Reading Summary Week 3

Aakash Agrawal
HDSI
University of California San Diego
San Diego, CA, 92092
aaa015@ucsd.edu

1 GPU Performance Background

Link to the Guide

This reading mainly talks about the different facets of GPU architecture. It delves into the GPU's execution model, introduces key performance metrics, and concludes by discussing different categories of deep learning operations along with their specific performance constraints.

1.1 GPU Architecture Fundamentals

A GPU (Graphics Processing Unit) is a specialized processor with a highly parallel architecture featuring numerous ALUs. It leverages the SIMD (Single Instruction, Multiple Data) approach, where a single instruction is executed across multiple data points at the same time, enabling the GPU to handle many simple operations in parallel, which is ideal for tasks like graphics rendering, video processing, and data-intensive computations in fields like AI and scientific research.

NVIDIA GPUs are composed of Streaming Multiprocessors (SMs) for arithmetic operations, L1 caches per core for fast data retrieval, a shared L2 cache, and High Bandwidth Memory (HBM) for efficient data access. With **multiply-add** operations being prevalent in training deep learning models, this reading compares the per-clock, per-SM performance across various NVIDIA GPU architectures for different precisions. The **per clock per SM** metric indicates the computational capacity of a Streaming Multiprocessor (SM) in one cycle. These operations can be executed either in CUDA cores or Tensor cores.

- **CUDA Cores** - General purpose processors that handle wide variety of parallel computing tasks. They are suitable for all purpose parallel computing. They are versatile, supporting various precision levels (FP32, FP64).
- **Tensor Cores** - Specialized for tensor operations like matrix multiplication and accumulation, these units are optimized for deep learning tasks. They execute matrix multiplications much faster than CUDA cores and support mixed-precision computing (FP16, INT8), allowing for higher precision accumulation (FP32) from lower precision inputs (FP16). Operations not fitting into matrix block formats are executed on CUDA cores.

1.2 GPU execution model

GPUs leverage their parallel processing capabilities by running numerous threads simultaneously. This leads to a **two-level hierarchy**:

- **Thread**: The smallest unit of execution within a process.
- **Blocks**: Groups of threads that share memory, with each block allocated to one SM.
- **Grid**: A collection of blocks that execute the same kernel.

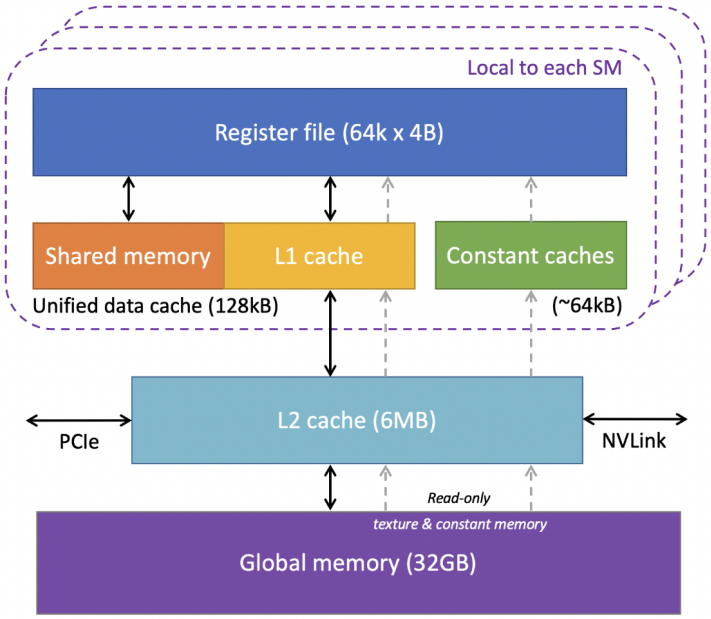

Figure 1: A simplified view of the GPU architecture showing GPU memory levels for the NVIDIA Tesla V100. Source: `https://cvw.cac.cornell.edu/gpu-architecture/gpu-memory/memory_levels`

Q. Why a large number of threads is desired?

GPUs hide **dependent instruction latency** by executing other independent threads while waiting for the dependent instruction to complete. By switching to other threads that are not dependent on the result of the current instruction, GPUs maintain high throughput and keep the execution units (like CUDA cores) busy. If there are only a few threads, the GPU may end up waiting on dependent instructions for too long, leading to underutilization of the execution units.

Q. Why the number of thread blocks should be higher than the number of SMs?

Launching a function (kernel) with a single thread block will only give work to a single SM. To fully utilize a GPU with multiple SMs, one needs to launch many thread blocks. The reason for this is to minimize the **tail effect**. It occurs when only a few thread blocks remain active after most of the work has been completed, causing the remaining available Streaming Multiprocessors (SMs) to be underutilized.

1.3 Understanding GPU Performance

The performance of a kernel on a given GPU is mainly limited by three factors: memory bandwidth, math bandwidth, and latency. If the time spent accessing memory is larger than the time spent performing math operations, we say that the kernel is memory-limited; otherwise, we say the function is math-limited. Performance can also be understood using **arithmetic intensity** and **ops:byte ratio**. 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 ops:byte ratio.

This type of analysis assumes that the GPU workload is sufficiently large to saturate a processor's math and memory pipelines; however, if the workload does not have sufficient parallelism, the processor will be underutilized, and the performance will be **limited by latency**.

1.4 DNN Operation Categories

Most of the operations of deep learning models fall into the following three main categories according to the nature of their computation:

- **Elementwise Operations**: Here, the operation on each element is performed independently of all other elements in a tensor. Examples: non-linearity like ReLU and Softmax, element-wise addition, and subtraction. The layers performing this operation tend to be memory-limited, as they perform few operations per byte accessed.
- **Reduction Operations**: These operations produce values computed over a range of input tensor values. Examples: pooling layers, batch normalization, softmax, etc. Reduction operations have low arithmetic intensity and, hence, are memory-limited.
- **Dot-Product Operations**: Here, operations are expressed as dot-products of elements from two tensors. Examples: fully connected layers, convolutions, etc. Depending on the size of corresponding matrices, the dot-product category can be either math or memory-limited. If the matrices are of smaller sizes, these operations will end up being memory-limited and vice-versa.

2 Matrix Multiplication Background

Link to the Guide

GEMMs, or General Matrix Multiplications, serve as a crucial component in various neural network operations, including fully connected layers, recurrent layers like RNNs, LSTMs, or GRUs, and convolutional layers. GEMM is mathematically defined by the operation $C = \alpha AB + \beta C$, where A and B are input matrices, α and β are scalar multipliers, and C is an existing matrix that gets overwritten by the result.

2.1 Math and Memory Bounds

Consider a matrix A of size $M \times K$, matrix B of size $K \times N$, and matrix C, which is a product of A and B, of size $M \times N$. This will incur a total of $M \times N \times K$ fused multiply adds to compute the product. Since each multiply-add has two operations, we have a total of $2 \times M \times N \times K$ FLOPS. Now, to analyze the performance limiters of this matrix multiplication, we compare its arithmetic intensity (AI) to the ops:byte ratio of the GPU. Arithmetic intensity is given by:

$$AI = \frac{\#FLOPs}{\#BytesAccessed} = \frac{MNK}{MK + NK + MN}$$

Whether a given matrix multiplication is math or memory-limited depends on the values of M, N, and K. If the arithmetic intensity is lower than the ops:byte ratio, the operation will be memory-limited and vice-versa.

2.1.1 Tiled Matrix Multiplication

GPUs perform GEMM operations by dividing the output matrix into smaller sections called tiles, which are then assigned to thread blocks. Each thread block computes its designated tile by iterating through the K dimension in chunks, loading necessary data from matrices A and B, and performing multiplication and accumulation to form the output.

2.1.2 Tensor Core Requirements

NVIDIA GPUs leverage tensor cores to enhance the speed of tensor multiplications. Performance is notably improved when the matrix dimensions M, N, and K are multiples of 16 bytes due to optimized memory coalescing. For instance, if a 16-byte boundary aligns with four single-precision floating-point values (each 4 bytes), fetching 16 bytes at once can retrieve four elements simultaneously.

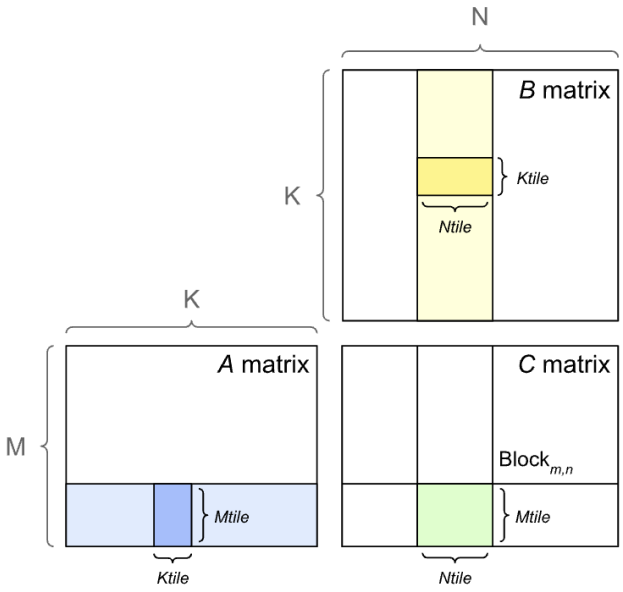

Figure 2: Tiled outer product approach to GEMMs. Source: `https://docs.nvidia.com/deeplearning/performance/dl-performance-matrix-multiplication/index.html`

2.1.3 Tile Dimensions and associated performance

Larger tiles facilitate greater data reuse, reduce bandwidth consumption, and increase efficiency compared to smaller tiles. However, this comes at the cost of fewer tiles running in parallel, which might lead to GPU underutilization. This balance between tile efficiency and parallelism indicates that for very large GEMMs, the trade-off becomes less significant: eventually, a GEMM operation has sufficient workload to use the largest possible tiles while still fully engaging the GPU.

For cuBLAS GEMMS, thread block tile sizes use a power of 2 dimensions. A few such efficient settings are 256×128 and 128×256. NVIDIA libraries can also tile along the K dimension if M and N are small but K is large, though this introduces a performance-limiting reduction step.

2.2 Dimension Quantization Effects

When GPU kernels are executed by launching multiple thread blocks, two important factors that impact GPU performance and efficiency come into play:

2.2.1 Tile Quantization

Tile quantization occurs when matrix dimensions are not divisible by the thread block tile size. This results in additional thread blocks being launched by the GPU, which do very little work, leading to underutilization of the GPU. The GPU utilization will be highest when the matrix dimensions are an integer multiple of the tile dimensions.

2.2.2 Wave Quantization

Wave refers to a set of thread blocks that run concurrently. In wave quantization, the total number of tiles is quantized to the number of multiprocessors (SMs) in the GPU.

The GPU utilization will be highest when the number of tiles is an integer multiple of the number of SMs in the GPU. When any dimension is not an integral multiple of the number of SMs, an additional tail wave of tiles is created along with the full wave tiles. An issue with this tail wave is that it takes nearly the same time to execute as the full wave, leading to lower performance and efficiency.

An important difference between this quantization is that tile quantization means work is quantized to the size of the tile, whereas wave quantization means work is quantized to the size of the GPU.

3 MI300X vs H100 vs H200 Benchmark

Link to the Blog

This blog by SemiAnalysis aims to curtail the excitement around AMD's MI300X GPUs by providing a realistic and unbiased assessment to the deep learning community. It compares the MI300X's performance against Nvidia's H100 and H200 GPUs in various training-focused benchmarks, debunking myths and exposing marketing overstatements by showing that the MI300X's theoretical specifications fall short in real-world scenarios.

The authors primarily attribute their findings to significant issues within AMD's software stack, citing a high number of bugs, an immature software environment, and a substandard Quality Assurance (QA) culture at AMD. While the article highlights these discrepancies, it also seeks to enhance the AMD ecosystem by offering constructive feedback and suggesting what AMD might consider to make the MI300X more competitive in the market.

3.1 Key Findings

- **Specs on paper don't tell the whole story**. You need to run benchmarks to see how things really perform in practice.
- NVIDIA's products work smoothly right out of the box, with no bugs. But AMD's are hard to use because their **software has a lot of issues**.
- AMD's performance in the real world doesn't match up with what they advertise. NVIDIA's performance might not meet expectations either, but it's closer.
- The biggest problem for the MI300X is **AMD's software**. By the time they fix it, competitors will likely have moved ahead.
- The MI300X might cost less overall, but when you look at performance for the money spent, it's **not as good as** NVIDIA's H100/H200 on AMD's standard software. However, if you use special development versions of AMD software, this can change.
- For training, **MI300X doesn't perform as well**, especially in matrix multiplication tests. Even on single-node setups, AMD's software lags behind NVIDIA's H100 and H200.
- Many of AMD's AI libraries are **based on** NVIDIA's, which can lead to problems and suboptimal results.

H100 vs H200 vs MI300X Basic Specifications			
	H100	H200	MI300X
Watts Per GPU (TDP)	700	700	750
All-in System Watts Per GPU	1,275	1,275	1,275
Memory Capacity (GB)	80GB	141GB	192GB
Memory Bandwidth (GB/s)	3,352	4,800	5,300
FP16/BF16 TFLOPS[1]	989	989	1,307
FP8 / FP6 / Int8 TFLOPS[1]	1,979	1,979	2,615
1. All FLOPS are dense			

Figure 3: On paper specs of H100, H200 and MI300X. Source: `https://semianalysis.com/2024/12/22/mi300x-vs-h100-vs-h200-benchmark-part-1-training/`

The on-paper TCO (Total Cost of Ownership) of MI300X seems to be extremely compelling, coupled with high performance. MI300X seems to perform well on every metric, as shown in Fig 3.

3.2 Actual Performance Benchmarks

3.2.1 GEMM Performance

- **BF16**: The H100 and H200 manage about 720 TFLOP/s out of the advertised 989.5 TFLOP/s. In contrast, the MI300X only hits around 620 TFLOP/s against its claimed 1,307 TFLOP/s. This means the MI300X is actually 14% slower than the H100 and H200 despite having a higher advertised performance.
- **FP8**: The H100/H200 achieves approximately 1,280 TFLOP/s from the marketed 1,979 TFLOP/s. Meanwhile, the MI300X reaches only about 990 TFLOP/s, making it 22% slower than the H100 for FP8 operations.

3.2.2 HBM Memory Bandwidth Performance

MI300X indeed has way better memory bandwidth than both the H200 and the H100. AMD MI300X offers 5.3 TB/s of bandwidth vs 4.8 TB/s for the H200 and 3.35 TB/s for the H100. *(High HBM is very useful for inference and sometimes for training as well).*

3.2.3 Single Node Training Performance

The performance figures for the H100/H200 are based on their standard, out-of-the-box capabilities, without any special tuning by Nvidia engineers. In contrast, AMD's performance metrics include results from custom, hand-crafted VIP builds and work-in-progress (WIP) development builds, which were necessary to bring their performance up to within 75% of the H100/H200's level.

- **BF16 training**: For the model training benchmark, the authors test four models: a simple GPT 1.5B DDP, standard Llama3 8B, Llama3 70B 4 Layer Proxy, and Mistral 7B v0.1. For all these models, the H100/H200 has better TFLOP/s compared to MI300X public releases/public nightly releases/Nov 25th build from source VIP image. None of the MI300X builds perform well on smaller models, such as GPT 1.5B. For Llama3 8B and Llama3 70B Proxy, the Dec 21st MI300X WIP development build performs better than H100/H200.
- **FP8 Training**: The H100/H200 achieves better TFLOPs as compared to MI300X on all the test models across all the builds.

3.3 Poor User Experience and Recommended Fixes

One of my favorite quotes from the reading is:

> *Getting reasonable training performance out of AMD MI300X is an NP-Hard problem.*

MI300X is not suitable for out of the box usage due to poor internal testing and a lack of automated testing on AMDs part. For instance, even though AMD's own advice is to use Pytorch's Flash attention, their version runs slower than a modern CPU, delivering less than 20 FLOPS. Plus, using the AMD Pytorch attention layer with `torch.compile()` caused crashes because of tensor rank errors, showing a lack of testing on software they release publicly. AMD needs to put more resources into testing and software development.

Also, the MI300X requires users to set many environment flags just to get it working, which is a hassle and a bad user experience. AMD should set these flags to helpful defaults so users can start working right away without fiddling around.

Moreover, AMD's libraries are adaptations (forks) of Nvidia's open-source ones. This is like trying to race on a track built by your competitor; it's tough to get ahead. For AMD to truly compete, they need to develop their own software from scratch.