Lecture 24: Parallel Deep Learning (Data Parallelism)

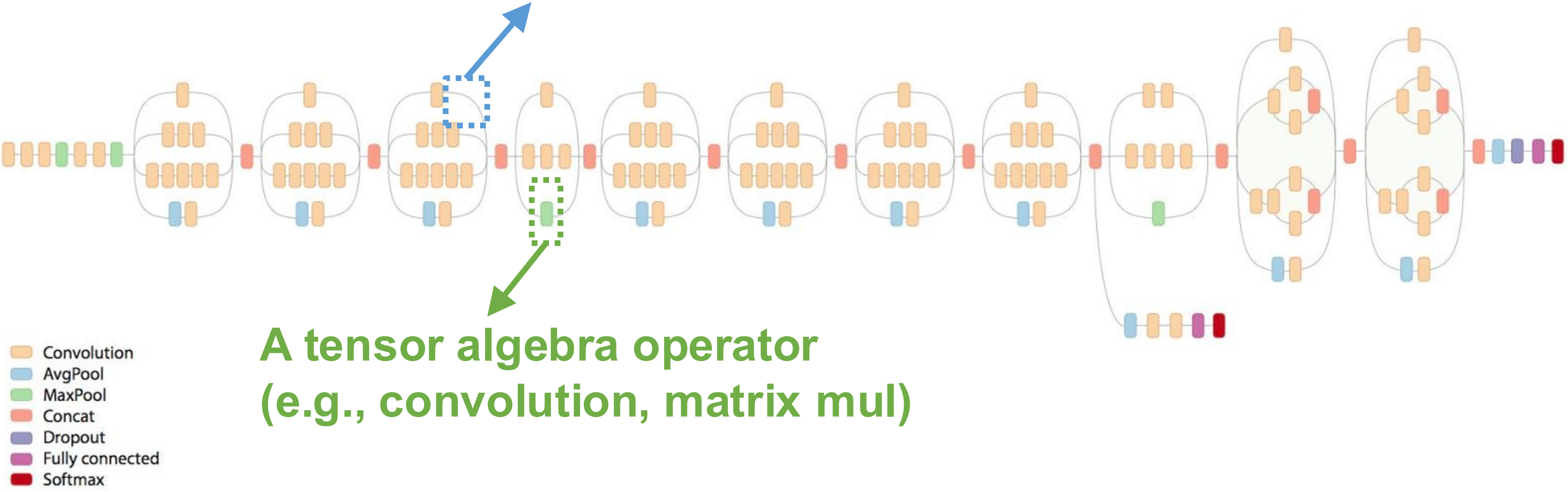
Parallel Computer Architecture and Programming

CMU 15-418/15-618, Fall 2025

10/28/2025

Recap: Deep Neural Network

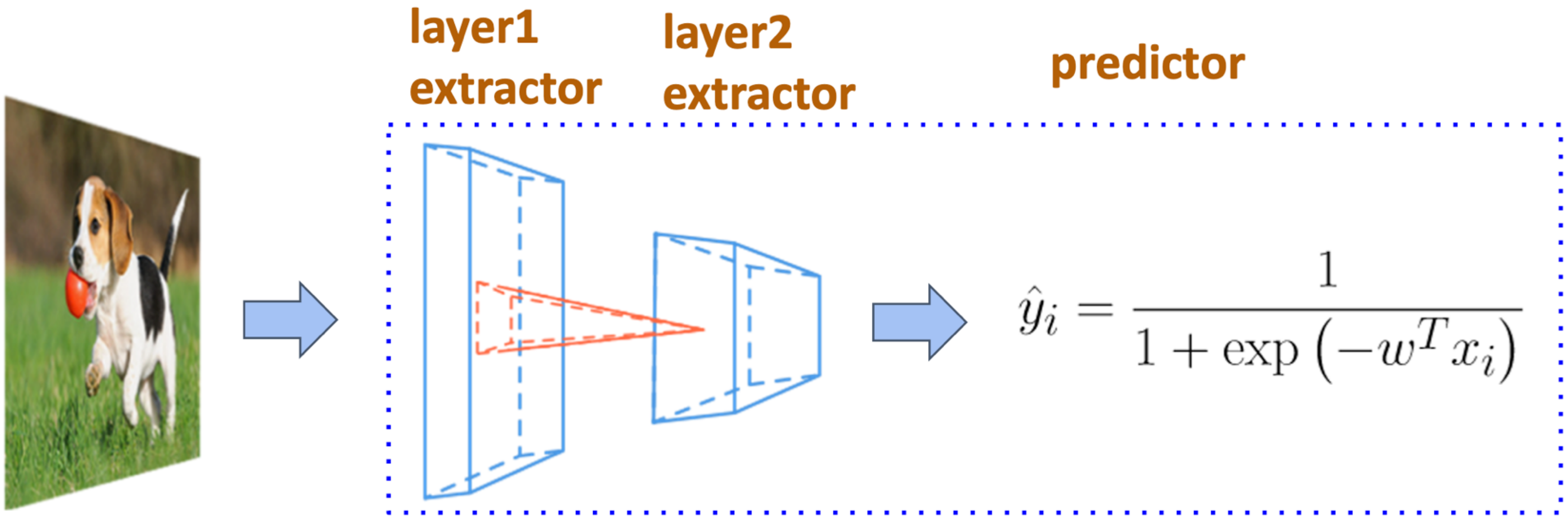
- Collection of simple trainable mathematical units that work together to solve complicated tasks



A tensor (i.e., *n*-dimensional array)

A tensor algebra operator (e.g., convolution, matrix mul)

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

$$w \leftarrow w - \boxed{\eta \nabla_w L(w)}$$

1. Compute how the loss changes with each parameter

2. Move w to the opposite direction to reduce the loss

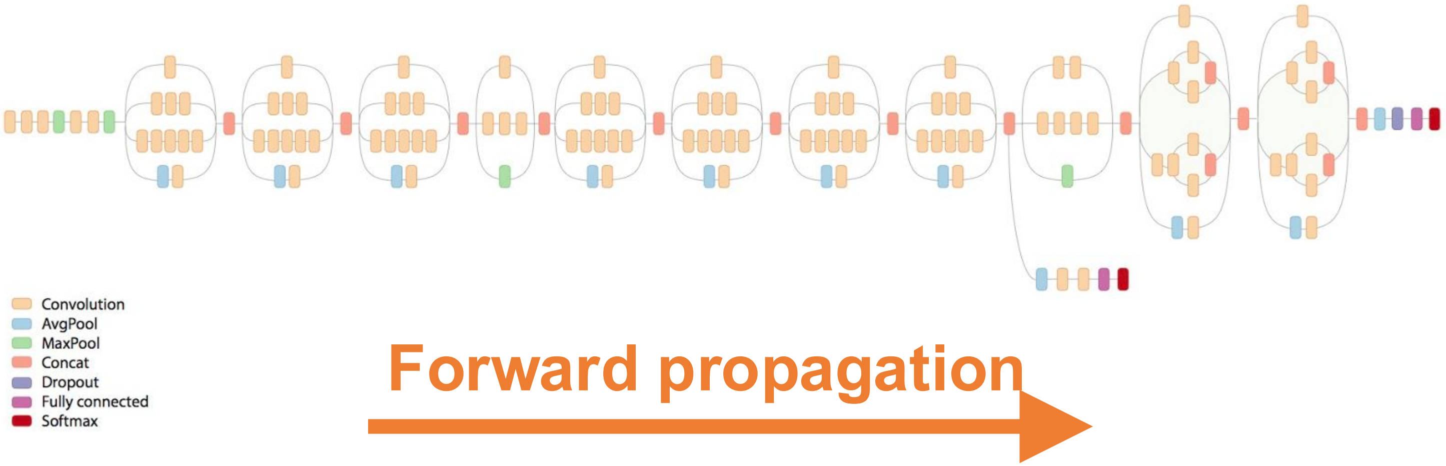
3. Repeat over data

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

Model inputs



Model prediction

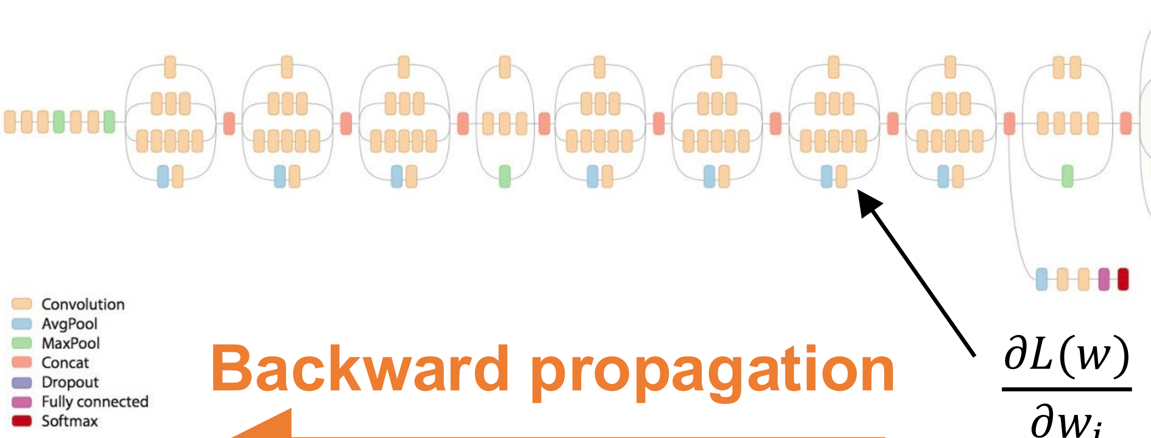
Forward propagation

DNN Training Process

Train ML models through many iterations of 3 stages

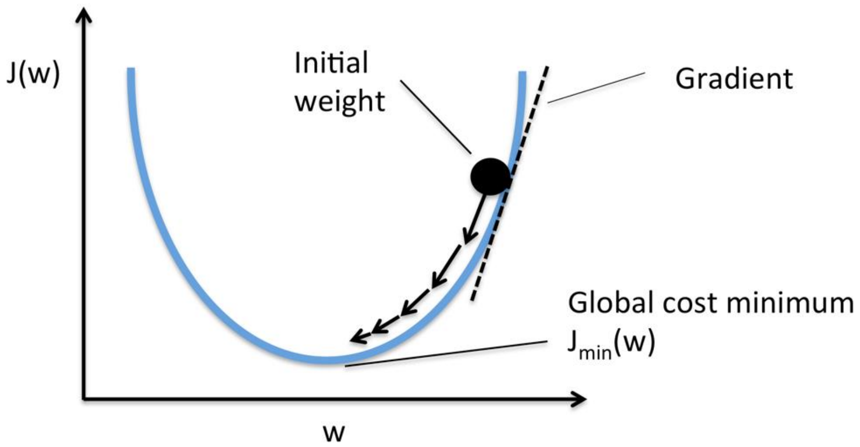
1. **Forward propagation**: apply model to a batch of input samples and run calculations through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce a gradient for each trainable weight
3. **Weight update**: use the loss value to update model weights

Model inputs



Backward propagation

$$\frac{\partial L(w)}{\partial w_i}$$



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

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

Gradients of individual samples

How can we parallelize DNN training?

The gradient of the loss function wrt the parameters

Batch Size

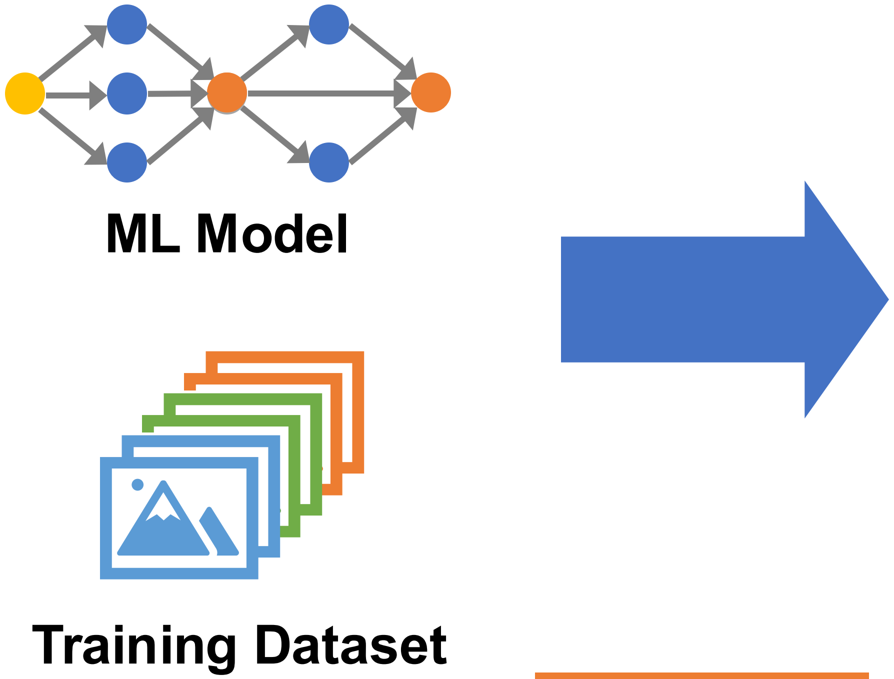
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\sum_{j=1}^{n} \nabla L_j(w_i)$$

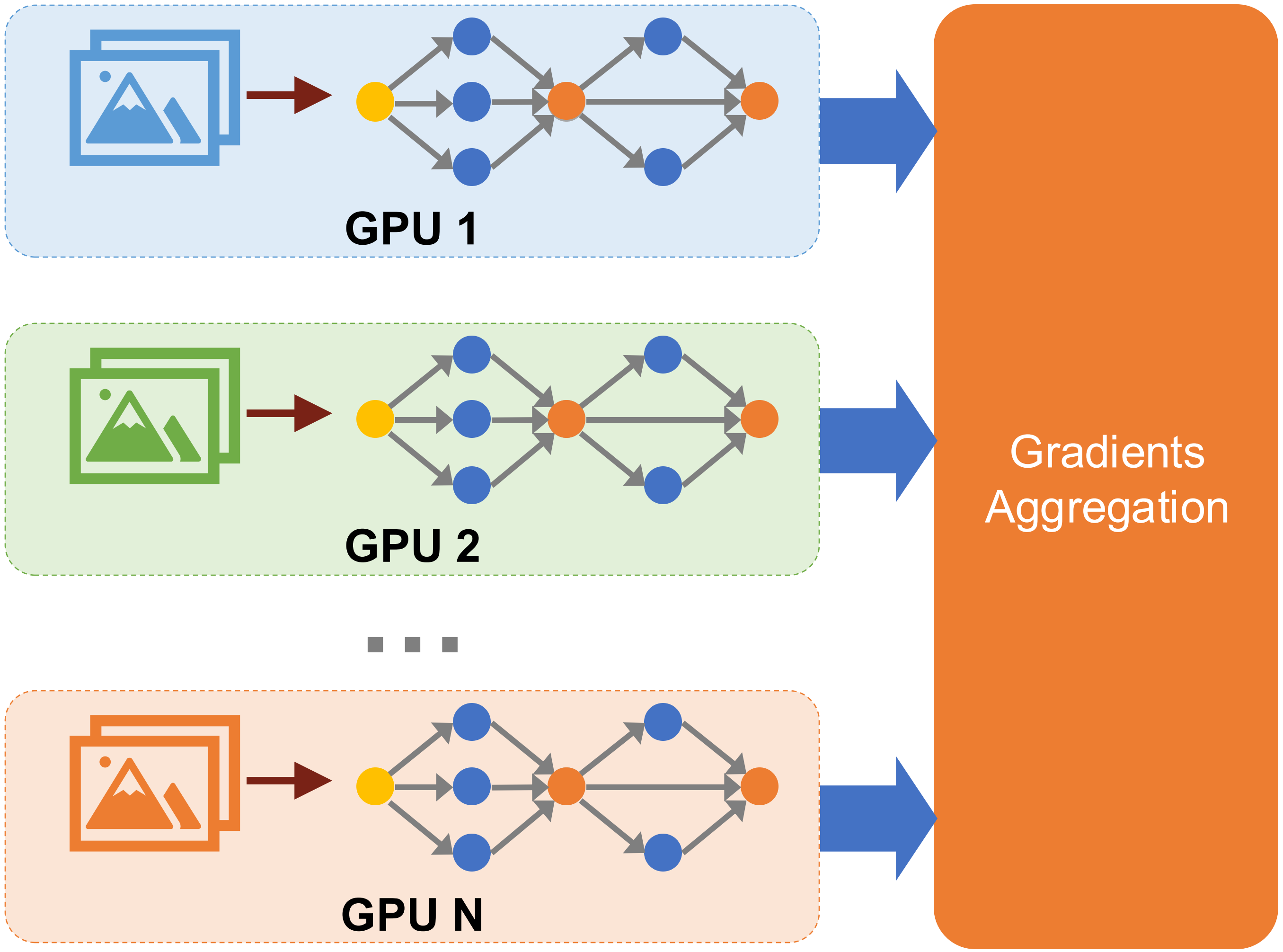
Parameter vector → weights at iter i

Loss function for w_i

Learning rate → step size

Data Parallelism





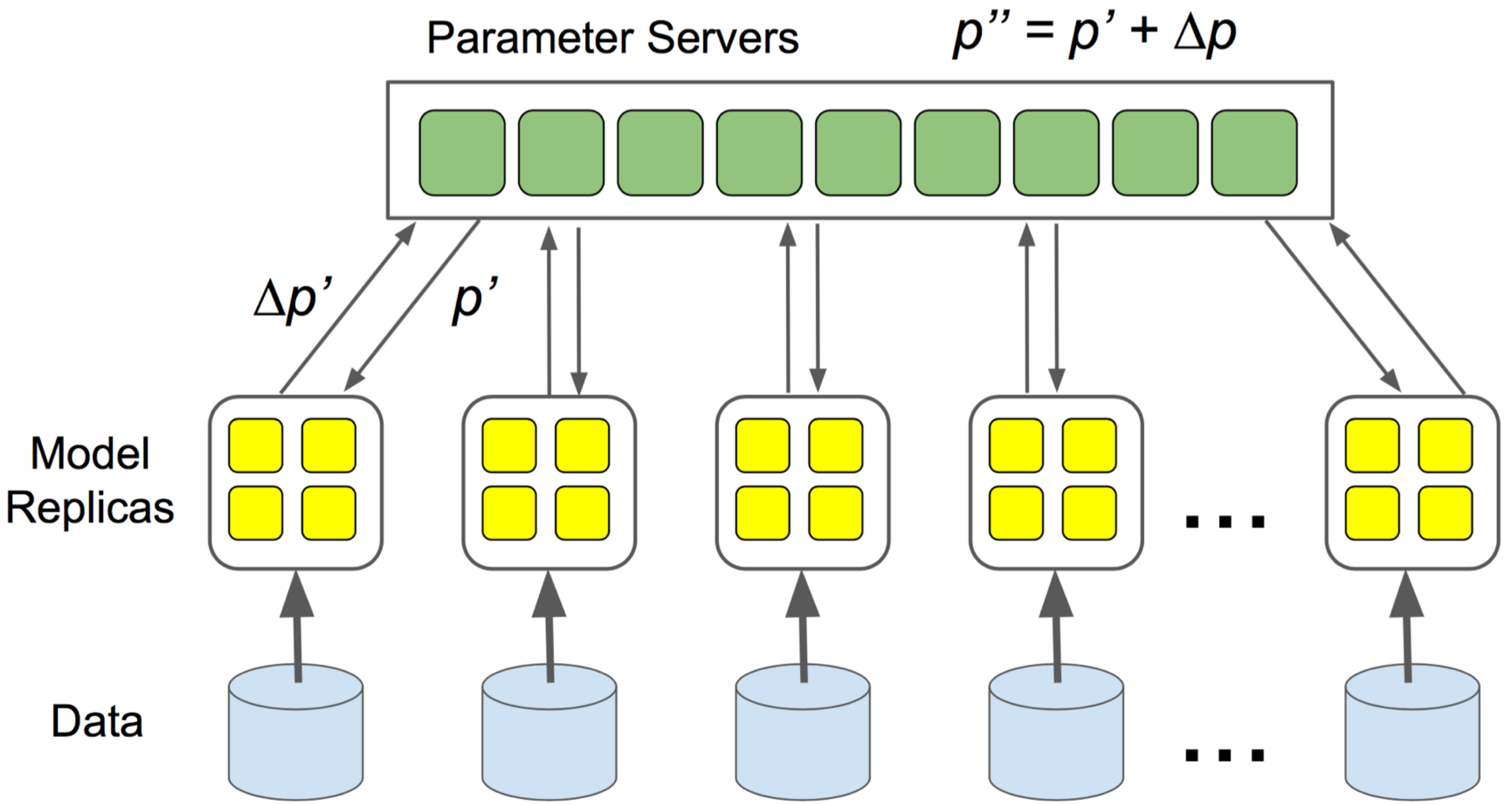
$$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n}\boxed{\sum_{j=1}^{n} \nabla L_j(w_i)}$$

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

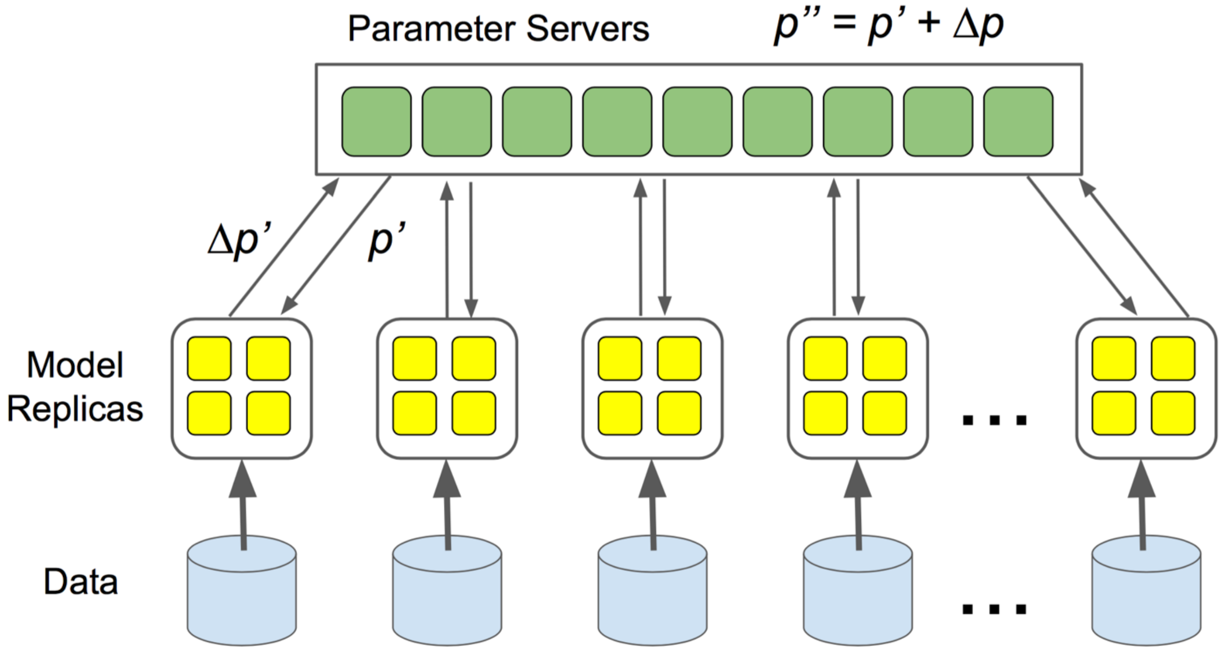


Workers push gradients to parameter servers and pull updated parameters back

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?

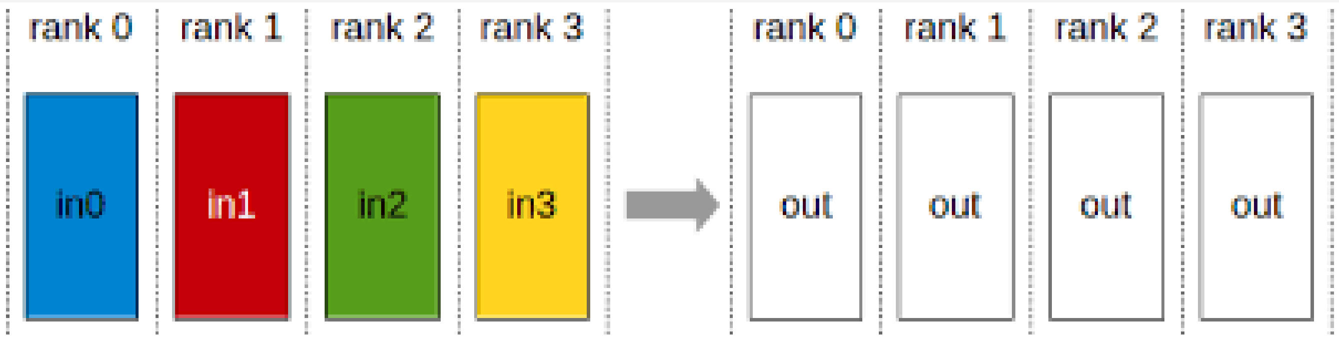


Inefficiency of Parameter Server

- **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



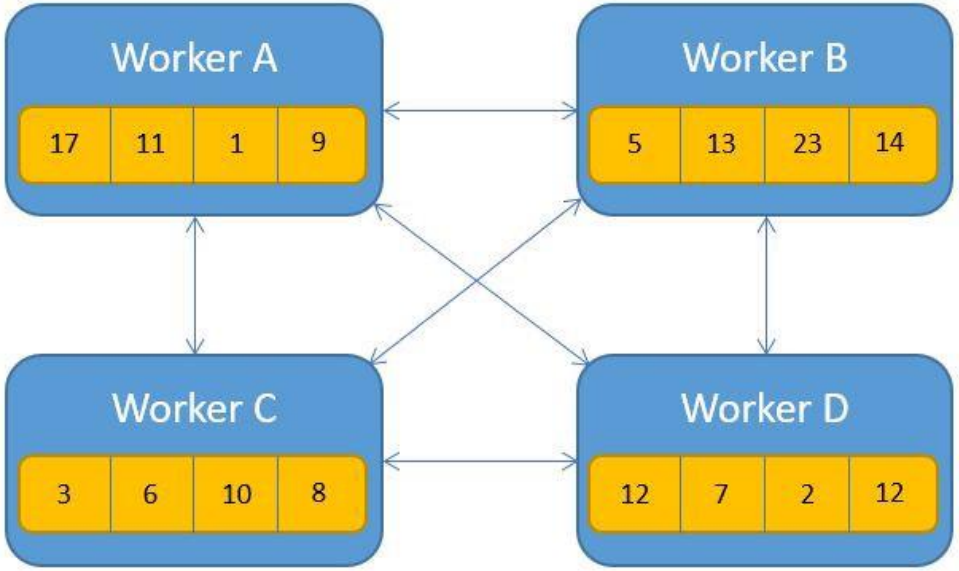
out[i] = sum(inX[i])

Different Ways to Perform AllReduce

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

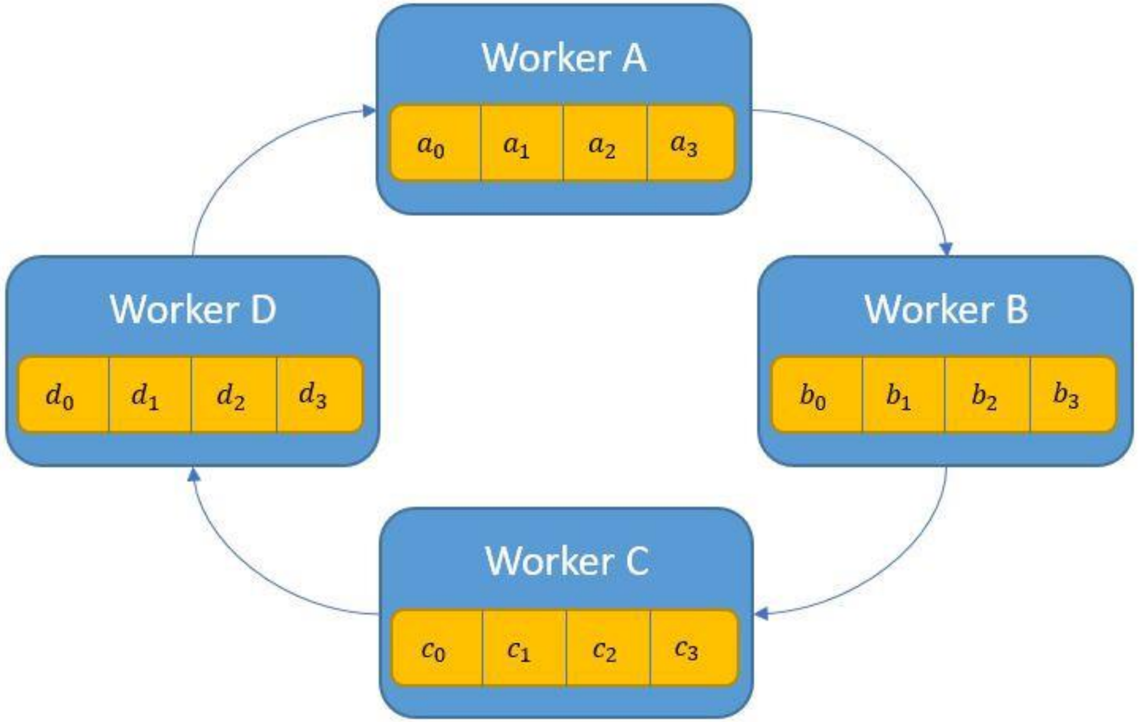
Naïve AllReduce

- 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



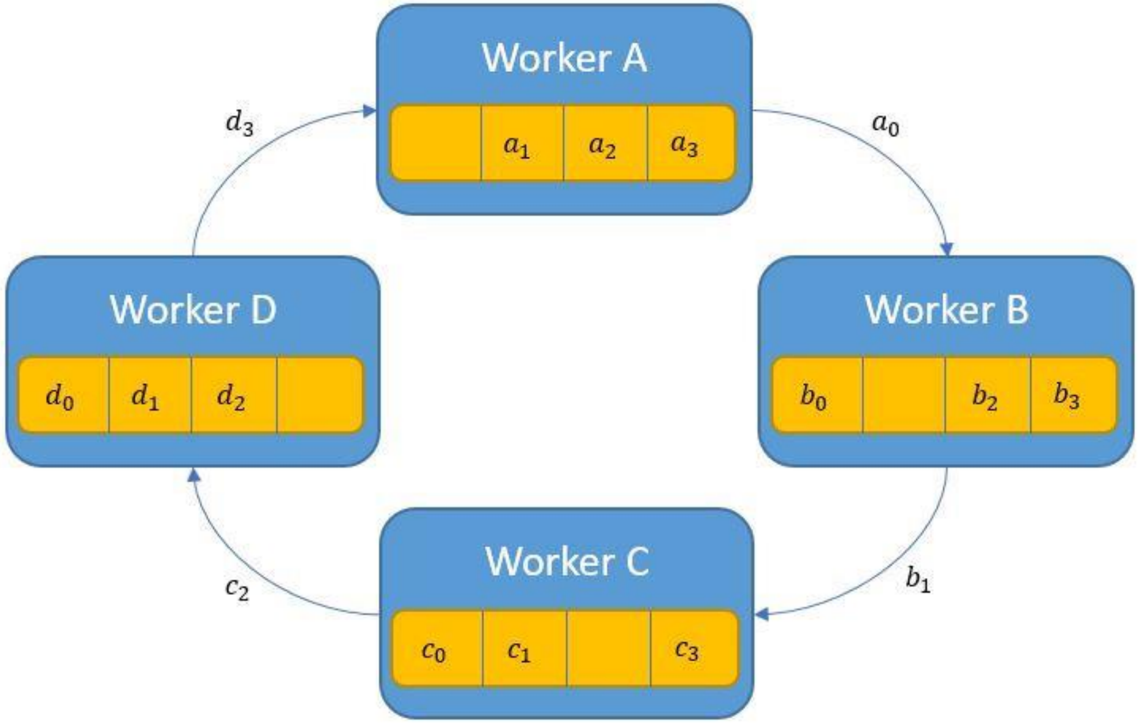
Ring AllReduce

- 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



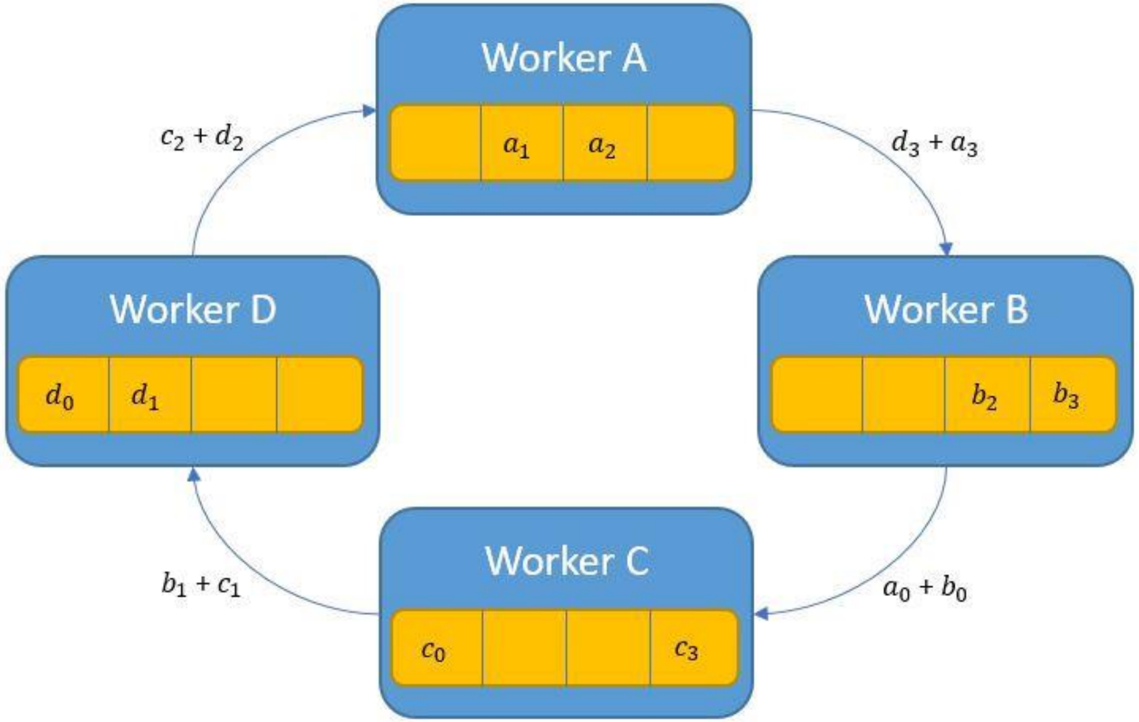
Ring AllReduce

- 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 times



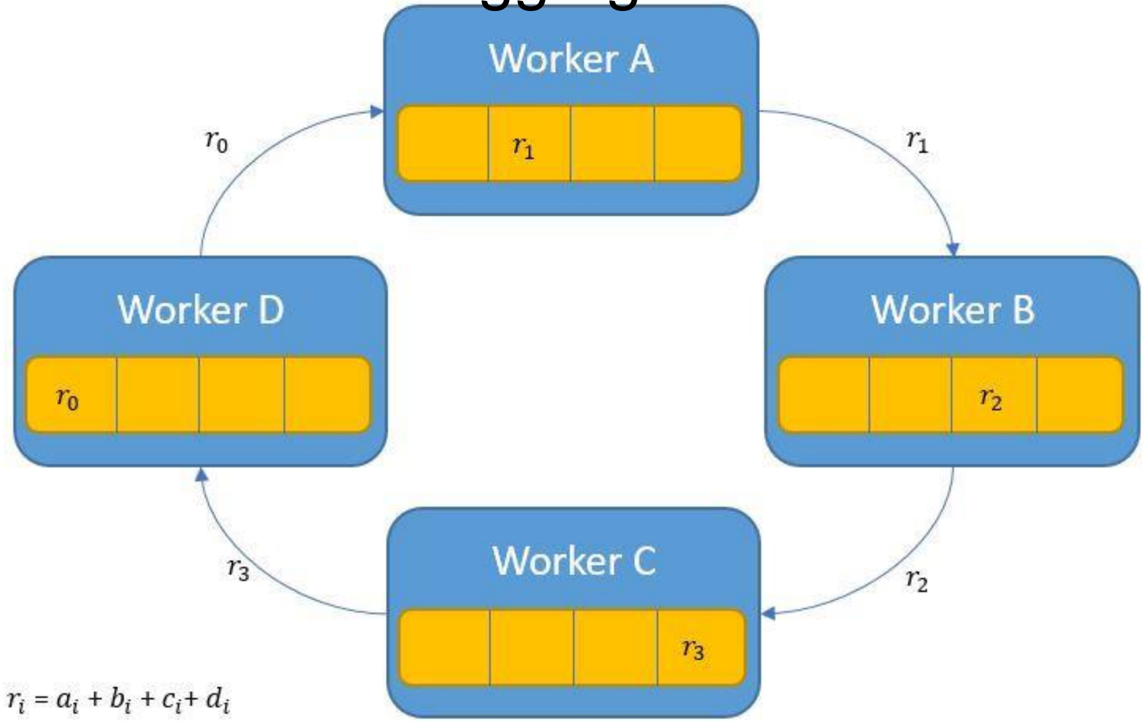
Ring AllReduce

- 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 times



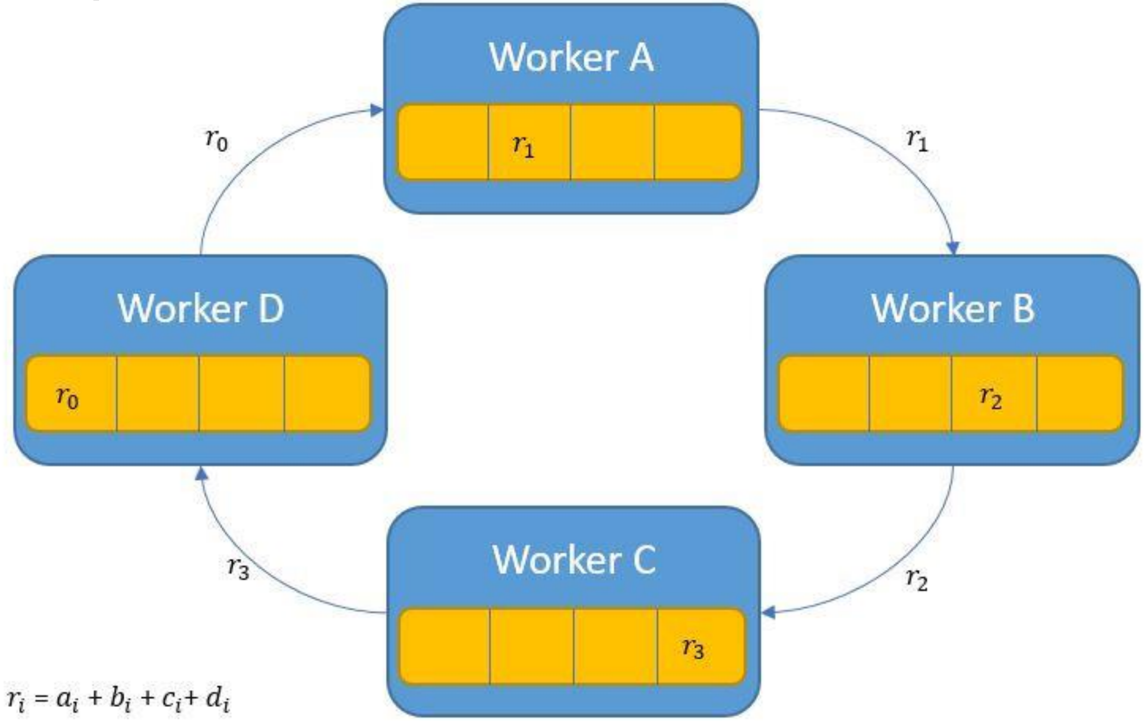
Ring AllReduce

- 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 times
- After step 1, each worker has the aggregated version of M/N parameters



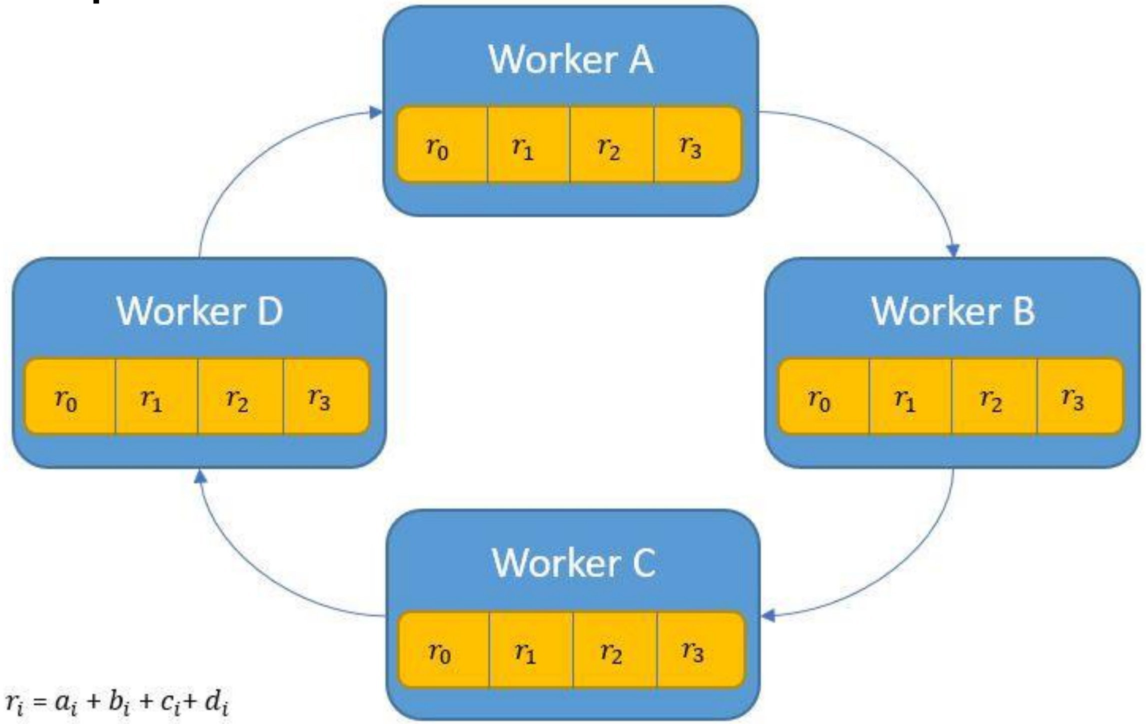
Ring AllReduce

- 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 times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; 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 times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times

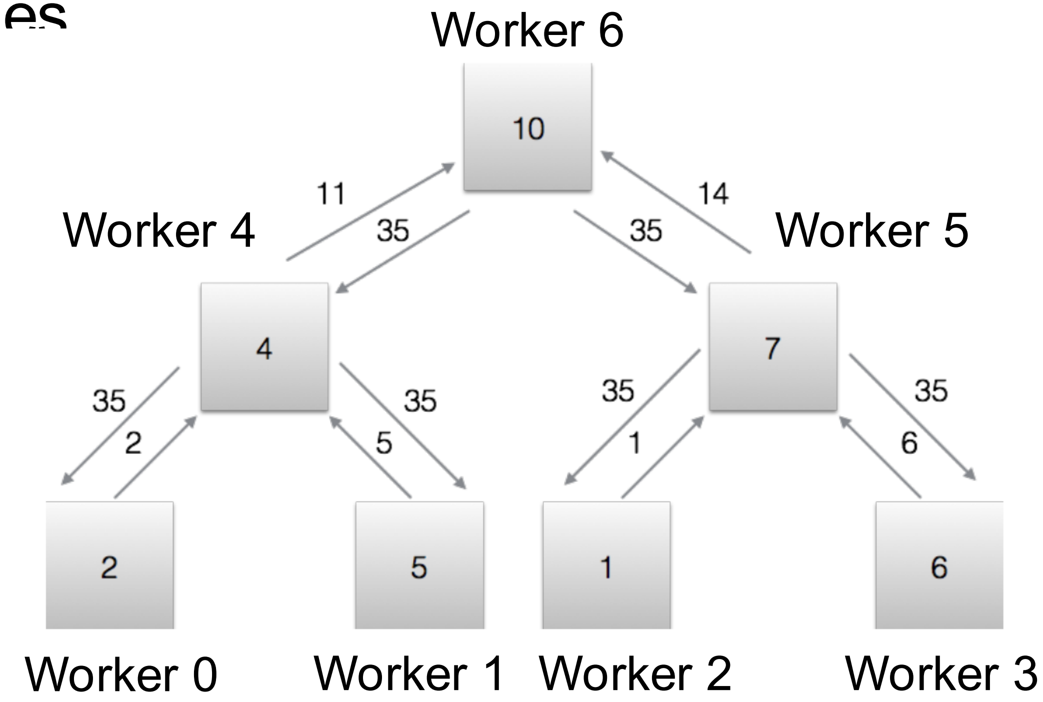


Ring AllReduce

- 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 times
- Step 2 (Broadcast): each worker send one slice of aggregated parameters to the next worker; repeat N times
- Overall communication: 2 * M * N parameters
  - Aggregation: M * N parameters
  - Broadcast: M * N parameters

Tree AllReduce

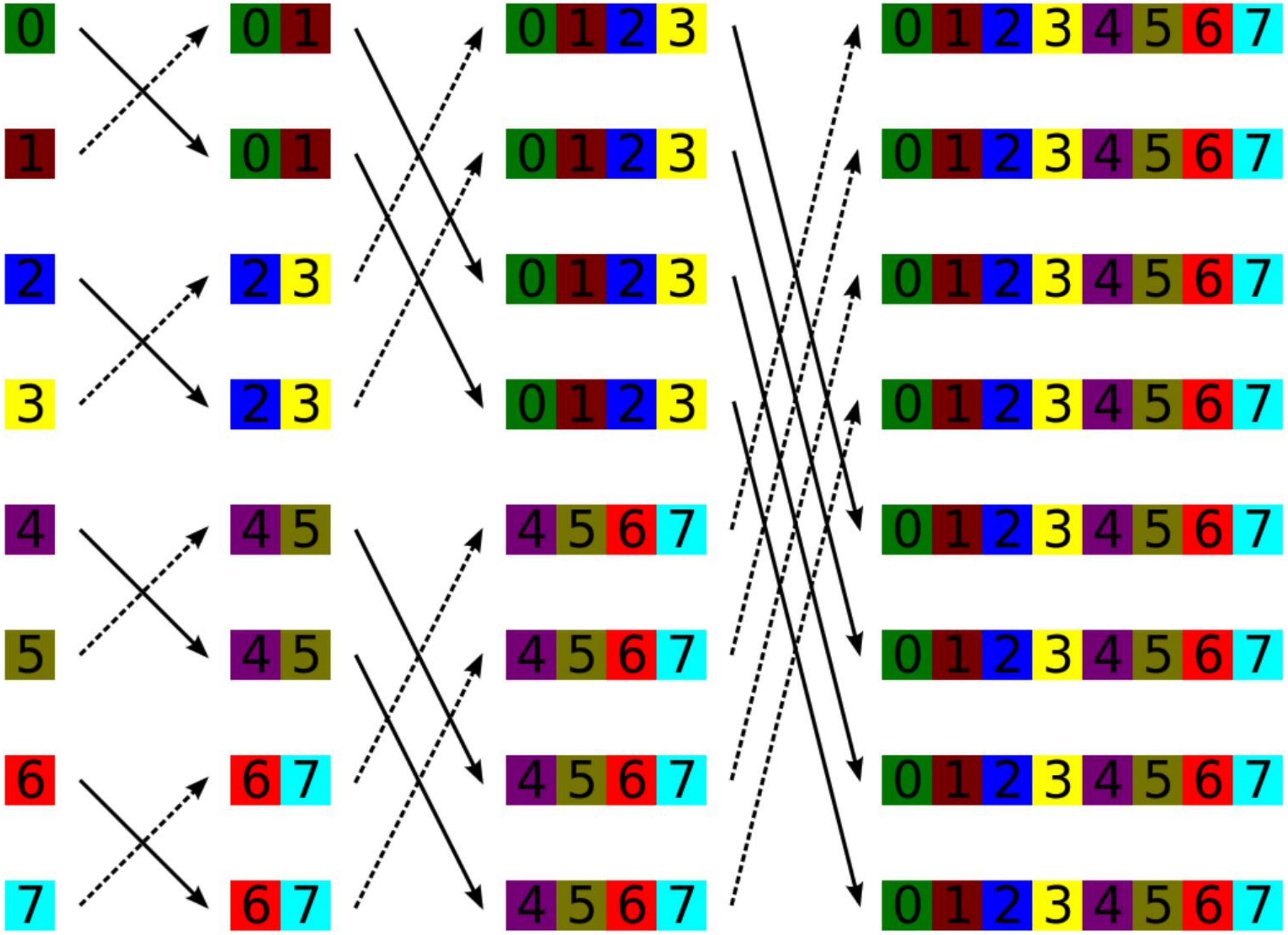
- Construct a tree of N workers;
- Step 1 (Aggregation): each worker sends M parameters to its parent; repeat log(N) times
- Step 2 (Broadcast): each worker sends M parameters to its children; repeat log(N) times



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

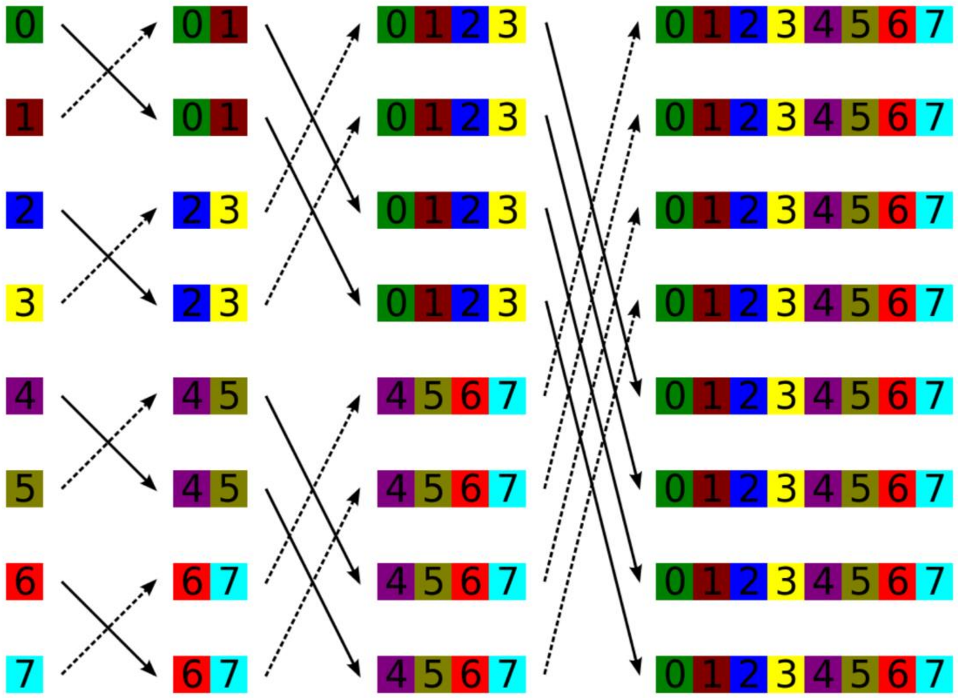
Butterfly Network



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

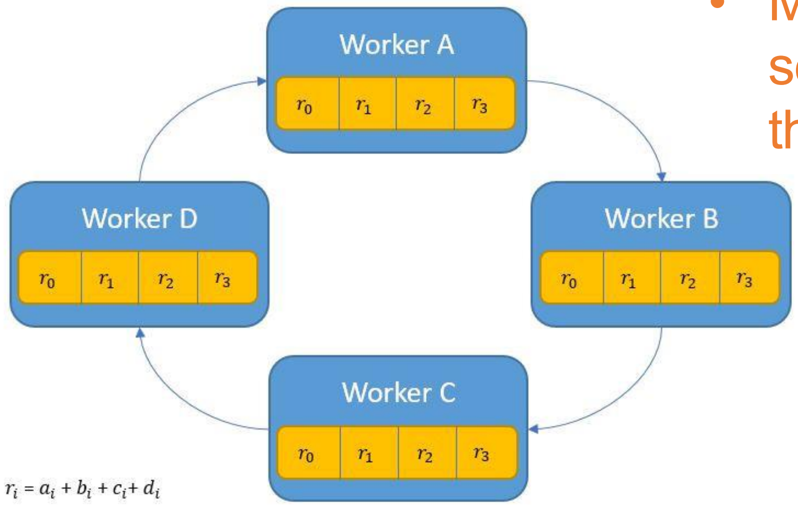
	Parameter Server	Naïve AllReduce	Ring AllReduce	Tree AllReduce	Butterfly AllReduce
Overall communication	$2 \times N \times M$	$N^2 \times M$	$2 \times N \times M$	$2 \times N \times M$	$N \times M \times \log N$

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

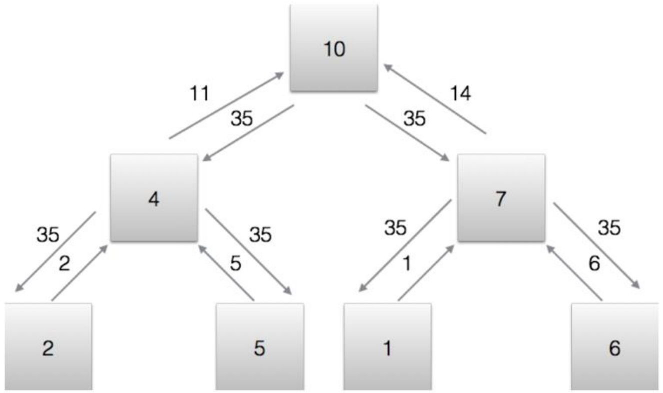
Ring AllReduce:

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



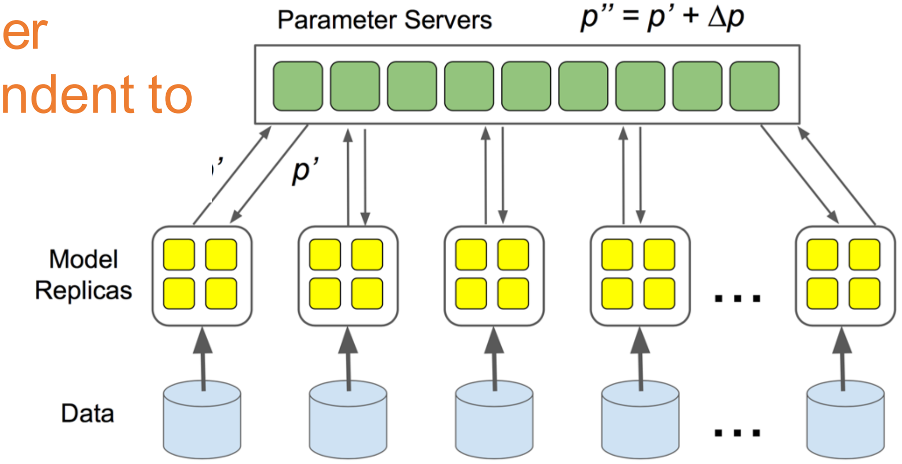
Each worker sends **M/N** parameters per iteration; repeat for **2*N** iterations

Latency: M/N * (2*N) / bandwidth



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

Latency: M * 2 * log(N) / bandwidth



All workers send **M** parameters to parameter servers and receive **M** parameters from servers

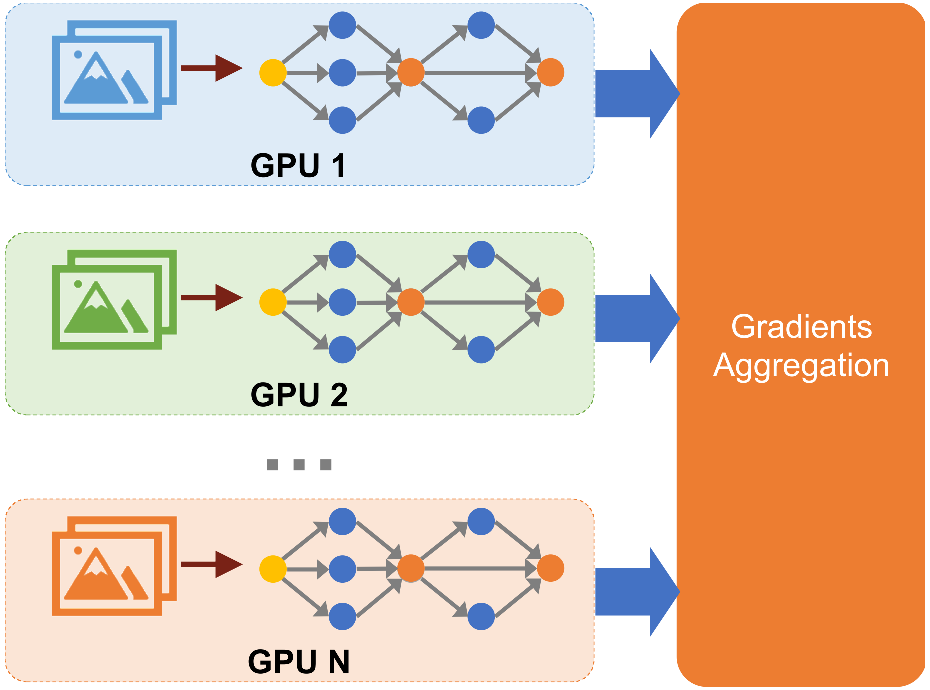
Latency: M * N / bandwidth

Recap: Data Parallelism (Quiz)

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

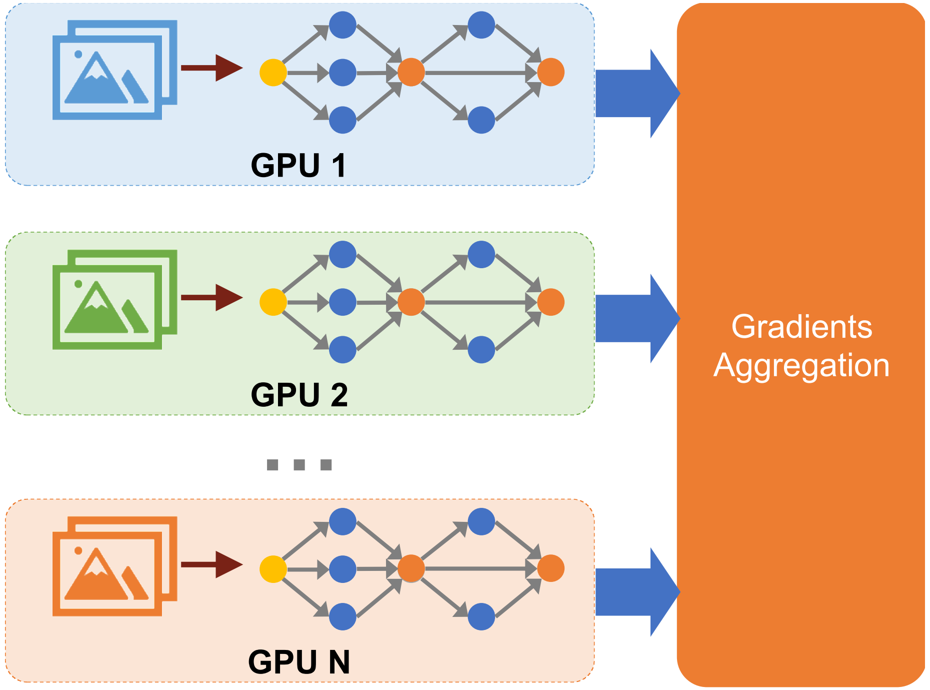
Gradients aggregation methods:

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



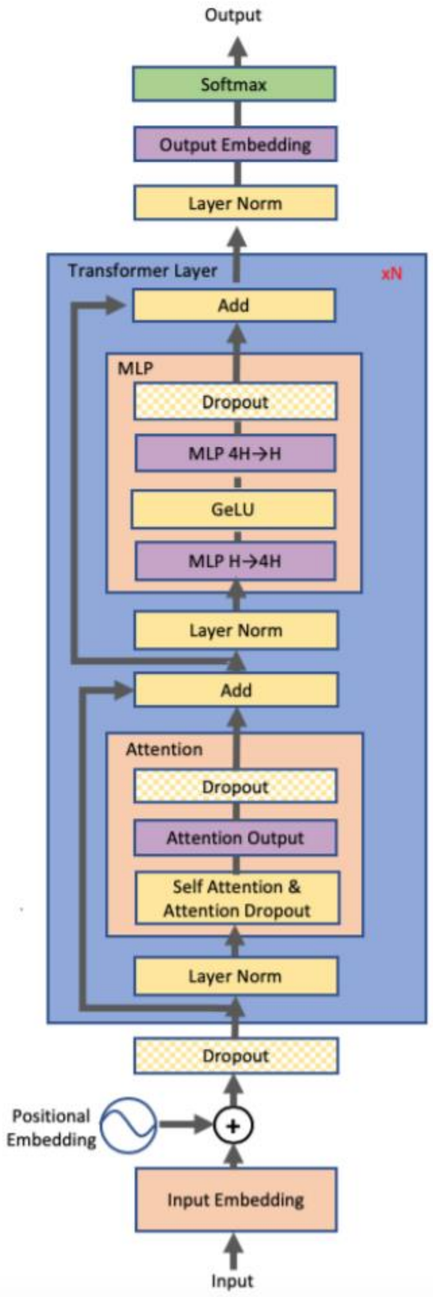
An Issue with Data Parallelism

- 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**

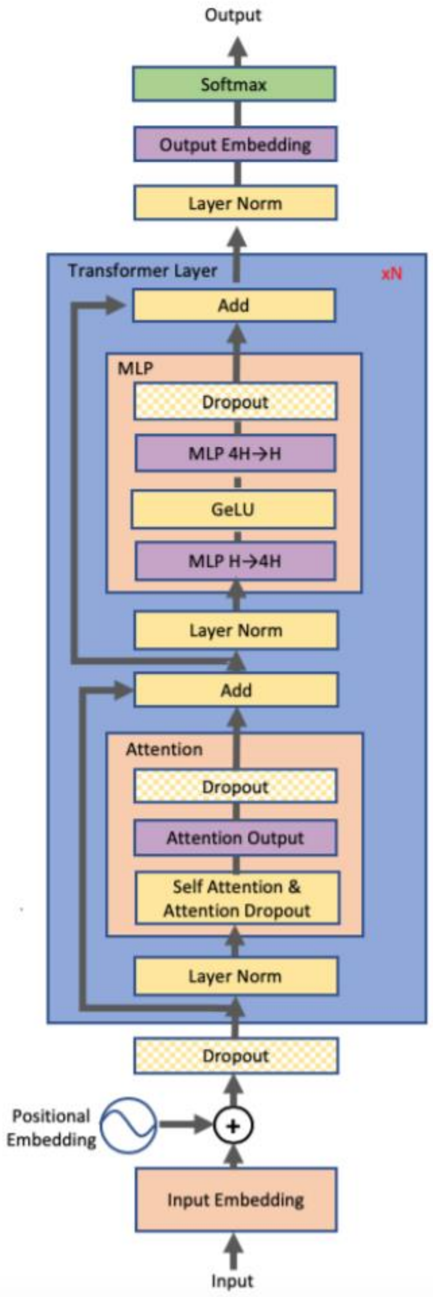


Large Model Training Challenges

	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

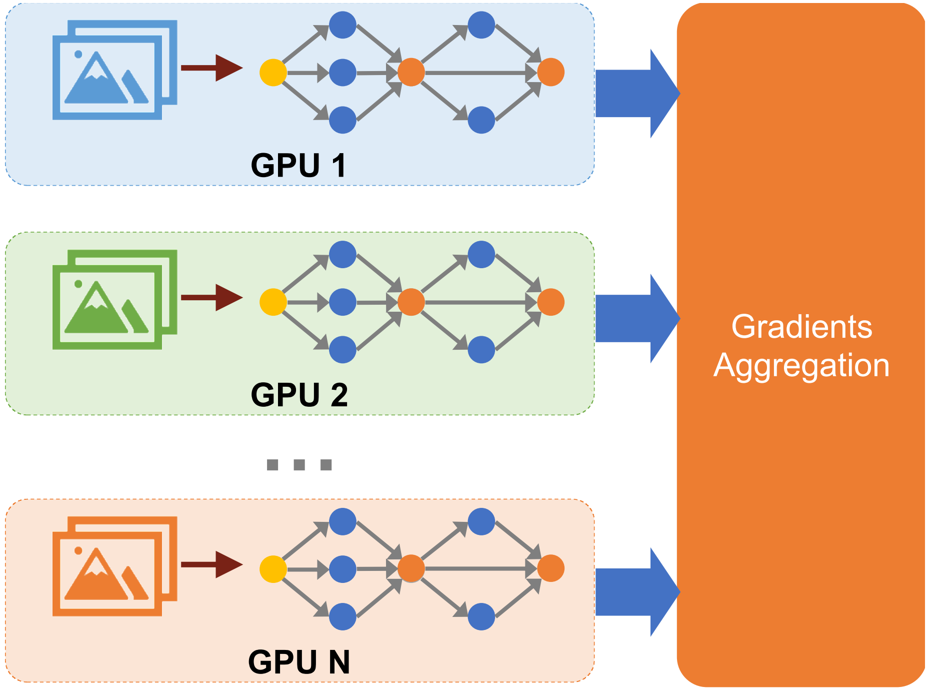
$\underbrace{\hspace{6em}}_{\text{Out of Memory}}$



ZeRO: Zero Redundancy Optimizer

deepspeed

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



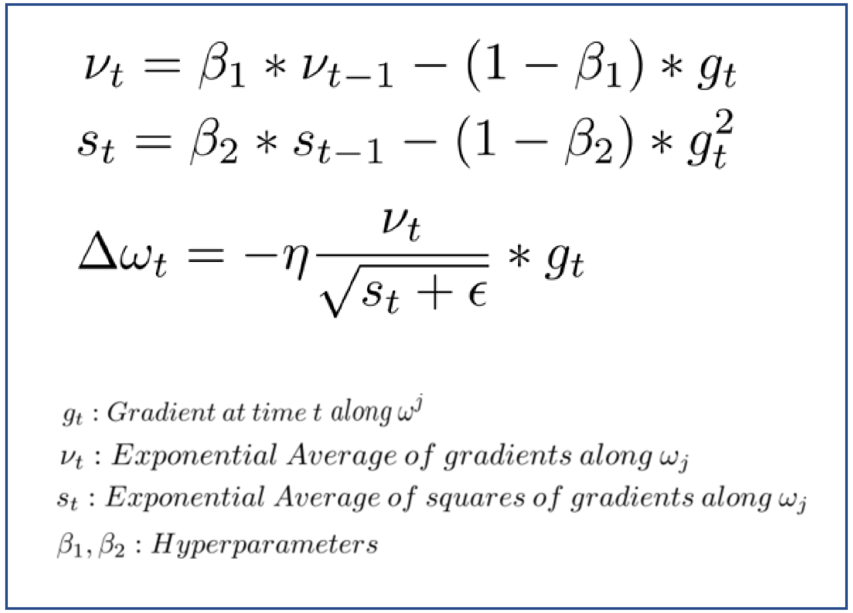
Revisit: Stocastic Gradient Descent

```
For t = 1 to T
    $\Delta w = \eta \ x \ \frac{1}{b}\sum_{i=1}^{b} \overset{\text{Backward pass}}{\boxed{\nabla}} \left( \overset{\text{Forward pass}}{\boxed{loss\left(f_w(x_i, y_i)\right)}} \right)$   // compute derivative and update
    $w$ -= $\Delta w$   // apply update
End
```

Adapted from Minjia Zhang, DeepSpeed Presentation

Adaptive Learning Rates (Adam)

```
For t = 1 to T
    g = $\frac{1}{b}\sum_{i=1}^{b} \nabla \left(loss\left(f_w(x_i, y_i)\right)\right)$
    $\boxed{\Delta w = adam(g)}$
    w -= $\Delta$w   // 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

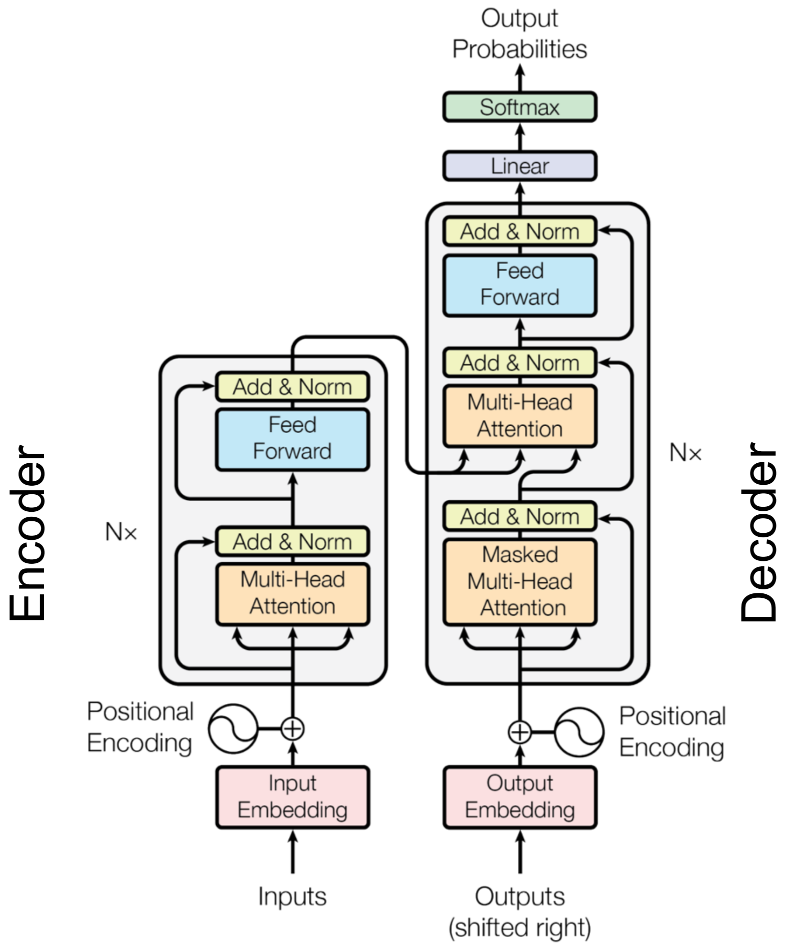
ν_t : *Exponential Average of gradients along* ω_j

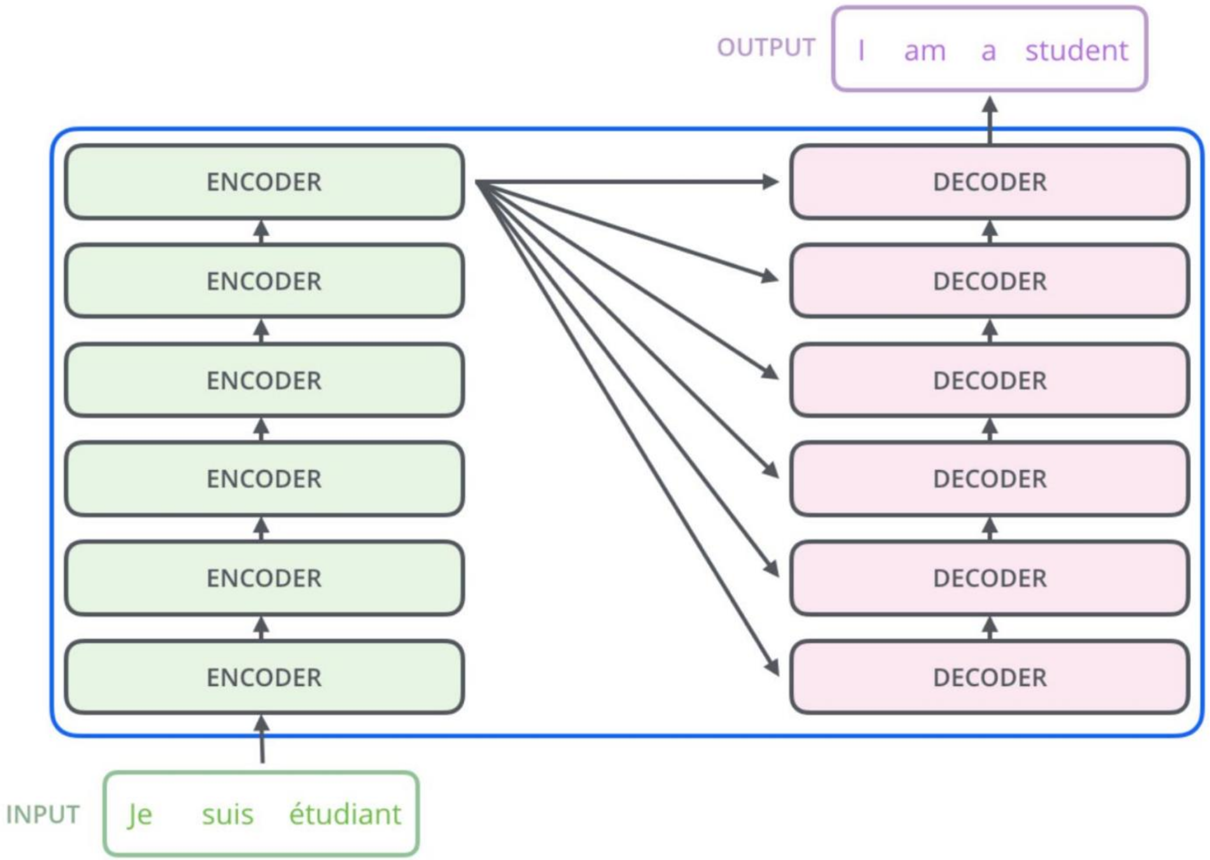
s_t : *Exponential Average of squares of gradients along* ω_j

β_1, β_2 : *Hyperparameters*

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

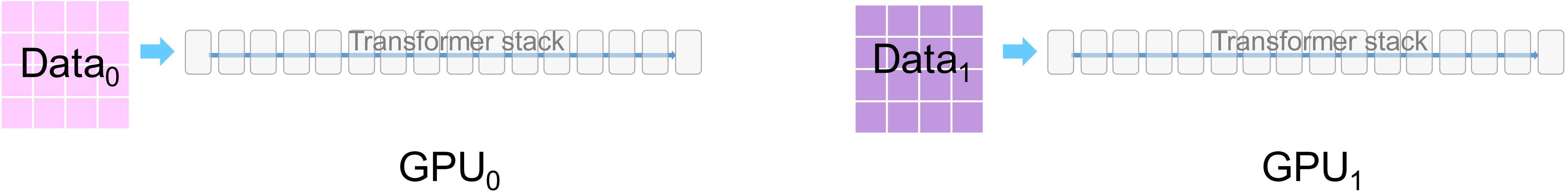
Recall: Transformer for Language Models





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

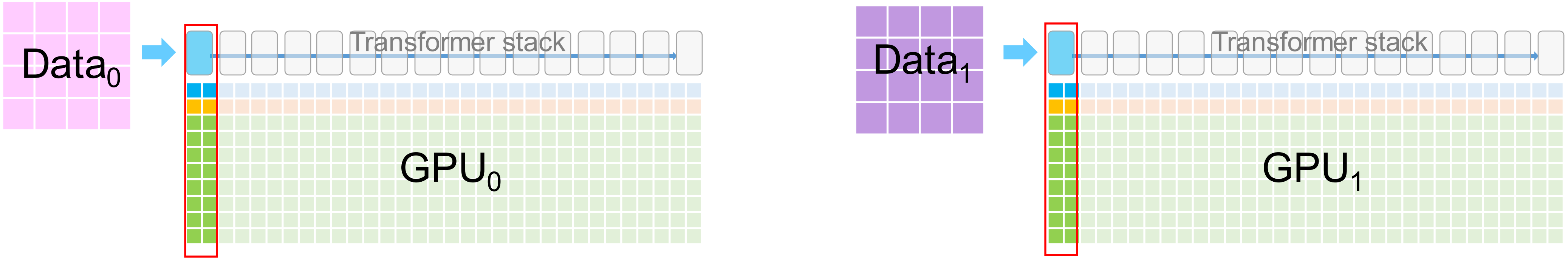
Understanding Memory Consumption



A 16-layer transformer model □ = 1 layer

Adapted from Minjia Zhang, DeepSpeed Presentation

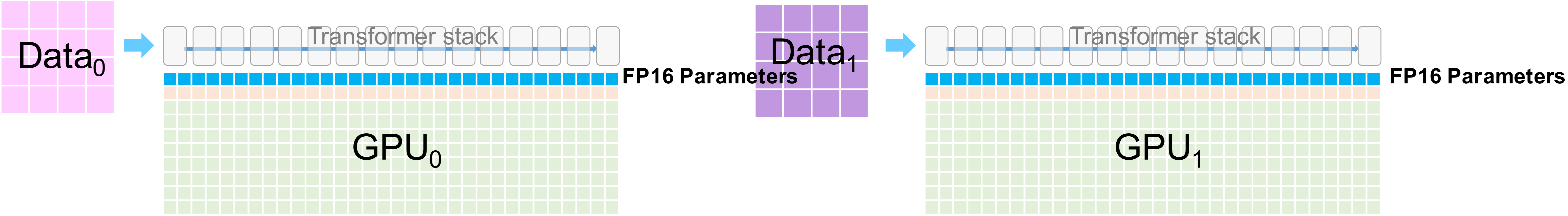
Understanding Memory Consumption



Each cell ▮ represents GPU memory used by its corresponding transformer layer ▢

Adapted from Minjia Zhang, DeepSpeed Presentation

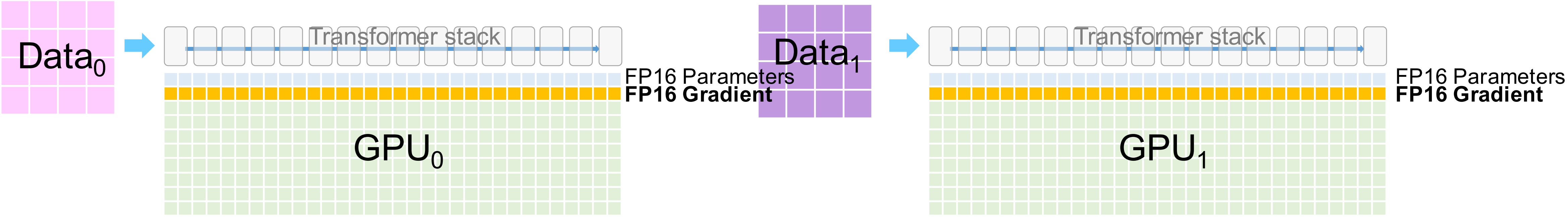
Understanding Memory Consumption



- FP16 parameter

Adapted from Minjia Zhang, DeepSpeed Presentation

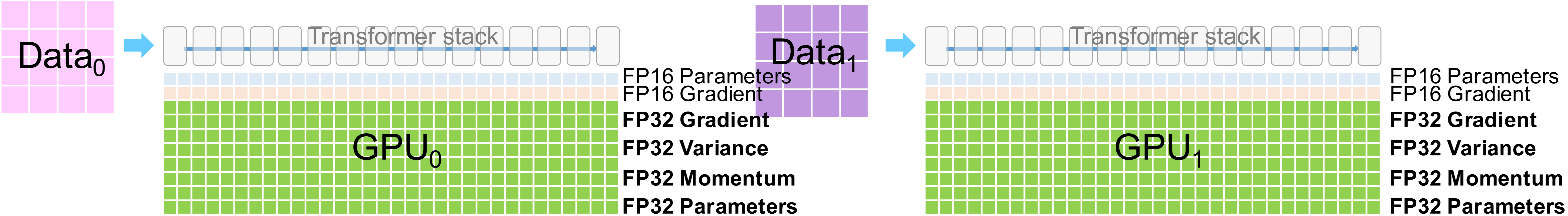
Understanding Memory Consumption



- FP16 parameter
- FP16 Gradients

Adapted from Minjia Zhang, DeepSpeed Presentation

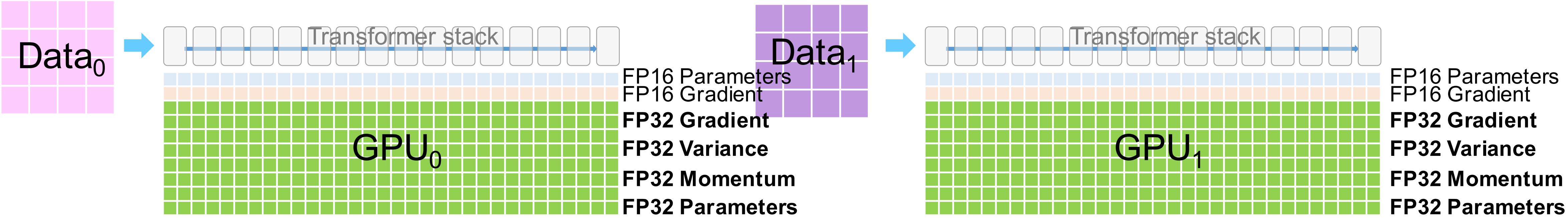
Understanding Memory Consumption



- 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

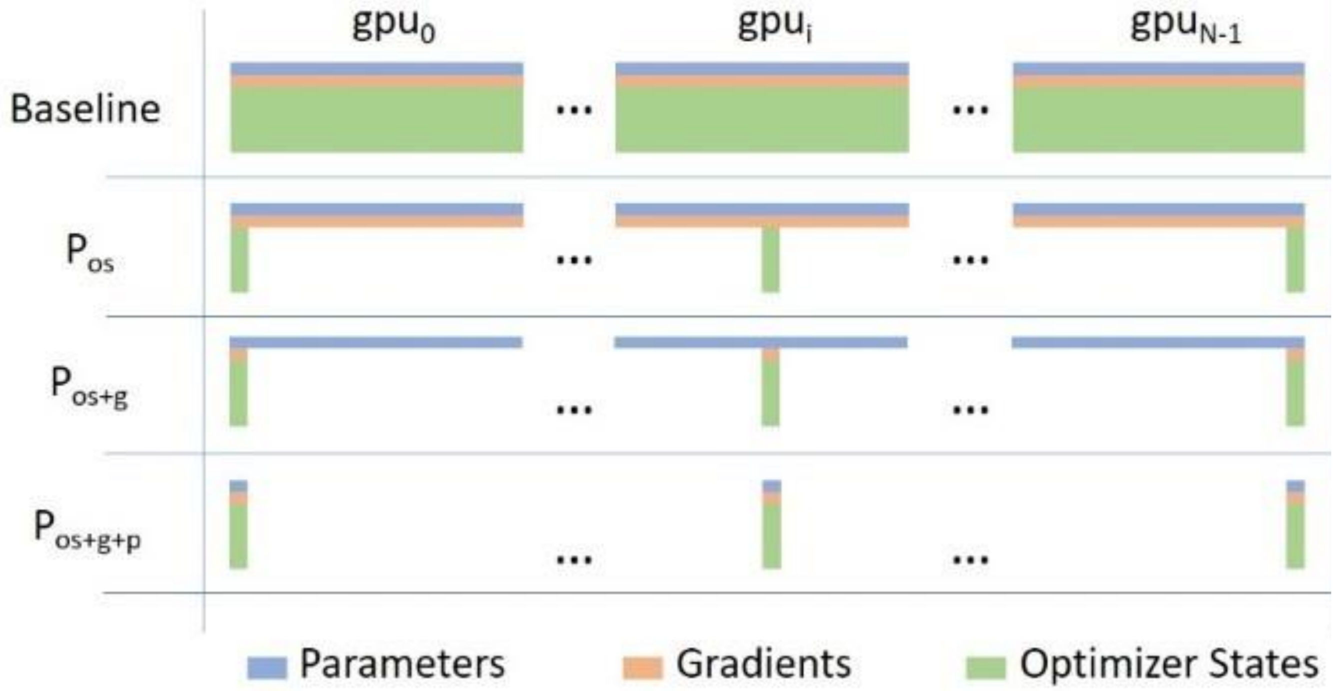
Memory consumption doesn’t include:

- Input batch + activations

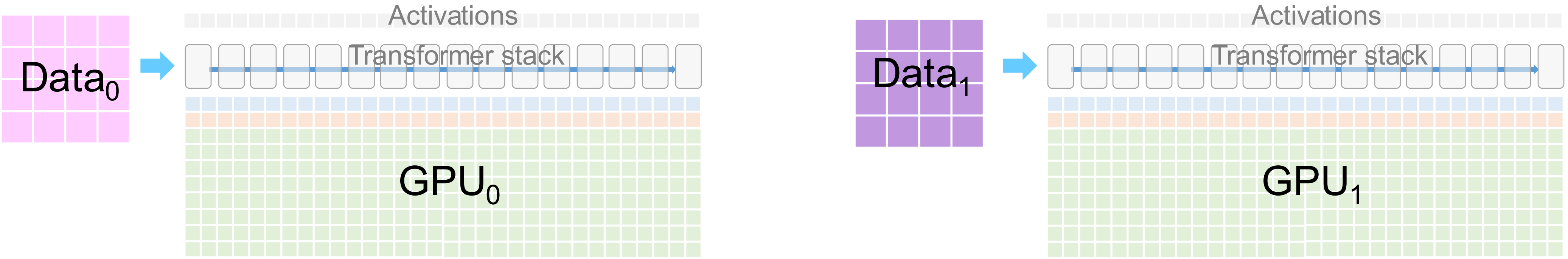
Adapted from Minjia Zhang, DeepSpeed Presentation

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



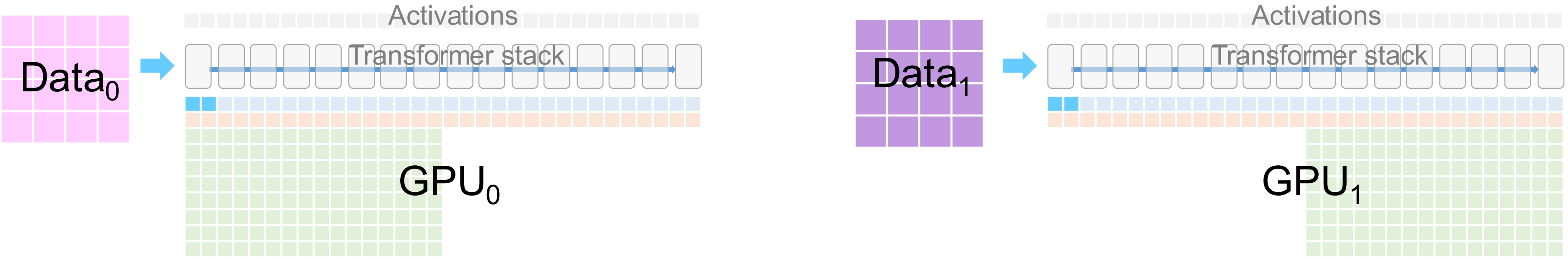
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1

Adapted from Minjia Zhang, DeepSpeed Presentation

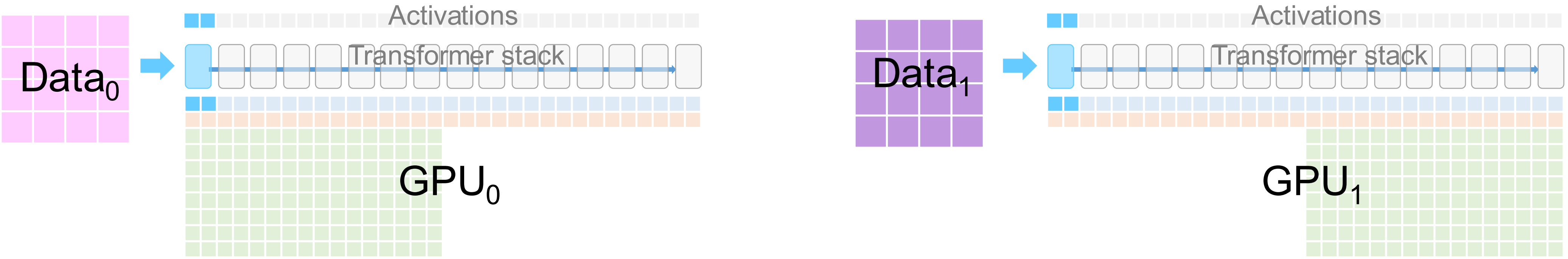
ZeRO Stage 1: Partitioning Optimizer States



- ZeRO Stage 1
- Partitions optimizer states across GPUs

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

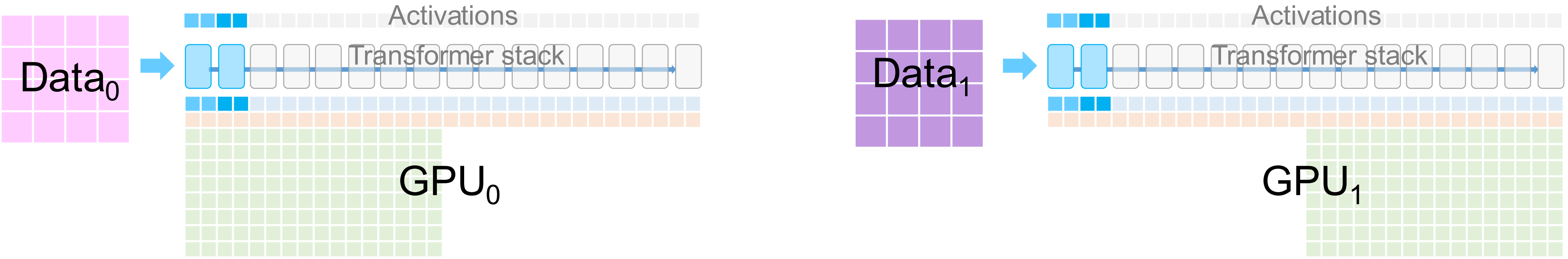


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

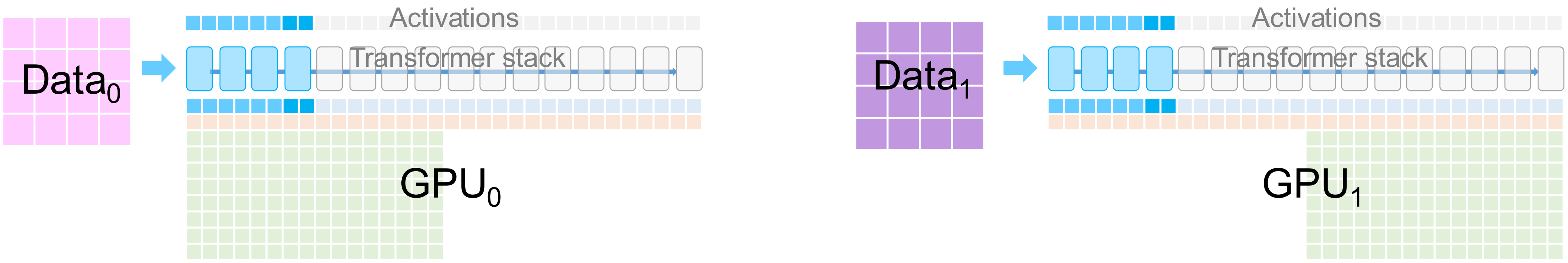


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

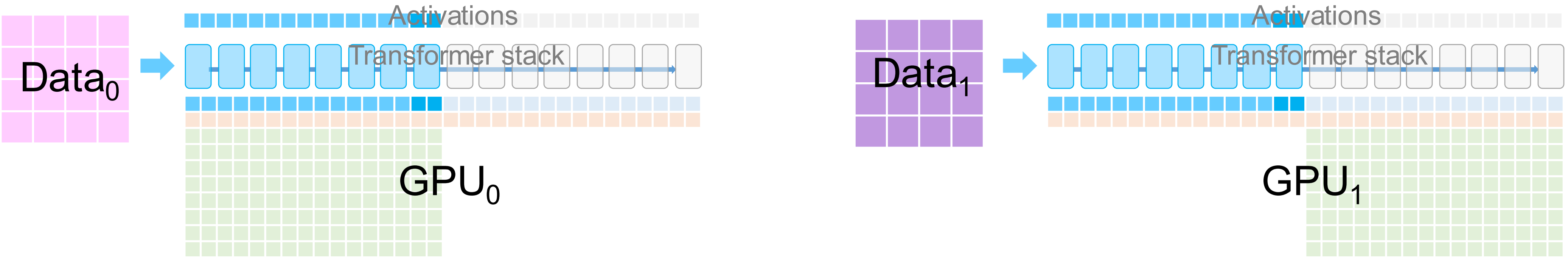


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

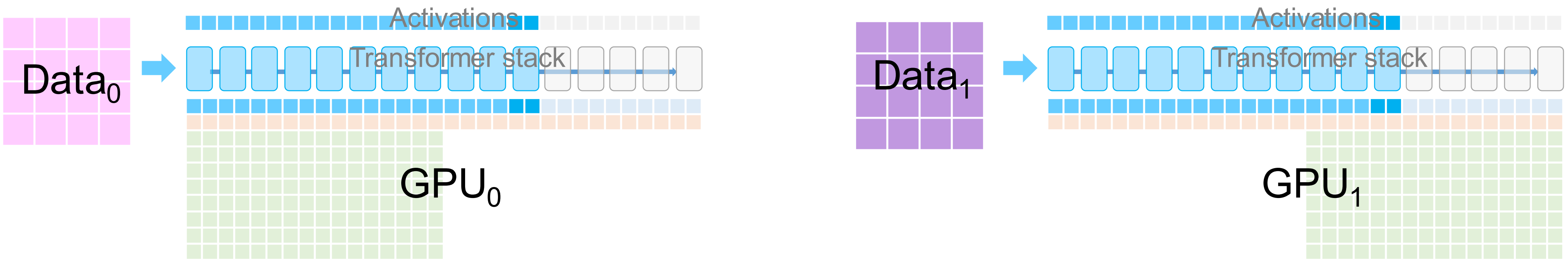


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

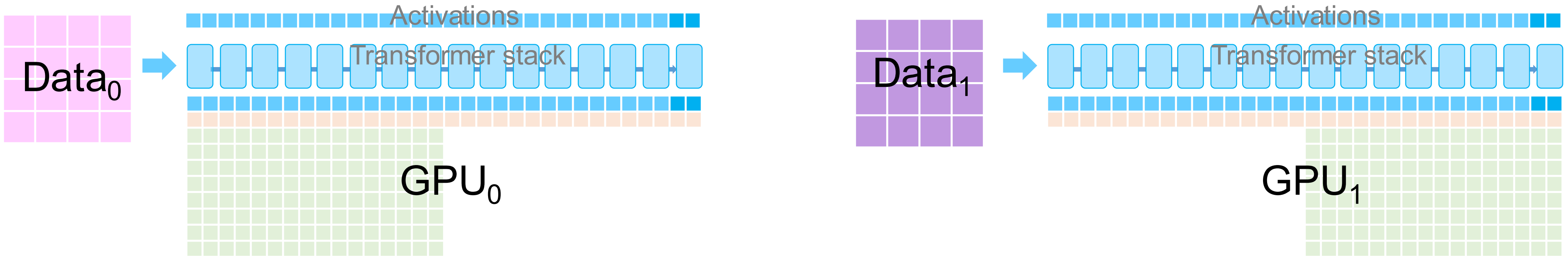


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

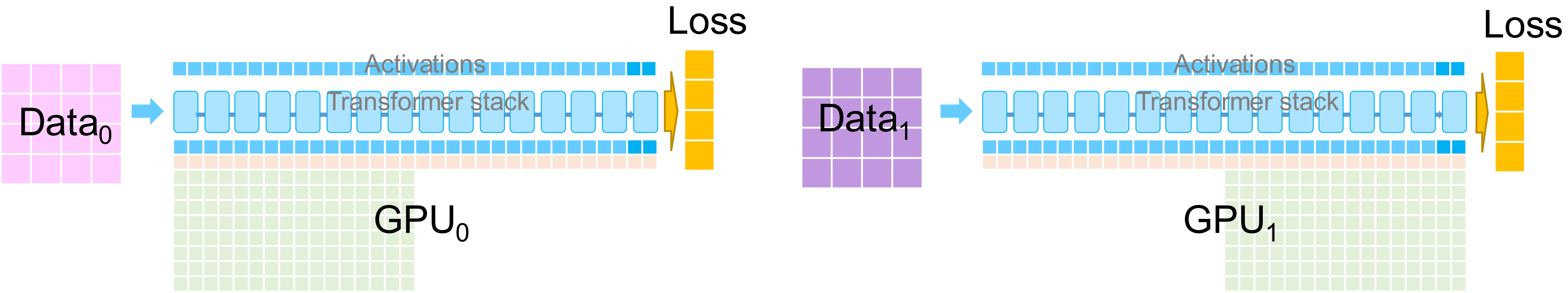


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

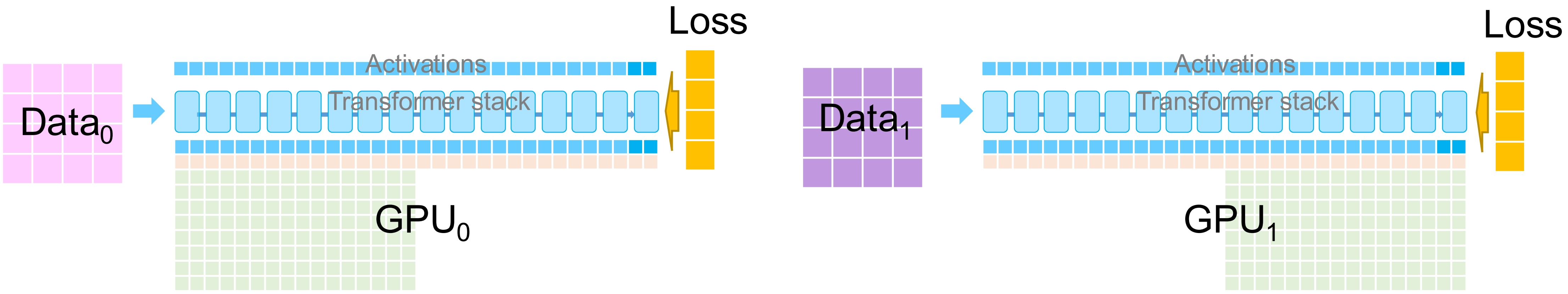


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

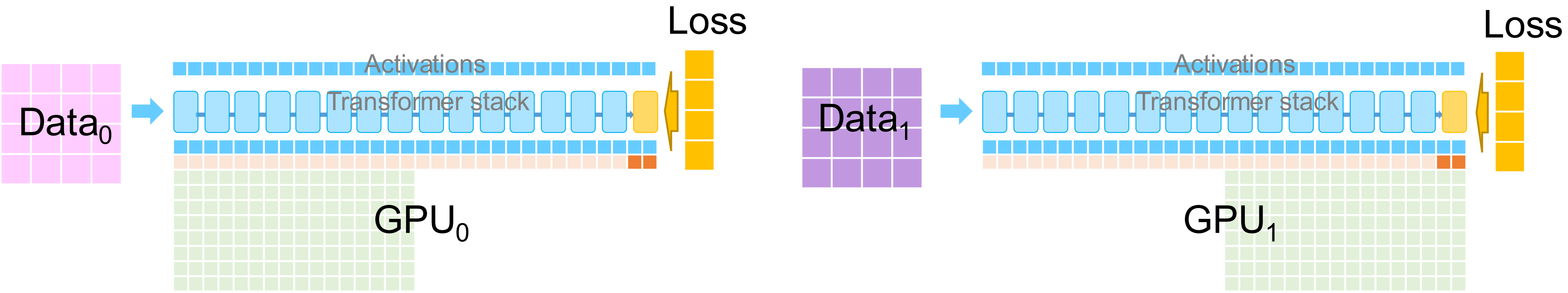


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

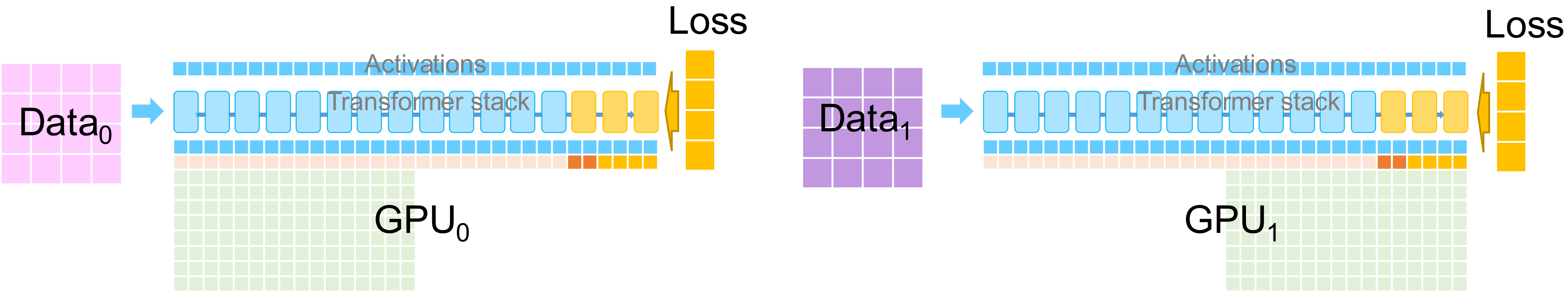


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

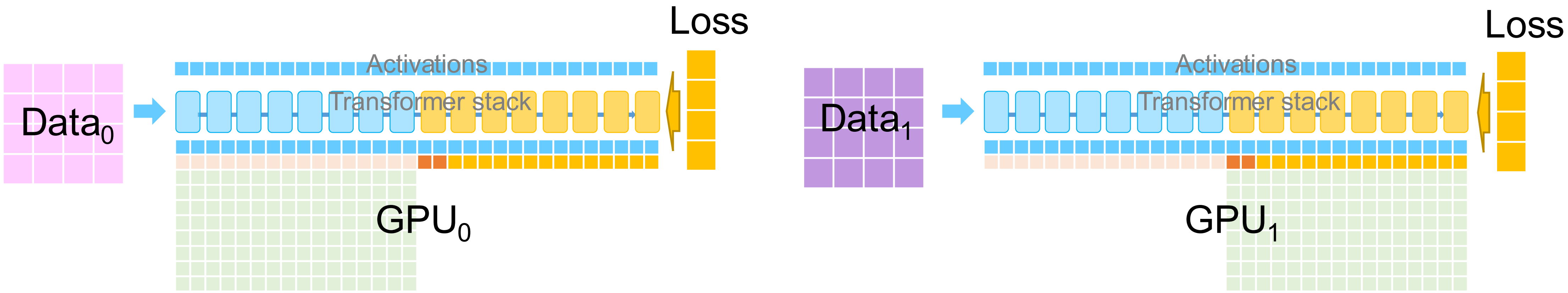


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

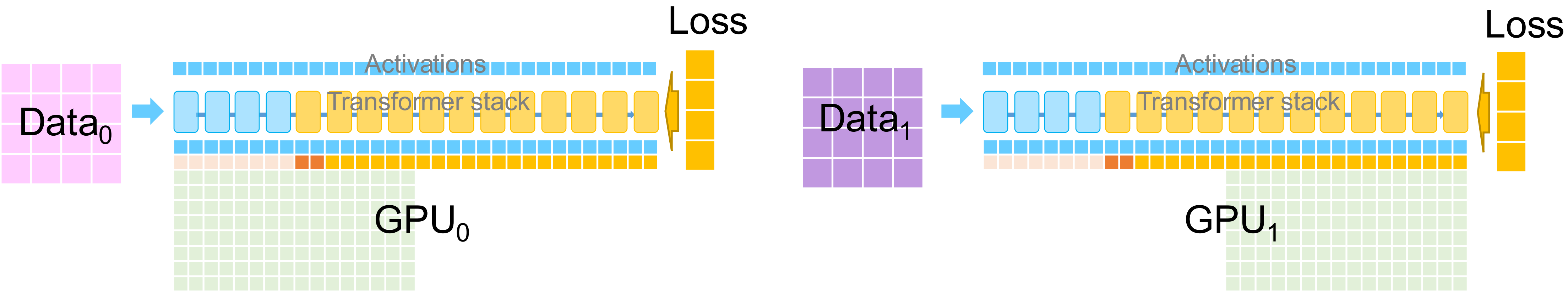


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

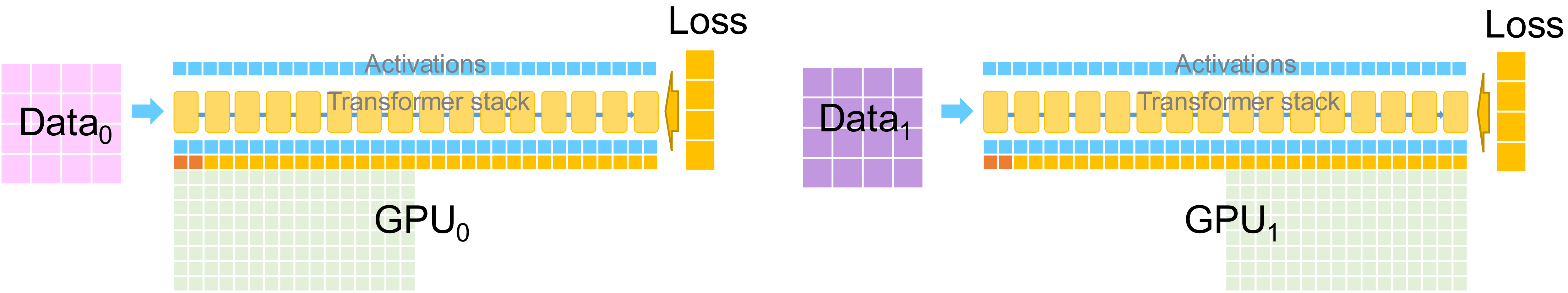


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

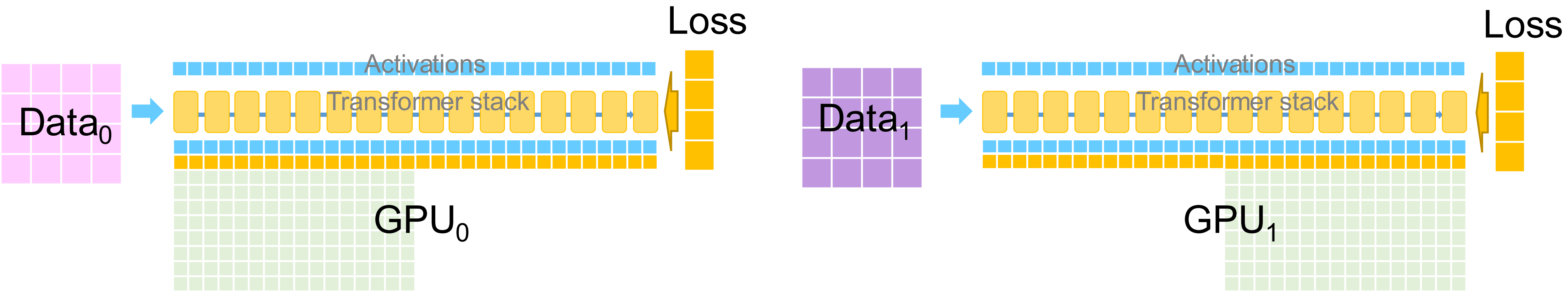


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

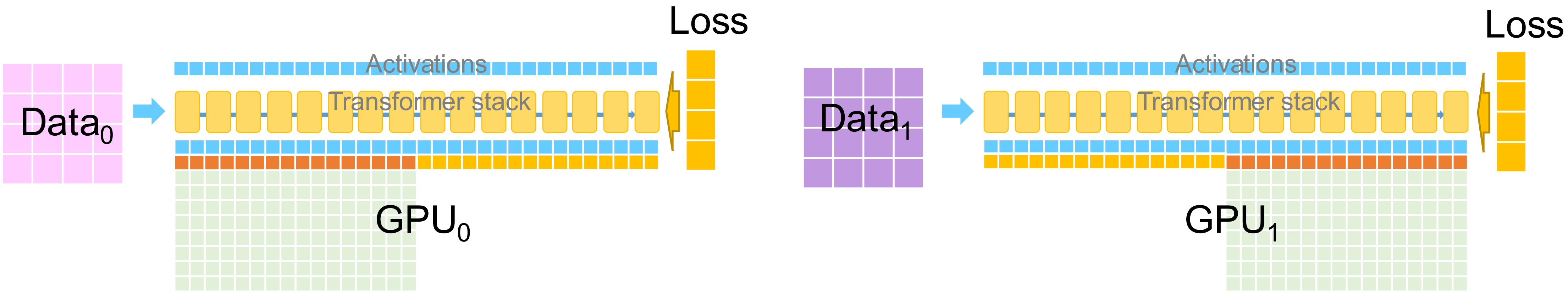


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

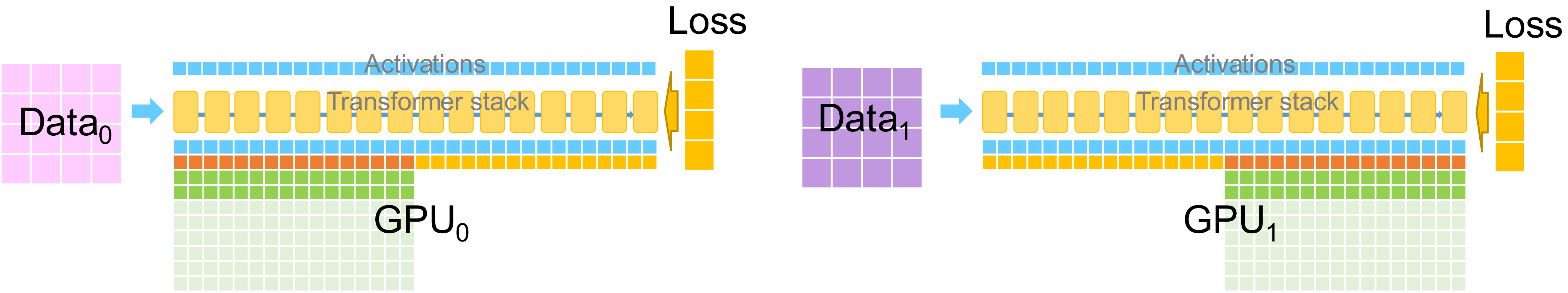


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

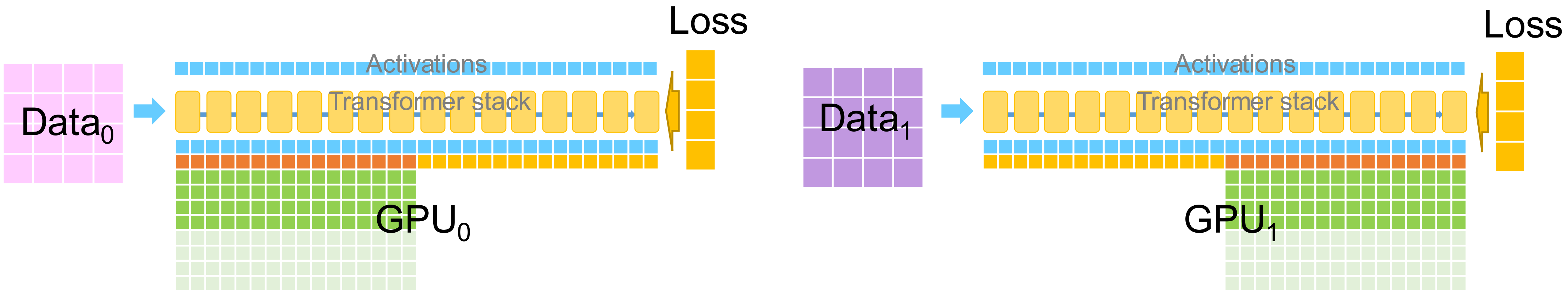


- ZeRO Stage 1
- Partitions optimizer states across GPUs
- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

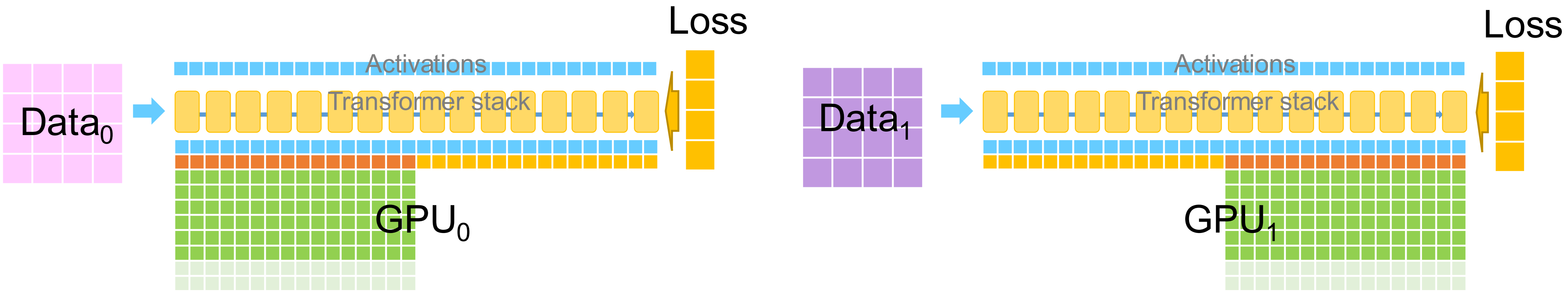


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

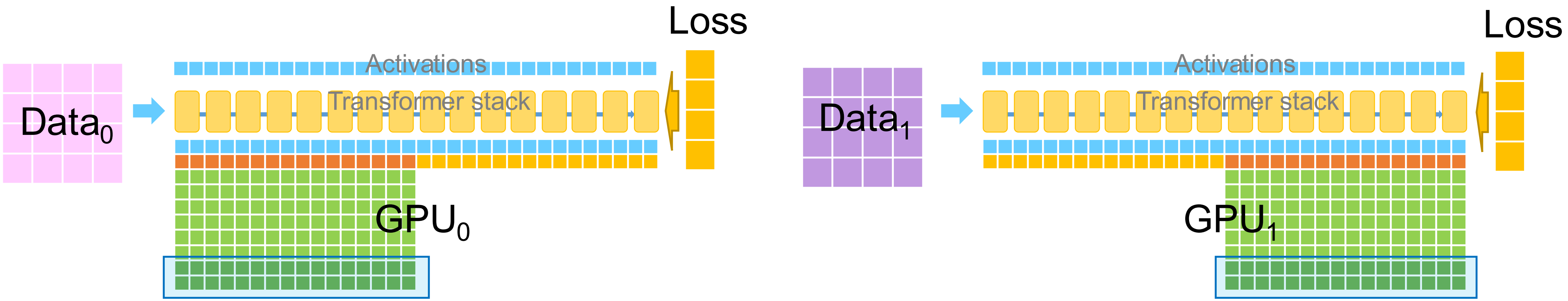


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

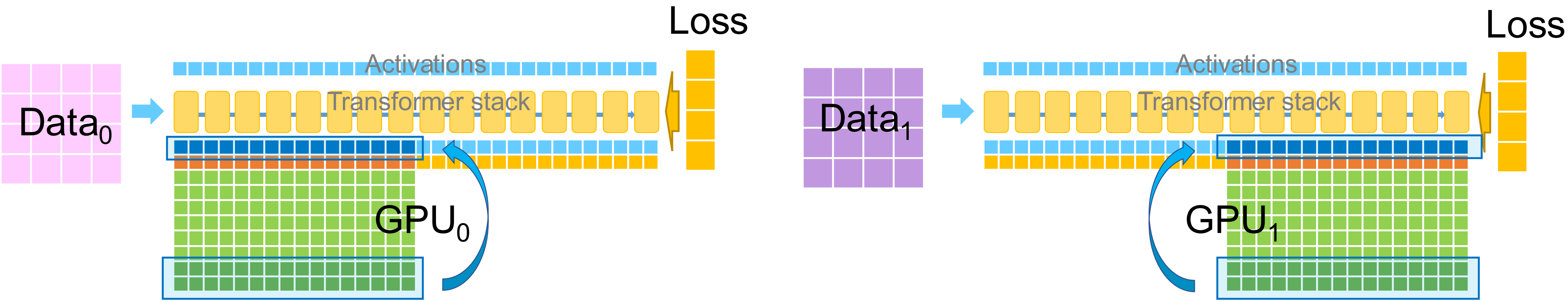


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

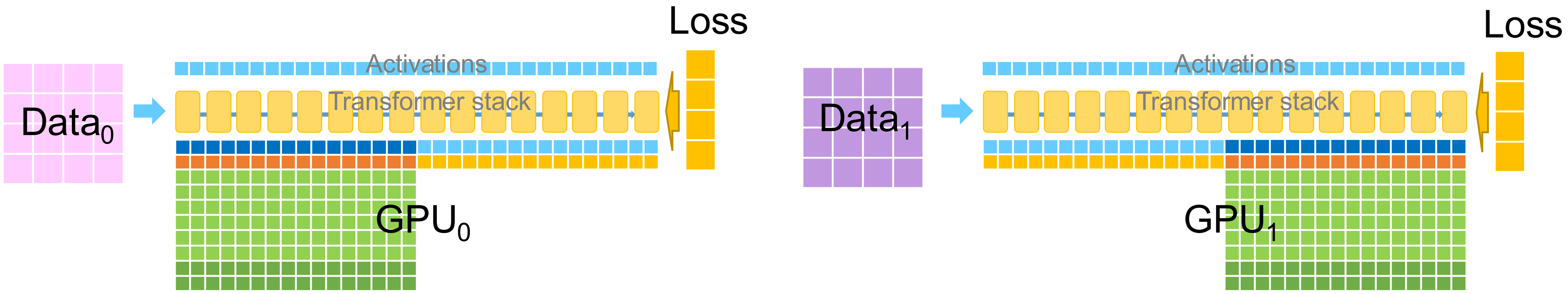
ZeRO Stage 1: Partitioning Optimizer States



- 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

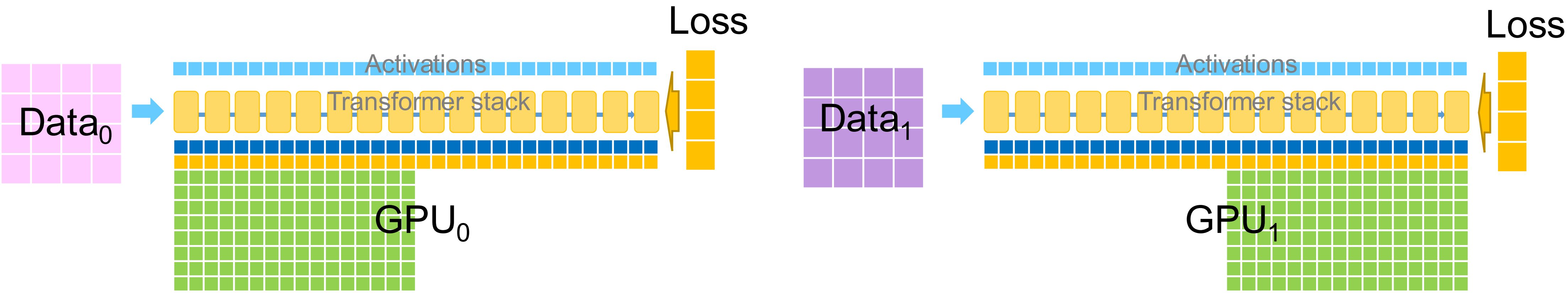


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer
- Update the FP16 weights
- All Gather the FP16 weights to complete the iteration

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

Adapted from Minjia Zhang, DeepSpeed Presentation

ZeRO Stage 1: Partitioning Optimizer States

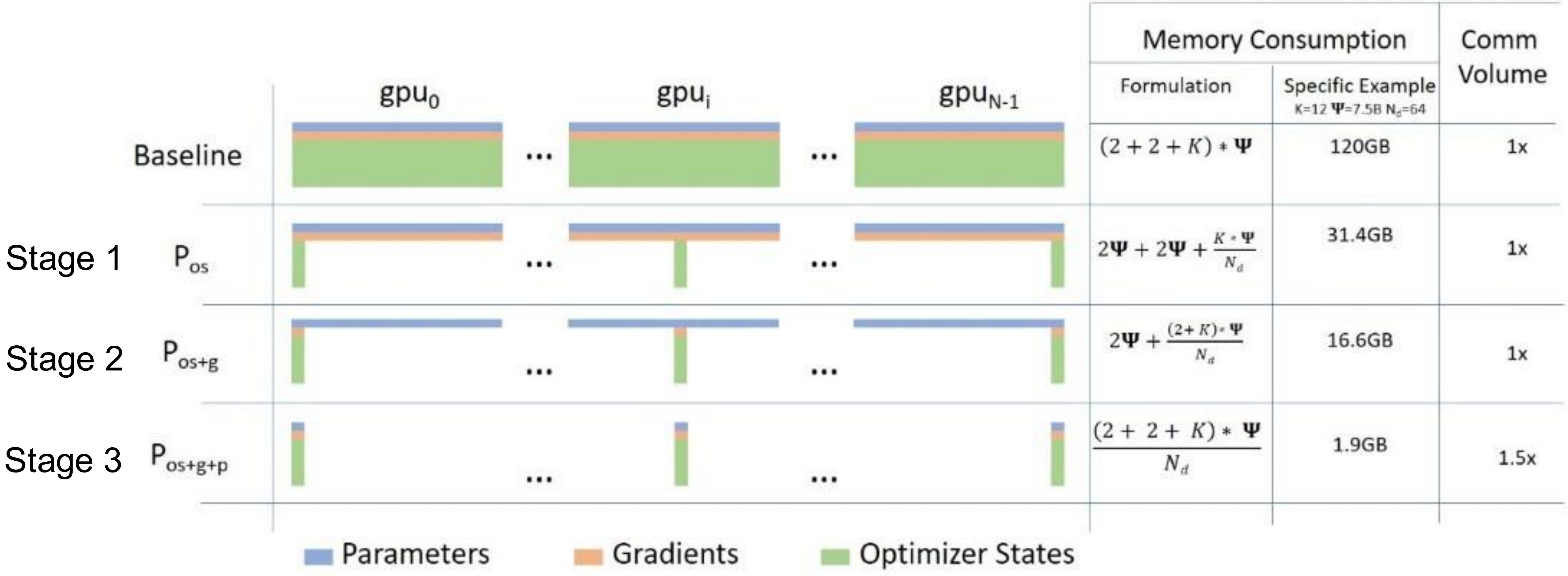


- Run Forward across the transformer blocks
- Backward propagation to generate FP16 gradients and AllReduce to average
- Update the FP32 weights with ADAM optimizer
- Update the FP16 weights
- All Gather the FP16 weights to complete the iteration

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

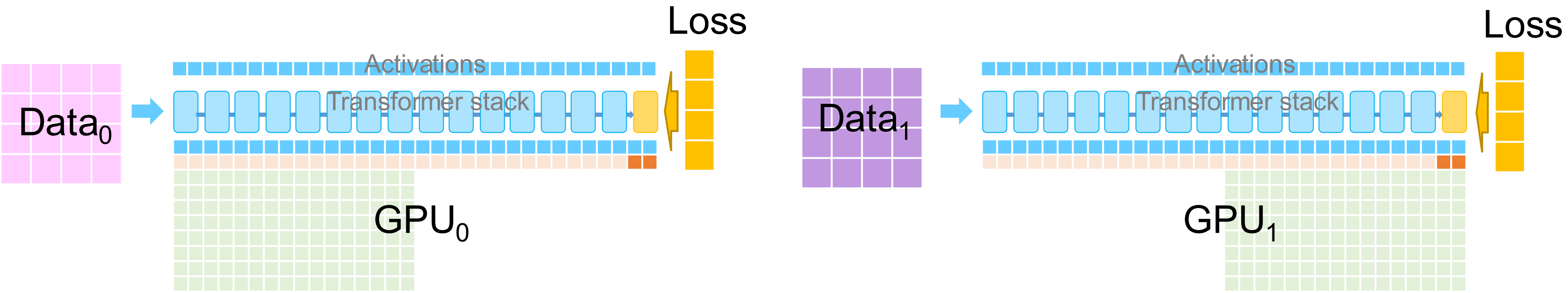
ZeRO: Zero Redundancy Optimizer

- 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

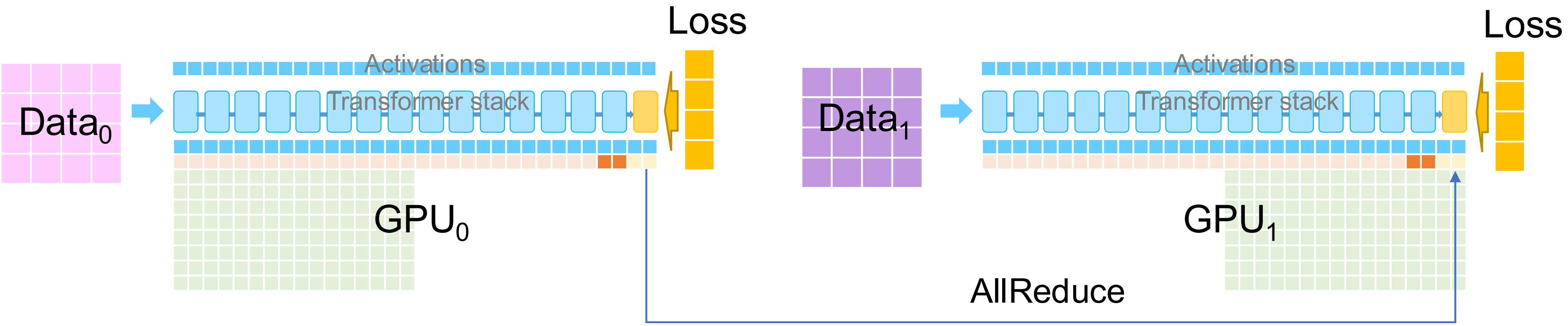
ZeRO Stage 2: Partitioning Gradients



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

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

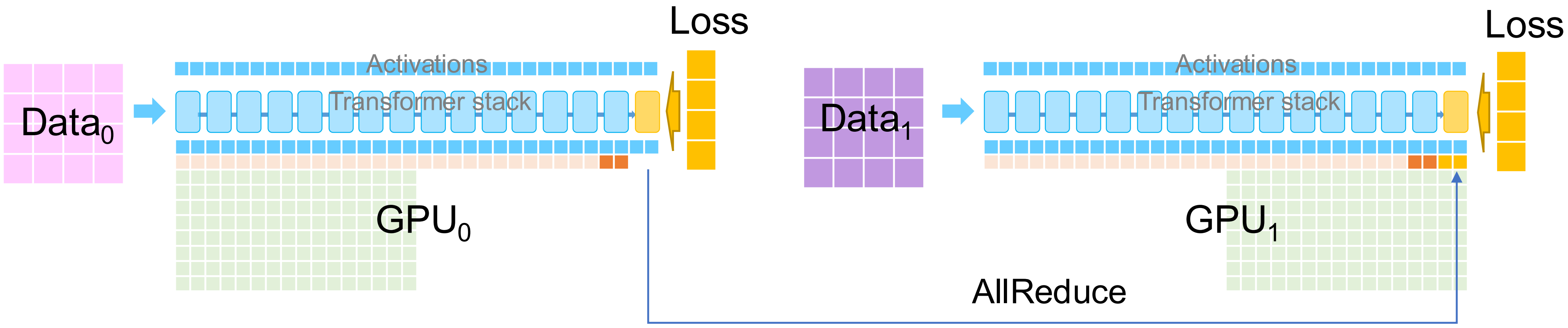
ZeRO Stage 2: Partitioning Gradients



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

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

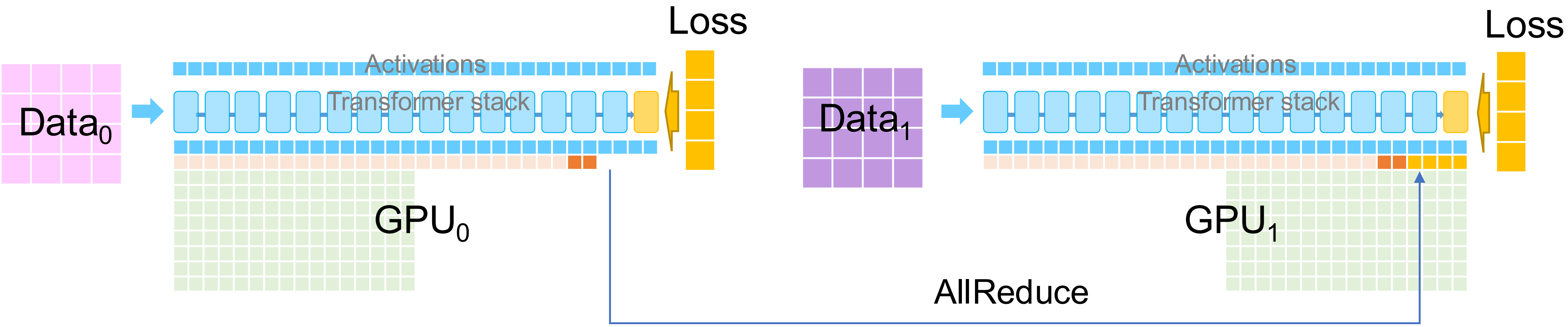
ZeRO Stage 2: Partitioning Gradients



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

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

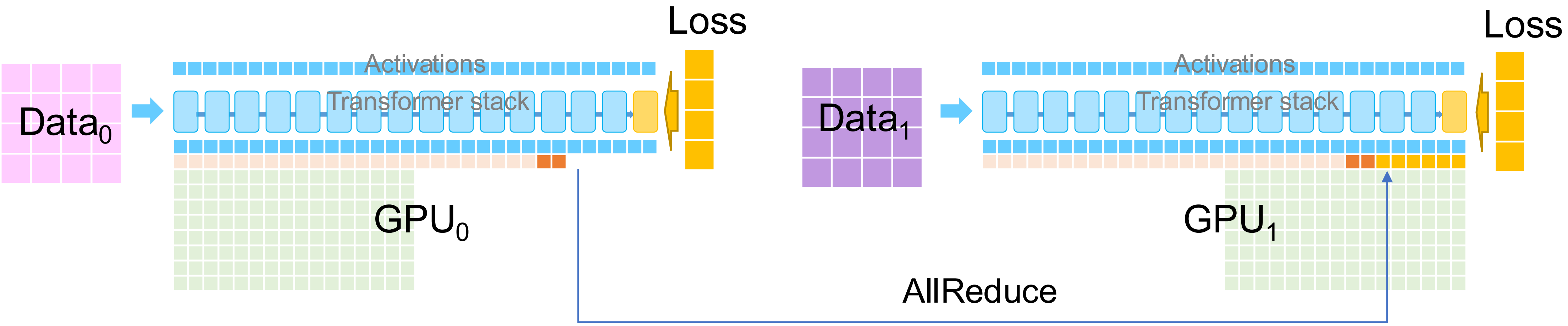
ZeRO Stage 2: Partitioning Gradients



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

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

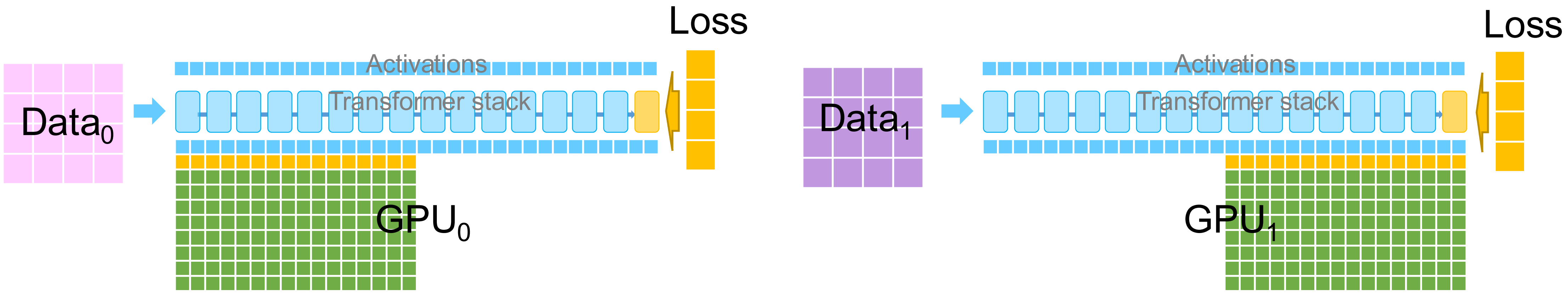
ZeRO Stage 2: Partitioning Gradients



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

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

ZeRO Stage 2: Partitioning Gradients

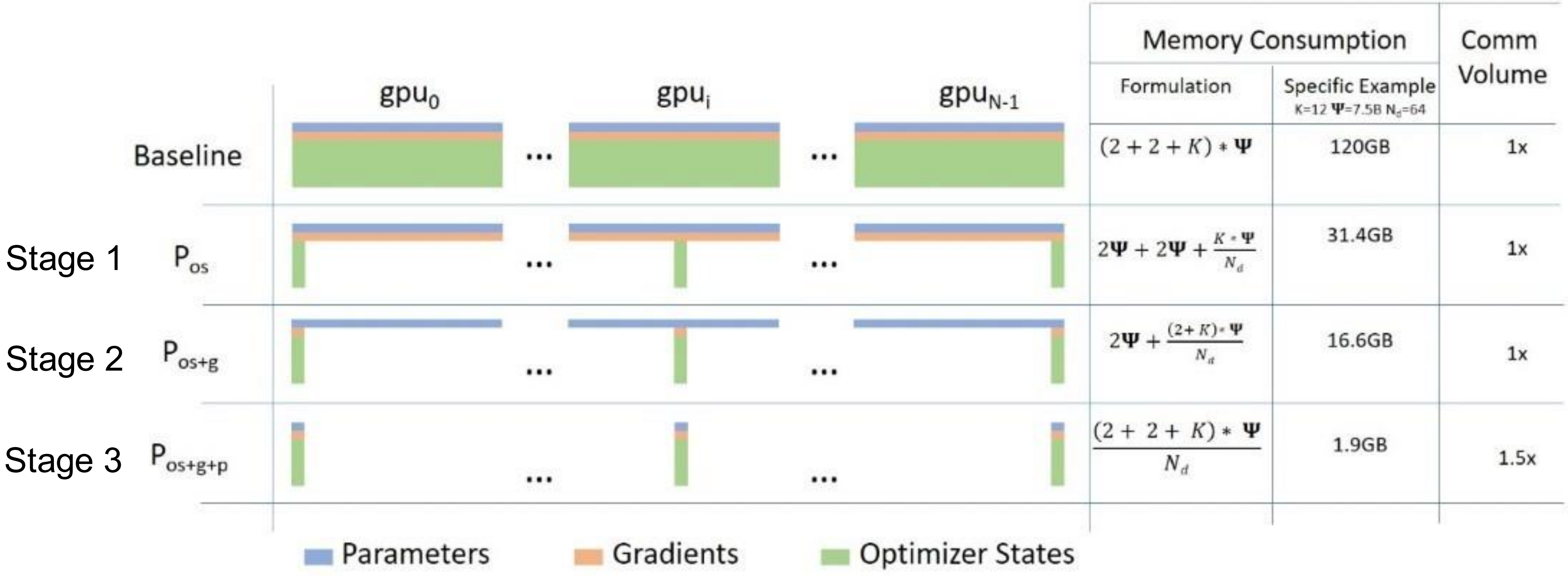


- Partitioning gradients across GPUs
- Reduce gradients on GPUs responsible for updating parameters

🟦 Parameters 🟧 Gradients 🟩 Optimizer States

ZeRO: Zero Redundancy Optimizer

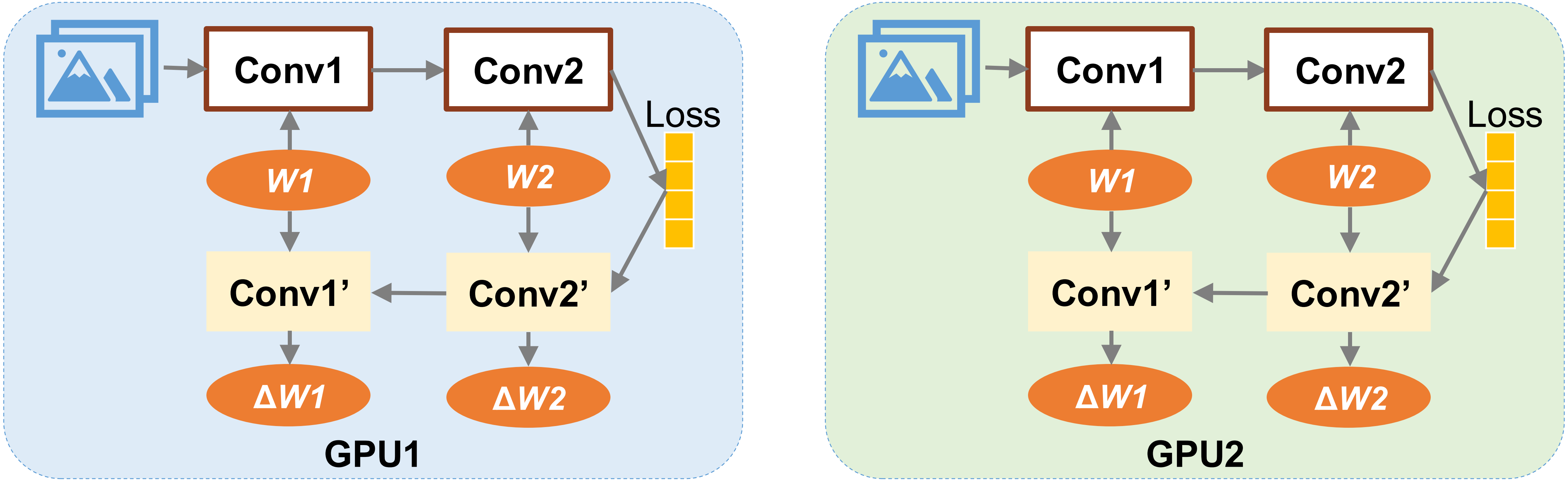
- Progressive memory savings and communication volume
- Turning NLR 17.2B is powered by Stage 1 and Megatron



		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

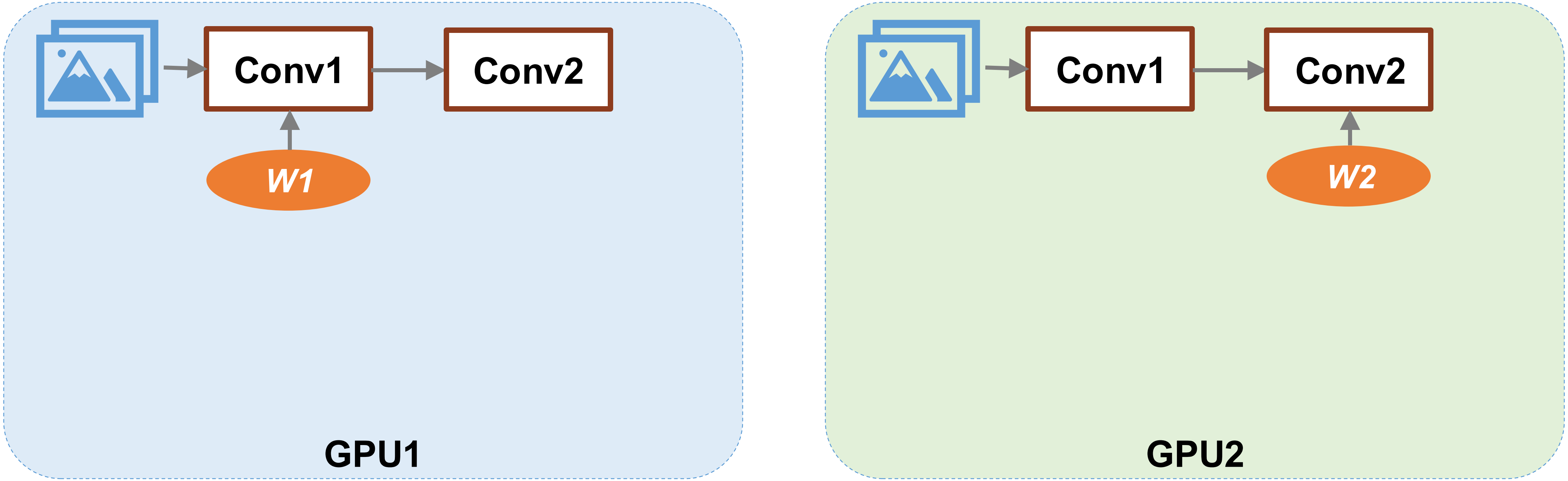
ZeRO Stage 3: Partitioning Parameters

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



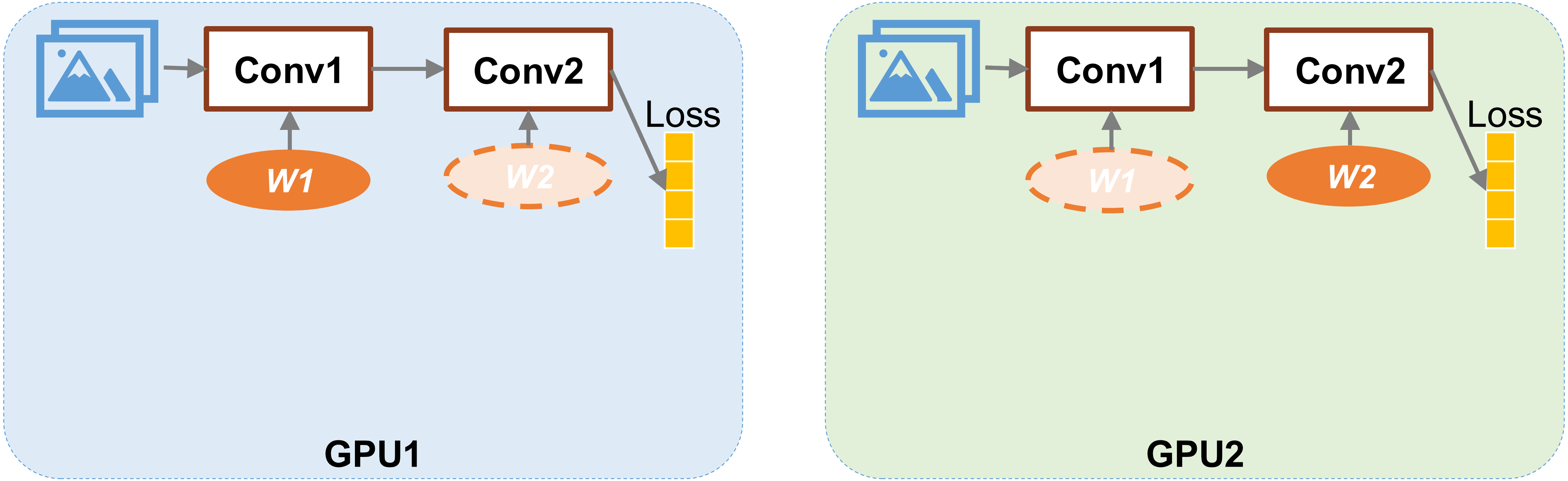
ZeRO Stage 3: Partitioning Parameters

- In ZeRO, model parameters are partitioned across GPUs



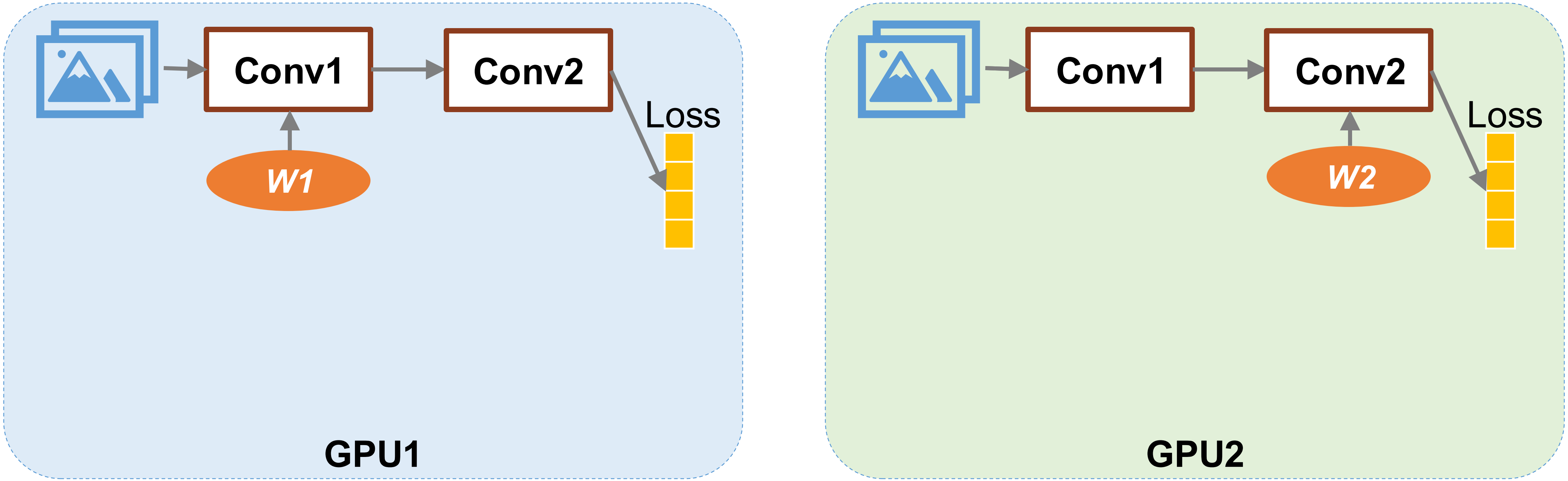
ZeRO Stage 3: Partitioning Parameters

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



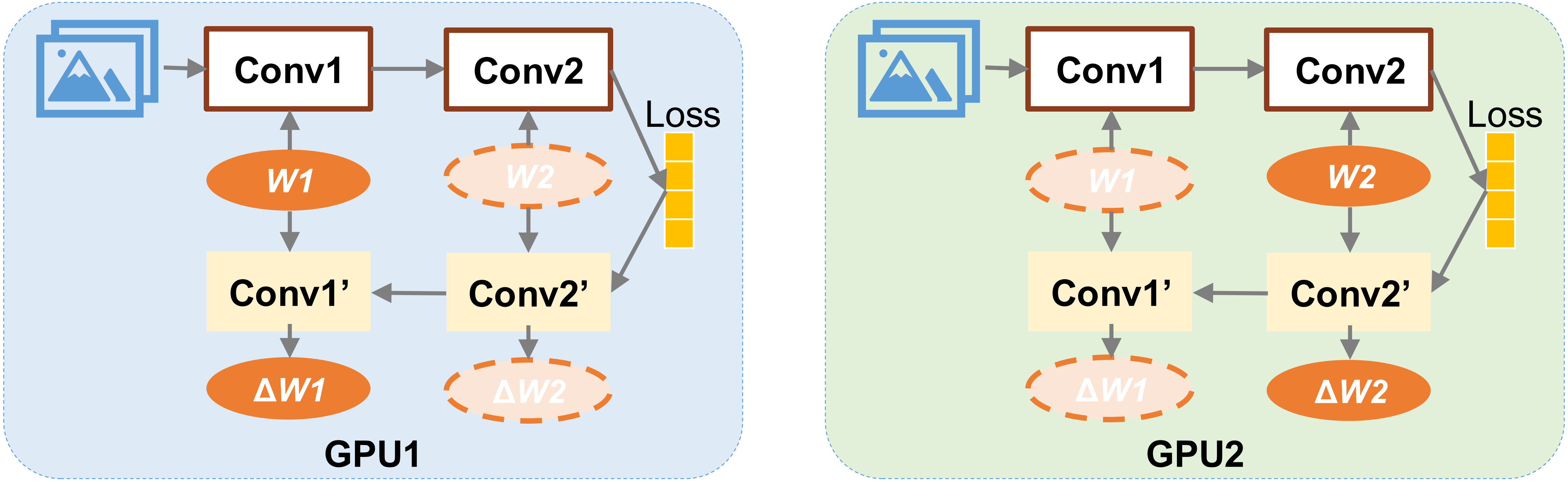
ZeRO Stage 3: Partitioning Parameters

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



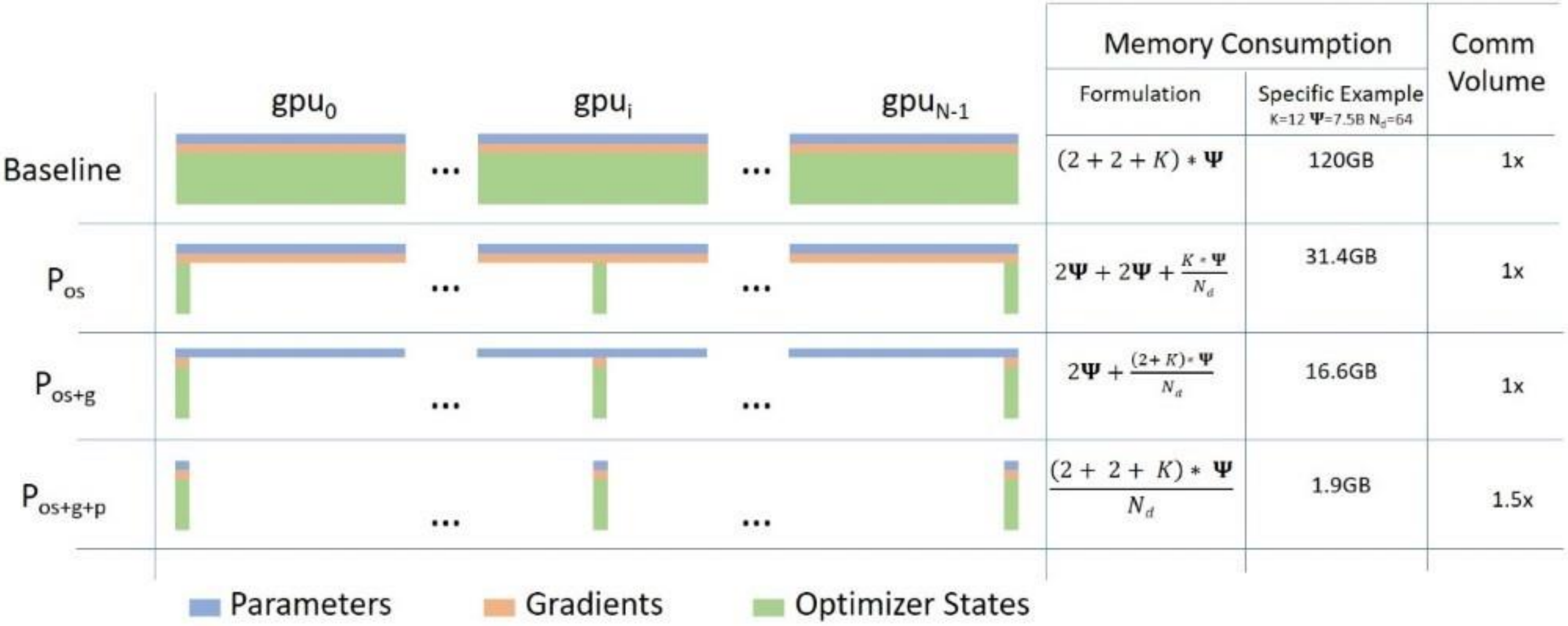
ZeRO Stage 3: Partitioning Parameters

- 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