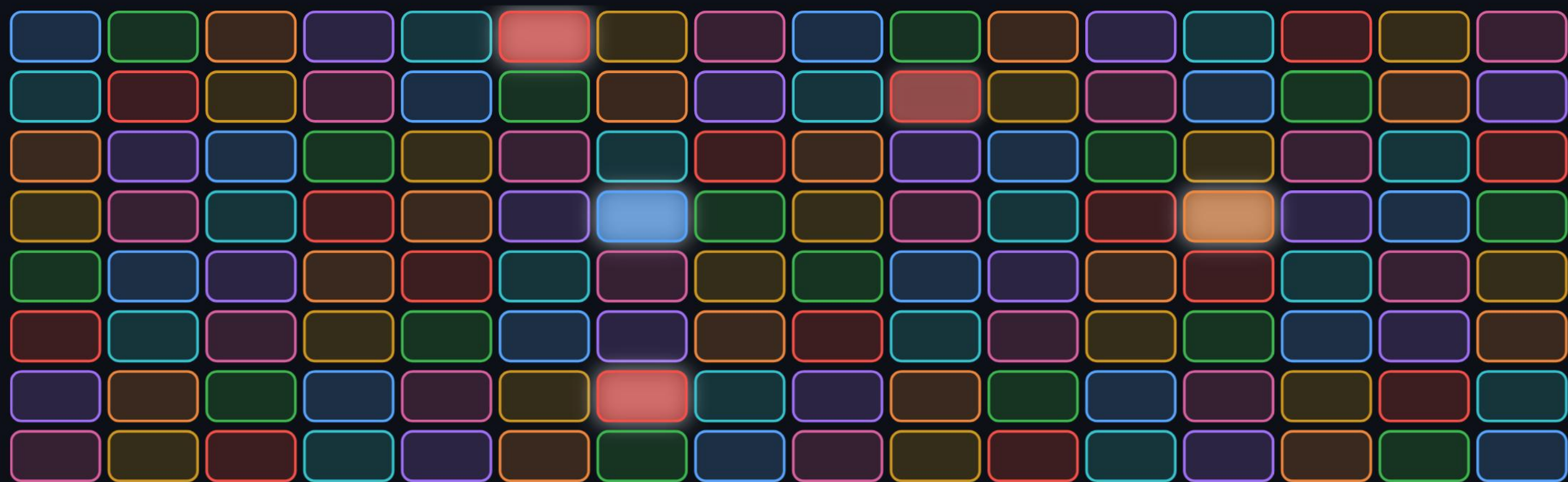


MASSIVELY PARALLEL TRAINING

Training Neural Networks at the Scale of Thousands of GPUs

Mathias Lechner

Co-founder and CTO @ Liquid AI



AGENDA



About Liquid AI

Foundation model company & MIT spin-off (2023)

Recently released the LFM2 series of on-device optimized language models

Today's Talk

Why We Scale

GPU vs CPU performance
Scaling laws
Data & model scaling

What We Scale

Data parallelism
Model parallelism
Sharding strategies

How We Scale

DeepSpeed ZeRO
FSDP
Megatron-LM

Case Study: Training LFM2

WHY TRAIN ON GPUS?

2011: AlexNet — Start of the Deep Learning Revolution

Trained on 2× GTX 580 GPUs

Training a Neural Network is Matrix Multiplication

$$[256 \times 4096] \times [4096 \times 4096] = 8.6 \text{ GFLOP}$$

CPU (2011)

Intel i7-3960X
0.1 TFLOPS
FP32
≈ 86 ms

GPU (2011)

GTX 580
1.58 TFLOPS
FP32
≈ 5.4 ms

GPU (2022)

NVIDIA H100
989 TFLOPS
BF16
≈ 8.7 μs

16× faster

~1000× since 2011

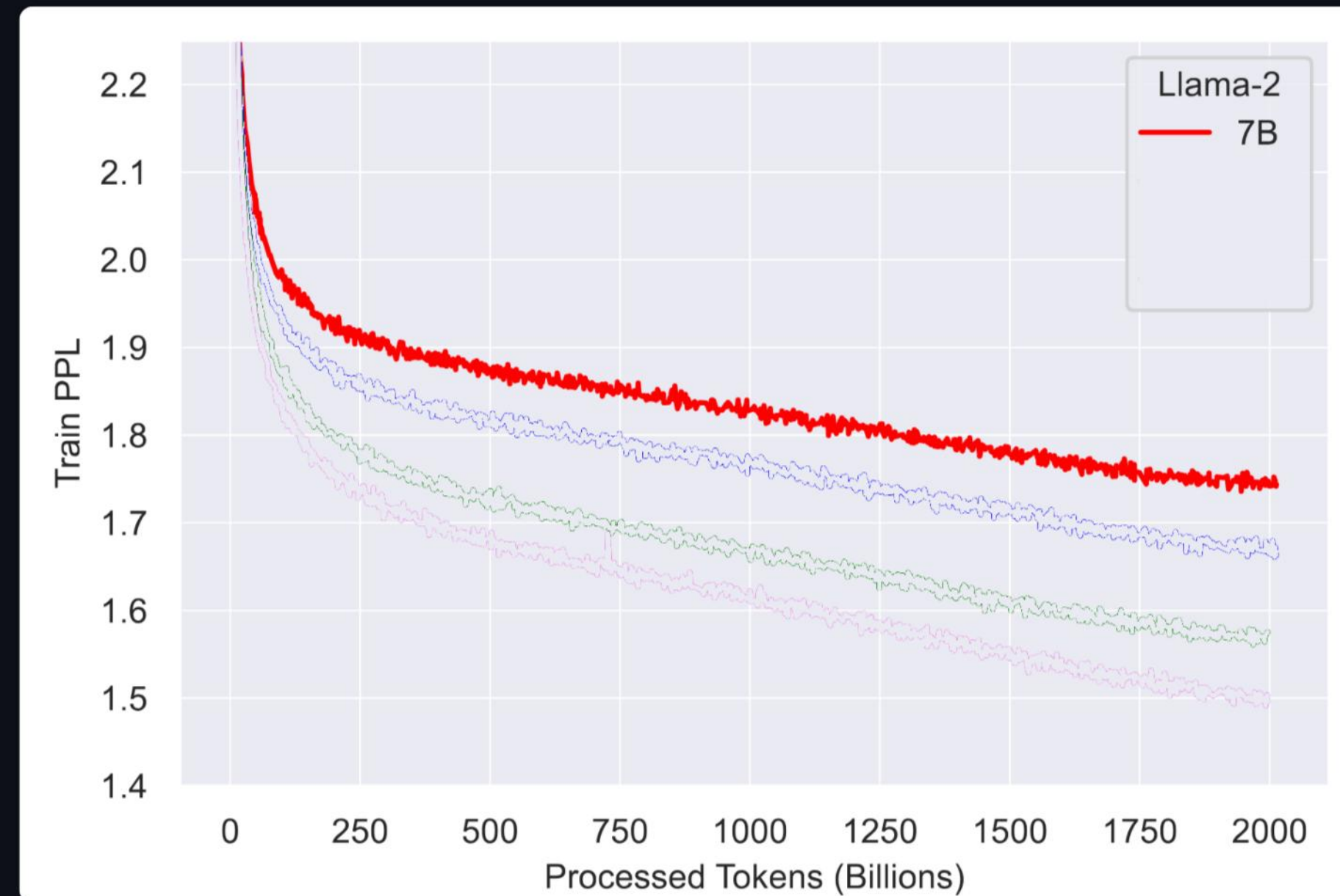
Key Takeaway

Training DNNs is
compute intensive

GPUs excel at parallel
matrix operations

WHY TRAIN ON LARGER DATASETS?

Llama 2 7B — Meta's open-source large language model (2023)

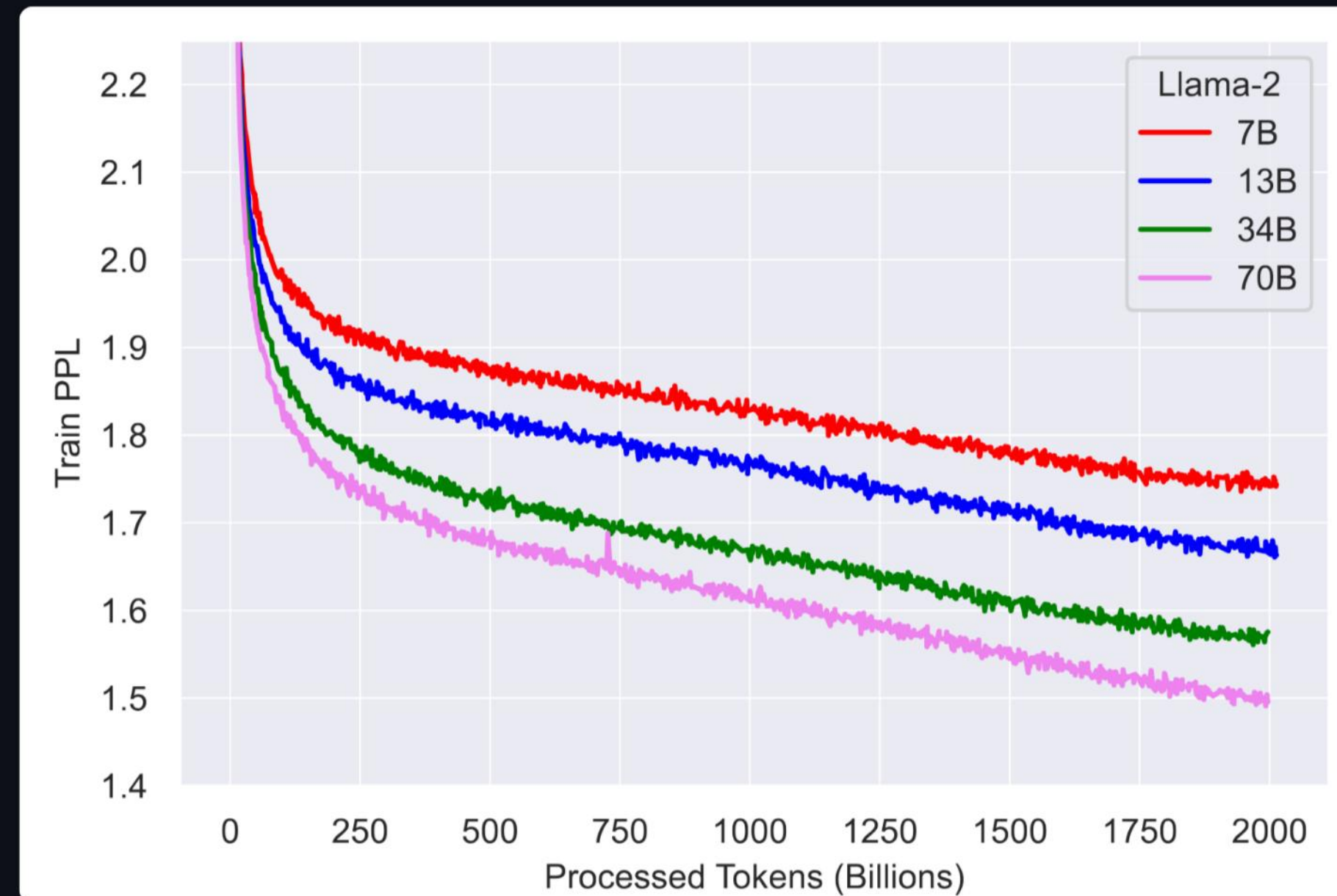


More training data → Lower loss → Better fit to the data

Kaplan et al. (2020) formalized this observation:
Loss decreases as a **power law** with dataset size

WHY TRAIN LARGER MODELS?

Llama 2 — 7B, 13B, 34B, and 70B parameter variants

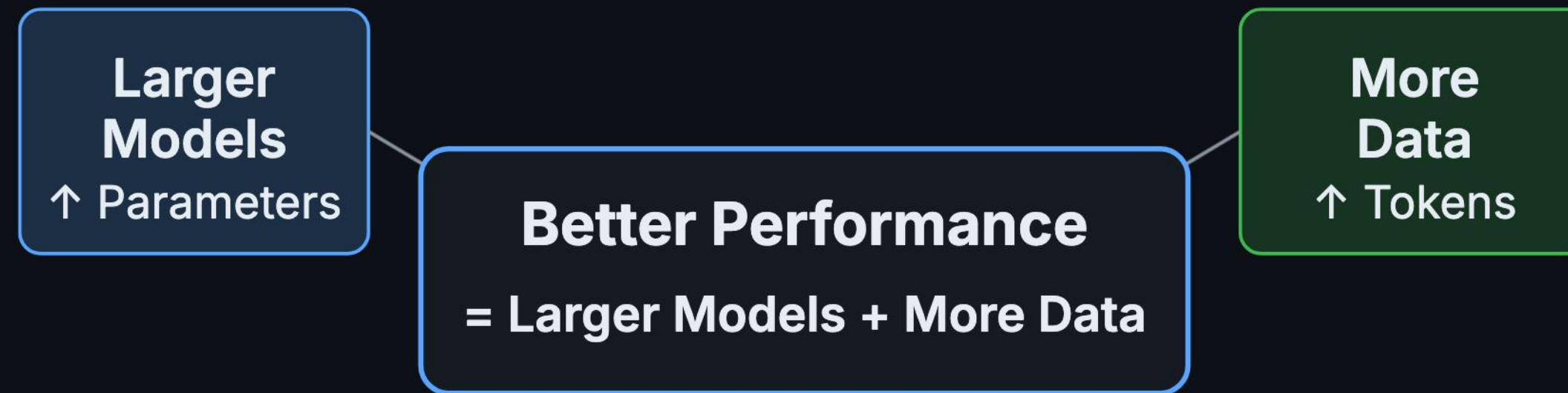


Larger model + same data → Lower loss → More capable model

Kaplan et al. (2020) formalized this observation:
Loss decreases as a **power law** with model size

RECAP: THE SCALING IMPERATIVE

Scaling Laws (Kaplan et al., 2020) + Chinchilla (Hoffmann et al., 2022)



GPT-3

(2020)

175B params, 300B tokens

LLaMA 2

(2023)

70B params, 2T tokens

LFM2.5-1.2B

(2026)

1.2B params, 28T tokens

DATA PARALLELISM

Scale dataset throughput by distributing batches across GPUs

Key insight: Samples in a batch are independent — only gradient aggregation requires communication

1. Replicate

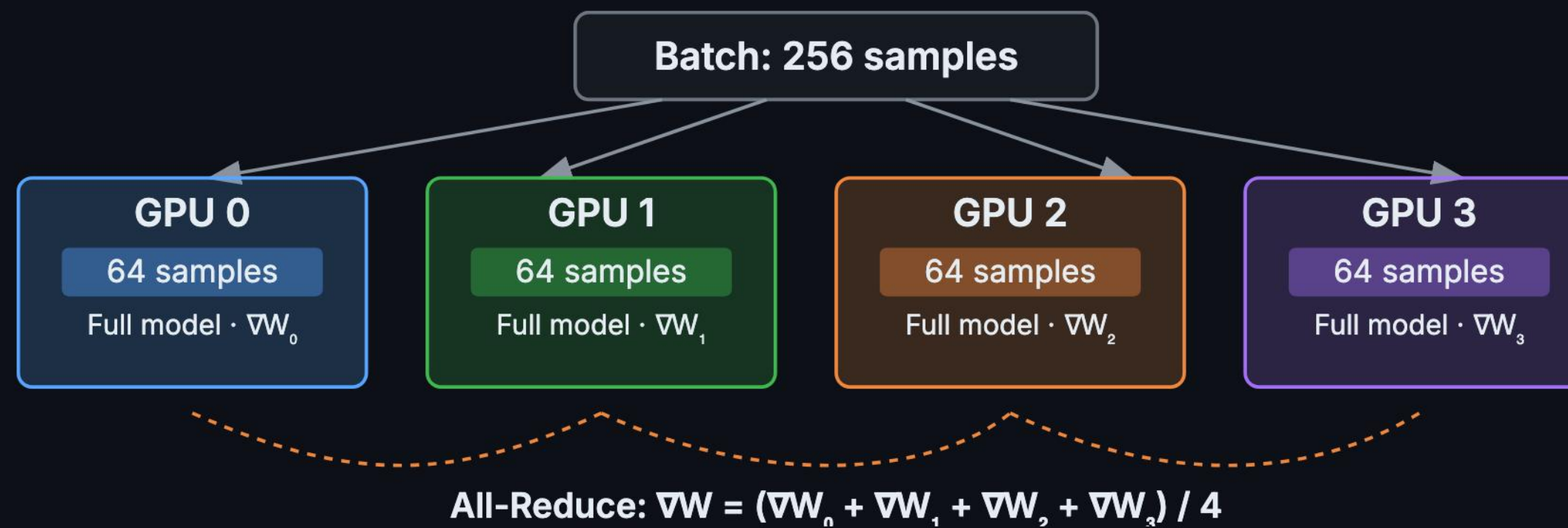
Full model copy on each GPU

2. Distribute

Each GPU processes different batch

3. Synchronize

Aggregate gradients across GPUs



Linear

Throughput scales linearly with GPUs

Batch Size

Effective batch size increases with GPUs

Limitation

Large batches can hurt generalization

MEMORY REQUIREMENTS FOR TRAINING

What's the largest model that fits on one GPU?

Example: 1B parameter model, batch size 16, 256 layers, hidden size 1024



Memory Breakdown

Parameters (bf16) 1B × 2 bytes	2 GB
Gradients (bf16) Same size as parameters	2 GB
Adam Optimizer States (fp32) 2 moments × 4 bytes × 1B params	8 GB
Activations (all layers) 256 layers × batch 16 × hidden 1024 × 2 bytes	~16 GB

Total ~28 GB

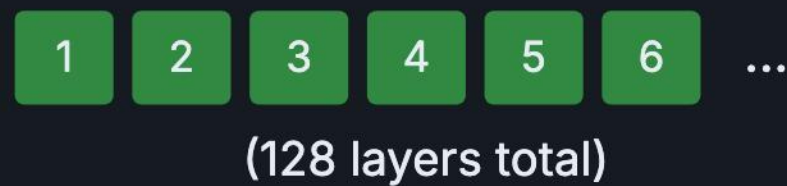
For 70B model: ~2 TB needed
→ Need strategies to reduce memory!

ACTIVATION CHECKPOINTING

Trade compute for memory: recompute activations instead of storing all

WITHOUT CHECKPOINTING

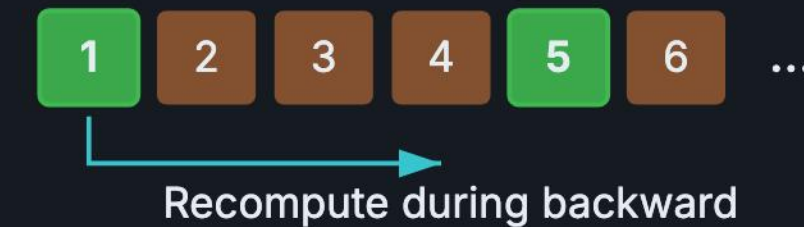
All layer activations stored



~8 GB activations

WITH CHECKPOINTING

Only checkpoints stored



Checkpoint Discarded

~2 GB activations

4x

Memory reduction

~30%

Extra compute (recompute)
 $\text{FWD} + 2 \times \text{BWD} \rightarrow \text{FWD} + 2 \times \text{BWD} + \text{FWD}$

Worth it

In practice, memory is more
precious than compute

OFFLOADING TO CPU MEMORY

Idea: Push infrequently used data to slower but larger memory tier

WHAT TO OFFLOAD?

- **Optimizer states** — only needed during parameter update
- **Inactive layer parameters** — fetch when needed
- **Activations** — stored for backward pass

WHY IT CAN WORK

Data is accessed *infrequently* → transfer overhead amortized over many compute operations

⚠ THE BANDWIDTH PROBLEM

3.35 TB/s

HBM (H100)

64 GB/s

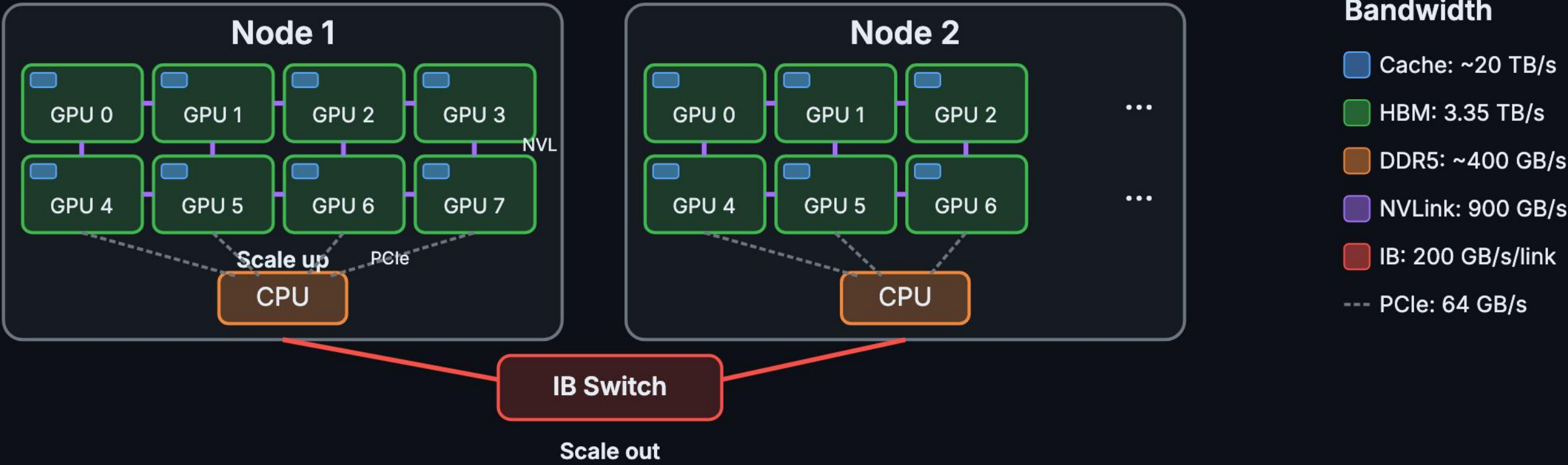
PCIe 5.0

52× slower to fetch from CPU memory
Must pass through PCIe bus + DDR memory

NICHE APPLICATIONS

- **Fine-tuning very large models** on limited hardware
- **Context length extension** (long sequences)
- Scenarios where *very little training* is needed

MEMORY & NETWORK BANDWIDTH



RDMA (Remote Direct Memory Access)

GPU-to-GPU transfer across nodes without CPU involvement



Bypasses CPU + OS

SHARDING

Definition: Sharding

Partition data across multiple machines to distribute compute and memory



For Neural Network Training

Model Weights
Parameters W, b

Optimizer States
Adam: m, v

Activations
Forward pass outputs

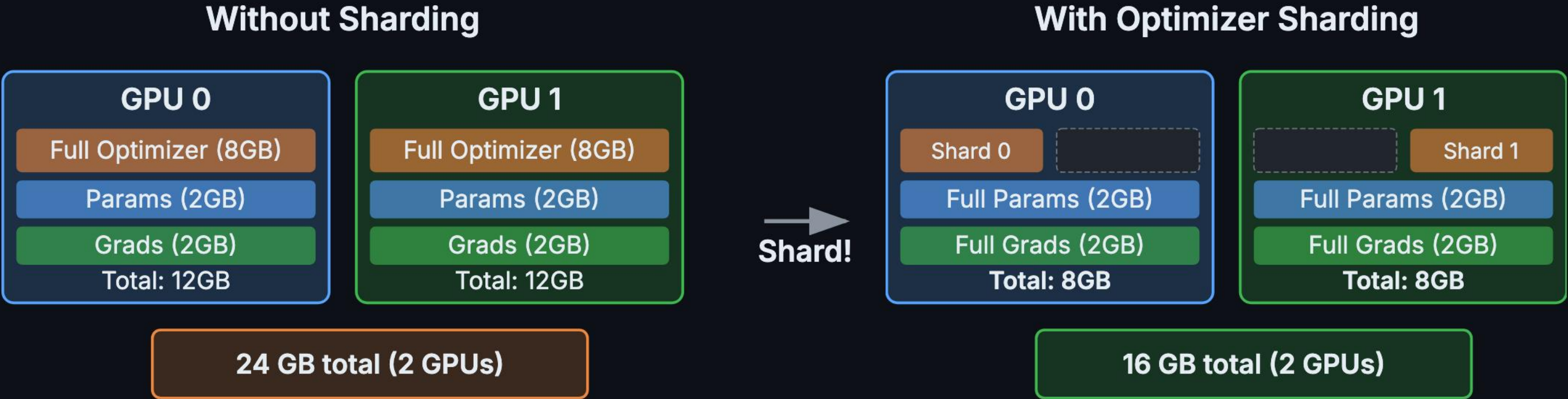
Data / Gradients
Batches, dL/dW

Key Trade-off

Sharding reduces memory per device, but requires communication to gather data

OPTIMIZER STATE SHARDING

Optimizer states are huge (~8 bytes/param for Adam) but only needed during weight update



Update Process:

- | | |
|---------------------|----------------------------|
| 1. Forward/Backward | — Compute local gradients |
| 2. Reduce-Scatter | — Each GPU gets grad shard |
| 3. Local Update | — Update params with shard |
| 4. All-Gather | — Broadcast updated params |

Memory Savings: 33% (optimizer states only)

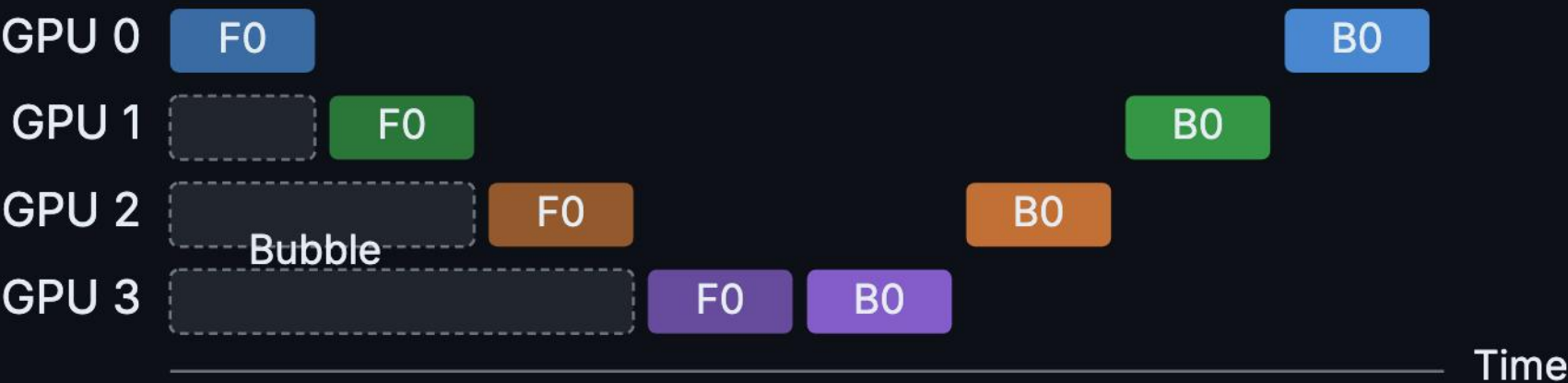
PIPELINE PARALLELISM

Split model into stages, process micro-batches in sequence

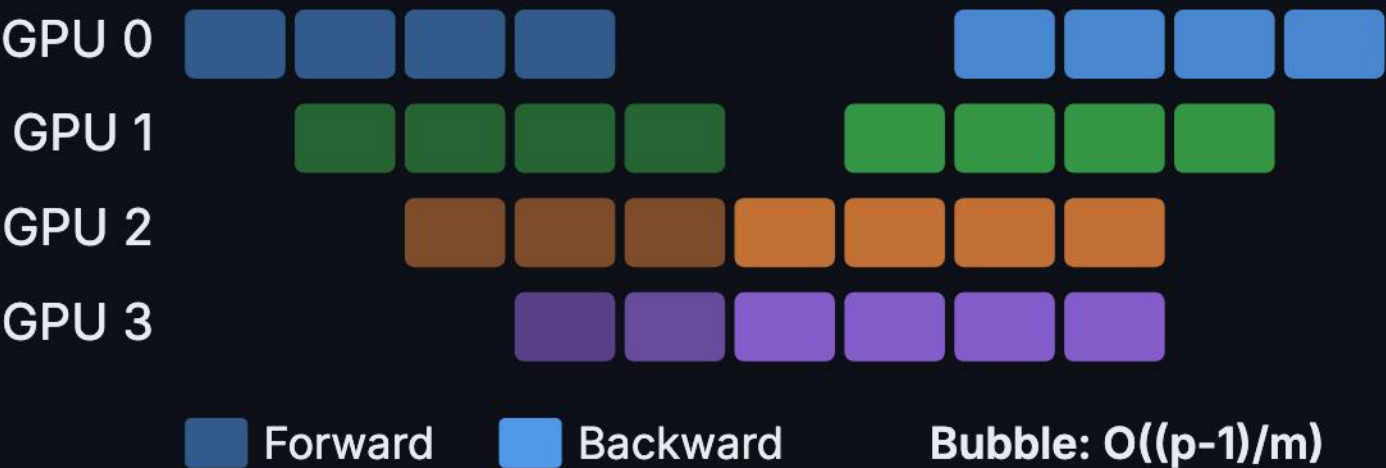
Model Split into 4 Stages



Naive Pipeline: Bubble Problem



Solution: Micro-batching



Further Optimizations

Llama 3: Bidirectional scheduling overlaps forward and backward passes

DualPipe (DeepSeek-V3): Finer granularity — separates backward-for-input vs backward-for-weights

TENSOR PARALLELISM

Split individual weight matrices across GPUs

Matrix Multiplication

$$X \times W = Y$$

Column Parallelism

$$X \times \begin{matrix} W1 \\ \text{GPU 0} \end{matrix} \begin{matrix} W2 \\ \text{GPU 1} \end{matrix} = \begin{matrix} Y1 \\ Y2 \end{matrix}$$

(No collective)

Row Parallelism

$$\begin{matrix} X1 \\ X2 \end{matrix} \times \begin{matrix} W1 \\ \text{GPU 0} \end{matrix} \begin{matrix} W2 \\ \text{GPU 1} \end{matrix} = \begin{matrix} Y1 \\ + \\ Y2 \end{matrix} \rightarrow Y$$

All-Reduce

Transformer MLP: Combine Both

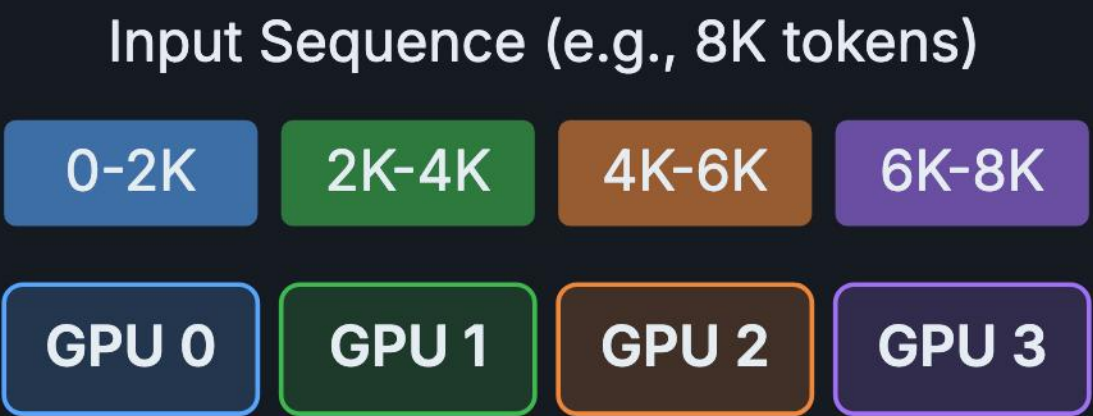


Column → Row pairing: Only 1 All-Reduce per layer
Requires high-bandwidth interconnect (NVLink)

SEQUENCE & CONTEXT PARALLELISM

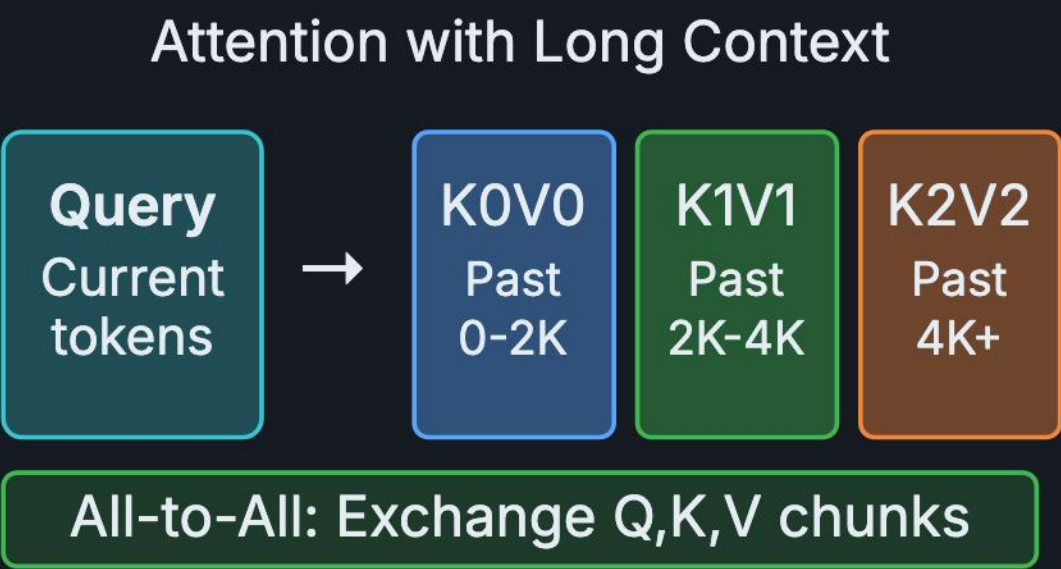
For very long sequences: split the sequence dimension across GPUs

SEQUENCE PARALLELISM



Shards activations along sequence
MLPs, LayerNorm are time-independent
Reduces activation memory per GPU

CONTEXT PARALLELISM



Shards KV cache across GPUs
Enables 100K+ token contexts

SP

Shards MLP/LayerNorm
activations

CP

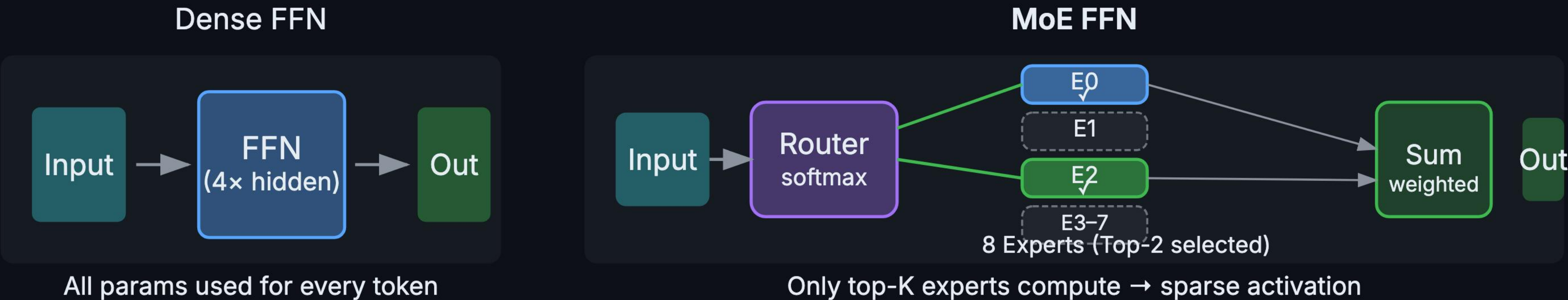
Shards attention KV cache

Combined

Train with 1M+ token contexts

MIXTURE OF EXPERTS (MOE)

Scale model capacity without proportionally increasing compute



Expert Parallelism

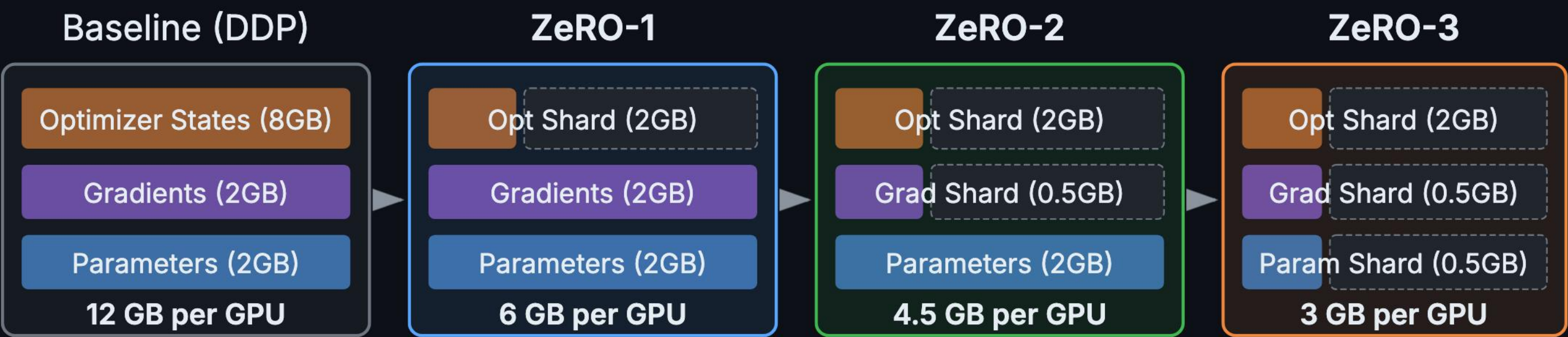


All-to-All: route tokens to the correct experts

LFM2-8B-A1B 8.3B total, 1.5B active	Capacity 8× params, ~2× compute	Challenge Load-balancing experts
---	---	--

DEEPSPEED ZERO

Zero Redundancy Optimizer: Progressive sharding for memory efficiency



Communication Patterns

Stage	Sharded	Communication	Memory
ZeRO-1	Opt states	Reduce-Scatter + All-Gather	4x reduction
ZeRO-2	Opt + Grad	+ Reduce-Scatter gradients	8x reduction
ZeRO-3	All	+ All-Gather params (fwd/bwd)	Linear scaling

ZeRO-3 enables training models that don't fit on a single GPU
Trade-off: More communication ↔ Less memory per device

FULLY SHARED DATA PARALLEL (FSDP)

PyTorch's native implementation of ZeRO-3 style sharding

FSDP Unit Wrapping

Transformer Model

FSDP(TransformerBlock 0)

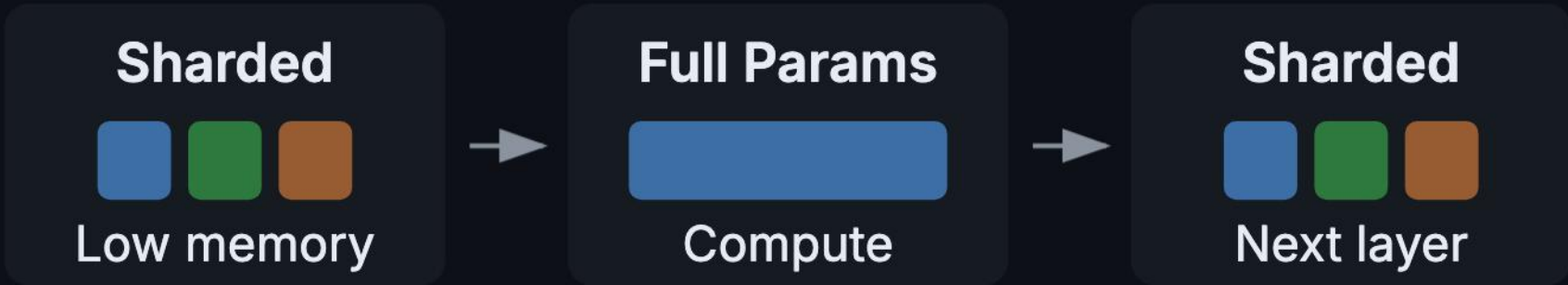
FSDP(TransformerBlock 1)

FSDP(TransformerBlock 2...N)

Each FSDP unit manages its own sharding
Params gathered just-in-time for compute

Execution Flow

Forward pass: gather → compute → discard



Backward Pass

All-Gather params → Compute grads → Reduce-Scatter
Gradients stay sharded across GPUs

FSDP Sharding Strategies

FULL_SHARD

Params + grads + optimizer
Max memory savings
Most communication

SHARD_GRAD_OP

Grads + optimizer only
Params replicated
Less communication

NO_SHARD

DDP behavior
Full replicas
Least communication

FSDP + activation checkpointing
Train large models on commodity hardware

MEGATRON-LM

NVIDIA's framework for training large transformer models

Distributed Training Frameworks

DeepSpeed ZeRO

Microsoft

FSDP

Meta (PyTorch)

Megatron-LM

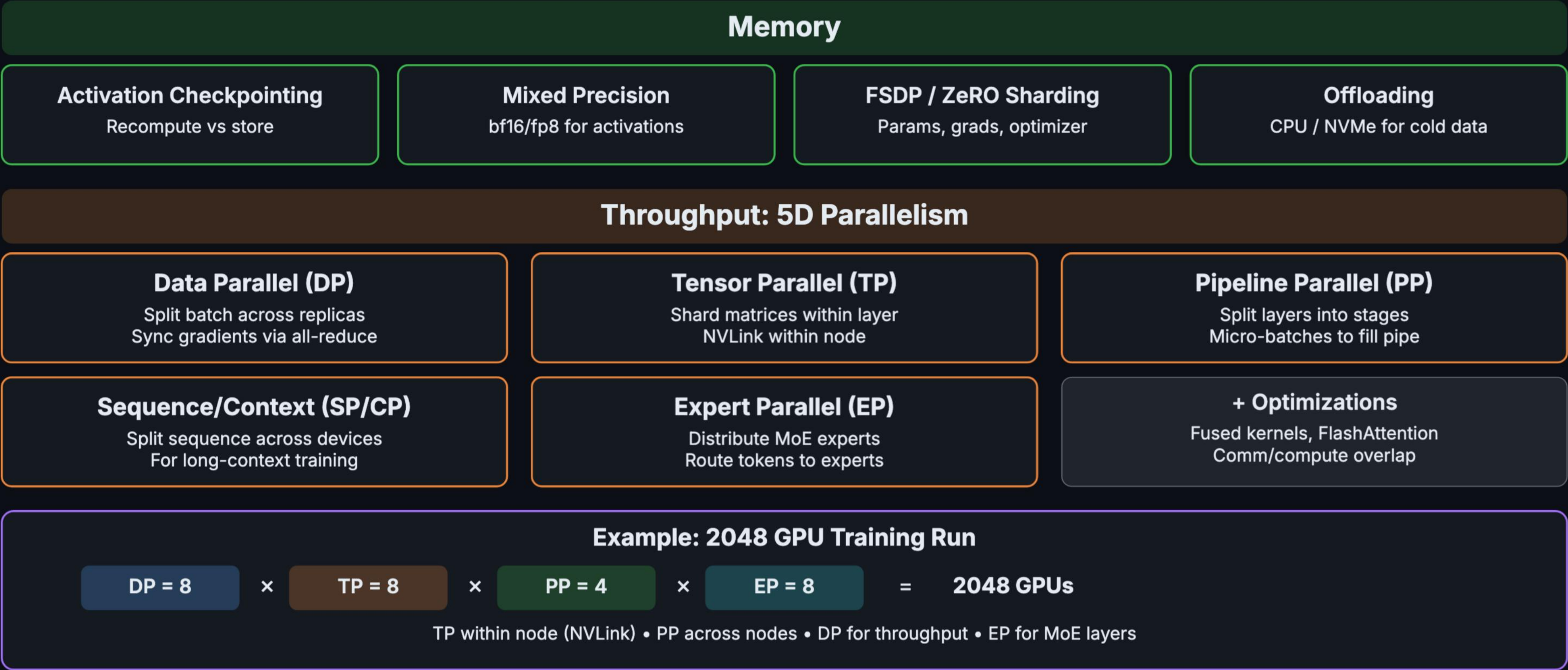
NVIDIA

Feature Comparison

Aspect	DeepSpeed ZeRO	FSDP	Megatron
Focus	Memory	Memory	Parallelism
PP / TP	Limited	No	Native
Philosophy	Wrap model	nn.Module	"MPU"
Ease of use	config.yaml	PyTorch native	Custom code

PUTTING IT ALL TOGETHER

In practice: combine techniques based on model size and cluster topology



SUMMARY

Why Scale?

- Scaling laws: bigger → better
- More data → lower loss
- GPUs: ~600× faster than CPUs
- One GPU can't fit big models

Memory Solutions

- Activation checkpointing
- CPU/NVMe offloading
- ZeRO / FSDP sharding
- Mixed precision (bf16/fp8)

Parallelism Strategies

- Data parallel: split batch
- Tensor parallel: shard tensors
- Pipeline parallel: split stages
- Expert parallel: split MoE

Key Principle

Match parallelism strategy to your bandwidth hierarchy

Communication Cost Hierarchy

Cache

>

HBM
(GPU memory)

>

NVLink
(within node)

>

InfiniBand
(within cluster)

>

...

No single strategy suffices — **combine them** based on model size, cluster topology, and batch requirements