TUNING CUDA APPLICATIONS FOR VOLTA

DA-08647-001_v10.0 | August 2018

Application Note

TABLE OF CONTENTS

Chapter 1. VOLTA TUNING GUIDE

1.1. NVIDIA Volta Compute Architecture

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

Volta architecture comprises a single variant: GV100. A detailed overview of the major improvements in GV100 over earlier NVIDIA architectures is provided in a white paper entitled NVIDIA Tesla V100 GPU Architecture: The World's Most Advanced Datacenter GPU.

For further 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,

[1] Throughout this guide, *Kepler* refers to devices of compute capability 3.x, *Maxwell* refers to devices of compute capability 5.x, *Pascal* refers to device of compute capability 6.x, and *Volta* refers to devices of compute capability 7.x.

- 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 Volta Compatibility Guide for CUDA Applications to ensure that your application is compiled in a way that is compatible with Volta.

1.4. Volta Tuning

1.4.1. Streaming Multiprocessor

The Volta Streaming Multiprocessor (SM) provides the following improvements over Pascal.

1.4.1.1. Instruction Scheduling

Each Volta SM includes 4 warp-scheduler units. Each scheduler handles a static set of warps and issues to a dedicated set of arithmetic instruction units. Instructions are performed over two cycles, and the schedulers can issue independent instructions every cycle. Dependent instruction issue latency for core FMA math operations are reduced to four clock cycles, compared to six cycles on Pascal. As a result, execution latencies of core math operations can be hidden by as few as 4 warps per SM, assuming 4-way instruction-level parallelism *ILP* per warp. Many more warps are, of course, recommended to cover the much greater latency of memory transactions and control-flow operations.

Similar to GP100, the GV100 SM provides 64 FP32 cores and 32 FP64 cores. The GV100 SM additionally includes 64 INT32 cores and 8 mixed-precision Tensor Cores. GV100 provides up to 84 SMs.

1.4.1.2. Independent Thread Scheduling

The Volta architecture introduces *Independent Thread Scheduling* among threads in a warp. This feature enables intra-warp synchronization patterns previously unavailable and simplifies code changes when porting CPU code. However, Independent Thread Scheduling can also lead to a rather different set of threads participating in the executed code than intended if the developer made assumptions about warp-synchronicity[2] of previous hardware architectures.

When porting existing codes to Volta, the following three code patterns need careful attention. For more details see the *CUDA C Programming Guide*.

- To avoid data corruption, applications using warp intrinsics (**`__shfl*`**, **`__any`**, **`__all`**, and **`__ballot`**) should transition to the new, safe, synchronizing

[2] The term warp-synchronous refers to code that implicitly assumes threads in the same warp are synchronized at every instruction.

counterparts, with the **`*_sync`** suffix. The new warp intrinsics take in a mask of threads that explicitly define which lanes (threads of a warp) must participate in the warp intrinsic.

- Applications that assume reads and writes are implicitly visible to other threads in the same warp need to insert the new **`__syncwarp()`** warp-wide barrier synchronization instruction between steps where data is exchanged between threads via global or shared memory. Assumptions that code is executed in lockstep or that reads/writes from separate threads are visible across a warp without synchronization are invalid.
- Applications using **`__syncthreads()`** or the PTX **`bar.sync`** (and their derivatives) in such a way that a barrier will not be reached by some non-exited thread in the thread block must be modified to ensure that all non-exited threads reach the barrier.

The **`racecheck`** and **`synccheck`** tools provided by **`cuda-memcheck`** can aid in locating violations of points 2 and 3.

1.4.1.3. Occupancy

The maximum number of concurrent warps per SM remains the same as in Pascal (i.e., 64), and other factors influencing warp occupancy remain similar as well:

- The register file size is 64k 32-bit registers per SM.
- The maximum registers per thread is 255.
- The maximum number of thread blocks per SM is 32.
- Shared memory capacity per SM is 96KB, similar to GP104, and a 50% increase compared to GP100.

Overall, developers can expect similar occupancy as on Pascal without changes to their application.

1.4.1.4. Integer Arithmetic

Unlike Pascal GPUs, the GV100 SM includes dedicated FP32 and INT32 cores. This enables simultaneous execution of FP32 and INT32 operations. Applications can now interleave pointer arithmetic with floating-point computations. For example, each iteration of a pipelined loop could update addresses and load data for the next iteration while simultaneously processing the current iteration at full FP32 throughput.

1.4.2. Tensor Core Operations

Each Tensor Core performs the following operation: D = AxB + C, where A, B, C, and D are 4x4 matrices. The matrix multiply inputs A and B are FP16 matrices, while the accumulation matrices C and D may be FP16 or FP32 matrices.

When accumulating in FP32, the FP16 multiply results in a full precision product that is then accumulated using FP32 addition with the other intermediate products for a 4x4x4 matrix multiply. In practice, Tensor Cores are used to perform much larger 2D or higher dimensional matrix operations, built up from these smaller elements.

The Volta tensor cores are exposed as Warp-Level Matrix Operations in the CUDA 9 C ++ API. The API exposes specialized matrix load, matrix multiply and accumulate, and matrix store operations to efficiently use Tensor Cores from a CUDA-C++ program. At the CUDA level, the warp-level interface assumes 16x16 size matrices spanning all 32 threads of the warp. See the *CUDA C Programming Guide* for more information.

1.4.3. Memory Throughput

1.4.3.1. High Bandwidth Memory

GV100 uses up to eight memory dies per HBM2 stack and four stacks, with a maximum of 32 GB of GPU memory. A faster and more efficient HBM2 implementation delivers up to 900 GB/s of peak memory bandwidth, compared to 732 GB/s for GP100. This combination of a new generation HBM2 memory, and a new generation memory controller, in Volta provides 1.5x delivered memory bandwidth, compared to Pascal GP100—and a greater than 95% memory bandwidth efficiency running many workloads.

In order to hide the DRAM latencies at full HBM2 bandwidth more memory accesses must be kept in flight, compared to GPUs equipped with traditional GDDR5. This is accomplished by the large complement of SMs in GV100, which typically boost the number of concurrent threads, and thus the reads-in-flight, compared to previous architectures. Resource-constrained kernels that are limited to low occupancy may benefit from increasing the number of concurrent memory accesses per thread.

1.4.3.2. Unified Shared Memory/L1/Texture Cache

In Volta the L1 cache, texture cache, and shared memory are backed by a combined 128 KB data cache. As in previous architectures, such as Kepler, the portion of the cache dedicated to shared memory (known as the *carveout*) can be selected at runtime using **`cudaFuncSetAttribute()`** with the attribute **`cudaFuncAttributePreferredSharedMemoryCarveout`**. Volta supports shared memory capacities of 0, 8, 16, 32, 64, or 96 KB per SM.

A new feature, Volta enables a single thread block to address the full 96 KB of shared memory. To maintain architectural compatibility, static shared memory allocations remain limited to 48 KB, and an explicit opt-in is also required to enable dynamic allocations above this limit. See the *CUDA C Programming Guide* for details.

Like Pascal, Volta combines the functionality of the L1 and texture caches into a unified L1/Texture cache which 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.

Volta increases the maximum capacity of the L1 cache to 128 KB, more than 7x larger than the GP100 L1. Another benefit of its union with shared memory, the Volta L1 improves in terms of both latency and bandwidth compared to Pascal. The result is that for many applications Volta narrows the performance gap between explicitly managed shared memory and direct access to device memory. Also, the cost of register spills is lowered compared to Pascal, and the balance of occupancy versus spilling should be re-evaluated to ensure best performance.

1.4.4. Cooperative Groups

1.4.5. Multi-Process Service

The Volta Multi-Process Service is significantly improved compared to previous architecutres, both in terms of performance and robustness. Intermediary software schedulers, used for MPS with previous architectures, have been replaced by hardware accelerated units within the GPU. MPS clients now submit tasks directly to the GPU work queues, significantly decreasing submission latency and increasing aggregate throughput. The limit on the number of MPS clients has also been increased by 3x to 48. Volta MPS also provides each client with an isolated address space,[3] and extends Unified Memory support for MPS applications.

Volta MPS also provides control for clients to restrict each client to a fraction of the GPU execution resources. Developers can use this feature to reduce or eliminate head-of-line blocking where work from one MPS client overwhelms GPU execution resources and prevents other clients from making progress, and thus improve average latency and jitter accross the system.

1.4.6. NVLink Interconnect

NVLink is NVIDIA's high-speed data interconnect. NVLink can be used to significantly increase performance for both GPU-to-GPU communication and for GPU access to system memory. GV100 supports up to six NVLink connections with each connection carrying up to 50 GB/s of bi-directional bandwidth.

NVLink operates transparently within the existing CUDA model. Transfers between NVLink-connected endpoints are automatically routed through NVLink, rather than PCIe. The **`cudaDeviceEnablePeerAccess()`** API call remains necessary to enable direct transfers (over either PCIe or NVLink) between GPUs. The **`cudaDeviceCanAccessPeer()`** can be used to determine if peer access is possible between any pair of GPUs.

[3] As with previous architectures, MPS does not provide fatal fault isolation between clients.

Appendix A. REVISION HISTORY

Version 1.0

- Initial Public Release