NVIDIA CUDA Compute Unified Device Architecture

Programming Guide

Version 1.0

6/23/2007

Table of Contents

List of Figures

Chapter 1. Introduction to CUDA

1.1 The Graphics Processor Unit as a Data-Parallel Computing Device

In a matter of just a few years, the programmable graphics processor unit has evolved into an absolute computing workhorse, as illustrated by Figure 1-1. With multiple cores driven by very high memory bandwidth, today's GPUs offer incredible resources for both graphics and non-graphics processing.

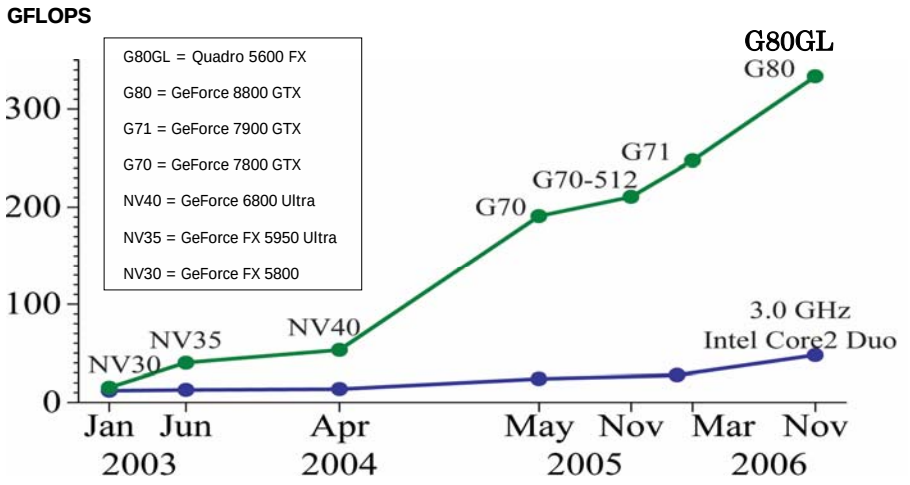


Figure 1-1. Floating-Point Operations per Second for the CPU and GPU

The main reason behind such an evolution is that the GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore is designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by Figure 1-2.

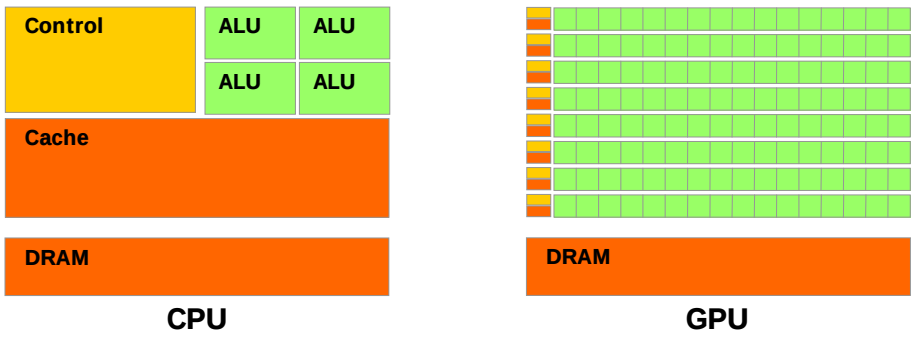


Figure 1-2. The GPU Devotes More Transistors to Data Processing

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 such as arrays 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.

Up until now, however, accessing all that computational power packed into the GPU and efficiently leveraging it for non-graphics applications remained tricky:

- The GPU could only be programmed through a graphics API, imposing a high learning curve to the novice and the overhead of an inadequate API to the non-graphics application.
- The GPU DRAM could be read in a general way – GPU programs can *gather* data elements from any part of DRAM – but could not be written in a general way – GPU programs cannot *scatter* information to any part of DRAM –, removing a lot of the programming flexibility readily available on the CPU.
- Some applications were bottlenecked by the DRAM memory bandwidth, under-utilizing the GPU's computational power.

This document describes a novel hardware and programming model that is a direct answer to these problems and exposes the GPU as a truly generic data-parallel computing device.

1.2 CUDA: A New Architecture for Computing on the GPU

CUDA stands for **Compute Unified Device Architecture** and is a new hardware and software architecture for issuing and managing computations on the GPU as a data-parallel computing device without the need of mapping them to a graphics API. It is available for the GeForce 8 Series, Quadro FX 5600/4600, and Tesla solutions. The operating system's multitasking mechanism is responsible for managing the access to the GPU by several CUDA and graphics applications running concurrently.

The CUDA software stack is composed of several layers as illustrated in Figure 1-3: a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries of common usage, CUFFT and CUBLAS that are both described in separate documents. The hardware has been designed to support lightweight driver and runtime layers, resulting in high performance.

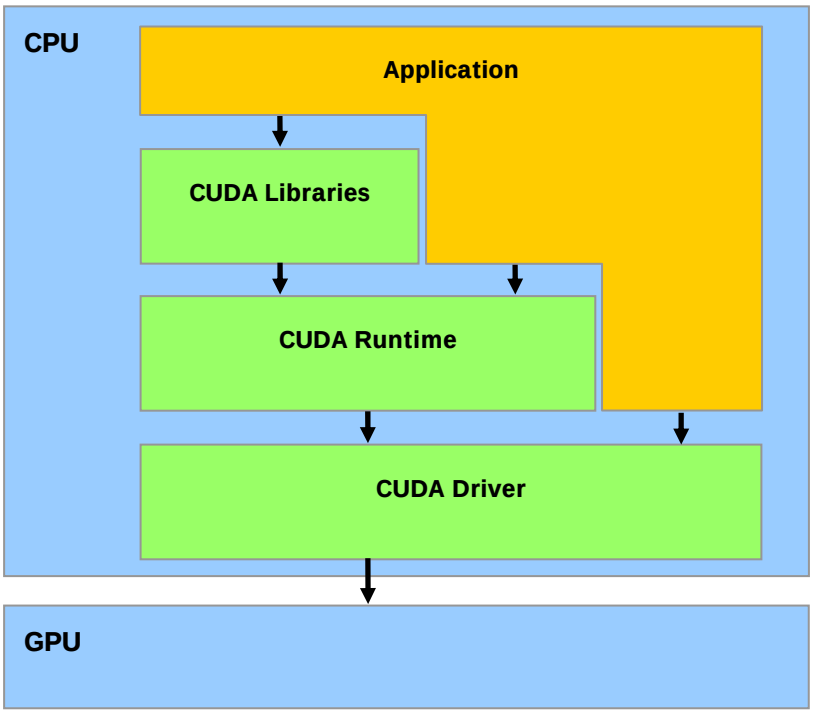


Figure 1-3. Compute Unified Device Architecture Software Stack

The CUDA API comprises an extension to the C programming language for a minimum learning curve (see Chapter 4).

CUDA provides general DRAM memory addressing as illustrated in Figure 1-4 for more programming flexibility: both scatter and gather memory operations. From a programming perspective, this translates into the ability to read and write data at any location in DRAM, just like on a CPU.

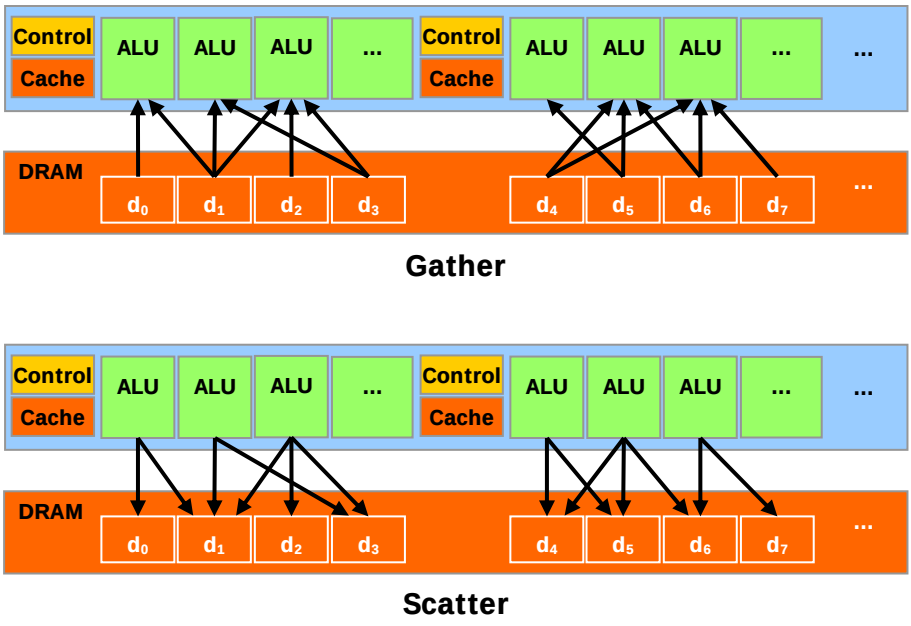


Figure 1-4. The *Gather* and *Scatter* Memory Operations

CUDA features a parallel data cache or on-chip shared memory with very fast general read and write access, that threads use to share data with each other (see Chapter 3). As illustrated in Figure 1-5, applications can take advantage of it by minimizing overfetch and round-trips to DRAM and therefore becoming less dependent on DRAM memory bandwidth.

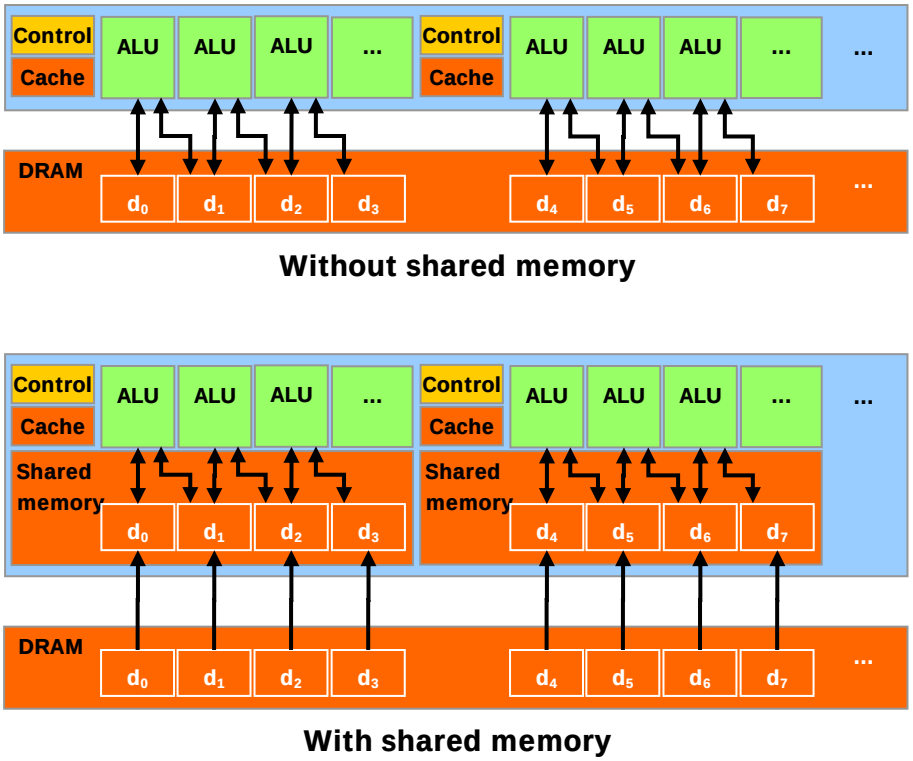


Figure 1-5. Shared Memory Brings Data Closer to the ALUs

1.3 Document's Structure

This document is organized into the following chapters:

- Chapter 1 contains a general introduction to CUDA.
- Chapter 2 outlines the programming model.
- Chapter 3 describes its hardware implementation.
- Chapter 4 describes the CUDA API and runtime.
- Chapter 5 gives some guidance on how to achieve maximum performance.
- Chapter 6 illustrates the previous chapters by walking through the code of some simple example.
- Appendix A gives the technical specifications of various devices.
- Appendix B lists the mathematical functions supported in CUDA.
- Appendix C lists the atomic functions supported in CUDA.
- Appendix D is the CUDA runtime API reference.
- Appendix E is the CUDA driver API reference.

Chapter 2. Programming Model

2.1 A Highly Multithreaded Coprocessor

When programmed through CUDA, the GPU is viewed as a *compute device* capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU, or *host*: In other words, data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the device.

More precisely, a portion of an application that is executed many times, but independently on different data, can be isolated into a function that is executed on the device as many different threads. To that effect, such a function is compiled to the instruction set of the device and the resulting program, called a *kernel*, is downloaded to the device.

Both the host and the device maintain their own DRAM, referred to as *host memory* and *device memory*, respectively. One can copy data from one DRAM to the other through optimized API calls that utilize the device's high-performance Direct Memory Access (DMA) engines.

2.2 Thread Batching

The batch of threads that executes a kernel is organized as a grid of thread blocks as described in Sections 2.2.1 and 2.2.2 and illustrated in Figure 2-1.

2.2.1 Thread Block

A thread block is a batch of threads that can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel, where threads in a block are suspended until they all reach the synchronization point.

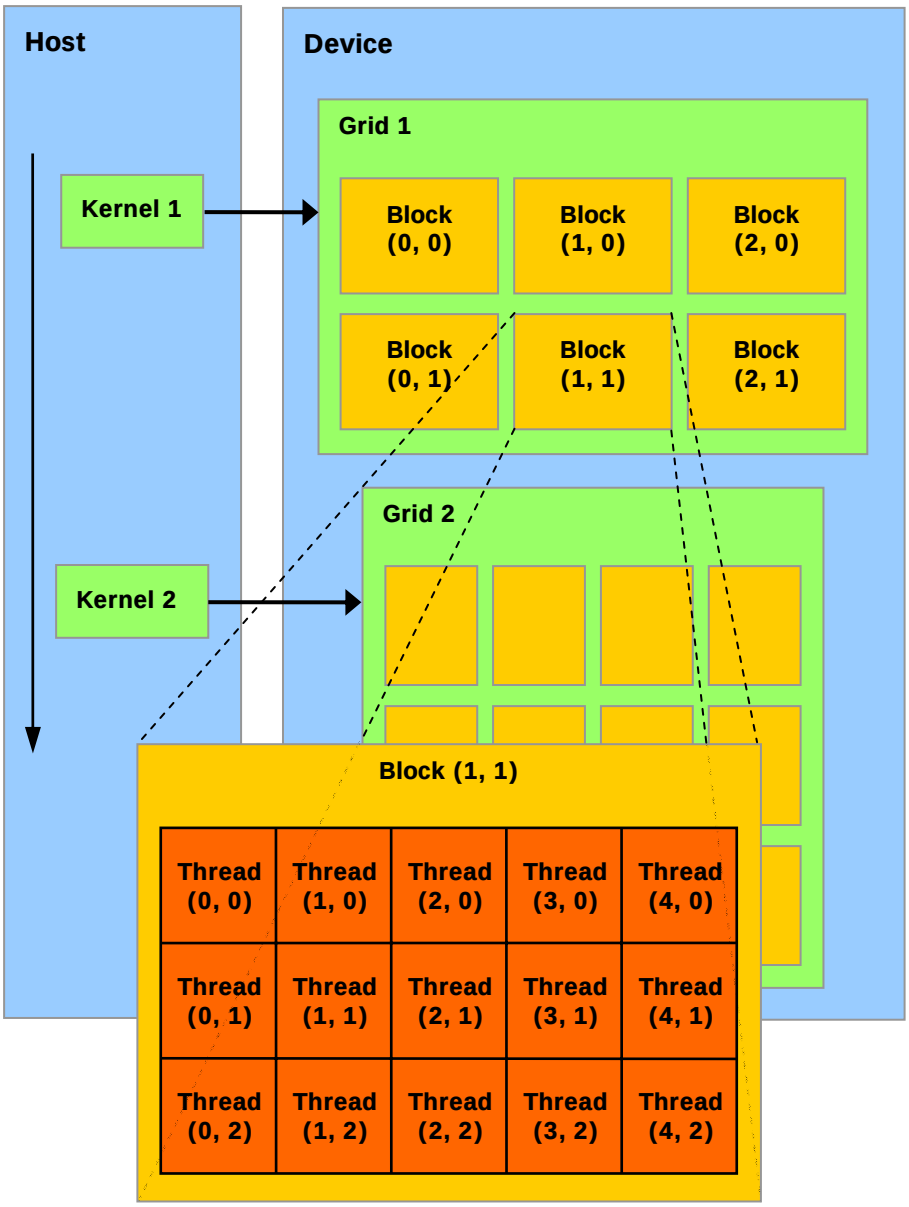
Each thread is identified by its *thread ID*, which is the thread number within the block. To help with complex addressing based on the thread ID, an application can also specify a block as a two- or three-dimensional array of arbitrary size and identify each thread using a 2- or 3-component index instead. For a two-

dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + y\,D_x)$ and 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 $(x + y\,D_x + z\,D_x\,D_y)$.

2.2.2 Grid of Thread Blocks

There is a limited maximum number of threads that a block can contain. However, blocks of same dimensionality and size that execute the same kernel can be batched together into a grid of blocks, so that the total number of threads that can be launched in a single kernel invocation is much larger. This comes at the expense of reduced thread cooperation, because threads in different thread blocks from the same grid cannot communicate and synchronize with each other. This model allows kernels to efficiently run without recompilation on various devices with different parallel capabilities: A device may run all the blocks of a grid sequentially if it has very few parallel capabilities, or in parallel if it has a lot of parallel capabilities, or usually a combination of both.

Each block is identified by its *block ID*, which is the block number within the grid. To help with complex addressing based on the block ID, an application can also specify a grid as a two-dimensional array of arbitrary size and identify each block using a 2-component index instead. For a two-dimensional block of size (D_x, D_y), the block ID of a block of index (x, y) is $(x + y\,D_x)$.



The host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks

Figure 2-1. Thread Batching

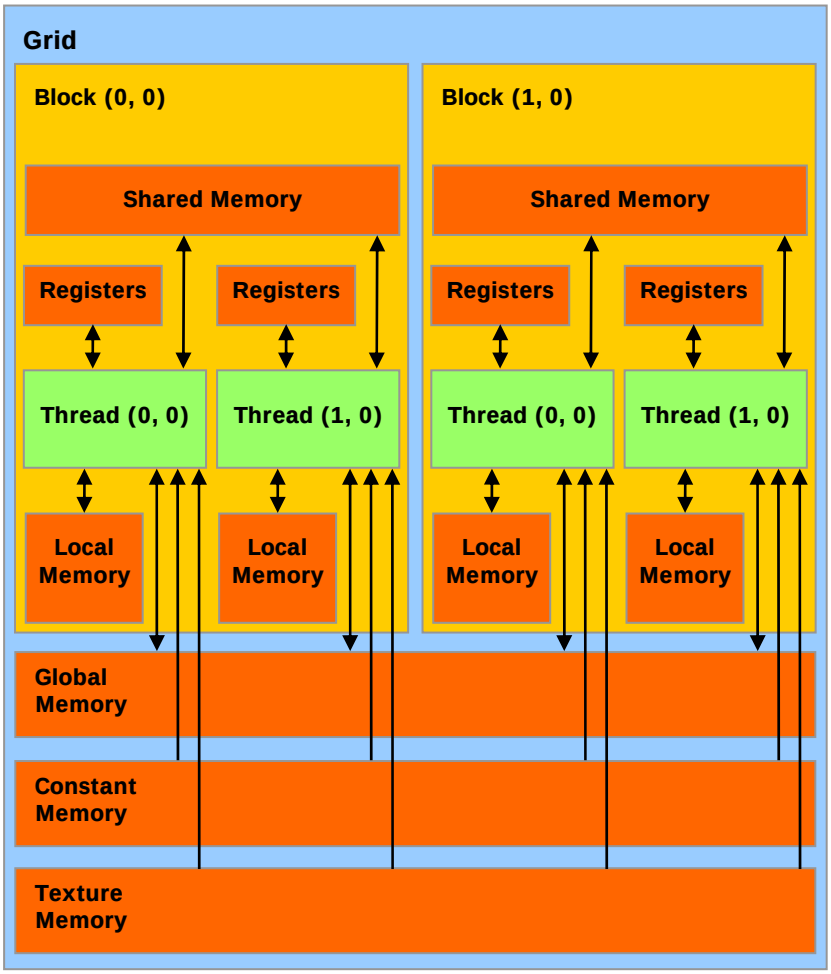
2.3 Memory Model

A thread that executes on the device has only access to the device's DRAM and on-chip memory through the following memory spaces, as illustrated in Figure 2-2:

- Read-write per-thread *registers*,
- Read-write per-thread *local memory*,
- Read-write per-block *shared memory*,
- Read-write per-grid *global memory*,
- Read-only per-grid *constant memory*,
- Read-only per-grid *texture memory*.

The global, constant, and texture memory spaces can be read from or written to by the host and are persistent across kernel launches by the same application.

The global, constant, and texture memory spaces are optimized for different memory usages (see Sections 5.1.2.1, 5.1.2.2, and 5.1.2.3). Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats (see Section 4.3.4).



A thread has access to the device's DRAM and on-chip memory through a set of memory spaces of various scopes.

Figure 2-2. Memory Model

Chapter 3. Hardware Implementation

3.1 A Set of SIMD Multiprocessors with On-Chip Shared Memory

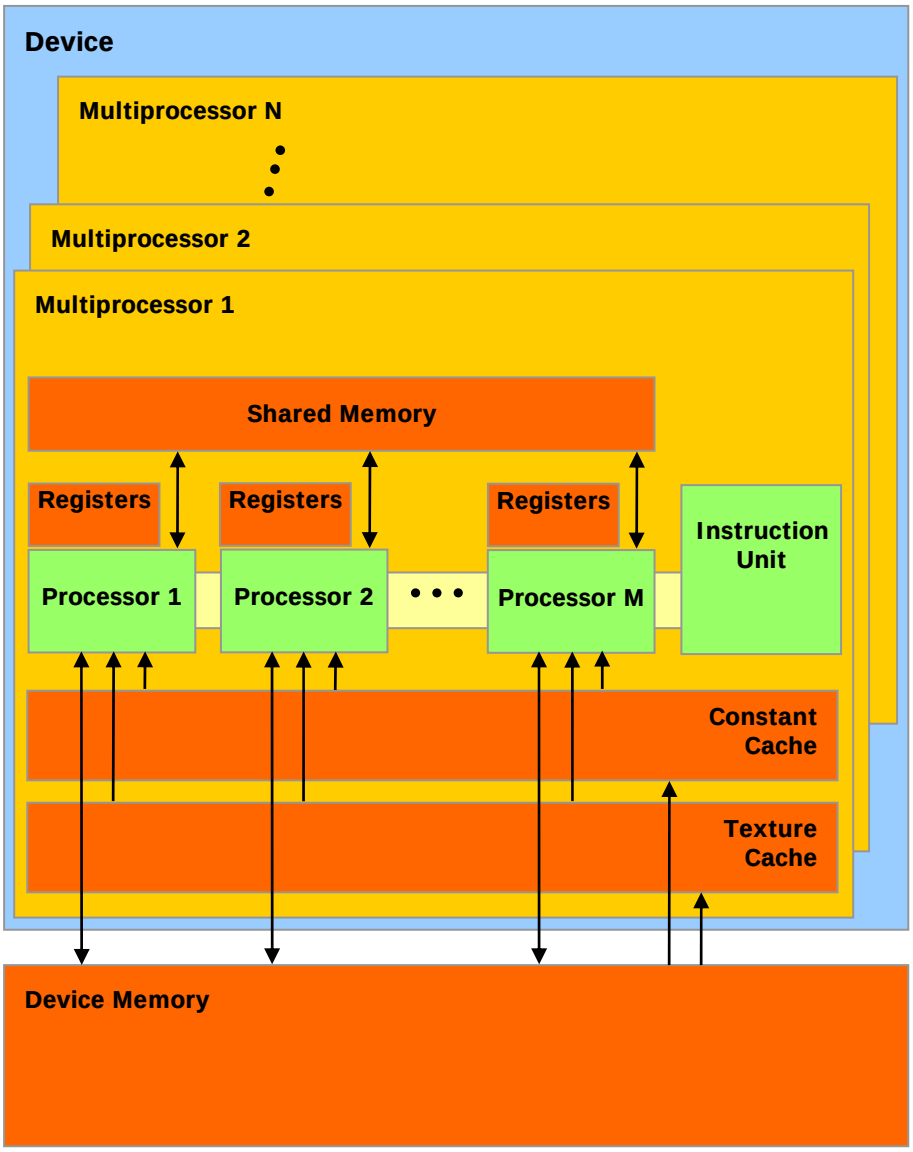
The device is implemented as a set of *multiprocessors* as illustrated in Figure 3-1. Each multiprocessor has a Single Instruction, Multiple Data architecture (SIMD): At any given clock cycle, each processor of the multiprocessor executes the same instruction, but operates on different data.

Each multiprocessor has on-chip memory of the four following types:

- One set of local 32-bit *registers* per processor,
- A parallel data cache or *shared memory* that is shared by all the processors and implements the shared memory space,
- A read-only *constant cache* that is shared by all the processors and speeds up reads from the constant memory space, which is implemented as a read-only region of device memory,
- A read-only *texture cache* that is shared by all the processors and speeds up reads from the texture memory space, which is implemented as a read-only region of device memory.

The local and global memory spaces are implemented as read-write regions of device memory and are not cached.

Each multiprocessor accesses the texture cache via a *texture unit* that implements the various addressing modes and data filtering mentioned in Section 2.3.



A set of SIMD multiprocessors with on-chip shared memory.

Figure 3-1. Hardware Model

3.2 Execution Model

A grid of thread blocks is executed on the device by executing one or more blocks on each multiprocessor using time slicing: Each block is split into SIMD groups of threads called *warps*; each of these warps contains the same number of threads, called the *warp size*, and is executed by the multiprocessor in a SIMD fashion; a *thread scheduler* periodically switches from one warp to another to maximize the use of the

multiprocessor's computational resources. A *half-warp* is either the first or second half of a warp.

The way a block is split into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. Section 2.2.1 describes how thread IDs relate to thread indices in the block.

A block is processed by only one multiprocessor, so that the shared memory space resides in the on-chip shared memory leading to very fast memory accesses. The multiprocessor's registers are allocated among the threads of the block. If the number of registers used per thread multiplied by the number of threads in the block is greater than the total number of registers per multiprocessor, the block cannot be executed and the corresponding kernel will fail to launch.

Several blocks can be processed by the same multiprocessor concurrently by allocating the multiprocessor's registers and shared memory among the blocks.

The issue order of the warps within a block is undefined, but their execution can be synchronized, as mentioned in Section 2.2.1, to coordinate global or shared memory accesses.

The issue order of the blocks within a grid of thread blocks is undefined and there is no synchronization mechanism between blocks, so threads from two different blocks of the same grid cannot safely communicate with each other through global memory during the execution of the grid.

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 and the order in which they occur is undefined, but one of the writes is guaranteed to succeed. If an atomic instruction (see Section 4.4.6) executed by a warp reads, modifies, and writes to the same location in global memory for more than one of the threads of the warp, each read, modify, write to that location occurs and they are all serialized, but the order in which they occur is undefined.

3.3 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 GeForce 8 Series, Quadro FX 5600/4600, and Tesla solutions are 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. The GeForce 8800 Series, Quadro FX 5600/4600, and Tesla solutions are of compute capability 1.0 (their minor revision number is 0) and the GeForce 8600 and 8500 Series of compute capability 1.1.

The technical specifications of the various compute capabilities are given in Appendix A.

3.4 Multiple Devices

The use of multiple GPUs as CUDA devices by an application running on a multi-GPU system is only guaranteed to work if theses GPUs are of the same type. If the system is in SLI mode however, only one GPU can be used as a CUDA device since all the GPUs are fused at the lowest levels in the driver stack. SLI mode needs to be turned off in the control panel for CUDA to be able to see each GPU as separate devices.

3.5 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, resulting in a crash of these applications.

Chapter 4. Application Programming Interface

4.1 An Extension to the C Programming Language

The goal of the CUDA programming interface is to provide a relatively 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, described in Section 4.2, that allow the programmer to target portions of the source code for execution on the device;
- A runtime library split into:
  - A host component, described in Section 4.5, that runs on the host and provides functions to control and access one or more compute devices from the host;
  - A device component, described in Section 4.4, that runs on the device and provides device-specific functions;
  - A common component, described in Section 4.3, that provides built-in vector types and a subset of the C standard library that are supported in both host and device code.

It should be emphasized that the only functions from the C standard library that are supported to run on the device are the functions provided by the common runtime component.

4.2 Language Extensions

The extensions to the C programming language are four-fold:

- Function type qualifiers to specify whether a function executes on the host or on the device and whether it is callable from the host or from the device (Section 4.2.1);
- Variable type qualifiers to specify the memory location on the device of a variable (Section 4.2.2);

- A new directive to specify how a kernel is executed on the device from the host (Section 4.2.3);
- Four built-in variables that specify the grid and block dimensions and the block and thread indices (Section 4.2.4).

Each source file containing these extensions must be compiled with the CUDA compiler **nvcc**, as briefly described in Section 4.2.5. A detailed description of **nvcc** can be found in a separate document.

Each of these extensions come with some restrictions described in each of the sections below. **nvcc** will give an error or a warning on some violations of these restrictions, but some of them cannot be detected.

4.2.1 Function Type Qualifiers

4.2.1.1 __device__

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

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

4.2.1.2 __global__

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

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

4.2.1.3 __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.

4.2.1.4 Restrictions

__device__ functions are always inlined.

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

__device__ and **__global__** functions **c**annot 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 4.2.3.

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

__global__ function parameters are currently passed via shared memory to the device and limited to 256 bytes.

4.2.2 Variable Type Qualifiers

4.2.2.1 __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.

4.2.2.2 __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.

4.2.2.3 __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.

There is full sequential consistency of shared variables within a thread, however relaxed ordering across threads. Only after the execution of a **__syncthreads()** (Section 4.4.2) do the writes from other threads are guaranteed to be visible. The compiler is free to optimize the reads and writes to shared memory as long as the previous statement is met.

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 4.2.3). 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];
}
```

4.2.2.4 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__`** cannot be used in combination with each other.

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

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

`__constant__` variables cannot be assigned to from the device, only from the host.

`__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. This is often the case for large structures or arrays that would consume too much register space, and arrays for which the compiler cannot determine that they are indexed with constant quantities. 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. 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.

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.

4.2.3 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. It is specified by inserting an expression of the form **<<< Dg, Db, Ns >>>** between the function name and the parenthesized argument list, where:

- **Dg** is of type **dim3** (see Section 4.3.1.2) and specifies the dimension and size of the grid, such that **Dg.x * Dg.y** equals the number of blocks being launched;
- **Db** is of type **dim3** (see Section 4.3.1.2) 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 4.2.2.3; **Ns** is an optional argument which defaults to 0.

The arguments to the execution configuration are evaluated before the actual function arguments.

As an example, a function declared as

```
__global__ void Func(float* parameter);
```

must be called like this:

```
Func<<< Dg, Db, Ns >>>(parameter);
```

4.2.4 Built-in Variables

4.2.4.1 gridDim

This variable is of type **dim3** (see Section 4.3.1.2) and contains the dimensions of the grid.

4.2.4.2 blockIdx

This variable is of type **uint3** (see Section 4.3.1.1) and contains the block index within the grid.

4.2.4.3 blockDim

This variable is of type **dim3** (see Section 4.3.1.2) and contains the dimensions of the block.

4.2.4.4 threadIdx

This variable is of type **uint3** (see Section 4.3.1.1) and contains the thread index within the block.

4.2.4.5 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.

4.2.5 Compilation with NVCC

nvcc is a compiler driver that simplifies the process of compiling CUDA code: It provides simple and familiar command line options and executes them by invoking the collection of tools that implement the different compilation stages.

nvcc's basic workflow consists in separating device code from host code and compiling the device code into a binary form or *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 invoking the host compiler during the last compilation stage.

Applications can either ignore the generated host code and load the *cubin* object onto the device and launch the device code using the CUDA driver API (see Section 4.5.3), or link to the generated host code, which includes the *cubin* object as a global initialized data array and contains a translation of the execution configuration syntax described in Section 4.2.3 into the necessary CUDA runtime startup code to load and launch each compiled kernel (see Section 4.5.2).

The front end of the compiler processes CUDA source files according to C++ syntax rules. However, only the C subset of C++ is supported. This means that C++ specific features such as classes, inheritance, or declaration of variables within basic blocks are not supported. 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.

A detailed description of **nvcc** can be found in a separate document.

4.3 Common Runtime Component

The common runtime component can be used by both host and device functions.

4.3.1 Built-in Vector Types

4.3.1.1 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, float1, float2, float3, float4

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)**.

4.3.1.2 `dim3` Type

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.

4.3.2 Mathematical Functions

Table B-1 contains a comprehensive list of the C/C++ standard library mathematical functions that are currently supported, along with their respective error bounds when executed on the device.

When executed in host code, a given function uses the C runtime implementation if available.

4.3.3 Time Function

```
clock_t clock();
```

returns the value of a 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.

4.3.4 Texture Type

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 5.4.

Texture memory is read from kernels using device functions called *texture fetches*, described in Section 4.4.5. 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. It must be bound through host runtime functions (Sections 0 and 4.5.3.7) 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*, or as a two-dimensional array using two 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.

4.3.4.1 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 floating-point types and any of the 1-, 2-, and 4-component vector types defined in Section 4.3.1.1;
- **Dim** specifies the dimensionality of the texture reference and is equal to 1 or 2; **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 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**.

4.3.4.2 Runtime Texture Reference Attributes

The other attributes of a texture reference are mutable and can be changed at runtime through the host runtime (Section 4.5.2.4 for the runtime API and Section 4.5.3.7 for the driver API). 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.

4.3.4.3 Texturing from Linear Memory versus CUDA Arrays

A texture can be any region of linear memory or a CUDA array (see Section 4.5.1.2).

Textures allocated in linear memory:

- Can only be of dimensionality equal to 1;
- Do not support texture filtering;
- Can only be addressed using a non-normalized integer texture coordinate;
- Do not support the various addressing modes: Out-of-range texture accesses return zero.

The hardware enforces an alignment requirement on texture base addresses. To abstract this alignment requirement from developers, the functions to bind texture references onto device memory pass back a byte offset that must be applied to texture fetches in order to read from the desired memory. The base pointers returned by CUDA's allocation routines conform to this alignment constraint, so applications can avoid the offsets altogether by passing allocated pointers to **`cudaBindTexture()`**/**`cuTexRefSetAddress()`**.

4.4 Device Runtime Component

The device runtime component can only be used in device functions.

4.4.1 Mathematical Functions

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

The compiler has an option (**`-use_fast_math`**) to force every function to compile to its less accurate counterpart if it exists.

4.4.2 Synchronization Function

```
void __syncthreads();
```

synchronizes all threads in a block. Once all threads have reached this point, execution resumes normally.

`__syncthreads()` is used to coordinate communication between the threads of a 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.

4.4.3 Type Conversion Functions

The suffixes in the function below indicate IEEE-754 rounding modes:

- **rn** is round-to-nearest-even,
- **rz** is round-towards-zero,
- **ru** is round-up (to positive infinity),
- **rd** is round-down (to negative infinity).

```
int __float2int_[rn,rz,ru,rd](float);
```

converts the floating-point argument to an integer, using the specified rounding mode.

```
unsigned int __float2uint_[rn,rz,ru,rd](float);
```

converts the floating-point argument to an unsigned integer, using the specified rounding mode.

```
float __int2float_[rn,rz,ru,rd](int);
```

converts the integer argument to a floating-point number, using the specified rounding mode.

```
float __uint2float_[rn,rz,ru,rd](unsigned int);
```

converts the unsigned integer argument to a floating-point number, using the specified rounding mode.

4.4.4 Type Casting Functions

```
float __int_as_float(int);
```

performs a floating-point type cast on the integer argument, leaving the value unchanged. For example, **__int_as_float(0xC0000000)** is equal to **-2**.

```
int __float_as_int(float);
```

performs an integer type cast on the floating-point argument, leaving the value unchanged. For example, **__float_as_int(1.0f)** is equal to **0x3f800000**.

4.4.5 Texture Functions

4.4.5.1 Texturing from Device Memory

When texturing from device memory, the texture is accessed with the **tex1Dfetch()** family of functions; for example:

```
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);
```

These functions fetch the region of linear memory bound to texture reference **texRef** using texture coordinate **x**. No texture filtering and addressing modes are supported. For integer types, these functions may optionally promote the integer to 32-bit 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 linear memory bound to texture reference **texRef** using texture coordinate **x**.

4.4.5.2 Texturing from CUDA Arrays

When texturing from CUDA arrays, the texture is accessed with the **tex1D()** or **tex2D()**:

```
template<class Type, enum cudaTextureReadMode readMode>
Type tex1D(texture<Type, 1, readMode> texRef, float x);

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

These functions fetches the CUDA array bound to texture reference **texRef** using texture coordinates **x** and **y**. A combination of the texture reference's immutable (compile-time) and mutable (runtime) attributes determine how the coordinates are interpreted, what processing occurs during the texture fetch, and the return value delivered by the texture fetch (see Sections 4.3.4.1 and 4.3.4.2).

4.4.6 Atomic Functions

Atomic functions are only available for devices of compute capability 1.1. They are listed in Appendix C.

An atomic function performs a read-modify-write atomic operation on one 32-bit word residing in global memory. For example, **atomicAdd()** reads a 32-bit word at some address in global memory, adds an integer 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 32-bit signed and unsigned integers.

4.5 Host Runtime Component

The host runtime component can only be used by host functions.

It provides functions to handle:

- Device management,
- Context management,
- Memory management,
- Code module management,
- Execution control,
- Texture reference management,
- Interoperability with OpenGL and Direct3D.

It is composed of two APIs:

- A low-level API called the *CUDA driver API*,
- A higher-level API called the *CUDA runtime API* that is implemented on top of the CUDA driver API.

These APIs are mutually exclusive: An application should use either one or the other.

The CUDA runtime eases device code management by providing implicit initialization, context management, and module management. The C host code generated by **nvcc** is based on the CUDA runtime (see Section 4.2.5), so applications that link to this code must use the CUDA runtime API.

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 only deals with *cubin* objects (see Section 4.2.5). In particular, it is more difficult to configure and launch kernels using the CUDA driver API, since the execution configuration and kernel parameters must be specified with explicit function calls instead of the execution configuration syntax described in Section 4.2.3. Also, device emulation (see Section 4.5.2.5) does not work with the CUDA driver API.

The CUDA driver API is delivered through the **cuda** dynamic library and all its entry points are prefixed with **cu**.

The CUDA runtime API is delivered through the **cudart** dynamic library and all its entry points are prefixed with **cuda**.

4.5.1 Common Concepts

4.5.1.1 Device

Both APIs provide a way to enumerate the devices available on the system, query their properties, and select one of them 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. 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.

4.5.1.2 Memory

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

Linear memory exists on the device in a 32-bit address space, so separately allocated entities can reference one another via pointers, for example, in a binary tree.

CUDA arrays are opaque memory layouts optimized for texture fetching. They are one-dimensional or two-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.

Both linear memory and CUDA arrays are only readable and writable by the host through the memory copy functions described in Sections 4.5.2.3 and 4.5.3.6.

4.5.1.3 OpenGL Interoperability

OpenGL buffer objects may be mapped into the address space of CUDA, either to enable CUDA to read data written by OpenGL or to enable CUDA to write data for consumption by OpenGL.

4.5.1.4 Direct3D Interoperability

Direct3D 9.0 vertex buffers may be mapped into the address space of CUDA, either to enable CUDA to read data written by Direct3D or to enable CUDA to write data for consumption by Direct3D.

A CUDA context may interoperate with only one Direct3D device at a time, bracketed by calls to the begin/end functions described in Sections 4.5.2.6 and 4.5.3.9. The Direct3D device must be created with the D3DCREATE_HARDWARE_VERTEXPROCESSING flag.

CUDA does not yet support:

- Versions other than Direct3D 9.0,
- Direct3D objects other than vertex buffers.

4.5.1.5 Asynchronicity

`__global__` functions and most runtime functions are asynchronous: Control is returned to the application before the device has completed the requested task.

`cudaThreadSynchronize()` for the runtime API and **`cuCtxSynchronize()`** for the driver API (described in Sections D.2.1 and E.3.5 respectively) provide applications with a way to explicitly force the runtime to wait until all preceding device tasks have finished. To avoid unnecessary slowdowns, these functions are best used for timing purposes or to isolate a launch or memory copy that is failing.

The only functions from the runtime that are not asynchronous are the functions that perform memory copies between the host and the device, the functions that initializes and terminates interoperability with a OpenGL or Direct3D, the functions that register, unregister, map, and unmap an OpenGL buffer object or a Direct3D vertex buffer, and the functions that free memory.

4.5.2 Runtime API

4.5.2.1 Initialization

There is no explicit initialization function for the runtime API; it initializes the first time a runtime function is called. 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.

4.5.2.2 Device Management

The functions from Section D.1 are used to manage the devices present in the system.

cudaGetDeviceCount() and **cudaGetDeviceProperties()** provide a way to enumerate these devices and retrieve their properties:

```
int deviceCount;
cudaGetDeviceCount(&deviceCount);
int device;
for (device = 0; device < deviceCount; ++device) {
    cudaDeviceProp deviceProp;
    cudaGetDeviceProperties(&deviceProp, device);
}
```

cudaSetDevice() is used to select the device associated to the host thread:

```
cudaSetDevice(device);
```

A device must be selected before any **__global__** function or any function from Appendix D is called. If this is not done by an explicit call to **cudaSetDevice()**, device 0 is automatically selected and any subsequent explicit call to **cudaSetDevice()** will have no effect.

4.5.2.3 Memory Management

The functions from Section D.3 are used to allocate and free device memory, access the memory allocated for any variable declared in global memory space, and transfer data between host and device memory.

Linear memory is allocated using **cudaMalloc()** or **cudaMallocPitch()** and freed using **cudaFree()**.

The following code sample allocates an array of 256 floating-point elements in linear memory:

```
float* devPtr;
cudaMalloc((void**)&devPtr, 256 * sizeof(float));
```

cudaMallocPitch() 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 5.1.2.1, therefore ensuring best performance when accessing the row addresses or performing copies between 2D arrays and other regions of device memory (using the **cudaMemcpy2D()** 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, 192>>>(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];
        }
    }
}
```

CUDA arrays are allocated using **cudaMallocArray()** and freed using **cudaFreeArray()**. **cudaMallocArray()** requires a format description created using **cudaCreateChannelDesc()**.

The following code sample allocates a **width×height** CUDA array of one 32-bit floating-point component:

```
cudaChannelFormatDesc channelDesc =
                             cudaCreateChannelDesc<float>();
cudaArray* cuArray;
cudaMallocArray(&cuArray, &channelDesc, width, height);
```

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()**.

Section D.3 lists all the various functions used to copy memory between linear memory allocated with **cudaMalloc()**, linear memory allocated with **cudaMallocPitch()**, CUDA arrays, and memory allocated for variables declared in global or constant memory space.

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

```
cudaMemcpy2DToArray(cuArray, 0, 0, devPtr, pitch,
                    width * sizeof(float), height,
                    cudaMemcpyDeviceToDevice);
```

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

```
float data[256];
int size = sizeof(data);
float* devPtr;
cudaMalloc((void**)&devPtr, size);
cudaMemcpy(devPtr, data, size, cudaMemcpyHostToDevice);
```

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

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

Finally, **cudaMallocHost()** from Section D.3.6 and **cudaFreeHost()** from Section D.3.7 can be used to allocate and free page-locked host memory. The bandwidth between host memory and device memory is higher for page-locked host memory than for regular pageable memory allocated using **malloc()**. However, page-locked memory is a scarce resource, 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

4.5.2.4 Texture Reference Management

The functions from Section D.4 are used to manage texture references.

The **texture** type defined by the high-level API is a structure publicly derived from the **textureReference** type defined by the low-level API as such:

```
struct textureReference
{
  int                           normalized;
  enum cudaTextureFilterMode    filterMode;
  enum cudaTextureAddressMode   addressMode[2];
  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]** or **[0,height-1]**, where **width** and **height** 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) or four (for a two-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 two whose first and second elements specify the addressing mode for the first and second 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. They only apply to texture references bound to CUDA arrays.

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()**.

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>();
cudaBindTexture(0, texRefPtr, devPtr, &channelDesc, size);
```

- Using the high-level API:

```
texture<float, 2, cudaReadModeElementType> texRef;
cudaBindTexture(0, texRef, devPtr, size);
```

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.

cudaUnbindTexture() is used to unbind a texture reference.

4.5.2.5 OpenGL Interoperability

The functions from Section D.6 are used to control interoperability with OpenGL.

A buffer object must be registered to CUDA before it can be mapped. This is done with **cudaGLRegisterBufferObject()**:

```
GLuint bufferObj;
cudaGLRegisterBufferObject(bufferObj);
```

Once it is registered, a buffer object can be read from or written to by kernels using the device memory address returned by **cudaGLMapBufferObject()**:

```
GLuint bufferObj;
float* devPtr;
cudaGLMapBufferObject((void**)&devPtr, bufferObj);
```

Unmapping is done with **cudaGLUnmapBufferObject()** and unregistering with **cudaGLUnregisterBufferObject()**.

4.5.2.6 Direct3D Interoperability

The functions from Section D.7 are used to control interoperability with Direct3D.

Interoperability with Direct3D must be initialized using **cudaD3D9Begin()** and terminated using **cudaD3D9End()**.

In between these calls, a vertex object must be registered to CUDA before it can be mapped. This is done with **cudaD3D9RegisterVertexBuffer()**:

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
cudaD3D9RegisterVertexBuffer(vertexBuffer);
```

Once it is registered, a vertex buffer can be read from or written to by kernels using the device memory address returned by **cudaD3D9MapVertexBuffer()**:

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
float* devPtr;
cudaD3D9MapVertexBuffer((void**)&devPtr, vertexBuffer);
```

Unmapping is done with **cudaD3D9UnmapVertexBuffer()** and unregistering with **cudaD3D9UnregisterVertexBuffer()**.

4.5.2.7 Debugging using the Device Emulation Mode

The programming environment does not include any native debug support for code that runs on the device, but comes with a device emulation mode for the purpose of debugging. When compiling an application is this mode (using the **-deviceemu** option), the device code is compiled for and runs on the host, allowing the developer 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 developer 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 developers 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, the runtime detects dead lock situations.

Developers 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:

- When a memory location is accessed in multiple threads within the grid at potentially the same time, the results when running in device emulation mode potentially differ from the results when running on the device, since in emulation mode threads execute sequentially.
- 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, developers 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.

  Unlike compute devices (see Appendix A), host platforms also usually support denormalized numbers. 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.

4.5.3 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 CUDA are summarized in Table 4-1.

Table 4-1. Objects Available in the CUDA Driver API

Object	Handle	Description
Device	CUdevice	CUDA-capable 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

4.5.3.1 Initialization

Initialization with **`cuInit()`** is required before any function from Appendix E is called (see Section E.1).

4.5.3.2 Device Management

The functions from Section E.2 are used to manage the devices present in the system.

`cuDeviceGetCount()` and **`cuDeviceGet()`** provide a way to enumerate these devices and other functions from Section E.2 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);
}
```

4.5.3.3 Context Management

The functions from Section E.3 are used to create, attach, and detach CUDA contexts.

A CUDA context is analogous to a CPU process. All resources and actions performed within the compute 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 CUDA contexts reference different memory locations.

Contexts have a one-to-one correspondence with host threads. 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.

To facilitate interoperability between third party authored code operating in the same context, the driver API maintains a usage count that is incremented by each distinct client of a given context. For example, if three libraries are loaded to use the same CUDA context, each library must call **cuCtxAttach()** to increment the usage count and **cuCtxDetach()** to decrement the usage count when the library is done using the context. The context is destroyed when the usage count goes to 0. For most libraries, it is expected that the application will have created a CUDA 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.

4.5.3.4 Module Management

The functions from Section E.4 are used to load and unload modules and to retrieve handles or pointers to variables or functions defined in the module.

Modules are dynamically loadable packages of device code and data, akin to DLLs in Windows, that are output by **nvcc** (see Section 4.2.5). 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.cubin”);
CUfunction cuFunction;
cuModuleGetFunction(&cuFunction, cuModule, “myKernel”);
```

4.5.3.5 Execution Control

The functions described in Section E.5 manage the execution of a kernel on the device. **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 specify the parameters that will be provided to the kernel the next time **cuLaunchGrid()** or **cuLaunch()** is invoked to launch the kernel:

```
cuFuncSetBlockShape(cuFunction, blockWidth, blockHeight, 1);
int offset = 0;
int i;
cuParamSeti(cuFunction, offset, i);
offset += sizeof(i);
float f;
cuParamSetf(cuFunction, offset, f);
offset += sizeof(f);
char data[256];
```

```
cuParamSetv(cuFunction, offset, (void*)data, sizeof(data));
offset += sizeof(data);
cuParamSetSize(cuFunction, offset);
cuFuncSetSharedSize(cuFunction, numElements * sizeof(float));
cuLaunchGrid(cuFunction, gridWidth, gridHeight);
```

4.5.3.6 Memory Management

The functions from Section E.6 are used to allocate and free device memory and transfer data between host and device memory.

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

The following code sample allocates an array of 256 floating-point elements in linear memory:

```
CUdeviceptr devPtr;
cuMemAlloc(&devPtr, 256 * sizeof(float));
```

`cuMemAllocPitch()` 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 5.1.2.1, 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
CUdeviceptr devPtr;
int pitch;
cuMemAllocPitch(&devPtr, &pitch,
                width * sizeof(float), height, 4);
cuModuleGetFunction(&cuFunction, cuModule, “myKernel”);
cuFuncSetBlockShape(cuFunction, 192, 1, 1);
cuParamSeti(cuFunction, 0, devPtr);
cuParamSetSize(cuFunction, sizeof(devPtr));
cuLaunchGrid(cuFunction, 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];
        }
    }
}
```

CUDA arrays are created using **`cuArrayCreate()`** and destroyed using **`cuArrayDestroy()`**.

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);
```

Section E.6 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 device memory:

```
float data[256];
int size = sizeof(data);
CUdeviceptr devPtr;
cuMemAlloc(&devPtr, size);
cuMemcpyHtoD(devPtr, data, size);
```

Finally, **cuMemAllocHost()** from Section E.6.5 and **cuMemFreeHost()** from Section E.6.6 can be used to allocate and free page-locked host memory. The bandwidth between host memory and device memory is higher for page-locked host memory than for regular pageable memory allocated using **malloc()**. However, page-locked memory is a scarce resource, 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.

4.5.3.7 Texture Reference Management

The functions from Section E.7 are used to manage texture references.

Before a kernel can use a texture reference to read from texture memory, the texture reference must be bound to a texture using **cuTexRefSetAddress()** or **cuTexRefSetArray()**.

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**:

```
cuTexRefSetAddress(NULL, cuTexRef, devPtr, size);
```

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

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

Section E.7 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.

4.5.3.8 OpenGL Interoperability

The functions from Section E.8 are used to control interoperability with OpenGL.

Interoperability with OpenGL must be initialized using **cuGLInit()**.

A buffer object must be registered to CUDA before it can be mapped. This is done with **cuGLRegisterBufferObject()**:

```
GLuint bufferObj;
cuGLRegisterBufferObject(bufferObj);
```

Once it is registered, a buffer object can be read from or written to by kernels using the device memory address returned by **cuGLMapBufferObject()**:

```
GLuint bufferObj;
CUdeviceptr devPtr;
int size;
cuGLMapBufferObject(&devPtr, &size, bufferObj);
```

Unmapping is done with **cuGLUnmapBufferObject()** and unregistering with **cuGLUnregisterBufferObject()**.

4.5.3.9 Direct3D Interoperability

The functions from Section D.7 are used to control interoperability with Direct3D.

Interoperability with Direct3D must be initialized using **cuD3D9Begin()** and terminated using **cuD3D9End()**.

In between these calls, a vertex object must be registered to CUDA before it can be mapped. This is done with **cuD3D9RegisterVertexBuffer()**:

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
cuD3D9RegisterVertexBuffer(vertexBuffer);
```

Once it is registered, a vertex buffer can be read from or written to by kernels using the device memory address returned by **cuD3D9MapVertexBuffer()**:

```
LPDIRECT3DVERTEXBUFFER9 vertexBuffer;
CUdeviceptr devPtr;
int size;
cuD3D9MapVertexBuffer(&devPtr, &size, vertexBuffer);
```

Unmapping is done with **cuD3D9UnmapVertexBuffer()** and unregistering with **cuD3D9UnregisterVertexBuffer()**.

Chapter 5. Performance Guidelines

5.1 Instruction Performance

To process an instruction for a warp of threads, a multiprocessor must:

- Read the instruction operands for each thread of the warp,
- Execute the instruction,
- Write the result for each thread of the warp.

Therefore, the effective instruction throughput depends on the nominal instruction throughput as well as the memory latency and bandwidth. It is maximized by:

- Minimizing the use of instructions with low throughput (see Section 5.1.1),
- Maximizing the use of the available memory bandwidth for each category of memory (see Section 5.1.2),
- Allowing the thread scheduler to overlap memory transactions with mathematical computations as much as possible, which requires that:
  - The program executed by the threads is of high arithmetic intensity, that is, has a high number of arithmetic operations per memory operation;
  - There are many threads that can be run concurrently as detailed in Section 5.2.

5.1.1 Instruction Throughput

5.1.1.1 Arithmetic Instructions

To issue one instruction for a warp, a multiprocessor takes:

- 4 clock cycles for floating-point add, floating-point multiply, floating-point multiply-add, integer add, bitwise operations, compare, min, max, type conversion instruction;
- 16 clock cycles for reciprocal, reciprocal square root, **`__log(x)`** (see Table B-2).

32-bit integer multiplication takes 16 clock cycles, but **`__mul24`** and **`__umul24`** (see Appendix B) provide signed and unsigned 24-bit integer multiplication in 4 clock cycles. On future architectures however, **`__[u]mul24`** will be slower than 32-bit integer multiplication, so we recommend to provide two kernels, one using

__[u]mul24 and the other using generic 32-bit integer multiplication, to be called appropriately by the application.

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.

Other functions take more clock cycles as they are implemented as combinations of several instructions.

Floating-point square root is implemented as a reciprocal square root followed by a reciprocal, so it takes 32 clock cycles for a warp.

Floating-point division takes 36 clock cycles, but **__fdividef(x, y)** provides a faster version at 20 clock cycles (see Appendix B).

__sin(x), **__cos(x)**, **__exp(x)** take 32 clock cycles.

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

- Functions operating on **char** or **short** whose operands generally need to be converted to **int**,
- Double-precision floating-point constants (defined without any type suffix) used as input to single-precision floating-point computations,
- Single-precision floating-point variables used as input parameters to the double-precision version of the mathematical functions defined in Table B-1.

The two last cases can be avoided by using:

- Single-precision floating-point constants, defined with an **f** suffix such as **3.141592653589793f**, **1.0f**, **0.5f**,
- The single-precision version of the mathematical functions, defined with an **f** suffix as well, such as **sinf()**, **logf()**, **expf()**.

For single precision code, we highly recommend use of the float type and the single precision math functions. When compiling for devices without native double precision support, such as devices of compute capability 1.x, the double type gets demoted to float by default and the double precision math functions are mapped to their single precision equivalents. However, on those future devices that will support double precision, these functions will map to double precision implementations.

5.1.1.2 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, that is, 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 3.2. A trivial example is when the controlling condition only depends on **(threadIdx / WSIZE)** where **WSIZE** 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.

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.

5.1.1.3 Memory Instructions

Memory instructions include any instruction that reads from or writes to shared or global memory. A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp. When accessing global memory, there are, in addition, 400 to 600 clock cycles of memory latency.

As an example, the assignment operator in the following sample code:

```
__shared__ float shared[32];
__device__ float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

takes 4 clock cycles to issue a read from global memory, 4 clock cycles to issue a write to shared memory, but above all 400 to 600 clock cycles to read a float from global memory.

Much of this global memory latency can be hidden by the thread scheduler if there are sufficient independent arithmetic instructions that can be issued while waiting for the global memory access to complete.

5.1.1.4 Synchronization Instruction

`__syncthreads` takes 4 clock cycles to issue for a warp if no thread has to wait for any other threads.

5.1.2 Memory Bandwidth

The effective bandwidth of each memory space depends significantly on the memory access pattern as detailed in the following sub-sections.

Since device memory is of much higher latency and lower bandwidth than on-chip memory, device memory accesses should be minimized. 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 written 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.

5.1.2.1 Global Memory

The global memory space is not cached, so it is all the more important to follow the right access pattern to get maximum memory bandwidth, especially given how costly accesses to device memory are.

First, the device is capable of reading 32-bit, 64-bit, or 128-bit words from global memory into registers in a single instruction. To have assignments such as:

```
__device__ type device[32];
type data = device[tid];
```

compile to a single load instruction, **`type`** must be such that **`sizeof(type)`** is equal to 4, 8, or 16 and variables of type **`type`** must be aligned to 4, 8, or 16 bytes (that is, have the 2, 3, or 4 least significant bits of their address equal to zero).

The alignment requirement is automatically fulfilled for built-in types of Section 4.3.1.1 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 a;
    float b;
};
```

or

```
struct __align__(16) {
    float a;
    float b;
    float c;
    float d;
};
```

For structures larger than 16 bytes, the compiler generates several load instructions. To ensure that it generates the minimum number of instructions, such structures should be defined with **`__align__(16)`** , such as

```
struct __align__(16) {
    float a;
    float b;
    float c;
    float d;
    float e;
};
```

which is compiled into two 128-bit load instructions instead of five 32-bit load instructions.

Second, the global memory addresses simultaneously accessed by each thread of a half-warp during the execution of a single read or write instruction should be

arranged so that the memory accesses can be coalesced into a single contiguous, aligned memory access.

More precisely, in each half-warp, thread number **N** within the half-warp should access address

```
HalfWarpBaseAddress + N
```

where **HalfWarpBaseAddress** is of type **type*** and **type** is such that it meets the size and alignment requirements discussed above. Moreover, **HalfWarpBaseAddress** should be aligned to **16*sizeof(type)** bytes; in other words, it should have its **log2(16*sizeof(type))** least significant bits equal to zero. Any address **BaseAddress** of a variable residing in global memory or returned by one of the memory allocation routines from Sections D.3 or E.6 is always aligned to at least 256 bytes, so to satisfy the memory alignment constraint, **HalfWarpBaseAddress-BaseAddress** should be a multiple of **16*sizeof(type)**.

Note that if a half-warp fulfills all the requirements above, the per-thread memory accesses are coalesced even if some threads of the half-warp do not actually access memory.

We recommend fulfilling the coalescing requirements for the entire warp as opposed to only each of its halves separately because future devices will necessitate it for proper coalescing.

A common global memory access pattern is when each thread of thread ID **tid** accesses one element of an array located at address **BaseAddress** of type **type*** using the following address:

```
BaseAddress + tid
```

To get memory coalescing, **type** must meet the size and alignment requirements discussed above. In particular, this means that if **type** is a structure larger than 16 bytes, it should be split into several structures that meet these requirements and the data should be laid out in memory as a list of several arrays of these structures instead of a single array of type **type***.

Another common global memory access pattern is when each thread of index **(tx,ty)** accesses one element of a 2D array located at address **BaseAddress** of type **type*** and of width **width** using the following address:

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

In such a case, one gets memory coalescing for all half-warps of the thread block only if:

- The width of the thread block is a multiple of half the warp size;
- **width** is a multiple of 16.

In particular, this means that an array whose width is not a multiple of 16 will be accessed much more efficiently if it is actually allocated with a width rounded up to the closest multiple of 16 and its rows padded accordingly. The **cuMemAllocPitch()** and **cudaMallocPitch()** functions and associated memory copy functions described in Sections D.3 and E.6 enable developers to write non-hardware-dependent code to allocate arrays that conform to these constraints.

5.1.2.2 Constant Memory

The constant memory space is cached so a read from constant memory costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the constant cache.

For all threads of a half-warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. The cost scales linearly with the number of different addresses read by all threads. We recommend having all threads of the entire warp read the same address as opposed to all threads within each of its halves only, as future devices will require it for full speed read.

5.1.2.3 Texture Memory

The texture memory space is cached so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from the 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 will achieve best performance.

Reading device memory through texture fetching can be an advantageous alternative to reading device memory from global or constant memory as detailed in Section 5.4.

5.1.2.4 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 the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads, as detailed below.

To achieve high memory bandwidth, shared memory is divided into equally-sized memory modules, called banks, which can be accessed simultaneously. So, any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an effective 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 the effective bandwidth 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.

In the case of the shared memory space, the banks are organized such that successive 32-bit words are assigned to successive banks and each bank has a bandwidth of 32 bits per two clock cycles.

For devices of compute capability 1.x, the warp size is 32 and the number of banks is 16 (see Section 5.1); a shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp. 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.

A common case 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, the threads **tid** and **tid+n** access the same bank whenever **s*n** is a multiple of the number of banks **m** or equivalently, whenever **n** is a multiple of **m/d** where **d** is the greatest common divisor of **m** and **s**. As a consequence, there will be no bank conflict only if half the warp size is less than or equal to **m/d**. For devices of compute capability 1.x, this translates to no bank conflict only if **d** is equal to 1, or in other words, only if **s** is odd since **m** is a power of two.

Figure 5-1 and Figure 5-2 show some examples of conflict-free memory accesses while Figure 5-3 shows some examples of memory accesses that cause bank conflicts.

Other cases worth mentioning are when each thread accesses an element that is smaller or larger than 32 bits in size. For example, there will be 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 will not be any bank conflict however, if the same array is accessed the following way:

```
char data = shared[BaseIndex + 4 * tid];
```

A structure assignment is compiled into as many memory requests as there are members in the structure, so the following code, for example:

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

results in:

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

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

  since each member is accessed with a stride of three 32-bit words;

- Two separate memory reads with bank conflicts if **type** is defined as

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

  since each member is accessed with a stride of two 32-bit words;

- Two separate memory reads with bank conflicts if **type** is defined as

  ```
  struct type {
      float f;
      char  c;
  };
  ```

  since each member is accessed with a stride of five bytes.

Finally, shared memory also 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

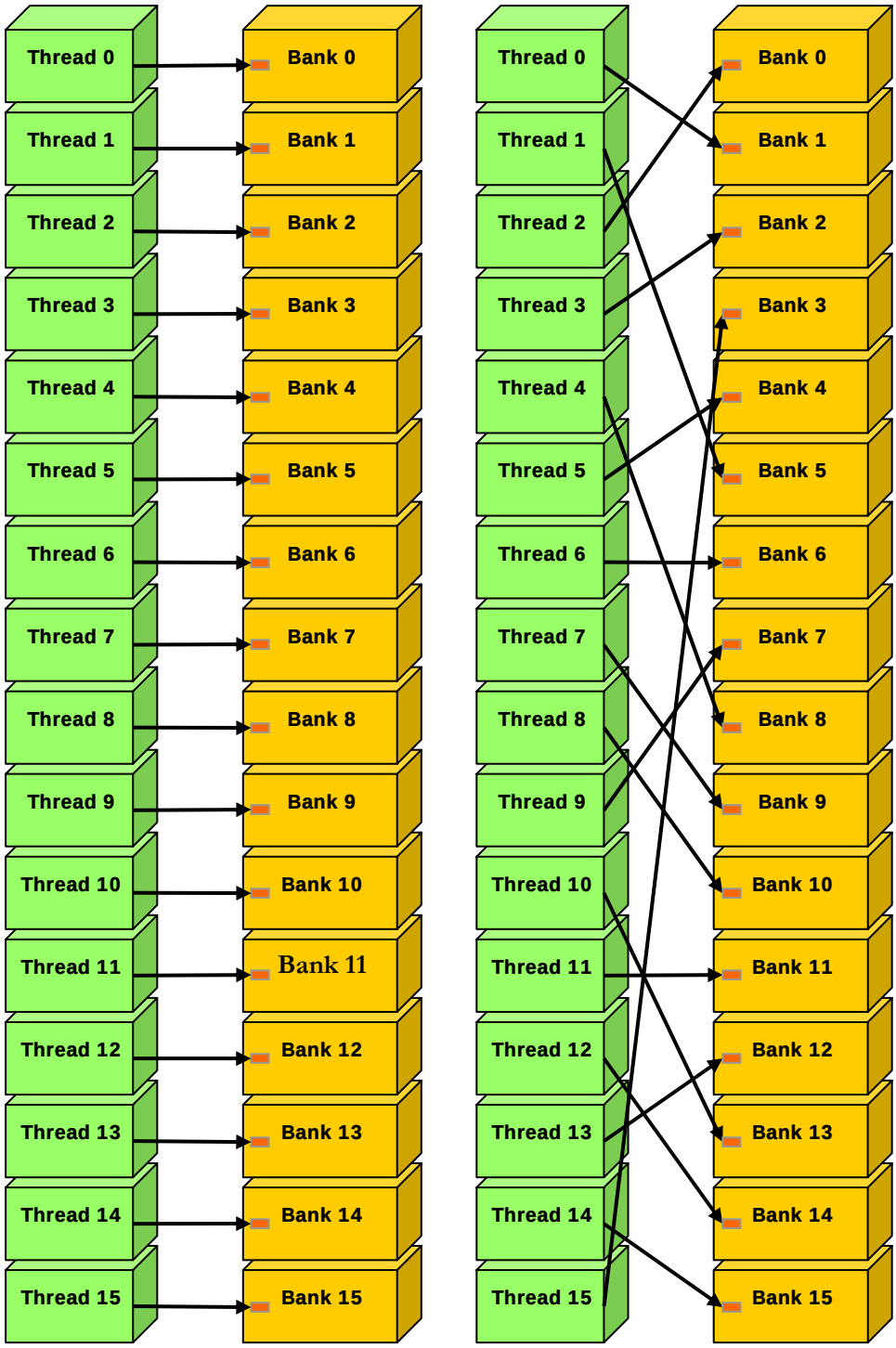
threads of a half-warp read from an address within the same 32-bit word. More precisely, a memory read request made of several addresses is serviced in several steps over time – one step every two clock cycles – by servicing one conflict-free subset of these addresses per step until all addresses have 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 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.

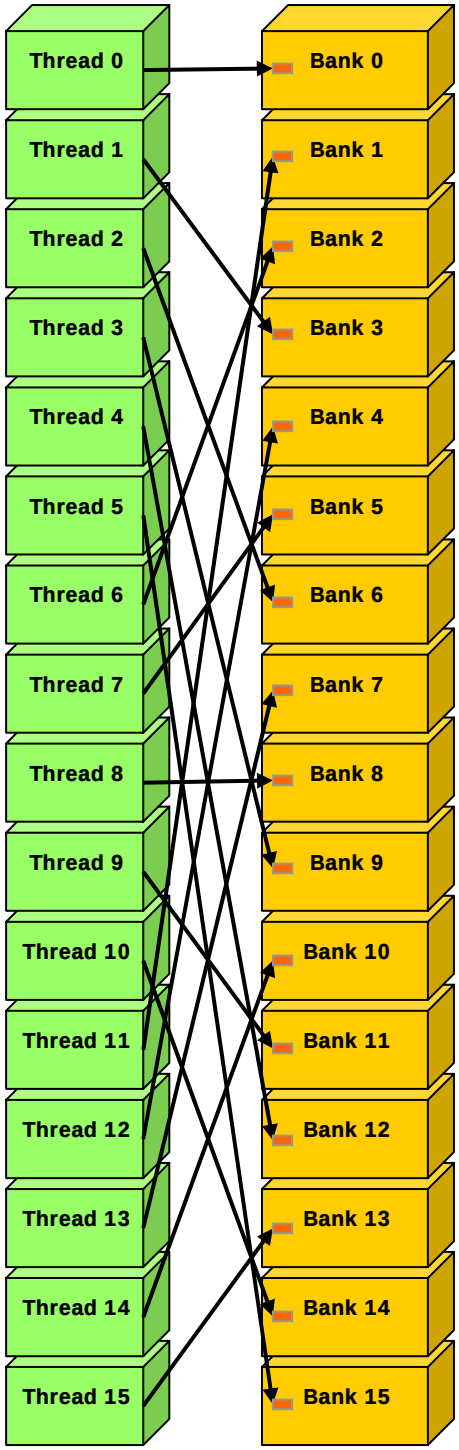
A common conflict-free case is when all threads of a half-warp read from an address within the same 32-bit word.

Figure 5-4 shows some examples of memory read accesses that involve the broadcast mechanism.



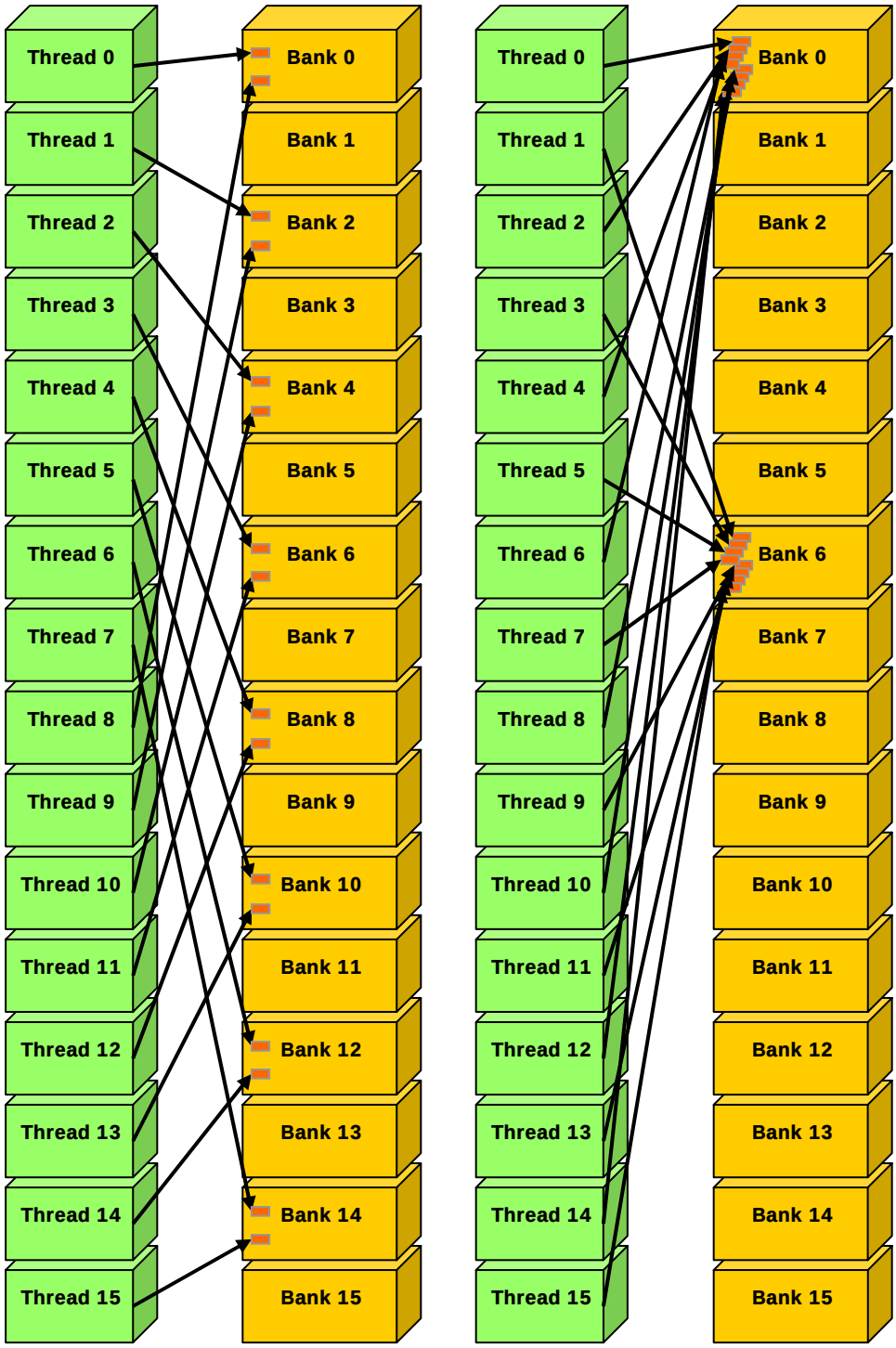
Left: linear addressing with a stride of one 32-bit word.
Right: random permutation.

Figure 5-1. Examples of Shared Memory Access Patterns without Bank Conflicts



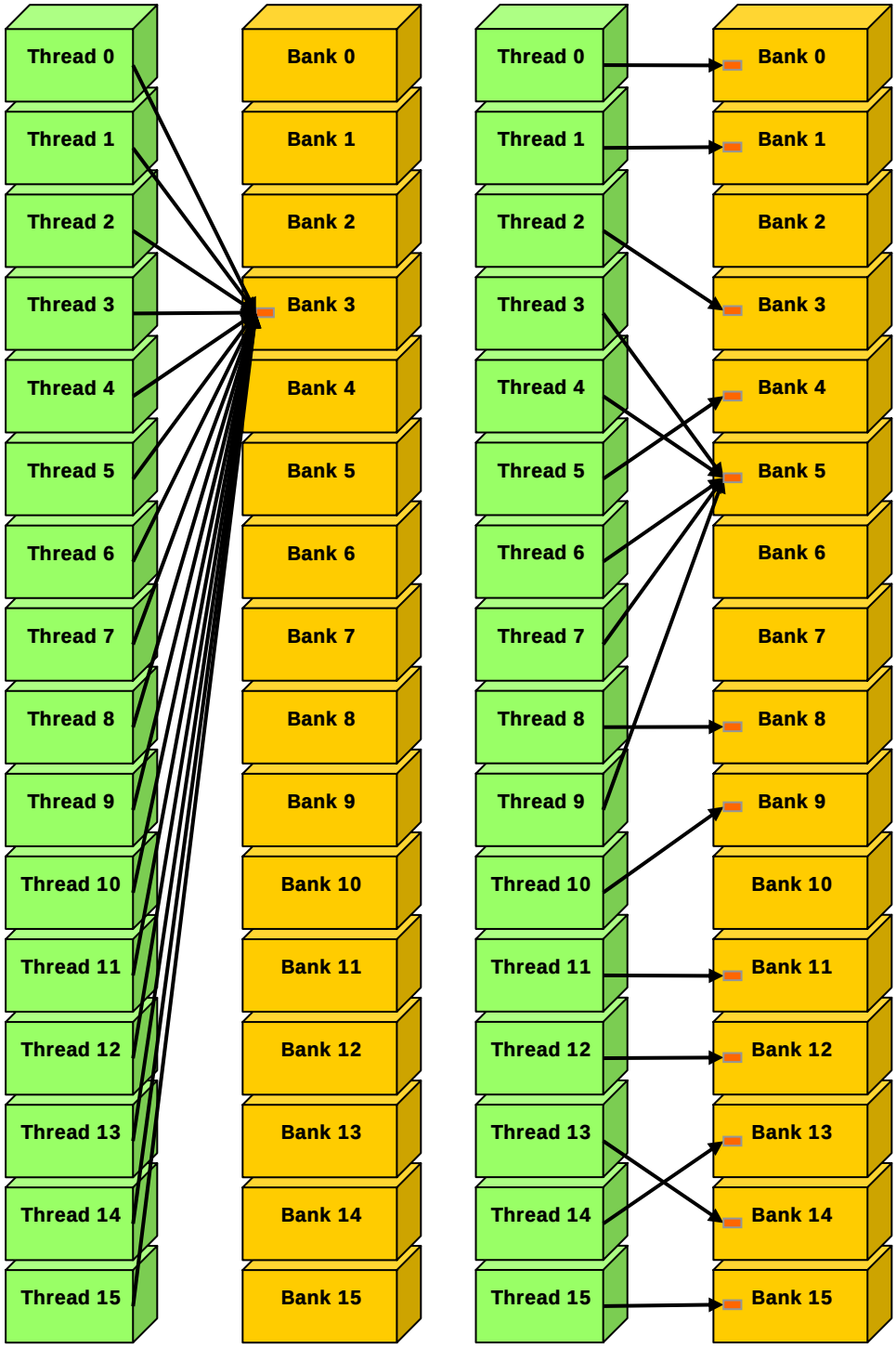
Linear addressing with a stride of three 32-bit words.

Figure 5-2. Example of a Shared Memory Access Pattern without Bank Conflicts



Left: Linear addressing with a stride of two 32-bit words causes 2-way bank conflicts.
Right: Linear addressing with a stride of eight 32-bit words causes 8-way bank conflicts.

Figure 5-3. Examples of Shared Memory Access Patterns with Bank Conflicts



Left: This access pattern is conflict-free since all threads read from an address within the same 32-bit word.

Right: This access pattern causes either no bank conflicts if the word from bank 5 is chosen as the broadcast word during the first step or 2-way bank conflicts, otherwise.

Figure 5-4. Example of Shared Memory Read Access Patterns with Broadcast

5.1.2.5 Registers

Generally, accessing a register is zero extra clock cycles per instruction, but delays may occur due to register read-after-write dependencies and register memory bank conflicts.

The delays introduced by read-after-write dependencies can be ignored as soon as there are at least 192 concurrent threads per multiprocessor to hide them.

The compiler and thread scheduler schedule the instructions as optimally as possible to avoid register memory bank conflicts. They achieve best results when the number of threads per block is a multiple of 64. Other than following this rule, an application has no direct control over these bank conflicts. In particular, there is no need to pack data into **`float4`** or **`int4`** types.

5.2 Number of Threads per Block

Given a total number of threads per grid, the number of threads per block, or equivalently the number of blocks, should be chosen to maximize the utilization of the available computing resources. This means that there should be at least as many blocks as there are multiprocessors in the device.

Furthermore, running only one block per multiprocessor will force the multiprocessor to idle during thread synchronization and also during device memory reads if there are not enough threads per block to cover the load latency. It is therefore better to allow for two or more blocks to run concurrently on each multiprocessor to allow overlap between blocks that wait and blocks that can run. For this to happen, not only should there be at least twice as many blocks as there are multiprocessors in the device, but also the amount of allocated shared memory per block should be at most half the total amount of shared memory available per multiprocessor (see Section 3.2). More thread blocks stream in pipeline fashion through the device and amortize overhead even more.

With a high enough number of blocks, 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, or better, a multiple of 64 for the reason invoked in Section 5.1.2.5. Allocating more threads per block is better for efficient time slicing, but the more threads per block, the fewer registers are available per thread. This might prevent a kernel invocation from succeeding if the kernel compiles to more registers than are allowed by the execution configuration.

For devices of compute capability 1.x, the number of registers available per thread is equal to:

$$\frac{\mathrm{R}}{B \times ceil(T, 32)}$$

where R is the total number of registers per multiprocessor given in Appendix A, B is the number of concurrent blocks, T is the number of threads per block, and $ceil(T, 32)$ is T rounded up to the nearest multiple of 32.

64 threads per block is minimal and makes sense only if there are multiple concurrent blocks. 192 or 256 threads per block is better and usually allows for enough registers to compile.

The number of blocks per grid should be at least 100 if one wants it to scale to future devices; 1000 blocks will scale across several generations.

The ratio of the number of warps running concurrently on a multiprocessor to the maximum number of warps that can run concurrently (given in Appendix A) is called the multiprocessor *occupancy*. In order to maximize occupancy, the compiler attempts to minimize register usage and programmers need to choose execution configurations with care. The CUDA Software Development Kit provides a spreadsheet to assist programmers in choosing thread block size based on shared memory and register requirements.

5.3 Data Transfer between Host and Device

The bandwidth between the device and the device memory is much higher than the bandwidth between the device memory and the host memory. Therefore, one should strive to minimize data transfer between the host and the device. For example, 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 big one always performs much better than making each transfer separately.

5.4 Benefits of Texture Memory

Device memory reads through texture fetching present several benefits over reads from global or constant memory:

- They are cached, potentially exhibiting higher bandwidth if there is locality in the texture fetches;
- They are not subject to the constraints on memory access patterns that global or constant memory reads must respect to get good performance (see Sections 5.1.2.1 and 5.1.2.2);
- The latency of addressing calculations is hidden better, possibly improving performance for applications that perform random accesses to the data;
- 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, 1].

Note that the texture cache is not kept coherent with respect to global memory accesses, so the global memory ranges that a kernel is operating on must not overlap with memory accessed by the texturing hardware. This restriction only applies within a given kernel launch, however; separate kernel launches may freely intermix writing to device memory and reading from the same device memory via texture, provided the device memory ranges do not overlap during a launch.

If the texture is a CUDA array (see Section 4.3.4.2), the hardware provides other capabilities that may be useful for different applications, especially image processing:

Feature	Useful for…	Caveat
Filtering	Fast, low-precision interpolation between texels	Only valid if the texture reference returns floating-point data
Normalized texture coordinates	Resolution-independent coding	
Addressing modes	Automatic handling of boundary cases	Can only be used with normalized texture coordinates

Chapter 6. Example of Matrix Multiplication

6.1 Overview

The task of computing the product C of two matrices A and B of dimensions (wA, hA) and (wB, wA) respectively, is split among several threads in the following way:

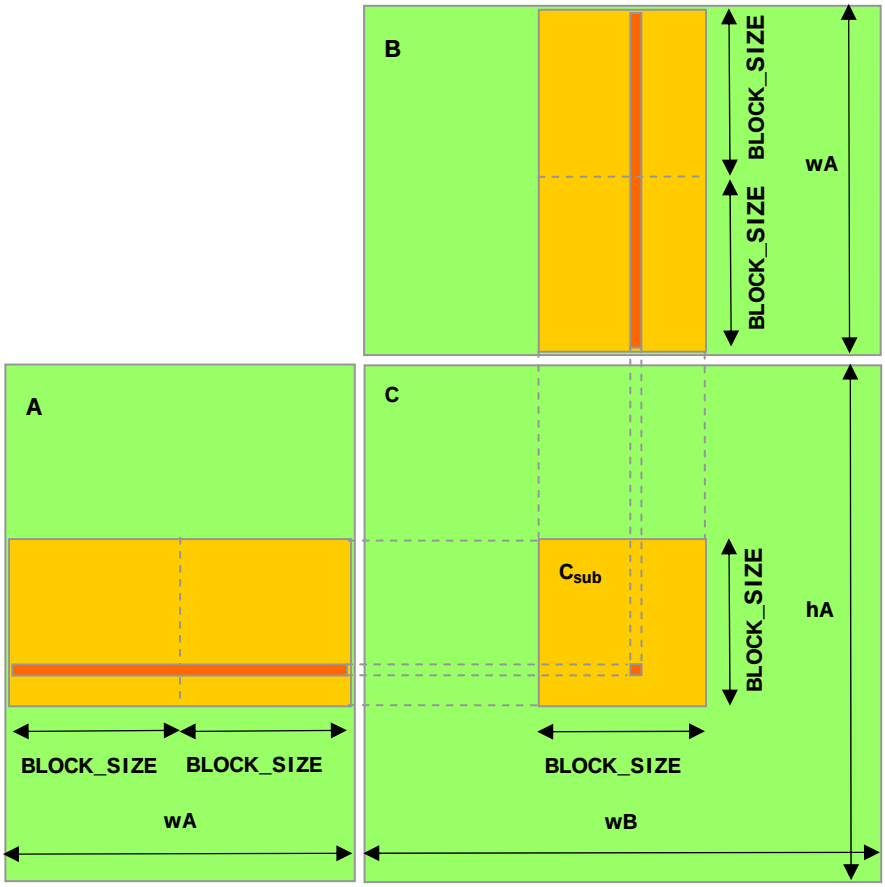
- Each thread block is responsible for computing one square sub-matrix C_{sub} of C;
- Each thread within the block is responsible for computing one element of C_{sub}.

The dimension *block_size* of C_{sub} is chosen equal to 16, so that the number of threads per block is a multiple of the warp size (Section 5.2) and remains below the maximum number of threads per block (Appendix A).

As illustrated in Figure 6-1, C_{sub} is equal to the product of two rectangular matrices: the sub-matrix of A of dimension $(wA, block\_size)$ that has the same line indices as C_{sub}, and the sub-matrix of B of dimension $(block\_size, wA)$ 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 and B are read from global memory only $(wA / block\_size)$ times.

Nonetheless, this example has been written for clarity of exposition to illustrate various CUDA programming principles, not with the goal of providing a high-performance kernel for generic matrix multiplication and should not be construed as such.



Each thread block computes one sub-matrix C_{sub} of C. Each thread within the block computes one element of C_{sub}.

Figure 6-1. Matrix Multiplication

6.2 Source Code Listing

```
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication function
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
//   hA is the height of A
//   wA is the width of A
//   wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB,
         float* C)
{
    int size;

    // Load A and B to the device
    float* Ad;
    size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
```

```
// Device multiplication function called by Mul()
// Compute C = A * B
//   wA is the width of A
//   wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C)
{
    // Block index
    int bx = blockIdx.x;
    int by = blockIdx.y;

    // Thread index
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd   = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep  = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep  = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {

        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
```

```
            Csub += As[ty][k] * Bs[k][tx];

        // 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 the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
```

6.3 Source Code Walkthrough

The source code contains two functions:

- **Mul()**, a host function serving as a wrapper to **Muld()**;
- **Muld()**, a kernel that executes the matrix multiplication on the device.

6.3.1 Mul()

Mul() takes as input:

- Two pointers to host memory that point to the elements of *A* and *B*,
- The height and width of *A* and the width of *B*,
- A pointer to host memory that points where *C* should be written.

Mul() performs the following operations:

- It allocates enough global memory to store *A*, *B*, and *C* using **cudaMalloc()**;
- It copies *A* and *B* from host memory to global memory using **cudaMemcpy()**;
- It calls **Muld()** to compute *C* on the device;
- It copies *C* from global memory to host memory using **cudaMemcpy()**;
- It frees the global memory allocated for *A*, *B*, and *C* using **cudaFree()**.

6.3.2 Muld()

Muld() has the same input as **Mul()**, except that pointers point to device memory instead of host memory.

For each block, **Muld()**iterates through all the sub-matrices of *A* and *B* required to compute C_{sub}. At each iteration:

- It loads one sub-matrix of *A* and one sub-matrix of *B* from global memory to shared memory;
- It synchronizes to make sure that both sub-matrices are fully loaded by all the threads within the block;
- It computes the product of the two sub-matrices and adds it to the product obtained during the previous iteration;

- It synchronizes again to make sure that the product of the two sub-matrices is done before starting the next iteration.

Once all sub-matrices have been handled, C_{sub} is fully computed and **`Muld()`** writes it to global memory.

`Muld()` is written to maximize memory performance according to Section 5.1.2.1 and 5.1.2.4.

Indeed, assuming that **`wA`** and **`wB`** are multiples of 16 as suggested in Section 5.1.2.1, global memory coalescing is ensured because **`a`**, **`b`**, and **`c`** are all multiples of **`BLOCK_SIZE`**, which is equal to 16.

There is also no shared memory bank conflict since for each half-warp, **`ty`** and **`k`** are the same for all threads and **`tx`** varies from **`0`** to **`15`**, so each thread accesses a different bank for the memory accesses **`As[ty][tx]`**, **`Bs[ty][tx]`**, and **`Bs[k][tx]`** and the same bank for the memory access **`As[ty][k]`**.

Appendix A. Technical Specifications

All devices with a 1.x compute capability follow the technical specifications detailed in this appendix.

Atomic functions are only available for devices of compute capability 1.1 (see Section 4.4.6).

The number of multiprocessors for the GeForce 8800 Series, Quadro FX 5600/4600, and Tesla solutions are given in the following table:

	Number of multiprocessors
GeForce 8800 Ultra	16
GeForce 8800 GTX	16
GeForce 8800 GTS	12
GeForce 8600 GTS	4
GeForce 8600 GT	2
GeForce 8500 GT	2
Quadro FX 5600	16
Quadro FX 4600	12
Tesla C870	16
Tesla D870	2x16
Tesla S870	4x16

The clock frequency and total amount of device memory can be queried using the runtime (Sections 4.5.2.2 and 4.5.3.2).

A.1 General Specifications

- The maximum number of threads per block is 512;
- The maximum sizes of the x-, y-, and z-dimension of a thread blocks are 512, 512, and 64, respectively;
- The maximum size of each dimension of a grid of thread blocks is 65535;
- The warp size is 32 threads;

- The number of registers per multiprocessor is 8192;
- The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks (see Section 5.1.2.4);
- The amount of constant memory available is 64 KB with a cache working set of 8 KB per multiprocessor;
- The cache working set for one-dimensional textures is 8 KB per multiprocessor;
- The maximum number of blocks that can run concurrently on a multiprocessor is 8;
- The maximum number of warps that can run concurrently on a multiprocessor is 24;
- The maximum number of threads that can run concurrently on a multiprocessor is 768;
- For a texture reference bound to a one-dimensional CUDA array, the maximum width is 2^{13};
- For a texture reference bound to a two-dimensional CUDA array, the maximum width is 2^{16} and the maximum height is 2^{15};
- For a texture reference bound to linear memory, the maximum width is 2^{27};
- The limit on kernel size is 2 million of native instructions;
- Each multiprocessor is composed of eight processors, so that a multiprocessor is able to process the 32 threads of a warp in four clock cycles.

A.2 Floating-Point Standard

Compute devices follow the IEEE-754 standard for single-precision binary floating-point arithmetic with the following deviations:

- Addition and multiplication are often combined into a single multiply-add instruction (FMAD);
- 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;
- There is no dynamically configurable rounding mode;
- 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;
- There is no mechanism for detecting that a floating-point exception has occurred and floating-point exceptions are always masked, but when an exception occurs the masked response is standard compliant;
- Signaling NaNs are not supported.
- The result of an operation involving one or more input NaNs is not one of the input NaNs, but a canonical NaN of bit pattern 0x7fffffff. Note that in

accordance to the IEEE-754R standard, if one of the input parameters to **`min()`** or **`max()`** 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 behaves.

Appendix B. Mathematical Functions

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

B.1 Common Runtime Component

Table B-1 below lists all the mathematical standard library functions supported by the CUDA runtime library. 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 no supply the function. They are generated from extensive but not exhaustive tests, so they are not guaranteed bounds.

Addition and multiplication are IEEE-compliant, so have a maximum error of 0.5 ulp. They are however often combined into a single multiply-add instruction (FMAD), which truncates the intermediate result of the multiplication.

The recommended way to round a floating-point operand to an integer, with the result being a floating-point number is **`rintf()`**, not **`roundf()`**. The reason is that **`roundf()`** maps to an 8-instruction sequence, whereas **`rintf()`** maps to a single instruction.

`truncf()`, **`ceilf()`**, and **`floorf()`** each map to a single instruction as well.

Table B-1. Mathematical Standard Library Functions with Maximum ULP Error

Function	Maximum ulp error
`x/y`	2 (full range)
`1/x`	1 (full range)
`1/sqrtf(x)` `rsqrtf(x)`	2 (full range)
`sqrtf(x)`	3 (full range)
`cbrtf(x)`	1 (full range)
`hypotf(x)`	3 (full range)
`expf(x)`	2 (full range)

Function	Maximum ulp error
`exp2f(x)`	2 (full range)
`exp10f(x)`	2 (full range)
`expm1f(x)`	4 (full range)
`logfx)`	1 (full range)
`log2f(x)`	3 (full range)
`log10f(x)`	3 (full range)
`log1pf(x)`	2 (full range)
`sinf(x)`	2 (inside interval -48039 ... +48039; larger outside) Total loss of accuracy occurs for $\|x\| > 10^7$.
`cosf(x)`	2 (inside interval -48039 ... +48039; larger outside) Total loss of accuracy occurs for $\|x\| > 10^7$.
`tanf(x)`	4 (inside interval -48039 ... +48039; larger outside) Total loss of accuracy occurs for $\|x\| > 10^7$.
`sincosf(x,sptr,cptr)`	2 (inside interval -48039 ... +48039; larger outside) Total loss of accuracy occurs for $\|x\| > 10^7$.
`asinf(x)`	4 (full range)
`acosf(x)`	3 (full range)
`atanf(x)`	2 (full range)
`atan2f(y,x)`	3 (full range)
`sinhf(x)`	3 (full range)
`coshf(x)`	2 (full range)
`tanhf(x)`	2 (full range)
`asinhf(x)`	3 (full range)
`acoshf(x)`	4 (full range)
`atanhf(x)`	3 (full range)
`powf(x,y)`	For x in [0.75, 1.27], the maximum ulp error is $9 + \text{trunc}(1.5 * \text{abs}(\log(\text{abs}(x^y))))$, and 16 otherwise.
`erff(x)`	4 (full range)
`erfcf(x)`	8 (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)

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

B.2 Device Runtime Component

For some of the functions of Table B-1, a less accurate, but faster version exists with the same name prefixed with __ (such as **__sinf(x)**). These instrinsic functions are listed in Table B-2. Their error bounds are GPU-specific.

__fadd_rz(x,y) computes the sum of floating-point parameters **x** and **y** using the round-towards-zero rounding mode.

__fmul_rz(x,y) computes the product of floating-point parameters **x** and **y** using the round-towards-zero rounding mode.

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 B-1. 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.

__[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.

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

Table B-2. Intrinsic Functions Supported by the CUDA Runtime Library with Respective Error Bounds for Devices of Compute Capability 1.x

Function	Error bounds
__fadd_rz(x,y)	IEEE-compliant.
__fmul_rz(x,y)	IEEE-compliant.
__fdividef(x,y)	For **y** in $[2^{-126}, 2^{126}]$, the maximum ulp error is 2.
__expf(x)	The maximum ulp error is **2 + floor(abs(1.16 * x))**.
__exp10f(x)	The maximum ulp error is **2 + floor(abs(2.95 * x))**.
__logf(x)	For **x** in [0.5, 2], the maximum absolute error is $2^{-21.41}$, otherwise, the maximum ulp error is 3.
__log2f(x)	For **x** in [0.5, 2], the maximum absolute error is 2^{-22}, otherwise, the maximum ulp error is 2.
__log10f(x)	For **x** in [0.5, 2], the maximum absolute error is 2^{-24}, otherwise, the maximum ulp error is 3.
__sinf(x)	For **x** in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
__cosf(x)	For **x** in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.19}$, and

	larger otherwise.
__sincosf(x,sptr,cptr)	Same as **sinf(x)** and **cosf(x)**.
__tanf(x)	Derived from its implementation as **__sinf(x) * (1 / __cosf(x))**.
__powf(x, y)	Derived from its implementation as **exp2f(y * __log2f(x))**.
__mul24(x,y) **__umul24(x,y)**	N/A
__mulhi(x,y) **__umulhi(x,y)**	N/A
__int_as_float(x)	N/A
__float_as_int(x)	N/A
__saturate(x)	N/A

Appendix C. Atomic Functions

Atomic functions can only be used in device functions.

C.1 Arithmetic Functions

C.1.1 atomicAdd()

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

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

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

C.1.3 atomicExch()

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

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

C.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 memory, computes the minimum of **old** and **val**, and stores the result back to global memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

C.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 memory, computes the maximum of **old** and **val**, and stores the result back to global memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

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

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

C.1.8 atomicCAS()

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

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

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

C.2 Bitwise Functions

C.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 memory, computes **(old & val)**, and stores the result back to global memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

C.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 memory, computes **(old | val)**, and stores the result back to global memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

C.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 memory, computes **(old ^ val)**, and stores the result back to global memory at the same address. These three operations are performed in one atomic transaction. The function returns **old**.

Appendix D. Runtime API Reference

There are two levels for the runtime API.

The low-level API (**cuda_runtime_api.h**) is a C-style interface that does not require compiling with **nvcc**.

The high-level API (**cuda_runtime.h**) is a C++-style interface built on top of the low-level API. It wraps some of the low level API routines, using overloading, references and default arguments. These wrappers can be used from C++ code and can be compiled with any C++ compiler. The high-level API also has some CUDA-specific wrappers that wrap low-level routines that deal with symbols, textures, and device functions. These wrappers require the use of **nvcc** because they depend on code being generated by the compiler (see Section 4.2.5). For example, the execution configuration syntax described in Section 4.2.3 to invoke kernels is only available in source code compiled with **nvcc**.

D.1 Device Management

D.1.1 cudaGetDeviceCount()

```
cudaError_t cudaGetDeviceCount(int* count);
```

returns in ***count** the number of devices with compute capability greater or equal to 1.0 that are available for execution. If there is no such device, **cudaGetDeviceCount()** returns 1 and device 0 only supports device emulation mode and is of compute capability less than 1.0.

D.1.2 cudaGetDeviceProperties()

```
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp* prop,
                                    int dev);
```

returns in ***prop** the properties of device **dev**. The **cudaDeviceProp** structure is defined as:

```
struct cudaDeviceProp {
  char   name[256];
  size_t totalGlobalMem;
```

```
  size_t sharedMemPerBlock;
  int    regsPerBlock;
  int    warpSize;
  size_t memPitch;
  int    maxThreadsPerBlock;
  int    maxThreadsDim[3];
  int    maxGridSize[3];
  size_t totalConstMem;
  int    major;
  int    minor;
  int    clockRate;
  size_t textureAlignment;
};
```

where:

- **name** is an ASCII string identifying the device;
- **totalGlobalMem** is the total amount of global memory available on the device in bytes;
- **sharedMemPerBlock** is the total amount of shared memory available per block in bytes;
- **regsPerBlock** is the total number of registers available per block;
- **warpSize** is the warp size;
- **memPitch** is the maximum pitch allowed by the memory copy functions of Section D.3 that involve memory regions allocated through **cudaMallocPitch()** (Section D.3.2);
- **maxThreadsPerBlock** is the maximum number of threads per block;
- **maxThreadsDim[3]** is the maximum sizes of each dimension of a block;
- **maxGridSize[3]** is the maximum sizes of each dimension of a grid;
- **totalConstMem** is the total amount of constant memory available on the device in bytes;
- **major** and **minor** are the major and minor revision numbers;
- **clockRate** is the clock frequency in kilohertz;
- **textureAlignment** is the alignment requirement mentioned in Section 4.3.4.3; texture base addresses that are aligned to **textureAlignment** bytes do not need an offset applied to texture fetches.

D.1.3 cudaChooseDevice()

```
cudaError_t cudaChooseDevice(int* dev,
                             const struct cudaDeviceProp* prop);
```

returns in ***dev** the device which properties best match ***prop**.

D.1.4 cudaSetDevice()

```
cudaError_t cudaSetDevice(int dev);
```

records **dev** as the device on which the active host thread executes the device code.

D.1.5 cudaGetDevice()

```
cudaError_t cudaGetDevice(int* dev);
```

returns in ***dev** the device on which the active host thread executes the device code.

D.2 Thread Management

D.2.1 cudaThreadSynchronize()

```
cudaError_t cudaThreadSynchronize(void);
```

blocks until the device has completed all preceding requested tasks. **cudaThreadSynchronize()** returns an error if one of the preceding tasks failed.

D.2.2 cudaThreadExit()

```
cudaError_t cudaThreadExit(void);
```

explicitly cleans up all runtime-related resources associated with the calling host thread. Any subsequent API call reinitializes the runtime. **cudaThreadExit()** is implicitly called on host thread exit.

D.3 Memory Management

D.3.1 cudaMalloc()

```
cudaError_t cudaMalloc(void** devPtr, size_t count);
```

allocates **count** bytes of linear memory on the device and returns in ***devPtr** a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. **cudaMalloc()** returns **cudaErrorMemoryAllocation** in case of failure.

D.3.2 cudaMallocPitch()

```
cudaError_t cudaMallocPitch(void** devPtr,
                            size_t* pitch,
                            size_t widthInBytes,
                            size_t height);
```

allocates at least **widthInBytes*height** bytes of linear memory on the device and returns in ***devPtr** a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row (see Section 5.1.2.1). The pitch returned in ***pitch** by **cudaMallocPitch()** is the width in bytes of the allocation. The intended usage of pitch is as a separate parameter of the allocation, used to compute addresses

within the 2D array. Given the row and column of an array element of type **T**, the address is computed as

```
T* pElement = (T*)((char*)BaseAddress + Row * pitch) + Column;
```

For allocations of 2D arrays, it is recommended that developers consider performing pitch allocations using **cudaMallocPitch()**. Due to pitch alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

D.3.3 cudaFree()

```
cudaError_t cudaFree(void* devPtr);
```

frees the memory space pointed to by **devPtr**, which must have been returned by a previous call to **cudaMalloc()** or **cudaMallocPitch()**. Otherwise, or if **cudaFree(devPtr)** has already been called before, an error is returned. If **devPtr** is 0, no operation is performed. **cudaFree()** returns **cudaErrorInvalidDevicePointer** in case of failure.

D.3.4 cudaMallocArray()

```
cudaError_t cudaMallocArray(struct cudaArray** array,
                            const struct cudaChannelFormatDesc* desc,
                               size_t width, size_t height);
```

allocates a CUDA array according to the **cudaChannelFormatDesc** structure **desc** and returns a handle to the new CUDA array in ***array**. **cudaChannelFormatDesc** is described in Section 4.3.4.

D.3.5 cudaFreeArray()

```
cudaError_t cudaFreeArray(struct cudaArray* array);
```

frees the CUDA array **array**.

D.3.6 cudaMallocHost()

```
cudaError_t cudaMallocHost(void** hostPtr, size_t size);
```

allocates **size** bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as **cudaMemcpy*()**. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as **malloc()**. Allocating excessive amounts of memory with **cudaMallocHost()** may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

D.3.7 cudaFreeHost()

```
cudaError_t cudaFreeHost(void* hostPtr);
```

frees the memory space pointed to by **hostPtr**, which must have been returned by a previous call to **cudaMallocHost()**.

D.3.8 cudaMemset()

```
cudaError_t cudaMemset(void* devPtr, int value, size_t count);
```

fills the first **count** bytes of the memory area pointed to by **devPtr** with the constant byte value **value**.

D.3.9 cudaMemset2D()

```
cudaError_t cudaMemset2D(void* dstPtr, size_t pitch,
                         int value, size_t width, size_t height);
```

sets to the specified value **value** a matrix (**height** rows of **width** bytes each) pointed to by **dstPtr**. **pitch** is the width in memory in bytes of the 2D array pointed to by **dstPtr**, including any padding added to the end of each row (see Section D.3.2).

D.3.10 cudaMemcpy()

```
cudaError_t cudaMemcpy(void* dst, const void* src,
                       size_t count,
                       enum cudaMemcpyKind kind);
```

copies **count** bytes from the memory area pointed to by **src** to the memory area pointed to by **dst**, where **kind** is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy. The memory areas may not overlap. Calling **cudaMemcpy()** with **dst** and **src** pointers that do not match the direction of the copy results in an undefined behavior.

D.3.11 cudaMemcpy2D()

```
cudaError_t cudaMemcpy2D(void* dst, size_t dpitch,
                         const void* src, size_t spitch,
                         size_t width, size_t height,
                         enum cudaMemcpyKind kind);
```

copies a matrix (**height** rows of **width** bytes each) from the memory area pointed to by **src** to the memory area pointed to by **dst**, where **kind** is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy. **dpitch** and **spitch** are the widths in memory in bytes of the 2D arrays pointed to by **dst** and **src**, including any padding added to the end of each row (see Section D.3.2). The memory areas may not overlap. Calling **cudaMemcpy2D()** with **dst** and **src** pointers that do not match the direction of the copy results in an undefined behavior.

D.3.12 cudaMemcpyToArray()

```
cudaError_t cudaMemcpyToArray(struct cudaArray* dstArray,
                              size_t dstX, size_t dstY,
                              const void* src, size_t count,
                              enum cudaMemcpyKind kind);
```

copies **count** bytes from the memory area pointed to by **src** to the CUDA array **dstArray** starting at the upper left corner (**dstX**, **dstY**), where kind is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy.

D.3.13 cudaMemcpy2DToArray()

```
cudaError_t cudaMemcpy2DToArray(struct cudaArray* dstArray,
                                size_t dstX, size_t dstY,
                                const void* src, size_t spitch,
                                size_t width, size_t height,
                                enum cudaMemcpyKind kind);
```

copies a matrix (**height** rows of **width** bytes each) from the memory area pointed to by **src** to the CUDA array **dstArray** starting at the upper left corner (**dstX**, **dstY**), where kind is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy. **spitch** is the width in memory in bytes of the 2D array pointed to by **src**, including any padding added to the end of each row (see Section D.3.2).

D.3.14 cudaMemcpyFromArray()

```
cudaError_t cudaMemcpyFromArray(void* dst,
                                const struct cudaArray* srcArray,
                                size_t srcX, size_t srcY,
                                size_t count,
                                enum cudaMemcpyKind kind);
```

copies **count** bytes from the CUDA array **srcArray** starting at the upper left corner (**srcX**, **srcY**) to the memory area pointed to by **dst**, where kind is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy.

D.3.15 cudaMemcpy2DFromArray()

```
cudaError_t cudaMemcpy2DFromArray(void* dst, size_t dpitch,
                                  const struct cudaArray* srcArray,
                                  size_t srcX, size_t srcY,
                                  size_t width, size_t height,
                                  enum cudaMemcpyKind kind);
```

copies a matrix (**height** rows of **width** bytes each) from the CUDA array **srcArray** starting at the upper left corner (**srcX**, **srcY**) to the memory area

pointed to by **dst**, where kind is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy. **dpitch** is the width in memory in bytes of the 2D array pointed to by **dst**, including any padding added to the end of each row (see Section D.3.2).

D.3.16 cudaMemcpyArrayToArray()

```
cudaError_t cudaMemcpyArrayToArray(struct cudaArray* dstArray,
                                   size_t dstX, size_t dstY,
                                 const struct cudaArray* srcArray,
                                   size_t srcX, size_t srcY,
                                   size_t count,
                                   enum cudaMemcpyKind kind);
```

copies **count** bytes from the CUDA array **srcArray** starting at the upper left corner (**srcX**, **srcY**) to the CUDA array **dstArray** starting at the upper left corner (**dstX**, **dstY**), where kind is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy.

D.3.17 cudaMemcpy2DArrayToArray()

```
cudaError_t cudaMemcpy2DArrayToArray(struct cudaArray* dstArray,
                                     size_t dstX, size_t dstY,
                                   const struct cudaArray* srcArray,
                                     size_t srcX, size_t srcY,
                                     size_t width, size_t height,
                                     enum cudaMemcpyKind kind);
```

copies a matrix (**height** rows of **width** bytes each) from the CUDA array **srcArray** starting at the upper left corner (**srcX**, **srcY**) to the CUDA array **dstArray** starting at the upper left corner (**dstX**, **dstY**), where kind is one of **cudaMemcpyHostToHost**, **cudaMemcpyHostToDevice**, **cudaMemcpyDeviceToHost**, or **cudaMemcpyDeviceToDevice**, and specifies the direction of the copy.

D.3.18 cudaMemcpyToSymbol()

```
template<class T>
cudaError_t cudaMemcpyToSymbol(const T& symbol, const void* src,
                               size_t count, size_t offset = 0,
                  enum cudaMemcpyKind kind = cudaMemcpyHostToDevice);
```

copies **count** bytes from the memory area pointed to by **src** to the memory area pointed to by **offset** bytes from the start of symbol **symbol**. The memory areas may not overlap. **symbol** can either be a variable that resides in global memory space, or it can be a character string, naming a variable that resides in global memory space. **kind** can be either **cudaMemcpyHostToDevice** or **cudaMemcpyDeviceToDevice**.

D.3.19 cudaMemcpyFromSymbol()

```
template<class T>
cudaError_t cudaMemcpyFromSymbol(void *dst, const T& symbol,
                                 size_t count, size_t offset = 0,
                enum cudaMemcpyKind kind = cudaMemcpyDeviceToHost);
```

copies **count** bytes from the memory area pointed to by **offset** bytes from the start of symbol **symbol** to the memory area pointed to by **dst**. The memory areas may not overlap. **symbol** can either be a variable that resides in global memory space, or it can be a character string, naming a variable that resides in global memory space. **kind** can be either **cudaMemcpyDeviceToHost** or **cudaMemcpyDeviceToDevice**.

D.3.20 cudaGetSymbolAddress()

```
template<class T>
cudaError_t cudaGetSymbolAddress(void** devPtr, const T& symbol);
```

returns in ***devPtr** the address of symbol **symbol** on the device. **symbol** can either be a variable that resides in global memory space, or it can be a character string, naming a variable that resides in global memory space. If **symbol** cannot be found, or if **symbol** is not declared in global memory space, ***devPtr** is unchanged and an error is returned. **cudaGetSymbolAddress()** returns **cudaErrorInvalidSymbol** in case of failure.

D.3.21 cudaGetSymbolSize()

```
template<class T>
cudaError_t cudaGetSymbolSize(size_t* size, const T& symbol);
```

returns in ***size** the size of symbol **symbol**. **symbol** can either be a variable that resides in global or constant memory space, or it can be a character string, naming a variable that resides in global or constant memory space. If **symbol** cannot be found, or if **symbol** is not declared in global or constant memory space, ***size** is unchanged and an error is returned. **cudaGetSymbolSize()** returns **cudaErrorInvalidSymbol** in case of failure.

D.4 Texture Reference Management

D.4.1 Low-Level API

D.4.1.1 cudaCreateChannelDesc()

```
struct cudaChannelFormatDesc
cudaCreateChannelDesc(int x, int y, int z, int w,
                      enum cudaChannelFormatKind f);
```

returns a channel descriptor with format **f** and number of bits of each component **x**, **y**, **z**, and **w**. **cudaChannelFormatDesc** is described in Section 4.3.4.

D.4.1.2 cudaGetChannelDesc()

```
cudaError_t cudaGetChannelDesc(struct cudaChannelFormatDesc* desc,
                               const struct cudaArray* array);
```

returns in ***desc** the channel descriptor of the CUDA array **array**.

D.4.1.3 cudaGetTextureReference()

```
cudaError_t cudaGetTextureReference(
                                  struct textureReference** texRef,
                                  const char* symbol);
```

returns in ***texRef** the structure associated to the texture reference defined by symbol **symbol**.

D.4.1.4 cudaBindTexture()

```
cudaError_t cudaBindTexture(size_t* offset,
                            const struct textureReference* texRef,
                            const void* devPtr,
                          const struct cudaChannelFormatDesc* desc,
                            size_t size = UINT_MAX);
```

binds **size** bytes of the memory area pointed to by **devPtr** to the texture reference **texRef**. **desc** describes how the memory is interpreted when fetching values from the texture. Any memory previously bound to **texRef** is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, **cudaBindTexture()** returns in ***offset** a byte offset that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the **tex1Dfetch()** function. If the device memory pointer was returned from **cudaMalloc()**, the offset is guaranteed to be 0 and NULL may be passed as the offset parameter.

D.4.1.5 cudaBindTextureToArray()

```
cudaError_t cudaBindTextureToArray(
                            const struct textureReference* texRef,
                            const struct cudaArray* array,
                          const struct cudaChannelFormatDesc* desc);
```

binds the CUDA array **array** to the texture reference **texRef**. **desc** describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to **texRef** is unbound.

D.4.1.6 cudaUnbindTexture()

```
cudaError_t cudaUnbindTexture(
                           const struct textureReference* texRef);
```

unbinds the texture bound to texture reference **texRef**.

D.4.1.7 cudaGetTextureAlignmentOffset()

```
cudaError_t cudaGetTextureAlignmentOffset(size_t* offset,
                           const struct textureReference* texRef);
```

returns in ***offset** the offset that was returned when texture reference **texRef** was bound.

D.4.2 High-Level API

D.4.2.1 cudaCreateChannelDesc()

```
template<class T>
struct cudaChannelFormatDesc cudaCreateChannelDesc<T>();
```

returns a channel descriptor with format matching type **T**. **T** can be any types of Section 4.3.1.1. 3-component types default to a 4-component format.

D.4.2.2 cudaBindTexture()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTexture(size_t* offset,
                const struct texture<T, dim, readMode>& texRef,
                const void* devPtr,
                const struct cudaChannelFormatDesc& desc,
                size_t size = UINT_MAX);
```

binds **size** bytes of the memory area pointed to by **devPtr** to texture reference **texRef**. **desc** describes how the memory is interpreted when fetching values from the texture. The offset parameter is an optional byte offset as with the low-level **cudaBindTexture()** function described in Section D.4.1.4. Any memory previously bound to **texRef** is unbound.

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTexture(size_t* offset,
                const struct texture<T, dim, readMode>& texRef,
                const void* devPtr,
                size_t size = UINT_MAX);
```

binds **size** bytes of the memory area pointed to by **devPtr** to texture reference **texRef**. The channel descriptor is inherited from the texture reference type. The offset parameter is an optional byte offset as with the low-level **cudaBindTexture()** function described in Section D.4.1.4. Any memory previously bound to **texRef** is unbound.

D.4.2.3 cudaBindTextureToArray()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTextureToArray(
                  const struct texture<T, dim, readMode>& texRef,
                  const struct cudaArray* cuArray,
                  const struct cudaChannelFormatDesc& desc);
```

binds the CUDA array **array** to texture reference **texRef**. **desc** describes how the memory is interpreted when fetching values from the texture. Any CUDA array previously bound to **texRef** is unbound.

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaBindTextureToArray(
                  const struct texture<T, dim, readMode>& texRef,
                  const struct cudaArray* cuArray);
```

binds the CUDA array **array** to texture reference **texRef**. The channel descriptor is inherited from the CUDA array. Any CUDA array previously bound to **texRef** is unbound.

D.4.2.4 cudaUnbindTexture()

```
template<class T, int dim, enum cudaTextureReadMode readMode>
static __inline__ __host__ cudaError_t
cudaUnbindTexture(const struct texture<T, dim, readMode>& texRef);
```

unbinds the texture bound to texture reference **texRef**.

D.5 Execution Control

D.5.1 cudaConfigureCall()

```
cudaError_t cudaConfigureCall(dim3 gridDim, dim3 blockDim,
                              size_t sharedMem = 0,
                              int tokens = 0);
```

specifies the grid and block dimensions for the device call to be executed similar to the execution configuration syntax described in Section 4.2.3. **cudaConfigureCall()** is stack based. Each call pushes data on top of an execution stack. This data contains the dimension for the grid and thread blocks, together with any arguments for the call.

D.5.2 cudaLaunch()

```
template<class T> cudaError_t cudaLaunch(T entry);
```

launches the function **entry** on the device. **entry** can either be a function that executes on the device, or it can be a character string, naming a function that executes on the device. **entry** must be declared as a **__global__** function. **cudaLaunch()** must be preceded by a call to **cudaConfigureCall()** since it pops the data that was pushed by **cudaConfigureCall()** from the execution stack.

D.5.3 cudaSetupArgument()

```
cudaError_t cudaSetupArgument(void* arg,
                              size_t count, size_t offset);
template<class T> cudaError_t cudaSetupArgument(T arg,
                                                size_t offset);
```

pushes **count** bytes of the argument pointed to by **arg** at **offset** bytes from the start of the parameter passing area, which starts at offset 0. The arguments are stored in the top of the execution stack. **cudaSetupArgument()** must be preceded by a call to **cudaConfigureCall()**.

D.6 OpenGL Interoperability

D.6.1 cudaGLRegisterBufferObject()

```
cudaError_t cudaGLRegisterBufferObject(GLuint bufferObj);
```

registers the buffer object of ID **bufferObj** for access by CUDA. This function must be called before CUDA can map the buffer object. While it is registered, the buffer object cannot be used by any OpenGL commands except as a data source for OpenGL drawing commands.

D.6.2 cudaGLMapBufferObject()

```
cudaError_t cudaGLMapBufferObject(void** devPtr,
                                  GLuint bufferObj);
```

maps the buffer object of ID **bufferObj** into the address space of CUDA and returns in ***devPtr** the base pointer of the resulting mapping.

D.6.3 cudaGLUnmapBufferObject()

```
cudaError_t cudaGLUnmapBufferObject(GLuint bufferObj);
```

unmaps the buffer object of ID **bufferObj** for access by CUDA.

D.6.4 cudaGLUnregisterBufferObject()

```
cudaError_t cudaGLUnregisterBufferObject(GLuint bufferObj);
```

unregisters the buffer object of ID **bufferObj** for access by CUDA.

D.7 Direct3D Interoperability

D.7.1 cudaD3D9Begin()

```
cudaError_t cudaD3D9Begin(IDirect3DDevice9* device);
```

initializes interoperability with the Direct3D device **device**. This function must be called before CUDA can map any objects from **device**. The application can then map vertex buffers owned by the Direct3D device until **cuD3D9End()** is called.

D.7.2 cudaD3D9End()

```
cudaError_t cudaD3D9End(void);
```

concludes interoperability with the Direct3D device previously specified to **cuD3D9Begin()**.

D.7.3 cudaD3D9RegisterVertexBuffer()

```
cudaError_t
cudaD3D9RegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

unregisters the Direct3D vertex buffer **VB** for access by CUDA.

D.7.4 cudaD3D9MapVertexBuffer()

```
cudaError_t cudaD3D9MapVertexBuffer(void** devPtr,
                                    unsigned int* size,
                                    IDirect3DVertexBuffer9* VB);
```

maps the Direct3D vertex buffer **VB** into the address space of the current CUDA context and returns in ***devPtr** and ***size** the base pointer and size of the resulting mapping.

D.7.5 cudaD3D9UnmapVertexBuffer()

```
cudaError_t cudaD3D9UnmapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

unmaps the vertex buffer **VB** for access by CUDA.

D.7.6 cudaD3D9UnregisterVertexBuffer()

```
cudaError_t cudaD3D9UnregisterVertexBuffer(IDirect3DVertexBuffer9*
VB);
```

unmaps the vertex buffer **VB** for access by CUDA.

D.8 Error Handling

D.8.1 cudaGetLastError()

```
cudaError_t cudaGetLastError(void);
```

returns the last error that was returned from any of the runtime calls in the same host thread and resets it to **cudaSuccess**.

D.8.2 cudaGetErrorString()

```
const char* cudaGetErrorString(cudaError_t error);
```

returns a message string from an error code.

Appendix E. Driver API Reference

E.1 Initialization

E.1.1 cuInit()

```
CUresult cuInit(unsigned int Flags);
```

initializes the driver API and must be called before any other function from the driver API. Currently, the **Flags** parameters must be 0. If **cuInit()** has not been called, any function from the driver API will return **CUDA_ERROR_NOT_INITIALIZED**.

E.2 Device Management

E.2.1 cuDeviceGetCount()

```
CUresult cuDeviceGetCount(int* count);
```

returns in ***count** the number of devices with compute capability greater or equal to 1.0 that are available for execution. If there is no such device, **cuDeviceGetCount()** returns 1 and device 0 only supports device emulation mode and is of compute capability less than 1.0.

E.2.2 cuDeviceGet()

```
CUresult cuDeviceGet(CUdevice* dev, int ordinal);
```

returns in ***dev** a device handle given an ordinal in the range **[0, cuDeviceGetCount()-1]**.

E.2.3 cuDeviceGetName()

```
CUresult cuDeviceGetName(char* name, int len, CUdevice dev);
```

returns an ASCII string identifying the device **dev** in the NULL-terminated string pointed to by **name**. **len** specifies the maximum length of the string that may be returned.

E.2.4 cuDeviceTotalMem()

```
CUresult cuDeviceTotalMem(unsigned int* bytes, CUdevice dev);
```

returns in ***bytes** the total amount of memory available on the device **dev** in bytes.

E.2.5 cuDeviceComputeCapability()

```
CUresult cuDeviceComputeCapability(int* major, int* minor,
                                   CUdevice dev);
```

returns in ***major** and ***minor** the major and minor revision numbers of device **dev**.

E.2.6 cuDeviceGetProperties()

```
cudaError_t cudaGetDeviceProperties(CUdevprop* prop,
                                    CUdevice dev);
```

returns in ***prop** the properties of device **dev**. The **CUdevprop** structure is defined as:

```
typedef struct CUdevprop_st {
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    int sharedMemPerBlock;
    int totalConstantMemory;
    int SIMDWidth;
    int memPitch;
    int regsPerBlock;
    int clockRate;
    int textureAlign;
} CUdevprop;
```

where:

- **maxThreadsPerBlock** is the maximum number of threads per block;
- **maxThreadsDim[3]** is the maximum sizes of each dimension of a block;
- **maxGridSize[3]** is the maximum sizes of each dimension of a grid;
- **sharedMemPerBlock** is the total amount of shared memory available per block in bytes;
- **totalConstantMemory** is the total amount of constant memory available on the device in bytes;
- **SIMDWidth** is the warp size;
- **memPitch** is the maximum pitch allowed by the memory copy functions of Section E.6 that involve memory regions allocated through **cuMemAllocPitch()** (Section E.6.3);

- **regsPerBlock** is the total number of registers available per block;
- **clockRate** is the clock frequency in kilohertz;
- **textureAlign** is the alignment requirement mentioned in Section 4.3.4.3; texture base addresses that are aligned to **textureAlign** bytes do not need an offset applied to texture fetches.

E.3 Context Management

E.3.1 cuCtxCreate()

```
CUresult cuCtxCreate(CUcontext* pCtx, unsigned int Flags, CUdevice dev);
```

creates a new context for a device and associates it with the calling thread. Currently the **Flags** parameter must be 0. The context is created with a usage count of 1 and the caller of **cuCtxCreate()** must call **cuCtxDetach()** when done using the context. This function fails if a context is already current to the thread.

E.3.2 cuCtxAttach()

```
CUresult cuCtxAttach(CUcontext* pCtx, unsigned int Flags);
```

increments the usage count of the context and passes back a context handle in ***pCtx** that must be passed to **cuCtxDetach()** when the application is done with the context. **cuCtxAttach()** fails if there is no context current to the thread.

Currently, the **Flags** parameter must be 0.

E.3.3 cuCtxDetach()

```
CUresult cuCtxDetach(CUcontext ctx);
```

decrements the usage count of the context, and destroys the context if the usage count goes to 0. The context must be a handle that was passed back by **cuCtxCreate()** or **cuCtxAttach()**, and must be current to the calling thread.

E.3.4 cuCtxGetDevice()

```
CUresult cuCtxGetDevice(CUdevice* device);
```

returns in ***device** the ordinal of the current context's device.

E.3.5 cuCtxSynchronize()

```
CUresult cuCtxSynchronize(void);
```

blocks until the device has completed all preceding requested tasks. **cuCtxSynchronize()** returns an error if one of the preceding tasks failed.

E.4 Module Management

E.4.1 cuModuleLoad()

```
CUresult cuModuleLoad(CUmodule* mod, const char* filename);
```

takes a file name **filename** and loads the corresponding module **mod** into the current context. The CUDA driver API does not attempt to lazily allocate the resources needed by a module; if the memory for functions and data (constant and global) needed by the module cannot be allocated, **cuModuleLoad()** fails. The file should be a *cubin* file as output by **nvcc** (see Section 4.2.5).

E.4.2 cuModuleLoadData()

```
CUresult cuModuleLoadData(CUmodule* mod, const void* image);
```

takes a pointer **image** and loads the corresponding module **mod** into the current context. The pointer may be obtained by mapping a *cubin* file, passing a *cubin* file as a text string, or incorporating a *cubin* object into the executable resources and using operation system calls such as Windows' **FindResource()** to obtain the pointer.

E.4.3 cuModuleLoadFatBinary()

```
CUresult cuModuleLoadFatBinary(CUmodule* mod, const void* fatBin);
```

takes a pointer **fatBin** and loads the corresponding module **mod** into the current context. The pointer represents a *fat binary* object, which is a collection of different *cubin* files, all representing the same device code but compiled and optimized for different architectures. There is currently no documented API for constructing and using fat binary objects by programmers, and therefore this function is an internal function in this version of CUDA. More information can be found in the **nvcc** document.

E.4.4 cuModuleUnload()

```
CUresult cuModuleUnload(CUmodule mod);
```

unloads a module **mod** from the current context.

E.4.5 cuModuleGetFunction()

```
CUresult cuModuleGetFunction(CUfunction* func,
                             CUmodule mod, const char* funcname);
```

returns in ***func** the handle of the function of name **funcname** located in module **mod**. If no function of that name exists, **cuModuleGetFunction()** returns **CUDA_ERROR_NOT_FOUND**.

E.4.6 cuModuleGetGlobal()

```
CUresult cuModuleGetGlobal(CUdeviceptr* devPtr,
                           unsigned int* bytes,
                           CUmodule mod, const char* globalname);
```

returns in ***devPtr** and ***bytes** the base pointer and size of the global of name **globalname** located in module **mod**. If no variable of that name exists, **cuModuleGetGlobal()** returns **CUDA_ERROR_NOT_FOUND**. Both parameters **devPtr** and **bytes** are optional. If one of them is null, it is ignored.

E.4.7 cuModuleGetTexRef()

```
CUresult cuModuleGetTexRef(CUtexref* texRef,
                           CUmodule hmod, const char* texrefname);
```

returns in ***texref** the handle of the texture reference of name **texrefname** in the module **mod**. If no texture reference of that name exists, **cuModuleGetTexRef()** returns **CUDA_ERROR_NOT_FOUND**. This texture reference handle should not be destroyed, since it will be destroyed when the module is unloaded.

E.5 Execution Control

E.5.1 cuFuncSetBlockShape()

```
CUresult cuFuncSetBlockShape(CUfunction func,
                             int x, int y, int z);
```

specifies the X, Y and Z dimensions of the thread blocks that are created when the kernel given by **func** is launched.

E.5.2 cuFuncSetSharedSize()

```
CUresult cuFuncSetSharedSize(CUfunction func, unsigned int bytes);
```

sets through **bytes** the amount of shared memory that will be available to each thread block when the kernel given by **func** is launched.

E.5.3 cuParamSetSize()

```
CUresult cuParamSetSize(CUfunction func, unsigned int numbytes);
```

sets through **numbytes** the total size in bytes needed by the function parameters of function **func**.

E.5.4 cuParamSeti()

```
CUresult cuParamSeti(CUfunction func,
                     int offset, unsigned int value);
```

sets an integer parameter that will be specified the next time the kernel corresponding to **func** will be invoked. **offset** is a byte offset.

E.5.5 cuParamSetf()

```
CUresult cuParamSetf(CUfunction func,
                     int offset, float value);
```

sets a floating point parameter that will be specified the next time the kernel corresponding to **func** will be invoked. **offset** is a byte offset.

E.5.6 cuParamSetv()

```
CUresult cuParamSetv(CUfunction func,
                     int offset, void* ptr,
                     unsigned int numbytes);
```

copies an arbitrary amount of data into the parameter space of the kernel corresponding to **func**. **offset** is a byte offset.

E.5.7 cuParamSetTexRef()

```
CUresult cuParamSetTexRef(CUfunction func,
                          int texunit, CUtexref texRef);
```

makes the CUDA array or linear memory bound to the texture reference **texRef** available to a device program as a texture. In this version of CUDA, the texture reference must be obtained via **cuModuleGetTexRef()** and the **texunit** parameter must be set to **CU_PARAM_TR_DEFAULT**.

E.5.8 cuLaunch()

```
CUresult cuLaunch(CUfunction func);
```

invokes the kernel **func** on a 1×1 grid of blocks. The block contains the number of threads specified by a previous call to **cuFuncSetBlockShape()**.

E.5.9 cuLaunchGrid()

```
CUresult cuLaunchGrid(CUfunction func,
                      int grid_width, int grid_height);
```

invokes the kernel on a **grid_width × grid_height** grid of blocks. Each block contains the number of threads specified by a previous call to **cuFuncSetBlockShape()**.

E.6 Memory Management

E.6.1 cuMemGetInfo()

```
CUresult cuMemGetInfo(unsigned int* free, unsigned int* total);
```

returns in ***free** and ***total** respectively, the free and total amount of memory available for allocation by the CUDA context, in bytes.

E.6.2 cuMemAlloc()

```
CUresult cuMemAlloc(CUdeviceptr* devPtr, unsigned int count);
```

allocates **count** bytes of linear memory on the device and returns in ***devPtr** a pointer to the allocated memory. The allocated memory is suitably aligned for any kind of variable. The memory is not cleared. If **count** is 0, **cuMemAlloc()** returns **CUDA_ERROR_INVALID_VALUE**.

E.6.3 cuMemAllocPitch()

```
CUresult cuMemAllocPitch(CUdeviceptr* devPtr,
                         unsigned int* pitch,
                         unsigned int widthInBytes,
                         unsigned int height,
                         unsigned int elementSizeBytes);
```

allocates at least **widthInBytes*height** bytes of linear memory on the device and returns in ***devPtr** a pointer to the allocated memory. The function may pad the allocation to ensure that corresponding pointers in any given row will continue to meet the alignment requirements for coalescing as the address is updated from row to row (see Section 5.1.2.1). **elementSizeBytes** specifies the size of the largest reads and writes that will be performed on the memory range. **elementSizeBytes** may be 4, 8 or 16 (since coalesced memory transactions are not possible on other data sizes). If **elementSizeBytes** is smaller than the actual read/write size of a kernel, the kernel will run correctly, but possibly at reduced speed. The pitch returned in ***pitch** by **cuMemAllocPitch()** is the width in bytes of the allocation. The intended usage of pitch is as a separate parameter of the allocation, used to compute addresses within the 2D array. Given the row and column of an array element of type **T**, the address is computed as

```
   T* pElement = (T*)((char*)BaseAddress + Row * Pitch) + Column;
```

The pitch returned by **cuMemAllocPitch()** is guaranteed to work with **cuMemcpy2D()** under all circumstances. For allocations of 2D arrays, it is recommended that developers consider performing pitch allocations using **cuMemAllocPitch()**. Due to alignment restrictions in the hardware, this is especially true if the application will be performing 2D memory copies between different regions of device memory (whether linear memory or CUDA arrays).

E.6.4 cuMemFree()

```
CUresult cuMemFree(CUdeviceptr devPtr);
```

frees the memory space pointed to by **devPtr**, which must have been returned by a previous call to **cuMalloc()** or **cuMallocPitch()**.

E.6.5 cuMemAllocHost()

```
CUresult cuMemAllocHost(void** hostPtr, unsigned int count);
```

allocates **count** bytes of host memory that is page-locked and accessible to the device. The driver tracks the virtual memory ranges allocated with this function and automatically accelerates calls to functions such as **cuMemcpy()**. Since the memory can be accessed directly by the device, it can be read or written with much higher bandwidth than pageable memory obtained with functions such as **malloc()**. Allocating excessive amounts of memory with **cuMemAllocHost()** may degrade system performance, since it reduces the amount of memory available to the system for paging. As a result, this function is best used sparingly to allocate staging areas for data exchange between host and device.

E.6.6 cuMemFreeHost()

```
CUresult cuMemFreeHost(void* hostPtr);
```

frees the memory space pointed to by **hostPtr**, which must have been returned by a previous call to **cuMemAllocHost()**.

E.6.7 cuMemGetAddressRange()

```
CUresult cuMemGetAddressRange(CUdeviceptr* basePtr,
                              unsigned int* size,
                              CUdeviceptr devPtr);
```

returns the base address in ***basePtr** and size and ***size** of the allocation by **cuMemAlloc()** or **cuMemAllocPitch()** that contains the input pointer **devPtr**. Both parameters **basePtr** and **size** are optional. If one of them is null, it is ignored.

E.6.8 cuArrayCreate()

```
CUresult cuArrayCreate(CUarray* array,
                       const CUDA_ARRAY_DESCRIPTOR* desc);
```

creates a CUDA array according to the **CUDA_ARRAY_DESCRIPTOR** structure **desc** and returns a handle to the new CUDA array in ***array**. The **CUDA_ARRAY_DESCRIPTOR** structure is defined as such:

```
typedef struct {
    unsigned int Width;
    unsigned int Height;
    CUarray_format Format;
    unsigned int NumChannels;
} CUDA_ARRAY_DESCRIPTOR;
```

where:

- **Width** and **Height** are the width and height of the CUDA array (in elements); the CUDA array is one-dimensional if **height** is 0, two-dimensional, otherwise;
- **NumChannels** specifies the number of packed components per CUDA array element.; it may be 1, 2 or 4;
- **Format** specifies the format of the elements; **CUarray_format** is defined as such:

```
typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8  = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8    = 0x08,
    CU_AD_FORMAT_SIGNED_INT16   = 0x09,
    CU_AD_FORMAT_SIGNED_INT32   = 0x0a,
    CU_AD_FORMAT_HALF           = 0x10,
    CU_AD_FORMAT_FLOAT          = 0x20
} CUarray_format;
```

Here are examples of CUDA array descriptions:

- Description for a CUDA array of 2048 floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 2048;
desc.Height = 1;
```

- Description for a 64×64 CUDA array of floats:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.Format = CU_AD_FORMAT_FLOAT;
desc.NumChannels = 1;
desc.Width = 64;
desc.Height = 64;
```

- Description for a **width×height** CUDA array of 64-bit, 4x16-bit float16's:

```
CUDA_ARRAY_DESCRIPTOR desc;
desc.FormatFlags = CU_AD_FORMAT_HALF;
desc.NumChannels = 4;
desc.Width = width;
desc.Height = height;
```

- Description for a **width×height** CUDA array of 16-bit elements, each of which is two 8-bit unsigned chars:

```
CUDA_ARRAY_DESCRIPTOR arrayDesc;
desc.FormatFlags = CU_AD_FORMAT_UNSIGNED_INT8;
desc.NumChannels = 2;
desc.Width = width;
desc.Height = height;
```

E.6.9 cuArrayGetDescriptor()

```
CUresult cuArrayGetDescriptor(CUDA_ARRAY_DESCRIPTOR* arrayDesc,
                              CUarray array);
```

returns in ***arrayDesc** the descriptor that was used to create the CUDA array **array**. It is useful for subroutines that have been passed a CUDA array, but need to know the CUDA array parameters for validation or other purposes.

E.6.10 cuArrayDestroy()

```
CUresult cuArrayDestroy(CUarray array);
```

destroys the CUDA array **array**.

E.6.11 cuMemset()

```
CUresult cuMemsetD8(CUdeviceptr dstDevPtr,
                    unsigned char value, unsigned int count);
CUresult cuMemsetD16(CUdeviceptr dstDevPtr,
                     unsigned short value, unsigned int count);
CUresult cuMemsetD32(CUdeviceptr dstDevPtr,
                     unsigned int value, unsigned int count);
```

sets the memory range of **count** 8-, 16-, or 32-bit values to the specified value **value**.

E.6.12 cuMemset2D()

```
CUresult cuMemsetD2D8(CUdeviceptr dstDevPtr,
                      unsigned int dstPitch,
                      unsigned char value,
                      unsigned int width, unsigned int height);
CUresult cuMemsetD2D16(CUdeviceptr dstDevPtr,
                       unsigned int dstPitch,
                       unsigned short value,
                       unsigned int width, unsigned int height);
CUresult cuMemsetD2D32(CUdeviceptr dstDevPtr,
                       unsigned int dstPitch,
                       unsigned int value,
                       unsigned int width, unsigned int height);
```

sets the 2D memory range of **width** 8-, 16-, or 32-bit values to the specified value **value**. **height** specifies the number of rows to set, and **dstPitch** specifies the number of bytes between each row (see Section E.6.3). These functions perform fastest when the pitch is one that has been passed back by **cuMemAllocPitch()**.

E.6.13 cuMemcpyHtoD()

```
CUresult cuMemcpyHtoD(CUdeviceptr dstDevPtr,
                      const void *srcHostPtr,
                      unsigned int count);
```

copies from host memory to device memory. **dstDevPtr** and **srcHostPtr** specify the base addresses of the destination and source, respectively. **count** specifies the number of bytes to copy.

E.6.14 cuMemcpyDtoH()

```
CUresult cuMemcpyDtoH(void* dstHostPtr,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
```

copies from device to host memory. **dstHostPtr** and **srcDevPtr** specify the base addresses of the source and destination, respectively. **count** specifies the number of bytes to copy.

E.6.15 cuMemcpyDtoD()

```
CUresult cuMemcpyDtoD(CUdeviceptr dstDevPtr,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
```

copies from device memory to device memory. **dstDevice** and **srcDevPtr** are the base pointers of the destination and source, respectively. **count** specifies the number of bytes to copy.

E.6.16 cuMemcpyDtoA()

```
CUresult cuMemcpyDtoA(CUarray dstArray, unsigned int dstIndex,
                      CUdeviceptr srcDevPtr,
                      unsigned int count);
```

copies from device memory to a 1D CUDA array. **dstArray** and **dstIndex** specify the CUDA array handle and starting index of the destination data. **srcDevPtr** specifies the base pointer of the source. **count** specifies the number of bytes to copy.

E.6.17 cuMemcpyAtoD()

```
CUresult cuMemcpyAtoD(CUdeviceptr dstDevPtr,
                      CUarray srcArray, unsigned int srcIndex,
                      unsigned int count);
```

copies from a 1D CUDA array to device memory. **dstDevPtr** specifies the base pointer of the destination and must be naturally aligned with the CUDA array elements. **srcArray** and **srcIndex** specify the CUDA array handle and the index (in array elements) of the array element where the copy is to begin. **count** specifies the number of bytes to copy and must be evenly divisible by the array element size.

E.6.18 cuMemcpyAtoH()

```
CUresult cuMemcpyAtoH(void* dstHostPtr,
                      CUarray srcArray, unsigned int srcIndex,
                      unsigned int count);
```

copies from a 1D CUDA array to host memory. **dstHostPtr** specifies the base pointer of the destination. **srcArray** and **srcIndex** specify the CUDA array handle and starting index of the source data. **count** specifies the number of bytes to copy.

E.6.19 cuMemcpyHtoA()

```
CUresult cuMemcpyHtoA(CUarray dstArray, unsigned int dstIndex,
                      const void *srcHostPtr,
```

```
                        unsigned int count);
```

copies from host memory to a 1D CUDA array. **dstArray** and **dstIndex** specify the CUDA array handle and starting index of the destination data. **srcHostPtr** specify the base addresse of the source. **count** specifies the number of bytes to copy.

E.6.20 cuMemcpyAtoA()

```
CUresult cuMemcpyAtoA(CUarray dstArray, unsigned int dstIndex,
                      CUarray srcArray, unsigned int srcIndex,
                      unsigned int count);
```

copies from one 1D CUDA array to another. **dstArray** and **srcArray** specify the handles of the destination and source CUDA arrays for the copy, respectively. **dstIndex** and **srcIndex** specify the destination and source indices into the CUDA array. These values are in the range **[0, Width-1]** for the CUDA array; they are not byte offsets. **count** is the number of bytes to be copied. The size of the elements in the CUDA arrays need not be the same format, but the elements must be the same size; and count must be evenly divisible by that size.

E.6.21 cuMemcpy2D()

```
CUresult cuMemcpy2D(const CUDA_MEMCPY2D* copyParam);
CUresult cuMemcpy2DUnaligned(const CUDA_MEMCPY2D* copyParam);
```

perform a 2D memory copy according to the parameters specified in copyParam. The **CUDA_MEMCPY2D** structure is defined as such:

```
typedef struct CUDA_MEMCPY2D_st {

    unsigned int srcXInBytes, srcY;
    CUmemorytype srcMemoryType;
        const void *srcHost;
        CUdeviceptr srcDevice;
        CUarray srcArray;
        unsigned int srcPitch;

    unsigned int dstXInBytes, dstY;
    CUmemorytype dstMemoryType;
        void *dstHost;
        CUdeviceptr dstDevice;
        CUarray dstArray;
        unsigned int dstPitch;

    unsigned int WidthInBytes;
    unsigned int Height;
} CUDA_MEMCPY2D;
```

where:

- **srcMemoryType** and **dstMemoryType** specify the type of memory of the source and destination, respectively; **Cumemorytype_enum** is defined as such:

```
typedef enum CUmemorytype_enum {
    CU_MEMORYTYPE_HOST = 0x01,
    CU_MEMORYTYPE_DEVICE = 0x02,
```

```
    CU_MEMORYTYPE_ARRAY = 0x03
} CUmemorytype;
```

If **srcMemoryType** is **CU_MEMORYTYPE_HOST**, **srcHost** and **srcPitch** specify the (host) base address of the source data and the bytes per row to apply. **srcArray** is ignored.

If **srcMemoryType** is **CU_MEMORYTYPE_DEVICE**, **srcDevice** and **srcPitch** specify the (device) base address of the source data and the bytes per row to apply. **srcArray** is ignored.

If **srcMemoryType** is **CU_MEMORYTYPE_ARRAY**, **srcArray** specifies the handle of the source data. **srcHost**, **srcDevice** and **srcPitch** are ignored.

If **dstMemoryType** is **CU_MEMORYTYPE_HOST**, **dstHost** and **dstPitch** specify the (host) base address of the destination data and the bytes per row to apply. **dstArray** is ignored.

If **dstMemoryType** is **CU_MEMORYTYPE_DEVICE**, **dstDevice** and **dstPitch** specify the (device) base address of the destination data and the bytes per row to apply. **dstArray** is ignored.

If **dstMemoryType** is **CU_MEMORYTYPE_ARRAY**, **dstArray** specifies the handle of the destination data. **dstHost**, **dstDevice** and **dstPitch** are ignored.

- **srcXInBytes** and **srcY** specify the base address of the source data for the copy.

  For host pointers, the starting address is

  ```
  void* Start =
          (void*)((char*)srcHost+srcY*srcPitch + srcXInBytes);
  ```

  For device pointers, the starting address is

  ```
  CUdeviceptr Start = srcDevice+srcY*srcPitch+srcXInBytes;
  ```

  For CUDA arrays, **srcXInBytes** must be evenly divisible by the array element size.

- **dstXInBytes** and **dstY** specify the base address of the destination data for the copy.

  For host pointers, the base address is

  ```
  void* dstStart =
          (void*)((char*)dstHost+dstY*dstPitch + dstXInBytes);
  ```

  For device pointers, the starting address is

  ```
  CUdeviceptr dstStart = dstDevice+dstY*dstPitch+dstXInBytes;
  ```

  For CUDA arrays, **dstXInBytes** must be evenly divisible by the array element size.

- **WidthInBytes** and **Height** specify the width (in bytes) and height of the 2D copy being performed. Any pitches must be greater than or equal to **WidthInBytes**.

cuMemAllocPitch() passes back pitches that always work with **cuMemcpy2D()**. On intra-device memory copies (device ↔ device, CUDA array ↔ device, CUDA array ↔ CUDA array), **cuMemcpy2D()** may fail for pitches not computed by **cuMemAllocPitch()**. **cuMemcpy2DUnaligned()** does not have this

restriction, but may run significantly slower in the cases where **cuMemcpy2D()** would have returned an error code.

E.7 Texture Reference Management

E.7.1 cuTexRefCreate()

```
CUresult cuTexRefCreate(CUtexref* texRef);
```

creates a texture reference and returns its handle in ***texRef**. Once created, the application must call **cuTexRefSetArray()** or **cuTexRefSetAddress()** to associate the reference with allocated memory. Other texture reference functions are used to specify the format and interpretation (addressing, filtering, etc.) to be used when the memory is read through this texture reference. To associate the texture reference with a texture ordinal for a given function, the application should call **cuParamSetTexRef()**.

E.7.2 cuTexRefDestroy()

```
CUresult cuTexRefDestroy(CUtexref texRef);
```

destroys the texture reference.

E.7.3 cuTexRefSetArray()

```
CUresult cuTexRefSetArray(CUtexref texRef,
                          CUarray array,
                          unsigned int flags);
```

binds the CUDA array **array** to the texture reference **texRef**. Any previous address or CUDA array state associated with the texture reference is superseded by this function. **flags** must be set to **CU_TRSA_OVERRIDE_FORMAT**. Any CUDA array previously bound to **texRef** is unbound.

E.7.4 cuTexRefSetAddress()

```
CUresult cuTexRefSetAddress(unsigned int* byteOffset,
                            CUtexref texRef,
                            CUdeviceptr devPtr,
                            int bytes);
```

binds a linear address range to the texture reference **texRef**. Any previous address or CUDA array state associated with the texture reference is superseded by this function. Any memory previously bound to **texRef** is unbound.

Since the hardware enforces an alignment requirement on texture base addresses, **cuTexRefSetAddress()** passes back a byte offset in ***byteOffset** that must be applied to texture fetches in order to read from the desired memory. This offset must be divided by the texel size and passed to kernels that read from the texture so they can be applied to the **tex1Dfetch()** function.

If the device memory pointer was returned from **cuMemAlloc()**, the offset is guaranteed to be 0 and NULL may be passed as the **ByteOffset** parameter.

E.7.5 cuTexRefSetFormat()

```
CUresult cuTexRefSetFormat(CUtexref texRef,
                           CUarray_format format,
                           int numPackedComponents);
```

specifies the format of the data to be read by the texture reference **texRef**. **format** and **numPackedComponents** are exactly analogous to the **Format** and **NumChannels** members of the **CUDA_ARRAY_DESCRIPTOR** structure: They specify the format of each component and the number of components per array element.

E.7.6 cuTexRefSetAddressMode()

```
CUresult cuTexRefSetAddressMode(CUtexref texRef,
                                int dim, CUaddress_mode mode);
```

specifies the addressing mode **mode** for the given dimension of the texture reference **texRef**. If **dim** is zero, the addressing mode is applied to the first parameter of the functions used to fetch from the texture (see Section 4.4.5); if **dim** is 1, the second, and so on. **CUaddress_mode** is defined as such:

```
typedef enum CUaddress_mode_enum {
     CU_TR_ADDRESS_MODE_WRAP = 0,
     CU_TR_ADDRESS_MODE_CLAMP = 1,
     CU_TR_ADDRESS_MODE_MIRROR = 2,
} CUaddress_mode;
```

Note that this call has no effect if **texRef** is bound to linear memory.

E.7.7 cuTexRefSetFilterMode()

```
CUresult cuTexRefSetFilterMode(CUtexref texRef,
                               CUfilter_mode mode);
```

specifies the filtering mode **mode** to be used when reading memory through the texture reference **texRef**. **CUfilter_mode_enum** is defined as such:

```
typedef enum CUfilter_mode_enum {
    CU_TR_FILTER_MODE_POINT = 0,
    CU_TR_FILTER_MODE_LINEAR = 1
} CUfilter_mode;
```

Note that this call has no effect if **texRef** is bound to linear memory.

E.7.8 cuTexRefSetFlags()

```
CUresult cuTexRefSetFlags(CUtexref texRef, unsigned int Flags);
```

specifies optional flags to control the behavior of data returned through the texture reference. The valid flags are:

- **CU_TRSF_READ_AS_INTEGER**, which suppresses the default behavior of having the texture promote integer data to floating point data in the range [0, 1];
- **CU_TRSF_NORMALIZED_COORDINATES**, which suppresses the default behavior of having the texture coordinates range from [0, Dim) where Dim is the width or height of the CUDA array. Instead, the texture coordinates [0, 1.0) reference the entire breadth of the array dimension.

E.7.9 cuTexRefGetAddress()

```
CUresult cuTexRefGetAddress(CUdeviceptr* devPtr, CUtexref texRef);
```

returns in ***devPtr** the base address bound to the texture reference **texRef**, or returns **CUDA_ERROR_INVALID_VALUE** if the texture reference is not bound to any device memory range.

E.7.10 cuTexRefGetArray()

```
CUresult cuTexRefGetArray(CUarray* array, CUtexref texRef);
```

returns in ***array** the CUDA array bound by the texture reference **texRef**, or returns **CUDA_ERROR_INVALID_VALUE** if the texture reference is not bound to any CUDA array.

E.7.11 cuTexRefGetAddressMode()

```
CUresult cuTexRefGetAddressMode(CUaddress_mode* mode,
                                CUtexref texRef,
                                int dim);
```

returns in ***mode** the addressing mode corresponding to the dimension **dim** of the texture reference **texRef**. Currently the only valid values for **dim** are 0 and 1.

E.7.12 cuTexRefGetFilterMode()

```
CUresult cuTexRefGetFilterMode(CUfilter_mode* mode,
                               CUtexref texRef);
```

returns in ***mode** the filtering mode of the texture reference **texRef**.

E.7.13 cuTexRefGetFormat()

```
CUresult cuTexRefGetFormat(CUarray_format* format,
                           int* numPackedComponents,
                           CUtexref texRef);
```

returns in ***format** and ***numPackedComponents** the format and number of components of the CUDA array bound to the texture reference **texRef**. If **format** or **numPackedComponents** is null, it will be ignored.

E.7.14 cuTexRefGetFlags()

```
CUresult cuTexRefGetFlags(unsigned int* flags, CUtexref texRef);
```

returns in ***flags** the flags of the texture reference **texRef**.

E.8 OpenGL Interoperability

E.8.1 cuGLInit()

```
CUresult cuGLInit(void);
```

initializes OpenGL interoperability. It must be called before performing any other OpenGL interoperability operations. It may fail if the needed OpenGL driver facilities are not available.

E.8.2 cuGLRegisterBufferObject()

```
CUresult cuGLRegisterBufferObject(GLuint bufferObj);
```

registers the buffer object of ID **bufferObj** for access by CUDA. This function must be called before CUDA can map the buffer object. While it is registered, the buffer object cannot be used by any OpenGL commands except as a data source for OpenGL drawing commands.

E.8.3 cuGLMapBufferObject()

```
CUresult cuGLMapBufferObject(CUdeviceptr* devPtr,
                             unsigned int* size,
                             GLuint bufferObj);
```

maps the buffer object of ID **bufferObj** into the address space of the current CUDA context and returns in ***devPtr** and ***size** the base pointer and size of the resulting mapping.

E.8.4 cuGLUnmapBufferObject()

```
CUresult cuGLUnmapBufferObject(GLuint bufferObj);
```

unmaps the buffer object of ID **bufferObj** for access by CUDA.

E.8.5 cuGLUnregisterBufferObject()

```
CUresult cuGLUnregisterBufferObject(GLuint bufferObj);
```

unregisters the buffer object of ID **bufferObj** for access by CUDA.

E.9 Direct3D Interoperability

E.9.1 cuD3D9Begin()

```
CUresult cuD3D9Begin(IDirect3DDevice9* device);
```

initializes interoperability with the Direct3D device **device**. This function must be called before CUDA can map any objects from **device**. The application can then map vertex buffers owned by the Direct3D device until **cuD3D9End()** is called.

E.9.2 cuD3D9End()

```
CUresult cuD3D9End(void);
```

concludes interoperability with the Direct3D device previously specified to **cuD3D9Begin()**.

E.9.3 cuD3D9RegisterVertexBuffer()

```
CUresult cuD3D9RegisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

registers the Direct3D vertex buffer **VB** for access by CUDA.

E.9.4 cuD3D9MapVertexBuffer()

```
CUresult cuD3D9MapVertexBuffer(CUdeviceptr* devPtr,
                               unsigned int* size,
                               IDirect3DVertexBuffer9* VB);
```

maps the Direct3D vertex buffer **VB** into the address space of the current CUDA context and returns in ***devPtr** and ***size** the base pointer and size of the resulting mapping.

E.9.5 cuD3D9UnmapVertexBuffer()

```
CUresult cuD3D9UnmapVertexBuffer(IDirect3DVertexBuffer9* VB);
```

unmaps the vertex buffer **VB** for access by CUDA.

E.9.6 cuD3D9UnregisterVertexBuffer()

```
CUresult cuD3D9UnregisterVertexBuffer(IDirect3DVertexBuffer9* VB);
```

unregisters the vertex buffer **VB** for access by CUDA.

Appendix F. Texture Fetching

This appendix gives the formula used to compute the value returned by the texture functions of Section 4.4.5 depending on the various attributes of the texture reference (see Section 4.3.4).

The texture bound to the texture reference is represented as an array T of N texels for a one-dimensional texture or $N \times M$ texels for a two-dimensional texture. It is fetched using texture coordinates x and y.

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 $N \le x$. If x is normalized:

- In clamp addressing mode, x is replaced by 0 if $x < 0$ and $1 - 1/N$ if $1 \le x$,
- 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 and y are the non-normalized texture coordinates remapped to T's valid addressing range. x and y are derived from the normalized texture coordinates $\hat{x}$ and $\hat{y}$ as such: $x = N\hat{x}$ and $y = M\hat{y}$.

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,

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

Figure F-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 4.3.4.1).

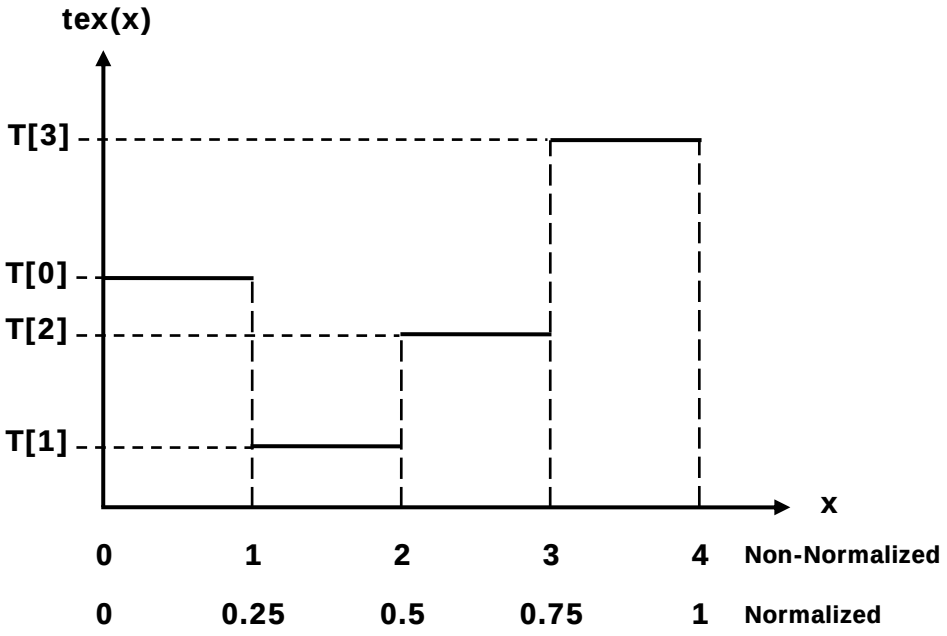


Figure F-1. Nearest-Point Sampling of a One-Dimensional Texture of Four Texels

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,

where:

- $i = floor(x_B)$, $\alpha = frac(x_B)$, $x_B = x - 0.5$,
- $j = floor(y_B)$, $\beta = frac(y_B)$, $y_B = y - 0.5$.

α and β are stored in 9-bit fixed point format with 8 bits of fractional value.

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

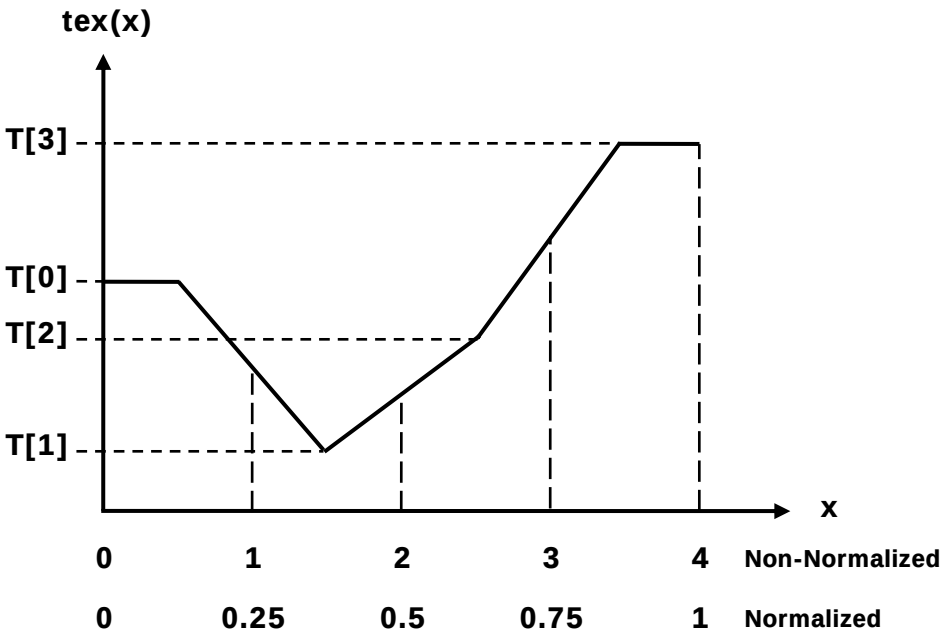


Figure F-2. Linear Filtering of a One-Dimensional Texture of Four Texels in Clamp Addressing Mode

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 $N = 4$.

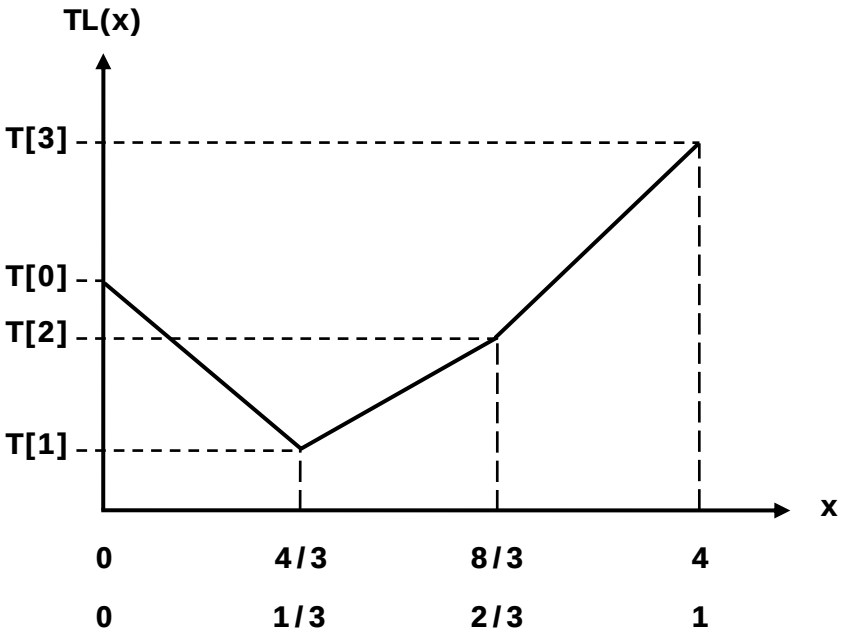


Figure F-3. One-Dimensional Table Lookup Using Linear Filtering

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