Tuning CUDA™ Applications for Fermi™

Version 1.3

8/20/2010

Next-Generation CUDA Compute Architecture

Fermi™ is NVIDIA's next-generation CUDA™ compute architecture. The Fermi whitepaper [1] gives a detailed overview of the major improvements over the Tesla™ architecture introduced in 2006 with G80 and later revised with GT200 in 2008.[1]

Because the Fermi and Tesla architectures are both CUDA compute architectures, the programming model is the same for both, and applications that follow the best practices for the Tesla architecture should typically see speedups on the Fermi architecture without any code changes (particularly for computations that are limited by double-precision floating-point performance).

This document gives an overview on how to tune applications for Fermi to further increase these speedups. More details are available in the CUDA C Programming Guide 3.2 as noted throughout the document.

CUDA Best Practices

The performance guidelines and best practices described in the CUDA C Programming Guide [2] and the CUDA C Best Practices Guide [3] apply to all CUDA architectures. Programmers must primarily focus on following those recommendations to achieve the best performance.

The high-priority recommendations from those guides are as follows:

- Find ways to parallelize sequential code,
- Minimize data transfers between the host and the device,
- Adjust kernel launch configuration to maximize device utilization,
- Ensure global memory accesses are coalesced,
- Replace global memory accesses with shared memory accesses whenever possible,
- Avoid different execution paths within the same warp.

[1] Here "Tesla" refers to the Tesla CUDA compute architecture rather the Tesla brand of GPUs targeted to high performance computing. There are Tesla-branded products for both the Tesla and Fermi architectures.

Application Compatibility

Before addressing the specific performance tuning issues covered in this guide, developers should refer to the *Fermi Compatibility Guide for CUDA Applications* to ensure that their applications are being compiled in a way that will be compatible with Fermi.

Fermi Tuning

All section references are to sections in the CUDA C Programming Guide 3.2.

Device Utilization

The only way to utilize all multiprocessors in a device of compute capability 1.x is to launch a single kernel with at least as many thread blocks as there are multiprocessors in the device (Section 5.2.2). Applications have more flexibility on devices of compute capability 2.x, since these devices can execute multiple kernels concurrently (Section 3.2.7.3) and therefore allow applications to also fill the device with several smaller kernel launches as opposed to a single larger one. This is done using CUDA streams (Section 3.2.7.5).

L1 Cache

Devices of compute capability 2.x come with an L1/L2 cache hierarchy that is used to cache local and global memory accesses. Programmers have some control over L1 caching:

- The same on-chip memory is used for both L1 and shared memory, and how much of it is dedicated to L1 versus shared memory is configurable for each kernel call (Section G.4.1);
- Global memory caching in L1 can be disabled at compile time (Section G.4.2);
- Local memory caching in L1 cannot be disabled (Section 5.3.2.2), but programmers can control local memory usage by limiting the amount of variables that the compiler is likely to place in local memory (Section 5.3.2.2) and by controlling register spilling via the **`__launch_bounds()__`** attribute (Section B.17) or the **`-maxrregcount`** compiler option.

Experimentation is recommended to find out the best combination for a given kernel: 16 KB or 48 KB of L1 cache (and vice versa for shared memory) with or without global memory caching in L1 and with more or less local memory usage. Kernels that use a lot of local memory (e.g., for register spilling (Section 5.3.2.2)) could benefit from 48 KB of L1 cache.

On devices of compute capability 1.x, some kernels can achieve a speedup when using (cached) texture fetches rather than regular global memory loads (e.g., when the regular loads do not coalesce well). Unless texture fetches provide other benefits such as address calculations or texture filtering (Section 5.3.2.5), this optimization can be counter-productive on devices of compute capability 2.x, however, since global memory loads are cached in L1 and the L1 cache has higher bandwidth than the texture cache.

Global Memory

On devices of compute capability 1.x, global memory accesses are processed per half-warp; on devices of compute capability 2.x, they are processed per warp. Adjusting kernel launch

configurations that assume per-half-warp accesses might therefore improve performance. Two-dimensional thread blocks, for example, should have their x-dimension be a multiple of the warp size as opposed to half the warp size so that each warp addresses a single cache line when accessing global memory.

Fermi-based GPUs of the Tesla and Quadro product lines (e.g., Tesla C2050) support ECC error correction, and ECC is enabled by default. While necessary for many scientific and professional applications, ECC comes at the cost of reduced peak memory bandwidth. With ECC enabled, effective peak bandwidth for global memory accesses is reduced by approximately 20% (more for scattered writes). If ECC is not needed, it can be disabled for improved performance using the `nvidia-smi` utility (or via Control Panel on Microsoft Windows systems). Note that toggling ECC on or off requires a reboot to take effect.

Shared Memory

The multiprocessor of devices of compute capability 2.x can be configured to have 48 KB of shared memory (Section G.4.1), which is three times more than in devices of compute capability 1.x. Kernels for which shared memory size is a performance limiter can benefit from this, either by increasing the number of resident blocks per multiprocessor (Section 4.2), or by increasing the amount of shared memory per thread block and adjusting the kernel launch configuration accordingly.

On devices of compute capability 1.x, shared memory has 16 banks and accesses are processed per half-warp; on devices of compute capability 2.x, shared memory has 32 banks and accesses are processed per warp (Section G.4.3). Bank conflicts can therefore occur between threads belonging to the first half of a warp and threads belonging to the second half of the same warp on devices of compute capability 2.x, and data layout in shared memory (e.g. padded arrays) might need to be adjusted to avoid bank conflicts.

The shared memory hardware is improved on devices of compute capability 2.x to support multiple broadcast words and to generate fewer bank conflicts for accesses of 8-bits, 16-bits, 64-bits, or 128-bits per thread (Section G.4.3).

Constant Cache

In addition to the constant memory space supported by devices of all compute capabilities (where **`__constant__`** variables reside), devices of compute capability 2.x support the LDU (LoaD Uniform) instruction that the compiler can use to load any variable via the constant cache under certain circumstances (Section G.4.4).

32-Bit Integer Multiplication

On devices of compute capability 1.x, 32-bit integer multiplication is implemented using multiple instructions as it is not natively supported. 24-bit integer multiplication is natively supported via the **`__[u]mul24`** intrinsic.

On devices of compute capability 2.x, however, 32-bit integer multiplication is natively supported, but 24-bit integer multiplication is not. **`__[u]mul24`** is therefore implemented using multiple instructions and should not be used (Section 5.4.1).

IEEE 754-2008 Compliance

Devices of compute capability 2.x have far fewer deviations from the IEEE 754-2008 floating point standard than devices of compute capability 1.x, particularly in single precision (Section G.2). This can cause slight changes in numeric results between devices of compute capability 1.x and devices of compute capability 2.x.

In particular, addition and multiplication are often combined into an FMAD instruction on devices of compute capability 1.x, which is not IEEE compliant, whereas they would be combined into an FFMA instruction on devices of compute capability 2.x, which *is* IEEE compliant.

By default n**vcc** generates IEEE compliant code, but it also provides options to generate code for devices of compute capability 2.x that is closer to the code generated for earlier devices: **`-ftz=true`** (denormalized numbers are flushed to zero), **`-prec-div=false`** (less precise division), and **`-prec-sqrt=false`** (less precise square root). Code compiled this way tends to have higher performance than code compiled with the default settings (Section 5.4.1).

C++ Support

Devices of compute capability 2.x support C++ classes with some restrictions, which are detailed in Section D.6 of the programming guide.

Advanced Functions

Devices of compute capability 2.x support additional advanced intrinsic functions:

- Another memory fence function: `__threadfence_system()` (Section B.5),
- Variants of the `__syncthreads()` synchronization function: `__syncthreads_count()`, `__syncthreads_and()`, and `__syncthreads_or()` (Section B.6),
- Surface functions (Section B.9),
- Floating-point atomic addition operating on 32-bit words in global and shared memory (Section B.11.1.1),
- Another warp vote function: `__ballot()` (Section B.12).

Atomic functions have higher throughput on devices of compute capability 2.x than on earlier devices.

References

[1] NVIDIA's Next Generation CUDA Compute Architecture: Fermi
http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIAFermiComputeArchitectureWhitepaper.pdf

[2] NVIDIA CUDA C Programming Guide from CUDA Toolkit 3.2

[3] NVIDIA CUDA C Best Practices Guide from CUDA Toolkit 3.2

[4] NVIDIA CUDA Reference Manual from CUDA Toolkit 3.2

Appendix A. Revision History

Version 1.0

- Initial public release.

Version 1.1

- Clarified discussion of 32-bit versus 64-bit device code.

Version 1.2

- For CUDA Driver API applications, the use of 32-bit device code within a 64-bit host application is no longer recommended, as support for this mixed-bitness mode will be removed in CUDA Toolkit 3.2. (Note that the CUDA Runtime API already requires that the device code and host code have matching bitness.)

Version 1.3

- Added reference to surface functions under *Advanced Functions*.
- Updated cross-references to CUDA C Programming Guide version 3.2.
- Updated references to compute capability 2.0 to 2.x.
- Added discussion of ECC to Global Memory section.

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