Week 3 Reading Summaries: GPU Performance + MI300X vs H100 Readings

Amogh Patankar
University of California, San Diego
3869 Miramar Street Box #3037, La Jolla, CA- 92092
apatankar@ucsd.edu

Abstract

The articles for this week's reading summary describe the background on graphical processing units (GPUs), which are common modern hardware to execute machine learning workloads efficiently. Moreover, we read about operation execution and limitations, and matrix multiplication and its performance on GPUs and in other ML models. Finally, we also read about benchmarking of various hardware platforms, namely, H100, MI300X, and H200, with respect to latency, throughput, and total cost of ownership (TCO).

1 TensorFlow

1.1 Introduction

Graphical processing units are the basic building blocks of modern ML hardware, and there needs to be a proper understanding of what a GPU entails. Namely, we look at the structure/architecture, operation execution, performance estimation and limitations, and applications of deep learning and their respective limitations when utilizing GPUs.

1.2 GPU Architecture Fundamentals

The GPU has a parallel processing architecture, with processing and memory hierarchies. To summarize, it consists of streaming multiprocessors, on chip L2 cache, and DRAM. The SMs, DRAM, and L2 are responsible for arithmetic and instructions, data, and code access, respectively.

SMs have schedulers and instruction execution pipelines, with MAC/MAD operation being the most frequent in modern ML; data types have different MAC ops/clock, with a MAC consisting of two operations. So, to calculate FLOPs, we multiple throughput by two, and for GPU FLOPs, we multiple that value by the # of SMs and SM clock rate.

Nvidia GPUs allow various datatype execution, like INT8 and FP16; tensor cores are used to speed up matmul, MAC, and other operations on small matrices (eg. 4x4). In doing so, the computation can occur in higher precision, and non-executable ops are executed in other cores.

1.3 GPU Execution Model

There are two main concepts to connect thread count to GPU performance: 1) grouping threads to equally sized thread blocks, and execution [thread] switching. These two put together make up the nuances of the execution model of GPUs.

Because GPUs contain many SMs, each has pipelines to execute threads, and in turn, have shared memory. This results in a two-level thread hierarchy. To fully utilize a GPU when executing a

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

function, we'd use multiple thread blocks, and the number of threads should be a multiple of the # of SMs.

To maximize efficiency, we launch functions executing in waves of thread blocks, minimizing tail effect (underutilization of GPU).

1.4 Understanding Performance

GPU Performance is limited by memory bandwidth, math bandwidth, and latency. Memory time = (# of bytes accessed in memory)/(process memory bandwidth). Math time = (# of ops)/(processor math bandwidth). All in all, algos are math limited if arithmetic intensity > processor BW_{math}/BW_{mem}.

1.5 DNN Operation Categories

1.5.1 Elementwise Ops

These are unary/binary ops, layers in this category perform mathematical ops independent of other tensor values.

1.5.2 Reduction Ops

These ops produce values computed over a range of input tensor vals. Examples include pooling layers, and SoftMax.

1.5.3 Dot-Product Ops

These ops are exressed as dot-products of elements from two tensors. Examples include FC layers, convolutions, and matmuls.

1.6 Summary

When evaluating GPU performance, we look at following criteria:

- # of SMs on GPU, determine ops:bytes ratio (i.e. $\frac{BW_{math}}{BW_{mem}}$)
- Compute arithmetic intensity, check parallelism with thread blocks and SM count.

Likely performance limiters are often latency, math, and/or memory (last two depend on ops:byte ratio).

2 Matrix Multiplication Background User Guide

2.1 Background

General Matrix Multiplications, known as GEMMs, make up neural network operations, in FC, recurrent, and convolutional layers alike. GEMMs are essentially an operation defined as $C = \alpha AB + \beta C$, including two matrices, two scalar inputs, and a pre-existing matrix.

2.2 Math and Memory Bounds

Given a product of two matrices, we have a total of M * N * K fused MACs, i.e. 2 * M * N * K, as each MAC is 2 FLOPs. The arithmatic intensity of a GEMM is as follows: $\frac{2*M*N*K}{2(M*K+N*K+M*N)}$

2.2.1 GPU Implementation

GEMMs' output matrices are partitioned into tiles, assigned to thread blocks on GPU. We step through K dimension in tiles, loading values from A and B, and performing MAC.

2.2.2 Tensor Core Requirements

Tensor cores maximize speed of tensor multiplies, and performance of GEMMs improves when M, N, and K are aligned to multiples of 16 bytes. For example, with FP16 data, each element is 2 bytes, hence dimensions should be multiples of 8. Scaling down (multiples of 16, then 8, then 4) is recommended.

2.2.3 Typical Tile Dimensions in cuBLAS and Performance

Larger tiles reuse more data, using less bandwidth, and increase efficiency compared to smaller tiles. However, you run fewer large tiles in parallel, which may lead to tailing problem as mentioned earlier. Tradeoff between tile efficiency and parallelism suggests that larger GEMMs impact the tradeoff less. Flipped, a smaller GEMM causes a larger tradeoff in the sense that the GPU won't run at maximum efficiency.

2.3 Dimension Quantization Effects

Tile and wave quantization afect execution efficiency.

2.3.1 Tile Quantization

This is when matrix dimensions aren't divisible by thread block tile size; for example, with 256x257 data vs 256x256 data using 128x128 tiles, utilization will be 6 vs 4 tiles, a 50% increase in ops, when in reality only has .39% more data.

2.3.2 Wave Quantization

Tile quantization quantizes problem size to the size of each tile, wave quantization quantizes the total # of tiles to the # of multiprocessors on the GPU. Essentially, wave quantization quantizes the work to the size of the GPU, not the size of tiles of thread blocks on the GPU.

3 MI300X vs H100

3.1 Intro

This article compares the performance of AMD and Nvidia GPU processors, in terms of specs and TCO. They find that the advantage of the MI300X isn't valid due to a lack of AMD's SW stack and less testing, including bugs.

3.2 Key Findings

They provide many key findings, but essentially, it all boils down to this: Nvidia has an incredible experience out-of-the-box (OOTB), while AMD struggles mightily. They compared TCO and performance, where AMD has a lower TCO but has worse performance too. They also note that AMD requires better coverage with regards to ROCm, internal testing, debugging, benchmarking, and scalability.

3.3 AMD vs Nvidia

While the H100 was launched in late 2022, AMD's alternative, MI300X was introduced in late 2023. It has a extremely low TCO on paper, and represents higher performance as well (on paper).

3.4 General Matrix Multiply (GEMM)

When benchmarking for GEMM for BF16, the H100 and 200 achieve 720 TFLOP/s, while the MI300X hits only 620 TFLOP/s. When compared to their marketed values, Nvidia struggles by only about 20%, while AMD drops by 50%. This gap only widens with FP8, looking at a 20% gap as opposed to 10% between Nvidia and AMD.

3.5 HBM Performance

The MI300X has better memory bandwidth compared to the H100/200, useful for inferencing and sometimes training.

3.6 Training + Testing Methodology

SemiAnalysis used a unique strategy to test performance; they used MLPerf's GPT3 175B Training to measure train time to convergence, but was difficult to run. Hence, they ran GPT 1.5B DDP, Llama3 8B, Llama3 70B, and Mistral 7B v0.1.

3.7 Single Node Training Performance

When comparing single node training performance, SemiAnalysis found the following:

- BF16: MI300X underperforms H100/200 for GPT/Llama3-8B/Mistral 7B, barely outperforms Nvidia on Llama3-70B using a private repo (not valid).
- FP8: MI300X underperforms on all benchmarks

3.8 Multi-Node Training Performance

Regardless of precision, H100 crushes the MI300X, and the H200 wasn't available for multinode training.

3.9 Scale Up NVLink/xGMI Topology

Scaleup fabric is important for GPU clusters as it provides fast tensor parallelism. As such, NVLink for Nvidia provides 450GB/s per GPU with 8 GPUs, while xGMI provides 448GB/s for 8 GPUs for AMD. However, xGMI provides point-to-point bandwidth, so for 8 GPUs, we only get 64 GB/s per per GPU. For NVLink, it's a switched topography, meaning we get full 450 GB/s due to NVSwitches.

3.10 Single Node NCCL Collective

For all tests undertaken, the H100 beats the MI300X, regardless of # of GU, precision, etc.

3.11 MultiNode RCCL/NCCL Collectives + Scaleout Network Benchmarks

FOr all the GPUs here (H100/200 + MI300X), each GPU connects to other nodes using scaleout network, with a 400G NIC. In any case, Nvidia's GPU's performance beats MI300X, due to AMD's RCCL being a fork of Nvidia's NCCL, restricting access to up-to-date features.

3.12 AMD Forked Libraries

SemiAnalysis points out that AMD forks off of Nvidia's ecosystem libraries. For NCCL, APex, Megatron, and Pytorch Eagers Kernel, AMD forks to ROCm/RCCL, Apex, Megatron, and PyTorch Eagers Kernel. This is a huge limitation for AMD MI300X performance due to SW restrictions.

3.13 Recommendations to fix AMD SW

SemiAnalysis provides a thorough recommendation to AMD in order to fix their software stack to allow for optimal MI300X performance. To summarize, they conclude that SW engineers (#, quality), lack of resources available to AMD engineers, integration of ROCm with Meta's PyTorch team, and using MLPerf/TorchBench to benchmark.