TUNING CUDA APPLICATIONS FOR MAXWELL

DA-07173-001_v11.0 | August 2020

Application Note

TABLE OF CONTENTS

Chapter 1. MAXWELL TUNING GUIDE

1.1. NVIDIA Maxwell Compute Architecture

Maxwell is NVIDIA's next-generation architecture for CUDA compute applications. Maxwell retains and extends the same CUDA programming model as in previous NVIDIA architectures such as Fermi and Kepler, and applications that follow the best practices for those architectures should typically see speedups on the Maxwell architecture without any code changes. This guide summarizes the ways that an application can be fine-tuned to gain additional speedups by leveraging Maxwell architectural features.[1]

Maxwell introduces an all-new design for the Streaming Multiprocessor (*SM*) that dramatically improves energy efficiency. Although the Kepler SMX design was extremely efficient for its generation, through its development, NVIDIA's GPU architects saw an opportunity for another big leap forward in architectural efficiency; the Maxwell SM is the realization of that vision. Improvements to control logic partitioning, workload balancing, clock-gating granularity, compiler-based scheduling, number of instructions issued per clock cycle, and many other enhancements allow the Maxwell SM (also called *SMM*) to far exceed Kepler SMX efficiency.

The first Maxwell-based GPU is codenamed *GM107* and is designed for use in power-limited environments like notebooks and small form factor (SFF) PCs. GM107 is described in a whitepaper entitled NVIDIA GeForce GTX 750 Ti: Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt.[2]

The first GPU using the second-generation Maxwell architecture is codenamed *GM204*. Second-generation Maxwell GPUs retain the power efficiency of the earlier generation while delivering significantly higher performance. GM204 is described in a whitepaper entitled *NVIDIA GeForce GTX 980: Featuring Maxwell, The Most Advanced GPU Ever Made*.

[1] Throughout this guide, *Fermi* refers to devices of compute capability 2.x, *Kepler* refers to devices of compute capability 3.x, and *Maxwell* refers to devices of compute capability 5.x.
[2] The features of GM108 are similar to those of GM107.

Compute programming features of GM204 are similar to those of GM107, except where explicitly noted in this guide. For details on the programming features discussed in this guide, please refer to the CUDA C++ Programming Guide.

1.2. CUDA Best Practices

The performance guidelines and best practices described in the CUDA C++ Programming Guide and the CUDA C++ Best Practices Guide apply to all CUDA-capable GPU 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,
- Minimize redundant accesses to global memory whenever possible,
- Avoid long sequences of diverged execution by threads within the same warp.

1.3. Application Compatibility

Before addressing specific performance tuning issues covered in this guide, refer to the Maxwell Compatibility Guide for CUDA Applications to ensure that your application is compiled in a way that is compatible with Maxwell.

1.4. Maxwell Tuning

1.4.1. SMM

The Maxwell Streaming Multiprocessor, SMM, is similar in many respects to the Kepler architecture's SMX. The key enhancements of SMM over SMX are geared toward improving efficiency without requiring significant increases in available parallelism per SM from the application.

1.4.1.1. Occupancy

The maximum number of concurrent warps per SMM remains the same as in SMX (i.e., 64), and factors influencing warp occupancy remain similar or improved over SMX:

- The register file size (64k 32-bit registers) is the same as that of SMX.
- The maximum registers per thread, 255, matches that of Kepler GK110. As with Kepler, experimentation should be used to determine the optimum balance of register spilling vs. occupancy, however.
- The maximum number of thread blocks per SM has been increased from 16 to 32. This should result in an automatic occupancy improvement for kernels with small

thread blocks of 64 or fewer threads (shared memory and register file resource requirements permitting). Such kernels would have tended to under-utilize SMX, but less so SMM.

- Shared memory capacity is increased (see Shared Memory Capacity).

As such, developers can expect similar or improved occupancy on SMM without changes to their application. At the same time, warp occupancy requirements (i.e., available parallelism) for maximum device utilization are similar to or less than those of SMX (see Instruction Latencies).

1.4.1.2. Instruction Scheduling

The number of CUDA Cores per SM has been reduced to a power of two, however with Maxwell's improved execution efficiency, performance per SM is usually within 10% of Kepler performance, and the improved area efficiency of SMM means CUDA Cores per GPU will be substantially higher vs. comparable Fermi or Kepler chips. SMM retains the same number of instruction issue slots per clock and reduces arithmetic latencies compared to the Kepler design.

As with SMX, each SMM has four warp schedulers. Unlike SMX, however, all SMM core functional units are assigned to a particular scheduler, with no shared units. Along with the selection of a power-of-two number of CUDA Cores per SM, which simplifies scheduling and reduces stall cycles, this partitioning of SM computational resources in SMM is a major component of the streamlined efficiency of SMM.

The power-of-two number of CUDA Cores per partition simplifies scheduling, as each of SMM's warp schedulers issue to a dedicated set of CUDA Cores equal to the warp width. Each warp scheduler still has the flexibility to dual-issue (such as issuing a math operation to a CUDA Core in the same cycle as a memory operation to a load/store unit), but single-issue is now sufficient to fully utilize all CUDA Cores.

1.4.1.3. Instruction Latencies

Another major improvement of SMM is that dependent math latencies have been significantly reduced; a consequence of this is a further reduction of stall cycles, as the available warp-level parallelism (i.e., occupancy) on SMM should be equal to or greater than that of SMX (see Occupancy), while at the same time each math operation takes *less* time to complete, improving utilization and throughput.

1.4.1.4. Instruction Throughput

The most significant changes to peak instruction throughputs in SMM are as follows:

- The change in number of CUDA Cores per SM brings with it a corresponding change in peak single-precision floating point operations per clock per SM. However, since the number of SMs is typically increased, the result is an increase in aggregate peak throughput; furthermore, the scheduling and latency improvements also discussed above make this peak easier to approach.
- The throughput of many integer operations including multiply, logical operations and shift is improved. In addition, there are now specialized integer instructions

that can accelerate pointer arithmetic. These instructions are most efficient when data structures are a power of two in size.

As was already the recommended best practice, signed arithmetic should be preferred over unsigned arithmetic wherever possible for best throughput on SMM. The C language standard places more restrictions on overflow behavior for unsigned math, limiting compiler optimization opportunities.

1.4.2. Memory Throughput

1.4.2.1. Unified L1/Texture Cache

Maxwell combines the functionality of the L1 and texture caches into a single unit.

As with Kepler, global loads in Maxwell are cached in L2 only, unless using the *LDG* read-only data cache mechanism introduced in Kepler.

In a manner similar to Kepler GK110B, GM204 retains this behavior by default but also allows applications to opt-in to caching of global loads in its unified L1/Texture cache. The opt-in mechanism is the same as with GK110B: pass the **`-Xptxas -dlcm=ca`** flag to **`nvcc`** at compile time.

Local loads also are cached in L2 only, which could increase the cost of register spilling if L1 local load hit rates were high with Kepler. The balance of occupancy versus spilling should therefore be reevaluated to ensure best performance. Especially given the improvements to arithmetic latencies, code built for Maxwell may benefit from somewhat lower occupancy (due to increased registers per thread) in exchange for lower spilling.

The unified L1/texture cache acts as a coalescing buffer for memory accesses, gathering up the data requested by the threads of a warp prior to delivery of that data to the warp. This function previously was served by the separate L1 cache in Fermi and Kepler.

Two new device attributes were added in CUDA Toolkit 6.0: **`globalL1CacheSupported`** and **`localL1CacheSupported`**. Developers who wish to have separately-tuned paths for various architecture generations can use these fields to simplify the path selection process.

Enabling caching of globals in GM204 can affect occupancy. If per-thread-block SM resource usage would result in zero occupancy with caching enabled, the CUDA driver will override the caching selection to allow the kernel launch to succeed. This situation is reported by the profiler.

1.4.3. Shared Memory

1.4.3.1. Shared Memory Capacity

With Fermi and Kepler, shared memory and the L1 cache shared the same on-chip storage. Maxwell, by contrast, provides dedicated space to the shared memory of each SMM, since the functionality of the L1 and texture caches have been merged in SMM.

This increases the shared memory space available per SMM as compared to SMX: GM107 provides 64 KB shared memory per SMM, and GM204 further increases this to 96 KB shared memory per SMM.

This presents several benefits to application developers:

- Algorithms with significant shared memory capacity requirements (e.g., radix sort) see an automatic 33% to 100% boost in capacity per SM on top of the aggregate boost from higher SM count.
- Applications no longer need to select a preference of the L1/shared split for optimal performance. For purposes of backward compatibility with Fermi and Kepler, applications may optionally continue to specify such a preference, but the preference will be ignored on Maxwell, with the full 64 KB per SMM always going to shared memory.

While the per-SM shared memory capacity is increased in SMM, the per-thread-block limit remains 48 KB. For maximum flexibility on possible future GPUs, NVIDIA recommends that applications use at most 32 KB of shared memory in any one thread block, which would for example allow at least two such thread blocks to fit per SMM.

1.4.3.2. Shared Memory Bandwidth

Kepler SMX introduced an optional 8-byte shared memory banking mode, which had the potential to increase shared memory bandwidth per SM over Fermi for shared memory accesses of 8 or 16 bytes. However, applications could only benefit from this when storing these larger elements in shared memory (i.e., integers and fp32 values saw no benefit), and only when the developer explicitly opted into the 8-byte bank mode via the API.

To simplify this, Maxwell returns to the Fermi style of shared memory banking, where banks are always four bytes wide. Aggregate shared memory bandwidth across the chip remains comparable to that of corresponding Kepler chips, given increased SM count. In this way, all applications using shared memory can now benefit from the higher bandwidth, even when storing only four-byte items into shared memory and without specifying any particular preference via the API.

1.4.3.3. Fast Shared Memory Atomics

Kepler introduced a dramatically higher throughput for atomic operations to *global* memory as compared to Fermi. However, atomic operations to *shared* memory remained essentially unchanged: both architectures implemented shared memory atomics using a lock/update/unlock pattern that could be expensive in the case of high contention for updates to particular locations in shared memory.

Maxwell improves upon this by implementing native shared memory atomic operations for 32-bit integers and native shared memory 32-bit and 64-bit compare-and-swap

(CAS), which can be used to implement other atomic functions with reduced overhead compared to the Fermi and Kepler methods.

Refer to the CUDA C++ Programming Guide for an example implementation of an fp64 `atomicAdd()` using `atomicCAS()`.

1.4.4. Dynamic Parallelism

GK110 introduced a new architectural feature called Dynamic Parallelism, which allows the GPU to create additional work for itself. A programming model enhancement leveraging this feature was introduced in CUDA 5.0 to enable kernels running on GK110 to launch additional kernels onto the same GPU.

SMM brings Dynamic Parallelism into the mainstream by supporting it across the product line, even in lower-power chips such as GM107. This will benefit developers, as it means that applications will no longer need special-case algorithm implementations for high-end GPUs that differ from those usable in more power-constrained environments.

Appendix A. REVISION HISTORY

Version 1.0

- Initial Public Release

Version 1.1

- Updated for second-generation Maxwell (compute capability 5.2).

Version 1.2

- Updated references to the CUDA C++ Programming Guide and CUDA C++ Best Practices Guide.

Notice

Trademarks

Copyright