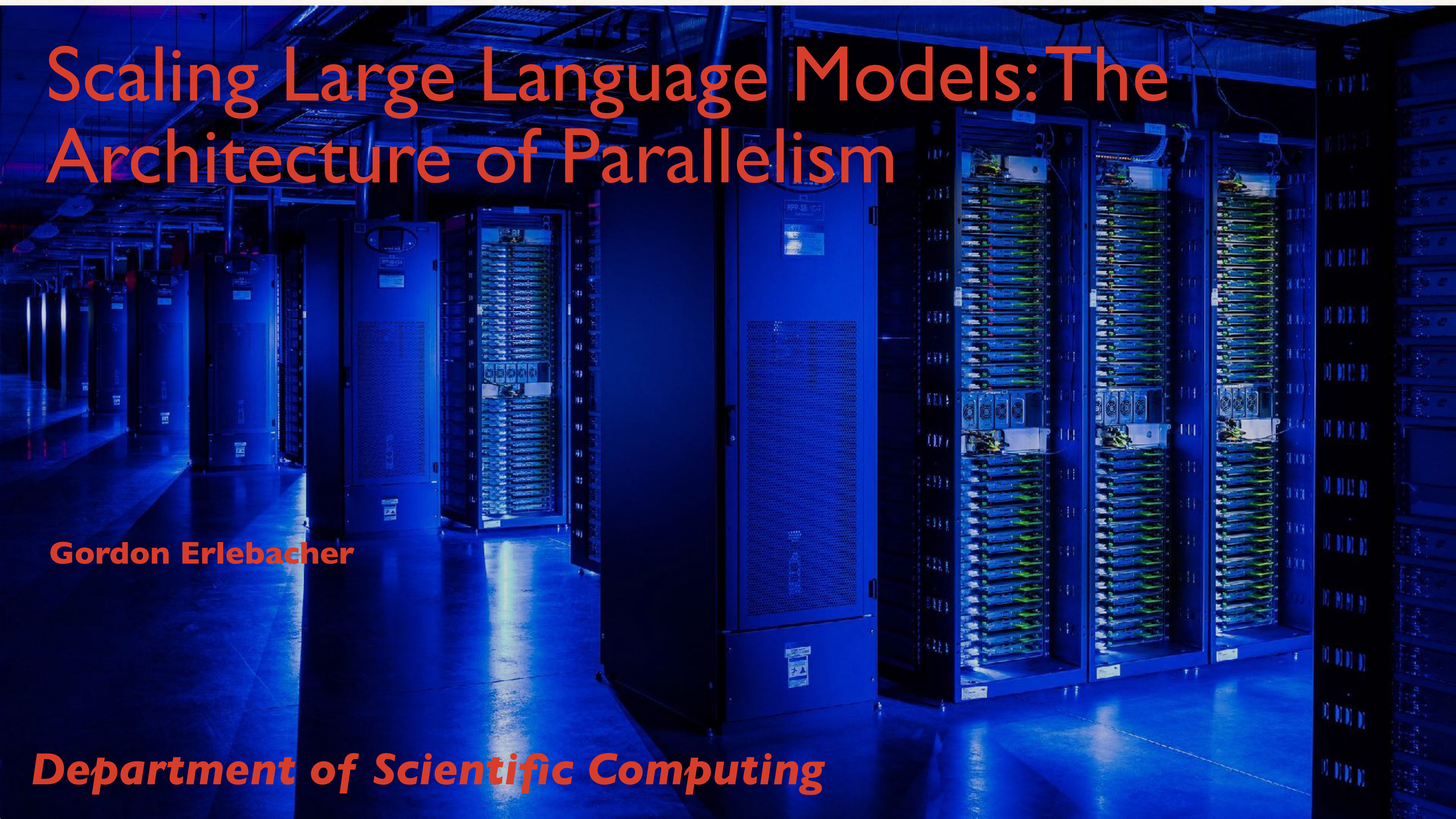
Scaling Large Language Models: The Architecture of Parallelism



Gordon Erlebacher

Department of Scientific Computing

100,000 GPUs at xAI and Meta!

LLM Decoder

Four main axes

Parallelism splits one or more of these axes

B — batch size

L — sequence length

d — embedding width

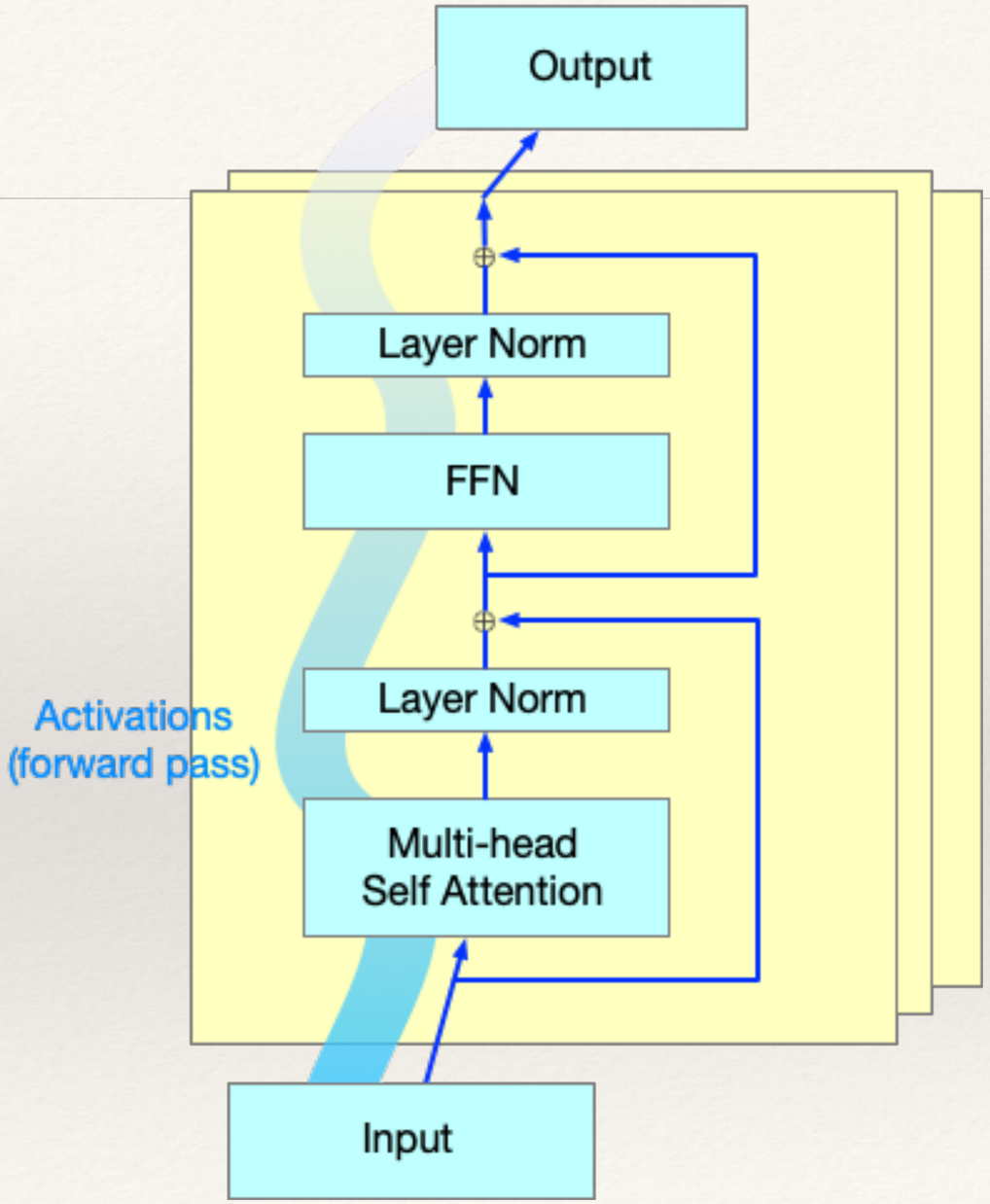
h — number of heads

Shapes

Activations: (B, L, d)

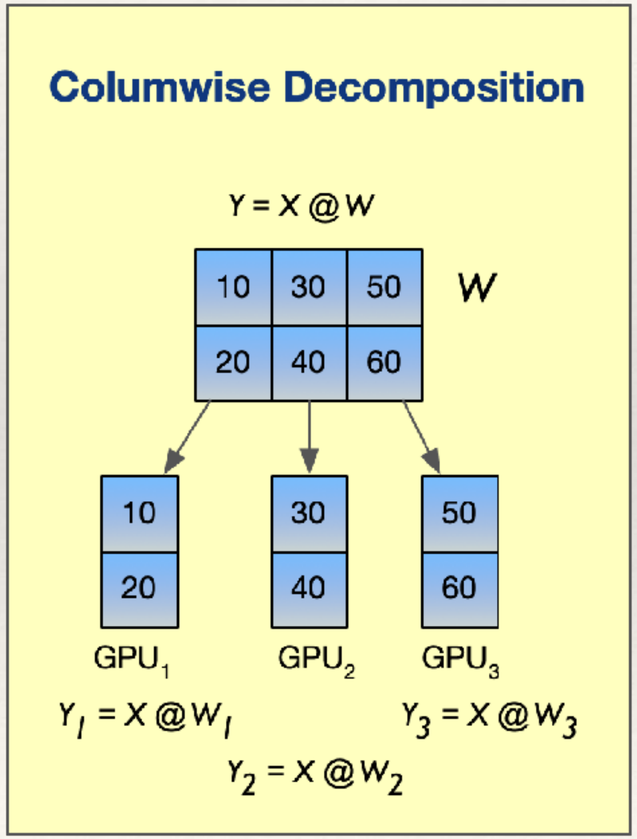
Self-Attention: (B, h, L, L)

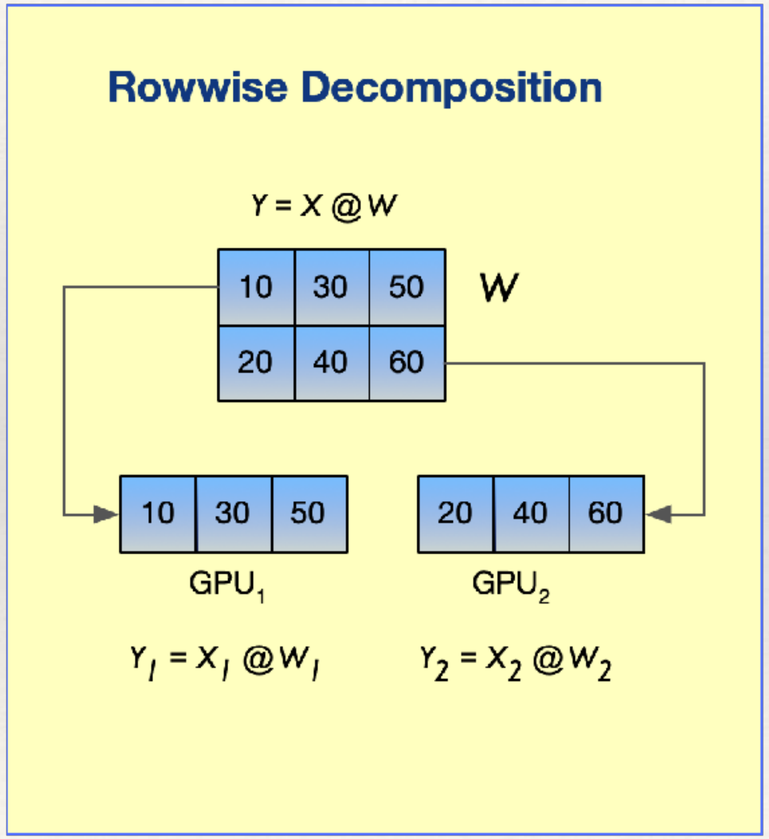
Gradients: #parameters



Matrix-Vector Multiplication

$X \in \mathbb{R}^{5\times2}$ $W \in \mathbb{R}^{2\times3}$ $Y \in \mathbb{R}^{5\times3}$



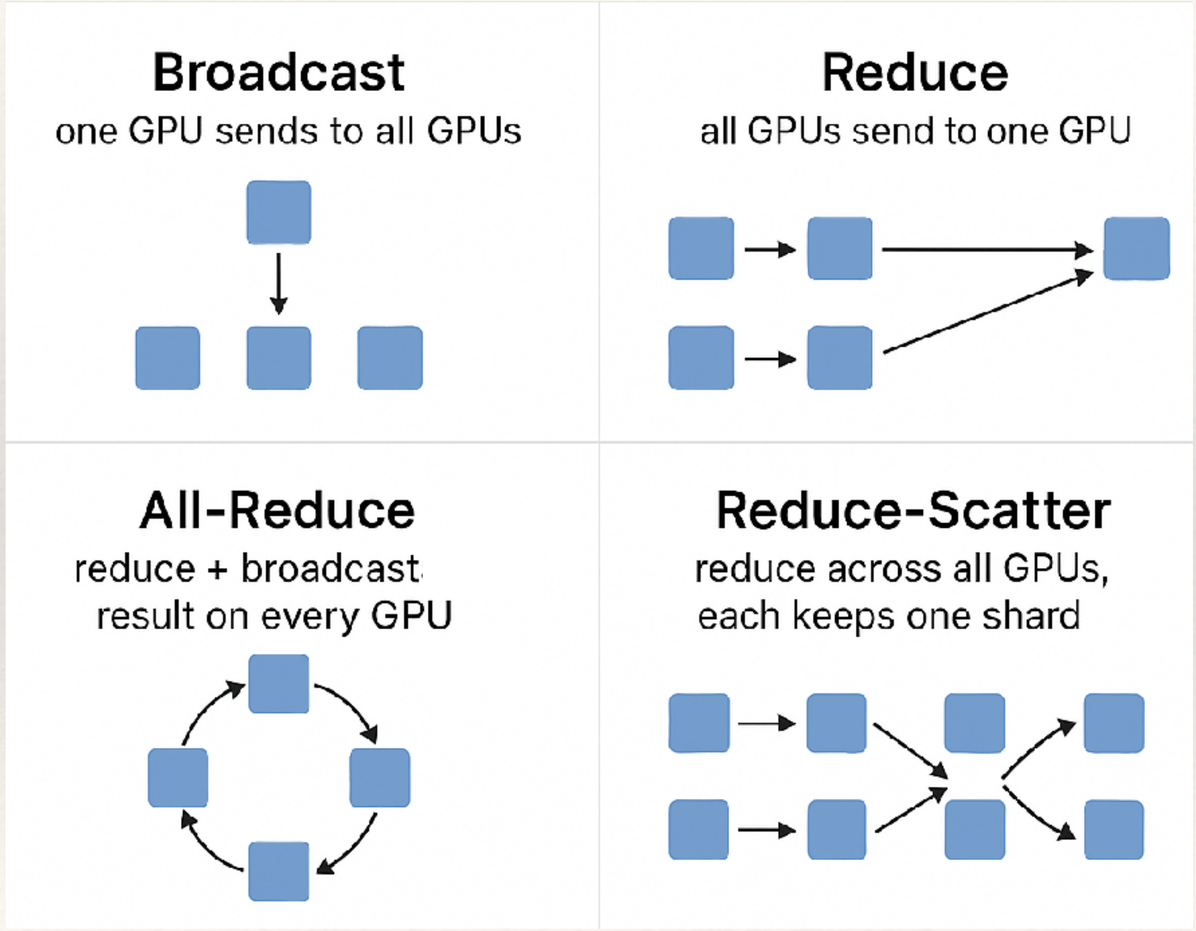


Two natural ways to distribute
$Y = X @ W$

- **Column-wise:** broadcast the input, split W by *columns*
- **Row-wise:** split the *input*, split the *columns* of W

- Each method computes **partial results**
- Subsequent layers (or collectives) combine partial results depending on the partitioning method.

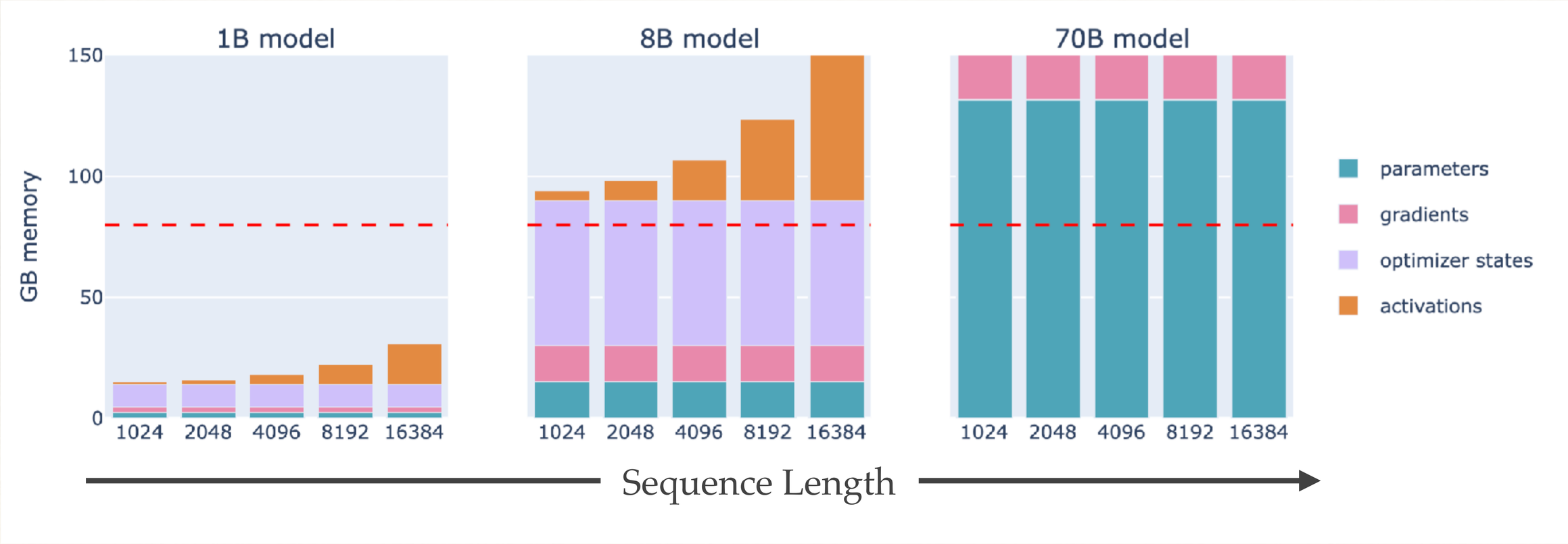
Collective Communication Primitives



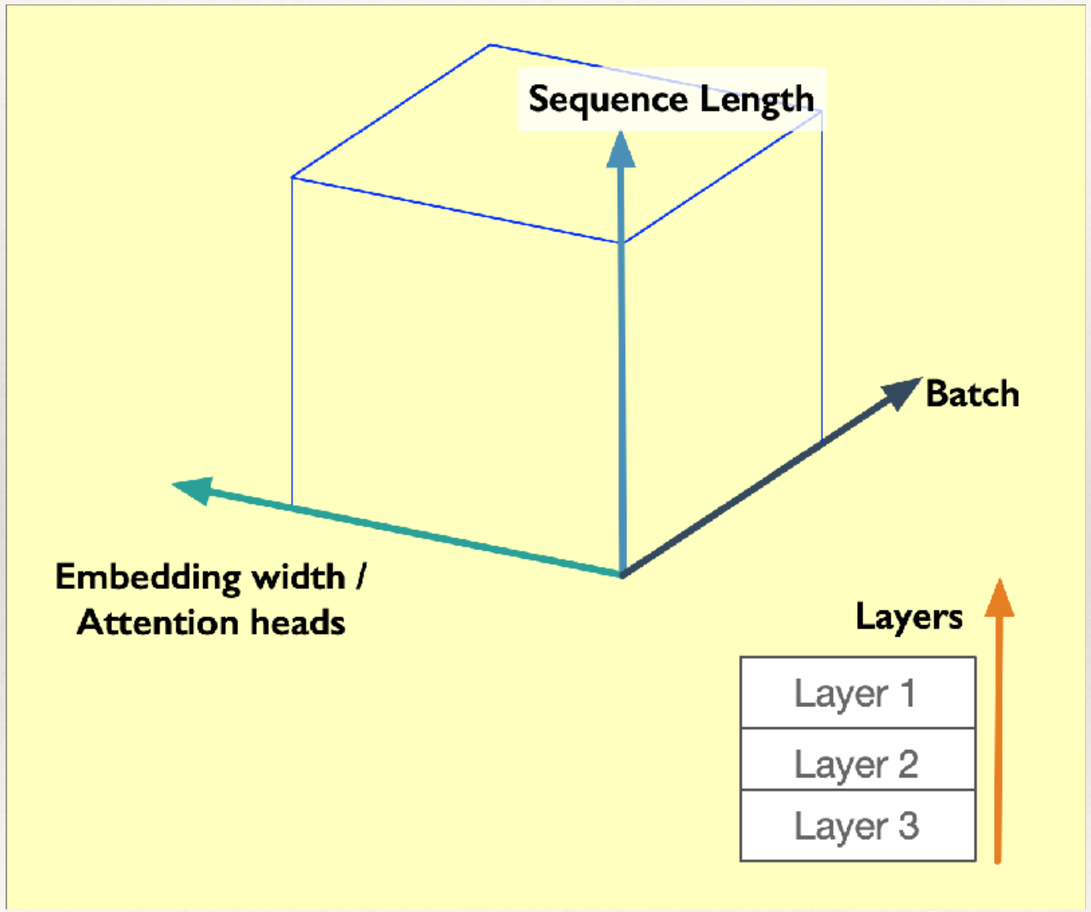
- These primitives move tensors across GPUs efficiently
- Each parallelism uses a distinct subset of collectives
- Communication cost determines how far each method scales

Why Parallelism

Matrix multiplications dominate training;
parallelism is the only path to reasonable wall-clock time.



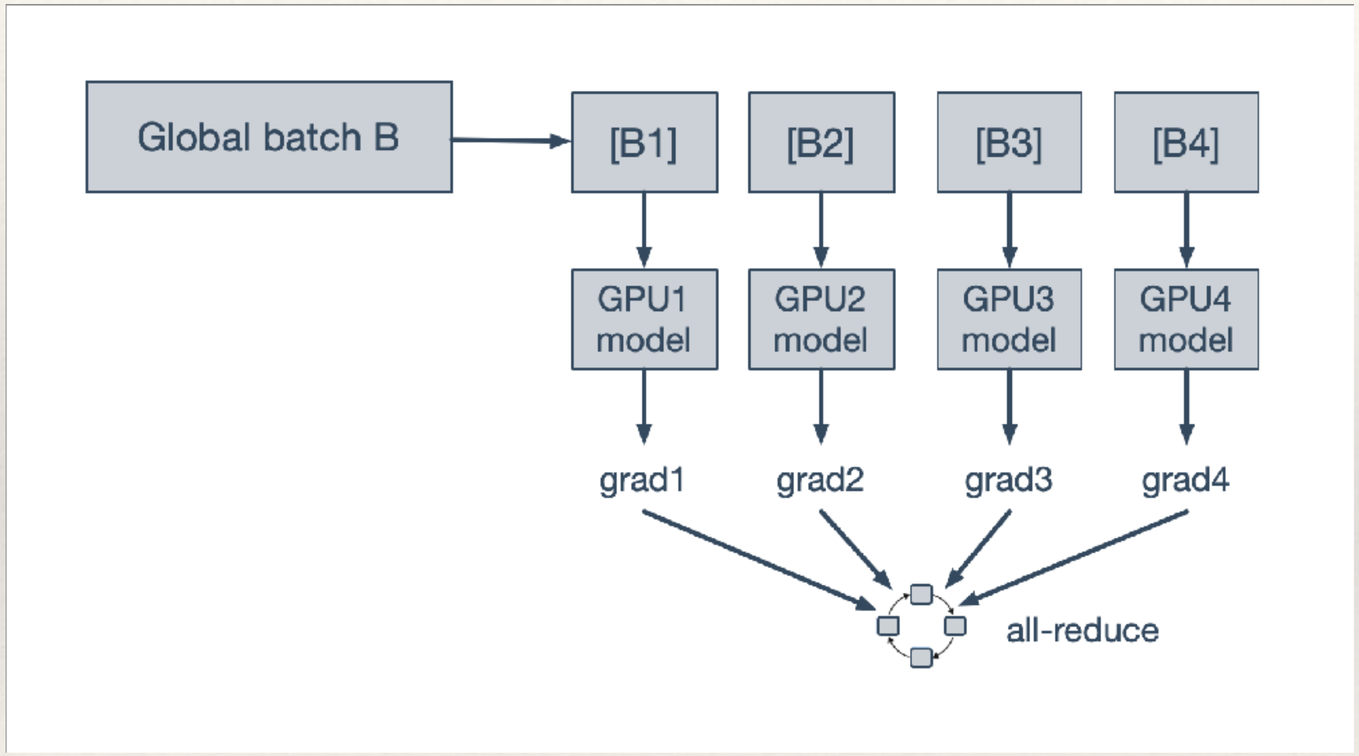
Axes of Scaling



The three Activation Axes define where we can parallelize computation:

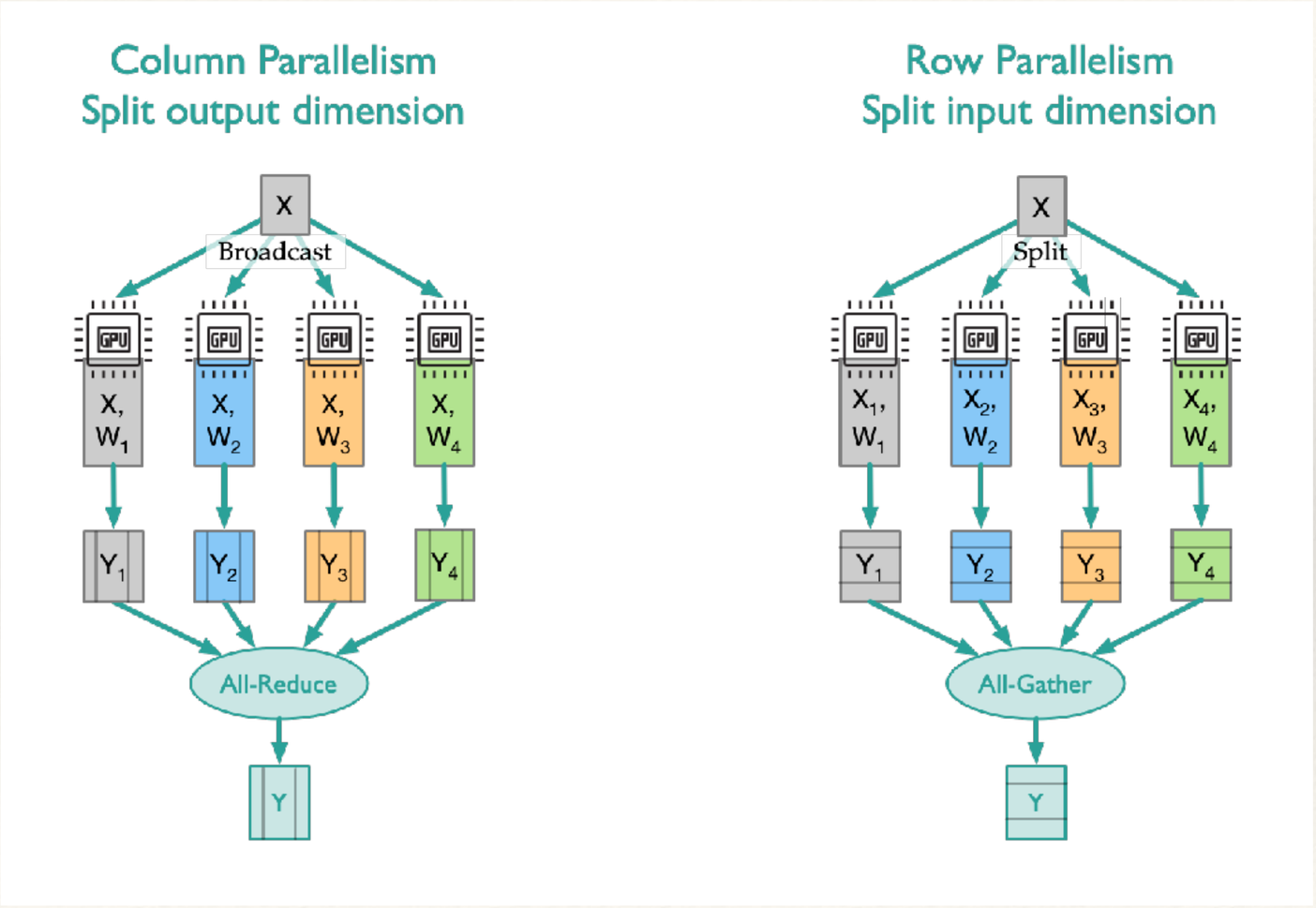
- $B \rightarrow$ Data Parallelism (DP)
- $d \rightarrow$ Tensor Parallelism (TP)
- Layers $\rightarrow$ Pipeline Parallelism (PP)
- $L \rightarrow$ Sequence Parallelism (SP)

Data Parallelism



- Replicate the model on each GPU.
- Split the global batch across devices.
- Each GPU performs a full forward/ backward on its mini-batch.
- Gradients are **all-reduced** so all replicas stay synchronized.

Tensor Parallelism

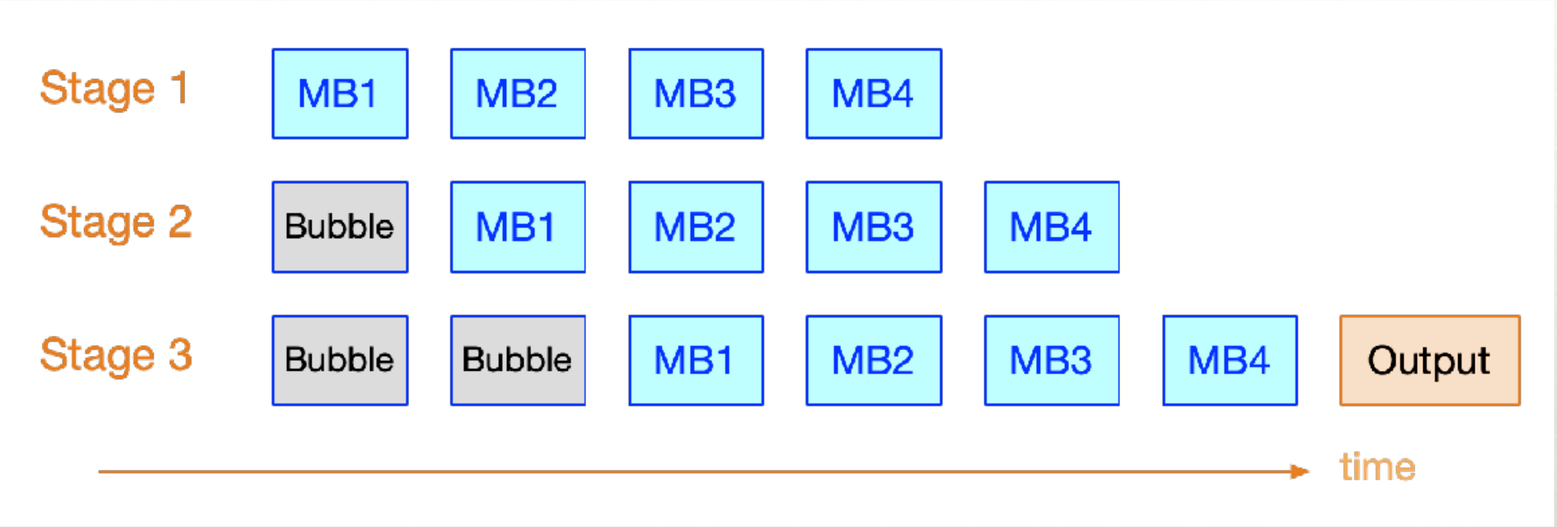


- Split within a layer; large GEMMs are partitioned across GPUs.
- Each GPU computes partial outputs and participates in collectives.
- High communication intensity; scales only on high bandwidth links (NVLink/ NVSwitch).

Column parallelism: $Y_i = X @ \text{Col}_i(W)$

Row parallelism: $Y_j = X_i @ W_{i,j}$

Pipeline Parallelism



- Split the model by layers into sequential stages across GPUs
- Micro-batches flow through stages like an assembly line, increasing utilization
- Introduces pipeline bubbles that require tuning micro-batch count and stage depth

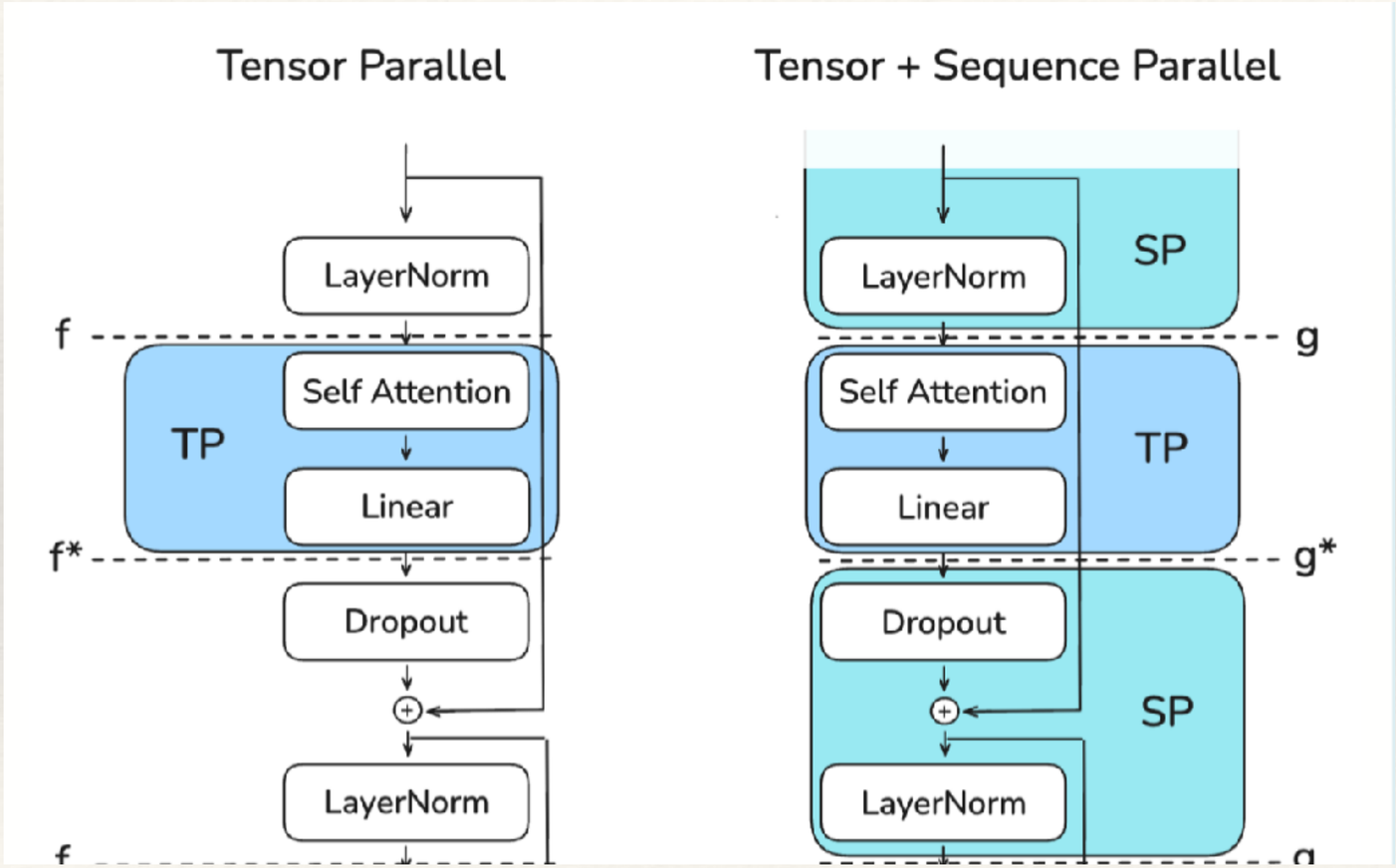
Stage 1: Layers 1-2

Stage 2: Layers 3-4

Stage 3: Layers 5-6

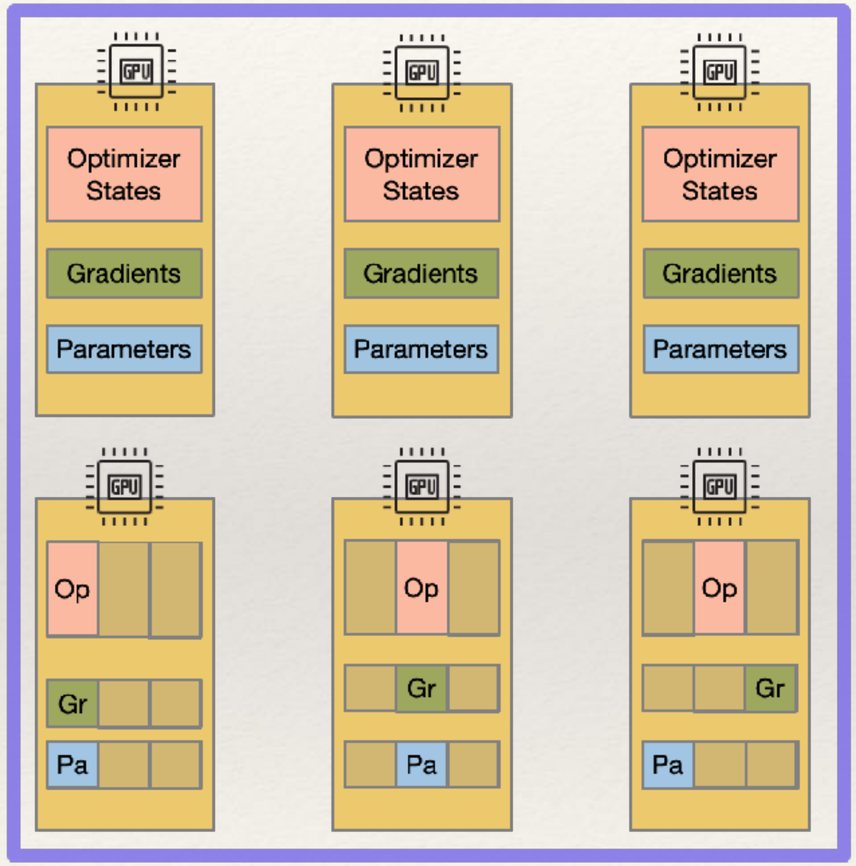
Micro-batches MB1, MB2, etc.
Batch = MB1 + MB2 + …

Sequence (Context) Parallelism



- Partition the sequence dimension **L** across GPUs
- Compute multi-head attention locally for each token block
- Exchange context via collective communication
- Achieve efficient scaling to very long sequences
- Reduce per-GPU activation width

ZeRO: Zero Redundancy Optimizer



- **ZeRO shards optimizer states, gradients, and parameters** across GPUs instead of replicating them
- **Each GPU stores only the pieces it updates**, requesting other shards when needed
- **Greatly reduces memory per GPU**, enabling much larger models without changing the computation
- **The ZeRO concept is framework-independent** and underlies DeepSpeed, PyTorch FSDP, and other sharded training systems

ZeRO Memory Partitioning

Stage	Partitioned Components	Memory Saved	Notes / Interaction
ZeRO-1	Optimizer States (m, v)	≈ 2 × model parameters	Reduces redundant optimizer copies across GPUs (works naturally with **DP**)
ZeRO-2	Optimizer + Gradients	+ O(params)	Further cuts gradient storage; still requires gradient all-reduce after backprop
ZeRO-3	Optimizer + Gradients + Parameters	≈ O(total params / N)	Fully partitions weights; compatible with **TP** and **PP**, adds comm overhead
ZeRO-Offload	Moves states to CPU / NVMe	GPU → host memory	Useful when GPU VRAM is limited; increases transfer latency

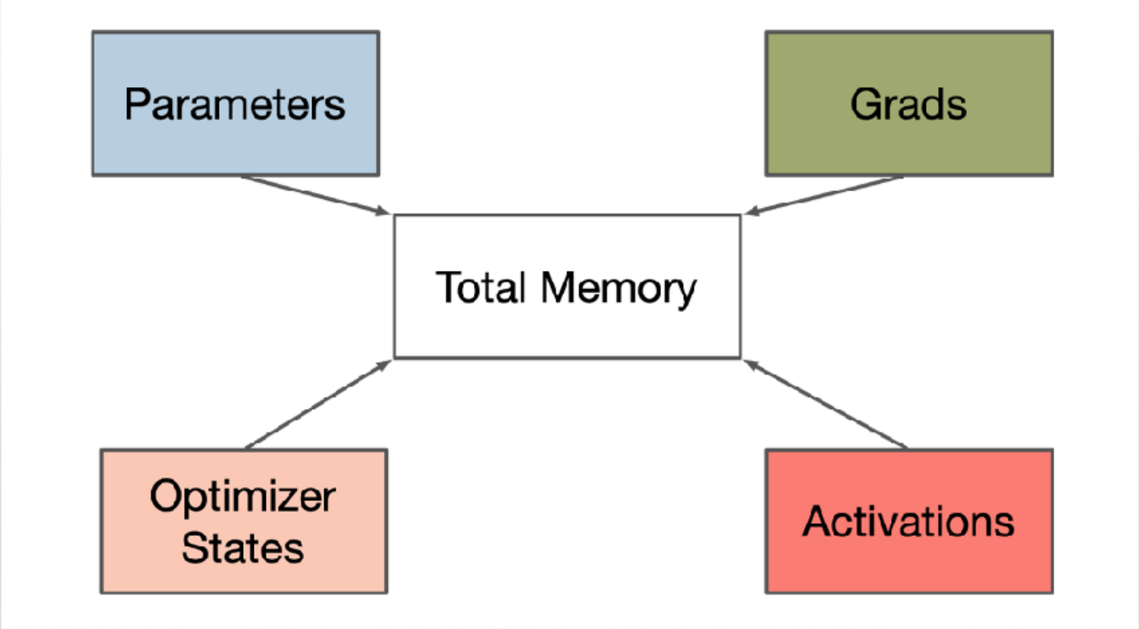
ZeRO shards what is **stored**, without altering computation.

Memory Elements, Shapes, and Applicable Parallelisms

Memory Element	Shape	DP	TP	PP	SP/CP
Input Activations (X)	(B, L, d)	Split B	split **d** (row)	per layer	splits **L**
Q, K, V Activations	(B, L, d)	B	d (row/col)	*id*	L
Attention Scores		B	H (heads)	*id*	L
Attention Output	(B, L, d)	B	d (col)	*id*	L
MLP Hidden Activations	(B, L, 4d)	B	4d (col)	*id*	L
LayerNorm / Residuals	(B ,L, d)	B	d	*id*	L
Attention Weights	(d, d)	—	d (col/row)	*id*	—
MLP Weights	(d, 4d), (4d, d)	—	d (col/row)	*id*	—
Gradients	see weights	—	d	*id*	—
Optimizer States	see weights	—	d	*id*	—
Embeddings (Token/Pos)	(Vocab, d)	B (via data)	d	shard	—

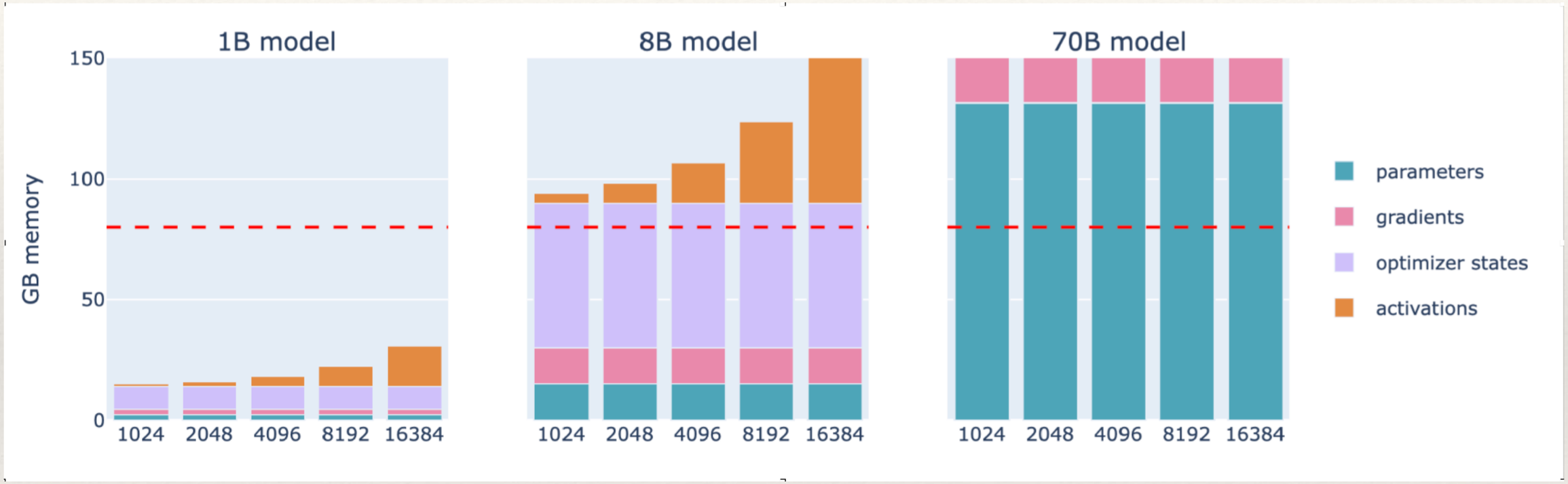
Memory and Compute Scaling

- $M_{tot} = M_p + M_g + M_o + M_a$
- p (trainable weights): N
- g (gradients): $O(N)$ (one per parameter)
- o (optimizer states): $O(N)$ (2-4 xN)
- a (activations): $O(BLd)$



- **Model size (↑ N):** parameters + gradients + optimizer states dominate
- **Batch & sequence size (↑ B or ↑ L):** Activations dominate

When Activations Dominate

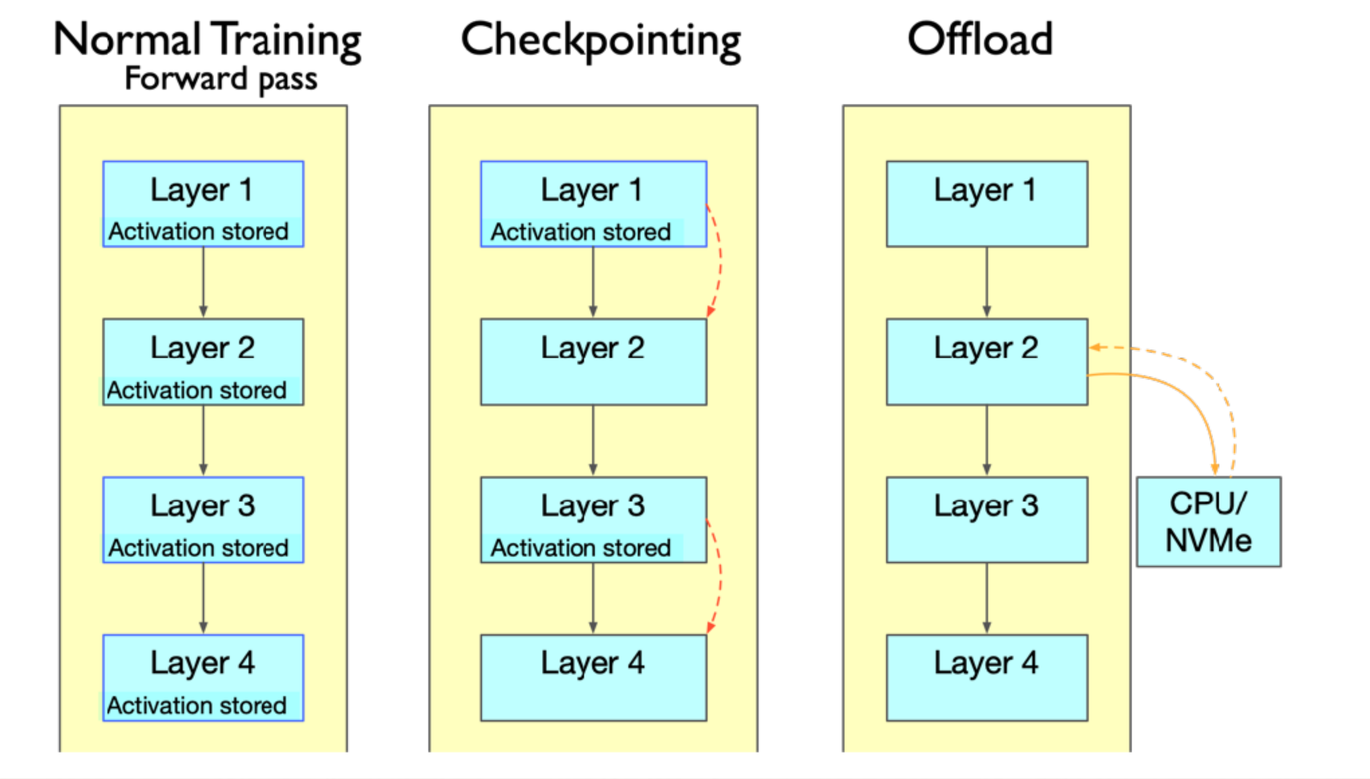


- Checkpointing and offload become essential at this point

Why Activations Become the Next Bottleneck

- ZeRO removes all redundant model states (parameters, grads, optimizer)
- Activations scale as $O(\text{batch} \times \text{sequence} \times \text{width} \times \text{layers})$
- Beyond ~50–70B, activations dominate per-GPU memory
- This becomes the new bottleneck even after ZeRO-3
- **Solutions:** recompute activations or offload them

Checkpointing and Offload



- Checkpointing trades compute for memory; offload trades bandwidth for memory

- Activations cannot be sharded — they're sample-specific
- Checkpointing stores only key activations and recomputes the rest
- Significantly reduces activation memory at moderate compute cost
- Offload moves activations to CPU/ NVMe before backward
- Both extend scalability beyond ZeRO-3 limits

Who Solves What?

Parallelism	Memory saved	Computation saved	Communication cost
Data Parallelism (DP)	⚪ None	🟢 Scales data throughput	🔴 High (global gradient all-reduce)
Tensor Parallelism (TP)	🟢 Reduces layer memory footprint	🟢 Scales width (tensor ops)	🔴 High (per-layer collectives)
Pipeline Parallelism (PP)	⚪ Moderate	🟢 Scales depth (layers)	🟠 Moderate (pipeline bubbles, stage sync)
Sequence Parallelism (SP)	🟢 Reduces context memory per GPU	🟢 Scales context length	🔴 High (attention exchange)
Zero memory Partitioning (ZeRO)	🟢 Major—shards states, grads, params	⚪ None (no compute gain)	🟠 Moderate (extra computation per step)

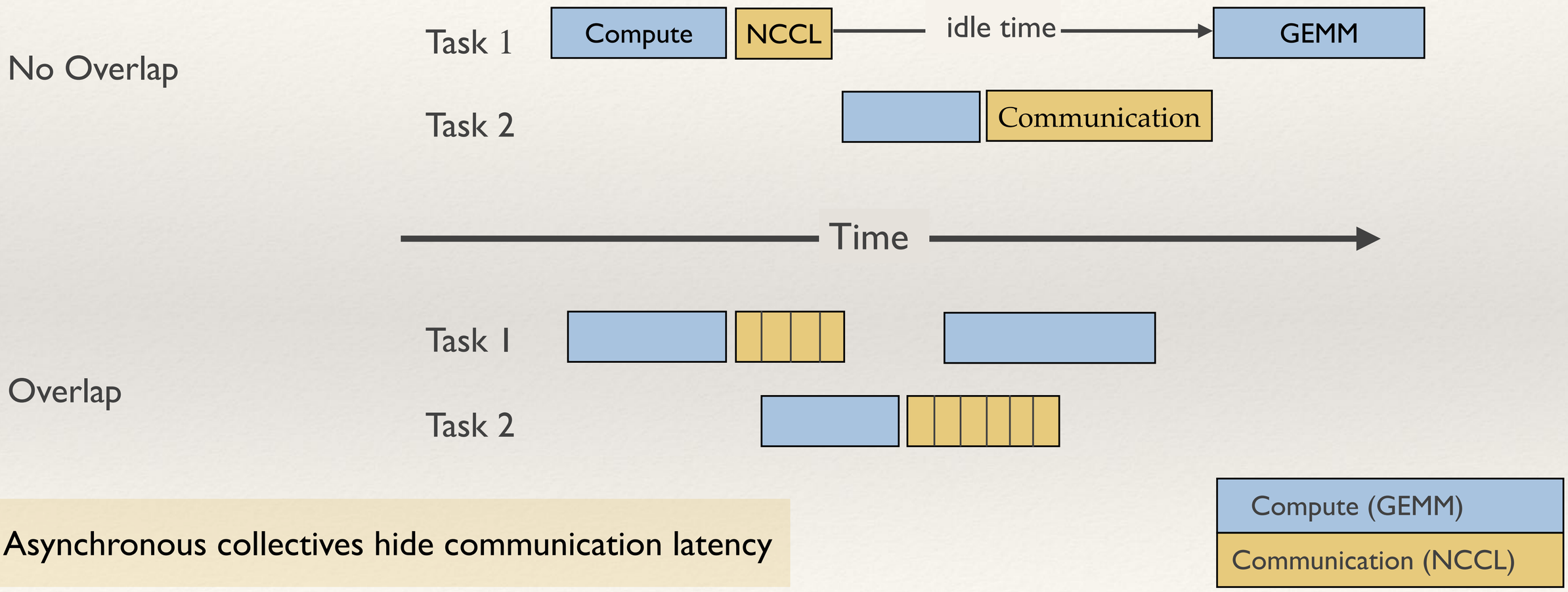
🟢 = Strong benefit ⚪ = Neutral or minimal effect 🔴 = Significant cost 🟠 = Moderate cost

DP scales batch dim PP scales depth TP scales width SP scales context ZeRO shards memory

Who Solves What?

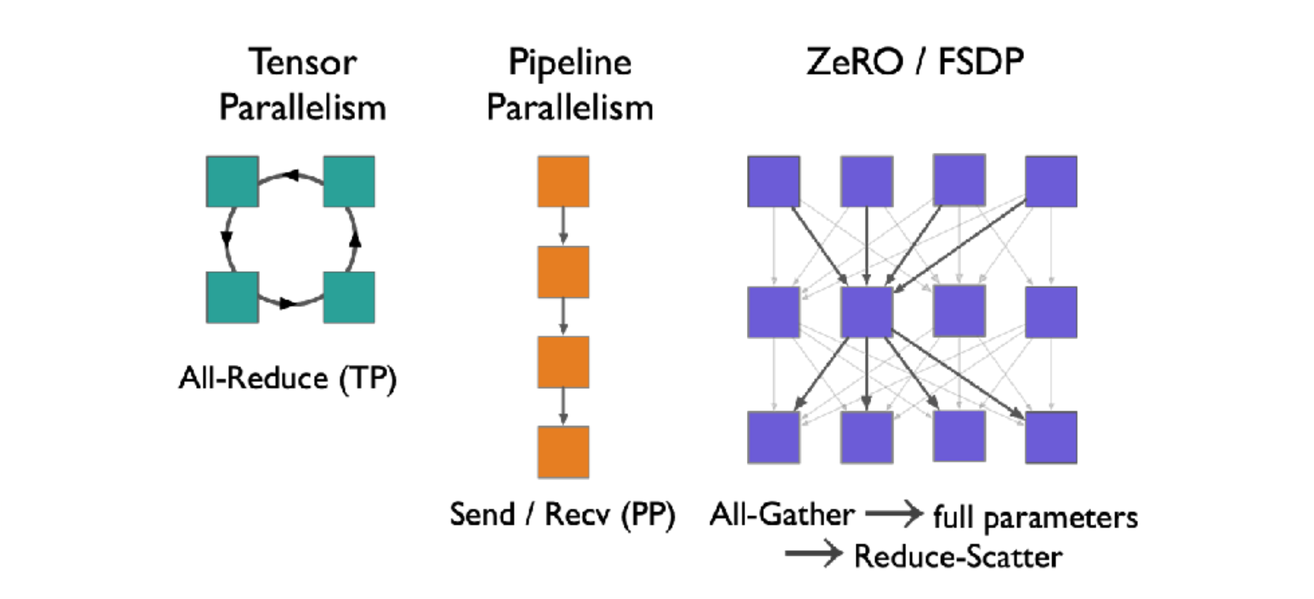
It's not just what we parallelize, but how we schedule compute and communication. Let's look at how overlap helps.

Overlapping Compute and Communication



Asynchronous collectives hide communication latency

Communication Domains



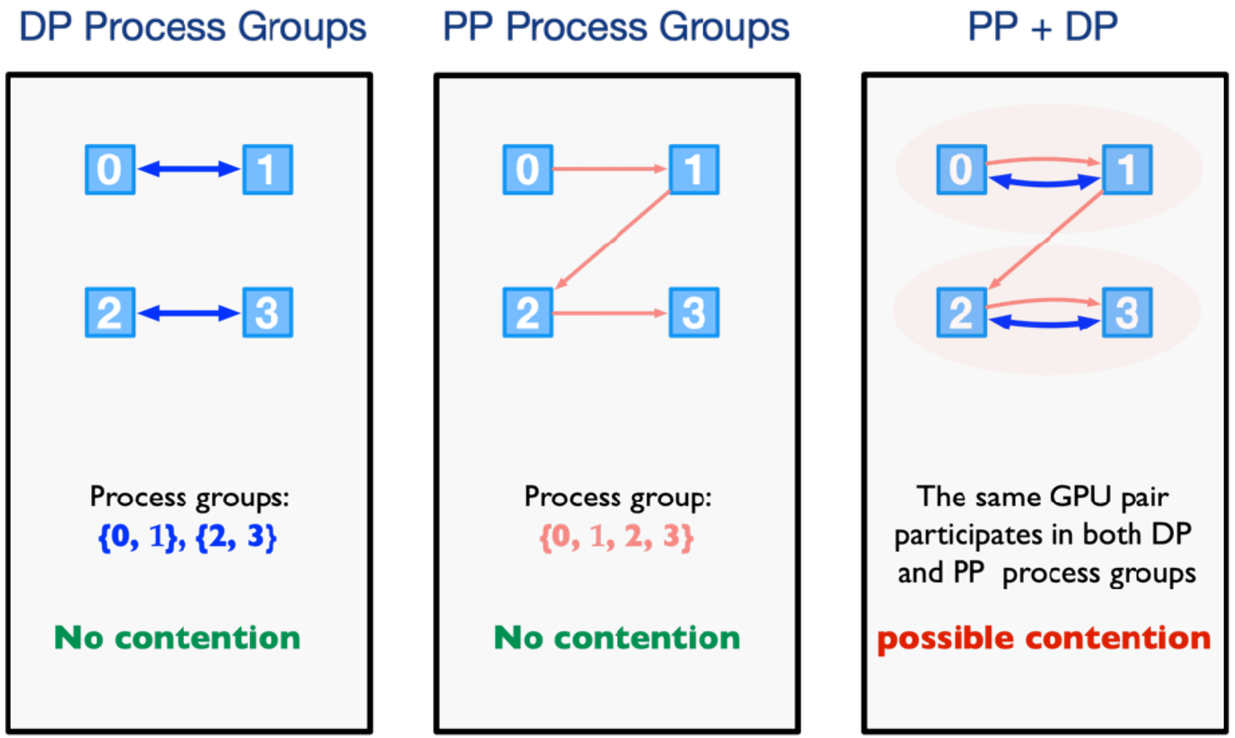
TP Group	PP Group	ZeRO Group
[■ ■ ■ ■]	[■ ■ ■ ■]	[■ ■ ■ ■]
(All-Reduce)	(Send/Recv)	(AG/RS)

- TP, PP, and ZeRO each use their own communication primitive
- Each runs inside its own dedicated process group
- Collectives are issued on separate CUDA streams
- Compute and communication overlap asynchronously
- This separation reduces contention and improves scaling efficiency

Communication Domains

All these communication patterns operate simultaneously in real training runs, which leads naturally to hybrid 3D and 4D schemes.

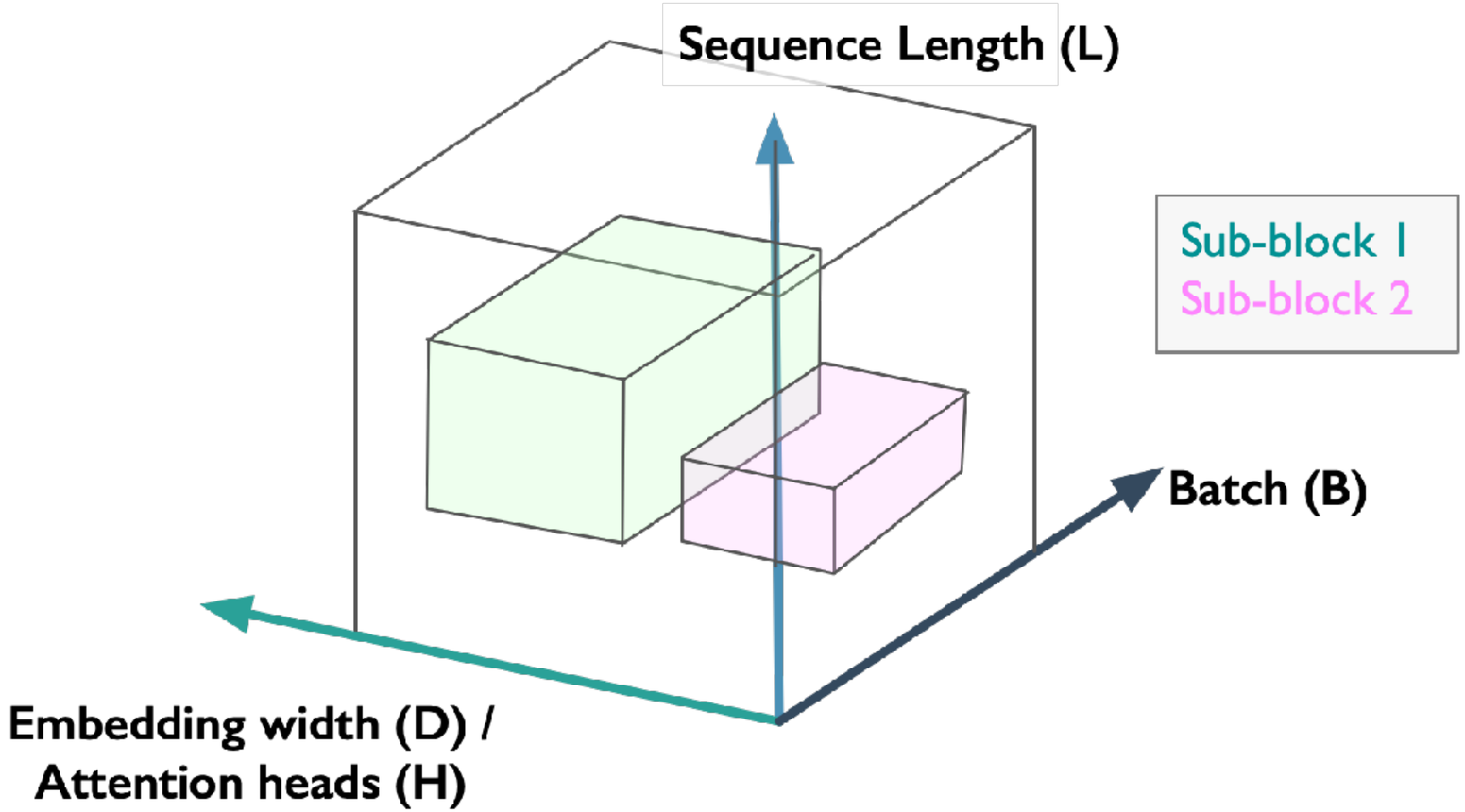
Overlapping Communication Groups



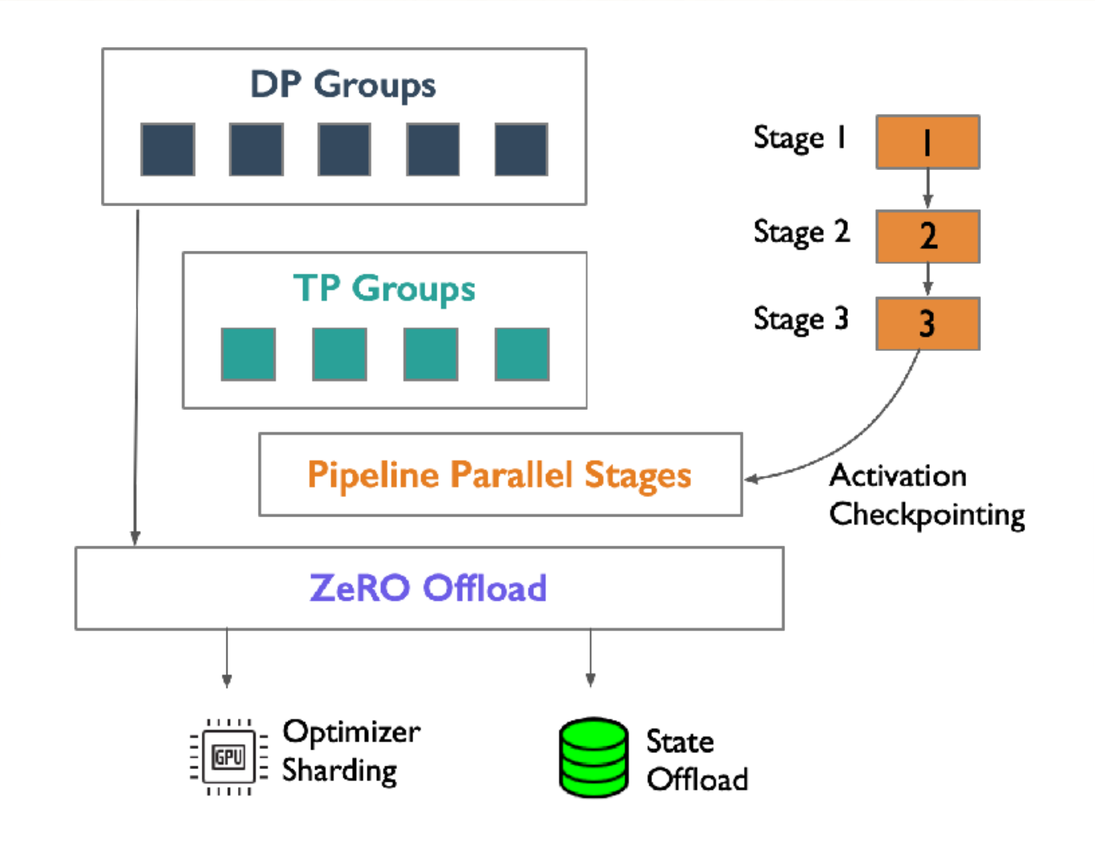
- The *same* GPUs can participate in multiple parallelisms ($DP \times TP \times PP \times$ ZeRO)
- When two parallelisms use the **same GPU pair**, communication can collide
- Whether contention happens depends on scheduling, streams, chunking, and framework design

Contention Can Occur

Hybrid Parallelism



LLaMA-3 Training Stack

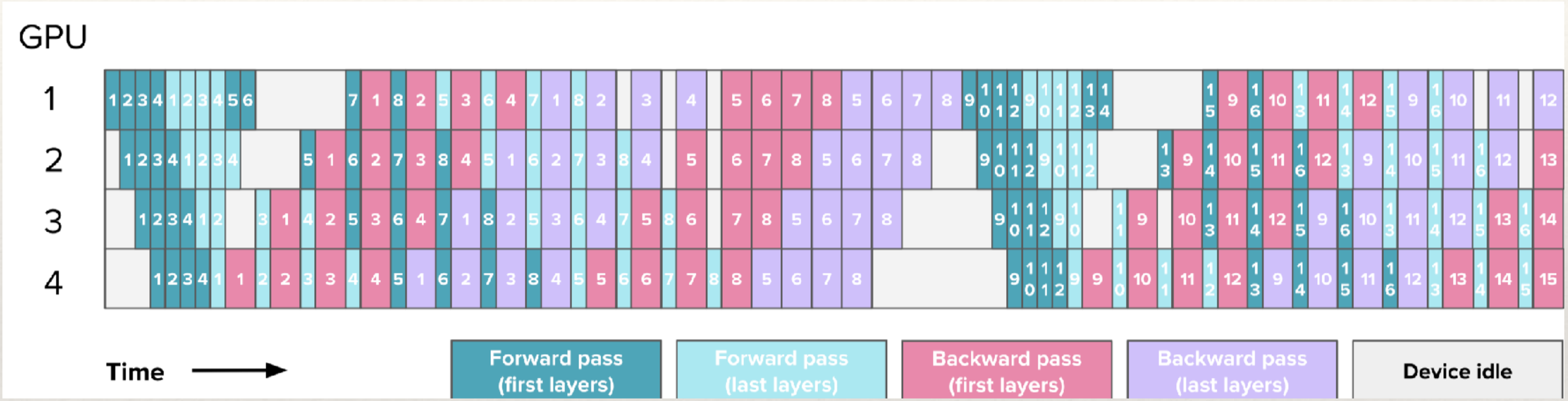


- **4-D hybrid:** FSDP × TP × PP × SP
- **FSDP:** ZeRO-style sharding (states + parameters)
- **TP:** Splits large matmuls
- **PP:** Splits layers
- **SP:** Splits sequences for extended context training
- 405B on 16K GPUs (ISCA'25)

Olmo Training Stack (Fully Open Source)

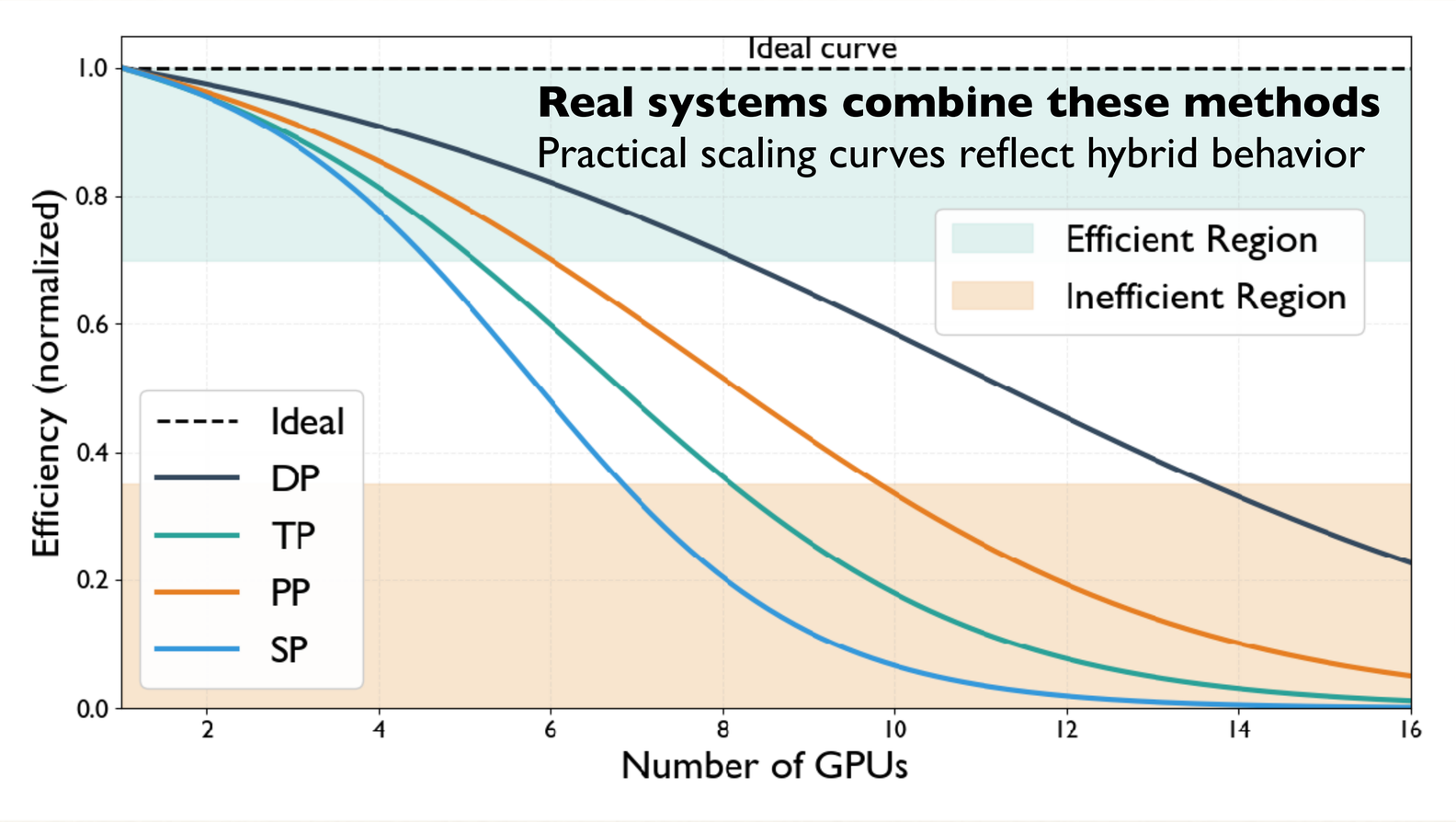
- **Framework:** DeepSpeed + Megatron-LM (hybrid)
- **Parallelism:**
  - DP × TP × PP (all configurations public)
  - ZeRO-1/2 for optimizer + gradient sharding
  - Activation checkpointing for memory reduction
- **Data Pipeline:**
  - Entire corpus (Dolma) fully open: sources, cleaning, filters
  - Public preprocessing + sequence packing scripts
- **Compute:**
  - Trained on A100 (80GB) clusters
  - Uses mixed precision (bfloat16) + fused kernels (GELU, softmax)
- **Topology Examples:**
  - **OLMo-7B:** DP × TP × PP = $8 \times 1 \times 1$
  - **Larger variants:** $48 \times 2 \times 2$ on 192 GPUs
- **Key Point**:
  - OLMo is one of the only modern LLMs with a fully open, end-to-end reproducible training stack—from data to checkpoints.

Interleaved Pipeline Parallelism



Many combinations possible!
Strongly dependent on architecture, hardware, data

Scaling Efficiency



DP scales the farthest due to minimal communication, long plateau

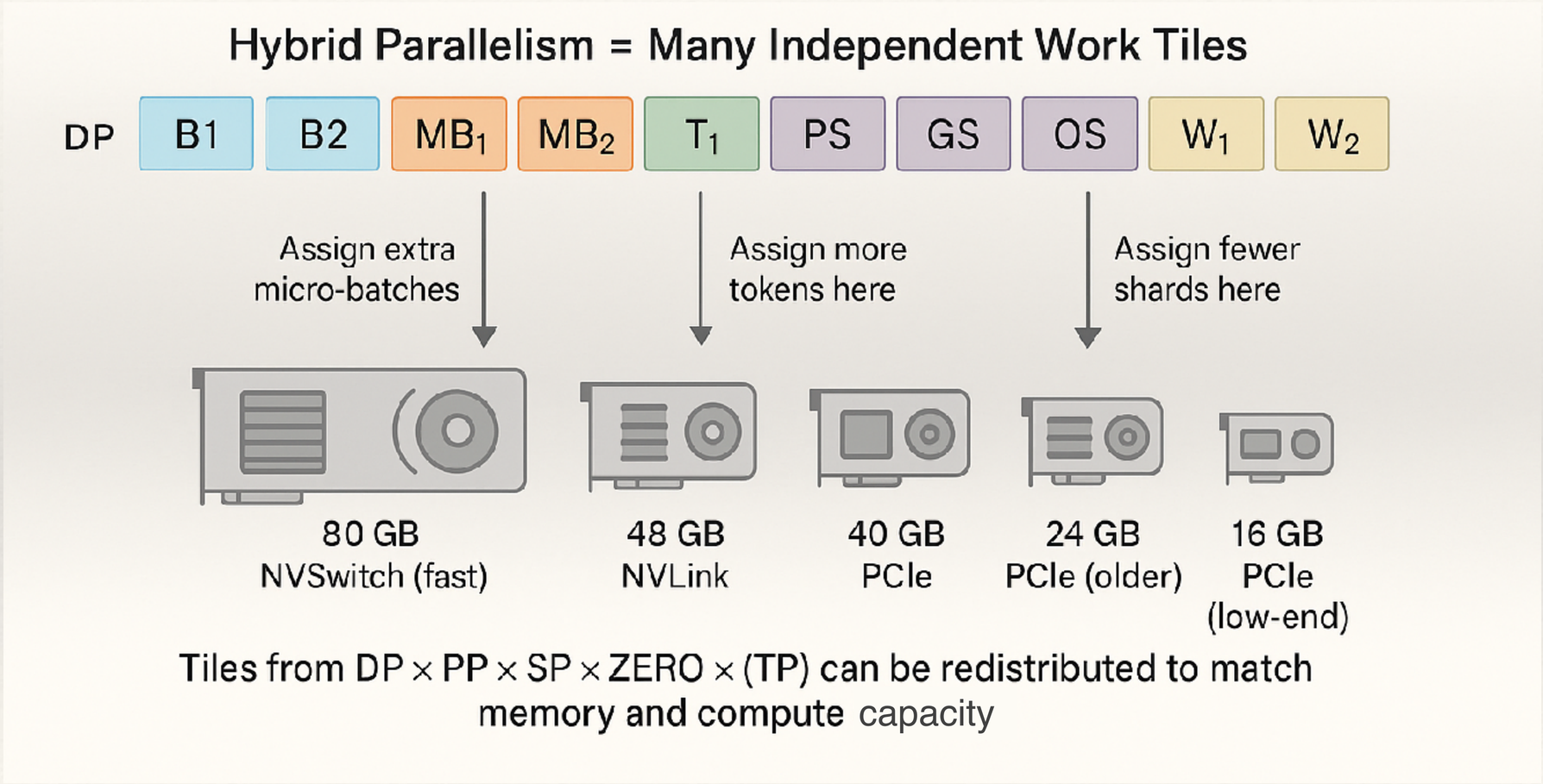
PP scales moderately well as efficiency limited by pipeline bubbles

TP loses efficiency earliest due to high intra-layer communication

SP drops fastest as sequence partitioning increases overhead

All methods show diminishing returns as GPU count rises

Hybrid Parallelism Enables Efficient Training on Inhomogeneous GPU Clusters



Parallelism Takeaways

- **Data Parallelism** (DP) scales to the highest GPU counts because it communicates only gradients once per step; communication volume grows modestly with model size, so efficiency stays high until very large GPU counts. DP is always present. The other methods remove orthogonal bottlenecks.
- **Tensor Parallelism** (TP) has the most restrictive scalability because it slices within layers. It requires extremely fast, low-latency links (NVLink, NVSwitch). Scaling TP across nodes rapidly destroys efficiency.
- **Pipeline Parallelism** (PP) sits in the middle: it can use many GPUs, but efficiency suffers from pipeline bubbles and increased synchronization at extreme scale.
- **Sequence Parallelism** (SP) is similar: communication pattern depends on the sequence dimension and is node-local in practice.
- **ZeRO** (esp. Stage 3) introduces heavy communication on every forward/backward step when parameters/grads/optimizer shards must be gathered; this makes it more like TP in terms of communication intensity.

Evolution of Parallelism

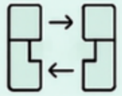
2012-2013

Data Parallelism (DP)

gradient all-reduce across devices

2018–2019

Tensor Parallelism (TP)



intra-layer model partitioning

2019–2020

Pipeline Parallelism (PP)

layer-wise micro-batch pipelining

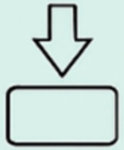
2019–2021

ZeRO—1/2/3

optimizer / gradient / parameter sharding

2020–2021

Memory-Saving Techniques



activation checkpointing, offloading

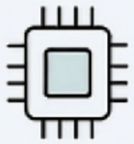
2022–2023

Sequence Parallelism (SP)

sharding the sequence dimension

2010

Single-GPU Training



small models no parallelism

2023–2024

Modern Multi-Dimensional Parallelism

3D/4D parallelism as used in modern LLMs

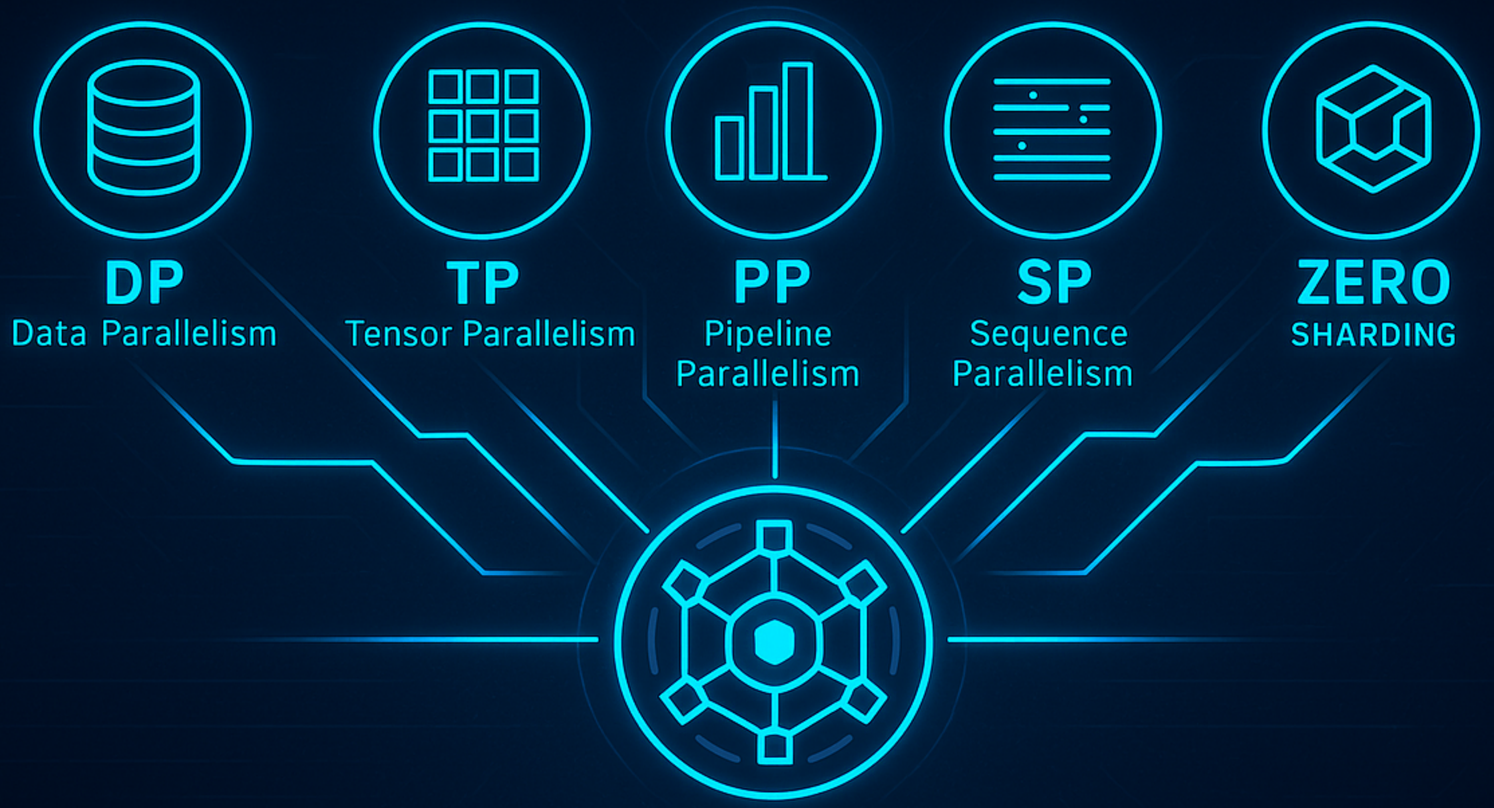
Parallelism evolved from simple replication to multi-dimensional hybrids that enable trillion-parameter model training

2021–2025

Hybrids (DP x TP x PP)

combining strategies for large models

TAKEAWAYS AND OUTLOOK



Parallelism is the architecture of scale