Large-scale Distributed Training for LLMs

Ayush Maheshwari

Sr. Solutions Architect, NVIDIA

Interested in research collaboration !

Fill this form - https://forms.gle/Q4mEWYhCqEsg6ngU6

Sessions

1. Cluster health-check using NCCL, MLPerf, HPL **(1 hour) - Completed**
    a) Understand the hardware and its performance on multiple GPUs.
    b) Ensure that your training performance aligns with the h/w benchmarks
    c) Evaluate the cluster to ensure platform fits within your needs.
2. Large scale data curation for LLM training **(1 hour) - Completed**
    a) Deep-dive into aspects of data curation
    b) Mixed-precision training
3. Distributed and stable LLM training on a large-scale cluster **(1.5 hour) - Today**
    a) Parallelism techniques
    b) Frameworks and wrappers
    c) Recipes and best practices
4. Post-training and evaluation of pre-trained LLM **(1 hour)**
    a) Sync between training data and expected performance
    b) Algorithms and frameworks
5. Fine-tuning and deployment **(1 hour)**
    a) Dynamic and static batching, state management, inference server
    b) Best practices for optimizing model

Agenda

1. Concepts of Parallelism – Data, Tensor and Pipeline

2. How does these work together ?

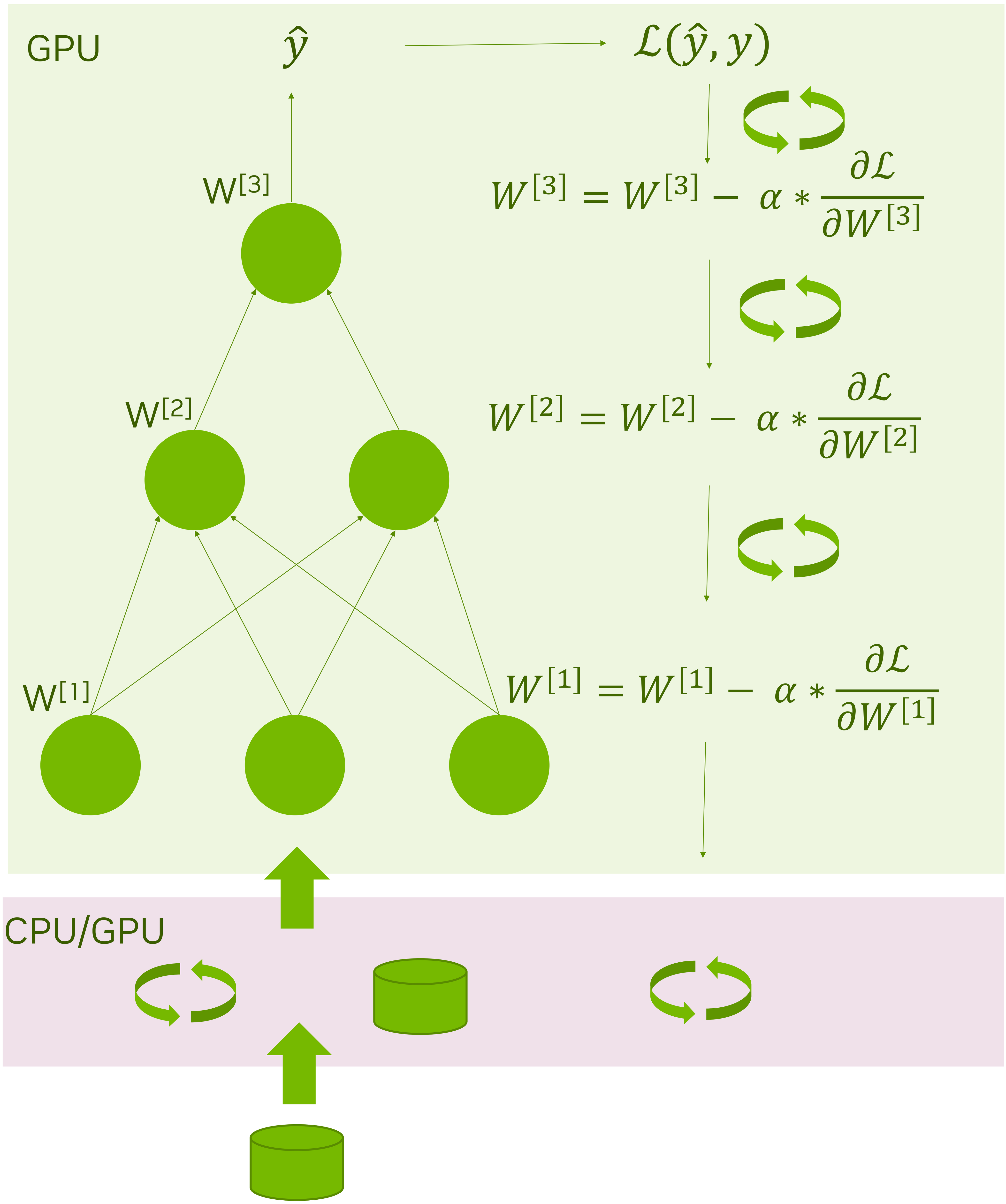
3. Mixed –precision training

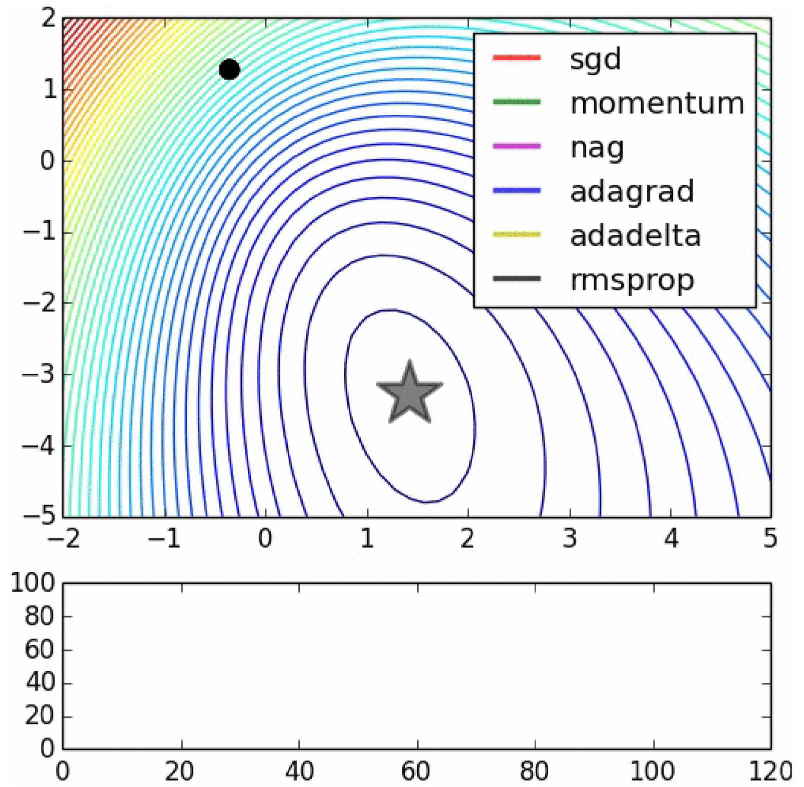
Not Covering

Transformer architecture, FP8 training

Training a Neural Network

Single GPU





1. Read the data
2. Transport the data
3. Pre-process the data
4. Queue the data
5. Transport the data
6. Calculate activations for layer one
7. Calculate activations for layer two
8. Calculate the output
9. Calculate the loss
10. Backpropagate through layer three
11. Backpropagate through layer two
12. Backpropagate through layer one
13. Execute optimization step
14. Update the weights
15. Return control

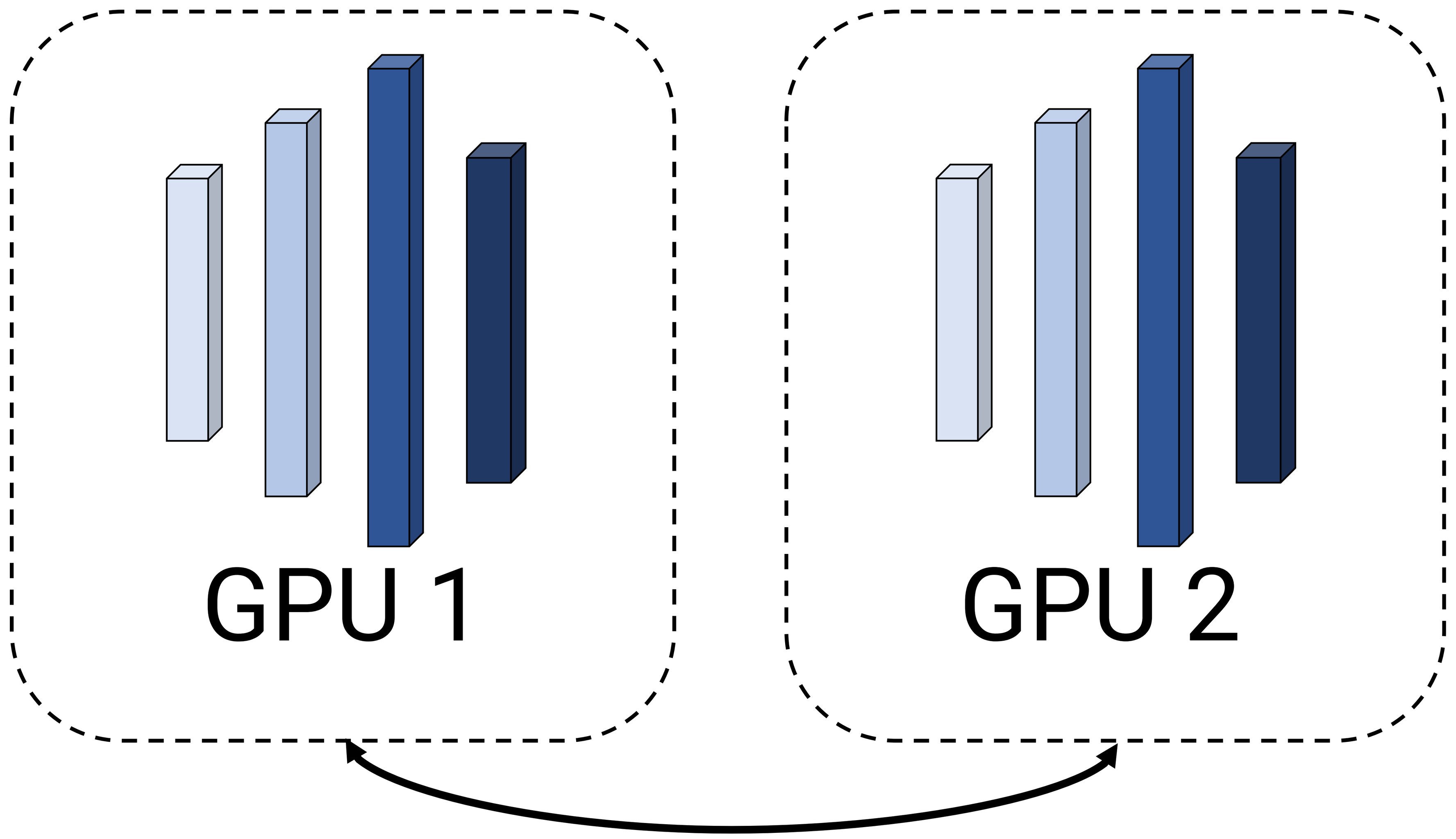
Training a Neural Network

Multiple GPUs with DDP

Distributed Training: Parallelism

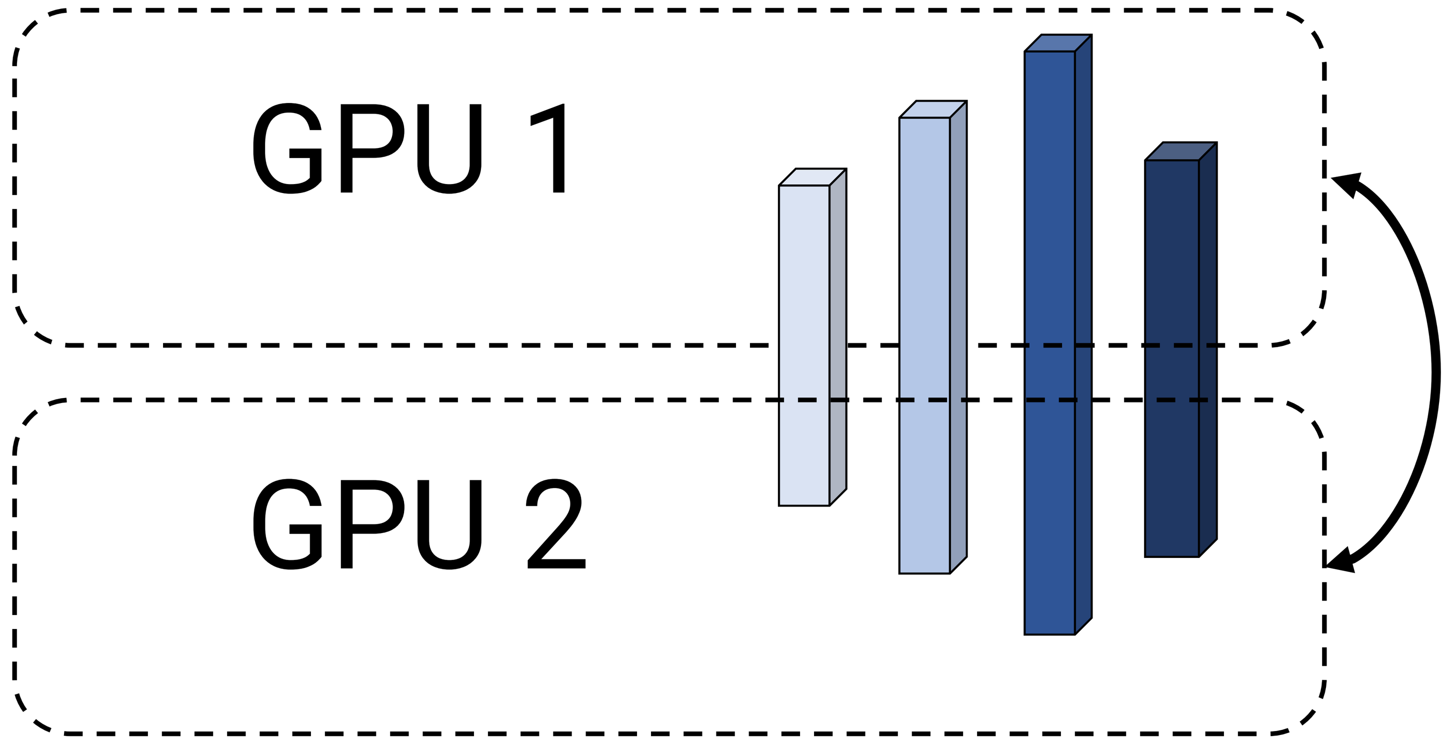
Parallelism : An overview

Data parallelism (DP)

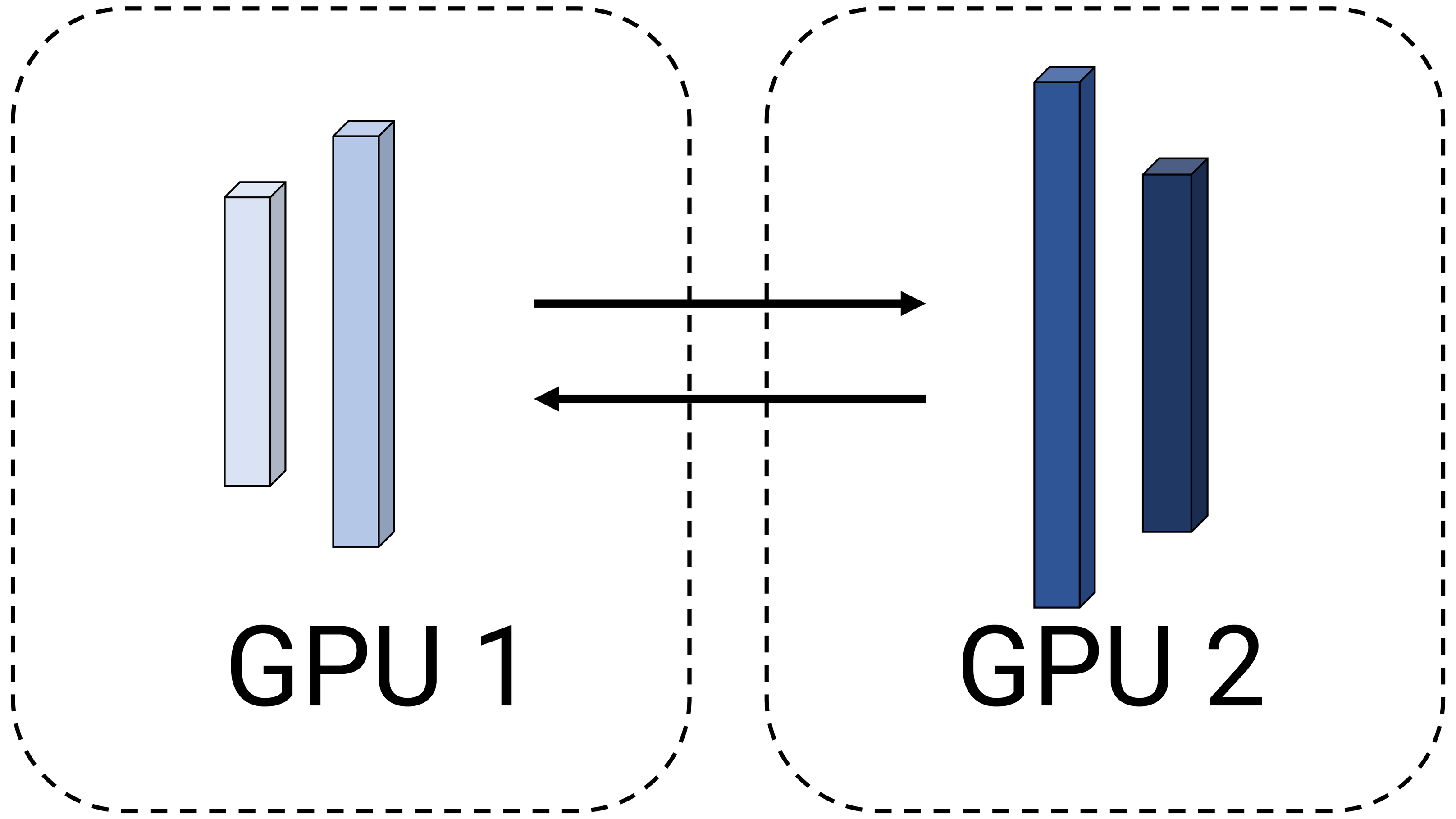


n copies of model parameters

Model parallelism (MP)



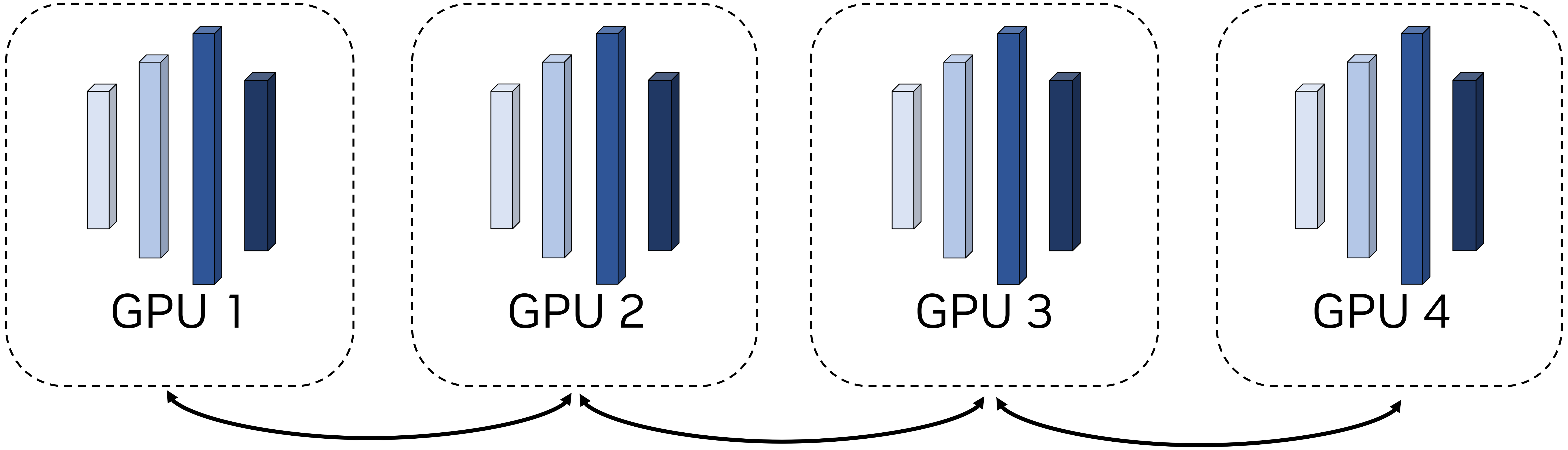
Tensor MP



Pipeline MP

Single copy of model parameters

Data parallelism



- Naïvely, model copy on each GPU
- Reductions of weight gradients at the end of every iteration to coalesce updates across replicas
- Our data parallelism implementation involves a simple DDP wrapper, with largely the same interface as PyTorch's DDP

Data Parallelism (Distributed Optimizer)

High Level Abstraction

Do {

Forward Path (activations) – calculate error

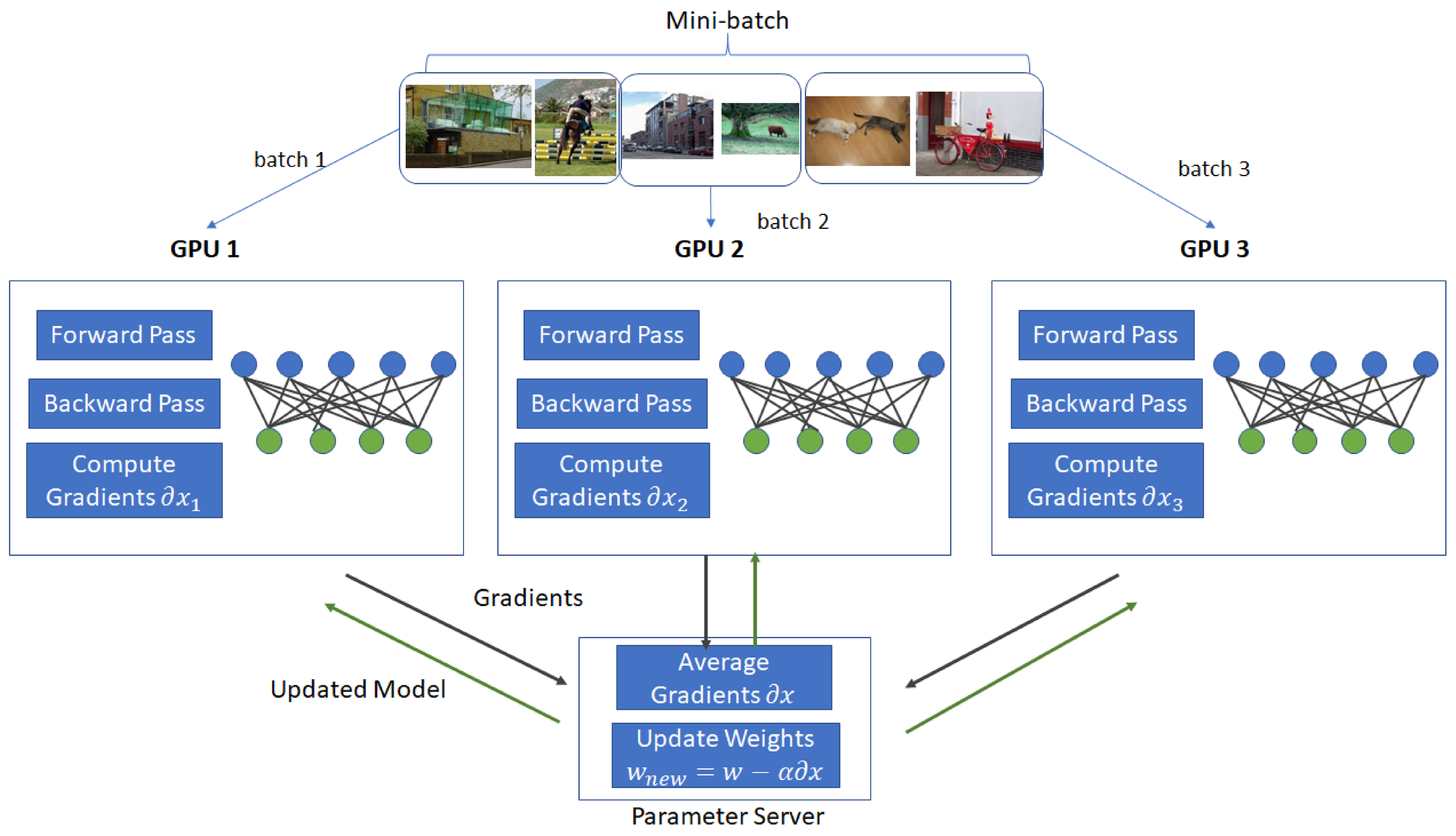
Backward Path – calculate gradients

Reduce Scatter gradients – each source gets a different part of the results

Update network weights (a.k.a. optimizer) **available local part of the gradient weights**

All Gather weights calculated in each GPU

} While error is above threshold / not decreasing anymore



Data Parallelism (Distributed Optimizer)

High Level Abstraction

Do {

Forward Path (activations) – calculate error

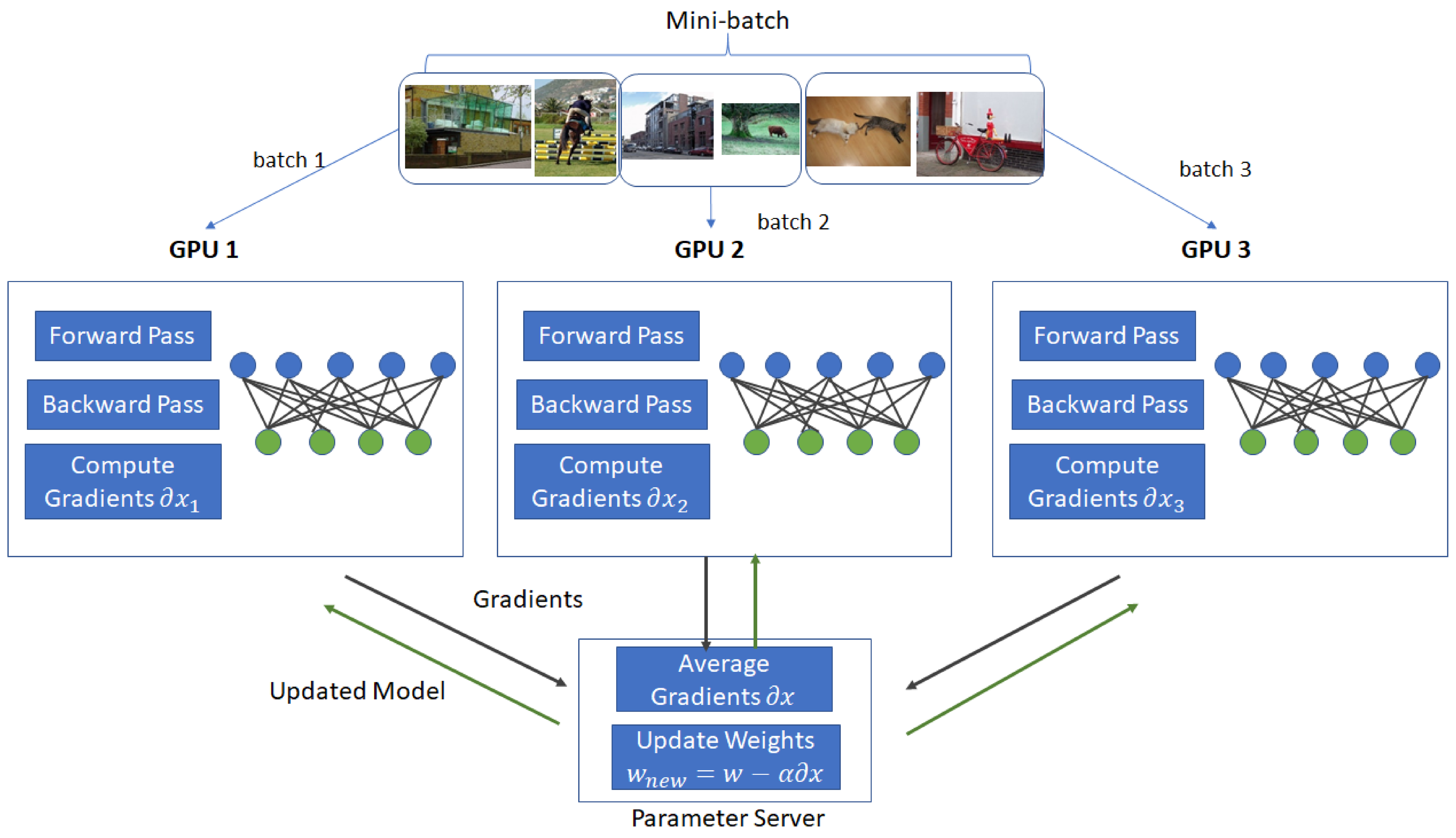
Backward Path – calculate gradients

Reduce Scatter gradients – each source gets a different part of the results

Update network weights (a.k.a. optimizer) **available local part of the gradient weights**

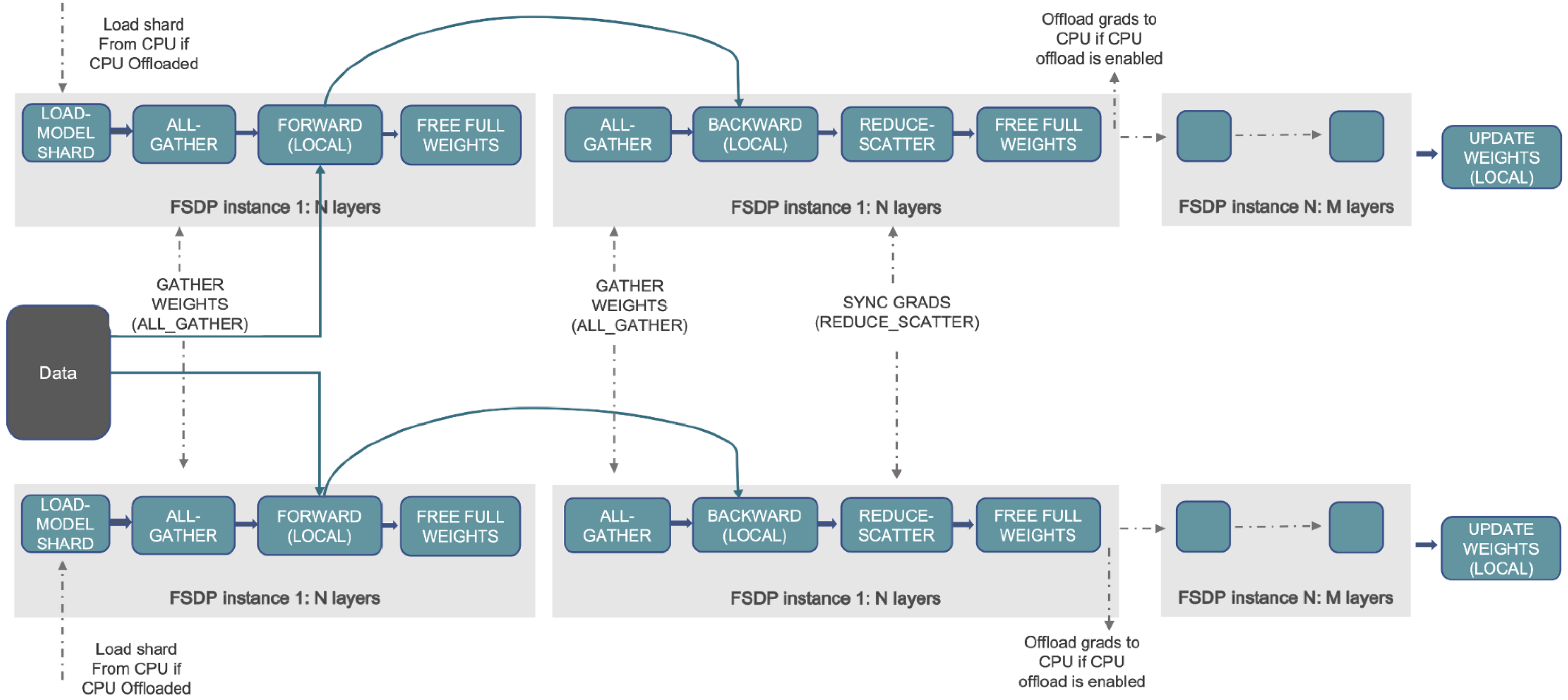
All Gather weights calculated in each GPU

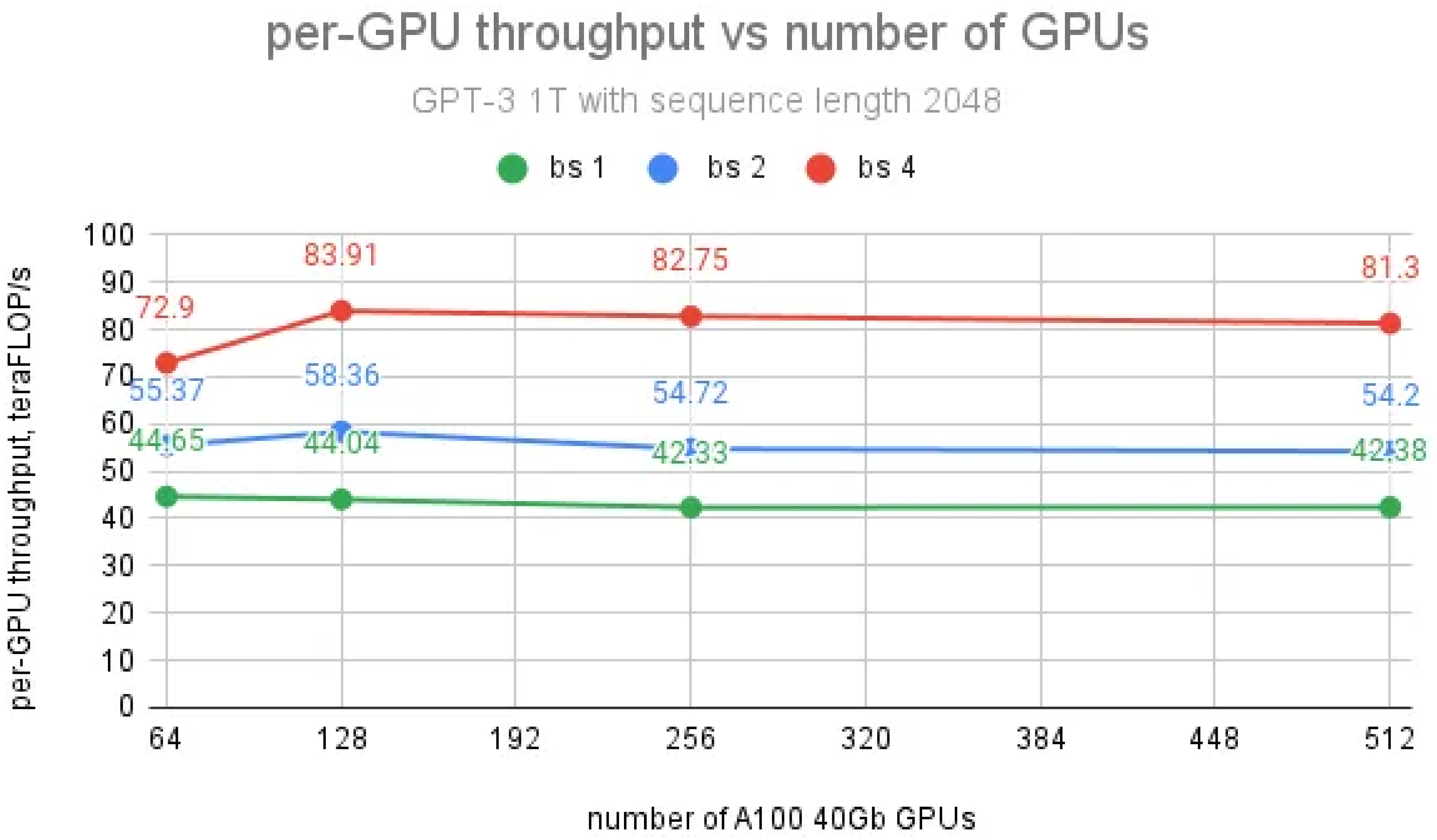
} While error is above threshold / not decreasing anymore



Distributed Data Parallel - DDP

PyTorch: Streamline API for Fully Sharded Data Parallel (FSDP)





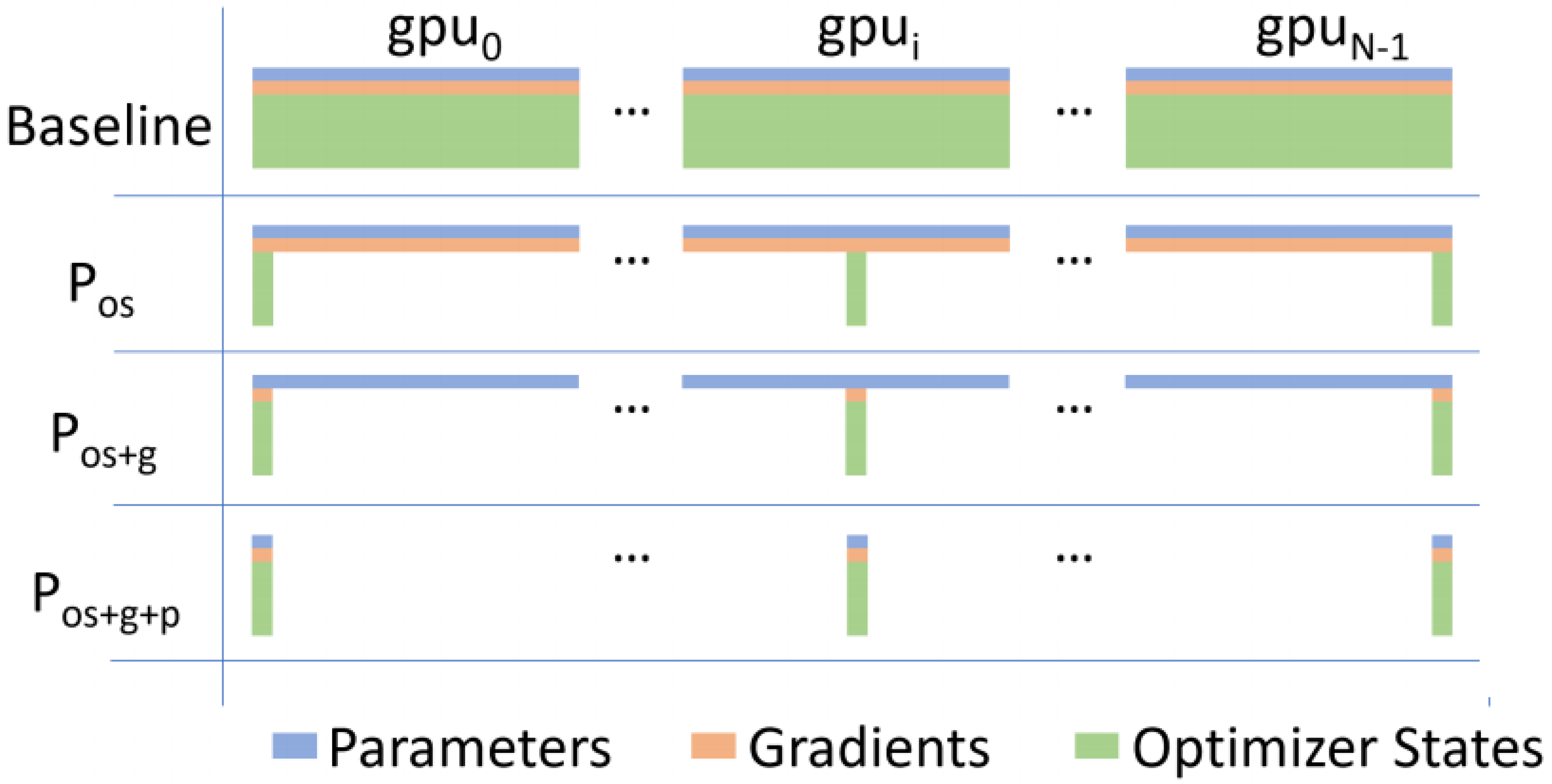
From 128 GPUs, further increase of the number of GPUs doesn't affect the per-GPU throughput significantly.

https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Sharded Data Parallelism

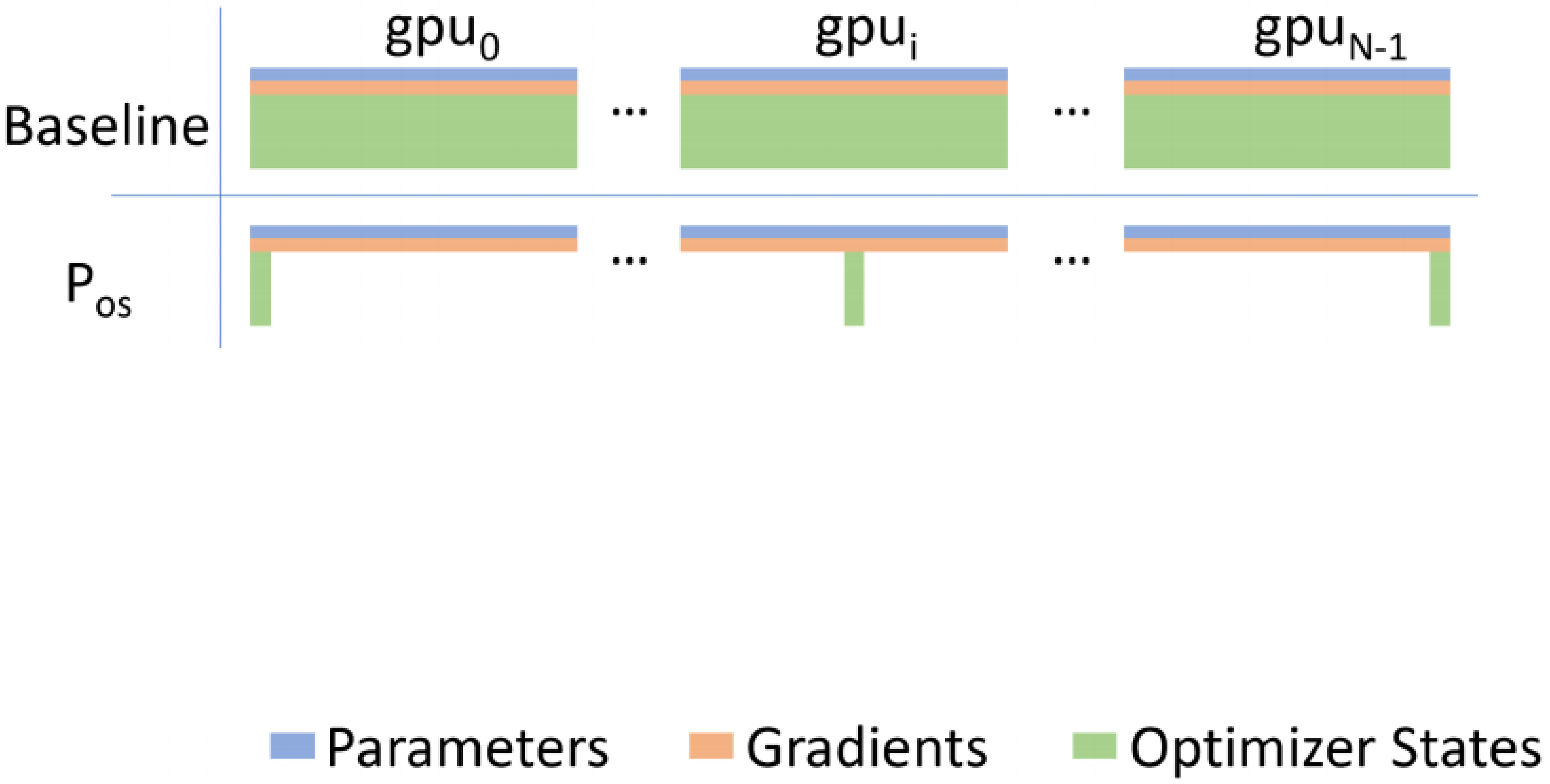
ZeRO: Zero Redundancy Optimizer

- ZeRO removes the redundancy across data parallel process
- Partitioning optimizer states, gradients and parameters (3 stages) for a progressive memory savings and Communication Volume



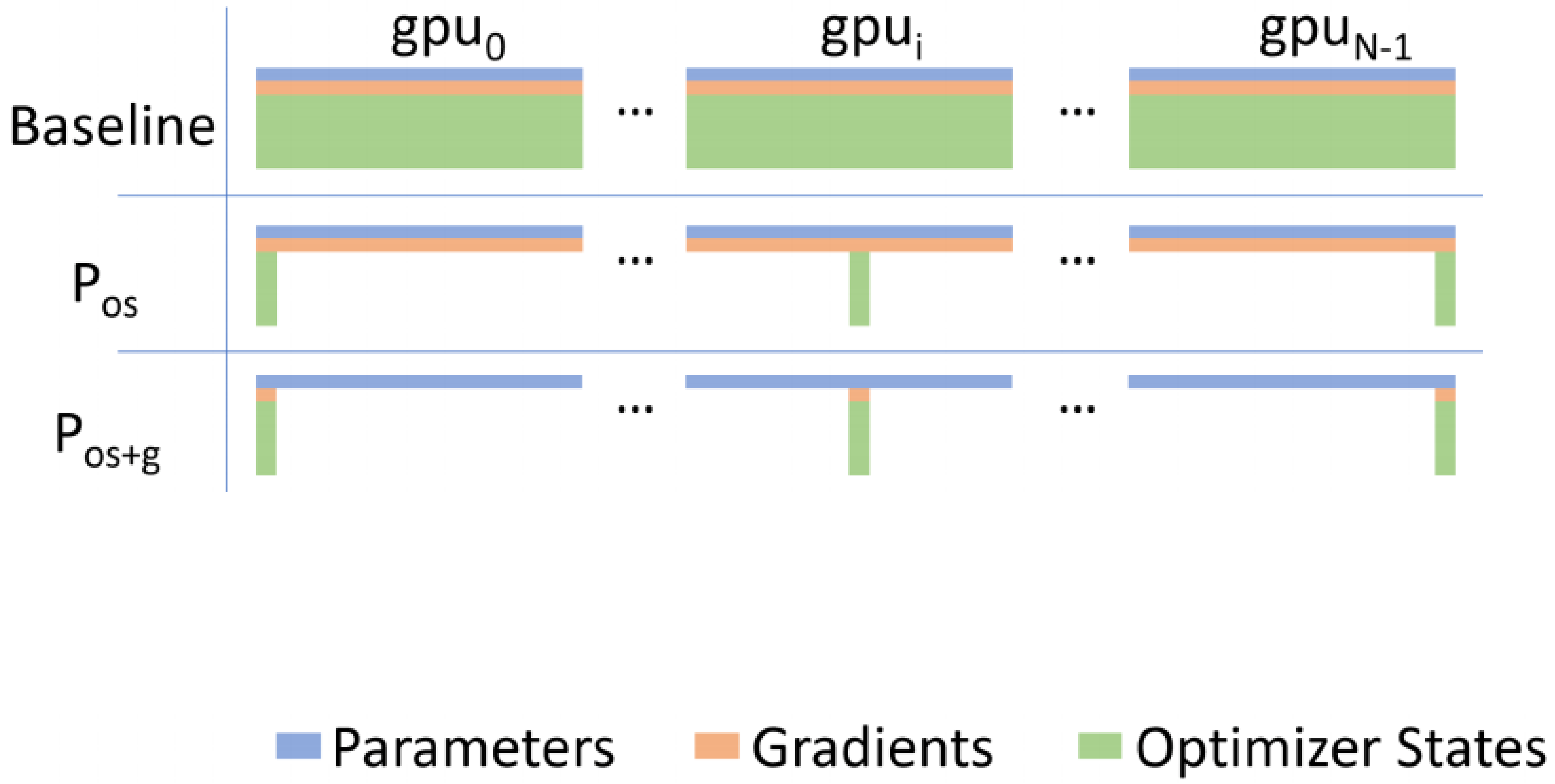
Sharded Data Parallelism

ZeRO: Stage 1



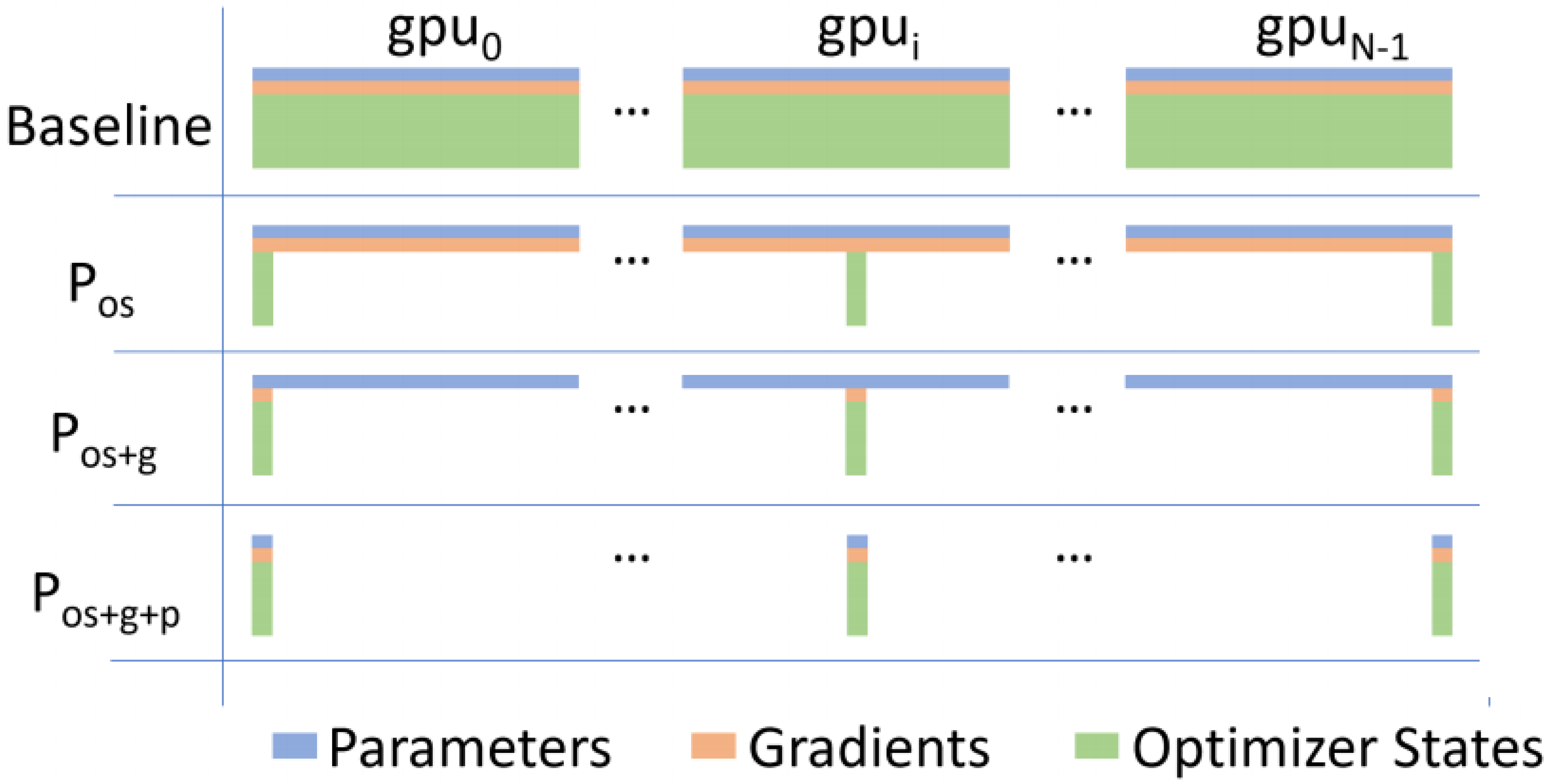
Sharded Data Parallelism

ZeRO: Stage 2



Sharded Data Parallelism

ZeRO: Stage 3



GPU Memory occupation

Let's review what is in your GPU memory

- **Model Weights**
  - 4 bytes * number of parameters for fp32 training
  - 6 bytes * number of parameters for mixed precision training (maintains a model in fp32 and one in fp16 in memory)
- **Optimizer States**
  - 8 bytes * number of parameters for normal AdamW (maintains 2 states)
  - 2 bytes * number of parameters for 8-bit AdamW optimizers like bitsandbytes
  - 4 bytes * number of parameters for optimizers like SGD with momentum (maintains only 1 state)
- **Gradients**
  - 4 bytes * number of parameters for either fp32 or mixed precision training (gradients are always kept in fp32)
- **Forward Activations**
  - size depends on many factors, the key ones being sequence length, hidden size and batch size

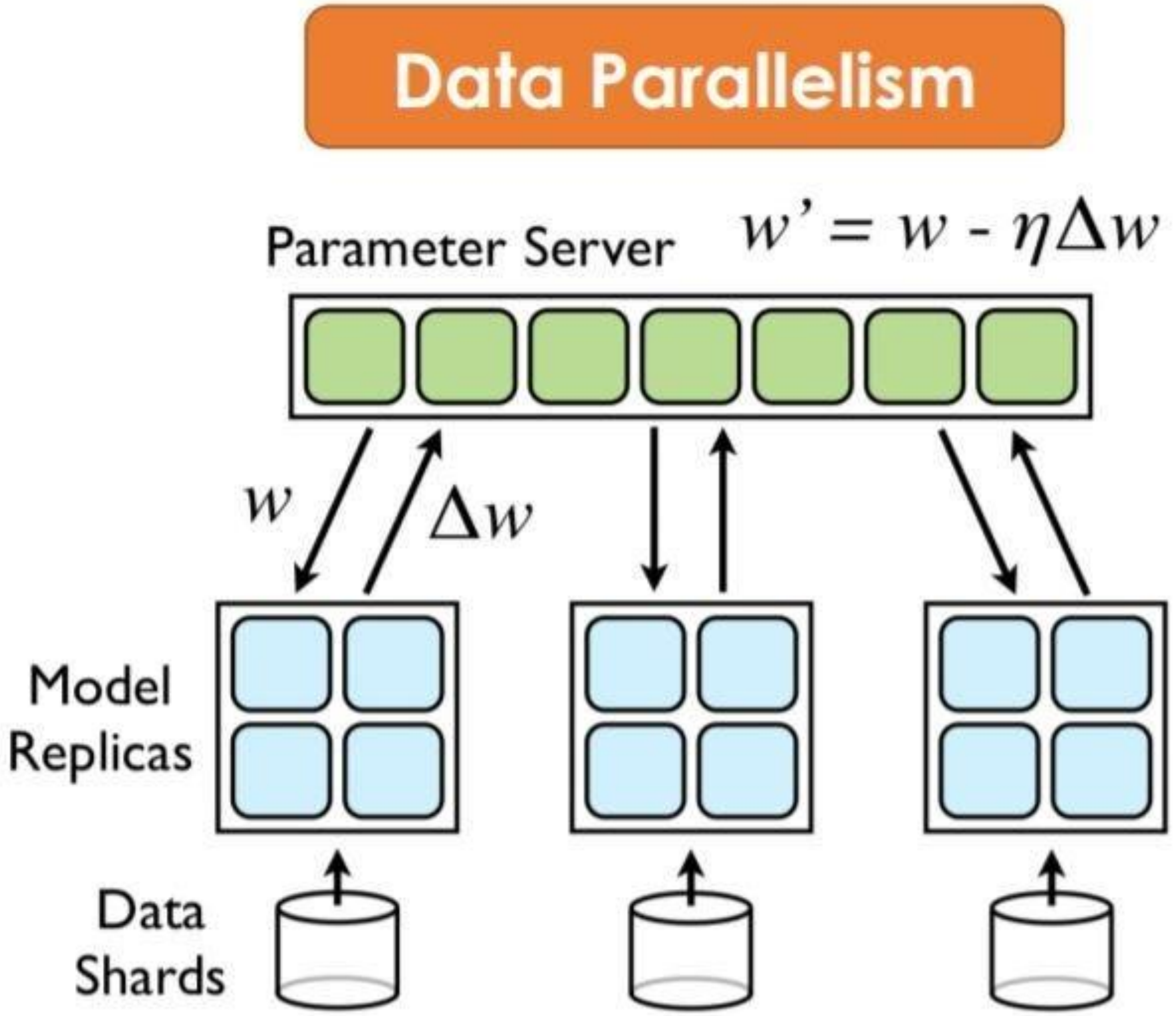
Distributed optimizer to reduce memory

$$\text{Number of bytes of state per parameter} = \underbrace{2 + 4}_{\substack{\textbf{bf16}\text{ params} \\ \textbf{fp32}\text{ grads}}} + \underbrace{4 + 4 + 4}_{\substack{\textbf{fp32}\text{ copy of params} \\ \textbf{fp32}\text{ Adam states}}}$$

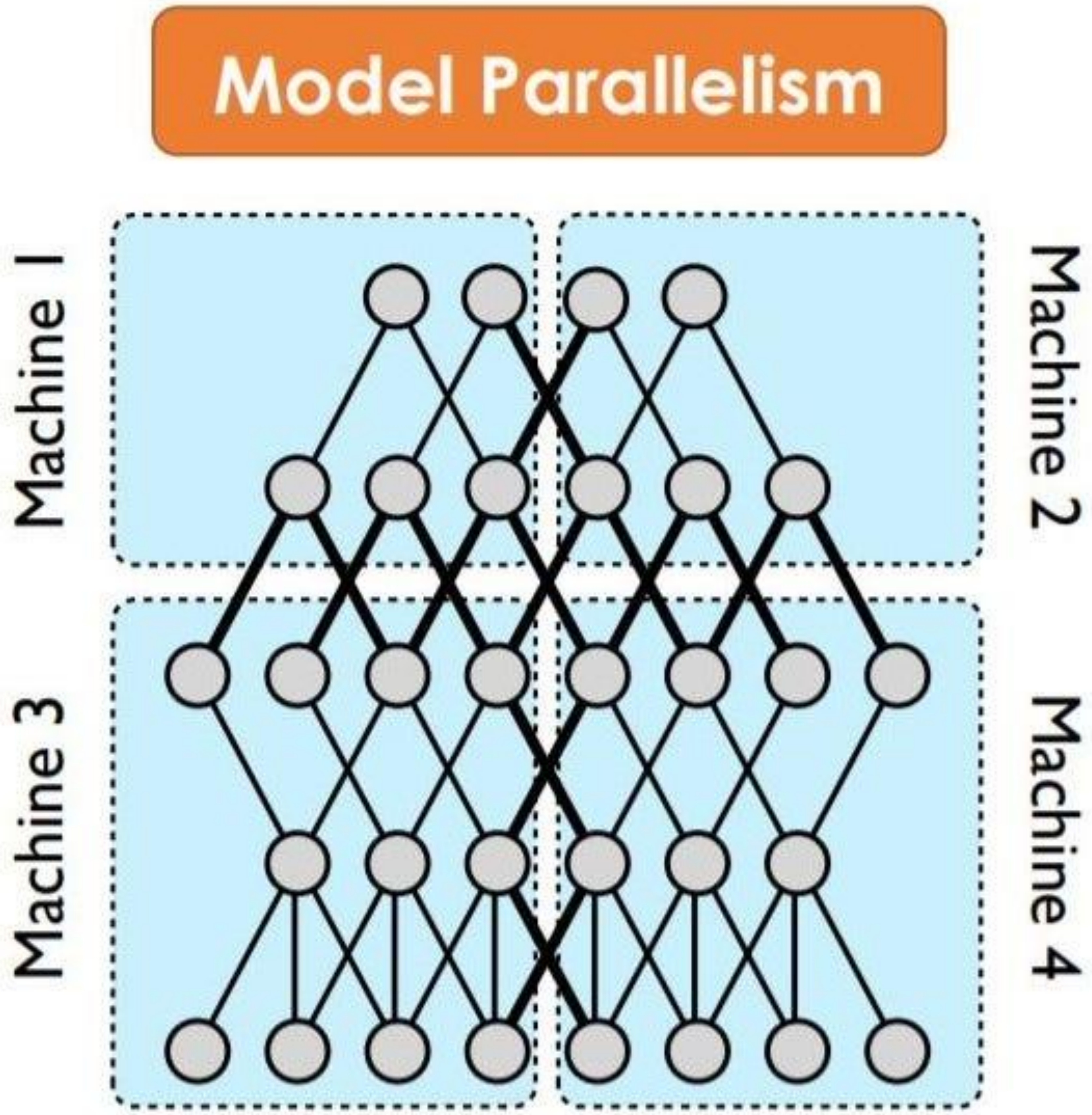
$$\text{Number of bytes of state for Nemotron-4 340B model} = 18 \cdot 340\text{B} = \mathbf{6120\ GB}$$

Redundant optimizer state over DP replicas can be partitioned
fp32 gradient all-reduces →
fp32 gradient reduce-scatters + **bf16** param all-gathers

Data Parallelism / Model Parallelism



Data is too large, accelerated by processing in parallel

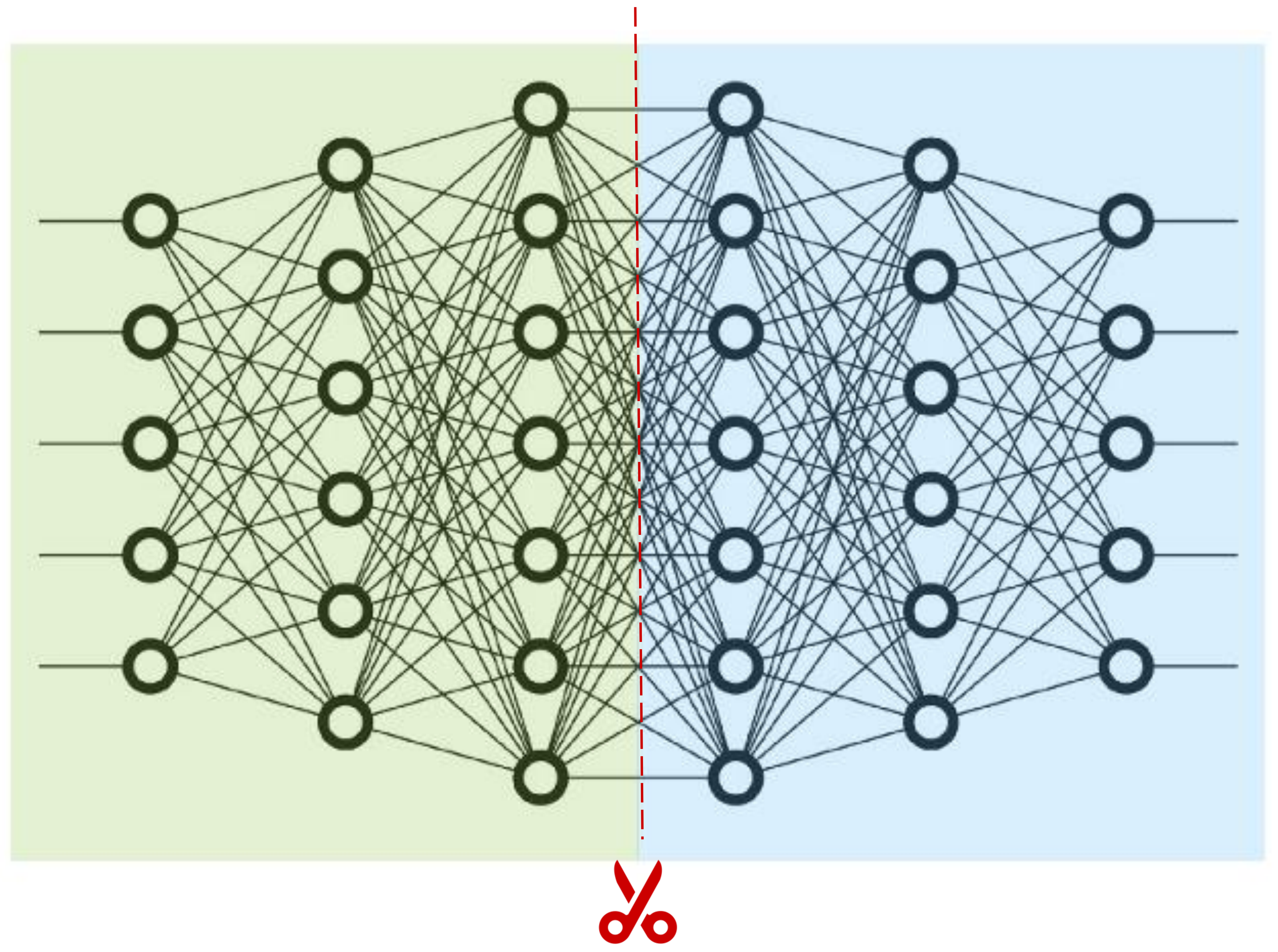


Model is too large, cannot fit in a single device / machine

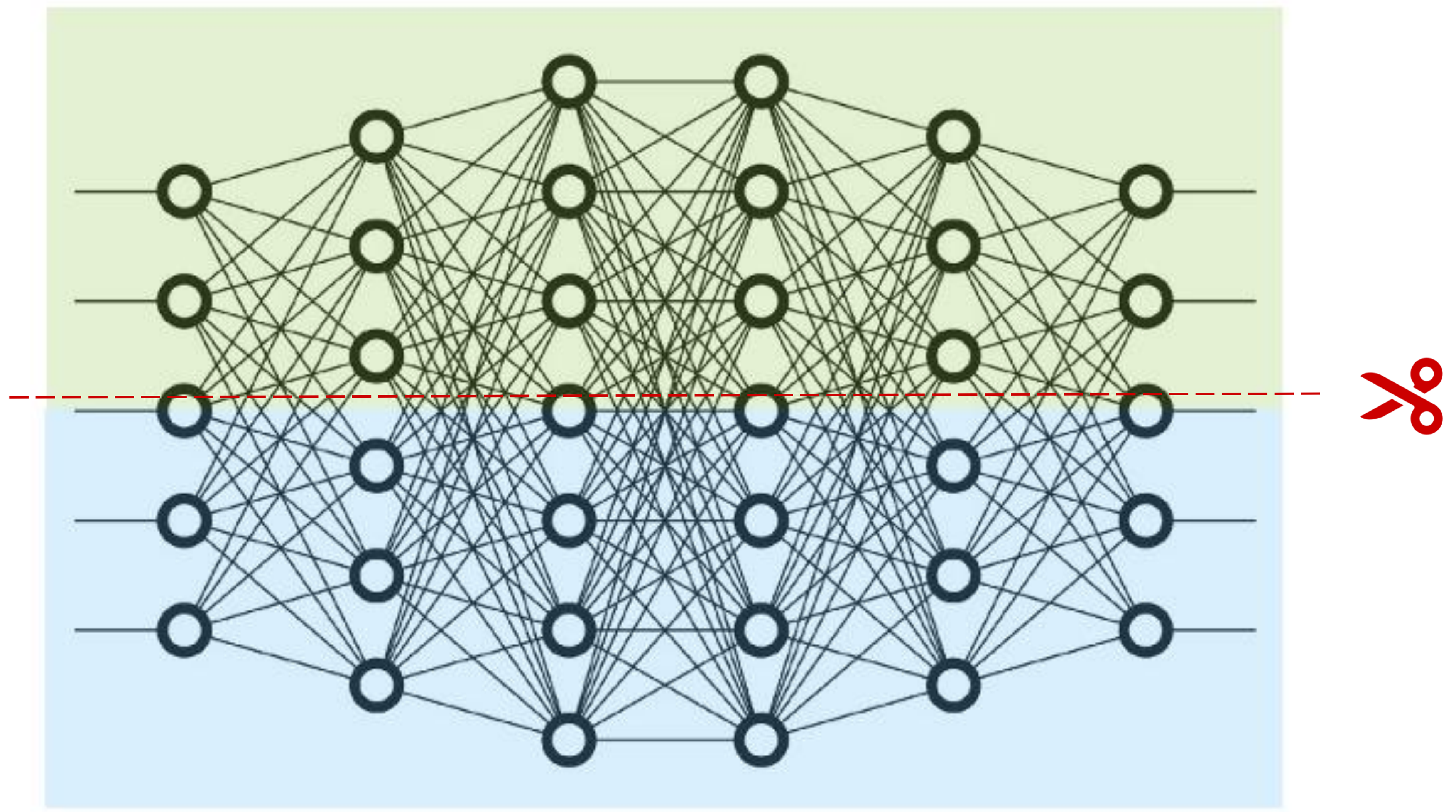
TECHNOLOGIES THAT ENABLE SCALING LARGE MODELS

Complementary Types of Parallelism

- Pipeline (Inter-Layer) Parallelism
  - Split contiguous sets of layers across multiple GPUs
  - Layers 0,1,2 and layers 3,4,5 are on different GPUs
  - *Maximizes GPU utilization in single-node*



- Tensor (Intra-Layer) Parallelism
  - Split individual layers across multiple GPUs
  - Both devices compute different parts of Layers 0,1,2,3,4,5
  - *Minimizes Latency in single-node*

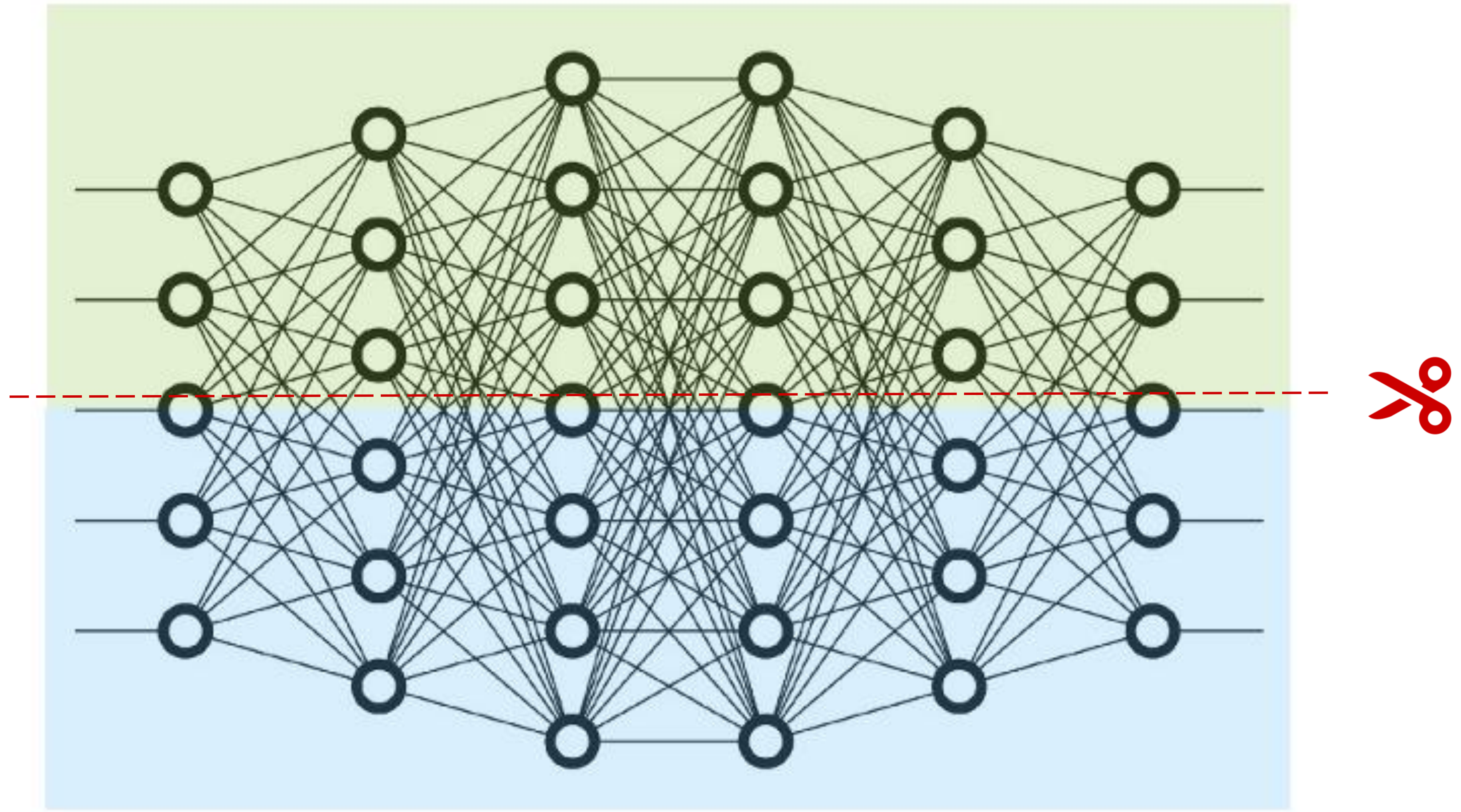


https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/

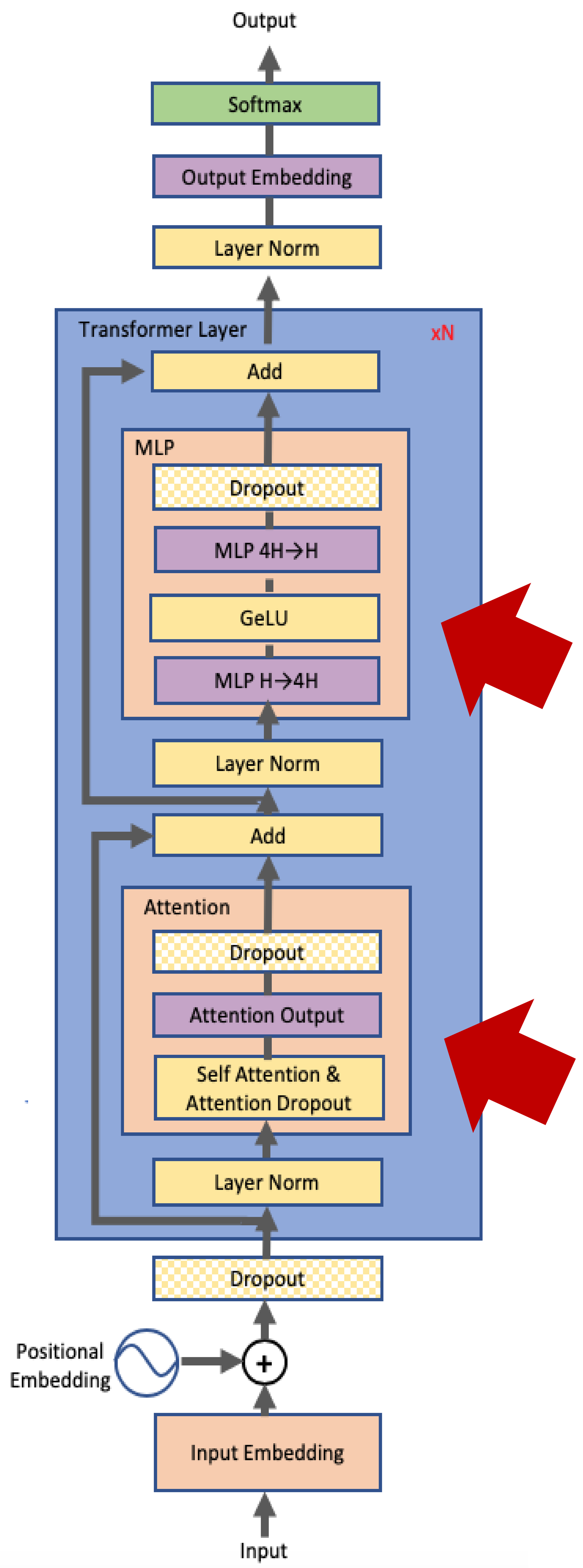
TENSOR PARALLELISM

Why?

- Relatively simple to implement
- Easier to load-balance
- Less restrictive on the batch-size (avoids bubble issue in pipelining)
  - Tensor parallelism is orthogonal to pipeline parallelism: very large models such as GPT-3 use both
- NVIDIA DGX servers with NVSwitch
  - DGX A100 has 600 GB/sec GPU-to-GPU bidirectional bandwidth
- Tensor parallelism works well for large matrices
  - Example: Transformers have large GEMMs



TRANSFORMERS CELL



MLP TENSOR PARTITIONING

Focus on the GeLU operation:

- **<u>Approach 1:</u>** Split X column-wise and A row-wise:

$$X = [X_1, X_2] \quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \Rightarrow Y = \mathrm{GeLU}(X_1 A_1 + X_2 A_2)$$

  - Before GeLU we will need a communication point

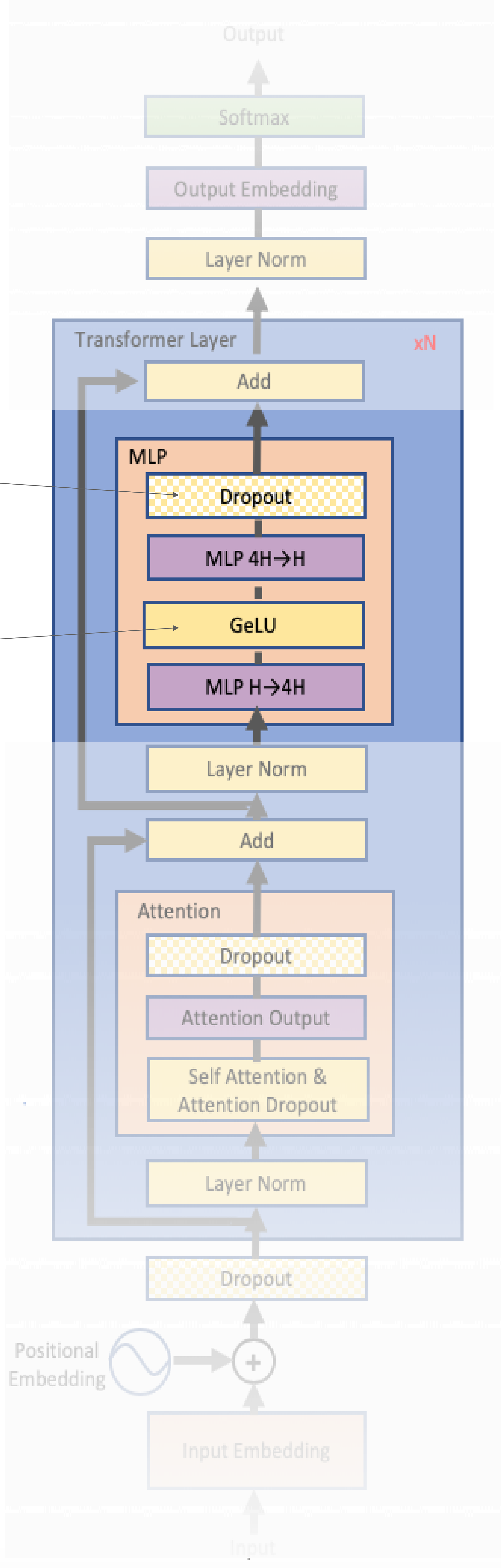
- **<u>Approach 2:</u>** Split A column-wise:

$$A = [A_1, A_2] \Rightarrow [Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$$

  - No communication is required

$$Z = \mathrm{Dropout}(YB)$$

$$Y = \mathrm{GeLU}(XA)$$



MLP TENSOR PARTITIONING

Focus on the GeLU operation:

- **Approach 1:** Split X column-wise and A row-wise:

$$X = [X_1, X_2] \quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \Rightarrow Y = \mathrm{GeLU}(X_1 A_1 + X_2 A_2)$$

  - Before GeLU we will need a communication point

- **Approach 2:** Split A column-wise:

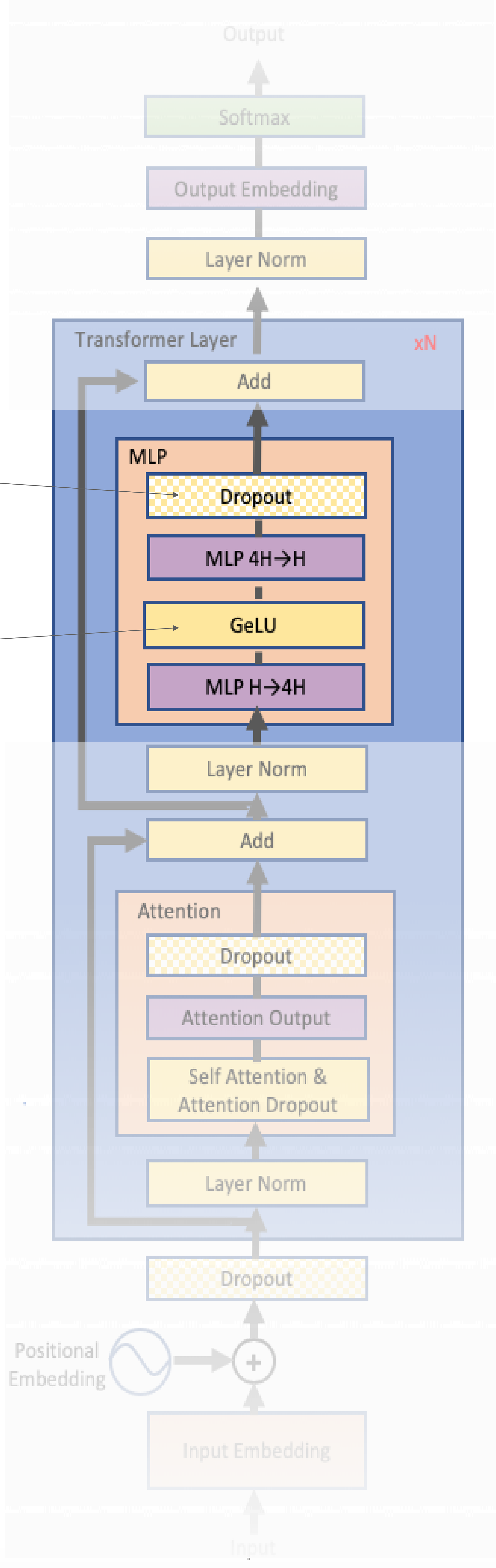
$$A = [A_1, A_2] \Rightarrow [Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$$

  - No communication is required

Chosen approach

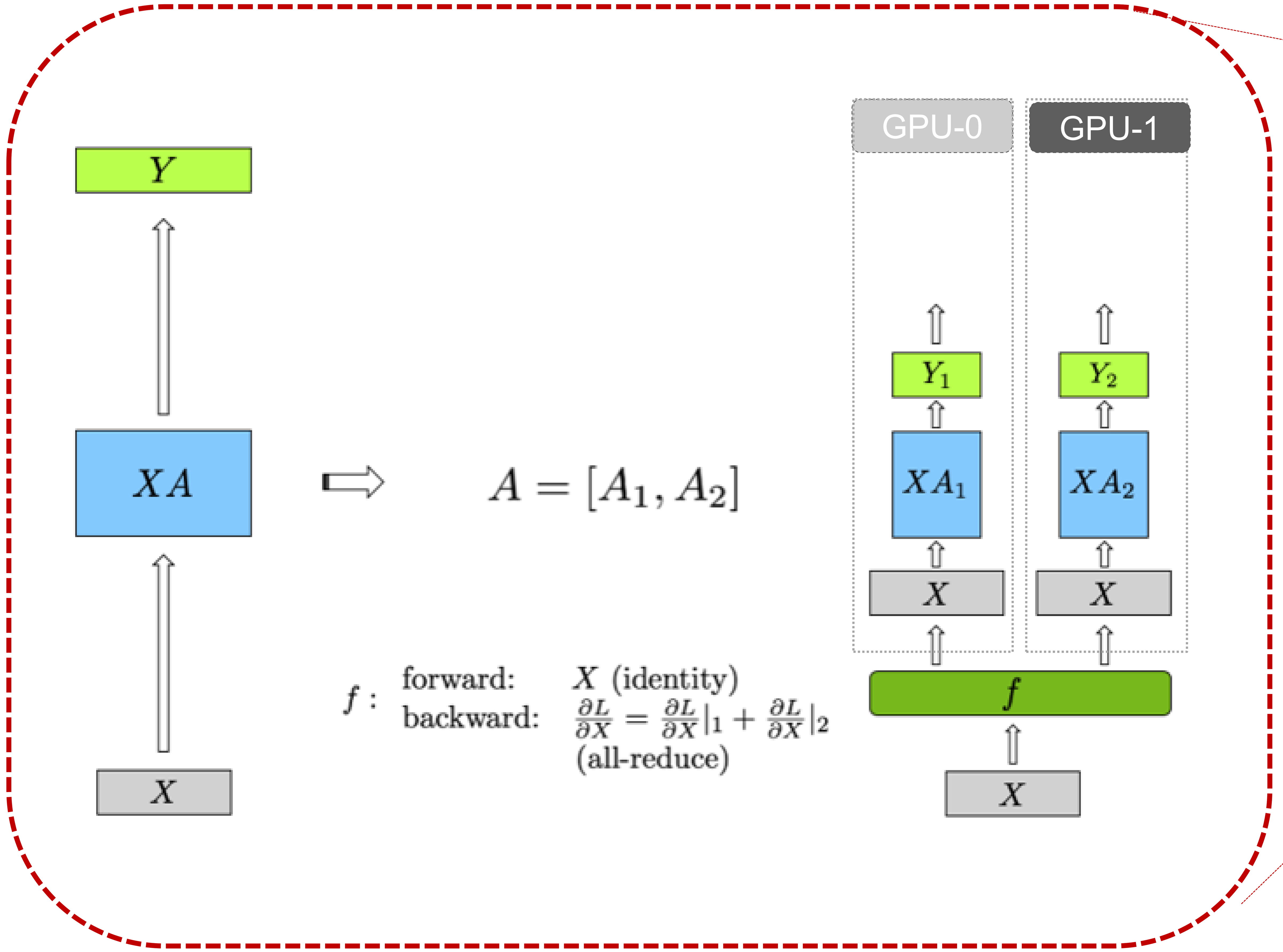
$$Z = \mathrm{Dropout}(YB)$$

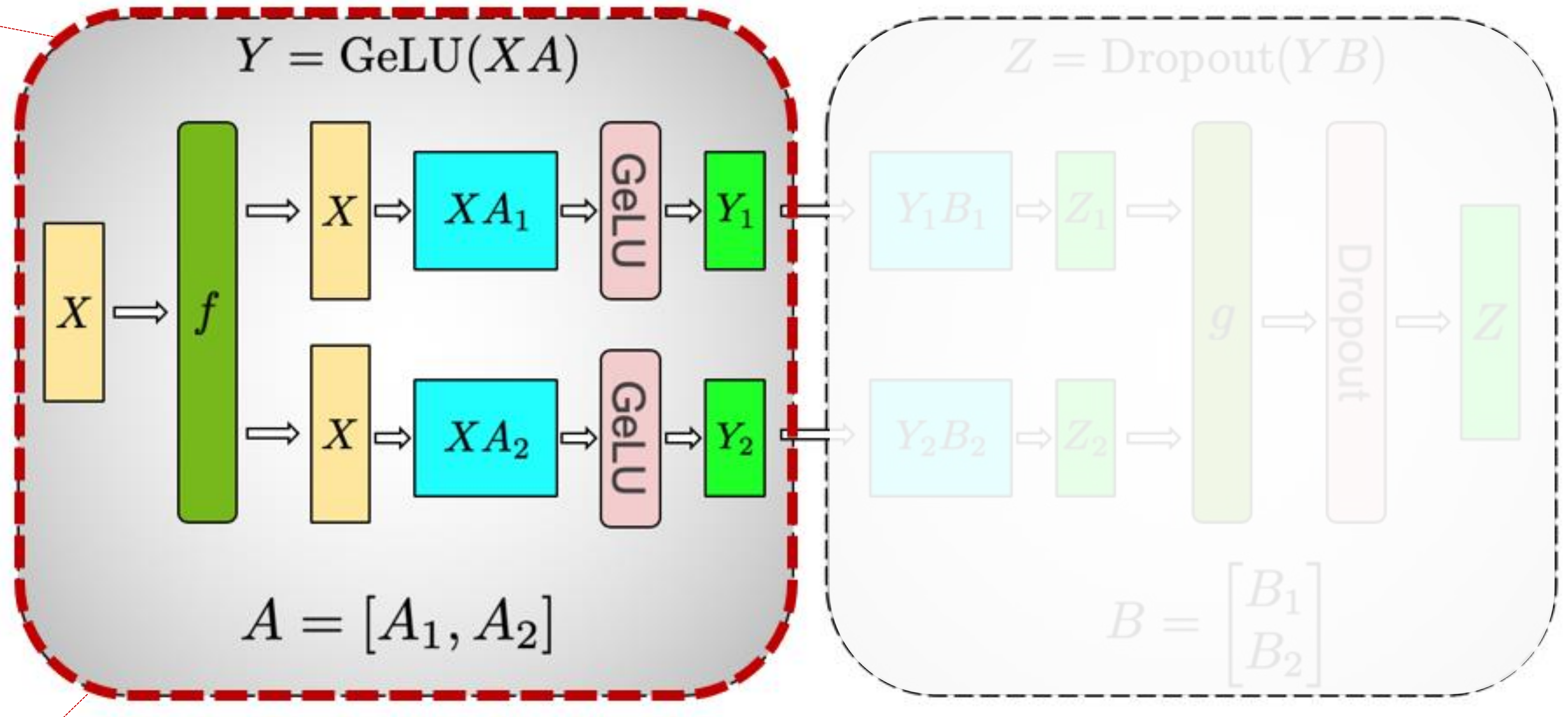
$$Y = \mathrm{GeLU}(XA)$$

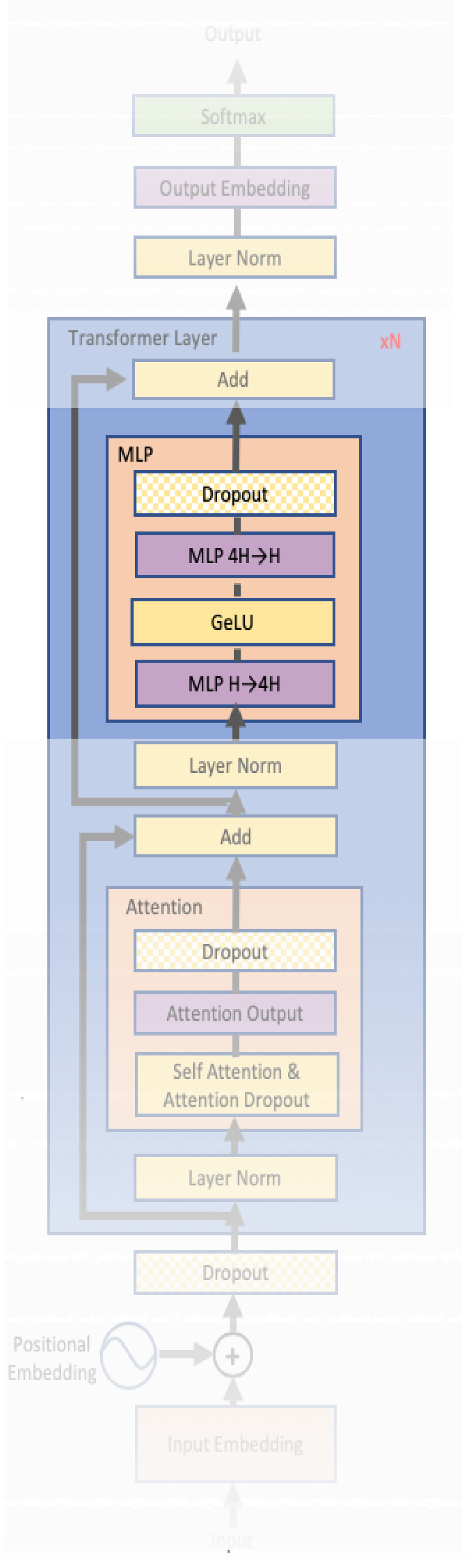


MLP TENSOR PARTITIONING

GeLU Column Parallel

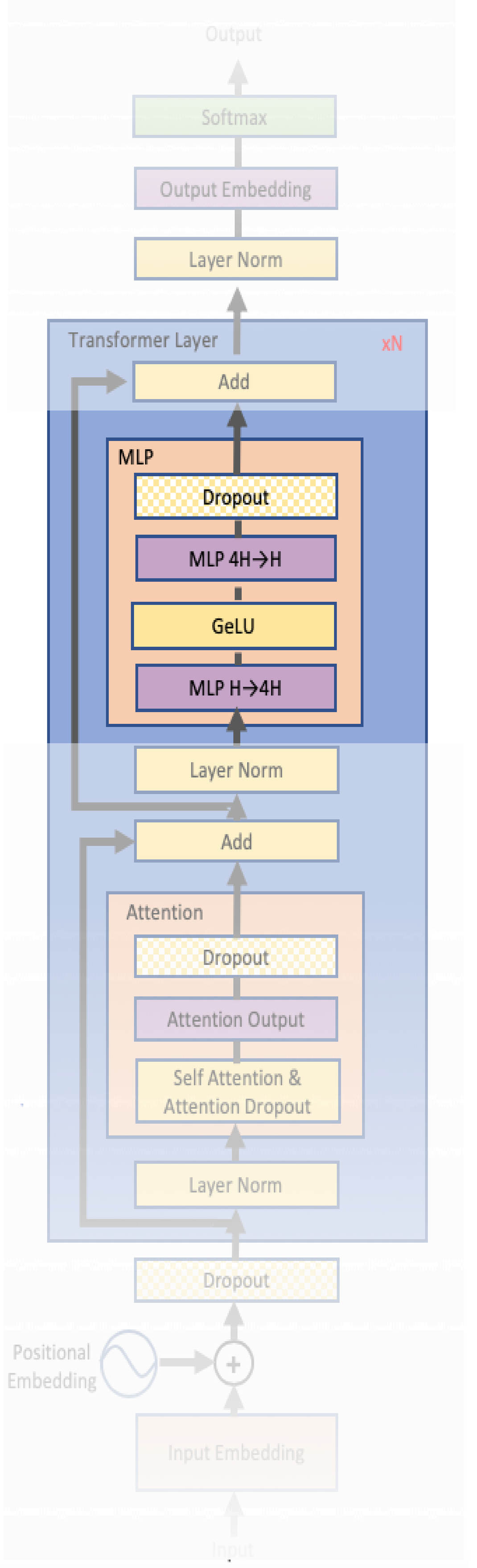






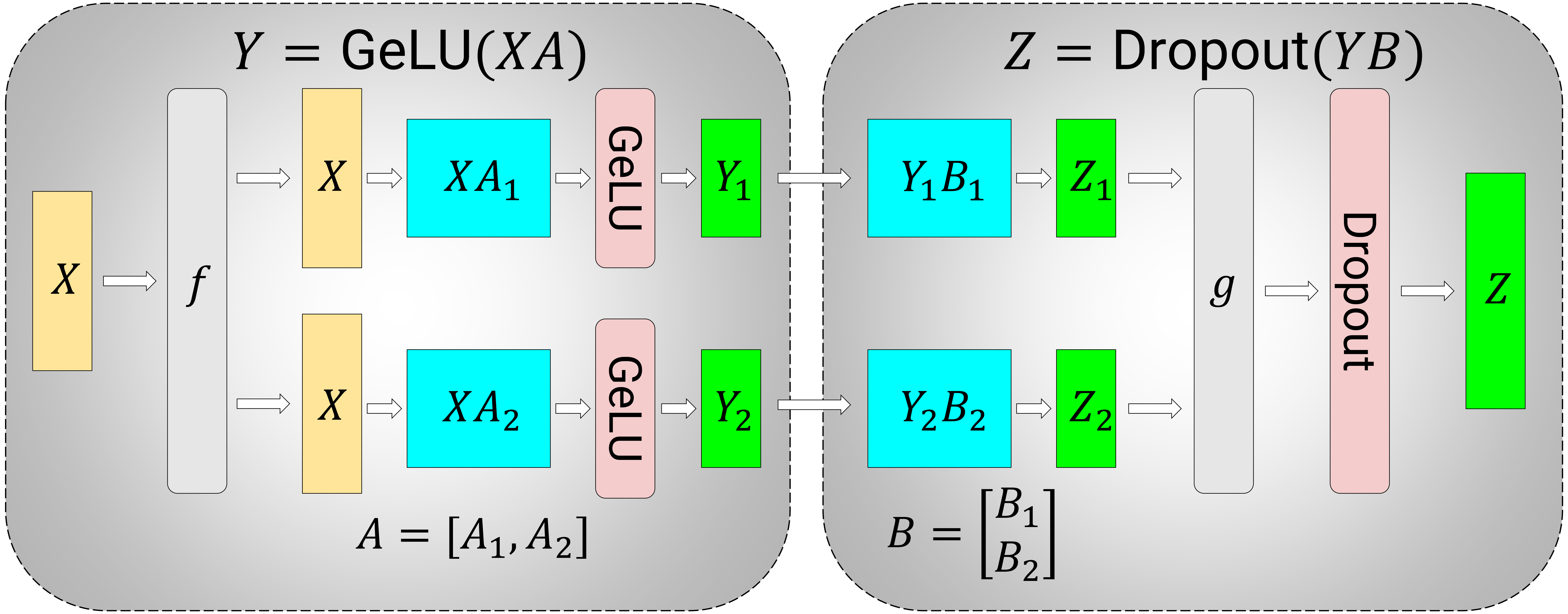
MLP TENSOR PARTITIONING

Dropout Row Parallel



TENSOR PARTITIONING

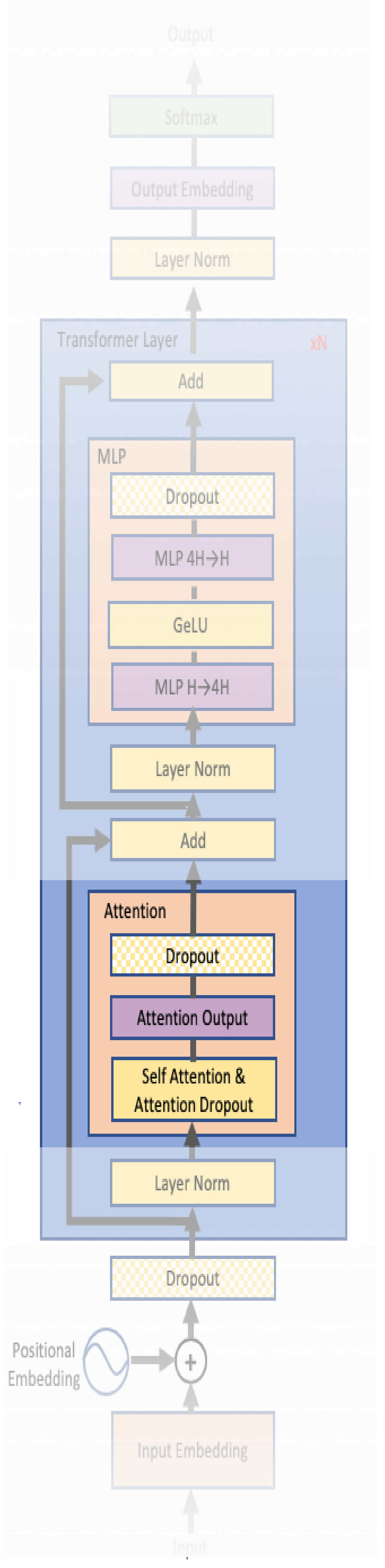
Each layer of model is partitioned over multiple GPUs



$g \rightarrow$ All-reduction $(Y_1B_1 + Y_2B_2)$ in forward pass

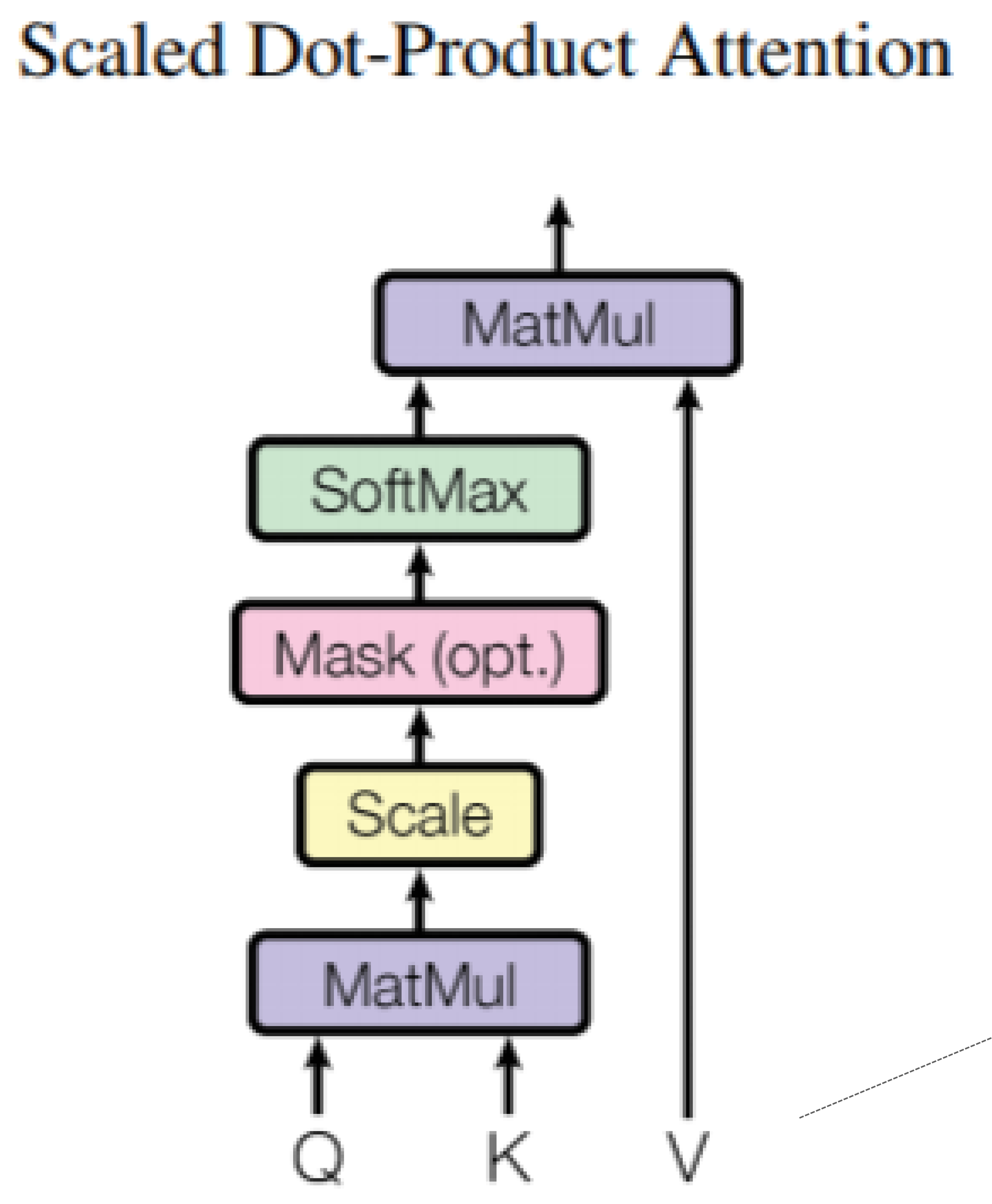
Slow across inter-server communication links

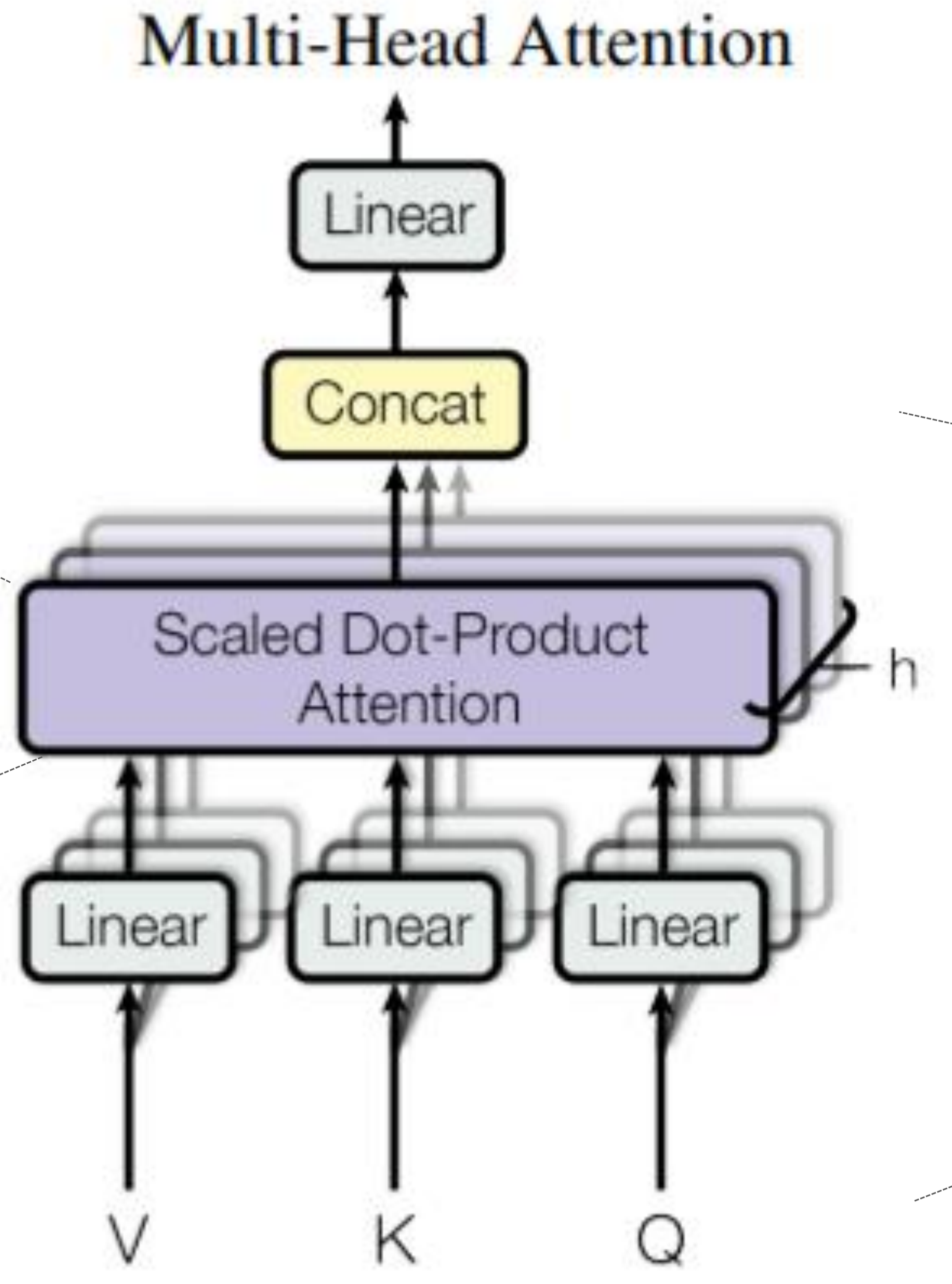
f and ***g*** are **conjugate**, ***f*** is **identity** operator in the forward pass and **all-reduce** in the backward pass while ***g*** is **all-reduce** in forward and **identity** in backward.

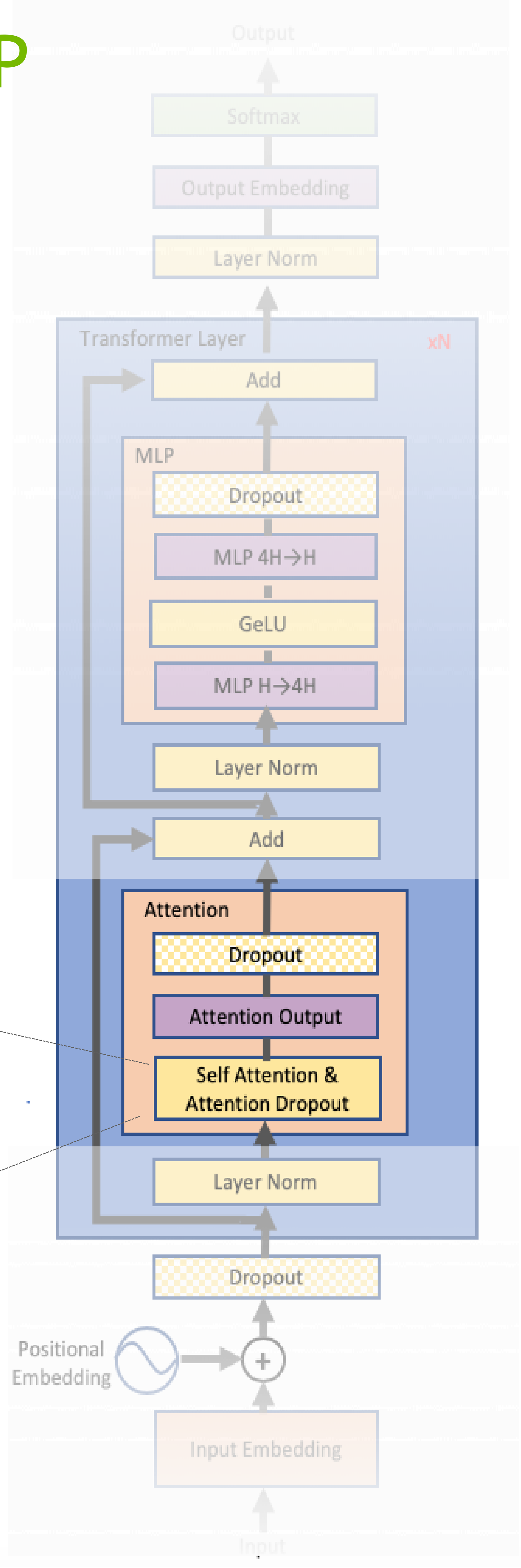


SELF-ATTENTION TENSOR PARTITIONING

Self-Attention is more complex than MLP

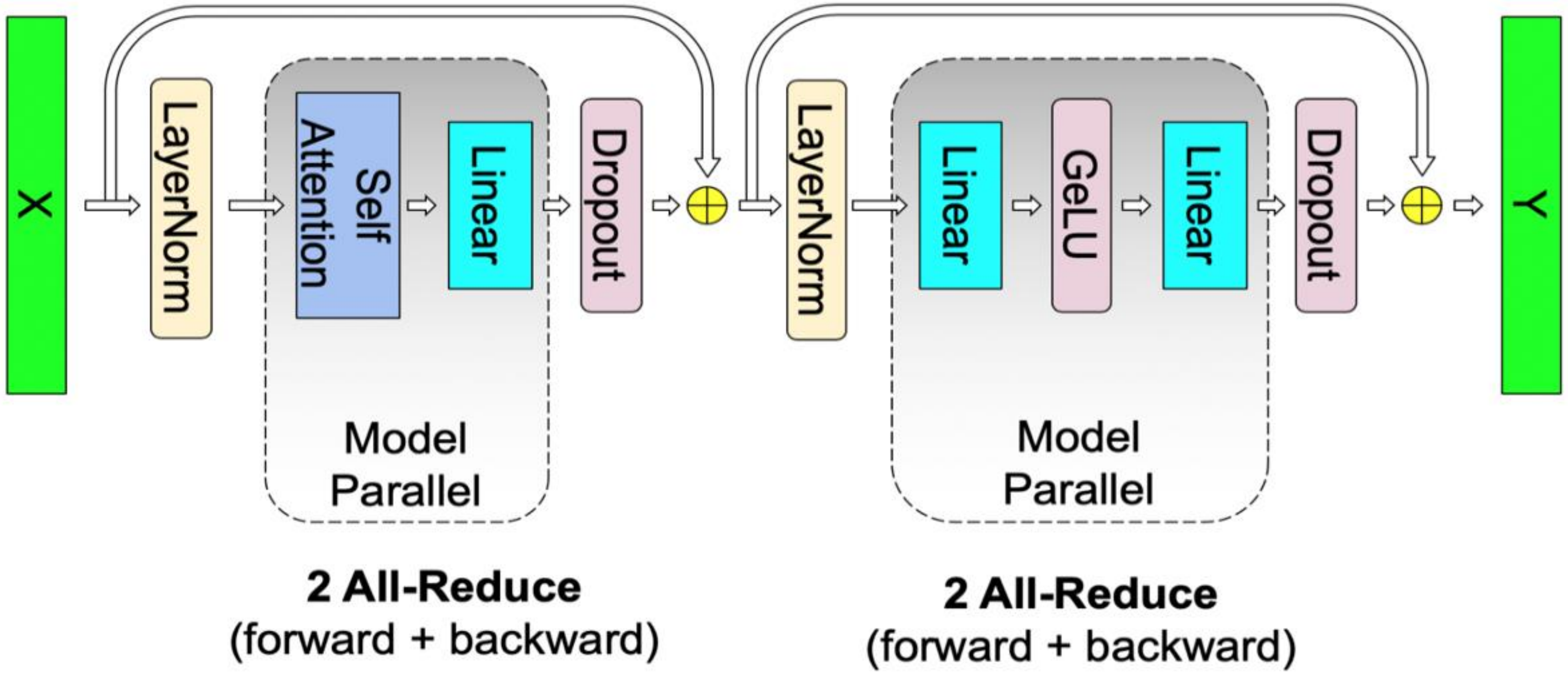






TENSOR PARALLEL TRANSFORMER LAYER

All Together



PIPELINE PARALLELISM

Challenges

PIPELINE PARALLELISM

Challenges – Idle Workers

PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

Pipeline model parallelism

- Layers / operators in model sharded over GPUs (i.e., each GPU is responsible for a subset of layers in the model)
- Each batch split into smaller microbatches and execution pipelined across these microbatches
- Point-to-point communication between consecutive pipeline stages
- Pipeline bubble at the start and end of every batch (equal to $(p-1)$ microbatches’ forward and backward passes)

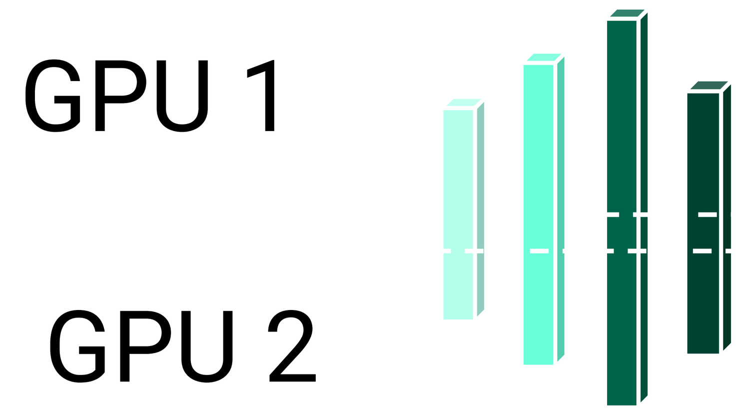
Interleaved pipeline parallelism

Smaller pipeline bubble but more communication

PIPELINE AND TENSOR PARALLELISM

Interleaved Pipeline

Tensor Parallelism



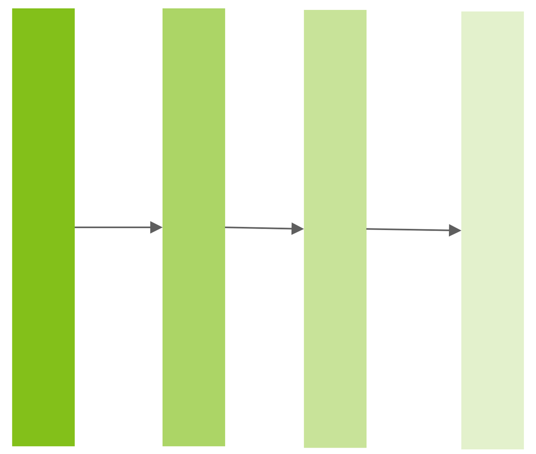
- Split individual layers across multiple GPUs where all devices compute different parts of Layers
- Challenge: Communication expensive
- Great performance within a server using NVSwitch
- Limitations: Limited number of Model Architectures | GPT-3 & T5

Pipeline Parallelism

- Split contiguous groups of layers across multiple GPUs so that Layers 0,1,2 and layers 3,4,5 are on different GPUs ...
- Communication cheap, maximizes GPU utilization over InfiniBand
- Good performance at larger batch sizes (pipeline stall amortized)
- Exceptions/Limitations: No Interleave Scheduling for Pipeline parallelism

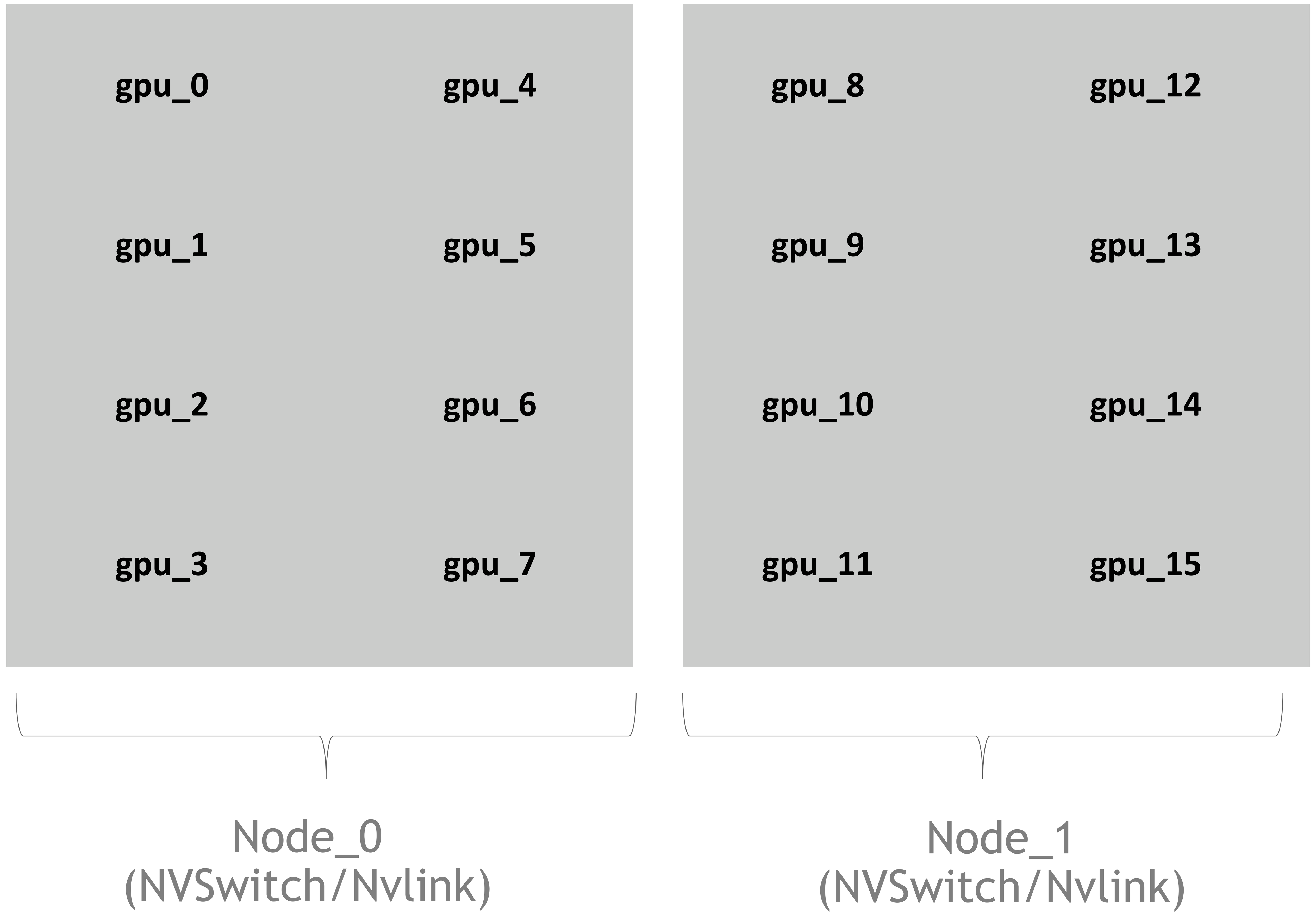
MEGATRON MODEL PARALLISM

GPU Affinity Grouping Example



Neural Network: 4 layers
Hardware: 2 nodes , 8 GPUs per node

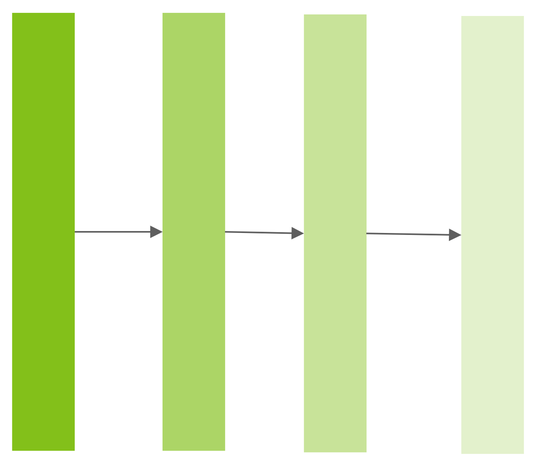
- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2



https://github.com/NVIDIA/Megatron-LM

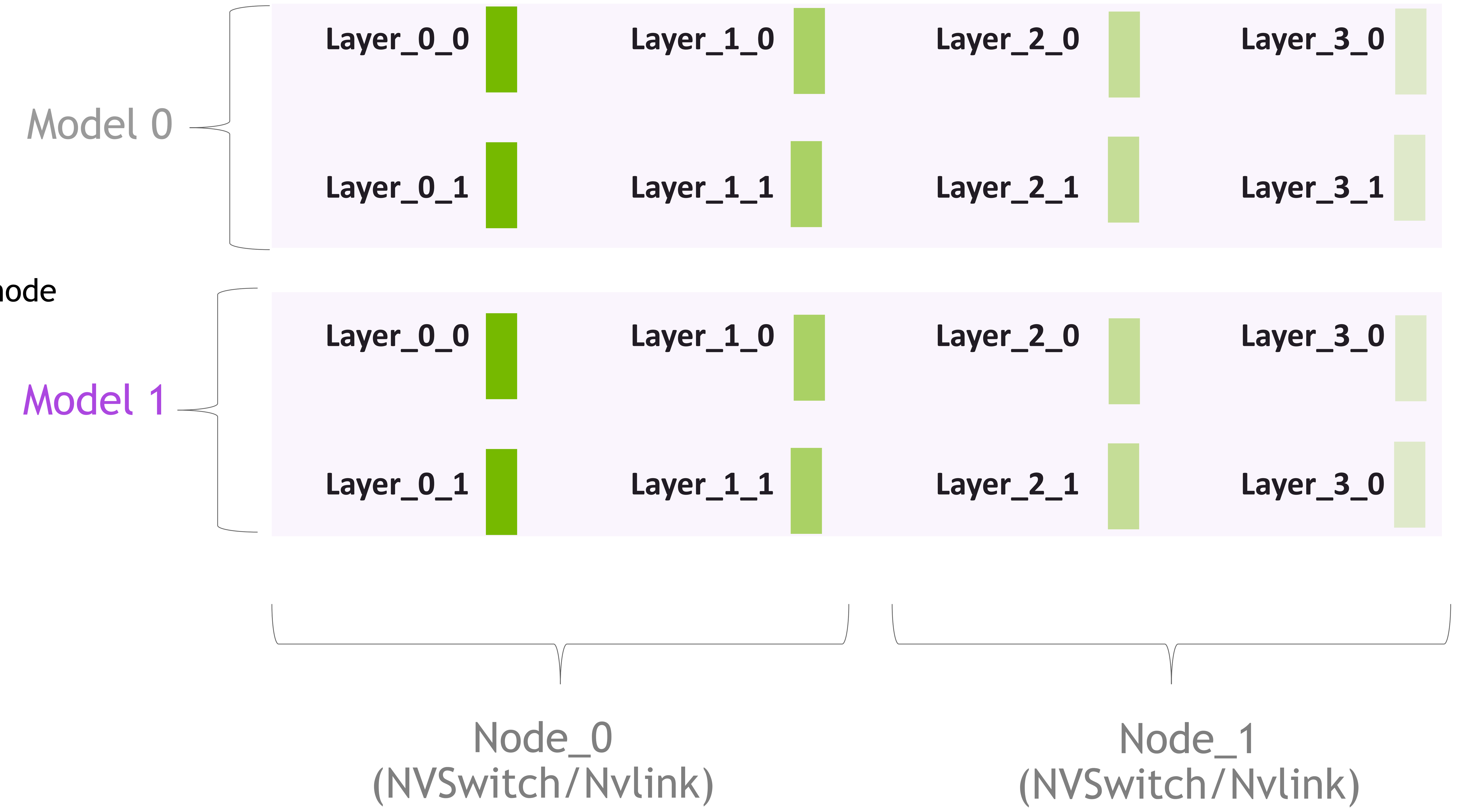
MEGATRON MODEL PARALLISM

GPU Affinity Grouping Example



Neural Network: 4 layers
Hardware: 2 nodes , 8 GPUs per node

- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2



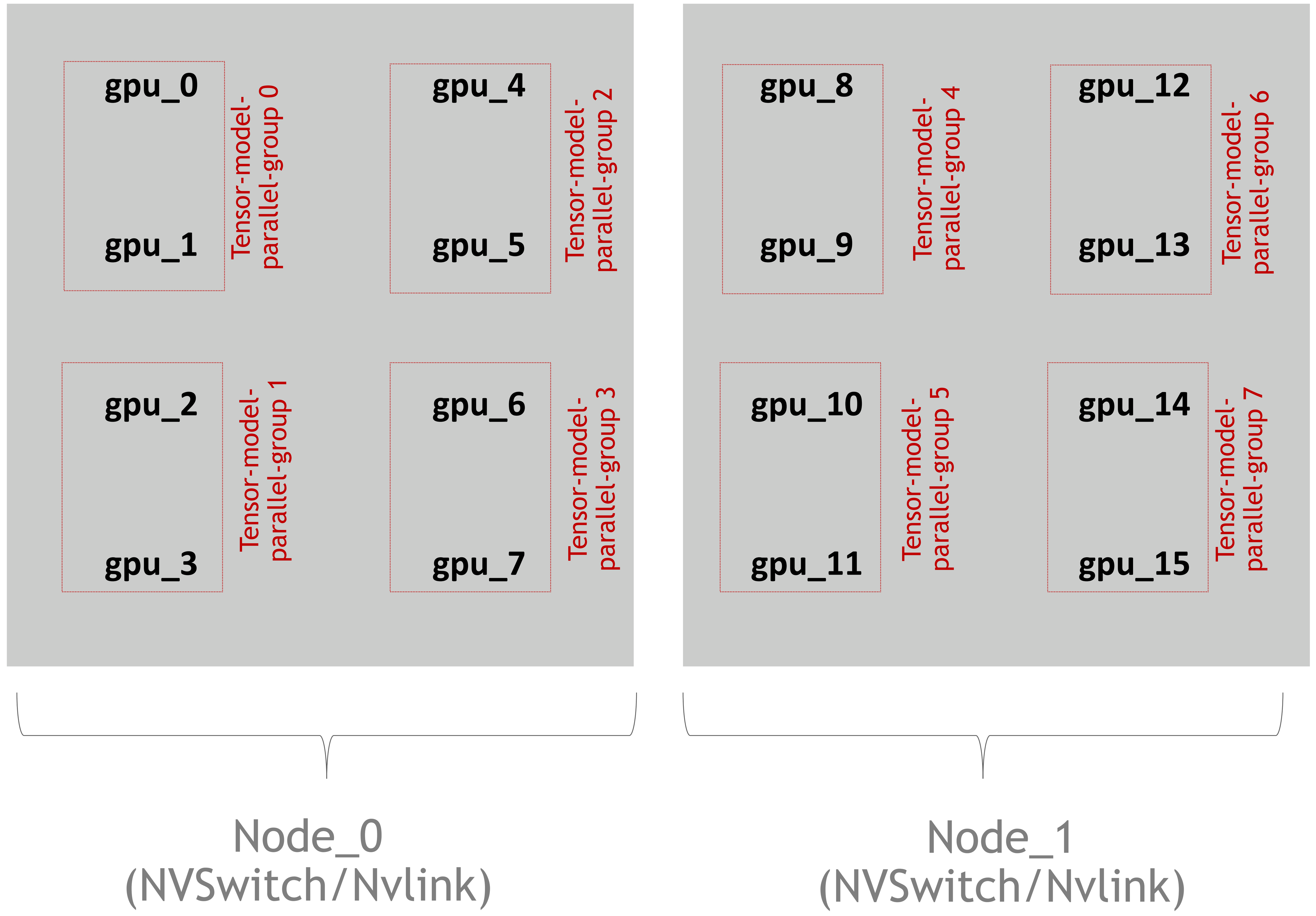
https://github.com/NVIDIA/Megatron-LM

MEGATRON MODEL PARALLISM

GPU Affinity Grouping Example

2 nodes , 8 GPUs per node

- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2



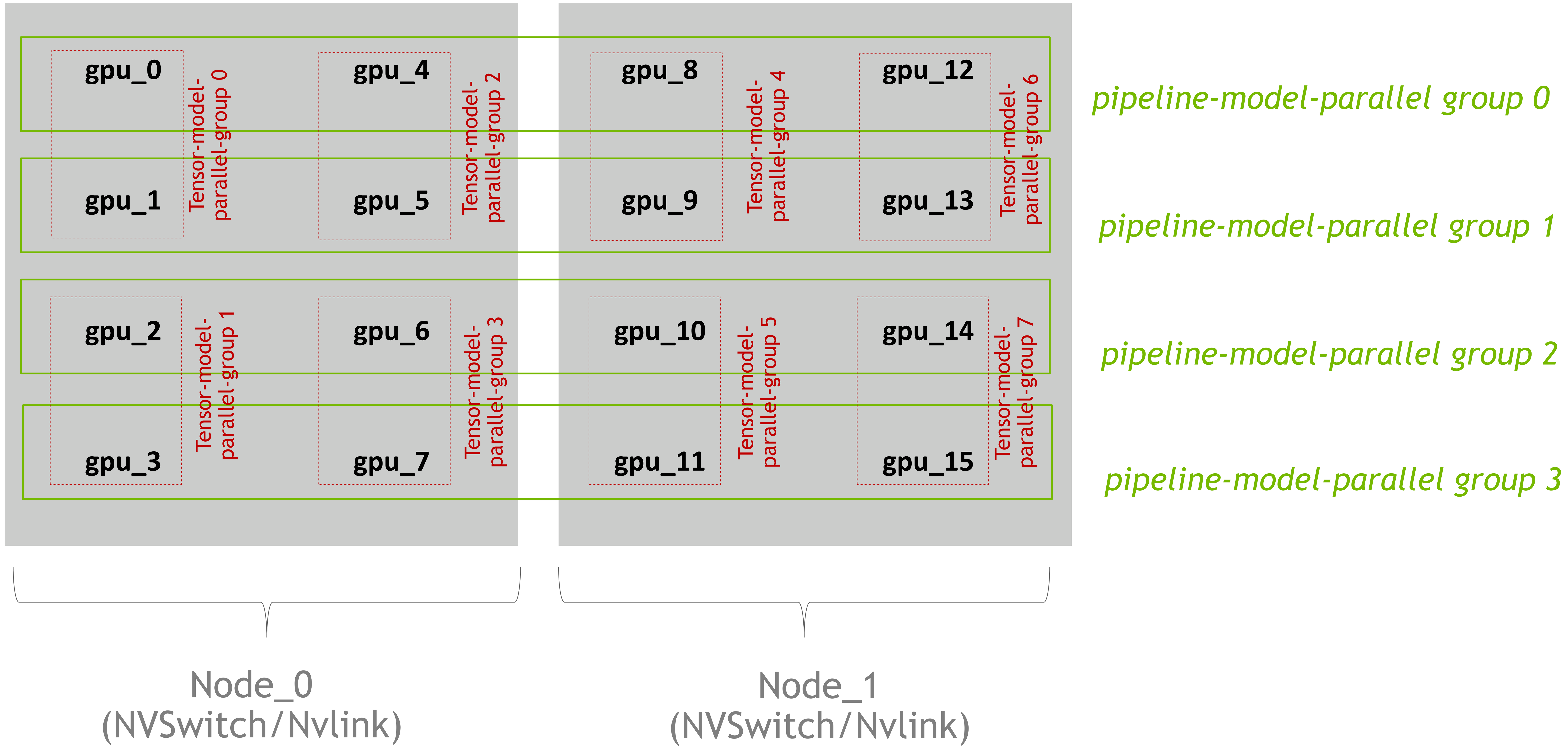
https://github.com/NVIDIA/Megatron-LM

MEGATRON MODEL PARALLISM

GPU Affinity Grouping Example

2 nodes , 8 GPUs per node

- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2



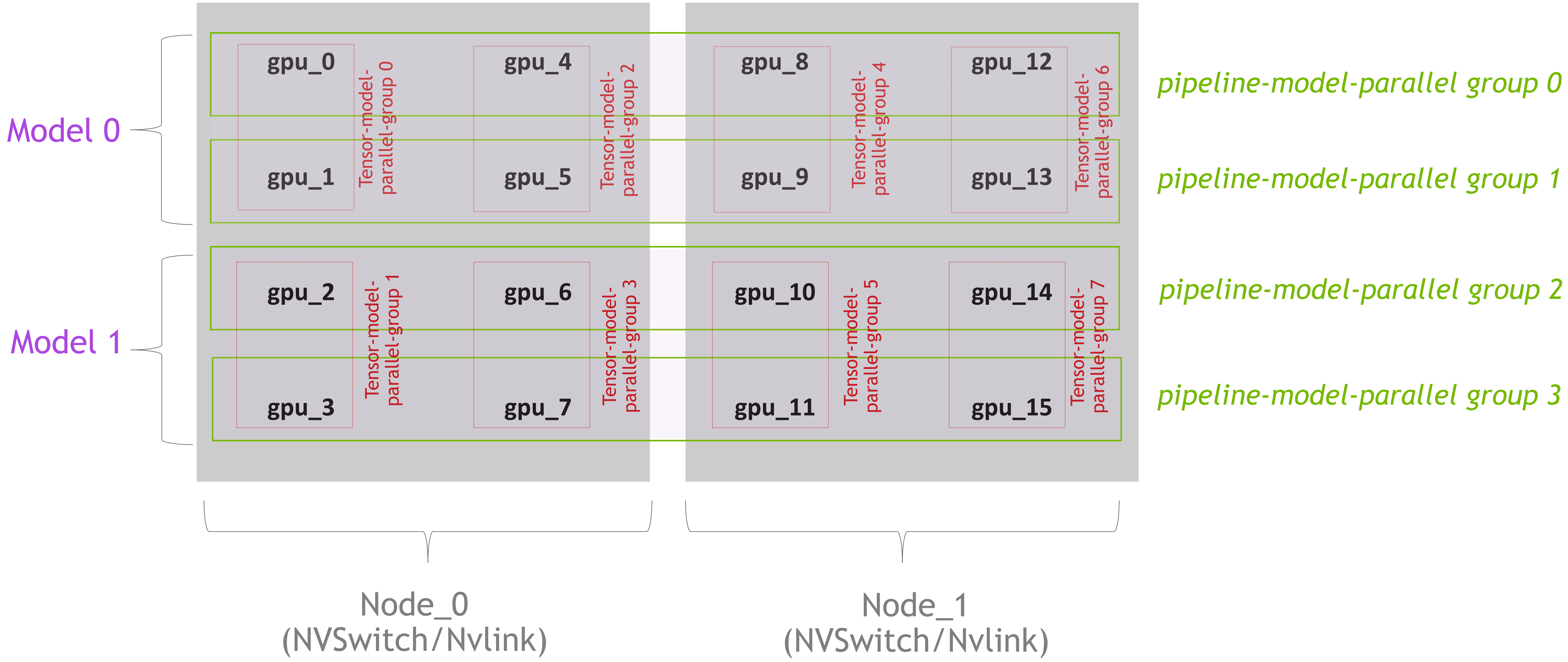
https://github.com/NVIDIA/Megatron-LM

MEGATRON MODEL PARALLISM

GPU Affinity Grouping Example

2 nodes , 8 GPUs per node

- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2



https://github.com/NVIDIA/Megatron-LM

Mixed Precision Training

Benefits of less bits

Memory
Weights and tensors occupy less space in memory

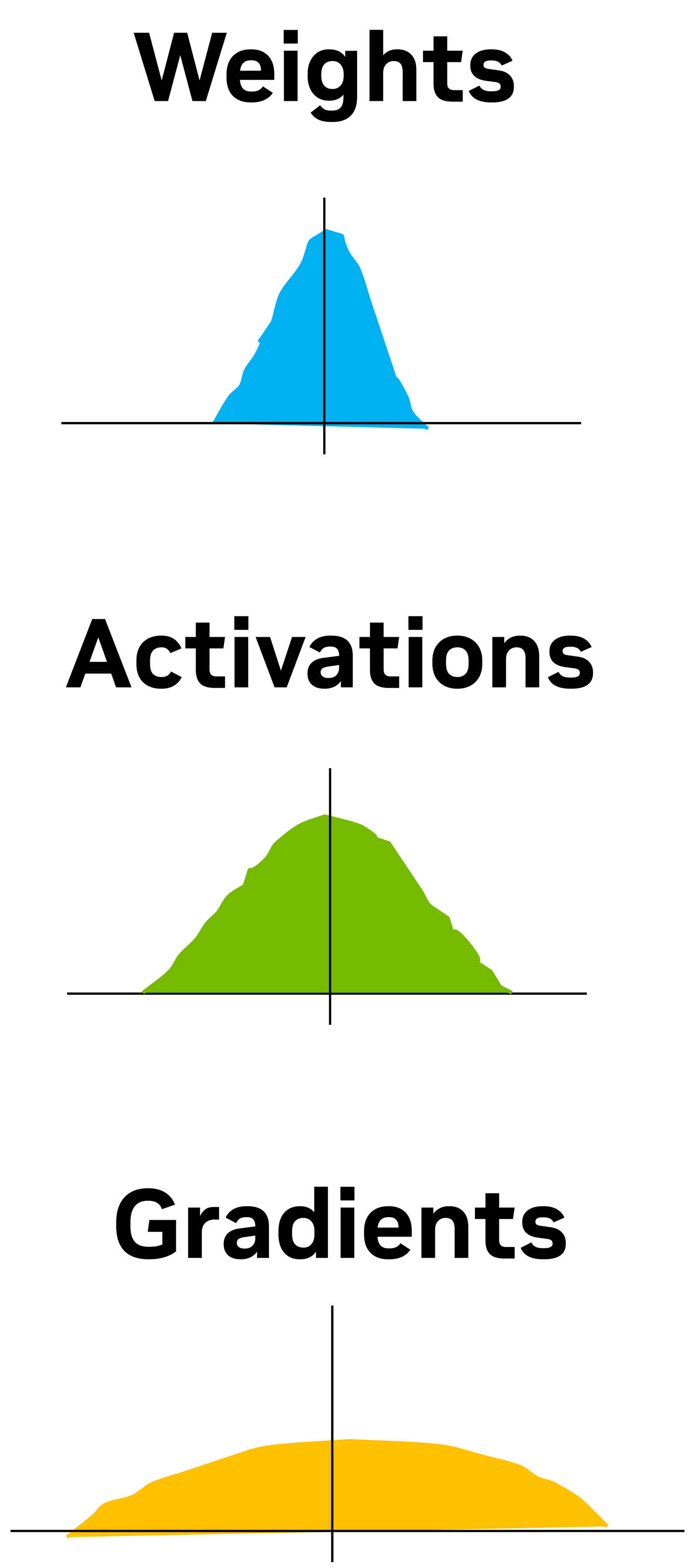
Bandwidth
Faster data movement from main memory HBM to cores (and vice versa)

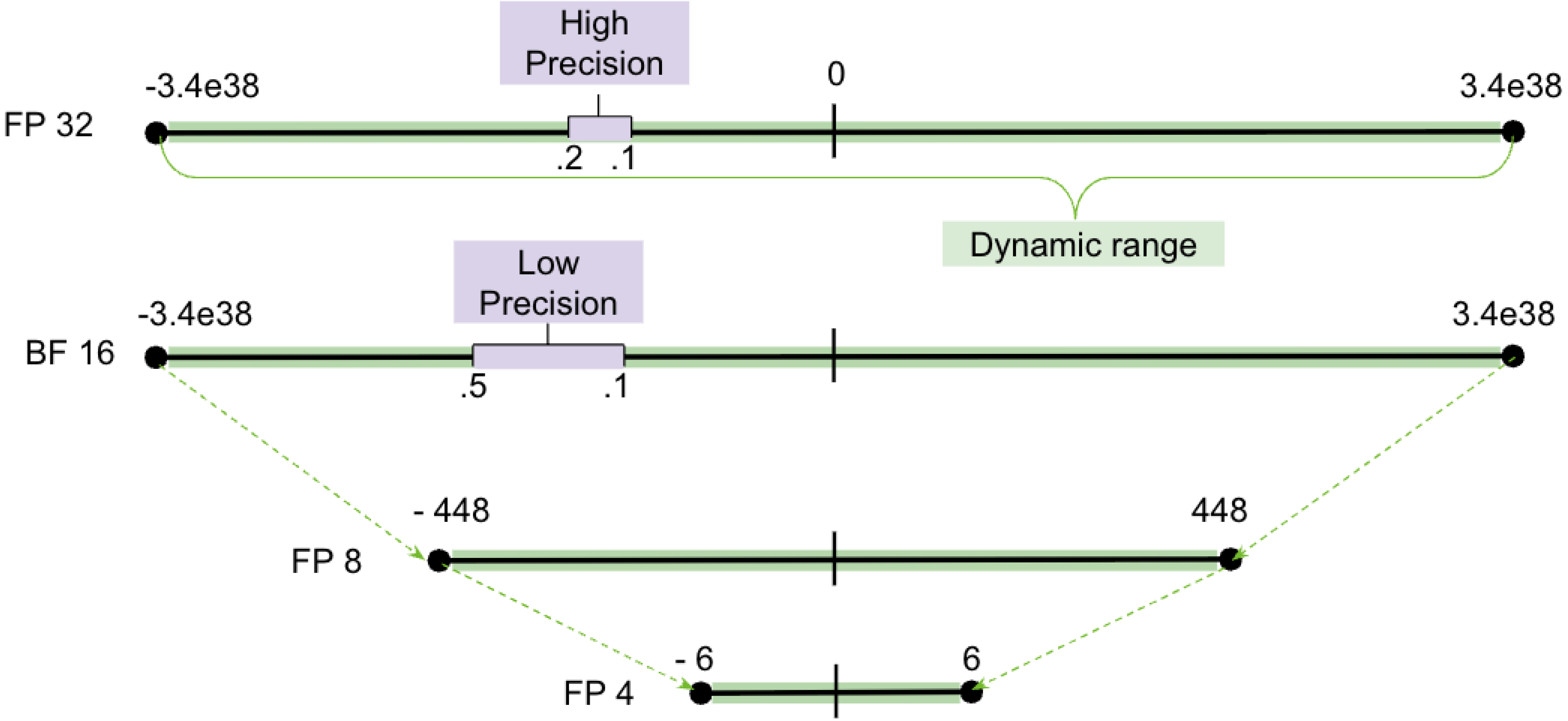
Compute
More TFLOPS with less bits
Faster matrix multiplications

	sign	exponent					mantissa										
FP16	0	0	1	1	0	1	1	0	0	1	0	1	0	0	1	1	= 0.395264
BF16	0	0	1	1	1	1	1	0	1	1	0	0	1	0	1	0	= 0.394531
FP8 E4M3	0	0	1	0	1	1	0	1									= 0.40625
FP8 E5M2	0	0	1	1	0	1	1	0									= 0.375

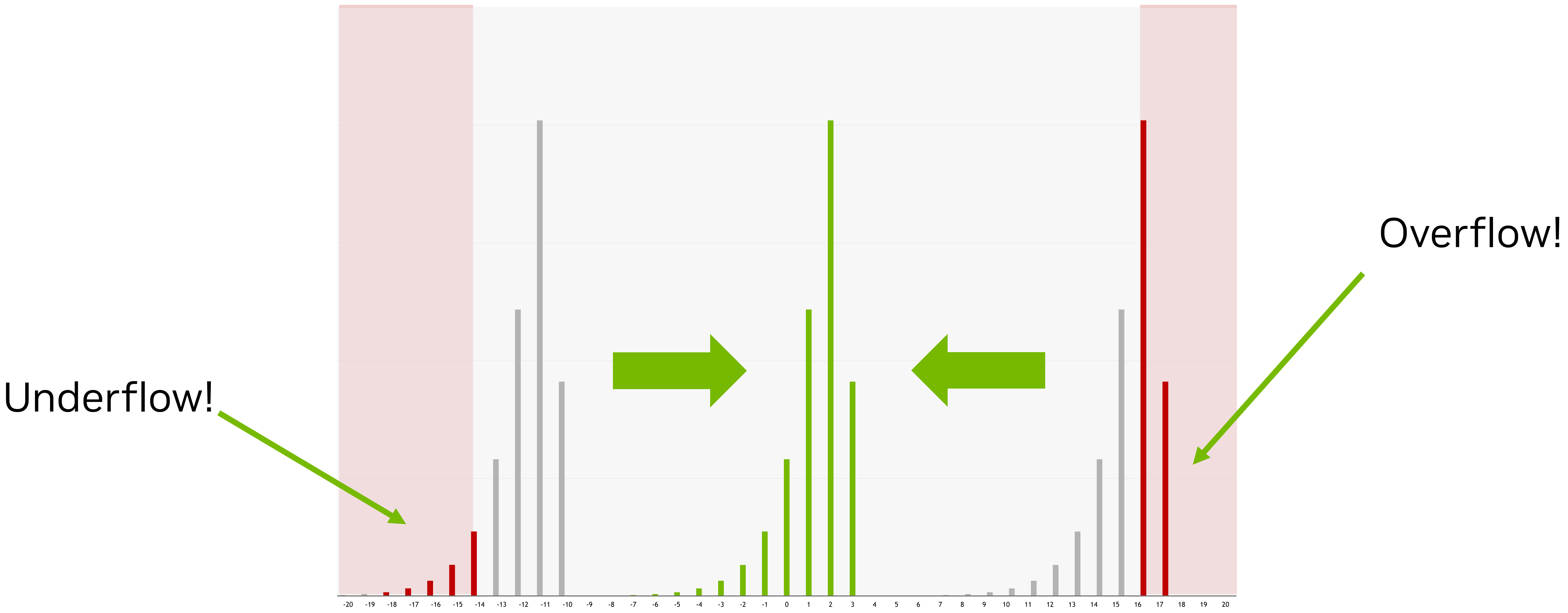
Wider distributions suffer more with Quantization

Gradients have the widest distributions





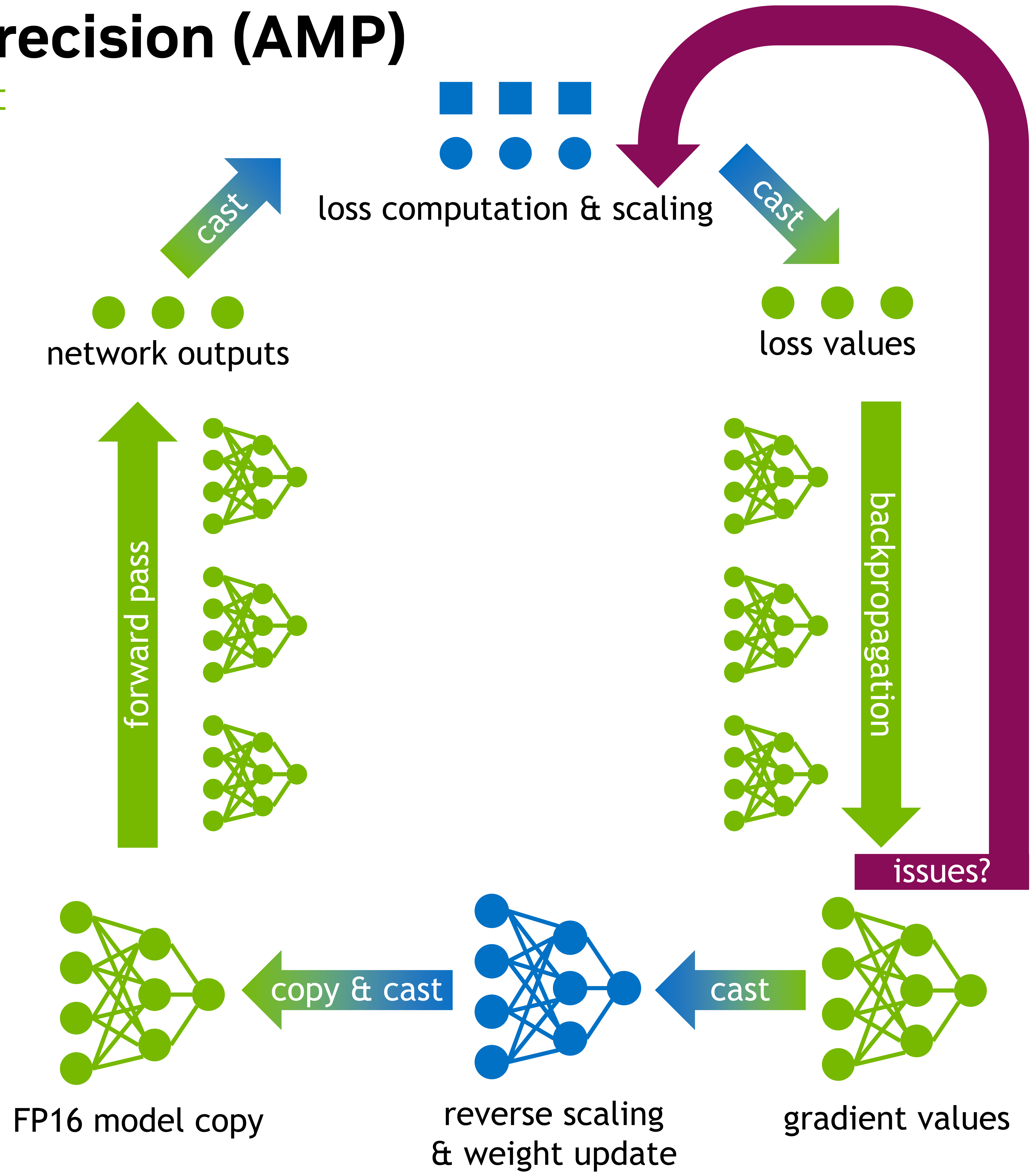
Scaling Factors to Keep Tensors within Range



Automatic Mixed Precision (AMP)

Concept

- Maintain a primary copy of weights in FP32.
- Initialize scaling factor S to a large value.
- For each iteration:
  - Make an FP16 copy of the weights.
  - Forward propagation (FP16 weights and activations).
  - Multiply the resulting loss with the scaling factor S.
  - Backward propagation (FP16 weights, activations, and their gradients).
  - If there is an Inf or NaN in weight gradients:
    - Reduce S.
    - Skip the weight update and move to the next iteration.
  - Multiply the weight gradient with 1/S.
  - Complete the weight update (including gradient clipping, etc.).
  - If there hasn't been an Inf or NaN in the last N iterations, increase S.



Towards finer Scaling Factors

Ampere

FP16 mixed precision

one scale for all tensors

Hopper

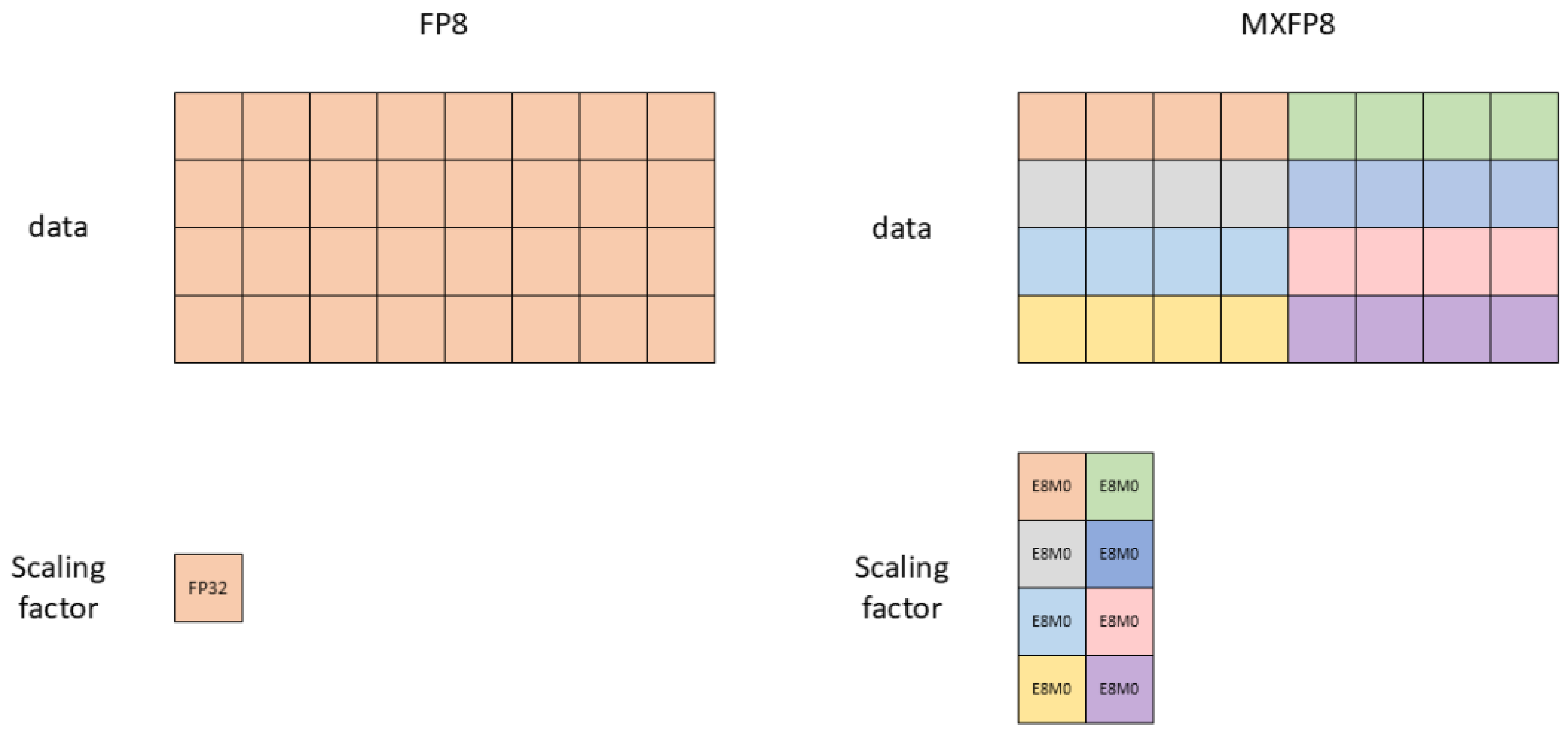
FP8 training

one scale per tensor

Blackwell

MXFP8 training*

one scale per block of 32 elements

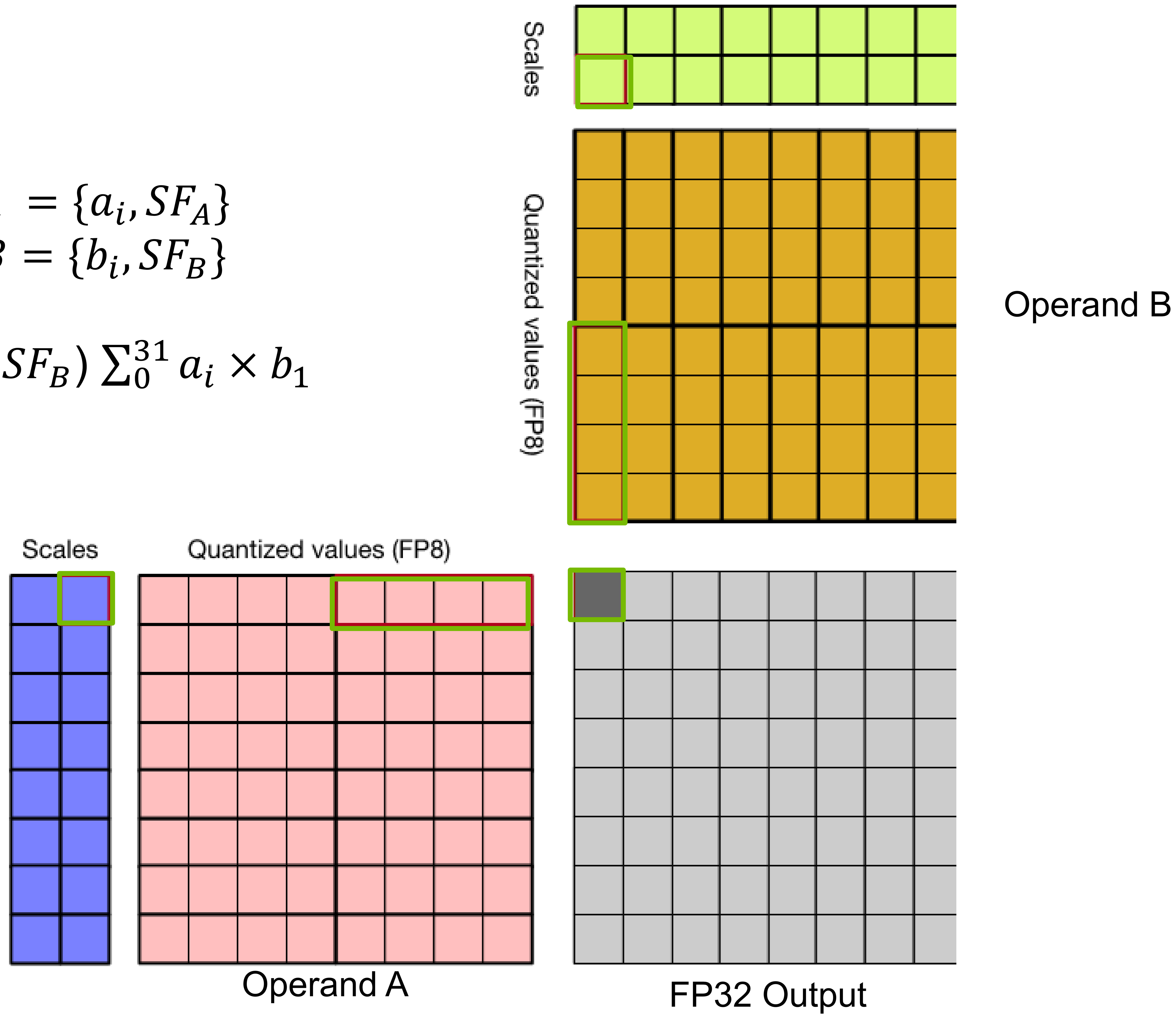


*MX stands for microscaled formats, read OCP spec

GEMM operation with MXFP8 operands

$$Operand\ A\ = \{a_i, SF_A\}$$
$$Operand\ B = \{b_i, SF_B\}$$

$$Output\ = (SF_A\ \times SF_B)\sum_0^{31} a_i \times b_1$$



Tensors are quantized to MXFP8 after FP32 accumulation

How to cast back to MXFP8?

New FP8 Recipes for Blackwell and Hopper

TransformerEngine

[Hopper] Current Scaling + 1st & last layer BF16

- Current scaling is more stable than delayed scaling, but a bit slower
- Keeps the more sensitive 1st and last layer in BF16
- E4M3 for weights and activations, E5M2 for gradients

[Hopper] NV Subchannel Recipe (DeepSeek-V3 like)

- As DeepSeek-V3 pretraining
- 1x128 blocks for input and output_grad, 128x128 blocks for weights
- E4M3 for all weights, acts, and grads

[Blackwell] MXFP8 Blockwise Scaling

- Different scaling factor for each block of 32 values in a tensor
- E4M3 for all weights, acts, and grads

Software libraries

Transformer Engine

- An open-source library implementing the FP8 recipe for Transformer building blocks
- Optimized for FP8 and other datatypes
- PyTorch and JAX are supported frameworks
- Composable with the native framework operators
- Supports different types of model parallelism
  - DP, TP, PP, CP
- cuDNN kernels available such as GroupedGEMM
- https://github.com/NVIDIA/TransformerEngine
- Docs:
  - https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/index.html

```python
import torch
import transformer_engine.pytorch as te
from transformer_engine.common import recipe

# Set dimensions.
in_features = 768
out_features = 3072
hidden_size = 2048

# Initialize model and inputs.
model = te.Linear(in_features, out_features, bias=True)
inp = torch.randn(hidden_size, in_features, device="cuda")

# Create MXFP8 recipe.
fp8_recipe = recipe.MXFP8BlockScaling()

# Enable autocasting to FP8.
with te.fp8_autocast(enabled=True, fp8_recipe=fp8_recipe):
    out = model(inp)

# Calculate loss and gradients.
loss = out.sum()
loss.backward()
```

Recipes Available in NeMo FW and Megatron-LM

The backend is Transformer Engine

- NVIDIA NeMo framework allows developers to easily run multi-node training of LLMs
- Megatron-LM is research-oriented framework to train LLMs
- Recipes available in https://github.com/NVIDIA/NeMo/blob/main/nemo/collections/llm/recipes/precision/mixed_precision.py

```python
trainer = nl.Trainer(
    devices=args.devices,
    num_nodes=args.num_nodes,
    max_steps=args.max_steps,
    log_every_n_steps=args.log_interval,
    val_check_interval=args.val_check_interval,
    limit_val_batches=args.limit_val_batches,
    strategy=strategy,
    accelerator="gpu",
    plugins=bf16_with_mxfp8_mixed(),
    use_distributed_sampler=False,
    callbacks=[itr_rate_callback],
)
```

```python
def fp16_with_mxfp8_mixed() -> run.Config[MegatronMixedPrecision]:
    """Create a MegatronMixedPrecision plugin configuration for mixed

    Returns:
        run.Config[MegatronMixedPrecision]: Configuration for FP16 wi
    """
    cfg = fp16_mixed()
    cfg.fp8 = 'hybrid'
    cfg.fp8_recipe = "mxfp8"
    cfg.fp8_param_gather = False
    return cfg
```

Developer Tools and Resources

Accelerate innovation and growth

Learn more: developer.nvidia.com

Individuals

Software
100s of APIs, models, SDKs, microservices, and early access to NVIDIA tech

Training
Hands-on self-paced courses, instructor-led workshops, and certifications

GPU Sandbox
Approval basis, multi-GPU and multi-node

Learning
Tutorials, self-paced courses, blogs, documentation, code samples

Community
Dedicated developer forums, meetups, hackathons

Ecosystem
GTC, NVIDIA Partner Network

Organizations

Startups
Cloud credits, engineering resources, technology discounts, exposure to VCs

Venture Capital
Deal flow and portfolio support for Venture Capital firms

Higher Education
Teaching kits, training, curriculum co-development, grants

ISVs and SIs
Engineering guidance, discounts, marketing opportunities

Research
Grant programs, collaboration opportunities

Enterprises
Tailored developer training, skills certification, technical support

Thank You