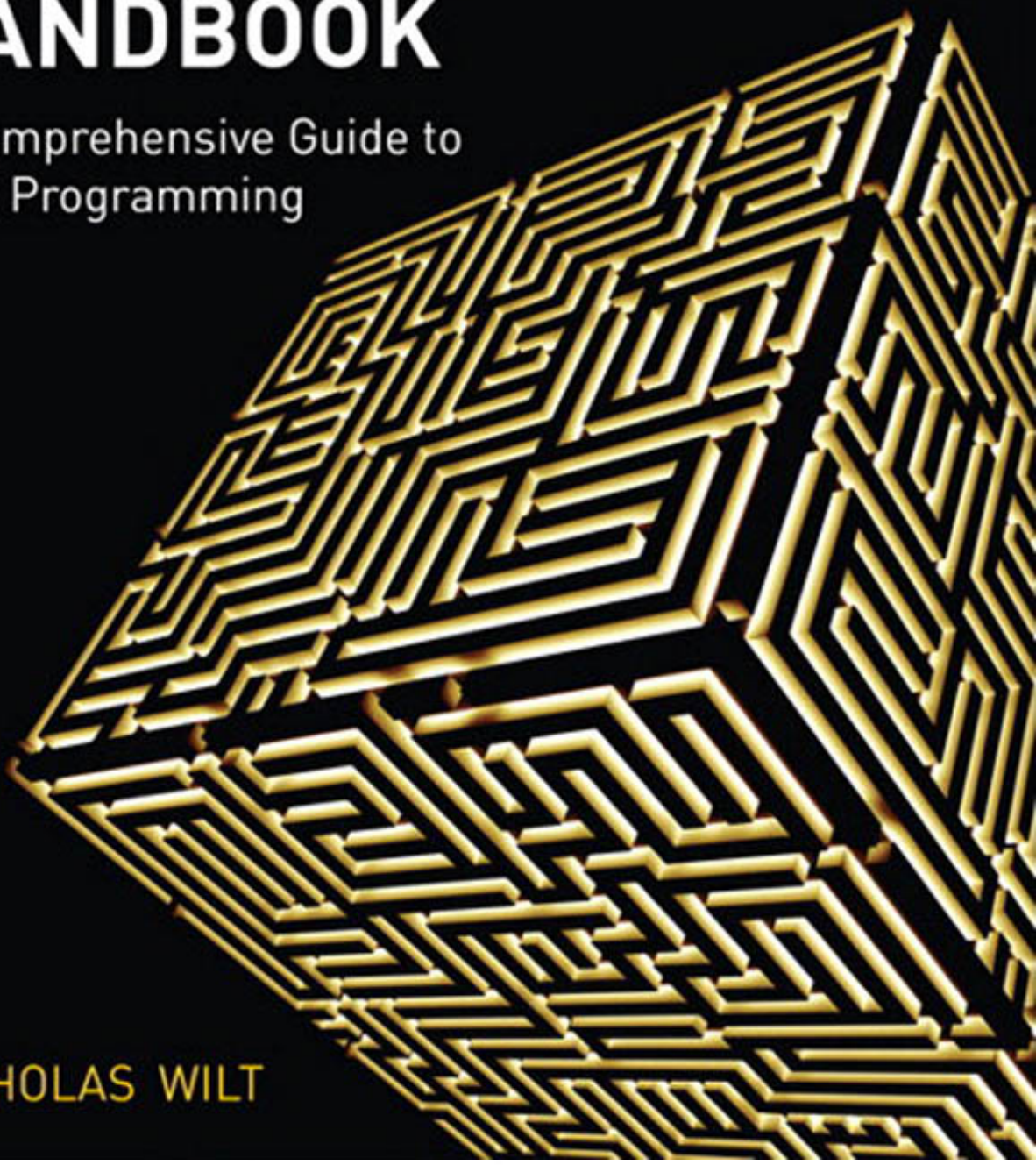
THE CUDA HANDBOOK

A Comprehensive Guide to GPU Programming



NICHOLAS WILT

The CUDA Handbook

This page intentionally left blank

The CUDA Handbook

A Comprehensive Guide to GPU Programming

Nicholas Wilt

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Cataloging in Publication Data is on file with the Library of Congress.

ISBN-13: 978-0-321-80946-9
ISBN-10: 0-321-80946-7
Text printed in the United States on recycled paper at RR Donelley in Crawfordsville, Indiana.
First printing, June 2013.

For Robin

This page intentionally left blank

Contents

This page intentionally left blank

Preface

If you are reading this book, I probably don't have to sell you on CUDA. Readers of this book should already be familiar with CUDA from using NVIDIA's SDK materials and documentation, taking a course on parallel programming, or reading the excellent introductory book *CUDA by Example* (Addison-Wesley, 2011) by Jason Sanders and Edward Kandrot.

Reviewing *CUDA by Example*, I am still struck by how much ground the book covers. Assuming no special knowledge from the audience, the authors manage to describe everything from memory types and their applications to graphics interoperability and even atomic operations. It is an excellent introduction to CUDA, but it is just that: an introduction. When it came to giving more detailed descriptions of the workings of the platform, the GPU hardware, the compiler driver `nvcc`, and important "building block" parallel algorithms like parallel prefix sum ("scan"), Jason and Edward rightly left those tasks to others.

This book is intended to help novice to intermediate CUDA programmers continue to elevate their game, building on the foundation laid by earlier work. In addition, while introductory texts are best read from beginning to end, *The CUDA Handbook* can be sampled. If you're preparing to build or program a new CUDA-capable platform, a review of Chapter 2 ("Hardware Architecture") might be in order. If you are wondering whether your application would benefit from using CUDA streams for additional concurrency, take a look at Chapter 6 ("Streams and Events"). Other chapters give detailed descriptions of the software architecture, GPU subsystems such as texturing and the streaming multiprocessors, and applications chosen according to their data access pattern and their relative importance in the universe of parallel algorithms. The chapters are relatively self-contained, though they do reference one another when appropriate.

The latest innovations, up to and including CUDA 5.0, also are covered here. In the last few years, CUDA and its target platforms have significantly evolved.

When *CUDA by Example* was published, the GeForce GTX 280 (GT200) was new, but since then, two generations of CUDA-capable hardware have become available. So besides more detailed discussions of existing features such as mapped pinned memory, this book also covers new instructions like Fermi's "ballot" and Kepler's "shuffle" and features such as 64-bit and unified virtual addressing and dynamic parallelism. We also discuss recent platform innovations, such as the integration of the PCI Express bus controller into Intel's "Sandy Bridge" CPUs.

However you choose to read the book—whether you read it straight through or keep it by your keyboard and consult it periodically—it's my sincerest hope that you will enjoy reading it as much as I enjoyed writing it.

Acknowledgments

I would like to take this opportunity to thank the folks at NVIDIA who have been patient enough to answer my questions, review my work, and give constructive feedback. Mark Harris, Norbert Juffa, and Lars Nyland deserve special thanks.

My reviewers generously took the time to examine the work before submission, and their comments were invaluable in improving the quality, clarity, and correctness of this work. I am especially indebted to Andre Brodtkorb, Scott Le Grand, Allan MacKinnon, Romelia Salomon-Ferrer, and Patrik Tennberg for their feedback.

My editor, the inimitable Peter Gordon, has been extraordinarily patient and supportive during the course of this surprisingly difficult endeavor. Peter's assistant, Kim Boedigheimer, set the standard for timeliness and professionalism in helping to complete the project. Her efforts at soliciting and coordinating review feedback and facilitating uploads to the Safari Web site are especially appreciated.

My wife Robin and my sons Benjamin, Samuel, and Gregory have been patient and supportive while I brought this project across the finish line.

This page intentionally left blank

About the Author

Nicholas Wilt has been programming computers professionally for more than twenty-five years in a variety of areas, including industrial machine vision, graphics, and low-level multimedia software. While at Microsoft, he served as the development lead for Direct3D 5.0 and 6.0, built the prototype for the Windows Desktop Manager, and did early GPU computing work. At NVIDIA, he worked on CUDA from the beginning, designing and often implementing most of CUDA's low-level abstractions. Now at Amazon, Mr. Wilt is working in cloud computing technologies relating to GPUs.

This page intentionally left blank

Chapter 8

Streaming Multiprocessors

The streaming multiprocessors (SMs) are the part of the GPU that runs our CUDA kernels. Each SM contains the following.

- Thousands of registers that can be partitioned among threads of execution
- Several caches:
  - *Shared memory* for fast data interchange between threads
  - *Constant cache* for fast broadcast of reads from constant memory
  - *Texture cache* to aggregate bandwidth from texture memory
  - *L1 cache* to reduce latency to local or global memory
- *Warp schedulers* that can quickly switch contexts between threads and issue instructions to warps that are ready to execute
- Execution cores for integer and floating-point operations:
  - Integer and single-precision floating point operations
  - Double-precision floating point
  - Special Function Units (SFUs) for single-precision floating-point transcendental functions

The reason there are many registers and the reason the hardware can context switch between threads so efficiently are to maximize the throughput of the hardware. The GPU is designed to have enough state to cover both execution latency and the memory latency of hundreds of clock cycles that it may take for data from device memory to arrive after a read instruction is executed.

The SMs are general-purpose processors, but they are designed very differently than the execution cores in CPUs: They target much lower clock rates; they support instruction-level parallelism, but not branch prediction or speculative execution; and they have less cache, if they have any cache at all. For suitable workloads, the sheer computing horsepower in a GPU more than makes up for these disadvantages.

The design of the SM has been evolving rapidly since the introduction of the first CUDA-capable hardware in 2006, with three major revisions, codenamed Tesla, Fermi, and Kepler. Developers can query the compute capability by calling `cudaGetDeviceProperties()` and examining `cudaDeviceProp.major` and `cudaDeviceProp.minor`, or by calling the driver API function `cuDeviceComputeCapability()`. Compute capability 1.x, 2.x, and 3.x correspond to Tesla-class, Fermi-class, and Kepler-class hardware, respectively. Table 8.1 summarizes the capabilities added in each generation of the SM hardware.

Table 8.1 SM Capabilities

COMPUTE LEVEL	INTRODUCED . . .
SM 1.1	Global memory atomics; mapped pinned memory; debuggable (e.g., breakpoint instruction)
SM 1.2	Relaxed coalescing constraints; warp voting (`any()` and `all()` intrinsics); atomic operations on shared memory
SM 1.3	Double precision support
SM 2.0	64-bit addressing; L1 and L2 cache; concurrent kernel execution; configurable 16K or 48K shared memory; bit manipulation instructions (`__clz()`, `__popc()`, `__ffs()`, `__brev()` intrinsics); directed rounding for single-precision floating-point values; fused multiply-add; 64-bit clock counter; surface load/store; 64-bit global atomic add, exchange, and compare-and-swap; global atomic add for single-precision floating-point values; warp voting (`ballot()` intrinsic); assertions and formatted output (`printf`).
SM 2.1	Function calls and indirect calls in kernels

Table 8.1 SM Capabilities *(Continued)*

COMPUTE LEVEL	INTRODUCED . . .
SM 3.0	Increase maximum grid size; warp shuffle; permute; 32K/32K shared memory configuration; configurable shared memory (32- or 64-bit mode) Bindless textures ("texture objects"); faster global atomics
SM 3.5	64-bit atomic min, max, AND, OR, and XOR; 64-bit funnel shift; read global memory via texture; dynamic parallelism

In Chapter 2, Figures 2.29 through 2.32 show block diagrams of different SMs. CUDA cores can execute integer and single-precision floating-point instructions; one double-precision unit implements double-precision support, if available; and Special Function Units implement reciprocal, recriprocal square root, sine/cosine, and logarithm/exponential functions. Warp schedulers dispatch instructions to these execution units as the resources needed to execute the instruction become available.

This chapter focuses on the instruction set capabilities of the SM. As such, it sometimes refers to the "SASS" instructions, the native instructions into which `ptxas` or the CUDA driver translate intermediate PTX code. Developers are not able to author SASS code directly; instead, NVIDIA has made these instructions visible to developers through the `cuobjdump` utility so they can direct optimizations of their source code by examining the compiled microcode.

8.1 Memory

8.1.1 REGISTERS

Each SM contains thousands of 32-bit registers that are allocated to threads as specified when the kernel is launched. Registers are both the fastest and most plentiful memory in the SM. As an example, the Kepler-class (SM 3.0) SMX contains 65,536 registers or 256K, while the texture cache is only 48K.

CUDA registers can contain integer or floating-point data; for hardware capable of performing double-precision arithmetic (SM 1.3 and higher), the operands are contained in even-valued register pairs. On SM 2.0 and higher hardware, register pairs also can hold 64-bit addresses.

CUDA hardware also supports wider memory transactions: The built-in `int2/float2` and `int4/float4` data types, residing in aligned register pairs or quads, respectively, may be read or written using single 64- or 128-bit-wide loads or stores. Once in registers, the individual data elements can be referenced as `.x/.y` (for `int2/float2`) or `.x/.y/.z/.w` (for `int4/float4`).

Developers can cause `nvcc` to report the number of registers used by a kernel by specifying the command-line option `--ptxas-options --verbose`. The number of registers used by a kernel affects the number of threads that can fit in an SM and often must be tuned carefully for optimal performance. The maximum number of registers used for a compilation may be specified with `--ptxas-options --maxregcount N`.

Register Aliasing

Because registers can hold floating-point or integer data, some intrinsics serve only to coerce the compiler into changing its view of a variable. The `__int_as_float()` and `__float_as_int()` intrinsics cause a variable to "change personalities" between 32-bit integer and single-precision floating point.

```
float __int_as_float( int i );
int __float_as_int( float f );
```

The `__double2loint()`, `__double2hiint()`, and `__hiloint2double()` intrinsics similarly cause registers to change personality (usually in-place). `__double_as_longlong()` and `__longlong_as_double()` coerce register pairs in-place; `__double2loint()` and `__double2hiint()` return the least and the most significant 32 bits of the input operand, respectively; and `__hiloint2double()` constructs a `double` out of the high and low halves.

```
int double2loint( double d );
int double2hiint( double d );
int hiloint2double( int hi, int lo );
double long_as_double(long long int i );
long long int __double_as_longlong( double d );
```

8.1.2 LOCAL MEMORY

Local memory is used to spill registers and also to hold local variables that are indexed and whose indices cannot be computed at compile time. Local memory is backed by the same pool of device memory as global memory, so it exhibits the same latency characteristics and benefits as the L1 and L2 cache hierarchy on Fermi and later hardware. Local memory is addressed in such a way that the memory transactions are automatically coalesced. The hardware includes

special instructions to load and store local memory: The SASS variants are `LLD/LST` for Tesla and `LDL/STL` for Fermi and Kepler.

8.1.3 GLOBAL MEMORY

The SMs can read or write global memory using `GLD/GST` instructions (on Tesla) and `LD/ST` instructions (on Fermi and Kepler). Developers can use standard C operators to compute and dereference addresses, including pointer arithmetic and the dereferencing operators `*`, `[]`, and `->`. Operating on 64- or 128-bit built-in data types (`int2/float2/int4/float4`) automatically causes the compiler to issue 64- or 128-bit load and store instructions. Maximum memory performance is achieved through *coalescing* of memory transactions, described in Section 5.2.9.

Tesla-class hardware (SM 1.x) uses special address registers to hold pointers; later hardware implements a load/store architecture that uses the same register file for pointers; integer and floating-point values; and the same address space for constant memory, shared memory, and global memory.[1]

Fermi-class hardware includes several features not available on older hardware.

- 64-bit addressing is supported via "wide" load/store instructions in which addresses are held in even-numbered register pairs. 64-bit addressing is not supported on 32-bit host platforms; on 64-bit host platforms, 64-bit addressing is enabled automatically. As a result, code generated for the same kernels compiled for 32- and 64-bit host platforms may have different register counts and performance.

- The L1 cache may be configured to be 16K or 48K in size.[2] (Kepler added the ability to split the cache as 32K L1/32K shared.) Load instructions can include cacheability hints (to tell the hardware to pull the read into L1 or to bypass the L1 and keep the data only in L2). These may be accessed via inline PTX or through the command line option `–X ptxas –dlcm=ca` (cache in L1 and L2, the default setting) or `–X ptxas –dlcm=cg` (cache only in L2).

Atomic operations (or just "atomics") update a memory location in a way that works correctly even when multiple GPU threads are operating on the same

1. Both constant and shared memory exist in address windows that enable them to be referenced by 32-bit addresses even on 64-bit architectures.
2. The hardware can change this configuration per kernel launch, but changing this state is expensive and will break concurrency for concurrent kernel launches.

memory location. The hardware enforces mutual exclusion on the memory location for the duration of the operation. Since the order of operations is not guaranteed, the operators supported generally are associative.[3]

Atomics first became available for global memory for SM 1.1 and greater and for shared memory for SM 1.2 and greater. Until the Kepler generation of hardware, however, global memory atomics were too slow to be useful.

The global atomic intrinsics, summarized in Table 8.2, become automatically available when the appropriate architecture is specified to `nvcc` via `--gpu-architecture`. All of these intrinsics can operate on 32-bit integers. 64-bit support for `atomicAdd()`, `atomicExch()`, and `atomicCAS()` was added

Table 8.2 Atomic Operations

MNEMONIC	DESCRIPTION
`atomicAdd`	Addition
`atomicSub`	Subtraction
`atomicExch`	Exchange
`atomicMin`	Minimum
`atomicMax`	Maximum
`atomicInc`	Increment (add 1)
`atomicDec`	Decrement (subtract 1)
`atomicCAS`	Compare and swap
`atomicAnd`	AND
`atomicOr`	OR
`atomicXor`	XOR

3. The only exception is single-precision floating-point addition. Then again, floating-point code generally must be robust in the face of the lack of associativity of floating-point operations; porting to different hardware, or even just recompiling the same code with different compiler options, can change the order of floating-point operations and thus the result.

in SM 1.2. `atomicAdd()` of 32-bit floating-point values (`float`) was added in SM 2.0. 64-bit support for `atomicMin()`, `atomicMax()`, `atomicAnd()`, `atomicOr()`, and `atomicXor()` was added in SM 3.5.

> **NOTE**
>
> Because atomic operations are implemented using hardware in the GPU's integrated memory controller, they do not work across the PCI Express bus and thus do not work correctly on device memory pointers that correspond to host memory or peer memory.

At the hardware level, atomics come in two forms: atomic operations that return the value that was at the specified memory location before the operator was performed, and reduction operations that the developer can "fire and forget" at the memory location, ignoring the return value. Since the hardware can perform the operation more efficiently if there is no need to return the old value, the compiler detects whether the return value is used and, if it is not, emits different instructions. In SM 2.0, for example, the instructions are called `ATOM` and `RED`, respectively.

8.1.4 CONSTANT MEMORY

Constant memory resides in device memory, but it is backed by a different, read-only cache that is optimized to broadcast the results of read requests to threads that all reference the same memory location. Each SM contains a small, latency-optimized cache for purposes of servicing these read requests. Making the memory (and the cache) read-only simplifies cache management, since the hardware has no need to implement write-back policies to deal with memory that has been updated.

SM 2.x and subsequent hardware includes a special optimization for memory that is not denoted as constant but that the compiler has identified as (1) read-only and (2) whose address is not dependent on the block or thread ID. The "load uniform" (LDU) instruction reads memory using the constant cache hierarchy and broadcasts the data to the threads.

8.1.5 SHARED MEMORY

Shared memory is very fast, on-chip memory in the SM that threads can use for data interchange within a thread block. Since it is a per-SM resource, shared

memory usage can affect occupancy, the number of warps that the SM can keep resident. SMs load and store shared memory with special instructions: `G2R/R2G` on SM 1.x, and `LDS/STS` on SM 2.x and later.

Shared memory is arranged as interleaved *banks* and generally is optimized for 32-bit access. If more than one thread in a warp references the same bank, a *bank conflict* occurs, and the hardware must handle memory requests consecutively until all requests have been serviced. Typically, to avoid bank conflicts, applications access shared memory with an interleaved pattern based on the thread ID, such as the following.

```
extern __shared__ float shared[];
float data = shared[BaseIndex + threadIdx.x];
```

Having all threads in a warp read from the same 32-bit shared memory location also is fast. The hardware includes a broadcast mechanism to optimize for this case. Writes to the same bank are serialized by the hardware, reducing performance. Writes to the same *address* cause race conditions and should be avoided.

For 2D access patterns (such as tiles of pixels in an image processing kernel), it's good practice to pad the shared memory allocation so the kernel can reference adjacent rows without causing bank conflicts. SM 2.x and subsequent hardware has 32 banks,[4] so for 2D tiles where threads in the same warp may access the data by row, it is a good strategy to pad the tile size to a multiple of 33 32-bit words.

On SM 1.x hardware, shared memory is about 16K in size;[5] on later hardware, there is a total of 64K of L1 cache that may be configured as 16K or 48K of shared memory, of which the remainder is used as L1 cache.[6]

Over the last few generations of hardware, NVIDIA has improved the hardware's handling of operand sizes other than 32 bits. On SM 1.x hardware, 8- and 16-bit reads from the same bank caused bank conflicts, while SM 2.x and later hardware can broadcast reads of any size out of the same bank. Similarly, 64-bit operands (such as `double`) in shared memory were so much slower than 32-bit operands on SM 1.x that developers sometimes had to resort to storing the data as separate high and low halves. SM 3.x hardware adds a new feature for

4. SM 1.x hardware had 16 banks (memory traffic from the first 16 threads and the second 16 threads of a warp was serviced separately), but strategies that work well on subsequent hardware also work well on SM 1.x.
5. 256 bytes of shared memory was reserved for parameter passing; in SM 2.x and later, parameters are passed via constant memory.
6. SM 3.x hardware adds the ability to split the cache evenly as 32K L1/32K shared.

kernels that predominantly use 64-bit operands in shared memory: a mode that increases the bank size to 64 bits.

Atomics in Shared Memory

SM 1.2 added the ability to perform atomic operations in shared memory. Unlike global memory, which implements atomics using single instructions (either GATOM or GRED, depending on whether the return value is used), shared memory atomics are implemented with explicit lock/unlock semantics, and the compiler emits code that causes each thread to loop over these lock operations until the thread has performed its atomic operation.

Listing 8.1 gives the source code to `atomic32Shared.cu`, a program specifically intended to be compiled to highlight the code generation for shared memory atomics. Listing 8.2 shows the resulting microcode generated for SM 2.0. Note how the LDSLK (load shared with lock) instruction returns a predicate that tells whether the lock was acquired, the code to perform the update is predicated, and the code loops until the lock is acquired and the update performed.

The lock is performed per 32-bit word, and the index of the lock is determined by bits 2–9 of the shared memory address. Take care to avoid contention, or the loop in Listing 8.2 may iterate up to 32 times.

Listing 8.1. `atomic32Shared.cu`.

```
__global__ void
Return32( int *sum, int *out, const int *pIn )
{
    extern __shared__ int s[];
    s[threadIdx.x] = pIn[threadIdx.x];
    __syncthreads();
    (void) atomicAdd( &s[threadIdx.x], *pIn );
    __syncthreads();
    out[threadIdx.x] = s[threadIdx.x];
}
```

Listing 8.2 `atomic32Shared.cubin` (microcode compiled for SM 2.0).

```
code for sm_20
    Function : _Z8Return32PiS_PKi
/*0000*/     MOV R1, c [0x1] [0x100];
/*0008*/     S2R R0, SR_Tid_X;
/*0010*/     SHL R3, R0, 0x2;
/*0018*/     MOV R0, c [0x0] [0x28];
/*0020*/     IADD R2, R3, c [0x0] [0x28];
```

```
/*0028*/     IMAD.U32.U32 RZ, R0, R1, RZ;
/*0030*/     LD R2, [R2];
/*0038*/     STS [R3], R2;
/*0040*/     SSY 0x80;
/*0048*/     BAR.RED.POPC RZ, RZ;
/*0050*/     LD R0, [R0];
/*0058*/     LDSLK P0, R2, [R3];
/*0060*/     @P0 IADD R2, R2, R0;
/*0068*/     @P0 STSUL [R3], R2;
/*0070*/     @!P0 BRA 0x58;
/*0078*/     NOP.S CC.T;
/*0080*/     BAR.RED.POPC RZ, RZ;
/*0088*/     LDS R0, [R3];
/*0090*/     IADD R2, R3, c [0x0] [0x24];
/*0098*/     ST [R2], R0;
/*00a0*/     EXIT;
    ................................
```

8.1.6 BARRIERS AND COHERENCY

The familiar `__syncthreads()` intrinsic waits until all the threads in the thread block have arrived before proceeding. It is needed to maintain coherency of shared memory within a thread block.[7] Other, similar memory barrier instructions can be used to enforce some ordering on broader scopes of memory, as described in Table 8.3.

Table 8.3 Memory Barrier Intrinsics

INTRINSIC	DESCRIPTION
`__syncthreads()`	Waits until all shared memory accesses made by the calling thread are visible to all threads in the threadblock
`threadfence_block()`	Waits until all global and shared memory accesses made by the calling thread are visible to all threads in the threadblock
`threadfence()`	Waits until all global and shared memory accesses made by the calling thread are visible to • All threads in the threadblock for shared memory accesses • All threads in the device for global memory accesses

7. Note that threads within a warp run in lockstep, sometimes enabling developers to write so-called "warp synchronous" code that does not call `__syncthreads()`. Section 7.3 describes thread and warp execution in detail, and Part III includes several examples of warp synchronous code.

Table 8.3 Memory Barrier Intrinsics *(Continued)*

INTRINSIC	DESCRIPTION
`threadfence_system()` (SM 2.x only)	Waits until all global and shared memory accesses made by the calling thread are visible to • All threads in the threadblock for shared memory accesses • All threads in the device for global memory accesses • Host threads for page-locked host memory accesses

8.2 Integer Support

The SMs have the full complement of 32-bit integer operations.

- Addition with optional negation of an operand for subtraction
- Multiplication and multiply-add
- Integer division
- Logical operations
- Condition code manipulation
- Conversion to/from floating point
- Miscellaneous operations (e.g., SIMD instructions for narrow integers, population count, find first zero)

CUDA exposes most of this functionality through standard C operators. Nonstandard operations, such as 24-bit multiplication, may be accessed using inline PTX assembly or intrinsic functions.

8.2.1 MULTIPLICATION

Multiplication is implemented differently on Tesla- and Fermi-class hardware. Tesla implements a 24-bit multiplier, while Fermi implements a 32-bit multiplier. As a consequence, full 32-bit multiplication on SM 1.x hardware requires four instructions. For performance-sensitive code targeting Tesla-class

Table 8.4 Multiplication Intrinsics

INTRINSIC	DESCRIPTION
`__[u]mul24`	Returns the least significant 32 bits of the product of the 24 least significant bits of the integer parameters. The 8 most significant bits of the inputs are ignored.
`__[u]mulhi`	Returns the most significant 32 bits of the product of the inputs.
`__[u]mul64hi`	Returns the most significant 64 bits of the products of the 64-bit inputs.

hardware, it is a performance win to use the intrinsics for 24-bit multiply.[8] Table 8.4 shows the intrinsics related to multiplication.

8.2.2 MISCELLANEOUS (BIT MANIPULATION)

The CUDA compiler implements a number of intrinsics for bit manipulation, as summarized in Table 8.5. On SM 2.x and later architectures, these intrinsics

Table 8.5 Bit Manipulation Intrinsics

INTRINSIC	SUMMARY	DESCRIPTION
`__brev(x)`	Bit reverse	Reverses the order of bits in a word
`__byte_perm(x,y,s)`	Permute bytes	Returns a 32-bit word whose bytes were selected from the two inputs according to the selector parameter `s`
`__clz(x)`	Count leading zeros	Returns number of zero bits (0–32) before most significant set bit
`__ffs(x)`	Find first sign bit	Returns the position of the least significant set bit. The least significant bit is position 1. For an input of 0, `__ffs()` returns 0.
`__popc(x)`	Population count	Returns the number of set bits
`__[u]sad(x,y,z)`	Sum of absolute differences	Adds `|x-y|` to `z` and returns the result

8. Using `__mul24()` or `__umul24()` on SM 2.x and later hardware, however, is a performance *penalty*.

map to single instructions. On pre-Fermi architectures, they are valid but may compile into many instructions. When in doubt, disassemble and look at the microcode! 64-bit variants have "`ll`" (two ells for "long long") appended to the intrinsic name `__clzll(), ffsll(), popcll(), brevll()`.

8.2.3 FUNNEL SHIFT (SM 3.5)

GK110 added a 64-bit "funnel shift" instruction that concatenates two 32-bit values together (the least significant and most significant halves are specified as separate 32-bit inputs, but the hardware operates on an aligned register pair), shifts the resulting 64-bit value left or right, and then returns the most significant (for left shift) or least significant (for right shift) 32 bits.

Funnel shift may be accessed with the intrinsics given in Table 8.6. These intrinsics are implemented as inline device functions (using inline PTX assembler) in `sm_35_intrinsics.h`. By default, the least significant 5 bits of the shift count are masked off; the `_lc` and `_rc` intrinsics clamp the shift value to the range 0..32.

Applications for funnel shift include the following.

- Multiword shift operations
- Memory copies between misaligned buffers using aligned loads and stores
- Rotate

Table 8.6 Funnel Shift Intrinsics

INTRINSIC	DESCRIPTION
`__funnelshift_l(hi, lo, sh)`	Concatenates [hi:lo] into a 64-bit quantity, shifts it left by `(sh&31)` bits, and returns the most significant 32 bits
`__funnelshift_lc(hi, lo, sh)`	Concatenates [hi:lo] into a 64-bit quantity, shifts it left by `min(sh,32)` bits, and returns the most significant 32 bits
`__funnelshift_r(hi, lo, sh)`	Concatenates [hi:lo] into a 64-bit quantity, shifts it right by `(sh&31)` bits, and returns the least significant 32 bits
`__funnelshift_rc(hi, lo, sh)`	Concatenates [hi:lo] into a 64-bit quantity, shifts it right by `min(sh,32)` bits, and returns the least significant 32 bits

To right-shift data sizes greater than 64 bits, use repeated `__funnelshift_r()` calls, operating from the least significant to the most significant word. The most significant word of the result is computed using `operator>>`, which shifts in zero or sign bits as appropriate for the integer type. To left-shift data sizes greater than 64 bits, use repeated `__funnelshift_l()` calls, operating from the most significant to the least significant word. The least significant word of the result is computed using `operator<<`. If the `hi` and `lo` parameters are the same, the funnel shift effects a rotate operation.

8.3 Floating-Point Support

Fast native floating-point hardware is the *raison d'être* for GPUs, and in many ways they are equal to or superior to CPUs in their floating-point implementation. Denormals are supported at full speed,[9] directed rounding may be specified on a per-instruction basis, and the Special Function Units deliver high-performance approximation functions to six popular single-precision transcendentals. In contrast, x86 CPUs implement denormals in microcode that runs perhaps 100x slower than operating on normalized floating-point operands. Rounding direction is specified by a control word that takes dozens of clock cycles to change, and the only transcendental approximation functions in the SSE instruction set are for reciprocal and reciprocal square root, which give 12-bit approximations that must be refined with a Newton-Raphson iteration before being used.

Since GPUs' greater core counts are offset somewhat by their lower clock frequencies, developers can expect at most a 10x (or thereabouts) speedup on a level playing field. If a paper reports a 100x or greater speedup from porting an optimized CPU implementation to CUDA, chances are one of the above-described "instruction set mismatches" played a role.

8.3.1 FORMATS

Figure 8.2 depicts the three (3) IEEE standard floating-point formats supported by CUDA: double precision (64-bit), single precision (32-bit), and half precision (16-bit). The values are divided into three fields: sign, exponent, and mantissa.

9. With the exception that single-precision denormals are not supported at all on SM 1.x hardware.

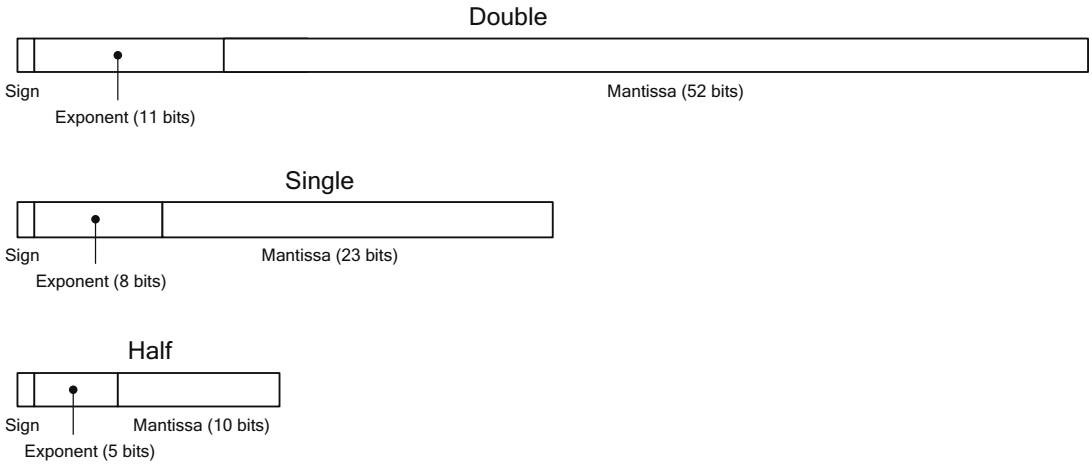


Figure 8.2 Floating-point formats.

For `double`, `single`, and `half`, the exponent fields are 11, 8, and 5 bits in size, respectively; the corresponding mantissa fields are 52, 23, and 10 bits.

The exponent field changes the interpretation of the floating-point value. The most common ("normal") representation encodes an implicit 1 bit into the mantissa and multiplies that value by 2^{e-bias}, where *bias* is the value added to the actual exponent before encoding into the floating-point representation. The bias for single precision, for example, is 127.

Table 8.7 summarizes how floating-point values are encoded. For most exponent values (so-called "normal" floating-point values), the mantissa is assumed to have an implicit 1, and it is multiplied by the biased value of the exponent. The maximum exponent value is reserved for infinity and Not-A-Number values. Dividing by zero (or overflowing a division) yields infinity; performing an invalid operation (such as taking the square root or logarithm of a negative number) yields a NaN. The minimum exponent value is reserved for values too small to represent with the implicit leading 1. As the so-called *denormals*[10] get closer to zero, they lose bits of effective precision, a phenomenon known as *gradual underflow*. Table 8.8 gives the encodings and values of certain extreme values for the three formats.

10. Sometimes called *subnormals*.

Table 8.7 Floating-Point Representations

DOUBLE PRECISION			
EXPONENT	**MANTISSA**	**VALUE**	**CASE NAME**
0	0	±0	Zero
0	Nonzero	$\pm 2^{-1022}(0.mantissa)$	Denormal
1 to 2046	Any	$\pm 2^{e-1023}(1.mantissa)$	Normal
2047	0	±∞	Infinity
2047	Nonzero		Not-A-Number
SINGLE PRECISION			
EXPONENT	**MANTISSA**	**VALUE**	**CASE NAME**
0	0	±0	Zero
0	Nonzero	$\pm 2^{-126}(0.mantissa)$	Denormal
1 to 254	Any	$\pm 2^{e-127}(1.mantissa)$	Normal
255	0	±∞	Infinity
255	Nonzero		Not-A-Number
HALF PRECISION			
EXPONENT	**MANTISSA**	**VALUE**	**CASE NAME**
0	0	±0	Zero
0	Nonzero	$\pm 2^{-14}(0.mantissa)$	Denormal
1 to 30	Any	$\pm 2^{e-15}(1.mantissa)$	Normal
31	0	±∞	Infinity
31	Nonzero		Not-A-Number

Table 8.8 Floating-Point Extreme Values

DOUBLE PRECISION		
	HEXADECIMAL	EXACT VALUE
Smallest denormal	0...0001	2^{-1074}
Largest denormal	000F...F	$2^{-1022}(1-2^{-52})$
Smallest normal	0010...0	2^{-1022}
1.0	3FF0...0	1
Maximum integer	4340...0	2^{53}
Largest normal	7F7FFFFF	$2^{1024}(1-2^{-53})$
Infinity	7FF00000	Infinity
SINGLE PRECISION		
	HEXADECIMAL	EXACT VALUE
Smallest denormal	00000001	2^{-149}
Largest denormal	007FFFFF	$2^{-126}(1-2^{-23})$
Smallest normal	00800000	2^{-126}
1.0	3F800000	1
Maximum integer	4B800000	2^{24}
Largest normal	7F7FFFFF	$2^{128}(1-2^{-24})$
Infinity	7F800000	Infinity

continues

Table 8.8 Floating-Point Extreme Values *(Continued)*

HALF PRECISION		
	HEXADECIMAL	EXACT VALUE
Smallest denormal	0001	2^{-24}
Largest denormal	07FF	$2^{-14}(1-2^{-10})$
Smallest normal	0800	2^{-14}
1.0	3c00	1
Maximum integer	6800	2^{11}
Largest normal	7BFF	$2^{16}(1-2^{-11})$
Infinity	7C00	Infinity

Rounding

The IEEE standard provides for four (4) round modes.

- Round-to-nearest-even (also called "round-to-nearest")
- Round toward zero (also called "truncate" or "chop")
- Round down (or "round toward negative infinity")
- Round up (or "round toward positive infinity")

Round-to-nearest, where intermediate values are rounded to the nearest representable floating-point value after each operation, is by far the most commonly used round mode. Round up and round down (the "directed rounding modes") are used for *interval arithmetic*, where a pair of floating-point values are used to bracket the intermediate result of a computation. To correctly bracket a result, the lower and upper values of the interval must be rounded toward negative infinity ("down") and toward positive infinity ("up"), respectively.

The C language does not provide any way to specify round modes on a per-instruction basis, and CUDA hardware does not provide a control word to implicitly specify rounding modes. Consequently, CUDA provides a set of intrinsics to specify the round mode of an operation, as summarized in Table 8.9.

Table 8.9 Intrinsics for Rounding

INTRINSIC	OPERATION
`__fadd_[rn|rz|ru|rd]`	Addition
`__fmul_[rn|rz|ru|rd]`	Multiplication
`__fmaf_[rn|rz|ru|rd]`	Fused multiply-add
`__frcp_[rn|rz|ru|rd]`	Recriprocal
`__fdiv_[rn|rz|ru|rd]`	Division
`__fsqrt_[rn|rz|ru|rd]`	Square root
`__dadd_[rn|rz|ru|rd]`	Addition
`__dmul_[rn|rz|ru|rd]`	Multiplication
`__fma_[rn|rz|ru|rd]`	Fused multiply-add
`__drcp_[rn|rz|ru|rd]`	Reciprocal
`__ddiv_[rn|rz|ru|rd]`	Division
`__dsqrt_[rn|rz|ru|rd]`	Square root

Conversion

In general, developers can convert between different floating-point representations and/or integers using standard C constructs: implicit conversion or explicit typecasts. If necessary, however, developers can use the intrinsics listed in Table 8.10 to perform conversions that are not in the C language specification, such as those with directed rounding.

Because `half` is not standardized in the C programming language, CUDA uses `unsigned short` in the interfaces for `__half2float()` and `__float2half()`. `__float2half()` only supports the round-to-nearest rounding mode.

```
float __half2float( unsigned short );
unsigned short __float2half( float );
```

Table 8.10 Intrinsics for Conversion

INTRINSIC	OPERATION
__float2int_[rn|rz|ru|rd]	float to int
__float2uint_[rn|rz|ru|rd]	float to unsigned int
__int2float_[rn|rz|ru|rd]	int to float
__uint2float_[rn|rz|ru|rd]	unsigned int to float
__float2ll_[rn|rz|ru|rd]	float to 64-bit int
__ll2float_[rn|rz|ru|rd]	64-bit int to float
__ull2float_[rn|rz|ru|rd]	unsigned 64-bit int to float
__double2float_[rn|rz|ru|rd]	double to float
__double2int_[rn|rz|ru|rd]	double to int
__double2uint_[rn|rz|ru|rd]	double to unsigned int
__double2ll_[rn|rz|ru|rd]	double to 64-bit int
__double2ull_[rn|rz|ru|rd]	double to 64-bit unsigned int
__int2double_rn	int to double
__uint2double_rn	unsigned int to double
__ll2double_[rn|rz|ru|rd]	64-bit int to double
__ull2double_[rn|rz|ru|rd]	unsigned 64-bit int to double

8.3.2 SINGLE PRECISION (32-BIT)

Single-precision floating-point support is the workhorse of GPU computation. GPUs have been optimized to natively deliver high performance on this data

type,[11] not only for core standard IEEE operations such as addition and multiplication, but also for nonstandard operations such as approximations to transcendentals such as `sin()` and `log()`. The 32-bit values are held in the same register file as integers, so coercion between single-precision floating-point values and 32-bit integers (with `__float_as_int()` and `__int_as_float()`) is free.

Addition, Multiplication, and Multiply-Add

The compiler automatically translates +, –, and * operators on floating-point values into addition, multiplication, and multiply-add instructions. The `__fadd_rn()` and `__fmul_rn()` intrinsics may be used to suppress fusion of addition and multiplication operations into multiply-add instructions.

Reciprocal and Division

For devices of compute capability 2.x and higher, the division operator is IEEE-compliant when the code is compiled with `--prec-div=true`. For devices of compute capability 1.x or for devices of compute capability 2.x when the code is compiled with `--prec-div=false`, the division operator and `__fdividef(x,y)` have the same accuracy, but for $2^{126}<y<2^{128}$, `__fdividef(x,y)` delivers a result of zero, whereas the division operator delivers the correct result. Also, for $2^{126}<y<2^{128}$, if x is infinity, `__fdividef(x,y)` returns NaN, while the division operator returns infinity.

Transcendentals (SFU)

The Special Function Units (SFUs) in the SMs implement very fast versions of six common transcendental functions.

- Sine and cosine
- Logarithm and exponential
- Reciprocal and reciprocal square root

Table 8.11, excerpted from the paper on the Tesla architecture[12] summarizes the supported operations and corresponding precision. The SFUs do not implement full precision, but they are reasonably good approximations of these functions and they are *fast*. For CUDA ports that are significantly faster than an optimized CPU equivalent (say, 25x or more), the code most likely relies on the SFUs.

---

11. In fact, GPUs had full 32-bit floating-point support before they had full 32-bit integer support. As a result, some early GPU computing literature explained how to implement integer math with floating-point hardware!
12. Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, March–April 2008, p. 47.

Table 8.11 SFU Accuracy

FUNCTION	ACCURACY (GOOD BITS)	ULP ERROR
1/x	24.02	0.98
1/sqrt(x)	23.40	1.52
2^x	22.51	1.41
$log2_x$	22.57	n/a
sin/cos	22.47	n/a

The SFUs are accessed with the intrinsics given in Table 8.12. Specifying the `--fast-math` compiler option will cause the compiler to substitute conventional C runtime calls with the corresponding SFU intrinsics listed above.

Table 8.12 SFU Intrinsics

INTRINSIC	OPERATION
`__cosf(x)`	cos x
`__exp10f(x)`	10^x
`__expf(x)`	e^x
`__fdividef(x,y)`	x/y
`__logf(x)`	ln x
`__log2f(x)`	$\log_2$ x
`__log10f(x)`	$\log_{10}$ x
`__powf(x,y)`	x^y
`__sinf(x)`	sin x
`__sincosf(x,sptr,cptr)`	`*s=sin(x);` `*c=cos(x);`
`__tanf(x)`	tan x

Miscellaneous

`__saturate(x)` returns 0 if `x<0`, 1 if `x>1`, and `x` otherwise.

8.3.3 DOUBLE PRECISION (64-BIT)

Double-precision floating-point support was added to CUDA with SM 1.3 (first implemented in the GeForce GTX 280), and much improved double-precision support (both functionality and performance) became available with SM 2.0. CUDA's hardware support for double precision features full-speed denormals and, starting in SM 2.x, a native fused multiply-add instruction (FMAD), compliant with IEEE 754 c. 2008, that performs only one rounding step. Besides being an intrinsically useful operation, FMAD enables full accuracy on certain functions that are converged with the Newton-Raphson iteration.

As with single-precision operations, the compiler automatically translates standard C operators into multiplication, addition, and multiply-add instructions. The `__dadd_rn()` and `__dmul_rn()` intrinsics may be used to suppress fusion of addition and multiplication operations into multiply-add instructions.

8.3.4 HALF PRECISION (16-BIT)

With 5 bits of exponent and 10 bits of significand, `half` values have enough precision for HDR (high dynamic range) images and can be used to hold other types of values that do not require `float` precision, such as angles. Half precision values are intended for storage, not computation, so the hardware only provides instructions to convert to/from 32-bit.[13] These instructions are exposed as the `__halftofloat()` and `__floattohalf()` intrinsics.

```
float __halftofloat( unsigned short );
unsigned short __floattohalf( float );
```

These intrinsics use `unsigned short` because the C language has not standardized the `half` floating-point type.

8.3.5 CASE STUDY: `float→half` CONVERSION

Studying the `float→half` conversion operation is a useful way to learn the details of floating-point encodings and rounding. Because it's a simple unary

13. `half` floating-point values are supported as a texture format, in which case the TEX intrinsics return `float` and the conversion is automatically performed by the texture hardware.

operation, we can focus on the encoding and rounding without getting distracted by the details of floating-point arithmetic and the precision of intermediate representations.

When converting from `float` to `half`, the correct output for any `float` too large to represent is `half` infinity. Any float too small to represent as a `half` (even a denormal `half`) must be clamped to `0.0`. The maximum `float` that rounds to `half` 0.0 is `0x32FFFFFF`, or 2.98^{-8}, while the smallest `float` that rounds to `half` infinity is 65520.0. `float` values inside this range can be converted to `half` by propagating the sign bit, rebiasing the exponent (since float has an 8-bit exponent biased by 127 and half has a 5-bit exponent biased by 15), and rounding the `float` mantissa to the nearest `half` mantissa value. Rounding is straightforward in all cases except when the input value falls exactly between the two possible output values. When this is the case, the IEEE standard specifies rounding to the "nearest even" value. In decimal arithmetic, this would mean rounding 1.5 to 2.0, but also rounding 2.5 to 2.0 and (for example) rounding 0.5 to 0.0.

Listing 8.3 shows a C routine that exactly replicates the `float`-to-`half` conversion operation, as implemented by CUDA hardware. The variables `exp` and `mag` contain the input exponent and "magnitude," the mantissa and exponent together with the sign bit masked off. Many operations, such as comparisons and rounding operations, can be performed on the magnitude without separating the exponent and mantissa.

The macro `LG_MAKE_MASK`, used in Listing 8.3, creates a mask with a given bit count: `#define LG_MAKE_MASK(bits) ((1<<bits)-1)`. A `volatile union` is used to treat the same 32-bit value as `float` and `unsigned int`; idioms such as `*((float *) (&u))` are not portable. The routine first propagates the input sign bit and masks it off the input.

After extracting the magnitude and exponent, the function deals with the special case when the input `float` is INF or NaN, and does an early exit. Note that INF is signed, but NaN has a canonical unsigned value. Lines 50–80 clamp the input `float` value to the minimum or maximum values that correspond to representable `half` values and recompute the magnitude for clamped values. Don't be fooled by the elaborate code constructing `f32MinRInfin` and `f32MaxRf16_zero`; those are constants with the values `0x477ff000` and `0x32ffffff`, respectively.

The remainder of the routine deals with the cases of output normal and denormal (input denormals are clamped in the preceding code, so `mag` corresponds to a normal `float`). As with the clamping code, `f32Minf16Normal` is a constant, and its value is `0x38ffffff`.

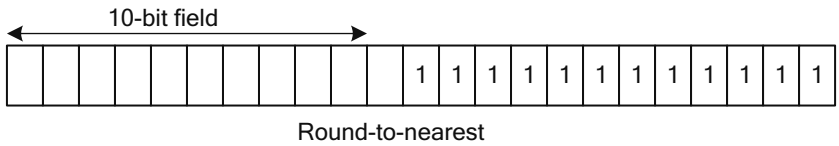


Figure 8.3 Rounding mask (`half`).

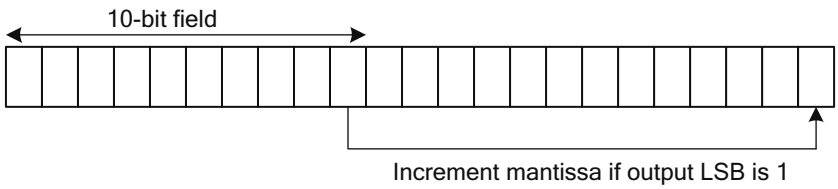


Figure 8.4 Round-to-nearest-even (`half`).

To construct a normal, the new exponent must be computed (lines 92 and 93) and the correctly rounded 10 bits of mantissa shifted into the output. To construct a denormal, the implicit 1 must be OR'd into the output mantissa and the resulting mantissa shifted by the amount corresponding to the input exponent. For both normals and denormals, the rounding of the output mantissa is accomplished in two steps. The rounding is accomplished by adding a mask of 1's that ends just short of the output's LSB, as seen in Figure 8.3.

This operation increments the output mantissa if bit 12 of the input is set; if the input mantissa is all 1's, the overflow causes the output exponent to correctly increment. If we added one more 1 to the MSB of this adjustment, we'd have elementary school–style rounding where the tiebreak goes to the larger number. Instead, to implement round-to-nearest even, we conditionally increment the output mantissa if the LSB of the 10-bit output is set (Figure 8.4). Note that these steps can be performed in either order or can be reformulated in many different ways.

Listing 8.3 `ConvertToHalf()`.

```
/*
 * exponent shift and mantissa bit count are the same.
 *    When we are shifting, we use [f16|f32]ExpShift
 *    When referencing the number of bits in the mantissa,
 *        we use [f16|f32]MantissaBits
 */
```

```
const int f16ExpShift = 10;
const int f16MantissaBits = 10;

const int f16ExpBias = 15;
const int f16MinExp = -14;
const int f16MaxExp = 15;
const int f16SignMask = 0x8000;

const int f32ExpShift = 23;
const int f32MantissaBits = 23;
const int f32ExpBias = 127;
const int f32SignMask = 0x80000000;

unsigned short
ConvertFloatToHalf( float f )
{
    /*
     * Use a volatile union to portably coerce
     * 32-bit float into 32-bit integer
     */
    volatile union {
        float f;
        unsigned int u;
    } uf;
    uf.f = f;

    // return value: start by propagating the sign bit.
    unsigned short w = (uf.u >> 16) & f16SignMask;

    // Extract input magnitude and exponent
    unsigned int mag = uf.u & ~f32SignMask;
    int exp = (int) (mag >> f32ExpShift) - f32ExpBias;

    // Handle float32 Inf or NaN
    if ( exp == f32ExpBias+1 ) {    // INF or NaN

        if ( mag & LG_MAKE_MASK(f32MantissaBits) )
            return 0x7fff; // NaN

        // INF - propagate sign
        return w|0x7c00;
    }

    /*
     * clamp float32 values that are not representable by float16
     */
    {
        // min float32 magnitude that rounds to float16 infinity

        unsigned int f32MinRInfin = (f16MaxExp+f32ExpBias) <<
            f32ExpShift;
        f32MinRInfin |= LG_MAKE_MASK( f16MantissaBits+1 ) <<
            (f32MantissaBits-f16MantissaBits-1);

        if (mag > f32MinRInfin)
            mag = f32MinRInfin;
    }
```

```
{
    // max float32 magnitude that rounds to float16 0.0

    unsigned int f32MaxRf16_zero = f16MinExp+f32ExpBias-
        (f32MantissaBits-f16MantissaBits-1);
    f32MaxRf16_zero <<= f32ExpShift;
    f32MaxRf16_zero |= LG_MAKE_MASK( f32MantissaBits );

    if (mag < f32MaxRf16_zero)
        mag = f32MaxRf16_zero;
}

/*
 * compute exp again, in case mag was clamped above
 */
exp = (mag >> f32ExpShift) - f32ExpBias;

// min float32 magnitude that converts to float16 normal
unsigned int f32Minf16Normal = ((f16MinExp+f32ExpBias)<<
    f32ExpShift);
f32Minf16Normal |= LG_MAKE_MASK( f32MantissaBits );
if ( mag >= f32Minf16Normal ) {
    //
    // Case 1: float16 normal
    //

    // Modify exponent to be biased for float16, not float32
    mag += (unsigned int) ((f16ExpBias-f32ExpBias)<<
        f32ExpShift);

    int RelativeShift = f32ExpShift-f16ExpShift;

    // add rounding bias
    mag += LG_MAKE_MASK(RelativeShift-1);

    // round-to-nearest even
    mag += (mag >> RelativeShift) & 1;

    w |= mag >> RelativeShift;
}
else {
    /*
     * Case 2: float16 denormal
     */

    // mask off exponent bits - now fraction only
    mag &= LG_MAKE_MASK(f32MantissaBits);

    // make implicit 1 explicit
    mag |= (1<<f32ExpShift);

    int RelativeShift = f32ExpShift-f16ExpShift+f16MinExp-exp;

    // add rounding bias
    mag += LG_MAKE_MASK(RelativeShift-1);
```

```
        // round-to-nearest even
        mag += (mag >> RelativeShift) & 1;

        w |= mag >> RelativeShift;
    }
    return w;
}
```

In practice, developers should convert `float` to `half` by using the `__floattohalf()` intrinsic, which the compiler translates to a single `F2F` machine instruction. This sample routine is provided purely to aid in understanding floating-point layout and rounding; also, examining all the special-case code for INF/NAN and denormal values helps to illustrate why these features of the IEEE spec have been controversial since its inception: They make hardware slower, more costly, or both due to increased silicon area and engineering effort for validation.

In the code accompanying this book, the `ConvertFloatToHalf()` routine in Listing 8.3 is incorporated into a program called `float_to_float16.cu` that tests its output for every 32-bit floating-point value.

8.3.6 MATH LIBRARY

CUDA includes a built-in math library modeled on the C runtime library, with a few small differences: CUDA hardware does not include a rounding mode register (instead, the round mode is encoded on a per-instruction basis),[14] so functions such as `rint()` that reference the current rounding mode always round-to-nearest. Additionally, the hardware does not raise floating-point exceptions; results of aberrant operations, such as taking the square root of a negative number, are encoded as NaNs.

Table 8.13 lists the math library functions and the maximum error in ulps for each function. Most functions that operate on `float` have an "f" appended to the function name—for example, the functions that compute the sine function are as follows.

```
double sin( double angle );
float sinf( float angle );
```

These are denoted in Table 8.13 as, for example, `sin[f]`.

14. Encoding a round mode per instruction and keeping it in a control register are not irreconcilable. The Alpha processor had a 2-bit encoding to specify the round mode per instruction, one setting of which was to use the rounding mode specified in a control register! CUDA hardware just uses a 2-bit encoding for the four round modes specified in the IEEE specification.

Table 8.13 Math Library

			ULP ERROR	
FUNCTION	OPERATION	EXPRESSION	32	64
`x+y`	Addition	x+y	0[1]	0
`x*y`	Multiplication	x*y	0[1]	0
`x/y`	Division	x/y	2[2]	0
`1/x`	Reciprocal	1/x	1[2]	0
`acos[f](x)`	Inverse cosine	$\cos^{-1} x$	3	2
`acosh[f](x)`	Inverse hyperbolic cosine	$\ln\left(x+\sqrt{x^2+1}\right)$	4	2
`asin[f](x)`	Inverse sine	$\sin^{-1} x$	4	2
`asinh[f](x)`	Inverse hyperbolic sine	$\text{sign}(x)\ln\left(\lvert x\rvert+\sqrt{1+x^2}\right)$	3	2
`atan[f](x)`	Inverse tangent	$\tan^{-1} x$	2	2
`atan2[f](y,x)`	Inverse tangent of y/x	$\tan^{-1} x\left(\frac{y}{x}\right)$	3	2
`atanh[f](x)`	Inverse hyperbolic tangent	$\tanh^{-1}$	3	2
`cbrt[f](x)`	Cube root	$\sqrt[3]{x}$	1	1
`ceil[f](x)`	"Ceiling," nearest integer greater than or equal to *x*	$\lceil x \rceil$	0	
`copysign[f](x,y)`	Sign of y, magnitude of x		n/a	
`cos[f](x)`	Cosine	cos x	2	1
`cosh[f](x)`	Hyperbolic cosine	$\frac{e^x+e^{-x}}{2}$	2	
`cospi[f](x)`	Cosine, scaled by π	cos πx	2	

continues

Table 8.13 Math Library *(Continued)*

			ULP ERROR	
FUNCTION	**OPERATION**	**EXPRESSION**	**32**	**64**
`erf[f](x)`	Error function	$\frac{2}{\pi}\int_0^x e^{-t^2}$	3	2
`erfc[f](x)`	Complementary error function	$1-\frac{2}{\pi}\int_0^x e^{-t^2}$	6	4
`erfcinv[f](y)`	Inverse complementary error function	Return x for which `y=1-erff(x)`	7	8
`erfcx[f](x)`	Scaled error function	e^{x^2} (`erff(x)`)	6	3
`erfinv[f](y)`	Inverse error function	Return x for which `y=erff(x)`	3	5
`exp[f](x)`	Natural exponent	e^x	2	1
`exp10[f](x)`	Exponent (base 10)	10^x	2	1
`exp2[f](x)`	Exponent (base 2)	2^x	2	1
`expm1[f](x)`	Natural exponent, minus one	$e^x - 1$	1	1
`fabs[f](x)`	Absolute value	$\|x\|$	0	0
`fdim[f](x,y)`	Positive difference	$\begin{cases} x-y, x>y \\ +0, x \le y \\ \text{NAN, x or y NaN} \end{cases}$	0	0
`floor[f](x)`	"Floor," nearest integer less than or equal to *x*	$\lfloor x \rfloor$	0	0
`fma[f](x,y,z)`	Multiply-add	xy + z	0	0
`fmax[f](x,y)`	Maximum	$\begin{cases} x, x>y \text{ or isNaN(y)} \\ y, \text{otherwise} \end{cases}$	0	0

Table 8.13 Math Library *(Continued)*

			ULP ERROR	
FUNCTION	OPERATION	EXPRESSION	32	64
`fmin[f](x,y)`	Minimum	$\begin{cases} x, x < y \text{ or } isNaN(y) \\ y, otherwise \end{cases}$	0	0
`fmod[f](x,y)`	Floating-point remainder		0	0
`frexp[f](x,exp)`	Fractional component		0	0
`hypot[f](x,y)`	Length of hypotenuse	$\sqrt{x^2 + y^2}$	3	2
`ilogb[f](x)`	Get exponent		0	0
`isfinite(x)`	Nonzero if x is not ±INF		n/a	
`isinf(x)`	Nonzero if x is ±INF		n/a	
`isnan(x)`	Nonzero if x is a NaN		n/a	
`j0[f](x)`	Bessel function of the first kind (n=0)	$J_0(x)$	9^3	7^3
`j1[f](x)`	Bessel function of the first kind (n=1)	$J_1(x)$	9^3	7^3
`jn[f](n,x)`	Bessel function of the first kind	$J_n(x)$	*	
`ldexp[f](x,exp)`	Scale by power of 2	$x2^{exp}$	0	0
`lgamma[f](x)`	Logarithm of gamma function	$\ln\left(\Gamma(x)\right)$	6^4	4^4
`llrint[f](x)`	Round to long long		0	0
`llround[f](x)`	Round to long long		0	0
`lrint[f](x)`	Round to long		0	0
`lround[f](x)`	Round to long		0	0
`log[f](x)`	Natural logarithm	$\ln(x)$	1	1
`log10[f](x)`	Logarithm (base 10)	$\log_{10} x$	3	1
`log1p[f](x)`	Natural logarithm of *x*+1	$\ln(x + 1)$	2	1

continues

Table 8.13 Math Library *(Continued)*

			ULP ERROR	
FUNCTION	**OPERATION**	**EXPRESSION**	**32**	**64**
`log2[f](x)`	Logarithm (base 2)	$\log_2 x$	3	1
`logb[f](x)`	Get exponent		0	0
`modff(x,iptr)`	Split fractional and integer parts		0	0
`nan[f](cptr)`	Returns NaN	NaN	n/a	
`nearbyint[f](x)`	Round to integer		0	0
`nextafter[f](x,y)`	Returns the FP value closest to x in the direction of y		n/a	
`normcdf[f](x)`	Normal cumulative distribution		6	5
`normcdinv[f](x)`	Inverse normal cumulative distribution		5	8
`pow[f](x,y)`	Power function	x^y	8	2
`rcbrt[f](x)`	Inverse cube root	$\frac{1}{\sqrt[3]{x}}$	2	1
`remainder[f](x,y)`	Remainder		0	0
`remquo[f](x,y,iptr)`	Remainder (also returns quotient)		0	0
`rsqrt[f](x)`	Reciprocal	$\frac{1}{\sqrt{x}}$	2	1
`rint[f](x)`	Round to nearest `int`		0	0
`round[f](x)`	Round to nearest `int`		0	0
`scalbln[f](x,n)`	Scale x by 2^n (n is `long int`)	$x2^n$	0	0
`scalbn[f](x,n)`	Scale x by 2^n (n is `int`)	$x2^n$	0	0
`signbit(x)`	Nonzero if x is negative		n/a	0
`sin[f](x)`	Sine	sin x	2	1

Table 8.13 Math Library *(Continued)*

			ULP ERROR	
FUNCTION	OPERATION	EXPRESSION	32	64
`sincos[f](x,s,c)`	Sine and cosine	`*s=sin(x);` `*c=cos(x);`	2	1
`sincospi[f](x,s,c)`	Sine and cosine	`*s=sin(πx);` `*c=cos(πx);`	2	1
`sinh[f](x)`	Hyperbolic sine	$\frac{e^x - e^{-x}}{2}$	3	1
`sinpi[f](x)`	Sine, scaled by π	$\sin \pi x$	2	1
`sqrt[f](x)`	Square root	$\sqrt{x}$	3[5]	0
`tan[f](x)`	Tangent	$\tan x$	4	2
`tanh[f](x)`	Hyperbolic tangent	$\frac{\sinh x}{\cosh x}$	2	1
`tgamma[f](x)`	True gamma function	$\Gamma(x)$	11	8
`trunc[f](x)`	Truncate (round to integer toward zero)		0	0
`y0[f](x)`	Bessel function of the second kind (n=0)	$Y_0(x)$	9[3]	7[3]
`y1[f](x)`	Bessel function of the second kind (n=1)	$Y_1(x)$	9[3]	7[3]
`yn[f](n,x)`	Bessel function of the second kind	$Y_n(x)$	**	

* For the Bessel functions `jnf(n,x)` and `jn(n,x)`, for `n=128` the maximum absolute error is 2.2×10^{-6} and 5×10^{-12}, respectively.

** For the Bessel function `ynf(n,x)`, the error is $\lceil 2 + 2.5n \rceil$ for |x|; otherwise, the maximum absolute error is 2.2×10^{-6} for `n=128`. For `yn(n,x)`, the maximum absolute error is 5×10^{-12}.

1. On SM 1.x class hardware, the precision of addition and multiplication operation that are merged into FMAD instructions will suffer due to truncation of the intermediate mantissa.
2. On SM 2.x and later hardware, developers can reduce this error rate to 0 ulps by specifying `--prec-div=true`.
3. For `float`, the error is 9 ulps for |x|<8; otherwise, the maximum absolute error is 2.2×10^{-6}. For `double`, the error is 7 ulps for |x|<8; otherwise, the maximum absolute error is 5×10^{-12}.
4. The error for `lgammaf()` is greater than 6 inside the interval –10.001, –2.264. The error for `lgamma()` is greater than 4 inside the interval –11.001, –2.2637.
5. On SM 2.x and later hardware, developers can reduce this error rate to 0 ulps by specifying `--prec-sqrt=true`.

Conversion to Integer

According to the C runtime library definition, the `nearbyint()` and `rint()` functions round a floating-point value to the nearest integer using the "current rounding direction," which in CUDA is always round-to-nearest-even. In the C runtime, `nearbyint()` and `rint()` differ only in their handling of the INEXACT exception. But since CUDA does not raise floating-point exceptions, the functions behave identically.

`round()` implements elementary school–style rounding: For floating-point values halfway between integers, the input is always rounded away from zero. NVIDIA recommends against using this function because it expands to eight (8) instructions as opposed to one for `rint()` and its variants. `trunc()` truncates or "chops" the floating-point value, rounding toward zero. It compiles to a single instruction.

Fractions and Exponents

```
float frexpf(float x, int *eptr);
```

`frexpf()` breaks the input into a floating-point significand in the range [0.5, 1.0) and an integral exponent for 2, such that

$$x = Significand \cdot 2^{Exponent}$$

```
float logbf( float x );
```

`logbf()` extracts the exponent from x and returns it as a floating-point value. It is equivalent to `floorf(log2f(x))`, except it is faster. If `x` is a denormal, `logbf()` returns the exponent that x would have if it were normalized.

```
float ldexpf( float x, int exp );
float scalbnf( float x, int n );
float scanblnf( float x, long n );
```

`ldexpf()`, `scalbnf()`, and `scalblnf()` all compute $x2^n$ by direct manipulation of floating-point exponents.

Floating-Point Remainder

`modff()` breaks the input into fractional and integer parts.

```
float modff( float x, float *intpart );
```

The return value is the fractional part of x, with the same sign.

`remainderf(x,y)` computes the floating-point remainder of dividing x by y. The return value is `x-n*y`, where `n` is x/y, rounded to the nearest integer. If |x – ny| = 0.5, `n` is chosen to be even.

```
float remquof(float x, float y, int *quo);
```

computes the remainder and passes back the lower bits of the integral quotient x/y, with the same sign as x/y.

Bessel Functions

The Bessel functions of order *n* relate to the differential equation

$$x^2 \frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - n^2)y = 0$$

n can be a real number, but for purposes of the C runtime, it is a nonnegative integer.

The solution to this second-order ordinary differential equation combines Bessel functions of the first kind and of the second kind.

$$y(x) = c_1 J_n(x) + c_2 Y_n(x)$$

The math runtime functions `jn[f]()` and `yn[f]()` compute $J_n(x)$ and $Y_n(x)$, respectively. `j0f()`, `j1f()`, `y0f()`, and `y1f()` compute these functions for the special cases of n=0 and n=1.

Gamma Function

The gamma function Γ is an extension of the factorial function, with its argument shifted down by 1, to real numbers. It has a variety of definitions, one of which is as follows.

$$\Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

The function grows so quickly that the return value loses precision for relatively small input values, so the library provides the `lgamma()` function, which returns the natural logarithm of the gamma function, in addition to the `tgamma()` ("true gamma") function.

8.3.7 ADDITIONAL READING

Goldberg's survey (with the captivating title "What Every Computer Scientist Should Know About Floating Point Arithmetic") is a good introduction to the topic.

http://download.oracle.com/docs/cd/E19957-01/806-3568/ncg_goldberg.html

Nathan Whitehead and Alex Fit-Florea of NVIDIA have coauthored a white paper entitled "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs."

http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf

Increasing Effective Precision

Dekker and Kahan developed methods to almost double the effective precision of floating-point hardware using pairs of numbers in exchange for a slight reduction in exponent range (due to intermediate underflow and overflow at the far ends of the range). Some papers on this topic include the following.

Dekker, T.J. Point technique for extending the available precision. *Numer. Math.* 18, 1971, pp. 224–242.

Linnainmaa, S. Software for doubled-precision floating point computations. *ACM TOMS* 7, pp. 172–283 (1981).

Shewchuk, J.R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18, 1997, pp. 305–363.

Some GPU-specific work on this topic has been done by Andrew Thall, Da Graça, and Defour.

Guillaume, Da Graça, and David Defour. Implementation of float-float operators on graphics hardware, *7th Conference on Real Numbers and Computers*, RNC7 (2006).

http://hal.archives-ouvertes.fr/docs/00/06/33/56/PDF/float-float.pdf

Thall, Andrew. Extended-precision floating-point numbers for GPU computation. 2007.

http://andrewthall.org/papers/df64_qf128.pdf

8.4 Conditional Code

The hardware implements "condition code" or CC registers that contain the usual 4-bit state vector (sign, carry, zero, overflow) used for integer comparison. These CC registers can be set using comparison instructions such as `ISET`, and they can direct the flow of execution via *predication* or *divergence*. Predication allows (or suppresses) the execution of instructions on a per-thread basis within a warp, while divergence is the conditional execution of longer instruction sequences. Because the processors within an SM execute instructions in SIMD fashion at warp granularity (32 threads at a time), divergence can result in fewer instructions executed, provided all threads within a warp take the same code path.

8.4.1 PREDICATION

Due to the additional overhead of managing divergence and convergence, the compiler uses predication for short instruction sequences. The effect of most instructions can be predicated on a condition; if the condition is not TRUE, the instruction is suppressed. This suppression occurs early enough that predicated execution of instructions such as load/store and `TEX` inhibits the memory traffic that the instruction would otherwise generate. Note that predication has no effect on the eligibility of memory traffic for global load/store coalescing. The addresses specified to all load/store instructions in a warp must reference consecutive memory locations, even if they are predicated.

Predication is used when the number of instructions that vary depending on a condition is small; the compiler uses heuristics that favor predication up to about 7 instructions. Besides avoiding the overhead of managing the branch synchronization stack described below, predication also gives the compiler more optimization opportunities (such as instruction scheduling) when emitting microcode. The ternary operator in C (`? :`) is considered a compiler hint to favor predication.

Listing 8.2 gives an excellent example of predication, as expressed in microcode. When performing an atomic operation on a shared memory location, the compiler emits code that loops over the shared memory location until it has successfully performed the atomic operation. The `LDSLK` (load shared and lock) instruction returns a condition code that tells whether the lock was acquired. The instructions to perform the operation then are predicated on that condition code.

```
/*0058*/ LDSLK P0, R2, [R3];
/*0060*/ @P0 IADD R2, R2, R0;
/*0068*/ @P0 STSUL [R3], R2;
/*0070*/ @!P0 BRA 0x58;
```

This code fragment also highlights how predication and branching sometimes work together. The last instruction, a conditional branch to attempt to reacquire the lock if necessary, also is predicated.

8.4.2 DIVERGENCE AND CONVERGENCE

Predication works well for small fragments of conditional code, especially `if` statements with no corresponding `else`. For larger amounts of conditional code, predication becomes inefficient because every instruction is executed, regardless of whether it will affect the computation. When the larger number of instructions causes the costs of predication to exceed the benefits, the compiler will use conditional branches. When the flow of execution within a warp takes different paths depending on a condition, the code is called *divergent*.

NVIDIA is close-mouthed about the details of how their hardware supports divergent code paths, and it reserves the right to change the hardware implementation between generations. The hardware maintains a bit vector of active threads within each warp. For threads that are marked inactive, execution is suppressed in a way similar to predication. Before taking a branch, the compiler executes a special instruction to push this active-thread bit vector onto a stack. The code is then executed *twice*, once for threads for which the condition was TRUE, then for threads for which the predicate was FALSE. This two-phased execution is managed with a *branch synchronization stack*, as described by Lindholm et al.[15]

> 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 reconverge to the original execution path. The SM uses a branch synchronization stack to manage independent threads that diverge and converge. Branch divergence only occurs within a warp; different warps execute independently regardless of whether they are executing common or disjoint code paths.

The PTX specification makes no mention of a branch synchronization stack, so the only publicly available evidence of its existence is in the disassembly output of `cuobjdump`. The `SSY` instruction pushes a state such as the program counter and active thread mask onto the stack; the `.S` instruction prefix pops this state

15. Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, March–April 2008, pp. 39–55.

and, if any active threads did not take the branch, causes those threads to execute the code path whose state was snapshotted by `SSY`.

`SSY/.S` is only necessary when threads of execution may diverge, so if the compiler can guarantee that threads will stay uniform in a code path, you may see branches that are not bracketed by `SSY/.S`. The important thing to realize about branching in CUDA is that in all cases, it is most efficient for all threads within a warp to follow the same execution path.

The loop in Listing 8.2 also includes a good self-contained example of divergence and convergence. The `SSY` instruction (offset 0x40) and `NOP.S` instruction (offset 0x78) bracket the points of divergence and convergence, respectively. The code loops over the `LDSLK` and subsequent predicated instructions, retiring active threads until the compiler knows that all threads will have converged and the branch synchronization stack can be popped with the `NOP.S` instruction.

```
/*0040*/ SSY 0x80;
/*0048*/ BAR.RED.POPC RZ, RZ;
/*0050*/ LD R0, [R0];
/*0058*/ LDSLK P0, R2, [R3];
/*0060*/ @P0 IADD R2, R2, R0;
/*0068*/ @P0 STSUL [R3], R2;
/*0070*/ @!P0 BRA 0x58;
/*0078*/ NOP.S CC.T;
```

8.4.3 SPECIAL CASES: MIN, MAX, AND ABSOLUTE VALUE

Some conditional operations are so common that they are supported natively by the hardware. Minimum and maximum operations are supported for both integer and floating-point operands and are translated to a single instruction. Additionally, floating-point instructions include modifiers that can negate or take the absolute value of a source operand.

The compiler does a good job of detecting when min/max operations are being expressed, but if you want to take no chances, call the `min()/max()` intrinsics for integers or `fmin()/fmax()` for floating-point values.

8.5 Textures and Surfaces

The instructions that read and write textures and surfaces refer to much more implicit state than do other instructions; parameters such as the base address, dimensions, format, and interpretation of the texture contents are contained in

a *header*, an intermediate data structure whose software abstraction is called a *texture reference* or *surface reference*. As developers manipulate the texture or surface references, the CUDA runtime and driver must translate those changes into the headers, which the texture or surface instruction references as an index.[16]

Before launching a kernel that operates on textures or surfaces, the driver must ensure that all this state is set correctly on the hardware. As a result, launching such kernels may take longer. Texture reads are serviced through a specialized cache subsystem that is separate from the L1/L2 caches in Fermi, and also separate from the constant cache. Each SM has an L1 texture cache, and the TPCs (texture processor clusters) or GPCs (graphics processor clusters) each additionally have L2 texture cache. Surface reads and writes are serviced through the same L1/L2 caches that service global memory traffic.

Kepler added two technologies of note with respect to textures: the ability to read from global memory via the texture cache hierarchy without binding a texture reference, and the ability to specify a texture header by address rather than by index. The latter technology is known as "bindless textures."

On SM 3.5 and later hardware, reading global memory via the texture cache can be requested by using `const __restrict` pointers or by explicitly invoking the `ldg()` intrinsics in `sm_35_intrinsics.h`.

8.6 Miscellaneous Instructions

8.6.1 WARP-LEVEL PRIMITIVES

It did not take long for the importance of warps as a primitive unit of execution (naturally residing between threads and blocks) to become evident to CUDA programmers. Starting with SM 1.x, NVIDIA began adding instructions that specifically operate on warps.

Vote

That CUDA architectures are 32-bit and that warps are comprised of 32 threads made an irresistible match to instructions that can evaluate a condition and

16. SM 3.x added *texture objects*, which enable texture and surface headers to be referenced by address rather than an index. Previous hardware generations could reference at most 128 textures or surfaces in a kernel, but with SM 3.x the number is limited only by memory.

broadcast a 1-bit result to every thread in the warp. The VOTE instruction (first available in SM 1.2) evaluates a condition and broadcasts the result to all threads in the warp. The `__any()` intrinsic returns 1 if the predicate is true for *any* of the 32 threads in the warp. The `__all()` intrinsic returns 1 if the predicate is true for *all* of the 32 threads in the warp.

The Fermi architecture added a new variant of VOTE that passes back the predicate result for every thread in the warp. The `__ballot()` intrinsic evaluates a condition for all threads in the warp and returns a 32-bit value where each bit gives the condition for the corresponding thread in the warp.

Shuffle

Kepler added *shuffle* instructions that enable data interchange between threads within a warp without staging the data through shared memory. Although these instructions execute with the same latency as shared memory, they have the benefit of doing the exchange without performing both a read and a write, and they can reduce shared memory usage.

The following instruction is wrapped in a number of device functions that use inline PTX assembly defined in `sm_30_intrinsics.h`.

```
int __shfl(int var, int srcLane, int width=32);
int __shfl_up(int var, unsigned int delta, int width=32);
int __shfl_down(int var, unsigned int delta, int width=32);
int __shfl_xor(int var, int laneMask, int width=32);
```

The `width` parameter, which defaults to the warp width of 32, must be a power of 2 in the range 2..32. It enables subdivision of the warp into segments; if `width<32`, each subsection of the warp behaves as a separate entity with a starting logical lane ID of 0. A thread may only exchange data with other threads in its subsection.

`__shfl()` returns the value of `var` held by the thread whose ID is given by `srcLane`. If `srcLane` is outside the range `0..width-1`, the thread's own value of `var` is returned. This variant of the instruction can be used to broadcast values within a warp. `__shfl_up()` calculates a source lane ID by subtracting `delta` from the caller's lane ID and clamping to the range `0..width-1`. `__shfl_down()` calculates a source lane ID by adding `delta` to the caller's lane ID.

`__shfl_up()` and `__shfl_down()` enable warp-level scan and reverse scan operations, respectively. `__shfl_xor()` calculates a source lane ID by performing a bitwise XOR of the caller's lane ID with `laneMask`; the value of `var` held by the resulting lane ID is returned. This variant can be used to do a

reduction across the warps (or subwarps); each thread computes the reduction using a differently ordered series of the associative operator.

8.6.2 BLOCK-LEVEL PRIMITIVES

The `__syncthreads()` intrinsic serves as a barrier. It causes all threads to wait until every thread in the threadblock has arrived at the `__syncthreads()`. The Fermi instruction set (SM 2.x) added several new block-level barriers that aggregate information about the threads in the threadblock.

- `__syncthreads_count()`: evaluates a predicate and returns the sum of threads for which the predicate was true
- `__syncthreads_or()`: returns the OR of all the inputs across the threadblock
- `__syncthreads_and()`: returns the AND of all the inputs across the threadblock

8.6.3 PERFORMANCE COUNTER

Developers can define their own set of performance counters and increment them in live code with the `__prof_trigger()` intrinsic.

```
void __prof_trigger(int counter);
```

Calling this function increments the corresponding counter by 1 per warp. `counter` must be in the range 0..7; counters 8..15 are reserved. The value of the counters may be obtained by listing `prof_trigger_00..prof_trigger_07` in the profiler configuration file.

8.6.4 VIDEO INSTRUCTIONS

The video instructions described in this section are accessible only via the inline PTX assembler. Their basic functionality is described here to help developers to decide whether they might be beneficial for their application. Anyone intending to use these instructions, however, should consult the PTX ISA specification.

Scalar Video Instructions

The scalar video instructions, added with SM 2.0 hardware, enable efficient operations on the short (8- and 16-bit) integer types needed for video

processing. As described in the PTX 3.1 ISA Specification, the format of these instructions is as follows.

```
vop.dtype.atype.btype{.sat} d, a{.asel}, b{.bsel};
vop.dtype.atype.btype{.sat}.secop d, a{.asel}, b{.bsel}, c;
```

The source and destination operands are all 32-bit registers. `dtype`, `atype`, and `btype` may be `.u32` or `.s32` for unsigned and signed 32-bit integers, respectively. The `asel/bsel` specifiers select which 8- or 16-bit value to extract from the source operands: `b0`, `b1`, `b2`, and `b3` select bytes (numbering from the least significant), and `h0/h1` select the least significant and most significant 16 bits, respectively.

Once the input values are extracted, they are sign- or zero-extended internally to signed 33-bit integers, and the primary operation is performed, producing a 34-bit intermediate result whose sign depends on `dtype`. Finally, the result is clamped to the output range, and one of the following operations is performed.

1. Apply a second operation (add, min or max) to the intermediate result and a third operand.
2. Truncate the intermediate result to an 8- or 16-bit value and merge into a specified position in the third operand to produce the final result.

The lower 32 bits are then written to the destination operand.

The `vset` instruction performs a comparison between the 8-, 16-, or 32-bit input operands and generates the corresponding predicate (1 or 0) as output. The PTX scalar video instructions and the corresponding operations are given in Table 8.14.

Table 8.14 Scalar Video Instructions.

MNEMONIC	OPERATION
vabsdiff	abs(a-b)
vadd	a+b
vavrg	(a+b)/2
vmad	a*b+c
vmax	max(a,b)

continues

Table 8.14 Scalar Video Instructions. *(Continued)*

MNEMONIC	OPERATION
vmin	min(a,b)
vset	Compare a and b
vshl	a<<b
vshr	a>>b
vsub	a-b

Vector Video Instructions (SM 3.0 only)

These instructions, added with SM 3.0, are similar to the scalar video instructions in that they promote the inputs to a canonical integer format, perform the core operation, and then clamp and optionally merge the output. But they deliver higher performance by operating on pairs of 16-bit values or quads of 8-bit values.

Table 8.15 summarizes the PTX instructions and corresponding operations implemented by these instructions. They are most useful for video processing and certain image processing operations (such as the median filter).

Table 8.15 Vector Video Instructions

MNEMONIC	OPERATION
vabsdiff[2|4]	abs(a-b)
vadd[2|4]	a+b
vavrg[2|4]	(a+b)/2
vmax[2|4]	max(a,b)
vmin[2|4]	min(a,b)
vset[2|4]	Compare a and b
vsub[2|4]	a-b

8.6.5 SPECIAL REGISTERS

Many special registers are accessed by referencing the built-in variables `threadIdx`, `blockIdx`, `blockDim`, and `gridDim`. These pseudo-variables, described in detail in Section 7.3, are 3-dimensional structures that specify the thread ID, block ID, thread count, and block count, respectively.

Besides those, another special register is the SM's clock register, which increments with each clock cycle. This counter can be read with the `__clock()` or `__clock64()` intrinsic. The counters are separately tracked for each SM and, like the time stamp counters on CPUs, are most useful for measuring relative performance of different code sequences and best avoided when trying to calculate wall clock times.

8.7 Instruction Sets

NVIDIA has developed three major architectures: Tesla (SM 1.x), Fermi (SM 2.x), and Kepler (SM 3.x). Within those families, new instructions have been added as NVIDIA updated their products. For example, global atomic operations were not present in the very first Tesla-class processor (the G80, which shipped in 2006 as the GeForce GTX 8800), but all subsequent Tesla-class GPUs included them. So when querying the SM version via `cuDeviceComputeCapability()`, the major and minor versions will be 1.0 for G80 and 1.1 (or greater) for all other Tesla-class GPUs. Conversely, if the SM version is 1.1 or greater, the application can use global atomics.

Table 8.16 gives the SASS instructions that may be printed by `cuobjdump` when disassembling microcode for Tesla-class (SM 1.x) hardware. The Fermi and Kepler instruction sets closely resemble each other, with the exception of the instructions that support surface load/store, so their instruction sets are given together in Table 8.17. In both tables, the middle column specifies the first SM version to support a given instruction.

Table 8.16 SM 1.x Instruction Set

OPCODE	SM	DESCRIPTION
FLOATING POINT		
COS	1.0	Cosine
DADD	1.3	Double-precision floating-point add
DFMA	1.3	Double-precision floating-point fused multiply-add
DMAX	1.3	Double-precision floating-point maximum
DMIN	1.3	Double-precision floating-point minimum
DMUL	1.3	Double-precision floating-point multiply
DSET	1.3	Double-precision floating-point condition set
EX2	1.0	Exponential (base 2)
FADD/FADD32/FADD32I	1.0	Single-precision floating-point add
FCMP	1.0	Single-precision floating-point compare
FMAD/FMAD32/FMAD32I	1.0	Single-precision floating-point multiply-add
FMAX	1.0	Single-precision floating-point maximum
FMIN	1.0	Single-precision floating-point minimum
FMUL/FMUL32/FMUL32I	1.0	Single-precision floating-point multiply
FSET	1.0	Single-precision floating-point conditional set
LG2	1.0	Single-precision floating-point logarithm (base 2)
RCP	1.0	Single-precision floating-point reciprocal
RRO	1.0	Range reduction operator (used before SIN/COS)
RSQ	1.0	Reciprocal square root
SIN	1.0	Sine

Table 8.16 SM 1.x Instruction Set *(Continued)*

OPCODE	SM	DESCRIPTION
FLOW CONTROL		
BAR	1.0	Barrier synchronization/ `__syncthreads()`
BRA	1.0	Conditional branch
BRK	1.0	Conditional break from loop
BRX	1.0	Fetch an address from constant memory and branch to it
C2R	1.0	Condition code to data register
CAL	1.0	Unconditional subroutine call
RET	1.0	Conditional return from subroutine
SSY	1.0	Set synchronization point; used before potentially divergent instructions
DATA CONVERSION		
F2F	1.0	Copy floating-point value with conversion to floating point
F2I	1.0	Copy floating-point value with conversion to integer
I2F	1.0	Copy integer value to floating-point with conversion
I2I	1.0	Copy integer value to integer with conversion
INTEGER		
IADD/ IADD32/ IADD32I	1.0	Integer addition
IMAD/ IMAD32/ IMAD32I	1.0	Integer multiply-add
IMAX	1.0	Integer maximum
IMIN	1.0	Integer minimum
IMUL/ IMUL32/ IMUL32I	1.0	Integer multiply
ISAD/ ISAD32	1.0	Integer sum of absolute difference

continues

Table 8.16 SM 1.x Instruction Set *(Continued)*

OPCODE	SM	DESCRIPTION
ISET	1.0	Integer conditional set
SHL	1.0	Shift left
SHR	1.0	Shift right
MEMORY OPERATIONS		
A2R	1.0	Move address register to data register
ADA	1.0	Add immediate to address register
G2R	1.0	Move from shared memory to register. The .LCK suffix, used to implement shared memory atomics, causes the bank to be locked until an R2G.UNL has been performed.
GATOM.IADD/ EXCH/ CAS/ IMIN/ IMAX/ INC/ DEC/ IAND/ IOR/ IXOR	1.2	Global memory atomic operations; performs an atomic operation and returns the original value.
GLD	1.0	Load from global memory
GRED.IADD/ IMIN/ IMAX/ INC/ DEC/ IAND/ IOR/ IXOR	1.2	Global memory reduction operations; performs an atomic operation with no return value.
GST	1.0	Store to global memory
LLD	1.0	Load from local memory
LST	1.0	Store to local memory
LOP	1.0	Logical operation (AND/OR/XOR)
MOV/ MOV32	1.0	Move source to destination
MVC	1.0	Move from constant memory
MVI	1.0	Move immediate
R2A	1.0	Move register to address register
R2C	1.0	Move data register to condition code
R2G	1.0	Store to shared memory. When used with the .UNL suffix, releases a previously held lock on that shared memory bank.

Table 8.16 SM 1.x Instruction Set *(Continued)*

OPCODE	SM	DESCRIPTION
MISCELLANEOUS		
NOP	1.0	No operation
TEX/ TEX32	1.0	Texture fetch
VOTE	1.2	Warp-vote primitive.
S2R	1.0	Move special register (e.g., thread ID) to register

Table 8.17 SM 2.x and SM 3.x Instruction Sets

OPCODE	SM	DESCRIPTION
FLOATING POINT		
DADD	2.0	Double-precision add
DMUL	2.0	Double-precision multiply
DMNMX	2.0	Double-precision minimum/maximum
DSET	2.0	Double-precision set
DSETP	2.0	Double-precision predicate
DFMA	2.0	Double-precision fused multiply-add
FFMA	2.0	Single-precision fused multiply-add
FADD	2.0	Single-precision floating-point add
FCMP	2.0	Single-precision floating-point compare
FMUL	2.0	Single-precision floating-point multiply
FMNMX	2.0	Single-precision floating-point minimum/maximum
FSWZ	2.0	Single-precision floating-point swizzle

continues

Table 8.17 SM 2.x and SM 3.x Instruction Sets *(Continued)*

OPCODE	SM	DESCRIPTION
FSET	2.0	Single-precision floating-point set
FSETP	2.0	Single-precision floating-point set predicate
MUFU	2.0	MultiFunk (SFU) operator
RRO	2.0	Range reduction operator (used before MUFU sin/cos)
INTEGER		
BFE	2.0	Bit field extract
BFI	2.0	Bit field insert
FLO	2.0	Find leading one
IADD	2.0	Integer add
ICMP	2.0	Integer compare and select
IMAD	2.0	Integer multiply-add
IMNMX	2.0	Integer minimum/maximum
IMUL	2.0	Integer multiply
ISAD	2.0	Integer sum of absolute differences
ISCADD	2.0	Integer add with scale
ISET	2.0	Integer set
ISETP	2.0	Integer set predicate
LOP	2.0	Logical operation (AND/OR/XOR)
SHF	3.5	Funnel shift
SHL	2.0	Shift left
SHR	2.0	Shift right
POPC	2.0	Population count

Table 8.17 SM 2.x and SM 3.x Instruction Sets *(Continued)*

OPCODE	SM	DESCRIPTION
DATA CONVERSION		
F2F	2.0	Floating point to floating point
F2I	2.0	Floating point to integer
I2F	2.0	Integer to floating point
I2I	2.0	Integer to integer
SCALAR VIDEO		
VABSDIFF	2.0	Scalar video absolute difference
VADD	2.0	Scalar video add
VMAD	2.0	Scalar video multiply-add
VMAX	2.0	Scalar video maximum
VMIN	2.0	Scalar video minimum
VSET	2.0	Scalar video set
VSHL	2.0	Scalar video shift left
VSHR	2.0	Scalar video shift right
VSUB	2.0	Scalar video subtract
VECTOR (SIMD) VIDEO		
VABSDIFF2(4)	3.0	Vector video 2x16-bit (4x8-bit) absolute difference
VADD2(4)	3.0	Vector video 2x16-bit (4x8-bit) addition
VAVRG2(4)	3.0	Vector video 2x16-bit (4x8-bit) average
VMAX2(4)	3.0	Vector video 2x16-bit (4x8-bit) maximum
VMIN2(4)	3.0	Vector video 2x16-bit (4x8-bit) minimum
VSET2(4)	3.0	Vector video 2x16-bit (4x8-bit) set
VSUB2(4)	3.0	Vector video 2x16-bit (4x8-bit) subtraction

continues

Table 8.17 SM 2.x and SM 3.x Instruction Sets *(Continued)*

OPCODE	SM	DESCRIPTION
DATA MOVEMENT		
MOV	2.0	Move
PRMT	2.0	Permute
SEL	2.0	Select (conditional move)
SHFL	3.0	Warp shuffle
PREDICATE/CONDITION CODES		
CSET	2.0	Condition code set
CSETP	2.0	Condition code set predicate
P2R	2.0	Predicate to register
R2P	2.0	Register to predicate
PSET	2.0	Predicate set
PSETP	2.0	Predicate set predicate
TEXTURE		
TEX	2.0	Texture fetch
TLD	2.0	Texture load
TLD4	2.0	Texture load 4 texels
TXQ	2.0	Texture query
MEMORY OPERATIONS		
ATOM	2.0	Atomic memory operation
CCTL	2.0	Cache control
CCTLL	2.0	Cache control (local)
LD	2.0	Load from memory

Table 8.17 SM 2.x and SM 3.x Instruction Sets *(Continued)*

OPCODE	SM	DESCRIPTION
LDC	2.0	Load constant
LDG	3.5	Noncoherence global load (reads via texture cache)
LDL	2.0	Load from local memory
LDLK	2.0	Load and lock
LDS	2.0	Load from shared memory
LDSLK	2.0	Load from shared memory and lock
LDU	2.0	Load uniform
LD_LDU	2.0	Combines generic load LD with a load uniform LDU
LDS_LDU	2.0	Combines shared memory load LDS with a load uniform LDU
MEMBAR	2.0	Memory barrier
RED	2.0	Atomic memory reduction operation
ST	2.0	Store to memory
STL	2.0	Store to local memory
STUL	2.0	Store and unlock
STS	2.0	Store to shared memory
STSUL	2.0	Store to shared memory and unlock
SURFACE MEMORY (FERMI)		
SULD	2.0	Surface load
SULEA	2.0	Surface load effective address
SUQ	2.0	Surface query
SURED	2.0	Surface reduction
SUST	2.0	Surface store

continues

Table 8.17 SM 2.x and SM 3.x Instruction Sets *(Continued)*

OPCODE	SM	DESCRIPTION
SURFACE MEMORY (KEPLER)		
SUBFM	3.0	Surface bit field merge
SUCLAMP	3.0	Surface clamp
SUEAU	3.0	Surface effective address
SULDGA	3.0	Surface load generic address
SUSTGA	3.0	Surface store generic address
FLOW CONTROL		
BRA	2.0	Branch to relative address
BPT	2.0	Breakpoint/trap
BRK	2.0	Break from loop
BRX	2.0	Branch to relative indexed address
CAL	2.0	Call to relative address
CONT	2.0	Continue in loop
EXIT	2.0	Exit program
JCAL	2.0	Call to absolute address
JMP	2.0	Jump to absolute address
JMX	2.0	Jump to absolute indexed address
LONGJMP	2.0	Long jump
PBK	2.0	Pre–break relative address
PCNT	2.0	Pre–continue relative address
PLONGJMP	2.0	Pre–long jump relative address
PRET	2.0	Pre–return relative address

Table 8.17 SM 2.x and SM 3.x Instruction Sets *(Continued)*

OPCODE	SM	DESCRIPTION
RET	2.0	Return from call
SSY	2.0	Set synchronization point; used before potentially divergent instructions
MISCELLANEOUS		
B2R	2.0	Barrier to register
BAR	2.0	Barrier synchronization
LEPC	2.0	Load effective program counter
NOP	2.0	No operation
S2R	2.0	Special register to register (used to read, for example, the thread or block ID)
VOTE	2.0	Query condition across warp

This page intentionally left blank

Index

D

E

F

G

H

I

K

O

P

Q

R

S

T

U

V

W

Z