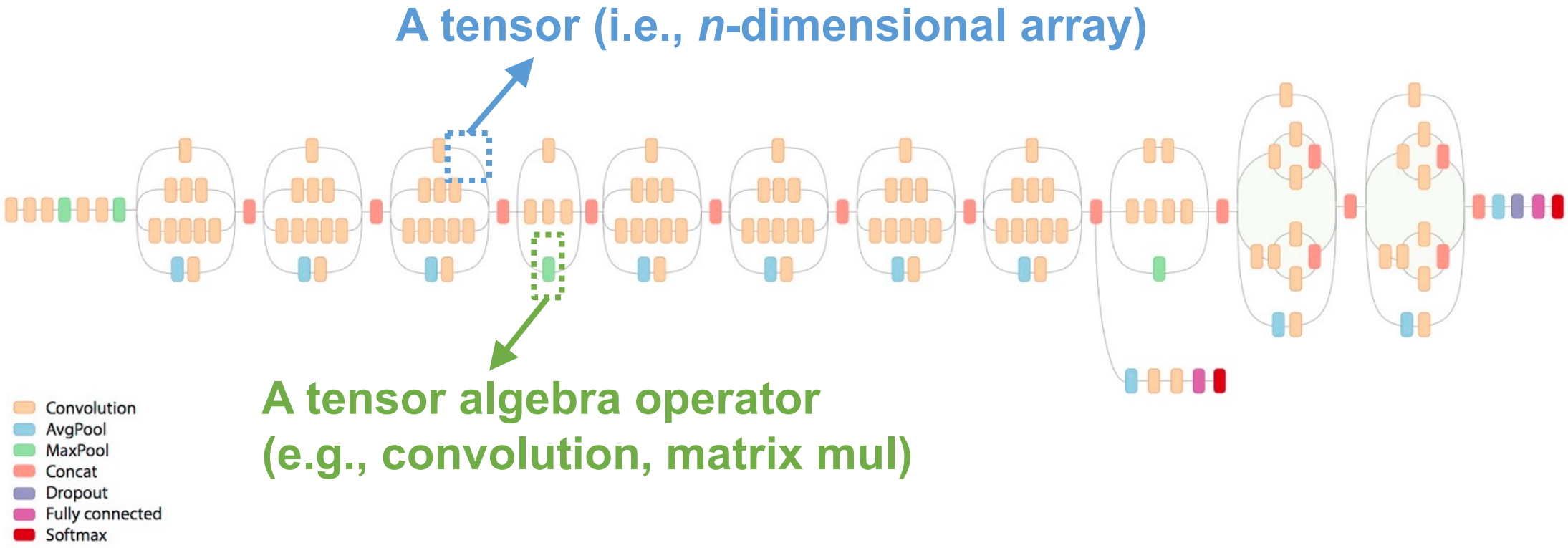
Lecture 21: Parallel Deep Learning (Data Parallelism)

Parallel Computer Architecture and Programming

CMU 15-418/15-618, Fall 2026

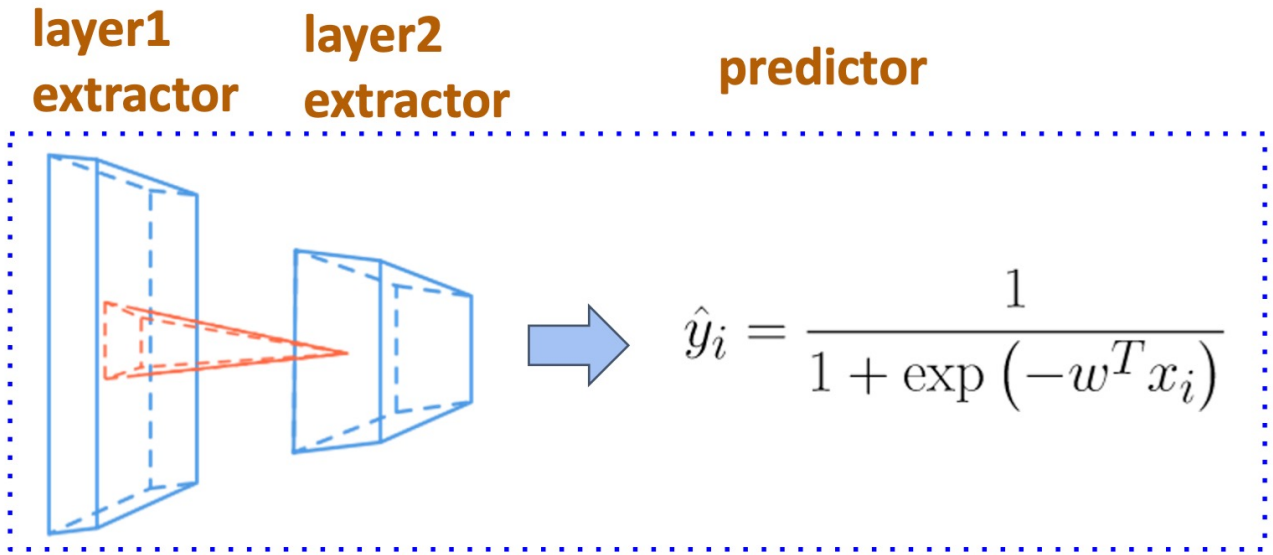
Recap: Deep Neural Network

- Collection of simple trainable mathematical units that work together to solve complicated tasks



Recap: DNN Training Overview





$$\hat{y}_i = \frac{1}{1 + \exp\left(-w^T x_i\right)}$$

Loss function

True label

Objective

$$L(w) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \lambda \|w\|^2$$

Training

$$w \leftarrow w - \eta \nabla_w L(w)$$

1. Compute how the loss changes with each parameter
2. Move w to the opposite direction to reduce the loss
3. Repeat over data

Recap: What Each Symbol Means

Predictor

$$\hat{y}_i = \frac{1}{1 + \exp\left(-w^T x_i\right)}$$

$\hat{y}_i$	prediction for example i (a probability)
x_i	input features for example i
w	model weights, learned in training
$w^T x_i$	weights · input = one score (^{T} = transpose)
$1/(1+e^{-z})$	sigmoid: turns score z into a 0–1 probability

Objective

$$L(w) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \lambda \|w\|^2$$

$L(w)$	total loss for a given set of weights
$\Sigma_i = 1...n$	sum over all n training examples
$l(y_i, \hat{y}_i)$	per-example loss: true label y_i vs. prediction $\hat{y}_i$
$\lambda \|w\|^2$	penalty on large weights; λ sets its strength

Training

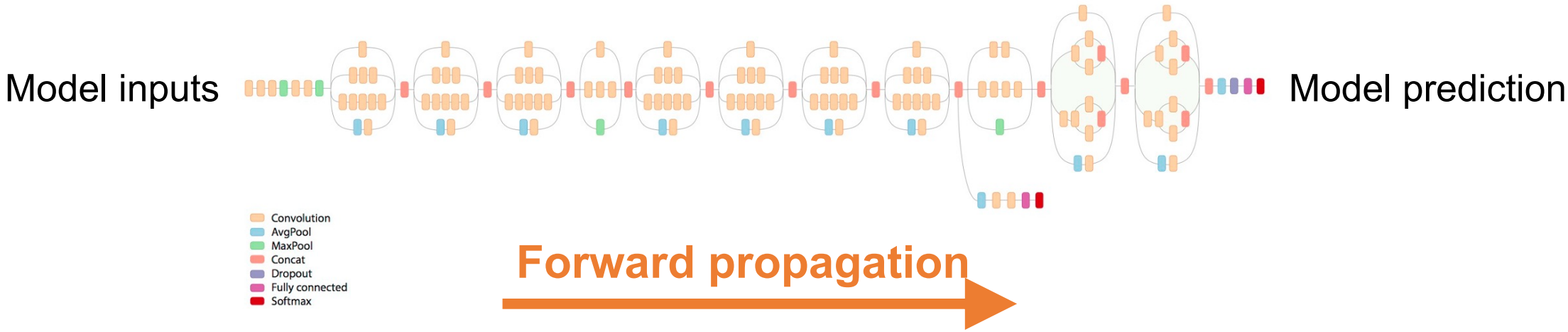
$$w \leftarrow w - \eta \nabla_w L(w)$$

$\leftarrow$	update: replace w with the right-hand side
η	learning rate: how big a step to take
$\nabla_w L(w)$	gradient: direction the loss rises fastest
$w - \eta \nabla_w L(w)$	step against the gradient, so loss goes down

DNN Training Process

Train ML models through many iterations of 3 stages

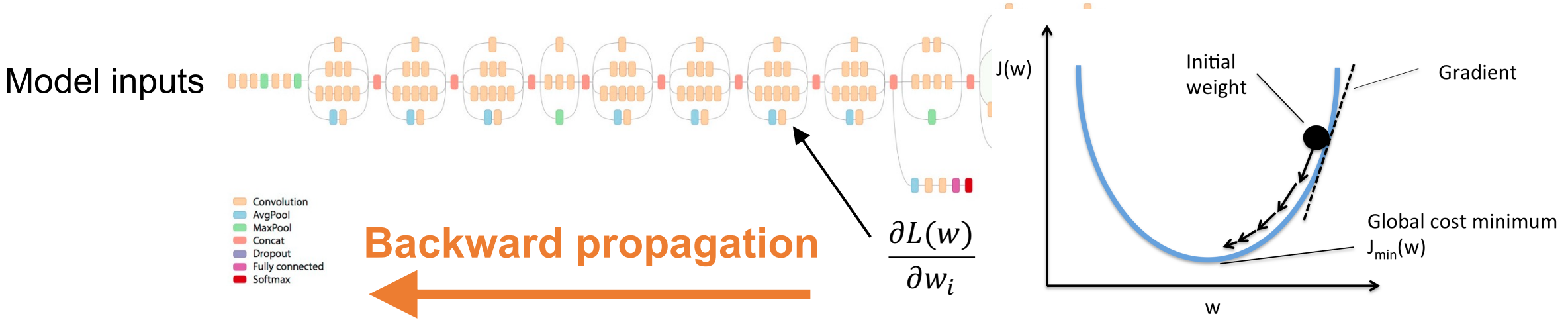
1. **Forward propagation**: apply model to a batch of input samples and run calculations through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce error for each trainable weight
3. **Weight update**: use the loss value to update model weights



DNN Training Process

Train ML models through many iterations of 3 stages

1. **Forward propagation**: apply model to a batch of input samples and run calculations through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce a gradient for each trainable weight
3. **Weight update**: use the loss value to update model weights



DNN Training Process

Train ML models through many iterations of 3 stages

1. **Forward propagation**: apply model to a batch of input samples and run calculations through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce a gradient for each trainable weight
3. **Weight update**: use the gradients to update model weights

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

Gradients of individual samples

How can we parallelize DNN training?

The gradient of the loss function wrt the parameters

Batch Size

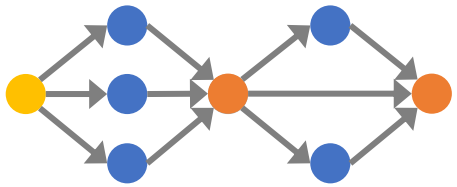
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

Parameter vector → weights at iter i

Learning rate → step size

Loss function for w_i

Data Parallelism

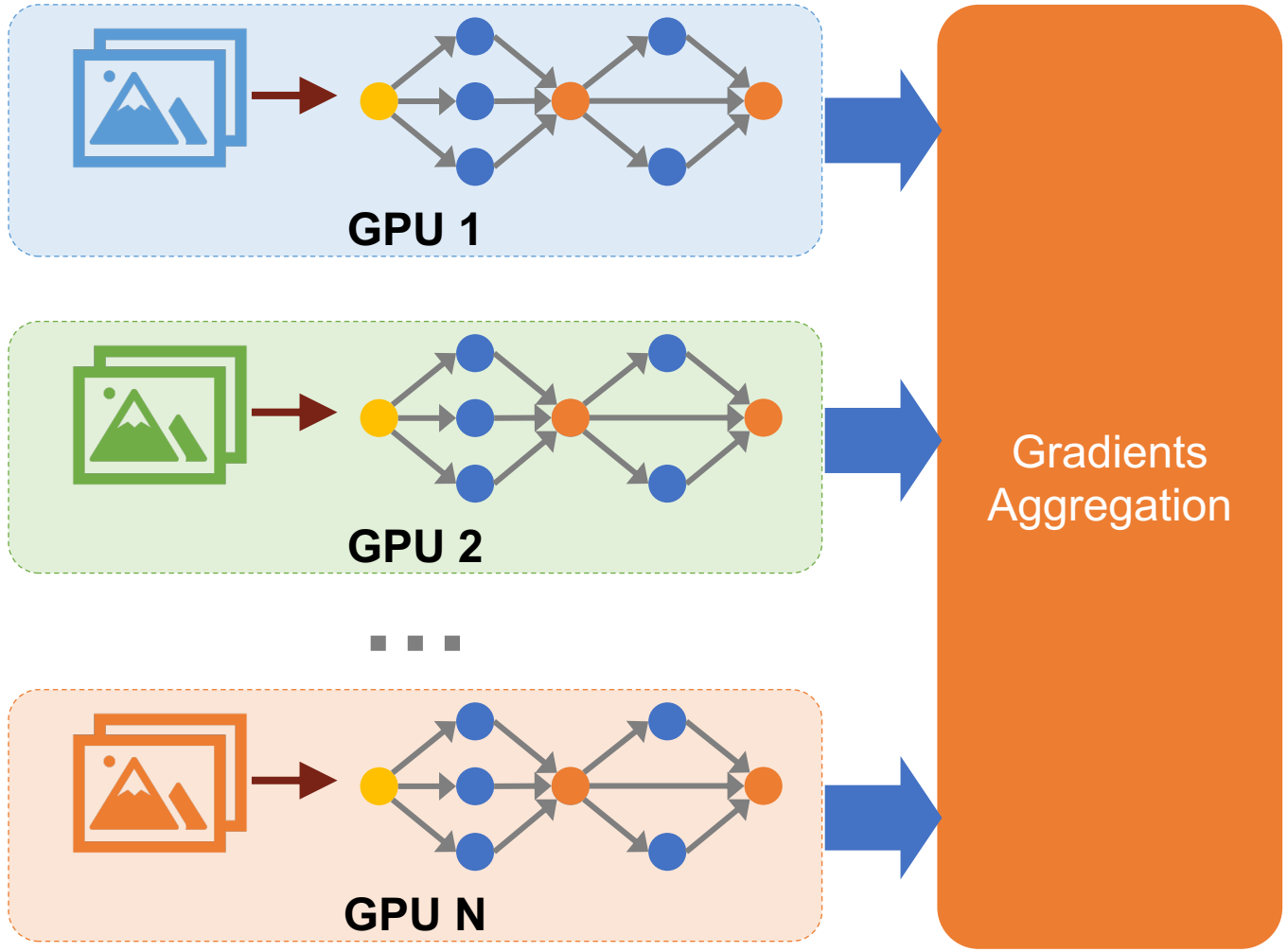


ML Model



Training Dataset

$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \sum_{j=1}^{n} \nabla L_j(w_i)$$

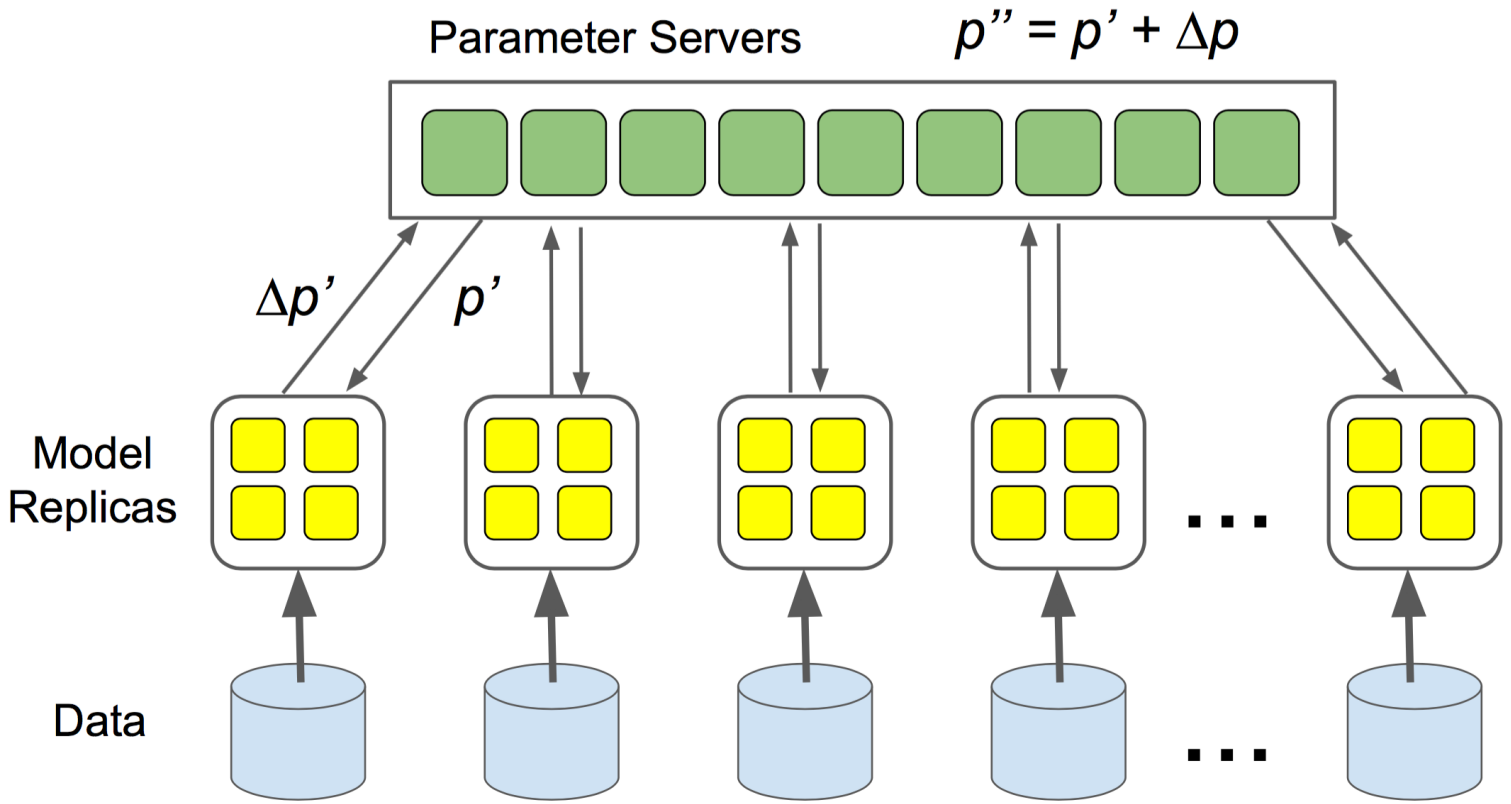


1. Partition training data into batches

2. Compute the gradients of each batch on a GPU

3. Aggregate gradients across GPUs

Data Parallelism: Parameter Server



Workers push gradients to parameter servers and pull updated parameters back

$\Delta p = \Sigma \Delta p_k'$

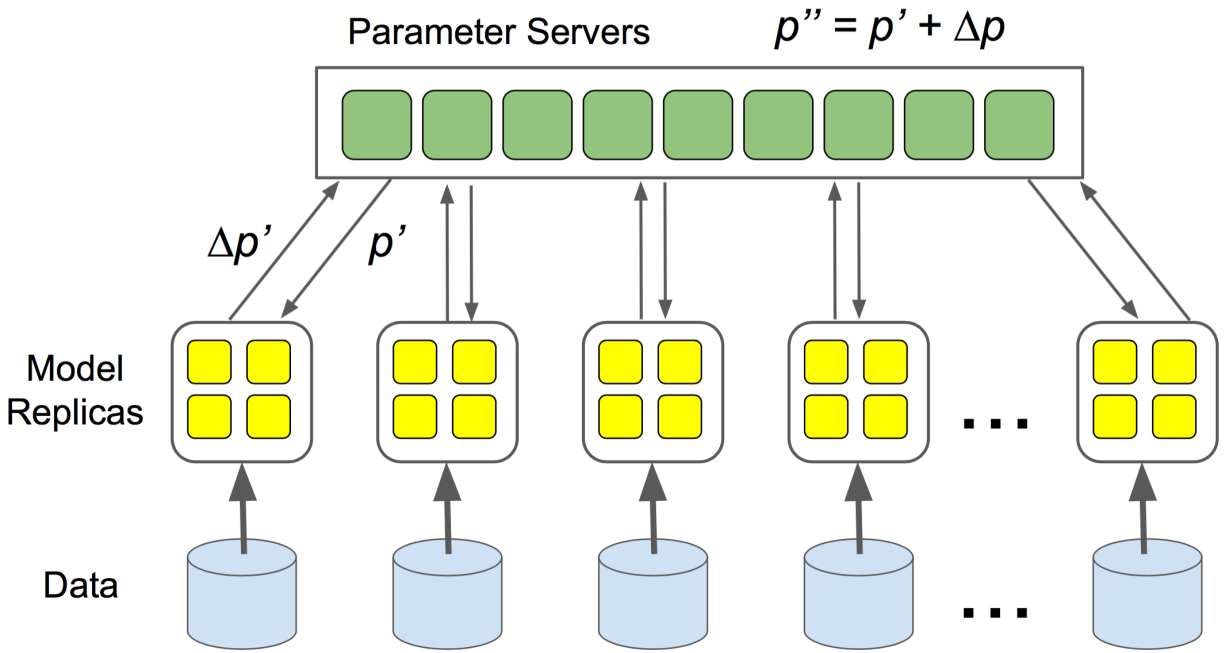
over workers k = 1…N

Servers sum all workers' gradients, then update once

Inefficiency of Parameter Server

- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers

- How can we decentralize communication in DNN training?



Inefficiency of Parameter Server

- **Centralized communication**: all workers communicate with parameter servers for weights update; cannot scale to large numbers of workers

- How can we decentralize communication in DNN training?

- **AllReduce**: perform element-wise reduction across multiple devices

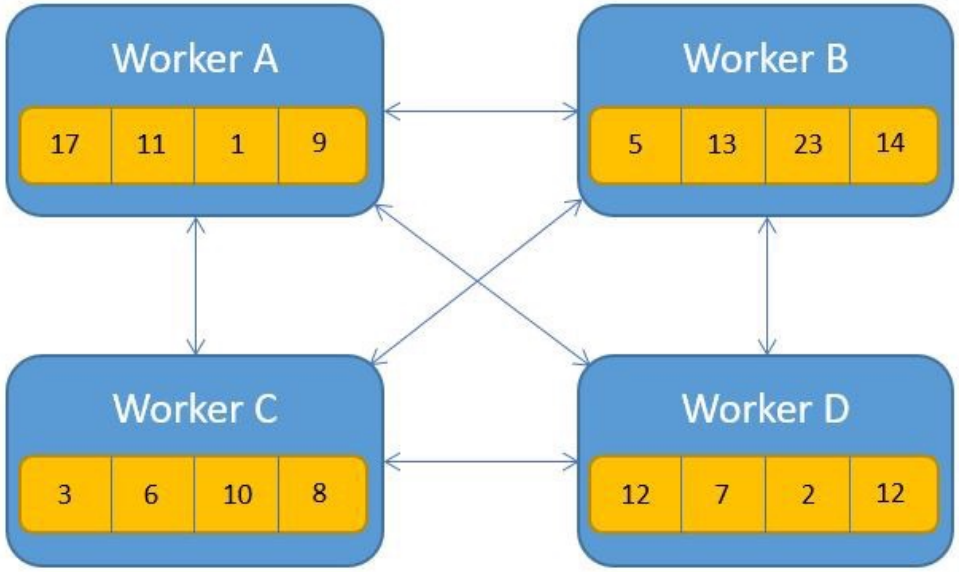


Different Ways to Perform AllReduce

- Naïve AllReduce
- Ring AllReduce
- Tree AllReduce
- Butterfly AllReduce

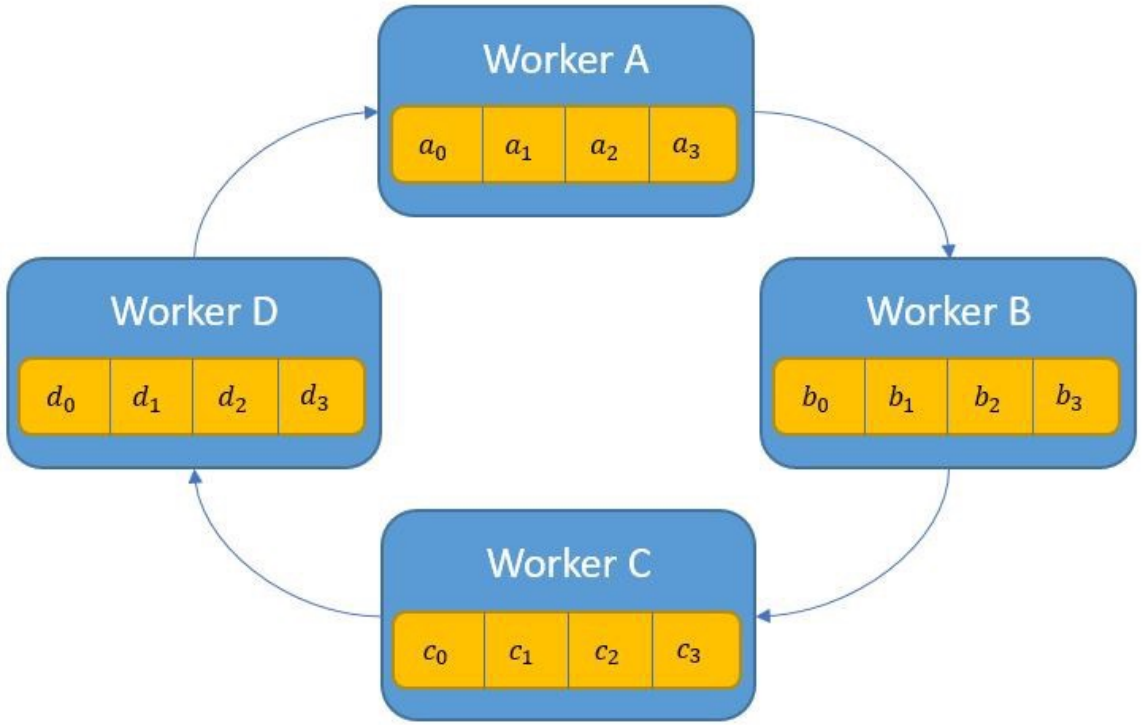
Naïve AllReduce

- Each worker can send its local gradients to all other workers
- If we have N workers and each worker contains M parameters
- Overall communication: N * (N-1) * M parameters
- **Issue**: each worker communicates with all other workers; have the same scalability issue as parameter server



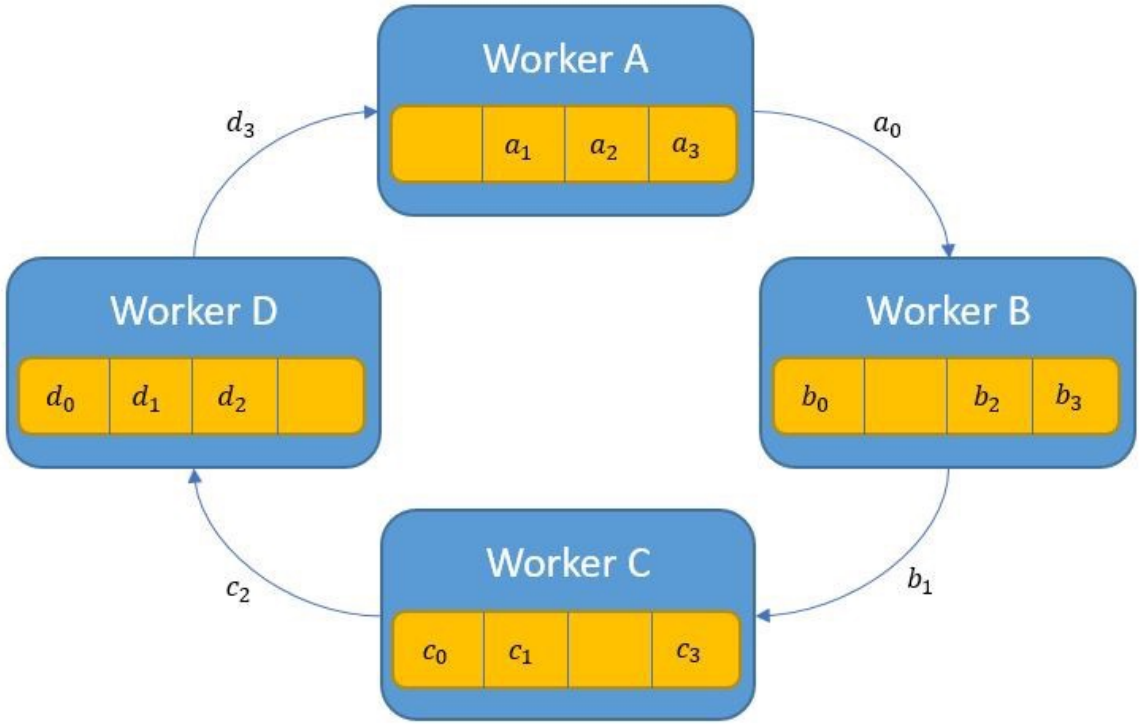
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N times



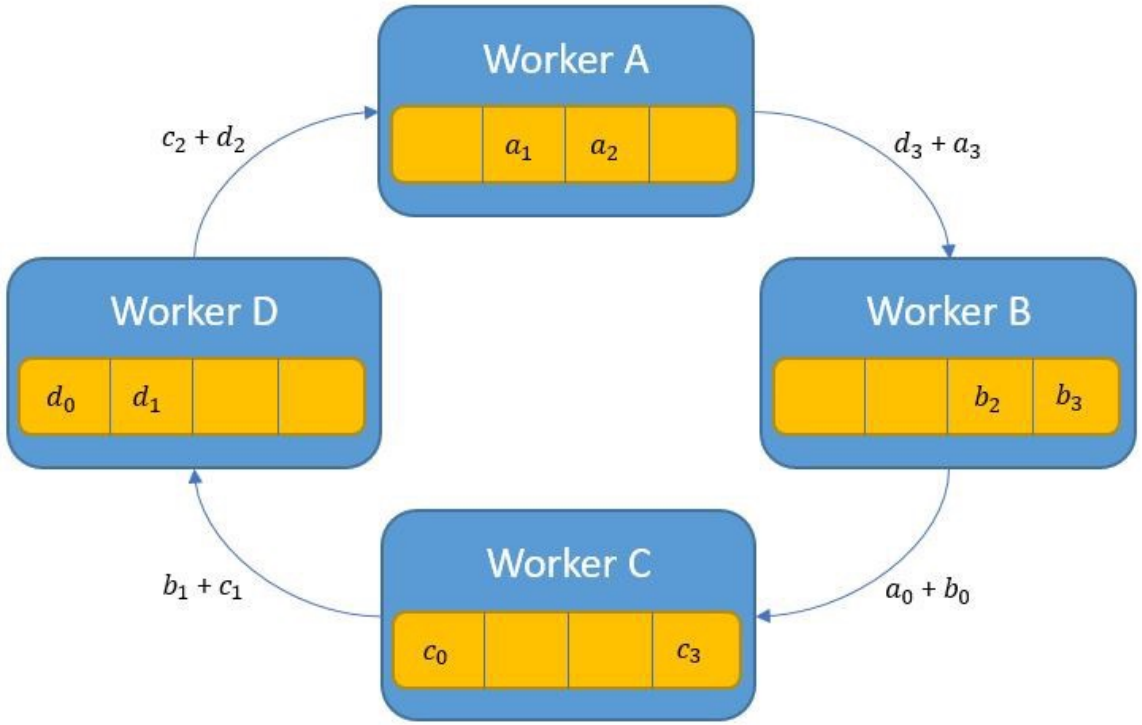
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N-1 times



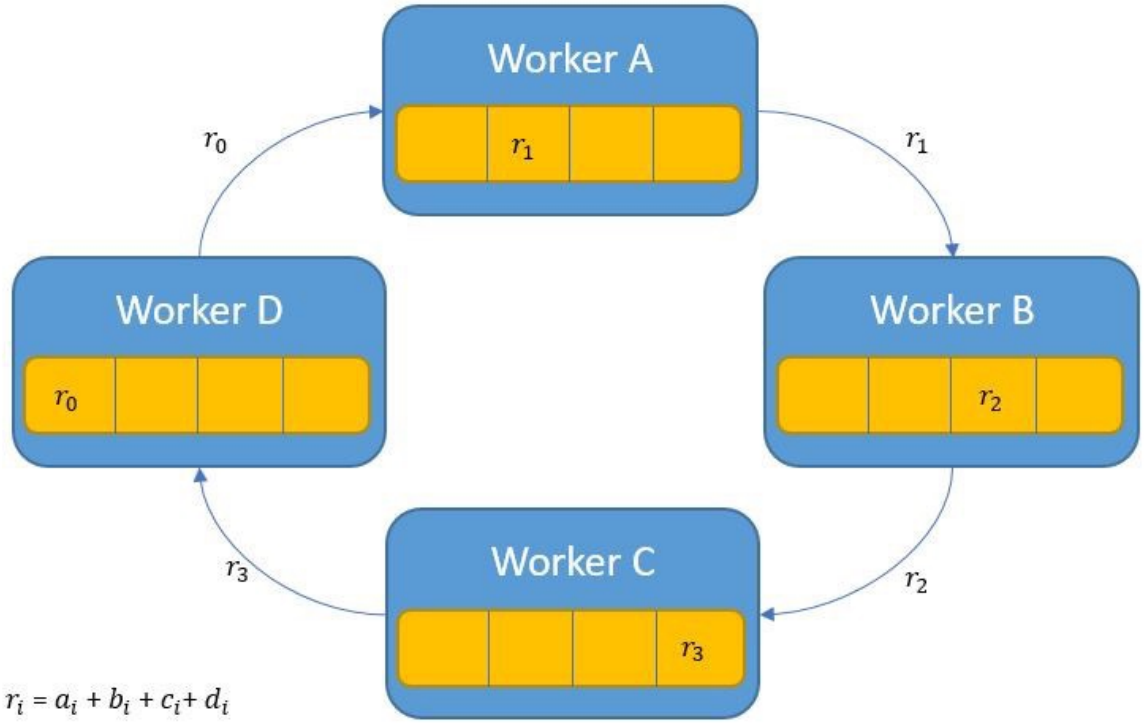
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N-1 times



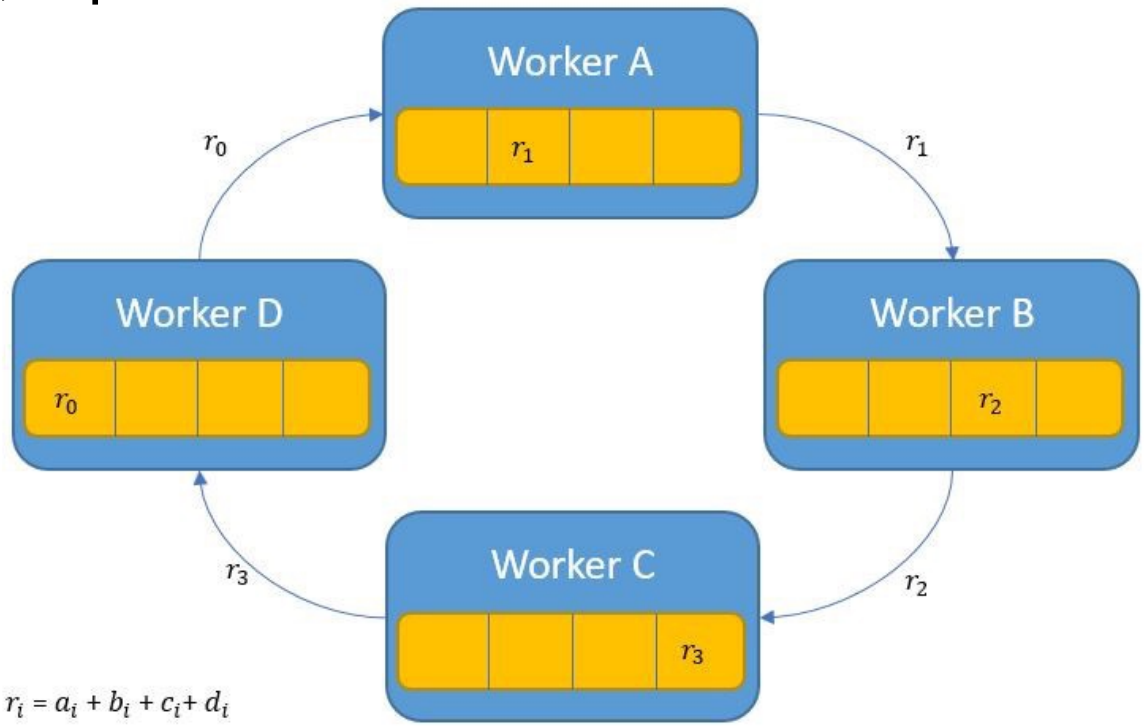
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N-1 times
- After step 1, each worker has the aggregated version of M/N parameters



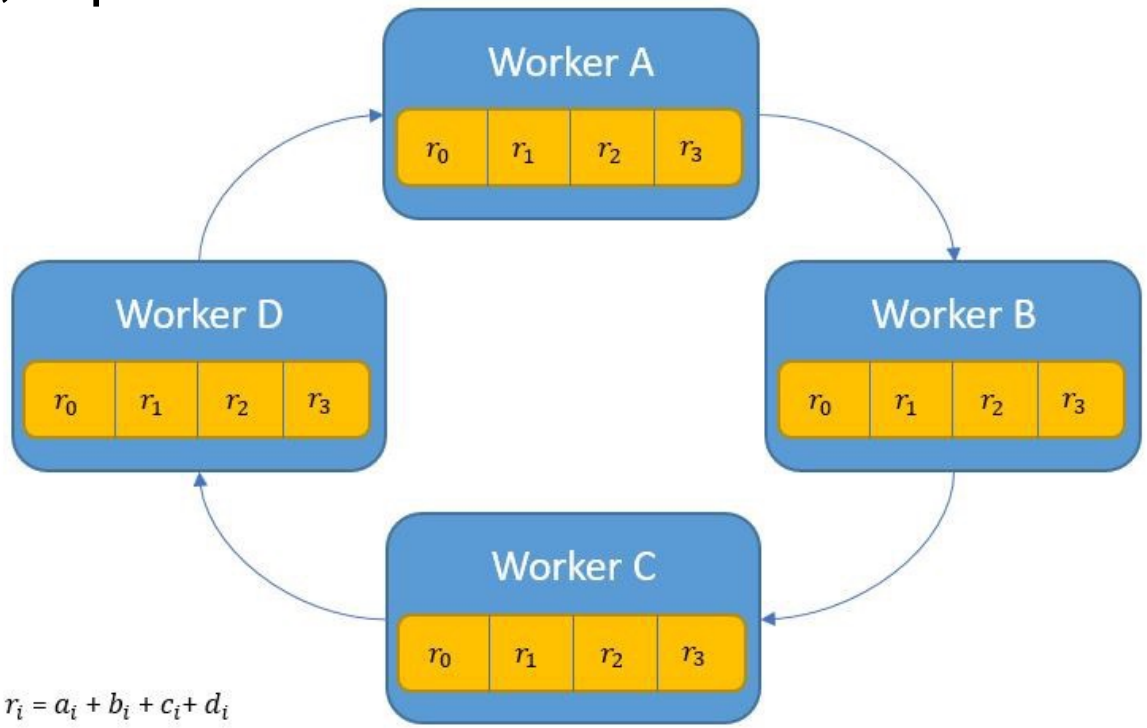
Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N-1 times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N-1 times



Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N-1 times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N-1 times

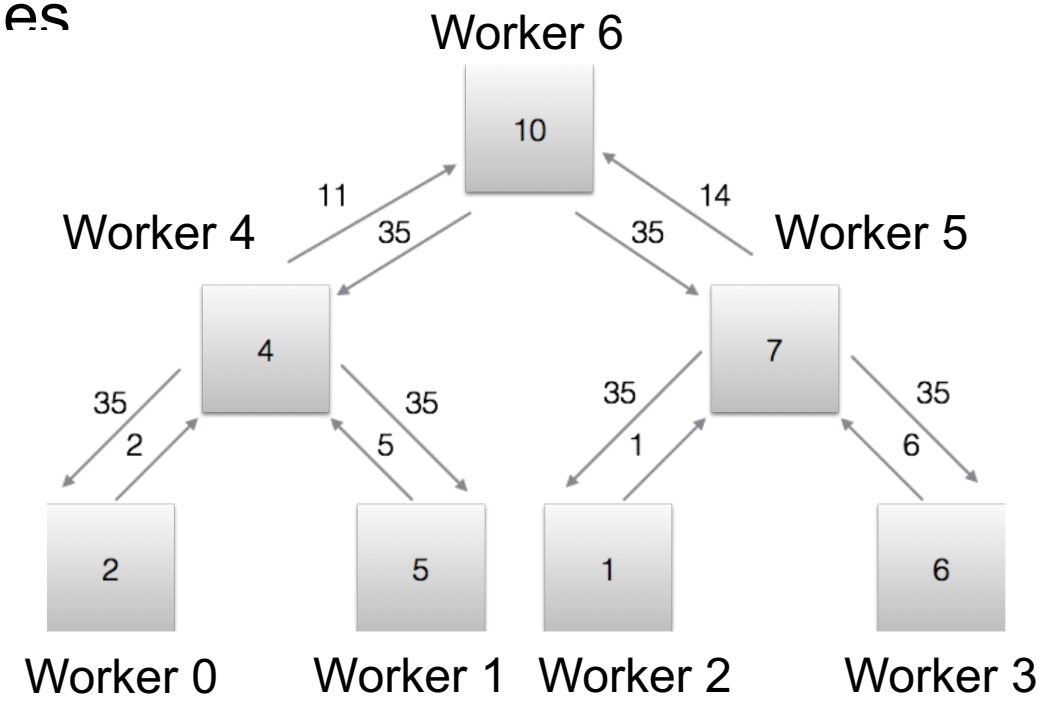


Ring AllReduce

- Construct a ring of N workers, divide M parameters into N slices
- Step 1 (Aggregation): each worker send one slice (M/N parameters) to the next worker on the ring; repeat N-1 times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N-1 times
- Overall communication: 2 * (N-1) * M parameters → 2*N*M
  - Aggregation: (N-1) * M parameters → N*M
  - Broadcast: (N-1) * M parameters → N*M

Tree AllReduce

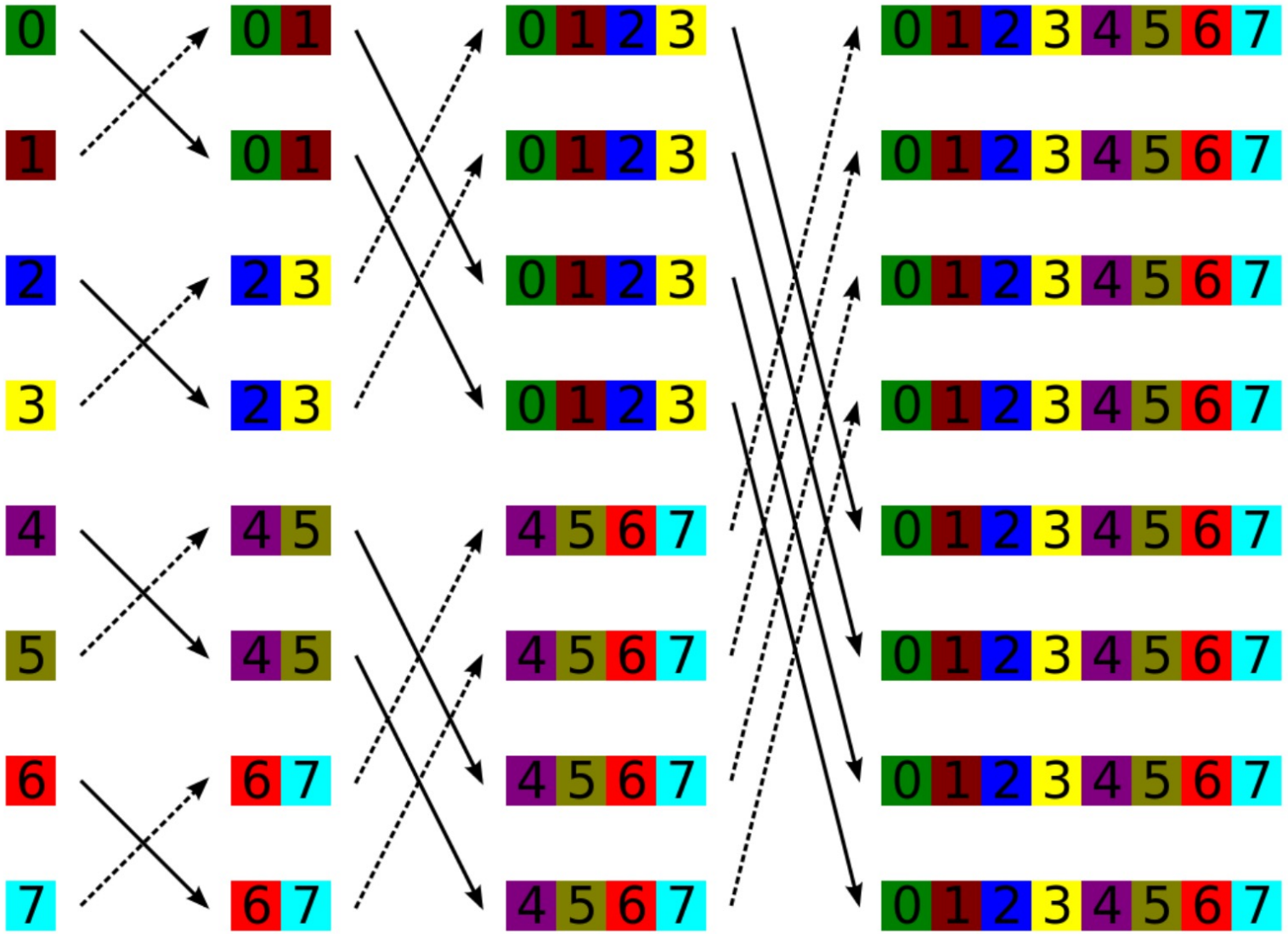
- Construct a tree of N workers;
- Step 1 (Aggregation): each worker sends M parameters to its parent; repeat log(N) times
- Step 2 (Broadcast): each worker sends M parameters to its children; repeat log(N) times



Tree AllReduce

- Construct a tree of N workers;
- Step 1 (Aggregation): each worker sends M parameters to its parent; repeat log(N) times
- Step 2 (Broadcast): each worker sends M parameters to its children; repeat log(N) times
- Overall communication: 2 * N * M parameters
  - Aggregation: M * N parameters
  - Broadcast: M * N parameters

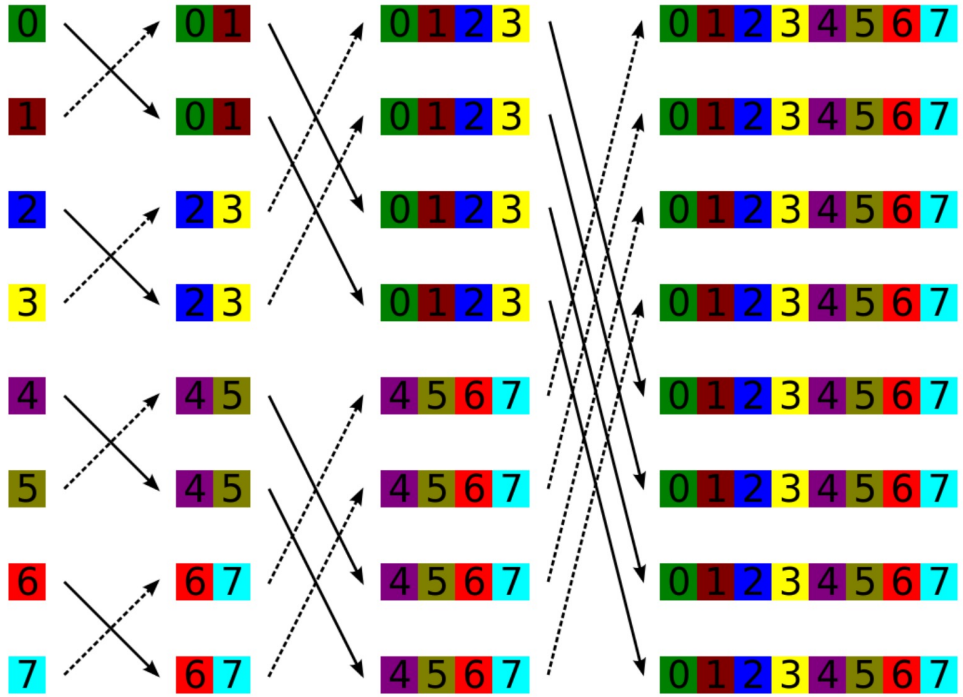
Butterfly Network



Butterfly AllReduce

- Repeat log(N) times:
  1. Each worker sends M parameters to its target node in the butterfly network
  2. Each worker aggregates gradients locally

- Overall communication: N * M * log(N) parameters



Comparing different AllReduce Methods

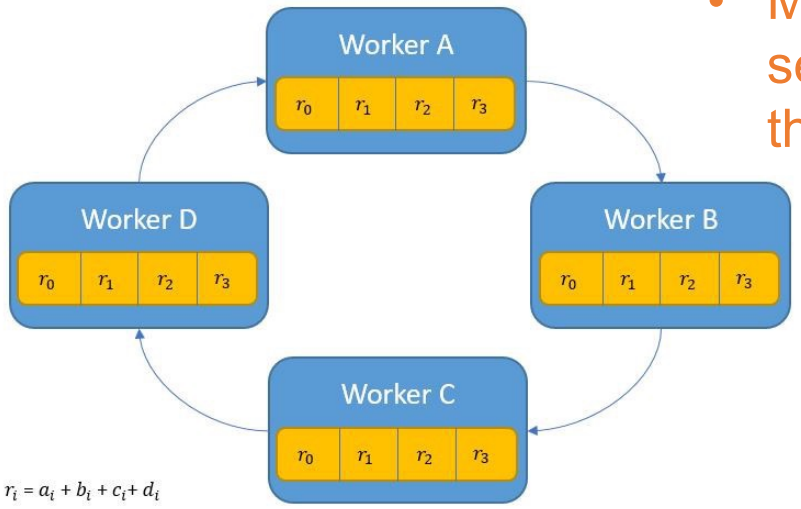
	Parameter Server	Naïve AllReduce	Ring AllReduce	Tree AllReduce	Butterfly AllReduce
Overall communication	$2\times N\times M$	$N^2\times M$	$2\times N\times M$	$2\times N\times M$	$N\times M$ $\times \log N$

Question: Ring AllReduce is more efficient and scalable then Tree AllReduce and Parameter Server, why?

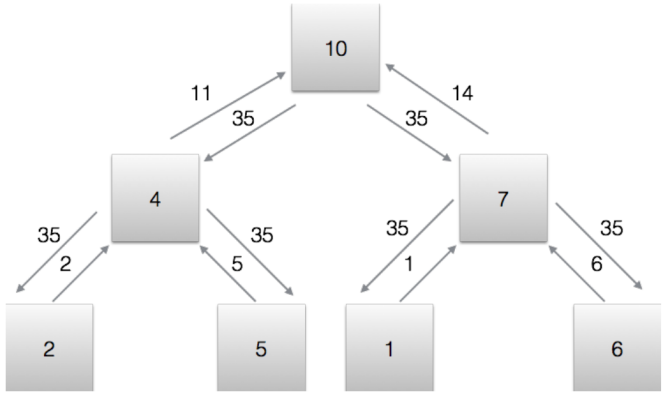
Ring AllReduce v.s. Tree AllReduce v.s. Parameter Server

Ring AllReduce:

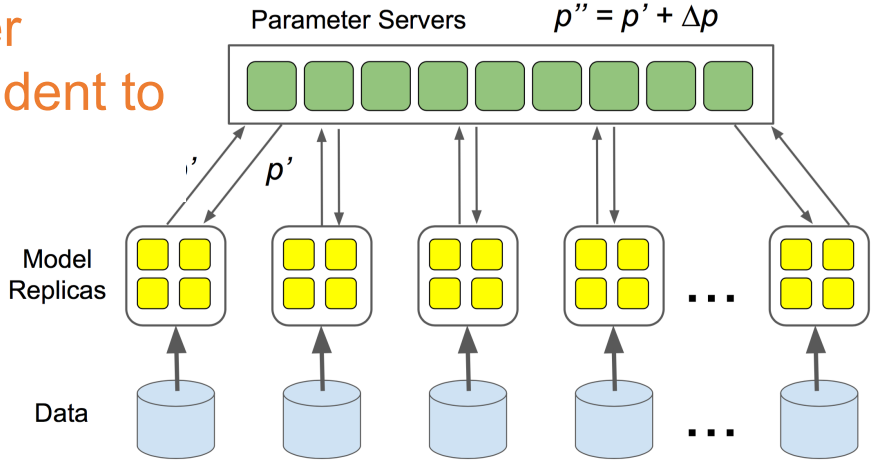
- Best latency
- Balanced workload across workers
- More scalable since each worker sends 2*M parameters (independent to the number of workers)



Each worker sends **M/N** parameters per iteration; repeat for **2*N** iterations
Latency: O(M/N * (2*N))



Each worker sends **M** parameters per iteration; repeat for **2*log(N)** iterations
Latency: O(M * 2 * log(N))



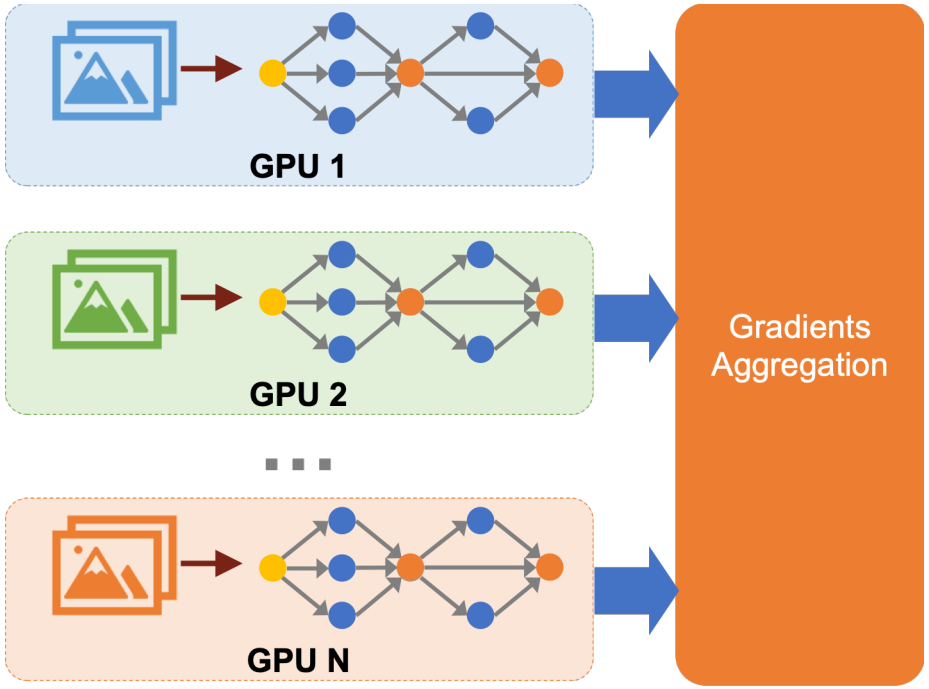
All workers send **M** parameters to parameter servers and receive **M** parameters from servers
Latency: O(2*M * N)

Recap: Data Parallelism

Each worker keeps a replica of the entire model and communicates with other workers to synchronize weights updates

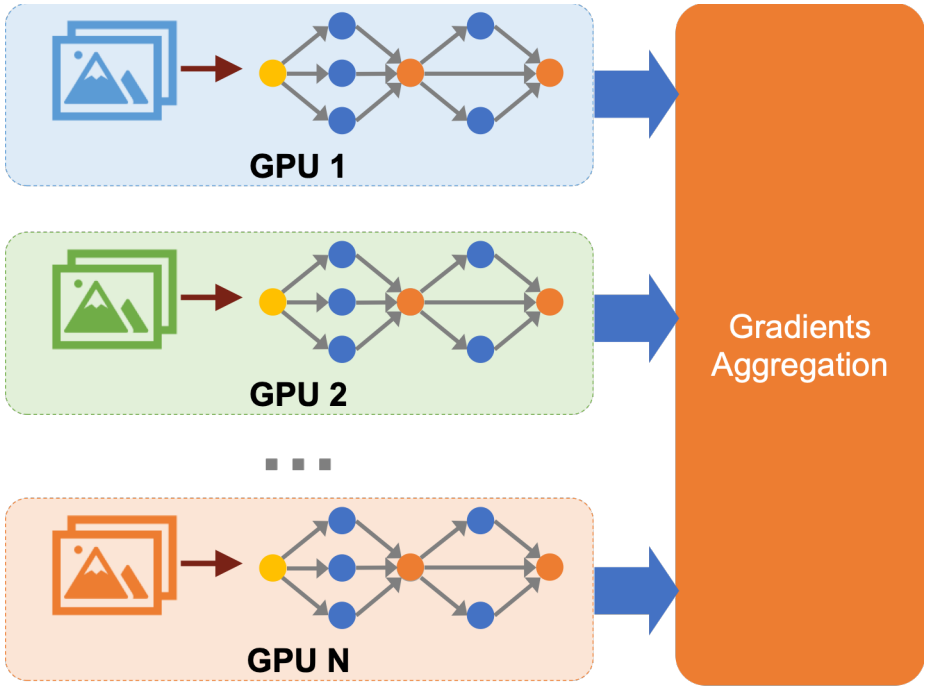
Gradients aggregation methods:

- Parameter Server
- Ring AllReduce
- Tree AllReduce
- Butterfly AllReduce
- Etc.



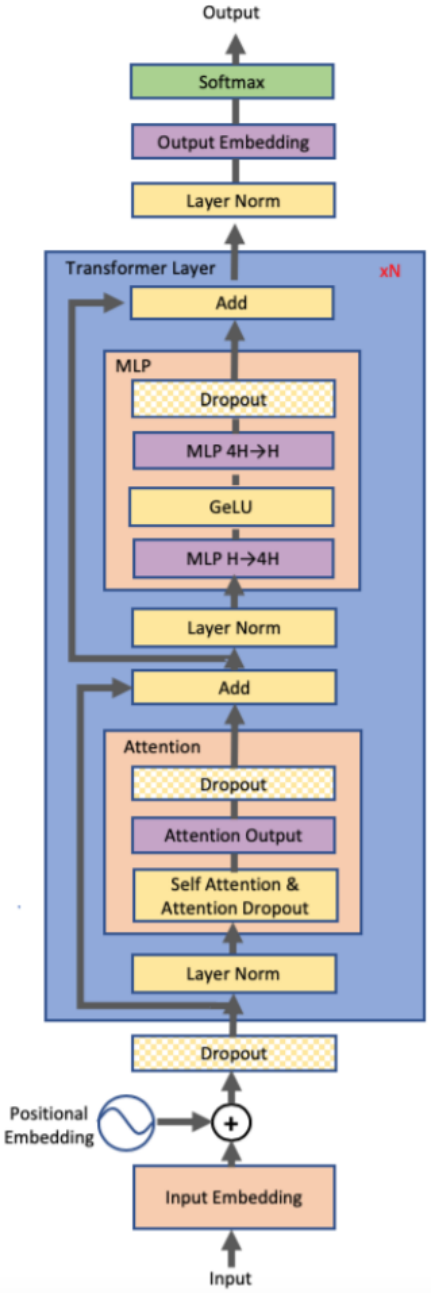
An Issue with Data Parallelism

- Each GPU saves a replica of the entire model
- Cannot train large models that exceed GPU device memory



Large Model Training Challenges

	Bert-Large	GPT-2	Turing 17.2 NLG	GPT-3
Parameters	0.32B	1.5B	17.2B	175B
Layers	24	48	78	96
Hidden Dimension	1024	1600	4256	12288
Relative Computation	1x	4.7x	**54x**	**547x**
Memory Footprint	5.12GB	24GB	**275GB**	**2800GB**

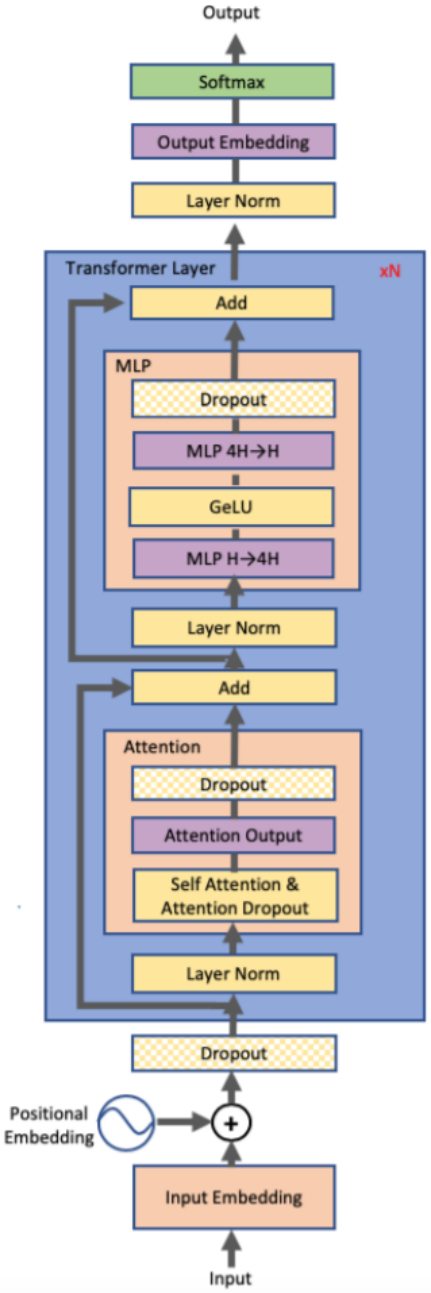


Large Model Training Challenges

	Bert-Large	GPT-2	Turing 17.2 NLG	GPT-3
Parameters	0.32B	1.5B	17.2B	175B
Layers	24	48	78	96
Hidden Dimension	1024	1600	4256	12288
Relative Computation	1x	4.7x	**54x**	**547x**
Memory Footprint	5.12GB	24GB	**275GB**	**2800GB**

NVIDIA V100 GPU memory capacity: 16G/32G
NVIDIA A100 GPU memory capacity: 40G/80G

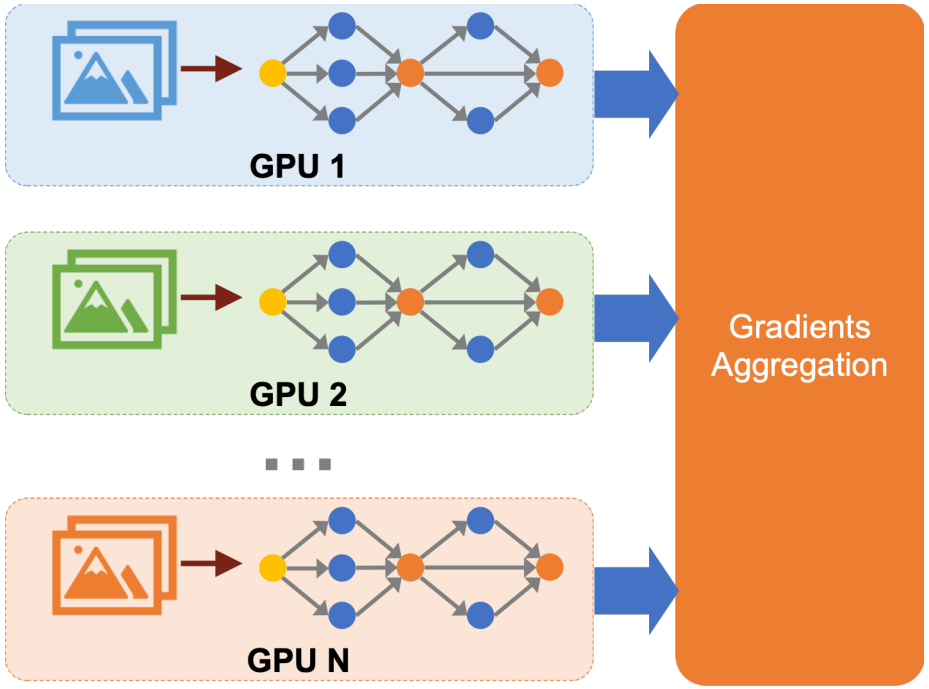
Out of Memory



ZeRO: Zero Redundancy Optimizer

deepspeed

- Eliminating data redundancy in data parallel training
- A widely used technique for data parallel training of large models



Revisit: Stocastic Gradient Descent

For t = 1 to T

Backward pass

Forward pass

$\Delta w = \eta \times \frac{1}{b} \sum_{i=1}^{b} \nabla \left(loss\left(f_w(x_i, y_i) \right) \right)$ // compute derivative and update

w -= Δw // apply update

End

Adapted from Minjia Zhang, DeepSpeed Presentation

Adaptive Learning Rates (Adam)

For t = 1 to T

$g = \frac{1}{b} \sum_{i=1}^{b} \nabla \left(loss\left(f_w(x_i, y_i) \right) \right)$

Δw = adam(g)

w -= Δw // apply update

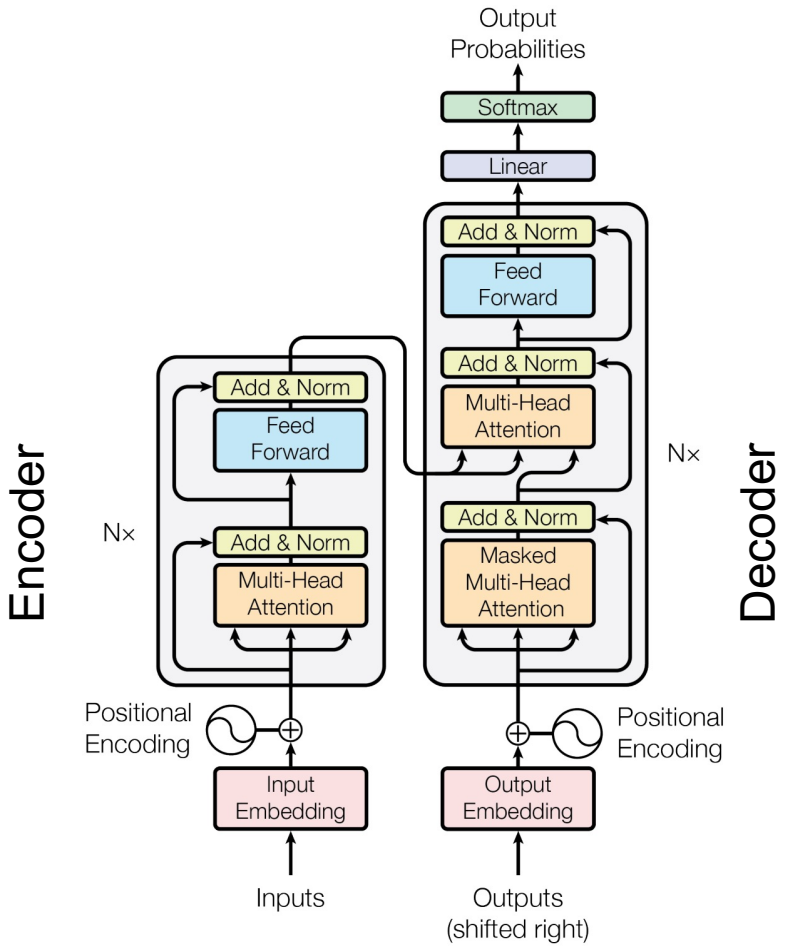
End

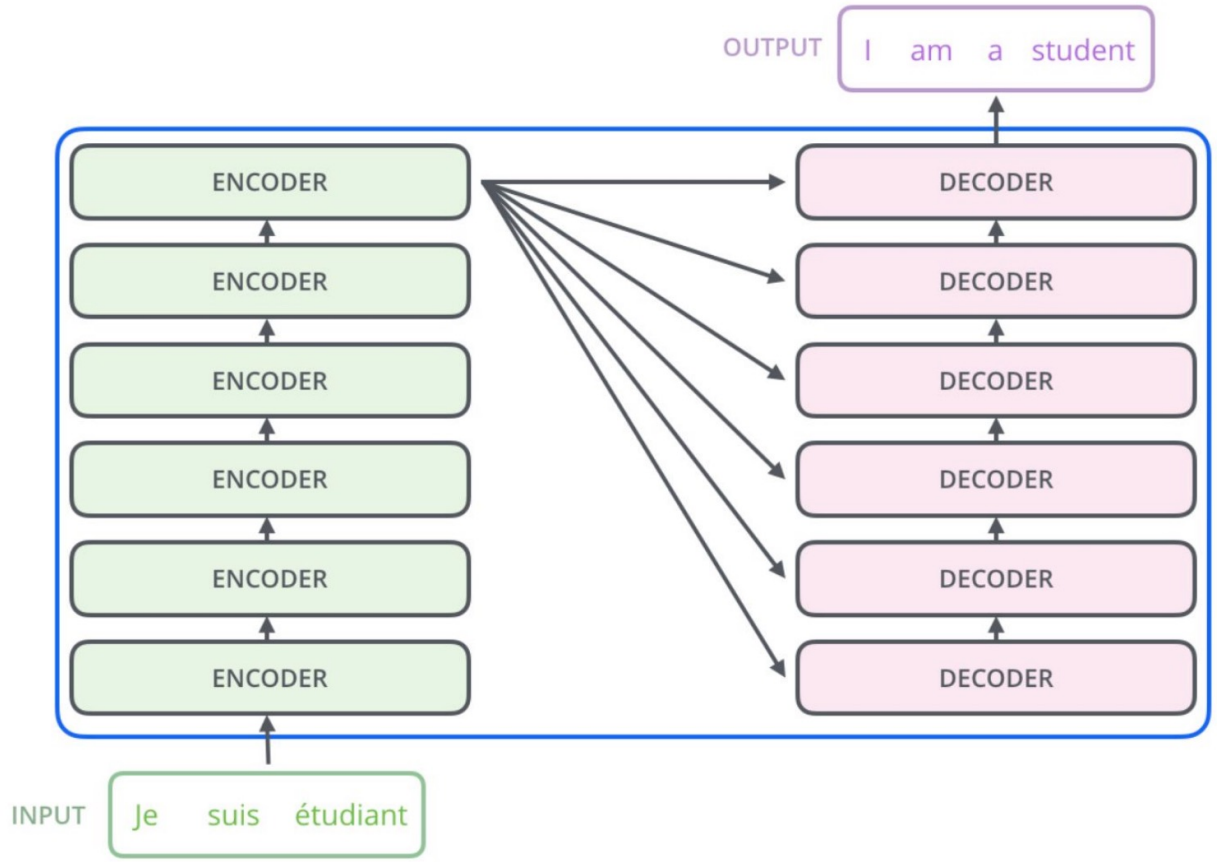
$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$

g_t : *Gradient at time t along* ω^j
ν_t : *Exponential Average of gradients along* ω_j
s_t : *Exponential Average of squares of gradients along* ω_j
β_1, β_2 : *Hyperparameters*

[1] Kingma and Ba, “Adam: A Method for Stochastic Optimization”, 2014, https://arxiv.org/abs/1412.6980

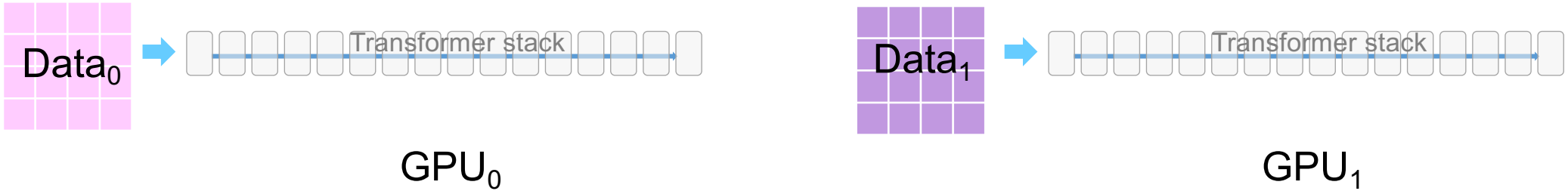
Recall: Transformer for Language Models





Ashish Vaswani et. al. Attention is all you need.

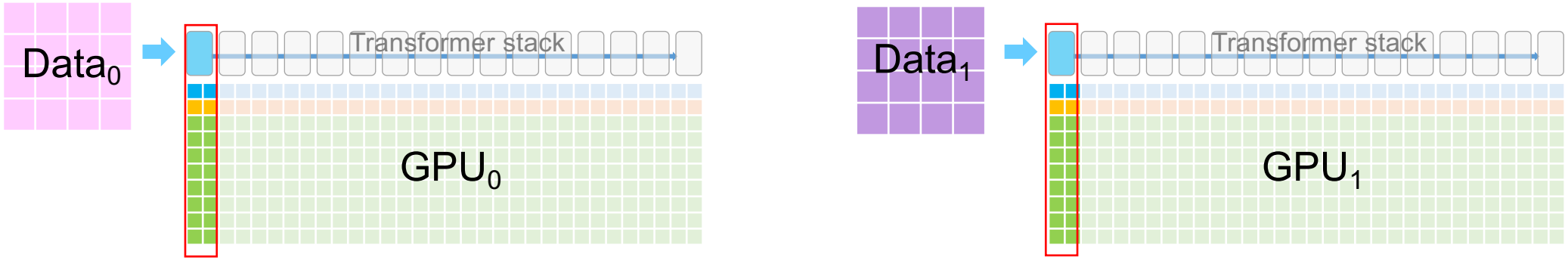
Understanding Memory Consumption



A 16-layer transformer model = 1 layer

Adapted from Minjia Zhang, DeepSpeed Presentation

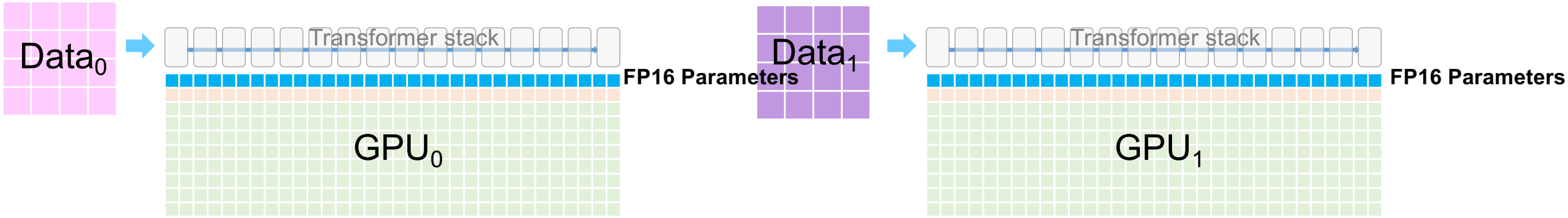
Understanding Memory Consumption



Each cell represents GPU memory used by its corresponding transformer layer

Adapted from Minjia Zhang, DeepSpeed Presentation

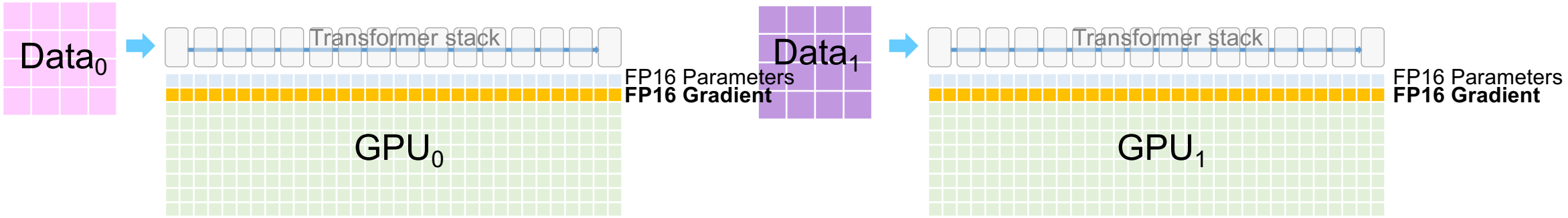
Understanding Memory Consumption



- FP16 parameter

Adapted from Minjia Zhang, DeepSpeed Presentation

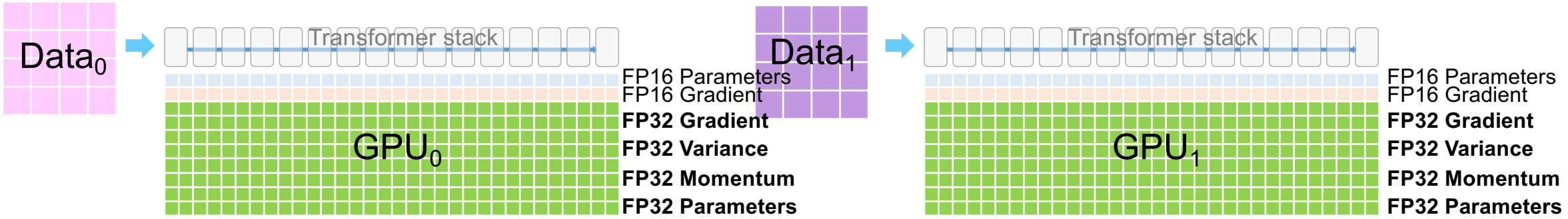
Understanding Memory Consumption



- FP16 parameter
- FP16 Gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

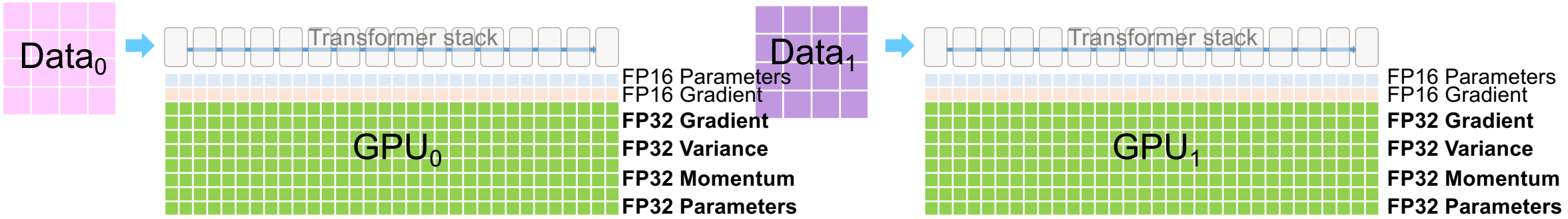
Understanding Memory Consumption



- FP16 parameter
- FP16 Gradients
- FP32 Optimizer States
  - Gradients, Variance, Momentum, Parameters

Adapted from Minjia Zhang, DeepSpeed Presentation

Understanding Memory Consumption



- FP16 parameter **: 2M bytes**
- FP16 Gradients **: 2M bytes**
- FP32 Optimizer States **: 16M bytes**
  - Gradients, Variance, Momentum, Parameters

M = number of parameters in the model

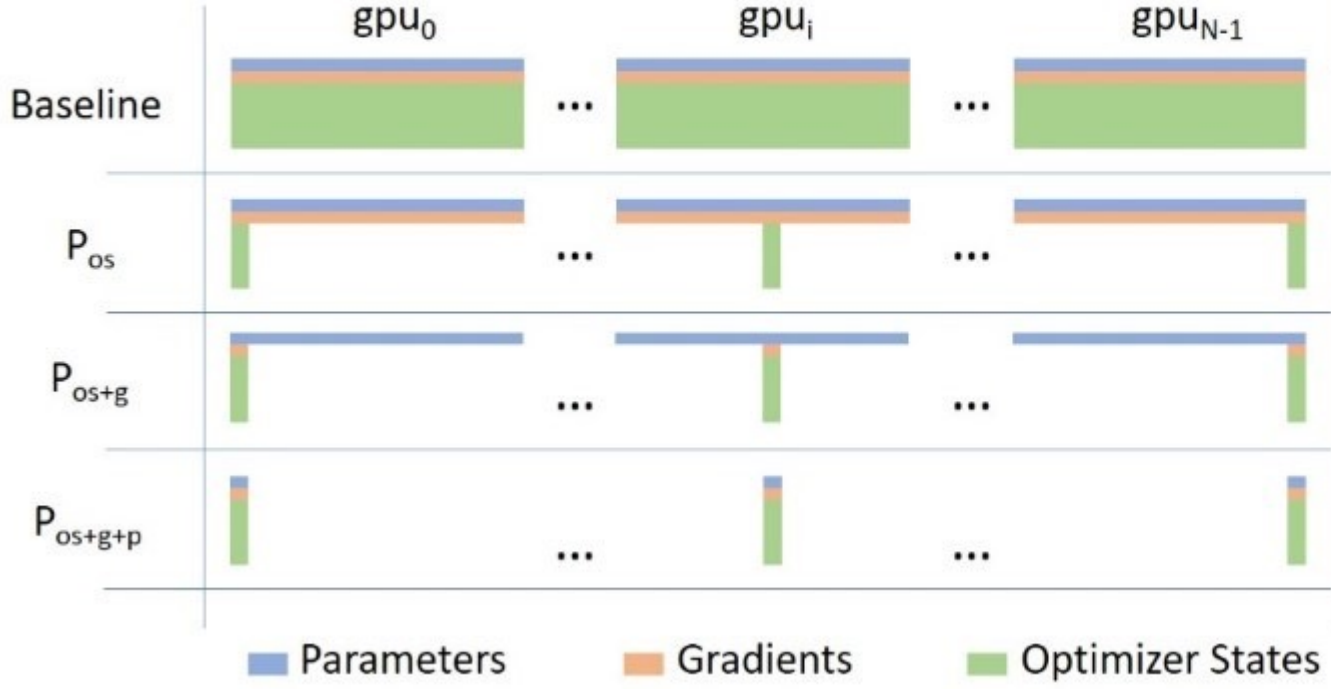
Example 1B parameter model -> 20GB/GPU

Memory consumption doesn't include:
- Input batch + activations

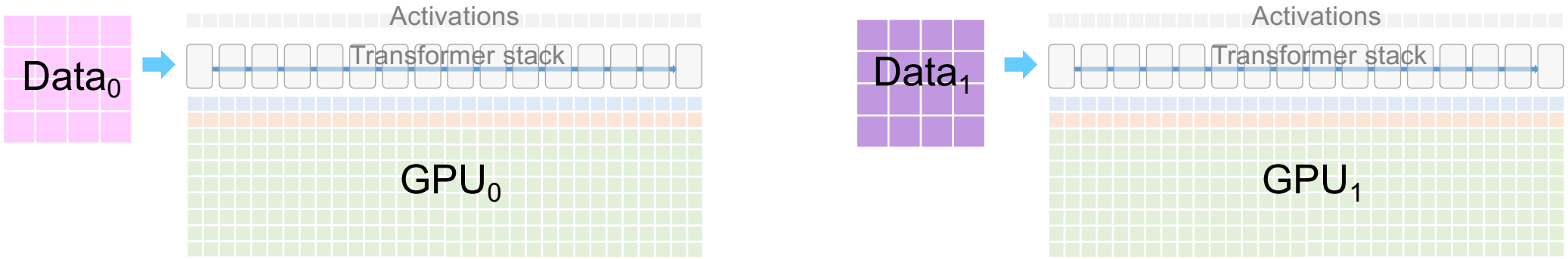
Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO-DP: ZeRO powered Data Parallelism

- ZeRO removes the redundancy across data parallel process
- Stage 1: partitioning optimizer states
- Stage 2: partitioning gradients
- Stage 3: partitioning parameters



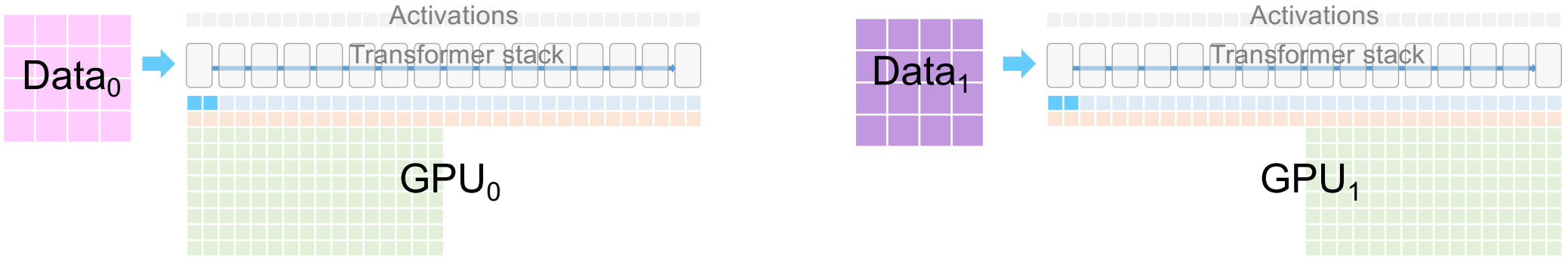
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1

Adapted from Minjia Zhang, DeepSpeed Presentation

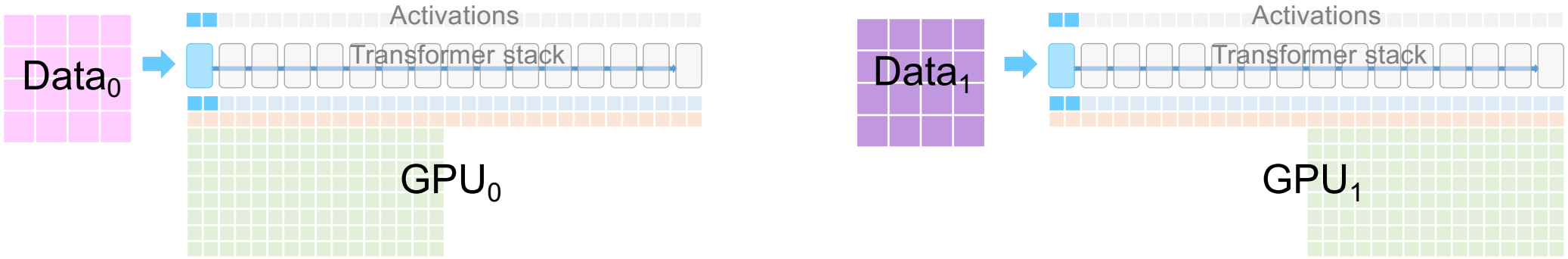
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

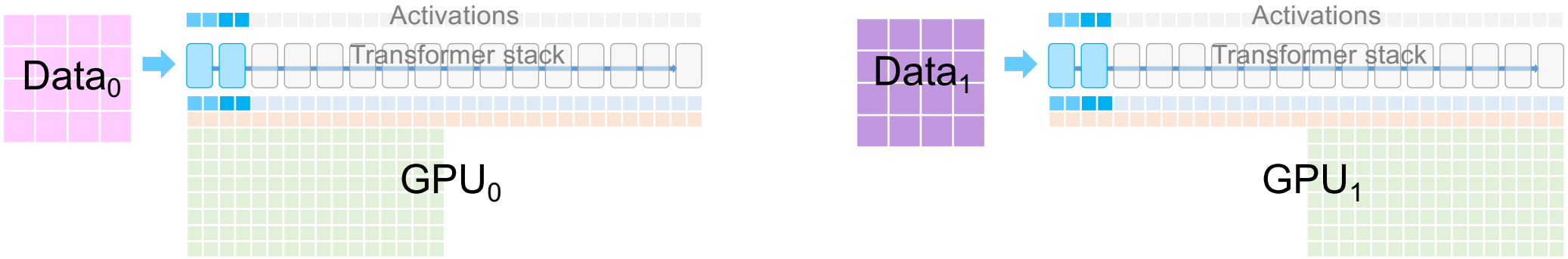


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks



Adapted from Minjia Zhang, DeepSpeed Presentation

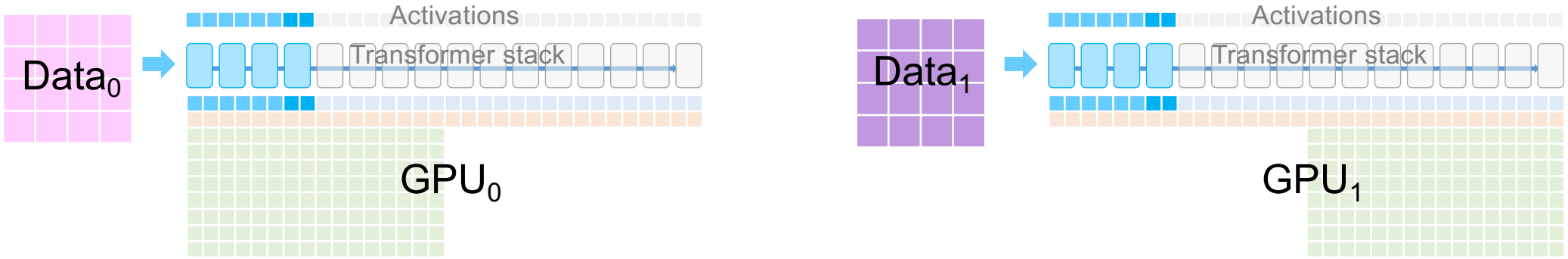
ZeRO Stage 1: Partitioning Optimizer States



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

Adapted from Minjia Zhang, DeepSpeed Presentation

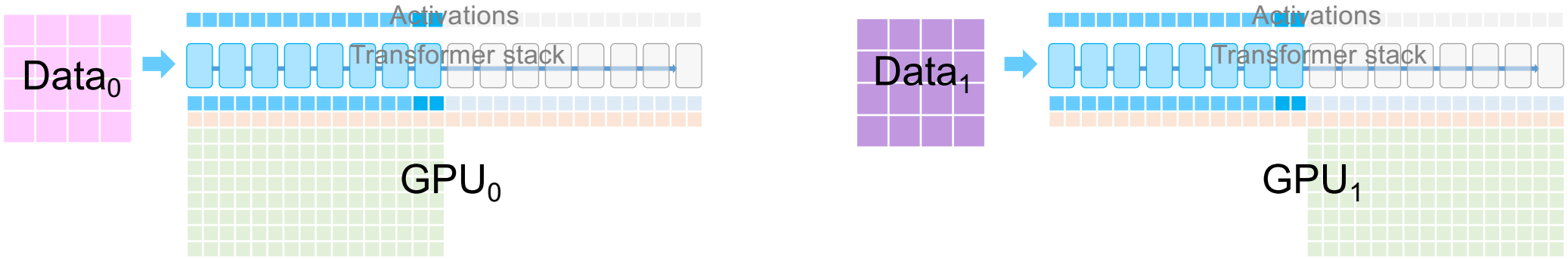
ZeRO Stage 1: Partitioning Optimizer States



- 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

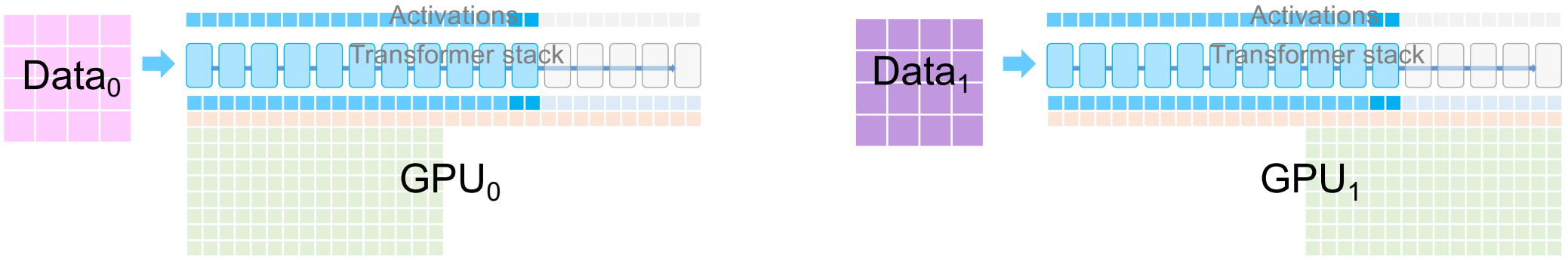


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks



Adapted from Minjia Zhang, DeepSpeed Presentation

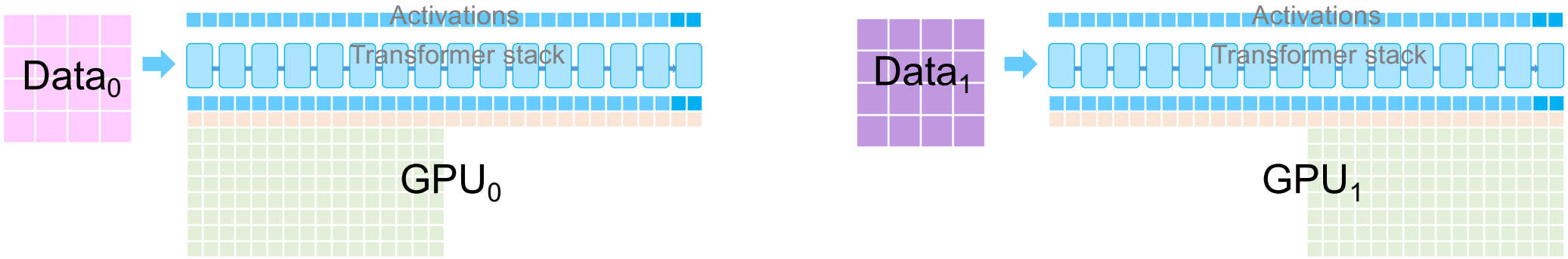
ZeRO Stage 1: Partitioning Optimizer States



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

Adapted from Minjia Zhang, DeepSpeed Presentation

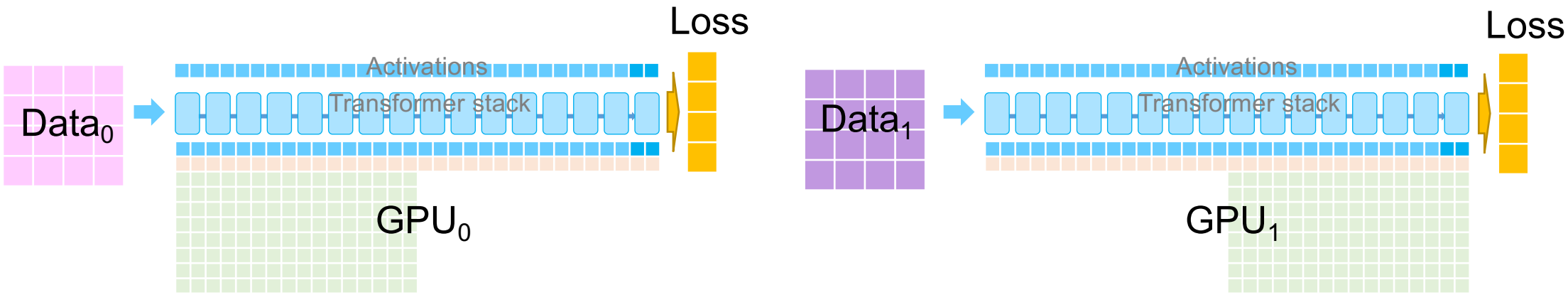
ZeRO Stage 1: Partitioning Optimizer States



- 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

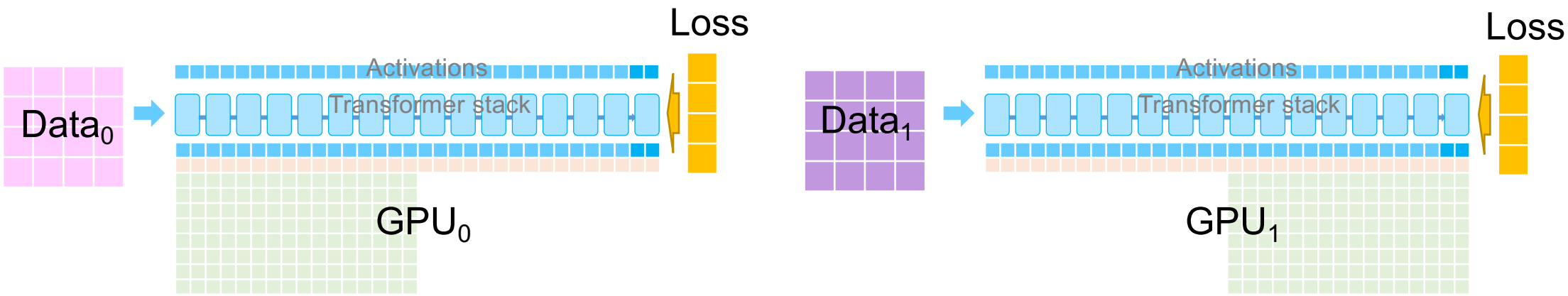


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

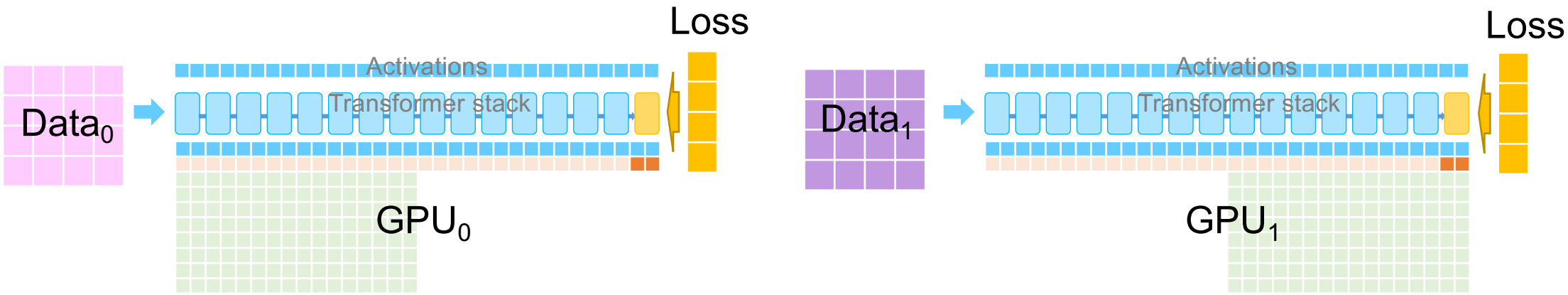


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

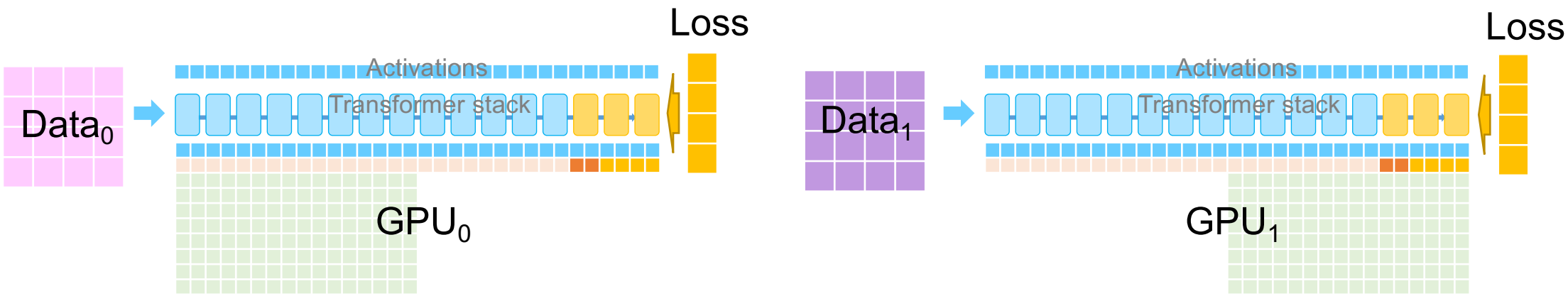


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

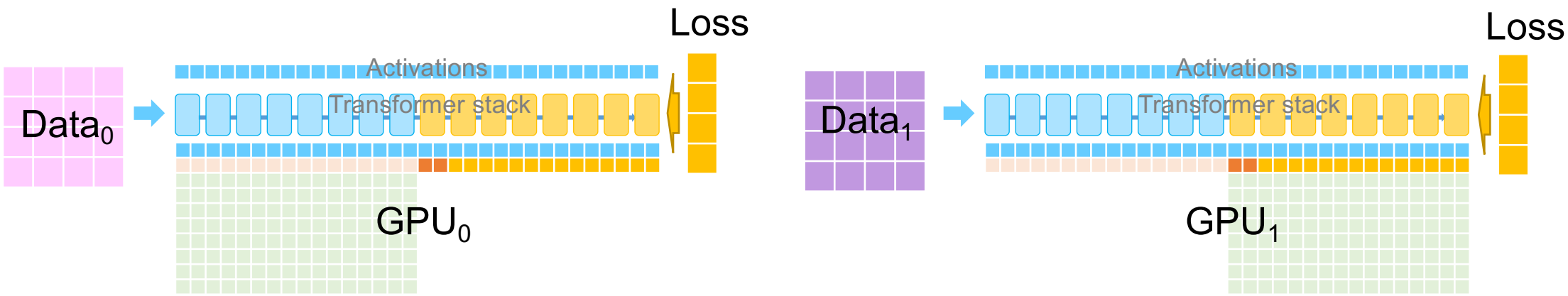


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

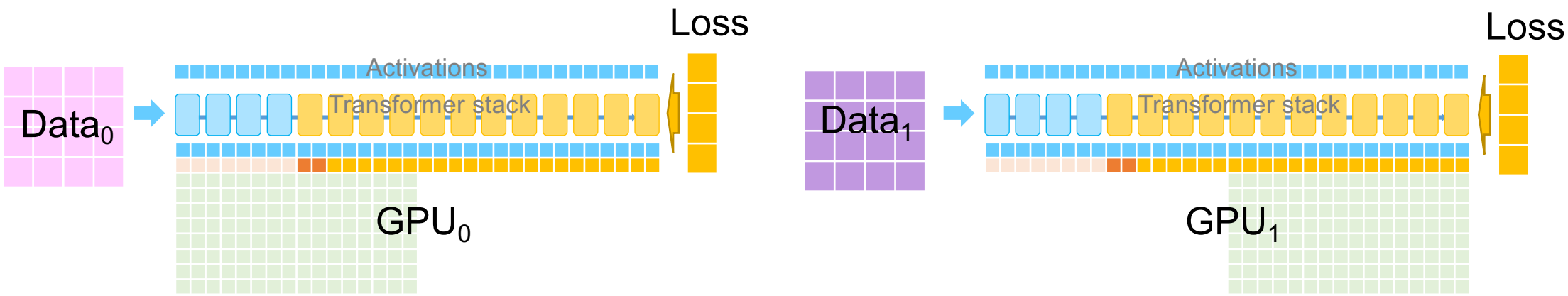


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

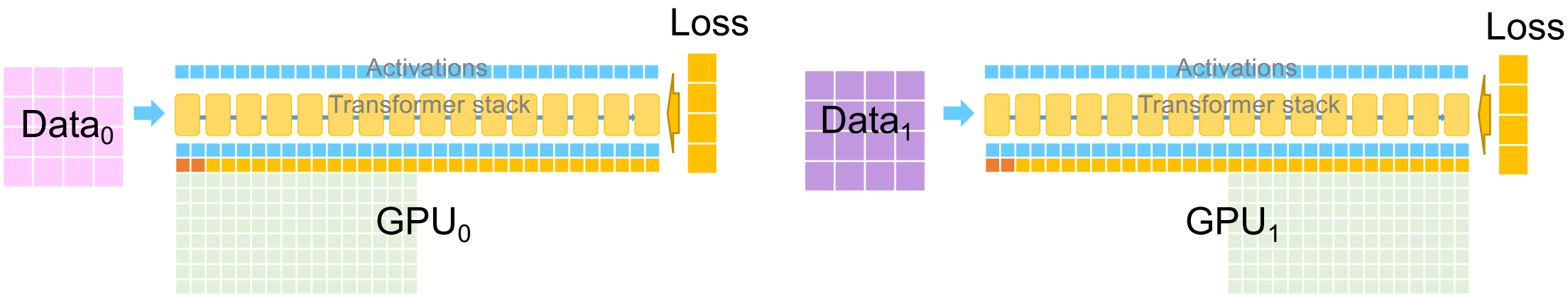


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

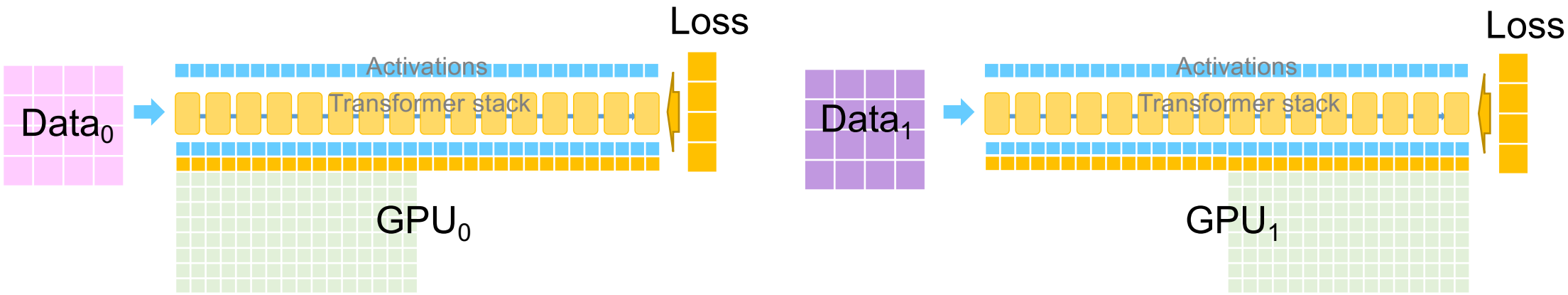


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

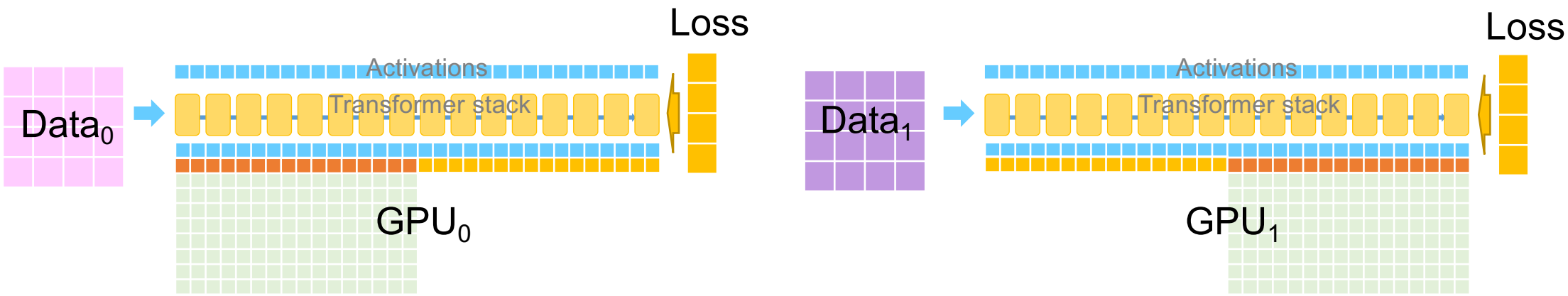


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

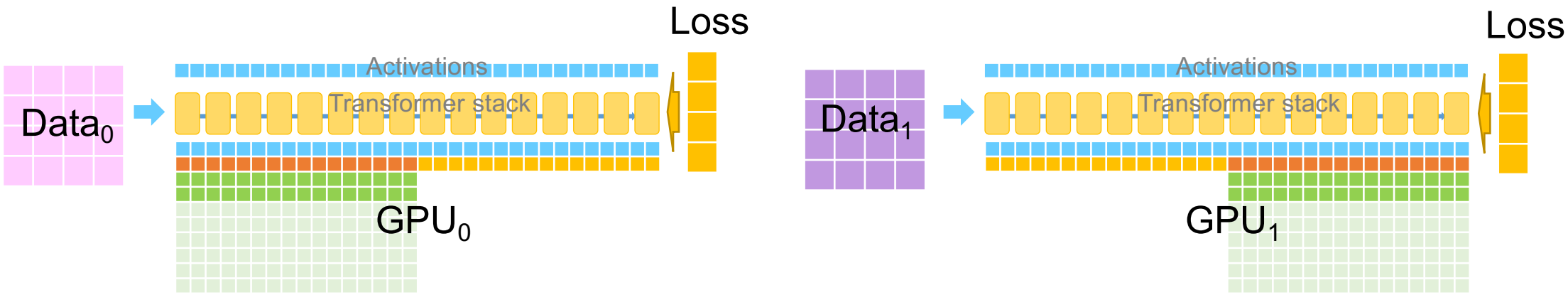


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

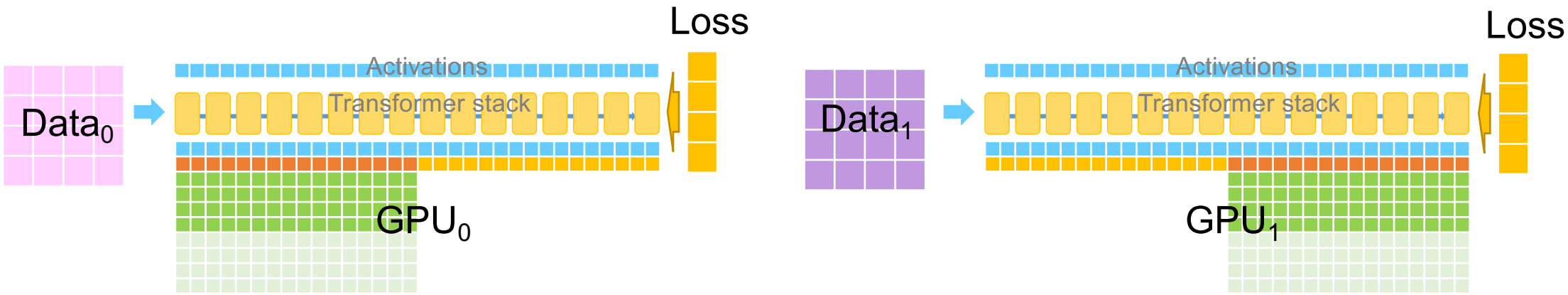


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

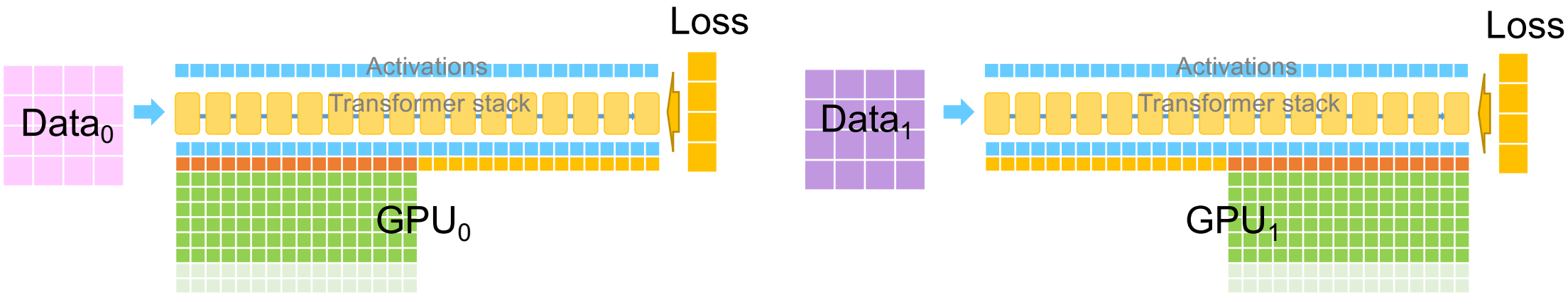


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

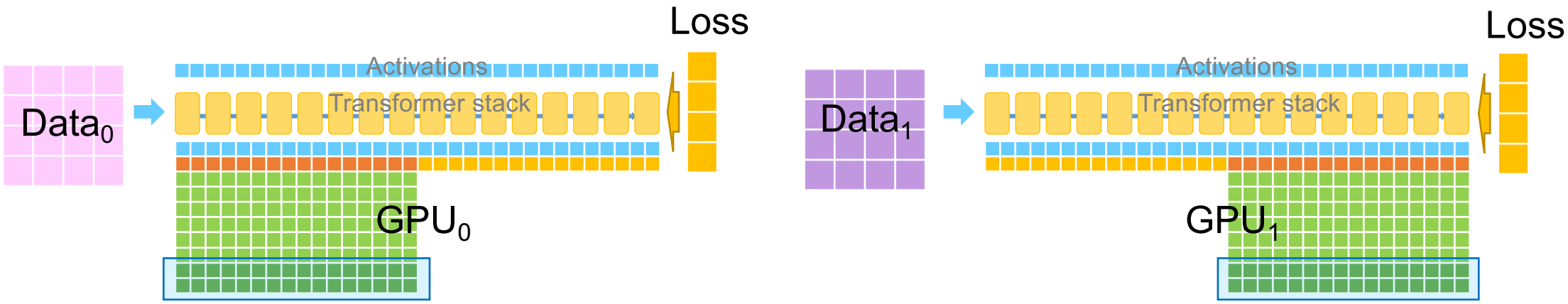


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

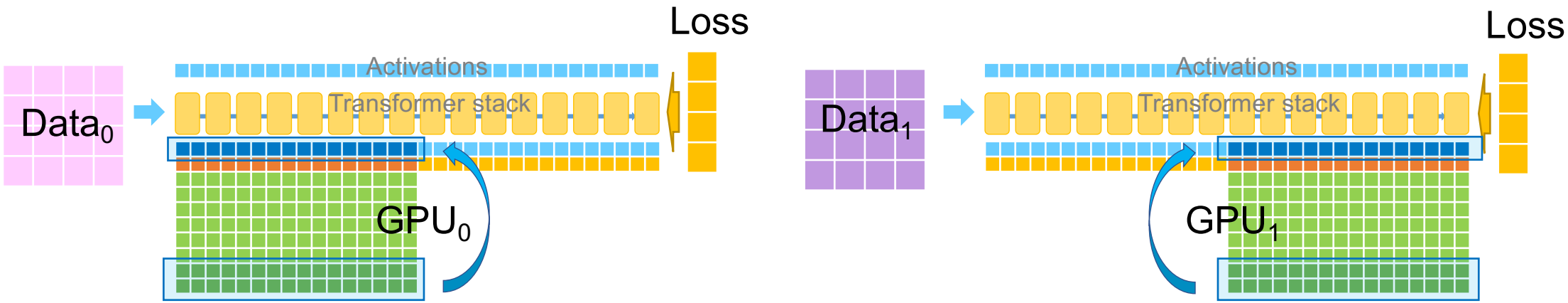


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

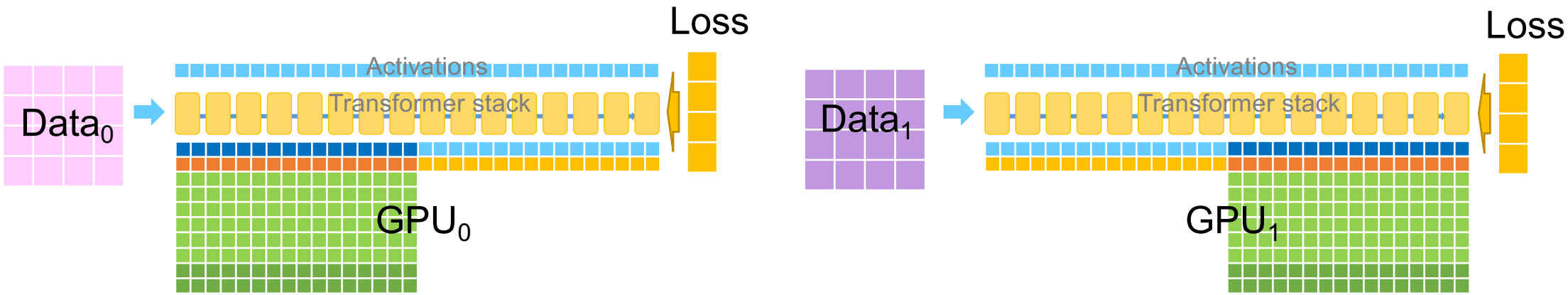
ZeRO Stage 1: Partitioning Optimizer States



- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce 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

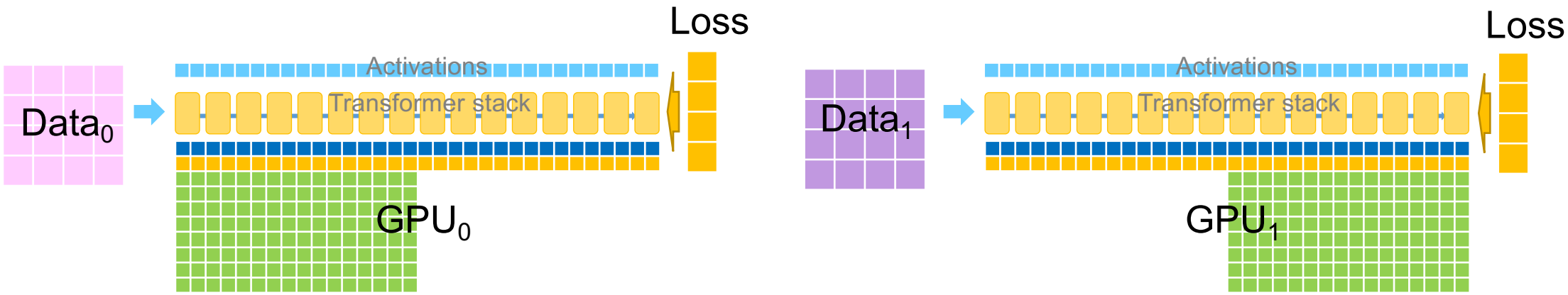


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer
- Update the FP16 weights
- All Gather the FP16 weights to complete the iteration

Parameters Gradients Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

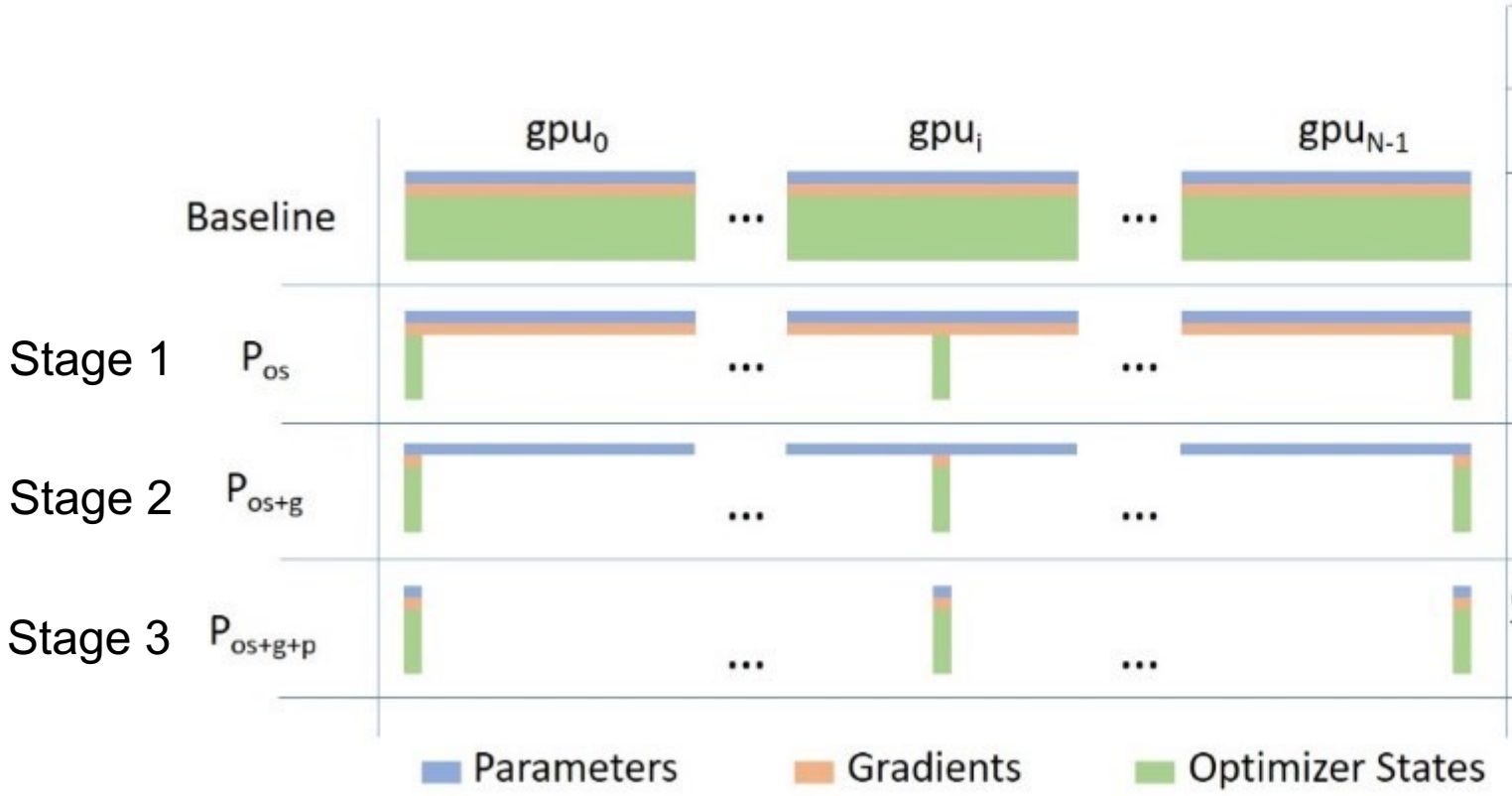
ZeRO Stage 1: Partitioning Optimizer States



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

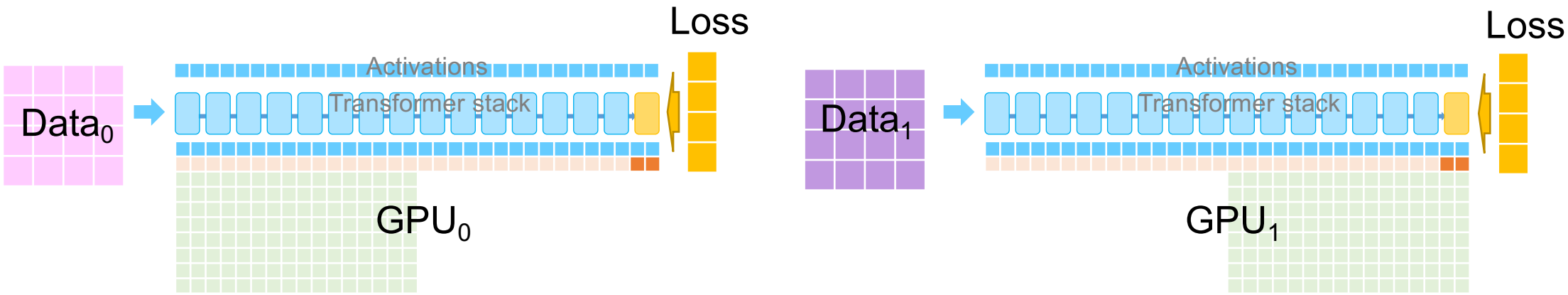
ZeRO: Zero Redundancy Optimizer

- Progressive memory savings and communication volume
- Turning NLR 17.2B is powered by Stage 1 and Megatron



	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

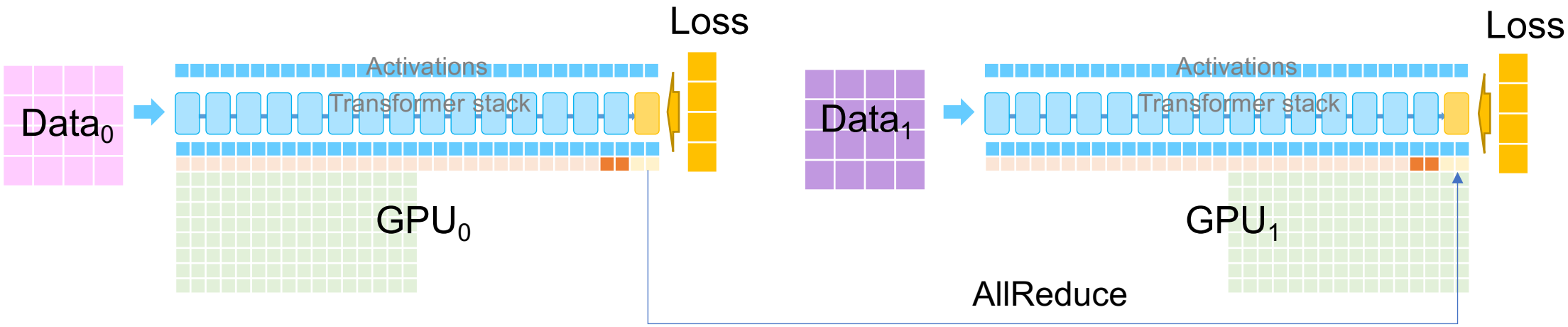
ZeRO Stage 2: Partitioning Gradients



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

Parameters Gradients Optimizer States

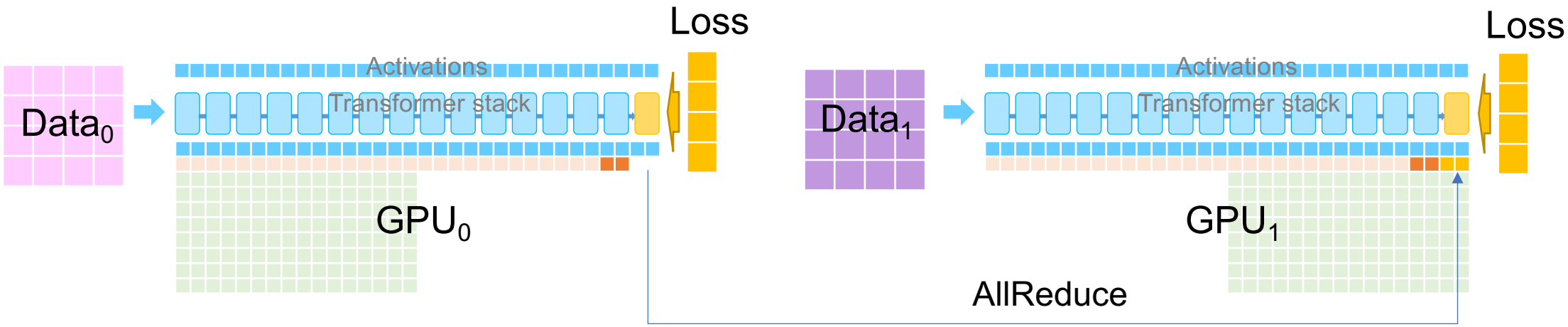
ZeRO Stage 2: Partitioning Gradients



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

Parameters Gradients Optimizer States

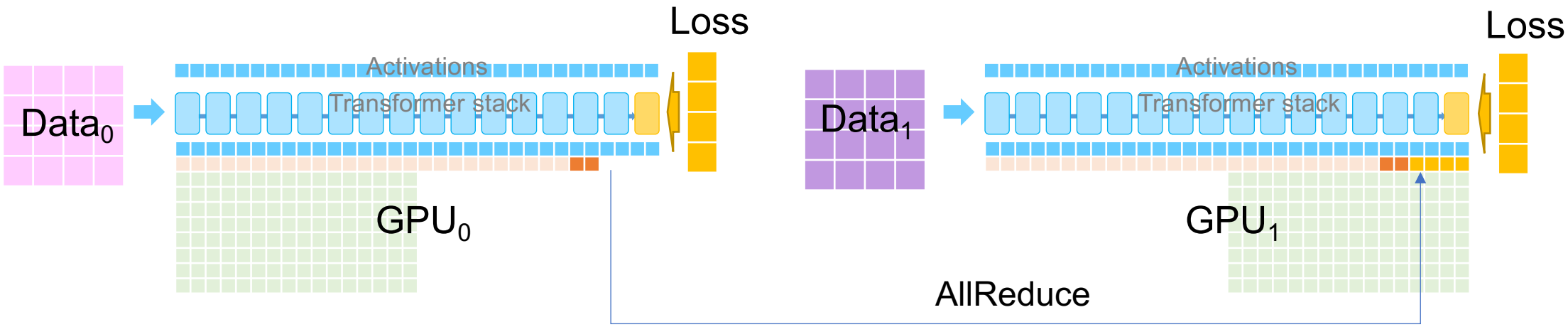
ZeRO Stage 2: Partitioning Gradients



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

Parameters Gradients Optimizer States

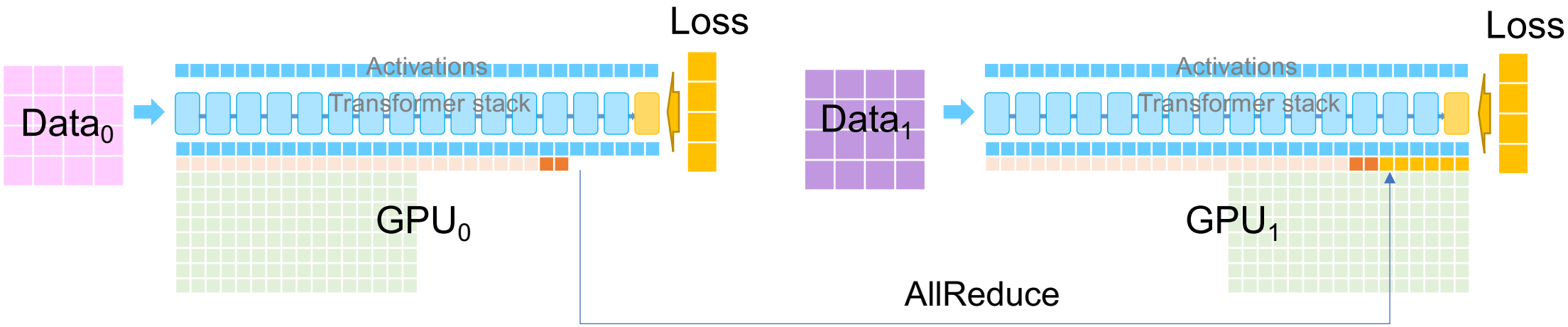
ZeRO Stage 2: Partitioning Gradients



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

Parameters Gradients Optimizer States

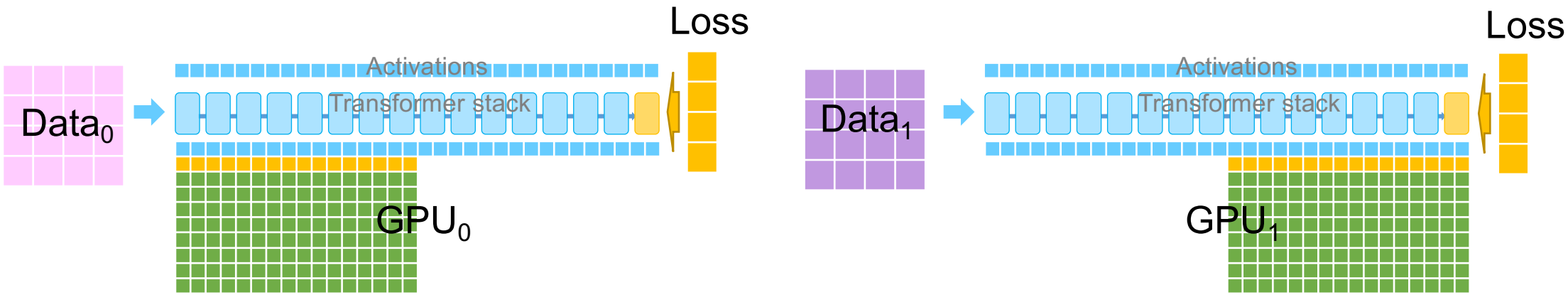
ZeRO Stage 2: Partitioning Gradients



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

Parameters Gradients Optimizer States

ZeRO Stage 2: Partitioning Gradients

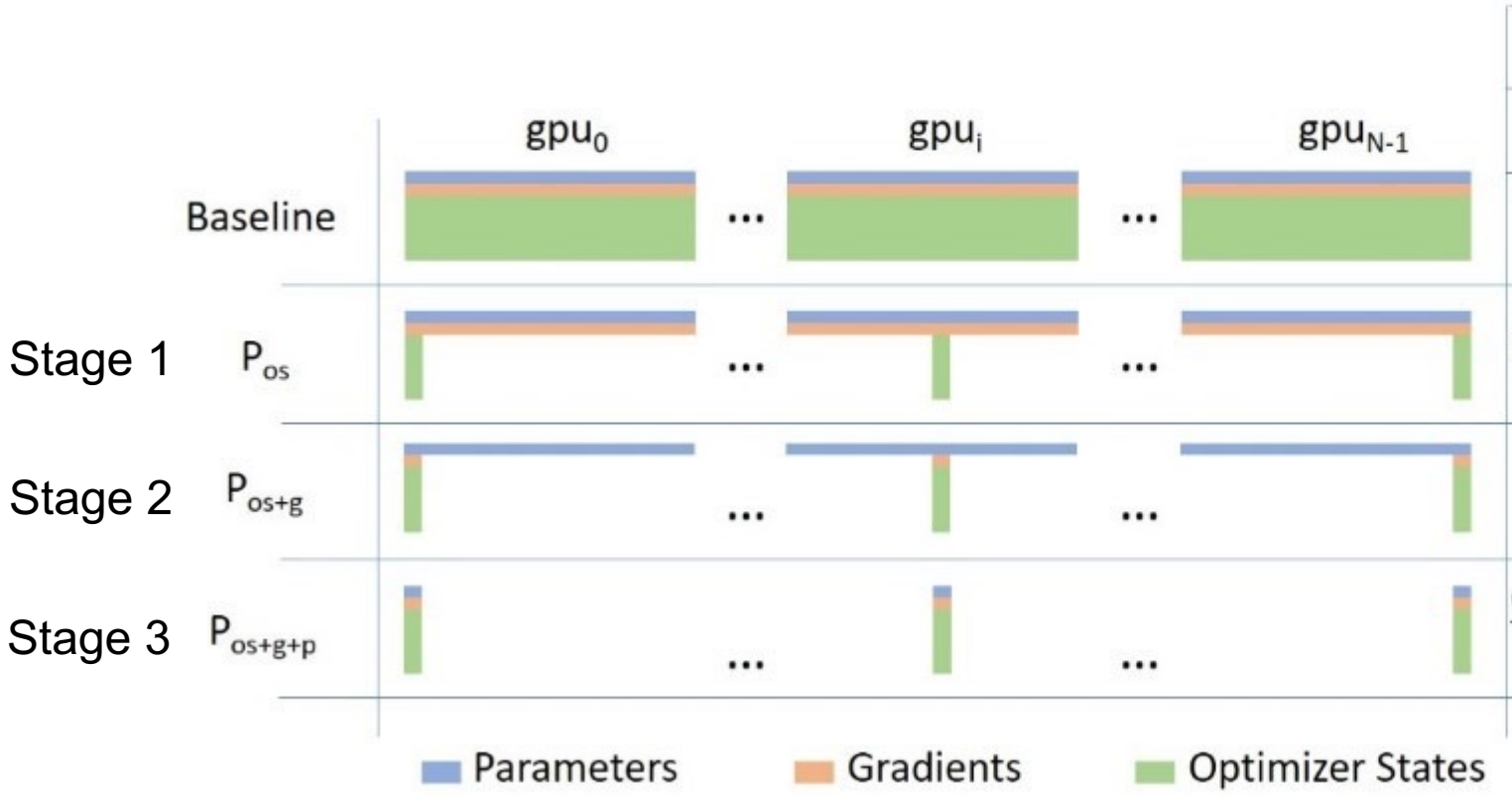


- Partitioning gradients across GPUs
- Reduce gradients on GPUs responsible for updating parameters

Parameters Gradients Optimizer States

ZeRO: Zero Redundancy Optimizer

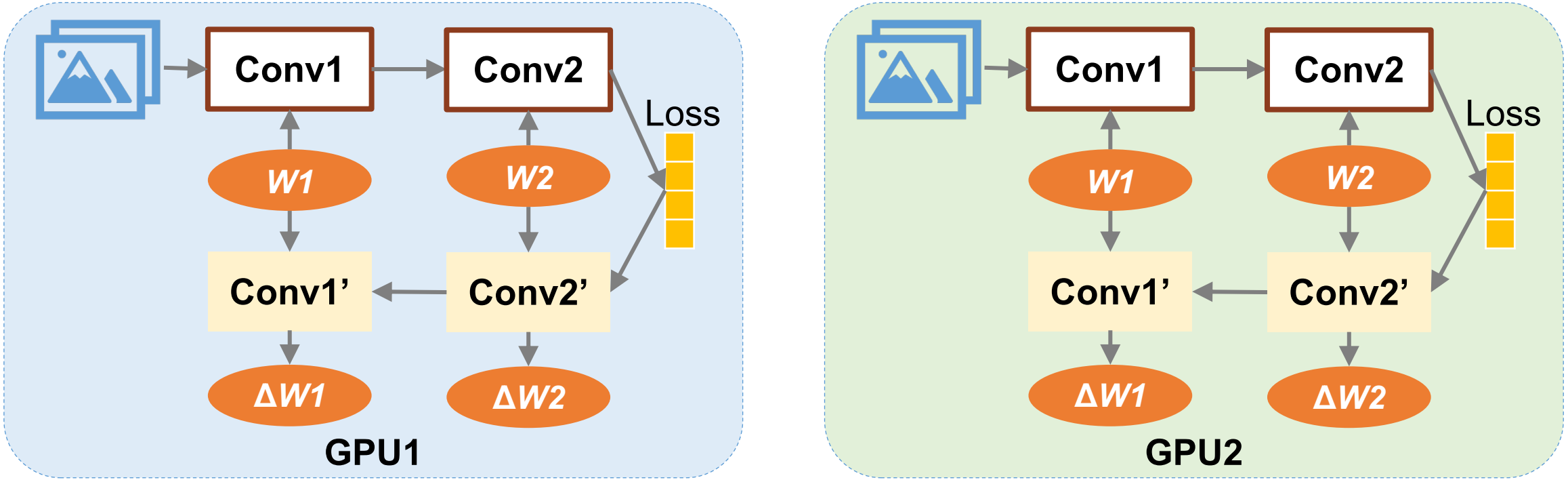
- Progressive memory savings and communication volume
- Turning NLR 17.2B is powered by Stage 1 and Megatron



	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

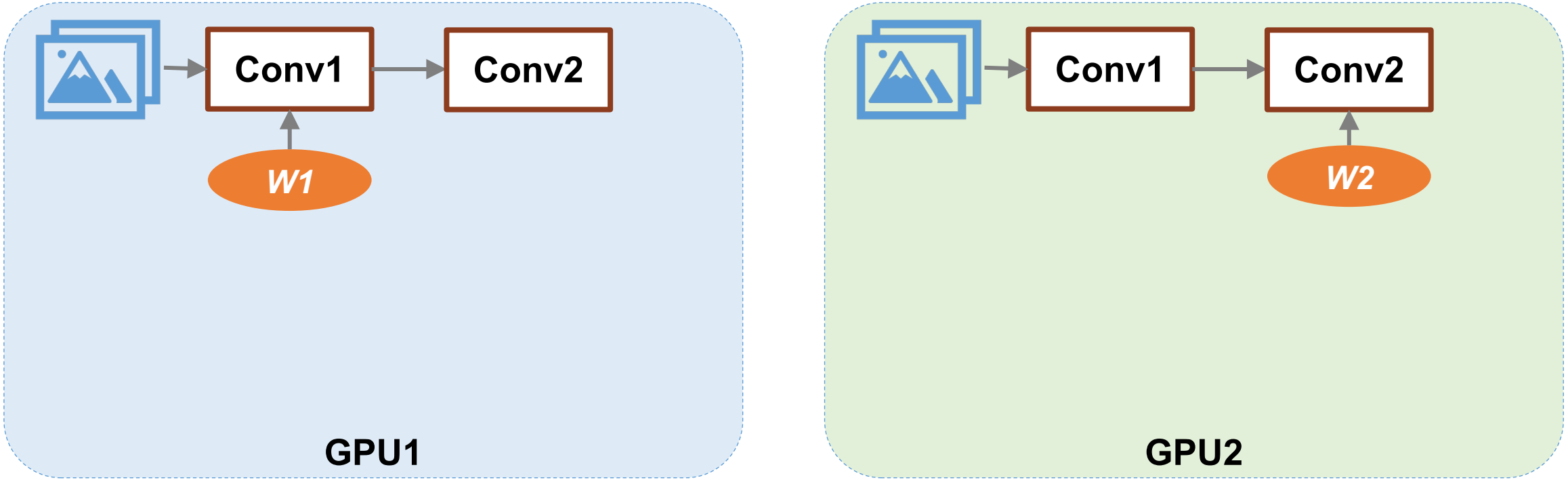
ZeRO Stage 3: Partitioning Parameters

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



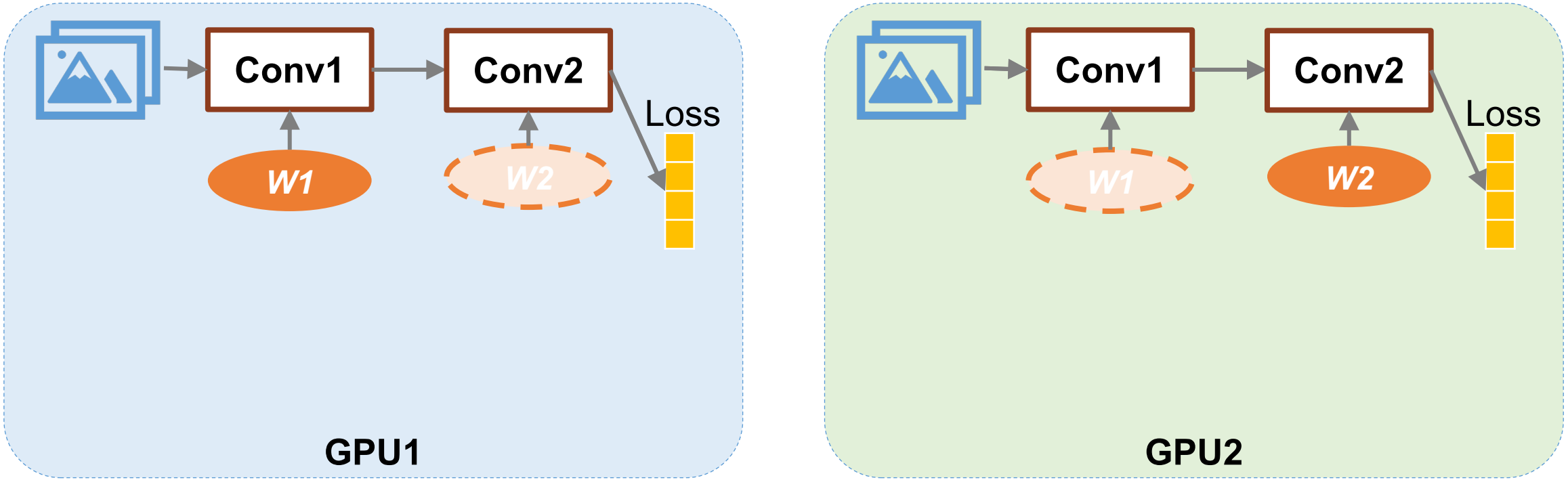
ZeRO Stage 3: Partitioning Parameters

- In ZeRO, model parameters are partitioned across GPUs



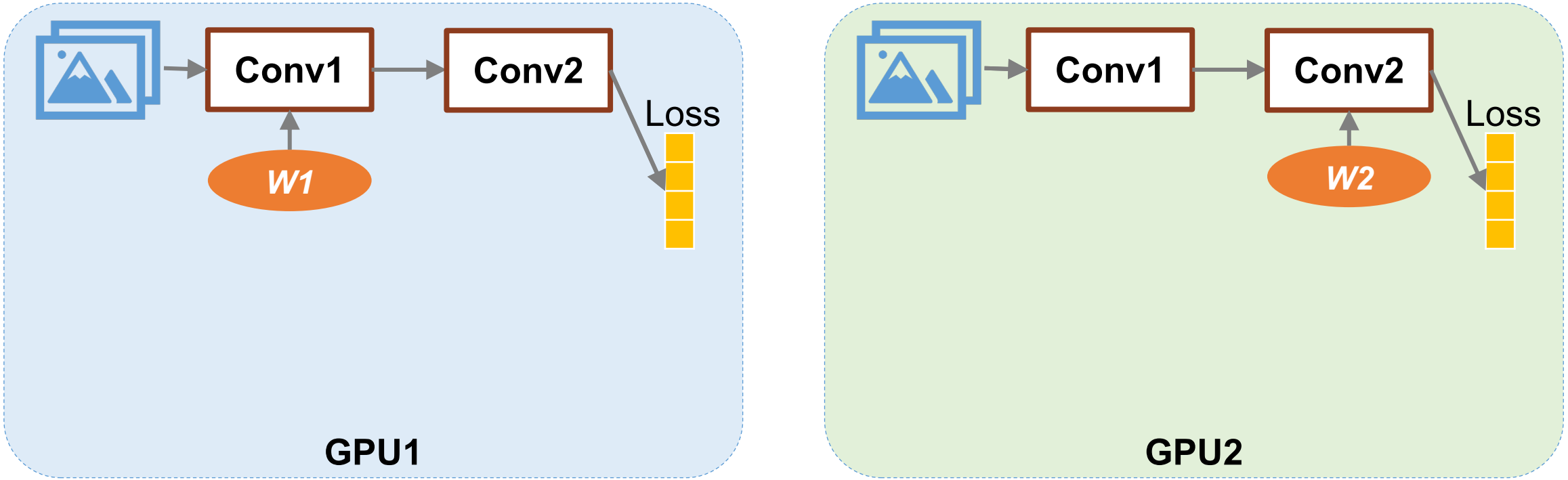
ZeRO Stage 3: Partitioning Parameters

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



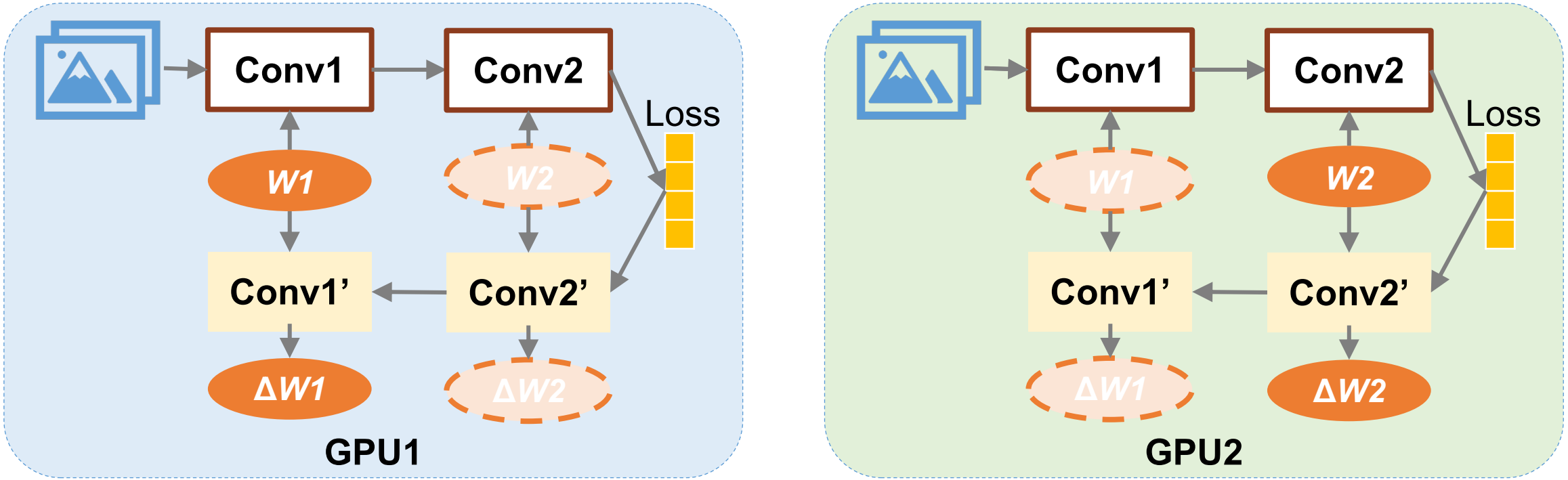
ZeRO Stage 3: Partitioning Parameters

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



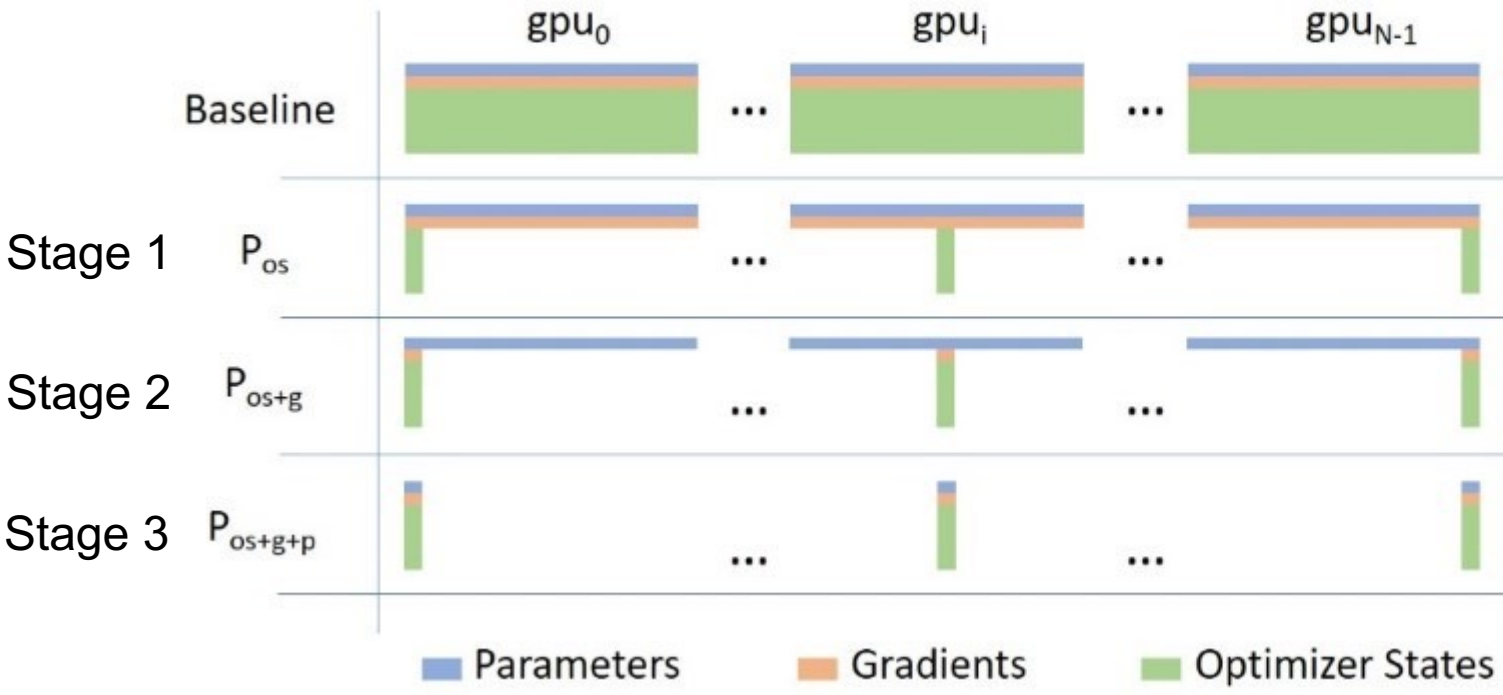
ZeRO Stage 3: Partitioning Parameters

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



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

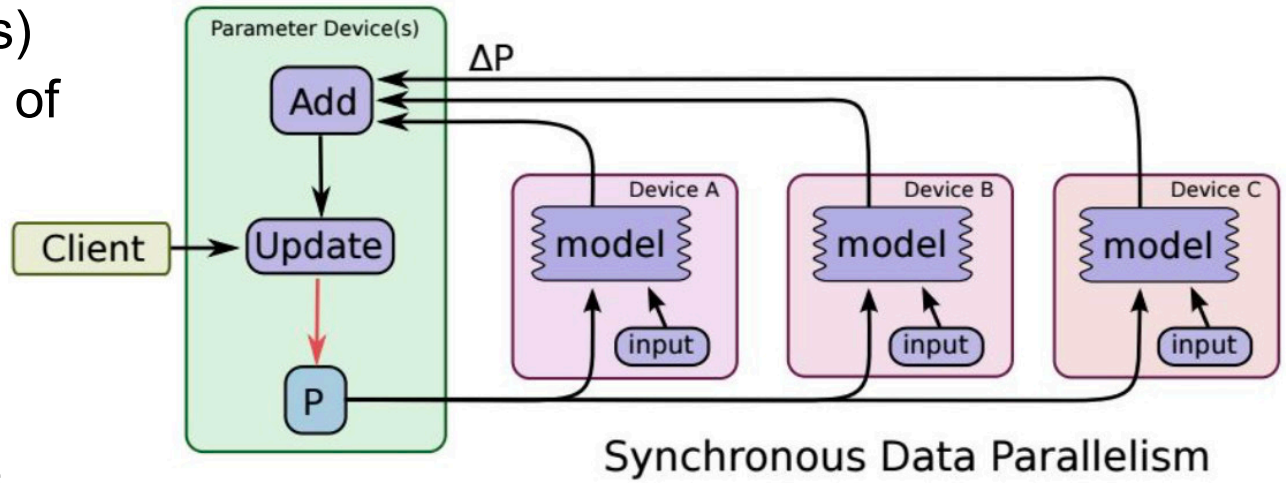
Summary

- Data-parallel training
  - Parameter server
  - Ring AllReduce
  - Tree AllReduce
  - Butterfly AllReduce
- ZeRO: zero redundancy optimizer

Synchronous Data Training

- All workers (e.g., CPUs, GPUs, TPUs)
- Must complete their assigned portion of the data processing for an iteration
- Iteration:
  - e.g. a batch of data during gradient descent

- This coordination ensures that all updates to the model parameters are based
  1. Complete data for that iteration
  2. Training process deterministic
  3. Easier to debug and optimize.



Key Characteristics Synchronous Data Training



Consistency: All updates to the model parameters are made after aggregating the gradients computed by all workers on their respective data subsets. This ensures consistent and stable updates.



Communication Overhead:

There is a significant communication overhead because all workers need to synchronize their updates at the end of each training iteration.

This can lead to idle times where some workers wait for others to finish their computations.



Fault Tolerance: The synchronous approach can be less fault-tolerant compared to asynchronous methods.

If one worker fails or is significantly slower, it can halt or slow down the entire training process.



Scalability: While synchronous training can scale to a large number of workers, its efficiency can be impacted by the slowest worker, which can limit scalability.

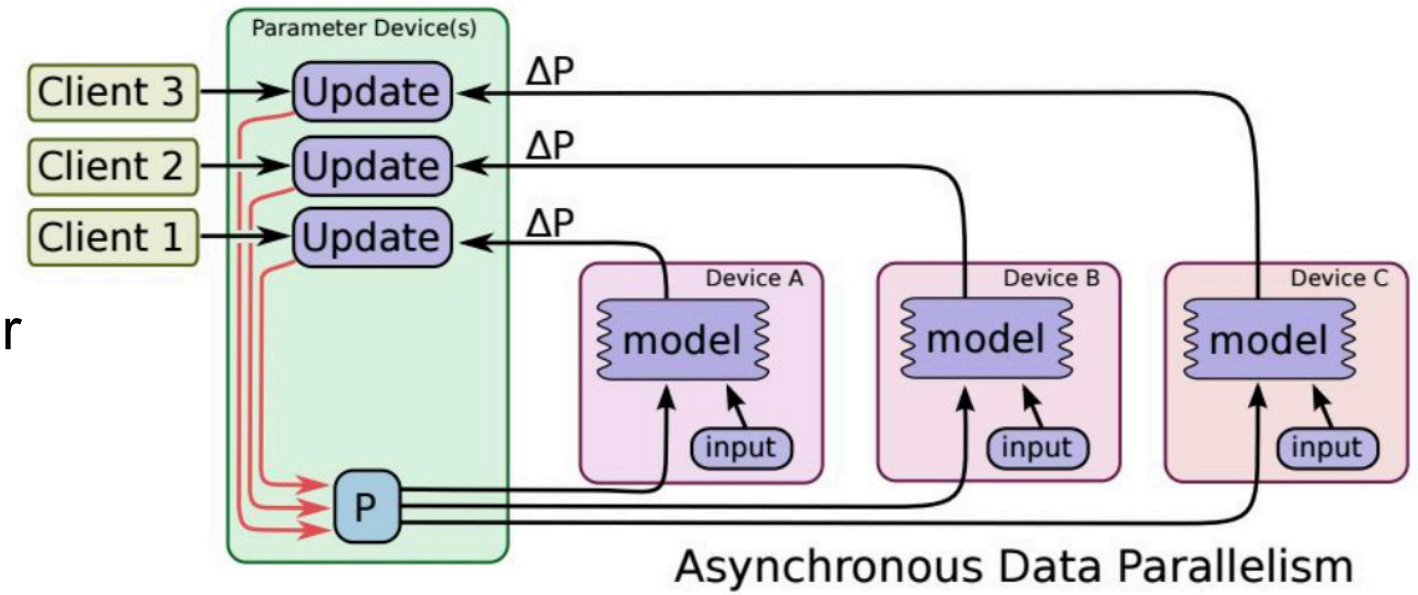


Reproducibility: Synchronous training tends to offer more reproducible results compared to asynchronous training

As the order and manner of updates are consistent across different runs, assuming the same initial conditions.

Asynchronous Data Training

- Multiple workers (e.g., CPUs, GPUs, TPUs) update the model's parameters independently without waiting for other workers to complete their updates
- Allows for more flexible and often faster training processes, particularly in heterogeneous computing environments where different workers may have varying computational speeds



Asynchronous Data Training



Independence: Workers update the model parameters as soon as they finish processing their assigned data, without waiting for other workers. This can lead to more efficient use of computational resources.



Staleness: The model parameters that a worker reads at the beginning of its computation might become outdated ("stale") by the time it applies its updates, due to the independent updates from other workers.



Efficiency: Asynchronous training can lead to faster training times since there's less idle time waiting for all workers to synchronize. It can effectively utilize resources in environments with variable computational performance across workers.



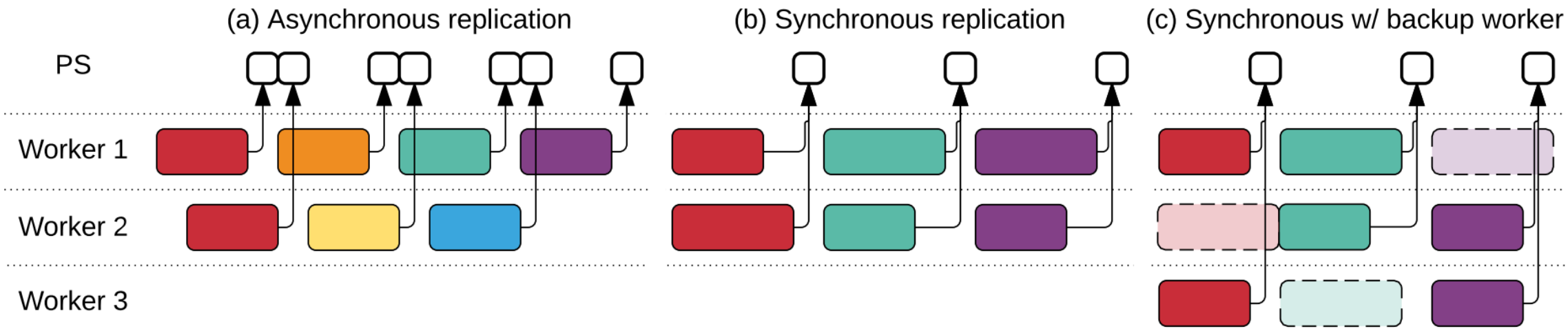
Scalability: It scales well to large numbers of workers, as adding more workers doesn't introduce additional synchronization overhead.



Fault Tolerance: The training process is more fault-tolerant since the failure of a single worker doesn't halt the entire training process; other workers can continue updating the model.

TensorFlow: Synchronous Training with Backup Workers

1. Need gradients from m workers
2. Launch n (n > m) workers
3. Accept gradients from the first m of n workers



Why this works?

ML Property:
SGD samples training data randomly at each step!

TensorFlow: Synchronous Training with Backup Workers



Additional Workers: Beyond the primary set of workers needed for a training iteration, extra or "backup" workers are also assigned tasks.



Early Completion: The training step proceeds as soon as the required number of workers (the primary set) completes their tasks, without waiting for all workers to finish. The first workers to finish, which could include backup workers, contribute to the model update for that iteration.

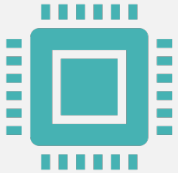


Dynamic Adjustment: Backup workers can dynamically join the pool of active workers for an iteration if they are ready to proceed before some of the primary workers, effectively replacing stragglers

TensorFlow: Synchronous Training with Backup Workers



Reduced Idle Time: By not waiting for the slowest workers, the overall training process can proceed faster, reducing the idle time of faster workers.



Scalability: This method allows the training process to scale more effectively by adding more computational resources without being limited by the slowest component.



Fault Tolerance: Incorporating backup workers enhances fault tolerance, as the failure of a primary worker doesn’t significantly delay the training process; a backup worker can take its place.