OpenCL Programming Guide for the CUDA Architecture

Version 2.3

8/27/2009

Table of Contents

List of Figures

Chapter 1. Introduction

1.1 From Graphics Processing to General-Purpose Parallel Computing

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

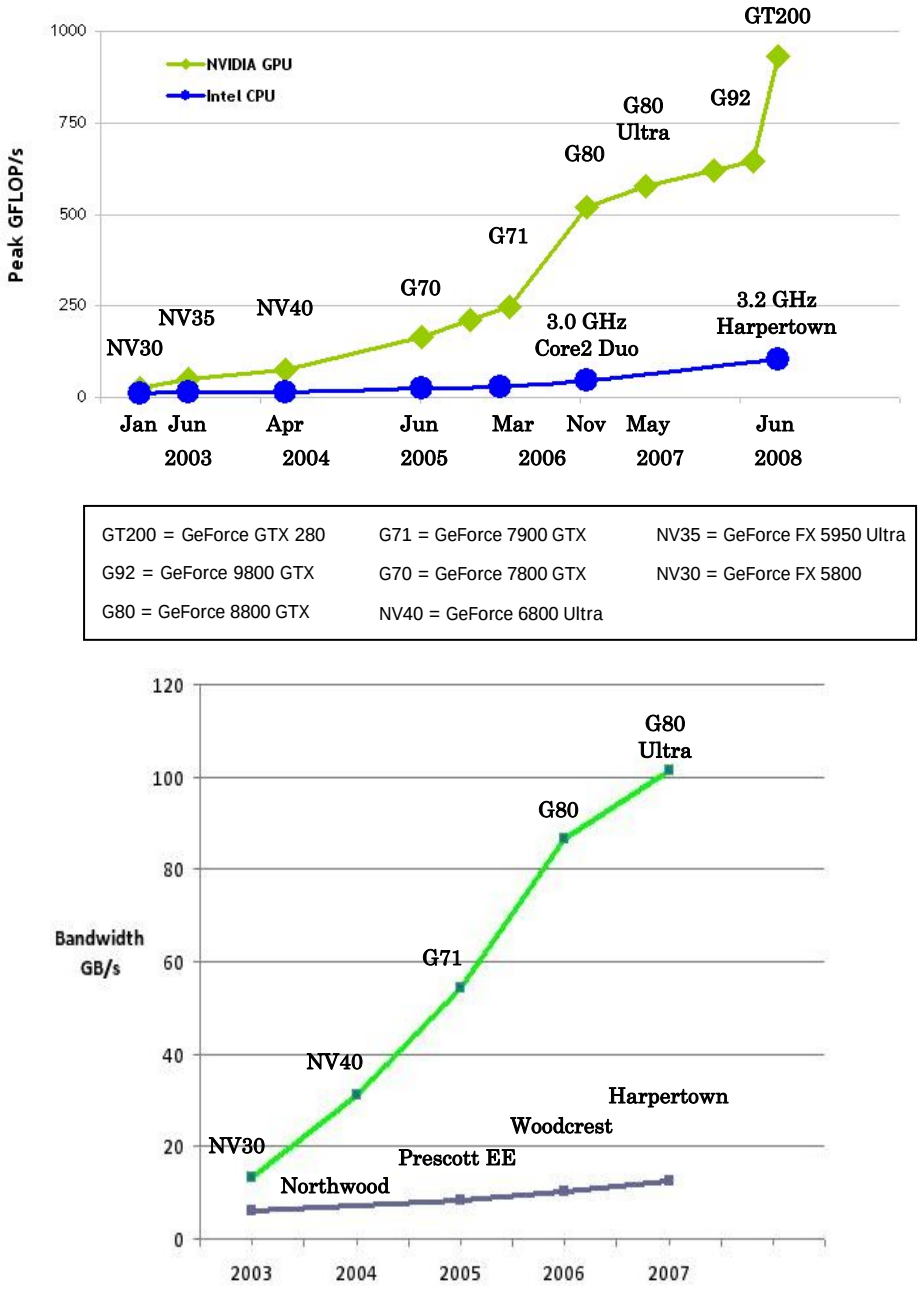


Figure 1-1. Floating-Point Operations per Second and Memory Bandwidth for the CPU and GPU

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

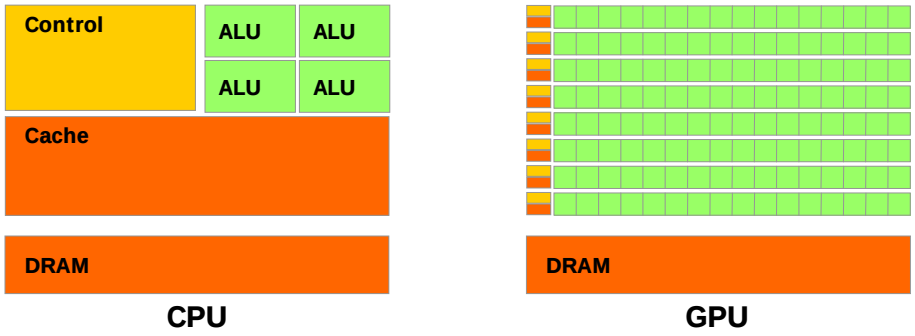


Figure 1-2. The GPU Devotes More Transistors to Data Processing

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

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

1.2 CUDA™: a General-Purpose Parallel Computing Architecture

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

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

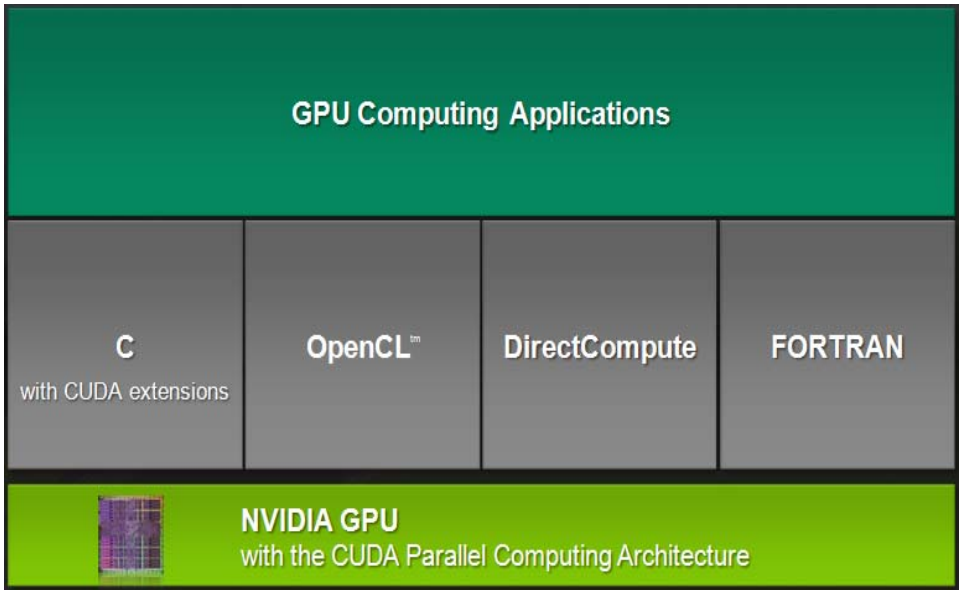


Figure 1-3. CUDA is Designed to Support Various Languages and Application Programming Interfaces

1.3 CUDA's 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.

CUDA's 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, and then into finer pieces that can be solved cooperatively in parallel. Such a decomposition preserves language expressivity by allowing threads to cooperate when solving each sub-problem, and at the same time enables transparent scalability since each sub-problem can be scheduled to be solved on any of the available processor cores: A compiled program can therefore execute on any number of processor cores, 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).

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.

Chapter 2. OpenCL on the CUDA Architecture

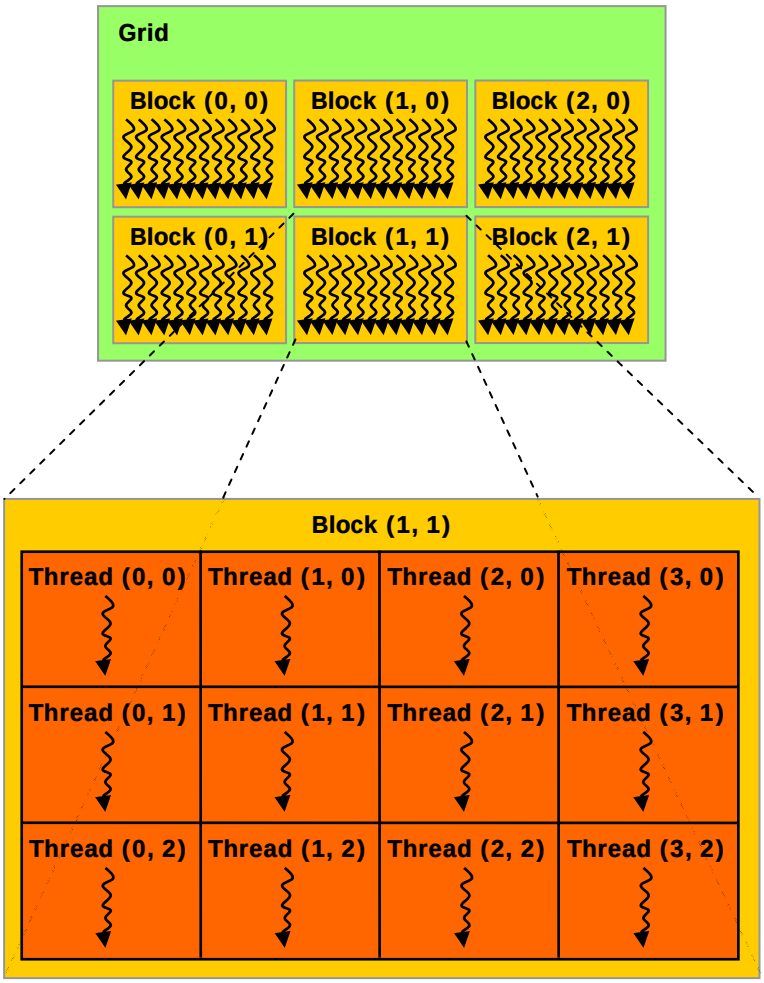
2.1 CUDA Architecture

2.1.1 Execution Model

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.

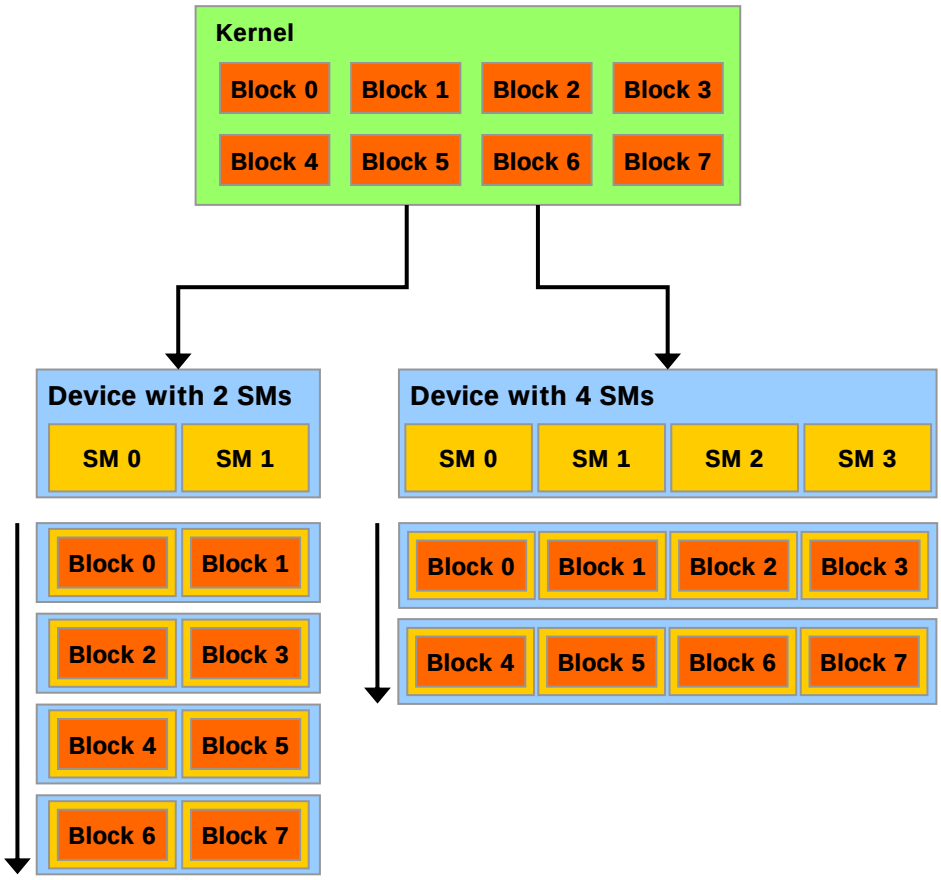
Figure 2-1.Grid of Thread Blocks

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.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling

programmers to write code that scales with the number of cores, as illustrated in Figure 2-2.



A device with more multiprocessors will automatically execute a kernel in less time than a device with fewer multiprocessors.

Figure 2-2. Automatic Scalability

A multiprocessor consists of eight Scalar Processor (SP) cores, two special function units for transcendentals, a multithreaded instruction unit, and on-chip shared memory, which is used to implement OpenCL local memory. The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. It implements the work-group barrier function with a single instruction. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support very fine-grained parallelism, allowing, for example, a low granularity decomposition of problems by assigning one thread to each data element (such as a pixel in an image, a voxel in a volume, a cell in a grid-based computation).

To manage hundreds of threads running several different programs, the multiprocessor employs a new architecture we call *SIMT* (single-instruction, multiple-thread). The multiprocessor maps each thread to one SP, and each scalar

thread executes independently with its own instruction address and register state. The multiprocessor SIMT unit creates, manages, schedules, and executes threads in groups of 32 parallel threads called *warps*. (This term originates from weaving, the first parallel thread technology. A *half-warp* is either the first or second half of a warp.) Individual threads composing a SIMT warp start together at the same program address but are otherwise free to branch and execute independently.

When a multiprocessor is given one or more thread blocks to execute, it splits them into warps that get scheduled by the SIMT unit. The way a block is split into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing the thread of thread ID zero.

Every instruction issue time, the SIMT unit selects a warp that is ready to execute and issues the next instruction to the active threads of the warp. 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 disjointed code paths.

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.

2.1.2 Memory Model

As illustrated by Figure 2-3, each multiprocessor has on-chip memory of the four following types:

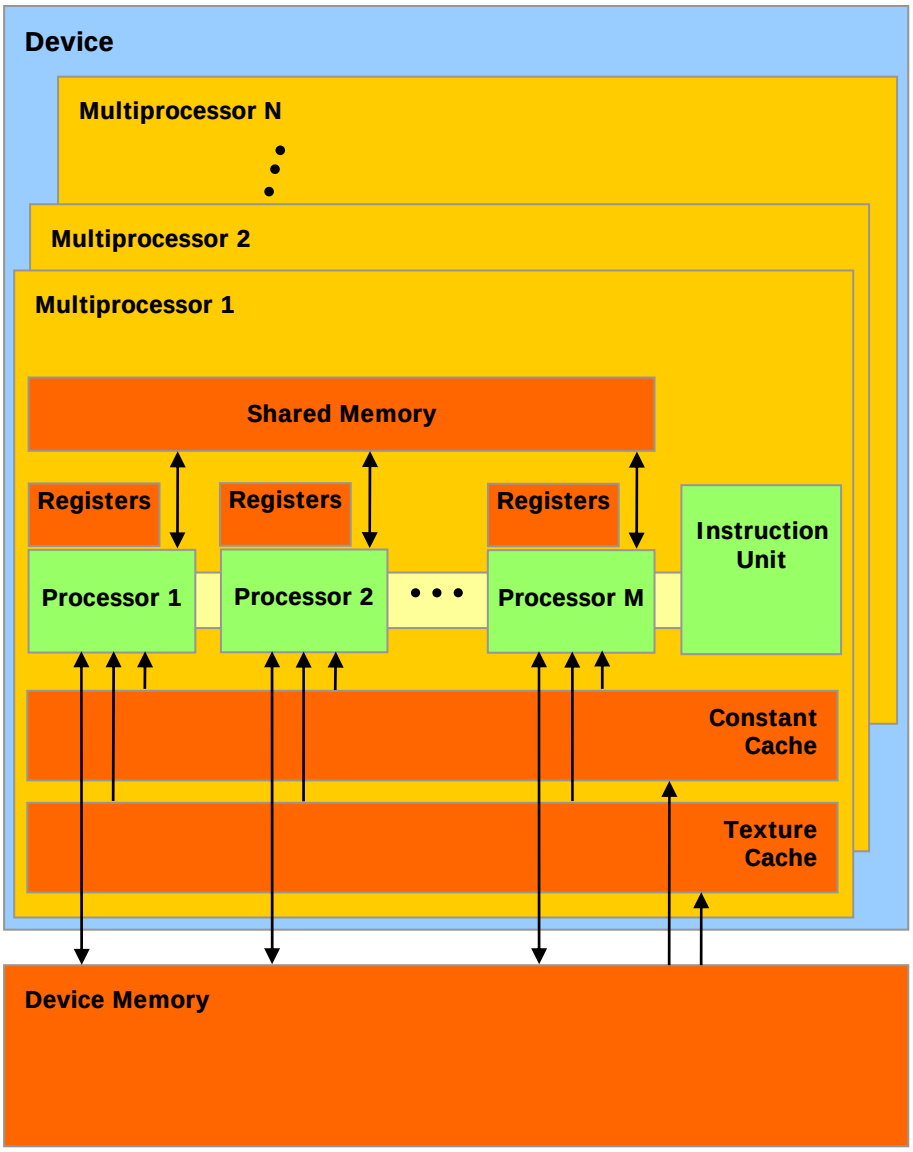
- One set of local 32-bit *registers* per processor,
- A parallel data cache or *shared memory* that is shared by all scalar processor cores and is where OpenCL local memory resides,
- A read-only *constant cache* that is shared by all scalar processor cores and speeds up reads from OpenCL constant memory,
- A read-only *texture cache* that is shared by all scalar processor cores and speeds up reads from OpenCL image objects; each multiprocessor accesses the texture cache via a *texture unit* that implements the various addressing modes and data filtering specified by OpenCL sampler objects; the region of device memory addressed by image objects is referred to a *texture memory*.

There is also a global memory address space that is used for OpenCL global memory and a local memory address space that is private to each thread (and should not be confused with OpenCL local memory). Both memory spaces are read-write regions of device memory and are not cached.

A variable in OpenCL private memory generally resides in a register. However in some cases the compiler might choose to place it in CUDA local memory, which can have adverse performance consequences because of local memory high latency and bandwidth (see Section 3.1.2.2). Variables that are likely to be placed in CUDA local memory are large structures or arrays that would consume too much register space, and arrays for which the compiler cannot determine that they are indexed with constant quantities.

The number of blocks a multiprocessor can process at once – referred to as the number of *active* blocks per multiprocessor – depends on how many registers per thread and how much shared memory per block are required for a given kernel since the multiprocessor's registers and shared memory are split among all the threads of the active blocks. 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 maximum number of active blocks per multiprocessor, as well as the maximum number of active warps and maximum number of active threads are given in Appendix A.

If a non-atomic instruction executed by a warp writes to the same location in global or shared memory for more than one of the threads of the warp, the number of serialized writes that occur to that location and the order in which they occur is undefined, but one of the writes is guaranteed to succeed. If an atomic instruction 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.



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

Figure 2-3.CUDA Architecture

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`** and can be passed to

clCreateProgramWithBinary() to create a program object, but 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 as mentioned in Section 3.4.

2.3 Compute Capability

The *compute capability* of a CUDA 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 devices listed in Appendix A 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.

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

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 mentioned in Section 2.1.2 and detailed in Section 3.1.2.5, 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 2-4. *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);
}
```

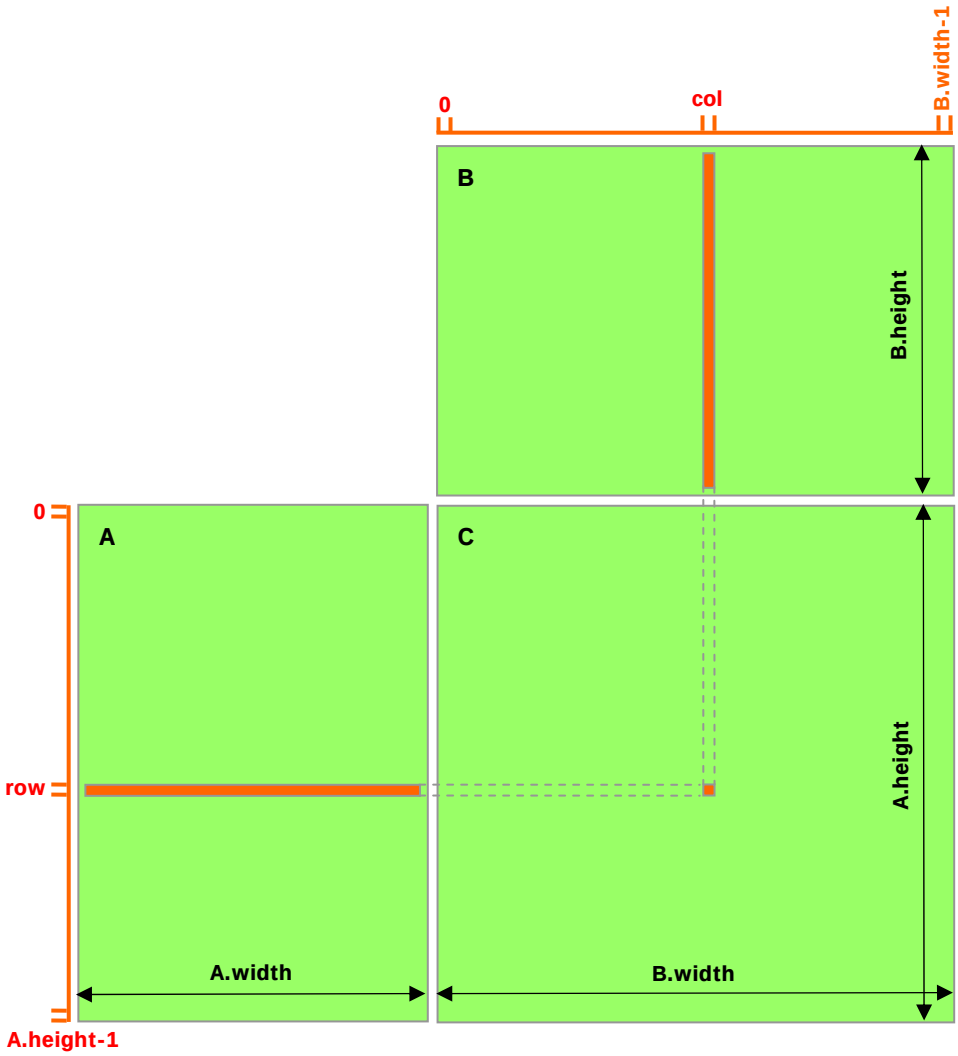


Figure 2-4. Matrix Multipliation without Shared Memory

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 2-5, 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);
}
```

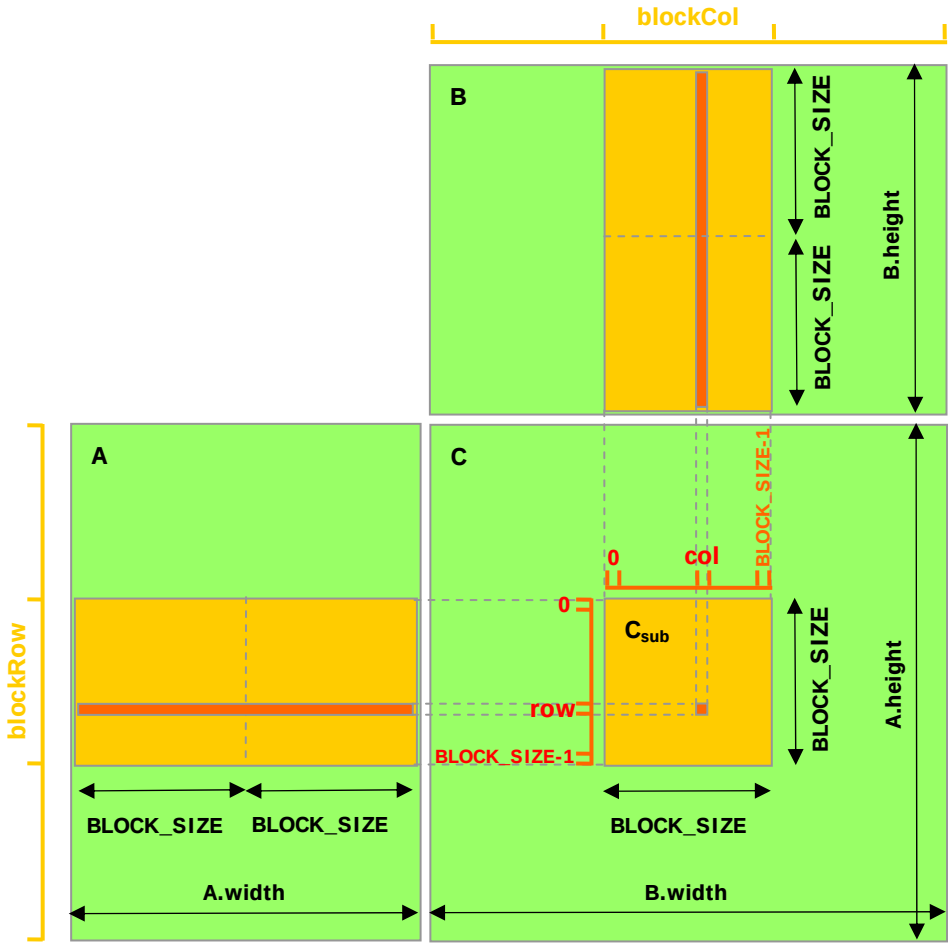


Figure 2-5.Matrix Multipliation with Shared Memory

Chapter 3. Performance Guidelines

3.1 Instruction Performance

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

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

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

- Minimizing the use of instructions with low throughput (see Section 3.1.1),
- Maximizing the use of the available memory bandwidth for each category of memory (see Section 3.1.2),
- Allowing the thread scheduler to overlap memory transactions with mathematical computations as much as possible, which requires that:
  - The program executed by the threads is of high arithmetic intensity, that is, has a high number of arithmetic operations per memory operation;
  - There are many active threads per multiprocessor, as detailed in Section 3.2.

3.1.1 Instruction Throughput

In this section, throughputs are given in number of operations per clock cycle per multiprocessor. For a warp size of 32, an instruction is made of 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.

3.1.1.1 Arithmetic Instructions

For single-precision floating-point code, we highly recommend use of the **`float`** type and the single-precision floating-point mathematical functions. When compiling for devices without native double-precision floating-point support, such as devices of compute capability 1.2 and lower, each **`double`** variable gets 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.

We also recommend using **`native_*`** functions wherever possible and the **`-cl-mad-enable`** build option, both of them can lead to large performance gains.

Single-Precision Floating-Point Basic Arithmetic

Throughput of single-precision floating-point add, multiply, and multiply-add is 8 operations per clock cycle.

Throughput of reciprocal is 2 operations per clock cycle.

Throughput of single-precision floating-point division is 0.88 operations per clock cycle, but **`native_divide(x, y)`** provides a faster version with a throughput of 1.6 operations per clock cycle.

Single-Precision Floating-Point Square Root and Reciprocal Square Root

Throughput of reciprocal square root is 2 operations per clock cycle.

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.

Single-Precision Floating-Point Logarithm

Throughput of **`native_log(x)`** (see Section B.2) is 2 operations per clock cycle.

Sine and Cosine

Throughput of **`native_sin(x)`**, **`native_cos(x)`**, **`native_exp(x)`** is 1 operation per clock cycle.

`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 3.1.2.2). At present, 28 bytes of CUDA 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 CUDA local memory in the slow path, the trigonometric functions throughput is lower by one order of magnitude when the slow path reduction is used as opposed to the fast path reduction.

Integer Arithmetic

Throughput of integer add is 8 operations per clock cycle.

Throughput of 32-bit integer multiplication is 2 operations per clock cycle, but **`mul24`** provide 24-bit integer multiplication with a troughput of 8 operations per clock cycle. On future architectures however, **`mul24`** will be slower than 32-bit integer multiplication, so we recommend to provide two kernels, one using **`mul24`** and the other using generic 32-bit integer multiplication, to be called appropriately by the application.

Integer division and modulo operation are particularly costly and should be avoided if possible or replaced with bitwise operations whenever possible: If **`n`** is a power of 2, **`(i/n)`** is equivalent to **`(i>>log2(n))`** and **`(i%n)`** is equivalent to **`(i&(n-1))`**; the compiler will perform these conversions if **`n`** is literal.

Comparison

Throughput of compare, min, max is 8 operations per clock cycle.

Bitwise Operations

Throughput of any bitwise operation is 8 operations per clock cycle.

Type Conversion

Throughput of type conversion operations is 8 operations per clock cycle.

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

- Functions operating on **`char`** or **`short`** whose operands generally need to be converted to **`int`**,
- Double-precision floating-point constants (defined without any type suffix) used as input to single-precision floating-point computations.

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`**.

3.1.1.2 Control Flow Instructions

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

To obtain best performance in cases where the control flow depends on the thread ID, the controlling condition should be written so as to minimize the number of divergent warps. This is possible because the distribution of the warps across the block is deterministic as mentioned in Section 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.

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.

3.1.1.3 Memory Instructions

Memory instructions include any instruction that reads from or writes to CUDA shared, local, or global memory.

Throughput of memory operations is 8 operations per clock cycle. When accessing CUDA local or global memory, there are, in addition, 400 to 600 clock cycles of memory latency.

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

```
__local float shared[32];
__global float device[32];
shared[threadIdx.x] = device[threadIdx.x];
```

is 8 operations per clock cycle for the read from global memory, 8 operations per clock cycle for the write to shared memory, but above all, there is a latency of 400 to 600 clock cycles to read data from global memory.

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

3.1.1.4 Synchronization Instruction

Throughput for the **`barrier`** function is 8 operations per clock cycle in the case where no thread has to wait for any other threads.

3.1.2 Memory Bandwidth

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

Since device memory is of much higher latency and lower bandwidth than on-chip memory, device memory accesses should be minimized. A typical programming pattern is to stage data coming from device memory into shared memory; in other words, to have each thread of a block:

- Load data from device memory to shared memory,

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

3.1.2.1 Global Memory

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

First, the device is capable of reading 4-byte, 8-byte, or 16-byte words from global memory into registers in a single instruction. To have assignments such as:

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

compile to a single load instruction, **type** must be such that **sizeof(type)** is equal to 4, 8, or 16 and variables of type **type** must be aligned to **sizeof(type)** bytes (that is, have their address be a multiple of **sizeof(type)**).

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)));
```

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

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

which is compiled into two 16-byte load instructions instead of five 4-byte load instructions.

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.

Second, global memory bandwidth is used most efficiently when the simultaneous memory accesses by threads in a half-warp (during the execution of a single read or write instruction) can be *coalesced* into a single memory transaction of 32, 64, or 128 bytes.

The rest of this section describes the various requirements for memory accesses to coalesce based on the compute capability of the device. If a half-warp fulfills these requirements, coalescing is achieved even if the warp is divergent and some threads of the half-warp do not actually access memory.

For the purpose of the following discussion, global memory is considered to be partitioned into segments of size equal to 32, 64, or 128 bytes *and* aligned to this size.

Coalescing on Devices with Compute Capability 1.0 and 1.1

The global memory access by all threads of a half-warp is coalesced into one or two memory transactions if it satisfies the following three conditions:

- Threads must access
  - Either 4-byte words, resulting in one 64-byte memory transaction,
  - Or 8-byte words, resulting in one 128-byte memory transaction,
  - Or 16-byte words, resulting in two 128-byte memory transactions;
- All 16 words must lie in the same segment of size equal to the memory transaction size (or twice the memory transaction size when accessing 16-byte words);
- Threads must access the words in sequence: The k^{th} thread in the half-warp must access the k^{th} word.

If a half-warp does not fulfill all the requirements above, a separate memory transaction is issued for each thread and throughput is significantly reduced.

Figure 3-1 shows some examples of coalesced memory accesses, while Figure 3-2 and Figure 3-3 show some examples of memory accesses that are non-coalesced for devices of compute capability 1.0 or 1.1.

Coalesced 8-byte accesses deliver a little lower bandwidth than coalesced 4-byte accesses and coalesced 16-byte accesses deliver a noticeably lower bandwidth than coalesced 4-byte accesses. But, while bandwidth for non-coalesced accesses is around an order of magnitude lower than for coalesced accesses when these accesses are 4-byte, it is only around four times lower when they are 8-byte and around two times when they are 16-byte.

Coalescing on Devices with Compute Capability 1.2 and Higher

The global memory access by all threads of a half-warp is coalesced into a single memory transaction as soon as the words accessed by all threads lie in the same segment of size equal to:

- 32 bytes if all threads access 1-byte words,
- 64 bytes if all threads access 2-byte words,
- 128 bytes if all threads access 4-byte or 8-byte words.

Coalescing is achieved for any pattern of addresses requested by the half-warp, including patterns where multiple threads access the same address. This is in

contrast with devices of lower compute capabilities where threads need to access words in sequence.

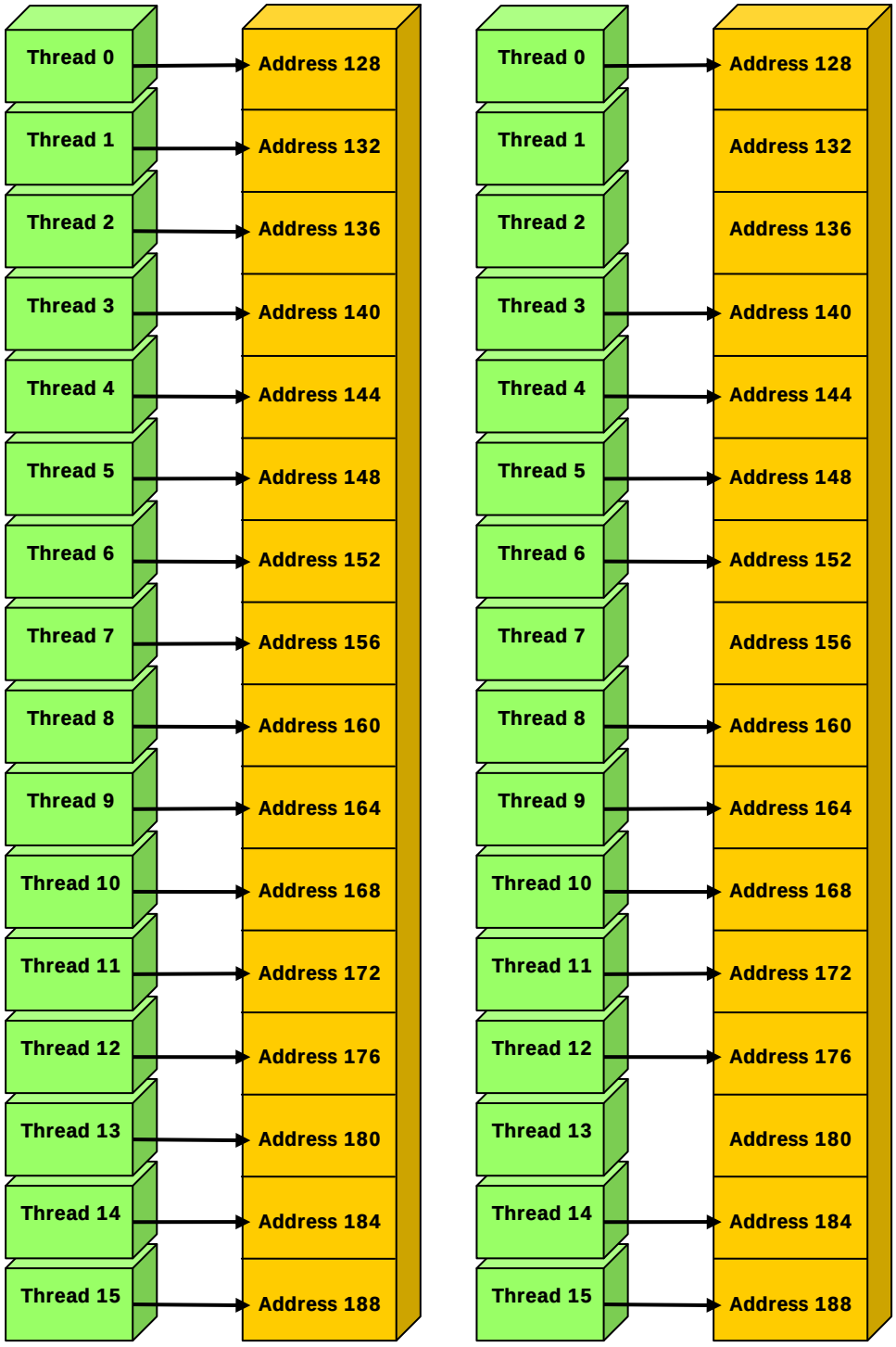
If a half-warp addresses words in *n* different segments, *n* memory transactions are issued (one for each segment), whereas devices with lower compute capabilities would issue 16 transactions as soon as *n* is greater than 1. In particular, if threads access 16-byte words, at least two memory transactions are issued.

Unused words in a memory transaction are still read, so they waste bandwidth. To reduce waste, hardware will automatically issue the smallest memory transaction that contains the requested words. For example, if all the requested words lie in one half of a 128-byte segment, a 64-byte transaction will be issued.

More precisely, the following protocol is used to issue a memory transaction for a half-warp:

- Find the memory segment that contains the address requested by the lowest numbered active thread. Segment size is 32 bytes for 1-byte data, 64 bytes for 2-byte data, 128 bytes for 4-, 8- and 16-byte data.
- 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 and only the lower or upper half is used, reduce the transaction sizez to 32 bytes.
- Carry out the transaction and mark the serviced threads as inactive.
- Repeat until all threads in the half-warp are serviced.

Figure 3-4 shows some examples of global memory accesses for devices of compute capability 1.2 and higher.



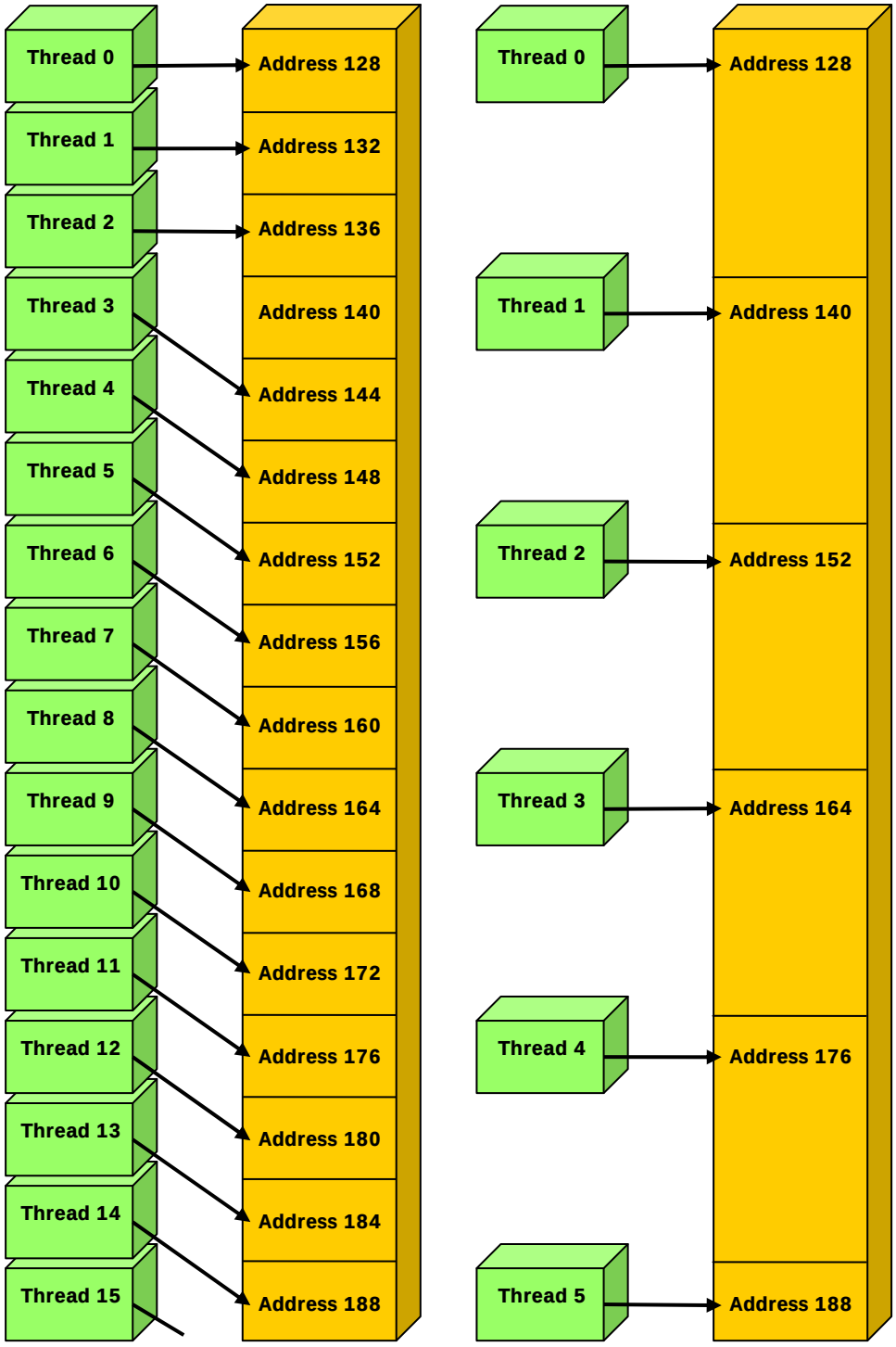
Left: coalesced **`float`** memory access, resulting in a single memory transaction.
Right: coalesced **`float`** memory access (divergent warp), resulting in a single memory transaction.

Figure 3-1. Examples of Coalesced Global Memory Access Patterns

Left: non-sequential `float` memory access, resulting in 16 memory transactions.

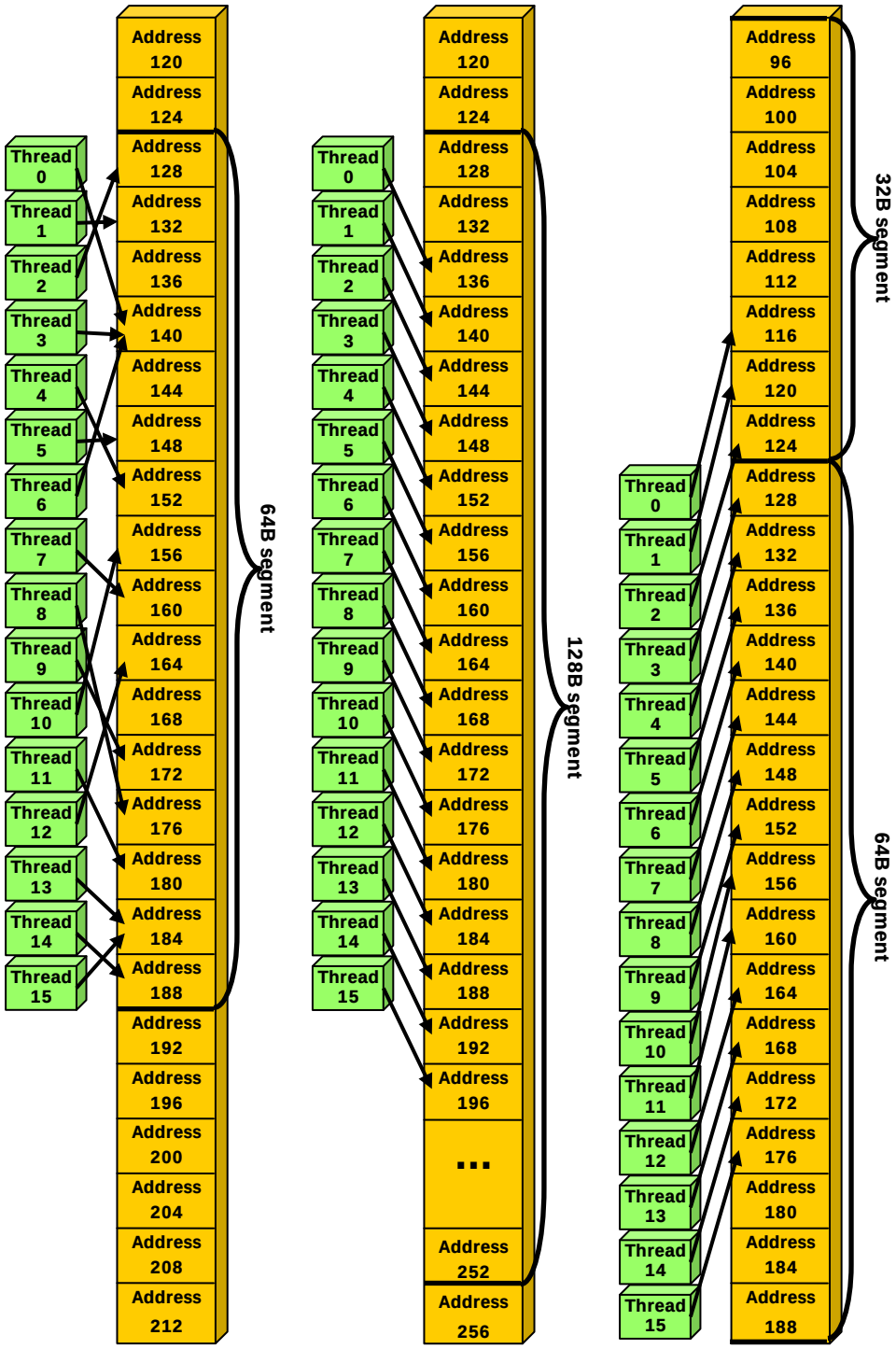
Right: access with a misaligned starting address, resulting in 16 memory transactions.

Figure 3-2. Examples of Global Memory Access Patterns That Are Non-Coalesced for Devices of Compute Capability 1.0 or 1.1



Left: non-contiguous **`float`** memory access, resulting in 16 memory transactions.
Right: non-coalesced **`float3`** memory access, resulting in 16 memory transactions.

Figure 3-3. Examples of Global Memory Access Patterns That Are Non-Coalesced for Devices of Compute Capability 1.0 or 1.1



Left: random **`float`** memory access within a 64B segment, resulting in one memory transaction.
Center: misaligned **`float`** memory access, resulting in one transaction.
Right: misaligned **`float`** memory access, resulting in two transactions.

Figure 3-4. Examples of Global Memory Access by Devices with Compute Capability 1.2 and Higher

Common Access Patterns

Array of Structures

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

```
    BaseAddress + tid
```

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

Two-Dimensional Array

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

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

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

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

In particular, this means that an array whose width is not a multiple of 16 will be accessed much more efficiently if it is actually allocated with a width rounded up to the closest multiple of 16 and its rows padded accordingly.

3.1.2.2 Local Memory

Like global memory, CUDA local memory is not cached, so accesses to local memory are as expensive as accesses to global memory. Local memory accesses are always coalesced though since they are per-thread by definition.

3.1.2.3 Constant Memory

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

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

3.1.2.4 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 3.1.2.1 and 3.1.2.3), higher bandwidth can be achieved providing that there is locality in the image reads;
- The latency of addressing calculations is hidden better, possibly improving performance for applications that perform random accesses to the data;
- Packed data may be broadcast to separate variables in a single operation;
- 8-bit and 16-bit integer input data may be optionally converted to 32-bit floating-point values in the range [0.0, 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.

3.1.2.5 Shared Memory

Shared memory is where OpenCL local memory resides.

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

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

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

To get maximum performance, it is therefore important to understand how memory addresses map to memory banks in order to schedule the memory requests so as to minimize bank conflicts.

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

For devices of compute capability 1.x, the warp size is 32 and the number of banks is 16 (see Section 5.1); a shared memory request for a warp is split into one request for the first half of the warp and one request for the second half of the warp. As a consequence, there can be no bank conflict between a thread belonging to the first half of a warp and a thread belonging to the second half of the same warp.

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

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

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

Figure 3-5 and Figure 3-6 show some examples of conflict-free memory accesses while Figure 3-7 shows some examples of memory accesses that cause bank conflicts.

Other cases worth mentioning are when each thread accesses an element that is smaller or larger than 32 bits in size. For example, there 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];
```

There are also 2-way bank conflicts for arrays of **double**:

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

since the memory request is compiled into two separate 32-bit requests. 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];

double dataIn;
shared_lo[BaseIndex + tid] = __double2loint(dataIn);
shared_hi[BaseIndex + tid] = __double2hiint(dataIn);

double dataOut =
            __hiloint2double(shared_hi[BaseIndex + tid],
                             shared_lo[BaseIndex + tid]);
```

It might not always improve performance though and will perform worse on future architectures.

A structure assignment is compiled into as many memory requests as necessary for each member in the structure, so the following code, for example:

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

results in:

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

```
struct type {
```

```
        float x, y, z;
};
```

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

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

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

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

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

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

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

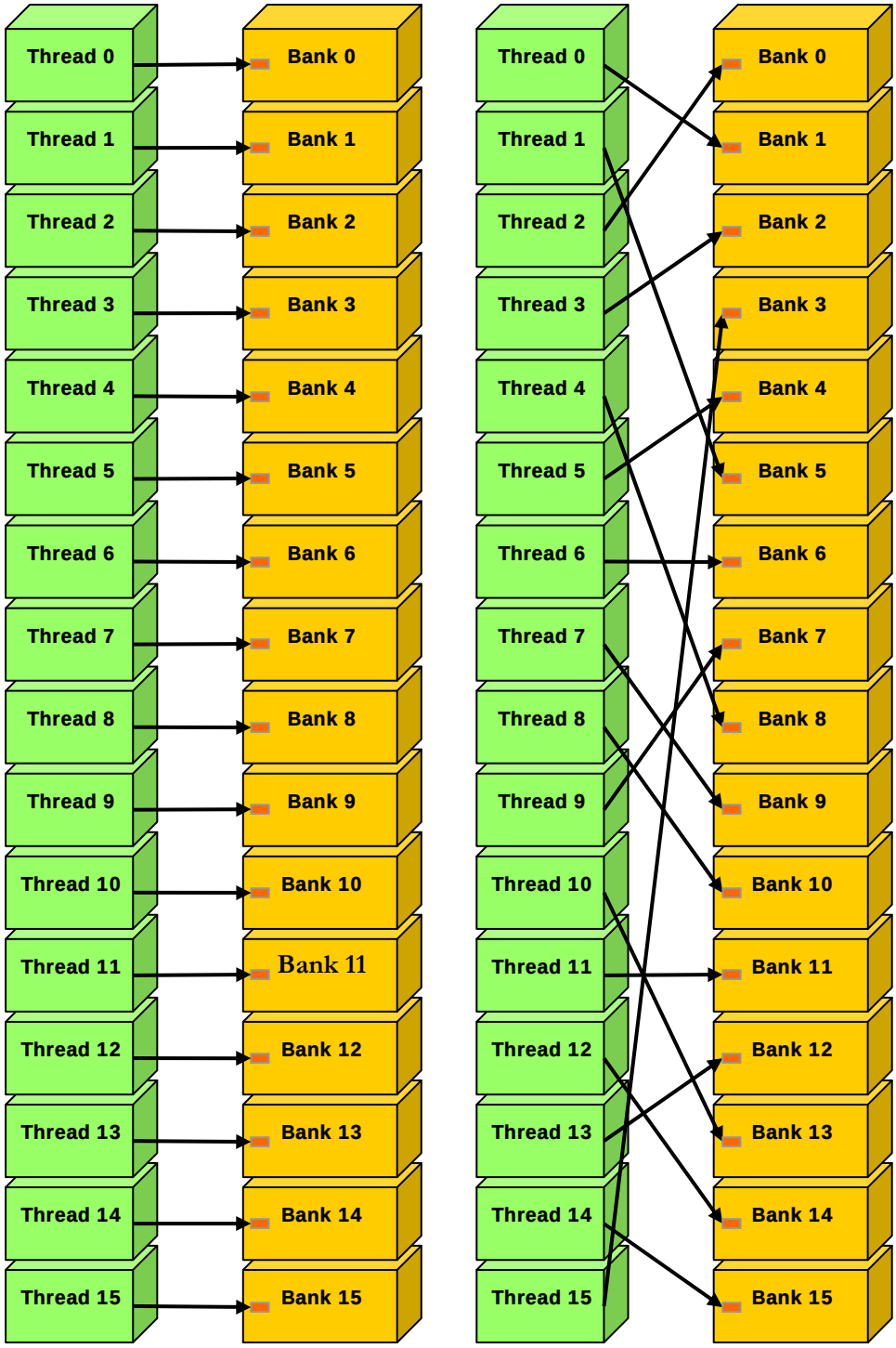
Finally, shared memory also features a broadcast mechanism whereby a 32-bit word can be read and broadcast to several threads simultaneously when servicing one memory read request. This reduces the number of bank conflicts when several threads of a half-warp read from an address within the same 32-bit word. More precisely, a memory read request made of several addresses is serviced in several steps over time – one step every two clock cycles – by servicing one conflict-free subset of these addresses per step until all addresses have been serviced; at each step, the subset is built from the remaining addresses that have yet to be serviced using the following procedure:

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

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

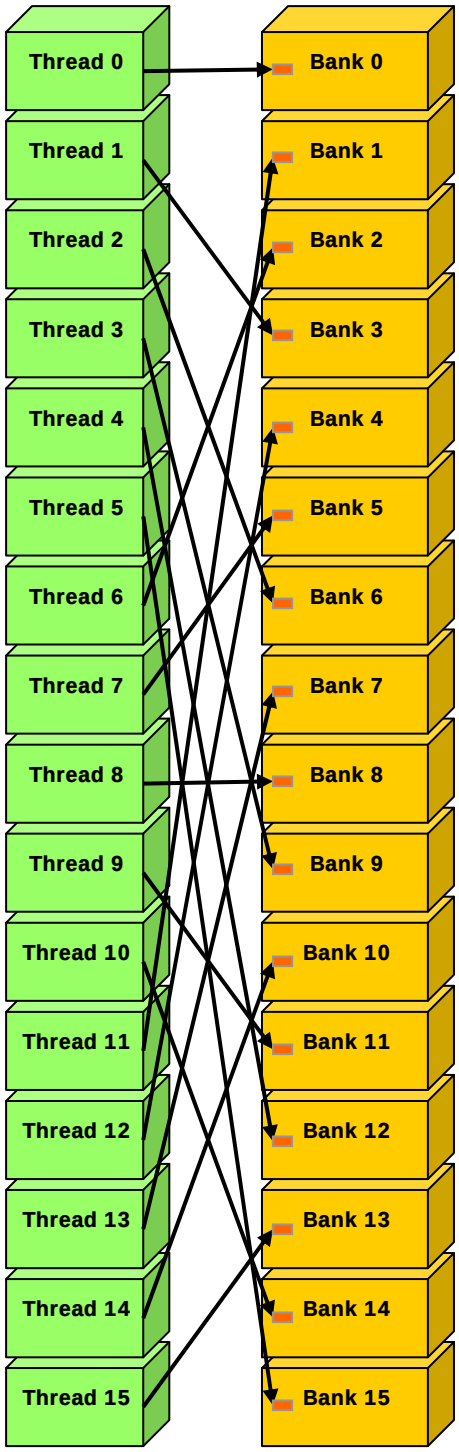
A common conflict-free case is when all threads of a half-warp read from an address within the same 32-bit word.

Figure 3-8 shows some examples of memory read accesses that involve the broadcast mechanism.



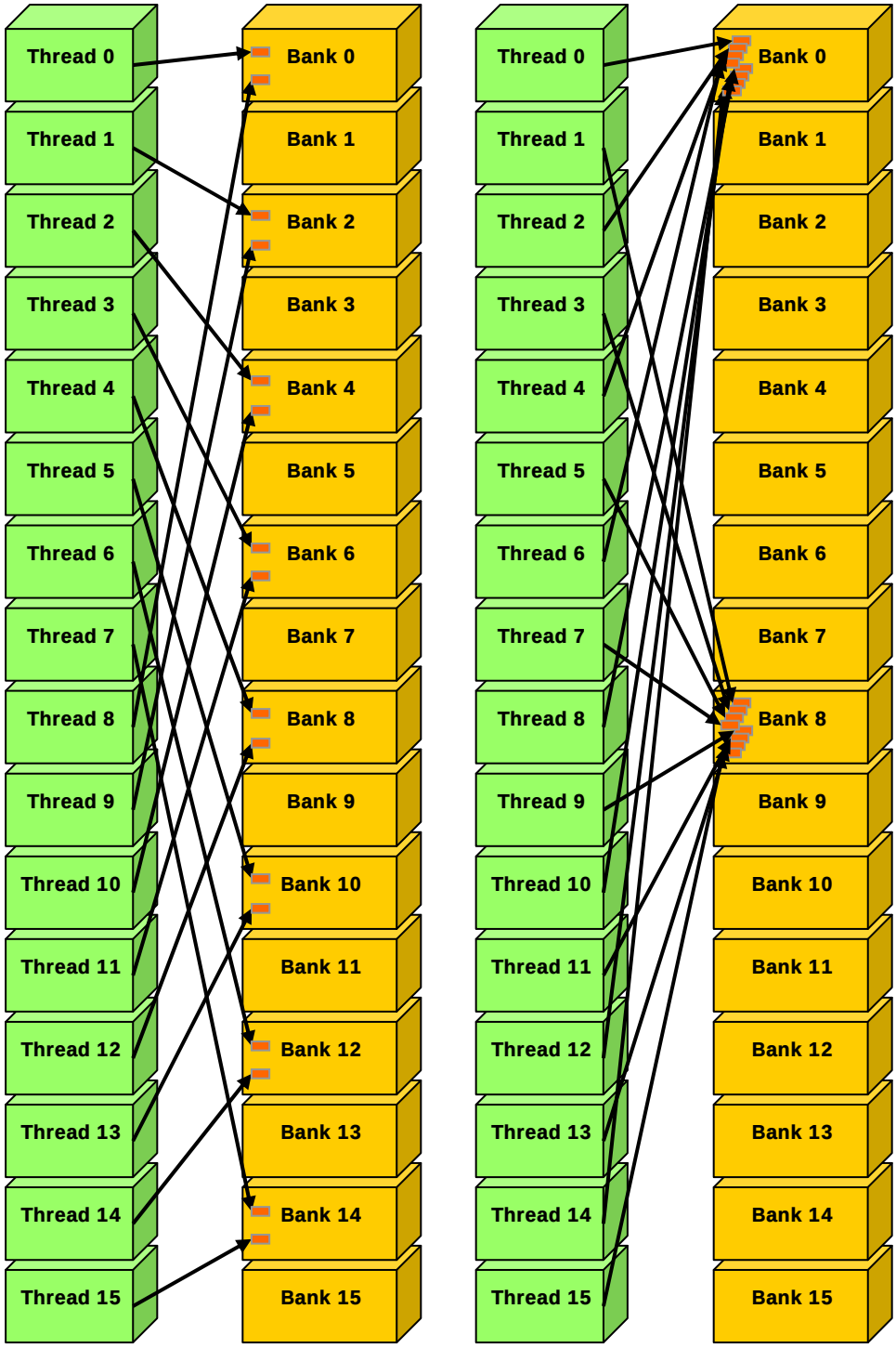
Left: linear addressing with a stride of one 32-bit word.
Right: random permutation.

Figure 3-5. Examples of Shared Memory Access Patterns without Bank Conflicts



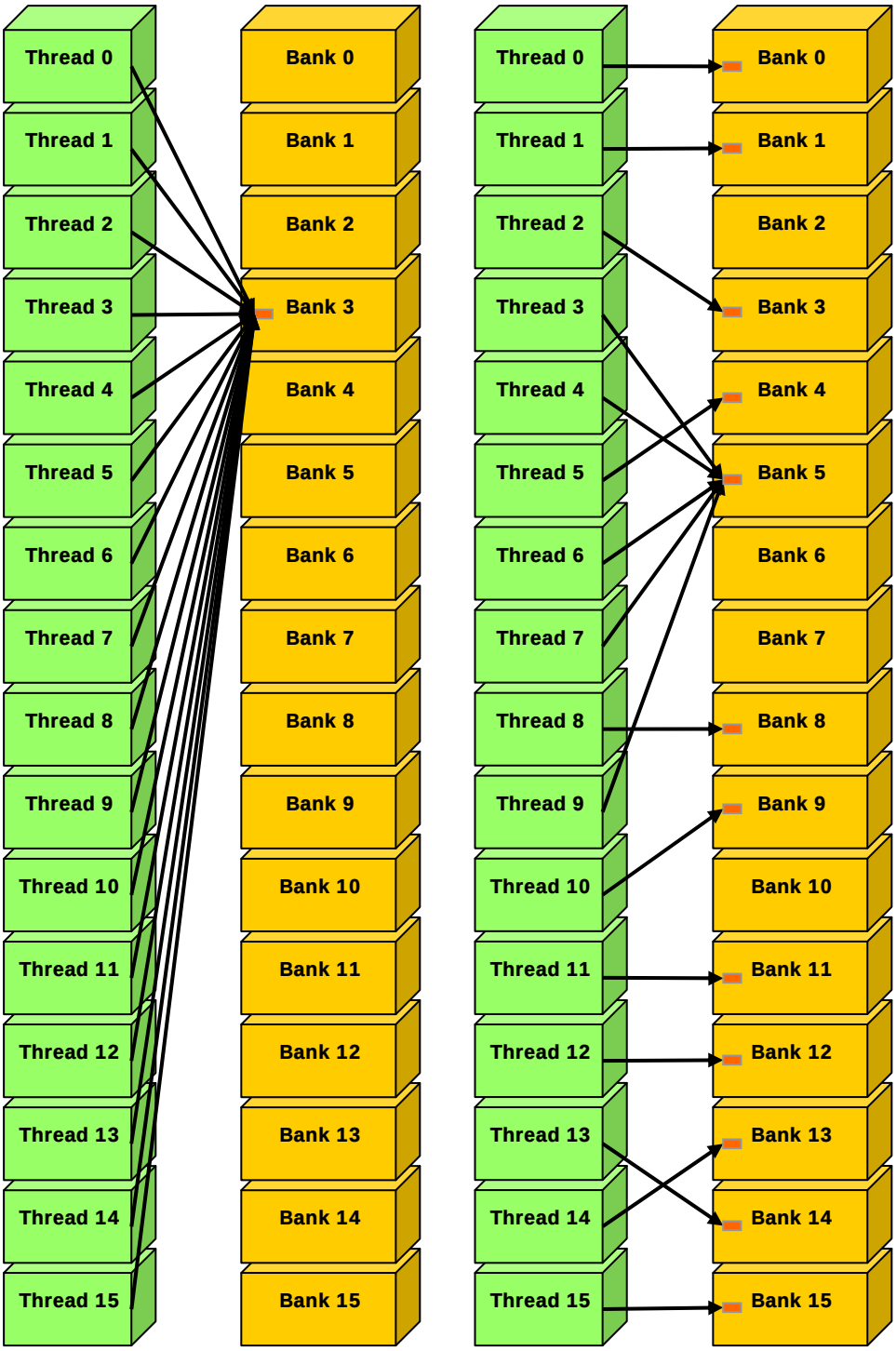
Linear addressing with a stride of three 32-bit words.

Figure 3-6. Example of a Shared Memory Access Pattern without Bank Conflicts



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

Figure 3-7. Examples of Shared Memory Access Patterns with Bank Conflicts



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

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

Figure 3-8.Example of Shared Memory Read Access Patterns with Broadcast

3.1.2.6 Registers

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

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

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

3.2 NDRange

How the NDRange affects the execution time of a kernel launch 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()`). There are however general guidelines, described in this section.

For a start, the kernel will simply fail to launch if the number of threads per block either is above the maximum number of threads per block as specified in Appendix A, or requires too many registers or shared memory than available per multiprocessor as mentioned in Section 2.1.2. The total number of registers required for a block is equal to

$$ceil(R \times ceil(T, 32), \frac{R_{\max}}{32})$$

where R is the number of registers required for the kernel, $R_{\max}$ is the number of registers per multiprocessor given in Appendix A, T is the number of threads per block, and $ceil(x, y)$ is equal to x rounded up to the nearest multiple of y. 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 the amount of shared memory used to pass the kernel's arguments. Note that each `double` or `long long` variable uses two registers. However, devices of compute capability 1.2 and higher have twice as many registers per multiprocessor as devices with lower compute capability.

Then, given a total number of threads per grid, the number of threads per block might be dictated by the need to have enough blocks in the grid to maximize the utilization of the available computing resources. First, there should be at least as many blocks as there are multiprocessors in the device. Then, running only one block per multiprocessor will force the multiprocessor to idle during thread synchronization and also during device memory reads if there are not enough threads per block to cover the load latency. It is therefore usually better to allow for two or more blocks to be active on each multiprocessor to allow overlap between blocks that wait and blocks that can run. For this to happen, not only should there be at least twice as many blocks as there are multiprocessors in the device, but also

the amount of registers and shared memory required per block must be low enough to allow for more than one active block (see Section 2.1.2). More thread blocks stream in pipeline fashion through the device and amortize overhead even more. The number of blocks per grid should be at least 100 if one wants it to scale to future devices; 1000 blocks will scale across several generations.

With a high enough number of blocks, the number of threads per block should be chosen as a multiple of the warp size to avoid swasting computing resources with under-populated warps, or better, a multiple of 64 for the reason invoked in Section 3.1.2.6. Allocating more threads per block is better for efficient time slicing, but the more threads per block, the fewer registers are available per thread, which might prevent the kernel invocation from succeeding.

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

The ratio of the number of active warps per multiprocessor to the maximum number of active warps (given in Appendix A) is called the multiprocessor *occupancy*. In order to maximize occupancy, the compiler attempts to minimize register usage while keeping the number of instructions and CUDA local memory usage to a minimum. The CUDA Software Development Kit provides a spreadsheet to assist programmers in choosing thread block size based on shared memory and register requirements.

3.3 Data Transfer between Host and Device

The bandwidth between device memory and the device is much higher than the bandwidth between device memory and host memory. Therefore, one should strive to minimize data transfer between the host and the device, for example, by moving 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 big one always performs much 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:

- Bandwidth between host memory and device memory is higher 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 3.1.2.1). 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.

3.4 Warp-Level Synchronization

Because a warp executes one common instruction at a time, threads within a warp are implicitly synchronized and this can be used to omit calls to the **`barrier()`** function 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()`** call.

```
// 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 call to **`barrier()`** is not enough however; **`myArray`** also needs to be declared as volatile as described in Section 2.2.2.

3.5 Overall Performance Optimization Strategies

Performance optimization revolves around three basic strategies:

- Maximizing parallel execution;
- Optimizing memory usage to achieve maximum memory bandwidth;
- Optimizing instruction usage to achieve maximum instruction throughput.

Maximizing parallel execution starts with structuring the algorithm in a way that exposes as much data parallelism as possible. At points in the algorithm where parallelism is broken because some threads need to synchronize in order to share data between each other, there are two cases: Either these threads belong to the same block, in which case they should use the **`barrier()`** function and share data through shared memory within the same kernel call, 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.

Once the parallelism of the algorithm has been exposed it needs to be mapped to the hardware as efficiently as possible. This is done by carefully choosing the NDRange of each kernel invocation as detailed in Section 3.2.

The application should also maximize parallel execution at a higher level by explicitly exposing concurrent execution on the device through queues, as well as maximizing concurrent execution between host and device.

Optimizing memory usage starts with minimizing data transfers with low-bandwidth. That means minimizing data transfers between the host and the device, as detailed in Section 3.3, since these have much lower bandwidth than data transfers between device and global memory. That also means minimizing data transfers between device and global memory by maximizing use of shared memory on the device, as mentioned in Section 3.1.2. Sometimes, the best optimization might even be to avoid any data transfer in the first place by simply recomputing the data instead whenever it is needed.

As detailed in Sections 3.1.2.1, 3.1.2.3, 3.1.2.4, and 3.1.2.5, the effective bandwidth can vary by an order of magnitude depending on access pattern for each type of memory. The next step in optimizing memory usage is therefore to organize memory accesses as optimally as possible based on the optimal memory access patterns. This optimization is especially important for global memory accesses as global memory bandwidth is low and its latency is hundreds of clock cycles (see Section 3.1.1.3). Shared memory accesses, on the other hand, are usually worth optimizing only in case they have a high degree of bank conflicts.

As for optimizing instruction usage, the use of arithmetic instructions with low throughput (see Section 3.1.1.1) should be minimized. This includes trading precision for speed when it does not affect the end result, such as using intrinsic instead of regular functions (intrinsic functions are listed in Section B.2) or single-precision instead of double-precision. Particular attention must be paid to control flow instructions due to the SIMT nature of the device as detailed in Section 3.1.1.2.

Appendix A. Technical Specifications

A.1 General Specifications

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

The following sections describe the technical specifications and features associated to each compute capability. The specifications for a given compute capability are the same as for the compute capability just below unless otherwise mentioned. Similarly, any feature supported for a given compute capability is supported for any higher compute capability.

The compute capability and number of multiprocessors of all CUDA-enabled devices are given in the following table:

	Number of Multiprocessors (1 Multiprocessor = 8 Processors)	**Compute Capability**
GeForce GTX 295	2x30	1.3
GeForce GTX 285, GTX 280	30	1.3
GeForce GTX 260	24	1.3
GeForce 9800 GX2	2x16	1.1
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	16	1.1
GeForce 8800 Ultra, 8800 GTX	16	1.0
GeForce 9800 GT, 8800 GT, 9800M GTX	14	1.1
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, 9800M GT	12	1.1
GeForce 8800 GTS	12	1.0
GeForce 9600 GT, 8800M GTS, 9800M GTS	8	1.1
GeForce 9700M GT	6	1.1
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	4	1.1
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU,	2	1.1

8100 mGPU		
GeForce 9300M GS, 9200M GS, 9100M G, 8400M G	1	1.1
Tesla S1070	4x30	1.3
Tesla C1060	30	1.3
Tesla S870	4x16	1.0
Tesla D870	2x16	1.0
Tesla C870	16	1.0
Quadro Plex 2200 D2	2x30	1.3
Quadro Plex 2100 D4	4x14	1.1
Quadro Plex 2100 Model S4	4x16	1.0
Quadro Plex 1000 Model IV	2x16	1.0
Quadro FX 5800	30	1.3
Quadro FX 4800	24	1.3
Quadro FX 4700 X2	2x14	1.1
Quadro FX 3700M	16	1.1
Quadro FX 5600	16	1.0
Quadro FX 3700	14	1.1
Quadro FX 3600M	12	1.1
Quadro FX 4600	12	1.0
Quadro FX 2700M	6	1.1
Quadro FX 1700, FX 570, NVS 320M, FX 1700M, FX 1600M, FX 770M, FX 570M	4	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	2	1.1
Quadro FX 370M, NVS 130M	1	1.1

The number of multiprocessors, the clock frequency and the total amount of device memory can be queried using the runtime.

A.1.1 Specifications for Compute Capability 1.0

- The maximum number of threads per block is 512;
- The maximum sizes of the x-, y-, and z-dimension of a thread block are 512, 512, and 64, respectively;
- The maximum size of each dimension of a grid of thread blocks is 65535;
- The warp size is 32 threads;
- The number of 32-bit registers per multiprocessor is 8192;
- The amount of shared memory available per multiprocessor is 16 KB organized into 16 banks (see Section 3.1.2.5);
- The total amount of constant memory is 64 KB;
- The total amount of local memory per thread is 16 KB;
- The cache working set for constant memory is 8 KB per multiprocessor;
- The cache working set for texture memory varies between 6 and 8 KB per multiprocessor;

- The maximum number of active blocks per multiprocessor is 8;
- The maximum number of active warps per multiprocessor is 24;
- The maximum number of active threads per multiprocessor is 768;
- The limit on kernel size is 2 million *PTX* instructions;

A.1.2 Specifications for Compute Capability 1.1

- Support for atomic functions operating on 32-bit words in global memory.

A.1.3 Specifications for Compute Capability 1.2

- Support for atomic functions operating in shared memory and atomic functions operating on 64-bit words in global memory;
- Support for warp vote functions;
- The number of registers per multiprocessor is 16384;
- The maximum number of active warps per multiprocessor is 32;
- The maximum number of active threads per multiprocessor is 1024.

A.1.4 Specifications for Compute Capability 1.3

- Support for double-precision floating-point numbers.

A.2 Floating-Point Standard

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

- There is no dynamically configurable rounding mode; however, most of the operations support IEEE rounding modes, exposed via device functions;
- 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;
- 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 only:
  - 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;
  - The result of an operation involving one or more input NaNs is the quiet NaN of bit pattern 0x7fffffff; note that;
  - Some instructions are not IEEE-compliant:

- Addition and multiplication are often combined into a single multiply-add instruction (FMAD), which truncates the intermediate result 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;

But, IEEE-compliant software (and therefore slower) implementations are provided through the following intrinsics from Appendix B:

- **`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,
- **`native_sqrt(float)`**: single-precision square root with IEEE rounding modes;

- For double-precision floating-point numbers only:
  - Round-to-nearest-even is the only supported IEEE rounding mode for reciprocal, division, and square root.

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 behaves.

A.3 Supported OpenCL Extensions

All compute devices supports the cl_khr_byte_addressable_store extension.

Devices of compute capability 1.1 and higher support the cl_khr_global_int32_base_atomics, cl_khr_global_int32_extended_atomics, cl_khr_local_int32_base_atomics, and cl_khr_local_int32_extended_atomics extensions.

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 when merged into an FMAD)
`x*y`	0 (IEEE-754 round-to-nearest-even) (except when merged into an FMAD)
`x/y`	2 (full range)
`1/x`	1 (full range)
`1/sqrt(x)` `rsqrt(x)`	2 (full range)
`sqrt(x)`	3 (full range)
`cbrt(x)`	1 (full range)
`hypot(x,y)`	3 (full range)
`exp(x)`	2 (full range)

Function	Maximum ulp error
`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)`	8 (full range)
`erfinv(x)`	5 (full range)
`erfcinv(x)`	7 (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)
`trunc(x)`	0 (full range)
`round(x)`	0 (full range)

Function	Maximum ulp error
`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)
`log(x)`	1 (full range)
`log2(x)`	1 (full range)

Function	Maximum ulp error
`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)`	7 (full range)
`erfinv(x)`	8 (full range)
`erfcinv(x)`	8 (full range)
`lgamma(x)`	4 (outside interval -11.0001 ... -2.2637; larger inside)
`tgamma(x)`	8 (full range)
`fma(x,y,z)`	0 (IEEE-754 round-to-nearest-even)
`frexp(x,exp)`	0 (full range)
`ldexp(x,exp)`	0 (full range)
`scalbn(x,n)`	0 (full range)
`scalbln(x,l)`	0 (full range)
`logb(x)`	0 (full range)
`ilogb(x)`	0 (full range)
`fmod(x,y)`	0 (full range)
`remainder(x,y)`	0 (full range)
`remquo(x,y,iptr)`	0 (full range)
`modf(x,iptr)`	0 (full range)
`fdim(x,y)`	0 (full range)
`trunc(x)`	0 (full range)
`round(x)`	0 (full range)
`rint(x)`	0 (full range)
`nearbyint(x)`	0 (full range)
`ceil(x)`	0 (full range)
`floor(x)`	0 (full range)
`lrint(x)`	0 (full range)

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)`	IEEE-compliant.
`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))`**.

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