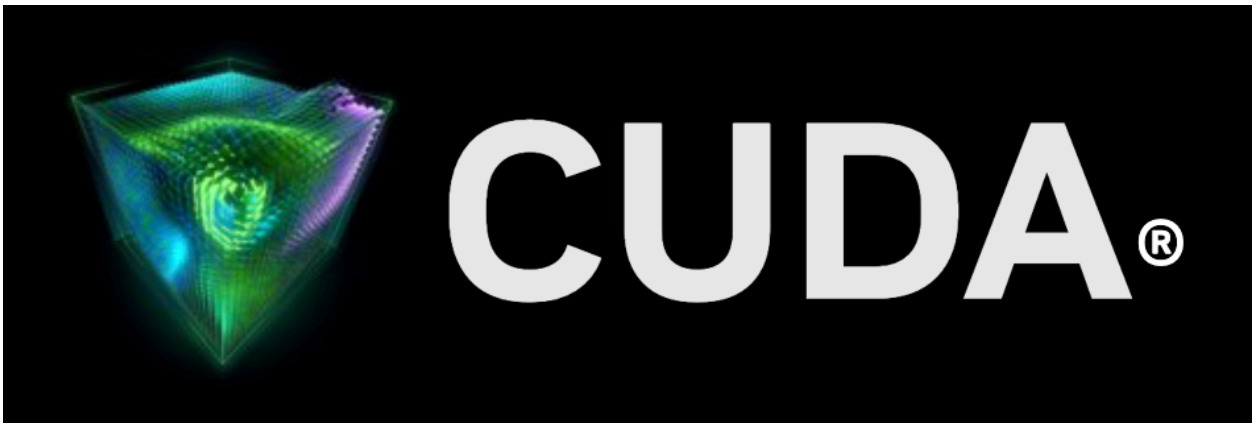

Volta Tuning Guide

Release 13.4

NVIDIA Corporation

Jul 15, 2026

Contents

Tuning CUDA Applications for Volta

The programming guide to tuning CUDA Applications for GPUs based on the NVIDIA Volta Architecture.

Chapter 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] Throughout this guide, *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.

Chapter 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.

Chapter 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.

Chapter 4. Volta Tuning

4.1. Streaming Multiprocessor

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

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.

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 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

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

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 `compute-sanitizer` can help with locating violations.

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.

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.

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.

4.3. Memory Throughput

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.

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, 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.

4.4. Cooperative Groups

The Volta architecture introduced Independent Thread Scheduling, which enables intra-warp synchronization patterns that were previously not possible. To efficiently express these new patterns, CUDA 9 introduces Cooperative Groups. This is an extension to the CUDA programming model for organizing groups of communicating threads. Cooperative Groups allows developers to express the granularity at which threads are communicating, helping them to express richer, more efficient parallel decompositions. See the *CUDA C++ Programming Guide* for more information.

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.

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.

Chapter 5. Revision History

Version 1.0

- Initial Public Release

Version 1.1

- Added Cooperative Groups section.
- Updated references to the *CUDA C++ Programming Guide* and *CUDA C++ Best Practices Guide*.

Chapter 6. Notices

6.1. 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.

NVIDIA products are not designed, authorized, or warranted to be suitable for use in medical, military, aircraft, space, or life support equipment, nor in applications where failure or malfunction of the NVIDIA product can reasonably be expected to result in personal injury, death, or property or environmental damage. NVIDIA accepts no liability for inclusion and/or use of NVIDIA products in such equipment or applications and therefore such inclusion and/or use is at customer’s own risk.

NVIDIA makes no representation or warranty that products based on this document will be suitable for any specified use. Testing of all parameters of each product is not necessarily performed by NVIDIA. It is customer’s sole responsibility to evaluate and determine the applicability of any information contained in this document, ensure the product is suitable and fit for the application planned by customer, and perform the necessary testing for the application in order to avoid a default of the application or the product. Weaknesses in customer’s product designs may affect the quality and reliability of the NVIDIA product and may result in additional or different conditions and/or requirements beyond those contained in this document. NVIDIA accepts no liability related to any default, damage, costs, or problem which may be based on or attributable to: (i) the use of the NVIDIA product in any manner that is contrary to this document or (ii) customer product designs.

No license, either expressed or implied, is granted under any NVIDIA patent right, copyright, or other NVIDIA intellectual property right under this document. Information published by NVIDIA regarding third-party products or services does not constitute a license from NVIDIA to use such products or

services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property rights of the third party, or a license from NVIDIA under the patents or other intellectual property rights of NVIDIA.

Reproduction of information in this document is permissible only if approved in advance by NVIDIA in writing, reproduced without alteration and in full compliance with all applicable export laws and regulations, and accompanied by all associated conditions, limitations, and notices.

THIS DOCUMENT AND ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL NVIDIA BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF NVIDIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Notwithstanding any damages that customer might incur for any reason whatsoever, NVIDIA's aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms of Sale for the product.

6.2. OpenCL

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

6.3. 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