CS 2281: How to Train Your Foundation Model

Sham Kakade

Fall 2024

Lect 1: Course Logistics + Auto-Differentiation / Compute Primitives

Sham Kakade and Nikhil Anand

CS 2281: How to Train Your Foundation Model
Fall 2024

Today

- Course Logistics
- A Word on Foundation Models
- Auto-Differentiation & computational graphs
  - checkpointing
- GPU/Infrastructure Background
- AD with Transformers

Course Logistics

Info:

Check the website for all policies:

https://shamulent.github.io/CS_2281_2024.html

- The course will be in person only.
  - Attendance/participation is expected.
  - A number of guest lectures
- Course requirements
  - 3 HWs, first one by Monday
  - Final Project in groups of 3-4.
- Course Staff: **Aayush Karan, Clara Mohri, Han Qi**

Background Knowledge & Responsibilities

Grad level topics: self/group study strongly encouraged.

- Transformer models
- Strong ML background (stat, lin alg)
- Python programming
- applied DL experience a plus
- motivated to learn material offline that you are not familiar with…

Today

- Course Logistics
- ✓ A Word on Foundation Models
- Auto-Differentiation & computational graphs
  - checkpointing
- GPU/Infrastructure Background
- AD with Transformers

Foundation Models

What is a Foundation Model?

A “model that is trained on broad data such that it can be applied across a wide range of use cases.” (wiki).

Examples + Grapevine Estimates (of #params, training data, compute):

- LLMs:
  - GPT3.5: 200B param model, trained on 1-5T tokens
  - GPT4.0: 1.6T (8x200B MoE model), ≈10T tokens, (flop equiv) 30K for several months
  - Gemini: 2T param model (also MoE?), ≈10T tokens (trained on TPUs)
  - Llama 3.1: 405B (dense), ≈10T tokens
- Code: Copilot ≈10-20B (?),
- Images/Video: MidJourney/Sora ≈10-20B (?), 10K gpu for 1 month (?)
- Bio: AlphaFold

This course: Training Foundation Models

What are the issues related to training foundation models?

- Models/architectures
- Algorithms
- Systems/Hardware Constraints
- Data:
    - Pre/mid/post training
    - supervised/instruction fine-tuning; RLHF

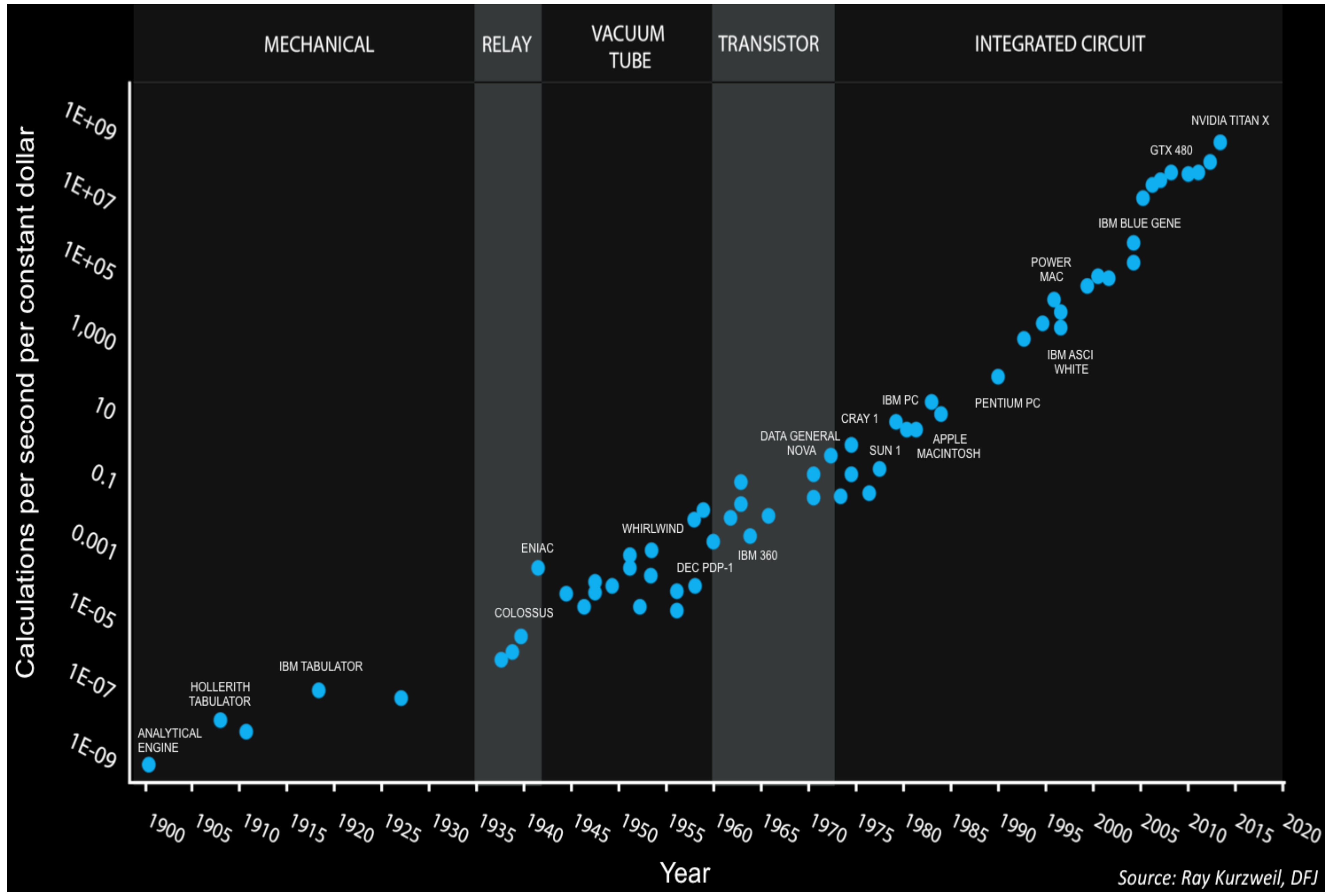
Other topics:

- Inference
- Reliability

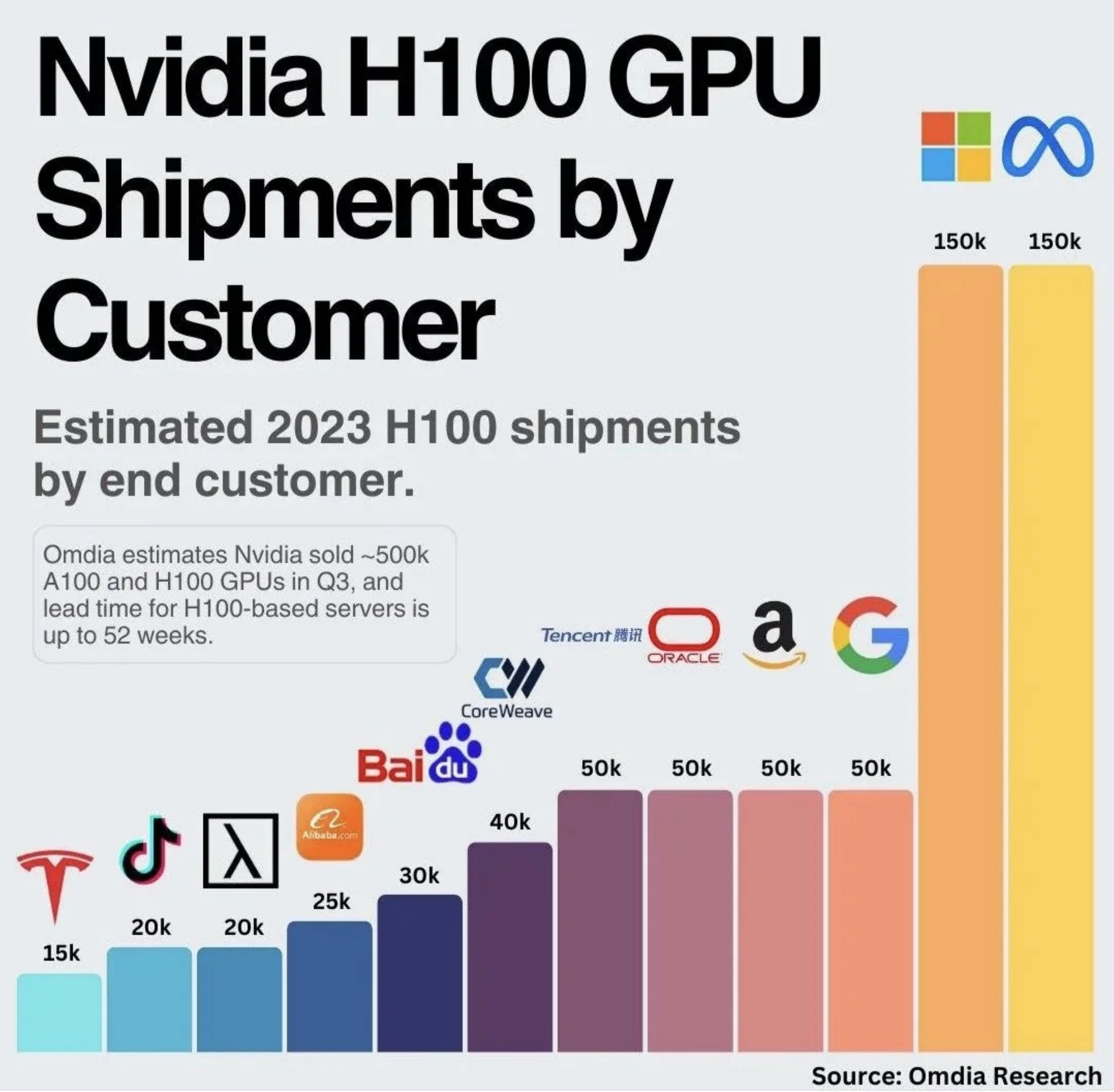
Should we bet on scale?

The course is (partly) designed around “scale” being a key component to human level AI.

- Current results used substantial amounts of computation.
- Moore’s law in flops per dollar.
- Markets:
  - Aggressive growth in compute infrastructure: **300K H100s at $10B** (e.g. Meta in 2024)
  - Nvidia market cap ≈ **3T**
    - reported profit: ≈ 30B
    - market cap suggests future yearly profit should be:
    - crudely, if this profit came from (in todays) terms/H100-equivalent, then:



Source: Ray Kurzweil, DFJ



Nvidia H100 GPU Shipments by Customer

Estimated 2023 H100 shipments by end customer.

Omdia estimates Nvidia sold ~500k A100 and H100 GPUs in Q3, and lead time for H100-based servers is up to 52 weeks.

Source: Omdia Research

Today

- Course Logistics
- A Word on Foundation Models
- ✓ Auto-Differentiation & computational graphs
  - checkpointing
- GPU/Infrastructure Background
- AD with Transformers

Auto-Differentiation

Automatic Differentiation

- The basic idea:
  - You write code to compute a scalar function $f : R^d \to R$.
  - AD computes $\nabla f(x)$ when you execute the code.
- This is the backbone of modern ML.
- Naively, one may expect that computing $\nabla f(x)$ to be more computationally expense than simply comping $f(x)$.

- “Theorem”: The Reverse Mode of AD computes $\nabla f(x)$ in time at most 5x that of the computing $f(x)$.
  (the computational model is “straight line” programs)

Straight Line Programs: An Example

- Suppose we are interested in computing the function:

$$f(w_1, w_2) = \big(\sin(2\pi w_1/w_2) + 3w_1/w_2 - \exp(2w_2)\big) \; * \; \big(3w_1/w_2 - \exp(2w_2)\big)$$

- Let us now consider a “straight line" program which computes our function f using “elementary” scalar functions at each step:

input: $z_0 = (w_1, w_2)$

$$z_1 = w_1/w_2$$

$$z_2 = \sin(2\pi z_1)$$

$$z_3 = \exp(2w_2)$$

$$z_4 = 3z_1 - z_3$$

$$z_5 = z_2 + z_4$$

$$z_6 = z_4 z_5$$

return: z_6

A Computational Graph (aka the “Evaluation Trace”)

- Compute $f(w_1, w_2)$:

  input: $z_0 = (w_1, w_2)$

  $z_1 = w_1/w_2$

  $z_2 = \sin(2\pi z_1)$

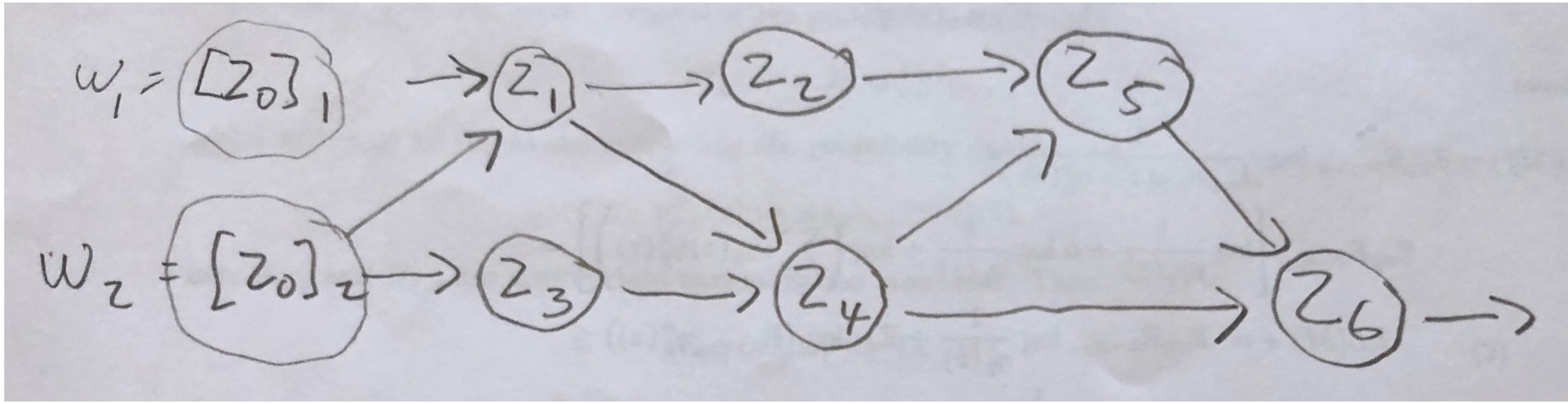
  $z_3 = \exp(2w_2)$

  $z_4 = 3z_1 - z_3$

  $z_5 = z_2 + z_4$

  $z_6 = z_4 z_5$

  return: z_6



- The computation graph is the flow of operations.
- We say that: z_2 and z_4 and children of z_1; z_5 is a child of z_2; etc.

Straight Line Programs

- Input: a vector $w \in R^d$
- All intermediate variables will be scalars (for clarity)
- Each step applies some differentiable real valued function $h \in \mathscr{H}$ to past variables, where each h is either
  - An affine functions.
  - A product of terms.
  - A fixed differentiable function, like $\cos(), \sin(), \exp(), \log()$, where we can compute $h'(x)$

Straight line program:

- **input:** $z_0 = w$.
  We actually have d (scalar) input nodes where$[z_0]_1 = w_1, [z_0]_2 = w_2, \ldots [z_0]_d = w_d$.
  1. $z_1 = h_1($a fixed subset of the variables in $w)$

  $\ldots$

  t. $z_t = h_t($a fixed a subset of the variables in $z_{1:t-1}, w)$

  $\ldots$

  T. $z_T = h_T($a fixed a subset of the variables in $z_{1:T-1}, w)$
- **return:** z_T

The Forward Mode of AD

We can compute $\frac{dz_T}{dz_0} = \frac{df}{dw}$ directly with the chain rule:

- **input:** $[z_0]_1 = w_1, [z_0]_2 = w_2, \ldots [z_0]_d = w_d$.
  1. $z_1 = h_1($a fixed subset of the variables in $w)$ & compute $\frac{dz_1}{dz_0}$

  $\ldots$

  t. $z_t = h_t($a fixed a subset of the variables in $z_{1:t-1}, w)$,

$$\frac{dz_t}{dz_0} = \sum_{p \text{ is a parent of } t} \frac{dz_t}{\partial z_p} \frac{dz_p}{\partial z_0}$$

  $\ldots$

- **return:** z_T and $\frac{dz_T}{dz_0}$

- How does the computational cost of this algo compare to just computing $f(w)$?

Can we do better? A different chain rule

```
input: $[z_0]_1 = w_1, [z_0]_2 = w_2, \ldots [z_0]_3 = w_d$.
 1. $z_1 = h_1$(a fixed subset of the variables in $w$)
  $\cdots$
 t.  $z_t = h_t$(a fixed a subset of the variables in $z_{1:t-1}, w$)
  $\cdots$
 T.  $z_T = h_T$(a fixed a subset of the variables in $z_{1:T-1}, w$)
return: $z_T$
```

- Let's think of $\frac{\partial z_T}{\partial z_t}$ as the derivative of z_T with respect to z_t, assuming that z_t is a "free" variable.
- By the chain rule:

$$\frac{\partial z_T}{\partial z_t} = \sum_{c \text{ is a child of } t} \frac{\partial z_T}{\partial z_c} \frac{\partial z_c}{\partial z_t}$$

The Reverse Mode of AD

Forward pass:

1.Compute $f(w)$ and store in memory all the intermediate variables $z_{0:T}$.

Backward pass:

2. Initialize:

$$\frac{dz_T}{dz_T} = 1$$

3.Proceeding recursively, starting at $t = T - 1$ and going to $t = 0$

$$\frac{\partial z_T}{\partial z_t} = \sum_{c \text{ is a child of } t} \frac{\partial z_T}{\partial z_c} \frac{\partial z_c}{\partial z_t}$$

4. **Return:**

$$\frac{dz_T}{dz_0} = \frac{df}{dw}$$

(which is the desired answer as $z_T = f, z_0 = w$)

Everything works if we allow z_t to be vectors or matrices.

Time Complexity

- History of AD: Linnainmaa (Lin76), Werbos(82), …

Theorem: [BaurStrassen 83] Suppose that $h \in \mathscr{H}$ are of the form:

- Affine functions.
- A product of terms.
- Fixed functions, like $\cos(), \sin(), \exp(), \log()$, where computing $h'(x)$ is no more than 5x the cost of computing $h(x)$

The Reverse Mode of AD computes $\nabla f(x)$ in time no more than a factor of 5 than the program used to compute $f(x)$.

Proof sketch (basically a book keeping argument):
in the forward pass, we associate the computation along edges from parents to a child. In the backward pass, note $\frac{\partial z_c}{\partial z_t}$ only is computed once.

$$\frac{\partial z_T}{\partial z_t} = \sum_{c \text{ is a child of } t} \frac{\partial z_T}{\partial z_c} \frac{\partial z_c}{\partial z_t}$$

Auto-Differentiation: Checkpointing and Memory

Neural Net Example

```
Compute $Loss()$:
input: parameters $W_1, W_2, \dots W_L \in R^{d \times d}, w \in R^d$, &
        batch data: $(X, Y), X \in R^{d \times m}, Y \in R^m$

For $\ell = 0, \dots L - 1$
        $X \leftarrow \sigma(W_{\ell+1} X)$
Compute loss: $L = \frac{1}{m} \|Y - X^\top w\|_2^2$

return: the loss $L$
```

- Parameter/input memory:

- What free memory is sufficient to execute this program?

- How much memory would we need if ran reverse mode AD?

The Reverse Mode of AD, with Checkpointing

Assume z_{t+1} is only a function of the variables z_t (here let the intermediate variables be vectors)

Checkpoint indexes: $C = \{\tau_1 \leq \tau_2 \ldots \leq \tau_k\}$, i.e. $C \subset \{1, \ldots T\}$.

Forward pass:

1.Compute $f(w)$ and store only the variables $\{z_\tau : \tau \in C\}$.

Backward pass:

2. Initialize: $\frac{dz_T}{dz_T} = 1$, set $\tau_{k+1} = T$

3.Proceeding recursively, for $i = k, \ldots 1$

- **Rematerialization:**
  Redo forward pass, computing/storing the graph in “block” k, from $t = \tau_i$ to $t = \tau_{i+1}$
- Backward pass in “block” k: Starting at $t = \tau_{i+1}$ and going to $t = \tau_i$
  $\frac{\partial z_T}{\partial z_t} = \sum_{c \text{ is a child of } t} \frac{\partial z_T}{\partial z_c} \frac{\partial z_c}{\partial z_t}$

4. **Return:** $\frac{dz_T}{dz_0}$

Memory required: store $\{z_\tau : \tau \in C\}$; store all variables in a “block” rematerialization pass

Compute overhead: need to recompute all the “blocks”, which is at most the cost to compute $f(x)$.

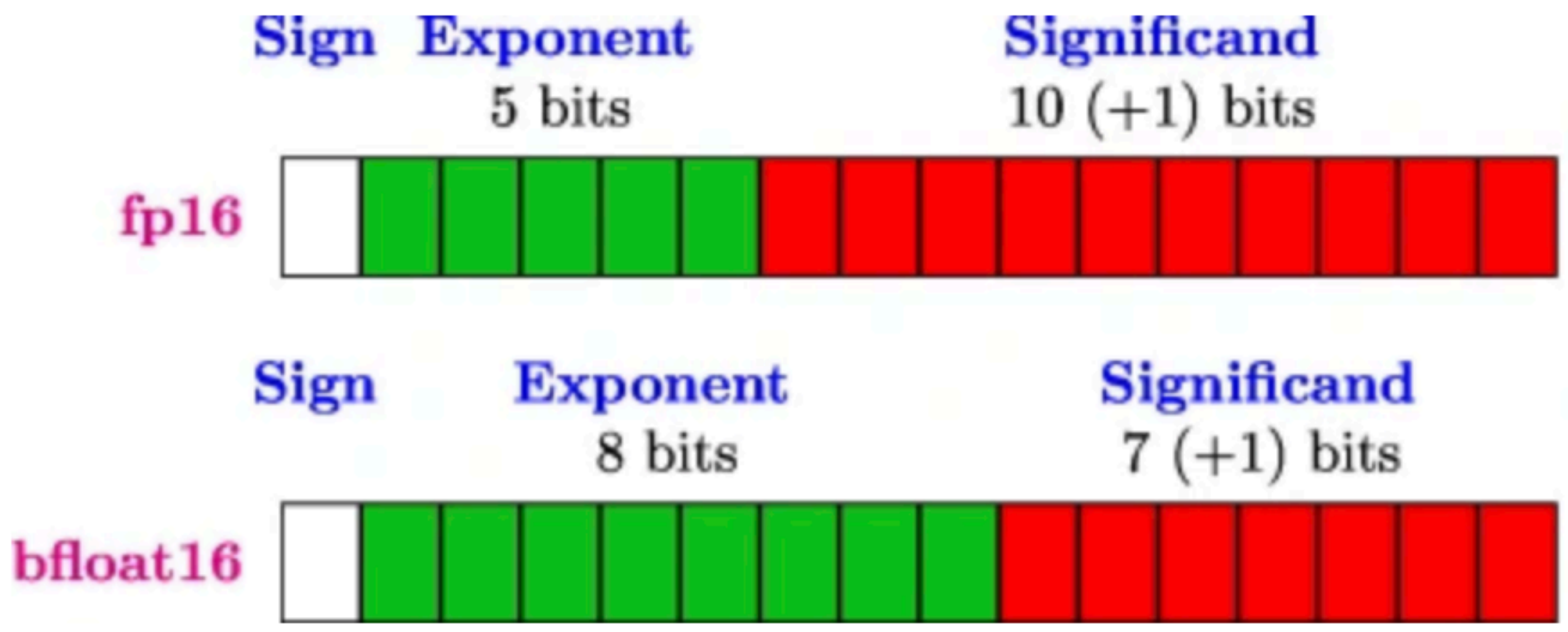
Let’s return to AD for some “big” models

- Llama3.1: 400B
  GPT4: ≈2T

- Bfloat16: 2 bytes/parameter
  - Specialized precision type for neural nets

- Memory required to store these models:
  Llama3.1: 0.8 Tbytes
  GPT4: 4 Tbytes

- H100s have 80GB memory each:
  Llama3.1:
  GPT4:



Today

- Course Logistics
- A Word on Foundation Models
- Auto-Differentiation & computational graphs
  - checkpointing
- ✓ GPU/Infrastructure Background
- AD with Transformers

GPU Background

GPU Background

- The goal of this course isn't to deep dive into hardware (though it is an extremely interesting topic)

- Goal is to understand roughly how GPUs work and **what the relevant scales are**, which lets us quickly estimate useful quantities that govern training efficiency

- There is an intricate tension between **compute and memory (I/O)**, and many useful insights have come about from understanding this tension deeply (FlashAttention 1/2/3, kernel fusions, etc.)

- We'll take a bottom-up perspective

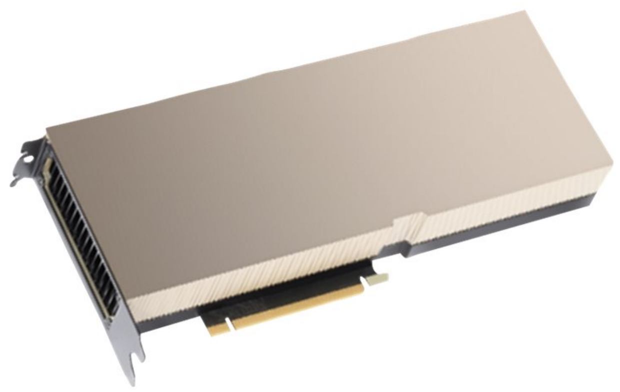
Why are GPUs useful?

- Modern ML stacks are complicated, but at the end of the day the primary operation we're doing is simple: **matrix multiplication (matmul)**
- GPUs are just blocks of transistors organized in a way that makes them really great for parallel matmuls (SIMD = single instruction, multidata)
- Exact details are complicated; our goal is to understand how computation and memory works w.r.t model training

Some numbers

A100 SXM

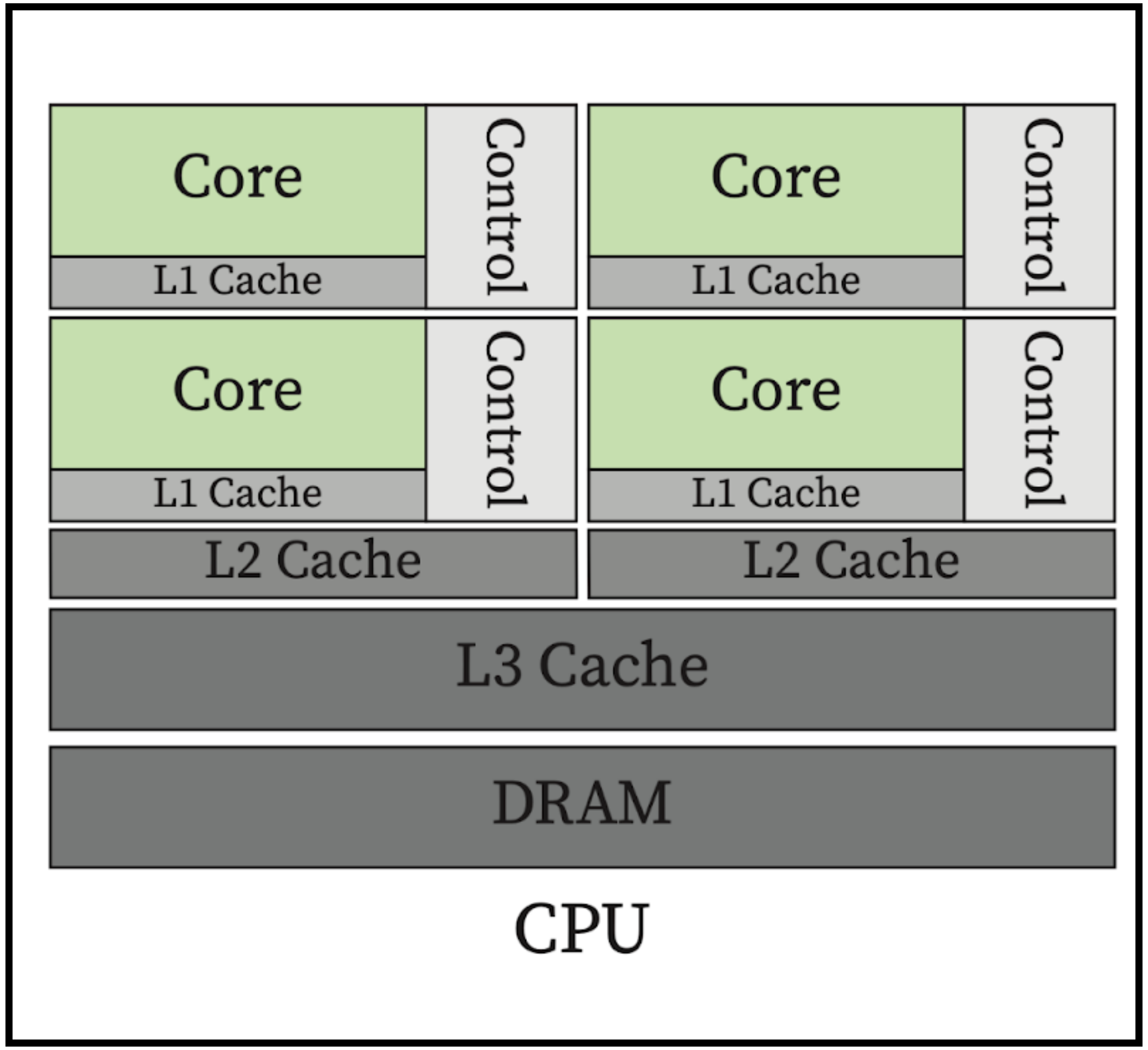
- 624 TFLOPS at fp16 with 128 SMs
- **80 GB memory (DRAM)**
- ~2 TB/s memory bandwidth
- Unit cost: $18-30,000



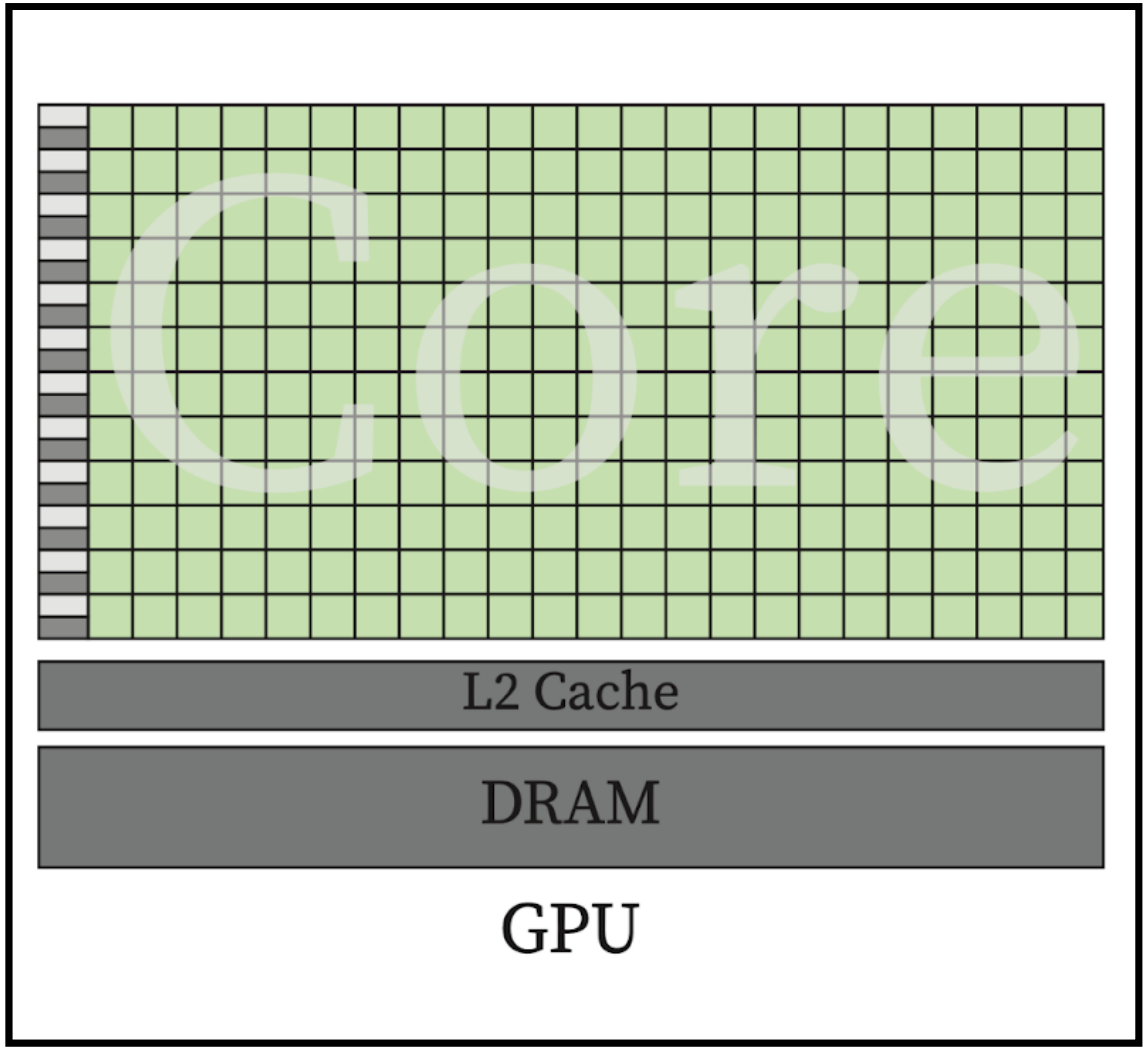
H100 SXM

- 1979 TFLOPS at fp16 with 132 SMs
- **80 GB memory (DRAM)**
- ~3.4 TB/s memory bandwidth
- Unit cost: $25-40,000

GPUs vs CPUs



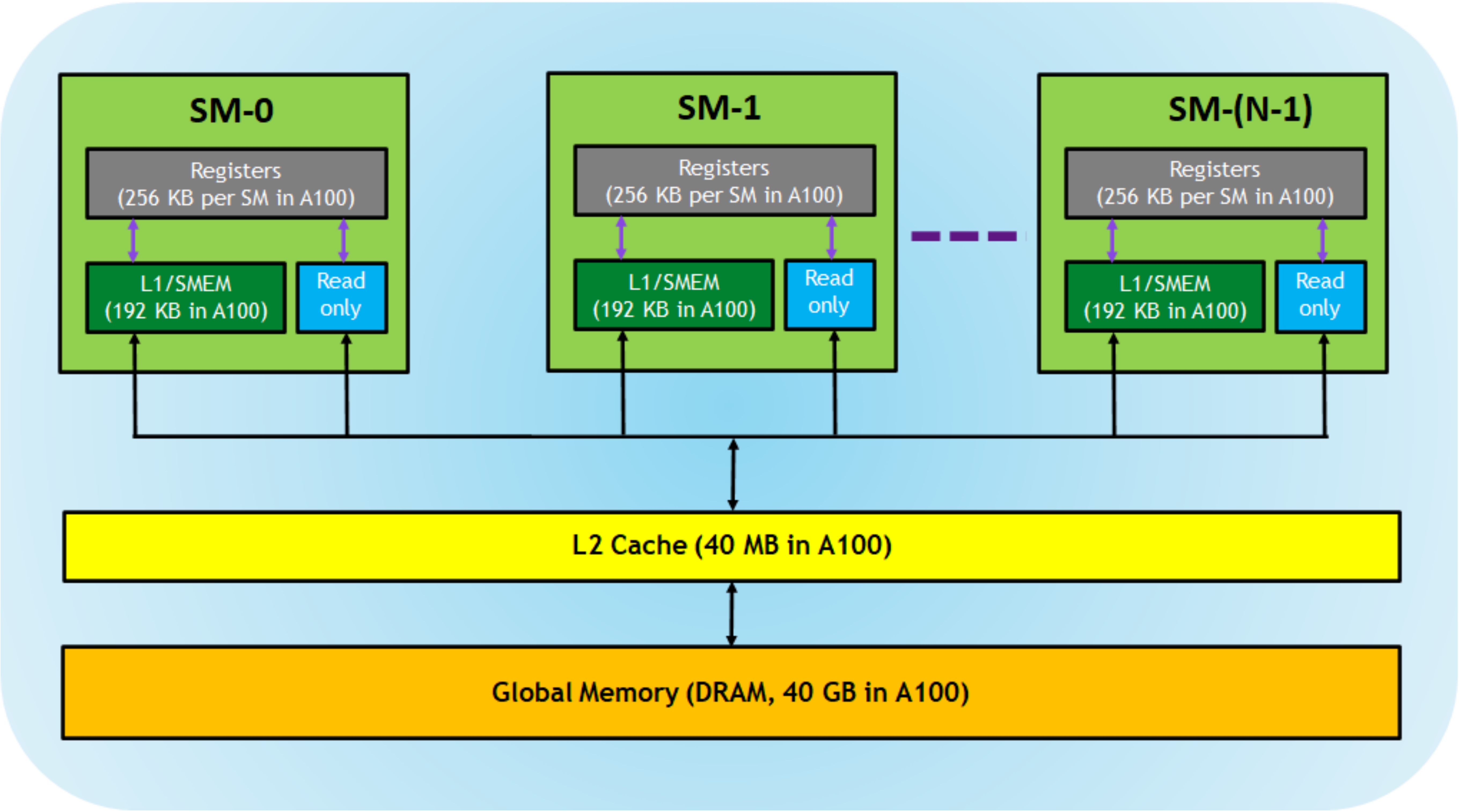
CPU

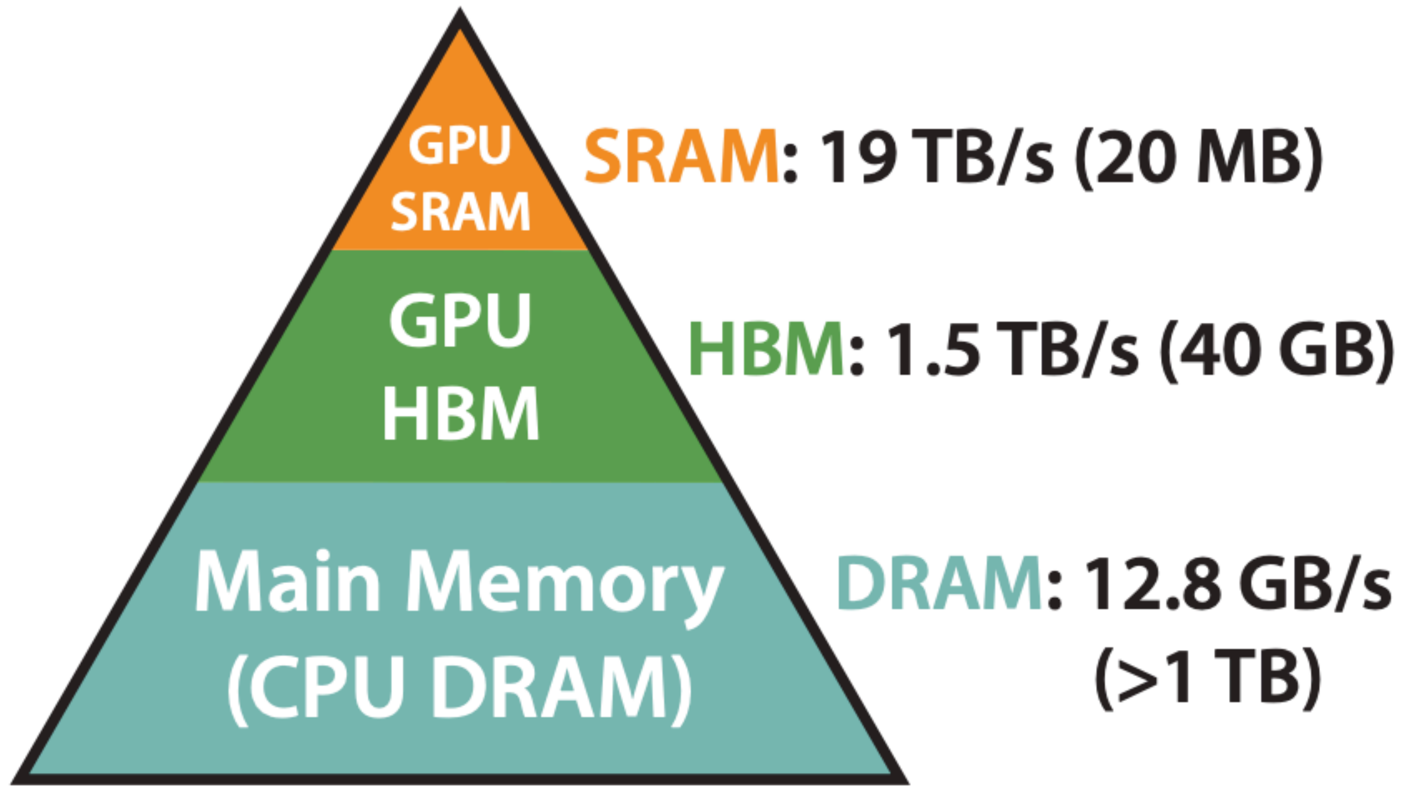


GPU

[Figure credit: Yasin Mazloumi]

Memory hierarchy

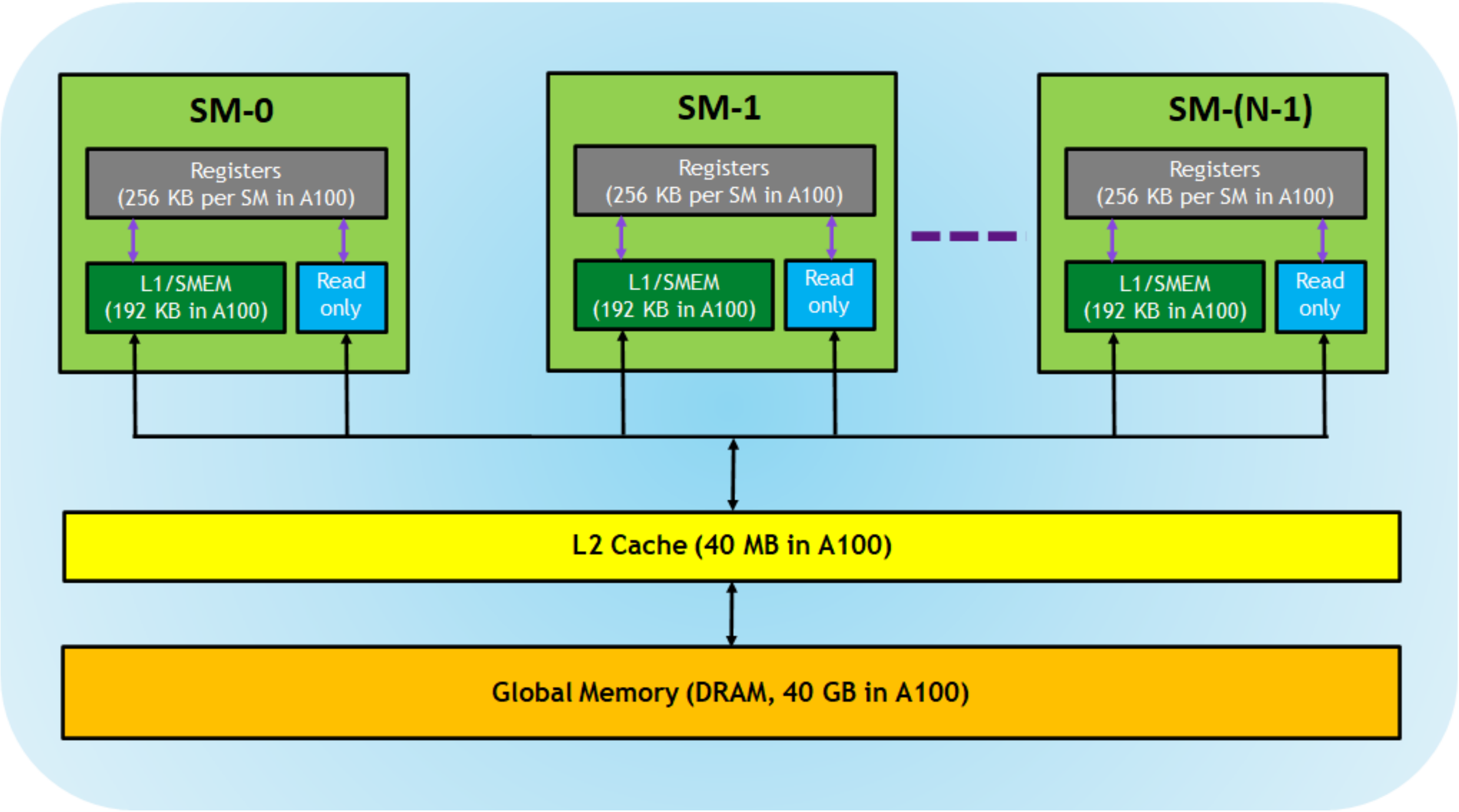




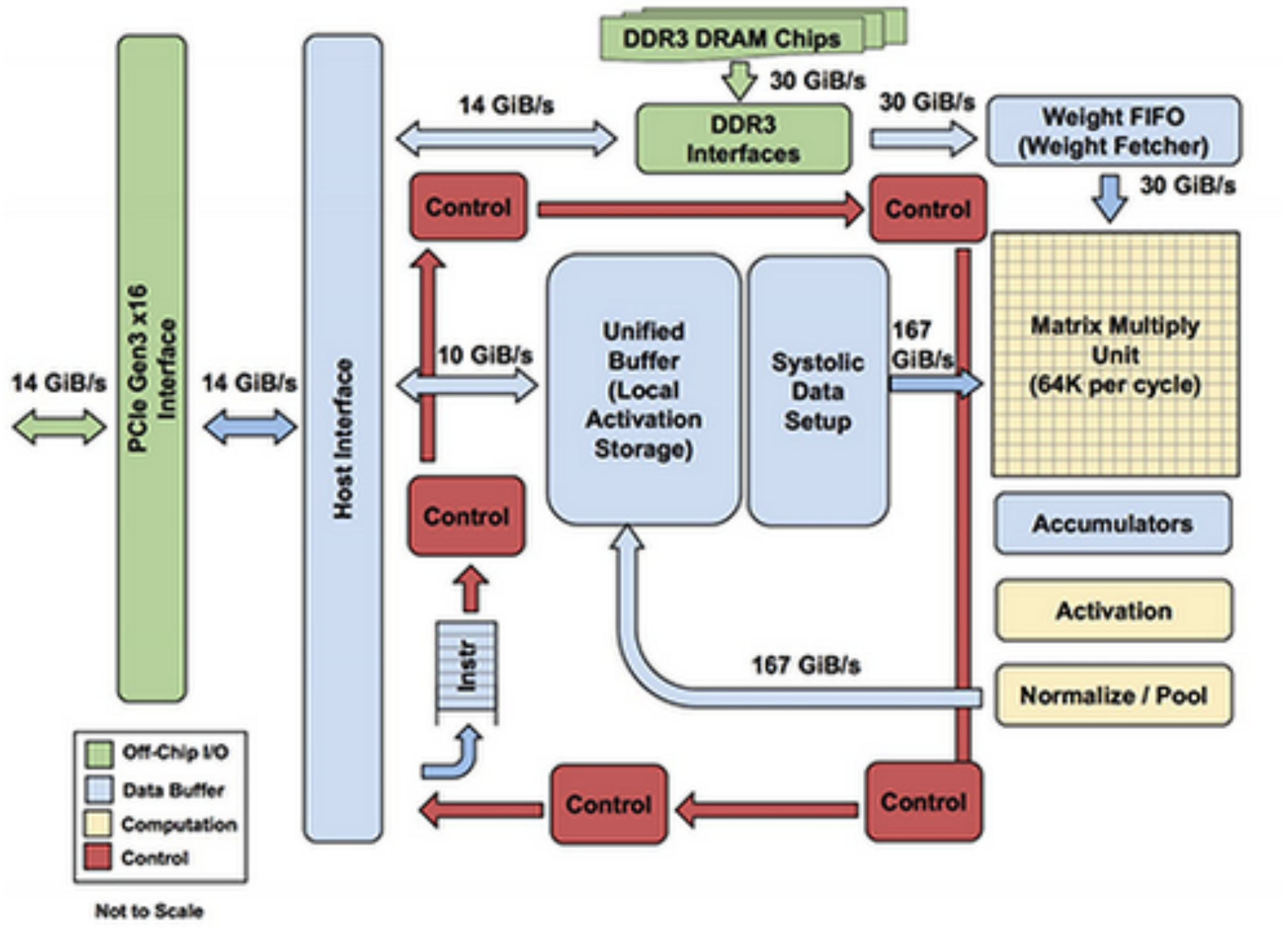
Memory Hierarchy with Bandwidth & Memory Size

[Figure credit: Dao et al. 2022]

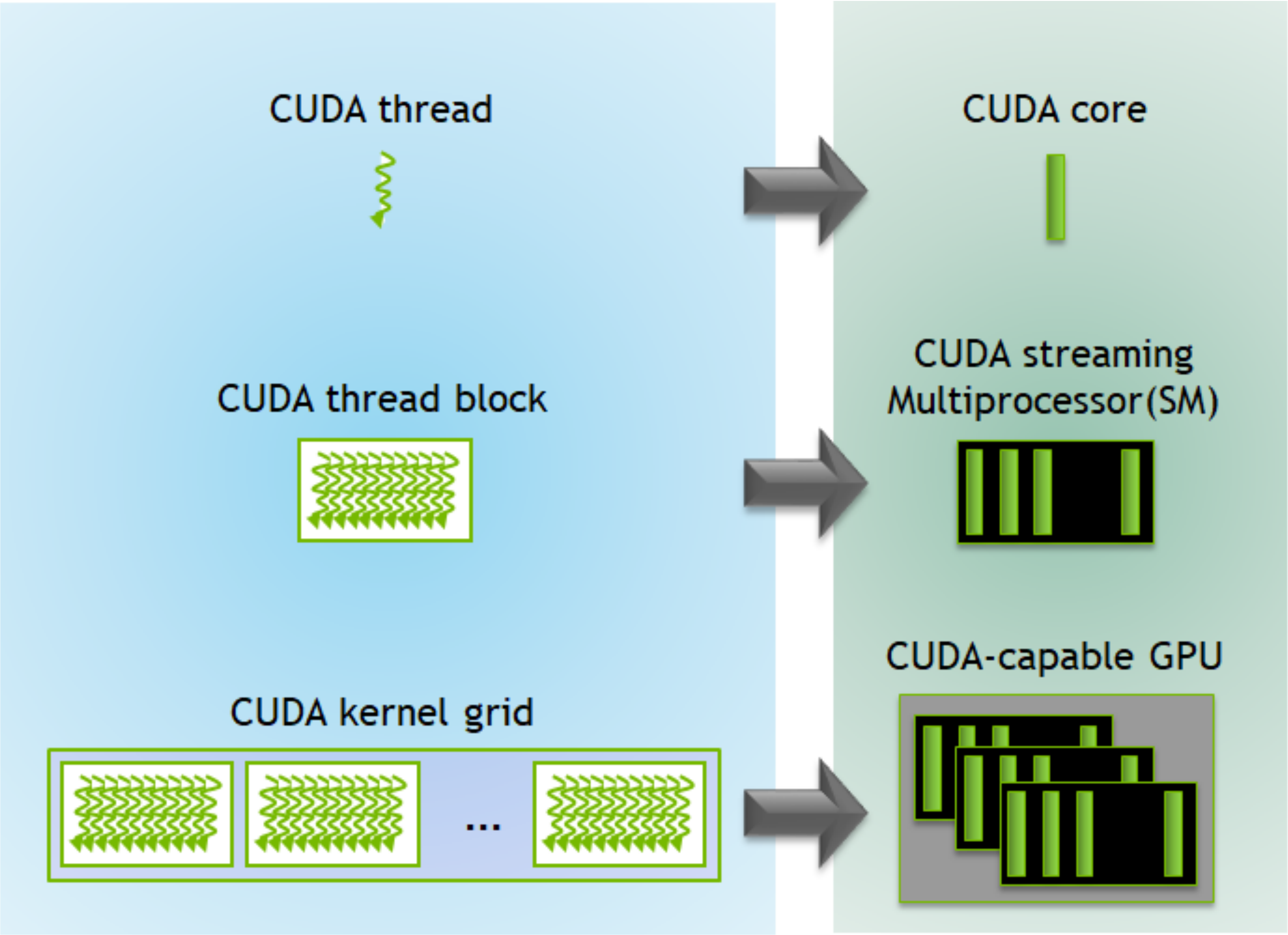
Memory hierarchy



Parenthetical comment: TPUs

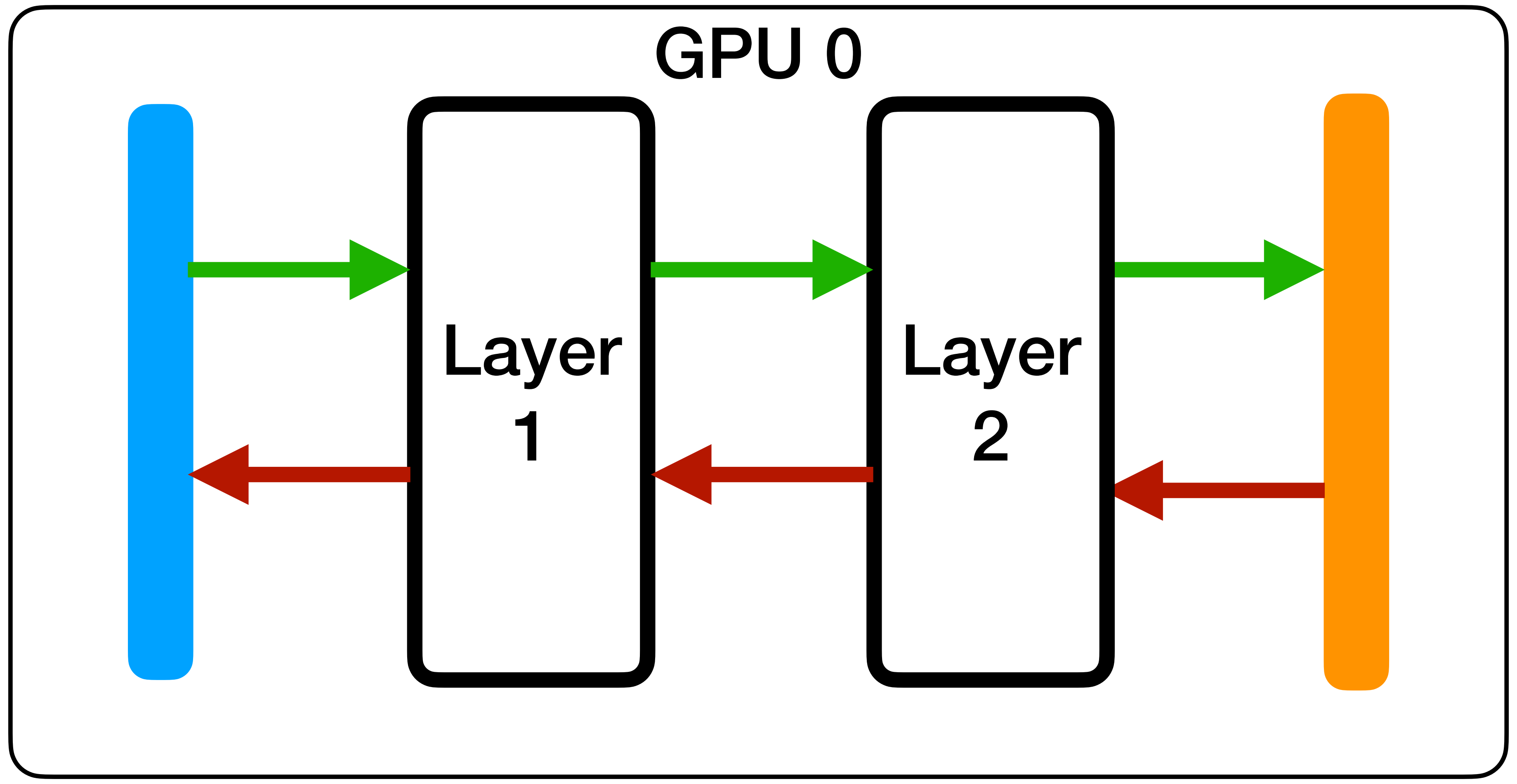


Computation organization

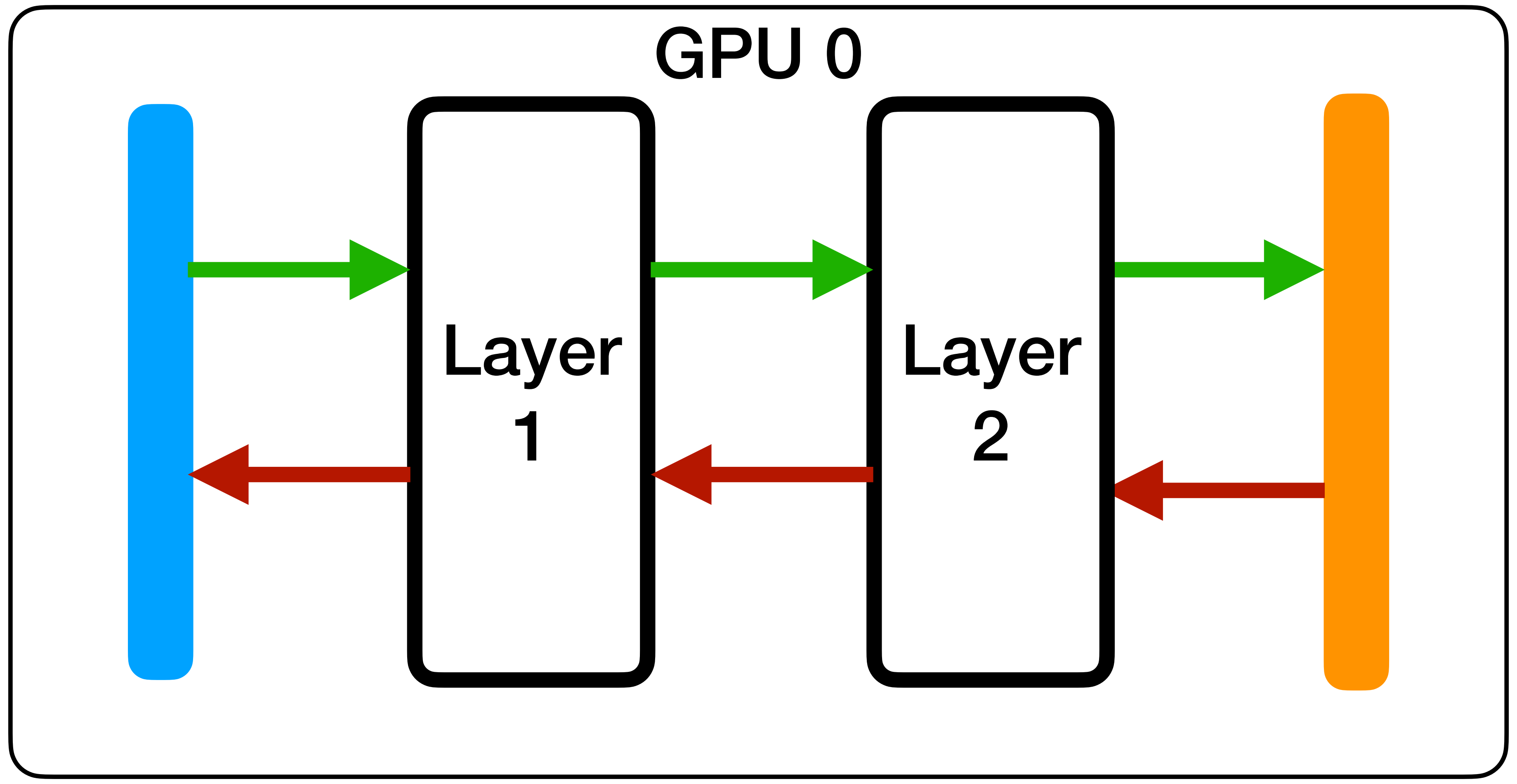


- **Thread**: unit of parallel execution
- **Block**: 1024 threads (sometimes warp is used = 32 threads)
- **Kernel**: function that’s running on GPU

Single GPU training



Single GPU training

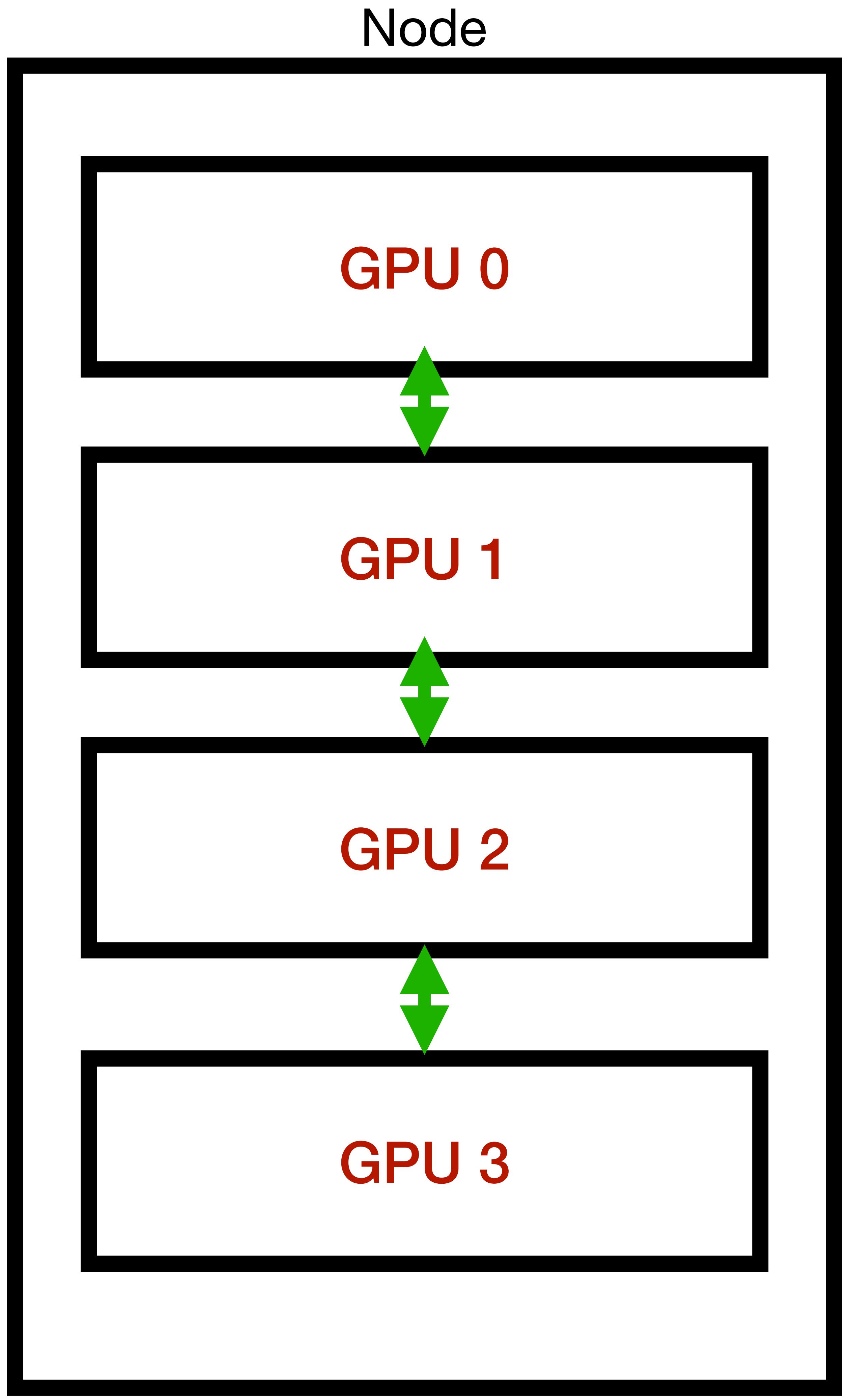


$$M_{\text{train}} = P_{\text{transformer}} + M_{\text{optimizer}} + M_{\text{activations}}$$

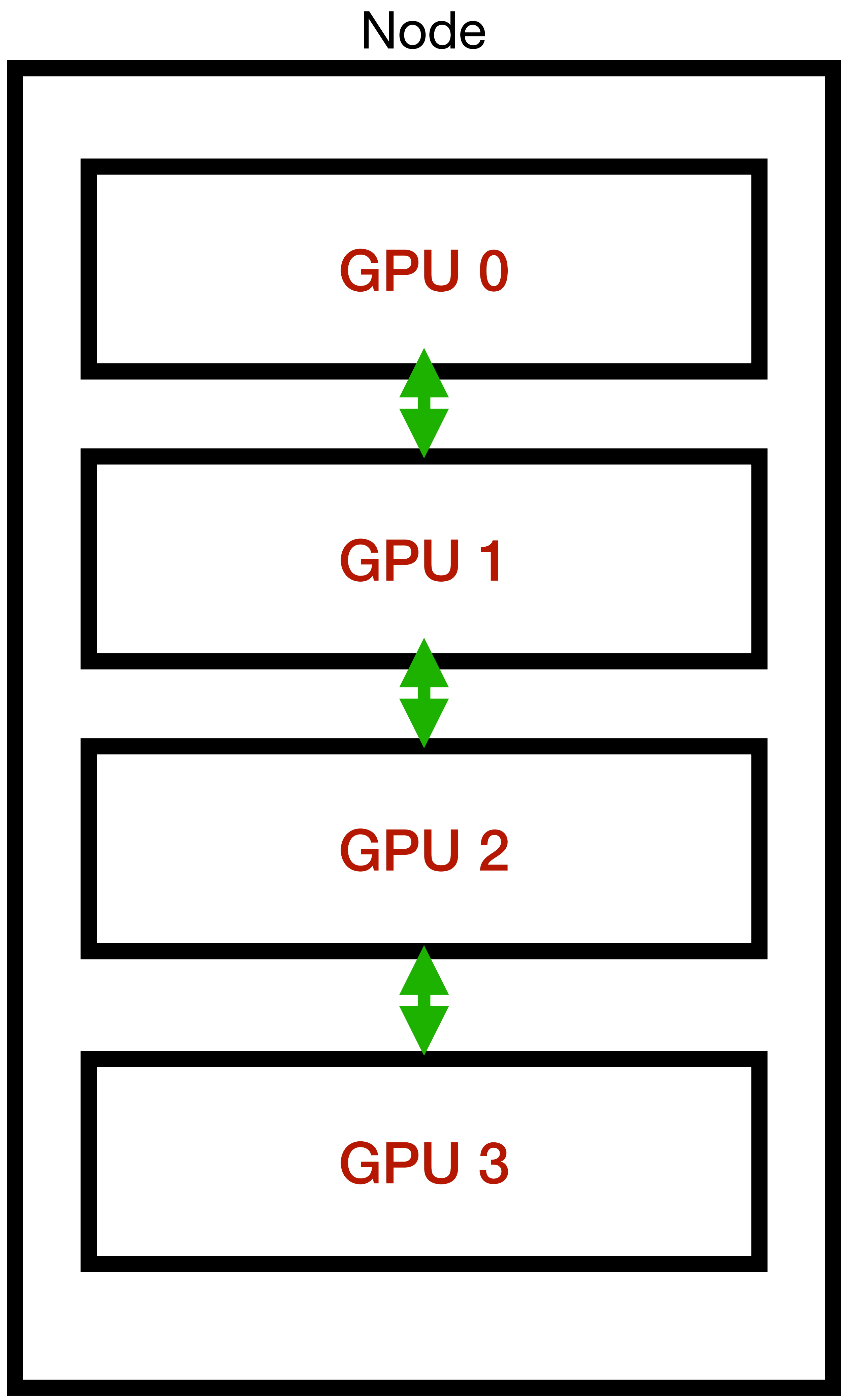
Resource	**Units**	**Multiplicative Factor**
Compute	FLOPs	2
Memory (Parameters)	Bytes	2 (bfloat16) or 4 (float32)
Memory (Optimizer)	Bytes	4 (float32)

For A100 40 GB, this is roughly a 2B model with batch size 16 with no optimizations

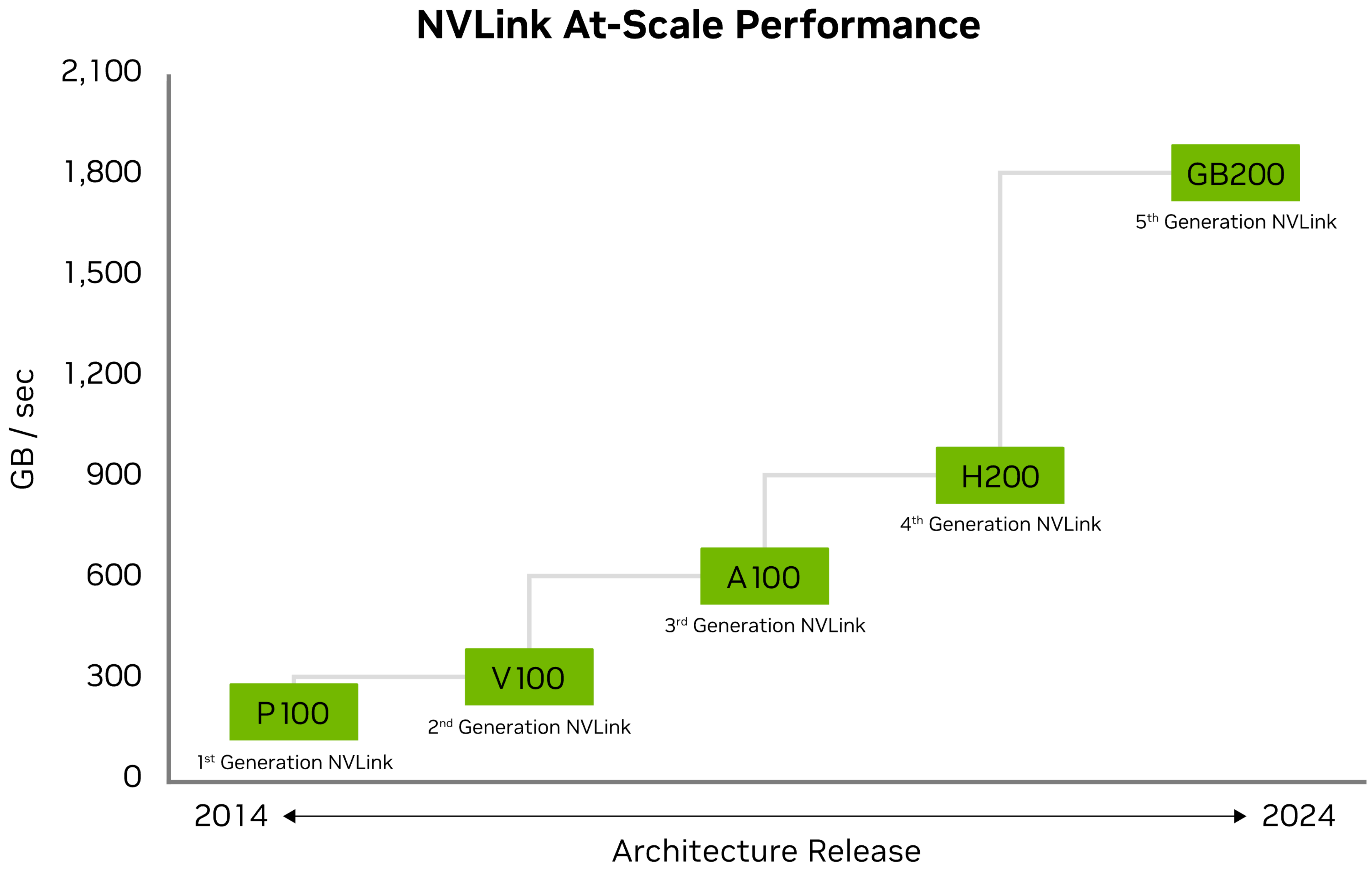
What do large models run on?



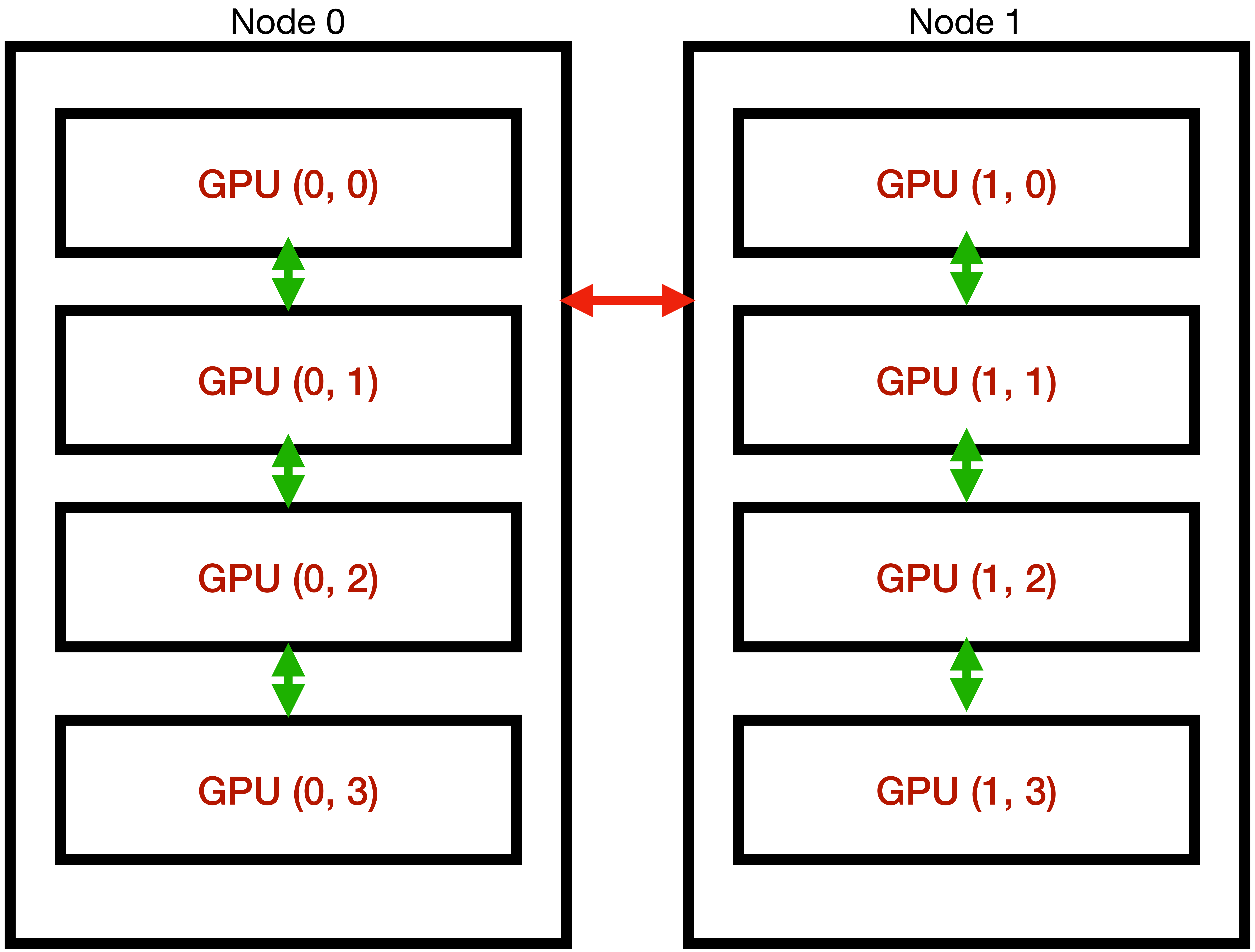
What do large models run on?



Communication *between* GPUs in a node is typically fast, approaching **O(TB/s)**



What do large models run on?

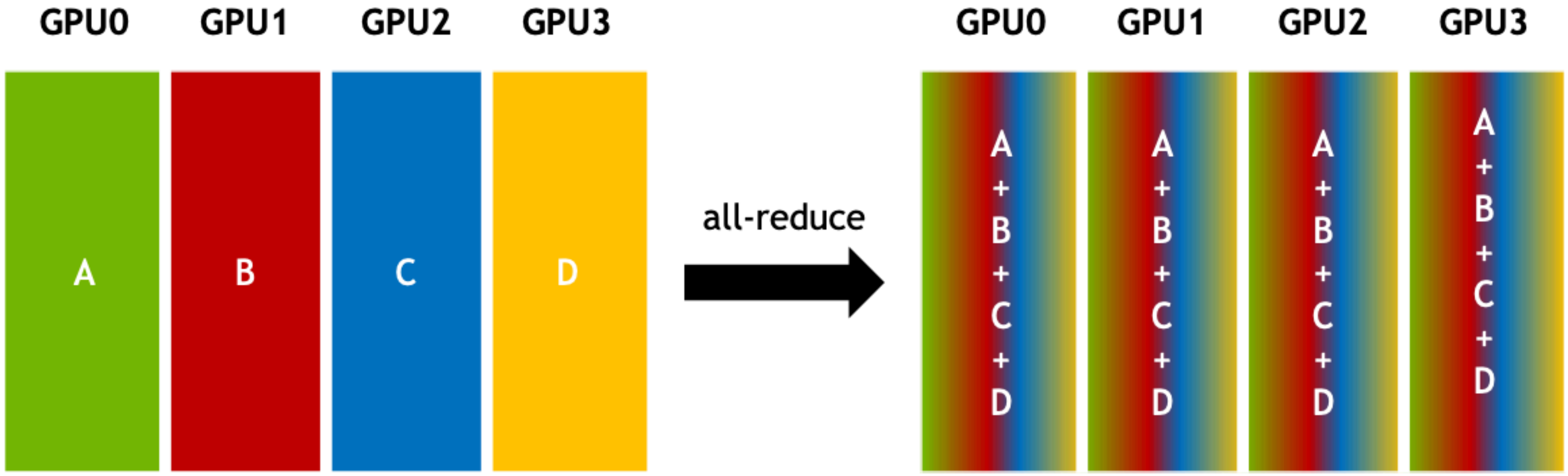


Communication *between* nodes is typically much slower, **O(few * 10 GB/s)**

“InfiniBand”

NCCL communication primitives

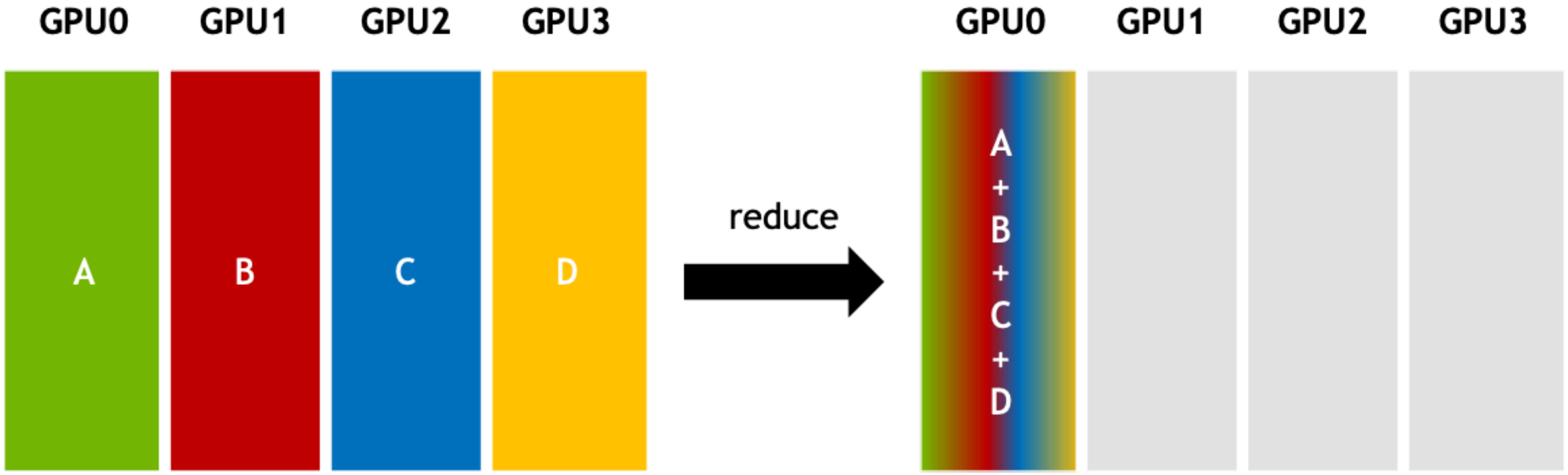
AllReduce



There are many possible implementations! E.g., compute in a ring

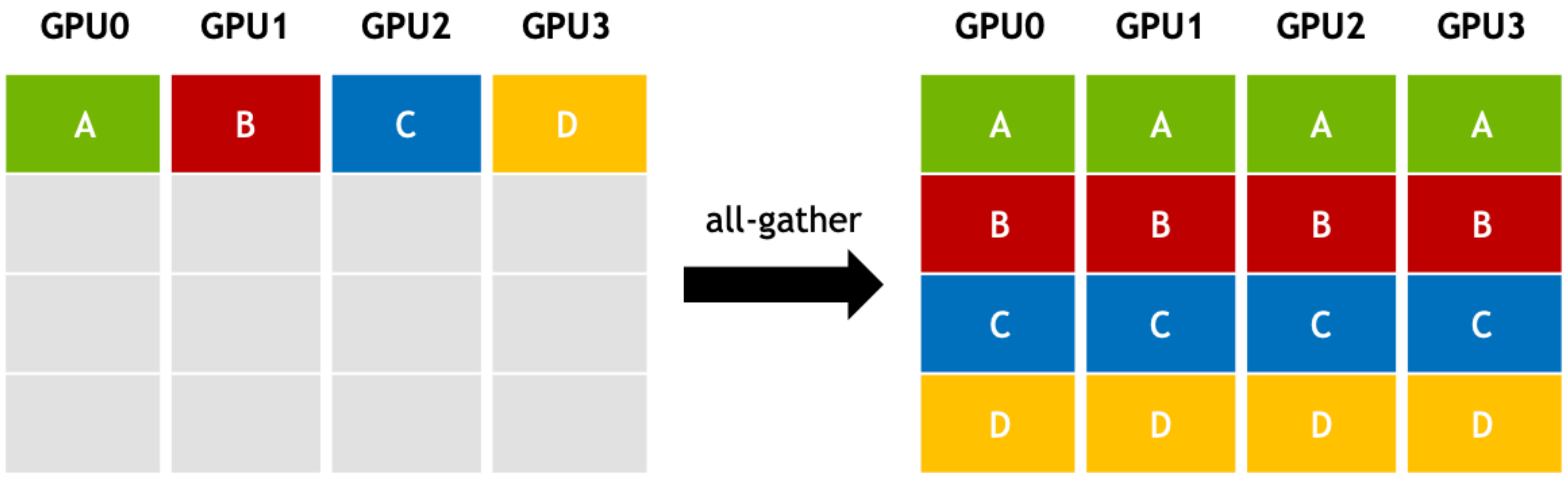
NCCL communication primitives

Reduce



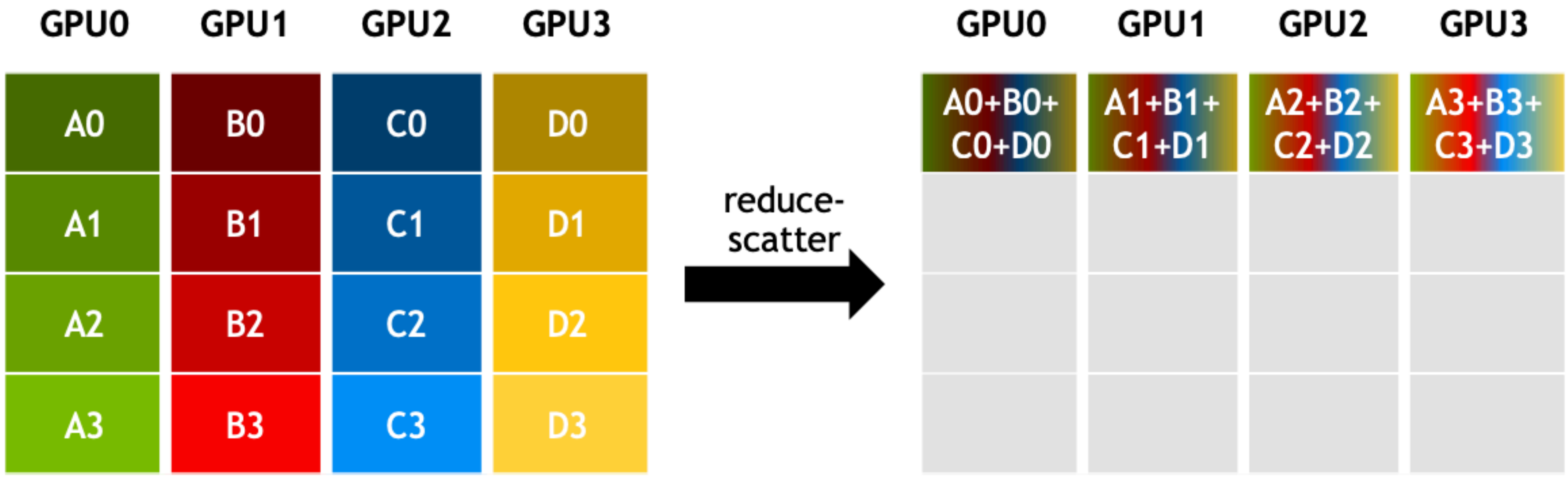
NCCL communication primitives

AllGather

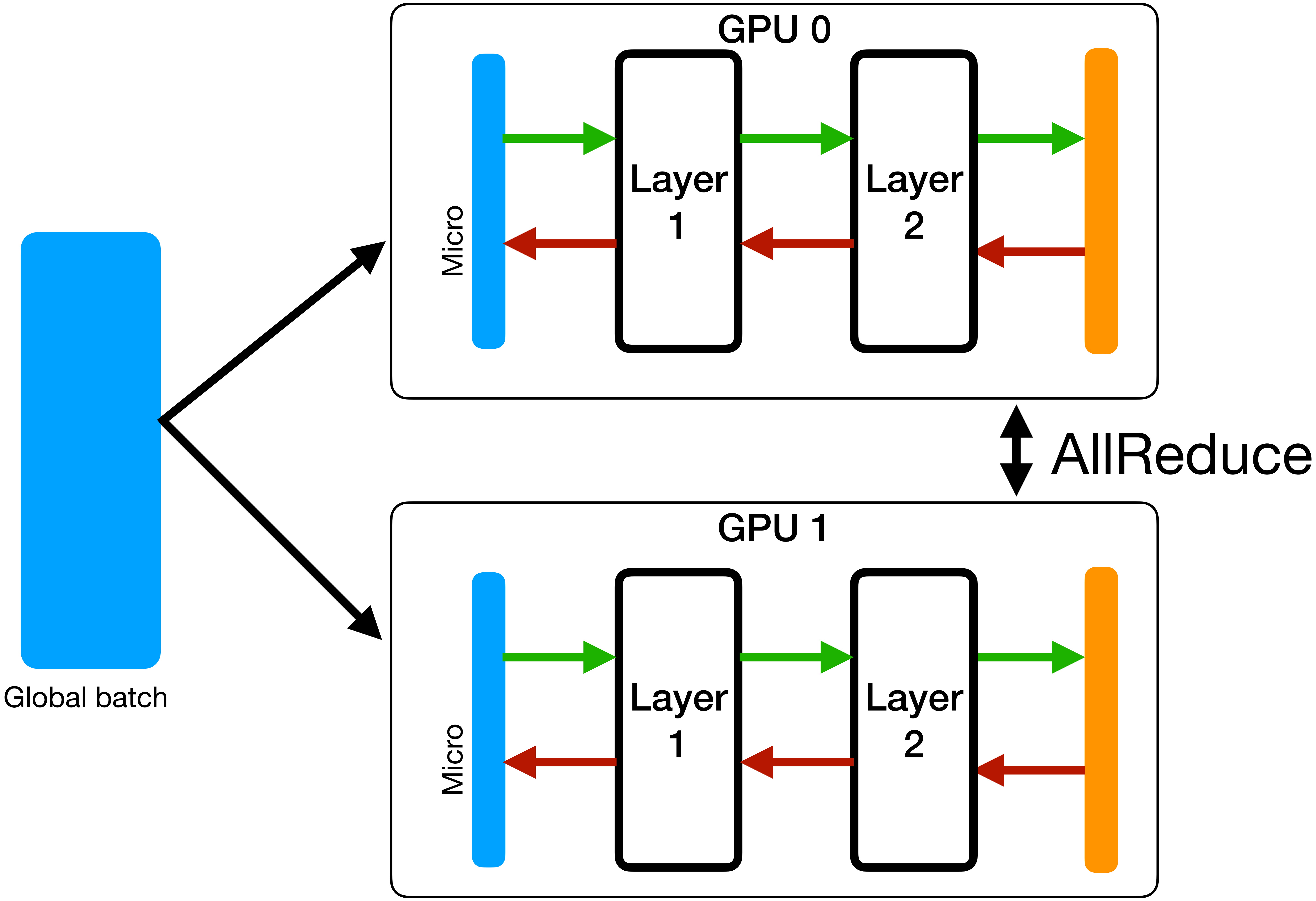


NCCL communication primitives

ReduceScatter



Multi GPU Training: DDP



When is DDP useful?

- When a model fits on a single GPU, and we want to increase data throughput i.e. train faster
- When it makes sense to keep inter-GPU communication as simple as possible (e.g., smaller scale experiments)
- Models that are large enough that cannot be fit on a single GPU are trained with other distributed frameworks (FSDP, etc.)

Bulk metrics for GPU performance

$$\text{MFU} = \frac{\text{actual FLOPs}}{\text{theoretical FLOPs}}.$$

$$\text{arithmetic intensity} = \frac{\text{total arithmetic operations}}{\text{bytes accessed}}$$

Bulk metrics for GPU performance

$$\text{MFU} = \frac{\text{actual FLOPs}}{\text{theoretical FLOPs}}$$

$$\text{arithmetic intensity} = \frac{\text{total arithmetic operations}}{\text{bytes accessed}}$$

A100 theoretical max: 312 TFLOPs

H100: 1979 TFLOPs, 1671 TFLOPs for SXM

Bulk metrics for GPU performance

$$\text{MFU} = \frac{\text{actual FLOPs}}{\text{theoretical FLOPs}}$$

Llama 3 largest model:
16k H100s, 405B param
on 16.5 T tokens over 54
days

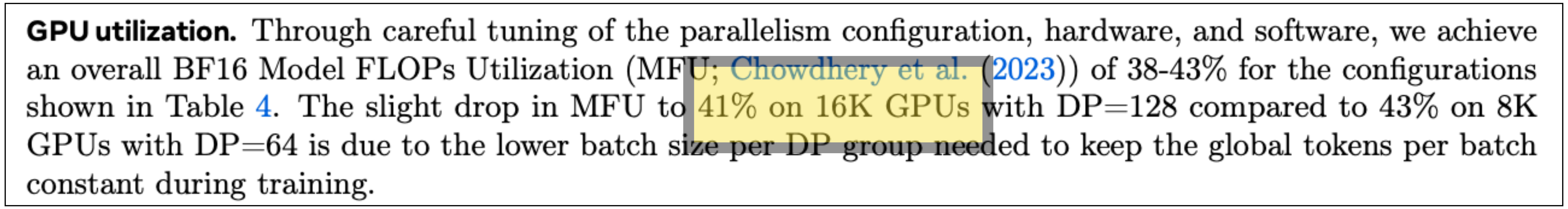
$$\text{arithmetic intensity} = \frac{\text{total arithmetic operations}}{\text{bytes accessed}}$$

$$\text{MFU} = \frac{6 \cdot 405 \times 10^{9} \times 16.5 \times 10^{12}}{16 \times 10^{3} \times 1671 \times 10^{12} \times 54 \times 24 \times 60 \times 60} \times 100 = 32\%$$

Bulk metrics for GPU performance

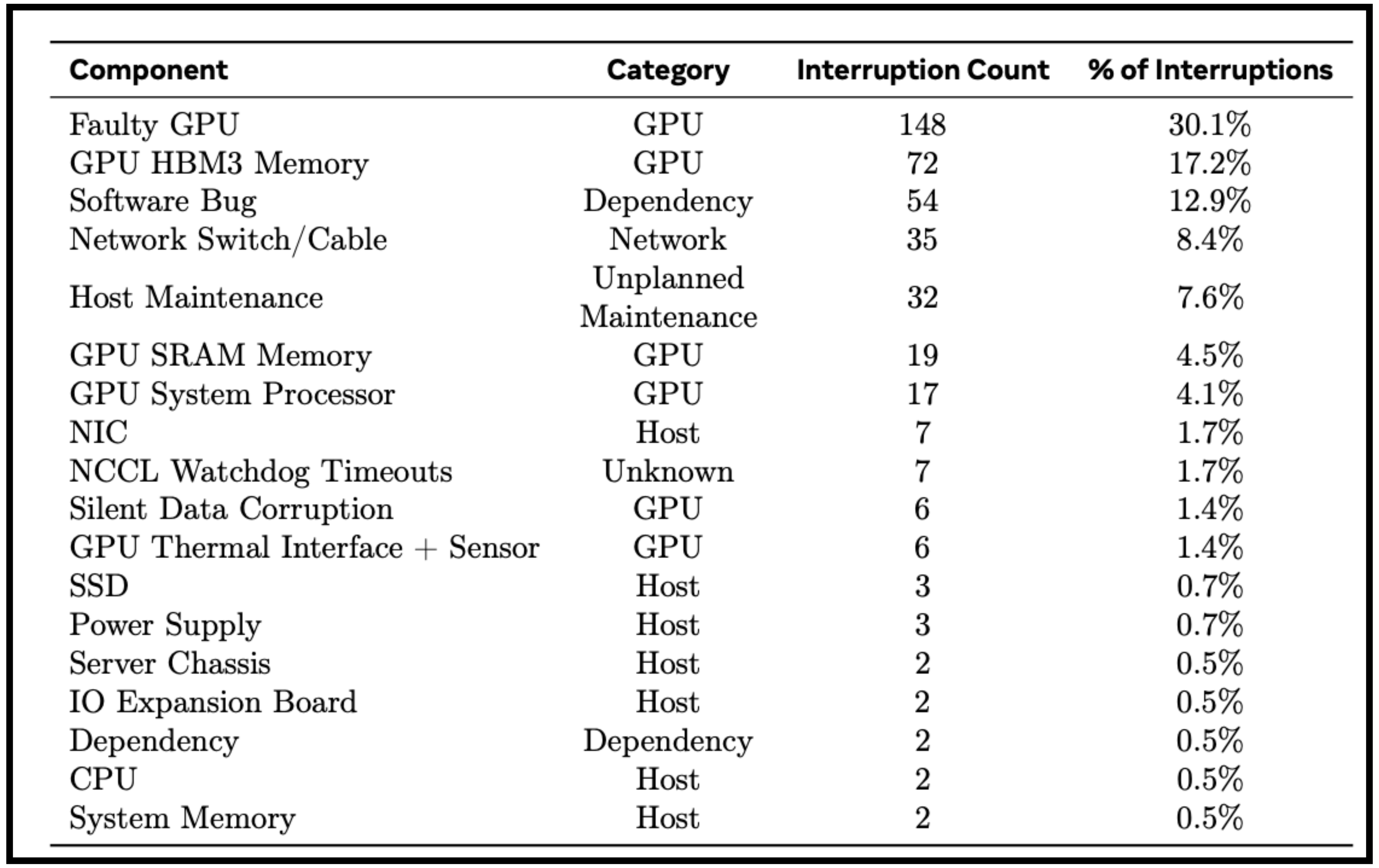
$$\text{MFU} = \frac{\text{actual FLOPs}}{\text{theoretical FLOPs}}.$$

Llama 3 largest model: 16k H100s, 405B param on 16.5 T tokens over 54 days



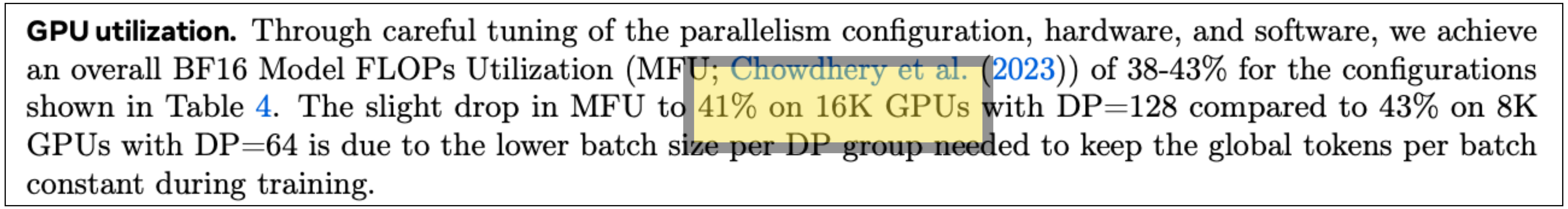
GPU utilization. Through careful tuning of the parallelism configuration, hardware, and software, we achieve an overall BF16 Model FLOPs Utilization (MFU; Chowdhery et al. (2023)) of 38-43% for the configurations shown in Table 4. The slight drop in MFU to 41% on 16K GPUs with DP=128 compared to 43% on 8K GPUs with DP=64 is due to the lower batch size per DP group needed to keep the global tokens per batch constant during training.

$$\text{MFU} = \frac{6 \cdot 405 \times 10^{9} \times 16.5 \times 10^{12}}{16 \times 10^{3} \times 1671 \times 10^{12} \times 54 \times 24 \times 60 \times 60} \times 100 = 32\%.$$



Component	Category	Interruption Count	% of Interruptions
Faulty GPU	GPU	148	30.1%
GPU HBM3 Memory	GPU	72	17.2%
Software Bug	Dependency	54	12.9%
Network Switch/Cable	Network	35	8.4%
Host Maintenance	Unplanned Maintenance	32	7.6%
GPU SRAM Memory	GPU	19	4.5%
GPU System Processor	GPU	17	4.1%
NIC	Host	7	1.7%
NCCL Watchdog Timeouts	Unknown	7	1.7%
Silent Data Corruption	GPU	6	1.4%
GPU Thermal Interface + Sensor	GPU	6	1.4%
SSD	Host	3	0.7%
Power Supply	Host	3	0.7%
Server Chassis	Host	2	0.5%
IO Expansion Board	Host	2	0.5%
Dependency	Dependency	2	0.5%
CPU	Host	2	0.5%
System Memory	Host	2	0.5%

Llama 3 largest model: 16k H100s, 405B param on 16.5 T tokens over 54 days



GPU utilization. Through careful tuning of the parallelism configuration, hardware, and software, we achieve an overall BF16 Model FLOPs Utilization (MFU; Chowdhery et al. (2023)) of 38-43% for the configurations shown in Table 4. The slight drop in MFU to 41% on 16K GPUs with DP=128 compared to 43% on 8K GPUs with DP=64 is due to the lower batch size per DP group needed to keep the global tokens per batch constant during training.

$$\text{MFU} = \frac{6 \cdot 405 \times 10^{9} \times 16.5 \times 10^{12}}{16 \times 10^{3} \times 1671 \times 10^{12} \times 54 \times 24 \times 60 \times 60} \times 100 = 32\%$$

Bulk metrics for GPU performance

$$\text{MFU} = \frac{\text{actual FLOPs}}{\text{theoretical FLOPs}}$$

$$\text{arithmetic intensity} = \frac{\text{total arithmetic operations}}{\text{bytes accessed}}$$

Llama 3 largest model: 16k H100s, 405B param on 16.5 T tokens over 54 days

H100: 3.35 TB/s promised

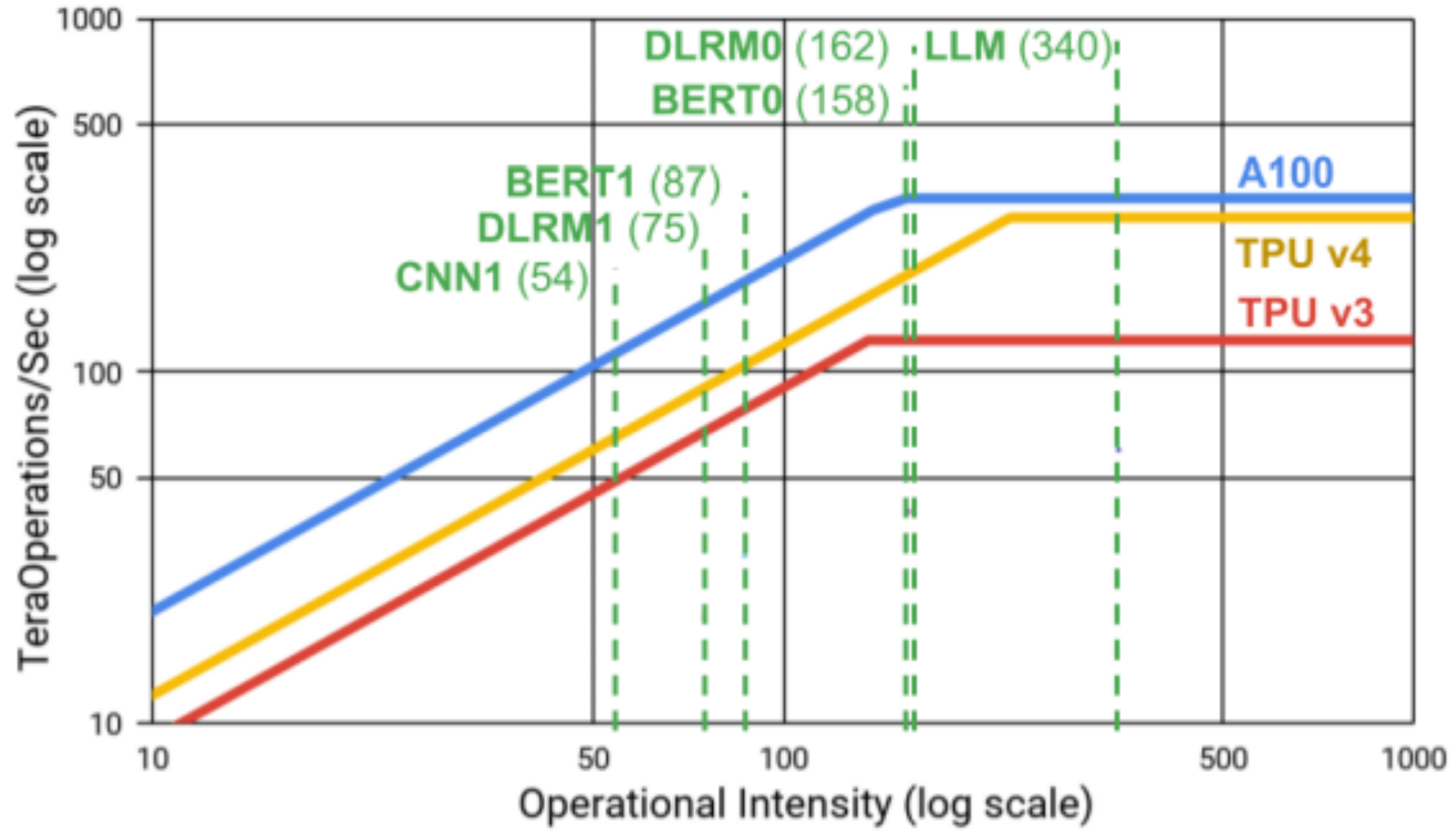
$$\text{arith. inten} = \frac{5.4 \times 10^{14} \text{ FLOPs s}^{-1}\text{GPU}^{-1}}{3.35 \times 10^{12} \text{ TB s}^{-1}} = 160 \text{ FLOPs byte}^{-1}$$

Bulk metrics for GPU performance

$$\text{MFU} = \frac{\text{actual FLOPs}}{\text{theoretical FLOPs}}$$

$$\text{arithmetic intensity} = \frac{\text{total arithmetic operations}}{\text{bytes accessed}}$$

Source: arXiv:2304.01433



$$\text{arith. inten} = \frac{5.4 \times 10^{14}\ \text{FLOPs s}^{-1}\text{GPU}^{-1}}{3.35 \times 10^{12}\ \text{TB s}^{-1}} = 160\ \text{FLOPs byte}^{-1}$$

Today

- Course Logistics
- A Word on Foundation Models
- Auto-Differentiation & computational graphs
  - checkpointing
- ✓ GPU/Infrastructure Background
- AD with Transformers

Transformers:

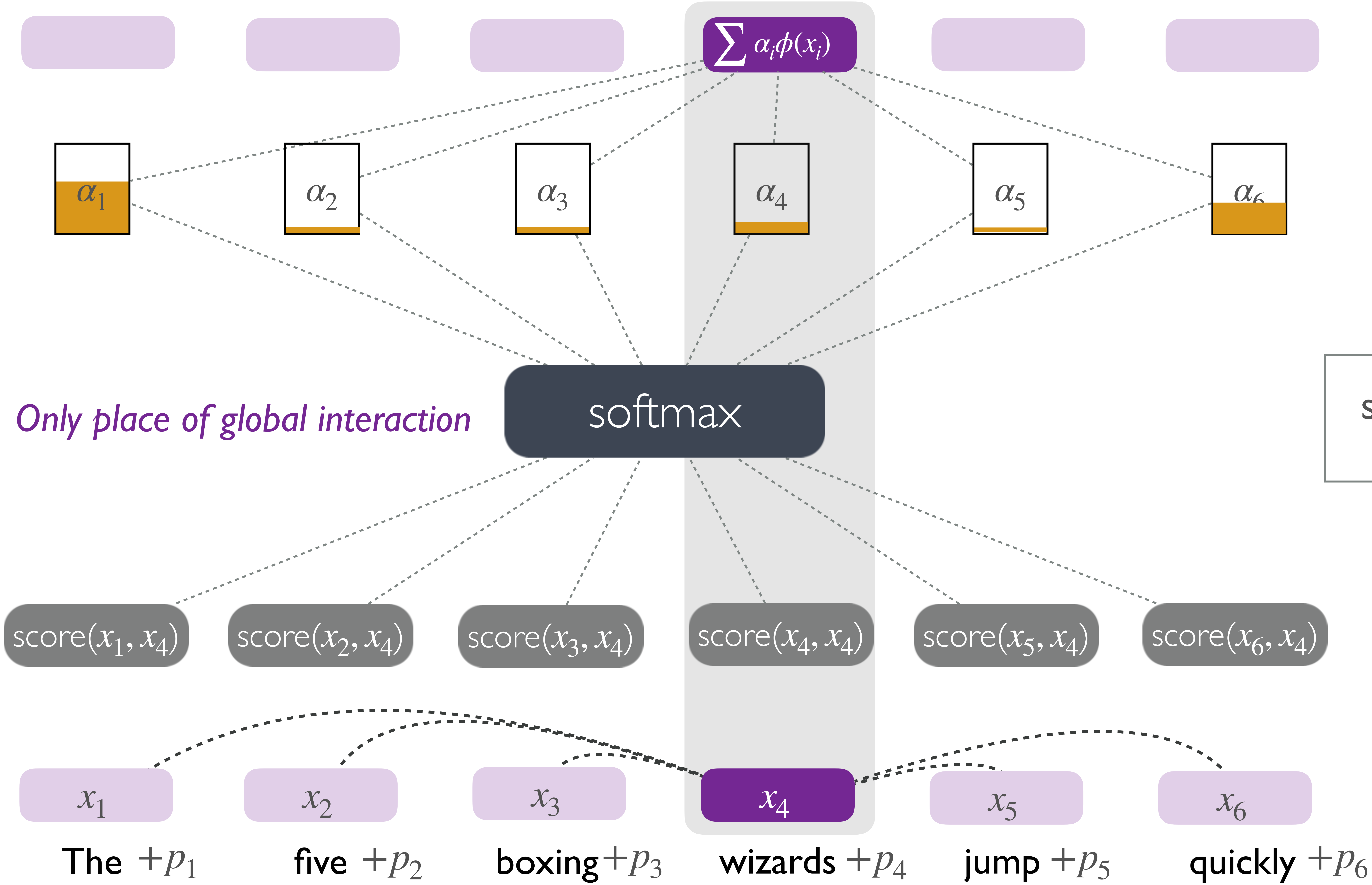
Brief Model Overview

TRANSFORMERS RECAP - SELF-ATTENTION

$x_i \in \mathbb{R}^d$

$W_V \in \mathbb{R}^{d\times d}$

$W_Q, W_K \in \mathbb{R}^{d\times d}$



$\sum \alpha_i \phi(x_i)$

α_1 α_2 α_3 α_4 α_5 α_6

Only place of global interaction

softmax

score(x_1, x_4) score(x_2, x_4) score(x_3, x_4) score(x_4, x_4) score(x_5, x_4) score(x_6, x_4)

x_1 x_2 x_3 x_4 x_5 x_6

The $+p_1$ five $+p_2$ boxing$+p_3$ wizards $+p_4$ jump $+p_5$ quickly $+p_6$

$$\phi(x) = W_V^\top x$$

Self-attention weights

$$\text{softmax}(\theta)_\tau = \frac{\exp(\theta_\tau)}{\sum_{t=1}^{T} \exp(\theta_t)}$$

$$\text{score}(x, z) = \langle W_K x, W_Q z \rangle$$

Key - Query

Word embeddings

Positional encodings

TRANSFORMERS RECAP - ATTENTION HEAD

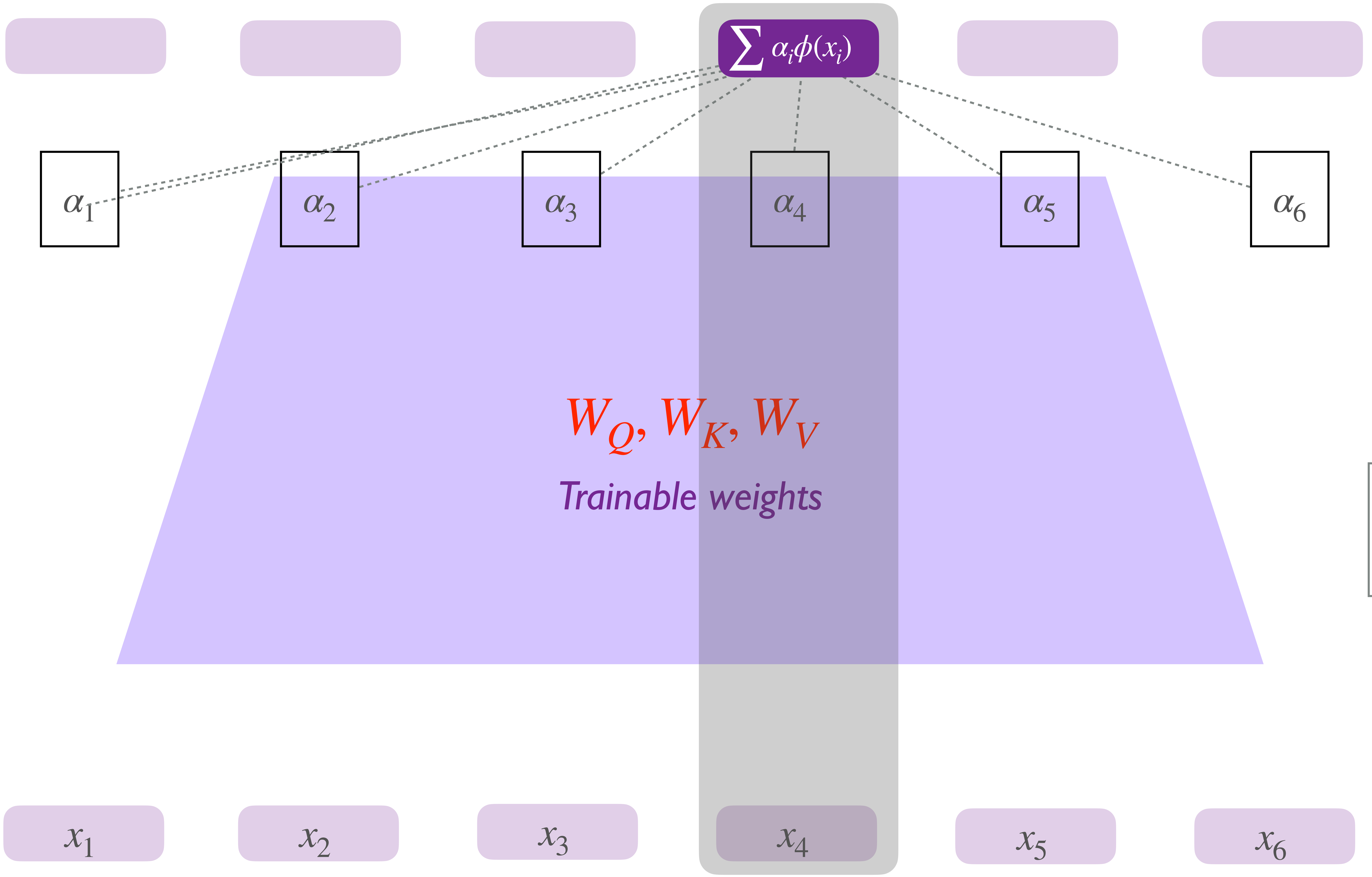
$x_i \in \mathbb{R}^d$

$W_V \in \mathbb{R}^{d \times d}$

$W_Q, W_K \in \mathbb{R}^{d \times d}$

$W_Q, W_K \in \mathbb{R}^{d \times d}$

$p_i \in \mathbb{R}^d$



The $+p_1$ five $+p_2$ boxing $+p_3$ wizards $+p_4$ jump $+p_5$ quickly $+p_6$

$\phi(x) = W_V^{\top} x$

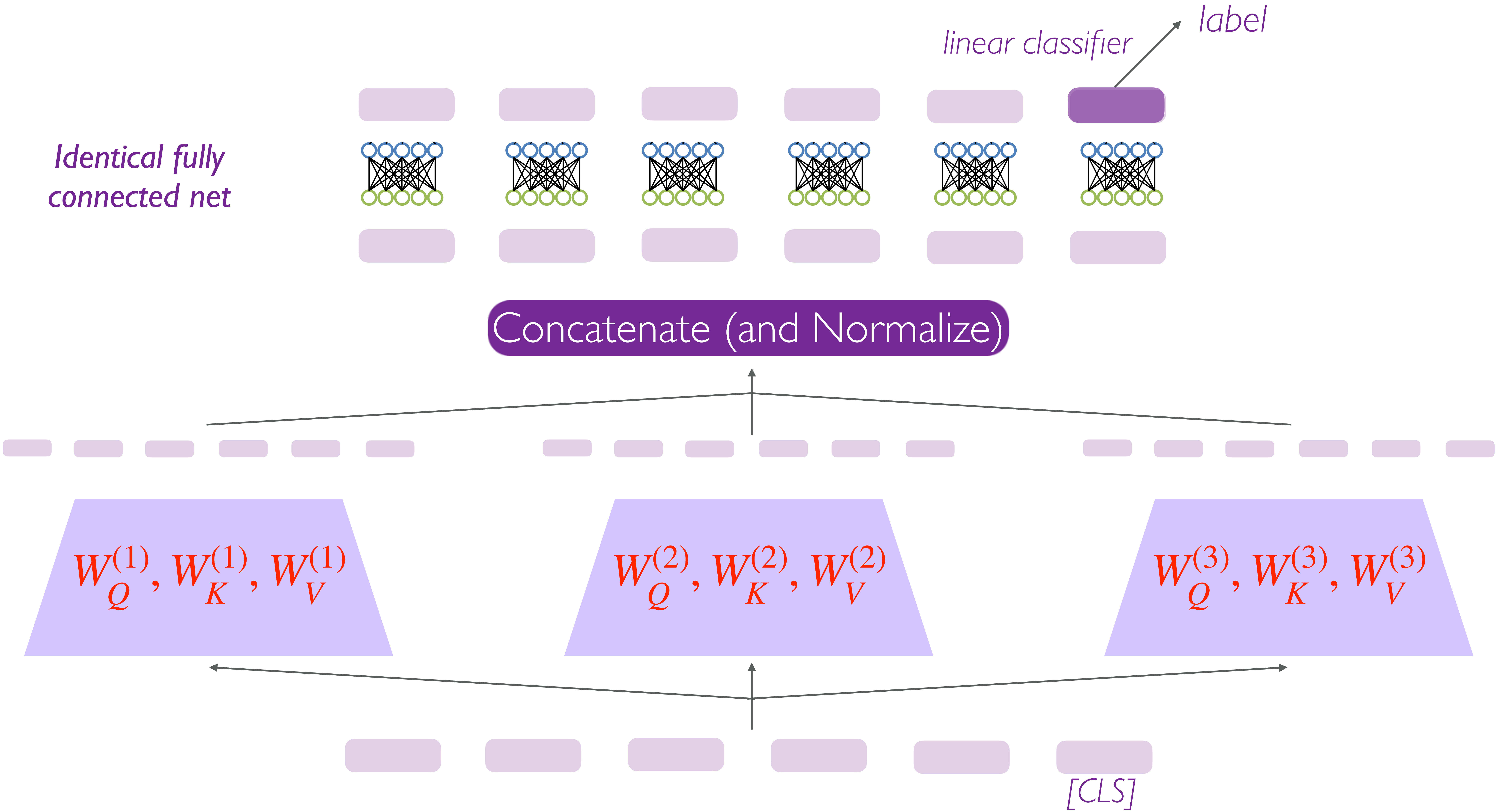
Self-attention weights

Attention head

$$\text{softmax}(\theta)_\tau = \frac{\exp(\theta_\tau)}{\sum_{t=1}^{T} \exp(\theta_t)}$$

$$\text{score}(x, z) = \langle W_K x, W_Q z \rangle$$

TRANSFORMERS RECAP - TRANSFORMER LAYER



Why Transformers?

Two important ideas:

- **computation:**
  - For an RNN/LSTM: the time to compute the loss, $\sum_{t=1} - \log \Pr(y_t \,|\, y_{<t}, \theta)$, on a T length sequence is O(T), and this is fundamentally a serial computation.
  - For a transformer, the serial compute can be O(1), i.e. no T, dependence, while the total computational complexity is $O(T^2)$

- **inductive bias:** (statistical/representational arguments)
  (granting the RNNs/LSTMs the serial overhead, they still seem to be worse)
  - The #parameters have no T-dependence
  - The transformers are able to create (sparse) features of things far apart.
  - Transformer are also able to “recall/copy” factual information from their context very easily.

Transformers:

Computational Graph Memory & Checkpointing

Transformer Memory: Forward & Computational Graph

Compute $Loss()$, (for $Bsz = 1, N_{heads} = 1$) :

input: parameters (embedding and MLP weights), data $X \in \{0,1\}^{TxV}$

d: hidden dim, B: batch size, T: context size, L: # layers, V: vocab size, N_{heads}: #heads

- Embed data: $X \leftarrow XW_{embed}$
- For $\ell = 0, \ldots L-1$
  - Attention:
    - $Q = XW_Q^\ell, K = XW_K^\ell, V = XW_V^\ell$
    - $X \leftarrow$ MaskedRowSoftmax$(QK^\top)V$
  - MLP layers: (dim $d \to 4d \to d$)
    - $X \leftarrow \sigma(XW_1^\ell)$
    - $X \leftarrow \sigma(XW_2^\ell)$
    - $X \leftarrow XW_{proj}^\ell$
- Compute $X \leftarrow XW_{unemebed}$, Return LogLoss

- Mem Transformer Params:

- Sufficient Memory for Forward pass:

- Memory Created in the Graph:

Transformer Memory: Forward & Computational Graph

Compute $Loss()$, (for $Bsz = 1, N_{heads} = 1$) :

input: parameters (embedding and MLP weights), data $X \in \{0,1\}^{TxV}$

d: hidden dim, B: batch size, T: context size, L: # layers, V: vocab size, N_{heads}: #heads

- Embed data: $X \leftarrow XW_{embed}$
- For $\ell = 0, \ldots L - 1$
  - Attention:
    - $Q = XW_Q^\ell, K = XW_K^\ell, V = XW_V^\ell$
    - $X \leftarrow$ MaskedRowSoftmax$(QK^\top)V$
  - MLP layers: ($d \rightarrow 4d \rightarrow d$)
    - $X \leftarrow \sigma(XW_1^\ell)$
    - $X \leftarrow \sigma(XW_2^\ell)$
    - $X \leftarrow XW_{proj}^\ell$
- Compute $X \leftarrow XW_{unemebed}$, Return LogLoss

With $N_{heads} \neq 1$

- Mem Transformer Params: $12d^2L + 2dV$
- Sufficient Memory for Forward pass: $4dT + N_{heads}T^2 + VT$
- Memory Created in the Graph: $12LdT + LN_{heads}T^2 + 2VT$

What about with $B \neq 1$?

Checkpointing ($B = 1$ case)

d: hidden dim, B: batch size, T: context size, L: # layers, V: vocab size, N_{heads}: #heads

- Embed data: $X \leftarrow XW_{embed}$
- For $\ell = 0, \ldots L - 1$
  - “block input” X (checkpoint here)
  - Attention:
    - $Q = XW_Q^\ell, K = XW_K^\ell, V = XW_V^\ell$
    - $X \leftarrow$ MaskedRowSoftmax$(QK^\top)V$
  - MLP layers: ($d \to 4d \to d$)
    - $X \leftarrow \sigma(XW_1^\ell)$
    - $X \leftarrow \sigma(XW_2^\ell)$
    - $X \leftarrow XW_{proj}^\ell$
- Compute $X \leftarrow XW_{unemebed}$, Return LogLoss

- Mem Transformer Params: $12d^2L + 2dV$
- Option 1: Checkpointing (ignoring the V part)
  - Memory for Checkpointing: LdT
  - Comp Graph Memory for Rematerialization: $12dT + N_{heads}T^2$
  - Computational overhead is basically a full factor of 2 (everything but W_{proj}^ℓ must be recomputed)
- Option 2: checkpoint everything but the $T \times T$ attention matrices
  - This saves on compute (the weight matrix multiples often costly) and needs $12LdT$ memory

Flash Attention Simplified:

- Single Head Attention: $X \leftarrow \text{MaskedRowSoftmax}(QK^\top)V$
  - This requires having T^2 free memory, even though our output is of size $d \times T$
  - The computational cost is $O(dT^2)$
- Simpler case: Suppose we just wanted to do the following, where $\exp()$ is componentwise:

$$X \leftarrow \exp(QK^\top)V$$

  - Again, this require T^2 free memory and the same flops.

Can we do better on memory, using the same flops?

- Observe that that row $X[t, :]$ is equal to the t-th row of $\exp(QK^\top)$ times V.
  - What is the t-th row of $\exp(QK^\top)$?
  - This implies: $X[t, :] = \exp(Q[t, :] \cdot K^\top)V$
- So we can compute X with a for loop over the T rows.
  - The excess memory is now $O(T)$
  - The flops is still $O(dT^2)$

- Now do you see how compute $X \leftarrow \text{MaskedRowSoftmax}(QK^\top)V$ with less memory?
- FlashAttention: Now just checkpoint this approach.
  - But why do we do this on a gpu?
  - Fusing: the exp operations can be “fused” with vector multiplies to reduce memory movements on the GPU itself (this is why it is done on the GPU)

Summary:

1. AutoDifferentiation+Checkpointing: computational backbone
2. GPU+Hardware Basics?

How do we put this together to build big models?