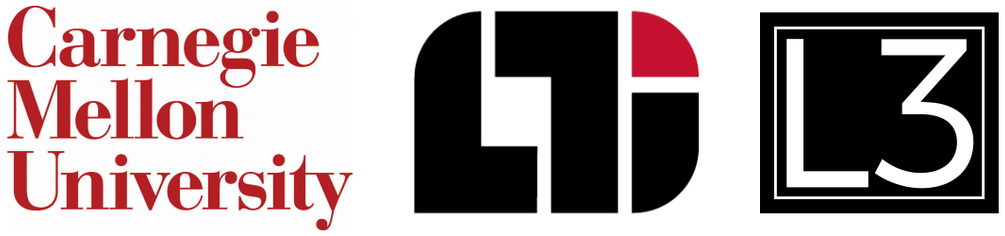
CS11-711 Advanced NLP

Parallelism and Scaling

Sean Welleck

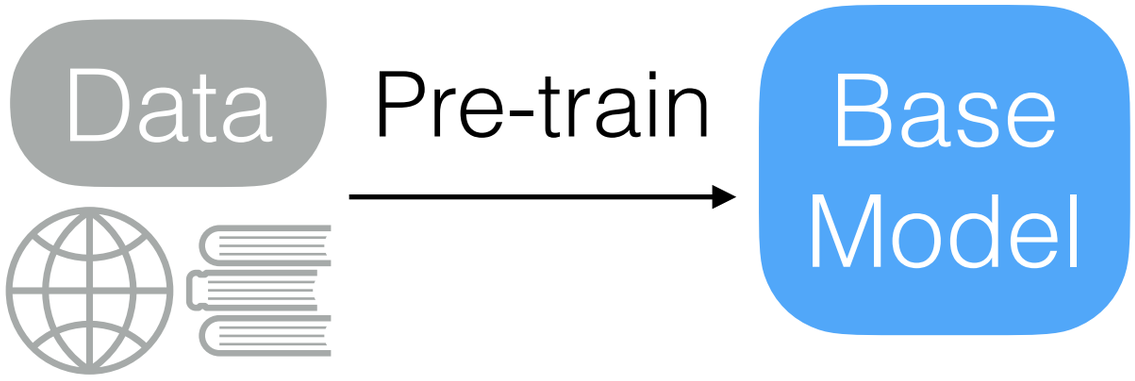


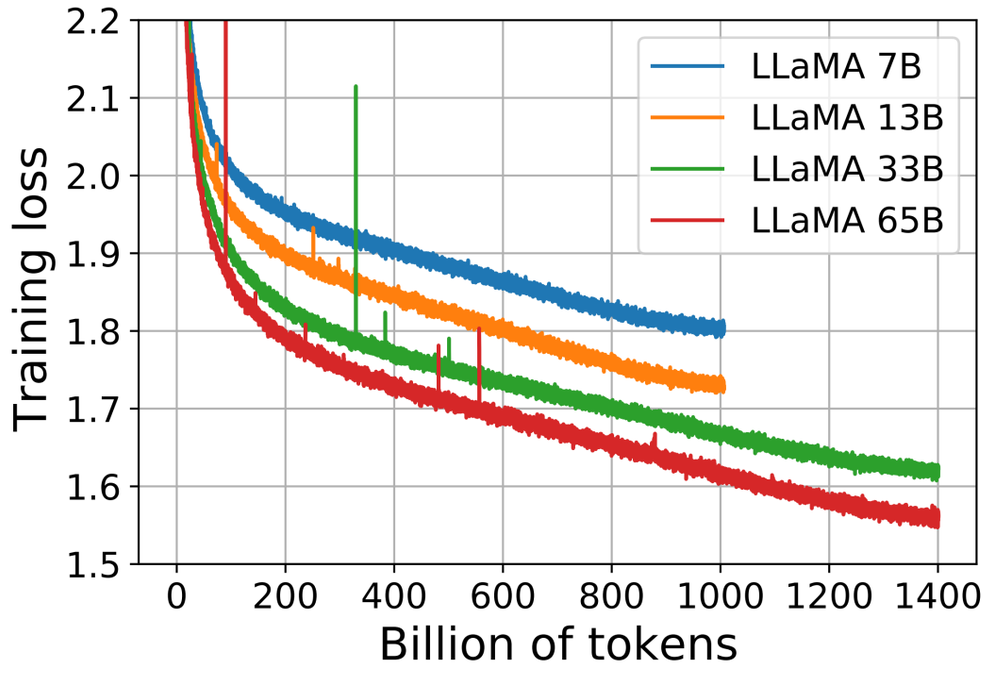
https://cmu-l3.github.io/anlp-fall2025/

https://github.com/cmu-l3/anlp-fall2025-code

Content based on *The Ultra-Scale Playbook: Training LLMs on GPU Clusters*

Recap: pre-training





Bigger model

Better Loss

More data

Scale the training of LLMs

- Key problem: take advantage of multiple devices (e.g., GPUs)
- Train larger models
- Process more tokens in a given amount of time

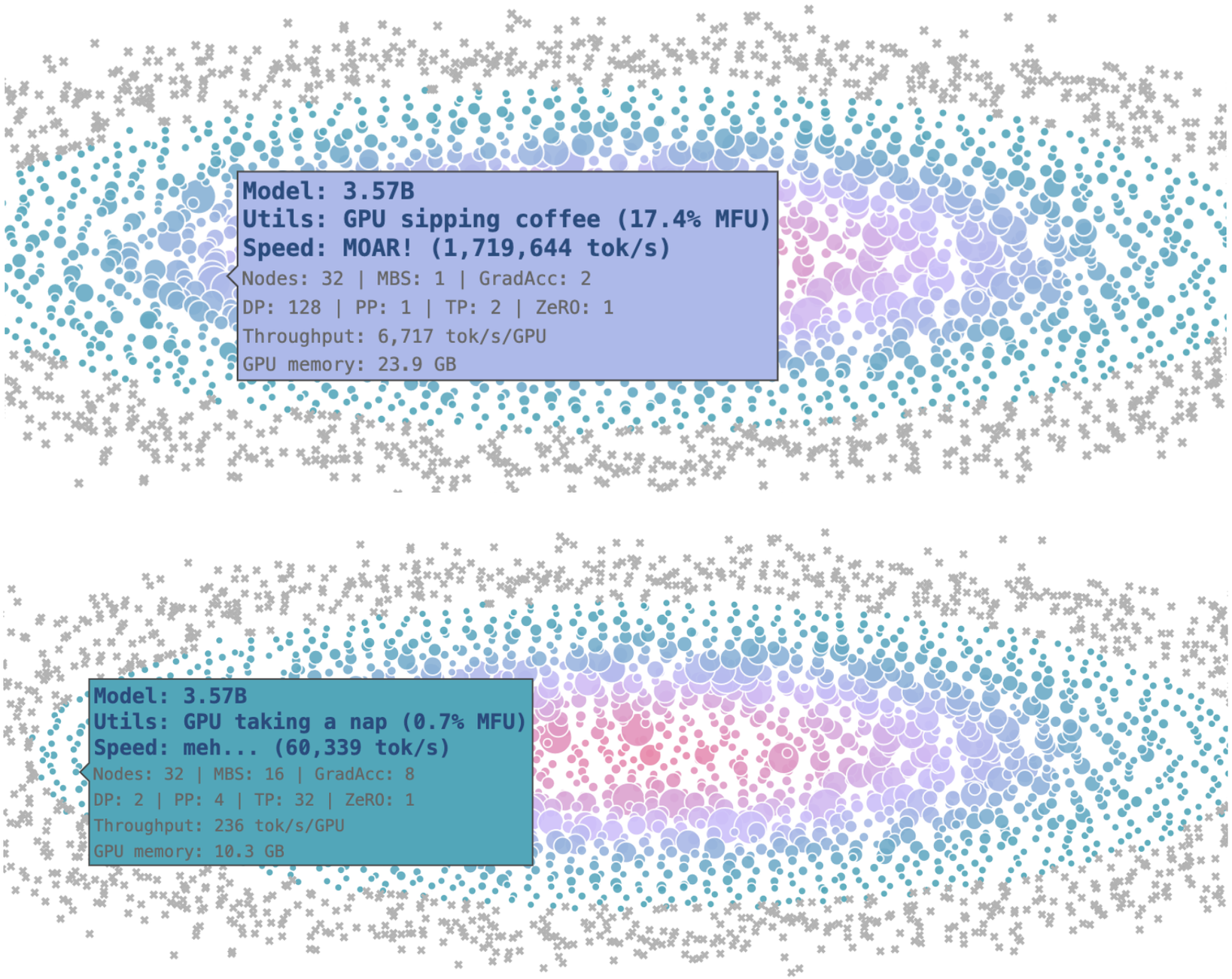
Scale the training of LLMs

- **Memory usage**: training steps need to fit in memory

- **Compute efficiency**: we want our hardware to spend most time computing

- **Communication overhead**: minimize since it keeps GPUs idle

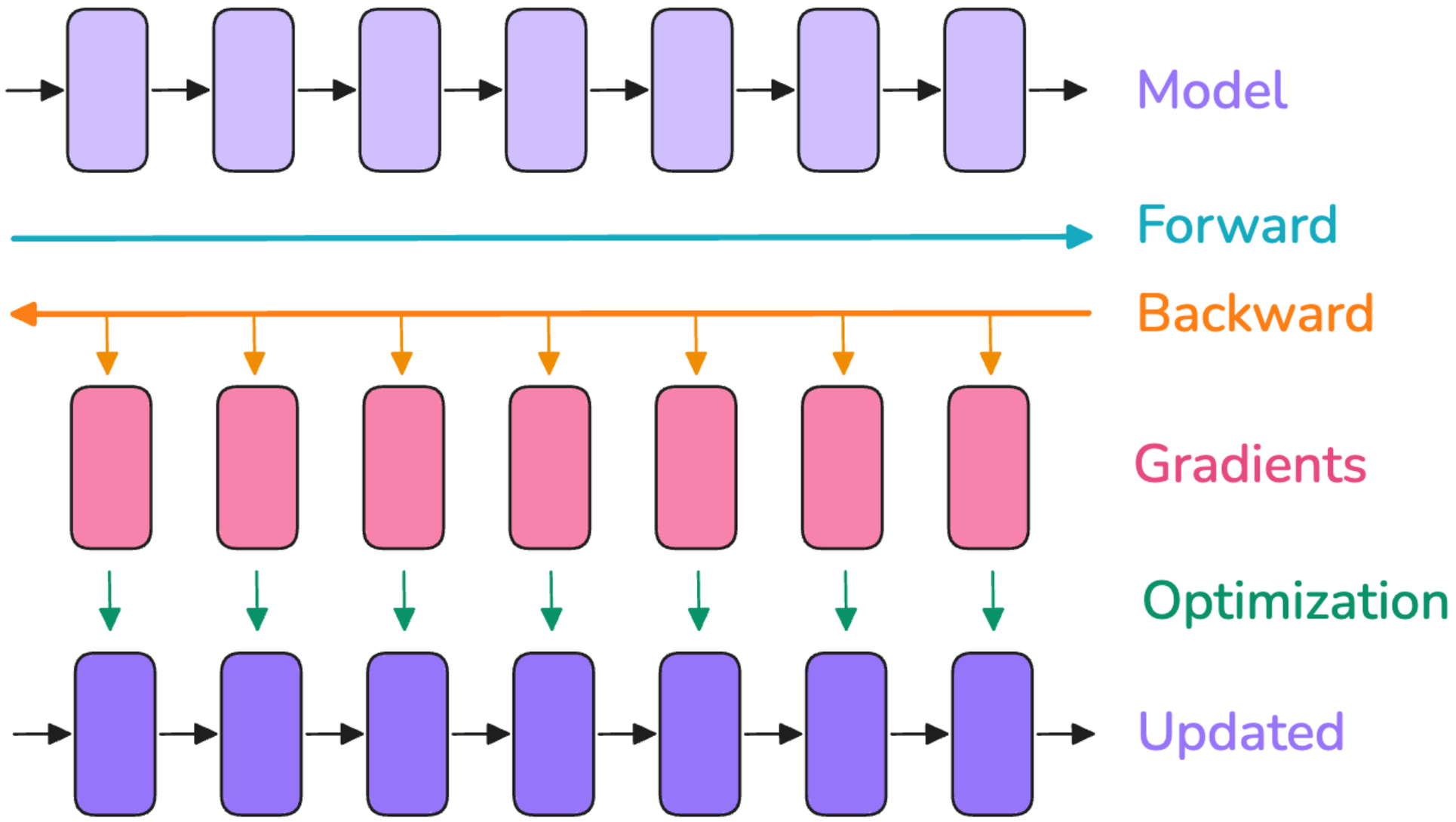
Large impact



Today's lecture

- Basics of training on one GPU
- Parallelization on multiple GPUs
  - Data, tensor, pipeline parallelism, ZeRO
- Choosing and comparing strategies

Training on one GPU



Training basics

- Compute
- Memory
  - Activation recomputation
  - Gradient accumulation

Compute

- Compute: floating point operations (FLOP)
  - Forward and backward pass:

$$6 \times \text{model\_parameters} \times \text{token\_batch\_size}$$

- FLOPS: floating point operations per second

Compute

- **Model FLOP Utilization (MFU)** measures how effectively available compute is used for training

$$\text{MFU} = \frac{\text{Achieved FLOPS}}{\text{Theoretical Peak FLOPS}}$$

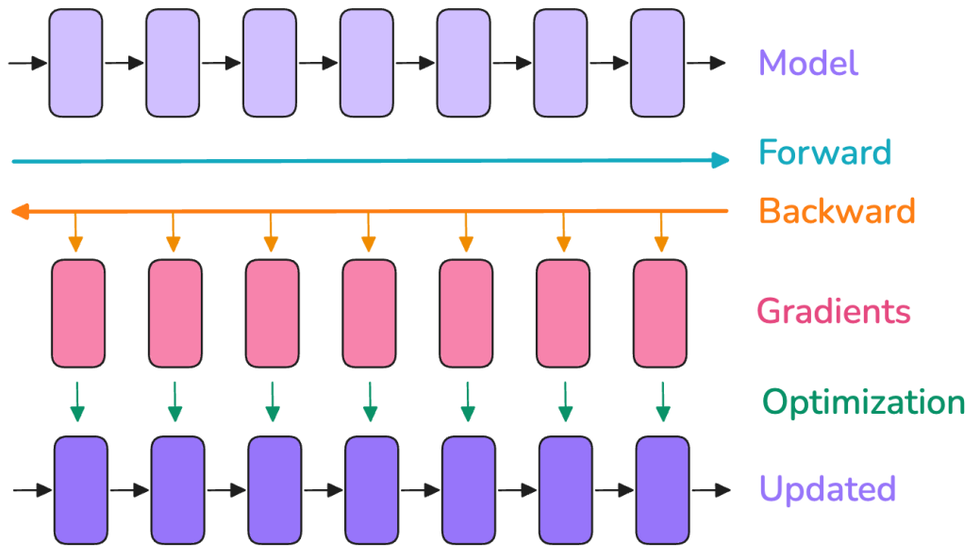
- Theoretical peak (H100):

Technical Specifications	
	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS

- Inefficiencies: communication, memory bandwidth, idle time (discussed later!)

Memory usage

- Weights, gradients, optimizer states, activations
  - Tensors with shapes and precisions



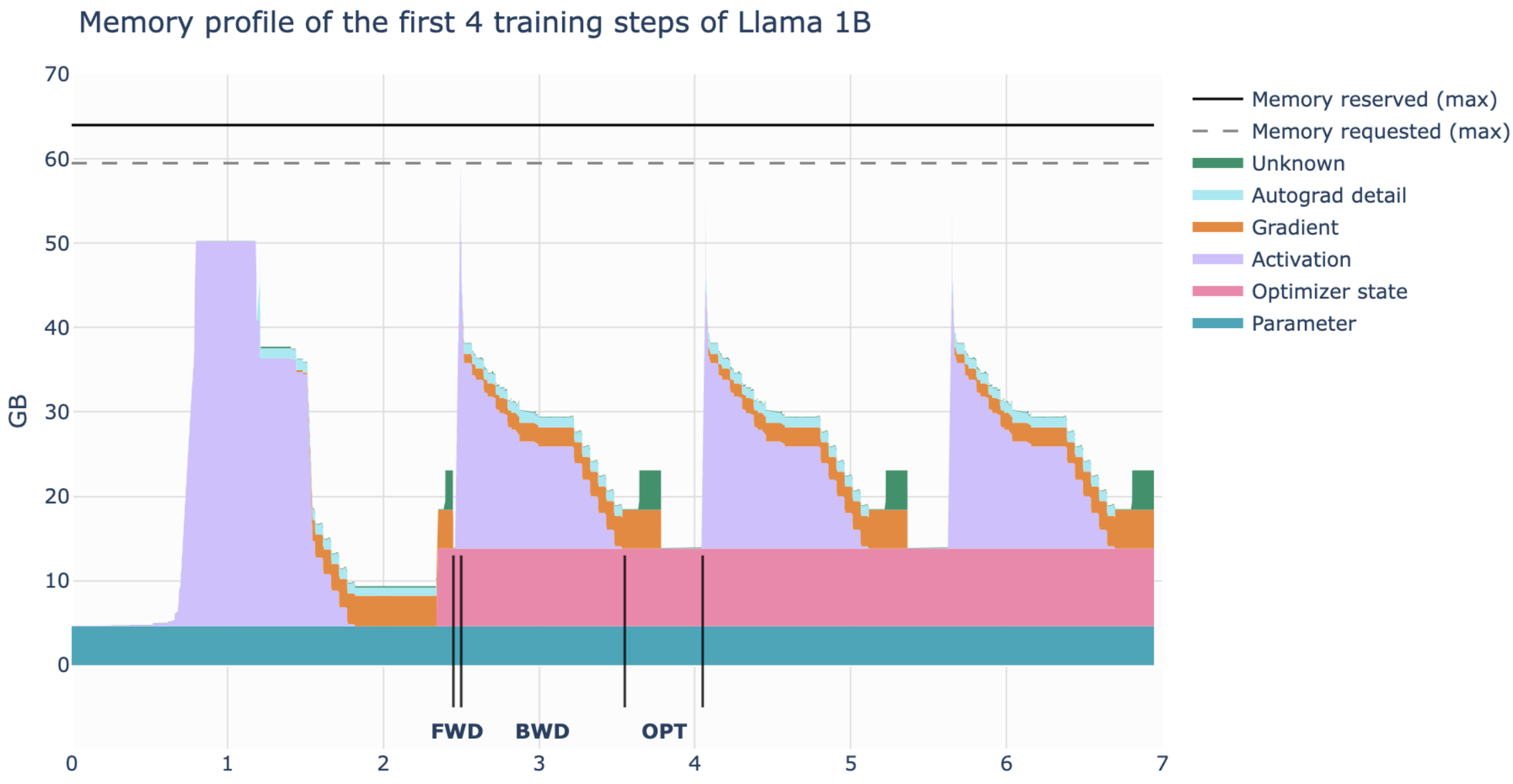
Memory usage

- A rough approximation for a training step:

peak_memory = model_bf16 + model_fp32 + grads_fp32 + optim_states + activations

- BF16 model: 2 * num_parameters
- FP32 model/grads: 4 * num_parameters
- FP32 optimizer states: (4 + 4) * num_parameters
    - Adam momentum and variance

Memory usage



Memory usage

Model parameters	FP32 or BF16 w/o FP32 grad acc	BF16 w/ FP32 grad acc
1B	16 GB	20 GB
7B	112 GB	140 GB
70B	1120 GB	1400 GB
405B	6480 GB	8100 GB

H100 GPU: 80 GB

Batch size

- **Small**: adjust parameters quickly but noisily
- **Large**: adjust parameters accurately, fewer steps to train on a given dataset

$$bst = bs * seq$$

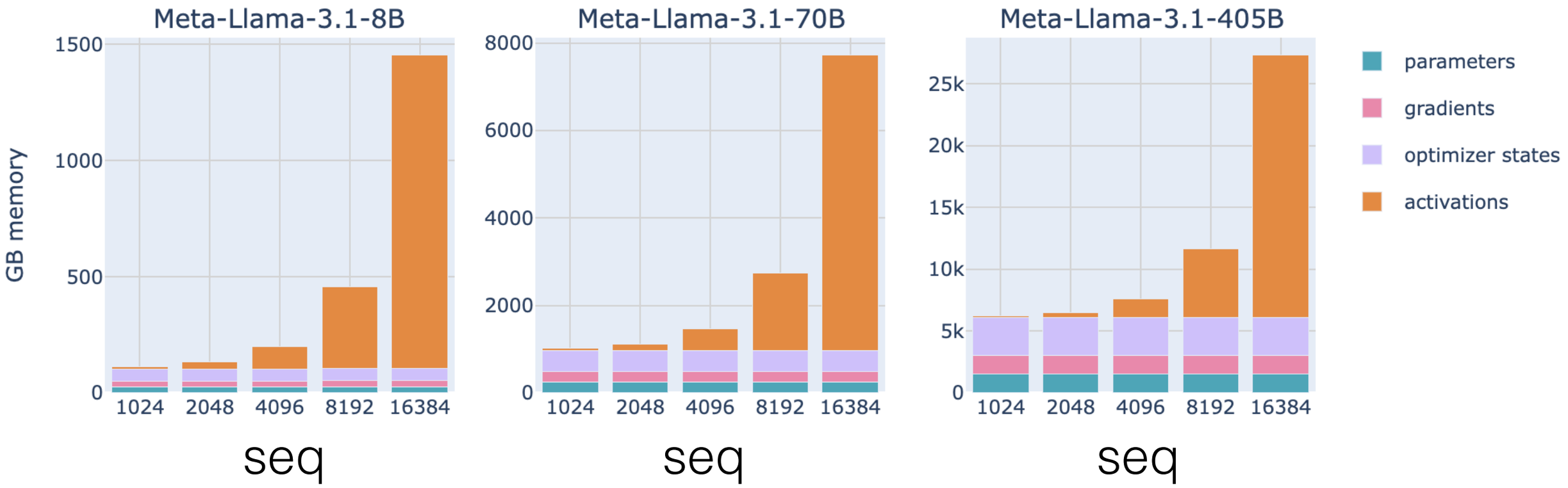
Typically ~4-60 million tokens per batch

- **Too large**: out of memory due to large activations!

Memory usage: activations

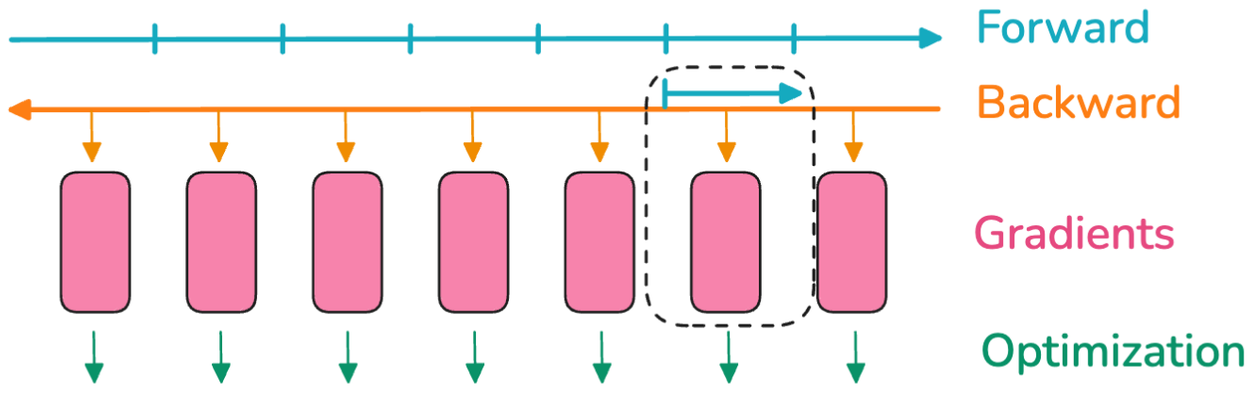
$$m_{act} = L \cdot seq \cdot bs \cdot h \cdot (34 + \frac{5 \cdot n_{heads} \cdot seq}{h})$$

- Linear with batch size, quadratic sequence length



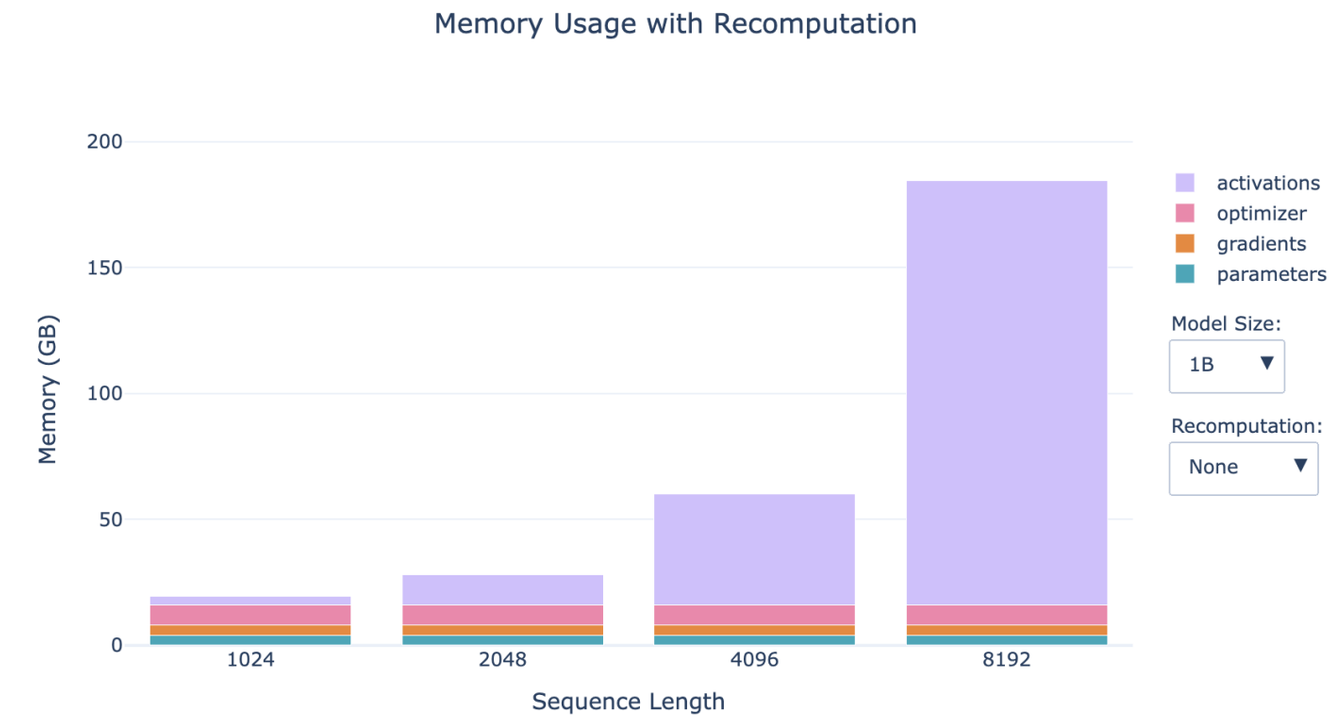
Activation recomputation

- Recompute some activations during the backward pass
  - Store some activations during the forward pass as “checkpoints”
  - Discard other activations and recompute them during the backward pass
- Increases compute, reduces activation memory requirements

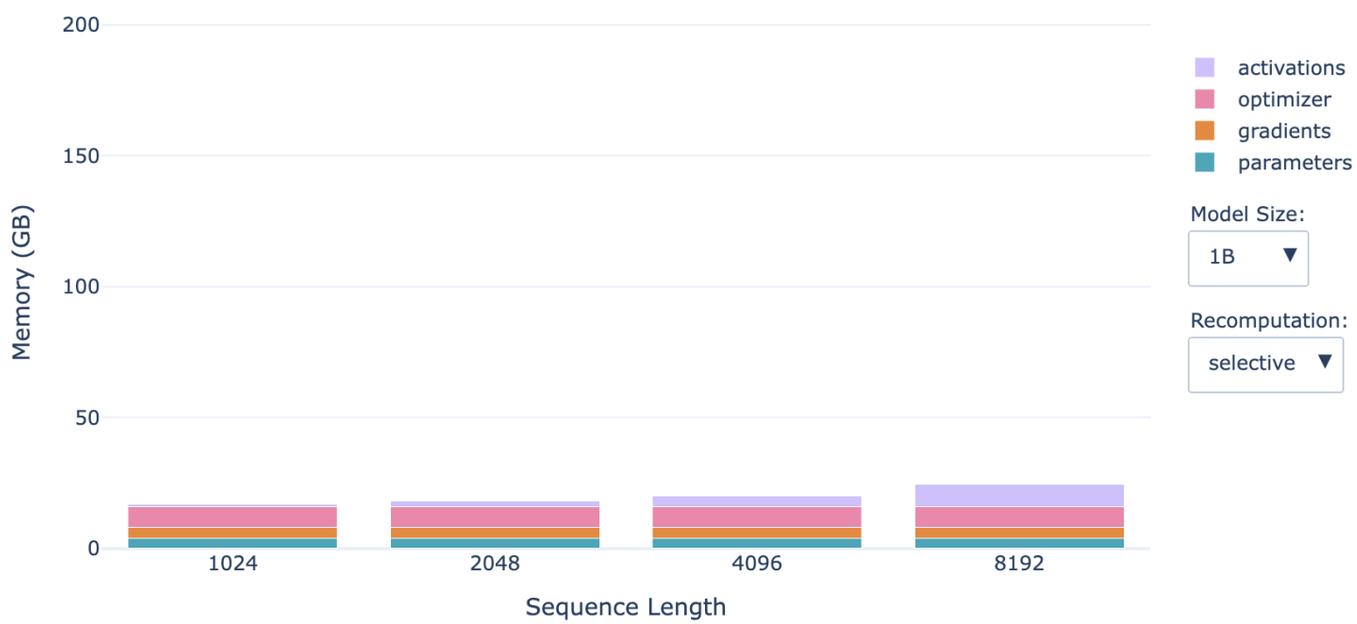


Activation recomputation

Without recomputation



With recomputation

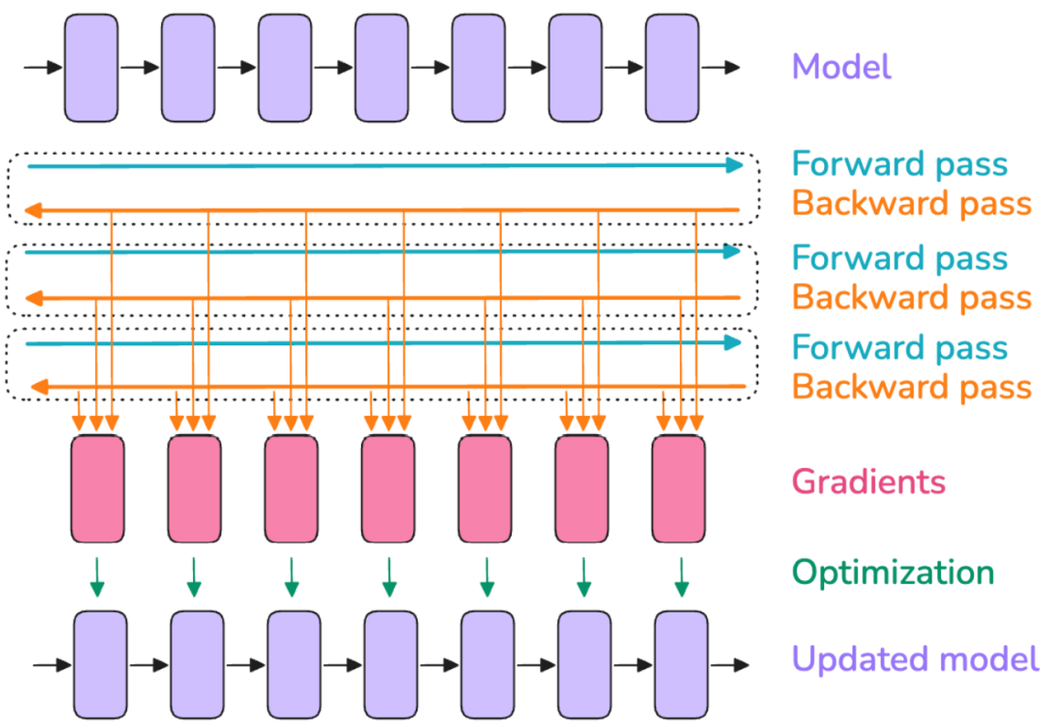


Gradient accumulation

- Split batch into micro-batches, do forward/backward passes on each micro-batch, average the gradients

$$bs = gbs = mbs \cdot grad\_acc$$

- Lets you increase batch size with constant memory



Recap: basics (single GPU)

- **Compute**: FLOPS and MFU
- **Memory**: parameters, gradients, optimizer states, activations
- **Activation recomputation**: save memory, add compute
- **Gradient accumulation**: save memory, add compute
- Use of memory savings: larger batch size and/or larger model

Multiple GPUs: Parallelism

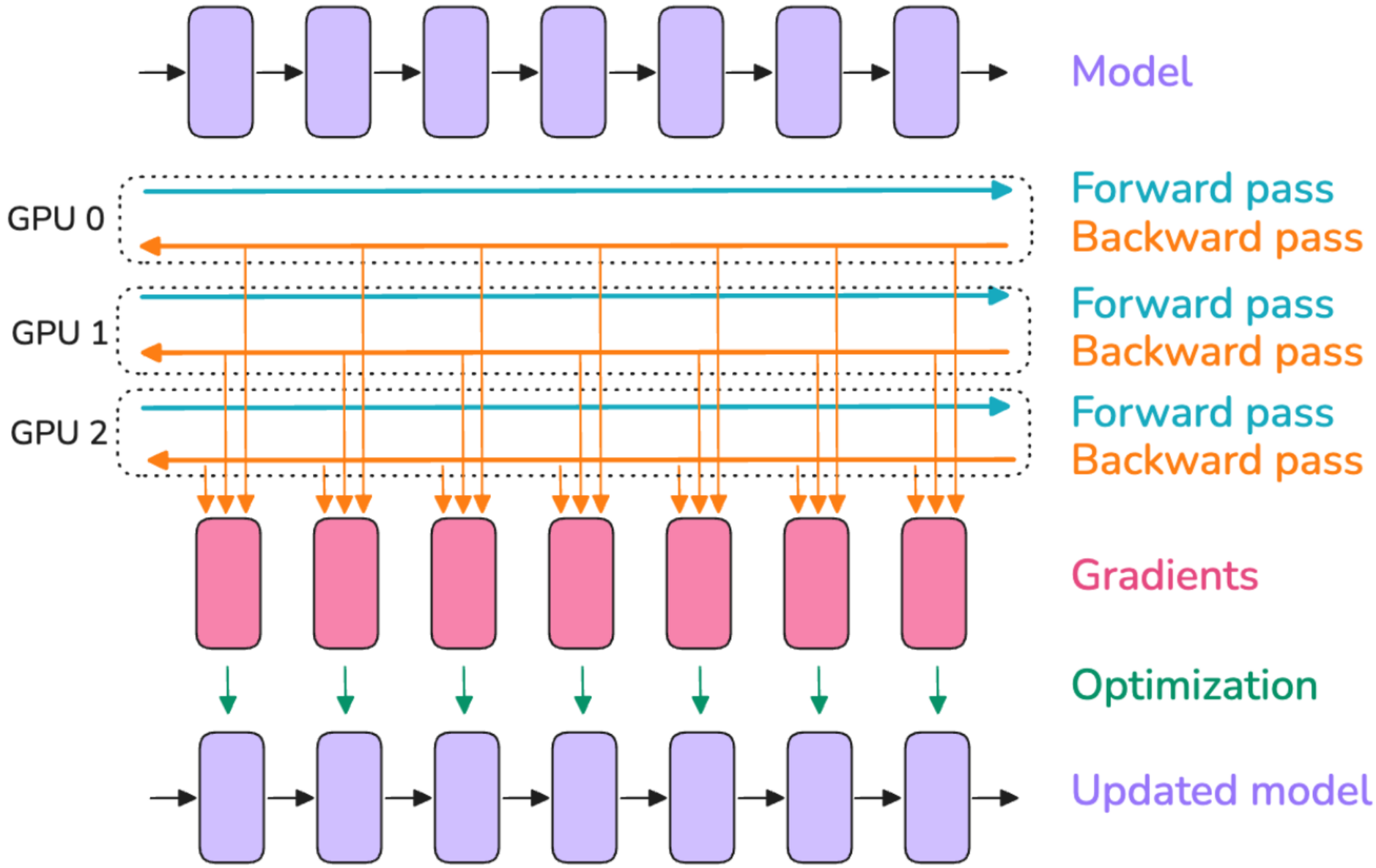
Parallelism

- Techniques for leveraging computation and memory from multiple GPUs
  - Data parallelism
  - Tensor parallelism
  - Pipeline parallelism
  - Memory optimization
  - Choosing parallelism strategies

Data Parallelism

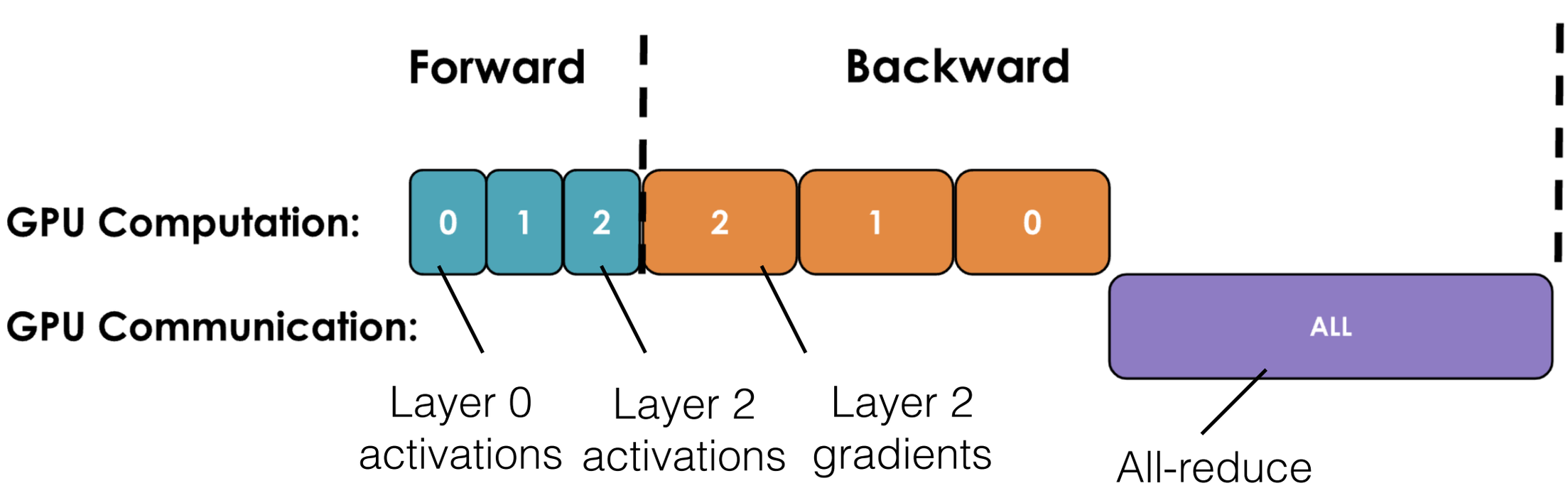
- Replicate model on several GPUs
- Run forward / backward passes on different micro-batches in parallel for each GPU
- Average the gradients across the GPUs

Data Parallelism



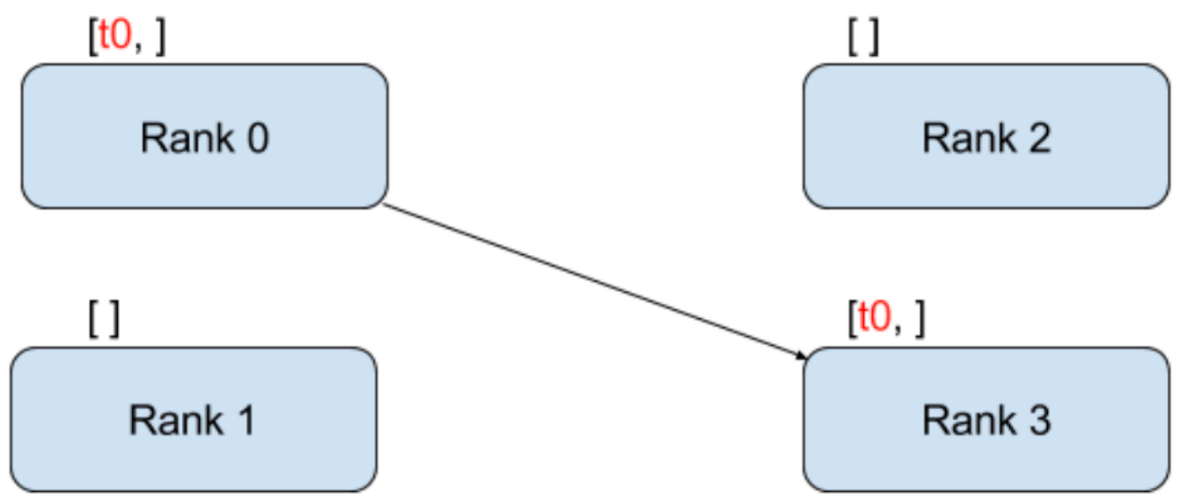
Data Parallelism: Naive

- Wait for all backward passes to finish, trigger an all-reduce over all GPUs



Aside: communication ops

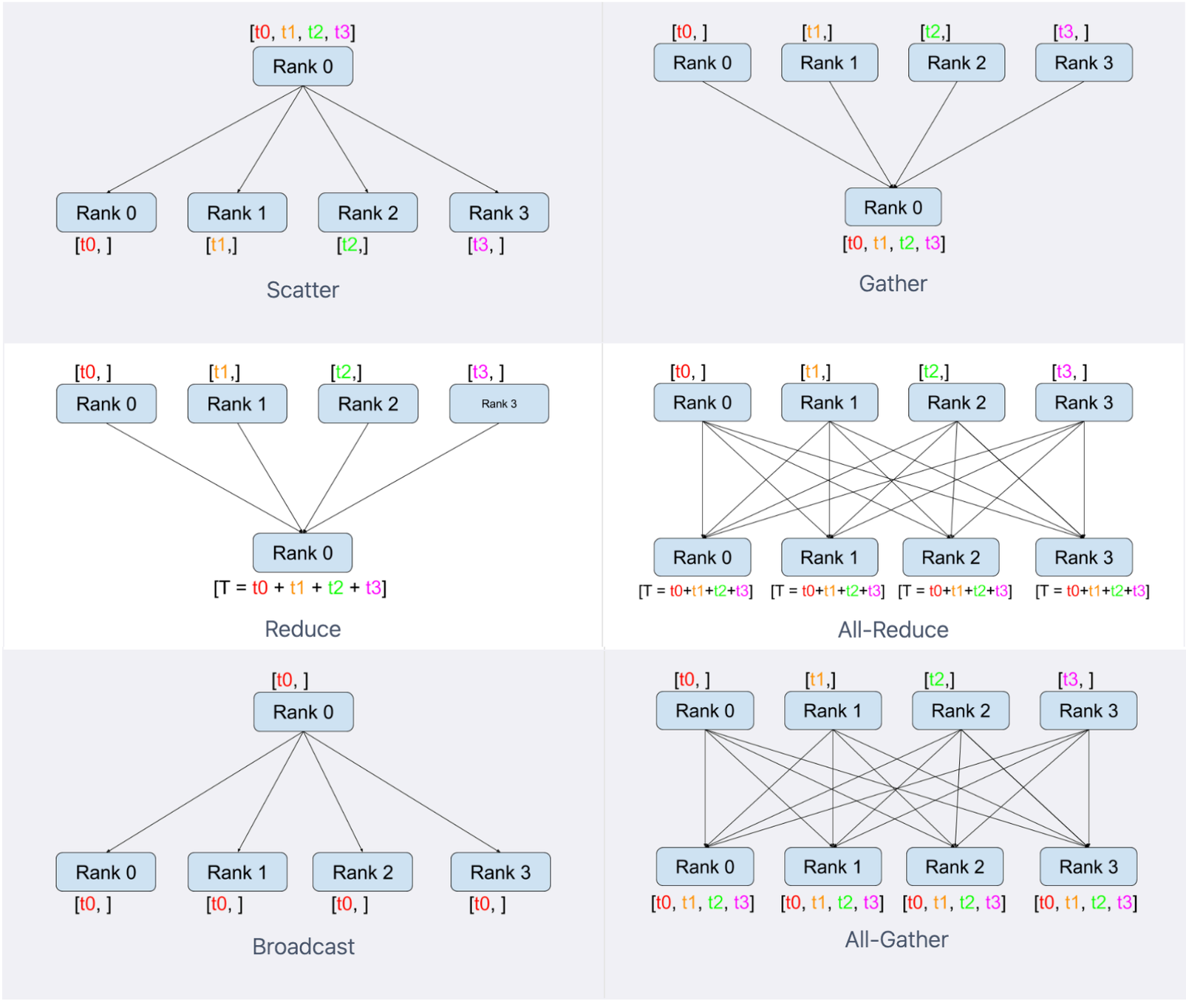
- Primitive operations for sending and aggregating information across multiple processes



Send and Recv

[code example]

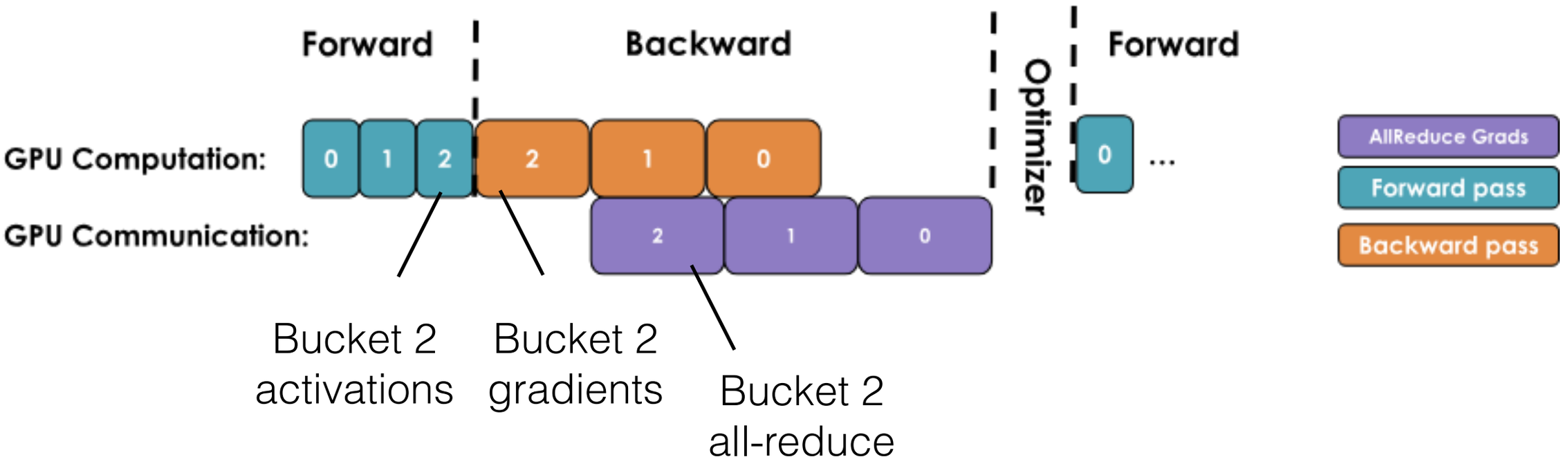
Aside: communication ops



[code example]

Overlap + bucketing

- Start all-reduce as soon as gradients are ready
- Group gradients into buckets and launch a single all-reduce for all the gradients in the same bucket



Data Parallelism: + bucketing

```
59    class BucketManager:
60        def __init__(self, params: List[torch.nn.Parameter], process_group: torch.distributed.ProcessGroup, bucket_size:
83
84        def _initialize_buckets(self) -> None:
85            """
86            Divides model parameters into buckets for gradient synchronization based on the bucket size.
87            """
88            cur_bucket_size = 0
89            cur_bucket_idx = 0
90
91            # Assign parameters to buckets.
92            for param in self.params:
93                if not param.requires_grad:
94                    continue
95
96                # If the bucket is empty, add the parameter to the bucket.
97                if cur_bucket_size == 0:
98                    self.params_to_bucket_location[param] = (0, param.numel(), cur_bucket_idx)
99                    cur_bucket_size = param.numel()
100                   continue
101
102               # If the parameter cannot fit in the current bucket, create a new bucket
103               if cur_bucket_size + param.numel() > self.bucket_size:
104                   cur_bucket_idx += 1
105                   self.params_to_bucket_location[param] = (0, param.numel(), cur_bucket_idx)
106                   cur_bucket_size = param.numel()
107               else:
108                   self.params_to_bucket_location[param] = (cur_bucket_size, cur_bucket_size + param.numel(), cur_bucke
109                   cur_bucket_size += param.numel()
```

Batch size summary

$$\text{global batch size} = mbs \cdot grad\_acc \cdot dp$$

- mbs: micro batch size
- grad_acc: gradient accumulation steps
- dp: number of parallel instances

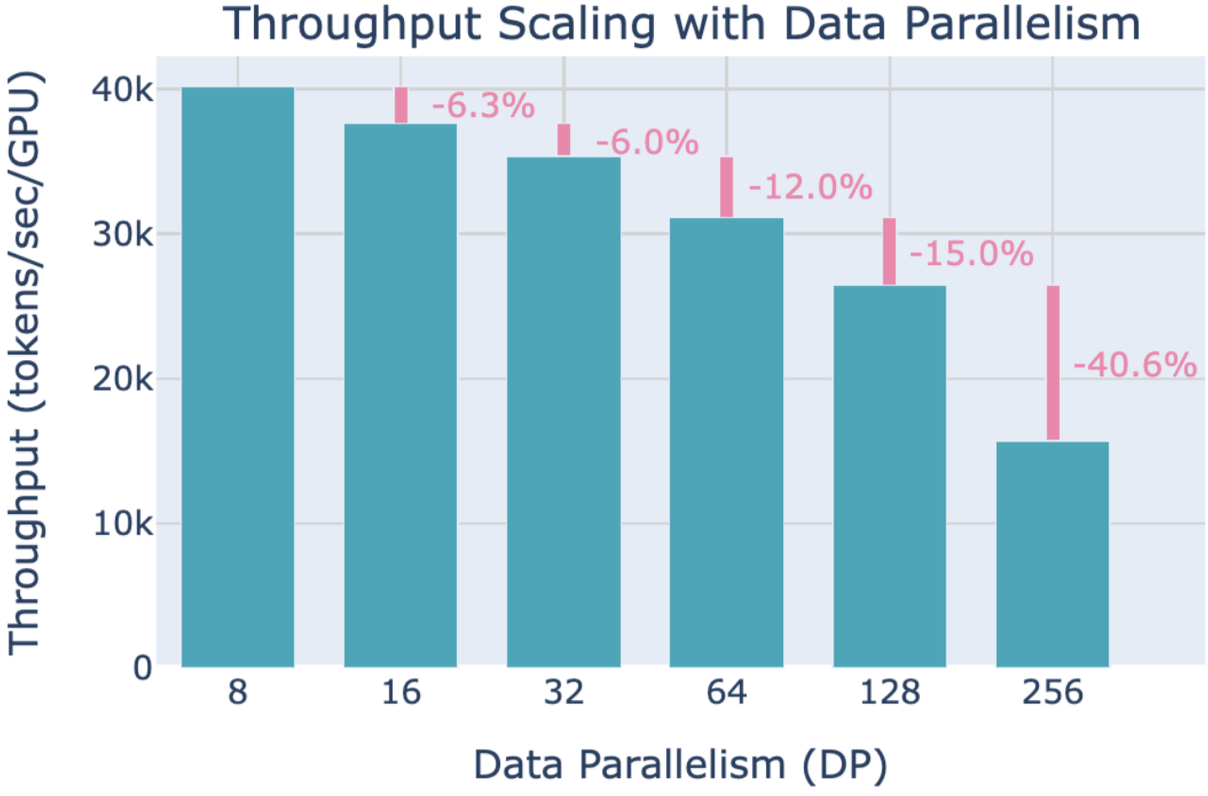
Putting it all together

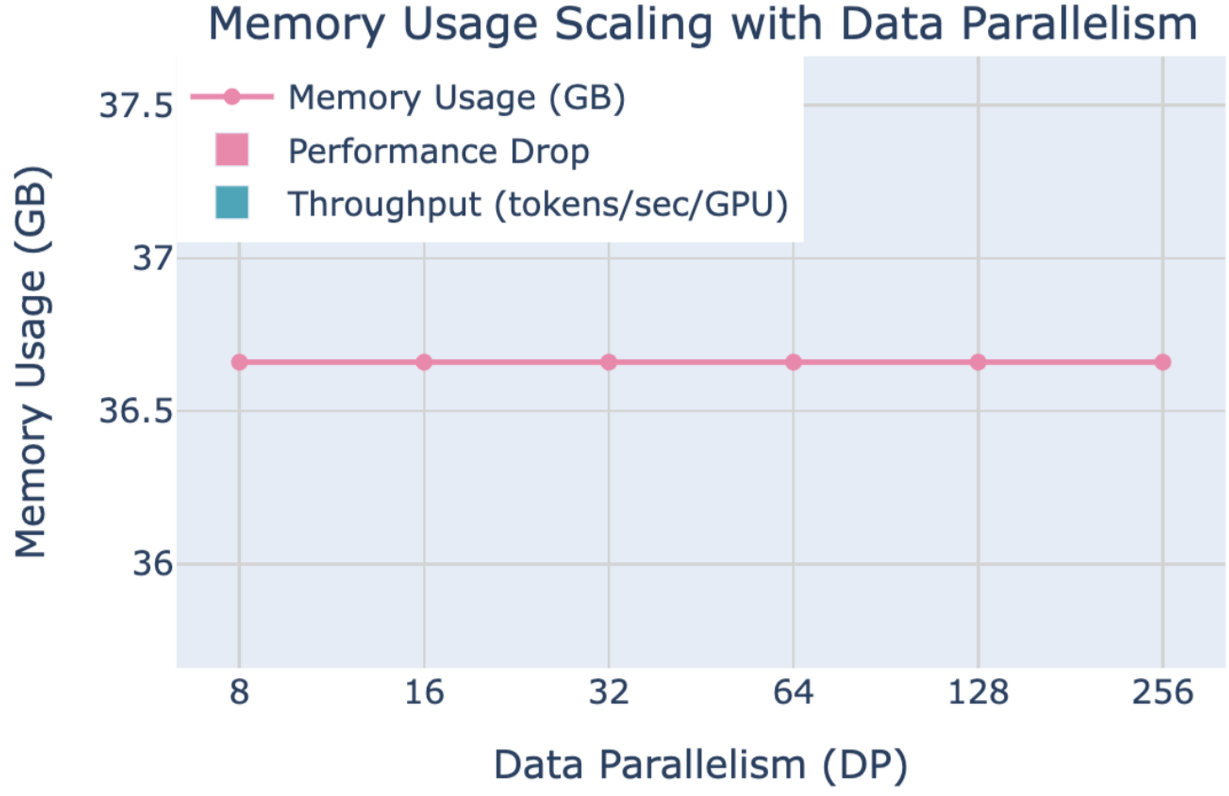
- Global batch size: 4 million tokens
- Sequence length: 4,000 tokens
  - $\Longrightarrow$ batch size: 1024 sequences
- mbs: Suppose 1 GPU fits 2 sequences
- dp: 128 GPUs: 2*128 = 256
- grad_acc of 4: 256*4 = 1024

Quiz: what if we had 512 GPUs?

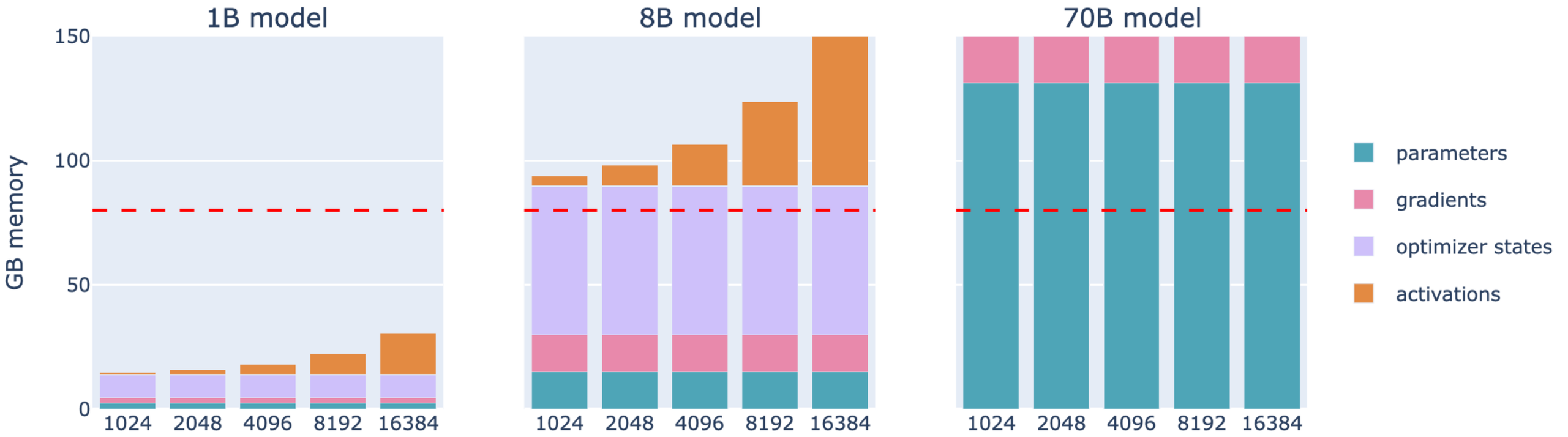
Data Parallelism scaling

- More GPUs means more coordination (e.g., all-reduce, network communication, stragglers)





What if the model is too large?



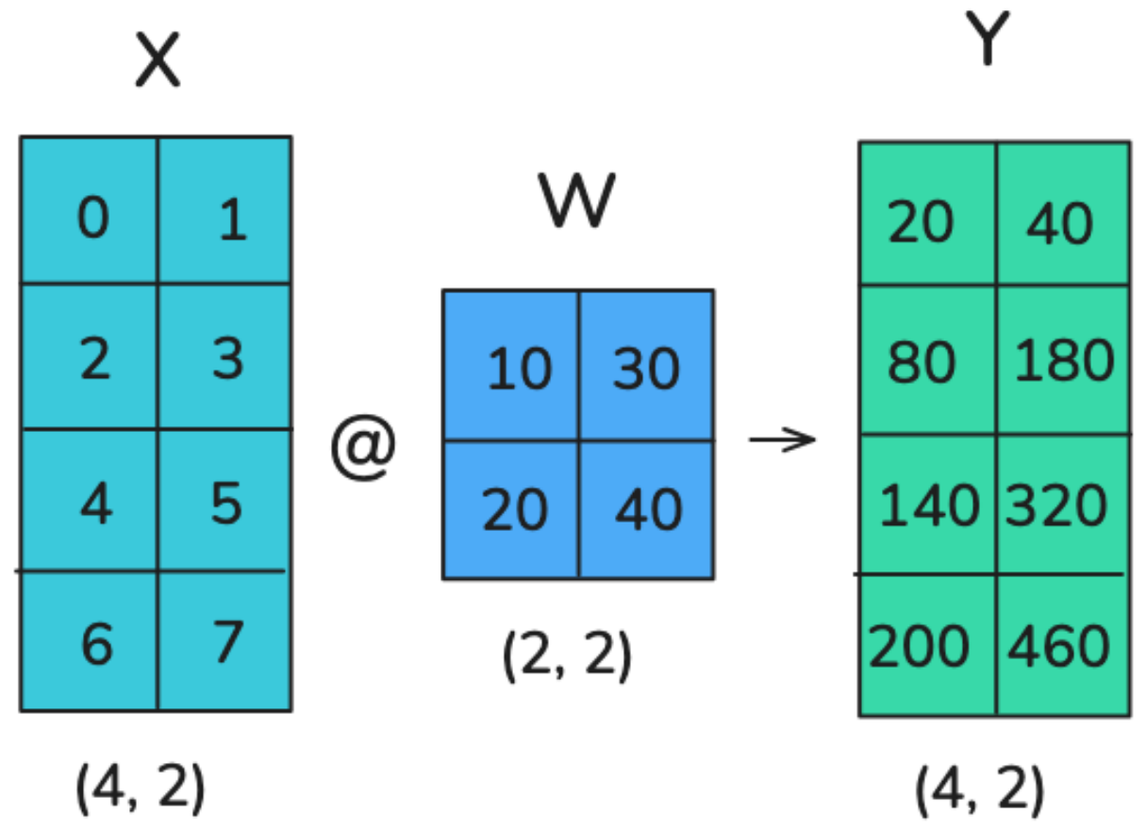
- Split tensors:
  - Parallelism (e.g., tensor, pipeline)
  - Sharding (DeepSpeed ZeRO or PyTorch FSDP)

Parallelism

- Techniques for leveraging computation and memory from multiple GPUs
  - Data parallelism
  - **Tensor parallelism**
  - Pipeline parallelism
  - Memory optimization
  - Choosing parallelism strategies

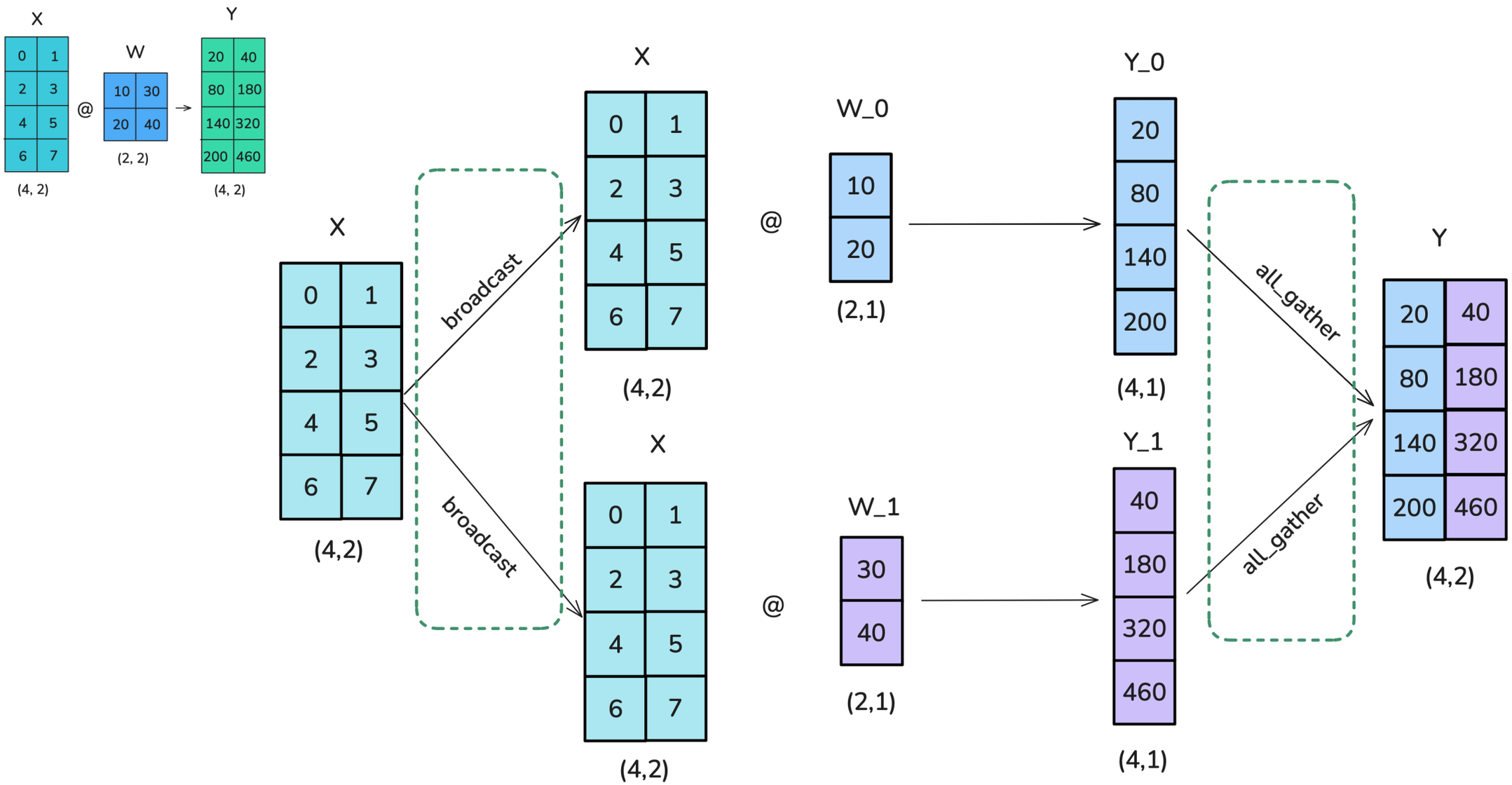
Tensor Parallelism

- Basic idea: take advantage of the structure of matrix multiplication to distribute computation across multiple GPUs.



Column-wise

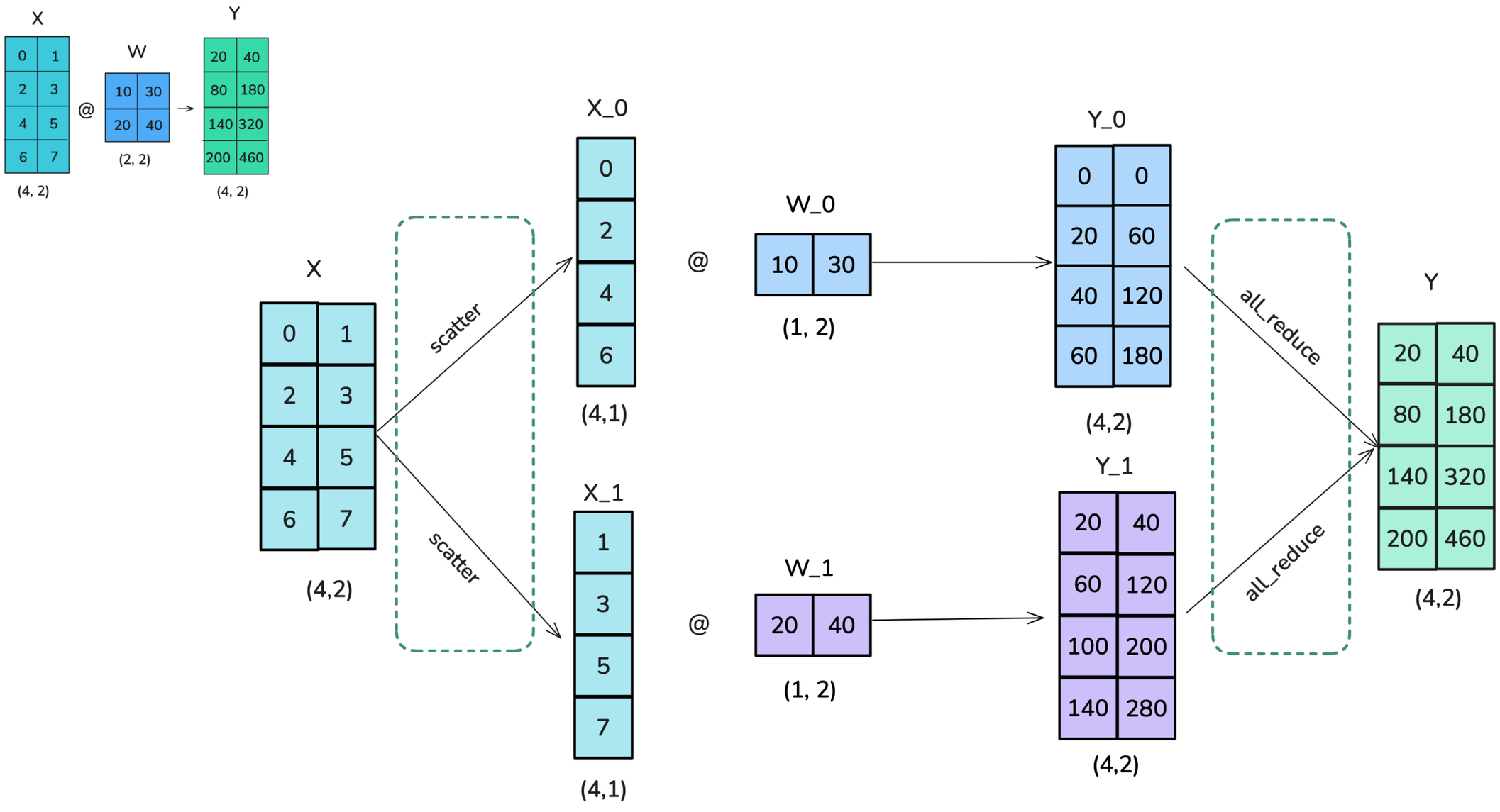
- Split weight matrix into columns, each GPU handles a column chunk



Column linear

Row-wise

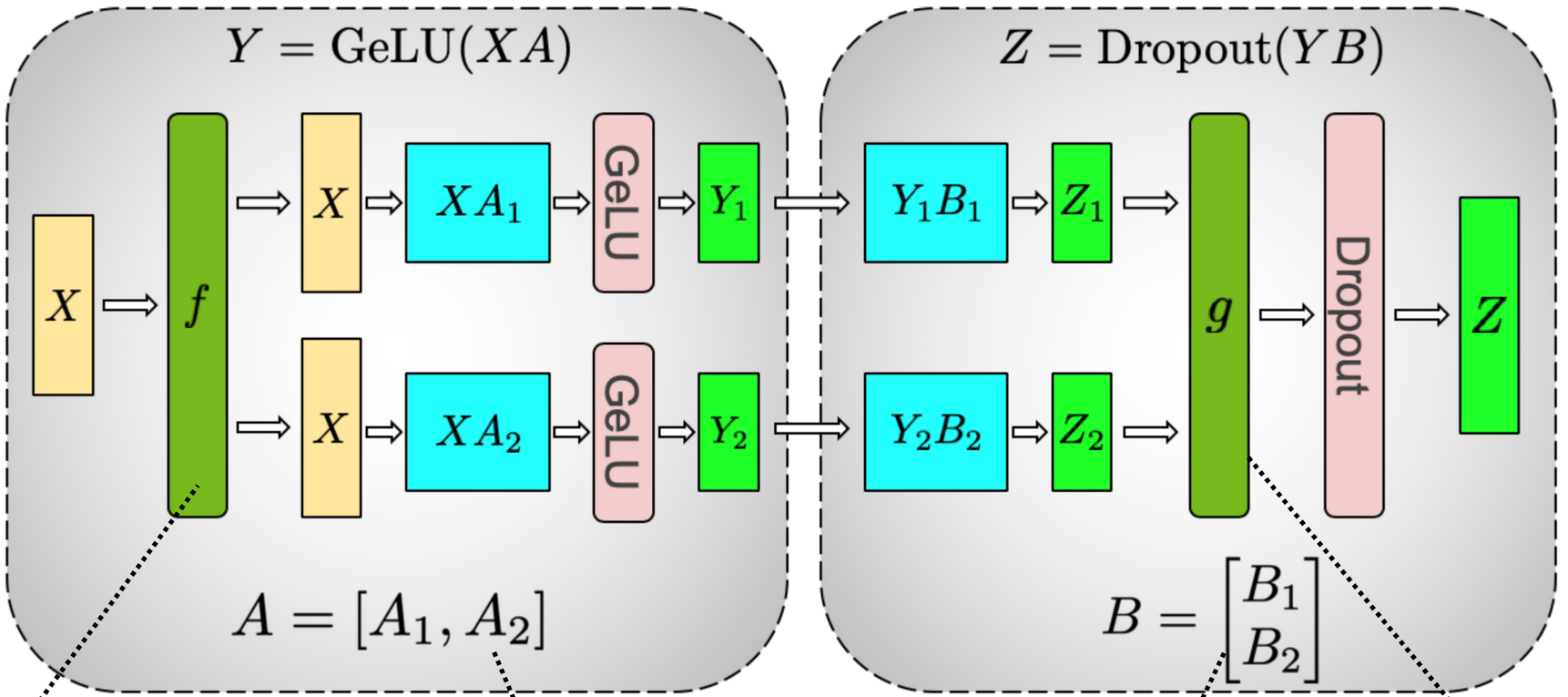
- Split weight matrix into rows (and split inputs into columns), then sum



Row linear

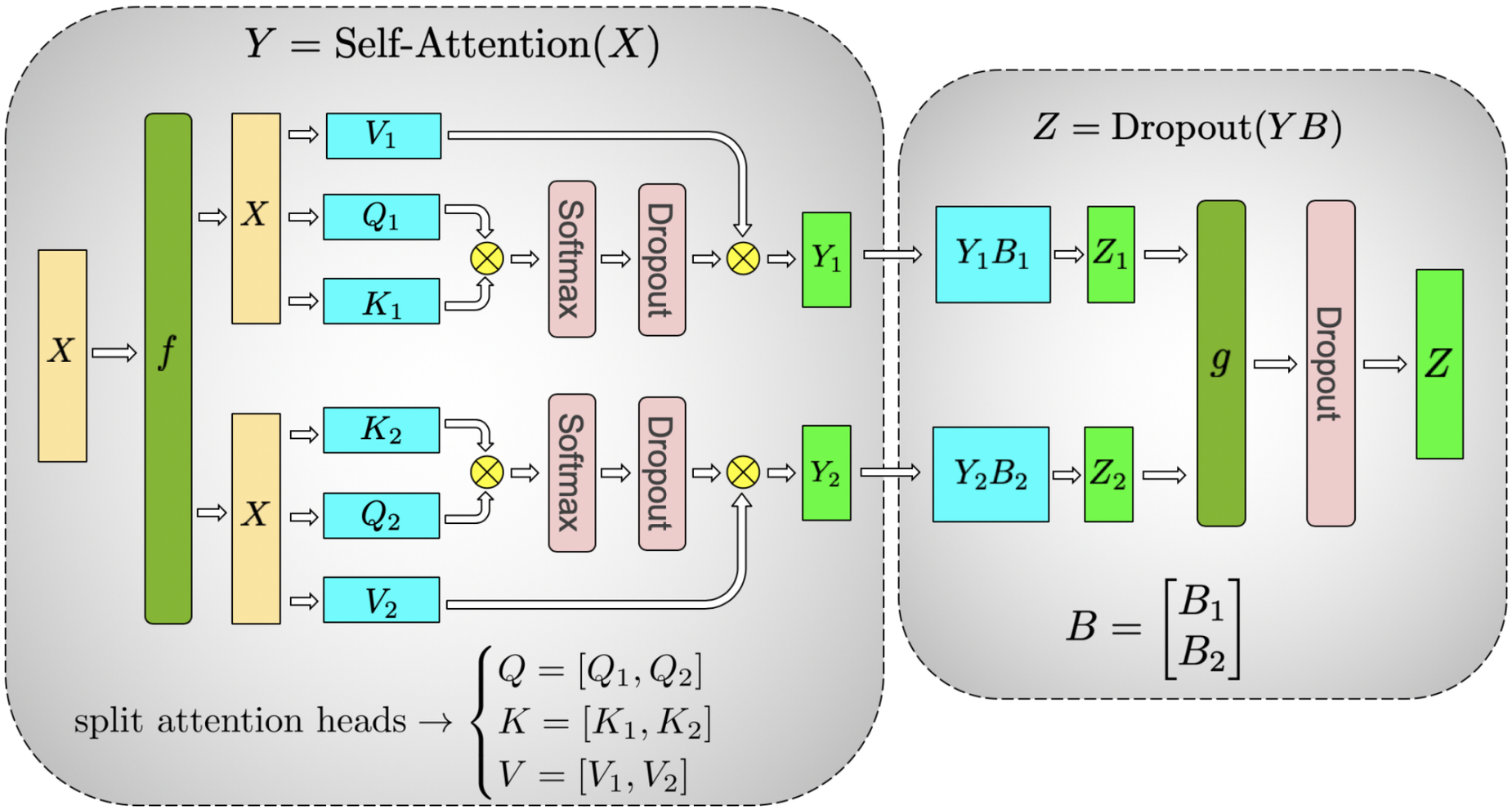
Example: feedforward

- Use column parallel, then row parallel (benefit: no intermediate all-reduce/gather)



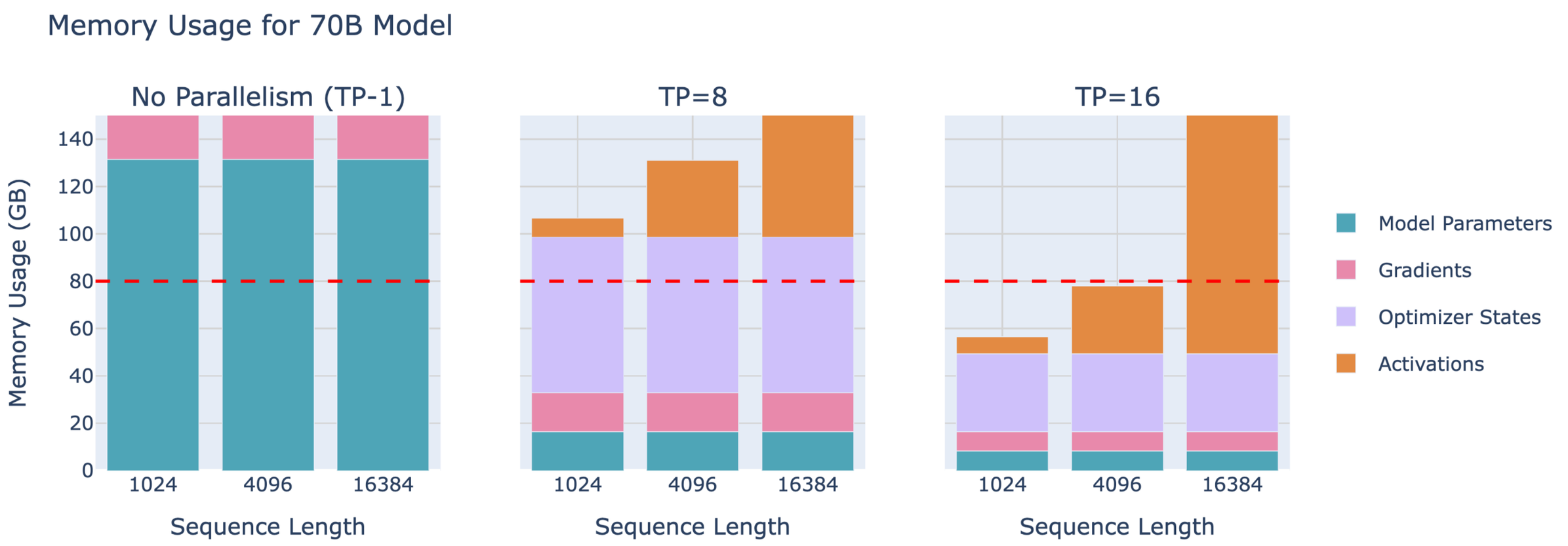
Example: attention

- Each GPU handles a subset of attention heads



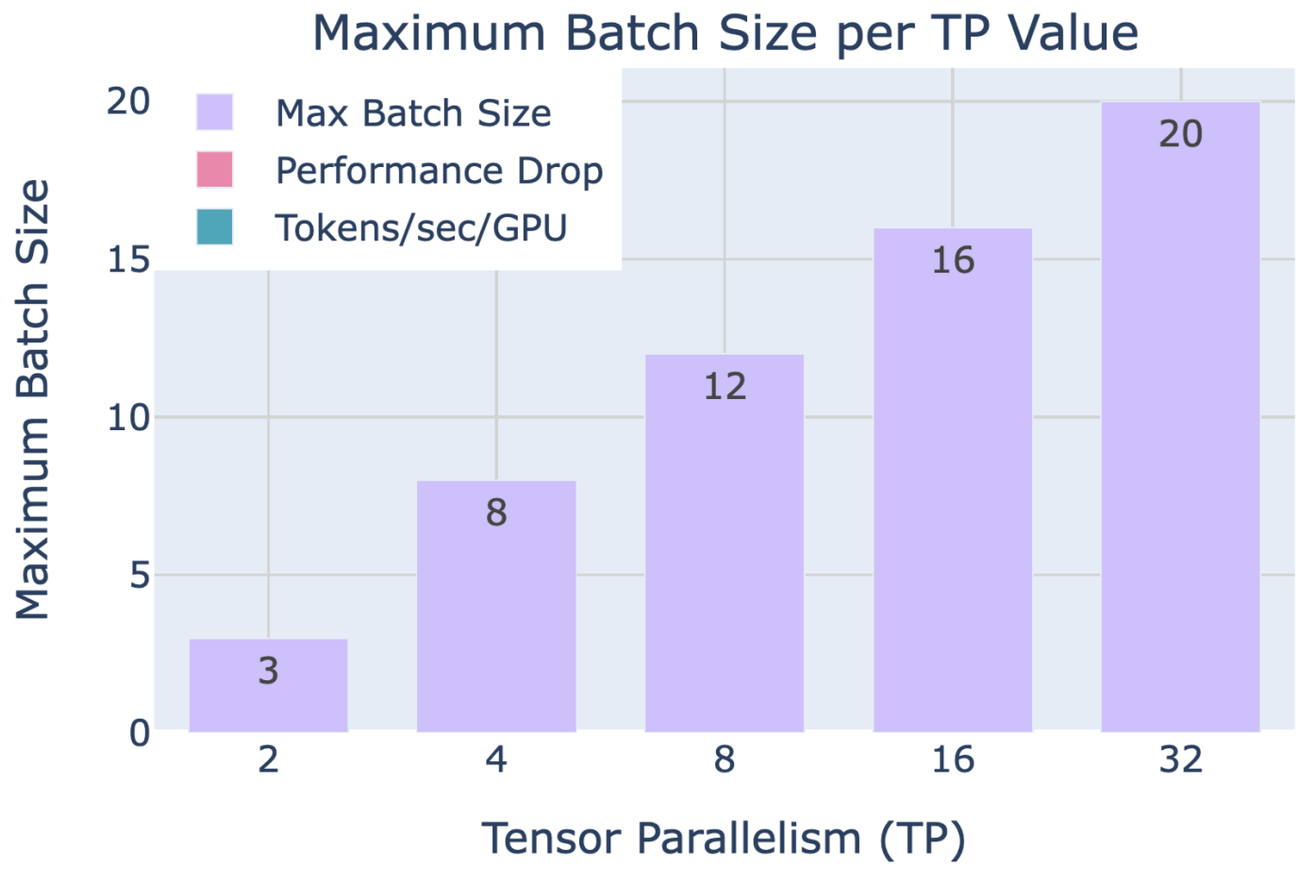
Tensor Parallelism

- Benefit: reduce memory requirements



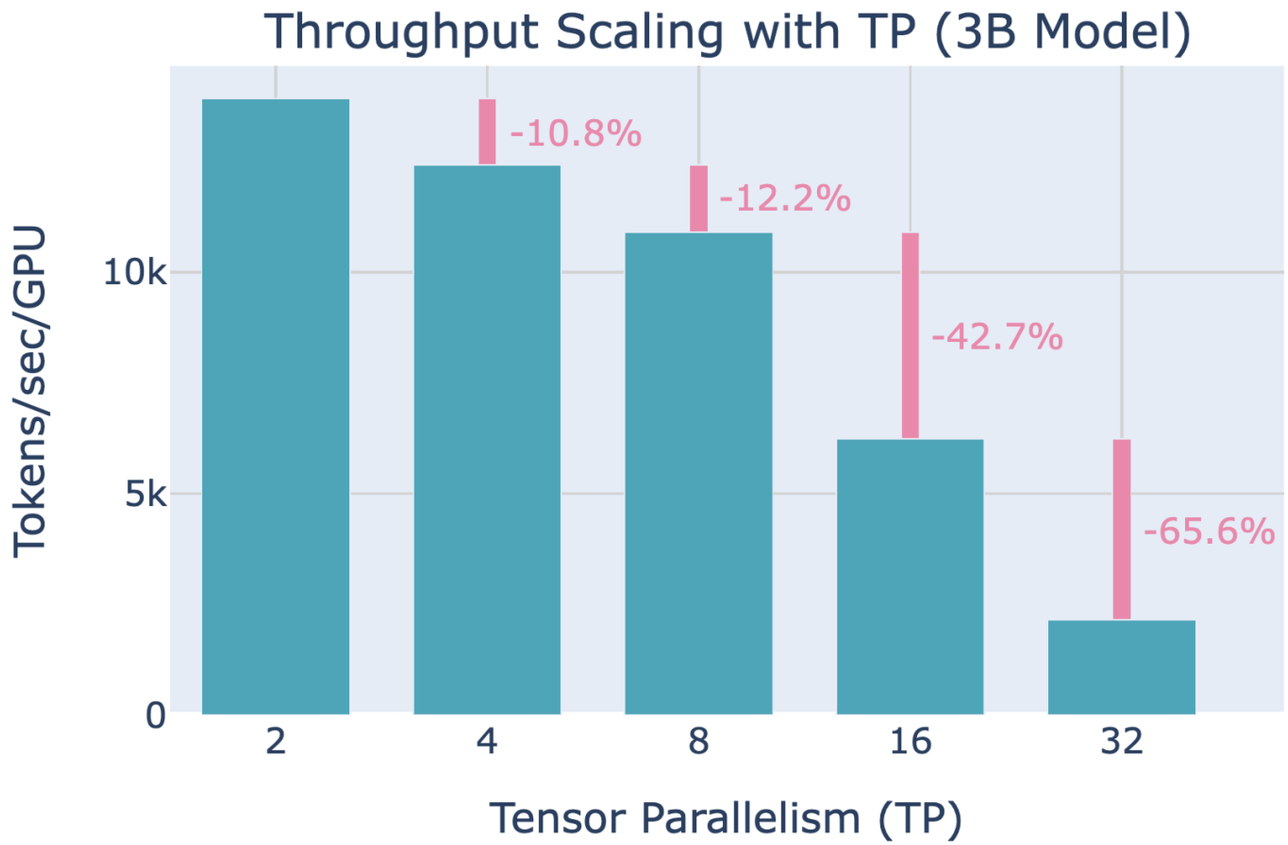
Tensor Parallelism

- Benefit: reduce memory requirements



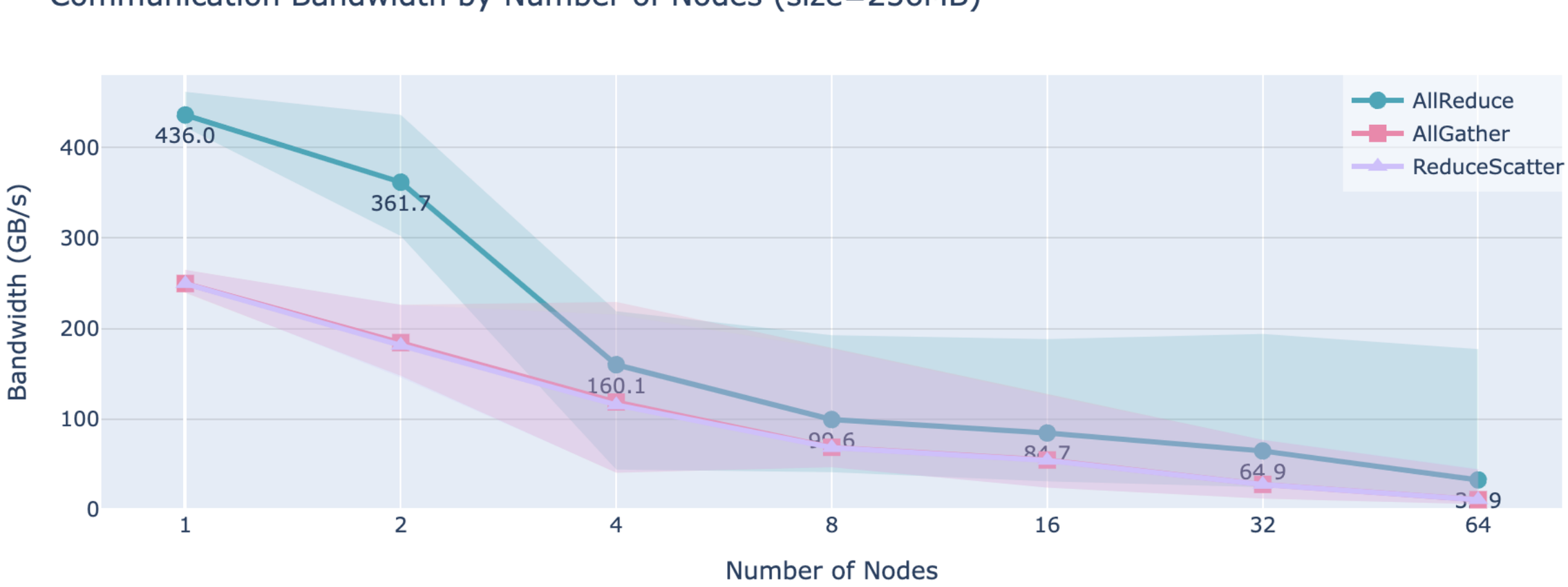
Tensor Parallelism

- Tradeoff: communication costs (e.g., all-reduce)



Tensor Parallelism

- Tradeoff: communication costs (e.g., all-reduce)
  - Cross-node connections particularly slow

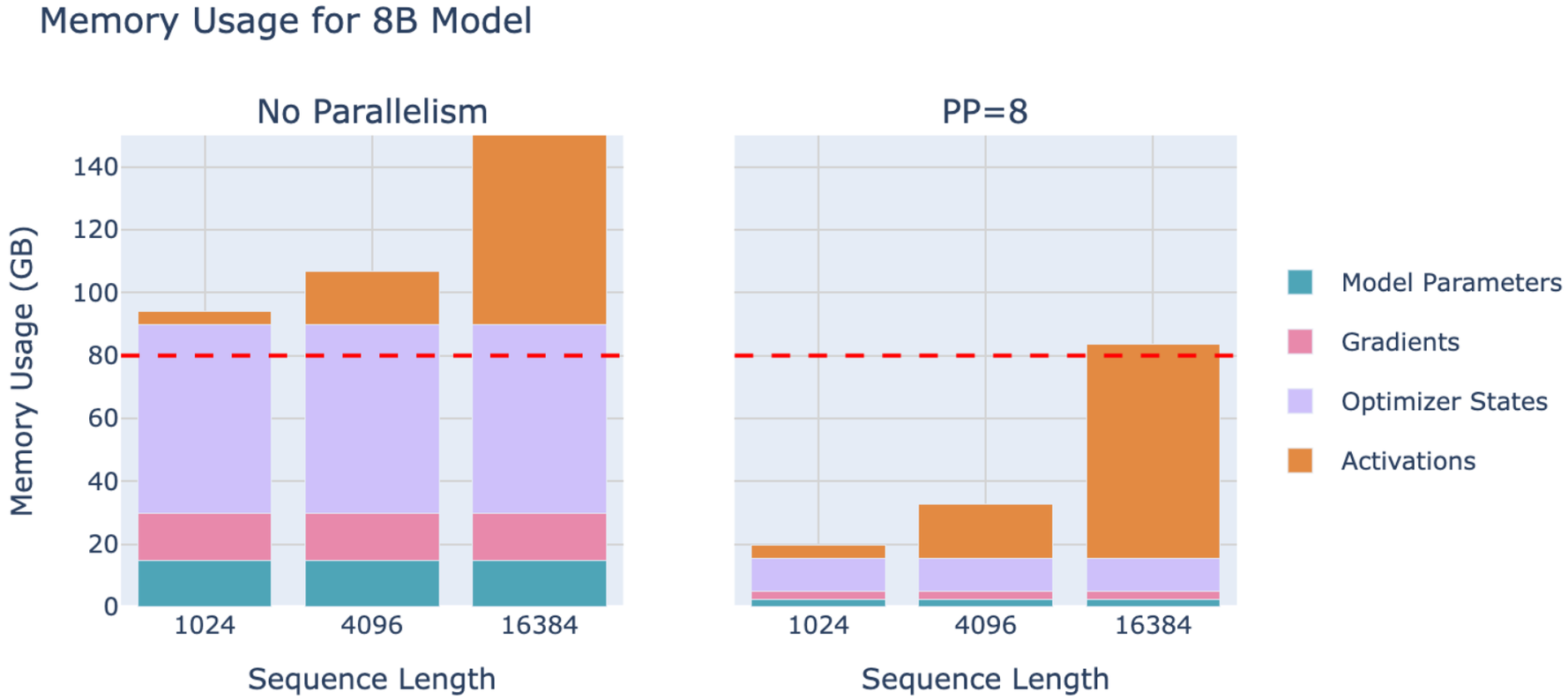


Parallelism

- Techniques for leveraging computation and memory from multiple GPUs
  - Data parallelism
  - Tensor parallelism
  - **Pipeline parallelism**
  - Memory optimization
  - Choosing parallelism strategies

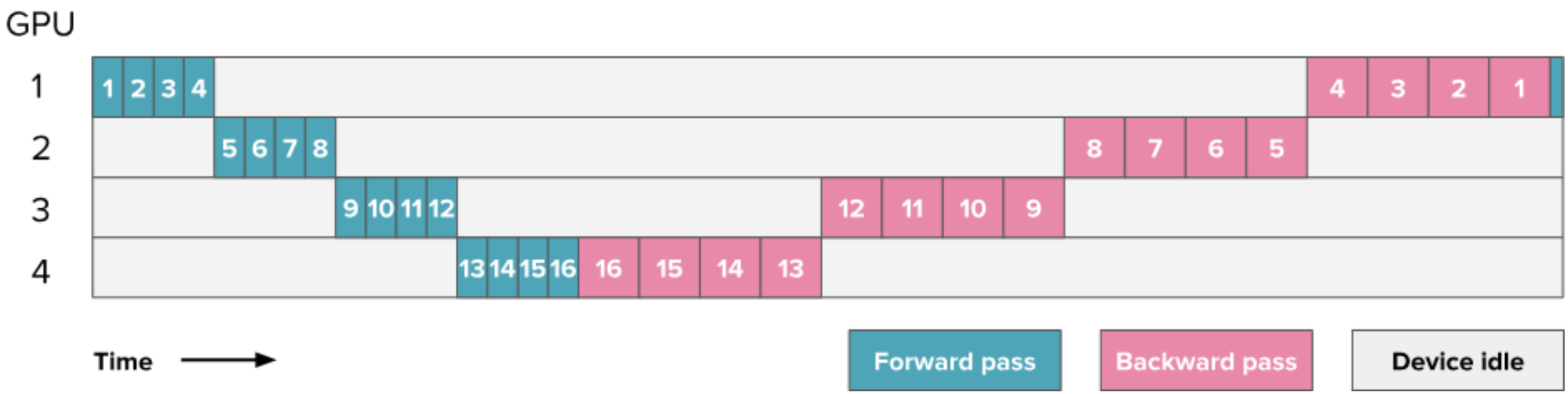
Pipeline Parallelism

- Basic idea: split *layers* across multiple GPUs
  - E.g., layers 1-4 on GPU 1, layers 5-8 on GPU 2



Pipeline Parallelism

- Basic idea: split layers across multiple GPUs

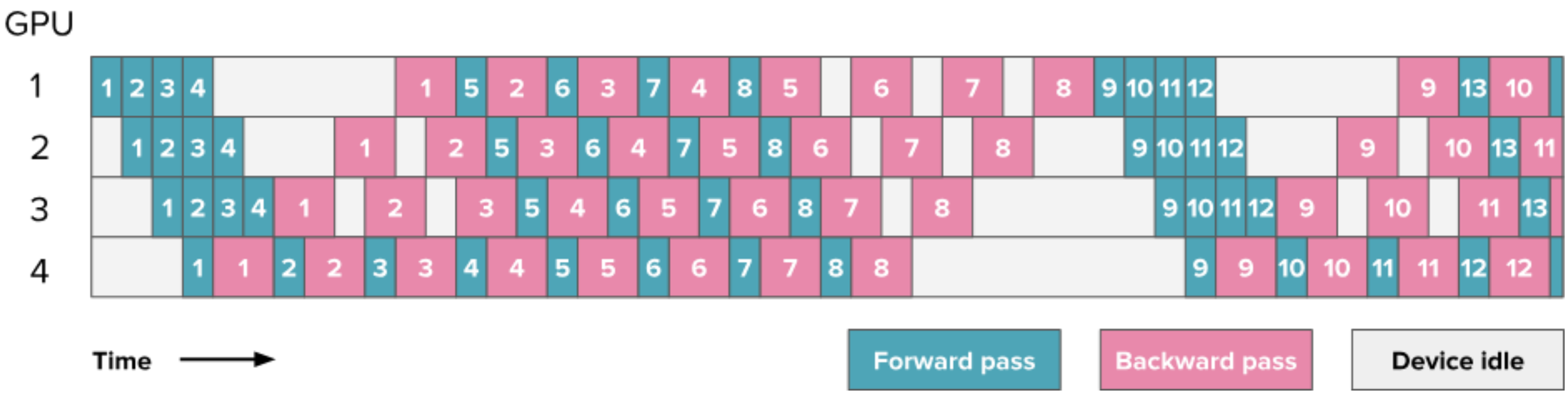


An example of Pipeline parallelism for a model with 16 layers distributed across 4 GPUs. The numbers correspond to the layer IDs.

Key challenge: reducing time lost due to the “bubble” (grey)

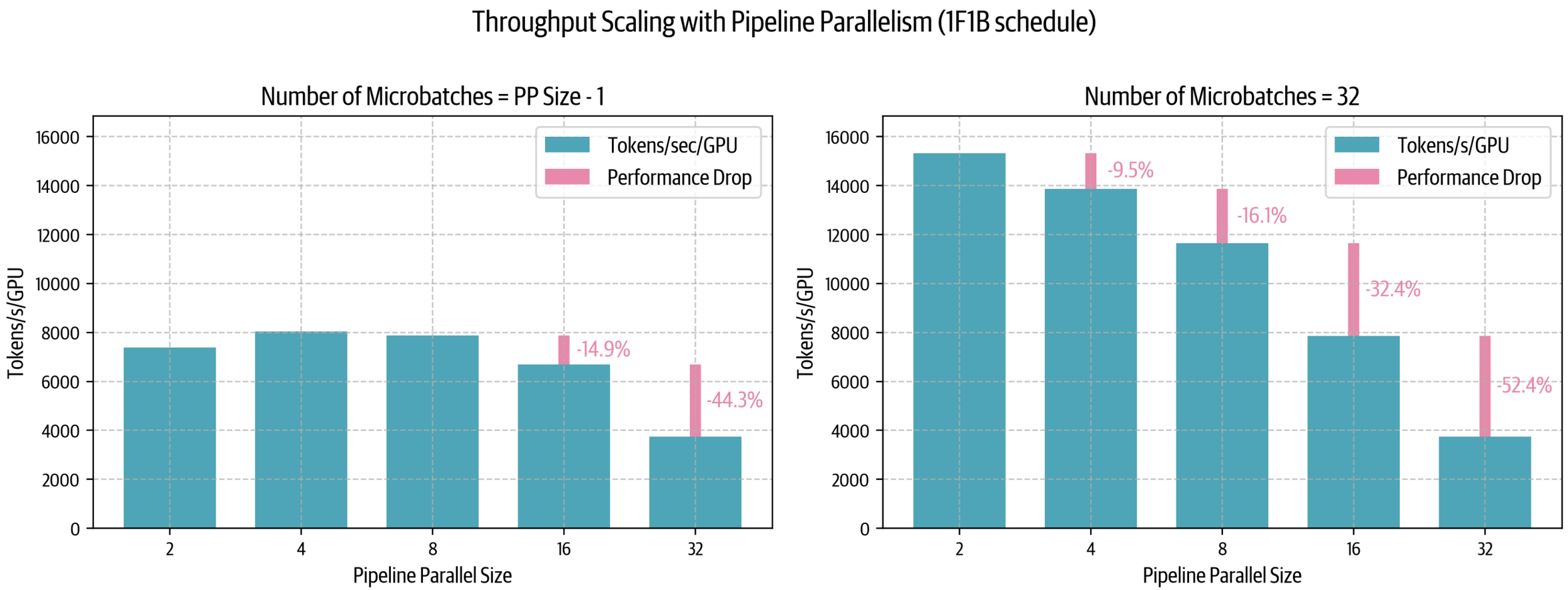
One-forward one-backward

- Start performing backward pass as soon as possible



Numbers: microbatch

One-forward one-backward



Small # of microbatches: inefficient due to bubble

Better scaling with a larger # of microbatches

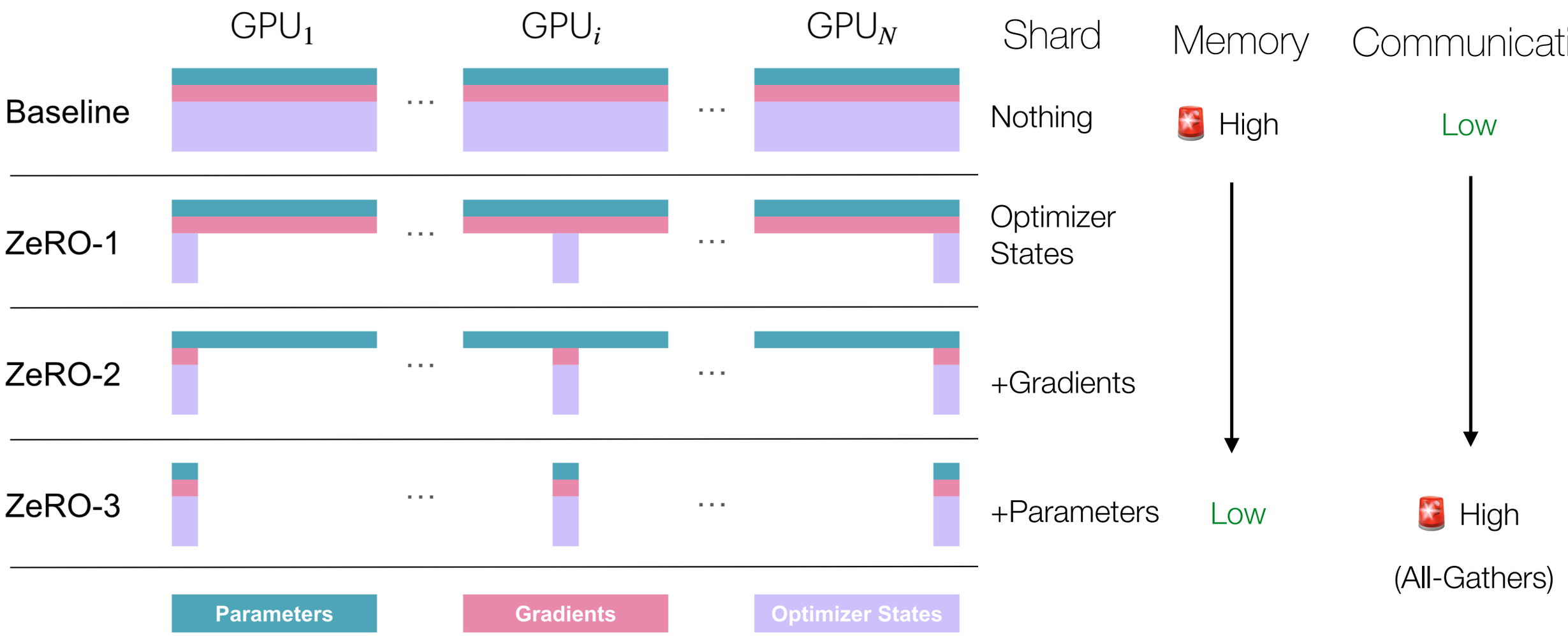
Scaling training

- Parallelism
  - Data parallelism
  - Tensor parallelism
  - Pipeline parallelism
- **Memory optimization**
- Choosing strategies

Memory optimization: ZeRO

- In standard Data Parallelism, each GPU replicates:
  - Model parameters
  - Gradients
  - Optimizer states
- Zero Redundancy Optimizer (ZeRO) partitions these across GPUs

Memory optimization: ZeRO



Memory optimization: ZeRO

- Key idea: load parameters just-in-time. Example:
  - Model: 1B parameters
  - 4 GPUs, each storing 250M parameters
    - At each layer ℓ:
      - GPU uses all-gather to fetch parameters for layer ℓ, computes activations
      - Free fetched parameter memory and continue to next layer
- Different than TP / PP! Only memory sharding, not sharding the computation

Recap of strategies

	Key Idea	Tradeoffs	Use Case
Data Parallelism (DP)	Parallelize on batch dimension	Redundancy. Need to fit model on GPU.	Standard models that fit in GPU memory
Tensor Parallelism (TP)	Parallelize on hidden dimension	Fine-grained => high communication costs.	Large layers (e.g. MLP). Parallelize within a node.
Pipeline Parallelism (PP)	Parallelize on model dimension	Pipeline bubbles	Large deep models. Parallelize across nodes.
ZeRO	Sharding model, optimizer, gradients in DP	High communication costs (all-gather)	Big models that don't fit in GPU memory

Often combined for efficient training (next)!

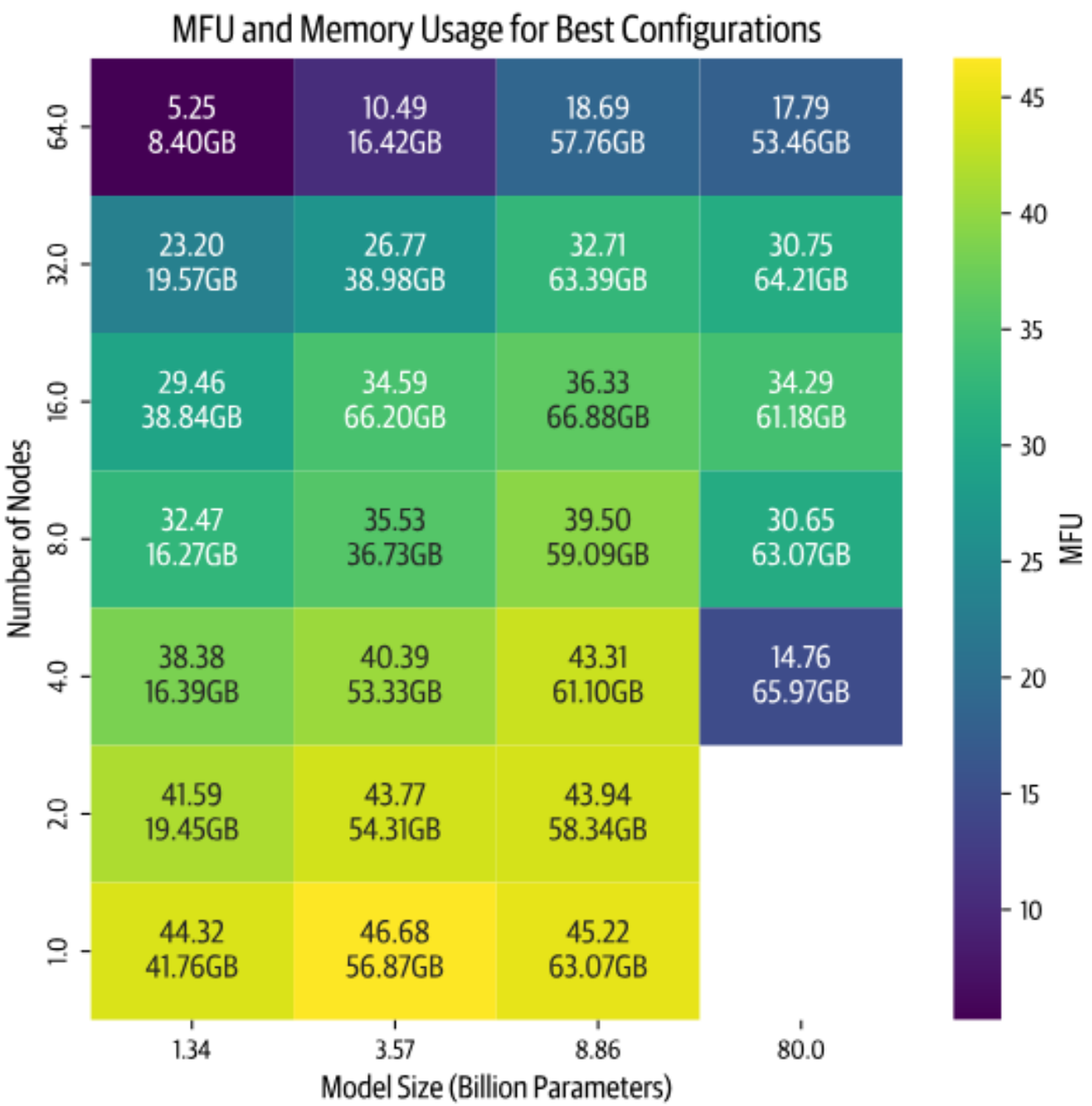
Scaling training

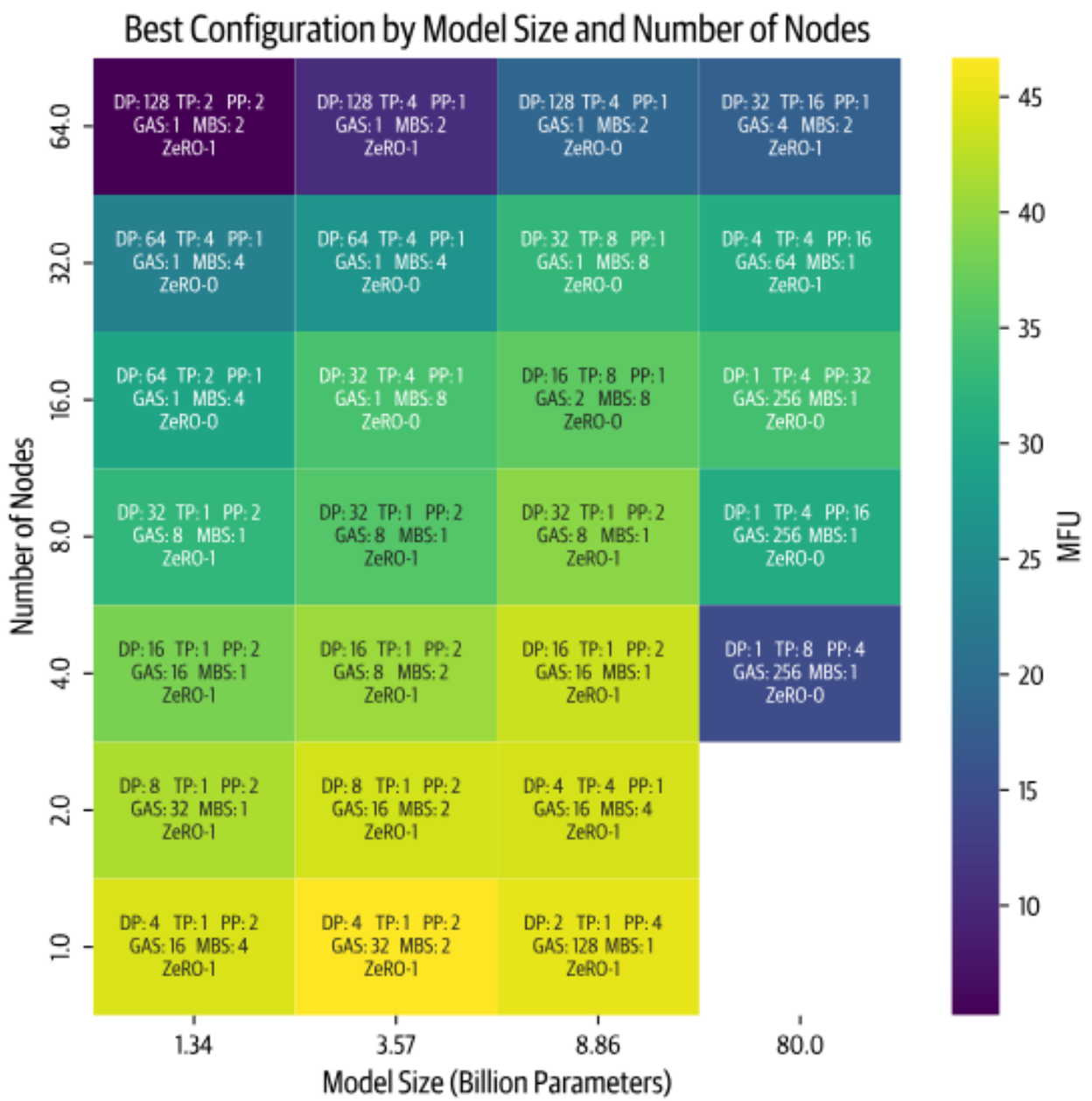
- Parallelism
  - Data parallelism
  - Tensor parallelism
  - Pipeline parallelism
- Memory optimization
- **Choosing strategies**

Choosing strategies

- Fit model into memory
- Satisfy target global batch size
- Optimize training throughput

Best configuration experiment (from book)





GBS 1M tokens, sequence length 4096, 1-64 8xH100 nodes

Scaling training

- Parallelism
  - Data parallelism
  - Tensor parallelism
  - Pipeline parallelism
- Memory optimization
- Choosing strategies

Example: torchtitan

torchtitan

A PyTorch native platform for training generative AI models

8 GPU Featur

BSD 3-Clause

Key features available

1. Multi-dimensional composable parallelisms
    - FSDP2 with per-parameter sharding
    - Tensor Parallel (including async TP)
    - Pipeline Parallel
    - Context Parallel
2. Meta device initialization
3. Selective (layer or operator) and full activation checkpointing
4. Distributed checkpointing (including async checkpointing)
    - Interoperable checkpoints which can be loaded directly into `torchtune` for fine-tuning
5. `torch.compile` support
6. Float8 support (how-to)
7. DDP and HSDP
8. TorchFT integration
9. Checkpointable data-loading, with the C4 dataset pre-configured (144M entries) and support for custom datasets
10. Gradient accumulation, enabled by giving an additional `--training.global_batch_size` argument in configuration
11. Flexible learning rate scheduler (warmup-stable-decay)
12. Loss, GPU memory, throughput (tokens/sec), TFLOPs, and MFU displayed and logged via Tensorboard or Weights & Biases

Example: Megatron-LM

Megatron-LM & Megatron Core

GPU-optimized library for training transformer models at scale

```
Megatron-LM/
├── megatron/
│   ├── core/                    # Megatron Core (kernels, parallelism, building blocks)
│   │   ├── models/              # Transformer models
│   │   ├── transformer/         # Transformer building blocks
│   │   ├── tensor_parallel/     # Tensor parallelism
│   │   ├── pipeline_parallel/   # Pipeline parallelism
│   │   ├── distributed/         # Distributed training (FSDP, DDP)
│   │   ├── optimizer/           # Optimizers
│   │   ├── datasets/            # Dataset loaders
│   │   ├── inference/           # Inference engines
│   │   └── export/              # Model export (e.g. TensorRT-LLM)
│   ├── training/                # Training scripts
│   ├── inference/               # Inference server
│   ├── legacy/                  # Legacy components
│   └── post_training/           # Post-training (RLHF, etc.)
├── examples/                    # Ready-to-use training examples
├── tools/                       # Utility tools
├── tests/                       # Comprehensive test suite
└── docs/                        # Documentation
```

Thank you