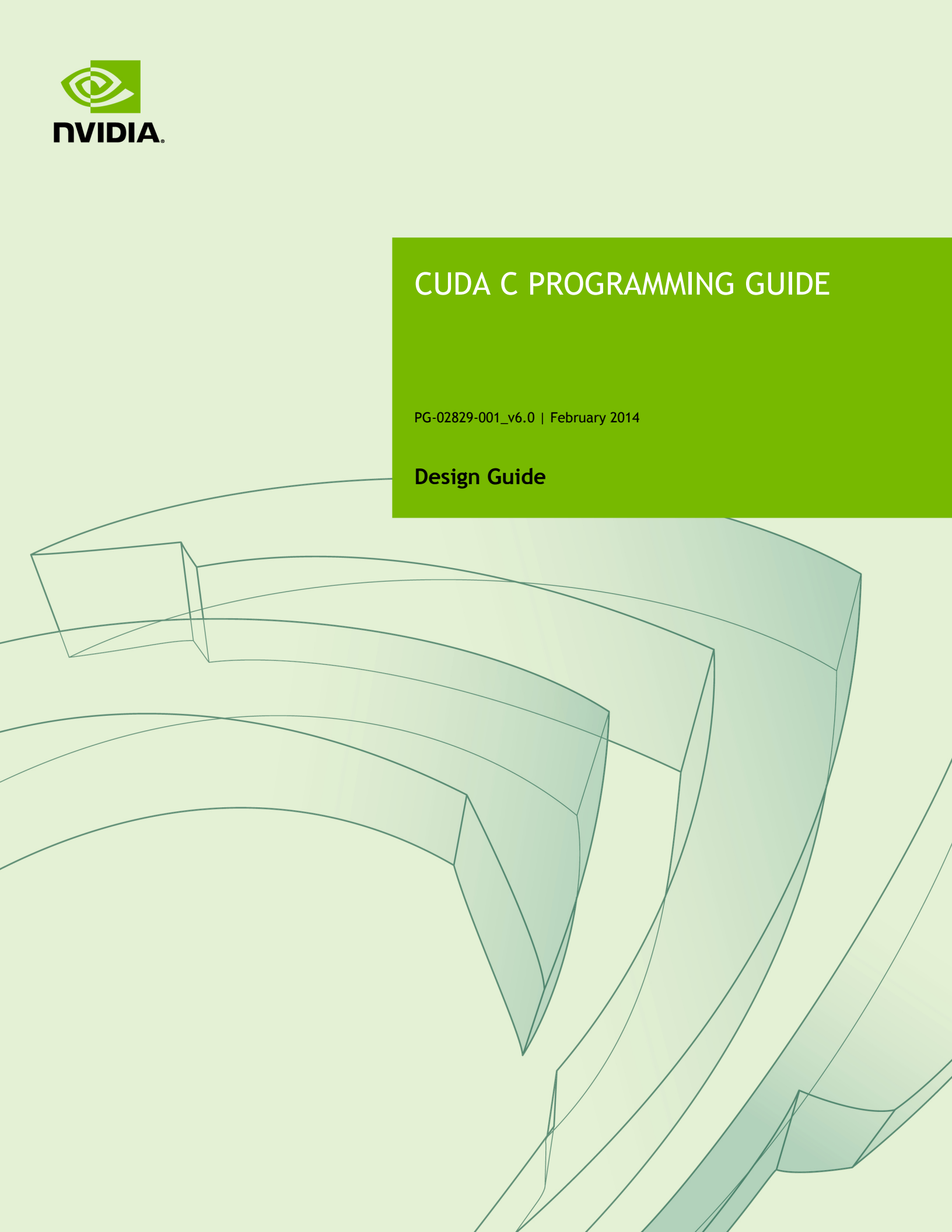


CUDA C PROGRAMMING GUIDE

PG-02829-001_v6.0 | February 2014

Design Guide

CHANGES FROM VERSION 5.5

- Added new appendix Unified Memory Programming.
- Added new appendix Compute Capability 5.0.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. INTRODUCTION

1.1. From Graphics Processing to General Purpose Parallel Computing

Driven by the insatiable market demand for realtime, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth, as illustrated by Figure 1 and Figure 2.

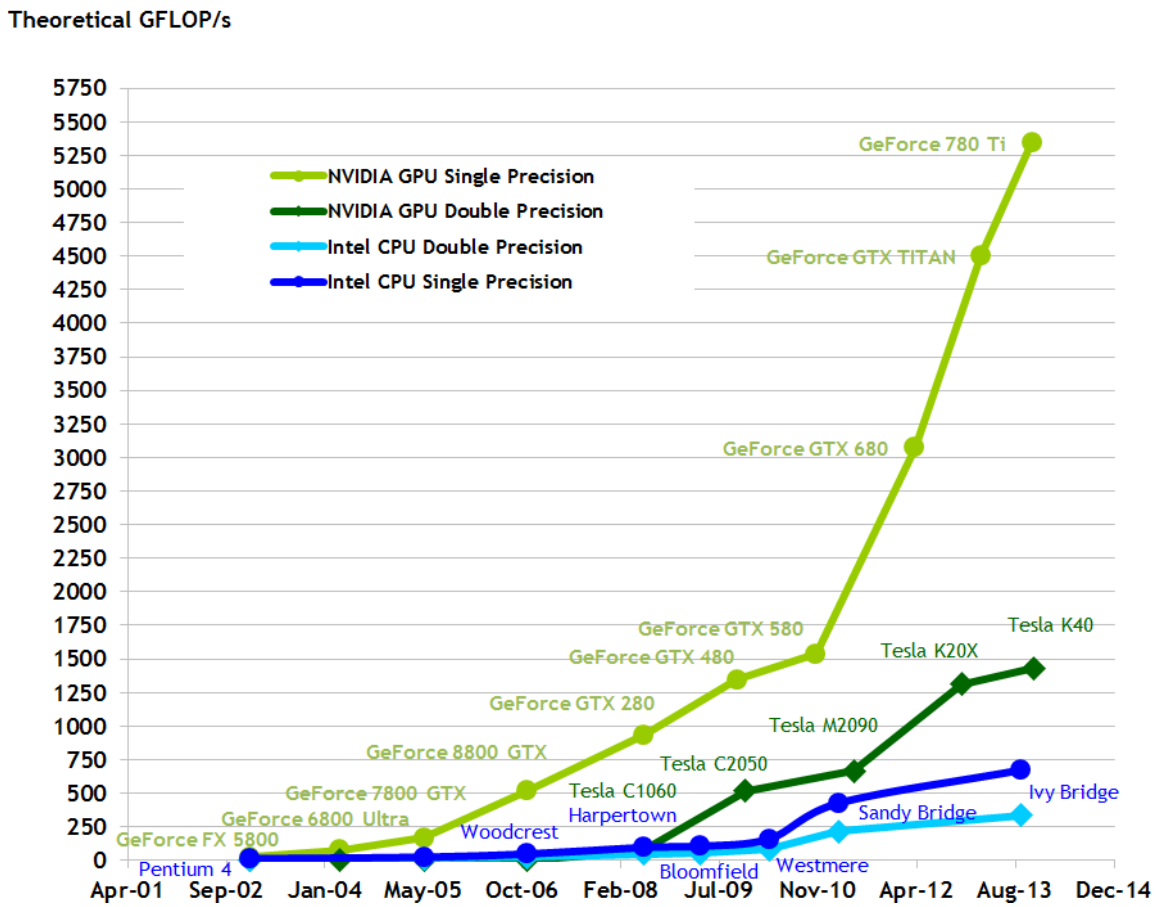


Figure 1 Floating-Point Operations per Second for the CPU and GPU

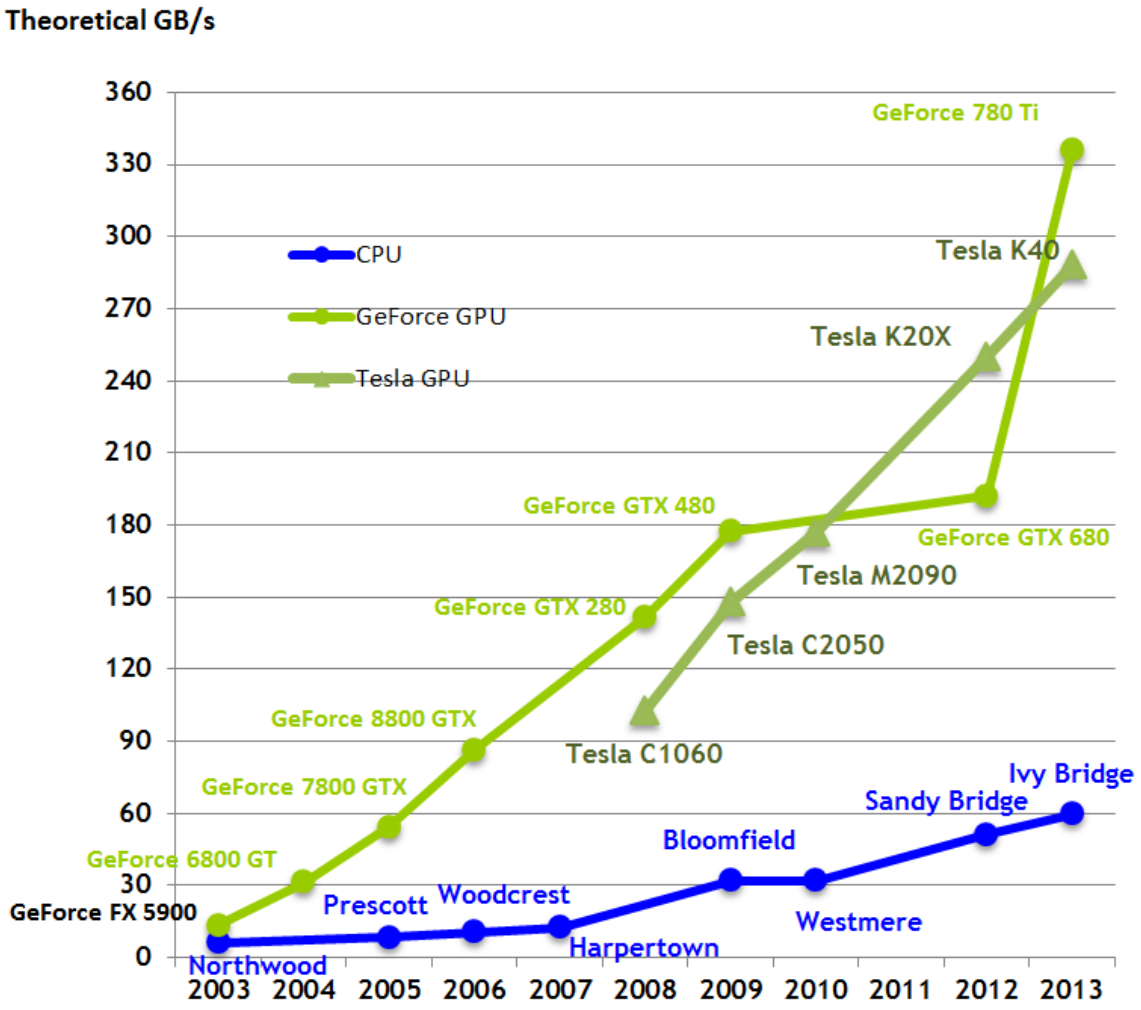


Figure 2 Memory Bandwidth for the CPU and GPU

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation - exactly what graphics rendering is about - and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 3.

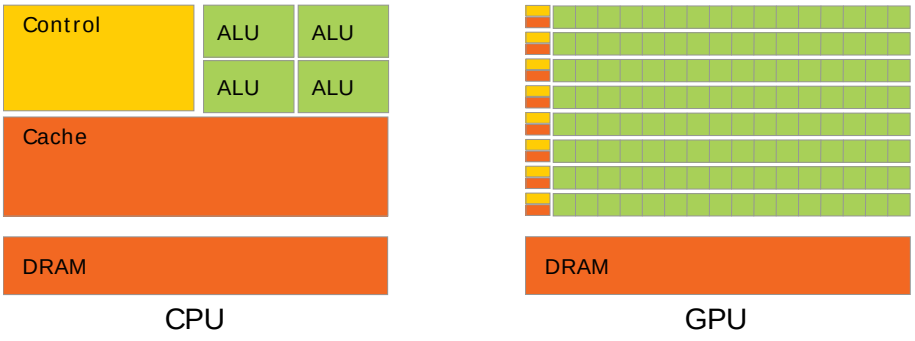


Figure 3 The GPU Devotes More Transistors to Data Processing

More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. Because the same program is executed for each data element, there is a lower requirement for sophisticated flow control, and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets can use a data-parallel programming model to speed up the computations. In 3D rendering, large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of rendered images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. In fact, many algorithms outside the field of image rendering and processing are accelerated by data-parallel processing, from general signal processing or physics simulation to computational finance or computational biology.

1.2. CUDA™: A General-Purpose Parallel Computing Platform and Programming Model

In November 2006, NVIDIA introduced CUDA™, a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU.

CUDA comes with a software environment that allows developers to use C as a high-level programming language. As illustrated by Figure 4, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenACC.

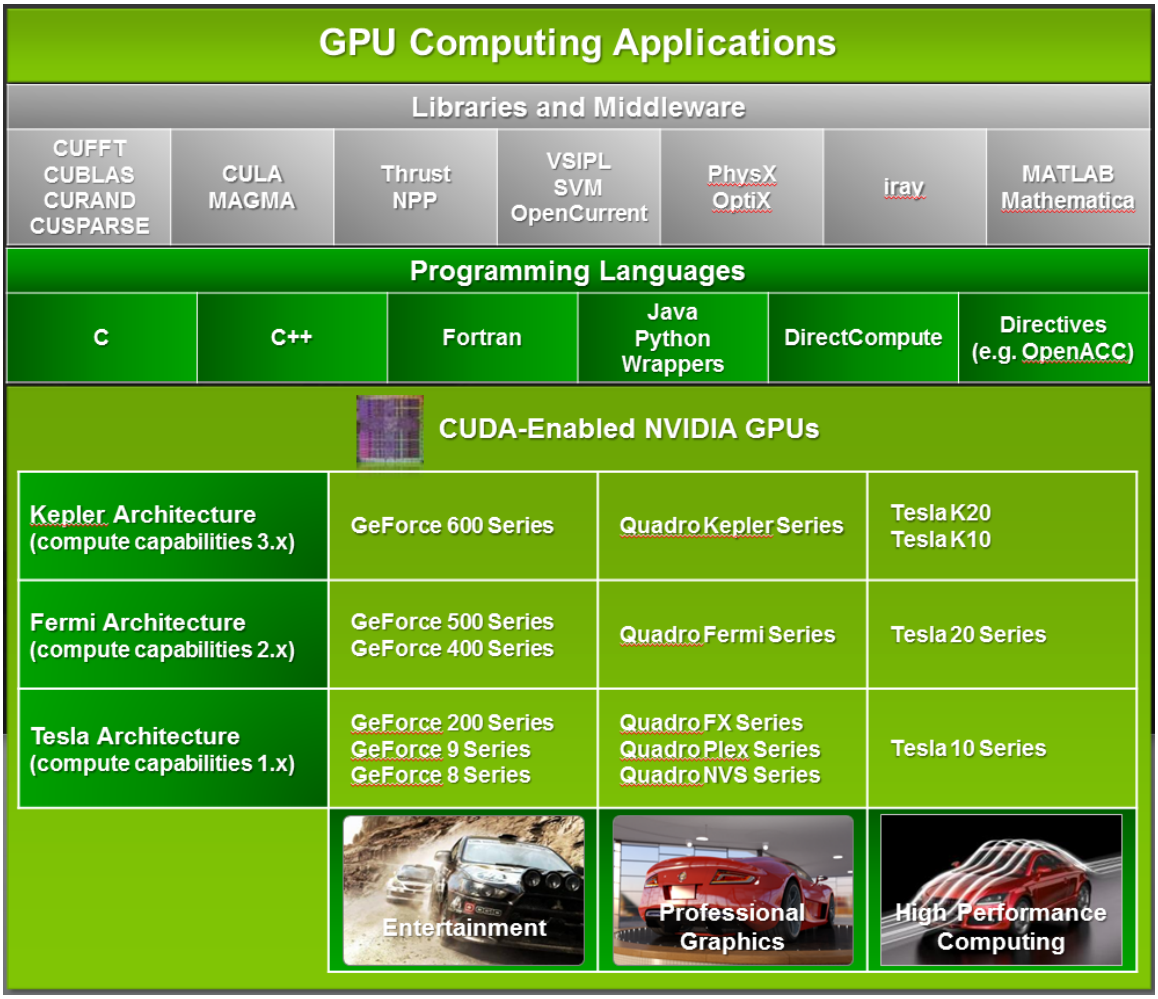


Figure 4 GPU Computing Applications

CUDA is designed to support various languages and application programming interfaces.

1.3. A Scalable Programming Model

The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to manycore GPUs with widely varying numbers of cores.

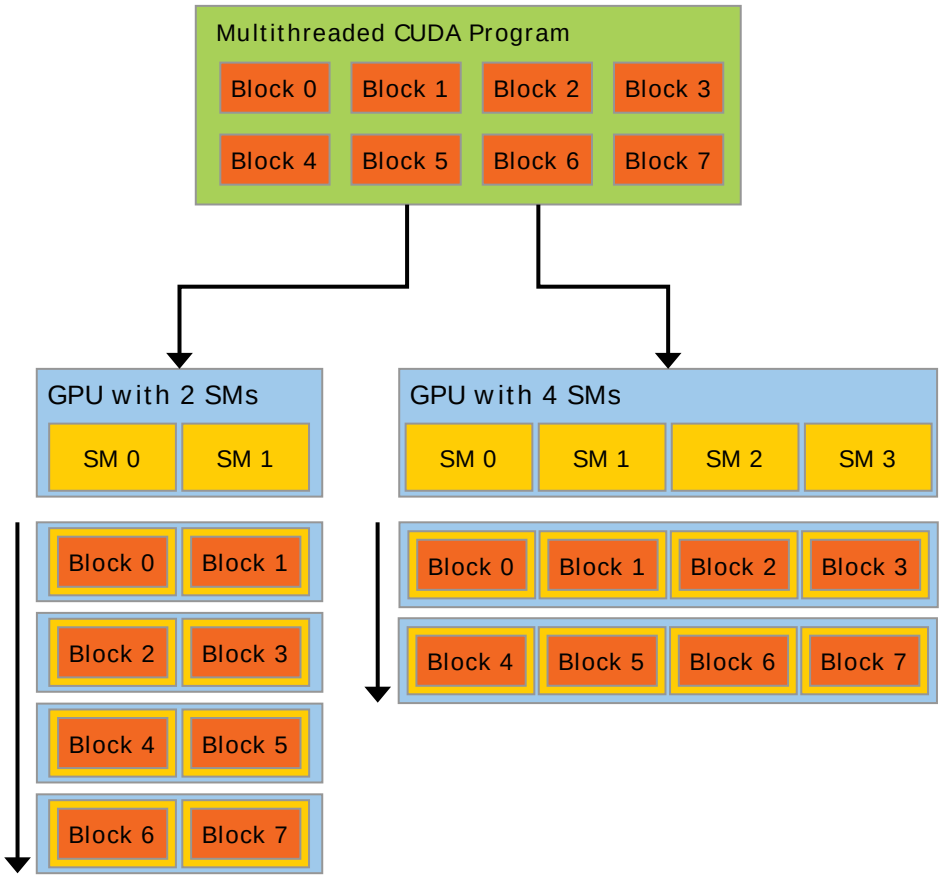
The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

At its core are three key abstractions - a hierarchy of thread groups, shared memories, and barrier synchronization - that are simply exposed to the programmer as a minimal set of language extensions.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available multiprocessors within a GPU, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of multiprocessors as illustrated by Figure 5, and only the runtime system needs to know the physical multiprocessor count.

This scalable programming model allows the GPU architecture to span a wide market range by simply scaling the number of multiprocessors and memory partitions: from the high-performance enthusiast GeForce GPUs and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs (see CUDA-Enabled GPUs for a list of all CUDA-enabled GPUs).



A GPU is built around an array of Streaming Multiprocessors (SMs) (see Hardware Implementation for more details). A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Figure 5 Automatic Scalability

1.4. Document Structure

This document is organized into the following chapters:

- Chapter Introduction is a general introduction to CUDA.
- Chapter Programming Model outlines the CUDA programming model.
- Chapter Programming Interface describes the programming interface.
- Chapter Hardware Implementation describes the hardware implementation.
- Chapter Performance Guidelines gives some guidance on how to achieve maximum performance.
- Appendix CUDA-Enabled GPUs lists all CUDA-enabled devices.
- Appendix C Language Extensions is a detailed description of all extensions to the C language.

- Appendix Mathematical Functions lists the mathematical functions supported in CUDA.
- Appendix C/C++ Language Support lists the C++ features supported in device code.
- Appendix Texture Fetching gives more details on texture fetching
- Appendix Compute Capabilities gives the technical specifications of various devices, as well as more architectural details.
- Appendix Driver API introduces the low-level driver API.
- Appendix CUDA Environment Variables lists all the CUDA environment variables.

Chapter 2. PROGRAMMING MODEL

This chapter introduces the main concepts behind the CUDA programming model by outlining how they are exposed in C. An extensive description of CUDA C is given in Programming Interface.

Full code for the vector addition example used in this chapter and the next can be found in the **vectorAdd** CUDA sample.

2.1. Kernels

CUDA C extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed N times in parallel by N different *CUDA threads*, as opposed to only once like regular C functions.

A kernel is defined using the **__global__** declaration specifier and the number of CUDA threads that execute that kernel for a given kernel call is specified using a new **<<<...>>>** *execution configuration* syntax (see C Language Extensions). Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in **threadIdx** variable.

As an illustration, the following sample code adds two vectors *A* and *B* of size *N* and stores the result into vector *C*:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

Here, each of the *N* threads that execute **VecAdd()** performs one pair-wise addition.

2.2. Thread Hierarchy

For convenience, **`threadIdx`** is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional *thread index*, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or volume.

The index of a thread and its thread ID relate to each other in a straightforward way: For a one-dimensional block, they are the same; for a two-dimensional block of size (D_x, D_y),the thread ID of a thread of index (x, y) is $(x + y\,D_x)$; for a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + y\,D_x + z\,D_x\,D_y)$.

As an example, the following code adds two matrices A and B of size NxN and stores the result into matrix C:

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                       float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

There is a limit to the number of threads per block, since all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads.

However, a kernel can be executed by multiple equally-shaped thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional *grid* of thread blocks as illustrated by Figure 6. The number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed.

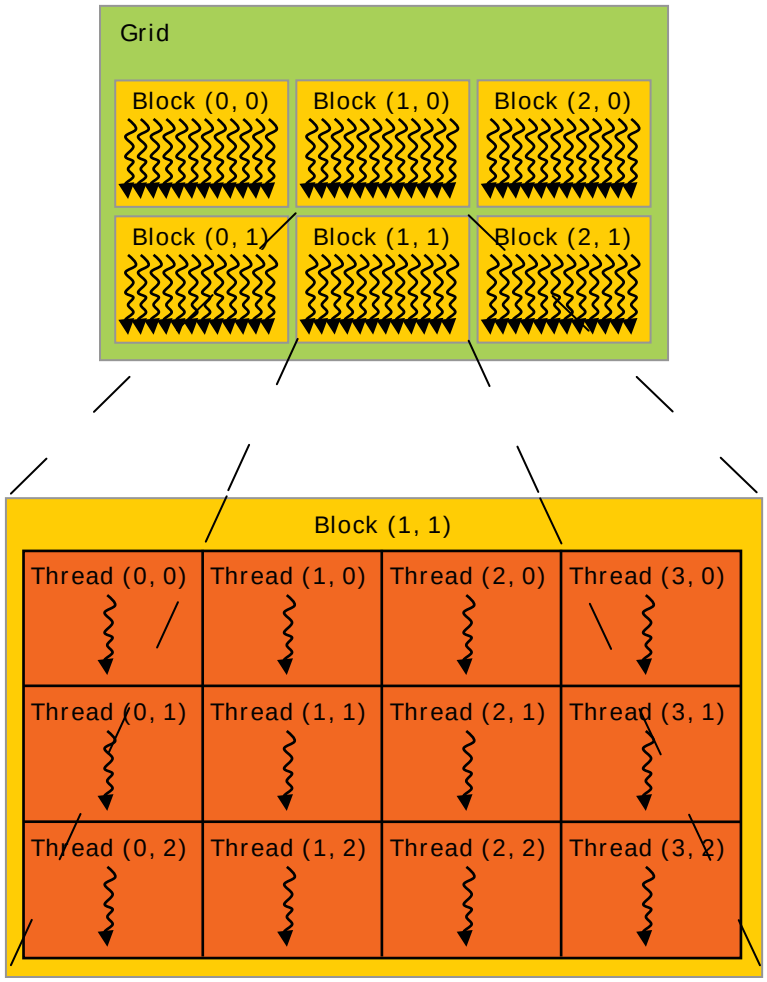


Figure 6 Grid of Thread Blocks

The number of threads per block and the number of blocks per grid specified in the **<<<...>>>** syntax can be of type **int** or **dim3**. Two-dimensional blocks or grids can be specified as in the example above.

Each block within the grid can be identified by a one-dimensional, two-dimensional, or three-dimensional index accessible within the kernel through the built-in **blockIdx** variable. The dimension of the thread block is accessible within the kernel through the built-in **blockDim** variable.

Extending the previous **MatAdd()** example to handle multiple blocks, the code becomes as follows.

```
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

A thread block size of 16x16 (256 threads), although arbitrary in this case, is a common choice. The grid is created with enough blocks to have one thread per matrix element as before. For simplicity, this example assumes that the number of threads per grid in each dimension is evenly divisible by the number of threads per block in that dimension, although that need not be the case.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores as illustrated by Figure 1 4, enabling programmers to write code that scales with the number of cores.

Threads within a block can cooperate by sharing data through some *shared memory* and by synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the **__syncthreads()** intrinsic function; **__syncthreads()** acts as a barrier at which all threads in the block must wait before any is allowed to proceed. Shared Memory gives an example of using shared memory.

For efficient cooperation, the 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.

2.3. Memory Hierarchy

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 7. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory.

There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages (see Device Memory Accesses). Texture

memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Texture and Surface Memory).

The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

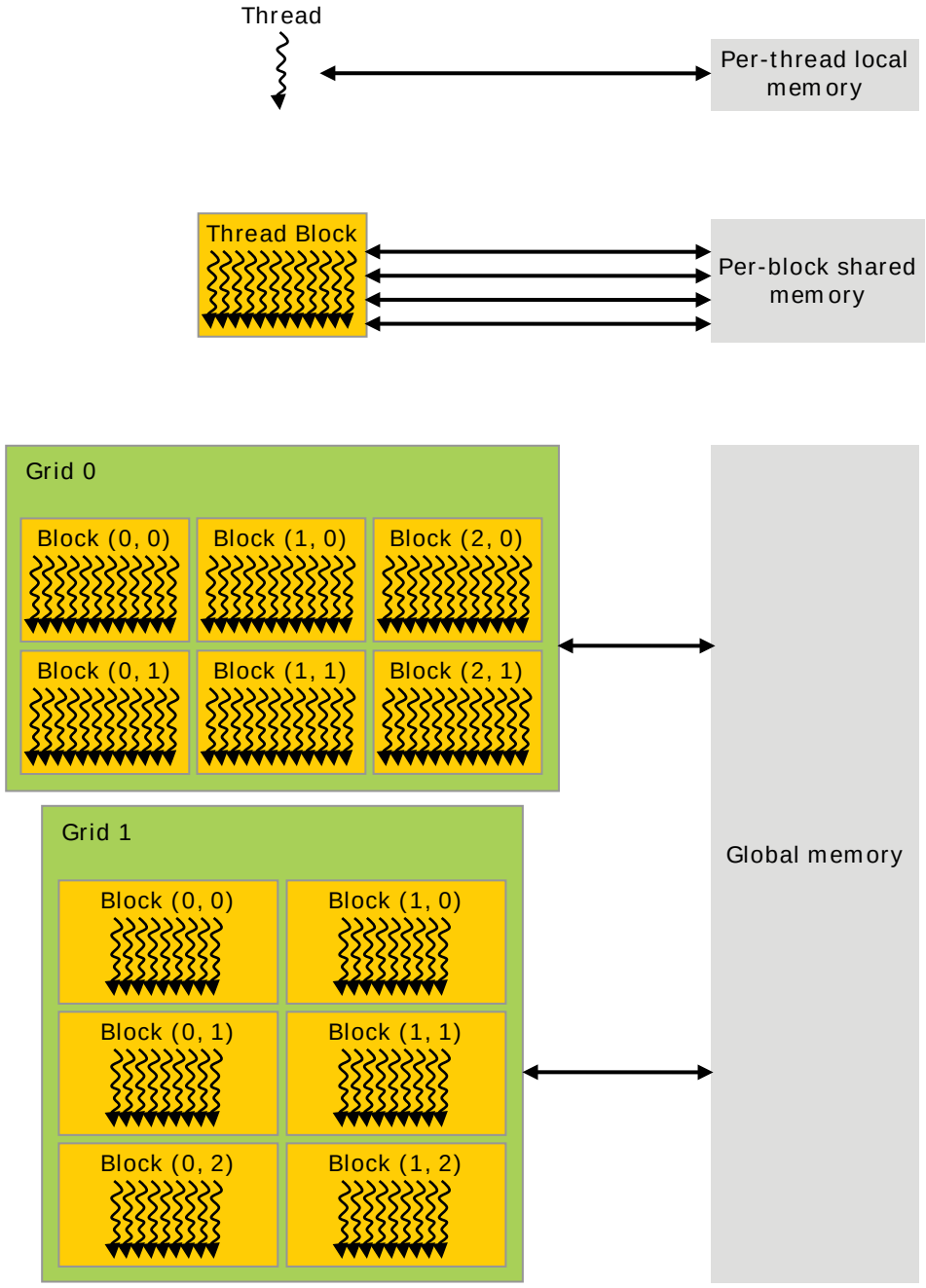
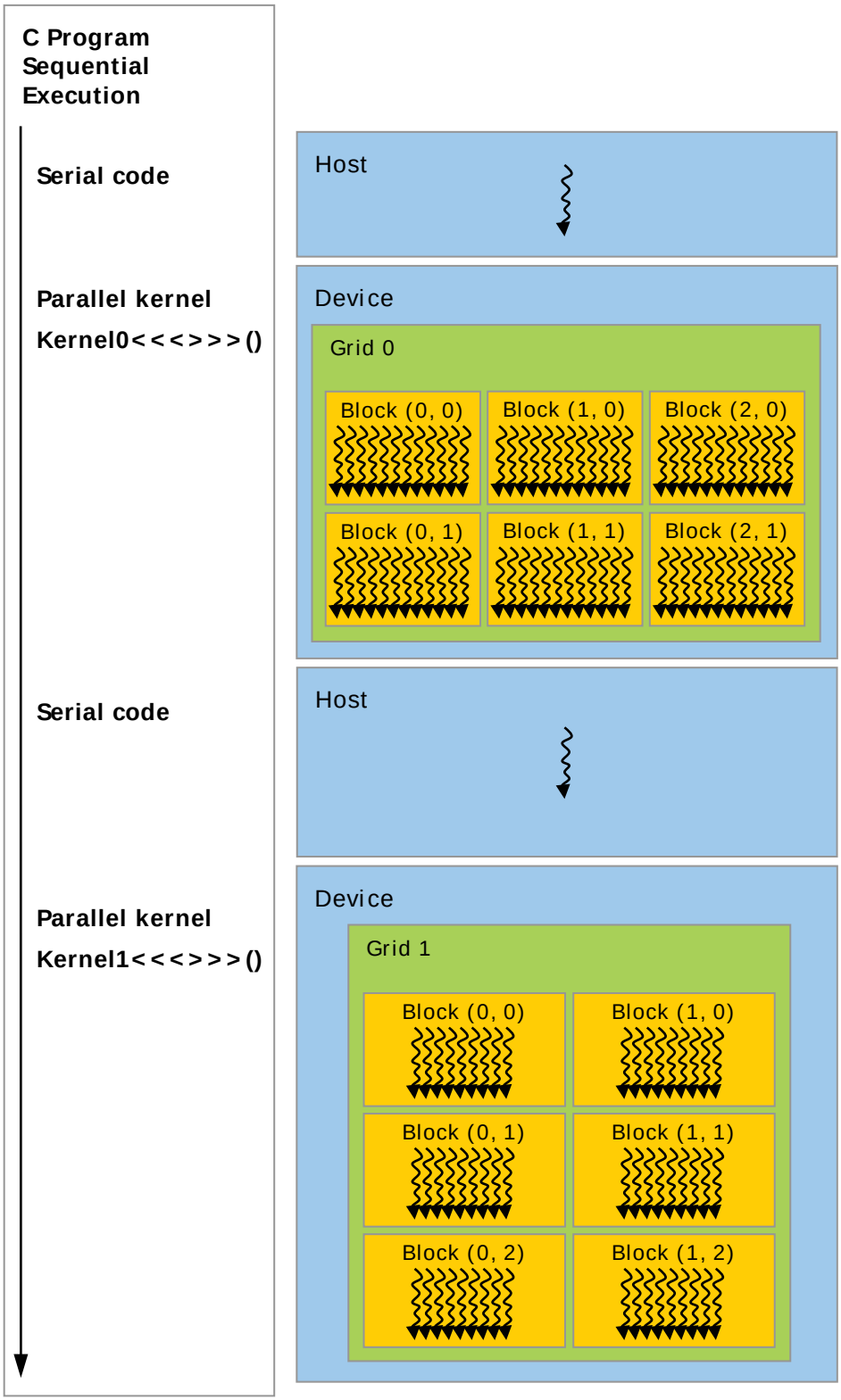


Figure 7 Memory Hierarchy

2.4. Heterogeneous Programming

As illustrated by Figure 8, the CUDA programming model assumes that the CUDA threads execute on a physically separate *device* that operates as a coprocessor to the host running the C program. This is the case, for example, when the kernels execute on a GPU and the rest of the C program executes on a CPU.

The CUDA programming model also assumes that both the *host* and the device maintain their own separate memory spaces in DRAM, referred to as *host memory* and *device memory*, respectively. Therefore, a program manages the global, constant, and texture memory spaces visible to kernels through calls to the CUDA runtime (described in Programming Interface). This includes device memory allocation and deallocation as well as data transfer between host and device memory.



Serial code executes on the host while parallel code executes on the device.

Figure 8 Heterogeneous Programming

2.5. Compute Capability

The *compute capability* of a device is defined by a major revision number and a minor revision number.

Devices with the same major revision number are of the same core architecture. The major revision number is 5 for devices based on the *Maxwell* architecture, 3 for devices based on the *Kepler* architecture, 2 for devices based on the *Fermi* architecture, and 1 for devices based on the *Tesla* architecture.

The minor revision number corresponds to an incremental improvement to the core architecture, possibly including new features.

CUDA-Enabled GPUs lists of all CUDA-enabled devices along with their compute capability. Compute Capabilities gives the technical specifications of each compute capability.

Chapter 3. PROGRAMMING INTERFACE

CUDA C provides a simple path for users familiar with the C programming language to easily write programs for execution by the device.

It consists of a minimal set of extensions to the C language and a runtime library.

The core language extensions have been introduced in Programming Model. They allow programmers to define a kernel as a C function and use some new syntax to specify the grid and block dimension each time the function is called. A complete description of all extensions can be found in C Language Extensions. Any source file that contains some of these extensions must be compiled with **nvcc** as outlined in Compilation with NVCC.

The runtime is introduced in Compilation Workflow. It provides C functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc. A complete description of the runtime can be found in the CUDA reference manual.

The runtime is built on top of a lower-level C API, the CUDA driver API, which is also accessible by the application. The driver API provides an additional level of control by exposing lower-level concepts such as CUDA contexts - the analogue of host processes for the device - and CUDA modules - the analogue of dynamically loaded libraries for the device. Most applications do not use the driver API as they do not need this additional level of control and when using the runtime, context and module management are implicit, resulting in more concise code. The driver API is introduced in Driver API and fully described in the reference manual.

3.1. Compilation with NVCC

Kernels can be written using the CUDA instruction set architecture, called *PTX*, which is described in the PTX reference manual. It is however usually more effective to use a high-level programming language such as C. In both cases, kernels must be compiled into binary code by **nvcc** to execute on the device.

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 section gives

an overview of **nvcc** workflow and command options. A complete description can be found in the **nvcc** user manual.

3.1.1. Compilation Workflow

3.1.1.1. Offline Compilation

Source files compiled with **nvcc** can include a mix of host code (i.e., code that executes on the host) and device code (i.e., code that executes on the device). **nvcc**'s basic workflow consists in separating device code from host code and then:

- compiling the device code into an assembly form (*PTX* code) and/or binary form (*cubin* object),
- and modifying the host code by replacing the **<<<...>>>** syntax introduced in Kernels (and described in more details in Execution Configuration) by the necessary CUDA C runtime function calls to load and launch each compiled kernel from the *PTX* code and/or *cubin* object.

The modified host code is output either as C code that is left to be compiled using another tool or as object code directly by letting **nvcc** invoke the host compiler during the last compilation stage.

Applications can then:

- Either link to the compiled host code (this is the most common case),
- Or ignore the modified host code (if any) and use the CUDA driver API (see Driver API) to load and execute the *PTX* code or *cubin* object.

3.1.1.2. Just-in-Time Compilation

Any *PTX* code loaded by an application at runtime is compiled further to binary code by the device driver. This is called *just-in-time 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 is also the only way for applications to run on devices that did not exist at the time the application was compiled, as detailed in Application Compatibility.

When the device driver just-in-time compiles some *PTX* code for some 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 - referred to as *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.

Environment variables are available to control just-in-time compilation as described in CUDA Environment Variables

3.1.2. Binary Compatibility

Binary code is architecture-specific. A *cubin* object is generated using the compiler option **-code** that specifies the targeted architecture: For example, compiling with **-code=sm_13** produces binary code for devices of compute capability 1.3 (see Compute

Capability for a description of the compute capability). Binary compatibility is guaranteed from one minor revision to the next one, but not from one minor revision to the previous one or across major revisions. In other words, a *cubin* object generated for compute capability *X.y* is only guaranteed to execute on devices of compute capability *X.z* where *z≥y*.

3.1.3. PTX Compatibility

Some *PTX* instructions are only supported on devices of higher compute capabilities. For example, atomic instructions on global memory are only supported on devices of compute capability 1.1 and above; double-precision instructions are only supported on devices of compute capability 1.3 and above. The **-arch** compiler option specifies the compute capability that is assumed when compiling C to *PTX* code. So, code that contains double-precision arithmetic, for example, must be compiled with **-arch=sm_13** (or higher compute capability), otherwise double-precision arithmetic will get demoted to single-precision arithmetic.

PTX code produced for some specific compute capability can always be compiled to binary code of greater or equal compute capability.

3.1.4. Application Compatibility

To execute code on devices of specific compute capability, an application must load binary or *PTX* code that is compatible with this compute capability as described in Binary Compatibility and PTX Compatibility. In particular, to be able to execute code on future architectures with higher compute capability (for which no binary code can be generated yet), an application must load *PTX* code that will be just-in-time compiled for these devices (see Just-in-Time Compilation).

Which *PTX* and binary code gets embedded in a CUDA C application is controlled by the **-arch** and **-code** compiler options or the **-gencode** compiler option as detailed in the **nvcc** user manual. For example,

```
nvcc x.cu
        -gencode arch=compute_10,code=sm_10
        -gencode arch=compute_11,code=\'compute_11,sm_11\'
```

embeds binary code compatible with compute capability 1.0 (first **-gencode** option) and *PTX* and binary code compatible with compute capability 1.1 (second **-gencode** option).

Host code is generated to automatically select at runtime the most appropriate code to load and execute, which, in the above example, will be:

- 1.0 binary code for devices with compute capability 1.0,
- 1.1 binary code for devices with compute capability 1.1, 1.2, 1.3,
- binary code obtained by compiling 1.1 *PTX* code for devices with compute capabilities 2.0 and higher.

x.cu can have an optimized code path that uses atomic operations, for example, which are only supported in devices of compute capability 1.1 and higher. The **__CUDA_ARCH__** macro can be used to differentiate various code paths based on compute capability. It is only defined for device code. When compiling with **-arch=compute_11** for example, **__CUDA_ARCH__** is equal to **110**.

Applications using the driver API must compile code to separate files and explicitly load and execute the most appropriate file at runtime.

The **nvcc** user manual lists various shorthand for the **-arch**, **-code**, and **-gencode** compiler options. For example, **-arch=sm_13** is a shorthand for **-arch=compute_13 -code=compute_13,sm_13** (which is the same as **-gencode arch=compute_13,code=\'compute_13,sm_13\'**).

3.1.5. C/C++ Compatibility

The front end of the compiler processes CUDA source files according to C++ syntax rules. Full C++ is supported for the host code. However, only a subset of C++ is fully supported for the device code as described in C/C++ Language Support.

3.1.6. 64-Bit Compatibility

The 64-bit version of **nvcc** compiles device code in 64-bit mode (i.e., pointers are 64-bit). Device code compiled in 64-bit mode is only supported with host code compiled in 64-bit mode.

Similarly, the 32-bit version of **nvcc** compiles device code in 32-bit mode and device code compiled in 32-bit mode is only supported with host code compiled in 32-bit mode.

The 32-bit version of **nvcc** can compile device code in 64-bit mode also using the **-m64** compiler option.

The 64-bit version of **nvcc** can compile device code in 32-bit mode also using the **-m32** compiler option.

3.2. CUDA C Runtime

The runtime is implemented in the **cudart** library, which is linked to the application, either statically via **cudart.lib** or **libcudart.a**, or dynamically via **cudart.dll** or **libcudart.so**. Applications that require **cudart.dll** and/or **cudart.so** for dynamic linking typically include them as part of the application installation package.

All its entry points are prefixed with **cuda**.

As mentioned in Heterogeneous Programming, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Device Memory gives an overview of the runtime functions used to manage device memory.

Shared Memory illustrates the use of shared memory, introduced in Thread Hierarchy, to maximize performance.

Page-Locked Host Memory introduces page-locked host memory that is required to overlap kernel execution with data transfers between host and device memory.

Asynchronous Concurrent Execution describes the concepts and API used to enable asynchronous concurrent execution at various levels in the system.

Multi-Device System shows how the programming model extends to a system with multiple devices attached to the same host.

Error Checking describes how to properly check the errors generated by the runtime.

Call Stack mentions the runtime functions used to manage the CUDA C call stack.

Texture and Surface Memory presents the texture and surface memory spaces that provide another way to access device memory; they also expose a subset of the GPU texturing hardware.

Graphics Interoperability introduces the various functions the runtime provides to interoperate with the two main graphics APIs, OpenGL and Direct3D.

3.2.1. Initialization

There is no explicit initialization function for the runtime; it initializes the first time a runtime function is called (more specifically any function other than functions from the device and version management sections of the reference manual). One needs to keep this in mind when timing runtime function calls and when interpreting the error code from the first call into the runtime.

During initialization, the runtime creates a CUDA context for each device in the system (see Context for more details on CUDA contexts). This context is the *primary context* for this device and it is shared among all the host threads of the application. As part of this context creation, the device code is just-in-time compiled if necessary (see Just-in-Time Compilation) and loaded into device memory. This all happens under the hood and the runtime does not expose the primary context to the application.

When a host thread calls **`cudaDeviceReset()`**, this destroys the primary context of the device the host thread currently operates on (i.e., the current device as defined in Device Selection). The next runtime function call made by any host thread that has this device as current will create a new primary context for this device.

3.2.2. Device Memory

As mentioned in Heterogeneous Programming, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels operate out of device memory, so the runtime provides functions to allocate, deallocate, and copy device memory, as well as transfer data between host memory and device memory.

Device memory can be allocated either as *linear memory* or as *CUDA arrays*.

CUDA arrays are opaque memory layouts optimized for texture fetching. They are described in Texture and Surface Memory.

Linear memory exists on the device in a 32-bit address space for devices of compute capability 1.x and 40-bit address space of devices of higher compute capability, so separately allocated entities can reference one another via pointers, for example, in a binary tree.

Linear memory is typically allocated using **`cudaMalloc()`** and freed using **`cudaFree()`** and data transfer between host memory and device memory are typically done using

cudaMemcpy(). In the vector addition code sample of Kernels, the vectors need to be copied from host memory to device memory:

```
// Device code
__global__ void VecAdd(float* A, float* B, float* C, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}

// Host code
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
    ...

    // Allocate vectors in device memory
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C, N);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Linear memory can also be allocated through **cudaMallocPitch()** and **cudaMalloc3D()**. These functions are recommended for allocations of 2D or 3D arrays as it makes sure that the allocation is appropriately padded to meet the alignment requirements described in Device Memory Accesses, therefore ensuring best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the **cudaMemcpy2D()** and **cudaMemcpy3D()** functions). The returned pitch (or stride) must be used to access array elements. The

following code sample allocates a **width** x **height** 2D array of floating-point values and shows how to loop over the array elements in device code:

```
// Host code
int width = 64, height = 64;
float* devPtr;
size_t pitch;
cudaMallocPitch(&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch, width, height);

// Device code
__global__ void MyKernel(float* devPtr,
                         size_t pitch, int width, int height)
{
    for (int r = 0; r < height; ++r) {
        float* row = (float*)((char*)devPtr + r * pitch);
        for (int c = 0; c > width; ++c) {
            float element = row[c];
        }
    }
}
```

The following code sample allocates a **width** x **height** x **depth** 3D array of floating-point values and shows how to loop over the array elements in device code:

```
// Host code
int width = 64, height = 64, depth = 64;
cudaExtent extent = make_cudaExtent(width * sizeof(float),
                                    height, depth);
cudaPitchedPtr devPitchedPtr;
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, width, height, depth);

// Device code
__global__ void MyKernel(cudaPitchedPtr devPitchedPtr,
                         int width, int height, int depth)
{
    char* devPtr = devPitchedPtr.ptr;
    size_t pitch = devPitchedPtr.pitch;
    size_t slicePitch = pitch * height;
    for (int z = 0; z < depth; ++z) {
        char* slice = devPtr + z * slicePitch;
        for (int y = 0; y < height; ++y) {
            float* row = (float*)(slice + y * pitch);
            for (int x = 0; x < width; ++x) {
                float element = row[x];
            }
        }
    }
}
```

The reference manual lists all the various functions used to copy memory between linear memory allocated with **cudaMalloc()**, linear memory allocated with **cudaMallocPitch()** or **cudaMalloc3D()**, CUDA arrays, and memory allocated for variables declared in global or constant memory space.

The following code sample illustrates various ways of accessing global variables via the runtime API:

```
__constant__ float constData[256];
float data[256];
cudaMemcpyToSymbol(constData, data, sizeof(data));
cudaMemcpyFromSymbol(data, constData, sizeof(data));

__device__ float devData;
float value = 3.14f;
cudaMemcpyToSymbol(devData, &value, sizeof(float));

__device__ float* devPointer;
float* ptr;
cudaMalloc(&ptr, 256 * sizeof(float));
cudaMemcpyToSymbol(devPointer, &ptr, sizeof(ptr));
```

`cudaGetSymbolAddress()` is used to retrieve the address pointing to the memory allocated for a variable declared in global memory space. The size of the allocated memory is obtained through **`cudaGetSymbolSize()`**.

3.2.3. Shared Memory

As detailed in Variable Type Qualifiers shared memory is allocated using the **`__shared__`** qualifier.

Shared memory is expected to be much faster than global memory as mentioned in Thread Hierarchy and detailed in Shared Memory. Any opportunity to replace global memory accesses by shared memory accesses should therefore be exploited as illustrated by the following matrix multiplication example.

The following code sample is a straightforward implementation of matrix multiplication that does not take advantage of shared memory. Each thread reads one row of A and one

column of *B* and computes the corresponding element of *C* as illustrated in Figure 9. *A* is therefore read *B.width* times from global memory and *B* is read *A.height* times.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.width + col)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
               cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
    cudaMemcpy(C.elements, Cd.elements, size,
               cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A.elements);
    cudaFree(d_B.elements);
    cudaFree(d_C.elements);
}

// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C)
{
    // Each thread computes one element of C
    // by accumulating results into Cvalue
    float Cvalue = 0;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}
```

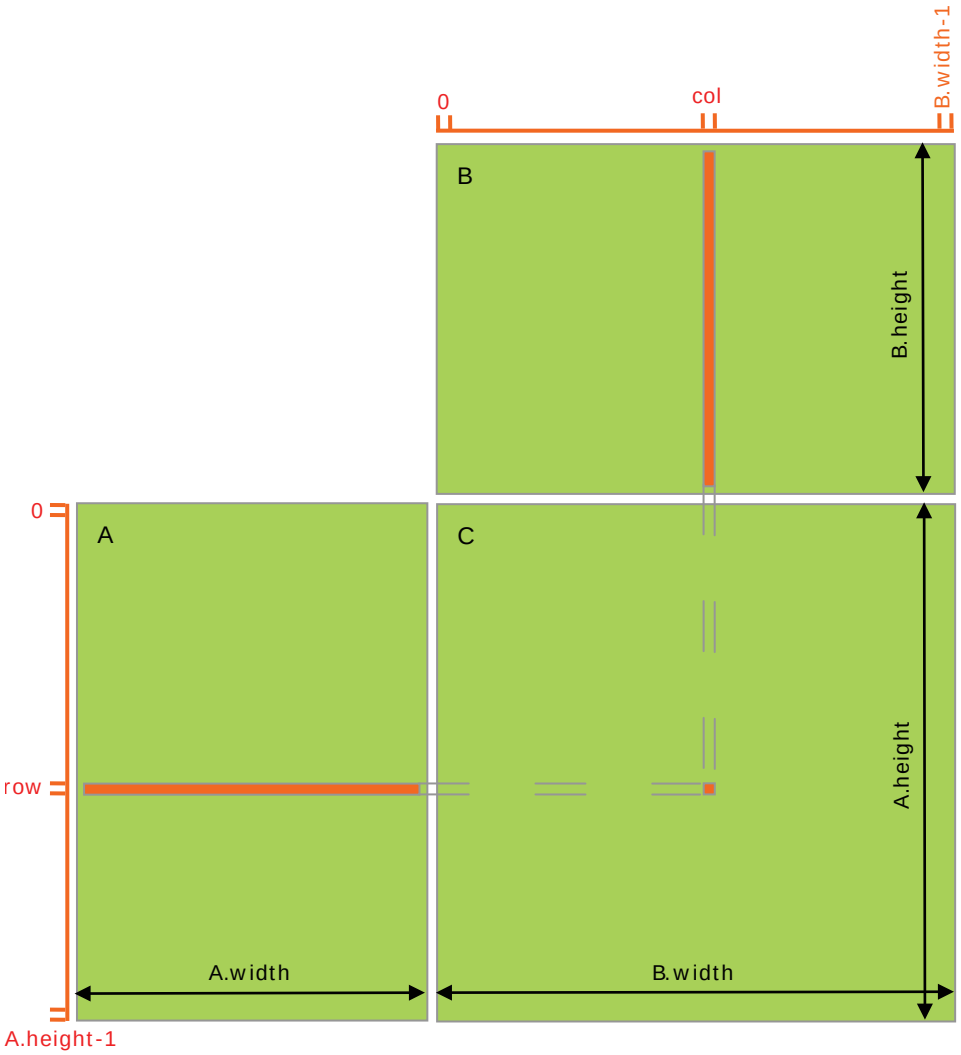


Figure 9 Matrix Multiplication without Shared Memory

The following code sample is an implementation of matrix multiplication that does take advantage of shared memory. In this implementation, each thread block is responsible for computing one square sub-matrix C_{sub} of C and each thread within the block is responsible for computing one element of C_{sub}. As illustrated in Figure 10, C_{sub} is equal to the product of two rectangular matrices: the sub-matrix of A of dimension (*A.width*, *block_size*) that has the same row indices as C_{sub}, and the sub-matrix of B of dimension (*block_size*, *A.width*)that has the same column indices as C_{sub}. In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension *block_size* as necessary and C_{sub} is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.

By blocking the computation this way, we take advantage of fast shared memory and save a lot of global memory bandwidth since *A* is only read (*B.width* / *block_size*) times from global memory and *B* is read (*A.height* / *block_size*) times.

The *Matrix* type from the previous code sample is augmented with a *stride* field, so that sub-matrices can be efficiently represented with the same type. **`__device__`** functions

(see __device__) are used to get and set elements and build any sub-matrix from a matrix.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
    float* elements;
} Matrix;

// Get a matrix element
__device__ float GetElement(const Matrix A, int row, int col)
{
    return A.elements[row * A.stride + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value)
{
    A.elements[row * A.stride + col] = value;
}

// Get the BLOCK_SIZExBLOCK_SIZE sub-matrix Asub of A that is
// located col sub-matrices to the right and row sub-matrices down
// from the upper-left corner of A
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col)
{
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A.stride;
    Asub.elements = &A.elements[A.stride * BLOCK_SIZE * row
                                         + BLOCK_SIZE * col];
    return Asub;
}

// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the matrix multiplication kernel
__global__ void MatMulKernel(const Matrix, const Matrix, Matrix);

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMul(const Matrix A, const Matrix B, Matrix C)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    cudaMalloc(&d_A.elements, size);
    cudaMemcpy(d_A.elements, A.elements, size,
               cudaMemcpyHostToDevice);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cudaMalloc(&d_B.elements, size);
    cudaMemcpy(d_B.elements, B.elements, size,
    cudaMemcpyHostToDevice);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = d_C.stride = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    cudaMalloc(&d_C.elements, size);

    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
    MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);

    // Read C from device memory
```

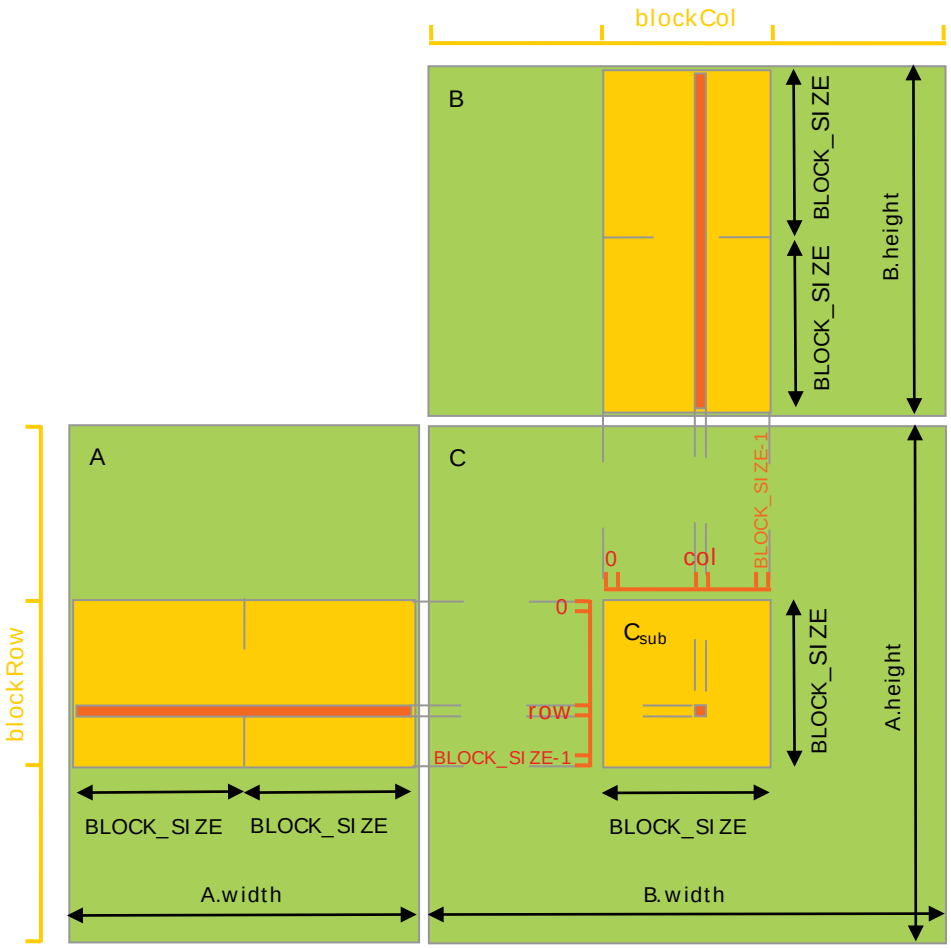


Figure 10 Matrix Multiplication with Shared Memory

3.2.4. Page-Locked Host Memory

The runtime provides functions to allow the use of *page-locked* (also known as *pinned*) host memory (as opposed to regular pageable host memory allocated by **`malloc()`**):

- **`cudaHostAlloc()`** and **`cudaFreeHost()`** allocate and free page-locked host memory;
- **`cudaHostRegister()`** page-locks a range of memory allocated by **`malloc()`** (see reference manual for limitations).

Using page-locked host memory has several benefits:

- Copies between page-locked host memory and device memory can be performed concurrently with kernel execution for some devices as mentioned in Asynchronous Concurrent Execution.
- On some devices, page-locked host memory can be mapped into the address space of the device, eliminating the need to copy it to or from device memory as detailed in Mapped Memory.
- On systems with a front-side bus, bandwidth between host memory and device memory is higher if host memory is allocated as page-locked and even higher if

in addition it is allocated as write-combining as described in Write-Combining Memory.

Page-locked host memory is a scarce resource however, so allocations in page-locked memory will start failing long before allocations in pageable memory. In addition, by reducing the amount of physical memory available to the operating system for paging, consuming too much page-locked memory reduces overall system performance.

The simple zero-copy CUDA sample comes with a detailed document on the page-locked memory APIs.

3.2.4.1. Portable Memory

A block of page-locked memory can be used in conjunction with any device in the system (see Multi-Device System for more details on multi-device systems), but by default, the benefits of using page-locked memory described above are only available in conjunction with the device that was current when the block was allocated (and with all devices sharing the same unified address space, if any, as described in Unified Virtual Address Space). To make these advantages available to all devices, the block needs to be allocated by passing the flag **`cudaHostAllocPortable`** to **`cudaHostAlloc()`** or page-locked by passing the flag **`cudaHostRegisterPortable`** to **`cudaHostRegister()`**.

3.2.4.2. Write-Combining Memory

By default page-locked host memory is allocated as cacheable. It can optionally be allocated as *write-combining* instead by passing flag **`cudaHostAllocWriteCombined`** to **`cudaHostAlloc()`**. Write-combining memory frees up the host's L1 and L2 cache resources, making more cache available to the rest of the application. In addition, write-combining memory is not snooped during transfers across the PCI Express bus, which can improve transfer performance by up to 40%.

Reading from write-combining memory from the host is prohibitively slow, so write-combining memory should in general be used for memory that the host only writes to.

3.2.4.3. Mapped Memory

On devices of compute capability greater than 1.0, a block of page-locked host memory can also be mapped into the address space of the device by passing flag **`cudaHostAllocMapped`** to **`cudaHostAlloc()`** or by passing flag **`cudaHostRegisterMapped`** to **`cudaHostRegister()`**. Such a block has therefore in general two addresses: one in host memory that is returned by **`cudaHostAlloc()`** or **`malloc()`**, and one in device memory that can be retrieved using **`cudaHostGetDevicePointer()`** and then used to access the block from within a kernel. The only exception is for pointers allocated with **`cudaHostAlloc()`** and when a unified address space is used for the host and the device as mentioned in Unified Virtual Address Space.

Accessing host memory directly from within a kernel has several advantages:

- There is no need to allocate a block in device memory and copy data between this block and the block in host memory; data transfers are implicitly performed as needed by the kernel;

- There is no need to use streams (see Concurrent Data Transfers) to overlap data transfers with kernel execution; the kernel-originated data transfers automatically overlap with kernel execution.

Since mapped page-locked memory is shared between host and device however, the application must synchronize memory accesses using streams or events (see Asynchronous Concurrent Execution) to avoid any potential read-after-write, write-after-read, or write-after-write hazards.

To be able to retrieve the device pointer to any mapped page-locked memory, page-locked memory mapping must be enabled by calling **`cudaSetDeviceFlags()`** with the **`cudaDeviceMapHost`** flag before any other CUDA call is performed. Otherwise, **`cudaHostGetDevicePointer()`** will return an error.

`cudaHostGetDevicePointer()` also returns an error if the device does not support mapped page-locked host memory. Applications may query this capability by checking the **`canMapHostMemory`** device property (see Device Enumeration), which is equal to 1 for devices that support mapped page-locked host memory.

Note that atomic functions (see Atomic Functions) operating on mapped page-locked memory are not atomic from the point of view of the host or other devices.

3.2.5. Asynchronous Concurrent Execution

3.2.5.1. Concurrent Execution between Host and Device

In order to facilitate concurrent execution between host and device, some function calls are asynchronous: Control is returned to the host thread before the device has completed the requested task. These are:

- Kernel launches;
- Memory copies between two addresses to the same device memory;
- Memory copies from host to device of a memory block of 64 KB or less;
- Memory copies performed by functions that are suffixed with **`Async`**;
- Memory set function calls.

Programmers can globally disable asynchronous kernel launches for all CUDA applications running on a system by setting the **`CUDA_LAUNCH_BLOCKING`** environment variable to 1. This feature is provided for debugging purposes only and should never be used as a way to make production software run reliably.

Kernel launches are synchronous in the following cases:

- The application is run via a debugger or memory checker (cuda-gdb, cuda-memcheck, Nsight) on a device of compute capability 1.x;
- Hardware counters are collected via a profiler (Nsight, Visual Profiler).

3.2.5.2. Overlap of Data Transfer and Kernel Execution

Some devices of compute capability 1.1 and higher can perform copies between page-locked host memory and device memory concurrently with kernel execution. Applications may query this capability by checking the **`asyncEngineCount`** device

property (see Device Enumeration), which is greater than zero for devices that support it. For devices of compute capability 1.x, this capability is only supported for memory copies that do not involve CUDA arrays or 2D arrays allocated through **`cudaMallocPitch()`** (see Device Memory).

3.2.5.3. Concurrent Kernel Execution

Some devices of compute capability 2.x and higher can execute multiple kernels concurrently. Applications may query this capability by checking the **`concurrentKernels`** device property (see Device Enumeration), which is equal to 1 for devices that support it.

The maximum number of kernel launches that a device can execute concurrently is 32 on devices of compute capability 3.5 and 16 on devices of lower compute capability.

A kernel from one CUDA context cannot execute concurrently with a kernel from another CUDA context.

Kernels that use many textures or a large amount of local memory are less likely to execute concurrently with other kernels.

3.2.5.4. Concurrent Data Transfers

Some devices of compute capability 2.x and higher can perform a copy from page-locked host memory to device memory concurrently with a copy from device memory to page-locked host memory.

Applications may query this capability by checking the **`asyncEngineCount`** device property (see Device Enumeration), which is equal to 2 for devices that support it.

3.2.5.5. Streams

Applications manage concurrency through *streams*. A stream is a sequence of commands (possibly issued by different host threads) that execute in order. Different streams, on the other hand, may execute their commands out of order with respect to one another or concurrently; this behavior is not guaranteed and should therefore not be relied upon for correctness (e.g., inter-kernel communication is undefined).

3.2.5.5.1. Creation and Destruction

A stream is defined by creating a stream object and specifying it as the stream parameter to a sequence of kernel launches and host **`<->`** device memory copies. The following code sample creates two streams and allocates an array **`hostPtr`** of **`float`** in page-locked memory.

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Each of these streams is defined by the following code sample as a sequence of one memory copy from host to device, one kernel launch, and one memory copy from device to host:

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel <<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
```

Each stream copies its portion of input array **hostPtr** to array **inputDevPtr** in device memory, processes **inputDevPtr** on the device by calling **MyKernel()**, and copies the result **outputDevPtr** back to the same portion of **hostPtr**. Overlapping Behavior describes how the streams overlap in this example depending on the capability of the device. Note that **hostPtr** must point to page-locked host memory for any overlap to occur.

Streams are released by calling **cudaStreamDestroy()**.

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

cudaStreamDestroy() waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread.

3.2.5.5.2. Default Stream

Kernel launches and host **<->** device memory copies that do not specify any stream parameter, or equivalently that set the stream parameter to zero, are issued to the default stream. They are therefore executed in order.

3.2.5.5.3. 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 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. The stream can be 0, in which case all the commands added to any stream after the call to **cudaStreamWaitEvent()** wait on the event.

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

To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing.

3.2.5.5.4. 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 default stream,
- a switch between the L1/shared memory configurations described in Compute Capability 2.x and Compute Capability 3.x.

For devices that support concurrent kernel execution and are of compute capability 3.0 or lower, any operation that requires a dependency check to see if a streamed kernel launch is complete:

- Can start executing only when all thread blocks of all prior kernel launches from any stream in the CUDA context have started executing;
- Blocks all later kernel launches from any stream in the CUDA context until the kernel launch being checked is complete.

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.

3.2.5.5.5. Overlapping Behavior

The amount of execution overlap between two streams depends on the order in which the commands are issued to each stream and whether or not the device supports overlap of data transfer and kernel execution (see Overlap of Data Transfer and Kernel Execution), concurrent kernel execution (see Concurrent Kernel Execution), and/or concurrent data transfers (see Concurrent Data Transfers).

For example, on devices that do not support concurrent data transfers, the two streams of the code sample of Creation and Destruction do not overlap at all because the memory copy from host to device is issued to stream[1] after the memory copy from device to host is issued to stream[0], so it can only start once the memory copy from device to host issued to stream[0] has completed. If the code is rewritten the following way (and assuming the device supports overlap of data transfer and kernel execution)

```
for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
for (int i = 0; i < 2; ++i)
    MyKernel<<<100, 512, 0, stream[i]>>>
          (outputDevPtr + i * size, inputDevPtr + i * size, size);
    for (int i = 0; i < 2; ++i)
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
```

then the memory copy from host to device issued to stream[1] overlaps with the kernel launch issued to stream[0].

On devices that do support concurrent data transfers, the two streams of the code sample of Creation and Destruction do overlap: The memory copy from host to device issued to stream[1] overlaps with the memory copy from device to host issued to stream[0] and even with the kernel launch issued to stream[0] (assuming the device supports overlap of data transfer and kernel execution). However, for devices of compute capability 3.0 or lower, the kernel executions cannot possibly overlap because the second kernel launch is issued to stream[1] after the memory copy from device to host is issued to stream[0], so it is blocked until the first kernel launch issued to stream[0] is complete as per Implicit Synchronization. If the code is rewritten as above, the kernel executions overlap (assuming the device supports concurrent kernel execution) since the second kernel launch is issued to stream[1] before the memory copy from device to host is issued to stream[0]. In that case however, the memory copy from device to host issued to stream[0] only overlaps with the last thread blocks of the kernel launch issued to stream[1] as per Implicit Synchronization, which can represent only a small portion of the total execution time of the kernel.

3.2.5.5.6. Callbacks

The runtime provides a way to insert a callback at any point into a stream via **`cudaStreamAddCallback()`**. A callback is a function that is executed on the host once all commands issued to the stream before the callback have completed. Callbacks in stream 0 are executed once all preceding tasks and commands issued in all streams before the callback have completed.

The following code sample adds the callback function **`MyCallback`** to each of two streams after issuing a host-to-device memory copy, a kernel launch and a device-to-host memory copy into each stream. The callback will begin execution on the host after each of the device-to-host memory copies completes.

```
void CUDART_CB MyCallback(cudaStream_t stream, cudaError_t status, void *data){
    printf("Inside callback %d\n", (size_t)data);
}
...
for (size_t i = 0; i < 2; ++i) {
    cudaMemcpyAsync(devPtrIn[i], hostPtr[i], size, cudaMemcpyHostToDevice,
 stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(devPtrOut[i], devPtrIn[i], size);
    cudaMemcpyAsync(hostPtr[i], devPtrOut[i], size, cudaMemcpyDeviceToHost,
 stream[i]);
    cudaStreamAddCallback(stream[i], MyCallback, (void*)i, 0);
}
```

The commands that are issued in a stream (or all commands issued to any stream if the callback is issued to stream 0) after a callback do not start executing before the callback has completed. The last parameter of **`cudaStreamAddCallback()`** is reserved for future use.

A callback must not make CUDA API calls (directly or indirectly), as it might end up waiting on itself if it makes such a call leading to a deadlock.

3.2.5.5.7. Stream Priorities

The relative priorities of streams can be specified at creation using **`cudaStreamCreateWithPriority()`**. The range of allowable priorities, ordered as [highest priority, lowest priority] can be obtained using the **`cudaDeviceGetStreamPriorityRange()`** function. At runtime, as blocks in low-priority schemes finish, waiting blocks in higher-priority streams are scheduled in their place.

The following code sample obtains the allowable range of priorities for the current device, and creates streams with the highest and lowest available priorities

```
// get the range of stream priorities for this device
int priority_high, priority_low;
cudaDeviceGetStreamPriorityRange(&priority_low, &priority_high);
// create streams with highest and lowest available priorities
cudaStream_t st_high, st_low;
cudaStreamCreateWithPriority(&st_high, cudaStreamNonBlocking, &priority_high);
cudaStreamCreateWithPriority(&st_low, cudaStreamNonBlocking, &priority_low);
```

3.2.5.6. Events

The runtime also provides a way to closely monitor the device's progress, as well as perform accurate timing, by letting the application asynchronously record *events* at any point in the program and query when these events are completed. An event has completed when all tasks - or optionally, all commands in a given stream - preceding the event have completed. Events in stream zero are completed after all preceding tasks and commands in all streams are completed.

3.2.5.6.1. Creation and Destruction

The following code sample creates two events:

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
```

They are destroyed this way:

```
cudaEventDestroy(start);
cudaEventDestroy(stop);
```

3.2.5.6.2. Elapsed Time

The events created in Creation and Destruction can be used to time the code sample of Creation and Destruction the following way:

```
cudaEventRecord(start, 0);
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDev + i * size, inputHost + i * size,
                    size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>
               (outputDev + i * size, inputDev + i * size, size);
    cudaMemcpyAsync(outputHost + i * size, outputDev + i * size,
                    size, cudaMemcpyDeviceToHost, stream[i]);
}
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);
```

3.2.5.7. Synchronous Calls

When a synchronous function is called, control is not returned to the host thread before the device has completed the requested task. Whether the host thread will then yield, block, or spin can be specified by calling **cudaSetDeviceFlags()** with some specific flags (see reference manual for details) before any other CUDA call is performed by the host thread.

3.2.6. Multi-Device System

3.2.6.1. Device Enumeration

A host system can have multiple devices. The following code sample shows how to enumerate these devices, query their properties, and determine the number of CUDA-enabled devices.

```
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);
}
```

3.2.6.2. Device Selection

A host thread can set the device it operates on at any time by calling **cudaSetDevice()**. Device memory allocations and kernel launches are made on the currently set device; streams and events are created in association with the currently set device. If no call to **cudaSetDevice()** is made, the current device is device 0.

The following code sample illustrates how setting the current device affects memory allocation and kernel execution.

```
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
```

3.2.6.3. Stream and Event 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:
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 (see 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.

3.2.6.4. Peer-to-Peer Memory Access

When the application is run as a 64-bit process, devices of compute capability 2.0 and higher from the Tesla series may address each other's memory (i.e., a kernel executing on one device can dereference a pointer to the memory of the other device). This peer-to-peer memory access feature is supported between two devices if **cudaDeviceCanAccessPeer()** returns true for these two devices.

Peer-to-peer memory access must be enabled between two devices by calling **cudaDeviceEnablePeerAccess()** as illustrated in the following code sample.

A unified 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);
```

3.2.6.5. Peer-to-Peer Memory Copy

Memory copies can be performed between the memories of two different devices.

When a unified address space is used for both devices (see Unified Virtual Address Space), this is done using the regular memory copy functions mentioned in Device Memory.

Otherwise, this is done 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 Concurrent 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.

Note that if peer-to-peer access is enabled between two devices via **`cudaDeviceEnablePeerAccess()`** as described in Peer-to-Peer Memory Access, peer-to-peer memory copy between these two devices no longer needs to be staged through the host and is therefore faster.

3.2.7. Unified Virtual Address Space

When the application is run as a 64-bit process, a single address space is used for the host and all the devices of compute capability 2.0 and higher. This address space is used for all allocations made in host memory via `cudaHostAlloc()` and in any of the device memories via `cudaMalloc*()`. Which memory a pointer points to - host memory or any of the device memories - can be determined from the value of the pointer using `cudaPointerGetAttributes()`. As a consequence:

- When copying from or to the memory of one of the devices for which the unified address space is used, the `cudaMemcpyKind` parameter of `cudaMemcpy*()` becomes useless and can be set to `cudaMemcpyDefault`;
- Allocations via `cudaHostAlloc()` are automatically portable (see Portable Memory) across all the devices for which the unified address space is used, and pointers returned by `cudaHostAlloc()` can be used directly from within kernels running on these devices (i.e., there is no need to obtain a device pointer via `cudaHostGetDevicePointer()` as described in Mapped Memory.

Applications may query if the unified address space is used for a particular device by checking that the `unifiedAddressing` device property (see Device Enumeration) is equal to 1.

3.2.8. Interprocess Communication

Any device memory pointer or event handle created by a host thread can be directly referenced by any other thread within the same process. It is not valid outside this process however, and therefore cannot be directly referenced by threads belonging to a different process.

To share device memory pointers and events across processes, an application must use the Inter Process Communication API, which is described in detail in the reference manual. The IPC API is only supported for 64-bit processes on Linux and for devices of compute capability 2.0 and higher.

Using this API, an application can get the IPC handle for a given device memory pointer using `cudaIpcGetMemHandle()`, pass it to another process using standard IPC mechanisms (e.g., interprocess shared memory or files), and use `cudaIpcOpenMemHandle()` to retrieve a device pointer from the IPC handle that is a valid pointer within this other process. Event handles can be shared using similar entry points.

An example of using the IPC API is where a single master process generates a batch of input data, making the data available to multiple slave processes without requiring regeneration or copying.

3.2.9. Error Checking

All runtime functions return an error code, but for an asynchronous function (see Asynchronous Concurrent Execution), this error code cannot possibly report any of the asynchronous errors that could occur on the device since the function returns before the

device has completed the task; the error code only reports errors that occur on the host prior to executing the task, typically related to parameter validation; if an asynchronous error occurs, it will be reported by some subsequent unrelated runtime function call.

The only way to check for asynchronous errors just after some asynchronous function call is therefore to synchronize just after the call by calling **cudaDeviceSynchronize()** (or by using any other synchronization mechanisms described in Asynchronous Concurrent Execution) and checking the error code returned by **cudaDeviceSynchronize()**.

The runtime maintains an error variable for each host thread that is initialized to **cudaSuccess** and is overwritten by the error code every time an error occurs (be it a parameter validation error or an asynchronous error). **cudaPeekAtLastError()** returns this variable. **cudaGetLastError()** returns this variable and resets it to **cudaSuccess**.

Kernel launches do not return any error code, so **cudaPeekAtLastError()** or **cudaGetLastError()** must be called just after the kernel launch to retrieve any pre-launch errors. To ensure that any error returned by **cudaPeekAtLastError()** or **cudaGetLastError()** does not originate from calls prior to the kernel launch, one has to make sure that the runtime error variable is set to cudaSuccess just before the kernel launch, for example, by calling **cudaGetLastError()** just before the kernel launch. Kernel launches are asynchronous, so to check for asynchronous errors, the application must synchronize in-between the kernel launch and the call to **cudaPeekAtLastError()** or **cudaGetLastError()**.

Note that **cudaErrorNotReady** that may be returned by **cudaStreamQuery()** and **cudaEventQuery()** is not considered an error and is therefore not reported by **cudaPeekAtLastError()** or **cudaGetLastError()**.

3.2.10. Call Stack

On devices of compute capability 2.x and higher, the size of the call stack can be queried using **cudaDeviceGetLimit()** and set using **cudaDeviceSetLimit()**.

When the call stack overflows, the kernel call fails with a stack overflow error if the application is run via a CUDA debugger (cuda-gdb, Nsight) or an unspecified launch error, otherwise.

3.2.11. Texture and Surface Memory

CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access texture and surface memory. Reading data from texture or surface memory instead of global memory can have several performance benefits as described in Device Memory Accesses.

There are two different APIs to access texture and surface memory:

- The texture reference API that is supported on all devices,
- The texture object API that is only supported on devices of compute capability 3.x.

The texture reference API has limitations that the texture object API does not have. They are mentioned in Texture Reference API.

3.2.11.1. Texture Memory

Texture memory is read from kernels using the device functions described in Texture Functions. The process of reading a texture calling one of these functions is called a *texture fetch*. Each texture fetch specifies a parameter called a *texture object* for the texture object API or a *texture reference* for the texture reference API.

The texture object or the texture reference specifies:

- The *texture*, which is the piece of texture memory that is fetched. Texture objects are created at runtime and the texture is specified when creating the texture object as described in Texture Object API. Texture references are created at compile time and the texture is specified at runtime by bounding the texture reference to the texture through runtime functions as described in Texture Reference API; several distinct texture references might be bound to the same texture or to textures that overlap in memory. A texture can be any region of linear memory or a CUDA array (described in CUDA Arrays).
- Its *dimensionality* that specifies whether the texture is addressed as a one dimensional array using one texture coordinate, a two-dimensional array using two texture coordinates, or a three-dimensional array using three texture coordinates. Elements of the array are called *texels*, short for *texture elements*. The *texture width*, *height*, and *depth* refer to the size of the array in each dimension. Table 12 lists the maximum texture width, height, and depth depending on the compute capability of the device.
- The type of a texel, which is restricted to the basic integer and single-precision floating-point types and any of the 1-, 2-, and 4-component vector types defined in char, short, int, long, longlong, float, double that are derived from the basic integer and single-precision floating-point types.
- The *read mode*, which is equal to **`cudaReadModeNormalizedFloat`** or **`cudaReadModeElementType`**. If it is **`cudaReadModeNormalizedFloat`** and the type of the texel is a 16-bit or 8-bit integer type, the value returned by the texture fetch is actually returned as floating-point type and the full range of the integer type is mapped to [0.0, 1.0] for unsigned integer type and [-1.0, 1.0] for signed integer type; for example, an unsigned 8-bit texture element with the value 0xff reads as 1. If it is **`cudaReadModeElementType`**, no conversion is performed.
- Whether texture coordinates are normalized or not. By default, textures are referenced (by the functions of Texture Functions) using floating-point coordinates in the range [0, N-1] where N is the size of the texture in the dimension corresponding to the coordinate. For example, a texture that is 64x32 in size will be referenced with coordinates in the range [0, 63] and [0, 31] for the x and y dimensions, respectively. Normalized texture coordinates cause the coordinates to be specified in the range [0.0, 1.0-1/N] instead of [0, N-1], so the same 64x32 texture would be addressed by normalized coordinates in the range [0, 1-1/N] in both the x and y dimensions. Normalized texture coordinates are a natural fit to some applications' requirements, if it is preferable for the texture coordinates to be independent of the texture size.
- The *addressing mode*. It is valid to call the device functions of Section B.8 with coordinates that are out of range. The addressing mode defines what happens

in that case. The default addressing mode is to clamp the coordinates to the valid range: [0, N) for non-normalized coordinates and [0.0, 1.0) for normalized coordinates. If the border mode is specified instead, texture fetches with out-of-range texture coordinates return zero. For normalized coordinates, the warp mode and the mirror mode are also available. When using the wrap mode, each coordinate x is converted to *frac(x)=x floor(x)* where *floor(x)* is the largest integer not greater than *x*. When using the mirror mode, each coordinate *x* is converted to *frac(x)* if *floor(x)* is even and *1-frac(x)* if *floor(x)* is odd. The addressing mode is specified as an array of size three whose first, second, and third elements specify the addressing mode for the first, second, and third texture coordinates, respectively; the addressing mode are **`cudaAddressModeBorder`**, **`cudaAddressModeClamp`**, **`cudaAddressModeWrap`**, and **`cudaAddressModeMirror`**; **`cudaAddressModeWrap`** and **`cudaAddressModeMirror`** are only supported for normalized texture coordinates

- The *filtering* mode which specifies how the value returned when fetching the texture is computed based on the input texture coordinates. Linear texture filtering may be done only for textures that are configured to return floating-point data. It performs low-precision interpolation between neighboring texels. When enabled, the texels surrounding a texture fetch location are read and the return value of the texture fetch is interpolated based on where the texture coordinates fell between the texels. Simple linear interpolation is performed for one-dimensional textures, bilinear interpolation for two-dimensional textures, and trilinear interpolation for three-dimensional textures. Texture Fetching gives more details on texture fetching. The filtering mode is equal to **`cudaFilterModePoint`** or **`cudaFilterModeLinear`**. If it is **`cudaFilterModePoint`**, the returned value is the texel whose texture coordinates are the closest to the input texture coordinates. If it is **`cudaFilterModeLinear`**, the returned value is the linear interpolation of the two (for a one-dimensional texture), four (for a two dimensional texture), or eight (for a three dimensional texture) texels whose texture coordinates are the closest to the input texture coordinates. **`cudaFilterModeLinear`** is only valid for returned values of floating-point type.

Texture Object API introduces the texture object API.

Texture Reference API introduces the texture reference API.

16-Bit Floating-Point Textures explains how to deal with 16-bit floating-point textures.

Textures can also be layered as described in Layered Textures.

Cubemap Textures and Cubemap Layered Textures describe a special type of texture, the cubemap texture.

Texture Gather describes a special texture fetch, texture gather.

3.2.11.1.1. Texture Object API

A texture object is created using **`cudaCreateTextureObject()`** from a resource description of type struct **`cudaResourceDesc`**, which specifies the texture, and from a texture description defined as such:

```
struct cudaTextureDesc
{
    enum cudaTextureAddressMode addressMode[3];
    enum cudaTextureFilterMode  filterMode;
    enum cudaTextureReadMode    readMode;
    int                         sRGB;
    int                         normalizedCoords;
    unsigned int                maxAnisotropy;
    enum cudaTextureFilterMode  mipmapFilterMode;
    float                       mipmapLevelBias;
    float                       minMipmapLevelClamp;
    float                       maxMipmapLevelClamp;
};
```

- **`addressMode`** specifies the addressing mode;
- **`filterMode`** specifies the filter mode;
- **`readMode`** specifies the read mode;
- **`normalizedCoords`** specifies whether texture coordinates are normalized or not;
- See reference manual for **`sRGB`**, **`maxAnisotropy`**, **`mipmapFilterMode`**, **`mipmapLevelBias`**, **`minMipmapLevelClamp`**, and **`maxMipmapLevelClamp`**.

The following code sample applies some simple transformation kernel to a texture.

```
// Simple transformation kernel
__global__ void transformKernel(float* output,
                                cudaTextureObject_t texObj,
                                int width, int height,
                                float theta)
{
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = x / (float)width;
    float v = y / (float)height;

    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;

    // Read from texture and write to global memory
    output[y * width + x] = tex2D<float>(texObj, tu, tv);
}

// Host code
int main()
{
    // Allocate CUDA array in device memory
    cudaChannelFormatDesc channelDesc =
              cudaCreateChannelDesc(32, 0, 0, 0,
                                    cudaChannelFormatKindFloat);
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);

    // Specify texture
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;
    resDesc.res.array.array = cuArray;

    // Specify texture object parameters
    struct cudaTextureDesc texDesc;
    memset(&texDesc, 0, sizeof(texDesc));
    texDesc.addressMode[0]   = cudaAddressModeWrap;
    texDesc.addressMode[1]   = cudaAddressModeWrap;
    texDesc.filterMode       = cudaFilterModeLinear;
    texDesc.readMode         = cudaReadModeElementType;
    texDesc.normalizedCoords = 1;

    // Create texture object
    cudaTextureObject_t texObj = 0;
    cudaCreateTextureObject(&texObj, &resDesc, &texDesc, NULL);

    // Allocate result of transformation in device memory
    float* output;
    cudaMalloc(&output, width * height * sizeof(float));

    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);
    transformKernel<<<dimGrid, dimBlock>>>(output,
                                           texObj, width, height,
                                           angle);

    // Destroy texture object
    cudaDestroyTextureObject(texObj);

    // Free device memory
    cudaFreeArray(cuArray);
```

3.2.11.1.2. Texture Reference API

Some of the attributes of a texture reference are immutable and must be known at compile time; they are specified when declaring the texture reference. A texture reference is declared at file scope as a variable of type **texture**:

```
texture<DataType, Type, ReadMode> texRef;
```

where:

- **DataType** specifies the type of the texel;
- **Type** specifies the type of the texture reference and is equal to **cudaTextureType1D**, **cudaTextureType2D**, or **cudaTextureType3D**, for a one-dimensional, two-dimensional, or three-dimensional texture, respectively, or **cudaTextureType1DLayered** or **cudaTextureType2DLayered** for a one-dimensional or two-dimensional layered texture respectively; Type is an optional argument which defaults to **cudaTextureType1D**;
- **ReadMode** specifies the read mode; it is an optional argument which defaults to **cudaReadModeElementType**.

A texture reference can only be declared as a static global variable and cannot be passed as an argument to a function.

The other attributes of a texture reference are mutable and can be changed at runtime through the host runtime. As explained in the reference manual, the runtime API has a *low-level* C-style interface and a *high-level* C++-style interface. The **texture** type is defined in the high-level API as a structure publicly derived from the **textureReference** type defined in the low-level API as such:

```
struct textureReference {
    int                          normalized;
    enum cudaTextureFilterMode   filterMode;
    enum cudaTextureAddressMode  addressMode[3];
    struct cudaChannelFormatDesc channelDesc;
    int                          sRGB;
    unsigned int                 maxAnisotropy;
    enum cudaTextureFilterMode   mipmapFilterMode;
    float                        mipmapLevelBias;
    float                        minMipmapLevelClamp;
    float                        maxMipmapLevelClamp;
}
```

- **normalized** specifies whether texture coordinates are normalized or not;
- **filterMode** specifies the filtering mode;
- **addressMode** specifies the addressing mode;
- **channelDesc** describes the format of the texel; it must match the **DataType** argument of the texture reference declaration; **channelDesc** is of the following type:

  ```
  struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
  };
  ```

  where x, y, z, and w are equal to the number of bits of each component of the returned value and f is:

- **cudaChannelFormatKindSigned** if these components are of signed integer type,
- **cudaChannelFormatKindUnsigned** if they are of unsigned integer type,
- **cudaChannelFormatKindFloat** if they are of floating point type.

- See reference manual for **sRGB**, **maxAnisotropy**, **mipmapFilterMode**, **mipmapLevelBias**, **minMipmapLevelClamp**, and **maxMipmapLevelClamp**.

normalized, **addressMode**, and **filterMode** may be directly modified in host code.

Before a kernel can use a texture reference to read from texture memory, the texture reference must be bound to a texture using **cudaBindTexture()** or **cudaBindTexture2D()** for linear memory, or **cudaBindTextureToArray()** for CUDA arrays. **cudaUnbindTexture()** is used to unbind a texture reference. Once a texture reference has been unbound, it can be safely rebound to another array, even if kernels that use the previously bound texture have not completed. It is recommended to allocate two-dimensional textures in linear memory using **cudaMallocPitch()** and use the pitch returned by **cudaMallocPitch()** as input parameter to **cudaBindTexture2D()**.

The following code samples bind a 2D texture reference to linear memory pointed to by **devPtr**:

- Using the low-level API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, &texRef);
cudaChannelFormatDesc channelDesc =
                                cudaCreateChannelDesc<float>();
size_t offset;
cudaBindTexture2D(&offset, texRefPtr, devPtr, &channelDesc,
                  width, height, pitch);
```

- Using the high-level API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc =
                                cudaCreateChannelDesc<float>();
size_t offset;
cudaBindTexture2D(&offset, texRef, devPtr, channelDesc,
                  width, height, pitch);
```

The following code samples bind a 2D texture reference to a CUDA array **cuArray**:

- Using the low-level API:

```
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, &texRef);
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindTextureToArray(texRef, cuArray, &channelDesc);
```

- Using the high-level API:

```
texture<float, cudaTextureType2D,
texture<float, cudaTextureType2D,
        cudaReadModeElementType> texRef;
cudaBindTextureToArray(texRef, cuArray);
```

The format specified when binding a texture to a texture reference must match the parameters specified when declaring the texture reference; otherwise, the results of texture fetches are undefined.

There is a limit to the number of textures that can be bound to a kernel as specified in Table 12.

The following code sample applies some simple transformation kernel to a texture.

```
// 2D float texture
texture<float, cudaTextureType2D, cudaReadModeElementType> texRef;

// Simple transformation kernel
__global__ void transformKernel(float* output,
                                int width, int height,
                                float theta)
{
    // Calculate normalized texture coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;

    float u = x / (float)width;
    float v = y / (float)height;

    // Transform coordinates
    u -= 0.5f;
    v -= 0.5f;
    float tu = u * cosf(theta) - v * sinf(theta) + 0.5f;
    float tv = v * cosf(theta) + u * sinf(theta) + 0.5f;


    // Read from texture and write to global memory
    output[y * width + x] = tex2D(texRef, tu, tv);
}

// Host code
int main()
{
    // Allocate CUDA array in device memory
    cudaChannelFormatDesc channelDesc =
              cudaCreateChannelDesc(32, 0, 0, 0,
                                    cudaChannelFormatKindFloat);
    cudaArray* cuArray;
    cudaMallocArray(&cuArray, &channelDesc, width, height);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);

    // Set texture reference parameters
    texRef.addressMode[0] = cudaAddressModeWrap;
    texRef.addressMode[1] = cudaAddressModeWrap;
    texRef.filterMode     = cudaFilterModeLinear;
    texRef.normalized     = true;

    // Bind the array to the texture reference
    cudaBindTextureToArray(texRef, cuArray, channelDesc);

    // Allocate result of transformation in device memory
    float* output;
    cudaMalloc(&output, width * height * sizeof(float));

    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);
    transformKernel<<<dimGrid, dimBlock>>>(output, width, height,
                                           angle);

    // Free device memory
    cudaFreeArray(cuArray);
    cudaFree(output);

    return 0;
}
```

3.2.11.1.3. 16-Bit Floating-Point Textures

The 16-bit floating-point or *half* format supported by CUDA arrays is the same as the IEEE 754-2008 binary2 format.

CUDA C does not support a matching data type, but provides intrinsic functions to convert to and from the 32-bit floating-point format via the **`unsigned short`** type: **`__float2half_rn(float)`** and **`__half2float(unsigned short)`**. These functions are only supported in device code. Equivalent functions for the host code can be found in the OpenEXR library, for example.

16-bit floating-point components are promoted to 32 bit float during texture fetching before any filtering is performed.

A channel description for the 16-bit floating-point format can be created by calling one of the **`cudaCreateChannelDescHalf*()`** functions.

3.2.11.1.4. Layered Textures

A one-dimensional or two-dimensional layered texture (also known as *texture array* in Direct3D and *array texture* in OpenGL) is a texture made up of a sequence of layers, all of which are regular textures of same dimensionality, size, and data type.

A one-dimensional layered texture is addressed using an integer index and a floating-point texture coordinate; the index denotes a layer within the sequence and the coordinate addresses a texel within that layer. A two-dimensional layered texture is addressed using an integer index and two floating-point texture coordinates; the index denotes a layer within the sequence and the coordinates address a texel within that layer.

A layered texture can only be a CUDA array by calling **`cudaMalloc3DArray()`** with the **`cudaArrayLayered`** flag (and a height of zero for one-dimensional layered texture).

Layered textures are fetched using the device functions described in tex1DLayered(), tex1DLayered(), tex2DLayered(), and tex2DLayered(). Texture filtering (see Texture Fetching) is done only within a layer, not across layers.

Layered textures are only supported on devices of compute capability 2.0 and higher.

3.2.11.1.5. Cubemap Textures

A *cubemap* texture is a special type of two-dimensional layered texture that has six layers representing the faces of a cube:

- The width of a layer is equal to its height.
- The cubemap is addressed using three texture coordinates x, y, and z that are interpreted as a direction vector emanating from the center of the cube and pointing to one face of the cube and a texel within the layer corresponding to that face. More specifically, the face is selected by the coordinate with largest magnitude m and the corresponding layer is addressed using coordinates $(s/m+1)/2$ and $(t/m+1)/2$ where s and t are defined in Table 1.

Table 1 Cubemap Fetch

		face	m	s	t
|x| > |y| and |x| > |z|	x ≥ 0	0	x	-z	-y
	x < 0	1	-x	z	-y
|y| > |x| and |y| > |z|	y ≥ 0	2	y	x	z
	y < 0	3	-y	x	-z
|z| > |x| and |z| > |y|	z ≥ 0	4	z	x	-y
	z < 0	5	-z	-x	-y

A layered texture can only be a CUDA array by calling **`cudaMalloc3DArray()`** with the **`cudaArrayCubemap`** flag.

Cubemap textures are fetched using the device function described in texCubemap() and texCubemap().

Cubemap textures are only supported on devices of compute capability 2.0 and higher.

3.2.11.1.6. Cubemap Layered Textures

A *cubemap layered* texture is a layered texture whose layers are cubemaps of same dimension.

A cubemap layered texture is addressed using an integer index and three floating-point texture coordinates; the index denotes a cubemap within the sequence and the coordinates address a texel within that cubemap.

A layered texture can only be a CUDA array by calling **`cudaMalloc3DArray()`** with the **`cudaArrayLayered`** and **`cudaArrayCubemap`** flags.

Cubemap layered textures are fetched using the device function described in texCubemapLayered() and texCubemapLayered(). Texture filtering (see Texture Fetching) is done only within a layer, not across layers.

Cubemap layered textures are only supported on devices of compute capability 2.0 and higher.

3.2.11.1.7. Texture Gather

Texture gather is a special texture fetch that is available for two-dimensional textures only. It is performed by the **`tex2Dgather()`** function, which has the same parameters as **`tex2D()`**, plus an additional **`comp`** parameter equal to 0, 1, 2, or 3 (see tex2Dgather() and tex2Dgather()). It returns four 32-bit numbers that correspond to the value of the component comp of each of the four texels that would have been used for bilinear filtering during a regular texture fetch. For example, if these texels are of values (253, 20, 31, 255), (250, 25, 29, 254), (249, 16, 37, 253), (251, 22, 30, 250), and **`comp`** is 2, **`tex2Dgather()`** returns (31, 29, 37, 30).

Texture gather is only supported for CUDA arrays created with the **`cudaArrayTextureGather`** flag and of width and height less than the maximum specified in Table 12 for texture gather, which is smaller than for regular texture fetch.

Texture gather is only supported on devices of compute capability 2.0 and higher.

3.2.11.2. Surface Memory

For devices of compute capability 2.0 and higher, a CUDA array (described in Cubemap Surfaces), created with the **`cudaArraySurfaceLoadStore`** flag, can be read and written via a *surface object* or *surface reference* using the functions described in Surface Functions.

Table 12 lists the maximum surface width, height, and depth depending on the compute capability of the device.

3.2.11.2.1. Surface Object API

A surface object is created using **`cudaCreateSurfaceObject()`** from a resource description of type **`struct cudaResourceDesc`**.

The following code sample applies some simple transformation kernel to a texture.

```
// Simple copy kernel
__global__ void copyKernel(cudaSurfaceObject_t inputSurfObj,
                           cudaSurfaceObject_t outputSurfObj,
                           int width, int height)
{
    // Calculate surface coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        uchar4 data;
        // Read from input surface
        surf2Dread(&data,  inputSurfObj, x * 4, y);
        // Write to output surface
        surf2Dwrite(data, outputSurfObj, x * 4, y);
    }
}

// Host code
int main()
{
    // Allocate CUDA arrays in device memory
    cudaChannelFormatDesc channelDesc =
             cudaCreateChannelDesc(8, 8, 8, 8,
                                   cudaChannelFormatKindUnsigned);
    cudaArray* cuInputArray;
    cudaMallocArray(&cuInputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);
    cudaArray* cuOutputArray;
    cudaMallocArray(&cuOutputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);

    // Specify surface
    struct cudaResourceDesc resDesc;
    memset(&resDesc, 0, sizeof(resDesc));
    resDesc.resType = cudaResourceTypeArray;

    // Create the surface objects
    resDesc.res.array.array = cuInputArray;
    cudaSurfaceObject_t inputSurfObj = 0;
    cudaCreateSurfaceObject(&inputSurfObj, &resDesc);
    resDesc.res.array.array = cuOutputArray;
    cudaSurfaceObject_t outputSurfObj = 0;
    cudaCreateSurfaceObject(&outputSurfObj, &resDesc);

    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);
    copyKernel<<<dimGrid, dimBlock>>>(inputSurfObj,
                                      outputSurfObj,
                                      width, height);


    // Destroy surface objects
    cudaDestroySurfaceObject(inputSurfObj);
    cudaDestroySurfaceObject(outputSurfObj);

    // Free device memory
    cudaFreeArray(cuInputArray);
    cudaFreeArray(cuOutputArray);

    return 0;
}
```

3.2.11.2.2. Surface Reference API

A surface reference is declared at file scope as a variable of type surface:

```
surface<void, Type> surfRef;
```

where **Type** specifies the type of the surface reference and is equal to **cudaSurfaceType1D**, **cudaSurfaceType2D**, **cudaSurfaceType3D**, **cudaSurfaceTypeCubemap**, **cudaSurfaceType1DLayered**, **cudaSurfaceType2DLayered**, or **cudaSurfaceTypeCubemapLayered**; **Type** is an optional argument which defaults to cudaSurfaceType1D. A surface reference can only be declared as a static global variable and cannot be passed as an argument to a function.

Before a kernel can use a surface reference to access a CUDA array, the surface reference must be bound to the CUDA array using **cudaBindSurfaceToArray()**.

The following code samples bind a surface reference to a CUDA array **cuArray**:

- Using the low-level API:

```
surface<void, cudaSurfaceType2D> surfRef;
surfaceReference* surfRefPtr;
cudaGetSurfaceReference(&surfRefPtr, "surfRef");
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindSurfaceToArray(surfRef, cuArray, &channelDesc);
```

- Using the high-level API:

```
surface<void, cudaSurfaceType2D> surfRef;
cudaBindSurfaceToArray(surfRef, cuArray);
```

A CUDA array must be read and written using surface functions of matching dimensionality and type and via a surface reference of matching dimensionality; otherwise, the results of reading and writing the CUDA array are undefined.

Unlike texture memory, surface memory uses byte addressing. This means that the x-coordinate used to access a texture element via texture functions needs to be multiplied by the byte size of the element to access the same element via a surface function. For example, the element at texture coordinate x of a one-dimensional floating-point CUDA array bound to a texture reference **texRef** and a surface reference **surfRef** is read using **tex1d(texRef, x)** via **texRef**, but **surf1Dread(surfRef, 4*x)** via **surfRef**. Similarly, the element at texture coordinate x and y of a two-dimensional floating-point CUDA array bound to a texture reference **texRef** and a surface reference **surfRef** is accessed using **tex2d(texRef, x, y)** via **texRef**, but **surf2Dread(surfRef, 4*x, y)** via **surfRef** (the byte offset of the y-coordinate is internally calculated from the underlying line pitch of the CUDA array).

The following code sample applies some simple transformation kernel to a texture.

```
// 2D surfaces
surface<void, 2> inputSurfRef;
surface<void, 2> outputSurfRef;

// Simple copy kernel
__global__ void copyKernel(int width, int height)
{
    // Calculate surface coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        uchar4 data;
        // Read from input surface
        surf2Dread(&data,  inputSurfRef, x * 4, y);
        // Write to output surface
        surf2Dwrite(data, outputSurfRef, x * 4, y);
    }
}

// Host code
int main()
{
    // Allocate CUDA arrays in device memory
    cudaChannelFormatDesc channelDesc =
             cudaCreateChannelDesc(8, 8, 8, 8,
                                   cudaChannelFormatKindUnsigned);
    cudaArray* cuInputArray;
    cudaMallocArray(&cuInputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);
    cudaArray* cuOutputArray;
    cudaMallocArray(&cuOutputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);

    // Bind the arrays to the surface references
    cudaBindSurfaceToArray(inputSurfRef, cuInputArray);
    cudaBindSurfaceToArray(outputSurfRef, cuOutputArray);

    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);
    copyKernel<<<dimGrid, dimBlock>>>(width, height);


    // Free device memory
    cudaFreeArray(cuInputArray);
    cudaFreeArray(cuOutputArray);

    return 0;
}
```

3.2.11.2.3. Cubemap Surfaces

Cubemap surfaces are accessed using**`surfCubemapread()`** and **`surfCubemapwrite()`** (surfCubemapread and surfCubemapwrite) as a two-dimensional layered surface, i.e., using an integer index denoting a face and two floating-point texture coordinates addressing a texel within the layer corresponding to this face. Faces are ordered as indicated in Table 1.

3.2.11.2.4. Cubemap Layered Surfaces

Cubemap layered surfaces are accessed using **`surfCubemapLayeredread()`** and **`surfCubemapLayeredwrite()`** (surfCubemapLayeredread() and surfCubemapLayeredwrite()) as a two-dimensional layered surface, i.e., using an integer index denoting a face of one of the cubemaps and two floating-point texture coordinates addressing a texel within the layer corresponding to this face. Faces are ordered as indicated in Table 1, so index ((2 * 6) + 3), for example, accesses the fourth face of the third cubemap.

3.2.11.3. CUDA Arrays

CUDA arrays are opaque memory layouts optimized for texture fetching. They are one dimensional, two dimensional, or three-dimensional and composed of elements, each of which has 1, 2 or 4 components that may be signed or unsigned 8 , 16 or 32 bit integers, 16 bit floats, or 32 bit floats. CUDA arrays are only accessible by kernels through texture fetching as described in Texture Memory or surface reading and writing as described in Surface Memory .

3.2.11.4. Read/Write Coherency

The texture and surface memory is cached (see Device Memory Accesses) and within the same kernel call, the cache is not kept coherent with respect to global memory writes and surface memory writes, so any texture fetch or surface read to an address that has been written to via a global write or a surface write in the same kernel call returns undefined data. In other words, a thread can safely read some texture or surface memory location only if this memory location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread from the same kernel call.

3.2.12. Graphics Interoperability

Some resources from OpenGL and Direct3D may be mapped into the address space of CUDA, either to enable CUDA to read data written by OpenGL or Direct3D, or to enable CUDA to write data for consumption by OpenGL or Direct3D.

A resource must be registered to CUDA before it can be mapped using the functions mentioned in OpenGL Interoperability and Direct3D Interoperability. These functions return a pointer to a CUDA graphics resource of type **`struct cudaGraphicsResource`**. Registering a resource is potentially high-overhead and therefore typically called only once per resource. A CUDA graphics resource is unregistered using **`cudaGraphicsUnregisterResource()`**.

Once a resource is registered to CUDA, it can be mapped and unmapped as many times as necessary using **`cudaGraphicsMapResources()`** and **`cudaGraphicsUnmapResources()`**. **`cudaGraphicsResourceSetMapFlags()`** can be called to specify usage hints (write-only, read-only) that the CUDA driver can use to optimize resource management.

A mapped resource can be read from or written to by kernels using the device memory address returned by **cudaGraphicsResourceGetMappedPointer()** for buffers and **cudaGraphicsSubResourceGetMappedArray()** for CUDA arrays.

Accessing a resource through OpenGL or Direct3D while it is mapped to CUDA produces undefined results. OpenGL Interoperability and Direct3D Interoperability give specifics for each graphics API and some code samples. SLI Interoperability gives specifics for when the system is in SLI mode.

3.2.12.1. OpenGL Interoperability

Interoperability with OpenGL requires that the CUDA device be specified by **cudaGLSetGLDevice()** before any other runtime calls. Note that **cudaSetDevice()** and **cudaGLSetGLDevice()** are mutually exclusive.

The OpenGL resources that may 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 device pointer and can therefore be read and written by kernels or via **cudaMemcpy()** calls.

A texture or renderbuffer object is registered using **cudaGraphicsGLRegisterImage()**. In CUDA, it appears as a CUDA array. Kernels can read from the array by binding it to a texture or surface reference. They can also write to it via the surface write functions if the resource has been registered with the **cudaGraphicsRegisterFlagsSurfaceLoadStore** flag. The array can also be read and written via **cudaMemcpy2D()** calls. **cudaGraphicsGLRegisterImage()** supports all texture formats with 1, 2, or 4 components and an internal type of float (e.g., **GL_RGBA_FLOAT32**), normalized integer (e.g., **GL_RGBA8, GL_INTENSITY16**), and unnormalized integer (e.g., **GL_RGBA8UI**) (please note that since unnormalized integer formats require OpenGL 3.0, they can only be written by shaders, not the fixed function pipeline).

The OpenGL context whose resources are being shared has to be current to the host thread making any OpenGL interoperability API calls.

The following code sample uses a kernel to dynamically modify a 2D **width** x **height** grid of vertices stored in a vertex buffer object:

```
GLuint positionsVBO;
struct cudaGraphicsResource* positionsVBO_CUDA;

int main()
{
    // Initialize OpenGL and GLUT for device 0
    // and make the OpenGL context current
    ...
    glutDisplayFunc(display);

    // Explicitly set device 0
    cudaGLSetGLDevice(0);

    // Create buffer object and register it with CUDA
    glGenBuffers(1, &positionsVBO);
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    unsigned int size = width * height * 4 * sizeof(float);
    glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    cudaGraphicsGLRegisterBuffer(&positionsVBO_CUDA,
                                 positionsVBO,
                                 cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    glutMainLoop();

    ...
}

void display()
{
    // Map buffer object for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVBO_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVBO_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                          width, height);

    // Unmap buffer object
    cudaGraphicsUnmapResources(1, &positionsVBO_CUDA, 0);

    // Render from buffer object
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindBuffer(GL_ARRAY_BUFFER, positionsVBO);
    glVertexPointer(4, GL_FLOAT, 0, 0);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_POINTS, 0, width * height);
    glDisableClientState(GL_VERTEX_ARRAY);

    // Swap buffers
    glutSwapBuffers();
    glutPostRedisplay();
}

void deleteVBO()
{
    cudaGraphicsUnregisterResource(positionsVBO_CUDA);
    glDeleteBuffers(1, &positionsVBO);
}

__global__ void createVertices(float4* positions, float time,
                               unsigned int width, unsigned int height)
{
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
```

On Windows and for Quadro GPUs, **cudaWGLGetDevice()** can be used to retrieve the CUDA device associated to the handle returned by **wglEnumGpusNV()**. Quadro GPUs offer higher performance OpenGL interoperability than GeForce and Tesla GPUs in a multi-GPU configuration where OpenGL rendering is performed on the Quadro GPU and CUDA computations are performed on other GPUs in the system.

3.2.12.2. Direct3D Interoperability

Direct3D interoperability is supported for Direct3D 9, Direct3D 10, and Direct3D 11.

A CUDA context may interoperate with only one Direct3D device at a time and the CUDA context and Direct3D device must be created on the same GPU. In addition the following considerations must be taken when creating the device: Direct3D 9 devices must be created with **DeviceType** set to **D3DDEVTYPE_HAL** and **BehaviorFlags** with the **D3DCREATE_HARDWARE_VERTEXPROCESSING** flag. Direct3D 10 and Direct3D 11 devices must be created with **DriverType** set to **D3D_DRIVER_TYPE_HARDWARE**.

Interoperability with Direct3D requires that the Direct3D device be specified by **cudaD3D9SetDirect3DDevice()**, **cudaD3D10SetDirect3DDevice()** and **cudaD3D11SetDirect3DDevice()**, before any other runtime calls. **cudaD3D9GetDevice()**, **cudaD3D10GetDevice()**, and **cudaD3D11GetDevice()** can be used to retrieve the CUDA device associated to some adapter.

A set of calls is also available to allow the creation of CUDA contexts with interoperability with Direct3D devices that use NVIDIA SLI in AFR (Alternate Frame Rendering) mode: **cudaD3D[9|10|11]GetDevices()**. A call to **cudaD3D[9|10|11]GetDevices()** can be used to obtain a list of CUDA device handles that can be passed as the (optional) last parameter to **cudaD3D[9|10|11]SetDirect3DDevice()**.

The application has the choice to either create multiple CPU threads, each using a different CUDA context, or a single CPU thread using multiple CUDA context. If using separate CPU threads for each GPU each of the CUDA contexts would be created by the CUDA runtime by calling in a separate CPU thread **cudaD3D[9|10|11]SetDirect3DDevice()** using one of the CUDA device handles returned by **cudaD3D[9|10|11]GetDevices()**.

If using a single CPU thread the CUDA contexts would have to be created using the CUDA driver API context creation functions for interoperability with Direct3D devices that use NVIDIA SLI (**cuD3D[9|10|11]CtxCreateOnDevice()**). The application relies on the interoperability between CUDA driver and runtime APIs (Interoperability between Runtime and Driver APIs), which allows it to call **cuCtxPushCurrent()** and **cuCtxPopCurrent()** to change the CUDA context active at a given time.

The Direct3D resources that may be mapped into the address space of CUDA are Direct3D buffers, textures, and surfaces. These resources are registered using **cudaGraphicsD3D9RegisterResource()**, **cudaGraphicsD3D10RegisterResource()**, and **cudaGraphicsD3D11RegisterResource()**.

The following code sample uses a kernel to dynamically modify a 2D **width** x **height** grid of vertices stored in a vertex buffer object.

3.2.12.2.1. Direct3D 9 Version

```
IDirect3D9* D3D;
IDirect3DDevice9* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
IDirect3DVertexBuffer9* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main()
{
    // Initialize Direct3D
    D3D = Direct3DCreate9(D3D_SDK_VERSION);

    // Get a CUDA-enabled adapter
    unsigned int adapter = 0;
    for (; adapter < g_pD3D->GetAdapterCount(); adapter++) {
        D3DADAPTER_IDENTIFIER9 adapterId;
        g_pD3D->GetAdapterIdentifier(adapter, 0, &adapterId);
        int dev;
        if (cudaD3D9GetDevice(&dev, adapterId.DeviceName)
            == cudaSuccess)
            break;
    }

     // Create device
    ...
    D3D->CreateDevice(adapter, D3DDEVTYPE_HAL, hWnd,
                      D3DCREATE_HARDWARE_VERTEXPROCESSING,
                      &params, &device);

    // Register device with CUDA
    cudaD3D9SetDirect3DDevice(device);

    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    device->CreateVertexBuffer(size, 0, D3DFVF_CUSTOMVERTEX,
                               D3DPOOL_DEFAULT, &positionsVB, 0);
    cudaGraphicsD3D9RegisterResource(&positionsVB_CUDA,
                                     positionsVB,
                                     cudaGraphicsRegisterFlagsNone);
    cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                    cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    while (...) {
        ...
        Render();
        ...
    }
    ...
}

void Render()
{
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVB_CUDA));

    // Execute kernel
    dim3 dimBlock(16, 16, 1);
    dim3 dimGrid(width / dimBlock.x, height / dimBlock.y, 1);
    createVertices<<<dimGrid, dimBlock>>>(positions, time,
                                          width, height);

    // Unmap vertex buffer
    cudaGraphicsUnmapResources(1, &positionsVB_CUDA, 0);
```

3.2.12.2.2. Direct3D 10 Version

```
ID3D10Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D10Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main()
{
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        int dev;
        if (cudaD3D10GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
    }
    factory->Release();

    // Create swap chain and device
    ...
    D3D10CreateDeviceAndSwapChain(adapter,
                                  D3D10_DRIVER_TYPE_HARDWARE, 0,
                                  D3D10_CREATE_DEVICE_DEBUG,
                                  D3D10_SDK_VERSION,
                                  &swapChainDesc, &swapChain,
                                  &device);
    adapter->Release();

    // Register device with CUDA
    cudaD3D10SetDirect3DDevice(device);

    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    D3D10_BUFFER_DESC bufferDesc;
    bufferDesc.Usage          = D3D10_USAGE_DEFAULT;
    bufferDesc.ByteWidth      = size;
    bufferDesc.BindFlags      = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags      = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cudaGraphicsD3D10RegisterResource(&positionsVB_CUDA,
                                      positionsVB,
                                      cudaGraphicsRegisterFlagsNone);

 cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                      cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    while (...) {
        ...
        Render();
        ...
    }
    ...
}

void Render()
{
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
                                         &num_bytes,
                                         positionsVB_CUDA));
```

3.2.12.2.3. Direct3D 11 Version

```
ID3D11Device* device;
struct CUSTOMVERTEX {
    FLOAT x, y, z;
    DWORD color;
};
ID3D11Buffer* positionsVB;
struct cudaGraphicsResource* positionsVB_CUDA;

int main()
{
    // Get a CUDA-enabled adapter
    IDXGIFactory* factory;
    CreateDXGIFactory(__uuidof(IDXGIFactory), (void**)&factory);
    IDXGIAdapter* adapter = 0;
    for (unsigned int i = 0; !adapter; ++i) {
        if (FAILED(factory->EnumAdapters(i, &adapter))
            break;
        int dev;
        if (cudaD3D11GetDevice(&dev, adapter) == cudaSuccess)
            break;
        adapter->Release();
    }
    factory->Release();

    // Create swap chain and device
    ...
    sFnPtr_D3D11CreateDeviceAndSwapChain(adapter,
                                         D3D11_DRIVER_TYPE_HARDWARE,
                                         0,
                                         D3D11_CREATE_DEVICE_DEBUG,
                                         featureLevels, 3,
                                         D3D11_SDK_VERSION,
                                         &swapChainDesc, &swapChain,
                                         &device,
                                         &featureLevel,
                                         &deviceContext);
    adapter->Release();

    // Register device with CUDA
    cudaD3D11SetDirect3DDevice(device);

    // Create vertex buffer and register it with CUDA
    unsigned int size = width * height * sizeof(CUSTOMVERTEX);
    D3D11_BUFFER_DESC bufferDesc;
    bufferDesc.Usage          = D3D11_USAGE_DEFAULT;
    bufferDesc.ByteWidth      = size;
    bufferDesc.BindFlags      = D3D11_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags      = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cudaGraphicsD3D11RegisterResource(&positionsVB_CUDA,
                                      positionsVB,
                                      cudaGraphicsRegisterFlagsNone);
    cudaGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                    cudaGraphicsMapFlagsWriteDiscard);

    // Launch rendering loop
    while (...) {
        ...
        Render();
        ...
    }
    ...
}

void Render()
{
    // Map vertex buffer for writing from CUDA
    float4* positions;
    cudaGraphicsMapResources(1, &positionsVB_CUDA, 0);
    size_t num_bytes;
    cudaGraphicsResourceGetMappedPointer((void**)&positions,
```

3.2.12.3. SLI Interoperability

In a system with multiple GPUs, all CUDA-enabled GPUs are accessible via the CUDA driver and runtime as separate devices. There are however special considerations as described below when the system is in SLI mode.

First, 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.

Second, applications have to create multiple CUDA contexts, one for each GPU in the SLI configuration and deal with the fact that a different GPU is used for rendering by the Direct3D or OpenGL device at every frame. The application can use the **`cudaD3D[9|10|11]GetDevices()`** for Direct3D and **`cudaGLGetDevices()`** for OpenGL set of calls to identify the CUDA device handle(s) for the device(s) that are performing the rendering in the current and next frame. Given this information the application will typically map Direct3D or OpenGL resources to the CUDA context corresponding to the CUDA device returned by **`cudaD3D[9|10|11]GetDevices()`** or **`cudaGLGetDevices()`** when the **`deviceList`** parameter is set to **`CU_D3D10_DEVICE_LIST_CURRENT_FRAME`** or **`cudaGLDeviceListCurrentFrame`**.

See Direct3D Interoperability and OpenGL Interoperability for details on how the CUDA runtime interoperate with Direct3D and OpenGL, respectively.

3.3. Versioning and Compatibility

There are two version numbers that developers should care about when developing a CUDA application: The compute capability that describes the general specifications and features of the compute device (see Compute Capability) and the version of the CUDA driver API that describes the features supported by the driver API and runtime.

The version of the driver API is defined in the driver header file as **`CUDA_VERSION`**. It allows developers to check whether their application requires a newer device driver than the one currently installed. This is important, because the driver API is *backward compatible*, meaning that applications, plug-ins, and libraries (including the C runtime) compiled against a particular version of the driver API will continue to work on subsequent device driver releases as illustrated in Figure 11. The driver API is not *forward compatible*, which means that applications, plug-ins, and libraries (including the C runtime) compiled against a particular version of the driver API will not work on previous versions of the device driver.

It is important to note that mixing and matching versions is not supported; specifically:

- All applications, plug-ins, and libraries on a system must use the same version of the CUDA driver API, since only one version of the CUDA device driver can be installed on a system.
- All plug-ins and libraries used by an application must use the same version of the runtime.
- All plug-ins and libraries used by an application must use the same version of any libraries that use the runtime (such as cuFFT, cuBLAS, ...).

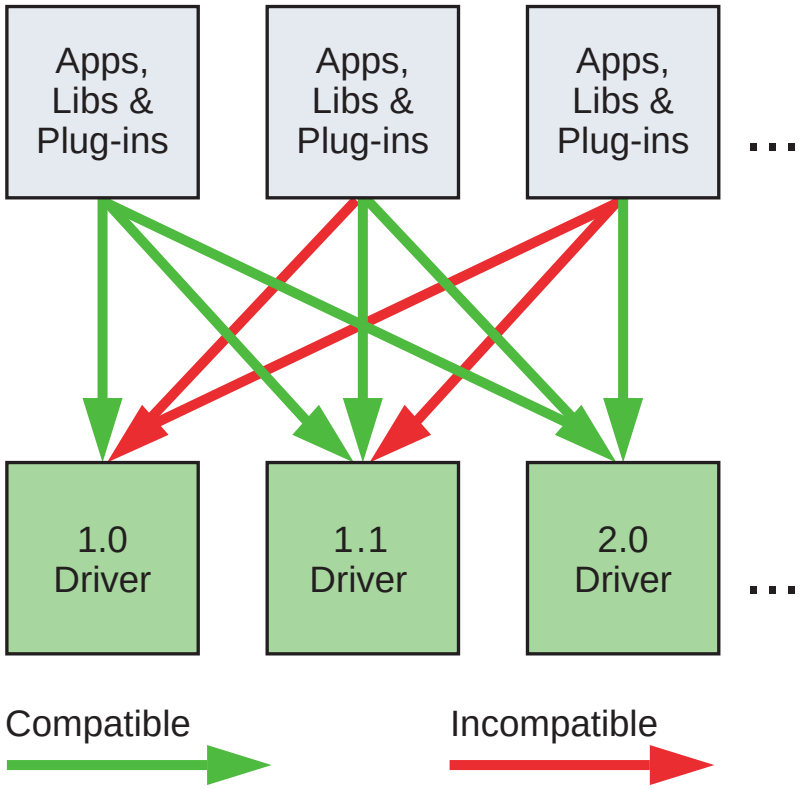


Figure 11 The Driver API Is Backward, but Not Forward Compatible

3.4. Compute Modes

On Tesla solutions running Windows Server 2008 and later or Linux, one can set any device in a system in one of the three following modes using NVIDIA's System Management Interface (nvidia-smi), which is a tool distributed as part of the driver:

- *Default* compute mode: Multiple host threads can use the device (by calling **`cudaSetDevice()`** on this device, when using the runtime API, or by making current a context associated to the device, when using the driver API) at the same time.
- *Exclusive-process* compute mode: Only one CUDA context may be created on the device across all processes in the system and that context may be current to as many threads as desired within the process that created that context.
- *Exclusive-process-and-thread* compute mode: Only one CUDA context may be created on the device across all processes in the system and that context may only be current to one thread at a time.
- *Prohibited* compute mode: No CUDA context can be created on the device.

This means, in particular, that a host thread using the runtime API without explicitly calling **`cudaSetDevice()`** might be associated with a device other than device 0 if device 0 turns out to be in the exclusive-process mode and used by another process, or in the exclusive-process-and-thread mode and used by another thread, or in prohibited

mode. **`cudaSetValidDevices()`** can be used to set a device from a prioritized list of devices.

Applications may query the compute mode of a device by checking the **`computeMode`** device property (see Device Enumeration).

3.5. Mode Switches

GPUs that have a display output dedicate some DRAM memory to the so-called *primary surface*, which is used to refresh the display device whose output is viewed by the user. When users initiate a *mode switch* of the display by changing the resolution or bit depth of the display (using NVIDIA control panel or the Display control panel on Windows), the amount of memory needed for the primary surface changes. For example, if the user changes the display resolution from 1280x1024x32-bit to 1600x1200x32-bit, the system must dedicate 7.68 MB to the primary surface rather than 5.24 MB. (Full-screen graphics applications running with anti-aliasing enabled may require much more display memory for the primary surface.) On Windows, other events that may initiate display mode switches include launching a full-screen DirectX application, hitting Alt +Tab to task switch away from a full-screen DirectX application, or hitting Ctrl+Alt+Del to lock the computer.

If a mode switch increases the amount of memory needed for the primary surface, the system may have to cannibalize memory allocations dedicated to CUDA applications. Therefore, a mode switch results in any call to the CUDA runtime to fail and return an invalid context error.

3.6. Tesla Compute Cluster Mode for Windows

Using NVIDIA's System Management Interface (*nvidia-smi*), the Windows device driver can be put in TCC (Tesla Compute Cluster) mode for devices of the Tesla and Quadro Series of compute capability 2.0 and higher.

This mode has the following primary benefits:

- It makes it possible to use these GPUs in cluster nodes with non-NVIDIA integrated graphics;
- It makes these GPUs available via Remote Desktop, both directly and via cluster management systems that rely on Remote Desktop;
- It makes these GPUs available to applications running as a Windows service (i.e., in Session 0).

However, the TCC mode removes support for any graphics functionality.

Chapter 4. HARDWARE IMPLEMENTATION

The NVIDIA GPU architecture is built around a scalable array of multithreaded *Streaming Multiprocessors* (*SMs*). When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor, and multiple thread blocks can execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called *SIMT* (*Single-Instruction, Multiple-Thread*) that is described in SIMT Architecture. The instructions are pipelined to leverage instruction-level parallelism within a single thread, as well as thread-level parallelism extensively through simultaneous hardware multithreading as detailed in Hardware Multithreading. Unlike CPU cores they are issued in order however and there is no branch prediction and no speculative execution.

SIMT Architecture and Hardware Multithreading describe the architecture features of the streaming multiprocessor that are common to all devices. Compute Capability 1.x, Compute Capability 2.x, Compute Capability 3.x, and Compute Capability 5.0 provide the specifics for devices of compute capabilities 1.x, 2.x, 3.x, and 5.0, respectively.

The NVIDIA GPU architecture uses a little-endian representation.

4.1. SIMT Architecture

The multiprocessor 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.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps and each warp gets scheduled by a *warp scheduler* for execution. 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.

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 serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution 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 in traditional code: 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.

Notes

The threads of a warp that are on that warp's current execution path are called the *active* threads, whereas threads not on the current path are *inactive* (disabled). Threads can be inactive because they have exited earlier than other threads of their warp, or because they are on a different branch path than the branch path currently executed by the warp, or because they are 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 for more than one of the threads of the warp, the number of serialized writes that occur to that location varies depending on the compute capability of the device (see Compute Capability 1.x, Compute Capability 2.x, Compute Capability 3.x, and Compute Capability 5.0), and 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.

4.2. Hardware Multithreading

The execution context (program counters, registers, etc.) for each warp processed by a multiprocessor is maintained on-chip during the entire lifetime of the warp. Therefore, switching from one execution context to another has no cost, and at every instruction issue time, a warp scheduler selects a warp that has threads ready to execute its next instruction (the active threads of the warp) and issues the instruction to those threads.

In particular, each multiprocessor has a set of 32-bit registers that are partitioned among the warps, and a *parallel data cache* or *shared memory* that is partitioned among the thread blocks.

The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the amount of registers and shared memory used by the kernel and the amount of registers and shared memory available on the multiprocessor. There are also a maximum number of resident blocks and a maximum number of resident warps per multiprocessor. These limits as well the amount of registers and shared memory available on the multiprocessor are a function of the compute capability of the device and are given in Appendix F. If there are not enough registers or shared memory available per multiprocessor to process at least one block, the kernel will fail to launch.

The total number of warps in a block is as follows:

$$\text{ceil}\left(\frac{T}{W_{size}}, 1\right)$$

- T is the number of threads per block,
- W_{size} is the warp size, which is equal to 32,
- ceil(x, y) is equal to x rounded up to the nearest multiple of y.

The total number of registers and total amount of shared memory allocated for a block are documented in the CUDA Occupancy Calculator provided in CUDA Software Development Kit.

Chapter 5. PERFORMANCE GUIDELINES

5.1. Overall Performance Optimization Strategies

Performance optimization revolves around three basic strategies:

- Maximize parallel execution to achieve maximum utilization;
- Optimize memory usage to achieve maximum memory throughput;
- Optimize instruction usage to achieve maximum instruction throughput.

Which strategies will yield the best performance gain for a particular portion of an application depends on the performance limiters for that portion; optimizing instruction usage of a kernel that is mostly limited by memory accesses will not yield any significant performance gain, for example. Optimization efforts should therefore be constantly directed by measuring and monitoring the performance limiters, for example using the CUDA profiler. Also, comparing the floating-point operation throughput or memory throughput - whichever makes more sense - of a particular kernel to the corresponding peak theoretical throughput of the device indicates how much room for improvement there is for the kernel.

5.2. Maximize Utilization

To maximize utilization the application should be structured in a way that it exposes as much parallelism as possible and efficiently maps this parallelism to the various components of the system to keep them busy most of the time.

5.2.1. Application Level

At a high level, the application should maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using asynchronous functions calls and streams as described in Asynchronous Concurrent Execution. It should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices.

For the parallel workloads, at points in the algorithm where parallelism is broken because some threads need to synchronize in order to share data with each other, there are two cases: Either these threads belong to the same block, in which case they should use `__syncthreads()` and share data through shared memory within the same kernel invocation, or they belong to different blocks, in which case they must share data through global memory using two separate kernel invocations, one for writing to and one for reading from global memory. The second case is much less optimal since it adds the overhead of extra kernel invocations and global memory traffic. Its occurrence should therefore be minimized by mapping the algorithm to the CUDA programming model in such a way that the computations that require inter-thread communication are performed within a single thread block as much as possible.

5.2.2. Device Level

At a lower level, the application should maximize parallel execution between the multiprocessors of a device.

For devices of compute capability 1.x, only one kernel can execute on a device at one time, so the kernel should be launched with at least as many thread blocks as there are multiprocessors in the device.

For devices of compute capability 2.x and higher, multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using streams to enable enough kernels to execute concurrently as described in Asynchronous Concurrent Execution.

5.2.3. Multiprocessor Level

At an even lower level, the application should maximize parallel execution between the various functional units within a multiprocessor.

As described in Hardware Multithreading, a GPU multiprocessor relies on thread-level parallelism to maximize utilization of its functional units. Utilization is therefore directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute its next instruction, if any, and issues the instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called the *latency*, and full utilization is achieved when all warp schedulers always have some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when latency is completely "hidden". The number of instructions required to hide a latency of L clock cycles depends on the respective throughputs of these instructions (see Arithmetic Instructions for the throughputs of various arithmetic instructions); assuming maximum throughput for all instructions, it is:

- *L/4* (rounded up to nearest integer) for devices of compute capability 1.x since a multiprocessor issues one instruction per warp over four clock cycles, as mentioned in Compute Capability 1.x,
- *L* for devices of compute capability 2.0 since a multiprocessor issues one instruction per warp over two clock cycles for two warps at a time, as mentioned in Compute Capability 2.x,

- *2L* for devices of compute capability 2.1 since a multiprocessor issues a pair of instructions per warp over two clock cycles for two warps at a time, as mentioned in Compute Capability 2.x,
- *8L* for devices of compute capability 3.x since a multiprocessor issues a pair of instructions per warp over one clock cycle for four warps at a time, as mentioned in Compute Capability 3.x.

For devices of compute capability 2.0, the two instructions issued every other cycle are for two different warps. For devices of compute capability 2.1, the four instructions issued every other cycle are two pairs for two different warps, each pair being for the same warp.

For devices of compute capability 3.x, the eight instructions issued every cycle are four pairs for four different warps, each pair being for the same warp.

The most common reason a warp is not ready to execute its next instruction is that the instruction's input operands are not available yet.

If all input operands are registers, latency is caused by register dependencies, i.e., some of the input operands are written by some previous instruction(s) whose execution has not completed yet. In the case of a back-to-back register dependency (i.e., some input operand is written by the previous instruction), the latency is equal to the execution time of the previous instruction and the warp schedulers must schedule instructions for different warps during that time. Execution time varies depending on the instruction, but it is typically about 22 clock cycles for devices of compute capability 1.x and 2.x and about 11 clock cycles for devices of compute capability 3.x, which translates to 6 warps for devices of compute capability 1.x, 22 warps for devices of compute capability 2.x, and 44 warps for devices of compute capability 3.x and higher (still assuming that warps execute instructions with maximum throughput, otherwise fewer warps are needed). For devices of compute capability 2.1 and higher, this is also assuming enough instruction-level parallelism so that schedulers are always able to issue pairs of instructions for each warp.

If some input operand resides in off-chip memory, the latency is much higher: 400 to 800 clock cycles for devices of compute capability 1.x and 2.x and about 200 to 400 clock cycles for devices of compute capability 3.x. The number of warps required to keep the warp schedulers busy during such high latency periods depends on the kernel code and its degree of instruction-level parallelism. In general, more warps are required if the ratio of the number of instructions with no off-chip memory operands (i.e., arithmetic instructions most of the time) to the number of instructions with off-chip memory operands is low (this ratio is commonly called the arithmetic intensity of the program). For example, assume this ratio is 30, also assume the latencies are 600 cycles on devices of compute capability 1.x and 2.x and 300 cycles on devices of compute capability 3.x. Then about 5 warps are required for devices of compute capability 1.x, about 20 for devices of compute capability 2.x and about 40 for devices of compute capability 3.x (with the same assumptions as in the previous paragraph).

Another reason a warp is not ready to execute its next instruction is that it is waiting at some memory fence (Memory Fence Functions) or synchronization point (Memory Fence Functions). A synchronization point can force the multiprocessor to idle as more and more warps wait for other warps in the same block to complete execution of

instructions prior to the synchronization point. Having multiple resident blocks per multiprocessor can help reduce idling in this case, as warps from different blocks do not need to wait for each other at synchronization points.

The number of blocks and warps residing on each multiprocessor for a given kernel call depends on the execution configuration of the call (Execution Configuration), the memory resources of the multiprocessor, and the resource requirements of the kernel as described in Hardware Multithreading. To assist programmers in choosing thread block size based on register and shared memory requirements, the CUDA Software Development Kit provides a spreadsheet, called the CUDA Occupancy Calculator, where occupancy is defined as the ratio of the number of resident warps to the maximum number of resident warps (given in Compute Capabilities for various compute capabilities).

Register, local, shared, and constant memory usages are reported by the compiler when compiling with the **`-ptxas-options=-v`** option.

The total amount of shared memory required for a block is equal to the sum of the amount of statically allocated shared memory, the amount of dynamically allocated shared memory, and for devices of compute capability 1.x, the amount of shared memory used to pass the kernel's arguments (see __noinline__ and __forceinline__).

The number of registers used by a kernel can have a significant impact on the number of resident warps. For example, for devices of compute capability 1.2, if a kernel uses 16 registers and each block has 512 threads and requires very little shared memory, then two blocks (i.e., 32 warps) can reside on the multiprocessor since they require 2x512x16 registers, which exactly matches the number of registers available on the multiprocessor. But as soon as the kernel uses one more register, only one block (i.e., 16 warps) can be resident since two blocks would require 2x512x17 registers, which are more registers than are available on the multiprocessor. Therefore, the compiler attempts to minimize register usage while keeping register spilling (see Device Memory Accesses) and the number of instructions to a minimum. Register usage can be controlled using the **`maxrregcount`** compiler option or launch bounds as described in Launch Bounds.

Each **`double`** variable (on devices that supports native double precision, i.e., devices of compute capability 1.2 and higher) and each long long variable uses two registers. However, devices of compute capability 1.2 and higher have at least twice as many registers per multiprocessor as devices with lower compute capability.

The effect of execution configuration on performance for a given kernel call generally depends on the kernel code. Experimentation is therefore recommended. Applications can also parameterize execution configurations based on register file size and shared memory size, which depends on the compute capability of the device, as well as on the number of multiprocessors and memory bandwidth of the device, all of which can be queried using the runtime (see reference manual).

The number of threads per block should be chosen as a multiple of the warp size to avoid wasting computing resources with under-populated warps as much as possible.

5.3. Maximize Memory Throughput

The first step in maximizing overall memory throughput for the application is to minimize data transfers with low bandwidth.

That means minimizing data transfers between the host and the device, as detailed in Data Transfer between Host and Device, since these have much lower bandwidth than data transfers between global memory and the device.

That also means minimizing data transfers between global memory and the device by maximizing use of on-chip memory: shared memory and caches (i.e., L1 cache available on devices of compute capability 2.x and 3.x, L2 cache available on devices of compute capability 2.x and higher, texture cache and constant cache available on all devices).

Shared memory is equivalent to a user-managed cache: The application explicitly allocates and accesses it. As illustrated in CUDA C Runtime, a typical programming pattern is to stage data coming from device memory into shared memory; in other words, to have each thread of a block:

- Load data from device memory to shared memory,
- Synchronize with all the other threads of the block so that each thread can safely read shared memory locations that were populated by different threads,
- Process the data in shared memory,
- Synchronize again if necessary to make sure that shared memory has been updated with the results,
- Write the results back to device memory.

For some applications (e.g., for which global memory access patterns are data-dependent), a traditional hardware-managed cache is more appropriate to exploit data locality. As mentioned in Compute Capability 2.x and Compute Capability 3.x, for devices of compute capability 2.x and 3.x, the same on-chip memory is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call.

The throughput of memory accesses by a kernel can vary by an order of magnitude depending on access pattern for each type of memory. The next step in maximizing memory throughput is therefore to organize memory accesses as optimally as possible based on the optimal memory access patterns described in Device Memory Accesses. This optimization is especially important for global memory accesses as global memory bandwidth is low, so non-optimal global memory accesses have a higher impact on performance.

5.3.1. Data Transfer between Host and Device

Applications should strive to minimize data transfer between the host and the device. One way to accomplish this is to move more code from the host to the device, even if that means running kernels with low parallelism computations. Intermediate data structures may be created in device memory, operated on by the device, and destroyed without ever being mapped by the host or copied to host memory.

Also, because of the overhead associated with each transfer, batching many small transfers into a single large transfer always performs better than making each transfer separately.

On systems with a front-side bus, higher performance for data transfers between host and device is achieved by using page-locked host memory as described in Page-Locked Host Memory.

In addition, when using mapped page-locked memory (Mapped Memory), there is no need to allocate any device memory and explicitly copy data between device and host memory. Data transfers are implicitly performed each time the kernel accesses the mapped memory. For maximum performance, these memory accesses must be coalesced as with accesses to global memory (see Device Memory Accesses). Assuming that they are and that the mapped memory is read or written only once, using mapped page-locked memory instead of explicit copies between device and host memory can be a win for performance.

On integrated systems where device memory and host memory are physically the same, any copy between host and device memory is superfluous and mapped page-locked memory should be used instead. Applications may query a device is `integrated` by checking that the integrated device property (see Device Enumeration) is equal to 1.

5.3.2. Device Memory Accesses

An instruction that accesses addressable memory (i.e., global, local, shared, constant, or texture memory) might need to be re-issued multiple times depending on the distribution of the memory addresses across the threads within the warp. How the distribution affects the instruction throughput this way is specific to each type of memory and described in the following sections. For example, for global memory, as a general rule, the more scattered the addresses are, the more reduced the throughput is.

Global Memory

Global memory resides in device memory and device memory is accessed via 32-, 64-, or 128-byte memory transactions. These memory transactions must be naturally aligned: Only the 32-, 64-, or 128-byte segments of device memory that are aligned to their size (i.e., whose first address is a multiple of their size) can be read or written by memory transactions.

When a warp executes an instruction that accesses global memory, it coalesces the memory accesses of the threads within the warp into one or more of these memory transactions depending on the size of the word accessed by each thread and the distribution of the memory addresses across the threads. In general, the more transactions are necessary, the more unused words are transferred in addition to the words accessed by the threads, reducing the instruction throughput accordingly. For example, if a 32-byte memory transaction is generated for each thread's 4-byte access, throughput is divided by 8.

How many transactions are necessary and how much throughput is ultimately affected varies with the compute capability of the device. For devices of compute capability 1.0

and 1.1, the requirements on the distribution of the addresses across the threads to get any coalescing at all are very strict. They are much more relaxed for devices of higher compute capabilities. For devices of compute capability 2.x and higher, the memory transactions are cached, so data locality is exploited to reduce impact on throughput. Compute Capability 1.x, Compute Capability 2.x, Compute Capability 3.x, and Compute Capability 5.0 give more details on how global memory accesses are handled for various compute capabilities.

To maximize global memory throughput, it is therefore important to maximize coalescing by:

- Following the most optimal access patterns based on Compute Capability 1.x, Compute Capability 2.x and Compute Capability 3.x,
- Using data types that meet the size and alignment requirement detailed in Device Memory Accesses,
- Padding data in some cases, for example, when accessing a two-dimensional array as described in Device Memory Accesses.

Size and Alignment Requirement

Global memory instructions support reading or writing words of size equal to 1, 2, 4, 8, or 16 bytes. Any access (via a variable or a pointer) to data residing in global memory compiles to a single global memory instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is naturally aligned (i.e., its address is a multiple of that size).

If this size and alignment requirement is not fulfilled, the access compiles to multiple instructions with interleaved access patterns that prevent these instructions from fully coalescing. It is therefore recommended to use types that meet this requirement for data that resides in global memory.

The alignment requirement is automatically fulfilled for the built-in types of char, short, int, long, longlong, float, double like **`float2`** or **`float4`**.

For structures, the size and alignment requirements can be enforced by the compiler using the alignment specifiers **`__align__(8) or __align__(16)`**, such as

```
struct __align__(8) {
    float x;
    float y;
};
```

or

```
struct __align__(16) {
    float x;
    float y;
    float z;
};
```

Any address of a variable residing in global memory or returned by one of the memory allocation routines from the driver or runtime API is always aligned to at least 256 bytes.

Reading non-naturally aligned 8-byte or 16-byte words produces incorrect results (off by a few words), so special care must be taken to maintain alignment of the starting address of any value or array of values of these types. A typical case where this might be easily overlooked is when using some custom global memory allocation scheme, whereby the allocations of multiple arrays (with multiple calls to **cudaMalloc()** or **cuMemAlloc()**) is replaced by the allocation of a single large block of memory partitioned into multiple arrays, in which case the starting address of each array is offset from the block's starting address.

Two-Dimensional Arrays

A common global memory access pattern is when each thread of index (tx,ty) uses the following address to access one element of a 2D array of width width, located at address BaseAddress of type **type*** (where type meets the requirement described in Maximize Utilization):

```
BaseAddress + width * ty + tx
```

For these accesses to be fully coalesced, both the width of the thread block and the width of the array must be a multiple of the warp size (or only half the warp size for devices of compute capability 1.x).

In particular, this means that an array whose width is not a multiple of this size will be accessed much more efficiently if it is actually allocated with a width rounded up to the closest multiple of this size and its rows padded accordingly. The cudaMallocPitch() and cuMemAllocPitch() functions and associated memory copy functions described in the reference manual enable programmers to write non-hardware-dependent code to allocate arrays that conform to these constraints.

Local Memory

Local memory accesses only occur for some automatic variables as mentioned in Variable Type Qualifiers. 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 (this is also known as *register spilling*).

Inspection of the *PTX* assembly code (obtained by compiling with the **-ptx** or **-keep** option) will tell if a variable has been placed in local memory during the first compilation phases as it will be declared using the **.local** mnemonic and accessed using the **ld.local** and **st.local** mnemonics. Even if it has not, subsequent compilation phases might still decide otherwise though if they find it consumes too much register space for the targeted architecture: Inspection of the *cubin* object using **cuobjdump** will tell if this is the case. Also, the compiler reports total local memory usage per kernel (**lmem**) when compiling with the **--ptxas-options=-v** option. Note

that some mathematical functions have implementation paths that might access local memory.

The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing as described in Device Memory Accesses. 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 (e.g., same index in an array variable, same member in a structure variable).

On devices of compute capability 2.x and 3.x, local memory accesses are always cached in L1 and L2 in the same way as global memory accesses (see Compute Capability 2.x and Compute Capability 3.x).

On devices of compute capability 5.0, local memory accesses are always cached in L2 in the same way as global memory accesses (see Compute Capability 5.0).

Shared Memory

Because it is on-chip, shared memory has much higher bandwidth and much lower latency than local or global memory.

To achieve high bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. Any memory read or write request made of n addresses that fall in n distinct memory banks can therefore be serviced simultaneously, yielding an overall bandwidth that is n times as high as the bandwidth of a single module.

However, if two addresses of a memory request fall in the same memory bank, there is a bank conflict and the access has to be serialized. The hardware splits a memory request with bank conflicts into as many separate conflict-free requests as necessary, decreasing throughput by a factor equal to the number of separate memory requests. If the number of separate memory requests is n, the initial memory request is said to cause n-way bank conflicts.

To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts. This is described in Compute Capability 1.x,Compute Capability 2.x, Compute Capability 3.x, and Compute Capability 5.0 for devices of compute capability 1.x, 2.x, 3.x, and 5.0, respectively.

Constant Memory

The constant memory space resides in device memory and is cached in the constant cache mentioned in Compute Capability 1.x and Compute Capability 2.x.

For devices of compute capability 1.x, a constant memory request for a warp is first split into two requests, one for each half-warp, that are issued independently.

A request is then split into as many separate requests as there are different memory addresses in the initial request, decreasing throughput by a factor equal to the number of separate requests.

The resulting requests are then serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise.

Texture and Surface Memory

The texture and surface memory spaces reside in device memory and are cached in texture cache, so a texture fetch or surface read costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache. The texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture or surface addresses that are close together in 2D will achieve best performance. Also, it is designed for streaming fetches with a constant latency; a cache hit reduces DRAM bandwidth demand but not fetch latency.

Reading device memory through texture or surface fetching present some benefits that can make it an advantageous alternative to reading device memory from global or constant memory:

- If the memory reads do not follow the access patterns that global or constant memory reads must follow to get good performance, higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads;
- Addressing calculations are performed outside the kernel by dedicated units;
- Packed data may be broadcast to separate variables in a single operation;
- 8-bit and 16-bit integer input data may be optionally converted to 32 bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0] (see Texture Memory).

5.4. Maximize Instruction Throughput

To maximize instruction throughput the application should:

- Minimize the use of arithmetic instructions with low throughput; this includes trading precision for speed when it does not affect the end result, such as using intrinsic instead of regular functions (intrinsic functions are listed in Intrinsic Functions), single-precision instead of double-precision, or flushing denormalized numbers to zero;
- Minimize divergent warps caused by control flow instructions as detailed in Control Flow Instructions
- Reduce the number of instructions, for example, by optimizing out synchronization points whenever possible as described in Synchronization Instruction or by using restricted pointers as described in __restrict__.

In this section, throughputs are given in number of operations per clock cycle per multiprocessor. For a warp size of 32, one instruction corresponds to 32 operations, so if N is the number of operations per clock cycle, the instruction throughput is N/32 instructions per clock cycle.

All throughputs are for one multiprocessor. They must be multiplied by the number of multiprocessors in the device to get throughput for the whole device.

5.4.1. Arithmetic Instructions

Table 2 gives the throughputs of the arithmetic instructions that are natively supported in hardware for devices of various compute capabilities.

Table 2 Throughput of Native Arithmetic Instructions

(Number of Operations per Clock Cycle per Multiprocessor)

	Compute Capability						
	1.0 1.1 1.2	1.3	2.0	2.1	3.0	3.5	5.0
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192	192	128
64-bit floating-point add, multiply, multiply-add	N/A	1	16[1]	4	8	64[2]	1
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	2	2	4	8	32	32	32
32-bit integer add, extended-precision add, subtract, extended-precision subtract	10	10	32	48	160	160	128
32-bit integer multiply, multiply-add, extended-precision multiply-add	Multiple instructions	Multiple instructions	16	16	32	32	Multiple instructions

[1] 4 for GeForce GPUs
[2] 8 for GeForce GPUs

	Compute Capability						
	1.0 1.1 1.2	1.3	2.0	2.1	3.0	3.5	5.0
24-bit integer multiply (`__[u]mul24`)	8	8	Multiple instructions	Multiple instructions	Multiple instructions	Multiple instructions	Multiple instructions
32-bit integer shift	8	8	16	16	32	64[3]	64
compare, minimum, maximum	10	10	32	48	160	160	64
32-bit integer bit reverse, bit field extract/insert	Multiple instructions	Multiple instructions	16	16	32	32	64
32-bit bitwise AND, OR, XOR	8	8	32	48	160	160	128
count of leading zeros, most significant non-sign bit	Multiple instructions	Multiple instructions	16	16	32	32	Multiple instructions
population count	Multiple instructions	Multiple instructions	16	16	32	32	32
warp shuffle	N/A	N/A	N/A	N/A	32	32	32
sum of absolute difference	Multiple instructions	Multiple instructions	16	16	32	32	64
SIMD video instructions **`vabsdiff2`**	N/A	N/A	N/A	N/A	160	160	N/A
SIMD video instructions **`vabsdiff4`**	N/A	N/A	N/A	N/A	160	160	64
All other SIMD video instructions	N/A	N/A	16	16	32	32	64
Type conversions from 8-bit and 16-bit integer to 32-bit types	8	8	16	16	128	128	32
Type conversions from and to 64-bit types	Multiple instructions	1	16[4]	4	8	32[5]	4
All other type conversions	8	8	16	16	32	32	32

[3] 32 for GeForce GPUs
[4] 4 for GeForce GPUs
[5] 8 for GeForce GPUs

Other instructions and functions are implemented on top of the native instructions. The implementation may be different for devices of different compute capabilities, and the number of native instructions after compilation may fluctuate with every compiler version. For complicated functions, there can be multiple code paths depending on input. **cuobjdump** can be used to inspect a particular implementation in a **cubin** object.

The implementation of some functions are readily available on the CUDA header files (**math_functions.h**, **device_functions.h**, ...).

In general, code compiled with **-ftz=true** (denormalized numbers are flushed to zero) tends to have higher performance than code compiled with **-ftz=false**. Similarly, code compiled with **-prec div=false** (less precise division) tends to have higher performance code than code compiled with **-prec div=true**, and code compiled with **-prec-sqrt=false** (less precise square root) tends to have higher performance than code compiled with **-prec-sqrt=true**. The nvcc user manual describes these compilation flags in more details.

Single-Precision Floating-Point Addition and Multiplication Intrinsics

__fadd_r[d,u], **__fsub_r[d,u]**, **__fmul_r[d,u]**, and **__fmaf_r[n,z,d,u]** (see Intrinsic Functions) compile to tens of instructions for devices of compute capability 1.x, but map to a single native instruction for devices of compute capability 2.x and higher.

Single-Precision Floating-Point Division

__fdividef(x, y) (see Intrinsic Functions) provides faster single-precision floating-point division than the division operator.

Single-Precision Floating-Point Reciprocal Square Root

To preserve IEEE-754 semantics the compiler can optimize **1.0/sqrtf()** into **rsqrtf()** only when both reciprocal and square root are approximate, (i.e., with **-prec-div=false** and **-prec-sqrt=false**). It is therefore recommended to invoke **rsqrtf()** directly where desired.

Single-Precision Floating-Point Square Root

Single-precision floating-point square root is implemented as a reciprocal square root followed by a reciprocal instead of a reciprocal square root followed by a multiplication so that it gives correct results for 0 and infinity.

Sine and Cosine

sinf(x), **cosf(x)**, **tanf(x)**, **sincosf(x)**, and corresponding double-precision instructions are much more expensive and even more so if the argument x is large in magnitude.

More precisely, the argument reduction code (see Mathematical Functions for implementation) comprises two code paths referred to as the fast path and the slow path, respectively.

The fast path is used for arguments sufficiently small in magnitude and essentially consists of a few multiply-add operations. The slow path is used for arguments large in magnitude and consists of lengthy computations required to achieve correct results over the entire argument range.

At present, the argument reduction code for the trigonometric functions selects the fast path for arguments whose magnitude is less than `48039.0f` for the single-precision functions, and less than `2147483648.0` for the double-precision functions.

As the slow path requires more registers than the fast path, an attempt has been made to reduce register pressure in the slow path by storing some intermediate variables in local memory, which may affect performance because of local memory high latency and bandwidth (see Device Memory Accesses). At present, 28 bytes of local memory are used by single-precision functions, and 44 bytes are used by double-precision functions. However, the exact amount is subject to change.

Due to the lengthy computations and use of local memory in the slow path, the throughput of these trigonometric functions is lower by one order of magnitude when the slow path reduction is required as opposed to the fast path reduction.

Integer Arithmetic

On devices of compute capability 1.x, 32-bit integer multiplication is implemented using multiple instructions as it is not natively supported. 24-bit integer multiplication is natively supported however via the `__[u]mul24` intrinsic. Using `__[u]mul24` instead of the 32-bit multiplication operator whenever possible usually improves performance for instruction bound kernels. It can have the opposite effect however in cases where the use of `__[u]mul24` inhibits compiler optimizations.

On devices of compute capability 2.x and beyond, 32-bit integer multiplication is natively supported, but 24-bit integer multiplication is not. `__[u]mul24` is therefore implemented using multiple instructions and should not be used.

Integer division and modulo operation are costly: tens of instructions on devices of compute capability 1.x, below 20 instructions on devices of compute capability 2.x and higher. They can be replaced with bitwise operations in some cases: If `n` is a power of 2, (`i/n`) is equivalent to `(i>>log2(n))` and `(i%n)` is equivalent to (`i&(n-1)`); the compiler will perform these conversions if `n` is literal.

`__brev`, `__brevll`, `__popc`, and `__popcll` compile to tens of instructions for devices of compute capability 1.x, but `__brev` and `__popc` map to a single instruction for devices of compute capability 2.x and higher and `__brevll` and `__popcll` to just a few.

__clz, **__clzll**, **__ffs**, and **__ffsll** compile to fewer instructions for devices of compute capability 2.x and higher than for devices of compute capability 1.x.

Type Conversion

Sometimes, the compiler must insert conversion instructions, introducing additional execution cycles. This is the case for:

- Functions operating on variables of type **char** or **short** whose operands generally need to be converted to **int**,
- Double-precision floating-point constants (i.e., those constants defined without any type suffix) used as input to single-precision floating-point computations (as mandated by C/C++ standards).

This last case can be avoided by using single-precision floating-point constants, defined with an **f** suffix such as **3.141592653589793f**, **1.0f**, **0.5f**.

5.4.2. Control Flow Instructions

Any flow control instruction (**if**, **switch**, **do**, **for**, **while**) can significantly impact the effective instruction throughput by causing threads of the same warp to diverge (i.e., to follow different execution paths). If this happens, the different executions paths have to be serialized, increasing the total number of instructions executed for this warp. When all the different execution paths have completed, the threads converge back to the same execution path.

To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps. This is possible because the distribution of the warps across the block is deterministic as mentioned in SIMT Architecture. A trivial example is when the controlling condition only depends on (**threadIdx / warpSize**) where **warpSize** is the warp size. In this case, no warp diverges since the controlling condition is perfectly aligned with the warps.

Sometimes, the compiler may unroll loops or it may optimize out **if** or **switch** statements by using branch predication instead, as detailed below. In these cases, no warp can ever diverge. The programmer can also control loop unrolling using the **#pragma unroll** directive (see #pragma unroll).

When using branch predication none of the instructions whose execution depends on the controlling condition gets skipped. Instead, each of them is associated with a per-thread condition code or predicate that is set to true or false based on the controlling condition and although each of these instructions gets scheduled for execution, only the instructions with a true predicate are actually executed. Instructions with a false predicate do not write results, and also do not evaluate addresses or read operands.

The compiler replaces a branch instruction with predicated instructions only if the number of instructions controlled by the branch condition is less or equal to a certain threshold: If the compiler determines that the condition is likely to produce many divergent warps, this threshold is 7, otherwise it is 4.

5.4.3. Synchronization Instruction

Throughput for **`__syncthreads()`** is 8 operations per clock cycle for devices of compute capability 1.x, 16 operations per clock cycle for devices of compute capability 2.x, and 128 operations per clock cycle for devices of compute capability 3.x.

Note that **`__syncthreads()`** can impact performance by forcing the multiprocessor to idle as detailed in Device Memory Accesses.

Appendix A.
CUDA-ENABLED GPUS

http://developer.nvidia.com/cuda-gpus lists all CUDA-enabled devices with their compute capability.

The compute capability, number of multiprocessors, clock frequency, total amount of device memory, and other properties can be queried using the runtime (see reference manual).

Appendix B.
C LANGUAGE EXTENSIONS

B.1. Function Type Qualifiers

Function type qualifiers specify whether a function executes on the host or on the device and whether it is callable from the host or from the device.

B.1.1. __device__

The **__device__** qualifier declares a function that is:

- Executed on the device,
- Callable from the device only.

B.1.2. __global__

The **__global__** qualifier declares a function as being a kernel. Such a function is:

- Executed on the device,
- Callable from the host,
- Callable from the device for devices of compute capability 3.x (see CUDA Dynamic Parallelism for more details).

__global__ functions must have void return type.

Any call to a **__global__** function must specify its execution configuration as described in Execution Configuration.

A call to a **__global__** function is asynchronous, meaning it returns before the device has completed its execution.

B.1.3. __host__

The **__host__** qualifier declares a function that is:

- Executed on the host,
- Callable from the host only.

It is equivalent to declare a function with only the **__host__** qualifier or to declare it without any of the **__host__**, **__device__**, or **__global__** qualifier; in either case the function is compiled for the host only.

The **__global__** and **__host__** qualifiers cannot be used together.

The **__device__** and **__host__** qualifiers can be used together however, in which case the function is compiled for both the host and the device. The **__CUDA_ARCH__** macro introduced in Application Compatibility can be used to differentiate code paths between host and device:

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ >= 300
    // Device code path for compute capability 3.x
#elif __CUDA_ARCH__ >= 200
    // Device code path for compute capability 2.x
#elif __CUDA_ARCH__ >= 100
   // Device code path for compute capability 1.x
#elif !defined(__CUDA_ARCH__)
   // Host code path
#endif
}
```

B.1.4. __noinline__ and __forceinline__

When compiling code for devices of compute capability 1.x, a **__device__** function is always inlined by default. When compiling code for devices of compute capability 2.x and higher, a **__device__** function is only inlined when deemed appropriate by the compiler.

The **__noinline__** function qualifier can be used as a hint for the compiler not to inline the function if possible. The function body must still be in the same file where it is called. For devices of compute capability 1.x, the compiler will not honor the **__noinline__** qualifier for functions with pointer parameters and for functions with large parameter lists. For devices of compute capability 2.x and higher, the compiler will always honor the **__noinline__** qualifier.

The **__forceinline__** function qualifier can be used to force the compiler to inline the function.

B.2. Variable Type Qualifiers

Variable type qualifiers specify the memory location on the device of a variable.

An automatic variable declared in device code without any of the **__device__**, **__shared__** and **__constant__** qualifiers described in this section generally resides in a register. However in some cases the compiler might choose to place it in local memory,

which can have adverse performance consequences as detailed in Device Memory Accesses.

B.2.1. __device__

The **__device__** qualifier declares a variable that resides on the device.

At most one of the other type qualifiers defined in the next two sections may be used together with **__device__** to further specify which memory space the variable belongs to. If none of them is present, the variable:

- Resides in global memory space.
- Has the lifetime of an application.
- Is accessible from all the threads within the grid and from the host through the runtime library (**cudaGetSymbolAddress()** / **cudaGetSymbolSize()** / **cudaMemcpyToSymbol()** / **cudaMemcpyFromSymbol()**).
- May be additionally qualified with the **__managed__** qualifier. Such a variable can be directly referenced from host code, e.g., its address can be taken or it can read or written directly from a host function. As a convenience, **__managed__** implies **__managed__ __device__** i.e., the **__device__** qualifier is implicit when the **__managed__** qualifier is specified.

B.2.2. __constant__

The **__constant__** qualifier, optionally used together with **__device__**, declares a variable that:

- Resides in constant memory space,
- Has the lifetime of an application,
- Is accessible from all the threads within the grid and from the host through the runtime library (**cudaGetSymbolAddress()** / **cudaGetSymbolSize()** / **cudaMemcpyToSymbol()** / **cudaMemcpyFromSymbol()**).

B.2.3. __shared__

The __shared__ qualifier, optionally used together with __device__, declares a variable that:

- Resides in the shared memory space of a thread block,
- Has the lifetime of the block,
- Is only accessible from all the threads within the block.

When declaring a variable in shared memory as an external array such as

```
extern __shared__ float shared[];
```

the size of the array is determined at launch time (see Execution Configuration). All variables declared in this fashion, start at the same address in memory, so that the layout

of the variables in the array must be explicitly managed through offsets. For example, if one wants the equivalent of

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

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

```
extern __shared__ float array[];
__device__ void func()      // __device__ or __global__ function
{
    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[];
__device__ void func()      // __device__ or __global__ function
{
    short* array0 = (short*)array;
    float* array1 = (float*)&array0[127];
}
```

Alignment requirements for the built-in vector types are listed in Table 3.

B.2.4. __restrict__

nvcc supports restricted pointers via the **__restrict__** keyword.

Restricted pointers were introduced in C99 to alleviate the aliasing problem that exists in C-type languages, and which inhibits all kind of optimization from code re-ordering to common sub-expression elimination.

Here is an example subject to the aliasing issue, where use of restricted pointer can help the compiler to reduce the number of instructions:

```
void foo(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];
    ...
}
```

In C-type languages, the pointers **a**, **b**, and **c** may be aliased, so any write through **c** could modify elements of **a** or **b**. This means that to guarantee functional correctness, the compiler cannot load **a[0]** and **b[0]** into registers, multiply them, and store the result to both **c[0]** and **c[1]**, because the results would differ from the abstract execution model if, say, **a[0]** is really the same location as **c[0]**. So the compiler cannot take advantage of the common sub-expression. Likewise, the compiler cannot just reorder the

computation of **c[4]** into the proximity of the computation of **c[0]** and **c[1]** because the preceding write to **c[3]** could change the inputs to the computation of **c[4]**.

By making **a**, **b**, and **c** restricted pointers, the programmer asserts to the compiler that the pointers are in fact not aliased, which in this case means writes through **c** would never overwrite elements of **a** or **b**. This changes the function prototype as follows:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c);
```

Note that all pointer arguments need to be made restricted for the compiler optimizer to derive any benefit. With the **__restrict__** keywords added, the compiler can now reorder and do common sub-expression elimination at will, while retaining functionality identical with the abstract execution model:

```
void foo(const float* __restrict__ a,
         const float* __restrict__ b,
         float* __restrict__ c)
{
    float t0 = a[0];
    float t1 = b[0];
    float t2 = t0 * t2;
    float t3 = a[1];
    c[0] = t2;
    c[1] = t2;
    c[4] = t2;
    c[2] = t2 * t3;
    c[3] = t0 * t3;
    c[5] = t1;
    ...
}
```

The effects here are a reduced number of memory accesses and reduced number of computations. This is balanced by an increase in register pressure due to "cached" loads and common sub-expressions.

Since register pressure is a critical issue in many CUDA codes, use of restricted pointers can have negative performance impact on CUDA code, due to reduced occupancy.

B.3. Built-in Vector Types

B.3.1. char, short, int, long, longlong, float, double

These are vector types derived from the basic integer and floating-point types. They are structures and the 1st, 2nd, 3rd, and 4th components are accessible through the fields **x**, **y**, **z**, and **w**, respectively. They all come with a constructor function of the form **make_<type name>**; for example,

```
int2 make_int2(int x, int y);
```

which creates a vector of type **int2** with value **(x, y)**.

In host code, the alignment requirement of a vector type is equal to the alignment requirement of its base type. This is not always the case in device code as detailed in Table 3.

Table 3 Alignment Requirements in Device Code

Type	Alignment
char1, uchar1	1
char2, uchar2	2
char3, uchar3	1
char4, uchar4	4
short1, ushort1	2
short2, ushort2	4
short3, ushort3	2
short4, ushort4	8
int1, uint1	4
int2, uint2	8
int3, uint3	4
int4, uint4	16
long1, ulong1	4 if sizeof(long) is equal to sizeof(int) 8, otherwise
long2, ulong2	8 if sizeof(long) is equal to sizeof(int), 16, otherwise
long3, ulong3	4 if sizeof(long) is equal to sizeof(int), 8, otherwise
long4, ulong4	16
longlong1, ulonglong1	8
longlong2, ulonglong2	16
float1	4
float2	8
float3	4
float4	16
double1	8
double2	16

B.3.2. dim3

This type is an integer vector type based on **uint3** that is used to specify dimensions. When defining a variable of type **dim3**, any component left unspecified is initialized to 1.

B.4. Built-in Variables

Built-in variables specify the grid and block dimensions and the block and thread indices. They are only valid within functions that are executed on the device.

B.4.1. gridDim

This variable is of type **dim3** (see dim3) and contains the dimensions of the grid.

B.4.2. blockIdx

This variable is of type **uint3** (see char, short, int, long, longlong, float, double) and contains the block index within the grid.

B.4.3. blockDim

This variable is of type **dim3** (see dim3) and contains the dimensions of the block.

B.4.4. threadIdx

This variable is of type **uint3** (see char, short, int, long, longlong, float, double) and contains the thread index within the block.

B.4.5. warpSize

This variable is of type **int** and contains the warp size in threads (see SIMT Architecture for the definition of a warp).

B.5. Memory Fence Functions

The CUDA programming model assumes a device with a weakly-ordered memory model, that is:

- 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 the data is observed being written by another CUDA or host thread;
- The order in which a CUDA thread reads data from shared memory, global memory, page-locked host memory, or the memory of a peer device is not necessarily the

order in which the read instructions appear in the program for instructions that are independent of each other.

For example, if thread 0 executes **writeXY()** and thread 1 executes **readXY()** as defined in the following code sample

```
__device__ int X = 1, Y = 2;
__device__ void writeXY()
{
    X = 10;
    Y = 20;
}

__device__ void readXY()
{
    int A = X;
    int B = Y;
}
```

it is possible that **A** ends up equal to 1 and **B** equal to 20 for thread 1:

- either because at the time thread 1 reads **X** and **Y**, thread 0's write to **Y** has happened from thread 1's perspective, but thread 0's write to **X** has not,
- or because thread 1 reads **Y** before **X** and thread 0's writes to **X** and **Y** happen after thread 1's read of **Y** and before thread 1's read of **X**.

In a strongly-ordered memory model, the only possibilities would be:

- **A** equal to 1 and **B** equal to 2 (thread 0's writes to **X** and **Y** happen after thread 1's read of **X** and **Y**),
- **A** equal to 10 and **B** equal to 2 (thread 0's write to **X** happens before thread 1's read of **X** and thread 0's write to **Y** happens after thread 1's read of **Y**),
- **A** equal to 10 and **B** equal to 20 (thread 0's writes to **X** and **Y** happen before thread 1's read of **X** and **Y**),

Memory fence functions can be used to enforce some ordering:

```
void __threadfence_block();
```

ensures that:

- All writes to shared and global memory made by the calling thread before the call to **__threadfence_block()** are observed by all threads in the block of the calling thread as occurring before all writes to shared memory and global memory made by the calling thread after the call to **__threadfence_block()**;
- All reads from shared memory and global memory made by the calling thread before the call to **__threadfence_block()** are performed before all reads from shared memory and global memory made by the calling thread after the call to **__threadfence_block()**.

```
void __threadfence();
```

acts as **__threadfence_block()** for all threads in the block of the calling thread and also ensures that all writes to global memory made by the calling thread before the call to **__threadfence()** are observed by all threads in the device as occurring before all writes to global memory made by the calling thread after the call to **__threadfence()**.

```
void __threadfence_system();
```

acts as **__threadfence_block()** for all threads in the block of the calling thread and also ensures that:

- All writes to global memory, page-locked host memory, and the memory of a peer device 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 global memory, page-locked host memory, and the memory of a peer device made by the calling thread after the call to **__threadfence_system()**.
- All reads from shared memory, global memory, page-locked host memory, and the memory of a peer device made by the calling thread before the call to **__threadfence_system()** are performed before all reads from shared memory, global memory, page-locked host memory, and the memory of a peer device made by the calling thread after the call to **__threadfence_system()**.

__threadfence_system() is only supported by devices of compute capability 2.x and higher.

In the previous code sample, inserting a fence function call between **X = 10;** and **Y = 20;** and between **int A = X;** and **int B = Y;** would ensure that for thread 1, **A** will always be equal to 10 if **B** is equal to 20. If thread 0 and 1 belong to the same block, it is enough to use **__threadfence_block()**. If thread 0 and 1 do not belong to the same block, **__threadfence()** must be used if they are CUDA threads from the same device and **__threadfence_system()** must be used if they are CUDA threads from two different devices.

A common use case is when threads consume some data produced by other threads as illustrated by the following code sample of a kernel that computes the sum of an array of N numbers in one call. Each block first sums a subset of the array and stores the result in global memory. When all blocks are done, the last block done reads each of these partial sums from global memory and sums them to obtain the final result. In order to determine which block is finished last, each block atomically increments a counter to signal that it is done with computing and storing its partial sum (see Atomic Functions about atomic functions). The last block is the one that receives the counter value equal to **gridDim.x-1**. If no fence is placed between storing the partial sum and incrementing the counter, the counter might increment before the partial sum is stored and therefore,

might reach **gridDim.x-1** and let the last block start reading partial sums before they have been actually updated in memory.

```
__device__ unsigned int count = 0;
__shared__ bool isLastBlockDone;
__global__ void sum(const float* array, unsigned int N,
                    float* result)
{
    // 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
        result[blockIdx.x] = partialSum;

        // Thread 0 makes sure that the threads of the
        // last block will read its correct partial sum
        __threadfence();

        // Thread 0 of each block signals that it is done
        unsigned int value = atomicInc(&count, gridDim.x);

        // Thread 0 of each block determines if its block is
        // the last block to be done
        isLastBlockDone = (value == (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);

        if (threadIdx.x == 0) {

            // Thread 0 of last block stores total sum
            // to global memory and resets count so that
            // next kernel call works properly
            result[0] = totalSum;
            count = 0;
        }
    }
}
```

B.6. Synchronization Functions

```
void __syncthreads();
```

waits until all threads in the thread block have reached this point and all global and shared memory accesses made by these threads prior to **__syncthreads()** are visible to all threads in the block.

__syncthreads() is used to coordinate communication between the threads of the same block. When some threads within a block access the same addresses in shared or global memory, there are potential read-after-write, write-after-read, or write-after-

write hazards for some of these memory accesses. These data hazards can be avoided by synchronizing threads in-between these accesses.

__syncthreads() is allowed in conditional code but only if the conditional evaluates identically across the entire thread block, otherwise the code execution is likely to hang or produce unintended side effects.

Devices of compute capability 2.x and higher support three variations of **__syncthreads()** described below.

```
int __syncthreads_count(int predicate);
```

is identical to **__syncthreads()** with the additional feature that it evaluates predicate for all threads of the block and returns the number of threads for which predicate evaluates to non-zero.

```
int __syncthreads_and(int predicate);
```

is identical to **__syncthreads()** with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for all of them.

```
int __syncthreads_or(int predicate);
```

is identical to **__syncthreads()** with the additional feature that it evaluates predicate for all threads of the block and returns non-zero if and only if predicate evaluates to non-zero for any of them.

B.7. Mathematical Functions

The reference manual lists all C/C++ standard library mathematical functions that are supported in device code and all intrinsic functions that are only supported in device code.

Mathematical Functions provides accuracy information for some of these functions when relevant.

B.8. Texture Functions

Texture objects are described in Texture Object API

Texture references are described in Texture Reference API

Texture fetching is described in Texture Fetching.

B.8.1. Texture Object API

B.8.1.1. tex1Dfetch()

```
template<class T>
T tex1Dfetch(cudaTextureObject_t texObj, int x);
```

fetches the region of linear memory specified by the one-dimensional texture object **texObj** using integer texture coordinate x. **tex1Dfetch()** only works with non-normalized coordinates, so only the border and clamp addressing modes are supported. It does not perform any texture filtering. For integer types, it may optionally promote the integer to single-precision floating point.

B.8.1.2. tex1D()

```
template<class T>
T tex1D(cudaTextureObject_t texObj, float x);
template<class T>
T tex1D(cudaTextureObject_t texObj, float x);
```

fetches the CUDA array specified by the one-dimensional texture object **texObj** using texture coordinate x.

B.8.1.3. tex2D()

```
template<class T>
T tex2D(cudaTextureObject_t texObj, float x, float y);
```

fetches the CUDA array or the region of linear memory specified by the two-dimensional texture object **texObj** using texture coordinates x and y.

B.8.1.4. tex3D()

```
template<class T>
T tex3D(cudaTextureObject_t texObj, float x, float y, float z);
```

fetches the CUDA array specified by the three-dimensional texture object **texObj** using texture coordinates x, y, and z.

B.8.1.5. tex1DLayered()

```
template<class T>
T tex1DLayered(cudaTextureObject_t texObj, float x, int layer);
```

fetches the CUDA array specified by the one-dimensional texture object **texObj** using texture coordinate x and index layer, as described in Layered Textures

B.8.1.6. tex2DLayered()

```
template<class T>
T tex2DLayered(cudaTextureObject_t texObj,
               float x, float y, int layer);
```

fetches the CUDA array specified by the two-dimensional texture object **texObj** using texture coordinates x and y, and index layer, as described in Layered Textures.

B.8.1.7. texCubemap()

```
template<class T>
T texCubemap(cudaTextureObject_t texObj, float x, float y, float z);
```

fetches the CUDA array specified by the three-dimensional texture object **texObj** using texture coordinates x, y, and z, as described in Section Cubemap Textures.

B.8.1.8. texCubemapLayered()

```
template<class T>
T texCubemapLayered(cudaTextureObject_t texObj,
                    float x, float y, float z, int layer);
```

fetches the CUDA array specified by the cubemap layered texture object **texObj** using texture coordinates x, y, and z, and index layer, as described in Cubemap Layered Textures.

B.8.1.9. tex2Dgather()

```
template<class T>
T tex2Dgather(cudaTextureObject_t texObj,
              float x, float y, int comp = 0);
```

fetches the CUDA array specified by the 2D texture object **texObj** using texture coordinates **x** and **y** and the **comp** parameter as described in Texture Gather.

B.8.2. Texture Reference API

B.8.2.1. tex1Dfetch()

```
template<class DataType>
Type tex1Dfetch(
   texture<DataType, cudaTextureType1D,
           cudaReadModeElementType> texRef,
   int x);

float tex1Dfetch(
   texture<unsigned char, cudaTextureType1D,
           cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<signed char, cudaTextureType1D,
           cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<unsigned short, cudaTextureType1D,
           cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<signed short, cudaTextureType1D,
           cudaReadModeNormalizedFloat> texRef,
   int x);
```

fetches the region of linear memory bound to the one-dimensional texture reference **texRef** using integer texture coordinate x. **tex1Dfetch()** only works with non-normalized coordinates, so only the border and clamp addressing modes are supported. It does not perform any texture filtering. For integer types, it may optionally promote the integer to single-precision floating point.

Besides the functions shown above, 2-, and 4-tuples are supported; for example:

```
float4 tex1Dfetch(
   texture<uchar4, cudaTextureType1D,
           cudaReadModeNormalizedFloat> texRef,
   int x);
```

fetches the region of linear memory bound to texture reference texRef using texture coordinate x.

B.8.2.2. tex1D()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex1D(texture<DataType, cudaTextureType1D, readMode> texRef,
           float x);
```

fetches the CUDA array bound to the one-dimensional texture reference **`texRef`** using texture coordinate **`x`**. **`Type`** is equal to **`DataType`** except when **`readMode`** is equal to **`cudaReadModeNormalizedFloat`** (see Texture Reference API), in which case **`Type`** is equal to the matching floating-point type.

B.8.2.3. tex2D()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2D(texture<DataType, cudaTextureType2D, readMode> texRef,
           float x, float y);
```

fetches the CUDA array or the region of linear memory bound to the two-dimensional texture reference **`texRef`** using texture coordinates **`x`** and **`y`**. **`Type`** is equal to **`DataType`** except when **`readMode`** is equal to **`cudaReadModeNormalizedFloat`** (see Texture Reference API), in which case **`Type`** is equal to the matching floating-point type.

B.8.2.4. tex3D()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex3D(texture<DataType, cudaTextureType3D, readMode> texRef,
           float x, float y, float z);
```

fetches the CUDA array bound to the three-dimensional texture reference **`texRef`** using texture coordinates **`x`**, **`y`**, and **`z`**. **`Type`** is equal to **`DataType`** except when **`readMode`** is equal to **`cudaReadModeNormalizedFloat`** (see Texture Reference API), in which case **`Type`** is equal to the matching floating-point type.

B.8.2.5. tex1DLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex1DLayered(
     texture<DataType, cudaTextureType1DLayered, readMode> texRef,
     float x, int layer);
```

fetches the CUDA array bound to the one-dimensional layered texture reference **`texRef`** using texture coordinate x and index **`layer`**, as described in Layered Textures. **`Type`** is equal to **`DataType`** except when **`readMode`** is equal to **`cudaReadModeNormalizedFloat`** (see Texture Reference API), in which case **`Type`** is equal to the matching floating-point type.

B.8.2.6. tex2DLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2DLayered(
     texture<DataType, cudaTextureType2DLayered, readMode> texRef,
     float x, float y, int layer);
```

fetches the CUDA array bound to the two-dimensional layered texture reference **texRef** using texture coordinates **x** and **y**, and index **layer**, as described in Texture Memory. **Type** is equal to **DataType** except when **readMode** is equal to **cudaReadModeNormalizedFloat** (see Texture Reference API), in which case **Type** is equal to the matching floating-point type.

B.8.2.7. texCubemap()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type texCubemap(
     texture<DataType, cudaTextureTypeCubemap, readMode> texRef,
     float x, float y, float z);
```

fetches the CUDA array bound to the cubemap texture reference **texRef** using texture coordinates **x**, **y**, and **z**, as described in Cubemap Textures. **Type** is equal to **DataType** except when **readMode** is equal to **cudaReadModeNormalizedFloat** (see Texture Reference API), in which case **Type** is equal to the matching floating-point type.

B.8.2.8. texCubemapLayered()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type texCubemapLayered(
texture<DataType, cudaTextureTypeCubemapLayered, readMode> texRef,
float x, float y, float z, int layer);
```

fetches the CUDA array bound to the cubemap layered texture reference **texRef** using texture coordinates **x**, **y**, and **z**, and index layer, as described in Cubemap Layered Textures. **Type** is equal to **DataType** except when **readMode** is equal to **cudaReadModeNormalizedFloat** (see Texture Reference API), in which case **Type** is equal to the matching floating-point type.

B.8.2.9. tex2Dgather()

```
template<class DataType, enum cudaTextureReadMode readMode>
Type tex2Dgather(
     texture<DataType, cudaTextureType2D, readMode> texRef,
     float x, float y, int comp = 0);
```

fetches the CUDA array bound to the 2D texture reference **texRef** using texture coordinates **x** and **y** and the **comp** parameter as described in Texture Gather. **Type** is a 4-component vector type. It is based on the base type of **DataType** except when **readMode** is equal to **cudaReadModeNormalizedFloat** (see Texture Reference API), in which case it is always **float4**.

B.9. Surface Functions

Surface functions are only supported by devices of compute capability 2.0 and higher.

Surface objects are described in described in Surface Object API

Surface references are described in Surface Reference API.

In the sections below, **boundaryMode** specifies the boundary mode, that is how out-of-range surface coordinates are handled; it is equal to either **cudaBoundaryModeClamp**, in which case out-of-range coordinates are clamped to the valid range, or **cudaBoundaryModeZero**, in which case out-of-range reads return zero and out-of-range writes are ignored, or **cudaBoundaryModeTrap**, in which case out-of-range accesses cause the kernel execution to fail.

B.9.1. Surface Object API

B.9.1.1. surf1Dread()

```
template<class T>
T surf1Dread(cudaSurfaceObject_t surfObj, int x,
               boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the one-dimensional surface object **surfObj** using coordinate x.

B.9.1.2. surf1Dwrite

```
template<class T>
void surf1Dwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the one-dimensional surface object **surfObj** at coordinate x.

B.9.1.3. surf2Dread()

```
template<class T>
T surf2Dread(cudaSurfaceObject_t surfObj,
             int x, int y,
             boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf2Dread(T* data,
                cudaSurfaceObject_t surfObj,
                int x, int y,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the two-dimensional surface object **surfObj** using coordinates x and y.

B.9.1.4. surf2Dwrite()

```
template<class T>
void surf2Dwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the two-dimensional surface object **`surfObj`** at coordinate x and y.

B.9.1.5. surf3Dread()

```
template<class T>
T surf3Dread(cudaSurfaceObject_t surfObj,
             int x, int y, int z,
             boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf3Dread(T* data,
                cudaSurfaceObject_t surfObj,
                int x, int y, int z,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the three-dimensional surface object **`surfObj`** using coordinates x, y, and z.

B.9.1.6. surf3Dwrite()

```
template<class T>
void surf3Dwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int z,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the three-dimensional object **`surfObj`** at coordinate x, y, and z.

B.9.1.7. surf1DLayeredread()

```
template<class T>
T surf1DLayeredread(
                cudaSurfaceObject_t surfObj,
                int x, int layer,
                boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf1DLayeredread(T data,
                cudaSurfaceObject_t surfObj,
                int x, int layer,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the one-dimensional layered surface object **`surfObj`** using coordinate x and index **`layer`**.

B.9.1.8. surf1DLayeredwrite()

```
template<class Type>
void surf1DLayeredwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int layer,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the two-dimensional layered surface object **`surfObj`** at coordinate x and index **`layer`**.

B.9.1.9. surf2DLayeredread()

```
template<class T>
T surf2DLayeredread(
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int layer,
                 boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surf2DLayeredread(T data,
                        cudaSurfaceObject_t surfObj,
                        int x, int y, int layer,
                        boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the two-dimensional layered surface object **`surfObj`** using coordinate x and y, and index **`layer`**.

B.9.1.10. surf2DLayeredwrite()

```
template<class T>
void surf2DLayeredwrite(T data,
                         cudaSurfaceObject_t surfObj,
                         int x, int y, int layer,
                         boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the one-dimensional layered surface object **`surfObj`** at coordinate x and y, and index **`layer`**.

B.9.1.11. surfCubemapread()

```
template<class T>
T surfCubemapread(
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int face,
                 boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surfCubemapread(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int face,
                 boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the cubemap surface object **`surfObj`** using coordinate x and y, and face index face.

B.9.1.12. surfCubemapwrite()

```
template<class T>
void surfCubemapwrite(T data,
                      cudaSurfaceObject_t surfObj,
                      int x, int y, int face,
                      boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the cubemap object **surfObj** at coordinate x and y, and face index face.

B.9.1.13. surfCubemapLayeredread()

```
template<class T>
T surfCubemapLayeredread(
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int layerFace,
                 boundaryMode = cudaBoundaryModeTrap);
template<class T>
void surfCubemapLayeredread(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int layerFace,
                 boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array specified by the cubemap layered surface object **surfObj** using coordinate x and y, and index **layerFace**.

B.9.1.14. surfCubemapLayeredwrite()

```
template<class T>
void surfCubemapLayeredwrite(T data,
                 cudaSurfaceObject_t surfObj,
                 int x, int y, int layerFace,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value data to the CUDA array specified by the cubemap layered object surfObj at coordinate x and y, and index **layerFace**.

B.9.2. Surface Reference API

A surface reference is declared at file scope as a variable of type surface:

```
surface<void, Type> surfRef;
```

where **Type** specifies the type of the surface reference and is equal to **cudaSurfaceType1D**, **cudaSurfaceType2D**, **cudaSurfaceType3D**, **cudaSurfaceTypeCubemap**, **cudaSurfaceType1DLayered**, **cudaSurfaceType2DLayered**, or **cudaSurfaceTypeCubemapLayered**; **Type** is an optional argument which defaults to cudaSurfaceType1D. A surface reference can only be declared as a static global variable and cannot be passed as an argument to a function.

Before a kernel can use a surface reference to access a CUDA array, the surface reference must be bound to the CUDA array using **cudaBindSurfaceToArray()**.

The following code samples bind a surface reference to a CUDA array **cuArray**:

- Using the low-level API:

```
surface<void, cudaSurfaceType2D> surfRef;
surfaceReference* surfRefPtr;
cudaGetSurfaceReference(&surfRefPtr, "surfRef");
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindSurfaceToArray(surfRef, cuArray, &channelDesc);
```

- Using the high-level API:

```
surface<void, cudaSurfaceType2D> surfRef;
cudaBindSurfaceToArray(surfRef, cuArray);
```

A CUDA array must be read and written using surface functions of matching dimensionality and type and via a surface reference of matching dimensionality; otherwise, the results of reading and writing the CUDA array are undefined.

Unlike texture memory, surface memory uses byte addressing. This means that the x-coordinate used to access a texture element via texture functions needs to be multiplied by the byte size of the element to access the same element via a surface function. For example, the element at texture coordinate x of a one-dimensional floating-point CUDA array bound to a texture reference **`texRef`** and a surface reference **`surfRef`** is read using **`tex1d(texRef, x)`** via **`texRef`**, but **`surf1Dread(surfRef, 4*x)`** via **`surfRef`**. Similarly, the element at texture coordinate x and y of a two-dimensional floating-point CUDA array bound to a texture reference **`texRef`** and a surface reference **`surfRef`** is accessed using **`tex2d(texRef, x, y)`** via **`texRef`**, but **`surf2Dread(surfRef, 4*x, y)`** via **`surfRef`** (the byte offset of the y-coordinate is internally calculated from the underlying line pitch of the CUDA array).

The following code sample applies some simple transformation kernel to a texture.

```
// 2D surfaces
surface<void, 2> inputSurfRef;
surface<void, 2> outputSurfRef;

// Simple copy kernel
__global__ void copyKernel(int width, int height)
{
    // Calculate surface coordinates
    unsigned int x = blockIdx.x * blockDim.x + threadIdx.x;
    unsigned int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x < width && y < height) {
        uchar4 data;
        // Read from input surface
        surf2Dread(&data,  inputSurfRef, x * 4, y);
        // Write to output surface
        surf2Dwrite(data, outputSurfRef, x * 4, y);
    }
}

// Host code
int main()
{
    // Allocate CUDA arrays in device memory
    cudaChannelFormatDesc channelDesc =
             cudaCreateChannelDesc(8, 8, 8, 8,
                                   cudaChannelFormatKindUnsigned);
    cudaArray* cuInputArray;
    cudaMallocArray(&cuInputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);
    cudaArray* cuOutputArray;
    cudaMallocArray(&cuOutputArray, &channelDesc, width, height,
                    cudaArraySurfaceLoadStore);

    // Copy to device memory some data located at address h_data
    // in host memory
    cudaMemcpyToArray(cuInputArray, 0, 0, h_data, size,
                      cudaMemcpyHostToDevice);

    // Bind the arrays to the surface references
    cudaBindSurfaceToArray(inputSurfRef, cuInputArray);
    cudaBindSurfaceToArray(outputSurfRef, cuOutputArray);

    // Invoke kernel
    dim3 dimBlock(16, 16);
    dim3 dimGrid((width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);
    copyKernel<<<dimGrid, dimBlock>>>(width, height);


    // Free device memory
    cudaFreeArray(cuInputArray);
    cudaFreeArray(cuOutputArray);

    return 0;
}
```

B.9.2.1. surf1Dread()

```
template<class Type>
Type surf1Dread(surface<void, cudaSurfaceType1D> surfRef,
                int x,
                boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf1Dread(Type data,
                surface<void, cudaSurfaceType1D> surfRef,
                int x,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array bound to the one-dimensional surface reference **`surfRef`** using coordinate **`x`**.

B.9.2.2. surf1Dwrite

```
template<class Type>
void surf1Dwrite(Type data,
                 surface<void, cudaSurfaceType1D> surfRef,
                 int x,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value **`data`** to the CUDA array bound to the one-dimensional surface reference **`surfRef`** at coordinate **`x`**.

B.9.2.3. surf2Dread()

```
template<class Type>
Type surf2Dread(surface<void, cudaSurfaceType2D> surfRef,
                int x, int y,
                boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf2Dread(Type* data,
                surface<void, cudaSurfaceType2D> surfRef,
                int x, int y,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array bound to the two-dimensional surface reference **`surfRef`** using coordinates **`x`** and **`y`**.

B.9.2.4. surf2Dwrite()

```
template<class Type>
void surf3Dwrite(Type data,
                 surface<void, cudaSurfaceType3D> surfRef,
                 int x, int y, int z,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value **`data`** to the CUDA array bound to the two-dimensional surface reference **`surfRef`** at coordinate **`x`** and **`y`**.

B.9.2.5. surf3Dread()

```
template<class Type>
Type surf3Dread(surface<void, cudaSurfaceType3D> surfRef,
                int x, int y, int z,
                boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf3Dread(Type* data,
                surface<void, cudaSurfaceType3D> surfRef,
                int x, int y, int z,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array bound to the three-dimensional surface reference **surfRef** using coordinates **x**, **y**, and **z**.

B.9.2.6. surf3Dwrite()

```
template<class Type>
void surf3Dwrite(Type data,
                 surface<void, cudaSurfaceType3D> surfRef,
                 int x, int y, int z,
                 boundaryMode = cudaBoundaryModeTrap);
```

writes value **data** to the CUDA array bound to the three-dimensional surface reference **surfRef** at coordinate **x**, **y**, and **z**.

B.9.2.7. surf1DLayeredread()

```
template<class Type>
Type surf1DLayeredread(
                surface<void, cudaSurfaceType1DLayered> surfRef,
                int x, int layer,
                boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf1DLayeredread(Type data,
                surface<void, cudaSurfaceType1DLayered> surfRef,
                int x, int layer,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array bound to the one-dimensional layered surface reference **surfRef** using coordinate **x** and index **layer**.

B.9.2.8. surf1DLayeredwrite()

```
template<class Type>
void surf1DLayeredwrite(Type data,
                surface<void, cudaSurfaceType1DLayered> surfRef,
                int x, int layer,
                boundaryMode = cudaBoundaryModeTrap);
```

writes value **data** to the CUDA array bound to the two-dimensional layered surface reference **surfRef** at coordinate **x** and index **layer**.

B.9.2.9. surf2DLayeredread()

```
template<class Type>
Type surf2DLayeredread(
                surface<void, cudaSurfaceType2DLayered> surfRef,
                int x, int y, int layer,
                boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surf2DLayeredread(Type data,
                surface<void, cudaSurfaceType2DLayered> surfRef,
                int x, int y, int layer,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array bound to the two-dimensional layered surface reference **`surfRef`** using coordinate **`x`** and **`y`**, and index **`layer`**.

B.9.2.10. surf2DLayeredwrite()

```
template<class Type>
void surf2DLayeredwrite(Type data,
                surface<void, cudaSurfaceType2DLayered> surfRef,
                int x, int y, int layer,
                boundaryMode = cudaBoundaryModeTrap);
```

writes value **`data`** to the CUDA array bound to the one-dimensional layered surface reference **`surfRef`** at coordinate **`x`** and **`y`**, and index **`layer`**.

B.9.2.11. surfCubemapread()

```
template<class Type>
Type surfCubemapread(
                surface<void, cudaSurfaceTypeCubemap> surfRef,
                int x, int y, int face,
                boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surfCubemapread(Type data,
                surface<void, cudaSurfaceTypeCubemap> surfRef,
                int x, int y, int face,
                boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array bound to the cubemap surface reference **`surfRef`** using coordinate **`x`** and **`y`**, and face index **`face`**.

B.9.2.12. surfCubemapwrite()

```
template<class Type>
void surfCubemapwrite(Type data,
                surface<void, cudaSurfaceTypeCubemap> surfRef,
                int x, int y, int face,
                boundaryMode = cudaBoundaryModeTrap);
```

writes value **`data`** to the CUDA array bound to the cubemap reference **`surfRef`** at coordinate **`x`** and **`y`**, and face index **`face`**.

B.9.2.13. surfCubemapLayeredread()

```
template<class Type>
Type surfCubemapLayeredread(
            surface<void, cudaSurfaceTypeCubemapLayered> surfRef,
            int x, int y, int layerFace,
            boundaryMode = cudaBoundaryModeTrap);
template<class Type>
void surfCubemapLayeredread(Type data,
            surface<void, cudaSurfaceTypeCubemapLayered> surfRef,
            int x, int y, int layerFace,
            boundaryMode = cudaBoundaryModeTrap);
```

reads the CUDA array bound to the cubemap layered surface reference **surfRef** using coordinate **x** and **y**, and index **layerFace**.

B.9.2.14. surfCubemapLayeredwrite()

```
template<class Type>
void surfCubemapLayeredwrite(Type data,
            surface<void, cudaSurfaceTypeCubemapLayered> surfRef,
            int x, int y, int layerFace,
            boundaryMode = cudaBoundaryModeTrap);
```

writes value **data** to the CUDA array bound to the cubemap layered reference **surfRef** at coordinate **x** and **y**, and index **layerFace**.

B.10. Read-Only Data Cache Load Function

The read-only data cache load function is only supported by devices of compute capability 3.5 and higher.

```
T __ldg(const T* address);
```

returns the data of type **T** located at address **address**, where **T** is **char**, **short**, **int**, **long long** **unsigned char**, **unsigned short**, **unsigned int**, **unsigned long long**, **int2**, **int4**, **uint2**, **uint4**, **float**, **float2**, **float4**, **double**, or **double2**. The operation is cached in the read-only data cache (see Global Memory).

B.11. Time Function

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

when executed in device code, returns the value of a per-multiprocessor counter that is incremented every clock cycle. Sampling this counter at the beginning and at the end of a kernel, taking the difference of the two samples, and recording the result per thread provides a measure for each thread of the number of clock cycles taken by the device to completely execute the thread, but not of the number of clock cycles the device actually spent executing thread instructions. The former number is greater that the latter since threads are time sliced.

B.12. Atomic Functions

An atomic function performs a read-modify-write atomic operation on one 32-bit or 64-bit word residing in global or shared memory. For example, **atomicAdd()** reads a word at some address in global or shared memory, adds a number to it, and writes the result back to the same address. The operation is atomic in the sense that it is guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. Atomic functions can only be used in device functions and atomic functions operating on mapped page-locked memory (Mapped Memory) are not atomic from the point of view of the host or other devices.

As mentioned in Table 11, the support for atomic operations varies with the compute capability:

- Atomic functions are only available for devices of compute capability 1.1 and higher.
- Atomic functions operating on 32-bit integer values in shared memory and atomic functions operating on 64-bit integer values in global memory are only available for devices of compute capability 1.2 and higher.
- Atomic functions operating on 64-bit integer values in shared memory are only available for devices of compute capability 2.x and higher.
- Only **atomicExch()** and **atomicAdd()** can operate on 32-bit floating-point values:
  - in global memory for **atomicExch()** and devices of compute capability 1.1 and higher.
  - in shared memory for **atomicExch()** and devices of compute capability 1.2 and higher.
  - in global and shared memory for **atomicAdd()** and devices of compute capability 2.x and higher.

Note however that any atomic operation can be implemented based on **atomicCAS()** (Compare And Swap). For example, **atomicAdd()** for double-precision floating-point numbers can be implemented as follows:

```
__device__ double atomicAdd(double* address, double val)
{
    unsigned long long int* address_as_ull =
                              (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;
    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +
                               __longlong_as_double(assumed)));
    } while (assumed != old);
    return __longlong_as_double(old);
}
```

B.12.1. Arithmetic Functions

B.12.1.1. atomicAdd()

```
int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
                                 unsigned long long int val);
float atomicAdd(float* address, float val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old + val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

The floating-point version of **atomicAdd()** is only supported by devices of compute capability 2.x and higher.

B.12.1.2. atomicSub()

```
int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(old - val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

B.12.1.3. atomicExch()

```
int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address,
                        unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
                                  unsigned long long int val);
float atomicExch(float* address, float val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory and stores **val** back to memory at the same address. These two operations are performed in one atomic transaction. The function returns **old**.

B.12.1.4. atomicMin()

```
int atomicMin(int* address, int val);
unsigned int atomicMin(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicMin(unsigned long long int* address,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes the minimum of **old** and **val**, and stores the result back to memory

at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

The 64-bit version of **atomicMin()** is only supported by devices of compute capability 3.5 and higher.

B.12.1.5. atomicMax()

```
int atomicMax(int* address, int val);
unsigned int atomicMax(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicMax(unsigned long long int* address,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes the maximum of **old** and **val**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

The 64-bit version of **atomicMax()** is only supported by devices of compute capability 3.5 and higher.

B.12.1.6. atomicInc()

```
unsigned int atomicInc(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **((old >= val) ? 0 : (old+1))**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

B.12.1.7. atomicDec()

```
unsigned int atomicDec(unsigned int* address,
                       unsigned int val);
```

reads the 32-bit word **old** located at the address **address** in global or shared memory, computes **(((old == 0) | (old > val)) ? val : (old-1))**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

B.12.1.8. atomicCAS()

```
int atomicCAS(int* address, int compare, int val);
unsigned int atomicCAS(unsigned int* address,
                       unsigned int compare,
                       unsigned int val);
unsigned long long int atomicCAS(unsigned long long int* address,
                                 unsigned long long int compare,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old == compare ? val : old)** , and stores the result back

to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old** (Compare And Swap).

B.12.2. Bitwise Functions

B.12.2.1. atomicAnd()

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicAnd(unsigned long long int* address,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old & val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

The 64-bit version of **atomicAnd()** is only supported by devices of compute capability 3.5 and higher.

B.12.2.2. atomicOr()

```
int atomicOr(int* address, int val);
unsigned int atomicOr(unsigned int* address,
                      unsigned int val);
unsigned long long int atomicOr(unsigned long long int* address,
                                unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old | val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

The 64-bit version of **atomicOr()** is only supported by devices of compute capability 3.5 and higher.

B.12.2.3. atomicXor()

```
int atomicXor(int* address, int val);
unsigned int atomicXor(unsigned int* address,
                       unsigned int val);
unsigned long long int atomicXor(unsigned long long int* address,
                                 unsigned long long int val);
```

reads the 32-bit or 64-bit word **old** located at the address **address** in global or shared memory, computes **(old ^ val)**, and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

The 64-bit version of **atomicXor()** is only supported by devices of compute capability 3.5 and higher.

B.13. Warp Vote Functions

```
int __all(int predicate);
int __any(int predicate);
unsigned int __ballot(int predicate);
```

The warp vote functions allow the threads of a given warp to perform a reduction-and-broadcast operation. These functions take as input an integer **predicate** from each thread in the warp and compare those values with zero. The results of the comparisons are 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(predicate):
Evaluate **predicate** for all active threads of the warp and return non-zero if and only if **predicate** evaluates to non-zero for all of them. Supported by devices of compute capability 1.2 and higher.

__any(predicate):
Evaluate **predicate** for all active threads of the warp and return non-zero if and only if **predicate** evaluates to non-zero for any of them. Supported by devices of compute capability 1.2 and higher.

__ballot(predicate):
Evaluate **predicate** for all active threads of the warp and return an integer whose Nth bit is set if and only if **predicate** evaluates to non-zero for the Nth thread of the warp and the Nth thread is active. Supported by devices of compute capability 2.0 and higher.

Notes

For each of these warp vote operations, the result excludes threads that are inactive (e.g., due to warp divergence). Inactive threads are represented by 0 bits in the value returned by **__ballot()** and are not considered in the reductions performed by **__all()** and **__any()**.

B.14. Warp Shuffle Functions

__shfl, **__shfl_up**, **__shfl_down**, **__shfl_xor** exchange a variable between threads within a warp.

Supported by devices of compute capability 3.x.

B.14.1. Synopsis

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

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

B.14.2. Description

The **__shfl()** intrinsics permit exchanging of a variable between threads within a warp without use of shared memory. The exchange occurs simultaneously for all active threads within the warp, moving 4 bytes of data per thread. Exchange of 8-byte quantities must be broken into two separate invocations of **__shfl()**.

Threads within a warp are referred to as *lanes*, and for devices of compute capability 3.x may have an index between 0 and **warpSize-1** (inclusive). Four source-lane addressing modes are supported:

__shfl()
Direct copy from indexed lane

__shfl_up()
Copy from a lane with lower ID relative to caller

__shfl_down()
Copy from a lane with higher ID relative to caller

__shfl_xor()
Copy from a lane based on bitwise XOR of own lane ID

Threads may only read data from another thread which is actively participating in the **__shfl()** command. If the target thread is inactive, the retrieved value is undefined.

All the **__shfl()** intrinsics take an optional width parameter which permits sub-division of the warp into segments - for example to exchange data between 4 groups of 8 lanes in a SIMD manner. 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. A thread may only exchange data with others in its own subsection. width must have a value which is a power of 2 so that the warp can be subdivided equally; results are undefined if width is not a power of 2, or is a number greater than **warpSize**.

__shfl() returns the value of **var** held by the thread whose ID is given by **srcLane**. If **srcLane** is outside the range **[0:width-1]**, then the thread's own value of var is returned.

__shfl_up() calculates a source lane ID by subtracting **delta** from the caller's lane ID. The value of **var** held by the resulting lane ID is returned: in effect, **var** is shifted up the

warp by **delta** lanes. The source lane index will not wrap around the value of **width**, so effectively the lower **delta** lanes will be unchanged.

__shfl_down() calculates a source lane ID by adding **delta** to the caller's lane ID. The value of **var** held by the resulting lane ID is returned: this has the effect of shifting **var** down the warp by **delta** lanes. As for **__shfl_up()**, the ID number of the source lane will not wrap around the value of width and so the upper **delta** lanes will remain unchanged.

__shfl_xor() calculates a source line ID by performing a bitwise XOR of the caller's lane ID with **laneMask**: the value of **var** held by the resulting lane ID is returned. If the resulting lane ID falls outside the range permitted by **width**, the thread's own value of **var** is returned. This mode implements a butterfly addressing pattern such as is used in tree reduction and broadcast.

B.14.3. Return Value

All **__shfl()** intrinsics return the 4-byte word referenced by var from the source lane ID as an unsigned integer. If the source lane ID is out of range or the source thread has exited, the calling thread's own var is returned.

B.14.4. Notes

All **__shfl()** intrinsics share the same semantics with respect to code motion as the vote intrinsics **__any()** and **__all()**.

Threads may only read data from another thread which is actively participating in the **__shfl()** command. If the target thread is inactive, the retrieved value is undefined.

width must be a power-of-2 (i.e., 2, 4, 8, 16 or 32). Results are unspecified for other values.

Types other than int or float must first be cast in order to use the **__shfl()** intrinsics.

B.14.5. Examples

B.14.5.1. Broadcast of a single value across a warp

```
      __global__ void bcast(int arg) {
    int laneId = threadIdx.x & 0x1f;
    int value;
    if (laneId == 0)        // Note unused variable for
        value = arg;        // all threads except lane 0
    value = __shfl(value, 0);   // Get "value" from lane 0
    if (value != arg)
        printf("Thread %d failed.\n", threadIdx.x);
}

void main() {
    bcast<<< 1, 32 >>>(1234);
    cudaDeviceSynchronize();
}
```

B.14.5.2. Inclusive plus-scan across sub-partitions of 8 threads

```
__global__ void scan4() {
    int laneId = threadIdx.x & 0x1f;
    // Seed sample starting value (inverse of lane ID)
    int value = 31 - laneId;

    // Loop to accumulate scan within my partition.
    // Scan requires log2(n) == 3 steps for 8 threads
    // It works by an accumulated sum up the warp
    // by 1, 2, 4, 8 etc. steps.
    for (int i=1; i<=4; i*=2) {
        // Note: shfl requires all threads being
        // accessed to be active. Therefore we do
        // the __shfl unconditionally so that we
        // can read even from threads which won't do a
        // sum, and then conditionally assign the result.
        int n = __shfl_up(value, i, 8);
        if (laneId >= i)
            value += n;
    }

    printf("Thread %d final value = %d\n", threadIdx.x, value);
}

void main() {
    scan4<<< 1, 32 >>>();
    cudaDeviceSynchronize();
}
```

B.14.5.3. Reduction across a warp

```
__global__ void warpReduce() {
    int laneId = threadIdx.x & 0x1f;
    // Seed starting value as inverse lane ID
    int value = 31 - laneId;

    // Use XOR mode to perform butterfly reduction
    for (int i=16; i>=1; i/=2)
        value += __shfl_xor(value, i, 32);

    // "value" now contains the sum across all threads
    printf("Thread %d final value = %d\n", threadIdx.x, value);
}

void main() {
    warpReduce<<< 1, 32 >>>();
    cudaDeviceSynchronize();
}
```

B.15. Profiler Counter Function

Each multiprocessor has a set of sixteen hardware counters that an application can increment with a single instruction by calling the **`__prof_trigger()`** function.

```
[void __prof_trigger(int counter);
```

increments by one per warp the per-multiprocessor hardware counter of index **`counter`**. Counters 8 to 15 are reserved and should not be used by applications.

The value of counters 0, 1, ..., 7 can be obtained via **nvprof** by **nvprof --events prof_trigger_0x** where **x** is 0, 1, ..., 7. The value of those counters for the first multiprocessor can also be obtained via the old CUDA command-line profiler by listing **prof_trigger_00, prof_trigger_01, ..., prof_trigger_07** , etc. in the **profiler.conf** file (see the profiler manual for more details). All counters are reset before each kernel launch (note that when collecting counters, kernel launches are synchronous as mentioned in Concurrent Execution between Host and Device).

B.16. Assertion

Assertion is only supported by devices of compute capability 2.x and higher. It is not supported on MacOS, regardless of the device, and loading a module that references the assert function on Mac OS will fail.

```
void assert(int expression);
```

stops the kernel execution if **expression** is equal to zero. If the program is run within a debugger, this triggers a breakpoint and the debugger can 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 synchronization with the host via **cudaDeviceSynchronize()**, **cudaStreamSynchronize()**, or **cudaEventSynchronize()**. The format of this message is as follows:

```
<filename>:<line number>:<function>:
block: [blockId.x,blockId.x,blockIdx.z],
thread: [threadIdx.x,threadIdx.y,threadIdx.z]
Assertion `<expression>` failed.
```

Any subsequent host-side synchronization calls made for the same device will return **cudaErrorAssert**. No more commands can be sent to this device until **cudaDeviceReset()** is called to reinitialize the device.

If **expression** is different from zero, the kernel execution is unaffected.

For example, the following program from source file *test.cu*

```
#include <assert.h>

// assert() is only supported
// for devices of compute capability 2.0 and higher
#if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ < 200)
#undef  assert
#define assert(arg)
#endif

__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(int argc, char* argv[])
{
    testAssert<<<1,1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
test.cu:19: void testAssert(): block: [0,0,0], thread: [0,0,0] Assertion
 `should_be_one` failed.
```

Assertions are for debugging purposes. They can affect performance and it is therefore recommended to disable them in production code. They can be disabled at compile time by defining the **`NDEBUG`** preprocessor macro before including **`assert.h`**. Note that **`expression`** should not be an expression with side effects (something like **`(++i > 0)`**, for example), otherwise disabling the assertion will affect the functionality of the code.

B.17. Formatted Output

Formatted output is only supported by devices of compute capability 2.x and higher.

```
int printf(const char *format[, arg, ...]);
```

prints formatted output from a kernel to a host-side output stream.

The in-kernel **`printf()`** function behaves in a similar way to the standard C-library printf() function, and the user is referred to the host system's manual pages for a complete description of **`printf()`** behavior. In essence, the string passed in as **`format`** is output to a stream on the host, with substitutions made from the argument list wherever a format specifier is encountered. Supported format specifiers are listed below.

The **`printf()`** command is executed as any other device-side function: per-thread, and in the context of the calling thread. From a multi-threaded kernel, this means that a straightforward call to **`printf()`** will be executed by every thread, using that thread's

data as specified. Multiple versions of the output string will then appear at the host stream, once for each thread which encountered the **printf()**.

It is up to the programmer to limit the output to a single thread if only a single output string is desired (see Examples for an illustrative example).

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.

B.17.1. Format Specifiers

As for standard **printf()**, format specifiers take the form: **%[flags][width][.precision][size]type**

The following fields are supported (see widely-available documentation for a complete description of all behaviors):

- *Flags*: `#' ` ' `0' `+' `-'
- *Width*: `*' `0-9'
- *Precision*: `0-9'
- *Size*: `h' `l' `ll'
- *Type*: `%cdiouxXpeEfgGaAs'

Note that CUDA's **printf()** will accept any combination of flag, width, precision, size and type, whether or not overall they form a valid format specifier. In other words, "**%hd**" will be accepted and printf will expect a double-precision variable in the corresponding location in the argument list.

B.17.2. Limitations

Final formatting of the **printf()** output takes place on the host system. This means that the format string must be understood by the host-system's compiler and C library. Every effort has been made to ensure that the format specifiers supported by CUDA's printf function form a universal subset from the most common host compilers, but exact behavior will be host-OS-dependent.

As described in Format Specifiers, **printf()** will accept *all* combinations of valid 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. The effect of this is that output may be undefined if the program emits a format string which contains invalid combinations.

The **printf()** command can accept at most 32 arguments in addition to the format string. Additional arguments beyond this will be ignored, and the format specifier output as-is.

Owing to the differing size of the **long** type on 64-bit Windows platforms (four bytes on 64-bit Windows platforms, eight bytes on other 64-bit platforms), a kernel which is compiled on a non-Windows 64-bit machine but then run on a win64 machine will see corrupted output for all format strings which include "**%ld**". It is recommended that the compilation platform matches the execution platform to ensure safety.

The output buffer for **printf()** is set to a fixed size before kernel launch (see Associated Host-Side API). It is circular and if more output is produced during kernel execution than can fit in the buffer, older output is overwritten. It is flushed only when one of these 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()**.

Note that the buffer is not flushed automatically when the program exits. The user must call **cudaDeviceReset()** or **cuCtxDestroy()** explicitly, as shown in the examples below.

Internally **printf()** uses a shared data structure and so it is possible that calling **printf()** might change the order of execution of threads. In particular, a thread which calls **printf()** might take a longer execution path than one which does not call **printf()**, and that path length is dependent upon the parameters of the **printf()**. Note, however, that CUDA makes no guarantees of thread execution order except at explicit **__syncthreads()** barriers, so it is impossible to tell whether execution order has been modified by **printf()** or by other scheduling behaviour in the hardware.

B.17.3. Associated Host-Side API

The following API functions get and set the size of the buffer used to transfer the **printf()** arguments and internal metadata to the host (default is 1 megabyte):

- **cudaDeviceGetLimit(size_t* size,cudaLimitPrintfFifoSize)**
- **cudaDeviceSetLimit(cudaLimitPrintfFifoSize, size_t size)**

B.17.4. Examples

The following code sample:

```
#include "stdio.h"

// printf() is only supported
// for devices of compute capability 2.0 and higher
#if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ < 200)
  # error printf is only supported on devices of compute capability 2.0 and
 higher, please compile with -arch=sm_20 or higher
#endif

__global__ void helloCUDA(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

int main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Hello thread 2, f=1.2345
Hello thread 1, f=1.2345
Hello thread 4, f=1.2345
Hello thread 0, f=1.2345
Hello thread 3, f=1.2345
```

Notice how each thread encounters the **`printf()`** command, so there are as many lines of output as there were threads launched in the grid. As expected, global values (i.e., **`float f`**) are common between all threads, and local values (i.e., **`threadIdx.x`**) are distinct per-thread.

The following code sample:

```
#include "stdio.h"

// printf() is only supported
// for devices of compute capability 2.0 and higher
#if defined(__CUDA_ARCH__) && (__CUDA_ARCH__ < 200)
     # error printf is only supported on devices of compute capability 2.0 and
 higher, please compile with -arch=sm_20 or higher
#endif

__global__ void helloCUDA(float f)
{
    if (threadIdx.x == 0)
        printf("Hello thread %d, f=%f\n", threadIdx.x, f) ;
}

int main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Hello thread 0, f=1.2345
```

Self-evidently, the **if()** statement limits which threads will call **printf**, so that only a single line of output is seen.

B.18. Dynamic Global Memory Allocation and Operations

Dynamic global memory allocation and operations are only supported by devices of compute capability 2.x and higher.

```
void* malloc(size_t size);
void free(void* ptr);
```

allocate and free memory dynamically from a fixed-size heap in global memory.

```
void* memcpy(void* dest, const void* src, size_t size);
```

copy **size** bytes from the memory location pointed by **src** to the memory location pointed by **dest**.

```
void* memset(void* ptr, int value, size_t size);
```

set **size** bytes of memory block pointed by **ptr** to **value** (interpreted as an unsigned char).

The CUDA in-kernel **malloc()** function allocates at least **size** bytes from the device heap and returns a pointer to the allocated memory or NULL if insufficient memory exists to fulfill the request. The returned pointer is guaranteed to be aligned to a 16-byte boundary.

The CUDA in-kernel **free()** function deallocates the memory pointed to by **ptr**, which must have been returned by a previous call to **malloc()**. If **ptr** is NULL, the call to **free()** is ignored. Repeated calls to **free()** with the same **ptr** has undefined behavior.

The memory allocated by a given CUDA thread via **malloc()** remains allocated for the lifetime of the CUDA context, or until it is explicitly released by a call to **free()**. It can be used by any other CUDA threads even from subsequent kernel launches. Any CUDA thread may free memory allocated by another thread, but care should be taken to ensure that the same pointer is not freed more than once.

B.18.1. Heap Memory Allocation

The device memory heap has a fixed size that must be specified before any program using **malloc()** or **free()** is loaded into the context. A default heap of eight megabytes is allocated if any program uses **malloc()** without explicitly specifying the heap size.

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 via the CUDA driver API (see Module), or implicitly via the CUDA runtime API (see CUDA C Runtime). If the memory allocation fails, the module load will generate a **CUDA_ERROR_SHARED_OBJECT_INIT_FAILED** error.

Heap size cannot be changed once a module load has occurred and it does not resize dynamically according to need.

Memory reserved for the device heap is in addition to memory allocated through host-side CUDA API calls such as **cudaMalloc()**.

B.18.2. Interoperability with Host Memory API

Memory allocated via **malloc()** cannot be freed using the runtime (i.e., by calling any of the free memory functions from Device Memory).

Similarly, memory allocated via the runtime (i.e., by calling any of the memory allocation functions from Device Memory) cannot be freed via **free()**.

B.18.3. Examples

B.18.3.1. Per Thread Allocation

The following code sample:

```
  #include <stdlib.h>
#include <stdio.h>

__global__ void mallocTest()
{
    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);
    mallocTest<<<1, 5>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

will output:

```
Thread 0 got pointer: 00057020
Thread 1 got pointer: 0005708c
Thread 2 got pointer: 000570f8
Thread 3 got pointer: 00057164
Thread 4 got pointer: 000571d0
```

Notice how each thread encounters the **`malloc()`** and **`memset()`** commands and so receives and initializes its own allocation. (Exact pointer values will vary: these are illustrative.)

B.18.3.2. Per Thread Block Allocation

```
      #include <stdlib.h>

__global__ void mallocTest()
{
    __shared__ int* data;

    // The first thread in the block does the allocation and initialization
    // and then shares the pointer with all other threads through shared memory,
    // so that access can easily be coalesced.
    // 64 bytes per thread are allocated.
    if (threadIdx.x == 0) {
        size_t size = blockDim.x * 64;
        data = (int*)malloc(size);
        memset(data, 0, size);
    }
    __syncthreads();

    // Check for failure
    if (data == NULL)
        return;

    // Threads index into the memory, ensuring coalescence
    int* ptr = data;
    for (int i = 0; i < 64; ++i)
        ptr[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);
    mallocTest<<<10, 128>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

B.18.3.3. Allocation Persisting Between Kernel Launches

```
#include <stdlib.h>
#include <stdio.h>

#define NUM_BLOCKS 20

__device__ int* dataptr[NUM_BLOCKS]; // Per-block pointer

__global__ void allocmem()
{
    // Only the first thread in the block does the allocation
    // since we want only one allocation per block.
    if (threadIdx.x == 0)
        dataptr[blockIdx.x] = (int*)malloc(blockDim.x * 4);
    __syncthreads();

    // Check for failure
    if (dataptr[blockIdx.x] == NULL)
        return;

    // Zero the data with all threads in parallel
    dataptr[blockIdx.x][threadIdx.x] = 0;
}

// Simple example: store thread ID into each element
__global__ void usemem()
{
    int* ptr = dataptr[blockIdx.x];
    if (ptr != NULL)
        ptr[threadIdx.x] += threadIdx.x;
}

// Print the content of the buffer before freeing it
__global__ void freemem()
{
    int* ptr = dataptr[blockIdx.x];
    if (ptr != NULL)
        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
    allocmem<<< NUM_BLOCKS, 10 >>>();

    // Use memory
    usemem<<< NUM_BLOCKS, 10 >>>();
    usemem<<< NUM_BLOCKS, 10 >>>();
    usemem<<< NUM_BLOCKS, 10 >>>();

    // Free memory
    freemem<<< NUM_BLOCKS, 10 >>>();

    cudaDeviceSynchronize();

    return 0;
}
```

B.19. Execution Configuration

Any call to a **__global__** function must specify the *execution configuration* for that call. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device, as well as the associated stream (see CUDA C Runtime for a description of streams).

The execution configuration is specified by inserting an expression of the form **<<< Dg, Db, Ns, S >>>** between the function name and the parenthesized argument list, where:

- **Dg** is of type **dim3** (see dim3) and specifies the dimension and size of the grid, such that **Dg.x * Dg.y * Dg.z** equals the number of blocks being launched; Dg.z must be equal to 1 for devices of compute capability 1.x;
- **Db** is of type **dim3** (see dim3) and specifies the dimension and size of each block, such that **Db.x * Db.y * Db.z** equals the number of threads per block;
- **Ns** is of type **size_t** and specifies the number of bytes in shared memory that is dynamically allocated per block for this call in addition to the statically allocated memory; this dynamically allocated memory is used by any of the variables declared as an external array as mentioned in __shared__; **Ns** is an optional argument which defaults to 0;
- **S** is of type **cudaStream_t** and specifies the associated stream; **S** is an optional argument which defaults to 0.

As an example, a function declared as

```
__global__ void Func(float* parameter);
```

must be called like this:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

The arguments to the execution configuration are evaluated before the actual function arguments. For devices of compute capability 1.x, they are passed via shared memory to the device.

The function call will fail if **Dg** or **Db** are greater than the maximum sizes allowed for the device as specified in Compute Capabilities, or if **Ns** is greater than the maximum amount of shared memory available on the device, minus the amount of shared memory required for static allocation. For devices of compute capability 1.x, the execution configuration and the function arguments also consume shared memory.

B.20. Launch Bounds

As discussed in detail in Multiprocessor Level, the fewer registers a kernel uses, the more threads and thread blocks are likely to reside on a multiprocessor, which can improve performance.

Therefore, the compiler uses heuristics to minimize register usage while keeping register spilling (see Device Memory Accesses) and instruction count to a minimum. An application 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)
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.

If launch bounds are specified, the compiler first derives from them the upper limit *L* on the number of registers the kernel should use to ensure that **minBlocksPerMultiprocessor** blocks (or a single block if **minBlocksPerMultiprocessor** is not specified) of **maxThreadsPerBlock** threads can reside on the multiprocessor (see Hardware Multithreading 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 in the following way:

- If the initial register usage is higher than *L*, the compiler reduces it further until it becomes less or equal to *L*, usually at the expense of more local memory usage and/or higher number of instructions;
- If the initial register usage is lower than *L*
  - If **maxThreadsPerBlock** is specified and **minBlocksPerMultiprocessor** is not, the compiler uses **maxThreadsPerBlock** to determine the register usage thresholds for the transitions between **n** and **n+1** resident blocks (i.e., when using one less register makes room for an additional resident block as in the example of Multiprocessor Level) and then applies similar heuristics as when no launch bounds are specified;

- If both **minBlocksPerMultiprocessor** and **maxThreadsPerBlock** are specified, the compiler may increase register usage as high as *L* to reduce the number of instructions and better hide single thread instruction latency.

A kernel will fail to launch if it is executed with more threads per block than its launch bound **maxThreadsPerBlock**.

Optimal launch bounds for a given kernel will usually differ across major architecture revisions. The sample code below shows how this is typically handled in device code using the **__CUDA_ARCH__** macro introduced in Application Compatibility

```
#define THREADS_PER_BLOCK          256
#if __CUDA_ARCH__ >= 200
    #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

// Device code
__global__ void
__launch_bounds__(MY_KERNEL_MAX_THREADS, MY_KERNEL_MIN_BLOCKS)
MyKernel(...)
{
    ...
}
```

In the common case where **MyKernel** is invoked with the maximum number of threads per block (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>>>(...);
```

This will not work however since **__CUDA_ARCH__** is undefined in host code as mentioned in Application Compatibility, so **MyKernel** will launch with 256 threads per block even when **__CUDA_ARCH__** is greater or equal to 200. Instead the number of threads per block should be determined:

- Either at compile time using a macro 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 >= 2 ?
                      2 * THREADS_PER_BLOCK : THREADS_PER_BLOCK);
  MyKernel<<<blocksPerGrid, threadsPerBlock>>>(...);
  ```

Register usage is reported by the **--ptxas options=-v** compiler option. The number of resident blocks can be derived from the occupancy reported by the CUDA profiler (see Device Memory Accessesfor a definition of occupancy).

Register usage can also be controlled for all **__global__** functions in a file using the **maxrregcount** compiler option. The value of **maxrregcount** is ignored for functions with launch bounds.

B.21. #pragma unroll

By default, the compiler unrolls small loops with a known trip count. The **#pragma unroll** directive however can be used to control unrolling of any given loop. It must be placed immediately before the loop and only applies to that loop. It is optionally followed by a number that specifies how many times the loop must be unrolled.

For example, in this code sample:

```
#pragma unroll 5
for (int i = 0; i < n; ++i)
```

the loop will be unrolled 5 times. The compiler will also insert code to ensure correctness (in the example above, to ensure that there will only be **n** iterations if **n** is less than 5, for example). It is up to the programmer to make sure that the specified unroll number gives the best performance.

#pragma unroll 1 will prevent the compiler from ever unrolling a loop.

If no number is specified after **#pragma unroll**, the loop is completely unrolled if its trip count is constant, otherwise it is not unrolled at all.

B.22. SIMD Video Instructions

PTX ISA version 3.0 includes SIMD (Single Instruction, Multiple Data) video instructions which operate on pairs of 16-bit values and quads of 8-bit values. These are available on devices of compute capability 3.0.

The SIMD video instructions are:

- vadd2, vadd4
- vsub2, vsub4
- vavrg2, vavrg4
- vabsdiff2, vabsdiff4
- vmin2, vmin4
- vmax2, vmax4
- vset2, vset4

PTX instructions, such as the SIMD video instructions, can be included in CUDA programs by way of the assembler, **asm()**, statement.

The basic syntax of an asm() statement is:

```
asm("template-string" : "constraint"(output) : "constraint"(input));
```

An example of using the **vabsdiff4** PTX instruction is:

```
asm("vabsdiff4.u32.u32.u32.add" " %0, %1, %2, %3;": "=r" (result):"r" (A), "r"
 (B), "r" (C));
```

This uses the **vabsdiff4** instruction to compute an integer quad byte SIMD sum of absolute differences. The absolute difference value is computed for each byte of the unsigned integers A and B in SIMD fashion. The optional accumulate operation (**.add**) is specified to sum these differences.

Refer to the document "Using Inline PTX Assembly in CUDA" for details on using the assembly statement in your code. Refer to the PTX ISA documentation ("Parallel Thread Execution ISA Version 3.0" for example) for details on the PTX instructions for the version of PTX that you are using.

Appendix C. CUDA DYNAMIC PARALLELISM

C.1. Introduction

C.1.1. Overview

Dynamic Parallelism is an extension to the CUDA programming model enabling a CUDA kernel to create and synchronize with new work directly on the GPU. The creation of parallelism dynamically at whichever point in a program that it is needed offers exciting new capabilities.

The ability to create work directly from the GPU can reduce the need to transfer execution control and data between host and device, as launch configuration decisions can now be made at runtime by threads executing on the device. Additionally, data-dependent parallel work can be generated inline within a kernel at run-time, taking advantage of the GPU's hardware schedulers and load balancers dynamically and adapting in response to data-driven decisions or workloads. Algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism may more transparently be expressed.

This document describes the extended capabilities of CUDA which enable Dynamic Parallelism, including the modifications and additions to the CUDA programming model necessary to take advantage of these, as well as guidelines and best practices for exploiting this added capacity.

Dynamic Parallelism is only supported by devices of compute capability 3.5 and higher.

C.1.2. Glossary

Definitions for terms used in this guide.

Grid

A Grid is a collection of *Threads*. Threads in a Grid execute a *Kernel Function* and are divided into *Thread Blocks*.

Thread Block
A Thread Block is a group of threads which execute on the same multiprocessor (*SMX*). Threads within a Thread Block have access to shared memory and can be explicitly synchronized.

Kernel Function
A Kernel Function is an implicitly parallel subroutine that executes under the CUDA execution and memory model for every Thread in a Grid.

Host
The Host refers to the execution environment that initially invoked CUDA. Typically the thread running on a system's CPU processor.

Parent
A *Parent Thread*, Thread Block, or Grid is one that has launched new grid(s), the *Child* Grid(s). The Parent is not considered completed until all of its launched Child Grids have also completed.

Child
A Child thread, block, or grid is one that has been launched by a Parent grid. A Child grid must complete before the Parent Thread, Thread Block, or Grid are considered complete.

Thread Block Scope
Objects with Thread Block Scope have the lifetime of a single Thread Block. They only have defined behavior when operated on by Threads in the Thread Block that created the object and are destroyed when the Thread Block that created them is complete.

Device Runtime
The Device Runtime refers to the runtime system and APIs available to enable Kernel Functions to use Dynamic Parallelism.

C.2. Execution Environment and Memory Model

C.2.1. Execution Environment

The CUDA execution model is based on primitives of threads, thread blocks, and grids, with kernel functions defining the program executed by individual threads within a thread block and grid. When a kernel function is invoked the grid's properties are described by an execution configuration, which has a special syntax in CUDA. Support for dynamic parallelism in CUDA extends the ability to configure, launch, and synchronize upon new grids to threads that are running on the device.

C.2.1.1. Parent and Child Grids

A device thread that configures and launches a new grid belongs to the parent grid, and the grid created by the invocation is 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. Even if the invoking threads do not explicitly synchronize on the child grids

launched, the runtime guarantees an implicit synchronization between the parent and child.

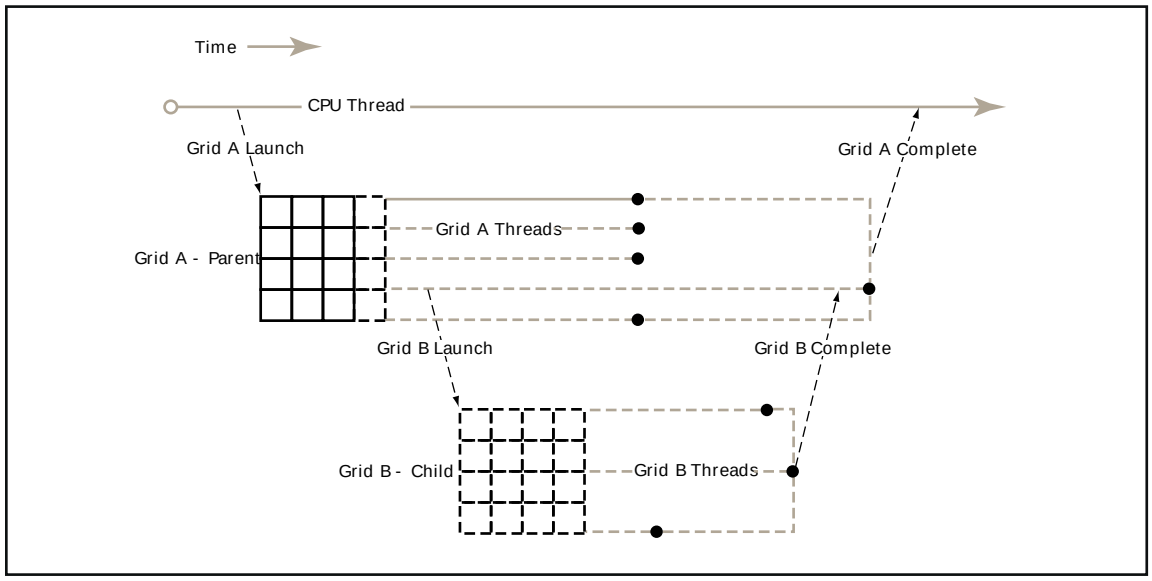


Figure 12 Parent-Child Launch Nesting

C.2.1.2. Scope of CUDA Primitives

On both host and device, the CUDA runtime offers an API for launching kernels, for waiting for launched work to complete, and for tracking dependencies between launches via streams and events. On the host system, the state of launches and the CUDA primitives referencing streams and events are shared by all threads within a process; however processes execute independently and may not share CUDA objects.

A similar hierarchy exists on the device: launched kernels and CUDA objects are visible to all threads in a thread block, but are independent between thread blocks. This means for example that a stream may be created by one thread and used by any other thread in the same thread block, but may not be shared with threads in any other thread block.

C.2.1.3. Synchronization

CUDA runtime operations from any thread, including kernel launches, are visible across a thread block. This means that an invoking thread in the parent grid may perform synchronization on the grids launched by that thread, by other threads in the thread block, or on streams created within the same thread block. Execution of a thread block is not considered complete until all launches by all threads in the block have completed. If all threads in a block exit before all child launches have completed, a synchronization operation will automatically be triggered.

C.2.1.4. Streams and Events

CUDA *Streams* and *Events* allow control over dependencies between grid launches: grids 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 thread block scope but have undefined behavior when used outside of the thread block where they were created. As described above, all work launched by a thread block is implicitly synchronized when the block 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 thread block 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.

C.2.1.5. Ordering and Concurrency

The ordering of kernel launches from the device runtime follows CUDA Stream ordering semantics. Within a thread block, all kernel launches into the same stream are executed in-order. With multiple threads in the same thread block launching into the same stream, the ordering within the stream is dependent on the thread scheduling within the block, which may be controlled with synchronization primitives such as **`__syncthreads()`**.

Note that because streams are shared by all threads within a thread block, the implicit *NULL* stream is also shared. If multiple threads in a thread block launch into the implicit stream, then these launches will be executed in-order. If concurrency is desired, explicit named streams should be used.

Dynamic Parallelism enables concurrency to be expressed more easily within a program; however, the device runtime introduces no new concurrency guarantees within the CUDA execution model. There is no guarantee of concurrent execution between any number of different thread blocks on a device.

The lack of concurrency guarantee extends to parent thread blocks and their child grids. When a parent thread block launches a child grid, the child is not guaranteed to begin execution until the parent thread block reaches an explicit synchronization point (e.g. **`cudaDeviceSynchronize()`**).

While concurrency will often easily be achieved, it may vary as a function of deviceconfiguration, application workload, and runtime scheduling. It is therefore unsafe to depend upon any concurrency between different thread blocks.

C.2.1.6. 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.

C.2.2. Memory Model

Parent and child grids share the same global and constant memory storage, but have distinct local and shared memory.

C.2.2.1. Coherence and Consistency

C.2.2.1.1. Global Memory

Parent and child grids have coherent access to global memory, with weak consistency guarantees between child and parent. There are two points in the execution of a child grid when its view of memory is fully consistent with the parent thread: when the child grid is invoked by the parent, and when the child grid completes as signaled by a synchronization API invocation in the parent thread.

All global memory operations in the parent thread prior to the child grid's invocation are visible to the child grid. All memory operations of the child grid are visible to the parent after the parent has synchronized on the child grid's completion.

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]** would be guaranteed to be seen by the child). When the child grid returns, thread 0 is guaranteed to see modifications made by the threads in its child grid. Those modifications become available to the other threads of the parent grid only after the second **__syncthreads()** call:

```
__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);
        cudaDeviceSynchronize();
    }

    __syncthreads();
}

void host_launch(int *data) {
    parent_launch<<< 1, 256 >>>(data);
}
```

C.2.2.1.2. Zero Copy Memory

Zero-copy system memory has identical coherence and consistency guarantees to global memory, and follows the semantics detailed above. A kernel may not allocate or free zero-copy memory, but may use pointers to zero-copy passed in from the host program.

C.2.2.1.3. Constant Memory

Constants are immutable and may not be modified from the device, even between parent and child launches. That is to say, the value of all **__constant__** variables must

be set from the host prior to launch. Constant memory is inherited automatically by all child kernels from their respective parents.

Taking the address of a constant memory object from within a kernel thread has the same semantics as for all CUDA programs, and passing that pointer from parent to child or from a child to parent is naturally supported.

C.2.2.1.4. 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.

Note that 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.

C.2.2.1.5. 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 will be 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);
}
```

C.2.2.1.6. 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, writes to memory by a child will be reflected in the texture memory accesses by a parent, but only after the parent synchronizes on the child's completion. Concurrent accesses by parent and child may result in inconsistent data.

C.3. Programming Interface

C.3.1. CUDA C/C++ Reference

This section describes changes and additions to the CUDA C/C++ language extensions for supporting *Dynamic Parallelism*.

The language interface and API available to CUDA kernels using CUDA C/C++ for Dynamic Parallelism, referred to as the *Device Runtime*, is substantially like that of the CUDA Runtime API available on the host. 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/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.

C.3.1.1. 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 and 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 thread block where the call is being made. **`S`** is an optional argument that defaults to 0.

C.3.1.1.1. 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 explicit launch-synchronization point such as **`cudaDeviceSynchronize()`**. The grid launch is posted to the device and will execute independently of the parent thread. The child grid

may begin execution at any time after launch, but is not guaranteed to begin execution until the launching thread reaches an explicit launch-synchronization point.

C.3.1.1.2. Launch Environment Configuration

All global device configuration settings (e.g., shared memory and L1 cache size as returned from **`cudaDeviceGetCacheConfig()`**, and device limits returned from **`cudaDeviceGetLimit()`**) will be inherited from the parent. That is to say if, when the parent is launched, execution is configured globally for 16k of shared memory and 48k of L1 cache, then the child's execution state will be configured identically. 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.

C.3.1.1.3. Launch from __host__ __device__ Functions

Although the device runtime enables kernel launches from either the host or device, kernel launches from **`__host__ __device__`** functions are unsupported. The compiler will fail to compile if a **`__host__ device__`** function is used to launch a kernel.

C.3.1.2. Streams

Both named and unnamed (NULL) streams are available from the device runtime. Named streams may be used by any thread within a thread-block, but stream handles may not be passed to other blocks or child/parent kernels. In other words, a stream should be treated as private to the block in which it is created. Stream handles are not guaranteed to be unique between blocks, so using a stream handle within a block that did not allocate it will result in undefined behavior.

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.

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()`** call is a host-runtime- only API and will fail to compile for the device.

As **`cudaStreamSynchronize()`** and **`cudaStreamQuery()`** are unsupported by the device runtime, **`cudaDeviceSynchronize()`** should be used instead when the application needs to know that stream-launched child kernels have completed.

C.3.1.2.1. The Implicit (NULL) Stream

Within a host program, the unnamed (NULL) stream has additional barrier synchronization semantics with other streams (see Default Stream for details). The device runtime offers a single implicit, unnamed stream shared between all threads in

a 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.

C.3.1.3. 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 for all device runtime objects, event objects may be shared between all threads withinthe thread-block which created them but are local to that block and may not be passed to other kernels, or between blocks within the same kernel. Event handles are not guaranteed to be unique between blocks, so using an event handle within a block that did not create it will result in undefined behavior.

C.3.1.4. Synchronization

The **cudaDeviceSynchronize()** function will synchronize on all work launched by any thread in the thread-block up to the point where cudaDeviceSynchronize() was called. Note that **cudaDeviceSynchronize()** may be called from within divergent code (see Block Wide Synchronization).

It is up to the program to perform sufficient additional inter-thread synchronization, for example via a call to **__syncthreads()**, if the calling thread is intended to synchronize with child grids invoked from other threads.

C.3.1.4.1. Block Wide Synchronization

The **cudaDeviceSynchronize()** function does not imply intra-block synchronization. In particular, without explicit synchronization via a **__syncthreads()** directive the calling thread can make no assumptions about what work has been launched by any thread other than itself. For example if multiple threads within a block are each launching work and synchronization is desired for all this work at once (perhaps because of event-based dependencies), it is up to the program to guarantee that this work is submitted by all threads before calling **cudaDeviceSynchronize()**.

Because the implementation is permitted to synchronize on launches from any thread in the block, it is quite possible that simultaneous calls to **cudaDeviceSynchronize()** by multiple threads will drain all work in the first call and then have no effect for the later calls.

C.3.1.5. 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 **cudaGetDeviceProperty()** 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.

C.3.1.6. Memory Declarations

C.3.1.6.1. Device and Constant Memory

Memory declared at file scope with **__device__** or **__constant__** qualifiers behave 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.

C.3.1.6.2. Textures & Surfaces

CUDA supports dynamically created texture and surface objects[1], where a texture reference may be created on the host, passed to a kernel, used by that kernel, and then destroyed from the host. The device runtime does not allow creation or destruction of texture or surface objects from within device code, but 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.

The device runtime does not support legacy module-scope (i.e., 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).

C.3.1.6.3. Shared Memory Variable Declarations

In CUDA C/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.

[1] Dynamically created texture and surface objects are an addition to the CUDA memory model introduced with CUDA 5.0. Please see the *CUDA Programming Guide* for details.

```
__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();

   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);
}
```

C.3.1.6.4. 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.

Given that device-side symbols can be referenced directly, those CUDA runtime APIs which reference symbols (e.g., **cudaMemcpyToSymbol()** or **cudaGetSymbolAddress()**) are redundant and hence not supported by the device runtime. Note 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.

C.3.1.7. 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 instead of being returned by the parent's call to **cudaDeviceSynchronize()**.

C.3.1.7.1. Launch Setup APIs

Kernel launch is a system-level mechanism exposed through the device runtime library, and as such is available directly from PTX via the underlying **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 is so that users targeting PTX will be able to enact a launch, and so that the compiler front-end can translate **<<<>>>** into these calls.

Table 4 New Device-only Launch Implementation Functions

Runtime API Launch Functions	Description of Difference From Host Runtime Behaviour (behaviour 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);
```

C.3.1.8. 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 table below provides an overview of the API relative to the version available from the host.

Table 5 Supported API Functions

Runtime API Functions	Details
cudaDeviceSynchronize	Synchronizes on work launched from thread's own block only
cudaDeviceGetCacheConfig	
cudaDeviceGetLimit	
cudaGetLastError	Last error is per-thread state, not per-block state

Runtime API Functions	Details
`cudaPeekAtLastError`	
`cudaGetErrorString`	
`cudaGetDeviceCount`	
`cudaGetDeviceProperty`	Will return properties 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`	

C.3.2. Device-side Launch from PTX

This section is for the programming language and compiler implementers who target *Parallel Thread Execution* (PTX) and plan to support *Dynamic Parallelism* in their language. It provides the low-level details related to supporting kernel launches at the PTX level.

C.3.2.1. 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.

C.3.2.1.1. 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
)
;
```

```
// PTX-level Declaration of cudaLaunchDevice() when .address_size is 32
.extern .func(.param .b32 func_retval0) cudaLaunchDevice
(
  .param .b32 func,
  .param .b32 parameterBuffer,
  .param .align 4 .b8 gridDimension[12],
  .param .align 4 .b8 blockDimension[12],
  .param .b32 sharedMemSize,
  .param .b32 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 is 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 Execution Configuration for the detailed description of launch configuration.

C.3.2.1.2. 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
// When .address_size is 64
.extern .func(.param .b64 func_retval0) cudaGetParameterBuffer
(
  .param .b64 alignment,
  .param .b64 size
)
;
```

```
// PTX-level Declaration of cudaGetParameterBuffer() when .address_size is 32
.extern .func(.param .b32 func_retval0) cudaGetParameterBuffer
(
  .param .b32 alignment,
  .param .b32 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.

C.3.2.2. 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.

C.3.3. Toolkit Support for Dynamic Parallelism

C.3.3.1. 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.

C.3.3.2. Compiling and Linking

CUDA programs are automatically linked with the host runtime library when compiled with *nvcc*, but the device runtime is shipped as a static library which must explicitly be linked with a program which wishes to use it.

The device runtime is offered as a static library (**`cudadevrt.lib`** on Windows, **`libcudadevrt.a`** under Linux and MacOS), against which a GPU application that uses the device runtime must be linked. Linking of device libraries can be accomplished through *nvcc* and/or *nvlink*. Two simple examples are shown below.

A device runtime program may be compiled and linked in a single step, if all required source files can be specified from the command line:

```
$ nvcc -arch=sm_35 -rdc=true hello_world.cu -o hello -lcudadevrt
```

It is also possible to compile CUDA .cu source files first to object files, and then link these together in a two-stage process:

```
$ nvcc -arch=sm_35 -dc hello_world.cu -o hello_world.o
$ nvcc -arch=sm_35 -rdc=true hello_world.o -o hello -lcudadevrt
```

Please see the *Using Separate Compilation* section of *The CUDA Driver Compiler NVCC* guide for more details.

C.4. Programming Guidelines

C.4.1. Basics

The device runtime is a functional subset of the host runtime. API level device management, kernel launching, device memcpy, stream management, and event management are exposed from the device runtime.

Programming for the device runtime should be familiar to someone who already has experience with CUDA. Device runtime syntax and semantics are largely the same as that of the host API, with any exceptions detailed earlier in this document.

The following example shows a simple *Hello World* program incorporating dynamic parallelism:

```
#include <stdio.h>

__global__ void childKernel()
{
    printf("Hello ");
}

__global__ void parentKernel()
{
    // launch child
    childKernel<<<1,1>>>();
    if (cudaSuccess != cudaGetLastError()) {
        return;
    }

    // wait for child to complete
    if (cudaSuccess != cudaDeviceSynchronize()) {
        return;
    }

    printf("World!\n");
}

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_35 -rdc=true hello_world.cu -o hello -lcudadevrt
```

C.4.2. Performance

C.4.2.1. Synchronization

Synchronization by one thread may impact the performance of other threads in the same *Thread Block*, even when those other threads do not call **`cudaDeviceSynchronize()`** themselves. This impact will depend upon the underlying implementation.

C.4.2.2. 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 for e.g., library

calls when made from the device compared to from the host side. This overhead is, in general, incurred for applications that link against the device runtime library.

C.4.3. 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.

C.4.3.1. Runtime

C.4.3.1.1. Memory Footprint

The device runtime system software reserves memory for various management purposes, in particular one reservation which is used for saving parent-grid state during synchronization, and a second reservation for tracking pending grid launches. Configuration controls are available to reduce the size of these reservations in exchange for certain launch limitations. See Configuration Options, below, for details.

The majority of reserved memory is allocated as backing-store for parent kernel state, for use when synchronizing on a child launch. Conservatively, this memory must support storing of state for the maximum number of live threads possible on the device. This means that each parent generation at which **`cudaDeviceSynchronize()`** is callable may require up to 150MB of device memory, depending on the device configuration, which will be unavailable for program use even if it is not all consumed.

C.4.3.1.2. Nesting and Synchronization Depth

Using the device runtime, one kernel may launch another kernel, and that kernel may launch another, and so on. Each subordinate launch is considered a new *nesting level*, and the total number of levels is the *nesting depth* of the program. The *synchronization depth* is defined as the deepest level at which the program will explicitly synchronize on a child launch. Typically this is one less than the nesting depth of the program, but if the program does not need to call **`cudaDeviceSynchronize()`** at all levels then the synchronization depth might be substantially different to the nesting depth.

The overall maximum nesting depth is limited to 24, but practically speaking the real limit will be the amount of memory required by the system for each new level (see Memory Footprint above). Any launch which would result in a kernel at a deeper level than the maximum will fail. Note that this may also apply to **`cudaMemcpyAsync()`**, which might itself generate a kernel launch. See Configuration Options for details.

By default, sufficient storage is reserved for two levels of synchronization. This maximum synchronization depth (and hence reserved storage) may be controlled by calling **`cudaDeviceSetLimit()`** and specifying **`cudaLimitDevRuntimeSyncDepth`**. The number of levels to be supported must be configured before the top-level kernel is launched from the host, in order to guarantee successful execution of a nested program. Calling **`cudaDeviceSynchronize()`** at a depth greater than the specified maximum synchronization depth will return an error.

An optimization is permitted where the system detects that it need not reserve space for the parent's state in cases where the parent kernel never calls **cudaDeviceSynchronize()**. In this case, because explicit parent/child synchronization never occurs, the memory footprint required for a program will be much less than the conservative maximum. Such a program could specify a shallower maximum synchronization depth to avoid over-allocation of backing store.

C.4.3.1.3. 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 launch pool is divided into a fixed-size pool and a virtualized pool with lower performance. The device runtime system software will try to track launch data in the fixed-size pool first. The virtualized pool will be used to track new launches when the fixed-size pool is full.

The size of the fixed-size launch pool is configurable by calling **cudaDeviceSetLimit()** from the host and specifying **cudaLimitDevRuntimePendingLaunchCount**.

C.4.3.1.4. 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
cudaLimitDevRuntimeSyncDepth	Sets the maximum depth at which **cudaDeviceSynchronize()** may be called. Launches may be performed deeper than this, but explicit synchronization deeper than this limit will return the **cudaErrorLaunchMaxDepthExceeded**. The default maximum sync depth is 2.
cudaLimitDevRuntimePendingLaunchCount	Controls the amount of memory set aside for buffering kernel launches which have not yet begun to execute, due either to unresolved dependencies or lack of execution resources. When the buffer is full, the device runtime system software will attempt to track new pending launches in a lower performance virtualized buffer. If the virtualized buffer is also full, i.e. when all available heap space is consumed, launches will not occur, and the thread's last error will be set to **cudaErrorLaunchPendingCountExceeded**. The default pending launch count is 2048 launches.

C.4.3.1.5. Memory 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	Free device memory	**cudaLimitMallocHeapSize**

C.4.3.1.6. 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.

C.4.3.1.7. 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).

Appendix D. MATHEMATICAL FUNCTIONS

The reference manual lists, along with their description, all the functions of the C/C++ standard library mathematical functions that are supported in device code, as well as all intrinsic functions (that are only supported in device code).

This appendix provides accuracy information for some of these functions when applicable.

D.1. Standard Functions

The functions from this section can be used in both host and device code.

This section specifies the error bounds of each function when executed on the device and also when executed on the host in the case where the host does not supply the function.

The error bounds are generated from extensive but not exhaustive tests, so they are not guaranteed bounds.

Single-Precision Floating-Point Functions

Addition and multiplication are IEEE-compliant, so have a maximum error of 0.5 ulp. However, on the device, the compiler often combines them into a single multiply-add instruction (FMAD) and for devices of compute capability 1.x, FMAD truncates the intermediate result of the multiplication as mentioned in Floating-Point Standard.This combination can be avoided by using the **`__fadd_[rn,rz,ru,rd]()`** and **`__fmul_[rn,rz,ru,rd]()`** intrinsic functions (see Intrinsic Functions).

The recommended way to round a single-precision floating-point operand to an integer, with the result being a single-precision floating-point number is **`rintf()`**, not **`roundf()`**. The reason is that **`roundf()`** maps to an 8-instruction sequence on the device, whereas **`rintf()`** maps to a single instruction. **`truncf()`**, **`ceilf()`**, and **`floorf()`** each map to a single instruction as well.

Table 6 Single-Precision Mathematical Standard Library Functions with Maximum ULP Error

The maximum error is stated as the absolute value of the difference in ulps between a correctly rounded single-precision result and the result returned by the CUDA library function.

Function	Maximum ulp error
`x+y`	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when addition is merged into an FMAD)
`x*y`	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when multiplication is merged into an FMAD)
`x/y`	0 for compute capability ≥ 2 when compiled with `-prec-div=true` 2 (full range), otherwise
`1/x`	0 for compute capability ≥ 2 when compiled with `-prec-div=true` 1 (full range), otherwise
`rsqrtf(x)` `1/sqrtf(x)`	2 (full range) Applies to `1/sqrtf(x)` only when it is converted to `rsqrtf(x)` by the compiler.
`sqrtf(x)`	0 for compute capability ≥ 2 when compiled with `-prec-sqrt=true` 3 (full range), otherwise
`cbrtf(x)`	1 (full range)
`rcbrtf(x)`	2 (full range)
`hypotf(x,y)`	3 (full range)
`rhypotf(x,y)`	4 (full range)
`expf(x)`	2 (full range)
`exp2f(x)`	2 (full range)
`exp10f(x)`	2 (full range)
`expm1f(x)`	1 (full range)

Function	Maximum ulp error
`logf(x)`	1 (full range)
`log2f(x)`	3 (full range)
`log10f(x)`	3 (full range)
`log1pf(x)`	2 (full range)
`sinf(x)`	2 (full range)
`cosf(x)`	2 (full range)
`tanf(x)`	4 (full range)
`sincosf(x,sptr,cptr)`	2 (full range)
`sinpif(x)`	2 (full range)
`cospif(x)`	2 (full range)
`sincospif(x,sptr,cptr)`	2 (full range)
`asinf(x)`	4 (full range)
`acosf(x)`	3 (full range)
`atanf(x)`	2 (full range)
`atan2f(y,x)`	3 (full range)
`sinhf(x)`	3 (full range)
`coshf(x)`	2 (full range)
`tanhf(x)`	2 (full range)
`asinhf(x)`	3 (full range)
`acoshf(x)`	4 (full range)
`atanhf(x)`	3 (full range)
`powf(x,y)`	8 (full range)
`erff(x)`	2 (full range)
`erfcf(x)`	6 (full range)
`erfinvf(x)`	3 (full range)
`erfcinvf(x)`	4 (full range)
`erfcxf(x)`	6 (full range)
`normcdff(x)`	6 (full range)

Function	Maximum ulp error
`normcdfinvf(x)`	5 (full range)
`lgammaf(x)`	6 (outside interval -10.001 ... -2.264; larger inside)
`tgammaf(x)`	11 (full range)
`fmaf(x,y,z)`	0 (full range)
`frexpf(x,exp)`	0 (full range)
`ldexpf(x,exp)`	0 (full range)
`scalbnf(x,n)`	0 (full range)
`scalblnf(x,l)`	0 (full range)
`logbf(x)`	0 (full range)
`ilogbf(x)`	0 (full range)
`j0f(x)`	9 for $\|x\| < 8$ otherwise, the maximum absolute error is 2.2×10^{-6}
`j1f(x)`	9 for $\|x\| < 8$ otherwise, the maximum absolute error is 2.2×10^{-6}
`jnf(x)`	For n = 128, the maximum absolute error is 2.2×10^{-6}
`y0f(x)`	9 for $\|x\| < 8$ otherwise, the maximum absolute error is 2.2×10^{-6}
`y1f(x)`	9 for $\|x\| < 8$ otherwise, the maximum absolute error is 2.2×10^{-6}
`ynf(x)`	ceil(2 + 2.5n) for $\|x\| < n$ otherwise, the maximum absolute error is 2.2×10^{-6}
`fmodf(x,y)`	0 (full range)
`remainderf(x,y)`	0 (full range)
`remquof(x,y,iptr)`	0 (full range)
`modff(x,iptr)`	0 (full range)
`fdimf(x,y)`	0 (full range)
`truncf(x)`	0 (full range)
`roundf(x)`	0 (full range)

Function	Maximum ulp error
`rintf(x)`	0 (full range)
`nearbyintf(x)`	0 (full range)
`ceilf(x)`	0 (full range)
`floorf(x)`	0 (full range)
`lrintf(x)`	0 (full range)
`lroundf(x)`	0 (full range)
`llrintf(x)`	0 (full range)
`llroundf(x)`	0 (full range)

Double-Precision Floating-Point Functions

The errors listed below only apply when compiling for devices with native double-precision support. When compiling for devices without such support, such as devices of compute capability 1.2 and lower, the `double` type gets demoted to `float` by default and the double-precision math functions are mapped to their single-precision equivalents.

The recommended way to round a double-precision floating-point operand to an integer, with the result being a double-precision floating-point number is `rint()`, not `round()`. The reason is that `round()` maps to an 8-instruction sequence on the device, whereas `rint()` maps to a single instruction. `trunc()`, `ceil()`, and `floor()` each map to a single instruction as well.

Table 7 Double-Precision Mathematical Standard Library Functions with Maximum ULP Error

The maximum error is stated as the absolute value of the difference in ulps between a correctly rounded double-precision result and the result returned by the CUDA library function.

Function	Maximum ulp error
`x+y`	0 (IEEE-754 round-to-nearest-even)
`x*y`	0 (IEEE-754 round-to-nearest-even)
`x/y`	0 (IEEE-754 round-to-nearest-even)
`1/x`	0 (IEEE-754 round-to-nearest-even)
`sqrt(x)`	0 (IEEE-754 round-to-nearest-even)

Function	Maximum ulp error
`rsqrt(x)`	1 (full range)
`cbrt(x)`	1 (full range)
`rcbrt(x)`	1 (full range)
`hypot(x,y)`	2 (full range)
`rhypot(x,y)`	1 (full range)
`exp(x)`	1 (full range)
`exp2(x)`	1 (full range)
`exp10(x)`	1 (full range)
`expm1(x)`	1 (full range)
`log(x)`	1 (full range)
`log2(x)`	1 (full range)
`log10(x)`	1 (full range)
`log1p(x)`	1 (full range)
`sin(x)`	1 (full range)
`cos(x)`	1 (full range)
`tan(x)`	2 (full range)
`sincos(x,sptr,cptr)`	1 (full range)
`sinpi(x)`	1 (full range)
`cospi(x)`	1 (full range)
`sincospi(x,sptr,cptr)`	1 (full range)
`asin(x)`	2 (full range)
`acos(x)`	2 (full range)
`atan(x)`	2 (full range)
`atan2(y,x)`	2 (full range)
`sinh(x)`	1 (full range)
`cosh(x)`	1 (full range)
`tanh(x)`	1 (full range)
`asinh(x)`	2 (full range)

Function	Maximum ulp error
`acosh(x)`	2 (full range)
`atanh(x)`	2 (full range)
`pow(x,y)`	2 (full range)
`erf(x)`	2 (full range)
`erfc(x)`	4 (full range)
`erfinv(x)`	5 (full range)
`erfcinv(x)`	6 (full range)
`erfcx(x)`	3 (full range)
`normcdf(x)`	5 (full range)
`normcdfinv(x)`	7 (full range)
`lgamma(x)`	4 (outside interval -11.0001 ... -2.2637; larger inside)
`tgamma(x)`	8 (full range)
`fma(x,y,z)`	0 (IEEE-754 round-to-nearest-even)
`frexp(x,exp)`	0 (full range)
`ldexp(x,exp)`	0 (full range)
`scalbn(x,n)`	0 (full range)
`scalbln(x,l)`	0 (full range)
`logb(x)`	0 (full range)
`ilogb(x)`	0 (full range)
`j0(x)`	7 for |x| < 8 otherwise, the maximum absolute error is 5 x 10^{-12}
`j1(x)`	7 for |x| < 8 otherwise, the maximum absolute error is 5 x 10^{-12}
`jn(x)`	For n = 128, the maximum absolute error is 5 x 10^{-12}
`y0(x)`	7 for |x| < 8

Function	Maximum ulp error
	otherwise, the maximum absolute error is 5 x 10^{-12}
`y1(x)`	7 for |x| < 8 otherwise, the maximum absolute error is 5 x 10^{-12}
`yn(x)`	For |x| > 1.5n, the maximum absolute error is 5 x 10^{-12}
`fmod(x,y)`	0 (full range)
`remainder(x,y)`	0 (full range)
`remquo(x,y,iptr)`	0 (full range)
`mod(x,iptr)`	0 (full range)
`fdim(x,y)`	0 (full range)
`trunc(x)`	0 (full range)
`round(x)`	0 (full range)
`rint(x)`	0 (full range)
`nearbyint(x)`	0 (full range)
`ceil(x)`	0 (full range)
`floor(x)`	0 (full range)
`lrint(x)`	0 (full range)
`lround(x)`	0 (full range)
`llrint(x)`	0 (full range)
`llround(x)`	0 (full range)

D.2. Intrinsic Functions

The functions from this section can only be used in device code.

Among these functions are the less accurate, but faster versions of some of the functions of Standard Functions .They have the same name prefixed with `__` (such as **`__sinf(x)`**). They are faster as they map to fewer native instructions. The compiler has an option (**`-use_fast_math`**) that forces each function in Table 8 to compile to its intrinsic counterpart. In addition to reducing the accuracy of the affected functions, it may also cause some differences in special case handling. A more robust approach is to selectively replace mathematical function calls by calls to intrinsic functions only where

it is merited by the performance gains and where changed properties such as reduced accuracy and different special case handling can be tolerated.

Table 8 Functions Affected by -use_fast_math

Operator/Function	Device Function
`x/y`	`__fdividef(x,y)`
`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)`

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 (to positive infinity) rounding mode.

Functions suffixed with **`_rd`** operate using the round down (to negative infinity) rounding mode.

Single-Precision Floating-Point Functions

`__fadd_[rn,rz,ru,rd]()` and **`__fmul_[rn,rz,ru,rd]()`** map to addition and multiplication operations that the compiler never merges into FMADs. By contrast, additions and multiplications generated from the '*' and '+' operators will frequently be combined into FMADs.

The accuracy of floating-point division varies depending on the compute capability of the device and whether the code is compiled with **`-prec-div=false`** or **`-prec-div=true`**. For devices of compute capability 2.x and higher when the code is compiled with **`-prec-div=false`** or for devices of compute capability 1.x, both the regular division **`/`** operator and **`__fdividef(x,y)`** have the same accuracy, but for $2^{126} < $ **`y`** $< 2^{128}$, **`__fdividef(x,y)`** delivers a result of zero, whereas the **`/`** operator delivers the correct result to within the accuracy stated in Table 9. Also, for $2^{126} < $ **`y`** $< 2^{128}$, if **`x`** is infinity, **`__fdividef(x,y)`** delivers a **`NaN`** (as a result of multiplying infinity by zero), while the **`/`** operator returns infinity. On the other hand, the **`/`** operator is IEEE-compliant on devices of compute capability 2.x and higher when the code is compiled

with **-prec-div=true** or without any **-prec-div** option at all since its default value is true.

Table 9 Single-Precision Floating-Point Intrinsic Functions

(Supported by the CUDA Runtime Library with Respective Error Bounds)

Function	Error bounds
__fadd_[rn,rz,ru,rd](x,y)	IEEE-compliant.
__fsub_[rn,rz,ru,rd](x,y)	IEEE-compliant.
__fmul_[rn,rz,ru,rd](x,y)	IEEE-compliant.
__fmaf_[rn,rz,ru,rd](x,y,z)	IEEE-compliant.
__frcp_[rn,rz,ru,rd](x)	IEEE-compliant.
__fsqrt_[rn,rz,ru,rd](x)	IEEE-compliant.
__frsqrt_rn(x)	IEEE-compliant.
__fdiv_[rn,rz,ru,rd](x,y)	IEEE-compliant.
__fdividef(x,y)	For **y** in $[2^{-126}, 2^{126}]$, the maximum ulp error is 2.
__expf(x)	The maximum ulp error is **2 + floor(abs(1.16 * x))**.
__exp10f(x)	The maximum ulp error is **2+ floor(abs(2.95 * x))**.
__logf(x)	For **x** in [0.5, 2], the maximum absolute error is $2^{-21.41}$, otherwise, the maximum ulp error is 3.
__log2f(x)	For **x** in [0.5, 2], the maximum absolute error is 2^{-22}, otherwise, the maximum ulp error is 2.
__log10f(x)	For **x** in [0.5, 2], the maximum absolute error is 2^{-24}, otherwise, the maximum ulp error is 3.
__sinf(x)	For **x** in [-π,π], the maximum absolute error is $2^{-21.41}$, and larger otherwise.
__cosf(x)	For **x** in [-π,π], the maximum absolute error is $2^{-21.19}$, and larger otherwise.
__sincosf(x,sptr,cptr)	Same as **__sinf(x)** and **__cosf(x)**.
__tanf(x)	Derived from its implementation as **__sinf(x) * (1/__cosf(x))**.

Function	Error bounds
`__powf(x, y)`	Derived from its implementation as `exp2f(y * __log2f(x))`.

Double-Precision Floating-Point Functions

`__dadd_rn()` and `__dmul_rn()` map to addition and multiplication operations that the compiler never merges into FMADs. By contrast, additions and multiplications generated from the '*' and '+' operators will frequently be combined into FMADs.

Table 10 Double-Precision Floating-Point Intrinsic Functions

(Supported by the CUDA Runtime Library with Respective Error Bounds)

Function	Error bounds
`__dadd_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__dsub_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__dmul_[rn,rz,ru,rd](x,y)`	IEEE-compliant.
`__fma_[rn,rz,ru,rd](x,y,z)`	IEEE-compliant.
`__ddiv_[rn,rz,ru,rd](x,y)(x,y)`	IEEE-compliant. Requires compute capability ≥ 2.
`__drcp_[rn,rz,ru,rd](x)`	IEEE-compliant. Requires compute capability ≥ 2.
`__dsqrt_[rn,rz,ru,rd](x)`	IEEE-compliant. Requires compute capability ≥ 2.

Appendix E.
C/C++ LANGUAGE SUPPORT

As described in Compilation with NVCC, source files compiled with **nvcc** can include a mix of host code and device code.

For the host code, **nvcc** supports whatever part of the C++ ISO/IEC 14882:2003 specification the host c++ compiler supports.

For the device code, **nvcc** supports the features illustrated in Code Samples with some restrictions described in Restrictions; it does not support run time type information (RTTI), exception handling, and the C++ Standard Library.

E.1. Code Samples

E.1.1. Data Aggregation Class

```
class PixelRGBA {
public:
    __device__ PixelRGBA(): r_(0), g_(0), b_(0), a_(0) { }

    __device__ PixelRGBA(unsigned char r, unsigned char g,
                         unsigned char b, unsigned char a = 255):
                         r_(r), g_(g), b_(b), a_(a) { }

private:
    unsigned char r_, g_, b_, a_;

    friend PixelRGBA operator+(const PixelRGBA const PixelRGBA&);
};

__device__
PixelRGBA operator+(const PixelRGBA& p1, const PixelRGBA& p2)
{
    return PixelRGBA(p1.r_ + p2.r_, p1.g_ + p2.g_,
                     p1.b_ + p2.b_, p1.a_ + p2.a_);
}

__device__ void func(void)
{
    PixelRGBA p1, p2;
    // ...      // Initialization of p1 and p2 here
    PixelRGBA p3 = p1 + p2;
}
```

E.1.2. Derived Class

```
__device__ void* operator new(size_t bytes, MemoryPool& p);
__device__ void operator delete(void*, MemoryPool& p);
class Shape {
public:
    __device__ Shape(void) { }
    __device__ void putThis(PrintBuffer *p) const;
    __device__ virtual void Draw(PrintBuffer *p) const {
        p->put("Shapeless");
    }
    __device__ virtual ~Shape() {}
};
class Point : public Shape {
public:
    __device__ Point() : x(0), y(0) {}
    __device__ Point(int ix, int iy) : x(ix), y(iy) { }
    __device__ void PutCoord(PrintBuffer *p) const;
    __device__ void Draw(PrintBuffer *p) const;
    __device__ ~Point() {}
private:
    int x, y;
};
__device__ Shape* GetPointObj(MemoryPool& pool)
{
    Shape* shape = new(pool) Point(rand(-20,10), rand(-100,-20));
    return shape;
}
```

E.1.3. Class Template

```
template <class T>
class myValues {
    T values[MAX_VALUES];
public:
    __device__ myValues(T clear) { ... }
    __device__ void setValue(int Idx, T value) { ... }
    __device__ void putToMemory(T* valueLocation) { ... }
};

template <class T>
void __global__ useValues(T* memoryBuffer) {
    myValues<T> myLocation(0);
    ...
}

__device__ void* buffer;

int main()
{
    ...
    useValues<int><<<blocks, threads>>>(buffer);
    ...
}
```

E.1.4. Function Template

```
template <typename T>
__device__ bool func(T x)
{
   ...
   return (...);
}

template <>
__device__ bool func<int>(T x) // Specialization
{
   return true;
}

// Explicit argument specification
bool result = func<double>(0.5);

// Implicit argument deduction
int x = 1;
bool result = func(x);
```

E.1.5. Functor Class

```
class Add {
public:
    __device__  float operator() (float a, float b) const
    {
        return a + b;
    }
};

class Sub {
public:
    __device__  float operator() (float a, float b) const
    {
        return a - b;
    }
};

// Device code
template<class O> __global__
void VectorOperation(const float * A, const float * B, float * C,
                     unsigned int N, O op)
{
    unsigned int iElement = blockDim.x * blockIdx.x + threadIdx.x;
    if (iElement < N)
        C[iElement] = op(A[iElement], B[iElement]);
}

// Host code
int main()
{
    ...
    VectorOperation<<<blocks, threads>>>(v1, v2, v3, N, Add());
    ...
}
```

E.2. Restrictions

E.2.1. Preprocessor Symbols

E.2.1.1. __CUDA_ARCH__

1. The type signature of the following entities shall not depend on whether **__CUDA_ARCH__** is defined or not, or on a particular value of **__CUDA_ARCH__**:
   - **__global__** functions and function templates
   - **__device__** and **__constant__** variables
   - textures and surfaces

   Example:

```
#if !defined(__CUDA_ARCH__)
typedef int mytype;
#else
typedef double mytype;
#endif

__device__ mytype xxx;          // error: xxx's type depends on __CUDA_ARCH__
__global__ void foo(mytype in, // error: foo's type depends on __CUDA_ARCH__
                    mytype *ptr)
{
  *ptr = in;
}
```

2. If a **__global__** function template is instantiated and launched from the host, then the function template must be instantiated with the same template arguments irrespective of whether **__CUDA_ARCH__** is defined and regardless of the value of **__CUDA_ARCH__**.

   Example:

```
__device__ int result;
template <typename T>
__global__ void kern(T in)
{
  result = in;
}

__host__ __device__ void foo(void)
{
#if !defined(__CUDA_ARCH__)
  kern<<<1,1>>>(1);      // error: "kern<int>" instantiation only
                         // when __CUDA_ARCH__ is undefined!
#endif
}

int main(void)
{
  foo();
  cudaDeviceSynchronize();
  return 0;
}
```

3. In separate compilation mode, the presence or absence of a definition of a function or variable with external linkage shall not depend on whether **__CUDA_ARCH__** is defined or on a particular value of **__CUDA_ARCH__** [6].

   Example:

```
#if !defined(__CUDA_ARCH__)
void foo(void) { }                  // error: The definition of foo()
                                    // is only present when __CUDA_ARCH__
                                    // is undefined
#endif
```

The compiler does not guarantee that a diagnostic will be generated for the unsupported uses of **__CUDA_ARCH__** described above.

E.2.2. Qualifiers

E.2.2.1. Device Memory Qualifiers

The **__device__**, **__shared__** and **__constant__** qualifiers are not allowed on:

- **class**, **struct**, and **union** data members,
- formal parameters,
- local variables within a function that executes on the host.

__shared__ and **__constant__** variables have implied static storage.

__device__ and **__constant__** variable definitions are only allowed in namespace scope (including global namespace scope).

__device__, **__constant__** and **__shared__** variables defined in namespace scope, that are of class type, cannot have a non-empty constructor or a non-empty destructor. A constructor for a class type is considered empty at a point in the translation unit, if it is either a trivial constructor or it satisfies all of the following conditions:

- The constructor function has been defined.
- The constructor function has no parameters, the initializer list is empty and the function body is an empty compound statement.
- Its class has no virtual functions and no virtual base classes.
- The default constructors of all base classes of its class can be considered empty.
- For all the nonstatic data members of its class that are of class type (or array thereof), the default constructors can be considered empty.

A destructor for a class is considered empty at a point in the translation unit, if it is either a trivial destructor or it satisfies all of the following conditions:

- The destructor function has been defined.
- The destructor function body is an empty compound statement.
- Its class has no virtual functions and no virtual base classes.
- The destructors of all base classes of its class can be considered empty.

[6] This does not apply to entities that may be defined in more than one translation unit, such as compiler generated template instantiations.

- For all the nonstatic data members of its class that are of class type (or array thereof), the destructor can be considered empty.

When compiling in the whole program compilation mode (see the nvcc user manual for a description of this mode), **__device__**, **__shared__**, and **__constant__** variables cannot be defined as external using the **extern** keyword. The only exception is for dynamically allocated **__shared__** variables as described in __shared__.

When compiling in the separate compilation mode (see the nvcc user manual for a description of this mode), **__device__**, **__shared__**, and **__constant__** variables can be defined as external using the **extern** keyword. **nvlink** will generate an error when it cannot find a definition for an external variable (unless it is a dynamically allocated **__shared__** variable).

E.2.2.2. __managed__ Qualifier

Variables marked with the **__managed__** qualifier ("managed" variables) have the following restrictions:

- The address of a managed variable is not a constant expression.
- A managed variable shall not have a const qualified type.
- A managed variable shall not have a reference type.
- 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 (e.g., a function marked with gcc's "attribute((destructor))").
  - In code that executes when CUDA runtime may not be initialized (e.g., a function marked with gcc's "attribute((constructor))").
- Managed variables have the same coherence and consistency behavior as specified for dynamically allocated managed memory.
- When a CUDA program containing managed variables is run on an execution platform with multiple GPUs, the variables are allocated only once, and not per GPU.

Here are examples of legal and illegal uses of managed variables:

```
__device__ __managed__ int xxx = 10;          // OK

int *ptr = &xxx;                              // error: use of managed variable
                                              // (xxx) in static initialization
struct S1_t {
  int field;
  S1_t(void) : field(xxx) { };
};
struct S2_t {
  ~S2_t(void) { xxx = 10; }
};

S1_t temp1;                                   // error: use of managed variable
                                              // (xxx) in dynamic initialization

S2_t temp2;                                   // error: use of managed variable
                                              // (xxx) in the destructor of
                                              // object with static storage
                                              // duration

__device__ __managed__ const int yyy = 10;   // error: const qualified type

__device__ __managed__ int &zzz = xxx;       // error: reference type

template <int *addr> struct S3_t { };
S3_t<&xxx> temp;                              // error: address of managed
                                              // variable(xxx) not a
                                              // constant expression

__global__ void kern(int *ptr)
{
  assert(ptr == &xxx);                        // OK
  xxx = 20;                                   // OK
}
int main(void)
{
  int *ptr = &xxx;                            // OK
  kern<<<1,1>>>(ptr);
  cudaDeviceSynchronize();
  xxx++;                                      // OK
}
```

E.2.2.3. Volatile Qualifier

Only after the execution of a **__threadfence_block()**, **__threadfence()**, or **__syncthreads()** (Memory Fence Functions and Synchronization Functions) are prior writes to global or shared memory guaranteed to be visible by other threads. As long as this requirement is met, the compiler is free to optimize reads and writes to global or shared memory.

This behavior can be changed using the **volatile** keyword: If a variable located in global or shared memory is declared as volatile, the compiler assumes that its value can be changed or used at any time by another thread and therefore any reference to this variable compiles to an actual memory read or write instruction.

E.2.3. Pointers

For devices of compute capability 1.x, pointers in code that is executed on the device are supported as long as the compiler is able to resolve whether they point to either the shared memory space, the global memory space, or the local memory space, otherwise

they are restricted to only point to memory allocated or declared in the global memory space. For devices of compute capability 2.x and higher, pointers are supported without any restriction.

Dereferencing a pointer either to global or shared memory in code that is executed on the host, or to host memory in code that is executed on the device results in an undefined behavior, most often in a segmentation fault and application termination.

The address obtained by taking the address of a **__device__**, **__shared__** or **__constant__** variable can only be used in device code. The address of a **__device__** or **__constant__** variable obtained through **cudaGetSymbolAddress()** as described in Device Memory can only be used in host code.

As a consequence of the use of C++ syntax rules, **void** pointers (e.g., returned by **malloc()**) cannot be assigned to non-**void** pointers without a typecast.

E.2.4. Operators

E.2.4.1. Assignment Operator

__constant__ variables can only be assigned from the host code through runtime functions (Device Memory); they cannot be assigned from the device code.

__shared__ variables cannot have an initialization as part of their declaration.

It is not allowed to assign values to any of the built-in variables defined in Built-in Variables.

E.2.4.2. Address Operator

It is not allowed to take the address of any of the built-in variables defined in Built-in Variables.

E.2.5. Functions

E.2.5.1. External Linkage

A call within some device code of a function declared with the extern qualifier is only allowed if the function is defined within the same compilation unit as the device code, i.e., a single file or several files linked together with relocatable device code and nvlink.

E.2.5.2. Compiler generated functions

The execution space qualifiers (**__host__**, **__device__**) for a compiler generated function are the union of the execution space qualifiers of all the functions that invoke it

(note that a **__global__** caller will be treated as a **__device__** caller for this analysis). For example:

```
class Base {
  int x;
public:
  __host__ __device__ Base(void) : x(10) {}
};

class Derived : public Base {
  int y;
};

class Other: public Base {
  int z;
};

__device__ void foo(void)
{
  Derived D1;
  Other D2;
}

__host__ void bar(void)
{
  Other D3;
}
```

Here, the compiler generated constructor function "Derived::Derived" will be treated as a **__device__** function, since it is invoked only from the **__device__** function "foo". The compiler generated constructor function "Other::Other" will be treated as a **__host__** **__device__** function, since it is invoked both from a **__device__** function "foo" and a **__host__** function "bar".

E.2.5.3. Function Parameters

__global__ function parameters are passed to the device:

- via shared memory and are limited to 256 bytes on devices of compute capability 1.x,
- via constant memory and are limited to 4 KB on devices of compute capability 2.x and higher.

__device__ and **__global__** functions cannot have a variable number of arguments.

E.2.5.4. Static Variables within Function

Within the body of a **__device__** or **__global__** function, only **__shared__** variables may be declared with static storage class.

E.2.5.5. Function Pointers

Function pointers to **__global__** functions are supported in host code, but not in device code.

Function pointers to **__device__** functions are only supported in device code compiled for devices of compute capability 2.x and higher.

It is not allowed to take the address of a **__device__** function in host code.

E.2.5.6. Function Recursion

__global__ functions do not support recursion.

__device__ functions only support recursion in device code compiled for devices of compute capability 2.x and higher.

E.2.6. Classes

E.2.6.1. Data Members

Static data members are not supported.

The layout of bit-fields in device code may currently not match the layout in host code on Windows.

E.2.6.2. Function Members

Static member functions cannot be **__global__** functions.

E.2.6.3. Virtual Functions

When a function in a derived class overrides a virtual function in a base class, the execution space qualifiers (i.e., **__host__**, **__device__**) on the overridden and overriding functions must match.

It is not allowed to pass as an argument to a **__global__** function an object of a class with virtual functions.

The virtual function table is placed in global or constant memory by the compiler.

E.2.6.4. Virtual Base Classes

It is not allowed to pass as an argument to a **__global__** function an object of a class derived from virtual base classes.

Use of virtual base class members is unsupported on devices of compute capability 1.x. For example, **nvcc** gives an error when the code below is compiled with **-arch compute_10**.

```
class Base {
public:
  int x;
};

class Derived : public virtual Base {
public:
  int y;
};

__global__ void k(Derived *d)
{
  int v = d->x; // Unsupported on compute capability 1.x.
  ...
}
```

Casting an object to its virtual base type is also unsupported on devices of compute capability 1.x. An example of such cases is shown below.

```
class Base { };

class Derived : public virtual Base { };

__device__ Base *get(Derived *x)
{
  return static_cast<Base *>(x); // Unsupported on compute capability 1.x
}
```

E.2.6.5. Anonymous Unions

Member variables of a namespace scope anonymous union cannot be referenced in a **__global__** or **__device__** function.

E.2.6.6. Windows-Specific

On Windows, the CUDA compiler may produce a different memory layout, compared to the host Microsoft compiler, for a C++ object of class type T that satisfies any of the following conditions:

- T has virtual functions or derives from a direct or indirect base class that has virtual functions;
- T has a direct or indirect virtual base class;
- T has multiple inheritance with more than one direct or indirect empty base class.

The size for such an object may also be different in host and device code. As long as type T is used exclusively in host or device code, the program should work correctly. Do not pass objects of type T between host and device code (e.g., as arguments to **__global__** functions or through **cudaMemcpy*()** calls).

E.2.7. Templates

A **__global__** function template cannot be instantiated with a type or typedef that is defined within a function or is private to a class or structure, as illustrated in the following code sample:

```
template <typename T>
__global__ void myKernel1(void) { }

template <typename T>
__global__ void myKernel2(T par) { }

class myClass {
private:
    struct inner_t { };
public:
    static void launch(void)
    {
       // Both kernel launches below are disallowed
       // as myKernel1 and myKernel2 are instantiated
       // with private type inner_t

       myKernel1<inner_t><<<1,1>>>();

       inner_t var;
       myKernel2<<<1,1>>>(var);
    }
};
```

Appendix F. TEXTURE FETCHING

This appendix gives the formula used to compute the value returned by the texture functions of Texture Functions depending on the various attributes of the texture reference (see Texture and Surface Memory).

The texture bound to the texture reference is represented as an array T of

- N texels for a one-dimensional texture,
- $N \times M$ texels for a two-dimensional texture,
- $N \times M \times L$ texels for a three-dimensional texture.

It is fetched using non-normalized texture coordinates x, y, and z, or the normalized texture coordinates x/N, y/M, and z/L as described in Texture Memory. In this appendix, the coordinates are assumed to be in the valid range. Texture Memory explained how out-of-range coordinates are remapped to the valid range based on the addressing mode.

F.1. Nearest-Point Sampling

In this filtering mode, the value returned by the texture fetch is

- $tex(x)=T[i]$ for a one-dimensional texture,
- $tex(x,y)=T[i,j]$ for a two-dimensional texture,
- $tex(x,y,z)=T[i,j,k]$ for a three-dimensional texture,

where $i=floor(x)$, $j=floor(y)$, and $k=floor(z)$.

Figure 13 illustrates nearest-point sampling for a one-dimensional texture with $N=4$.

For integer textures, the value returned by the texture fetch can be optionally remapped to [0.0, 1.0] (see Texture Memory).

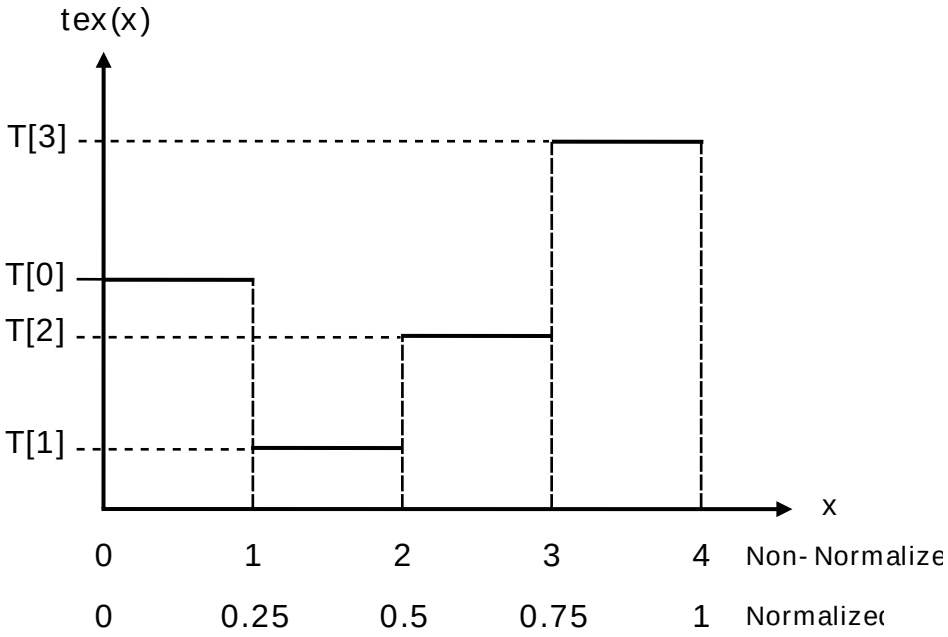


Figure 13 Nearest-Point Sampling Filtering Mode
Nearest-point sampling of a one-dimensional texture of four texels.

F.2. Linear Filtering

In this filtering mode, which is only available for floating-point textures, the value returned by the texture fetch is

- $tex(x)=(1-\alpha)T[i]+\alpha T[i+1]$ for a one-dimensional texture,
- $tex(x,y)=(1-\alpha)(1-\beta)T[i,j]+\alpha(1-\beta)T[i+1,j]+(1-\alpha)\beta T[i,j+1]+\alpha\beta T[i+1,j+1]$ for a two-dimensional texture,
- $tex(x,y,z) =$

  $(1-\alpha)(1-\beta)(1-\gamma)T[i,j,k]+\alpha(1-\beta)(1-\gamma)T[i+1,j,k]+$

  $(1-\alpha)\beta(1-\gamma)T[i,j+1,k]+\alpha\beta(1-\gamma)T[i+1,j+1,k]+$

  $(1-\alpha)(1-\beta)\gamma T[i,j,k+1]+\alpha(1-\beta)\gamma T[i+1,j,k+1]+$

  $(1-\alpha)\beta\gamma T[i,j+1,k+1]+\alpha\beta\gamma T[i+1,j+1,k+1]$

  for a three-dimensional texture,

where:

- $i=floor(x_B)$, $\alpha=frac(x_B)$, $x_B=x-0.5$,
- $j=floor(y_B)$, $\beta=frac(y_B)$, $y_B=y-0.5$,
- $k=floor(z_B)$, $\gamma=frac(z_B)$, $z_B= z-0.5$,

α, β, and γ are stored in 9-bit fixed point format with 8 bits of fractional value (so 1.0 is exactly represented).

Figure 14 illustrates linear filtering of a one-dimensional texture with $N=4$.

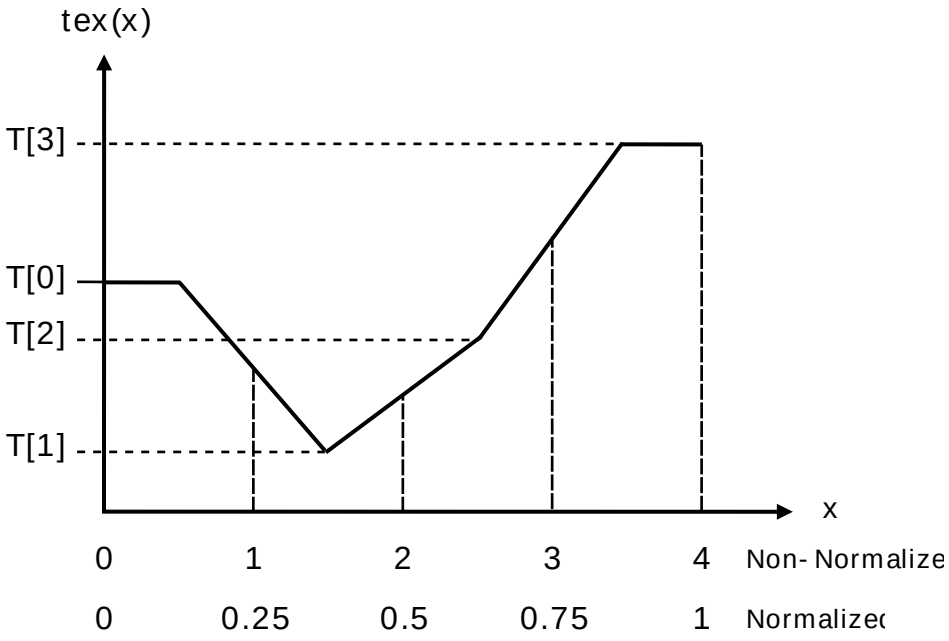


Figure 14 Linear Filtering Mode

Linear filtering of a one-dimensional texture of four texels in clamp addressing mode.

F.3. Table Lookup

A table lookup *TL(x)* where *x* spans the interval *[0,R]* can be implemented as *TL(x)=tex((N-1)/R)x+0.5)* in order to ensure that *TL(0)=T[0]* and *TL(R)=T[N-1]*.

Figure 15 illustrates the use of texture filtering to implement a table lookup with *R=4* or *R=1* from a one-dimensional texture with *N=4*.

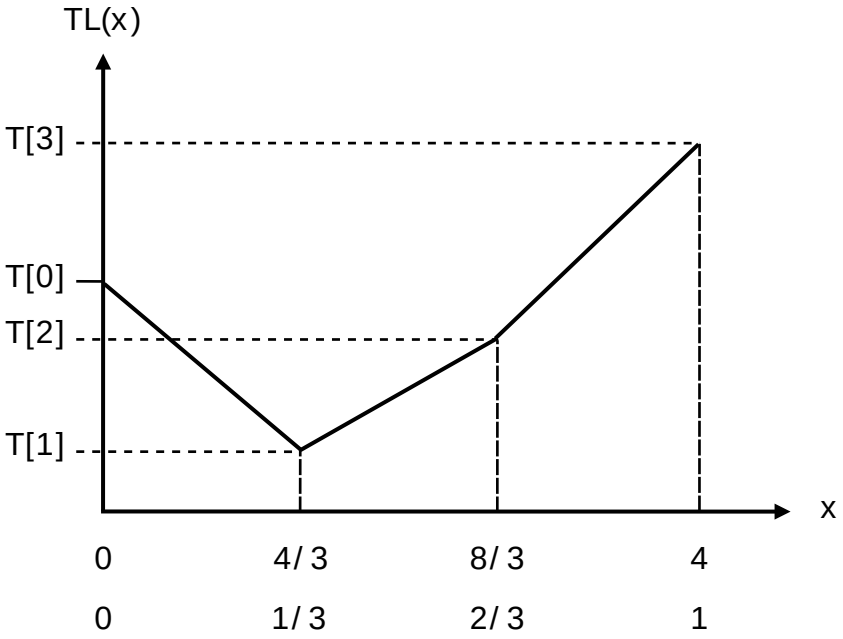


Figure 15 One-Dimensional Table Lookup Using Linear Filtering

Appendix G. COMPUTE CAPABILITIES

The general specifications and features of a compute device depend on its compute capability (see Compute Capability).

Table 11 gives the features and technical specifications associated to each compute capability.

Floating-Point Standard reviews the compliance with the IEEE floating-point standard.

Section Compute Capability 1.x, Compute Capability 2.x, Compute Capability 3.x, and Compute Capability 5.0 give more details on the architecture of devices of compute capability 1.x, 2.x, 3.x, and 5.0, respectively.

G.1. Features and Technical Specifications

Table 11 Feature Support per Compute Capability

<table>
<tr><th>Feature Support</th><th colspan="7">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.5, 5.0</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td rowspan="2">No</td><td colspan="6" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td><td colspan="2" rowspan="3">No</td><td colspan="5" rowspan="3">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td></tr>
</table>

Feature Support	Compute Capability						
(Unlisted features are supported for all compute capabilities)	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5, 5.0**
Warp vote functions (Warp Vote Functions)							
Double-precision floating-point numbers	No				Yes		
Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)	No				Yes		
Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())							
__ballot() (Warp Vote Functions)							
__threadfence_system() (Memory Fence Functions)							
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Synchronization Functions)							
Surface functions (Surface Functions)							
3D grid of thread blocks							
Unified Memory Programming	No					Yes	
Funnel shift (see reference manual)	No						Yes
Dynamic Parallelism							

Table 12 Technical Specifications per Compute Capability

	Compute Capability							
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**	**5.0**
Maximum dimensionality of grid of thread blocks	2				3			
Maximum x-dimension of a grid of thread blocks	65535					2^{31}-1		
Maximum y- or z-dimension of a grid of thread blocks	65535							
Maximum dimensionality of thread block	3							
Maximum x- or y-dimension of a block	512				1024			
Maximum z-dimension of a block	64							
Maximum number of threads per block	512				1024			
Warp size	32							

	Compute Capability							
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**	**5.0**
Maximum number of resident blocks per multiprocessor	8					16		32
Maximum number of resident warps per multiprocessor	24		32		48	64		
Maximum number of resident threads per multiprocessor	768		1024		1536	2048		
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K	64 K		
Maximum number of 32-bit registers per thread	128				63		255	
Maximum amount of shared memory per multiprocessor	16 KB				48 KB			64 KB
Maximum amount of shared memory per thread block	16 KB				48 KB			
Number of shared memory banks	16				32			
Amount of local memory per thread	16 KB				512 KB			
Constant memory size	64 KB							
Cache working set per multiprocessor for constant memory	8 KB							10 KB
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				12 KB		Between 12 KB and 48 KB	
Maximum width for a 1D texture reference bound to a CUDA array	8192				65536			
Maximum width for a 1D texture reference bound to linear memory	2^{27}							
Maximum width and number of layers for a 1D layered texture reference	8192 x 512				16384 x 2048			
Maximum width and height for a 2D texture reference bound to a CUDA array	65536 x 32768				65536 x 65535			
Maximum width and height for a 2D texture reference bound to linear memory	65000 x 65000				65000 x 65000			
Maximum width and height for a 2D texture reference bound to a CUDA array supporting texture gather	N/A				16384 x 16384			
Maximum width, height, and number of layers for a 2D layered texture reference	8192 x 8192 x 512				16384 x 16384 x 2048			
Maximum width, height, and depth for a 3D texture reference bound to a CUDA array	2048 x 2048 x 2048					4096 x 4096 x 4096		

	Compute Capability							
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**	**3.5**	**5.0**
Maximum width (and height) for a cubemap texture reference	N/A				16384			
Maximum width (and height) and number of layers for a cubemap layered texture reference	N/A				16384 x 2046			
Maximum number of textures that can be bound to a kernel	128					256		
Maximum width for a 1D surface reference bound to a CUDA array	N/A				65536			
Maximum width and number of layers for a 1D layered surface reference					65536 x 2048			
Maximum width and height for a 2D surface reference bound to a CUDA array					65536 x 32768			
Maximum width, height, and number of layers for a 2D layered surface reference					65536 x 32768 x 2048			
Maximum width, height, and depth for a 3D surface reference bound to a CUDA array					65536 x 32768 x 2048			
Maximum width (and height) for a cubemap surface reference bound to a CUDA array					32768			
Maximum width (and height) and number of layers for a cubemap layered surface reference					32768 x 2046			
Maximum number of surfaces that can be bound to a kernel					8	16		
Maximum number of instructions per kernel	2 million				512 million			

G.2. Floating-Point Standard

All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations:

- There is no dynamically configurable rounding mode; however, most of the operations support multiple IEEE rounding modes, exposed via device intrinsics;
- There is no mechanism for detecting that a floating-point exception has occurred and all operations behave as if the IEEE-754 exceptions are always masked, and deliver the masked response as defined by IEEE-754 if there is an exceptional event; for the same reason, while SNaN encodings are supported, they are not signaling and are handled as quiet;

- The result of a single-precision floating-point operation involving one or more input NaNs is the quiet NaN of bit pattern 0x7fffffff;
- Double-precision floating-point absolute value and negation are not compliant with IEEE-754 with respect to NaNs; these are passed through unchanged;
- For *single-precision* floating-point numbers on devices of *compute capability 1.x*:
  - Denormalized numbers are not supported; floating-point arithmetic and comparison instructions convert denormalized operands to zero prior to the floating-point operation;
  - Underflowed results are flushed to zero;
  - Some instructions are not IEEE-compliant:
    - Addition and multiplication are often combined into a single multiply-add instruction (FMAD), which truncates (i.e., without rounding) the intermediate mantissa of the multiplication;
    - Division is implemented via the reciprocal in a non-standard-compliant way;
    - Square root is implemented via the reciprocal square root in a non-standard-compliant way;
    - For addition and multiplication, only round-to-nearest-even and round-towards-zero are supported via static rounding modes; directed rounding towards +/- infinity is not supported.

    To mitigate the impact of these restrictions, IEEE-compliant software (and therefore slower) implementations are provided through the following intrinsics (c.f. Intrinsic Functions):
    - **`__fmaf_r{n,z,u,d}(float, float, float)`**: single-precision fused multiply-add with IEEE rounding modes,
    - **`__frcp_r[n,z,u,d](float)`**: single-precision reciprocal with IEEE rounding modes,
    - **`__fdiv_r[n,z,u,d](float, float)`**: single-precision division with IEEE rounding modes,
    - **`__fsqrt_r[n,z,u,d](float)`**: single-precision square root with IEEE rounding modes,
    - **`__fadd_r[u,d](float, float)`**: single-precision addition with IEEE directed rounding,
    - **`__fsub_r[u,d](float, float)`**: single-precision subtraction with IEEE directed rounding,
    - **`__fmul_r[u,d](float, float)`**: single-precision multiplication with IEEE directed rounding;
- For *double-precision* floating-point numbers on devices of *compute capability 1.x:*
  - Round-to-nearest-even is the only supported IEEE rounding mode for reciprocal, division, and square root.

When compiling for devices without native double-precision floating-point support, i.e., devices of compute capability 1.2 and lower, each **`double`** variable is converted to single-precision floating-point format (but retains its size of 64 bits) and double-precision floating-point arithmetic gets demoted to single-precision floating-point arithmetic.

For devices of compute capability 2.x and higher, code must be compiled with **-ftz=false**, **-prec-div=true**, and **-prec-sqrt=true** to ensure IEEE compliance (this is the default setting; see the **nvcc** user manual for description of these compilation flags); code compiled with **-ftz=true**, **-prec-div=false**, and **-prec-sqrt=false** comes closest to the code generated for devices of compute capability 1.x.

Addition and multiplication are often combined into a single multiply-add instruction:

- FMAD for single precision on devices of compute capability 1.x,
- FFMA for single precision on devices of compute capability 2.x and higher.

As mentioned above, FMAD truncates the mantissa prior to use it in the addition. FFMA, on the other hand, is an IEEE-754(2008) compliant fused multiply-add instruction, so the full-width product is being used in the addition and a single rounding occurs during generation of the final result. While FFMA in general has superior numerical properties compared to FMAD, the switch from FMAD to FFMA can cause slight changes in numeric results and can in rare circumstances lead to slightly larger error in final results.

In accordance to the IEEE-754R standard, if one of the input parameters to **fminf()**, **fmin()**, **fmaxf()**, or **fmax()** is NaN, but not the other, the result is the non-NaN parameter.

The conversion of a floating-point value to an integer value in the case where the floating-point value falls outside the range of the integer format is left undefined by IEEE-754. For compute devices, the behavior is to clamp to the end of the supported range. This is unlike the x86 architecture behavior.

The behavior of integer division by zero and integer overflow is left undefined by IEEE-754. For compute devices, there is no mechanism for detecting that such integer operation exceptions have occurred. Integer division by zero yields an unspecified, machine-specific value.

http://developer.nvidia.com/content/precision-performance-floating-point-and-ieee-754-compliance-nvidia-gpus includes more information on the floating point accuracy and compliance of NVIDIA GPUs.

G.3. Compute Capability 1.x

G.3.1. Architecture

For devices of compute capability 1.x, a multiprocessor consists of:

- 8 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
- 1 double-precision floating-point unit for double-precision floating-point arithmetic operations (this is only for devices of compute capability 1.3 and above),
- 2 special function units for single-precision floating-point transcendental functions (these units can also handle single-precision floating-point multiplications),
- 1 warp scheduler.

To execute an instruction for all threads of a warp, the warp scheduler must therefore issue the instruction over:

- 4 clock cycles for an integer or single-precision floating-point arithmetic instruction,
- 32 clock cycles for a double-precision floating-point arithmetic instruction (this is only for devices of compute capability 1.3 and above),
- 16 clock cycles for a single-precision floating-point transcendental instruction.

A multiprocessor also has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory.

Multiprocessors are grouped into *Texture Processor Clusters (TPCs)*. The number of multiprocessors per TPC is:

- 2 for devices of compute capabilities 1.0 and 1.1,
- 3 for devices of compute capabilities 1.2 and 1.3.

Each TPC has a read-only texture cache that is shared by all multiprocessors and speeds up reads from the texture memory space, which resides in device memory. Each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory.

The local and global memory spaces reside in device memory and are not cached.

G.3.2. Global Memory

A global memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. Devices of Compute Capability 1.0 and 1.1 and Devices of Compute Capability 1.2 and 1.3 describe how the memory accesses of threads within a half-warp are *coalesced* into one or more memory transactions depending on the compute capability of the device. Figure 16 shows some examples of global memory accesses and corresponding memory transactions based on compute capability.

The resulting memory transactions are serviced at the throughput of device memory.

Devices of Compute Capability 1.0 and 1.1

To coalesce, the memory request for a half-warp must satisfy the following conditions:

- The size of the words accessed by the threads must be 4, 8, or 16 bytes;
- If this size is:
  - 4, all 16 words must lie in the same 64-byte segment,
  - 8, all 16 words must lie in the same 128-byte segment,
  - 16, the first 8 words must lie in the same 128-byte segment and the last 8 words in the following 128-byte segment;
- Threads must access the words in sequence: The k^{th} thread in the half-warp must access the k^{th} word.

If the half-warp meets these requirements, a 64-byte memory transaction, a 128-byte memory transaction, or two 128-byte memory transactions are issued if the size of the

words accessed by the threads is 4, 8, or 16, respectively. Coalescing is achieved even if the warp is divergent, i.e., there are some inactive threads that do not actually access memory.

If the half-warp does not meet these requirements, 16 separate 32-byte memory transactions are issued.

Devices of Compute Capability 1.2 and 1.3

Threads can access any words in any order, including the same words, and a single memory transaction for each segment addressed by the half-warp is issued. This is in contrast with devices of compute capabilities 1.0 and 1.1 where threads need to access words in sequence and coalescing only happens if the half-warp addresses a single segment.

More precisely, the following protocol is used to determine the memory transactions necessary to service all threads in a half-warp:

- Find the memory segment that contains the address requested by the active thread with the lowest thread ID. The segment size depends on the size of the words accessed by the threads:
  - 32 bytes for 1-byte words,
  - 64 bytes for 2 byte words,
  - 128 bytes for 4-, 8- and 16-byte words.
- Find all other active threads whose requested address lies in the same segment.
- Reduce the transaction size, if possible:
  - If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
  - If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.
- Carry out the transaction and mark the serviced threads as inactive.
- Repeat until all threads in the half-warp are serviced.

G.3.3. Shared Memory

Shared memory has 16 banks that are organized such that successive 32-bit words map to successive banks. Each bank has a bandwidth of 32 bits per two clock cycles.

A shared memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. As a consequence, there can be no bank conflict between a thread belonging to the first half of a warp and a thread belonging to the second half of the same warp.

If a non-atomic instruction executed by a warp writes to the same location in shared memory for more than one of the threads of the warp, only one thread per half-warp performs a write and which thread performs the final write is undefined.

32-Bit Strided Access

A common access pattern is for each thread to access a 32-bit word from an array indexed by the thread ID **tid** and with some stride s:

```
extern __shared__ float shared[];
float data = shared[BaseIndex + s * tid];
```

In this case, threads **tid** and **tid+n** access the same bank whenever **s*n** is a multiple of the number of banks (i.e., 16) or, equivalently, whenever n is a multiple of **16/d** where **d** is the greatest common divisor of 16 and **s**. As a consequence, there will be no bank conflict only if half the warp size (i.e., 16) is less than or equal to **16/d**, i.e., only if **d** is equal to 1, i.e., **s** is odd.

Figure 17 shows some examples of strided access for devices of compute capability 3.x. The same examples apply for devices of compute capability 1.x, but with 16 banks instead of 32. Also, the access pattern for the example in the middle generates 2-way bank conflicts for devices of compute capability 1.x.

32-Bit Broadcast Access

Shared memory features a broadcast mechanism whereby a 32-bit word can be read and broadcast to several threads simultaneously when servicing one memory read request. This reduces the number of bank conflicts when several threads read from an address within the same 32-bit word. More precisely, a memory read request made of several addresses is serviced in several steps over time by servicing one conflict-free subset of these addresses per step until all addresses have been serviced; at each step, the subset is built from the remaining addresses that have yet to be serviced using the following procedure:

- Select one of the words pointed to by the remaining addresses as the broadcast word;
- Include in the subset:
  - All addresses that are within the broadcast word,
  - One address for each bank (other than the broadcasting bank) pointed to by the remaining addresses.

Which word is selected as the broadcast word and which address is picked up for each bank at each cycle are unspecified.

A common conflict-free case is when all threads of a half-warp read from an address within the same 32-bit word.

Figure 18 shows some examples of memory read accesses that involve the broadcast mechanism for devices of compute capability 3.x. The same examples apply for devices of compute capability 1.x, but with 16 banks instead of 32. Also, the access pattern for the example at the right generates 2-way bank conflicts for devices of compute capability 1.x.

8-Bit and 16-Bit Access

8-bit and 16-bit accesses typically generate bank conflicts. For example, there are bank conflicts if an array of **char** is accessed the following way:

```
extern __shared__ char shared[];
char data = shared[BaseIndex + tid];
```

because **shared[0]**, **shared[1]**, **shared[2]**, and **shared[3]**, for example, belong to the same bank. There are no bank conflicts however, if the same array is accessed the following way:

```
char data = shared[BaseIndex + 4 * tid];
```

Larger Than 32-Bit Access

Accesses that are larger than 32-bit per thread are split into 32-bit accesses that typically generate bank conflicts.

For example, there are 2-way bank conflicts for arrays of *doubles* accessed as follows:

```
extern __shared__ double shared[];
double data = shared[BaseIndex + tid];
```

as the memory request is compiled into two separate 32-bit requests with a stride of two. One way to avoid bank conflicts in this case is two split the double operands like in the following sample code:

```
__shared__ int shared_lo[32];
__shared__ int shared_hi[32];

double dataIn;
shared_lo[BaseIndex + tid] = __double2loint(dataIn);
shared_hi[BaseIndex + tid] = __double2hiint(dataIn);

double dataOut =
            __hiloint2double(shared_hi[BaseIndex + tid],
                             shared_lo[BaseIndex + tid]);
```

This might not always improve performance however and does perform worse on devices of compute capabilities 2.x and higher.

The same applies to structure assignments. The following code, for example:

```
extern __shared__ struct type shared[];
struct type data = shared[BaseIndex + tid];
```

results in:

- Three separate reads without bank conflicts if type is defined as

  ```
  struct type {
   float x, y, z;
  };
  ```

  since each member is accessed with an odd stride of three 32-bit words;
- Two separate reads with bank conflicts if type is defined as

  ```
  struct type {
      float x, y;
  };
  ```

  since each member is accessed with an even stride of two 32-bit words.

G.4. Compute Capability 2.x

G.4.1. Architecture

For devices of compute capability 2.x, a multiprocessor consists of:

- For devices of compute capability 2.0:
  - 32 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 4 special function units for single-precision floating-point transcendental functions,
- For devices of compute capability 2.1:
  - 48 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
  - 8 special function units for single-precision floating-point transcendental functions,
- 2 warp schedulers.

At every instruction issue time, each scheduler issues:

- One instruction for devices of compute capability 2.0,
- Two independent instructions for devices of compute capability 2.1,

for some warp that is ready to execute, if any. The first scheduler is in charge of the warps with an odd ID and the second scheduler is in charge of the warps with an even ID. Note that when a scheduler issues a double-precision floating-point instruction, the other scheduler cannot issue any instruction.

A warp scheduler can issue an instruction to only half of the CUDA cores. To execute an instruction for all threads of a warp, a warp scheduler must therefore issue the instruction over two clock cycles for an integer or floating-point arithmetic instruction.

A multiprocessor also has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory.

There is an L1 cache for each multiprocessor and an L2 cache shared by all multiprocessors, both of which are used to cache accesses to local or global memory, including temporary register spills. The cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load or store instruction.

The same on-chip memory is used for both L1 and shared memory: It can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache, using **cudaFuncSetCacheConfig()/cuFuncSetCacheConfig()**:

```
// Device code
__global__ void MyKernel(int* foo, int* bar, int a)
{
...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)

// Or via a function pointer:
void (*funcPtr)(int*, int*, int);
funcPtr = MyKernel;
cudaFuncSetCacheConfig(*funcPtr, cudaFuncCachePreferShared);
```

The default cache configuration is "prefer none," meaning "no preference." If a kernel is configured to have no preference, then it will default to the preference of the current thread/context, which is set using **cudaDeviceSetCacheConfig()/cuCtxSetCacheConfig()** (see the reference manual for details). If the current thread/context also has no preference (which is again the default setting), then whichever cache configuration was most recently used for any kernel will be the one that is used, unless a different cache configuration is required to launch the kernel (e.g., due to shared memory requirements). The initial configuration is 48 KB of shared memory and 16 KB of L1 cache.

Applications may query the L2 cache size by checking the **l2CacheSize** device property (see Device Enumeration). The maximum L2 cache size is 768 KB.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes four multiprocessors.

Each multiprocessor has a read-only texture cache to speed up reads from the texture memory space, which resides in device memory. It accesses the texture cache via a texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory.

G.4.2. Global Memory

Global memory accesses are cached. Using the **-dlcm** compilation flag, they can be configured at compile time to be cached in both L1 and L2 (**-Xptxas -dlcm=ca**) (this is the default setting) or in L2 only (**-Xptxas -dlcm=cg**).

A cache line is 128 bytes and maps to a 128 byte aligned segment in device memory. Memory accesses that are cached in both L1 and L2 are serviced with 128-byte memory transactions whereas memory accesses that are cached in L2 only are serviced with 32-byte memory transactions. Caching in L2 only can therefore reduce over-fetch, for example, in the case of scattered memory accesses.

If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently:

- Two memory requests, one for each half-warp, if the size is 8 bytes,
- Four memory requests, one for each quarter-warp, if the size is 16 bytes.

Each memory request is then broken down into cache line requests that are issued independently. A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Note that threads can access any words in any order, including the same words.

If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined.

Figure 16 shows some examples of global memory accesses and corresponding memory transactions based on compute capability.

G.4.3. Shared Memory

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 two clock cycles.

A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank): In that case, for read accesses, the word is broadcast to the requesting threads (and unlike for devices of compute capability 1.x, multiple words can be broadcast in a single transaction) and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).

This means, in particular, that unlike for devices of compute capability 1.x, there are no bank conflicts if an array of **`char`** is accessed as follows, for example:

```
extern __shared__ char shared[];
char data = shared[BaseIndex + tid];
```

Also, unlike for devices of compute capability 1.x, there may be bank conflicts between a thread belonging to the first half of a warp and a thread belonging to the second half of the same warp.

Figure 18 shows some examples of memory read accesses that involve the broadcast mechanism for devices of compute capability 3.x. The same examples apply for devices of compute capability 2.x.

32-Bit Strided Access

A common access pattern is for each thread to access a 32-bit word from an array indexed by the thread ID **`tid`** and with some stride s:

```
extern __shared__ float shared[];
float data = shared[BaseIndex + s * tid];
```

In this case, threads **`tid`** and **`tid+n`** access the same bank whenever **`s*n`** is a multiple of the number of banks (i.e., 32) or, equivalently, whenever **`n`** is a multiple of **`32/d`** where

d is the greatest common divisor of **32** and **s**. As a consequence, there will be no bank conflict only if the warp size (i.e., 32) is less than or equal to **32/d**, i.e., only if **d** is equal to 1, i.e., **s** is odd.

Figure 17 shows some examples of strided access for devices of compute capability 3.x. The same examples apply for devices of compute capability 2.x. However, the access pattern for the example in the middle generates 2-way bank conflicts for devices of compute capability 2.x.

Larger Than 32-Bit Access

64-bit and 128-bit accesses are specifically handled to minimize bank conflicts as described below.

Other accesses larger than 32-bit are split into 32-bit, 64-bit, or 128-bit accesses. The following code, for example:

```
struct type {
       float x, y, z;
};

extern __shared__ struct type shared[];
struct type data = shared[BaseIndex + tid];
```

results in three separate 32-bit reads without bank conflicts since each member is accessed with a stride of three 32-bit words.

64-Bit Accesses: For 64-bit accesses, a bank conflict only occurs if two threads in either of the half-warps access different addresses belonging to the same bank.

Unlike for devices of compute capability 1.x, there are no bank conflicts for arrays of **double**s accessed as follows, for example:

```
extern __shared__ double shared[];
double data = shared[BaseIndex + tid];
```

128-Bit Accesses: The majority of 128-bit accesses will cause 2-way bank conflicts, even if no two threads in a quarter-warp access different addresses belonging to the same bank. Therefore, to determine the ways of bank conflicts, one must add 1 to the maximum number of threads in a quarter-warp that access different addresses belonging to the same bank.

G.4.4. Constant Memory

In addition to the constant memory space supported by devices of all compute capabilities (where **__constant__ variables** reside), devices of compute capability 2.x support the **LDU** (LoaD Uniform) instruction that the compiler uses to load any variable that is:

- pointing to global memory,
- read-only in the kernel (programmer can enforce this using the const keyword),
- not dependent on thread ID.

G.5. Compute Capability 3.x

G.5.1. Architecture

A multiprocessor consists of:

- 192 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
- 32 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

When a multiprocessor is given warps to execute, it first distributes them among the four schedulers. Then, at every instruction issue time, each scheduler issues two independent instructions for one of its assigned warps that is ready to execute, if any.

A multiprocessor has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory.

There is an L1 cache for each multiprocessor and an L2 cache shared by all multiprocessors. The L1 cache is used to cache accessess to local memory, including temporary register spills. The L2 cache is used to cache accesses to local and global memory. The cache behavior (e.g., whether reads are cached in both L1 and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load or store instruction.

The same on-chip memory is used for both L1 and shared memory: It can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache or as 32 KB of shared memory and 32 KB of L1 cache, using **`cudaFuncSetCacheConfig()`**/**`cuFuncSetCacheConfig()`**:

```
// Device code
__global__ void MyKernel()
{
    ...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 KB
// cudaFuncCachePreferEqual: shared memory is 32 KB
// cudaFuncCachePreferL1: shared memory is 16 KB
// cudaFuncCachePreferNone: no preference
cudaFuncSetCacheConfig(MyKernel, cudaFuncCachePreferShared)
```

The default cache configuration is "prefer none," meaning "no preference." If a kernel is configured to have no preference, then it will default to the preference of the current thread/context, which is set using **`cudaDeviceSetCacheConfig()`**/**`cuCtxSetCacheConfig()`** (see the reference manual for details). If the current thread/context also has no preference (which is again the default setting), then whichever cache configuration was most recently used for any kernel will be the one that is used, unless a different cache configuration is required to

launch the kernel (e.g., due to shared memory requirements). The initial configuration is 48 KB of shared memory and 16 KB of L1 cache.

Applications may query the L2 cache size by checking the l2CacheSize device property (see Device Enumeration). The maximum L2 cache size is 1.5 MB.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes three multiprocessors.

Each multiprocessor has a read-only data cache of 48 KB to speed up reads from device memory. It accesses this cache either directly (for devices of compute capability 3.5 only), or via a texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory. When accessed via the texture unit, the read-only data cache is also referred to as texture cache.

G.5.2. Global Memory

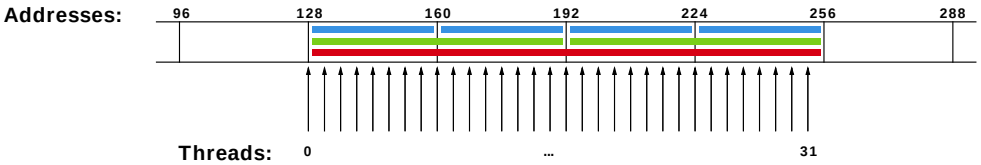
Global memory accesses for devices of compute capability 3.x are cached in L2 and for devices of compute capability 3.5, may also be cached in the read-only data cache described in the previous section; they are not cached in L1.

Caching in L2 behaves in the same way as for devices of compute capability 2.x (see Global Memory).

A global memory read performed using the **`__ldg()`** function (see Read-Only Data Cache Load Function) is always cached in the read-only data cache. When applicable, the compiler will also compile any regular global memory read to **`__ldg()`**. A requirement for data to be cached in the read-only cache is that it must be read-only for the entire lifetime of the kernel. In order to make it easier for the compiler to detect that this condition is satisfied, pointers used for loading such data should be marked with both the **`const`** and **`__restrict__`** qualifiers.

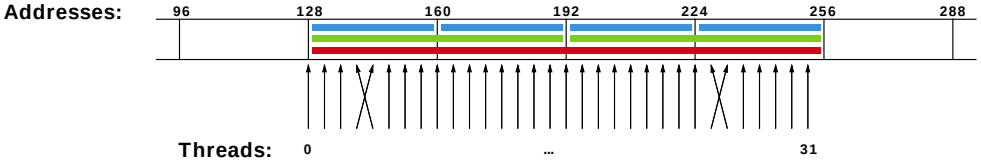
Figure 16 shows some examples of global memory accesses and corresponding memory transactions based on compute capability.

Aligned and sequential



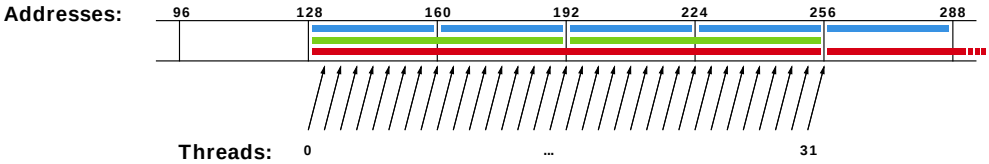
Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	1x 64B at 128 1x 64B at 192	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Aligned and non- sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	8x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224	1x 64B at 128 1x 64B at 192	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224	1x 128B at 128

Mis- aligned and sequential



Compute capability:	1.0 and 1.1	1.2 and 1.3	2.x and 3.x	
Memory transactions:	Uncached		Uncached	Cached
	7x 32B at 128 8x 32B at 160 8x 32B at 192 8x 32B at 224 1x 32B at 256	1x 128B at 128 1x 64B at 192 1x 32B at 256	1x 32B at 128 1x 32B at 160 1x 32B at 192 1x 32B at 224 1x 32B at 256	1x 128B at 128 1x 128B at 256

Figure 16 Examples of Global Memory Accesses

Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

G.5.3. Shared Memory

Shared memory has 32 banks with two addressing modes that are described below.

The addressing mode can be queried using **`cudaDeviceGetSharedMemConfig()`** and set using **`cudaDeviceSetSharedMemConfig()`** (see reference manual for more details). Each bank has a bandwidth of 64 bits per clock cycle.

Figure 17 shows some examples of strided access.

Figure 18 shows some examples of memory read accesses that involve the broadcast mechanism.

64-Bit Mode

Successive 64-bit words map to successive banks.

A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 64-bit word (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 64-bit word is broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined).

In this mode, the same access pattern generates fewer bank conflicts than on devices of compute capability 2.x for 64-bit accesses and as many or fewer for 32-bit accesses.

32-Bit Mode

Successive 32-bit words map to successive banks.

A shared memory request for a warp does not generate a bank conflict between two threads that access any sub-word within the same 32-bit word or within two 32-bit words whose indices i and j are in the same 64-word aligned segment (i.e., a segment whose first index is a multiple of 64) and such that $j=i+32$ (even though the addresses of the two sub-words fall in the same bank): In that case, for read accesses, the 32-bit words are broadcast to the requesting threads and for write accesses, each sub-word is written by only one of the threads (which thread performs the write is undefined).

In this mode, the same access pattern generates as many or fewer bank conflicts than on devices of compute capability 2.x.

G.6. Compute Capability 5.0

G.6.1. Architecture

A multiprocessor consists of:

- 128 CUDA cores for arithmetic operations (see Arithmetic Instructions for throughputs of arithmetic operations),
- 32 special function units for single-precision floating-point transcendental functions,
- 4 warp schedulers.

When a multiprocessor is given warps to execute, it first distributes them among the four schedulers. Then, at every instruction issue time, each scheduler issues one instruction for one of its assigned warps that is ready to execute, if any.

A multiprocessor has:

- a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory,
- a read-only data cache of 24 KB used to cache reads from global memory,
- 64 KB of shared memory.

The read-only data cache is also used by the texture unit that implements the various addressing modes and data filtering mentioned in Texture and Surface Memory. When accessed via the texture unit, it is also referred to as texture cache.

There is also an L2 cache shared by all multiprocessors that is used to cache accesses to local or global memory, including temporary register spills. Applications may query the L2 cache size by checking the l2CacheSize device property (see Device Enumeration).

The cache behavior (e.g., whether reads are cached in both the data cache and L2 or in L2 only) can be partially configured on a per-access basis using modifiers to the load instruction.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes four multiprocessors.

G.6.2. Global Memory

Global memory accesses are cached in L2 and may also be cached in the read-only data cache described in the previous section.

A global memory read performed using the **`__ldg()`** function (see Read-Only Data Cache Load Function) is always cached in the read-only data cache. When applicable, the compiler will also compile any regular global memory read to **`__ldg()`**. A requirement for data to be cached in the read-only cache is that it must be read-only for the entire lifetime of the kernel. In order to make it easier for the compiler to detect that this condition is satisfied, pointers used for loading such data should be marked with both the **`const`** and **`__restrict__`** qualifiers.

Caching in L2 behaves in the same way as for devices of compute capability 2.x (see Global Memory).

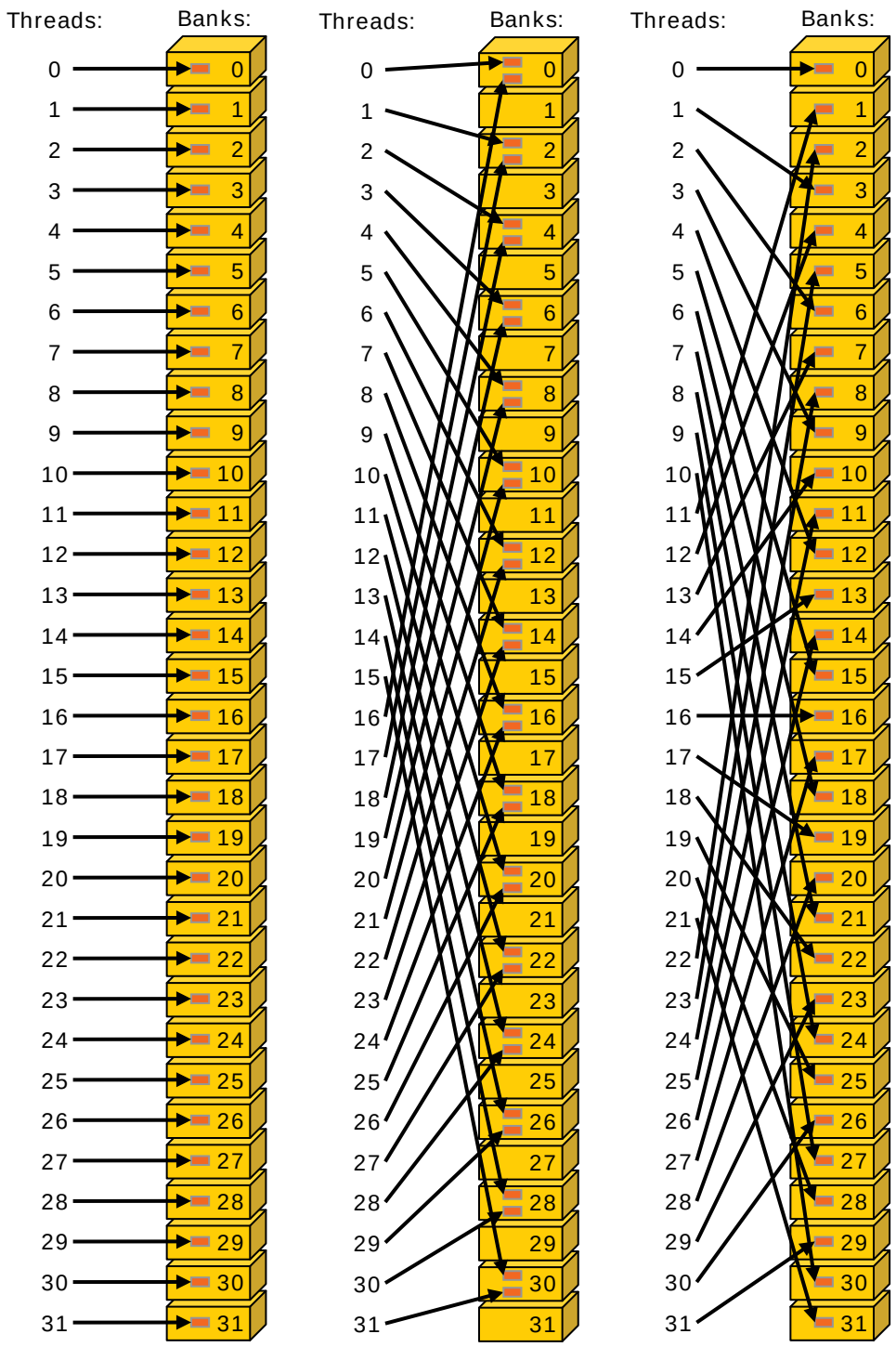
G.6.3. Shared Memory

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.

A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank): In that case, for read accesses, the word is broadcast to the requesting threads and for write accesses, each address is written by only one of the threads (which thread performs the write is undefined).

Figure 17 shows some examples of strided access.

Figure 18 shows some examples of memory read accesses that involve the broadcast mechanism.



Left

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

Middle

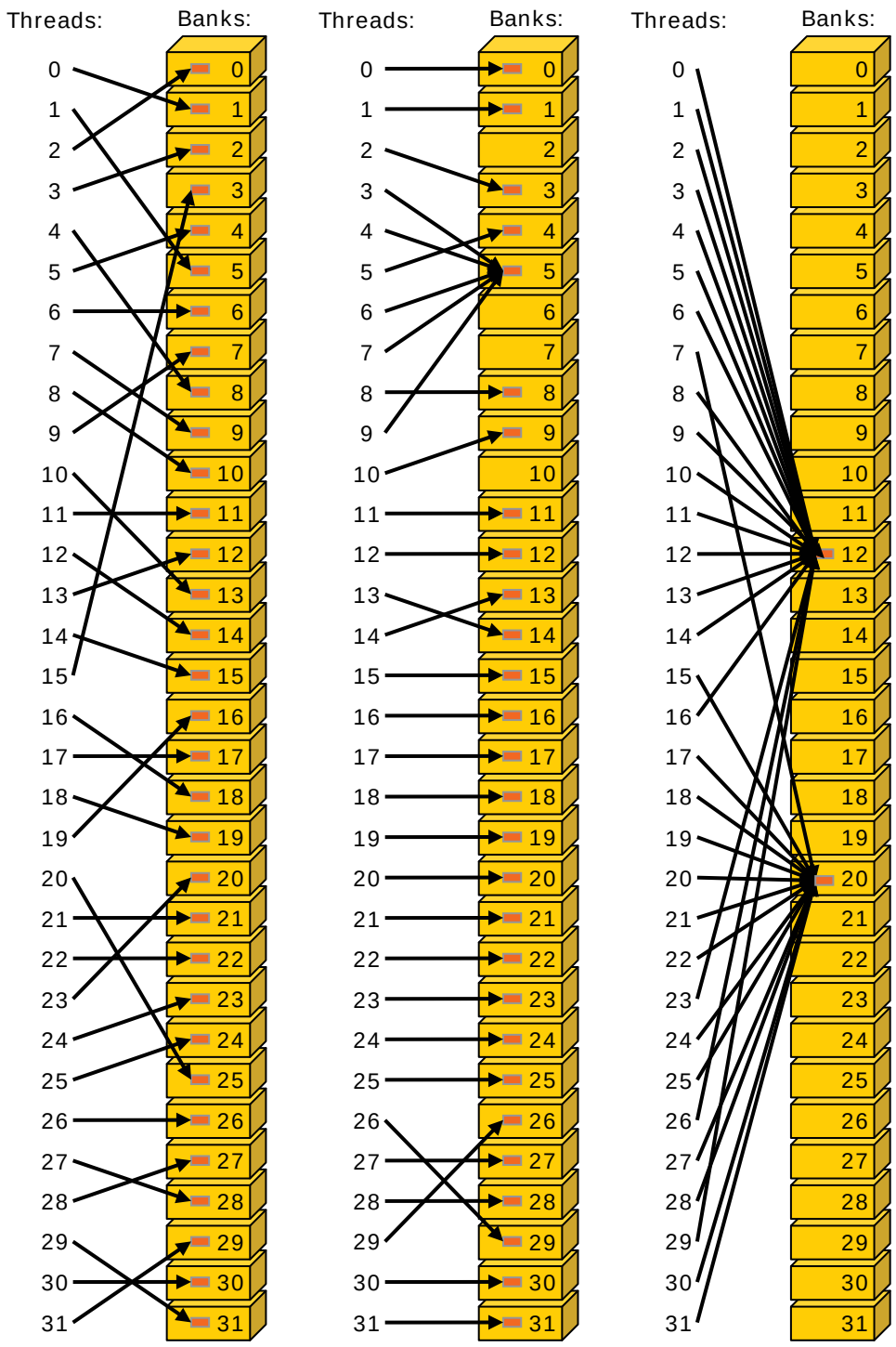
Linear addressing with a stride of two 32-bit words (no bank conflict).

Right

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

Figure 17 Strided Shared Memory Accesses

Examples for devices of compute capability 3.x (in 32-bit mode) or compute capability 5.0.



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).

Figure 18 Irregular Shared Memory Accesses

Examples for devices of compute capability 3.x or 5.0.

Appendix H. DRIVER API

This appendix assumes knowledge of the concepts described in CUDA C Runtime.

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 13.

Table 13 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 reference	CUtexref	Object that describes how to interpret texture memory data
Surface reference	CUsurfref	Object that describes how to read or write CUDA arrays
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
    ...

    // 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.

H.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()`**.

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 Figure 19.

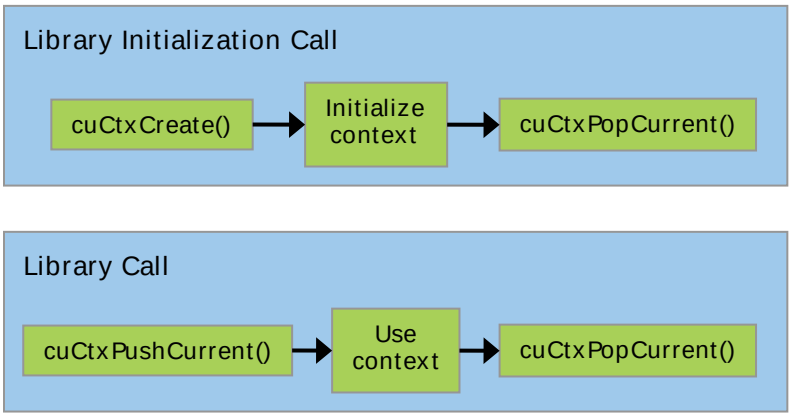


Figure 19 Library Context Management

H.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;
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);
```

Full code can be found in the **ptxjit** CUDA sample.

H.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 3. 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;
```

H.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 pick up this context instead of creating a new one.

If the runtime is initialized (implicitly as mentioned in CUDA C Runtime), **cuCtxGetCurrent()** can be used to retrieve the context created during initialization. This context can be used by subsequent driver API calls.

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.

Appendix I.
CUDA ENVIRONMENT VARIABLES

Environment variables related to the Multi-Process Service are documented in the Multi-Process Service section of the GPU Deployment and Management guide.

Table 14 CUDA Environment Variables

Category	Variable	Values	Description
Device Enumeration and Properties	CUDA_VISIBLE_DEVICES	A comma-separated sequence of integers	Only the devices whose index is present in the sequence are visible to CUDA applications and they are enumerated in the order of the sequence. If one of the indices is invalid, only the devices whose index precedes the invalid index are visible to CUDA applications. For example, setting CUDA_VISIBLE_DEVICES to 2,1 causes device 0 to be invisible and device 2 to be enumerated before device 1. Setting CUDA_VISIBLE_DEVICES to 0,2,-1,1 causes devices 0 and 2 to be visible and device 1 to be invisible.
	CUDA_MANAGED_FORCE_DEVICE_ALLOC	0 or 1 (default is 0)	Forces the driver to place all managed allocations in device memory.
Compilation	CUDA_CACHE_DISABLE	0 or 1 (default is 0)	Disables caching (when set to 1) or enables caching (when set to 0) for just-in-time-compilation. When

Category	Variable	Values	Description
			disabled, no binary code is added to or retrieved from the cache.
	CUDA_CACHE_PATH	filepath	Specifies the folder where the just-in-time compiler caches binary codes; the default values are: ▸ on Windows, **%APPDATA%\NVIDIA\ComputeCache**, ▸ on MacOS, **$HOME/Library/Application\ Support/NVIDIA/ComputeCache**, ▸ on Linux, **~/.nv/ComputeCache**
	CUDA_CACHE_MAXSIZE	integer (default is 33554432 (32 MB) and maximum is 4294967296 (4 GB))	Specifies the size in bytes of the cache used by the just-in-time compiler. Binary codes whose size exceeds the cache size are not cached. Older binary codes are evicted from the cache to make room for newer binary codes if needed.
	CUDA_FORCE_PTX_JIT	0 or 1 (default is 0)	When set to 1, forces the device driver to ignore any binary code embedded in an application (see Application Compatibility) and to just-in-time compile embedded *PTX* code instead. If a kernel does not have embedded *PTX* code, it will fail to load. This environment variable can be used to validate that *PTX* code is embedded in an application and that its just-in-time compilation works as expected to guarantee application forward compatibility with future architectures

Category	Variable	Values	Description
			(see Just-in-Time Compilation).
Execution	CUDA_LAUNCH_BLOCKING	0 or 1 (default is 0)	Disables (when set to 1) or enables (when set to 0) asynchronous kernel launches.
	CUDA_DEVICE_MAX_CONNECTIONS	1 to 32 (default is 8)	Sets the number of compute and copy engine concurrent connections (work queues) from the host to each device of compute capability 3.5 and above.
cuda-gdb (on Mac and Linux platforms)	CUDA_DEVICE_WAITS_ON_EXCEPTION	0 or 1 (default is 0)	When set to 1, a CUDA application will halt when a device exception occurs, allowing a debugger to be attached for further debugging.
Driver-Based Profiler (these variables have no impact on the Visual Profiler or the command line profiler nvprof)	CUDA_DEVICE	Integer (default is 0)	Specifies the index of the device to profile.
	COMPUTE_PROFILE	0 or 1 (default is 0)	Disables profiling (when set to 0) or enables profiling (when set to 1).
	COMPUTE_PROFILE_CONFIG	Path	Specifies the configuration file to set profiling options and select performance counters.
	COMPUTE_PROFILE_LOG	Path	Specifies the file used to save the profiling output. In case of multiple contexts, use '%d' in the COMPUTE_PROFILE_LOG to generate separate output files for each context - with '%d' substituted by the context number.
	COMPUTE_PROFILE_CSV	0 or 1 (default is 0)	When set to 1, the output will be in comma-separated format.

Appendix J.
UNIFIED MEMORY PROGRAMMING

J.1. Unified Memory Introduction

Unified Memory is a component of the CUDA programming model, first introduced in CUDA 6.0, that defines a new *managed* memory space in which all processors see a single coherent memory image with a common address space.

A *processor* refers to any independent execution unit with a dedicated MMU. This includes both CPUs and GPUs of any type and architecture.

The underlying system manages data access and locality within a CUDA program without need for explicit memory copy calls. This benefits GPU programming in two primary ways:

- GPU programming is simplified by unifying memory spaces coherently across all GPUs and CPUs in the system and by providing tighter and more straightforward language integration for CUDA programmers.
- Data access speed is maximized by transparently migrating data towards the processor using it.

In simple terms, Unified Memory eliminates the need for explicit data movement via the **`cudaMemcpy*()`** routines without the performance penalty incurred by placing all data into zero-copy memory. Data movement, of course, still takes place, so a program's run time typically does not decrease; Unified Memory instead enables the writing of simpler and more maintainable code.

Unified Memory offers a "single-pointer-to-data" model that is conceptually similar to CUDA's zero-copy memory. One key difference between the two is that with zero-copy allocations the physical location of memory is pinned in CPU system memory such that a program may have fast or slow access to it depending on where it is being accessed from. Unified Memory, on the other hand, decouples memory and execution spaces so that all data accesses are fast.

The term *Unified Memory* describes as a system that provides memory management services to a wide range of programs, from those targeting the Runtime API down to

those using the Virtual ISA (PTX). Part of this system defines the managed memory space that opts in to Unified Memory services.

Managed memory is interoperable and interchangeable with device-specific allocations, such as those created using the **cudaMalloc()** routine. All CUDA operations that are valid on device memory are also valid on managed memory; the primary difference is that the host portion of a program is able to reference and access the memory as well.

J.1.1. Simplifying GPU Programming

Unification of memory spaces means that there is no longer any need for explicit memory transfers between host and device. Any allocation created in the managed memory space is automatically migrated to where it is needed.

A program allocates managed memory in one of two ways: via the new **cudaMallocManaged()** routine, which is semantically similar to **cudaMalloc()**; or by defining a global **__managed__** variable, which is semantically similar to a **__device__** variable. Precise definitions of these are found later in this document.

The following code examples illustrate how the use of managed memory can change the way in which host code is written. First, a simple program written without the benefit of unified memory:

```
__global__  void  AplusB( int  *ret,  int  a,  int  b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    int *ret;
    cudaMalloc(&ret, 1000 * sizeof(int));

    AplusB<<< 1, 1000 >>>(ret, 10, 100);

    int *host_ret = (int *)malloc(1000 * sizeof(int));
    cudaMemcpy(host_ret, ret, 1000 * sizeof(int), cudaMemcpyDefault);

    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, host_ret[i]);

    free(host_ret);
    cudaFree(ret);
 return  0;
}
```

This first example combines two numbers together on the GPU with a per-thread ID and returns the values in an array. Without managed memory, both host- and device-side storage for the return values is required (**host_ret** and **ret** in the example), as is an explicit copy between the two using **cudaMemcpy()**.

Compare this with the Unified Memory version of the program, which allows direct access of GPU data from the host. Notice the new **cudaMallocManaged()** routine, which returns a pointer valid from both host and device code. This allows **ret** to be

used without a separate **host_ret** copy, greatly simplifying and reducing the size of the program.

```
__global__ void AplusB(int *ret, int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    int *ret;
    cudaMallocManaged(&ret, 1000 * sizeof(int));

    AplusB<<< 1, 1000 >>>(ret, 10, 100);
        cudaDeviceSynchronize();

    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);

    cudaFree(ret);
 return  0;
}
```

Finally, language integration allows direct reference of a GPU-declared **__managed__** variable and simplifies a program further when global variables are used.

```
__device__ __managed__  int  ret[1000];
__global__ void AplusB(int a, int b) {
    ret[threadIdx.x] = a + b + threadIdx.x;
}

int main() {
    AplusB<<< 1, 1000 >>>(10, 100);
    cudaDeviceSynchronize();

    for(int i=0; i<1000; i++)
        printf("%d: A+B = %d\n", i, ret[i]);
 return  0;
}
```

Note the absence of explicit **cudaMemcpy()** commands and the fact that the return array **ret** is visible on both CPU and GPU.

It is worth a comment on the synchronization between host and device. Notice how in the non-managed example, the synchronous **cudaMemcpy()** routine is used both to synchronize the kernel (that is, to wait for it to finish running), and to transfer the data to the host. The Unified Memory examples do not call **cudaMemcpy()** and so require an explicit **cudaDeviceSynchronize()** before the host program can safely use the output from the GPU.

An alternative here would be to set the environment variable **CUDA_LAUNCH_BLOCKING=1**, ensuring that all kernel launches complete synchronously. This simplifies the code by eliminating all explicit synchronization, but obviously has broader impact on execution behavior as a whole.

J.1.2. Data Migration and Coherency

Unified Memory attempts to optimize memory performance by migrating data towards the device where it is being accessed (that is, moving data to host memory if the CPU is accessing it and to device memory if the GPU will access it). Data migration is fundamental to Unified Memory, but is transparent to a program. The system will try

to place data in the location where it can most efficiently be accessed without violating coherency.

The physical location of data is invisible to a program and may be changed at any time, but accesses to the data's virtual address will remain valid and coherent from any processor regardless of locality. Note that maintaining coherence is the primary requirement, ahead of performance; within the constraints of the host operating system, the system is permitted to either fail accesses or move data in order to maintain global coherence between processors.

J.1.3. Multi-GPU Support

Multi-GPU systems are able to use managed memory, but data does not migrate between GPUs. Managed memory allocation behaves identically to unmanaged memory allocated using **`cudaMalloc()`**: the current active device is the home for the physical allocation, and all other GPUs receive peer mappings to the memory. This means that other GPUs in the system will access the memory at reduced bandwidth over the PCIe bus.

If peer mappings are not supported between the GPUs in the system, then the managed memory pages are placed in CPU system memory ("zero-copy" memory), and all GPUs will experience PCIe bandwidth restrictions. See Managed Memory with Multi-GPU Programs for details.

J.1.4. System Requirements

Unified Memory has three basic requirements:

- a GPU with SM architecture 3.0 or higher (Kepler class or newer)
- a 64-bit host application and operating system, except on Android
- Linux or Windows

J.2. Programming Model

J.2.1. Managed Memory Opt In

CUDA requires a program to opt in to automatic data management by either annotating a **`__device__ variable`** with the **`new __managed__ keyword`** (see the Language Integration section) or by using a new **`cudaMallocManaged()`** call to allocate data.

Managed memory must always be allocated on the heap, either with an allocator or by declaring global storage. It is not possible either to associate previously allocated memory with Unified Memory, or to have the Unified Memory system manage a CPU or a GPU stack pointer.

J.2.1.1. Explicit Allocation Using `cudaMallocManaged()`

Unified memory is most commonly created using an allocation function that is semantically and syntactically similar to the standard CUDA allocator, **`cudaMalloc()`**. The function description is as follows:

```
    cudaError_t cudaMallocManaged(void **devPtr,
                                  size_t size,
                                  unsigned int flags=0);
```

The **`cudaMallocManaged()`** function allocates **`size`** bytes of managed memory on the GPU and returns a pointer in **`devPtr`**. The pointer is valid on all GPUs and the CPU in the system, although program accesses to this pointer must obey the concurrency rules of the Unified Memory programming model (see Coherency and Concurrency). Below is a simple example, showing the use of **`cudaMallocManaged()`**:

```
__global__ void printme(char *str) {
    printf(str);
}

int main() {
    // Allocate 100 bytes of memory, accessible to both Host and Device code
    char *s;
    cudaMallocManaged(&s, 100);

    // Note direct Host-code use of "s"
    strncpy(s, "Hello Unified Memory\n", 99);

    // Here we pass "s" to a kernel without explicitly copying
    printme<<< 1, 1 >>>(s);
    cudaDeviceSynchronize();

    // Free as for normal CUDA allocations
    cudaFree(s);
 return  0;
}
```

A program's behavior is functionally unchanged when **`cudaMalloc()`** is replaced with **`cudaMallocManaged()`**; however, the program should go on to eliminate explicit memory copies and take advantage of automatic migration. Additionally, dual pointers (one to host and one to device memory) can be eliminated.

In CUDA 6.0, device code is not able to call **`cudaMallocManaged()`**. All managed memory must be allocated from the host or at global scope (see the next section). Allocations on the device heap using **`malloc()`** in a kernel will not be created in the managed memory space, and so will not be accessible to CPU code.

J.2.1.2. Global-Scope Managed Variables Using __managed__

File-scope and global-scope CUDA **__device__** variables may also opt-in to Unified Memory management by adding a new **__managed__** annotation to the declaration. These may then be referenced directly from either host or device code, as follows:

```
__device__ __managed__ int x[2];
__device__ __managed__ int y;

__global__ void kernel() {
    x[1] = x[0] + y;
}

int main() {
    x[0] = 3;
    y = 5;

    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();

    printf("result = %d\n", x[1]);
 return  0;
}
```

All semantics of the original **__device__** memory space, along with some additional unified-memory-specific constraints, are inherited by the managed variable. See Compilation with NVCC) in the *CUDA C Programming Guide* for details.

Note that variables marked **__constant__** may not also be marked as **__managed__**; this annotation is reserved for **__device__** variables only. Constant memory must be set either statically at compile time or by using **cudaMemcpyToSymbol()** as usual in CUDA.

J.2.2. Coherency and Concurrency

J.2.2.1. GPU Exclusive Access To Managed Memory

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 while any kernel operation is executing, regardless of whether the specific kernel is actively using the data. When managed data is used with **cudaMemcpy*()** or **cudaMemset*()**, the system may choose to access the source or destination from the host or the device, which will put constraints on concurrent CPU access to that data while the **cudaMemcpy*()** or **cudaMemset*()** is executing. See Memcpy()/Memset() Behavior With Managed Memory for further details.

In general, it is not permitted for the CPU to access any managed allocations or variables while the GPU is active. Concurrent CPU/GPU accesses, even to different managed

memory allocations, will cause a segmentation fault because the page is considered inaccessible to the CPU.

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}

int main() {
    kernel<<< 1, 1 >>>();

    y = 20;   // ERROR: CPU access concurrent with GPU
    cudaDeviceSynchronize();

 return  0;
}
```

In example above, the GPU program **kernel** is still active when the CPU touches **y**. (Note how it occurs before **cudaDeviceSynchronize()**.) This access is invalid even though the CPU is accessing different data than the GPU. The program must explicitly synchronize with the GPU before accessing **y**:

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}

int main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();

    y = 20;   //  Success - GPU is idle so access is OK
 return  0;
}
```

As this example shows, a CPU thread may not access any managed data in between performing a kernel launch and a subsequent synchronization call, regardless of whether the GPU kernel actually touches that same data (or any managed data at all). The mere potential for concurrent CPU and GPU access is sufficient for a process-level exception to be raised.

J.2.2.2. Explicit Synchronization and Logical GPU Activity

Note that explicit synchronization is required even if **kernel** runs quickly and finishes before the CPU touches **y** in the above example. Unified Memory uses logical activity to determine whether the GPU is idle. This aligns with the CUDA programming model, which specifies that a kernel can run at any time following a launch and is not guaranteed to have finished until the host issues a synchronization call.

Any function call that logically guarantees the GPU completes its work is valid. This includes **cudaDeviceSynchronize()**; **cudaStreamSynchronize()** and **cudaStreamQuery()** (provided it returns **cudaSuccess** and not **cudaErrorNotReady**) where the specified stream is the only stream still executing on the GPU; **cudaEventSynchronize()** and **cudaEventQuery()** in cases where the specified event is not followed by any device work; as well as uses of **cudaMemcpy()** and **cudaMemset()** that are documented as being fully synchronous with respect to the host.

Dependencies created between streams will be followed to infer completion of other streams by synchronizing on a stream or event. Dependencies can be created via **`cudaStreamWaitEvent()`** or implicitly when using the default (NULL) stream.

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:

- It is always permitted for the CPU to access non-managed zero-copy data while the GPU is active.
- 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.
- 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.
- 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. As mentioned previously, managed memory functions identically to non-managed memory from the perspective of the GPU. 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.
    // 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;
}
```

J.2.2.3. Managing Data Visibility and Concurrent CPU + GPU Access

Until now it was assumed that any active kernel may use any managed memory and that it was invalid to use managed memory from the CPU while a kernel is active. Here we present a system for finer-grained control of managed 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. Streams describe independence between work items and hence allow potentially greater efficiency through concurrency.

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. A new function exists to control this:

```
cudaError_t cudaStreamAttachMemAsync(cudaStream_t stream,
                                     void *ptr,
                                     size_t length=0,
                                     unsigned int flags=0);
```

The **cudaStreamAttachMemAsync()** function associates **length** bytes of memory starting from **ptr** with the specified **stream**. (Currently, **length** must always be 0 to indicate that the entire region should be attached.) Because of this association, the Unified Memory system allows CPU access to this 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.

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: it is the programmer's responsibility to ensure that guarantee is honored.

In addition to allowing greater concurrency, the use of **cudaStreamAttachMemAsync()** can (and typically does) enable data transfer optimizations within the Unified Memory system that may affect latencies and other overhead.

J.2.2.4. Stream Association Examples

Associating data with a stream allows fine-grained control over CPU + GPU concurrency, but what data is visible to which streams must be kept in mind. Looking at the earlier synchronization example:

```
__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;
}
```

Here we explicitly associate **y** with host accessibility, thus enabling access at all times from the CPU. (As before, note the absence of **cudaDeviceSynchronize()** before the access.) Accesses to **y** by the GPU running **kernel** will now produce undefined results.

Now, here's an example of where things can go wrong. This may seem like it should be okay, but it's not:

```
__device__ __managed__ int x, y=2;
__global__  void  kernel() {
    x = 10;
}

int main() {
    cudaStream_t stream1;
    cudaStreamCreate(&stream1);

    cudaStreamAttachMemAsync(stream1, &x);// Associate "x" with stream1.
    cudaDeviceSynchronize();              // Wait for "x" attachment to occur.

    kernel<<< 1, 1, 0, stream1 >>>();     // Note: Launches into stream1.
    y = 20;                       // ERROR: "y" is still associated globally
                                  // with all streams by default
 return  0;
}
```

Note how the access to **y** will cause an error because, even though **x** has been associated with a stream, we have told the system nothing about who can see **y**. The system therefore conservatively assumes that **kernel** might access it and prevents the CPU from doing so.

J.2.2.5. Stream Attach With 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 private stream.
void run_task(int *in, int *out, int length) {
    // Create a stream for us to use.
    cudaStream_t stream;
    cudaStreamCreate(&amp;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.

J.2.2.6. Advanced Topic: Modular Programs and Data Access Constraints

In the previous example **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 (for example, within a base-class constructor) 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.

J.2.2.7. Memcpy()/Memset() Behavior With Managed Memory

Since managed memory can be accessed from either the host or the device, **cudaMemcpy*()** relies on the type of transfer, specified using **cudaMemcpyKind**, to determine whether the data should be accessed as a host pointer or a device pointer.

If **cudaMemcpyHostTo*()** is specified and the source data is managed, then it will be accessed from the device if it has global visibility or if it's associated with the stream being used for the copy operation; otherwise, it will be accessed from the host. Similar rules apply to the destination when **cudaMemcpy*ToHost()** is specified and the destination is managed memory. Note that a segmentation fault can occur during the copy operation if data is being accessed from the host and its associated stream is active on the GPU.

If **cudaMemcpyDeviceTo*()** is specified and the source data is managed, then it will be accessed from the device. The source must either have global visibility or it must be associated with the copy stream; otherwise, an error is returned. Similar rules apply to the destination when **cudaMemcpy*ToDevice()** is specified and the destination is managed memory.

If **cudaMemcpyDefault()** is specified, then managed data will be accessed from the device if it has global visibility or if it's associated with the copy stream; otherwise, it will be accessed from the host.

When using **cudaMemset*()** with managed memory, the data is always accessed from the device. The data must either have global visibility, or it must be associated with the stream being used for the **Memset()** operation; 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. The program must synchronize appropriately to ensure the operation has completed before accessing any associated data from the CPU.

J.2.3. Language Integration

Users of the CUDA Runtime API who compile their host code using **nvcc** have access to additional language integration features, such as shared symbol names and inline kernel launch via the **<<<...>>>** operator. Unified Memory adds one additional element to

CUDA's language integration: variables annotated with the **__managed__** keyword can be referenced directly from both host and device code.

The following example, seen earlier in Simplifying GPU Programming, illustrates a simple use of **__managed__** global declarations:

```
// Managed variable declaration is an extra annotation with __device__
__device__ __managed__  int  x;

__global__  void  kernel() {
    // Reference "x" directly - it's a normal variable on the GPU.
 printf( "GPU sees: x = %d\n" , x);
}

int  main() {
    // Set "x" from Host code. Note it's just a normal variable on the CPU.
 x = 1234;

    // Launch a kernel which uses "x" from the GPU.
 kernel<<< 1, 1 >>>();
 cudaDeviceSynchronize();

 return  0;
}
```

The new capability introduced with **__managed__** variables is that the symbol is available in both device code and in host code without the need to dereference a pointer, and the data is shared by all. This makes it particularly easy to exchange data between host and device programs without the need for explicit allocations or copying.

Semantically, the behavior of **__managed__** variables is identical to that of storage allocated via **cudaMallocManaged()**. Data is hosted in physical GPU storage and is visible to all GPUs in the system as well as the CPU. Stream visibility defaults to **cudaMemAttachGlobal**, but may be constrained using **cudaStreamAttachMemAsync()**.

A valid CUDA context is necessary for the correct operation of **__managed__** variables. Accessing **__managed__** variables can trigger CUDA context creation if a context for the current device hasn't already been created. In the example above, accessing **x** before the kernel launch triggers context creation on device 0. In the absence of that access, the kernel launch would have triggered context creation.

C++ objects declared as **__managed__** are subject to certain specific constraints, particularly where static initializers are concerned. Please refer to C/C++ Language Support in the *CUDA C Programming Guide* for a list of these constraints.

J.2.3.1. Host Program Errors with __managed__ Variables

The use of **__managed__** variables depends upon the underlying Unified Memory system functioning correctly. Incorrect functioning can occur if, for example, the CUDA installation failed or if the CUDA context creation was unsuccessful.

When CUDA-specific operations fail, typically an error is returned that indicates the source of the failure. Using **__managed__** variables introduces a new failure mode whereby a non-CUDA operation (for example, CPU access to what should be a valid host memory address) can fail if the Unified Memory system is not operating correctly. Such invalid memory accesses cannot easily be attributed to the underlying CUDA

subsystem, although a debugger such as **cuda-gdb** will indicate that a managed memory address is the source of the failure.

J.2.4. Querying Unified Memory Support

J.2.4.1. Device Properties

Unified Memory is supported only on devices with compute capability 3.0 or higher. A program may query whether a GPU device supports managed memory by using **cudaGetDeviceProperties()** and checking the new **managedMemSupported** property. The capability can also be determined using the individual attribute query function **cudaDeviceGetAttribute()** with the attribute **cudaDevAttrManagedMemSupported**.

Either property will be set to 1 if managed memory allocations are permitted on the GPU and under the current operating system. Note that Unified Memory is not supported for 32-bit applications (unless on Android), even if a GPU is of sufficient capability.

J.2.4.2. Pointer Attributes

To determine if a given pointer refers to managed memory, a program can call **cudaPointerGetAttributes()** and check the value of the **isManaged** attribute. This attribute is set to 1 if the pointer refers to managed memory and to 0 if not.

J.2.5. Advanced Topics

J.2.5.1. Managed Memory with Multi-GPU Programs

Managed allocations are automatically visible to all GPUs in a system via the peer-to-peer capabilities of the GPUs. If peer mappings are not available (for example, between GPUs of different architectures), then the system will fall back to using zero-copy memory in order to guarantee data visibility. This fallback happens automatically, 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. On a system with more than two GPUs, so long as peer mapping is supported between all visible GPUs, managed allocations will not fall back to zero-copy memory.

Alternatively, 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 Appendix CUDA Environment Variables.

J.2.5.2. Using `fork()` with Managed Memory

The Unified Memory system does not allow sharing of managed memory pointers between processes. It will not correctly manage memory handles that have been duplicated via a **`fork()`** operation. Results will be undefined if either the child or parent accesses managed data following a **`fork()`**.

It is safe, however, to **`fork()`** a child process that then immediately exits via an **`exec()`** call, because the child drops the memory handles and the parent becomes the sole owner once again. It is not safe for the parent to exit and leave the child to access the handles.