Tuning CUDA Applications for Turing

Application Note

DA-09073-001_v12.0 | December 2022

Table of Contents

Chapter 1. Turing Tuning Guide

1.1. NVIDIA Turing Compute Architecture

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

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,
- Minimize redundant accesses to global memory whenever possible,
- Avoid long sequences of diverged execution by threads within the same warp.

[1] Throughout this guide, *Maxwell* refers to devices of compute capability 5.x, *Pascal* refers to devices of compute capability 6.x, *Volta* refers to devices of compute capability 7.0, and *Turing* refers to devices of compute capability 7.5.

1.3. Application Compatibility

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

1.4. Turing Tuning

1.4.1. Streaming Multiprocessor

The Turing Streaming Multiprocessor (SM) is based on the same major architecture (7.x) as Volta, and provides similar improvements over Pascal.

1.4.1.1. Instruction Scheduling

Each Turing 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 is four clock cycles, like Volta, 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, or by 16 warps per SM without any instuction-level parallelism.

Like Volta, the Turing SM provides 64 FP32 cores, 64 INT32 cores and 8 improved mixed-precision Tensor Cores. Turing has a lower double precision throughput than Volta with only 2 FP64 cores.

1.4.1.2. Independent Thread Scheduling

The Turing architecture features the same *Independent Thread Scheduling* introduced with Volta. This 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 or Turing, 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 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.

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

- 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.

1.4.1.3. Occupancy

The maximum number of concurrent warps per SM is 32 on Turing (versus 64 on Volta). Other factors influencing warp occupancy remain otherwise similar:

- 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 16.
- Shared memory capacity per SM is 64KB.

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

1.4.1.4. Integer Arithmetic

Similar to Volta, the Turing SM includes dedicated FP32 and INT32 cores. This enables simultaneous execution of FP32 and INT32 operations. Applications can 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

Volta introduced Tensor Cores to accelerate matrix multiply operations on mixed precision floating point data. Turing adds acceleration for integer matrix multiply operations. The tensor cores are exposed as Warp-Level Matrix Operations in the CUDA 10 C++ API. The API provides specialized matrix load, matrix multiply and accumulate, and matrix store operations, where each warp processes a small matrix fragment, allowing to efficiently use Tensor Cores from a CUDA-C++ program. In practice, Tensor Cores are used to perform much larger 2D or higher dimensional matrix operations, built up from these smaller matrix fragments.

Each Tensor Core performs the matrix multiply-accumulate: D = A x B + C. The Tensor Cores support half precision matrix multiplication, where the matrix multiply inputs A and B are FP16 matrices, while the accumulation matrices C and D may be either FP16 or FP32 matrices. When accumulating in FP32, the FP16 multiply results in a full precision product that is then accumulated using FP32 addition. CUDA 10 supports several fragment sizes, 16x16x16, 32x8x16, and 8x32x16 to use the Tensor Cores on Volta or Turing with FP16 inputs.

Any binary compiled for Volta will run on Turing, but Volta binaries using Tensor Cores will only be able to reach half of Turing's Tensor Core peak performance. Recompiling the binary specifically for Turing would allow it to reach the peak performance. See the *Turing Compatibility Guide* for more information.

Turing's Tensor Core supports integer matrix multiply operations, which can operate on 8-bit, 4-bit and 1-bit integer inputs, with 32-bit integer accumulation. When operating on 8-bit inputs, CUDA exposes fragment sizes of 16x16x16, 32x8x16, and 8x32x16. For sub-byte operations the fragment sizes available are 8x8x32 for 4-bit inputs, or 8x8x128 for 1-bit inputs.

See the *CUDA C++ Programming Guide* for more information.

1.4.3. Memory Throughput

1.4.3.1. Unified Shared Memory/L1/Texture Cache

Turing features a unified L1 / Shared Memory cache similar to the one introduced in Volta, but with a smaller size. The total size of the unified L1 / Shared Memory cache in Turing is 96 KB. The portion of the cache dedicated to shared memory or L1 (known as the *carveout*) can be changed at runtime, either automatically by the driver, or manually using the `cudaFuncSetAttribute()` with the attribute `cudaFuncAttributePreferredSharedMemoryCarveout`. Turing supports two carveout configurations, either with 64 KB of shared memory and 32 KB of L1, or with 32 KB of shared memory and 64 KB of L1.

Turing allows a single thread block to address the full 64 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 and Volta, Turing 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.

The state-of-the-art L1 cache in Volta and Turing offers lower latency, higher bandwidth, and higher capacity compared to the earlier architectures. Like Volta, Turing's L1 can cache write operations (write-through). The result is that for many applications Volta and Turing narrow 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.

Appendix A. Revision History

Version 1.0

- Initial Public Release

Version 1.1

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

Notice

This document is provided for information purposes only and shall not be regarded as a warranty of a certain functionality, condition, or quality of a product. NVIDIA Corporation ("NVIDIA") makes no representations or warranties, expressed or implied, as to the accuracy or completeness of the information contained in this document and assumes no responsibility for any errors contained herein. NVIDIA shall have no liability for the consequences or use of such information or for any infringement of patents or other rights of third parties that may result from its use. This document is not a commitment to develop, release, or deliver any Material (defined below), code, or functionality.

NVIDIA reserves the right to make corrections, modifications, enhancements, improvements, and any other changes to this document, at any time without notice.

Customer should obtain the latest relevant information before placing orders and should verify that such information is current and complete.

NVIDIA products are sold subject to the NVIDIA standard terms and conditions of sale supplied at the time of order acknowledgement, unless otherwise agreed in an individual sales agreement signed by authorized representatives of NVIDIA and customer ("Terms of Sale"). NVIDIA hereby expressly objects to applying any customer general terms and conditions with regards to the purchase of the NVIDIA product referenced in this document. No contractual obligations are formed either directly or indirectly by this document.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Copyright

NVIDIA Corporation | 2788 San Tomas Expressway, Santa Clara, CA 95051
www.nvidia.com