Practical Aspects

Henry Bosch
MATH 289 Stanford University
November 9, 2026

Understanding Hardware
Performance and cost estimation



Purpose of This Talk

- Understand how to reason about practical aspects of running deep learning workloads.
- Intended for final projects which are performing training or inference of AI models on GPUs.

CPU and GPUs

CPUs (Central Processing Units)

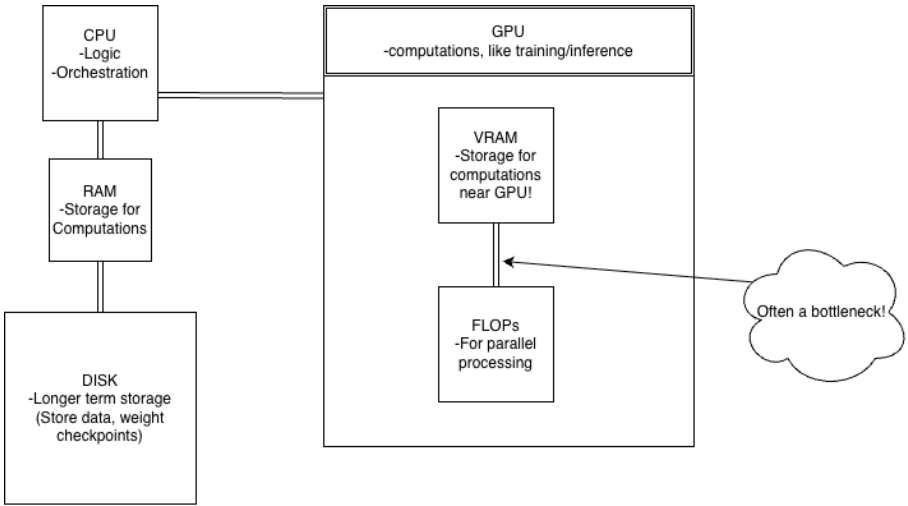
Optimized for sequential processing. They have a few powerful cores designed to handle complex logic branching and single-threaded tasks quickly.

GPUs (Graphics Processing Units)

Optimized for parallel processing. They consist of thousands of smaller, simpler cores designed to execute repetitive mathematical operations (like matrix multiplication) at the same time.

Essential for AI, because the computations occurring in neural networks mainly consist of matrix multiplications.

The Anatomy of a Deep Learning Machine (Oversimplified)



Three quantities to remember for your GPU

Quantity	Units	Notes
Memory capacity	GB	Can the data fit? (Example: model weights)
Memory bandwidth	GB/s or TB/s	Data transfer per unit time (we care about between GPU compute and GPU VRAM) Compute throughput
FLOP/s	A FLOP is a floating point operation. An addition or a multiplication would be considered one FLOP.	Floating point computations per unit time. Depends on what **precision** you are considering

Model Weights

1. A model will come with a set of weights, which are the tensors that represent the linear transformations forming the neural network.
2. These weights are (almost always) floating point. Floating point datatypes are classified by their widths (number of bits). Example: FP64 is 64 bits, FP32 is 32 bits. But there are more distinctions.
3. FP16: 16 bits, 10 bit mantissa, 5 bit exponent
4. BF16: 16 bits, 7 bit mantissa, 8 bit exponent
5. FP8: 8 bits. Can either do 3 mantissa bits and 4 exponent bits (E4M3) or 2 mantissa bits and 5 exponent bits (E5M2).
6. FP4: 4 bits. 2 exponent bits, 1 mantissa bit.
7. NVFP4, MXFP8: datatypes with extra *scales* that scale multiple weights at once.

How Big is Your Model?

For N parameters stored with b bits each, the idealized weight storage is

$$M_{\text{weights}} = N\frac{b}{8} \text{ bytes.}$$

Storage format	Bytes / parameter	8 billion parameters
32-bit	4	32 GB
16-bit	2	16 GB
8-bit	1	8 GB
4-bit	0.5	4 GB

These are approximate, since some models have parameters of different bit widths. Unlike scientific computing, where high precision is useful, in machine learning, very little precision (say 4 bits per weight) is needed, if everything is well-tuned.

How Big is Your Model?

Quick math:

- 40B parameters, FP32?
- 200B paramters, FP16?
- 10B parameters, FP8?
- 120B parameters, FP4?
- 1T parameters, FP4?

Other notes:

- You need to load the model's weights fully onto the GPU in order to perform inference *without* swapping from disk (which would take really long). Therefore, you need to allocate VRAM on your GPUs to store all the weights of your model.
- If you have multiple GPUs, you can use parallelism strategies to *shard* the model across them. In the simplest setting, you just split the model onto separate GPUs.

Other Notes on Memory

- You need to load the model's weights fully onto the GPU in order to perform inference *without* swapping from disk (which would take really long). Therefore, you need to allocate VRAM on your GPUs to store all the weights of your model.
- If you have multiple GPUs, you can use parallelism strategies to *shard* the model across them. In the simplest setting, you just split the model onto separate GPUs.
- When performing training, memory is allocated to both the model's weights, and the model's optimizer state. **You should think carefully about this** in order to avoid an out-of-memory error (OOM) due to misallocation, which will crash your entire training run (unless you have built a failsafe, but the OOM will likely reoccur since something is wrong with how you accounted for the total memory available).
- If you are running out of memory, LoRA steeply reduces the memory required, because it only computes gradients for a set of low-rank perturbations of the weight matrices. Researchers have shown that LoRA often tends to perform as well as a full fine-tune. Strongly consider using LoRA if you are trying out fine-tuning.

How to Measure Speed

Latency

Time for one request to finish.

Question for interactive work:
How long am I waiting?

Throughput

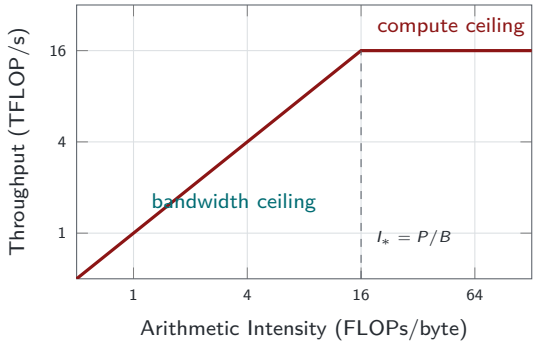
Completed requests or tokens per unit time.
Aggregated over parallel processing.

Question for an overnight run:
How much useful work finishes?

The roofline model

F: floating-point operations
Q: bytes moved from device memory
P: peak arithmetic rate
B: device-memory bandwidth
R: achieved arithmetic rate

$$I = \frac{F}{Q}, \qquad R \le \min(P, BI).$$



The vertex of the graph is called the **ridge point**. If you are at the bandwidth ceiling, you are *memory bandwidth-bound*. If you are at the compute ceiling, you are *compute-bound*. If you are in the interior (under the curve), you are *latency-bound*.

Illustrative hardware: $P = 16$ TFLOP/s, $B = 1$ TB/s.

Training vs Inference

- *Training* is taking data (golden inputs and outputs) and updating the model's weights so that the model's outputs on given inputs match the golden outputs. Training involves, in its most basic form, a *forward* and *backward* pass.
- *Inference* is running inputs through a model. Because there is no backpropagation, much less memory is needed.

LoRA

- **LoRA** is a technique which keeps weight matrices frozen, and adds a trainable low-rank update. I.e. we store $W \in \mathbb{R}^{m \times n}$ a fixed weight matrix, and a low rank update $L = AB \in \mathbb{R}^{m \times n}$, where $A \in \mathbb{R}^{m \times r}$ and $B \in \mathbb{R}^{r \times n}$, where $r \in \mathbb{N}_+$ is the rank. Then $W' = W + AB$ is the weight used for computations.
- Thus, we reduce the trainable parameters from mn to $r(m+n)$.
- **Question.** Given m and n, what is the maximum value of r which you would use?
- This save compute and memory. It also allows for "swapping" the LoRAs out while keeping the based weights the same, reducing memory transfers needed for different fine-tuned versions of a model.
- It often performs as well as a full fine-tune. Even $r = 1$ can perform well! Google for best-practices.
- **Question.** How would you backpropogate through a LoRA?

Prefill and decode

LLM Inference consists of processing a prompt in parallel (prefill), followed by generating a response token by token (decode). The compute shapes of these two operations are different, which results in them having different billing costs.

Phase	Work	Common regime
Prefill	Process the prompt tokens together	Often compute-bound
Decode	Produce new tokens autoregressively	Often bandwidth-bound at small batches

Estimating API spend

For ordinary input and output tokens priced per million,

$$C_{\text{API}} = \frac{p_{\text{in}} T_{\text{in}} + p_{\text{out}} T_{\text{out}}}{10^6},$$

where T_{in} and T_{out} are the number of input and output tokens, respectively, and p_{in} and p_{out} are prices from the service provider (you can find them on their website).

- Sum usage over **all billable calls**, including repeated context and tool messages.
- Add separate cache, tool, and other billing categories when the provider uses them.
- For extended thinking, visible answer length can understate billable output, due to hidden thinking tokens.
- Subscription and API terms differ. Claude bills API usage separately, with API credits included in some plans.

Routes to compute

Route	Examples	What to check
Hosted notebook	Google Colab	Hardware and fluctuating session limits
Managed execution	Modal	Requested GPU, concurrency and billing
Stanford cluster	Sherlock, Marlowe	Scheduler and allocation access
GPU instance	Lambda, Nebius	Instance resources and billed uptime
Model API	Hosted providers	Token categories and usage

Stanford: Sherlock requires an explicit GPU request. Marlowe access for students and postdocs is through a faculty PI's allocation.

Training as a service

Tinker (but there are others too)

- Write the training loop and choose data and objectives locally.
- Call a service that manages distributed GPU execution.
- Uses LoRA fine-tuning through an API for supported models.
- Tinker is a great way to fine-tune *large* open source models (up to 2.8T parameters for Kimi-K3.1-Pro) while avoiding implementation details (which become more complex the larger the model, as the program must run on multiple gpus, or nodes of gpus).

Worked API budget

Example

1,000 calls, each using 2,000 input tokens and 4,000 output tokens.
Assumed rates: \$2 / million input and \$8 / million output.

$$C_{\text{call}} = \frac{2(2{,}000) + 8(4{,}000)}{10^6} = \$0.036,$$

$$\boxed{C_{\text{run}} = 1{,}000 \times \$0.036 = \$36.}$$

Question. In this price model, are you actually paying a constant amount per compute spent by the provider?

Estimating self-hosted cost

For g GPUs, h billed hours, and rate r per GPU-hour,

$$\boxed{C_{\mathrm{GPU}} = g\,h\,r.}$$

1. *Dedicated* means that you own the GPUs and cannot be kicked off (more expensive).
2. *Spot* means that you can be pre-empted by someone else at a higher priority level at any time (cheaper).
3. **Make sure to look at** the CPU, disk space, RAM, and interconnects between GPUs. These are all important!

Hypothetical example

2 GPUs for 6 billed hours at \$3 per GPU-hour:

$$2 \times 6 \times \$3 = \$36.$$

Training FLOPs and their limits

For approximate **dense full training**, let N be parameters and D tokens processed:

$$\boxed{F_{\text{train}} \approx 6ND}$$

- The estimate includes forward and backward work. Attention and architecture can change the count.
- LoRA and reinforcement-learning pipelines need their own accounting.

Questions & Discussion

- Questions?
- What are the current bottlenecks you are facing on your projects?