TUNING CUDA APPLICATIONS FOR KEPLER

DA-06288-001_v5.5 for POWER8 | August 2014

Application Note

TABLE OF CONTENTS

Chapter 1. KEPLER TUNING GUIDE

1.1. NVIDIA Kepler Compute Architecture

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

The Kepler architecture comes in two variants, GK104 and GK110. A detailed overview of the major improvements in GK104[2] and GK110 over the earlier Fermi architecture are described in a pair of whitepapers [1][2] entitled *NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built* for GK104 and *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110* for GK110.

For details on the programming features discussed in this guide, please refer to the *CUDA C Programming Guide 5.0*. Details on the architectural features are covered in the architecture whitepapers referenced above. Some of the Kepler features described in this guide are specific to GK110, as noted; if not specified, Kepler features refer to both GK104 and GK110.

1.2. CUDA Best Practices

The performance guidelines and best practices described in the *CUDA C Programming Guide* [3] and the *CUDA C Best Practices Guide* [4] 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:

[1] Throughout this guide, *Fermi* refers to devices of compute capability 2.x and *Kepler* refers to devices of compute capability 3.x. GK104 has compute capability 3.0; GK110 has compute capability 3.5.

[2] The features of GK107 are similar to those of GK104.

- 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 different execution paths within the same warp.

1.3. Application Compatibility

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

Note that many of the GK110 architectural features described in this document require the device code in the application to be compiled for its native compute capability 3.5 target architecture (**`sm_35`**).

1.4. Kepler Tuning

1.4.1. Device Utilization and Occupancy

Kepler's new Streaming Multiprocessor, called SMX, has significantly more CUDA Cores than the SM of Fermi GPUs, yielding a throughput improvement of 2-3x per clock.[3] Furthermore, GK110 has increased memory bandwidth over Fermi and GK104. To match these throughput increases, we need roughly twice as much parallelism per multiprocessor on Kepler GPUs, via either an increased number of active warps of threads or increased instruction-level parallelism (ILP) or some combination thereof.

Balancing this is the fact that GK104 ships with only 8 multiprocessors, half of the size of Fermi GF110, meaning that GK104 needs roughly the same total amount of parallelism as is needed by Fermi GF110, though it needs more parallelism per multiprocessor to achieve this. Since GK110 can have up to 15 multiprocessors, which is similar to the number of multiprocessors of Fermi GF110, then GK110 typically needs a larger amount of parallelism than Fermi or GK104.

To enable the increased per-multiprocessor warp occupancy beneficial to both GK104 and GK110, several important multiprocessor resources have been significantly increased in SMX:

- Kepler increases the size of the register file over Fermi by 2x per multiprocessor. On Fermi, the number of registers available was the primary limiting factor of occupancy for many kernels. On Kepler, these kernels can automatically fit more thread blocks per multiprocessor. For example, a kernel using 63 registers per thread and 256 threads per block can fit at most 16 concurrent warps per multiprocessor on Fermi (out of a maximum of 48, i.e., 33% theoretical occupancy). The same

[3] Note, however, that Kepler clocks are generally lower than Fermi clocks for improved power efficiency.

configuration can fit 32 warps per multiprocessor on Kepler (out of a maximum of 64, i.e., 50% theoretical occupancy).

- Kepler has increased the maximum number of simultaneous blocks per multiprocessor from 8 to 16. As a result, kernels having their occupancy limited due to reaching the maximum number of thread blocks per multiprocessor will see increased theoretical occupancy in Kepler.

Note that these automatic occupancy improvements require kernel launches with sufficient total thread blocks to fill Kepler. For this reason, it remains a best practice to launch kernels with significantly more thread blocks than necessary to fill current GPUs, allowing this kind of scaling to occur naturally without modifications to the application. The *CUDA Occupancy Calculator* [5] spreadsheet is a valuable tool in visualizing the achievable occupancy for various kernel launch configurations

Also note that Kepler GPUs can utilize ILP in place of thread/warp-level parallelism (TLP) more readily than Fermi GPUs can. Furthermore, some degree of ILP in conjunction with TLP is *required* by Kepler GPUs in order to approach peak performance, since SMX's warp scheduler issues one or two independent instructions from each of four warps per clock.

1.4.2. Managing Coarse-Grained Parallelism

Since GK110 requires more concurrently active threads than either GK104 or Fermi, GK110 introduces several features that can assist applications having more limited parallelism, where the expanded multiprocessor resources described in Device Utilization and Occupancy are difficult to leverage from any single kernel launch. These improvements allow the application to more readily use several concurrent kernel grids to fill GK110:

1.4.2.1. Concurrent Kernels

Since the introduction of Fermi, applications have had the ability to launch several kernels concurrently. This provides a mechanism by which applications can fill the device with several smaller kernel launches simultaneously as opposed to a single larger one. On Fermi and on GK104, at most 16 kernels can execute concurrently; GK110 allows up to 32 concurrent kernels to execute, which can provide a speedup for applications with necessarily small (but independent) kernel launches.

1.4.2.2. Hyper-Q

GK110 further improves this with the addition of Hyper-Q, which removes the false dependencies that can be introduced among CUDA streams in cases of suboptimal kernel launch or memory copy ordering across streams in Fermi or GK104. Hyper-Q allows GK110 to handle the 32 concurrent kernels in separate CUDA streams truly independently, rather than serializing the several streams into a single work queue at the hardware level. This allows applications to enqueue work into separate CUDA streams without considering the relative order of insertion of otherwise independent work, making concurrency of multiple kernels as well as overlapping of memory copies with computation much more readily achievable on GK110.

1.4.2.3. Dynamic Parallelism

GK110 also introduces a new architectural feature called Dynamic Parallelism, which allows the GPU to create additional work for itself. CUDA 5.0 introduces a programming model enhancement that leverages this architectural feature to enable kernels running on GK110 to launch additional kernels onto the same GPU. Nested kernel launches are done via the same **<<<>>>** triple-angle bracket notation used from the host and can make use of the familiar CUDA streams interface to specify whether or not the kernels launched are independent of one another. More than one GPU thread can simultaneously launch kernel grids (of the same or different kernels), further increasing the application's flexibility in keeping the GPU filled with parallel work.

1.4.3. Shared Memory and Warp Shuffle

1.4.3.1. Shared Memory Bandwidth

In balance with the increased computational throughput in Kepler's SMX described in Device Utilization and Occupancy, shared memory bandwidth in SMX is twice that of Fermi's SM. This bandwidth increase is exposed to the application through a configurable new 8-byte shared memory bank mode. When this mode is enabled, 64-bit (8-byte) shared memory accesses (such as loading a double-precision floating point number from shared memory) achieve twice the effective bandwidth of 32-bit (4-byte) accesses. Applications that are sensitive to shared memory bandwidth can benefit from enabling this mode as long as their kernels' accesses to shared memory are for 8-byte entities wherever possible.

1.4.3.2. Shared Memory Capacity

Fermi introduced an L1 cache in addition to the shared memory available since the earliest CUDA-capable GPUs. In Fermi, the shared memory and the L1 cache share the same physical on-chip storage, and a split of 48 KB shared memory / 16 KB L1 cache or vice versa can be selected per application or per kernel launch. Kepler continues this pattern and introduces an additional setting of 32 KB shared memory / 32 KB L1 cache, the use of which may benefit L1 hit rate in kernels that need more than 16 KB but less than 48 KB of shared memory per multiprocessor.

Since the maximum shared memory capacity per multiprocessor remains 48 KB, however, applications that depend on shared memory capacity either at a per-block level for data exchange or at a per-thread level for additional thread-private storage may require some rebalancing on Kepler to improve their achievable occupancy. The Warp Shuffle operation for data-exchange uses of shared memory and the Increased Addressable Registers Per Thread as an alternative to thread-private uses of shared memory present two possible alternatives to achieve this rebalancing.

1.4.3.3. Warp Shuffle

Kepler introduces a new warp-level intrinsic called the *shuffle* operation. This feature allows the threads of a warp to exchange data with each other directly without going through shared (or global) memory. The shuffle instruction also has lower latency than

shared memory access and does not consume shared memory space for data exchange, so this can present an attractive way for applications to rapidly interchange data among threads.

1.4.4. Memory Throughput

1.4.4.1. Increased Addressable Registers Per Thread

GK110 increases the maximum number of registers addressable per thread from 63 to 255. This can improve performance of bandwidth-limited kernels that have significant register spilling on Fermi or GK104. Experimentation should be used to determine the optimum balance of spilling vs. occupancy, however, as significant increases in the number of registers used per thread naturally decreases the warp occupancy that can be achieved, which trades off latency due to memory traffic for arithmetic latency due to fewer concurrent warps.

1.4.4.2. L1 Cache

L1 caching in Kepler GPUs is reserved only for local memory accesses, such as register spills and stack data. Global loads are cached in L2 only (or in the Read-Only Data Cache).

1.4.4.3. Read-Only Data Cache

GK110 adds the ability for read-only data in global memory to be loaded through the same cache used by the texture pipeline via a standard pointer without the need to bind a texture beforehand and without the sizing limitations of standard textures. Since this is a separate cache with a separate memory pipe and with relaxed memory coalescing rules, use of this feature can benefit the performance of bandwidth-limited kernels. This feature will be automatically enabled and utilized where possible by the compiler when compiling for GK110 as long as certain conditions are met. Foremost among these requirements is that the data *must* be guaranteed read-only *for the duration of the kernel*, as the read-only data cache is incoherent with respect to writes. In order to allow the compiler to detect that this condition is satisfied, a necessary (but not always sufficient) condition is that pointers used for loading such data should be marked with both the **`const`** *and* **`__restrict__`** qualifiers. Note that adding these qualifiers where applicable can improve code generation quality via other mechanisms on earlier GPUs as well.

In cases where more explicit control over the read-only data cache mechanism is desired than the **`const __restrict__`** qualifiers provide, or where the code is sufficiently complex that the compiler is unable to detect that the read-only data cache is safe to use, the **`__ldg()`** intrinsic can be used in place of a normal pointer dereference to force the load to go through the read-only data cache.

Note that the read-only data cache accessed via **`const __restrict__`** is separate and distinct from the constant cache acessed via the **`__constant__`** qualifier. Data loaded through the constant cache must be relatively small and must be accessed uniformly for good performance (i.e., all threads of a warp should access the same location at any given time), whereas data loaded through the read-only data cache can be much

larger and can be accessed in a non-uniform pattern. These two data paths can be used simultaneously for different data if desired.

1.4.4.4. Fast Global Memory Atomics

Global memory atomic operations have dramatically higher throughput on Kepler than on Fermi. Algorithms requiring multiple threads to update the same location in memory concurrently have at times on earlier GPUs resorted to complex data rearrangements in order to minimize the number of atomics required. Given the improvements in global memory atomic performance, many atomics can be performed on Kepler nearly as quickly as memory loads. This may simplify implementations requiring atomicity or enable algorithms previously deemed impractical.

1.4.4.5. 2D Memory Copies

The effective bandwidth of **`cudaMemcpy2D()`** operations is best when avoiding the use of small device pitches together with large host pitches (>64 KB).

1.4.5. Instruction Throughput

While the maximum instructions per clock (IPC) of both floating-point and integer operations has been either increased or maintained in Kepler as compared to Fermi, the relative ratios of maximum IPC for various specific instructions has changed somewhat. Refer to the *CUDA C Programming Guide* for details.

1.4.5.1. Single-precision vs. Double-precision

As one example of these instruction throughput ratios, an important difference between GK104 and GK110 is the ratio of peak single-precision to peak double-precision floating point performance. Whereas GK104 focuses primarily on high single-precision throughput, GK110 significantly improves the peak double-precision throughput over Fermi as well. Applications that depend heavily on high double-precision performance will generally perform best with GK110.

1.4.6. Multi GPU

NVIDIA's Tesla K10 GPU Accelerator is a dual-GK104 *Gemini* board. As with other dual-GPU NVIDIA boards, the two GPUs on the board will appear as two separate CUDA devices; they have separate memories and operate independently. As such, applications that will target the Tesla K10 GPU Accelerator but that are not yet multi-GPU aware should begin preparing for the multi-GPU paradigm. Since dual-GPU boards appear to the host application exactly the same as two separate single-GPU boards, enabling applications for multi-GPU can benefit application performance on a wide range of systems where more than one CUDA-capable GPU can be installed.

1.4.7. PCIe 3.0

Kepler's interconnection to the host system has been enhanced to support PCIe 3.0. For applications where host-to-device, device-to-host, or device-to-device transfer time is significant and not easily overlapped with computation, the additional bandwidth

provided by PCIe 3.0, given the requisite host system support, over the earlier PCIe 2.0 specification supported by Fermi GPUs should boost application performance without modifications to the application. Note that best PCIe transfer speeds to or from system memory with either PCIe generation are achieved when using pinned system memory.

Note that in the Tesla K10 GPU Accelerator, the two GPUs sharing a board are connected via an on-board PCIe 3.0 switch. Since these GPUs are also capable of GPUDirect Peer-to-Peer transfers, the inter-device memory transfers between GPUs on the same board can run at PCIe 3.0 speeds even if the host system supports only PCIe 2.0 or earlier.

1.4.8. Warp-synchronous Programming

As a means of mitigating the cost of repeated block-level synchronizations, particularly in parallel primitives such as reduction and prefix sum, some programmers exploit the knowledge that threads in a warp execute in lock-step with each other to omit **`__syncthreads()`** in some places where it is semantically necessary for correctness in the CUDA programming model.

The absence of an explicit synchronization in a program where different threads communicate via memory constitutes a data race condition or synchronization error. Warp-synchronous programs are unsafe and easily broken by evolutionary improvements to the optimization strategies used by the CUDA compiler toolchain, which generally has no visibility into cross-thread interactions of this variety in the absence of barriers, or by changes to the hardware memory subsystem's behavior. Such programs also tend to assume that the warp size is 32 threads, which may not necessarily be the case for all future CUDA-capable architectures.

Therefore, programmers should avoid warp-synchronous programming to ensure future-proof correctness in CUDA applications. When threads in a block must communicate or synchronize with each other, regardless of whether those threads are expected to be in the same warp or not, the appropriate barrier primitives should be used. Note that the Warp Shuffle primitive presents a future-proof, supported mechanism for intra-warp communication that can safely be used as an alternative in many cases.

1.5. References

[1] *NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built*.

http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[2] *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*.

http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[3] *CUDA C Programming Guide* from CUDA Toolkit 5.0.

[4] *CUDA C Best Practices Guide* from CUDA Toolkit 5.0.

[5] *CUDA Occupancy Calculator* spreadsheet from CUDA Toolkit 5.0.

Appendix A. REVISION HISTORY

Version 0.9

- CUDA 5.0 Preview Release

Version 1.0

- Added discussion of ILP vs TLP (see Device Utilization and Occupancy).
- Expanded discussion of cache behaviors (see Memory Throughput).
- Added section regarding Warp-synchronous Programming.
- Added section regarding 2D Memory Copies.
- Minor corrections and clarifications.

Version 1.1

- Clarified **`const __restrict__`** discussion and mentioned **`__ldg()`** intrinsic in Read-Only Data Cache.