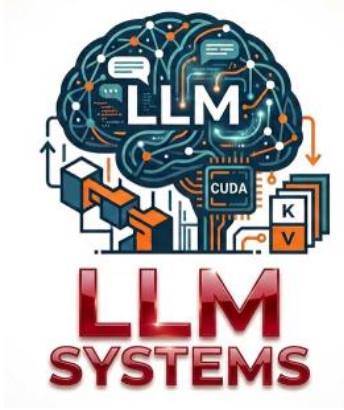




Memory Optimization in Distributed Training

Lei Li



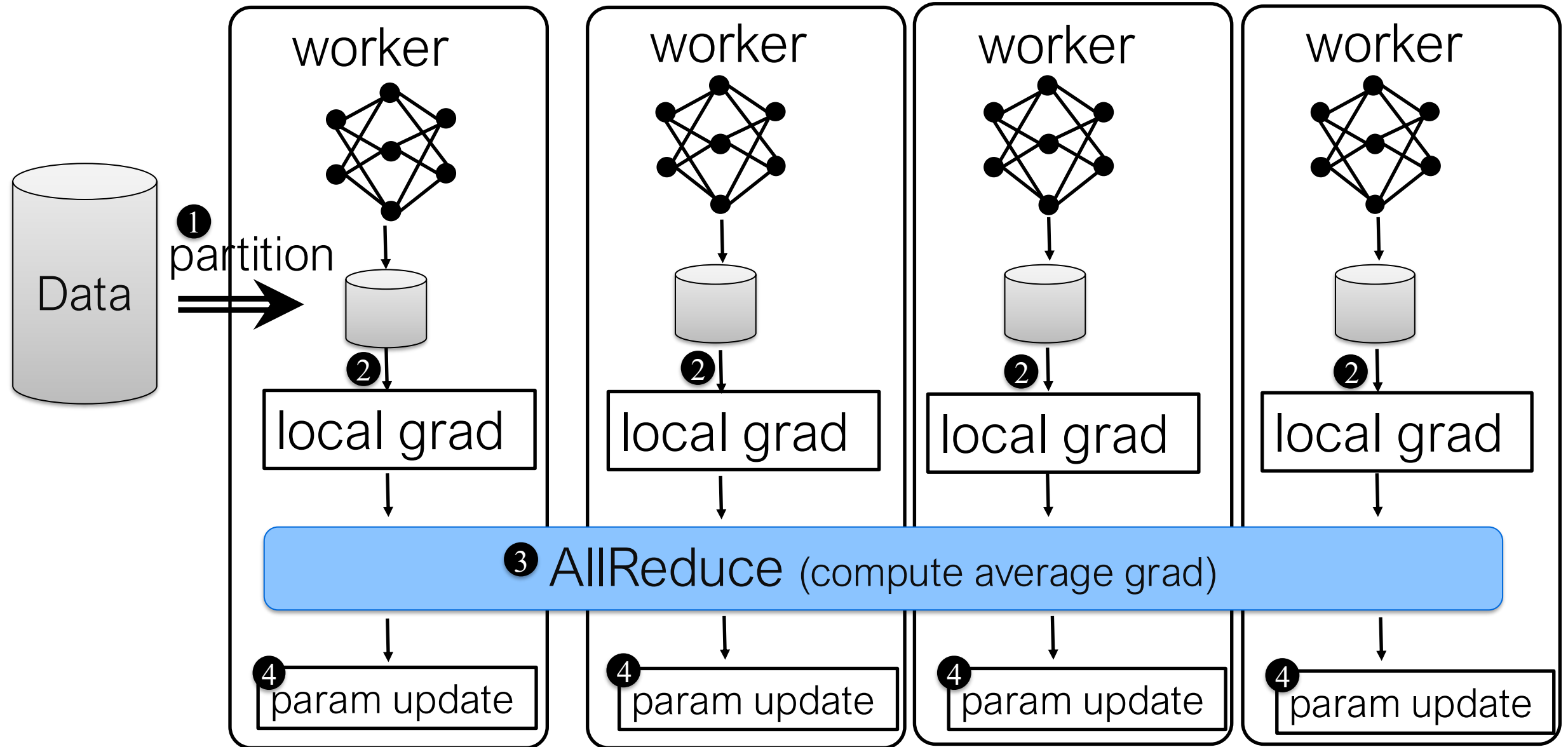
Language
Technologies
Institute

Carnegie Mellon University
School of Computer Science

Outline

- Memory Consumption for LLM training
- Partitioning and reducing the memory for data parallel training
 - Partition optimizer states
 - Partition gradients
 - Partition parameters
- Other memory optimizations

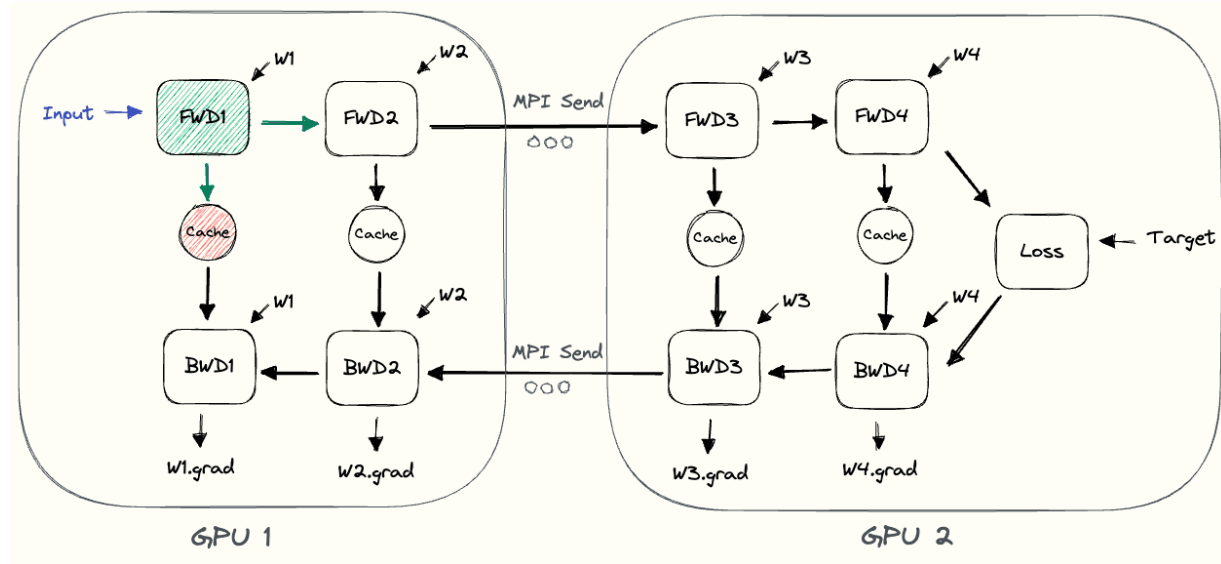
Distributed Data Parallel Training



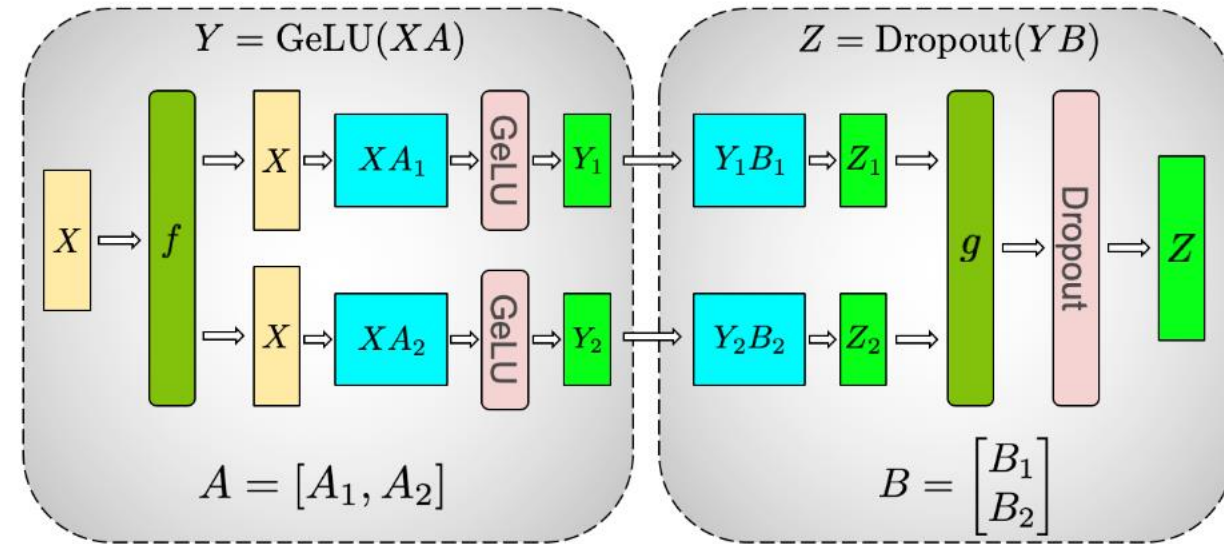
Model Parallel Training

Model Parallel: memory usage and computation of a model is distributed across multiple workers.

Pipeline Parallelism



Tensor parallelism



Comparing Data and Model Parallelism

	Pros	Cons
Model Parallelism	Good memory efficiency	Poor compute /communication efficiency (5% of peak perf in training 40B model with Megatron)
Data parallelism	Good compute/communication efficiency	Poor memory efficiency (Every device has one copy of model)

Memory Consumption in DDP (Mixed Precision)

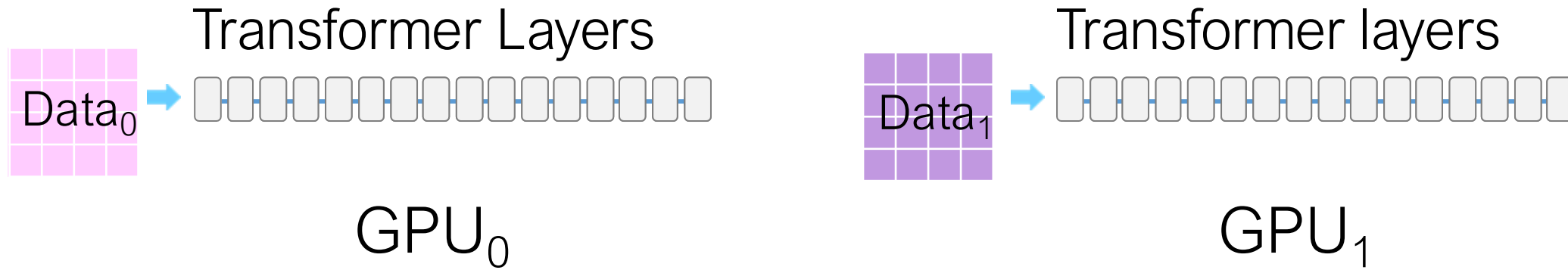
Each GPU needs to store (20N + act)

- Model parameters (N * 2 bytes)
- Forward activation for each layer (b * len * n_layer * d * 2bytes)
- Backward gradients (N * 2 bytes)
- Optimizer state (for Adam) N * 4 * 4
 - parameters, gradients, momentum, variance

Adam Optimizer

$$\begin{aligned}m_{t+1} &= \beta_1 m_t - (1 - \beta_1) \nabla \ell(x_t) \\v_{t+1} &= \beta_2 v_t + (1 - \beta_2) (\nabla \ell(x_t))^2 \\\hat{m}_{t+1} &= \frac{m_{t+1}}{1 - \beta_1^{t+1}} \\\hat{v}_{t+1} &= \frac{v_{t+1}}{1 - \beta_2^{t+1}} \\x_{t+1} &= x_t - \frac{\eta}{\sqrt{\hat{v}_{t+1}} + \epsilon} \hat{m}_{t+1}\end{aligned}$$

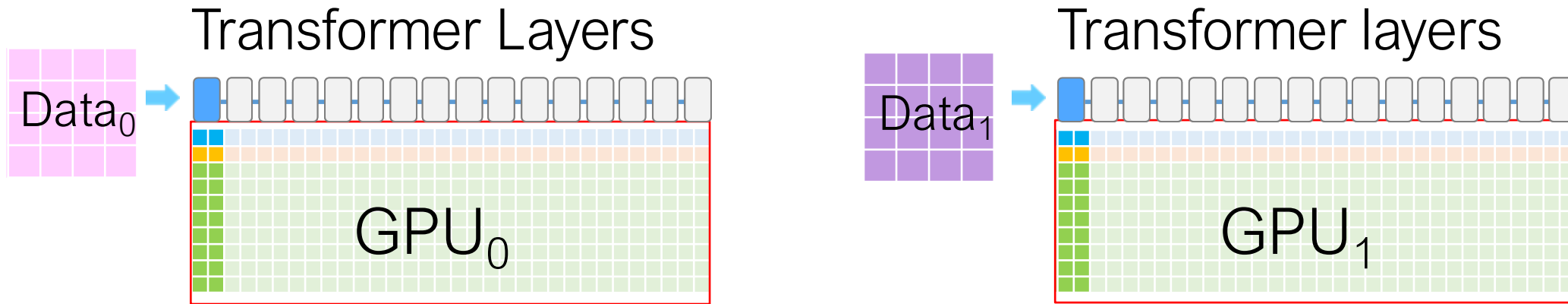
DDP Memory Consumption



Example: two data partitions on two GPUs

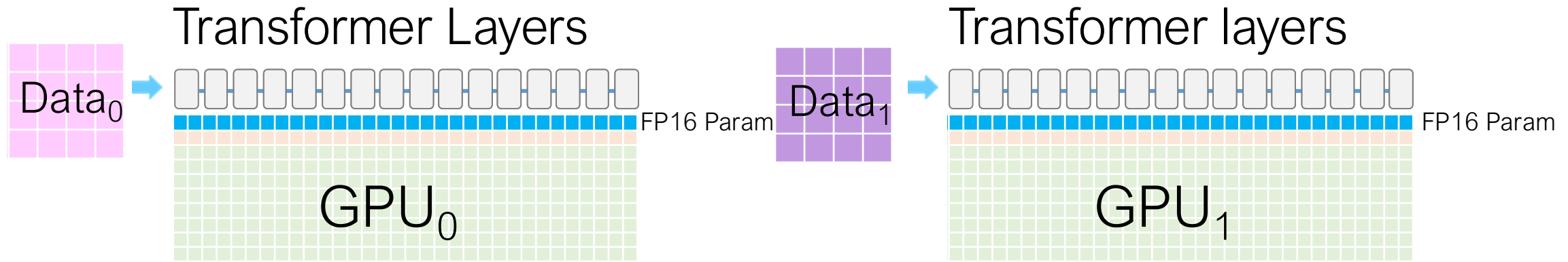
- 16 layer Transformer Model

DDP Memory Consumption



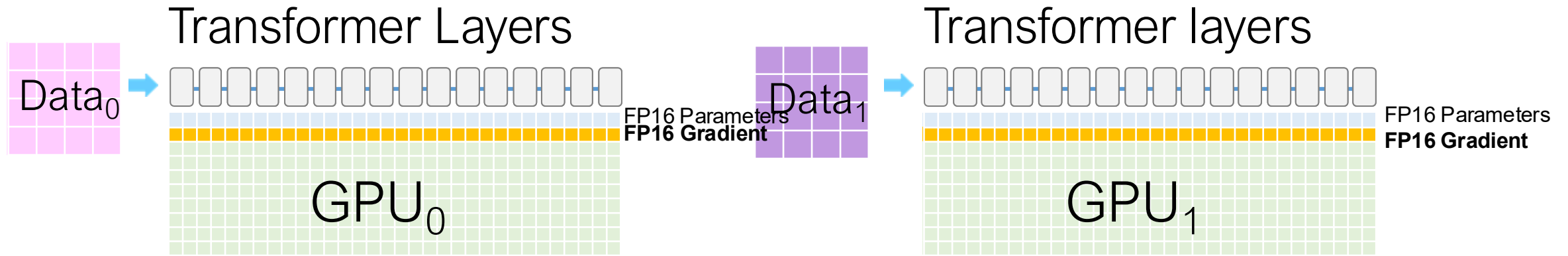
- Each cell represents GPU memory used by the corresponding transformer layer

DDP Memory Consumption



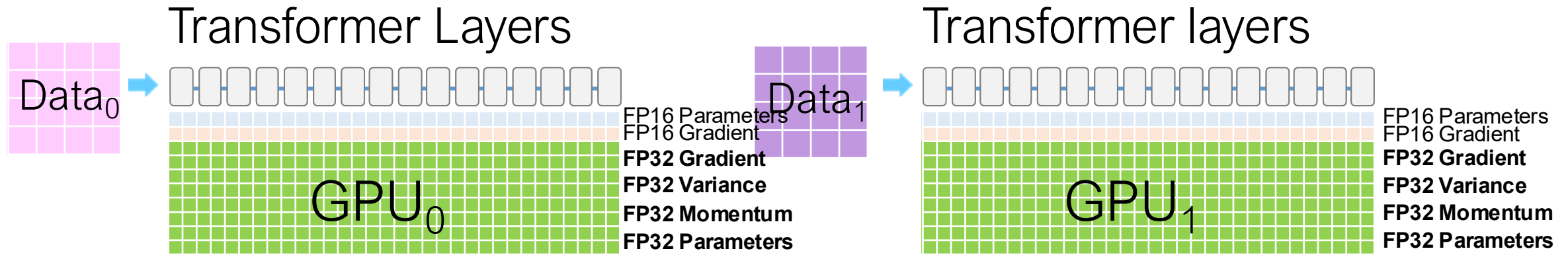
- Each cell represents GPU memory used by the corresponding transformer layer
- FP16 parameters, FP16 Gradients, FP32 Optimizer States (Gradients, Variance, Momentum, Parameters)

DDP Memory Consumption



- Each cell represents GPU memory used by the corresponding transformer layer
- FP16 parameters, FP16 Gradients, FP32 Optimizer States (Gradients, Variance, Momentum, Parameters)

DDP Memory Consumption



- Each cell represents GPU memory used by the corresponding transformer layer
- FP16 parameters, FP16 Gradients, FP32 Optimizer States (Gradients, Variance, Momentum, Parameters)

DDP Memory Usage

LLaMA-3 8B

- Parameters: 16GB
- Gradients: 16GB
- Optimizer states: 128GB
- Total: 160GB

GPT-3 175B

- Parameters: 350GB
- Gradients: 350GB
- Optimizer states: 2800GB
- Total: 3500GB

Other Memory Usages in DDP

- Temporary Buffers:
 - Storing intermediate results. Operations such as gradient all-reduce, or gradient norm computation tend to fuse all the gradients into a single flattened buffer before applying the operation in an effort to improve throughput.
- Memory Fragmentation:
 - In extreme cases can be 30%.

Goal: Reduce Memory Usage
in DDP → Training Extremely
Large Models

Outline

- Memory Consumption for LLM training
- • Partitioning and reducing the memory for data parallel training
 - Partition optimizer states
 - Partition gradients
 - Partition parameters
- Other memory optimization

Common Approaches to Reduce Memory

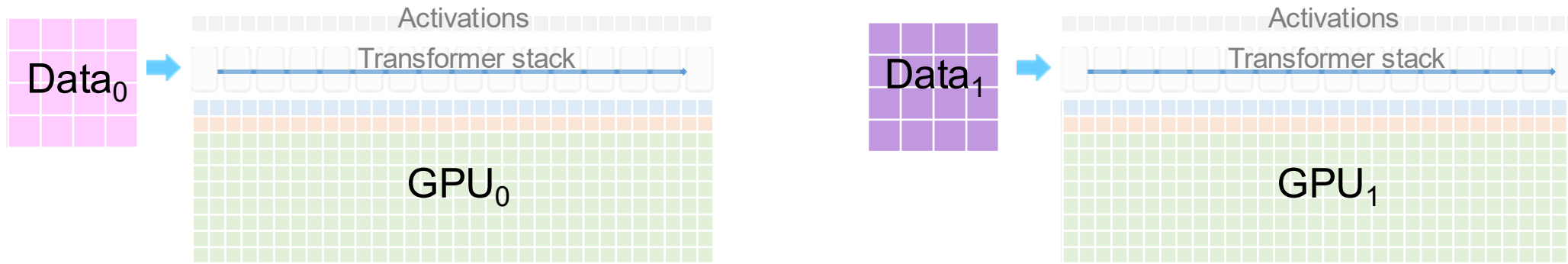
- ZeRO: Partition Optimizer States, Gradients, Parameters
- Reducing Activation Memory
 - Activation Checkpoint, Compression
- CPU Offload
 - Requires CPU-GPU-CPU transfer, which can take 50% time
- Memory Efficient Optimizer
 - Maintaining coarser-grained statistics of model parameters and gradients

ZeRO - Zero Redundancy Optimizer

- Key idea:
 - Eliminating data redundancy in DDP by partitioning the optimizer states (zero-1), gradients (zero-2), parameters (zero-3)
- Widely used for large language model training.
 - 7B model memory: 120GB → 30GB (with 4GPUs)
- Implemented in Deepspeed.

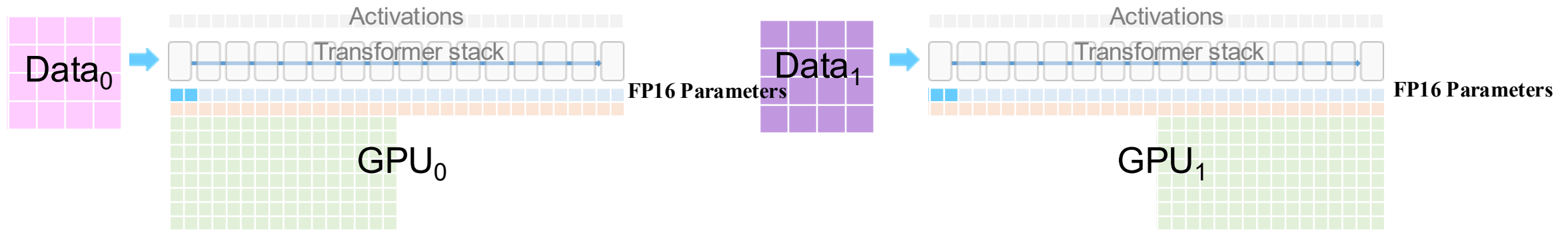
ZeRO Stage 1: Partitioning Optimizer States

K GPUs (=2 in example)



ZeRO Stage 1: Partitioning Optimizer States

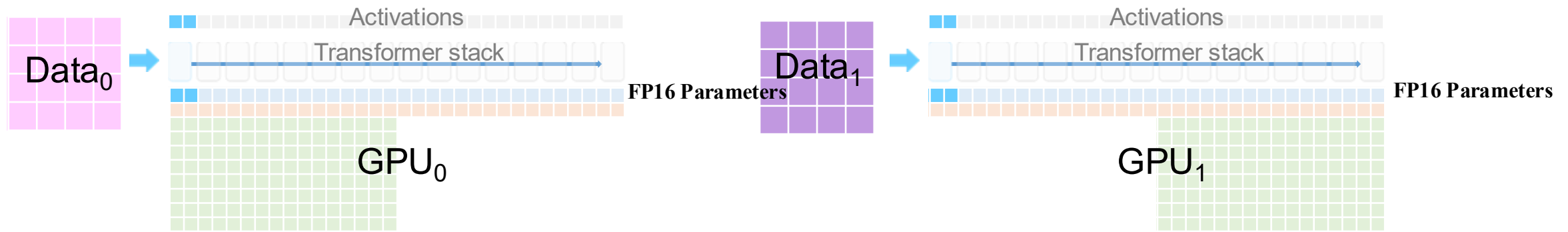
K GPUs (=2 in example)



- Partition the optimizer states to K parts, each GPU process one partition
- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

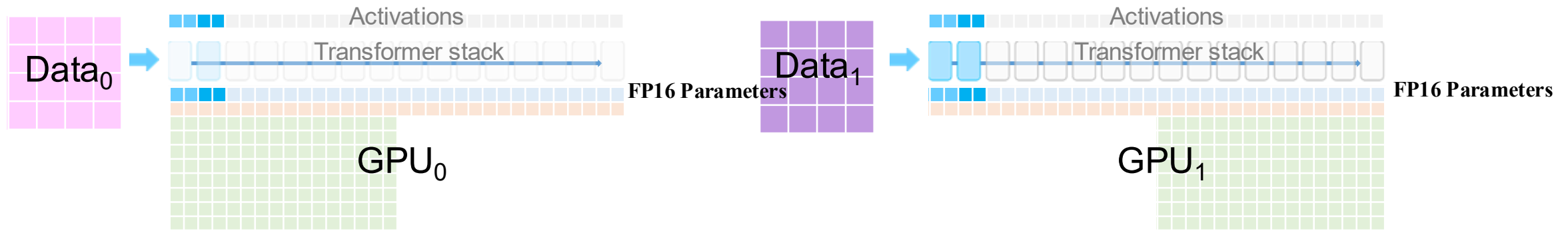
K GPUs (=2 in example)



- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

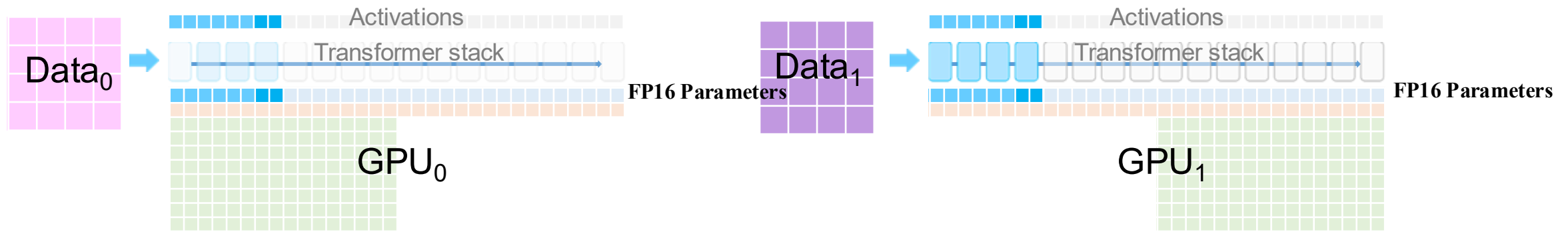
K GPUs (=2 in example)



- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

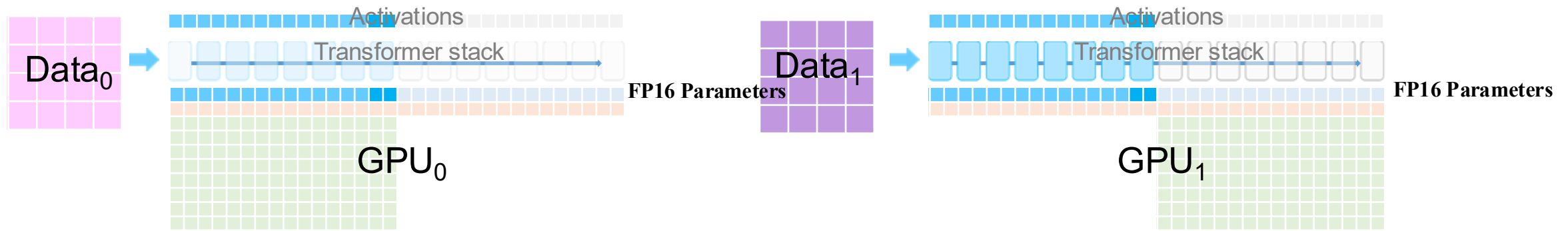
K GPUs (=2 in example)



- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

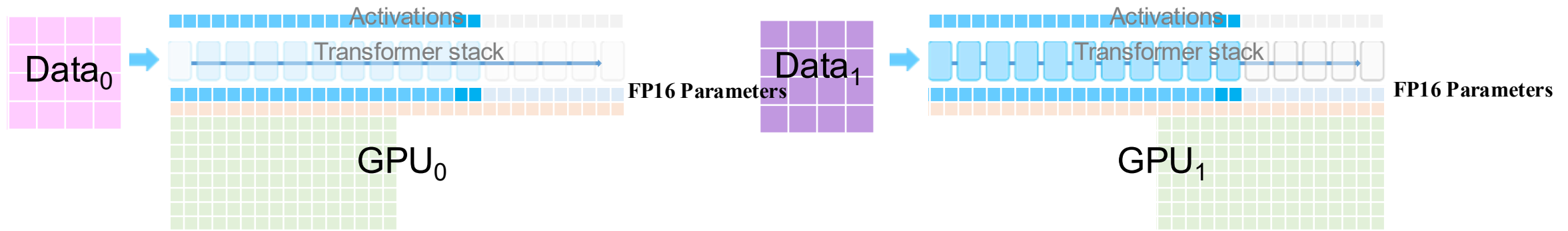
K GPUs (=2 in example)



- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

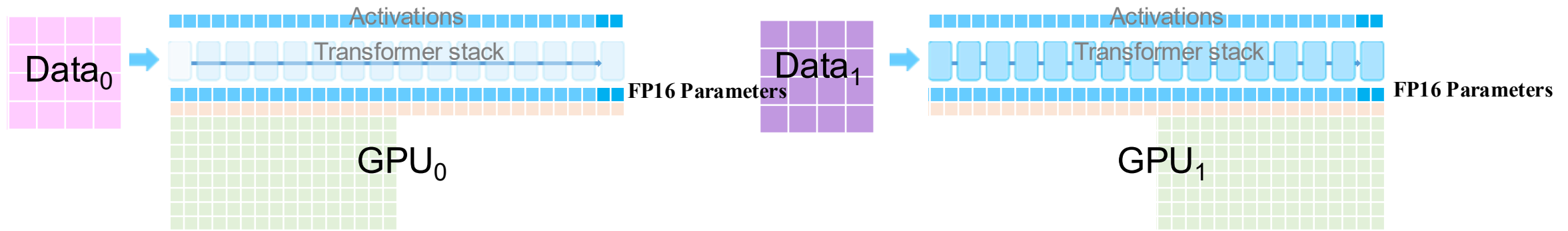
K GPUs (=2 in example)



- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

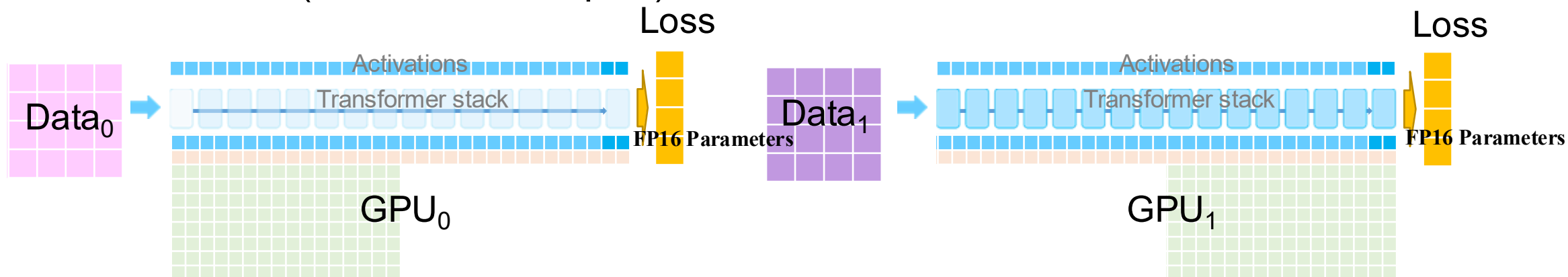
K GPUs (=2 in example)



- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

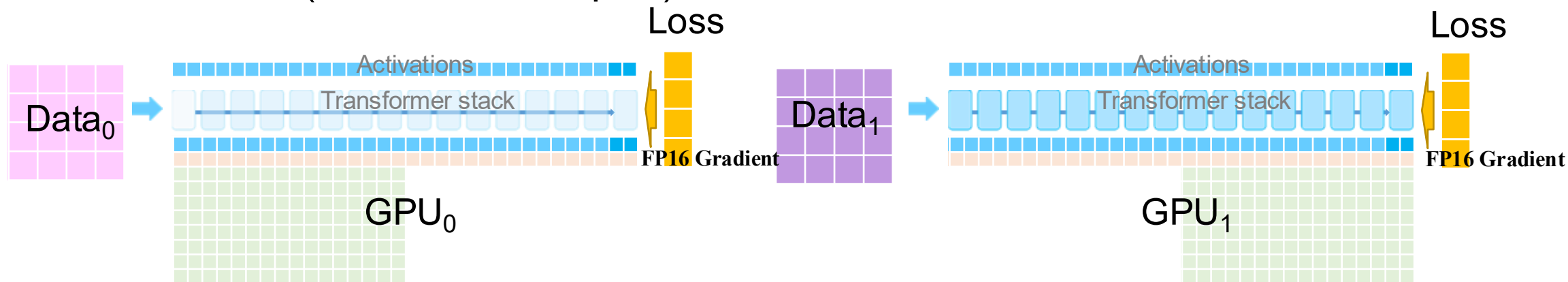
K GPUs (=2 in example)



- forward pass to produce activations and loss (by fp16 parameters)

ZeRO Stage 1: Partitioning Optimizer States

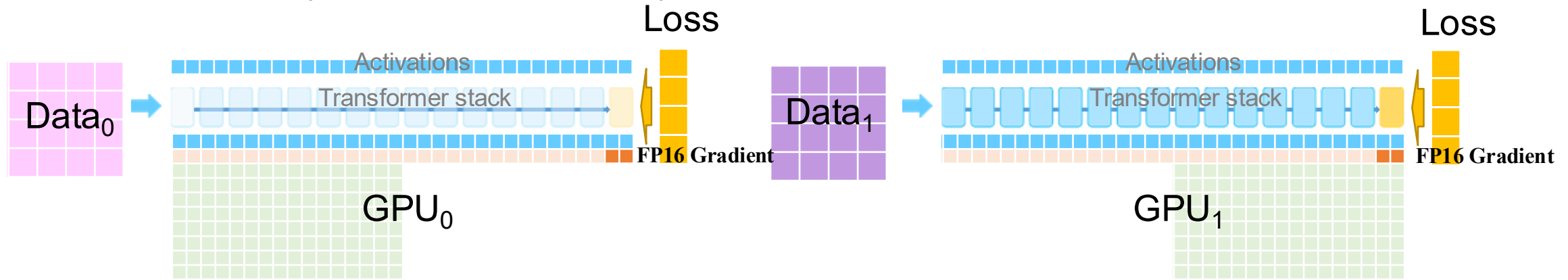
K GPUs (=2 in example)



- backward from loss to calculate fp16 local gradients

ZeRO Stage 1: Partitioning Optimizer States

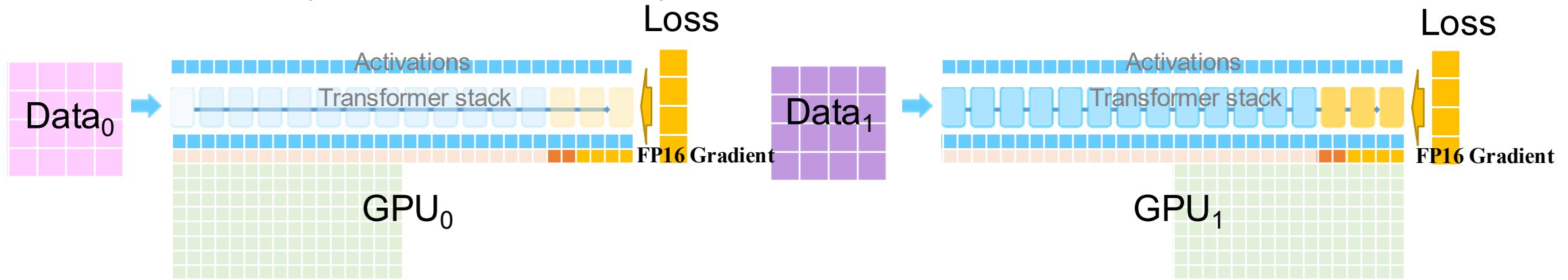
K GPUs (=2 in example)



- backward from loss to calculate fp16 local gradients

ZeRO Stage 1: Partitioning Optimizer States

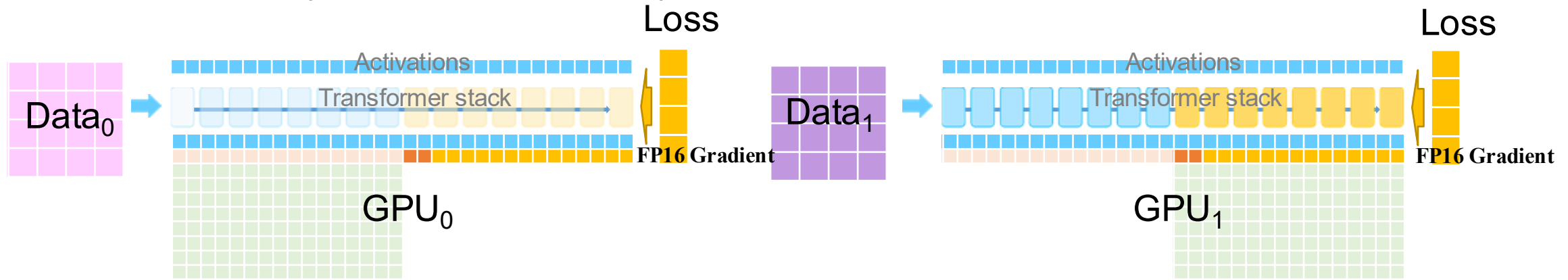
K GPUs (=2 in example)



- backward from loss to calculate fp16 local gradients

ZeRO Stage 1: Partitioning Optimizer States

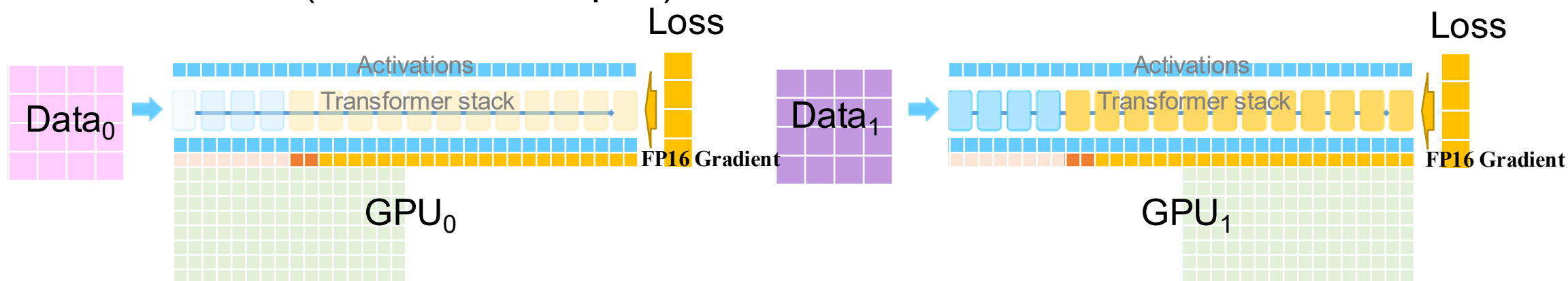
K GPUs (=2 in example)



- backward from loss to calculate fp16 local gradients

ZeRO Stage 1: Partitioning Optimizer States

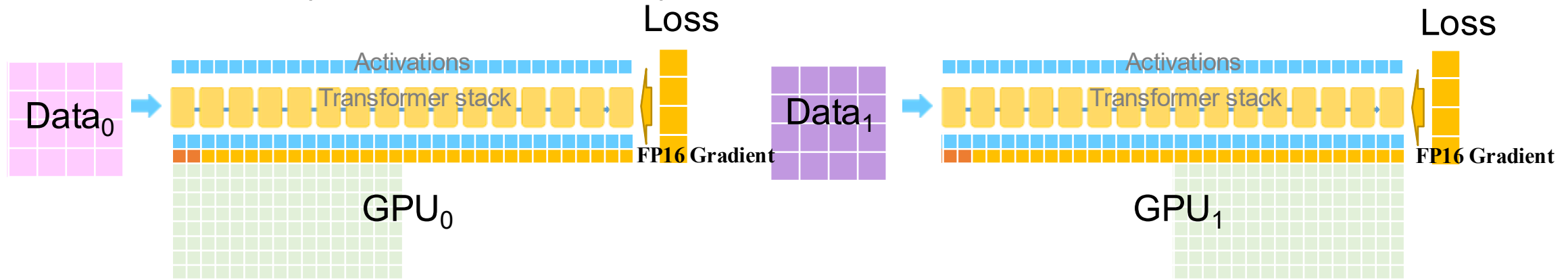
K GPUs (=2 in example)



- backward from loss to calculate fp16 local gradients

ZeRO Stage 1: Partitioning Optimizer States

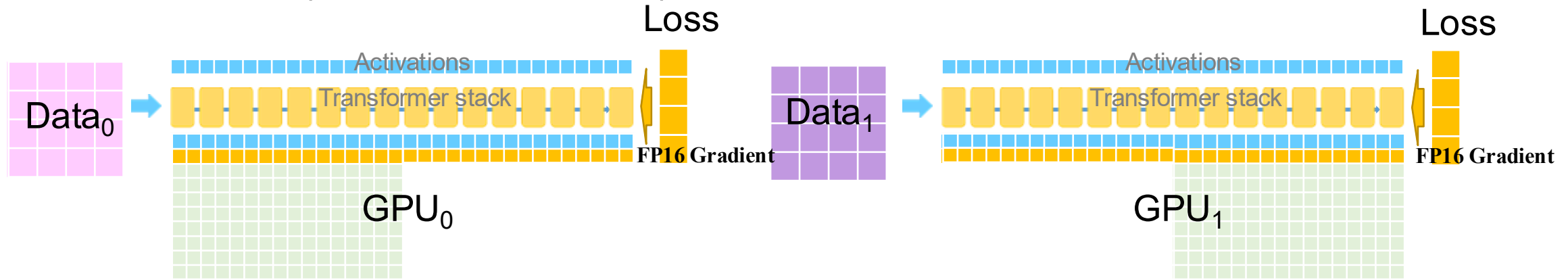
K GPUs (=2 in example)



- backward from loss to calculate fp16 local gradients

ZeRO Stage 1: Partitioning Optimizer States

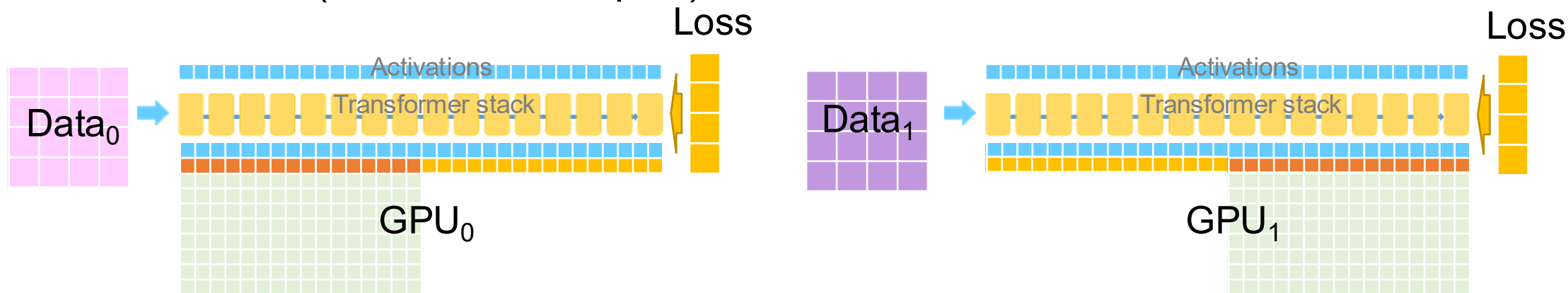
K GPUs (=2 in example)



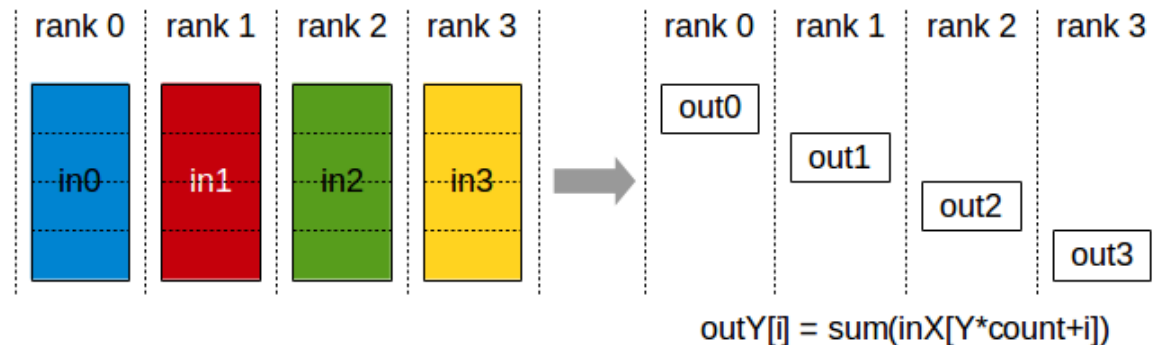
- backward from loss to calculate fp16 local gradients

ZeRO Stage 1: Partitioning Optimizer States

K GPUs (=2 in example)

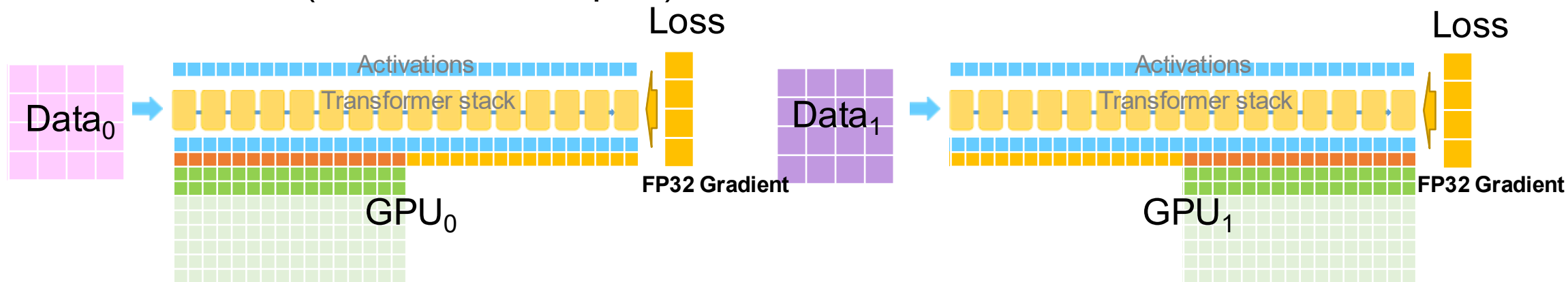


- gradient gathering from another GPU and average gradient calculation → global gradient
- ncclReduceScatter



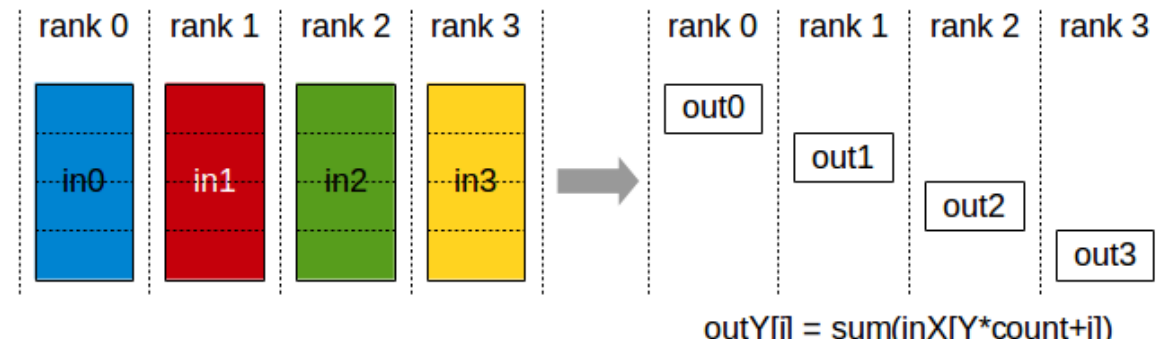
ZeRO Stage 1: Partitioning Optimizer States

K GPUs (=2 in example)



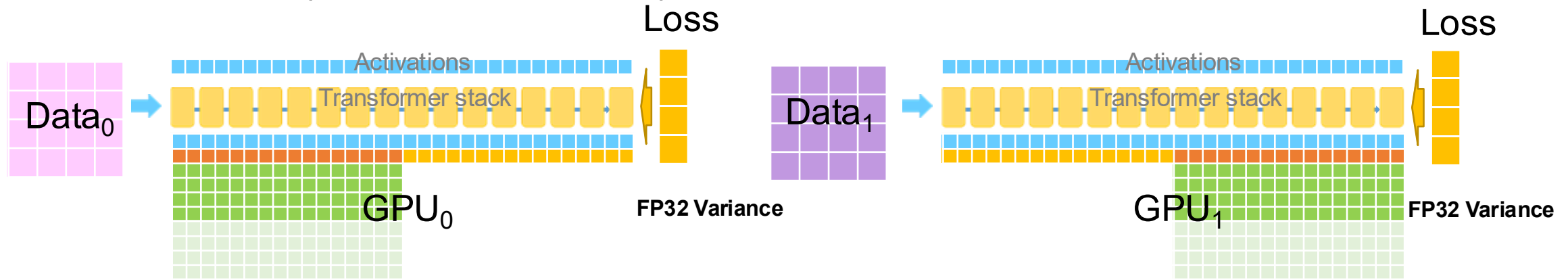
- gradient gathering from another GPU and average gradient calculation → FP32 global gradient

- ncclReduceScatter



ZeRO Stage 1: Partitioning Optimizer States

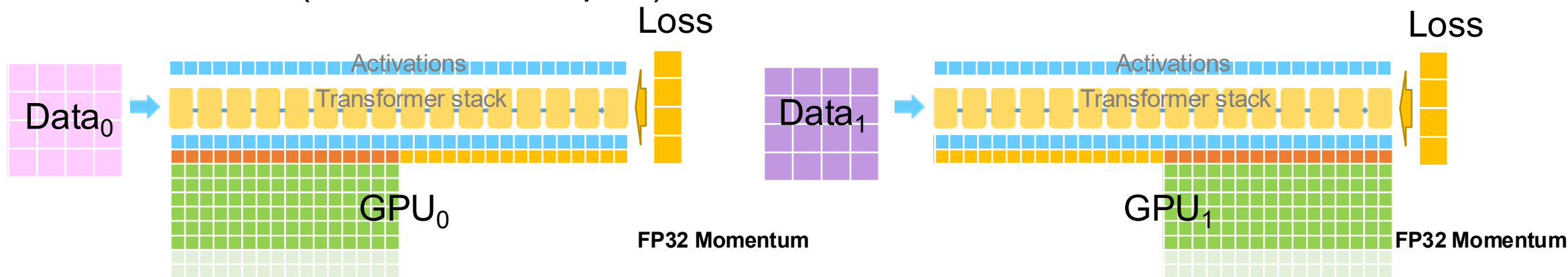
K GPUs (=2 in example)



- fp32 variance update

ZeRO Stage 1: Partitioning Optimizer States

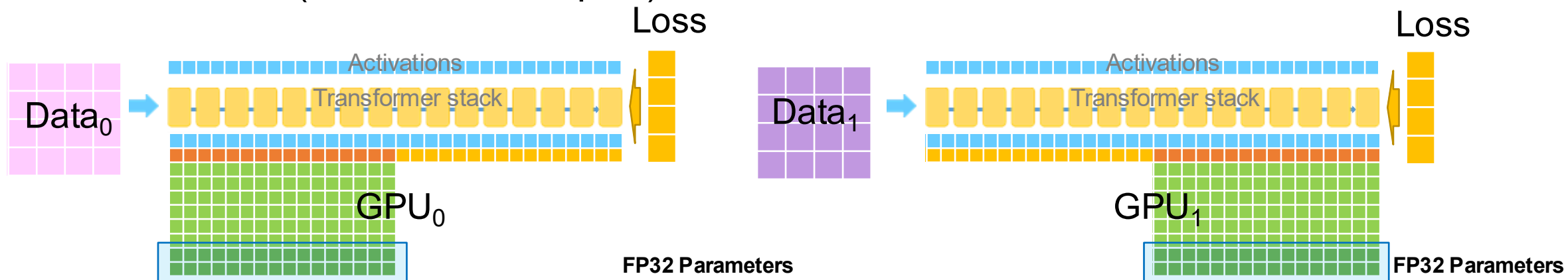
K GPUs (=2 in example)



- fp32 momentum update

ZeRO Stage 1: Partitioning Optimizer States

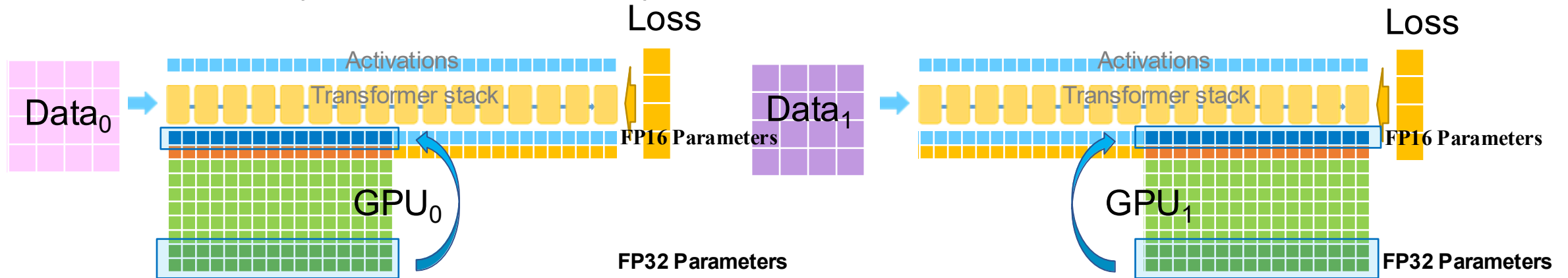
K GPUs (=2 in example)



- fp32 parameters update

ZeRO Stage 1: Partitioning Optimizer States

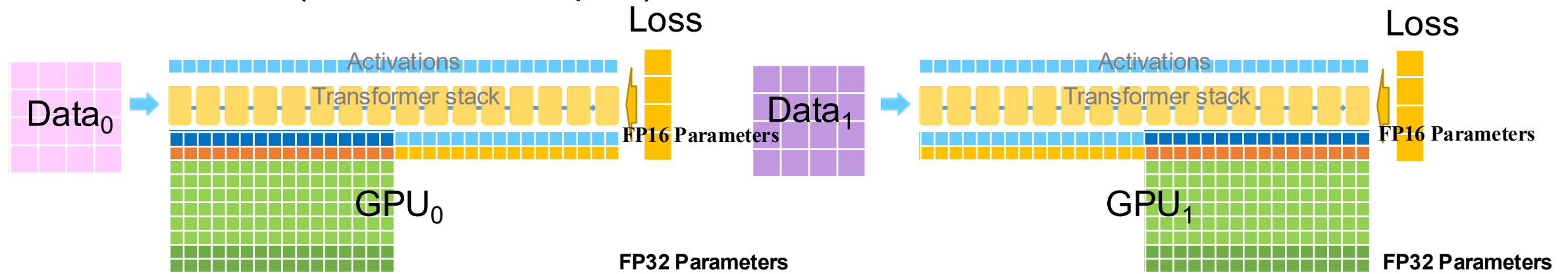
K GPUs (=2 in example)



- copy fp32 parameters to fp16 parameters

ZeRO Stage 1: Partitioning Optimizer States

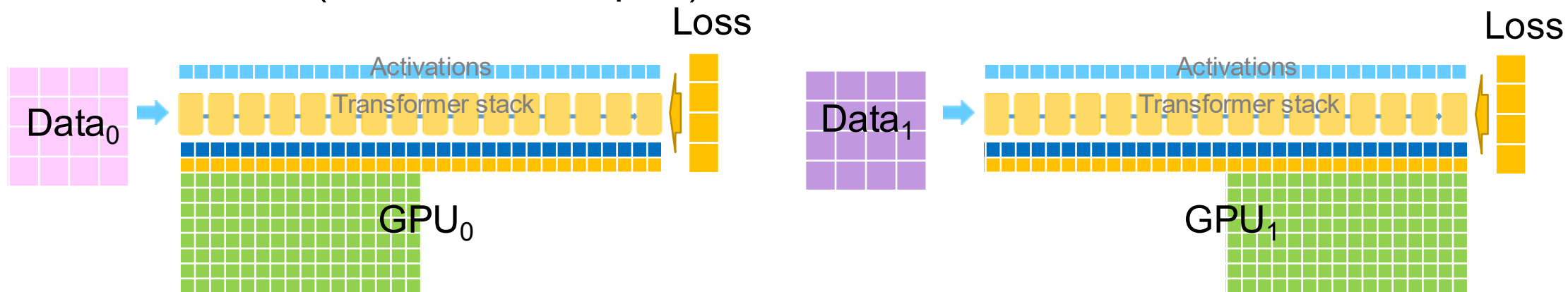
K GPUs (=2 in example)



- fp16 parameters ready

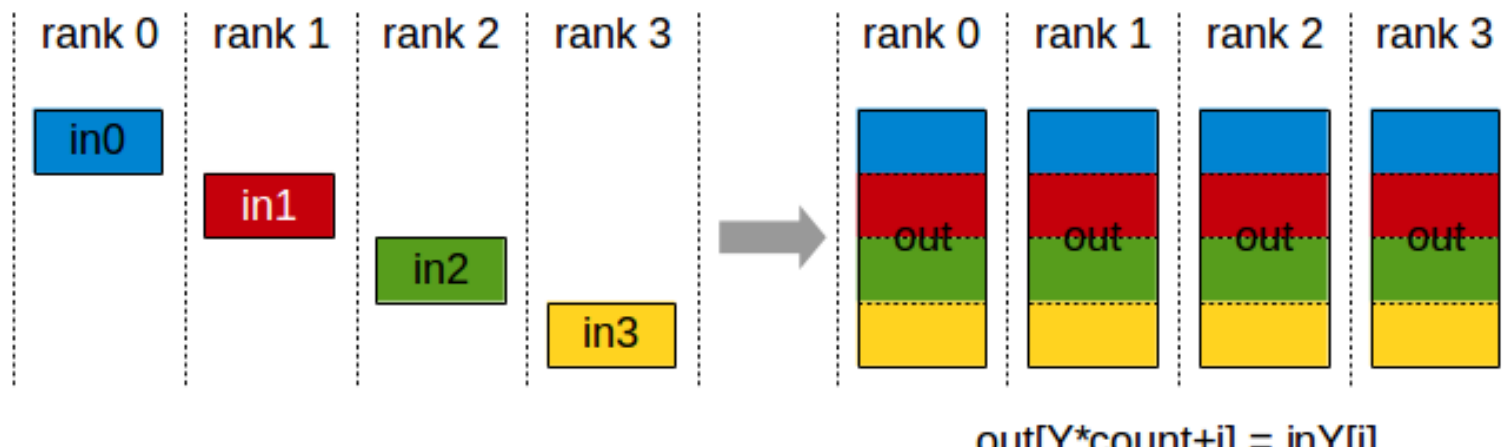
ZeRO Stage 1: Partitioning Optimizer States

K GPUs (=2 in example)



- AllGather the fp16 weights to complete the iteration

- ncclAllGather

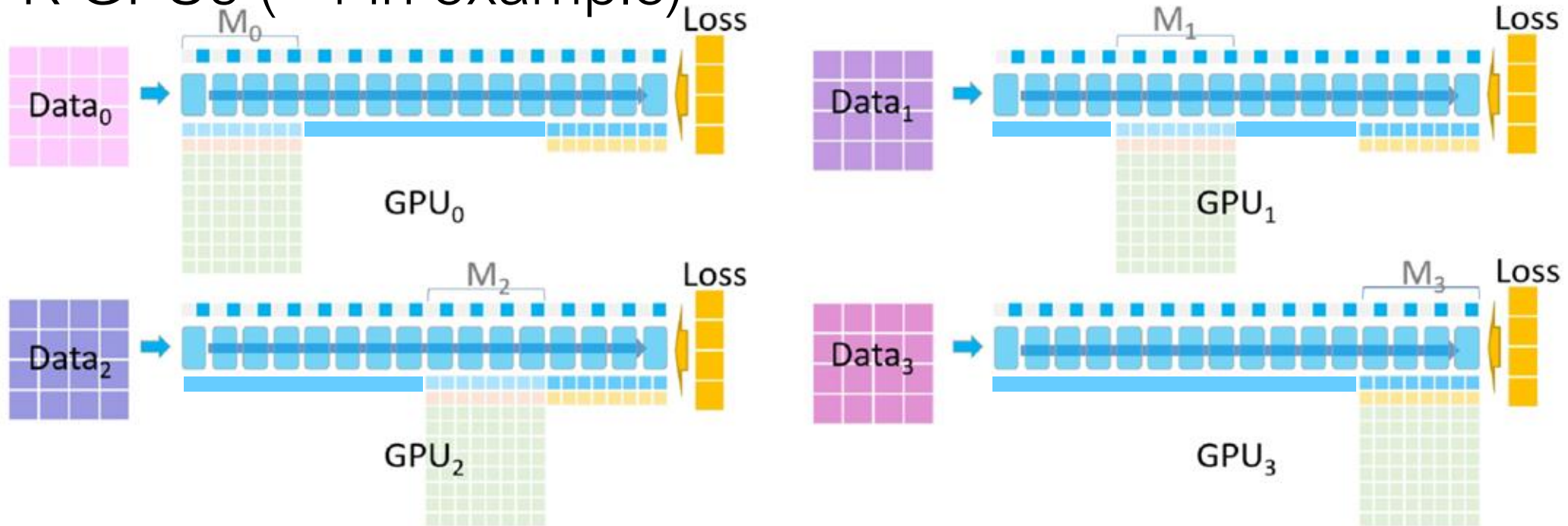


ZeRO Stage 2: Partition Gradients

- Key idea:
 - Each GPU compute all parameter gradients for its data partition
 - but only stores one partition of gradients instead of all gradients.
 - Passing the gradients out of its responsibility to the GPU responsible for those gradients.
- Memory for gradients reduced by K times. ($K = \#$ of GPUs)

ZeRO Stage 2: Partition Gradients

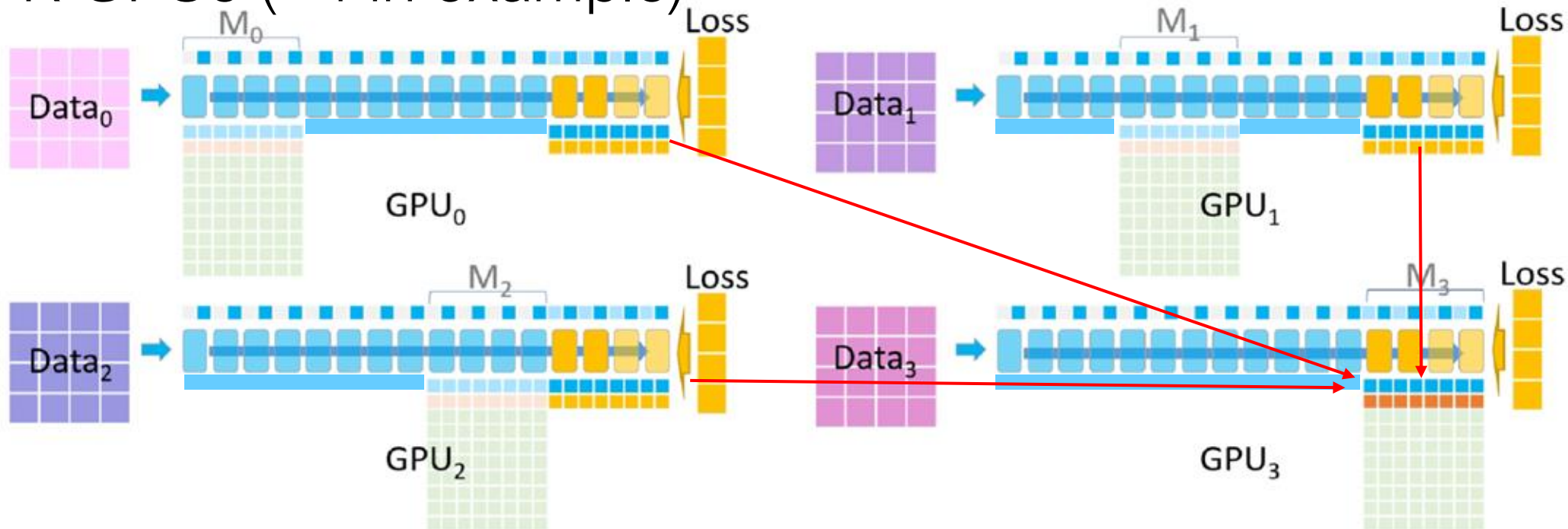
K GPUs (=4 in example)



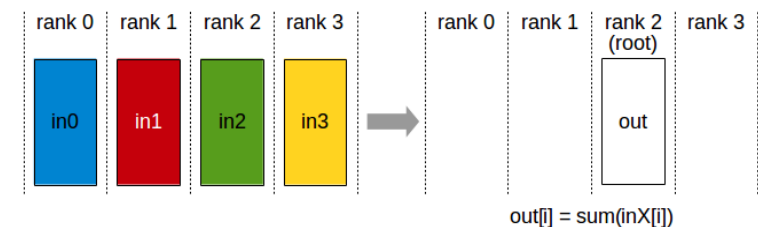
- In backward pass, GPU 0,1,2 hold temporary buffers for the gradients that GPU 3 is responsible for (M₃)

ZeRO Stage 2: Partition Gradients

K GPUs (=4 in example)

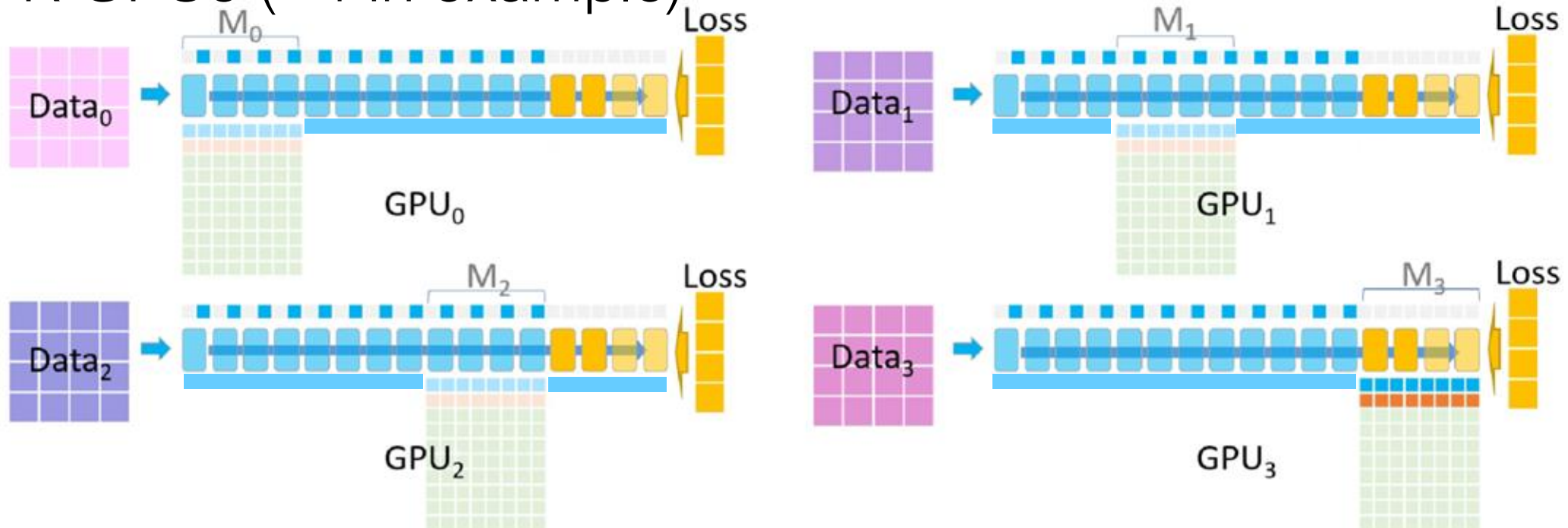


- GPU 0,1,2 pass the M3 gradients to GPU 3 (ncclReduce)



ZeRO Stage 2: Partition Gradients

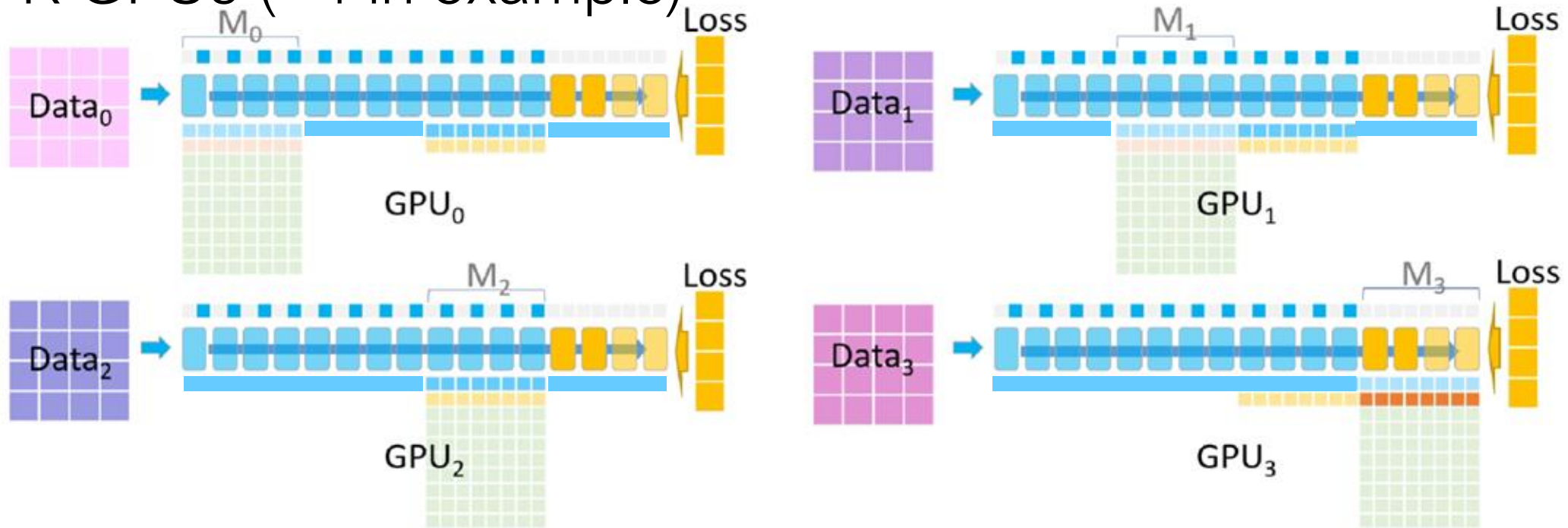
K GPUs (=4 in example)



Then GPU₀, GPU₁, GPU₂ delete M₃ gradients, GPU₃ keeps M₃ gradients.

ZeRO Stage 2: Partition Gradients

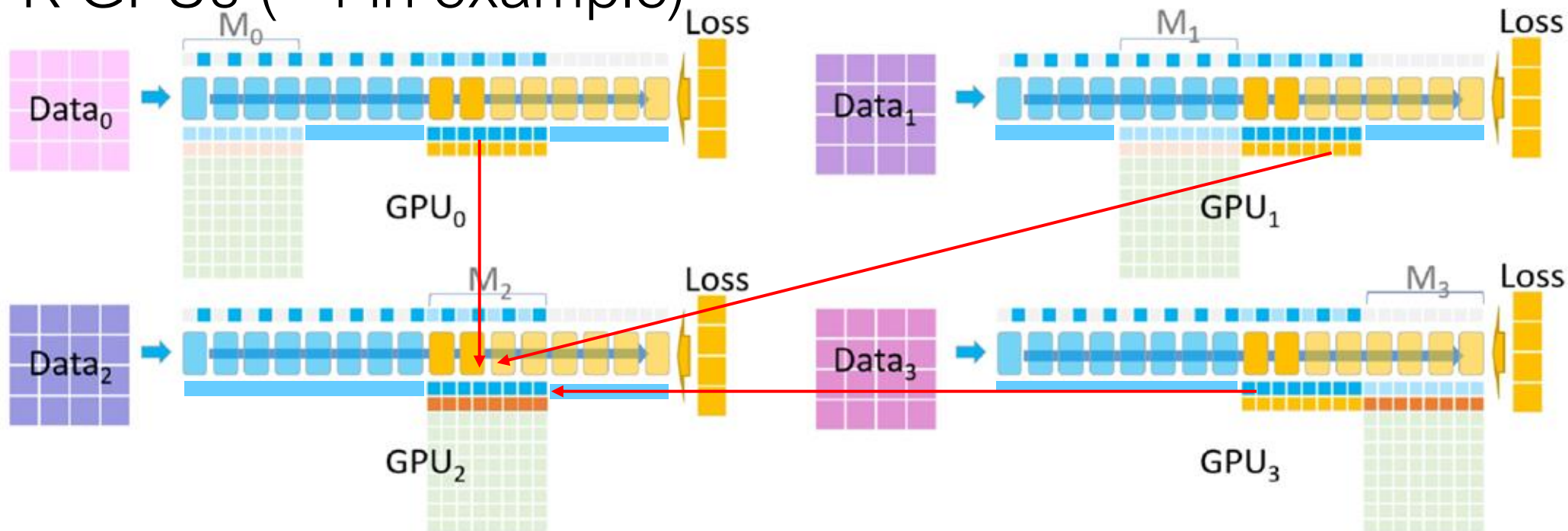
K GPUs (=4 in example)



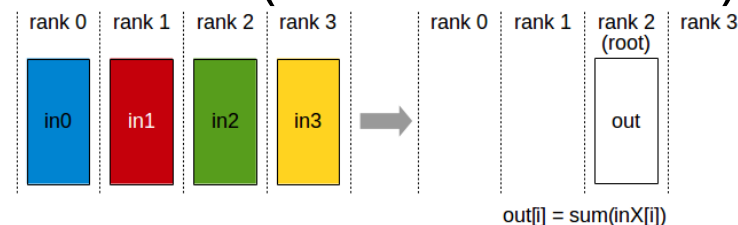
- Continue backward pass: GPU 0,1,3 hold temporary buffers for the gradients that GPU 2 is responsible for (M₂)

ZeRO Stage 2: Partition Gradients

K GPUs (=4 in example)

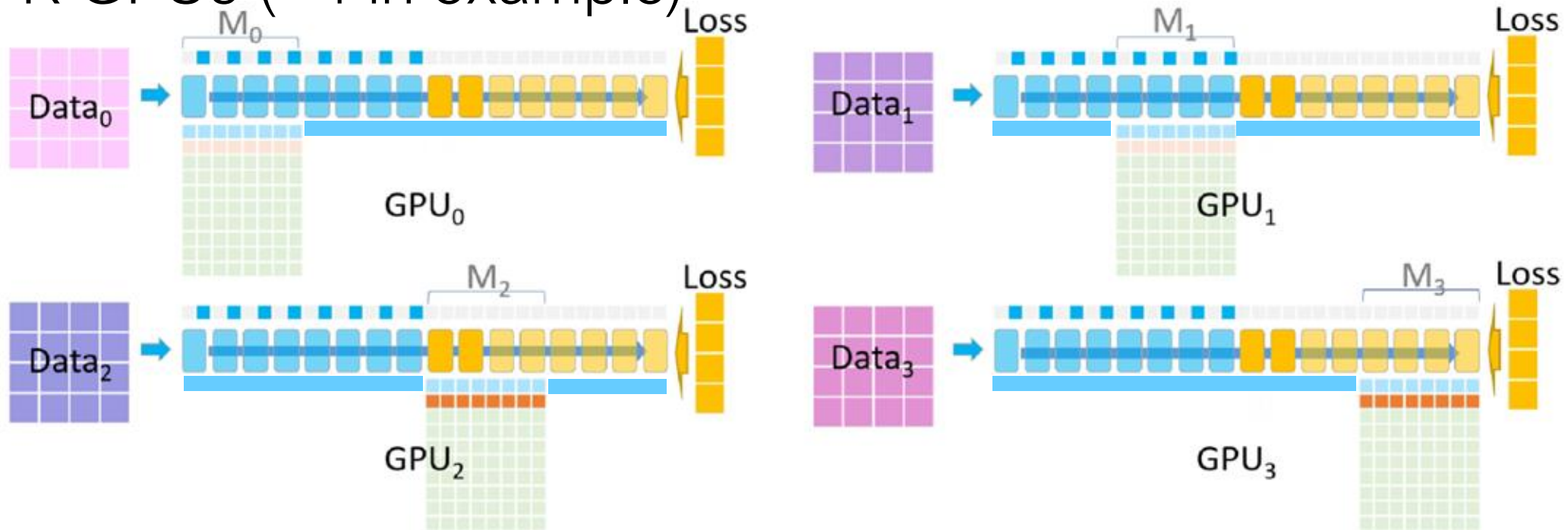


- GPU 0,1,3 pass the M2 gradients to GPU2 (ncclReduce)



ZeRO Stage 2: Partition Gradients

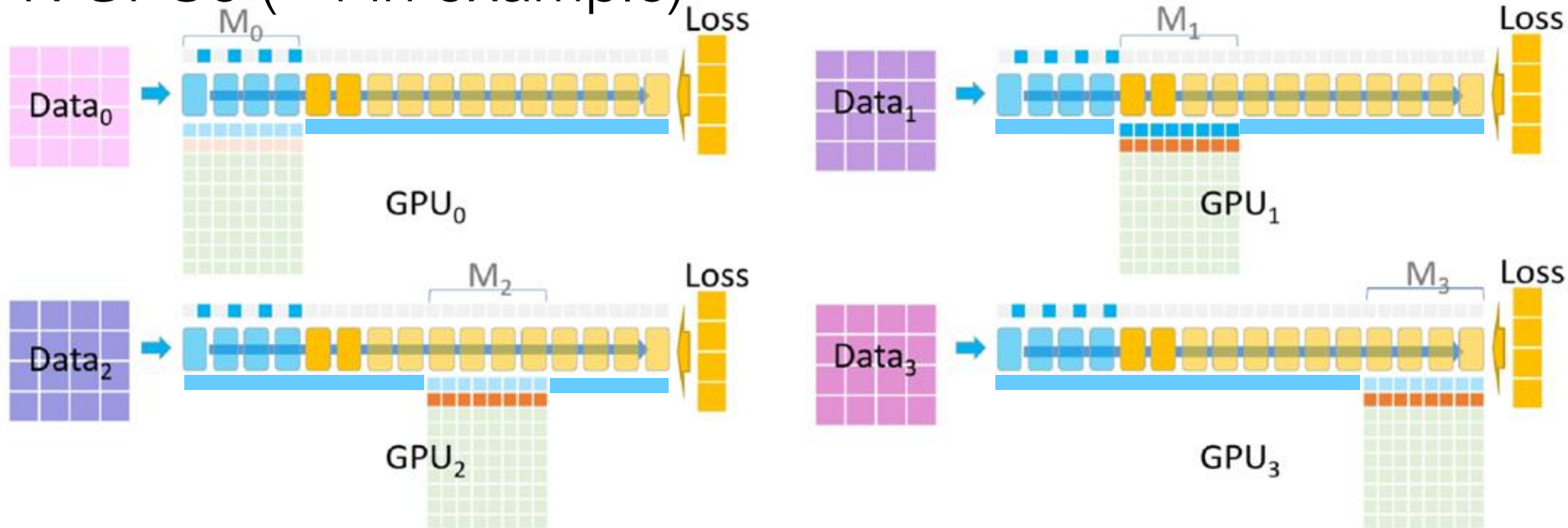
K GPUs (=4 in example)



- Then GPU₀, GPU₁, GPU₃ delete M₂ gradients, GPU₂ will keep M₂ gradients.

ZeRO Stage 2: Partition Gradients

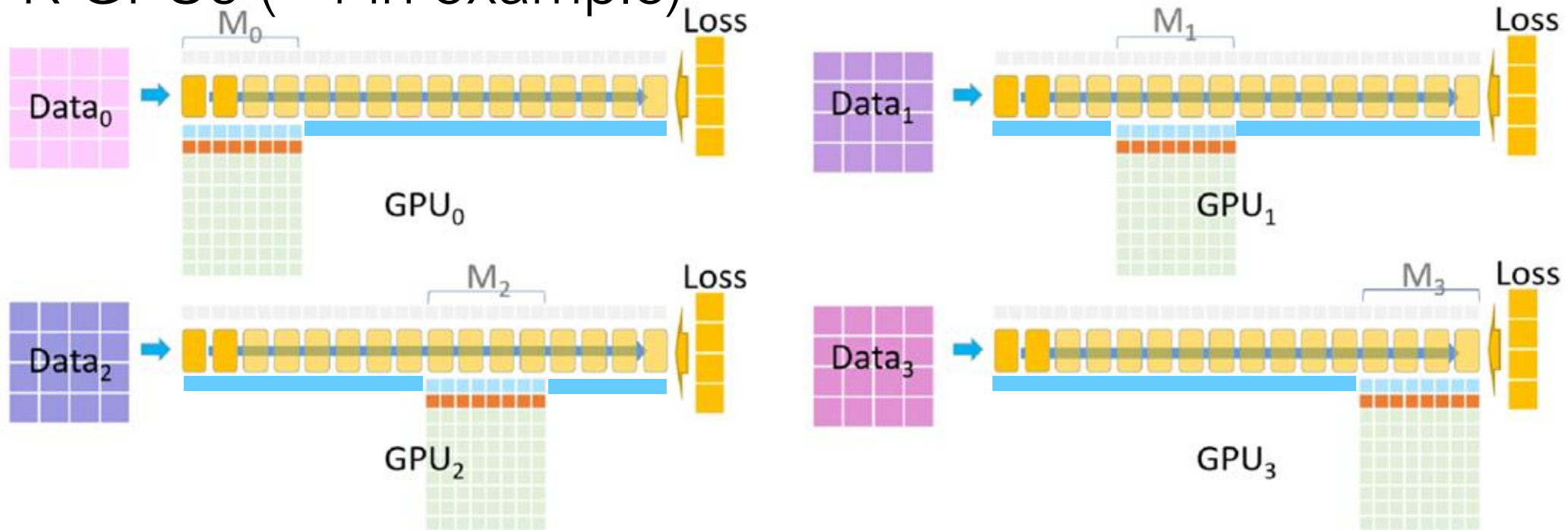
K GPUs (=4 in example)



- Continue backward pass for M1 gradients

ZeRO Stage 2: Partition Gradients

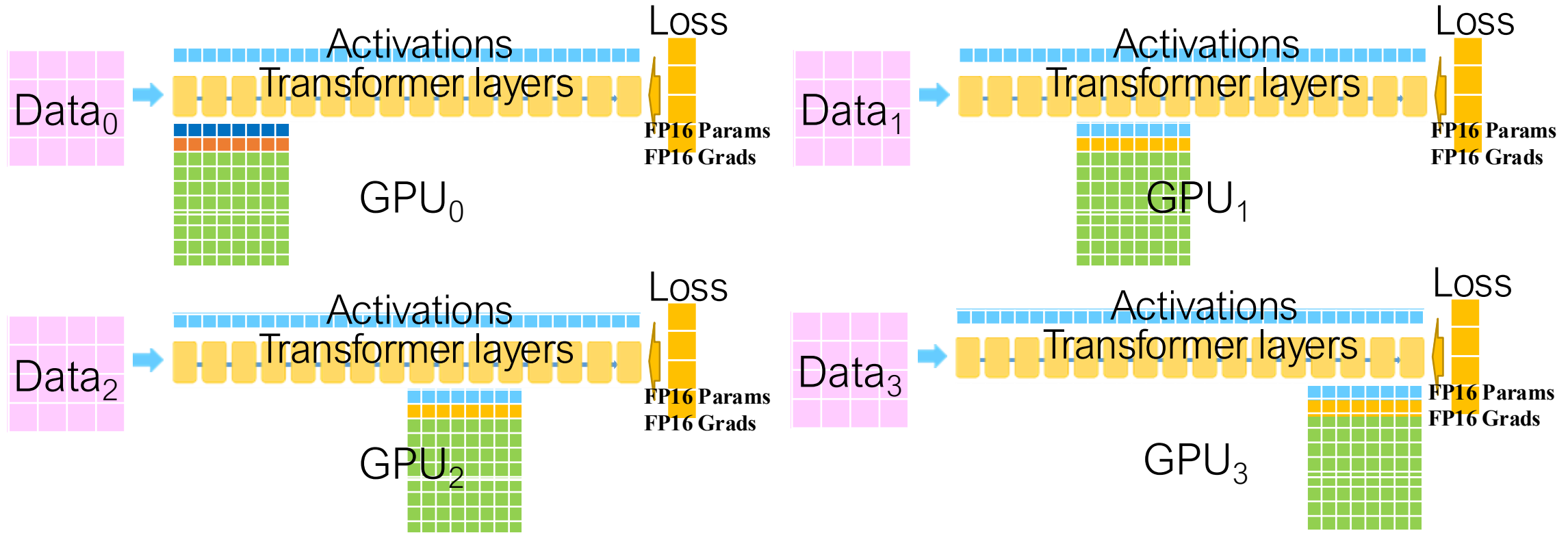
K GPUs (=4 in example)



- Continue backward pass for M₀ gradients

ZeRO Stage 3: Partition Parameters

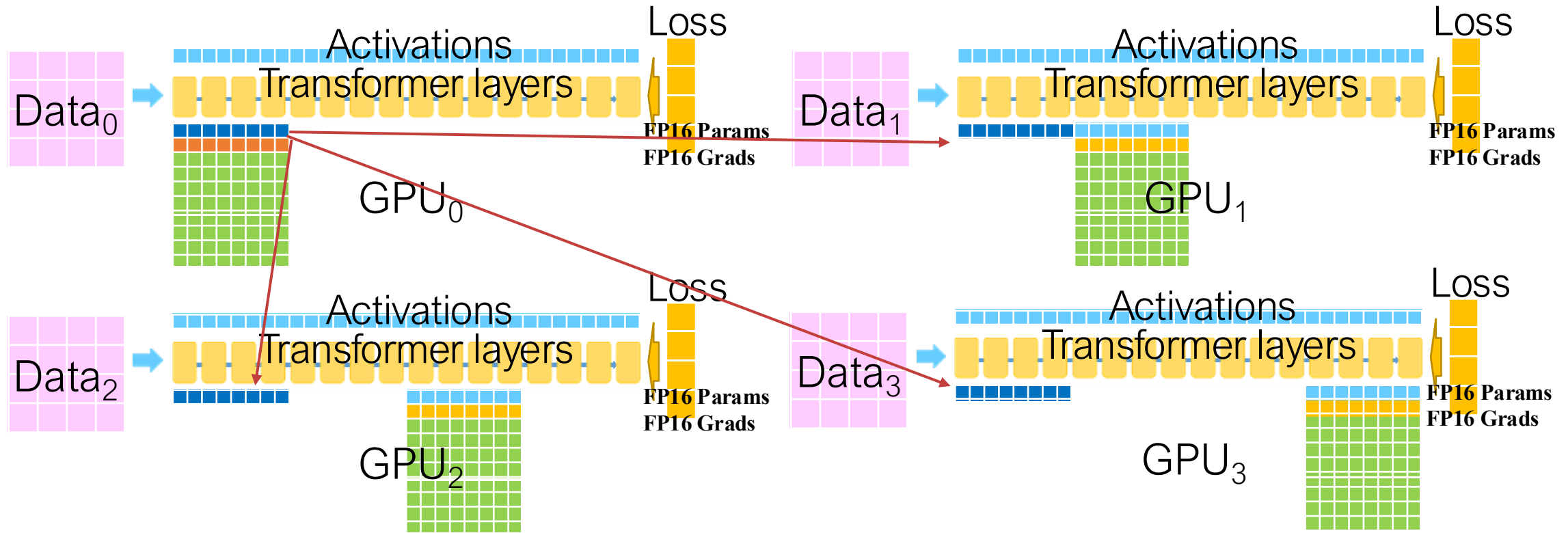
K GPUs (=4 in example)



Partition parameters to K parts

ZeRO Stage 3: Partition Parameters

K GPUs (=4 in example)

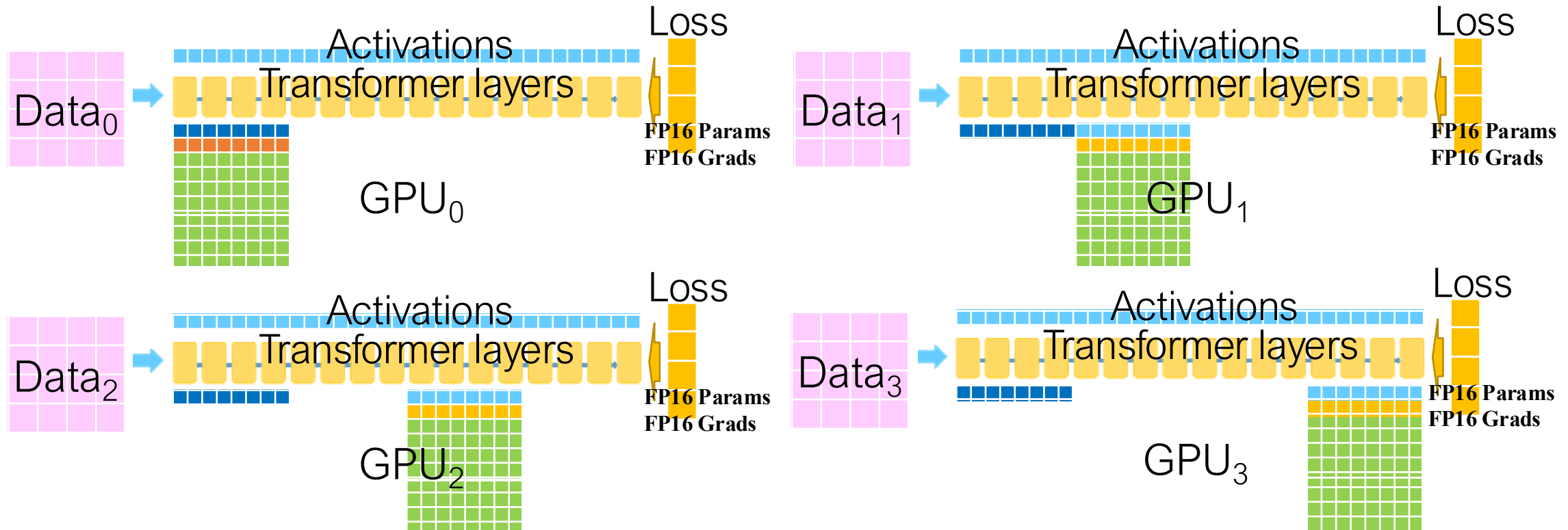


Partition parameters to K parts

During forward, GPU0 send first params to other GPUs(ncclBroadcast)

ZeRO Stage 3: Partition Parameters

K GPUs (=4 in example)



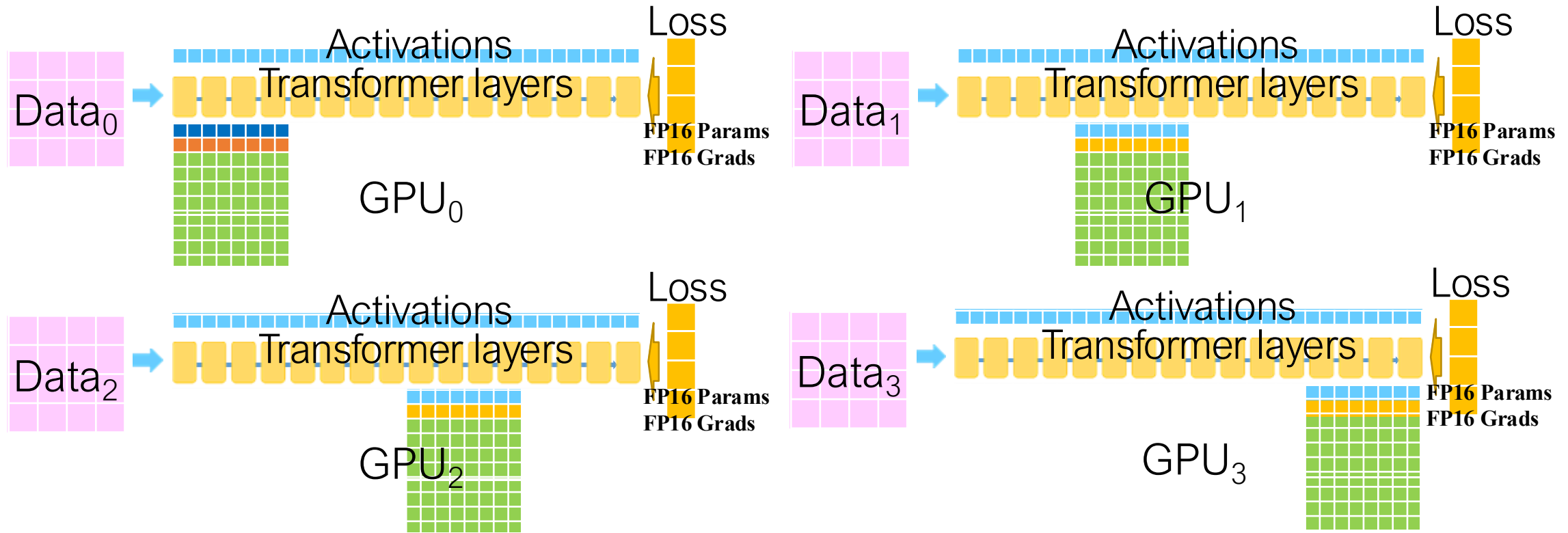
Partition parameters to K parts

During forward, GPU0 send first params to other GPUs(ncclBroadcast)

compute forward for first part layers

ZeRO Stage 3: Partition Parameters

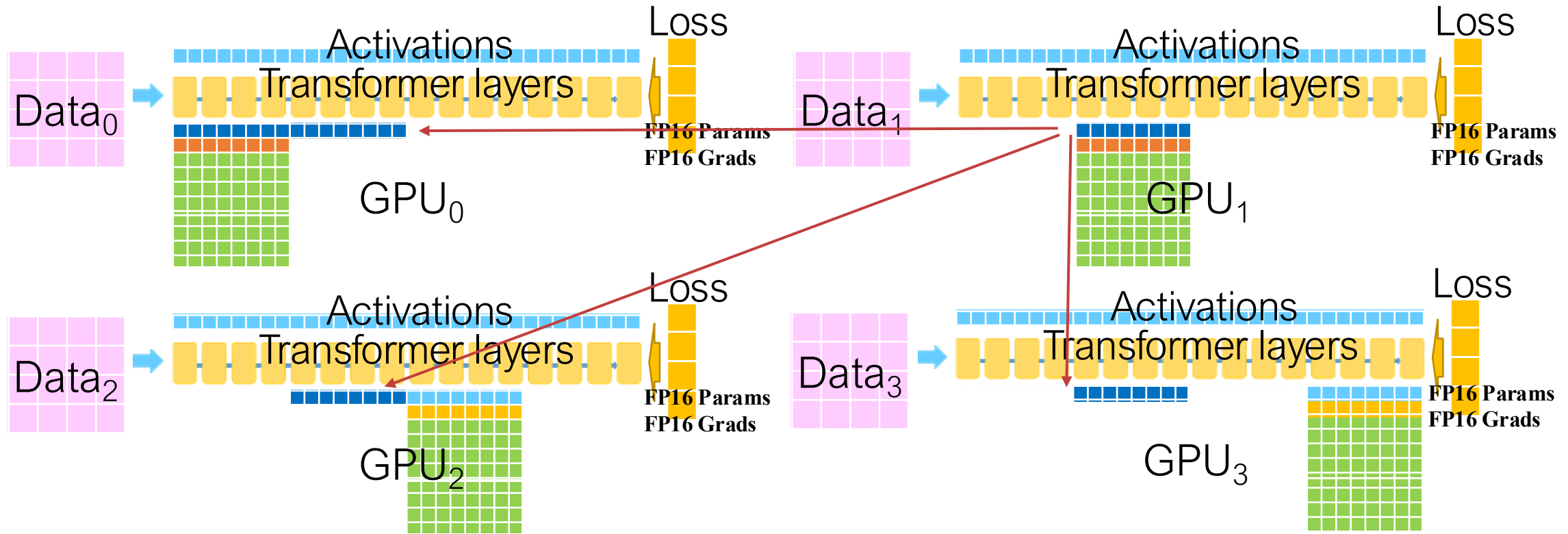
K GPUs (=4 in example)



GPU1,2,3 delete first part parameters, GPU0 still keep first part params

ZeRO Stage 3: Partition Parameters

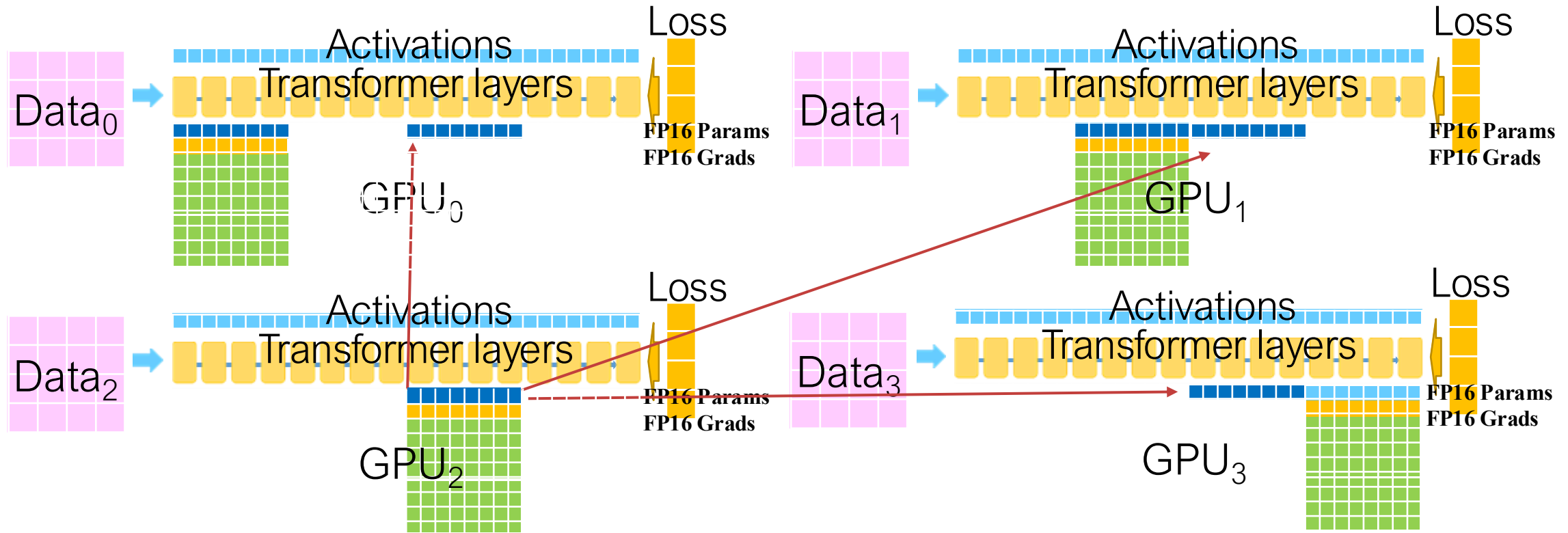
K GPUs (=4 in example)



Continue forward for next part, GPU₁ send its params to other GPUs (ncclBroadcast), compute forward for second part layers

ZeRO Stage 3: Partition Parameters

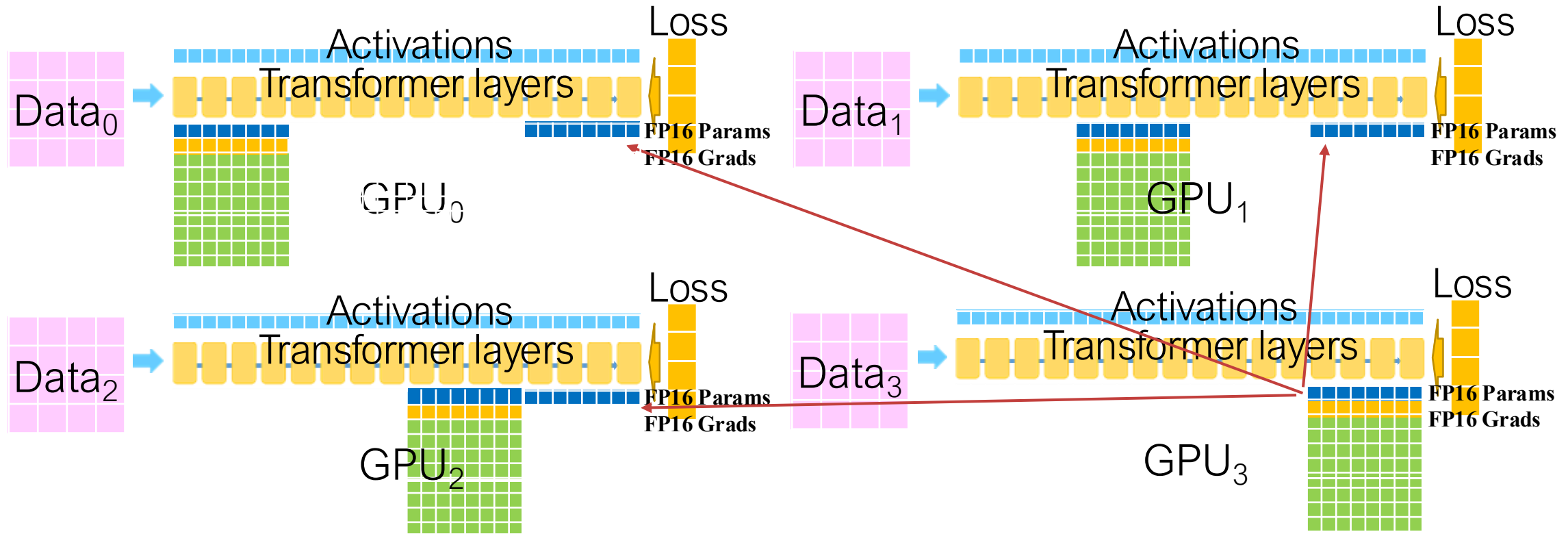
K GPUs (=4 in example)



Continue forward for next part, GPU2 send its params to other GPUs (ncclBroadcast), compute forward for 3rd part layers

ZeRO Stage 3: Partition Parameters

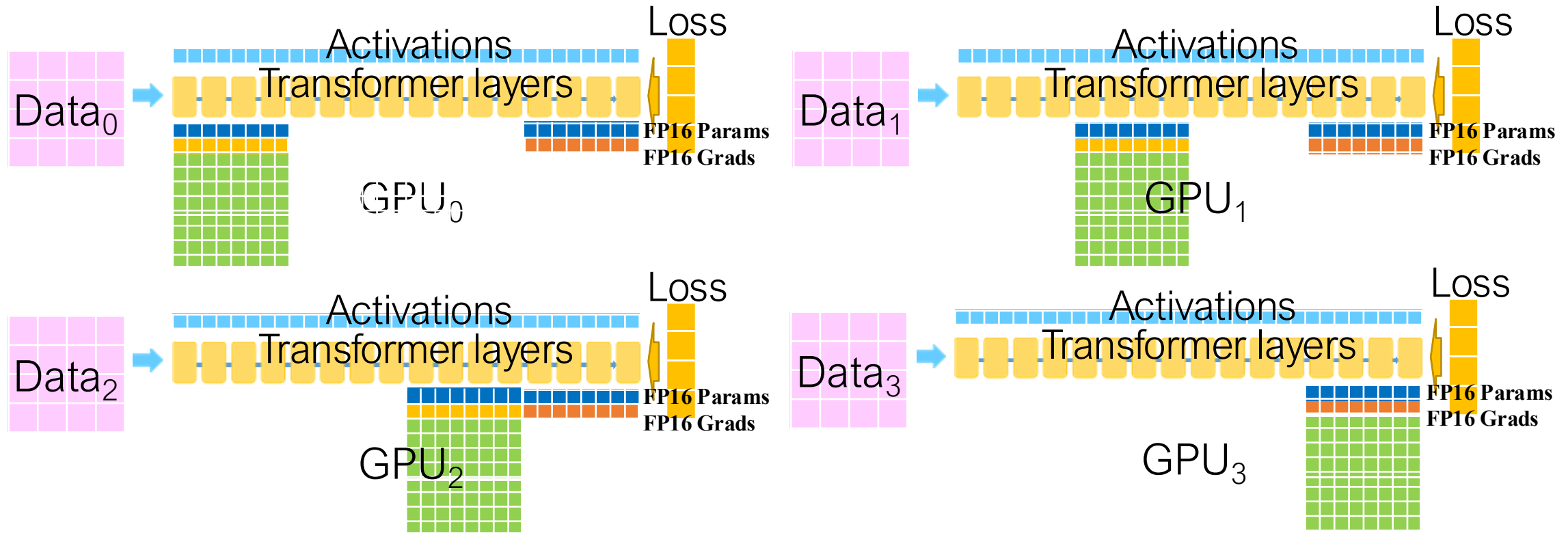
K GPUs (=4 in example)



Continue forward for next part, GPU3 send its params to other GPUs (ncclBroadcast), compute forward for 4th part layers

ZeRO Stage 3: Partition Parameters

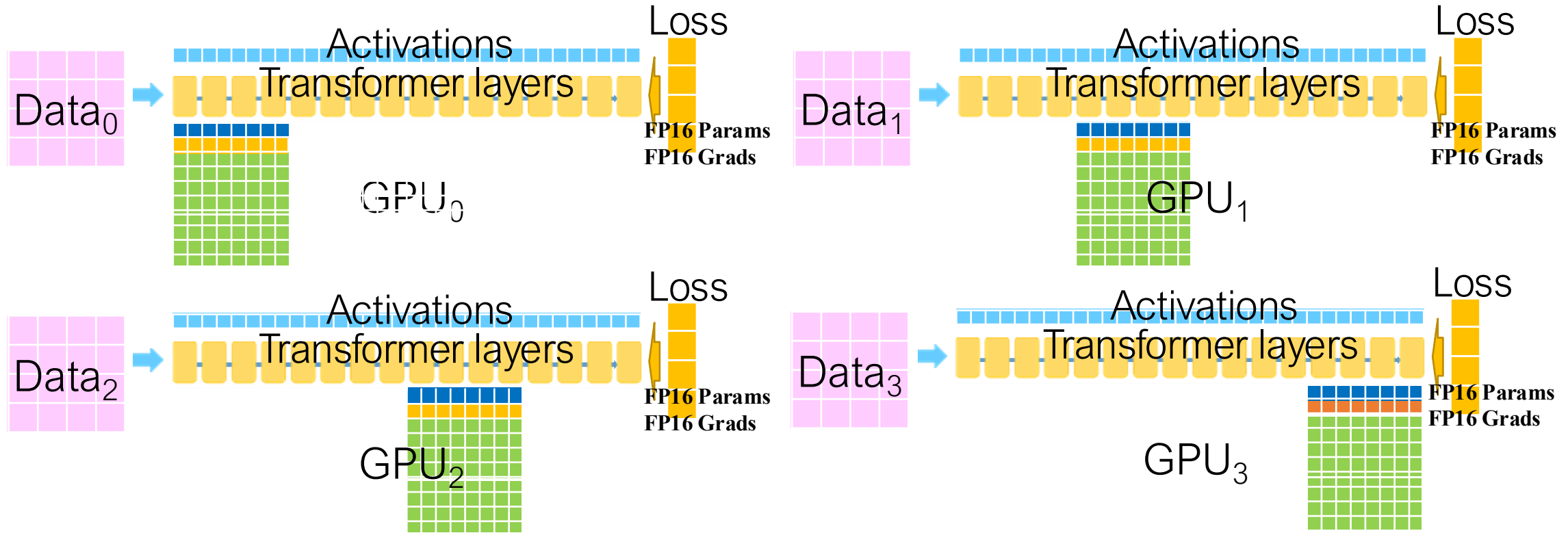
K GPUs (=4 in example)



During backward, compute backward for 4th part layers
Reduce grads to GPU3 (same as ZeRO stage 2)

ZeRO Stage 3: Partition Parameters

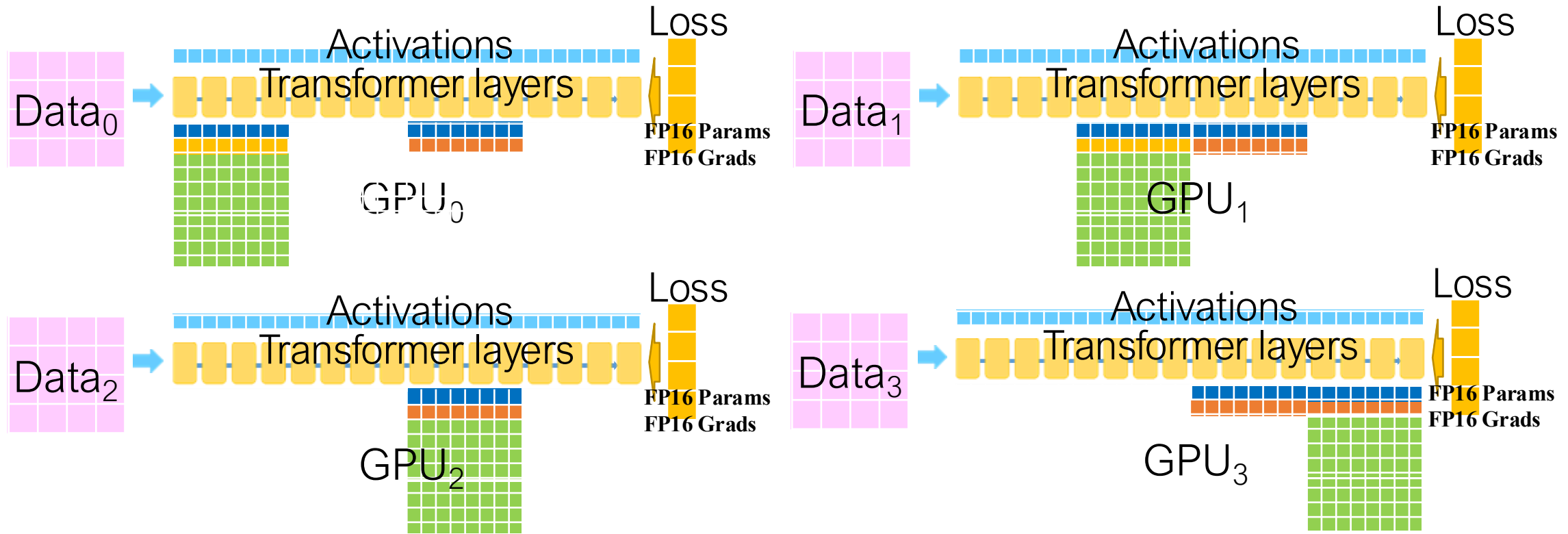
K GPUs (=4 in example)



GPU₀, GPU₁, GPU₂ delete 4th part parameters and gradients

ZeRO Stage 3: Partition Parameters

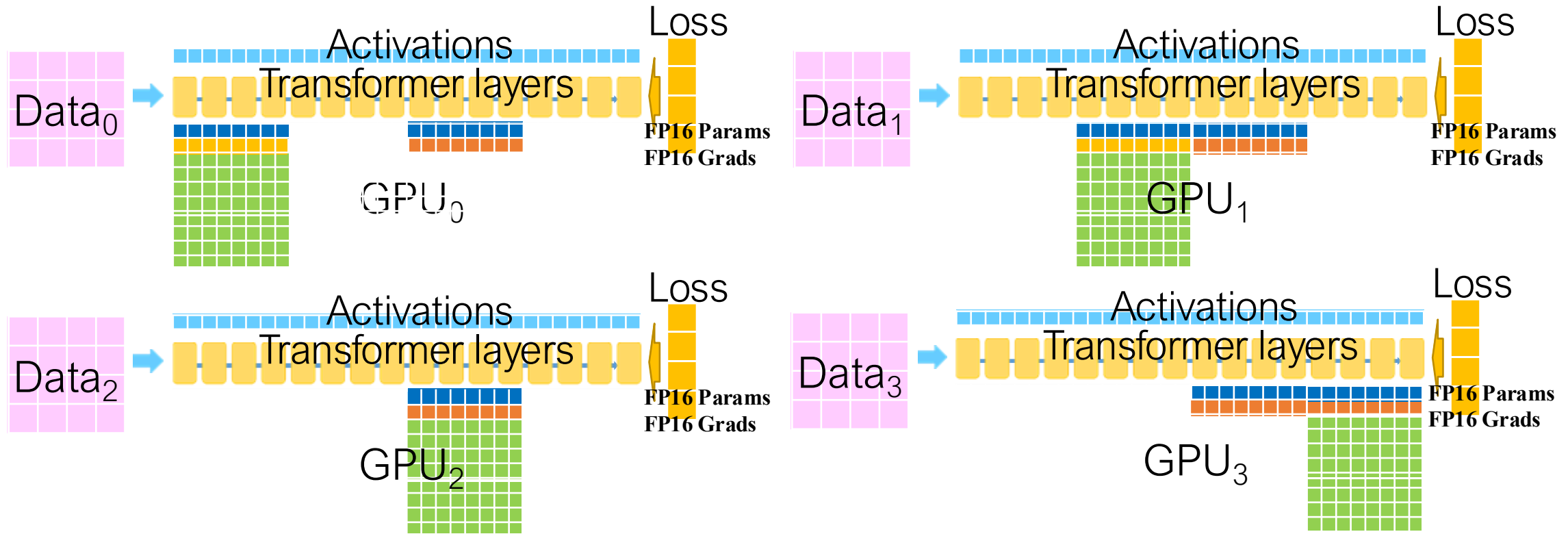
K GPUs (=4 in example)



Continue backward, GPU2 broadcast its parameters to GPU0, 1, 3
All GPUs perform backward to calculate local gradients for part 3

ZeRO Stage 3: Partition Parameters

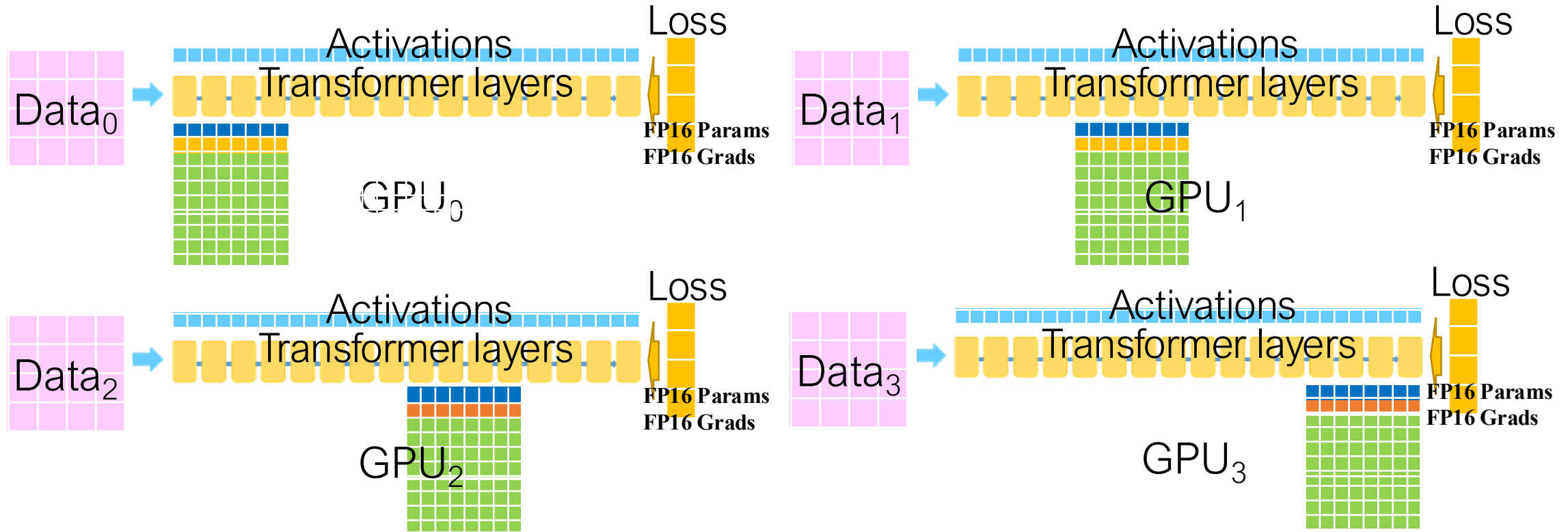
K GPUs (=4 in example)



GPU0, 1, 3 sends its gradients for part 3 to GPU2 (the same as ZeRO stage 2)

ZeRO Stage 3: Partition Parameters

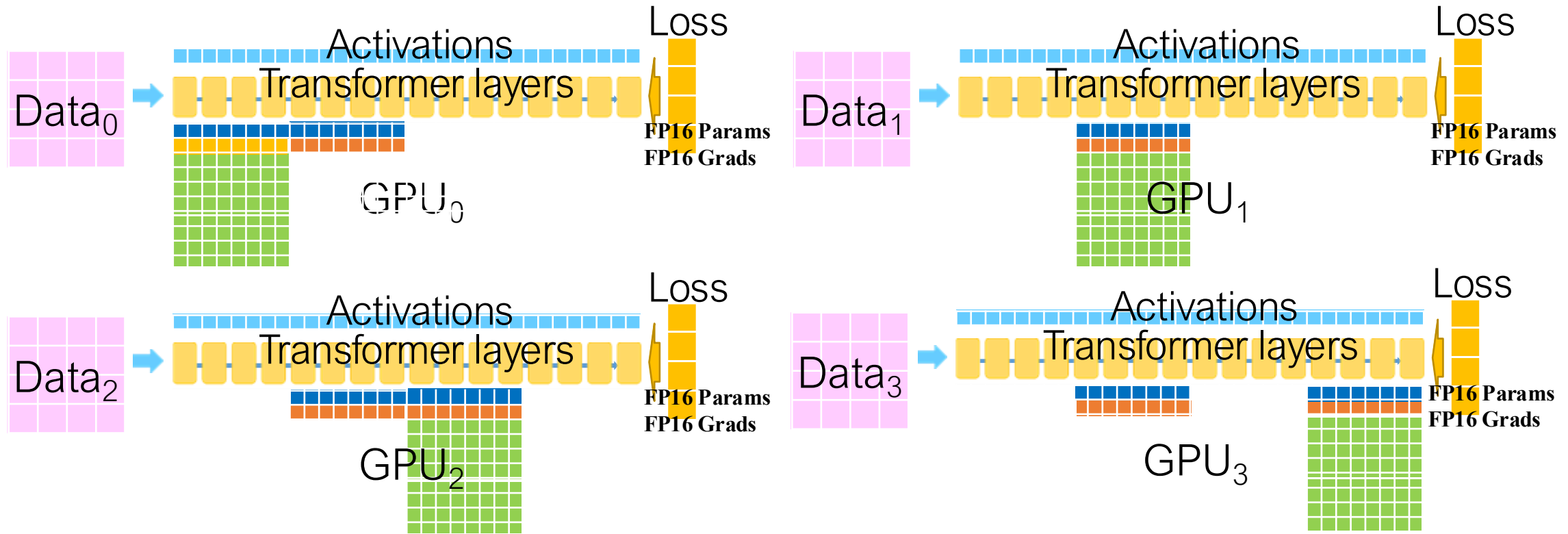
K GPUs (=4 in example)



GPU₀, 1, 3 deletes its parameters and gradients for part 3

ZeRO Stage 3: Partition Parameters

K GPUs (=4 in example)



Continue backward for part 2, part 1

Memory Consumption for ZeRO in DDP

- LLM with N parameters, Optimizer needs M bytes for each parameter (e.g. $M=8, 12$ or 16)
- Original memory: $4N + M*N$
- ZeRO-1: $4N + M*N / K$
- ZeRO-2: $2N + (2+M) * N / K$
- ZeRO-3: $(4+M)*N / K$
- But, at the cost of additional parameter transfer

ZeRO Communication Cost

- Zero stage 1 and 2 (optimizer state and gradient) doesn't introduce additional communication, while enabling up to 8x memory reduction
- Zero stage 3 (parameter) communication
 - 2x broadcast during forward/backward, plus Reduce in Zero2.
Total 3x communication overhead.
 - Baseline needs ScatterReduce + AllGather

Outline

- Memory Consumption for LLM training
- Partitioning and reducing the memory for data parallel training
 - Partition optimizer states
 - Partition gradients
 - Partition parameters
- ➔ • Other memory optimization

ZeRO: Reduce Memory for Activations

- Partitioned Activation Checkpointing
 - Tensor Parallelism by design requires a replication of the activations
 - Split every activation to different devices. Gather them when needed.

ZeRO: Buffers

- Constant Size Buffers (similar to Bucketing in pytorch ddp)
 - Buffer is used in doing all-reduce to improve bandwidth.
 - Modern implementations fuses all the parameters into a single buffer.
 - ZeRO uses constant size buffers to be more efficient for a large model.

ZeRO: Memory Defragmentation

- Memory Defragmentation
 - Long-lived memory (Model parameters, Optimizer state): Store together
 - Short-lived memory (Discarded activations)
- We can further improve by memory reuse as in LightSeq.

Reduce Communication in ZeRO DDP

- Applying block-wise quantization technique to parameters during forward (broadcast params). FP16 → INT 8.
 - zeropoint quantization
- Apply quantization during backward ReduceScatter, FP16 → INT8 or INT4
- better partition parameters: maintain a full set of parameters on each node

Memory Benchmarking of ZeRO

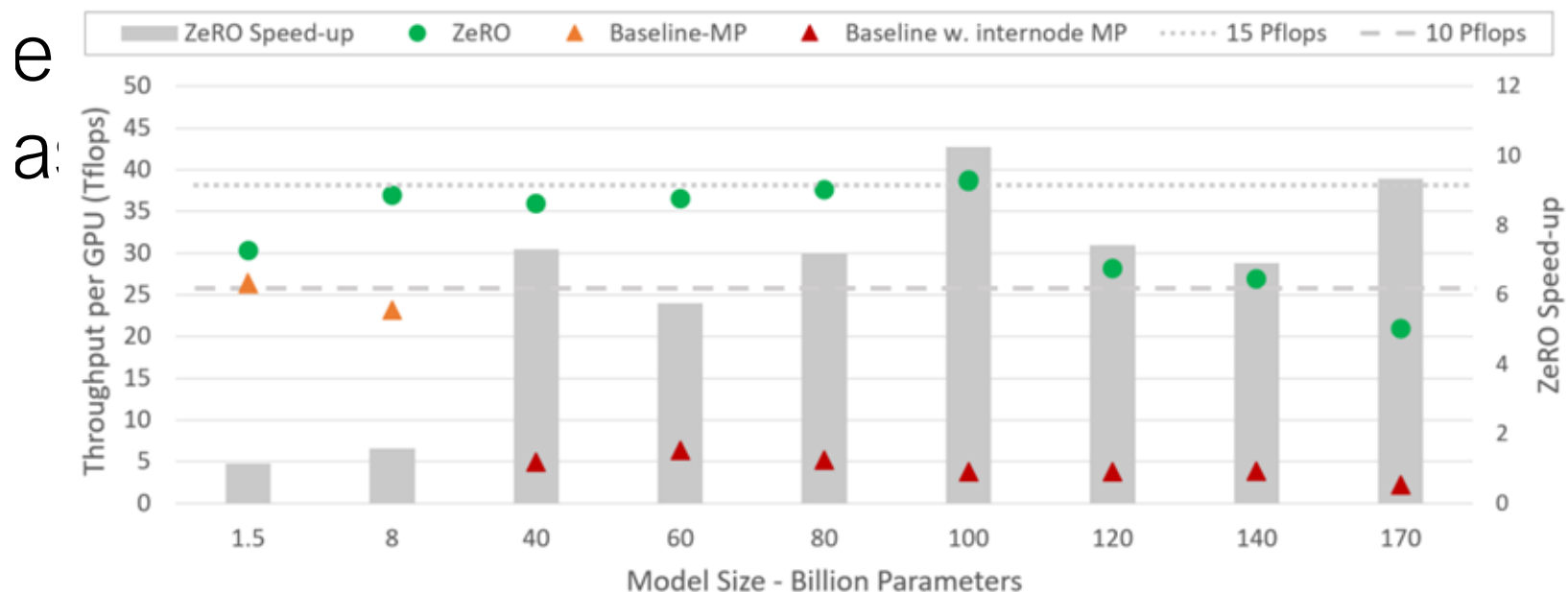
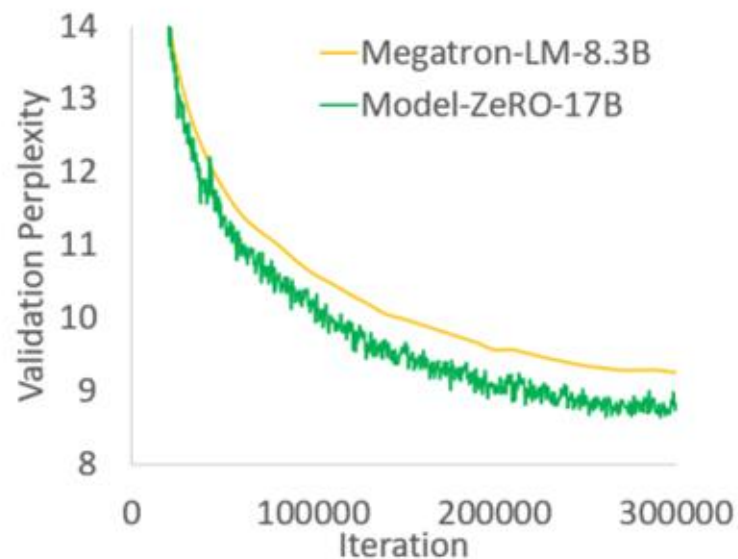
- Theoretical: On a 32GB V100 clusters (Up to 1024 V100),
- Enable the training of a model with 1 Trillion (1000B)

DP	7.5B Model (GB)			128B Model (GB)			1T Model (GB)		
	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}	P_{os}	P_{os+g}	P_{os+g+p}
1	120	120	120	2048	2048	2048	16000	16000	16000
4	52.5	41.3	30	896	704	512	7000	5500	4000
16	35.6	21.6	7.5	608	368	128	4750	2875	1000
64	31.4	16.6	1.88	536	284	32	4187	2218	250
256	30.4	15.4	0.47	518	263	8	4046	2054	62.5
1024	30.1	15.1	0.12	513	257	2	4011	2013	15.6

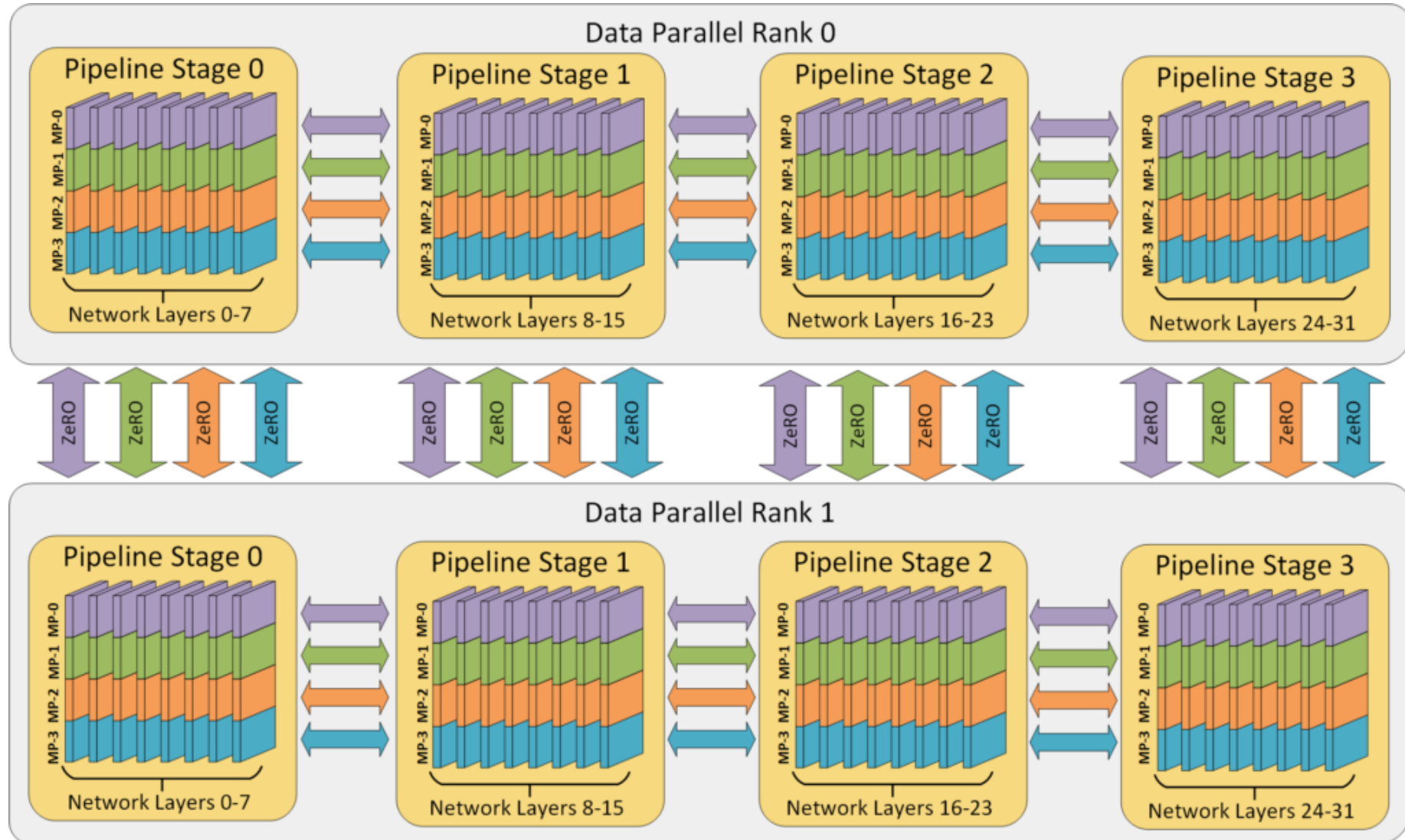
e)

Training Performance of ZeRO

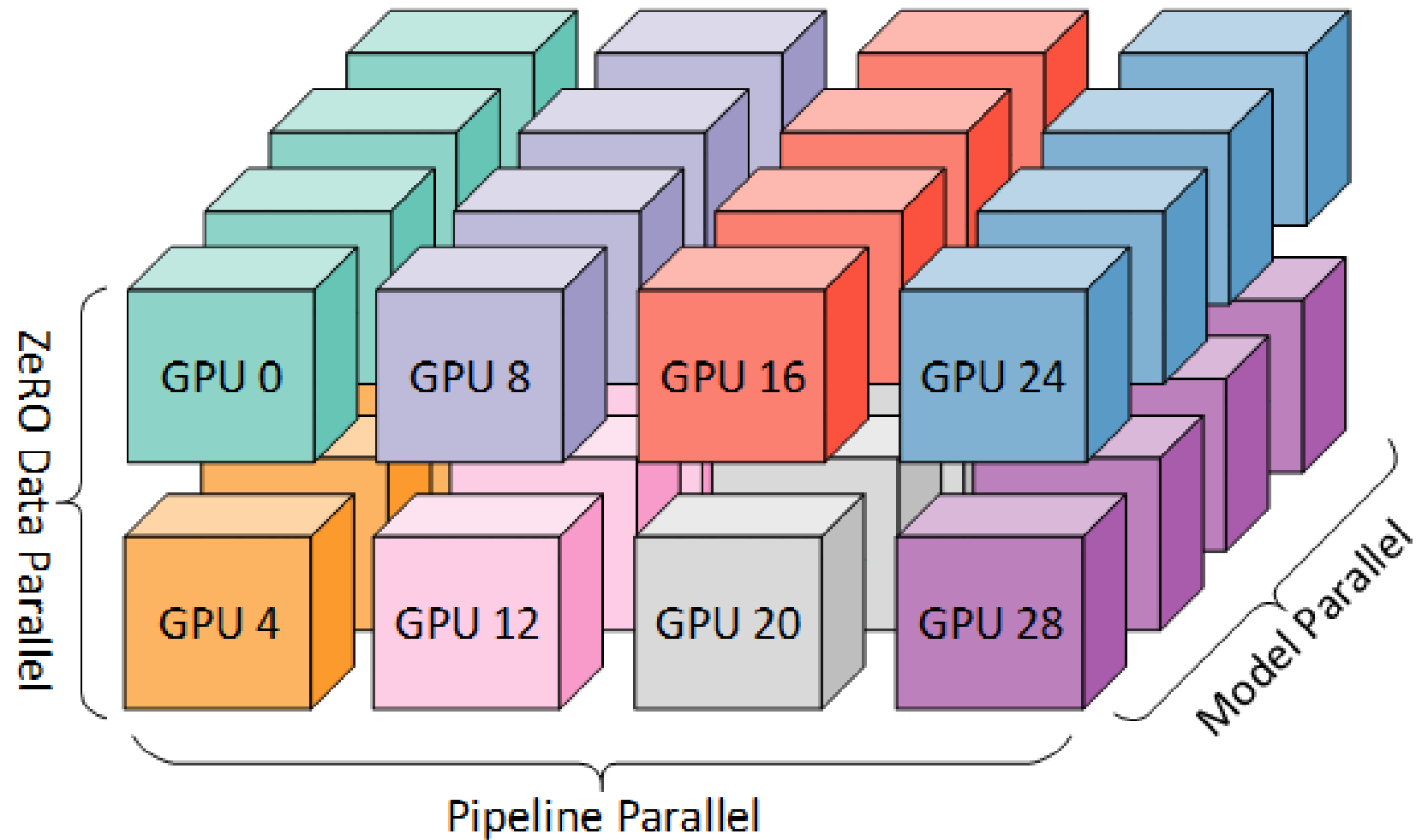
- Train a 17B model (Turing-NLG. The largest as of 2020.1) and has SOTA perplexity in Webtext-103.



Combining ZeRO with Pipeline Parallelism and Tensor Parallelism (3D Parallelism)



Combining ZeRO with Pipeline Parallelism and Tensor Parallelism (3D Parallelism)



Summary

- ZeRO : reducing memory footprint in distributed training
 - ➔ enables training significantly larger models
- Key idea: partition optimizer states, gradients, and parameters.
- Pros:
 - Lower memory usages significantly.
 - Scalable, flexible, easy-to-use.
- Cons:
 - Some stages introduce extra communication overhead, depending on infrastructure (PCI-E / NVLink)

Code Example

- https://github.com/lmsystem/lmsys_code_examples/blob/main/deepspeed_example/DeepSpeed-Example.ipynb

Reading for Next

- Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, Tijmen Blankevoort. A White Paper on Neural Network Quantization. 2021
- Hubara et al. Improving Post Training Neural Quantization: Layer-wise Calibration and Integer Programming. 2020.
- Dettmers et al. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. Neurips 2022.

Quiz