GPU Performance Background

User's Guide | NVIDIA Docs

DU-09798-001_v001 | February 2023

Table of Contents

List of Figures

List of Tables

Chapter 1. Overview

It is helpful to understand the basics of GPU execution when reasoning about how efficiently particular layers or neural networks are utilizing a given GPU.

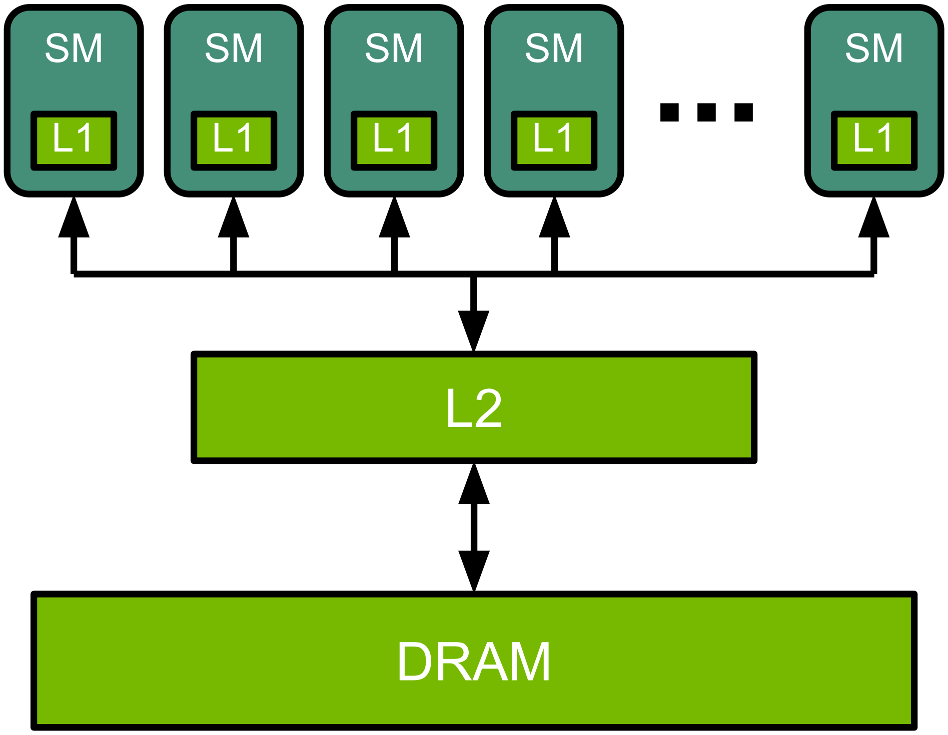
This guide describes:

- The basic structure of a GPU (GPU Architecture Fundamentals)
- How operations are divided and executed in parallel (GPU Execution Model)
- How to estimate performance limitations with arithmetic intensity (Understanding Performance)
- Loose categories of deep learning operations and the performance limitations that tend to apply to each (DNN Operation Categories)

Chapter 2. GPU Architecture Fundamentals

The GPU is a highly parallel processor architecture, composed of processing elements and a memory hierarchy. At a high level, NVIDIA® GPUs consist of a number of Streaming Multiprocessors (SMs), on-chip L2 cache, and high-bandwidth DRAM. Arithmetic and other instructions are executed by the SMs; data and code are accessed from DRAM via the L2 cache. As an example, an NVIDIA A100 GPU contains 108 SMs, a 40 MB L2 cache, and up to 2039 GB/s bandwidth from 80 GB of HBM2 memory.

Figure 1. Simplified view of the GPU architecture



Each SM has its own instruction schedulers and various instruction execution pipelines. Multiply-add is the most frequent operation in modern neural networks, acting as a building block for fully-connected and convolutional layers, both of which can be viewed as a collection of vector dot-products. The following table shows a single SM's multiply-add operations per clock for various data types on NVIDIA's recent GPU architectures. Each multiply-add comprises two operations, thus one would multiply the throughput in the table by 2 to get FLOP counts per clock. To get the FLOPS rate for GPU one would then multiply these by the number of SMs and SM clock rate. For example, an A100 GPU with 108 SMs and 1.41 GHz clock rate has peak dense throughputs of 156 TF32 TFLOPS and 312 FP16 TFLOPS (throughputs achieved by applications depend on a number of factors discussed throughout this document).

Figure 2. Multiply-add operations per clock per SM

	CUDA Cores				Tensor Cores					
NVIDIA Architecture	FP64	FP32	FP16	INT8	FP64	TF32	FP16	INT8	INT4	INT1
Volta	32	64	128	256			512			
Turing	2	64	128	256			512	1024	2048	8192
Ampere (A100)	32	64	256	256	64	512	1024	2048	4096	16384
Ampere, sparse						1024	2048	4096	8192	

As shown in Figure 2, FP16 operations can be executed in either Tensor Cores or NVIDIA CUDA® cores. Furthermore, the NVIDIA Turing™ architecture can execute INT8 operations in either Tensor Cores or CUDA cores. Tensor Cores were introduced in the NVIDIA Volta™ GPU architecture to accelerate matrix multiply and accumulate operations for machine learning and scientific applications. These instructions operate on small matrix blocks (for example, 4x4 blocks). Note that Tensor Cores can compute and accumulate products in higher precision than the inputs. For example, during training with FP16 inputs, Tensor Cores can compute products without loss of precision and accumulate in FP32. When math operations cannot be formulated in terms of matrix blocks they are executed in other CUDA cores. For example, the element-wise addition of two half-precision tensors would be performed by CUDA cores, rather than Tensor Cores.

Chapter 3. GPU Execution Model

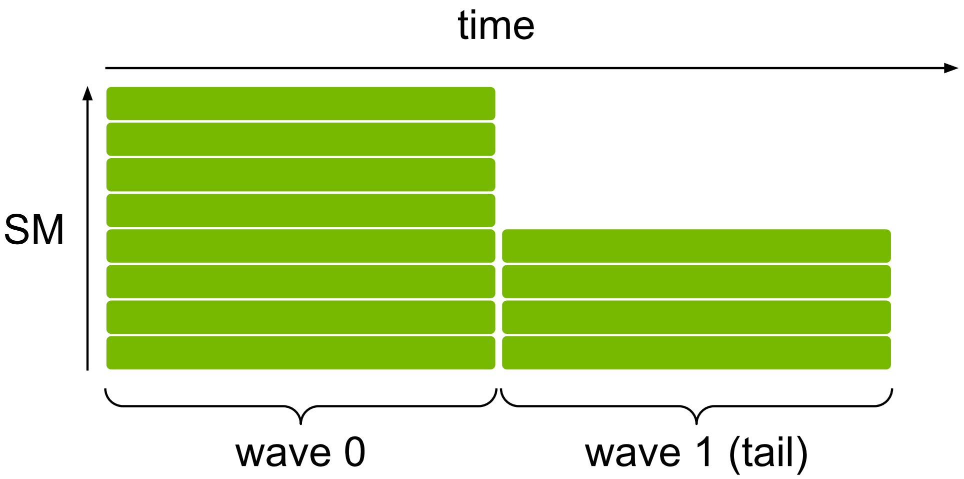
To utilize their parallel resources, GPUs execute many threads concurrently.

There are two concepts critical to understanding how thread count relates to GPU performance:

1. GPUs execute functions using a 2-level hierarchy of threads. A given function's threads are grouped into equally-sized *thread blocks*, and a set of thread blocks are launched to execute the function.
2. GPUs hide dependent instruction latency by switching to the execution of other threads. Thus, the number of threads needed to effectively utilize a GPU is much higher than the number of cores or instruction pipelines.

The 2-level thread hierarchy is a result of GPUs having many SMs, each of which in turn has pipelines for executing many threads and enables its threads to communicate via shared memory and synchronization. At runtime, a thread block is placed on an SM for execution, enabling all threads in a thread block to communicate and synchronize efficiently. Launching a function with a single thread block would only give work to a single SM, therefore to fully utilize a GPU with multiple SMs one needs to launch many thread blocks. Since an SM can execute multiple thread blocks concurrently, typically one wants the number of thread blocks to be several times higher than the number of SMs. The reason for this is to minimize the "tail" effect, where at the end of a function execution only a few active thread blocks remain, thus underutilizing the GPU for that period of time as illustrated in Figure 3.

Figure 3. Utilization of an 8-SM GPU when 12 thread blocks with an occupancy of 1 block/SM at a time are launched for execution. Here, the blocks execute in 2 waves, the first wave utilizes 100% of the GPU, while the 2nd wave utilizes only 50%.



We use the term *wave* to refer to a set of thread blocks that run concurrently. It is most efficient to launch functions that execute in several waves of thread blocks - a smaller percentage of time is spent in the tail wave, minimizing the tail effect and thus the need to do anything about it. For the higher-end GPUs, typically only launches with fewer than 300 thread blocks should be examined for tail effects.

Chapter 4. Understanding Performance

Performance of a function on a given processor is limited by one of the following three factors; memory bandwidth, math bandwidth and latency.

Consider a simplified model where a function reads its input from memory, performs math operations, then writes its output to memory. Say T_{mem} time is spent in accessing memory, T_{math} time is spent performing math operations. If we further assume that memory and math portions of different threads can be overlapped, the total time for the function is $\max\left(T_{\mathrm{mem}},\ T_{\mathrm{math}}\right)$. The longer of the two times demonstrates what limits performance: If math time is longer we say that a function is *math limited*, if memory time is longer then it is *memory limited*.

How much time is spent in memory or math operations depends on both the algorithm and its implementation, as well as the processor's bandwidths. Memory time is equal to the number of bytes accessed in memory divided by the processor's memory bandwidth. Math time is equal to the number of operations divided by the processor's math bandwidth. Thus, on a given processor a given algorithm is math limited if $T_{\mathrm{math}} > T_{\mathrm{mem}}$ which can be expressed as $\#\,\mathrm{ops} \,/\, \mathrm{BW}_{\mathrm{math}} > \#\,\mathrm{bytes} \,/\, \mathrm{BW}_{\mathrm{mem}}$

By simple algebra, the inequality can be rearranged to $\#\,\mathrm{ops} \,/\, \#\,\mathrm{bytes} > \mathrm{BW}_{\mathrm{math}} \,/\, \mathrm{BW}_{\mathrm{mem}}$

The left-hand side, the ratio of algorithm implementation operations and the number of bytes accessed, is known as the algorithm's *arithmetic intensity*. The right-hand side, the ratio of a processor's math and memory bandwidths, is sometimes called *ops:byte* ratio. Thus, an algorithm is math limited on a given processor if the algorithm's arithmetic intensity is higher than the processor's ops:byte ratio. Conversely, an algorithm is memory limited if its arithmetic intensity is lower than the processor's `ops:byte` ratio.

Let's consider some concrete examples from deep neural networks, listed in Table 1 below. For these examples, we will compare the algorithm's arithmetic intensity to the ops:byte ratio on an NVIDIA Volta V100 GPU. V100 has a peak math rate of 125 FP16 Tensor TFLOPS, an off-chip memory bandwidth of approx. 900 GB/s, and an on-chip L2 bandwidth of 3.1 TB/s, giving it a ops:byte ratio between 40 and 139, depending on the source of an operation's data (on-chip or off-chip memory).

Table 1. Examples of neural network operations with their arithmetic intensities. Limiters assume FP16 data and an NVIDIA V100 GPU.

Operation	Arithmetic Intensity	Usually limited by...
Linear layer (4096 outputs, 1024 inputs, batch size 512)	315 FLOPS/B	arithmetic
Linear layer (4096 outputs, 1024 inputs, batch size 1)	1 FLOPS/B	memory
Max pooling with 3x3 window and unit stride	2.25 FLOPS/B	memory
ReLU activation	0.25 FLOPS/B	memory
Layer normalization	< 10 FLOPS/B	memory

As the table shows, many common operations have low arithmetic intensities - sometimes only performing a single operation per two-byte element read from and written to memory. Note that this type of analysis is a simplification, as we're counting only the *algorithmic* operations used. In practice, functions also contain instructions for operations not explicitly expressed in the algorithm, such as memory access instructions, address calculation instructions, control flow instructions, and so on.

The arithmetic intensity and ops:byte ratio analysis assumes that a workload is sufficiently large to saturate a given processor's math and memory pipelines. However, if the workload is not large enough, or does not have sufficient parallelism, the processor will be under-utilized and performance will be limited by latency. For example, consider the launch of a single thread that will access 16 bytes and perform 16000 math operations. While the arithmetic intensity is 1000 FLOPS/B and the execution should be math-limited on a V100 GPU, creating only a single thread grossly under-utilizes the GPU, leaving nearly all of its math pipelines and execution resources idle. Furthermore, the arithmetic intensity calculation assumes that inputs and outputs are accessed from memory exactly once. It is not unusual for algorithm implementations to read input elements multiple times, which would effectively reduce arithmetic intensity. Thus, the arithmetic intensity is a first-order approximation; profiler information should be used if more accurate analysis is needed.

Chapter 5. DNN Operation Categories

While modern neural networks are built from a variety of layers, their operations fall into three main categories according to the nature of computation.

5.1. Elementwise Operations

Elementwise operations may be unary or binary operations; the key is that layers in this category perform mathematical operations on each element independently of all other elements in the tensor.

For example, a ReLU layer returns `max(0, x)` for each x in the input tensor. Similarly, element-wise addition of two tensors computes each output sum value independently of other sums. Layers in this category include most non-linearities (sigmoid, tanh, etc.), scale, bias, add, and others. These layers tend to be memory-limited, as they perform few operations per byte accessed. Further details on activations, in particular, can be found within the Activations section in the *Optimizing Memory-Bound Layers User's Guide*.

5.2. Reduction Operations

Reduction operations produce values computed over a range of input tensor values.

For example, pooling layers compute values over some neighborhoods in the input tensor. Batch normalization computes the mean and standard deviation over a tensor before using them in operations for each output element. In addition to pooling and normalization layers, SoftMax also falls into the reduction category. Typical reduction operations have a low arithmetic intensity and thus are memory limited. Further details on pooling layers can be found within Pooling.

5.3. Dot-Product Operations

Operations in this category can be expressed as dot-products of elements from two tensors, usually a weight (learned parameter) tensor and an activation tensor.

These include fully-connected layers, occurring on their own and as building blocks of recurrent and attention cells. Fully-connected layers are naturally expressed as matrix-vector and matrix-matrix multiplies. Convolutions can also be expressed as collections of dot-products - one vector is the set of parameters for a given filter, the other is an “unrolled” activation region to which that filter is being applied. Since filters are applied in multiple locations, convolutions too can be viewed as matrix-vector or matrix-matrix multiply operations (refer to Convolution Algorithms).

Operations in the dot-product category can be math-limited if the corresponding matrices are large enough. However, for the smaller sizes, these operations end up being memory-limited. For example, a fully-connected layer applied to a single vector (a tensor for a mini-batch of size 1)) is memory limited.

Matrix-matrix multiplication performance is discussed in more detail in the NVIDIA Matrix Multiplication Background User's Guide. Information on modeling a type of layer as a matrix multiplication can be found in the corresponding guides:

- NVIDIA Optimizing Linear/Fully-Connected Layers User's Guide
- NVIDIA Optimizing Convolutional Layers User's Guide
- NVIDIA Optimizing Recurrent Layers User's Guide

Chapter 6. Summary

To roughly approximate what limits the performance of a particular function on a given GPU, one can take the following steps:

- Look up the number of SMs on the GPU, and determine the ops:bytes ratio for the GPU.
- Compute the arithmetic intensity for the algorithm.
- Determine if there is sufficient parallelism to saturate the GPU by estimating the number and size of thread blocks. If the number of thread blocks is at least roughly 4x higher than the number of SMs, and thread blocks consist of a few hundred threads each, then there is likely sufficient parallelism.
  - The guide for a particular layer type provides more intuition on parallelization (refer to NVIDIA Optimizing Linear/Fully-Connected Layers User's Guide, NVIDIA Optimizing Convolutional Layers User's Guide, and NVIDIA Optimizing Recurrent Layers User's Guide; NVIDIA Optimizing Memory-Bound Layers User's Guide may also be helpful, though such layers are naturally expected to be memory-limited).
- The most likely performance limiter is:
  - Latency if there is not sufficient parallelism
  - Math if there is sufficient parallelism and algorithm arithmetic intensity is higher than the GPU ops:byte ratio.
  - Memory if there is sufficient parallelism and algorithm arithmetic intensity is lower than the GPU ops:byte ratio.

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.

Google

Android, Android TV, Google Play and the Google Play logo are trademarks of Google, Inc.

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

Trademarks

Copyright

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