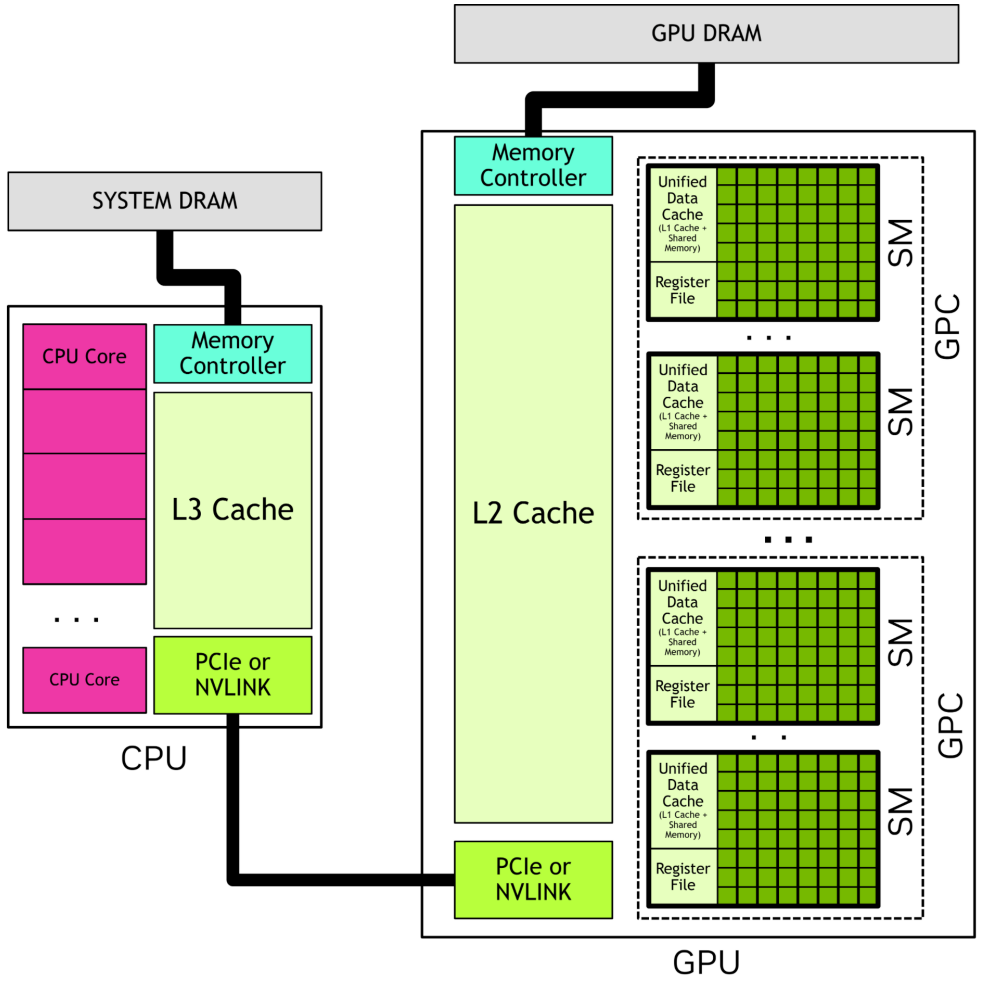
Training Systems

CSCI 5942

Lecture 7

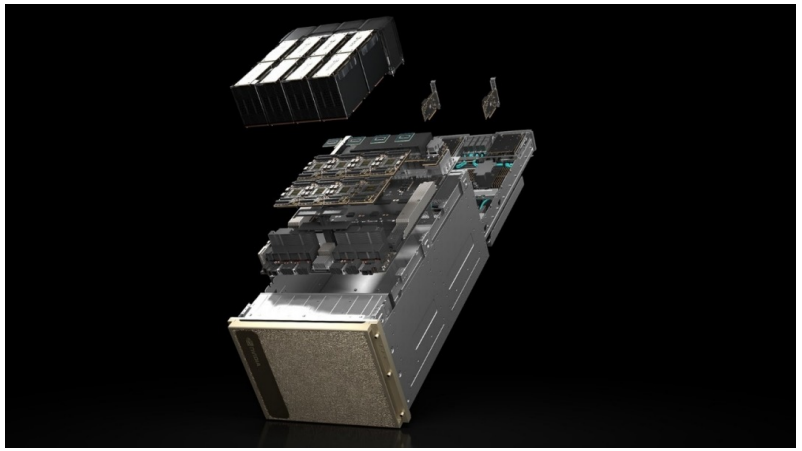
College of Engineering & Applied Science
UNIVERSITY OF COLORADO **BOULDER**

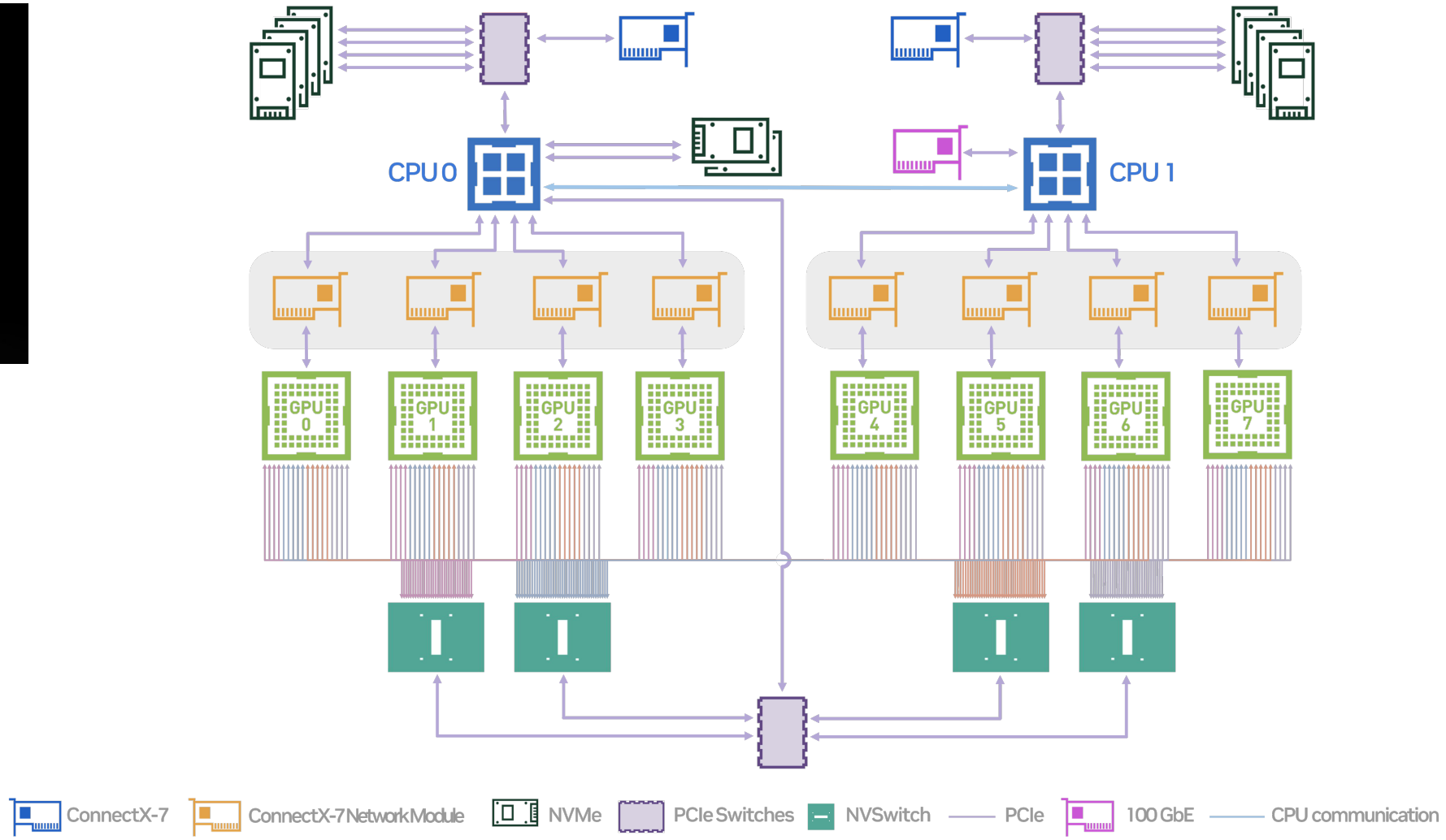
Recap: The GPU



https://docs.nvidia.com/cuda/cuda-programming-guide/01-introduction/programming-model.html

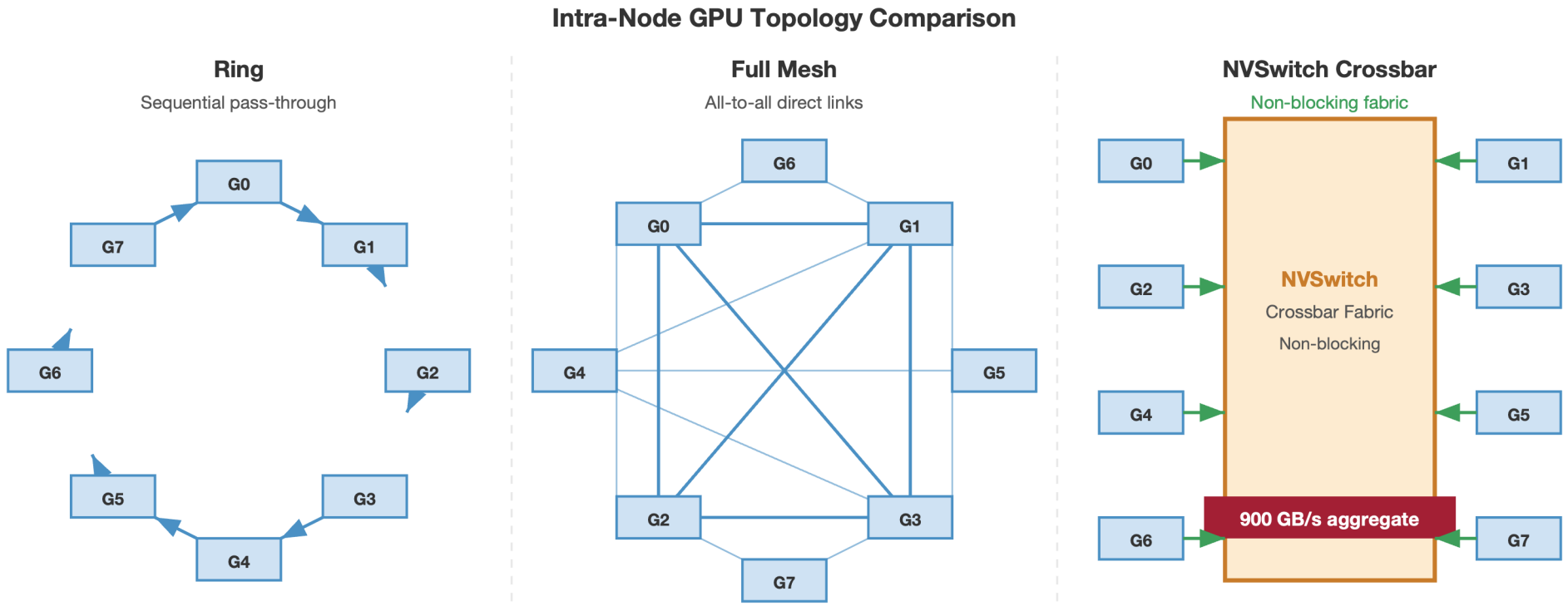
The GPU Node





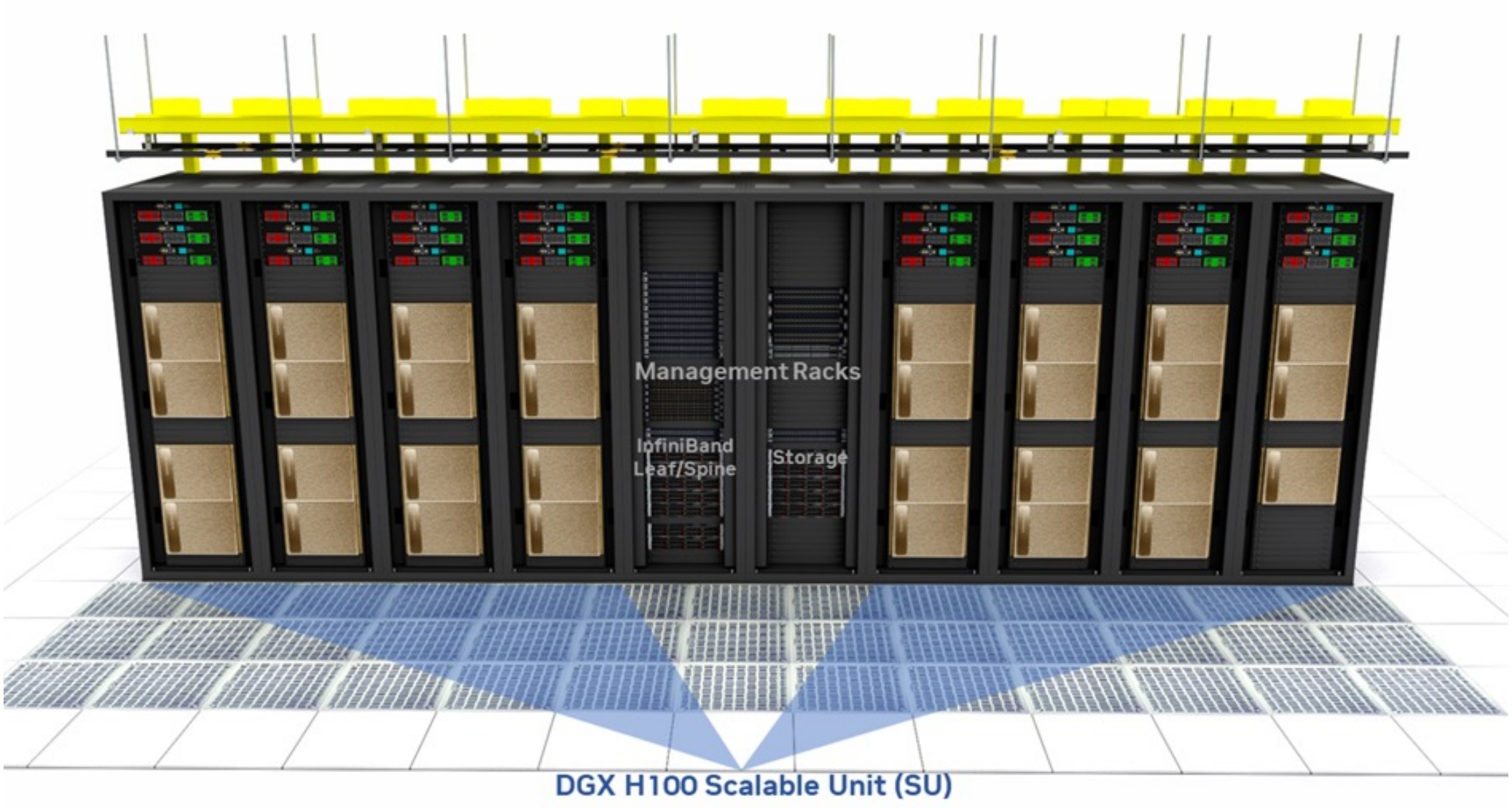
https://docs.nvidia.com/dgx/dgxh100-user-guide/introduction-to-dgxh100.html

Intra-node / “Scale-Up” Interconnect

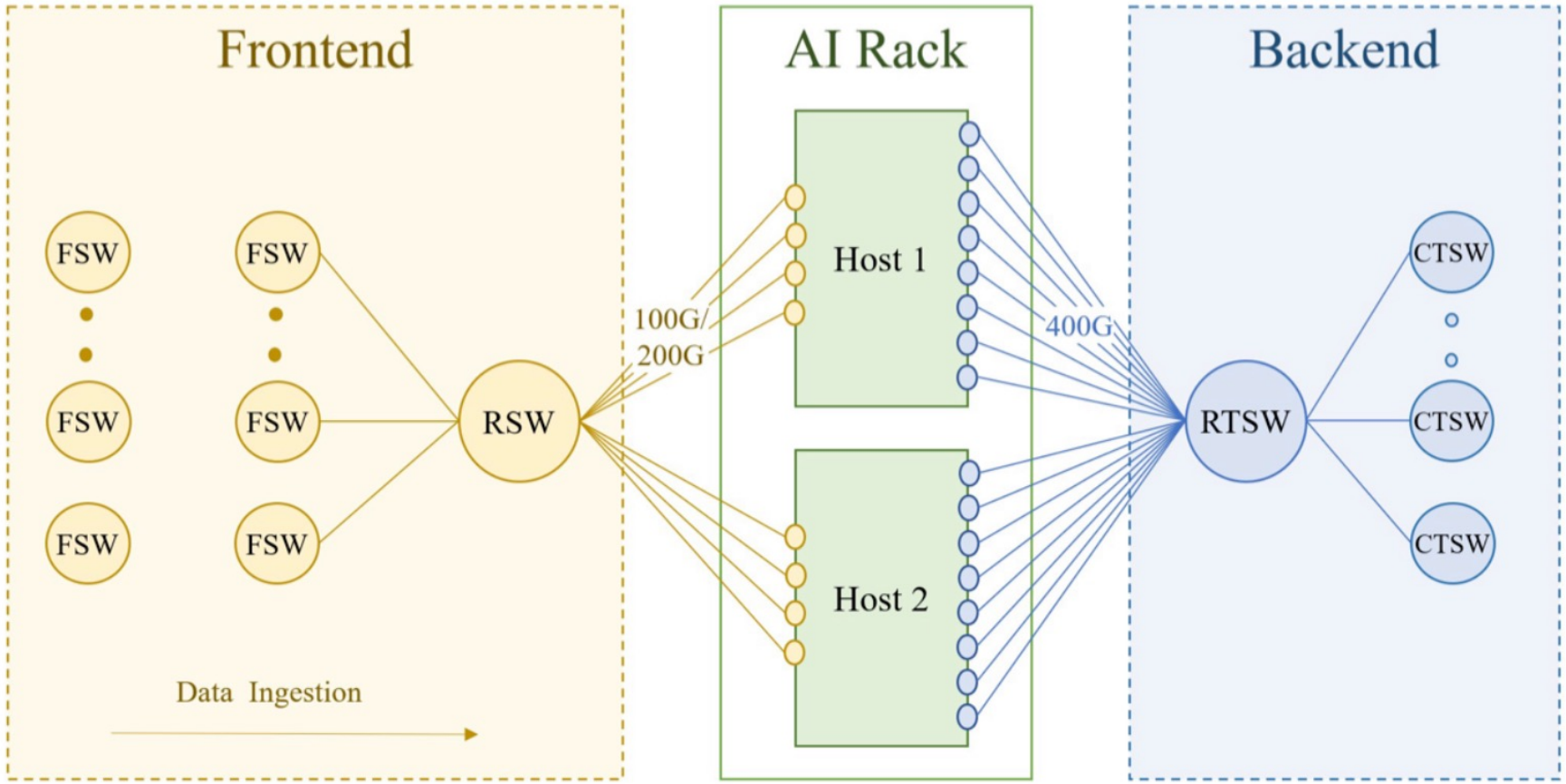


https://mlsysbook.ai/vol2/compute_infrastructure/compute_infrastructure.html#sec-compute-node

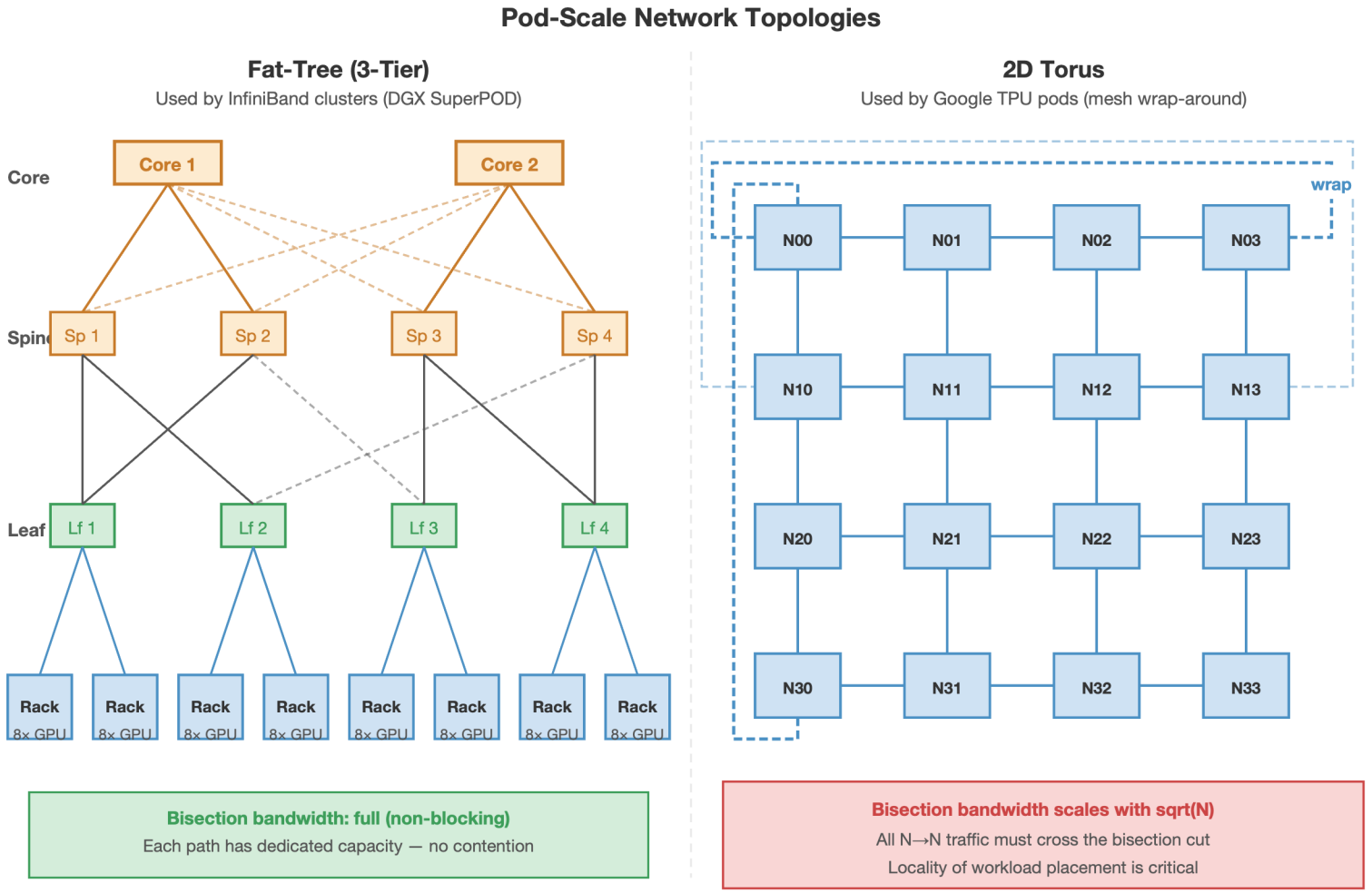
The Rack / “Pod”


Management Racks
InfiniBand
Leaf/Spine
Storage
DGX H100 Scalable Unit (SU)

Inter-node / “Scale-out” Network



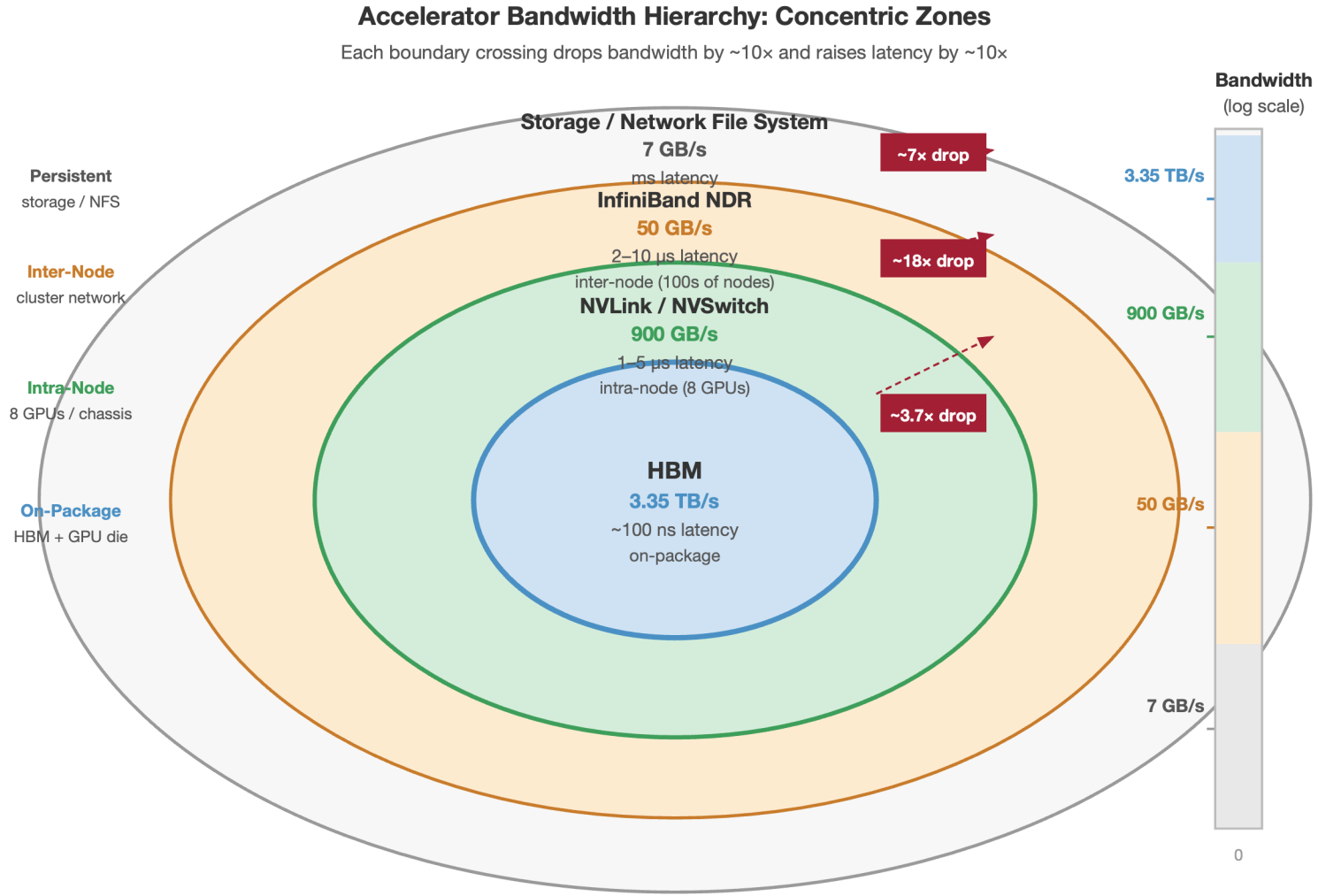
Inter-node / “Scale-out” Network



Fat-tree provides full bisection bandwidth for any traffic pattern; torus is cheaper but rewards workloads with strong locality.

https://mlsysbook.ai/vol2/compute_infrastructure/compute_infrastructure.html#sec-compute-rack

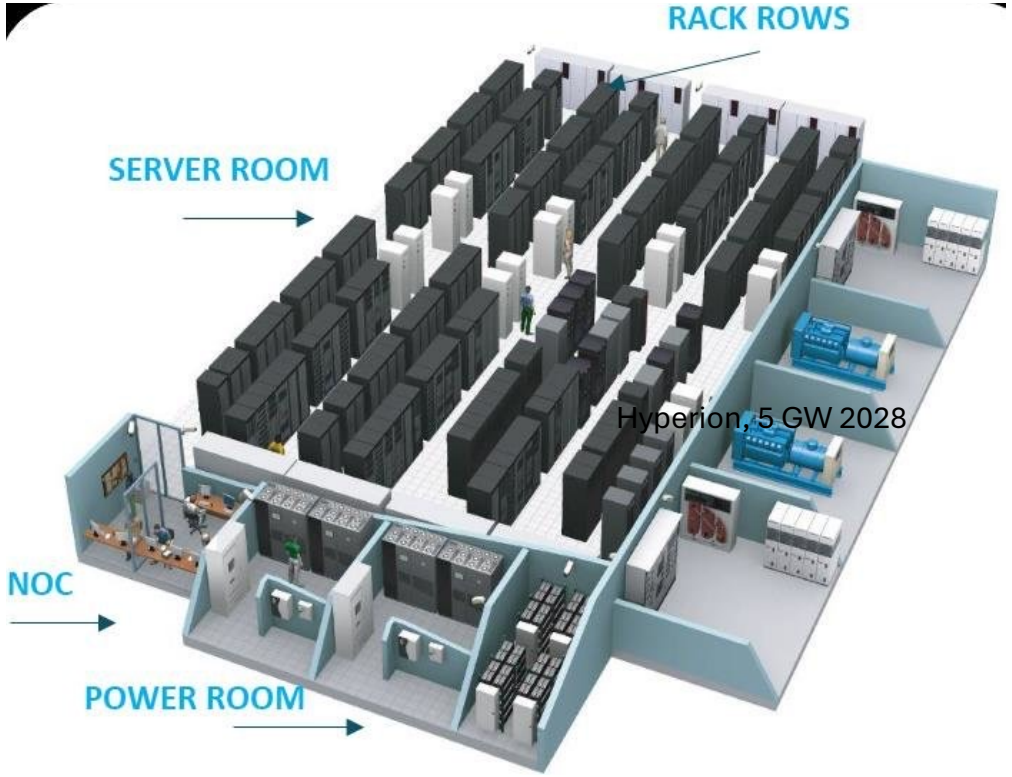
The Bandwidth Hierarchy



Parallelism strategy must respect zone boundaries: tensor parallelism stays intra-node (NVLink), pipeline parallelism spans nodes (InfiniBand).

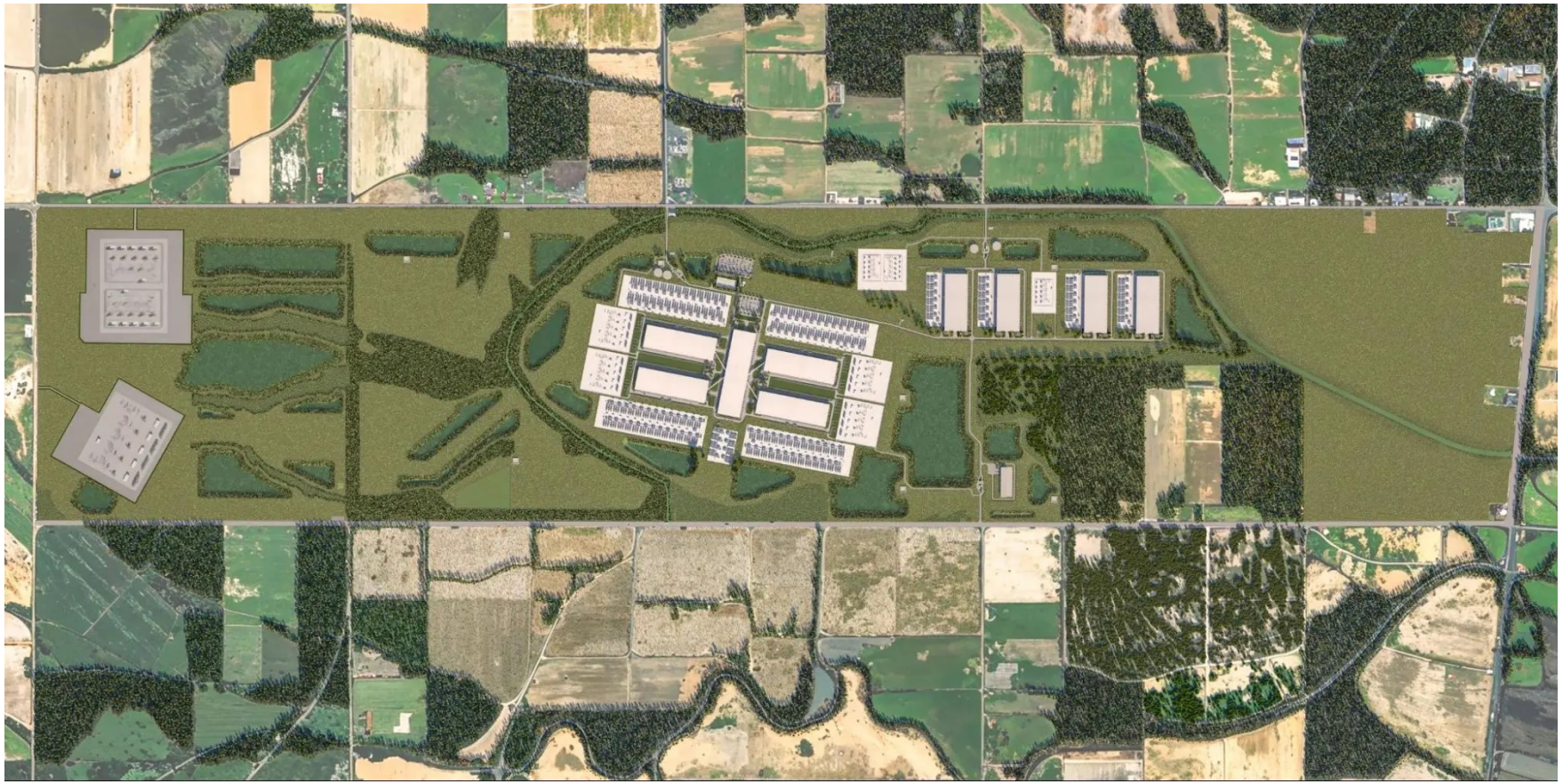
https://mlsysbook.ai/vol2/compute_infrastructure/compute_infrastructure.html#sec-compute-bandwidth-hierarchy

Datacenters and Regions

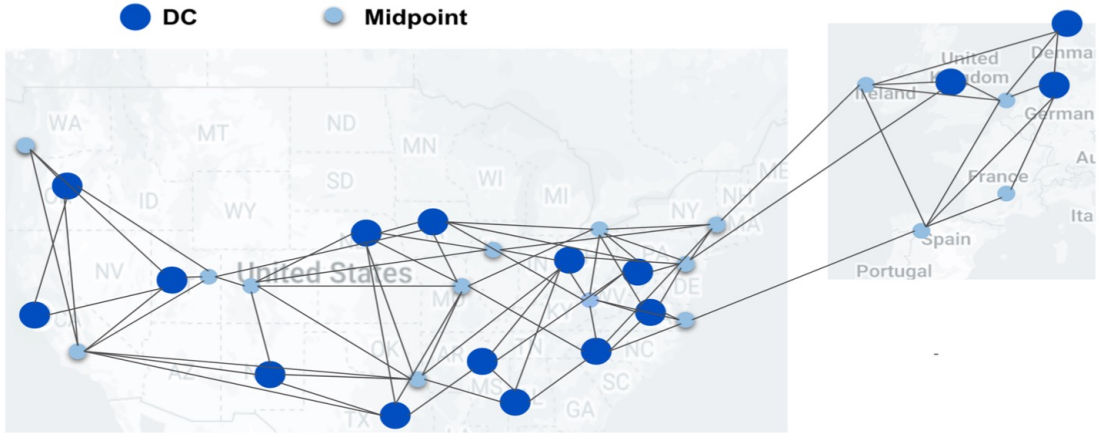


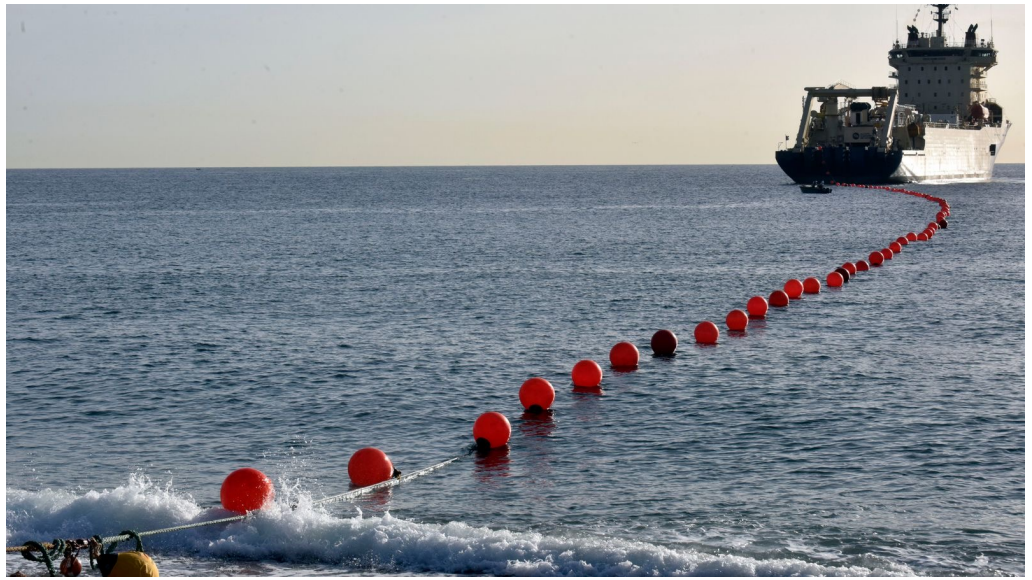
Hyperion, 5 GW 2028

A region-wide training cluster



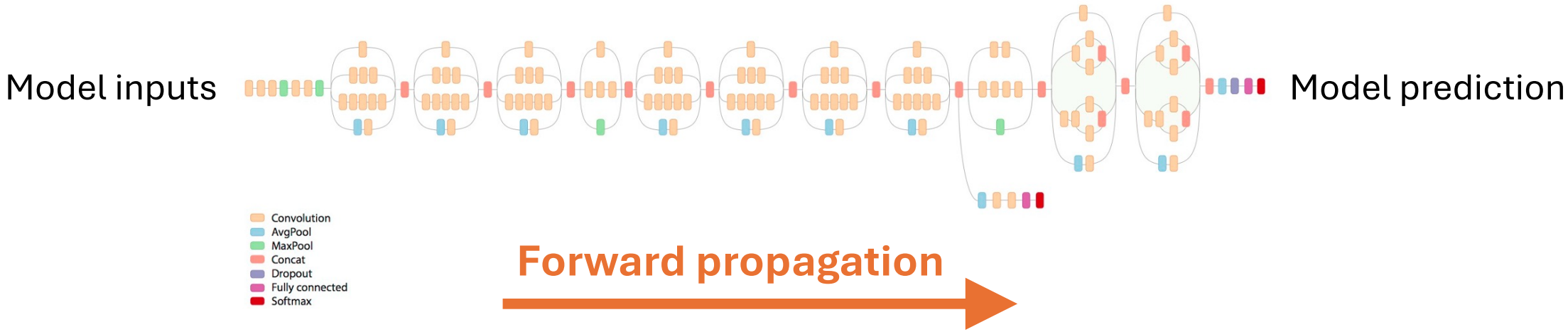
Global Regions





Recap: DNN Training

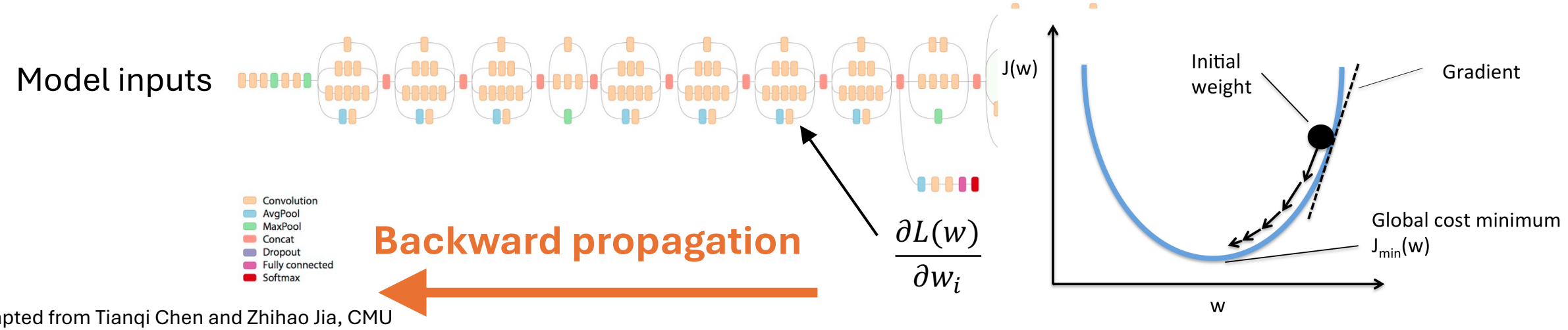
- Forward propagation: Apply model to a batch of input samples and run calculation through operators to produce a prediction



Slides adapted from Tianqi Chen and Zhihao Jia, CMU

Recap: DNN Training

- Forward propagation: Apply model to a batch of input samples and run calculation through operators to produce a prediction
- Backwards propagation: Run the model in reverse to produce a gradient for each trainable weight



Slides adapted from Tianqi Chen and Zhihao Jia, CMU

Recap: DNN Training

- Forward propagation: Apply model to a batch of input samples and run calculation through operators to produce a prediction
- Backwards propagation: Run the model in reverse to produce a gradient for each trainable weight
- Weight update: Use the gradients to update model weights

$$w_i := w_i - \gamma \frac{\partial L(w)}{\partial w_i} = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \frac{\partial l_i(w)}{\partial w_i}$$

Gradients of individual samples

Slides adapted from Tianqi Chen and Zhihao Jia, CMU

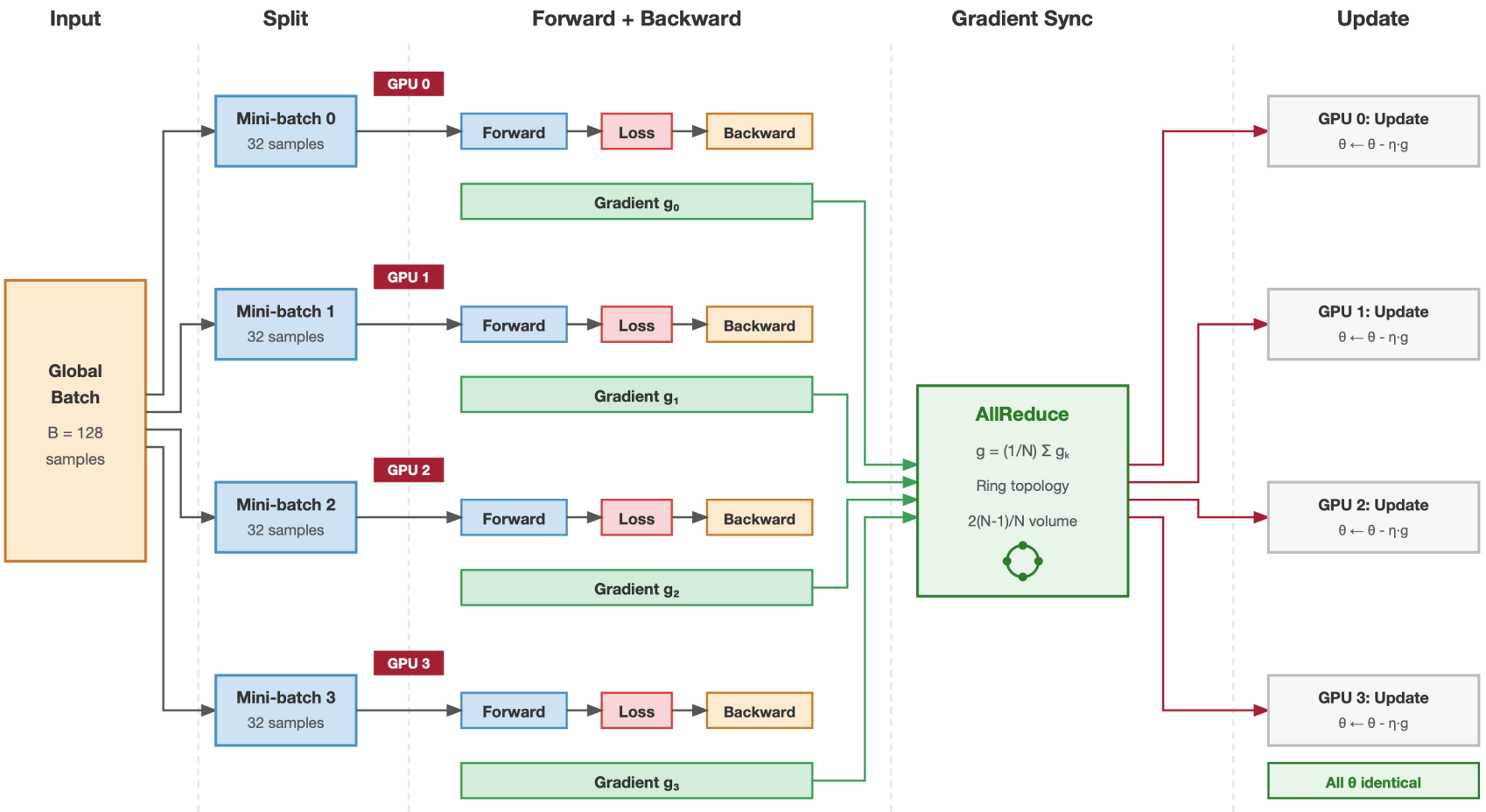
Why do we want more GPUs?

Data Parallelism

- Goal: Scale compute (i.e., process more data faster)

- Key idea: Partition training *data* across many GPUs

Data Parallelism



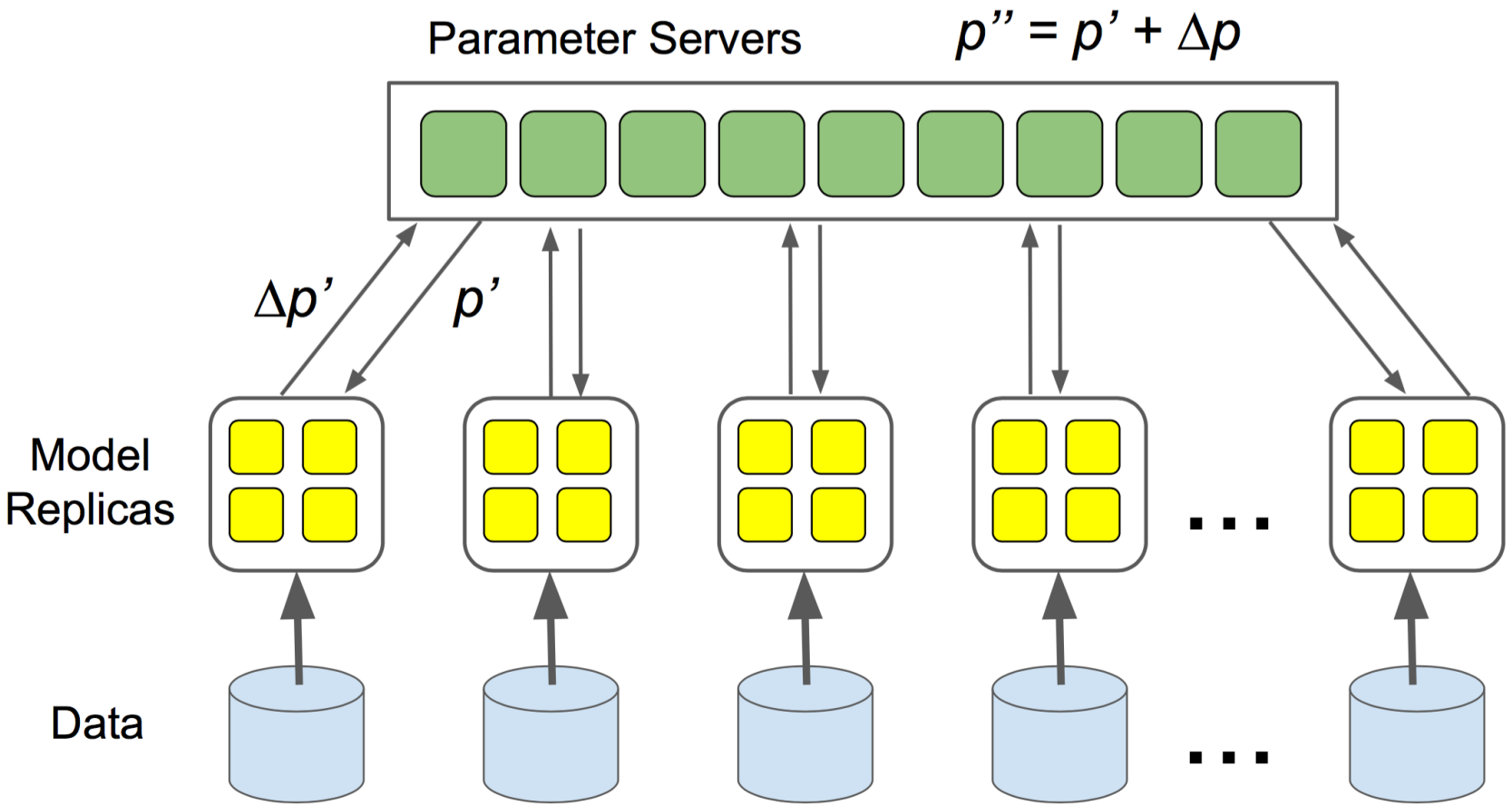
Embarrassingly parallel compute; communication only at gradient sync (10-40% of step time)

N = 4 GPUs, global batch B = 128, per-GPU micro-batch = 32

https://mlsysbook.ai/vol2/distributed_training/distributed_training.html

Key question: How do we synchronize gradients across all parallel workers?

Option 1: Parameter Servers



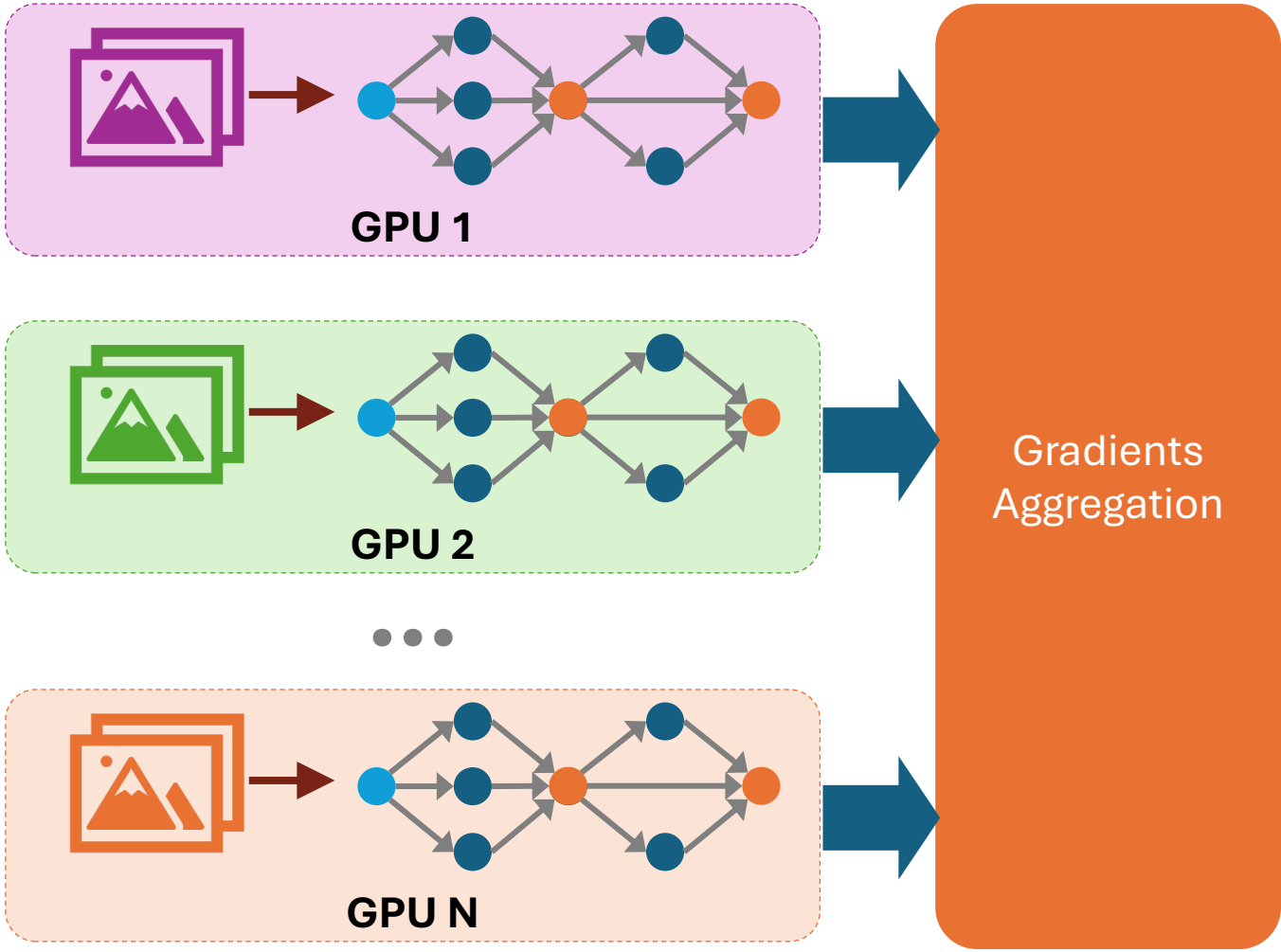
Workers push gradients to parameter servers and pull updated parameters back.

Downsides?

Slides adapted from Tianqi Chen and Zhihao Jia, CMU

Synchronous Data Parallelism

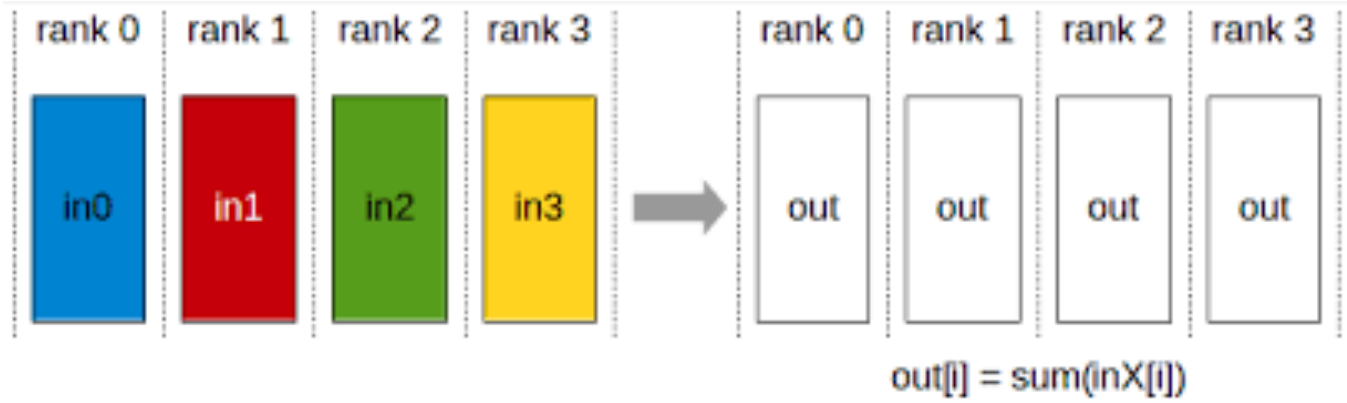
- *Replicate* model on each GPU
- For each iteration:
  - *Partition* data across GPUs
  - Each GPU: Fwd/Bwd pass
    - Generate gradients unique to each GPU
  - Sync gradients (e.g., avg) across all GPUs
  - Update weights (synced) on all GPUs



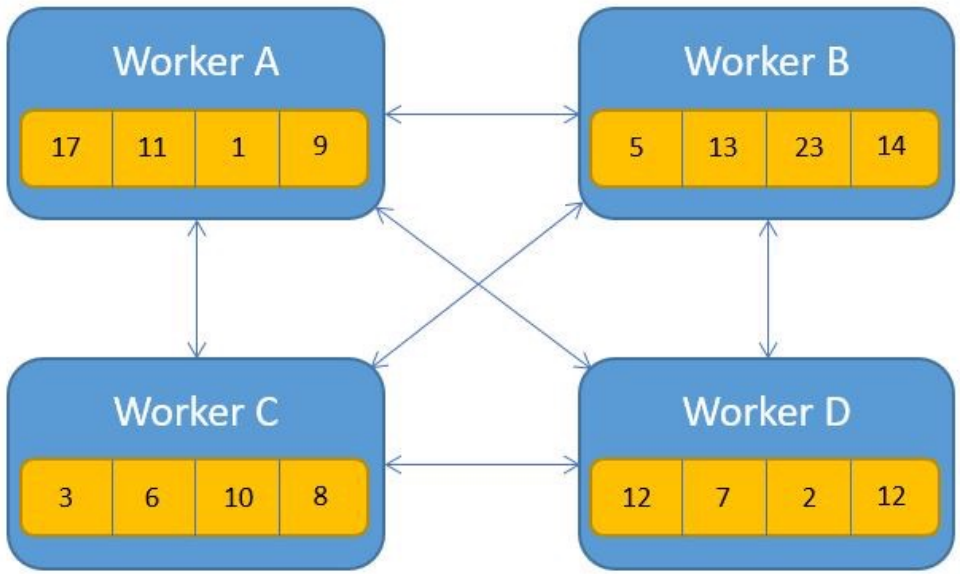
Slides adapted from Tianqi Chen and Zhihao Jia, CMU

All Reduce

- Goal: Perform element-wise reduction across multiple devices



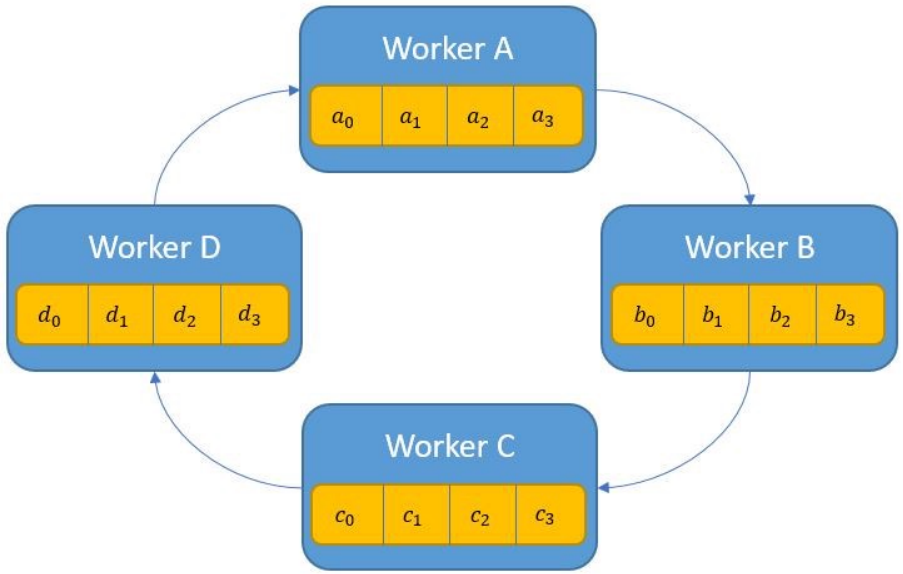
Naïve AllReduce



https://medium.com/data-science/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

Ring AllReduce

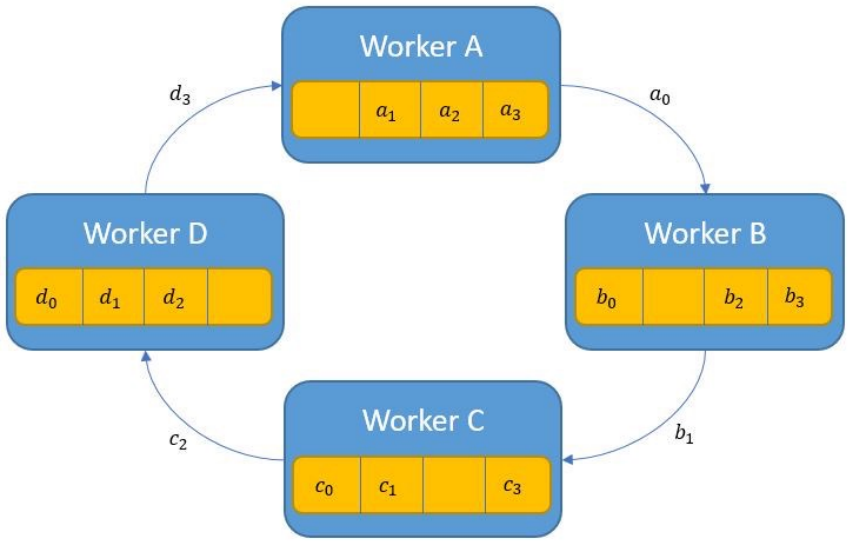
- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation):
  - Each worker sends one slice (M/N params) to the next worker on the ring
  - Worker reduces its received slice with current slice
  - Repeat N times



https://medium.com/data-science/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

Ring AllReduce

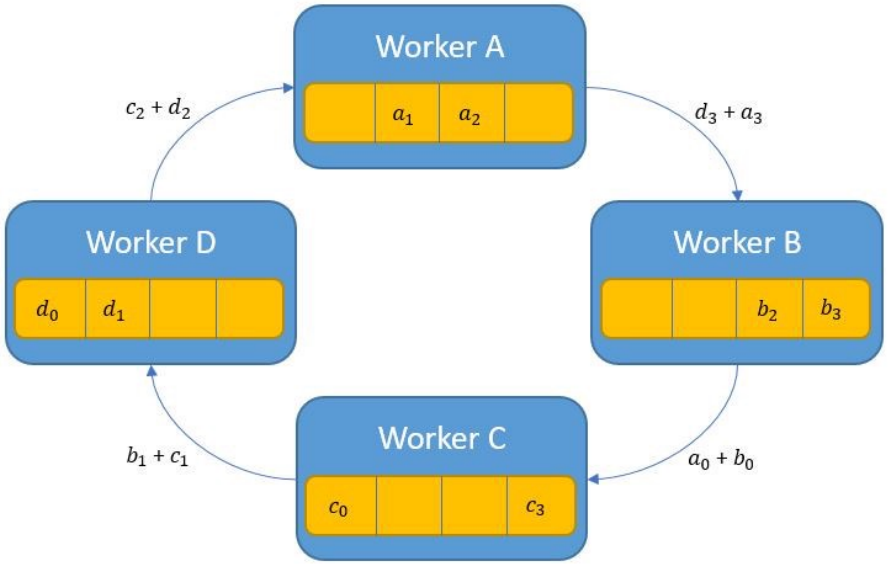
- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation):
  - Each worker sends one slice (M/N params) to the next worker on the ring
  - Worker reduces its received slice with current slice
  - Repeat N times



https://medium.com/data-science/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

Ring AllReduce

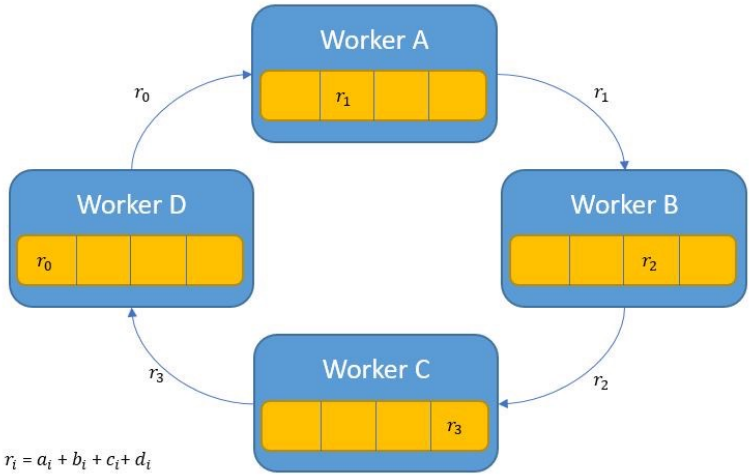
- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation):
  - Each worker sends one slice (M/N params) to the next worker on the ring
  - Worker reduces its received slice with current slice
  - Repeat N times



https://medium.com/data-science/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

Ring AllReduce

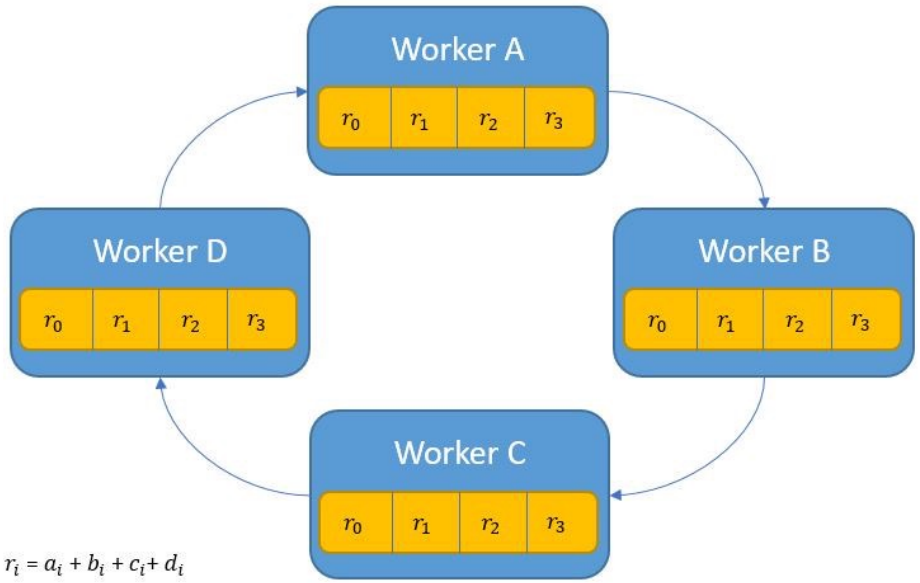
- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation):
  - Each worker sends one slice (M/N params) to the next worker on the ring
  - Worker reduces its received slice with current slice
  - Repeat N times
  - End up with reduced version of one slice



https://medium.com/data-science/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

Ring AllReduce

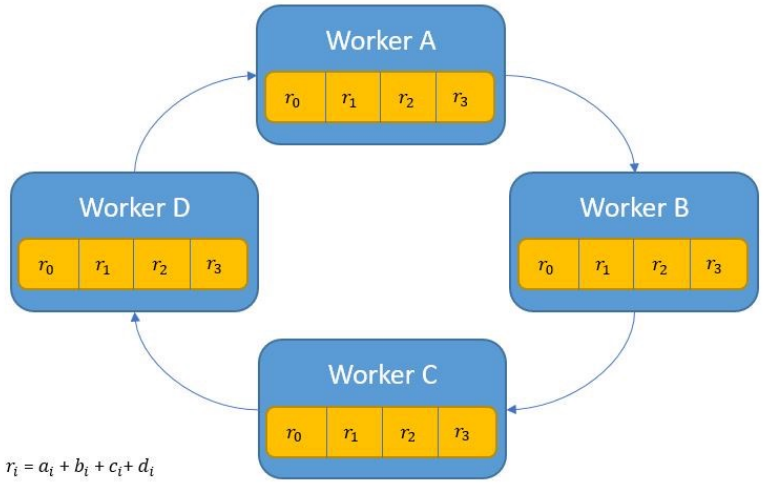
- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation):
- Step 2 (Broadcast):
  - Each worker sends one slice of reduced parameters to the next worker
  - Repeat N times



https://medium.com/data-science/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation):
  - Each worker sends one slice (M/N params) to the next worker on the ring
  - Worker reduces its received slice with current slice
  - Repeat N times
- Step 2 (Broadcast):
  - Each worker sends one slice of reduced parameters to the next worker
  - Repeat N times



https://medium.com/data-science/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da

Issues with Data Parallelism

- Each GPU needs to store entire model!
- Ex: Training a 7B parameter Llama model in mixed precision. Does the full training state fit on an H100-80GB GPU?

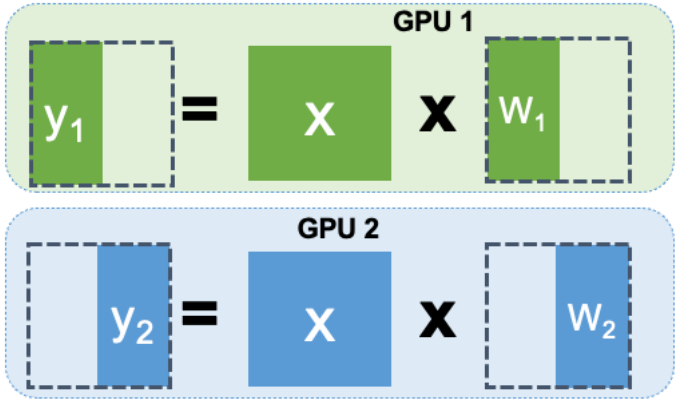
Model Parallelism

- Goal: Scale **parameters (memory)**
- Key idea: Shard **model** across GPUs
- Two main flavors:
  - Tensor Parallelism
  - Pipeline Parallelism

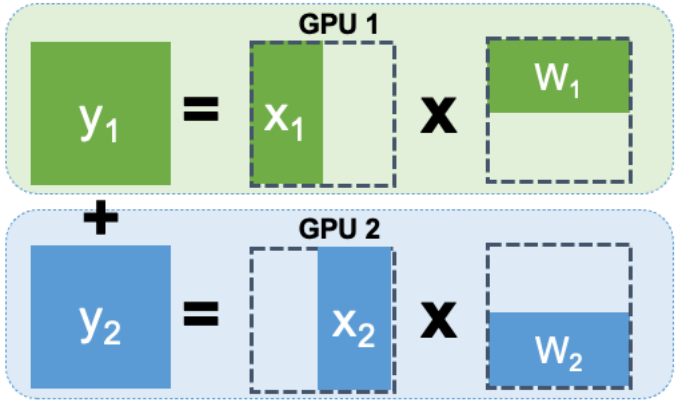
Tensor Parallelism

- Shard *model weights* across GPUs

Column-Parallel



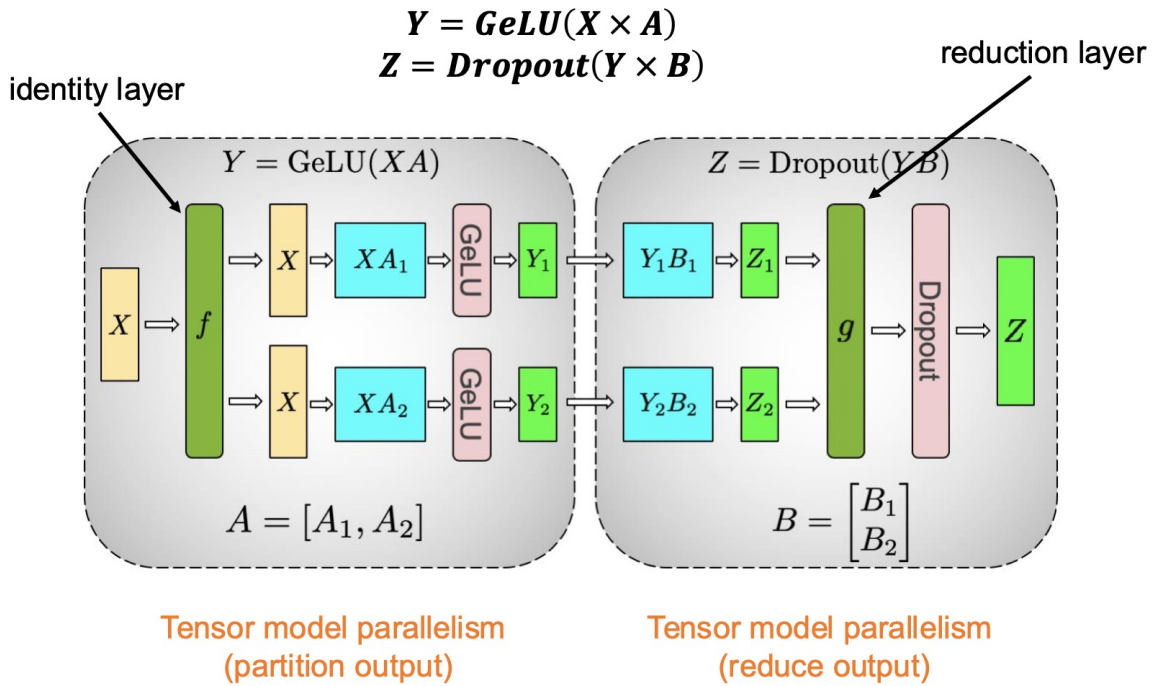
Row-Parallel



Megatron-LM Tensor Parallelism

Idea: Column-wise partition followed by row-wise to avoid an collective between them - only need two AR (fwd, bwd)

- QKV projection (column), attention output projection (row)
- (shown below) FFN 1 (column), FFN 2 (row)



Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism

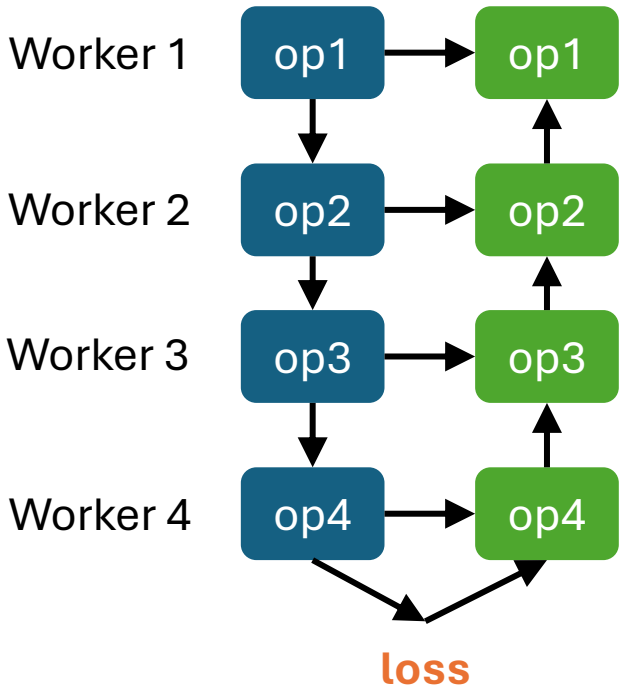
Downsides of TP?

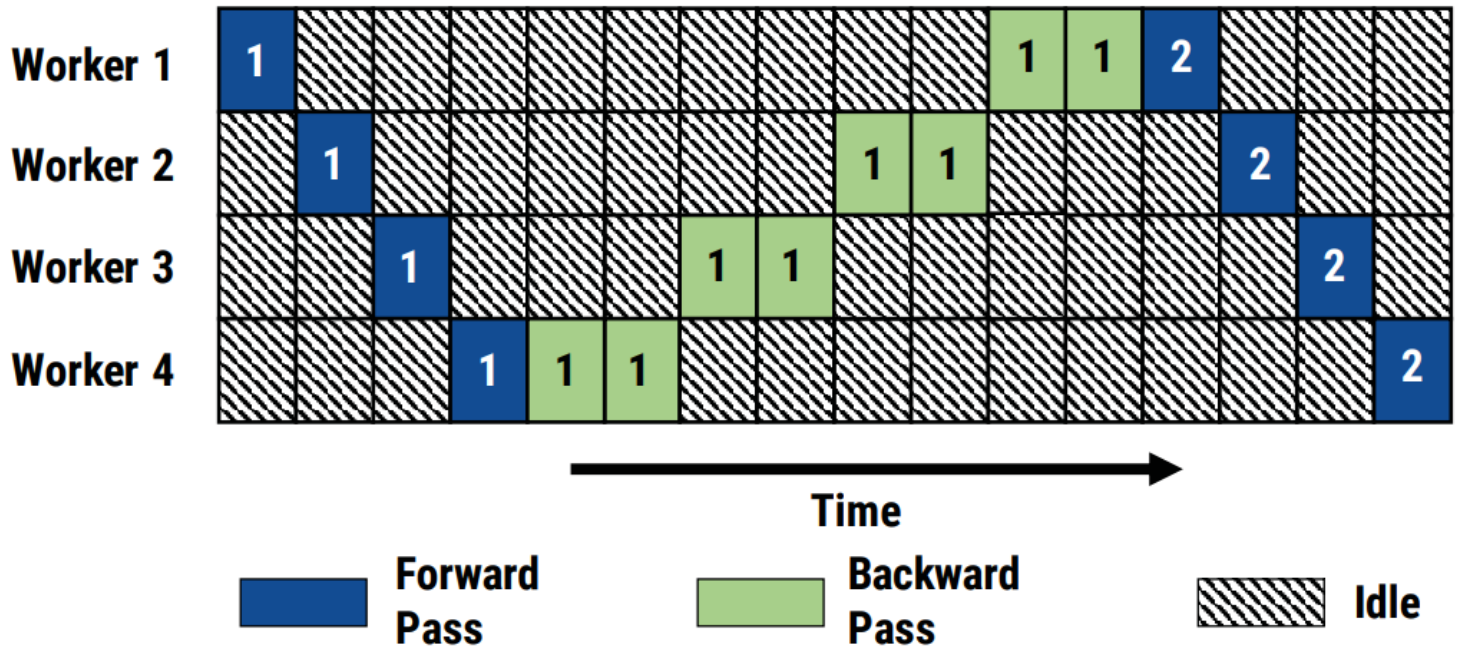
- GPT-3 175B: H = 12288, L = 96 layers, S = 2048, microbatch = 1, bf16
- AllReduce Size = B * S * H * 2 bytes
  - 50 MB
- Per Layer (4 per layer)
  - 200 MB
- Per batch (96 layers)
  - **19.3 GB**

Pipeline Parallelism

- Goal: Reduce communication requirements between GPUs
- Idea: Shard *layers* across GPUs

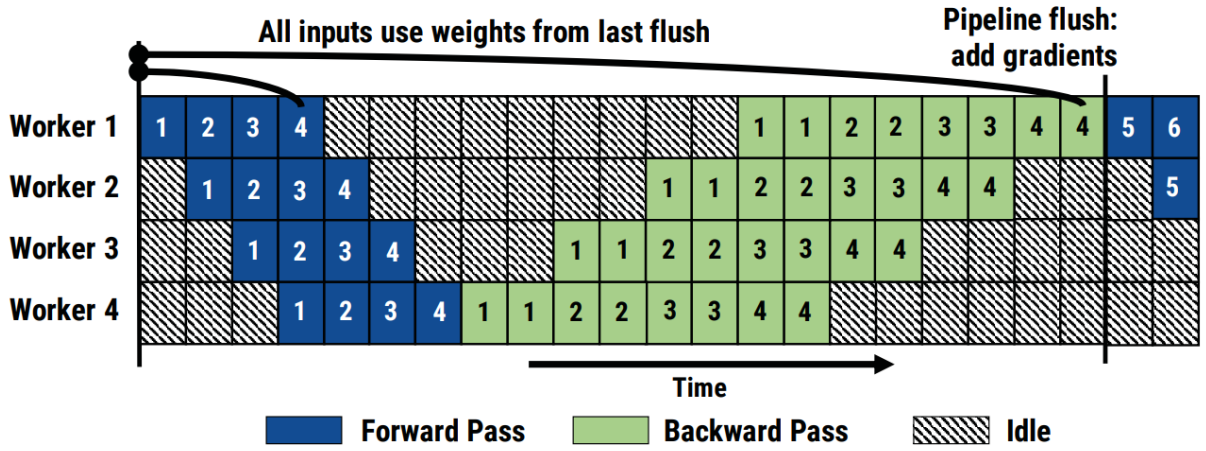
Pipeline Parallelism





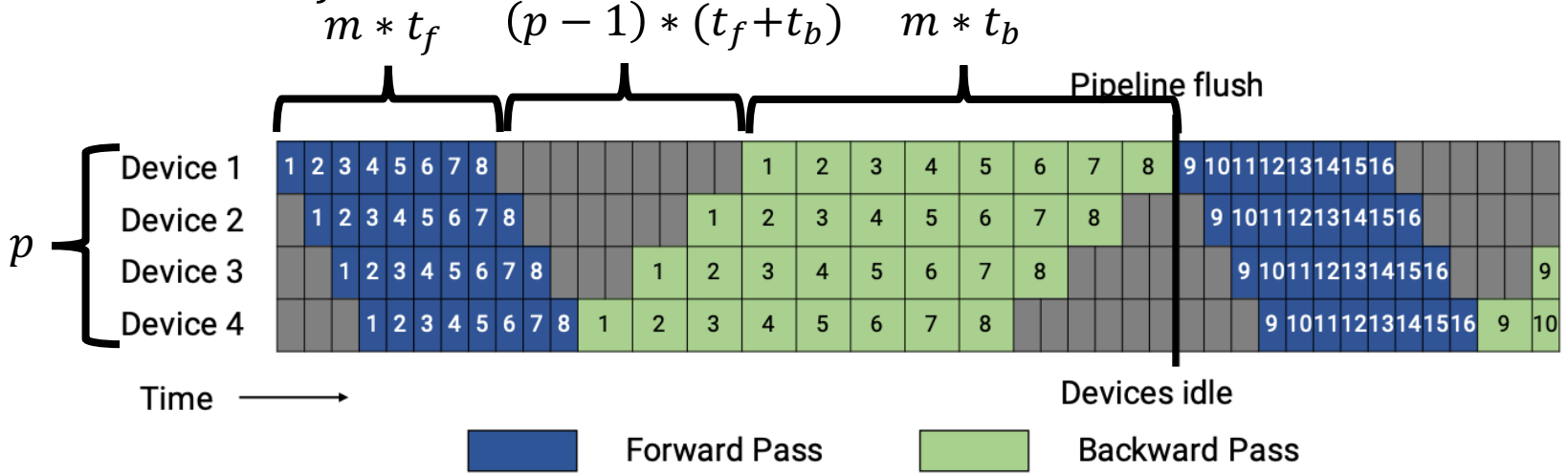
Micro-batching

- Divide mini-batch into multiple **micro-batches**
- Pipeline forward and backward computations across micro-batches



Device Utilization

- m : micro-batches in a mini-batch
- p: number of pipeline stages
- All stages take t_f/ t_b to process a forward (backward) micro-batch



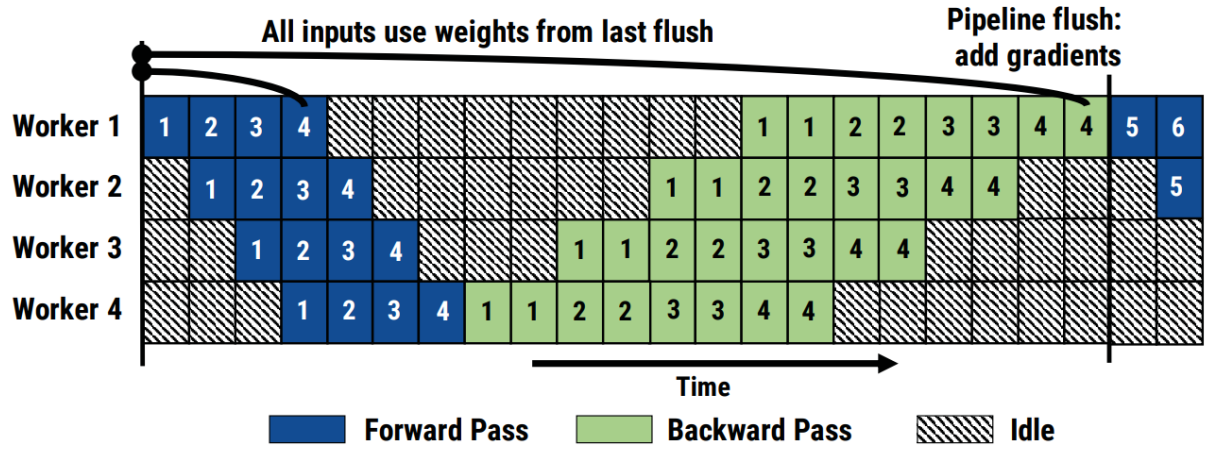
$$\textbf{BubbleFraction} = \frac{(p-1) * (t_f + t_b)}{m * t_f + m * t_b} = \frac{p-1}{m}$$

How to improve pipeline parallelism efficiency?

$$\boldsymbol{BubbleFraction} = \frac{(\boldsymbol{p}-\boldsymbol{1}) * (\boldsymbol{t_f}+\boldsymbol{t_b})}{\boldsymbol{m} * \boldsymbol{t_f} + \boldsymbol{m} * \boldsymbol{t_b}} = \frac{\boldsymbol{p}-\boldsymbol{1}}{\boldsymbol{m}}$$

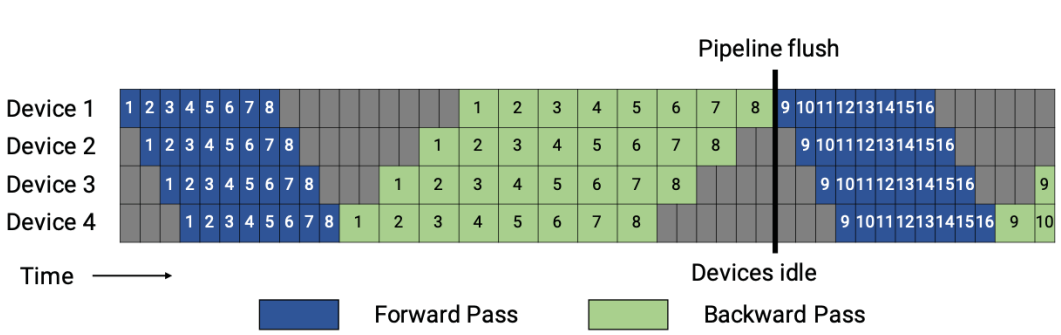
Memory requirements

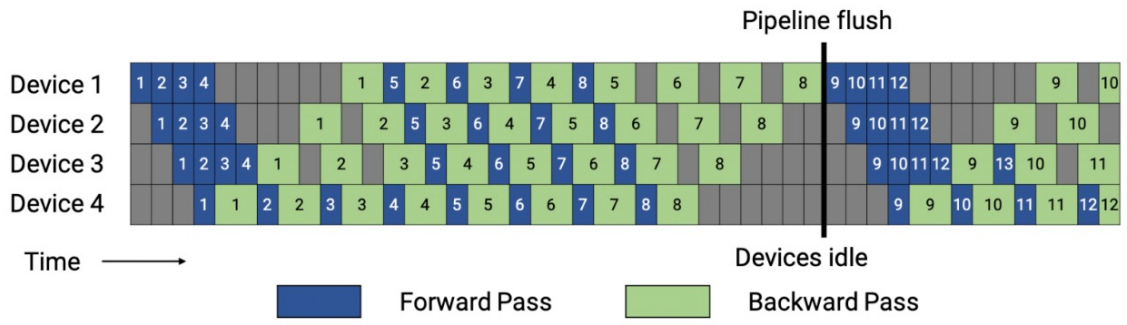
- Need to keep intermediate activations of **all micro-batches** before backprop. How can we improve the pipeline schedule to reduce memory requirements?



1F1B

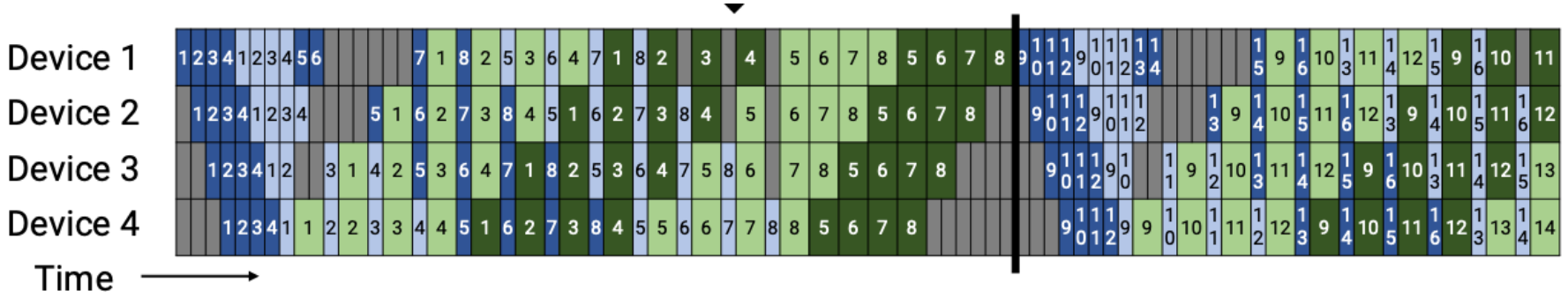
- One Forward, One Backward in the steady state
- Limits the number of in-flight microbatches to the pipeline depth
- **Reduces memory footprint of pipeline parallelism, but doesn't reduce pipeline bubbles**





Interleaved 1F1B

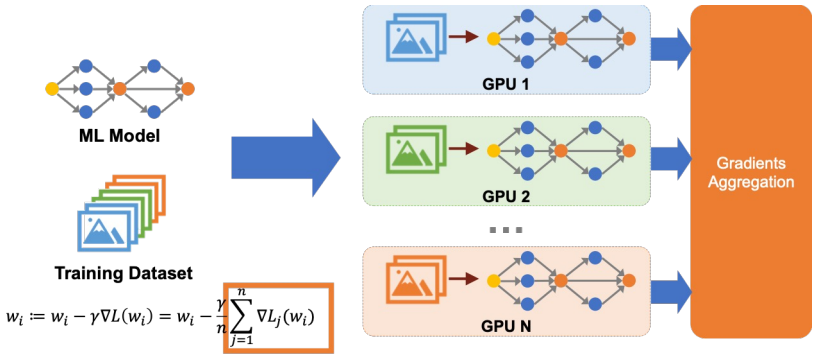
- Idea: Further divide each stage into *v* sub-stages
  - Before: Device 1 has layers 1-4. After: Device 1 has layers 1-2, 9-10
- Assign each device multiple pipeline stages, each with less computation than before
  - Forward (backward) time of each sub-stage is $\frac{t_f}{v}$ ($\frac{t_b}{v}$)
- Costs?

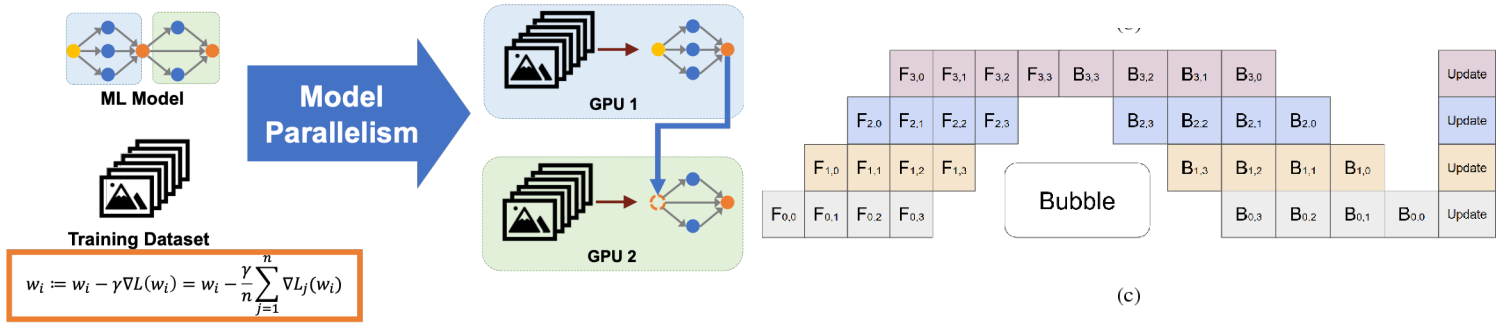


Each device is assigned two chunks. Dark colors show the first chunk and light colors show the second chunk.

$$\boldsymbol{BubbleFraction} = \frac{(\boldsymbol{p}-\boldsymbol{1}) * \frac{(\boldsymbol{t_f}+\boldsymbol{t_b})}{\boldsymbol{v}}}{\boldsymbol{m} * \boldsymbol{t_f} + \boldsymbol{m} * \boldsymbol{t_b}} = \frac{\boldsymbol{1}}{\boldsymbol{v}} * \frac{\boldsymbol{p}-\boldsymbol{1}}{\boldsymbol{m}}$$

Summary





	Data Parallelism	Tensor Model Parallelism	Pipeline Model Parallelism
Pros	✓ Massively parallelizable ✓ Require no communication during forward	✓ Support training large models ✓ Efficient for models with large numbers of parameters	✓ Support large-batch training ✓ Efficient for deep models
Cons	❖ Do not work for models that cannot fit on a GPU ❖ Do not scale for models with large numbers of parameters	❖ Limited parallelizability; cannot scale to large numbers of GPUs ❖ Need to transfer intermediate results in forward/backward	❖ Limited utilization: bubbles in forward/backward

Slides adapted from Tianqi Chen and Zhihao Jia, CMU

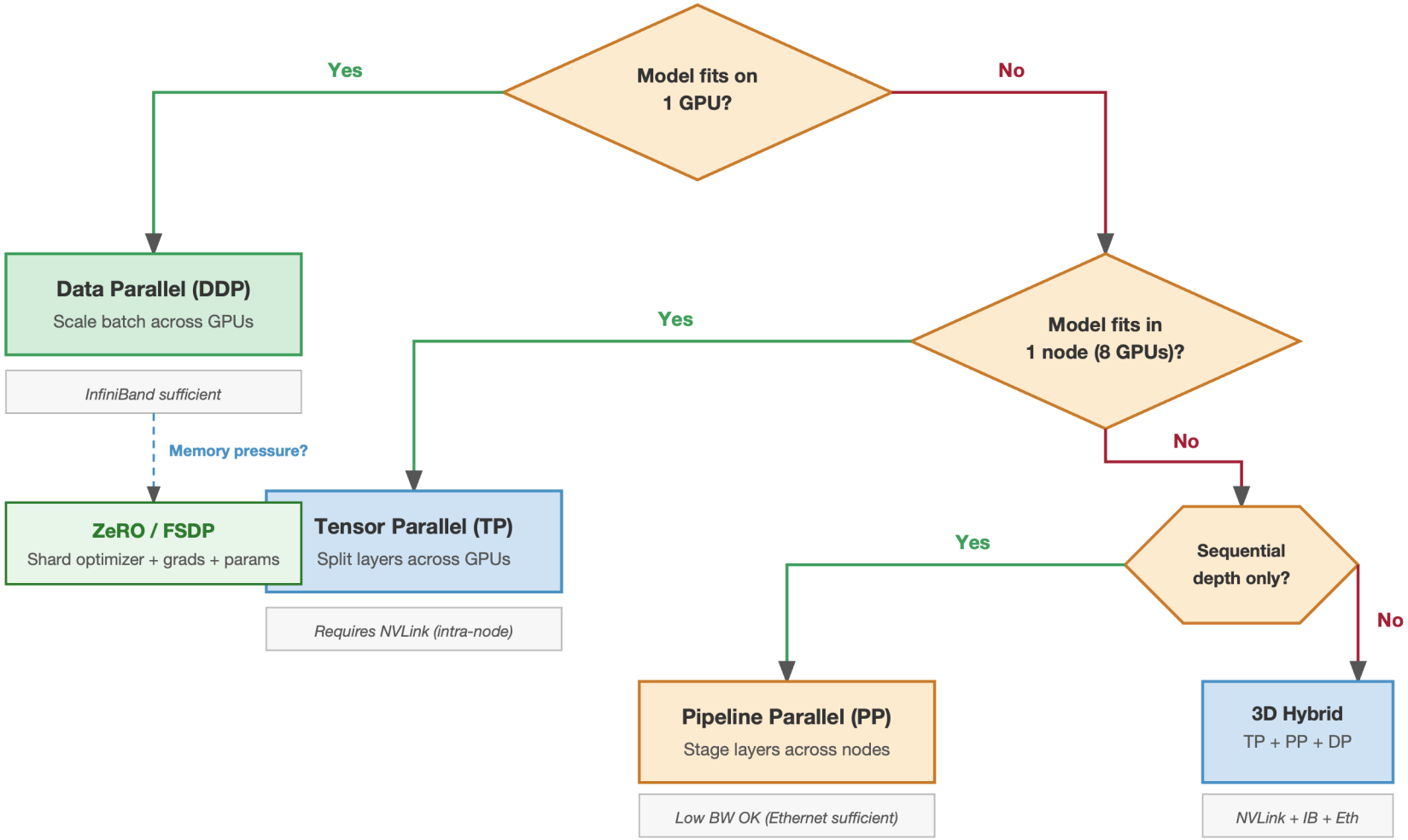
Hybrid Parallelism

- Data Parallelism: Higher training throughput if model fits in memory
- Tensor Parallelism: Fit large model into GPU, and there exists high BW interconnect between few GPUs
- Pipeline Parallelism: Fit large model into GPU, and we need to scale to many servers over larger network. However, bubbles limit utilization.

Hybrid Parallelism

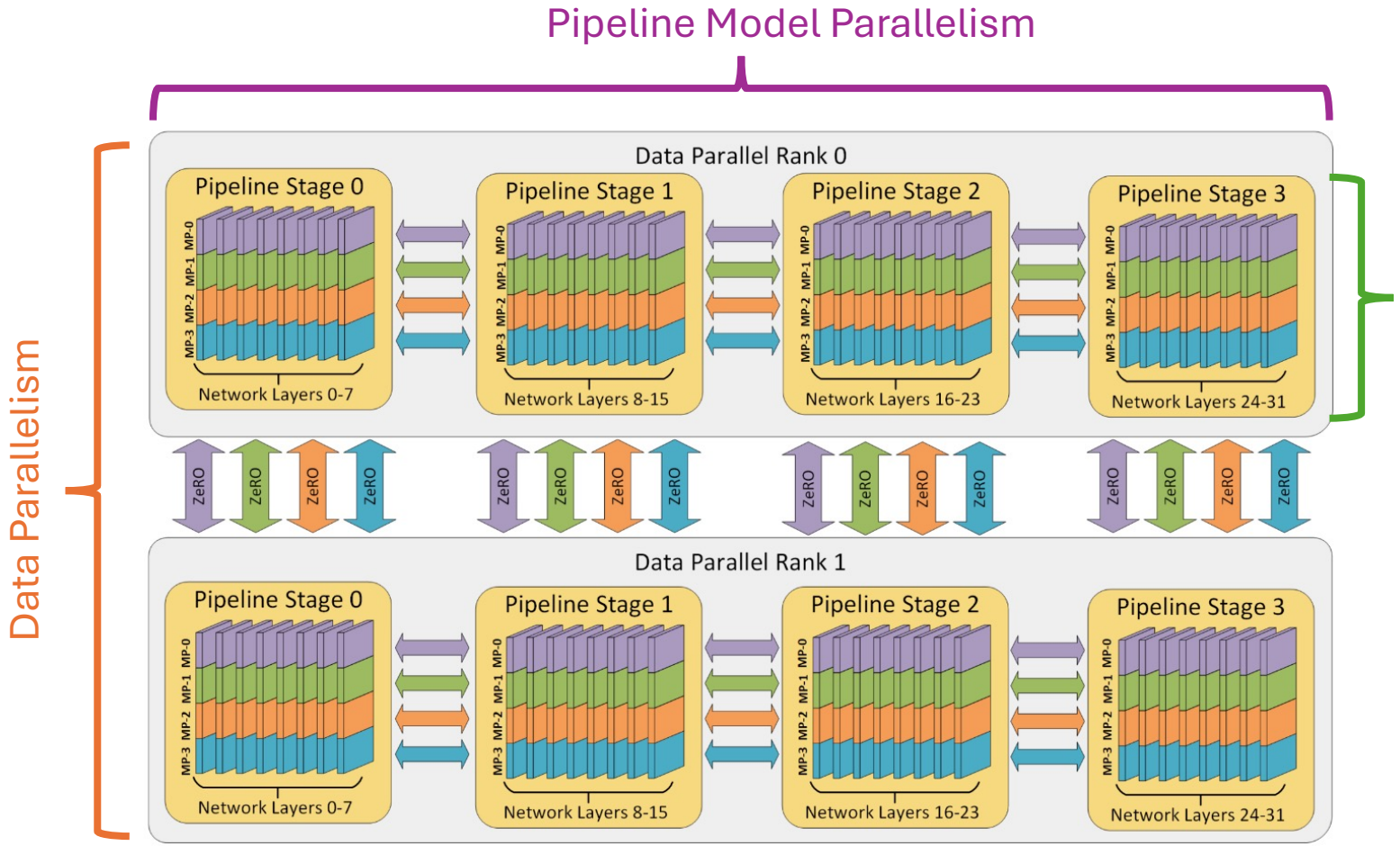
- We have a large model and lots of GPUs, what do we do?
- Heuristic
  - Use Tensor Parallelism *within* server (NVLink)
  - Use Pipeline Parallelism *between* servers (RDMA) until model can fit into GPU
  - Use Data Parallelism to train faster on more GPUs

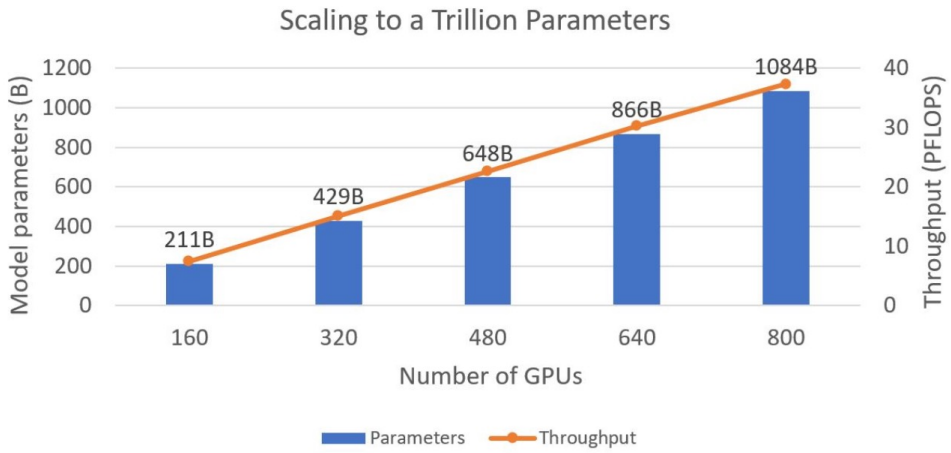
Parallelism Flowchart



https://mlsysbook.ai/vol2/distributed_training/distributed_training.html

DeepSpeed: 3D Model Parallelism

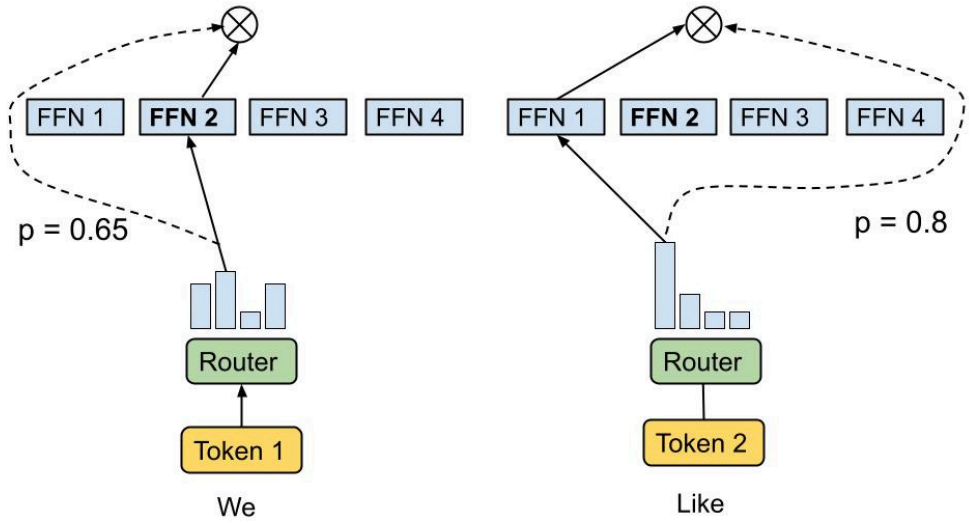




https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone/

Other Flavors of Parallelism

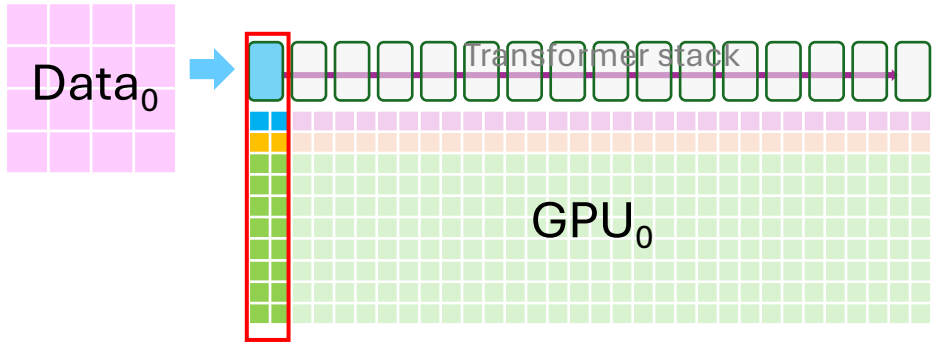
- **Expert Parallelism**: Shard/Replicate Experts (we'll discuss later) across GPUs
- **Sequence Parallelism:** Parallelize Layernorm, Dropout along sequence dimension of transformer
- **Context Parallelism**: Partition entire layer along sequence dimension
- …



How else can we train larger models?

- Each GPU needs to store entire model!
- Ex: Training a 7B parameter Llama model in mixed precision. Does the full training state fit on an H100-80GB GPU?

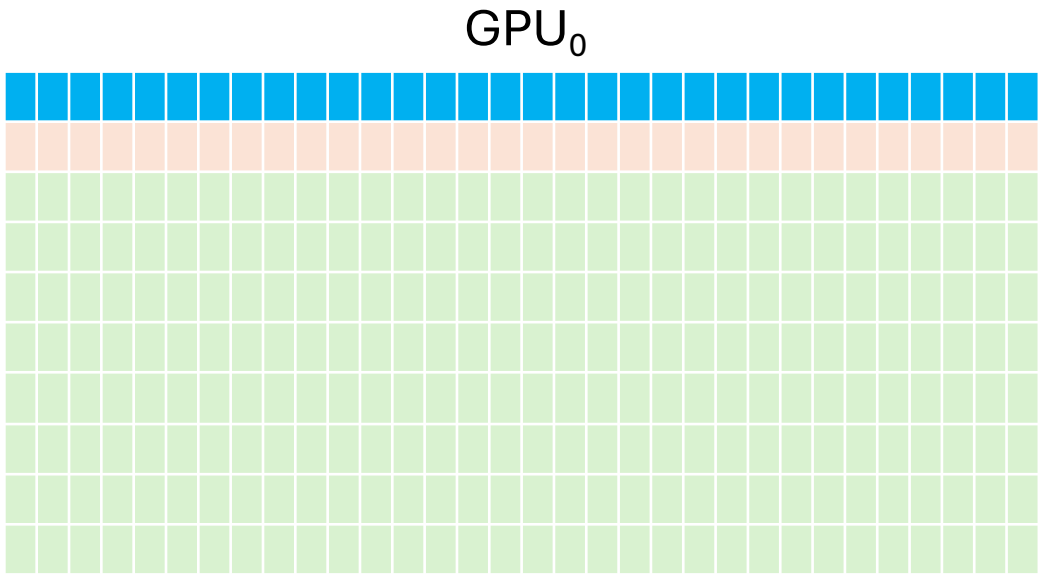
Understanding Training Memory Consumption



Each cell represents GPU memory used by its corresponding transformer layer

Adapted from Minjia Zhang, DeepSpeed

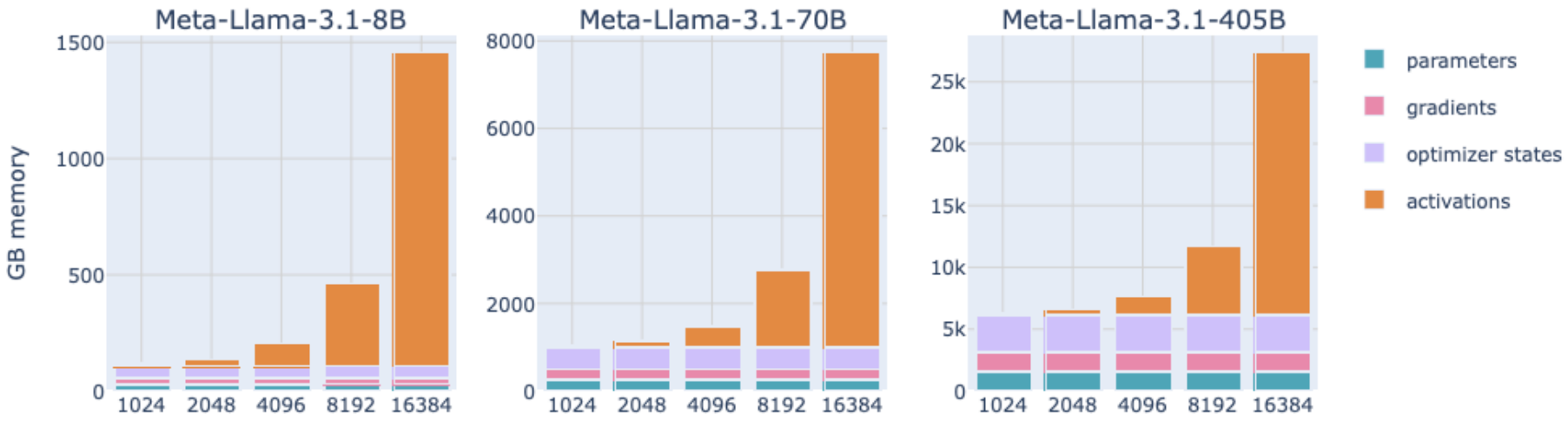
Counting Parameters



Not including input batch + activation memory

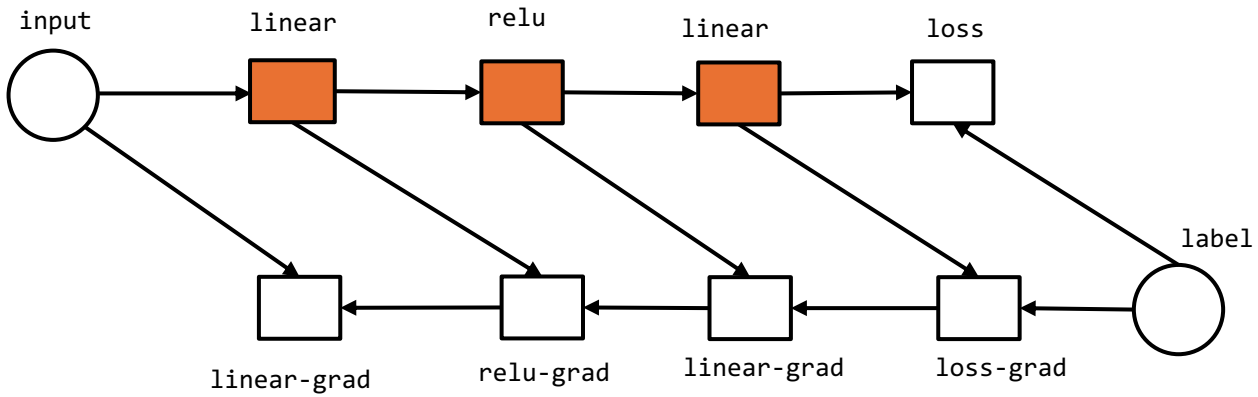
Adapted from Minjia Zhang, DeepSpeed

Activation Memory



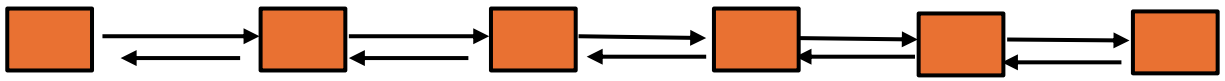
https://nanotron-ultrascale-playbook.static.hf.space/index.html#memory_for_activations

Activation Memory



Because the need to keep intermediate value around (checkpoint) for the gradient steps. Training a N-layer neural network would require $O(N)$ memory.

We will use the following simplified view to combine gradient and forward computation

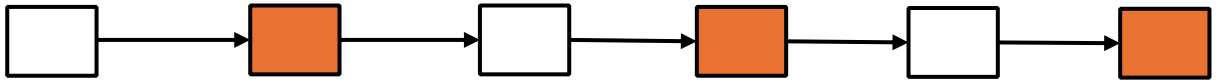


Slides adapted from Tianqi Chen and Zhihao Jia, CMU

Activation Checkpointing

Idea: Discard activations during forward pass to save memory, recompute when necessary during backward pass

Step 0:



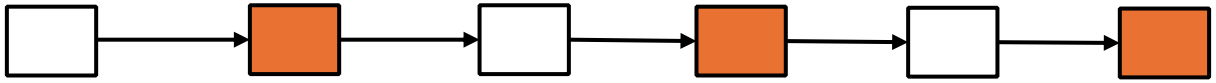
Step 1:

Step 2:

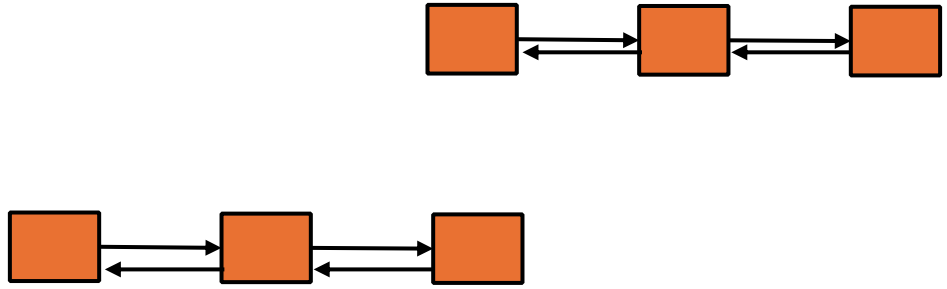
- Only checkpoint colored nodes (step 0)
- Recompute the missing intermediate nodes in small segments (step 1, 2)

Activation Checkpointing

Forward computation



Gradient per segment
with re-computation



For a N layer neural network,
if we checkpoint every K layers, what is the memory cost?

ZeRO

- Zero Redundancy Optimizer
- Memory optimization techniques for LLMs
- Deployed via DeepSpeed https://www.deepspeed.ai

ZeRO: Memory Optimizations Toward Training Trillion Parameter Models

Samyam Rajbhandari*, Jeff Rasley*, Olatunji Ruwase, Yuxiong He
{samyamr, jerasley, olruwase, yuxhe}@microsoft.com

ZeRO-Offload: Democratizing Billion-Scale Model Training

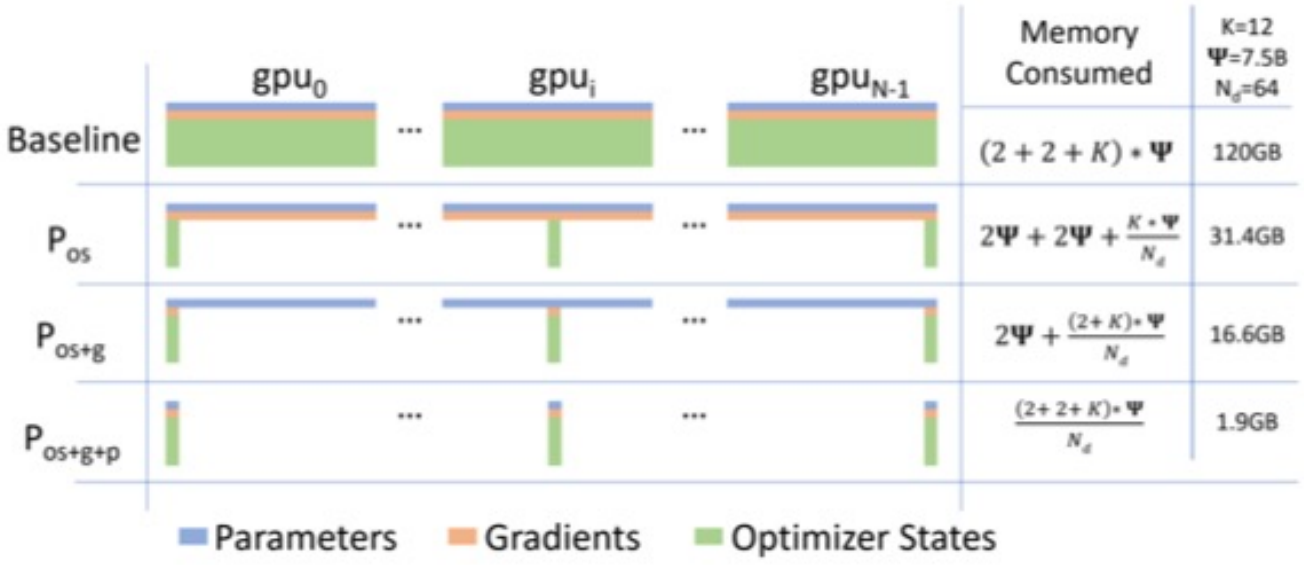
Jie Ren , Samyam Rajbhandari† , Reza Yazdani Aminabadi† , Olatunji Ruwase†*
Shuangyan Yang , Minjia Zhang† , Dong Li* , Yuxiong He†*
*†Microsoft , *University of California, Merced*
{jren6, syang127, dli35}@ucmerced.edu, {samyamr, yazdani.reza, olruwase, minjiaz, yuxhe}@microsoft.com

ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning

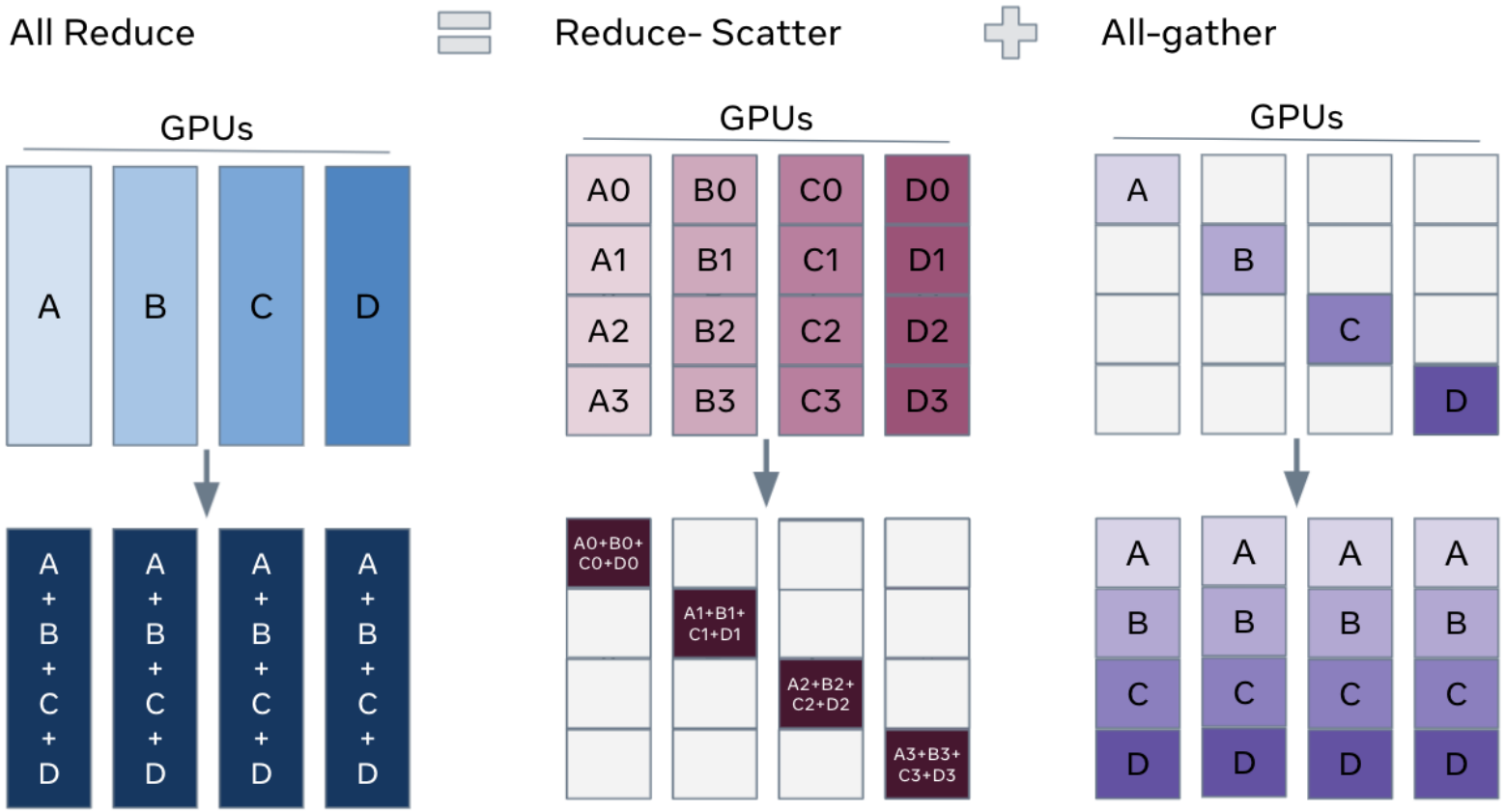
Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, Yuxiong He
{samyamr, olruwase, jerasley, shsmit, yuxhe}@microsoft.com

ZeRO-DP: ZeRO Powered DP

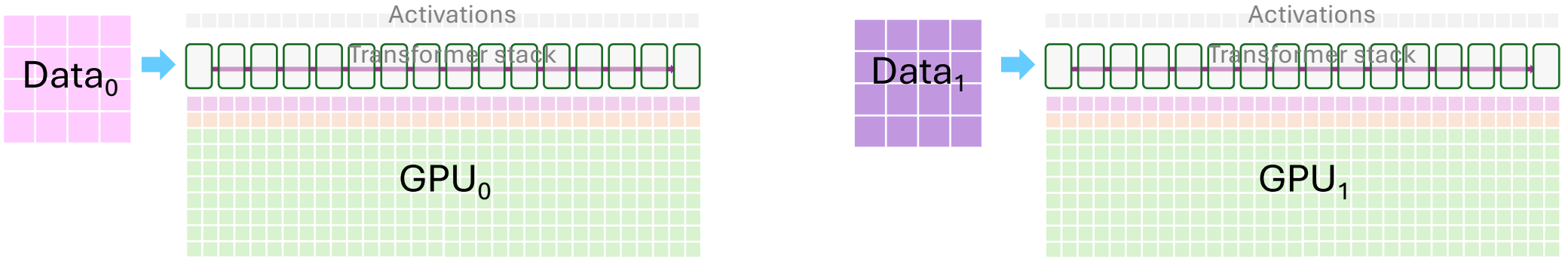
- Remove memory redundancy across data parallel processes
- Stage 1: Partition optimizer state
- Stage 2: Partition gradients
- Stage 3: Partition parameters



ReduceScatter, AllGather



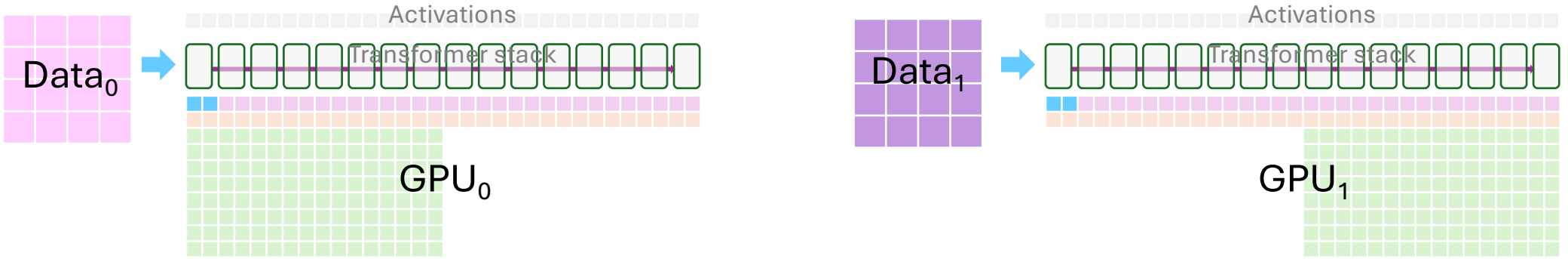
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1

Adapted from Minjia Zhang, DeepSpeed Presentation

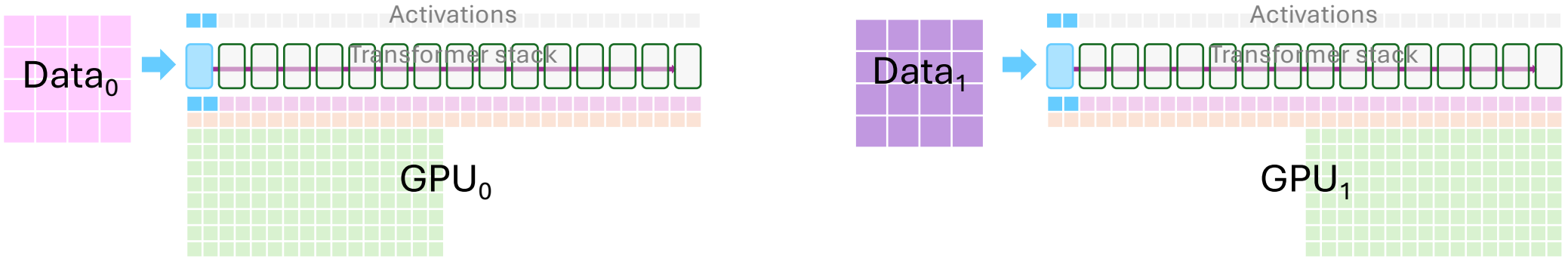
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs

Adapted from Minjia Zhang, DeepSpeed Presentation

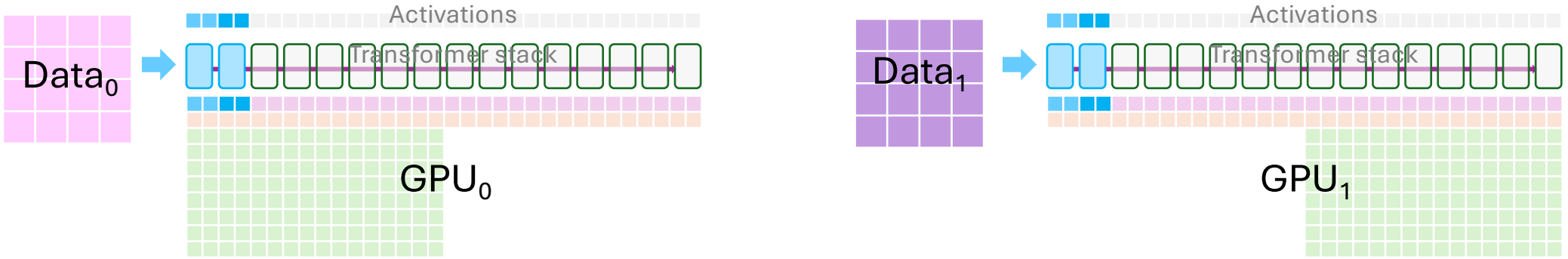
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Adapted from Minjia Zhang, DeepSpeed Presentation

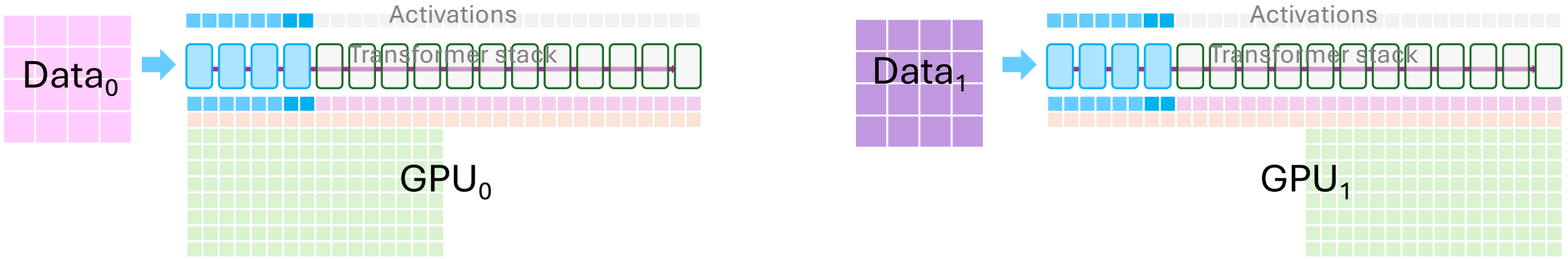
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Adapted from Minjia Zhang, DeepSpeed Presentation

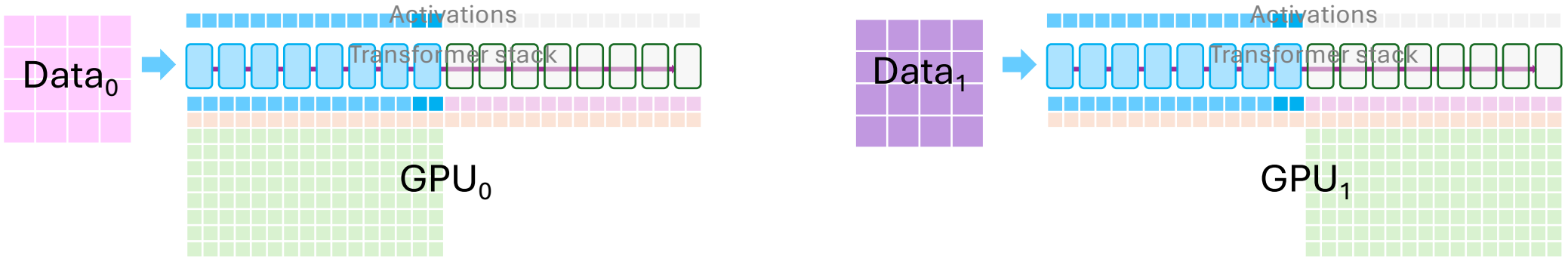
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Adapted from Minjia Zhang, DeepSpeed Presentation

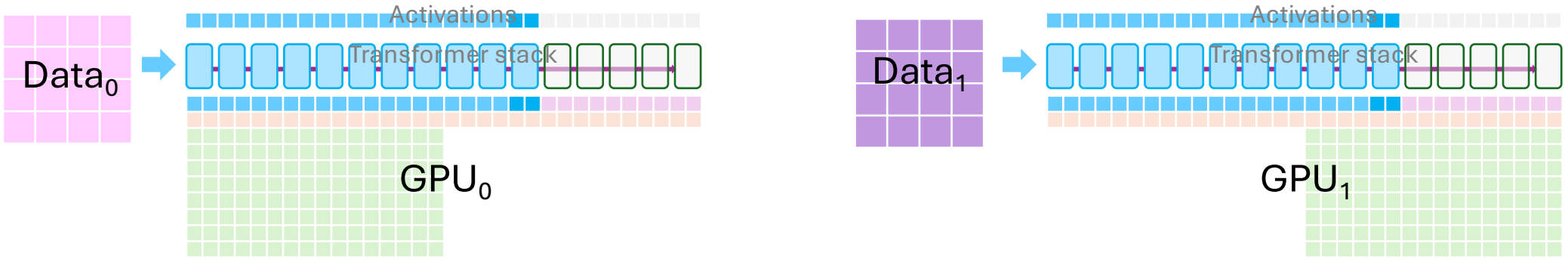
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Adapted from Minjia Zhang, DeepSpeed Presentation

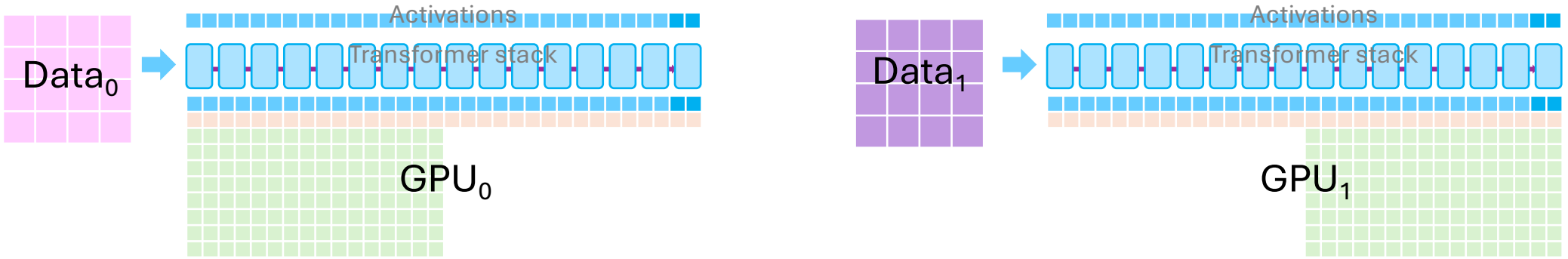
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Adapted from Minjia Zhang, DeepSpeed Presentation

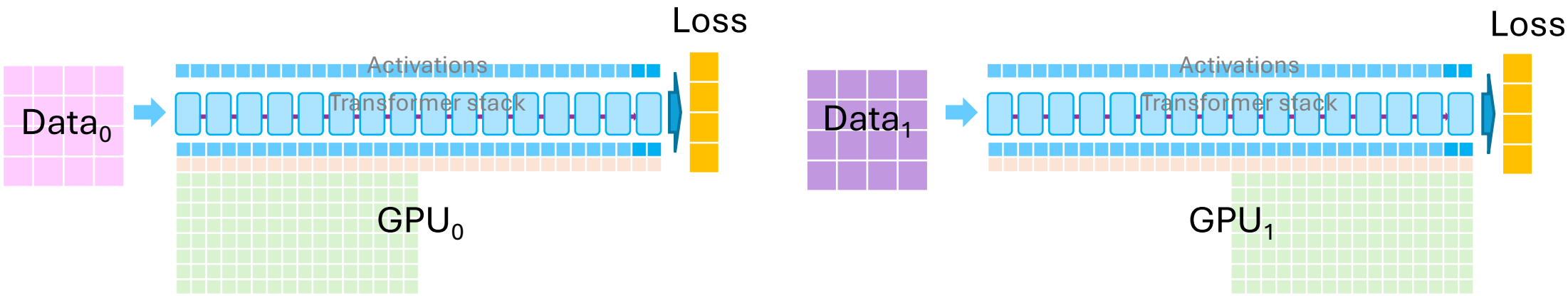
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Adapted from Minjia Zhang, DeepSpeed Presentation

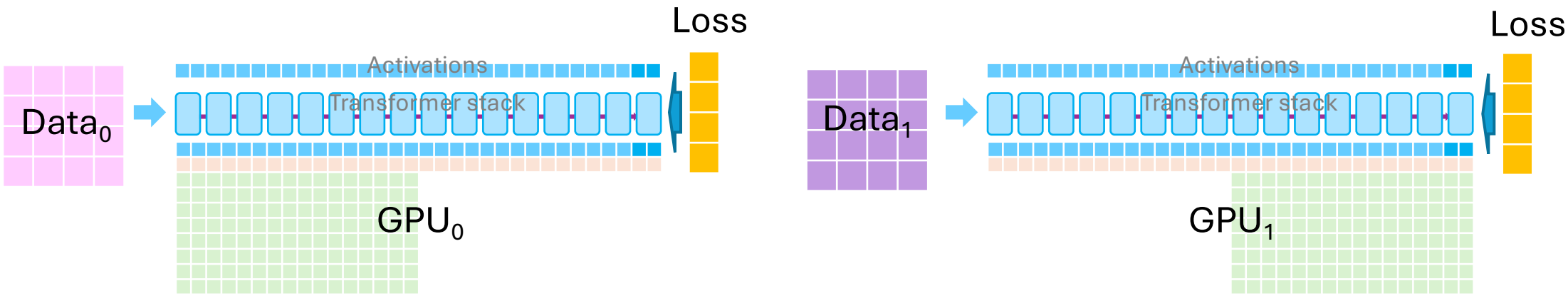
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Adapted from Minjia Zhang, DeepSpeed Presentation

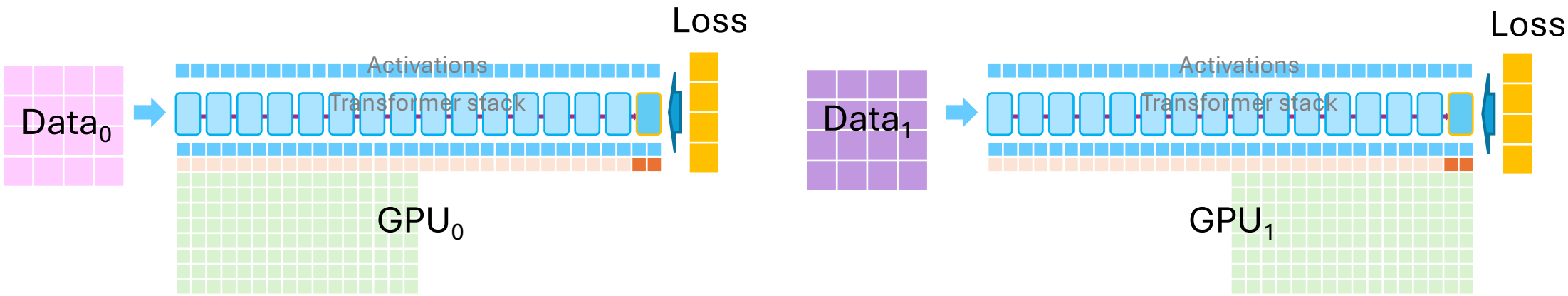
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

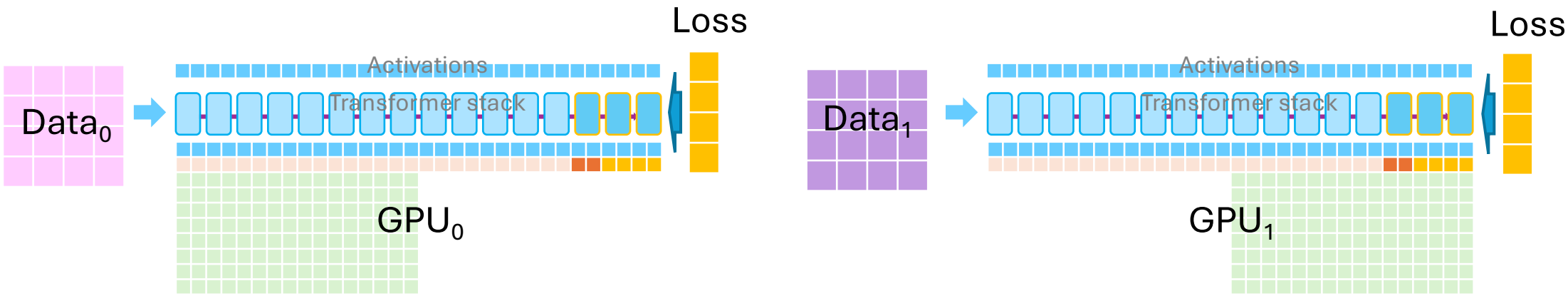
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

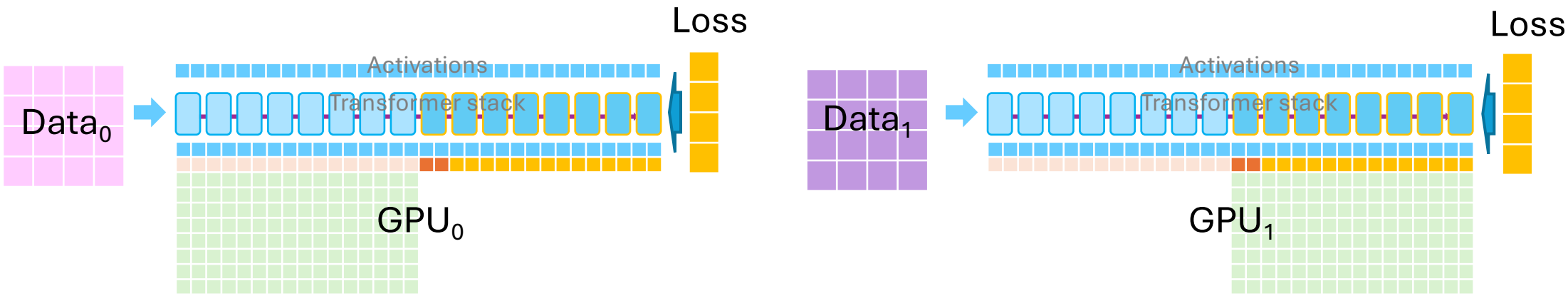
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

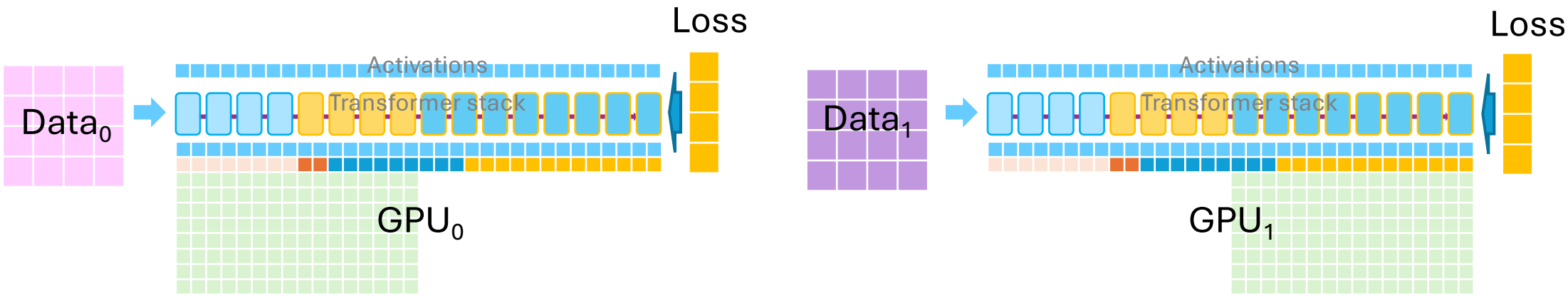
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

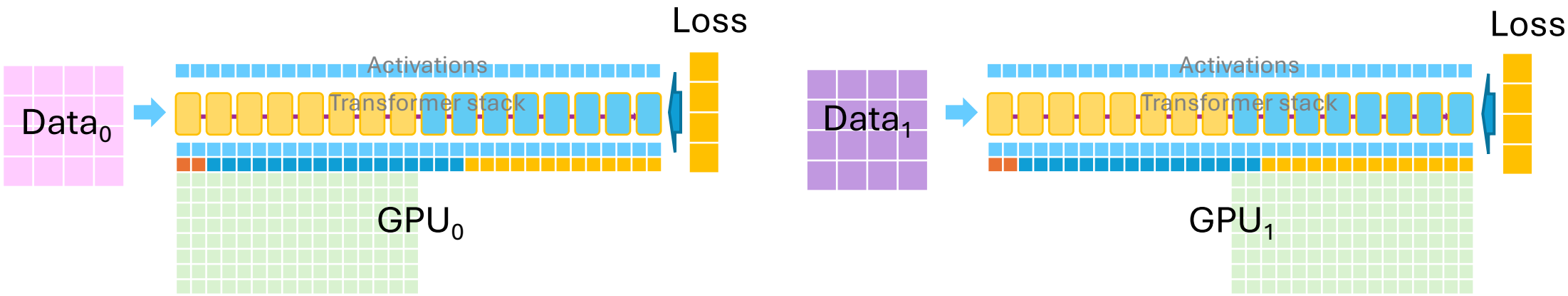
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

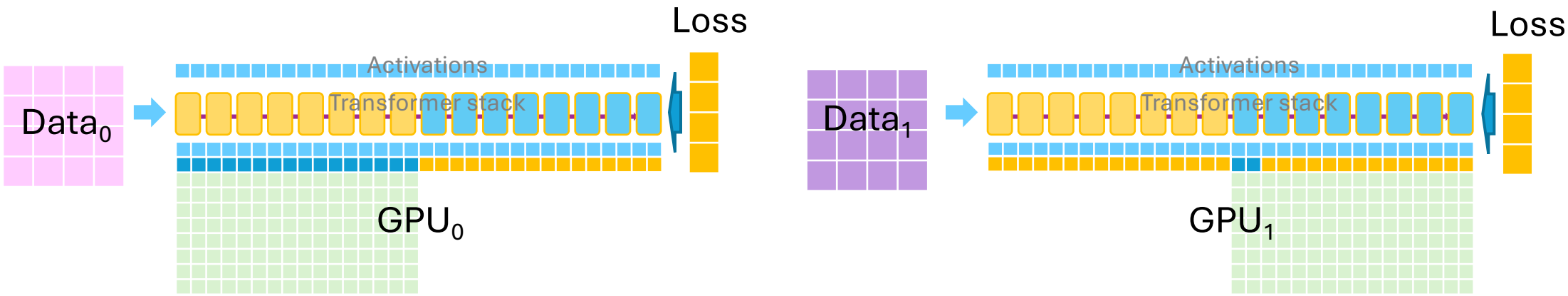
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

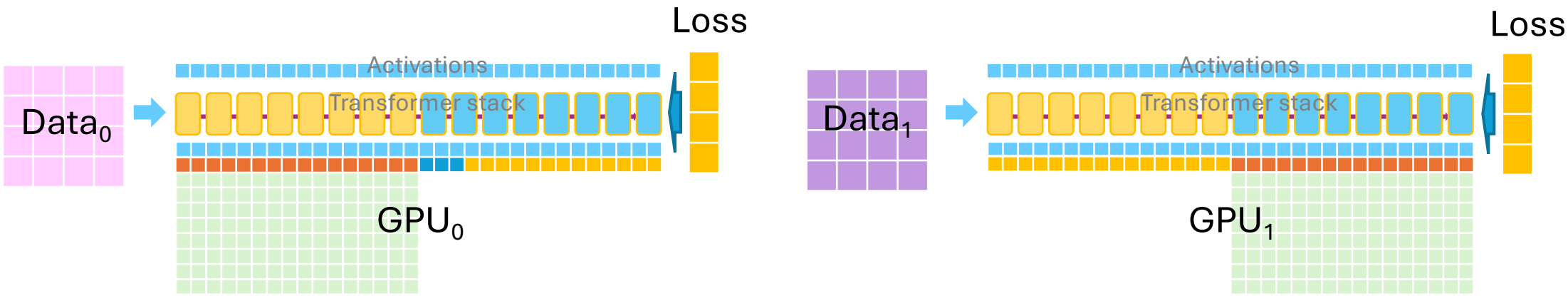
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average

Adapted from Minjia Zhang, DeepSpeed Presentation

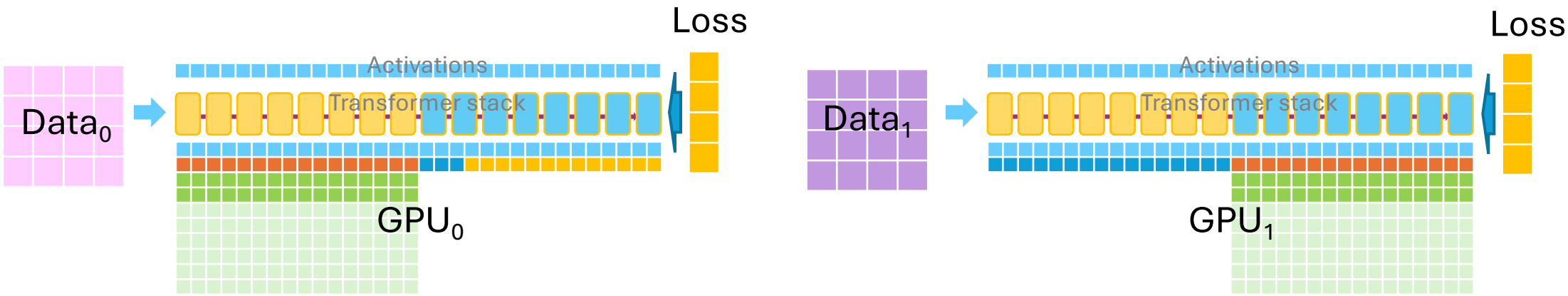
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average

Adapted from Minjia Zhang, DeepSpeed Presentation

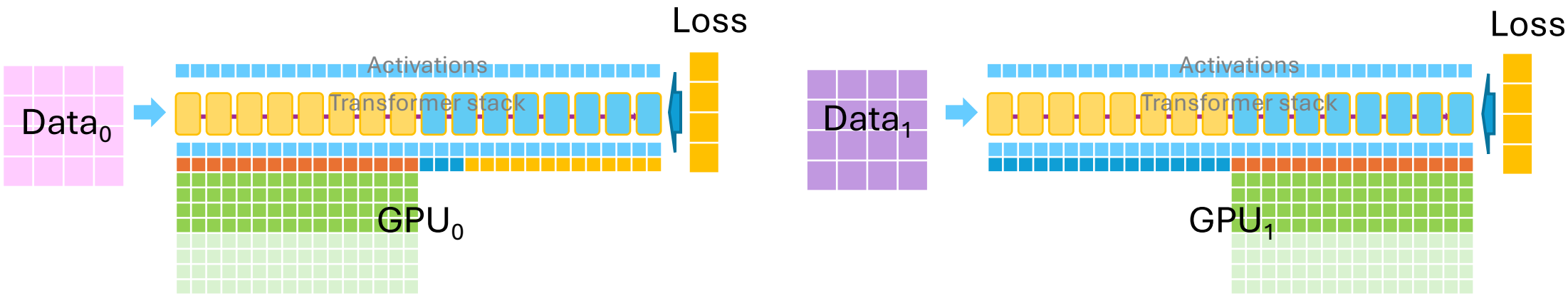
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average
- Update the FP32 weights with ADAM optimizer

Adapted from Minjia Zhang, DeepSpeed Presentation

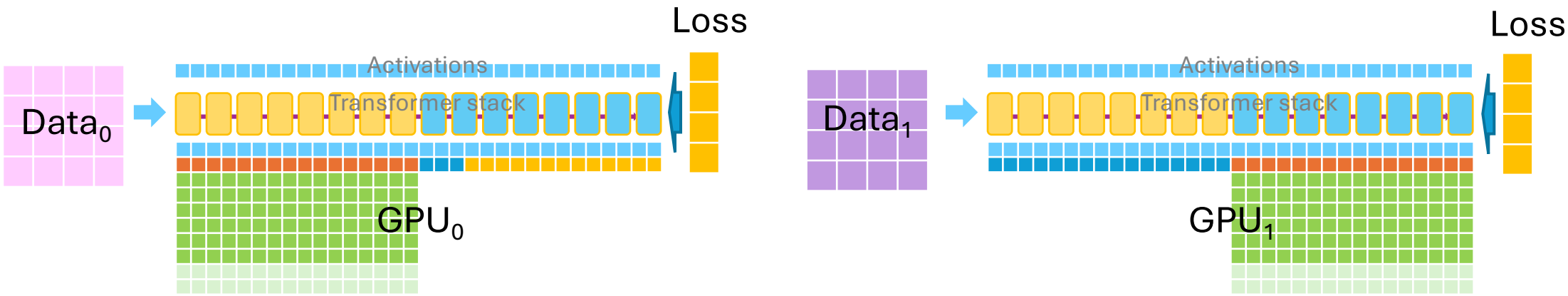
ZeRO Stage 1: Partitioning Optimizer States



- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average
- Update the FP32 weights with ADAM optimizer

Adapted from Minjia Zhang, DeepSpeed Presentation

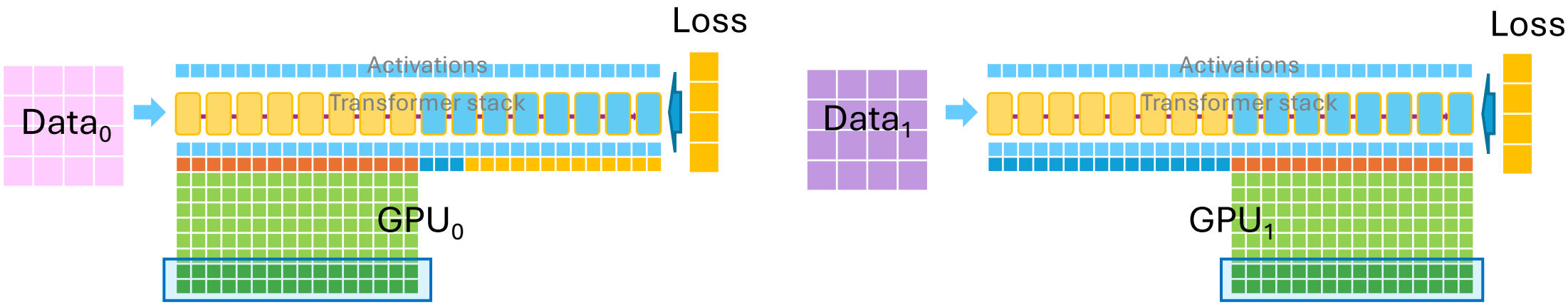
ZeRO Stage 1: Partitioning Optimizer States



- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average
- Update the FP32 weights with ADAM optimizer

Adapted from Minjia Zhang, DeepSpeed Presentation

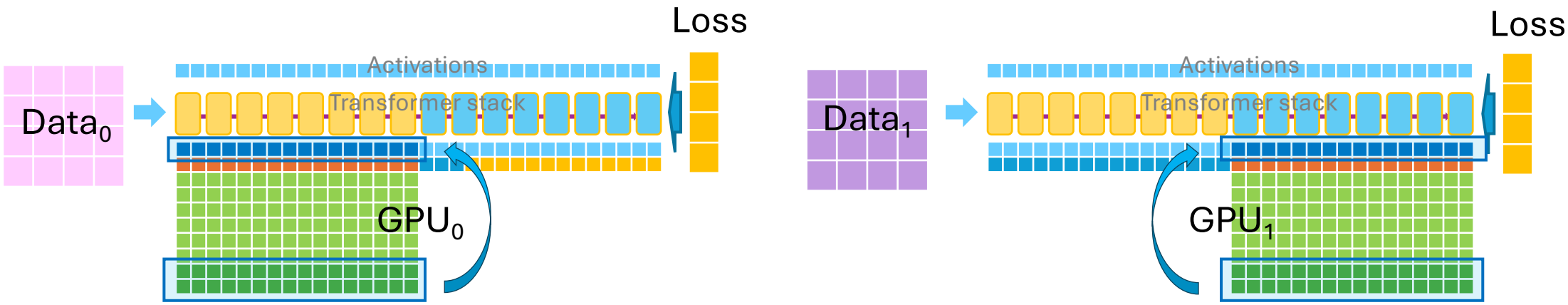
ZeRO Stage 1: Partitioning Optimizer States



- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average
- Update the FP32 weights with ADAM optimizer

Adapted from Minjia Zhang, DeepSpeed Presentation

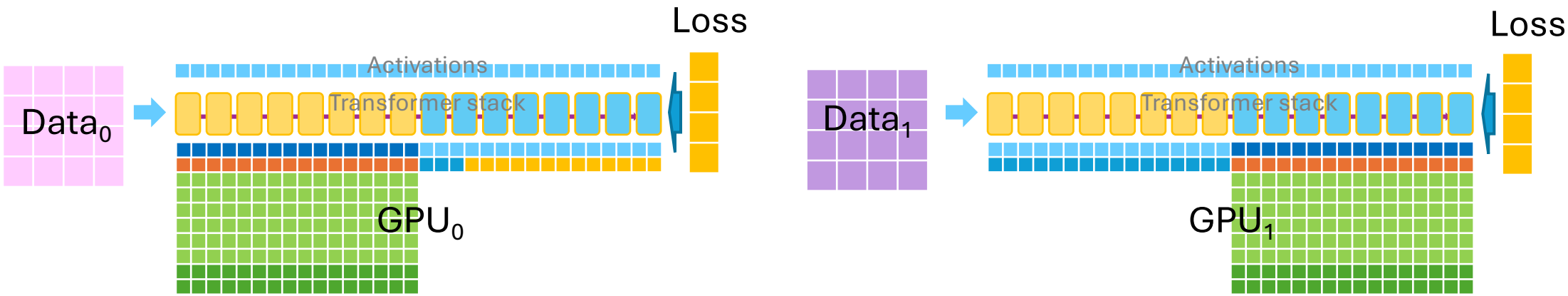
ZeRO Stage 1: Partitioning Optimizer States



- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average
- Update the FP32 weights with ADAM optimizer
- Update the FP16 weights

Adapted from Minjia Zhang, DeepSpeed Presentation

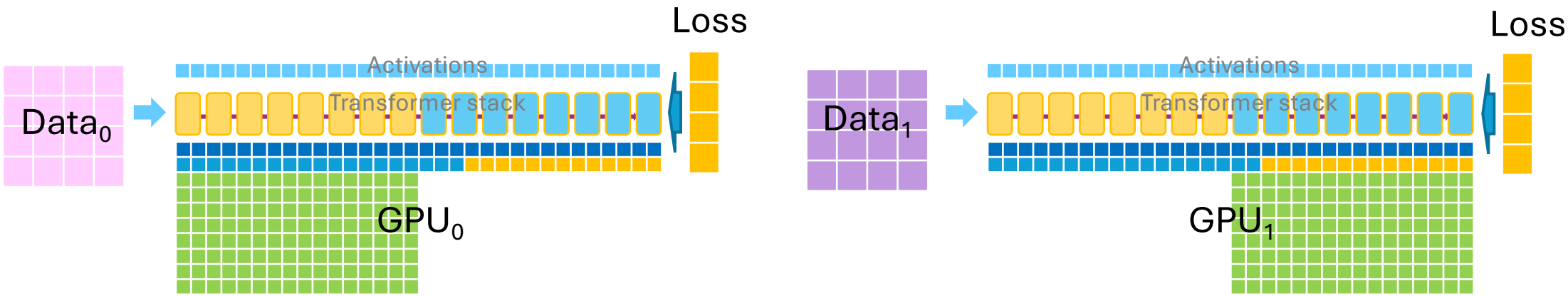
ZeRO Stage 1: Partitioning Optimizer States



- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average
- Update the FP32 weights with ADAM optimizer
- Update the FP16 weights
- All Gather the FP16 weights to complete the iteration

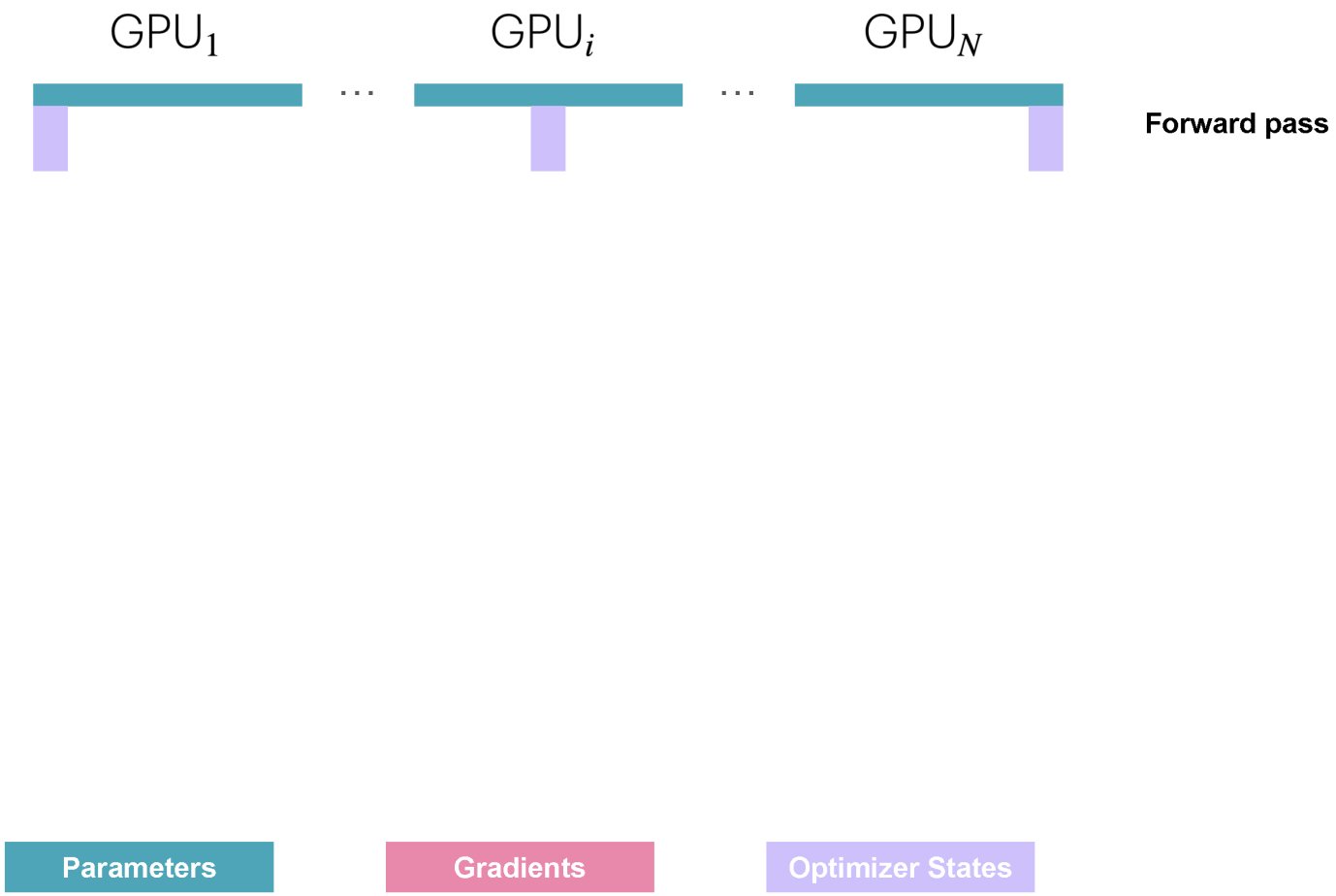
Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States



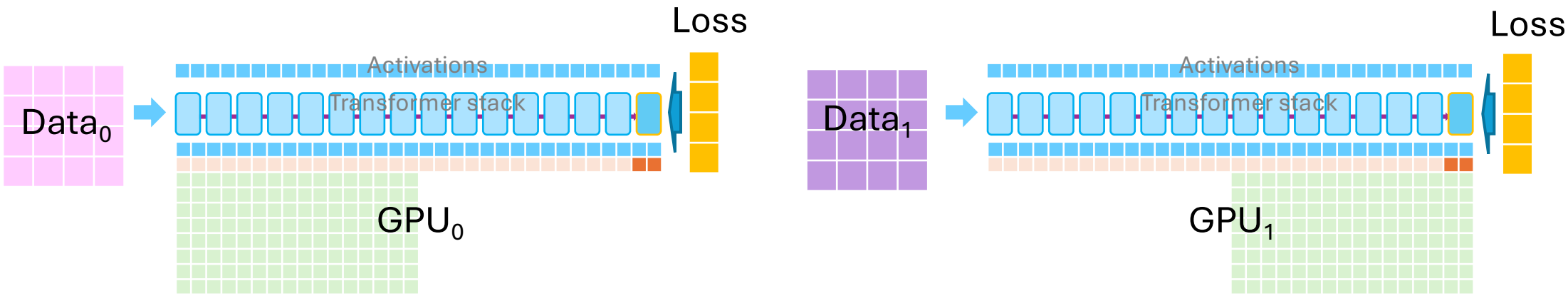
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and ReduceScatter to average
- Update the FP32 weights with ADAM optimizer
- Update the FP16 weights
- All Gather the FP16 weights to complete the iteration

ZeRO Stage 1: Partitioning Optimizer States



https://nanotron-ultrascale-playbook.static.hf.space/index.html#memory_for_activations

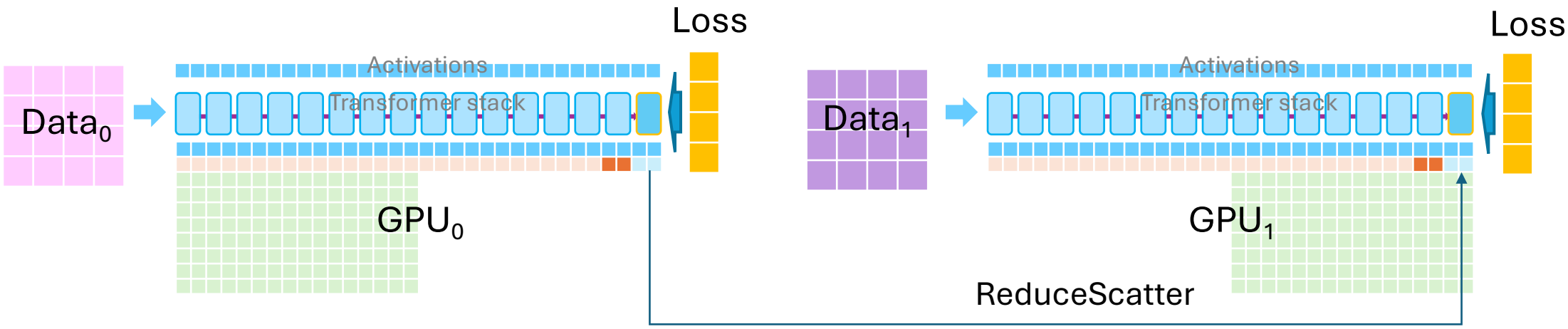
ZeRO Stage 2: Partitioning Gradients



- Partitioning gradients across GPUs
- The forward process remains the same as stage 1

Slides from Tianqi Chen and Zhihao Jia, CMU

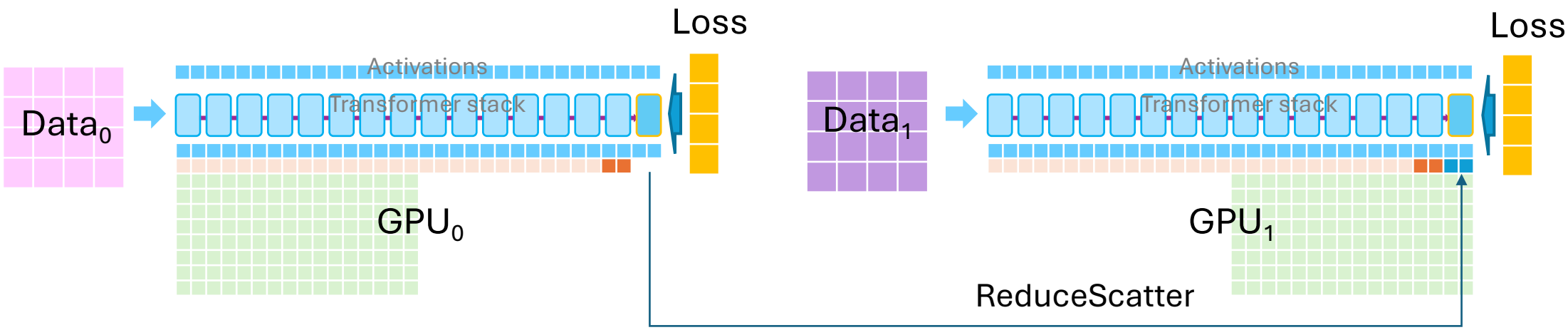
ZeRO Stage 2: Partitioning Gradients



- Partitioning gradients across GPUs
- Perform ReduceScatter right after back propagation of each layer

Slides from Tianqi Chen and Zhihao Jia, CMU

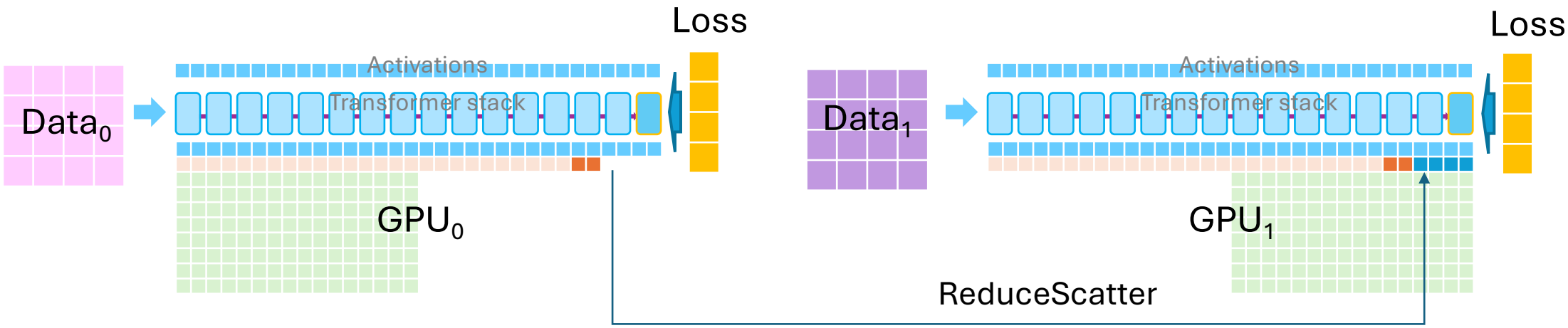
ZeRO Stage 2: Partitioning Gradients



- Partitioning gradients across GPUs
- Only one GPU keeps the gradients after ReduceScatter

Slides from Tianqi Chen and Zhihao Jia, CMU

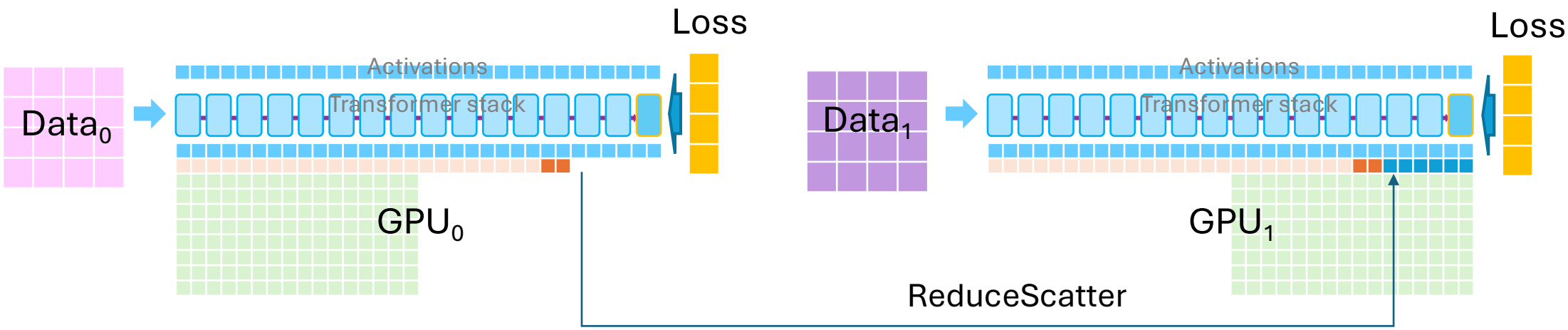
ZeRO Stage 2: Partitioning Gradients



- Partitioning gradients across GPUs
- Reduce gradients on GPUs responsible for updating parameters

Slides from Tianqi Chen and Zhihao Jia, CMU

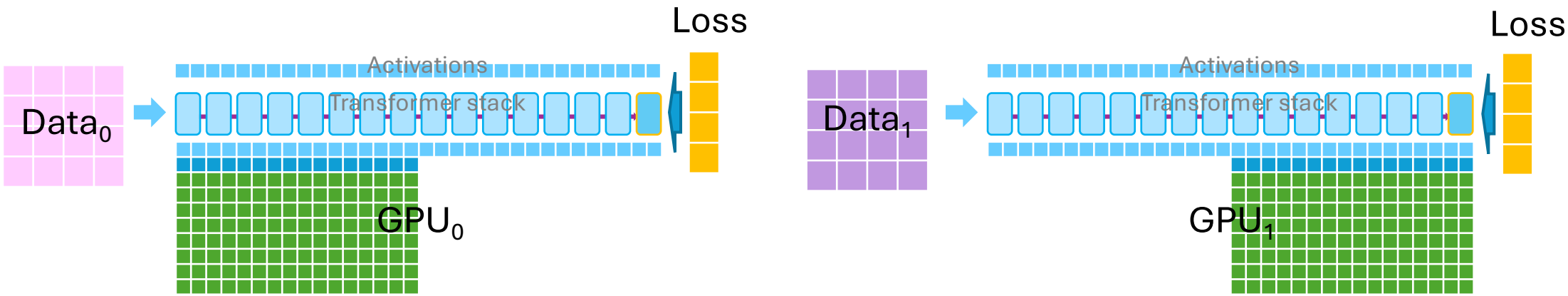
ZeRO Stage 2: Partitioning Gradients



- Partitioning gradients across GPUs
- Reduce gradients on GPUs responsible for updating parameters

Slides from Tianqi Chen and Zhihao Jia, CMU

ZeRO Stage 2: Partitioning Gradients

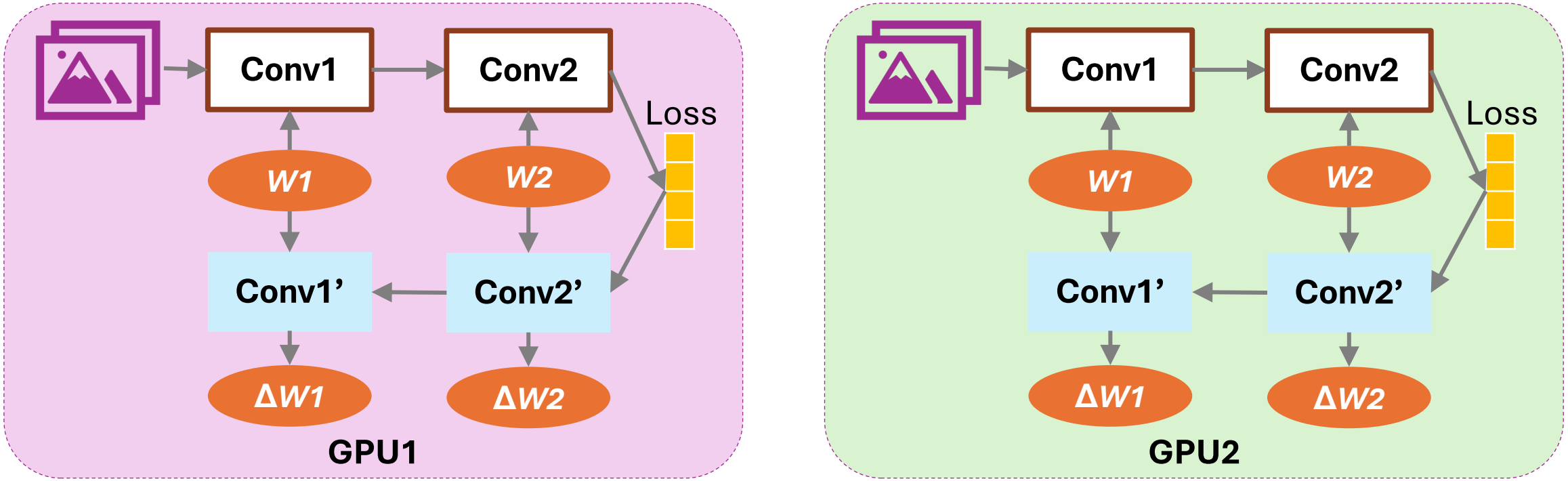


- Partitioning gradients across GPUs
- Reduce gradients on GPUs responsible for updating parameters

Slides from Tianqi Chen and Zhihao Jia, CMU

ZeRO Stage 3: Partitioning Parameters

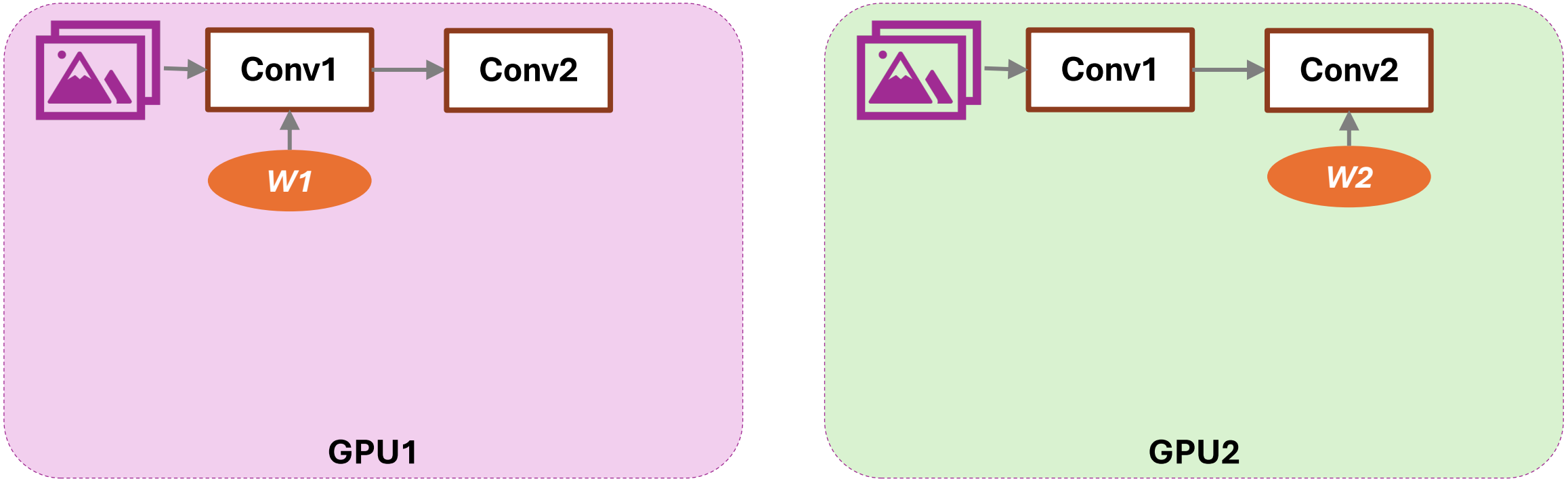
- In data parallel training, all GPUs keep **all** parameters during training



Slides from Tianqi Chen and Zhihao Jia, CMU

ZeRO Stage 3: Partitioning Parameters

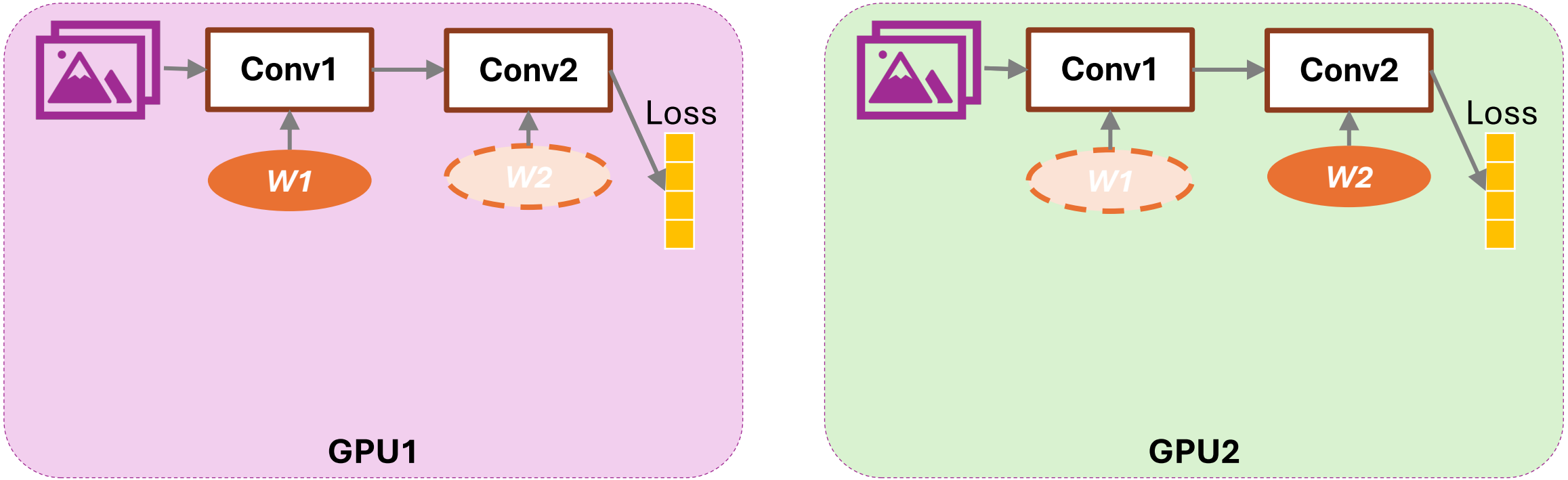
- In ZeRO, model parameters are partitioned across GPUs



Slides from Tianqi Chen and Zhihao Jia, CMU

ZeRO Stage 3: Partitioning Parameters

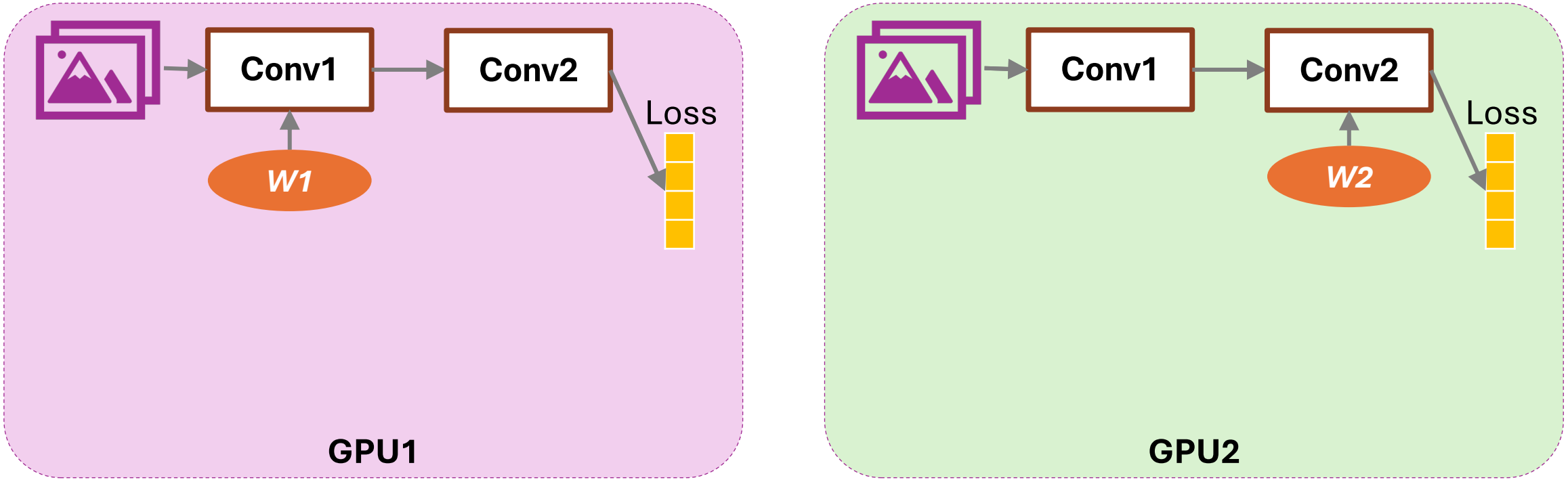
- In ZeRO, model parameters are partitioned across GPUs
- GPUs broadcast their parameters during forward



Slides from Tianqi Chen and Zhihao Jia, CMU

ZeRO Stage 3: Partitioning Parameters

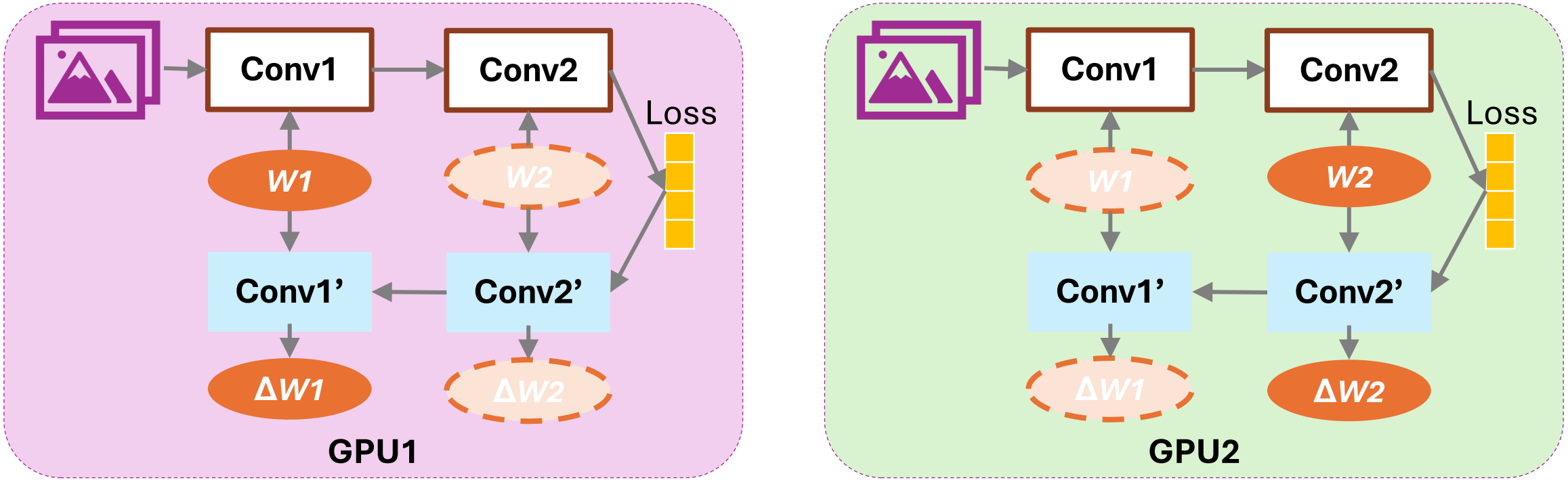
- In ZeRO, model parameters are partitioned across GPUs
- Parameters are discarded right after use



Slides from Tianqi Chen and Zhihao Jia, CMU

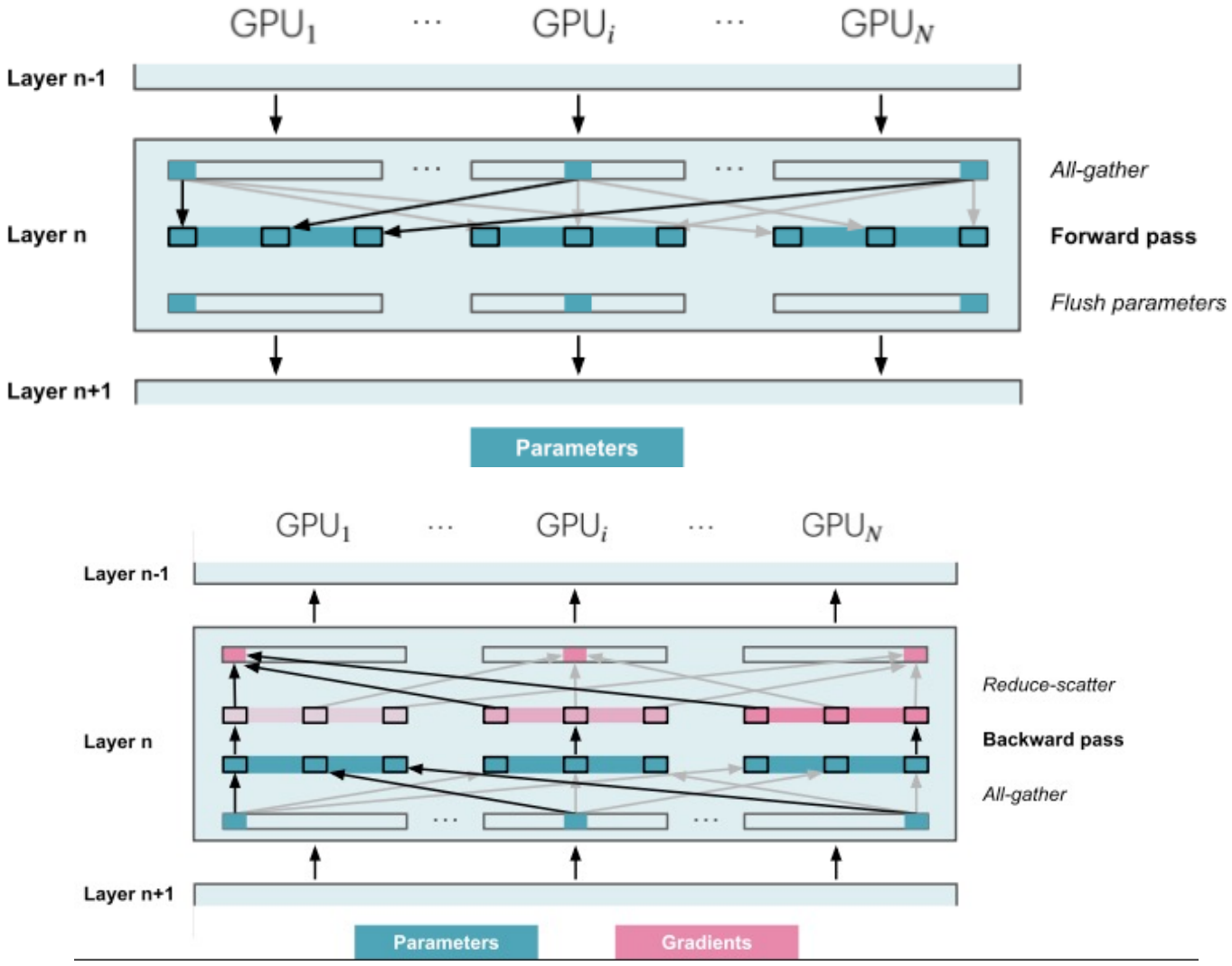
ZeRO Stage 3: Partitioning Parameters

- In ZeRO, model parameters are partitioned across GPUs
- GPUs broadcast their parameters again during backward



Slides from Tianqi Chen and Zhihao Jia, CMU

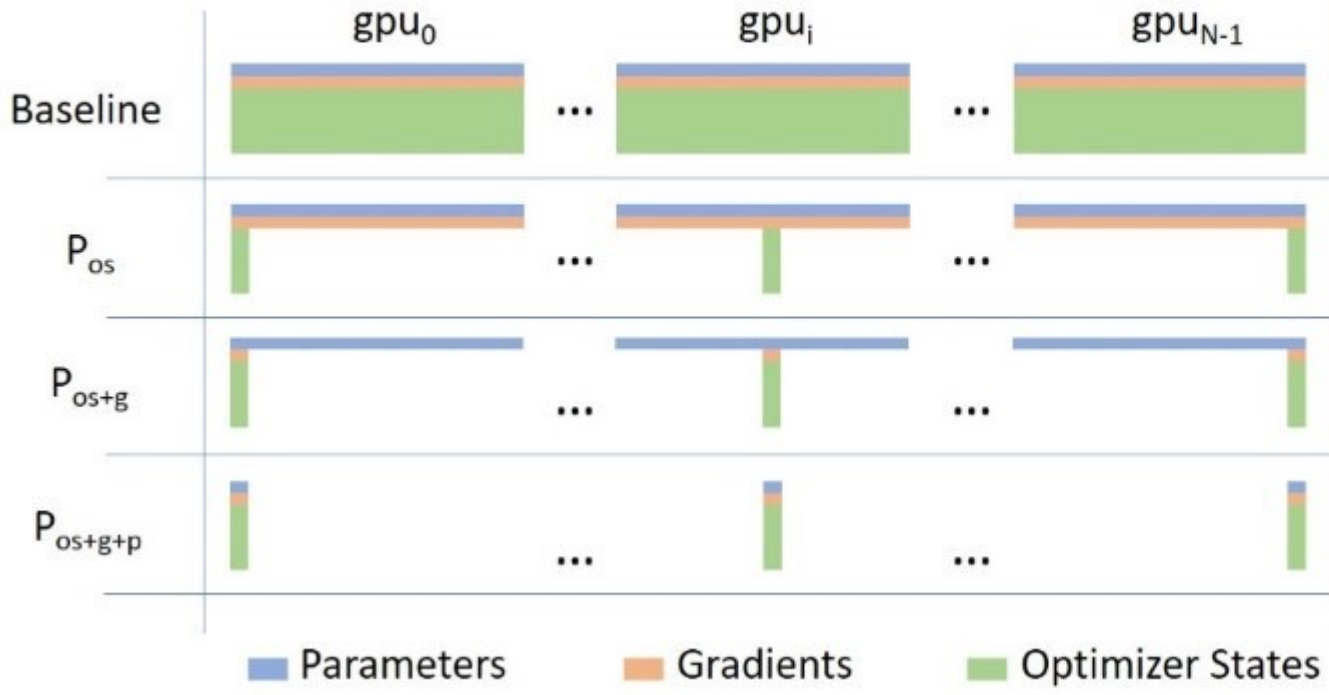
ZeRO Stage 3: Partitioning Parameters



https://huggingface.co/spaces/nanotron/ultrascale-playbook?section=zero-3:_adding_parameter_partitioning_(fsdp)

ZeRO: Zero Redundancy Optimizer

- ZeRO has three different stages
- Progressive memory savings and communication volume



	Memory Consumption		Comm Volume
	Formulation	Specific Example (K=12 Ψ=7.5B N_d=64)	
Baseline	$(2+2+K)*\Psi$	120GB	1x
Stage 1 (P_{os})	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
Stage 2 (P_{os+g})	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
Stage 3 (P_{os+g+p})	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x