NVIDIA CUDA™

Programming Guide

Version 3.0

2/9/2010

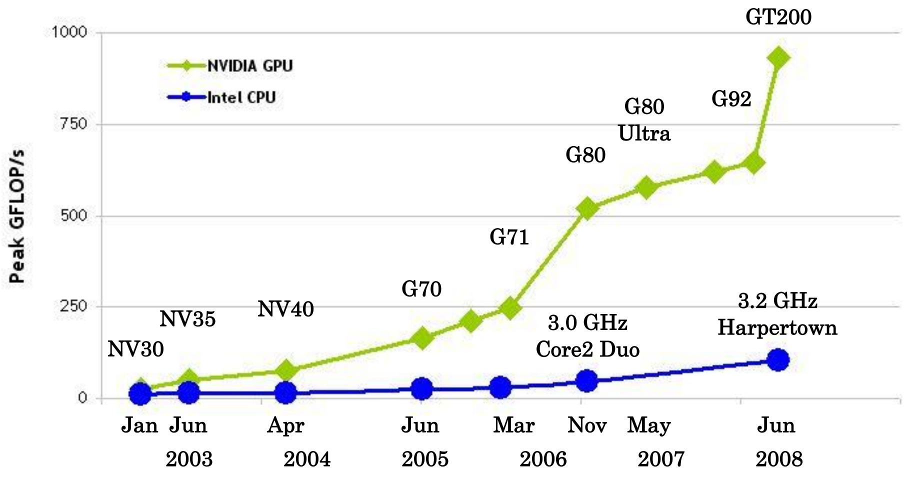
Table of Contents

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.



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	

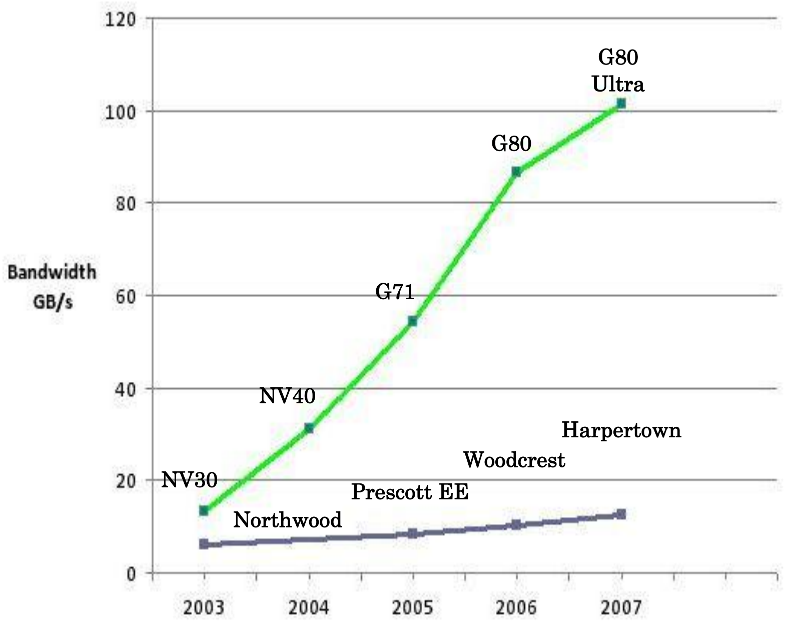


Figure 1-1. Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

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.

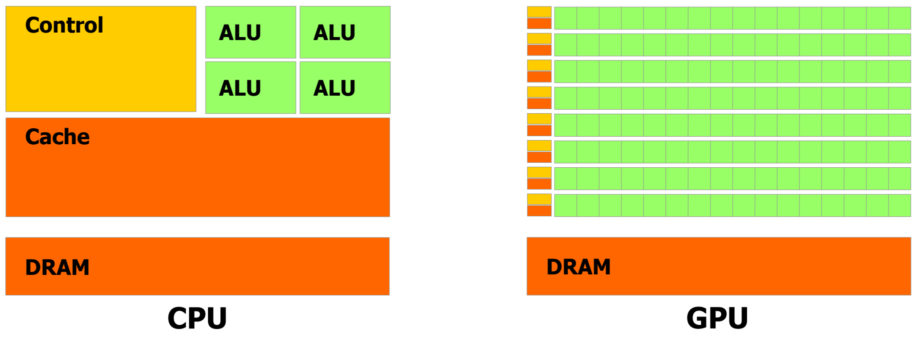


Figure 1-2. The GPU Devotes More Transistors to Data Processing

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 or application programming interfaces are supported, such as CUDA FORTRAN, OpenCL, and DirectCompute.

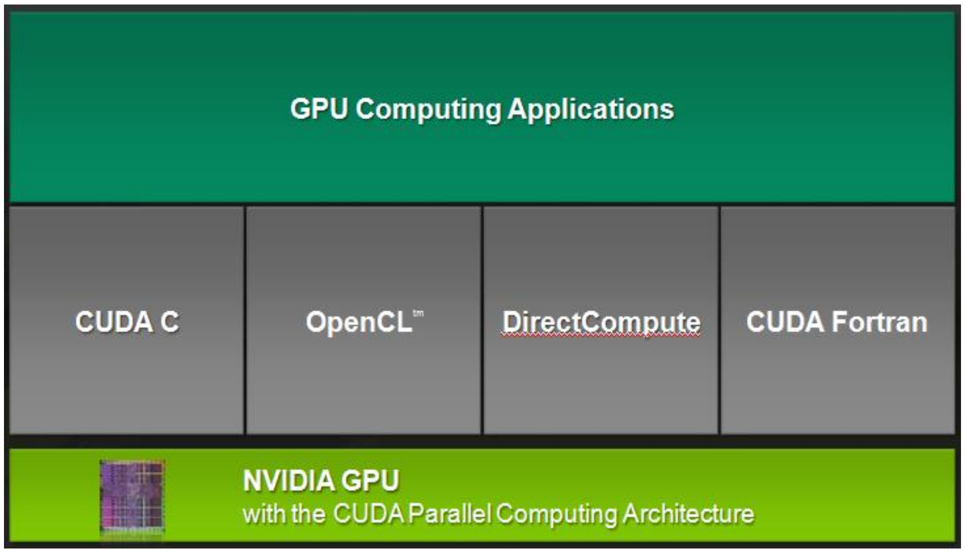

GPU Computing Applications
CUDA C
OpenCL™
DirectCompute
CUDA Fortran
NVIDIA GPU
with the CUDA Parallel Computing Architecture

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

1.3 A Scalable Programming Model

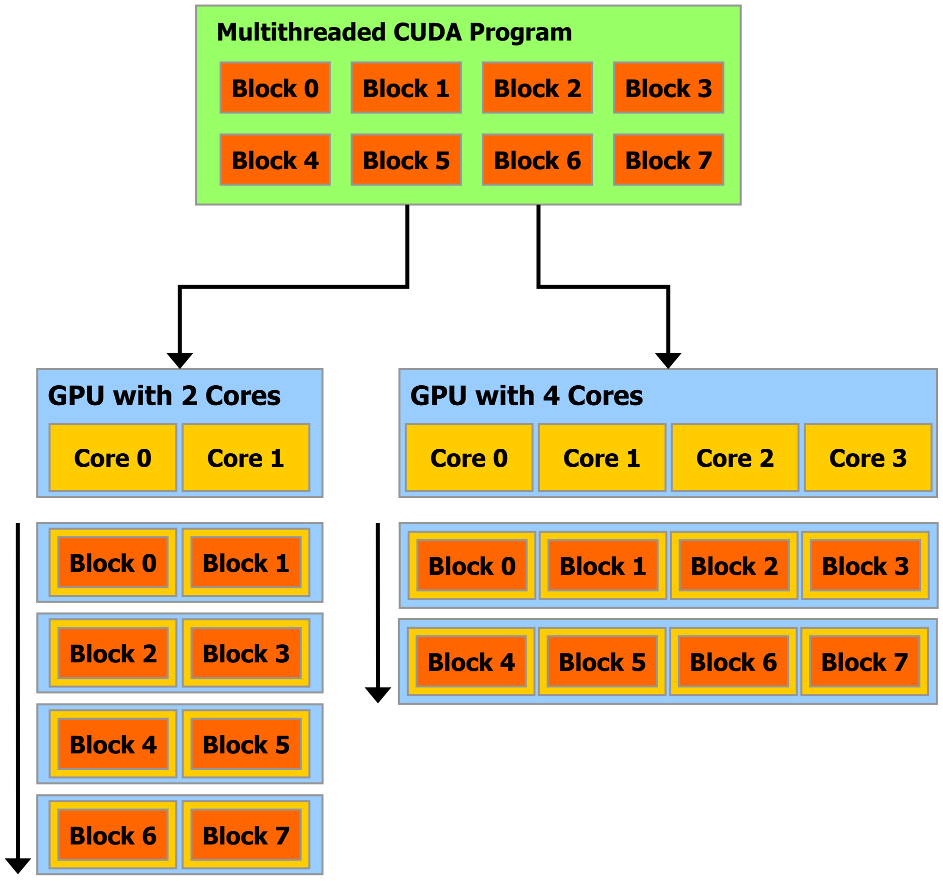
The advent of multicore CPUs and manycore GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moore's law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number 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 processor cores, in any order, concurrently or sequentially, so that a compiled CUDA program can execute on any number of processor cores as illustrated by Figure 1-4, and only the runtime system needs to know the physical processor count.

This scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of processors 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 multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

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++ constructs supported in device code.
- Appendix E lists the specific keywords and directives supported by **`nvcc`**.
- Appendix F gives more details on texture fetching.
- Appendix G gives the technical specifications of various devices, as well as more architectural details.

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 Section 3.2.

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.13). 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 512 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 or two-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.

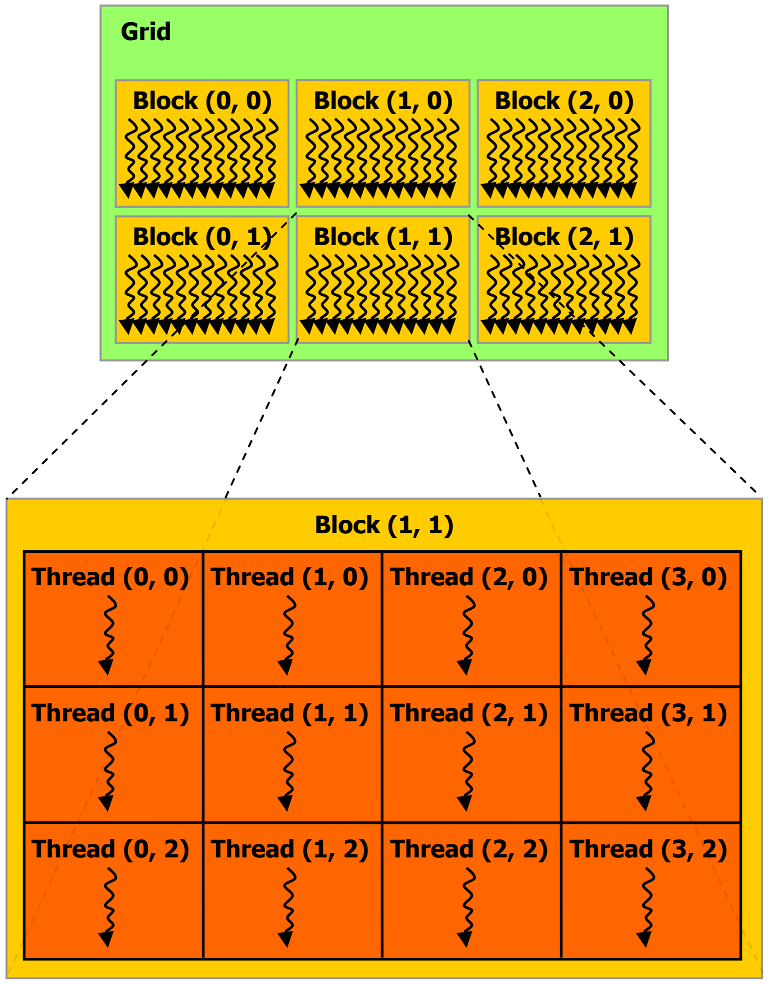


Figure 2-1. Grid of Thread Blocks

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 or two-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 3.2.2 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 3.2.4).

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

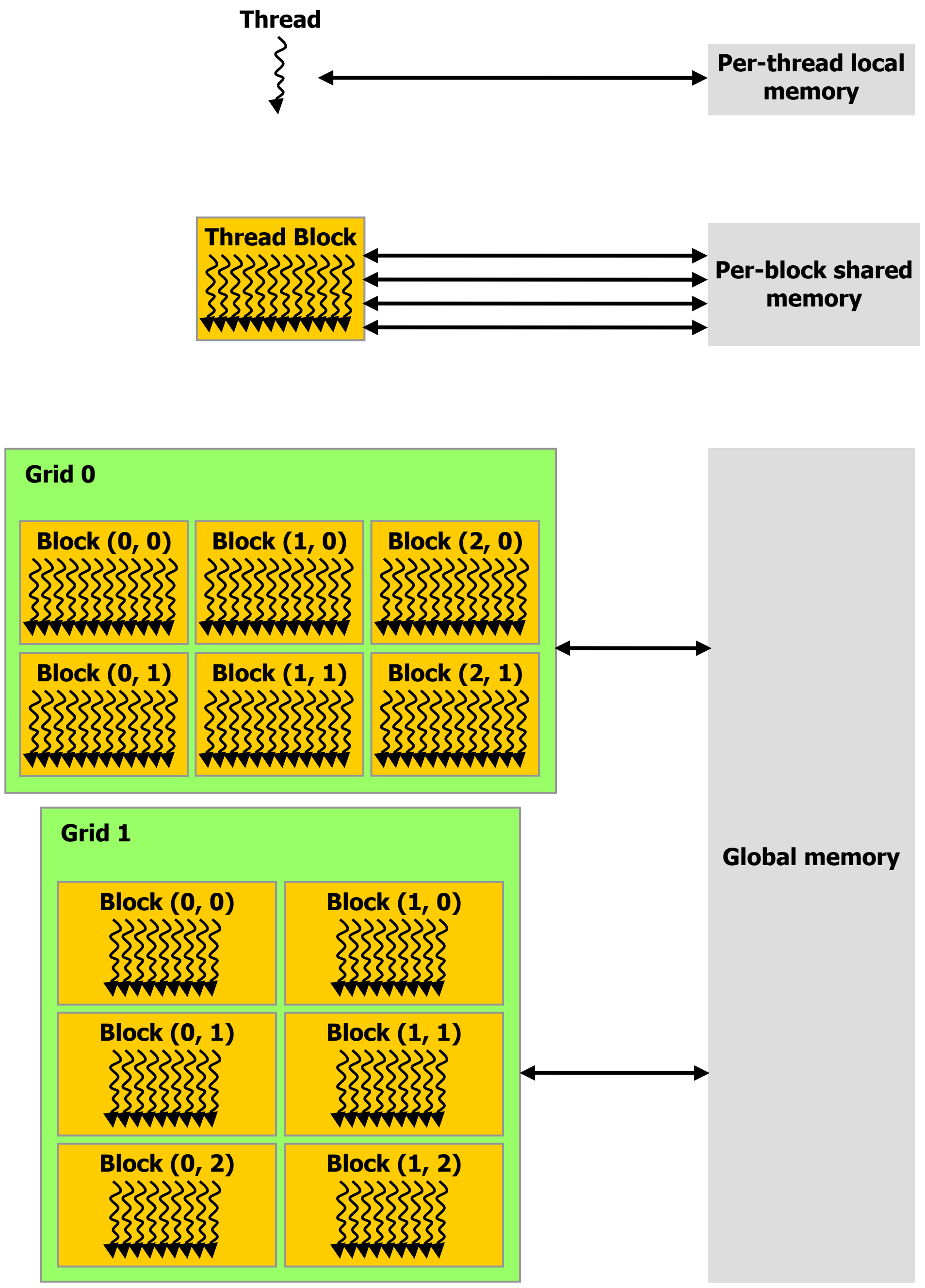
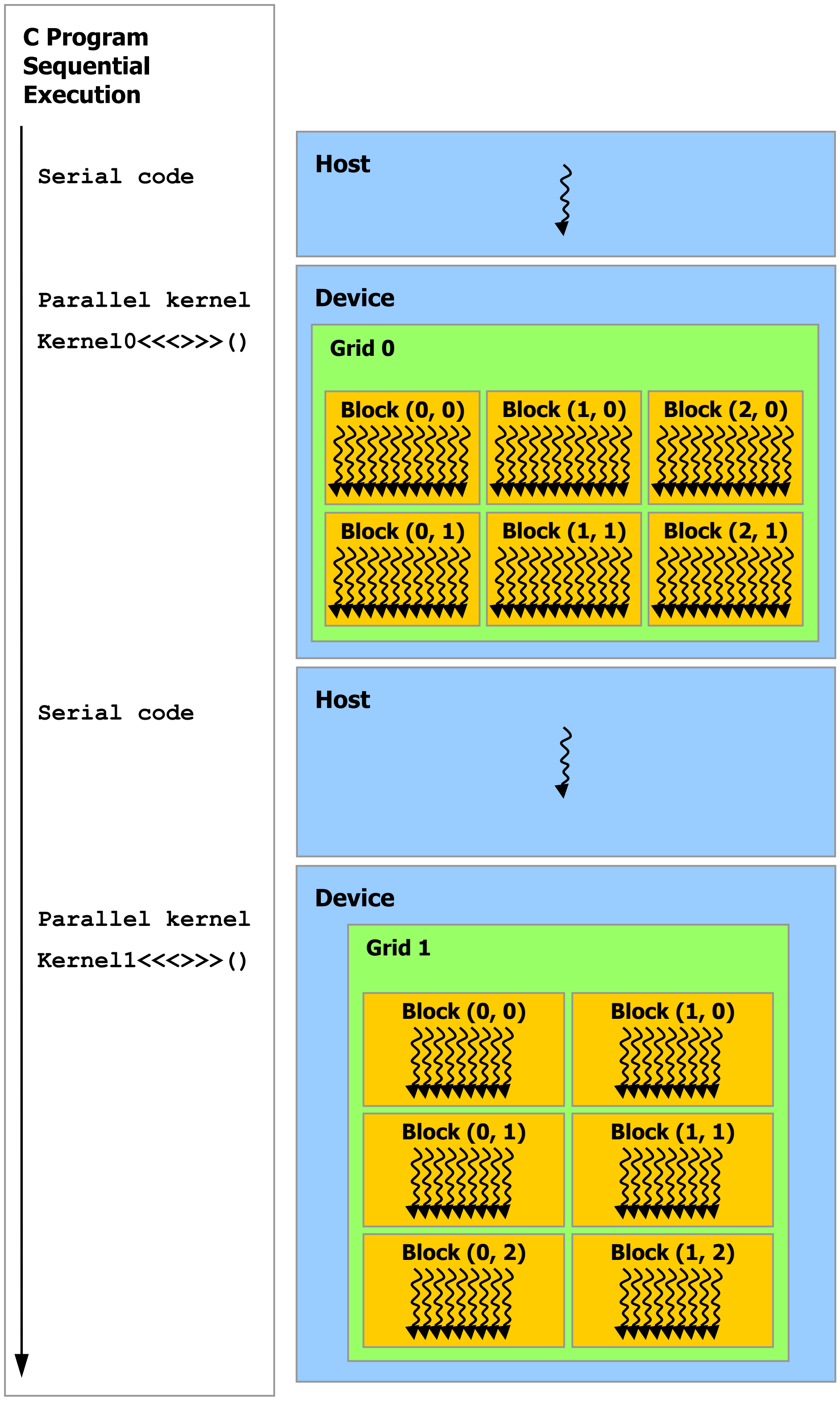


Figure 2-2. Memory Hierarchy

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 of devices based on the Fermi architecture is 2. Prior devices are all of compute capability 1.x (Their major revision number is 1).

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 G gives the technical specifications of each compute capability.

Chapter 3. Programming Interface

Two interfaces are currently supported to write CUDA programs: *CUDA C* and the *CUDA driver API*. An application typically uses either one or the other, but it can use both under the limitations described in Section 3.4.

CUDA C exposes the CUDA programming model as a minimal set of extensions to the C language. Any source file that contains some of these extensions must be compiled with **`nvcc`** as outlined in Section 3.1. These extensions 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.

The CUDA driver API is a lower-level C API that provides functions to load kernels as modules of CUDA binary or assembly code, to inspect their parameters, and to launch them. Binary and assembly codes are usually obtained by compiling kernels written in C.

CUDA C comes with a *runtime API* and both the runtime API and the driver API provide functions to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc.

The runtime API is built on top of the CUDA driver API. Initialization, context, and module management are all implicit and resulting code is more concise. CUDA C also supports device emulation, which facilitates debugging (see Section 3.2.8).

In contrast, the CUDA driver API requires more code, is harder to program and debug, but offers a better level of control and is language-independent since it handles binary or assembly code.

Section 3.2 continues the description of CUDA C started in Chapter 2. It also introduces concepts that are common to both CUDA C and the driver API: linear memory, CUDA arrays, shared memory, texture memory, page-locked host memory, device enumeration, asynchronous execution, interoperability with graphics APIs. Section 3.3 assumes knowledge of these concepts and describes how they are exposed by the driver API.

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

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 compiling the device code into an assembly form (*PTX* code) and/or binary form (*cubin* object). The generated 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 load and execute the *PTX* code or *cubin* object on the device using the CUDA driver API (see Section 3.3) and ignore the generated host code (if any);
- Or link to the generated host code; the generated host code includes the *PTX* code and/or *cubin* object as a global initialized data array and a translation of the **<<<…>>>** syntax introduced in Section 2.1 (and described in more details in Section B.13) into the necessary CUDA C runtime function calls to load and launch each compiled kernel.

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 allow 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.

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 compiled just-in-time for these devices.

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 or 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 detail 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.

`nvcc` also support specific keywords and directives detailed in Appendix E.

3.2 CUDA C

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. This section continues with an introduction to the runtime. A complete description of all extensions can be found in Appendix B and a complete description of the runtime in the CUDA reference manual.

The runtime is implemented in the **`cudart`** dynamic library and all its entry points are prefixed with **`cuda`**.

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.

Once the runtime has been initialized in a host thread, any resource (memory, stream, event, etc.) allocated via some runtime function call in the host thread is only valid within the context of the host thread. Therefore only runtime functions calls made by the host thread (memory copies, kernel launches, …) can operate on these resources. This is because a *CUDA context* (see Section 3.3.1) is created under the hood as part of initialization and made current to the host thread, and it cannot be made current to any other host thread.

On system with multiple devices, kernels are executed on device 0 by default as detailed in Section 3.2.3.

3.2.1 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 3.2.4.

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 compute capability 2.0, 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((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&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
float* devPtr;
int pitch;
cudaMallocPitch((void**)&devPtr, &pitch,
                width * sizeof(float), height);
MyKernel<<<100, 512>>>(devPtr, pitch);

// Device code
__global__ void MyKernel(float* devPtr, int pitch)
{
    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
cudaPitchedPtr devPitchedPtr;
cudaExtent extent = make_cudaExtent(64, 64, 64);
cudaMalloc3D(&devPitchedPtr, extent);
MyKernel<<<100, 512>>>(devPitchedPtr, extent);

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

As an example, the following code sample copies some host memory array to constant memory:

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

`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()`**.

3.2.2 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((void**)&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((void**)&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((void**)&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;
}
```

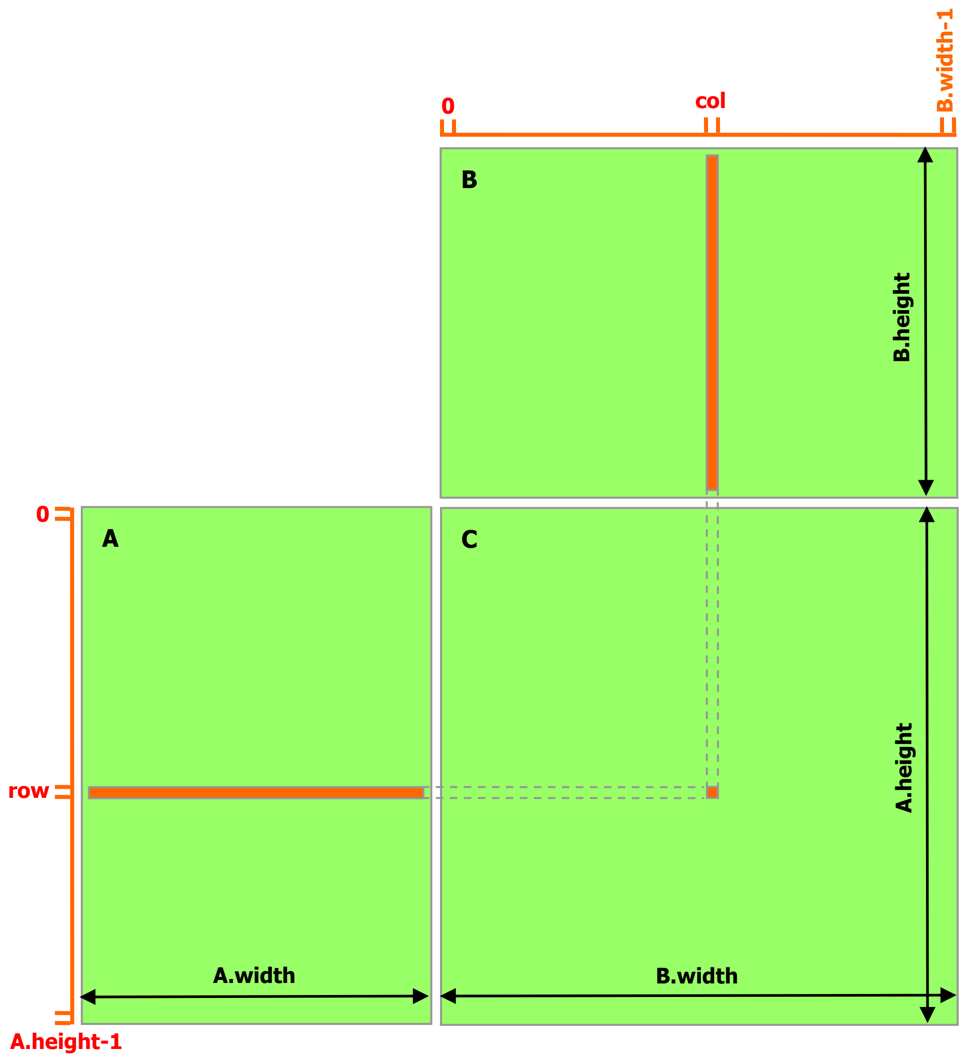


Figure 3-1. Matrix Multiplication without Shared Memory

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 line 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((void**)&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((void**)&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((void**)&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);
}
```

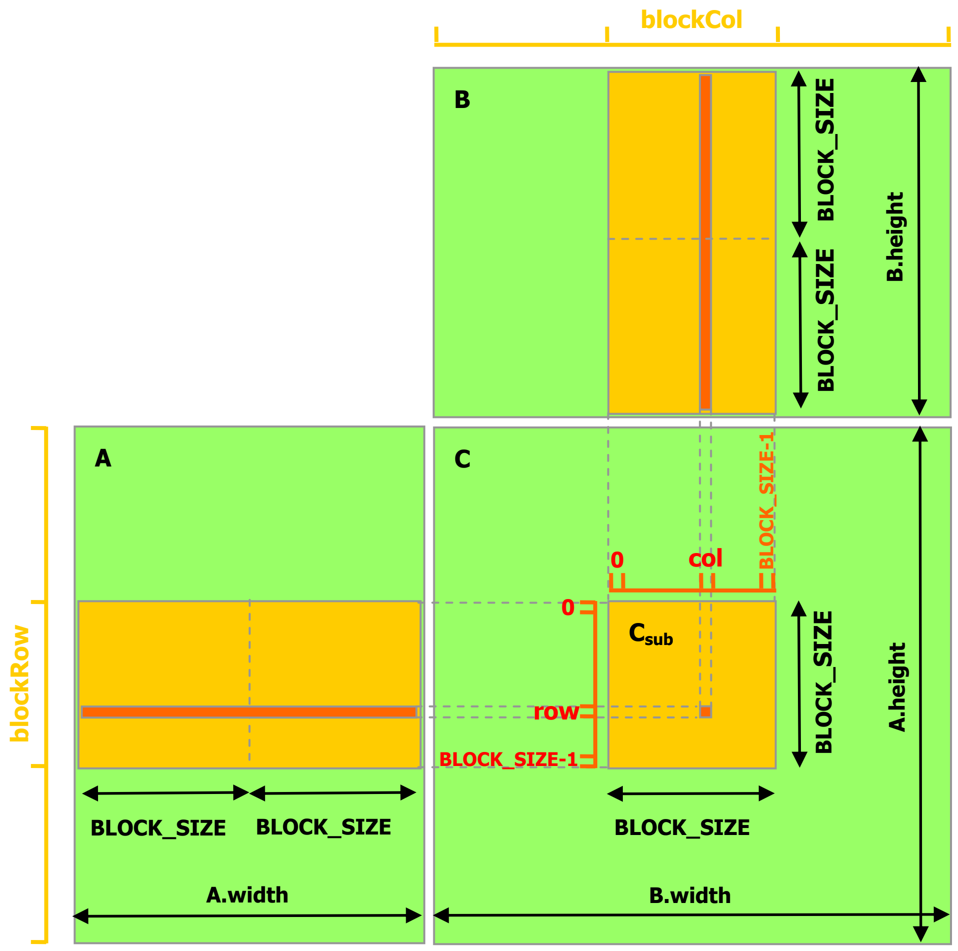


Figure 3-2. Matrix Multiplication with Shared Memory

3.2.3 Multiple Devices

A host system can have multiple devices. These devices can be enumerated, their properties can be queried, and one of them can be selected for kernel executions.

Several host threads can execute device code on the same device, but by design, a host thread can execute device code on only one device at any given time. As a consequence, multiple host threads are required to execute device code on multiple devices. Also, any CUDA resources created through the runtime in one host thread cannot be used by the runtime from another host thread.

The following code sample enumerates all devices in the system and retrieves their properties. It also determines the number of CUDA-enabled devices.

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
    if (dev == 0) {
        if (deviceProp.major == 9999 && deviceProp.minor == 9999)
            printf("There is no device supporting CUDA.\n");
        else if (deviceCount == 1)
            printf("There is 1 device supporting CUDA\n");
        else
            printf("There are %d devices supporting CUDA\n",
                   deviceCount);
    }
}
```

By default, the device associated to the host thread is implicitly selected as device 0 as soon as a non-device management runtime function is called (see Section 3.6 for exceptions). Any other device can be selected by calling **`cudaSetDevice()`** first. After a device has been selected, either implicitly or explicitly, any subsequent explicit call to **`cudaSetDevice()`** will fail up until **`cudaThreadExit()`** is called. **`cudaThreadExit()`** cleans up all runtime-related resources associated with the calling host thread. Any subsequent API call reinitializes the runtime.

3.2.4 Texture Memory

CUDA supports a subset of the texturing hardware that the GPU uses for graphics to access texture memory. Reading data from texture memory instead of global memory can have several performance benefits as described in Section 5.3.2.5.

Texture memory is read from kernels using device functions called *texture fetches*, described in Section B.8. 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 3.2.4.3, 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."

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.

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 (currently only supported through the driver API), 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.

3.2.4.1 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<Type, Dim, ReadMode> texRef;
```

where:

- **`Type`** 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;
- **`Dim`** specifies the dimensionality of the texture reference and is equal to 1, 2, or 3; **`Dim`** is an optional argument which defaults to 1;
- **`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`**.

3.2.4.2 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 using floating-point coordinates in the range [0, N) 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) instead of [0, N), so the same 64×32 texture would be addressed by normalized coordinates in the range [0, 1) in both the x and y dimensions. Normalized texture coordinates are a natural fit to some applications' requirements, if it is preferable for the texture coordinates to be independent of the texture size.

The addressing mode defines what happens when texture coordinates are out of range. When using unnormalized texture coordinates, texture coordinates outside the range [0, N) are clamped: Values below 0 are set to 0 and values greater or equal to N are set to N-1. Clamping is also the default addressing mode when using normalized texture coordinates: Values below 0.0 or above 1.0 are clamped to the range [0.0, 1.0). For normalized coordinates, the "wrap" addressing mode also may be specified. Wrap addressing is usually used when the texture contains a periodic signal. It uses only the fractional part of the texture coordinate; for example, 1.25 is treated the same as 0.25 and -1.25 is treated the same as 0.75.

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 and bilinear interpolation is performed for two-dimensional textures.

Appendix F gives more details on texture fetching.

3.2.4.3 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; if it is non-zero, all elements in the texture are addressed with texture coordinates in the range **`[0,1]`** rather than in the range **`[0,width-1]`**, **`[0,height-1]`**, or **`[0,depth-1]`** where **`width`**, **`height`**, and **`depth`** are the texture sizes;
- **`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, that is how out-of-range texture coordinates are handled; **`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 is equal to either **`cudaAddressModeClamp`**, in which case out-of-range texture coordinates are clamped to the valid range, or **`cudaAddressModeWrap`**, in which case out-of-range texture coordinates are wrapped to the valid range; **`cudaAddressModeWrap`** is 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 **cudaBindTextureToArray()**. **cudaUnbindTexture()** is used to unbind a texture reference.

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

- Using the low-level API:

```
texture<float, 2, cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, "texRef");
cudaChannelFormatDesc channelDesc =
                             cudaCreateChannelDesc<float>();
cudaBindTexture2D(0, texRefPtr, devPtr, &channelDesc,
                  width, height, pitch);
```

- Using the high-level API:

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaChannelFormatDesc channelDesc =
                             cudaCreateChannelDesc<float>();
cudaBindTexture2D(0, 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, 2, cudaReadModeElementType> texRef;
textureReference* texRefPtr;
cudaGetTextureReference(&texRefPtr, "texRef");
cudaChannelFormatDesc channelDesc;
cudaGetChannelDesc(&channelDesc, cuArray);
cudaBindTextureToArray(texRef, cuArray, &channelDesc);
```

- Using the high-level API:

```
texture<float, 2, 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

```
// 2D float texture
texture<float, 2, 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(tex, 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
    cudaBindTextureToArray(texRef, cuArray, channelDesc);

    // Allocate result of transformation in device memory
    float* output;
    cudaMalloc((void**)&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);
}
```

3.2.5 Page-Locked Host Memory

The runtime also provides functions to allocate and free *page-locked* (also known as *pinned*) host memory – as opposed to regular pageable host memory allocated by **`malloc()`**: **`cudaHostAlloc()`** and **`cudaFreeHost()`**.

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 3.2.6;
- 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 3.2.5.3;
- 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 3.2.5.2.

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, allocating 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.

3.2.5.1 Portable Memory

A block of page-locked memory can be used by any host threads, but by default, the benefits of using page-locked memory described above are only available for the thread that allocates it. To make these advantages available to all threads, it needs to be allocated by passing flag **`cudaHostAllocPortable`** to **`cudaHostAlloc()`**.

3.2.5.2 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 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.

3.2.5.3 Mapped Memory

On some devices, a block of page-locked host memory can also be mapped into the address space of the device by passing flag **`cudaHostAllocMapped`** to **`cudaHostAlloc()`**. Such a block has therefore two addresses: one in host memory and one in device memory. The host memory pointer is returned by **`cudaHostAlloc()`** and the device memory pointer can be retrieved using

cudaHostGetDevicePointer() and then used to access the block from within a kernel.

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 3.2.6.4) 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 3.2.6) to avoid any potential read-after-write, write-after-read, or write-after-write hazards.

A block of page-locked host memory can be allocated as both mapped and portable (see Section 3.2.5.1), in which case each host thread that needs to map the block to its device address space must call **cudaHostGetDevicePointer()** to retrieve a device pointer, as device pointers will generally differ from one host thread to the other.

To be able to retrieve the device pointer to any mapped page-locked memory within a given host thread, page-locked memory mapping must be enabled by calling **cudaSetDeviceFlags()** with the **cudaDeviceMapHost** flag before any other CUDA calls is performed by the thread. 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 whether a device supports mapped page-locked host memory or not by calling **cudaGetDeviceProperties()** and checking the **canMapHostMemory** property.

Note that atomic functions (Section 5.4.3B.10) operating on mapped page-locked memory are not atomic from the point of view of the host or other devices.

3.2.6 Asynchronous Concurrent Execution

3.2.6.1 Concurrent Execution between Host and Device

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

- Kernel launches;
- The functions that perform memory copies and are suffixed with **Async**;
- The functions that perform device ↔ device memory copies;
- The functions that set memory.

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 the CUDA debugger or the CUDA profiler, all launches are synchronous.

3.2.6.2 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 calling **`cudaGetDeviceProperties()`** and checking the **`deviceOverlap`** property. This capability is currently supported only for memory copies that do not involve CUDA arrays or 2D arrays allocated through **`cudaMallocPitch()`** (see Section 3.2.1).

3.2.6.3 Concurrent Kernel Execution

Some devices of compute capability 2.0 can execute multiple kernels concurrently. Applications may query this capability by calling **`cudaGetDeviceProperties()`** and checking the **`concurrentKernels`** property.

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

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.

3.2.6.4 Stream

Applications manage concurrency through *streams*. A stream is a sequence of commands 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).

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((void**)&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]);
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]);
cudaThreadSynchronize();
```

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`**. Processing **`hostPtr`** using two streams allows for the memory copies of one stream to overlap with the kernel execution of the other stream. **`hostPtr`** must point to page-locked host memory for any overlap to occur.

`cudaThreadSynchronize()` is called in the end to make sure all streams are finished before proceeding further. It forces the runtime to wait until all preceding device tasks in all streams have completed. **`cudaStreamSynchronize()`** forces the runtime to wait until all preceding commands in a stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device. **`cudaStreamQuery()`** provides applications with a way to know if all preceding commands in a stream have completed. To avoid unnecessary slowdowns, these functions are best used for timing purposes or to isolate a launch or memory copy that is failing.

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

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

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

Two commands from different streams cannot run concurrently if either a page-locked host memory allocation, a device memory allocation, a device memory set, a device ↔ device memory copy, or any CUDA command to stream 0 (including kernel launches and host ↔ device memory copies that do not specify any stream parameter) is called in-between them by the host thread.

Any operation that requires a dependency check to see if a streamed kernel launch is complete blocks all later kernel launches from any stream in the CUDA context until the 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.

Switching between the L1/shared memory configurations described in Section G.4.1 inserts a device-side synchronization barrier for all outstanding kernel launches.

3.2.6.5 Event

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 actually recorded. An event is recorded when all tasks – or optionally, all commands in a given stream –

preceding the event have completed. Events in stream zero are recorded after all preceding tasks/commands from all streams are completed by the device.

The following code sample creates two events:

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

These events can be used to time the code sample of the previous section the following way:

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

They are destroyed this way:

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

3.2.6.6 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.

3.2.7 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 3.2.7.1 and 3.2.7.2. 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 3.2.7.1 and 3.2.7.2 give specifics for each graphics API and some code samples.

3.2.7.1 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 and can therefore be bound to a texture reference and be read and written by kernels or via **`cudaMemcpy2D()`** calls. **`cudaGraphicsGLRegisterImage()`** supports all texture formats with a internal type of float (e.g. **`GL_RGBA_FLOAT32`**) and unnormalized integer (e.g. **`GL_RGBA8UI`**). It does not currently support normalized integer formats (e.g. **`GL_RGBA8`**). Please note that since **`GL_RGBA8UI`** is an OpenGL 3.0 texture format, it can only be written by shaders, not the fixed function pipeline.

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()
{
    // Explicitly set device
    cudaGLSetGLDevice(0);

    // Initialize OpenGL and GLUT
    ...
    glutDisplayFunc(display);

    // Create buffer object and register it with CUDA
    glGenBuffers(1, positionsVBO);
    glBindBuffer(GL_ARRAY_BUFFER, &vbo);
    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.

3.2.7.2 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. Moreover, the Direct3D device must be created with the **`D3DCREATE_HARDWARE_VERTEXPROCESSING`** flag.

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.

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));
}
```

3.2.8 Error Handling

All runtime functions return an error code, but for an asynchronous function (see Section 3.2.6), 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 **`cudaThreadSynchronize()`** (or by using any other synchronization mechanisms described in Section 3.2.6) and checking the error code returned by **`cudaThreadSynchronize()`**.

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). **`cudaGetLastError()`** returns this variable and resets it to **`cudaSuccess`**.

Kernel launches do not return any error code, so **`cudaGetLastError()`** must be called just after the kernel launch to retrieve any pre-launch errors. To ensure that any error returned by **`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 **`cudaGetLastError()`**.

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

3.2.9 Debugging using the Device Emulation Mode

CUDA-GDB can be used to debug devices of compute capability greater than 1.0 (see the CUDA-GDB user manual for supported platforms). The compiler and runtime also supports an emulation mode for the purpose of debugging that can be used even in the absence of any CUDA-enabled device. When compiling an application in this mode (using the **`-deviceemu`** option), the device code is compiled for and runs on the host, allowing the programmer to use the host's native debugging support to debug the application as if it were a host application. The preprocessor macro **`__DEVICE_EMULATION__`** is defined in this mode. All code for an application, including any libraries used, must be compiled consistently either for device emulation or for device execution. Linking code compiled for device emulation with code compiled for device execution causes the following runtime error to be returned upon initialization: **`cudaErrorMixedDeviceExecution`**.

When running an application in device emulation mode, the programming model is emulated by the runtime. For each thread in a thread block, the runtime creates a thread on the host. The programmer needs to make sure that:

- The host is able to run up to the maximum number of threads per block, plus one for the master thread.
- Enough memory is available to run all threads, knowing that each thread gets 256 KB of stack.

Many features provided through the device emulation mode make it a very effective debugging tool:

- By using the host's native debugging support programmers can use all features that the debugger supports, like setting breakpoints and inspecting data.
- Since device code is compiled to run on the host, the code can be augmented with code that cannot run on the device, like input and output operations to files or to the screen (**`printf()`**, etc.).
- Since all data resides on the host, any device- or host-specific data can be read from either device or host code; similarly, any device or host function can be called from either device or host code.
- In case of incorrect usage of the synchronization intrinsic function, the runtime detects dead lock situations.

Programmers must keep in mind that device emulation mode is emulating the device, not simulating it. Therefore, device emulation mode is very useful in finding algorithmic errors, but certain errors are hard to find:

- Race conditions are less likely to manifest themselves in device-emulation mode, since the number of threads executing simultaneously is much smaller than on an actual device.
- When dereferencing a pointer to global memory on the host or a pointer to host memory on the device, device execution almost certainly fails in some undefined way, whereas device emulation can produce correct results.
- Most of the time the same floating-point computation will not produce exactly the same result when performed on the device as when performed on the host in device emulation mode. This is expected since in general, all you need to get different results for the same floating-point computation are slightly different compiler options, let alone different compilers, different instruction sets, or different architectures.

  In particular, some host platforms store intermediate results of single-precision floating-point calculations in extended precision registers, potentially resulting in significant differences in accuracy when running in device emulation mode. When this occurs, programmers can try any of the following methods, none of which is guaranteed to work:

  - Declare some floating-point variables as volatile to force single-precision storage;
  - Use the **-ffloat-store** compiler option of **gcc**,
  - Use the **/Op** or **/fp** compiler options of the Visual C++ compiler,
  - Use **_FPU_GETCW()** and **_FPU_SETCW()** on Linux or **_controlfp()** on Windows to force single-precision floating-point computation for a portion of the code by surrounding it with

    ```
    unsigned int originalCW;
    _FPU_GETCW(originalCW);
    unsigned int cw = (originalCW & ~0x300) | 0x000;
    _FPU_SETCW(cw);
    ```

    or

    ```
    unsigned int originalCW = _controlfp(0, 0);
    _controlfp(_PC_24, _MCW_PC);
    ```

    at the beginning, to store the current value of the control word and change it to force the mantissa to be stored in 24 bits using, and with

    ```
    _FPU_SETCW(originalCW);
    ```

    or

    ```
    _controlfp(originalCW, 0xfffff);
    ```

    at the end, to restore the original control word.

  Also, for single-precision floating-point numbers, compute devices of compute capability 1.x do not support denormalized numbers (see Appendix G) as host platforms usually do. This can lead to dramatically different results between device emulation and device execution modes since some computation might produce a finite result in one case and an infinite result in the other.
- The warp size is equal to 1 in device emulation mode (see Section 4.1 for the definition of a warp). Therefore, the warp vote functions (described in Section B.11) produce different results than in device execution mode.

3.3 Driver API

The driver API 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 3-1.

Table 3-1. 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 references
Texture reference	CUtexref	Object that describes how to interpret texture memory data

The driver API is implemented in the **`nvcuda`** dynamic library and all its entry points are prefixed with **`cu`**.

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 3.3.1.

Within a CUDA context, kernels are explicitly loaded as *PTX* or binary objects by the host code as described in Section 3.3.2. 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 3.3.3.

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 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
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr;
    ptr = (void*)(size_t)A;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)B;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)C;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(N));
    cuParamSeti(vecAdd, offset, N);
    offset += sizeof(N);
    cuParamSetSize(vecAdd, offset);
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock - 1) / threadsPerBlock;
    cuFuncSetBlockShape(vecAdd, threadsPerBlock, 1, 1);
    cuLaunchGrid(vecAdd, blocksPerGrid, 1);

    ...
}
```

Full code can be found in the *vectorAddDrv* SDK code sample.

3.3.1 Context

A CUDA context is analogous to a CPU process. All resources and actions performed within the driver API are encapsulated inside a CUDA context, and the system automatically cleans up these resources when the context is destroyed. Besides objects such as modules and texture references, each context has its own distinct 32-bit 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 3-3.

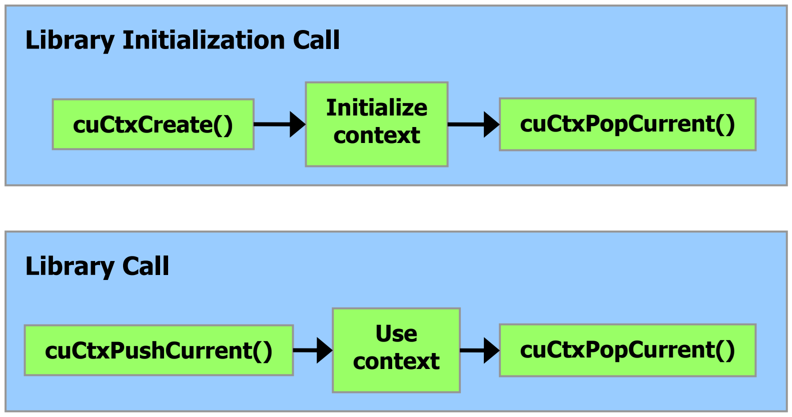


Figure 3-3. Library Context Management

3.3.2 Module

Modules are dynamically loadable packages of device code and data, akin to DLLs in Windows, that are output by **nvcc** (see Section 3.1). The names for all symbols, including functions, global variables, and texture 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;
CUptxas_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
}
```

3.3.3 Kernel Execution

cuFuncSetBlockShape() sets the number of threads per block for a given function, and how their threadIDs are assigned.

`cuFuncSetSharedSize()` sets the size of shared memory for the function.

The **`cuParam*()`** family of functions is used to specify the parameters that will be provided to the kernel the next time **`cuLaunchGrid()`** or **`cuLaunch()`** is invoked to launch the kernel.

The second argument of each of the **`cuParam*()`** functions specifies the offset of the parameter in the parameter stack. This offset must match the alignment requirement for the 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. Also, **`CUdeviceptr`** is an integer, but represents a pointer, so its alignment requirement is **`__alignof(void*)`**. The following code sample uses a macro to adjust the offset of each parameter to meet its alignment requirement.

```
#define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
int offset = 0;

int i;
ALIGN_UP(offset, __alignof(i));
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);

float4 f4;
ALIGN_UP(offset, 16); // float4's alignment is 16
cuParamSetv(cuFunction, offset, &f4, sizeof(f4));
offset += sizeof(f4);

char c;
ALIGN_UP(offset, __alignof(c));
cuParamSeti(cuFunction, offset, c);
offset += sizeof(c);

float f;
ALIGN_UP(offset, __alignof(f));
cuParamSeti(cuFunction, offset, f);
offset += sizeof(f);

CUdeviceptr dptr;
// void* should be used to determine CUdeviceptr's alignment
void* ptr = (void*)(size_t)dptr;
ALIGN_UP(offset, __alignof(ptr));
cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
offset += sizeof(ptr);

float2 f2;
ALIGN_UP(offset, 8); // float2's alignment is 8
cuParamSetv(cuFunction, offset, &f2, sizeof(f2));
offset += sizeof(f2);
```

```
cuParamSetSize(cuFunction, offset);

cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
cuLaunchGrid(cuFunction, gridWidth, gridHeight);
```

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;
```

Any parameter of type **myStruct** must therefore be passed using separate calls to **cuParam*()**, such as:

```
myStruct s;
int offset = 0;

cuParamSetv(cuFunction, offset, &s.f, sizeof(s.f));
offset += sizeof(s.f);

ALIGN_UP(offset, 16); // float4's alignment is 16
cuParamSetv(cuFunction, offset, &s.f4, sizeof(s.f4));
offset += sizeof(s.f4);
```

3.3.4 Device Memory

Linear memory is allocated using **cuMemAlloc()** or **cuMemAllocPitch()** and freed using **cuMemFree()**.

Here is the host code of the sample from Section 3.2.1 written using the driver API:

```
// Host code
int main()
{
    // Initialize
    if (cuInit(0) != CUDA_SUCCESS)
        exit (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 = 0;
    cuDeviceGet(&cuDevice, 0);

    // Create context
    CUcontext cuContext;
    cuCtxCreate(&cuContext, 0, cuDevice);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “VecAdd.ptx”);

    // Get function handle from module
    CUfunction vecAdd;
    cuModuleGetFunction(&vecAdd, cuModule, "VecAdd");

    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    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
    // h_A and h_B are input vectors stored in host memory
    cuMemcpyHtoD(d_A, h_A, size);
    cuMemcpyHtoD(d_B, h_B, size);

    // Invoke kernel
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) – 1) & ~((alignment) – 1)
    int offset = 0;
    void* ptr;
    ptr = (void*)(size_t)d_A;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_B;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_C;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(vecAdd, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    cuParamSetSize(VecAdd, offset);
    int threadsPerBlock = 256;
    int blocksPerGrid =
            (N + threadsPerBlock – 1) / threadsPerBlock;
    cuFuncSetBlockShape(vecAdd, threadsPerBlock, 1, 1);
    cuLaunchGrid(VecAdd, blocksPerGrid, 1);

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

    // Free device memory
    cuMemFree(d_A);
    cuMemFree(d_B);
    cuMemFree(d_C);
```

```
}
```

Linear memory can also be allocated through **`cuMemAllocPitch()`**. This function is recommended for allocations of 2D 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 **`cuMemcpy2D()`**). 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 (assuming cuModule has been loaded)
CUdeviceptr devPtr;
int pitch;
cuMemAllocPitch(&devPtr, &pitch,
                width * sizeof(float), height, 4);
CUfunction myKernel;
cuModuleGetFunction(&myKernel, cuModule, "MyKernel");
void* ptr = (void*)(size_t)devPtr;
cuParamSetv(myKernel, 0, &ptr, sizeof(ptr));
cuParamSetSize(myKernel, sizeof(ptr));
cuFuncSetBlockShape(myKernel, 512, 1, 1);
cuLaunchGrid(myKernel, 100, 1);

// Device code
__global__ void MyKernel(float* devPtr)
{
    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`** CUDA array of one 32-bit floating-point component:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = width;
desc.Height = height;
CUarray cuArray;
cuArrayCreate(&cuArray, &desc);
```

The reference manual lists all the various functions used to copy memory between linear memory allocated with **`cuMemAlloc()`**, linear memory allocated with **`cuMemAllocPitch()`**, and CUDA arrays.

The following code sample copies the 2D array to the CUDA array allocated in the previous code samples:

```
CUDA_MEMCPY2D copyParam;
memset(&copyParam, 0, sizeof(copyParam));
copyParam.dstMemoryType = CU_MEMORYTYPE_ARRAY;
copyParam.dstArray = cuArray;
copyParam.srcMemoryType = CU_MEMORYTYPE_DEVICE;
copyParam.srcDevice = devPtr;
copyParam.srcPitch = pitch;
copyParam.WidthInBytes = width * sizeof(float);
copyParam.Height = height;
cuMemcpy2D(&copyParam);
```

The following code sample copies some host memory array to constant memory:

```
__constant__ float constData[256];
float data[256];
CUdeviceptr devPtr;
unsigned int bytes;
cuModuleGetGlobal(&devPtr, &bytes, cuModule, "constData");
cuMemcpyHtoD(devPtr, data, bytes);
```

3.3.5 Shared Memory

The following code sample is the driver version of the host code of the sample from Section 3.2.2. Note how the **`Matrix`** type must be declared differently in host code since device pointers are represented as a handle of type **`CUdeviceptr`**.

In this sample, shared memory is statically allocated within the kernel as opposed to allocated at runtime through **`cuFuncSetSharedSize()`**.

```
// Matrices are stored in row-major order:
// M(row, col) = *(M.elements + row * M.stride + col)
typedef struct {
    int width;
    int height;
    int stride;
#ifdef __CUDACC__
    float* elements;
#else
    CUdeviceptr elements;
#endif
} 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);
    cuMemAlloc(&d_A.elements, size);
    cuMemcpyHtoD(d_A.elements, A.elements, size);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    cuMemAlloc(&d_B.elements, size);
    cuMemcpyHtoD(d_B.elements, B.elements, size);

    // 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);
    cuMemAlloc(&d_C.elements, size);
    // Invoke kernel (assuming cuModule has been loaded)
    CUfunction matMulKernel;
    cuModuleGetFunction(&matMulKernel, cuModule, "MatMulKernel");
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr;
    ptr = (void*)(size_t)d_A;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(matMulKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_B;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(matMulKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)d_C;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(matMulKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    cuParamSetSize(matMulKernel, offset);
    cuFuncSetBlockShape(matMulKernel, BLOCK_SIZE, BLOCK_SIZE, 1);
    cuLaunchGrid(matMulKernel,
                 B.width / dimBlock.x, A.height / dimBlock.y);

    // Read C from device memory
    cuMemcpyDtoH(C.elements, d_C.elements, size);

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

3.3.6 Multiple Devices

`cuDeviceGetCount()` and **`cuDeviceGet()`** provide a way to enumerate the devices present in the system and other functions (described in the reference manual) to retrieve their properties:

```
int deviceCount;
cuDeviceGetCount(&deviceCount);
int device;
for (int device = 0; device < deviceCount; ++device) {
    CUdevice cuDevice;
    cuDeviceGet(&cuDevice, device);
    int major, minor;
    cuDeviceComputeCapability(&major, &minor, cuDevice);
}
```

3.3.7 Texture Memory

Texure binding is done using **`cuTexRefSetAddress()`** for linear memory and **`cuTexRefSetArray()`** for CUDA arrays.

If a module **cuModule** contains some texture reference **texRef** defined as

```
texture<float, 2, cudaReadModeElementType> texRef;
```

the following code sample retrieves **texRef**'s handle:

```
CUtexref cuTexRef;
cuModuleGetTexRef(&cuTexRef, cuModule, “texRef”);
```

The following code sample binds **texRef** to some linear memory pointed to by **devPtr**:

```
CUDA_ARRAY_DESCRIPTOR desc;
cuTexRefSetAddress2D(cuTexRef, &desc, devPtr, pitch);
```

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

```
cuTexRefSetArray(cuTexRef, cuArray, CU_TRSA_OVERRIDE_FORMAT);
```

The reference manual lists various functions used to set address mode, filter mode, format, and other flags for some texture reference. 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 is the driver version of the host code of the sample from Section 3.2.4.3.

```
// Host code
int main()
{
    // Allocate CUDA array in device memory
    CUarray cuArray;
    CUDA_ARRAY_DESCRIPTOR desc;
    desc.Format      = CU_AD_FORMAT_FLOAT;
    desc.NumChannels = 1;
    desc.Width       = width;
    desc.Height      = height;
    cuArrayCreate(&cuArray, &desc);

    // Copy to device memory some data located at address h_data
    // in host memory
    CUDA_MEMCPY2D copyParam;
    memset(&copyParam, 0, sizeof(copyParam));
    copyParam.dstMemoryType = CU_MEMORYTYPE_ARRAY;
    copyParam.dstArray      = cuArray;
    copyParam.srcMemoryType = CU_MEMORYTYPE_HOST;
    copyParam.srcHost       = h_data;
    copyParam.srcPitch      = width * sizeof(float);
    copyParam.WidthInBytes  = copyParam.srcPitch;
    copyParam.Height        = height;
    cuMemcpy2D(&copyParam);

    // Set texture parameters
    CUtexref texRef;
    cuModuleGetTexRef(&texRef, cuModule, "texRef"));
    cuTexRefSetAddressMode(texRef, 0, CU_TR_ADDRESS_MODE_WRAP);
    cuTexRefSetAddressMode(texRef, 1, CU_TR_ADDRESS_MODE_WRAP);
    cuTexRefSetFilterMode(texRef, CU_TR_FILTER_MODE_LINEAR);
    cuTexRefSetFlags(texRef, CU_TRSF_NORMALIZED_COORDINATES);
    cuTexRefSetFormat(texRef, CU_AD_FORMAT_FLOAT, 1);
    // Bind the array to the texture
    cuTexRefSetArray(texRef, cuArray, CU_TRSA_OVERRIDE_FORMAT);

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

    // Invoke kernel (assuming cuModule has been loaded)
    CUfunction transformKernel;
    cuModuleGetFunction(&transformKernel,
                        cuModule, "transformKernel");
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr = (void*)(size_t)output;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(transformKernel, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(transformKernel, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(transformKernel, offset, height);
    offset += sizeof(height);
    ALIGN_UP(offset, __alignof(angle));
    cuParamSetf(transformKernel, offset, angle);
    offset += sizeof(angle);
    cuParamSetSize(transformKernel, offset));
    cuParamSetTexRef(transformKernel,
                     CU_PARAM_TR_DEFAULT, texRef);
    cuFuncSetBlockShape(transformKernel, 16, 16, 1);
    cuLaunchGrid(transformKernel,
                 (width  + dimBlock.x - 1) / dimBlock.x,
                 (height + dimBlock.y - 1) / dimBlock.y);

    // Free device memory
    cuArrayDestroy(cuArray);
    cuMemFree(output);
}
```

3.3.8 Page-Locked Host Memory

Page-locked host memory can be allocated using **`cuMemHostAlloc()`** with optional mutually non-exclusive flags:

- **`CU_MEMHOSTALLOC_PORTABLE`** to allocate memory that is portable across CUDA contexts (see Section 3.2.5.1) 3.2.5.2;
- **`CU_MEMHOSTALLOC_WRITECOMBINED`** to allocate memory as write-combining (see Section 3.2.5.2);
- **`CU_MEMHOSTALLOC_DEVICEMAP`** to allocate mapped page-locked memory (see Section 3.2.5.3).

Page-locked host memory is freed using **`cuMemFreeHost()`**.

Page-locked memory mapping is enabled for a CUDA context by creating the context with the **CU_CTX_MAP_HOST** flag and device pointers to mapped page-locked memory are retrieved using **cuMemHostGetDevicePointer()**.

Applications may query whether a device supports mapped page-locked host memory or not by checking the **CU_DEVICE_ATTRIBUTE_CAN_MAP_HOST_MEMORY** attribute using **cuDeviceGetAttribute()**.

3.3.9 Asynchronous Concurrent Execution

Applications may query if a device can perform copies between page-locked host memory and device memory concurrently with kernel execution by checking the **CU_DEVICE_ATTRIBUTE_GPU_OVERLAP** attribute using **cuDeviceGetAttribute()**.

Applications may query if a device supports multiple kernels running concurrently by checking the **CU_DEVICE_ATTRIBUTE_CONCURRENT_KERNELS** attribute using **cuDeviceGetAttribute()**.

3.3.9.1 Stream

The driver API provides functions similar to the runtime API to manage streams. The following code sample is the driver version of the code sample from Section 3.2.6.4.

```
CUstream stream[2];
for (int i = 0; i < 2; ++i)
    cuStreamCreate(&stream[i], 0);
float* hostPtr;
cuMemAllocHost((void**)&hostPtr, 2 * size);
```

```
for (int i = 0; i < 2; ++i)
    cuMemcpyHtoDAsync(inputDevPtr + i * size, hostPtr + i * size,
                      size, stream[i]);
for (int i = 0; i < 2; ++i) {
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr;
    ptr = (void*)(size_t)outputDevPtr;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)inputDevPtr;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(size));
    cuParamSeti(cuFunction, offset, size);
    offset += sizeof(int);
    cuParamSetSize(cuFunction, offset);
    cuFuncSetBlockShape(cuFunction, 512, 1, 1);
    cuLaunchGridAsync(cuFunction, 100, 1, stream[i]);
}
```

```
for (int i = 0; i < 2; ++i)
    cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size,
                      size, stream[i]);
cuCtxSynchronize();
```

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

3.3.9.2 Event Management

The driver API provides functions similar to the runtime API to manage events. The following code sample is the driver version of the code sample from Section 3.2.6.5.

```
CUevent start, stop;
cuEventCreate(&start);
cuEventCreate(&stop);
```

```
cuEventRecord(start, 0);
for (int i = 0; i < 2; ++i)
    cuMemcpyHtoDAsync(inputDevPtr + i * size, hostPtr + i * size,
                      size, stream[i]);
for (int i = 0; i < 2; ++i) {
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr;
    ptr = (void*)(size_t)outputDevPtr;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ptr = (void*)(size_t)inputDevPtr;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(cuFunction, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(size));
    cuParamSeti(cuFunction, offset, size);
    offset += sizeof(size);
    cuParamSetSize(cuFunction, offset);
    cuFuncSetBlockShape(cuFunction, 512, 1, 1);
    cuLaunchGridAsync(cuFunction, 100, 1, stream[i]);
}
for (int i = 0; i < 2; ++i)
    cuMemcpyDtoHAsync(hostPtr + i * size, outputDevPtr + i * size,
                      size, stream[i]);
cuEventRecord(stop, 0);
cuEventSynchronize(stop);
float elapsedTime;
cuEventElapsedTime(&elapsedTime, start, stop);
```

They are destroyed this way:

```
cuEventDestroy(start);
cuEventDestroy(stop);
```

3.3.9.3 Synchronous Calls

Whether the host thread will yield, block, or spin on a synchronous function call can be specified by calling **`cuCtxCreate()`** with some specific flags as described in the reference manual.

3.3.10 Graphics Interoperability

The driver API provides functions similar to the runtime API to manage graphics interoperability.

A resource must be registered to CUDA before it can be mapped using the functions mentioned in Sections 3.3.10.1 and 3.3.10.2. These functions return a CUDA graphics resource of type **`CUgraphicsResource`**. Registering a resource is potentially high-overhead and therefore typically called only once per resource. A CUDA graphics resource is unregistered using **`cuGraphicsUnregisterResource()`**.

Once a resource is registered to CUDA, it can be mapped and unmapped as many times as necessary using **`cuGraphicsMapResources()`** and **`cuGraphicsUnmapResources()`**. **`cuGraphicsResourceSetMapFlags()`** 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 **`cuGraphicsResourceGetMappedPointer()`** for buffers and **`cuGraphicsSubResourceGetMappedArray()`** for CUDA arrays.

Accessing a resource through OpenGL or Direct3D while it is mapped to CUDA produces undefined results.

Sections 3.3.10.1 and 3.3.10.2 give specifics for each graphics API and some code samples.

3.3.10.1 OpenGL Interoperability

Interoperability with OpenGL requires that the CUDA context be specifically created using **`cuGLCtxCreate()`** instead of **`cuCtxCreate()`**.

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 **`cuGraphicsGLRegisterBuffer()`**. A texture or renderbuffer object is registered using **`cuGraphicsGLRegisterImage()`**. The same restrictions described in Section 3.2.7.1 apply.

The following code sample is the driver version of the code sample from Section 3.2.7.1.

```
CUfunction createVertices;
GLuint positionsVBO;
struct cudaGraphicsResource* positionsVBO_CUDA;

int main()
{
    // Initialize driver API
    ...
    // Get handle for device 0
    CUdevice cuDevice = 0;
    cuDeviceGet(&cuDevice, 0);

    // Create context
    CUcontext cuContext;
    cuGLCtxCreate(&cuContext, 0, cuDevice);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, "createVertices.ptx");

    // Get function handle from module
    cuModuleGetFunction(&createVertices,
                        cuModule, "createVertices");

    // Initialize OpenGL and GLUT
    ...
    glutDisplayFunc(display);

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

    // Launch rendering loop
    glutMainLoop();
}

void display()
{
    // Map OpenGL buffer object for writing from CUDA
    CUdeviceptr positions;
    cuGraphicsMapResources(1, &positionsVBO_CUDA, 0);
    size_t num_bytes;
    cuGraphicsResourceGetMappedPointer((void**)&positions,
                                       &num_bytes,
                                       positionsVBO_CUDA));

    // Execute kernel
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices,
                        threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices,
                 width / threadsPerBlock, height / threadsPerBlock);

    // Unmap buffer object
    cuGraphicsUnmapResources(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()
{
    cuGraphicsUnregisterResource(positionsVBO_CUDA);
    glDeleteBuffers(1, &positionsVBO);
}
```

On Windows and for Quadro GPUs, **`cuWGLGetDevice()`** can be used to retrieve the CUDA device associated to the handle returned by **`wglEnumGpusNV()`**.

3.3.10.2 Direct3D Interoperability

Interoperability with Direct3D requires that the Direct3D device be specified when the CUDA context is created. This is done by creating the CUDA context using **`cuD3D9CtxCreate()`** (resp. **`cuD3D10CtxCreate()`**) instead of **`cuCtxCreate()`**.

The Direct3D resources that may be mapped into the address space of CUDA are Direct3D buffers, textures, and surfaces. These resources are registered using **`cuGraphicsD3D9RegisterResource()`**, **`cuGraphicsD3D10RegisterResource()`**, and **`cuGraphicsD3D11RegisterResource()`**.

The following code sample is the driver version of the host code of the sample from Section 3.2.7.2.

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 (cuD3D9GetDevice(&dev, adapterId.DeviceName)
            == cudaSuccess)
            break;
    }

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

    // Initialize driver API
    ...

    // Create context
    CUdevice cuDevice;
    CUcontext cuContext;
    cuD3D9CtxCreate(&cuContext, &cuDevice, 0, &device);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, "createVertices.ptx");

    // Get function handle from module
    cuModuleGetFunction(&createVertices,
                        cuModule, "createVertices");

    // 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);
    cuGraphicsD3D9RegisterResource(&positionsVB_CUDA,
                                   positionsVB,
                                   cudaGraphicsRegisterFlagsNone);
    cuGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                  cudaGraphicsMapFlagsWriteDiscard);

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

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

    // Execute kernel
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices,
                        threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices,
                 width / threadsPerBlock, height / threadsPerBlock);

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

    // Draw and present
    ...
}

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

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 (cuD3D10GetDevice(&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();

    // Initialize driver API
    ...

    // Create context
    CUdevice cuDevice;
    CUcontext cuContext;
    cuD3D10CtxCreate(&cuContext, &cuDevice, 0, &device);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “createVertices.ptx”);

    // Get function handle from module
    cuModuleGetFunction(&createVertices,
                        cuModule, "createVertices");

    // 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);
    cuGraphicsD3D10RegisterResource(&positionsVB_CUDA,
                               positionsVB,
                               cudaGraphicsRegisterFlagsNone);
    cuGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                      cudaGraphicsMapFlagsWriteDiscard);

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

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

    // Execute kernel
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices,
                        threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices,
              width / threadsPerBlock, height / threadsPerBlock);

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

    // Draw and present
    ...
}

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

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 (cuD3D11GetDevice(&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();

    // Initialize driver API
    ...

    // Create context
    CUdevice cuDevice;
    CUcontext cuContext;
    cuD3D11CtxCreate(&cuContext, &cuDevice, 0, &device);

    // Create module from binary file
    CUmodule cuModule;
    cuModuleLoad(&cuModule, “createVertices.ptx”);

    // Get function handle from module
    cuModuleGetFunction(&createVertices,
                        cuModule, "createVertices");

    // 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      = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags = 0;
    bufferDesc.MiscFlags      = 0;
    device->CreateBuffer(&bufferDesc, 0, &positionsVB);
    cuGraphicsD3D11RegisterResource(&positionsVB_CUDA,
                                    positionsVB,
                                    cudaGraphicsRegisterFlagsNone);
    cuGraphicsResourceSetMapFlags(positionsVB_CUDA,
                                  cudaGraphicsMapFlagsWriteDiscard);

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

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

    // Execute kernel
    #define ALIGN_UP(offset, alignment) \
      (offset) = ((offset) + (alignment) - 1) & ~((alignment) - 1)
    int offset = 0;
    void* ptr = (void*)(size_t)positions;
    ALIGN_UP(offset, __alignof(ptr));
    cuParamSetv(createVertices, offset, &ptr, sizeof(ptr));
    offset += sizeof(ptr);
    ALIGN_UP(offset, __alignof(time));
    cuParamSetf(createVertices, offset, time);
    offset += sizeof(time);
    ALIGN_UP(offset, __alignof(width));
    cuParamSeti(createVertices, offset, width);
    offset += sizeof(width);
    ALIGN_UP(offset, __alignof(height));
    cuParamSeti(createVertices, offset, height);
    offset += sizeof(height);
    cuParamSetSize(createVertices, offset);
    int threadsPerBlock = 16;
    cuFuncSetBlockShape(createVertices,
                        threadsPerBlock, threadsPerBlock, 1);
    cuLaunchGrid(createVertices,
                 width / threadsPerBlock, height / threadsPerBlock);

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

    // Draw and present
    ...
}

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

3.3.11 Error Handling

All driver functions return an error code, but for an asynchronous function (see Section 3.2.6), 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 **`cuCtxSynchronize()`** (or by using any other synchronization mechanisms described in Section 3.3.9) and checking the error code returned by **`cuCtxSynchronize()`**.

3.4 Interoperability between Runtime and Driver APIs

An application can mix runtime API code with driver API code within the limits described in this section.

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), **`cuCtxAttach()`** 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((void**)&d_data, size);
devPtr = (CUdeviceptr)(size_t)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.

The following features are not supported for applications that mix runtime API code with driver API code:

- Device emulation,
- Context stack manipulation via **`cuCtxPopCurrent()`** and **`cuCtxPushCurrent()`** (see Section 3.3.1).

3.5 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 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 driver releases as illustrated in Figure 3-4. 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 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 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, …).

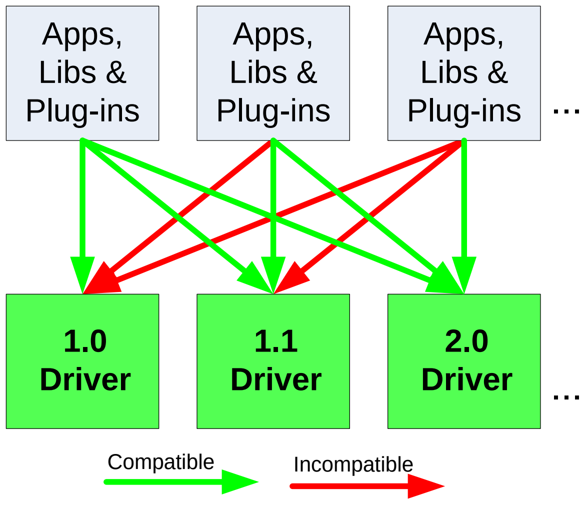


Figure 3-4. The Driver API is Backward, but Not Forward Compatible

3.6 Compute Modes

On Tesla solutions running 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 Linux 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* compute mode: Only one host thread can use the device at any given time.
- *Prohibited* compute mode: No host thread can use 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 prohibited compute mode or in exclusive compute mode and used by another host thread. **`cudaSetValidDevices()`** can be used to set a device from a prioritized list of devices.

Applications may query the compute mode of a device by calling **`cudaGetDeviceProperties()`** and checking the **`computeMode`** property or checking the **`CU_DEVICE_COMPUTE_MODE`** attribute using **`cuDeviceGetAttribute()`**.

3.7 Mode Switches

GPUs 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.

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. To maximize utilization of its functional units, it leverages thread-level parallelism by using hardware multithreading as detailed in Section 4.2, more so than instruction-level parallelism within a single thread (instructions are pipelined, but unlike CPU cores they are executed in order 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 G.3.1 and G.4.1 provide the specifics for devices of compute capabilities 1.x and 2.0, 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 that get 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 G.3.2, G.3.3, G.4.2, and G.4.3) and which thread performs the final write is undefined.

If an atomic instruction (see Section B.10) 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. Switching from one execution context to another therefore has no cost, and at every instruction issue time, the warp scheduler selects a warp that has threads ready to execute (*active threads*) and issues the next 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 G. 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 2.0:

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

- 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_T is the thread allocation granularity, equal to 256 for devices of compute capability 1.0 and 1.1, and 512 for devices of compute capability 1.2 and 1.3, and 64 for devices of compute capability 2.0.

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 and 128 for devices of compute capability 2.0.

4.3 Multiple Devices

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. Because of this, allocations may fail earlier than otherwise expected.

Second, when a Direct3D application runs in SLI Alternate Frame Rendering mode, the Direct3D device(s) created by that application can be used for CUDA-Direct3D interoperability (i.e., passed as a parameter to **`cudaD3D[9|10]SetDirect3DDevice()`** when using the runtime API), but only one CUDA device can be created at a time from one of these Direct3D devices. This CUDA device only executes the CUDA work on one of the GPUs in the SLI configuration. As a consequence, real interoperability only happens with the copy of a Direct3D resource in that GPU (note: in AFR mode Direct3D resources that must be in GPU memory are duplicated in the GPU memory of each GPU in the SLI configuration). In some cases this is not the desired behavior and an application may need to forfeit use of the CUDA-Direct3D interoperability API and manually copy the output of its CUDA work to Direct3D resources using the existing CUDA and Direct3D API.

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 3.2.6. 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 2.0, 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 3.2.6.

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 and issues the next 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 *latency*, and full utilization is achieved when the warp scheduler always has some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when the latency of each warp is completely "hidden" by other warps. How many instructions are required to hide latency depends on the instruction throughput. For example, to hide a latency of L clock cycles with basic single-precision floating-point arithmetic instructions (scheduled on CUDA cores):

- $L/4$ (rounded up to nearest integer) instructions are required for devices of compute capability 1.x since a multiprocessor issues one such instruction per warp over 4 clock cycles, as mentioned in Section G.3.1,
- $L/2$ (rounded up to nearest integer) instructions are required for devices of compute capability 2.0 since a multiprocessor issues the two instructions for a pair of warps over 2 clock cycles, as mentioned in Section G.4.1.

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

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 scheduler must schedule instructions for different warps during that time. Execution time varies depending on the instruction, but it is typically about 22 clock cycles, which translates to 6 warps for devices of compute capability 1.x and 11 warps for devices of compute capability 2.0.

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 scheduler busy during such high latency periods depends on the kernel code; 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 10, for example, then to hide latencies of about 600 clock cycles, about 15 warps are required for devices of compute capability 1.x and about 30 for devices of compute capability 2.0.

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.13), 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 G for various compute capabilities).

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 is 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.14.

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).

Register, local, shared, and constant memory usages are reported by the compiler when compiling with the **`--ptxas-options=-v`** option. Note that 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 number of threads per block should be chosen as a multiple of the warp size to avoid wasting computing resources with under-populated warps if possible.

As mentioned above, 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 or driver API (see reference manual).

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 2.0, 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 3.2.2, 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 accesses are data-dependent), a traditional hardware-managed cache is more appropriate to exploit data locality. As mentioned in Section G.4.1, for devices of compute capability 2.0, 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 3.2.5.

In addition, when using mapped page-locked memory (Section 3.2.5.3), 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 whether a device is integrated or not by calling **`cudaGetDeviceProperties()`** and checking the **`integrated`** property or checking the **`CU_DEVICE_ATTRIBUTE_INTEGRATED`** attribute using **`cuDeviceGetAttribute()`**.

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 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 2.0, the memory transactions are cached, so data locality is exploited to reduce impact on throughput. Sections G.3.2 and G.4.2 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 G.3.2 and G.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.2.5. 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 2.0, local memory accesses are always cached in L1 and L2 in the same way as global memory accesses (see Section G.4.2).

5.3.2.3 Shared Memory

Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing shared memory is fast as long as there are no bank conflicts between the threads, as detailed below.

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 G.3.3 and G.4.3 for devices of compute capability 1.x and 2.0, 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 G.3.1 and G.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 Memory

The texture memory space resides in device memory and is cached in texture cache, so a texture fetch 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 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 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 (this is less likely for devices of compute capability 2.0 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 3.2.4.1).

However, within the same kernel call, the texture cache is not kept coherent with respect to global memory writes, so that any texture fetch to an address that has been written to via a global write in the same kernel call returns undefined data. In other words, a thread can safely read via texture some 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. This is only relevant when fetching from linear memory as a kernel cannot write to CUDA arrays in any case.

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 E.3.

In this section, throughputs are given in number of operations per clock cycle per multiprocessor. For a warp size of 32, one instruction results in 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. For devices of compute capability 2.0, two different warps execute half of the operations each clock cycle (see Section G.4.1).

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

	Compute Capability 1.x	**Compute Capability 2.0**
32-bit floating-point add, multiply, multiply-add	8	32
64-bit floating-point add, multiply, multiply-add	1	16
32-bit integer add, logical operation, shift, compare	8	32
24-bit integer multiply (**`__[u]mul24(x,y)`**)	8	Multiple instructions
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	32
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (**`__log2f`**), base-2 exponential (**`exp2f`**), sine (**`__sinf`**), cosine (**`__cosf`**)	2	4
Type conversions	8	32

Other instructions and functions are implemented on top of the native instructions. The implementation may be different for devices of compute capability 1.x and devices of compute capability 2.0, 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 2.0.

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. Therefore, its throughput is 1 operation per clock cycle for devices of compute capability 1.x and 2 operations per clock cycle for devices of compute capability 2.0.

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 (see Section C.2.3). 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 2.0, 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 particularly costly and should be avoided if possible or replaced with bitwise operations whenever possible: 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`** (see Section C.2.3) 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 2.0 and **`__brevll`** and **`__popcll`** to just a few.

`__clz`, **`__clzll`**, **`__ffs`**, and **`__ffsll`** (see Section C.2.3) compile to fewer instructions for devices of compute capability 2.0 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 E.2).

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 and 16 operations per clock cycle for devices of compute capability 2.0.

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, both calls to **__syncthreads()** are required to get the expected result (i.e. **result[i] = 2 * myArray[i]** for **i > 0**). Without synchronization, any of the two references to **myArray[tid]** could return either 2 or the value initially stored in **myArray**, depending on whether the memory read occurs before or after the memory write from **myArray[tid + 1] = 2**.

```
// myArray is an array of integers located in global or shared
// memory
__global__ void MyKernel(int* result) {
    int tid = threadIdx.x;
    ...
    int ref1 = myArray[tid];
    __syncthreads();
    myArray[tid + 1] = 2;
    __syncthreads();
    int ref2 = myArray[tid];
    result[tid] = ref1 * ref2;
    ...
}
```

However, in the following slightly modified code sample, threads are guaranteed to belong to the same warp, so that there is no need for any **`__syncthreads()`**.

```
// myArray is an array of integers located in global or shared
// memory
__global__ void MyKernel(int* result) {
    int tid = threadIdx.x;
    ...
    if (tid < warpSize) {
        int ref1 = myArray[tid];
        myArray[tid + 1] = 2;
        int ref2 = myArray[tid];
        result[tid] = ref1 * ref2;
    }
    ...
}
```

Simply removing the **`__syncthreads()`** is not enough however; **`myArray`** must also be declared as volatile as described in Section B.2.4.

Appendix A. CUDA-Enabled GPUs

Table A-1 lists all CUDA-enabled devices with their compute capability, number of multiprocessors, and number of CUDA cores.

These, as well as the clock frequency and the total amount of device memory, can be queried using the runtime or driver API (see reference manual).

Table A-1. CUDA-Enabled Devices with Compute Capability, Number of Multiprocessors, and Number of CUDA Cores

	Compute Capability	**Number of Multiprocessors**	**Number of CUDA Cores**
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	1.1	14	112
GeForce GT 240	1.2	12	96
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX 260M, 9800M GT	1.1	12	96
GeForce 8800 GTS	1.0	12	96
GeForce 9600 GT, 8800M GTS, 9800M GTS	1.1	8	64
GeForce GT 220	1.2	6	48
GeForce 9700M GT	1.1	6	48
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT,	1.1	4	32

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
8600M GS			
GeForce 210	1.2	2	16
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU	1.1	2	16
GeForce 9300M GS, 9200M GS, 9100M G, 8400M G	1.1	1	8
Tesla S1070	1.3	4x30	4x240
Tesla C1060	1.3	30	240
Tesla S870	1.0	4x16	4x128
Tesla D870	1.0	2x16	2x128
Tesla C870	1.0	16	128
Quadro Plex 2200 D2	1.3	2x30	2x240
Quadro Plex 2100 D4	1.1	4x14	4x112
Quadro Plex 2100 Model S4	1.0	4x16	4x128
Quadro Plex 1000 Model IV	1.0	2x16	2x128
Quadro FX 5800	1.3	30	240
Quadro FX 4800	1.3	24	192
Quadro FX 4700 X2	1.1	2x14	2x112
Quadro FX 3700M, FX 3800M	1.1	16	128
Quadro FX 5600	1.0	16	128
Quadro FX 3700	1.1	14	112
Quadro FX 2800M	1.1	12	96
Quadro FX 4600	1.0	12	96
Quadro FX 1800M	1.2	9	72
Quadro FX 3600M	1.1	8	64
Quadro FX 880M	1.2	6	48
Quadro FX 2700M	1.1	6	48
Quadro FX 1700, FX 570, NVS 320M, FX 1700M, FX 1600M, FX 770M, FX 570M	1.1	4	32
Quadro FX 380M	1.2	2	16
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	1.1	2	16
Quadro FX 370M, NVS 130M	1.1	1	8

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.

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.

However, the **`__host__`** qualifier can also be used in combination with the **`__device__`** qualifier, in which case the function is compiled for both the host and the device.

B.1.4 Restrictions

`__device__` and **`__global__`** functions do not support recursion.

`__device__` and **`__global__`** functions cannot declare static variables inside their body.

`__device__` and **`__global__`** functions cannot have a variable number of arguments.

`__device__` functions cannot have their address taken; function pointers to **`__global__`** functions, on the other hand, are supported.

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

`__global__` functions must have void return type.

Any call to a **`__global__`** function must specify its execution configuration as described in Section B.13.

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

`__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 2.0.

B.2 Variable Type Qualifiers

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

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.

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.

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.13). 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__ char 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__ char 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 volatile

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. For example, in the code sample below, the first reference to **`myArray[tid]`** compiles into a global or shared memory read instruction, but the second reference does not as the compiler simply reuses the result of the first read.

```
// myArray is an array of non-zero integers
// located in global or shared memory
__global__ void MyKernel(int* result) {
    int tid = threadIdx.x;
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```

Therefore, **`ref2`** cannot possibly be equal to **`2`** in thread **`tid`** as a result of thread **`tid-1`** overwriting **`myArray[tid]`** by 2.

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 at any time by another thread and therefore any reference to this variable compiles to an actual memory read instruction.

Note that even if **`myArray`** is declared as volatile in the code sample above, there is no guarantee, in general, that **`ref2`** will be equal to 2 in thread **`tid`** since thread **`tid`** might read **`myArray[tid]`** into **`ref2`** before thread **`tid-1`** overwrites its value by 2. Synchronization is required as mentioned in Section 5.4.3.

B.2.5 Restrictions

These qualifiers are not allowed on **`struct`** and **`union`** members, on formal parameters and on local variables within a function that executes on the host.

`__shared__` and **`__constant__`** variables have implied static storage.

`__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.

`__device__` and **`__constant__`** variables are only allowed at file scope.

`__constant__` variables cannot be assigned to from the device, only from the host through host runtime functions (Sections 3.2.1 and 3.3.4).

`__shared__` variables cannot have an initialization as part of their declaration.

An automatic variable declared in device code without any of these qualifiers 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.

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 or the global memory space, otherwise they are restricted to only point to memory allocated or declared in the global memory space.

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 3.3.4 can only be used in host code.

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, longlong2, 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	Same as int1 or longlong1 depending on platform

long2, ulong2	Same as int2 or longlong2 depending on platform
long3, ulong3	Same as int3 depending on platform
long4, ulong4	Same as int4 depending on platform
longlong1	8
longlong2	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.4.6 Restrictions

- It is not allowed to take the address of any of the built-in variables.
- It is not allowed to assign values to any of the built-in variables.

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 3.2.5.3).

`__threadfence_system()` is only supported by devices of compute capability 2.0.

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.10 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 2.0 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

Section C.1 contains a comprehensive list of the C/C++ standard library mathematical functions that are currently supported in device code, along with their respective error bounds. When executed in host code, a given function uses the C runtime implementation if available.

For some of the functions of Section C.1, a less accurate, but faster version exists in the device runtime component; it has the same name prefixed with __ (such as **`__sinf(x)`**). These intrinsic functions are listed in Section C.2, along with their respective error bounds.

The compiler has an option (**`-use_fast_math`**) that forces each function in Table B-2 to compile to its intrinsic counterpart. In addition to reduce 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 B-2. 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)`

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 3.2.4.1. Mutable attributes are described in Section 3.2.4.2. Texture fetching is described in Appendix F.

Appendix G lists the maximum texture width, height, and depth depending on the compute capability of the device.

B.8.1 tex1Dfetch()

```
template<class Type>
Type tex1Dfetch(
   texture<Type, 1, cudaReadModeElementType> texRef,
   int x);

float tex1Dfetch(
   texture<unsigned char, 1, cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<signed char, 1, cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<unsigned short, 1, cudaReadModeNormalizedFloat> texRef,
   int x);

float tex1Dfetch(
   texture<signed short, 1, cudaReadModeNormalizedFloat> texRef,
   int x);
```

fetch the region of linear memory bound to texture reference **`texRef`** using integer texture coordinate **`x`**. No texture filtering and addressing modes are supported. For integer types, these functions 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, 1, 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 Type, enum cudaTextureReadMode readMode>
Type tex1D(texture<Type, 1, readMode> texRef,
           float x);
```

fetches the CUDA array bound to texture reference **texRef** using floating-point texture coordinates **x**.

B.8.3 tex2D()

```
template<class Type, enum cudaTextureReadMode readMode>
Type tex2D(texture<Type, 2, readMode> texRef,
           float x, float y);
```

fetches the CUDA array or the region of linear memory bound to texture reference **texRef** using texture coordinates **x** and **y**.

B.8.4 tex3D()

```
template<class Type, enum cudaTextureReadMode readMode>
Type tex3D(texture<Type, 3, readMode> texRef,
           float x, float y, float z);
```

fetches the CUDA array bound to texture reference **texRef** using texture coordinates **x**, **y**, and **z**.

B.9 Time Function

```
clock_t clock();
```

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.10 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 operations only work with signed and unsigned integers with the exception of **atomicAdd()** for devices of compute capability 2.0 and **atomicExch()** for all devices, which also work for single-precision floating-point numbers.

Atomic functions can only be used in device functions and are only available for devices of compute capability 1.1 and above.

Atomic functions operating on shared memory and atomic functions operating on 64-bit words are only available for devices of compute capability 1.2 and above.

Note that atomic functions operating on mapped page-locked memory (Section 3.2.5.3) are not atomic from the point of view of the host or other devices.

B.10.1 Arithmetic Functions

B.10.1.1 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**.

64-bit words are only supported for global memory.

The floating-point version of **atomicAdd()** is only supported by devices of compute capability 2.0.

B.10.1.2 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.10.1.3 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`**.

64-bit words are only supported for global memory.

B.10.1.4 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.10.1.5 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.10.1.6 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.10.1.7 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.10.1.8 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).

64-bit words are only supported for global memory.

B.10.2 Bitwise Functions

B.10.2.1 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.10.2.2 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.10.2.3 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.11 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 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 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 threads of the warp and returns an integer whose Nth bit is set if and only if **`predicate`** evaluates to non-zero for the Nth thread of the warp. This function is only supported by devices of compute capability 2.0.

B.12 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 the CUDA debugger or the CUDA profiler, all launches are synchronous).

B.13 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 3.3.9.1 for a description of streams).

When using the driver API, the execution configuration is specified through a series of driver function calls as detailed in Section 3.3.3.

When using the runtime API (Section 3.2), 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`** equals the number of blocks being launched; **`Dg.z`** must be equal to 1;
- **`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 G, 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.14 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.

Appendix C. Mathematical Functions

Functions from Section C.1 can be used in both host and device code whereas functions from Section C.2 can only be used in device code.

Note that floating-point functions are overloaded, so that in general, there are three prototypes for a given function **`<func-name>`**:
(1) **`double <func-name>(double)`**, e.g. **`double log(double)`**
(2) **`float <func-name>(float)`**, e.g. **`float log(float)`**
(3) **`float <func-name>f(float)`**, e.g. **`float logf(float)`**
This means, in particular, that passing a **`float`** argument always results in a **`float`** result (variants (2) and (3) above).

C.1 Standard Functions

This section lists all the mathematical standard library functions supported in device code. It also specifies the error bounds of each function when executed on the device. These error bounds also apply when the function is executed on the host in the case where the host does not supply the function. They 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 G.2. This combination can be avoided by using the **`__fadd_rn()`** and **`__fmul_rn()`** 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)
`asinf(x)`	4 (full range)
`acosf(x)`	3 (full range)
`atanf(x)`	2 (full range)
`atan2f(y,x)`	3 (full range)
`sinhf(x)`	3 (full range)
`coshf(x)`	2 (full range)

Function	Maximum ulp error
`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)`	5 (full range)
`erfcinvf(x)`	7 (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)
`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)
`lroundf(x)`	0 (full range)
`llrintf(x)`	0 (full range)
`llroundf(x)`	0 (full range)
`signbit(x)`	N/A
`isinf(x)`	N/A
`isnan(x)`	N/A
`isfinite(x)`	N/A
`copysignf(x,y)`	N/A
`fminf(x,y)`	N/A
`fmaxf(x,y)`	N/A
`fabsf(x)`	N/A
`nanf(cptr)`	N/A

Function	Maximum ulp error
`nextafterf(x,y)`	N/A

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)
`tan(x)`	2 (full range)
`sincos(x,sptr,cptr)`	2 (full range)
`sinpi(x)`	2 (full range)

Function	Maximum ulp error
`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)`	5 (full range)
`erfinv(x)`	8 (full range)
`erfcinv(x)`	8 (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)
`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)
`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)
`signbit(x)`	N/A
`isinf(x)`	N/A
`isnan(x)`	N/A

Function	Maximum ulp error
`isfinite(x)`	N/A
`copysign(x,y)`	N/A
`fmin(x,y)`	N/A
`fmax(x,y)`	N/A
`fabs(x)`	N/A
`nan(cptr)`	N/A
`nextafter(x,y)`	N/A

C.1.3 Integer Functions

Integer **`min(x,y)`** and **`max(x,y)`** are supported and map to a single instruction on the device.

C.2 Intrinsic Functions

This section lists the intrinsic functions that are only supported 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)`**).

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.

Unlike type conversion functions (such as **`__int2float_rn`**) that convert from one type to another, type casting functions simply perform a type cast on the argument, leaving the value unchanged. For example, **`__int_as_float(0xC0000000)`** is equal to **`-2`**, **`__float_as_int(1.0f)`** is equal to **`0x3f800000`**.

C.2.1 Single-Precision Floating-Point Functions

`__fadd_rn()` and **`__fmul_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.

Both the regular floating-point division and **`__fdividef(x,y)`** have the same accuracy, but for $2^{126} <$ **`y`** $< 2^{128}$, **`__fdividef(x,y)`** delivers a result of zero, whereas the regular division delivers the correct result to within the accuracy stated in Table C-3. Also, for $2^{126} <$ **`y`** $< 2^{128}$, if **`x`** is infinity, **`__fdividef(x,y)`** delivers a **`NaN`** (as a result of multiplying infinity by zero), while the regular division returns infinity.

`__saturate(x)` returns 0 if **`x`** is less than 0, 1 if **`x`** is more than 1, and **`x`** otherwise.

`__float2ll_[rn,rz,ru,rd](x)` (respectively **`__float2ull_[rn,rz,ru,rd](x)`**) converts single-precision floating-point parameter **`x`** to 64-bit signed (respectively unsigned) integer with specified IEEE-754 rounding modes.

Table C-3. 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 [$-\pi$, π], the maximum absolute error is $2^{-21.41}$, and larger otherwise.
`__cosf(x)`	For **`x`** in [$-\pi$, π], the maximum absolute error is $2^{-21.19}$, and larger otherwise.
`__sincosf(x,sptr,cptr)`	Same as **`sinf(x)`** and **`cosf(x)`**.
`__tanf(x)`	Derived from its implementation as **`__sinf(x) * (1 / __cosf(x))`**.
`__powf(x, y)`	Derived from its implementation as **`exp2f(y * __log2f(x))`**.
`__int_as_float(x)`	N/A
`__float_as_int(x)`	N/A
`__saturate(x)`	N/A
`__float2int_[rn,rz,ru,rd](x)`	N/A
`__float2uint_[rn,rz,ru,rd](x)`	N/A

`__int2float_[rn,rz,ru,rd](x)`	N/A
`__uint2float_[rn,rz,ru,rd](x)`	N/A
`__float2ll_[rn,rz,ru,rd](x)`	N/A
`__float2ull_[rn,rz,ru,rd](x)`	N/A
`__ll2float_[rn,rz,ru,rd](x)`	N/A
`__ull2float_[rn,rz,ru,rd](x)`	N/A

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-4. 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.
`__double2float_[rn,rz](x)`	N/A
`__double2int_[rn,rz,ru,rd](x)`	N/A
`__double2uint_[rn,rz,ru,rd](x)`	N/A
`__double2ll_[rn,rz,ru,rd](x)`	N/A
`__double2ull_[rn,rz,ru,rd](x)`	N/A
`__int2double_rn(x)`	N/A
`__uint2double_rn(x)`	N/A
`__ll2double_[rn,rz,ru,rd](x)`	N/A
`__ull2double_[rn,rz,ru,rd](x)`	N/A
`__double_as_longlong(x)`	N/A
`__longlong_as_double(x)`	N/A
`__double2hiint(x)`	N/A
`__double2loint(x)`	N/A
`__hiloint2double(x, ys)`	N/A
`__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
`__double2float_[ru,rd](x)`	N/A

C.2.3 Integer Functions

`__[u]mul24(x,y)` computes the product of the 24 least significant bits of the integer parameters **`x`** and **`y`** and delivers the 32 least significant bits of the result. The 8 most significant bits of **`x`** or **`y`** are ignored.

`__[u]mulhi(x,y)` computes the product of the integer parameters **`x`** and **`y`** and delivers the 32 most significant bits of the 64-bit result.

`__[u]mul64hi(x,y)` computes the product of the 64-bit integer parameters **`x`** and **`y`** and delivers the 64 most significant bits of the 128-bit result.

`__[u]sad(x,y,z)` (Sum of Absolute Difference) returns the sum of integer parameter **`z`** and the absolute value of the difference between integer parameters **`x`** and **`y`**.

`__clz(x)` returns the number, between 0 and 32 inclusive, of consecutive zero bits starting at the most significant bit (i.e. bit 31) of integer parameter **`x`**.

`__clzll(x)` returns the number, between 0 and 64 inclusive, of consecutive zero bits starting at the most significant bit (i.e. bit 63) of 64-bit integer parameter **`x`**.

`__ffs(x)` returns the position of the first (least significant) bit set in integer parameter **`x`**. The least significant bit is position 1. If **`x`** is 0, **`__ffs()`** returns 0. Note that this is identical to the Linux function **`ffs`**.

`__ffsll(x)` returns the position of the first (least significant) bit set in 64-bit integer parameter **`x`**. The least significant bit is position 1. If **`x`** is 0, **`__ffsll()`** returns 0. Note that this is identical to the Linux function **`ffsll`**.

`__popc(x)` returns the number of bits that are set to 1 in the binary representation of 32-bit integer parameter **`x`**.

`__popcll(x)` returns the number of bits that are set to 1 in the binary representation of 64-bit integer parameter **`x`**.

`__brev(x)` reverses the bits of 32-bit unsigned integer parameter **`x`**, i.e. bit N of the result corresponds to bit 31-N of **`x`**.

`__brevll(x)` reverses the bits of 64-bit unsigned long long parameter **`x`**, i.e. bit N of the result corresponds to bit 63-N of **`x`**.

Appendix D. C++ Language Constructs

CUDA supports the following C++ language constructs for device code:

- Polymorphism
- Default Parameters
- Operator Overloading
- Namespaces
- Function Templates
- Classes for devices of compute capability 2.0

These C++ constructs are implemented as specified in "The C++ Programming Langue" reference. It is valid to use any of these constructs in .cu CUDA files for host, device, and kernel (__global__) functions. Any restrictions detailed in previous parts of this programming guide, like the lack of support for recursion, still apply.

The following subsections provide examples of the various constructs.

D.1 Polymorphism

Generally, polymorphism is the ability to define that functions or operators behave differently in different contexts. This is also referred to as function (and operator, see below) overloading.

In practical terms, this means that it is permissible to define two different functions within the same scope (namespace) as long as they have a distinguishable function signature. That means that the two functions either consume a different number of parameters or parameters of different types. When either of the multiple functions gets invoked the compiler resolves to the function's implementation that matches the function signature.

Because of implicit typecasting, a compiler may encounter multiple potential matches for a function invocation and in that case the matching rules as described in the C++ Language Standard apply. In practice this means that the compiler will pick the closest match in case of multiple potential matches.

Example: The following is valid CUDA code:

```
__device__ void f(float x)
{
    // do something with x
}

__device__ void f(int i)
{
    // do something with i
}

__device__ void f(double x, double y)
{
    // do something with x and y
}
```

D.2 Default Parameters

With support for polymorphism as described in the previous subsection and the function signature matching rules in place it becomes possible to provide support for default values for function parameters.

Example:

```
__device__ void f(float x = 0.0f)
{
    // do something with x
}
```

Kernel or other device functions can now invoke this version of f in one of two ways:

```
f();
// or
float x = /* some value */;
f(x);
```

Default parameters can only be given for the last n parameters of a function.

D.3 Operator Overloading

Operator overloading allows programmers to define operators for new data-types. Examples of overloadable operators in C++ are: +, -, *, /, +=, &, [], etc.

Example: The following is valid CUDA code, implementing the + operation between two **`uchar4`** vectors:

```
__device__ uchar4 operator+ (const uchar4 & a, const uchar4 & b)
{
    uchar4 r;
    r.x = a.x + b.x;
    ...
    return r;
}
```

This new operator can now be used like this:

```
uchar4 a, b, c;
a = b = /* some initial value */;
```

```
c = a + b;
```

D.4 Namespaces

Namespaces in C++ allow for the creation of a hierarchy of scopes of visibility. All the symbols inside a namespace can be used within this namespaces without additional syntax.

The use of namespaces can be used to solve the problem of name-clashes (two different symbols using identical names), which commonly occurs when using multiple function libraries from different sources.

Example: The following code defines two functions "f()" in two separate namespaces ("nvidia" and "other"):

```
namespace nvidia {
  __device__ void f(float x)
{ /* do something with x */ ;}
}

namespace other {
  __device__ void f(float x)
{ /* do something with x */ ;}
}
```

The functions can now be used anywhere via fully qualified names:

```
nvidia::f(0.5f);
```

All the symbols in a namespace can be imported into another namespace (scope) like this:

```
using namespace nvidia;
f(0.5f);
```

D.5 Function Templates

Function templates are a form of meta-programming that allows writing a generic function in a data-type independent fashion. CUDA supports function templates to the full extent of the C++ standard, including the following concepts:

- Implicit template parameter deduction.
- Explicit instantiation.
- Template specialization.

Example:

```
template <T>
__device__ bool f(T x)
{ return /* some clever code that turns x into a bool here */ }
```

This function will convert x of any data-type to a bool as long as the code in the function's body can be compiled for the actually type (T) of the variable x.

`f()` can be invoked in two ways:

```
int x = 1;
```

```
bool result = f(x);
```

This first type of invocation relies on the compiler's ability to **implicitly deduce** the correct function type for T. In this case the compiler would deduce `T` to be `int` and instantiate `f<int>(x)`.

The second type of invoking the template function is via **explicit instantiation** like this:

```
bool result = f<double>(0.5);
```

Function templates may be **specialized**:

```
template <T>
__device__ bool f(T x)
{ return false; }

template <>
__device__ bool
f<int>(T x)
{ return true; }
```

In this case the implementation for T representing the int type are specialized to return true, all other types will be caught by the more general template and return false.

The complete set of matching rules (for implicitly deducing template parameters) and matching polymorphous functions apply as specified in the C++ standard.

D.6 Classes

Code compiled for devices with compute capability 2.0 and greater may make use of C++ classes, as long as none of the member functions are virtual (this restriction will be removed in some future release).
There are two common use cases for classes without virtual member functions:

- Small-data aggregations. E.g. data types like pixels (r, g, b, a), 2D and 3D points, vectors, etc.
- Functor classes. The use of functors is necessitated by the fact that device-function pointers are not supported and thus it is not possible to pass functions as template parameters. A workaround for this restriction is the use of functor classes (see code sample below).

D.6.1 Example 1 Pixel Data Type

The following is an example of a data type for RGBA pixels with 8 bit per channel depth:

```
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)
    { ; }

    // other methods and operators left out for sake of brevity

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_);
}
```

Other device code can now make use of this new data type as one would expect:

```
PixelRGBA p1, p2;

// [...] initialization of p1 and p2 here

PixelRGBA p3 = p1 + p2;
```

D.6.2 Example 2 Functor Class

The following example shows how functors may be used as function template parameters to implement a set of vector arithmetic operations.

Here are two functors for float addition and subtraction:

```
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;
    }
};
```

The following templatized kernel makes use of the functors like the ones above in order to implement operations on vectors of floats:

```
// 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]);
    }
}
```

The `VectorOperation` kernel may now be launched like this in order to get a vector addition:

```
// Host code
VectorOperation<<<blocks, threads>>>(v1, v2, v3, N, Add());
```

Appendix E. NVCC Specifics

E.1 __noinline__

By default, a **__device__** function is always inlined. The **__noinline__** function qualifier however 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.

E.2 #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.

E.3 __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.

Appendix F. 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 3.2.4).

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, or $N \times M \times L$ texels for a three-dimensional texture. It is fetched using texture coordinates x, y, and z.

A texture coordinate must fall within T's valid addressing range before it can be used to address T. The addressing mode specifies how an out-of-range texture coordinate x is remapped to the valid range. If x is non-normalized, only the clamp addressing mode is supported and x is replaced by 0 if $x < 0$ and $N - 1$ if $N \leq x$. If x is normalized:

- In clamp addressing mode, x is replaced by 0 if $x < 0$ and $1 - \frac{1}{N}$ if $1 \leq x$,
- In wrap addressing mode, x is replaced by $frac(x)$, where $frac(x) = x - floor(x)$ and $floor(x)$ is the largest integer not greater than x.

In the remaining of the appendix, x, y, and z are the non-normalized texture coordinates remapped to T's valid addressing range. x, y, and z are derived from the normalized texture coordinates $\hat{x}$, $\hat{y}$, and $\hat{z}$ as such: $x = N\hat{x}$, $y = M\hat{y}$, and $z = L\hat{z}$.

F.1 Nearest-Point Sampling

In this filtering mode, the value returned by the texture fetch is

- $tex(x) = T[i]$ for a one-dimensional texture,
- $tex(x, y) = T[i, j]$ for a two-dimensional texture,
- $tex(x, y, z) = T[i, j, k]$ for a three-dimensional texture,

where $i = floor(x)$, $j = floor(y)$, and $k = floor(z)$.

Figure D-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 3.2.4.1).

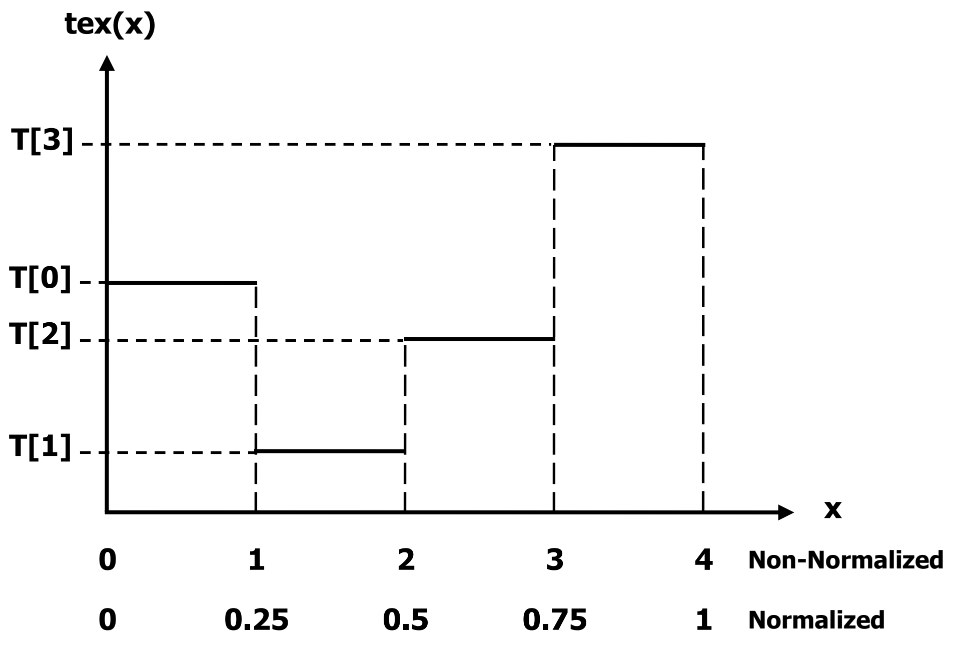


Figure F-1. Nearest-Point Sampling of a One-Dimensional Texture of Four Texels

F.2 Linear Filtering

In this filtering mode, which is only available for floating-point textures, the value returned by the texture fetch is

- $tex(x) = (1-\alpha)T[i] + \alpha T[i+1]$ for a one-dimensional texture,

- $tex(x,y) = (1-\alpha)(1-\beta)T[i,j] + \alpha(1-\beta)T[i+1,j] + (1-\alpha)\beta T[i,j+1] + \alpha\beta T[i+1,j+1]$ for a two-dimensional texture,
- $tex(x,y,z) =$
- $$\begin{aligned}&(1-\alpha)(1-\beta)(1-\gamma)T[i,j,k] + \alpha(1-\beta)(1-\gamma)T[i+1,j,k] +\\ &(1-\alpha)\beta(1-\gamma)T[i,j+1,k] + \alpha\beta(1-\gamma)T[i+1,j+1,k] +\\ &(1-\alpha)(1-\beta)\gamma T[i,j,k+1] + \alpha(1-\beta)\gamma T[i+1,j,k+1] +\\ &(1-\alpha)\beta\gamma T[i,j+1,k+1] + \alpha\beta\gamma T[i+1,j+1,k+1]\end{aligned}$$
  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 F-2 illustrates nearest-point sampling for a one-dimensional texture with $N = 4$.

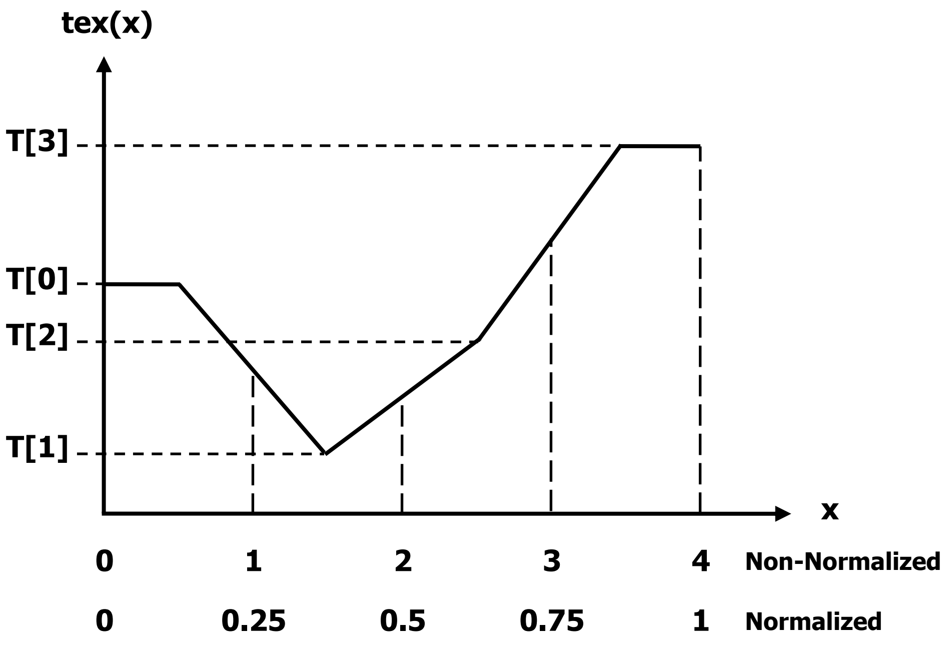


Figure F-2. Linear Filtering of a One-Dimensional Texture of Four Texels in Clamp Addressing Mode

F.3 Table Lookup

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

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

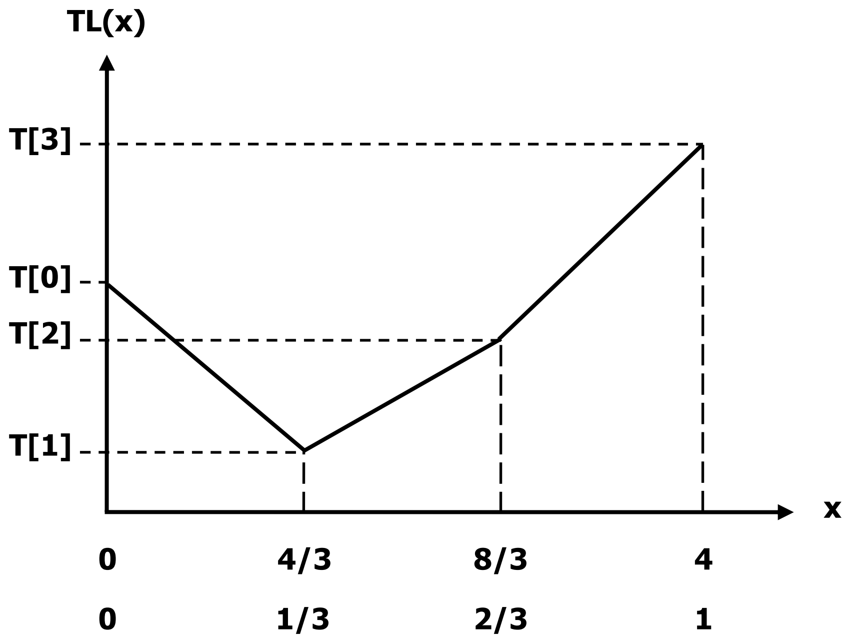


Figure F-3. One-Dimensional Table Lookup Using Linear Filtering

Appendix G. Compute Capabilities

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

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

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

Section G.3 and 0 give more details on the architecture of devices of compute capability 1.x and 2.0, respectively.

G.1 Features and Technical Specifications

	Compute Capability				
Feature Support **(Unlisted features are supported for all compute capabilities)**	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Integer atomic functions operating on 32-bit words in global memory (Section B.10)	No	yes	yes	yes	yes
Integer atomic functions operating on 64-bit words in global memory (Section B.10)	No	No	Yes	Yes	Yes
Integer atomic functions operating on 32-bit words in shared memory (Section B.10)	No	No	Yes	Yes	Yes
Warp vote functions (Section B.11)	No	No	Yes	Yes	Yes
Double-precision floating-point numbers	No	No	No	Yes	Yes
Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.10)	No	No	No	No	Yes
__ballot() (Section B.11)	No	No	No	No	Yes
__threadfence_system() (Section B.5)	No	No	No	No	Yes
__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)	No	No	No	No	Yes

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 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				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of instructions per kernel	2 million				

G.2 Floating-Point Standard

All compute devices follow the IEEE 754-2008 standard for binary floating-point arithmetic with the following deviations:

- There is no dynamically configurable rounding mode; however, most of the operations support multiple IEEE rounding modes, exposed via device intrinsics;
- There is no mechanism for detecting that a floating-point exception has occurred and all operations behave as if the IEEE-754 exceptions are always masked, and deliver the masked response as defined by IEEE-754 if there is an exceptional event; for the same reason, while SNaN encodings are supported, they are not signaling and are handled as quiet;
- The result of a single-precision floating-point operation involving one or more input NaNs is the quiet NaN of bit pattern 0x7fffffff;
- Double-precision floating-point absolute value and negation are not compliant with IEEE-754 with respect to NaNs; these are passed through unchanged;
- For **single-precision** floating-point numbers on devices of **compute capability 1.x**:
  - Denormalized numbers are not supported; floating-point arithmetic and comparison instructions convert denormalized operands to zero prior to the floating-point operation;
  - Underflowed results are flushed to zero;
  - Some instructions are not IEEE-compliant:
    - Addition and multiplication are often combined into a single multiply-add instruction (FMAD), which truncates (i.e. without rounding) the intermediate mantissa of the multiplication;
    - Division is implemented via the reciprocal in a non-standard-compliant way;
    - Square root is implemented via the reciprocal square root in a non-standard-compliant way;
    - For addition and multiplication, only round-to-nearest-even and round-towards-zero are supported via static rounding modes; directed rounding towards +/- infinity is not supported;

  To mitigate the impact of these restrictions, IEEE-compliant software (and therefore slower) implementations are provided through the following intrinsics (c.f. 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 2.0 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 2.0.

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.

G.3 Compute Capability 1.x

G.3.1 Architecture

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

- 8 CUDA cores for integer and single-precision floating-point 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 3.2.4.

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

G.3.2 Global Memory

A global memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. Sections G.3.2.1 and G.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 G-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.

G.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.

G.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 lowest numbered active thread. The segment size depends on the size of the words accessed by the threads:
  - 32 bytes for 1-byte words,
  - 64 bytes for 2-byte words,
  - 128 bytes for 4-, 8- and 16-byte words.
- Find all other active threads whose requested address lies in the same segment.
- Reduce the transaction size, if possible:
  - If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
  - If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.
- Carry out the transaction and mark the serviced threads as inactive.
- Repeat until all threads in the half-warp are serviced.

G.3.3 Shared Memory

Shared memory has 16 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e. interleaved. 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.

G.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:

```
__shared__ float shared[32];
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 G-2 shows some examples of strided access for devices of compute capability 2.0. The same examples apply for devices of compute capability 1.x, but with 16 banks instead of 32.

G.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 G-3 shows some examples of memory read accesses that involve the broadcast mechanism. The same examples apply for devices of compute capability 1.x, but with 16 banks instead of 32.

G.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:

```
__shared__ char shared[32];
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];
```

G.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:

```
__shared__ double shared[32];
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 2.0.

The same applies to structure assignments. The following code, for example:

```
__shared__ struct type shared[32];
struct type data = shared[BaseIndex + tid];
```

results in:

- Three separate reads without bank conflicts if **type** is defined as

  ```
  struct type {
      float x, y, z;
  };
  ```

  since each member is accessed with an odd stride of three 32-bit words;
- Two separate reads with bank conflicts if **type** is defined as

  ```
  struct type {
      float x, y;
  };
  ```

  since each member is accessed with an even stride of two 32-bit words.

G.4 Compute Capability 2.0

G.4.1 Architecture

For devices of compute capability 2.0, a multiprocessor consists of:

- 32 CUDA cores for integer and floating-point arithmetic operations,
- 4 special function units for single-precision floating-point transcendental functions,

- 2 warp schedulers.

At every instruction issue time, the first scheduler issues an instruction for some warp with an odd ID and the second scheduler issues an instruction for some warp with an even ID. The only exception is when a scheduler issues a double-precision floating-point instruction: In this case, 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:

- 2 clock cycles for an integer or floating-point arithmetic instruction,
- 2 clock cycles for a double-precision floating-point arithmetic instruction,
- 8 clock cycles for a single-precision floating-point transcendental instruction.

A multiprocessor also has a read-only uniform 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 with 16 KB of L1 cache (default setting) or as 16 KB of shared memory with 48 KB of L1 cache using **`cudaFuncSetCacheConfig()`** or **`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)

// Driver API
// CU_FUNC_CACHE_PREFER_SHARED: shared memory is 48 KB
// CU_FUNC_CACHE_PREFER_L1: shared memory is 16 KB
// CU_FUNC_CACHE_PREFER_NONE: no preference
CUfunction myKernel;
cuFuncSetCacheConfig(myKernel, CU_FUNC_CACHE_PREFER_SHARED)
```

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 3.2.4.

G.4.2 Global Memory

Global memory accesses are cached. Using the **`-dlcm`** compilation flag, they can be configured at compile time to be cached in both L1 and L2 (**`-Xptxas -dlcm=ca`**) or in L2 only (**`-Xptxas -dlcm=cg`**).

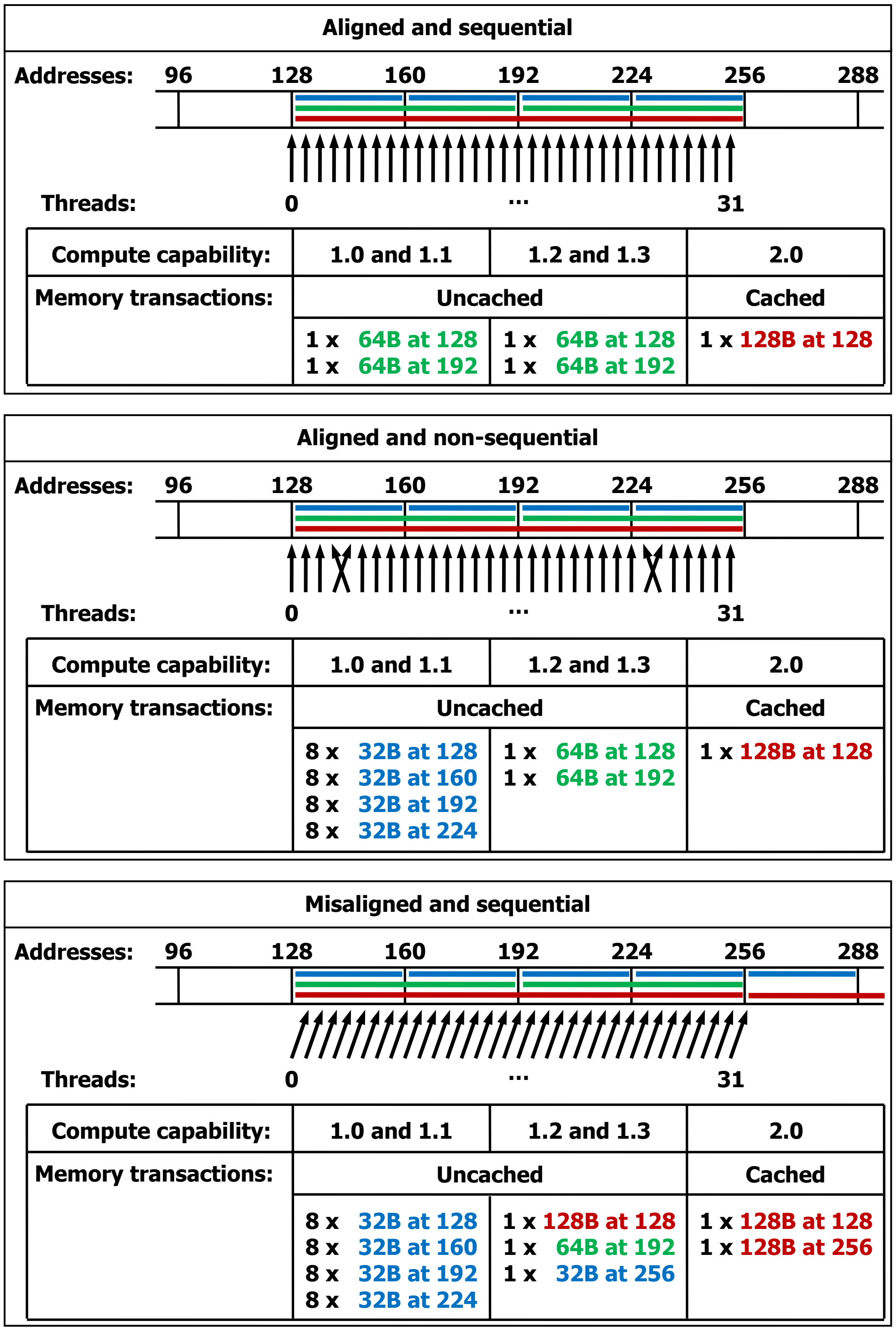
A cache line in L1 or L2 is 128 bytes and maps to a 128-byte aligned segment in device memory.

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.

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.


Aligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 … 31
Aligned and non-sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 … 31
Misaligned and sequential
Addresses: 96 128 160 192 224 256 288
Threads: 0 … 31

Aligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	1 x 64B at 128 1 x 64B at 192	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Aligned and non-sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 64B at 128 1 x 64B at 192	1 x 128B at 128

Misaligned and sequential

Compute capability:	1.0 and 1.1	1.2 and 1.3	2.0
Memory transactions:	Uncached		Cached
	8 x 32B at 128 8 x 32B at 160 8 x 32B at 192 8 x 32B at 224	1 x 128B at 128 1 x 64B at 192 1 x 32B at 256	1 x 128B at 128 1 x 128B at 256

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

G.4.3 Shared Memory

Shared memory has 32 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e. interleaved. Each bank has a bandwidth of 32 bits per two clock cycles. Therefore, unlike for devices of lower compute capability, 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.

A bank conflict only occurs if two or more threads access any bytes within *different* 32-bit words belonging to the same bank. If two or more threads access any bytes within the same 32-bit word, there is no bank conflict between these threads: For read accesses, the word is broadcast to the requesting threads (unlike for devices of compute capability 1.x, multiple words can be broadcast in a single transaction); for write accesses, each byte 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:

```
__shared__ char shared[32];
char data = shared[BaseIndex + tid];
```

G.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:

```
__shared__ float shared[32];
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 G-2 shows some examples of strided access.

G.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;
};

__shared__ struct type shared[32];
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 or more 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:

```
__shared__ double shared[32];
double data = shared[BaseIndex + tid];
```

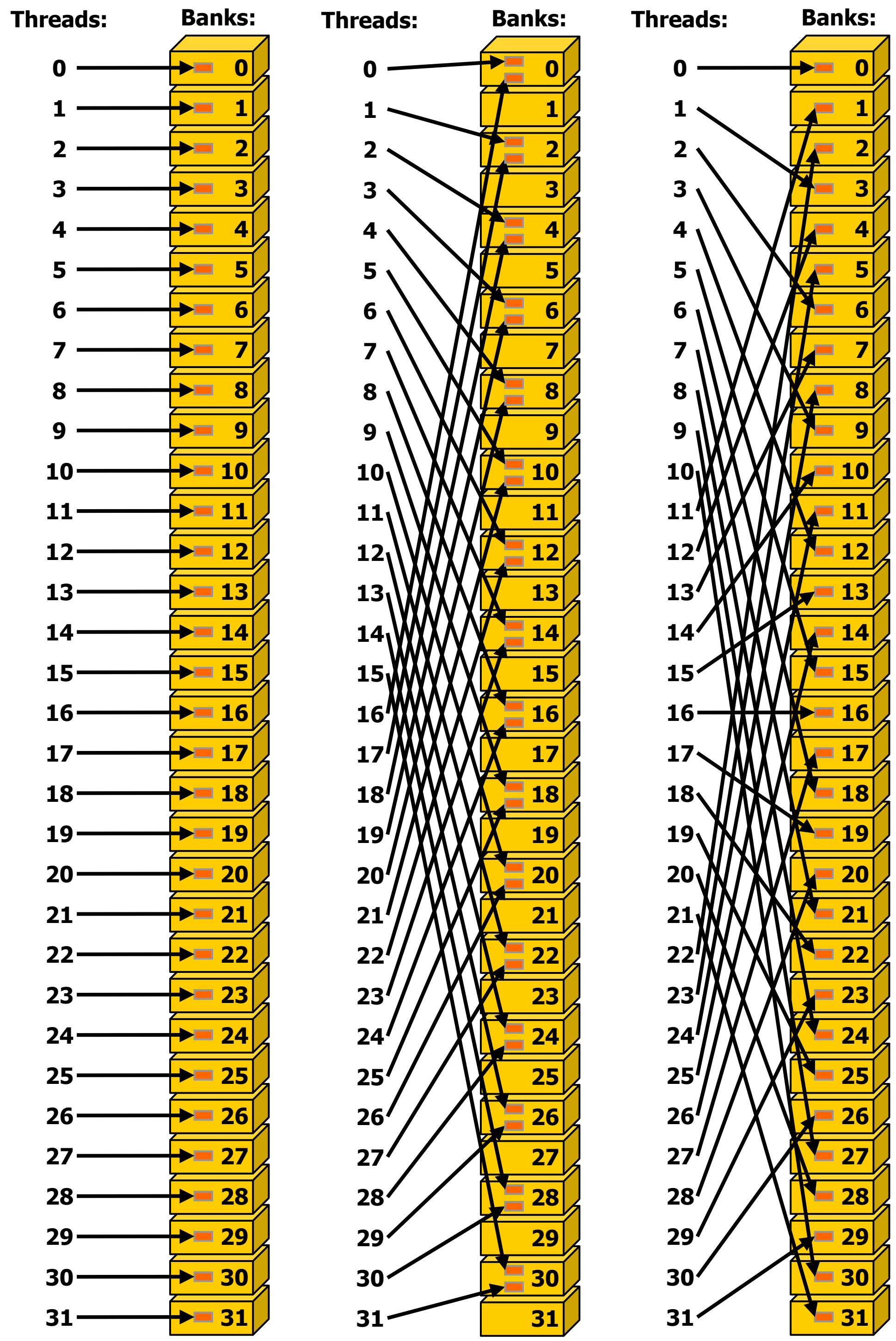
128-Bit Accesses

The majority of 128-bit accesses will cause 2-way bank conflicts, even if no two threads in a quarter-warp access different addresses belonging to the same bank. Therefore, to determine the ways of bank conflicts, one must add 1 to the maximum number of threads in a quarter-warp that access different addresses belonging to the same bank.

G.4.4 Constant Memory

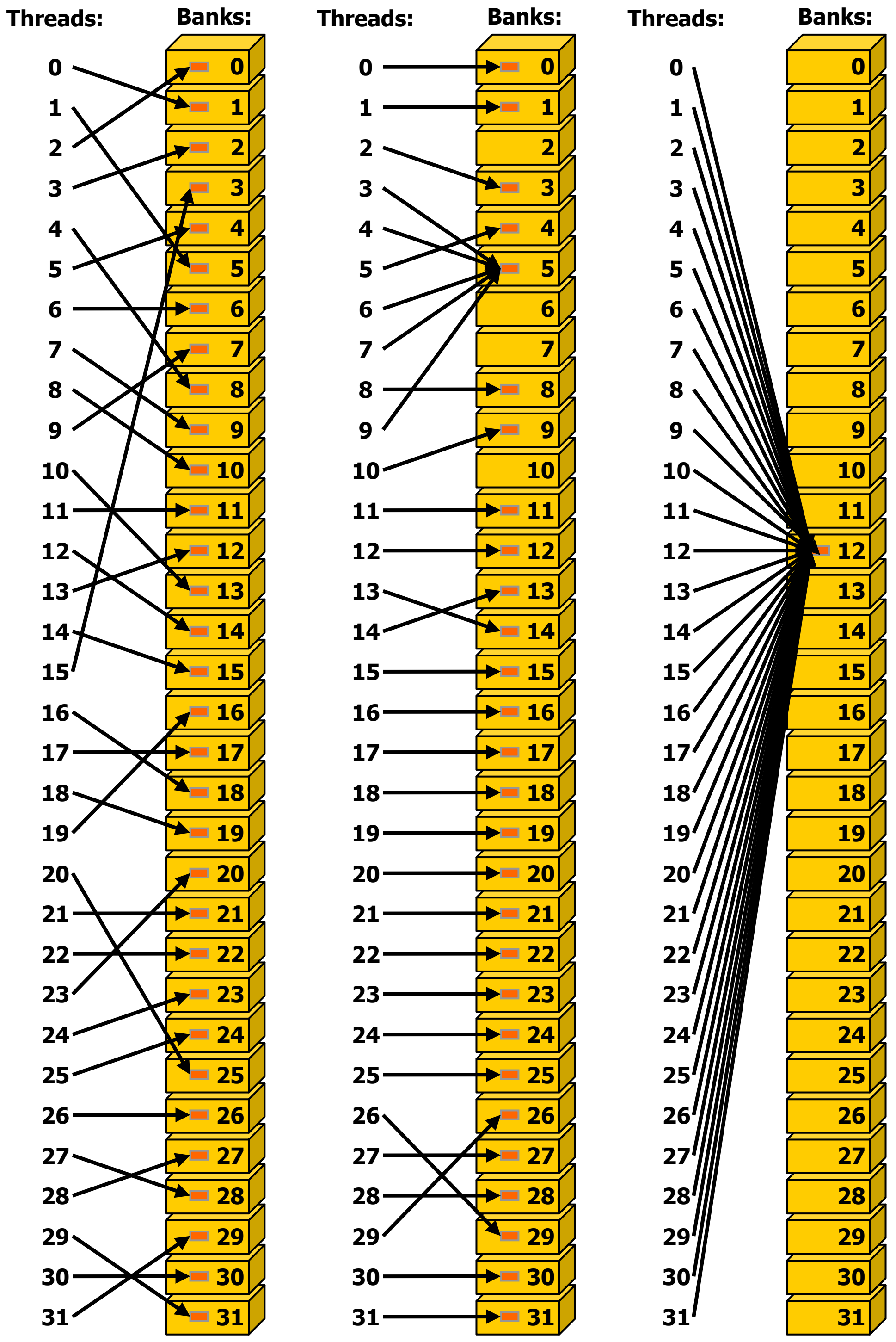
In addition to the constant memory space supported by devices of all compute capabilities (where **`__constant__`** variables reside), devices of compute capability 2.0 support the LDU (LoaD Uniform) instruction that the compiler use to load any variable that is:

- pointing to global memory,
- read-only in the kernel (programmer can enforce this using the **`const`** keyword),
- not dependent on thread ID.



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 (2-way bank conflicts).
Right: Linear addressing with a stride of three 32-bit words (no bank conflict).

Figure G-2 Examples of Strided Shared Memory Accesses for Devices of Compute Capability 2.0



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 (all threads access the same word).

Figure G-3 Examples of Irregular and Colliding Shared Memory Accesses for Devices of Compute Capability 2.0

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