the function:

```
CUdeviceptr ptr;
// `ptr`: address in the address range previously reserved by
↪cuMemAddressReserve.
// `allocHandle`: CUmemGenericAllocationHandle obtained by a previous call to
```

(continues on next page)

(continued from previous page)

```
↪cuMemCreate.
CUresult result = cuMemMap(ptr, size, 0, allocHandle, 0);
```

4.16.3.4 Access Rights

CUDA's virtual memory management APIs enable applications to explicitly protect their VA ranges with access control mechanisms. Mapping the allocation to a region of the address range using cuMemMap does not make the address accessible, and would result in a program crash if accessed by a CUDA kernel. Users must specifically select access control using the cuMemSetAccess function on source and accessing devices. This allows or restricts access for specific devices to a mapped address range. The following code snippet illustrates the usage for the function:

```
void setAccessOnDevice(int device, CUdeviceptr ptr, size_t size) {
    CUmemAccessDesc accessDesc = {};
    accessDesc.location.type = CU_MEM_LOCATION_TYPE_DEVICE;
    accessDesc.location.id = device;
    accessDesc.flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;

    // Make the address accessible
    cuMemSetAccess(ptr, size, &accessDesc, 1);
}
```

The access control mechanism exposed with VMM allows users to be explicit about which allocations they want to share with other peer devices on the system. As specified earlier, cudaEnablePeerAccess forces all prior and future allocations made with cudaMalloc to be mapped to the target peer device. This can be convenient in many cases as user doesn't have to worry about tracking the mapping state of every allocation to every device in the system. But this approach has performance implications. With access control at allocation granularity, VMM allows peer mappings with minimal overhead.

The vectorAddMMAP sample can be used as an example for using the Virtual Memory Management APIs.

4.16.3.5 Releasing the Memory

To release the allocated memory and address space, both the source and target processes should use *cuMemUnmap*, *cuMemRelease*, and *cuMemAddressFree* functions in that order. The *cuMemUnmap* function un-maps a previously mapped memory region from an address range, effectively detaching the physical memory from the reserved virtual address space. Next, *cuMemRelease* deallocates the physical memory that was previously created, returning it to the system. Finally, *cuMemAddressFree* frees a virtual address range that was previously reserved, making it available for future use. This specific order ensures a clean and complete deallocation of both the physical memory and the virtual address space.

```
cuMemUnmap(ptr, size);
cuMemRelease(handle);
cuMemAddressFree(ptr, size);
```

Note

In the OS-specific case, the exported handle must be closed using *fclose*. This step is not applicable to the fabric-based case.

4.16.4. Multicast Memory Sharing

The Multicast Object Management APIs provide a way for the application to create multicast objects and, in combination with the Virtual Memory Management APIs described above, allow applications to leverage NVLink SHARP on supported NVLink connected GPUs connected with NVSwitch. NVLink SHARP allows CUDA applications to leverage in-fabric computing to accelerate operations like broadcast and reductions between GPUs connected with NVSwitch. For this to work, multiple NVLink connected GPUs form a multicast team and each GPU from the team backs up a multicast object with physical memory. So a multicast team of N GPUs has N physical replicas of a multicast object, each local to one participating GPU. The multimem PTX instructions using mappings of multicast objects work with all replicas of the multicast object.

To work with multicast objects, an application needs to

- Query multicast support
- Create a multicast handle with `cuMulticastCreate`.
- Share the multicast handle with all processes that control a GPU which should participate in a multicast team. This works with `cuMemExportToShareableHandle` as described above.
- Add all GPUs that should participate in the multicast team with `cuMulticastAddDevice`.
- For each participating GPU, bind physical memory allocated with `cuMemCreate` as described above to the multicast handle. All devices need to be added to the multicast team before binding memory on any device.
- Reserve an address range, map the multicast handle and set access rights as described above for regular unicast mappings. Unicast and multicast mappings to the same physical memory are possible. See the *Virtual Aliasing Support* section above on how to ensure consistency between multiple mappings to the same physical memory.
- Use the multimem PTX instructions with the multicast mappings.

The `multi_node_p2p` example in the Multi GPU Programming Models GitHub repository contains a complete example using fabric memory including multicast objects to leverage NVLink SHARP. Please note that this example is for developers of libraries like NCCL or NVSHMEM. It shows how higher-level programming models like NVSHMEM work internally within a (multi-node) NVLink domain. Application developers generally should use the higher-level MPI, NCCL, or NVSHMEM interfaces instead of this API.

4.16.4.1 Allocating Multicast Objects

Multicast objects can be created with `cuMulticastCreate`:

```
CUmemGenericAllocationHandle createMCHandle(int numDevices, size_t size) {
    CUmemAllocationProp mcProp = {};
    mcProp.numDevices = numDevices;
    mcProp.handleTypes = CU_MEM_HANDLE_TYPE_FABRIC; // or on single node CU_
↪MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR

    size_t granularity = 0;
    cuMulticastGetGranularity(&granularity, &mcProp, CU_MEM_ALLOC_GRANULARITY_
↪MINIMUM);

    // Ensure size matches granularity requirements for the allocation
    size_t padded_size = ROUND_UP(size, granularity);
```

(continues on next page)

(continued from previous page)

```
    mcProp.size = padded_size;

    // Create Multicast Object this has no devices and no physical memory
↪associated yet
    CUmemGenericAllocationHandle mcHandle;
    cuMulticastCreate(&mcHandle, &mcProp);

    return mcHandle;
}
```

4.16.4.2 Add Devices to Multicast Objects

Devices can be added to a multicast team with `cuMulticastAddDevice`:

```
cuMulticastAddDevice(&mcHandle, device);
```

This step needs to be completed on all processes controlling devices that participate in a multicast team before memory on any device is bound to the multicast object.

4.16.4.3 Bind Memory to Multicast Objects

After a multicast object has been created and all participating devices have been added to the multicast object it needs to be backed with physical memory allocated with `cuMemCreate` for each device:

```
cuMulticastBindMem(mcHandle, mcOffset, memHandle, memOffset, size, 0 /*flags*/
↪);
```

4.16.4.4 Use Multicast Mappings

To use multicast mappings in CUDA C++, it is necessary to use the multimem PTX instructions with inline PTX:

```
__global__ void all_reduce_norm_barrier_kernel(float* l2_norm,
                                               float* partial_l2_norm_mc,
                                               unsigned int* arrival_counter_
↪uc, unsigned int* arrival_counter_mc,
                                               const unsigned int expected_
↪count) {
    assert( 1 == blockDim.x * blockDim.y * blockDim.z * gridDim.x * gridDim.y
↪* gridDim.z );
    float l2_norm_sum = 0.0;
#if __CUDA_ARCH__ >= 900

    // atomic reduction to all replicas
    // this can be conceptually thought of as __threadfence_system();
↪atomicAdd_system(arrival_counter_mc, 1);
    cuda::ptx::multimem_red(cuda::ptx::release_t, cuda::ptx::scope_sys_t,
↪cuda::ptx::op_add_t, arrival_counter_mc, n);

    // Need a fence between Multicast (mc) and Unicast (uc) access to the same
↪memory `arrival_counter_uc` and `arrival_counter_mc`:
```

(continues on next page)

(continued from previous page)

```
    // - fence.proxy instructions establish an ordering between memory
↪accesses that may happen through different proxies
    // - Value .alias of the .proxykind qualifier refers to memory accesses
↪performed using virtually aliased addresses to the same memory location.
    // from https://docs.nvidia.com/cuda/parallel-thread-execution/#parallel-
↪synchronization-and-communication-instructions-membar
    cuda::ptx::fence_proxy_alias();

    // spin wait with acquire ordering on UC mapping till all peers have
↪arrived in this iteration
    // Note: all ranks need to reach another barrier after this kernel, such
↪that it is not possible for the barrier to be unblocked by an
    // arrival of a rank for the next iteration if some other rank is slow.
    cuda::atomic_ref<unsigned int,cuda::thread_scope_system> ac(arrival_
↪counter_uc);
    while (expected_count > ac.load(cuda::memory_order_acquire));

    // Atomic load reduction from all replicas. It does not provide ordering
↪so it can be relaxed.
    asm volatile ("multimem.ld_reduce.relaxed.sys.global.add.f32 %0, [%1];" :
↪"=f"(l2_norm_sum) : "l"(partial_l2_norm_mc) : "memory");

#else
    #error "ERROR: multimem instructions require compute capability 9.0 or
↪larger."
#endif

    *l2_norm = std::sqrt(l2_norm_sum);
}
```

4.16.5. Advanced Configuration

4.16.5.1 Memory Type

VMM also provides a mechanism for applications to allocate special types of memory that certain devices may support. With `cuMemCreate`, applications can specify memory type requirements using the `CUmemAllocationProp::allocFlags` to opt-in to specific memory features. Applications must ensure that the requested memory type is supported by the device.

4.16.5.2 Compressible Memory

Compressible memory can be used to accelerate accesses to data with unstructured sparsity and other compressible data patterns. Compression can save DRAM bandwidth, L2 read bandwidth, and L2 capacity depending on the data. Applications that want to allocate compressible memory on devices that support compute data compression can do so by setting `CUmemAllocationProp::allocFlags::compressionType` to `CU_MEM_ALLOCATION_COMP_GENERIC`. Users must query if device supports Compute Data Compression by using `CU_DEVICE_ATTRIBUTE_GENERIC_COMPRESSION_SUPPORTED`. The following code snippet illustrates querying compressible memory support `cuDeviceGetAttribute`.

```
int compressionSupported = 0;
cuDeviceGetAttribute(&compressionSupported, CU_DEVICE_ATTRIBUTE_GENERIC_
↪COMPRESSION_SUPPORTED, device);
```

On devices that support compute data compression, users must opt in at allocation time as shown below:

```
prop.allocFlags.compressionType = CU_MEM_ALLOCATION_COMP_GENERIC;
```

For a variety of reasons such as limited hardware resources, the allocation may not have compression attributes. To verify that the flags worked, the user query the properties of the allocated memory using `cuMemGetAllocationPropertiesFromHandle`.

```
CUmemAllocationProp allocationProp = {};
cuMemGetAllocationPropertiesFromHandle(&allocationProp, allocationHandle);

if (allocationProp.allocFlags.compressionType == CU_MEM_ALLOCATION_COMP_
↪GENERIC)
{
    // Obtained compressible memory allocation
}
```

4.16.5.3 Virtual Aliasing Support

The virtual memory management APIs provide a way to create multiple virtual memory mappings or "proxies" to the same allocation using multiple calls to `cuMemMap` with different virtual addresses. This is called virtual aliasing. Unless otherwise noted in the PTX ISA, writes to one proxy of the allocation are considered inconsistent and incoherent with any other proxy of the same memory until the writing device operation (grid launch, memcpy, memset, and so on) completes. Grids present on the GPU prior to a writing device operation but reading after the writing device operation completes are also considered to have inconsistent and incoherent proxies.

For example, the following snippet is considered undefined, assuming device pointers A and B are virtual aliases of the same memory allocation:

```
__global__ void foo(char *A, char *B) {
  *A = 0x1;
  printf("%d\n", *B);    // Undefined behavior!  *B can take on either
// the previous value or some value in-between.
}
```

The following is defined behavior, assuming these two kernels are ordered monotonically (by streams or events).

```
__global__ void foo1(char *A) {
  *A = 0x1;
}

__global__ void foo2(char *B) {
  printf("%d\n", *B);    // *B == *A == 0x1 assuming foo2 waits for foo1
// to complete before launching
}

cudaMemcpyAsync(B, input, size, stream1);    // Aliases are allowed at
```

(continues on next page)

(continued from previous page)

```
// operation boundaries
foo1<<<1,1,0,stream1>>>(A);                   // allowing foo1 to access A.
cudaEventRecord(event, stream1);
cudaStreamWaitEvent(stream2, event);
foo2<<<1,1,0,stream2>>>(B);
cudaStreamWaitEvent(stream3, event);
cudaMemcpyAsync(output, B, size, stream3);  // Both launches of foo2 and
                                            // cudaMemcpy (which both
                                            // read) wait for foo1 (which
↪writes)
                                            // to complete before proceeding
```

If accessing same allocation through different "proxies" is required in the same kernel, a `fence.proxy.alias` can be used between the two accesses. The above example can thus be made legal with inline PTX assembly:

```
__global__ void foo(char *A, char *B) {
  *A = 0x1;
  cuda::ptx::fence_proxy_alias();
  printf("%d\n", *B);    // *B == *A == 0x1
}
```

4.16.5.4 OS-Specific Handle Details for IPC

With `cuMemCreate`, users have can indicate at allocation time that they have earmarked a particular allocation for inter-process communication or graphics interop purposes. Applications can do this by setting `CUmemAllocationProp::requestedHandleTypes` to a platform-specific field. On Windows, when `CUmemAllocationProp::requestedHandleTypes` is set to `CU_MEM_HANDLE_TYPE_WIN32` applications must also specify an LPSECURITYATTRIBUTES attribute in `CUmemAllocationProp::win32HandleMetaData`. This security attribute defines the scope of which exported allocations may be transferred to other processes.

Users must ensure they query for support of the requested handle type before attempting to export memory allocated with `cuMemCreate`. The following code snippet illustrates query for handle type support in a platform-specific way.

```
int deviceSupportsIpcHandle;
#if defined(__linux__)
    cuDeviceGetAttribute(&deviceSupportsIpcHandle, CU_DEVICE_ATTRIBUTE_HANDLE_
↪TYPE_POSIX_FILE_DESCRIPTOR_SUPPORTED, device));
#else
    cuDeviceGetAttribute(&deviceSupportsIpcHandle, CU_DEVICE_ATTRIBUTE_HANDLE_
↪TYPE_WIN32_HANDLE_SUPPORTED, device));
#endif
```

Users should set the `CUmemAllocationProp::requestedHandleTypes` appropriately as shown below:

```
#if defined(__linux__)
    prop.requestedHandleTypes = CU_MEM_HANDLE_TYPE_POSIX_FILE_DESCRIPTOR;
#else
    prop.requestedHandleTypes = CU_MEM_HANDLE_TYPE_WIN32;
    prop.win32HandleMetaData = // Windows specific LPSECURITYATTRIBUTES
```

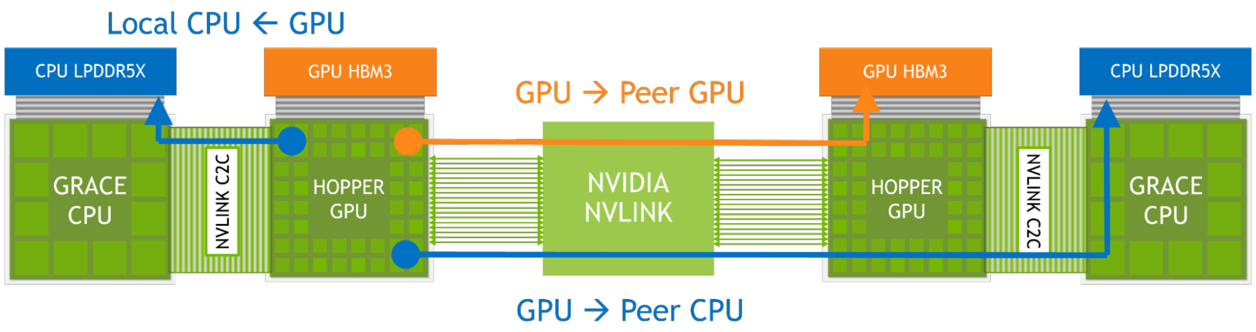
(continues on next page)

(continued from previous page)

```
↪attribute.
#endif
```

4.17. Extended GPU Memory

The Extended GPU Memory (EGM) feature, utilizing the high-bandwidth NVLink-C2C, facilitates efficient access to all system memory by GPUs, in both single-node and multi-node systems. EGM applies to integrated CPU-GPU NVIDIA systems by allowing physical memory allocation that can be accessed from any GPU thread within the setup. EGM ensures that all GPUs can access its resources at the speed of either GPU-GPU NVLink or NVLink-C2C.



In this setup, memory accesses occur via the local high-bandwidth NVLink-C2C. For remote memory accesses, GPU NVLink and, in some cases, NVLink-C2C are used. With EGM, GPU threads gain the capability to access all available memory resources, including CPU attached memory and HBM3, over the NVSwitch fabric.

4.17.1. Preliminaries

Before diving into API changes for EGM functionalities, we are going to cover currently supported topologies, identifier assignment, prerequisites for virtual memory management, and CUDA types for EGM.

4.17.1.1 EGM Platforms: System topology

Currently, EGM can be enabled in several platforms: **(1) Single-Node, Single-GPU**: Consists of an Arm-based CPU, CPU attached memory, and a GPU. Between the CPU and the GPU there is a high bandwidth C2C (Chip-to-Chip) interconnect. **(2) Single-Node, Multi-GPU**: Consists of ARM-based CPUs, each with attached memory, and multiple GPUs connected through an NVLink-based network. **(3) Multi-Node, Multi-GPU**: Two or more single-node systems, each as in (1) or (2) above, connected through an NVLink-based network.

Note

Using `cgroups` to limit available devices will block routing over EGM and cause performance issues. Use `CUDA_VISIBLE_DEVICES` instead.

4.17.1.2 Socket Identifiers: What are they? How to access them?

NUMA (Non-Uniform Memory Access) is a memory architecture used in multi-processor computer systems such that the memory is divided into multiple nodes. Each node has its own processors and memory. In such a system, NUMA divides the system into nodes and assigns a unique identifier (*numaID*) to every node.

EGM uses the NUMA node identifier which is assigned by the operating system. Note that, this identifier is different from the ordinal of a device and it is associated with the closest host node. In addition to the existing methods, the user can obtain the identifier of the host node (*numaID*) by calling cuDeviceGetAttribute with `CU_DEVICE_ATTRIBUTE_HOST_NUMA_ID` attribute type as follows:

```
int numaId;
cuDeviceGetAttribute(&numaId, CU_DEVICE_ATTRIBUTE_HOST_NUMA_ID,
↪deviceOrdinal);
```

4.17.1.3 Allocators and EGM support

Mapping system memory as EGM does not cause any performance issues. In fact, accessing a remote socket's system memory mapped as EGM is going to be faster. Because, with EGM traffic is guaranteed to be routed over NVLinks. Currently, `cuMemCreate` and `cudaMemPoolCreate` allocators are supported with appropriate location type and NUMA identifiers.

4.17.1.4 Memory management extensions to current APIs

Currently, EGM memory can be mapped with Virtual Memory (`cuMemCreate`) or Stream Ordered Memory (`cudaMemPoolCreate`) allocators. The user is responsible for allocating physical memory and mapping it to a virtual memory address space on all sockets.

Note

Multi-node, multi-GPU platforms require interprocess communication. Therefore we encourage the reader to see Chapter 4.15.

Note

We encourage readers to read CUDA Programming Guide's Chapter 4.16 and Chapter 4.3 for a better understanding.

New CUDA property types have been added to APIs for allowing those approaches to understand allocation locations using NUMA-like node identifiers:

CUDA Type	Used with
`CU_MEM_LOCATION_TYPE_HOST_NUMA`	`CUmemAllocationProp` for `cuMemCreate`
`cudaMemLocationTypeHostNuma`	`cudaMemPoolProps` for `cudaMemPoolCreate`

Note

Please see CUDA Driver API and CUDA Runtime Data Types to find more about NUMA specific CUDA types.

4.17.2. Using the EGM Interface

4.17.2.1 Single-Node, Single-GPU

Any of the existing CUDA host allocators as well as system allocated memory can be used to benefit from high-bandwidth C2C. To the user, local access is what a host allocation is today.

Note

Refer to the tuning guide for more information about memory allocators and page sizes.

4.17.2.2 Single-Node, Multi-GPU

In a multi-GPU system, the user has to provide host information for the placement. As we mentioned, a natural way to express that information would be by using NUMA node IDs and EGM follows this approach. Therefore, using the `cuDeviceGetAttribute` function the user should be able to learn the closest NUMA node id. (See *Socket Identifiers: What are they? How to access them?*). Then the user can allocate and manage EGM memory using VMM (Virtual Memory Management) API or CUDA Memory Pool.

4.17.2.2.1 Using VMM APIs

The first step in memory allocation using Virtual Memory Management APIs is to create a physical memory chunk that will provide a backing for the allocation. See CUDA Programming Guide's *Virtual Memory Management section* for more details. In EGM allocations the user has to explicitly provide `CU_MEM_LOCATION_TYPE_HOST_NUMA` as the location type and *numaID* as the location identifier. Also in EGM, allocations must be aligned to appropriate granularity of the platform. The following code snippet shows allocating physical memory with `cuMemCreate`:

```
CUmemAllocationProp prop{};
prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
prop.location.type = CU_MEM_LOCATION_TYPE_HOST_NUMA;
prop.location.id = numaId;
size_t granularity = 0;
cuMemGetAllocationGranularity(&granularity, &prop, MEM_ALLOC_GRANULARITY_
↪MINIMUM);
size_t padded_size = ROUND_UP(size, granularity);
CUmemGenericAllocationHandle allocHandle;
cuMemCreate(&allocHandle, padded_size, &prop, 0);
```

After physical memory allocation, we have to reserve an address space and map it to a pointer. These procedures do not have EGM-specific changes:

```
CUdeviceptr dptr;
cuMemAddressReserve(&dptr, padded_size, 0, 0, 0);
cuMemMap(dptr, padded_size, 0, allocHandle, 0);
```

Finally, the user has to explicitly protect mapped virtual address ranges. Otherwise access to the mapped space would result in a crash. Similar to the memory allocation, the user has to provide `CU_MEM_LOCATION_TYPE_HOST_NUMA` as the location type and *numaId* as the location identifier. Following code snippet create an access descriptors for the host node and the GPU to give read and write access for the mapped memory to both of them:

```
CUmemAccessDesc accessDesc[2]{{}};
accessDesc[0].location.type = CU_MEM_LOCATION_TYPE_HOST_NUMA;
accessDesc[0].location.id = numaId;
accessDesc[0].flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
accessDesc[1].location.type = CU_MEM_LOCATION_TYPE_DEVICE;
accessDesc[1].location.id = currentDev;
accessDesc[1].flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
cuMemSetAccess(dptr, size, accessDesc, 2);
```

4.17.2.2.2 Using CUDA Memory Pool

To define EGM, the user can create a memory pool on a node and give access to peers. In this case, the user has to explicitly define `cudaMemLocationTypeHostNuma` as the location type and *numaId* as the location identifier. The following code snippet shows creating a memory pool `cudaMemPoolCreate`:

```
cudaSetDevice(homeDevice);
cudaMemPoolProps props{};
props.allocType = cudaMemAllocationTypePinned;
props.location.type = cudaMemLocationTypeHostNuma;
props.location.id = numaId;
cudaMemPoolCreate(&memPool, &props);
```

Additionally, for direct connect peer access, it is also possible to use the existing peer access API, `cudaMemPoolSetAccess`. An example for an *accessingDevice* is shown in the following code snippet:

```
cudaMemAccessDesc desc{};
desc.flags = cudaMemAccessFlagsProtReadWrite;
desc.location.type = cudaMemLocationTypeDevice;
desc.location.id = accessingDevice;
cudaMemPoolSetAccess(memPool, &desc, 1);
```

When the memory pool is created, and accesses are given, the user can set created memory pool to the *residentDevice* and start allocating memory using `cudaMallocAsync`:

```
cudaDeviceSetMemPool(residentDevice, memPool);
cudaMallocAsync(&ptr, size, memPool, stream);
```

Note

EGM is mapped with 2MB pages. Therefore, users may encounter more TLB misses when accessing very large allocations.

4.17.2.3 Multi-Node, Multi-GPU

Beyond memory allocation, remote peer access does not have EGM-specific modification and it follows CUDA inter process (IPC) protocol. See CUDA Programming Guide for more details in IPC.

The user should allocate memory using `cuMemCreate` and again the user has to explicitly provide `CU_MEM_LOCATION_TYPE_HOST_NUMA` as the location type and *numaID* as the location identifier. In addition `CU_MEM_HANDLE_TYPE_FABRIC` should be defined as the requested handle type. The following code snippet shows allocating physical memory on Node A:

```
CUmemAllocationProp prop{};
prop.type = CU_MEM_ALLOCATION_TYPE_PINNED;
prop.requestedHandleTypes = CU_MEM_HANDLE_TYPE_FABRIC;
prop.location.type = CU_MEM_LOCATION_TYPE_HOST_NUMA;
prop.location.id = numaId;
size_t granularity = 0;
cuMemGetAllocationGranularity(&granularity, &prop,
                              MEM_ALLOC_GRANULARITY_MINIMUM);
size_t padded_size = ROUND_UP(size, granularity);
size_t page_size = ...;
assert(padded_size % page_size == 0);
CUmemGenericAllocationHandle allocHandle;
cuMemCreate(&allocHandle, padded_size, &prop, 0);
```

After creating allocation handle using `cuMemCreate` the user can export that handle to the other node, Node B, calling `cuMemExportToShareableHandle`:

```
cuMemExportToShareableHandle(&fabricHandle, allocHandle,
                             CU_MEM_HANDLE_TYPE_FABRIC, 0);
// At this point, fabricHandle should be sent to Node B via TCP/IP.
```

On Node B, the handle can be imported using `cuMemImportFromShareableHandle` and treated as any other fabric handle

```
// At this point, fabricHandle should be received from Node A via TCP/IP.
CUmemGenericAllocationHandle allocHandle;
cuMemImportFromShareableHandle(&allocHandle, &fabricHandle,
                               CU_MEM_HANDLE_TYPE_FABRIC);
```

When handle is imported at Node B, then the user can reserve an address space and map it locally in a regular fashion:

```
size_t granularity = 0;
cuMemGetAllocationGranularity(&granularity, &prop,
                              MEM_ALLOC_GRANULARITY_MINIMUM);
size_t padded_size = ROUND_UP(size, granularity);
size_t page_size = ...;
assert(padded_size % page_size == 0);
CUdeviceptr dptr;
cuMemAddressReserve(&dptr, padded_size, 0, 0, 0);
cuMemMap(dptr, padded_size, 0, allocHandle, 0);
```

As the final step, the user should give appropriate accesses to each of the local GPUs at Node B. An example code snippet that gives read and write access to eight local GPUs:

```
// Give all 8 local  GPUS access to exported EGM memory located on Node A.
↪                                                                      |
CUmemAccessDesc accessDesc[8];
for (int i = 0; i < 8; i++) {
   accessDesc[i].location.type = CU_MEM_LOCATION_TYPE_DEVICE;
   accessDesc[i].location.id = i;
   accessDesc[i].flags = CU_MEM_ACCESS_FLAGS_PROT_READWRITE;
}
cuMemSetAccess(dptr, size, accessDesc, 8);
```

4.18. CUDA Dynamic Parallelism

4.18.1. Introduction

4.18.1.1 Overview

CUDA dynamic parallelism - often abbreviated CDP - is a feature of the CUDA programming model that enables code running on the GPU to create new GPU work. That is, new work can be added to the GPU in the form of additional kernel launches from device code already running on the GPU. This feature can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can be made at runtime by threads executing on the device.

Data-dependent parallel work can be generated by a kernel at run-time. Before CDP was added to CUDA, some algorithms and programming patterns required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism. These program structures may be expressed more naturally using CUDA dynamic parallelism.

Note

This section documents the updated version of CUDA Dynamic Parallelism, sometimes called CDP2, which is the default in CUDA 12.0 and later. CDP2 is the only version of CUDA dynamic parallelism available on devices of CC 9.0 and higher. Developers can still opt in to the legacy CUDA Dynamic Parallelism, CDP1, for devices of CC lower than 9.0 with the compiler argument *-DCUDA_FORCE_CDP1_IF_SUPPORTED*. CDP1 documentation can be found in Legacy versions of the CUDA Programming Guide. CDP1 is slated for removal from a future version of CUDA.

4.18.2. Execution Environment

Dynamic parallelism in CUDA allows GPU threads to configure, launch, and implicitly synchronize new grids. A grid is an instance of a kernel launch, including the specific shape of the thread blocks and the grid of thread blocks. The distinction between a kernel function itself and the specific invocation of that kernel, i.e. a grid, is important to note in the following sections.

4.18.2.1 Parent and Child Grids

A device thread that configures and launches a new grid belongs to the parent grid. The new grid that is created by the invocation is called a child grid.

The invocation and completion of child grids is properly nested, meaning that the parent grid is not considered complete until all child grids created by its threads have completed, and the runtime guarantees an implicit synchronization between the parent and child.

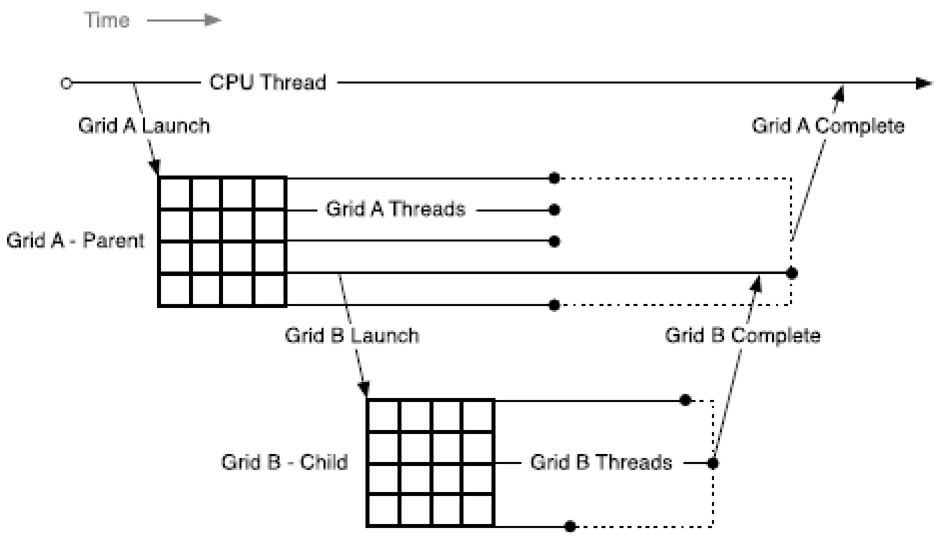


Figure 54: Parent-Child Launch Nesting

4.18.2.2 Scope of CUDA Primitives

CUDA Dynamic Parallelism relies on the *CUDA Device Runtime*, which enables calling a limited set of APIs which are syntactically similar to the CUDA Runtime API, but available in device code. The behavior of the device runtime APIs are similar to their host counterparts, but there are some differences. These differences are captured in the table in section *API Reference*.

On both host and device, the CUDA runtime offers an API for launching kernels and for tracking dependencies between launches via streams and events. On the device, launched kernels and CUDA objects are visible to all threads in the invoking grid. This means, for example, that a stream may be created by one thread and used by any other thread in the same grid. However, CUDA objects such as streams and events which were created on by device API calls are only valid within the grid where they were created.

4.18.2.3 Streams and Events

CUDA *Streams* and *Events* allow control over dependencies between kernel launches: kernels launched into the same stream execute in-order, and events may be used to create dependencies between streams. Streams and events created on the device serve this exact same purpose.

Streams and events created within a grid exist within grid scope, but have undefined behavior when used outside of the grid where they were created. As described above, all work launched by a grid is implicitly synchronized when the grid exits; work launched into streams is included in this, with all dependencies resolved appropriately. The behavior of operations on a stream that has been modified outside of grid scope is undefined.

Streams and events created on the host have undefined behavior when used within any kernel, just as streams and events created by a parent grid have undefined behavior if used within a child grid.

4.18.2.4 Ordering and Concurrency

The ordering of kernel launches from the device runtime follows CUDA Stream ordering semantics. Within a grid, all kernel launches into the same stream (with the exception of the *The Fire-and-Forget Stream*) are executed in-order. With multiple threads in the same grid launching into the same stream, the ordering within the stream is dependent on the thread scheduling within the grid, which may be controlled with synchronization primitives such as `__syncthreads()`.

Note that while named streams are shared by all threads within a grid, the implicit *NULL* stream is only shared by all threads within a thread block. If multiple threads in a thread block launch into the implicit stream, then these launches will be executed in-order. If threads in different thread blocks launch into the implicit stream, these launches may be executed concurrently. If concurrency is desired for launches from multiple threads within a thread block, explicit named streams should be used.

The device runtime introduces no new concurrency guarantees within the CUDA execution model. That is, there is no guarantee of concurrent execution between any number of different thread blocks on a device.

The lack of concurrency guarantee extends to a parent grid and their child grids. When a parent grid launches a child grid, the child may start to execute once stream dependencies are satisfied and hardware resources are available, but is not guaranteed to begin execution until the parent grid reaches an implicit synchronization point.

Concurrency may vary as a function of device configuration, application workload, and runtime scheduling. It is therefore unsafe to depend on any concurrency between different thread blocks.

4.18.3. Memory Coherence and Consistency

Parent and child grids share the same global and constant memory storage, but have distinct local and shared memory. The following table shows which memory spaces allow parent and child to access via the same pointers. Child grids can never access the local or shared memory of parent grids, nor can parent grids access local or shared memory of child grids.

Table 26: Dynamic Parallelism: Memory Scope Accessibility Between Parent and Child Grids

Memory Space	Parent/Child use same pointers?
Global Memory	Yes
Mapped Memory	Yes
Local Memory	No
Shared Memory	No
Texture Memory	Yes (read-only)

4.18.3.1 Global Memory

Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There is only one point of time in the execution of a child grid when its view of memory is fully consistent with the parent thread: at the point when the child grid is invoked by the parent.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. With the removal of `cudaDeviceSynchronize()`, it is no longer possible to access the modifications made by the threads in the child grid from the parent grid. The only way to access the modifications made by the threads in the child grid before the parent grid exits is via a kernel launched into the `cudaStreamTailLaunch` stream.

In the following example, the child grid executing `child_launch` is only guaranteed to see the modifications to `data` made before the child grid was launched. Since thread 0 of the parent is performing the launch, the child will be consistent with the memory seen by thread 0 of the parent. Due to the first `__syncthreads()` call, the child will see `data[0]=0`, `data[1]=1`, ..., `data[255]=255` (without the `__syncthreads()` call, only `data[0]=0` would be guaranteed to be seen by the child). The child grid is only guaranteed to return at an implicit synchronization. This means that the modifications made by the threads in the child grid are never guaranteed to become available to the parent grid. To access modifications made by `child_launch`, a `tail_launch` kernel is launched into the `cudaStreamTailLaunch` stream.

```
__global__ void tail_launch(int *data) {
   data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void child_launch(int *data) {
   data[threadIdx.x] = data[threadIdx.x]+1;
}

__global__ void parent_launch(int *data) {
   data[threadIdx.x] = threadIdx.x;

   __syncthreads();

   if (threadIdx.x == 0) {
       child_launch<<< 1, 256 >>>(data);
       tail_launch<<< 1, 256, 0, cudaStreamTailLaunch >>>(data);
   }
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

4.18.3.2 Mapped Memory

Mapped system memory has identical coherence and consistency guarantees to global memory, and follows the semantics detailed above. A kernel may not allocate or free mapped memory, but may use pointers to mapped memory passed in from the host program.

4.18.3.3 Shared and Local Memory

Shared and Local memory is private to a thread block or thread, respectively, and is not visible or coherent between parent and child. Behavior is undefined when an object in one of these locations is referenced outside of the scope within which it belongs, and may cause an error.

The NVIDIA compiler will attempt to warn if it can detect that a pointer to local or shared memory is being passed as an argument to a kernel launch. At runtime, the programmer may use the `__isGlobal()` intrinsic to determine whether a pointer references global memory and so may safely be passed to a child launch.

Calls to `cudaMemcpy*Async()` or `cudaMemset*Async()` may invoke new child kernels on the device in order to preserve stream semantics. As such, passing shared or local memory pointers to these APIs is illegal and will return an error.

4.18.3.4 Local Memory

Local memory is private storage for an executing thread, and is not visible outside of that thread. It is illegal to pass a pointer to local memory as a launch argument when launching a child kernel. The result of dereferencing such a local memory pointer from a child grid is undefined.

For example the following is illegal, with undefined behavior if `x_array` is accessed by `child_launch`:

```
int x_array[10];       // Creates x_array in parent's local memory
child_launch<<< 1, 1 >>>(x_array);
```

It is sometimes difficult for a programmer to be aware of when a variable is placed into local memory by the compiler. As a general rule, all storage passed to a child kernel should be allocated explicitly from the global-memory heap, either with `cudaMalloc()`, `new()` or by declaring `__device__` storage at global scope. For example:

```
// Correct - "value" is global storage
__device__ int value;
__device__ void x() {
    value = 5;
    child<<< 1, 1 >>>(&value);
}
```

```
// Invalid - "value" is local storage
__device__ void y() {
    int value = 5;
    child<<< 1, 1 >>>(&value);
}
```

4.18.3.4.1 Texture Memory

Writes to the global memory region over which a texture is mapped are incoherent with respect to texture accesses. Coherence for texture memory is enforced at the invocation of a child grid and when a child grid completes. This means that writes to memory prior to a child kernel launch are reflected in texture memory accesses of the child. Similarly to Global Memory above, writes to memory by a child are never guaranteed to be reflected in the texture memory accesses by a parent. The only way to access the modifications made by the threads in the child grid before the parent grid exits is via a kernel launched into the `cudaStreamTailLaunch` stream. Concurrent accesses by parent and child may result in inconsistent data.

4.18.4. Programming Interface

4.18.4.1 Basics

The following example shows a simple *Hello World* program incorporating dynamic parallelism:

```
#include <stdio.h>

__global__ void childKernel()
{
```

(continues on next page)

(continued from previous page)

```
    printf("Hello ");
}

__global__ void tailKernel()
{
    printf("World!\n");
}

__global__ void parentKernel()
{
    // launch child
    childKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return;
    }

    // launch tail into cudaStreamTailLaunch stream
    // implicitly synchronizes: waits for child to complete
    tailKernel<<<1,1,0,cudaStreamTailLaunch>>>();

}

int main(int argc, char *argv[])
{
    // launch parent
    parentKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return 1;
    }

    // wait for parent to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return 2;
    }

    return 0;
}
```

This program may be built in a single step from the command line as follows:

```
$ nvcc -arch=sm_75 -rdc=true hello_world.cu -o hello -lcudadevrt
```

4.18.4.2 C++ Language Interface for CDP

The language interface and API available to CUDA kernels using CUDA C++ for Dynamic Parallelism is called the *CUDA Device Runtime*.

Where possible the syntax and semantics of the CUDA Runtime API have been retained in order to facilitate ease of code reuse for routines that may run in either the host or device environments.

As with all code in CUDA C++, the APIs and code outlined here is per-thread code. This enables each thread to make unique, dynamic decisions regarding what kernel or operation to execute next. There are no synchronization requirements between threads within a block to execute any of the provided device runtime APIs, which enables the device runtime API functions to be called in arbitrarily divergent kernel code without deadlock.

4.18.4.2.1 Device-Side Kernel Launch

Kernels may be launched from the device using the standard CUDA <<< >>> syntax:

```
kernel_name<<< Dg, Db, Ns, S >>>([kernel arguments]);
```

- Dg is of type `dim3` and specifies the dimensions and size of the grid
- Db is of type `dim3` and specifies the dimensions and size of each thread block
- Ns is of type `size_t` and specifies the number of bytes of shared memory that is dynamically allocated per thread block for this call in addition to statically allocated memory. Ns is an optional argument that defaults to 0.
- S is of type `cudaStream_t` and specifies the stream associated with this call. The stream must have been allocated in the same grid where the call is being made. S is an optional argument that defaults to the NULL stream.

4.18.4.2.1.1 Launches are Asynchronous

Identical to host-side launches, all device-side kernel launches are asynchronous with respect to the launching thread. That is to say, the <<<>>> launch command will return immediately and the launching thread will continue to execute until it hits an implicit launch-synchronization point, such as at a kernel launched into the `cudaStreamTailLaunch` stream (*The Tail Launch Stream*). The child grid may begin execution at any time after launch, but is not guaranteed to begin execution until the launching thread reaches an implicit launch-synchronization point.

Similar to host-side launch, work launched into separate streams may run concurrently, but actual concurrency is not guaranteed. Programs that depend upon concurrency between child kernels are not supported by the CUDA programming model and will have undefined behavior.

4.18.4.2.1.2 Launch Environment Configuration

All global device configuration settings (for example, shared memory and `L1` cache size as returned from `cudaDeviceGetCacheConfig()`, and device limits returned from `cudaDeviceGetLimit()`) will be inherited from the parent. Likewise, device limits such as stack size will remain as-configured.

For host-launched kernels, per-kernel configurations set from the host will take precedence over the global setting. These configurations will be used when the kernel is launched from the device as well. It is not possible to reconfigure a kernel's environment from the device.

4.18.4.2.2 Events

Only the inter-stream synchronization capabilities of CUDA events are supported. This means that `cudaStreamWaitEvent()` is supported, but `cudaEventSynchronize()`, `cudaEventElapsedTime()`, and `cudaEventQuery()` are not. As `cudaEventElapsedTime()` is not supported, cudaEvents must be created via `cudaEventCreateWithFlags()`, passing the `cudaEventDisableTiming` flag.

As with streams, event objects may be shared between all threads within the grid which created them but are local to that grid and may not be passed to other kernels. Event handles are not guaranteed to be unique between grids, so using an event handle within a grid that did not create it will result in undefined behavior.

4.18.4.2.3 Synchronization

It is up to the program to perform sufficient inter-thread synchronization, for example via a CUDA Event, if the calling thread is intended to synchronize with child grids invoked from other threads.

As it is not possible to explicitly synchronize child work from a parent thread, there is no way to guarantee that changes occurring in child grids are visible to threads within the parent grid.

4.18.4.2.4 Device Management

Only the device on which a kernel is running will be controllable from that kernel. This means that device APIs such as `cudaSetDevice()` are not supported by the device runtime. The active device as seen from the GPU (returned from `cudaGetDevice()`) will have the same device number as seen from the host system. The `cudaDeviceGetAttribute()` call may request information about another device as this API allows specification of a device ID as a parameter of the call. Note that the catch-all `cudaGetDeviceProperties()` API is not offered by the device runtime - properties must be queried individually.

4.18.5. Programming Guidelines

4.18.5.1 Performance

4.18.5.1.1 Dynamic-parallelism-enabled Kernel Overhead

System software which is active when controlling dynamic launches may impose an overhead on any kernel which is running at the time, whether or not it invokes kernel launches of its own. This overhead arises from the device runtime's execution tracking and management software and may result in decreased performance. This overhead is, in general, incurred for applications that link against the device runtime library.

4.18.5.2 Implementation Restrictions and Limitations

Dynamic Parallelism guarantees all semantics described in this document, however, certain hardware and software resources are implementation-dependent and limit the scale, performance and other properties of a program which uses the device runtime.

4.18.5.2.1 Runtime

4.18.5.2.1.1 Memory Footprint

The device runtime system software reserves memory for various management purposes, in particular a reservation for tracking pending grid launches. Configuration controls are available to reduce the size of this reservation in exchange for certain launch limitations. See *Configuration Options*, below, for details.

4.18.5.2.1.2 Pending Kernel Launches

When a kernel is launched, all associated configuration and parameter data is tracked until the kernel completes. This data is stored within a system-managed launch pool.

The size of the fixed-size launch pool is configurable by calling `cudaDeviceSetLimit()` from the host and specifying `cudaLimitDevRuntimePendingLaunchCount`.

4.18.5.3 Compatibility and Interoperability

CDP2 is the default. Functions can be compiled with `-DCUDA_FORCE_CDP1_IF_SUPPORTED` to opt-out of using CDP2 on devices of compute capability less than 9.0.

	Function compiler with CUDA 12.0 and newer (default)	**Function compiled with pre-CUDA 12.0 or with CUDA 12.0 and newer with `-DCUDA_FORCE_CDP1_IF_SUPPORTED` specified**
Compilation	Compile error if device code references `cudaDeviceSynchronize`.	Compile error if code references `cudaStreamTailLaunch` or `cudaStreamFireAndForget`. Compile error if device code references `cudaDeviceSynchronize` and code is compiled for sm_90 or newer.
Compute capability < 9.0	New interface is used.	Legacy interface is used.
Compute capability 9.0 and higher	New interface is used.	New interface is used. If function references `cudaDeviceSynchronize` in device code, function load returns `cudaErrorSymbolNotFound` (this could happen if the code is compiled for devices of compute capability less than 9.0, but run on devices of compute capability 9.0 or higher using JIT).

Functions using CDP1 and CDP2 may be loaded and run simultaneously in the same context. The CDP1 functions are able to use CDP1-specific features (e.g. `cudaDeviceSynchronize`) and CDP2 functions are able to use CDP2-specific features (e.g. tail launch and fire-and-forget launch).

A function using CDP1 cannot launch a function using CDP2, and vice versa. If a function that would use CDP1 contains in its call graph a function that would use CDP2, or vice versa, `cudaErrorCdpVersionMismatch` would result during function load.

The behavior of legacy CDP1 is not included in this document. For information on CDP1, refer back to a legacy version of the CUDA Programming Guide

4.18.6. Device-side Launch from PTX

The previous sections have discussed using the *CUDA Device Runtime* to achieve dynamic parallelism. Dynamic parallelism can also be performed from PTX. For the programming language and compiler implementers who target *Parallel Thread Execution* (PTX) and plan to support *Dynamic Parallelism* in their language, this section provides the low-level details related to supporting kernel launches at the PTX level.

4.18.6.1 Kernel Launch APIs

Device-side kernel launches can be implemented using the following two APIs accessible from PTX: `cudaLaunchDevice()` and `cudaGetParameterBuffer()`. `cudaLaunchDevice()` launches the specified kernel with the parameter buffer that is obtained by calling `cudaGetParameterBuffer()` and filled with the parameters to the launched kernel. The parameter buffer can be NULL, i.e., no need to invoke `cudaGetParameterBuffer()`, if the launched kernel does not take any parameters.

4.18.6.1.1 cudaLaunchDevice

At the PTX level, cudaLaunchDevice()needs to be declared in one of the two forms shown below before it is used.

```
// PTX-level Declaration of cudaLaunchDevice() when .address_size is 64
.extern .func(.param .b32 func_retval0) cudaLaunchDevice
(
  .param .b64 func,
  .param .b64 parameterBuffer,
  .param .align 4 .b8 gridDimension[12],
  .param .align 4 .b8 blockDimension[12],
  .param .b32 sharedMemSize,
  .param .b64 stream
)
;
```

The CUDA-level declaration below is mapped to one of the aforementioned PTX-level declarations and is found in the system header file cuda_device_runtime_api.h. The function is defined in the cudadevrt system library, which must be linked with a program in order to use device-side kernel launch functionality.

```
// CUDA-level declaration of cudaLaunchDevice()
extern "C" __device__
cudaError_t cudaLaunchDevice(void *func, void *parameterBuffer,
                             dim3 gridDimension, dim3 blockDimension,
                             unsigned int sharedMemSize,
                             cudaStream_t stream);
```

The first parameter is a pointer to the kernel to be launched, and the second parameter is the parameter buffer that holds the actual parameters to the launched kernel. The layout of the parameter buffer is explained in *Parameter Buffer Layout*, below. Other parameters specify the launch configuration, i.e., as grid dimension, block dimension, shared memory size, and the stream associated with the launch (please refer to *Kernel Configuration* for the detailed description of launch configuration.

4.18.6.1.2 cudaGetParameterBuffer

cudaGetParameterBuffer() needs to be declared at the PTX level before it's used. The PTX-level declaration must be in one of the two forms given below, depending on address size:

```
// PTX-level Declaration of cudaGetParameterBuffer() when .address_size is 64
.extern .func(.param .b64 func_retval0) cudaGetParameterBuffer
(
  .param .b64 alignment,
  .param .b64 size
)
;
```

The following CUDA-level declaration of cudaGetParameterBuffer() is mapped to the aforementioned PTX-level declaration:

```
// CUDA-level Declaration of cudaGetParameterBuffer()
extern "C" __device__
void *cudaGetParameterBuffer(size_t alignment, size_t size);
```

The first parameter specifies the alignment requirement of the parameter buffer and the second parameter the size requirement in bytes. In the current implementation, the parameter buffer returned by `cudaGetParameterBuffer()` is always guaranteed to be 64- byte aligned, and the alignment requirement parameter is ignored. However, it is recommended to pass the correct alignment requirement value - which is the largest alignment of any parameter to be placed in the parameter buffer - to `cudaGetParameterBuffer()` to ensure portability in the future.

4.18.6.2 Parameter Buffer Layout

Parameter reordering in the parameter buffer is prohibited, and each individual parameter placed in the parameter buffer is required to be aligned. That is, each parameter must be placed at the n^{th} byte in the parameter buffer, where *n* is the smallest multiple of the parameter size that is greater than the offset of the last byte taken by the preceding parameter. The maximum size of the parameter buffer is 4KB.

For a more detailed description of PTX code generated by the CUDA compiler, please refer to the PTX-3.5 specification.

4.19. CUDA Interoperability with APIs

Directly accessing GPU data from APIs in CUDA allows to read and write the data with CUDA kernels and thereby offering CUDA features while consuming them from other APIs. There are two main concepts: the direct approach, *Graphics Interoperability* with openGL and Direct3D[9-11] which enables to map the resources from the OpenGL and the Direct3D to the CUDA address space; and the more flexible *External resource interoperability*, where memory and synchronization objects can be accessed by importing and exporting through OS-level handles. This is supported for the following APIs, Direct3D[11-12], Vulkan and the NVIDIA Software Communication Interface Interoperability.

4.19.1. Graphics Interoperability

Before accessing a Direct3D or an openGL resource, for example a VBO (vertex buffer object) with CUDA, it must be registered and mapped. The registering with the according CUDA functions, see examples below, returns a CUDA graphics resource of type `struct cudaGraphicsResource`, which holds a CUDA device pointer or array. To access the device data in a kernel, the resource must be mapped. While the resource is registered, it can be mapped and unmapped as many times as necessary. A mapped resource is accessed by kernels using the device memory address returned by `cudaGraphicsResourceGetMappedPointer()` for buffers and `cudaGraphicsSubResourceGetMappedArray()` for CUDA arrays. Once the resource is no longer needed by CUDA, it can be unregistered. These are the main steps: 1. Register the graphics buffer with CUDA 2. Map the resource 3. Access the device pointer or array of the mapped resource 4. Use device pointer or array in a CUDA kernel 4. Unmap the resource 5. Unregister the resource

Note that, registering a resource is costly and therefore ideally only called once per resource, however for each CUDA context which intends to use the resource, it is required to register the resource separately. `cudaGraphicsResourceSetMapFlags()` can be called to specify usage hints (write-only, read-only) that the CUDA driver can use to optimize resource management. Further note, that when accessing a resource through OpenGL, Direct3D, or a different CUDA context while it is mapped, it produces undefined results.

4.19.1.1 OpenGL Interoperability

The OpenGL resources that can be mapped into the address space of CUDA are OpenGL buffer, texture, and renderbuffer objects. A buffer object is registered using `cudaGraphicsGLRegisterBuffer()`, in CUDA, it appears as a normal device pointer. A texture or renderbuffer object is registered using `cudaGraphicsGLRegisterImage()`, in CUDA, it appears as a CUDA array.

If a texture or render buffer object has been registered with the `cudaGraphicsRegisterFlagsSurfaceLoadStore` flag, it can be written to. `cudaGraphicsGLRegisterImage()` supports all texture formats with 1, 2, or 4 components and an internal type of float (for example, `GL_RGBA_FLOAT32`), normalized integer (for example, `GL_RGBA8, GL_INTENSITY16`), and unnormalized integer (for example, `GL_RGBA8UI`).

Example:simpleGL interoperability

The following code sample uses a kernel to dynamically modify a 2D `width` x `height` grid of vertices stored in a vertex buffer object (VBO), and goes through the following main steps:

1. Register the VBO with CUDA
2. Loop: Map the VBO for writing from CUDA
3. Loop: Run CUDA kernel to modify the vertex positions
4. Loop: Unmap the VBO
5. Loop: Render the results using OpenGL
6. Unregister and delete VBO

The full example, simpleGL, of this section can be found here, https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/simpleGL .

```
__global__ void simple_vbo_kernel(float4 *pos, unsigned int width, unsigned
↪int height, float time)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    // calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time) * cosf(v * freq + time) * 0.5f;

    // write output vertex
    pos[y * width + x] = make_float4(u, w, v, 1.0f);
}

int main(int argc, char **argv)
{
    char *ref_file = NULL;

    pArgc = &argc;
    pArgv = argv;
```

(continues on next page)

(continued from previous page)

```
#if defined(__linux__)
    setenv("DISPLAY", ":0", 0);
#endif

    printf("%s starting...\n", sSDKsample);

    if (argc > 1) {
        if (checkCmdLineFlag(argc, (const char **)argv, "file")) {
            // In this mode, we are running non-OpenGL and doing a compare of
↪the VBO was generated correctly
            getCmdLineArgumentString(argc, (const char **)argv, "file", (char
↪**)&ref_file);
        }
    }

    printf("\n");

    // First initialize OpenGL context
    if (false == initGL(&argc, argv)) {
        return false;
    }

    // register callbacks
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMouseFunc(mouse);
    glutMotionFunc(motion);
    glutCloseFunc(cleanup);

    // Create an empty vertex buffer object (VBO)
    // 1. Register the VBO with CUDA
    createVBO(&vbo, &cuda_vbo_resource, cudaGraphicsMapFlagsWriteDiscard);

    // start rendering mainloop
    //  5. Render the results using OpenGL
    glutMainLoop();

    printf("%s completed, returned %s\n", sSDKsample, (g_TotalErrors == 0) ?
↪"OK" : "ERROR!");
    exit(g_TotalErrors == 0 ? EXIT_SUCCESS : EXIT_FAILURE);

}

void createVBO(GLuint *vbo, struct cudaGraphicsResource **vbo_res, unsigned
↪int vbo_res_flags)
{
    assert(vbo);

    // create buffer object
    glGenBuffers(1, vbo);
    glBindBuffer(GL_ARRAY_BUFFER, *vbo);
```

(continues on next page)

(continued from previous page)

```
    // initialize buffer object
    unsigned int size = mesh_width * mesh_height * 4 * sizeof(float);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, 0);

    // register this buffer object with CUDA
    checkCudaErrors(cudaGraphicsGLRegisterBuffer(vbo_res, *vbo, vbo_res_
↪flags));

    SDK_CHECK_ERROR_GL();
}

void display()
{
    float4 *dptr;
    // 2. Map the VBO for writing from CUDA
    checkCudaErrors(cudaGraphicsMapResources(1, &cuda_vbo_resource, 0));
    size_t num_bytes;
    checkCudaErrors(cudaGraphicsResourceGetMappedPointer((void **)&dptr, &num_
↪bytes, cuda_vbo_resource));

    // 3. Run CUDA kernel to modify the vertex positions
    //call the CUDA kernel
    dim3 block(8, 8, 1);
    dim3 grid(mesh_width / block.x, mesh_height / block.y, 1);
    simple_vbo_kernel<<<grid, block>>>(dptr, mesh_width, mesh_height, g_
↪fAnim);

    //  4. Unmap the VBO
    checkCudaErrors(cudaGraphicsUnmapResources(1, &cuda_vbo_resource, 0));

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // set view matrix
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, translate_z);
    glRotatef(rotate_x, 1.0, 0.0, 0.0);
    glRotatef(rotate_y, 0.0, 1.0, 0.0);

    // 5. Render the updated  using OpenGL
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glVertexPointer(4, GL_FLOAT, 0, 0);

    glEnableClientState(GL_VERTEX_ARRAY);
    glColor3f(1.0, 0.0, 0.0);
    glDrawArrays(GL_POINTS, 0, mesh_width * mesh_height);
    glDisableClientState(GL_VERTEX_ARRAY);

    glutSwapBuffers();
```

(continues on next page)

(continued from previous page)

```
    g_fAnim += 0.01f;

}

void deleteVBO(GLuint *vbo, struct cudaGraphicsResource *vbo_res)
{
    // 6. Unregister and delete VBO
    checkCudaErrors(cudaGraphicsUnregisterResource(vbo_res));

    glBindBuffer(1, *vbo);
    glDeleteBuffers(1, vbo);

    *vbo = 0;
}

void cleanup()
{

    if (vbo) {
        deleteVBO(&vbo, cuda_vbo_resource);
    }
}
```

Limitations and considerations.

- The OpenGL context whose resources are being shared has to be current to the host thread making any OpenGL interoperability API calls.
- When an OpenGL texture is made bindless (say for example by requesting an image or texture handle using the `glGetTextureHandle` or `glGetImageHandle` APIs) it cannot be registered with CUDA. The application needs to register the texture for interop before requesting an image or texture handle.

4.19.1.2 Direct3D Interoperability

Direct3D interoperability is supported for Direct3D9, Direct3D10, and Direct3D11 but not Direct3D12, here we focus on Direct3D11, for Direct3D9 and Direct3D10 please refer to the CUDA programming guide 12.9. The Direct3D resources that may be mapped into the address space of CUDA are Direct3D buffers, textures, and surfaces. These resources are registered using `cudaGraphicsD3D11RegisterResource()`.

A CUDA context may interoperate only with Direct3D11 devices created with `DriverType` set to `D3D_DRIVER_TYPE_HARDWARE`.

Example: 2D Texture Direct3D11 interoperability

The following code snippets are from the simpleD3D11Texture example, https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/simpleD3D11Texture. The full example includes a lot of boiler plate DX11 code, here we focus on the CUDA side.

The CUDA kernel `cuda_kernel_texture_2d` paints a 2D texture with a moving red/green hatch pattern on a strobing blue background, it is dependent on the previous texture values. The underlying data is a 2D CUDA array, where the row offsets are defined by the pitch.

```
/*
 * Paint a 2D texture with a moving red/green hatch pattern on a
 * strobing blue background.  Note that this kernel reads to and
 * writes from the texture, hence why this texture was not mapped
 * as WriteDiscard.
 */
__global__ void cuda_kernel_texture_2d(unsigned char *surface, int width,
                                        int height, size_t pitch, float t) {
  int x = blockIdx.x * blockDim.x + threadIdx.x;
  int y = blockIdx.y * blockDim.y + threadIdx.y;
  float *pixel;

  // in the case where, due to quantization into grids, we have
  // more threads than pixels, skip the threads which don't
  // correspond to valid pixels
  if (x >= width || y >= height) return;

  // get a pointer to the pixel at (x,y)
  pixel = (float *)(surface + y * pitch) + 4 * x;

  // populate it
  float value_x = 0.5f + 0.5f * cos(t + 10.0f * ((2.0f * x) / width - 1.0f));
  float value_y = 0.5f + 0.5f * cos(t + 10.0f * ((2.0f * y) / height - 1.0f));
  pixel[0] = 0.5 * pixel[0] + 0.5 * pow(value_x, 3.0f);  // red
  pixel[1] = 0.5 * pixel[1] + 0.5 * pow(value_y, 3.0f);  // green
  pixel[2] = 0.5f + 0.5f * cos(t);                       // blue
  pixel[3] = 1;                                          // alpha
}

extern "C" void cuda_texture_2d(void *surface, int width, int height,
                                size_t pitch, float t) {
  cudaError_t error = cudaSuccess;

  dim3 Db = dim3(16, 16);  // block dimensions are fixed to be 256 threads
  dim3 Dg = dim3((width + Db.x - 1) / Db.x, (height + Db.y - 1) / Db.y);

  cuda_kernel_texture_2d<<<Dg, Db>>>((unsigned char *)surface, width, height,
                                     pitch, t);

  error = cudaGetLastError();

  if (error != cudaSuccess) {
    printf("cuda_kernel_texture_2d() failed to launch error = %d\n", error);
  }
}
```

To keep the pointers and data buffers belonging together the following data structure is used:

```
// Data structure for 2D texture shared between DX11 and CUDA
struct {
  ID3D11Texture2D *pTexture;
  ID3D11ShaderResourceView *pSRView;
  cudaGraphicsResource *cudaResource;
```

(continues on next page)

(continued from previous page)

```
  void *cudaLinearMemory;
  size_t pitch;
  int width;
  int height;
  int offsetInShader;
} g_texture_2d;
```

After the initialization of the Direct3D device and the textures, the resources are registered with CUDA once. To match the Direct3D pixel format, the CUDA array is allocated with the same width and height, and a pitch matching the Direct3D texture row pitch.

```
    // register the Direct3D resources that are used in the CUDA kernel
    // we'll read to and write from g_texture_2d, so don't set any special map
↪flags for it
    cudaGraphicsD3D11RegisterResource(&g_texture_2d.cudaResource,
                                      g_texture_2d.pTexture,
                                      cudaGraphicsRegisterFlagsNone);
    getLastCudaError("cudaGraphicsD3D11RegisterResource (g_texture_2d) failed
↪");
    // CUDA cannot write into the texture directly : the texture is seen as a
    // cudaArray and can only be mapped as a texture
    // Create a buffer so that CUDA can write into it
    // the pixel fmt is DXGI_FORMAT_R32G32B32A32_FLOAT
    cudaMallocPitch(&g_texture_2d.cudaLinearMemory, &g_texture_2d.pitch,
                    g_texture_2d.width * sizeof(float) * 4,
                    g_texture_2d.height);
    getLastCudaError("cudaMallocPitch (g_texture_2d) failed");
    cudaMemset(g_texture_2d.cudaLinearMemory, 1,
               g_texture_2d.pitch * g_texture_2d.height);
```

In the rendering loop, the resources are mapped, the CUDA kernel is launched to update the texture data, and then the resources are unmapped. After this step the Direct3D device is used to draw the updated textures on the screen.

```
    cudaStream_t stream = 0;
    const int nbResources = 3;
    cudaGraphicsResource *ppResources[nbResources] = {
        g_texture_2d.cudaResource, g_texture_3d.cudaResource,
        g_texture_cube.cudaResource,
    };
    cudaGraphicsMapResources(nbResources, ppResources, stream);
    getLastCudaError("cudaGraphicsMapResources(3) failed");

    // run kernels which will populate the contents of those textures
    RunKernels();

    // unmap the resources
    cudaGraphicsUnmapResources(nbResources, ppResources, stream);
    getLastCudaError("cudaGraphicsUnmapResources(3) failed");
```

Finally, once the resources are no longer needed in CUDA, they are unregistered and the device array freed.

```
// unregister the Cuda resources
cudaGraphicsUnregisterResource(g_texture_2d.cudaResource);
getLastCudaError("cudaGraphicsUnregisterResource (g_texture_2d) failed");
cudaFree(g_texture_2d.cudaLinearMemory);
getLastCudaError("cudaFree (g_texture_2d) failed");
```

4.19.1.3 Interoperability in a Scalable Link Interface (SLI) configuration

In a system with multiple GPUs, all CUDA-enabled GPUs are accessible via the CUDA driver and runtime as separate devices. This is different when the system is in SLI mode. SLI is a hardware configured multi-GPU configuration that offers increased rendering performance by dividing the workload across multiple GPUs. Implicit SLI mode, where the driver makes assumption is no longer supported, however explicit SLI is still supported. Explicit SLI means applications know and manage the SLI state through APIs (e.g. Vulkan, DirectX, GL) for all devices in the SLI group.

There are special considerations when the system is in SLI mode:

- An allocation in one CUDA device on one GPU will consume memory on other GPUs that are part of the SLI configuration of the Direct3D or OpenGL device. Because of this, allocations may fail earlier than otherwise expected.
- An applications should create multiple CUDA contexts, one for each GPU in the SLI configuration. While this is not a strict requirement, it avoids unnecessary data transfers between devices. The application can use the `cudaD3D[9|10|11]GetDevices()` for Direct3D and `cudaGLGetDevices()` for OpenGL set of calls to identify the CUDA device handles for the devices that are performing the rendering in the current and next frame. Given this information the application will typically choose the appropriate device and map Direct3D or OpenGL resources to the CUDA device returned by `cudaD3D[9|10|11]GetDevices()` or `cudaGLGetDevices()` when the `deviceList` parameter is set to `cudaD3D[9|10|11]DeviceListCurrentFrame` or `cudaGLDeviceListCurrentFrame`.
- Resource returned from `cudaGraphicsD3D[9|10|11]RegisterResource` and `cudaGraphicsGLRegister[Buffer|Image]` must be only used on the device where the registration happened. Therefore, in SLI configurations when data for different frames is computed on different CUDA devices it is necessary to register the resources for each separately.

4.19.2. External resource interoperability

External resource interoperability allows CUDA to import certain resources that are explicitly exported by APIs. These objects are typically exported using handles native to the Operating System, like file descriptors on Linux or NT handles on Windows. This allows to efficiently share the resource between other APIs and CUDA without the need to copy or duplicate in between. It is supported for the following APIs, Direct3D[11-12], Vulkan and the NVIDIA Software Communication Interface Interoperability. There are two types of resources that can be imported:

- **Memory objects**

  can be imported into CUDA using `cudaImportExternalMemory()`. An imported memory object can then be accessed from within kernels using device pointers mapped onto the memory object with `cudaExternalMemoryGetMappedBuffer()` or CUDA mipmapped arrays mapped with `cudaExternalMemoryGetMappedMipmappedArray()`. Depending on the type of memory object, it may be possible for more than one mapping to be setup on a single memory object. The mappings must match the mappings setup of the exporting API. Any mismatched mappings result in undefined behavior. Imported memory objects must be freed using `cudaDestroyExternalMemory()`. Freeing a memory object does not free any mappings to that object. Therefore, any device pointers mapped onto that object must be explicitly freed using `cudaFree()` and any CUDA mipmapped arrays mapped onto that object must be explicitly freed using `cudaFreeMipmappedArray()`. It is illegal to access mappings to an object after it has been destroyed.

▶**Synchronization objects**

can be imported into CUDA using `cudaImportExternalSemaphore()`. An imported synchronization object can then be signaled using `cudaSignalExternalSemaphoresAsync()` and waited on using `cudaWaitExternalSemaphoresAsync()`. It is illegal to issue a wait before the corresponding signal has been issued. Also, depending on the type of the imported synchronization object, there may be additional constraints imposed on how they can be signaled and waited on, as described in subsequent sections. Imported semaphore objects must be freed using `cudaDestroyExternalSemaphore()`. All outstanding signals and waits must have completed before the semaphore object is destroyed.

4.19.2.1 Vulkan interoperability

Coupled execution of Vulkan graphics and compute workloads on the same hardware can maximize GPU utilization and avoid unnecessary copies. Note, this is not a Vulkan guide, we only focus on the interoperability with CUDA, for a Vulkan guide please refer to https://www.vulkan.org/learn#vulkan-tutorials.

The main steps to get a Vulkan-CUDA interoperability working involve:

1. Initialize Vulkan, create and export the external buffers and/or synchronization objects
2. Set the CUDA device where Vulkan is running with the matching devices UUIDs
3. Get the memory and/or synchronization handle
4. Import the memory and/or synchronization object in CUDA using these handles
5. Map the device pointer or mipmapped array onto the memory object
6. Use the imported memory objects in CUDA and Vulkan interchangeably by defining an order of execution through signaling and waiting on the synchronization objects.

In this section the steps above are explained with the help of the *simpleVulkan* example, https://github.com/NVIDIA/cuda-samples/tree/master/Samples/5_Domain_Specific/simpleVulkan. We walk through the example step by step, focusing on the parts needed for the CUDA interoperability. Some variation are explained with standalone snippets.

Note

The code example used in this section, uses the direct memory allocation and resource creation. Which is not state of the art due to several reasons, including the limitation to the number of instances that can be created. However, to understand the interoperability, one needs to know the underlying Vulkan code and the specific flags. For a more state of the art example, using the VulkanMemoryAllocator please refer to the *sample_cuda_interop* in the NVProSamples repository.

The following data structure is used throughout the example:

```
class VulkanCudaSineWave : public VulkanBaseApp {
  typedef struct UniformBufferObject_st {
    mat4x4 modelViewProj;
  } UniformBufferObject;
```

(continues on next page)

(continued from previous page)

```
  VkBuffer m_heightBuffer, m_xyBuffer, m_indexBuffer;
  VkDeviceMemory m_heightMemory, m_xyMemory, m_indexMemory;
  UniformBufferObject m_ubo;
  VkSemaphore m_vkWaitSemaphore, m_vkSignalSemaphore;
  SineWaveSimulation m_sim;
  cudaStream_t m_stream;
  cudaExternalSemaphore_t m_cudaWaitSemaphore, m_cudaSignalSemaphore, m_
↪cudaTimelineSemaphore;
  cudaExternalMemory_t m_cudaVertMem;
  float *m_cudaHeightMap;
  // ...
```

4.19.2.1.1 Setting up a Vulkan device

In order to export memory objects, a Vulkan instance must be created with the VK_KHR_external_memory_capabilities extension enabled and the device with VK_KHR_external_memory. In addition to the the platform specific handle types must be enabled, for Windows VK_KHR_external_memory_win32 and for UNIX based systems VK_KHR_external_memory_fd.

Similarly for exporting synchronization objects, on the device level VK_KHR_external_semaphore_capabilities and VK_KHR_external_semaphore on the instance level need to be enabled. As well as the platform specific extensions for the handles, that is VK_KHR_external_semaphore_win32 for Windows and VK_KHR_external_semaphore_fd for Unix based systems.

In the *simpleVulkan* example these extensions are enabled with the following enums.

```
  std::vector<const char *> getRequiredExtensions() const {
    std::vector<const char *> extensions;
    extensions.push_back(VK_KHR_EXTERNAL_MEMORY_CAPABILITIES_EXTENSION_NAME);
    extensions.push_back(VK_KHR_EXTERNAL_SEMAPHORE_CAPABILITIES_EXTENSION_
↪NAME);
    return extensions;
  }

  std::vector<const char *> getRequiredDeviceExtensions() const {
    std::vector<const char *> extensions;
    extensions.push_back(VK_KHR_EXTERNAL_MEMORY_EXTENSION_NAME);
    extensions.push_back(VK_KHR_EXTERNAL_SEMAPHORE_EXTENSION_NAME);
    extensions.push_back(VK_KHR_TIMELINE_SEMAPHORE_EXTENSION_NAME);
#ifdef _WIN64
    extensions.push_back(VK_KHR_EXTERNAL_MEMORY_WIN32_EXTENSION_NAME);
    extensions.push_back(VK_KHR_EXTERNAL_SEMAPHORE_WIN32_EXTENSION_NAME);
#else
    extensions.push_back(VK_KHR_EXTERNAL_MEMORY_FD_EXTENSION_NAME);
    extensions.push_back(VK_KHR_EXTERNAL_SEMAPHORE_FD_EXTENSION_NAME);
#endif /* _WIN64 */
    return extensions;
  }
```

These are then added to the Vulkan instance and device creation info, please see the *simpleVulkan* example for details.

4.19.2.1.2 Initializing CUDA with matching device UUIDs

When importing memory and synchronization objects exported by Vulkan, they must be imported and mapped on the same device as they were created on. The CUDA device that corresponds to the Vulkan physical device on which the objects were created can be determined by comparing the UUID of a CUDA device with that of the Vulkan physical device, as shown in the following code snippet from the simpleVulkan example, where `vkDeviceUUID` is the member of the Vulkan API structure `vkPhysicalDeviceIDProperties.deviceUUID` and defines the physical devices id of the current Vulkan instance.

```
// from the CUDA example `simpleVulkan`
int SineWaveSimulation::initCuda(uint8_t *vkDeviceUUID, size_t UUID_SIZE) {
  int current_device = 0;
  int device_count = 0;
  int devices_prohibited = 0;

  cudaDeviceProp deviceProp;
  checkCudaErrors(cudaGetDeviceCount(&device_count));

  if (device_count == 0) {
    fprintf(stderr, "CUDA error: no devices supporting CUDA.\n");
    exit(EXIT_FAILURE);
  }

  // Find the GPU which is selected by Vulkan
  while (current_device < device_count) {
    cudaGetDeviceProperties(&deviceProp, current_device);

    if ((deviceProp.computeMode != cudaComputeModeProhibited)) {
      // Compare the cuda device UUID with vulkan UUID
      int ret = memcmp((void *)&deviceProp.uuid, vkDeviceUUID, UUID_SIZE);
      if (ret == 0) {
        checkCudaErrors(cudaSetDevice(current_device));
        checkCudaErrors(cudaGetDeviceProperties(&deviceProp, current_device));
        printf("GPU Device %d: \"%s\" with compute capability %d.%d\n\n",
               current_device, deviceProp.name, deviceProp.major,
               deviceProp.minor);

        return current_device;
      }

    } else {
      devices_prohibited++;
    }

    current_device++;
  }

  if (devices_prohibited == device_count) {
    fprintf(stderr,
            "CUDA error:"
            " No Vulkan-CUDA Interop capable GPU found.\n");
    exit(EXIT_FAILURE);
```

(continues on next page)

(continued from previous page)

```
  }

  return -1;
}
```

Note that the Vulkan physical device should not be part of a device group that contains more than one Vulkan physical device. That is, the device group as returned by `vkEnumeratePhysicalDeviceGroups` that contains the given Vulkan physical device must have a physical device count of 1.

4.19.2.1.3 Exporting Vulkan memory objects

In order to export a Vulkan memory object, a buffer with the according export flags must be created. Note that the enums for the handle types are platform specific.

```
void VulkanBaseApp::createExternalBuffer(
    VkDeviceSize size, VkBufferUsageFlags usage,
    VkMemoryPropertyFlags properties,
    VkExternalMemoryHandleTypeFlagsKHR extMemHandleType, VkBuffer &buffer,
    VkDeviceMemory &bufferMemory) {
  VkBufferCreateInfo bufferInfo = {};
  bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
  bufferInfo.size = size;
  bufferInfo.usage = usage;
  bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

  VkExternalMemoryBufferCreateInfo externalMemoryBufferInfo = {};
  externalMemoryBufferInfo.sType =
      VK_STRUCTURE_TYPE_EXTERNAL_MEMORY_BUFFER_CREATE_INFO;
  externalMemoryBufferInfo.handleTypes = extMemHandleType;
  bufferInfo.pNext = &externalMemoryBufferInfo;

  if (vkCreateBuffer(m_device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
    throw std::runtime_error("failed to create buffer!");
  }

  VkMemoryRequirements memRequirements;
  vkGetBufferMemoryRequirements(m_device, buffer, &memRequirements);

#ifdef _WIN64
  WindowsSecurityAttributes winSecurityAttributes;

  VkExportMemoryWin32HandleInfoKHR vulkanExportMemoryWin32HandleInfoKHR = {};
  vulkanExportMemoryWin32HandleInfoKHR.sType =
      VK_STRUCTURE_TYPE_EXPORT_MEMORY_WIN32_HANDLE_INFO_KHR;
  vulkanExportMemoryWin32HandleInfoKHR.pNext = NULL;
  vulkanExportMemoryWin32HandleInfoKHR.pAttributes = &winSecurityAttributes;
  vulkanExportMemoryWin32HandleInfoKHR.dwAccess =
      DXGI_SHARED_RESOURCE_READ | DXGI_SHARED_RESOURCE_WRITE;
  vulkanExportMemoryWin32HandleInfoKHR.name = (LPCWSTR)NULL;
#endif /* _WIN64 */
  VkExportMemoryAllocateInfoKHR vulkanExportMemoryAllocateInfoKHR = {};
  vulkanExportMemoryAllocateInfoKHR.sType =
```

(continues on next page)

(continued from previous page)

```
      VK_STRUCTURE_TYPE_EXPORT_MEMORY_ALLOCATE_INFO_KHR;
#ifdef _WIN64
  vulkanExportMemoryAllocateInfoKHR.pNext =
      extMemHandleType & VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT_KHR
          ? &vulkanExportMemoryWin32HandleInfoKHR
          : NULL;
  vulkanExportMemoryAllocateInfoKHR.handleTypes = extMemHandleType;
#else
  vulkanExportMemoryAllocateInfoKHR.pNext = NULL;
  vulkanExportMemoryAllocateInfoKHR.handleTypes =
      VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT;
#endif /* _WIN64 */
  VkMemoryAllocateInfo allocInfo = {};
  allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
  allocInfo.pNext = &vulkanExportMemoryAllocateInfoKHR;
  allocInfo.allocationSize = memRequirements.size;
  allocInfo.memoryTypeIndex = findMemoryType(
      m_physicalDevice, memRequirements.memoryTypeBits, properties);

  if (vkAllocateMemory(m_device, &allocInfo, nullptr, &bufferMemory) !=
      VK_SUCCESS) {
    throw std::runtime_error("failed to allocate external buffer memory!");
  }

  vkBindBufferMemory(m_device, buffer, bufferMemory, 0);
}
```

4.19.2.1.4 Exporting Vulkan synchronization objects

Vulkan API calls which are executed on the GPU are asynchronous. To define an order of execution there are semaphores and fences available in Vulkan which can be shared with CUDA. Similar to the memory objects, semaphores can be exported by Vulkan, they need to be created with the export flags depending on the type of semaphore. There are binary and timeline semaphores. Binary semaphores only have a 1 bit counter, either signaled or not. Timeline semaphores have a 64 bit counter, which can be used to define an order of execution with the same semaphore. In the *simpleVulkan* example there are code paths for both timeline and binary semaphores.

```
void VulkanBaseApp::createExternalSemaphore(
    VkSemaphore &semaphore, VkExternalSemaphoreHandleTypeFlagBits handleType)
↪{
  VkSemaphoreCreateInfo semaphoreInfo = {};
  semaphoreInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_CREATE_INFO;
  VkExportSemaphoreCreateInfoKHR exportSemaphoreCreateInfo = {};
  exportSemaphoreCreateInfo.sType =
      VK_STRUCTURE_TYPE_EXPORT_SEMAPHORE_CREATE_INFO_KHR;

#ifdef _VK_TIMELINE_SEMAPHORE
  VkSemaphoreTypeCreateInfo timelineCreateInfo;
  timelineCreateInfo.sType = VK_STRUCTURE_TYPE_SEMAPHORE_TYPE_CREATE_INFO;
  timelineCreateInfo.pNext = NULL;
  timelineCreateInfo.semaphoreType = VK_SEMAPHORE_TYPE_TIMELINE;
```

(continues on next page)

(continued from previous page)

```
  timelineCreateInfo.initialValue = 0;
  exportSemaphoreCreateInfo.pNext = &timelineCreateInfo;
#else
  exportSemaphoreCreateInfo.pNext = NULL;
#endif /* _VK_TIMELINE_SEMAPHORE */
  exportSemaphoreCreateInfo.handleTypes = handleType;
  semaphoreInfo.pNext = &exportSemaphoreCreateInfo;

  if (vkCreateSemaphore(m_device, &semaphoreInfo, nullptr, &semaphore) !=
      VK_SUCCESS) {
    throw std::runtime_error(
        "failed to create synchronization objects for a CUDA-Vulkan!");
  }
}
```

4.19.2.1.5 Importing memory objects

Both dedicated and non-dedicated memory objects exported by Vulkan can be imported into CUDA. When importing a Vulkan dedicated memory object, the flag `cudaExternalMemoryDedicated` must be set.

In Windows, a Vulkan memory object exported using `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT` can be imported into CUDA using the NT handle associated with that object as shown below. Note that CUDA does not assume ownership of the NT handle and it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying memory can be freed.

In Linux, a Vulkan memory object exported using `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_FD_BIT` can be imported into CUDA using the file descriptor associated with that object as shown below. Note that CUDA assumes ownership of the file descriptor once it is imported. Using the file descriptor after a successful import results in undefined behavior.

```
  // from the CUDA example `simpleVulkan`
  void importCudaExternalMemory(void **cudaPtr, cudaExternalMemory_t &cudaMem,
                                VkDeviceMemory &vkMem, VkDeviceSize size,
                                VkExternalMemoryHandleTypeFlagBits
↪handleType) {
    cudaExternalMemoryHandleDesc externalMemoryHandleDesc = {};

    if (handleType & VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT) {
      externalMemoryHandleDesc.type = cudaExternalMemoryHandleTypeOpaqueWin32;
    } else if (handleType &
               VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT) {
      externalMemoryHandleDesc.type =
          cudaExternalMemoryHandleTypeOpaqueWin32Kmt;
    } else if (handleType & VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT) {
      externalMemoryHandleDesc.type = cudaExternalMemoryHandleTypeOpaqueFd;
    } else {
      throw std::runtime_error("Unknown handle type requested!");
    }

    externalMemoryHandleDesc.size = size;
```

(continues on next page)

(continued from previous page)

```
#ifdef _WIN64
    externalMemoryHandleDesc.handle.win32.handle =
        (HANDLE)getMemHandle(vkMem, handleType);
#else
    externalMemoryHandleDesc.handle.fd =
        (int)(uintptr_t)getMemHandle(vkMem, handleType);
#endif

    checkCudaErrors(
        cudaImportExternalMemory(&cudaMem, &externalMemoryHandleDesc));
```

A Vulkan memory object exported using `VK_EXTERNAL_MEMORY_HANDLE_TYPE_OPAQUE_WIN32_BIT` can also be imported using a named handle if one exists as shown in the standalone snippet below.

```
cudaExternalMemory_t importVulkanMemoryObjectFromNamedNTHandle(LPCWSTR name,
↪unsigned long long size, bool isDedicated) {
   cudaExternalMemory_t extMem = NULL;
   cudaExternalMemoryHandleDesc desc = {};

   memset(&desc, 0, sizeof(desc));

   desc.type = cudaExternalMemoryHandleTypeOpaqueWin32;
   desc.handle.win32.name = (void *)name;
   desc.size = size;
   if (isDedicated) {
       desc.flags |= cudaExternalMemoryDedicated;
   }

   cudaImportExternalMemory(&extMem, &desc);

   return extMem;
}
```

4.19.2.1.6 Mapping buffers onto imported memory objects

After importing a memory object, they have to be mapped before they can be used. A device pointer can be mapped onto an imported memory object as shown below. The offset and size of the mapping must match that specified when creating the mapping using the corresponding Vulkan API. All mapped device pointers must be freed using `cudaFree()`.

```
    // from the CUDA example `simpleVulkan`, continuation of function
↪`importCudaExternalMemory`
    cudaExternalMemoryBufferDesc externalMemBufferDesc = {};
    externalMemBufferDesc.offset = 0;
    externalMemBufferDesc.size = size;
    externalMemBufferDesc.flags = 0;

    checkCudaErrors(cudaExternalMemoryGetMappedBuffer(cudaPtr, cudaMem,
                                                             &
↪externalMemBufferDesc));
  }
```

4.19.2.1.7 Mapping mipmapped arrays onto imported memory objects

A CUDA mipmapped array can be mapped onto an imported memory object as shown below. The offset, dimensions, format and number of mip levels must match that specified when creating the mapping using the corresponding Vulkan API. Additionally, if the mipmapped array is bound as a color target in Vulkan, the flag`cudaArrayColorAttachment` must be set. All mapped mipmapped arrays must be freed using `cudaFreeMipmappedArray()`. The following code standalone snippet shows how to convert Vulkan parameters into the corresponding CUDA parameters when mapping mipmapped arrays onto imported memory objects.

```
cudaMipmappedArray_t mapMipmappedArrayOntoExternalMemory(cudaExternalMemory_t
↪extMem, unsigned long long offset, cudaChannelFormatDesc *formatDesc,
↪cudaExtent *extent, unsigned int flags, unsigned int numLevels) {
    cudaMipmappedArray_t mipmap = NULL;
    cudaExternalMemoryMipmappedArrayDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.formatDesc = *formatDesc;
    desc.extent = *extent;
    desc.flags = flags;
    desc.numLevels = numLevels;

    // Note: 'mipmap' must eventually be freed using cudaFreeMipmappedArray()
    cudaExternalMemoryGetMappedMipmappedArray(&mipmap, extMem, &desc);

    return mipmap;
}
//end mapMipmappedArrayOntoExternalMemory

//begin getCudaChannelFormatDescForVulkanFormat
cudaChannelFormatDesc getCudaChannelFormatDescForVulkanFormat(VkFormat format)
{
    cudaChannelFormatDesc d;

    memset(&d, 0, sizeof(d));

    switch (format) {
       case VK_FORMAT_R8_UINT:             d.x = 8;  d.y = 0;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindUnsigned; break;
       case VK_FORMAT_R8_SINT:             d.x = 8;  d.y = 0;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindSigned;   break;
       case VK_FORMAT_R8G8_UINT:           d.x = 8;  d.y = 8;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindUnsigned; break;
       case VK_FORMAT_R8G8_SINT:           d.x = 8;  d.y = 8;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindSigned;   break;
       case VK_FORMAT_R8G8B8A8_UINT:       d.x = 8;  d.y = 8;  d.z = 8;  d.w
↪= 8;  d.f = cudaChannelFormatKindUnsigned; break;
       case VK_FORMAT_R8G8B8A8_SINT:       d.x = 8;  d.y = 8;  d.z = 8;  d.w
↪= 8;  d.f = cudaChannelFormatKindSigned;   break;
       case VK_FORMAT_R16_UINT:            d.x = 16; d.y = 0;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindUnsigned; break;
```

(continues on next page)

(continued from previous page)

```
        case VK_FORMAT_R16_SINT:            d.x = 16; d.y = 0;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindSigned;   break;
        case VK_FORMAT_R16G16_UINT:         d.x = 16; d.y = 16; d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case VK_FORMAT_R16G16_SINT:         d.x = 16; d.y = 16; d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindSigned;   break;
        case VK_FORMAT_R16G16B16A16_UINT:   d.x = 16; d.y = 16; d.z = 16; d.w
↪= 16; d.f = cudaChannelFormatKindUnsigned; break;
        case VK_FORMAT_R16G16B16A16_SINT:   d.x = 16; d.y = 16; d.z = 16; d.w
↪= 16; d.f = cudaChannelFormatKindSigned;   break;
        case VK_FORMAT_R32_UINT:            d.x = 32; d.y = 0;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case VK_FORMAT_R32_SINT:            d.x = 32; d.y = 0;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindSigned;   break;
        case VK_FORMAT_R32_SFLOAT:          d.x = 32; d.y = 0;  d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindFloat;    break;
        case VK_FORMAT_R32G32_UINT:         d.x = 32; d.y = 32; d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case VK_FORMAT_R32G32_SINT:         d.x = 32; d.y = 32; d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindSigned;   break;
        case VK_FORMAT_R32G32_SFLOAT:       d.x = 32; d.y = 32; d.z = 0;  d.w
↪= 0;  d.f = cudaChannelFormatKindFloat;    break;
        case VK_FORMAT_R32G32B32A32_UINT:   d.x = 32; d.y = 32; d.z = 32; d.w
↪= 32; d.f = cudaChannelFormatKindUnsigned; break;
        case VK_FORMAT_R32G32B32A32_SINT:   d.x = 32; d.y = 32; d.z = 32; d.w
↪= 32; d.f = cudaChannelFormatKindSigned;   break;
        case VK_FORMAT_R32G32B32A32_SFLOAT: d.x = 32; d.y = 32; d.z = 32; d.w
↪= 32; d.f = cudaChannelFormatKindFloat;    break;
        default: assert(0);
    }
    return d;
}
//end getCudaChannelFormatDescForVulkanFormat

//begin getCudaExtentForVulkanExtent
cudaExtent getCudaExtentForVulkanExtent(VkExtent3D vkExt, uint32_t
↪arrayLayers, VkImageViewType vkImageViewType) {
    cudaExtent e = { 0, 0, 0 };

    switch (vkImageViewType) {
        case VK_IMAGE_VIEW_TYPE_1D:         e.width = vkExt.width; e.height =
↪0;             e.depth = 0;           break;
        case VK_IMAGE_VIEW_TYPE_2D:         e.width = vkExt.width; e.height =
↪vkExt.height; e.depth = 0;           break;
        case VK_IMAGE_VIEW_TYPE_3D:         e.width = vkExt.width; e.height =
↪vkExt.height; e.depth = vkExt.depth; break;
        case VK_IMAGE_VIEW_TYPE_CUBE:       e.width = vkExt.width; e.height =
↪vkExt.height; e.depth = arrayLayers; break;
        case VK_IMAGE_VIEW_TYPE_1D_ARRAY:   e.width = vkExt.width; e.height =
↪0;             e.depth = arrayLayers; break;
        case VK_IMAGE_VIEW_TYPE_2D_ARRAY:   e.width = vkExt.width; e.height =
↪vkExt.height; e.depth = arrayLayers; break;
```

(continues on next page)

(continued from previous page)

```
        case VK_IMAGE_VIEW_TYPE_CUBE_ARRAY: e.width = vkExt.width; e.height =
↪vkExt.height; e.depth = arrayLayers; break;
        default: assert(0);
    }

    return e;
}
//end getCudaExtentForVulkanExtent

//begin getCudaMipmappedArrayFlagsForVulkanImage
unsigned int getCudaMipmappedArrayFlagsForVulkanImage(VkImageViewType
↪vkImageViewType,
                                                      VkImageUsageFlags
↪vkImageUsageFlags,
                                                      bool
↪allowSurfaceLoadStore) {
    unsigned int flags = 0;

    switch (vkImageViewType) {
        case VK_IMAGE_VIEW_TYPE_CUBE:       flags |= cudaArrayCubemap;
↪           break;
        case VK_IMAGE_VIEW_TYPE_CUBE_ARRAY: flags |= cudaArrayCubemap |
↪cudaArrayLayered; break;
        case VK_IMAGE_VIEW_TYPE_1D_ARRAY:   flags |= cudaArrayLayered;
↪           break;
        case VK_IMAGE_VIEW_TYPE_2D_ARRAY:   flags |= cudaArrayLayered;
↪           break;
        default: break;
    }
    if (vkImageUsageFlags & VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT) {
        flags |= cudaArrayColorAttachment;
    }

    if (allowSurfaceLoadStore) {
        flags |= cudaArraySurfaceLoadStore;
    }

    return flags;
}
```

4.19.2.1.8 Importing Synchronization Objects

A Vulkan semaphore object exported using VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT can be imported into CUDA using the file descriptor associated with that object as shown below. Note that CUDA assumes ownership of the file descriptor once it is imported. Using the file descriptor after a successful import results in undefined behavior.

Whereas a Vulkan semaphore object exported using VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT can be imported into CUDA using the NT handle associated with that object as shown below. Note that CUDA does not assume ownership of the NT handle and it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying semaphore can be freed.

And, a Vulkan semaphore object exported using VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT can be imported into CUDA using the globally shared D3DKMT handle associated with that object as shown below. Since a globally shared D3DKMT handle does not hold a reference to the underlying semaphore it is automatically destroyed when all other references to the resource are destroyed.

```
  void importCudaExternalSemaphore(
      cudaExternalSemaphore_t &cudaSem, VkSemaphore &vkSem,
      VkExternalSemaphoreHandleTypeFlagBits handleType) {
    cudaExternalSemaphoreHandleDesc externalSemaphoreHandleDesc = {};

#ifdef _VK_TIMELINE_SEMAPHORE
    if (handleType & VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT) {
      externalSemaphoreHandleDesc.type =
          cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32;
    } else if (handleType &
               VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT) {
      externalSemaphoreHandleDesc.type =
          cudaExternalSemaphoreHandleTypeTimelineSemaphoreWin32;
    } else if (handleType & VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT) {
      externalSemaphoreHandleDesc.type =
          cudaExternalSemaphoreHandleTypeTimelineSemaphoreFd;
    }
#else
    if (handleType & VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_BIT) {
      externalSemaphoreHandleDesc.type =
          cudaExternalSemaphoreHandleTypeOpaqueWin32;
    } else if (handleType &
               VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_WIN32_KMT_BIT) {
      externalSemaphoreHandleDesc.type =
          cudaExternalSemaphoreHandleTypeOpaqueWin32Kmt;
    } else if (handleType & VK_EXTERNAL_SEMAPHORE_HANDLE_TYPE_OPAQUE_FD_BIT) {
      externalSemaphoreHandleDesc.type =
          cudaExternalSemaphoreHandleTypeOpaqueFd;
    }
#endif /* _VK_TIMELINE_SEMAPHORE */
    else {
      throw std::runtime_error("Unknown handle type requested!");
    }

#ifdef _WIN64
    externalSemaphoreHandleDesc.handle.win32.handle =
        (HANDLE)getSemaphoreHandle(vkSem, handleType);
#else
    externalSemaphoreHandleDesc.handle.fd =
        (int)(uintptr_t)getSemaphoreHandle(vkSem, handleType);
#endif

    externalSemaphoreHandleDesc.flags = 0;

    checkCudaErrors(
        cudaImportExternalSemaphore(&cudaSem, &externalSemaphoreHandleDesc));
  }
```

4.19.2.1.9 Signaling/Waiting on Imported Synchronization Objects

An imported Vulkan semaphore can be signaled and waited on as shown below. Signaling a semaphore sets it to the signaled state and in the case of timeline semaphores it sets the counter to the value specified in the signal call. The corresponding wait that waits on this signal must be issued in Vulkan. Additionally, in the case of a binary semaphore, the wait that waits on this signal must be issued after this signal has been issued.

Waiting on a semaphore waits until it reaches the signaled state or the assigned wait value. A signaled binary semaphore then resets it back to the unsignaled state. The corresponding signal that this wait is waiting on must be issued in Vulkan. Additionally, in the case of a binary semaphore, the signal must be issued before this wait can be issued.

In the following code extract from the *simpleVulkan* example the simulation step / the CUDA kernel is only called once the semaphore around the vertex buffers is signaled by Vulkan. After the simulation step another semaphore is signaled, or in the case of the timeline semaphore the same one is increased by CUDA, such that the Vulkan part that is waiting on this semaphore can continue rendering with the updated vertex buffers.

```
#ifdef _VK_TIMELINE_SEMAPHORE
    static uint64_t waitValue = 1;
    static uint64_t signalValue = 2;

    cudaExternalSemaphoreWaitParams waitParams = {};
    waitParams.flags = 0;
    waitParams.params.fence.value = waitValue;

    cudaExternalSemaphoreSignalParams signalParams = {};
    signalParams.flags = 0;
    signalParams.params.fence.value = signalValue;
    // Wait for vulkan to complete it's work
    checkCudaErrors(cudaWaitExternalSemaphoresAsync(&m_cudaTimelineSemaphore,
                                                    &waitParams, 1, m_
↪stream));
    // Now step the simulation, call CUDA kernel
    m_sim.stepSimulation(time, m_stream);
    // Signal vulkan to continue with the updated buffers
    checkCudaErrors(cudaSignalExternalSemaphoresAsync(
        &m_cudaTimelineSemaphore, &signalParams, 1, m_stream));

    waitValue += 2;
    signalValue += 2;
#else
    cudaExternalSemaphoreWaitParams waitParams = {};
    waitParams.flags = 0;
    waitParams.params.fence.value = 0;

    cudaExternalSemaphoreSignalParams signalParams = {};
    signalParams.flags = 0;
    signalParams.params.fence.value = 0;

    // Wait for vulkan to complete it's work
    checkCudaErrors(cudaWaitExternalSemaphoresAsync(&m_cudaWaitSemaphore,
                                                    &waitParams, 1, m_
```

(continues on next page)

(continued from previous page)

```
↪stream));
    // Now step the simulation, call CUDA kernel
    m_sim.stepSimulation(time, m_stream);
    // Signal vulkan to continue with the updated buffers
    checkCudaErrors(cudaSignalExternalSemaphoresAsync(
        &m_cudaSignalSemaphore, &signalParams, 1, m_stream));
#endif /* _VK_TIMELINE_SEMAPHORE */
```

4.19.2.1.10 OpenGL Interoperability

Traditional OpenGL-CUDA interop as outlined in *OpenGL Interoperability* works by CUDA directly consuming handles created in OpenGL. However, since OpenGL can also consume memory and synchronization objects created in Vulkan, there exists an alternative approach to doing OpenGL-CUDA interop. Essentially, memory and synchronization objects exported by Vulkan could be imported into both, OpenGL and CUDA, and then used to coordinate memory accesses between OpenGL and CUDA. Please refer to the following OpenGL extensions for further details on how to import memory and synchronization objects exported by Vulkan:

- `GL_EXT_memory_object`
- `GL_EXT_memory_object_fd`
- `GL_EXT_memory_object_win32`
- `GL_EXT_semaphore`
- `GL_EXT_semaphore_fd`
- `GL_EXT_semaphore_win32`

4.19.2.2 Direct3D Interoperability

Importing Direct3D[11|12] resources to CUDA is supported for Direct3D11 and Direct3D12. We are only looking at the Direct3D12, for Direct3D11 please refer to the CUDA programming guide 12.9.

4.19.2.2.1 Matching Device LUIDs

When importing memory and synchronization objects exported by Direct3D12, they must be imported and mapped on the same device as they were created on. The CUDA device that corresponds to the Direct3D12 device on which the objects were created can be determined by comparing the LUID of a CUDA device with that of the Direct3D12 device, as shown in the following code sample. Note that the Direct3D12 device must not be created on a linked node adapter, i.e. the node count as returned by `ID3D12Device::GetNodeCount` must be 1.

```
int getCudaDeviceForD3D12Device(ID3D12Device *d3d12Device) {
    LUID d3d12Luid = d3d12Device->GetAdapterLuid();

    int cudaDeviceCount;
    cudaGetDeviceCount(&cudaDeviceCount);

    for (int cudaDevice = 0; cudaDevice < cudaDeviceCount; cudaDevice++) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, cudaDevice);
        char *cudaLuid = deviceProp.luid;
```

(continues on next page)

(continued from previous page)

```
        if (!memcmp(&d3d12Luid.LowPart, cudaLuid, sizeof(d3d12Luid.LowPart)) &
↪&
            !memcmp(&d3d12Luid.HighPart, cudaLuid + sizeof(d3d12Luid.LowPart),
↪ sizeof(d3d12Luid.HighPart))) {
            return cudaDevice;
        }
    }
    return cudaInvalidDeviceId;
}
```

4.19.2.2.2 Importing Memory Objects

There are several different ways how to import memory objects from NT handles. Note that it is the application's responsibility to close the NT handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying memory can be freed. When importing a Direct3D resource, the flag `cudaExternalMemoryDedicated` must be set as in the snippets below.

A shareable Direct3D12 heap memory object, created by setting the flag `D3D12_HEAP_FLAG_SHARED` in the call to `ID3D12Device::CreateHeap`, can be imported into CUDA using the NT handle associated with that object as shown below.

```
cudaExternalMemory_t importD3D12HeapFromNTHandle(HANDLE handle, unsigned long
↪long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D12Heap;
    desc.handle.win32.handle = (void *)handle;
    desc.size = size;

    cudaImportExternalMemory(&extMem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extMem;
}
```

A shareable Direct3D12 heap memory object can also be imported using a named handle if one exists:

```
cudaExternalMemory_t importD3D12HeapFromNamedNTHandle(LPCWSTR name, unsigned
↪long long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D12Heap;
```

(continues on next page)

(continued from previous page)

```
    desc.handle.win32.name = (void *)name;
    desc.size = size;

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

A shareable Direct3D12 committed resource, created by setting the flag D3D12_HEAP_FLAG_SHARED in the call to D3D12Device::CreateCommittedResource, can be imported into CUDA using the NT handle associated with that object as shown below. When importing a Direct3D12 committed resource, the flag cudaExternalMemoryDedicated must be set.

```
cudaExternalMemory_t importD3D12CommittedResourceFromNTHandle(HANDLE handle,
↪unsigned long long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D12Resource;
    desc.handle.win32.handle = (void *)handle;
    desc.size = size;
    desc.flags |= cudaExternalMemoryDedicated;

    cudaImportExternalMemory(&extMem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extMem;
}
```

A shareable Direct3D12 committed resource can also be imported using a named handle if one exists as shown below.

```
cudaExternalMemory_t importD3D12CommittedResourceFromNamedNTHandle(LPCWSTR
↪name, unsigned long long size) {
    cudaExternalMemory_t extMem = NULL;
    cudaExternalMemoryHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalMemoryHandleTypeD3D12Resource;
    desc.handle.win32.name = (void *)name;
    desc.size = size;
    desc.flags |= cudaExternalMemoryDedicated;

    cudaImportExternalMemory(&extMem, &desc);

    return extMem;
}
```

4.19.2.2.3 Mapping Buffers onto Imported Memory Objects

A device pointer can be mapped onto an imported memory object as shown below. The offset and size of the mapping must match that specified when creating the mapping using the corresponding Direct3D12 API. All mapped device pointers must be freed using cudaFree().

```
void * mapBufferOntoExternalMemory(cudaExternalMemory_t extMem, unsigned long
↪long offset, unsigned long long size) {
    void *ptr = NULL;
    cudaExternalMemoryBufferDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.size = size;

    cudaExternalMemoryGetMappedBuffer(&ptr, extMem, &desc);

    // Note: 'ptr' must eventually be freed using cudaFree()
    return ptr;
}
```

4.19.2.2.4 Mapping Mipmapped Arrays onto Imported Memory Objects

A CUDA mipmapped array can be mapped onto an imported memory object as shown below. The offset, dimensions, format and number of mip levels must match that specified when creating the mapping using the corresponding Direct3D12 API. Additionally, if the mipmapped array can be bound as a render target in Direct3D12, the flag cudaArrayColorAttachment must be set. All mapped mipmapped arrays must be freed using cudaFreeMipmappedArray(). The following code sample shows how to convert parameters into the corresponding CUDA parameters when mapping mipmapped arrays onto imported memory objects.

```
cudaMipmappedArray_t mapMipmappedArrayOntoExternalMemory(cudaExternalMemory_t
↪extMem, unsigned long long offset, cudaChannelFormatDesc *formatDesc,
↪cudaExtent *extent, unsigned int flags, unsigned int numLevels) {
    cudaMipmappedArray_t mipmap = NULL;
    cudaExternalMemoryMipmappedArrayDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.formatDesc = *formatDesc;
    desc.extent = *extent;
    desc.flags = flags;
    desc.numLevels = numLevels;

    // Note: 'mipmap' must eventually be freed using cudaFreeMipmappedArray()
    cudaExternalMemoryGetMappedMipmappedArray(&mipmap, extMem, &desc);

    return mipmap;
}

cudaChannelFormatDesc getCudaChannelFormatDescForDxgiFormat(DXGI_FORMAT
```

(continues on next page)

(continued from previous page)

```
↪dxgiFormat)
{
    cudaChannelFormatDesc d;

    memset(&d, 0, sizeof(d));

    switch (dxgiFormat) {
        case DXGI_FORMAT_R8_UINT:              d.x = 8;  d.y = 0;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R8_SINT:              d.x = 8;  d.y = 0;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R8G8_UINT:            d.x = 8;  d.y = 8;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R8G8_SINT:            d.x = 8;  d.y = 8;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R8G8B8A8_UINT:        d.x = 8;  d.y = 8;  d.z = 8;  d.
↪w = 8;  d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R8G8B8A8_SINT:        d.x = 8;  d.y = 8;  d.z = 8;  d.
↪w = 8;  d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R16_UINT:             d.x = 16; d.y = 0;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R16_SINT:             d.x = 16; d.y = 0;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R16G16_UINT:          d.x = 16; d.y = 16; d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R16G16_SINT:          d.x = 16; d.y = 16; d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R16G16B16A16_UINT:    d.x = 16; d.y = 16; d.z = 16; d.
↪w = 16; d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R16G16B16A16_SINT:    d.x = 16; d.y = 16; d.z = 16; d.
↪w = 16; d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R32_UINT:             d.x = 32; d.y = 0;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R32_SINT:             d.x = 32; d.y = 0;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R32_FLOAT:            d.x = 32; d.y = 0;  d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindFloat;    break;
        case DXGI_FORMAT_R32G32_UINT:          d.x = 32; d.y = 32; d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R32G32_SINT:          d.x = 32; d.y = 32; d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R32G32_FLOAT:         d.x = 32; d.y = 32; d.z = 0;  d.
↪w = 0;  d.f = cudaChannelFormatKindFloat;    break;
        case DXGI_FORMAT_R32G32B32A32_UINT:    d.x = 32; d.y = 32; d.z = 32; d.
↪w = 32; d.f = cudaChannelFormatKindUnsigned; break;
        case DXGI_FORMAT_R32G32B32A32_SINT:    d.x = 32; d.y = 32; d.z = 32; d.
↪w = 32; d.f = cudaChannelFormatKindSigned;   break;
        case DXGI_FORMAT_R32G32B32A32_FLOAT: d.x = 32; d.y = 32; d.z = 32; d.
↪w = 32; d.f = cudaChannelFormatKindFloat;    break;
        default: assert(0);
    }
    return d;
```

(continues on next page)

(continued from previous page)

```
}

cudaExtent getCudaExtentForD3D12Extent(UINT64 width, UINT height, UINT16
↪depthOrArraySize, D3D12_SRV_DIMENSION d3d12SRVDimension) {
    cudaExtent e = { 0, 0, 0 };

    switch (d3d12SRVDimension) {
        case D3D12_SRV_DIMENSION_TEXTURE1D:         e.width = width; e.height
↪= 0;      e.depth = 0;                break;
        case D3D12_SRV_DIMENSION_TEXTURE2D:         e.width = width; e.height
↪= height; e.depth = 0;                break;
        case D3D12_SRV_DIMENSION_TEXTURE3D:         e.width = width; e.height
↪= height; e.depth = depthOrArraySize; break;
        case D3D12_SRV_DIMENSION_TEXTURECUBE:       e.width = width; e.height
↪= height; e.depth = depthOrArraySize; break;
        case D3D12_SRV_DIMENSION_TEXTURE1DARRAY:    e.width = width; e.height
↪= 0;      e.depth = depthOrArraySize; break;
        case D3D12_SRV_DIMENSION_TEXTURE2DARRAY:    e.width = width; e.height
↪= height; e.depth = depthOrArraySize; break;
        case D3D12_SRV_DIMENSION_TEXTURECUBEARRAY: e.width = width; e.height
↪= height; e.depth = depthOrArraySize; break;
        default: assert(0);
    }

    return e;
}

unsigned int getCudaMipmappedArrayFlagsForD3D12Resource(D3D12_SRV_DIMENSION
↪d3d12SRVDimension, D3D12_RESOURCE_FLAGS d3d12ResourceFlags, bool
↪allowSurfaceLoadStore) {
    unsigned int flags = 0;

    switch (d3d12SRVDimension) {
        case D3D12_SRV_DIMENSION_TEXTURECUBE:       flags |= cudaArrayCubemap;
↪                 break;
        case D3D12_SRV_DIMENSION_TEXTURECUBEARRAY: flags |= cudaArrayCubemap
↪| cudaArrayLayered; break;
        case D3D12_SRV_DIMENSION_TEXTURE1DARRAY:    flags |= cudaArrayLayered;
↪                 break;
        case D3D12_SRV_DIMENSION_TEXTURE2DARRAY:    flags |= cudaArrayLayered;
↪                 break;
        default: break;
    }

    if (d3d12ResourceFlags & D3D12_RESOURCE_FLAG_ALLOW_RENDER_TARGET) {
        flags |= cudaArrayColorAttachment;
    }
    if (allowSurfaceLoadStore) {
        flags |= cudaArraySurfaceLoadStore;
    }

    return flags;
```

(continues on next page)

(continued from previous page)

```
}
```

4.19.2.2.5 Importing Synchronization Objects

A shareable Direct3D12 fence object, created by setting the flag D3D12_FENCE_FLAG_SHARED in the call to ID3D12Device::CreateFence, can be imported into CUDA using the NT handle associated with that object as shown below. Note that it is the application's responsibility to close the handle when it is not required anymore. The NT handle holds a reference to the resource, so it must be explicitly freed before the underlying semaphore can be freed.

```
cudaExternalSemaphore_t importD3D12FenceFromNTHandle(HANDLE handle) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeD3D12Fence;
    desc.handle.win32.handle = handle;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Input parameter 'handle' should be closed if it's not needed anymore
    CloseHandle(handle);

    return extSem;
}
```

A shareable Direct3D12 fence object can also be imported using a named handle if one exists as shown below.

```
cudaExternalSemaphore_t importD3D12FenceFromNamedNTHandle(LPCWSTR name) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeD3D12Fence;
    desc.handle.win32.name = (void *)name;

    cudaImportExternalSemaphore(&extSem, &desc);

    return extSem;
}
```

4.19.2.2.6 Signaling/Waiting on Imported Synchronization Objects

Once the semaphores with fences have been imported from Direct3D12 they can be signaled and waited on.

Signaling a fence object sets its value. The corresponding wait that waits on this signal must be issued in Direct3D12. Note that the wait that waits on this signal must be issued after this signal has been issued.

```
void signalExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long
↪long value, cudaStream_t stream) {
    cudaExternalSemaphoreSignalParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.fence.value = value;

    cudaSignalExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

A fence object waits until its value becomes equal or greater than to the specified value. The corresponding signal that it is waiting on must be issued in Direct3D12. Note that, the signal must be issued before this wait can be issued.

```
void waitExternalSemaphore(cudaExternalSemaphore_t extSem, unsigned long long
↪value, cudaStream_t stream) {
    cudaExternalSemaphoreWaitParams params = {};

    memset(&params, 0, sizeof(params));

    params.params.fence.value = value;

    cudaWaitExternalSemaphoresAsync(&extSem, &params, 1, stream);
}
```

4.19.2.3 NVIDIA Software Communication Interface Interoperability (NVSCI)

NvSciBuf and NvSciSync are interfaces developed for serving the following purposes:

- NvSciBuf: Allows applications to allocate and exchange buffers in memory
- NvSciSync: Allows applications to manage synchronization objects at operation boundaries

More details on these interfaces are available at: https://docs.nvidia.com/drive.

4.19.2.3.1 Importing Memory Objects

For allocating an NvSciBuf object compatible with a given CUDA device, the corresponding GPU id must be set with `NvSciBufGeneralAttrKey_GpuId` in the NvSciBuf attribute list as shown below. Optionally, applications can specify the following attributes -

- `NvSciBufGeneralAttrKey_NeedCpuAccess`: Specifies if CPU access is required for the buffer
- `NvSciBufRawBufferAttrKey_Align`: Specifies the alignment requirement of `NvSciBufType_RawBuffer`
- `NvSciBufGeneralAttrKey_RequiredPerm`: Different access permissions can be configured for different UMDs per NvSciBuf memory object instance. For example, to provide the GPU with read-only access permissions to the buffer, create a duplicate NvSciBuf object using `NvSciBufObjDupWithReducePerm()` with `NvSciBufAccessPerm_Readonly` as the input parameter. Then import this newly created duplicate object with reduced permission into CUDA as shown
- `NvSciBufGeneralAttrKey_EnableGpuCache`: To control GPU L2 cacheability
- `NvSciBufGeneralAttrKey_EnableGpuCompression`: To specify GPU compression

Note

For more details on these attributes and their valid input options, refer to NvSciBuf Documentation.

The following code snippet illustrates their sample usage.

```
NvSciBufObj createNvSciBufObject() {
   // Raw Buffer Attributes for CUDA
    NvSciBufType bufType = NvSciBufType_RawBuffer;
    uint64_t rawsize = SIZE;
    uint64_t align = 0;
    bool cpuaccess_flag = true;
    NvSciBufAttrValAccessPerm perm = NvSciBufAccessPerm_ReadWrite;

    NvSciRmGpuId gpuid[] ={};
    CUuuid uuid;
    cuDeviceGetUuid(&uuid, dev));

    memcpy(&gpuid[0].bytes, &uuid.bytes, sizeof(uuid.bytes));
    // Disable cache on dev
    NvSciBufAttrValGpuCache gpuCache[] = {{gpuid[0], false}};
    NvSciBufAttrValGpuCompression gpuCompression[] = {{gpuid[0],
↪NvSciBufCompressionType_GenericCompressible}};
    // Fill in values
    NvSciBufAttrKeyValuePair rawbuffattrs[] = {
         { NvSciBufGeneralAttrKey_Types, &bufType, sizeof(bufType) },
         { NvSciBufRawBufferAttrKey_Size, &rawsize, sizeof(rawsize) },
         { NvSciBufRawBufferAttrKey_Align, &align, sizeof(align) },
         { NvSciBufGeneralAttrKey_NeedCpuAccess, &cpuaccess_flag,
↪sizeof(cpuaccess_flag) },
         { NvSciBufGeneralAttrKey_RequiredPerm, &perm, sizeof(perm) },
         { NvSciBufGeneralAttrKey_GpuId, &gpuid, sizeof(gpuid) },
         { NvSciBufGeneralAttrKey_EnableGpuCache &gpuCache, sizeof(gpuCache) }
↪,
         { NvSciBufGeneralAttrKey_EnableGpuCompression &gpuCompression,
↪sizeof(gpuCompression) }
    };

    // Create list by setting attributes
    err = NvSciBufAttrListSetAttrs(attrListBuffer, rawbuffattrs,
            sizeof(rawbuffattrs)/sizeof(NvSciBufAttrKeyValuePair));

    NvSciBufAttrListCreate(NvSciBufModule, &attrListBuffer);

    // Reconcile And Allocate
    NvSciBufAttrListReconcile(&attrListBuffer, 1, &attrListReconciledBuffer,
                       &attrListConflictBuffer)
    NvSciBufObjAlloc(attrListReconciledBuffer, &bufferObjRaw);
    return bufferObjRaw;
}
```

```
NvSciBufObj bufferObjRo; // Readonly NvSciBuf memory obj
```

(continues on next page)

(continued from previous page)

```
// Create a duplicate handle to the same memory buffer with reduced permissions
NvSciBufObjDupWithReducePerm(bufferObjRaw, NvSciBufAccessPerm_Readonly, &
↪bufferObjRo);
return bufferObjRo;
```

The allocated NvSciBuf memory object can be imported in CUDA using the NvSciBufObj handle as shown below. Application should query the allocated NvSciBufObj for attributes required for filling CUDA External Memory Descriptor. Note that the attribute list and NvSciBuf objects should be maintained by the application. If the NvSciBuf object imported into CUDA is also mapped by other drivers, then based on `NvSciBufGeneralAttrKey_GpuSwNeedCacheCoherency` output attribute value the application must use NvSciSync objects (refer to *Importing Synchronization Objects*) as appropriate barriers to maintain coherence between CUDA and the other drivers.

Note

For more details on how to allocate and maintain NvSciBuf objects refer to NvSciBuf API Documentation.

```
cudaExternalMemory_t importNvSciBufObject (NvSciBufObj bufferObjRaw) {

    /*************** Query NvSciBuf Object **************/
    NvSciBufAttrKeyValuePair bufattrs[] = {
                { NvSciBufRawBufferAttrKey_Size, NULL, 0 },
                { NvSciBufGeneralAttrKey_GpuSwNeedCacheCoherency, NULL, 0 },
                { NvSciBufGeneralAttrKey_EnableGpuCompression, NULL, 0 }
    };
    NvSciBufAttrListGetAttrs(retList, bufattrs,
        sizeof(bufattrs)/sizeof(NvSciBufAttrKeyValuePair)));
                ret_size = *(static_cast<const uint64_t*>(bufattrs[0].value));

    // Note cache and compression are per GPU attributes, so read values for
↪specific gpu by comparing UUID
    // Read cacheability granted by NvSciBuf
    int numGpus = bufattrs[1].len / sizeof(NvSciBufAttrValGpuCache);
    NvSciBufAttrValGpuCache[] cacheVal = (NvSciBufAttrValGpuCache
↪*)bufattrs[1].value;
    bool ret_cacheVal;
    for (int i = 0; i < numGpus; i++) {
        if (memcmp(gpuid[0].bytes, cacheVal[i].gpuId.bytes, sizeof(CUuuid))
↪== 0) {
            ret_cacheVal = cacheVal[i].cacheability);
        }
    }

    // Read compression granted by NvSciBuf
    numGpus = bufattrs[2].len / sizeof(NvSciBufAttrValGpuCompression);
    NvSciBufAttrValGpuCompression[] compVal = (NvSciBufAttrValGpuCompression
↪*)bufattrs[2].value;
    NvSciBufCompressionType ret_compVal;
    for (int i = 0; i < numGpus; i++) {
        if (memcmp(gpuid[0].bytes, compVal[i].gpuId.bytes, sizeof(CUuuid)) ==
```

(continues on next page)

(continued from previous page)

```
↪0) {
                ret_compVal = compVal[i].compressionType);
            }
        }

        /*************** NvSciBuf Registration With CUDA **************/

        // Fill up CUDA_EXTERNAL_MEMORY_HANDLE_DESC
        cudaExternalMemoryHandleDesc memHandleDesc;
        memset(&memHandleDesc, 0, sizeof(memHandleDesc));
        memHandleDesc.type = cudaExternalMemoryHandleTypeNvSciBuf;
        memHandleDesc.handle.nvSciBufObject = bufferObjRaw;
        // Set the NvSciBuf object with required access permissions in this step
        memHandleDesc.handle.nvSciBufObject = bufferObjRo;
        memHandleDesc.size = ret_size;
        cudaImportExternalMemory(&extMemBuffer, &memHandleDesc);
        return extMemBuffer;
    }
```

4.19.2.3.2 Mapping Buffers onto Imported Memory Objects

A device pointer can be mapped onto an imported memory object as shown below. The offset and size of the mapping can be filled as per the attributes of the allocated `NvSciBufObj`. All mapped device pointers must be freed using `cudaFree()`.

```
void * mapBufferOntoExternalMemory(cudaExternalMemory_t extMem, unsigned long
↪long offset, unsigned long long size) {
    void *ptr = NULL;
    cudaExternalMemoryBufferDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.size = size;

    cudaExternalMemoryGetMappedBuffer(&ptr, extMem, &desc);

    // Note: 'ptr' must eventually be freed using cudaFree()
    return ptr;
}
```

4.19.2.3.3 Mapping Mipmapped Arrays onto Imported Memory Objects

A CUDA mipmapped array can be mapped onto an imported memory object as shown below. The offset, dimensions and format can be filled as per the attributes of the allocated `NvSciBufObj`. All mapped mipmapped arrays must be freed using `cudaFreeMipmappedArray()`. The following code sample shows how to convert NvSciBuf attributes into the corresponding CUDA parameters when mapping mipmapped arrays onto imported memory objects.

Note

The number of mip levels must be 1.

```
cudaMipmappedArray_t mapMipmappedArrayOntoExternalMemory(cudaExternalMemory_t
↪extMem, unsigned long long offset, cudaChannelFormatDesc *formatDesc,
↪cudaExtent *extent, unsigned int flags, unsigned int numLevels) {
    cudaMipmappedArray_t mipmap = NULL;
    cudaExternalMemoryMipmappedArrayDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.offset = offset;
    desc.formatDesc = *formatDesc;
    desc.extent = *extent;
    desc.flags = flags;
    desc.numLevels = numLevels;

    // Note: 'mipmap' must eventually be freed using cudaFreeMipmappedArray()
    cudaExternalMemoryGetMappedMipmappedArray(&mipmap, extMem, &desc);

    return mipmap;
}
```

4.19.2.3.4 Importing Synchronization Objects

NvSciSync attributes that are compatible with a given CUDA device can be generated using `cudaDeviceGetNvSciSyncAttributes()`. The returned attribute list can be used to create a `NvSciSyncObj` that is guaranteed compatibility with a given CUDA device.

```
NvSciSyncObj createNvSciSyncObject() {
    NvSciSyncObj nvSciSyncObj
    int cudaDev0 = 0;
    int cudaDev1 = 1;
    NvSciSyncAttrList signalerAttrList = NULL;
    NvSciSyncAttrList waiterAttrList = NULL;
    NvSciSyncAttrList reconciledList = NULL;
    NvSciSyncAttrList newConflictList = NULL;

    NvSciSyncAttrListCreate(module, &signalerAttrList);
    NvSciSyncAttrListCreate(module, &waiterAttrList);
    NvSciSyncAttrList unreconciledList[2] = {NULL, NULL};
    unreconciledList[0] = signalerAttrList;
    unreconciledList[1] = waiterAttrList;

    cudaDeviceGetNvSciSyncAttributes(signalerAttrList, cudaDev0, CUDA_
↪NVSCISYNC_ATTR_SIGNAL);
    cudaDeviceGetNvSciSyncAttributes(waiterAttrList, cudaDev1, CUDA_NVSCISYNC_
↪ATTR_WAIT);

    NvSciSyncAttrListReconcile(unreconciledList, 2, &reconciledList, &
```

(continues on next page)

(continued from previous page)

```
↪newConflictList);

    NvSciSyncObjAlloc(reconciledList, &nvSciSyncObj);

    return nvSciSyncObj;
}
```

An NvSciSync object (created as above) can be imported into CUDA using the NvSciSyncObj handle as shown below. Note that ownership of the NvSciSyncObj handle continues to lie with the application even after it is imported.

```
cudaExternalSemaphore_t importNvSciSyncObject(void* nvSciSyncObj) {
    cudaExternalSemaphore_t extSem = NULL;
    cudaExternalSemaphoreHandleDesc desc = {};

    memset(&desc, 0, sizeof(desc));

    desc.type = cudaExternalSemaphoreHandleTypeNvSciSync;
    desc.handle.nvSciSyncObj = nvSciSyncObj;

    cudaImportExternalSemaphore(&extSem, &desc);

    // Deleting/Freeing the nvSciSyncObj beyond this point will lead to
↪undefined behavior in CUDA

    return extSem;
}
```

4.19.2.3.5 Signaling/Waiting on Imported Synchronization Objects

An imported `NvSciSyncObj` object can be signaled as outlined below. Signaling NvSciSync backed semaphore object initializes the *fence* parameter passed as input. This fence parameter is waited upon by a wait operation that corresponds to the aforementioned signal. Additionally, the wait that waits on this signal must be issued after this signal has been issued. If the flags are set to `cudaExternalSemaphoreSignalSkipNvSciBufMemSync` then memory synchronization operations (over all the imported NvSciBuf in this process) that are executed as a part of the signal operation by default are skipped. When `NvsciBufGeneralAttrKey_GpuSwNeedCacheCoherency` is FALSE, this flag should be set.

```
void signalExternalSemaphore(cudaExternalSemaphore_t extSem, cudaStream_t
↪stream, void *fence) {
    cudaExternalSemaphoreSignalParams signalParams = {};

    memset(&signalParams, 0, sizeof(signalParams));

    signalParams.params.nvSciSync.fence = (void*)fence;
    signalParams.flags = 0; //OR cudaExternalSemaphoreSignalSkipNvSciBufMemSync

    cudaSignalExternalSemaphoresAsync(&extSem, &signalParams, 1, stream);

}
```

An imported `NvSciSyncObj` object can be waited upon as outlined below. Waiting on NvSciSync backed semaphore object waits until the input *fence* parameter is signaled by the corresponding signaler. Additionally, the signal must be issued before the wait can be issued. If the flags are set to cudaExternalSemaphoreWaitSkipNvSciBufMemSync then memory synchronization operations (over all the imported NvSciBuf in this process) that are executed as a part of the signal operation by default are skipped. When NvsciBufGeneralAttrKey_GpuSwNeedCacheCoherency is FALSE, this flag should be set.

```
void waitExternalSemaphore(cudaExternalSemaphore_t extSem, cudaStream_t
↪stream, void *fence) {
     cudaExternalSemaphoreWaitParams waitParams = {};

    memset(&waitParams, 0, sizeof(waitParams));

    waitParams.params.nvSciSync.fence = (void*)fence;
    waitParams.flags = 0; //OR cudaExternalSemaphoreWaitSkipNvSciBufMemSync

    cudaWaitExternalSemaphoresAsync(&extSem, &waitParams, 1, stream);
}
```

4.20. Driver Entry Point Access

4.20.1. Introduction

The Driver Entry Point Access APIs provide a way to retrieve the address of a CUDA driver function. Starting from CUDA 11.3, users can call into available CUDA driver APIs using function pointers obtained from these APIs.

These APIs provide functionality similar to their counterparts, dlsym on POSIX platforms and GetProcAddress on Windows. The provided APIs will let users:

- Retrieve the address of a driver function using the CUDA Driver API.
- Retrieve the address of a driver function using the CUDA Runtime API.
- Request *per-thread default stream* version of a CUDA driver function. For more details, see *Retrieve Per-thread Default Stream Versions*.
- Access new CUDA features on older toolkits but with a newer driver.

4.20.2. Driver Function Typedefs

To help retrieve the CUDA Driver API entry points, the CUDA Toolkit provides access to headers containing the function pointer definitions for all CUDA driver APIs. These headers are installed with the CUDA Toolkit and are made available in the toolkit's include/ directory. The table below summarizes the header files containing the typedefs for each CUDA API header file.

Table 27: Typedefs header files for CUDA driver APIs

API header file	API Typedef header file
cuda.h	cudaTypedefs.h
cudaGL.h	cudaGLTypedefs.h
cudaProfiler.h	cudaProfilerTypedefs.h
cudaVDPAU.h	cudaVDPAUTypedefs.h
cudaEGL.h	cudaEGLTypedefs.h
cudaD3D9.h	cudaD3D9Typedefs.h
cudaD3D10.h	cudaD3D10Typedefs.h
cudaD3D11.h	cudaD3D11Typedefs.h

The above headers do not define actual function pointers themselves; they define the typedefs for function pointers. For example, `cudaTypedefs.h` has the below typedefs for the driver API `cuMemAlloc`:

```
typedef CUresult (CUDAAPI *PFN_cuMemAlloc_v3020)(CUdeviceptr_v2 *dptr, size_t
↪bytesize);
typedef CUresult (CUDAAPI *PFN_cuMemAlloc_v2000)(CUdeviceptr_v1 *dptr,
↪unsigned int bytesize);
```

CUDA driver symbols have a version based naming scheme with a `_v*` extension in its name except for the first version. When the signature or the semantics of a specific CUDA driver API changes, we increment the version number of the corresponding driver symbol. In the case of the `cuMemAlloc` driver API, the first driver symbol name is `cuMemAlloc` and the next symbol name is `cuMemAlloc_v2`. The typedef for the first version which was introduced in CUDA 2.0 (2000) is `PFN_cuMemAlloc_v2000`. The typedef for the next version which was introduced in CUDA 3.2 (3020) is `PFN_cuMemAlloc_v3020`.

The `typedefs` can be used to more easily define a function pointer of the appropriate type in code:

```
PFN_cuMemAlloc_v3020 pfn_cuMemAlloc_v2;
PFN_cuMemAlloc_v2000 pfn_cuMemAlloc_v1;
```

The above method is preferable if users are interested in a specific version of the API. Additionally, the headers have predefined macros for the latest version of all driver symbols that were available when the installed CUDA toolkit was released; these typedefs do not have a `_v*` suffix. For CUDA 11.3 toolkit, `cuMemAlloc_v2` was the latest version and so we can also define its function pointer as below:

```
PFN_cuMemAlloc pfn_cuMemAlloc;
```

4.20.3. Driver Function Retrieval

Using the Driver Entry Point Access APIs and the appropriate typedef, we can get the function pointer to any CUDA driver API.

4.20.3.1 Using the Driver API

The driver API requires CUDA version as an argument to get the ABI compatible version for the requested driver symbol. CUDA Driver APIs have a per-function ABI denoted with a `_v*` extension. For example, consider the versions of `cuStreamBeginCapture` and their corresponding `typedefs` from `cudaTypedefs.h`:

```
// cuda.h
CUresult CUDAAPI cuStreamBeginCapture(CUstream hStream);
CUresult CUDAAPI cuStreamBeginCapture_v2(CUstream hStream,
↪CUstreamCaptureMode mode);

// cudaTypedefs.h
typedef CUresult (CUDAAPI *PFN_cuStreamBeginCapture_v10000)(CUstream hStream);
typedef CUresult (CUDAAPI *PFN_cuStreamBeginCapture_v10010)(CUstream hStream,
↪CUstreamCaptureMode mode);
```

From the above `typedefs` in the code snippet, version suffixes `_v10000` and `_v10010` indicate that the above APIs were introduced in CUDA 10.0 and CUDA 10.1 respectively.

```
#include <cudaTypedefs.h>

// Declare the entry points for cuStreamBeginCapture
PFN_cuStreamBeginCapture_v10000 pfn_cuStreamBeginCapture_v1;
PFN_cuStreamBeginCapture_v10010 pfn_cuStreamBeginCapture_v2;

// Get the function pointer to the cuStreamBeginCapture driver symbol
cuGetProcAddress("cuStreamBeginCapture", &pfn_cuStreamBeginCapture_v1, 10000,
↪CU_GET_PROC_ADDRESS_DEFAULT, &driverStatus);
// Get the function pointer to the cuStreamBeginCapture_v2 driver symbol
cuGetProcAddress("cuStreamBeginCapture", &pfn_cuStreamBeginCapture_v2, 10010,
↪CU_GET_PROC_ADDRESS_DEFAULT, &driverStatus);
```

Referring to the code snippet above, to retrieve the address to the _v1 version of the driver API `cuStreamBeginCapture`, the CUDA version argument should be exactly 10.0 (10000). Similarly, the CUDA version for retrieving the address to the _v2 version of the API should be 10.1 (10010). Specifying a higher CUDA version for retrieving a specific version of a driver API might not always be portable. For example, using 11030 here would still return the _v2 symbol, but if a hypothetical _v3 version is released in CUDA 11.3, the `cuGetProcAddress` API would start returning the newer _v3 symbol instead when paired with a CUDA 11.3 driver. Since the ABI and function signatures of the _v2 and _v3 symbols might differ, calling the _v3 function using the `_v10010` typedef intended for the _v2 symbol would exhibit undefined behavior.

To retrieve the latest version of a driver API for a given CUDA Toolkit, we can also specify CUDA_VERSION as the `version` argument and use the unversioned typedef to define the function pointer. Since _v2 is the latest version of the driver API `cuStreamBeginCapture` in CUDA 11.3, the below code snippet shows a different method to retrieve it.

```
// Assuming we are using CUDA 11.3 Toolkit

#include <cudaTypedefs.h>

// Declare the entry point
PFN_cuStreamBeginCapture pfn_cuStreamBeginCapture_latest;
```

(continues on next page)

(continued from previous page)

```
// Initialize the entry point. Specifying CUDA_VERSION will give the function
↪pointer to the
// cuStreamBeginCapture_v2 symbol since it is latest version on CUDA 11.3.
cuGetProcAddress("cuStreamBeginCapture", &pfn_cuStreamBeginCapture_latest,
↪CUDA_VERSION, CU_GET_PROC_ADDRESS_DEFAULT, &driverStatus);
```

Note that requesting a driver API with an invalid CUDA version will return an error `CUDA_ERROR_NOT_FOUND`. In the above code examples, passing in a version less than 10000 (CUDA 10.0) would be invalid.

4.20.3.2 Using the Runtime API

The runtime API `cudaGetDriverEntryPoint` uses the CUDA runtime version to get the ABI compatible version for the requested driver symbol. In the below code snippet, the minimum CUDA runtime version required would be CUDA 11.2 as `cuMemAllocAsync` was introduced then.

```
#include <cudaTypedefs.h>

// Declare the entry point
PFN_cuMemAllocAsync pfn_cuMemAllocAsync;

// Initialize the entry point. Assuming CUDA runtime version >= 11.2
cudaGetDriverEntryPoint("cuMemAllocAsync", &pfn_cuMemAllocAsync,
↪cudaEnableDefault, &driverStatus);

// Call the entry point
if(driverStatus == cudaDriverEntryPointSuccess && pfn_cuMemAllocAsync) {
    pfn_cuMemAllocAsync(...);
}
```

The runtime API `cudaGetDriverEntryPointByVersion` uses the user provided CUDA version to get the ABI compatible version for the requested driver symbol. This allows more specific control over the requested ABI version.

4.20.3.3 Retrieve Per-thread Default Stream Versions

Some CUDA driver APIs can be configured to have *default stream* or *per-thread default stream* semantics. Driver APIs having *per-thread default stream* semantics are suffixed with *_ptsz* or *_ptds* in their name. For example, `cuLaunchKernel` has a *per-thread default stream* variant named `cuLaunchKernel_ptsz`. With the Driver Entry Point Access APIs, users can request for the *per-thread default stream* version of the driver API `cuLaunchKernel` instead of the *default stream* version. Configuring the CUDA driver APIs for *default stream* or *per-thread default stream* semantics affects the synchronization behavior. More details can be found here.

The *default stream* or *per-thread default stream* versions of a driver API can be obtained by one of the following ways:

- Use the compilation flag `--default-stream per-thread` or define the macro `CUDA_API_PER_THREAD_DEFAULT_STREAM` to get *per-thread default stream* behavior.
- Force *default stream* or *per-thread default stream* behavior using the flags `CU_GET_PROC_ADDRESS_LEGACY_STREAM/cudaEnableLegacyStream` or `CU_GET_PROC_ADDRESS_PER_THREAD_DEFAULT_STREAM/cudaEnablePerThreadDefaultStream` respectively.

4.20.3.4 Access New CUDA features

It is always recommended to install the latest CUDA toolkit to access new CUDA driver features, but if for some reason, a user does not want to update or does not have access to the latest toolkit, the API can be used to access new CUDA features with only an updated CUDA driver. For discussion, let us assume the user is on CUDA 11.3 and wants to use a new driver API cuFoo available in the CUDA 12.0 driver. The below code snippet illustrates this use-case:

```
int main()
{
    // Assuming we have CUDA 12.0 driver installed.

    // Manually define the prototype as cudaTypedefs.h in CUDA 11.3 does not
↪have the cuFoo typedef
    typedef CUresult (CUDAAPI *PFN_cuFoo)(...);
    PFN_cuFoo pfn_cuFoo = NULL;
    CUdriverProcAddressQueryResult driverStatus;

    // Get the address for cuFoo API using cuGetProcAddress. Specify CUDA
↪version as
    // 12000 since cuFoo was introduced then or get the driver version
↪dynamically
    // using cuDriverGetVersion
    int driverVersion;
    cuDriverGetVersion(&driverVersion);
    CUresult status = cuGetProcAddress("cuFoo", &pfn_cuFoo, driverVersion, CU_
↪GET_PROC_ADDRESS_DEFAULT, &driverStatus);

    if (status == CUDA_SUCCESS && pfn_cuFoo) {
        pfn_cuFoo(...);
    }
    else {
        printf("Cannot retrieve the address to cuFoo - driverStatus = %d.
↪Check if the latest driver for CUDA 12.0 is installed.\n", driverStatus);
        assert(0);
    }

    // rest of code here

}
```

4.20.4. Potential Implications with cuGetProcAddress

Below is a set of concrete and theoretical examples of potential issues with `cuGetProcAddress` and `cudaGetDriverEntryPoint`.

4.20.4.1 Implications with cuGetProcAddress vs Implicit Linking

`cuDeviceGetUuid` was introduced in CUDA 9.2. This API has a newer revision (`cuDeviceGetUuid_v2`) introduced in CUDA 11.4. To preserve minor version compatibility, `cuDeviceGetUuid` will not be version bumped to `cuDeviceGetUuid_v2` in cuda.h until CUDA 12.0. This means that calling it by obtaining a function pointer to it via `cuGetProcAddress` might have different behavior. Example using the API directly:

```
#include <cuda.h>

CUuuid uuid;
CUdevice dev;
CUresult status;

status = cuDeviceGet(&dev, 0); // Get device 0
// handle status

status = cuDeviceGetUuid(&uuid, dev) // Get uuid of device 0
```

In this example, assume the user is compiling with CUDA 11.4. Note that this will perform the behavior of `cuDeviceGetUuid`, not _v2 version. Now an example of using `cuGetProcAddress`:

```
#include <cudaTypedefs.h>

CUuuid uuid;
CUdevice dev;
CUresult status;
CUdriverProcAddressQueryResult driverStatus;

status = cuDeviceGet(&dev, 0); // Get device 0
// handle status

PFN_cuDeviceGetUuid pfn_cuDeviceGetUuid;
status = cuGetProcAddress("cuDeviceGetUuid", &pfn_cuDeviceGetUuid, CUDA_
↪VERSION, CU_GET_PROC_ADDRESS_DEFAULT, &driverStatus);
if(CUDA_SUCCESS == status && pfn_cuDeviceGetUuid) {
    // pfn_cuDeviceGetUuid points to ???
}
```

In this example, assume the user is compiling with CUDA 11.4. This will get the function pointer of `cuDeviceGetUuid_v2`. Calling the function pointer will then invoke the new _v2 function, not the same `cuDeviceGetUuid` as shown in the previous example.

4.20.4.2 Compile Time vs Runtime Version Usage in cuGetProcAddress

Let's take the same issue and make one small tweak. The last example used the compile time constant of CUDA_VERSION to determine which function pointer to obtain. More complications arise if the user queries the driver version dynamically using `cuDriverGetVersion` or `cudaDriverGetVersion` to pass to `cuGetProcAddress`. Example:

```
#include <cudaTypedefs.h>

CUuuid uuid;
CUdevice dev;
CUresult status;
int cudaVersion;
CUdriverProcAddressQueryResult driverStatus;

status = cuDeviceGet(&dev, 0); // Get device 0
// handle status
```

(continues on next page)

(continued from previous page)

```
status = cuDriverGetVersion(&cudaVersion);
// handle status

PFN_cuDeviceGetUuid pfn_cuDeviceGetUuid;
status = cuGetProcAddress("cuDeviceGetUuid", &pfn_cuDeviceGetUuid,
↪cudaVersion, CU_GET_PROC_ADDRESS_DEFAULT, &driverStatus);
if(CUDA_SUCCESS == status && pfn_cuDeviceGetUuid) {
    // pfn_cuDeviceGetUuid points to ???
}
```

In this example, assume the user is compiling with CUDA 11.3. The user would debug, test, and deploy this application with the known behavior of getting `cuDeviceGetUuid` (not the _v2 version). Since CUDA has guaranteed ABI compatibility between minor versions, this same application is expected to run after the driver is upgraded to CUDA 11.4 (without updating the toolkit and runtime) without requiring recompilation. This will have undefined behavior though, because now the typedef for `PFN_cuDeviceGetUuid` will still be of the signature for the original version, but since `cudaVersion` would now be 11040 (CUDA 11.4), `cuGetProcAddress` would return the function pointer to the _v2 version, meaning calling it might have undefined behavior.

Note in this case the original (not the _v2 version) typedef looks like:

```
typedef CUresult (CUDAAPI *PFN_cuDeviceGetUuid_v9020)(CUuuid *uuid, CUdevice_
↪v1 dev);
```

But the _v2 version typedef looks like:

```
typedef CUresult (CUDAAPI *PFN_cuDeviceGetUuid_v11040)(CUuuid *uuid, CUdevice_
↪v1 dev);
```

So in this case, the API/ABI is going to be the same and the runtime API call will likely not cause issues– only the potential for unknown uuid return. In *Implications to API/ABI*, we discuss a more problematic case of API/ABI compatibility.

4.20.4.3 API Version Bumps with Explicit Version Checks

Above, was a specific concrete example. Now for instance let's use a theoretical example that still has issues with compatibility across driver versions. Example:

```
CUresult cuFoo(int bar); // Introduced in CUDA 11.4
CUresult cuFoo_v2(int bar); // Introduced in CUDA 11.5
CUresult cuFoo_v3(int bar, void* jazz); // Introduced in CUDA 11.6

typedef CUresult (CUDAAPI *PFN_cuFoo_v11040)(int bar);
typedef CUresult (CUDAAPI *PFN_cuFoo_v11050)(int bar);
typedef CUresult (CUDAAPI *PFN_cuFoo_v11060)(int bar, void* jazz);
```

Notice that the API has been modified twice since original creation in CUDA 11.4 and the latest in CUDA 11.6 also modified the API/ABI interface to the function. The usage in user code compiled against CUDA 11.5 is:

```
#include <cuda.h>
#include <cudaTypedefs.h>

```

(continues on next page)

(continued from previous page)

```
CUresult status;
int cudaVersion;
CUdriverProcAddressQueryResult driverStatus;

status = cuDriverGetVersion(&cudaVersion);
// handle status

PFN_cuFoo_v11040 pfn_cuFoo_v11040;
PFN_cuFoo_v11050 pfn_cuFoo_v11050;
if(cudaVersion < 11050 ) {
    // We know to get the CUDA 11.4 version
    status = cuGetProcAddress("cuFoo", &pfn_cuFoo_v11040, cudaVersion, CU_GET_
↪PROC_ADDRESS_DEFAULT, &driverStatus);
    // Handle status and validating pfn_cuFoo_v11040
}
else {
    // Assume >= CUDA 11.5 version we can use the second version
    status = cuGetProcAddress("cuFoo", &pfn_cuFoo_v11050, cudaVersion, CU_GET_
↪PROC_ADDRESS_DEFAULT, &driverStatus);
    // Handle status and validating pfn_cuFoo_v11050
}
```

In this example, without updates for the new typedef in CUDA 11.6 and recompiling the application with those new typedefs and case handling, the application will get the cuFoo_v3 function pointer returned and any usage of that function would then cause undefined behavior. The point of this example was to illustrate that even explicit version checks for `cuGetProcAddress` may not safely cover the minor version bumps within a CUDA major release.

4.20.4.4 Issues with Runtime API Usage

The above examples were focused on the issues with the Driver API usage for obtaining the function pointers to driver APIs. Now we will discuss the potential issues with the Runtime API usage for `cudaApiGetDriverEntryPoint`.

We will start by using the Runtime APIs similar to the above.

```
#include <cuda.h>
#include <cudaTypedefs.h>
#include <cuda_runtime.h>

CUresult status;
cudaError_t error;
int driverVersion, runtimeVersion;
CUdriverProcAddressQueryResult driverStatus;

// Ask the runtime for the function
PFN_cuDeviceGetUuid pfn_cuDeviceGetUuidRuntime;
error = cudaGetDriverEntryPoint ("cuDeviceGetUuid", &pfn_
↪cuDeviceGetUuidRuntime, cudaEnableDefault, &driverStatus);
if(cudaSuccess == error && pfn_cuDeviceGetUuidRuntime) {
    // pfn_cuDeviceGetUuid points to ???
}
```

The function pointer in this example is even more complicated than the driver only examples above because there is no control over which version of the function to obtain; it will always get the API for the current CUDA Runtime version. See the following table for more information:

	Static Runtime Version Linkage	
Driver Version Installed	**V11.3**	**V11.4**
V11.3	v1	v1x
V11.4	v1	v2

```
V11.3 => 11.3 CUDA Runtime and Toolkit (includes header files cuda.h and
↪cudaTypedefs.h)
V11.4 => 11.4 CUDA Runtime and Toolkit (includes header files cuda.h and
↪cudaTypedefs.h)
v1 => cuDeviceGetUuid
v2 => cuDeviceGetUuid_v2

x => Implies the typedef function pointer won't match the returned
     function pointer.  In these cases, the typedef at compile time
     using a CUDA 11.4 runtime, would match the _v2 version, but the
     returned function pointer would be the original (non _v2) function.
```

The problem in the table comes in with a newer CUDA 11.4 Runtime and Toolkit and older driver (CUDA 11.3) combination, labeled as v1x in the above. This combination would have the driver returning the pointer to the older function (non _v2), but the typedef used in the application would be for the new function pointer.

4.20.4.5 Issues with Runtime API and Dynamic Versioning

More complications arise when we consider different combinations of the CUDA version with which an application is compiled, CUDA runtime version, and CUDA driver version that an application dynamically links against.

```
#include <cuda.h>
#include <cudaTypedefs.h>
#include <cuda_runtime.h>

CUresult status;
cudaError_t error;
int driverVersion, runtimeVersion;
CUdriverProcAddressQueryResult driverStatus;
enum cudaDriverEntryPointQueryResult runtimeStatus;

PFN_cuDeviceGetUuid pfn_cuDeviceGetUuidDriver;
status = cuGetProcAddress("cuDeviceGetUuid", &pfn_cuDeviceGetUuidDriver, CUDA_
↪VERSION, CU_GET_PROC_ADDRESS_DEFAULT, &driverStatus);
if(CUDA_SUCCESS == status && pfn_cuDeviceGetUuidDriver) {
    // pfn_cuDeviceGetUuidDriver points to ???
}

// Ask the runtime for the function
PFN_cuDeviceGetUuid pfn_cuDeviceGetUuidRuntime;
```

(continues on next page)

(continued from previous page)

```
error = cudaGetDriverEntryPoint ("cuDeviceGetUuid", &pfn_
↪cuDeviceGetUuidRuntime, cudaEnableDefault, &runtimeStatus);
if(cudaSuccess == error && pfn_cuDeviceGetUuidRuntime) {
    // pfn_cuDeviceGetUuidRuntime points to ???
}

// Ask the driver for the function based on the driver version (obtained via
↪runtime)
error = cudaDriverGetVersion(&driverVersion);
PFN_cuDeviceGetUuid pfn_cuDeviceGetUuidDriverDriverVer;
status = cuGetProcAddress ("cuDeviceGetUuid", &pfn_
↪cuDeviceGetUuidDriverDriverVer, driverVersion, CU_GET_PROC_ADDRESS_DEFAULT,
↪&driverStatus);
if(CUDA_SUCCESS == status && pfn_cuDeviceGetUuidDriverDriverVer) {
    // pfn_cuDeviceGetUuidDriverDriverVer points to ???
}
```

The following matrix of function pointers is expected:

Function Pointer	**Application Compiled/Runtime Dynamic Linked Version/Driver Version**							
	(3 => CUDA 11.3 and 4 => CUDA 11.4)							
	3/3/3	**3/3/4**	**3/4/3**	**3/4/4**	**4/3/3**	**4/3/4**	**4/4/3**	**4/4/4**
pfn_cuDeviceGetUuidDriver	t1/v1	t1/v1	t1/v1	t1/v1	N/A	N/A	**t2/v1**	t2/v2
pfn_cuDeviceGetUuidRuntim	t1/v1	t1/v1	t1/v1	**t1/v2**	N/A	N/A	**t2/v1**	t2/v2
pfn_cuDeviceGetUuidDriver	t1/v1	**t1/v2**	t1/v1	**t1/v2**	N/A	N/A	**t2/v1**	t2/v2

```
tX -> Typedef version used at compile time
vX -> Version returned/used at runtime
```

If the application is compiled against CUDA Version 11.3, it would have the typedef for the original function, but if compiled against CUDA Version 11.4, it would have the typedef for the _v2 function. Because of that, notice the number of cases where the typedef does not match the actual version returned/used.

4.20.4.6 Issues with Runtime API allowing CUDA Version

Unless specified otherwise, the CUDA runtime API `cudaGetDriverEntryPointByVersion` will have similar implications as the driver entry point `cuGetProcAddress` since it allows for the user to request a specific CUDA driver version.

4.20.4.7 Implications to API/ABI

In the above examples using `cuDeviceGetUuid`, the implications of the mismatched API are minimal, and may not be entirely noticeable to many users as the _v2 was added to support Multi-Instance GPU (MIG) mode. So, on a system without MIG, the user might not even realize they are getting a different API.

More problematic is an API which changes its application signature (and hence ABI) such as cuCtx-Create. The _v2 version, introduced in CUDA 3.2 is currently used as the default cuCtxCreate when using cuda.h but now has a newer version introduced in CUDA 11.4 (cuCtxCreate_v3). The API signature has been modified as well, and now takes extra arguments. So, in some of the cases above, where the typedef to the function pointer doesn't match the returned function pointer, there is a chance for non-obvious ABI incompatibility which would lead to undefined behavior.

For example, assume the following code compiled against a CUDA 11.3 toolkit with a CUDA 11.4 driver installed:

```
PFN_cuCtxCreate cuUnknown;
CUdriverProcAddressQueryResult driverStatus;

status = cuGetProcAddress("cuCtxCreate", (void**)&cuUnknown, cudaVersion, CU_
→GET_PROC_ADDRESS_DEFAULT, &driverStatus);
if(CUDA_SUCCESS == status && cuUnknown) {
    status = cuUnknown(&ctx, 0, dev);
}
```

Running this code where cudaVersion is set to anything >=11040 (indicating CUDA 11.4) could have undefined behavior due to not having adequately supplied all the parameters required for the _v3 version of the cuCtxCreate_v3 API.

4.20.5. Determining cuGetProcAddress Failure Reasons

There are two types of errors with cuGetProcAddress. Those are (1) API/usage errors and (2) inability to find the driver API requested. The first error type will return error codes from the API via the CUresult return value. Things like passing NULL as the pfn variable or passing invalid flags.

The second error type encodes in the CUdriverProcAddressQueryResult *symbolStatus and can be used to help distinguish potential issues with the driver not being able to find the symbol requested. Take the following example:

```
// cuDeviceGetExecAffinitySupport was introduced in release CUDA 11.4
#include <cuda.h>
CUdriverProcAddressQueryResult driverStatus;
cudaVersion = ...;
status = cuGetProcAddress("cuDeviceGetExecAffinitySupport", &pfn, cudaVersion,
→ 0, &driverStatus);
if (CUDA_SUCCESS == status) {
    if (CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT == driverStatus) {
        printf("We can use the new feature when you upgrade cudaVersion to 11.
→4, but CUDA driver is good to go!\n");
        // Indicating cudaVersion was < 11.4 but run against a CUDA driver >=
→11.4
    }
    else if (CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND == driverStatus) {
        printf("Please update both CUDA driver and cudaVersion to at least 11.
→4 to use the new feature!\n");
        // Indicating driver is < 11.4 since string not found, doesn't matter
→what cudaVersion was
    }
    else if (CU_GET_PROC_ADDRESS_SUCCESS == driverStatus && pfn) {
        printf("You're using cudaVersion and CUDA driver >= 11.4, using new
```

(continues on next page)

(continued from previous page)

```
↪feature!\n");
        pfn();
    }
}
```

The first case with the return code CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT indicates that the symbol was found when searching in the CUDA driver but it was added later than the cudaVersion supplied. In the example, specifying cudaVersion as anything 11030 or less and when running against a CUDA driver >= CUDA 11.4 would give this result of CU_GET_PROC_ADDRESS_VERSION_NOT_SUFFICIENT. This is because cuDeviceGetExecAffinitySupport was added in CUDA 11.4 (11040).

The second case with the return code CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND indicates that the symbol was not found when searching in the CUDA driver. This can be due to a few reasons such as unsupported CUDA function due to older driver as well as just having a typo. In the latter, similar to the last example if the user had put symbol as CUDeviceGetExecAffinitySupport - notice the capital CU to start the string - cuGetProcAddress would not be able to find the API because the string doesn't match. In the former case an example might be the user developing an application against a CUDA driver supporting the new API, and deploying the application against an older CUDA driver. Using the last example, if the developer developed against CUDA 11.4 or later but was deployed against a CUDA 11.3 driver, during their development they may have had a successful cuGetProcAddress, but when deploying an application running against a CUDA 11.3 driver the call would no longer work with the CU_GET_PROC_ADDRESS_SYMBOL_NOT_FOUND returned in driverStatus.

Chapter 5. Technical Appendices

5.1. Compute Capabilities

The general specifications and features of a compute device depend on its compute capability (see *Compute Capability and Streaming Multiprocessor Versions*).

Table 29, Table 30, and Table 31 show the features and technical specifications associated with each compute capability that is currently supported.

All NVIDIA GPU architectures use a little-endian representation.

5.1.1. Obtain the GPU Compute Capability

The CUDA GPU Compute Capability page provides a comprehensive mapping from NVIDIA GPU models to their compute capability.

Alternatively, the nvidia-smi tool, provided with the NVIDIA Driver, can be used to get the compute capability of a GPU. For example, the following command will output the GPU names and compute capabilities available on the system:

```
nvidia-smi --query-gpu=name,compute_cap
```

At runtime, the compute capability can be obtained using the CUDA Runtime API cudaDeviceGetAttribute() , CUDA Driver API cuDeviceGetAttribute(), or NVML API nvmlDeviceGetCudaComputeCapability():

```
#include <cuda_runtime_api.h>

int computeCapabilityMajor, computeCapabilityMinor;
cudaDeviceGetAttribute(&computeCapabilityMajor,
↪cudaDevAttrComputeCapabilityMajor, device_id);
cudaDeviceGetAttribute(&computeCapabilityMinor,
↪cudaDevAttrComputeCapabilityMinor, device_id);
```

```
#include <cuda.h>

int computeCapabilityMajor, computeCapabilityMinor;
cuDeviceGetAttribute(&computeCapabilityMajor, CU_DEVICE_ATTRIBUTE_COMPUTE_
↪CAPABILITY_MAJOR, device_id);
cuDeviceGetAttribute(&computeCapabilityMinor, CU_DEVICE_ATTRIBUTE_COMPUTE_
↪CAPABILITY_MINOR, device_id);
```

```
#include <nvml.h> // required linking with -lnvidia-ml

int computeCapabilityMajor, computeCapabilityMinor;
nvmlDeviceGetCudaComputeCapability(nvmlDevice, &computeCapabilityMajor, &
↪computeCapabilityMinor);
```

5.1.2. Feature Availability

Most compute features introduced with a compute architecture are intended to be available on all subsequent architectures. This is shown in Table 29 by the “yes” for availability of a feature on compute capabilities subsequent to its introduction.

5.1.2.1 Architecture-Specific Features

Beginning with devices of Compute Capability 9.0, specialized compute features that are introduced with an architecture may not be guaranteed to be available on all subsequent compute capabilities. These features are called *architecture-specific* features and target acceleration of specialized operations, such as Tensor Core operations, which are not intended for all classes of compute capabilities or may significantly change in future generations. Code must be compiled with an architecture-specific compiler target (see *Feature Set Compiler Targets*) to enable architecture-specific features. Code compiled with an architecture-specific compiler target can only be run on the exact compute capability it was compiled for.

5.1.2.2 Family-Specific Features

Beginning with devices of Compute Capability 10.0, some architecture-specific features are common to devices of more than one compute capability. The devices that contain these features are part of the same family and these features can also be called *family-specific* features. Family-specific features are guaranteed to be available on all devices in the same family. A family-specific compiler target is required to enable family-specific features. See Section 5.1.2.3. Code compiled for a family-specific target can only be run on GPUs which are members of that family.

5.1.2.3 Feature Set Compiler Targets

There are three sets of compute features which the compiler can target:

Baseline Feature Set: The predominant set of compute features that are introduced with the intent to be available for subsequent compute architectures. These features and their availability are summarized in Table 29.

Architecture-Specific Feature Set: A small and highly specialized set of features called architecture-specific, that are introduced to accelerate specialized operations, which are not guaranteed to be available or might change significantly on subsequent compute architectures. These features are summarized in the respective “Compute Capability #.#” subsections. The architecture-specific feature set is a superset of the family-specific feature set. Architecture-specific compiler targets were introduced with Compute Capability 9.0 devices and are selected by using an **a** suffix in the compilation target, for example by specifying `compute_100a` or `compute_120a` as the compute target.

Family-Specific Feature Set: Some architecture-specific features are common to GPUs of more than one compute capability. These features are summarized in the respective “Compute Capability #.#” subsections. With a few exceptions, later-generation devices with the same major compute capability are in the same family. Table 28 indicates the compatibility of family-specific targets with device compute capability, including exceptions. The family-specific feature set is a superset of the baseline feature set. Family-specific compiler targets were introduced with Compute Capability 10.0 devices and are selected by using an **f** suffix in the compilation target, for example by specifying `compute_100f` or `compute_120f` as the compute target.

All devices starting from compute capability 9.0 have a set of features that are architecture-specific. To utilize the complete set of these features on a specific GPU, the architecture-specific compiler target with the suffix **a** must be used. Additionally, starting from compute capability 10.0, there are sets of features that appear in multiple devices with different minor compute capabilities. These sets of instructions are called family-specific features, and the devices which share these features are said to be part of the same family. The family-specific features are a subset of the architecture-specific features that are shared by all members of that GPU family. The family-specific compiler target with the suffix **f** allows the compiler to generate code that uses this common subset of architecture-specific features.

For example:

- The `compute_100` compilation target does not allow the use of architecture-specific features. This target will be compatible with all devices of compute capability 10.0 and later.
- The `compute_100f` *family-specific* compilation target allows the use of the subset of architecture-specific features that are common across the GPU family. This target will only be compatible with devices that are part of the GPU family. In this example, it is compatible with devices of Compute Capability 10.0 and Compute Capability 10.3. The features available in the family-specific `compute_100f` target are a superset of the features available in the baseline `compute_100` target.
- The `compute_100a` *architecture-specific* compilation target allows the use of the complete set of architecture-specific features in Compute Capability 10.0 devices. This target will only be compatible with devices of Compute Capability 10.0 and no others. The features available in the `compute_100a` target form a superset of the features available in the `compute_100f` target.

Table 28: Family-Specific Compatibility

Compilation Target	Compatible with Compute Capability	
`compute_100f`	10.0	10.3
`compute_103f`	10.3[1]	
`compute_110f`	11.0[1]	
`compute_120f`	12.0	12.1
`compute_121f`	12.1[1]	

5.1.3. Features and Technical Specifications

[1] Some families only contain a single member when they are created. They may be expanded in the future to include more devices.

Table 29: Feature Support per Compute Capability

Feature Support	Compute Capability					
(Unlisted features are supported for all compute capabilities)	7.x	8.x	9.0	10.x	11.0	12.x
Atomic functions operating on 128-bit integer values in shared and global memory (*Atomic Functions*)	No		Yes			
Atomic addition operating on `float2` and `float4` floating point vectors in global memory (*atomicAdd()*)	No		Yes			
Warp reduce functions (*Warp Reduce Functions*)	No	Yes				
Bfloat16-precision floating-point operations	No	Yes				

continues on next page

Table 29 – continued from previous page

Feature Support	Compute Capability				
128-bit-precision floating-point operations	No			Yes	
Hardware-accelerated `memcpy_async` (*Pipelines*)	No	Yes			
Hardware-accelerated Split Arrive/Wait Barrier (*Asynchronous Barriers*)	No	Yes			
L2 Cache Residency Management (*L2 Cache Control*)	No	Yes			
DPX Instructions for Accelerated Dynamic Programming (*Dynamic Programming eXtension (DPX) Instructions*)	Multiple Instr.		Native		Multiple Instr.
Distributed Shared Memory	No		Yes		

continues on next page

Table 29 – continued from previous page

Feature Support	Compute Capability	
Thread Block Cluster (*Thread Block Clusters*)	No	Yes
Tensor Memory Accelerator (TMA) unit (*Using the Tensor Memory Accelerator (TMA)*)	No	Yes

Note that the KB and K units used in the following tables correspond to 1024 bytes (i.e., a KiB) and 1024 respectively.

Table 30: Device and Streaming Multiprocessor (SM) Information per Compute Capability

	Compute Capability									
	7.5	8.0	8.6	8.7	8.9	9.0	10.0	10.3	11.0	12.x
Ratio of FP32 to FP64 Throughput[2]	32:1	2:1	64:1			2:1		64:1		
Maximum number of resident grids per device (Concurrent Kernel Execution)	128									
Maximum dimensionality of a grid	3									
Maximum x-dimension of a grid	2^{31}-1									
Maximum y- or z-dimension of a grid	65535									
Maximum dimensionality of a thread block	3									
Maximum x- or y-dimensionality of a thread block	1024									
Maximum z-dimension of a thread block	64									
Maximum number of threads per block	1024									
Warp size	32									
Maximum number of resident blocks per SM	16	32	16		24	32			24	

continues on next page

Table 30 – continued from previous page

	Compute Capability				
Maximum number of resident warps per SM	32	64	48	64	48
Maximum number of resident threads per SM	1024	2048	1536	2048	1536
Green contexts: minimum SM partition size for useFlags 0	2	4		8	
Green contexts: SM co-scheduled alignment per partition for useFlags 0	2			8	

Table 31: Memory Information per Compute Capability

	Compute Capability								
	7.5	8.0	8.6	8.7	8.9	9.0	10.x	11.0	12.x
Number of 32-bit registers per SM	64 K								
Maximum number of 32-bit registers per thread block	64 K								
Maximum number of 32-bit registers per thread	255								
Maximum amount of shared memory per SM	64 KB	164 KB	100 KB	164 KB	100 KB	228 KB			100 KB
Maximum amount of shared memory per thread block[3]	64 KB	163 KB	99 KB	163 KB	99 KB	227 KB			99 KB
Number of shared memory banks	32								
Maximum amount of local memory per thread	512 KB								
Constant memory size	64 KB								
Cache working set per SM for constant memory	8 KB								
Cache working set per SM for texture memory	32 or 64 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 192 KB	28 KB ~ 128 KB	28 KB ~ 256 KB			28 KB ~ 128 KB

[2] Non-Tensor Core throughputs. For more information on throughput see the CUDA Best Practices Guide

[3] Kernels relying on shared memory allocations over 48 KB per block must use dynamic shared memory and require an explicit opt-in, see *Configuring L1/Shared Memory Balance*.

Table 32: Shared Memory Capacity per Compute Capability

Compute Capability	Unified Data Cache Size (KB)	SMEM Capacity Sizes (KB)
7.5	96	32, 64
8.0	192	0, 8, 16, 32, 64, 100, 132, 164
8.6	128	0, 8, 16, 32, 64, 100
8.7	192	0, 8, 16, 32, 64, 100, 132, 164
8.9	128	0, 8, 16, 32, 64, 100
9.0	256	0, 8, 16, 32, 64, 100, 132, 164, 196, 228
10.x	256	0, 8, 16, 32, 64, 100, 132, 164, 196, 228
11.0	256	0, 8, 16, 32, 64, 100, 132, 164, 196, 228
12.x	128	0, 8, 16, 32, 64, 100

Table 33 shows the input data types supported by Tensor Core acceleration. The Tensor Core feature set is available within the CUDA compilation toolchain through inline PTX. It is strongly recommended that applications use this feature set through CUDA-X libraries such as cuDNN, cuBLAS, and cuFFT, for example, or through CUTLASS, a collection of CUDA C++ template abstractions and Python domain-specific languages (DSLs) designed to enable high-performance matrix-matrix multiplication (GEMM) and related computations across all levels within CUDA.

Table 33: Input Data Types Supported by Tensor Core Acceleration per Compute Capability

Compute Capability	Tensor Core Input Data Types								
	FP64	TF32	BF16	FP16	FP8	FP6	FP4	INT8	INT4
7.5				Yes				Yes	Yes
8.0	Yes	Yes	Yes	Yes				Yes	Yes
8.6		Yes	Yes	Yes				Yes	Yes
8.7		Yes	Yes	Yes				Yes	Yes
8.9		Yes	Yes	Yes	Yes			Yes	Yes
9.0	Yes	Yes	Yes	Yes	Yes			Yes	
10.0	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	
10.3		Yes	Yes	Yes	Yes	Yes	Yes	Yes	
11.0		Yes	Yes	Yes	Yes	Yes	Yes	Yes	
12.x		Yes	Yes	Yes	Yes	Yes	Yes	Yes	

5.2. CUDA Environment Variables

The following section lists the CUDA environment variables. Those related to the Multi-Process Service (MPS) are documented in the GPU Deployment and Management Guide.

5.2.1. Device Enumeration and Properties

5.2.1.1 CUDA_VISIBLE_DEVICES

The environment variable controls which GPU devices are visible to a CUDA application and in what order they are enumerated.

- If the variable is not set, all GPU devices are visible.
- If the variable is set to an empty string, no GPU devices are visible.

Possible Values: A comma-separated sequence of GPU identifiers.

GPU identifiers are provided as:

- **Integer indices**: These correspond to the ordinal number of the GPU in the system, as determined by `nvidia-smi`, starting from 0. For example, setting `CUDA_VISIBLE_DEVICES=2,1` makes device 0 not visible and enumerates device 2 before device 1.
  - If an invalid index is encountered, only devices with indices that appear before the invalid index in the list are visible. For example, setting `CUDA_VISIBLE_DEVICES=0,2,-1,1` makes devices 0 and 2 visible, while device 1 is not visible because it appears after the invalid index `-1`.
- **GPU UUID strings**: These should follow the same format as given by `nvidia-smi -L`, such as `GPU-8932f937-d72c-4106-c12f-20bd9faed9f6`. However, for convenience, abbreviated forms are allowed; simply specify enough digits from the beginning of the GPU UUID to uniquely identify that GPU in the target system. For example, `CUDA_VISIBLE_DEVICES=GPU-8932f937` may be a valid way to refer to the above GPU UUID, assuming no other GPU in the system shares this prefix.
- Multi-Instance GPU (MIG) support: `MIG-<GPU-UUID>/<GPU instance ID>/<compute instance ID>`. For example, `MIG-GPU-8932f937-d72c-4106-c12f-20bd9faed9f6/1/2`. Only single MIG instance enumeration is supported.

The device count returned by the `cudaGetDeviceCount()` API includes only the visible devices, so CUDA APIs that use integer device identifiers only support ordinals in the range [0, visible device count - 1]. The enumeration order of the GPU devices determines the ordinal values. For example, with `CUDA_VISIBLE_DEVICES=2,1`, calling `cudaSetDevice(0)` will set device 2 as the current device, as it is enumerated first and assigned an ordinal of 0. Calling `cudaGetDevice(&device_ordinal)` after that will also set `device_ordinal` to 0, which corresponds to device 2.

Examples:

```
nvidia-smi -L # Get list of GPU UUIDs
CUDA_VISIBLE_DEVICES=0,1
CUDA_VISIBLE_DEVICES=GPU-8932f937-d72c-4106-c12f-20bd9faed9f6
CUDA_VISIBLE_DEVICES=MIG-GPU-8932f937-d72c-4106-c12f-20bd9faed9f6/1/2
```

5.2.1.2 CUDA_DEVICE_ORDER

The environment variable controls the order in which CUDA enumerates the available devices.

Possible Values:

- FASTEST_FIRST: The available devices are enumerated from fastest to slowest using a simple heuristic (default).
- PCI_BUS_ID: The available devices are enumerated by PCI bus ID in ascending order. The PCI bus IDs can be obtained with `nvidia-smi --query-gpu=name,pci.bus_id`.

Examples:

```
CUDA_DEVICE_ORDER=FASTEST_FIRST
CUDA_DEVICE_ORDER=PCI_BUS_ID
nvidia-smi --query-gpu=name,pci.bus_id # Get list of PCI bus IDs
```

5.2.1.3 CUDA_MANAGED_FORCE_DEVICE_ALLOC

The environment variable alters how *Unified Memory* is physically stored in multi-GPU systems.

Possible Values: Numerical value, either zero or non-zero.

- **Non-zero value**: Forces the driver to use device memory for physical storage. All devices used in the process that support managed memory must be peer-to-peer compatible. Otherwise, `cudaErrorInvalidDevice` is returned.
- `0`: Default behavior.

Examples:

```
CUDA_MANAGED_FORCE_DEVICE_ALLOC=0
CUDA_MANAGED_FORCE_DEVICE_ALLOC=1 # force device memory
```

5.2.2. JIT Compilation

5.2.2.1 CUDA_CACHE_DISABLE

The environment variable controls the behavior of the on-disk *Just-In-Time (JIT) compilation* cache. Disabling the JIT cache forces PTX to CUBIN compilation for a CUDA application each time it is executed, unless the CUBIN code for the running architecture is found in the binary.

Disabling the JIT cache increases an application's load time during initial execution. However, it can be useful for reducing the application's disk space and for diagnosing differences across driver versions or build flags.

Possible Values:

- 1: Disables PTX JIT caching.
- `0`: Enables PTX JIT caching (default).

Examples:

```
CUDA_CACHE_DISABLE=1 # disables caching
CUDA_CACHE_DISABLE=0 # enables caching
```

5.2.2.2 CUDA_CACHE_PATH

The environment variable specifies the directory path for the *Just-In-Time (JIT) compilation* cache.

Possible Values: The absolute path to the cache directory (with appropriate access permissions). The default values are:

- on Windows, `%APPDATA%\NVIDIA\ComputeCache`
- on Linux, `~/.nv/ComputeCache`

Example:

```
CUDA_CACHE_PATH=~/tmp
```

5.2.2.3 CUDA_CACHE_MAXSIZE

The environment variable specifies the *Just-In-Time (JIT) compilation* cache size in bytes. Binaries that exceed this size are not cached. If needed, older binaries are evicted from the cache to make room for newer ones.

Possible Values: Number of bytes. The default values are:

- On desktop/server platforms, `1073741824` (1 GiB)
- On embedded platforms, `268435456` (256 MiB)

`4294967296` (4 GiB) is the maximum size.

Example:

```
CUDA_CACHE_MAXSIZE=268435456 # 256 MiB
```

5.2.2.4 CUDA_FORCE_PTX_JIT and CUDA_FORCE_JIT

The environment variables instruct the CUDA driver to ignore any CUBIN embedded in an application and perform *Just-In-Time (JIT) compilation* of the embedded PTX code instead.

Forcing JIT compilation increases an application's load time during initial execution. However, it can be used to validate that PTX code is embedded in an application and that its Just-In-Time compilation is functioning properly. This ensures forward compatibility with future architectures.

`CUDA_FORCE_PTX_JIT` overrides `CUDA_FORCE_JIT`.

Possible Values:

- 1: Forces PTX JIT compilation.
- `0`: Default behavior.

Example:

```
CUDA_FORCE_PTX_JIT=1
```

5.2.2.5 CUDA_DISABLE_PTX_JIT and CUDA_DISABLE_JIT

The environment variables disable the *Just-In-Time (JIT) compilation* of embedded PTX code and use the compatible CUBIN embedded in an application.

A kernel will fail to load if it does not have embedded binary code, or if the embedded binary was compiled for an incompatible architecture. These environment variables can be used to validate that an application has compatible CUBIN code generated for each kernel. See the *Binary Compatibility* section for more details.

`CUDA_DISABLE_PTX_JIT` overrides `CUDA_DISABLE_JIT`.

Possible Values:

- 1: Disables PTX JIT compilation.
- `0`: Default behavior.

Example:

```
CUDA_DISABLE_PTX_JIT=1
```

5.2.2.6 CUDA_FORCE_PRELOAD_LIBRARIES

The environment variable affects the preloading of libraries required for NVVM and *Just-In-Time (JIT) compilation*.

Possible Values:

- 1: This forces the driver to preload the libraries required for NVVM and *Just-In-Time (JIT) compilation* during initialization. This increases the memory footprint and the time required for CUDA driver initialization. Setting this environment variable is necessary to avoid certain deadlock situations involving multiple threads.
- `0`: Default behavior.

Example:

```
CUDA_FORCE_PRELOAD_LIBRARIES=1
```

5.2.3. Execution

5.2.3.1 CUDA_LAUNCH_BLOCKING

The environment variable specifies whether to disable or enable asynchronous kernel launches.

Disabling asynchronous execution results in slower execution but is useful for debugging. It forces GPU work to run synchronously from the CPU's perspective. This allows CUDA API errors to be observed at the exact API call that triggered them, rather than later in the execution. Synchronous execution is useful for debugging purposes.

Possible Values:

- 1: Disables asynchronous execution.
- 0: Asynchronous execution (default).

Example:

```
CUDA_LAUNCH_BLOCKING=1
```

5.2.3.2 CUDA_DEVICE_MAX_CONNECTIONS

The environment variable controls the number of concurrent compute and copy engine connections (work queues), setting both to the specified value. If independent GPU tasks, namely kernels or copy operations launched from different CUDA streams, map to the same work queue, a false dependency is created which can lead to GPU work serialization, since the same underlying resource(s) are used. To reduce the probability of such false dependencies, it is recommended that the work queue count, controlled via this environment variable, be greater than or equal to the number of active CUDA streams per context.

Setting this environment variable also modifies the number of copy connections, unless they are explicitly set via the `CUDA_DEVICE_MAX_COPY_CONNECTIONS` environment variable.

Possible Values: 1 to 32 connections, default is 8 (assumes no MPS)

Example:

```
CUDA_DEVICE_MAX_CONNECTIONS=16
```

5.2.3.3 CUDA_DEVICE_MAX_COPY_CONNECTIONS

The environment variable controls the number of concurrent copy connections (work queues) involved in copy operations. It affects only devices of *compute capability* 8.0 and above.

The `CUDA_DEVICE_MAX_COPY_CONNECTIONS` overrides the value of copy connections set via `CUDA_DEVICE_MAX_CONNECTIONS`, if both were set.

Possible Values: 1 to 32 connections, default is 8 (assumes no MPS)

Example:

```
CUDA_DEVICE_MAX_COPY_CONNECTIONS=16
```

5.2.3.4 CUDA_SCALE_LAUNCH_QUEUES

The environment variable specifies the scaling factor for the size of the queues available for launching work (command buffer), namely the total number of pending kernels or host/device copy operations that can be enqueued on a device.

Possible Values: `0.25x`, `0.5x`, `2x`, `4x`

- Any value other than `0.25x`, `0.5x`, `2x` or `4x` is interpreted as `1x`.

Example:

```
CUDA_SCALE_LAUNCH_QUEUES=2x
```

5.2.3.5 CUDA_GRAPHS_USE_NODE_PRIORITY

The environment variable controls the CUDA graph's execution priority relative to the stream priority it inherits from the stream in which it is launched.

`CUDA_GRAPHS_USE_NODE_PRIORITY` overrides the cudaGraphInstantiateFlagUseNodePriority flag on graph instantiation.

Possible Values:

- `0`: Inherit the priority of the stream the graph is launched into (default).
- 1: Honor per-node launch priorities. The CUDA runtime treats node-level priorities as a scheduling hint for ready-to-run graph nodes.

Example:

```
CUDA_GRAPHS_USE_NODE_PRIORITY=1
```

5.2.3.6 CUDA_DEVICE_WAITS_ON_EXCEPTION

The environment variable controls the behavior of a CUDA application when an exception (error) occurs.

When enabled, a CUDA application will halt and wait when a device-side exception occurs, allowing a debugger, such as cuda-gdb, to be attached to inspect the live GPU state before the process exits or continues.

Possible Values:

- `0`: Default behavior.
- 1: Halt when a device exception occurs.

Example:

```
CUDA_DEVICE_WAITS_ON_EXCEPTION=1
```

5.2.3.7 CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT

The environment variable controls the default "set-aside" portion of the GPU's L2 cache reserved for *persisting accesses*, expressed as a percentage of the L2 size.

It is relevant for GPUs that support persistent L2 cache, specifically devices with *compute capability* 8.0 or higher when using the CUDA Multi-Process Service (MPS). The environment variable must be set before starting the CUDA MPS Control Daemon, namely before running the `nvidia-cuda-mps-control -d` command.

Possible Values: Percentage value between 0 and 100, default is 0.

Example:

```
CUDA_DEVICE_DEFAULT_PERSISTING_L2_CACHE_PERCENTAGE_LIMIT=25 # 25%
```

5.2.3.8 CUDA_AUTO_BOOST [[deprecated]]

The environment variable affects the GPU clock "auto boost" behavior, namely dynamic clock boosting. It overrides the "auto boost" option of the nvidia-smi tool, namely `nvidia-smi --auto-boost-default=0`.

Note

This environment variable is deprecated. It is strongly suggested to use `nvidia-smi --applications-clocks=<memory,graphics>` or the NVML API instead of the CUDA_AUTO_BOOST environment variable.

5.2.4. Module Loading

5.2.4.1 CUDA_MODULE_LOADING

The environment variable affects how the CUDA runtime loads modules, specifically how it initializes the device code.

Possible Values:

- DEFAULT: Default behavior, equivalent to LAZY.
- LAZY: The loading of specific kernels is delayed until a CUDA function handle, `CUfunc`, is extracted using the `cuModuleGetFunction()` or `cuKernelGetFunction()` API calls. In this case, the data from the CUBIN is loaded when the first kernel in the CUBIN is loaded or when the first variable in the CUBIN is accessed.
  - The driver loads the required code on the first call to a kernel; subsequent calls incur no extra overhead. This reduces startup time and GPU memory footprint.
- EAGER: Fully loads CUDA modules and kernels at program initialization. All kernels and data from a CUBIN, FATBIN, or PTX file are fully loaded upon the corresponding `cuModuleLoad*` and `cuLibraryLoad*` driver API call.
  - Higher startup time and GPU memory footprint. Kernel launch overhead is predictable.

Examples:

```
CUDA_MODULE_LOADING=EAGER
CUDA_MODULE_LOADING=LAZY
```

5.2.4.2 CUDA_MODULE_DATA_LOADING

The environment variable affects how the CUDA runtime loads data associated to modules.

This is a complementary setting to the kernel-focused setting in CUDA_MODULE_LOADING. This environment variable does not affect the LAZY or EAGER loading of kernels. Data loading behavior is inherited from CUDA_MODULE_LOADING if this environment variable is not set.

Possible Values:

- DEFAULT: Default behavior, equivalent to LAZY.
- LAZY: The loading of module data is delayed until a CUDA function handle, CUfunc, is required. In this case, the data from the CUBIN is loaded when the first kernel in the CUBIN is loaded or when the first variable in the CUBIN is accessed.
  - Lazy data loads can require context synchronization, which can slow down concurrent execution.
- EAGER: All data from a CUBIN, FATBIN, or PTX file are fully loaded upon the corresponding cuModuleLoad* and cuLibraryLoad* API call.

Example:

```
CUDA_MODULE_DATA_LOADING=EAGER
```

5.2.4.3 CUDA_BINARY_LOADER_THREAD_COUNT

Sets the number of CPU threads to use when loading device binaries. When set to 0, the number of CPU threads used is set to a default value of 1.

Possible Values:

- Integer number of threads to use. Defaults to 0, which uses 1 thread.

Example:

```
CUDA_BINARY_LOADER_THREAD_COUNT=4
```

5.2.5. CUDA Error Log Management

5.2.5.1 CUDA_LOG_FILE

The environment variable specifies a location where descriptive error log messages will be printed as they occur for supported CUDA API calls that returned an error.

For example, if one attempts to launch a kernel with an invalid grid configuration, such as `kernel<<<1, dim3(1,1,128)>>>(...)`, that kernel will fail to launch and `cudaGetLastError()` will return a generic `invalid configuration argument` error. If the CUDA_LOG_FILE environment variable is set, the user can see the following descriptive error message in the log: `[CUDA][E] Block Dimensions (1,1,128) include one or more values that exceed the device limit of (1024,1024,64)` and easily determine that the specified z-dimension of the block was invalid. See *Error Log Management* for more details.

Possible Values: `stdout`, `stderr`, or a valid file path (with appropriate access permissions)

Examples:

```
CUDA_LOG_FILE=stdout
CUDA_LOG_FILE=/tmp/dbg_cuda_log
```

5.3. C++ Language Support

nvcc processes CUDA and device code according to the following specifications:

- **C++03** (ISO/IEC 14882:2003), `--std=c++03` flag.
- **C++11** (ISO/IEC 14882:2011), `--std=c++11` flag.
- **C++14** (ISO/IEC 14882:2014), `--std=c++14` flag.
- **C++17** (ISO/IEC 14882:2017), `--std=c++17` flag.
- **C++20** (ISO/IEC 14882:2020), `--std=c++20` flag.

Passing nvcc `-std=c++<version>` flag turns on all C++ features related to the specified version and also invokes the host preprocessor, compiler and linker with the corresponding C++ dialect option.

The compiler supports all language features of the supported standards, subject to the restrictions reported in the following sections.

5.3.1. C++11 Language Features

Table 34: C++11 Language Features Supported by NVCC for device code

Language Feature	**C++11 Proposal**	**NVCC/CUDA Toolkit 7.x**
Rvalue references	N2118	▯
Rvalue references for `*this`	N2439	▯
Initialization of class objects by rvalues	N1610	▯
Non-static data member initializers	N2756	▯
Variadic templates	N2242	▯
Extending variadic template template parameters	N2555	▯
Initializer lists	N2672	▯
Static assertions	N1720	▯
`auto`-typed variables	N1984	▯
Multi-declarator `auto`	N1737	▯
Removal of auto as a storage-class specifier	N2546	▯
New function declarator syntax	N2541	▯
Lambda expressions	N2927	▯

continues on next page

Table 34 – continued from previous page

Language Feature	C++11 Proposal	NVCC/CUDA Toolkit 7.x
Declared type of an expression	N2343	▯
Incomplete return types	N3276	▯
Right angle brackets	N1757	▯
Default template arguments for function templates	DR226	▯
Solving the SFINAE problem for expressions	DR339	▯
Alias templates	N2258	▯
Extern templates	N1987	▯
Null pointer constant	N2431	▯
Strongly-typed enums	N2347	▯
Forward declarations for enums	N2764 DR1206	▯
Standardized attribute syntax	N2761	▯
Generalized constant expressions	N2235	▯
Alignment support	N2341	▯
Conditionally-supported behavior	N1627	▯
Changing undefined behavior into diagnosable errors	N1727	▯
Delegating constructors	N1986	▯
Inheriting constructors	N2540	▯
Explicit conversion operators	N2437	▯
New character types	N2249	▯
Unicode string literals	N2442	▯
Raw string literals	N2442	▯
Universal character names in literals	N2170	▯
User-defined literals	N2765	▯
Standard Layout Types	N2342	▯
Defaulted functions	N2346	▯
Deleted functions	N2346	▯
Extended friend declarations	N1791	▯
Extending `sizeof`	N2253 DR850	▯
Inline namespaces	N2535	▯
Unrestricted unions	N2544	▯
Local and unnamed types as template arguments	N2657	▯

continues on next page

Table 34 – continued from previous page

Language Feature	C++11 Proposal	NVCC/CUDA Toolkit 7.x
Range-based for	N2930	▯
Explicit `virtual` overrides	N2928 N3206 N3272	▯
Minimal support for garbage collection and reachability-based leak detection	N2670	▯
Allowing move constructors to throw [noexcept]	N3050	▯
Defining move special member functions	N3053	▯
Concurrency		
Sequence points	N2239	▯
Atomic operations	N2427	▯
Strong Compare and Exchange	N2748	▯
Bidirectional Fences	N2752	▯
Memory model	N2429	▯
Data-dependency ordering: atomics and memory model	N2664	▯
Propagating exceptions	N2179	▯
Allow atomics use in signal handlers	N2547	▯
Thread-local storage	N2659	▯
Dynamic initialization and destruction with concurrency	N2660	▯
C99 Features in C++11		
`__func__` predefined identifier	N2340	▯
C99 preprocessor	N1653	▯
`long long`	N1811	▯
Extended integral types	N1988	▯

5.3.2. C++14 Language Features

Table 35: C++14 Language Features Supported by NVCC for device code

Language Feature	C++14 Proposal	NVCC/CUDA Toolkit 9.x
Tweak to certain C++ contextual conversions	N3323	▯
Binary literals	N3472	▯
Functions with deduced return type	N3638	▯
Generalized lambda capture (init-capture)	N3648	▯
Generic (polymorphic) lambda expressions	N3649	▯
Variable templates	N3651	▯
Relaxing requirements on constexpr functions	N3652	▯
Member initializers and aggregates	N3653	▯
Clarifying memory allocation	N3664	▯
Sized deallocation	N3778	▯
`[[deprecated]]` attribute	N3760	▯
Single-quotation-mark as a digit separator	N3781	▯

5.3.3. C++17 Language Features

Table 36: C++17 Language Features Supported by NVCC for device code

Language Feature	C++17 Proposal	NVCC/CUDA Toolkit 11.x
Removing trigraphs	N4086	▯
u8 character literals	N4267	▯
Folding expressions	N4295	▯
Attributes for namespaces and enumerators	N4266	▯
Nested namespace definitions	N4230	▯
Allow constant evaluation for all non-type template arguments	N4268	▯
Extending `static_assert`	N3928	▯
New Rules for `auto` deduction from braced-init-list	N3922	▯
Allow typename in a template template parameter	N4051	▯

continues on next page

Table 36 – continued from previous page

Language Feature	C++17 Proposal	NVCC/CUDA Toolkit 11.x
`[[fallthrough]]` attribute	P0188R1	▯
`[[nodiscard]]` attribute	P0189R1	▯
`[[maybe_unused]]` attribute	P0212R1	▯
Extension to aggregate initialization	P0017R1	▯
Wording for `constexpr` lambda	P0170R1	▯
Unary Folds and Empty Parameter Packs	P0036R0	▯
Generalizing the Range-Based For Loop	P0184R0	▯
Lambda capture of `*this` by Value	P0018R3	▯
Construction Rules for `enum class` variables	P0138R2	▯
Hexadecimal floating literals for C++	P0245R1	▯
Dynamic memory allocation for over-aligned data	P0035R4	▯
Guaranteed copy elision	P0135R1	▯
Refining Expression Evaluation Order for Idiomatic C++	P0145R3	▯
`constexpr if`	P0292R2	▯
Selection statements with initializer	P0305R1	▯
Template argument deduction for class templates	P0091R3 P0512R0	▯
Declaring non-type template parameters with `auto`	P0127R2	▯
Using attribute namespaces without repetition	P0028R4	▯
Ignoring unsupported non-standard attributes	P0283R2	▯
Structured bindings	P0217R3	▯
Remove Deprecated Use of the `register` Keyword	P0001R1	▯
Remove Deprecated `operator++(bool)`	P0002R1	▯
Make exception specifications be part of the type system	P0012R1	▯
`__has_include` for C++17	P0061R1	▯
Rewording inheriting constructors (core issue 1941 et al)	P0136R1	▯
Inline variables	P0386R2	▯
DR 150, Matching of template template arguments	P0522R0	▯
Removing dynamic exception specifications	P0003R5	▯
Pack expansions in using-declarations	P0195R2	▯
A `byte` type definition	P0298R0	▯

continues on next page

Table 36 – continued from previous page

Language Feature	C++17 Proposal	NVCC/CUDA Toolkit 11.x
DR 727, In-class explicit instantiations	CWG727	□

5.3.4. C++20 Language Features

GCC version □ 10.0, Clang version □ 10.0, Microsoft Visual Studio □ 2022, and nvc++ version □ 20.7.

Table 37: C++20 Language Features Supported by NVCC for device code

Language Feature	C++20 Proposal	NVCC/CUDA Toolkit 12.x
Default member initializers for bit-fields	P0683R1	□
Fixing `const`-qualified pointers to members	P0704R1	□
Allow lambda capture `[=, this]`	P0409R2	□
`__VA_OPT__` for preprocessor comma elision	P0306R4 P1042R1	□
Designated initializers	P0329R4	□
Familiar template syntax for generic lambdas	P0428R2	□
List deduction of vector	P0702R1	□
Concepts	P0734R0 P0857R0 P1084R2 P1141R2 P0848R3 P1616R1 P1452R2 P1972R0 P1980R0 P2092R0 P2103R0 P2113R0	□
Range-based for statements with initializer	P0614R1	□
Simplifying implicit lambda capture	P0588R1	□
ADL and function templates that are not visible	P0846R0	□
`const` mismatch with defaulted copy constructor	P0641R2	□
Less eager instantiation of `constexpr` functions	P0859R0	□
Consistent comparison (`operator<=>`)	P0515R3 P0905R1 P1120R0 P1185R2 P1186R3 P1630R1 P1946R0 P1959R0 P2002R1 P2085R0	□
Access checking on specializations	P0692R1	□
Default constructible and assignable stateless lambdas	P0624R2	□

continues on next page

Table 37 – continued from previous page

Language Feature	C++20 Proposal	NVCC/CUDA Toolkit 12.x
Lambdas in unevaluated contexts	P0315R4	▯
Language support for empty objects	P0840R2	▯
Relaxing the range-for loop customization point finding rules	P0962R1	▯
Allow structured bindings to accessible members	P0969R0	▯
Relaxing the structured bindings customization point finding rules	P0961R1	▯
Down with typename!	P0634R3	▯
Allow pack expansion in lambda init-capture	P0780R2 P2095R0	▯
Proposed wording for `likely` and `unlikely` attributes	P0479R5	▯
Deprecate implicit capture of this via `[=]`	P0806R2	▯
Class Types in Non-Type Template Parameters	P0732R2	▯
Inconsistencies with non-type template parameters	P1907R1	▯
Atomic Compare-and-Exchange with Padding Bits	P0528R3	▯
Efficient sized delete for variable sized classes	P0722R3	▯
Allowing Virtual Function Calls in Constant Expressions	P1064R0	▯
Prohibit aggregates with user-declared constructors	P1008R1	▯
`explicit(bool)`	P0892R2	▯
Signed integers are two's complement	P1236R1	▯
`char8_t`	P0482R6	▯
Immediate functions (`consteval`)	P1073R3 P1937R2	▯
`std::is_constant_evaluated`	P0595R2	▯
Nested `inline` namespaces	P1094R2	▯
Relaxations of `constexpr` restrictions	P1002R1 P1327R1 P1330R0 P1331R2 P1668R1 P0784R7	▯
Feature test macros	P0941R2	▯
Modules	P1103R3 P1766R1 P1811R0 P1703R1 P1874R1 P1979R0 P1779R3 P1857R3 P2115R0 P1815R2	▯
Coroutines	P0912R5	▯
Parenthesized initialization of aggregates	P0960R3 P1975R0	▯

continues on next page

Table 37 – continued from previous page

Language Feature	C++20 Proposal	NVCC/CUDA Toolkit 12.x
DR: array size deduction in new-expression	P1009R2	▯
DR: Converting from T* to bool should be considered narrowing	P1957R2	▯
Stronger Unicode requirements	P1041R4 P1139R2	▯
Structured binding extensions	P1091R3 P1381R1	▯
Deprecate `a[b,c]`	P1161R3	▯
Deprecating some uses of `volatile`	P1152R4	▯
`[[nodiscard("with reason")]]`	P1301R4	▯
`using enum`	P1099R5	▯
Class template argument deduction for aggregates	P1816R0 P2082R1	▯
Class template argument deduction for alias templates	P1814R0	▯
Permit conversions to arrays of unknown bound	P0388R4	▯
`constinit`	P1143R2	▯
Layout-compatibility and Pointer-interconvertibility Traits	P0466R5	▯
DR: Checking for abstract class types	P0929R2	▯
DR: More implicit moves	P1825R0	▯
DR: Pseudo-destructors end object lifetimes	P0593R6	▯

5.3.5. CUDA C++ Standard Library

CUDA provides an implementation of the C++ Standard Library (STL), called libcu++. The library presents the following benefits:

- The functionalities are available on both host and device.
- Compatible with all Linux and Windows platforms supported by the CUDA Toolkit.
- Compatible with all GPU architectures supported by the last two major versions of the CUDA Toolkit.
- Compatible with all CUDA Toolkits with the current and previous major versions.
- Provides C++17 backports of C++ Standard Library features available in recent standard versions, including C++20, C++23, and C++26.
- Supports extended data types, such as 128-bit integers (`__int128`), half-precision floats (`__half`), Bfloat16 (`__nv_bfloat16`), and quad-precision floats (`__float128`).
- Highly optimized for device code.

In addition, `libcu++` provides extended features that are not available in the C++ Standard Library to improve productivity and application performance. Such features include mathematical functions, memory operations, synchronization primitives, container extensions, high-level abstractions of CUDA intrinsics, C++ PTX wrappers, and more.

`libcu++` is available as part of the CUDA Toolkit, as well as part of the open-source CCCL repository.

5.3.6. C Standard Library Functions

5.3.6.1 `clock()` and `clock64()`

```
__host__ __device__ clock_t   clock();
__device__          long long clock64();
```

When executed in device code, it returns the value of a per-multiprocessor counter that increments every clock cycle. Sampling this counter at the beginning and end of a kernel, subtracting the two values, and recording the result for each thread provides an estimate of the number of clock cycles the device spends executing the thread. However, this value does not represent the actual number of clock cycles the device spends executing the thread's instructions. The former number is greater than the latter because threads are time-sliced.

Hint

- The corresponding CUDA C++ function `cuda::std::clock()` is provided in the `<cuda/std/ctime>` header.
- A portable C++ `<chrono>` implementation is also provided in the `<cuda/std/chrono>` header for similar purposes.

5.3.6.2 `printf()`

```
int printf(const char* format[, arg, ...]);
```

The function prints formatted output from a kernel to a host-side output stream.

The in-kernel `printf()` function behaves similarly to the standard C library `printf()` function. Users should refer to their host system's manual pages for complete descriptions of `printf()` behavior. Essentially, the string passed in as `format` is output to a stream on the host.

The `printf()` command is executed like any other device-side function: per thread and in the context of the calling thread. In a multi-threaded kernel, a straightforward call to `printf()` will be executed by every thread using the data specified by that thread. Consequently, multiple versions of the output string will appear at the host stream, each corresponding to a thread that encountered the `printf()`.

Unlike the C standard `printf()`, which returns the number of characters printed, CUDA's `printf()` returns the number of arguments parsed. If no arguments follow the format string, 0 is returned. If the format string is NULL, -1 is returned. If an internal error occurs, -2 is returned.

Internally, `printf()` uses a shared data structure, so it is possible that calling `printf()` may alter the execution order of threads. In particular, a thread that calls `printf()` might take a longer execution path than a thread that does not call `printf()`, and the length of that path depends on the parameters of `printf()`. However, note that CUDA makes no guarantees about the order of thread execution except at explicit `__syncthreads()` barriers. Therefore, it is impossible to tell whether the order of execution has been modified by `printf()` or by other scheduling behaviors in the hardware.

Format Specifiers

As for standard `printf()`, format specifiers take the form: `%[flags][width][.precision][size]type`

The following fields are supported. See the widely available documentation for a complete description of all behaviors.

- Flags: `#`, `' '`, `0`, `+`, `-`
- Width: `*`, `0-9`
- Precision: `0-9`
- Size: `h`, `l`, `ll`
- Type: `%cdiouxXpeEfgGaAs`

Limitations

The final formatting of the `printf()` output takes place on the host system. This means that the format string must be understood by the compiler and C library of the host system. While every effort has been made to ensure that the format specifiers supported by CUDA's `printf()` function are a universal subset of those supported by the most common host compilers, the exact behavior will be dependent on the host operating system.

`printf()` accepts all valid combinations of flags and types. This is because it cannot determine what will and will not be valid on the host system where the final output is formatted. Consequently, output may be undefined if the program emits a format string containing invalid combinations.

The `printf()` function can accept up to 32 arguments, in addition to the format string. Any additional arguments will be ignored, and the format specifier will be output as is.

Due to the different sizes of the `long` type on Windows platforms (32-bit) and Linux platforms (64-bit), a kernel compiled on a Linux machine and then run on a Windows machine will produce corrupted output for all format strings that include `%ld`. To ensure safety, it is recommended that the compilation and execution platforms match.

Host-Side Buffer

The output buffer for `printf()` is set to a fixed size before kernel launch. The buffer is circular, so if more output is produced during kernel execution than can fit in the buffer, older output is overwritten. The buffer is flushed only when one of the following actions is performed:

- Kernel launch via `<<< >>>` or `cuLaunchKernel()`: at the start of the launch, and if the `CUDA_LAUNCH_BLOCKING` environment variable is set to 1, at the end of the launch as well,
- Synchronization via `cudaDeviceSynchronize()`, `cuCtxSynchronize()`, `cudaStreamSynchronize()`, `cuStreamSynchronize()`, `cudaEventSynchronize()`, or `cuEventSynchronize()`,
- Memory copies via any blocking version of `cudaMemcpy*()` or `cuMemcpy*()`,
- Module loading/unloading via `cuModuleLoad()` or `cuModuleUnload()`,
- Context destruction via `cudaDeviceReset()` or `cuCtxDestroy()`.
- Prior to executing a stream callback added by `cudaLaunchHostFunc()` or `cuLaunchHostFunc()`.

Note that the buffer is not automatically flushed when the program exits.

The following API functions set and retrieve the size of the buffer used to transfer `printf()` arguments and internal metadata to the host. The default size is one megabyte.

- `cudaDeviceGetLimit(size_t* size,cudaLimitPrintfFifoSize)`
- `cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size_t size)`

Examples

The following code sample:

```
#include <stdio.h>

__global__ void helloCUDA(float value) {
    printf("Hello thread %d, value=%f\n", threadIdx.x, value);
}

int main() {
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Hello thread 2, value=1.2345
Hello thread 1, value=1.2345
Hello thread 4, value=1.2345
Hello thread 0, value=1.2345
Hello thread 3, value=1.2345
```

Notice that each thread encounters the `printf()` command. Therefore, there are as many lines of output as there are threads in the grid.

See the example on Compiler Explorer.

The following code sample:

```
#include <stdio.h>

__global__ void helloCUDA(float value) {
    if (threadIdx.x == 0)
        printf("Hello thread %d, value=%f\n", threadIdx.x, value);
}

int main() {
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Hello thread 0, value=1.2345
```

Clearly, the `if()` statement limits which threads call `printf()`, so only one line of output is seen.

See the example on Compiler Explorer.

5.3.6.3 memcpy() and memset()

```
__host__ __device__ void* memcpy(void* dest, const void* src, size_t size);
```

The function copies `size` bytes from the memory location pointed by `src` to the memory location pointed by `dest`.

```
__host__ __device__ void* memset(void* ptr, int value, size_t size);
```

The function sets `size` bytes of memory block pointed by `ptr` to `value`, interpreted as an `unsigned char`.

Hint

It is suggested to use the `cuda::std::memcpy()` and `cuda::std::memset()` functions provided in the `<cuda/std/cstring>` header as safer versions of `memcpy` and `memset`.

5.3.6.4 malloc() and free()

```
__host__ __device__ void* malloc(size_t size);
// or cuda::std::malloc(), cuda::std::calloc() in the <cuda/std/cstdlib> header
```

The functions `malloc()` (device-side), `cuda::std::malloc()`, and `cuda::std::calloc()` allocate at least `size` bytes from the device heap and return a pointer to the allocated memory. If insufficient memory exists to fulfill the request, it returns NULL. The returned pointer is guaranteed to be aligned to a 16-byte boundary.

```
__device__ void* __nv_aligned_device_malloc(size_t size, size_t align);
// or cuda::std::aligned_alloc() in the <cuda/std/cstdlib> header
```

The functions `__nv_aligned_device_malloc()` and C++ `cuda::std::aligned_alloc()` allocate at least `size` bytes from the device heap and return a pointer to the allocated memory. If there is insufficient memory to fulfill the requested size or alignment, it returns NULL. The address of the allocated memory is a multiple of `align`. `align` must be a non-zero power of two.

```
__host__ __device__ void free(void* ptr);
// or cuda::std::free() in the <cuda/std/cstdlib> header
```

The device-side functions `free()` and `cuda::std::free()` deallocate the memory pointed to by `ptr`, which must have been returned by a previous call to `malloc()`, `cuda::std::malloc()`, `cuda::std::calloc()`, `__nv_aligned_device_malloc()`, or `cuda::std::aligned_alloc()`. If `ptr` is NULL, the call to `free()` or `cuda::std::free()` is ignored. Repeated calls to `free()` or `cuda::std::free()` with the same `ptr` have undefined behavior.

Memory allocated by a given CUDA thread via `malloc()`, `cuda::std::malloc()`, `cuda::std::calloc()`, `__nv_aligned_device_malloc()`, or `cuda::std::aligned_alloc()` remain allocated for the lifetime of the CUDA context, or until it is explicitly released by a call to `free()` or `cuda::std::free()`. This memory can be used by other CUDA threads, even those from subsequent kernel launches. Any CUDA thread can free memory allocated by another thread; however, care should be taken to ensure that the same pointer is not freed more than once.

Heap Memory API

The size of the device memory heap must be specified before any program that allocates or frees memory in device code, including the `new` and `delete` keywords. If any program uses the device memory heap without explicitly specifying the heap size, a default heap of eight megabytes is allocated.

The following API functions get and set the heap size:

- `cudaDeviceGetLimit(size_t* size, cudaLimitMallocHeapSize)`
- `cudaDeviceSetLimit(cudaLimitMallocHeapSize, size_t size)`

The heap size granted will be at least `size` bytes. cuCtxGetLimit() and cudaDeviceGetLimit() return the currently requested heap size.

The actual memory allocation for the heap occurs when a module is loaded into the context, either explicitly through the CUDA driver API (see *Module*) or implicitly through the CUDA runtime API. If memory allocation fails, the module load generates a `CUDA_ERROR_SHARED_OBJECT_INIT_FAILED` error.

The heap size cannot be changed after a module has been loaded, and it does not dynamically resize according to need.

The memory reserved for the device heap is in addition to the memory allocated through host-side CUDA API calls such as `cudaMalloc()`.

Interoperability with the Host Memory API

Memory allocated via the device-side functions `malloc()`, `cuda::std::malloc()`, `cuda::std::calloc()`, `__nv_aligned_device_malloc()`, `cuda::std::aligned_alloc()`, or the `new` keyword cannot be used or freed with runtime or driver API calls such as `cudaMalloc`, `cudaMemcpy`, or `cudaMemset`. Similarly, memory allocated via the host runtime API cannot be freed using the device-side functions `free()`, `cuda::std::free()`, or the `delete` keyword.

Per-Thread Allocation example:

```
#include <stdlib.h>
#include <stdio.h>

__global__ void single_thread_allocation_kernel() {
    size_t size = 123;
    char*  ptr  = (char*) malloc(size);
    memset(ptr, 0, size);
    printf("Thread %d got pointer: %p\n", threadIdx.x, ptr);
    free(ptr);
}

int main() {
    // Set a heap size of 128 megabytes.
    // Note that this must be done before any kernel is launched.
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128 * 1024 * 1024);
    single_thread_allocation_kernel<<<1, 5>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Thread 0 got pointer: 0x20d5ffe20
Thread 1 got pointer: 0x20d5ffec0
Thread 2 got pointer: 0x20d5fff60
Thread 3 got pointer: 0x20d5f97c0
Thread 4 got pointer: 0x20d5f9720
```

Notice how each thread encounters the `malloc()` and `memset()` commands and so receives and initializes its own allocation.

See the example on Compiler Explorer.

Per-Thread-Block Allocation example:

```
#include <stdlib.h>

__global__ void block_level_allocation_kernel() {
    __shared__ int* data;
    // The first thread in the block performs the allocation and shares the
↪pointer
    // with all other threads through shared memory, so that access can be
↪coalesced.
    if (threadIdx.x == 0) {
        size_t size = blockDim.x * 64; // 64 bytes per thread are allocated.
        data = (int*) malloc(size);
    }
    __syncthreads();
    // Check for failure
    if (data == nullptr)
        return;

    // Threads index into the memory, ensuring coalescence
    for (int i = 0; i < 64; ++i)
        data[i * blockDim.x + threadIdx.x] = threadIdx.x;
    // Ensure all threads complete before freeing
    __syncthreads();

    // Only one thread may free the memory!
    if (threadIdx.x == 0)
        free(data);
}

int main() {
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128 * 1024 * 1024);
    block_level_allocation_kernel<<<10, 128>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

See the example on Compiler Explorer.

Allocation Persisting Between Kernel Launches example:

```
#include <stdlib.h>
#include <stdio.h>

const int NUM_BLOCKS = 20;

__device__ int* data_ptrs[NUM_BLOCKS]; // Per-block pointer

__global__ void allocate_memory_kernel() {
    // Only the first thread in the block performs the allocation
    // since we need only one allocation per block.
    if (threadIdx.x == 0)
        data_ptrs[blockIdx.x] = (int*) malloc(blockDim.x * 4);
    __syncthreads();
    // Check for failure
    if (data_ptrs[blockIdx.x] == nullptr)
        return;
    // Zero the data with all threads in parallel
    data_ptrs[blockIdx.x][threadIdx.x] = 0;
}

// Simple example: store the thread ID into each element
__global__ void use_memory_kernel() {
    int* ptr = data_ptrs[blockIdx.x];
    if (ptr != nullptr)
        ptr[threadIdx.x] += threadIdx.x;
}

// Print the content of the buffer before freeing it
__global__ void free_memory_kernel() {
    int* ptr = data_ptrs[blockIdx.x];
    if (ptr != nullptr)
        printf("Block %d, Thread %d: final value = %d\n",
            blockIdx.x, threadIdx.x, ptr[threadIdx.x]);
    // Only free from one thread!
    if (threadIdx.x == 0)
        free(ptr);
}

int main() {
    cudaDeviceSetLimit(cudaLimitMallocHeapSize, 128*1024*1024);
    // Allocate memory
    allocate_memory_kernel<<<NUM_BLOCKS, 10>>>();

    // Use memory
    use_memory_kernel<<<NUM_BLOCKS, 10>>>();
    use_memory_kernel<<<NUM_BLOCKS, 10>>>();
    use_memory_kernel<<<NUM_BLOCKS, 10>>>();

    // Free memory
    free_memory_kernel<<<NUM_BLOCKS, 10>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

See the example on Compiler Explorer.

5.3.6.5 alloca()

```
__host__ __device__ void* alloca(size_t size);
```

The `alloca()` function allocates `size` bytes of memory within the caller's stack frame. The returned value is a pointer to the allocated memory. When the function is invoked from device code, the beginning of the memory is 16-byte aligned. The memory is automatically freed when the caller returns from `alloca()`.

Note

On the Windows platform, the `<malloc.h>` header file must be included before using the `alloca()` function. Calls to `alloca()` may cause the stack to overflow; the user needs to adjust the stack size accordingly.

Example:

```
__device__ void device_function(int num_items) {
    int4* ptr = (int4*) alloca(num_items * sizeof(int4));
    // use of ptr
    ...
}
```

5.3.7. Lambda Expressions

The compiler determines the execution space of a lambda expression or closure type (C++11) by associating it with the execution space of the innermost enclosing function scope. If there is no enclosing function scope, the execution space is specified as `__host__`.

The execution space can also be specified explicitly with the *extended lambda syntax*.

Examples:

```
auto global_lambda = [](){ return 0; }; // __host__

void host_function() {
    auto lambda1 = [](){ return 1; };   // __host__
    [](){ return 3; };                  // __host__, closure type (body of a
↪lambda expression)
}

__device__ void device_function() {
    auto lambda2 = [](){ return 2; };   // __device__
}

__global__ void kernel_function(void) {
    auto lambda3 = [](){ return 3; };   // __device__
}
```

(continues on next page)

(continued from previous page)

```
__host__ __device__ void host_device_function() {
    auto lambda4 = [](){ return 4; };   // __host__ __device__
}

using function_ptr_t = int (*)();

__device__ void device_function(float          value,
                                function_ptr_t ptr = [](){ return 4; } /* __
↪host__ */) {}
```

See the example on Compiler Explorer.

5.3.7.1 Lambda Expressions and `__global__` Function Parameters

A lambda expression or a closure type can only be used as an argument to a `__global__` function if its execution space is `__device__` or `__host__ __device__`. Global or namespace scope lambda expressions cannot be used as arguments in a `__global__` function.

Examples:

```
template <typename T>
 __global__ void kernel(T input) {}

 __device__ void device_function() {
     // device kernel call requires separate compilation (-rdc=true flag)
     kernel<<<1, 1>>>([](){});
     kernel<<<1, 1>>>([] __device__() {});           // extended lambda
     kernel<<<1, 1>>>([] __host__ __device__() {}); // extended lambda
 }

 auto global_lambda = [] __host__ __device__() {};

 void host_function() {
     kernel<<<1, 1>>>([] __device__() {});           // CORRECT, extended
↪lambda
     kernel<<<1, 1>>>([] __host__ __device__() {}); // CORRECT, extended
↪lambda
 //  kernel<<<1, 1>>>([](){});                      // ERROR, closure type
↪with host execution space
 //  kernel<<<1, 1>>>(global_lambda);               // ERROR, extended lambda,
↪but at global scope
 }
```

See the example on Compiler Explorer.

5.3.7.2 Extended Lambdas

The nvcc flag `--extended-lambda` allows explicit annotations of execution spaces in a lambda expression. These annotations should appear after the lambda introducer and before the optional lambda declarator.

nvcc defines the macro `__CUDACC_EXTENDED_LAMBDA__` when the `--extended-lambda` flag is specified.

- An *extended lambda* is defined within the scope of an immediate or nested block of a `__host__` or `__host__ __device__` function.
- An *extended device lambda* is a lambda expression annotated with the `__device__` keyword.
- An *extended host-device lambda* is a lambda expression annotated with the `__host__ __device__` keywords.

Unlike standard lambda expressions, extended lambdas can be used as type arguments in `__global__` functions.

Example:

```
void host_function() {
    auto lambda1 = [] {};                    // NOT an extended lambda: no
↪explicit execution space annotations
    auto lambda2 = [] __device__ {};         // extended lambda
    auto lambda3 = [] __host__ __device__ {}; // extended lambda
    auto lambda4 = [] __host__ {};           // NOT an extended lambda
}

__host__ __device__ void host_device_function() {
    auto lambda1 = [] {};                    // NOT an extended lambda: no
↪explicit execution space annotations
    auto lambda2 = [] __device__ {};         // extended lambda
    auto lambda3 = [] __host__ __device__ {}; // extended lambda
    auto lambda4 = [] __host__ {};           // NOT an extended lambda
}

__device__ void device_function() {
    // none of the lambdas within this function are extended lambdas,
    // because the enclosing function is not a __host__ or __host__ __device__
↪ function.
    auto lambda1 = [] {};
    auto lambda2 = [] __device__ {};
    auto lambda3 = [] __host__ __device__ {};
    auto lambda4 = [] __host__ {};
}

auto global_lambda = [] __host__ __device__ { }; // NOT an extended lambda
↪because it is not defined
                                                  // within a __host__ or __
↪host__ __device__ function
```

5.3.7.3 Extended Lambda Type Traits

The compiler provides type traits to detect closure types for extended lambdas at compile time.

```
bool __nv_is_extended_device_lambda_closure_type(type);
```

The function returns `true` if `type` is the closure class created for an extended `__device__` lambda, `false` otherwise.

```
bool __nv_is_extended_device_lambda_with_preserved_return_type(type);
```

The function returns `true` if `type` is the closure class created for an extended `__device__` lambda and the lambda is defined with trailing return type, `false` otherwise. If the trailing return type definition refers to any lambda parameter name, the return type is not preserved.

```
bool __nv_is_extended_host_device_lambda_closure_type(type);
```

The function returns `true` if `type` is the closure class created for an extended `__host__ __device__` lambda, `false` otherwise.

The lambda type traits can be used in all compilation modes, regardless of whether lambdas or extended lambdas are enabled. The traits will always return `false` if extended lambda mode is inactive.

Example:

```
auto lambda0 = [] __host__ __device__ { };

void host_function() {
    auto lambda1 = [] { };
    auto lambda2 = [] __device__ { };
    auto lambda3 = [] __host__ __device__ { };
    auto lambda4 = [] __device__ () -> double { return 3.14; }
    auto lambda5 = [] __device__ (int x) -> decltype(&x) { return 0; }

    using lambda0_t = decltype(lambda0);
    using lambda1_t = decltype(lambda1);
    using lambda2_t = decltype(lambda2);
    using lambda3_t = decltype(lambda3);
    using lambda4_t = decltype(lambda4);
    using lambda5_t = decltype(lambda5);

    // 'lambda0' is not an extended lambda because it is defined outside
↪function scope
    static_assert(!__nv_is_extended_device_lambda_closure_type(lambda0_t));
    static_assert(!__nv_is_extended_device_lambda_with_preserved_return_
↪type(lambda0_t));
    static_assert(!__nv_is_extended_host_device_lambda_closure_type(lambda0_
↪t));

    // 'lambda1' is not an extended lambda because it has no execution space
↪annotations
    static_assert(!__nv_is_extended_device_lambda_closure_type(lambda1_t));
    static_assert(!__nv_is_extended_device_lambda_with_preserved_return_
↪type(lambda1_t));
    static_assert(!__nv_is_extended_host_device_lambda_closure_type(lambda1_
↪t));

    // 'lambda2' is an extended device-only lambda
    static_assert(__nv_is_extended_device_lambda_closure_type(lambda2_t));
    static_assert(!__nv_is_extended_device_lambda_with_preserved_return_
↪type(lambda2_t));
    static_assert(!__nv_is_extended_host_device_lambda_closure_type(lambda2_
↪t));

    // 'lambda3' is an extended host-device lambda
```

(continues on next page)

(continued from previous page)

```
    static_assert(!__nv_is_extended_device_lambda_closure_type(lambda3_t));
    static_assert(!__nv_is_extended_device_lambda_with_preserved_return_
↪type(lambda3_t));
    static_assert(__nv_is_extended_host_device_lambda_closure_type(lambda3_
↪t));

    // 'lambda4' is an extended device-only lambda with preserved return type
    static_assert(__nv_is_extended_device_lambda_closure_type(lambda4_t));
    static_assert(__nv_is_extended_device_lambda_with_preserved_return_
↪type(lambda4_t));
    static_assert(!__nv_is_extended_host_device_lambda_closure_type(lambda4_
↪t));

    // 'lambda5' is not an extended device-only lambda with preserved return
↪type
    // because it references the operator()'s parameter types in the trailing
↪return type.
    static_assert(__nv_is_extended_device_lambda_closure_type(lambda5_t));
    static_assert(!__nv_is_extended_device_lambda_with_preserved_return_
↪type(lambda5_t));
    static_assert(!__nv_is_extended_host_device_lambda_closure_type(lambda5_
↪t));
}
```

5.3.7.4 Extended Lambda Restrictions

Before invoking the host compiler, the CUDA compiler replaces an extended lambda expression with an instance of a placeholder type defined in namespace scope. The placeholder type's template argument requires taking the address of a function that encloses the original extended lambda expression. This is necessary for correctly executing any `__global__` function template whose template argument involves the closure type of an extended lambda. The enclosing function is computed as follows.

By definition, an extended lambda is present within the immediate or nested block scope of a `__host__` or `__host__ __device__` function.

- If the function is not the `operator()` of a lambda expression, it is considered the enclosing function for the extended lambda.
- Otherwise, the extended lambda is defined within the immediate or nested block scope of the `operator()` of one or more enclosing lambda expressions.
  - If the outermost lambda expression is defined within the immediate or nested block scope of a function F, then F is the computed enclosing function.
  - Otherwise, the enclosing function does not exist.

Example:

```
void host_function() {
    auto lambda1 = [] __device__ { }; // enclosing function for lambda1 is
↪"host_function()"
    auto lambda2 = [] {
        auto lambda3 = [] {
            auto lambda4 = [] __host__ __device__ { }; // enclosing function
↪for lambda4 is "host_function"
```

(continues on next page)

(continued from previous page)

```
        };
    };
}

auto global_lambda = [] {
    auto lambda5 = [] __host__ __device__ { }; // enclosing function for
↪lambda5 does not exist
};
```

Extended Lambda Restrictions

1. An extended lambda cannot be defined inside another extended lambda expression. Example:

```
void host_function() {
    auto lambda1 = [] __host__ __device__  {
         // ERROR, extended lambda defined within another extended lambda
        auto lambda2 = [] __host__ __device__ { };
    };
}
```

2. An extended lambda cannot be defined inside a generic lambda expression. Example:

```
void host_function() {
    auto lambda1 = [] (auto) {
         // ERROR, extended lambda defined within a generic lambda
        auto lambda2 = [] __host__ __device__ { };
    };
}
```

3. If an extended lambda is defined within the immediate or nested block scope of one or more nested lambda expressions, then the outermost lambda expression must be defined within the immediate or nested block scope of a function. Example:

```
auto lambda1 = []  {
    // ERROR, outer enclosing lambda is not defined within a non-lambda-
↪operator() function
    auto lambda2 = [] __host__ __device__ { };
};
```

4. The enclosing function of the extended lambda must be named, and its address must be accessible. If the enclosing function is a class member, the following conditions must be met:

   - All classes enclosing the member function must have a name.
   - The member function must not have private or protected access within its parent class.
   - All enclosing classes must not have private or protected access within their respective parent classes.

   Example:

```
void host_function() {
    auto lambda1 = [] __device__ { return 0; }; // OK
    {
        auto lambda2 = [] __device__          { return 0; }; // OK
        auto lambda3 = [] __device__ __host__ { return 0; }; // OK
    }
}

struct MyStruct1 {
    MyStruct1() {
        auto lambda4 = [] __device__ { return 0; }; // ERROR, address of
↪the enclosing function is not accessible
    }
};

class MyStruct2 {
    void foo() {
        auto temp1 = [] __device__ { return 10; }; // ERROR, enclosing
↪function has private access in parent class
    }

    struct MyStruct3 {
        void foo() {
            auto temp1 = [] __device__ { return 10; };  // ERROR,
↪enclosing class MyStruct3 has private access in its parent class
        }
    };
};
```

5. At the point where the extended lambda has been defined, it must be possible to unambiguously take the address of the enclosing routine. However, this may not always be feasible, for example, when an alias declaration shadows a template type argument with the same name. Example:

```
template <typename T>
struct A {
    using Bar = void;
    void test();
};

template<>
struct A<void> { };

template <typename Bar>
void A<Bar>::test() {
    // In code sent to host compiler, nvcc will inject an address
↪expression here, of the form:
    //   (void (A< Bar> ::*)(void))(&A::test))
    //  However, the class typedef 'Bar' (to void) shadows the template
↪argument 'Bar',
    //  causing the address expression in A<int>::test to actually refer
↪to:
    //    (void (A< void> ::*)(void))(&A::test))
    //  which doesn't take the address of the enclosing routine 'A<int>
```

(continues on next page)

(continued from previous page)

```
↪::test' correctly.
    auto lambda1 = [] __host__ __device__ { return 4; };
}

int main() {
    A<int> var;
    var.test();
}
```

6. An extended lambda cannot be defined in a class that is local to a function. Example:

```
void host_function() {
    struct MyStruct {
        void bar() {
            // ERROR, bar() is member of a class that is local to a
↪function
            auto lambda2 = [] __host__ __device__ { return 0; };
        }
    };
}
```

7. The enclosing function for an extended lambda cannot have deduced return type. Example:

```
auto host_function() {
    // ERROR, the return type of host_function() is deduced
    auto lambda3 = [] __host__ __device__ { return 0; };
}
```

8. A host-device extended lambda cannot be a generic lambda, namely a lambda with an `auto` parameter type. Example:

```
void host_function() {
    // ERROR, __host__ __device__ extended lambdas cannot be a generic
↪lambda
    auto lambda1 = [] __host__ __device__ (auto i) { return i; };

    // ERROR, a host-device extended lambda cannot be a generic lambda
    auto lambda2 = [] __host__ __device__ (auto... i) {
        return sizeof...(i);
    };
}
```

9. If the enclosing function is an instantiation of a function or member template, or if the function is a member of a class template, then the template(s) must satisfy the following constraints:

   - The template must have at most one variadic parameter, and it must be listed last in the template parameter list.
   - The template parameters must be named.
   - The template instantiation argument types cannot involve types that are either local to a function (except for closure types for extended lambdas), or are `private` or `protected` class members.

   Example 1:

```
template <template <typename...> class T,
          typename... P1,
          typename... P2>
void bar1(const T<P1...>, const T<P2...>) {
    // ERROR, enclosing function has multiple parameter packs
    auto lambda = [] __device__ { return 10; };
}

template <template <typename...> class T,
          typename... P1,
          typename    T2>
void bar2(const T<P1...>, T2) {
    // ERROR, for enclosing function, the parameter pack is not last in
↪the template parameter list
    auto lambda = [] __device__ { return 10; };
}

template <typename T, T>
void bar3() {
    // ERROR, for enclosing function, the second template parameter is not
↪named
    auto lambda = [] __device__ { return 10; };
}
```

Example 2:

```
template <typename T>
void bar4() {
    auto lambda1 = [] __device__ { return 10; };
}

class MyStruct {
    struct MyNestedStruct {};

    friend int main();
};

int main() {
    struct MyLocalStruct {};
    // ERROR, enclosing function for device lambda in bar4() is
↪instantiated with a type local to main
    bar4<MyLocalStruct>();

    // ERROR, enclosing function for device lambda in bar4 is instantiated
↪with a type
    //        that is a private member of a class
    bar4<MyStruct::MyNestedStruct>();
}
```

10. With Microsoft Visual Studio host compilers, the enclosing function must have external linkage. This restriction exists because the host compiler does not support using the addresses of non-extern linkage functions as template arguments. The CUDA compiler transformations require these addresses to support extended lambdas.

11. With Microsoft Visual Studio host compilers, an extended lambda shall not be defined within the body of an `if constexpr` block.
12. An extended lambda has the following restrictions on captured variables:
    - The variable may be passed by value to a sequence of helper functions in the code sent to the host compiler before being used to directly initialize the field of the class type representing the closure type for the extended lambda. However, the C++ standard specifies that the captured variable should be used for direct initialization of the closure type's field.
    - A variable can only be captured by value.
    - A variable of array type cannot be captured if the number of array dimensions is greater than 7.
    - For an array-type variable, the array field of the closure type is first default-initialized and then each array element is copy-assigned from the corresponding element of the captured array variable in the code sent to the host compiler. Therefore, the array element type must be both default-constructible and copy-assignable in the host code.
    - A function parameter that is an element of a variadic argument pack cannot be captured.
    - The captured variable type cannot be local to a function, except for extended lambda closure types, or `private` or `protected` class members.
    - Init-capture is not supported for host-device extended lambdas. However, it is supported for device extended lambdas, except when the initializer is an array or of type `std::initializer_list`.
    - The function call operator for an extended lambda is not a `constexpr`. The closure type of an extended lambda is not a literal type. The `constexpr` and `consteval` specifiers cannot be used when declaring an extended lambda.
    - A variable cannot be implicitly captured inside an `if-constexpr` block that is lexically nested inside an extended lambda unless the variable has been implicitly captured outside the `if-constexpr` block or appears in the extended lambda's explicit capture list.

    Examples:

```
void host_function() {
    // CORRECT, an init-capture is allowed for an extended device-only
↪lambda
    auto lambda1 = [x = 1] __device__ () { return x; };

    // ERROR, an init-capture is not allowed for an extended host-device
↪lambda
    auto lambda2 = [x = 1] __host__ __device__ () { return x; };

    int a = 1;
    // ERROR, an extended __device__ lambda cannot capture variables by
↪reference
    auto lambda3 = [&a] __device__ () { return a; };

    // ERROR, by-reference capture is not allowed for an extended device-
↪only lambda
    auto lambda4 = [&x = a] __device__ () { return x; };

    struct MyStruct {};
    MyStruct s1;
```

(continues on next page)

(continued from previous page)

```
    // ERROR, a type local to a function cannot be used in the type of a
↪captured variable
    auto lambda6 = [s1] __device__ () { };

    // ERROR, an init-capture cannot be of type std::initializer_list
    auto lambda7 = [x = {11}] __device__ () { };

    std::initializer_list<int> b = {11,22,33};
    // ERROR, an init-capture cannot be of type std::initializer_list
    auto lambda8 = [x = b] __device__ () { };

    int  var     = 4;
    auto lambda9 = [=] __device__ {
        int result = 0;
        if constexpr(false) {
            //ERROR, An extended device-only lambda cannot first-capture
↪'var' in if-constexpr context
            result += var;
        }
        return result;
    };

    auto lambda10 = [var] __device__ {
        int result = 0;
        if constexpr(false) {
            // CORRECT, 'var' already listed in explicit capture list for
↪the extended lambda
            result += var;
        }
        return result;
    };

    auto lambda11 = [=] __device__ {
        int result = var;
        if constexpr(false) {
            // CORRECT, 'var' already implicit captured outside the 'if-
↪constexpr' block
            result += var;
        }
        return result;
    };
}
```

13. When parsing a function, the CUDA compiler assigns a counter value to each extended lambda in the function. This counter value is used in the substituted named type that is passed to the host compiler. Therefore, the presence or absence of an extended lambda within a function should not depend on a particular value of __CUDA_ARCH__, nor on __CUDA_ARCH__ being undefined. Example:

```
template <typename T>
__global__ void kernel(T in) { in(); }
```

(continues on next page)

(continued from previous page)

```
__host__ __device__ void host_device_function() {
    // ERROR, the number and relative declaration order of
    //        extended lambdas depend on __CUDA_ARCH__
#if defined(__CUDA_ARCH__)
    auto lambda1 = [] __device__ { return 0; };
    auto lambda2 = [] __host__ __device__ { return 10; };
#endif
    auto lambda3 = [] __device__ { return 4; };
    kernel<<<1, 1>>>(lambda3);
}
```

14. As described above, the CUDA compiler replaces a device extended lambda defined in a host function with a placeholder type defined in namespace scope. The placeholder type does not define an `operator()` function equivalent to the original lambda declaration unless the trait `__nv_is_extended_device_lambda_with_preserved_return_type()` returns `true` for the closure type of the extended lambda. Therefore, an attempt to determine the return type or parameter types of the `operator()` function of such a lambda may work incorrectly in host code because the code processed by the host compiler is semantically different from the input code processed by the CUDA compiler. However, introspecting the return type or parameter types of the `operator()` function within device code is acceptable. Note that this restriction does not apply to host or device extended lambdas for which the trait `__nv_is_extended_device_lambda_with_preserved_return_type()` returns `true`. Example:

```
#include <cuda/std/type_traits>

const char& getRef(const char* p) { return *p; }

void foo() {
    auto lambda1 = [] __device__ { return "10"; };

    // ERROR, attempt to extract the return type of a device lambda in
↪host code
    cuda::std::result_of<decltype(lambda1)()>::type xx1 = "abc";

    auto lambda2 = [] __host__ __device__ { return "10"; };

    // CORRECT, lambda2 represents a host-device extended lambda
    cuda::std::result_of<decltype(lambda2)()>::type xx2 = "abc";

    auto lambda3 = [] __device__ () -> const char* { return "10"; };

    // CORRECT, lambda3 represents a device extended lambda with preserved
↪return type
    cuda::std::result_of<decltype(lambda3)()>::type xx2 = "abc";
    static_assert(cuda::std::is_same_v<cuda::std::result_of
↪<decltype(lambda3)()>::type, const char*>);

    auto lambda4 = [] __device__ (char x) -> decltype(getRef(&x)) {
↪return 0; };
    // lambda4's return type is not preserved because it references the
↪operator()'s
```

(continues on next page)

(continued from previous page)

```
    // parameter types in the trailing return type.
    static_assert(!__nv_is_extended_device_lambda_with_preserved_return_
↪type(decltype(lambda4)));
}
```

15. For an extended device-only lambda:
    - Introspection of the parameter type of `operator()` is only supported in device code.
    - Introspection of the return type of `operator()` is supported only in device code, unless the trait function `__nv_is_extended_device_lambda_with_preserved_return_type()` returns `true`.
16. If an extended lambda is passed from host to device code as an argument to a `__global__` function, for example, then any expression in the lambda's body that captures variables must remain unchanged, regardless of whether the `__CUDA_ARCH__` macro is defined and what value it has. This restriction arises because the lambda's closure class layout depends on the order in which the compiler encounters the captured variables when processing the lambda expression. The program may execute incorrectly if the closure class layout differs between device and host compilations. Example:

```
__device__ int result;

template <typename T>
__global__ void kernel(T in) { result = in(); }

void foo(void) {
    int x1 = 1;
    // ERROR, "x1" is only captured when __CUDA_ARCH__ is defined.
    auto lambda1 = [=] __host__ __device__ {
#ifdef __CUDA_ARCH__
        return x1 + 1;
#else
        return 10;
#endif
    };
    kernel<<<1, 1>>>(lambda1);
}
```

17. As previously described, the CUDA compiler replaces an extended device-only lambda expression with a placeholder type instance in the code sent to the host compiler. The placeholder type does not define a pointer-to-function conversion operator in the host code; however, the conversion operator is provided in the device code. Note that this restriction does not apply to host-device extended lambdas. Example:

```
template <typename T>
__global__ void kernel(T in) {
    int (*fp)(double) = in;
    fp(0); // CORRECT, conversion in device code is supported
    auto lambda1 = [](double) { return 1; };
}

void foo() {
    auto lambda_device      = [] __device__ (double) { return 1; };
```

(continues on next page)

(continued from previous page)

```
    auto lambda_host_device = [] __host__ __device__ (double) { return 1;
↪};
    kernel<<<1, 1>>>(lambda_device);
    kernel<<<1, 1>>>(lambda_host_device);

    // CORRECT, conversion for a __host__ __device__ lambda is supported
↪in host code
    int (*fp1)(double) = lambda_host_device;

    // ERROR, conversion for a device lambda is not supported in host code
    int (*fp2)(double) = lambda_device;
}
```

18. As previously described, the CUDA compiler replaces an extended device-only or host-device lambda expression with a placeholder type instance in the code sent to the host compiler. This placeholder type may define C++ special member functions, such as constructors and destructors. Consequently, some standard C++ type traits may yield different results for the closure type of the extended lambda in the CUDA front-end compiler than in the host compiler. The following type traits are affected: : `std::is_trivially_copyable`, `std::is_trivially_constructible`, `std::is_trivially_copy_constructible`, `std::is_trivially_move_constructible`, `std::is_trivially_destructible`. Care must be taken to ensure that the results of these traits are not used in the instantiation of the `__global__`, `__device__`, `__constant__`, or `__managed__` function or variable templates. Example:

```
#include <cstdio>
#include <type_traits>

template <bool b>
void __global__ kernel() { printf("hi"); }

template <typename T>
void kernel_launch() {
    // ERROR, this kernel launch may fail, because CUDA frontend compiler
↪and host compiler
    //        may disagree on the result of std::is_trivially_copyable_v
↪trait on the
    //        closure type of the extended lambda
    kernel<std::is_trivially_copyable_v<T>><<<1,1>>>();
    cudaDeviceSynchronize();
}

int main() {
    int  x      = 0;
    auto lambda1 = [=] __host__ __device__ () { return x; };
    kernel_launch<decltype(lambda1)>();
}
```

The CUDA compiler will generate compiler diagnostics for a subset of cases described in 1-12; no diagnostic will be generated for cases 13-17, but the host compiler may fail to compile the generated code.

5.3.7.5 Host-Device Lambda Optimization Notes

Unlike device-only lambdas, host-device lambdas can be called from host code. As previously mentioned, the CUDA compiler replaces an extended lambda expression defined in host code with an instance of a named placeholder type. The placeholder type for an extended host-device lambda invokes the original lambda's `operator()` with an indirect function call. The traits will always return false if extended lambda mode is not active.

The presence of an indirect function call may cause the host compiler to optimize an extended host-device lambda less than lambdas that are implicitly or explicitly `__host__` only. In the latter case, the host compiler can easily inline the lambda body into the calling context. However, when it encounters an extended host-device lambda, the host compiler may not be able to easily inline the original lambda body.

5.3.7.6 *this Capture By-Value

According to C++11/C++14 rules, when a lambda is defined within a non-`static` class member function and the lambda's body refers to a class member variable, the `this` pointer of the class must be captured by value rather than the referenced member variable. If the lambda is an extended device-only or host-device lambda defined in a host function and executed on the GPU, accessing the referenced member variable on the GPU will cause a runtime error if the `this` pointer points to host memory.

Example:

```
#include <cstdio>

template <typename T>
__global__ void foo(T in) { printf("value = %d\n", in()); }

struct MyStruct {
    int var;

    __host__ __device__ MyStruct() : var(10) {};

    void run() {
        auto lambda1 = [=] __device__ {
            // reference to "var" causes the 'this' pointer (MyStruct*) to be
↪captured by value
            return var + 1;
        };
        // Kernel launch fails at run time because 'this->var' is not
↪accessible from the GPU
        foo<<<1, 1>>>(lambda1);
        cudaDeviceSynchronize();
    }
};

int main() {
    MyStruct s1;
    s1.run();
}
```

C++17 solves this problem by introducing a new `*this` capture mode. In this mode, the compiler copies the object denoted by `*this` instead of capturing the `this` pointer by value. The `*this` capture mode is described in more detail in P0018R3.

The CUDA compiler supports the `*this` capture mode for lambdas defined within `__device__` and `__global__` functions and for extended device-only lambdas defined in host code, when the `--extended-lambda` flag is used.

Here's the above example modified to use `*this` capture mode:

```
#include <cstdio>

template <typename T>
__global__ void foo(T in) { printf("\n value = %d", in()); }

struct MyStruct {
    int var;
    __host__ __device__ MyStruct() : var(10) { };

    void run() {
        // note the "*this" capture specification
        auto lambda1 = [=, *this] __device__ {
            // reference to "var" causes the object denoted by '*this' to be
↪captured by
            // value, and the GPU code will access 'copy_of_star_this->var'
            return var + 1;
        };
        // Kernel launch succeeds
        foo<<<1, 1>>>(lambda1);
        cudaDeviceSynchronize();
    }
};

int main() {
    MyStruct s1;
    s1.run();
}
```

`*this` capture mode is not allowed for non-annotated lambdas defined in host code, or for extended host-device lambdas, unless `*this` capture is enabled by the selected language dialect. The following are examples of supported and unsupported usage:

```
struct MyStruct {
    int var;
    __host__ __device__ MyStruct() : var(10) { };

    void host_function() {
        // CORRECT, use in an extended device-only lambda
        auto lambda1 = [=, *this] __device__ { return var; };

        // Use in an extended host-device lambda
        // Error if *this capture not enabled by language dialect
        auto lambda2 = [=, *this] __host__ __device__ { return var; };

        // Use in an non-annotated lambda in host function
        // Error if *this capture not enabled by language dialect
        auto lambda3 = [=, *this]  { return var; };
    }
```

(continues on next page)

(continued from previous page)

```
    __device__ void device_function() {
        // CORRECT, use in a lambda defined in a device-only function
        auto lambda1 = [=, *this] __device__ { return var; };

        // CORRECT, use in a lambda defined in a device-only function
        auto lambda2 = [=, *this] __host__ __device__ { return var; };

        // CORRECT, use in a lambda defined in a device-only function
        auto lambda3 = [=, *this]  { return var; };
    }

    __host__ __device__ void host_device_function() {
        // CORRECT, use in an extended device-only lambda
        auto lambda1 = [=, *this] __device__ { return var; };

        // Use in an extended host-device lambda
        // Error if *this capture not enabled by language dialect
        auto lambda2 = [=, *this] __host__ __device__ { return var; };

        // Use in an unannotated lambda in a host-device function
        // Error if *this capture not enabled by language dialect
        auto lambda3 = [=, *this]  { return var; };
    }
};
```

5.3.7.7 Argument Dependent Lookup (ADL)

As previously mentioned, the CUDA compiler replaces an extended lambda expression with a placeholder type before invoking the host compiler. One template argument of the placeholder type uses the address of the function that encloses the original lambda expression. This may cause additional namespaces to participate in Argument-Dependent Lookup (ADL) for any host function call whose argument types involve the closure type of the extended lambda expression. Consequently, an incorrect function may be selected by the host compiler.

Example:

```
namespace N1 {

struct MyStruct {};

template <typename T>
void my_function(T);

}; // namespace N1

namespace N2 {

template <typename T>
int my_function(T);

template <typename T>
```

(continues on next page)

(continued from previous page)

```
void run(T in) { my_function(in); }

} // namespace N2

void bar(N1::MyStruct in) {
    // For extended device-only lambda, the code sent to the host compiler is
↪replaced with
    // the placeholder type instantiation expression
    //    ' __nv_dl_wrapper_t< __nv_dl_tag<void (*)(N1::MyStruct in),(&bar),1>
↪> { }'
    //
    // As a result, the namespace 'N1' participates in ADL lookup of the
    // call to "my_function()" in the body of N2::run, causing ambiguity.
    auto lambda1 = [=] __device__ { };
    N2::run(lambda1);
}
```

In the above example, the CUDA compiler replaced the extended lambda with a placeholder type involving the N1 namespace. Consequently, the N1 namespace participates in the ADL lookup for `my_function(in)` in the body of `N2::run()`, resulting in a host compilation failure due to the discovery of multiple overload candidates: `N1::my_function` and `N2::my_function`.

5.3.8. Polymorphic Function Wrappers

The `nvfunctional` header provides a polymorphic function wrapper class template, `nvstd::function`. Instances of this class template can store, copy, and invoke any callable target, such as lambda expressions. `nvstd::function` can be used in both host and device code.

Example:

```
#include <nvfunctional>

__host__            int host_function()        { return 1; }
__device__          int device_function()      { return 2; }
__host__ __device__ int host_device_function() { return 3; }

__global__ void kernel(int* result) {
    nvstd::function<int()> fn1 = device_function;
    nvstd::function<int()> fn2 = host_device_function;
    nvstd::function<int()> fn3 = [](){ return 10; };
    *result                    = fn1() + fn2() + fn3();
}

__host__ __device__ void host_device_test(int* result) {
    nvstd::function<int()> fn1 = host_device_function;
    nvstd::function<int()> fn2 = [](){ return 10; };
    *result                    = fn1() + fn2();
}

__host__ void host_test(int* result) {
    nvstd::function<int()> fn1 = host_function;
    nvstd::function<int()> fn2 = host_device_function;
```

(continues on next page)

(continued from previous page)

```
    nvstd::function<int()> fn3 = [](){ return 10; };
    *result                    = fn1() + fn2() + fn3();
}
```

Invalid cases:

- Instances of `nvstd::function` in host code cannot be initialized with the address of a `__device__` function or with a functor whose `operator()` is a `__device__` function.
- Similarly, instances of `nvstd::function` in device code cannot be initialized with the address of a `__host__` function or with a functor whose `operator()` is a `__host__` function.
- `nvstd::function` instances cannot be passed from host code to device code (or vice versa) at runtime.
- `nvstd::function` cannot be used in the parameter type of a `__global__` function if the `__global__` function is launched from host code.

Examples of invalid cases:

```
#include <nvfunctional>

__device__ int device_function() { return 1; }
__host__   int host_function() { return 3; }
auto       lambda_host  = [] { return 0; };

__global__ void k() {
    nvstd::function<int()> fn1 = host_function; // ERROR, initialized with
↪address of __host__ function
    nvstd::function<int()> fn2 = lambda_host;   // ERROR, initialized with
↪address of functor with
                                                //        __host__ operator()
↪function
}

__global__ void kernel(nvstd::function<int()> f1) {}

void foo(void) {
    auto lambda_device = [=] __device__ { return 1; };

    nvstd::function<int()> fn1 = device_function; // ERROR, initialized with
↪address of __device__ function
    nvstd::function<int()> fn2 = lambda_device;   // ERROR, initialized with
↪address of functor with
                                                  //        __device__
↪operator() function
    kernel<<<1, 1>>>(fn2);                        // ERROR, passing
↪nvstd::function from host to device
}
```

`nvstd::function` is defined in the `nvfunctional` header as follows:

```
namespace nvstd {

template <typename RetType, typename ...ArgTypes>
class function<RetType(ArgTypes...)> {
public:
    // constructors
    __device__ __host__ function() noexcept;
    __device__ __host__ function(nullptr_t) noexcept;
    __device__ __host__ function(const function&);
    __device__ __host__ function(function&&);

    template<typename F>
    __device__ __host__ function(F);

    // destructor
    __device__ __host__ ~function();

    // assignment operators
    __device__ __host__ function& operator=(const function&);
    __device__ __host__ function& operator=(function&&);
    __device__ __host__ function& operator=(nullptr_t);
    template<typename F>
    __device__ __host__ function& operator=(F&&);

    // swap
    __device__ __host__ void swap(function&) noexcept;

    // function capacity
    __device__ __host__ explicit operator bool() const noexcept;

    // function invocation
    __device__ RetType operator()(ArgTypes...) const;
};

// null pointer comparisons
template <typename R, typename... ArgTypes>
__device__ __host__
bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

template <typename R, typename... ArgTypes>
__device__ __host__
bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

template <typename R, typename... ArgTypes>
__device__ __host__
bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;

template <typename R, typename... ArgTypes>
__device__ __host__
bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

// specialized algorithms
template <typename R, typename... ArgTypes>
```

(continues on next page)

(continued from previous page)

```
__device__ __host__
void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

} // namespace nvstd
```

5.3.9. C/C++ Language Restrictions

5.3.9.1 Unsupported Features

- Run-Time Type Information (RTTI) and exceptions are not supported in device code:
  - `typeid` keyword
  - `dynamic_cast` keyword
  - `try/catch/throw` keywords
- `long double` is not supported in device code.
- Trigraphs are not supported on any platform. Digraphs are not supported on Windows.
- User-defined `operator new`, `operator new[]`, `operator delete`, or `operator delete[]` cannot be used to replace the corresponding built-ins provided by the compiler, and it is considered undefined behavior on both host and device.

5.3.9.2 Namespace Reservations

Unless otherwise noted, adding definitions to top-level namespaces `cuda::`, `nv::`, or `cooperative_groups::`, or to any nested namespace within them, is undefined behavior. We allow `cuda::` as a subnamespace as depicted below:

Examples:

```
namespace cuda {   // same for "nv" and "cooperative_groups" namespaces

struct foo;        // ERROR, class declaration in the "cuda" namespace

void bar();        // ERROR, function declaration in the "cuda" namespace

namespace utils {} // ERROR, namespace declaration in the "cuda" namespace

} // namespace cuda
```

```
namespace utils {
namespace cuda {

// CORRECT, namespace "cuda" may be used nested within a non-reserved namespace
void bar();

} // namespace cuda
} // namespace utils

// ERROR, Equivalent to adding symbols to namespace "cuda" at global scope
using namespace utils;
```

5.3.9.3 Pointers and Memory Addresses

Pointer dereferencing (`*pointer`, `pointer->member`, `pointer[0]`) is allowed only in the same execution space where the associated memory resides. The following cases result in undefined behavior, most often a segmentation fault and application termination.

- Dereferencing a pointer either to *global memory*, *shared memory*, or *constant memory* on the host.
- Dereferencing a pointer to host memory in device code.

The following restrictions apply to functions:

- It is not allowed to take the address of a `__device__` function in host code.
- The address of a `__global__` function taken in host code cannot be used in device code. Similarly, the address of a `__global__` function taken in device code cannot be used in host code.

The address of a `__device__` or `__constant__` variable obtained through `cudaGetSymbolAddress()` as described in the *Memory Space Specifiers* section can only be used in host code.

5.3.9.4 Variables

5.3.9.4.1 Local Variables

The `__device__`, `__shared__`, `__managed__`, and `__constant__` memory space specifiers are not allowed on non-`extern` variable declarations within a function that executes on the host.

Examples:

```
__host__ void host_function() {
    int x;                   // CORRECT, __host__ variable
    __device__   int y;      // ERROR,   __device__ variable declaration
↪within a host function
    __shared__   int z;      // ERROR,   __shared__ variable declaration
↪within a host function
    __managed__  int w;      // ERROR,   __managed__ variable  declaration
↪within a host function
    __constant__ int h;      // ERROR,   __constant__ variable declaration
↪within a host function
    extern __device__ int k; // CORRECT, extern __device__ variable
}
```

The `__device__`, `__constant__`, and `__managed__` memory space specifiers are not allowed on variable declarations that are neither `extern` nor `static` within a function that executes on the device.

```
__device__ void device_function() {
    int x;                   // CORRECT, __device__ variable
    __constant__      int y; // ERROR,   __constant__ variable declaration
↪within a device function
    __managed__       int z; // ERROR,   __managed__ variable  declaration
↪within a device function
    extern __device__ int k; // CORRECT, extern __device__ variable
}
```

see also the *static variables* section.

5.3.9.4.2 const-qualified Variables

A const-qualified variable without memory space annotations (`__device__` or `__constant__`) declared at global, namespace, or class scope is considered to be a host variable. Device code cannot contain a reference or take the address of the variable.

The variable may be directly used in device code, if

- it has been initialized with a constant expression before the point of use,
- the type is not `volatile`-qualified, and
- it has one of the following types:
  - built-in integral type, or
  - built-in floating point type, except when the host compiler is Microsoft Visual Studio.

Starting with C++14, it is recommended to use `constexpr` or `inline constexpr` (C++17) variables instead of `const`-qualified ones. `constexpr` variables are not subject to the same type restrictions and can be utilized directly in device code.

`__managed__` variables don't support `const`-qualified types.

Examples:

```
const                int   ConstVar           = 10;
const                float ConstFloatVar      = 5.0f;
inline constexpr float ConstexprFloatVar = 5.0f; // C++17

struct MyStruct {
    static const               int   ConstVar           = 20;
//  static const               float ConstFloatVar      = 5.0f; // ERROR, static
↪const variables cannot be float
    static inline constexpr float ConstexprFloatVar = 5.0f; // CORRECT
};

extern const int ExternVar;

__device__ void foo() {
    int array1[ConstVar];                         // CORRECT
    int array2[MyStruct::ConstVar];               // CORRECT

    const     float var1 = ConstFloatVar;         // CORRECT, except when the
↪host compiler is Microsoft Visual Studio.
    constexpr float var2 = ConstexprFloatVar; // CORRECT
//  int             var3 = ExternVar;             // ERROR, "ExternVar" is not
↪initialized with a constant expression
//  int&            var4 = ConstVar;              // ERROR, reference to host
↪variable
//  int*            var5 = &ConstVar;             // ERROR, address of host
↪variable
}
```

See the example on Compiler Explorer.

5.3.9.4.3 volatile-qualified Variables

Note

The `volatile` keyword is supported to maintain compatibility with ISO C++. However, few, if any, of its remaining non-deprecated uses apply to GPUs.

Reading and writing to `volatile`-qualified objects are not atomic and are compiled into one or more volatile instructions that do not guarantee:

- ordering of memory operations, or
- that the number of memory operations performed by the hardware matches the number of PTX instructions.

CUDA C++ `volatile` is NOT suitable for:

- **Inter-Thread Synchronization**: Use atomic operations via cuda::atomic_ref, cuda::atomic, or *Atomic Functions* instead.

  Atomic memory operations provide inter-thread synchronization guarantees and deliver better performance than `volatile` operations. However, CUDA C++ `volatile` operations do not provide any inter-thread synchronization guarantees and are therefore not suitable for this purpose. The following example shows how to pass a message between two threads using atomic operations.

  cuda::atomic_ref

```
#include <cuda/atomic>

__global__ void kernel(int* flag, int* data) {
    cuda::atomic_ref<int, cuda::thread_scope_device> atomic_ref{*flag};
    if (threadIdx.x == 0) {
        // Consumer: blocks until flag is set by producer, then reads
↪data
        while(atomic_ref.load(cuda::memory_order_acquire) == 0)
            ;
        if (*data != 42)
            __trap(); // Errors if wrong data read
    }
    else if (threadIdx.x == 1) {
        // Producer: writes data then sets flag
        *data = 42;
        atomic_ref.store(1, cuda::memory_order_release);
    }
}
```

cuda::atomic

```
#include <cuda/atomic>

__global__ void kernel(cuda::atomic<int, cuda::thread_scope_device>*
↪flag, int* data) {
    if (threadIdx.x == 0) {
        // Consumer: blocks until flag is set by producer, then reads
↪data
        while(flag->load(cuda::memory_order_acquire) == 0)
            ;
        if (*data != 42)
            __trap(); // Errors if wrong data read
    }
    else if (threadIdx.x == 1) {
        // Producer: writes data then sets flag
        *data = 42;
        flag->store(1, cuda::memory_order_release);
    }
}
```

Atomic Functions (`atomicAdd` and `atomicExch`)

```
__global__ void kernel(int* flag, int* data) {
    if (threadIdx.x == 0) {
        // Consumer: blocks until flag is set by producer, then reads
↪data
        while(atomicAdd(flag, 0) == 0)
            ;                 // Load with Relaxed Read-Modify-Write
        __threadfence();      // SequentiallyConsistent fence
        if (*data != 42)
            __trap();         // Errors if wrong data read
    } else if (threadIdx.x == 1) {
        // Producer: writes data then sets flag
        *data = 42;
        __threadfence();      // SequentiallyConsistent fence
        atomicExch(flag, 1); // Store with Relaxed Read-Modify-Write
    }
}
```

- **Memory Mapped IO** (MMIO): Use PTX MMIO operations via inline PTX instead.

  PTX MMIO operations strictly preserve the number of memory accesses performed. However, CUDA C++ `volatile` operations do not preserve the number of memory accesses performed and may perform more or fewer accesses than requested in an undetermined way. This makes them unsuitable for MMIO. The following example shows how to read from and write to a register using PTX MMIO operations.

```
__global__ void kernel(int* mmio_reg0, int* mmio_reg1) {
    // Write to MMIO register:
    int value = 13;
    asm volatile("st.relaxed.mmio.sys.u32 [%0], %1;"
        :
        : "l"(mmio_reg0), "r"(value) : "memory");

    // Read MMIO register:
    asm volatile("ld.relaxed.mmio.sys.u32 %0, [%1];"
        : "=r"(value)
        : "l"(mmio_reg1) : "memory");

    if (value != 42)
        __trap(); // Errors if wrong data read
}
```

5.3.9.4.4 static Variables

static variables are allowed in device code in the following cases:

- Within __global__ or __device__-only functions.
- Within __host__ __device__ functions:
  - static variables without an explicit memory space (automatic deduction).
  - static variables with an explicit memory space, such as static __device__/ __constant__/__shared__/__managed__, are allowed only when __CUDA_ARCH__ is defined.

A static variable within a __host__ __device__ function holds a different value depending on the execution space.

Examples of legal and illegal uses of function-scope static variables are shown below.

```
struct TrivialStruct {
    int x;
};

struct NonTrivialStruct {
    __device__ NonTrivialStruct(int x) {}
};

__device__ void device_function(int x) {
    static int v1;              // CORRECT, implicit __device__ memory space
↪specifier
    static int v2 = 11;         // CORRECT, implicit __device__ memory space
↪specifier
//  static int v3 = x;          // ERROR, dynamic initialization is not allowed

    static __managed__  int v4; // CORRECT, explicit
    static __device__   int v5; // CORRECT, explicit
    static __constant__ int v6; // CORRECT, explicit
    static __shared__   int v7; // CORRECT, explicit
```

(continues on next page)

(continued from previous page)

```
    static TrivialStruct    s1;     // CORRECT, implicit __device__ memory
↪space specifier
    static TrivialStruct    s2{22}; // CORRECT, implicit __device__ memory
↪space specifier
//  static TrivialStruct    s3{x};   // ERROR, dynamic initialization is not
↪allowed
//  static NonTrivialStruct s4{3};   // ERROR, dynamic initialization is not
↪allowed
}
```

See the example on Compiler Explorer.

```
__host__ __device__ void host_device_function() {
    static             int v1; // CORRECT, implicit __device__ memory space
↪specifier
//  static __device__ int v2;  // ERROR, __device__-only variable inside a host-
↪device function
#ifdef __CUDA_ARCH__
    static __device__ int v3; // CORRECT, declaration is only visible during
↪device compilation
#else
    static int v4;            // CORRECT, declaration is only visible during
↪host compilation
#endif
}
```

See the example on Compiler Explorer.

```
#include <cassert>

__host__ __device__ int host_device_function() {
    static int v = 0;
    v++;
    return v;
}

__global__ void kernel() {
    int ret = host_device_function(); // v = 1
    assert(ret == 4);                 // FAIL
}

int main() {
    host_device_function();           // v = 1
    host_device_function();           // v = 2
    int ret = host_device_function(); // v = 3
    assert(ret == 3);                 // OK
    kernel<<<1, 1>>>();
    cudaDeviceSynchronize();
}
```

See the example on Compiler Explorer.

5.3.9.4.5 extern Variables

When compiling in the *whole program compilation mode*, `__device__`, `__shared__`, `__managed__`, and `__constant__` variables cannot be defined with external linkage using the `extern` keyword.

The only exception is for dynamically allocated `__shared__` variables as described in the *Dynamic Allocation of Shared Memory* section.

```
__device__        int x; // OK
extern __device__ int y; // ERROR in whole program compilation mode
extern __shared__ int z; // OK
```

5.3.9.5 Functions

5.3.9.5.1 Recursion

`__global__` functions do not support recursion, while `__device__` and `__host__ __device__` functions do not have such restriction.

5.3.9.5.2 External Linkage

Device variables or functions with external linkage require *separate compilation mode* across multiple translation units.

In separate compilation mode, if a `__device__` or `__global__` function definition is required to exist in a particular translation unit, then the parameters and return types of the function must be complete in that translation unit. The concept is also known as One Definition Rule-use, or ODR-use.

Example:

```
//first.cu:
struct S;                    // forward declaration
__device__ void foo(S);      // ERROR, type 'S' is an incomplete type
__device__ auto* ptr = foo; // ODR-use, address taken

int main() {}
```

```
//second.cu:
struct S {};               // struct definition
__device__ void foo(S) {}  // function definition
```

```
# compiler invocation
$ nvcc -std=c++14 -rdc=true first.cu second.cu -o prog
nvlink error   : Prototype doesn't match for '_Z3foo1S' in '/tmp/tmpxft_
↪00005c8c_00000000-18_second.o',
                 first defined in '/tmp/tmpxft_00005c8c_00000000-18_second.o'
nvlink fatal   : merge_elf failed
```

5.3.9.5.3 Formal Parameters

The `__device__`, `__shared__`, `__managed__` and `__constant__` memory space specifiers are not allowed on formal parameters.

```
void device_function1(__device__ int x) { } // ERROR, __device__ parameter
void device_function2(__shared__ int x) { } // ERROR, __shared__ parameter
```

5.3.9.5.4 `__global__` Function Parameters

A `__global__` function has the following restrictions:

- It cannot have a variable number of arguments, namely the C ellipsis syntax `...` and the `va_list` type. C++11 variadic template is allowed, subject to the restrictions described in the *__global__ Variadic Template* section.
- Function parameters are passed to the device via *constant memory* and their total size is limited to 32,764 bytes.
- Function parameters cannot be pass-by-reference or by pass-by-rvalue reference.
- Function parameters cannot be of type `std::initializer_list`.
- Polymorphic class parameters (`virtual`) are considered undefined behavior.
- Lambda expressions and closure types are allowed, subject to the restrictions described in the *Lambda Expressions and __global__ Function Parameters* section.

5.3.9.5.5 `__global__` Function Arguments Passing

When launching a `__global__` function *from device code*, each argument must be trivially copyable and trivially destructible.

When a `__global__` function is launched from host code, each argument type may be non-trivially copyable or non-trivially destructible. However, the processing of these types does not follow the standard C++ model, as described below. The user code must ensure that this workflow does not affect program correctness. The workflow diverges from standard C++ in two areas:

1. **Raw memory copy instead of copy constructor invocation**

   The CUDA Runtime passes the kernel arguments to the `__global__` function by copying the raw memory content, eventually using `memcpy`. If an argument is non-trivially copyable and provides a user-defined copy constructor, the operations and side effects of the invocation are skipped in the host-to-device copy.

   Example:

```
#include <cassert>

struct MyStruct {
    int  value = 1;
    int* ptr;

    MyStruct() = default;

    __host__ __device__ MyStruct(const MyStruct&) { ptr = &value; }
};

__global__ void device_function(MyStruct my_struct) {
    // this assert fails because "my_struct" is obtained by copying
    // the raw memory content and the copy constructor is skipped.
    assert(my_struct.ptr == &my_struct.value); // FAIL
}

void host_function(MyStruct my_struct) {
    assert(my_struct.ptr == &my_struct.value); // CORRECT
```

(continues on next page)

(continued from previous page)

```
}

int main() {
    MyStruct my_struct;
    host_function(my_struct);
    device_function<<<1, 1>>>(my_struct); // copy constructor invoked in
↪the host-side only
    cudaDeviceSynchronize();
}
```

See the example on Compiler Explorer.

2. **Destructor may be invoked before the `__global__` function has finished**

   Kernel launches are asynchronous with host execution. As a result, if a `__global__` function argument has a non-trivial destructor, the destructor may execute in host code even before the `__global__` function has finished execution. This may break programs where the destructor has side effects.

   Example:

```
#include <cassert>

__managed__ int var = 0;

struct MyStruct {
    __host__ __device__ ~MyStruct() { var = 3; }
};

__global__ void device_function(MyStruct my_struct) {
    assert(var == 0); // FAIL, MyStruct::~MyStruct() sets the value to 3
}

int main() {
    MyStruct my_struct;
    // GPU kernel execution is asynchronous with host execution.
    // As a result, MyStruct::~MyStruct() could be executed before
    // the kernel finishes executing.
    device_function<<<1, 1>>>(my_struct);
    cudaDeviceSynchronize();
}
```

   See the example on Compiler Explorer.

5.3.9.6 Classes

5.3.9.6.1 Class-type Variables

A variable definition with `__device__`, `__constant__`, `__managed__` or `__shared__` memory space cannot have a class type with a non-empty constructor or a non-empty destructor. A constructor for a class type is considered empty if it is either trivial or satisfies all of the following conditions at a point in the translation unit:

- The constructor function has been defined.

- The constructor function has no parameters, an empty initializer list, and an empty compound statement function body.
- Its class has no `virtual` functions, `virtual` base classes, or non-`static` data member initializers.
- The default constructors of all of its base classes can be considered empty.
- For all non-`static` data members of the class that are of a class type (or an array thereof), the default constructors can be considered empty.

A class's destructor is considered empty if it is either trivial or satisfies all of the following conditions at a point in the translation unit:

- The destructor function has been defined.
- The destructor function body is an empty compound statement.
- Its class has no `virtual` functions or `virtual` base classes.
- The destructors of all of its base classes can be considered empty.
- For all non-`static` data members of the class that are of a class type (or an array thereof), the destructor can be considered empty.

5.3.9.6.2 Data Members

The `__device__`, `__shared__`, `__managed__` and `__constant__` memory space specifiers are not allowed on `class`, `struct`, and `union` data members.

Only `static` data members evaluated at compile time are supported, such as *const-qualified* and `constexpr` variables.

```
struct MyStruct {
   static inline constexpr int value1 = 10; // C++17
   static constexpr        int value2 = 10; // C++11
   static const            int value3 = 10;
// static                  int value4; // ERROR
};
```

5.3.9.6.3 Function Members

`__global__` functions cannot be members of a `struct`, `class`, or `union`.

A `__global__` function is allowed in a `friend` declaration, but cannot be defined.

Example:

```
struct MyStruct {
    friend __global__ void f();   // CORRECT, friend declaration only

//  friend __global__ void g() {} // ERROR, friend definition
};
```

See the example on Compiler Explorer.

5.3.9.6.4 Implicitly-Declared and Non-Virtual Explicitly-Defaulted functions

Implicitly-declared special member functions are those the compiler declares for a class when the user does not declare them; Explicitly-defaulted functions are ones the user declares but marks with `= default`. The special member functions that are implicitly-declared or explicitly-defaulted are default constructor, copy constructor, move constructor, copy assignment operator, move assignment operator, and destructor.

Let F denote a non-`virtual` function that is either implicitly declared or explicitly defaulted on its first declaration. The execution space specifiers for F are the union of the execution space specifiers of all functions that invoke it. Note that for this analysis, a `__global__` caller will be treated as a `__device__` caller. For example:

```
class Base {
    int x;
public:
    __host__ __device__ Base() : x(10) {}
};

class Derived : public Base {
    int y;
};

class Other: public Base {
    int z;
};

__device__ void foo() {
    Derived D1;
    Other D2;
}

__host__ void bar() {
    Other D3;
}
```

In this case, the implicitly declared constructor function `Derived::Derived()` will be treated as a `__device__` function because it is only invoked from the `__device__` function `foo()`. The implicitly declared constructor function `Other::Other()` will be treated as a `__host__ __device__` function since it is invoked both from both a `__device__` function `foo()` and a `__host__` function `bar()`.

Additionally, if F is an implicitly-declared `virtual` function (for example, a `virtual` destructor), the execution spaces of each virtual function D that is overridden by F are added to the set of execution spaces for F if D not implicitly-declared.

For example:

```
struct Base1 {
    virtual __host__ __device__ ~Base1() {}
};

struct Derived1 : Base1 {}; // implicitly-declared virtual destructor
                            // ~Derived1() has __host__ __device__  execution
↪space specifiers
```

(continues on next page)

(continued from previous page)

```
struct Base2 {
    virtual __device__ ~Base2() = default;
};

struct Derived2 : Base2 {}; // implicitly-declared virtual destructor
                            // ~Derived2() has __device__ execution space
↪specifiers
```

5.3.9.6.5 Polymorphic Classes

Polymorphic classes, namely those with `virtual` functions, derived from other polymorphic classes, or with polymorphic data members, are subject to the following restrictions:

- Copying polymorphic objects from device to host or from host to device, including `__global__` function arguments is undefined behavior.
- The execution space of an overridden `virtual` function must match the execution space of the function in the base class.

Example:

```
struct MyClass {
    virtual __host__ __device__ void f() {}
};

__global__ void kernel(MyClass my_class) {
    my_class.f(); // undefined behavior
}

int main() {
    MyClass my_class;
    kernel<<<1, 1>>>(my_class);
    cudaDeviceSynchronize();
}
```

See the example on Compiler Explorer.

```
struct BaseClass {
    virtual __host__ __device__ void f() {}
};

struct DerivedClass : BaseClass {
    __device__ void f() override {} // ERROR
};
```

See the example on Compiler Explorer.

5.3.9.6.6 Windows-Specific Class Layout

The CUDA compiler follows the IA64 ABI for class layout, while Microsoft Visual Studio does not. This prevents bitwise copy of special objects between host and device code as described below.

Let T denote a pointer to member type, or a class type that satisfies any of the following conditions:

- T is a *polymorphic class*
- T has multiple inheritance with more than one direct or indirect *empty base class*.
- All direct and indirect base classes B are *empty* and the type of the first field F of T uses B in its definition, such that B is laid out at offset 0 in the definition of F.

Classes of type T, with a base class of type T, or with data members of type T, may have a different class layout and size between host and device when compiled with Microsoft Visual Studio.

Copying such objects from device to host or from host to device, including `__global__` function arguments is undefined behavior.

5.3.9.7 Templates

A type cannot be used as template argument of a `__global__` function or a `__device__/__constant__` variable (C++14) if either:

- The type is defined within a `__host__` or `__host__ __device__` function scope.
- The type is unnamed, such as an anonymous struct or a lambda expression, unless the type is local to a `__device__` or `__global__` function.
- The type is a class member with `private` or `protected`, unless the class is local to a `__device__` or `__global__` function.
- The type is compounded from any of the types above.

Example:

```
template <typename T>
__global__ void kernel() {}

template <typename T>
__device__ int device_var; // C++14

struct {
    int v;
} unnamed_struct;

void host_function() {
    struct LocalStruct {};
//  kernel<LocalStruct><<<1, 1>>>(); // ERROR, LocalStruct is defined within a
↪host function
    int data = 4;
//  cudaMemcpyToSymbol(device_var<LocalStruct>, &data, sizeof(data)); // ERROR,
↪same as above

    auto lambda = [](){};
//  kernel<decltype(lambda)><<<1, 1>>>();         // ERROR, unnamed type
//  kernel<decltype(unnamed_struct)><<<1, 1>>>(); // ERROR, unnamed type
}

class MyClass {
private:
    struct PrivateStruct {};
public:
    static void launch() {
```

(continues on next page)

(continued from previous page)

```
//      kernel<PrivateStruct><<<1, 1>>>(); // ERROR, private type
    }
};
```

See the example on Compiler Explorer.

5.3.10. C++11 Restrictions

5.3.10.1 inline Namespaces

It is not allowed to define one of the following entities within an `inline` namespace when another entity of the same name and type signature is defined in an enclosing namespace:

- `__global__` function.
- `__device__`, `__constant__`, `__managed__`, `__shared__` variables.
- Variables with surface or texture type, such as `cudaSurfaceObject_t` or `cudaTextureObject_t`.

Example:

```
__device__ int my_var; // global scope

inline namespace NS {

__device__ int my_var; // namespace scope

} // namespace NS
```

5.3.10.2 inline Unnamed Namespaces

The following entities cannot be declared in namespace scope within an `inline` unnamed namespace:

- `__global__` function.
- `__device__`, `__constant__`, `__managed__`, `__shared__` variables.
- Variables with surface or texture type, such as `cudaSurfaceObject_t` or `cudaTextureObject_t`.

5.3.10.3 constexpr Functions

By default, a `constexpr` function cannot be called from a function with incompatible execution space, in the same way as standard functions.

- Calling a device-only `constexpr` function from a host-function during host code generation phase, namely when `__CUDA_ARCH__` macro is undefined. Example:

  ```
  constexpr __device__ int device_function() { return 0; }

  int main() {
      int x = device_function();  // ERROR, calling a device-only constexpr
  ↪function from host code
  }
  ```

- Calling a host-only constexpr function from a __device__ or __global__ function, during device code generation phase, namely when __CUDA_ARCH__ macro is defined. Example:

```
constexpr int host_function() { return 0; }

__device__ void device_function() {
    int x = host_function();  // ERROR, calling a host-only constexpr
↪function from device code
}
```

Note that a function template specialization may not be a constexpr function even if the corresponding template function is marked with the keyword constexpr.

Relaxed constexpr-Function Support

The experimental nvcc flag --expt-relaxed-constexpr can be used to relax this constraint for both __host__ and __device__ functions. However, a __global__ function cannot be declared as constexpr. nvcc will also define the macro __CUDACC_RELAXED_CONSTEXPR__.

When this flag is specified, the compiler will support cross execution space calls described above, as follows:

1. A call to a constexpr function in a cross-execution space is supported if it occurs in a context that requires constant evaluation, such as the initializer of a constexpr variable. Example:

```
constexpr __host__ int host_function(int x) { return x + 1; };

__global__ void kernel() {
    constexpr int val = host_function(1); // CORRECT, the call is in a
↪context that requires constant evaluation.
}

constexpr __device__ int device_function(int x) { return x + 1; }

int main() {
    constexpr int val = device_function(1); // CORRECT, the call is in a
↪context that requires constant evaluation.
}
```

2. Device code is generated during device code generation for the body of a host-only constexpr function, unless it is not used or is only called in a constexpr context. Example:

```
// NOTE: "host_function" is emitted in generated device code because
//       it is called from device code in a non-constexpr context
constexpr int host_function(int x) { return x + 1; }

__device__ int device_function(int in) {
    return host_function(in);  // CORRECT, even though argument is not a
↪constant expression
}
```

3. All code restrictions that apply to a device function also apply to the constexpr host-only function called from the device code. However, the compiler may not emit any build-time diagnostics for restrictions related to the compilation process.

   For example, the following code patterns are not supported in the body of the host function. This is similar to any device function; however, no compiler diagnostic may be generated.

- One-Definition Rule (ODR)-use of a host variable or host-only non-`constexpr` function. Example:

```
int host_var1, host_var2;

constexpr int* host_function(bool b) { return b ? &host_var1 : &host_
↪var2; };

__device__ int device_function(bool flag) {
    return *host_function(flag); // ERROR, host_function() attempts to
↪refer to the host variables
                                 //        'host_var1' and 'host_var2'.
                                 //        The code will compile, but
↪will NOT execute correctly.
}
```

- Use of exceptions `throw/catch` and Run-Time Type Information `typeid/dynamic_cast`. Example:

```
struct Base { };
struct Derived : public Base { };

// NOTE: "host_function" is emitted in generated device code
constexpr int host_function(bool b, Base *ptr) {
    if (b) {
        return 1;
    }
    else if (typeid(ptr) == typeid(Derived)) { // ERROR, use of
↪typeid in code executing on the GPU
        return 2;
    }
    else {
        throw int{4}; // ERROR, use of throw in code executing on the
↪GPU
    }
}

__device__ void device_function(bool flag) {
    Derived d;
    int val = host_function(flag, &d); //ERROR, host_function()
↪attempts use typeid and throw(),
                                       //       which are not allowed
↪in code that executes on the GPU
}
```

4. During host code generation, the body of a device-only `constexpr` function is preserved in the code sent to the host compiler. However, if the body of a device function attempts to ODR-use a namespace-scope device variable or a non-`constexpr` device function, the call to the device function from host code is not supported. While the code may build without compiler diagnostics, it may behave incorrectly at runtime. Example:

```
__device__ int host_var1, host_var2;

constexpr __device__ int* device_function(bool b) { return b ? &host_var1
```

(continues on next page)

(continued from previous page)

```
↪: &host_var2; };

int host_function(bool flag) {
    return *device_function(flag); // ERROR, device_function() attempts
↪to refer to device variables
                                   //        'host_var1' and 'host_var2'
                                   // The code will compile, but will NOT
↪execute correctly.
}
```

Warning

Due to the above restrictions and the lack of compiler diagnostics for incorrect usage, it is recommended to avoid calling a function in the Standard C++ headers `std::` from device code. The implementation of such functions varies depending on the host platform. Instead, it is strongly suggested to call the equivalent functionality in the CUDA C++ Standard Library libcu++, in the `cuda::std::` namespace.

5.3.10.4 constexpr Variables

By default, a `constexpr` variable cannot be used in a function with incompatible execution space, in the same way of standard variables.

A `constexpr` variable can be directly used in device code in the following cases:

- C++ scalar types, excluding pointer and pointer-to-member types:
  - `nullptr_t`.
  - `bool`.
  - Integral types: `char`, `signed char`, `unsigned`, `long long`, etc.
  - Floating point types: `float`, `double`.
  - Enumerators: `enum` and `enum class`.
- Class types: `class`, `struct`, and `union` with a `constexpr` constructor.
- Raw array of the types above, for example `int[]`, only when they are used inside a `constexpr` `__device__` or `__host__ __device__` function.

`constexpr __managed__` and `constexpr __shared__` variables are not allowed.

Examples:

```
constexpr int ConstexprVar = 4; // scalar type

struct MyStruct {
    static constexpr int ConstexprVar = 100;
};

constexpr MyStruct my_struct = MyStruct{}; // class type

constexpr int array[] = {1, 2, 3};
```

(continues on next page)

(continued from previous page)

```
__device__ constexpr int get_value(int idx) {
    return array[idx];                      // CORRECT
}

__device__ void foo(int idx) {
    int        v1 = ConstexprVar;           // CORRECT
    int        v2 = MyStruct::ConstexprVar; // CORRECT
//  const int &v3 = ConstexprVar1;         // ERROR, reference to host
↪constexpr variable
//  const int *v4 = &ConstexprVar1;        // ERROR, address of host constexpr
↪variable
    int        v5 = get_value(2);           // CORRECT, 'get_value(2)' is a
↪constant expression.
//  int        v6 = get_value(idx);        // ERROR, 'get_value(idx)' is not a
↪constant expression
//  int        v7 = array[2];              // ERROR, 'array' is not scalar
↪type.
    MyStruct   v8 = my_struct;              // CORRECT
}
```

See the example on Compiler Explorer.

5.3.10.5 `__global__` Variadic Template

A variadic `__global__` function template has the following restrictions:

- Only a single pack parameter is allowed.
- The pack parameter must be listed last in the template parameter list.

Examples:

```
template <typename... Pack>
__global__ void kernel1(); // CORRECT

// template <typename... Pack, template T>
// __global__ void kernel2(); // ERROR, parameter pack is not the last parameter

template <typename... TArgs>
struct MyStruct {};

// template <typename... Pack1, typename... Pack2>
// __global__ void kernel3(MyStruct<Pack1...>, MyStruct<Pack2...>); // ERROR,
↪more than one parameter pack
```

See the example on Compiler Explorer.

5.3.10.6 Defaulted Functions `= default`

The CUDA compiler infers the execution space of explicitly-defaulted member functions as described in *Implicitly-declared and explicitly-defaulted functions*.

Execution space specifiers on explicitly-defaulted functions are ignored by the compiler, except in the case the function is defined out-of-line or is a `virtual` function.

Examples:

```
struct MyStruct1 {
    MyStruct1() = default;
};

void host_function() {
    MyStruct1 my_struct; // __host__ __device__ constructor
}

__device__ void device_function() {
    MyStruct1 my_struct; // __host__ __device__ constructor
}

struct MyStruct2 {
    __device__ MyStruct2() = default; // WARNING: __device__ annotation is
↪ignored
};

struct MyStruct3 {
    __host__ MyStruct3();
};
MyStruct3::MyStruct3() = default; // out-of-line definition, not ignored

__device__ void device_function2() {
//  MyStruct3 my_struct; // ERROR, __host__ constructor
}

struct MyStruct4 {
    //  MyStruct4::~MyStruct4 has host execution space, not ignored because
↪virtual
    virtual __host__ ~MyStruct4() = default;
};

__device__ void device_function3() {
    MyStruct4 my_struct4;
    // implicit destructor call for 'my_struct4':
    //    ERROR: call from a __device__ function 'device_function3' to a
    //    __host__ function 'MyStruct4::~MyStruct4'
}
```

See the example on Compiler Explorer.

5.3.10.7 `[cuda::]std::initializer_list`

By default, the CUDA compiler implicitly considers the member functions of `[cuda::]std::initializer_list` to have `__host__ __device__` execution space specifiers, and therefore they can be invoked directly from device code.

The nvcc flag `--no-host-device-initializer-list` disables this behavior; member functions of `[cuda::]std::initializer_list` will then be considered as `__host__` functions and will not be directly invocable from device code.

A `__global__` function cannot have a parameter of type `[cuda::]std::initializer_list`.

Example:

```
#include <initializer_list>

__device__ void foo(std::initializer_list<int> in) {}

__device__ void bar() {
    foo({4,5,6}); // (a) initializer list containing only constant expressions.
    int i = 4;
    foo({i,5,6}); // (b) initializer list with at least one  non-constant
↪element.
                  // This form may have better performance than (a).
}
```

See the example on Compiler Explorer.

5.3.10.8 [cuda::]std::move, [cuda::]std::forward

By default, the CUDA compiler implicitly considers `std::move` and `std::forward` function templates to have `__host__ __device__` execution space specifiers, and therefore they can be invoked directly from device code. The nvcc flag `--no-host-device-move-forward` disables this behavior; `std::move` and `std::forward` will then be considered as `__host__` functions and will not be directly invocable from device code.

Hint

`cuda::std::move` and `cuda::std::forward` on the contrary always have `__host__ __device__` execution space.

5.3.11. C++14 Restrictions

5.3.11.1 Functions with Deduced Return Type

A `__global__` function cannot have a deduced return type `auto`.

Introspection of the return type of a `__device__` function with a deduced return type is not allowed in host code.

Note

The CUDA frontend compiler changes the function declaration to have a `void` return type, before invoking the host compiler. This may break introspection of the deduced return type of the `__device__` function in host code. Thus, the CUDA compiler will issue a compile-time error for referencing such a deduced return type outside of device function bodies.

Examples:

```
__device__ auto device_function(int x) { // deduced return type
    return x;                            // decltype(auto) has the same
↪behavior
}
```

(continues on next page)

(continued from previous page)

```
 __global__ void kernel() {
     int x = sizeof(device_function(2));         // CORRECT, device code scope
 }

 // const int size = sizeof(device_function(2)); // ERROR, return type
↪deduction on host

 void host_function() {
 //  using T = decltype(device_function(2));     // ERROR, return type
↪deduction on host
 }

void host_fn1() {
  // ERROR, referenced outside device function bodies
  int (*p1)(int) = fn1;

  struct S_local_t {
    // ERROR, referenced outside device function bodies
    decltype(fn2(10)) m1;

    S_local_t() : m1(10) { }
  };
}

// ERROR, referenced outside device function bodies
template <typename T = decltype(fn2)>
void host_fn2() { }

template<typename T> struct MyStruct { };

// ERROR, referenced outside device function bodies
struct S1_derived_t : MyStruct<decltype(fn1)> { };
```

5.3.11.2 Variable Templates

A `__device__` or `__constant__` variable template cannot be `const`-qualified when using the Microsoft compiler.

Examples:

```
// ERROR on Windows (non-portable), const-qualified
template <typename T>
__device__ const T var = 0;

 // CORRECT, ptr1 is not const-qualified
template <typename T>
__device__ const T* ptr1 = nullptr;

// ERROR on Windows (non-portable), ptr2 is const-qualified
template <typename T>
__device__ const T* const ptr2 = nullptr;
```

See the example on Compiler Explorer.

5.3.12. C++17 Restrictions

5.3.12.1 inline Variables

In a single translation unit, using an `inline` variable provides no additional functionality beyond a regular variable and does not provide any practical advantage.

nvcc allows `inline` variables with `__device__`, `__constant__`, or `__managed__` memory space only in *Separate Compilation* mode or for variables with internal linkage.

Note

When using `gcc/g++` host compiler, an `inline` variable declared with `__managed__` memory space specifier may not be visible to the debugger.

Examples:

```
inline        __device__ int device_var1;  // CORRECT, when compiled in
↪Separate Compilation mode (-rdc=true or -dc)
                                           // ERROR, when compiled in Whole
↪Program Compilation mode

static inline __device__ int device_var2;  // CORRECT, internal linkage

namespace {

inline __device__ int device_var3;         // CORRECT, internal linkage

inline __shared__ int shared_var;          // CORRECT, internal linkage

static inline __device__ int device_var4;  // CORRECT, internal linkage

inline __device__ int device_var5;         // CORRECT, internal linkage

} // namespace
```

See the example on Compiler Explorer.

5.3.12.2 Structured Binding

A structured binding cannot be declared with a memory space specifier, such as `__device__`, `__shared__`, `__constant__`, or `__managed__`.

Example:

```
struct S {
    int x, y;
};
// __device__ auto [a, b] = S{4, 5}; // ERROR
```

5.3.13. C++20 Restrictions

5.3.13.1 Three-way Comparison Operator

The three-way comparison operator (<=>) is supported in device code, but some uses implicitly rely on functionality from the C++ Standard Library, which is provided by the host implementation. Using those operators may require specifying the flag `--expt-relaxed-constexpr` to silence warnings, and the functionality requires the host implementation to satisfy the requirements of the device code.

Examples:

```
#include <compare> // std::strong_ordering implementation

struct S {
    int x, y;

    auto operator<=>(const S&) const = default; // (a)

    __host__ __device__ bool operator<=>(int rhs) const { return false; } //
↪(b)
};

__host__ __device__ bool host_device_function(S a, S b) {
    if (a <=> 1)  // CORRECT, calls a user-defined host-device overload (b)
        return true;
    return a < b; // CORRECT, call to an implicitly-declared function (a)
                  // Note: it requires a device-compatible std::strong_ordering
                  //       implementation provided in the header <compare>
                  //       and the flag --expt-relaxed-constexpr
}
```

See the example on Compiler Explorer.

5.3.13.2 `consteval` Functions

`consteval` functions can be called from both host and device code, independently of their execution space.

Examples:

```
consteval int host_consteval() {
    return 10;
}

__device__ consteval int device_consteval() {
    return 10;
}

__device__ int device_function() {
    return host_consteval();   // CORRECT, even if called from device code
}

__host__ __device__ int host_device_function() {
    return device_function();  // CORRECT, even if called from host-device code
}
```

5.4. C/C++ Language Extensions

5.4.1. Function and Variable Annotations

5.4.1.1 Execution Space Specifiers

The execution space specifiers `__host__`, `__device__`, and `__global__` indicate whether a function executes on the host or the device.

Table 38: Execution Space Specifier

Execution Space Specifier	Executed on		Callable from	
	Host	**Device**	**Host**	**Device**
`__host__`, no specifier	▯	▯	▯	▯
`__device__`	▯	▯	▯	▯
`__global__`	▯	▯	▯	▯
`__host__ __device__`	▯	▯	▯	▯

Constraints for `__global__` functions:

- Must return `void`.
- Cannot be a member of a `class`, `struct`, or `union`.
- Requires an execution configuration as described in *Kernel Configuration*.
- Does not support recursion.
- Refer to `__global__` *function parameters* for additional restrictions.

Calls to a `__global__` function are asynchronous. They return to the host thread before the device completes execution.

Functions declared with `__host__ __device__` are compiled for both the host and the device. The `__CUDA_ARCH__` *macro* can be used to differentiate host and device code paths:

```
__host__ __device__ void func() {
#if defined(__CUDA_ARCH__)
    // Device code path
#else
    // Host code path
#endif
}
```

5.4.1.2 Memory Space Specifiers

The memory space specifiers `__device__`, `__managed__`, `__constant__`, and `__shared__` indicate the storage location of a variable on the device.

The following table summarizes the memory space properties:

Table 39: Memory Space Specifier

Memory Space Specifier	Location	Accessible by	Lifetime	Unique instance
`__device__`	Device global memory	Device Threads (grid) / CUDA Runtime API	Program/*CUDA context*	Per device
`__constant__`	Device constant memory	Device Threads (grid) / CUDA Runtime API	Program/*CUDA context*	Per device
`__managed__`	Host and Device (automatic)	Host/Device Threads	Program	Per program
`__shared__`	Device (streaming multiprocessor)	Block Threads	Block	Block
no specifier	Device (registers)	Single Thread	Single Thread	Single Thread

- Both `__device__` and `__constant__` variables can be accessed from the host using the CUDA Runtime API functions `cudaGetSymbolAddress()`, `cudaGetSymbolSize()`, `cudaMemcpyToSymbol()`, and `cudaMemcpyFromSymbol()`.
- `__constant__` variables are read-only in device code and can only be modified from the host using the CUDA Runtime API.

The following example illustrates how to use these APIs:

```
__device__   float device_var       = 4.0f; // Variable in device memory
__constant__ float constant_mem_var = 4.0f; // Variable in constant memory
                                            // For readability, the following
↪example focuses on a device variable.
int main() {
    float* device_ptr;
    cudaGetSymbolAddress((void**) &device_ptr, device_var);        // Gets
↪address of device_var

    size_t symbol_size;
    cudaGetSymbolSize(&symbol_size, device_var);                   //
↪Retrieves the size of the symbol (4 bytes).

    float host_var;
    cudaMemcpyFromSymbol(&host_var, device_var, sizeof(host_var)); // Copies
↪from device to host.

    host_var = 3.0f;
    cudaMemcpyToSymbol(device_var, &host_var, sizeof(host_var));   // Copies
↪from host to device.
}
```

See the example on Compiler Explorer.

5.4.1.2.1 `__shared__` Memory

`__shared__` memory variables can have a static size, which is determined at compile time, or a dynamic size, which is determined at kernel launch time. See the *Kernel Configuration* section for details on specifying the shared memory size at run time.

Shared memory constraints:

- Variables with a dynamic size must be declared as an external array or as a pointer.
- Variables with a static size cannot be initialized in their declaration.

The following example illustrates how to declare and size `__shared__` variables:

```
extern __shared__ char dynamic_smem_pointer[];
// extern __shared__ char* dynamic_smem_pointer; alternative syntax

__global__ void kernel() { // or a __device__ function
    __shared__ int smem_var1[4];                    // static size
    auto smem_var2 = (int*) dynamic_smem_pointer; // dynamic size
}

int main() {
    size_t shared_memory_size = 16;
    kernel<<<1, 1, shared_memory_size>>>();
    cudaDeviceSynchronize();
}
```

See the example on Compiler Explorer.

5.4.1.2.2 `__managed__` Memory

`__managed__` variables have the following restrictions:

- The address of a `__managed__` variable is not a constant expression.
- A `__managed__` variable shall not have a reference type T&.
- The address or value of a `__managed__` variable shall not be used when the CUDA runtime may not be in a valid state, including the following cases:
  - In static/dynamic initialization or destruction of an object with `static` or `thread_local` storage duration.
  - In code that executes after `exit()` has been called. For example, a function marked with `__attribute__((destructor))`.
  - In code that executes when the CUDA runtime may not be initialized. For example, a function marked with `__attribute__((constructor))`.
- A `__managed__` variable cannot be used as an unparenthesized id-expression argument to a `decltype()` expression.
- `__managed__` variables have the same coherence and consistency behavior as specified for *dynamically allocated managed memory*.
- See also the restrictions for *local variables*.

Here are examples of legal and illegal uses of `__managed__` variables:

```
#include <cassert>

__device__ __managed__ int global_var = 10; // OK

int* ptr = &global_var;                    // ERROR: use of a managed
↪variable in static initialization

struct MyStruct1 {
    int field;
    MyStruct1() : field(global_var) {};
};

struct MyStruct2 {
    ~MyStruct2() { global_var = 10; }
};

MyStruct1 temp1; // ERROR: use of managed variable in dynamic initialization

MyStruct2 temp2; // ERROR: use of managed variable in the destructor of
                 //        object with static storage duration

__device__ __managed__ const int const_var = 10;         // ERROR: const-
↪qualified type

__device__ __managed__ int&      reference = global_var; // ERROR: reference
↪type

template <int* Addr>
struct MyStruct3 {};

MyStruct3<&global_var> temp;     // ERROR: address of managed variable is not
↪a constant expression

__global__ void kernel(int* ptr) {
    assert(ptr == &global_var);  // OK
    global_var = 20;             // OK
}

int main() {
    int* ptr = &global_var;      // OK
    kernel<<<1, 1>>>(ptr);
    cudaDeviceSynchronize();
    global_var++;                // OK
    decltype(global_var) var1;   // ERROR: managed variable used as
↪unparenthesized argument to decltype

    decltype((global_var)) var2; // OK
}
```

5.4.1.3 Inlining Specifiers

The following specifiers can be used to control inlining for `__host__` and `__device__` functions:

- `__noinline__`: Instructs `nvcc` not to inline the function.
- `__forceinline__`: Forces `nvcc` to inline the function within a single translation unit.
- `__inline_hint__`: Enables aggressive inlining across translation units when using *Link-Time Optimization*.

These specifiers are mutually exclusive.

5.4.1.4 `__restrict__` Pointers

`nvcc` supports restricted pointers via the `__restrict__` keyword.

Pointer aliasing occurs when two or more pointers refer to overlapping memory regions. This can inhibit optimizations such as code reordering and common sub-expression elimination.

A restrict-qualified pointer is a promise from the programmer that for the lifetime of the pointer, the memory it points to will only be accessed through that pointer. This allows the compiler to perform more aggressive optimizations.

- all threads that access the device function only read from it; or
- at most one thread writes to it, and no other thread reads from it.

The following example illustrates an aliasing issue and demonstrates how using a restricted pointer can help the compiler reduce the number of instructions:

```
__device__
void device_function(const float* a, const float* b, float* c) {
    c[0] = a[0] * b[0];
    c[1] = a[0] * b[0];
    c[2] = a[0] * b[0] * a[1];
    c[3] = a[0] * a[1];
    c[4] = a[0] * b[0];
    c[5] = b[0];
    ...
}
```

Because the pointers a, b, and c may be aliased, any write through c could modify elements of a or b. To guarantee functional correctness, the compiler cannot load `a[0]` and `b[0]` into registers, multiply them, and store the result in both `c[0]` and `c[1]`. This is because the results would differ from the abstract execution model if `a[0]` and `c[0]` were at the same location. The compiler cannot take advantage of the common sub-expression. Similarly, the compiler cannot reorder the computation of `c[4]` with the computations of `c[0]` and `c[1]` because a preceding write to `c[3]` could alter the inputs to the computation of `c[4]`.

By declaring a, b, and c as restricted pointers, the programmer informs the compiler that the pointers are not aliased. This means that writing to c will never overwrite the elements of a or b. This changes the function prototype as follows:

```
__device__
void device_function(const float* __restrict__ a, const float* __restrict__ b,
 ↪ float* __restrict__ c);
```

Note that all pointer arguments must be restricted for the compiler optimizer to be effective. With the addition of the `__restrict__` keywords, the compiler can reorder and perform common sub-expression elimination at will while maintaining identical functionality to the abstract execution model.

```
__device__
void device_function(const float* __restrict__ a, const float* __restrict__ b,
↪ float* __restrict__ c) {
    float t0 = a[0];
    float t1 = b[0];
    float t2 = t0 * t1;
    float t3 = a[1];
    c[0]     = t2;
    c[1]     = t2;
    c[4]     = t2;
    c[2]     = t2 * t3;
    c[3]     = t0 * t3;
    c[5]     = t1;
    ...
}
```

See the example on Compiler Explorer.

The result is a reduced number of memory accesses and computations, balanced by an increase in register pressure from caching loads and common sub-expressions in registers.

Since register pressure is a critical issue in many CUDA codes, the use of restricted pointers can negatively impact performance by reducing occupancy.

Accesses to `__global__` function `const` pointers marked with `__restrict__` are compiled as read-only cache loads, similar to the PTX `ld.global.nc` or `__ldg()` *low-level load and store functions* instructions.

```
__global__
void kernel1(const float* in, float* out) {
    *out = *in; // PTX: ld.global
}

__global__
void kernel2(const float* __restrict__ in, float* out) {
    *out = *in;  // PTX: ld.global.nc
}
```

See the example on Compiler Explorer.

5.4.1.5 `__grid_constant__` Parameters

Annotating a `__global__` function parameter with `__grid_constant__` prevents the compiler from creating a per-thread copy of the parameter. Instead, all threads in the grid will access the parameter through a single address, which can improve performance.

The `__grid_constant__` parameter has the following properties:

- It has the lifetime of the kernel.
- It is private to a single kernel, meaning the object is not accessible to threads from other grids, including sub-grids.

- All threads in the kernel see the same address.
- It is read-only. Modifying a `__grid_constant__` object or any of its sub-objects, including `mutable` members, is undefined behavior.

Requirements:

- Kernel parameters annotated with `__grid_constant__` must have `const`-qualified non-reference types.
- All function declarations must be consistent with any `__grid_constant__` parameters.
- Function template specializations must match the primary template declaration with respect to any `__grid_constant__` parameters.
- Function template instantiations must also match the primary template declaration with respect to any `__grid_constant__` parameters.

Examples:

```
struct MyStruct {
    int         x;
    mutable int y;
};

__device__ void external_function(const MyStruct&);

__global__ void kernel(const __grid_constant__ MyStruct s) {
    // s.x++; // Compile error: tried to modify read-only memory
    // s.y++; // Undefined Behavior: tried to modify read-only memory

    // Compiler will NOT create a per-thread local copy of "s":
    external_function(s);
}
```

See the example on Compiler Explorer.

5.4.1.6 Annotation Summary

The following table summarizes the CUDA annotations and reports which execution space each annotation applies to and where it is valid.

Table 40: Annotation Summary

Annotation	`__host__` / `__device__` / `__host__ __device__`	`__global__`
__noinline__, *__forceinline__*, *__inline_hint__*	Function	▯
__restrict__	Pointer Parameter	Pointer Parameter
__grid_constant__	▯	Parameter
__launch_bounds__	▯	Function
__maxnreg__	▯	Function
__cluster_dims__	▯	Function

5.4.2. Built-in Types and Variables

5.4.2.1 Host Compiler Type Extensions

The use of non-standard arithmetic types is permitted by CUDA, as long as the host compiler supports it. The following types are supported:

- 128-bit integer type `__int128`.
  - Supported on Linux when the host compiler defines the `__SIZEOF_INT128__` macro.
- 128-bit floating-point types `__float128` and `_Float128` are available on GPU devices with compute capability 10.0 and later. A constant expression of `__float128` type may be processed by the compiler in a floating-point representation with lower precision.
  - Supported on Linux x86 when the host compiler defines the `__SIZEOF_FLOAT128__` or `__FLOAT128__` macros.
- `_Complex` types are only supported in host code.

5.4.2.2 Built-in Variables

The values used to specify and retrieve the kernel configuration for the grid and blocks along the x, y, and z dimensions are of type `dim3`. The variables used to obtain the block and thread indices are of type `uint3`. Both `dim3` and `uint3` are trivial structures consisting of three unsigned values named `x`, `y`, and `z`. In C++11 and later, the default value of all components of `dim3` is 1.

Built-in device-only variables:

- `dim3 gridDim`: contains the dimensions of the grid, namely the number of thread blocks, along the x, y, and z dimensions.
- `dim3 blockDim`: contains the dimensions of the thread block, namely the number of threads, along the x, y, and z dimensions.
- `uint3 blockIdx`: contains the block index within the grid, along the x, y, and z dimensions.
- `uint3 threadIdx`: contains the thread index within the block, along the x, y, and z dimensions.
- `int warpSize` : A run-time value defined as the number of threads in a warp, commonly 32. See also *Warps and SIMT* for the definition of a warp.

5.4.2.3 Built-in Types

CUDA provides vector types derived from basic integer and floating-point types that are supported for both the host and the device. The following table shows the available vector types.

Table 41: Vector Types

C++ Fundamental Type	Vector X1	Vector X2	Vector X3	Vector X4
`signed char`	`char1`	`char2`	`char3`	`char4`
`unsigned char`	`uchar1`	`uchar2`	`uchar3`	`uchar4`
`signed short`	`short1`	`short2`	`short3`	`short4`
`unsigned short`	`ushort1`	`ushort2`	`ushort3`	`ushort4`
`signed int`	`int1`	`int2`	`int3`	`int4`
`unsigned`	`uint1`	`uint2`	`uint3`	`uint4`
`signed long`	`long1`	`long2`	`long3`	`long4_16a/ long4_32a`
`unsigned long`	`ulong1`	`ulong2`	`ulong3`	`ulong4_16a/ ulong4_32a`
`signed long long`	`longlong1`	`longlong2`	`longlong3`	`longlong4_16a/ longlong4_32a`
`unsigned long long`	`ulonglong1`	`ulonglong2`	`ulonglong3`	`ulonglong4_16a/ ulonglong4_32a`
`float`	`float1`	`float2`	`float3`	`float4`
`double`	`double1`	`double2`	`double3`	`double4_16a/ double4_32a`

Note that `long4`, `ulong4`, `longlong4`, `ulonglong4`, and `double4` have been deprecated in CUDA 13, and may be removed in a future release.

The following table details the byte size and alignment requirements of the vector types:

Table 42: Alignment Requirements

Type	Size	Alignment
`char1`, `uchar1`	1	1
`char2`, `uchar2`	2	2
`char3`, `uchar3`	3	1
`char4`, `uchar4`	4	4
`short1`, `ushort1`	2	2

continues on next page

Table 42 – continued from previous page

Type	Size	Alignment
`short2`, `ushort2`	4	4
`short3`, `ushort3`	6	2
`short4`, `ushort4`	8	8
`int1`, `uint1`	4	4
`int2`, `uint2`	8	8
`int3`, `uint3`	12	4
`int4`, `uint4`	16	16
`long1`, `ulong1`	4/8 *****	4/8 *****
`long2`, `ulong2`	8/16 *****	8/16 *****
`long3`, `ulong3`	12/24 *****	4/8 *****
`long4`, `ulong4` (deprecated)	16/32 *****	16 *****
`long4_16a`, `ulong4_16a`	16/32 *****	16
`long4_32a`, `ulong4_32a`	16/32 *****	32
`longlong1`, `ulonglong1`	8	8
`longlong2`, `ulonglong2`	16	16
`longlong3`, `ulonglong3`	24	8
`longlong4`, `ulonglong4` (deprecated)	32	16
`longlong4_16a`, `ulonglong4_16a`	32	16
`longlong4_32a`, `ulonglong4_32a`	32	32
`float1`	4	4
`float2`	8	8
`float3`	12	4
`float4`	16	16
`double1`	8	8
`double2`	16	16
`double3`	24	8
`double4` (deprecated)	32	16
`double4_16a`	32	16
`double4_32a`	32	32

***** `long` is 4 bytes on C++ LLP64 data model (Windows 64-bit), while it is 8 bytes on C++ LP64 data model (Linux 64-bit).

Vector types are structures. Their first, second, third, and fourth components are accessible through the x, y, z, and w fields, respectively.

```
int sum(int4 value) {
    return value.x + value.y + value.z + value.w;
}
```

They all have a factory function of the form `make_<type_name>()`; for example:

```
int4 add_one(int x, int y, int z, int w) {
    return make_int4(x + 1, y + 1, z + 1, w + 1);
}
```

If host code is not compiled with nvcc, the vector types and related functions can be imported by including the `cuda_runtime.h` header provided in the CUDA toolkit.

5.4.3. Kernel Configuration

Any call to a `__global__` function must specify an *execution configuration* for that call. This execution configuration defines the dimensions of the grid and blocks that will be used to execute the function on the device, as well as the associated *stream*.

The execution configuration is specified by inserting an expression in the form `<<<grid_dim, block_dim, dynamic_smem_bytes, stream>>>` between the function name and the parenthesized argument list, where:

- `grid_dim` is of type *dim3* and specifies the dimension and size of the grid, such that `grid_dim.x * grid_dim.y * grid_dim.z` equals the number of blocks being launched;
- `block_dim` is of type *dim3* and specifies the dimension and size of each block, such that `block_dim.x * block_dim.y * block_dim.z` equals the number of threads per block;
- `dynamic_smem_bytes` is an optional `size_t` argument that defaults to zero. It specifies the number of bytes in shared memory that are dynamically allocated per block for this call in addition to the statically allocated memory. This memory is used by `extern __shared__` arrays (see *__shared__ Memory*).
- `stream` is of type `cudaStream_t` (pointer) and specifies the associated stream. `stream` is an optional argument that defaults to NULL.

The following example shows a kernel function declaration and call:

```
__global__ void kernel(float* parameter);

kernel<<<grid_dim, block_dim, dynamic_smem_bytes>>>(parameter);
```

The arguments for the execution configuration are evaluated before the arguments for the actual function.

The function call fails if `grid_dim` or `block_dim` exceeds the maximum sizes allowed for the device, as specified in *Compute Capabilities*, or if `dynamic_smem_bytes` is greater than the available shared memory after accounting for statically allocated memory.

5.4.3.1 Thread Block Cluster

Compute capability 9.0 and higher allow users to specify compile-time thread block cluster dimensions so that the kernels can use the *cluster hierarchy* in CUDA. The compile-time cluster dimension can be specified using the `__cluster_dims__` attribute with the following syntax: `__cluster_dims__([x, [y, [z]]])`. The example below shows a compile-time cluster size of 2 in the X dimension and 1 in the Y and Z dimensions.

```
__global__ void __cluster_dims__(2, 1, 1) kernel(float* parameter);
```

The default form of `__cluster_dims__()` specifies that a kernel is to be launched as a grid cluster. If a cluster dimension is not specified, the user can specify it at launch time. Failing to specify a dimension at launch time will result in a launch-time error.

The dimensions of the thread block cluster can also be specified at runtime, and the kernel with the cluster can be launched using the `cudaLaunchKernelEx` API. This API takes a configuration argument of type `cudaLaunchConfig_t`, a kernel function pointer, and kernel arguments. The example below shows runtime kernel configuration.

```
__global__ void kernel(float parameter1, int parameter2) {}

int main() {
    cudaLaunchConfig_t config = {0};
    // The grid dimension is not affected by cluster launch, and is still
↪enumerated
    // using the number of blocks.
    // The grid dimension should be a multiple of cluster size.
    config.gridDim          = dim3{4};  // 4 blocks
    config.blockDim         = dim3{32}; // 32 threads per block
    config.dynamicSmemBytes = 1024;     // 1 KB

    cudaLaunchAttribute attribute[1];
    attribute[0].id               = cudaLaunchAttributeClusterDimension;
    attribute[0].val.clusterDim.x = 2; // Cluster size in X-dimension
    attribute[0].val.clusterDim.y = 1;
    attribute[0].val.clusterDim.z = 1;
    config.attrs    = attribute;
    config.numAttrs = 1;

    float parameter1 = 3.0f;
    int   parameter2 = 4;
    cudaLaunchKernelEx(&config, kernel, parameter1, parameter2);
}
```

See the example on Compiler Explorer.

5.4.3.2 Launch Bounds

As discussed in the *Kernel Launch and Occupancy* section, using fewer registers allows more threads and thread blocks to reside on a multiprocessor, which improves performance.

Therefore, the compiler uses heuristics to minimize register usage while keeping *register spilling* and instruction count to a minimum. Applications can optionally aid these heuristics by providing additional information to the compiler in the form of launch bounds that are specified using the `__launch_bounds__()` qualifier in the definition of a `__global__` function:

```
__global__ void
__launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor,
↪maxBlocksPerCluster)
MyKernel(...) {
    ...
}
```

- maxThreadsPerBlock specifies the maximum number of threads per block with which the application will ever launch MyKernel(); it compiles to the .maxntid PTX directive.
- minBlocksPerMultiprocessor is optional and specifies the desired minimum number of resident blocks per multiprocessor; it compiles to the .minnctapersm PTX directive.
- maxBlocksPerCluster is optional and specifies the desired maximum number of thread blocks per cluster with which the application will ever launch MyKernel(); it compiles to the .maxclusterrank PTX directive.

If launch bounds are specified, the compiler first derives the upper limit, L, on the number of registers that the kernel should use. This ensures that minBlocksPerMultiprocessor blocks (or a single block, if minBlocksPerMultiprocessor is not specified) of maxThreadsPerBlock threads can reside on the multiprocessor. See the *occupancy* section for the relationship between the number of registers used by a kernel and the number of registers allocated per block. The compiler then optimizes register usage as follows:

- If the initial register usage exceeds L, the compiler reduces it until it is less than or equal to L. This usually results in increased local memory usage and/or a higher number of instructions.
- If the initial register usage is lower than L
  - If maxThreadsPerBlock is specified but minBlocksPerMultiprocessor is not, the compiler uses maxThreadsPerBlock to determine the register usage thresholds for the transitions between n and n + 1 resident blocks. This occurs when using one less register makes room for an additional resident block. Then, the compiler applies similar heuristics as when no launch bounds are specified.
  - If both minBlocksPerMultiprocessor and maxThreadsPerBlock are specified, the compiler may increase register usage up to L in order to reduce the number of instructions and better hide the latency of single-threaded instructions.

A kernel will fail to launch if it is executed with:

- more threads per block than its launch bound maxThreadsPerBlock.
- more thread blocks per cluster than its launch bound maxBlocksPerCluster.

The per-thread resources required by a CUDA kernel may limit the maximum block size in an undesirable way. To maintain forward compatibility with future hardware and toolkits, and to ensure that at least one thread block can run on a streaming multiprocessor, developers should include the single argument __launch_bounds__(maxThreadsPerBlock) which specifies the largest block size with which the kernel will launch. Failure to do so could result in "too many resources requested for launch" errors. Providing the two-argument version of __launch_bounds__(maxThreadsPerBlock, minBlocksPerMultiprocessor) can improve performance in some cases. The best value for minBlocksPerMultiprocessor should be determined through a detailed analysis of each kernel.

The optimal launch bounds for a kernel typically differ across major architecture revisions. The following code sample illustrates how this is managed in device code with the __CUDA_ARCH__ *macro*.

```
#define THREADS_PER_BLOCK  256

#if __CUDA_ARCH__ >= 900
    #define MY_KERNEL_MAX_THREADS  (2 * THREADS_PER_BLOCK)
    #define MY_KERNEL_MIN_BLOCKS   3
#else
    #define MY_KERNEL_MAX_THREADS  THREADS_PER_BLOCK
    #define MY_KERNEL_MIN_BLOCKS   2
#endif
```

(continues on next page)

(continued from previous page)

```
__global__ void
__launch_bounds__(MY_KERNEL_MAX_THREADS, MY_KERNEL_MIN_BLOCKS)
MyKernel(...) {
    ...
}
```

When `MyKernel` is invoked with the maximum number of threads per block, which is specified as the first parameter of `__launch_bounds__()`, it is tempting to use `MY_KERNEL_MAX_THREADS` as the number of threads per block in the execution configuration:

```
// Host code
MyKernel<<<blocksPerGrid, MY_KERNEL_MAX_THREADS>>>(...);
```

However, this will not work, since `__CUDA_ARCH__` is undefined in host code as mentioned in the *Execution Space Specifiers* section. Therefore, `MyKernel` will launch with 256 threads per block. The number of threads per block should instead be determined:

- Either at compile time using a macro or constant that does not depend on `__CUDA_ARCH__`, for example

  ```
  // Host code
  MyKernel<<<blocksPerGrid, THREADS_PER_BLOCK>>>(...);
  ```

- Or at runtime based on the compute capability

  ```
  // Host code
  cudaGetDeviceProperties(&deviceProp, device);
  int threadsPerBlock = (deviceProp.major >= 9) ? 2 * THREADS_PER_BLOCK :
   ↪THREADS_PER_BLOCK;
  MyKernel<<<blocksPerGrid, threadsPerBlock>>>(...);
  ```

The `--resource-usage` compiler option reports register usage. The CUDA profiler reports occupancy, which can be used to derive the number of resident blocks.

5.4.3.3 Maximum Number of Registers per Thread

To enable low-level performance tuning, CUDA C++ offers the `__maxnreg__()` function qualifier, which passes performance tuning information to the backend optimizing compiler. The `__maxnreg__()` qualifier specifies the maximum number of registers that can be allocated to a single thread in a thread block. In the definition of a `__global__` function:

```
__global__ void
__maxnreg__(maxNumberRegistersPerThread)
MyKernel(...) {
    ...
}
```

The `maxNumberRegistersPerThread` variable specifies the maximum number of registers to be allocated to a single thread in a thread block of the kernel `MyKernel()`; it compiles to the `.maxnreg` PTX directive.

The `__launch_bounds__()` and `__maxnreg__()` qualifiers cannot be applied to the same kernel together.

The --maxrregcount <N> compiler option can be used to control register usage for all __global__ functions in a file. This option is ignored for kernel functions with the __maxnreg__ qualifier.

5.4.4. Synchronization Primitives

5.4.4.1 Thread Block Synchronization Functions

```
void __syncthreads();
int  __syncthreads_count(int predicate);
int  __syncthreads_and(int predicate);
int  __syncthreads_or(int predicate);
```

The intrinsics coordinate communication among threads within the same block. When threads in a block access the same addresses in shared or global memory, read-after-write, write-after-read, or write-after-write hazards can occur. These hazards can be avoided by synchronizing threads between such accesses.

The intrinsics have the following semantics:

- __syncthreads*() wait until all non-exited threads in the thread block simultaneously reach the same __syncthreads*() intrinsic call in the program or exit.
- __syncthreads*() provide memory ordering among participating threads: the call to __syncthreads*() intrinsics strongly happens before (see C++ specification [intro.races]) any participating thread is unblocked from the wait or exits.

The following example shows how to use __syncthreads() to synchronize threads within a thread block and safely sum the elements of an array shared among the threads:

```
// assuming blockDim.x is 128
__global__ void example_syncthreads(int* input_data, int* output_data) {
    __shared__ int shared_data[128];
    // Every thread writes to a distinct element of 'shared_data':
    shared_data[threadIdx.x] = input_data[threadIdx.x];

    // All threads synchronize, guaranteeing all writes to 'shared_data' are
↪ordered
    // before any thread is unblocked from '__syncthreads()':
    __syncthreads();

    // A single thread safely reads 'shared_data':
    if (threadIdx.x == 0) {
        int sum = 0;
        for (int i = 0; i < blockDim.x; ++i) {
            sum += shared_data[i];
        }
        output_data[blockIdx.x] = sum;
    }
}
```

The __syncthreads*() intrinsics are permitted in conditional code, but only if the condition evaluates uniformly across the entire thread block. Otherwise, execution may hang or produce unintended side effects.

The following example demonstrates a valid behavior:

```
// assuming blockDim.x is 128
__global__ void syncthreads_valid_behavior(int* input_data, int* output_data)
↪{
    __shared__ int shared_data[128];
    shared_data[threadIdx.x] = input_data[threadIdx.x];
    if (blockIdx.x > 0) { // CORRECT, uniform condition across all block
↪threads
        __syncthreads();
        output_data[threadIdx.x] = shared_data[128 - threadIdx.x];
    }
}
```

while the following examples exhibit invalid behavior, such as kernel hang, or undefined behavior:

```
// assuming blockDim.x is 128
__global__ void syncthreads_invalid_behavior1(int* input_data, int* output_
↪data) {
    __shared__ int shared_data[256];
    shared_data[threadIdx.x] = input_data[threadIdx.x];
    if (threadIdx.x > 0) { // WRONG, non-uniform condition
        __syncthreads();   // Undefined Behavior
        output_data[threadIdx.x] = shared_data[128 - threadIdx.x];
    }
}
```

```
// assuming blockDim.x is 128
__global__ void syncthreads_invalid_behavior2(int* input_data, int* output_
↪data) {
    __shared__ int shared_data[256];
    shared_data[threadIdx.x] = input_data[threadIdx.x];
    for (int i = 0; i < blockDim.x; ++i) {
        if (i == threadIdx.x) { // WRONG, non-uniform condition
            __syncthreads();    // Undefined Behavior
        }
    }
    output_data[threadIdx.x] = shared_data[128 - threadIdx.x];
}
```

`__syncthreads()` **variants with predicate**:

```
int __syncthreads_count(int predicate);
```

is identical to `__syncthreads()` except that it evaluates a predicate for all non-exited threads in the block and returns the number of threads for which the predicate evaluates to a non-zero value.

```
int __syncthreads_and(int predicate);
```

is identical to `__syncthreads()` except that it evaluates the predicate for all non-exited threads in the block. It returns a non-zero value if and only if the predicate evaluates to a non-zero value for all of them.

```
int __syncthreads_or(int predicate);
```

is identical to `__syncthreads()` except that it evaluates the predicate for all non-exited threads in the block. It returns a non-zero value if and only if the predicate evaluates to a non-zero value one or more of them.

5.4.4.2 Warp Synchronization Function

```
void __syncwarp(unsigned mask = 0xFFFFFFFF);
```

The intrinsic function `__syncwarp()` coordinates communication between the threads within the same warp. When some threads within a warp access the same addresses in shared or global memory, potential read-after-write, write-after-read, or write-after-write hazards may occur. These data hazards can be avoided by synchronizing the threads between these accesses.

Calling `__syncwarp(mask)` provides memory ordering among the participating threads within a warp named in `mask`: the call to `__syncwarp(mask)` strongly happens before (see C++ specification [intro.races]) any warp thread named in `mask` is unblocked from the wait or exits.

The functions are subject to the *Warp __sync Intrinsic Constraints*.

The following example demonstrates how to use `__syncwarp()` to synchronize threads within a warp to safely access a shared memory array:

```
__global__ void example_syncwarp(int* input_data, int* output_data) {
    if (threadIdx.x < warpSize) {
        __shared__ int shared_data[warpSize];
        shared_data[threadIdx.x] = input_data[threadIdx.x];

        __syncwarp(); // equivalent to __syncwarp(0xFFFFFFFF)
        if (threadIdx.x == 0)
            output_data[0] = shared_data[1];
    }
}
```

5.4.4.3 Memory Fence Functions

The CUDA programming model assumes a weakly ordered memory model. In other words, the order in which a CUDA thread writes data to shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the order in which another CUDA or host thread observes the data being written. Reading from or writing to the same memory location without memory fences or synchronization results in undefined behavior.

In the following example, thread 1 executes `writeXY()`, while thread 2 executes `readXY()`.

```
__device__ int X = 1, Y = 2;

__device__ void writeXY() {
    X = 10;
    Y = 20;
}

__device__ void readXY() {
    int B = Y;
```

(continues on next page)

(continued from previous page)

```
    int A = X;
}
```

The two threads simultaneously read and write to the same memory locations, X and Y. Any data race results in undefined behavior and has no defined semantics. Therefore, the resulting values for A and B can be anything.

Memory fence and synchronization functions enforce a sequentially consistent ordering of memory accesses. These functions differ in the thread scope in which orderings are enforced, but are independent of the accessed memory space, including shared memory, global memory, page-locked host memory, and the memory of a peer device.

Hint

It is suggested to use `cuda::atomic_thread_fence` provided by libcu++ whenever possible for safety and portability reasons.

Block-level memory fence

CUDA C++

```
// <cuda/atomic> header
cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_
↪block);
```

ensures that:

- All writes to all memory made by the calling thread before the call to `cuda::atomic_thread_fence()` are observed by all threads in the calling thread's block as occurring before all writes to all memory made by the calling thread after the call to `cuda::atomic_thread_fence()`;
- All reads from all memory made by the calling thread before the call to `cuda::atomic_thread_fence()` are ordered before all reads from all memory made by the calling thread after the call to `cuda::atomic_thread_fence()`.

Intrinsics

```
void __threadfence_block();
```

ensures that:

- All writes to all memory made by the calling thread before the call to `__threadfence_block()` are observed by all threads in the calling thread's block as occurring before all writes to all memory made by the calling thread after the call to `__threadfence_block()`;
- All reads from all memory made by the calling thread before the call to `__threadfence_block()` are ordered before all reads from all memory made by the calling thread after the call to `__threadfence_block()`.

Device-level memory fence

CUDA C++

```
cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_
→device);
```

ensures that:

- No writes to all memory made by the calling thread after the call to `cuda::atomic_thread_fence()` are observed by any thread in the device as occurring before any write to all memory made by the calling thread before the call to `cuda::atomic_thread_fence()`.

Intrinsics

```
void __threadfence();
```

ensures that:

- No writes to all memory made by the calling thread after the call to `__threadfence()` are observed by any thread in the device as occurring before any write to all memory made by the calling thread before the call to `__threadfence()`.

System-level memory fence

CUDA C++

```
cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_
→system);
```

ensures that:

- All writes to all memory made by the calling thread before the call to `cuda::atomic_thread_fence()` are observed by all threads in the device, host threads, and all threads in peer devices as occurring before all writes to all memory made by the calling thread after the call to `cuda::atomic_thread_fence()`.

Intrinsics

```
void __threadfence_system();
```

ensures that:

- All writes to all memory made by the calling thread before the call to `__threadfence_system()` are observed by all threads in the device, host threads, and all threads in peer devices as occurring before all writes to all memory made by the calling thread after the call to `__threadfence_system()`.

In the previous code sample, we can insert memory fences in the code as follows:

CUDA C++

```
#include <cuda/atomic>

__device__ int X = 1, Y = 2;

```

(continues on next page)

(continued from previous page)

```
__device__ void writeXY() {
    X = 10;
    cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_
→device);
    Y = 20;
}

__device__ void readXY() {
    int B = Y;
    cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_scope_
→device);
    int A = X;
}
```

Intrinsics

```
__device__ int X = 1, Y = 2;

__device__ void writeXY() {
    X = 10;
    __threadfence();
    Y = 20;
}

__device__ void readXY() {
    int B = Y;
    __threadfence();
    int A = X;
}
```

For this code, the following outcomes can be observed:

- A equal to 1 and B equal to 2, namely `readXY()` is executed before `writeXY()`,
- A equal to 10 and B equal to 20, namely `writeXY()` is executed before `readXY()`.
- A equal to 10 and B equal to 2.
- The case where A is 1 and B is 20 is not possible, as the memory fence ensures that the write to X is visible before the write to Y.

If threads 1 and 2 belong to the same block, it is enough to use a block-level fence. If threads 1 and 2 do not belong to the same block, a device-level fence must be used if they are CUDA threads from the same device, and a system-level fence must be used if they are CUDA threads from two different devices.

A common use case is illustrated by the following code sample, where threads consume data produced by other threads. This kernel computes the sum of an array of N numbers in a single call.

- Each block first sums a subset of the array and stores the result in global memory.
- When all the blocks have finished, the last block reads each of these partial sums from global memory and adds them together to obtain the final result.
- To determine which block finished last, each block atomically increments a counter to signal completion of computing and storing its partial sum (see the *Atomic Functions* section for further details). The last block receives a counter value equal to `gridDim.x - 1`.

Without a fence between storing the partial sum and incrementing the counter, the counter may increment before the partial sum is stored. This could cause the counter to reach `gridDim.x - 1` and allow the last block to start reading partial sums before they are updated in memory.

Note

The memory fence only affects the order in which memory operations are executed; it does not guarantee visibility of these operations to other threads.

In the code sample below, the visibility of the memory operations on the `result` variable is ensured by declaring it as `volatile`. For more details, see the `volatile`-*qualified variables* section.

```
#include <cuda/atomic>

__device__ int count = 0;

__global__ void sum(const float*    array,
                    int             N,
                    volatile float* result) {
    __shared__ bool isLastBlockDone;
    // Each block sums a subset of the input array.
    float partialSum = calculatePartialSum(array, N);

    if (threadIdx.x == 0) {
        // Thread 0 of each block stores the partial sum to global memory.
        // The compiler will use a store operation that bypasses the L1 cache
        // since the "result" variable is declared as volatile.
        // This ensures that the threads of the last block will read the
↪correct
        // partial sums computed by all other blocks.
        result[blockIdx.x] = partialSum;

        // Thread 0 makes sure that the increment of the "count" variable is
        // only performed after the partial sum has been written to global
↪memory.
        cuda::atomic_thread_fence(cuda::memory_order_seq_cst, cuda::thread_
↪scope_device);

        // Thread 0 signals that it is done.
        int count_old = atomicInc(&count, gridDim.x);

        // Thread 0 determines if its block is the last block to be done.
        isLastBlockDone = (count_old == (gridDim.x - 1));
    }
    // Synchronize to make sure that each thread reads the correct value of
    // isLastBlockDone.
    __syncthreads();

    if (isLastBlockDone) {
        // The last block sums the partial sums stored in result[0 .. gridDim.
↪x-1]
        float totalSum = calculateTotalSum(result);
```

(continues on next page)

(continued from previous page)

```
        if (threadIdx.x == 0) {
            // Thread 0 of last block stores the total sum to global memory and
            // resets the count variable, so that the next kernel call works
            // properly.
            result[0] = totalSum;
            count     = 0;
        }
    }
}
```

5.4.5. Atomic Functions

Atomic functions perform read-modify-write operations on shared data, making them appear to execute in a single step. Atomicity ensures that each operation either completes fully or not at all, providing all participating threads with a consistent view of the data.

CUDA provides atomic functions in four ways:

Extended CUDA C++ atomic functions, cuda::atomic and cuda::atomic_ref.

- They are allowed in both host and device code.
- They follow the C++ standard atomic operations semantics.
- They allow specifying the thread scope of the atomic operations.

Standard C++ atomic functions, cuda::std::atomic and cuda::std::atomic_ref.

- They are allowed in both host and device code.
- They follow the C++ standard atomic operations semantics.
- They do not allow specifying the thread scope of the atomic operations.

Compiler *built-in atomic functions*, `__nv_atomic_<op>()`.

- They have been available since CUDA 12.8.
- They are only allowed in device code.
- They follow the C++ standard atomic memory order semantics.
- They allow specifying the thread scope of the atomic operations.
- They have the same memory ordering semantics as C++ standard atomic operations.
- They support a subset of the data types allowed by cuda::std::atomic and cuda::std::atomic_ref, except for 128-bit data types.

***Legacy atomic functions*, `atomic<Op>()`.**

- They are only allowed in device code.
- They only support `memory_order_relaxed` C++ atomic memory semantics.

- They allow specifying the thread scope of the atomic operations as part of the function name.
- Unlike *built-in atomic functions*, legacy atomic functions only ensure atomicity and do not introduce synchronization points (fences).
- They support a subset of the data types allowed by *built-in atomic functions*. The atomic add operation supports additional data types.

Hint

Using the Extended CUDA C++ atomic functions provided by `libcu++` is recommended for efficiency, safety, and portability.

5.4.5.1 Legacy Atomic Functions

Legacy atomic functions perform atomic read-modify-write operations on a 32-, 64-, or 128-bit word stored in global or shared memory. For example, the `atomicAdd()` function reads a word at a specific address in global or shared memory, adds a number to it, and writes the result back to the same address.

- Atomic functions can only be used in device functions.
- For vector types such as `__half2`, `__nv_bfloat162`, `float2`, and `float4`, the read-modify-write operation is performed on each element of the vector. The entire vector is not guaranteed to be atomic in a single access.

The atomic functions described in this section have a memory ordering of `cuda::std::memory_order_relaxed` and are only atomic at a particular thread scope:

- Atomic APIs without a suffix, for example `atomicAdd`, are atomic at scope `cuda::thread_scope_device`.
- Atomic APIs with the `_block` suffix, for example, `atomicAdd_block`, are atomic at scope `cuda::thread_scope_block`.
- Atomic APIs with the `_system` suffix, for example, `atomicAdd_system`, are atomic at scope `cuda::thread_scope_system` if they meet particular conditions.

The following example shows the CPU and GPU atomically updating an integer value at address `addr`:

```
#include <cuda_runtime.h>

__global__ void atomicAdd_kernel(int* addr) {
    atomicAdd_system(addr, 10);
}

void test_atomicAdd(int device_id) {
    int* addr;
    cudaMallocManaged(&addr, 4);
    *addr = 0;

    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device_id);
    if (deviceProp.concurrentManagedAccess != 1) {
        return; // the device does not coherently access managed memory
↪concurrently with the CPU
```

(continues on next page)

(continued from previous page)

```
    }

    atomicAdd_kernel<<<...>>>(addr);
    __sync_fetch_and_add(addr, 10);  // CPU atomic operation
}
```

Note that any atomic operation can be implemented based on `atomicCAS()` (Compare and Swap). For example, `atomicAdd()` for single-precision floating-point numbers can be implemented as follows:

```
#include <cuda/memory>
#include <cuda/std/bit>

__device__ float customAtomicAdd(float* d_ptr, float value) {
    unsigned* d_ptr_unsigned = reinterpret_cast<unsigned*>(d_ptr);
    unsigned  old_value      = *d_ptr_unsigned;
    unsigned  assumed;
    do {
        assumed                            = old_value;
        float    assumed_float             = cuda::std::bit_cast<float>
↪(assumed);
        float    expected_value            = assumed_float + value;
        unsigned expected_value_unsigned = cuda::std::bit_cast<unsigned>
↪(expected_value);
        old_value                          = atomicCAS(d_ptr_unsigned, assumed,
↪expected_value_unsigned);
    // Note: uses integer comparison to avoid hang in case of NaN (since NaN !
↪= NaN)
    } while (assumed != old_value);
    return cuda::std::bit_cast<float>(old_value);
}
```

See the example on Compiler Explorer.

5.4.5.1.1 `atomicAdd()`

```
T atomicAdd(T* address, T val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old + val`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

`atomicAdd()` supports the following data types:

- `int`, `unsigned`, `unsigned long long`, `float`, `double`, `__half2`, `__half`.
- `__nv_bfloat16`, `__nv_bfloat162` on devices of compute capability 8.x and higher.
- `float2`, `float4` on devices of compute capability 9.x and higher, and only supported for global memory addresses.

The atomicity of `atomicAdd()` applied to vector types, for example `__half2` or `float4`, is guaranteed separately for each of the components; the entire vector is not guaranteed to be atomic as a single access.

5.4.5.1.2 `atomicSub()`

```
T atomicSub(T* address, T val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old - val`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

`atomicSub()` supports the following data types:

- `int`, `unsigned`

5.4.5.1.3 `atomicInc()`

```
unsigned atomicInc(unsigned* address, unsigned val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old >= val ? 0 : (old + 1)`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

5.4.5.1.4 `atomicDec()`

```
unsigned atomicDec(unsigned* address, unsigned val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `(old == 0 || old > val) ? val : (old - 1)`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

5.4.5.1.5 `atomicAnd()`

```
T atomicAnd(T* address, T val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old & val`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

`atomicAnd()` supports the following data types:

- `int`, `unsigned`, `unsigned long long`.

5.4.5.1.6 `atomicOr()`

```
T atomicOr(T* address, T val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old | val`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

`atomicOr()` supports the following data types:

- `int`, `unsigned`, `unsigned long long`.

5.4.5.1.7 `atomicXor()`

```
T atomicXor(T* address, T val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old ^ val`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

`atomicXor()` supports the following data types:

- `int`, `unsigned`, `unsigned long long`.

5.4.5.1.8 `atomicMin()`

```
T atomicMin(T* address, T val);
```

The function performs the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes the minimum of `old` and `val`.
3. Stores the result back to memory at the same address.

The function returns the `old` value.

`atomicMin()` supports the following data types:

- `int`, `unsigned`, `unsigned long long`, `long long`.

5.4.5.1.9 atomicMax()

```
T atomicMax(T* address, T val);
```

The function performs the following operations in one atomic transaction:

1. Reads the old value located at the address address in global or shared memory.
2. Computes the maximum of old and val.
3. Stores the result back to memory at the same address.

The function returns the old value.

atomicMax() supports the following data types:

- int, unsigned, unsigned long long, long long.

5.4.5.1.10 atomicExch()

```
T atomicExch(T* address, T val);
```

```
template<typename T>
T atomicExch(T* address, T val); // only 128-bit types, compute capability 9.x
↪and higher
```

The function performs the following operations in one atomic transaction:

1. Reads the old value located at the address address in global or shared memory.
2. Stores val back to memory at the same address.

The function returns the old value.

atomicExch() supports the following data types:

- int, unsigned, unsigned long long, float.

The C++ template function atomicExch() supports 128-bit types with the following requirements:

- Compute capability 9.x and higher.
- T must be aligned to 16 bytes, namely alignof(T) >= 16.
- T must be trivially copyable, namely std::is_trivially_copyable_v<T>.
- For C++03 and older: T must be trivially constructible, namely std::is_default_constructible_v<T>.

5.4.5.1.11 atomicCAS()

```
T atomicCAS(T* address, T compare, T val);
```

```
template<typename T>
T atomicCAS(T* address, T compare, T val);  // only 128-bit types, compute
↪capability 9.x and higher
```

The function performs the following operations in one atomic transaction:

1. Reads the old value located at the address address in global or shared memory.
2. Computes old == compare ? val : old.

3. Stores the result back to memory at the same address.

The function returns the old value.

atomicCAS() supports the following data types:

- int, unsigned, unsigned long long, unsigned short.

The C++ template function atomicCAS() supports 128-bit types with the following requirements:

- Compute capability 9.x and higher.
- T must be aligned to 16 bytes, namely alignof(T) >= 16.
- T must be trivially copyable, namely std::is_trivially_copyable_v<T>.
- For C++03 and older: T must be trivially constructible, namely std::is_default_constructible_v<T>.

5.4.5.2 Built-in Atomic Functions

CUDA 12.8 and later support CUDA compiler built-in functions for atomic operations, following the same memory ordering semantics as C++ standard atomic operations and the CUDA thread scopes. The functions follow the GNU's atomic built-in function signature with an extra argument for thread scope.

nvcc defines the macro __CUDACC_DEVICE_ATOMIC_BUILTINS__ when built-in atomic functions are supported.

Below are listed the raw enumerators for the memory orders and thread scopes, which are used as the order and scope arguments of the built-in atomic functions:

```
// atomic memory orders
enum {
   __NV_ATOMIC_RELAXED,
   __NV_ATOMIC_CONSUME,
   __NV_ATOMIC_ACQUIRE,
   __NV_ATOMIC_RELEASE,
   __NV_ATOMIC_ACQ_REL,
   __NV_ATOMIC_SEQ_CST
};
```

```
// thread scopes
enum {
   __NV_THREAD_SCOPE_THREAD,
   __NV_THREAD_SCOPE_BLOCK,
   __NV_THREAD_SCOPE_CLUSTER,
   __NV_THREAD_SCOPE_DEVICE,
   __NV_THREAD_SCOPE_SYSTEM
};
```

- The memory order corresponds to C++ standard atomic operations' memory order.
- The thread scope follows the cuda::thread_scope definition.
- __NV_ATOMIC_CONSUME memory order is currently implemented using stronger __NV_ATOMIC_ACQUIRE memory order.
- __NV_THREAD_SCOPE_THREAD thread scope is currently implemented using wider __NV_THREAD_SCOPE_BLOCK thread scope.

Example:

```
__device__ T __nv_atomic_load_n(T*  pointer,
                                int memory_order,
                                int thread_scope = __NV_THREAD_SCOPE_SYSTEM);
```

Atomic built-in functions have the following restrictions:

- They can only be used in device functions.
- They cannot operate on local memory.
- The addresses of these functions cannot be taken.
- The `order` and `scope` arguments must be integer literals; they cannot be variables.
- The thread scope `__NV_THREAD_SCOPE_CLUSTER` is supported on architectures `sm_90` and higher.

Example of unsupported cases:

```
 // Not permitted in a host function
 __host__ void bar() {
     unsigned u1 = 1, u2 = 2;
     __nv_atomic_load(&u1, &u2, __NV_ATOMIC_RELAXED, __NV_THREAD_SCOPE_
↪SYSTEM);
 }

 // Not permitted to be applied to local memory
__device__ void foo() {
   unsigned a = 1, b;
   __nv_atomic_load(&a, &b, __NV_ATOMIC_RELAXED, __NV_THREAD_SCOPE_SYSTEM);
}

 // Not permitted as a template default argument.
 // The function address cannot be taken.
 template<void *F = __nv_atomic_load_n>
 class X {
     void *f = F; // The function address cannot be taken.
 };

 // Not permitted to be called in a constructor initialization list.
 class Y {
     int a;
 public:
     __device__ Y(int *b): a(__nv_atomic_load_n(b, __NV_ATOMIC_RELAXED)) {}
 };
```

5.4.5.2.1 `__nv_atomic_fetch_add()`, `__nv_atomic_add()`

```
__device__ T    __nv_atomic_fetch_add(T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_add      (T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old + val`.
3. Stores the result back to memory at the same address.

- `__nv_atomic_fetch_add` returns the `old` value.
- `__nv_atomic_add` has no return value.

The functions support the following data types:

- `int`, `unsigned`, `unsigned long long`, `float`, `double`.

5.4.5.2.2 `__nv_atomic_fetch_sub()`, `__nv_atomic_sub()`

```
__device__ T    __nv_atomic_fetch_sub(T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_sub      (T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old - val`.
3. Stores the result back to memory at the same address.

- `__nv_atomic_fetch_sub` returns the `old` value.
- `__nv_atomic_sub` has no return value.

The functions support the following data types:

- `int`, `unsigned`, `unsigned long long`, `float`, `double`.

5.4.5.2.3 `__nv_atomic_fetch_and()`, `__nv_atomic_and()`

```
__device__ T    __nv_atomic_fetch_and(T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_and      (T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old & val`.
3. Stores the result back to memory at the same address.

- `__nv_atomic_fetch_and` returns the `old` value.
- `__nv_atomic_and` has no return value.

The functions support the following data types:

- Any integral type of size 4 or 8 bytes.

5.4.5.2.4 `__nv_atomic_fetch_or()`, `__nv_atomic_or()`

```
__device__ T    __nv_atomic_fetch_or(T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_or      (T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old | val`.
3. Stores the result back to memory at the same address.

- `__nv_atomic_fetch_or` returns the `old` value.
- `__nv_atomic_or` has no return value.

The functions support the following data types:

- Any integral type of size 4 or 8 bytes.

5.4.5.2.5 `__nv_atomic_fetch_xor()`, `__nv_atomic_xor()`

```
__device__ T    __nv_atomic_fetch_xor(T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_xor      (T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes `old ^ val`.
3. Stores the result back to memory at the same address.

- `__nv_atomic_fetch_xor` returns the `old` value.
- `__nv_atomic_xor` has no return value.

The functions support the following data types:

- Any integral type of size 4 or 8 bytes.

5.4.5.2.6 `__nv_atomic_fetch_min()`, `__nv_atomic_min()`

```
__device__ T    __nv_atomic_fetch_min(T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_min      (T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes the minimum of `old` and `val`.
3. Stores the result back to memory at the same address.

- `__nv_atomic_fetch_min` returns the `old` value.

- `__nv_atomic_min` has no return value.

The functions support the following data types:

- `unsigned`, `int`, `unsigned long long`, `long long`.

5.4.5.2.7 `__nv_atomic_fetch_max()`, `__nv_atomic_max()`

```
__device__ T    __nv_atomic_fetch_max(T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_max      (T* address, T val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. Computes the maximum of `old` and `val`.
3. Stores the result back to memory at the same address.

- `__nv_atomic_fetch_max` returns the `old` value.
- `__nv_atomic_max` has no return value.

The functions support the following data types:

- `unsigned`, `int`, `unsigned long long`, `long long`

5.4.5.2.8 `__nv_atomic_exchange()`, `__nv_atomic_compare_exchange_n()`

```
__device__ T    __nv_atomic_exchange_n(T* address, T val,          int order,
↪int scope = __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_exchange  (T* address, T* val, T* ret, int order,
↪int scope = __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the `old` value located at the address `address` in global or shared memory.
2. `__nv_atomic_exchange_n` stores `val` to where `address` points to.
   `__nv_atomic_exchange` stores `old` to where `ret` points to and stores the value located at the address `val` to where `address` points to.

- `__nv_atomic_exchange_n` returns the `old` value.
- `__nv_atomic_exchange` has no return value.

The functions support the following data types:

- Any data type of size of 4, 8 or 16 bytes.
- The 16-byte data type is supported on devices of compute capability 9.x and higher.

5.4.5.2.9 `__nv_atomic_compare_exchange()`, `__nv_atomic_compare_exchange_n()`

```
__device__ bool __nv_atomic_compare_exchange  (T* address, T* expected, T*
↪desired, bool weak, int success_order, int failure_order,
                                               int scope = __NV_THREAD_SCOPE_
↪SYSTEM);
```

(continues on next page)

(continued from previous page)

```
__device__ bool __nv_atomic_compare_exchange_n(T* address, T* expected, T
↪desired, bool weak, int success_order, int failure_order,
                                                 int scope = __NV_THREAD_SCOPE_
↪SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the old value located at the address address in global or shared memory.
2. Compare old with the value where expected points to.
3. If they are equal, the return value is true and desired is stored to where address points to. Otherwise, it returns false and old is stored to where expected points to.

The parameter weak is ignored and it picks the stronger memory order between success_order and failure_order to execute the compare-and-exchange operation.

The functions support the following data types:

- Any data type of size of 2, 4, 8 or 16 bytes.
- The 16-byte data type is supported on devices with compute capability 9.x and higher.

5.4.5.2.10 `__nv_atomic_load()`, `__nv_atomic_load_n()`

```
__device__ void __nv_atomic_load  (T* address, T* ret, int order, int scope =
↪__NV_THREAD_SCOPE_SYSTEM);
__device__ T    __nv_atomic_load_n(T* address,         int order, int scope =
↪__NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the old value located at the address address in global or shared memory.
2. __nv_atomic_load stores old to where ret points to.
   __nv_atomic_load_n returns old.

The functions support the following data types:

- Any data type of size 1, 2, 4, 8 or 16 bytes.

order cannot be __NV_ATOMIC_RELEASE or __NV_ATOMIC_ACQ_REL.

5.4.5.2.11 `__nv_atomic_store()`, `__nv_atomic_store_n()`

```
__device__ void __nv_atomic_store  (T* address, T* val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
__device__ void __nv_atomic_store_n(T* address, T  val, int order, int scope
↪= __NV_THREAD_SCOPE_SYSTEM);
```

The functions perform the following operations in one atomic transaction:

1. Reads the old value located at the address address in global or shared memory.
2. __nv_atomic_store reads the value where val points to and stores to where address points to.
   __nv_atomic_store_n stores val to where address points to.

`order` cannot be `__NV_ATOMIC_CONSUME`, `__NV_ATOMIC_ACQUIRE` or `__NV_ATOMIC_ACQ_REL`.

5.4.5.2.12 `__nv_atomic_thread_fence()`

```
__device__ void __nv_atomic_thread_fence(int order, int scope = __NV_THREAD_
↪SCOPE_SYSTEM);
```

This atomic function establishes an ordering between memory accesses requested by this thread based on the specified memory order. The thread scope parameter specifies the set of threads that may observe the ordering effect of this operation.

5.4.6. Warp Functions

The following section describes the warp functions that allow threads within a warp to communicate with each other and perform computations.

Hint

It is suggested to use the CUB Warp-Wide "Collective" Primitives to perform warp operations whenever possible for efficiency, safety, and portability reasons.

5.4.6.1 Warp Active Mask

```
unsigned __activemask();
```

The function returns a 32-bit integer mask representing all currently active threads in the calling warp. The Nth bit is set if the Nth lane in the warp is active when `__activemask()` is called. *Inactive threads* are represented by 0 bits in the returned mask. Threads that have exited the program are always marked as inactive.

Warning

`__activemask()` cannot be used to determine which warp lanes execute a given branch. This function is intended for opportunistic warp-level programming and only provides an instantaneous snapshot of the active threads within a warp.

```
// Check whether at least one thread's predicate evaluates to true
if (pred) {
    // Invalid: the value of 'at_least_one' is non-deterministic
    // and could vary between executions.
    at_least_one = __activemask() > 0;
}
```

Note that threads convergent at an `__activemask()` call are not guaranteed to remain convergent at subsequent instructions unless those instructions are warp synchronizing intrinsics (`__sync`).

For example, the compiler could reorder instructions, and the set of active threads might not be preserved:

```
unsigned mask      = __activemask();             // Assume mask ==
↪0xFFFFFFFF (all bits set, all threads active)
int      predicate = threadIdx.x % 2 == 0;        // 1 for even threads, 0
↪for odd threads
int      result    = __any_sync(mask, predicate); // Active threads might not
↪be preserved
```

5.4.6.2 Warp Vote Functions

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

The warp vote functions enable the threads of a given *warp* to perform a reduction-and-broadcast operation. These functions take an integer `predicate` as input from each non-exited thread in the warp and compare those values with zero. The results of the comparisons are then combined (reduced) across the *active threads* of the warp in one of the following ways, broadcasting a single return value to each participating thread:

`__all_sync(unsigned mask, predicate):`
Evaluates `predicate` for all non-exited threads in `mask` and returns non-zero if `predicate` evaluates to non-zero for all of them.

`__any_sync(unsigned mask, predicate):`
Evaluates `predicate` for all non-exited threads in `mask` and returns non-zero if `predicate` evaluates to non-zero for one or more of them.

`__ballot_sync(unsigned mask, predicate):`
Evaluates `predicate` for all non-exited threads in `mask` and returns an integer whose Nth bit is set if `predicate` evaluates to non-zero for the Nth thread of the warp and the Nth thread is active. Otherwise, the Nth bit is zero.

The functions are subject to the *Warp __sync Intrinsic Constraints*.

Warning

These intrinsics do not provide any memory ordering.

5.4.6.3 Warp Match Functions

Hint

It is suggested to use the libcu++ `cuda::device::warp_match_all()` function as a generalized and safer alternative to `__match_all_sync` function.

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

The warp match functions perform a broadcast-and-compare operation of a variable between non-exited threads within a *warp*.

__match_any_sync
Returns the mask of non-exited threads that have the same bitwise `value` in `mask`.

__match_all_sync
Returns `mask` if all non-exited threads in `mask` have the same bitwise `value`; otherwise 0 is returned. Predicate `pred` is set to `true` if all non-exited threads in `mask` have the same bitwise `value`; otherwise the predicate is set to false.

T can be `int`, `unsigned`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` or `double`.

The functions are subject to the *Warp __sync Intrinsic Constraints*.

Warning

These intrinsics do not provide any memory ordering.

5.4.6.4 Warp Reduce Functions

Hint

It is suggested to use the CUB Warp-Wide "Collective" Primitives to perform a Warp Reduction whenever possible for efficiency, safety, and portability reasons.

Supported by devices of compute capability 8.x or higher.

```
T        __reduce_add_sync(unsigned mask, T value);
T        __reduce_min_sync(unsigned mask, T value);
T        __reduce_max_sync(unsigned mask, T value);

unsigned __reduce_and_sync(unsigned mask, unsigned value);
unsigned __reduce_or_sync (unsigned mask, unsigned value);
unsigned __reduce_xor_sync(unsigned mask, unsigned value);
```

The `__reduce_<op>_sync` intrinsics perform a reduction operation on the data provided in `value` after synchronizing all non-exited threads named in `mask`.

__reduce_add_sync, __reduce_min_sync, __reduce_max_sync
Returns the result of applying an arithmetic add, min, or max reduction operation on the values provided in `value` by each non-exited thread named in `mask`. T can be an `unsigned` or `signed` integer.

__reduce_and_sync, __reduce_or_sync, __reduce_xor_sync
Returns the result of applying a bitwise AND, OR, or XOR reduction operation on the values provided in `value` by each non-exited thread named in `mask`.

The functions are subject to the *Warp __sync Intrinsic Constraints*.

Warning

These intrinsics do not provide any memory ordering.

5.4.6.5 Warp Shuffle Functions

Hint

It is suggested to use the libcu++ `cuda::device::warp_shuffle()` functions as a generalized and safer alternative to `__shfl_sync()` and `__shfl_<op>_sync()` intrinsics.

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

Warp shuffle functions exchange a value between non-exited threads within a *warp* without the use of shared memory.

`__shfl_sync()`: Direct copy from indexed lane.
The intrinsic function returns the value of `value` held by the thread whose ID is given by `srcLane`.

- If `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0.
- If `srcLane` is outside the range `[0, width - 1]`, the result corresponds to the value held by the `srcLane % width`, which is within the same subsection.

`__shfl_up_sync()`: Copy from a lane with a lower ID than the caller's.
The intrinsic function calculates a source lane ID by subtracting `delta` from the caller's lane ID. The value of `value` held by the resulting lane ID is returned: in effect, `value` is shifted up the warp by `delta` lanes.

- If `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0.
- The source lane index will not wrap around the value of `width`, so the lower `delta` lanes will remain unchanged.

`__shfl_down_sync()`: Copy from a lane with a higher ID than the caller's.
The intrinsic function calculates a source lane ID by adding `delta` to the caller's lane ID. The value of `value` held by the resulting lane ID is returned: this has the effect of shifting `value` down the warp by `delta` lanes.

- If `width` is less than `warpSize`, then each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0.
- As for `__shfl_up_sync()`, the ID number of the source lane will not wrap around the value of width and so the upper `delta` lanes will effectively remain unchanged.

`__shfl_xor_sync()`: Copy from a lane based on bitwise XOR of own lane ID.

The intrinsic function calculates a source lane ID by performing a bitwise XOR of the caller's lane ID and `laneMask`: the value of `value` held by the resulting lane ID is returned. This mode implements a butterfly addressing pattern, which is used in tree reduction and broadcast.

- If `width` is less than `warpSize`, then each group of `width` consecutive threads are able to access elements from earlier groups. However, if they attempt to access elements from later groups of threads their own value of `value` will be returned.

T can be:

- `int`, `unsigned`, `long`, `unsigned long`, `long long`, `unsigned long long`, `float` or `double`.
- `__half` and `__half2` with the `cuda_fp16.h` header included.
- `__nv_bfloat16` and `__nv_bfloat162` with the `cuda_bf16.h` header included.

Threads may only read data from another thread that is actively participating in the intrinsics. If the target thread is *inactive*, the retrieved value is undefined.

`width` must be a power of two in the range `[1, warpSize]`, namely 1, 2, 4, 8, 16, or 32. Other values will produce undefined results.

The functions are subject to the *Warp __sync Intrinsic Constraints*.

Examples of valid warp shuffle usage:

```
int laneId = threadIdx.x % warpSize;
int data   = ...

// all warp threads get 'data' from lane 0
int result1 = __shfl_sync(0xFFFFFFFF, data, 0);

if (laneId < 4) {
    // lanes 0, 1, 2, 3 get 'data' from lane 1
    int result2 = __shfl_sync(0xb1111, data, 1);
}

// lanes [0 - 15] get 'data' from lane 0
// lanes [16 - 31] get 'data' from lane 16
int result3 = __shfl_sync(0xFFFFFFFF, value, warpSize / 2);

// each lane gets 'data' from the lane two positions above
// lanes 30, 31 get their original value
int result4 = __shfl_down_sync(0xFFFFFFFF, data, 2);
```

Examples of invalid warp shuffle usage:

```
int laneId = threadIdx.x % warpSize;
int value  = ...
 // undefined behavior: lane 0 does not participate in the call
int result = (laneId > 0) ? __shfl_sync(0xFFFFFFFF, value, 0) : 0;

if (laneId <= 4) {
    // undefined behavior: destination lanes 5, 6 are not active for lanes 3, 4
    result = __shfl_down_sync(0b11111, value, 2);
```

(continues on next page)

(continued from previous page)

```
}

// undefined behavior: width is not a power of 2
__shfl_sync(0xFFFFFFFF, value, 0, /*width=*/31);
```

Warning

These intrinsics do not imply a memory barrier. They do not guarantee any memory ordering.

Example 1: Broadcast of a single value across a warp

CUDA C++

```
#include <cassert>
#include <cuda/warp>

__global__ void warp_broadcast_kernel(int input) {
    int laneId = threadIdx.x % 32;
    int value;
    if (laneId == 0) { // unused variable for all threads except lane 0
        value = input;
    }
    value = cuda::device::warp_shuffle_idx(value, 0); // Synchronize all
↪threads in warp, and get "value" from lane 0
    assert(value == input);
}

int main() {
    warp_broadcast_kernel<<<1, 32>>>(1234);
    cudaDeviceSynchronize();
    return 0;
}
```

Intrinsics

```
#include <assert.h>

__global__ void warp_broadcast_kernel(int input) {
    int laneId = threadIdx.x % 32;
    int value;
    if (laneId == 0) { // unused variable for all threads except lane 0
        value = input;
    }
    value = __shfl_sync(0xFFFFFFFF, value, 0); // Synchronize all threads in
↪warp, and get "value" from lane 0
    assert(value == input);
}

int main() {
```

(continues on next page)

(continued from previous page)

```
    warp_broadcast_kernel<<<1, 32>>>(1234);
    cudaDeviceSynchronize();
    return 0;
}
```

See the example on Compiler Explorer.

Example 2: Inclusive plus-scan across sub-partitions of 8 threads

Hint

It is suggested to use the cub::WarpScan function for efficient and generalized warp scan functions.

CUDA C++

```
#include <cstdio>
#include <cub/cub.cuh>

__global__ void scan_sub_partition_with_8_threads_kernel() {
    using WarpScan    = cub::WarpScan<int, 8>;
    using TempStorage = typename WarpScan::TempStorage;
    __shared__ TempStorage temp_storage;

    int laneId = threadIdx.x % 32;
    int value  = 31 - laneId; // starting value to accumulate
    int partial_sum;
    WarpScan(temp_storage).InclusiveSum(value, partial_sum);
    printf("Thread %d final value = %d\n", threadIdx.x, partial_sum);
}

int main() {
    scan_sub_partition_with_8_threads_kernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Intrinsics

```
#include <stdio.h>

__global__ void scan_sub_partition_with_8_threads_kernel() {
    int laneId = threadIdx.x % 32;
    int value  = 31 - laneId; // starting value to accumulate
    // Loop to accumulate scan within my partition.
    // Scan requires log2(8) == 3 steps for 8 threads
    for (int delta = 1; delta <= 4; delta *= 2) {
        int tmp         = __shfl_up_sync(0xFFFFFFFF, value, delta, /*width=*/
↪8); // read from laneId - delta
        int source_lane = laneId % 8 - delta;
        if (source_lane >= 0) // lanes with 'source_lane < 0' have their value
```

(continues on next page)

(continued from previous page)

```
↪unchanged
            value += tmp;
        }
    printf("Thread %d final value = %d\n", threadIdx.x, value);
}

int main() {
    scan_sub_partition_with_8_threads_kernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

See the example on Compiler Explorer.

Example 3: Reduction across a warp

Hint

It is suggested to use the cub::WarpReduce function for efficient and generalized warp reduction functions.

CUDA C++

```
#include <cstdio>
#include <cub/cub.cuh>
#include <cuda/warp>

__global__ void warp_reduce_kernel() {
    using WarpReduce  = cub::WarpReduce<int>;
    using TempStorage = typename WarpReduce::TempStorage;
    __shared__ TempStorage temp_storage;

    int laneId     = threadIdx.x % 32;
    int value      = 31 - laneId; // starting value to accumulate
    auto aggregate = WarpReduce(temp_storage).Sum(value);
    aggregate      = cuda::device::warp_shuffle_idx(aggregate, 0);
    printf("Thread %d final value = %d\n", threadIdx.x, aggregate);
}

int main() {
    warp_reduce_kernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

Intrinsics

```
#include <stdio.h>

__global__ void warp_reduce_kernel() {
```

(continues on next page)

(continued from previous page)

```
    int laneId = threadIdx.x % 32;
    int value  = 31 - laneId; // starting value to accumulate
    // Use XOR mode to perform butterfly reduction
    // A full-warp reduction requires log2(32) == 5 steps
    for (int i = 1; i <= 16; i *= 2)
        value += __shfl_xor_sync(0xFFFFFFFF, value, i);
    // "value" now contains the sum across all threads
    printf("Thread %d final value = %d\n", threadIdx.x, value);
}

int main() {
    warp_reduce_kernel<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

See the example on Compiler Explorer.

5.4.6.6 Warp __sync Intrinsic Constraints

All warp __sync intrinsics, such as:

- `__shfl_sync`, `__shfl_up_sync`, `__shfl_down_sync`, `__shfl_xor_sync`
- `__match_any_sync`, `__match_all_sync`
- `__reduce_add_sync`, `__reduce_min_sync`, `__reduce_max_sync`, `__reduce_and_sync`, `__reduce_or_sync`, `__reduce_xor_sync`
- `__syncwarp`

use the `mask` parameter to indicate which warp threads participate in the call. This parameter ensures proper convergence before the hardware executes the intrinsic.

Each bit in the `mask` corresponds to a thread's lane ID (`threadIdx.x % warpSize`). The intrinsic waits until all non-exited warp threads specified in the `mask` reach the call.
The following constraints must be met for correct execution:

- Each calling thread must have its corresponding bit set in the `mask`.
- Each non-calling thread must have its corresponding bit set to zero in the `mask`. Exited threads are ignored.
- All non-exited threads specified in the `mask` must execute the intrinsic with the same `mask` value.
- Warp threads may call the intrinsic concurrently with different `mask` values, provided the masks are disjoint. Such condition is valid even in divergent control flow.

The behavior of warp __sync functions is invalid, such as kernel hang, or undefined if:

- A calling thread is not specified in the `mask`.
- A non-exited thread specified in the `mask` fails to either eventually exit or call the intrinsic at the same program point with the same `mask` value.
- In conditional code, all conditions must evaluate identically across all non-exited threads specified in the `mask`.

Note

The intrinsics achieve the best efficiency when all warp threads participate in the call, namely when the mask is set to 0xFFFFFFFF.

Examples of valid warp intrinsics usage:

```
__global__ void valid_examples() {
    if (threadIdx.x < 4) {        // threads 0, 1, 2, 3 are active
        __all_sync(0b1111, pred); // CORRECT, threads 0, 1, 2, 3 participate
↪in the call
    }

    if (threadIdx.x == 0)
        return; // exit
    // CORRECT, all non-exited threads participate in the call
    __all_sync(0xFFFFFFFF, pred);
}
```

Disjoint mask examples:

```
__global__ void example_syncwarp_with_mask(int* input_data, int* output_data)
↪{
    if (threadIdx.x < warpSize) {
        __shared__ int shared_data[warpSize];
        shared_data[threadIdx.x] = input_data[threadIdx.x];

        unsigned mask = threadIdx.x < 16 ? 0xFFFF : 0xFFFF0000; // CORRECT
        __syncwarp(mask);
        if (threadIdx.x == 0 || threadIdx.x == 16)
            output_data[threadIdx.x] = shared_data[threadIdx.x + 1];
    }
}
```

```
__global__ void example_syncwarp_with_mask_branches(int* input_data, int*
↪output_data) {
    if (threadIdx.x < warpSize) {
        __shared__ int shared_data[warpSize];
        shared_data[threadIdx.x] = input_data[threadIdx.x];

        if (threadIdx.x < 16) {
            unsigned mask = 0xFFFF; // CORRECT
            __syncwarp(mask);
            output_data[threadIdx.x] = shared_data[15 - threadIdx.x];
        }
        else {
            unsigned mask = 0xFFFF0000; // CORRECT
            __syncwarp(mask);
            output_data[threadIdx.x] = shared_data[31 - threadIdx.x];
        }
    }
}
```

Examples of invalid warp intrinsics usage:

```
if (threadIdx.x < 4) {              // threads 0, 1, 2, 3 are active
    __all_sync(0b0000011, pred); // WRONG, threads 2, 3 are active but not set
↪in mask
    __all_sync(0b1111111, pred); // WRONG, threads 4, 5, 6 are not active but
↪set in mask
}

// WRONG, participating threads have a different and overlapping mask
__all_sync(threadIdx.x == 0 ? 1 : 0xFFFFFFFF, pred);
```

5.4.7. CUDA-Specific Macros

5.4.7.1 __CUDA_ARCH__

The macro __CUDA_ARCH__ represents the virtual architecture of the NVIDIA GPU for which the code is being compiled. Its value may differ from the device's actual compute capability. This macro enables the writing of code paths that are specialized for particular GPU architectures, which may be necessary for optimal performance or to use architecture-specific features and instructions. The macro can also be used to distinguish between host and device code.

__CUDA_ARCH__ is only defined in device code, namely in the __device__, __host__ __device__, and __global__ functions. The value of the macro is associated with the nvcc option compute_<version>, with the relation __CUDA_ARCH__ = <version> * 10.

Example:

```
nvcc --generate-code arch=compute_80,code=sm_90 prog.cu
```

defines __CUDA_ARCH__ as 800.

__CUDA_ARCH__ Constraints

1. The type signatures of the following entities shall not depend on whether __CUDA_ARCH__ is defined, nor on its value.

- __global__ functions and function templates.
- __device__ and __constant__ variables.
- Textures and surfaces.

Example:

```
#if !defined(__CUDA_ARCH__)
    typedef int my_type;
#else
    typedef double my_type;
#endif

__device__ my_type my_var;            // ERROR: my_var's type depends on __CUDA_
↪ARCH__

__global__ void kernel(my_type in) { // ERROR: kernel's type depends on __CUDA_
↪ARCH__
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

2. If a `__global__` function template is instantiated and launched from the host, then it must be instantiated with the same template arguments, regardless of whether `__CUDA_ARCH__` is defined or its value.

Example:

```
__device__ int result;

template <typename T>
__global__ void kernel(T in) {
    result = in;
}

__host__ __device__ void host_device_function(void) {
#if !defined(__CUDA_ARCH__)
    kernel<<<1, 1>>>(1); // ERROR: "kernel<int>" instantiation only
                         //        when __CUDA_ARCH__ is undefined!
#endif
}

int main(void) {
    host_device_function();
    cudaDeviceSynchronize();
    return 0;
}
```

3. In separate compilation mode, the presence or absence of a function or variable definition with external linkage shall not depend on the definition of `__CUDA_ARCH__` or on its value.

Example:

```
#if !defined(__CUDA_ARCH__)
    void host_function(void) {} // ERROR: The definition of host_function()
                                //        is only present when __CUDA_ARCH__
                                //        is undefined
#endif
```

4. In separate compilation, the preprocessor macro `__CUDA_ARCH__` must not be used in headers to prevent objects from having different behaviors. Alternatively, all objects must be compiled for the same virtual architecture. If a weak or template function is defined in a header and its behavior depends on `__CUDA_ARCH__`, then instances of that function in different objects could conflict if those objects are compiled for different compute architectures.

For example, if a header file `a.h` contains:

```
template<typename T>
__device__ T* get_ptr() {
#if __CUDA_ARCH__ == 900
    return nullptr; /* no address */
#else
    __shared__ T arr[256];
```

(continues on next page)

(continued from previous page)

```
    return arr;
#endif
}
```

Then if a.cu and b.cu both include a.h and instantiate get_ptr() for the same type, and b.cu expects a non-NULL address, and compile with:

```
nvcc -arch=compute_70 -dc a.cu
nvcc -arch=compute_80 -dc b.cu
nvcc -arch=sm_80 a.o b.o

Only one version of the ``get_ptr()`` function is used at link time, so the
↪behavior depends on which version is chosen. To avoid this issue, either
↪``a.cu`` and ``b.cu`` must be compiled for the same compute architecture,
↪or ``__CUDA_ARCH__`` should not be used in the shared header function.
```

The compiler does not guarantee that a diagnostic will be generated for the unsupported uses of __CUDA_ARCH__ described above.

5.4.7.2 __CUDA_ARCH_SPECIFIC__ and __CUDA_ARCH_FAMILY_SPECIFIC__

The macros __CUDA_ARCH_SPECIFIC__ and __CUDA_ARCH_FAMILY_SPECIFIC__ are defined to identify GPU devices with *architecture-* and *family-* specific features, respectively. See *Feature Set Compiler Targets* section for more information.

Similarly to __CUDA_ARCH__, __CUDA_ARCH_SPECIFIC__ and __CUDA_ARCH_FAMILY_SPECIFIC__ are only defined in the device code, namely in the __device__, __host__ __device__, and __global__ functions. The macros are associated with the nvcc options compute_<version>a and compute_<version>f.

```
nvcc --generate-code arch=compute_100a,code=sm_100a prog.cu
```

- __CUDA_ARCH__ == 1000.
- __CUDA_ARCH_SPECIFIC__ == 1000.
- __CUDA_ARCH_FAMILY_SPECIFIC__ == 1000.

```
nvcc --generate-code arch=compute_100f,code=sm_103f prog.cu
```

- __CUDA_ARCH__ == 1000.
- __CUDA_ARCH_FAMILY_SPECIFIC__ == 1000.
- __CUDA_ARCH_SPECIFIC__ is not defined.

```
nvcc -arch=sm_100 prog.cu
```

- __CUDA_ARCH__ == 1000.
- __CUDA_ARCH_FAMILY_SPECIFIC__ is not defined.
- __CUDA_ARCH_SPECIFIC__ is not defined.

```
nvcc -arch=sm_100a prog.cu
# equivalent to:
nvcc --generate-code arch=sm_100a,compute_100,compute_100a prog.cu
```

- `__CUDA_ARCH__ == 1000`.
- `__CUDA_ARCH_FAMILY_SPECIFIC__` is not defined.
- `__CUDA_ARCH_SPECIFIC__ == 1000` and `__CUDA_ARCH_SPECIFIC__` not defined are both generated.

5.4.7.3 CUDA Feature Testing Macros

nvcc provides the following preprocessor macros for feature testing. The macros are defined when a particular feature is supported by the CUDA front-end compiler.

- `__CUDACC_DEVICE_ATOMIC_BUILTINS__`: Supports *device atomic compiler builtins*.
- `__NVCC_DIAG_PRAGMA_SUPPORT__`: Supports *diagnostic control pragmas*.
- `__CUDACC_EXTENDED_LAMBDA__`: Supports *extended lambdas*. Enabled by `--expt-extended-lambda` or `--extended-lambda` flag.
- `__CUDACC_RELAXED_CONSTEXPR__`: Support for *relaxed constexpr functions*. Enabled by the `--expt-relaxed-constexpr` flag.

5.4.7.4 `__nv_pure__` Attribute

In C/C++, a pure function has no side effects on its parameters and can access global variables, though it does not modify them.

CUDA provides `__nv_pure__` attribute supported for both host and device functions. The compiler translates `__nv_pure__` to the `pure` GNU attribute or to the Microsoft Visual Studio `noalias` attribute.

```
__device__ __nv_pure__
int add(int a, int b) {
    return a + b;
}
```

5.4.8. CUDA-Specific Functions

5.4.8.1 Address Space Predicate Functions

Address space predicate functions are used to determine the address space of a pointer.

Hint

It is suggested to use the `cuda::device::is_address_from()` and `cuda::device::is_object_from()` functions provided by libcu++ as a portable and safer alternative to Address Space Predicate intrinsic functions.

```
__device__ unsigned __isGlobal      (const void* ptr);
__device__ unsigned __isShared      (const void* ptr);
__device__ unsigned __isConstant    (const void* ptr);
__device__ unsigned __isGridConstant(const void* ptr);
__device__ unsigned __isLocal       (const void* ptr);
```

The functions return 1 if `ptr` contains the generic address of an object in the specified address space, `0` otherwise. Their behavior is unspecified if the argument is a NULL pointer.

- `__isGlobal()`: global memory space.
- `__isShared()`: shared memory space.
- `__isConstant()`: constant memory space.
- `__isGridConstant()`: kernel parameter annotated with `__grid_constant__`.
- `__isLocal()`: local memory space.

5.4.8.2 Address Space Conversion Functions

CUDA pointers (T*) can access objects regardless of where the objects are stored. For example, an `int*` can access `int` objects whether they reside in global or shared memory.

Address space conversion functions are used to convert between generic addresses and addresses in specific address spaces. These functions are useful when the compiler cannot determine a pointer's address space, for example, when crossing translation units or interacting with PTX instructions.

```
__device__ size_t __cvta_generic_to_global  (const void* ptr); // PTX: cvta.
↪to.global
__device__ size_t __cvta_generic_to_shared  (const void* ptr); // PTX: cvta.
↪to.shared
__device__ size_t __cvta_generic_to_constant(const void* ptr); // PTX: cvta.
↪to.const
__device__ size_t __cvta_generic_to_local   (const void* ptr); // PTX: cvta.
↪to.local
```

```
__device__ void* __cvta_global_to_generic  (size_t raw_ptr); // PTX: cvta.
↪global
__device__ void* __cvta_shared_to_generic  (size_t raw_ptr); // PTX: cvta.
↪shared
__device__ void* __cvta_constant_to_generic(size_t raw_ptr); // PTX: cvta.
↪const
__device__ void* __cvta_local_to_generic   (size_t raw_ptr); // PTX: cvta.
↪local
```

As an example of inter-operating with PTX instructions, the `ld.shared.s32  r0,  [ptr];` PTX instruction expects `ptr` to refer to the shared memory address space. A CUDA program with an `int*` pointer to an object in `__shared__` memory needs to convert this pointer to the shared address space before passing it to the PTX instruction by calling `__cvta_generic_to_shared` as follows:

```
__shared__ int smem_var;
smem_var        = 42;
size_t smem_ptr = __cvta_generic_to_shared(&smem_var);
int    output;
asm volatile("ld.shared.s32 %0, [%1];" : "=r"(output) : "l"(smem_ptr) :
↪"memory");
assert(output == 42);
```

A common optimization that exploits these address representations is reducing data structure size by leveraging the fact that the address ranges of shared, local, and constant spaces are smaller than 32 bits, which allows storing 32-bit addresses instead of 64-bit pointers and save registers. Additionally, 32-bit arithmetic is faster than 64-bit arithmetic. To obtain the 32-bit integer representation of these addresses, truncate the 64-bit value to 32 bits by casting from an unsigned 64-bit integer to an unsigned 32-bit integer:

```
__shared__ int smem_var;
uint32_t       smem_ptr_32bit = static_cast<uint32_t>(__cvta_generic_to_
↪shared(&smem_var));
```

To recover a generic address from such a 32-bit representation, zero-extend the address back to an unsigned 64-bit integer and then call the corresponding address space conversion function:

```
size_t smem_ptr_64bit = static_cast<size_t>(smem_ptr_32bit); // zero-extend
↪to 64 bits
void*  generic_ptr    = __cvta_shared_to_generic(smem_ptr_64bit);
assert(generic_ptr == &smem_var);
```

5.4.8.3 Low-Level Load and Store Functions

```
T __ldg(const T* address);
```

The function `__ldg()` performs a read-only L1/Tex cache load. It supports all C++ fundamental types, CUDA vector types (except x3 components), and extended floating-point types, such as `__half`, `__half2`, `__nv_bfloat16`, and `__nv_bfloat162`.

```
T __ldcg(const T* address);
T __ldca(const T* address);
T __ldcs(const T* address);
T __ldlu(const T* address);
T __ldcv(const T* address);
```

The functions perform a load using the cache operator specified in the PTX ISA guide. They support all C++ fundamental types, CUDA vector types (except x3 components), and extended floating-point types, such as `__half`, `__half2`, `__nv_bfloat16`, and `__nv_bfloat162`.

```
void __stwb(T* address, T value);
void __stcg(T* address, T value);
void __stcs(T* address, T value);
void __stwt(T* address, T value);
```

The functions perform a store using the cache operator specified in the PTX ISA guide. They support all C++ fundamental types, CUDA vector types (except x3 components), and extended floating-point types, such as `__half`, `__half2`, `__nv_bfloat16`, and `__nv_bfloat162`.

5.4.8.4 `__trap()`

Hint

It is suggested to use the `cuda::std::terminate()` function provided by libcu++ (C++ reference) as a portable alternative to `__trap()`.

A trap operation can be initiated by calling the `__trap()` function from any device thread.

```
void __trap();
```

Execution of the kernel is aborted, raising an interrupt in the host program. Calling `__trap()` results in a corrupted CUDA context, causing subsequent CUDA calls and kernel invocations to fail.

5.4.8.5 `__nanosleep()`

```
__device__ void __nanosleep(unsigned nanoseconds);
```

The function `__nanosleep(ns)` suspends the thread for a sleep duration of approximately `ns` nanoseconds. The maximum sleep duration is approximately one millisecond.

Example:

The following code implements a mutex with exponential back-off.

```
__device__ void mutex_lock(unsigned* mutex) {
    unsigned ns = 8;
    while (atomicCAS(mutex, 0, 1) == 1) {
        __nanosleep(ns);
        if (ns < 256) {
            ns *= 2;
        }
    }
}

__device__ void mutex_unlock(unsigned *mutex) {
    atomicExch(mutex, 0);
}
```

5.4.8.6 Dynamic Programming eXtension (DPX) Instructions

The DPX set of functions enables finding minimum and maximum values, as well as fused addition and minimum/maximum for up to three 16- or 32-bit signed or unsigned integer parameters. There is an optional ReLU, namely clamping to zero, feature.

Comparison functions:

- Three parameters. Semantic: `max(a, b, c)`, `min(a, b, c)`.

```
     int __vimax3_s32  (     int,      int,      int);
unsigned __vimax3_s16x2(unsigned, unsigned, unsigned);
unsigned __vimax3_u32  (unsigned, unsigned, unsigned);
unsigned __vimax3_u16x2(unsigned, unsigned, unsigned);

     int __vimin3_s32  (     int,      int,      int);
unsigned __vimin3_s16x2(unsigned, unsigned, unsigned);
unsigned __vimin3_u32  (unsigned, unsigned, unsigned);
unsigned __vimin3_u16x2(unsigned, unsigned, unsigned);
```

- Two parameters, with ReLU. Semantic: `max(a, b, 0)`, `max(min(a, b), 0)`.

```
     int __vimax_s32_relu  (     int,      int);
unsigned __vimax_s16x2_relu(unsigned, unsigned);

     int __vimin_s32_relu  (     int,      int);
unsigned __vimin_s16x2_relu(unsigned, unsigned);
```

- Three parameters, with ReLU. Semantic: `max(a, b, c, 0)`, `max(min(a, b, c), 0)`.

```
     int __vimax3_s32_relu  (     int,      int,      int);
unsigned __vimax3_s16x2_relu(unsigned, unsigned, unsigned);

     int __vimin3_s32_relu  (     int,      int,      int);
unsigned __vimin3_s16x2_relu(unsigned, unsigned, unsigned);
```

- Two parameters, also returning which parameter was smaller/larger:

```
     int __vibmax_s32  (     int,      int, bool* pred);
unsigned __vibmax_u32  (unsigned, unsigned, bool* pred);
unsigned __vibmax_s16x2(unsigned, unsigned, bool* pred);
unsigned __vibmax_u16x2(unsigned, unsigned, bool* pred);

     int __vibmin_s32  (     int,      int, bool* pred);
unsigned __vibmin_u32  (unsigned, unsigned, bool* pred);
unsigned __vibmin_s16x2(unsigned, unsigned, bool* pred);
unsigned __vibmin_u16x2(unsigned, unsigned, bool* pred);
```

Fused addition and minimum/maximum:

- Three parameters, comparing (first + second) with the third. Semantic: `max(a + b, c)`, `min(a + b, c)`

```
     int __viaddmax_s32  (     int,      int,      int);
unsigned __viaddmax_s16x2(unsigned, unsigned, unsigned);
unsigned __viaddmax_u32  (unsigned, unsigned, unsigned);
unsigned __viaddmax_u16x2(unsigned, unsigned, unsigned);

     int __viaddmin_s32  (     int,      int,      int);
unsigned __viaddmin_s16x2(unsigned, unsigned, unsigned);
unsigned __viaddmin_u32  (unsigned, unsigned, unsigned);
unsigned __viaddmin_u16x2(unsigned, unsigned, unsigned);
```

- Three parameters, with ReLU, comparing (first + second) with the third and a zero. Semantic: `max(a + b, c, 0)`, `max(min(a + b, c), 0)`

```
     int __viaddmax_s32_relu  (     int,      int,      int);
unsigned __viaddmax_s16x2_relu(unsigned, unsigned, unsigned);

     int __viaddmin_s32_relu  (     int,      int,      int);
unsigned __viaddmin_s16x2_relu(unsigned, unsigned, unsigned);
```

These instructions are hardware-accelerated or software emulated depending on compute capability. See Arithmetic Instructions section for the compute capability requirements.

The full API can be found in CUDA Math API documentation.

The DPX is an exceptionally useful tool for implementing dynamic programming algorithms such as the Smith-Waterman and Needleman-Wunsch algorithms in genomics and the Floyd-Warshall algorithm in route optimization.

Maximum value of three signed 32-bit integers, with ReLU:

```
int a           = -15;
int b           = 8;
int c           = 5;
int max_value_0 = __vimax3_s32_relu(a, b, c); // max(-15, 8, 5, 0) = 8
int d           = -2;
int e           = -4;
int max_value_1 = __vimax3_s32_relu(a, d, e); // max(-15, -2, -4, 0) = 0
```

Minimum value of the sum of two 32-bit signed integers, another 32-bit signed integer and a zero (ReLU):

```
int a           = -5;
int b           = 6;
int c           = -2;
int max_value_0 = __viaddmax_s32_relu(a, b, c); // max(-5 + 6, -2, 0) = max(1,
↪ -2, 0) = 1
int d           = 4;
int max_value_1 = __viaddmax_s32_relu(a, d, c); // max(-5 + 4, -2, 0) = max(-
↪1, -2, 0) = 0
```

Minimum value of two unsigned 32-bit integers and determining which value is smaller:

```
unsigned a = 9;
unsigned b = 6;
bool     smaller_value;
unsigned min_value = __vibmin_u32(a, b, &smaller_value); // min_value is 6,
↪smaller_value is true
```

Maximum values of three pairs of unsigned 16-bit integers:

```
unsigned a         = 0x00050002;
unsigned b         = 0x00070004;
unsigned c         = 0x00020006;
unsigned max_value = __vimax3_u16x2(a, b, c); // max(5, 7, 2) and max(2, 4, 6),
↪ so max_value is 0x00070006
```

5.4.9. Compiler Optimization Hints

Compiler optimization hints decorate code with additional information to help the compiler optimize generated code.

- The built-in functions are always available in the device code.
- Host code support depends on the host compiler.

5.4.9.1 #pragma unroll

The compiler unrolls small loops with a known trip count by default. However, the `#pragma unroll` directive can be used to control the unrolling of any given loop. This directive must be placed immediately before the loop and only applies to that loop.

An integral constant expression may optionally follow. The following are cases for an integral constant expression:

- If it is absent, the loop will be completely unrolled if its trip count is constant.
- If it evaluates to `0` or 1, the loop will not be unrolled.
- If it is a non-positive integer or greater than `INT_MAX`, the pragma will be ignored, and a warning will be issued.

Examples:

```
struct MyStruct {
    static constexpr int value = 4;
};

inline constexpr int Count = 4;

__device__ void foo(int* p1, int* p2) {
    // no argument specified, the loop will be completely unrolled
    #pragma unroll
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 2;

    // unroll value = 5
    #pragma unroll (Count + 1)
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 4;

    // unroll value = 1, loop unrolling disabled
    #pragma unroll 1
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 8;

    // unroll value = 4
    #pragma unroll (MyStruct::value)
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 16;

    // negative value, pragma unroll ignored
    #pragma unroll -1
    for (int i = 0; i < 12; ++i)
        p1[i] += p2[i] * 2;
}
```

See the example on Compiler Explorer.

5.4.9.2 `__builtin_assume_aligned()`

Hint

It is suggested to use the `cuda::std::assume_aligned()` function provided by libcu++ (C++ reference) as a portable and safer alternative to the built-in functions.

```
void* __builtin_assume_aligned(const void* ptr, size_t align)
void* __builtin_assume_aligned(const void* ptr, size_t align, <integral type>
↪offset)
```

The built-in functions enable the compiler to assume that the returned pointer is aligned to at least `align` bytes.

- The three parameter version enables the compiler to assume that `(char*) ptr - offset` is aligned to at least `align` bytes.

`align` must be a power of two and an integer literal.

Examples:

```
void* res1 = __builtin_assume_aligned(ptr, 32);    // compiler can assume
↪'res1' is at least 32-byte aligned
void* res2 = __builtin_assume_aligned(ptr, 32, 8); // compiler can assume
↪'res2 = (char*) ptr - 8' is at least 32-byte aligned
```

5.4.9.3 `__builtin_assume()` and `__assume()`

```
void __builtin_assume(bool predicate)
void __assume         (bool predicate) // only with Microsoft Compiler
```

The built-in function enables the compiler to assume that the boolean argument is true. If the argument is false at runtime, the behavior is undefined. Note that if the argument has side effects, the behavior is unspecified.

Example:

```
__device__ bool is_greater_than_zero(int value) {
    return value > 0;
}

__device__ bool f(int value) {
    __builtin_assume(value > 0);
    return is_greater_than_zero(value); // returns true, without evaluating
↪the condition
}
```

5.4.9.4 `__builtin_expect()`

```
long __builtin_expect(long input, long expected)
```

The built-in function tells the compiler that `input` is expected to equal `expected`, and returns the value of `input`. It is typically used to provide branch prediction information to the compiler. It behaves like the C++20 `[[likely]]` and `[[unlikely]]` attributes.

Example:

```
// indicate to the compiler that likely "var == 0"
if (__builtin_expect(var, 0))
    doit();
```

5.4.9.5 `__builtin_unreachable()`

```
void __builtin_unreachable(void)
```

The built-in function tells the compiler that the control flow will never reach the point at which the function is called. If the control flow does reach this point at runtime, the program has undefined behavior.

This function is useful for avoiding code generation of unreachable branches and disabling compiler warnings for unreachable code.

Example:

```
// indicates to the compiler that the default case label is never reached.
switch (in) {
    case 1:  return 4;
    case 2:  return 10;
    default: __builtin_unreachable();
}
```

5.4.9.6 Custom ABI Pragmas

The `#pragma nv_abi` directive enables applications compiled in *separate compilation* mode to achieve performance similar to that of *whole program compilation* by preserving the number of registers used by a function.

The syntax for using this pragma is as follows, where EXPR refers to any integral constant expression:

```
#pragma nv_abi preserve_n_data(EXPR) preserve_n_control(EXPR)
```

- The arguments that follow `#pragma nv_abi` are optional and may be provided in any order; however, at least one argument is required.
- The `preserve_n` arguments limit the number of registers preserved during a function call:
  - `preserve_n_data(EXPR)` limits the number of data registers.
  - `preserve_n_control(EXPR)` limits the number of control registers.

The `#pragma nv_abi` directive can be placed immediately before a device function declaration or definition.

```
#pragma nv_abi preserve_n_data(16)
__device__ void dev_func();

#pragma nv_abi preserve_n_data(16) preserve_n_control(8)
__device__ int dev_func() {
    return 0;
}
```

Alternatively, it can be placed directly before an indirect function call within a C++ expression statement inside a device function. Note that while indirect function calls to free functions are supported, indirect calls to function references or class member functions are not supported.

```
__device__ int dev_func1();

struct MyStruct {
    __device__ int member_func2();
};

__device__ void test() {
    auto* dev_func_ptr = &dev_func1; // type: int (*)(void)
    #pragma nv_abi preserve_n_control(8)
    int v1 = dev_func_ptr();          // CORRECT, indirect call

    #pragma nv_abi preserve_n_control(8)
    int v2 = dev_func1();             // WRONG, direct call; the pragma has no
↪effect
                                      // dev_func1 has type: int(void)

    auto& dev_func_ref = &dev_func1; // type: int (&)(void)
    #pragma nv_abi preserve_n_control(8)
    int v3 = dev_func_ref();          // WRONG, call to a reference
                                      // the pragma has no effect

    auto member_function_ptr = &MyStruct::member_func2; // type: int
↪(MyStruct::*)(void)
    #pragma nv_abi preserve_n_control(8)
    int v4 = member_function_ptr();  // WRONG, indirect call to member function
                                      // the pragma has no effect
}
```

When applied to a device function's declaration or definition, the pragma modifies the custom ABI properties for any calls to that function. When placed at an indirect function call site, it affects the ABI properties only for that specific call. Note that the pragma only affects indirect function calls when placed at a call site; it has no effect on direct function calls.

```
#pragma nv_abi preserve_n_control(8)
__device__ int dev_func3();

__device__ int dev_func4();

__device__ void test() {
    int v1 = dev_func3();             // CORRECT, the pragma affects the direct
↪call

    auto* dev_func_ptr = &dev_func4; // type: int (*)(void)
    #pragma nv_abi preserve_n_control(8)
    int v2 = dev_func_ptr();          // CORRECT, the pragma affects the
↪indirect call

    int v3 = dev_func_ptr();          // WRONG, the pragma has no effect
}
```

Note that a program is ill-formed if the pragma arguments for a function declaration and its corresponding definition do not match.

5.4.10. Debugging and Diagnostics

5.4.10.1 Assertion

```
void assert(int expression);
```

The assert() macro stops kernel execution if expression is equal to zero. If the program is run within a debugger, a breakpoint is triggered, allowing the debugger to be used to inspect the current state of the device. Otherwise, each thread for which expression is equal to zero prints a message to stderr after synchronizing with the host via cudaDeviceSynchronize(), cudaStreamSynchronize(), or cudaEventSynchronize(). The format of this message is as follows:

```
<filename>:<line number>:<function>:
block: [blockIdx.x,blockIdx.y,blockIdx.z],
thread: [threadIdx.x,threadIdx.y,threadIdx.z]
Assertion `<expression>` failed.
```

Execution of the kernel is aborted, raising an interrupt in the host program. The assert() macro results in a corrupted CUDA context, causing any subsequent CUDA calls or kernel invocations to fail with cudaErrorAssert.

The kernel execution is unaffected if expression is different from zero.

For example, the following program from source file test.cu

```
#include <assert.h>

 __global__ void testAssert(void) {
     int is_one        = 1;
     int should_be_one = 0;

     // This will have no effect
     assert(is_one);

     // This will halt kernel execution
     assert(should_be_one);
 }

 int main(void) {
     testAssert<<<1,1>>>();
     cudaDeviceSynchronize();
     return 0;
 }
```

will output:

```
test.cu:11: void testAssert(): block: [0,0,0], thread: [0,0,0] Assertion
↪`should_be_one` failed.
```

Assertions are intended for debugging purposes. Since they can affect performance, it is recommended that they be disabled in production code. They can be disabled at compile time by defining the NDEBUG preprocessor macro before including assert.h or <cassert>, or by using the compiler flag -DNDEBUG. Note that the expression should not have side effects; otherwise, disabling the assertion will affect the functionality of the code.

5.4.10.2 Breakpoint Function

The execution of a kernel function can be suspended by calling the `__brkpt()` function from any device thread.

```
void __brkpt();
```

5.4.10.3 Diagnostic Pragmas

The following pragmas can be used to manage the severity of errors that are triggered when a specific diagnostic message is raised.

```
#pragma nv_diag_suppress
#pragma nv_diag_warning
#pragma nv_diag_error
#pragma nv_diag_default
#pragma nv_diag_once
```

The uses of these pragmas are as follows:

```
#pragma nv_diag_xxx <error_number1>, <error_number2> ...
```

The affected diagnostic is specified using the error number shown in the warning message. Any diagnostic can be changed to an error, but only warnings can have their severity suppressed or restored after being changed to an error. The `nv_diag_default` pragma returns the severity of a diagnostic to the severity that was in effect before any other pragmas were issued, namely, the normal severity of the message as modified by any command-line options. The following example suppresses the `declared but never referenced` warning of `foo()`:

```
#pragma nv_diag_suppress 177 // "declared but never referenced"
void foo() {
    int i = 0;
}

#pragma nv_diag_default 177
void bar() {
    int i = 0;
}
```

The following pragmas may be used to save and restore the current diagnostic pragma state:

```
#pragma nv_diagnostic push
#pragma nv_diagnostic pop
```

Examples:

```
#pragma nv_diagnostic push
#pragma nv_diag_suppress 177 // "declared but never referenced"
void foo() {
    int i = 0;
}
```

(continues on next page)

(continued from previous page)

```
#pragma nv_diagnostic pop
void bar() {
    int i = 0; // raise a warning
}
```

Note that these directives only affect the nvcc CUDA front-end compiler. They have no effect on the host compiler.

nvcc defines the macro `__NVCC_DIAG_PRAGMA_SUPPORT__` when diagnostic pragmas are supported.

5.4.11. Warp Matrix Functions

C++ warp matrix operations leverage Tensor Cores to accelerate matrix problems of the form `D=A*B+C`. These operations are supported on mixed-precision floating point data for devices of compute capability 7.0 or higher. This requires co-operation from all threads in a *warp*. In addition, these operations are allowed in conditional code only if the condition evaluates identically across the entire *warp*, otherwise the code execution is likely to hang.

5.4.11.1 Description

All following functions and types are defined in the namespace `nvcuda::wmma`. Sub-byte operations are considered preview, i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases. This extra functionality is defined in the `nvcuda::wmma::experimental` namespace.

```
template<typename Use, int m, int n, int k, typename T, typename Layout=void>
↪class fragment;

void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm);
void load_matrix_sync(fragment<...> &a, const T* mptr, unsigned ldm, layout_t
↪layout);
void store_matrix_sync(T* mptr, const fragment<...> &a, unsigned ldm, layout_
↪t layout);
void fill_fragment(fragment<...> &a, const T& v);
void mma_sync(fragment<...> &d, const fragment<...> &a, const fragment<...> &
↪b, const fragment<...> &c, bool satf=false);
```

`fragment`

An overloaded class containing a section of a matrix distributed across all threads in the warp. The mapping of matrix elements into `fragment` internal storage is unspecified and subject to change in future architectures.

Only certain combinations of template arguments are allowed. The first template parameter specifies how the fragment will participate in the matrix operation. Acceptable values for `Use` are:

- `matrix_a` when the fragment is used as the first multiplicand, `A`,
- `matrix_b` when the fragment is used as the second multiplicand, `B`, or
- `accumulator` when the fragment is used as the source or destination accumulators (`C` or `D`, respectively).

  The `m`, `n` and `k` sizes describe the shape of the warp-wide matrix tiles participating in the multiply-accumulate operation. The dimension of each tile depends on its role. For `matrix_a` the tile takes dimension `m x k`; for `matrix_b` the dimension is `k x n`, and `accumulator` tiles are `m x n`.

The data type, T, may be `double`, `float`, `__half`, `__nv_bfloat16`, `char`, or `unsigned char` for multiplicands and `double`, `float`, `int`, or `__half` for accumulators. As documented in *Element Types and Matrix Sizes*, limited combinations of accumulator and multiplicand types are supported. The Layout parameter must be specified for `matrix_a` and `matrix_b` fragments. `row_major` or `col_major` indicate that elements within a matrix row or column are contiguous in memory, respectively. The `Layout` parameter for an `accumulator` matrix should retain the default value of `void`. A row or column layout is specified only when the accumulator is loaded or stored as described below.

`load_matrix_sync`
Waits until all warp lanes have arrived at load_matrix_sync and then loads the matrix fragment a from memory. `mptr` must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. `ldm` describes the stride in elements between consecutive rows (for row major layout) or columns (for column major layout) and must be a multiple of 8 for `__half` element type or multiple of 4 for `float` element type. (i.e., multiple of 16 bytes in both cases). If the fragment is an `accumulator`, the `layout` argument must be specified as either `mem_row_major` or `mem_col_major`. For `matrix_a` and `matrix_b` fragments, the layout is inferred from the fragment's `layout` parameter. The values of `mptr`, `ldm`, `layout` and all template parameters for a must be the same for all threads in the warp. This function must be called by all threads in the warp, or the result is undefined.

`store_matrix_sync`
Waits until all warp lanes have arrived at store_matrix_sync and then stores the matrix fragment a to memory. `mptr` must be a 256-bit aligned pointer pointing to the first element of the matrix in memory. `ldm` describes the stride in elements between consecutive rows (for row major layout) or columns (for column major layout) and must be a multiple of 8 for `__half` element type or multiple of 4 for `float` element type. (i.e., multiple of 16 bytes in both cases). The layout of the output matrix must be specified as either `mem_row_major` or `mem_col_major`. The values of `mptr`, `ldm`, `layout` and all template parameters for a must be the same for all threads in the warp.

`fill_fragment`
Fill a matrix fragment with a constant value v. Because the mapping of matrix elements to each fragment is unspecified, this function is ordinarily called by all threads in the warp with a common value for v.

`mma_sync`
Waits until all warp lanes have arrived at mma_sync, and then performs the warp-synchronous matrix multiply-accumulate operation `D=A*B+C`. The in-place operation, `C=A*B+C`, is also supported. The value of `satf` and template parameters for each matrix fragment must be the same for all threads in the warp. Also, the template parameters `m`, `n` and `k` must match between fragments `A`, `B`, `C` and `D`. This function must be called by all threads in the warp, or the result is undefined.

If `satf` (saturate to finite value) mode is `true`, the following additional numerical properties apply for the destination accumulator:

- If an element result is +Infinity, the corresponding accumulator will contain `+MAX_NORM`
- If an element result is -Infinity, the corresponding accumulator will contain `-MAX_NORM`
- If an element result is NaN, the corresponding accumulator will contain `+0`

Because the map of matrix elements into each thread's `fragment` is unspecified, individual matrix elements must be accessed from memory (shared or global) after calling `store_matrix_sync`. In the special case where all threads in the warp will apply an element-wise operation uniformly to all fragment elements, direct element access can be implemented using the following `fragment` class members.

```
enum fragment<Use, m, n, k, T, Layout>::num_elements;
T fragment<Use, m, n, k, T, Layout>::x[num_elements];
```

As an example, the following code scales an `accumulator` matrix tile by half.

```
wmma::fragment<wmma::accumulator, 16, 16, 16, float> frag;
float alpha = 0.5f; // Same value for all threads in warp
/*...*/
for(int t=0; t<frag.num_elements; t++)
frag.x[t] *= alpha;
```

5.4.11.2 Alternate Floating Point

Tensor Cores support alternate types of floating point operations on devices with compute capability 8.0 and higher.

`__nv_bfloat16`

This data format is an alternate fp16 format that has the same range as f32 but reduced precision (7 bits). You can use this data format directly with the `__nv_bfloat16` type available in `cuda_bf16.h`. Matrix fragments with `__nv_bfloat16` data types are required to be composed with accumulators of `float` type. The shapes and operations supported are the same as with `__half`.

`tf32`

This data format is a special floating-point format supported by Tensor Cores, with the same range as f32 and reduced precision (>=10 bits). The internal layout of this format is implementation-defined. To use this floating-point format with WMMA operations, the input matrices must be manually converted to tf32 precision.

To facilitate conversion, a new intrinsic `__float_to_tf32` is provided. While the input and output arguments to the intrinsic are of `float` type, the output will be `tf32` numerically. This new precision is intended to be used with Tensor Cores only, and if mixed with other `float`type operations, the precision and range of the result will be undefined.

Once an input matrix (`matrix_a` or `matrix_b`) is converted to tf32 precision, the combination of a `fragment` with `precision::tf32` precision, and a data type of `float` to `load_matrix_sync` will take advantage of this new capability. Both the accumulator fragments must have `float` data types. The only supported matrix size is 16x16x8 (m-n-k).

The elements of the fragment are represented as `float`, hence the mapping from `element_type<T>` to `storage_element_type<T>` is:

```
precision::tf32 -> float
```

5.4.11.3 Double Precision

Tensor Cores support double-precision floating point operations on devices with compute capability 8.0 and higher. To use this new functionality, a `fragment` with the `double` type must be used. The `mma_sync` operation will be performed with the .rn (rounds to nearest even) rounding modifier.

5.4.11.4 Sub-byte Operations

Sub-byte WMMA operations provide a way to access the low-precision capabilities of Tensor Cores. They are considered a preview feature i.e. the data structures and APIs for them are subject to change and may not be compatible with future releases. This functionality is available via the `nvcuda::wmma::experimental` namespace:

```
namespace experimental {
    namespace precision {
        struct u4; // 4-bit unsigned
        struct s4; // 4-bit signed
        struct b1; // 1-bit
   }
    enum bmmaBitOp {
        bmmaBitOpXOR = 1, // compute_75 minimum
        bmmaBitOpAND = 2  // compute_80 minimum
    };
    enum bmmaAccumulateOp { bmmaAccumulateOpPOPC = 1 };
}
```

For 4 bit precision, the APIs available remain the same, but you must specify `experimental::precision::u4` or `experimental::precision::s4` as the fragment data type. Since the elements of the fragment are packed together, `num_storage_elements` will be smaller than `num_elements` for that fragment. The `num_elements` variable for a sub-byte fragment, hence returns the number of elements of sub-byte type `element_type<T>`. This is true for single bit precision as well, in which case, the mapping from `element_type<T>` to `storage_element_type<T>` is as follows:

```
experimental::precision::u4 -> unsigned (8 elements in 1 storage element)
experimental::precision::s4 -> int (8 elements in 1 storage element)
experimental::precision::b1 -> unsigned (32 elements in 1 storage element)
T -> T  //all other types
```

The allowed layouts for sub-byte fragments is always `row_major` for `matrix_a` and `col_major` for `matrix_b`.

For sub-byte operations the value of `ldm` in `load_matrix_sync` should be a multiple of 32 for element type `experimental::precision::u4` and `experimental::precision::s4` or a multiple of 128 for element type `experimental::precision::b1` (i.e., multiple of 16 bytes in both cases).

Note

Support for the following variants for MMA instructions is deprecated and will be removed in sm_90:

- `experimental::precision::u4`
- `experimental::precision::s4`
- `experimental::precision::b1` with bmmaBitOp set to bmmaBitOpXOR

bmma_sync

Waits until all warp lanes have executed bmma_sync, and then performs the warp-synchronous bit matrix multiply-accumulate operation `D = (A op B) + C`, where op consists of a logical operation `bmmaBitOp` followed by the accumulation defined by `bmmaAccumulateOp`. The available operations are:

bmmaBitOpXOR, a 128-bit XOR of a row in matrix_a with the 128-bit column of matrix_b

bmmaBitOpAND, a 128-bit AND of a row in matrix_a with the 128-bit column of matrix_b, available on devices with compute capability 8.0 and higher.

The accumulate op is always bmmaAccumulateOpPOPC which counts the number of set bits.

5.4.11.5 Restrictions

The special format required by tensor cores may be different for each major and minor device architecture. This is further complicated by threads holding only a fragment (opaque architecture-specific ABI data structure) of the overall matrix, with the developer not allowed to make assumptions on how the individual parameters are mapped to the registers participating in the matrix multiply-accumulate.

Since fragments are architecture-specific, it is unsafe to pass them from function A to function B if the functions have been compiled for different link-compatible architectures and linked together into the same device executable. In this case, the size and layout of the fragment will be specific to one architecture and using WMMA APIs in the other will lead to incorrect results or potentially, corruption.

An example of two link-compatible architectures, where the layout of the fragment differs, is sm_70 and sm_75.

```
fragA.cu: void foo() { wmma::fragment<...> mat_a; bar(&mat_a); }
fragB.cu: void bar(wmma::fragment<...> *mat_a) { // operate on mat_a }
```

```
// sm_70 fragment layout
$> nvcc -dc -arch=compute_70 -code=sm_70 fragA.cu -o fragA.o
// sm_75 fragment layout
$> nvcc -dc -arch=compute_75 -code=sm_75 fragB.cu -o fragB.o
// Linking the two together
$> nvcc -dlink -arch=sm_75 fragA.o fragB.o -o frag.o
```

This undefined behavior might also be undetectable at compilation time and by tools at runtime, so extra care is needed to make sure the layout of the fragments is consistent. This linking hazard is most likely to appear when linking with a legacy library that is both built for a different link-compatible architecture and expecting to be passed a WMMA fragment.

Note that in the case of weak linkages (for example, a CUDA C++ inline function), the linker may choose any available function definition which may result in implicit passes between compilation units.

To avoid these sorts of problems, the matrix should always be stored out to memory for transit through external interfaces (e.g. wmma::store_matrix_sync(dst, …);) and then it can be safely passed to bar() as a pointer type [e.g. float *dst].

Note that since sm_70 can run on sm_75, the above example sm_75 code can be changed to sm_70 and correctly work on sm_75. However, it is recommended to have sm_75 native code in your application when linking with other sm_75 separately compiled binaries.

5.4.11.6 Element Types and Matrix Sizes

Tensor Cores support a variety of element types and matrix sizes. The following table presents the various combinations of matrix_a, matrix_b and accumulator matrix supported:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
__half	__half	__half	8x32x16
unsigned char	unsigned char	int	16x16x16
unsigned char	unsigned char	int	32x8x16
unsigned char	unsigned char	int	8x32x16
signed char	signed char	int	16x16x16
signed char	signed char	int	32x8x16
signed char	signed char	int	8x32x16

Alternate floating-point support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__nv_bfloat16	__nv_bfloat16	float	16x16x16
__nv_bfloat16	__nv_bfloat16	float	32x8x16
__nv_bfloat16	__nv_bfloat16	float	8x32x16
precision::tf32	precision::tf32	float	16x16x8

Double Precision Support:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
double	double	double	8x8x4

Experimental support for sub-byte operations:

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
precision::u4	precision::u4	int	8x8x32
precision::s4	precision::s4	int	8x8x32
precision::b1	precision::b1	int	8x8x128

5.4.11.7 Example

The following code implements a 16x16x16 matrix multiplication in a single warp.

```
#include <mma.h>
using namespace nvcuda;

__global__ void wmma_ker(half *a, half *b, float *c) {
   // Declare the fragments
   wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
   wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
   wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

   // Initialize the output to zero
   wmma::fill_fragment(c_frag, 0.0f);

   // Load the inputs
   wmma::load_matrix_sync(a_frag, a, 16);
   wmma::load_matrix_sync(b_frag, b, 16);

   // Perform the matrix multiplication
   wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

   // Store the output
   wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

5.5. Floating-Point Computation

5.5.1. Floating-Point Introduction

Since the adoption of the IEEE-754 Standard for Binary Floating-Point Arithmetic in 1985, virtually all mainstream computing systems, including NVIDIA's CUDA architectures, have implemented the standard. The IEEE-754 standard specifies how the results of floating-point arithmetic should be approximated.

To get accurate results and achieve the highest performance with the required precision, it is important to consider many aspects of floating-point behavior. This is particularly important in a heterogeneous computing environment where operations are performed on different types of hardware.

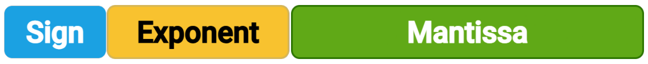
The following sections review the basic properties of floating-point computation and cover Fused Multiply-Add (FMA) operations and the dot product. These examples illustrate how different implementation choices affect accuracy.

5.5.1.1 Floating-Point Format

Floating-point format and functionality are defined in the IEEE-754 Standard.

The standard mandates that binary floating-point data be encoded on three fields:

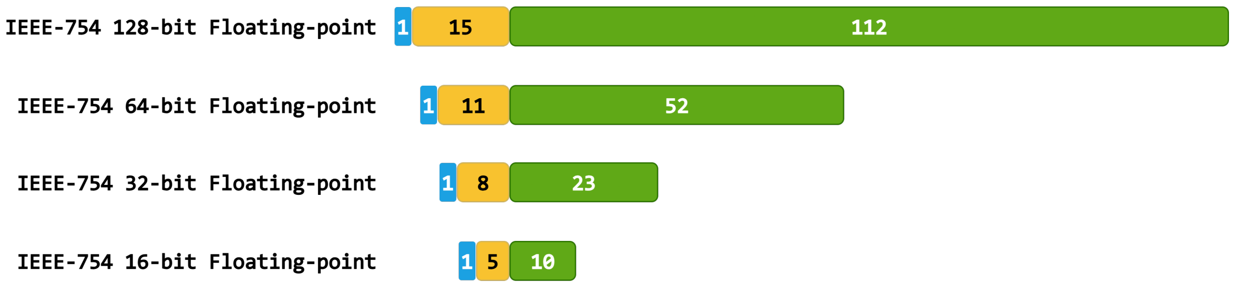
- **Sign**: one bit to indicate a positive or negative number.
- **Exponent**: encodes the exponent offset by a numeric bias in base 2.
- **Significand** (also called *mantissa* or *fraction*): encodes the fractional value of the number.



The latest IEEE-754 standard defines the encodings and properties of the following binary formats:

- 16-bit, also known as half-precision, corresponding to the `__half` data type in CUDA.
- 32-bit, also known as single-precision, corresponding to the `float` data type in C, C++, and CUDA.
- 64-bit, also known as double-precision, corresponding to the `double` data type in C, C++, and CUDA.
- 128-bit, also known as quad-precision, corresponding to the `__float128` or `_Float128` data types in CUDA.

These types have the following bit lengths:

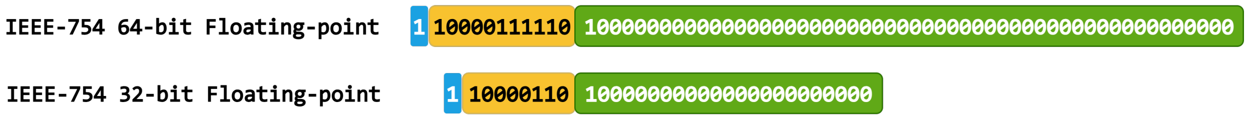

IEEE-754 128-bit Floating-point
1
15
112
IEEE-754 64-bit Floating-point
1
11
52
IEEE-754 32-bit Floating-point
1
8
23
IEEE-754 16-bit Floating-point
1
5
10

The numeric value associated with floating-point encoding for *normal* values is computed as follows:

$$(-1)^{\text{sign}} \times 1.\text{mantissa} \times 2^{\text{exponent}-\text{bias}}$$

for *subnormal* values, the leading 1 in the formula is absent.

The exponents are biased by 127 and 1023 for single- and double-precision, respectively. The integral part of $1.$ is implicit in the fraction.
For example, the value $-192 = (-1)^1 \times 2^7 \times 1.5$, and is encoded as a negative sign, an exponent of 7, and a fractional part 0.5. Hence the exponent 7 is represented by bit strings with values `7 + 127 = 134 = 10000110` for `float` and `7 + 1023 = 1030 = 10000111110` for `double`. The mantissa `0.5 = 2^-1` is represented by a binary value with 1 in the first position. The binary encoding of -192 in single-precision and double-precision is shown in the following figure:



Since the fraction field uses a limited number of bits, not all real numbers can be represented exactly. For instance, the binary representation of the mathematical value of the fraction $2/3$ is `0.10101010...`, which has an infinite number of bits after the binary point. Therefore, $2/3$ must be rounded before it can be represented as a floating-point number with limited precision. The rounding rules and modes are specified in IEEE-754. The most frequently used mode is *round-to-nearest-or-even*, abbreviated round-to-nearest.

5.5.1.2 Normal and Subnormal Values

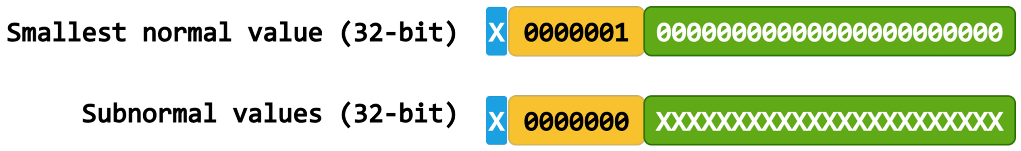
Any floating-point value that can be represented with at least one bit set in the exponent is called *normal*.

An important aspect of floating-point normal values is the wide gap between the smallest representable non-zero floating-point number, `FLT_MIN`, and zero. This gap is much wider than the gap between `FLT_MIN` and the second-smallest representable non-zero floating-point number.

Floating-point *subnormal* numbers, also called *denormals*, were introduced to address this issue. A subnormal floating-point value is represented with all bits in the exponent set to zero and at least one bit set in the significand. Subnormals are a required part of the IEEE-754 floating-point standard.

Subnormal numbers allow for a gradual loss of precision as an alternative to sudden rounding toward zero. However, subnormal numbers are computationally more expensive. Therefore, applications that don't require strict accuracy may choose to avoid them to improve performance. The `nvcc` compiler allows disabling subnormal numbers by setting the `-ftz=true` option (flush-to-zero), which is also included in `--use_fast_math`.

A simplified visualization of the encoding of the smallest normal value and subnormal values in single-precision is shown in the following figure:



where X represents both `0` and 1.

5.5.1.3 Special Values

The IEEE-754 standard defines three special values for floating-point numbers:

Zero:

- Mathematical zero.
- Note that there are two possible representations of floating-point zero: `+0` and `-0`. This differs from the representation of integer zero.
- `+0 == -0` evaluates to `true`.
- `0` is encoded with all bits set to `0` in the exponent and significand.

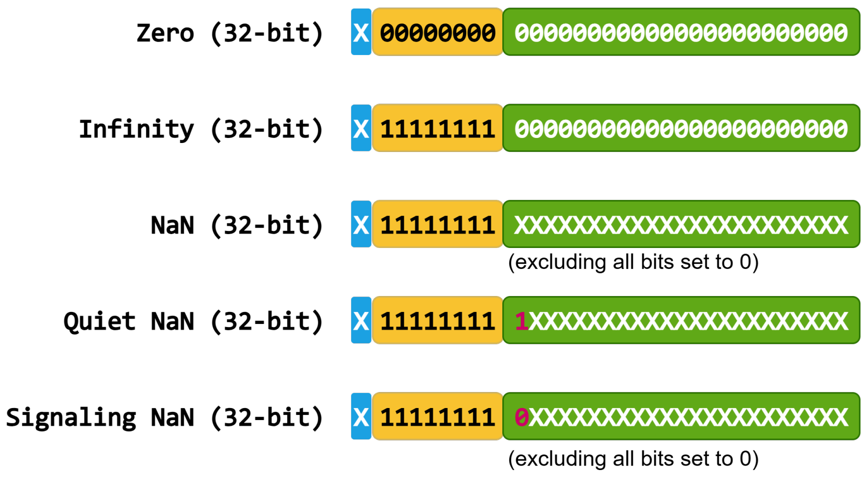
Infinity:

- Floating-point numbers behave according to saturation arithmetic, in which operations that overflow the representable range result in `+Infinity` or `-Infinity`.
- Infinity is encoded with all bits in the exponent set to 1 and all bits in the significand set to `0`. There are exactly two encodings for infinity values.
- Any arithmetic operation that applies a finite number to infinity will result in infinity, except for division by zero and multiplication by zero, which result in NaN.

Not-a-Number (NaN):

- NaN is a special symbol that represents an undefined or non-representable value. Common examples are `0.0 / 0.0`, `sqrt(-1.0)`, or `+Inf - Inf`.
- NaN is encoded with all bits in the exponent set to 1 and any bit pattern in the significand, except for all bits set to 0. There are $2^{\text{mantissa}+1} - 2$ possible encodings.
- Any arithmetic operation involving a NaN will result in NaN.
- Any comparison operation involving a NaN will result in `false`, including `NaN == NaN` (non-reflexive).
- NaNs are provided in two forms:
  - Quiet NaNs `qNaN` are used to propagate errors resulting from invalid operations or values. Invalid arithmetic operations generally produce a quiet NaN. They are encoded with the most significant bit of the significand set to 1.
  - Signaling NaNs `sNaN` are designed to raise an invalid-operation exception. Signaling NaNs are generally explicitly created. They are encoded with the most significant bit of the significand set to `0`.
  - The exact bit patterns for Quiet and Signaling NaNs are implementation-defined. CUDA provides the cuda::std::numeric_limits<T>::quiet_NaN and cuda::std::numeric_limits<T>::signaling_NaN constants to get their special values.

A simplified visualization of the encodings of special values is shown in the following figure:


Zero (32-bit)
X 00000000 00000000000000000000000
Infinity (32-bit)
X 11111111 00000000000000000000000
NaN (32-bit)
X 11111111 XXXXXXXXXXXXXXXXXXXXXXX
(excluding all bits set to 0)
Quiet NaN (32-bit)
X 11111111 1XXXXXXXXXXXXXXXXXXXXXX
Signaling NaN (32-bit)
X 11111111 0XXXXXXXXXXXXXXXXXXXXXX
(excluding all bits set to 0)

where X represents both `0` and 1.

5.5.1.4 Associativity

It is important to note that the rules and properties of mathematical arithmetic do not directly apply to floating-point arithmetic due to its limited precision. The example below shows single-precision values A, B, and C and the exact mathematical value of their sum computed using different associativity.

$$A = 2^1 \times 1.00000000000000000000001$$
$$B = 2^0 \times 1.00000000000000000000001$$
$$C = 2^3 \times 1.00000000000000000000001$$
$$(A + B) + C = 2^3 \times 1.01100000000000000000001011$$
$$A + (B + C) = 2^3 \times 1.01100000000000000000001011$$

Mathematically, $(A + B) + C$ is equal to $A + (B + C)$.

Let rn(x) denote one rounding step on x. Performing the same computations in single-precision floating-point arithmetic in round-to-nearest mode according to IEEE-754, we obtain:

$$\begin{aligned}
A + B &= 2^1 \times 1.1000000000000000000000110000\ldots \\
\text{rn}(A + B) &= 2^1 \times 1.10000000000000000000010 \\
B + C &= 2^3 \times 1.0010000000000000000000100100\ldots \\
\text{rn}(B + C) &= 2^3 \times 1.00100000000000000000001 \\
A + B + C &= 2^3 \times 1.0110000000000000000000101100\ldots \\
\text{rn}\big(\text{rn}(A + B) + C\big) &= 2^3 \times 1.01100000000000000000010 \\
\text{rn}\big(A + \text{rn}(B + C)\big) &= 2^3 \times 1.01100000000000000000001
\end{aligned}$$

For reference, the exact mathematical results are also computed above. The results computed according to IEEE-754 differ from the exact mathematical results. Additionally, the results corresponding to the sums rn$(\text{rn}(A+B)+C)$ and rn$(A+\text{rn}(B+C))$ differ from each other. In this case, rn$(A+\text{rn}(B+C))$ is closer to the correct mathematical result than rn$(\text{rn}(A + B) + C)$.

This example shows that seemingly identical computations can produce different results, even when all basic operations comply with IEEE-754.

5.5.1.5 Fused Multiply-Add (FMA)

The Fused Multiply-Add (FMA) operation computes the result with only one rounding step. Without the FMA, the result would require two rounding steps: one for multiplication and one for addition. Because the FMA uses only one rounding step, it produces a more accurate result.

The Fused Multiply-Add operation can affect the propagation of NaNs differently than two separate operations. However, FMA NaN handling is not universally identical across all targets. Different implementations with multiple NaN operands may prefer a quiet NaN or propagate one operand's payload. Additionally, IEEE-754 does not strictly mandate a deterministic payload selection order when multiple NaN operands are present. NaNs may also occur in intermediate computations, for example, $\infty \times 0 + 1$ or $1 \times \infty - \infty$, resulting in an implementation-defined NaN payload.

For clarity, first consider an example using decimal arithmetic to illustrate how the FMA operation works. We will compute $x^2 - 1$ using five total digits of precision, with four digits after the decimal point.

- For $x = 1.0008$, the correct mathematical result is $x^2 - 1 = 1.60064 \times 10^{-4}$. The closest number using only four digits after the decimal point is 1.6006×10^{-4}.
- The Fused Multiply-Add operation achieves the correct result using only one rounding step rn$(x \times x - 1) = 1.6006 \times 10^{-4}$.
- The alternative is to compute the multiply and add steps separately. $x^2 = 1.00160064$ translates to rn$(x \times x) = 1.0016$. The final result is rn$(\text{rn}(x \times x) - 1) = 1.6000 \times 10^{-4}$.

Rounding the multiply and add separately yields a result that is off by 0.00064. The corresponding FMA computation is wrong by only 0.00004 and its result is closest to the correct mathematical answer. The results are summarized below:

$$\begin{aligned} x &= 1.0008 \\ x^2 &= 1.00160064 \\ x^2 - 1 &= 1.60064 \times 10^{-4} && \text{true value} \\ \text{rn}\big(x^2 - 1\big) &= 1.6006 \times 10^{-4} && \text{fused multiply-add} \\ \text{rn}\big(x^2\big) &= 1.0016 \\ \text{rn}\big(\text{rn}(x^2) - 1\big) &= 1.6000 \times 10^{-4} && \text{multiply, then add} \end{aligned}$$

Below is another example, using binary single precision values:

$$\begin{aligned} A &= 2^0 \times 1.00000000000000000000001 \\ B &= -2^0 \times 1.00000000000000000000010 && \text{fused multiply-add} \\ \text{rn}\big(A \times A + B\big) &= 2^{-46} \times 1.00000000000000000000000 && \text{multiply, then add} \\ \text{rn}\big(\text{rn}(A \times A) + B\big) &= 0 \end{aligned}$$

- Computing multiplication and addition separately results in the loss of all bits of precision, yielding 0.
- Computing the FMA, on the other hand, provides a result equal to the mathematical value.

Fused multiply-add helps prevent loss of precision during subtractive cancellation. Subtractive cancellation occurs when quantities of similar magnitude with opposite signs are added. In this case, many of the leading bits cancel out, resulting in fewer meaningful bits. The fused multiply-add computes a double-width product during multiplication. Thus, even if subtractive cancellation occurs during addition, there are enough valid bits remaining in the product to yield a precise result.

Fused Multiply-Add Support in CUDA:

CUDA provides the Fused Multiply-Add operation in several ways for both `float` and `double` data types:

- `x * y + z` when compiled with the flags `-fmad=true` or `--use_fast_math`.
- `fma(x, y, z)` and `fmaf(x, y, z)` C Standard Library functions.
- `__fmaf_[rd, rn, ru, rz]`, `__fmaf_ieee_[rd, rn, ru, rz]`, and `__fma_[rd, rn, ru, rz]` CUDA mathematical intrinsic functions.
- `cuda::std::fma(x, y, z)` and `cuda::std::fmaf(x, y, z)` CUDA C++ Standard Library functions.

Fused Multiply-Add Support on Host Platforms:

Whether to use the fused operation depends on the availability of the operation on the platform and how the code is compiled. It is important to understand the host platform's support for Fused Multiply-Add when comparing CPU and GPU results.

- Compiler flags and Fused Multiply-Add hardware support:
  - `-mfma` with GCC and Clang, `-Mfma` with NVC++, and `/fp:contract` with Microsoft Visual Studio.

 - x86 platforms with the AVX2 ISA, for example, code compiled with the `-mavx2` flag using GCC or Clang, and `/arch:AVX2` with Microsoft Visual Studio.
 - Arm64 (AArch64) platforms with Advanced SIMD (Neon) ISA.
- `fma(x, y, z)` and `fmaf(x, y, z)` C Standard Library functions.
- `std::fma(x, y, z)` and `std::fmaf(x, y, z)` C++ Standard Library functions.
- `cuda::std::fma(x, y, z)` and `cuda::std::fmaf(x, y, z)` CUDA C++ Standard Library functions.

5.5.1.6 Dot Product Example

Consider the problem of finding the dot product of two short vectors $\vec{a}$ and $\vec{b}$ both with four elements.

$$\vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} \qquad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} \qquad \vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3 + a_4 b_4$$

Although this operation is easy to write down mathematically, implementing it in software involves several alternatives that could lead to slightly different results. All of the strategies presented here use operations that are fully compliant with IEEE-754.

Example Algorithm 1: The simplest way to compute the dot product is to use a sequential sum of products, keeping the multiplications and additions separate.

The final result can be represented as $((((a_1 \times b_1) + (a_2 \times b_2)) + (a_3 \times b_3)) + (a_4 \times b_4))$.

Example Algorithm 2: Compute the dot product sequentially using fused multiply-add.

The final result can be represented as $(a_4 \times b_4) + ((a_3 \times b_3) + ((a_2 \times b_2) + (a_1 \times b_1 + 0)))$.

Example Algorithm 3: Compute the dot product using a divide-and-conquer strategy. First, we find the dot products of the first and second halves of the vectors. Then, we combine these results using addition. This algorithm is called the "parallel algorithm" because the two subproblems can be computed in parallel since they are independent of each other. However, the algorithm does not require a parallel implementation; it can be implemented with a single thread.

The final result can be represented as $((a_1 \times b_1) + (a_2 \times b_2)) + ((a_3 \times b_3) + (a_4 \times b_4))$.

5.5.1.7 Rounding

The IEEE-754 standard requires support for several operations. These include arithmetic operations such as addition, subtraction, multiplication, division, square root, fused multiply-add, finding the remainder, conversion, scaling, sign, and comparison operations. The results of these operations are guaranteed to be consistent across all implementations of the standard for a given format and rounding mode.

Rounding Modes

The IEEE-754 standard defines four rounding modes: *round-to-nearest*, *round towards positive*, *round towards negative*, and *round towards zero*. CUDA supports all four modes. By default, operations use *round-to-nearest*. *Intrinsic mathematical functions* can be used to select other rounding modes for individual operations.

Rounding Mode	Interpretation
`rn`	Round to nearest, ties to even
`rz`	Round towards zero
`ru`	Round towards ∞
`rd`	Round towards $-\infty$

5.5.1.8 Notes on Host/Device Computation Accuracy

The accuracy of a floating-point computation result is affected by several factors. This section summarizes important considerations for achieving reliable results in floating-point computations. Some of these aspects have been described in greater detail in previous sections.

These aspects are also important when comparing the results between CPU and GPU. Differences between host and device execution must be interpreted carefully. The presence of differences does not necessarily mean the GPU's result is incorrect or that there is a problem with the GPU.

Associativity:

Floating-point addition and multiplication in finite precision are not *associative* because they often result in mathematical values that cannot be directly represented in the target format, requiring rounding. The order in which these operations are evaluated affects how rounding errors accumulate and can significantly alter the final result.

Fused Multiply-Add:

Fused Multiply-Add computes $a \times b + c$ in a single operation, resulting in greater accuracy and a faster execution time. The accuracy of the final result can be affected by its use. Fused Multiply-Add relies on hardware support and can be enabled either explicitly by calling the related function or implicitly through compiler optimization flags.

Precision:

Increasing the floating-point precision can potentially improve the accuracy of the results. Higher precision reduces loss of significance and enables the representation of a wider range of values. However, higher precision types have lower throughput and consumes more registers. Additionally, using them to explicitly store input and output increases memory usage and data movement.

Compiler Flags and Optimizations:

All major compilers provide a variety of optimization flags to control the behavior of floating-point operations.

- The highest optimization level for GCC (-O3), Clang (-O3), nvcc (-O3), and Microsoft Visual Studio (/O2) does not affect floating-point semantics. However, inlining, loop unrolling, vectorization, and common subexpression elimination could affect the results. The NVC++ compiler also requires the flags `-Kieee -Mnofma` for IEEE-754-compliant semantics.
- Refer to the GCC, Clang, Microsoft Visual Studio Compiler, nvc++, and Arm C/C++ compiler documentation for detailed information about options that affect floating-point behavior.

Library Implementations:

Functions defined outside the IEEE-754 standard are not guaranteed to be correctly rounded and depend on implementation-defined behavior. Therefore, the results may differ across different platforms, including between host, device, and different device architectures.

Deterministic Results:

A deterministic result refers to computing the same bit-wise numerical outputs every time when run with the same inputs under the same specified conditions. Such conditions include:

- Hardware dependencies, such as execution on the same CPU processor or GPU device.
- Compiler aspects, such as the version of the compiler and the compiler flags.
- Run-time conditions that affect the computation, such as *rounding mode* or environment variables.
- Identical inputs to the computation.
- Thread configuration, including the number of threads involved in the computation and their organization, for example block and grid size.
- The ordering of *arithmetic atomic operations* depends on hardware scheduling which can vary between runs.

Taking Advantage of the CUDA Libraries:

The CUDA Math Libraries, C Standard Library Mathematical functions, and C++ Standard Library Mathematical functions are designed to boost developer productivity for common functionalities, particularly for floating-point math and numerics-intensive routines. These functionalities provide a consistent high-level interface, are optimized, and are widely tested across platforms and edge cases. Users are encouraged to take full advantage of these libraries and avoid tedious manual reimplementations.

5.5.2. Floating-Point Data Types

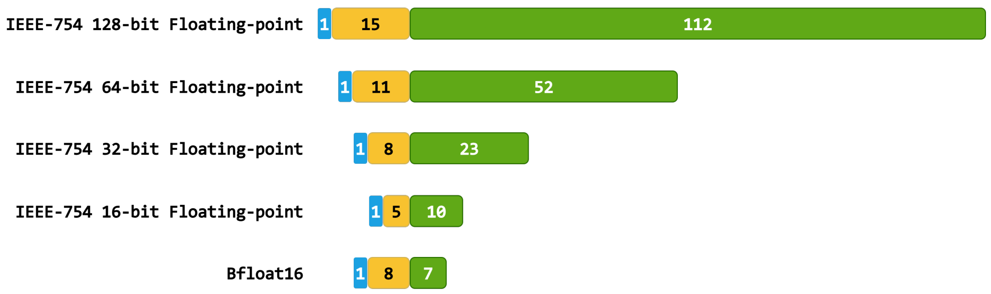
CUDA supports the Bfloat16, half-, single-, double-, and quad-precision floating-point data types. The following table summarizes the supported floating-point data types in CUDA and their requirements.

Table 43: Supported Floating-Point Types

Precision / Name	Data Type	IEEE-754	Header / Built-in	Requirements
Bfloat1	`__nv_bfl`	□	`<cuda_bf16.h>`	Compute Capability 8.0 or higher.
Half Precision	`__half`	□	`<cuda_fp16.h>`	
Single Precision	`float`	□	Built-in	
Double Precision	`double`	□	Built-in	
Quad Precision	`__float12`	□	Built-in `<crt/device_fp128_functions.h>` for mathematical functions	Host compiler support and Compute Capability 10.0 or higher. The C or C++ spelling, `_Float128` and `__float128` respectively, also depends on the host compiler support.

CUDA also supports TensorFloat-32 (TF32), microscaling (MX) floating-point types, and other lower precision numerical formats that are not intended for general-purpose computation, but rather for specialized purposes involving tensor cores. These include 4-, 6-, and 8-bit floating-point types. See the CUDA Math API for more details.

The following figure reports the mantissa and exponent sizes of the supported floating-point data types.



The following table reports the ranges of the supported floating-point data types.

Table 44: Supported Floating-Point Types Properties

Precision / Name	Largest Value		Smallest Positive Value		Smallest Positive Denormal	Epsilon
Bfloat16	$\approx 2^{128}$	$\approx 3.39 \cdot 10^{38}$	2^{-126}	$\approx 1.18 \cdot 10^{-38}$	2^{-133}	2^{-7}
Half Precision	$\approx 2^{16}$	65504	2^{-14}	$\approx 6.1 \cdot 10^{-5}$	2^{-24}	2^{-10}
Single Precision	$\approx 2^{128}$	$\approx 3.39 \cdot 10^{38}$	2^{-126}	$\approx 1.18 \cdot 10^{-38}$	2^{-149}	2^{-23}
Double Precision	$\approx 2^{1024}$	$\approx 1.8 \cdot 10^{308}$	2^{-1022}	$\approx 2.22 \cdot 10^{-308}$	2^{-1074}	2^{-52}
Quad Precision	$\approx 2^{16384}$	$\approx 1.19 \cdot 10^{4932}$	2^{-16382}	$\approx 3.36 \cdot 10^{-4032}$	2^{-16494}	2^{-112}

Hint

The *CUDA C++ Standard Library* provides `cuda::std::numeric_limits` in the `<cuda/std/limits>` header to query the properties and the ranges of the supported floating-point types, including microscaling formats (MX). See the C++ reference for the list of queryable properties.

Complex numbers support:

- The *CUDA C++ Standard Library* supports complex numbers with the cuda::std::complex type in the `<cuda/std/complex>` header. See also the libcu++ documentation for more details.
- CUDA also provides basic support for complex numbers with the `cuComplex` and `cuDoubleComplex` types in the `cuComplex.h` header.

5.5.3. CUDA and IEEE-754 Compliance

All GPU devices follow the IEEE 754-2019 standard for binary floating-point arithmetic with the following limitations:

- There is no dynamically configurable rounding mode; however, most of the operations support multiple constant IEEE rounding modes, selectable via specifically named *device intrinsics functions*.
- There is no mechanism to detect floating-point exceptions, so all operations behave as if IEEE-754 exceptions are always masked. If there is an exceptional event, the default masked response defined by IEEE-754 is delivered. For this reason, although signaling NaN `SNaN` encodings are supported, they are not signaling and are handled as quiet exceptions.
- Floating-point operations may alter the bit patterns of input NaN payloads. Operations such as absolute value and negation may also not comply with the IEEE 754 requirement, which could result in the sign of a NaN being updated in an implementation-defined manner.

To maximize the portability of results, users are recommended to use the default settings of the nvcc compiler's floating-point options: `-ftz=false`, `-prec-div=true`, and `-prec-sqrt=true`, and not use the `--use_fast_math` option. Note that floating-point expression re-associations and contractions are allowed by default, similarly to the `--fmad=true` option. See also the `nvcc` User Manual for a detailed description of these compilation flags.

The IEEE-754 and C/C++ language standards do not explicitly address the conversion of a floating-point value to an integer value in cases where the rounded-to-integer value falls outside the range of the target integer format. The clamping behavior to the range of GPU devices is delineated in the PTX ISA conversion instructions section. However, compiler optimizations may leverage the unspecified behavior clause when out-of-range conversion is not invoked directly via a PTX instruction, consequently resulting in undefined behavior and an invalid CUDA program. The CUDA Math documentation issues warnings to users on a per-function/intrinsic basis. For instance, consider the __double2int_rz() instruction. This may differ from how host compilers and library implementations behave.

Atomic Functions Denormals Behavior:

Atomic operations have the following behavior regarding floating-point denormals, regardless of the setting of the compiler flag `-ftz`:

- Atomic single-precision floating-point adds on global memory always operate in flush-to-zero mode, namely behave equivalent to PTX `add.rn.ftz.f32` semantic.
- Atomic single-precision floating-point adds on shared memory always operate with denormal support, namely behave equivalent to PTX `add.rn.f32` semantic.

5.5.4. CUDA and C/C++ Compliance

Floating-Point Exceptions:

Unlike the host implementation, the mathematical operators and functions supported in device code do not set the global `errno` variable nor report floating-point exceptions to indicate errors. Thus, if error diagnostic mechanisms are required, users should implement additional input and output screening for the functions.

Undefined Behavior with Floating-Point Operations:

Common conditions of undefined behavior for mathematical operations include:

- Invalid arguments to mathematical operators and functions:
  - Using an uninitialized floating-point variable.
  - Using a floating-point variable outside its lifetime.
  - Signed integer overflow.
  - Dereferencing an invalid pointer.
- Floating-point specific undefined behavior:
  - Converting a floating-point value to an integer type for which the result is not representable is undefined behavior. This also includes NaN and infinity.

Users are responsible for ensuring the validity of a CUDA program. Invalid arguments may result in undefined behavior and be subject to compiler optimizations.

Contrary to integer division by zero, floating-point division by zero is not undefined behavior and not subject to compiler optimizations; rather, it is implementation-specific behavior. C++ implementations that conform to IEC-60559 (IEEE-754), including CUDA, produce infinity. Note that invalid floating-point operations produce NaN and should not be misinterpreted as undefined behavior. Examples include zero divided by zero and infinity divided by infinity.

Floating-Point Literals Portability:

Both C and C++ allow for the representation of floating-point values in either decimal or hexadecimal notation. Hexadecimal floating-point literals, which are supported in C99 and C++17, denote a real value in scientific notation that can be precisely expressed in base-2. However, this does not guarantee that the literal will map to an actual value stored in a target variable (see the next paragraph). Conversely, a decimal floating-point literal may represent a numeric value that cannot be expressed in base-2.

According to the C++ standard rules, hexadecimal and decimal floating-point literals are rounded to the nearest representable value, larger or smaller, chosen in an implementation-defined manner. This rounding behavior may differ between the host and the device.

```
float f1 = 0.5f;    // 0.5, '0.5f' is a decimal floating-point literal
float f2 = 0x1p-1f; // 0.5, '0x1p-1f' is a hexadecimal floating-point literal
float f3 = 0.1f;
// f1, f2 are represented as 0 01111110 00000000000000000000000
// f3     is represented as  0 01111011 10011001100110011001101
```

The run-time and compile-time evaluations of the same floating-point expression are subject to the following portability issues:

- The run-time evaluation of a floating-point expression may be affected by the selected rounding mode, floating-point contraction (FMA) and reassociation compiler settings, as well as floating-point exceptions. Note that CUDA does not support floating-point exceptions and the *rounding mode* is always *round-to-nearest-ties-to-even*.
- The compiler may use a higher-precision internal representation for constant expressions.
- The compiler may perform optimizations, such as constant folding, constant propagation, and common subexpression elimination, which can lead to a different final value or comparison result.

C Standard Math Library Notes:

The host implementations of common mathematical functions are mapped to C Standard Math Library functions in a platform-specific way. These functions are provided by the host compiler and the respective host `libm`, if available.

- Functions not available from the host compilers are implemented in the `crt/math_functions.h` header file. For example, `erfinv()` is implemented there.
- Less common functions, such as `rhypot()` and `cyl_bessel_i0()`, are only available in the device code.

As previously mentioned, the host and device implementations of mathematical functions are independent. For more details on the behavior of these functions, please refer to the host implementation's documentation.

5.5.5. Floating-Point Functionality Exposure

The mathematical functions supported by CUDA are exposed through the following methods:

Built-in C/C++ language arithmetic operators:

- `x + y`, `x - y`, `x * y`, `x / y`, `x++`, `x--`, `+=`, `-=`, `*=`, `/=`.

- Support single-, double-, and quad-precision types, `float`, `double`, and `__float128`/`_Float128` respectively.
  - `__half` and `__nv_bfloat16` types are also supported by including the `<cuda_fp16.h>` and `<cuda_bf16.h>` headers, respectively.
  - `__float128`/`_Float128` type support relies on the host compiler and device compute capability, see the *Supported Floating-Point Types* table.
- They are available in both host and device code.
- Their behavior is affected by the `nvcc` optimization flags.

CUDA C++ Standard Library Mathematical functions:

- Expose the full set of C++ `<cmath>` header functions exposed through the `<cuda/std/cmath>` header and the `cuda::std::` namespace.
- Support IEEE-754 standard floating-point types, `__half`, `float`, `double`, `__float128`, as well as Bfloat16 `__nv_bfloat16`.
  - `__float128` support relies on the host compiler and device compute capability, see the *Supported Floating-Point Types* table.
- They are available in both host and device code.
- They often rely on the CUDA Math API functions. Therefore, there could be different levels of accuracy between the host and device code.
- Their behavior is affected by the `nvcc` *optimization flags*.
- A subset of functionalities is also supported on constant expressions, such as `constexpr` functions, in accordance with the C++23 and C++26 standard specifications.

CUDA C Standard Library Mathematical functions (CUDA Math API):

- Expose a subset of the C `<math.h>` header functions.
- Support single and double-precision types, `float` and `double` respectively.
  - They are available in both host and device code.
  - They don't require additional headers.
  - Their behavior is affected by the `nvcc` *optimization flags*.
- A subset of the `<math.h>` header functionalities is also available for `__half`, `__nv_bfloat16`, and `__float128`/`_Float128` types. These functions have names that resemble those of the C Standard Library.
  - `__half` and `__nv_bfloat16` types require the `<cuda_fp16.h>` and `<cuda_bf16.h>` headers, respectively. Their host and device code availability is defined on a per-function basis.
  - `__float128`/`_Float128` type support relies on the host compiler and device compute capability, see the *Supported Floating-Point Types* table. The related functions require the `crt/device_fp128_functions.h` header and they are only available in device code.
- They can have a different accuracy between host and device code.

Non-standard CUDA Mathematical functions (CUDA Math API):

- Expose mathematical functionalities that are not part of the C/C++ Standard Library.
- Mainly support single- and double-precision types, `float` and `double` respectively.
  - Their host and device code availability is defined on a per-function basis.

  - They don't require additional headers.
  - They can have a different accuracy between host and device code.
- `__nv_bfloat16`, `__half`, `__float128`/`_Float128` are supported for a limited set of functions.
  - `__half` and `__nv_bfloat16` types require the `<cuda_fp16.h>` and `<cuda_bf16.h>` headers, respectively.
  - `__float128`/`_Float128` type support relies on the host compiler and device compute capability, see the *Supported Floating-Point Types* table. The related functions require the `crt/device_fp128_functions.h` header.
  - They are only available in device code.
- Their behavior is affected by the `nvcc` *optimization flags*.

Intrinsic Mathematical functions (CUDA Math API):

- Support single- and double-precision types, `float` and `double` respectively.
- They are only available in device code.
- They are faster but less accurate than the respective CUDA Math API functions.
- Their behavior is not affected by the `nvcc` *floating-point optimization flags* `-prec-div=false`, `-prec-sqrt=false`, and `-fmad=true`. The only exception is `-ftz=true`, which is also included in `-use_fast_math`.

Table 45: Summary of Math Functionality Features

Functionality	Supported Types	Host	Device	Affected by Floating-Point Optimization Flags (only for `float` and `double`)
Built-in C/C++ language arithmetic operators	`float`, `double`, `__half`, `__nv_bfloat16`, `__float128/_Float128`, `cuda::std::complex`	▯	▯	▯
CUDA C++ Standard Library Mathematical functions	`float`, `double`, `__half`, `__nv_bfloat16`, `__float128`, `cuda::std::complex`	▯	▯	▯
	`__nv_fp8_e4m3`, `__nv_fp8_e5m2`, `__nv_fp8_e8m0`, `__nv_fp6_e2m3`, `__nv_fp6_e3m2`, `__nv_fp4_e2m1` *****			
CUDA C Standard Library Mathematical functions	`float`, `double`	▯	▯	▯
	`__nv_bfloat16`, `__half` with limited support and similar names	On a per-function basis		
	`__float128/_Float128` with limited support and similar names	▯	▯	
Non-standard CUDA Mathematical functions	`float`, `double`	On a per-function basis		▯
	`__nv_bfloat16`, `__half`, `__float128/_Float128` with limited support	▯	▯	
Intrinsic functions	`float`, `double`	▯	▯	Only with `-ftz=true`, also included in `-use_fast_math`

***** The *CUDA C++ Standard Library functions* support queries for small floating-point types, such as numeric_limits<T>, fpclassify(), isfinite(), isnormal(), isinf(), and isnan().

The following sections provide accuracy information for some of these functions, when applicable. It uses ULP for quantification. For more information on the definition of the Unit in the Last Place (ULP), please see Jean-Michel Muller's paper On the definition of ulp(x).

5.5.6. Built-In Arithmetic Operators

The built-in C/C++ language operators, such as `x + y`, `x - y`, `x * y`, `x / y`, `x++`, `x--`, and reciprocal `1 / x`, for single-, double-, and quad-precision types comply with the IEEE-754 standard. They guarantee a maximum ULP error of zero using a *round-to-nearest-even* rounding mode. They are available in both host and device code.

The `nvcc` compilation flag `-fmad=true`, also included in `--use_fast_math`, enables contraction of floating-point multiplies and adds/subtracts into floating-point multiply-add operations and has the following effect on the maximum ULP error for the single-precision type `float`:

- `x * y + z` □ __fmaf_rn(x, y, z): 0 ULP

The nvcc compilation flag `-prec-div=false`, also included in `--use_fast_math`, has the following effect on the maximum ULP error for the division operator `/` for the single-precision type `float`:

- `x / y` □ __fdividef(x, y): 2 ULP
- `1 / x`: 1 ULP

5.5.7. CUDA C++ Mathematical Standard Library Functions

CUDA provides comprehensive support for C++ Standard Library mathematical functions through the `cuda::std::` namespace. The functionalities are part of the `<cuda/std/cmath>` header. They are available on both host and device code.

The following sections specify the mapping with the CUDA Math APIs and the error bounds of each function when executed on the device.

- The maximum ULP error is defined as the absolute value of the difference between the ULPs returned by the function and a correctly rounded result of the corresponding precision obtained according to the *round-to-nearest ties-to-even* rounding mode.
- The error bounds are derived from extensive, though not exhaustive, testing. Therefore, they are not guaranteed.

5.5.7.1 Basic Operations

CUDA Math API for basic operations are available in both host and device code, except for `__float128`.

All the following functions have a maximum ULP error of zero.

Table 46: C++ Mathematical Standard Library Functions C Math API Mapping **Basic Operations**

`cuda::std` Function	Meaning	`__nv_bfloat`	`__half`	`float`	`double`	`__float128`
fabs(x)	$\lvert x \rvert$	__habs(x)	__habs(x)	fabsf(x)	fabs(x)	__nv_fp128_fabs(x)
fmod(x, y)	Remainder of $\frac{x}{y}$, computed as $x - \text{trunc}\left(\frac{x}{y}\right) \cdot y$	N/A	N/A	fmodf(x, y)	fmod(x, y)	__nv_fp128_fmod(x, y)
remainder(x, y)	Remainder of $\frac{x}{y}$, computed as $x - \text{rint}\left(\frac{x}{y}\right) \cdot y$	N/A	N/A	remainderf(x, y)	remainder(x, y)	__nv_fp128_remainder(x, y)
remquo(x, y, iptr)	Remainder and quotient of $\frac{x}{y}$	N/A	N/A	remquof(y, iptr)	remquo(y, iptr)	N/A
fma(x, y, z)	$x \cdot y + z$	__hfma(x, y, z), device-only	__hfma(x, y, z), device-only	fmaf(x, y, z)	fma(x, y, z)	__nv_fp128_fma(x, y, z)
fmax(x, y)	$\max(x, y)$	__hmax(x, y)	__hmax(x, y)	fmaxf(x, y)	fmax(x, y)	__nv_fp128_fmax(x, y)
fmin(x, y)	$\min(x, y)$	__hmin(x, y)	__hmin(x, y)	fminf(x, y)	fmin(x, y)	__nv_fp128_fmin(x, y)
fdim(x, y)	$\max(x - y, 0)$	N/A	N/A	fdimf(x, y)	fdim(x, y)	__nv_fp128_fdim(x, y)
nan(str)	NaN value from string representation	N/A	N/A	nanf(str)	nan(str)	N/A

* Mathematical functions marked with "N/A" are not natively available for CUDA-extended floating-point types, such as __half and __nv_bfloat16. In these cases, the fu

5.5.7.2 Exponential Functions

CUDA Math API for exponential functions are available in both host and device code only for `float` and `double` types.

Table 47: C++ Mathematical Standard Library Functions C Math API Mapping and Accuracy (Maximal ULP) **Exponential Functions**

`cuda::std` Function	Meaning	`__nv_bfloat16`	`__half`	`float`	`double`	`__float128`
exp(x)	e^x	hexp(x) 0 ULP	hexp(x) 0 ULP	expf(x) 2 ULP	exp(x) 1 ULP	__nv_fp128_exp(x) 1 ULP
exp2(x)	2^x	hexp2(x) 0 ULP	hexp2(x) 0 ULP	exp2f(x) 2 ULP	exp2(x) 1 ULP	__nv_fp128_exp2(x) 1 ULP
expm1(x)	$e^x - 1$	N/A	N/A	expm1f(x) 1 ULP	expm1(x) 1 ULP	__nv_fp128_expm1(x) 1 ULP
log(x)	$\ln(x)$	hlog(x) 0 ULP	hlog(x) 0 ULP	logf(x) 1 ULP	log(x) 1 ULP	__nv_fp128_log(x) 1 ULP
log10(x)	$\log_{10}(x)$	hlog10(x) 0 ULP	hlog10(x) 0 ULP	log10f(x) 2 ULP	log10(x) 1 ULP	__nv_fp128_log10(x) 1 ULP
log2(x)	$\log_2(x)$	hlog2(x) 0 ULP	hlog2(x) 0 ULP	log2f(x) 1 ULP	log2(x) 1 ULP	__nv_fp128_log2(x) 1 ULP
log1p(x)	$\ln(1 + x)$	N/A	N/A	log1pf(x) 1 ULP	log1p(x) 1 ULP	__nv_fp128_log1p(x) 1 ULP

* Mathematical functions marked with "N/A" are not natively available for CUDA-extended floating-point types, such as __half and __nv_bfloat16. In these cases, the fu

5.5.7.3 Power Functions

CUDA Math API for power functions are available in both host and device code only for `float` and `double` types.

Table 48: C++ Mathematical Standard Library Functions C Math API Mapping and Accuracy (Maximal ULP) **Power Functions**

`cuda::std` Function	Meaning	`__nv_bfloat16`	`__half`	`float`	`double`	`__float128`
pow(x, y)	x^y	N/A	N/A	powf(x, y) 4 ULP	pow(x, y) 2 ULP	__nv_fp128_pow(x, y) 1 ULP
sqrt(x)	$\sqrt{x}$	hsqrt(x) 0 ULP	hsqrt(x) 0 ULP	sqrtf(x) ▯ 0 ULP ▯ 1 ULP with `--use_fast_math`	sqrt(x) 0 ULP	__nv_fp128_sqrt(x) 0 ULP
cbrt(x)	$\sqrt[3]{x}$	N/A	N/A	cbrtf(x) 1 ULP	cbrt(x) 1 ULP	N/A
hypot(x, y)	$\sqrt{x^2 +$	N/A	N/A	hypotf(x, y) 3 ULP	hypot(x, y) 2 ULP	__nv_fp128_hypot(x, y) 1 ULP

* Mathematical functions marked with "N/A" are not natively available for CUDA-extended floating-point types, such as __half and __nv_bfloat16. In these cases, the fu

5.5.7.4 Trigonometric Functions

CUDA Math API for trigonometric functions are available in both host and device code only for `float` and `double` types.

Table 49: C++ Mathematical Standard Library Functions C Math API Mapping and Accuracy (Maximal ULP) **Trigonometric Functions**

`cuda::std` Function	Meaning	`__nv_bfloa`	`__half`	`float`	`double`	`__float128`
sin(x)	sin(x)	hsin(x) 0 ULP	hsin(x) 0 ULP	sinf(x) 2 ULP	sin(x) 2 ULP	__nv_fp128_sin(x) 1 ULP
cos(x)	cos(x)	hcos(x) 0 ULP	hcos(x) 0 ULP	cosf(x) 2 ULP	cos(x) 2 ULP	__nv_fp128_cos(x) 1 ULP
tan(x)	tan(x)	N/A	N/A	tanf(x) 4 ULP	tan(x) 2 ULP	__nv_fp128_tan(x) 1 ULP
asin(x)	sin$^{-1}(x)$	N/A	N/A	asinf(x) 2 ULP	asin(x) 2 ULP	__nv_fp128_asin(x) 1 ULP
acos(x)	cos$^{-1}(x)$	N/A	N/A	acosf(x) 2 ULP	acos(x) 2 ULP	__nv_fp128_acos(x) 1 ULP
atan(x)	tan$^{-1}(x)$	N/A	N/A	atanf(x) 2 ULP	atan(x) 2 ULP	__nv_fp128_atan(x) 1 ULP
atan2(y, x)	tan$^{-1}\left(\frac{y}{x}\right)$	N/A	N/A	atan2f(y, x) 3 ULP	atan2(y, x) 2 ULP	N/A

* Mathematical functions marked with "N/A" are not natively available for CUDA-extended floating-point types, such as __half and __nv_bfloat16. In these cases, the fu

5.5.7.5 Hyperbolic Functions

CUDA Math API for hyperbolic functions are available in both host and device code only for `float` and `double` types.

Table 50: C++ Mathematical Standard Library Functions C Math API Mapping and Accuracy (Maximum ULP) **Hyperbolic Functions**

cuda::std Function	Meaning	__nv_bfloa	__half	float	double	__float128
sinh(x)	sinh(x)	N/A	N/A	sinhf(x) 3 ULP	sinh(x) 2 ULP	__nv_fp128_sinh(x) 1 ULP
cosh(x)	cosh(x)	N/A	N/A	coshf(x) 2 ULP	cosh(x) 1 ULP	__nv_fp128_cosh(x) 1 ULP
tanh(x)	tanh(x)	htanh(x) 0 ULP	htanh(x) 0 ULP	tanhf(x) 2 ULP	tanh(x) 1 ULP	__nv_fp128_tanh(x) 1 ULP
asinh(x)	sinh$^{-1}(x$	N/A	N/A	asinhf(x) 3 ULP	asinh(x) 3 ULP	__nv_fp128_asinh(x) 1 ULP
acosh(x)	cosh$^{-1}($	N/A	N/A	acoshf(x) 4 ULP	acosh(x) 3 ULP	__nv_fp128_acosh(x) 1 ULP
atanh(x)	tanh$^{-1}($	N/A	N/A	atanhf(x) 3 ULP	atanh(x) 2 ULP	__nv_fp128_atanh(x) 1 ULP

* Mathematical functions marked with "N/A" are not natively available for CUDA-extended floating-point types, such as __half and __nv_bfloat16. In these cases, the fu

5.5.7.6 Error and Gamma Functions

CUDA Math API for error and gamma functions are available in both host and device code for `float` and `double` types.

Error and Gamma functions are not natively available for CUDA-extended floating-point types, such as `__half` and `__nv_bfloat16`. In these cases, the functions are emulated by converting to a `float` type and then converting the result back.

Table 51: C++ Mathematical Standard Library Functions C Math API Mapping and Accuracy (Maximum ULP) **Error and Gamma Functions**

cuda::std Function	Meaning	float	double
erf(x)	$\frac{2}{\sqrt{\pi}}\int_0^x e^-$	erff(x) 2 ULP	erf(x) 2 ULP
erfc(x)	$1 -$ erf(x)	erfcf 4 ULP	erfc 5 ULP
tgamma(x)	$\Gamma(x)$	tgammaf(x) 5 ULP	tgamma(x) 10 ULP
lgamma(x)	ln $\lvert\Gamma(x)\rvert$	lgammaf(x) □ 6 ULP for $x \notin [-10.001, -2.264]$ □ larger otherwise	lgamma(x) 4 ULP for $x \notin [-23.0001, -2.2637]$ □ larger otherwise

5.5.7.7 Nearest Integer Floating-Point Operations

CUDA Math API for nearest integer floating-point operations are available in both host and device code only for `float` and `double` types.

All the following functions have a maximum ULP error of zero.

Table 52: C++ Mathematical Standard Library Functions C Math API Mapping **Nearest Integer Floating-Point Operations**

`cuda::std` Function	**Meaning**	**`__nv_bf`**	**`__ha`**	**`float`**	**`double`**	**`__float128`**
ceil(x)	$\lceil x \rceil$	hceil(x)	hceil(	ceilf(x)	ceil(x)	__nv_fp128_ceil(x)
floor(x)	$\lfloor x \rfloor$	hfloor(x)	hfloor	floorf(x)	floor(x)	__nv_fp128_floor(x)
trunc(x)	Truncate to integer	htrunc(x)	htrun	truncf(	trunc(x	__nv_fp128_trunc(x)
round(x)	Round to nearest integer, ties away from zero	N/A	N/A	roundf(	round(	__nv_fp128_round(x)
near-byint(x)	Round to integer, ties to even	N/A	N/A	near-by-intf(x)	near-byint(x)	N/A
rint(x)	Round to integer, ties to even	hrint(x)	hrint(	rintf(x)	rint(x)	__nv_fp128_rint(x)
lrint(x)	Round to nearest integer, ties to even (returns `long int`)	N/A	N/A	lrintf(x)	lrint(x)	N/A
llrint(x)	Round to nearest integer, ties to even (returns `long long int`)	N/A	N/A	ll-rintf(x)	ll-rint(x)	N/A
lround(x)	Round to nearest integer, ties away from zero (returns `long int`)	N/A	N/A	lroundf(	lround(	N/A
llround(x)	Round to nearest integer, ties away from zero (returns `long long int`)	N/A	N/A	ll-roundf(	ll-round(	N/A

* Mathematical functions marked with “N/A” are not natively available for CUDA-extended floating-point types, such as __half and __nv_bfloat16. In these cases, the f

Performance Considerations

The recommended way to round a single- or double-precision floating-point operand to an integer is to use the functions `rintf()` and `rint()`, not `roundf()` and `round()`. This is because `roundf()` and `round()` map to multiple instructions in device code, whereas `rintf()` and `rint()` map to a single instruction. `truncf()`, `trunc()`, `ceilf()`, `ceil()`, `floorf()`, and `floor()` each map to a single instruction as well.

5.5.7.8 Floating-Point Manipulation Functions

CUDA Math API for floating-point manipulation functions are available in both host and device code, except for `__float128`.

Floating-point manipulation functions are not natively available for CUDA-extended floating-point types, such as `__half` and `__nv_bfloat16`. In these cases, the functions are emulated by converting to a `float` type and then converting the result back.

All the following functions have a maximum ULP error of zero.

Table 53: C++ Mathematical Standard Library Functions C Math API Mapping **Floating-Point Manipulation Functions**

`cuda::std` **Function**	**Meaning**	`float`	`double`	`__float128`
frexp(x, exp)	Extract mantissa and exponent	frexpf(x, exp)	frexp(x, exp)	__nv_fp128_frexp(x, nptr)
ldexp(x, n)	$x \cdot 2^{\text{n}}$	ldexpf(x, n)	ldexp(x, n)	__nv_fp128_ldexp(x, n)
modf(x, iptr)	Extract integer and fractional parts	modff(x, iptr)	modf(x, iptr)	__nv_fp128_modf(x, iptr)
scalbn(x, n)	$x \cdot 2^n$	scalbnf(x, n)	scalbn(x, n)	N/A
scalbln(x, n)	$x \cdot 2^n$	scalblnf(x, n)	scalbln(x, n)	N/A
ilogb(x)	$\lfloor \log_2(\lvert x \rvert) \rfloor$	ilogbf(x)	ilogb(x)	__nv_fp128_ilogb(x)
logb(x)	$\lfloor \log_2(\lvert x \rvert) \rfloor$	logbf(x)	logb(x)	N/A
nextafter(x, y)	Next representable value toward y	nextafterf(x, y)	nextafter(x, y)	N/A
copysign(x, y)	Copy sign of y to x	copysignf(x, y)	copysign(x, y)	__nv_fp128_copysign(x, y)

5.5.7.9 Classification and Comparison

CUDA Math API for classification and comparison functions are available in both host and device code, except for `__float128`.

All the following functions have a maximum ULP error of zero.

Table 54: C++ Mathematical Standard Library Functions C Math API Mapping **Classification and Comparison Functions**

cuda::std Function	Meaning	__nv_bfloa	__half	float	double	__float128
fpclassify(x)	Classify x	N/A	N/A	N/A	N/A	N/A
isfinite(x)	Check if x is finite	N/A	N/A	isfinite(x)	isfinite(x)	N/A
isinf(x)	Check if x is infinite	__hisinf(x)	__hisinf(x)	isinf(x)	isinf(x)	N/A
isnan(x)	Check if x is NaN	__hisnan(x)	__hisnan(x)	isnan(x)	isnan(x)	__nv_fp128_isnan(x)
isnormal(x)	Check if x is normal	N/A	N/A	N/A	N/A	N/A
signbit(x)	Check if sign bit is set	N/A	N/A	signbit(x)	signbit(x)	N/A
isgreater(x, y)	Check if $x > y$	__hgt(x, y)	__hgt(x, y)	N/A	N/A	N/A
isgreaterequal(x, y)	Check if $x \geq y$	__hge(x, y)	__hge(x, y)	N/A	N/A	N/A
isless(x, y)	Check if $x < y$	__hlt(x, y)	__hlt(x, y)	N/A	N/A	N/A
islessequal(x, y)	Check if $x \leq y$	__hle(x, y)	__hle(x, y)	N/A	N/A	N/A
islessgreater(x, y)	Check if $x < y$ or $x > y$	__hge(x, y)	__hge(x, y)	N/A	N/A	N/A
isunordered(x, y)	Check if x, y, or both are NaN	N/A	N/A	N/A	N/A	__nv_fp128_isunordered(x, y)

* Mathematical functions marked with "N/A" are not natively available for CUDA-extended floating-point types, such as __half and __nv_bfloat16.

5.5.8. Non-Standard CUDA Mathematical Functions

CUDA provides mathematical functions that are not part of the C/C++ Standard Library and are instead offered as extensions. For single- and double-precision functions, host and device code availability is defined on a per-function basis.

This section specifies the error bounds of each function when executed on the device.

- The maximum ULP error is defined as the absolute value of the difference between the ULPs returned by the function and a correctly rounded result of the corresponding precision obtained according to the *round-to-nearest ties-to-even* rounding mode.
- The error bounds are derived from extensive, though not exhaustive, testing. Therefore, they are not guaranteed.

Table 55: **Non-standard CUDA Mathematical functions** `float` and `double` Mapping and Accuracy (Maximum ULP)

Meaning	`float`	`double`
$\frac{x}{y}$	fdividef(x, y), device-only 0 ULP, same as `x / y`	N/A
10^x	exp10f(x) 2 ULP	exp10(x) 1 ULP
$\sqrt{x^2+y^2+z^2}$	norm3df(x, y, z), device-only 3 ULP	norm3d(x, y, z), device-only 2 ULP
$\sqrt{x^2+y^2+z^2+t^2}$	norm4df(x, y, z, t), device-only 3 ULP	norm4d(x, y, z, t), device-only 2 ULP
$\sqrt{\sum_{i=0}^{\text{dim}-1} p_i^2}$	normf(dim, p), device-only An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off	norm(dim, p), device-only An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off
$\frac{1}{\sqrt{x}}$	rsqrtf(x) 2 ULP	rsqrt(x) 1 ULP
$\frac{1}{\sqrt[3]{x}}$	rcbrtf(x) 1 ULP	rcbrt(x) 1 ULP
$\frac{1}{\sqrt{x^2+y^2}}$	rhypotf(x, y), device-only 2 ULP	rhypot(x, y), device-only 1 ULP
$\frac{1}{\sqrt{x^2+y^2+z^2}}$	rnorm3df(x, y, z), device-only 2 ULP	rnorm3d(x, y, z), device-only 1 ULP
$\frac{1}{\sqrt{x^2+y^2+z^2+t^2}}$	rnorm4df(x, y, z, t), device-only 2 ULP	rnorm4d(x, y, z, t), device-only 1 ULP
$\frac{1}{\sqrt{\sum_{i=0}^{\text{dim}-1} p_i^2}}$	rnormf(dim, p), device-only An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off	rnorm(dim, p), device-only An error bound cannot be provided because a fast algorithm is used with accuracy loss due to round-off
cos(πx)	cospif(x) 1 ULP	cospi(x) 2 ULP
sin(πx)	sinpif(x) 1 ULP	sinpi(x) 2 ULP
sin(πx), cos(πx)	sincospif(x, sptr, cptr) 1 ULP	sincospi(x, sptr, cptr) 2 ULP
$\Phi(x)$	normcdff(x) 5 ULP	normcdf(x) 5 ULP
$\Phi^{-1}(x)$	normcdfinvf(x) 5 ULP	normcdfinv(x) 8 ULP
erfc$^{-1}(x)$	erfcinvf(x) 4 ULP	erfcinv(x) 6 ULP
e^{x^2}erfc(x)	erfcxf(x) 4 ULP	erfcx(x) 4 ULP
erf$^{-1}(x)$	erfinvf(x) 2 ULP	erfinv(x) 5 ULP
$I_0(x)$	cyl_bessel_i0f(x), device-only 6 ULP	cyl_bessel_i0(x), device-only 6 ULP
$I_1(x)$	cyl_bessel_i1f(x), device-only 6 ULP	cyl_bessel_i1(x), device-only 6 ULP
$J_0(x)$	j0f(x) 9 ULP for $\|x\| < 8$ the maximum absolute error $= 2.2 \cdot 10^{-6}$, otherwise	j0(x) 7 ULP for $\|x\| < 8$ the maximum absolute error $= 5 \cdot 10^{-12}$, otherwise
$J_1(x)$	j1f(x) 9 ULP for $\|x\| < 8$ the maximum absolute error $= 2.2 \cdot 10^{-6}$, otherwise	j1(x) 7 ULP for $\|x\| < 8$ the maximum absolute error $= 5 \cdot 10^{-12}$, otherwise
$J_n(x)$	jnf(n, x) For $n = 128$, the maximum absolute error $= 2.2 \cdot 10^{-6}$	jn(n, x) For $n = 128$, the maximum absolute error $= 5 \cdot 10^{-12}$
$Y_0(x)$	y0f(x) 9 ULP for $\|x\| < 8$ the maximum absolute error $= 2.2 \cdot 10^{-6}$, otherwise	y0(x) 7 ULP for $\|x\| < 8$ the maximum absolute error $= 5 \cdot 10^{-12}$, otherwise
$Y_1(x)$	y1f(x) 9 ULP for $\|x\| < 8$ the maximum absolute error $= 2.2 \cdot 10^{-6}$, otherwise	y1(x) 7 ULP for $\|x\| < 8$ the maximum absolute error $= 5 \cdot 10^{-12}$, otherwise

Non-standard CUDA Mathematical functions for `__half`, `__nv_bfloat16`, and `__float128/_Float128` are only available in device code.

Table 56: **Non-standard CUDA Mathematical functions** `__nv_bfloat16`, `__half`, `__float128/_Float128` Mapping and Accuracy (Maximum ULP)

Meaning	`__nv_bfloat16`	`__half`	`__float128/_Float128`
$\frac{1}{x}$	hrcp(x) 0 ULP	hrcp(x) 0 ULP	N/A
10^x	hexp10(x) 0 ULP	hexp10(x) 0 ULP	__nv_fp128_exp10(x) 0 ULP
$\frac{1}{\sqrt{x}}$	hrsqrt(x) 0 ULP	hrsqrt(x) 0 ULP	N/A
tanh(x) (approximate)	htanh_approx(x) 1 ULP	htanh_approx(x) 1 ULP	N/A

5.5.9. Intrinsic Functions

Intrinsic mathematical functions are faster and less accurate versions of their corresponding CUDA C Standard Library Mathematical functions.

- They have the same name prefixed with `__`, such as `__sinf(x)`.
- They are only available in device code.
- They are faster because they map to fewer native instructions.
- The flag `--use_fast_math` automatically translates the corresponding CUDA Math API functions into intrinsic functions. See the *–use_fast_math Effect* section for the full list of affected functions.

5.5.9.1 Basic Intrinsic Functions

A subset of mathematical intrinsic functions allow to specify the rounding mode:

- Functions suffixed with `_rn` operate using the *round to nearest even* rounding mode.
- Functions suffixed with `_rz` operate using the *round towards zero* rounding mode.
- Functions suffixed with `_ru` operate using the *round up* (toward positive infinity) rounding mode.
- Functions suffixed with `_rd` operate using the *round down* (toward negative infinity) rounding mode.

The `__fadd_[rn,rz,ru,rd]()`, `__dadd_rn()`, `__fmul_[rn,rz,ru,rd]()`, and `__dmul_rn()` functions map to addition and multiplication operations that the compiler never merges into the `FFMA` or `DFMA` instructions. In contrast, additions and multiplications generated from the `*` and `+` operators are often combined into `FFMA` or `DFMA`.

The following table lists the single-, double-, and quad-precision floating-point intrinsic functions. All of them have a maximum ULP error of 0 and are IEEE-compliant.

Table 57: Single- and Double-Precision Floating-Point Intrinsic Functions

Meaning	`float`	`double`
$x + y$	__fadd_[rn,rz,ru,rd](x, y)	__dadd_[rn,rz,ru,rd](x, y)
$x - y$	__fsub_[rn,rz,ru,rd](x, y)	__dsub_[rn,rz,ru,rd](x, y)
$x \cdot y$	__fmul_[rn,rz,ru,rd](x, y)	__dmul_[rn,rz,ru,rd](x, y)
$x \cdot y + z$	__fmaf_[rn,rz,ru,rd](x, y, z)	__fma_[rn,rz,ru,rd](x, y, z)
$\frac{x}{y}$	__fdiv_[rn,rz,ru,rd](x, y)	__ddiv_[rn,rz,ru,rd](x, y)
$\frac{1}{x}$	__frcp_[rn,rz,ru,rd](x)	__drcp_[rn,rz,ru,rd](x)
$\sqrt{x}$	__fsqrt_[rn,rz,ru,rd](x)	__dsqrt_[rn,rz,ru,rd](x)

5.5.9.2 Single-Precision-Only Intrinsic Functions

The following table lists the single-precision floating-point intrinsic functions with their maximum ULP error.

- The maximum ULP error is stated as the absolute value of the difference in ULPs between the result returned by the CUDA library function and a correctly rounded single-precision result obtained according to the *round-to-nearest ties-to-even* rounding mode.
- The error bounds are derived from extensive, though not exhaustive, testing. Therefore, they are not guaranteed.

Table 58: **Single-Precision Only Floating-Point Intrinsic Functions** Mapping and Accuracy (Maximum ULP)

Function	Meaning	Maximum ULP Error
__fdividef(x, y)	$\frac{x}{y}$	2 for $\|y\| \in [2^{-126}, 2^{126}]$
__expf(x)	e^x	$2 + \lfloor \|1.173 \cdot x\| \rfloor$
__exp10f(x)	10^x	$2 + \lfloor \|2.97 \cdot x\| \rfloor$
__powf(x, y)	x^y	Derived from `exp2f(y * __log2f(x))`
__logf(x)	$\ln(x)$	□ $2^{-21.41}$ abs error for $x \in [0.5, 2]$ □ 3 ULP, otherwise
__log2f(x)	$\log_2(x)$	□ 2^{-22} abs error for $x \in [0.5, 2]$ □ 2 ULP, otherwise
__log10f(x)	$\log_{10}(x)$	□ 2^{-24} abs error for $x \in [0.5, 2]$ □ 3 ULP, otherwise
__sinf(x)	$\sin(x)$	□ $2^{-21.41}$ abs error for $x \in [-\pi, \pi]$ □ larger otherwise
__cosf(x)	$\cos(x)$	□ $2^{-21.41}$ abs error for $x \in [-\pi, \pi]$ □ larger otherwise
__sincosf(x, sptr, cptr)	$\sin(x)$, cos	Component-wise, the same as `__sinf(x)` and `__cosf(x)`
__tanf(x)	$\tan(x)$	Derived from `__sinf(x) * (1 / __cosf(x))`
__tanhf(x)	$\tanh(x)$	□ Max relative error: 2^{-11} □ Subnormal results are not flushed to zero even under `-ftz=true` compiler flag.

5.5.9.3 `--use_fast_math` Effect

The nvcc compiler flag `--use_fast_math` translates a subset of CUDA Math API functions called in device code into their intrinsic counterpart. Note that the *CUDA C++ Standard Library functions* are also affected by this flag. See the *Intrinsic Functions* section for more details on the implications of using intrinsic functions instead of CUDA Math API functions.

> A more robust approach is to selectively replace mathematical function calls with intrinsic versions only where the performance gains justify it and where the changed properties, such as reduced accuracy and different special-case handling, are acceptable.

Table 59: Functions Directly Affected by `--use_fast_math`

Device Function	Intrinsic Function
sinf(x)	__sinf(x)
cosf(x)	__cosf(x)
tanf(x)	__tanf(x)
sincosf(x, sptr, cptr)	__sincosf(x, sptr, cptr)
logf(x)	__logf(x)
log2f(x)	__log2f(x)
log10f(x)	__log10f(x)
expf(x)	__expf(x)
exp10f(x)	__exp10f(x)
powf(x,y)	__powf(x,y)
tanhf(x)	__tanhf(x)

5.5.10. References

1. IEEE 754-2019 Standard for Floating-Point Arithmetic.
2. Jean-Michel Muller. On the definition of ulp(x). INRIA/LIP research report, 2005.
3. Nathan Whitehead, Alex Fit-Florea. Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs. Nvidia Report, 2011.
4. David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys, March 1991.
5. David Monniaux. The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems, May 2008.
6. Peter Dinda, Conor Hetland. Do Developers Understand IEEE Floating Point?. IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2018.

5.6. Device-Callable APIs and Intrinsics

This chapter contains reference material and API documentation for APIs and intrinsics which can be called from CUDA kernels and device code.

5.6.1. Memory Barrier Primitives Interface

The primitives API is a C-like interface to `cuda::barrier` functionality. These primitives are available by including the `<cuda_awbarrier_primitives.h>` header.

5.6.1.1 Data Types

```
typedef /* implementation defined */ __mbarrier_t;
typedef /* implementation defined */ __mbarrier_token_t;
```

5.6.1.2 Memory Barrier Primitives API

```
uint32_t __mbarrier_maximum_count();
void __mbarrier_init(__mbarrier_t* bar, uint32_t expected_count);
```

- `bar` must be a pointer to `__shared__` memory.
- `expected_count <= __mbarrier_maximum_count()`
- Initialize `*bar` expected arrival count for the current and next phase to `expected_count`.

```
void __mbarrier_inval(__mbarrier_t* bar);
```

- `bar` must be a pointer to the barrier object residing in shared memory.
- Invalidation of `*bar` is required before the corresponding shared memory can be repurposed.

```
__mbarrier_token_t __mbarrier_arrive(__mbarrier_t* bar);
```

- Initialization of `*bar` must happen before this call.
- Pending count must not be zero.
- Atomically decrement the pending count for the current phase of the barrier.
- Return an arrival token associated with the barrier state immediately prior to the decrement.

```
__mbarrier_token_t __mbarrier_arrive_and_drop(__mbarrier_t* bar);
```

- Initialization of `*bar` must happen before this call.
- Pending count must not be zero.
- Atomically decrement the pending count for the current phase and expected count for the next phase of the barrier.
- Return an arrival token associated with the barrier state immediately prior to the decrement.

```
bool __mbarrier_test_wait(__mbarrier_t* bar, __mbarrier_token_t token);
```

- `token` must be associated with the immediately preceding phase or current phase of `*bar`.

- Returns `true` if `token` is associated with the immediately preceding phase of `*bar`, otherwise returns `false`.

```
bool __mbarrier_test_wait_parity(__mbarrier_t* bar, bool phase_parity);
```

- `phase_parity` must indicate the parity of either the current phase or the immediately preceding phase of `*bar`. A value of `true` corresponds to odd-numbered phases and a value of `false` corresponds to even-numbered phases.
- Returns `true` if `phase_parity` indicates the integer parity of the immediately preceding phase of `*bar`, otherwise returns `false`.

```
bool __mbarrier_try_wait(__mbarrier_t* bar, __mbarrier_token_t token, uint32_
→t max_sleep_nanosec);
```

- `token` must be associated with the immediately preceding phase or current phase of `*bar`.
- Returns `true` if `token` is associated with the immediately preceding phase of `*bar`. Otherwise, the executing thread may be suspended. Suspended thread resumes execution when the specified phase completes (returns `true`) OR before the phase completes following a system-dependent time limit (returns `false`).
- `max_sleep_nanosec` specifies the time limit, in nanoseconds, that may be used for the time limit instead of the system-dependent limit.

```
bool __mbarrier_try_wait_parity(__mbarrier_t* bar, bool phase_parity, uint32_
→t max_sleep_nanosec);
```

- `phase_parity` must indicate the parity of either the current phase or the immediately preceding phase of `*bar`. A value of `true` corresponds to odd-numbered phases and a value of `false` corresponds to even-numbered phases.
- Returns `true` if `phase_parity` indicates the integer parity of the immediately preceding phase of `*bar`. Otherwise, the executing thread may be suspended. Suspended thread resumes execution when the specified phase completes (returns `true`) OR before the phase completes following a system-dependent time limit (returns `false`).
- `max_sleep_nanosec` specifies the time limit, in nanoseconds, that may be used for the time limit instead of the system-dependent limit.

5.6.2. Pipeline Primitives Interface

Pipeline primitives provide a C-like interface for the functionality available in `<cuda/pipeline>`. The pipeline primitives interface is available by including the `<cuda_pipeline.h>` header. When compiling without ISO C++ 2011 compatibility, include the `<cuda_pipeline_primitives.h>` header.

Note

The pipeline primitives API only supports tracking asynchronous copies from global memory to shared memory with specific size and alignment requirements. It provides equivalent functionality to a `cuda::pipeline` object with `cuda::thread_scope_thread`.

5.6.2.1 memcpy_async Primitive

```
void __pipeline_memcpy_async(void* __restrict__ dst_shared,
                             const void* __restrict__ src_global,
                             size_t size_and_align,
                             size_t zfill=0);
```

- Request that the following operation be submitted for asynchronous evaluation:

  ```
  size_t i = 0;
  for (; i < size_and_align - zfill; ++i) ((char*)dst_shared)[i] =
  ↪((char*)src_global)[i]; /* copy */
  for (; i < size_and_align; ++i) ((char*)dst_shared)[i] = 0; /* zero-fill
  ↪*/
  ```

- Requirements:
  - `dst_shared` must be a pointer to the shared memory destination for the `memcpy_async`.
  - `src_global` must be a pointer to the global memory source for the `memcpy_async`.
  - `size_and_align` must be 4, 8, or 16.
  - `zfill <= size_and_align`.
  - `size_and_align` must be the alignment of `dst_shared` and `src_global`.
- It is a race condition for any thread to modify the source memory or observe the destination memory prior to waiting for the `memcpy_async` operation to complete. Between submitting a `memcpy_async` operation and waiting for its completion, any of the following actions introduces a race condition:
  - Loading from `dst_shared`.
  - Storing to `dst_shared` or `src_global`.
  - Applying an atomic update to `dst_shared` or `src_global`.

5.6.2.2 Commit Primitive

```
void __pipeline_commit();
```

- Commit submitted `memcpy_async` to the pipeline as the current batch.

5.6.2.3 Wait Primitive

```
void __pipeline_wait_prior(size_t N);
```

- Let `{0, 1, 2, ..., L}` be the sequence of indices associated with invocations of `__pipeline_commit()` by a given thread.
- Wait for completion of batches *at least* up to and including `L-N`.

5.6.2.4 Arrive On Barrier Primitive

```
void __pipeline_arrive_on(__mbarrier_t* bar);
```

- `bar` points to a barrier in shared memory.

- Increments the barrier arrival count by one, when all memcpy_async operations sequenced before this call have completed, the arrival count is decremented by one and hence the net effect on the arrival count is zero. It is user's responsibility to make sure that the increment on the arrival count does not exceed `__mbarrier_maximum_count()`.

5.6.3. Cooperative Groups API

5.6.3.1 cooperative_groups.h

5.6.3.1.1 class thread_block

Any CUDA programmer is already familiar with a certain group of threads: the thread block. The Cooperative Groups extension introduces a new datatype, `thread_block`, to explicitly represent this concept within the kernel.

`class thread_block;`

Constructed via:

```
thread_block g = this_thread_block();
```

Public Member Functions:

`static void sync()`: Synchronize the threads named in the group, equivalent to `g.barrier_wait(g.barrier_arrive())`

`thread_block::arrival_token barrier_arrive()`: Arrive on the thread_block barrier, returns a token that needs to be passed into `barrier_wait()`.

`void barrier_wait(thread_block::arrival_token&& t)`: Wait on the `thread_block` barrier, takes arrival token returned from `barrier_arrive()` as an rvalue reference.

`static unsigned int thread_rank()`: Rank of the calling thread within [0, num_threads)

`static dim3 group_index()`: 3-Dimensional index of the block within the launched grid

`static dim3 thread_index()`: 3-Dimensional index of the thread within the launched block

`static dim3 dim_threads()`: Dimensions of the launched block in units of threads

`static unsigned int num_threads()`: Total number of threads in the group

Legacy member functions (aliases):

`static unsigned int size()`: Total number of threads in the group (alias of `num_threads()`)

`static dim3 group_dim()`: Dimensions of the launched block (alias of `dim_threads()`)

Example:

```
/// Loading an integer from global into shared memory
__global__ void kernel(int *globalInput) {
    __shared__ int x;
    thread_block g = this_thread_block();
    // Choose a leader in the thread block
    if (g.thread_rank() == 0) {
        // load from global into shared for all threads to work with
        x = (*globalInput);
    }
    // After loading data into shared memory, you want to synchronize
```

(continues on next page)

(continued from previous page)

```
    // if all threads in your thread block need to see it
    g.sync(); // equivalent to __syncthreads();
}
```

Note: that all threads in the group must participate in collective operations, or the behavior is undefined.

Related: The `thread_block` datatype is derived from the more generic `thread_group` datatype, which can be used to represent a wider class of groups.

5.6.3.1.2 class cluster_group

This group object represents all the threads launched in a single cluster. The APIs are available on all hardware with Compute Capability 9.0+. In such cases, when a non-cluster grid is launched, the APIs assume a 1x1x1 cluster.

`class cluster_group;`

Constructed via:

```
cluster_group g = this_cluster();
```

Public Member Functions:

`static void sync()`: Synchronize the threads named in the group, equivalent to `g.barrier_wait(g.barrier_arrive())`

`static cluster_group::arrival_token barrier_arrive()`: Arrive on the cluster barrier, returns a token that needs to be passed into `barrier_wait()`.

`static void barrier_wait(cluster_group::arrival_token&& t)`: Wait on the cluster barrier, takes arrival token returned from `barrier_arrive()` as a rvalue reference.

`static unsigned int thread_rank()`: Rank of the calling thread within [0, num_threads)

`static unsigned int block_rank()`: Rank of the calling block within [0, num_blocks)

`static unsigned int num_threads()`: Total number of threads in the group

`static unsigned int num_blocks()`: Total number of blocks in the group

`static dim3 dim_threads()`: Dimensions of the launched cluster in units of threads

`static dim3 dim_blocks()`: Dimensions of the launched cluster in units of blocks

`static dim3 block_index()`: 3-Dimensional index of the calling block within the launched cluster

`static unsigned int query_shared_rank(const void *addr)`: Obtain the block rank to which a shared memory address belongs

`static T* map_shared_rank(T *addr, int rank)`: Obtain the address of a shared memory variable of another block in the cluster

Legacy member functions (aliases):

`static unsigned int size()`: Total number of threads in the group (alias of `num_threads()`)

5.6.3.1.3 class grid_group

This group object represents all the threads launched in a single grid. APIs other than `sync()` are available at all times, but to be able to synchronize across the grid, you need to use the cooperative launch API.

`class grid_group;`

Constructed via:

```
grid_group g = this_grid();
```

Public Member Functions:

`bool is_valid() const`: Returns whether the grid_group can synchronize

`void sync() const`: Synchronize the threads named in the group, equivalent to `g.barrier_wait(g.barrier_arrive())`

`grid_group::arrival_token barrier_arrive()`: Arrive on the grid barrier, returns a token that needs to be passed into `barrier_wait()`.

`void barrier_wait(grid_group::arrival_token&& t)`: Wait on the grid barrier, takes arrival token returned from `barrier_arrive()` as a rvalue reference.

`static unsigned long long thread_rank()`: Rank of the calling thread within [0, num_threads)

`static unsigned long long block_rank()`: Rank of the calling block within [0, num_blocks)

`static unsigned long long cluster_rank()`: Rank of the calling cluster within [0, num_clusters)

`static unsigned long long num_threads()`: Total number of threads in the group

`static unsigned long long num_blocks()`: Total number of blocks in the group

`static unsigned long long num_clusters()`: Total number of clusters in the group

`static dim3 dim_blocks()`: Dimensions of the launched grid in units of blocks

`static dim3 dim_clusters()`: Dimensions of the launched grid in units of clusters

`static dim3 block_index()`: 3-Dimensional index of the block within the launched grid

`static dim3 cluster_index()`: 3-Dimensional index of the cluster within the launched grid

Legacy member functions (aliases):

`static unsigned long long size()`: Total number of threads in the group (alias of `num_threads()`)

`static dim3 group_dim()`: Dimensions of the launched grid (alias of `dim_blocks()`)

5.6.3.1.4 class thread_block_tile

A templated version of a tiled group, where a template parameter is used to specify the size of the tile - with this known at compile time there is the potential for more optimal execution.

```
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)
```

`Size` must be a power of 2 and less than or equal to 1024. Notes section describes extra steps needed to create tiles of size larger than 32 on hardware with Compute Capability 7.5 or lower.

`ParentT` is the parent-type from which this group was partitioned. It is automatically inferred, but a value of void will store this information in the group handle rather than in the type.

Public Member Functions:

`void sync() const`: Synchronize the threads named in the group

`unsigned long long num_threads() const`: Total number of threads in the group

`unsigned long long thread_rank() const`: Rank of the calling thread within [0, num_threads)

`unsigned long long meta_group_size() const`: Returns the number of groups created when the parent group was partitioned.

`unsigned long long meta_group_rank() const`: Linear rank of the group within the set of tiles partitioned from a parent group (bounded by meta_group_size)

`T shfl(T var, unsigned int src_rank) const`: Refer to *Warp Shuffle Functions*, **Note: For sizes larger than 32 all threads in the group have to specify the same src_rank, otherwise the behavior is undefined.**

`T shfl_up(T var, int delta) const`: Refer to *Warp Shuffle Functions*, available only for sizes lower or equal to 32.

`T shfl_down(T var, int delta) const`: Refer to *Warp Shuffle Functions*, available only for sizes lower or equal to 32.

`T shfl_xor(T var, int delta) const`: Refer to *Warp Shuffle Functions*, available only for sizes lower or equal to 32.

`int any(int predicate) const`: Refer to Warp Vote Functions

`int all(int predicate) const`: Refer to Warp Vote Functions

`unsigned int ballot(int predicate) const`: Refer to Warp Vote Functions, available only for sizes lower or equal to 32.

`unsigned int match_any(T val) const`: Refer to *Warp Match Functions*, available only for sizes lower or equal to 32.

`unsigned int match_all(T val, int &pred) const`: Refer to *Warp Match Functions*, available only for sizes lower or equal to 32.

Legacy member functions (aliases):

`unsigned long long size() const`: Total number of threads in the group (alias of `num_threads()`)

Notes:

- `thread_block_tile` templated data structure is being used here, the size of the group is passed to the `tiled_partition` call as a template parameter rather than an argument.
- `shfl`, `shfl_up`, `shfl_down`, and `shfl_xor` functions accept objects of any type when compiled with C++11 or later. This means it's possible to shuffle non-integral types as long as they satisfy the below constraints:

 - Qualifies as trivially copyable i.e., `is_trivially_copyable<T>::value == true`
 - `sizeof(T) <= 32` for tile sizes lower or equal 32, `sizeof(T) <= 8` for larger tiles
- On hardware with Compute Capability 7.5 or lower tiles of size larger than 32 need small amount of memory reserved for them. This can be done using `cooperative_groups::block_tile_memory` struct template that has to reside in either shared or global memory.

  ```
  template <unsigned int MaxBlockSize = 1024>
  struct block_tile_memory;
  ```

  `MaxBlockSize` Specifies the maximal number of threads in the current thread block. This parameter can be used to minimize the shared memory usage of `block_tile_memory` in kernels launched only with smaller thread counts.

  This `block_tile_memory` needs be then passed into `cooperative_groups::this_thread_block`, allowing the resulting `thread_block` to be partitioned into tiles of sizes larger than 32. Overload of `this_thread_block` accepting `block_tile_memory` argument is a collective operation and has to be called with all threads in the `thread_block`.

  `block_tile_memory` can be used on hardware with Compute Capability 8.0 or higher in order to be able to write one source targeting multiple different Compute Capabilities. It should consume no memory when instantiated in shared memory in cases where its not required.

Examples:

```
/// The following code will create two sets of tiled groups, of size 32 and 4
↪respectively:
/// The latter has the provenance encoded in the type, while the first stores
↪it in the handle
thread_block block = this_thread_block();
thread_block_tile<32> tile32 = tiled_partition<32>(block);
thread_block_tile<4, thread_block> tile4 = tiled_partition<4>(block);
```

```
/// The following code will create tiles of size 128 on all Compute
↪Capabilities.
/// block_tile_memory can be omitted on Compute Capability 8.0 or higher.
__global__ void kernel(...) {
    // reserve shared memory for thread_block_tile usage,
    //   specify that block size will be at most 256 threads.
    __shared__ block_tile_memory<256> shared;
    thread_block thb = this_thread_block(shared);

    // Create tiles with 128 threads.
    auto tile = tiled_partition<128>(thb);

    // ...
}
```

5.6.3.1.5 class coalesced_group

In CUDA's SIMT architecture, at the hardware level the multiprocessor executes threads in groups of 32 called warps. If there exists a data-dependent conditional branch in the application code such that threads within a warp diverge, then the warp serially executes each branch disabling threads not on that path. The threads that remain active on the path are referred to as coalesced. Cooperative Groups has functionality to discover, and create, a group containing all coalesced threads.

Constructing the group handle via `coalesced_threads()` is opportunistic. It returns the set of active threads at that point in time, and makes no guarantee about which threads are returned (as long as they are active) or that they will stay coalesced throughout execution (they will be brought back together for the execution of a collective but can diverge again afterwards).

`class coalesced_group;`

Constructed via:

```
coalesced_group active = coalesced_threads();
```

Public Member Functions:

`void sync() const`: Synchronize the threads named in the group

`unsigned long long num_threads() const`: Total number of threads in the group

`unsigned long long thread_rank() const`: Rank of the calling thread within [0, num_threads)

`unsigned long long meta_group_size() const`: Returns the number of groups created when the parent group was partitioned. If this group was created by querying the set of active threads, for example `coalesced_threads()` the value of `meta_group_size()` will be 1.

`unsigned long long meta_group_rank() const`: Linear rank of the group within the set of tiles partitioned from a parent group (bounded by meta_group_size). If this group was created by querying the set of active threads, e.g. `coalesced_threads()` the value of `meta_group_rank()` will always be 0.

`T shfl(T var, unsigned int src_rank) const`: Refer to *Warp Shuffle Functions*

`T shfl_up(T var, int delta) const`: Refer to *Warp Shuffle Functions*

`T shfl_down(T var, int delta) const`: Refer to *Warp Shuffle Functions*

`int any(int predicate) const`: Refer to Warp Vote Functions

`int all(int predicate) const`: Refer to Warp Vote Functions

`unsigned int ballot(int predicate) const`: Refer to Warp Vote Functions

`unsigned int match_any(T val) const`: Refer to *Warp Match Functions*

`unsigned int match_all(T val, int &pred) const`: Refer to *Warp Match Functions*

Legacy member functions (aliases):

`unsigned long long size() const`: Total number of threads in the group (alias of `num_threads()`)

Notes:

`shfl, shfl_up, and shfl_down` functions accept objects of any type when compiled with C++11 or later. This means it's possible to shuffle non-integral types as long as they satisfy the below constraints:

- Qualifies as trivially copyable i.e. `is_trivially_copyable<T>::value == true`
- `sizeof(T) <= 32`

Example:

```
/// Consider a situation whereby there is a branch in the
/// code in which only the 2nd, 4th and 8th threads in each warp are
/// active. The coalesced_threads() call, placed in that branch, will create
↪(for each
/// warp) a group, active, that has three threads (with
/// ranks 0-2 inclusive).
__global__ void kernel(int *globalInput) {
    // Lets say globalInput says that threads 2, 4, 8 should handle the data
    if (threadIdx.x == *globalInput) {
        coalesced_group active = coalesced_threads();
        // active contains 0-2 inclusive
        active.sync();
    }
}
```

5.6.3.2 cooperative_groups/async.h

5.6.3.2.1 memcpy_async

memcpy_async is a group-wide collective memcpy that utilizes hardware accelerated support for non-blocking memory transactions from global to shared memory. Given a set of threads named in the group, memcpy_async will move specified amount of bytes or elements of the input type through a single pipeline stage. Additionally for achieving best performance when using the memcpy_async API, an alignment of 16 bytes for both shared memory and global memory is required. It is important to note that while this is a memcpy in the general case, it is only asynchronous if the source is global memory and the destination is shared memory and both can be addressed with 16, 8, or 4 byte alignments. Asynchronously copied data should only be read following a call to wait or wait_prior which signals that the corresponding stage has completed moving data to shared memory.

Having to wait on all outstanding requests can lose some flexibility (but gain simplicity). In order to efficiently overlap data transfer and execution, its important to be able to kick off an **N+1**memcpy_async request while waiting on and operating on request **N**. To do so, use memcpy_async and wait on it using the collective stage-based wait_prior API. See *wait and wait_prior* for more details.

Usage 1

```
template <typename TyGroup, typename TyElem, typename TyShape>
void memcpy_async(
  const TyGroup &group,
  TyElem *__restrict__ _dst,
  const TyElem *__restrict__ _src,
  const TyShape &shape
);
```

Performs a copy of ``**shape`` bytes**.

Usage 2

```
template <typename TyGroup, typename TyElem, typename TyDstLayout, typename
↪TySrcLayout>
void memcpy_async(
  const TyGroup &group,
  TyElem *__restrict__ dst,
  const TyDstLayout &dstLayout,
```

(continues on next page)

(continued from previous page)

```
  const TyElem *__restrict__ src,
  const TySrcLayout &srcLayout
);
```

Performs a copy of ``**min(dstLayout, srcLayout)**`` **elements**. If layouts are of type `cuda::aligned_size_t<N>`, both must specify the same alignment.

Errata The `memcpy_async` API introduced in CUDA 11.1 with both src and dst input layouts, expects the layout to be provided in elements rather than bytes. The element type is inferred from `TyElem` and has the size `sizeof(TyElem)`. If `cuda::aligned_size_t<N>` type is used as the layout, the number of elements specified times `sizeof(TyElem)` must be a multiple of N and it is recommended to use `std::byte` or `char` as the element type.

If specified shape or layout of the copy is of type `cuda::aligned_size_t<N>`, alignment will be guaranteed to be at least `min(16, N)`. In that case both `dst` and `src` pointers need to be aligned to N bytes and the number of bytes copied needs to be a multiple of N.

Codegen Requirements: Compute Capability 5.0 minimum, Compute Capability 8.0 for asynchronicity, C++11

`cooperative_groups/memcpy_async.h` header needs to be included.

Example:

```
/// This example streams elementsPerThreadBlock worth of data from global memory
/// into a limited sized shared memory (elementsInShared) block to operate on.
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

namespace cg = cooperative_groups;

__global__ void kernel(int* global_data) {
    cg::thread_block tb = cg::this_thread_block();
    const size_t elementsPerThreadBlock = 16 * 1024;
    const size_t elementsInShared = 128;
    __shared__ int local_smem[elementsInShared];

    size_t copy_count;
    size_t index = 0;
    while (index < elementsPerThreadBlock) {
        cg::memcpy_async(tb, local_smem, elementsInShared, global_data +
↪index, elementsPerThreadBlock - index);
        copy_count = min(elementsInShared, elementsPerThreadBlock - index);
        cg::wait(tb);
        // Work with local_smem
        index += copy_count;
    }
}
```

5.6.3.2.2 wait and wait_prior

```
template <typename TyGroup>
void wait(TyGroup & group);
```

(continues on next page)

(continued from previous page)

```
template <unsigned int NumStages, typename TyGroup>
void wait_prior(TyGroup & group);
```

wait and wait_prior collectives allow to wait for memcpy_async copies to complete. wait blocks calling threads until all previous copies are done. wait_prior allows that the latest NumStages are still not done and waits for all the previous requests. So with N total copies requested, it waits until the first N-NumStages are done and the last NumStages might still be in progress. Both wait and wait_prior will synchronize the named group.

Codegen Requirements: Compute Capability 5.0 minimum, Compute Capability 8.0 for asynchronicity, C++11

cooperative_groups/memcpy_async.h header needs to be included.

Example:

```
/// This example streams elementsPerThreadBlock worth of data from global memory
/// into a limited sized shared memory (elementsInShared) block to operate on in
/// multiple (two) stages. As stage N is kicked off, we can wait on and operate
↪on stage N-1.
#include <cooperative_groups.h>
#include <cooperative_groups/memcpy_async.h>

namespace cg = cooperative_groups;

__global__ void kernel(int* global_data) {
    cg::thread_block tb = cg::this_thread_block();
    const size_t elementsPerThreadBlock = 16 * 1024 + 64;
    const size_t elementsInShared = 128;
    __align__(16) __shared__ int local_smem[2][elementsInShared];
    int stage = 0;
    // First kick off an extra request
    size_t copy_count = elementsInShared;
    size_t index = copy_count;
    cg::memcpy_async(tb, local_smem[stage], elementsInShared, global_data,
↪elementsPerThreadBlock - index);
    while (index < elementsPerThreadBlock) {
        // Now we kick off the next request...
        cg::memcpy_async(tb, local_smem[stage ^ 1], elementsInShared, global_
↪data + index, elementsPerThreadBlock - index);
        // ... but we wait on the one before it
        cg::wait_prior<1>(tb);

        // Its now available and we can work with local_smem[stage] here
        // (...)
        //

        // Calculate the amount fo data that was actually copied, for the next
↪iteration.
        copy_count = min(elementsInShared, elementsPerThreadBlock - index);
        index += copy_count;

        // A cg::sync(tb) might be needed here depending on whether
        // the work done with local_smem[stage] can release threads to race
```

(continues on next page)

(continued from previous page)

```
↪ahead or not
            // Wrap to the next stage
            stage ^= 1;
        }
        cg::wait(tb);
        // The last local_smem[stage] can be handled here
}
```

5.6.3.3 cooperative_groups/partition.h

5.6.3.3.1 tiled_partition

```
template <unsigned int Size, typename ParentT>
thread_block_tile<Size, ParentT> tiled_partition(const ParentT& g);
```

```
thread_group tiled_partition(const thread_group& parent, unsigned int tilesz);
```

The `tiled_partition` method is a collective operation that partitions the parent group into a one-dimensional, row-major, tiling of subgroups. A total of ((size(parent)/tilesz) subgroups will be created, therefore the parent group size must be evenly divisible by the `Size`. The allowed parent groups are `thread_block` or `thread_block_tile`.

The implementation may cause the calling thread to wait until all the members of the parent group have invoked the operation before resuming execution. Functionality is limited to native hardware sizes, 1/2/4/8/16/32 and the `cg::size(parent)` must be greater than the `Size` parameter. The templated version of `tiled_partition` supports 64/128/256/512 sizes as well, but some additional steps are required on Compute Capability 7.5 or lower, refer to *class thread_block_tile* for details.

Codegen Requirements: Compute Capability 5.0 minimum, C++11 for sizes larger than 32

5.6.3.3.2 labeled_partition

```
template <typename Label>
coalesced_group labeled_partition(const coalesced_group& g, Label label);
```

```
template <unsigned int Size, typename Label>
coalesced_group labeled_partition(const thread_block_tile<Size>& g, Label
↪label);
```

The `labeled_partition` method is a collective operation that partitions the parent group into one-dimensional subgroups within which the threads are coalesced. The implementation will evaluate a condition label and assign threads that have the same value for label into the same group.

`Label` can be any integral type.

The implementation may cause the calling thread to wait until all the members of the parent group have invoked the operation before resuming execution.

Note: This functionality is still being evaluated and may slightly change in the future.

Codegen Requirements: Compute Capability 7.0 minimum, C++11

5.6.3.3.3 binary_partition

```
coalesced_group binary_partition(const coalesced_group& g, bool pred);
```

```
template <unsigned int Size>
coalesced_group binary_partition(const thread_block_tile<Size>& g, bool pred);
```

The `binary_partition()` method is a collective operation that partitions the parent group into one-dimensional subgroups within which the threads are coalesced. The implementation will evaluate a predicate and assign threads that have the same value into the same group. This is a specialized form of `labeled_partition()`, where the label can only be 0 or 1.

The implementation may cause the calling thread to wait until all the members of the parent group have invoked the operation before resuming execution.

Example:

```
/// This example divides a 32-sized tile into a group with odd
/// numbers and a group with even numbers
_global__ void oddEven(int *inputArr) {
    auto block = cg::this_thread_block();
    auto tile32 = cg::tiled_partition<32>(block);

    // inputArr contains random integers
    int elem = inputArr[block.thread_rank()];
    // after this, tile32 is split into 2 groups,
    // a subtile where elem&1 is true and one where its false
    auto subtile = cg::binary_partition(tile32, (elem & 1));
}
```

5.6.3.4 cooperative_groups/reduce.h

5.6.3.4.1 Reduce Operators

Below are the prototypes of function objects for some of the basic operations that can be done with `reduce`.

```
namespace cooperative_groups {
  template <typename Ty>
  struct cg::plus;

  template <typename Ty>
  struct cg::less;

  template <typename Ty>
  struct cg::greater;

  template <typename Ty>
  struct cg::bit_and;

  template <typename Ty>
  struct cg::bit_xor;

  template <typename Ty>
```

(continues on next page)

(continued from previous page)

```
  struct cg::bit_or;
}
```

Reduce is limited to the information available to the implementation at compile time. Thus in order to make use of intrinsics introduced in CC 8.0, the `cg::` namespace exposes several functional objects that mirror the hardware. These objects appear similar to those presented in the C++ STL, with the exception of `less/greater`. The reason for any difference from the STL is that these function objects are designed to actually mirror the operation of the hardware intrinsics.

Functional description:

- `cg::plus`: Accepts two values and returns the sum of both using operator+.
- `cg::less`: Accepts two values and returns the lesser using operator<. This differs in that the **lower value is returned** rather than a Boolean.
- `cg::greater`: Accepts two values and returns the greater using operator<. This differs in that the **greater value is returned** rather than a Boolean.
- `cg::bit_and`: Accepts two values and returns the result of operator&.
- `cg::bit_xor`: Accepts two values and returns the result of operator^.
- `cg::bit_or`: Accepts two values and returns the result of operator|.

Example:

```
{
    // cg::plus<int> is specialized within cg::reduce and calls __reduce_add_
↪sync(...) on CC 8.0+
    cg::reduce(tile, (int)val, cg::plus<int>());

    // cg::plus<float> fails to match with an accelerator and instead performs
↪a standard shuffle based reduction
    cg::reduce(tile, (float)val, cg::plus<float>());

    // While individual components of a vector are supported, reduce will not
↪use hardware intrinsics for the following
    // It will also be necessary to define a corresponding operator for vector
↪and any custom types that may be used
    int4 vec = {...};
    cg::reduce(tile, vec, cg::plus<int4>())

    // Finally lambdas and other function objects cannot be inspected for
↪dispatch
    // and will instead perform shuffle based reductions using the provided
↪function object.
    cg::reduce(tile, (int)val, [](int l, int r) -> int {return l + r;});
}
```

5.6.3.4.2 reduce

```
template <typename TyGroup, typename TyArg, typename TyOp>
auto reduce(const TyGroup& group, TyArg&& val, TyOp&& op) -> decltype(op(val,
↪val));
```

`reduce` performs a reduction operation on the data provided by each thread named in the group passed in. This takes advantage of hardware acceleration (on compute 80 and higher devices) for the arithmetic add, min, or max operations and the logical AND, OR, or XOR, as well as providing a software fallback on older generation hardware. Only 4B types are accelerated by hardware.

`group`: Valid group types are `coalesced_group` and `thread_block_tile`.

`val`: Any type that satisfies the below requirements:

- Qualifies as trivially copyable i.e. `is_trivially_copyable<TyArg>::value == true`
- `sizeof(T) <= 32` for `coalesced_group` and tiles of size lower or equal 32, `sizeof(T) <= 8` for larger tiles
- Has suitable arithmetic or comparative operators for the given function object.

Note: Different threads in the group can pass different values for this argument.

`op`: Valid function objects that will provide hardware acceleration with integral types are `plus()`, `less()`, `greater()`, `bit_and()`, `bit_xor()`, `bit_or()`. These must be constructed, hence the TyVal template argument is required, i.e. `plus<int>()`. Reduce also supports lambdas and other function objects that can be invoked using `operator()`

Asynchronous reduce

```
template <typename TyGroup, typename TyArg, typename TyAtomic, typename TyOp>
void reduce_update_async(const TyGroup& group, TyAtomic& atomic, TyArg&& val,
↪TyOp&& op);

template <typename TyGroup, typename TyArg, typename TyAtomic, typename TyOp>
void reduce_store_async(const TyGroup& group, TyAtomic& atomic, TyArg&& val,
↪TyOp&& op);

template <typename TyGroup, typename TyArg, typename TyOp>
void reduce_store_async(const TyGroup& group, TyArg* ptr, TyArg&& val, TyOp&&
↪op);
```

`*_async` variants of the API are asynchronously calculating the result to either store to or update a specified destination by one of the participating threads, instead of returning it by each thread. To observe the effect of these asynchronous calls, calling group of threads or a larger group containing them need to be synchronized.

- In case of the atomic store or update variant, `atomic` argument can be either of `cuda::atomic` or `cuda::atomic_ref` available in CUDA C++ Standard Library. This variant of the API is available only on platforms and devices, where these types are supported by the CUDA C++ Standard Library. Result of the reduction is used to atomically update the atomic according to the specified op, eg. the result is atomically added to the atomic in case of `cg::plus()`. Type held by the `atomic` must match the type of `TyArg`. Scope of the atomic must include all the threads in the group and if multiple groups are using the same atomic concurrently, scope must include all threads in all groups using it. Atomic update is performed with relaxed memory ordering.
- In case of the pointer store variant, result of the reduction will be weakly stored into the `dst` pointer.

5.6.3.5 cooperative_groups/scan.h

5.6.3.5.1 inclusive_scan and exclusive_scan

```
template <typename TyGroup, typename TyVal, typename TyFn>
auto inclusive_scan(const TyGroup& group, TyVal&& val, TyFn&& op) ->
→decltype(op(val, val));

template <typename TyGroup, typename TyVal>
TyVal inclusive_scan(const TyGroup& group, TyVal&& val);

template <typename TyGroup, typename TyVal, typename TyFn>
auto exclusive_scan(const TyGroup& group, TyVal&& val, TyFn&& op) ->
→decltype(op(val, val));

template <typename TyGroup, typename TyVal>
TyVal exclusive_scan(const TyGroup& group, TyVal&& val);
```

`inclusive_scan` and `exclusive_scan` performs a scan operation on the data provided by each thread named in the group passed in. Result for each thread is a reduction of data from threads with lower `thread_rank` than that thread in case of `exclusive_scan`. `inclusive_scan` result also includes the calling thread data in the reduction.

`group`: Valid group types are `coalesced_group` and `thread_block_tile`.

`val`: Any type that satisfies the below requirements:

- Qualifies as trivially copyable i.e. `is_trivially_copyable<TyArg>::value == true`
- `sizeof(T) <= 32` for `coalesced_group` and tiles of size lower or equal 32, `sizeof(T) <= 8` for larger tiles
- Has suitable arithmetic or comparative operators for the given function object.

Note: Different threads in the group can pass different values for this argument.

`op`: Function objects defined for convenience are `plus()`, `less()`, `greater()`, `bit_and()`, `bit_xor()`, `bit_or()` described in *cooperative_groups/reduce.h*. These must be constructed, hence the TyVal template argument is required, i.e. `plus<int>()`. `inclusive_scan` and `exclusive_scan` also supports lambdas and other function objects that can be invoked using `operator()`. Overloads without this argument use `cg::plus<TyVal>()`.

Scan update

```
template <typename TyGroup, typename TyAtomic, typename TyVal, typename TyFn>
auto inclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&&
→val, TyFn&& op) -> decltype(op(val, val));

template <typename TyGroup, typename TyAtomic, typename TyVal>
TyVal inclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&&
→val);

template <typename TyGroup, typename TyAtomic, typename TyVal, typename TyFn>
auto exclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&&
→val, TyFn&& op) -> decltype(op(val, val));

template <typename TyGroup, typename TyAtomic, typename TyVal>
```

(continues on next page)

(continued from previous page)

```
TyVal exclusive_scan_update(const TyGroup& group, TyAtomic& atomic, TyVal&&
↪val);
```

*_scan_update collectives take an additional argument atomic that can be either of cuda::atomic or cuda::atomic_ref available in CUDA C++ Standard Library. These variants of the API are available only on platforms and devices, where these types are supported by the CUDA C++ Standard Library. These variants will perform an update to the atomic according to op with value of the sum of input values of all threads in the group. Previous value of the atomic will be combined with the result of scan by each thread and returned. Type held by the atomic must match the type of TyVal. Scope of the atomic must include all the threads in the group and if multiple groups are using the same atomic concurrently, scope must include all threads in all groups using it. Atomic update is performed with relaxed memory ordering.

Following pseudocode illustrates how the update variant of scan works:

```
/*
 inclusive_scan_update behaves as the following block,
 except both reduce and inclusive_scan is calculated simultaneously.
auto total = reduce(group, val, op);
TyVal old;
if (group.thread_rank() == selected_thread) {
    atomically {
        old = atomic.load();
        atomic.store(op(old, total));
    }
}
old = group.shfl(old, selected_thread);
return op(inclusive_scan(group, val, op), old);
*/
```

cooperative_groups/scan.h header needs to be included.

Example of stream compaction using exclusive_scan:

```
#include <cooperative_groups.h>
#include <cooperative_groups/scan.h>
namespace cg = cooperative_groups;

// put data from input into output only if it passes test_fn predicate
template<typename Group, typename Data, typename TyFn>
__device__ int stream_compaction(Group &g, Data *input, int count, TyFn&&
↪test_fn, Data *output) {
    int per_thread = count / g.num_threads();
    int thread_start = min(g.thread_rank() * per_thread, count);
    int my_count = min(per_thread, count - thread_start);

    // get all passing items from my part of the input
    //  into a contagious part of the array and count them.
    int i = thread_start;
    while (i < my_count + thread_start) {
        if (test_fn(input[i])) {
            i++;
        }
```

(continues on next page)

(continued from previous page)

```
        else {
            my_count--;
            input[i] = input[my_count + thread_start];
        }
    }

    // scan over counts from each thread to calculate my starting
    //  index in the output
    int my_idx = cg::exclusive_scan(g, my_count);

    for (i = 0; i < my_count; ++i) {
        output[my_idx + i] = input[thread_start + i];
    }
    // return the total number of items in the output
    return g.shfl(my_idx + my_count, g.num_threads() - 1);
}
```

Example of dynamic buffer space allocation using exclusive_scan_update:

```
#include <cooperative_groups.h>
#include <cooperative_groups/scan.h>
namespace cg = cooperative_groups;

// Buffer partitioning is static to make the example easier to follow,
// but any arbitrary dynamic allocation scheme can be implemented by replacing
↪this function.
__device__ int calculate_buffer_space_needed(cg::thread_block_tile<32>& tile)
↪{
    return tile.thread_rank() % 2 + 1;
}

__device__ int my_thread_data(int i) {
    return i;
}

__global__ void kernel() {
    __shared__ extern int buffer[];
    __shared__ cuda::atomic<int, cuda::thread_scope_block> buffer_used;

    auto block = cg::this_thread_block();
    auto tile = cg::tiled_partition<32>(block);
    buffer_used = 0;
    block.sync();

    // each thread calculates buffer size it needs
    int buf_needed = calculate_buffer_space_needed(tile);

    // scan over the needs of each thread, result for each thread is an offset
    // of that thread's part of the buffer. buffer_used is atomically updated
↪with
    // the sum of all thread's inputs, to correctly offset other tile's
↪allocations
```

(continues on next page)

(continued from previous page)

```
    int buf_offset =
        cg::exclusive_scan_update(tile, buffer_used, buf_needed);

    // each thread fills its own part of the buffer with thread specific data
    for (int i = 0 ; i < buf_needed ; ++i) {
        buffer[buf_offset + i] = my_thread_data(i);
    }

    block.sync();
    // buffer_used now holds total amount of memory allocated
    // buffer is {0, 0, 1, 0, 0, 1 ...};
}
```

5.6.3.6 cooperative_groups/sync.h

5.6.3.6.1 barrier_arrive and barrier_wait

```
T::arrival_token T::barrier_arrive();
void T::barrier_wait(T::arrival_token&&);
```

barrier_arrive and barrier_wait member functions provide a synchronization API similar to cuda::barrier *(read more)*. Cooperative Groups automatically initializes the group barrier, but arrive and wait operations have an additional restriction resulting from collective nature of those operations: All threads in the group must arrive and wait at the barrier once per phase. When barrier_arrive is called with a group, result of calling any collective operation or another barrier arrival with that group is undefined until completion of the barrier phase is observed with barrier_wait call. Threads blocked on barrier_wait might be released from the synchronization before other threads call barrier_wait, but only after all threads in the group called barrier_arrive. Group type T can be any of the *implicit groups*. This allows threads to do independent work after they arrive and before they wait for the synchronization to resolve, allowing to hide some of the synchronization latency. barrier_arrive returns an arrival_token object that must be passed into the corresponding barrier_wait. Token is consumed this way and can not be used for another barrier_wait call.

Example of barrier_arrive and barrier_wait used to synchronize initialization of shared memory across the cluster:

```
#include <cooperative_groups.h>

using namespace cooperative_groups;

void __device__ init_shared_data(const thread_block& block, int *data);
void __device__ local_processing(const thread_block& block);
void __device__ process_shared_data(const thread_block& block, int *data);

__global__ void cluster_kernel() {
    extern __shared__ int array[];
    auto cluster = this_cluster();
    auto block   = this_thread_block();

    // Use this thread block to initialize some shared state
    init_shared_data(block, &array[0]);
```

(continues on next page)

(continued from previous page)

```
    auto token = cluster.barrier_arrive(); // Let other blocks know this block
↪is running and data was initialized

    // Do some local processing to hide the synchronization latency
    local_processing(block);

    // Map data in shared memory from the next block in the cluster
    int *dsmem = cluster.map_shared_rank(&array[0], (cluster.block_rank() +
↪1) % cluster.num_blocks());

    // Make sure all other blocks in the cluster are running and initialized
↪shared data before accessing dsmem
    cluster.barrier_wait(std::move(token));

    // Consume data in distributed shared memory
    process_shared_data(block, dsmem);
    cluster.sync();
}
```

5.6.3.6.2 sync

```
static void T::sync();

template <typename T>
void sync(T& group);
```

sync synchronizes the threads named in the group. Group type T can be any of the existing group types, as all of them support synchronization. Its available as a member function in every group type or as a free function taking a group as parameter.

5.6.3.6.2.1 Grid Synchronization

Prior to the introduction of Cooperative Groups, the CUDA programming model only allowed synchronization between thread blocks at a kernel completion boundary. The kernel boundary carries with it an implicit invalidation of state, and with it, potential performance implications.

For example, in certain use cases, applications have a large number of small kernels, with each kernel representing a stage in a processing pipeline. The presence of these kernels is required by the current CUDA programming model to ensure that the thread blocks operating on one pipeline stage have produced data before the thread block operating on the next pipeline stage is ready to consume it. In such cases, the ability to provide global inter thread block synchronization would allow the application to be restructured to have persistent thread blocks, which are able to synchronize on the device when a given stage is complete.

To synchronize across the grid, from within a kernel, you would simply use the `grid.sync()` function:

```
grid_group grid = this_grid();
grid.sync();
```

And when launching the kernel it is necessary to use, instead of the `<<<...>>>` execution configuration syntax, the `cudaLaunchCooperativeKernel` CUDA runtime launch API or the CUDA `driver equivalent`.

Example:

To guarantee co-residency of the thread blocks on the GPU, the number of blocks launched needs to be carefully considered. For example, as many blocks as there are SMs can be launched as follows:

```
int dev = 0;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
// initialize, then launch
cudaLaunchCooperativeKernel((void*)my_kernel, deviceProp.multiProcessorCount,
↪numThreads, args);
```

Alternatively, you can maximize the exposed parallelism by calculating how many blocks can fit simultaneously per-SM using the occupancy calculator as follows:

```
/// This will launch a grid that can maximally fill the GPU, on the default
↪stream with kernel arguments
int numBlocksPerSm = 0;
 // Number of threads my_kernel will be launched with
int numThreads = 128;
cudaDeviceProp deviceProp;
cudaGetDeviceProperties(&deviceProp, dev);
cudaOccupancyMaxActiveBlocksPerMultiprocessor(&numBlocksPerSm, my_kernel,
↪numThreads, 0);
// launch
void *kernelArgs[] = { /* add kernel args */ };
dim3 dimBlock(numThreads, 1, 1);
dim3 dimGrid(deviceProp.multiProcessorCount*numBlocksPerSm, 1, 1);
cudaLaunchCooperativeKernel((void*)my_kernel, dimGrid, dimBlock, kernelArgs);
```

It is good practice to first ensure the device supports cooperative launches by querying the device attribute `cudaDevAttrCooperativeLaunch`:

```
int dev = 0;
int supportsCoopLaunch = 0;
cudaDeviceGetAttribute(&supportsCoopLaunch, cudaDevAttrCooperativeLaunch,
↪dev);
```

which will set `supportsCoopLaunch` to 1 if the property is supported on device 0. Only devices with compute capability of 6.0 and higher are supported. In addition, you need to be running on either of these:

- The Linux platform without MPS
- The Linux platform with MPS and on a device with compute capability 7.0 or higher
- The latest Windows platform

5.6.4. CUDA Device Runtime

The CUDA device runtime is an API available within kernel code which provides many of the same capabilities as the CUDA Runtime API on the host. These APIs are used most often in the contexts of *CUDA Dynamic Parallelism* or *Device Graph Launch*.

5.6.4.1 Including Device Runtime API in CUDA Code

Similar to the host-side runtime API, prototypes for the CUDA device runtime API are included automatically during program compilation. There is no need to include`cuda_device_runtime_api.h` explicitly.

5.6.4.2 Memory in the CUDA Device Runtime

5.6.4.2.1 Configuration Options

Resource allocation for the device runtime system software is controlled via the `cudaDeviceSetLimit()` API from the host program. Limits must be set before any kernel is launched, and may not be changed while the GPU is actively running programs.

The following named limits may be set:

Limit	Behavior
`cudaLimitDevRuntimePendingLaunchCount`	Controls the amount of memory set aside for buffering kernel launches and events which have not yet begun to execute, due either to unresolved dependencies or lack of execution resources. When the buffer is full, an attempt to allocate a launch slot during a device side kernel launch will fail and return `cudaErrorLaunchOutOfResources`, while an attempt to allocate an event slot will fail and return `cudaErrorMemoryAllocation`. The default number of launch slots is 2048. Applications may increase the number of launch and/or event slots by setting `cudaLimitDevRuntimePendingLaunchCount`. The number of event slots allocated is twice the value of that limit.
`cudaLimitStackSize`	Controls the stack size in bytes of each GPU thread. The CUDA driver automatically increases the per-thread stack size for each kernel launch as needed. This size isn't reset back to the original value after each launch. To set the per-thread stack size to a different value, `cudaDeviceSetLimit()` can be called to set this limit. The stack will be immediately resized, and if necessary, the device will block until all preceding requested tasks are complete. `cudaDeviceGetLimit()` can be called to get the current per-thread stack size.

5.6.4.2.2 Allocation and Lifetime

`cudaMalloc()` and `cudaFree()` have distinct semantics between the host and device environments. When invoked from the host, `cudaMalloc()` allocates a new region from unused device memory. When invoked from the device runtime these functions map to device-side `malloc()` and `free()`. This implies that within the device environment the total allocatable memory is limited to the device `malloc()` heap size, which may be smaller than the available unused device memory. Also, it is an error to invoke `cudaFree()` from the host program on a pointer which was allocated by `cudaMalloc()` on the device or vice-versa.

	`cudaMalloc()` on Host	`cudaMalloc()` on Device
`cudaFree()` on Host	Supported	Not Supported
`cudaFree()` on Device	Not Supported	Supported
Allocation limit	Available device memory	`cudaLimitMallocHeapSize`

5.6.4.2.2.1 Memory Declarations

5.6.4.2.2.2 Device and Constant Memory

Memory declared at file scope with `__device__` or `__constant__` memory space specifiers behaves identically when using the device runtime. All kernels may read or write device variables, whether the kernel was initially launched by the host or device runtime. Equivalently, all kernels will have the same view of `__constant__`s as declared at the module scope.

5.6.4.2.2.3 Textures and Surfaces

The device runtime does not allow creation or destruction of texture or surface objects from within device code. Texture and surface objects created from the host may be used and passed around freely on the device. Regardless of where they are created, dynamically created texture objects are always valid and may be passed to child kernels from a parent.

Note

The device runtime does not support legacy module-scope (i.e., compute capability 2.0 or Fermi-style) textures and surfaces within a kernel launched from the device. Module-scope (legacy) textures may be created from the host and used in device code as for any kernel, but may only be used by a top-level kernel (i.e., the one which is launched from the host).

5.6.4.2.2.4 Shared Memory Variable Declarations

In CUDA C++ shared memory can be declared either as a statically sized file-scope or function-scoped variable, or as an `extern` variable with the size determined at runtime by the kernel's caller via a launch configuration argument. Both types of declarations are valid under the device runtime.

```
__global__ void permute(int n, int *data) {
   extern __shared__ int smem[];
   if (n <= 1)
       return;

   smem[threadIdx.x] = data[threadIdx.x];
   __syncthreads();

   permute_data(smem, n);
   __syncthreads();

   // Write back to GMEM since we can't pass SMEM to children.
   data[threadIdx.x] = smem[threadIdx.x];
   __syncthreads();
```

(continues on next page)

(continued from previous page)

```
    if (threadIdx.x == 0) {
        permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data);
        permute<<< 1, 256, n/2*sizeof(int) >>>(n/2, data+n/2);
    }
}

void host_launch(int *data) {
    permute<<< 1, 256, 256*sizeof(int) >>>(256, data);
}
```

5.6.4.2.2.5 Constant Memory

Constants may not be modified from the device. They may only be modified from the host, but the behavior of modifying a constant from the host while there is a concurrent grid that access that constant at any point during its lifetime is undefined.

5.6.4.2.2.6 Symbol Addresses

Device-side symbols (i.e., those marked `__device__`) may be referenced from within a kernel simply via the & operator, as all global-scope device variables are in the kernel's visible address space. This also applies to `__constant__` symbols, although in this case the pointer will reference read-only data.

Since device-side symbols can be referenced directly, those CUDA runtime APIs which reference symbols (e.g., `cudaMemcpyToSymbol()` or `cudaGetSymbolAddress()`) are unnecessary and are not supported by the device runtime. This implies that constant data cannot be altered from within a running kernel, even ahead of a child kernel launch, as references to `__constant__` space are read-only.

5.6.4.3 SM Id and Warp Id

Note that in PTX `%smid` and `%warpid` are defined as volatile values. The device runtime may reschedule thread blocks onto different SMs in order to more efficiently manage resources. As such, it is unsafe to rely upon `%smid` or `%warpid` remaining unchanged across the lifetime of a thread or thread block.

5.6.4.4 Launch Setup APIs

Device-Side Kernel Launch describes the syntax for launching kernels from device code using the same triple chevron launch notation as the host CUDA Runtime API.

Kernel launch is a system-level mechanism exposed through the device runtime library. It is also available directly from PTX via `cudaGetParameterBuffer()` and `cudaLaunchDevice()` APIs. It is permitted for a CUDA application to call these APIs itself, with the same requirements as for PTX. In both cases, the user is then responsible for correctly populating all necessary data structures in the correct format according to specification. Backwards compatibility is guaranteed in these data structures.

As with host-side launch, the device-side operator <<<>>> maps to underlying kernel launch APIs. This allows users targeting PTX will to perform a launch. The NVCC compiler front-end translates <<<>>> into these calls.

Table 60: New Device-only Launch Implementation Functions

Runtime API Launch Functions	Description of Difference From Host Runtime Behavior (behavior is identical if no description)
cudaGetParameterBuffer	Generated automatically from <<<>>>. Note different API to host equivalent.
cudaLaunchDevice	Generated automatically from <<<>>>. Note different API to host equivalent.

The APIs for these launch functions are different to those of the CUDA Runtime API, and are defined as follows:

```
extern   device   cudaError_t cudaGetParameterBuffer(void **params);
extern __device__ cudaError_t cudaLaunchDevice(void *kernel,
                                        void *params, dim3 gridDim,
                                        dim3 blockDim,
                                        unsigned int sharedMemSize = 0,
                                        cudaStream_t stream = 0);
```

5.6.4.5 Device Management

There is no multi-GPU support from the device runtime; the device runtime is only capable of operating on the device upon which it is currently executing. It is permitted, however, to query properties for any CUDA capable device in the system.

5.6.4.6 API Reference

The portions of the CUDA Runtime API supported in the device runtime are detailed here. Host and device runtime APIs have identical syntax; semantics are the same except where indicated. The following table provides an overview of the API relative to the version available from the host.

Table 61: Supported API Functions

Runtime API Functions	Details
`cudaDeviceGetCacheConfig`	
`cudaDeviceGetLimit`	
`cudaGetLastError`	Last error is per-thread state, not per-block state
`cudaPeekAtLastError`	
`cudaGetErrorString`	
`cudaGetDeviceCount`	
`cudaDeviceGetAttribute`	Will return attributes for any device
`cudaGetDevice`	Always returns current device ID as would be seen from host
`cudaStreamCreateWithFlags`	Must pass `cudaStreamNonBlocking` flag
`cudaStreamDestroy`	
`cudaStreamWaitEvent`	
`cudaEventCreateWithFlags`	Must pass `cudaEventDisableTiming` flag
`cudaEventRecord`	
`cudaEventDestroy`	
`cudaFuncGetAttributes`	
`cudaMemcpyAsync`	Notes about all `memcpy/memset` functions: ▶ Only async `memcpy/set` functions are supported ▶ Only device-to-device `memcpy` is permitted ▶ May not pass in local or shared memory pointers
`cudaMemcpy2DAsync`	
`cudaMemcpy3DAsync`	
`cudaMemsetAsync`	
`cudaMemset2DAsync`	
`cudaMemset3DAsync`	
`cudaRuntimeGetVersion`	
`cudaMalloc`	May not call `cudaFree` on the device on a pointer created on the host, and vice-versa
`cudaFree`	
`cudaOccupancyMaxActiveBlocksPerMultiprocessor`	
`cudaOccupancyMaxPotentialBlockSize`	
`cudaOccupancyMaxPotentialBlockSizeVariableSMem`	

5.6.4.7 API Errors and Launch Failures

As usual for the CUDA runtime, any function may return an error code. The last error code returned is recorded and may be retrieved via the `cudaGetLastError()` call. Errors are recorded per-thread, so that each thread can identify the most recent error that it has generated. The error code is of type `cudaError_t`.

Similar to a host-side launch, device-side launches may fail for many reasons (invalid arguments, etc). The user must call `cudaGetLastError()` to determine if a launch generated an error, however lack of an error after launch does not imply the child kernel completed successfully.

For device-side exceptions, e.g., access to an invalid address, an error in a child grid will be returned to the host.

5.6.4.8 Device Runtime Streams

The CUDA device runtime exposes special named streams which provide specific behaviors for kernels and graphs launched from the device. The named streams relevant to device graph launch are documented in *Device Launch*. Two other named streams which can be used for kernels and memcpy operations in the CUDA device runtime are `cudaStreamTailLaunch` and `cudaStreamTailLaunch`. The specific behaviors of these named streams are documented in this section.

Both named and unnamed (NULL) streams are available from the device runtime. Stream handles may not be passed to parent or child grids. In other words, a stream should be treated as private to the grid in which it is created.

The host-side NULL stream's cross-stream barrier semantic is not supported on the device (see below for details). In order to retain semantic compatibility with the host runtime, all device streams must be created using the `cudaStreamCreateWithFlags()` API, passing the `cudaStreamNonBlocking` flag. The `cudaStreamCreate()` API is not available in the CUDA device runtime.

As `cudaStreamSynchronize()` and `cudaStreamQuery()` are unsupported by the device runtime. A kernel launched into the `cudaStreamTailLaunch` stream should be used instead when the application needs to know that stream-launched child kernels have completed.

5.6.4.8.1 The Implicit (NULL) Stream

Within a host program, the unnamed (NULL) stream has additional barrier synchronization semantics with other streams (see *Blocking and non-blocking streams and the default stream* for details). The device runtime offers a single implicit, unnamed stream shared between all threads in a thread block, but as all named streams must be created with the `cudaStreamNonBlocking` flag, work launched into the NULL stream will not insert an implicit dependency on pending work in any other streams (including NULL streams of other thread blocks).

5.6.4.8.2 The Fire-and-Forget Stream

The fire-and-forget named stream (`cudaStreamFireAndForget`) allows the user to launch fire-and-forget work with less boilerplate and without stream tracking overhead. It is functionally identical to, but faster than, creating a new stream per launch, and launching into that stream.

Fire-and-forget launches are immediately scheduled for launch without any dependency on the completion of previously launched grids. No other grid launches can depend on the completion of a fire-and-forget launch, except through the implicit synchronization at the end of the parent grid. So a tail launch or the next grid in parent grid's stream won't launch before a parent grid's fire-and-forget work has completed.

```
// In this example, C2's launch will not wait for C1's completion
__global__ void P( ... ) {
   C1<<< ... , cudaStreamFireAndForget >>>( ... );
   C2<<< ... , cudaStreamFireAndForget >>>( ... );
}
```

The fire-and-forget stream cannot be used to record or wait on events. Attempting to do so results in `cudaErrorInvalidValue`. The fire-and-forget stream is not supported when compiled with `CUDA_FORCE_CDP1_IF_SUPPORTED` defined. Fire-and-forget stream usage requires compilation to be in 64-bit mode.

5.6.4.8.3 The Tail Launch Stream

The tail launch named stream (`cudaStreamTailLaunch`) allows a grid to schedule a new grid for launch after its completion. It should be possible to to use a tail launch to achieve the same functionality as a `cudaDeviceSynchronize()` in most cases.

Each grid has its own tail launch stream. All non-tail launch work launched by a grid is implicitly synchronized before the tail stream is kicked off. I.e. A parent grid's tail launch does not launch until the parent grid and all work launched by the parent grid to ordinary streams or per-thread or fire-and-forget streams have completed. If two grids are launched to the same grid's tail launch stream, the later grid does not launch until the earlier grid and all its descendent work has completed.

```
// In this example, C2 will only launch after C1 completes.
__global__ void P( ... ) {
   C1<<< ... , cudaStreamTailLaunch >>>( ... );
   C2<<< ... , cudaStreamTailLaunch >>>( ... );
}
```

Grids launched into the tail launch stream will not launch until the completion of all work by the parent grid, including all other grids (and their descendants) launched by the parent in all non-tail launched streams, including work executed or launched after the tail launch.

```
// In this example, C will only launch after all X, F and P complete.
__global__ void P( ... ) {
   C<<< ... , cudaStreamTailLaunch >>>( ... );
   X<<< ... , cudaStreamPerThread >>>( ... );
   F<<< ... , cudaStreamFireAndForget >>>( ... )
}
```

The next grid in the parent grid's stream will not be launched before a parent grid's tail launch work has completed. In other words, the tail launch stream behaves as if it were inserted between its parent grid and the next grid in its parent grid's stream.

```
// In this example, P2 will only launch after C completes.
__global__ void P1( ... ) {
   C<<< ... , cudaStreamTailLaunch >>>( ... );
}

__global__ void P2( ... ) {
}

int main ( ... ) {
   ...
```

(continues on next page)

(continued from previous page)

```
    P1<<< ... >>>( ... );
    P2<<< ... >>>( ... );
    ...
}
```

Each grid only gets one tail launch stream. To tail launch concurrent grids, it can be done like the example below.

```
// In this example,  C1 and C2 will launch concurrently after P's completion
__global__ void T( ... ) {
    C1<<< ... , cudaStreamFireAndForget >>>( ... );
    C2<<< ... , cudaStreamFireAndForget >>>( ... );
}

__global__ void P( ... ) {
    ...
    T<<< ... , cudaStreamTailLaunch >>>( ... );
}
```

The tail launch stream cannot be used to record or wait on events. Attempting to do so results in `cudaErrorInvalidValue`. The tail launch stream is not supported when compiled with `CUDA_FORCE_CDP1_IF_SUPPORTED` defined. Tail launch stream usage requires compilation to be in 64-bit mode.

5.6.4.9 ECC Errors

No notification of ECC errors is available to code within a CUDA kernel. ECC errors are reported at the host side once the entire launch tree has completed. Any ECC errors which arise during execution of a nested program will either generate an exception or continue execution (depending upon error and configuration).

Chapter 6. Notices

6.1. Notice

6.2. OpenCL

6.3. Trademarks

Copyright