

Scaling Up LLM Pretraining: Parallel Training

Chenyan Xiong

11-667

Outline

Optimization

- Optimization Basics
- Numerical Types

Parallel Training

- Data Parallelism
- Pipeline Parallelism
- Tensor Parallelism
- Combination of Parallelism
- ZeRO Optimizer

Optimization: Recap of Stochastic Gradient Descent

In deep learning, mini-batch learning is the norm and Stochastic Gradient Descent (SGD) is the basis optimizer

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

Gradient at step t of loss function $f()$

$$\theta_t = \theta_{t-1} - \alpha g_t$$

Updating with step size α

Compared to classic convex optimization:

- Each step only uses a small sub sample of data: stochastic sampling
- Non-convex optimization has many local optimal with different effectiveness

Optimization: Challenge of SGD

In deep learning, mini-batch learning is the norm and Stochastic Gradient Descent (SGD) is the basis optimizer

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - \underline{\alpha} g_t$$

Gradient at step t of loss function $f()$

Updating with step size α

Challenge: How to select the right step size?

- Different parameters have different behaviors:
 - norm, sensitivity, influence to optimization process, etc.
 - thus have different preferences on step size
- No way to manually tune step size per parameter
 - Millions or billions of hyperparameters to tune

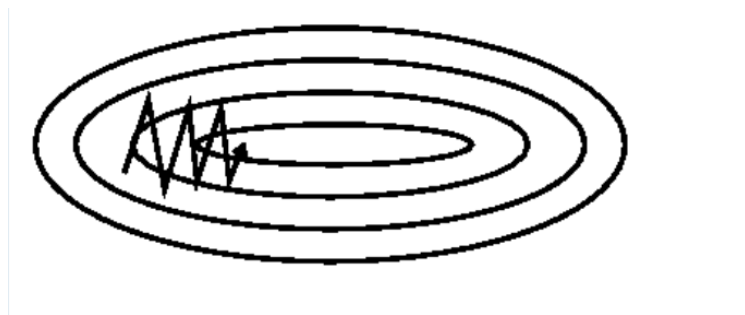


Figure 1: SGD on two parameter loss contours [1]

Optimization: Challenge of SGD

In deep learning, mini-batch learning is the norm and Stochastic Gradient Descent (SGD) is the basis optimizer

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

Gradient at step t of loss function $f()$

$$\theta_t = \theta_{t-1} - \underline{\alpha} g_t$$

Updating with step size α

Challenge: How to select the right step size?

→Solution: Dynamic learning rate per parameter

Adaptive gradient methods (AdaGrad [2])

$$\theta_t = \theta_{t-1} - \frac{\alpha g_t}{\sqrt{\sum_{i=1}^t g_i^2}}$$

Reweight per parameter step size by its accumulated past norm

Optimization: Challenge of SGD

In deep learning, mini-batch learning is the norm and Stochastic Gradient Descent (SGD) is the basis optimizer

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

Gradient at step t of loss function $f()$

$$\theta_t = \theta_{t-1} - \underline{\alpha} g_t$$

Updating with step size α

Challenge: How to select the right step size?

→Solution: Dynamic learning rate per parameter

Adaptive gradient methods (AdaGrad [2])

$$\theta_t = \theta_{t-1} - \frac{\alpha g_t}{\sqrt{\sum_{i=1}^t g_i^2}}$$

Reweight per parameter step size by its accumulated past norm

- The more a parameter has been updated previously $\sqrt{\sum_{i=1}^t g_i^2} \uparrow$, the less its step size
- Sparse features with fewer past gradients $\sqrt{\sum_{i=1}^t g_i^2} \downarrow$ get boosted

Optimization: Challenge of SGD

In deep learning, mini-batch learning is the norm and Stochastic Gradient Descent (SGD) is the basis optimizer

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

$$\theta_t = \theta_{t-1} - \alpha \underline{g_t}$$

Gradient at step t of loss function $f()$

Updating with step size α

Challenge: Local updates

- Only uses information from current mini-batch
 - Can easily stuck in local optima

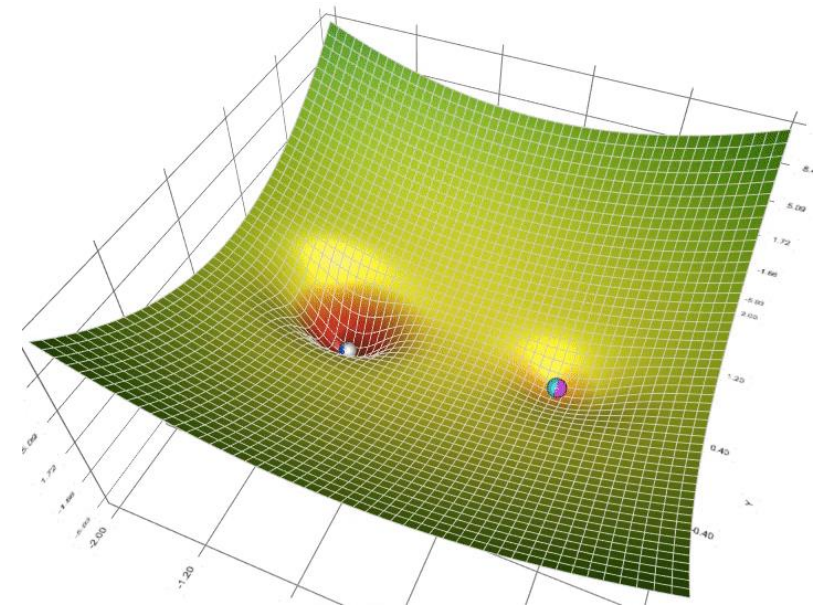


Figure 2: Optimization with Local Optima [3]

Optimization: Challenge of SGD

In deep learning, mini-batch learning is the norm and Stochastic Gradient Descent (SGD) is the basis optimizer

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

Gradient at step t of loss function $f()$

$$\theta_t = \theta_{t-1} - \alpha \underline{g_t}$$

Updating with step size α

Challenge: Local updates

→ Solution: Momentum [4]

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta} f_t(\theta_{t-1})$$

Momentum of Gradient

$$\theta_t = \theta_{t-1} - \alpha m_t$$

Updating with gradient momentum

Optimization: Challenge of SGD

In deep learning, mini-batch learning is the norm and Stochastic Gradient Descent (SGD) is the basis optimizer

$$g_t = \nabla_{\theta} f_t(\theta_{t-1})$$

Gradient at step t of loss function $f()$

$$\theta_t = \theta_{t-1} - \alpha \underline{g_t}$$

Updating with step size α

Challenge: Local updates

→ Solution: Momentum [4]

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_{\theta} f_t(\theta_{t-1})$$

Momentum of Gradient

$$\theta_t = \theta_{t-1} - \alpha m_t$$

Updating with gradient momentum



(a) SGD without momentum



(b) SGD with momentum

Figure 3: SGD with and without Momentum [1]

Optimization: Adam Optimizer

Adam: Adaptive Moment Estimation [4]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Optimization: Adam Optimizer

Adam: Adaptive Moment Estimation [4]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates } Hyperparameters that you can/should tune

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Initializations

Optimization: Adam Optimizer

Adam: Adaptive Moment Estimation [4]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates } Hyperparameters that you can/should tune

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)

Standard back-propagation for raw gradients

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Initializations

Optimization: Adam Optimizer

Adam: Adaptive Moment Estimation [4]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates } Hyperparameters that you can/should tune

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)

Standard back-propagation for raw gradients

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

} Get 1st and 2nd order momentum of gradient

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Initializations

Optimization: Adam Optimizer

Adam: Adaptive Moment Estimation [4]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates } Hyperparameters that you can/should tune

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

} Initializations

while θ_t not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)

Standard back-propagation for raw gradients

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

} Get 1st and 2nd order momentum of gradient

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)

} Correct momentum bias

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)

end while

return θ_t (Resulting parameters)

Optimization: Adam Optimizer

Adam: Adaptive Moment Estimation [4]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

Require: α : Stepsize

Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates } Hyperparameters that you can/should tune

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0 : Initial parameter vector

$$m_0 \leftarrow 0 \text{ (Initialize 1}^{\text{st}} \text{ moment vector)}$$
$$v_0 \leftarrow 0 \text{ (Initialize 2nd moment vector)}$$
$$t \leftarrow 0 \text{ (Initialize timestep)}$$
while θ_t not converged **do**
$$t \leftarrow t + 1$$
$$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1}) \text{ (Get gradients w.r.t. stochastic objective at timestep } t)$$

Standard back-propagation for raw gradients

$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t \text{ (Update biased first moment estimate)}$$
$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 \text{ (Update biased second raw moment estimate)}$$

Get 1st and 2nd order momentum of gradient

$$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t) \text{ (Compute bias-corrected first moment estimate)}$$
$$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t) \text{ (Compute bias-corrected second raw moment estimate)}$$

Correct momentum bias

$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) \text{ (Update parameters)}$$
end while

return θ_t (Resulting parameters)

Dynamic per-parameter step size by 2nd order momentum

Update by 1st order momentum

Optimization: Illustrations

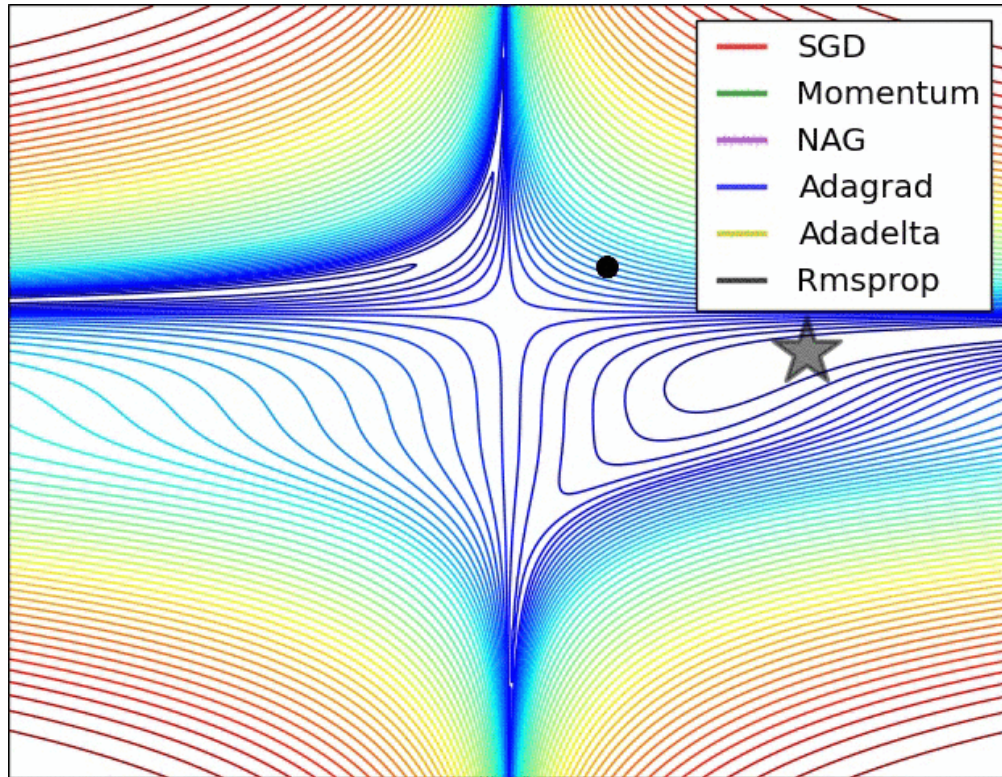


Figure 4: SGD optimization on loss surface contours [1]

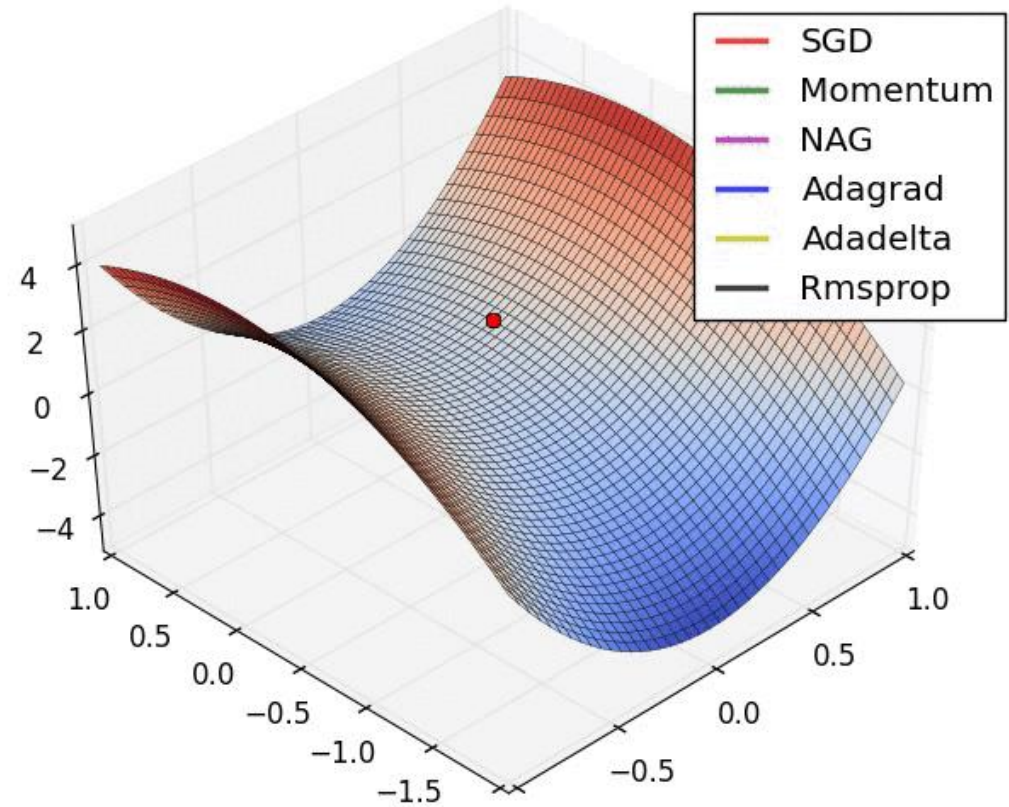


Figure 5: SGD optimization on saddle point [1]

Optimization: Extensions of Adams

Adam is the go-to optimizer for deep learning now

- Combines two effective idea: momentum and dynamic learning rates
- Works very well in a large range of network work architectures and tasks
- Many of LLMs are pretrained using Adam or its extensions. (Almost all common ones.)

Optimization: Extensions of Adams

Adam is the go-to optimizer for deep learning now

- Combines two effective idea: momentum and dynamic learning rates
- Works very well in a large range of network work architectures and tasks
- Many of LLMs are pretrained using Adam or its extensions. (Almost all common ones.)

Notable Extensions:

- Reducing the memory footprint of momentum states:
 - AdaFactor
 - 8-Bit Adam
- Better warmup optimizer stage:
 - RAdam
- More information in dynamic learning rate:
 - AdamSAGE (Sensitivity)
 - Sophia (2nd order optimizer approximation)

Outline

Optimization

- Optimization Basics
- **Numerical Types**

Parallel Training

- Data Parallelism
- Pipeline Parallelism
- Tensor Parallelism
- Combination of Combination
- ZeRO Optimizer

Numerical Types: Basic Types

Floating point formats supported by acceleration hardware

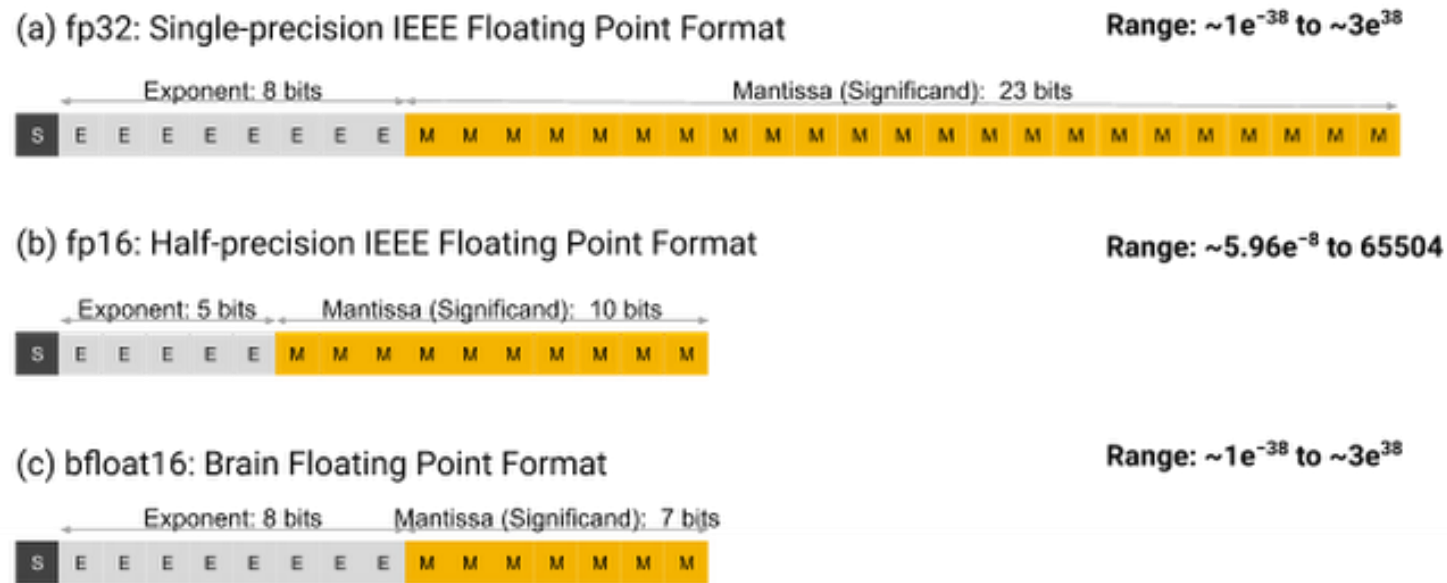


Figure 6: Floating Point Formats [5]

- BF16 is supported on TPU before LLM (2019 or earlier)
- FP32 and FP16 was the only option before A100. BF16 was not supported at hardware level
- BF16 was first supported in GPUs around 2021

Numerical Types: Neural Network Preferences

Neural networks prefer bigger range than better precision

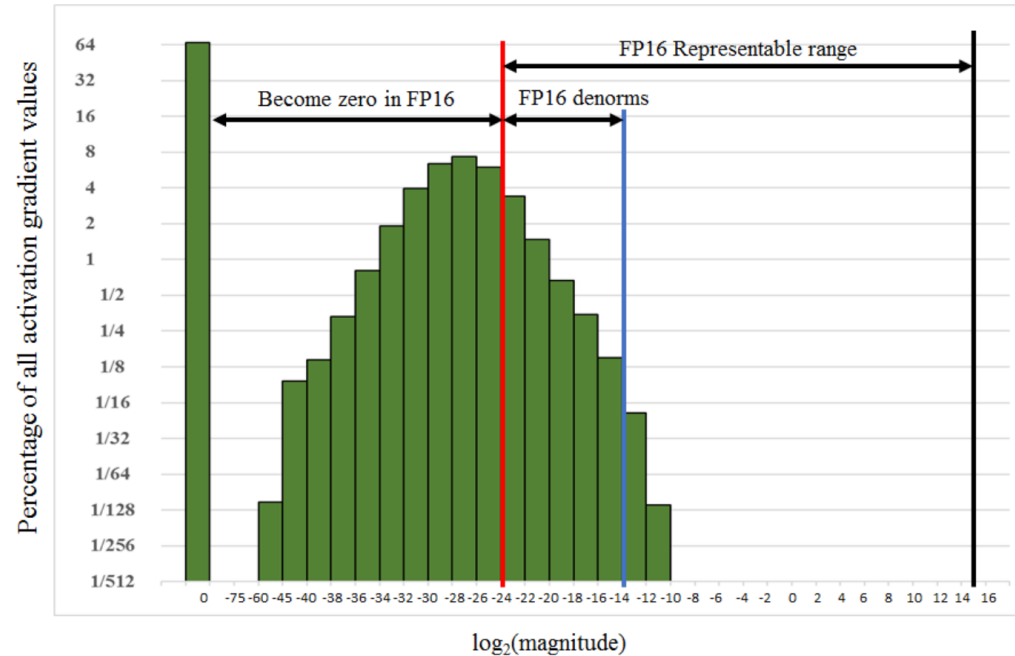


Figure 6: Histogram of gradient values in a FP32 training [6]

- Many computation needs bigger range than FP16

Numerical Types: Mixed Precision Training

Using different numerical types at different part of the training process

- Parameters, activations, and gradients often use FP16
- Optimizer states often needs FP32

Maintaining main copies of FP32 for calculations

Dynamically scaling up loss to fit gradients etc. in FP16 range

Numerical Types: Mixed Precision Training

Using different numerical types at different part of the training process

- Parameters, activations, and gradients often use FP16
- Optimizer states often needs FP32

Maintaining main copies of FP32 for calculations

Dynamically scaling up loss to fit gradients etc. in FP16 range

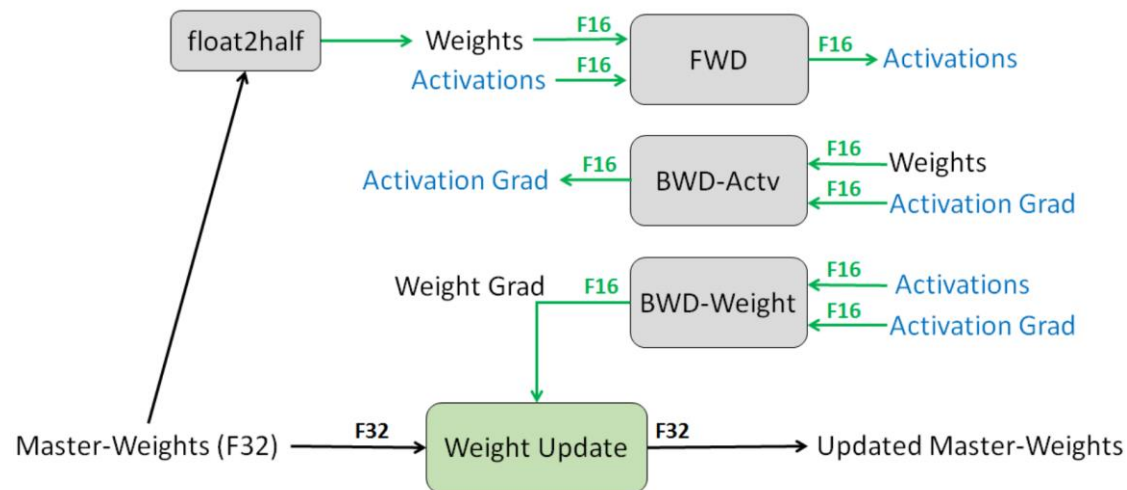


Figure 7: An Example Mixed Precision Training Set up [6]

Numerical Types: BF16

BF16 is the preferred numerical type on A100 and H100

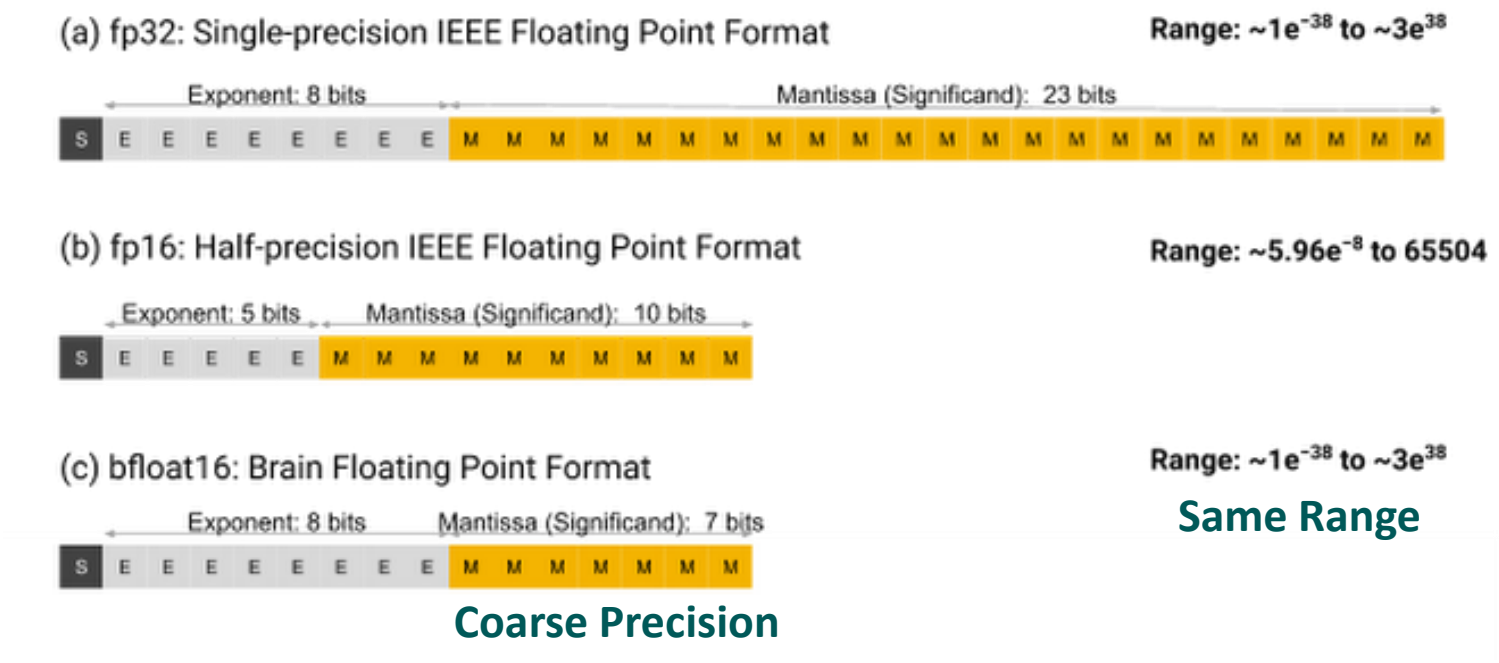


Figure 6: Floating Point Formats [5]

- Same range as FP32: eliminated the needs for mixed precision training while being way more stable
- Coarse precision: mostly fine, only a few places in neural network need more fine-grained precision

Outline

Optimization

- Optimization Basics
- Numerical Types

Parallel Training

- Data Parallelism
- Pipeline Parallelism
- Tensor Parallelism
- Combination of Parallelism
- ZeRO Optimizer

Parallel Training: Overview

As scale grows, training with one GPU is not enough

- There are many ways to improve efficiency on single-GPU training
 - Checkpointing: moving part of the operations to CPU memory
 - Quantizing different part of the optimization to reduce GPU memory cost
- Eventually more FLOPs are needed

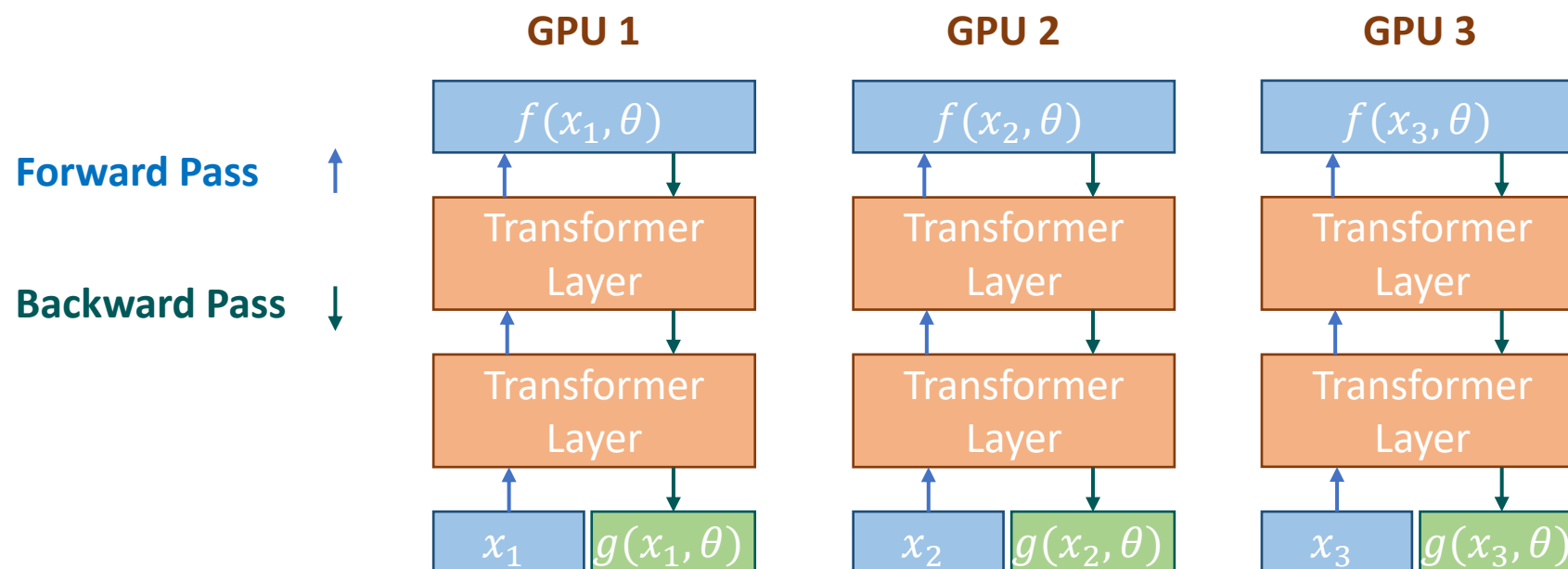
Different setups of parallel training:

- When model training can fit into single-GPU
 - Data parallelism
- When model training cannot fit into single-GPU
 - Model parallelism: pipeline or tensor

Parallel Training: Data Parallelism

Split training data batch into different GPUs

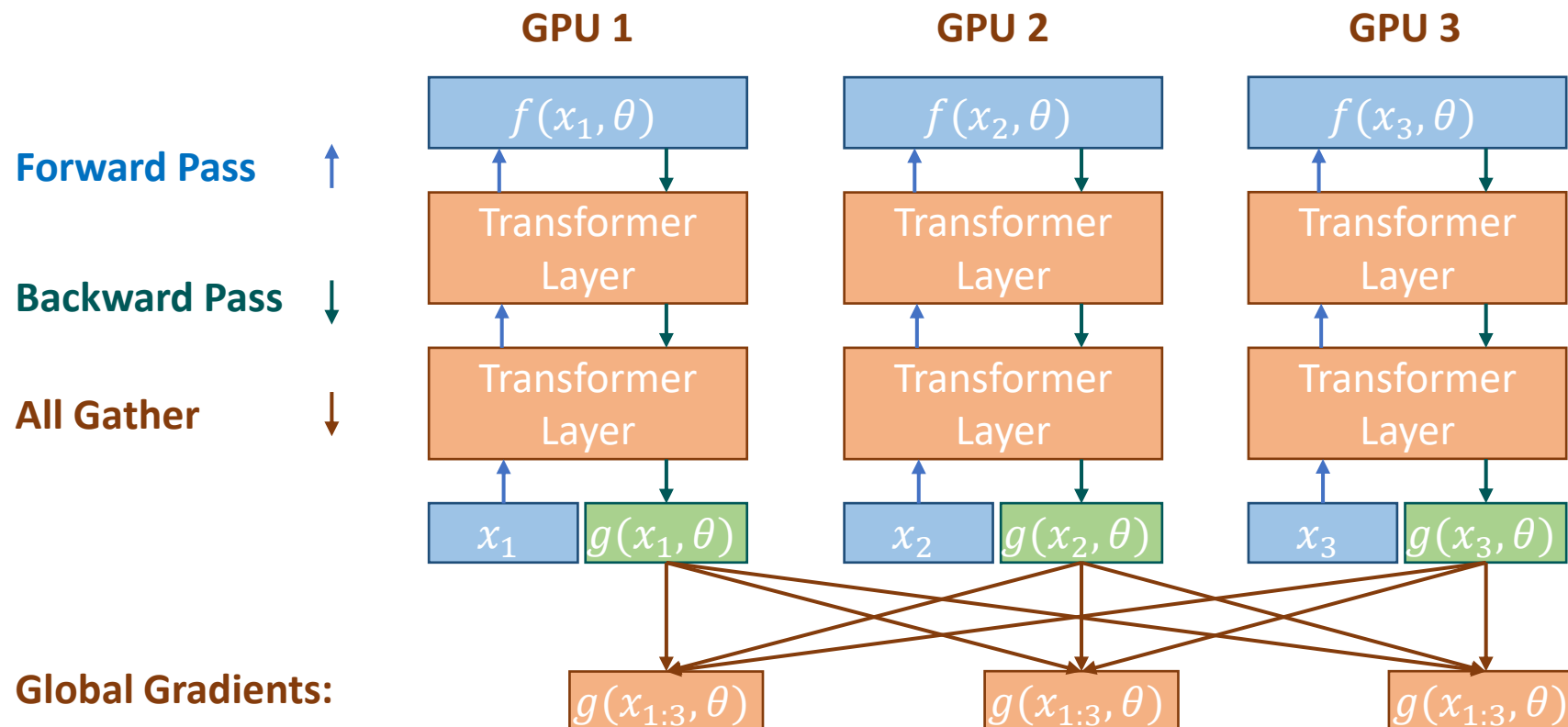
- Each GPU maintains its own copy of model and optimizer
- Each GPU gets a different local data batch, calculates its gradients
- Gather local gradients together to each GPU for global updates



Parallel Training: Data Parallelism

Split training data batch into different GPUs

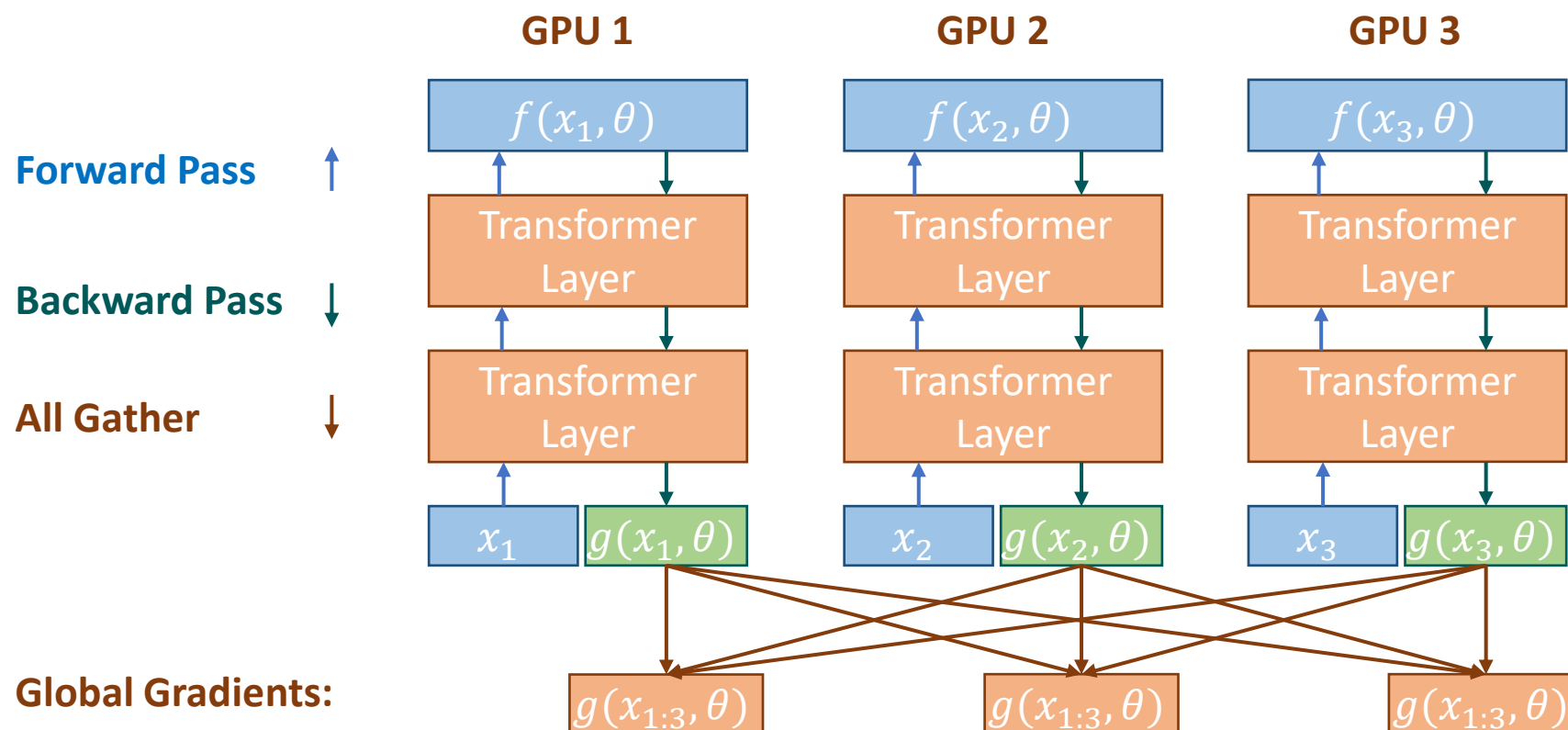
- Each GPU maintains its own copy of model and optimizer
- Each GPU gets a different local data batch, calculates its gradients
- Gather local gradients together to each GPU for global updates



Parallel Training: Data Parallelism

Split training data batch into different GPUs

- Each GPU maintains its own copy of model and optimizer
- Each GPU gets a different local data batch, calculates its gradients
- Gather local gradients together to each GPU for global updates



Communication:

- The full gradient tensor between every pair of GPUs, at each training batch.
- Not an issue between GPUs in the same machine or machines with infinity band
- Will need work around without fast cross-GPU connection

Parallel Training: Model Parallelism

LLM size grew quickly and passed the limit of single GPU memory

	Cost of 10B Model	Function to parameter count (Ψ)
Parameter Bytes	20GB	2Ψ
Gradient Bytes	20GB	2Ψ
Optimizer State: 1st Order Momentum	20GB	2Ψ
Optimizer State: 2nd Order Momentum	20GB	2Ψ
Total Per Model Instance	80GB	8Ψ

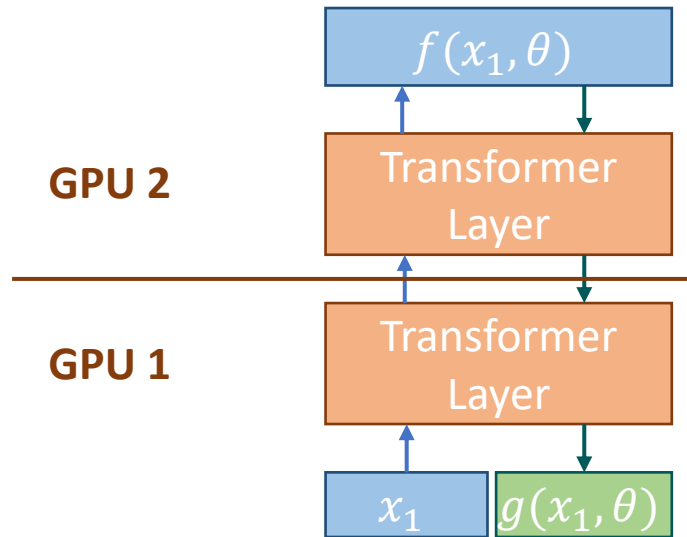
Table 1: Memory Consumption of Training Solely with BF16 (Ideal case) of a model sized Ψ

Solution: Split network parameters (thus their gradients and corresponding optimizer states) to different GPUs

Parallel Training: Model Parallelism

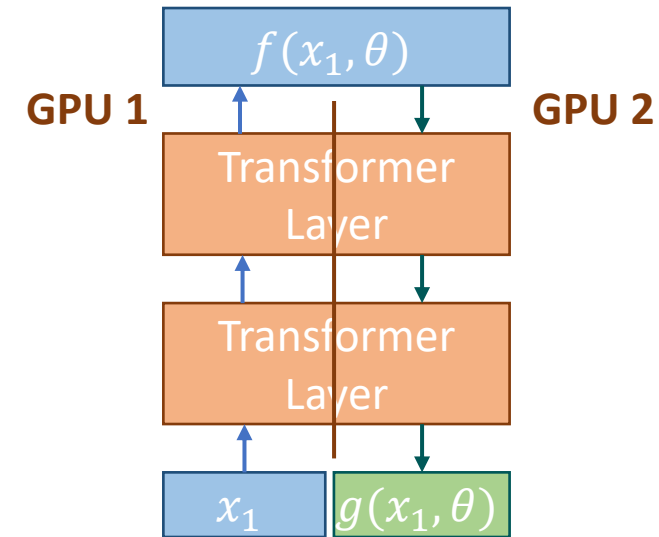
Two ways of splitting network parameters

Pipeline Parallelism



Split by Layers

Tensor Parallelism



Split Tensors

Parallel Training: Pipeline Parallelism

Split network by layers, aligning devices by layer order to a pipeline, and pass data through devices [7]

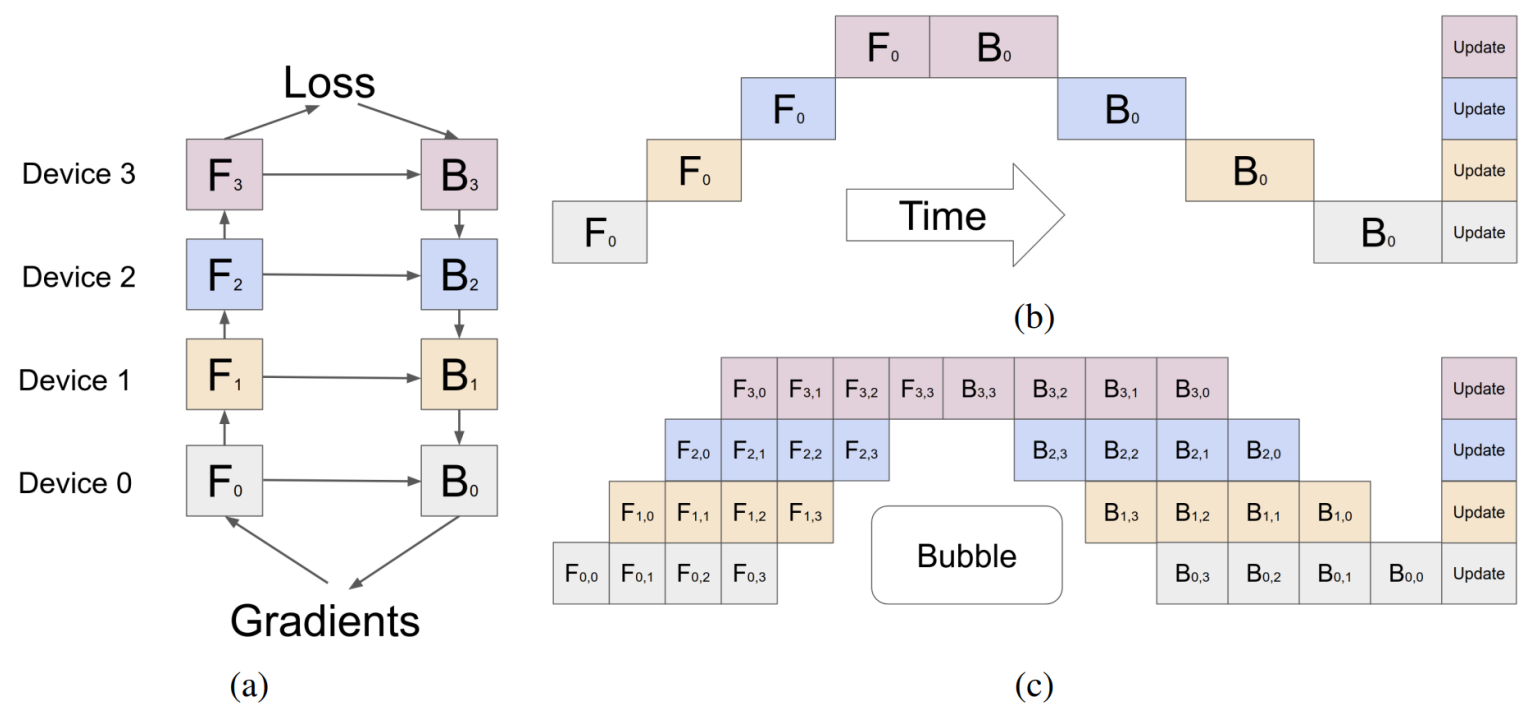


Figure 7: Illustration of Pipeline Parallelism [7]

Parallel Training: Pipeline Parallelism

Split network by layers, aligning devices by layer order to a pipeline, and pass data through devices [7]

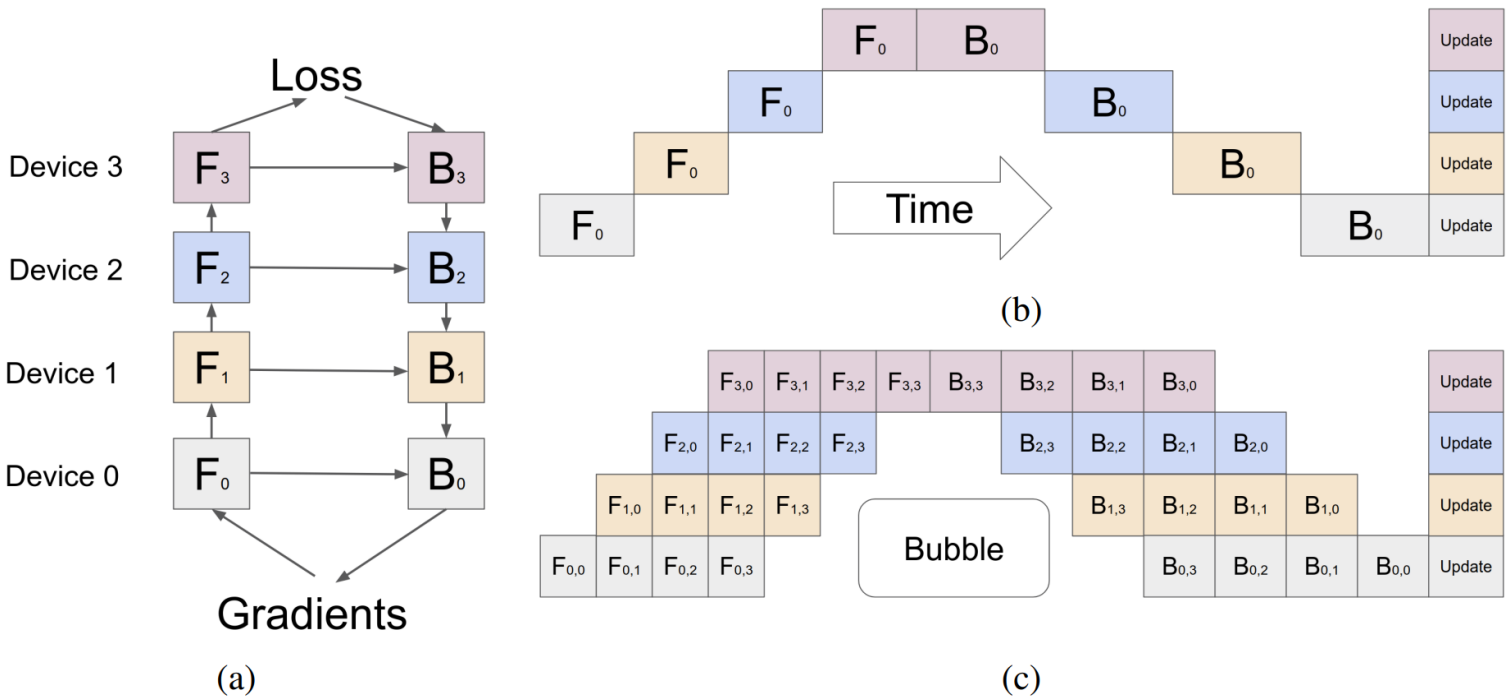


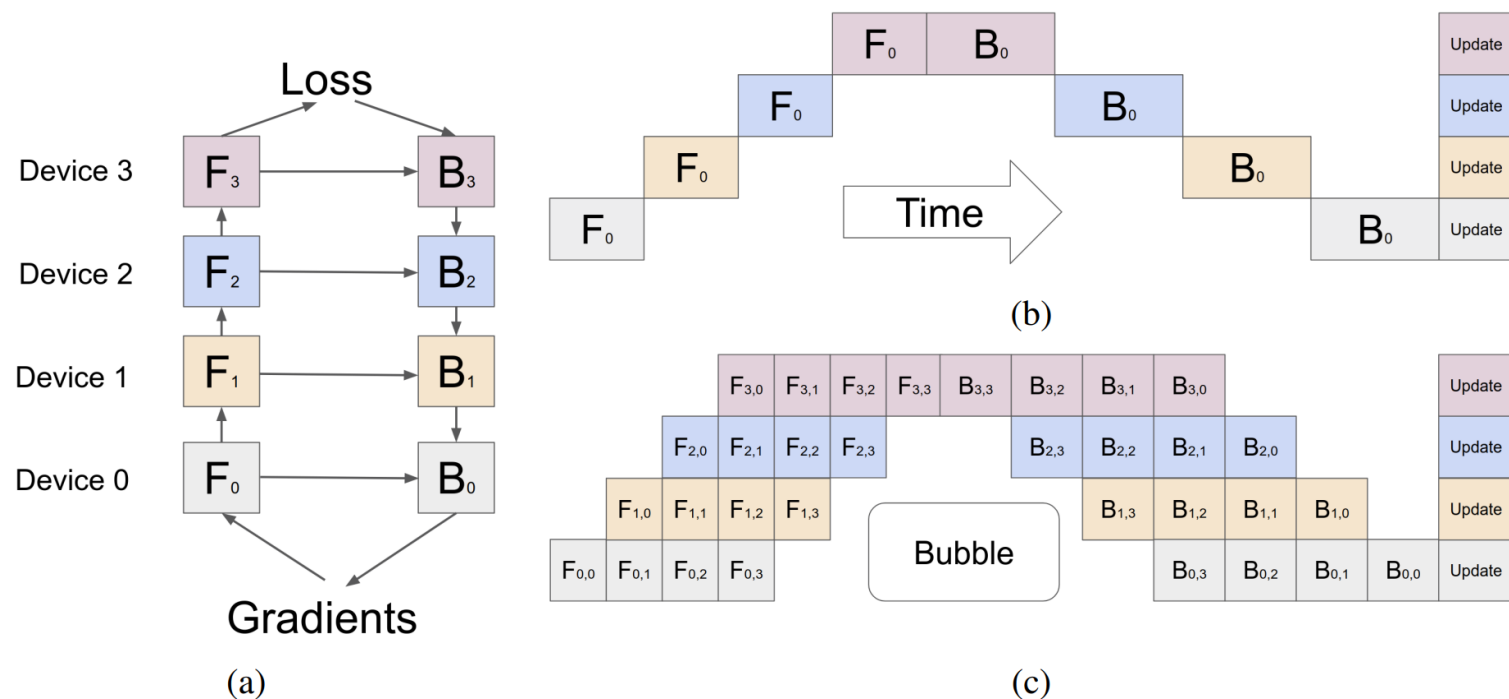
Figure 7: Illustration of Pipeline Parallelism [7]

Communication:

- Activations between nearby devices in forward pass
- Partial gradients between nearby devices in backward
- Full gradients from Device 0 to All others

Parallel Training: Pipeline Parallelism

Split network by layers, aligning devices by layer order to a pipeline, and pass data through devices [7]



Communication:

- Activations between nearby devices in forward pass
- Partial gradients between nearby devices in backward
- Full gradients from Device 0 to All others

Figure 7: Illustration of Pipeline Parallelism [7]

Pros: Conceptually simple and not coupled with network architectures. All networks have multiple layers.

Cons: Waste of compute in the Bubble. Bubble gets bigger with more devices and bigger batches.

Parallel Training: Tensor Parallelism

Split the parameter tensors of target layers into different devices

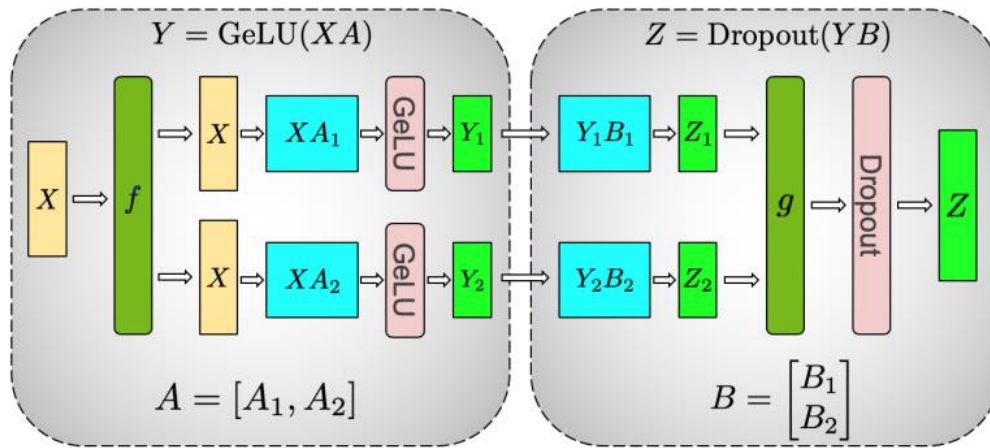


Figure 8: Tensor Parallelism of MLP blocks and Self-attention Blocks [8]

Parallel Training: Tensor Parallelism

Split the parameter tensors of target layers into different devices

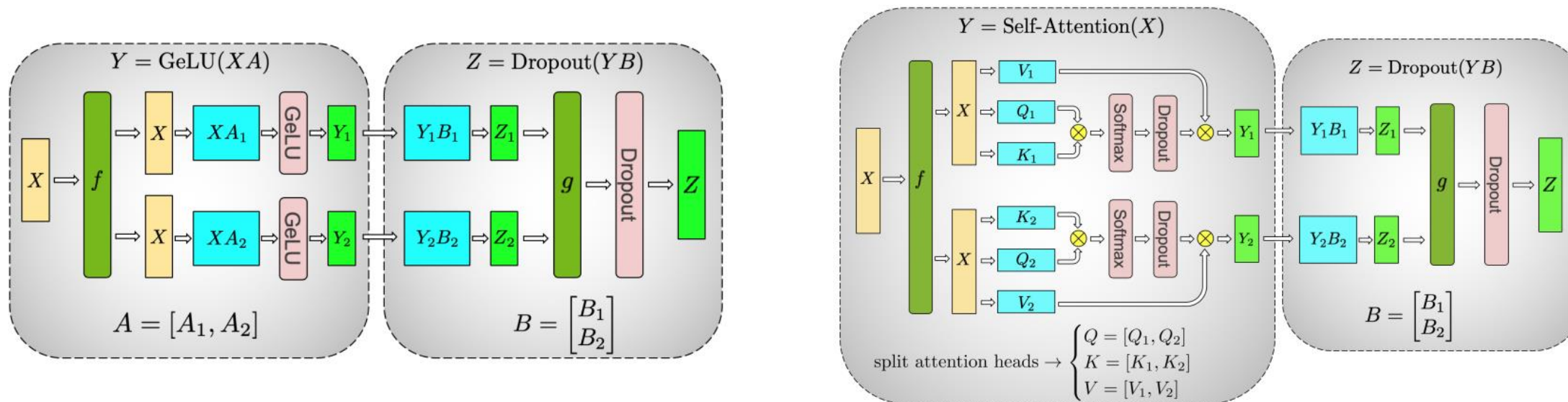


Figure 8: Tensor Parallelism of MLP blocks and Self-attention Blocks [8]

Parallel Training: Tensor Parallelism

Split the parameter tensors of target layers into different devices

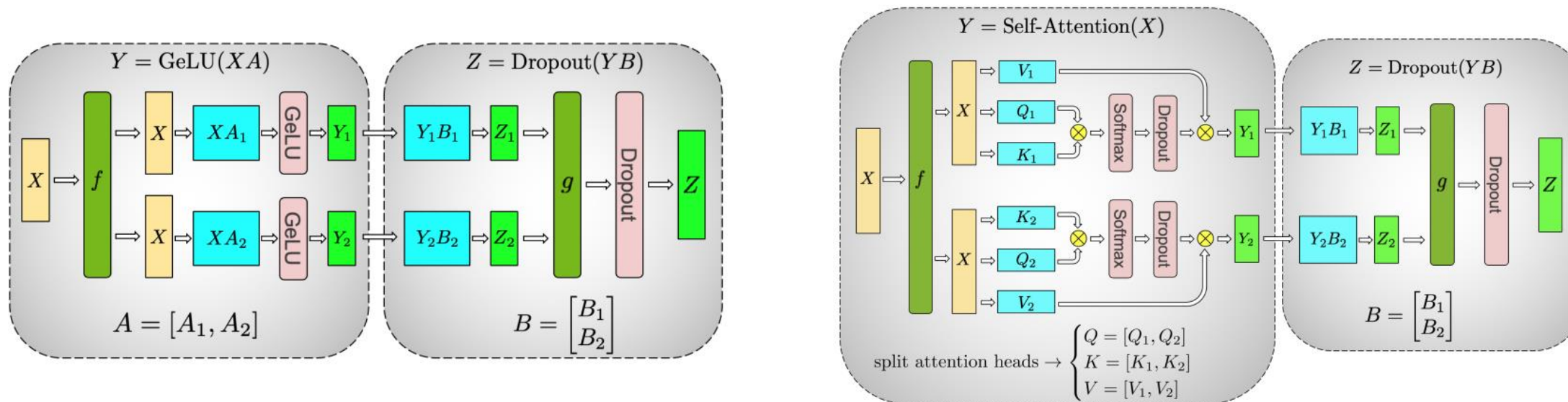


Figure 8: Tensor Parallelism of MLP blocks and Self-attention Blocks [8]

Pros: No bubble

Cons: Different blocks are better split differently, lots of customizations

Parallel Training: Tensor Parallelism

Split the parameter tensors of target layers into different devices

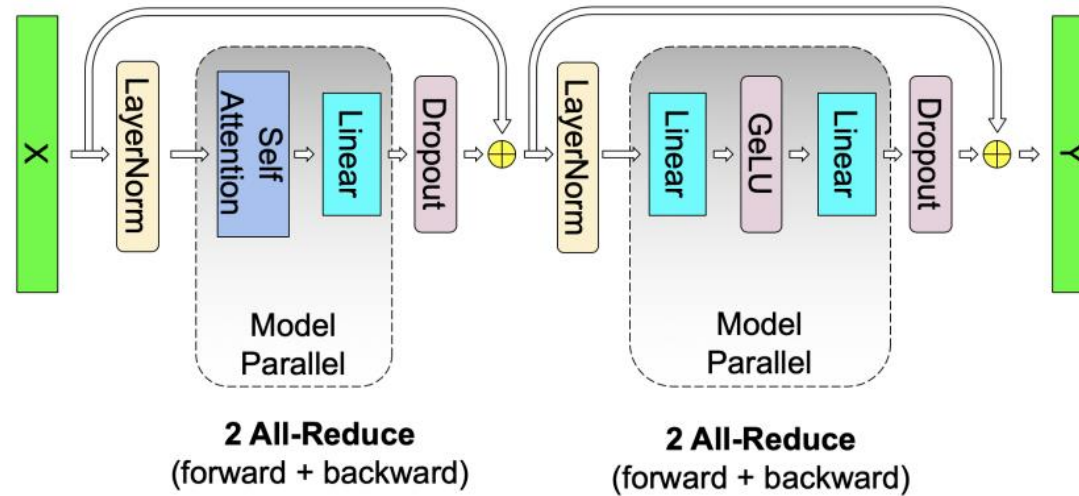


Figure 9: Communication of Tensor Parallelism for a Transformer Layer [8]

Communication:

- All-gather of partial activations and gradients for each split tensor

Parallel Training: Combining Different Parallelism

Often data parallelism and model parallelism are used together.

- No need not to use data parallelism

Pipeline Parallelism and Tensor Parallelism can also be used together.

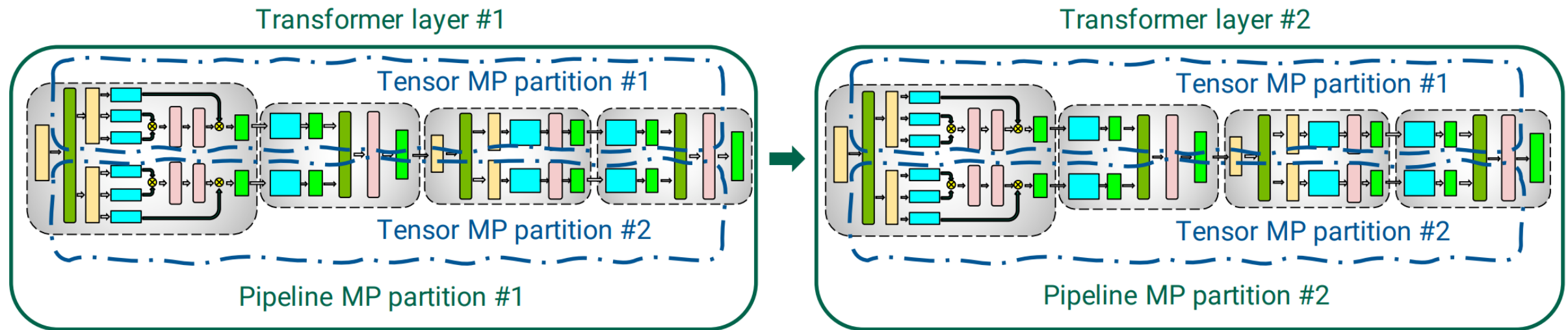


Figure 10: Combination of Tensor Parallelism and Pipeline Parallelism [9]

Outline

Optimization

- Optimization Basics
- Numerical Types

Parallel Training

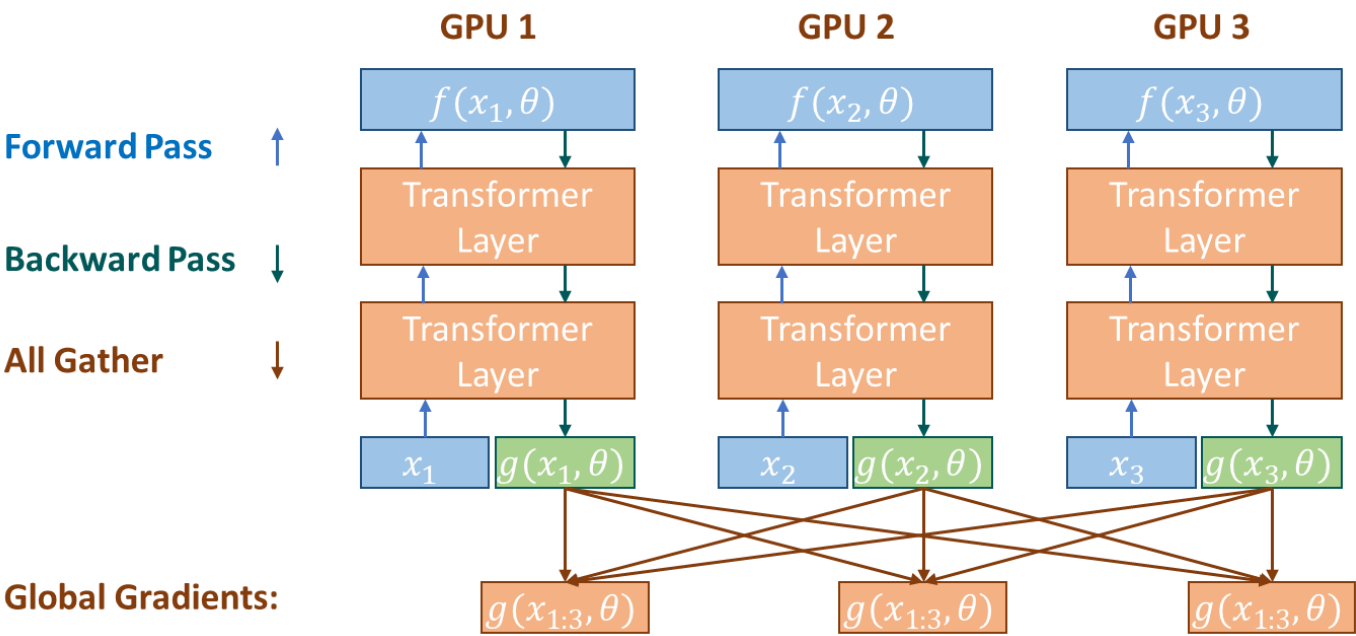
- Data Parallelism
- Pipeline Parallelism
- Tensor Parallelism
- Combination of Combination
- **ZeRO Optimizer**

ZeRO: Redundancy in Data Parallelism

Majority of GPU memory consumption is on the optimization side: gradients and optimizer momentums

	Cost of 10B Model	Function to parameter count (Ψ)
Parameter Bytes	20GB	2Ψ
Gradient Bytes	20GB	2Ψ
Optimizer State: 1st Order Momentum	20GB	2Ψ
Optimizer State: 2nd Order Momentum	20GB	2Ψ
Total Per Model Instance	80GB	8Ψ

Table 1: Memory Consumption of Training Solely with BF16 (Ideal case) of a model sized Ψ



Observation:

- In data parallelism, each device only has access to local gradient
- All gather operation required on all gradients anyway

ZeRO: Reduce Memory Redundancy

ZeRO Optimizer: Split GPU memory consumption into multiple GPUs during data parallelism

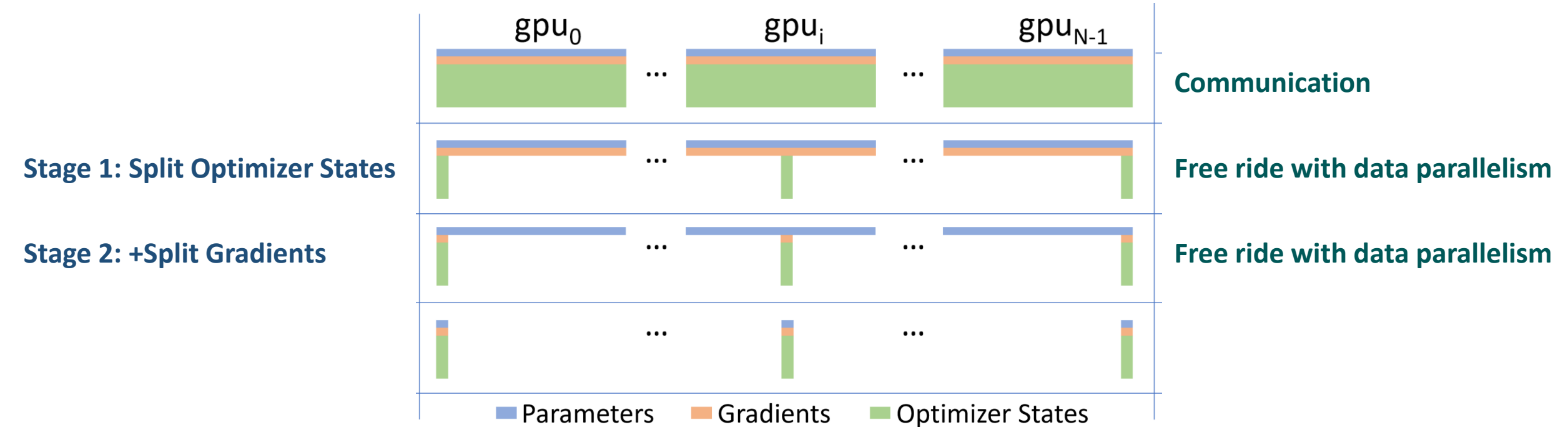


Figure 11: ZeRO Optimizer Stages [10]

ZeRO: Reduce Memory Redundancy

ZeRO Optimizer: Split GPU memory consumption into multiple GPUs during data parallelism

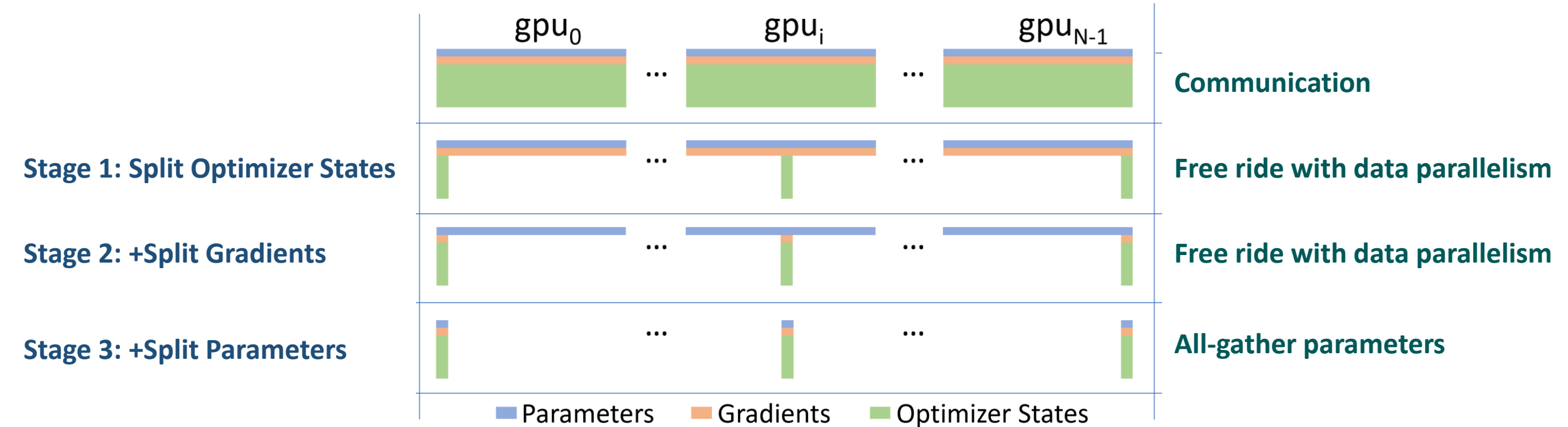


Figure 11: ZeRO Optimizer Stages [10]

ZeRO: Reduce Memory Redundancy

ZeRO Optimizer: Split GPU memory consumption into multiple GPUs during data parallelism

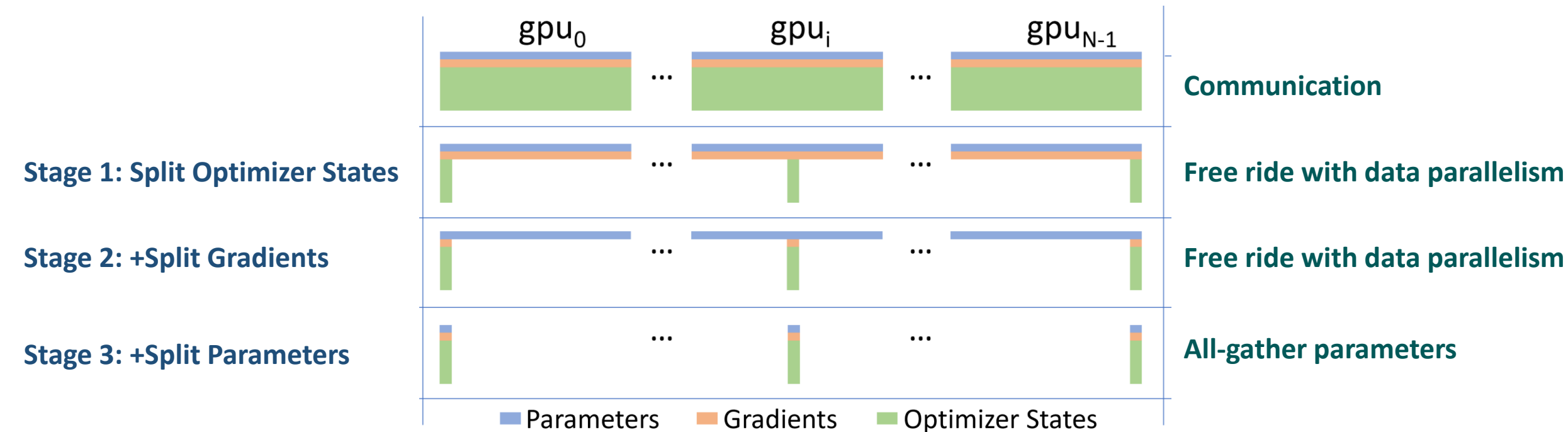


Figure 11: ZeRO Optimizer Stages [10]

Pros: Stage 1 and 2 free ride with data parallelism with huge GPU memory savings

Notes: Stage 3 is a variant of tensor parallelism, but passing parameters instead of activations and gradients

Cons: Open-source support not as good

Other Notable Literatures in Scaling Up

Different configurations of layer normalization: pre layernorm, post layernorm and their combination

- Xiong et al. “On Layer Normalization in the Transformer Architecture”. ICML 2020
- Zhang and Sennrich. “Root Mean Square Layer Normalization”. NeurIPS 2019

Position embeddings for longer contexts and expressiveness

- Su et al. “Roformer: Enhanced transformer with rotary position embedding.” arXiv 2021

Stability improvement from adaptive initialization

- Liu et al. “Understanding the Difficulty of Training Transformers”. EMNLP 2020