NVIDIA CUDA™

NVIDIA CUDA C Programming Guide

Version 4.2

4/16/2012

Changes from Version 4.1

- Updated Chapter 4, Chapter 5, and Appendix F to include information on devices of compute capability 3.0.
- Replaced each reference to "processor core" with "multiprocessor" in Section 1.3.
- Replaced Table A-1 by a reference to http://developer.nvidia.com/cuda-gpus.
- Added new Section B.13 on the warp shuffle functions.

Table of Contents

List of Figures

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 1-1.

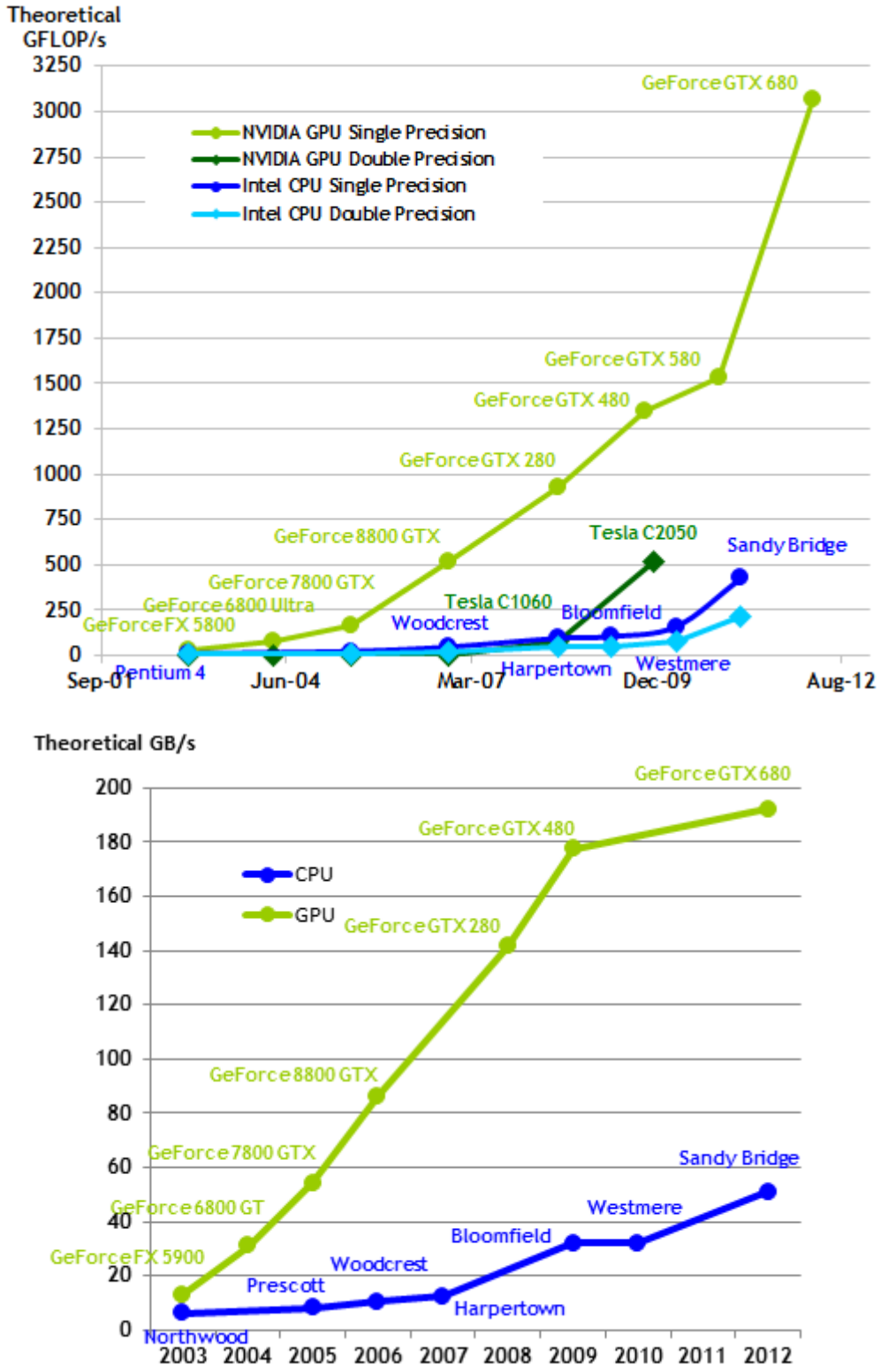


Figure 1-1. Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

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 1-2.

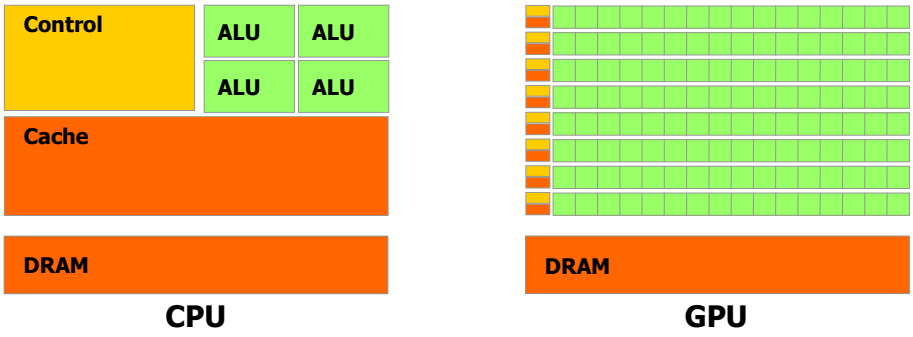


Figure 1-2. The GPU Devotes More Transistors to Data Processing

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 Architecture

In November 2006, NVIDIA introduced CUDA™, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – 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 1-3, other languages, application programming interfaces, or directives-based approaches are supported, such as FORTRAN, DirectCompute, OpenCL, OpenACC.

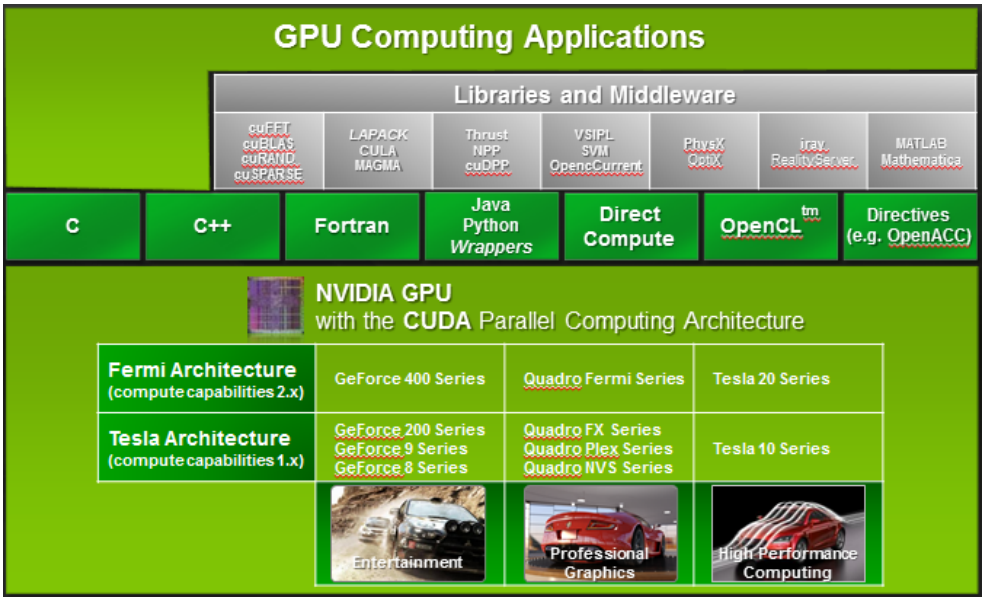


Figure 1-3. CUDA is Designed to Support Various Languages and Application Programming Interfaces

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.

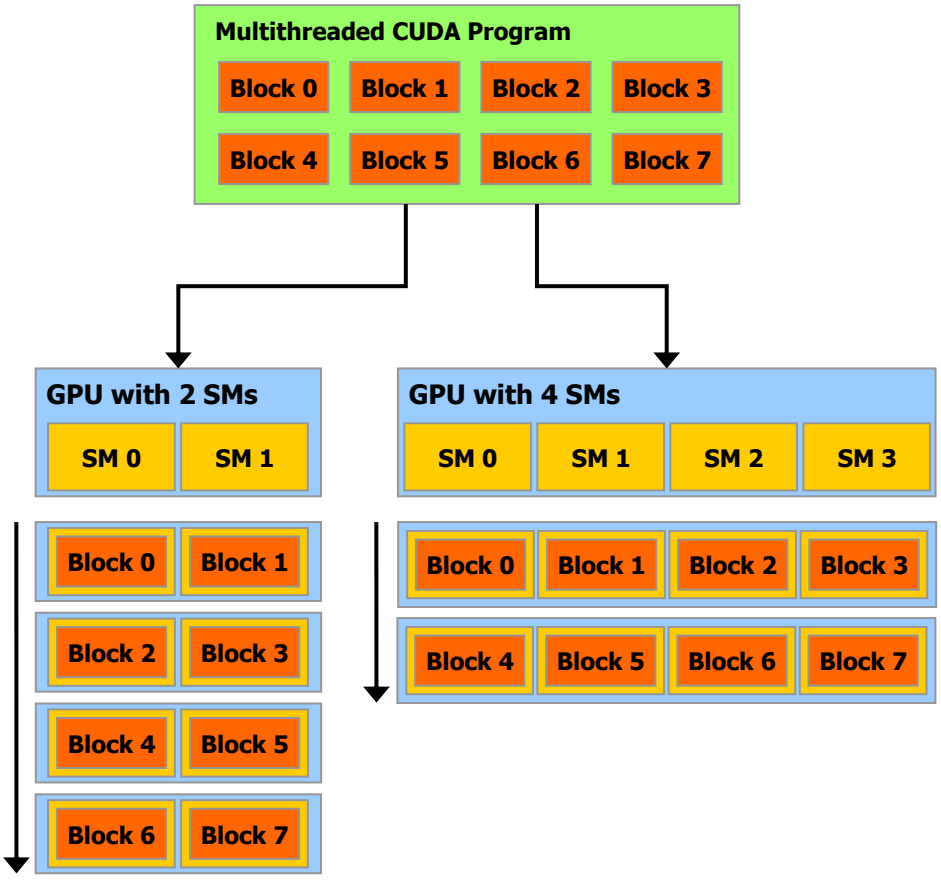
The CUDA parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

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 1-4, and only the runtime system needs to know the physical multiprocessor count.

This scalable programming model allows the CUDA 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 Appendix A for a list of all CUDA-enabled GPUs).



A GPU is built around an array of Streaming Multiprocessors (SMs) (see Chapter 4 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 1-4. Automatic Scalability

1.4 Document's Structure

This document is organized into the following chapters:

- Chapter 1 is a general introduction to CUDA.
- Chapter 2 outlines the CUDA programming model.
- Chapter 3 describes the programming interface.
- Chapter 4 describes the hardware implementation.
- Chapter 5 gives some guidance on how to achieve maximum performance.
- Appendix A lists all CUDA-enabled devices.
- Appendix B is a detailed description of all extensions to the C language.
- Appendix C lists the mathematical functions supported in CUDA.
- Appendix D lists the C++ features supported in device code.
- Appendix E gives more details on texture fetching.
- Appendix F gives the technical specifications of various devices, as well as more architectural details.
- Appendix G introduces the low-level driver API.

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 Chapter 3.

Full code for the vector addition example used in this chapter and the next can be found in the *vectorAdd* SDK code 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 Appendix B.18). 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 2-1. 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.

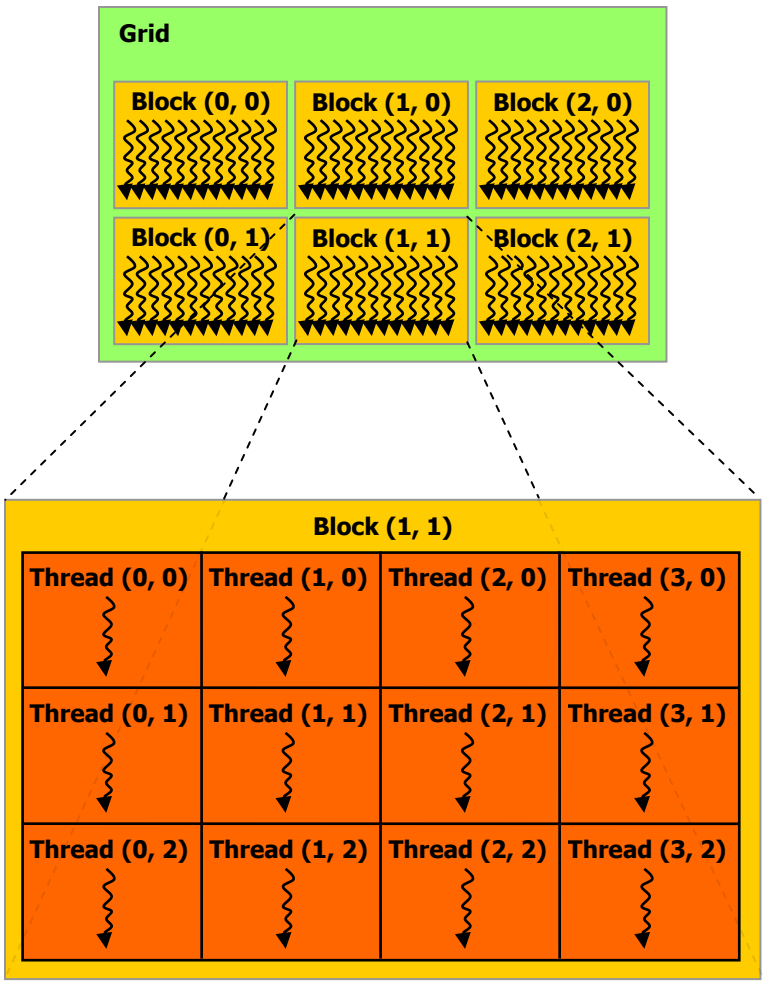


Figure 2-1. Grid of Thread Blocks

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. Section 3.2.3 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 2-2. 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 Sections 5.3.2.1, 5.3.2.4, and 5.3.2.5). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Section 3.2.10).

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

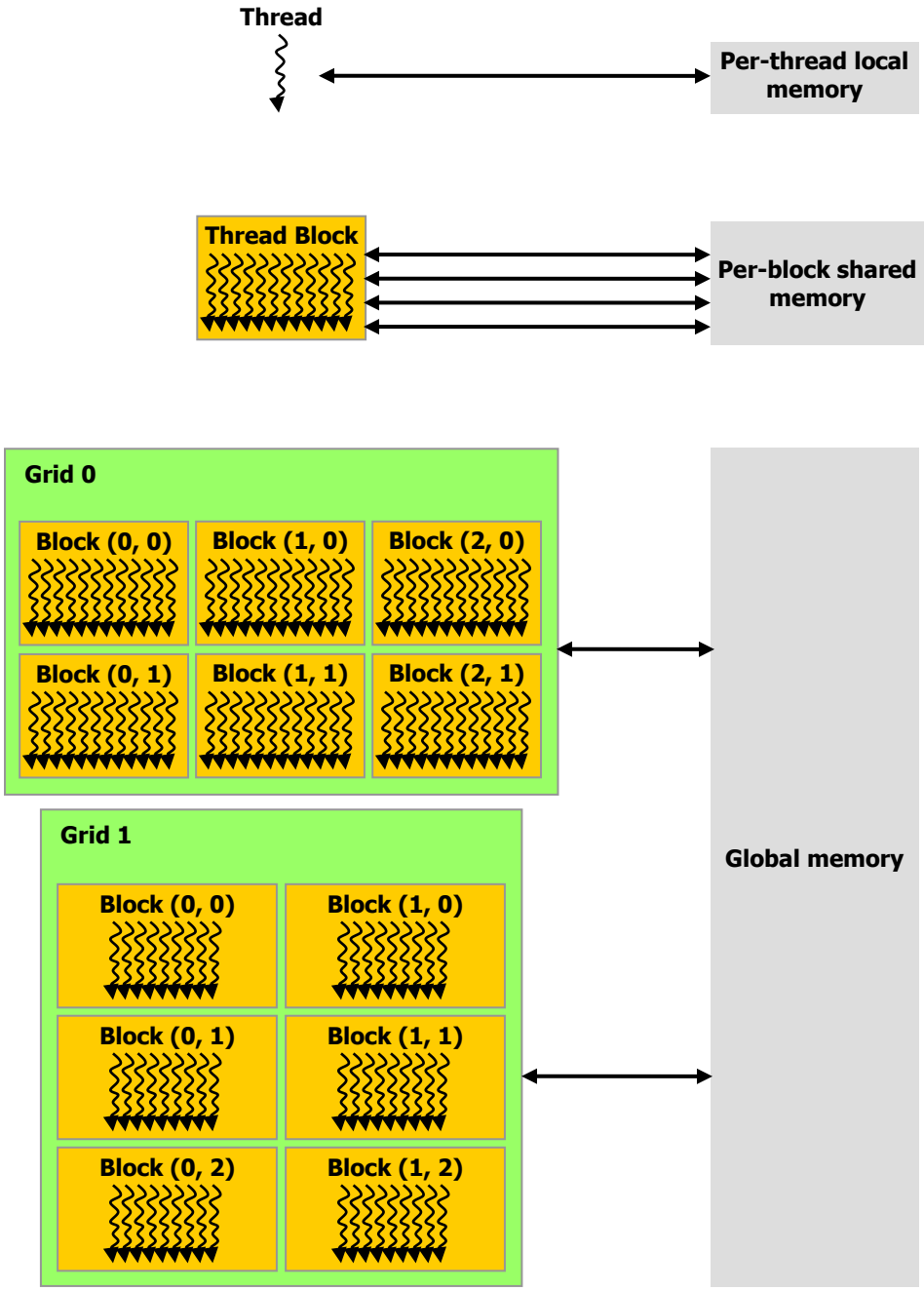
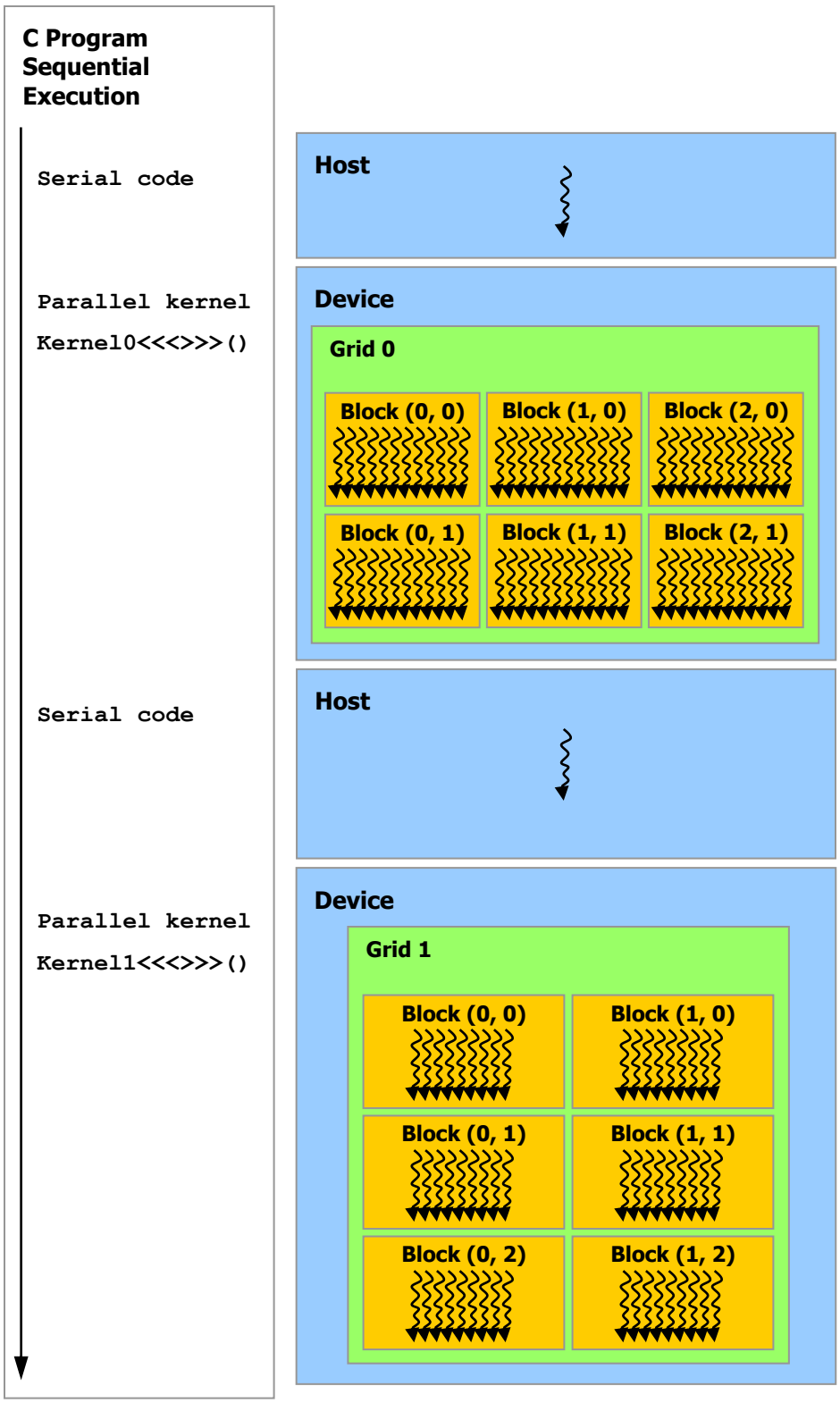


Figure 2-2. Memory Hierarchy

2.4 Heterogeneous Programming

As illustrated by Figure 2-3, 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 Chapter 3). 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 2-3. 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 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.

Appendix A lists of all CUDA-enabled devices along with their compute capability. Appendix F 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 Chapter 2. 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 Appendix B. Any source file that contains some of these extensions must be compiled with **nvcc** as outlined in Section 3.1.

The runtime is introduced in Section 3.2. 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 Appendix G 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 Section 2.1 (and described in more details in Section B.18) 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,
- Or ignore the modifed host code (if any) and use the CUDA driver API (see Appendix G) 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 applications to benefit from latest compiler improvements. 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 Section 3.1.4.

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:

- Setting **CUDA_CACHE_DISABLE** to 1 disables caching (i.e. no binary code is added to or retrieved from the cache).
- **CUDA_CACHE_MAXSIZE** specifies the size of the compute cache in bytes; the default size is 32 MB and the maximum size is 4 GB; 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_CACHE_PATH** specifies the folder where the compute cache files are stored; the default values are:
  - on Windows, **%APPDATA%\NVIDIA\ComputeCache**,

- on MacOS, **$HOME/Library/Application\ Support/NVIDIA/ComputeCache**,
- on Linux, **~/.nv/ComputeCache**.

- Setting **CUDA_FORCE_PTX_JIT** to 1 forces the device driver to ignore any binary code embedded in an application (see Section 3.1.4) 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.

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. 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 Sections 3.1.2 and 3.1.3. 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 Section 3.1.1.2).

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 shorthands 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 Appendix D. 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.

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** dynamic library which is typically included in the application installation package. All its entry points are prefixed with **cuda**.

As mentioned in Section 2.4, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Section 3.2.2 gives an overview of the runtime functions used to manage device memory.

Section 3.2.3 illustrates the use of shared memory, introduced in Section 2.2, to maximize performance.

Section 3.2.4 introduces page-locked host memory that is required to overlap kernel execution with data transfers between host and device memory.

Section 3.2.5 describes the concepts and API used to enable asynchronous concurrent execution at various levels in the system.

Section 3.2.6 shows how the programming model extends to a system with multiple devices attached to the same host.

Section 3.2.8 describe how to properly check the errors generated by the runtime.

Section 3.2.9 mentions the runtime functions used to manage the CUDA C call stack.

Section 3.2.10 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.

Section 3.2.11 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 Section G.1 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. 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 Section 3.2.6.2). 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 Section 2.4, the CUDA programming model assumes a system composed of a host and a device, each with their own separate memory. Kernels can only 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 Section 3.2.10.

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 Section 2.1, 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 Section 5.3.2.1, 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×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×height×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 Section B.2 shared memory is allocated using the **__shared__** qualifier.

Shared memory is expected to be much faster than global memory as mentioned in Section 2.2 and detailed in Section 5.3.2.3. 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 3-1. *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;
}
```

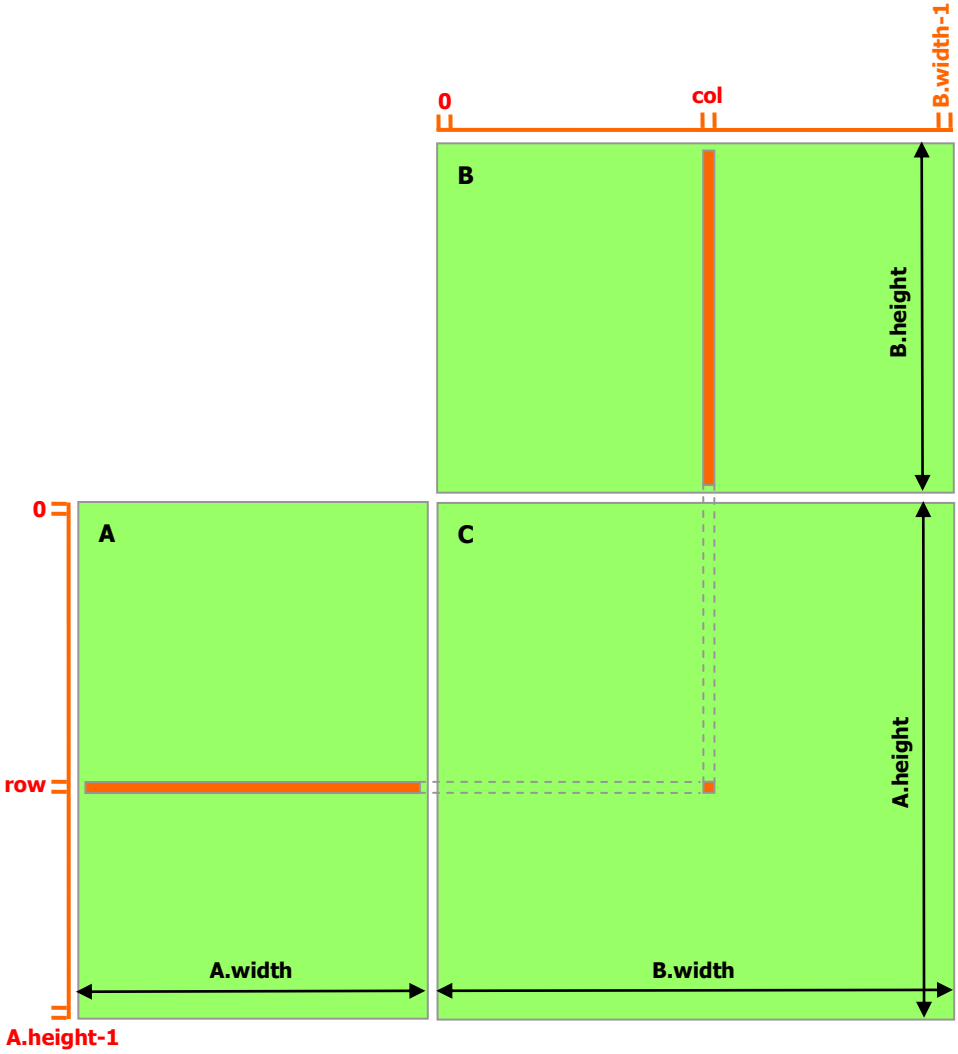


Figure 3-1. Matrix Multiplication without Shared Memory

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 3-2, 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 Section B.1.1) 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
    cudaMemcpy(C.elements, d_C.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)
{
    // Block row and column
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Each thread block computes one sub-matrix Csub of C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);
```

```
    // Each thread computes one element of Csub
    // by accumulating results into Cvalue
    float Cvalue = 0;

    // Thread row and column within Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Loop over all the sub-matrices of A and B that are
    // required to compute Csub
    // Multiply each pair of sub-matrices together
    // and accumulate the results
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

        // Get sub-matrix Asub of A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Get sub-matrix Bsub of B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Shared memory used to store Asub and Bsub respectively
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load Asub and Bsub from device memory to shared memory
        // Each thread loads one element of each sub-matrix
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);

        // Synchronize to make sure the sub-matrices are loaded
        // before starting the computation
        __syncthreads();

        // Multiply Asub and Bsub together
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();
    }

    // Write Csub to device memory
    // Each thread writes one element
    SetElement(Csub, row, col, Cvalue);
}
```

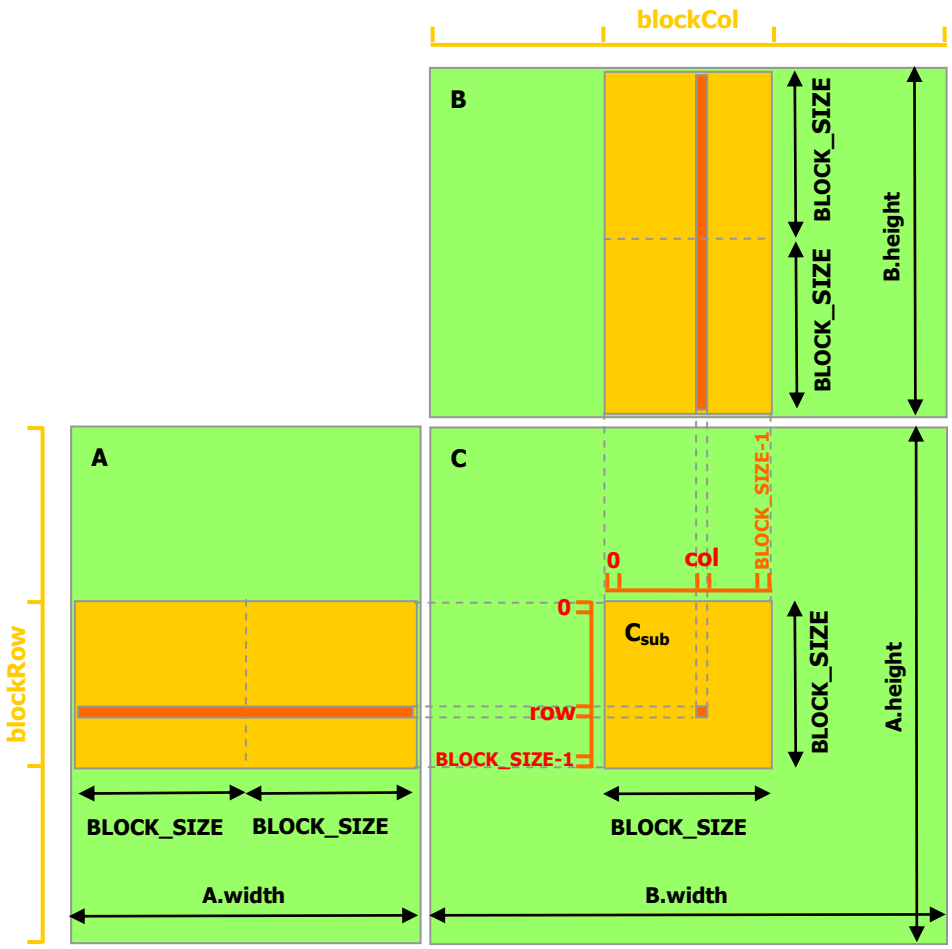


Figure 3-2. Matrix Multiplication with Shared Memory

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 Section 3.2.5;
- 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 Section 3.2.4.3;

- 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 Section 3.2.4.2.

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 SDK 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 Section 3.2.6 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 Section 3.2.7). 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 Section 3.2.7.

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 Section 3.2.5.4) 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 Section 3.2.5) 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 calls 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 Section 3.2.6.1), which is equal to 1 for devices that support mapped page-locked host memory.

Note that atomic functions (Section B.11) 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.

When an application is run via cuda-gdb, the Visual Profiler, or the Parallel Nsight CUDA Debugger, all launches are synchronous.

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 Section 3.2.6.1), 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 Section 3.2.2).

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 Section 3.2.6.1), which is equal to 1 for devices that support it.

The maximum number of kernel launches that a device can execute concurrently is sixteen.

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 Section 3.2.6.1), 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**.
Section 3.2.5.5.5 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 Section 3.2.5.6 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 either 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 Section F.4.1.

For devices that support concurrent kernel execution, 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 (Section 3.2.5.2), concurrent kernel execution (Section 3.2.5.3), and/or concurrent data transfers (Section 3.2.5.4).

For example, on devices that do not support concurrent data transfers, the two streams of the code sample of Section 3.2.5.5.1 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 Section 3.2.5.5.1 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, 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 Section 3.2.5.5.4. 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 Section 3.2.5.5.4, which can represent only a small portion of the total execution time of the kernel.

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 task 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 Section 3.2.5.6.1 can be used to time the code sample of Section 3.2.5.5.1 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 calls 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 or memory copy 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
```

`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 Section 3.2.5.5.2), 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 on Windows Vista/7 in TCC mode (see Section 3.6), on Windows XP, or on Linux, 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 Section 3.2.7), 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 Section 3.2.7), this is done using the regular memory copy functions mentioned in Section 3.2.2.

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 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 asynchronous commands (see Section 3.2.5) issued after the copy to either device can start.

Note that if peer-to-peer access is enabled between two devices via **cudaDeviceEnablePeerAccess()** as described in Section 3.2.6.4, 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

For 64-bit applications on Windows Vista/7 in TCC mode (see Section 3.6), on Windows XP, or on Linux, 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 Section 3.2.4.1) 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 Section 3.2.4.3).

Applications may query if the unified address space is used for a particular device by checking that the **unifiedAddressing** device property (see Section 3.2.6.1) is equal to 1.

3.2.8 Error Checking

All runtime functions return an error code, but for an asynchronous function (see Section 3.2.5), 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 Section 3.2.5) 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()**.

3.2.9 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, Parallel Nsight) or an unspecified launch error, otherwise.

3.2.10 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 Section 5.3.2.5.

3.2.10.1 Texture Memory

Texture memory is read from kernels using the device functions described in Section B.8. The process of reading a texture is called a *texture fetch*. The first parameter of a texture fetch specifies an object called a *texture reference*.

A texture reference defines which part of texture memory is fetched. As detailed in Section 3.2.10.1.3, it must be bound through runtime functions to some region of memory, called a *texture*, before it can be used by a kernel. Several distinct texture references might be bound to the same texture or to textures that overlap in memory.

A texture reference has several attributes. One of them is 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 type of a texel 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 Section B.3.1

Other attributes define the input and output data types of the texture fetch, as well as how the input coordinates are interpreted and what processing should be done.

A texture can be any region of linear memory or a CUDA array (described in Section 3.2.10.2.3).

Table F-2 lists the maximum texture width, height, and depth depending on the compute capability of the device.

Textures can also be layered as described in Section 3.2.10.1.5.

3.2.10.1.1 Texture Reference Declaration

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 data that is returned when fetching the texture; **Type** 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 Section B.3.1;
- **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** is equal to **cudaReadModeNormalizedFloat** or **cudaReadModeElementType**; if it is **cudaReadModeNormalizedFloat** and **Type** is a 16-bit or 8-bit integer type, the value 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; **ReadMode** 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.

3.2.10.1.2 Runtime Texture Reference Attributes

The other attributes of a texture reference are mutable and can be changed at runtime through the host runtime. They specify whether texture coordinates are normalized or not, the addressing mode, and texture filtering, as detailed below.

By default, textures are referenced (by the functions of Section B.8) 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 64×32 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 64×32 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.

It is valid to call the device functions of Section B.8 with coordinates that are be out of range. The addressing mode defines what happens if 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.

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.

Appendix E gives more details on texture fetching.

3.2.10.1.3 Texture Binding

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;
}
```

- **normalized** specifies whether texture coordinates are normalized or not, as described in Section 3.2.10.1.2;
- **filterMode** specifies the filtering mode, that is how the value returned when fetching the texture is computed based on the input texture coordinates; **filterMode** 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;

- **addressMode** specifies the addressing mode, as described in Section 3.2.10.1.2; **addressMode** is 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;
- **channelDesc** describes the format of the value that is returned when fetching the texture; **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.

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. 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 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 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,
        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.

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 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;
}
```

3.2.10.1.4 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.

3.2.10.1.5 Layered Textures

A one-dimensional or two-dimensional *layered texture* (also know 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 bound to a CUDA array created 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 Sections B.8.5 and B.8.6. Texture filtering (see Appendix E) is done only within a layer, not across layers.

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

3.2.10.1.6 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 3-1.

Table 3-1. 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 bound to a CUDA array created by calling **`cudaMalloc3DArray()`** with the **`cudaArrayCubemap`** flag.

Cubemap textures are fetched using the device function described in Sections B.8.7.

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

3.2.10.1.7 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 bound to a CUDA array created by calling **`cudaMalloc3DArray()`** with the **`cudaArrayLayered`** and **`cudaArrayCubemap`** flags.

Cubemap layered textures are fetched using the device function described in Sections B.8.8. Texture filtering (see Appendix E) is done only within a layer, not across layers.

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

3.2.10.1.8 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 Section B.8.9). 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 F-2 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.

3.2.10.2 Surface Memory

For devices of compute capability 2.0 and higher, a CUDA array (described in Section 3.2.10.2.3), created with the **`cudaArraySurfaceLoadStore`** flag, can be read and written via a *surface reference* using the functions described in Section B.9.

Table F-2 lists the maximum surface width, height, and depth depending on the compute capability of the device.

3.2.10.2.1 Surface Reference Declaration

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.

3.2.10.2.2 Surface Binding

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;
}
```

3.2.10.2.3 Cubemap Surfaces

Cubemap surfaces are accessed using **`surfCubemapread()`** and **`surfCubemapwrite()`** (Sections B.9.11 and B.9.12) 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 3-1.

3.2.10.2.4 Cubemap Layered Surfaces

Cubemap layered surfaces are accessed using **`surfCubemapLayeredread()`** and **`surfCubemapLayeredwrite()`** (Sections B.9.13 and B.9.14) 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 3-1, so index ((2 * 6) + 3), for example, accesses the fourth face of the third cubemap.

3.2.10.3 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 readable by kernels through texture fetching and may only be bound to texture references with the same number of packed components.

3.2.10.4 Read/Write Coherency

The texture and surface memory is cached (see Section 5.3.2.5) 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.

3.2.11 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 Sections 3.2.11.1 and 3.2.11.2. 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.

Sections 3.2.11.1 and 3.2.11.2 give specifics for each graphics API and some code samples.

Section 3.2.11.3 gives specifics for when the system is in SLI mode.

3.2.11.1 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;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;
```

```
    // Write positions
    positions[y * width + x] = make_float4(u, w, v, 1.0f);
}
```

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.

3.2.11.2 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 (Section G.4), 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.

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);

    // Draw and present
    ...
}

void releaseVB()
{
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}

__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;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
                 make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

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));

    // 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);

    // Draw and present
    ...
}

void releaseVB()
{
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}

__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;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
                 make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

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,
                                         &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);

    // Draw and present
    ...
}

void releaseVB()
{
    cudaGraphicsUnregisterResource(positionsVB_CUDA);
    positionsVB->Release();
}

__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;

    // Calculate uv coordinates
    float u = x / (float)width;
    float v = y / (float)height;
    u = u * 2.0f - 1.0f;
    v = v * 2.0f - 1.0f;

    // Calculate simple sine wave pattern
    float freq = 4.0f;
    float w = sinf(u * freq + time)
            * cosf(v * freq + time) * 0.5f;

    // Write positions
    positions[y * width + x] =
                 make_float4(u, w, v, __int_as_float(0xff00ff00));
}
```

3.2.11.3 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 Sections 3.2.11.2 and 3.2.11.1 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 Section 2.5) 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 3-3. 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, …).

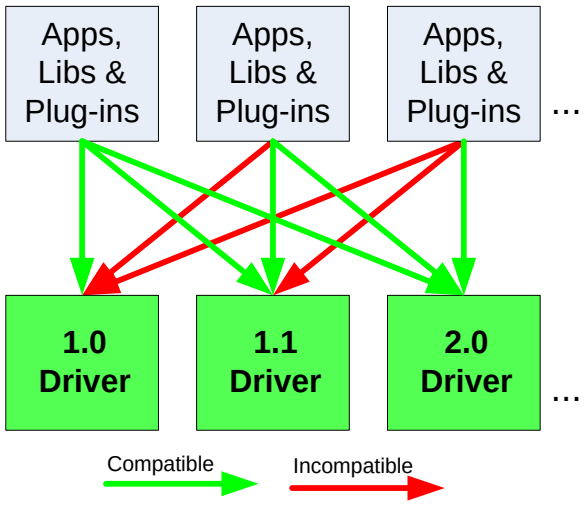


Figure 3-3. The Driver API is Backward, but Not Forward Compatible

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 Section 3.2.6.1).

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 CUDA 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 Section 4.1. 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 Section 4.2. Unlike CPU cores they are issued in order however and there is no branch prediction and no speculative execution.

Sections 4.1 and 4.2 describe the architecture features of the streaming multiprocessor that are common to all devices. Sections F.3.1, F.4.1, and F.5.1 provide the specifics for devices of compute capabilities 1.x, 2.x, and 3.0, respectively.

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. Section 2.2 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.

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 Sections F.3.2, F.3.3, F.4.2, F.4.3, F.5.2, and F.5.3) and which thread performs the final write is undefined.

If an atomic instruction (see Section B.11) 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 W_{block} in a block is as follows:

$$W_{block} = ceil(\frac{T}{W_{size}}, 1)$$

- 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 R_{block} allocated for a block is as follows:

For devices of compute capability 1.x:

$$R_{block} = ceil(ceil(W_{block}, G_W) \times W_{size} \times R_k, G_T)$$

For devices of compute capability 2.x:

$$R_{block} = ceil(R_k \times W_{size}, G_R) \times W_{block}$$

- G_W is the warp allocation granularity, equal to 2 (compute capability 1.x only),
- R_k is the number of registers used by the kernel,
- G_R is the register allocation granularity, which is equal to
  - 256 for devices of compute capability 1.0 and 1.1,
  - 512 for devices of compute capability 1.2 and 1.3,
  - 64 for devices of compute capability 2.x,
  - 256 for devices of compute capability 3.0.

The total amount of shared memory S_{block} in bytes allocated for a block is as follows:

$$S_{block} = ceil(S_k, G_S)$$

- S_k is the amount of shared memory used by the kernel in bytes,
- G_S is the shared memory allocation granularity, which is equal to
  - 512 for devices of compute capability 1.x,
  - 128 for devices of compute capability 2.x,
  - 256 for devices of compute capability 3.0.

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 Section 3.2.5. 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 Section 3.2.5.

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 Section 4.2, 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 Section 5.4.1 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 Section F.3.1,
- 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 Section F.4.1,
- $2L$ 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 Section F.4.1,

- *8L* for devices of compute capability 3.0 since a multiprocessor issues a pair of instructions per warp over one clock cycle for four warps at a time, as mentioned in Section F.5.1.

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 3.0, 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 3.0, which translates to 6 warps for devices of compute capability 1.x and 22 warps for devices of compute capability 2.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. 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). If this ratio is 15, for example, then to hide latencies of about 600 clock cycles, about 10 warps are required for devices of compute capability 1.x and about 40 for devices of compute capability 2.x and higher (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 (Section B.5) or synchronization point (Section B.6). 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 (Section B.18), the memory resources of the multiprocessor, and the resource requirements of the kernel as described in Section 4.2. 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 Appendix F 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 Section B.1.4).

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 Section 5.3.2.2) and the number of instructions to a minimum. Register usage can be controlled using the **`-maxrregcount`** compiler option or launch bounds as described in Section B.19.

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 Section 5.3.1, 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/L2 caches 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 Section 3.2.3, 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 Section F.4.1, for devices of compute capability 2.x and higher, 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 Sections 5.3.2.1, 5.3.2.3, 5.3.2.4, and 5.3.2.5. 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 Section 3.2.4.

In addition, when using mapped page-locked memory (Section 3.2.4.3), 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 Section 5.3.2.1). 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 Section 3.2.6.1) 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.

5.3.2.1 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. Sections F.3.2, F.4.2, and F.5.2 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 Sections F.3.2 and F.4.2,
- Using data types that meet the size and alignment requirement detailed in Section 5.3.2.1.1,
- Padding data in some cases, for example, when accessing a two-dimensional array as described in Section 5.3.2.1.2.

5.3.2.1.1 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 Section B.3.1 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.

5.3.2.1.2 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 Section 5.3.2.1.1):

```
    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.

5.3.2.2 Local Memory

Local memory accesses only occur for some automatic variables as mentioned in Section B.2. 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 Section 5.3.2.1. 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 higher, local memory accesses are always cached in L1 and L2 in the same way as global memory accesses (see Section F.4.2).

5.3.2.3 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 Sections F.3.3, F.4.3, and F.5.3 for devices of compute capability 1.x, 2.x, and 3.0, respectively.

5.3.2.4 Constant Memory

The constant memory space resides in device memory and is cached in the constant cache mentioned in Sections F.3.1 and F.4.1.

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.

5.3.2.5 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 respect to get good performance (see Sections 5.3.2.1 and 5.3.2.4), higher bandwidth can be achieved providing that there is locality in the texture fetches or surface reads (this is less likely for devices of compute capability 2.x and higher given that global memory reads are cached on these devices);
- 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 Section 3.2.10.1.1).

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

Section C.2), single-precision instead of double-precision, or flushing denormalized numbers to zero;

- Minimize divergent warps caused by control flow instructions as detailed in Section 5.4.2;
- Reduce the number of instructions, for example, by optimizing out synchronization points whenever possible as described in Section 5.4.3 or by using restricted pointers as described in Section B.2.4.

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. Therefore, if T is the number of operations per clock cycle, the instruction throughput is one instruction every 32/T clock cycles.

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 5-1 gives the throughputs of the arithmetic instructions that are natively supported in hardware for devices of various compute capabilities.

Table 5-1. Throughput of Native Arithmetic Instructions (Operations per Clock Cycle per Multiprocessor)

	Compute Capability				
	1.0 1.1 1.2	**1.3**	**2.0**	**2.1**	**3.0**
32-bit floating-point add, multiply, multiply-add	8	8	32	48	192
64-bit floating-point add, multiply, multiply-add	1	1	16(*)	4	8
32-bit integer add	10	10	32	48	192
32-bit integer compare	10	10	32	48	160
32-bit integer shift	8	8	16	16	32
Logical operations	8	8	32	48	160
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	Multiple instructions	16	16	32
24-bit integer multiply (`__[u]mul24`)	8	8	Multiple instructions	Multiple instructions	Multiple instructions
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base-2 exponential (`exp2f`),	2	2	4	8	32

sine (**__sinf**), cosine (**__cosf**)					
Type conversions from 8-bit and 16-bit integer to 32-bit integer	8	8	16	16	128
Type conversions from and to 64-bit types	Multiple instructions	1	16(*)	4	8
All other type conversions	8	8	16	16	32

(*) Throughput is lower 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], **__fmul_r[d,u]**, and **__fmaf_r[n,z,d,u]** (see Section C.2.1) 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 Section C.2.1) 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 **math_functions.h** 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 Section 5.3.2.2). 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 Section 4.1. 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 Section B.20).

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 3.0.

Note that **__syncthreads()** can impact performance by forcing the multiprocessor to idle as detailed in Section 5.2.3.

Because a warp executes one common instruction at a time, threads within a warp are implicitly synchronized and this can sometimes be used to omit **__syncthreads()** for better performance.

In the following code sample, for example, there is no need to call **__syncthreads()** after each of the additions performed within the body of the "**if (tid < 32) { }**" statement since they operate within a single warp (this is assuming the size of a warp is 32).

Simply removing the **__syncthreads()** is not enough however; **smem** must also be declared as volatile as described in Section D.2.1.2.

```
__device__ void sum(float* g_idata, float* g_odata)
{
    unsigned int tid = threadIdx.x;
    extern __shared__ float s_data[];

    // Assign initial value
    s_data[tid] = g_idata[...];
    __syncthreads();

    // Perform sum in shared memory.
    // This code sample assumes that the block size is 256
    // (see the reduction sample in the GPU Computing SDK
    // for a complete and general implementation
    if (tid < 128)
        s_data[tid] += s_data[tid + 128];
    __syncthreads();
    if (tid <  64)
        s_data[tid] += s_data[tid +  64];
    __syncthreads();
    if (tid <  32) {
        // No __syncthreads() necessary after each of the
        // following lines (as long as we access the data via
        // a pointer declared as volatile) because the 32 threads
        // in each warp execute in lock-step with each other
        volatile float* s_ptr = s_data;
        s_ptr[tid] += s_ptr[tid + 32];
        s_ptr[tid] += s_ptr[tid + 16];
        s_ptr[tid] += s_ptr[tid +  8];
        s_ptr[tid] += s_ptr[tid +  4];
        s_ptr[tid] += s_ptr[tid +  2];
        s_ptr[tid] += s_ptr[tid +  1];
    }

    // Write result for this thread block to global memory
    if (tid == 0)
        g_odata[blockIdx.x] = s_data[0];
}
```

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 only.

__global__ functions must have **void** return type.

Any call to a **__global__** function must specify its execution configuration as described in Section B.18.

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 Section 3.1.4 can be used to differentiate code paths between host and device:

```
__host__ __device__ func()
{
#if __CUDA_ARCH__ == 100
    // Device code path for compute capability 1.0
#elif __CUDA_ARCH__ == 200
    // Device code path for compute capability 2.0
#elif __CUDA_ARCH__ == 300
    // Device code path for compute capability 3.0
#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 Section 5.3.2.2.

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 three 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()**).

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 Section B.18). 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 B-1.

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 char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4, short1, ushort1, short2, ushort2, short3, ushort3, short4, ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1, ulong1, long2, ulong2, long3, ulong3, long4, ulong4, longlong1, ulonglong1, longlong2, ulonglong2, float1, float2, float3, float4, double1, double2

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 B-1.

Table B-1. 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 Section B.3.2) and contains the dimensions of the grid.

B.4.2 blockIdx

This variable is of type **uint3** (see Section B.3.1) and contains the block index within the grid.

B.4.3 blockDim

This variable is of type **dim3** (see Section B.3.2) and contains the dimensions of the block.

B.4.4 threadIdx

This variable is of type **uint3** (see Section B.3.1) 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 Section 4.1 for the definition of a warp).

B.5 Memory Fence Functions

```
void __threadfence_block();
```

waits until all global and shared memory accesses made by the calling thread prior to **__threadfence_block()** are visible to all threads in the thread block.

```
void __threadfence();
```

waits until all global and shared memory accesses made by the calling thread prior to **__threadfence()** are visible to:

- All threads in the thread block for shared memory accesses,
- All threads in the device for global memory accesses.

```
void __threadfence_system();
```

waits until all global and shared memory accesses made by the calling thread prior to **__threadfence_system()** are visible to:

- All threads in the thread block for shared memory accesses,
- All threads in the device for global memory accesses,
- Host threads for page-locked host memory accesses (see Section 3.2.4.3).

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

In general, when a thread issues a series of writes to memory in a particular order, other threads may see the effects of these memory writes in a different order. **__threadfence_block()**, **__threadfence()**, and **__threadfence_system()** can be used to enforce some ordering.

One 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 Section B.11 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 its result is visible to
        // all other threads
        __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.

Appendix C provides accuracy information for some of these functions when relevant.

B.8 Texture Functions

For texture functions, a combination of the texture reference's immutable (i.e. compile-time) and mutable (i.e. runtime) attributes determine how the texture coordinates are interpreted, what processing occurs during the texture fetch, and the return value delivered by the texture fetch. Immutable attributes are described in Section 3.2.10.1.1. Mutable attributes are described in Section 3.2.10.1.2. Texture fetching is described in Appendix E.

B.8.1 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);
```

fetch the region of linear memory bound to texture reference **`texRef`** using integer texture coordinate **`x`**. **`tex1Dfetch()`** only works with non-normalized coordinates (Section 3.2.10.1.2), so only the border and clamp addressing modes are supported. are supported. It does not perfrom 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 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**.

B.8.3 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**.

B.8.4 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**.

B.8.5 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 Section 3.2.10.1.5.

B.8.6 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 Section 3.2.10.1.5.

B.8.7 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 Section 3.2.10.1.6.

B.8.8 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 Section 3.2.10.1.7.

B.8.9 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 cubemap texture reference **texRef** using texture coordinates **x** and **y**, as described in Section 3.2.10.1.8.

B.9 Surface Functions

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

Surface reference declaration is described in Section 3.2.10.2.1 and surface binding in Section 3.2.10.2.2.

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 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 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.9.3 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.9.4 surf2Dwrite()

```
template<class Type>
void surf2Dwrite(Type data,
                 surface<void, cudaSurfaceType2D> surfRef,
                 int x, int y,
                 boundaryMode = cudaBoundaryModeTrap);
```

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

B.9.5 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.9.6 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.9.7 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.9.8 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.9.9 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.9.10 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.9.11 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.9.12 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.9.13 surfCubemabLayeredread()

```
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.9.14 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 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.11 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 32-bit 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 (Section 3.2.4.3) are not atomic from the point of view of the host or other devices.

As mentioned in Table F-1, 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.11.1 Arithmetic Functions

B.11.1.1 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.11.1.2 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.11.1.3 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.11.1.4 atomicMin()

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

reads the 32-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**.

B.11.1.5 atomicMax()

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

reads the 32-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**.

B.11.1.6 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.11.1.7 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.11.1.8 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.11.2 Bitwise Functions

B.11.2.1 atomicAnd()

```
int atomicAnd(int* address, int val);
unsigned int atomicAnd(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.11.2.2 atomicOr()

```
int atomicOr(int* address, int val);
unsigned int atomicOr(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.11.2.3 atomicXor()

```
int atomicXor(int* address, int val);
unsigned int atomicXor(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 Warp Vote Functions

Warp vote functions are only supported by devices of compute capability 1.2 and higher (see Section 4.1 for the definition of a warp).

```
int __all(int predicate);
```

evaluates **predicate** for all active threads of the warp and returns non-zero if and only if **predicate** evaluates to non-zero for all of them.

```
int __any(int predicate);
```

evaluates **predicate** for all active threads of the warp and returns non-zero if and only if **predicate** evaluates to non-zero for any of them.

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

evaluates **predicate** for all active threads of the warp and returns an integer whose N^{th} bit is set if and only if **predicate** evaluates to non-zero for the N^{th} thread of the warp. This function is only supported by devices of compute capability 2.x and higher.

B.13 Warp Shuffle Functions

__shfl, **__shfl_up**, **__shfl_down**, **__shfl_xor** exchange a variable between threads within a warp.

They are only supported by devices of compute capability 3.0 (see Section 4.1 for the definition of a warp).

B.13.1 Synopsys

```
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.13.2 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 3.0 may have an index between 0 and 31 (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 32 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.13.3 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.13.4 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.13.5 Examples

B.13.5.1 Broadcast of a single value across a warp

```
__global__ void bcast(int arg) {
    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);
}
```

B.13.5.2 Inclusive plus-scan across sub-partitions of 8 threads

```
__global__ void scan4() {
    // 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 >>>();
```

```
}
```

B.13.5.3 Reduction across a warp

```
__global__ void warpReduce() {
    // 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 >>>();
}
```

B.14 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 for the first multiprocessor can be obtained via the CUDA 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 call (note that when an application is run via cuda-gdb, the Visual Profiler, or the Parallel Nsight CUDA Debugger, all launches are synchronous).

B.15 Assertion

Assertion is only supported by devices of compute capability 2.x and higher.

```
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();
    cudaDeviceReset();
    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.16 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 Section B.16.4 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.16.1 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.16.2 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-O/S-dependent.

As described in Section B.16.1, **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 Section B.16.3). 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.

B.16.3 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.16.4 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)
```

```
    #define printf(f, ...) ((void)(f, __VA_ARGS__),0)
#endif

__global__ void helloCUDA(float f)
{
    printf("Hello thread %d, f=%f\n", threadIdx.x, f);
}

int main()
{
    helloCUDA<<<1, 5>>>(1.2345f);
    cudaDeviceReset();
    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)
    #define printf(f, ...) ((void)(f, __VA_ARGS__),0)
#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);
    cudaDeviceReset();
    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.17 Dynamic Global Memory Allocation

Dynamic global memory allocation is 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.

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.17.1 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 Section G.2), or implicitly via the CUDA runtime API (see Section 3.2). 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.17.2 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 Sections 3.2.2).

Similarly, memory allocated via the runtime (i.e. by calling any of the memory allocation functions from Sections 3.2.2) cannot be freed via **`free()`**.

Memory allocated via **`malloc()`** can be copied using the runtime (i.e. by calling any of the copy memory functions from Sections 3.2.2).

B.17.3 Examples

B.17.3.1 Per Thread Allocation

The following code sample:

```
#include <stdlib.h>
#include <stdio.h>

__global__ void mallocTest()
{
    char* ptr = (char*)malloc(123);
    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()`** command and so receives its own allocation. (Exact pointer values will vary: these are illustrative.)

B.17.3.2 Per Thread Block Allocation

```
#include <stdlib.h>

__global__ void mallocTest()
{
    __shared__ int* data;

    // The first thread in the block does the allocation
    // and 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)
       data = (int*)malloc(blockDim.x * 64);
    __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.17.3.3 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.18 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 Section 3.2.5.5 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 Section B.3.2) 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 Section B.3.2) 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 Section B.2.3; **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 and like the function arguments, are currently 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 Appendix F, 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, functions arguments (for devices of compute capability 1.x), and execution configuration.

B.19 Launch Bounds

As discussed in detail in Section 5.2.3, 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 Section 5.3.2.2) 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 Section 4.2 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 Section 5.2.3) 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 Section 3.1.4.

```
#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 Section 3.1.4, 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 Section 5.2.3 for 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.20 #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.

Appendix C. 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.

C.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.

C.1.1 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 Section F.2. This combination can be avoided by using the **`__fadd_[rn,rz,ru,rd]()`** and **`__fmul_[rn,rz,ru,rd]()`** intrinsic functions (see Section C.2).

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 C-1. 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)
`expf(x)`	2 (full range)
`exp2f(x)`	2 (full range)
`exp10f(x)`	2 (full range)
`expm1f(x)`	1 (full range)
`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)
`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)

Function	Maximum ulp error
`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)`	3 (full range)
`erfcf(x)`	6 (full range)
`erfinvf(x)`	3 (full range)
`erfcinvf(x)`	7 (full range)
`erfcxf(x)`	6 (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 $n \leq 128$, 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)
`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)

Function	Maximum ulp error
`lroundf(x)`	0 (full range)
`llrintf(x)`	0 (full range)
`llroundf(x)`	0 (full range)

C.1.2 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 C-2 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)
`rsqrt(x)`	1 (full range)
`cbrt(x)`	1 (full range)
`rcbrt(x)`	1 (full range)
`hypot(x,y)`	2 (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)`	2 (full range)
`cos(x)`	2 (full range)

Function	Maximum ulp error
`tan(x)`	2 (full range)
`sincos(x,sptr,cptr)`	2 (full range)
`sinpi(x)`	2 (full range)
`cospi(x)`	2 (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)
`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)`	8 (full range)
`erfcx(x)`	3 (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 5×10^{-12}
`j1(x)`	7 for $\|x\| < 8$ otherwise, the maximum absolute error is 5×10^{-12}
`jn(x)`	For $n \leq 128$, the maximum absolute error is 5×10^{-12}
`y0(x)`	7 for $\|x\| < 8$ otherwise, the maximum absolute error is 5×10^{-12}
`y1(x)`	7 for $\|x\| < 8$ otherwise, the maximum absolute error is 5×10^{-12}
`yn(x)`	For $\|x\| > 1.5n$, the maximum absolute error is 5×10^{-12}
`fmod(x,y)`	0 (full range)
`remainder(x,y)`	0 (full range)
`remquo(x,y,iptr)`	0 (full range)
`modf(x,iptr)`	0 (full range)

Function	Maximum ulp error
`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)

C.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 Section C.1. 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 C-3 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 C-3. 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.

C.2.1 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 1.x or for devices of compute capability 2.x and higher when the code is compiled with **-prec-div=false**, both the regular division "/" operator and **__fdividef(x,y)** have the same accuracy, but for $2^{126} < \mathbf{y} < 2^{128}$, **__fdividef(x,y)** delivers a result of zero, whereas the "/" operator delivers the correct result to within the accuracy stated in Table C-4. Also, for $2^{126} < \mathbf{y} < 2^{128}$, 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 C-4. 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.
__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.
__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 $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
__cosf(x)	For **x** in $[-\pi, \pi]$, 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))**.
__powf(x, y)	Derived from its implementation as **exp2f(y * __log2f(x))**.

C.2.2 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 C-5. 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.
__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 D. C/C++ Language Support

As described in Section 3.1, 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 Section D.1 with some restrictions described in Section D.2; it does not support run time type information (RTTI), exception handling, and the C++ Standard Library.

D.1 Code Samples

D.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;
}
```

D.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;
}
```

D.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);
    ...
}
```

D.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);
```

D.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());
    ...
}
```

D.2 Restrictions

D.2.1 Qualifiers

D.2.1.1 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__** variables are only allowed at file scope.

__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 Section B.2.3.

D.2.1.2 Volatile Qualifier

Only after the execution of a **__threadfence_block()**, **__threadfence()**, or **__syncthreads()** (Sections B.5 and B.6) 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.

For example, in the code sample of Section 5.4.3, if **s_ptr** were not declared as volatile, the compiler would optimize away the store to shared memory for each

assignment to **s_ptr[tid]**. It would accumulate the result into a register instead and only store the final result to shared memory, which would be incorrect.

D.2.2 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 Section 3.2.2 can only be used in host code.

D.2.3 Operators

D.2.3.1 Assignment Operator

__constant__ variables can only be assigned from the host code through runtime functions (Sections 3.2.2); 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 Section B.4.

D.2.3.2 Address Operator

It is not allowed to take the address of any of the built-in variables defined in Section B.4.

D.2.4 Functions

D.2.4.1 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.

D.2.4.2 Static Variables within Function

Static variables cannot be declared within the body of **__device__** and **__global__** functions.

D.2.4.3 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.

D.2.4.4 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.

D.2.5 Classes

D.2.5.1 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.

D.2.5.2 Function Members

Static member functions cannot be **__global__** functions.

D.2.5.3 Constructors and Destructors

Declaring global variables for which a constructor or a destructor needs to be called is not supported.

D.2.5.4 Virtual Functions

Declaring global variables of a class with virtual functions is not supported.

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.

D.2.5.5 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.

D.2.5.6 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).

D.2.6 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 E. Texture Fetching

This appendix gives the formula used to compute the value returned by the texture functions of Section B.8 depending on the various attributes of the texture reference (see Section 3.2.10).

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 Section 3.2.10.1.2. In this appendix, the coordinates are assumed to be in the valid range. Section 3.2.10.1.2 explained how out-of-range coordinates are remapped to the valid range based on the addressing mode.

E.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 E-1 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 Section 3.2.10.1.1).

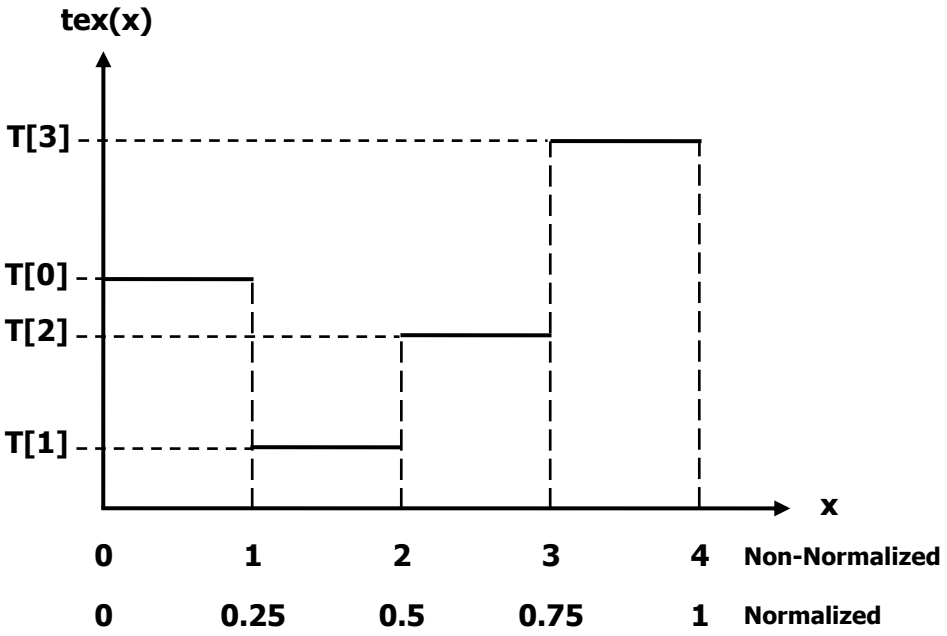


Figure E-1. Nearest-Point Sampling of a One-Dimensional Texture of Four Texels

E.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 E-2 illustrates nearest-point sampling for a one-dimensional texture with $N = 4$.

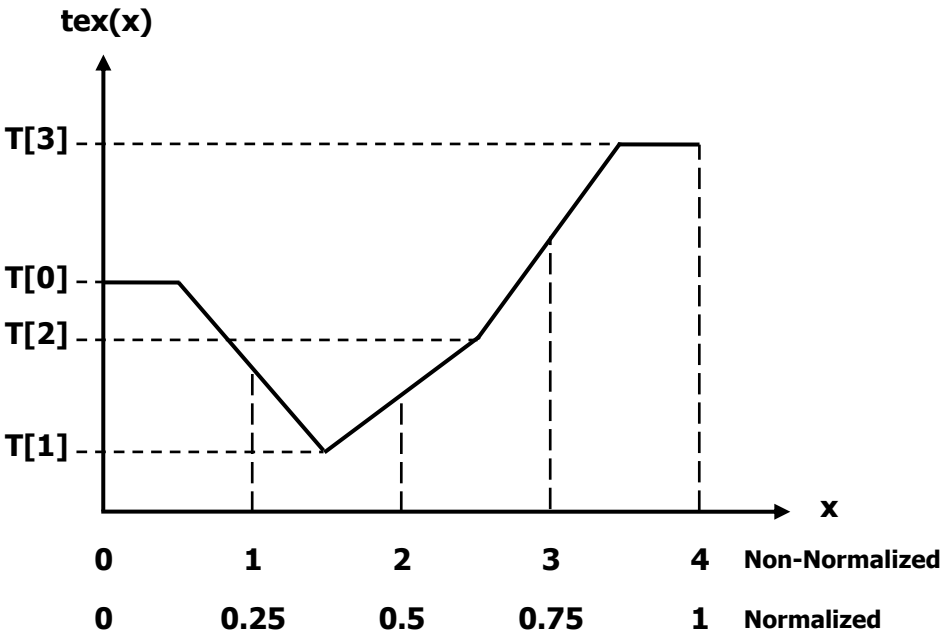


Figure E-2. Linear Filtering of a One-Dimensional Texture of Four Texels in Clamp Addressing Mode

E.3 Table Lookup

A table lookup $TL(x)$ where x spans the interval $[0, R]$ can be implemented as $TL(x) = tex(\frac{N-1}{R}x + 0.5)$ in order to ensure that $TL(0) = T[0]$ and $TL(R) = T[N-1]$.

Figure E-3 illustrates the use of texture filtering to implement a table lookup with $R = 4$ or $R = 1$ from a one-dimensional texture with $N = 4$.

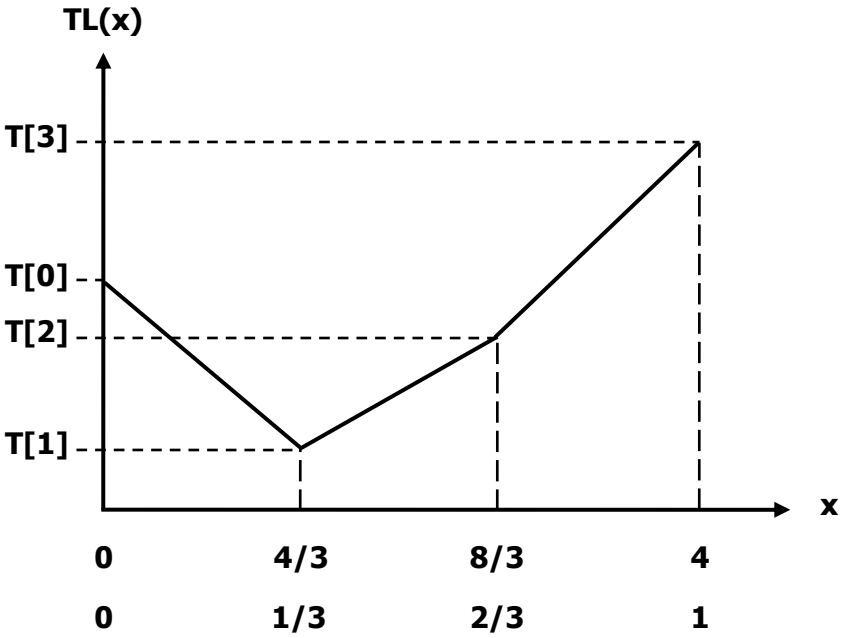


Figure E-3. One-Dimensional Table Lookup Using Linear Filtering

Appendix F.
Compute Capabilities

The general specifications and features of a compute device depend on its compute capability (see Section 2.5).

Section F.1 gives the features and technical specifications associated to each compute capability.

Section F.2 reviews the compliance with the IEEE floating-point standard.

Section F.3, F.4, and F.5 give more details on the architecture of devices of compute capability 1.x, 2.x, and 3.0, respectively.

F.1 Features and Technical Specifications

Table F-1. Feature Support per Compute Capability

<table>
<tr><th></th><th colspan="5">Compute Capability</th></tr>
<tr><th>Feature Support
(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
3.0</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Section B.11)</td><td rowspan="2">No</td><td colspan="4" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (Section B.11.1.3)</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Section B.11)</td><td colspan="2" rowspan="4">No</td><td colspan="3" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (Section B.11.1.3)</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Section B.11)</td></tr>
<tr><td>Warp vote functions (Section B.12)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Section B.11)</td><td colspan="4" rowspan="7">No</td><td rowspan="7">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (Section B.11.1.1)</td></tr>
<tr><td>__ballot() (Section B.12)</td></tr>
<tr><td>__threadfence_system() (Section B.5)</td></tr>
<tr><td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)</td></tr>
<tr><td>Surface functions (Section B.9)</td></tr>
<tr><td>3D grid of thread blocks</td></tr>
</table>

Table F-2. Technical Specifications per Compute Capability

<table>
<tr><th></th><th colspan="6">Compute Capability</th></tr>
<tr><th>Technical Specifications</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></tr>
<tr><td>Maximum dimensionality of grid of thread blocks</td><td colspan="4">2</td><td colspan="2">3</td></tr>
<tr><td>Maximum x-dimension of a grid of thread blocks</td><td colspan="5">65535</td><td>2^{31}-1</td></tr>
<tr><td>Maximum y- or z-</td><td colspan="6">65535</td></tr>
</table>

	Compute Capability					
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**
dimension of a grid of thread blocks						
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					
Maximum number of resident blocks per multiprocessor	8					16
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 amount of shared memory per multiprocessor	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					
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 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	

	Compute Capability					
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**
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
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	

	Compute Capability					
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.x**	**3.0**
Maximum number of surfaces that can be bound to a kernel					8	16
Maximum number of instructions per kernel	2 million				512 million	

F.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. Section C.2.1):

- **__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,
- **__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 slighty 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.

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.

F.3 Compute Capability 1.x

F.3.1 Architecture

For devices of compute capability 1.x, a multiprocessor consists of:

- 8 CUDA cores for arithmetic operations (see Section 5.4.1 for throughputs of arithmetic operations),
- 1 double-precision floating-point unit for double-precision floating-point arithmetic operations,
- 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,
- 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 Section 3.2.10.

The local and global memory spaces reside in device memory and are not cached.

F.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. Sections F.3.2.1 and F.3.2.2 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 F-1

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.

F.3.2.1 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.

F.3.2.2 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.

F.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.

F.3.3.1 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**., that is only if **d** is equal to 1, i.e. **s** is odd.

Figure F-2 shows some examples of strided access for devices of compute capability 3.0. 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.

F.3.3.2 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 F-3 shows some examples of memory read accesses that involve the broadcast mechanism for devices of compute capability 3.0. 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.

F.3.3.3 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__ float 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];
```

F.3.3.4 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 **double**s accessed as follows:

```
extern __shared__ float 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__ float 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.

F.4 Compute Capability 2.x

F.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 Section 5.4.1 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 Section 5.4.1 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()
{
    ...
}

// Host code

// Runtime API
// cudaFuncCachePreferShared: shared memory is 48 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 Section 3.2.6.1). 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 Section 3.2.10.

F.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 F-1 shows some examples of global memory accesses and corresponding memory transactions based on compute capability.

F.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__ float 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 F-3 shows some examples of memory read accesses that involve the broadcast mechanism for devices of compute capability 3.0. The same examples apply for devices of compute capability 2.x.

F.4.3.1 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**., that is only if **d** is equal to 1, i.e. **s** is odd.

Figure F-2 shows some examples of strided access for devices of compute capability 3.0. 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.

F.4.3.2 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__ float 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__ float 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.

F.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.

F.5 Compute Capability 3.0

F.5.1 Architecture

A multiprocessor consists of:

- 192 CUDA cores for arithmetic operations (see Section 5.4.1 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, 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 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 Section 3.2.6.1). The maximum L2 cache size is 512 KB.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes two 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 Section 3.2.10.

F.5.2 Global Memory

Global memory accesses for devices of compute capability 3.0 behave in the same way as for devices of compute capability 2.x (see Section F.4.2).

Figure F-1 shows some examples of global memory accesses and corresponding memory transactions based on compute capability.

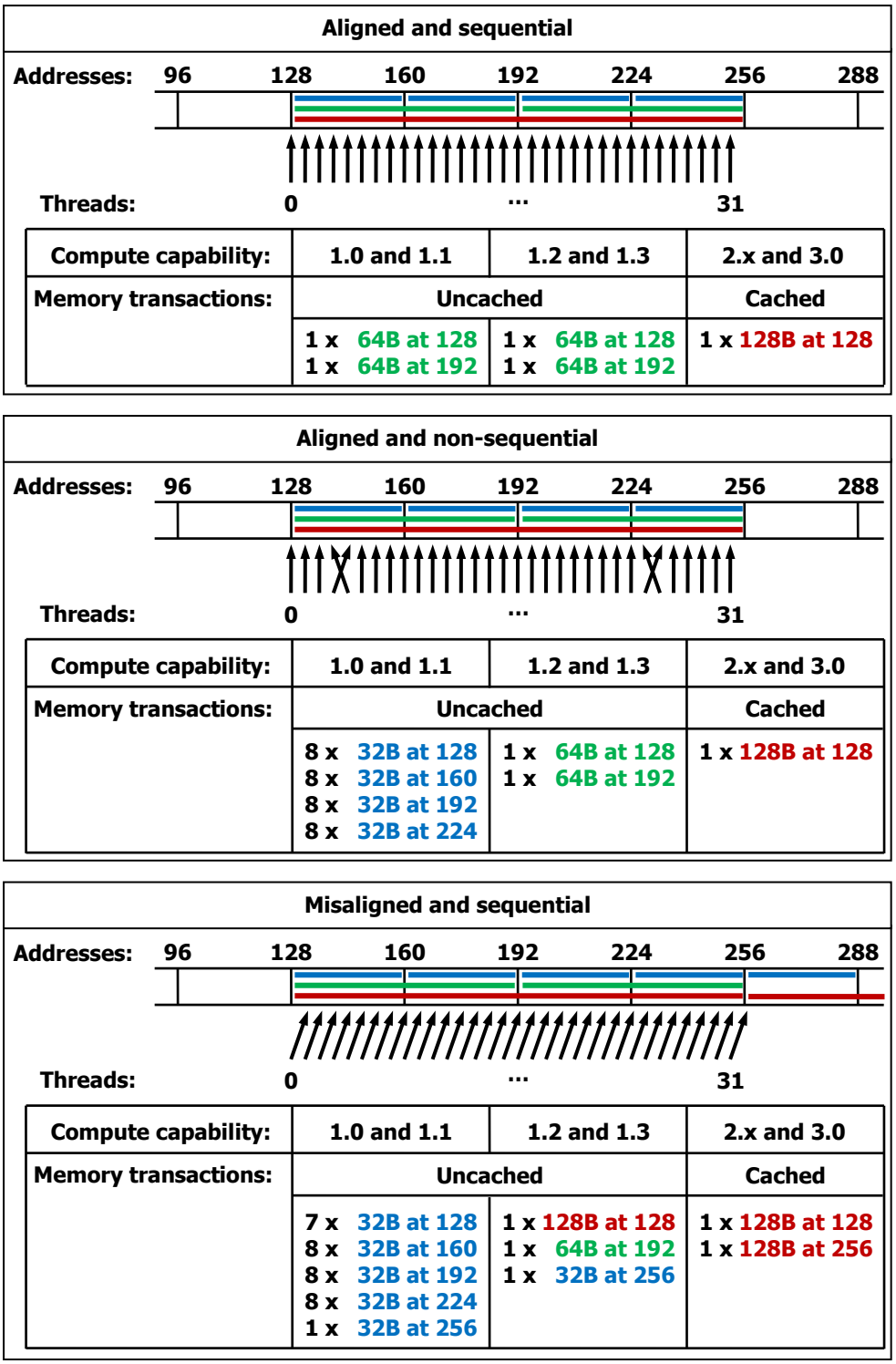


Figure F-1. Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

F.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 F-2 shows some examples of strided access.

Figure F-3 shows some examples of memory read accesses that involve the broadcast mechanism.

F.5.3.1 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 address within the same 64-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).

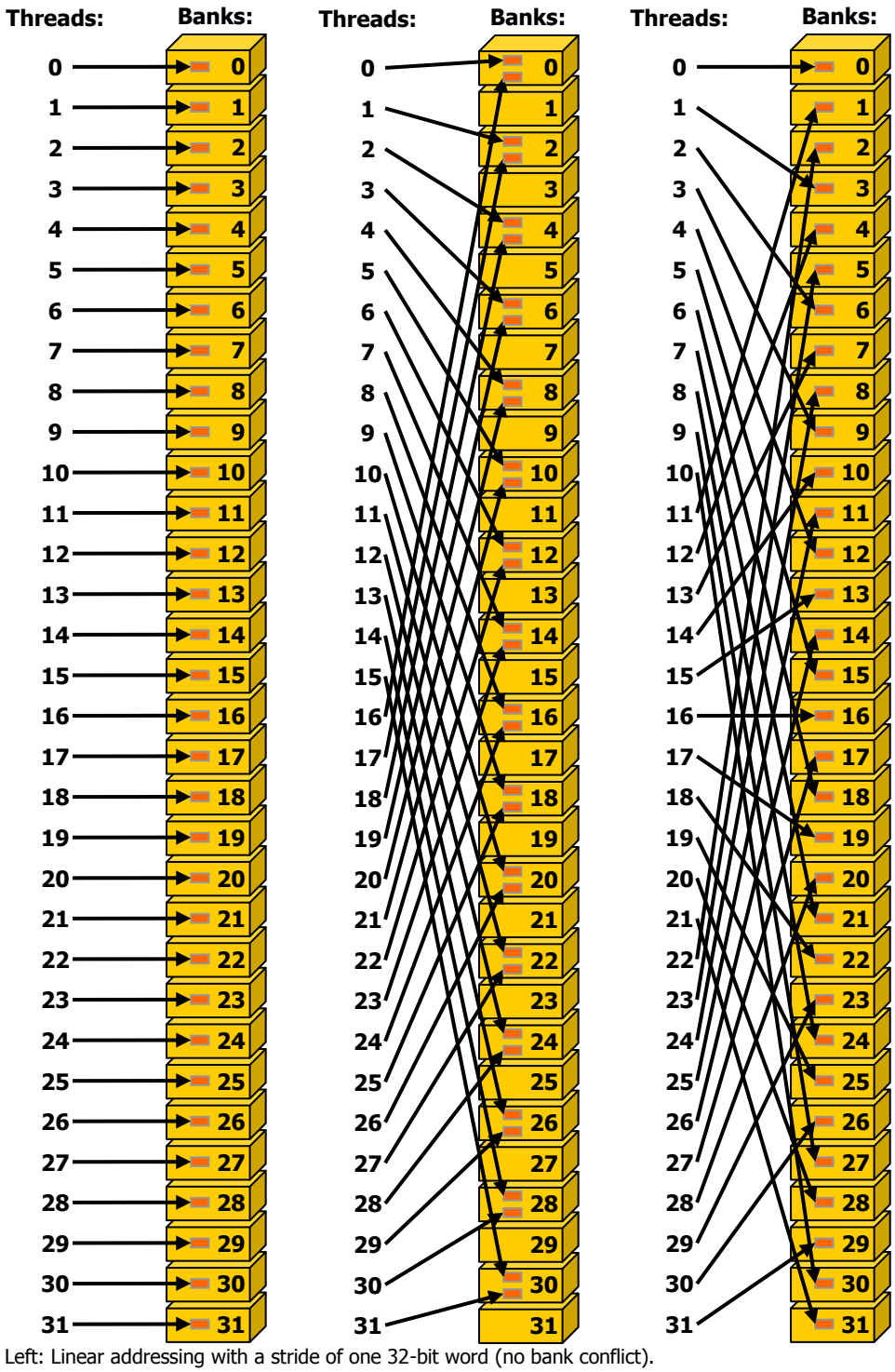
In this mode, the same access pattern generates fewer bank conflicts than on devices of compute capability 2.x for 64-bit accesses and as many or fewer for 32-bit accesses.

F.5.3.2 32-Bit Mode

Successive 32-bit words map to successive banks.

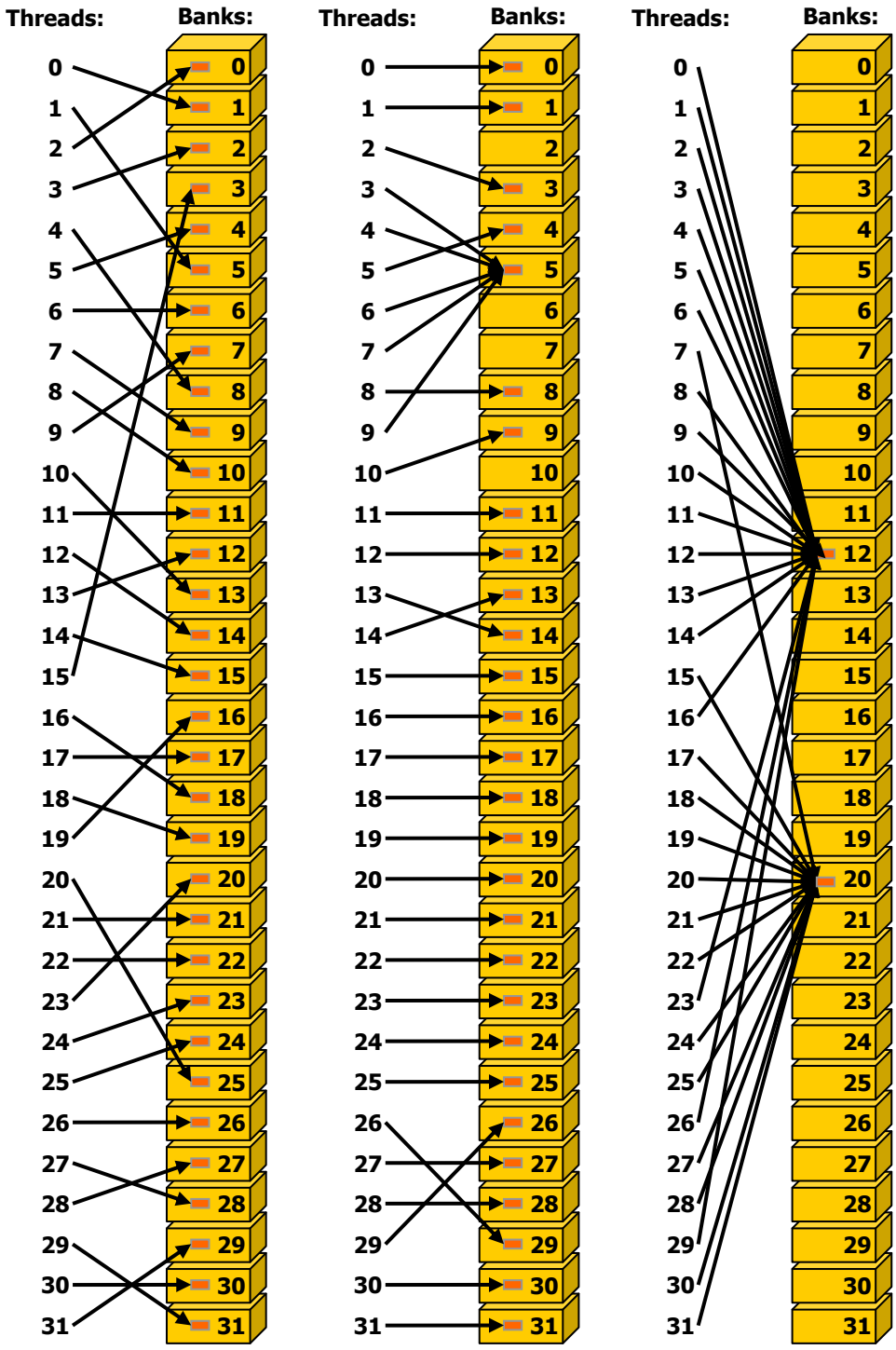
A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word or within two 32-bit words whose indices *i* and *j* are in the same 64-word aligned segment (i.e. a segment whose first index is a multiple of 64) and such that *j=i+32* (even though the two addresses 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 address 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.



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 F-2 Examples of Strided Shared Memory Accesses for Devices of Compute Capability 3.0



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 F-3 Examples of Irregular Shared Memory Accesses for Devices of Compute Capability 3.0

Appendix G. Driver API

This appendix assumes knowledge of the concepts described in Section 3.2.

The driver API is implemented in the **nvcuda** dynamic library 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 G-1.

Table G-1. 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 Section G.1.

Within a CUDA context, kernels are explicitly loaded as *PTX* or binary objects by the host code as described in Section G.2. Kernels written in C must therefore be compiled separately into *PTX* or binary objects. Kernels are launched using API entry points as described in Section G.3.

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 Section 2.1 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* SDK code sample.

G.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 G-1.

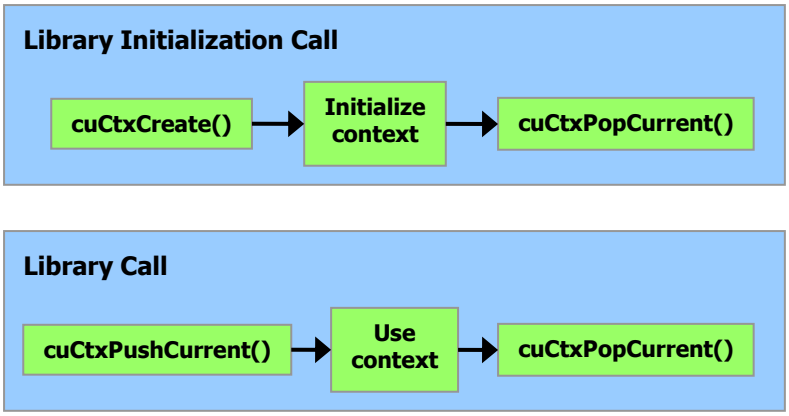


Figure G-1 Library Context Management

G.2 Module

Modules are dynamically loadable packages of device code and data, akin to DLLs in Windows, that are output by **nvcc** (see Section 3.1). 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 ERROR_BUFFER_SIZE 100
CUmodule cuModule;
CUjit_option options[3];
void* values[3];
char* PTXCode = "some PTX code";
options[0] = CU_ASM_ERROR_LOG_BUFFER;
values[0]  = (void*)malloc(ERROR_BUFFER_SIZE);
options[1] = CU_ASM_ERROR_LOG_BUFFER_SIZE_BYTES;
values[1]  = (void*)ERROR_BUFFER_SIZE;
options[2] = CU_ASM_TARGET_FROM_CUCONTEXT;
values[2]  = 0;
cuModuleLoadDataEx(&cuModule, PTXCode, 3, options, values);
for (int i = 0; i < values[1]; ++i) {
    // Parse error string here
}
```

G.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 B-1. 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;
```

G.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 Section 3.2), **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.

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com