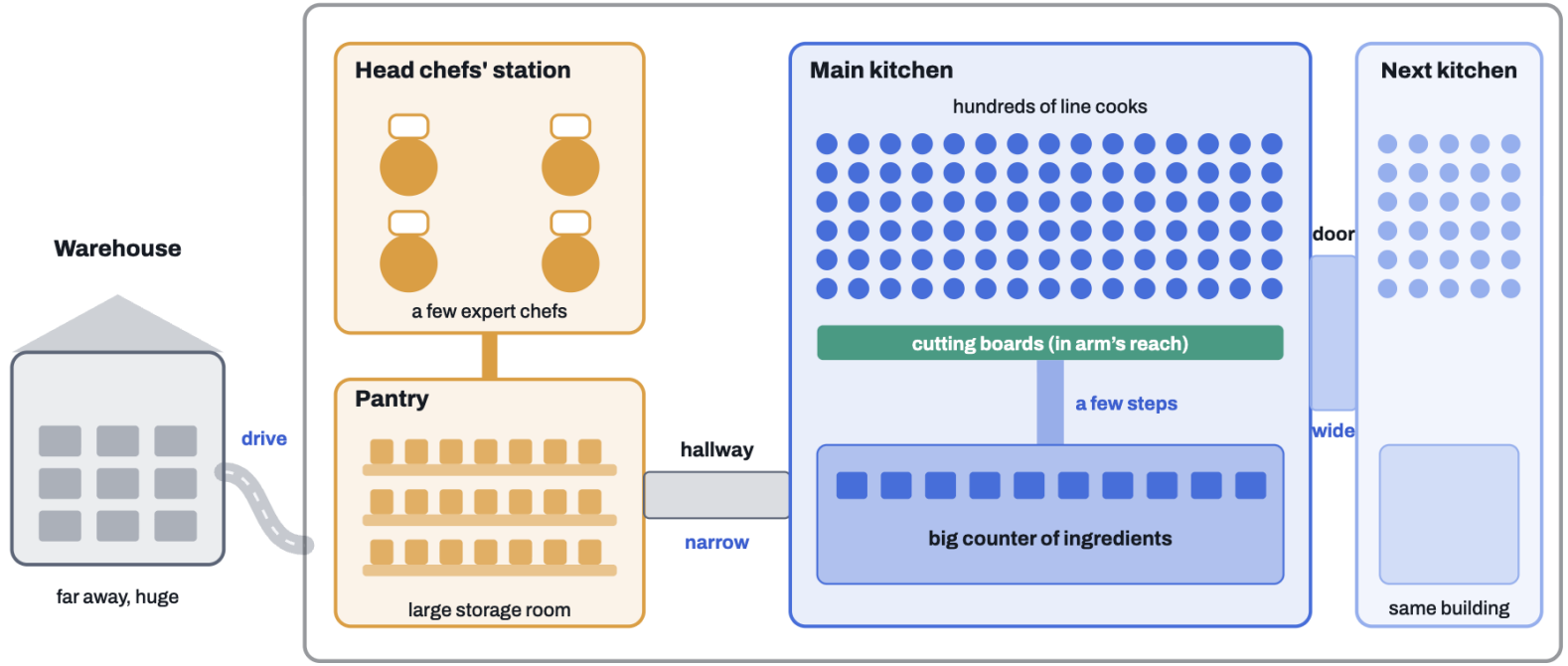
Distributed Training

Mohammad Salameh & Pat Virtue

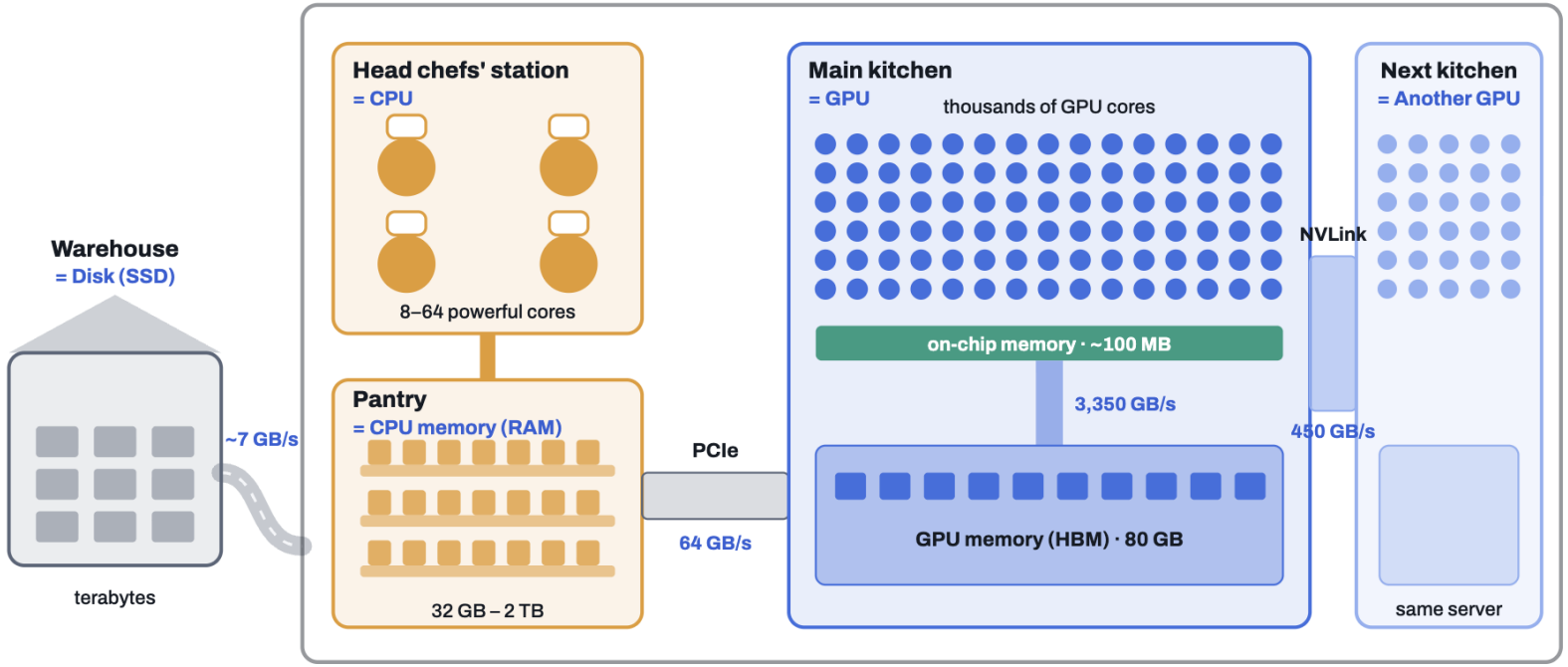
CPU, GPU and memory



The building = one computer. Food = data. Cooking = arithmetic.

Ingredients travel from the warehouse to the pantry, through a narrow hallway to the kitchen counter, and onto the cutting boards where hundreds of cooks work at once. The farther the trip, the longer the cooks wait.

CPU, GPU and memory



One server: disk, CPU, RAM, PCIe, GPUs. Every path has a speed limit.

Each room is a part of the computer: the pantry is RAM, the hallway is PCIe, the counter is GPU memory, and the line cooks are GPU cores.

CPU, GPU and memory

CPU: The computer's general-purpose processor. It runs your Python code and tells the GPU what to do.

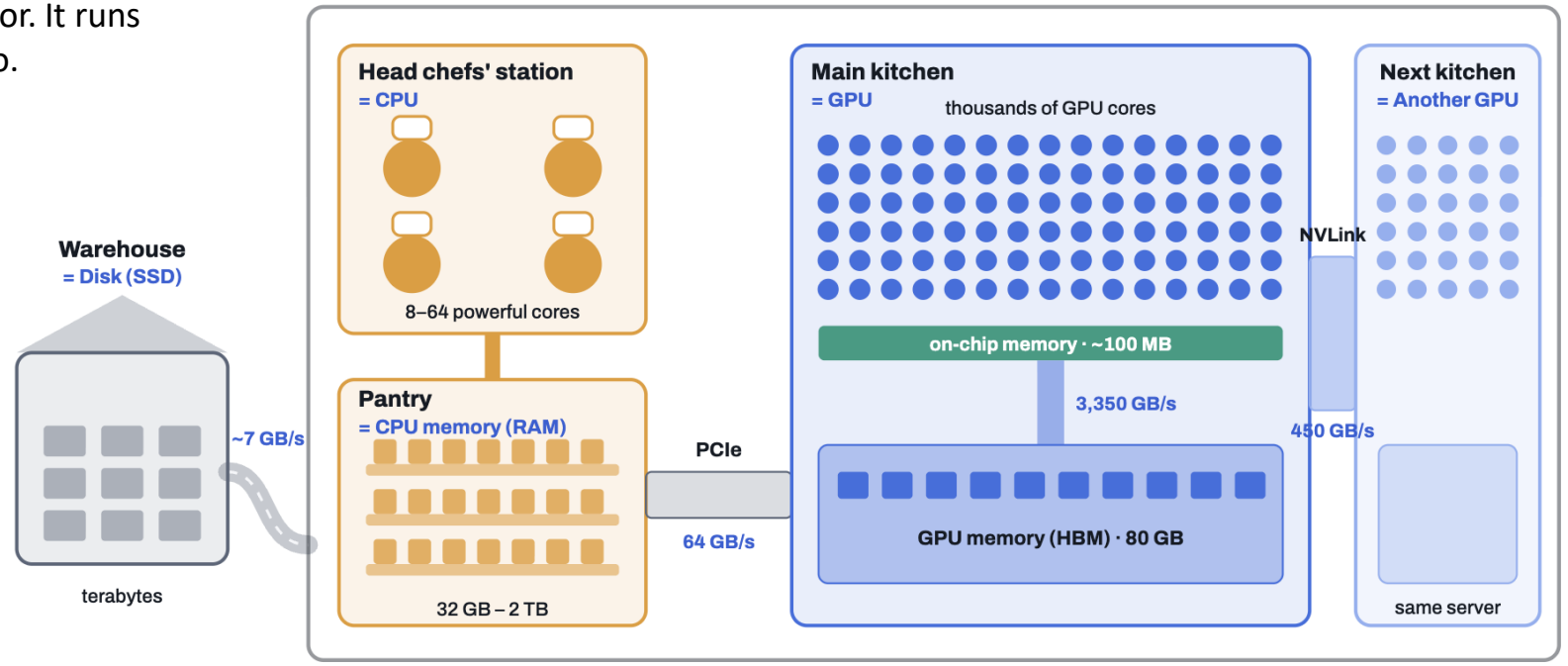
GPU: A processor with thousands of simple cores that do the same arithmetic on many numbers at once.

RAM: The CPU's main memory: fast, temporary storage. 16 GB to a few TB.

GPU memory (HBM): The GPU's own memory, right beside its chip. An H100 has 80 GB; a B200 has 192 GB.

Bandwidth How many bytes per second a connection can move, in GB/s or TB/s.

FLOP, FLOP/s One FLOP is one add or multiply on decimal numbers. FLOP/s counts them per second; 1 PFLOP/s is 10^{15} per second.



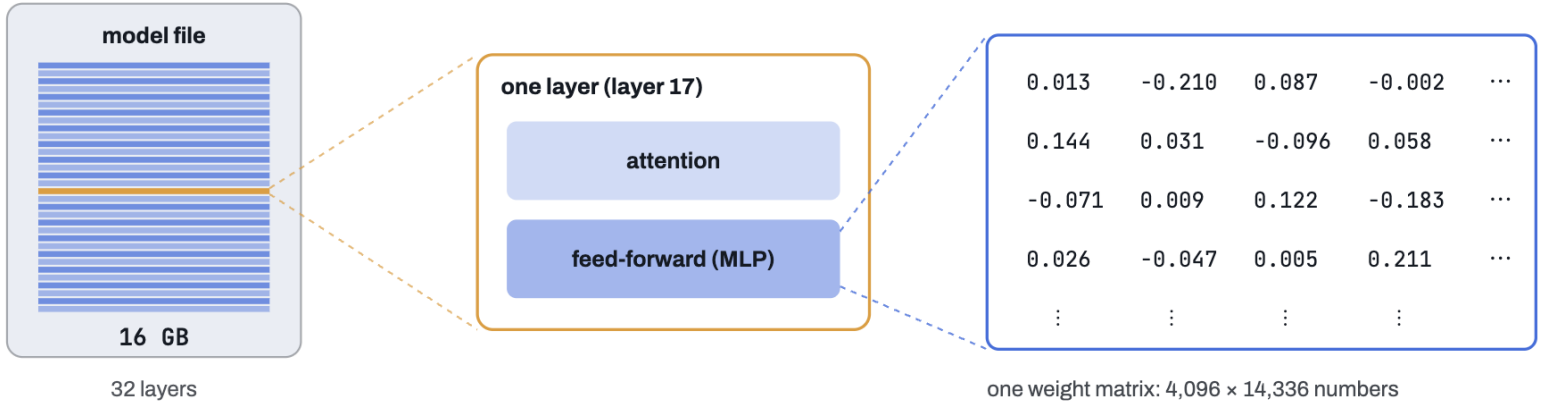
One server: disk, CPU, RAM, PCIe, GPUs. Every path has a speed limit.

Each room is a part of the computer: the pantry is RAM, the hallway is PCIe, the counter is GPU memory, and the line cooks are GPU cores.

Meet Llama 3 (8b)

- Open language model that Meta released in 2024.
- Give it some text and it predicts what comes next, one small piece of text at a time.
- Everything the model "knows" is stored as about **8 billion** ordinary decimal numbers, arranged in 32 layers.

What is inside the model file.
The file holds 32 layers. Each layer holds a few large matrices, and each matrix is a grid of millions of decimal numbers. 8.0 billion numbers × 2 bytes each = 16 GB.

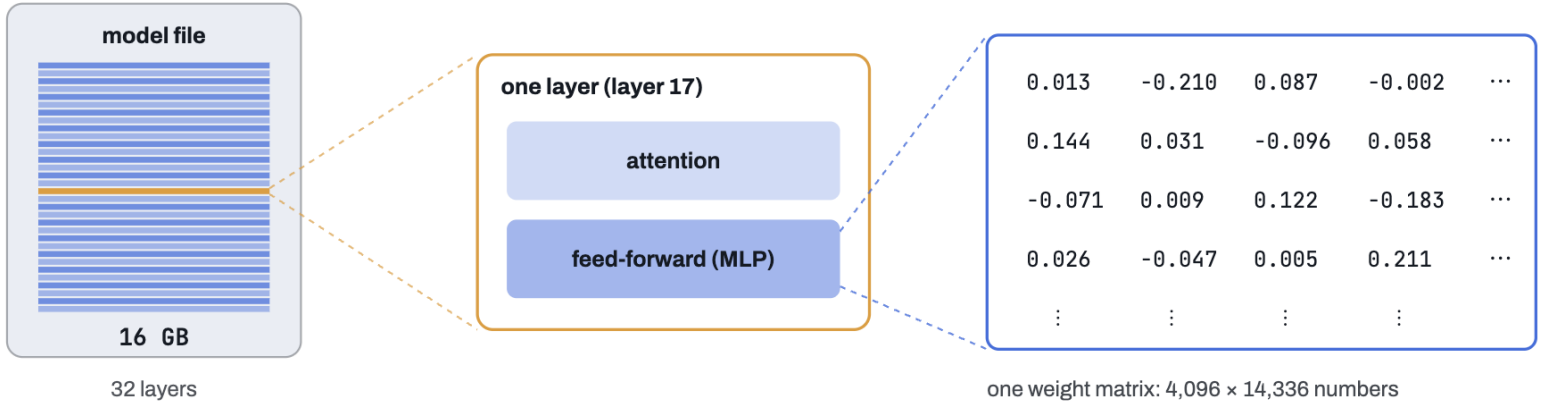


https://huggingface.co/unsloth/Llama-3.1-8B

Meet Llama 3 (8b)

- Open language model that Meta released in 2024.
- Give it some text and it predicts what comes next, one small piece of text at a time.
- Everything the model "knows" is stored as about **8 billion** ordinary decimal numbers, arranged in 32 layers.

What is inside the model file.
The file holds 32 layers. Each layer holds a few large matrices, and each matrix is a grid of millions of decimal numbers. 8.0 billion numbers × 2 bytes each = 16 GB.



Parameter (weight)

One of the numbers that make up the model. Llama 3 8B has about 8.0 billion.

Token

The unit of text a model reads and writes: a word or part of a word.

Tensor / matrix

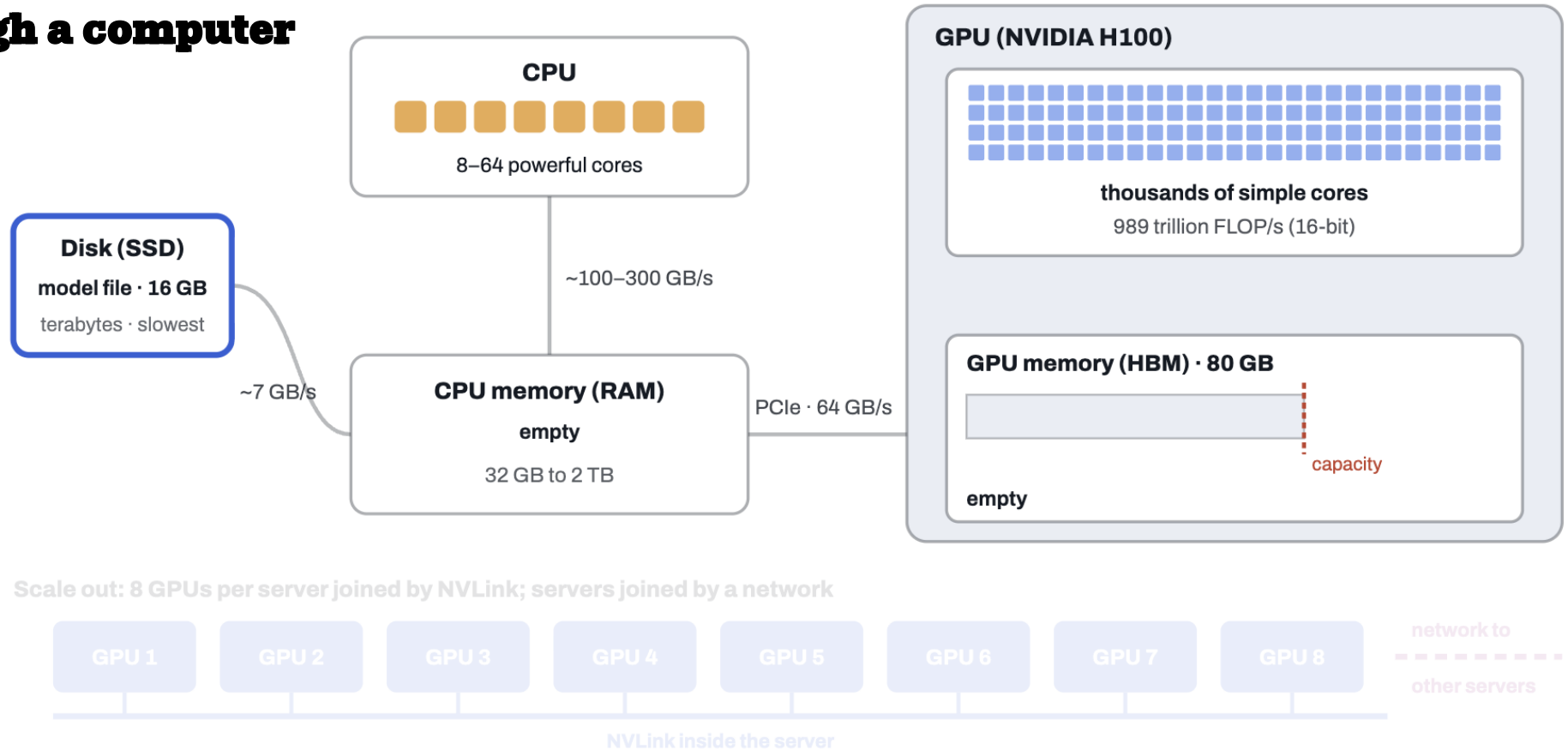
A table of numbers. A matrix is a 2-D grid; a tensor can have any number of dimensions.

Bytes and precision

FP32 takes 4 bytes per number; BF16 takes 2 bytes but is less precise.

https://huggingface.co/unsloth/Llama-3.1-8B

Follow Llama 3 8B through a computer

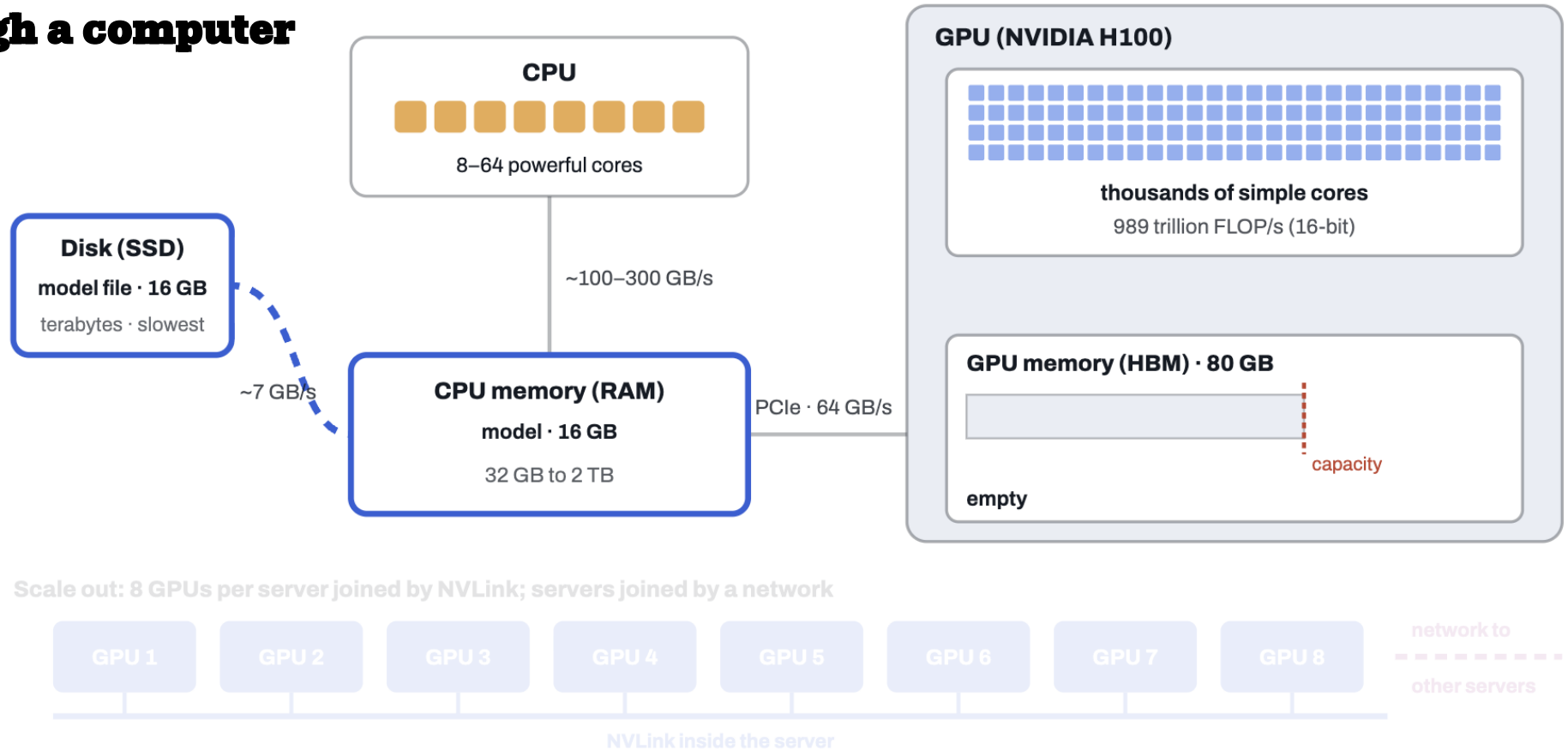


Stage 1 · On disk

The trained model is a 16 GB file of 8 billion numbers on the computer's disk (an SSD). A disk is large and keeps its data when the power is off, but it is far too slow to compute from directly.

8.0 billion parameters × 2 bytes each = 16 GB

Follow Llama 3 8B through a computer

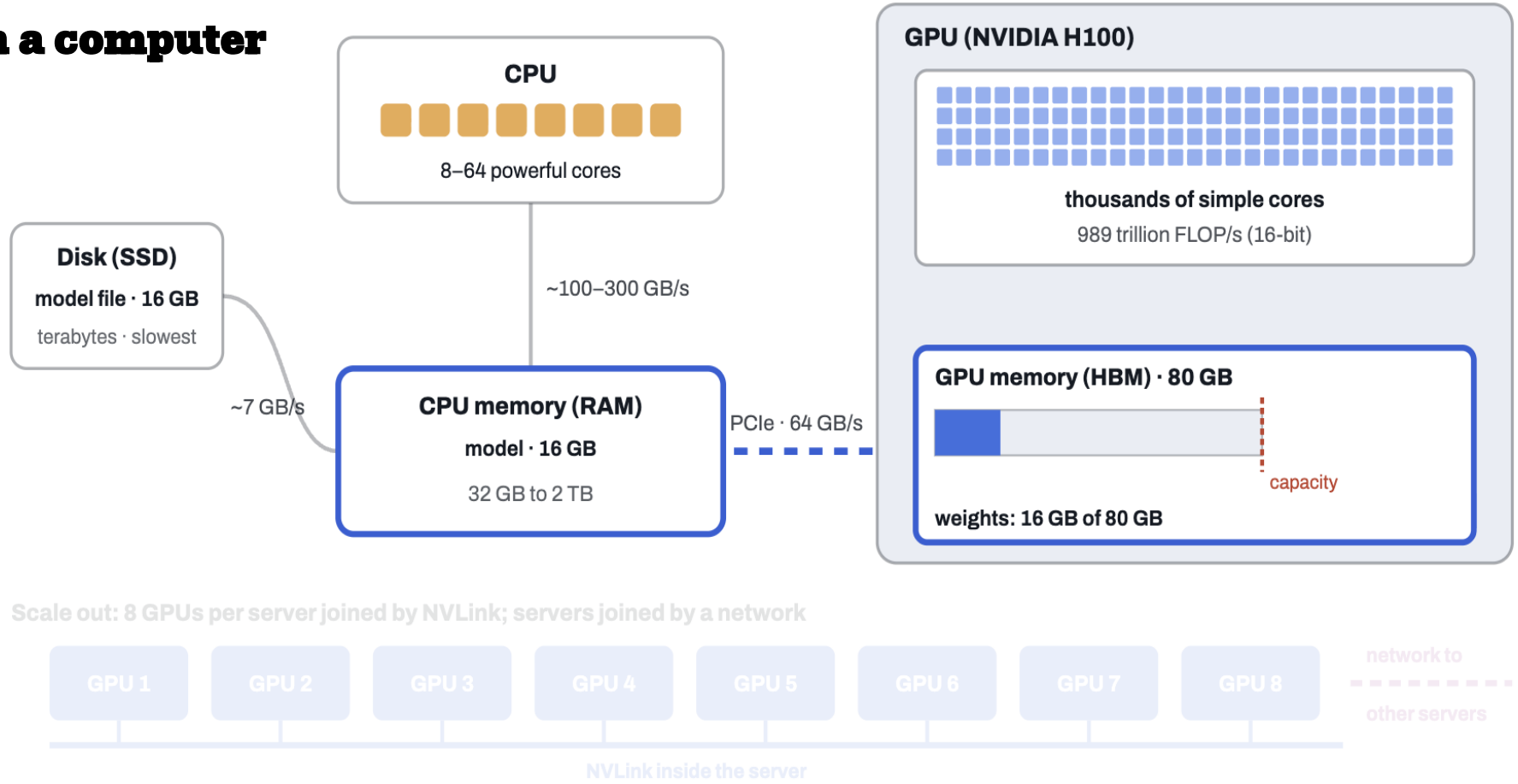


Stage 2 · Into CPU memory

To use the model, the computer first reads the file into its main memory (RAM). The CPU can reach RAM more than ten times faster than it can read the disk.

Reading 16 GB at about 7 GB/s takes 2–3 seconds

Follow Llama 3 8B through a computer

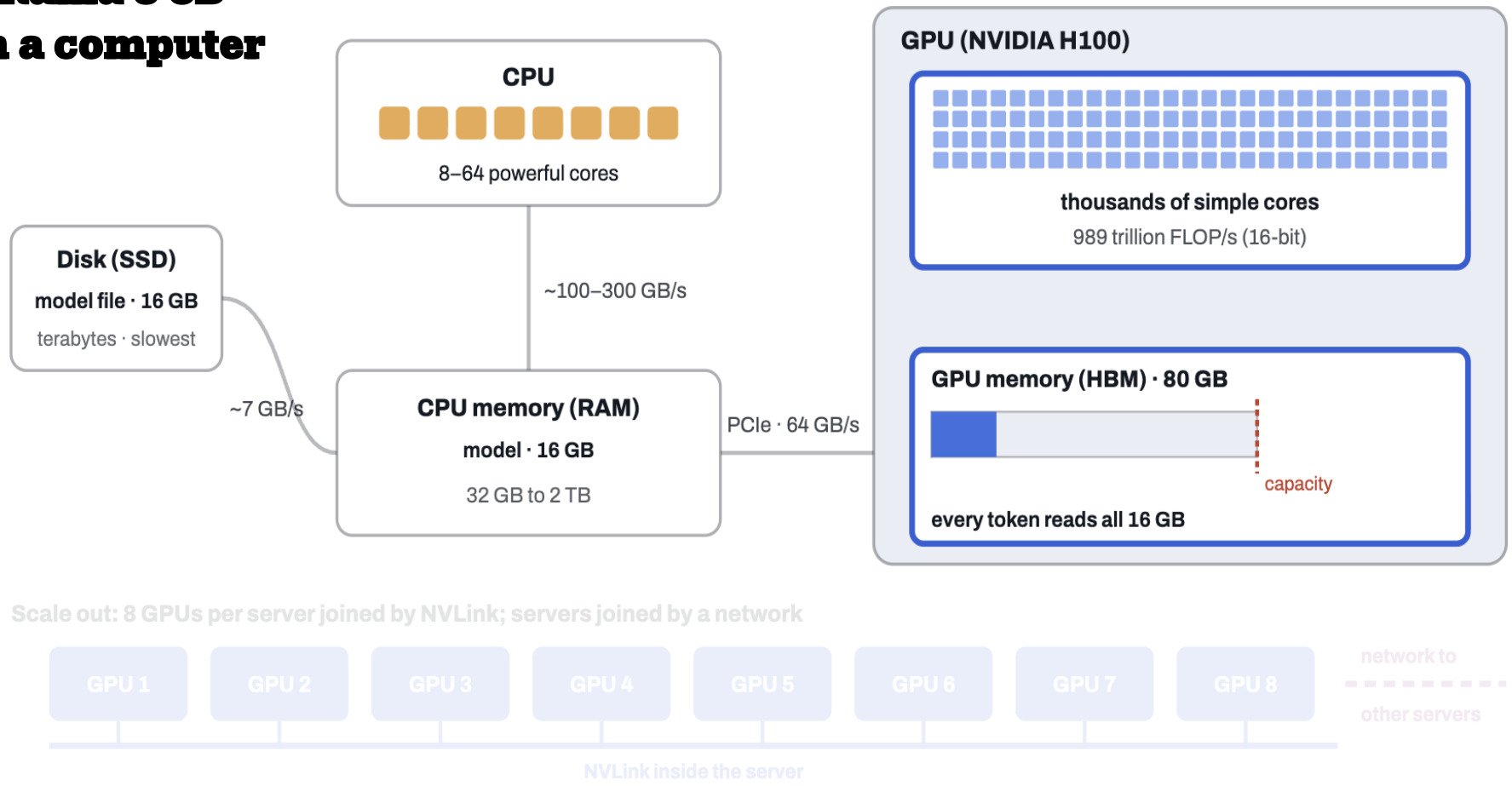


Stage 3 · Onto the GPU

The weights are copied over the PCIe connection into the GPU's own memory (HBM), right beside the GPU's compute cores. From now on the GPU does the arithmetic; the CPU only tells it what to do next.

Copying 16 GB at 64 GB/s takes about a quarter of a second

Follow Llama 3 8B through a computer

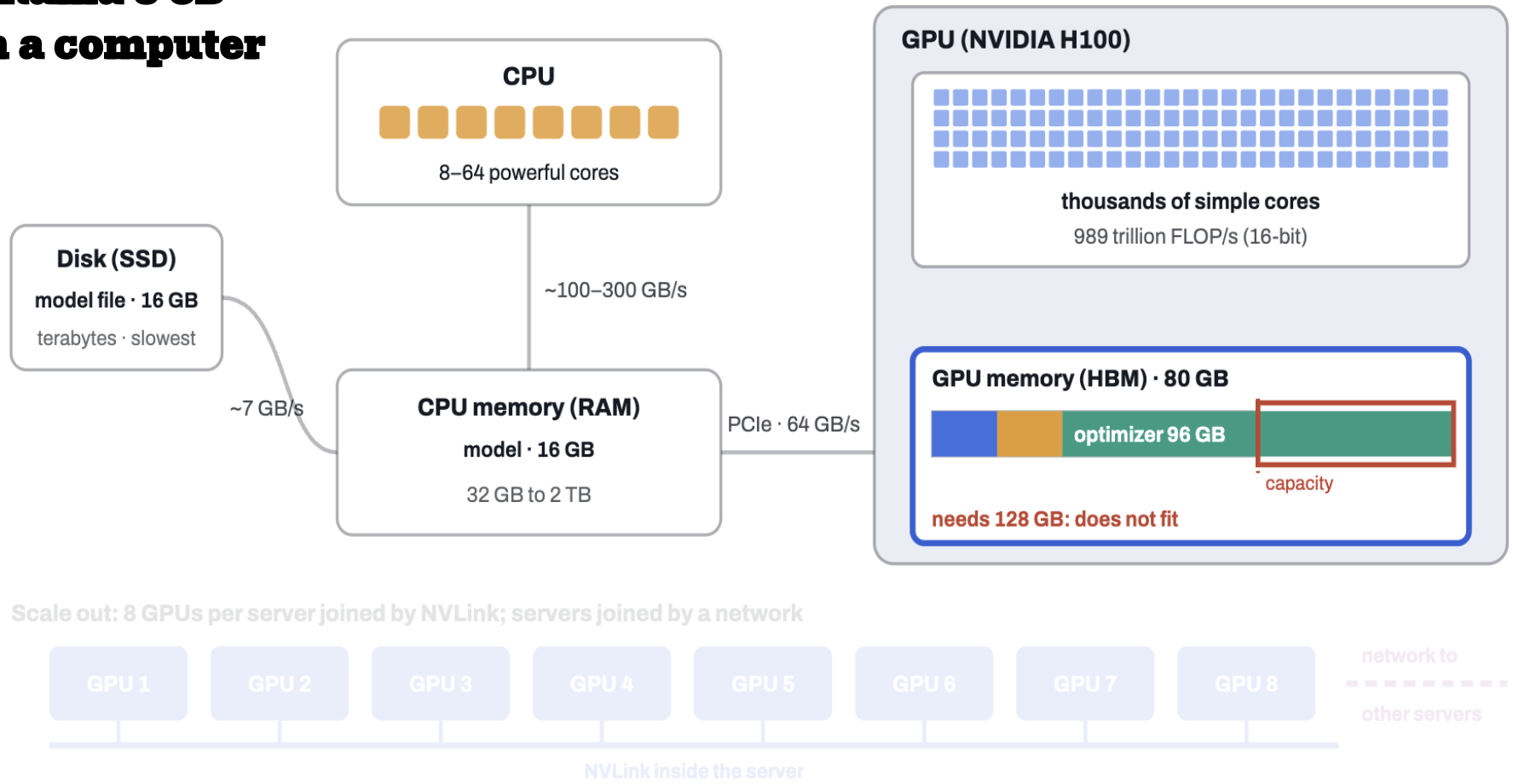


Stage 4 · Generating text (inference)

For every new token, the cores must read all 16 GB of weights from GPU memory and do about 8 billion multiply-adds (16 billion FLOPs). The arithmetic takes about 0.02 ms, but reading the weights takes about 5 ms, so the cores spend almost all their time waiting for data.

About 5 ms per token: at most ~200 tokens per second. From CPU memory at ~100 GB/s it would be ~160 ms per token.

Follow Llama 3 8B through a computer

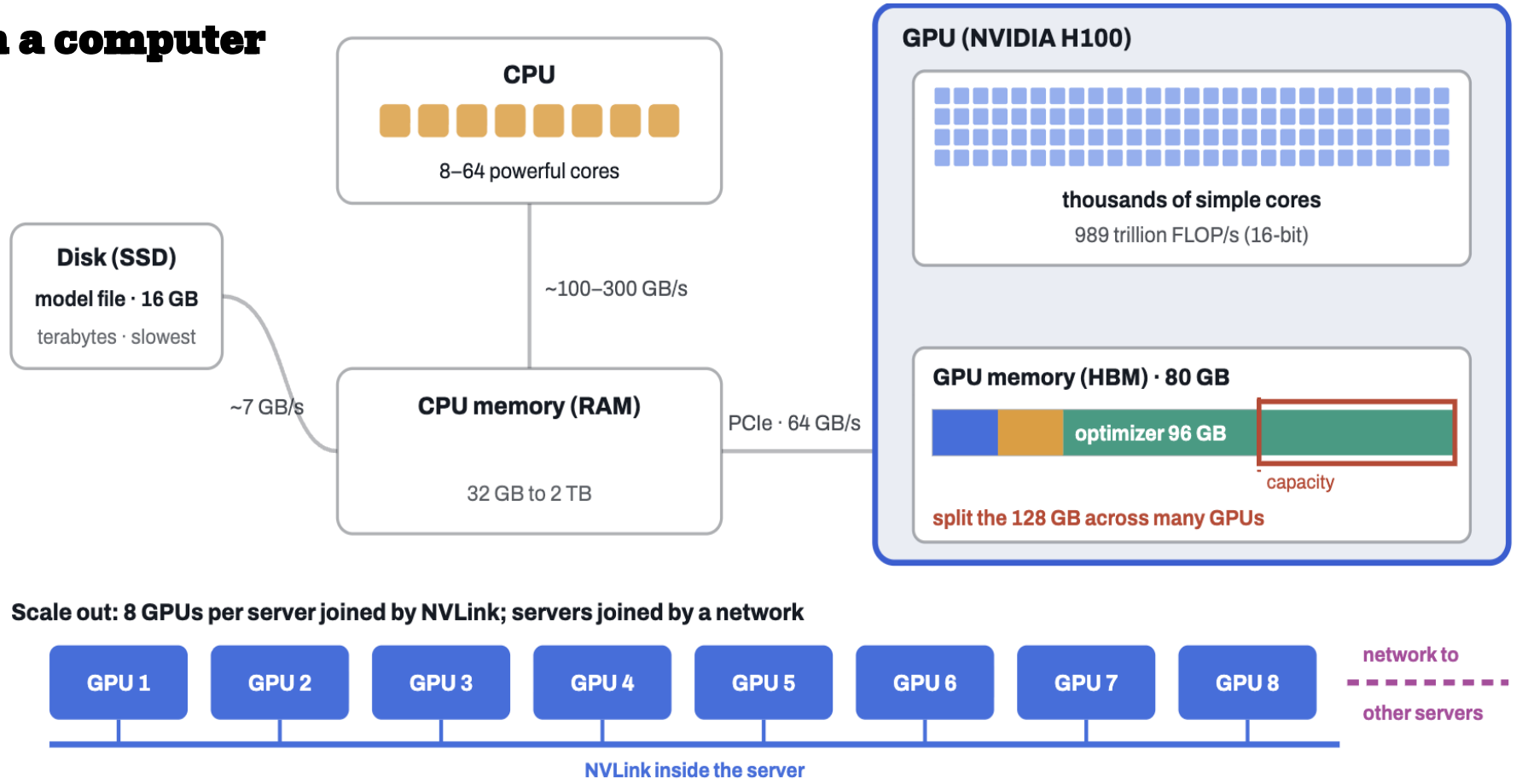


Stage 5 · Training

Training needs far more memory. Besides the weights, it keeps a gradient for every parameter, and the Adam optimizer keeps a 32-bit copy of each weight plus two running averages. That is 16 bytes per parameter, before counting activations.

weights 16 GB + gradients 16 GB + optimizer 96 GB = 128 GB, but the GPU holds only 80 GB

Follow Llama 3 8B through a computer



Stage 6 · Many GPUs

The way out is to spread training across many GPUs. Eight GPUs inside one server are joined by fast NVLink connections, and servers are joined by a network. Deciding what each GPU keeps and what the GPUs send each other is what the lecture is about.

Llama 3 8B needed ~7×10^{23} FLOPs: about 57 years on one GPU, or under 3 days on 8,192 GPUs working perfectly together

Memory Size

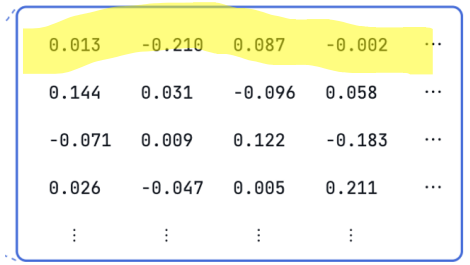
USED WHEN

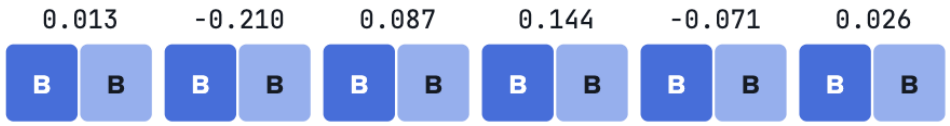
Deciding whether a model, or its training state, fits on a GPU; choosing the number format (precision) and how many GPUs to shard across

$$M = N \times b$$

UNIT

Bytes. Model sizes are usually quoted in GB (10^9 bytes); one H100 holds 80 GB.





× 8.03 billion values



SYMBOL	MEANING	UNIT
M	memory needed	bytes (GB = 10^9 bytes)
N	number of values stored (parameters, gradients, optimizer values)	count
b	bytes per value: 4 for FP32, 2 for BF16, 1 for FP8, 0.5 for 4-bit	bytes

Example. Llama 3 8B has 8.03 billion parameters. In BF16 (2 bytes each): $8.03 \times 10^9 \times 2 = 16.1 \times 10^9$ bytes ≈ **16 GB**.

In FP32 (4 bytes each) it would be 32 GB.

Efficiency ↓
lower M means fewer bytes to read and send, so faster

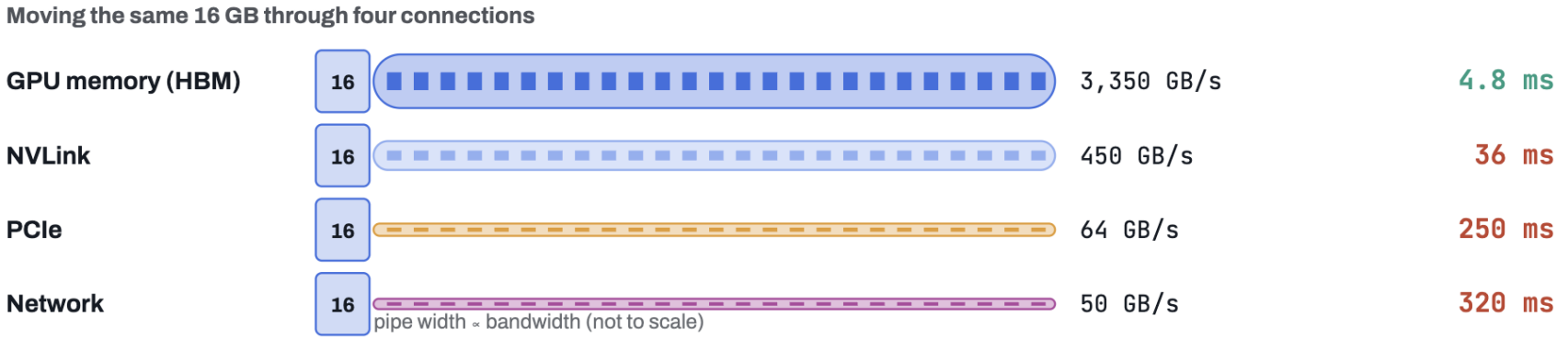
Size ↓
lower M means a smaller footprint and fewer GPUs

Greener ↓
fewer GPUs and less data movement use less energy

Bandwidth and transfer time

USED WHEN

Loading weights from GPU memory, copying data from CPU to GPU over PCIe, and exchanging gradients or activations between GPUs



$$t = \frac{M}{B}$$

WHAT EACH SYMBOL MEANS

SYMBOL	MEANING	UNIT
t	transfer time	seconds (ms, µs)
M	bytes to move	bytes
B	bandwidth of the link: bytes it moves per second	bytes/s (GB/s, TB/s)

Example

Reading Llama 3 8B's 16 GB from GPU memory at 3,350 GB/s takes 16 ÷ 3,350 = 0.0048 s = **4.8 ms**. Over PCIe at 64 GB/s, the same 16 GB takes 250 ms.

UNIT

Seconds for t; bytes per second for B (GB/s or TB/s).

Efficiency ↑ B, ↓ t
a faster link or fewer bytes shortens every transfer

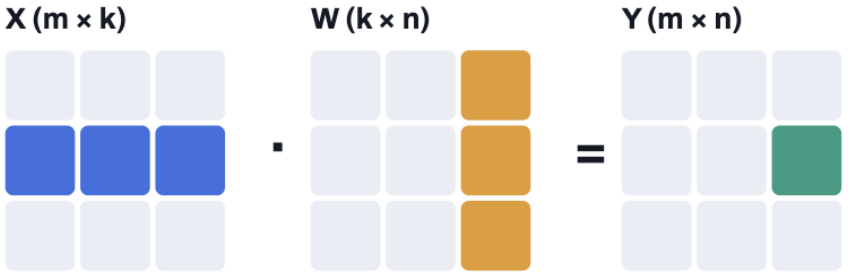
Size ↓ M
smaller tensors (lower precision, sharding) move faster

Greener ↓ M
moving data costs energy, so sending fewer bytes is greener

FLOPs of a matrix multiplication

USED WHEN

Estimating the cost of one layer, comparing model shapes, and predicting how long a kernel will take



One entry of Y (green), with k = 3:

$X_1 \cdot W_1 + X_2 \cdot W_2 + X_3 \cdot W_3$

3 multiplies + 2 adds ≈ 2k FLOPs

m × n entries × 2k FLOPs each = 2mkn

$$F = 2\,m\,k\,n$$

SYMBOL	MEANING	UNIT
F	floating-point operations (FLOPs) for one multiplication	FLOP
m	rows of X: number of tokens	count
k	inner dimension: hidden size	count
n	columns of W: output size	count

UNIT

FLOP (one add or multiply). Large counts use GFLOP (10^9), TFLOP (10^{12}) or PFLOP (10^{15}).

Example. One Llama 3 8B MLP matrix is 4,096 × 14,336. Multiplying 1,000 tokens by it costs 2 × 1,000 × 4,096 × 14,336 ≈ 1.2×10^{11} FLOPs, about 0.12 ms on an H100 at full speed.

Efficiency ↓
for the same result, fewer FLOPs finish sooner

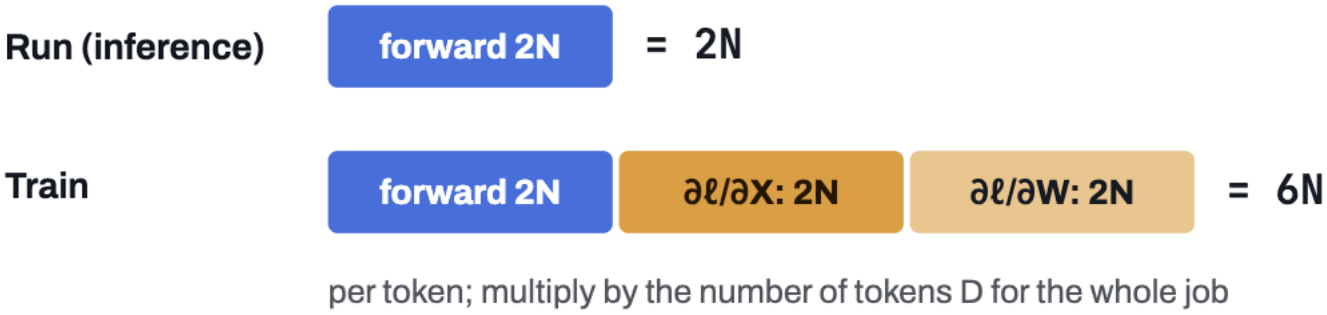
Size ↓ k·n
the matrix has k × n parameters, so a smaller matrix is also less memory

Greener ↓
every FLOP costs energy (about 0.7 pJ on an H100)

FLOPs per token: 2N to run, 6N to train

USED WHEN

Budgeting a training run, comparing models of different sizes, and explaining why MoE models are cheap per token.



Example. Llama 3 8B trained on 15 trillion tokens:
$6 \times 8.03\times10^{9} \times 15\times10^{12} \approx$ **7.2×10^{23} FLOPs**.

$$F_{\text{run}} \approx 2N$$

$$F_{\text{train}} \approx 6N$$

$$C \approx 6\,N\,D$$

SYMBOL	MEANING	UNIT
N	number of parameters (for MoE models, the active parameters)	count
D	number of training tokens	tokens
$F_{\text{run}}, F_{\text{train}}$	FLOPs per token to run or to train	FLOP per token
C	total training compute	FLOP

For one token ($m = 1$) and a $k \times n$ matrix, we have $2kn$ FLOPs. Each parameter costs 2 FLOPs: one multiply, one add. Over all matrices: **2N** for the forward pass.

The backward pass computes two gradients for every matrix : $\partial\ell/\partial X = G \cdot W^{\mathrm{T}}$ and $\partial\ell/\partial W = X^{\mathrm{T}} \cdot G$. Each costs **2N**, so **4N** in total.

Training = forward + backward = 2N + 4N = **6N** per token

Compute time and utilization (MFU)

USED WHEN

Planning a run, estimating cost (GPU-hours × price per hour), and comparing how well different systems use their GPUs.



$$T = \frac{C}{G \cdot P_{\text{peak}} \cdot \text{MFU}}$$

$$\text{MFU} = \frac{\text{achieved FLOP/s}}{P_{\text{peak}}}$$

$$\text{GPU-hours} = \frac{G \cdot T}{3600}$$

SYMBOL	MEANING	UNIT
T	wall-clock training time	seconds (or days)
C	total compute needed (from M5)	FLOP
G	number of GPUs	count
P_{peak}	peak speed of one GPU (989×10^{12} FLOP/s for H100 BF16)	FLOP/s
MFU	model FLOPs utilization: share of the peak actually achieved	fraction or %

speed × time = work. One GPU at its peak would finish ***F*** FLOPs in $F \div$ peak seconds. In practice a GPU achieves only a fraction of its peak, called **MFU** (model FLOPs utilization): the rest of the time it waits for data, for other GPUs, or for an idle pipeline stage. With ***G*** GPUs sharing the work evenly, each does F/G.

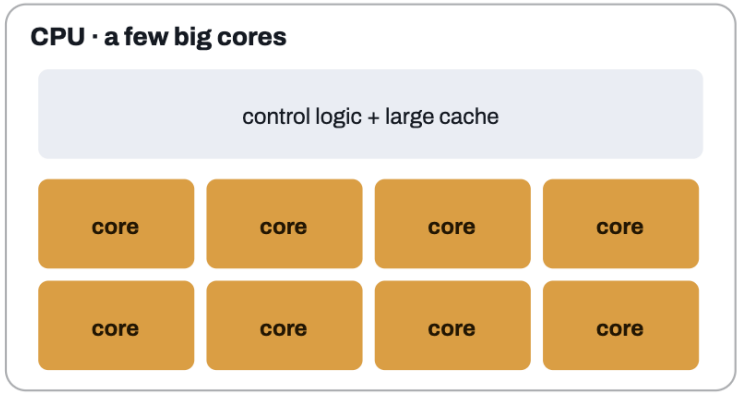
Hardware use is often reported in **GPU-hours**: number of GPUs × hours. 8,192 GPUs for 2.6 days ≈ 0.5 million GPU-hours. **At about $2 per GPU-hour, roughly $1 million.**

Example. 7.2×10^{23} FLOPs on one H100 (989×10^{12} FLOP/s peak) at 40% MFU: $7.2\times10^{23} \div (989\times10^{12} \times 0.4) \approx 1.8\times10^{9}$ s ≈ **58 years**. On 8,192 GPUs: about 2.6 days. Well-tuned large runs reach 38–43% MFU

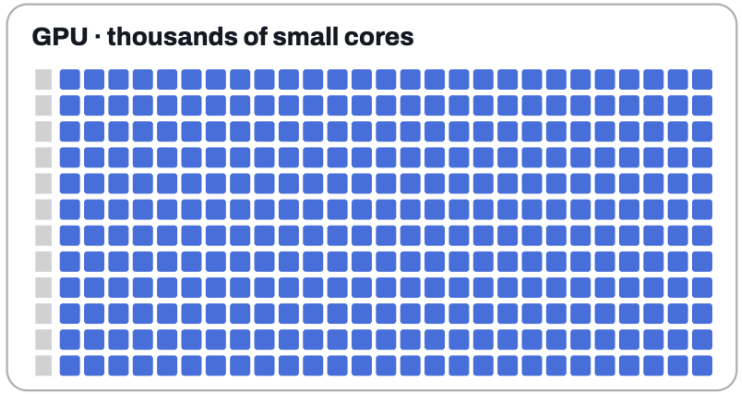
METRIC	EQUATION	UNIT	TYPICAL H100 VALUE
Memory size	`N × bytes per value`	GB	80 GB capacity
Transfer time	`M ÷ B`	s	3,350 GB/s memory · 450 GB/s NVLink · 64 GB/s PCIe
Latency	`α + M ÷ B`	µs	~10 µs for a small collective
Matmul FLOPs	`2mkn`	FLOP	—
FLOPs per token	`2N run · 6N train`	FLOP	—
Compute time	`F ÷ (G × peak × MFU)`	s, GPU-hours	989 TFLOP/s peak (BF16) · MFU 40%
Achieved speed	`min(peak, I × B)`	FLOP/s	ridge at ≈ 295 FLOP/byte
Energy	`P × t`	J, kWh	700 W max · ≈ 0.7 pJ per FLOP

CPU vs GPU

Few big cores, or thousands of small ones



Each core follows complicated instructions quickly

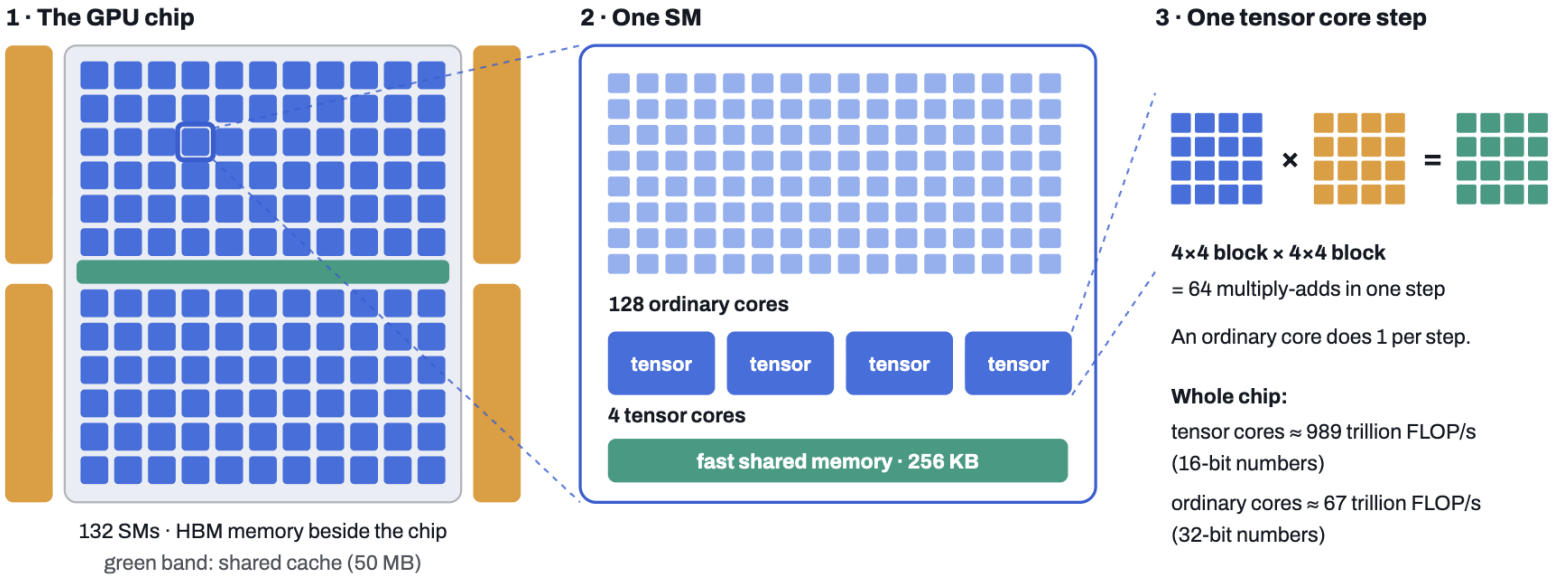


All cores do the same step on different numbers

- A **core** is one independent calculator inside a processor: it reads an instruction, carries it out, and moves on to the next one.
- A CPU has a few big cores (8 to 64), each able to follow complicated instructions with lots of decisions very quickly.
- A GPU has thousands of small, simpler cores that are designed to carry out the *same* instruction on *different* numbers at the same moment.

Inside the GPU

An NVIDIA H100's chip is divided into 132 identical processors called **streaming multiprocessors (SMs)**.
Each SM contains 128 ordinary cores, 4 **tensor cores**, and a small, very fast memory that its cores share.



SM (streaming multiprocessor)
One of the GPU's 132 identical building blocks, each with its own cores and fast memory.

Tensor core
A circuit that multiplies small blocks of matrices in one step. It provides most of a GPU's arithmetic power.

Kernel
One program the GPU runs, such as one matrix multiplication. A training step runs thousands of kernels.

Memory: How fast numbers reach the cores?

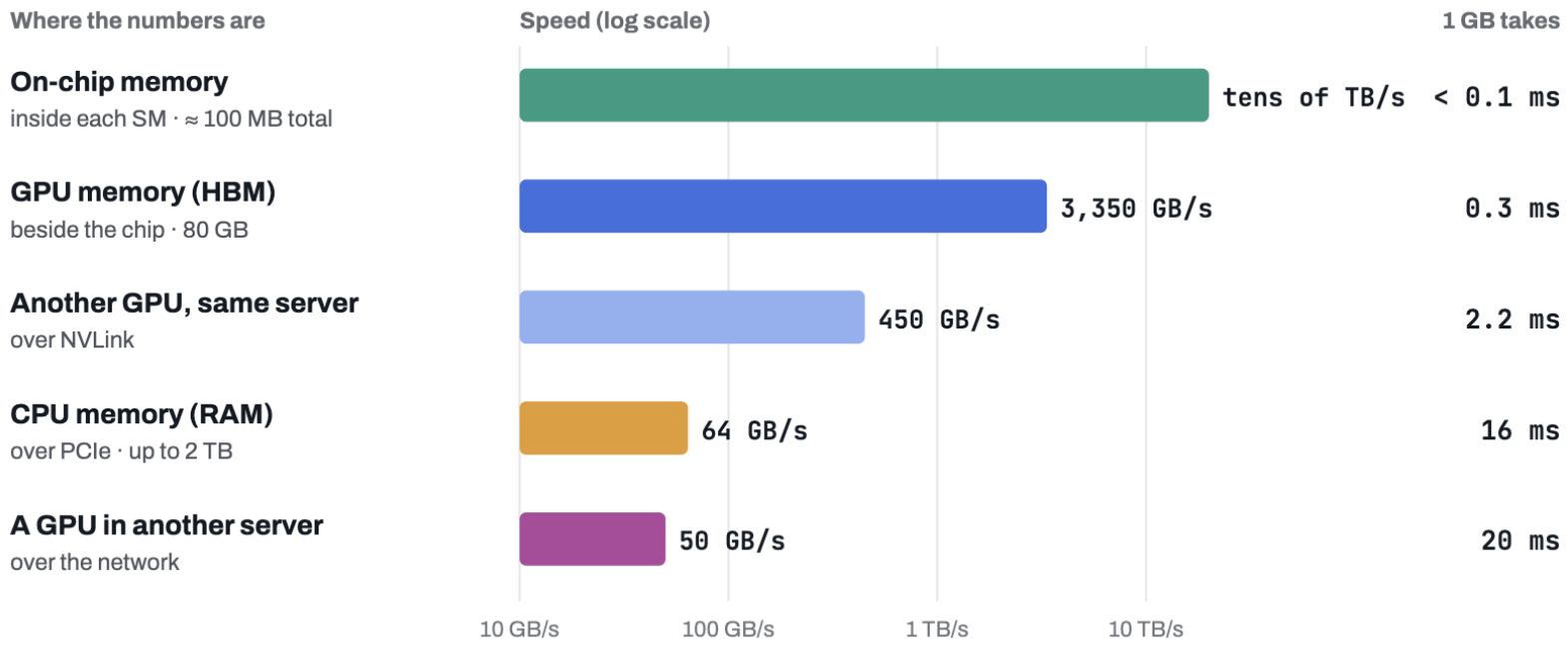
Before a core can compute, the numbers it needs must travel to it. Memory comes in levels: the closer a level is to the cores, the faster and smaller it is

On-chip memory: Small, extremely fast storage inside the chip: registers (each core's scratch space), shared memory (shared by one SM) and cache (recently used data). About 100 MB in total.

HBM (GPU memory): High-bandwidth memory: the GPU's main memory, 80 GB on an H100, stacked right beside the chip.

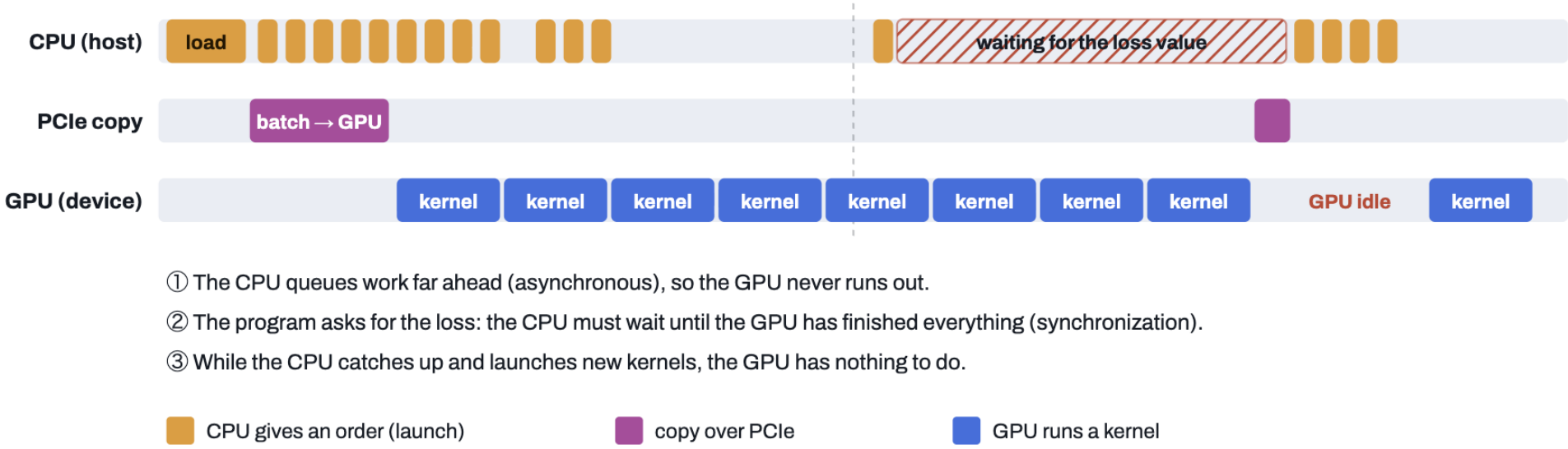
PCIe: The standard connection between the CPU and devices like the GPU. Much slower than the GPU's own memory

NVLink: NVIDIA's direct GPU-to-GPU connection inside a server: about 7× faster than PCIe.



How fast numbers reach the cores. Bars use a log scale: each grid line is 10× faster. Moving 1 GB from CPU memory takes about 50 times longer than from GPU memory. Speeds are per direction for an H100 server

How the CPU and GPU work together?



The CPU is the host and the GPU is the device. The CPU runs your Python program and decides what to do; the GPU does the arithmetic. A training step goes like this

1. The CPU loads a batch of training data from disk into CPU memory, then copies it to GPU memory over PCIe.
2. The CPU **launches** kernels: it places orders such as "multiply these matrices" into a queue on the GPU. It does not wait for each order to finish, it immediately moves on to the next line of code. This is called **asynchronous** execution.
3. The GPU works through its queue as fast as it can.
4. Results come back to the CPU only when the program asks for them, such as printing the loss.

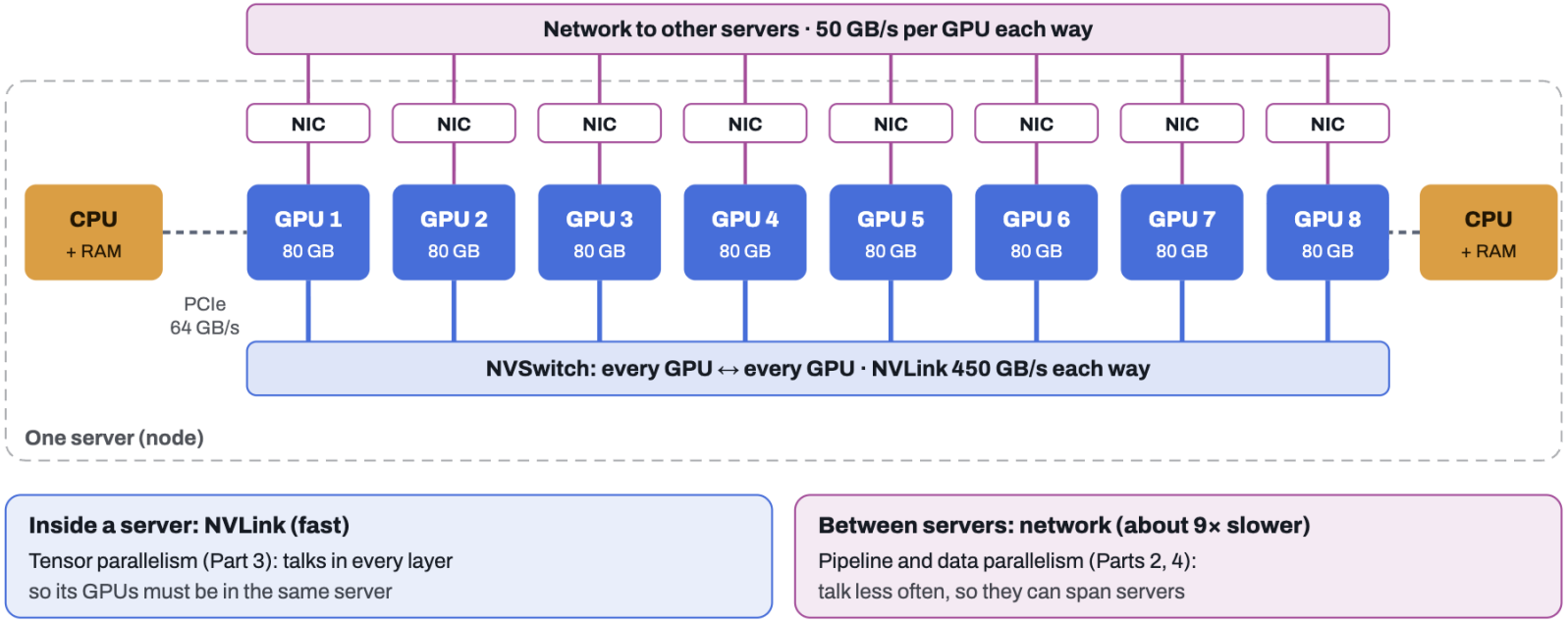
Many GPUs in one machine
Many machines in a cluster

A training **server** holds 8 GPUs.

Inside it, **NVSwitch** chips connect every GPU directly to every other GPU over NVLink.

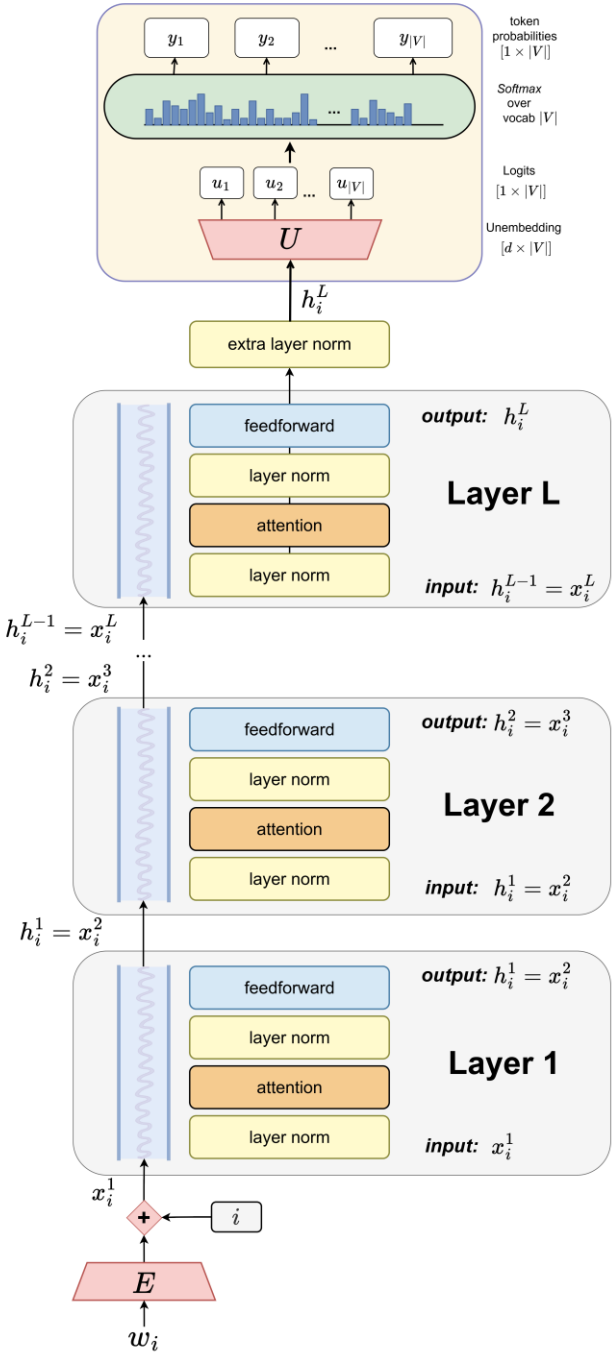
Each GPU also has its own **network card** for talking to GPUs in other servers, which is about 9× slower than NVLink.

In Distributed Training, techniques that requires more communication stay inside one server, while techniques that require less can stretch across servers.

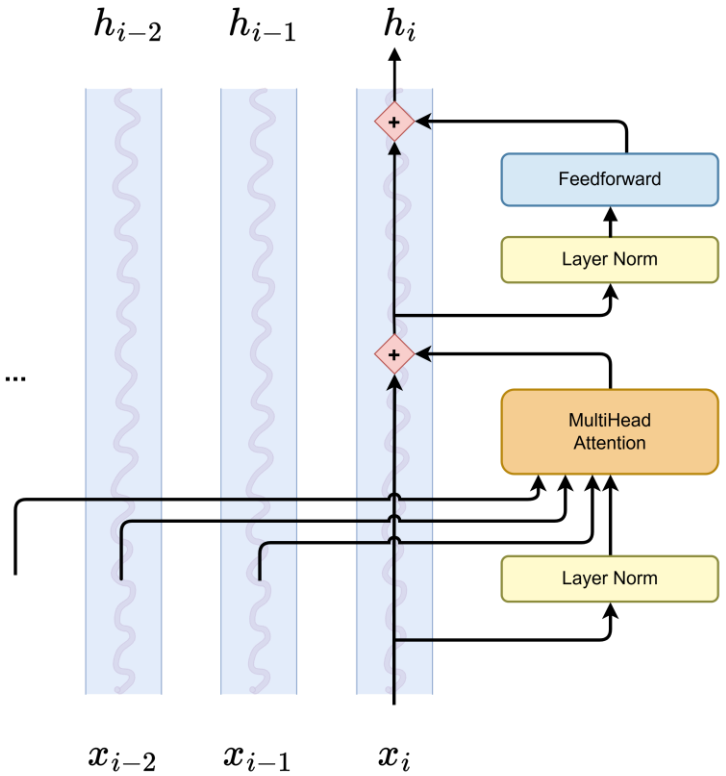


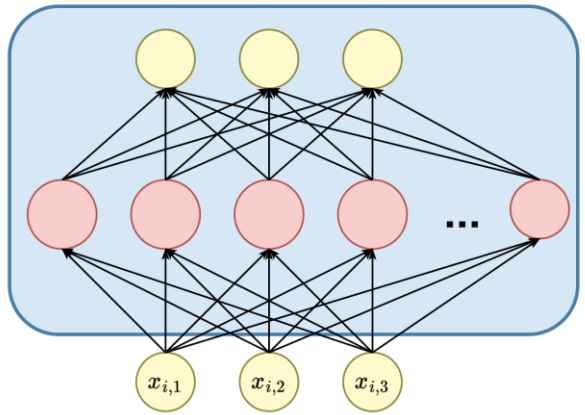
Inside a server: NVLink (fast)
Tensor parallelism (Part 3): talks in every layer
so its GPUs must be in the same server

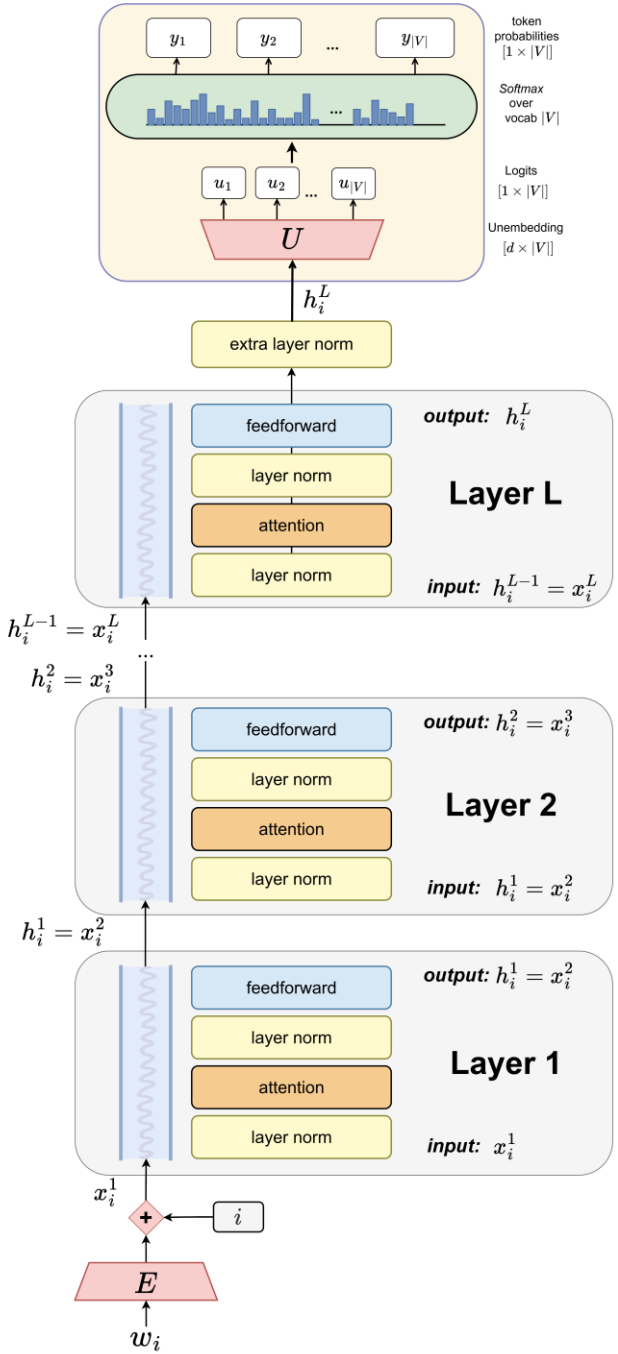
Between servers: network (about 9× slower)
Pipeline and data parallelism (Parts 2, 4):
talk less often, so they can span servers



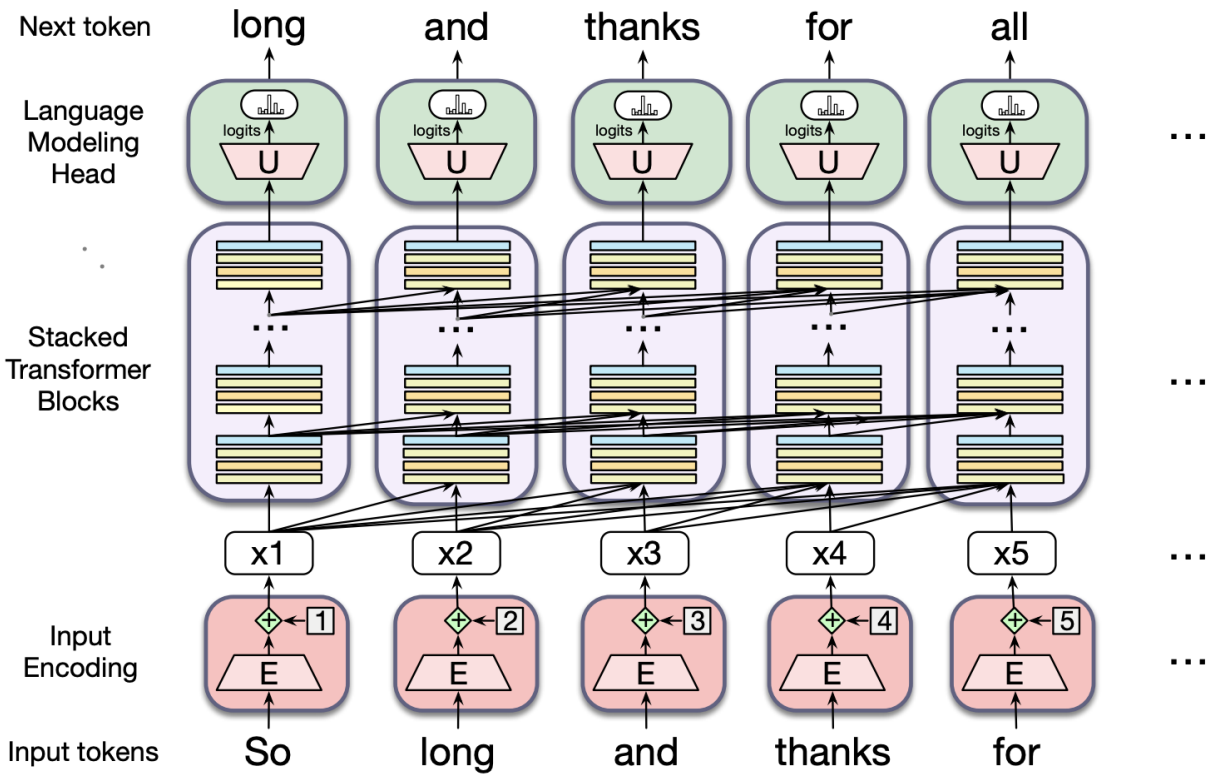
Transformer-based LM
Stacks multiple
transformer blocks







Decoding: generating one token at a time.
Each token only attends to previous ones



Speech and Language Processing. Daniel Jurafsky & James H. Martin
https://web.stanford.edu/~jurafsky/slp3/7.pdf

Why Distribute?

Training a large model needs thousands of GPUs.

GPUs keep getting faster, but the wires between them are much slower than the math inside them

The **art is to split** the work while sending as little data as possible.

	A100	H100	B200
GPU memory	80 GB	80 GB	192 GB
BF16 compute (dense)	0.31 PFLOP/s	0.99 PFLOP/s	2.25 PFLOP/s
Memory bandwidth	2.0 TB/s	3.35 TB/s	8 TB/s
GPU-to-GPU (NVLink)	0.6 TB/s	0.9 TB/s	1.8 TB/s
Approximate price	~$10K	~$25K	~$50K

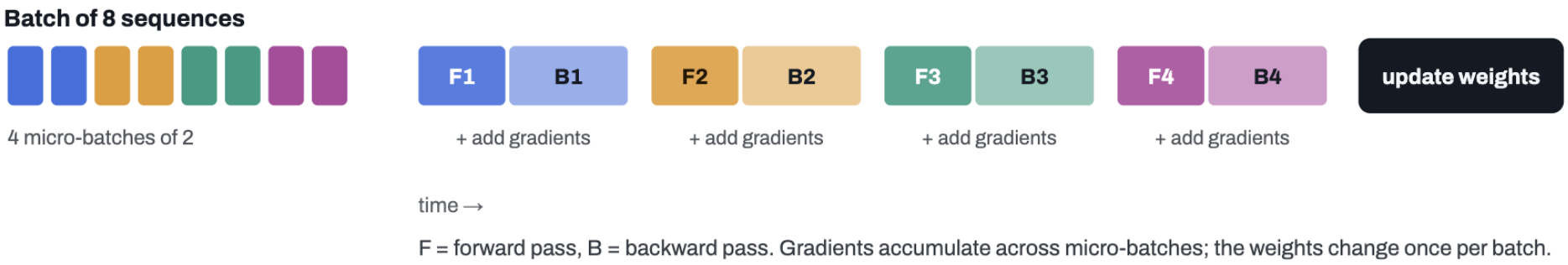
An H100 can do 989 trillion FLOPs per second but can receive only about 0.45 trillion bytes per second from another GPU over NVLink.

To keep it busy, it must do about **2,200 FLOPs for every byte** it receives. Over the network between servers (about 50 GB/s) the ratio is about **20,000 FLOPs per byte**. Any plan that makes GPUs talk a lot will leave them waiting.

Batches and micro-batches

A **batch** is the set of training examples used for one weight update. Large models use enormous batches: Llama 3 405B used 16 million tokens per step. That is far too much to push through a GPU at once.

So the batch is cut into **micro-batches**. The GPU runs forward and backward on one micro-batch at a time, adds up the gradients, and updates the weights only after the last one. Adding up gradients this way is called **gradient accumulation**.



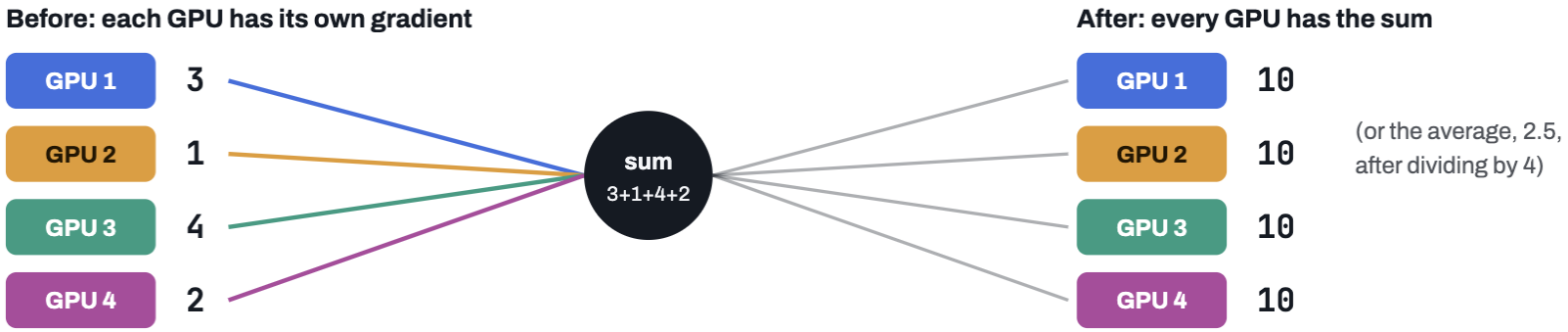
F = forward pass, B = backward pass. Gradients accumulate across micro-batches; the weights change once per batch.

GPUs working together

A **server** (also called a node) usually holds 8 GPUs. Inside a server, GPUs are linked by **NVLink**, NVIDIA's fast GPU-to-GPU connection (900 GB/s on an H100). Servers are linked by a slower **network**, about 50 GB/s per GPU in each direction.

A **cluster** is many servers. Meta's training clusters each hold 24,576 H100 GPUs.

Sending Llama 3 8B's 16 GB of gradients takes about 18 ms over NVLink but about 320 ms over the network.



All-Reduce: Sum and broadcast (gradients)

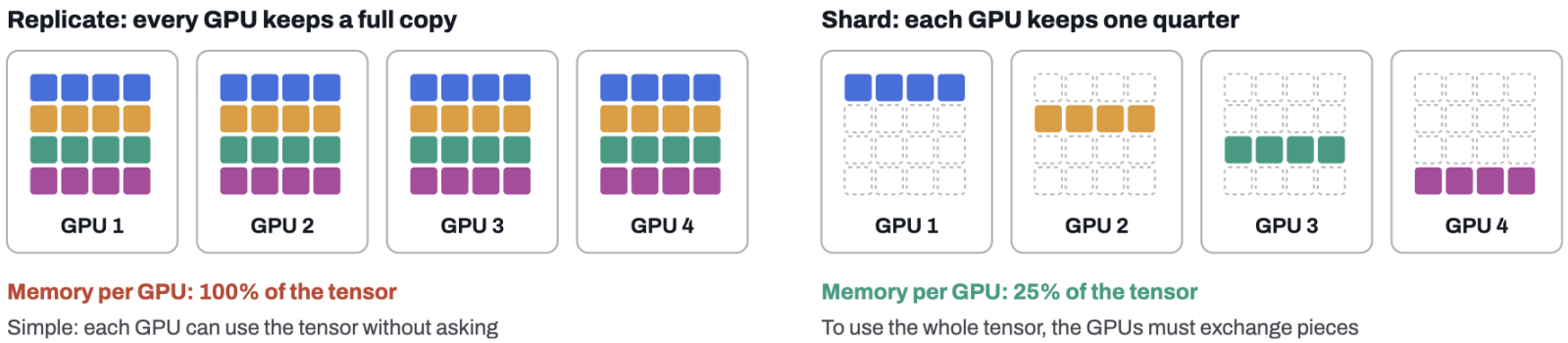
Every GPU contributes a number (or a whole tensor), they are added up, and every GPU receives the total

GPUs working together

A **server** (also called a node) usually holds 8 GPUs. Inside a server, GPUs are linked by **NVLink**, NVIDIA's fast GPU-to-GPU connection (900 GB/s on an H100). Servers are linked by a slower **network**, about 50 GB/s per GPU in each direction.

A **cluster** is many servers. Meta's training clusters each hold 24,576 H100 GPUs.

Sending Llama 3 8B's 16 GB of gradients takes about 18 ms over NVLink but about 320 ms over the network.

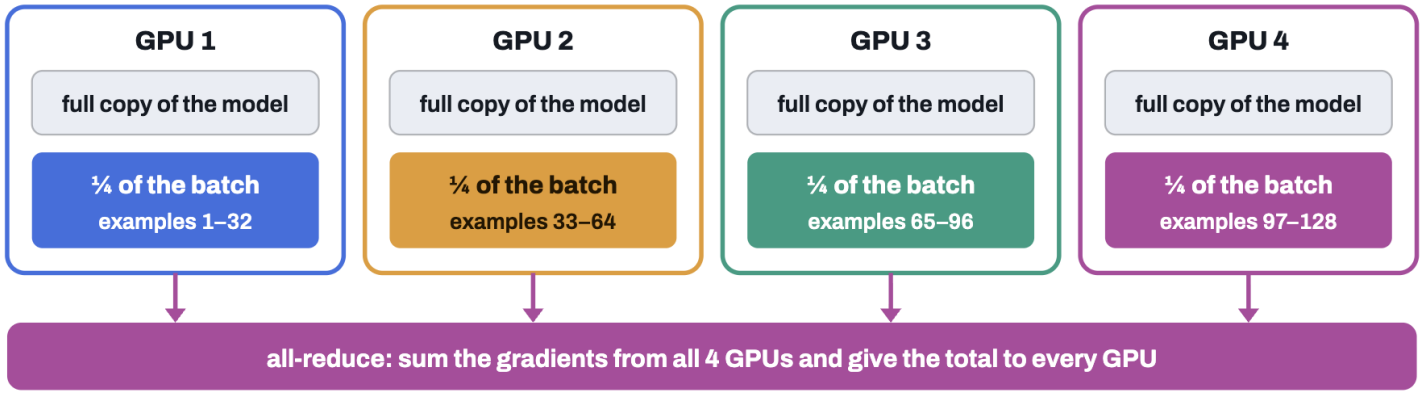


Replicate or Shard

Every technique in the lecture decides, for each tensor, whether every GPU keeps a full copy (**replicate**) or each GPU keeps only a piece (**shard**).

Data Parallelism

1. Copy the whole model onto every GPU.
2. Split each batch evenly: with 4 GPUs, each gets a quarter of the examples.
3. Each GPU runs the forward and backward pass on its own examples and gets its own gradients.
4. All-reduce the gradients: every GPU ends with the same summed gradient.
5. Every GPU applies the same update, so the copies stay identical.



Example: a batch of 128 examples, 32 per GPU. One exchange per training step, after the backward pass.

Data parallelism works well when the model is small.

Llama 3.2 1B needs 1.24B × 16 bytes ≈ **20 GB to train**, so every GPU can hold its own copy.

8 GPUs then process 8 slices of the batch at once and finish each step about 8× faster, exchanging 2.5 GB of gradients per step (about 10 ms over NVLink).

Data Parallelism

Llama 3.2 1B

2.5 GB

Does each GPU's copy fit?

With data parallelism, **every GPU must hold the whole model**. Pick a model, a job and the hardware, or press Play to step through the scenarios. Each tube is one GPU's memory; red means it overflows.

Llama 3.2 1B | Llama 4 Scout | Llama 4 Maverick | Llama 4 Behemoth

Run it | Train it

16-bit | 8-bit | 4-bit

H100 · 80 GB | B200 · 192 GB

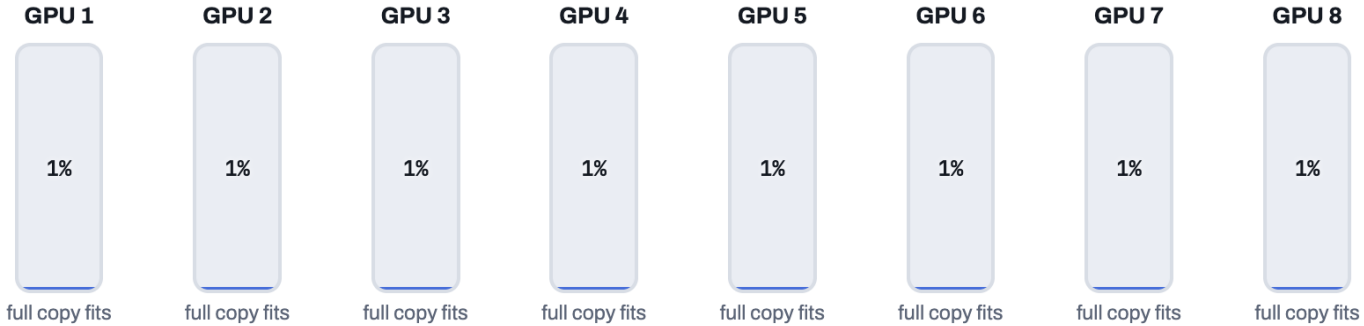
1 GPU | 8 GPUs

▶ Play

Weights of Llama 3.2 1B: 1.24B parameters × 2 bytes = 2.5 GB per GPU



dark line: what one GPU holds (192 GB)



Fits: each of the 8 GPUs holds its own full copy, so data parallelism works and runs about 8× faster.

Memory needed per copy `2.5 GB` Fill of each GPU `1%` If the model were split instead (Parts 3, 4, 6) `at least 1 GPU`

Parameters: Llama 3.2 1B 1.24B; Llama 4 Scout 109B total (17B active, 16 experts); Maverick 400B total (17B active, 128 experts); Behemoth about 2T total (288B active, 16 experts; announced, not released). Running stores 2, 1 or 0.5 bytes per parameter; training stores 16 (weights, gradients, Adam). Activations not included.

Data Parallelism

Llama 3.2 1B

2.5 GB

Does each GPU's copy fit?

With data parallelism, **every GPU must hold the whole model**. Pick a model, a job and the hardware, or press Play to step through the scenarios. Each tube is one GPU's memory; red means it overflows.

Llama 3.2 1B | Llama 4 Scout | Llama 4 Maverick | Llama 4 Behemoth

Run it | Train it

16-bit | 8-bit | 4-bit

H100 · 80 GB | B200 · 192 GB

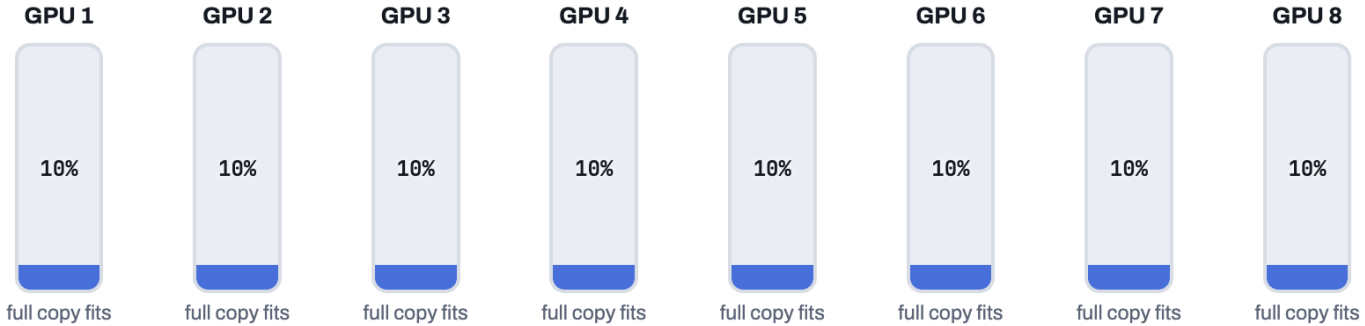
1 GPU | 8 GPUs

▶ Play

Training state of Llama 3.2 1B: 1.24B parameters × 16 bytes = 20 GB per GPU



dark line: what one GPU holds (192 GB)



Fits: each of the 8 GPUs holds its own full copy, so data parallelism works and runs about 8× faster.

Memory needed per copy `20 GB` Fill of each GPU `10%` If the model were split instead (Parts 3, 4, 6) `at least 1 GPU`

Parameters: Llama 3.2 1B 1.24B; Llama 4 Scout 109B total (17B active, 16 experts); Maverick 400B total (17B active, 128 experts); Behemoth about 2T total (288B active, 16 experts; announced, not released). Running stores 2, 1 or 0.5 bytes per parameter; training stores 16 (weights, gradients, Adam). Activations not included.

Data Parallelism

Llama 4 Scout

218 GB

Does each GPU's copy fit?

With data parallelism, **every GPU must hold the whole model**. Pick a model, a job and the hardware, or press Play to step through the scenarios. Each tube is one GPU's memory; red means it overflows.

Llama 3.2 1B | **Llama 4 Scout** | Llama 4 Maverick | Llama 4 Behemoth

Run it | Train it

16-bit | 8-bit | 4-bit

H100 · 80 GB | **B200 · 192 GB**

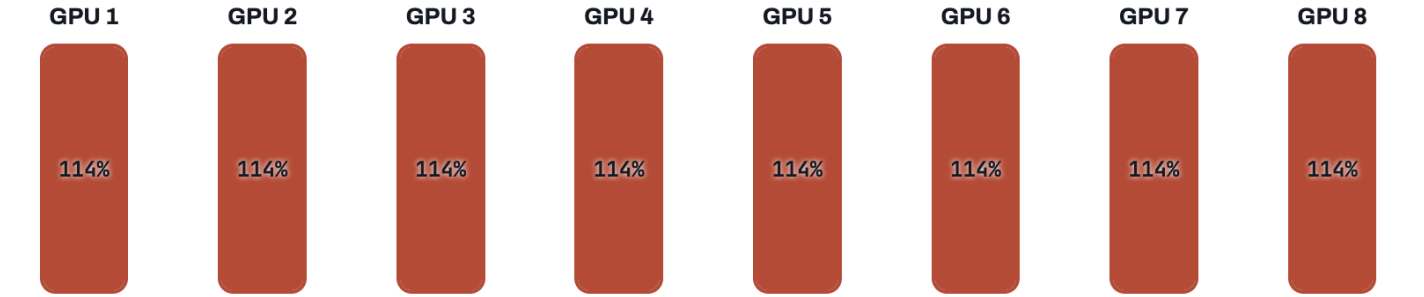
1 GPU | **8 GPUs**

▶ Play

Weights of Llama 4 Scout: 109B parameters × 2 bytes = 218 GB per GPU



dark line: what one GPU holds (192 GB)



Does not fit: one copy needs 218 GB but a GPU holds 192 GB. Adding more GPUs does not help, because data parallelism gives every GPU the same full copy.

Memory needed per copy **218 GB** Fill of each GPU **114%** If the model were split instead (Parts 3, 4, 6) **at least 2 GPUs**

Active per token **17B of 109B: compute is small, but memory must hold all experts**

Parameters: Llama 3.2 1B 1.24B; Llama 4 Scout 109B total (17B active, 16 experts); Maverick 400B total (17B active, 128 experts); Behemoth about 2T total (288B active, 16 experts; announced, not released). Running stores 2, 1 or 0.5 bytes per parameter; training stores 16 (weights, gradients, Adam). Activations not included.

Data Parallelism

Llama 4 Scout

218 GB

Does each GPU's copy fit?

With data parallelism, **every GPU must hold the whole model**. Pick a model, a job and the hardware, or press Play to step through the scenarios. Each tube is one GPU's memory; red means it overflows.

Llama 3.2 1B | **Llama 4 Scout** | Llama 4 Maverick | Llama 4 Behemoth

Run it | **Train it**

16-bit | 8-bit | 4-bit

H100 · 80 GB | **B200 · 192 GB**

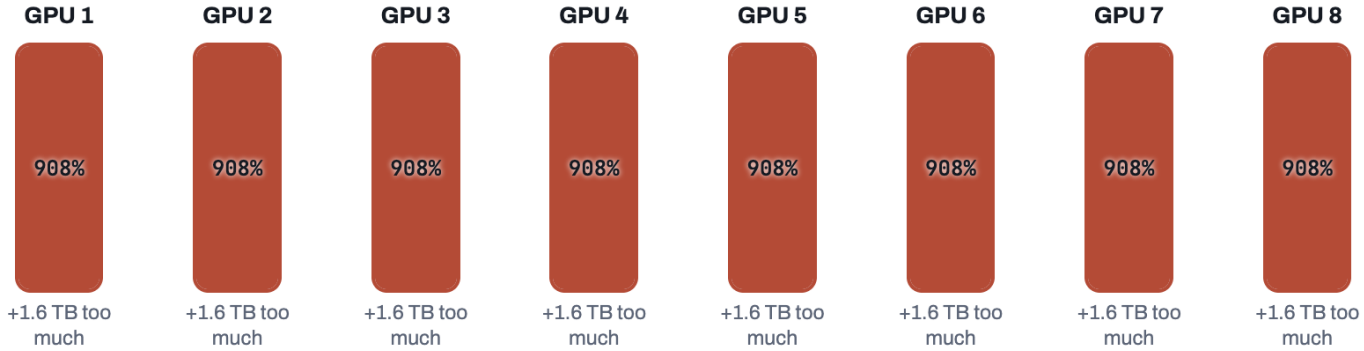
1 GPU | **8 GPUs**

▶ Play

Training state of Llama 4 Scout: 109B parameters × 16 bytes = 1.7 TB per GPU



dark line: what one GPU holds (192 GB)



Does not fit: one copy needs 1.7 TB but a GPU holds 192 GB. Adding more GPUs does not help, because data parallelism gives every GPU the same full copy.

Memory needed per copy `1.7 TB` Fill of each GPU `908%` If the model were split instead (Parts 3, 4, 6) `at least 10 GPUs`

Active per token `17B of 109B: compute is small, but memory must hold all experts`

Parameters: Llama 3.2 1B 1.24B; Llama 4 Scout 109B total (17B active, 16 experts); Maverick 400B total (17B active, 128 experts); Behemoth about 2T total (288B active, 16 experts; announced, not released). Running stores 2, 1 or 0.5 bytes per parameter; training stores 16 (weights, gradients, Adam). Activations not included.

Data Parallelism

EVERY GPU MUST HOLD THE WHOLE MODEL

Llama 4 **Maverick** has 400 billion parameters: at 2 bytes each, the weights alone are **800 GB**, and a B200 holds 192 GB.

Llama 4 **Behemoth** has about 2 trillion parameters: **4,000 GB** of weights. Training needs 8× more again (16 bytes per parameter).

Data parallelism alone cannot help, because it copies the model instead of splitting it.



Data Parallelism

EVERY GPU MUST HOLD THE WHOLE MODEL

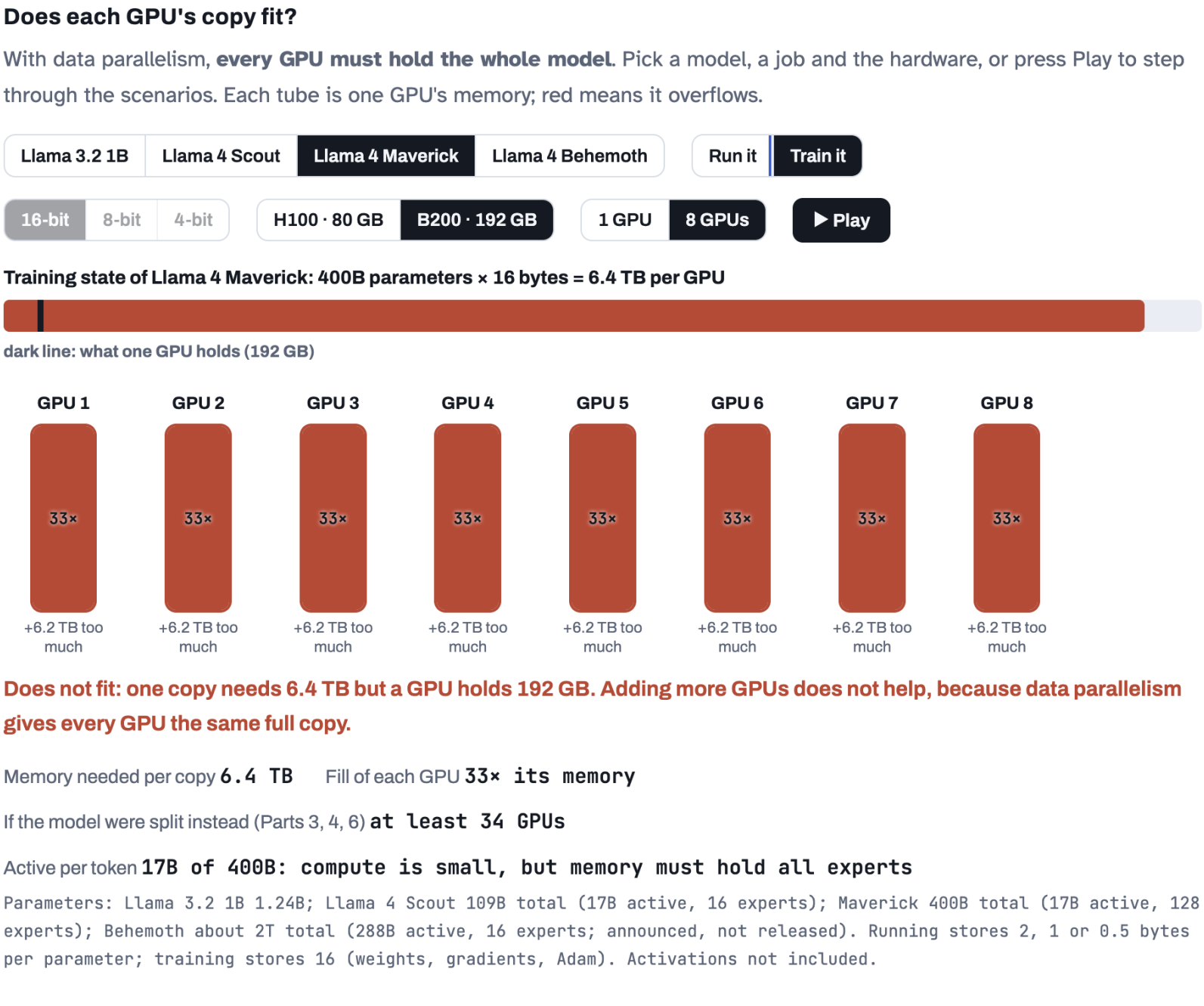
Llama 4 **Maverick** has 400 billion parameters: at 2 bytes each, the weights alone are **800 GB**, and a B200 holds 192 GB.

Llama 4 **Behemoth** has about 2 trillion parameters: **4,000 GB** of weights. Training needs 8× more again (16 bytes per parameter).

Data parallelism alone cannot help, because it copies the model instead of splitting it.

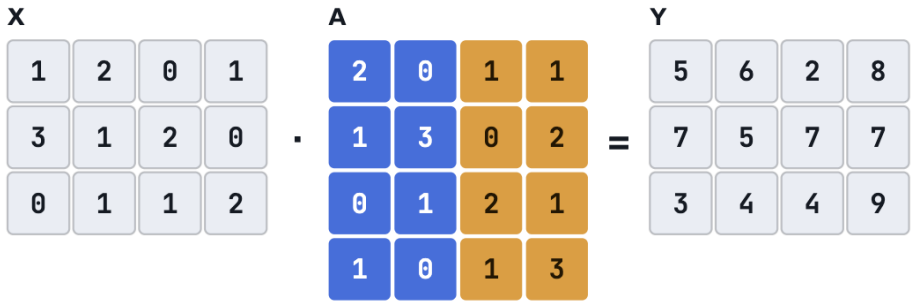
Solution:

split the model itself



Tensor Parallelism

1. Split each big matrix multiplication inside a layer across several GPUs.
2. Each GPU stores only its slice of the weight matrix and computes part of the answer
3. the GPUs then combine their parts.



The Backward pass is a Matrix Multiplication too

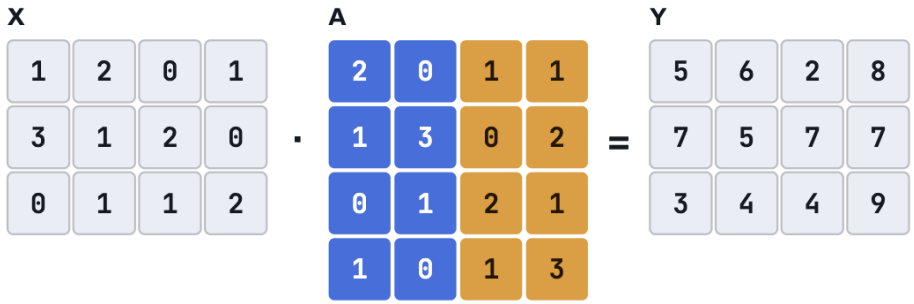
For one linear layer Y = X·W, let $G = \partial \ell / \partial Y$ be the gradient arriving from the layer above.

- $\partial \ell / \partial W = X^{\mathrm{T}} \cdot G$ (gradient for the weights, used by the optimizer)
- $\partial \ell / \partial X = G \cdot W^{\mathrm{T}}$ (gradient passed down to the layer below)

The backward pass is two matrix multiplications for every one in the forward pass, which is why it costs about twice as much.

Tensor Parallelism

1. Split each big matrix multiplication inside a layer across several GPUs.
2. Each GPU stores only its slice of the weight matrix and computes part of the answer
3. the GPUs then combine their parts.



The Backward pass is a Matrix Multiplication too

For one linear layer Y = X·W, let $G = \partial \ell / \partial Y$ be the gradient arriving from the layer above.

- $\partial \ell / \partial W = X^{\mathrm{T}} \cdot G$ (gradient for the weights, used by the optimizer)
- $\partial \ell / \partial X = G \cdot W^{\mathrm{T}}$ (gradient passed down to the layer below)

The backward pass is two matrix multiplications for every one in the forward pass, which is why it costs about twice as much.

Splitting happens at both the Forward and Backward passes. There are two ways to cut W across p GPUs:

- Column parallel
- Row Parallel

Tensor Parallelism

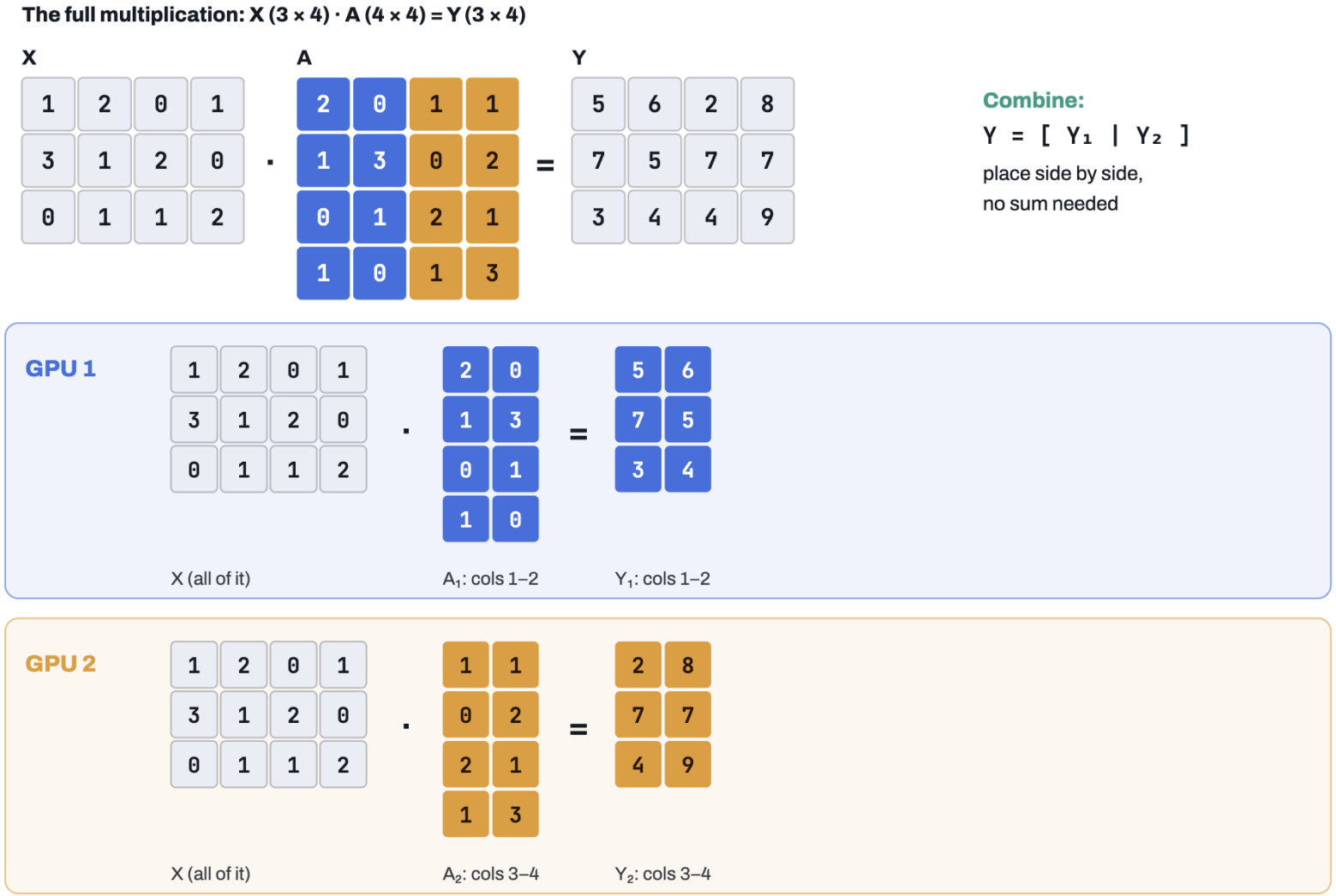
Column-parallel:

$W = [W_1, W_2, \ldots, W_p]$:

GPU i holds a block of columns $W_i \in \mathbb{R}^{D \times H/p}$

Each GPU needs all of X and computes the matching columns of Y: $Y_i = XW_i$

The pieces are placed side by side



Tensor Parallelism

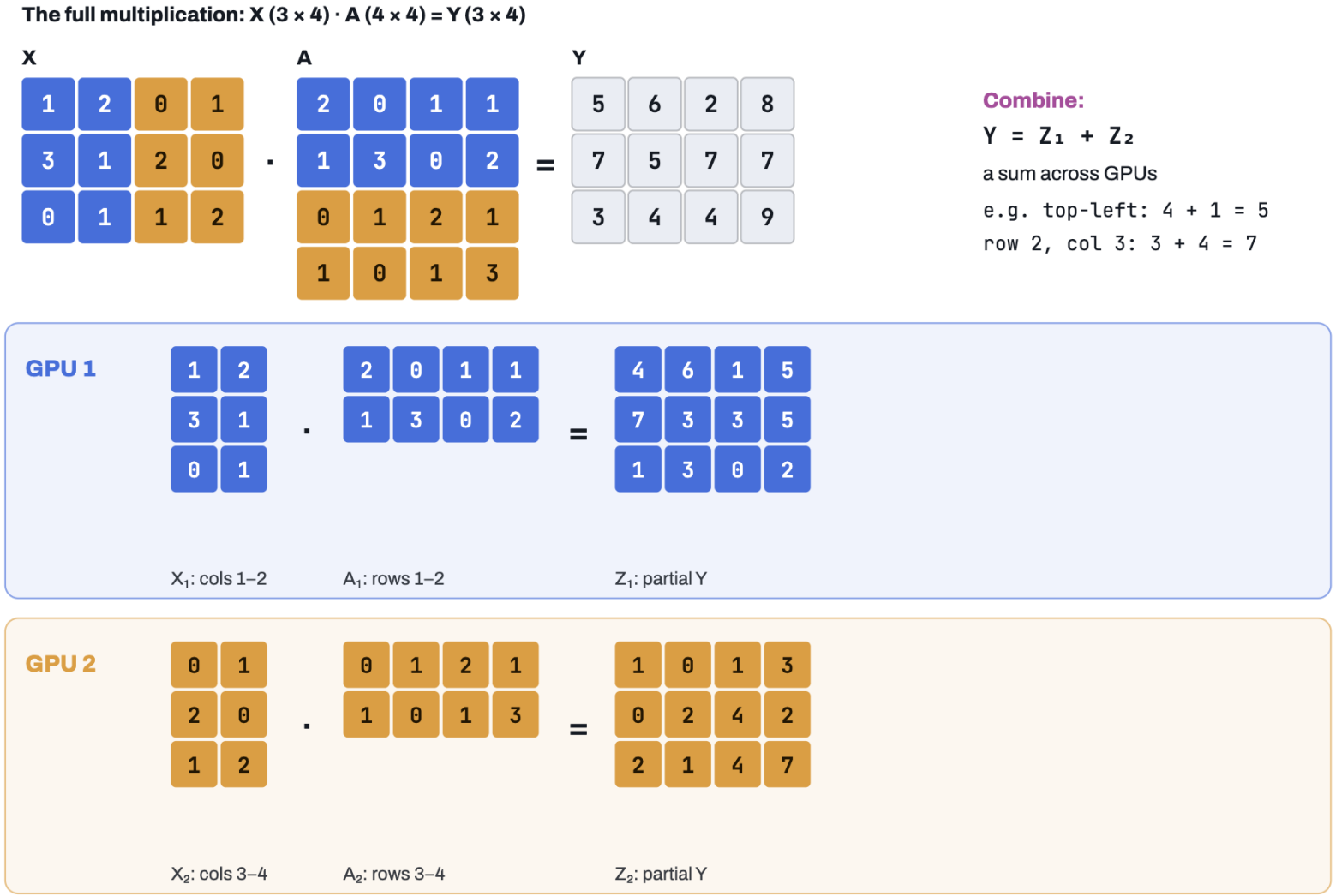
Row-parallel:

$W = [W_1; W_2; \ldots; W_p]$ (stacked)

GPU i holds a block of rows $W_i \in \mathbb{R}^{D/p \times H}$ and the matching columns X_i of X.

Each GPU computes a full size partial result $X_i W_i$

$Y = \Sigma_i X_i W_i$ must be **summed** across GPUs, which is an all-reduce.



Tensor Parallelism

Forward and Backward for each split

The sum signs mark where GPUs must communicate. Two observations make the method efficient:

1. The two splits are mirror images: column-parallel needs a sum only in the backward pass, row-parallel only in the forward pass.

2. A column-parallel layer produces Y already cut by columns, $[Y_1, \ldots, Y_p]$, which is exactly the input a row-parallel layer needs. So two consecutive layers, one column-parallel then one row-parallel, need only one sum in each direction.

	COLUMN-PARALLEL	**ROW-PARALLEL**
Forward $\boldsymbol{Y = XW}$	$XW = \left[XW_1, \ldots, XW_p\right]$	$XW = \Sigma_i X_i W_i$ (all-reduce)
Backward $\boldsymbol{\partial \ell / \partial X = GW^T}$	$GW^T = \Sigma_i G_i W_i{}^T$ (all-reduce)	$GW^T = \left[GW_1{}^T, \ldots, GW_p{}^T\right]$

Tensor Parallelism

Forward and Backward for each split

The sum signs mark where GPUs must communicate. Two observations make the method efficient:

1. The two splits are mirror images: column-parallel needs a sum only in the backward pass, row-parallel only in the forward pass.

2. A column-parallel layer produces Y already cut by columns, [Y_1, …, Y_p], which is exactly the input a row-parallel layer needs. So two consecutive layers, one column-parallel then one row-parallel, need only one sum in each direction.

	COLUMN-PARALLEL	**ROW-PARALLEL**
Forward $\boldsymbol{Y = XW}$	$XW = [XW_1, \dots, XW_p]$	$XW = \Sigma_i X_i W_i$ (all-reduce)
Backward $\boldsymbol{\partial \ell / \partial X = GW^T}$	$GW^T = \Sigma_i G_i W_i{}^T$ (all-reduce)	$GW^T = [GW_1{}^T, \dots, GW_p{}^T]$

The operators f and g

Operator	**Forward pass**	**Backward pass**	**Where**
f	identity: same X to every GPU	all-reduce the gradients	start of a block
g	all-reduce: sum partial outputs	identity: gradient to every GPU	end of a block

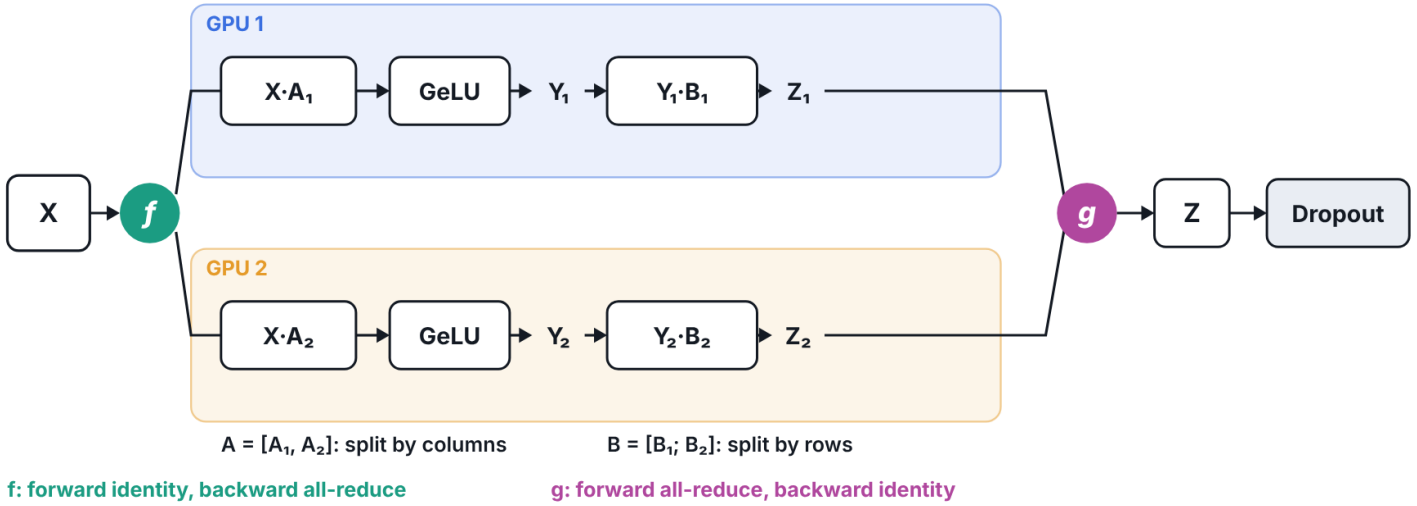
- **Block:** the MLP or the self-attention sub-layer each starts column-parallel and ends row-parallel

- **Why f sums backward:** X is used on every GPU, so $\partial \ell / \partial X = G_1 A_1{}^T + G_2 A_2{}^T$

- **Why g copies backward:** $Z = Z_1 + Z_2$, so $\partial \ell / \partial Z_i = \partial \ell / \partial Z$

Tensor Parallelism

Tensor Parallelism in MLP



The MLP computes two steps (Megatron's equations):

$$Y = GeLU(XA)$$
$$Z = Dropout(YB)$$

Step 1: split A by columns.
One could split A by rows, but then each GPU would hold a partial sum, and GeLU cannot be applied to partial sums:
$GeLU(X_1A_1 + X_2A_2) \neq GeLU(X_1A_1) + GeLU(X_2A_2)$,
because *GeLU is not linear*.

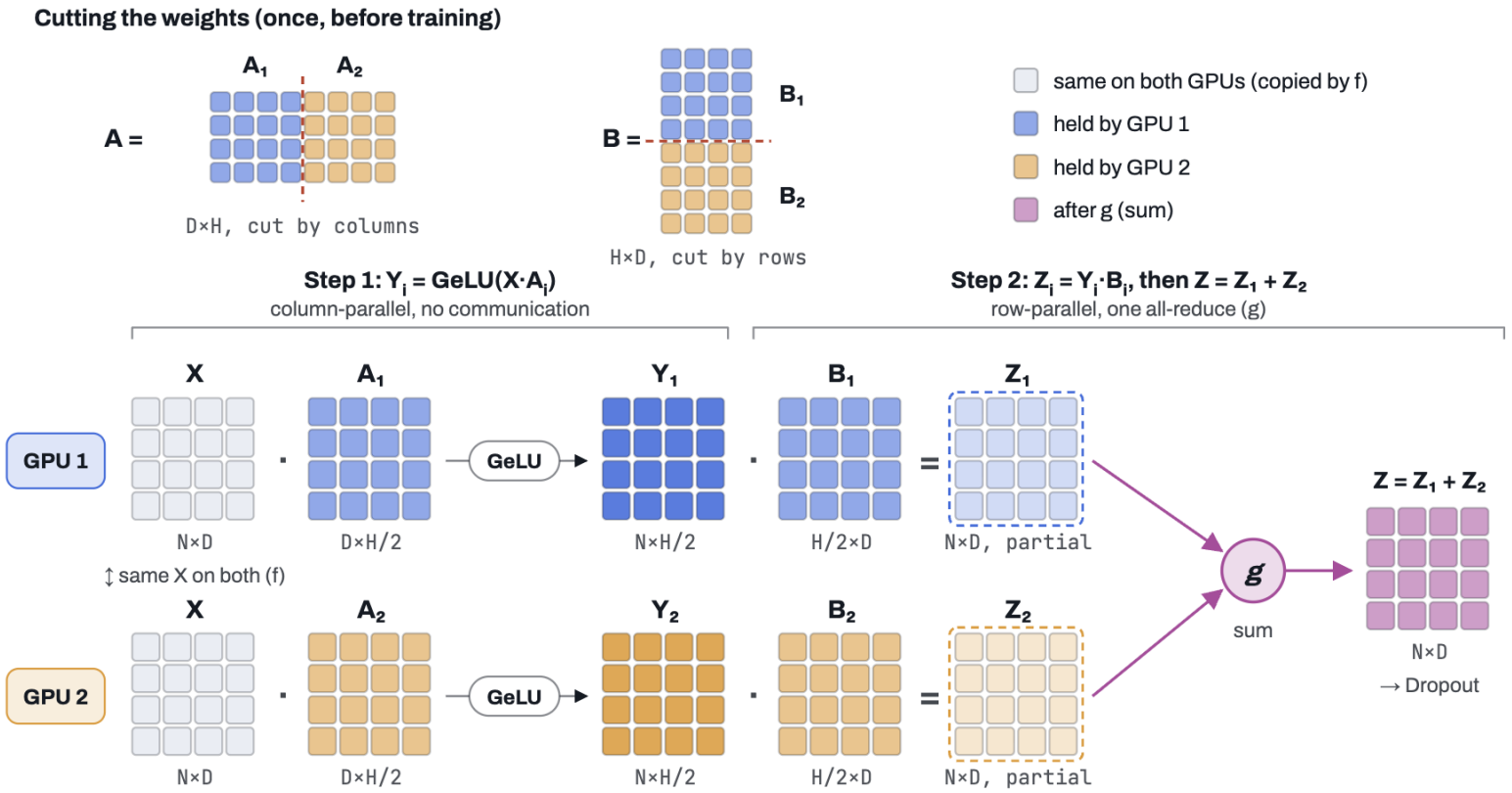
The GPUs would have to sum *before* GeLU. Splitting by columns, $A = [A_1, A_2]$, avoids this: each GPU gets whole columns, and GeLU acts on each number separately, so $[Y_1, Y_2] = [GeLU(XA_1), GeLU(XA_2)]$. (no communication)

Step 2: split B by rows.
Each GPU now holds Y_i, a block of Y's columns, which is exactly the input a row-parallel multiplication needs. With $B = [B_1; B_2]$: $Z = YB = Y_1B_1 + Y_2B_2$(one all-reduce: g)

Shoeybi et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019. arXiv:1909.08053

Tensor Parallelism

Tensor Parallelism in MLP



Step 1: split A by columns.
One could split A by rows, but then each GPU would hold a partial sum, and GeLU cannot be applied to partial sums:
$GeLU(X_1A_1 + X_2A_2) \neq GeLU(X_1A_1) + GeLU(X_2A_2)$,
because *GeLU is not linear*.

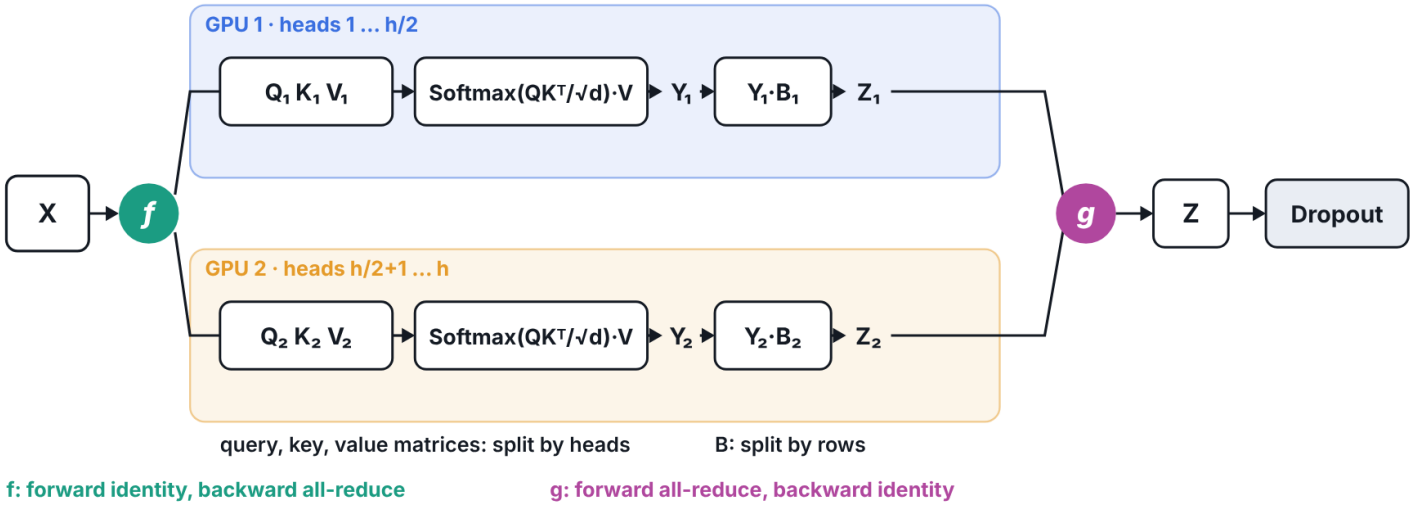
The GPUs would have to sum *before* GeLU. Splitting by columns, A = [A_1, A_2], avoids this: each GPU gets whole columns, and GeLU acts on each number separately, so $[Y_1, Y_2] = [GeLU(XA_1), GeLU(XA_2)]$. (no communication)

Step 2: split B by rows.
Each GPU now holds Y_i, a block of Y's columns, which is exactly the input a row-parallel multiplication needs. With B = [B_1; B_2]: $Z = YB = Y_1B_1 + Y_2B_2$(one all-reduce: g)

Shoeybi et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019. arXiv:1909.08053

Tensor Parallelism

Tensor Parallelism in Self-attention



Step 1: split by heads (column-parallel).
Heads never look at each other, so give each GPU a group of heads. GPU i holds the columns of W^Q, W^K and W^V that belong to its heads and computes Q_i, K_i, V_i and the full attention for those heads, with no communication. With 32 heads on 8 GPUs, each GPU computes 4 complete heads

Step 2: split the output projection B by rows.
GPU i's output Y_i is a block of Y's columns (its heads, side by side), exactly the input a row-parallel multiplication needs. So as in the MLP: split B by rows; $Z = Y_1B_1 + Y_2B_2$ via g

Self-attention with h heads first computes queries, keys and values, $Q = XW^Q$, $K = XW^K$, $V = XW^V$. Each head j uses only its own block of columns of Q, K and V:

$$head_j = Softmax\left(Q_j K_j^T/\sqrt{d_k}\right) \cdot V_j$$

$$Y = [head_1, \ldots, head_h]$$

Shoeybi et al. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. 2019. arXiv:1909.08053

Tensor Parallelism

A whole transformer layer is an attention block followed by an MLP block, so it needs **two all-reduces in the forward pass** (the two g's) and **two in the backward pass** (the two f's). Layer normalization, residual connections and dropout are cheap, so Megatron simply computes them on every GPU

- **EXAMPLE · LLAMA 3 70B ON 8 GPUS IN ONE SERVER**The weights are 141 GB; with tensor parallelism across 8 GPUs, each GPU stores about **17.6 GB**.
- Every layer needs 2 sums in the forward pass and 2 in the backward pass. For 4,096 tokens, each sum moves an activation tensor of 4,096 × 8,192 numbers × 2 bytes ≈ 67 MB, taking about 0.3 ms over NVLink.
- With 80 layers that is 320 sums per micro-batch, about 80 ms, against roughly half a second of computation.

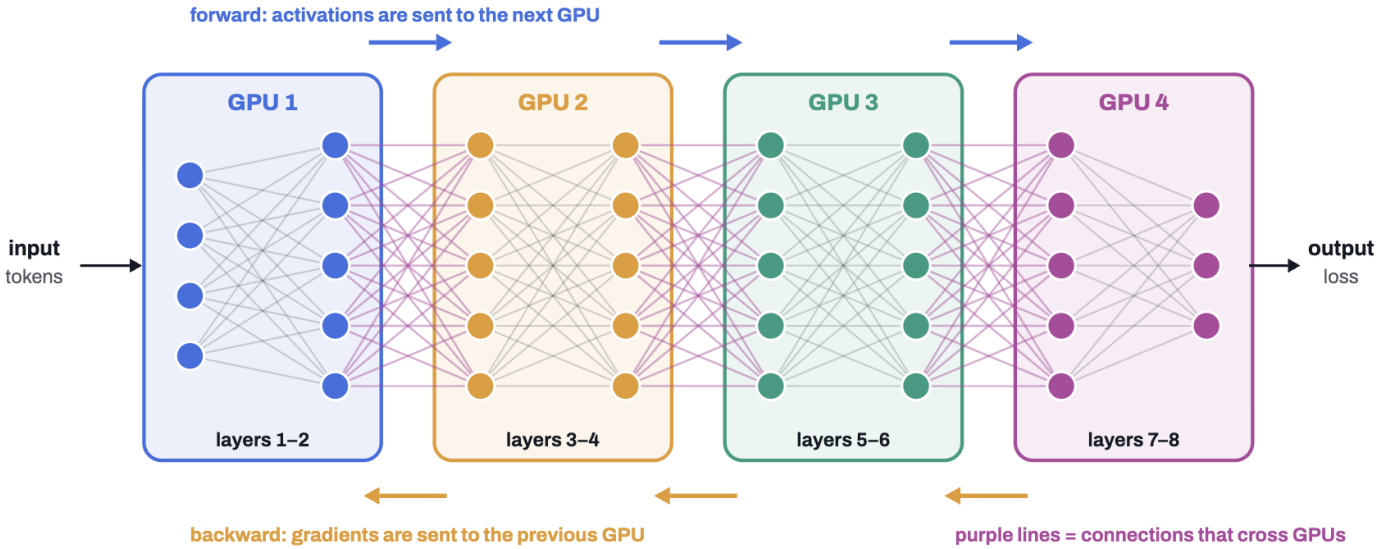
Communication in every layer
Tensor parallelism talks inside every layer, so it only works on the fast NVLink connections within one server. Over the network, the same sums would take about 9× longer, and the GPUs would spend more time waiting than computing. In practice the tensor-parallel group is at most the 8 GPUs of one server.

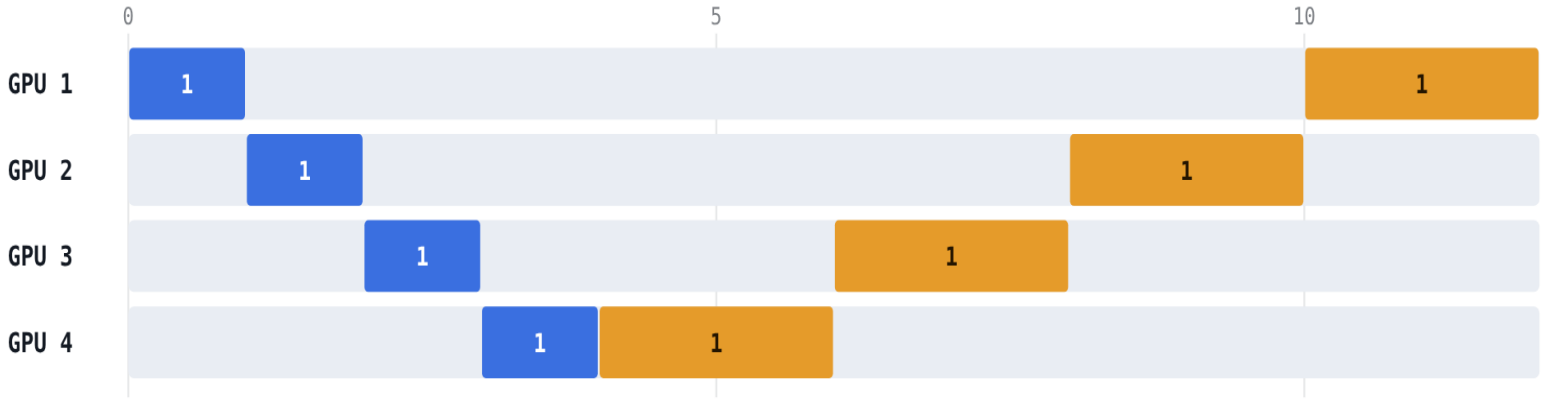
Pipeline Parallelism

Turn the model into an assembly line.
Each GPU owns a group of consecutive layers and passes its output to the next GPU.
The challenge is keeping every station busy.

The problem: layers run one after another

Layer 2 cannot start until layer 1 has finished. If GPU 1 holds layers 1–20 and GPU 2 holds layers 21–40, GPU 2 waits while GPU 1 works, and then GPU 1 waits while the backward pass comes back.





Bubble

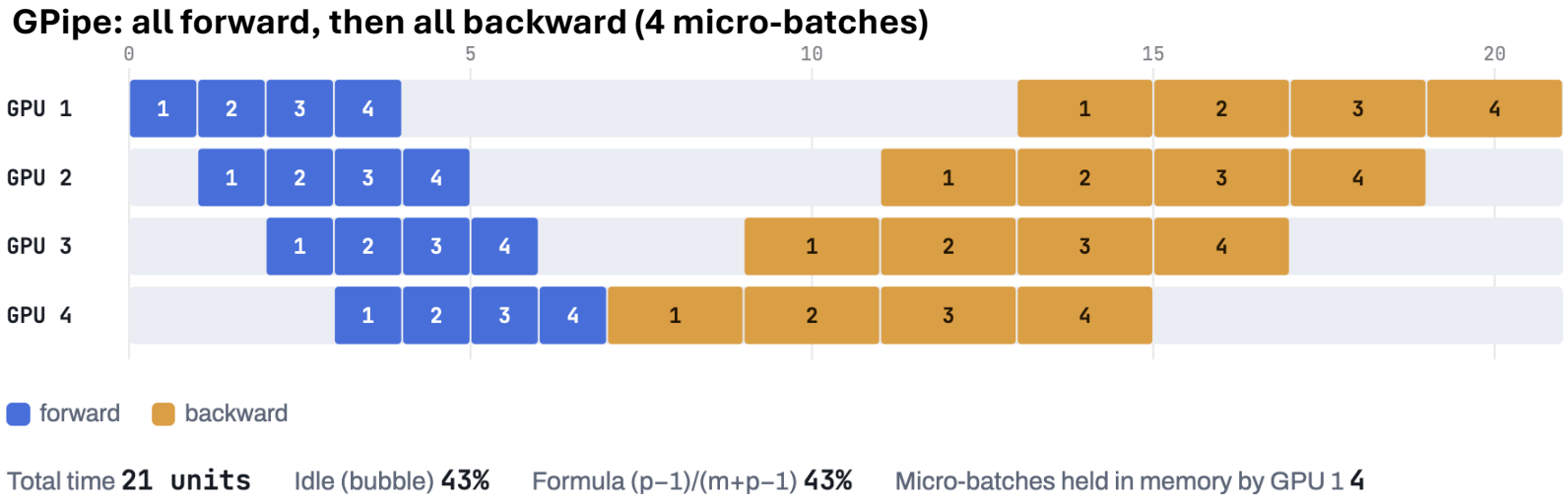
Idle time in a pipeline while stages wait for work. Shrinking it is the goal of every pipeline schedule

With 4 GPUs and one batch, each GPU is idle about 75%

Pipeline Parallelism

How to fix?

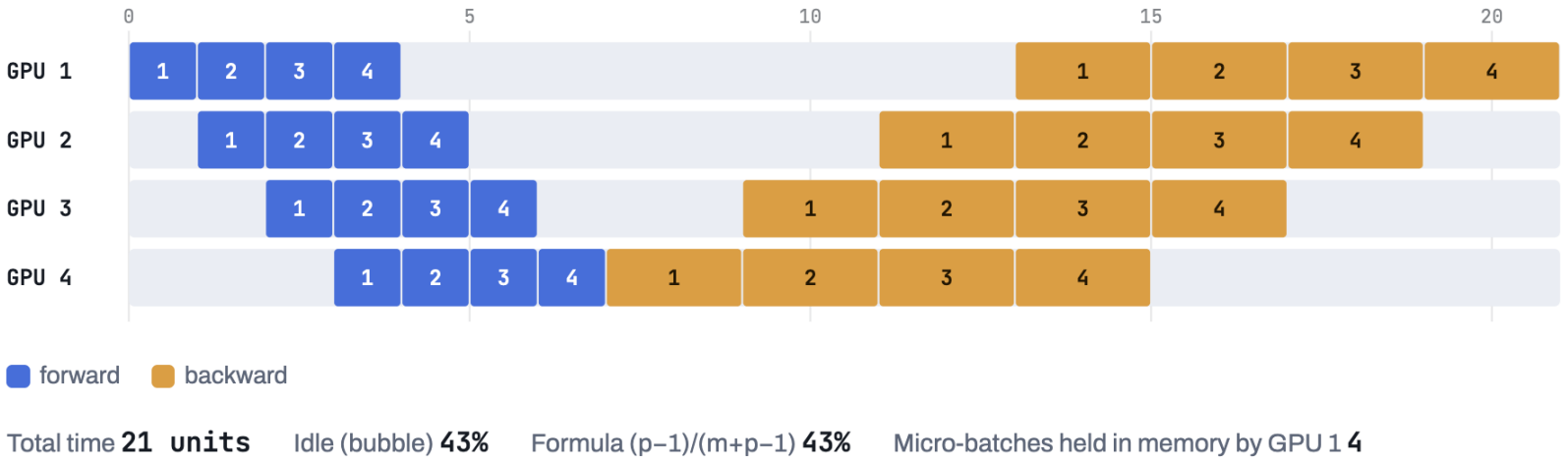
Gpipe:
Cut the batch into micro-batches. As soon as GPU 1 finishes micro-batch 1 and passes it on, it starts micro-batch 2, so several stages work at once .



Pipeline Parallelism

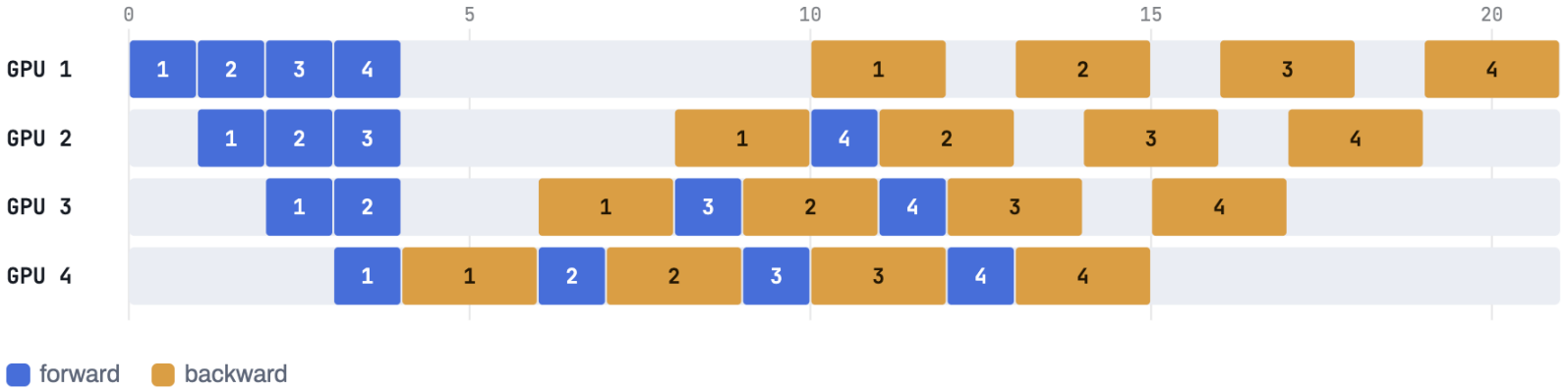
How to fix?

Gpipe:
Cut the batch into micro-batches. As soon as GPU 1 finishes micro-batch 1 and passes it on, it starts micro-batch 2, so several stages work at once .



Total time **21 units** Idle (bubble) **43%** Formula (p−1)/(m+p−1) **43%** Micro-batches held in memory by GPU 1 **4**

1F1B ("one forward, one backward"):
In the steady state each GPU alternates one forward and one backward pass. The bubble is the same size



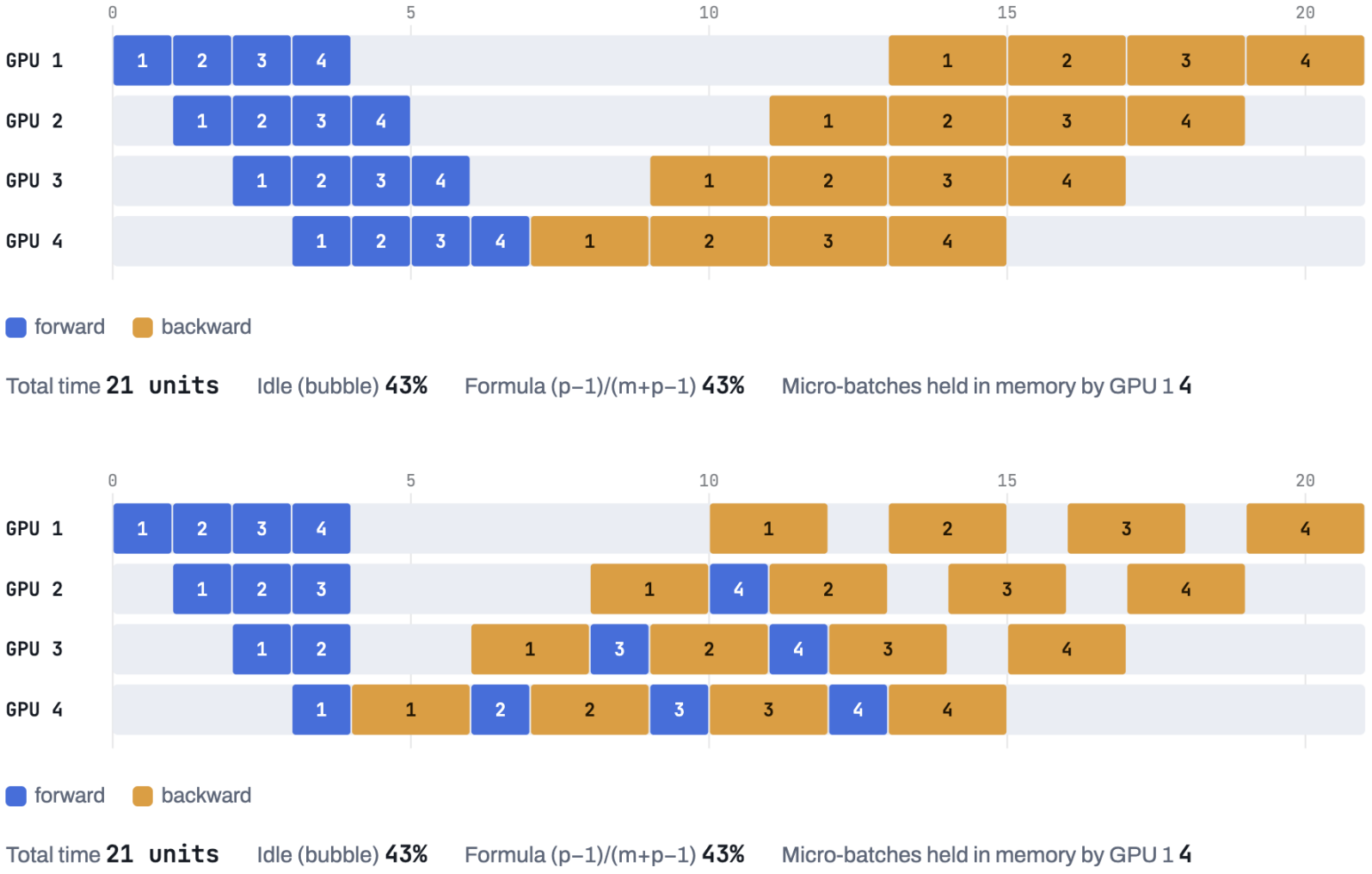
Total time **21 units** Idle (bubble) **43%** Formula (p−1)/(m+p−1) **43%** Micro-batches held in memory by GPU 1 **4**

Pipeline Parallelism

How to fix?

What is the advantage of 1F1B over GPipe?

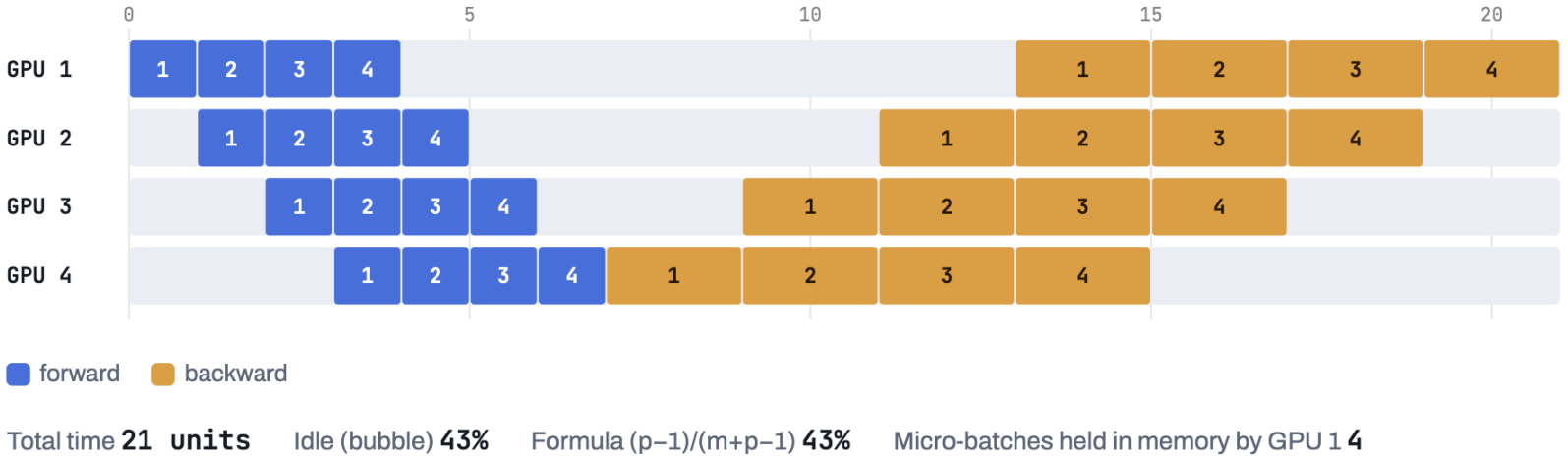
A) less idle time
B) less memory utilization
C) Shorter processing time
D) All of the above



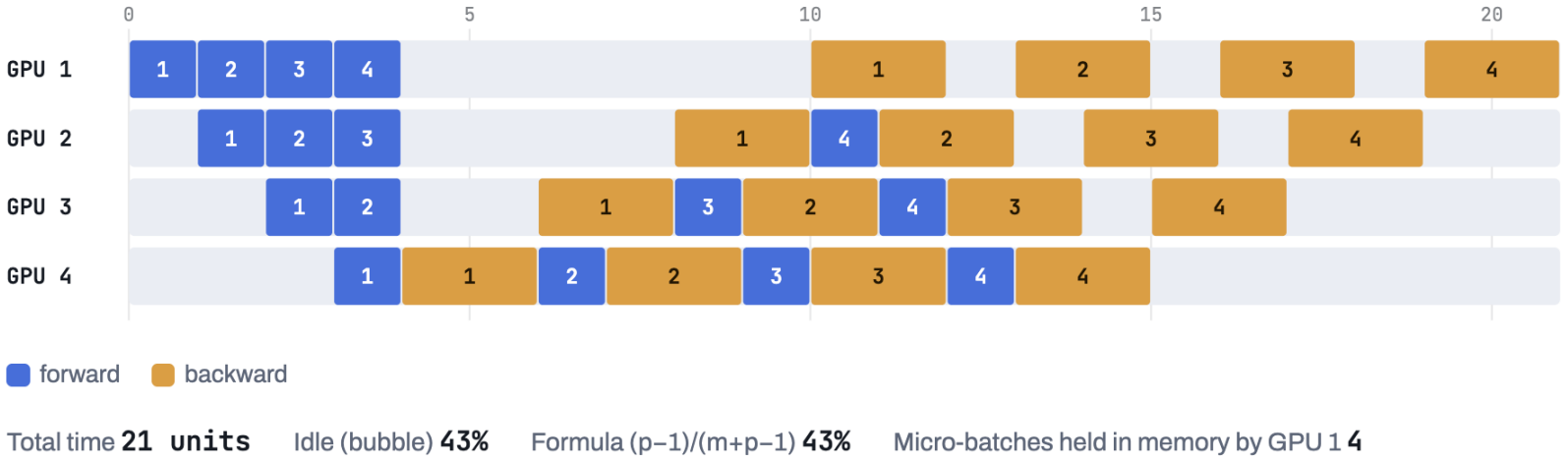
Pipeline Parallelism

How to fix?

Gpipe:
Cut the batch into micro-batches. As soon as GPU 1 finishes micro-batch 1 and passes it on, it starts micro-batch 2, so several stages work at once .



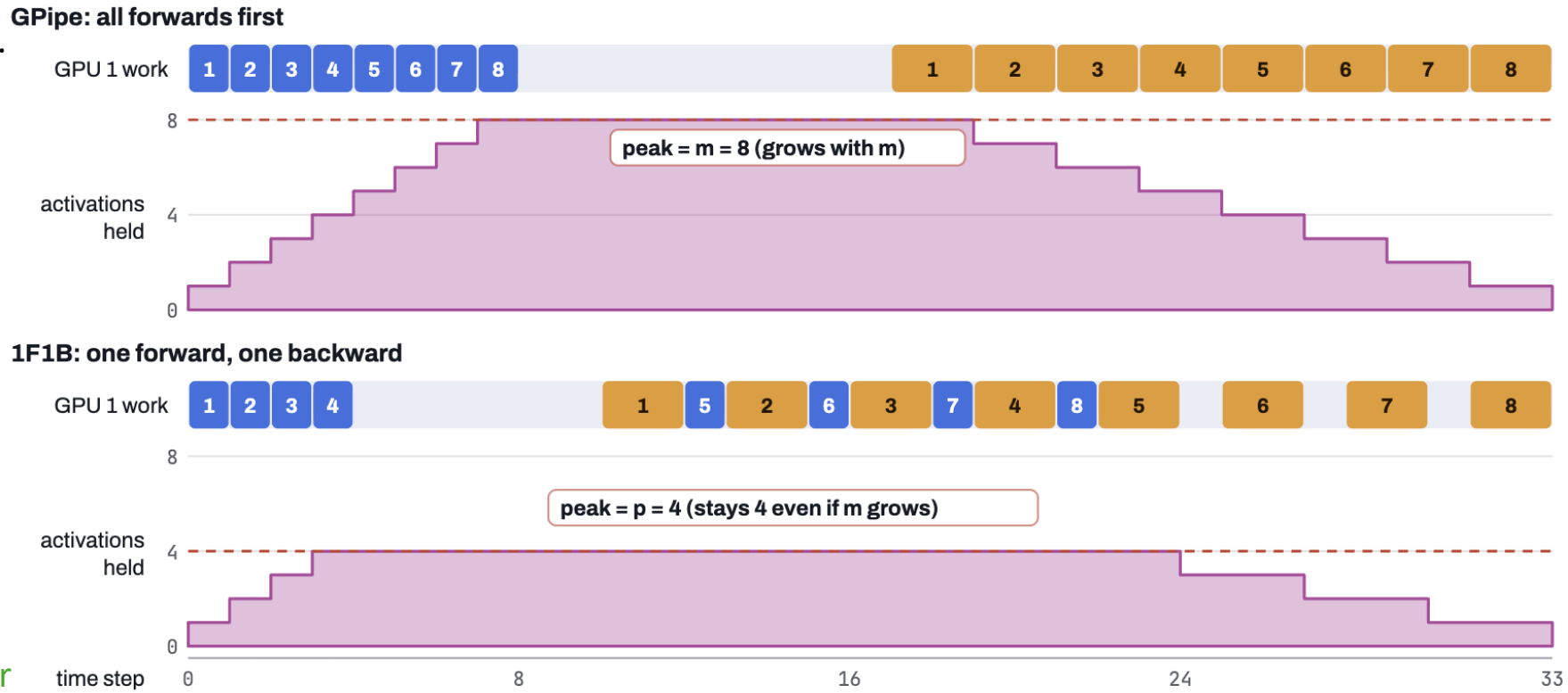
1F1B ("one forward, one backward"):
In the steady state each GPU alternates one forward and one backward pass. The bubble is the same size, but each GPU holds far fewer micro-batches' activations in memory at once.



Pipeline Parallelism

How to fix?

Gpipe:
Cut the batch into micro-batches. As soon as GPU 1 finishes micro-batch 1 and passes it on, it starts micro-batch 2, so several stages work at once .



1F1B ("one forward, one backward"):
In the steady state each GPU alternates one forward and one backward pass. The bubble is the same size, but each GPU holds far fewer micro-batches' activations in memory at once.

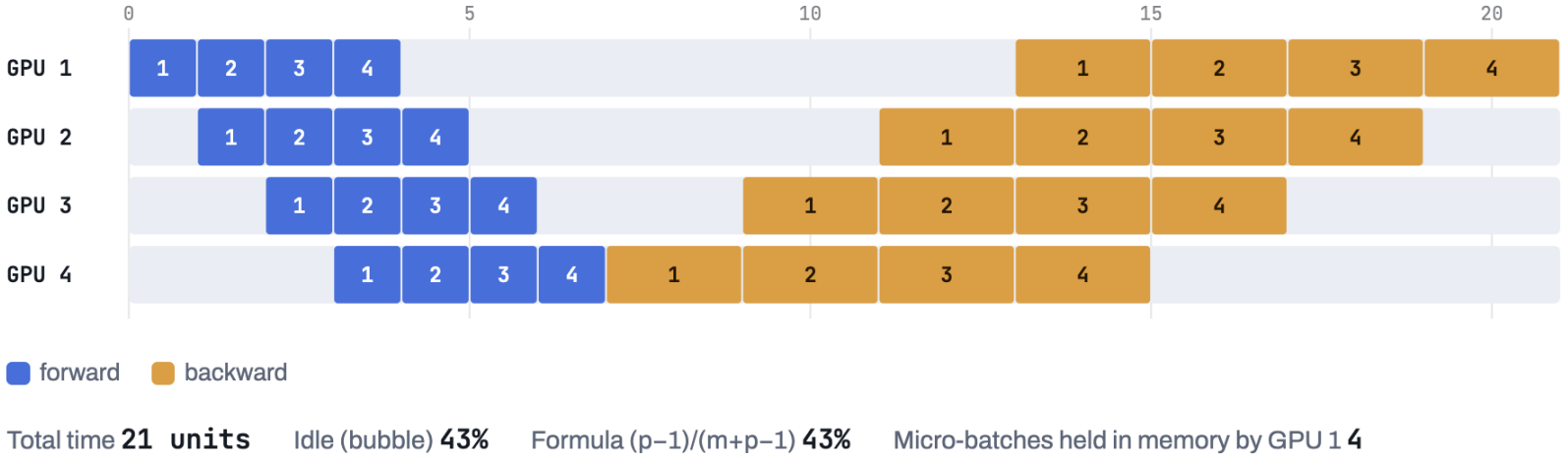
Pipeline Parallelism

How to fix?

Why less memory with 1F1B?

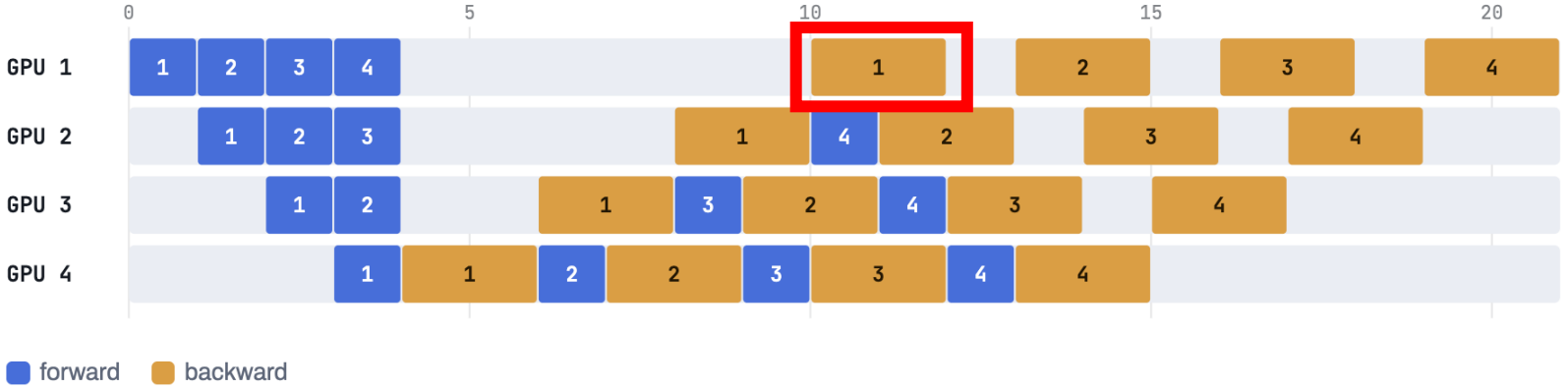
A GPU must keep a micro-batch's activations from its forward pass until its backward pass on that GPU, which then frees them. GPipe runs all forward passes first, so GPU 1 holds the activations of every micro-batch at once: *m* of them. In 1F1B, GPU 1 runs only *p* forward passes before its first backward arrives; after that, every new forward is paired with a backward that frees one micro-batch, so the count never rises above *p*.

GPipe: all forward, then all backward (4 micro-batches)



Total time **21 units** Idle (bubble) **43%** Formula (p−1)/(m+p−1) **43%** Micro-batches held in memory by GPU 1 **4**

1F1B: one forward, one backward (4 micro-batches)



Total time **21 units** Idle (bubble) **43%** Formula (p−1)/(m+p−1) **43%** Micro-batches held in memory by GPU 1 **4**

Narayanan et al. PipeDream: Generalized Pipeline Parallelism for DNN Training. SOSP 2019. arXiv:1806.03377

Pipeline Parallelism

How to fix?

Why the same bubble?

1F1B only reorders the work in the middle. The pipeline still has to fill up at the start (the last GPU waits $p - 1$ slots for its first micro-batch) and drain at the end (the first GPU waits while the last backward passes travel back), so the idle time is the same $(p - 1)/(m + p - 1)$.



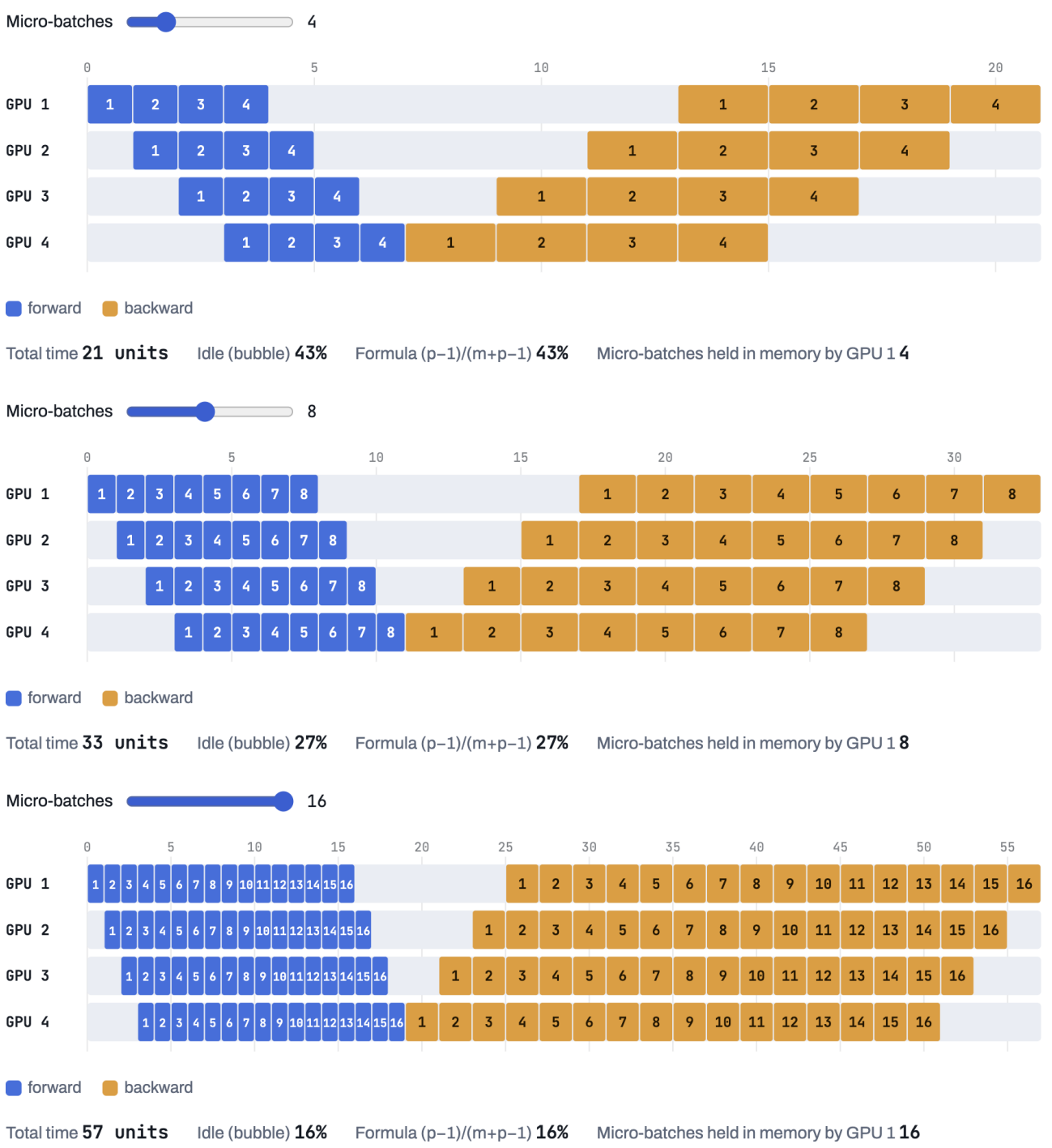
Narayanan et al. PipeDream: Generalized Pipeline Parallelism for DNN Training. SOSP 2019. arXiv:1806.03377

Pipeline Parallelism

$$\text{bubble} = (p-1)/(m+p-1)$$

- p = pipeline GPUs
- m = micro-batches

More micro-batches → smaller bubble



Pipeline Parallelism
Weight Stashing

PipeDream

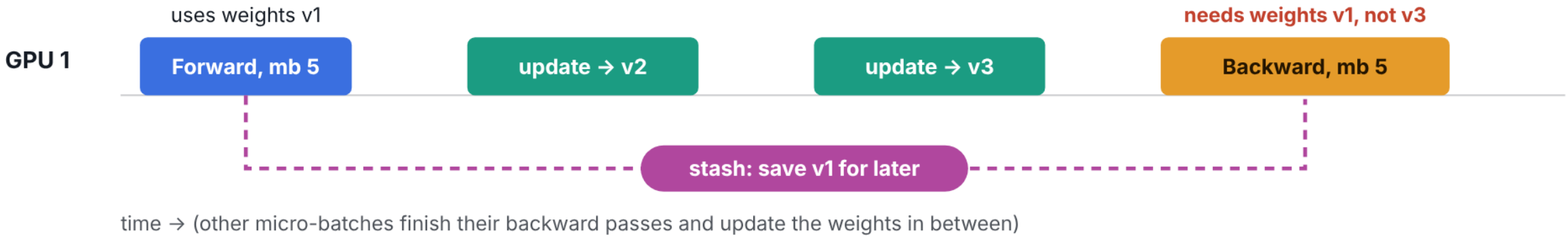
The original PipeDream system updated the weights after every backward pass, without waiting for the whole batch.

Problem: by the time a micro-batch's backward pass reaches GPU 1, the weights have changed since its forward pass, so the gradient is computed with the wrong weights.

Fix: **weight stashing**: save the weights used in each forward pass and reload them for the matching backward pass.

Pipeline flush (Megatron)

Megatron-LM's version avoids the issue differently: it **flushes** the pipeline at the end of every batch, updating weights only once all micro-batches are done. That costs a bubble per batch but keeps the math exact.



Narayanan et al. PipeDream: Generalized Pipeline Parallelism for DNN Training. SOSP 2019. arXiv:1806.03377

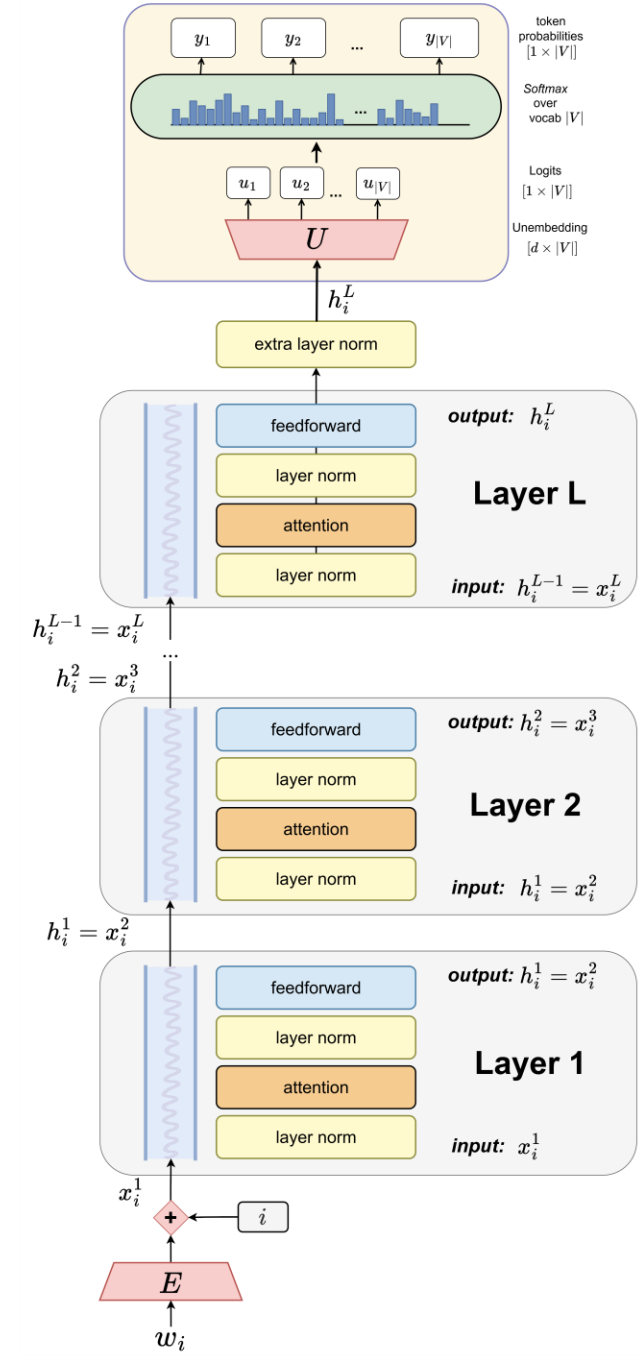
Pipeline Parallelism

How should the layers be divided?

The pipeline runs at the speed of its slowest stage, and layers are not all equally expensive.

Measure how long each layer takes, then use **dynamic programming** to choose the split points that make the slowest stage as fast as possible.

Pipelines waste time in the bubble, and shrinking it needs many micro-batches, which means a large batch. Stages must also be balanced carefully.



Optimizer Parallelism

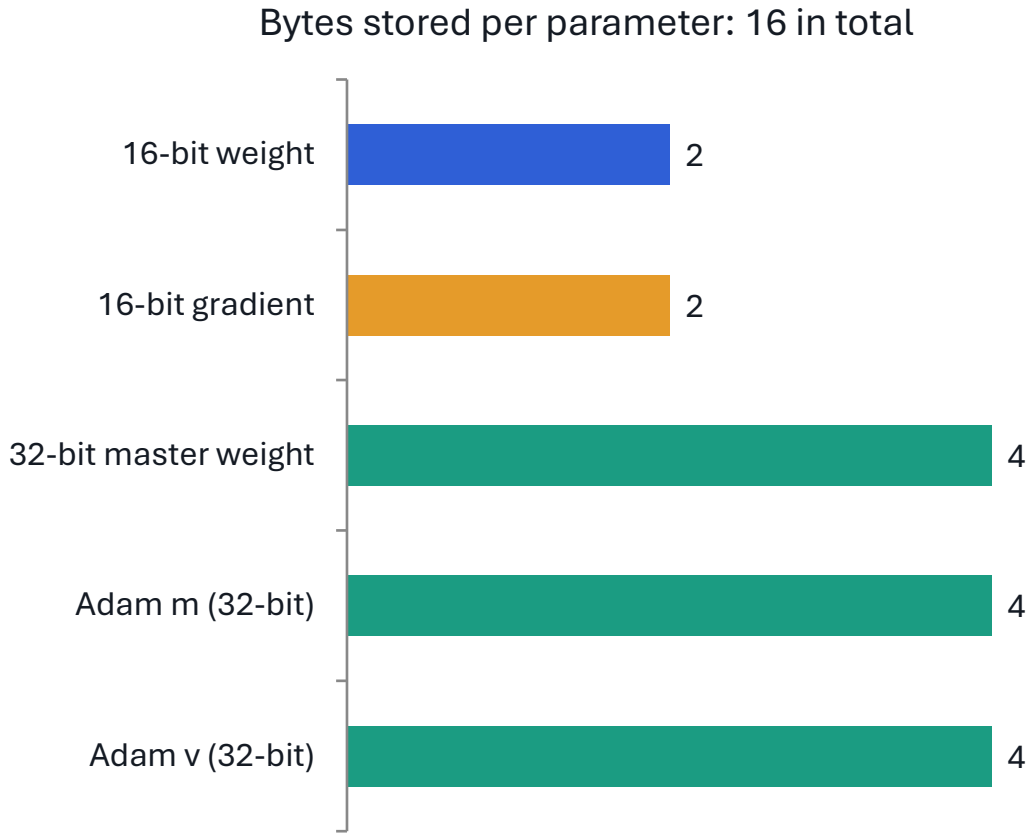
Data parallelism stores identical copies of the optimizer's extra numbers on every GPU, which wastes most of the memory.

Why the optimizer needs so much memory?

Plain gradient descent (SGD) only needs the weights and the gradients: $\theta \leftarrow \theta - \eta \cdot \nabla \ell$
Adam also keeps two running averages for every parameter:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t \odot g_t$$
$$\theta \leftarrow \theta - \eta \cdot \hat{m}_t / \sqrt{\hat{v}_t}$$

m: running average of gradients (momentum)
v: running average of squared gradients
Mixed precision: compute in 16-bit, keep m, v and master weights in 32-bit



7.5B parameters × 16 bytes = 120 GB per GPU
90 GB of it optimizer state

The **master weight** is a full-precision (32-bit) copy of each weight kept by the optimizer. The forward and backward passes use a 16-bit copy for speed, but **the update is applied to the 32-bit maste**r and then rounded back to 16-bit, so that tiny updates such as 0.0001 are not lost to rounding.

Optimizer Parallelism

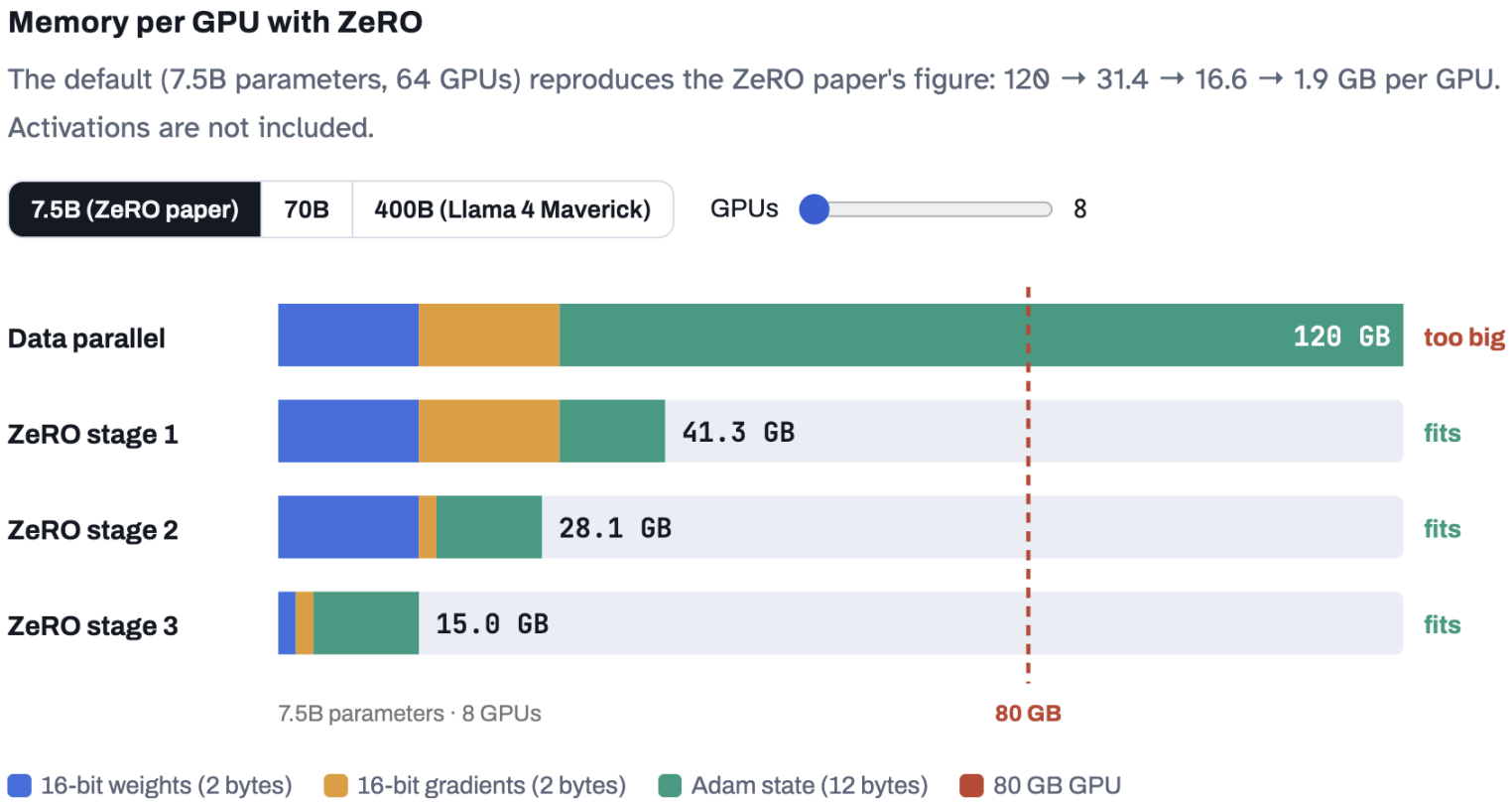
ZeRO gives each GPU only a slice of them, then also of the gradients, and finally of the weights.

With N data-parallel GPUs, ZeRO (Zero Redundancy Optimizer) removes the duplicates in three stages:

Stage 1: shard the optimizer state. Each GPU keeps 1/N of the 12 bytes and updates only its slice of the weights, then shares the updated weights.

Stage 2: also shard the gradients. Each GPU only needs the summed gradient for its own slice.

Stage 3: also shard the weights. Each layer's weights are gathered just before use and freed afterwards. This costs about 1.5× the communication of plain data parallelism.



Stage 3 saves the most memory but sends the most data, because every layer's weights must be collected before each use. It works best when there is enough computation to hide that communication.

Rajbhandari et al. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. SC 2020. arXiv:1910.02054

Optimizer Parallelism

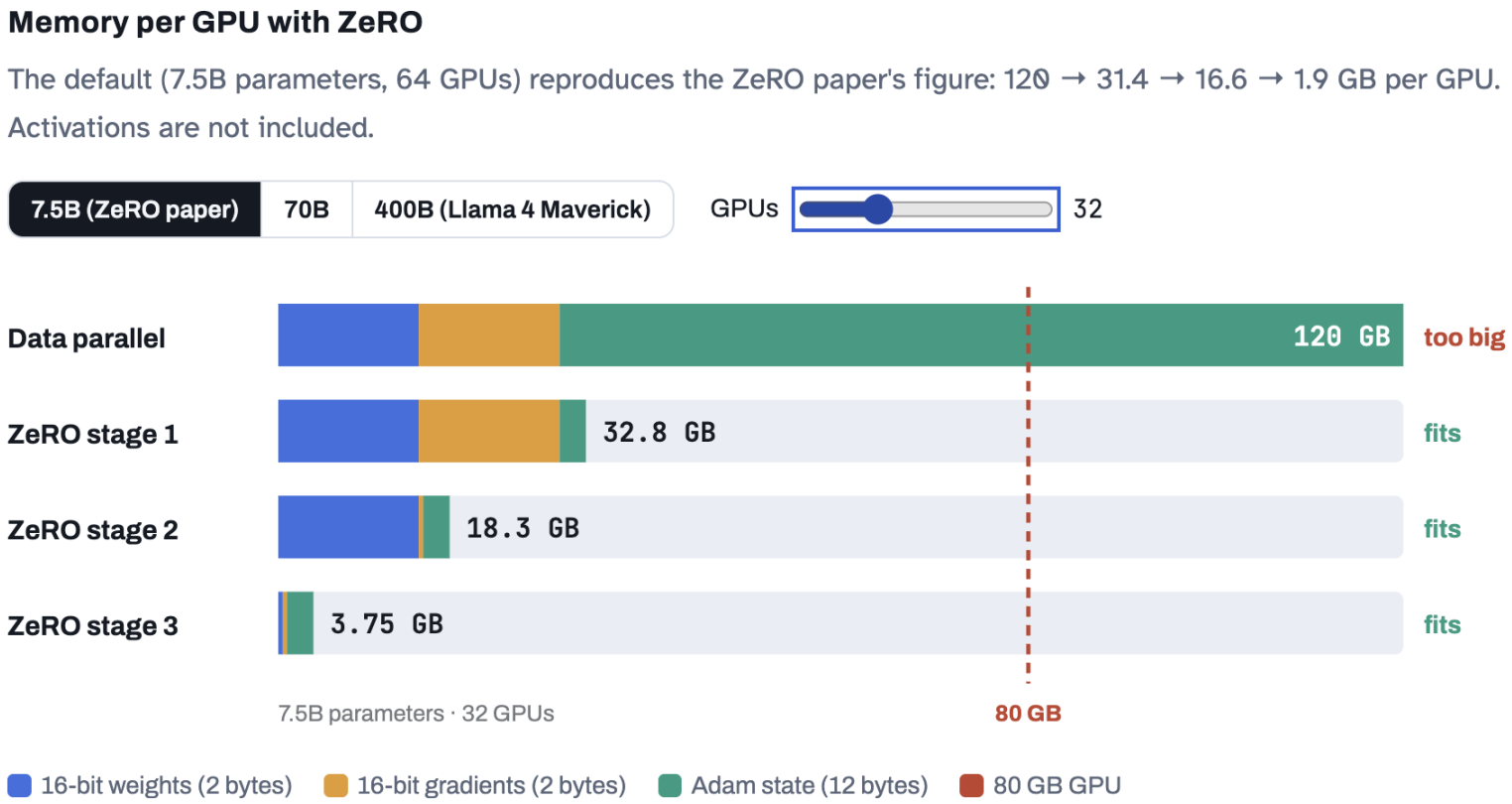
ZeRO gives each GPU only a slice of them, then also of the gradients, and finally of the weights.

With N data-parallel GPUs, ZeRO (Zero Redundancy Optimizer) removes the duplicates in three stages:

Stage 1: shard the optimizer state. Each GPU keeps 1/N of the 12 bytes and updates only its slice of the weights, then shares the updated weights.

Stage 2: also shard the gradients. Each GPU only needs the summed gradient for its own slice.

Stage 3: also shard the weights. Each layer's weights are gathered just before use and freed afterwards. This costs about 1.5× the communication of plain data parallelism.

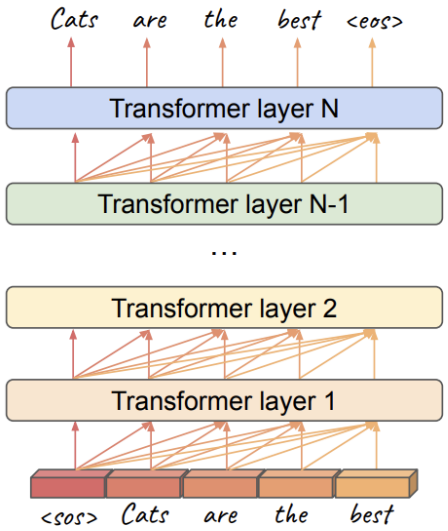


Stage 3 saves the most memory but sends the most data, because every layer's weights must be collected before each use. It works best when there is enough computation to hide that communication.

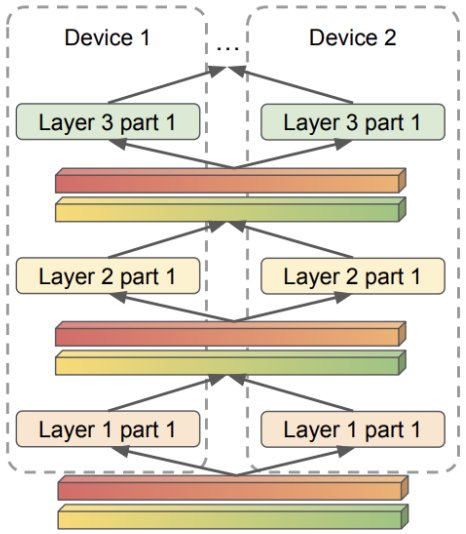
Rajbhandari et al. ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. SC 2020. arXiv:1910.02054

Token Parallelism

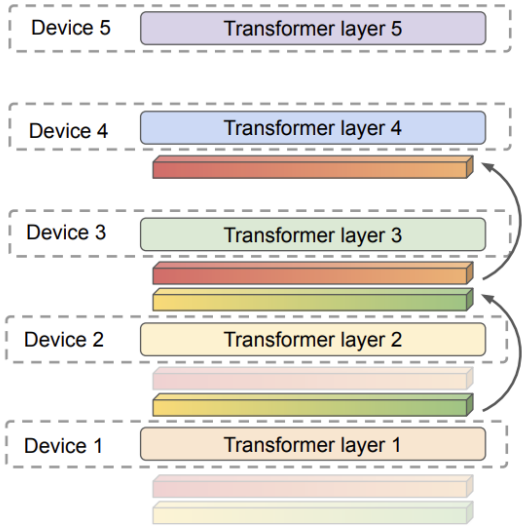
In a pipeline, don't wait for a whole sequence to finish a stage. Because of causal attention, the next stage can start on the first tokens while the current stage is still working on later ones



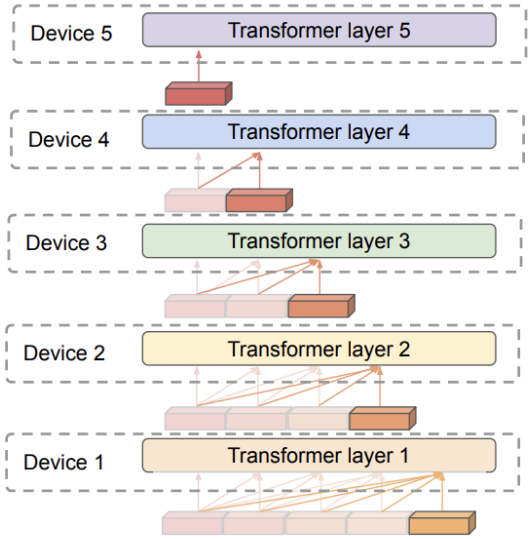
(a) Transformer-based LM



(b) Operation partitioning (Megatron-LM)



(c) Microbatch-based pipeline parallelism (GPipe)



(d) Token-based pipeline parallelism (TeraPipe)

Li et al. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. ICML 2021. arXiv:2102.07988

Token Parallelism

Token t only depends on tokens 1 to t. So in layer l, the work for token t can start as soon as tokens 1 to t are done in layer $l - 1$.

TeraPipe cuts each sequence into chunks of tokens and pipelines the *chunks* across the stages. Smaller pieces mean less idle time, and the gain grows with sequence length.



If one stage takes 4 time units for the whole sequence, the forward pass takes 4 × 4 = 16 units and each **GPU is idle 75% of the time**. Cut into 8 equal chunks of 0.5 units, the last chunk leaves the last stage after (8 + 3) × 0.5 = 5.5 units

Li et al. TeraPipe: Token-Level Pipeline Parallelism for Training Large-Scale Language Models. ICML 2021. arXiv:2102.07988

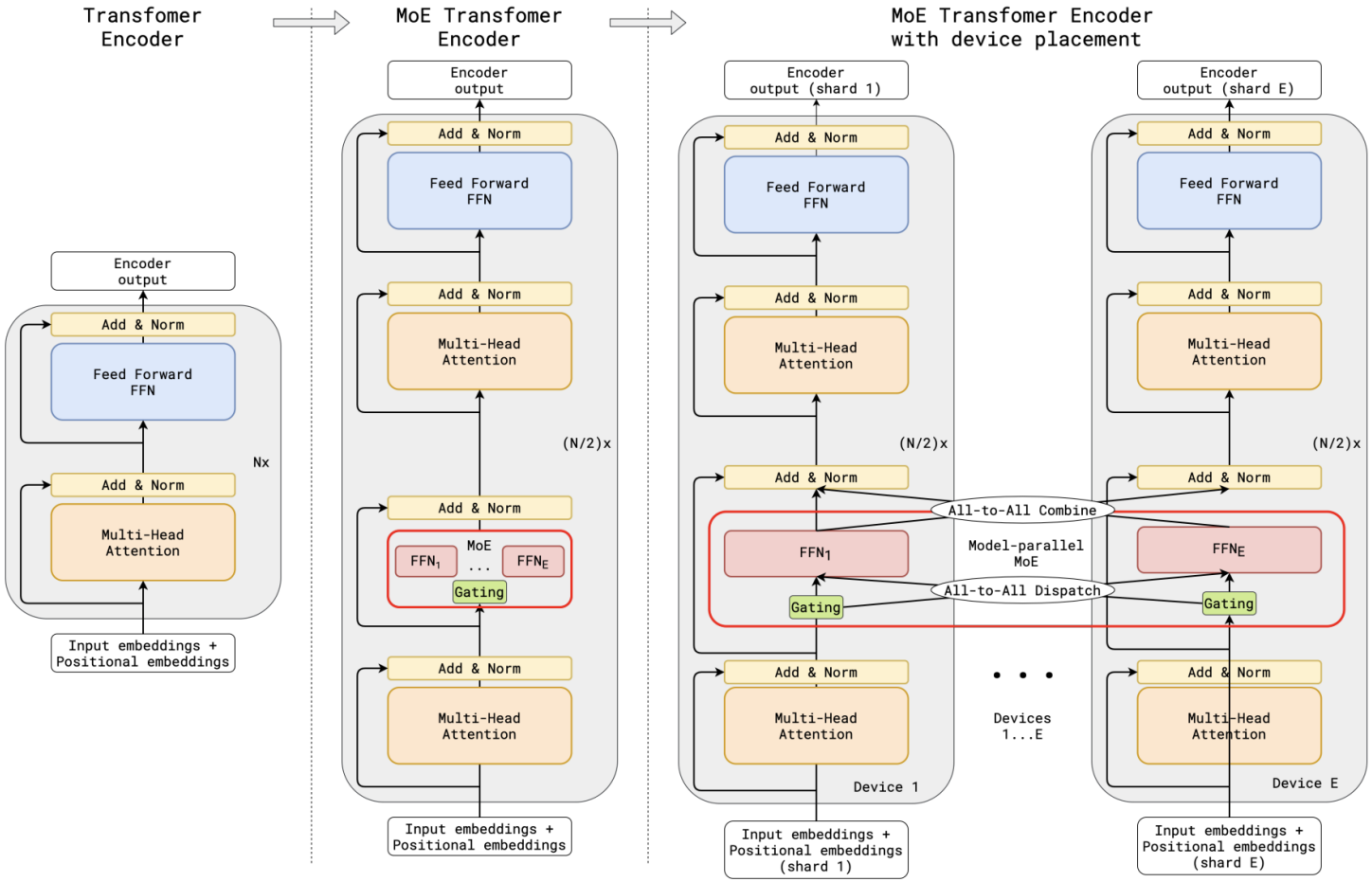
Expert Parallelism

In a **mixture-of-experts** (MoE) model, the single MLP in a layer is replaced by many smaller MLPs called **experts**.

A small network called the **router** looks at each token and sends it to just one or two experts.

The model can have a huge number of parameters while each token only uses a few of them.

Llama 4 Maverick, for example, has 400 billion parameters but uses only 17 billion for each token.

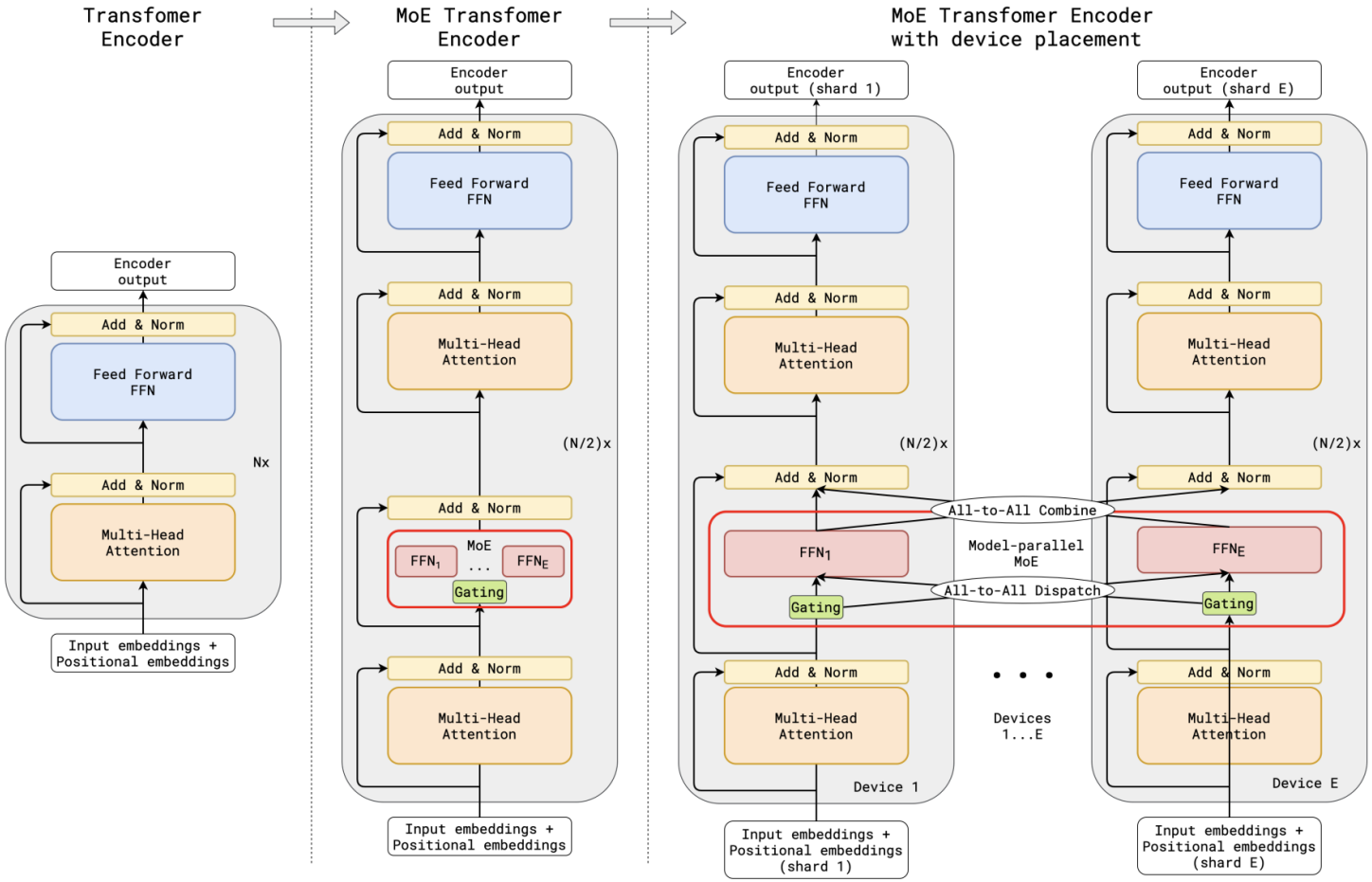


Lepikhin et al. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. ICLR 2021. arXiv:2006.16668

Expert Parallelism

In a mixture-of-experts model, put each expert on a different GPU. The router sends each token to the GPU holding its expert, and the result is sent back

1. Each GPU holds one (or a few) experts and its own batch of tokens.
2. The router picks an expert for every token.
3. Every GPU sends each token to the GPU that holds its expert: an **all-to-all** exchange, where every GPU sends something different to every other GPU.
4. Experts process their tokens, and a second all-to-all sends the results back



Lepikhin et al. GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding. ICLR 2021. arXiv:2006.16668

Expert Parallelism

If the router sends too many tokens to one expert, its GPU becomes the bottleneck. So each expert gets a **capacity**, a maximum number of tokens. Tokens beyond the capacity are **dropped**: they skip the expert and pass to the next layer through the residual connection.

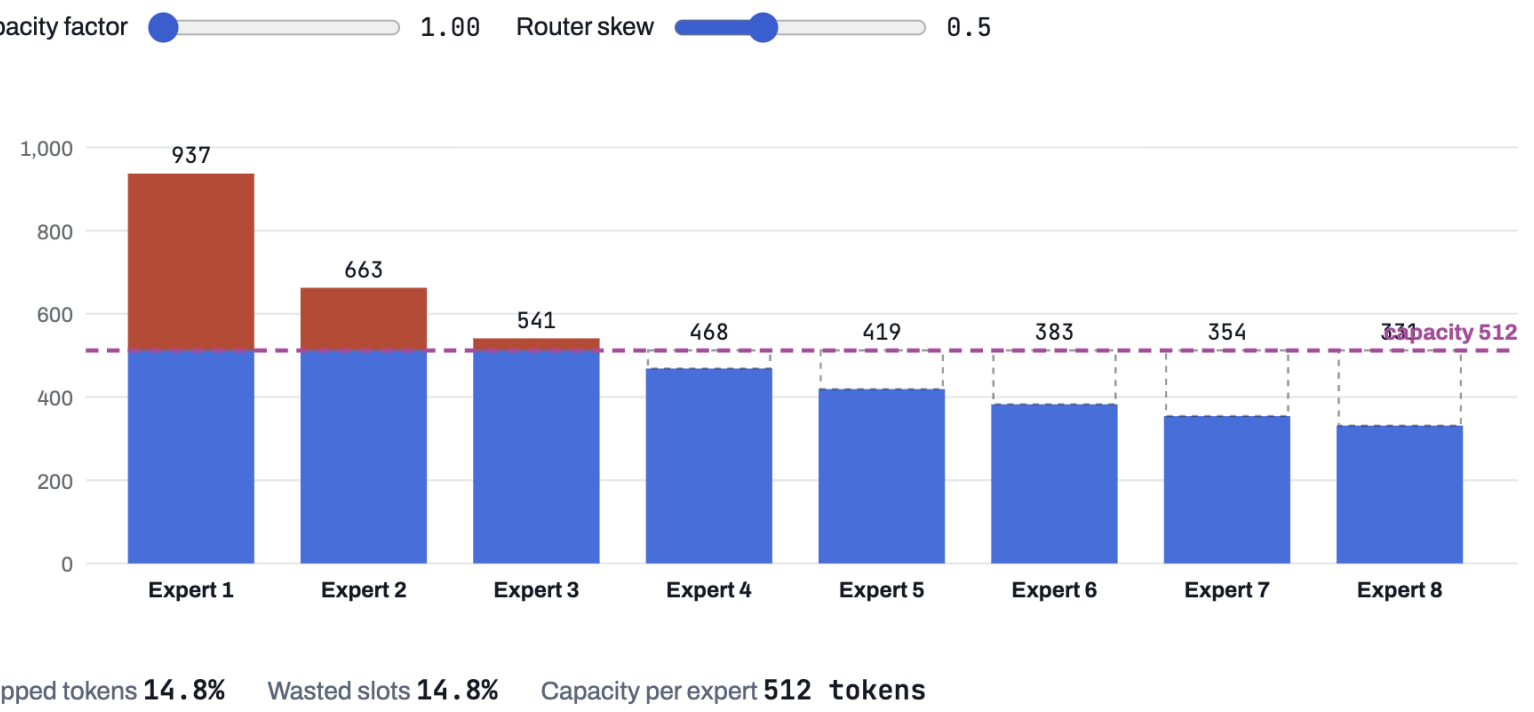
$$capacity = factor \times tokens \div experts$$

4,096 tokens, 8 experts, factor 1.25 → at most 640 tokens per expert.

A capacity factor that is too small drops many tokens, so those tokens get no expert at all. One that is too large reserves slots that stay empty, wasting GPU memory and compute.

Capacity factor versus uneven routing

4,096 tokens routed to 8 experts. "Router skew" makes some experts more popular. Red is dropped tokens; dashed outlines are unused, wasted slots.



The "router skew" controls how uneven it is (0 = perfectly even).

Expert Parallelism

If the router sends too many tokens to one expert, its GPU becomes the bottleneck. So each expert gets a **capacity**, a maximum number of tokens. Tokens beyond the capacity are **dropped**: they skip the expert and pass to the next layer through the residual connection.

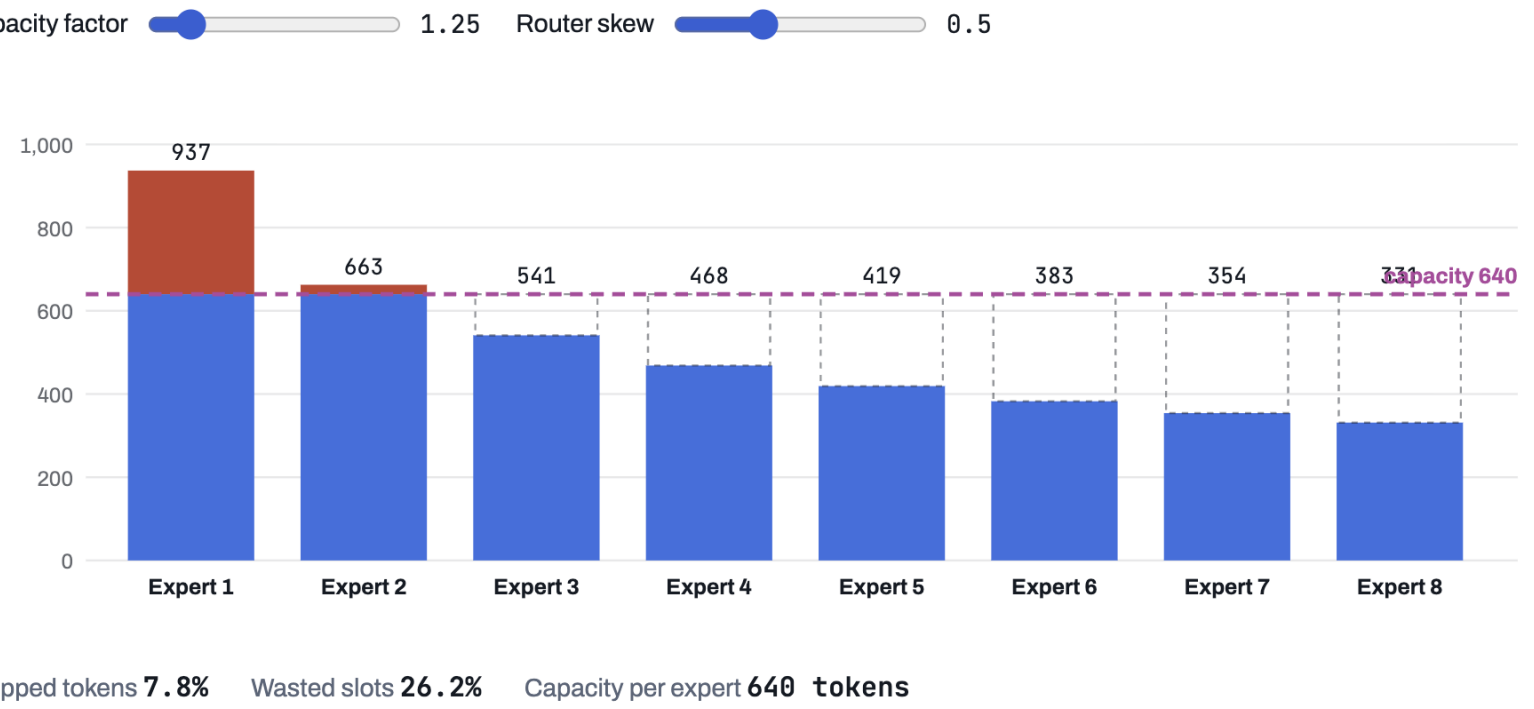
$$capacity = factor \times tokens \div experts$$

4,096 tokens, 8 experts, factor 1.25 → at most 640 tokens per expert.

A capacity factor that is too small drops many tokens, so those tokens get no expert at all. One that is too large reserves slots that stay empty, wasting GPU memory and compute.

Capacity factor versus uneven routing

4,096 tokens routed to 8 experts. "Router skew" makes some experts more popular. Red is dropped tokens; dashed outlines are unused, wasted slots.



The "router skew" controls how uneven it is (0 = perfectly even).

Expert Parallelism

If the router sends too many tokens to one expert, its GPU becomes the bottleneck. So each expert gets a **capacity**, a maximum number of tokens. Tokens beyond the capacity are **dropped**: they skip the expert and pass to the next layer through the residual connection.

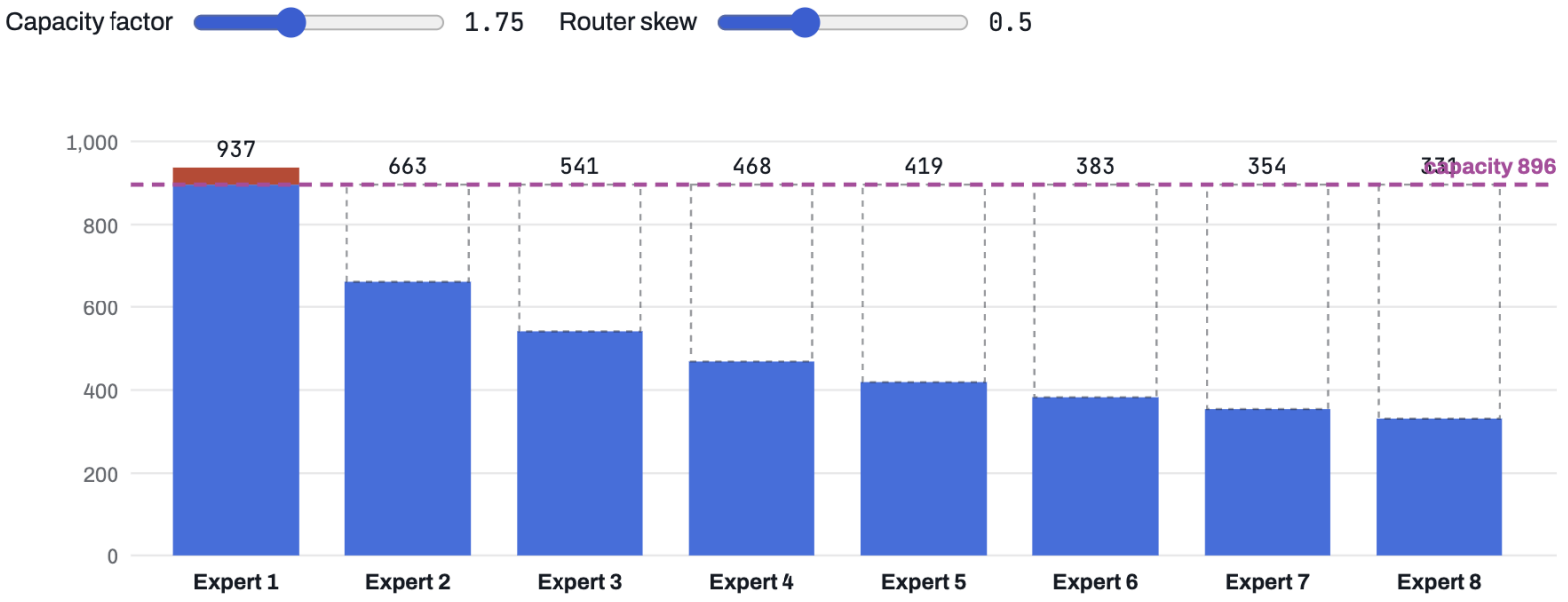
$$capacity = factor \times tokens \div experts$$

4,096 tokens, 8 experts, factor 1.25 → at most 640 tokens per expert.

A capacity factor that is too small drops many tokens, so those tokens get no expert at all. One that is too large reserves slots that stay empty, wasting GPU memory and compute.

Capacity factor versus uneven routing

4,096 tokens routed to 8 experts. "Router skew" makes some experts more popular. Red is dropped tokens; dashed outlines are unused, wasted slots.



Dropped tokens **1.0%** Wasted slots **43.4%** Capacity per expert **896 tokens**

The "router skew" controls how uneven it is (0 = perfectly even).