LLM Parallel Training
Part A

Autumn 2026

Lecturer: Yuedong (Steven) Xu

Shenzhen Loop Area Institute

yuedongxu@slai.edu.cn

Disclaimer

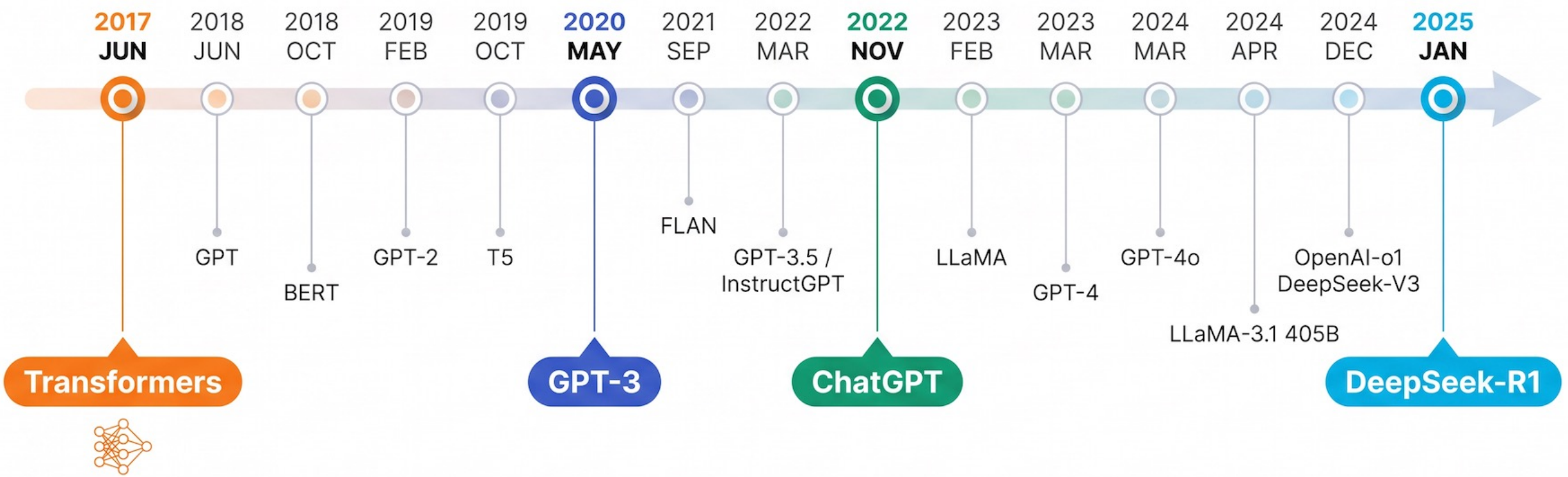
Machine learning systems is a broad and rapidly evolving field. The course material has been developed using a broad spectrum of resources, including research papers, lecture slides, blogposts, research talks, tutorial videos, and other materials shared by the research community. We sincerely appreciate their efforts and assistance, and try our best to cite the sources of the materials used in this course.

LLM Parallel Training: Outline

- **Transformer: An Overview**
- Data Parallelism
  - Parameter-Server
  - All-Reduce
  - Memory Optimization

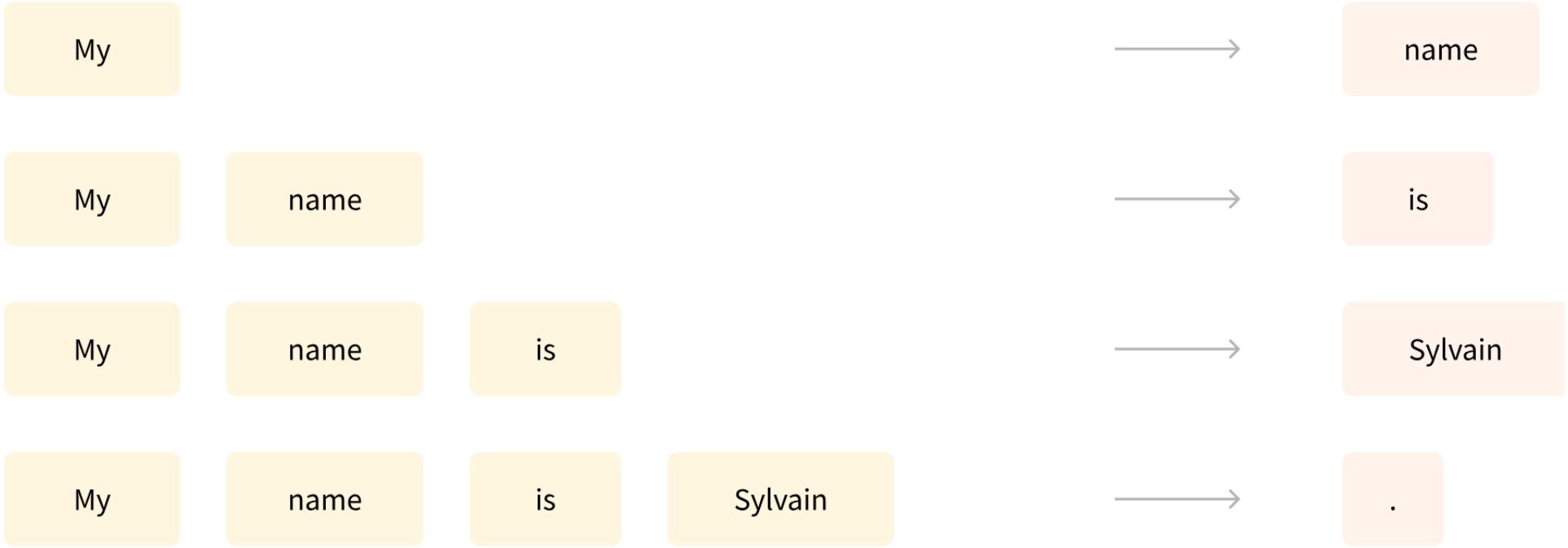
Transformer Overview

- A brief history of Transformer models



Transformer Overview

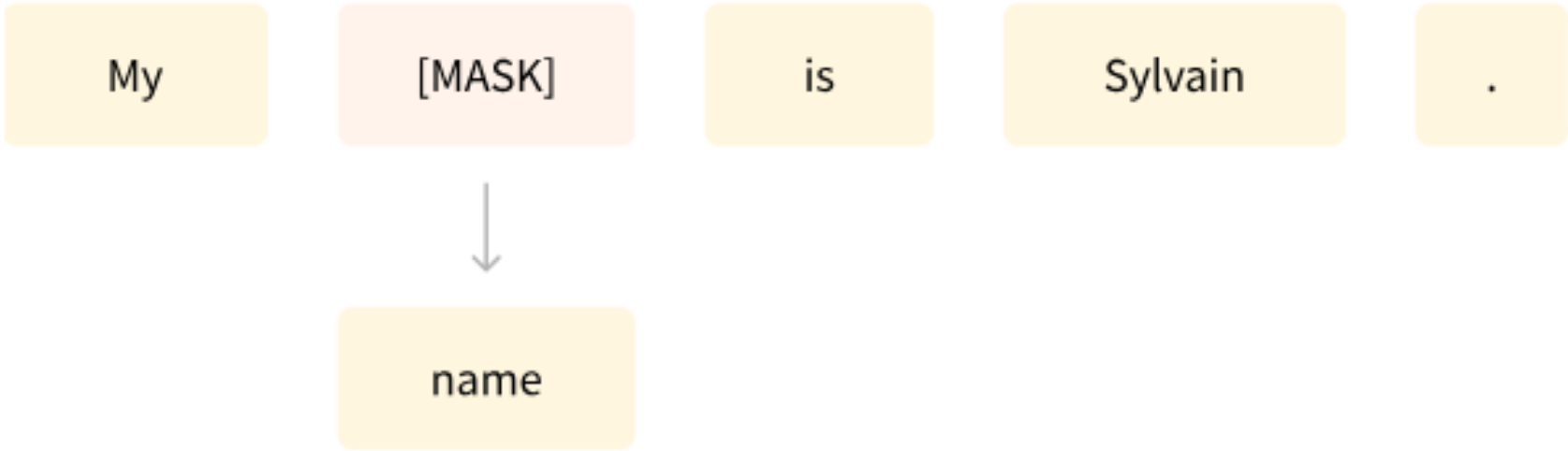
- Causal language model



Predicting the next word in a sentence

Transformer Overview

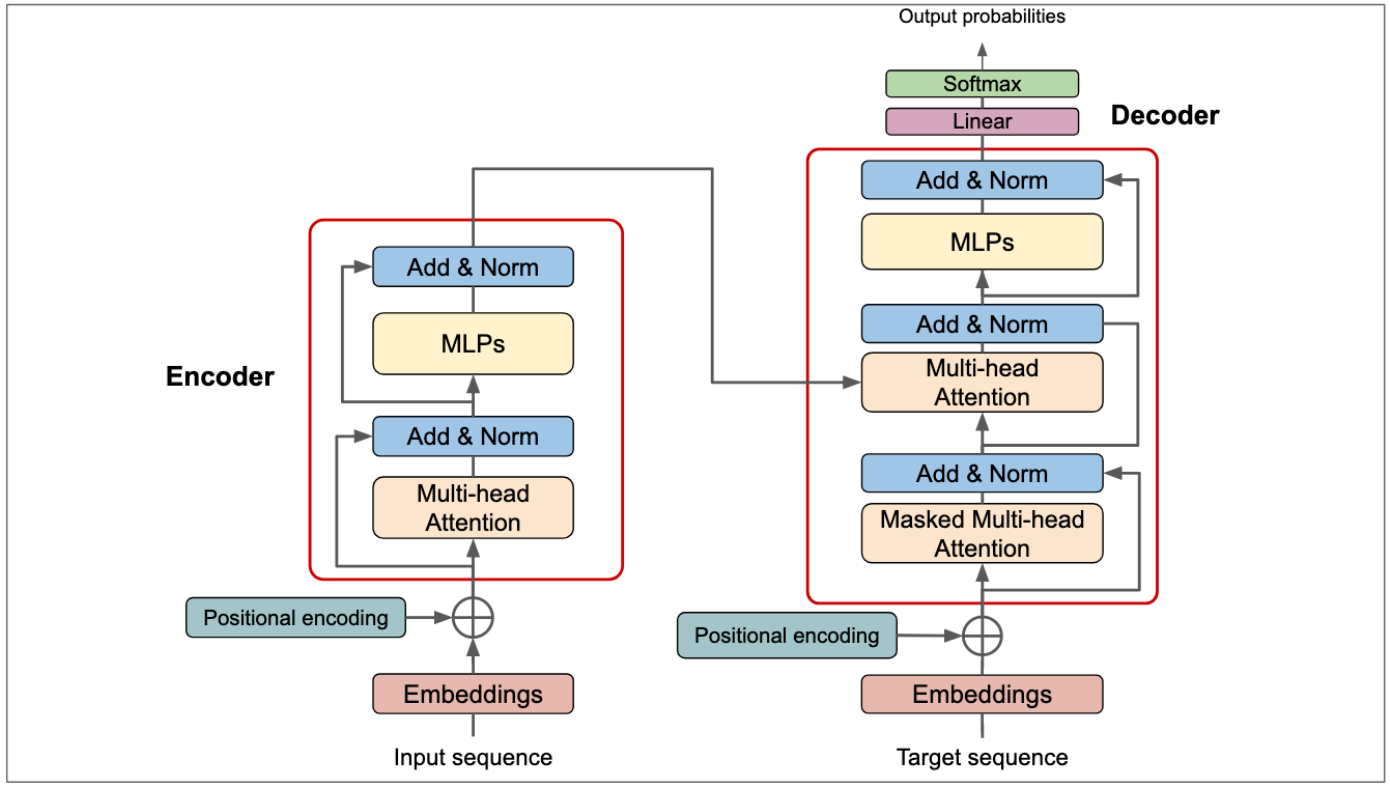
- Non-causal language model



Masked language modeling in which the model predicts a masked word in the sentence

Transformer Overview

- Transformer architecture



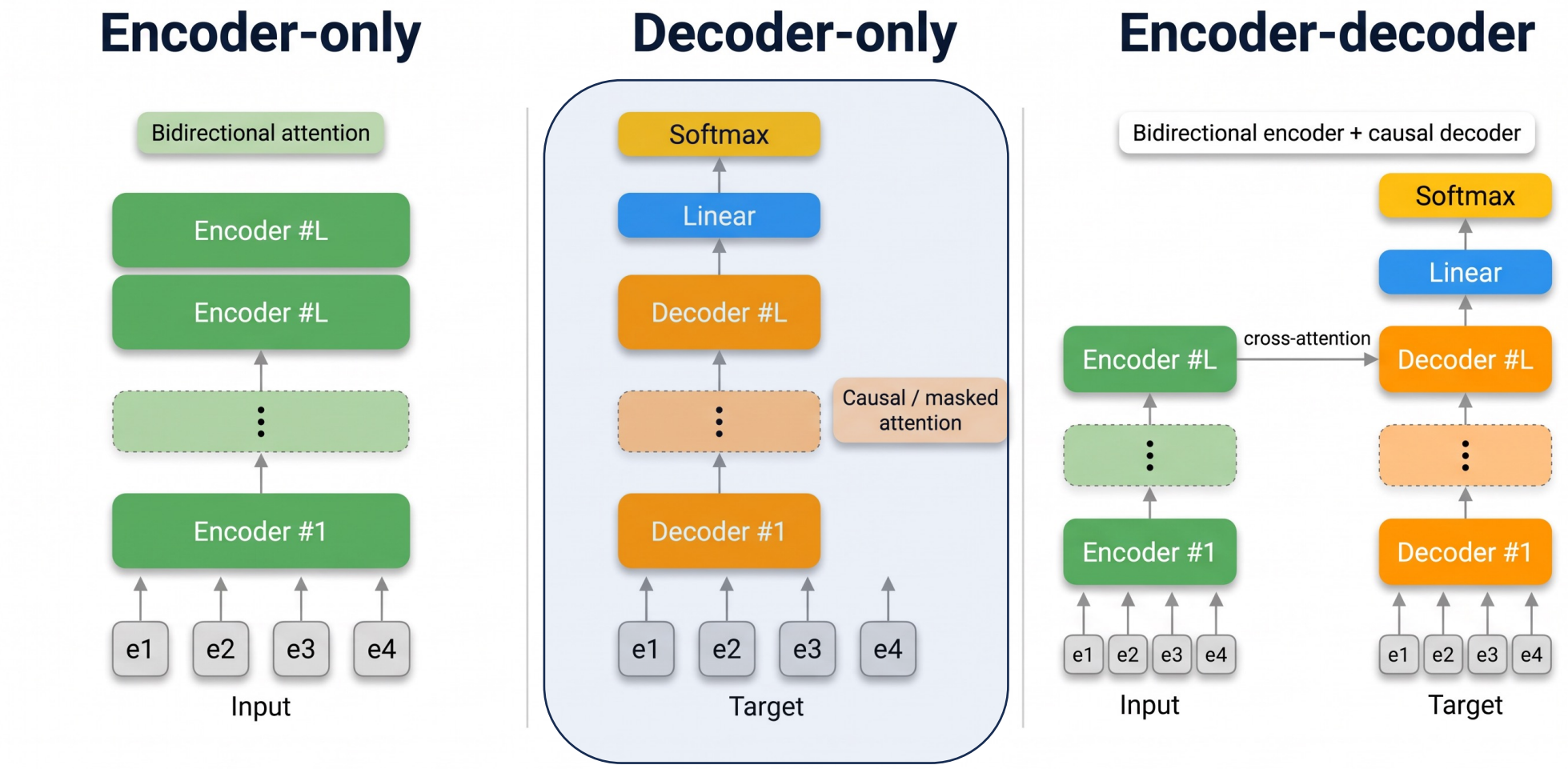
A minimalist description of Transformer *

Encoder (left): receiving an input and building a representation of it (its features), i.e. acquire understanding from the input.

Decoder (right): utilizing the encoder's representation along with other inputs to generate a target sequence, i.e. generating outputs.

* https://re-cinq.com/blog/llm-architectures

Transformer Overview



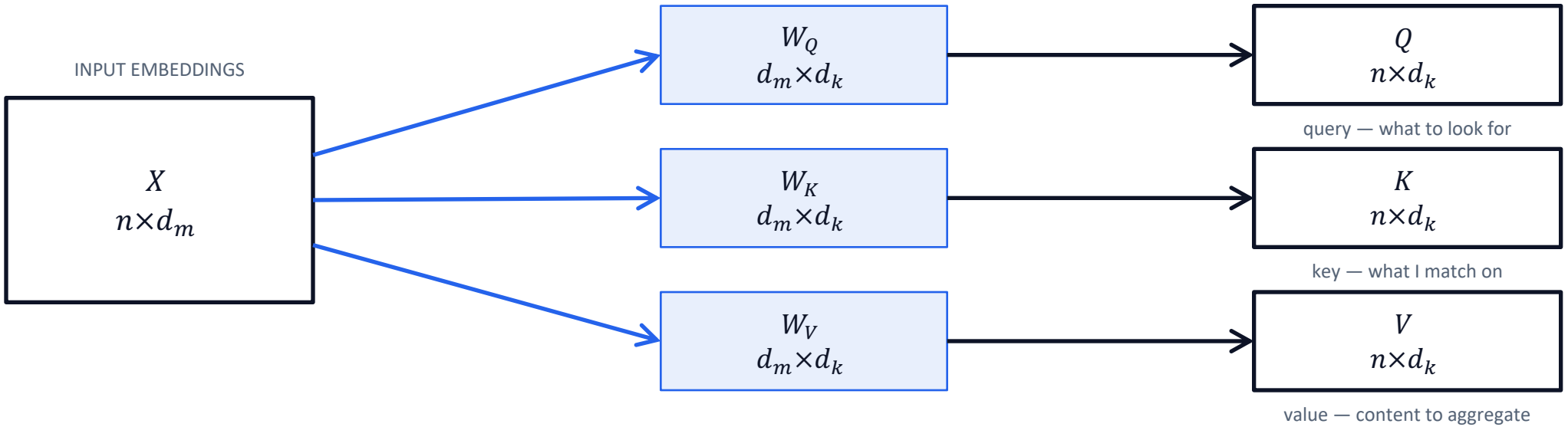
BERT-style · understanding tasks

GPT-style · generation tasks

T5 / BART-style · translation & summarization

One input X, three learned projections yield Q, K, V

The same token embeddings are linearly transformed by three independent weight matrices, each serving a distinct role.

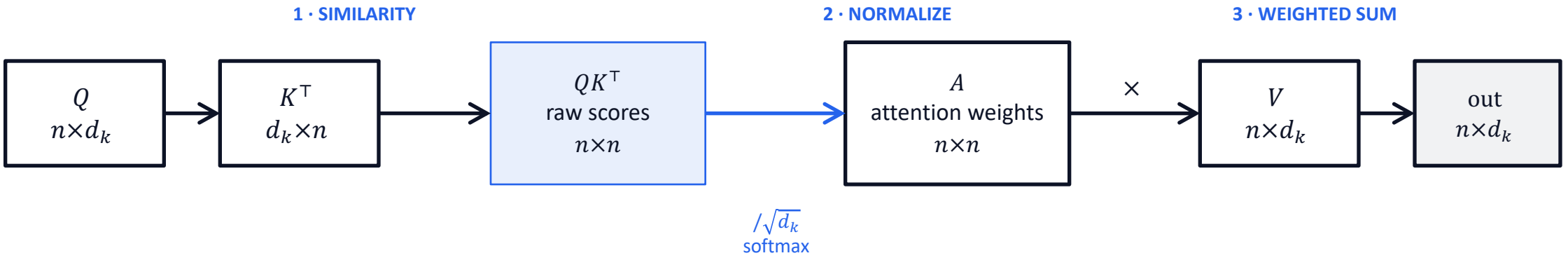


$$Q = XW_Q \qquad K = XW_K \qquad V = XW_V$$

Shapes: $X \in \mathbb{R}^{n \times d_m}$, $W_Q, W_K, W_V \in \mathbb{R}^{d_m \times d_k}$, hence $Q, K, V \in \mathbb{R}^{n \times d_k}$. The three matrices are distinct parameters trained end-to-end; in multi-head attention $d_k = d_m / h$.

Normalized $QK^\top$ scores, then a weighted sum over V

Similarity scores are scaled and normalized into row-stochastic weights, which then take a weighted average of the value rows.



$$\text{Attention}(\mathrm{Q}, \mathrm{K}, \mathrm{V}) = \text{softmax}\left(\frac{\mathrm{QK}^\top}{\sqrt{\mathrm{d_k}}}\right)\mathrm{V}$$

$QK^\top$ gives an unnormalized similarity score for every query–key pair. Dividing by $\sqrt{d_k}$ keeps the dot products at a moderate scale so softmax does not saturate; the row-wise softmax turns each row into weights that sum to 1. The final output is then a weighted sum of the value rows, where attention decides how much each token contributes.

The FFN transforms each token independently

A shared two-layer MLP expands features, applies nonlinearity, then returns to model width.

POSITION-WISE COMPUTATION



$$FFN(X) = \phi(XW_1 + b_1)W_2 + b_2$$

typically **d_ff = 4 × d_model**.

WHAT IT DOES

FFN mixes features within each token.

PRE-NORM RESIDUAL FORM

$$Z_{out} = Z + FFN(LN(Z))$$

Parameter count of a standard transformer block

One multi-head self-attention sublayer, one feed-forward network, and two affine LayerNorms. All linear layers include biases.

COMPONENT SUM

$$P_{block} = P_{MHA} + P_{FFN} + 2P_{LN}$$

Multi-head attention

Q, K, V projections + output projection

$$P_{MHA} = 3(d^2 + d) + (d^2 + d) = 4d^2 + 4d$$

With d_k = d_v = d / h, head splitting changes shape—not the parameter total.

Feed-forward network

Expansion projection + return projection

$$P_{FFN} = (dd_{ff} + d_{ff}) + (d_{ff}d + d) = 2dd_{ff} + d_{ff} + d$$

Weights: d × d_ff and d_ff × d. Biases: d_ff and d.

Each LayerNorm

$$P_{LN} = d + d = 2d$$

One scale vector and one shift vector; residual adds have no parameters.

CLOSED FORM

$$P_{block} \approx 4d^2 + 2dd_{ff}$$

d = d_model • EXCLUDES EMBEDDINGS, POSITIONAL PARAMETERS & OUTPUT HEAD

GPT3: a 12288-wide block has 1,812,099,072 parameters

Use d_model = 12,288, d_ff = 49,152, and h = 96 heads. Thus d_k = d_v = 128; each head uses a slice of the same full projection matrices.

SUBSTITUTE THE DIMENSIONS

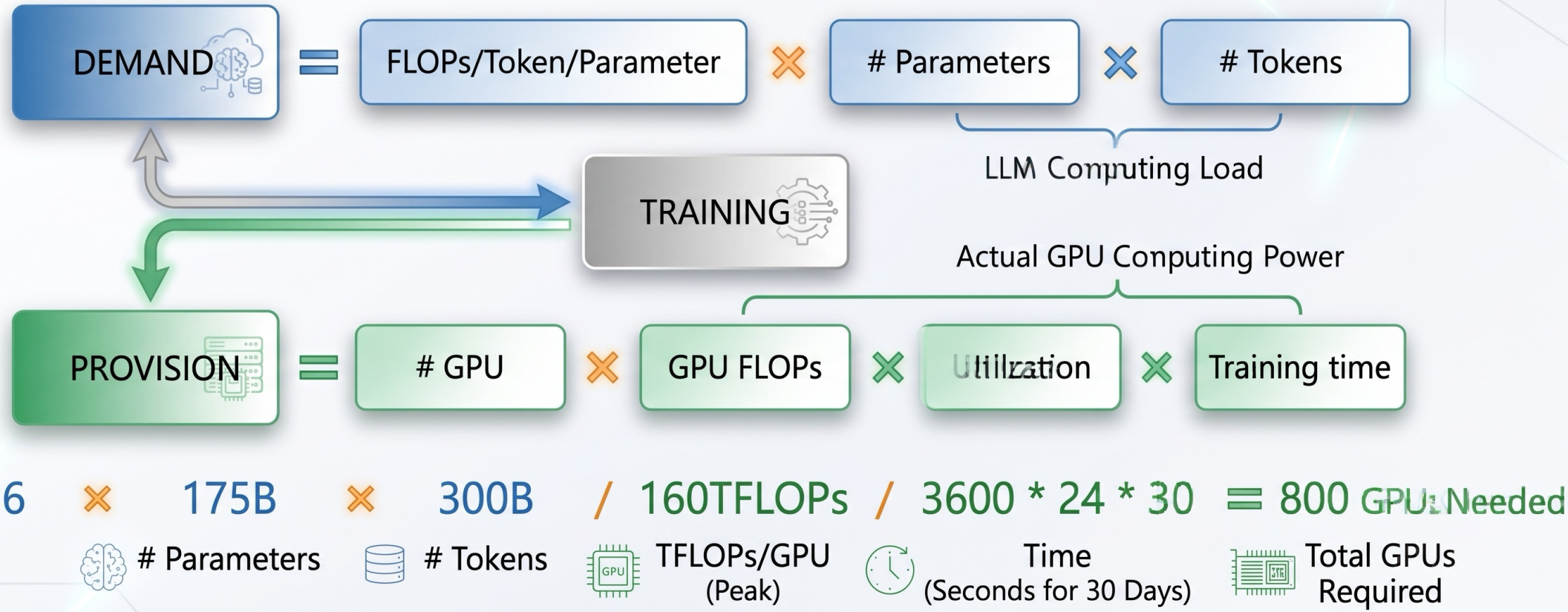
COMPONENT	CALCULATION	PARAMETERS
MHA — Q, K, V	$3 \times (12288^2 + 12288)$	453,021,696
MHA — output	$12288^2 + 12288$	151,007,232
FFN — linear 1	$12288 \times 49152 + 49152$	604,028,928
FFN — linear 2	$49152 \times 12288 + 12288$	603,992,064
Two LayerNorms	$2 \times (12288 + 12288)$	49,152
TOTAL BLOCK	MHA + FFN + LN	**1,812,099,072**

$$4(512)^2 + 2(512)(2048) + 2048 + 9(512) = 3{,}152{,}384$$

Independent of batch size and sequence length.

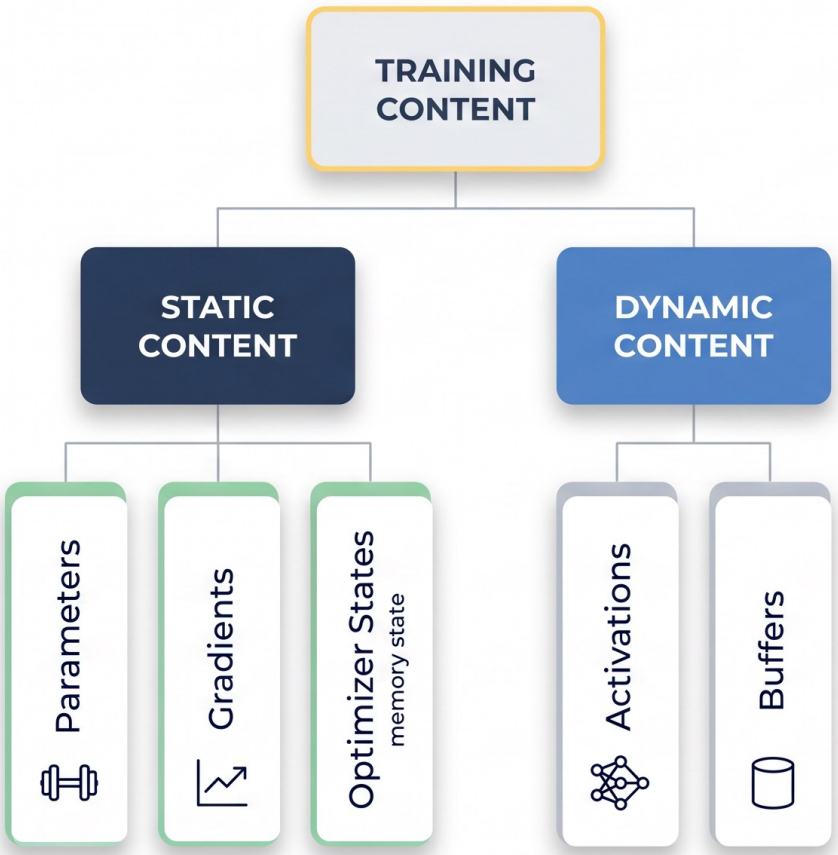
Why do we need “distributed training”?

How many GPUs do we need?



《Language Models are Few-Shot Learners》

GPU HBM Content During Training



An example of GPT-3 175B

OPTIMIZER STATES

32-bit Parameter	**700 GB**
Adam Moment	**700 GB**
Adam Variance	**700 GB**

16-bit Parameter
350 GB

16-bit Gradient
350 GB

Activations
(depending on batch size)

Buffer and Fragmentation

ADAM OPTIMIZER: ADAPTIVE MOMENT ESTIMATION

01. FIRST MOMENT (MEAN) ESTIMATE

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \frac{\partial L}{\partial w_t}$$

02. SECOND MOMENT (VARIANCE) ESTIMATE

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) \left(\frac{\partial L}{\partial w_t} \right)^2$$

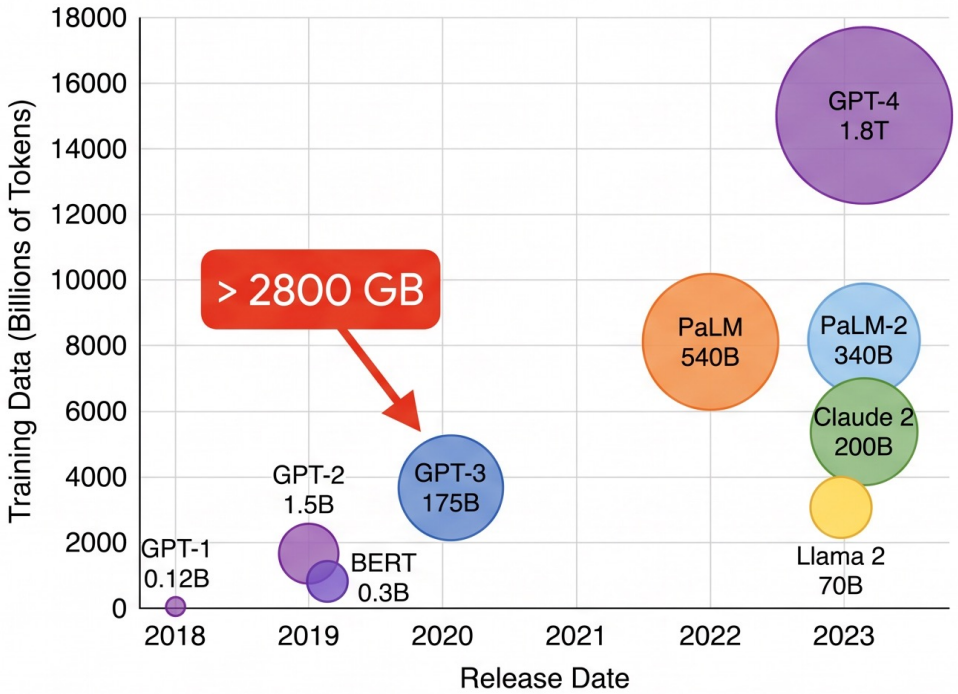
03. BIAS CORRECTION

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

04. FINAL PARAMETER UPDATE

$$w_{t+1} = w_t - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

GPT-175B models: storage

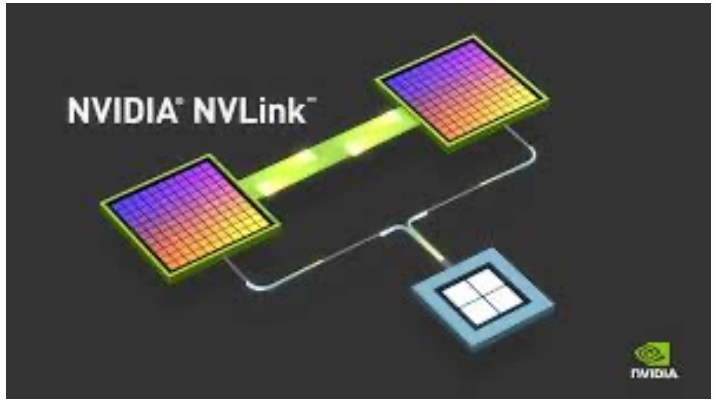


Models become larger and larger

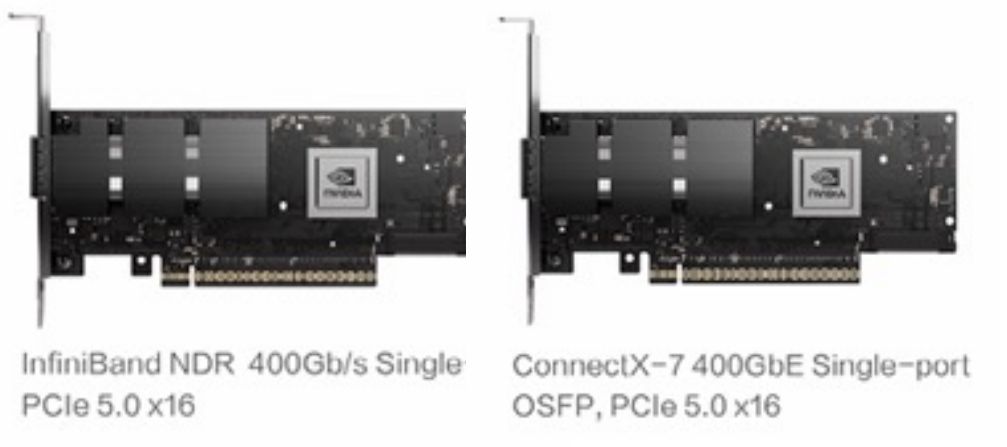
GPU	Memory	Type
Tesla V100	32GB	HBM2
NVIDIA A100	40/80GB	HBM2E
NVIDIA A800	80GB	HBM2E
NVIDIA H100	80GB	HBM3
NVIDIA H800	80GB	HBM3
NVIDIA B200	192GB	HBM3e

GPU HBM size remains smal.

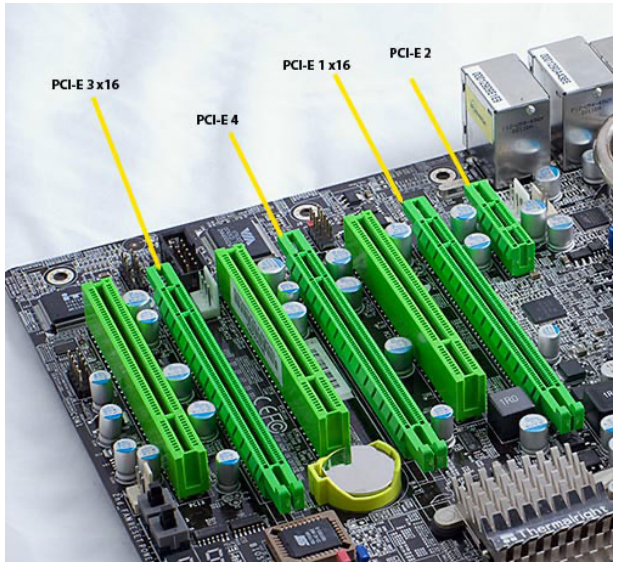
- Multiple GPUs compute collaboratively
  - Data Parallel, Pipeline Parallel, Tensor Parallel, Expert Parallel, Sequence Parallel, Context Parallel
- GPU interconnection



NVLink 5.0 (e.g. 1.8TB/s)



RDMA (RoCE/InfiniBand, e.g. 400Gbps)



PCIe 5.0 (e.g. 128GB/s, 16lanes)

- Importance of Communication
  - Traffic volume
    - 16-bit gradient of GPT3-175B: 350GB needs to be transmitted in every iteration on one GPU
    - Even more data communication if model is partitioned
  - Imbalanced technology upgrading
    - Ampere A100 (FP16 312TFLOPS, released in 2020) → Blackwell B300 (FP8 72PFLOPS, 2025) → Rubin (2026)
    - NVLink 3.0 (600GB/s, 2020) → NVLink 4.0 (900GB/s, 2022) → NVLink 5.0 (1.8TB/s, 2025) → NVLink 6.0 (3.6TB/s, 2026)

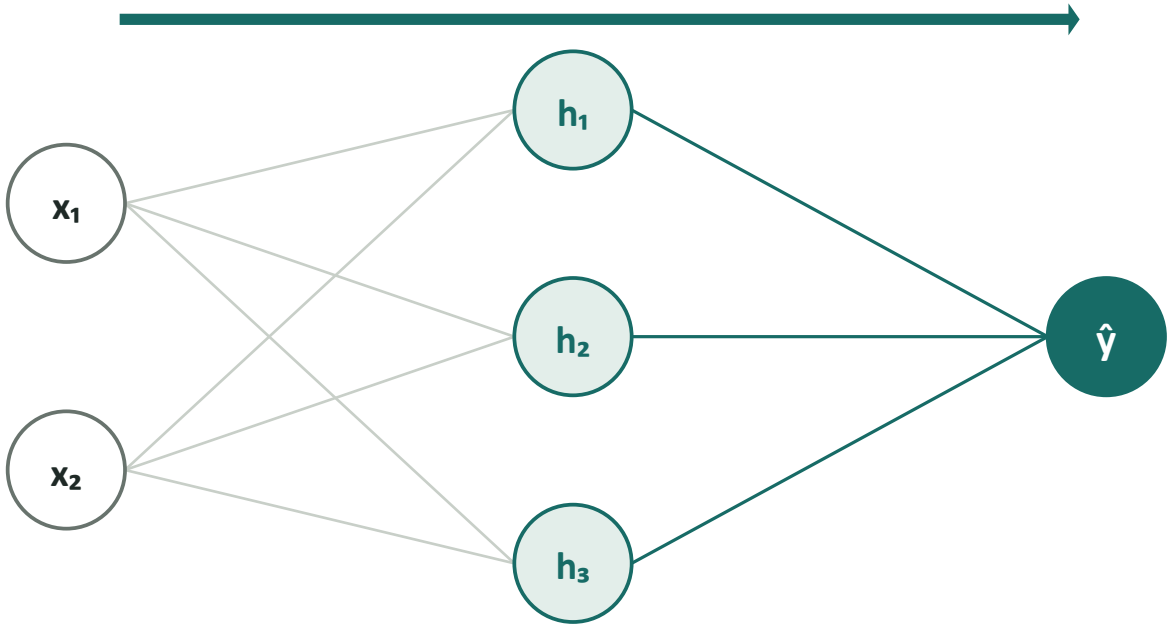
LLM Parallel Training: Outline

- Transformer: A Quick Overview
- Data Parallelism
  - **Parameter-Server**
  - All-Reduce
  - Memory Optimization

Forward propagation: turn inputs into a prediction

Each layer applies a linear transformation followed by a nonlinear activation; weights remain fixed.

ACTIVATION FLOW • LEFT TO RIGHT



INSIDE ONE NEURON

$$z = \sum_i^L w_i x_i + b$$
$$a = \sigma(z)$$

Weighted sum first; nonlinear activation second.

REPEAT AT EVERY LAYER ℓ

$$z^{(\ell)} = W^{(\ell)} a^{(\ell-1)} + b^{(\ell)}$$
$$a^{(\ell)} = \sigma(z^{(\ell)})$$

$$\hat{y} = a^{(L)}, \mathcal{L} = loss(\hat{y}, y)$$

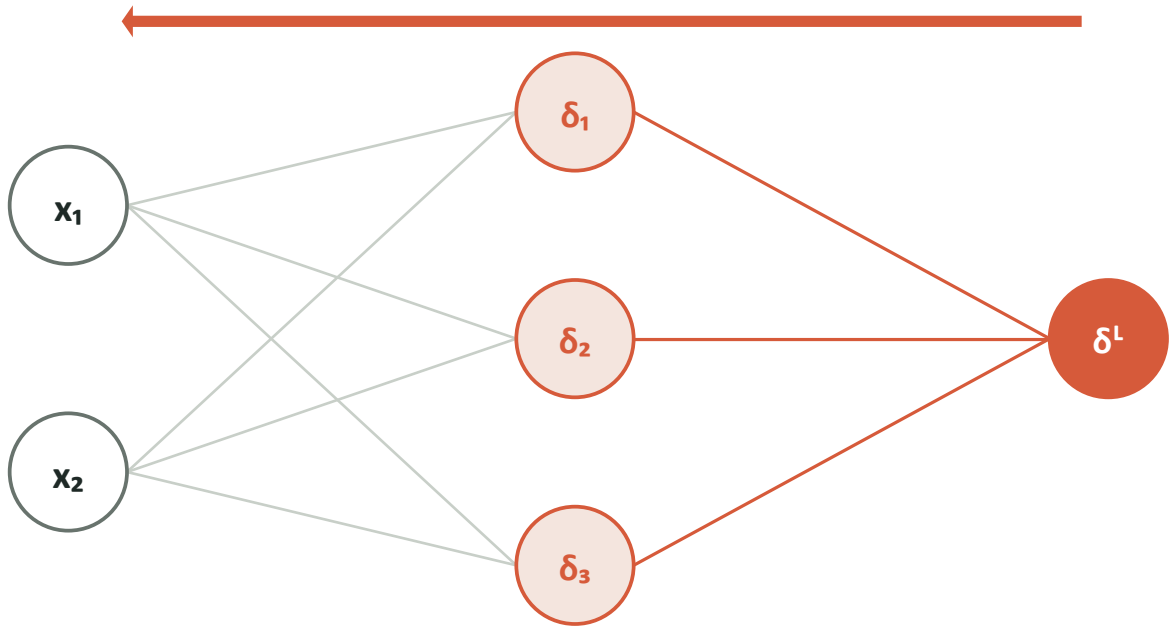
SAVE FOR BACKPROP

Cache each layer's z and a values; the backward pass will reuse them.

Backpropagation: distribute the loss into gradients

The chain rule moves error signals backward through the network; an optimizer then updates the parameters.

GRADIENT FLOW • RIGHT TO LEFT



NO UPDATE TO x | ERROR SIGNALS | START FROM LOSS

01 / OUTPUT ERROR

$$\delta^{(L)} = \nabla_{a^{(L)}}\mathcal{L} \odot \sigma'(z^{(L)})$$

Start with the loss gradient at the output.

02 / PROPAGATE ERROR

$$\delta^{(\ell)} = (W^{(\ell+1)})^T \delta^{(\ell+1)} \odot \sigma'(z^{(\ell)})$$

Weights carry error backward; cached activation derivatives scale it locally.

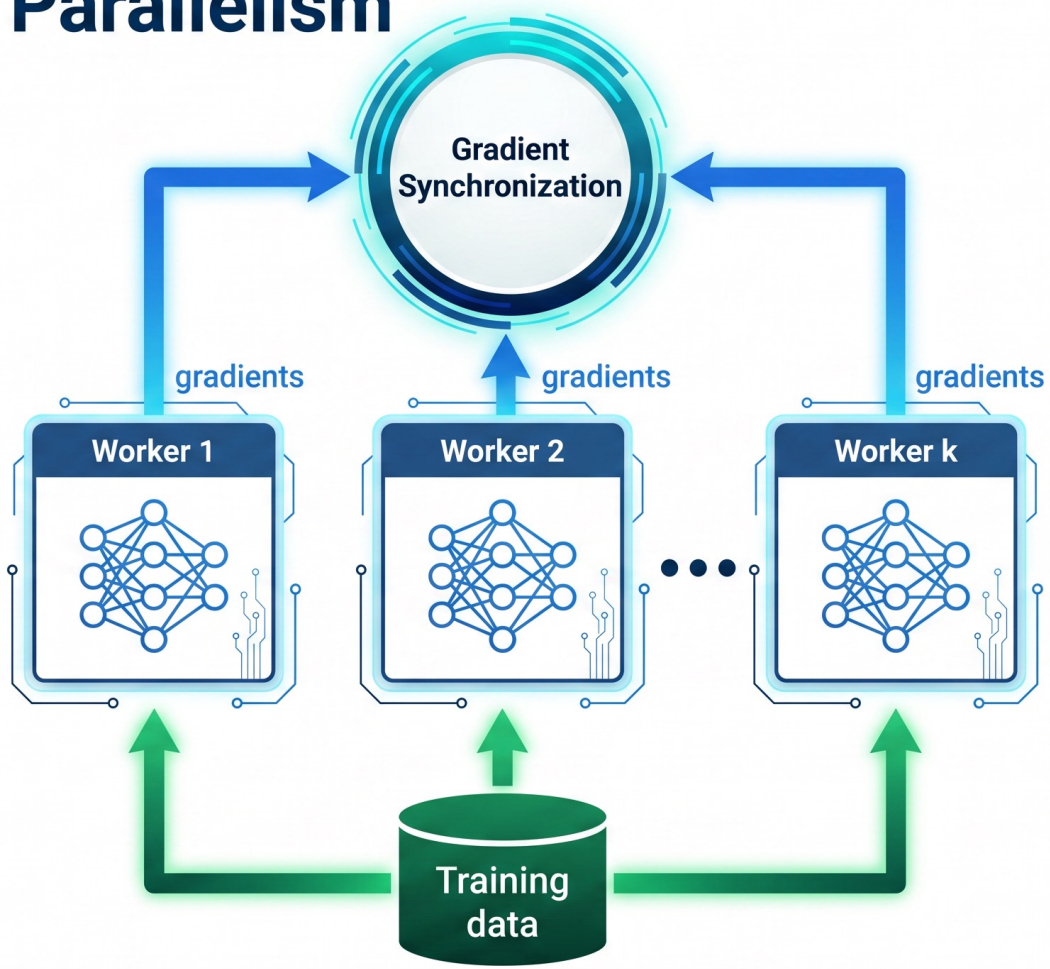
03 / UPDATE

$$\nabla_{W^{(\ell)}}\mathcal{L} = \delta^{(\ell)}(a^{(\ell-1)})^T, \nabla_{b^{(\ell)}}\mathcal{L} = \delta^{(\ell)}$$
$$\theta \leftarrow \theta - \eta\nabla_\theta\mathcal{L}$$

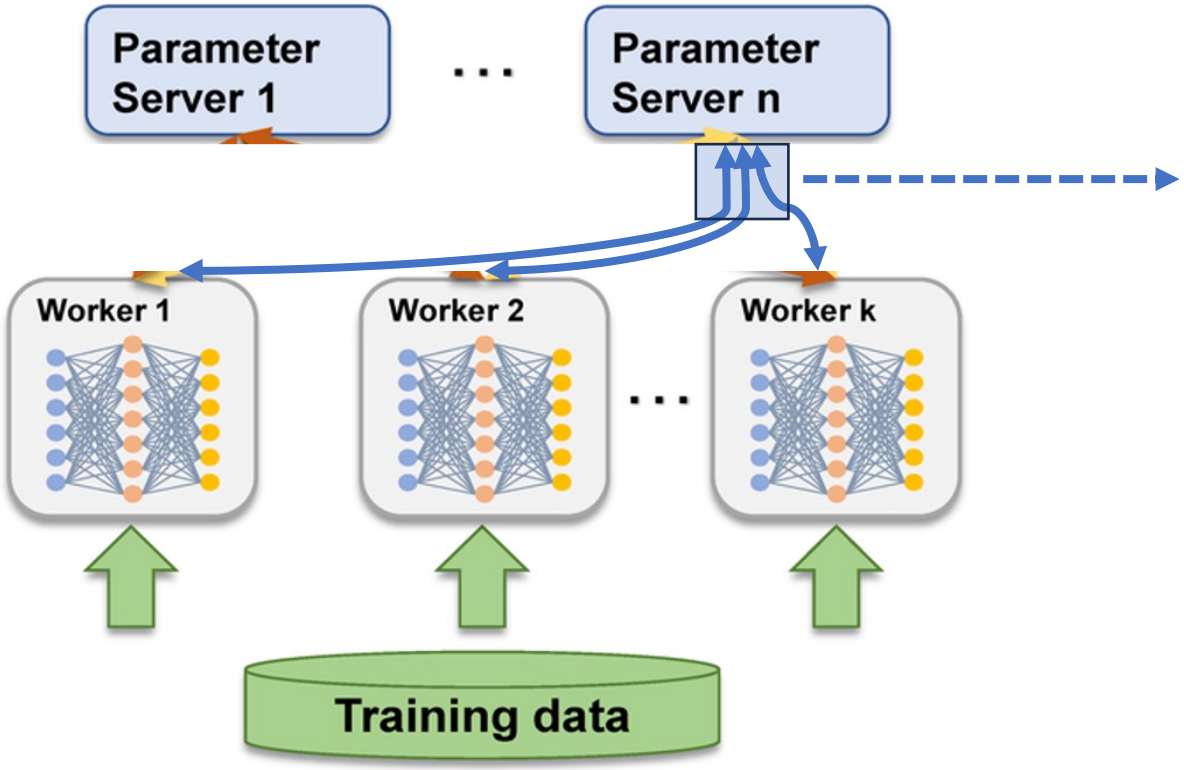
Compute parameter gradients, then step opposite the gradient.

Data Parallelism

- Distribute data to workers
- Each worker works independently
- Synchronize gradients via different approaches
- Repeat the above procedures until model convergence
- Aggregate compute power



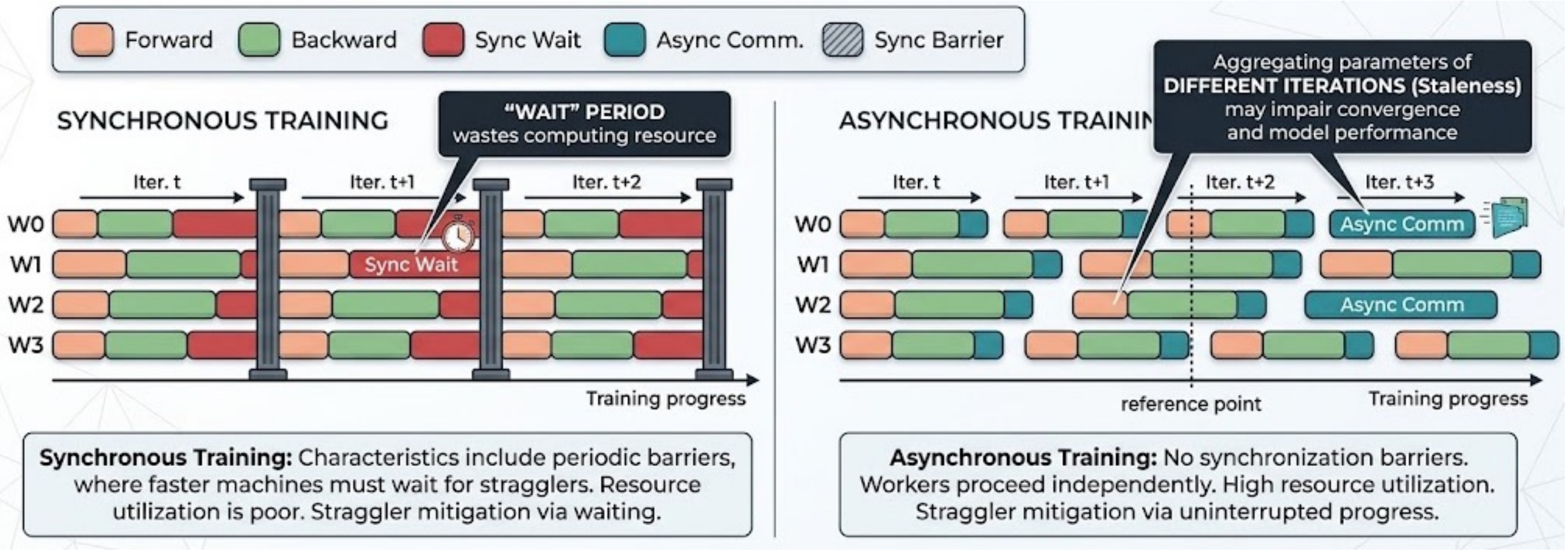
Parameter Server Architecture



- Shared bottleneck
  - Communication throttles computation
  - Reduced bandwidth efficiency due to multi-flow competition
  - Difficult to overlap comp. and comm., push and pull

Network Bottleneck

Straggler Problem



Asynchronous Training: Tolerating Stragglers

LLM Parallel Training: Outline

- Transformer: A Quick Overview
- Data Parallelism
  - Parameter-Server
  - **All-Reduce**
  - Memory optimization

- Collective Communication

> "***Collective communication*** *is communication that involves a group of processing elements (termed nodes in this entry) and affects a data transfer between all or some of these processing elements. Data transfer may include the application of a reduction operator or other transformation of the data."* *《Encyclopedia of Parallel Computing》*

集合通信是指一个进程组的所有进程都参与的全局通信操作。

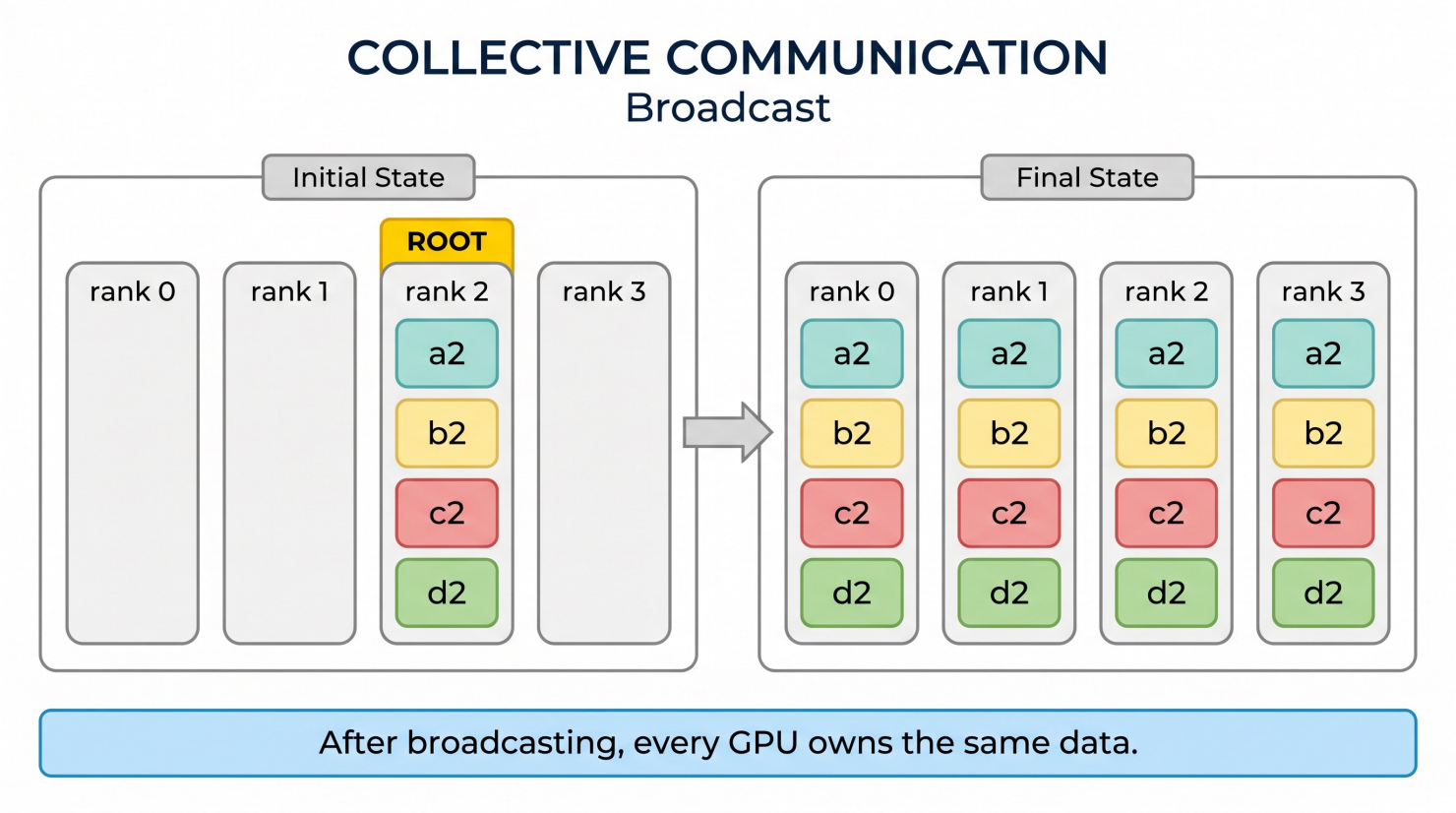


全部点对点通信完成才算集合通信完成

- Why Collective Communication
  - Simplified Programming Interface
    - Developers don't need to manually code complex synchronization and data distribution logic for every scenario.
  - Scalability for Large-Scale Systems
    - As systems grow to thousands of nodes or GPUs (e.g., in supercomputers or cloud clusters), managing communication becomes exponentially complex. These libraries support sparse data handling, fault tolerance, and observability features to ensure reliable operation at scale.
  - Decompose Compute and Communication
    - Allowing machine learning researchers and system engineers to work on their own.

Broadcast — one source, the same data everywhere

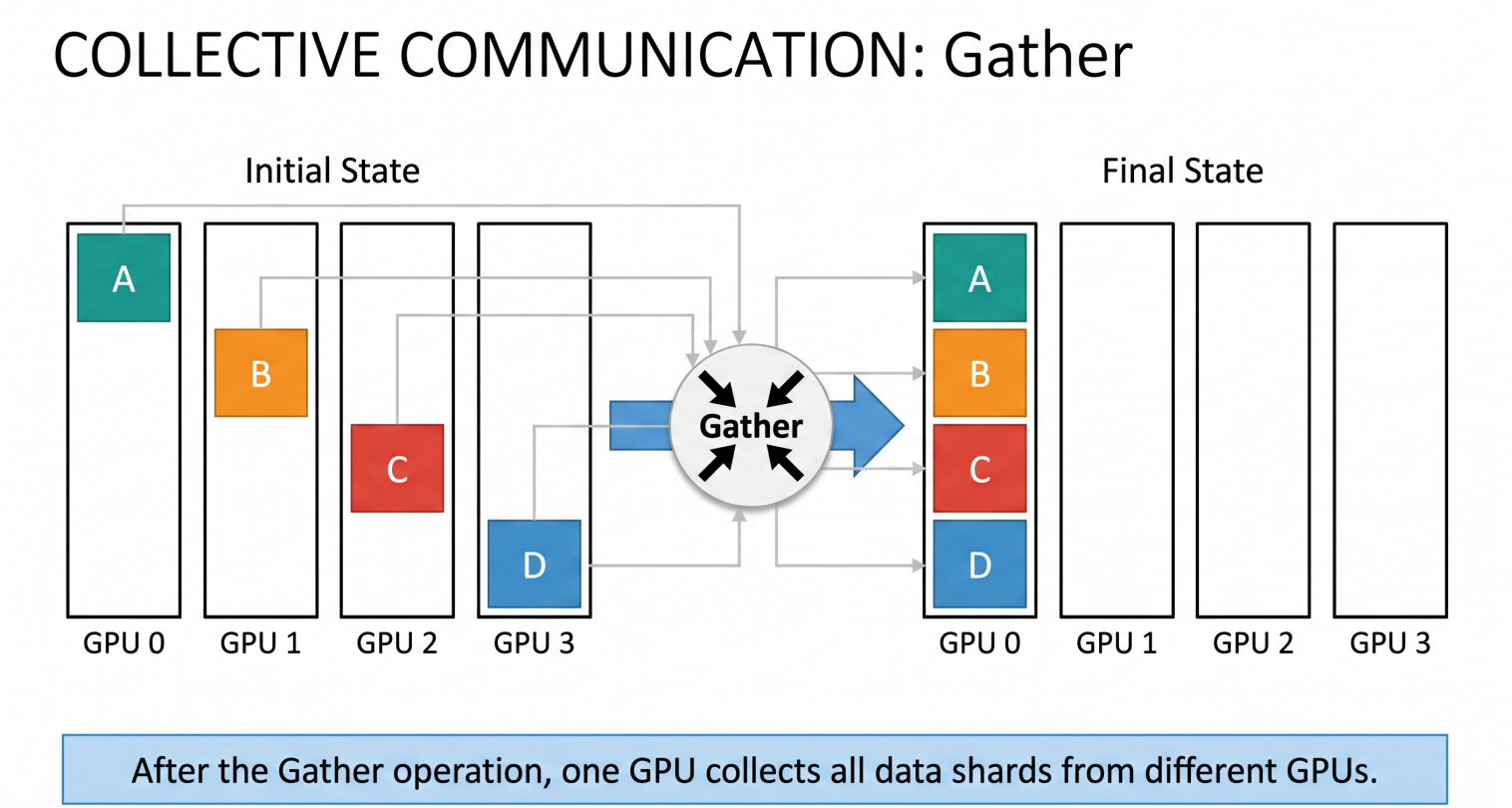
Root-owned data is replicated to all ranks; no values are combined.



Output: every rank holds an identical copy of the root's buffer.

Gather — collect every shard at one rank

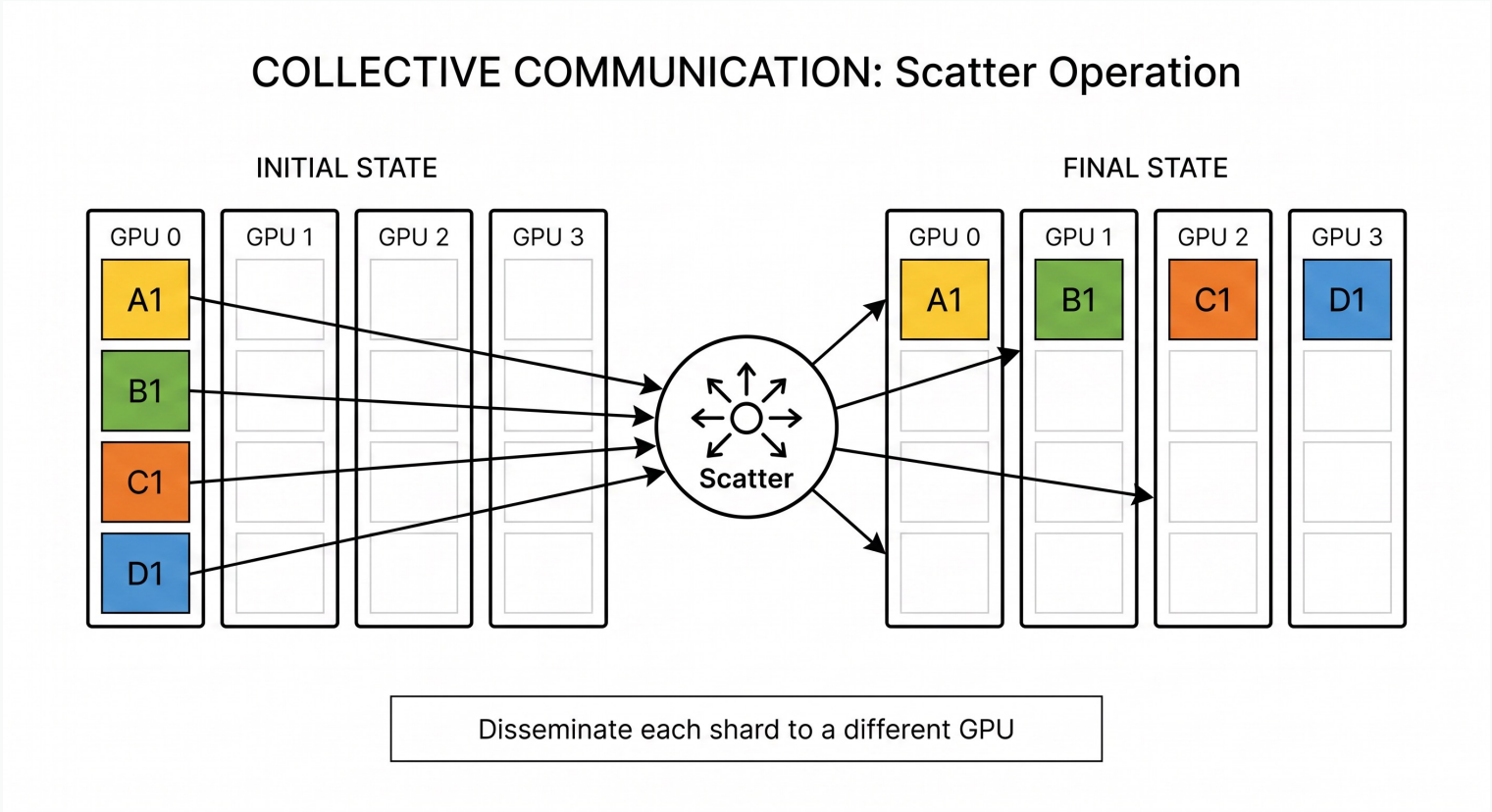
All participants contribute; only the root receives the complete collection.



Output: the root owns all shards; non-root ranks keep only their local data.

Scatter — distribute distinct shards from one rank

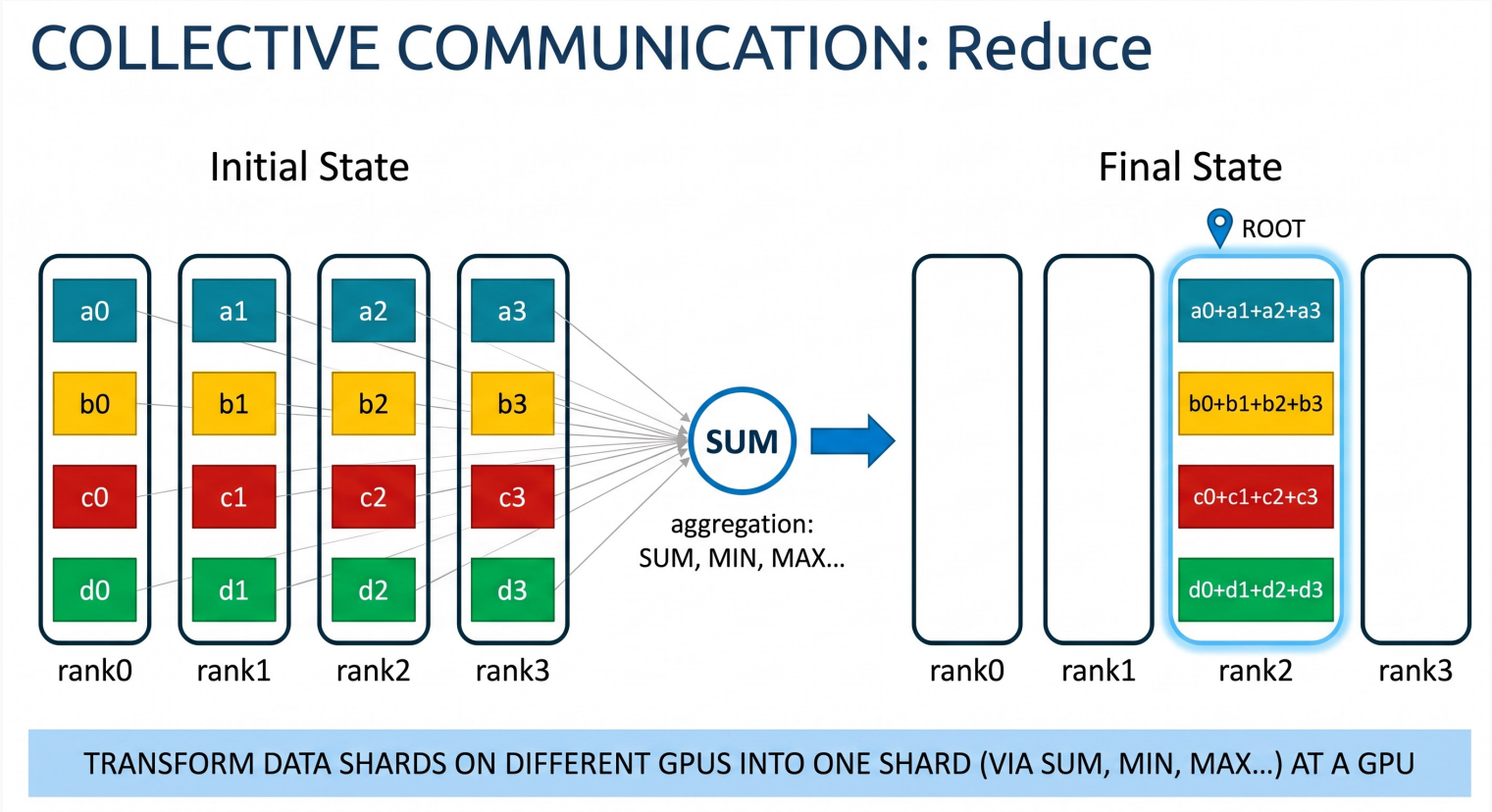
The root partitions its data and sends a different shard to each participant.



Output: each rank owns one shard; no rank receives the full source buffer.

Reduce — combine values, return one result at the root

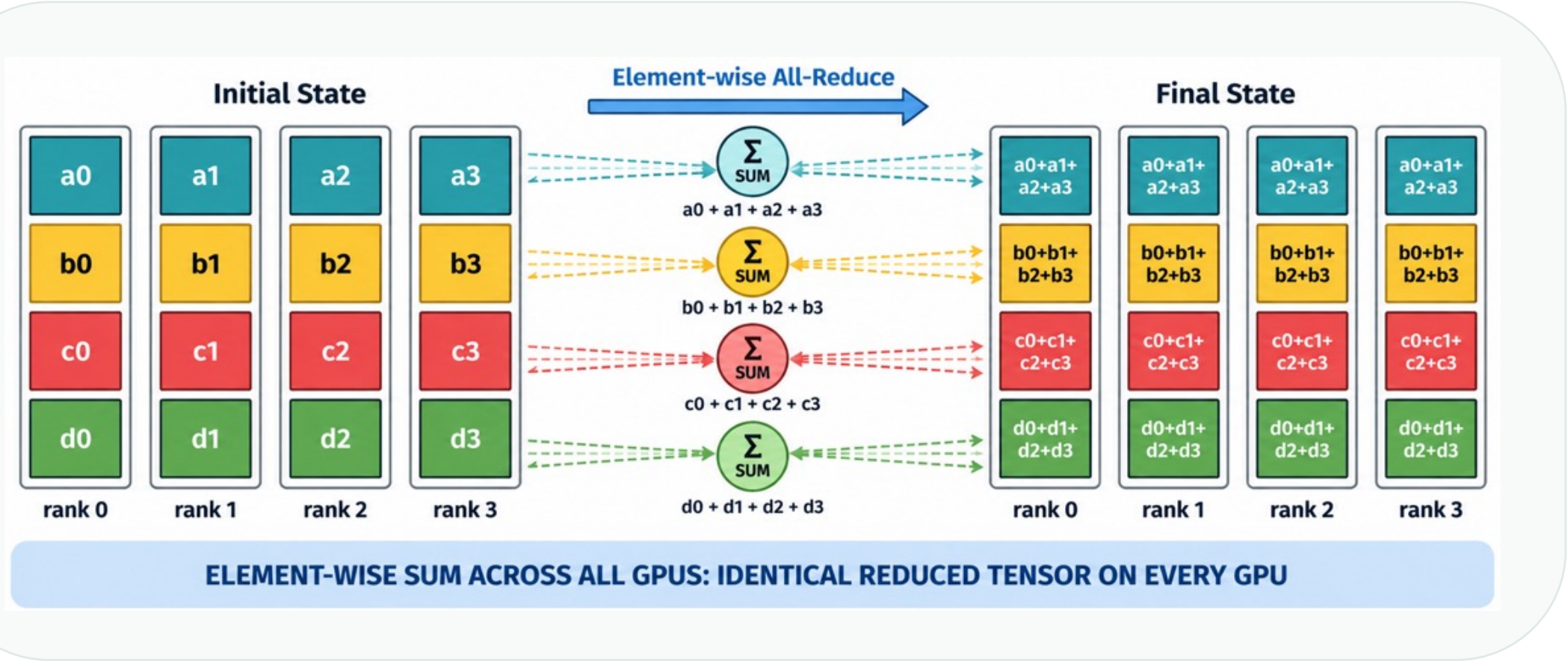
Matching elements are aggregated with sum, max, min, or another reduction operator.



Reduce = aggregation + one destination.

AllReduce — one reduced result on every rank

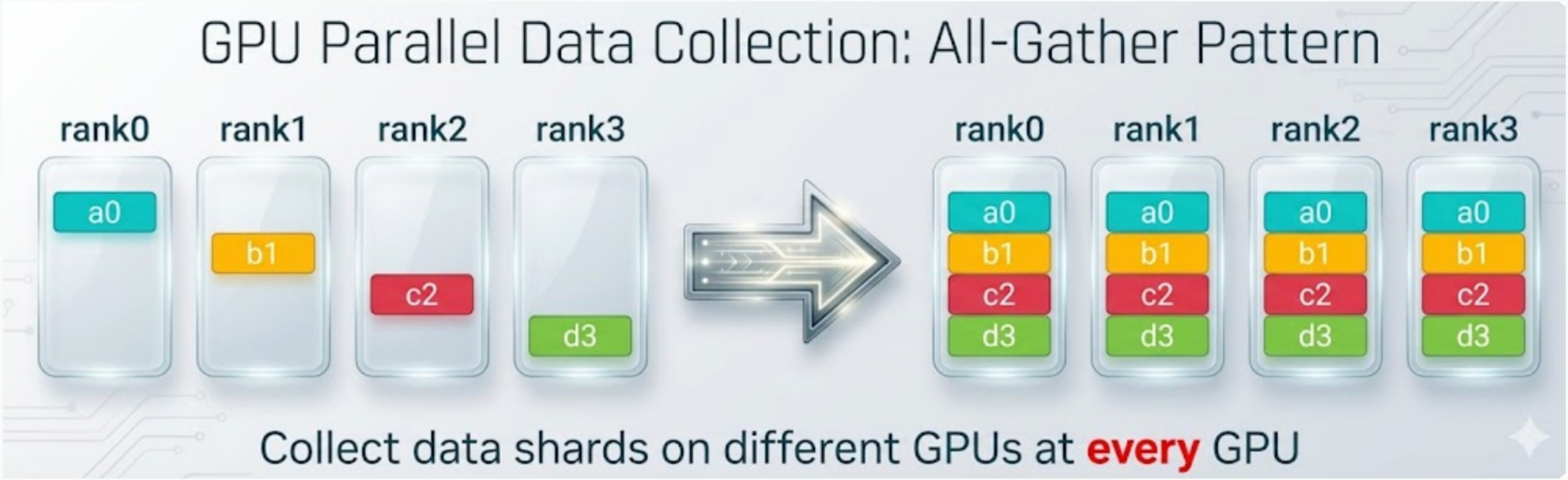
Every participant contributes data and receives the same aggregated output.



Typical use: synchronize gradients across data-parallel workers.

AllGather — every rank receives every shard

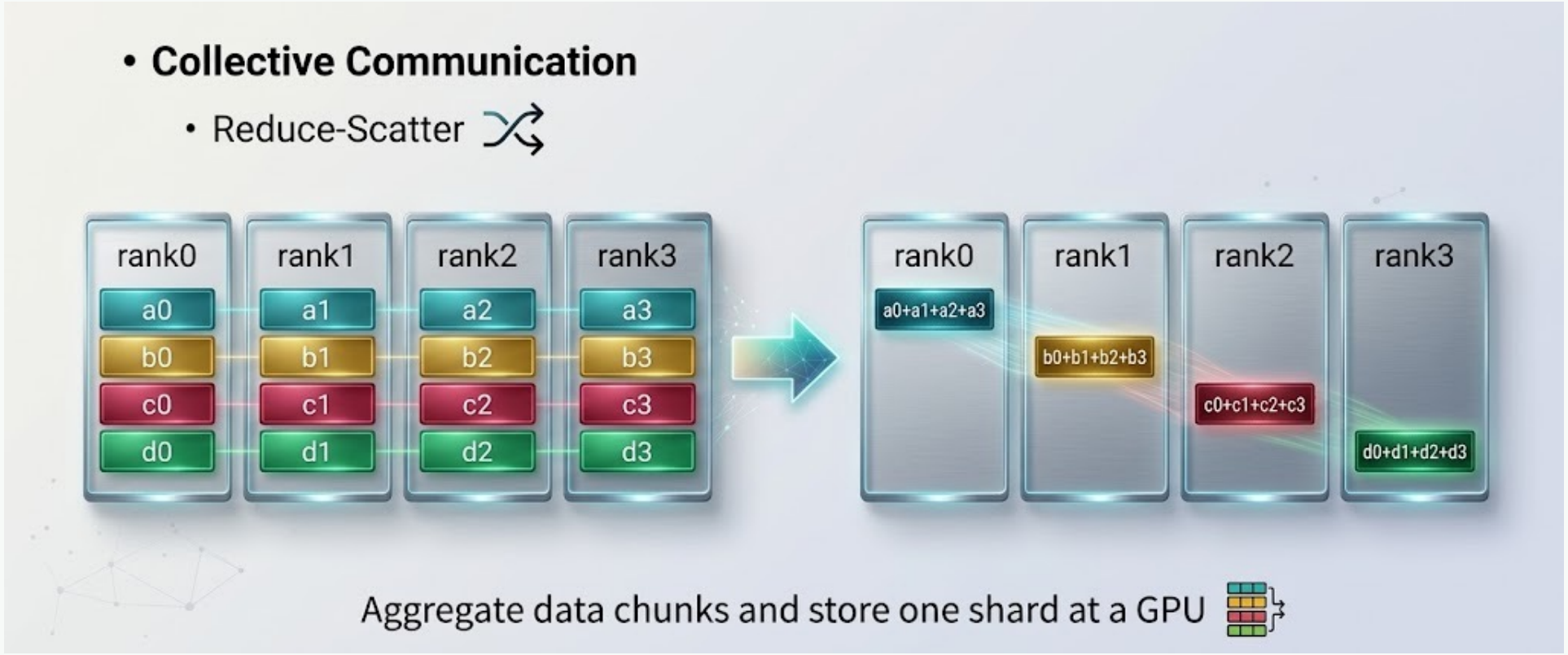
The collection is replicated across the group; values are copied, not reduced.



AllGather is the distribution phase used by several AllReduce algorithms.

ReduceScatter — reduce globally, distribute the result

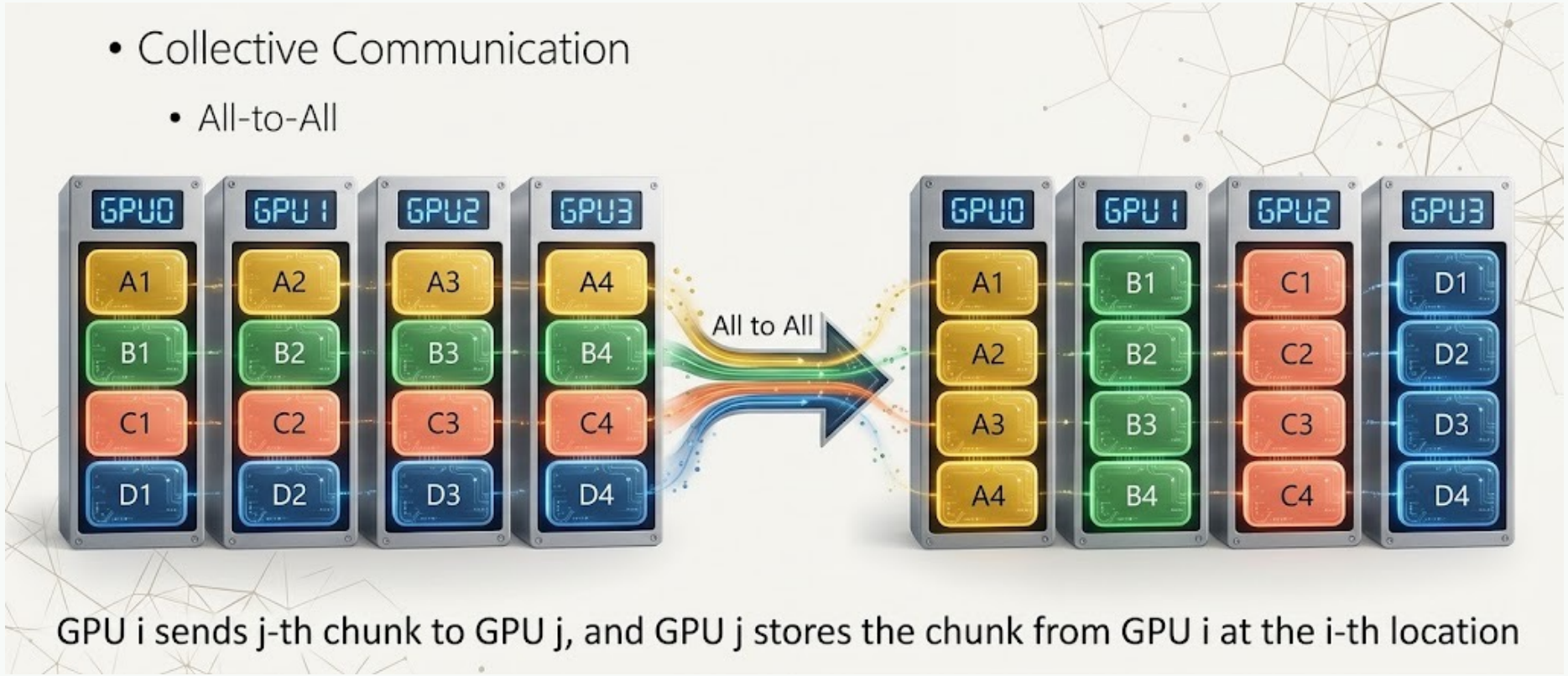
Each rank finishes with one fully reduced shard, not the complete tensor.



Ring AllReduce begins here: N − 1 rounds create N distributed reduced shards.

AllToAll — a different chunk for every destination

Rank i sends its j-th chunk to rank j; every pair exchanges distinct data.

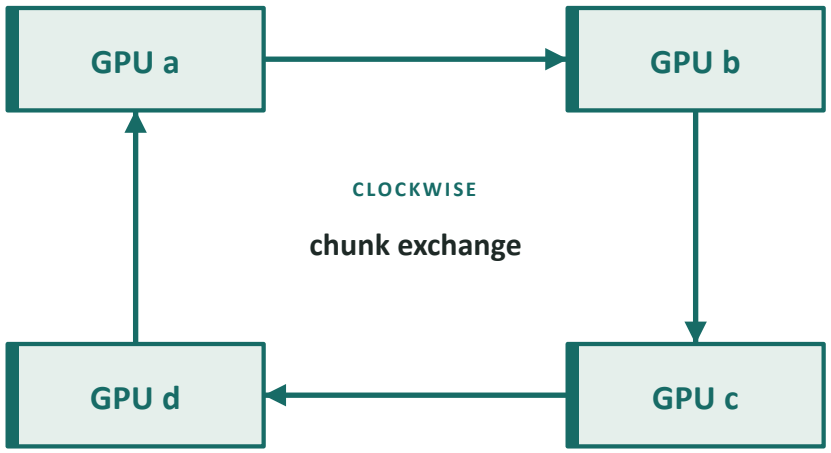


AllToAll redistributes personalized chunks; it does not aggregate or replicate them.

Starting from the most important “All-Reduce”

Ring all-reduce

Four GPUs • four gradient chunks per GPU



Example first rotation: GPU r sends chunk Cr to the next GPU.

INITIAL STATE / AFTER BACKPROPAGATION

Local gradients, partitioned before communication

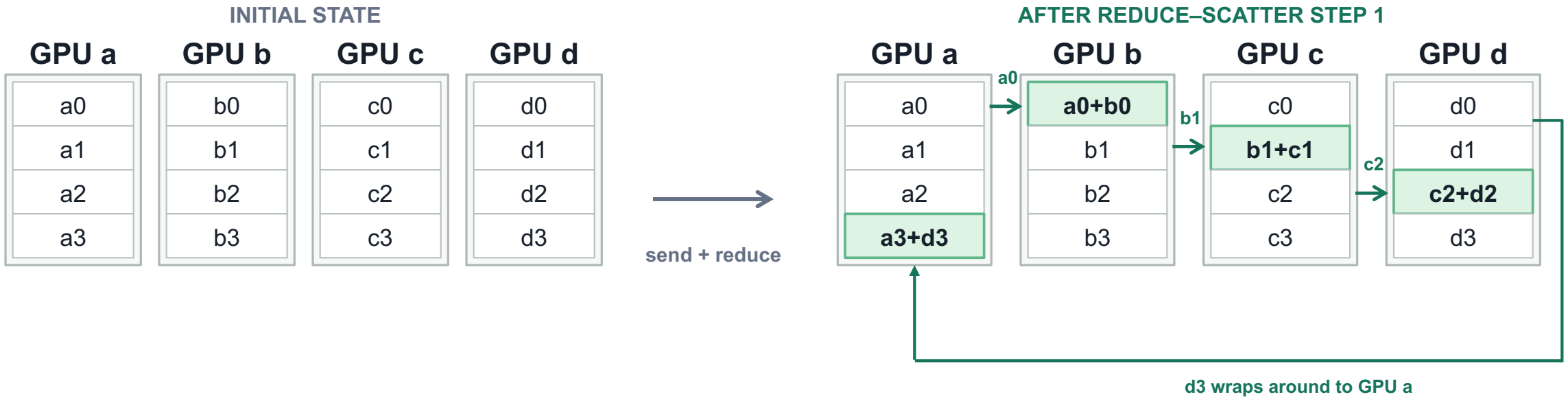
GPU a	GPU b	GPU c	GPU d
a0	b0	c0	d0
a1	b1	c1	d1
a2	b2	c2	d2
a3	b3	c3	d3
4 × M/4 bytes	4 × M/4 bytes	4 × M/4 bytes	4 × M/4 bytes

Each GPU holds one complete local gradient, subdivided into four equal chunks.

RING ALLREDUCE • FOUR-GPU EXAMPLE

Step 1: send one chunk, receive one chunk, then add

Each GPU sends a chunk clockwise; the receiver adds it to the same-indexed local chunk.



REDUCE–SCATTER

Repeat for 3 rounds. Each GPU ends with one fully reduced chunk.

ALL–GATHER

Circulate reduced chunks for 3 rounds. Every GPU receives all four results.

Ring AllReduce = ReduceScatter + AllGather • P = 4, so each phase needs P − 1 = 3 rounds

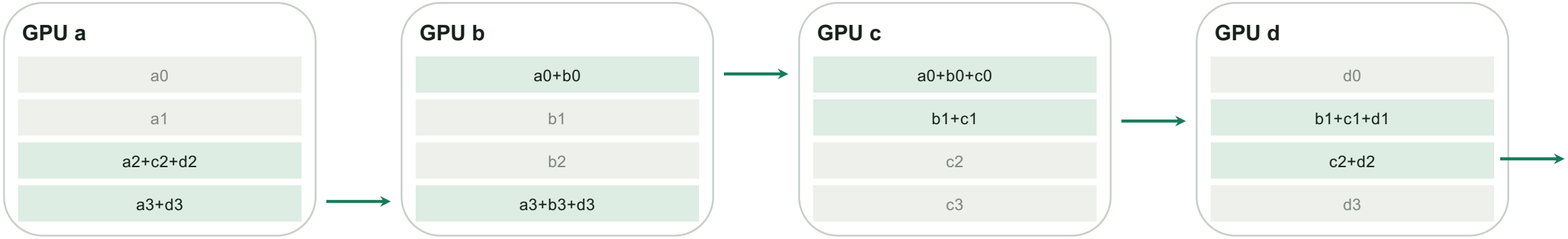
Step 2 and 3: add and forward

Also called scatter-reduce · the first pass of Ring AllReduce · after N−1 hops, every GPU owns one complete sum.

Local value only · Partial sum · Fully reduced

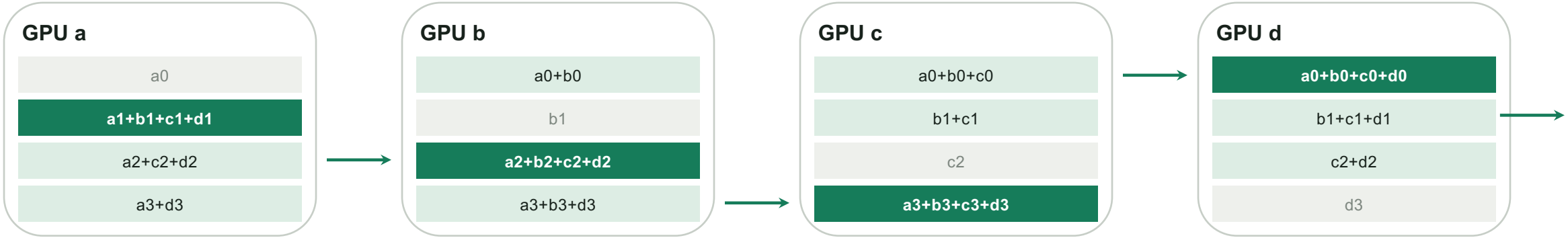
Ring direction: a → b → c → d → a

Step 2 · PARTIAL SUMS ARE STILL CIRCULATING



NEXT HOP · each GPU sends the next partial sum around the ring

Step 3 · EACH GPU FINISHES ONE CHUNK



1 · RECEIVE + ADD

Add the incoming partial sum to the matching local chunk.

2 · FORWARD THE NEXT

Send another partial sum; keep all N chunks in circulation.

3 · READY FOR ALL-GATHER

Each GPU now owns one fully reduced chunk of size S/N.

All-Gather: copy reduced chunks until every GPU is complete

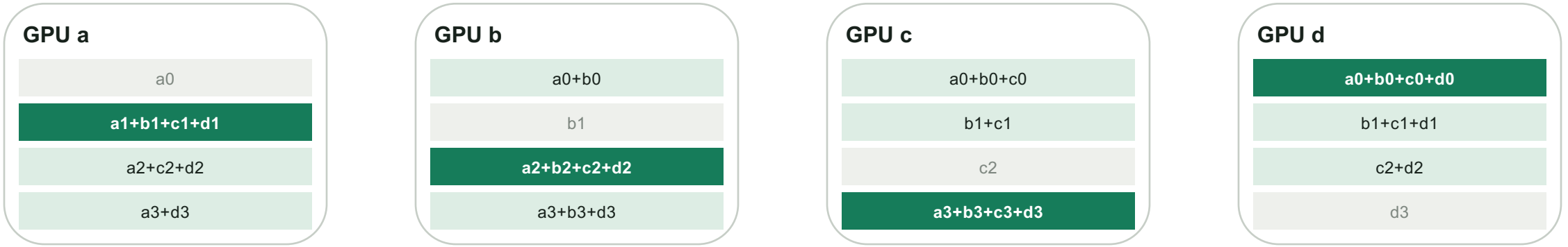
The second pass of Ring AllReduce · one new finished chunk per GPU per step · no further addition is needed.

Fully reduced chunk held here

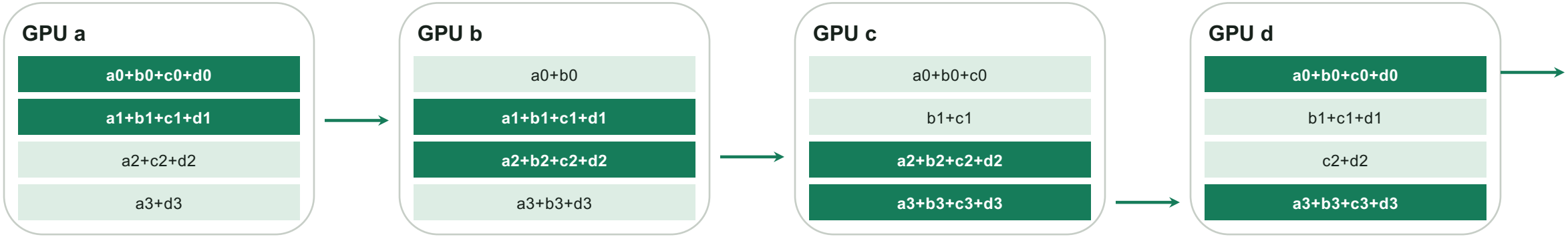
Not yet fully available on this GPU

Ring direction: a → b → c → d → a

Before Step 1 · ONE FULLY REDUCED CHUNKS PER GPU



Step 1 · ONE FULLY REDUCED CHUNKS PER GPU



NEXT STEP · forward the chunk received in the previous step

1 · FORWARD

Send the finished chunk received in the previous hop.

2 · COPY, DON'T REDUCE

The incoming values stay unchanged; no addition occurs.

3 · ONE FINAL STEP

Step 3 fills the last unfinished row on every GPU.

All-Gather · Step 3: the final step completes every GPU

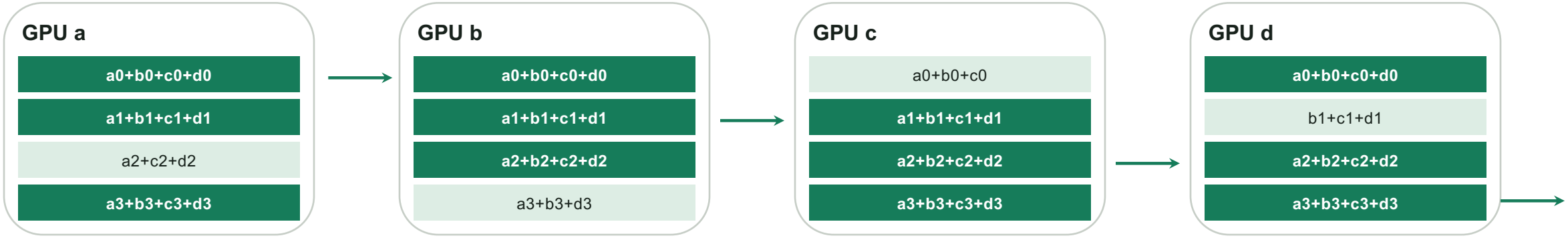
Each GPU receives its last missing reduced chunk; afterward, all four GPUs hold exactly the same tensor.

Fully reduced chunk held here

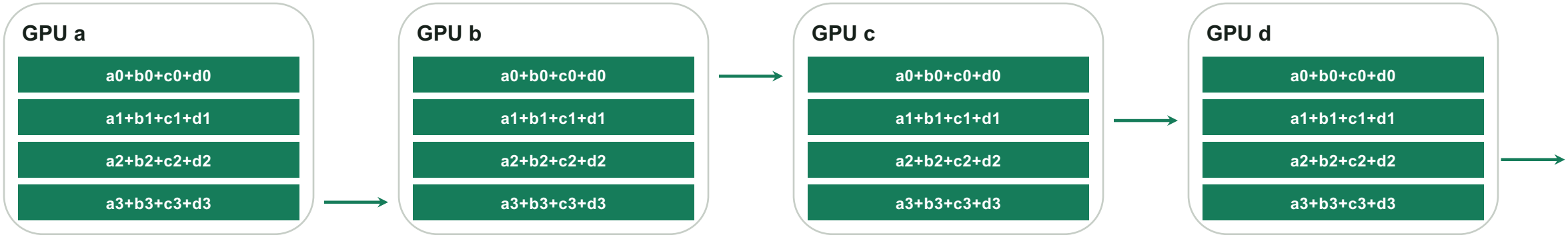
Last chunk still missing

Ring direction: a → b → c → d → a

Step 2 · THREE FULLY REDUCED CHUNKS PER GPU



Step 3 · EVERY GPU HAS THE COMPLETE REDUCED TENSOR



IDENTICAL OUTPUT

Every GPU holds reduced chunks 0, 1, 2 and 3.

COPY, DON'T REDUCE

Step 3 transfers values unchanged — no addition.

RING ALLREDUCE COMPLETE

The reduce-scatter + allgather is now commonly used.

RING ALLREDUCE · COST SUMMARY

Two phases, one predictable cost model

01 ReduceScatter

N − 1 rounds

Send S/N per round

Result: one reduced chunk per GPU

02 AllGather

N − 1 rounds

Send S/N per round

Result: complete tensor on every GPU

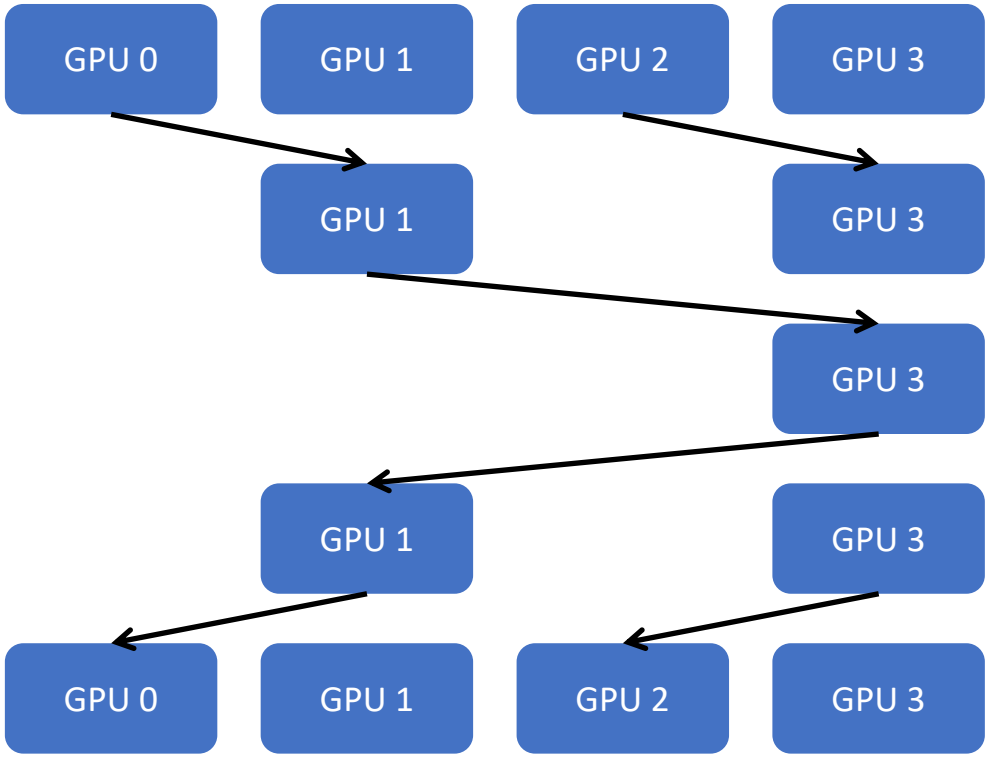
Rounds = 2(N − 1) · **Sent traffic / GPU = 2S(N − 1)/N**

Large-message bandwidth approaches 2S per GPU; startup count grows with N.

Best fit: large tensors and moderate GPU counts with identical ring bandwidth.

Ring All-Reduce = Reduce-Scatter + All-Gather

Tree AllReduce

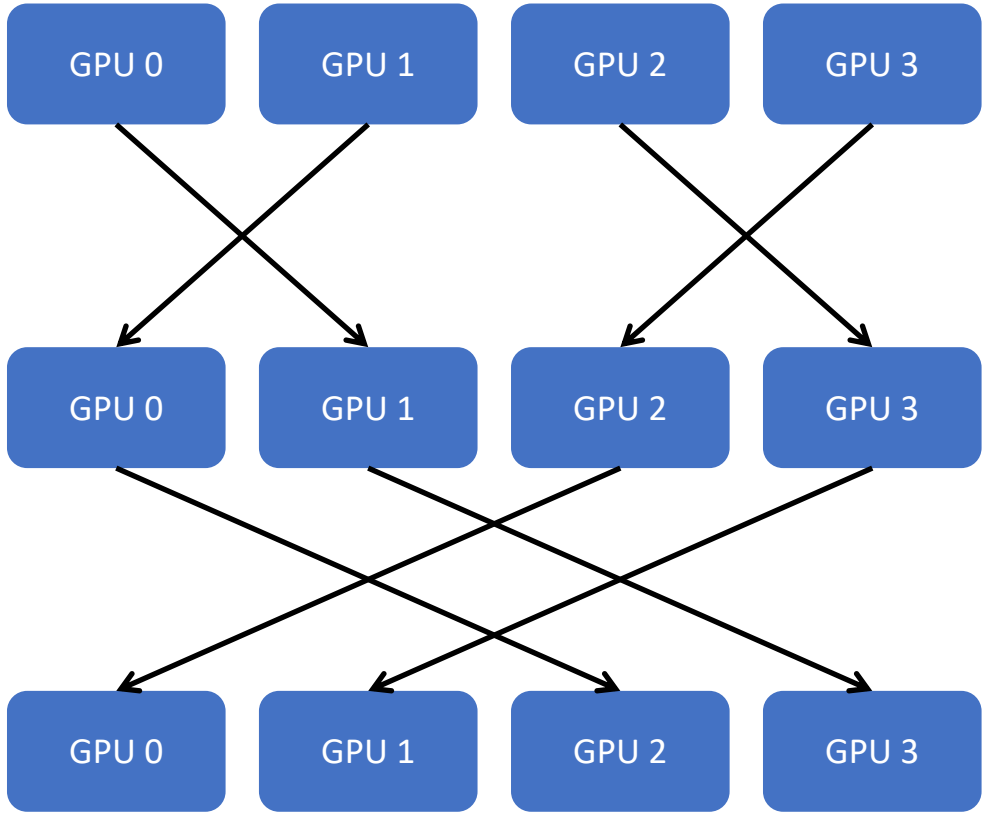


Data communication

- Upward
  - $\log_2 N$ rounds
  - At most S data per round
- Downward
  - $\log_2 N$ rounds
  - At most S data per round
- Total traffic per GPU
  - $2 * S * \log_2 N$ (naïve transmission without overlapping)

Recursive doubling AllReduce

At round k, rank r exchanges a full tensor with partner r XOR 2^k, then reduces.

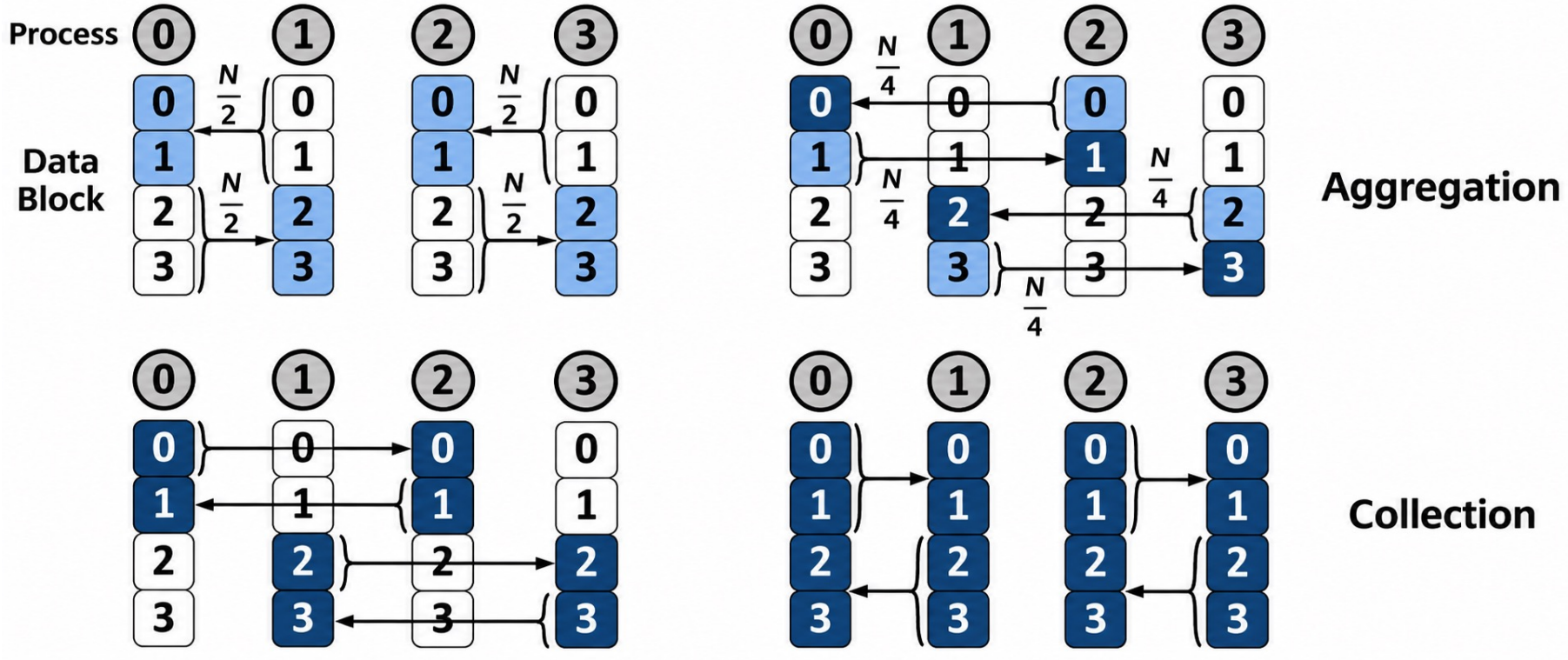


Butterfly Reduce: Utilizing bidirectional bandwidth

- Butterfly Reduce
  - $\log_2 N$ rounds
  - S data per round
- Total Traffic per-GPU
  - $\log_2 N * S$

Rabenseifner — recursive halving + recursive doubling

A logarithmic-step ReduceScatter followed by a logarithmic-step AllGather.



- **2 $\log_2$ N steps**
- $2\left(\frac{S}{2}+\frac{S}{4}+\cdots+\frac{S}{N}\right) \approx 2\frac{N-1}{N}S$ **sent per GPU**

CROSS-COMPARISON

Algorithm choice is a latency–bandwidth trade-off

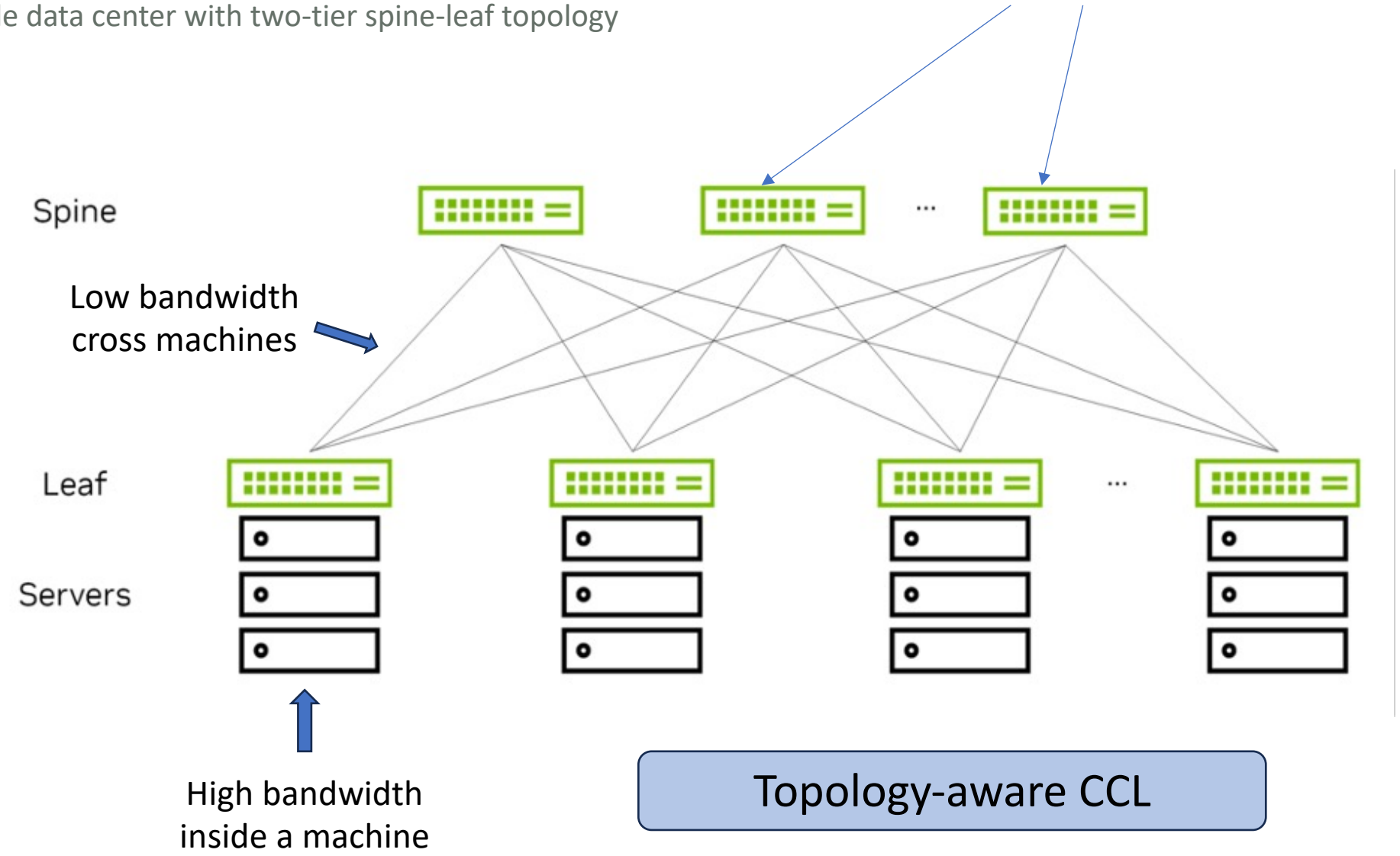
Idealized model: S = bytes per GPU, N = ranks, B = effective bandwidth; one transfer of size x costs $\alpha + x/B$.

Algorithm	Critical steps	Sent traffic / GPU	Strength	Main limitation
Ring	$2(N-1)$	$2S(N-1)/N$	Bandwidth-efficient; simple schedule	Startup latency grows linearly with N
Tree	$O(\log_2 N)$	Varies by tree position	Short critical path; topology-friendly	Possible hotspot near the root
Butterfly (recursive doubling)	$\log_2 N$	$S \log_2 N$	Balanced; few steps	Higher traffic for large tensors
Rabenseifner	$2 \log_2 N$	$\approx 2S(N-1)/N$	Efficient at larger scale	Partner pattern must suit topology

Small messages: prioritize fewer steps. Large messages: prioritize traffic efficiency and effective bandwidth.

Scaling out to a spine-leaf data center

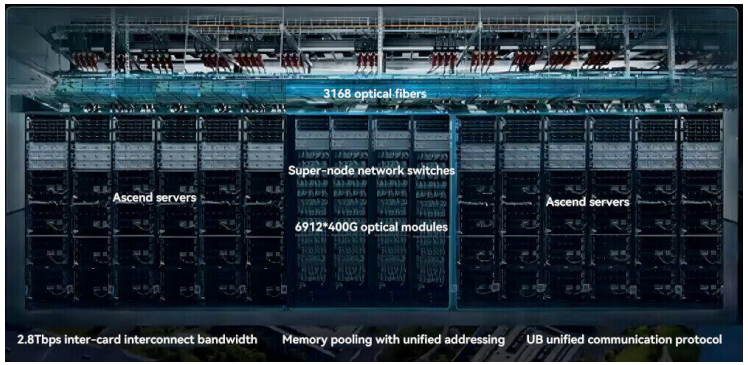
A large-scale data center with two-tier spine-leaf topology

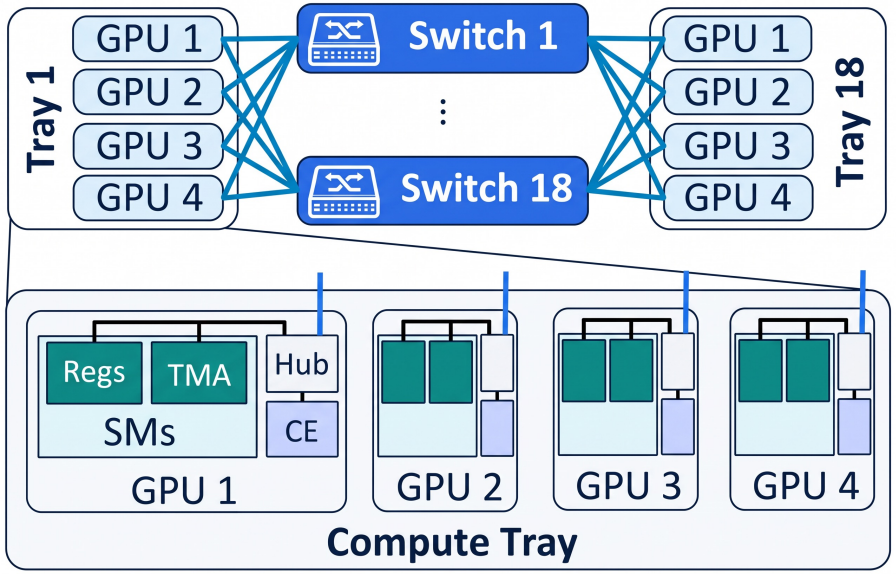


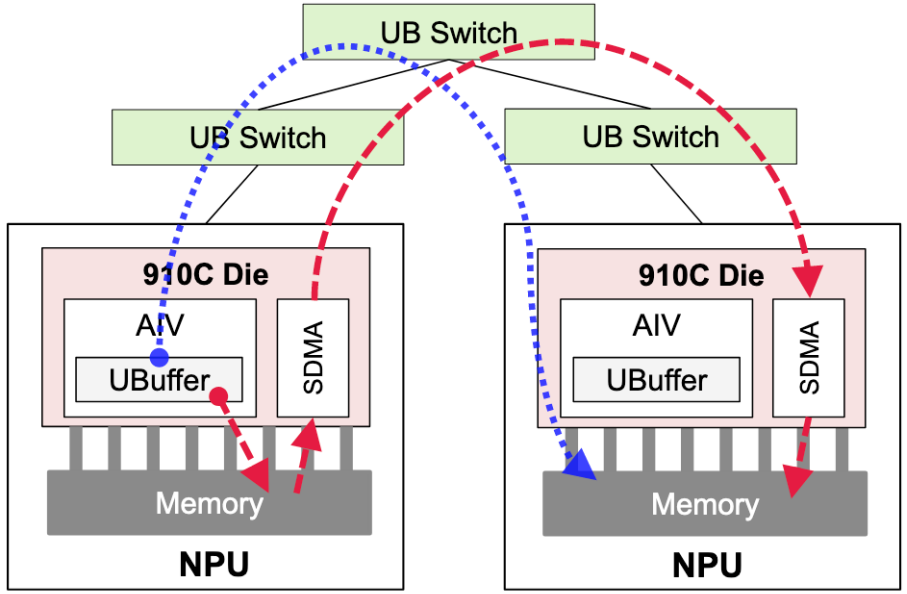
Scaling up to modern superpods

A large-scale data center with two-tier spine-leaf topology









LLM Parallel Training: Outline

- Data Parallelism
  - Parameter-Server
  - All-Reduce
  - **Memory optimization**

ROADMAP

From memory accounting to sharded training

01

Understand the memory budget

Weights, gradients, Adam state, activations, and temporary buffers

02

Remove replicated model states

ZeRO-1, ZeRO-2, and ZeRO-3 add sharding incrementally

03

Follow a ZeRO-3 training step

AllGather parameters, ReduceScatter gradients, update local shards

04

Extend capacity with offload

Use CPU memory and compute while managing transfer overhead

Core principle: shard persistent state; materialize full tensors only when computation requires them.

MEMORY ACCOUNTING

Training memory is more than model weights

Canonical mixed-precision Adam accounting used in the ZeRO paper

16 bytes per parameter · model states only



Model states

Parameters + gradients + optimizer state

Activations

Scale with batch, sequence, and checkpointing

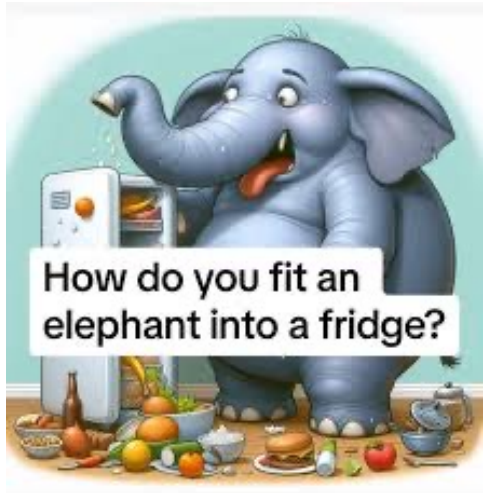
Workspace

Communication buffers, temporary tensors, fragmentation

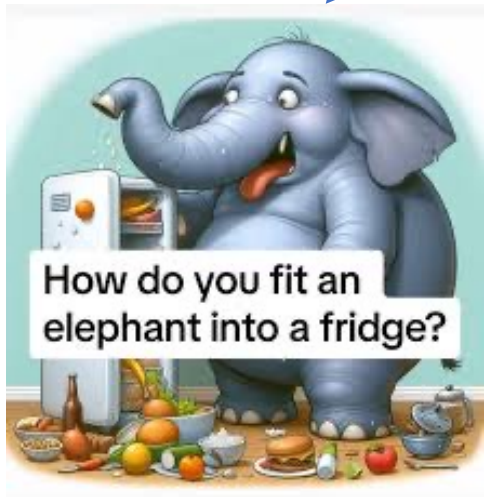
Source: Rajbhandari et al., ZeRO, SC 2020. Actual totals vary with precision and optimizer.

How to put an elephant into a fridge?

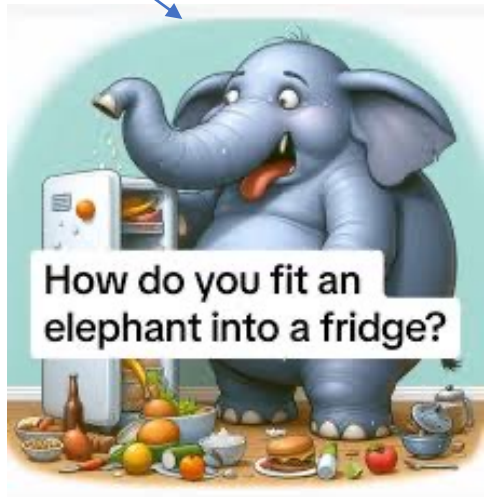
Optimizer States



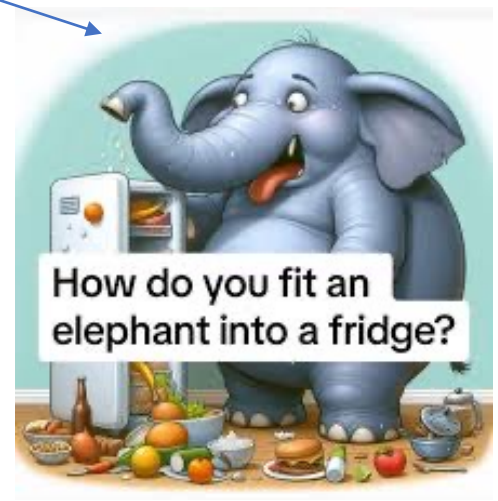
GPU0



GPU1



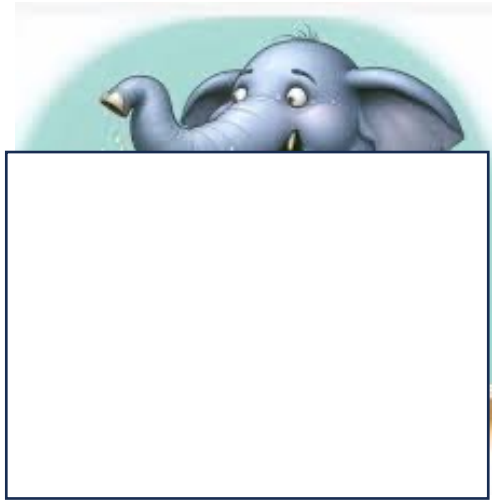
GPU2



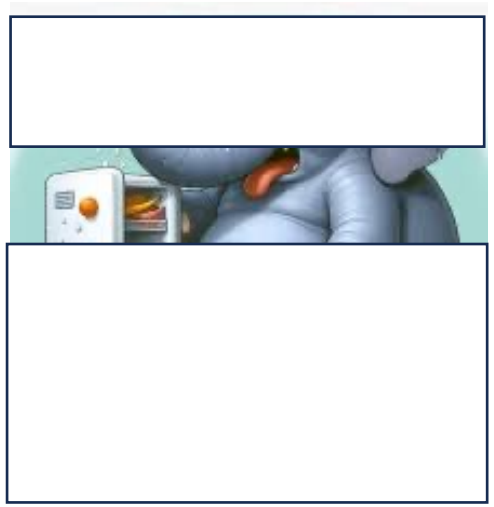
GPU3

How to put an elephant into a fridge?

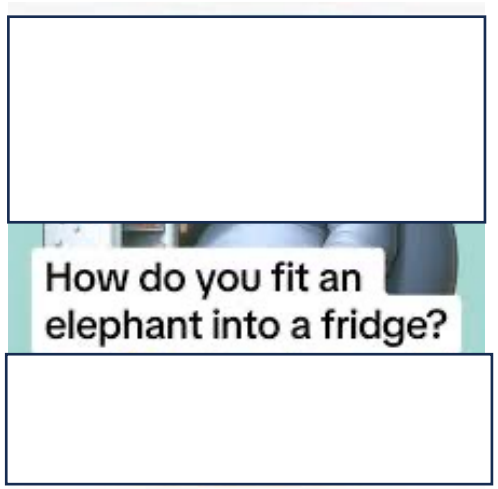
Sharding the elephant into four partitions, and each GPU HBM holds one chunk!



GPU0



GPU1



GPU2

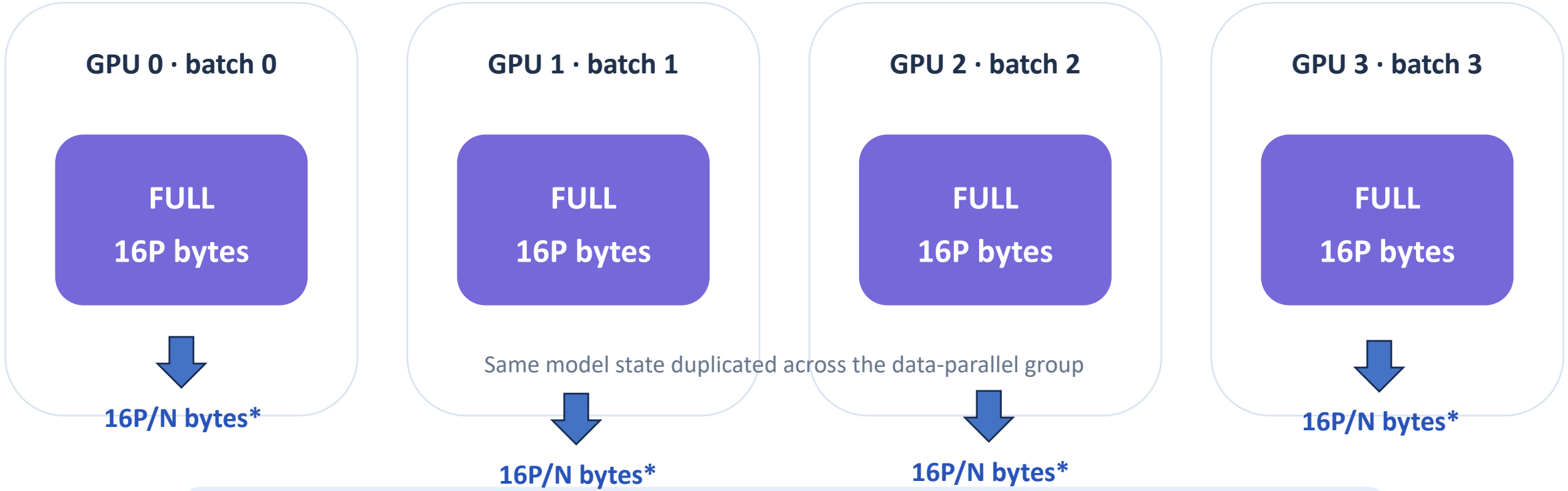


GPU3

THE REDUNDANCY PROBLEM

Data parallelism replicates the same state everywhere

Different data on each GPU; identical parameters, gradients, and optimizer state



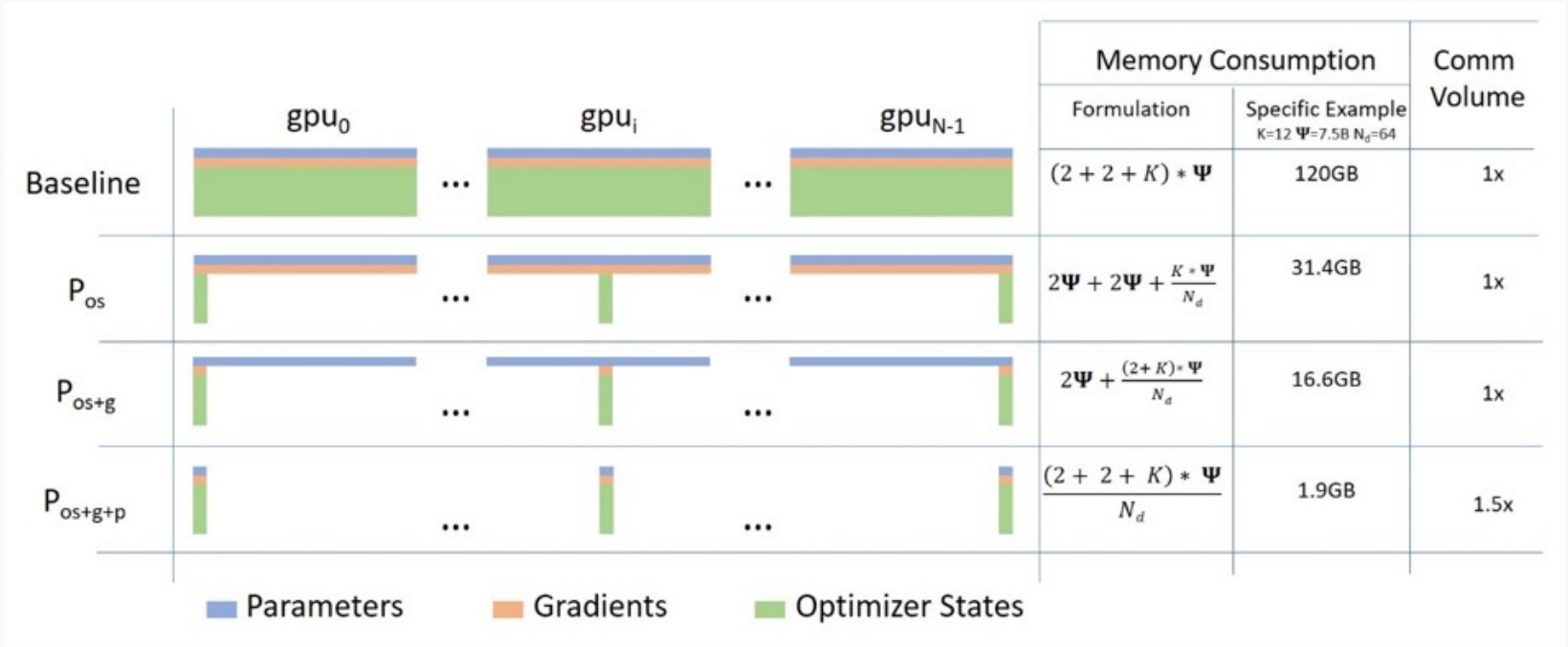
More GPUs do not reduce per-GPU model-state memory in standard DDP.

Three stages add sharding incrementally

ZeRO-1 · optimizer states

ZeRO-2 · + gradients

ZeRO-3 · + parameters



	Memory Consumption		Comm Volume
	Formulation	Specific Example (K=12 Ψ=7.5B N_d=64)	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

Source figure: ZeRO / DeepSpeed. Memory bars illustrate model-state sharding at data-parallel degree 64.

MEMORY ACCOUNTING

Model-state memory per GPU

P = parameter count · N = data-parallel degree · canonical 16-byte Adam example

Configuration	Parameters	Gradients + optimizer	Model states / GPU
Standard DP	2P replicated	14P replicated	**16P**
ZeRO-1	2P replicated	2P + 12P/N	**4P + 12P/N**
ZeRO-2	2P replicated	(2P + 12P)/N	**2P + 14P/N**
ZeRO-3	2P/N sharded	14P/N sharded	**16P/N**

Peak memory also includes activations, temporary gather buffers, communication workspace, and fragmentation.

As N goes infinity?

ZERO-1 AND ZERO-2 · TRAINING STEP

Full parameters enable local forward and backward

Gradient AllReduce is reorganized as **ReduceScatter** + parameter **AllGather**

1
Forward

Full parameters on every rank

2
Backward

Compute local gradients

3
ReduceScatter

Produce owned gradient shard

4
Local update

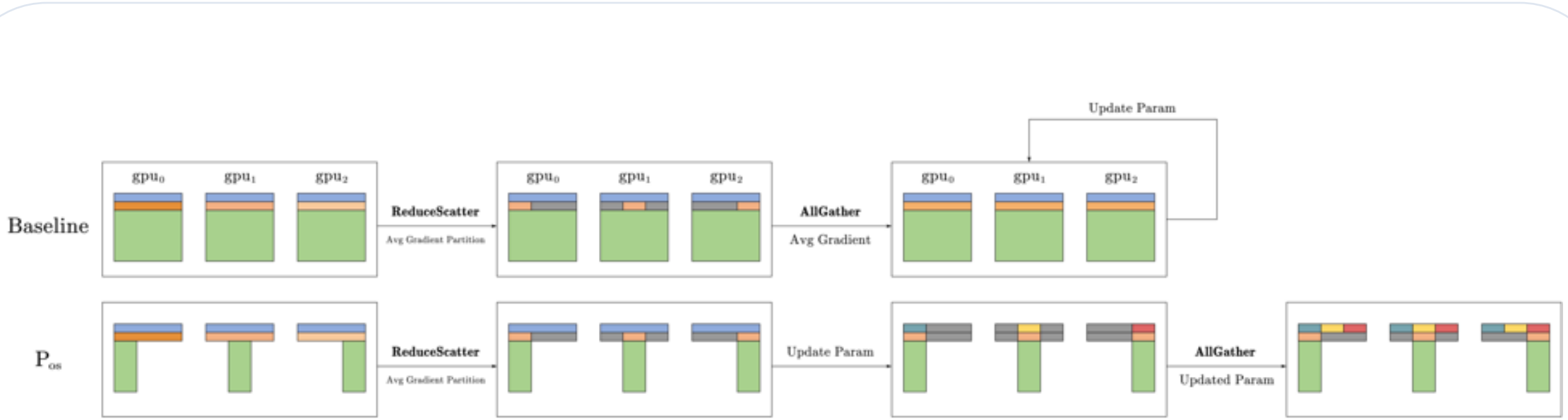
Update optimizer + weight shard

5
AllGather

Refresh full low-precision weights

ZeRO-2 additionally releases reduced gradient buckets during backward, lowering peak gradient memory.

ZERO-1: Update one shard; then refresh full parameters

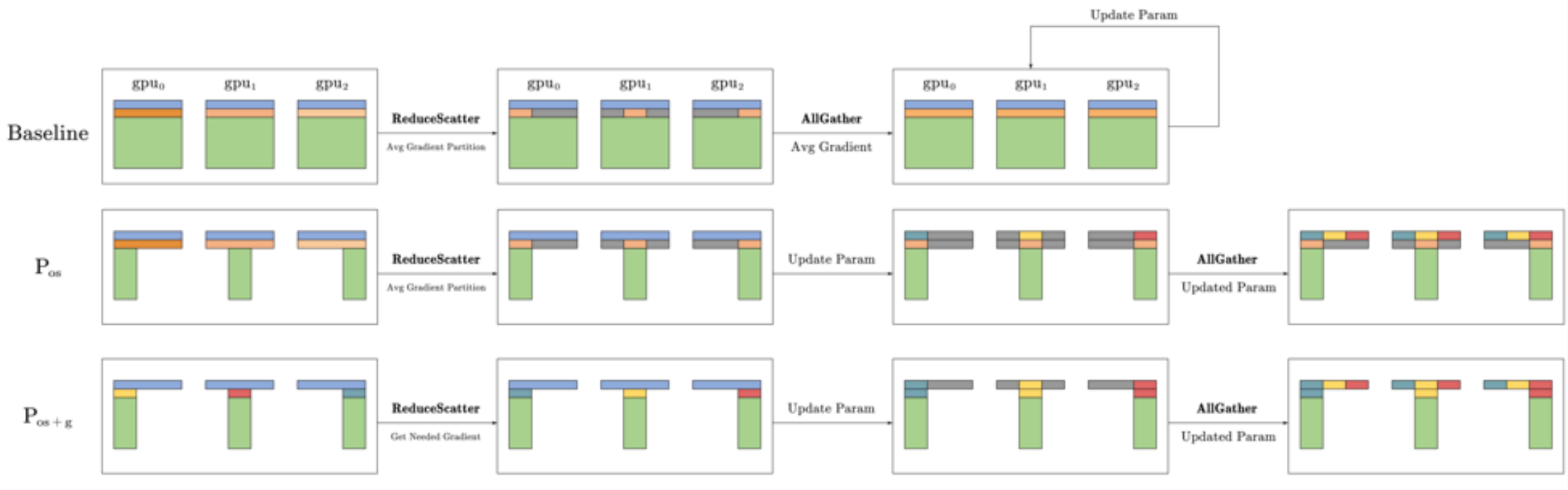


Replicated: parameters + gradients

Sharded: FP32 master + Adam moments

Memory: 4P + 12P/N

ZERO-2: Reduce gradient buckets and release replicas earlier



Replicated: parameters

Sharded: gradients + optimizer state

Memory: 2P + 14P/N

ZERO-3 · END-TO-END WORKFLOW

One training iteration in five stages

01

SHARD

Each rank owns parameters, gradients, and optimizer state for its partition

02

FORWARD

AllGather layer parameters, compute, release

03

BACKWARD

AllGather parameters again in reverse order

04

REDUCE

ReduceScatter gradients to their owners

05

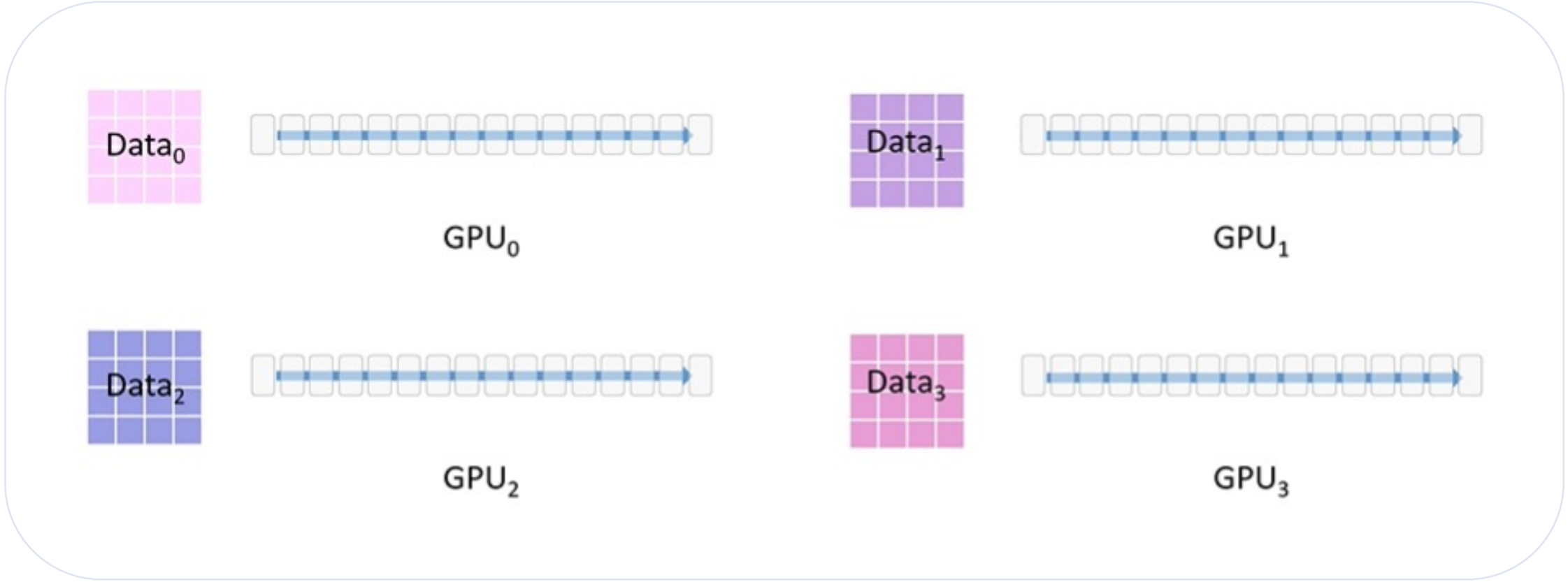
UPDATE

Update local state; leave parameters sharded

Full parameters are temporary working data—not persistent per-GPU model state.

Four GPUs, four different data shards

ZeRO-3 separately partitions model-state ownership



Each rank computes on a different mini-batch of the same global training step.

Each GPU owns one model-state partition

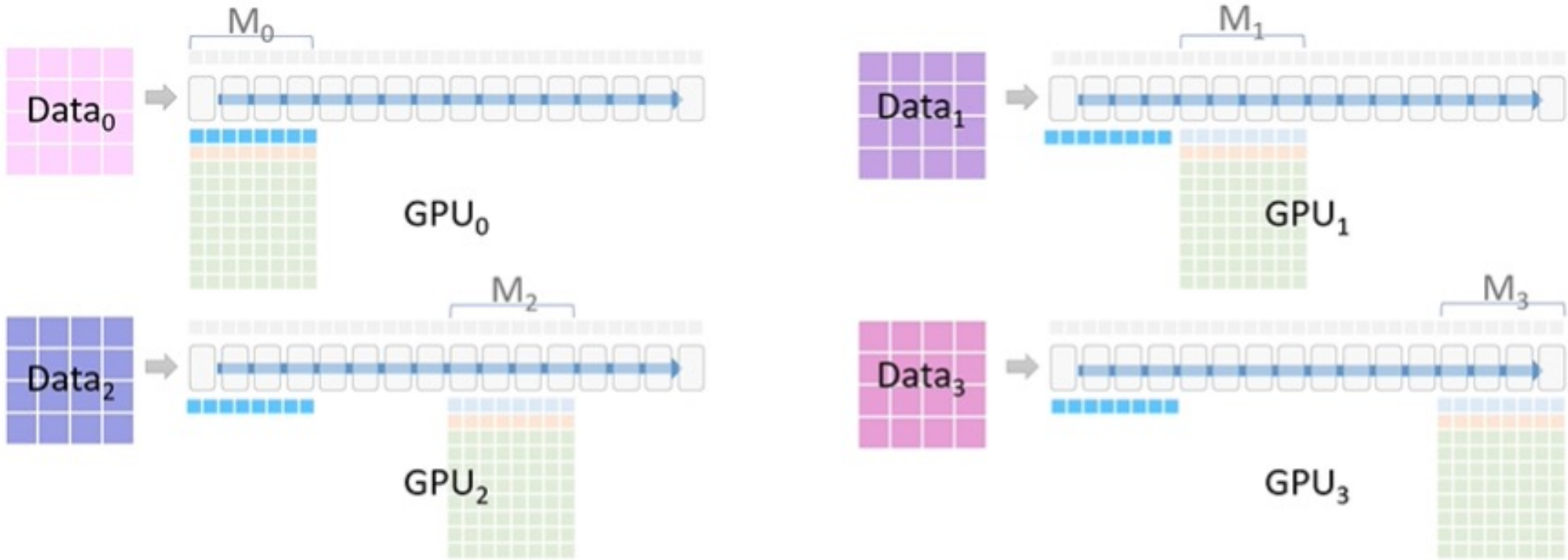
M0–M3 are conceptual partitions (NOT the real implementation!); actual systems may shard within layers



Persistent storage is partitioned; temporary gathers will reconstruct layers on demand.

Gather the parameters needed for the current layer

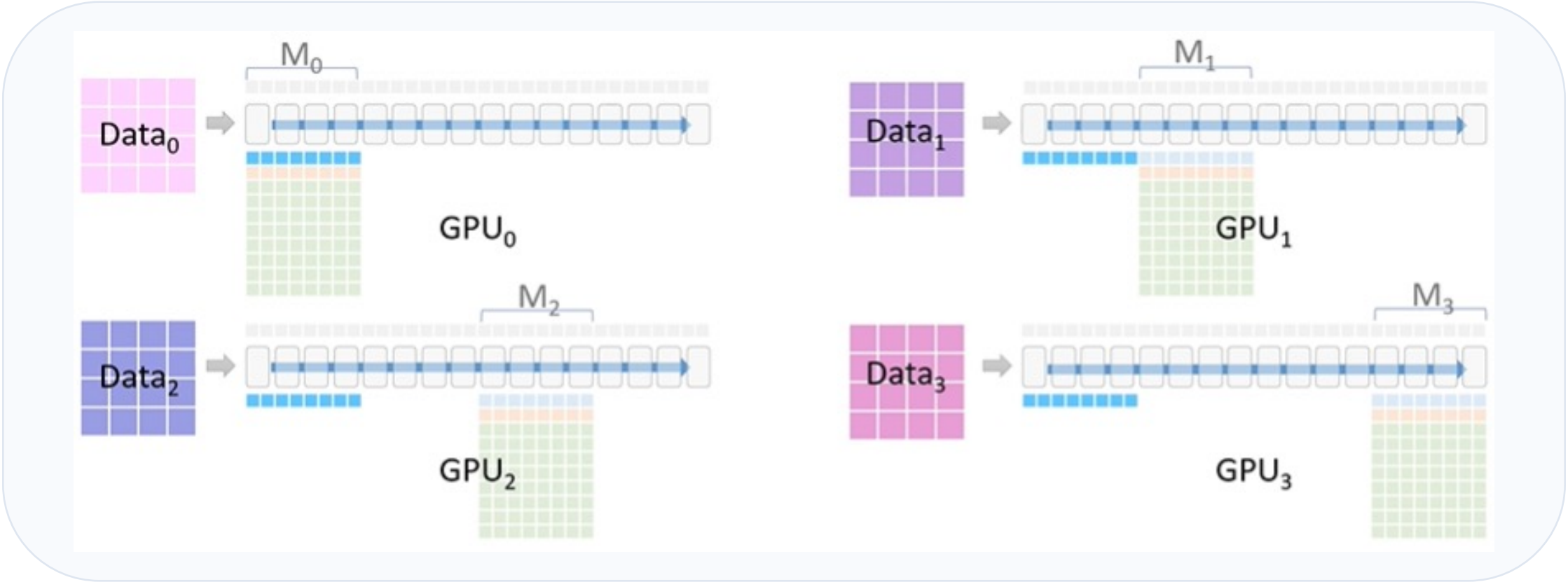
Generalized implementations use AllGather across shards



The gathered M0 tensor is temporary working memory, not a new persistent replica.

Compute locally, then release the gathered buffer

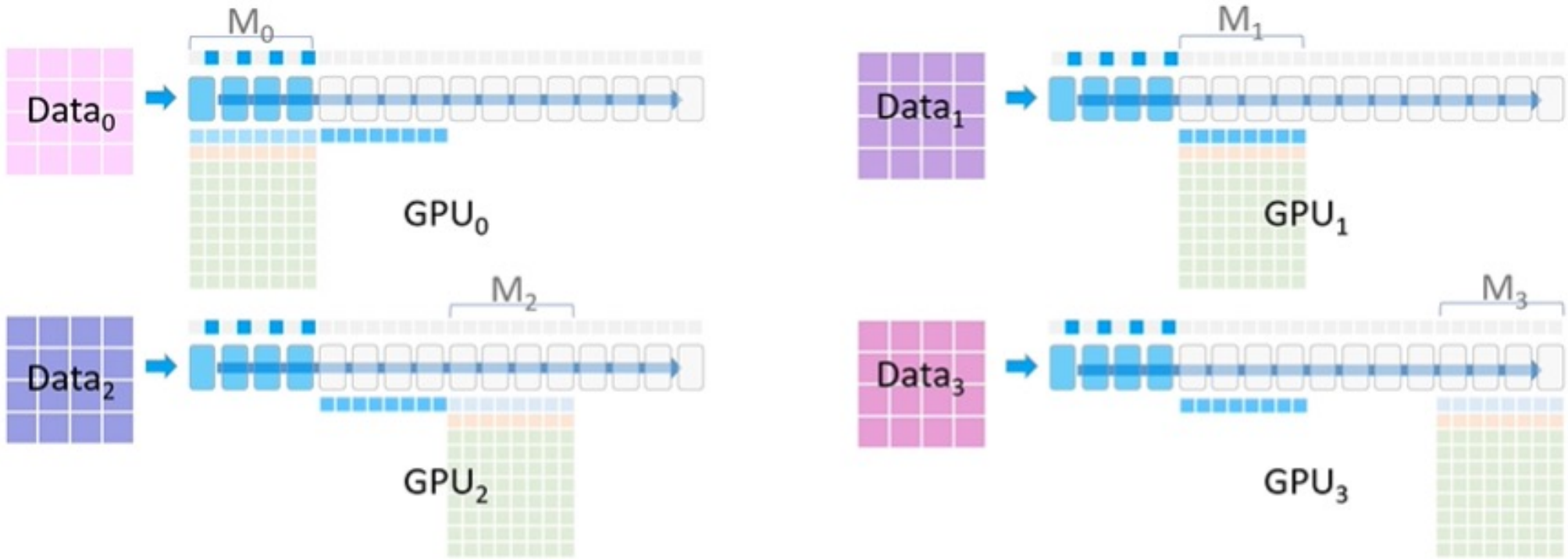
Only the required working parameters and activations need to remain live



Prefetch and compute–communication overlap help hide the next parameter gather.

Repeat the gather–compute–release cycle

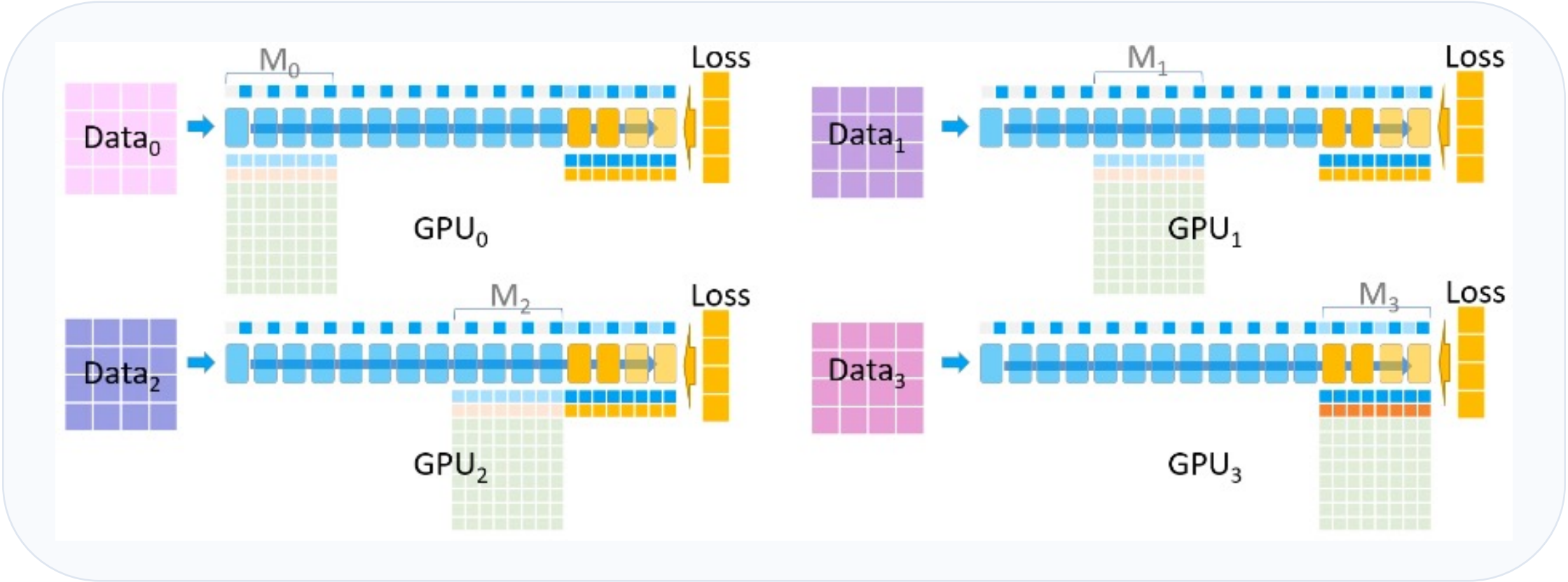
Each subsequent partition is materialized only when forward computation needs it



Peak memory depends on shard size, live layer buffers, prefetch depth, and activations.

ReduceScatter gradients to their owners

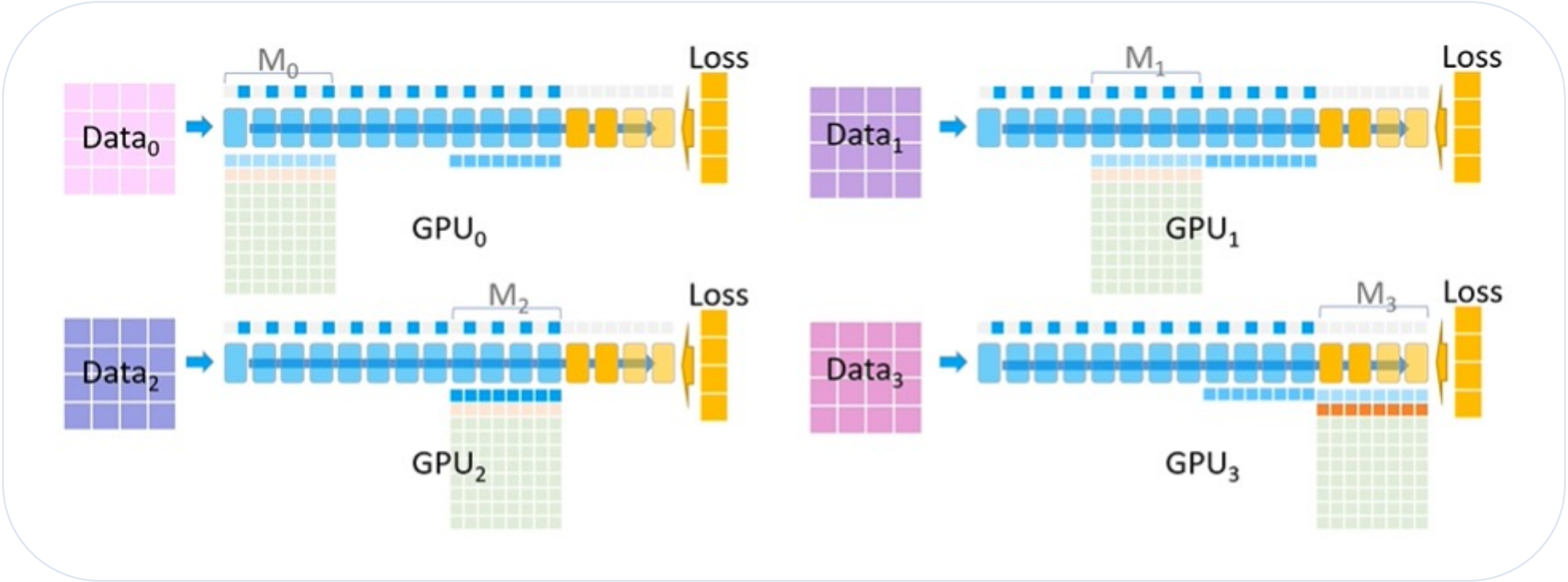
Global gradient reduction and shard distribution happen in one collective



GPU 3 receives the reduced M3 gradient shard needed for its local optimizer update.

Gather for compute; ReduceScatter for ownership

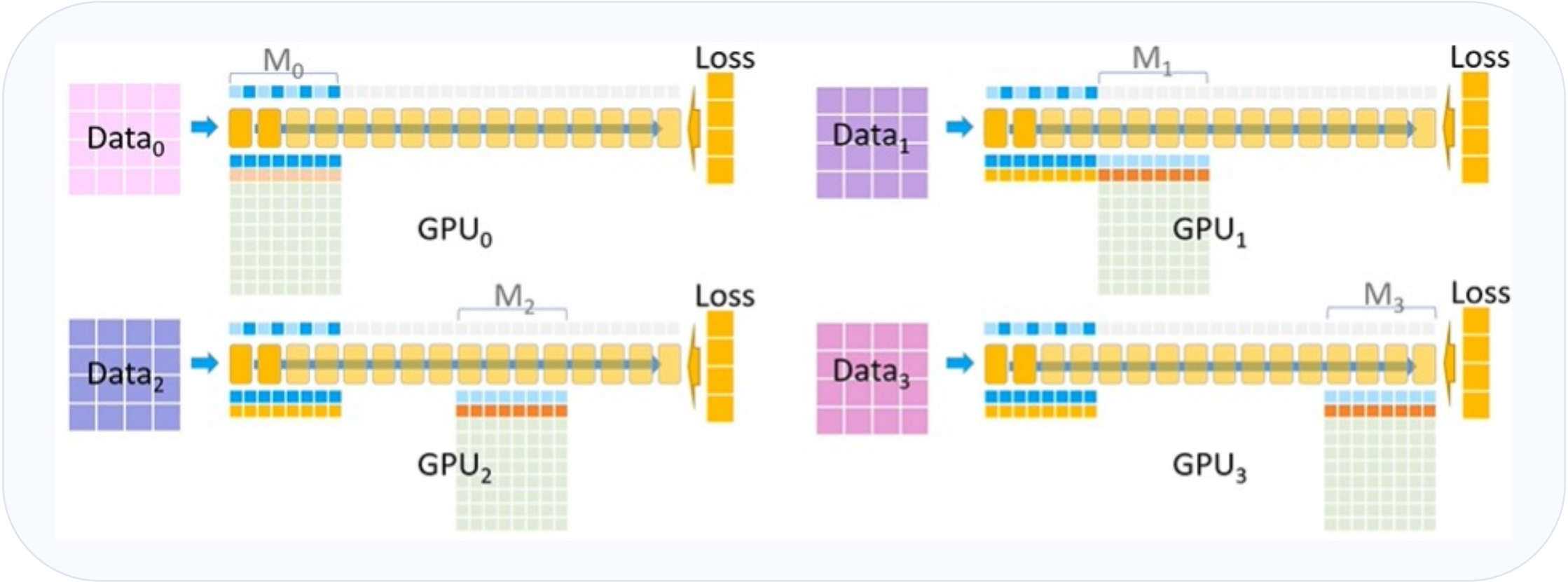
Parameter communication and gradient communication serve different roles



GPU 2 now owns the globally reduced M2 gradient shard.

Finish backward with reduced shards on every rank

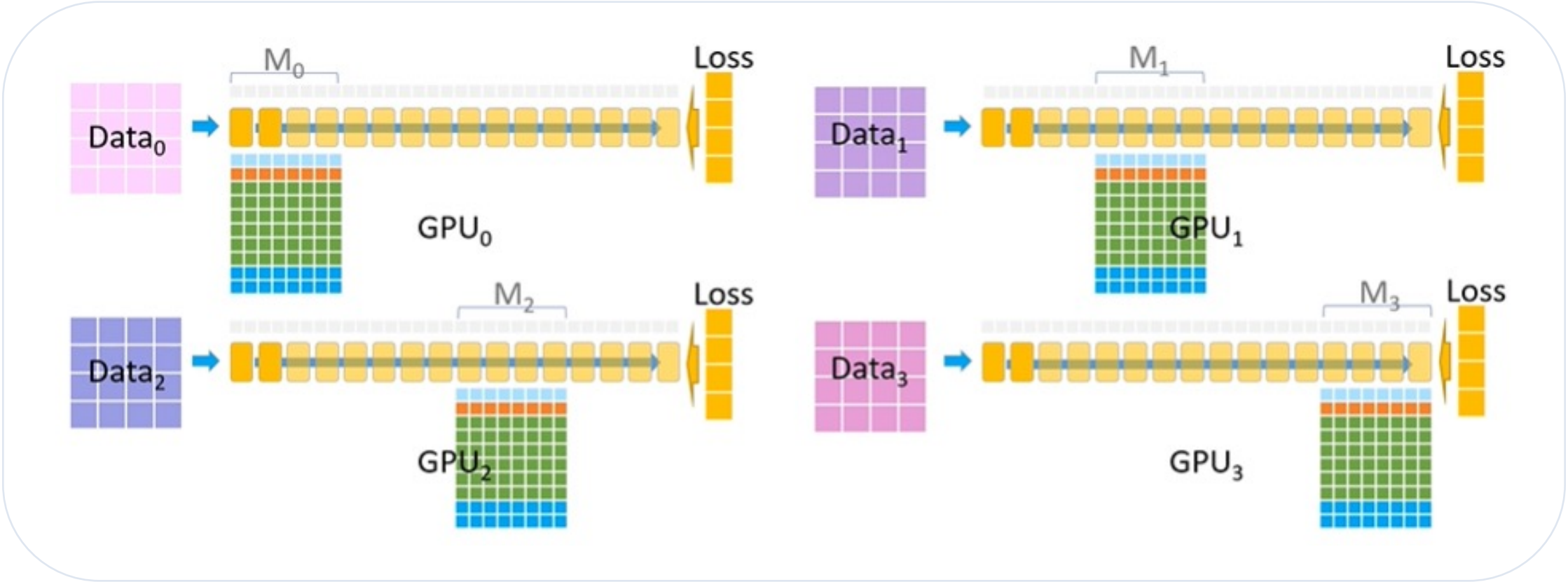
Parameter, gradient, and optimizer-state ownership are aligned by partition



No rank needs to retain a complete reduced gradient tensor.

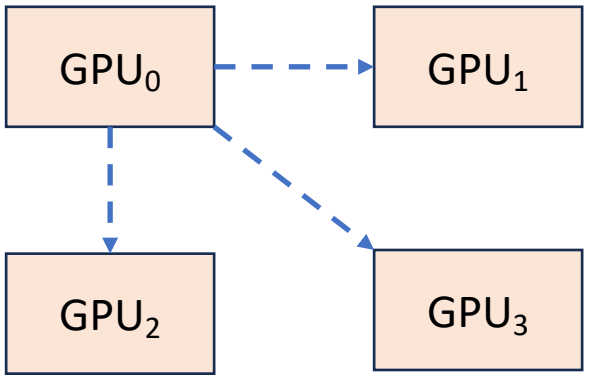
Update local shards in parallel—no final full-model gather

Updated parameters remain partitioned until the next on-demand gather

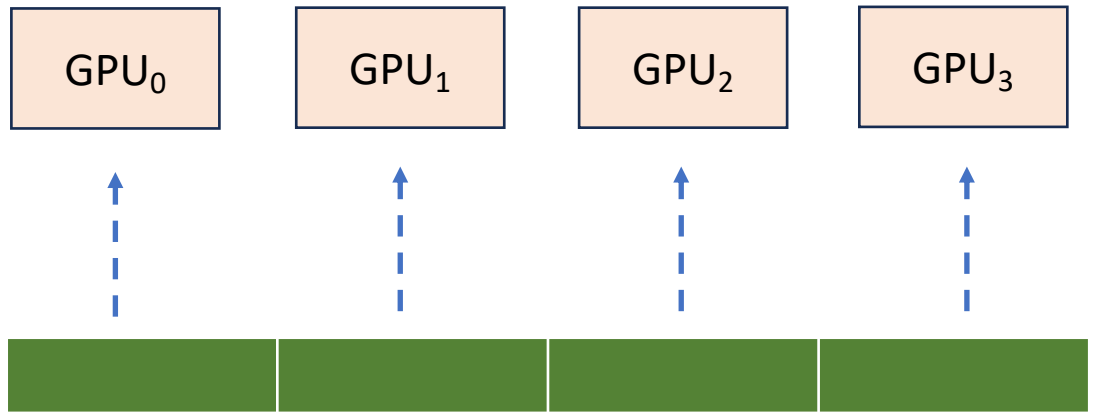


The next iteration gathers updated parameters on demand, preserving the memory savings.

Intra-layer partitioning instead of inter-layer partitioning

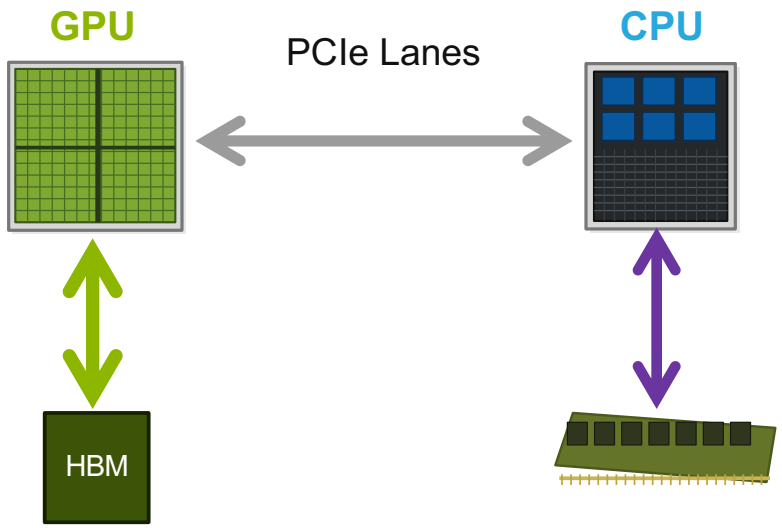


AllGather other than Broadcast



Intra-layer partitioning other than inter-layer partitioning

When sharding is not enough, add another memory tier

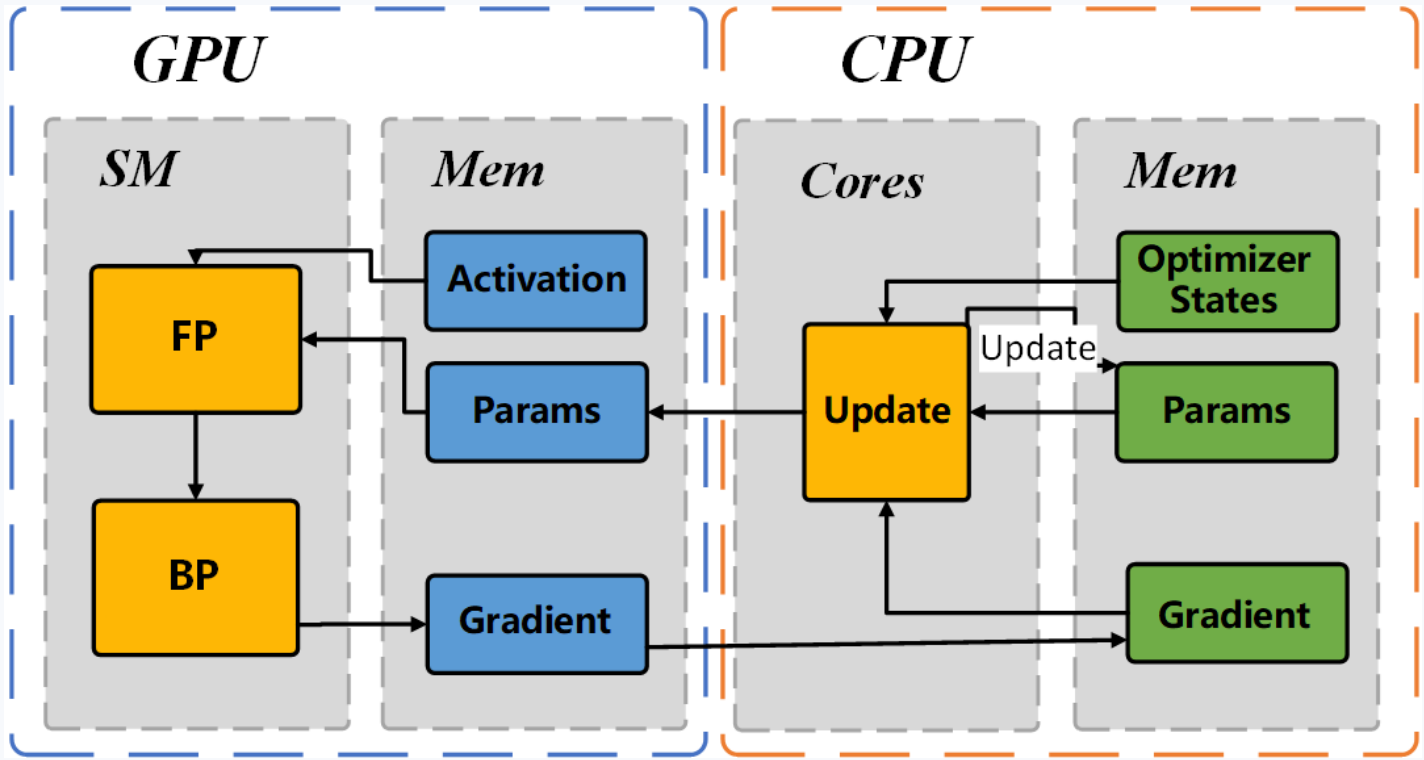


- CPU memory
  - ~1TB vs 80GB HBM
- Connection speed
  - PCIe 5.0 (16 lanes) ~ 64GB/s
  - NVIDIA H100 NVLink 4.0 ~ unidirectional 450GB/s
- GPU←→CPU bottleneck

Offload trades data movement for capacity; it is not free memory.

GPU computes the network; CPU can own the optimizer

Original ZeRO-Offload keeps forward/backward on GPU and moves optimizer state + update to CPU



GPU

FP16/BF16 compute, parameters, gradients, activations

HOST LINK

Gradients to CPU; updated parameters back to GPU

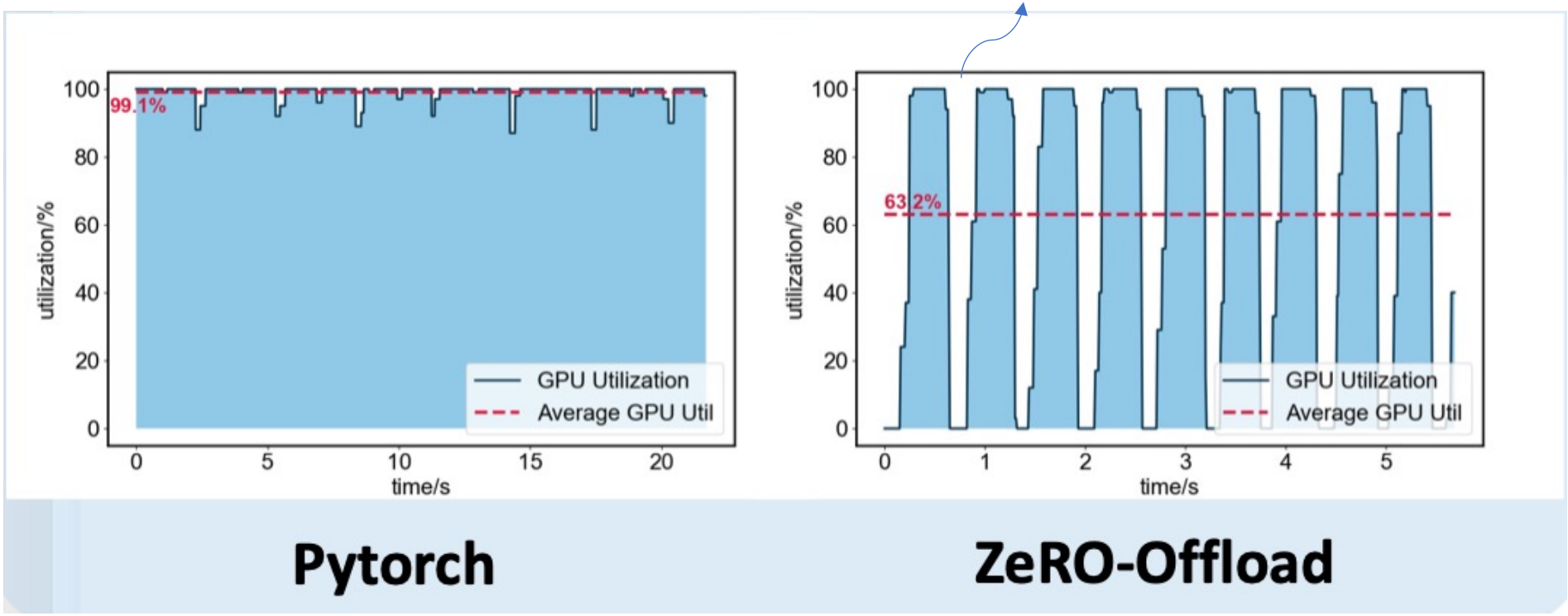
CPU

FP32 master weights, Adam moments, optimizer update

Pros and Cons

- Training GPT2 on A100: reduced GPU utilization

Where are these bubbles coming from?



Any more offloading

- Offloading activations
  - How to execute forward propagation, back propagation and gradient synchronization?
  - Interference between offloading traffic and ZERO-3/FSDP traffic?
- Offloading parameters
  - How does it work?
- Offloading to SSD
  - ZERO Infinity

MEMORY LANDSCAPE

Recomputation targets activations—not model states

Different memory components require different optimization techniques

MODEL STATES

Parameters

Gradients

Optimizer states

Typical optimizations

ZeRO / FSDP · sharding · offload

ACTIVATIONS + WORKSPACE

Layer inputs and outputs

Attention scores and masks

Temporary communication buffers

Typical optimizations

Recomputation · sequence parallelism · kernel fusion

Activation memory depends on the input workload and on which intermediate tensors remain live.

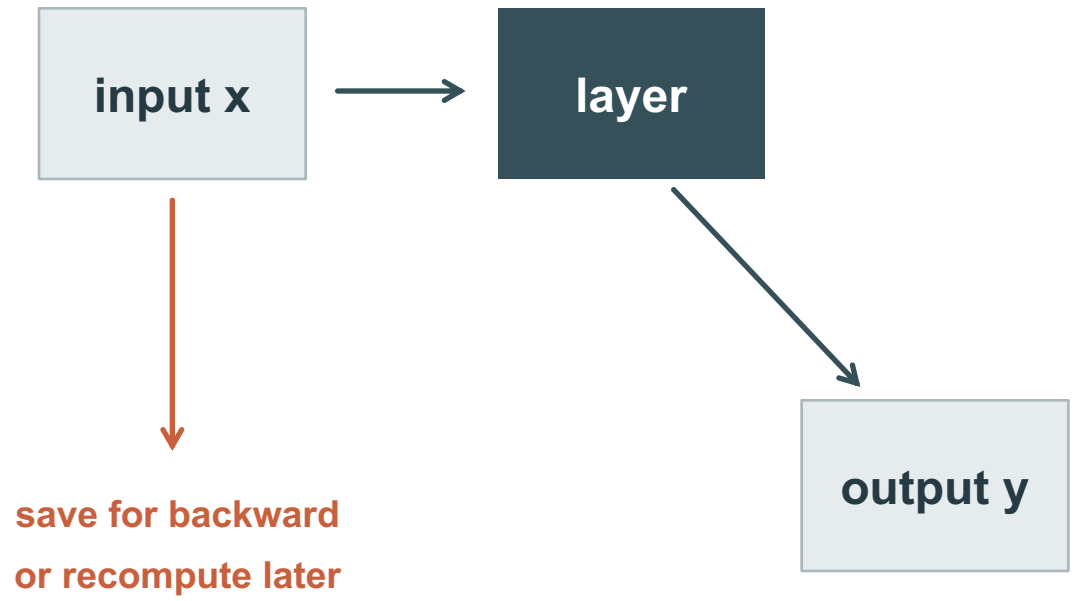
Backward propagation reuses forward intermediates

Forward: $y = Wx + b$

Backward: $\partial L/\partial W = (\partial L/\partial y)\, x^T$

The weight gradient needs x.

If x was discarded, it must be reconstructed.



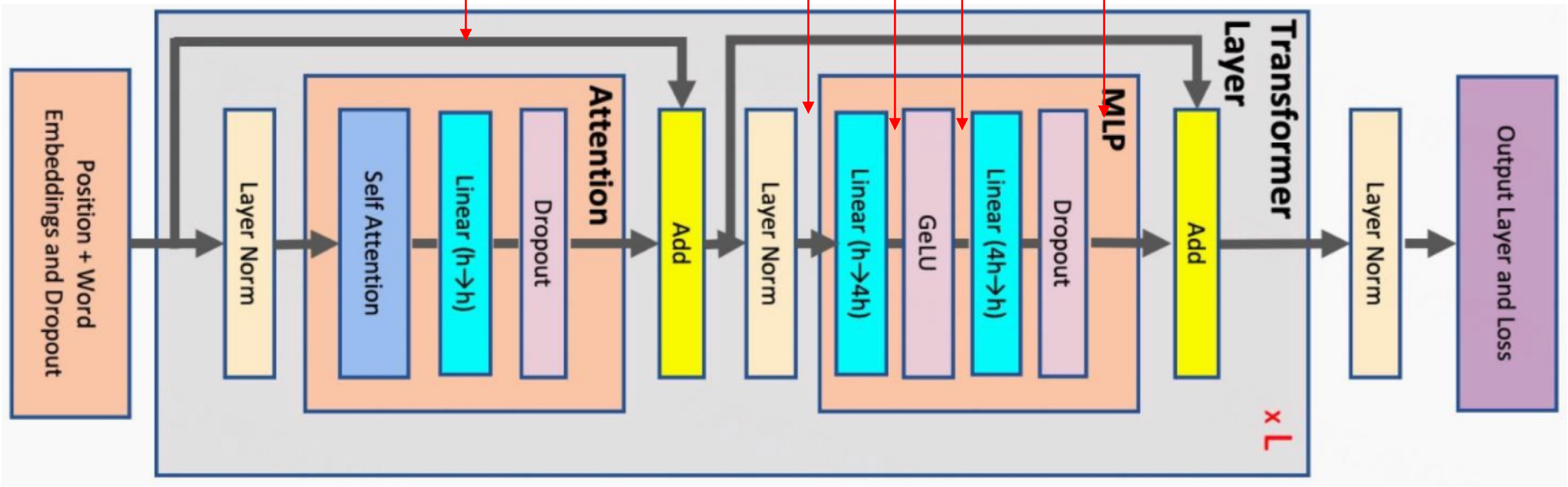
Checkpointing decides which forward values to keep and which to recreate during backward.

Attention adds a sequence-dependent memory term

11bsh+5assb 2bsh 8bsh 8bsh bsh

Activation bytes / layer ≈ sbh (34 + 5as/h) = 34sbh + 5as²b

s: sequence length · b: micro-batch size · h: hidden size · a: attention heads · 16-bit activations



34sbh

Main layer activations; broadly linear in s

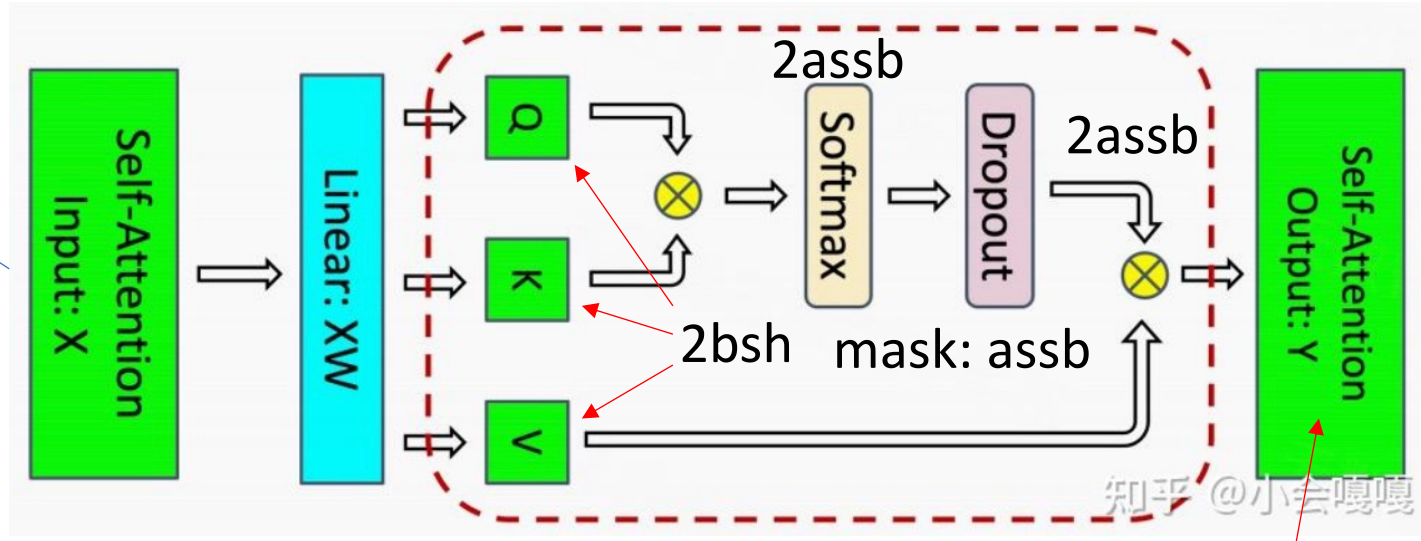
5as²b

Attention-related activations; quadratic in s

Source: Korthikanti et al., Reducing Activation Recomputation in Large Transformer Models, MLSys 2023.

Where does 34 come from?

bsh * 2 Bytes



3sbh: Linear layer
+ dropout layer

2bsh

In-total: 5abss + 11sbh

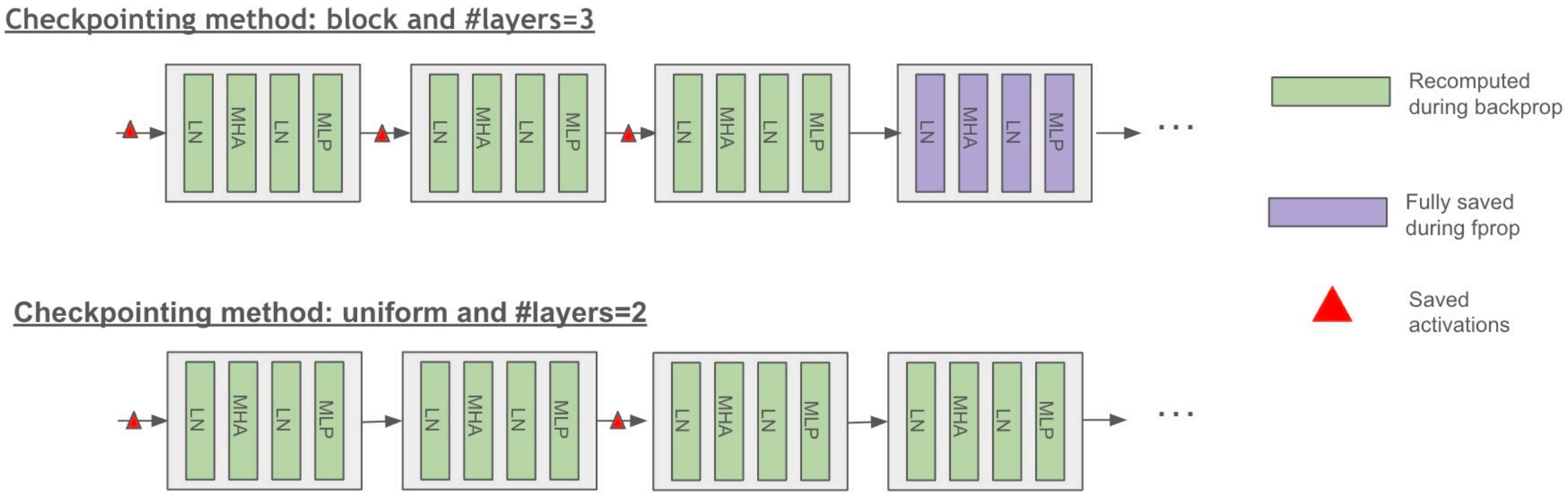
- Activation is non-negligible
  - b: 1 M tokens/sequence length, s: 2048 tokens, h: 12288, L: 96 layers
  - Activation size: 34bsh = 34*(1 M/2048)*2048*12288*96
  - Full activation re-computation: only keeping the initial input and recompute everything
    - Minimal memory occupation
    - Prolonged training time (doing forward propagation once again) by 30%~40%
  - Goal of strategic recomputation (not the scope of this class)
    - Significantly reducing memory occupation while slightly increasing training time
    - Softmax and Softmax dropout are more suitable to be re-computed

https://lambda.ai/blog/demystifying-gpt-3 《Reducing Activation Recomputation in LLMs》

CHECKPOINT PLACEMENT

Checkpoint placement controls each recomputation group

Block and uniform policies distribute saved boundaries differently across the network



Fewer checkpoints → less saved state, larger recompute regions

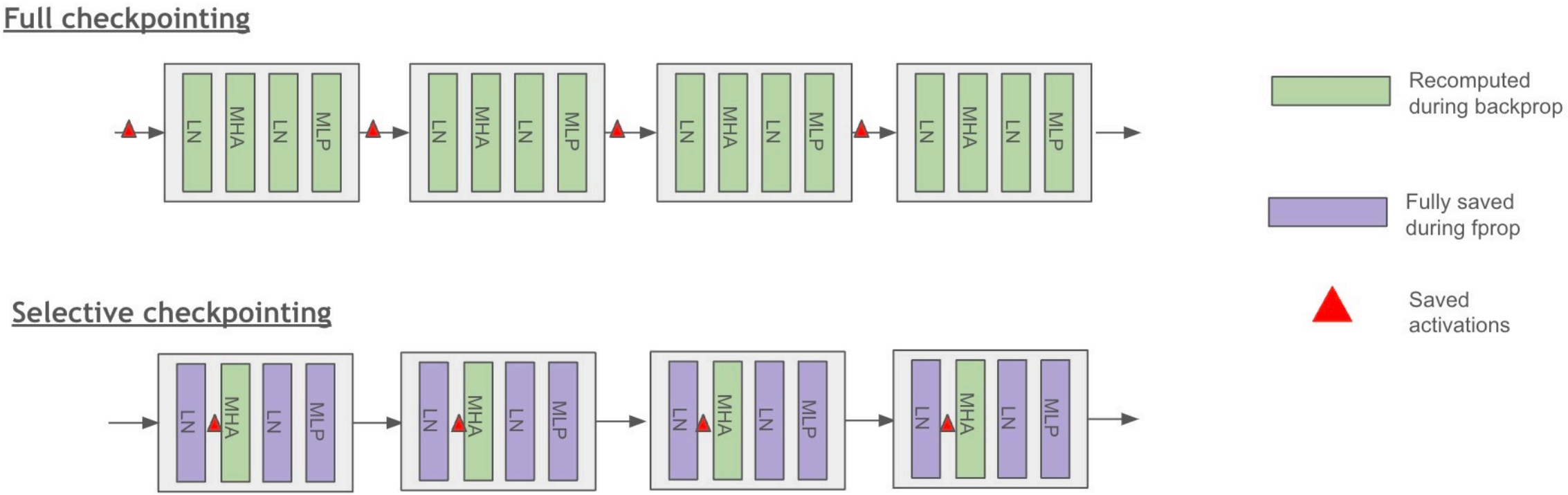
More checkpoints → more saved state, shorter recompute regions

Figure: NVIDIA Megatron Bridge documentation · Policy names and support are framework-specific

SELECTIVE RECOMPUTATION

Selective recomputation targets memory-heavy attention regions

Keep most layer activations; rebuild only the regions with a favorable memory–compute trade-off



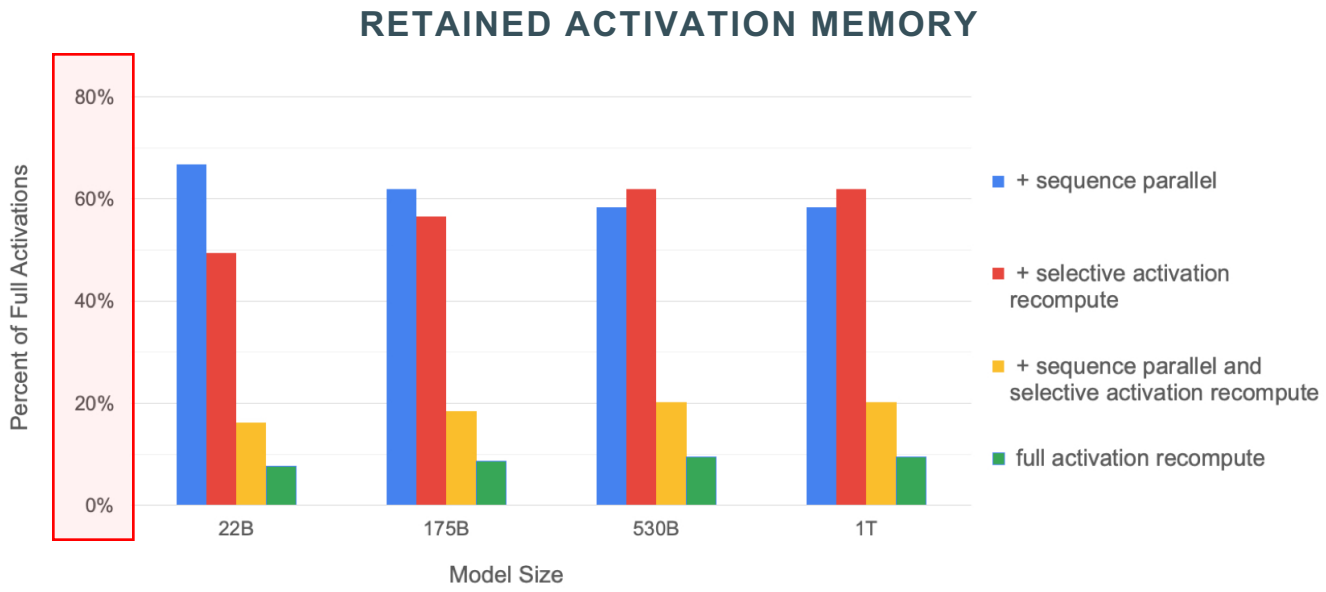
Full layer: minimum retained activations · maximum recomputation

Selective: more retained activations · much less recomputation

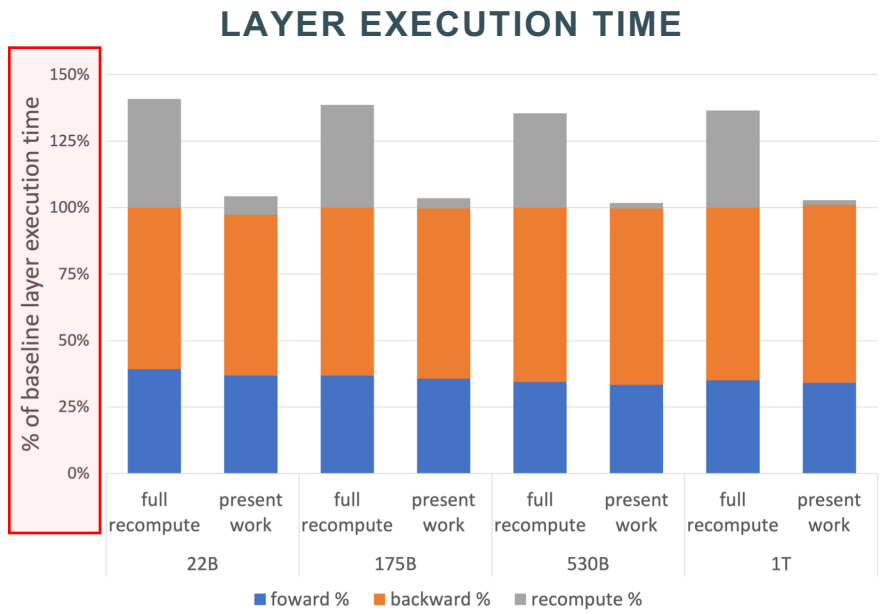
Source: Korthikanti et al., “Reducing Activation Recomputation in Large Transformer Models,” MLSys 2023

MEMORY–COMPUTE TRADE-OFF

Reducing > 30% memory while increasing 10% training time



Full recompute retains the least memory; selective + sequence parallelism preserves more.



Slightly increasing computational time.

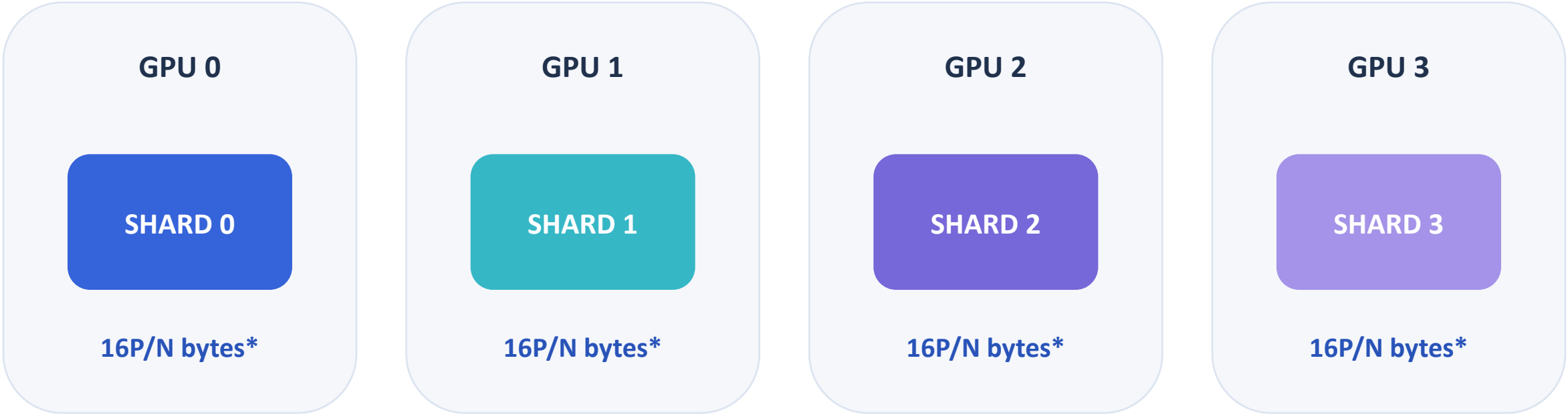
Thanks!



THE ZERO REDUNDANCY IDEA

Partition ownership; communicate only when needed

Each rank persistently owns a different shard instead of a complete replica



Persistent state is sharded; full tensors may appear temporarily in computation buffers.

* Full ZeRO-3 model-state accounting; excludes activations, workspace, and transient gathers.