Tuning CUDA Applications for Hopper

Application Note

DA-11076-001_v12.1 | February 2023

Table of Contents

Chapter 1. NVIDIA Hopper Tuning Guide....1
1.1. NVIDIA Hopper GPU Architecture....1
1.2. CUDA Best Practices....1
1.3. Application Compatibility....2
1.4. NVIDIA Hopper Tuning....2
1.4.1. Streaming Multiprocessor....2
1.4.1.1. Occupancy....2
1.4.1.2. Tensor Memory Accelerator....2
1.4.1.3. Thread Block Clusters....3
1.4.1.4. Improved FP32 Throughput....3
1.4.1.5. Dynamic Programming Instructions....3
1.4.2. Memory System....4
1.4.2.1. High-Bandwidth Memory HBM3 Subsystem....4
1.4.2.2. Increased L2 Capacity....4
1.4.2.3. Inline Compression....4
1.4.2.4. Unified Shared Memory/L1/Texture Cache....5
1.4.3. Fourth-Generation NVLink....5
Appendix A. Revision History....6

Chapter 1. NVIDIA Hopper Tuning Guide

1.1. NVIDIA Hopper GPU Architecture

The NVIDIA® Hopper GPU architecture is NVIDIA's latest architecture for CUDA® compute applications. The NVIDIA Hopper GPU architecture retains and extends the same CUDA programming model provided by previous NVIDIA GPU architectures such as NVIDIA Ampere GPU architecture and NVIDIA Turing, and applications that follow the best practices for those architectures should typically see speedups on the NVIDIA H100 GPU without any code changes. This guide summarizes the ways that an application can be fine-tuned to gain additional speedups by leveraging the NVIDIA Hopper GPU architecture's features.[1]

For further details on the programming features discussed in this guide, 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 that global memory accesses are coalesced.
- Minimize redundant accesses to global memory whenever possible.

[1] Throughout this guide, *NVIDIA Volta* refers to devices of compute capability 7.0, *NVIDIA Turing* refers to devices of compute capability 7.5, *NVIDIA Ampere GPU Architecture* refers to devices of compute capability 8.x, and *NVIDIA Hopper* refers to devices of compute capability 9.0.

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

1.4. NVIDIA Hopper Tuning

1.4.1. Streaming Multiprocessor

The NVIDIA Hopper Streaming Multiprocessor (SM) provides the following improvements over Turing and NVIDIA Ampere GPU architectures.

1.4.1.1. Occupancy

The maximum number of concurrent warps per SM remains the same as in NVIDIA Ampere GPU architecture (that is, 64), and other factors influencing warp occupancy are:

- The register file size is 64K 32-bit registers per SM.
- The maximum number of registers per thread is 255.
- The maximum number of thread blocks per SM is 32 for devices of compute capability 9.0 (that is, H100 GPUs).
- For devices of compute capability 9.0 (H100 GPUs), shared memory capacity per SM is 228 KB, a 39% increase compared to A100's capacity of 164 KB.
- For devices of compute capability 9.0 (H100 GPUs), the maximum shared memory per thread block is 227 KB.
- For applications using Thread Block Clusters, it is always recommended to compute the occupancy using `cudaOccupancyMaxActiveClusters` and launch cluster-based kernels accordingly.

Overall, developers can expect similar occupancy as on NVIDIA Ampere GPU architecture GPUs without changes to their application.

1.4.1.2. Tensor Memory Accelerator

The Hopper architecture builds on top of the asynchronous copies introduced by NVIDIA Ampere GPU architecture and provides a more sophisticated asynchronous copy engine: the Tensor Memory Accelerator (TMA).

TMA allows applications to transfer 1D and up to 5D tensors between global memory and shared memory, in both directions, as well as between the shared memory regions of different SMs in the same cluster (refer to Thread Block Clusters). Additionally, for writes from shared memory to global memory, it allows specifying element wise reduction operations such as add/min/max as well as bitwise and/or for most common data types.

This has several advantages:

- Avoids using registers for moving data between the different memory spaces.
- Avoids using SM instructions for moving data: a single thread can issue large data movement instructions to the TMA unit. The whole block can then continue working on other instructions while the data is in flight and only wait for the data to be consumed when actually necessary.
- Enables users to write warp specialized codes, where specific warps specialize on data movement between the different memory spaces while other warps only work on local data within the SM.

This feature will be exposed through `cuda::memcpy_async` along with the `cuda::barrier` and `cuda::pipeline` for synchronizing data movement.

1.4.1.3. Thread Block Clusters

NVIDIA Hopper Architecture adds a new optional level of hierarchy, Thread Block Clusters, that allows for further possibilities when parallelizing applications. A thread block can read from, write to, and perform atomics in shared memory of other thread blocks within its cluster. This is known as Distributed Shared Memory. As demonstrated in the CUDA C++ Programming Guide, there are applications that cannot fit required data within shared memory and must use global memory instead. Distributed shared memory can act as an intermediate step between these two options.

Distributed Shared Memory can be used by an SM simultaneously with L2 cache accesses. This can benefit applications that need to communicate data between SMs by utilizing the combined bandwidth of both distributed shared memory and L2.

The maximum portable cluster size supported is 8; however, NVIDIA Hopper H100 GPU allows for a nonportable cluster size of 16 by opting in. Launching a kernel with a nonportable cluster size requires setting the **cudaFuncAttributeNonPortableClusterSizeAllowed** function attribute. Using larger cluster sizes may reduce the maximum number of active blocks across the GPU (refer to Occupancy).

1.4.1.4. Improved FP32 Throughput

Devices of compute capability 9.0 have 2x more FP32 operations per cycle per SM than devices of compute capability 8.0.

1.4.1.5. Dynamic Programming Instructions

The NVIDIA Hopper architecture adds support for new instructions to accelerate dynamic programming algorithms, such as the Smith-Waterman algorithm for sequence alignment in bioinformatics, and algorithms in graph theory, game theory, ML, and finance problems. The new instructions permit computation of max and min values among three operands, max and min operations yielding predicates, combined add operation with max or min, operating on signed and unsigned 32-bit int and 16-bit short2 types, and half2. All DPX instructions with 16-bit short types DPX instructions enable 128 operations per cycle per SM.

1.4.2. Memory System

1.4.2.1. High-Bandwidth Memory HBM3 Subsystem

The NVIDIA H100 GPU has support for HBM3 and HBM2e memory, with capacity up to 80 GB. GPUs HBM3 memory system supports up to 3 TB/s memory bandwidth, a 93% increase over the 1.55 TB/s on A100-40GB.

1.4.2.2. Increased L2 Capacity

The NVIDIA Hopper architecture increases the L2 cache capacity from 40 MB in the A100 GPU to 50 MB in the H100 GPU. Along with the increased capacity, the bandwidth of the L2 cache to the SMs is also increased. The NVIDIA Hopper architecture allows CUDA users to control the persistence of data in L2 cache similar to the NVIDIA Ampere GPU Architecture. For more information on the persistence of data in L2 cache, refer to the section on managing L2 cache in the CUDA C++ Programming Guide.

1.4.2.3. Inline Compression

The NVIDIA Hopper architecture allows CUDA compute kernels to benefit from the new inline compression (ILC). This feature can be applied to individual memory allocation, and the compressor automatically chooses between several possible compression algorithms, or none if there is no suitable pattern.

In case compression can be used, this feature allows accessing global memory at significantly higher bandwidth than global memory bandwidth, since only compressed data needs to be transferred between global memory and SMs.

However, the feature does not allow for reducing memory footprint: since compression is automatic, even if compression is active, the memory region will use the same footprint as if there was no compression. This is because underlying data may be changed by the user application and may not be compressible during the entire duration of the application.

The feature is available through the CUDA driver API. See the CUDA C++ Programming Guide section on compressible memory:

```
CUmemGenericAllocationHandle allocationHandle;
CUmemAllocationProp prop = {};
memset(prop, 0, sizeof(CUmemAllocationProp));
prop->type = CU_MEM_ALLOCATION_TYPE_PINNED;
prop->location.type = CU_MEM_LOCATION_TYPE_DEVICE;
prop->location.id = currentDevice;
prop->allocFlags.compressionType = CU_MEM_ALLOCATION_COMP_GENERIC;
cuMemCreate(&allocationHandle, size, &prop, 0);
```

One can check whether compressible memory is available on the given device with:

```
cuDeviceGetAttribute(&compressionAvailable,
        CU_DEVICE_ATTRIBUTE_GENERIC_COMPRESSION_SUPPORTED, currentDevice)
```

Note that this example code does not handle errors and compiling this code requires linking against the CUDA library (`libcuda.so`).

1.4.2.4. Unified Shared Memory/L1/Texture Cache

The NVIDIA H100 GPU based on compute capability 9.0 increases the maximum capacity of the combined L1 cache, texture cache, and shared memory to 256 KB, from 192 KB in NVIDIA Ampere Architecture, an increase of 33%.

In the NVIDIA Hopper GPU architecture, the portion of the L1 cache dedicated to shared memory (known as the carveout) can be selected at runtime as in previous architectures such as NVIDIA Ampere Architecture and NVIDIA Volta, using `cudaFuncSetAttribute()` with the attribute `cudaFuncAttributePreferredSharedMemoryCarveout`. The NVIDIA H100 GPU supports shared memory capacities of 0, 8, 16, 32, 64, 100, 132, 164, 196 and 228 KB per SM.

CUDA reserves 1 KB of shared memory per thread block. Hence, the H100 GPU enables a single thread block to address up to 227 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 the NVIDIA Ampere Architecture and NVIDIA Volta GPU architectures, the NVIDIA Hopper GPU architecture 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 before delivery of that data to the warp. Another benefit of its union with shared memory, similar to previous architectures, is improvement in terms of both latency and bandwidth.

1.4.3. Fourth-Generation NVLink

The fourth generation of NVIDIA's high-speed NVLink interconnect is implemented in H100 GPUs, which significantly enhances multi-GPU scalability, performance, and reliability with more links per GPU, much faster communication bandwidth, and improved error-detection and recovery features. The fourth-generation NVLink has the same bidirectional data rate of 50 GB/s per link. The total number of links available is increased to 18 in H100, compared to 12 in A100, yielding 900 GB/s bidirectional bandwidth compared to 600 GB/s for A100.

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.

Appendix A. Revision History

Version 1.0

- Initial Public Release
- Added support for compute capability 9.0