OpenCL Programming Guide for the CUDA Architecture

Version 4.0

2/14/2011

Table of Contents

Chapter 1. Introduction 5
1.1 From Graphics Processing to General-Purpose Parallel Computing 5
1.2 CUDA™: a General-Purpose Parallel Computing Architecture 7
1.3 A Scalable Programming Model 8
1.4 Document's Structure 9
Chapter 2. OpenCL on the CUDA Architecture 11
2.1 CUDA Architecture 11
2.1.1 SIMT Architecture 13
2.1.2 Hardware Multithreading 14
2.2 Compilation 15
2.2.1 PTX 15
2.2.2 Volatile 15
2.3 Compute Capability 16
2.4 Mode Switches 16
2.5 Matrix Multiplication Example 16
Chapter 3. Performance Guidelines 25
3.1 Overall Performance Optimization Strategies 25
3.2 Maximize Utilization 25
3.2.1 Application Level 25
3.2.2 Device Level 26
3.2.3 Multiprocessor Level 26
3.3 Maximize Memory Throughput 28
3.3.1 Data Transfer between Host and Device 29
3.3.2 Device Memory Accesses 30
3.3.2.1 Global Memory 30
3.3.2.2 Local Memory 31
3.3.2.3 Shared Memory 32
3.3.2.4 Constant Memory 32

3.3.2.5 Texture Memory 33
3.4 Maximize Instruction Throughput 33
3.4.1 Arithmetic Instructions 34
3.4.2 Control Flow Instructions 36
3.4.3 Synchronization Instruction 36
Appendix A. CUDA-Enabled GPUs 39
Appendix B. Mathematical Functions Accuracy 41
B.1 Standard Functions 41
B.1.1 Single-Precision Floating-Point Functions 41
B.1.2 Double-Precision Floating-Point Functions 43
B.2 Native Functions 45
Appendix C. Compute Capabilities 47
C.1 Features and Technical Specifications 47
C.2 Floating-Point Standard 48
C.3 Compute Capability 1.x 50
C.3.1 Architecture 50
C.3.2 Global Memory 51
C.3.2.1 Devices of Compute Capability 1.0 and 1.1 51
C.3.2.2 Devices of Compute Capability 1.2 and 1.3 51
C.3.3 Shared Memory 52
C.3.3.1 32-Bit Strided Access 52
C.3.3.2 32-Bit Broadcast Access 52
C.3.3.3 8-Bit and 16-Bit Access 53
C.3.3.4 Larger Than 32-Bit Access 53
C.4 Compute Capability 2.0 54
C.4.1 Architecture 54
C.4.2 Global Memory 55
C.4.3 Shared Memory 57
C.4.3.1 32-Bit Strided Access 57
C.4.3.2 Larger Than 32-Bit Access 57
C.4.4 Constant Memory 58

List of Figures

Figure 1-1. Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU 6

Figure 1-2. The GPU Devotes More Transistors to Data Processing 7

Figure 1-3. CUDA is Designed to Support Various Languages and Application Programming Interfaces 8

Figure 1-4. Automatic Scalability 9

Figure 2-1. Grid of Thread Blocks 12

Figure 2-2. Matrix Multipliation without Shared Memory 19

Figure 2-3. Matrix Multipliation with Shared Memory 24

Chapter 1. Introduction

1.1 From Graphics Processing to General-Purpose Parallel Computing

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

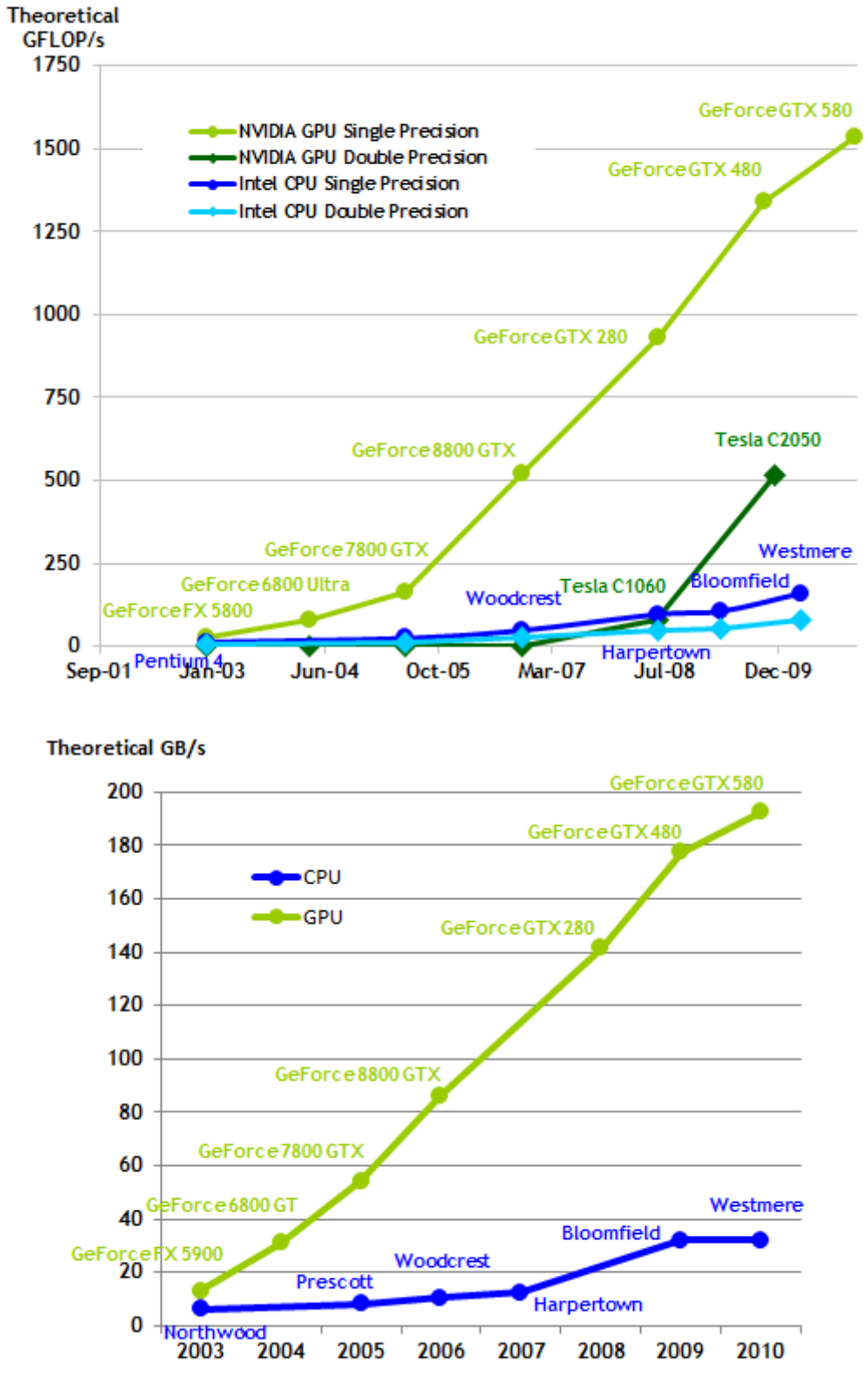


Figure 1-1. Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

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

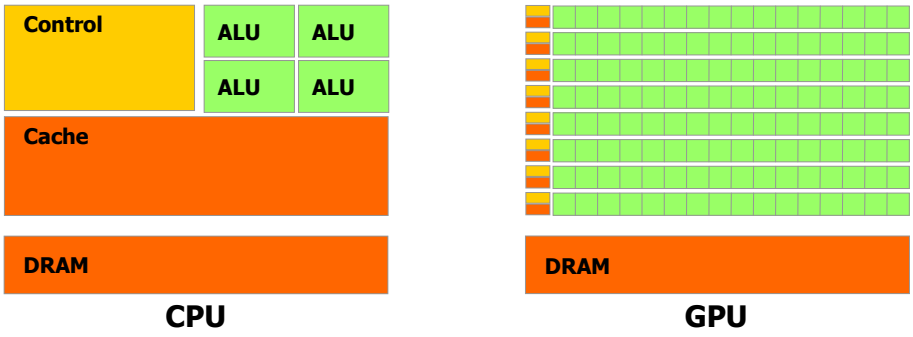


Figure 1-2. The GPU Devotes More Transistors to Data Processing

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

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

1.2 CUDA™: a General-Purpose Parallel Computing Architecture

In November 2006, NVIDIA introduced CUDA™, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that leverages the parallel compute engine in NVIDIA GPUs to

solve many complex computational problems in a more efficient way than on a CPU.

As illustrated by Figure 1-3, there are several languages and application programming interfaces that can be used to program the CUDA architecture.

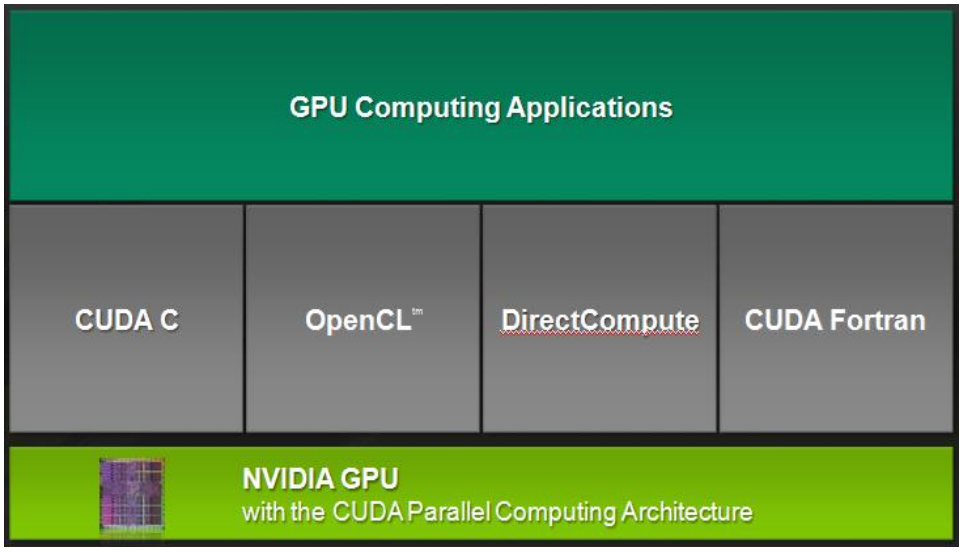


Figure 1-3. CUDA is Designed to Support Various Languages and Application Programming Interfaces

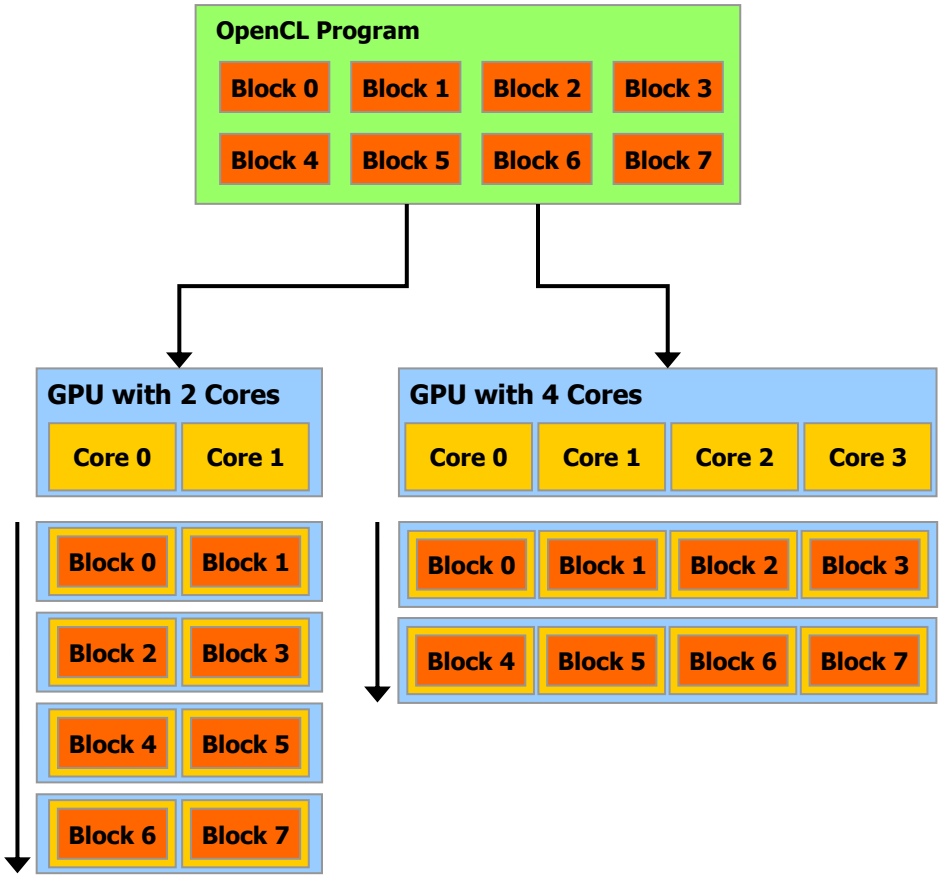
1.3 A Scalable Programming Model

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

The CUDA parallel programming model is designed to overcome this challenge with three key abstractions: a hierarchy of thread groups, a hierarchy of shared memories, and barrier synchronization.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block. This decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables automatic scalability. Indeed, each block of threads can be scheduled on any of the available processor cores, in any order, concurrently or sequentially, so that a compiled OpenCL program can execute on any number of processor cores as illustrated by Figure 1-4, and only the runtime system needs to know the physical processor count.

This scalable programming model allows the CUDA architecture to span a wide market range by simply scaling the number of processors and memory partitions: from the high-performance enthusiast GeForce GTX 280 GPU and professional Quadro and Tesla computing products to a variety of inexpensive, mainstream GeForce GPUs (see Appendix A for a list of all CUDA-enabled GPUs).



A multithreaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more cores will automatically execute the program in less time than a GPU with fewer cores.

Figure 1-4.Automatic Scalability

1.4 Document's Structure

This document is organized into the following chapters:

- Chapter 1 is a general introduction to GPU computing and the CUDA architecture.
- Chapter 2 describes how the OpenCL architecture maps to the CUDA architecture and the specifics of NVIDIA's OpenCL implementation.
- Chapter 3 gives some guidance on how to achieve maximum performance.
- Appendix A lists the CUDA-enabled GPUs with their technical specifications.

- Appendix B lists the accuracy of each mathematical function on the CUDA architecture.
- Appendix C gives the technical specifications of various devices, as well as more architectural details.

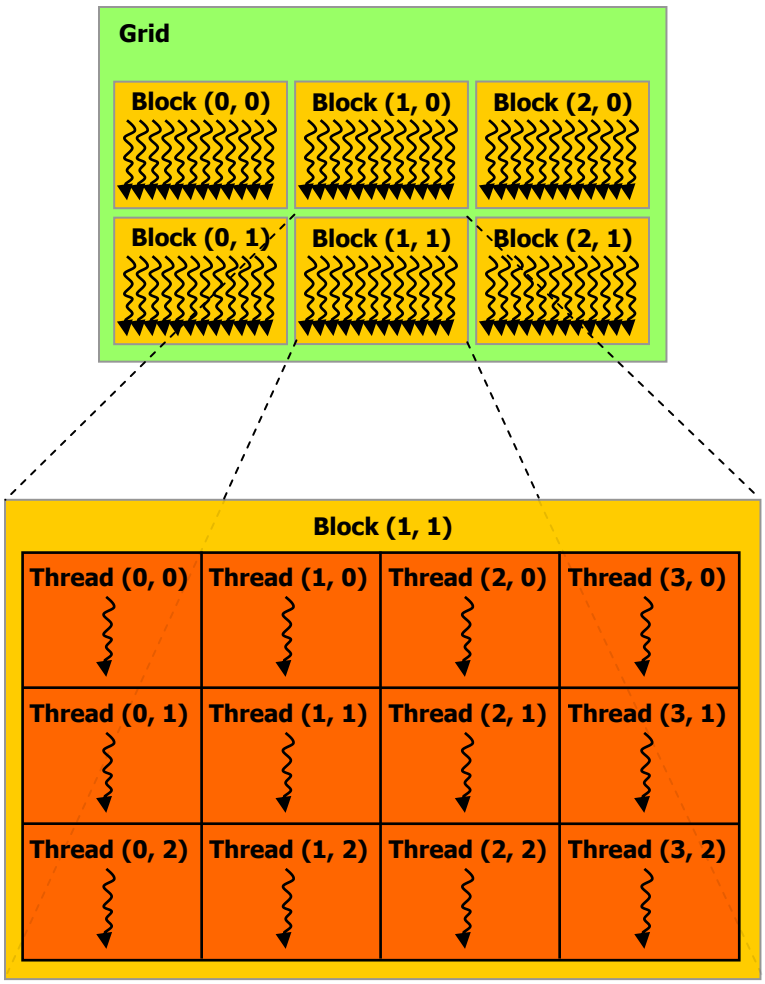
Chapter 2. OpenCL on the CUDA Architecture

2.1 CUDA Architecture

The CUDA architecture is a close match to the OpenCL architecture.

A CUDA device is built around a scalable array of multithreaded *Streaming Multiprocessors* (SMs). A multiprocessor corresponds to an OpenCL compute unit.

A multiprocessor executes a CUDA *thread* for each OpenCL work-item and a *thread block* for each OpenCL work-group. A kernel is executed over an OpenCL NDRange by a *grid of thread blocks*. As illustrated in Figure 2-1, each of the thread blocks that execute a kernel is therefore uniquely identified by its work-group ID, and each thread by its global ID or by a combination of its local ID and work-group ID.



A kernel is executed over an NDRange by a grid of thread blocks.

Figure 2-1.Grid of Thread Blocks

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

When an OpenCL program on the host invokes a kernel, the work-groups are enumerated and distributed as thread blocks to the multiprocessors with available execution capacity. The threads of a thread block execute concurrently on one multiprocessor. As thread blocks terminate, new blocks are launched on the vacated multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To manage such a large amount of threads, it employs a unique architecture called *SIMT* (*Single-Instruction, Multiple-Thread*) that is described in Section 2.1.1. To

maximize utilization of its functional units, it leverages thread-level parallelism by using hardware multithreading as detailed in Section 2.1.2, more so than instruction-level parallelism within a single thread (instructions are pipelined, but unlike CPU cores they are executed in order and there is no branch prediction and no speculative execution).

Sections 2.1.1 and 2.1.2 describe the architecture features of the streaming multiprocessor that are common to all devices. Sections C.3.1 and C.4.1 provide the specifics for devices of compute capabilities 1.x and 2.0, respectively (see Section 2.3 for the definition of compute capability).

2.1.1 SIMT Architecture

The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. Individual threads composing a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. The term *warp* originates from weaving, the first parallel thread technology. A *half-warp* is either the first or second half of a warp. A *quarter-warp* is either the first, second, third, or fourth quarter of a warp.

When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps that get scheduled by a *warp scheduler* for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread 0. Section 2.1 describes how thread IDs relate to thread indices in the block.

A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken, disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The SIMT architecture is akin to SIMD (Single Instruction, Multiple Data) vector organizations in that a single instruction controls multiple processing elements. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. For the purposes of correctness, the programmer can essentially ignore the SIMT behavior; however, substantial performance improvements can be realized by taking care that the code seldom requires threads in a warp to diverge. In practice, this is analogous to the role of cache lines in traditional code: Cache line size can be safely ignored when designing for correctness but must be considered in the code structure when designing for peak performance. Vector architectures, on the other hand, require the software to coalesce loads into vectors and manage divergence manually.

If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location varies depending on the compute

capability of the device (see Sections C.3.2, C.3.3, C.4.2, and C.4.3) and which thread performs the final write is undefined.

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

2.1.2 Hardware Multithreading

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

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

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

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

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

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

The total number of registers R_{block} allocated for a block is as follows:

For devices of compute capability 1.x:

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

For devices of compute capability 2.0:

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

- G_W is the warp allocation granularity, equal to 2 (compute capability 1.x only),
- R_k is the number of registers used by the kernel,
- G_T is the thread allocation granularity, equal to 256 for devices of compute capability 1.0 and 1.1, and 512 for devices of compute capability 1.2 and 1.3, and 64 for devices of compute capability 2.0.

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

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

- S_k is the amount of shared memory used by the kernel in bytes,

G_S is the shared memory allocation granularity, which is equal to 512 for devices of compute capability 1.x and 128 for devices of compute capability 2.0.

2.2 Compilation

2.2.1 PTX

Kernels written in OpenCL C are compiled into *PTX*, which is CUDA's instruction set architecture and is described in a separate document.

Currently, the PTX intermediate representation can be obtained by calling **clGetProgramInfo()** with **CL_PROGRAM_BINARIES**. It can be passed to **clCreateProgramWithBinary()** to create a program object only if it is produced and consumed by the same driver. This will likely not be supported in future versions.

2.2.2 Volatile

Only after the execution of **barrier()**, **mem_fence()**, **read_mem_fence()**, or **write_mem_fence()** are prior writes to global or shared memory of a given thread guaranteed to be visible by other threads. As long as this requirement is met, the compiler is free to optimize reads and writes to global or shared memory. For example, in the code sample below, the first reference to **myArray[tid]** compiles into a global or shared memory read instruction, but the second reference does not as the compiler simply reuses the result of the first read.

```
// myArray is an array of non-zero integers
// located in global or shared memory
__kernel void myKernel(__global int* result) {
    int tid = get_local_id(0);
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```

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

This behavior can be changed using the **volatile** keyword: If a variable located in global or shared memory is declared as volatile, the compiler assumes that its value can be changed at any time by another thread and therefore any reference to this variable compiles to an actual memory read instruction.

Note that even if **myArray** is declared as volatile in the code sample above, there is no guarantee, in general, that **ref2** will be equal to 2 in thread **tid** since thread

tid might read **myArray[tid]** into **ref2** before thread **tid-1** overwrites its value by 2. Synchronization is required.

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

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

Appendix A lists of all CUDA-enabled devices along with their compute capability. Appendix C gives the technical specifications of each compute capability.

The compute capability of a device can be programmatically queried using the **cl_nv_device_attribute_query** extension.

2.4 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 OpenCL applications. Therefore, a mode switch results in any call to the OpenCL runtime to fail and return an invalid context error.

2.5 Matrix Multiplication Example

The following matrix multiplication example illustrates the typical data-parallel approach used by OpenCL applications to achieve good performance on GPUs. It also illustrates the use of OpenCL local memory that maps to shared memory on the CUDA architecture. Shared memory is much faster than global memory as detailed in Section 3.3.2.3, so any opportunity to replace global memory accesses by shared memory accesses should be exploited.

The following code sample is a straightforward implementation of matrix multiplication that does not take advantage of shared memory. Each thread reads one row of *A* and one column of *B* and computes the corresponding element of *C* as illustrated in Figure 2-2. *A* is therefore read *B.width* times from global memory and *B* is read *A.height* times.

```
// Host code

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

// Thread block size
#define BLOCK_SIZE 16

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMulHost(const Matrix A, const Matrix B, Matrix C,
                const cl_context context,
                const cl_kernel matMulKernel,
                const cl_command_queue queue)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    d_A.elements = clCreateBuffer(context,
                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                        size, A.elements, 0);
    Matrix d_B;
    d_B.width = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    d_B.elements = clCreateBuffer(context,
                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                        size, B.elements, 0);

    // Allocate C in device memory
    Matrix d_C;
    d_C.width = C.width; d_C.height = C.height;
    size = C.width * C.height * sizeof(float);
    d_C.elements = clCreateBuffer(context,
                        CL_MEM_WRITE_ONLY, size, 0, 0);

    // Invoke kernel
    cl_uint i = 0;
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_A.width),    (void*)&d_A.width);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_A.height),   (void*)&d_A.height);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_A.elements), (void*)&d_A.elements);
```

```
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_B.width),    (void*)&d_B.width);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_B.height),   (void*)&d_B.height);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_B.elements), (void*)&d_B.elements);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_C.width),    (void*)&d_C.width);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_C.height),   (void*)&d_C.height);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_C.elements), (void*)&d_C.elements);
    size_t localWorkSize[] = { BLOCK_SIZE, BLOCK_SIZE };
    size_t globalWorkSize[] =
                { B.width / dimBlock.x, A.height / dimBlock.y };
    clEnqueueNDRangeKernel(queue, matMulKernel, 2, 0,
                           globalWorkSize, localWorkSize,
                           0, 0, 0);

    // Read C from device memory
    clEnqueueReadBuffer(queue, d_C.elements, CL_TRUE, 0, size,
                        C.elements, 0, 0, 0);

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

```
// Kernel code

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

// Thread block size
#define BLOCK_SIZE 16

// Matrix multiplication function called by MatMulKernel()
void MatMul(Matrix A, Matrix B, Matrix C)
{
    float Cvalue = 0;
    int row = get_global_id(1);
    int col = get_global_id(0);
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];
    C.elements[row * C.width + col] = Cvalue;
}

// Matrix multiplication kernel called by MatMulHost()
__kernel void MatMulKernel(
```

```
        int Awidth, int Aheight, __global float* Aelements,
        int Bwidth, int Bheight, __global float* Belements,
        int Cwidth, int Cheight, __global float* Celements)
{
    Matrix A = { Awidth, Aheight, Aelements };
    Matrix B = { Bwidth, Bheight, Belements };
    Matrix C = { Cwidth, Cheight, Celements };
    matrixMul(A, B, C);
}
```

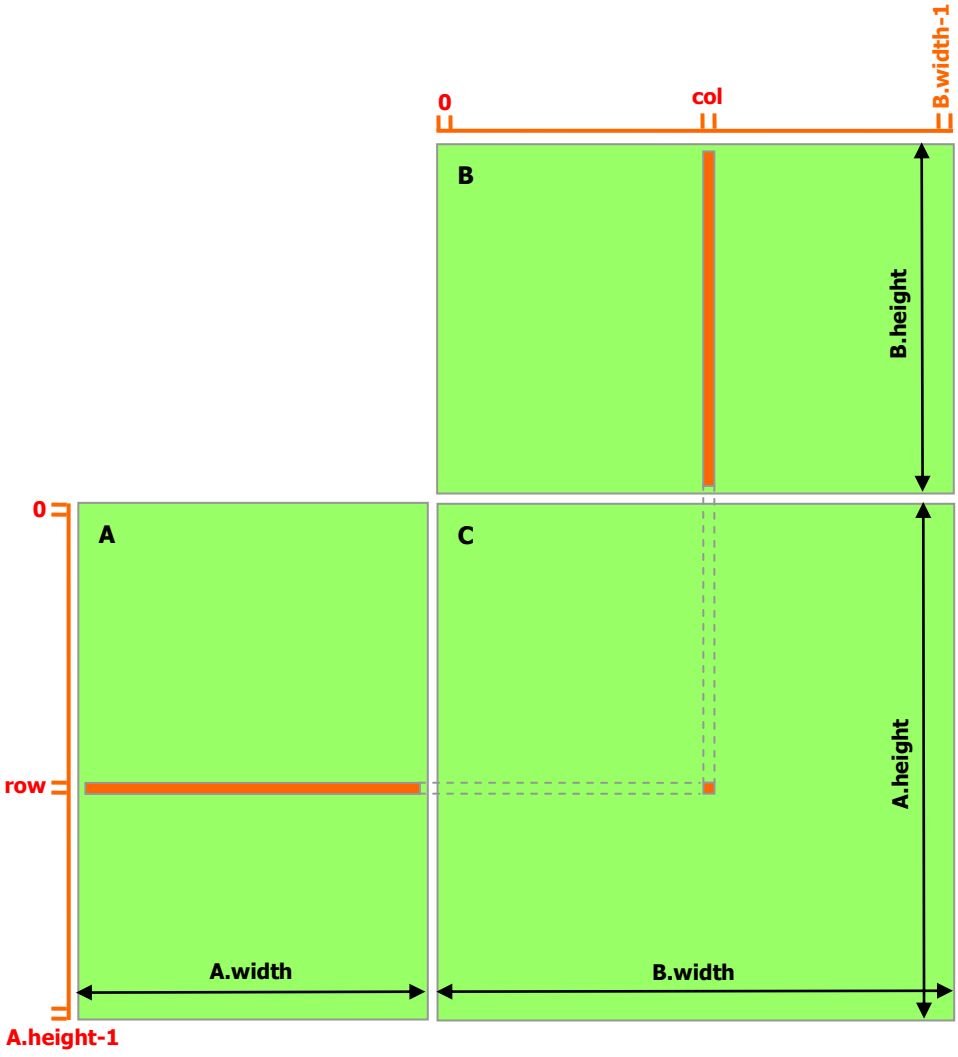


Figure 2-2.Matrix Multipliation without Shared Memory

The following code sample is an implementation of matrix multiplication that does take advantage of shared memory. In this implementation, each thread block is responsible for computing one square sub-matrix C_{sub} of C and each thread within the block is responsible for computing one element of C_{sub}. As illustrated in Figure 2-3, C_{sub} is equal to the product of two rectangular matrices: the sub-matrix of A of dimension *(A.width, block_size)* that has the same line indices as C_{sub}, and the sub-matrix of B of dimension *(block_size, A.width)* that has the same column indices as

C_{sub}. In order to fit into the device's resources, these two rectangular matrices are divided into as many square matrices of dimension *block_size* as necessary and C_{sub} is computed as the sum of the products of these square matrices. Each of these products is performed by first loading the two corresponding square matrices from global memory to shared memory with one thread loading one element of each matrix, and then by having each thread compute one element of the product. Each thread accumulates the result of each of these products into a register and once done writes the result to global memory.

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

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

```
// Host code

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

// Thread block size
#define BLOCK_SIZE 16

// Matrix multiplication - Host code
// Matrix dimensions are assumed to be multiples of BLOCK_SIZE
void MatMulHost(const Matrix A, const Matrix B, Matrix C,
                const cl_context context,
                const cl_kernel matMulKernel,
                const cl_command_queue queue)
{
    // Load A and B to device memory
    Matrix d_A;
    d_A.width = d_A.stride = A.width; d_A.height = A.height;
    size_t size = A.width * A.height * sizeof(float);
    d_A.elements = clCreateBuffer(context,
                          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                          size, A.elements, 0);
    Matrix d_B;
    d_B.width = d_B.stride = B.width; d_B.height = B.height;
    size = B.width * B.height * sizeof(float);
    d_B.elements = clCreateBuffer(context,
                          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                          size, B.elements, 0);

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

```
    d_C.elements = clCreateBuffer(context,
                         CL_MEM_WRITE_ONLY, size, 0, 0);

    // Invoke kernel
    cl_uint i = 0;
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_A.width),    (void*)&d_A.width);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_A.height),   (void*)&d_A.height);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_A.stride),   (void*)&d_A.stride);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_A.elements), (void*)&d_A.elements);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_B.width),    (void*)&d_B.width);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_B.height),   (void*)&d_B.height);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_B. stride),  (void*)&d_B.stride);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_B.elements), (void*)&d_B.elements);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_C.width),    (void*)&d_C.width);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_C.height),   (void*)&d_C.height);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_C.stride),   (void*)&d_C.stride);
    clSetKernelArg(matMulKernel, i++,
                   sizeof(d_C.elements), (void*)&d_C.elements);
    size_t localWorkSize[] = { BLOCK_SIZE, BLOCK_SIZE };
    size_t globalWorkSize[] =
                { B.width / dimBlock.x, A.height / dimBlock.y };
    clEnqueueNDRangeKernel(queue, matMulKernel, 2, 0,
                           globalWorkSize, localWorkSize,
                           0, 0, 0);

    // Read C from device memory
    clEnqueueReadBuffer(queue, d_C.elements, CL_TRUE, 0, size,
                        C.elements, 0, 0, 0);

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

```
// Kernel code

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

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

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

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

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

// Matrix multiplication function called by MatMulKernel()
void MatMul(Matrix C, Matrix A, Matrix B,
            __local float As[BLOCK_SIZE][BLOCK_SIZE],
            __local float Bs[BLOCK_SIZE][BLOCK_SIZE])
{
    // Block row and column
    int blockRow = get_group_id(1);
    int blockCol = get_group_id(0);

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

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

    // Thread row and column within Csub
    int row = get_local_id(1);
    int col = get_local_id(0);

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

        // Get sub-matrix Asub of A
```

```
        Matrix Asub = GetSubMatrix(A, blockRow, m);

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

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

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

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

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

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

// Matrix multiplication kernel called by MatMulHost()
__kernel void matrixMulKernel(
  int Cwidth, int Cheight, int Cstride, __global float* Celements,
  int Awidth, int Aheight, int Astride, __global float* Aelements,
  int Bwidth, int Bheight, int Bstride, __global float* Belements,
  __local float As[BLOCK_SIZE][BLOCK_SIZE],
  __local float Bs[BLOCK_SIZE][BLOCK_SIZE])
{
    Matrix C = { Cwidth, Cheight, Cstride, Celements };
    Matrix A = { Awidth, Aheight, Astride, Aelements };
    Matrix B = { Bwidth, Bheight, Bstride, Belements };
    MatMul(A, B, C, As, Bs);
}
```

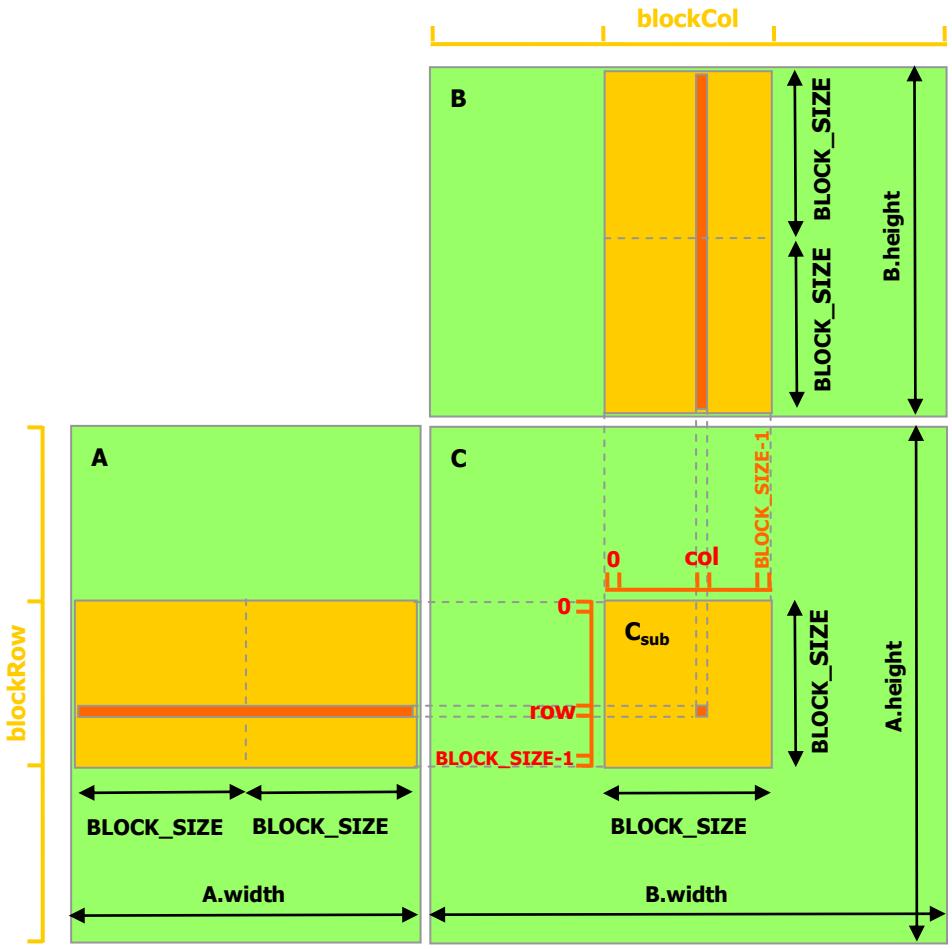


Figure 2-3.Matrix Multipliation with Shared Memory

Chapter 3. Performance Guidelines

3.1 Overall Performance Optimization Strategies

Performance optimization revolves around three basic strategies:

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

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

3.2 Maximize Utilization

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

3.2.1 Application Level

At a high level, the application should maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using queues. It should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices.

For the parallel workloads, at points in the algorithm where parallelism is broken because some threads need to synchronize in order to share data with each other, there are two cases: Either these threads belong to the same block, in which case they should use **`barrier()`** and share data through shared memory within the

same kernel invocation, or they belong to different blocks, in which case they must share data through global memory using two separate kernel invocations, one for writing to and one for reading from global memory. The second case is much less optimal since it adds the overhead of extra kernel invocations and global memory traffic. Its occurrence should therefore be minimized by mapping the algorithm to the OpenCL programming model in such a way that the computations that require inter-thread communication are performed within a single thread block as much as possible.

3.2.2 Device Level

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

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

For devices of compute capability 2.0, multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using queues to enable enough kernels to execute concurrently.

3.2.3 Multiprocessor Level

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

As described in Section 2.1.2, a GPU multiprocessor relies on thread-level parallelism to maximize utilization of its functional units. Utilization is therefore directly linked to the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute, if any, and issues the next instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called *latency*, and full utilization is achieved when the warp scheduler always has some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when the latency of each warp is completely "hidden" by other warps. How many instructions are required to hide latency depends on the instruction throughput. For example, to hide a latency of L clock cycles with basic single-precision floating-point arithmetic instructions (scheduled on CUDA cores):

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

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

If all input operands are registers, latency is caused by register dependencies, i.e., some of the input operands are written by some previous instruction(s) whose

execution has not completed yet. In the case of a back-to-back register dependency (i.e., some input operand is written by the previous instruction), the latency is equal to the execution time of the previous instruction and the warp scheduler must schedule instructions for different warps during that time. Execution time varies depending on the instruction, but it is typically about 22 clock cycles, which translates to 6 warps for devices of compute capability 1.x and 11 warps for devices of compute capability 2.0.

If some input operand resides in off-chip memory, the latency is much higher: 400 to 800 clock cycles. The number of warps required to keep the warp scheduler busy during such high latency periods depends on the kernel code; in general, more warps are required if the ratio of the number of instructions with no off-chip memory operands (i.e., arithmetic instructions most of the time) to the number of instructions with off-chip memory operands is low (this ratio is commonly called the arithmetic intensity of the program). If this ratio is 10, for example, then to hide latencies of about 600 clock cycles, about 15 warps are required for devices of compute capability 1.x and about 30 for devices of compute capability 2.0.

Another reason a warp is not ready to execute its next instruction is that it is waiting at some memory fence or synchronization point. A synchronization point can force the multiprocessor to idle as more and more warps wait for other warps in the same block to complete execution of instructions prior to the synchronization point. Having multiple resident blocks per multiprocessor can help reduce idling in this case, as warps from different blocks do not need to wait for each other at synchronization points.

The number of blocks and warps residing on each multiprocessor for a given kernel call depends on the NDRange of the call, the memory resources of the multiprocessor, and the resource requirements of the kernel as described in Section 2.1.2. To assist programmers in choosing thread block size based on register and shared memory requirements, the CUDA Software Development Kit provides a spreadsheet, called the CUDA Occupancy Calculator, where occupancy is defined as the ratio of the number of resident warps to the maximum number of resident warps (given in Appendix C for various compute capabilities).

Register, local, shared, and constant memory usages are reported by the compiler when compiling with the **`-cl-nv-verbose`** build option (see **`cl_nv_compiler_options`** extension).

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

The number of registers used by a kernel can have a significant impact on the number of resident warps. For example, for devices of compute capability 1.2, if a kernel uses 16 registers and each block has 512 threads and requires very little shared memory, then two blocks (i.e., 32 warps) can reside on the multiprocessor since they require 2x512x16 registers, which exactly matches the number of registers available on the multiprocessor. But as soon as the kernel uses one more register, only one block (i.e., 16 warps) can be resident since two blocks would require 2x512x17 registers, which is more registers than are available on the multiprocessor. Therefore, the compiler attempts to minimize register usage while keeping register

spilling (see Section 3.3.2.2) and the number of instructions to a minimum. Register usage can be controlled using the **`-cl-nv-maxrregcount`** build option.

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

The effect of NDRange on performance for a given kernel call generally depends on the kernel code. Experimentation is therefore recommended and applications should set the work-group size explicitly as opposed to rely on the OpenCL implementation to determine the right size (by setting *local_work_size* to NULL in **`clEnqueueNDRangeKernel()`**). Applications can also parameterize NDRanges based on register file size and shared memory size, which depends on the compute capability of the device, as well as on the number of multiprocessors and memory bandwidth of the device, all of which can be queried using the runtime or driver API (see reference manual).

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

3.3 Maximize Memory Throughput

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

That means minimizing data transfers between the host and the device, as detailed in Section 3.3.1, since these have much lower bandwidth than data transfers between global memory and the device.

That also means minimizing data transfers between global memory and the device by maximizing use of on-chip memory: shared memory and caches (i.e. L1/L2 caches available on devices of compute capability 2.0, texture cache and constant cache available on all devices).

Shared memory is equivalent to a user-managed cache: The application explicitly allocates and accesses it. As illustrated in Section 2.5, a typical programming pattern is to stage data coming from device memory into shared memory; in other words, to have each thread of a block:

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

For some applications (e.g. for which global memory accesses are data-dependent), a traditional hardware-managed cache is more appropriate to exploit data locality. As mentioned in Section C.4.1, for devices of compute capability 2.0, the same on-chip

memory is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call.

The throughput of memory accesses by a kernel can vary by an order of magnitude depending on access pattern for each type of memory. The next step in maximizing memory throughput is therefore to organize memory accesses as optimally as possible based on the optimal memory access patterns described in Sections 3.3.2.1, 3.3.2.3, 3.3.2.4, and 3.3.2.5. This optimization is especially important for global memory accesses as global memory bandwidth is low, so non-optimal global memory accesses have a higher impact on performance.

3.3.1 Data Transfer between Host and Device

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

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

Finally, higher performance for data transfers between host and device is achieved for memory objects allocated in *page-locked* (also known as *pinned*) host memory (as opposed to regular pageable host memory allocated by **`malloc()`**), which has several benefits:

- On systems with a front-side bus, higher performance for data transfers between host and device is achieved if host memory is allocated as page-locked.
- For some devices, copies between page-locked host memory and device memory can be performed concurrently with kernel execution.
- For some devices, page-locked host memory can be mapped into the device's address space. In this case, there is no need to allocate any device memory and to explicitly copy data between device and host memory. Data transfers are implicitly performed each time the kernel accesses the mapped memory. For maximum performance, these memory accesses must be coalesced like if they were accesses to global memory (see Section 3.3.2.1). Assuming that they are and that the mapped memory is read or written only once, avoiding explicit copies between device and host memory can be a win performance-wise. It is always a win on integrated systems where device memory and host memory are physically the same and therefore any copy between host and device memory is superfluous.

OpenCL applications do not have direct control over whether memory objects are allocated in page-locked memory or not, but they can create objects using the **`CL_MEM_ALLOC_HOST_PTR`** flag and such objects are likely to be allocated in page-locked memory by the driver for best performance.

3.3.2 Device Memory Accesses

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

3.3.2.1 Global Memory

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

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

How many transactions are necessary and how throughput is ultimately affected varies with the compute capability of the device. For devices of compute capability 1.0 and 1.1, the requirements on the distribution of the addresses across the threads to get any coalescing at all are very strict. They are much more relaxed for devices of higher compute capabilities. For devices of compute capability 2.0, the memory transactions are cached, so data locality is exploited to reduce impact on throughput. Sections C.3.2 and C.4.2 give more details on how global memory accesses are handled for various compute capabilities.

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

- Following the most optimal access patterns based on Sections C.3.2 and C.4.2,
- Using data types that meet the size and alignment requirement detailed in Section 3.3.2.1.1,
- Padding data in some cases, for example, when accessing a two-dimensional array as described in Section 3.3.2.1.2.

3.3.2.1.1 Size and Alignment Requirement

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

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

The alignment requirement is automatically fulfilled for built-in types.

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

```
struct {
    float a;
    float b;
} __attribute__ ((aligned(8)));
```

or

```
struct {
    float a;
    float b;
    float c;
} __attribute__ ((aligned(16)));
```

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

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

3.3.2.1.2 Two-Dimensional Arrays

A common global memory access pattern is when each thread of index **(tx,ty)** uses the following address to access one element of a 2D array of width **width**, located at address **BaseAddress** of type **type*** (where **type** meets the requirement described in Section 3.3.2.1.1):

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

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

In particular, this means that an array whose width is not a multiple of this size will be accessed much more efficiently if it is actually allocated with a width rounded up to the closest multiple of this size and its rows padded accordingly.

3.3.2.2 Local Memory

CUDA local memory accesses only occur for some automatic variables. Automatic variables that the compiler is likely to place in local memory are:

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

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

The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing as described in Section 3.3.2.1. Local memory is however organized such that consecutive 32-bit words are accessed by consecutive thread IDs. Accesses are therefore fully coalesced as long as all threads in a warp access the same relative address (e.g. same index in an array variable, same member in a structure variable).

On devices of compute capability 2.0, local memory accesses are always cached in L1 and L2 in the same way as global memory accesses (see Section C.4.2).

3.3.2.3 Shared Memory

Shared memory is where OpenCL local memory resides.

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

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

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

To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts. This is described in Sections C.3.3 and C.4.3 for devices of compute capability 1.x and 2.0, respectively.

3.3.2.4 Constant Memory

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

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

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

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

3.3.2.5 Texture Memory

Texture memory is cached so an image read 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 image addresses that are close together will achieve best performance. Also, it is designed for streaming reads with a constant latency, i.e. a cache hit reduces DRAM bandwidth demand, but not read latency.

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

- If the memory reads do not follow the access patterns that global or constant memory reads must respect to get good performance (see Sections 3.3.2.1 and 3.3.2.4), higher bandwidth can be achieved providing that there is locality in the texture fetches (this is less likely for devices of compute capability 2.0 given that global memory reads are cached on these devices);
- Addressing calculations are performed outside the kernel by dedicated units;
- Packed data may be broadcast to separate variables in a single operation;
- 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range [0.0, 1.0] or [-1.0, 1.0].

However, within the same kernel call, the texture cache is not kept coherent with respect to image writes, so that any image read to an address that has been written to via an image write in the same kernel call returns undefined data. In other words, a thread can safely read via an image object some memory location only if this memory location has been updated by a previous kernel call or memory copy, but not if it has been previously updated by the same thread or another thread from the same kernel call.

3.4 Maximize Instruction Throughput

To maximize instruction throughput the application should:

- Minimize the use of arithmetic instructions with low throughput; this includes trading precision for speed when it does not affect the end result, such as using **`native_*`** instead of regular functions (see Section B.2), single-precision instead of double-precision, or using the **`-cl-mad-enable`** build option;
- Minimize divergent warps caused by control flow instructions as detailed in Section 3.4.2;
- Reduce the number of instructions, for example, by optimizing out synchronization points whenever possible as described in Section 3.4.3 or by using restricted pointers.

In this section, throughputs are given in number of operations per clock cycle per multiprocessor. For a warp size of 32, one instruction results in 32 operations. Therefore, if T is the number of operations per clock cycle, the instruction throughput is one instruction every 32/T clock cycles.

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

3.4.1 Arithmetic Instructions

Table 3-1 gives the throughputs of the arithmetic instructions that are natively supported in hardware for devices of various compute capabilities. For devices of compute capability 2.0, two different warps execute half of the operations each clock cycle (see Section C.4.1).

Table 3-1. Throughput of Native Arithmetic Instructions (Operations per Clock Cycle per Multiprocessor)

	Compute Capability 1.x	Compute Capability 2.0
32-bit floating-point add, multiply, multiply-add	8	32
64-bit floating-point add, multiply, multiply-add	1	16
32-bit integer add, logical operation, shift, compare	8	32
24-bit integer multiply (`mul24(x,y)`)	8	Multiple instructions
32-bit integer multiply, multiply-add, sum of absolute difference	Multiple instructions	32
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`native_log`), base-2 exponential (`native_exp`), sine (`native_sin`), cosine (`native_cos`)	2	4
Type conversions	8	32

Other instructions and functions are implemented on top of the native instructions. The implementation may be different for devices of compute capability 1.x and devices of compute capability 2.0, and the number of native instructions after compilation may fluctuate with every compiler version.

Single-Precision Floating-Point Division

`native_divide(x, y)` provides faster single-precision floating-point division than the division operator.

Single-Precision Floating-Point Reciprocal Square Root

To preserve IEEE-754 semantics the compiler cannot optimize `1.0/sqrtf()` into `rsqrtf()`. It is therefore recommended to invoke `native_rsqrt()` directly where desired.

Single-Precision Floating-Point Square Root

Single-precision floating-point square root is implemented as a reciprocal square root followed by a reciprocal instead of a reciprocal square root followed by a multiplication so that it gives correct results for 0 and infinity. Therefore, its throughput is 1 operation per clock cycle for devices of compute capability 1.x and 2 operations per clock cycle for devices of compute capability 2.0.

Sine and Cosine

`sin(x)`, `cos(x)`, `tan(x)`, `sincos(x)` are much more expensive and even more so if the absolute value of `x` needs to be reduced.

More precisely, the argument reduction code comprises two code paths referred to as the fast path and the slow path, respectively.

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

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

As the slow path requires more registers than the fast path, an attempt has been made to reduce register pressure in the slow path by storing some intermediate variables in CUDA local memory, which may affect performance because of local memory high latency and bandwidth (see Section 3.3.2.2). At present, 28 bytes of local memory are used by single-precision functions, and 44 bytes are used by double-precision functions. However, the exact amount is subject to change.

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

Integer Arithmetic

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

On devices of compute capability 2.0, 32-bit integer multiplication is natively supported, but 24-bit integer multiplication is not. `[u]mul24` is therefore implemented using multiple instructions and should not be used.

Integer division and modulo operation are costly: tens of instructions on devices of compute capability 1.x, below 20 instructions on devices of compute capability 2.0. They can be replaced with bitwise operations in some cases: If `n` is a power of 2, `(i/n)` is equivalent to `(i>>log2(n))` and `(i%n)` is equivalent to `(i&(n-1))`; the compiler will perform these conversions if `n` is literal.

Type Conversion

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

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

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

3.4.2 Control Flow Instructions

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

To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps. This is possible because the distribution of the warps across the block is deterministic as mentioned in Section 2.1.1. A trivial example is when the controlling condition only depends on (**get_local_id(0) / 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. The programmer can also control loop unrolling using the **#pragma unroll** directive (see **cl_nv_pragma_unroll** extension).

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.

3.4.3 Synchronization Instruction

Throughput for **barrier()** is 8 operations per clock cycle for devices of compute capability 1.x and 16 operations per clock cycle for devices of compute capability 2.0.

Note that **barrier()** can impact performance by forcing the multiprocessor to idle as detailed in Section 3.2.3.

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

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

```
// myArray is an array of integers located in global or shared
// memory
__kernel void myKernel(__global int* result) {
    int tid = get_local_id(0);
    ...
    int ref1 = myArray[tid] * 1;
    barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);
    myArray[tid + 1] = 2;
    barrier(CLK_LOCAL_MEM_FENCE|CLK_GLOBAL_MEM_FENCE);
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
    ...
}
```

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

```
// myArray is an array of integers located in global or shared
// memory
__kernel void myKernel(__global int* result) {
    int tid = get_local_id(0);
    ...
    if (tid < warpSize) {
        int ref1 = myArray[tid] * 1;
        myArray[tid + 1] = 2;
        int ref2 = myArray[tid] * 1;
        result[tid] = ref1 * ref2;
    }
    ...
}
```

Simply removing the **barrier()** is not enough however; **myArray** must also be declared as volatile as described in Section 2.2.2.

Appendix A. CUDA-Enabled GPUs

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

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

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

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GeForce GTX 480	2.0	15	480
GeForce GTX 470	2.0	14	448
GeForce GTX 295	1.3	2x30	2x240
GeForce GTX 285, GTX 280	1.3	30	240
GeForce GTX 260	1.3	24	192
GeForce 9800 GX2	1.1	2x16	2x128
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512, GTX 285M, GTX 280M	1.1	16	128
GeForce 8800 Ultra, 8800 GTX	1.0	16	128
GeForce 9800 GT, 8800 GT, GTX 260M, 9800M GTX	1.1	14	112
GeForce GT 240, GTS 360M, GTS 350M	1.2	12	96
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTS 260M, GTS 250M, 9800M GT	1.1	12	96
GeForce 8800 GTS	1.0	12	96
GeForce GT 335M	1.2	9	72
GeForce 9600 GT, 8800M GTS, 9800M GTS	1.1	8	64
GeForce GT 220, GT 330M, GT 325M	1.2	6	48
GeForce 9700M GT, GT 240M,	1.1	6	48

	Compute Capability	Number of Multiprocessors	Number of CUDA Cores
GT 230M			
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	1.1	4	32
GeForce 210, 310M, 305M	1.2	2	16
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU, G210M, G110M	1.1	2	16
GeForce 9300M GS, 9200M GS, 9100M G, 8400M G, G105M	1.1	1	8
Tesla C2050	2.0	14	448
Tesla S1070	1.3	4x30	4x240
Tesla C1060	1.3	30	240
Tesla S870	1.0	4x16	4x128
Tesla D870	1.0	2x16	2x128
Tesla C870	1.0	16	128
Quadro Plex 2200 D2	1.3	2x30	2x240
Quadro Plex 2100 D4	1.1	4x14	4x112
Quadro Plex 2100 Model S4	1.0	4x16	4x128
Quadro Plex 1000 Model IV	1.0	2x16	2x128
Quadro FX 5800	1.3	30	240
Quadro FX 4800	1.3	24	192
Quadro FX 4700 X2	1.1	2x14	2x112
Quadro FX 3700M, FX 3800M	1.1	16	128
Quadro FX 5600	1.0	16	128
Quadro FX 3700	1.1	14	112
Quadro FX 2800M	1.1	12	96
Quadro FX 4600	1.0	12	96
Quadro FX 1800M	1.2	9	72
Quadro FX 3600M	1.1	8	64
Quadro FX 880M, NVS 5100M	1.2	6	48
Quadro FX 2700M	1.1	6	48
Quadro FX 1700, FX 570, NVS 320M, FX 1700M, FX 1600M, FX 770M, FX 570M	1.1	4	32
Quadro FX 380 LP, FX 380M, NVS 3100M, NVS 2100M	1.2	2	16
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	1.1	2	16
Quadro FX 370M, NVS 130M	1.1	1	8

Appendix B. Mathematical Functions Accuracy

B.1 Standard Functions

Error bounds in this section are generated from extensive but not exhaustive tests, so they are not guaranteed bounds.

B.1.1 Single-Precision Floating-Point Functions

Table C-1 lists errors for the standard single-precision floating-point functions.

The recommended way to round a single-precision floating-point operand to an integer, with the result being a single-precision floating-point number is **`rint()`**, not **`round()`**. The reason is that **`round()`** maps to an 8-instruction sequence on the device, whereas **`rint()`** maps to a single instruction. **`trunc()`**, **`ceil()`**, and **`floor()`** each map to a single instruction as well.

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

The maximum error is stated as the absolute value of the difference in ulps between a correctly rounded single-precision result and the result returned by the CUDA library function.

Function	Maximum ulp error
`x+y`	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when addition is merged into an FMAD)
`x*y`	0 (IEEE-754 round-to-nearest-even) (except for devices of compute capability 1.x when multiplication is merged into an FMAD)
`x/y`	0 for compute capability ≥ 2 2 (full range), otherwise
`1/x`	0 for compute capability ≥ 2 1 (full range), otherwise
`rsqrt(x)` `1/sqrt(x)`	2 (full range) Applies to `1/sqrt(x)` only when it is converted to `rsqrt(x)` by the compiler.
`sqrt(x)`	0 for compute capability ≥ 2

Function	Maximum ulp error
	3 (full range), otherwise
`cbrt(x)`	1 (full range)
`hypot(x,y)`	3 (full range)
`exp(x)`	2 (full range)
`exp2(x)`	2 (full range)
`exp10(x)`	2 (full range)
`expm1(x)`	1 (full range)
`log(x)`	1 (full range)
`log2(x)`	3 (full range)
`log10(x)`	3 (full range)
`log1p(x)`	2 (full range)
`sin(x)`	2 (full range)
`cos(x)`	2 (full range)
`tan(x)`	4 (full range)
`sincos(x,cptr)`	2 (full range)
`asin(x)`	4 (full range)
`acos(x)`	3 (full range)
`atan(x)`	2 (full range)
`atan2(y,x)`	3 (full range)
`sinh(x)`	3 (full range)
`cosh(x)`	2 (full range)
`tanh(x)`	2 (full range)
`asinh(x)`	3 (full range)
`acosh(x)`	4 (full range)
`atanh(x)`	3 (full range)
`pow(x,y)`	8 (full range)
`erf(x)`	3 (full range)
`erfc(x)`	6 (full range)
`lgamma(x)`	6 (outside interval -10.001 ... -2.264; larger inside)
`tgamma(x)`	11 (full range)
`fma(x,y,z)`	0 (full range)
`frexp(x,exp)`	0 (full range)
`ldexp(x,exp)`	0 (full range)
`scalbn(x,n)`	0 (full range)
`scalbln(x,l)`	0 (full range)
`logb(x)`	0 (full range)
`ilogb(x)`	0 (full range)
`fmod(x,y)`	0 (full range)
`remainder(x,y)`	0 (full range)
`remquo(x,y,iptr)`	0 (full range)
`modf(x,iptr)`	0 (full range)
`fdim(x,y)`	0 (full range)

Function	Maximum ulp error
`trunc(x)`	0 (full range)
`round(x)`	0 (full range)
`rint(x)`	0 (full range)
`nearbyint(x)`	0 (full range)
`ceil(x)`	0 (full range)
`floor(x)`	0 (full range)
`lrint(x)`	0 (full range)
`lround(x)`	0 (full range)
`llrint(x)`	0 (full range)
`llround(x)`	0 (full range)

B.1.2 Double-Precision Floating-Point Functions

Table C-2 lists errors for the standard double-precision floating-point functions.

These errors only apply when compiling for devices with native double-precision support. When compiling for devices without such support, such as devices of compute capability 1.2 and lower, the **`double`** type gets demoted to **`float`** by default and the double-precision math functions are mapped to their single-precision equivalents.

The recommended way to round a double-precision floating-point operand to an integer, with the result being a double-precision floating-point number is **`rint()`**, not **`round()`**. The reason is that **`round()`** maps to an 8-instruction sequence on the device, whereas **`rint()`** maps to a single instruction. **`trunc()`**, **`ceil()`**, and **`floor()`** each map to a single instruction as well.

Table C-2. Mathematical Standard Library Functions with Maximum ULP Error

The maximum error is stated as the absolute value of the difference in ulps between a correctly rounded double-precision result and the result returned by the CUDA library function.

Function	Maximum ulp error
`x+y`	0 (IEEE-754 round-to-nearest-even)
`x*y`	0 (IEEE-754 round-to-nearest-even)
`x/y`	0 (IEEE-754 round-to-nearest-even)
`1/x`	0 (IEEE-754 round-to-nearest-even)
`sqrt(x)`	0 (IEEE-754 round-to-nearest-even)
`rsqrt(x)`	1 (full range)
`cbrt(x)`	1 (full range)
`hypot(x,y)`	2 (full range)
`exp(x)`	1 (full range)
`exp2(x)`	1 (full range)
`exp10(x)`	1 (full range)
`expm1(x)`	1 (full range)

Function	Maximum ulp error
`log(x)`	1 (full range)
`log2(x)`	1 (full range)
`log10(x)`	1 (full range)
`log1p(x)`	1 (full range)
`sin(x)`	2 (full range)
`cos(x)`	2 (full range)
`tan(x)`	2 (full range)
`sincos(x,sptr,cptr)`	2 (full range)
`asin(x)`	2 (full range)
`acos(x)`	2 (full range)
`atan(x)`	2 (full range)
`atan2(y,x)`	2 (full range)
`sinh(x)`	1 (full range)
`cosh(x)`	1 (full range)
`tanh(x)`	1 (full range)
`asinh(x)`	2 (full range)
`acosh(x)`	2 (full range)
`atanh(x)`	2 (full range)
`pow(x,y)`	2 (full range)
`erf(x)`	2 (full range)
`erfc(x)`	4 (full range)
`lgamma(x)`	4 (outside interval -11.0001 ... -2.2637; larger inside)
`tgamma(x)`	8 (full range)
`fma(x,y,z)`	0 (IEEE-754 round-to-nearest-even)
`frexp(x,exp)`	0 (full range)
`ldexp(x,exp)`	0 (full range)
`scalbn(x,n)`	0 (full range)
`scalbln(x,l)`	0 (full range)
`logb(x)`	0 (full range)
`ilogb(x)`	0 (full range)
`fmod(x,y)`	0 (full range)
`remainder(x,y)`	0 (full range)
`remquo(x,y,iptr)`	0 (full range)
`modf(x,iptr)`	0 (full range)
`fdim(x,y)`	0 (full range)
`trunc(x)`	0 (full range)
`round(x)`	0 (full range)
`rint(x)`	0 (full range)
`nearbyint(x)`	0 (full range)
`ceil(x)`	0 (full range)
`floor(x)`	0 (full range)
`lrint(x)`	0 (full range)

Function	Maximum ulp error
`lround(x)`	0 (full range)
`llrint(x)`	0 (full range)
`llround(x)`	0 (full range)

B.2 Native Functions

Table C-3 lists the native single-precision floating-point functions supported on the CUDA architecture.

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

Table C-3. Single-Precision Floating-Point Native Functions with Respective Error Bounds

Function	Error bounds
`native_recip(x)`	IEEE-compliant.
`native_sqrt(x)`	The maximum ulp error is 2.67.
`native_divide(x,y)`	For **`y`** in $[2^{-126}, 2^{126}]$, the maximum ulp error is 2.
`native_exp(x)`	The maximum ulp error is `2 + floor(abs(1.16 * x))`.
`native_exp10(x)`	The maximum ulp error is `2 + floor(abs(2.95 * x))`.
`native_log(x)`	For **`x`** in [0.5, 2], the maximum absolute error is $2^{-21.41}$, otherwise, the maximum ulp error is 3.
`native_log2(x)`	For **`x`** in [0.5, 2], the maximum absolute error is 2^{-22}, otherwise, the maximum ulp error is 2.
`native_log10(x)`	For **`x`** in [0.5, 2], the maximum absolute error is 2^{-24}, otherwise, the maximum ulp error is 3.
`native_sin(x)`	For **`x`** in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.41}$, and larger otherwise.
`native_cos(x)`	For **`x`** in $[-\pi, \pi]$, the maximum absolute error is $2^{-21.19}$, and larger otherwise.
`native_tan(x)`	Derived from its implementation as `native_sin(x) * (1 / native_cos(x))`.
`native_pow(x,y)`	Derived from its implementation as `exp2(y * native_log2(x))`.

Appendix C. Compute Capabilities

The general specifications and features of a compute device depend on its compute capability (see Section 2.3).

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

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

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

C.1 Features and Technical Specifications

<table>
<tr><th></th><th colspan="5">Compute Capability</th></tr>
<tr><th>Extension Support</th><th>1.0</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.0</th></tr>
<tr><td>cl_khr_byte_addressable_store</td><td colspan="5" rowspan="9">Yes</td></tr>
<tr><td>cl_khr_icd</td></tr>
<tr><td>cl_nv_compiler_options</td></tr>
<tr><td>cl_nv_device_attribute_query</td></tr>
<tr><td>cl_nv_pragma_unroll</td></tr>
<tr><td>cl_khr_gl_sharing</td></tr>
<tr><td>cl_nv_d3d9_sharing</td></tr>
<tr><td>cl_nv_d3d10_sharing</td></tr>
<tr><td>cl_khr_d3d10_sharing</td></tr>
<tr><td>cl_nv_d3d11_sharing</td><td colspan="5"></td></tr>
<tr><td>cl_khr_global_int32_base_atomics</td><td rowspan="2">No</td><td colspan="4" rowspan="2">Yes</td></tr>
<tr><td>cl_khr_global_int32_extended_atomics</td></tr>
<tr><td>cl_khr_local_int32_base_atomics</td><td colspan="2" rowspan="2">No</td><td colspan="3" rowspan="2">Yes</td></tr>
<tr><td>cl_khr_local_int32_extended_atomics</td></tr>
<tr><td>cl_khr_fp64</td><td colspan="3">No</td><td colspan="2">Yes</td></tr>
</table>

	Compute Capability				
Technical Specifications	**1.0**	**1.1**	**1.2**	**1.3**	**2.0**
Maximum x- or y-dimension of a grid of thread blocks	65535				
Maximum number of threads per block	512				1024
Maximum x- or y-dimension of a block	512				1024
Maximum z-dimension of a block	64				
Warp size	32				
Maximum number of resident blocks per multiprocessor	8				
Maximum number of resident warps per multiprocessor	24		32		48
Maximum number of resident threads per multiprocessor	768		1024		1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K
Maximum amount of shared memory per multiprocessor	16 KB				48 KB
Number of shared memory banks	16				32
Amount of local memory per thread	16 KB				512 KB
Constant memory size	64 KB				
Cache working set per multiprocessor for constant memory	8 KB				
Cache working set per multiprocessor for texture memory	Device dependent, between 6 KB and 8 KB				
Maximum width for a 1D texture reference bound to a CUDA array	8192				32768
Maximum width for a 1D texture reference bound to linear memory	2^{27}				
Maximum width and height for a 2D texture reference bound to linear memory or a CUDA array	65536 x 32768				65536 x 65536
Maximum width, height, and depth for a 3D texture reference bound to linear memory or a CUDA array	2048 x 2048 x 2048				4096 x 4096 x 4096
Maximum number of textures that can be bound to a kernel	128				
Maximum number of instructions per kernel	2 million				

C.2 Floating-Point Standard

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

- There is no dynamically configurable rounding mode; however, most of the operations support multiple IEEE rounding modes, exposed via device intrinsics;

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

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

    - **`fma(float, float, float)`**: single-precision fused multiply-add with IEEE rounding modes,
    - **`native_recip(float)`**: single-precision reciprocal with IEEE rounding modes,
    - **`native_divide(float, float)`**: single-precision division with IEEE rounding modes,
- For **double-precision** floating-point numbers on devices of **compute capability 1.x**:
  - Round-to-nearest-even is the only supported IEEE rounding mode for reciprocal, division, and square root.

When compiling for devices without native double-precision floating-point support, i.e. devices of compute capability 1.2 and lower, each **`double`** variable is converted to single-precision floating-point format (but retains its size of 64 bits) and double-precision floating-point arithmetic gets demoted to single-precision floating-point arithmetic.

Addition and multiplication are often combined into a single multiply-add instruction:

- FMAD for single precision on devices of compute capability 1.x,
- FFMA for single precision on devices of compute capability 2.0.

As mentioned above, FMAD truncates the mantissa prior to use it in the addition. FFMA, on the other hand, is an IEEE-754(2008) compliant fused multiply-add instruction, so the full-width product is being used in the addition and a single rounding occurs during generation of the final result. While FFMA in general has superior numerical properties compared to FMAD, the switch from FMAD to FFMA can cause slight changes in numeric results and can in rare circumstances lead to slighty larger error in final results.

In accordance to the IEEE-754R standard, if one of the input parameters to **`fmin()`** or **`fmax()`** is NaN, but not the other, the result is the non-NaN parameter.

The conversion of a floating-point value to an integer value in the case where the floating-point value falls outside the range of the integer format is left undefined by IEEE-754. For compute devices, the behavior is to clamp to the end of the supported range. This is unlike the x86 architecture behavior.

C.3 Compute Capability 1.x

C.3.1 Architecture

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

- 8 CUDA cores for integer and single-precision floating-point arithmetic operations,
- 1 double-precision floating-point unit for double-precision floating-point arithmetic operations,
- 2 special function units for single-precision floating-point transcendental functions (these units can also handle single-precision floating-point multiplications),
- 1 warp scheduler.

To execute an instruction for all threads of a warp, the warp scheduler must therefore issue the instruction over:

- 4 clock cycles for an integer or single-precision floating-point arithmetic instruction,
- 32 clock cycles for a double-precision floating-point arithmetic instruction,
- 16 clock cycles for a single-precision floating-point transcendental instruction.

A multiprocessor also has a read-only constant cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory.

Multiprocessors are grouped into *Texture Processor Clusters* (*TPCs*). The number of multiprocessors per TPC is:

- 2 for devices of compute capabilities 1.0 and 1.1,
- 3 for devices of compute capabilities 1.2 and 1.3.

Each TPC has a read-only texture cache that is shared by all multiprocessors and speeds up reads from the texture memory space, which resides in device memory. Each multiprocessor accesses the texture cache via a texture unit that implements the various addressing modes and data filtering mentioned.

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

C.3.2 Global Memory

A global memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. Sections C.3.2.1 and C.3.2.2 describe how the memory accesses of threads within a half-warp are *coalesced* into one or more memory transactions depending on the compute capability of the device. Figure C-1 shows some examples of global memory accesses and corresponding memory transactions based on compute capability.

The resulting memory transactions are serviced at the throughput of device memory.

C.3.2.1 Devices of Compute Capability 1.0 and 1.1

To coalesce, the memory request for a half-warp must satisfy the following conditions:

- The size of the words accessed by the threads must be 4, 8, or 16 bytes;
- If this size is:
  - 4, all 16 words must lie in the same 64-byte segment,
  - 8, all 16 words must lie in the same 128-byte segment,
  - 16, the first 8 words must lie in the same 128-byte segment and the last 8 words in the following 128-byte segment;
- Threads must access the words in sequence: The k^{th} thread in the half-warp must access the k^{th} word.

If the half-warp meets these requirements, a 64-byte memory transaction, a 128-byte memory transaction, or two 128-byte memory transactions are issued if the size of the words accessed by the threads is 4, 8, or 16, respectively. Coalescing is achieved even if the warp is divergent, i.e. there are some inactive threads that do not actually access memory.

If the half-warp does not meet these requirements, 16 separate 32-byte memory transactions are issued.

C.3.2.2 Devices of Compute Capability 1.2 and 1.3

Threads can access any words in any order, including the same words, and a single memory transaction for each segment addressed by the half-warp is issued. This is in contrast with devices of compute capabilities 1.0 and 1.1 where threads need to access words in sequence and coalescing only happens if the half-warp addresses a single segment.

More precisely, the following protocol is used to determine the memory transactions necessary to service all threads in a half-warp:

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

C.3.3 Shared Memory

Shared memory has 16 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e. interleaved. Each bank has a bandwidth of 32 bits per two clock cycles.

A shared memory request for a warp is split into two memory requests, one for each half-warp, that are issued independently. As a consequence, there can be no bank conflict between a thread belonging to the first half of a warp and a thread belonging to the second half of the same warp.

If a non-atomic instruction executed by a warp writes to the same location in shared memory for more than one of the threads of the warp, only one thread per half-warp performs a write and which thread performs the final write is undefined.

C.3.3.1 32-Bit Strided Access

A common access pattern is for each thread to access a 32-bit word from an array indexed by the thread ID **`tid`** and with some stride `s`:

```
__local float shared[32];
float data = shared[BaseIndex + s * tid];
```

In this case, threads **`tid`** and **`tid+n`** access the same bank whenever **`s*n`** is a multiple of the number of banks (i.e. 16) or, equivalently, whenever **`n`** is a multiple of **`16/d`** where **`d`** is the greatest common divisor of 16 and **`s`**. As a consequence, there will be no bank conflict only if half the warp size (i.e. 16) is less than or equal to **`16/d`**., that is only if **`d`** is equal to 1, i.e. **`s`** is odd.

Figure C-2 shows some examples of strided access for devices of compute capability 2.0. The same examples apply for devices of compute capability 1.x, but with 16 banks instead of 32.

C.3.3.2 32-Bit Broadcast Access

Shared memory features a broadcast mechanism whereby a 32-bit word can be read and broadcast to several threads simultaneously when servicing one memory read

request. This reduces the number of bank conflicts when several threads read from an address within the same 32-bit word. More precisely, a memory read request made of several addresses is serviced in several steps over time by servicing one conflict-free subset of these addresses per step until all addresses have been serviced; at each step, the subset is built from the remaining addresses that have yet to be serviced using the following procedure:

- Select one of the words pointed to by the remaining addresses as the broadcast word;
- Include in the subset:
  - All addresses that are within the broadcast word,
  - One address for each bank (other than the broadcasting bank) pointed to by the remaining addresses.

Which word is selected as the broadcast word and which address is picked up for each bank at each cycle are unspecified.

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

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

C.3.3.3 8-Bit and 16-Bit Access

8-bit and 16-bit accesses typically generate bank conflicts. For example, there are bank conflicts if an array of **char** is accessed the following way:

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

because **shared[0]**, **shared[1]**, **shared[2]**, and **shared[3]**, for example, belong to the same bank. There are no bank conflicts however, if the same array is accessed the following way:

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

C.3.3.4 Larger Than 32-Bit Access

Accesses that are larger than 32-bit per thread are split into 32-bit accesses that typically generate bank conflicts.

For example, there are 2-way bank conflicts for arrays of **double**s accessed as follows:

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

as the memory request is compiled into two separate 32-bit requests with a stride of two. One way to avoid bank conflicts in this case is two split the **double** operands like in the following sample code:

```
__local int shared_lo[32];
__local int shared_hi[32];
int2 tmp;

double dataIn;
tmp = as_int2(dataIn);
shared_lo[BaseIndex + tid] = tmp.x;
```

```
shared_hi[BaseIndex + tid] = tmp.y;

tmp = (int2)(shared_hi[BaseIndex + tid],
             shared_lo[BaseIndex + tid]);
double dataOut = as_double(tmp);
```

This might not always improve performance however and does perform worse on devices of compute capabilities 2.0.

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

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

results in:

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

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

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

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

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

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

C.4 Compute Capability 2.0

C.4.1 Architecture

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

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

At every instruction issue time, each scheduler issues an instruction for some warp that is ready to execute, if any. The first scheduler is in charge of the warps with an odd ID and the second scheduler is in charge of the warps with an even ID. Note that when a scheduler issues a double-precision floating-point instruction, the other scheduler cannot issue any instruction.

A warp scheduler can issue an instruction to only half of the CUDA cores. To execute an instruction for all threads of a warp, a warp scheduler must therefore issue the instruction over:

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

A multiprocessor also has a read-only uniform cache that is shared by all functional units and speeds up reads from the constant memory space, which resides in device memory.

There is an L1 cache for each multiprocessor and an L2 cache shared by all multiprocessors, both of which are used to cache accesses to local or global memory, including temporary register spills.

Multiprocessors are grouped into *Graphics Processor Clusters* (*GPCs*). A GPC includes four multiprocessors.

Each multiprocessor has a read-only texture cache to speed up reads from the texture memory space, which resides in device memory. It accesses the texture cache via a texture unit that implements the various addressing modes and data filtering.

C.4.2 Global Memory

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

If the size of the words accessed by each thread is more than 4 bytes, a memory request by a warp is first split into separate 128-byte memory requests that are issued independently:

- Two memory requests, one for each half-warp, if the size is 8 bytes,
- Four memory requests, one for each quarter-warp, if the size is 16 bytes.

Each memory request is then broken down into cache line requests that are issued independently. A cache line request is serviced at the throughput of L1 or L2 cache in case of a cache hit, or at the throughput of device memory, otherwise.

Note that threads can access any words in any order, including the same words.

If a non-atomic instruction executed by a warp writes to the same location in global memory for more than one of the threads of the warp, only one thread performs a write and which thread does it is undefined.

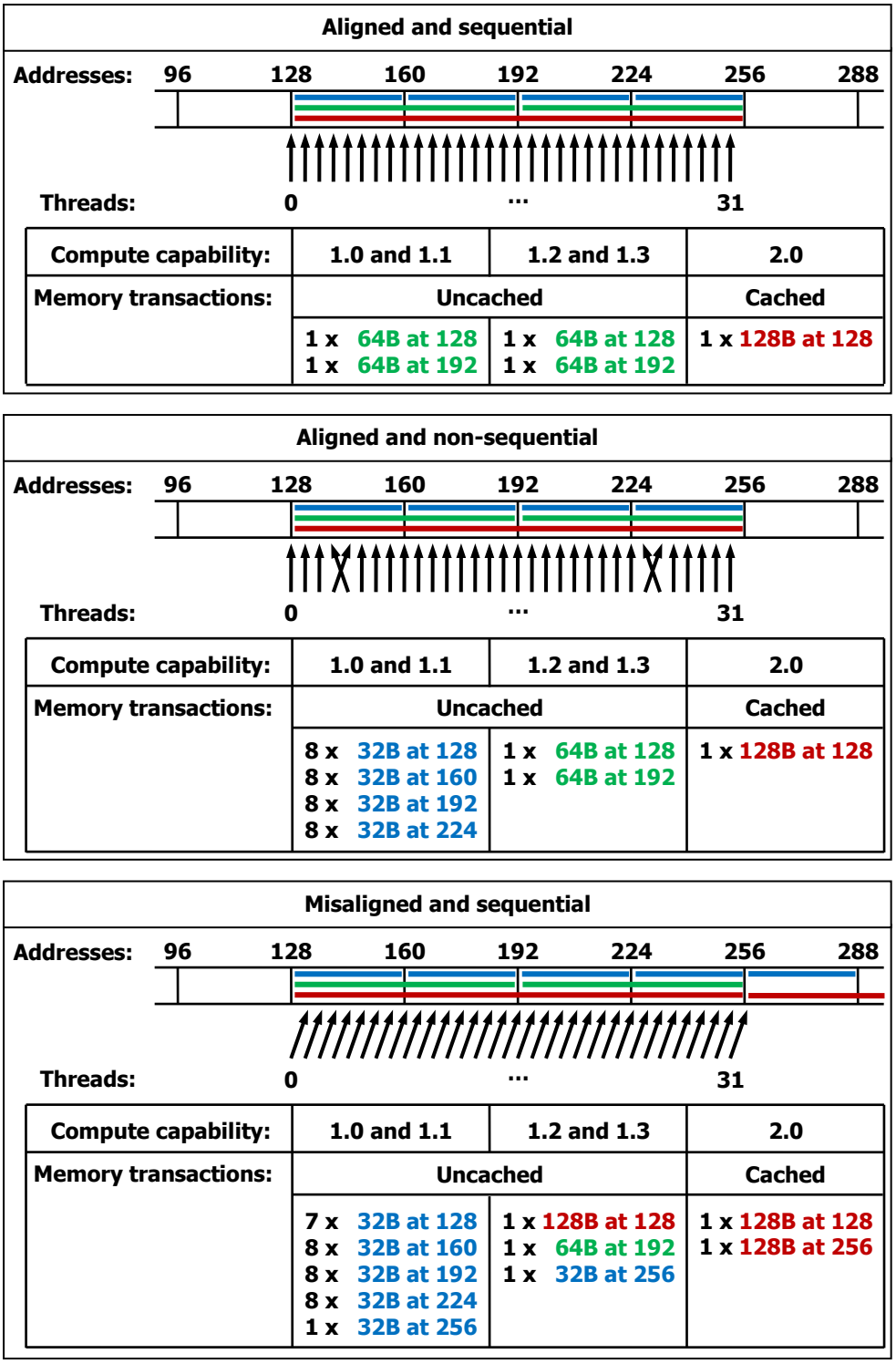


Figure C-1. Examples of Global Memory Accesses by a Warp, 4-Byte Word per Thread, and Associated Memory Transactions Based on Compute Capability

C.4.3 Shared Memory

Shared memory has 32 banks that are organized such that successive 32-bit words are assigned to successive banks, i.e. interleaved. Each bank has a bandwidth of 32 bits per two clock cycles. Therefore, unlike for devices of lower compute capability, there may be bank conflicts between a thread belonging to the first half of a warp and a thread belonging to the second half of the same warp.

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

This means, in particular, that unlike for devices of compute capability 1.x, there are no bank conflicts if an array of **`char`** is accessed as follows, for example:

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

C.4.3.1 32-Bit Strided Access

A common access pattern is for each thread to access a 32-bit word from an array indexed by the thread ID **`tid`** and with some stride `s`:

```
__local float shared[32];
float data = shared[BaseIndex + s * tid];
```

In this case, threads **`tid`** and **`tid+n`** access the same bank whenever **`s*n`** is a multiple of the number of banks (i.e. 32) or, equivalently, whenever **`n`** is a multiple of **`32/d`** where **`d`** is the greatest common divisor of 32 and **`s`**. As a consequence, there will be no bank conflict only if the warp size (i.e. 32) is less than or equal to **`32/d`**., that is only if **`d`** is equal to 1, i.e. **`s`** is odd.

Figure C-2 shows some examples of strided access.

C.4.3.2 Larger Than 32-Bit Access

64-bit and 128-bit accesses are specifically handled to minimize bank conflicts as described below.

Other accesses larger than 32-bit are split into 32-bit, 64-bit, or 128-bit accesses. The following code, for example:

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

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

results in three separate 32-bit reads without bank conflicts since each member is accessed with a stride of three 32-bit words.

64-Bit Accesses

For 64-bit accesses, a bank conflict only occurs if two or more threads in either of the half-warps access different addresses belonging to the same bank.

Unlike for devices of compute capability 1.x, there are no bank conflicts for arrays of **double**s accessed as follows, for example:

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

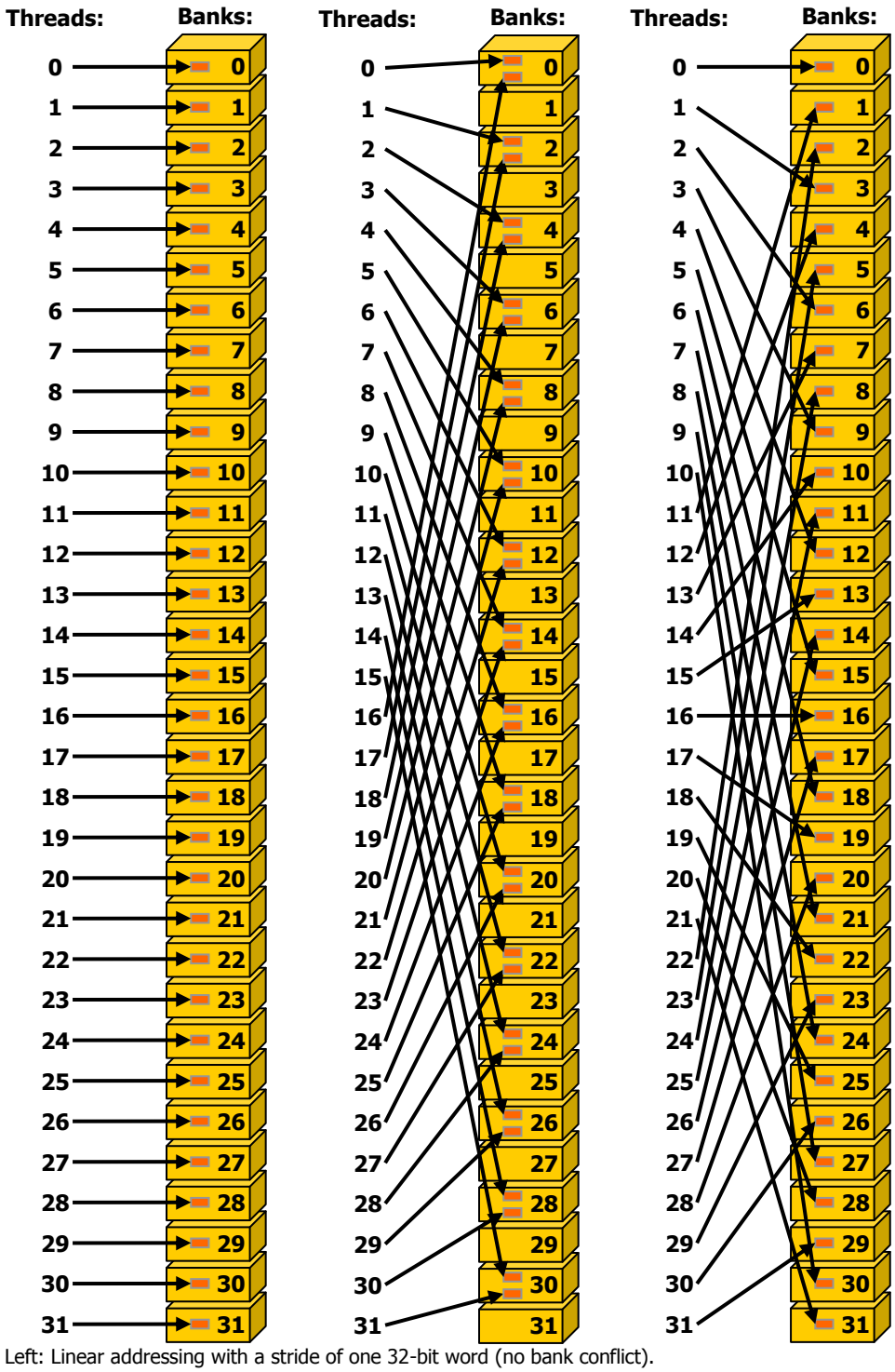
128-Bit Accesses

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

C.4.4 Constant Memory

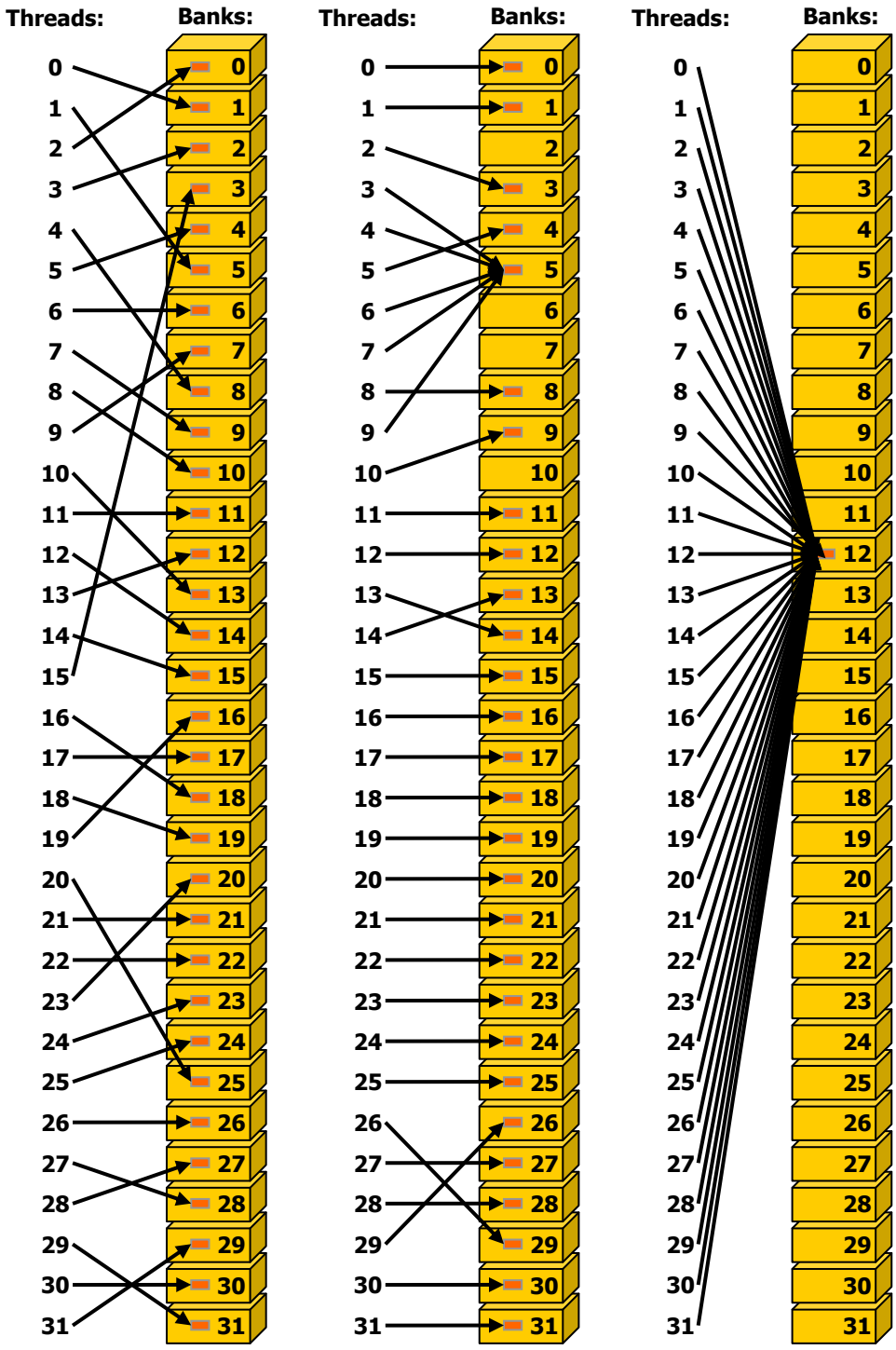
In addition to the constant memory space supported by devices of all compute capabilities (where **__constant** variables reside), devices of compute capability 2.0 support the LDU (LoaD Uniform) instruction that the compiler use to load any variable that is:

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



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

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



Left: Conflict-free access via random permutation.
Middle: Conflict-free access since threads 3, 4, 6, 7, and 9 access the same word within bank 5.
Right: Conflict-free broadcast access (all threads access the same word).

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

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