ML Performance Reading Group Notes

EleutherAI

1 Session 1: GPU Architecture, CUDA, NCCL

1.1 High-Level Summary

- **GPU Architecture**

  Key Idea

  One-sentence difference: CPUs are more latency-optimized and GPUs are more throughput-optimized.

  So what is the difference between GPUs and CPUs?

  In one sentence: CPUs are more latency optimized and GPUs are more throughput optimized.

  - CPU = sports car
  - GPU = bus

  Example: Latency for GPU operations like register access and FMA, and L1/L2 access, can be much slower than on a CPU (but the GPU wins by running **many** things in parallel).

  CPU architecture:

  - A small number of cores, each with its own L1 cache and control logic attached.
  - L2/L3 caches sit outside / above the cores (shared at higher levels).

  GPU architecture:

  - Many cores (lots of parallel lanes).
  - L1 cache + shared memory close to execution units; L2 cache shared; DRAM = high bandwidth memory (HBM).
  - Optimized for moving data fast and doing lots of arithmetic per unit time.

  These cores exist in SMs (Streaming Multiprocessors). There might be a few to 100+ SMs depending on how big the GPU is.

  There's a big variety of compute units on something like an A100/H100 (different ALUs for different floating-point types). GPUs also have specialized cores for matrix multiplication (Tensor Cores). There are register files that support fast context switching.

Key Idea

Warp concept (SIMT): A warp is an atomic scheduling unit of 32 threads. Threads in a warp execute the same instruction stream on different data (SIMT/SIMD-like behavior).

A GPU deals with warps as parallel groups of computation. Each SM has many warps; the warp scheduler rotates between them to hide memory access latency. While one warp waits for memory, the scheduler swaps to another warp so it hides the stall.

Divergence handling: If threads in the same warp take different control-flow paths (data-dependent logic), they serialize those paths and overall throughput drops.

Key Idea

Zooming out: whole GPU picture. In Ampere/Hopper-class systems:

– PCIe is the host interface (bus moving data between host and device).

– NVLink is a high-bandwidth interconnect between GPUs.

Latency comparison: memory vs compute (cleaned + preserved)

Speed breakdown (slow $\rightarrow$ fast):

$$\text{Global mem} < \text{L2} < \text{L1 / Shared} < \text{Registers} < \text{FMA} < \text{Tensor Cores}$$

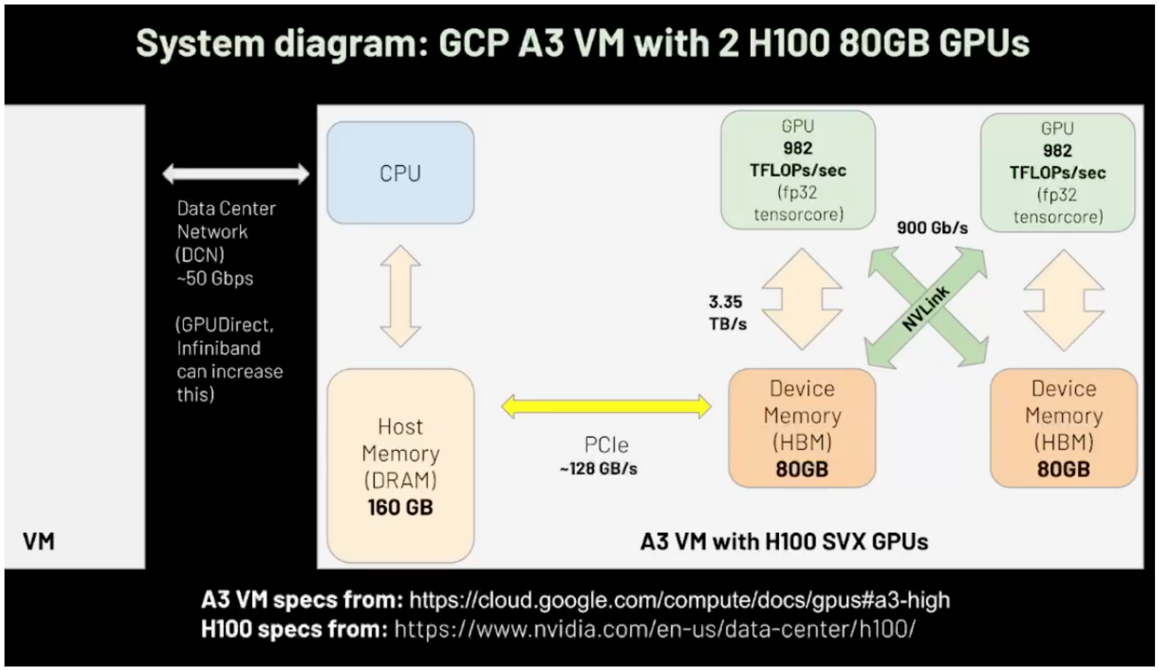
All we need to do is hide **memory latency** or **network communication** (those are often the slow parts).

Important

Performance intuition: GPUs usually have way more compute than memory bandwidth. If you can't feed the compute fast enough, compute units go idle.

System diagram (INSERT filled with image placeholder)

System Diagram


System diagram: GCP A3 VM with 2 H100 80GB GPUs
VM
Data Center Network (DCN) ~50 Gbps
(GPUDirect, Infiniband can increase this)
CPU
Host Memory (DRAM) 160 GB
PCIe ~128 GB/s
GPU 982 TFLOPs/sec (fp32 tensorcore)
GPU 982 TFLOPs/sec (fp32 tensorcore)
3.35 TB/s
900 Gb/s
NVLink
Device Memory (HBM) 80GB
Device Memory (HBM) 80GB
A3 VM with H100 SVX GPUs
A3 VM specs from: https://cloud.google.com/compute/docs/gpus#a3-high
H100 specs from: https://www.nvidia.com/en-us/data-center/h100/

When doing massive distributed training runs on 1000s of GPUs and 100s of hosts, this is what we are working with.

Host/device are connected to the data center network (DCN) at around 50 Gbps. If you want to increase this connection speed you use GPUDirect or InfiniBand.

Above is an example of a VM with 2 H100s.

- Bandwidth for PCIe is 128 GB/s.
- Moving to GPUs:
  * Device memory is 80 GB.
  * Between device memory (HBM) and SRAM there is 3.35 TB/s.
  * NVLink between both GPUs is 900 GB/s.

The bottleneck here is the connection between the data center hosts. If you want to send to a neighboring host, data often needs to go:

$$\text{device} \rightarrow \text{host} \rightarrow \text{NIC / buffers} \rightarrow \text{DCN}$$

(This part is usually the bottleneck.)

Ways to combat this: RDMA (skip host copies by allowing the GPU to talk to the NIC directly, avoiding extra copying).

Now the compute contrast on a GPU:

- H100 uses FP32/TF32/etc. and does matmul on Tensor Cores at massive rates (compute is huge).
- This number looks inconsistent with data movement because you cannot feed compute fast enough unless you have techniques to cover that latency.

One solution in the compiler:

- Kernel fusion → more FLOPs per data that arrives from HBM (better reuse, fewer reads/writes).

FLOPs achievable and the data movement imbalance grows as we use lower precision data types like FP16 or INT8.

Key Idea

Factory analogy (preserved + cleaned):
Think of the GPU as a factory and memory/storage as the warehouse.
If the conveyor belt (bandwidth) is saturated, you get suboptimal throughput even if the factory (compute) is fast.

VOCAB (preserved + clarified):

- Arithmetic intensity = FLOPs / bytes moved over HBM (device memory).
  Compute it via profiling, compare to hardware ratio:
  - If workload ratio $<$ hardware ratio $\Rightarrow$ not compute-bound (likely memory or network-bound).
  - If workload ratio $\approx$ hardware ratio $\Rightarrow$ likely compute-bound (need better kernels or more GPUs).
- MFU (Model FLOPs Utilization) = value between 0 and 1: actual FLOPs achieved / achievable FLOPs on hardware.

Quick reference tables (added for memory recall)

Concept	**CPU (sports car)**	**GPU (bus)**
Optimization target	Latency	Throughput
Parallelism	Low–medium	Very high
Best at	Branchy/control-heavy code	Dense math + parallel workloads
Bottlenecks	Single-thread latency	Memory bandwidth + communication

Link	**What it connects**	**Why it matters**
PCIe	CPU/host ↔ GPU	Host-device bandwidth/latency
NVLink	GPU ↔ GPU	High BW multi-GPU on-node
InfiniBand/RDMA	Node ↔ node	Faster distributed training comms
GPUDirect RDMA	GPU ↔ NIC	Avoid host copies (lower latency)

- **CUDA**

  How do we make use of this highly parallel throughput machine?

  CUDA uses `nvcc` compiler and provides a set of software abstractions: a parallel model to organize computations and map them to hardware (breaking them into blocks/warps).

Key Idea

CUDA mapping (fast recall):
Grid → many blocks across the GPU **Block** → runs on one SM **Warp** → 32 threads scheduled together

Kernel + matmul mapping (preserved + cleaned)

Kernel → (example) output matrix.

Let's say you have an output matrix:

- (1) Divide it into thread blocks (dimensions = how many threads tall and wide: `blockDim.y`, `blockDim.x`).
- (2) Once thread block is defined, the output has to be broken into the same blocks.

`gridDim.x, gridDim.y` = how many blocks wide/tall
`threadIdx.x, threadIdx.y` = each thread's local coordinates inside the block

Important

INSERT (filled): naive matmul CUDA code (educational baseline)
This is the simplest version (not optimized). Real performance comes from tiling + shared memory + Tensor Cores.

Listing 1: Naive CUDA matmul (each thread computes one C element)

```
__global__ void matmul_naive(const float* A, const float* B, float* C,
                             int M, int K, int N) {
    // A: MxK, B: KxN, C: MxN
    int row = blockIdx.y * blockDim.y + threadIdx.y; // i
    int col = blockIdx.x * blockDim.x + threadIdx.x; // j

    if (row < M && col < N) {
        float acc = 0.0f;
        for (int k = 0; k < K; k++) {
            acc += A[row * K + k] * B[k * N + col];
        }
        C[row * N + col] = acc;
    }
}
```

Simple CUDA program outline (preserved + cleaned)

- (1) Define input matrices A and B, sizes $M \times K$ and $K \times N$.
- (2) Matmul output: C, size $M \times N$.
- Divide output matrix into thread blocks.
- Use ceiling/floor division so there are always enough blocks to cover the full output matrix.
- Each thread computes one element in C (dot product).
- Kernel launch uses `<<<grid, block>>>` to map computation to hardware.

CPU vs GPU implementation (preserved + clarified):

- CPU: loops compute dot products and write into C; uses strides to index arrays.
- GPU: compute (i, j) for C using block index + thread index:

$$i = \texttt{blockIdx.y} \cdot \texttt{blockDim.y} + \texttt{threadIdx.y}, \quad j = \texttt{blockIdx.x} \cdot \texttt{blockDim.x} + \texttt{threadIdx.x}$$

When we run a benchmark on the program, we see a large speedup on big matrices.

- **NCCL**

Key Idea

NCCL: a collective communication library → distributed compute primitives for multi-GPU ops.

Example: broadcast data to multiple GPUs, or do a reduction across GPUs. It gives tools to scale from one GPU to many GPUs/nodes.

Collective ops (preserved + cleaned + table added)

Rank is a unique identifier for a GPU (e.g., one node broadcasts weights to all others).

Collective	**What it does (your wording, clarified)**	**Common use**
Broadcast	One rank sends data to all other ranks	Send weights/config
AllReduce	Reduce across all ranks and store result in every rank buffer	Gradient sync (DP)
Reduce	Reduce across ranks, result stored in **one** rank	Aggregation
AllGather	Gather shards so every rank ends with full data	Unshard (FSDP)
ReduceScatter	Reduce, then scatter shards so each rank keeps only a shard	FSDP grads

Used for gradient sync: data-parallel training. Each GPU has a different model replica, runs a separate mini-batch, forward + backward, produces partial grads, then sync grads and perform the weight update.

AllGather (preserved + clarified): used when data is sharded across multiple devices (e.g., a layer split across devices). Before forward pass or for certain steps, you may need to unshard so everything is available.

ReduceScatter (preserved + clarified): instead of every rank having a full copy, each rank only has a shard of the aggregated data. For example, each device has partial grads; do reduce-scatter so each device keeps a shard.

Then after the update, you can do **AllGather** again (common in FSDP workflows).

- **NEW VIDEO: FlashAttention**

Important

Key idea (preserved): be I/O aware!
Often it is faster to recompute than to load from HBM. Increase arithmetic intensity.

Motivation: long context is really hard with quadratic attention. Many heads and blocks. Each head must store, load, and compute huge matrices: you can run out of device memory, or need heavy offloading.

Arithmetic intensity = FLOPs / memory access.

More compute is available than bandwidth. When you load something, you don't want to do more saves than necessary. Crank arithmetic intensity up until you are compute-bound.

Standard attention device work (preserved + cleaned)

Device work:

1. Load blocks of query and key matrices onto compute from HBM.
2. Softmax.
3. Write back.
4. Read again.
5. Multiply by value matrix.

Often we do `torch.matmul` and everything "looks easy," but on GPU it has to tile it and decompose operations onto tensor cores.

On-chip view + tiling (preserved + clarified)

On chip: query and keys are loaded, softmax is applied. But softmax needs the entire row for stability—bad for long context (32k/64k).

Solution: tiling. GPUs do many small tile-wise accumulations:

- Load some tiles into SRAM.
- Compute smaller matmul accumulations (Tensor Cores).
- Accumulate partial results.

Key Idea

Tensor Cores (preserved + clarified):
Specialized hardware for dense matrix operations. Most peak FLOPs come from Tensor Cores.
If you do dense ops without Tensor Cores, performance will be much lower.

Tensor Core intuition: it operates on small blocks (e.g., 16×16 fragments).

Softmax stability (INSERT filled)

INSERT Softmax (x_i) formula:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Why subtract max?

$$\text{softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

Subtracting $\max(x)$ improves numerical stability (prevents overflow), without changing the result.

Online / tiled softmax idea (preserved + clarified)

We can take separate softmax tiles, but eventually we need the whole row. While tiling, we track:

- running max (m)
- running denominator (ℓ)

When we update the max, we rescale numerator/denominator so we never materialize huge e^x values.

FlashAttention algorithm (filled with a faithful outline)

Below is a clean outline (the "shape" of the algorithm you described). You can replace variable names with the paper's exact notation later:

Key Idea

FlashAttention outline (tile-wise):
Process keys/values in tiles, maintain running softmax stats, accumulate output in SRAM.

Listing 2: FlashAttention-style tiled attention (high-level pseudocode)

```
# Inputs: Q [B,H,seq,d], K [B,H,seq,d], V [B,H,seq,d]
# Output: O [B,H,seq,d]

for each query tile Qt:
    m = -inf            # running max per row
    l = 0               # running denom per row
    O_acc = 0           # running output accumulator

    for each key/value tile (Kt, Vt):
        S = Qt @ Kt.T             # scores tile
        S = S * scale             # scale
        S = apply_causal_mask(S)  # if causal

        m_new = max(m, rowmax(S))
        P = exp(S - m_new)        # stable exp for this tile

        l = l * exp(m - m_new) + rowsum(P)
        O_acc = O_acc * exp(m - m_new) + (P @ Vt)

        m = m_new

    O = O_acc / l                 # normalize at end
```

Key thing: based on SRAM sizes we choose block sizes so tiles fit in fast memory. Then we do normal tiling computation, maintain m and ℓ, and do final rescaling. Because the math is associative after rescaling, we can change operation order to be faster.

This process cut down training time significantly, especially for long context.

FlashAttention 2 and 3 (preserved + cleaned)

FlashAttention 2: reduced many redundant scaling operations.

- Tensor Cores are so fast that pointwise ops in SRAM are relatively cheap.
- Swap loop order to allow better parallelization of Q tiles; long context often means low batch size, so better residency over hardware matters.

FlashAttention 3: uses fancy async features of H100:

- async memory ops
- Tensor Core batching/pipelining

Triton kernel walkthrough (preserved + cleaned, kept long on purpose)

Just to go over the Triton kernel implementation for basic optimization:

This is a simpler implementation of FlashAttention intuitively.

Simple intuitive implementation of autograd: Starts with a forward pass.

First thing: define the grid (how to parallelize the computation on the hardware).

We are defining the thread blocks, but operating at the kernel level.

We are dividing our sequence length by our block size for Q, and each element is a query vector, then chunking up each vector into Q blocks.

Basically, we are parallelizing across the queries and batch attention and then each attention head.

Using Triton to launch kernel: We pass in strides from torch tensors; later when we load blocks of memory, strides help us index into arrays.

Looking at the actual kernel: We find which thread block we are in: query block, batch idx, head idx.

We decompose batch * head to figure out which idx we are looking at as well as the attention head.

Now we go from base Q pointer to the offset where this begins, again using strides to go to the particular subtensor in memory.

Make block pointer: gives us a 2D block of pointers for elements of memory we want to pull.

Once we have the parent subtensor, we need the Q block and grab a subset of Q vectors of block size Q with offsets.

KV: similar logic; note that K needs to be transposed with respect to Q. We iterate through each K block.

After pointer arithmetic to get K and V: We have our m_i called qk max, the ℓ_i which is our global softmax.

We load Q block pointer into SRAM:

Inner loop: We go through key/value block one at a time (causal attention: only references the past; stopping at the diagonal, going blocksizekv at a time).

We use offsets to create 2D row/column masks, jump to row id and compare for causal mask.

Then HBM $\rightarrow$ SRAM, matmul, scale, apply causal mask.

Compute m_i (local max for each row), compute corrective factor: subtract previous global max minus local scores then exponentiate, adjust scaling (because tiling).

Compute P block and row sum as cumulative denom.

To get new global softmax denom: corrective factor * prev global denom + row sum.

Update global max.

Update cumulative output O block: apply corrective factor to previous output and add local matmul contribution.

Iterate to the next block of keys and values by advancing block pointers:

- K block: iterate along seq by block size kv
- V: iterate by block size kv

After iterating across all K blocks for a single Q block, normalize by denom and write output block back to HBM.