# MODEL PARALLELISM: BUILDING AND DEPLOYING LARGE NEURAL NETWORKS

## ADVANCED CONCEPTS OF LARGE DISTRIBUTED TRAINING


DEEP
LEARNING
INSTITUTE

# PART 2

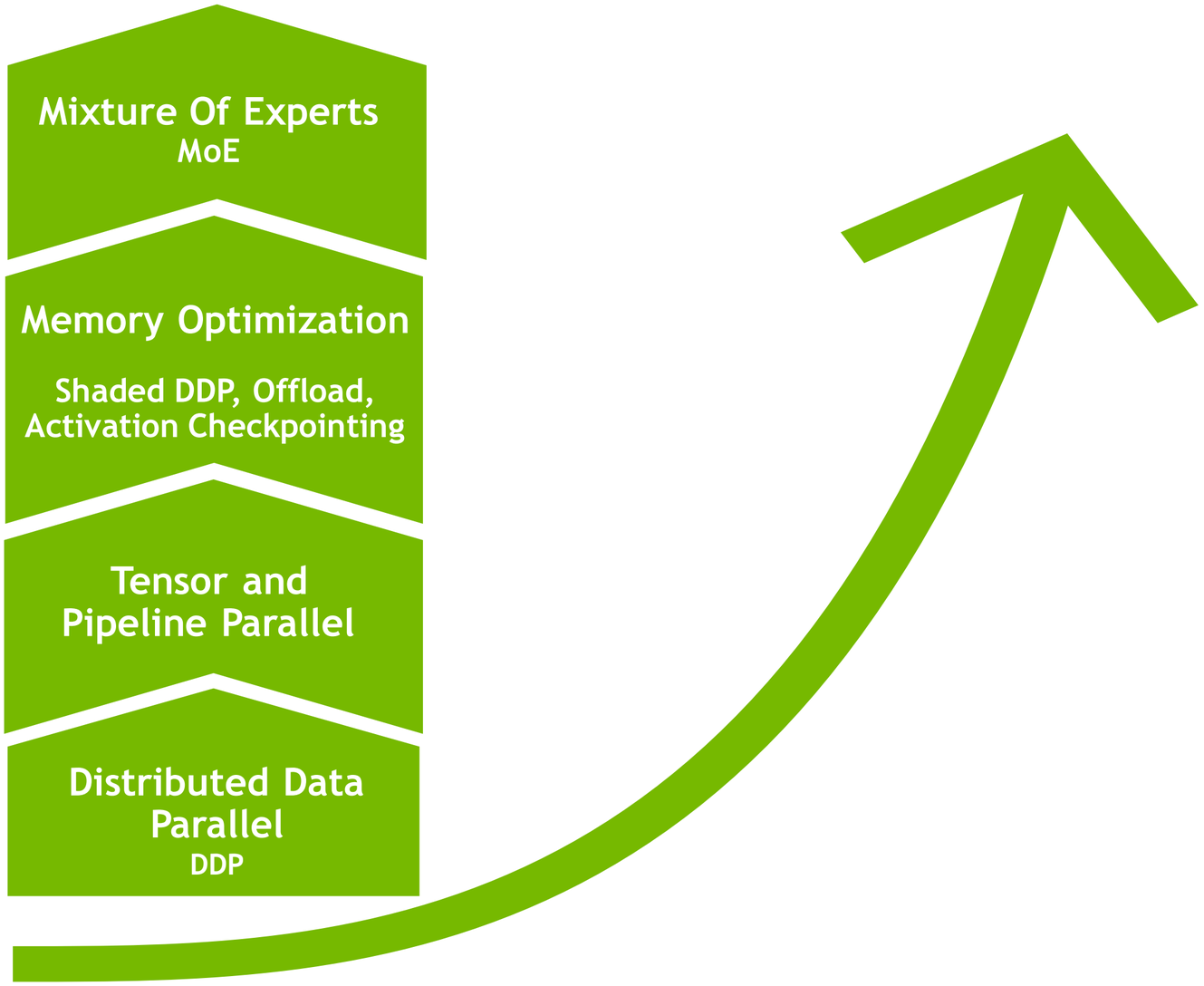
## Advanced concepts of Large Distributed Training

- Lecture
  - Distributed Data Parallel – DDP
  - Sharded data
  - Tensor/Pipeline Parallel
  - Sequence parallelism
  - Activations checkpointing
  - Offload
  - Mixture of Experts
  - Collective Communication
  - Hardware design
- Lab
  - Multi-Nodes Distributed Training for Computer Vision
  - Mixture of Experts

# SCALING CHALLENGES

# SCALING CHALLENGES

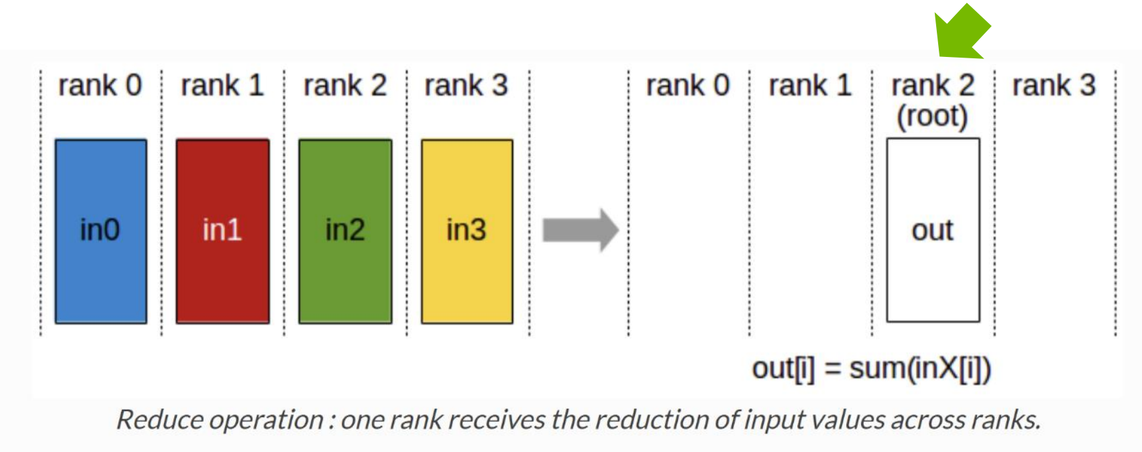
## Distributed Training



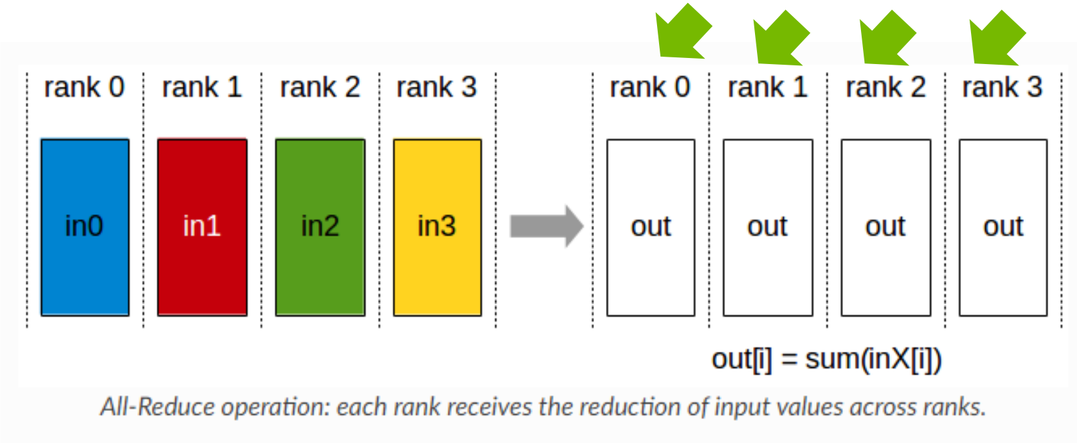
# BASIC COLLECTIVE COMMUNICATIONS FOR DISTRIBUTED TRAINING

# SOME NOTIONS

## Collective Communications




*Reduce operation : one rank receives the reduction of input values across ranks.*

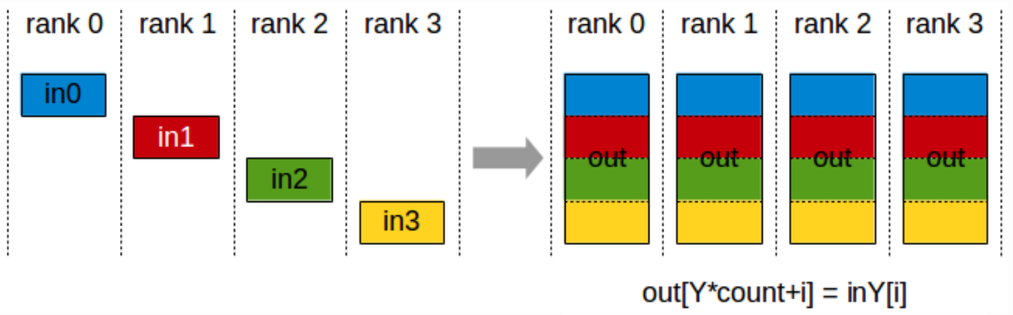



*All-Reduce operation: each rank receives the reduction of input values across ranks.*

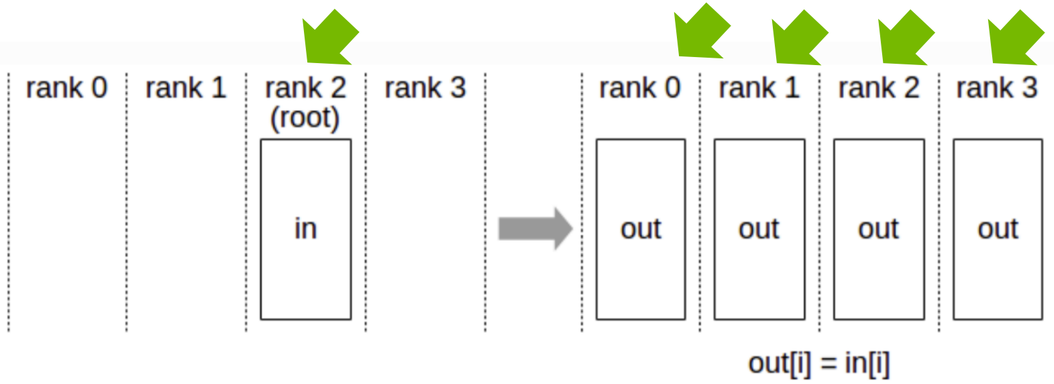
https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/operations.html

# SOME NOTIONS

## Collective Communications




*AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.*



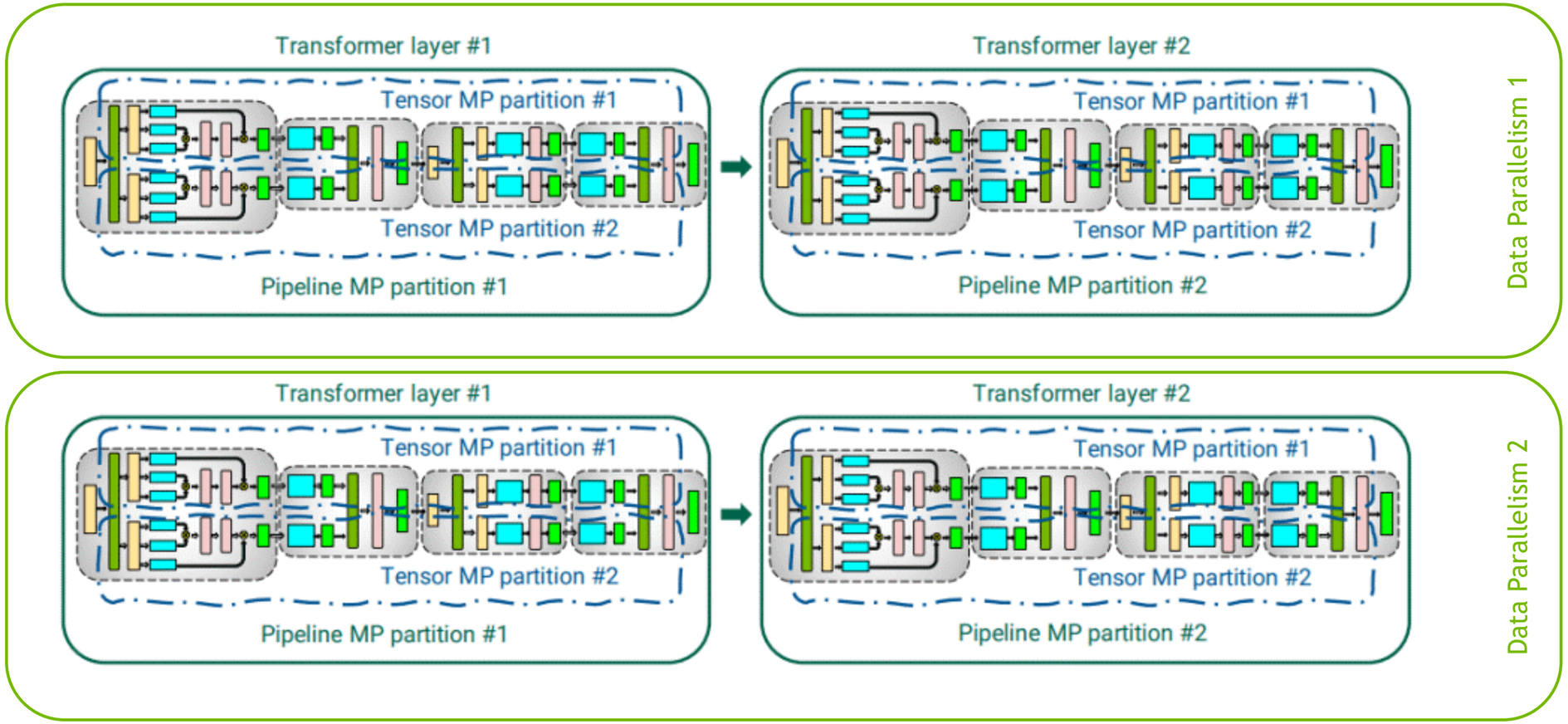

*Broadcast operation: all ranks receive data from a "root" rank.*

https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/operations.html

# DEALING WITH MEMORY CONSTRAINTS

# DEALING WITH MEMORY CONSTRAINTS

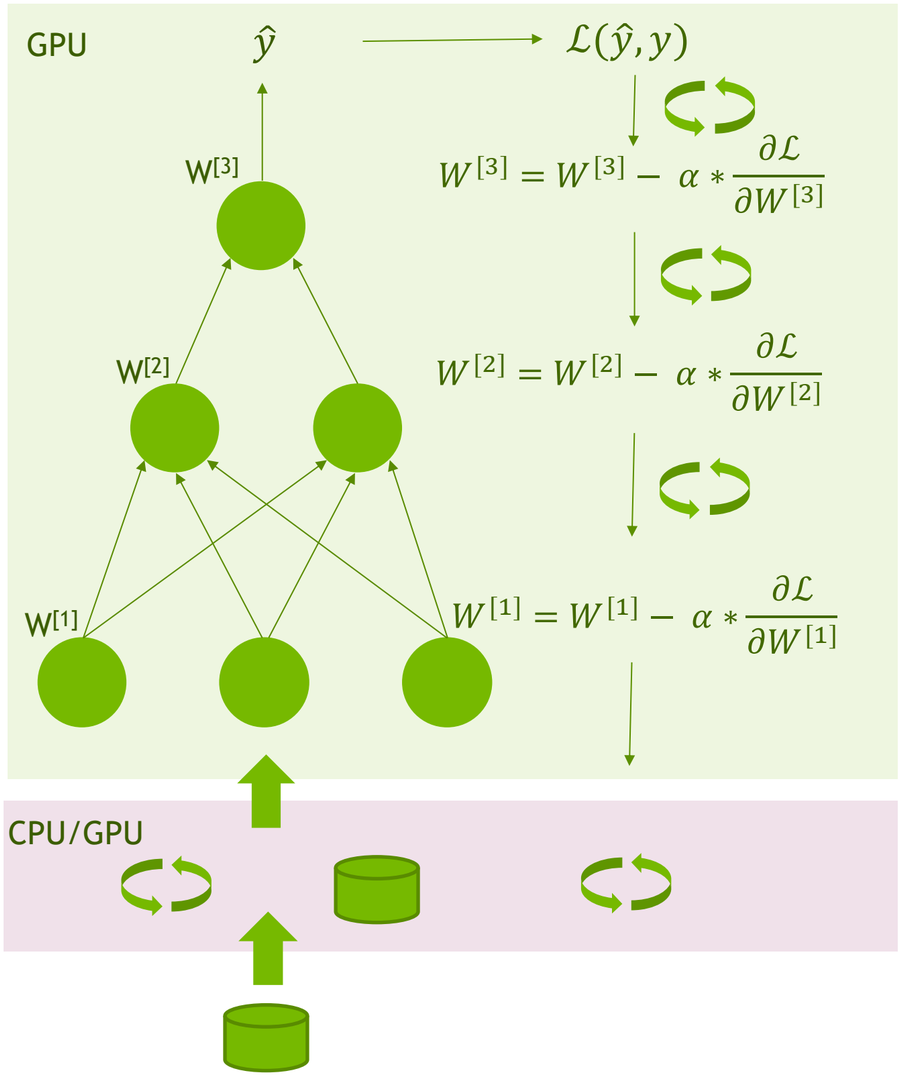
## Various forms of parallelism

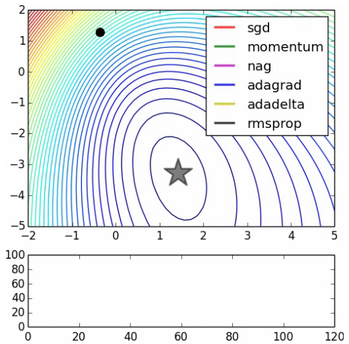


# DISTRIBUTED DATA PARALLEL - DDP

# TRAINING A NEURAL NETWORK

## Single GPU

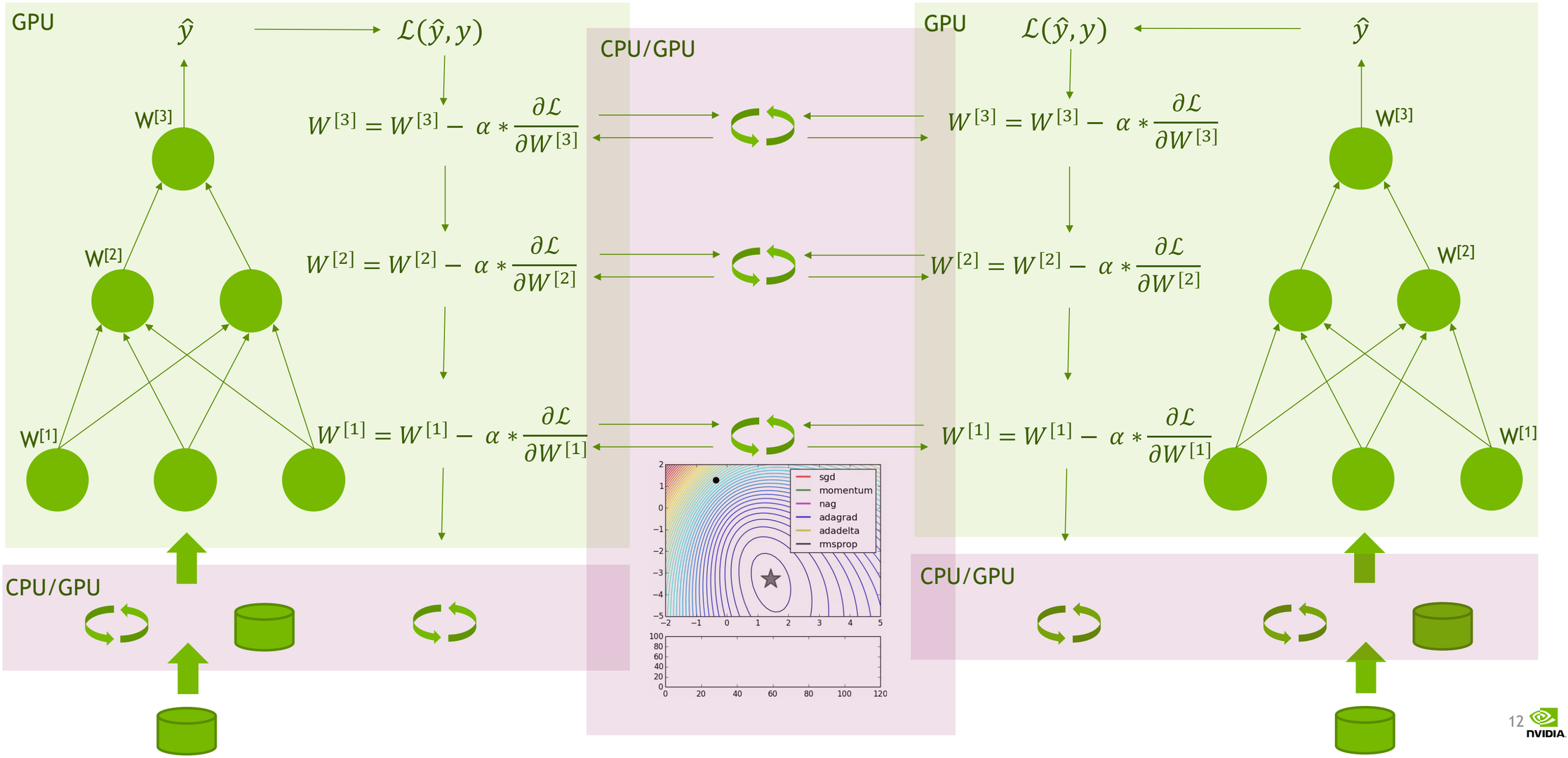



1. Read the data
2. Transport the data
3. Pre-process the data
4. Queue the data
5. Transport the data
6. Calculate activations for layer one
7. Calculate activations for layer two
8. Calculate the output
9. Calculate the loss
10. Backpropagate through layer three
11. Backpropagate through layer two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

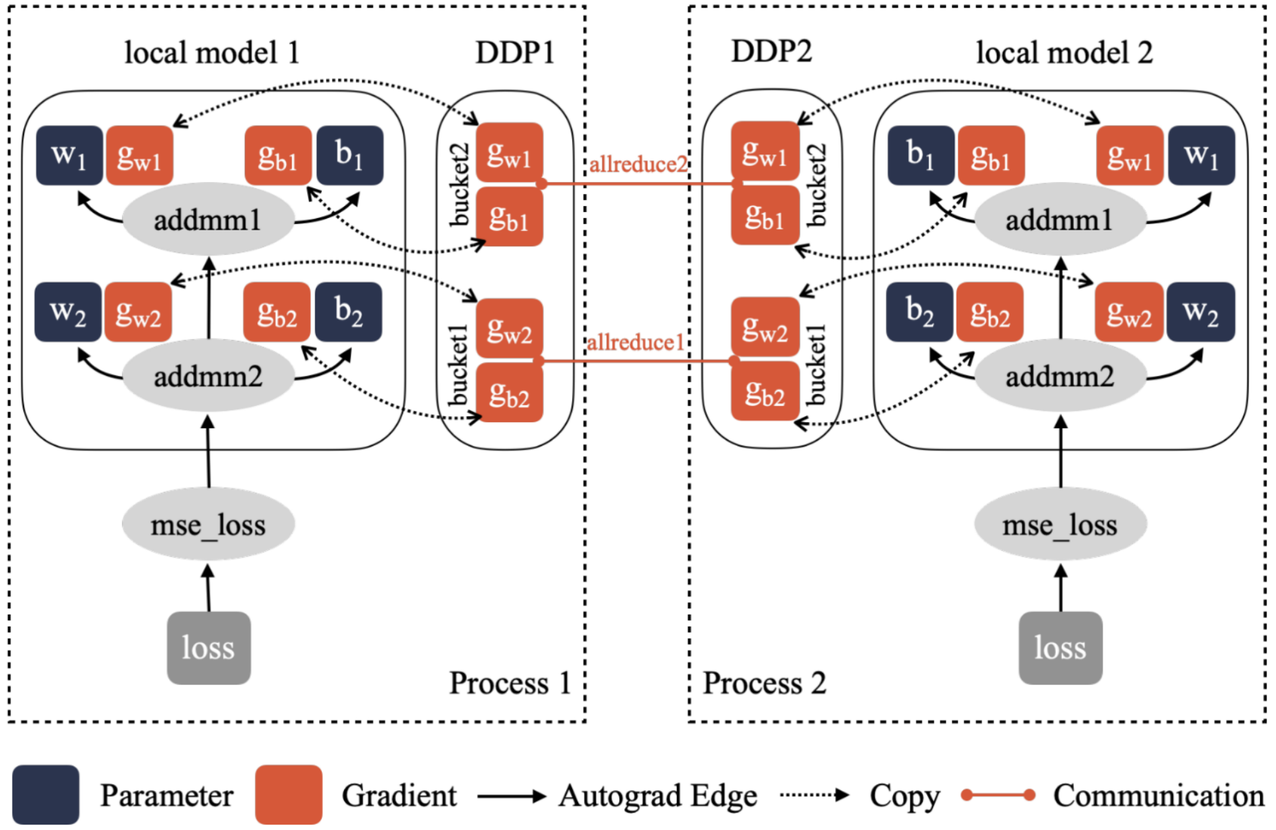
# TRAINING A NEURAL NETWORK

## Multiple GPUs with DDP



# DISTRIBUTED DATA PARALLEL - DDP

## PyTorch Distributed Data Parallel



# DISTRIBUTED DATA PARALLEL - DDP

## PyTorch Distributed Data Parallel

| Backend | gloo | | mpi | | nccl | |
|---|---|---|---|---|---|---|
| Device | CPU | GPU | CPU | GPU | CPU | GPU |
| send | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| recv | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| broadcast | ✓ | ✓ | ✓ | ? | ✗ | ✓ |
| all_reduce | ✓ | ✓ | ✓ | ? | ✗ | ✓ |
| reduce | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| all_gather | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| gather | ✓ | ✗ | ✓ | ? | ✗ | ✓ |
| scatter | ✓ | ✗ | ✓ | ? | ✗ | ✗ |
| reduce_scatter | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| all_to_all | ✗ | ✗ | ✓ | ? | ✗ | ✓ |
| barrier | ✓ | ✗ | ✓ | ? | ✗ | ✓ |

https://pytorch.org/docs/stable/distributed.html

# IT HELPS TO TRAIN FASTER BUT DOES IT HELP WITH MEMORY?

# DDP OVERHEADS
# REPLICATION MEMORY EXPLOSION

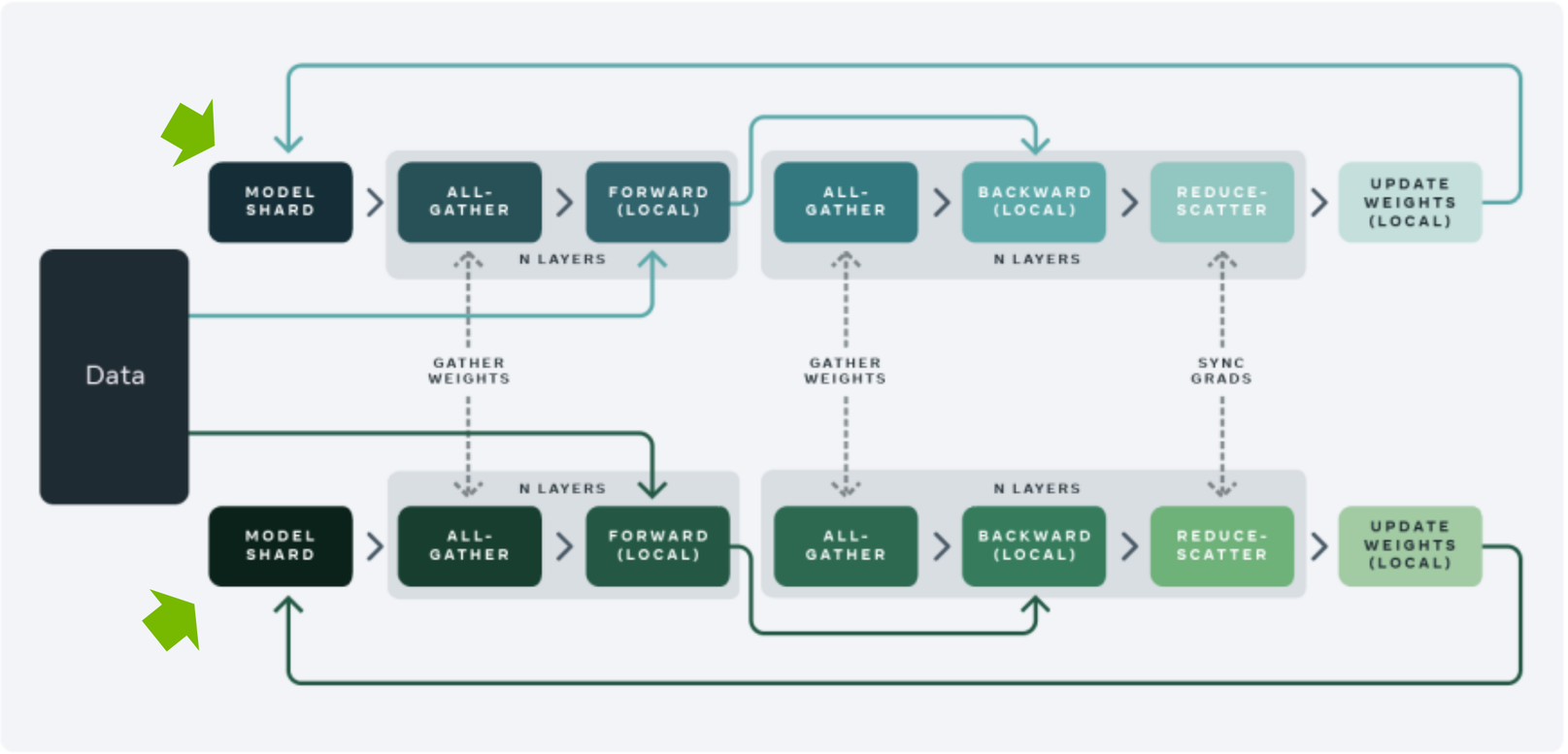
# SAVE GPU MEMORY
# SHARD DATA AND MODEL

# DISTRIBUTED DATA PARALLEL - DDP

## FairScale: Fully Sharded Data Parallel - FSDP

For each GPU:

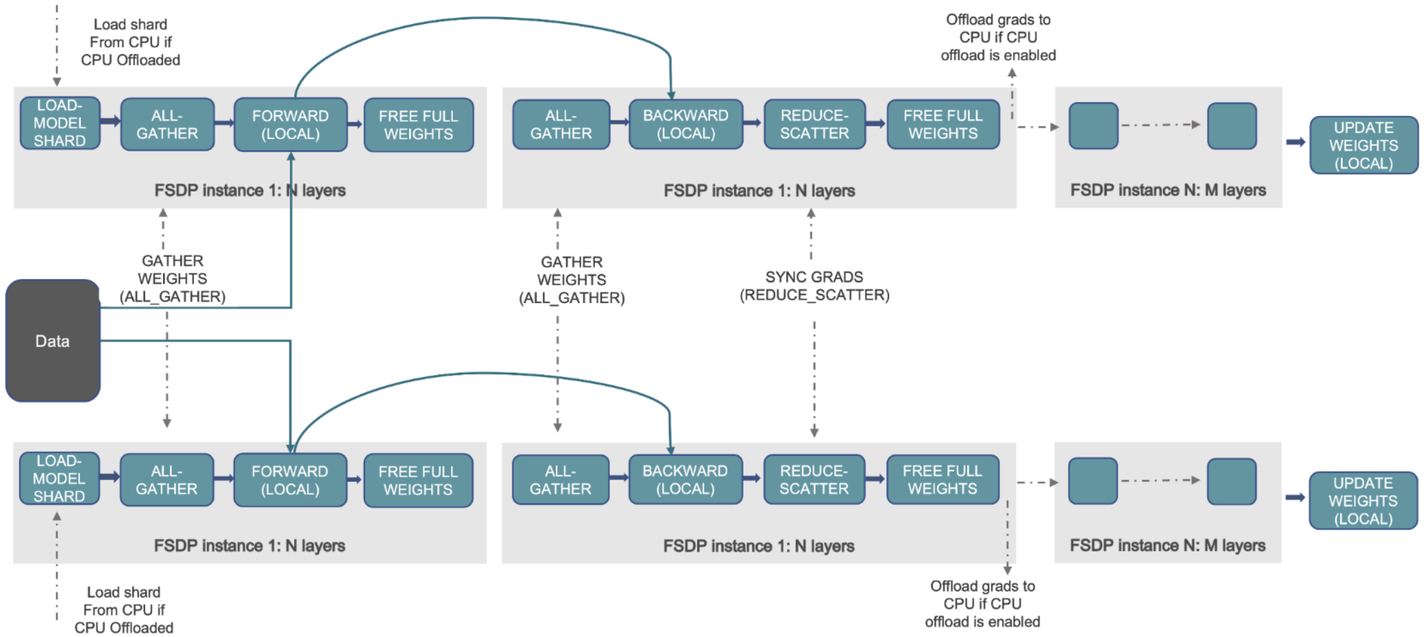
1. Get the shard of the model
2. Get the shard of the data
3. Local forward pass: Gather weights from the others
4. Local backward pass: Gather again weights from the others
5. Local weights shard update: Synchronize Gradients

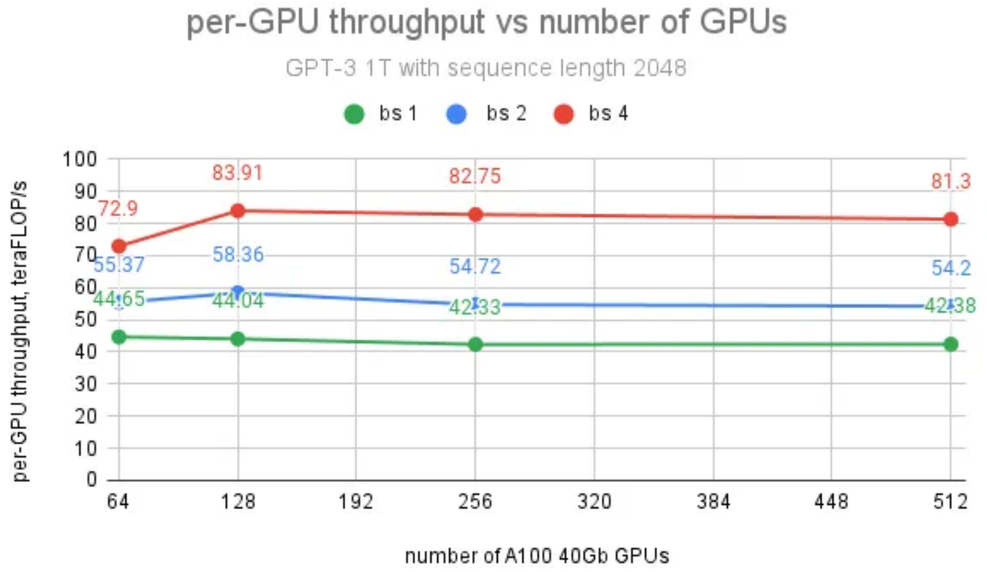



https://fairscale.readthedocs.io/en/stable/api/nn/fsdp.html

# 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.*

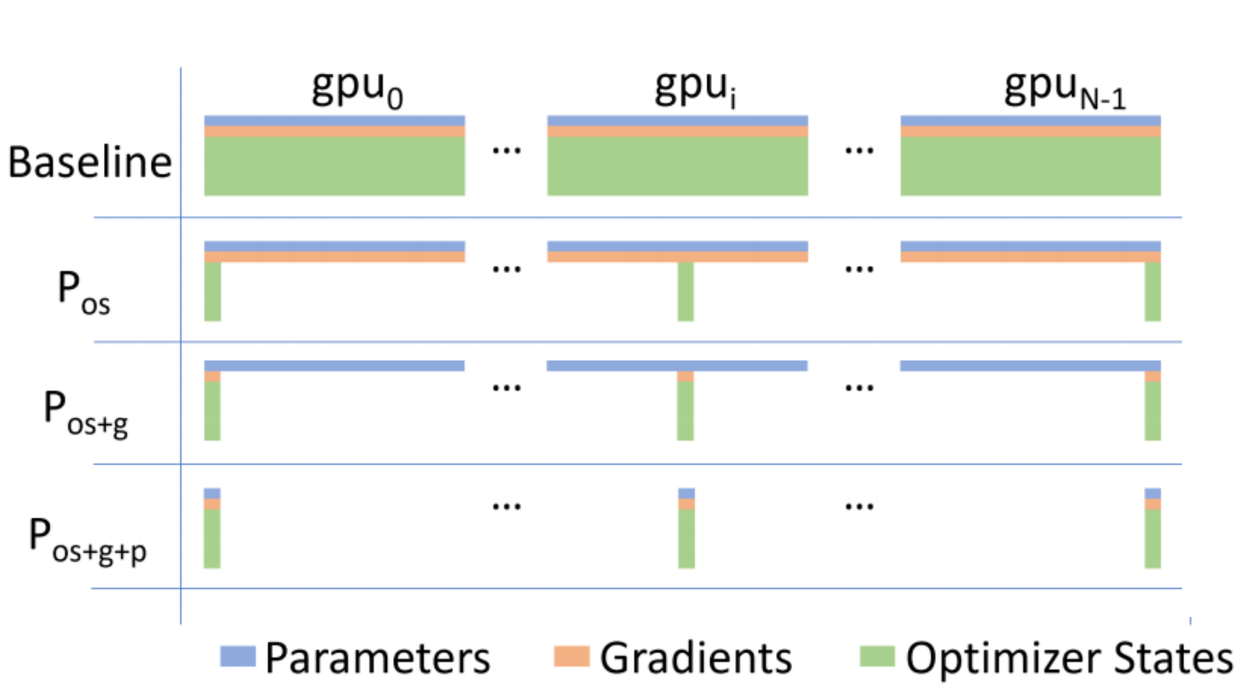
https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

deepspeed

# 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



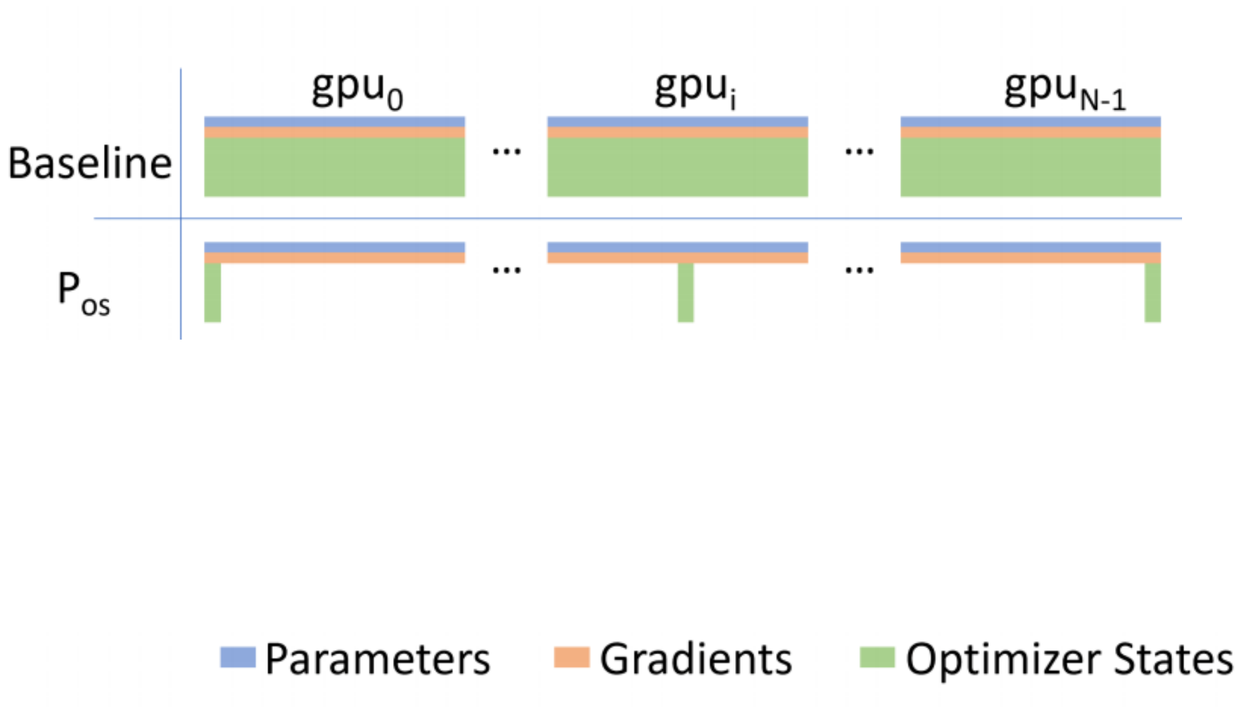


NVIDIA DEEP LEARNING INSTITUTE

deepspeed

# SHARDED DATA PARALLELISM

## ZeRO: Stage 1



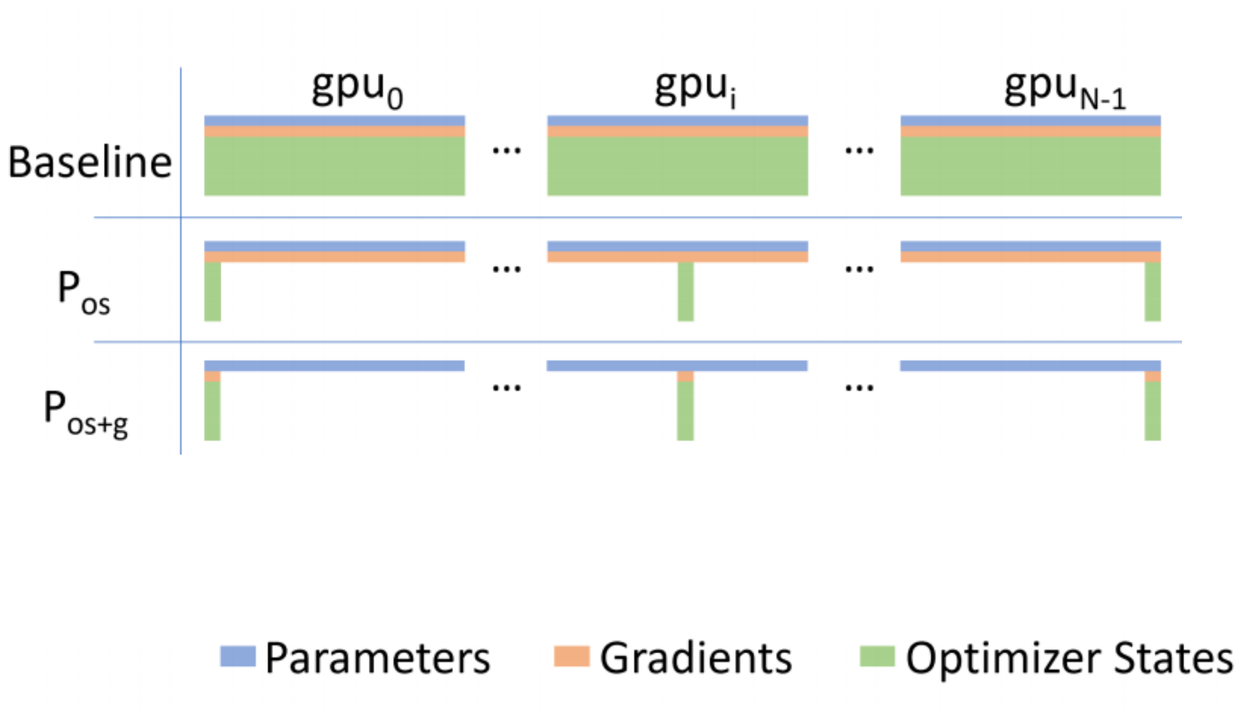


NVIDIA DEEP LEARNING INSTITUTE

deepspeed

# SHARDED DATA PARALLELISM

## ZeRO: Stage 2



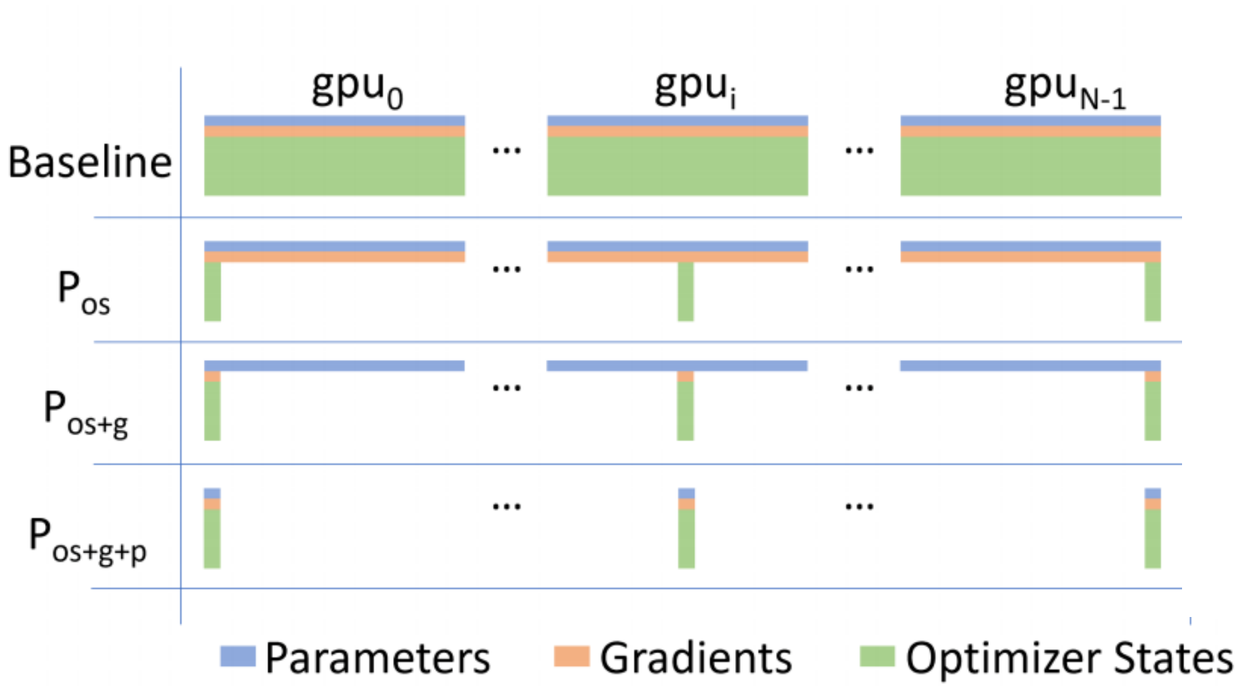


NVIDIA. DEEP LEARNING INSTITUTE

deepspeed

# SHARDED DATA PARALLELISM

## ZeRO: Stage 3





NVIDIA DEEP LEARNING INSTITUTE

# DOES IT ALLOW TO SCALE INDEFINITELY?

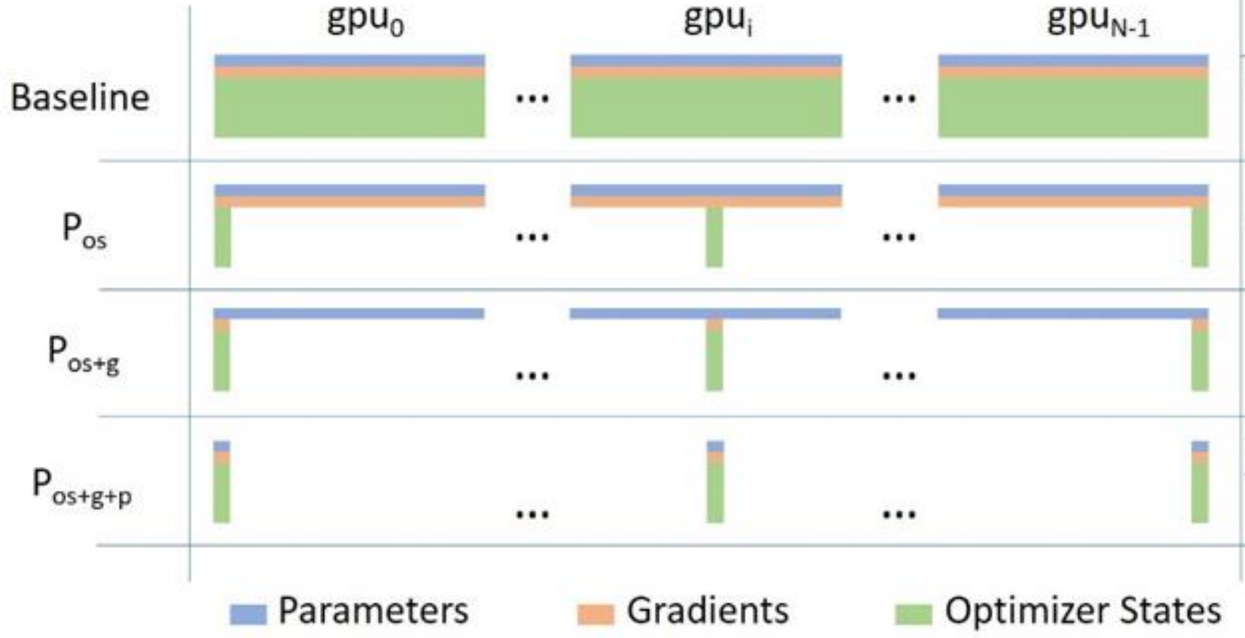
$$\frac{(2 + 2 + K) * \Psi}{N_d}$$

# IT RELIES ON BATCH SIZE INCREASE

$$\frac{(2+2+K)*\Psi}{N_d}$$

# SHARDED DATA PARALLELISM

## Communication overheads




| | Memory Consumption | | Comm Volume |
|---|---|---|---|
| | Formulation | Specific Example K=12 $\Psi$=7.5B $N_d$=64 | |
| Baseline | $(2+2+K)*\Psi$ | 120GB | 1x |
| $P_{os}$ | $2\Psi+2\Psi+\frac{K*\Psi}{N_d}$ | 31.4GB | 1x |
| $P_{os+g}$ | $2\Psi+\frac{(2+K)*\Psi}{N_d}$ | 16.6GB | 1x |
| $P_{os+g+p}$ | $\frac{(2+2+K)*\Psi}{N_d}$ | 1.9GB | 1.5x |

# SHARDED DATA PARALLELISM

## GPT2 example: Enable ZeRO optimization

```
# Construct FP16, distributed, GPT2 model
model = GPT2Model(num_layers=args.num_layers, ...)
model = FP16_Module(model)
model = DistributedDataParallel(model, ...)

...

# Construct FP16 Adam optimizer
optimizer = Adam(param_groups, ...)
optimizer = FP16_Optimizer(optimizer, ...)
```

```
# Forward pass
output = model(tokens, ...)

# Backward pass
optimizer.backward(loss)

# Parameter update
optimizer.step()
```

```
# Construct GPT2 model
model = GPT2Model(num_layers=args.num_layers, ...)

# Construct Adam optimizer
optimizer = Adam(param_groups, ...)

# Wrap model, optimizer, and lr scheduler
model, optimizer, lr_scheduler, _ = deepspeed.initialize(
    args=args,
    model=model,
    optimizer=optimizer,
    lr_scheduler=lr_scheduler,
    mpu=mpu
)
```

```
# Forward pass
output = model(tokens, ...)

# Backward pass
model.backward(loss)

# Parameter update
model.step()
```

For more details see: https://aka.ms/deepspeed-gpt2

# SHARDED DATA PARALLELISM

deepspeed

## GPT2 example: Enable ZeRO optimization

```
deepspeed pretraining_gpt.py <args> \
        --deepspeed_config ds_config.json
```

```
{
  "train_batch_size": 64,
  "gradient_accumulation_steps": 1,
  "steps_per_print": 1,
  "zero_optimization": {
    "stage": 3,
    "stage3_max_live_parameters": 1e9,
    "stage3_max_reuse_distance": 1e9,
    "stage3_prefetch_bucket_size": 1e7,
    "stage3_param_persitence_threshold": 1e5,
    "reduce_bucket_size": 1e7,
    "contiguous_gradients": true
  },
  "gradient_clipping": 1.0,
  "fp16": {
    "enabled": true,
    "loss_scale": 0,
    "loss_scale_window": 1000,
    "hysteresis": 2,
    "min_loss_scale": 1
  },
  "wall_clock_breakdown": true,
  "zero_allow_untested_optimizer": false
}
```

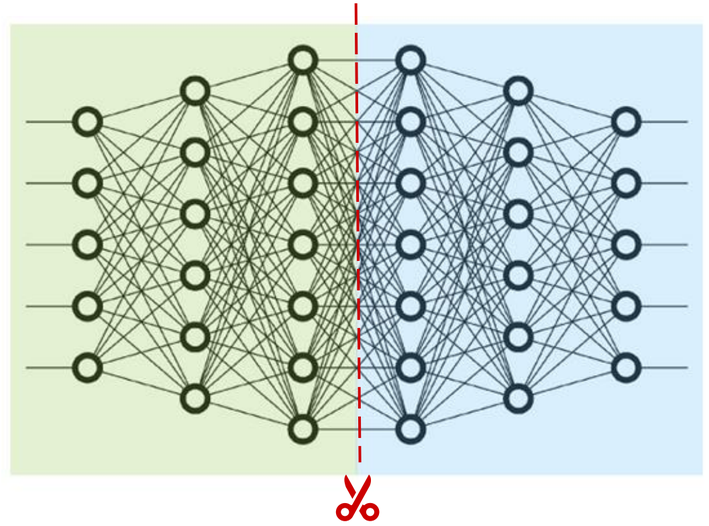
For more details see: https://aka.ms/deepspeed-gpt2

# TENSOR / PIPELINE PARALLELISM

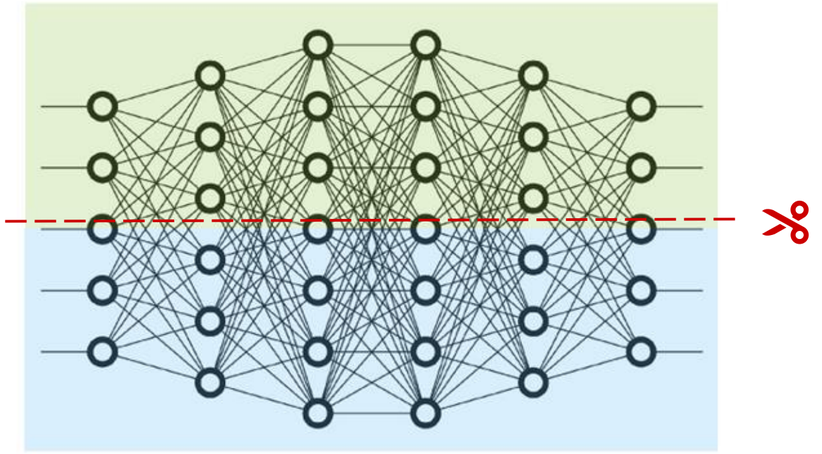
# TECHNOLOGIES THAT ENABLE SCALING LARGE MODELS

## 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*



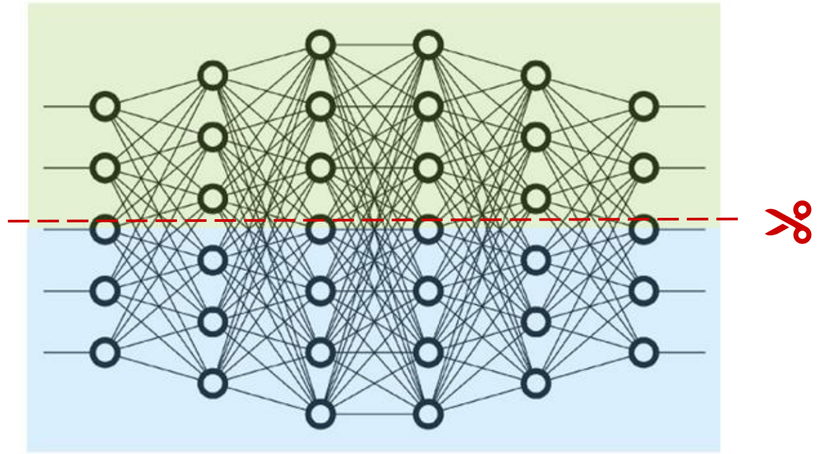
https://developer.nvidia.com/blog/scaling-language-model-training-to-a-trillion-parameters-using-megatron/

# TENSOR PARALLELISM

# 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



# TENSOR PARALLELISM

## Challenges

- Tensor splitting results in lower math intensity
  - This approach is not suitable for strong scaling
- We should make sure math intensity stays above that of the processor
  - A100 math intensity = 312 teraFLOPs/2039 GB/sec = 153
- Tensor parallelism requires more communication
  - NVSwitch mitigates this cost

**Parallel**

Operation: $Y_{n\times(n/p)} = X_{n\times n}A_{n\times(n/p)}$

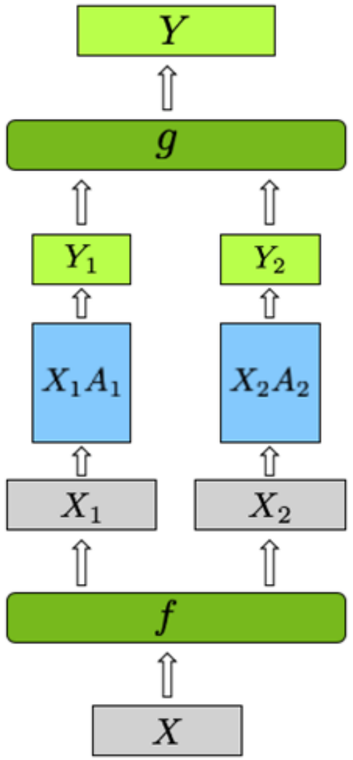
Flops: $2n^3/p$

Bandwidth: $2n^2(1+2/p)$

Intensity: $\frac{1}{2+p}n$

# SIMPLE EXAMPLE OF TENSOR PARALLELISM

## Row Parallel Linear Layer

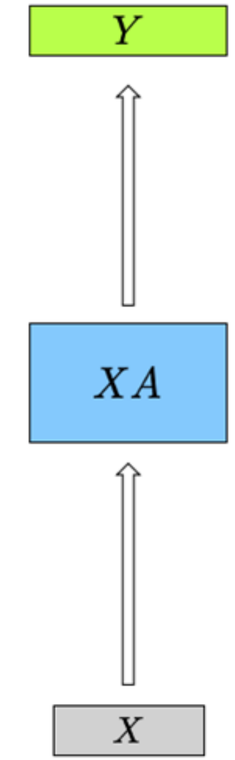



$g$ : forward: $Y = Y_1 + Y_2$ (all-reduce)
backward: $\frac{\partial L}{\partial Y_i} = \frac{\partial L}{\partial Y}$ (identity)

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$$

$$X = [X_1, X_2]$$

$f$ : forward: $X_i$ (split)
backward: $\frac{\partial L}{\partial X} = [\frac{\partial L}{\partial X_1}, \frac{\partial L}{\partial X_2}]$ (all-gather)

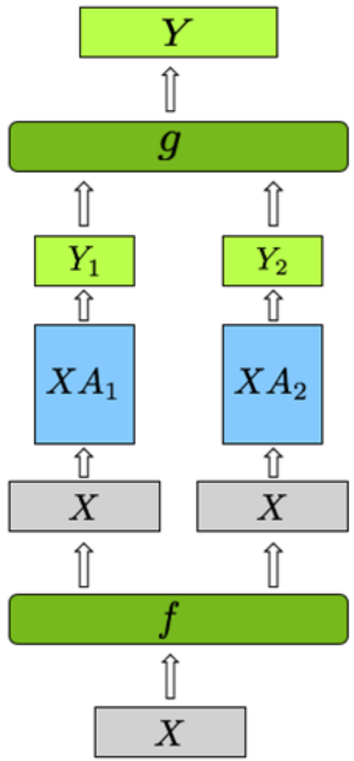



## Column Parallel Linear Layer

$g$ : forward: $Y = [Y_1, Y_2]$ (all-gather)
backward: $\frac{\partial L}{\partial Y_i}$ (split)

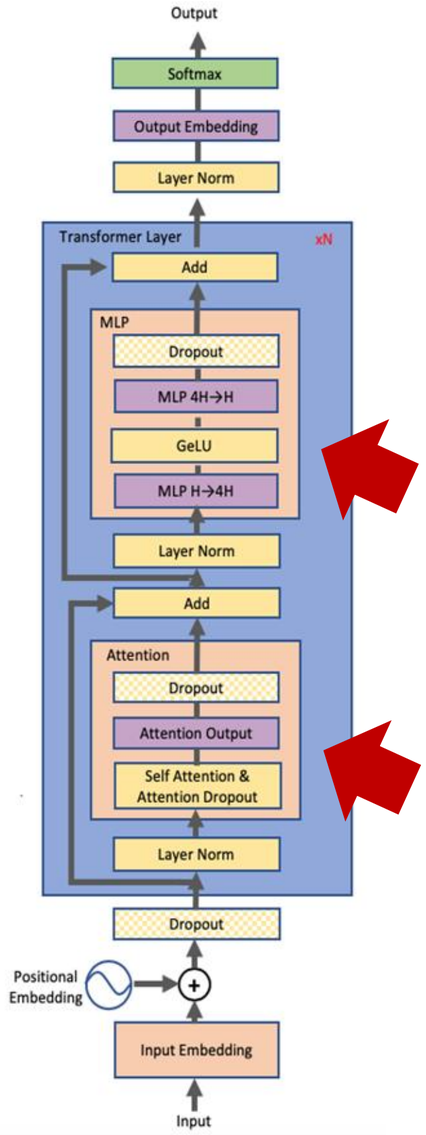
$$A = [A_1, A_2]$$

$f$ : forward: $X$ (identity)
backward: $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial X}|_1 + \frac{\partial L}{\partial X}|_2$ (all-reduce)



# TRANSFORMERS CELL EXAMPLE

# TRANSFORMERS CELL



# MLP TENSOR PARTITIONING

Focus on the GeLU operation:

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

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

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

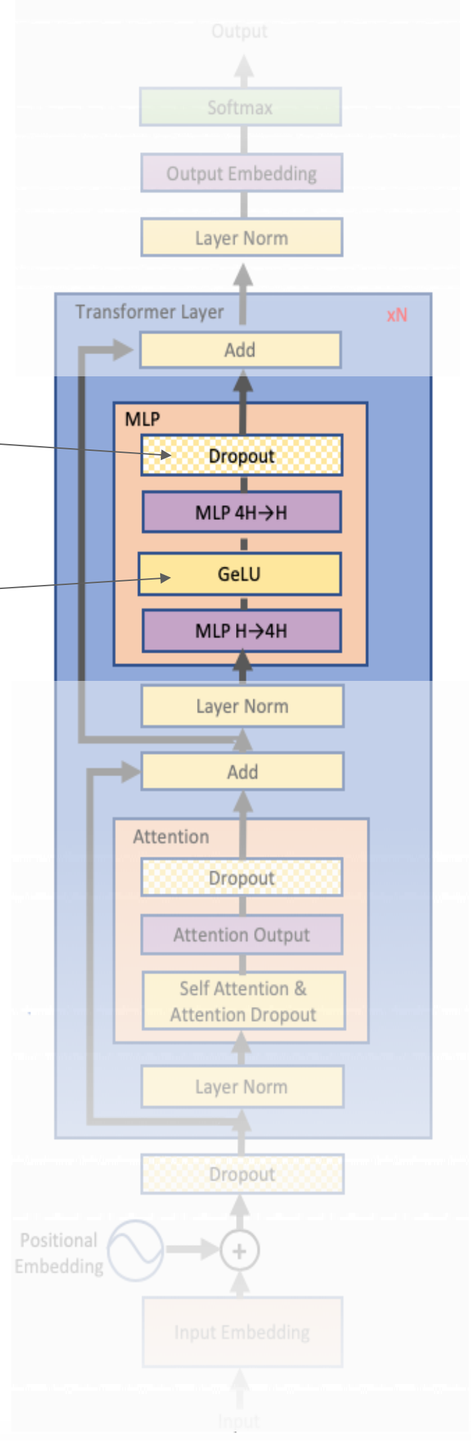
$$X = [X_1, X_2] \quad A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \Rightarrow Y = \text{GeLU}(X_1A_1 + X_2A_2)$$

  - Before GeLU we will need a communication point

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

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

  - No communication is required



# 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:

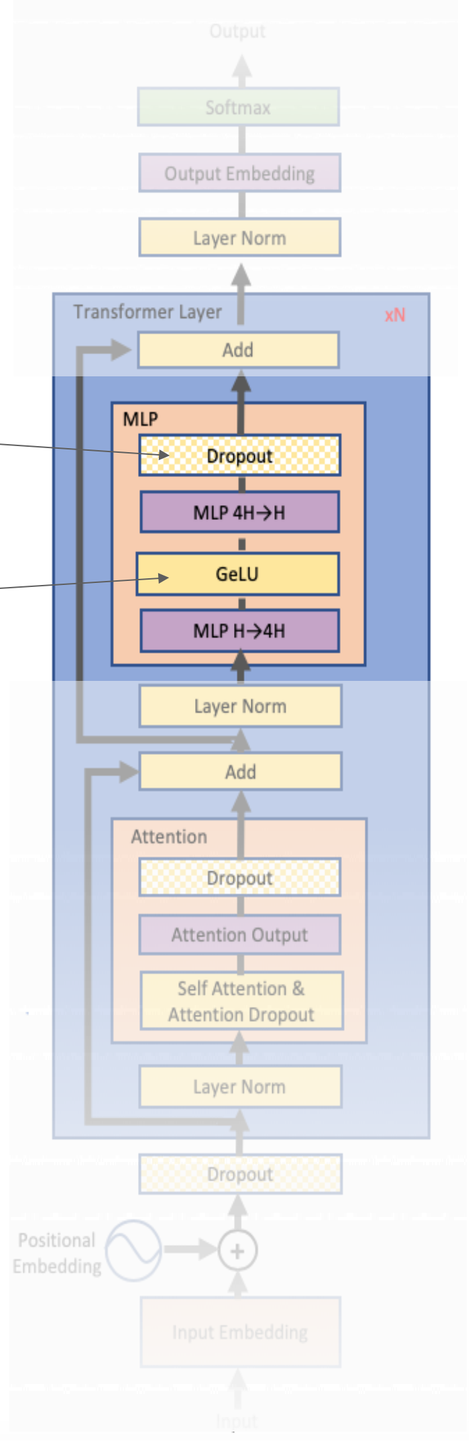
$$A = [A_1, A_2] \Rightarrow [Y_1, Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$$

  - No communication is required

Chosen approach

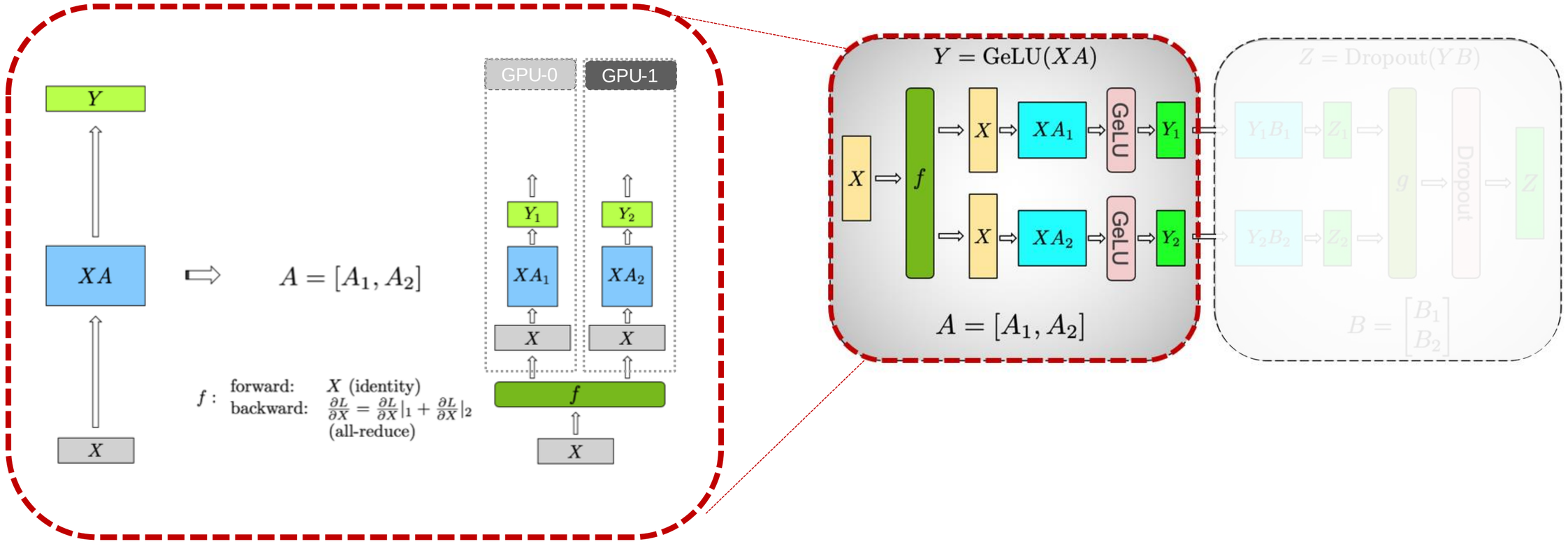
$$Z = \mathrm{Dropout}(YB)$$

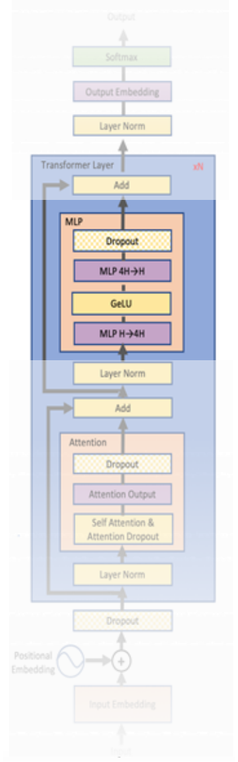
$$Y = \mathrm{GeLU}(XA)$$



# MLP TENSOR PARTITIONING

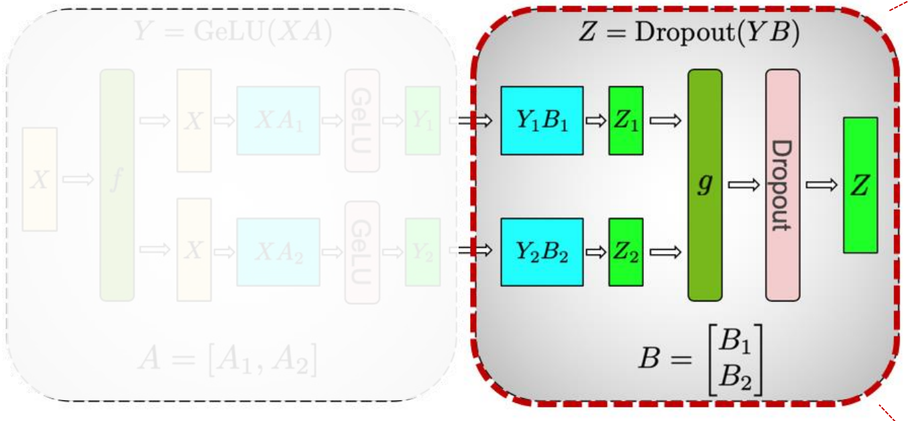
## GeLU Column Parallel



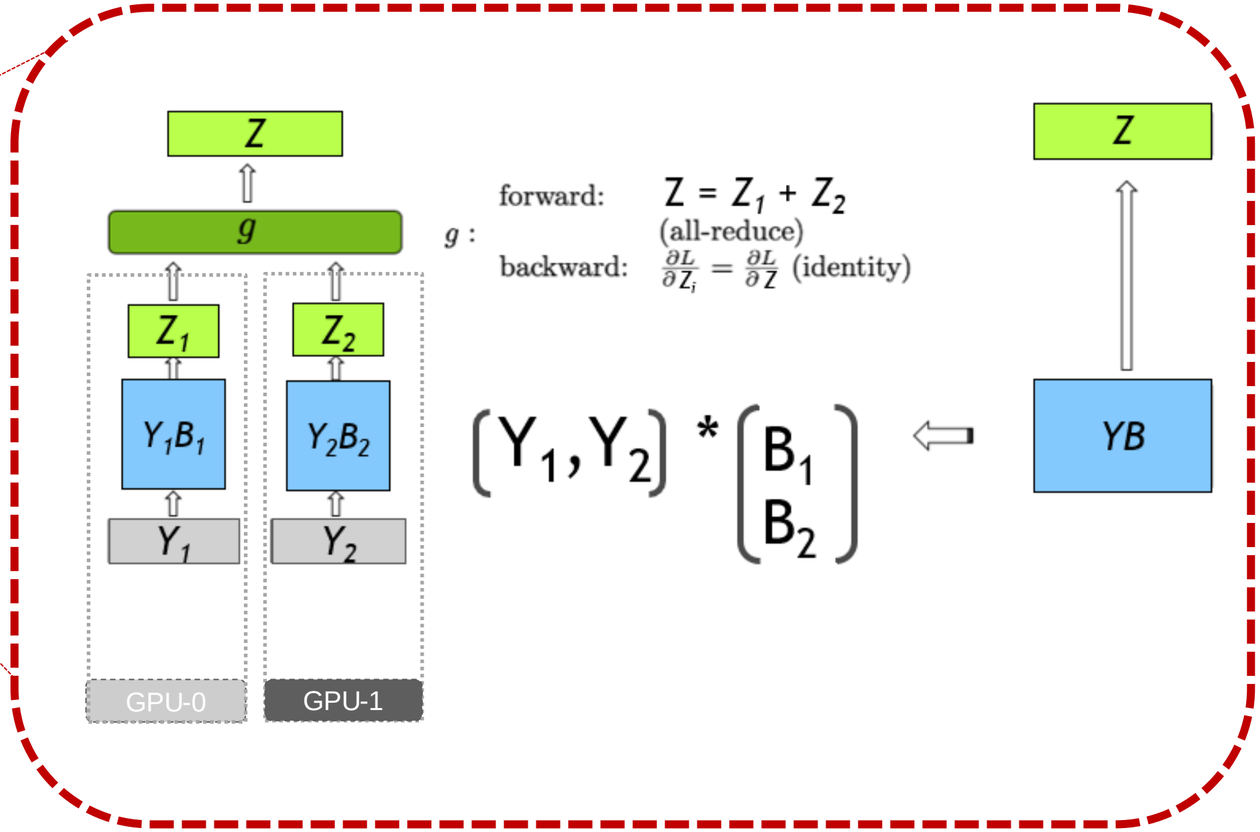
# MLP TENSOR PARTITIONING

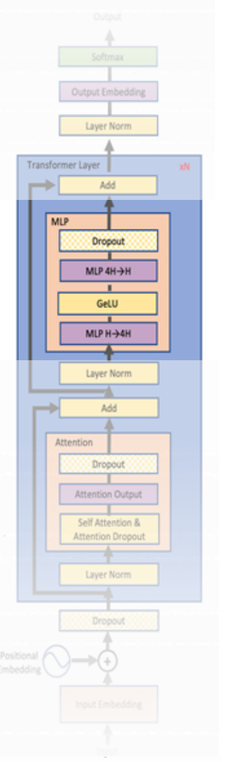
## Dropout Row Parallel





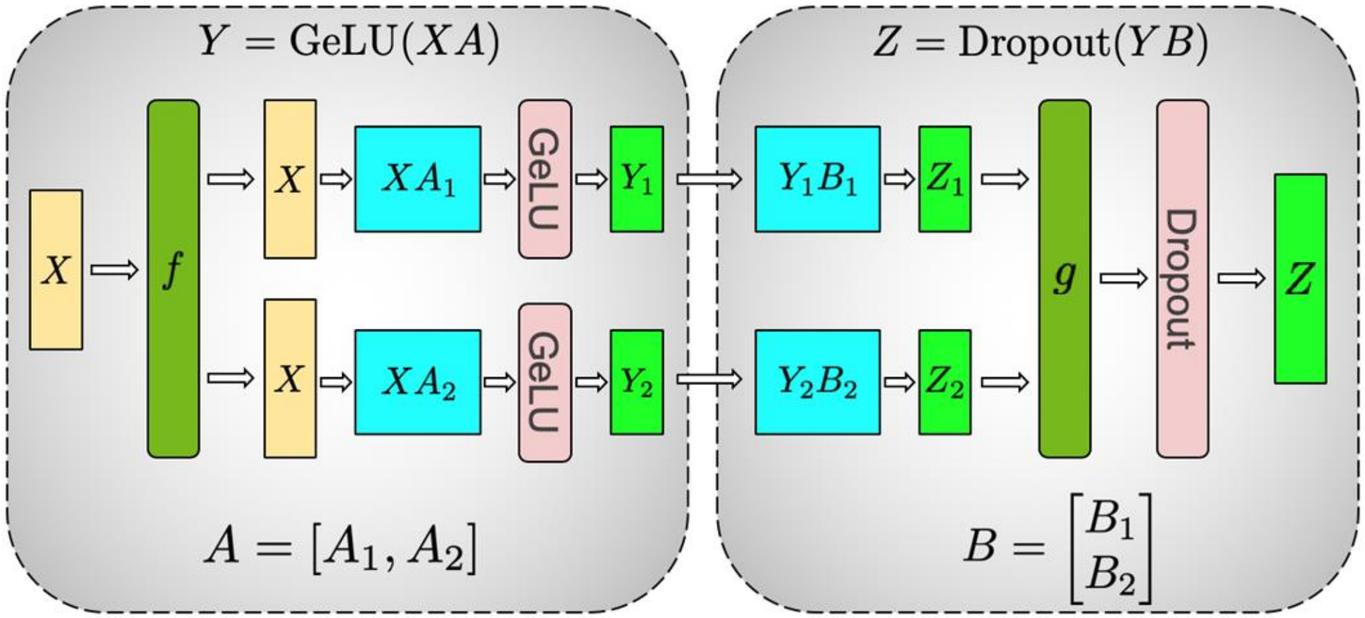


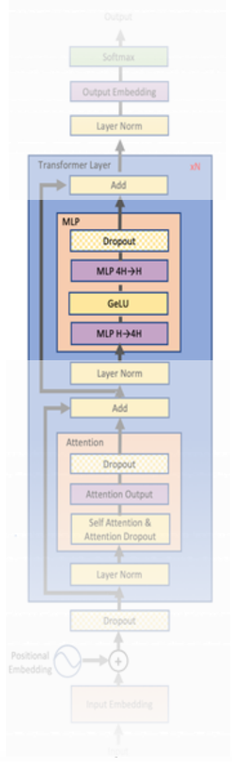


# MLP TENSOR PARTITIONING

## GeLU Column Parallel

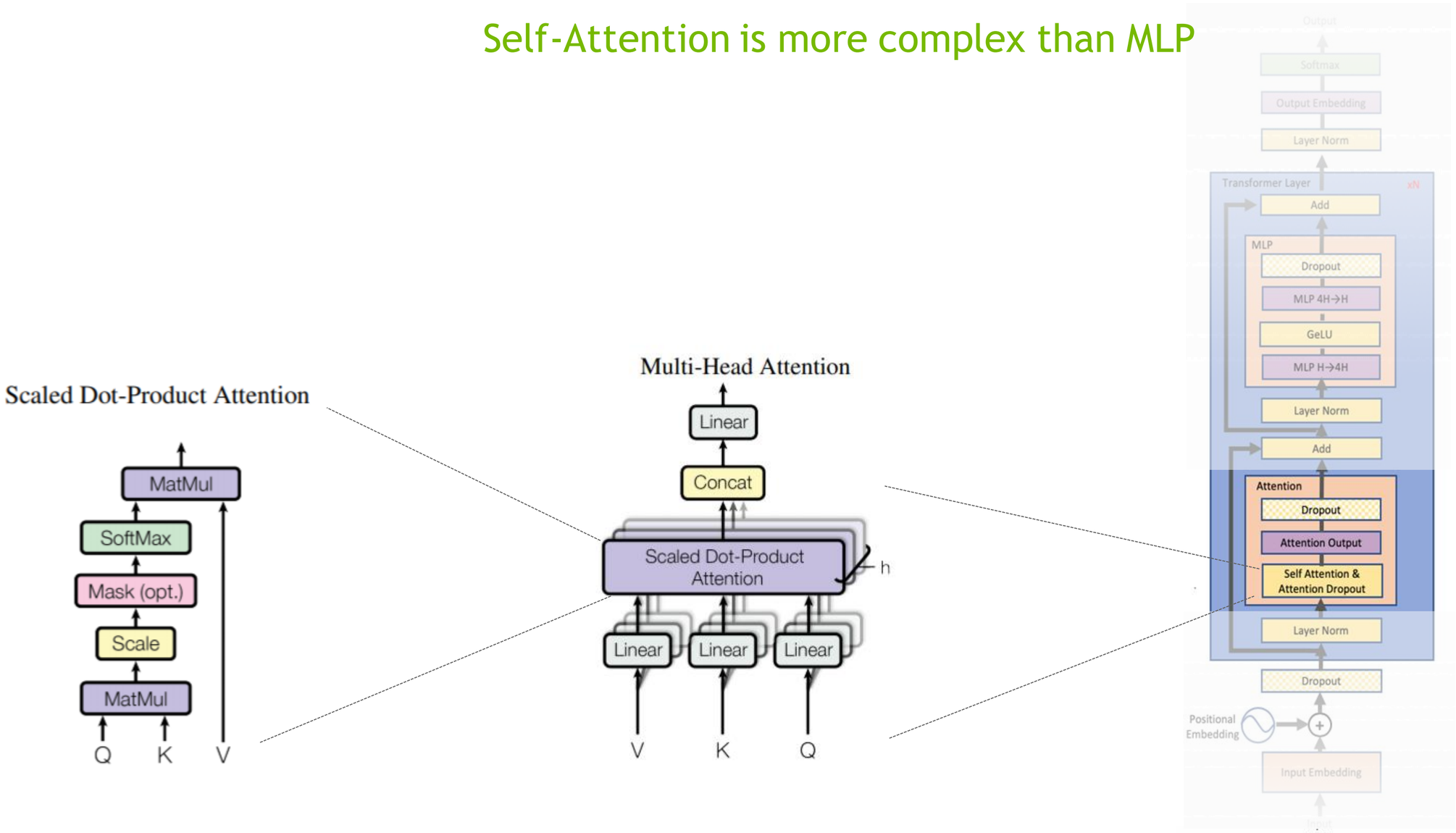



***f*** and ***g*** are **conjugate**, ***f*** is **identity** operator in the forward pass and **all-reduce** in the backward pass while ***g*** is **all-reduce** in forward and **identity** in backward.

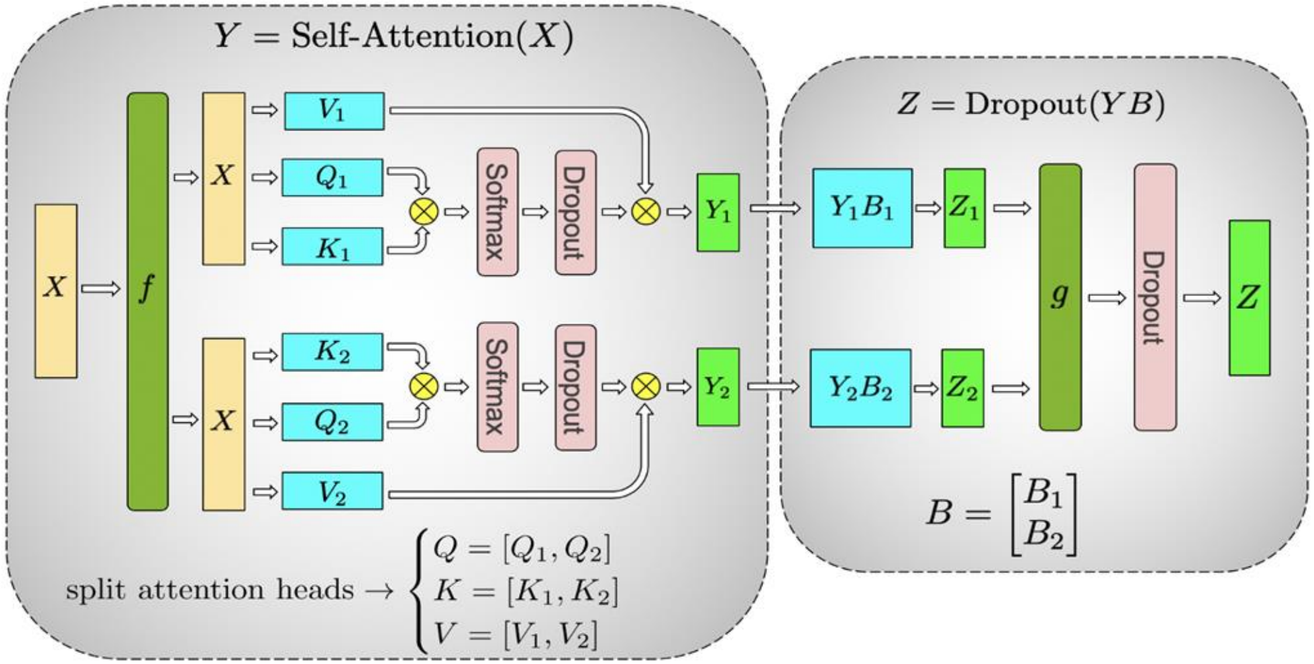
# SELF-ATTENTION TENSOR PARTITIONING

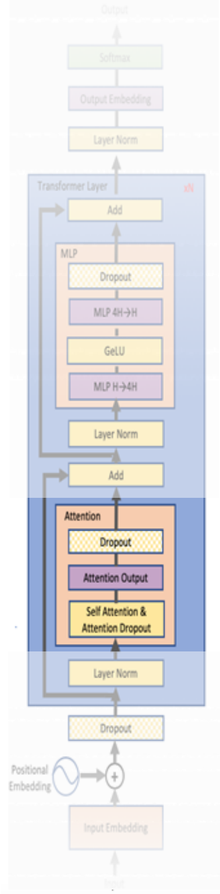
Self-Attention is more complex than MLP



# SELF-ATTENTION TENSOR PARTITIONING

## Same mechanism applies

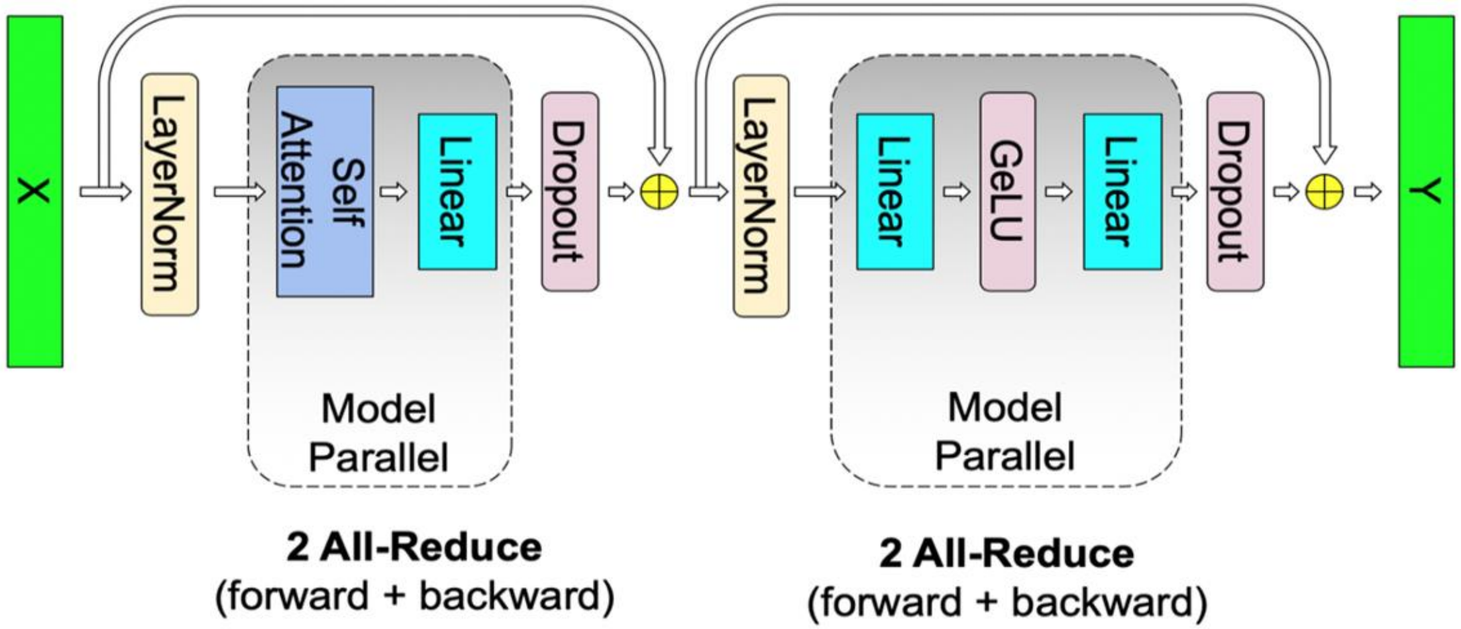



***f*** and ***g*** are **conjugate**, ***f*** is **identity** operator in the forward pass and **all-reduce** in the backward pass while ***g*** is **all-reduce** in forward and **identity** in backward.

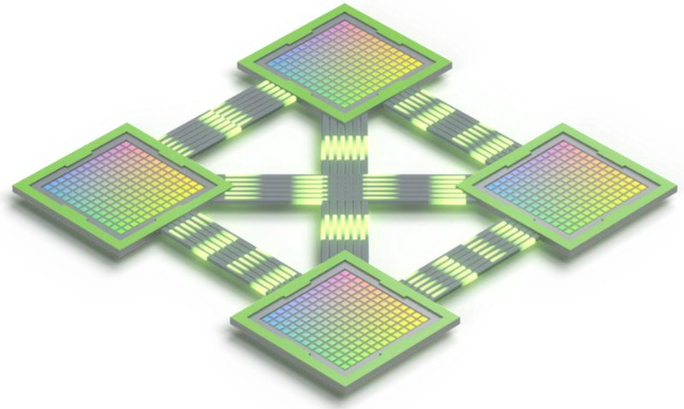
# TENSOR PARALLEL TRANSFORMER LAYER

## All Together



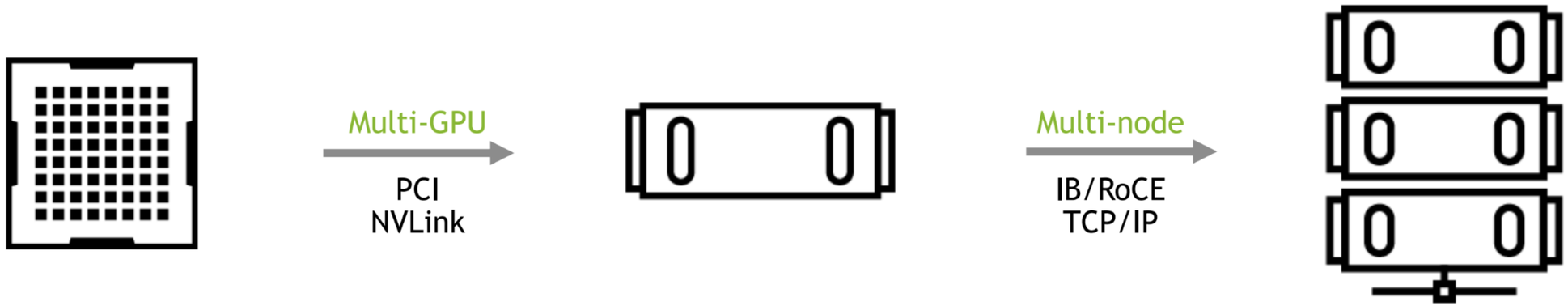
# TENSOR PARALLEL TRANSFORMER LAYER

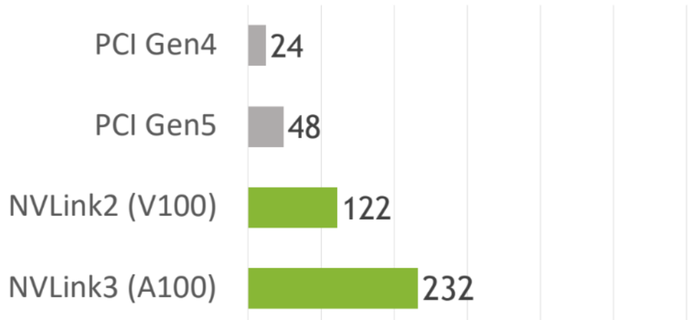
## Communication Expensive

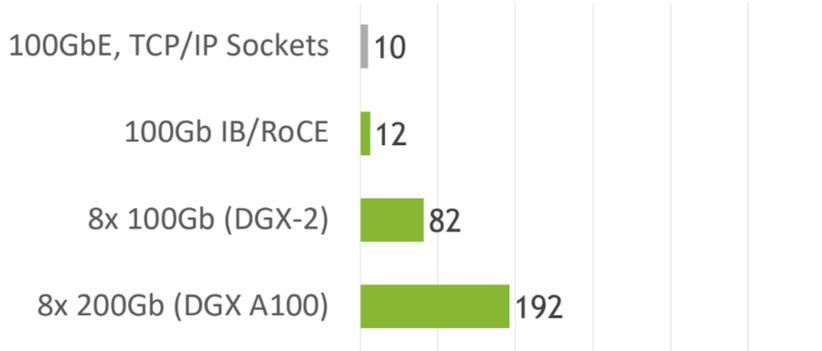


# TENSOR PARALLEL TRANSFORMER LAYER

## Communication Expensive



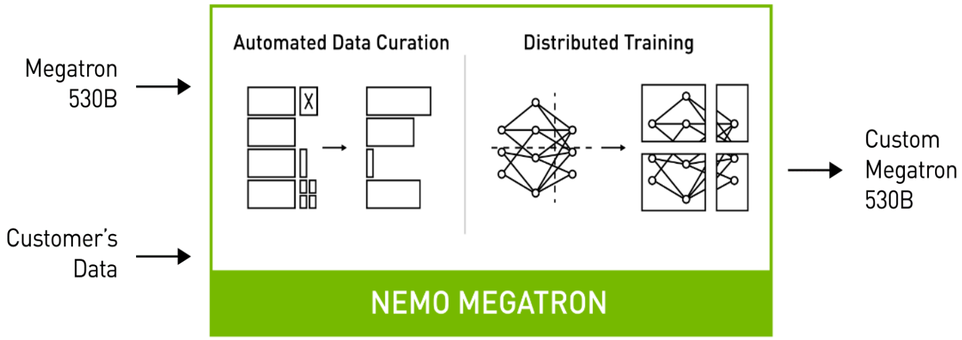

NCCL tests Allreduce Bus Bandwidth in GB/s.

# TENSOR PARALLEL IMPLEMENTATIONS

## Libraries examples

NVIDIA/**Megatron-LM**

Ongoing research training transformer models at scale




deepspeed

Mesh TensorFlow

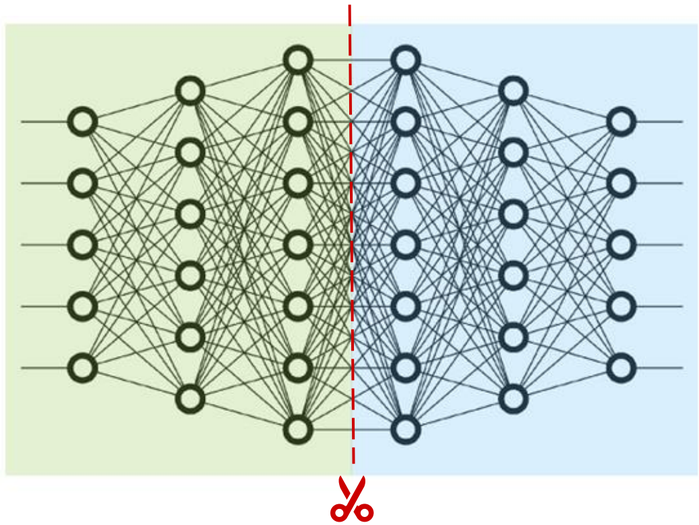
FairScale

Colossal-AI

# PIPELINE PARALLELISM

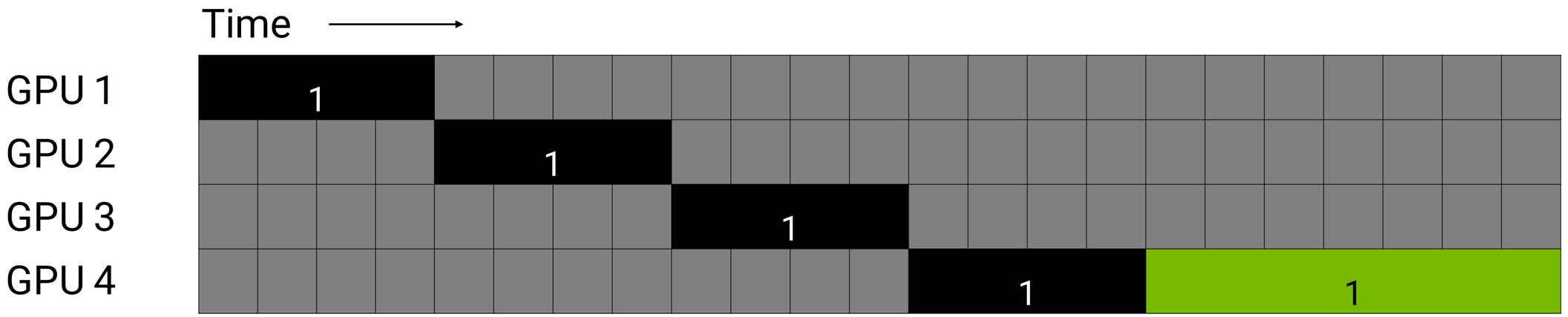
# PIPELINE PARALLELISM

Why?



# PIPELINE PARALLELISM

## Challenges

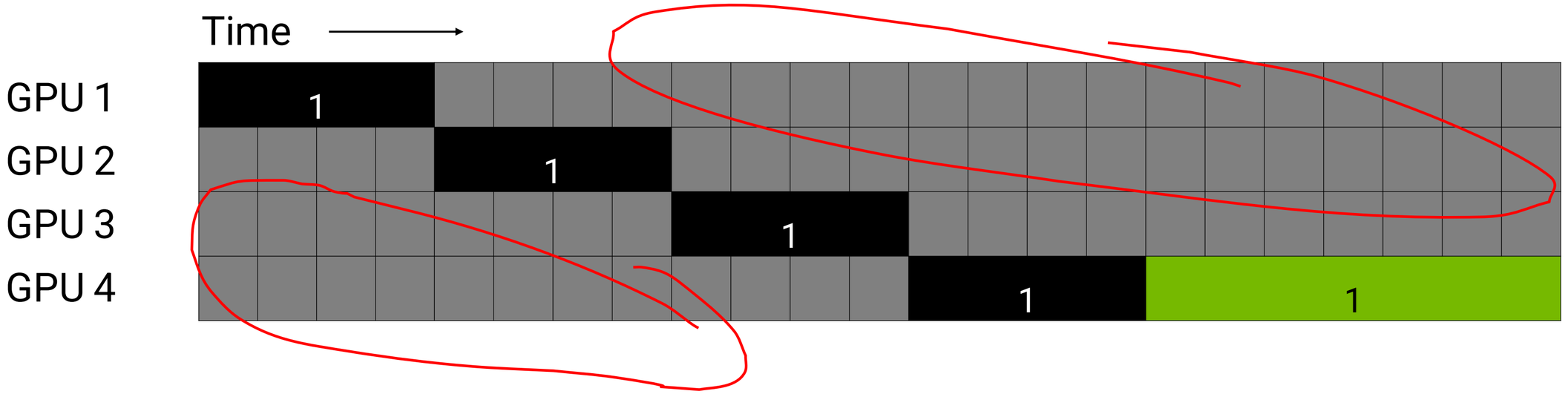



Forward pass

Backward pass

# PIPELINE PARALLELISM

## Challenges – Idle Workers

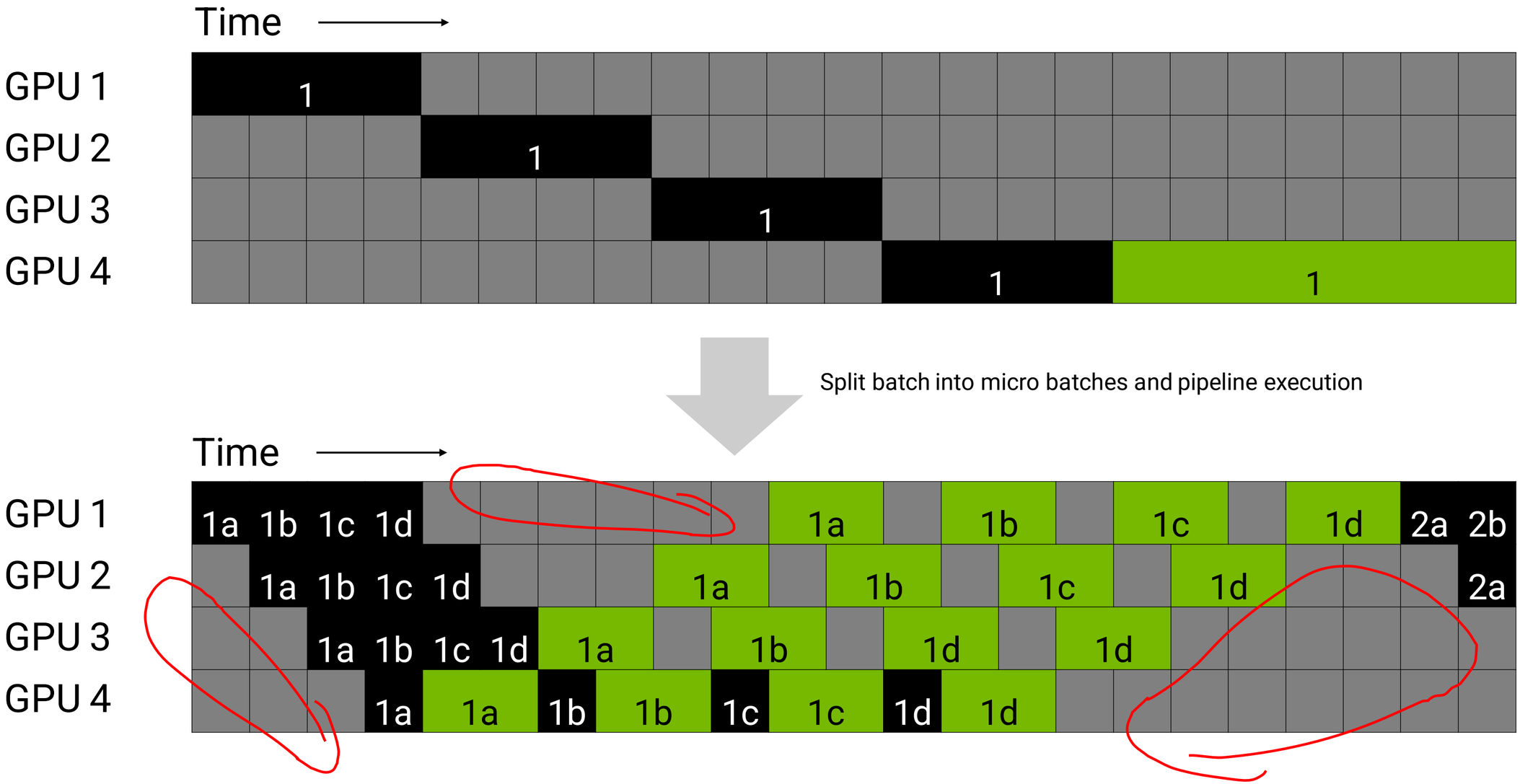



Forward pass

Backward pass

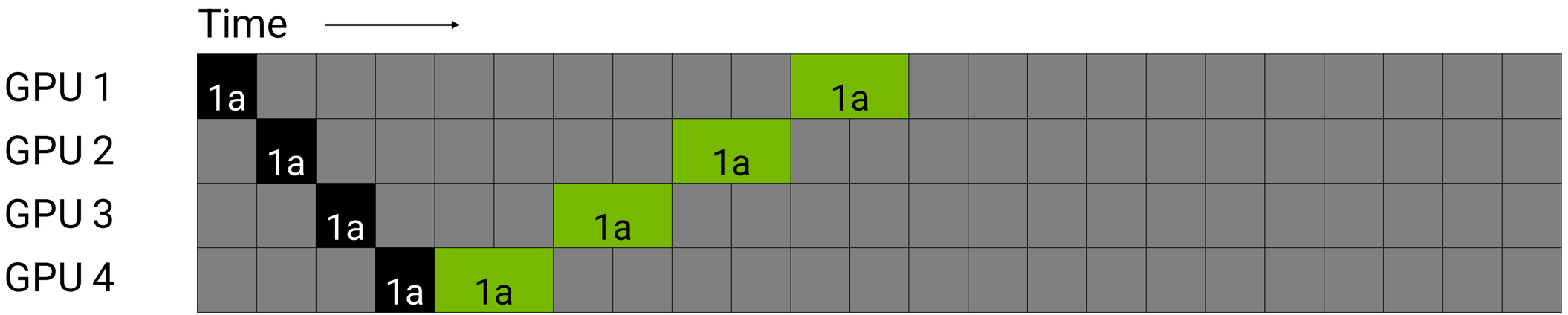
# PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



# PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

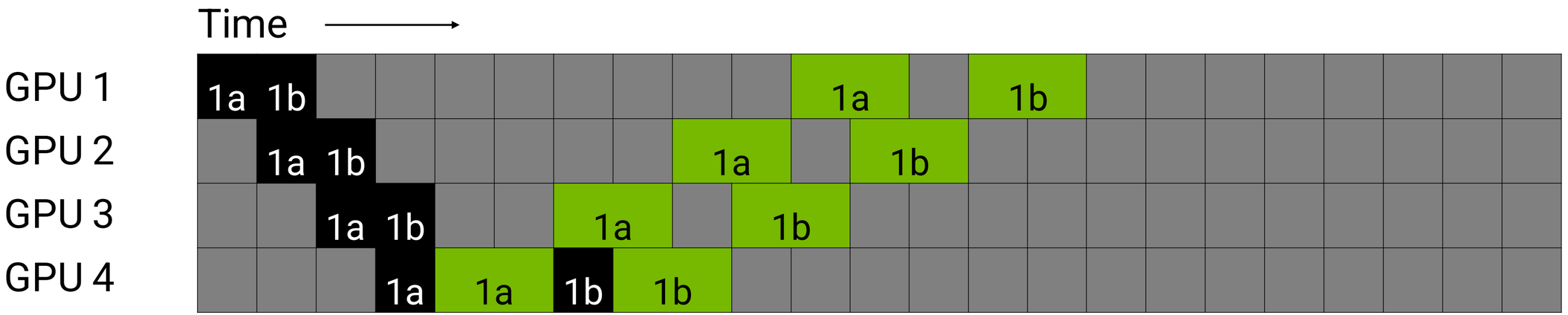



Forward pass

Backward pass

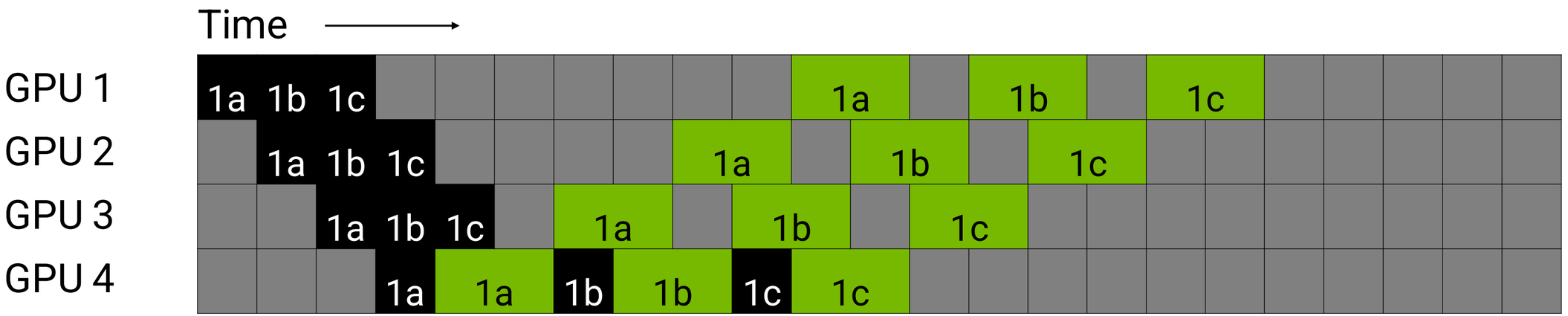
# PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



# PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution

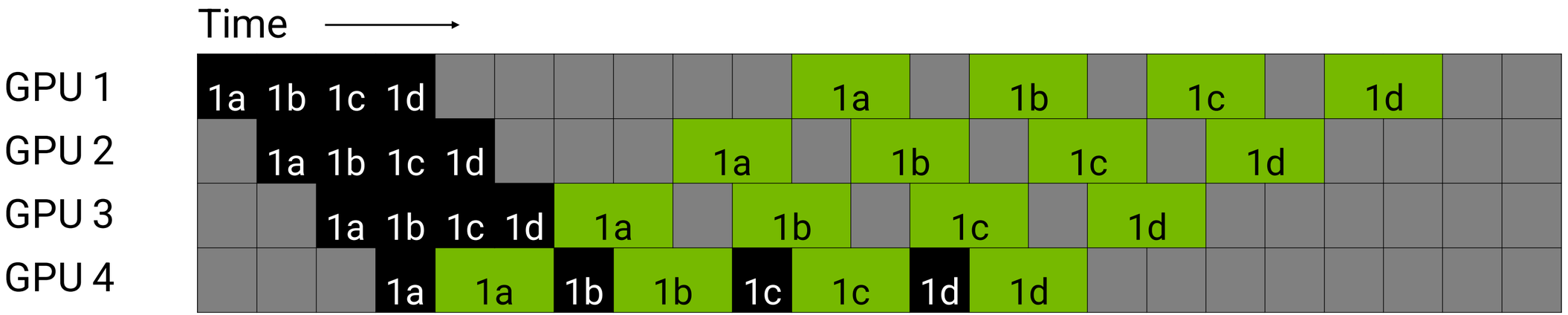



Forward pass

Backward pass

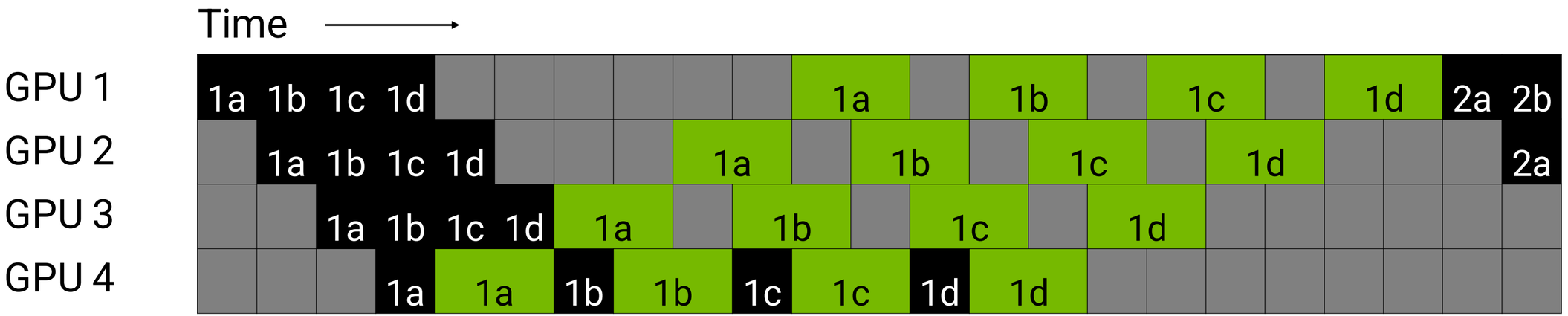
# PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



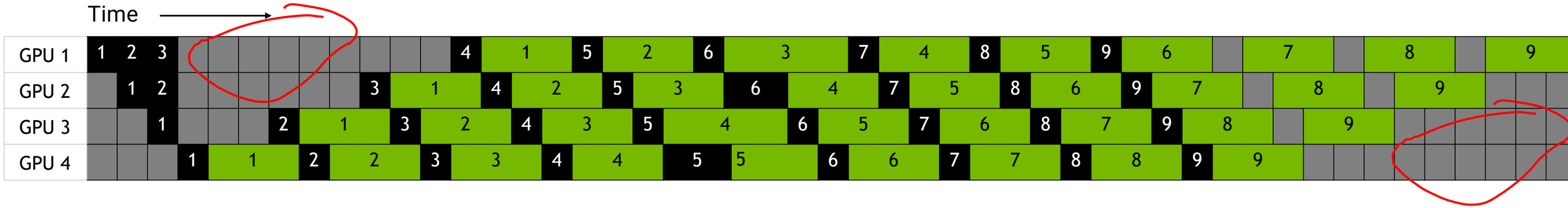
# PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution



# PIPELINE PARALLELISM

Split batch into micro batches and pipeline execution




$$\text{total time} = (m + p - 1) \times (t_f + t_b)$$
$$\text{ideal time} = m \times (t_f + t_b)$$
$$\text{bubble time} = (p - 1) \times (t_f + t_b)$$

$$\text{bubble time overhead} = \frac{\text{bubble time}}{\text{ideal time}} = \frac{p-1}{m}$$

$p$ : number of pipeline stages

$m$ : number of micro batches

$t_f$ : forward step time

$t_b$ : backward step time

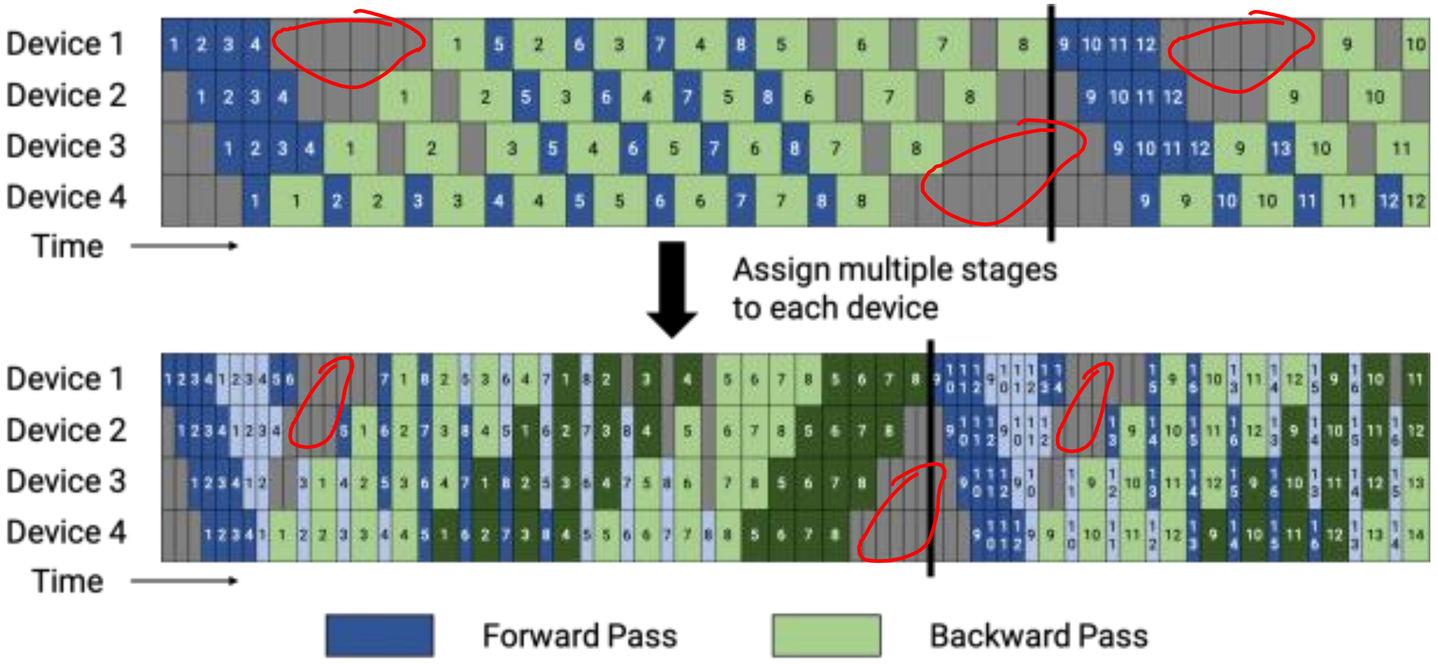
Forward pass

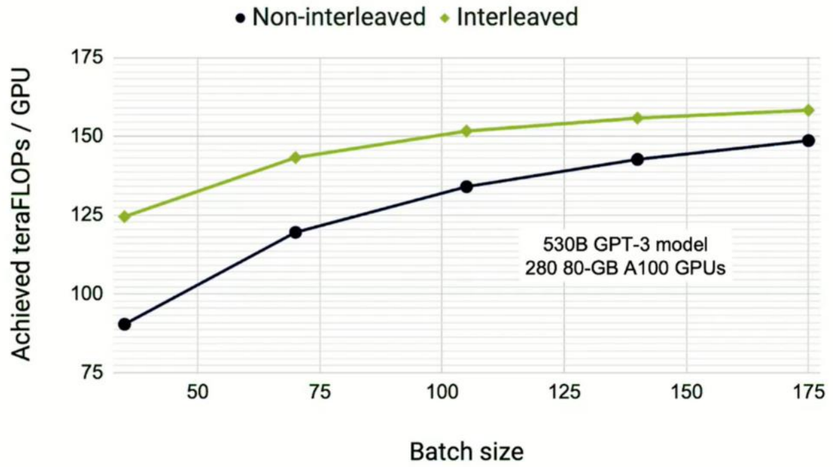
Backward pass

# PIPELINE PARALLELISM

## Interleaved Pipeline

Reduce pipeline Bubble with more communication
Interleaved Pipeline




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

# 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

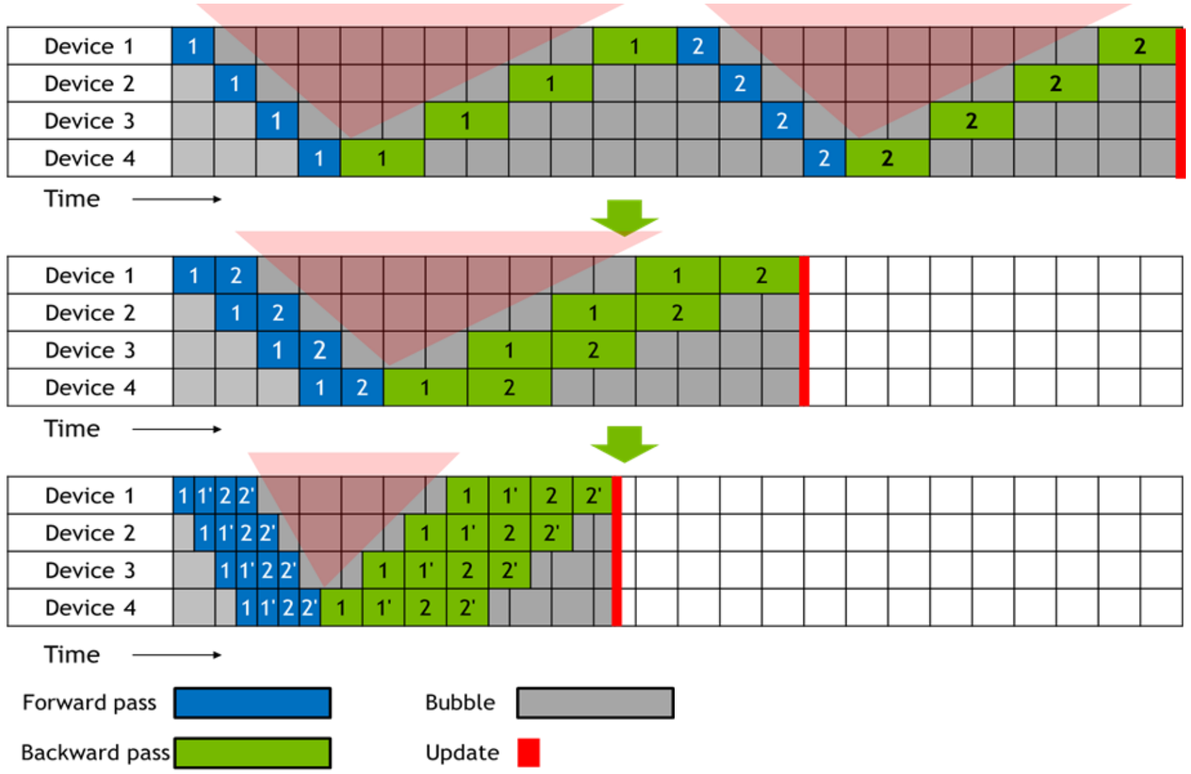
# PIPELINE PARALLELISM

**Value:** Complementary approaches for Training models at scale

**Goal:** Maximizes GPU utilization over InfiniBand

**Capabilities:**

- 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




**Exceptions/Limitations:**

- No Interleave Scheduling for Pipeline parallelism

# TENSOR PARALLELISM

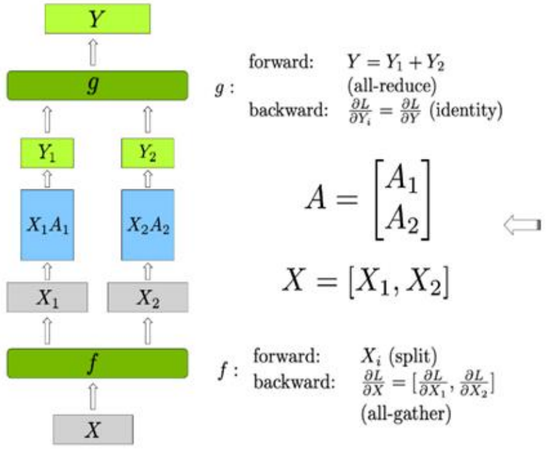
**Value:** Great performance within a server using NVSwitch

**Goal:** Minimizes Latency in single-node

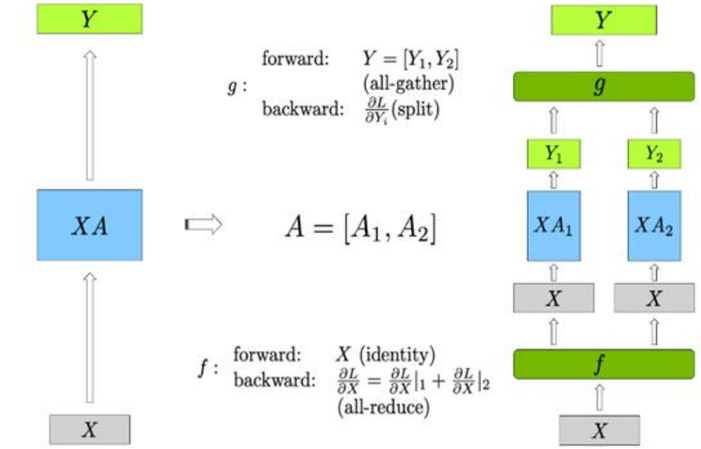
**Capabilities:**

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

## Row Parallel Linear Layer




## Column Parallel Linear Layer



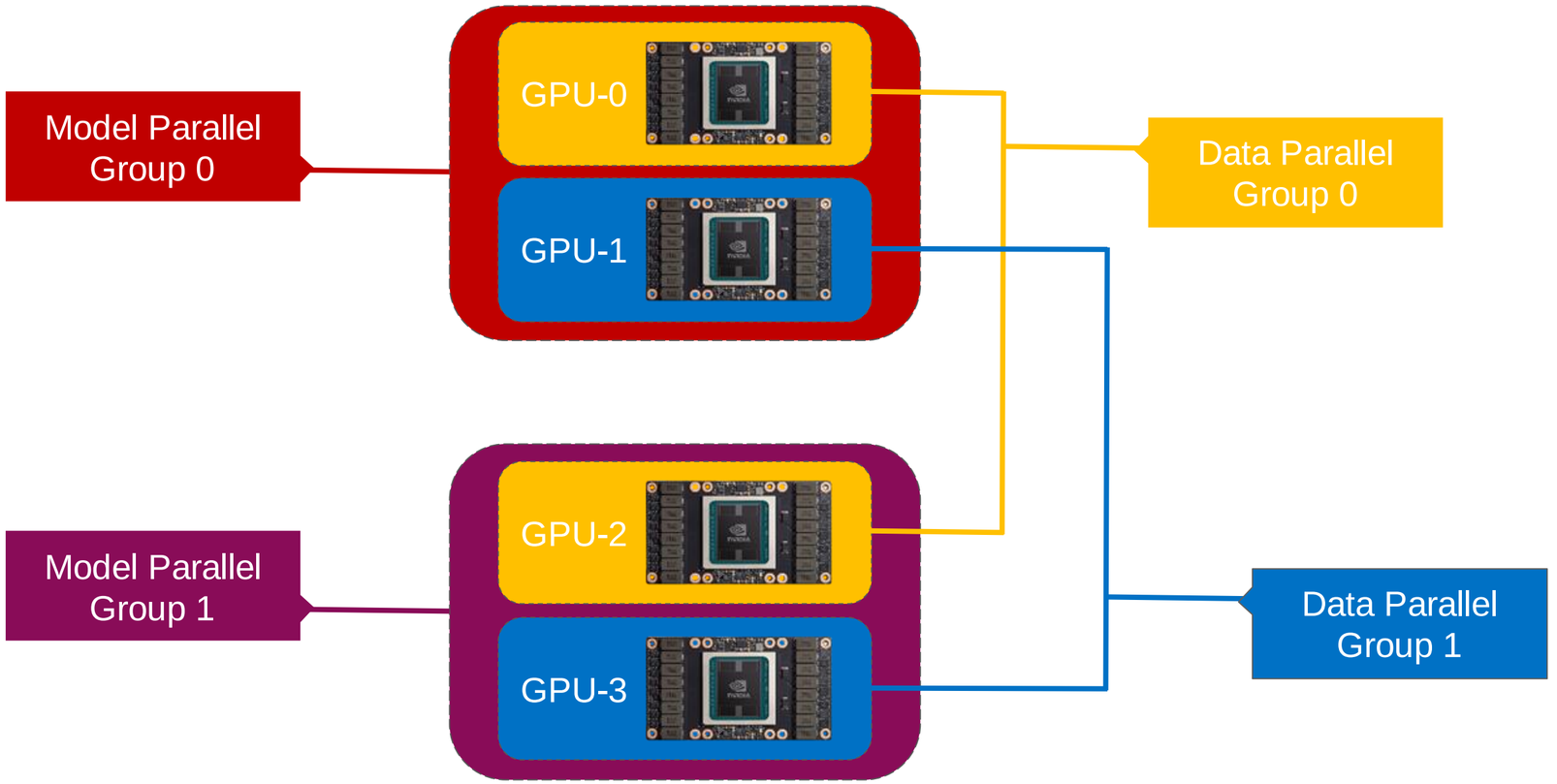

**Exceptions/Limitations:**

- Limited number of Model Architectures | GPT-3 & T5

# HYBRID MODEL PARALLELISM

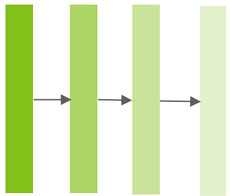
# HYBRID MODEL+DATA PARALLELISM

## Multiple groups of communicators



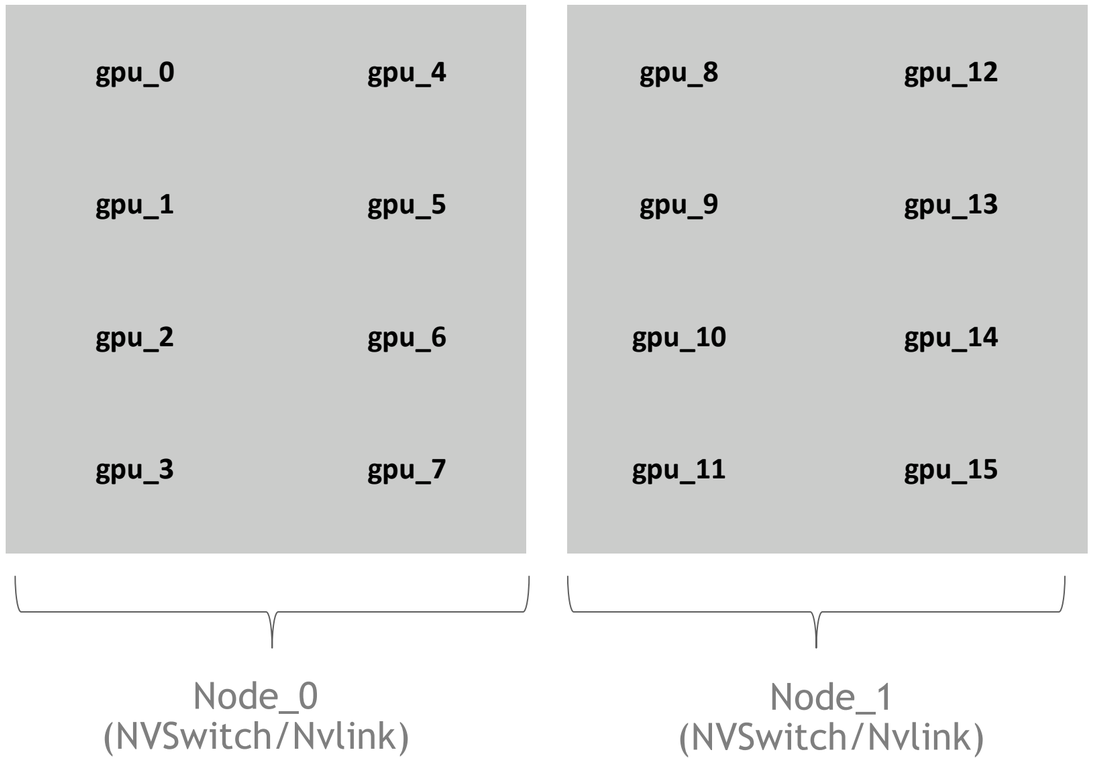
# MEGATRON MODEL PARALLISM

## GPU Affinity Grouping Example



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

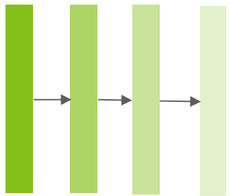
- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2




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

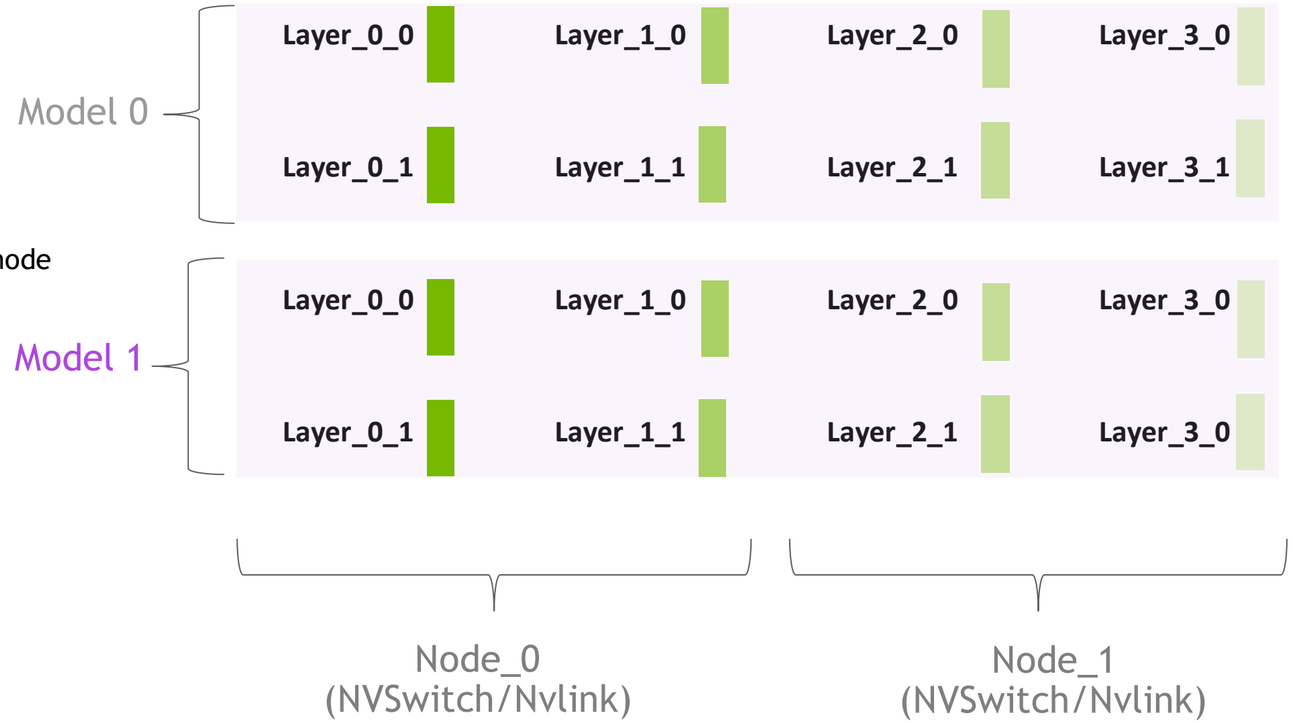
# MEGATRON MODEL PARALLISM

## GPU Affinity Grouping Example



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

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

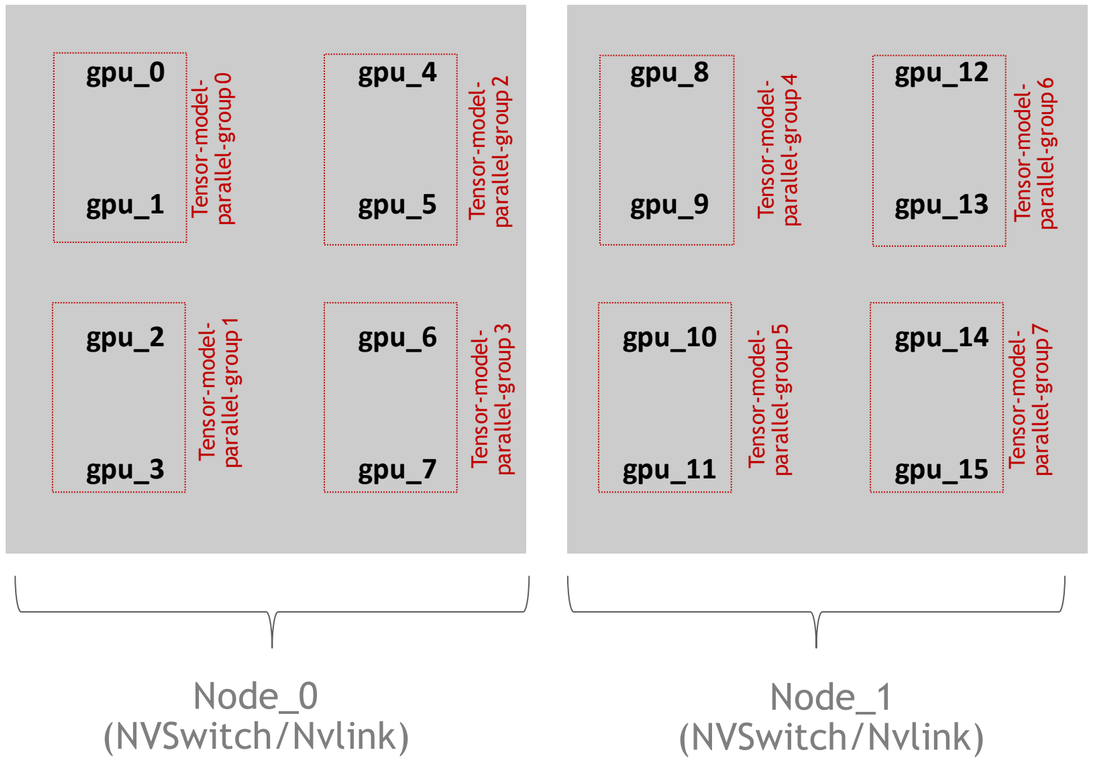



https://github.com/NVIDIA/Megatron-LM

# MEGATRON MODEL PARALLISM

## GPU Affinity Grouping Example

2 nodes , 8 GPUs per node
- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2

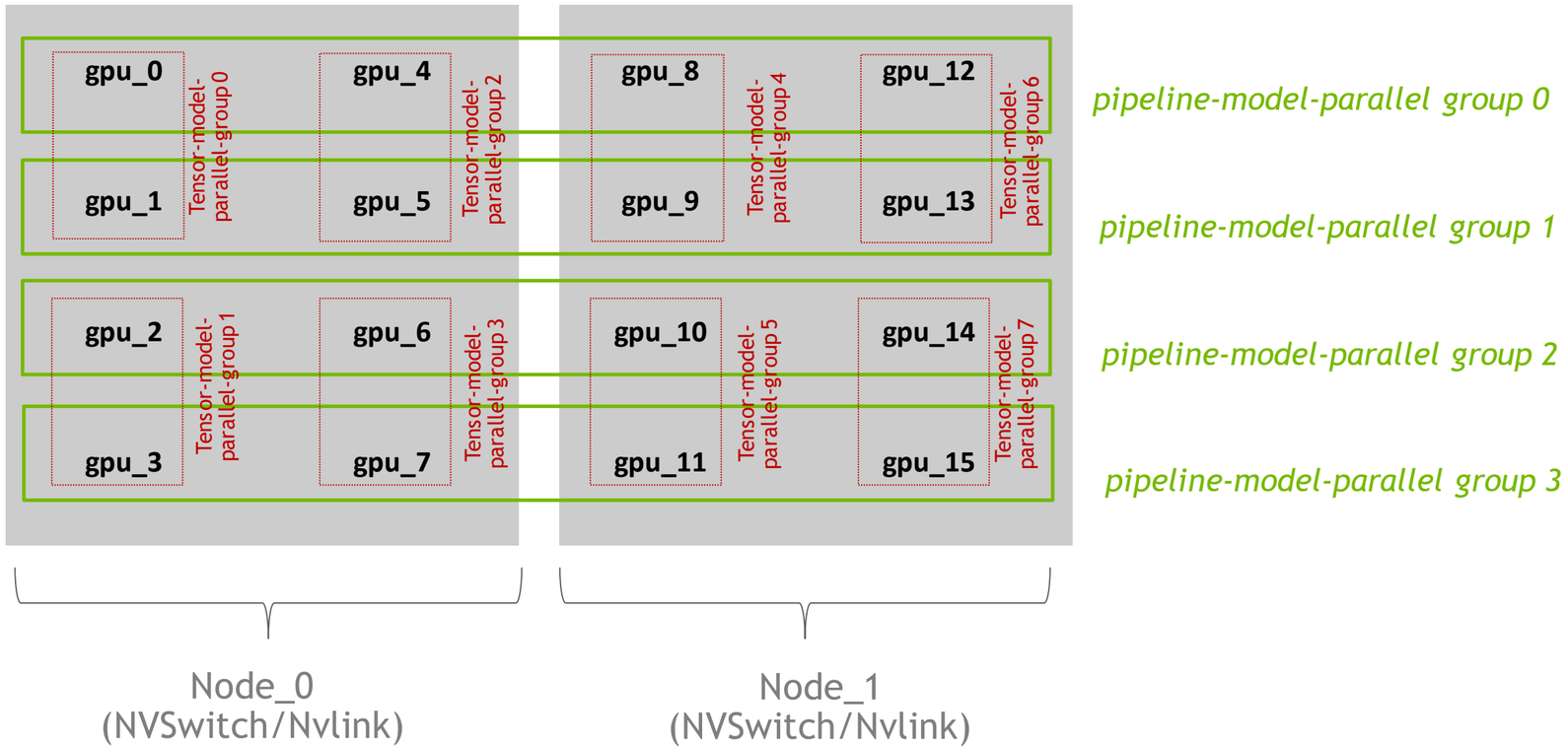



https://github.com/NVIDIA/Megatron-LM

# MEGATRON MODEL PARALLISM

## GPU Affinity Grouping Example

2 nodes , 8 GPUs per node
- Tensor parallel = 2
- Pipeline parallel = 4
- Data parallel = 2

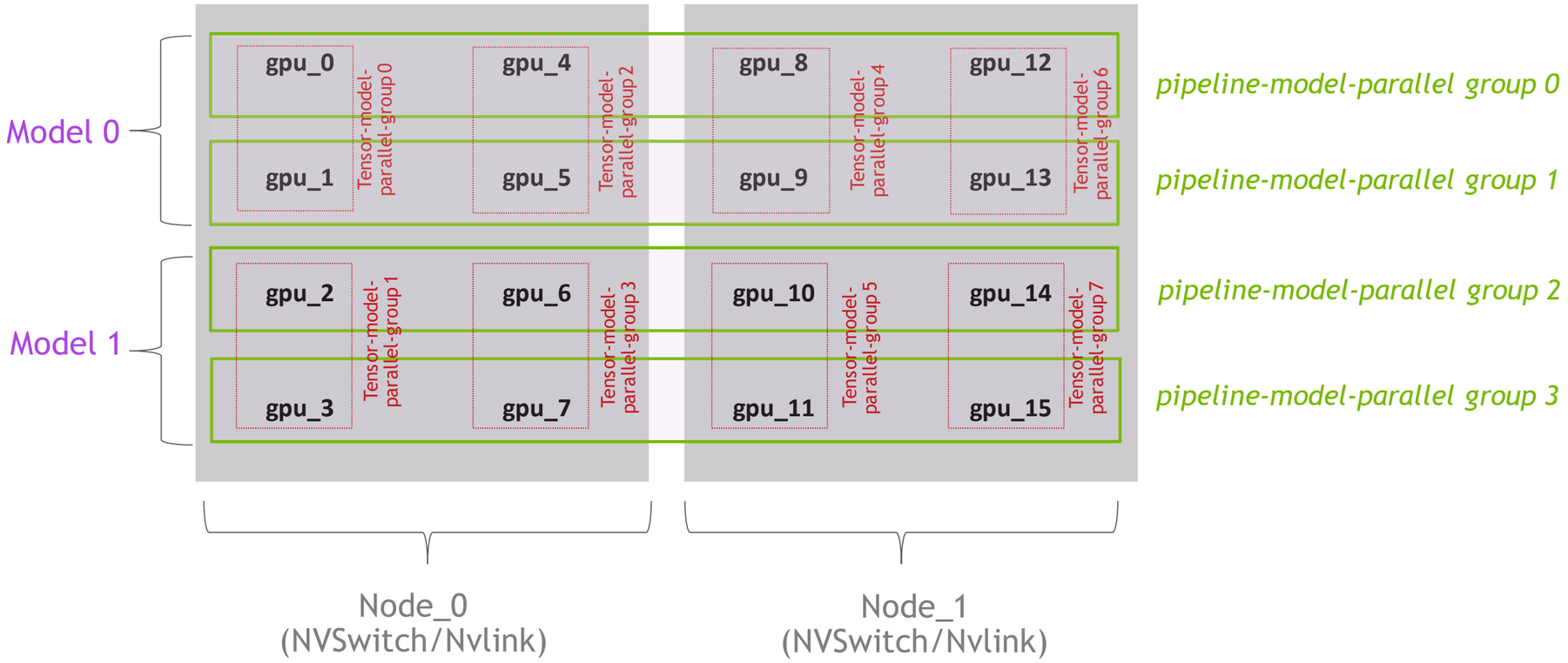



https://github.com/NVIDIA/Megatron-LM

# 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

# SEQUENCE PARALLELISM

# DEALING WITH MEMORY CONSTRAINTS

## Sequence Parallelism

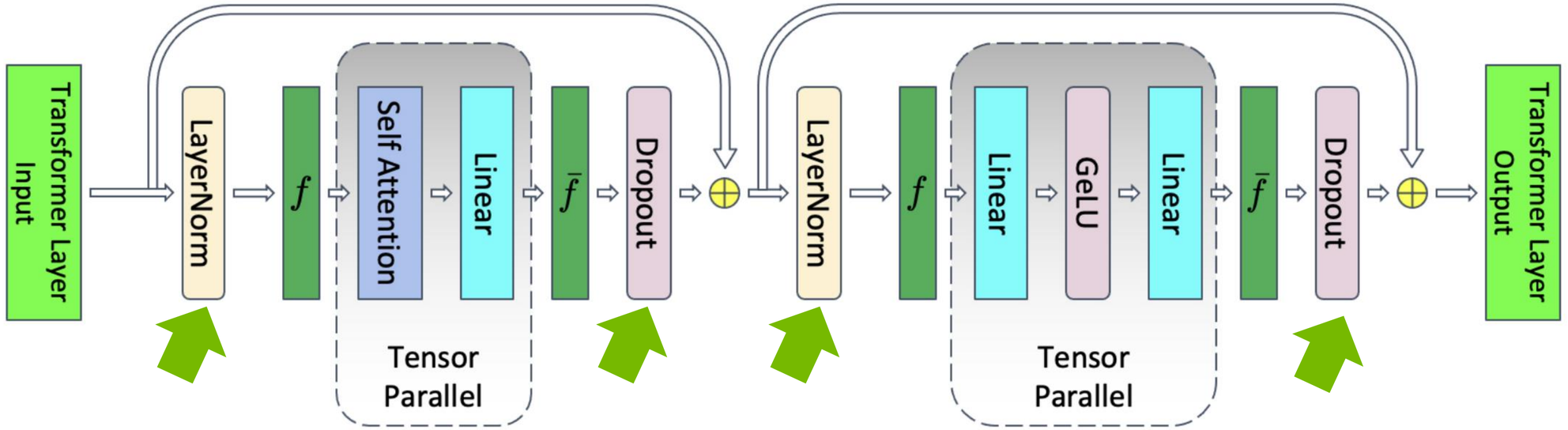



Figure 4: Transformer layer with tensor parallelism. $f$ and $\bar{f}$ are conjugate. $f$ is no operation in the forward pass and all-reduce in the backward pass. $\bar{f}$ is all-reduce in the forward pass and no operation in the backward pass.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# DEALING WITH MEMORY CONSTRAINTS

## Sequence parallelism

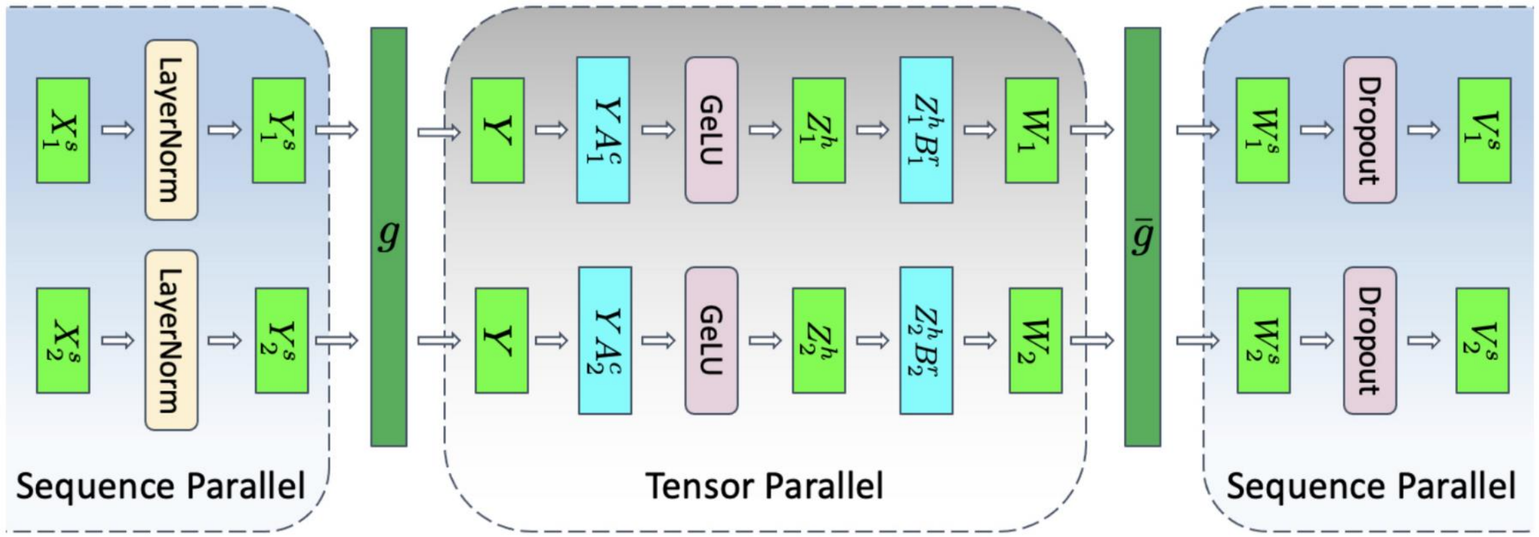



Figure 6: MLP layer with tensor and sequence parallelism. $g$ and $\bar{g}$ are conjugate. $g$ is all-gather in forward pass and reduce-scatter in backward pass. $\bar{g}$ is reduce-scatter in forward pass and all-gather in backward pass.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# DEALING WITH MEMORY CONSTRAINTS

## Megatron Sequence parallelism + Selective Checkpoiting

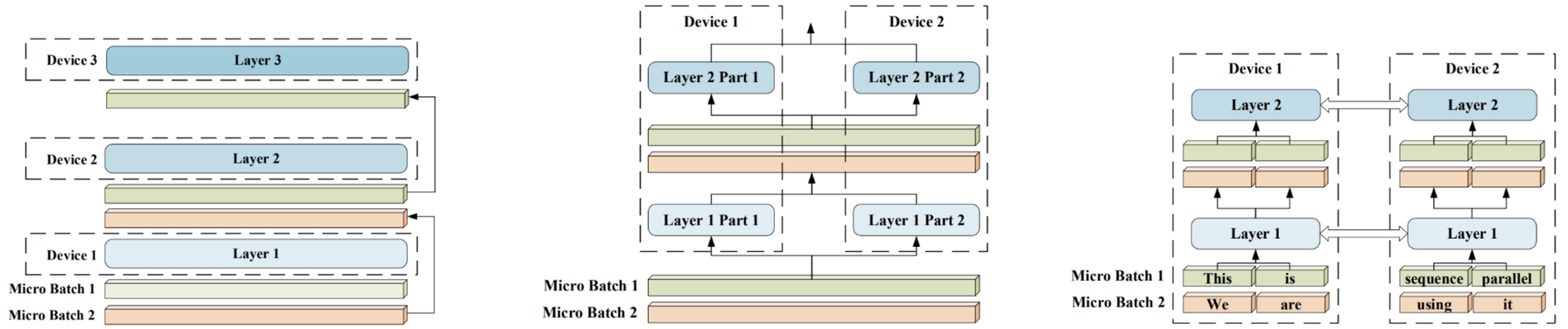
| Model Size | Iteration Time (seconds) | | Throughput Increase | Model FLOPs Utilization | Hardware FLOPs Utilization |
|---|---|---|---|---|---|
| | Full Recompute | Present Work | | | |
| 22B | 1.42 | 1.10 | 29.0% | 41.5% | 43.7% |
| 175B | 18.13 | 13.75 | 31.8% | 51.4% | 52.8% |
| 530B | 49.05 | 37.83 | 29.7% | 56.0% | 57.0% |
| 1T | 94.42 | 71.49 | 32.1% | 56.3% | 57.0% |

Table 5: End-to-end iteration time. Our approach results in throughput increase of around 30%.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# DEALING WITH MEMORY CONSTRAINTS

## Sequence Parallelism



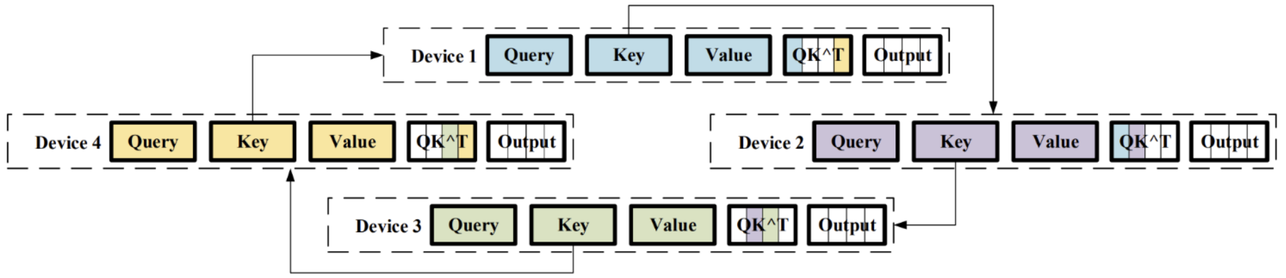

(a) Pipeline parallelism (b) Tensor parallelism (c) Sequence parallelism (Ours)

Figure 1: The overall architecture of the proposed sequence parallelism and existing parallel approaches. For sequence parallelism, Device 1 and Device 2 share the same trainable parameters.

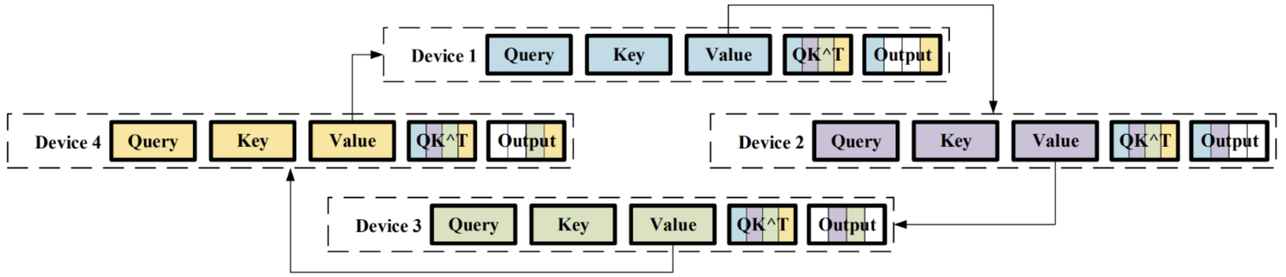
Sequence Parallelism: Long Sequence Training from System Perspective Shenggui Li, Fuzhao Xue1 Chaitanya Baranwal Yongbin Li Yang You. 2022

# DEALING WITH MEMORY CONSTRAINTS

## Sequence Parallelism




(a) Transmitting key embeddings among devices to calculate attention scores



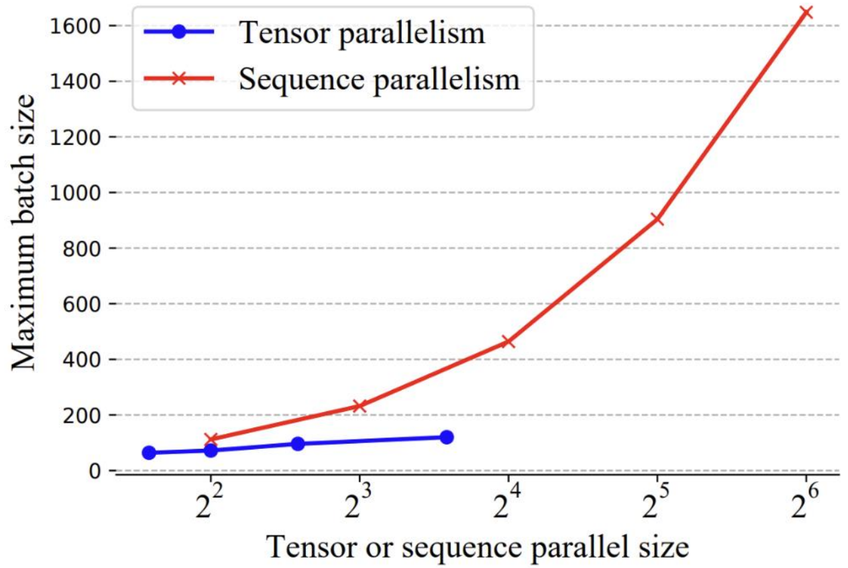

(b) Transmitting value embeddings among devices to calculate the output of attention layers

Figure 2: Ring Self-Attention

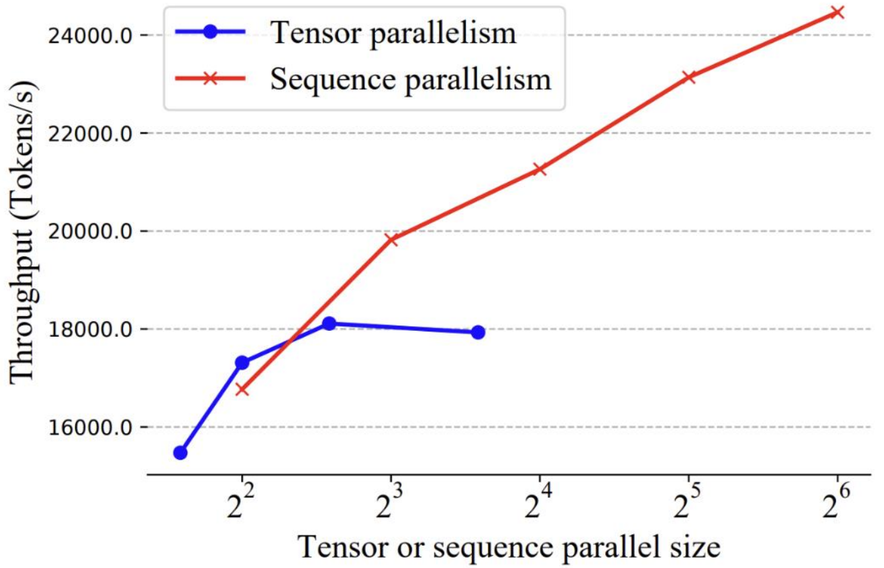
Sequence Parallelism: Long Sequence Training from System Perspective Shenggui Li, Fuzhao Xue1 Chaitanya Baranwal Yongbin Li Yang You. 2022

# DEALING WITH MEMORY CONSTRAINTS

## Sequence Parallelism




(a) Maximum batch size of BERT Base scaling along tensor or sequence parallel size




(b) Throughput of BERT Base scaling along tensor or sequence parallel size

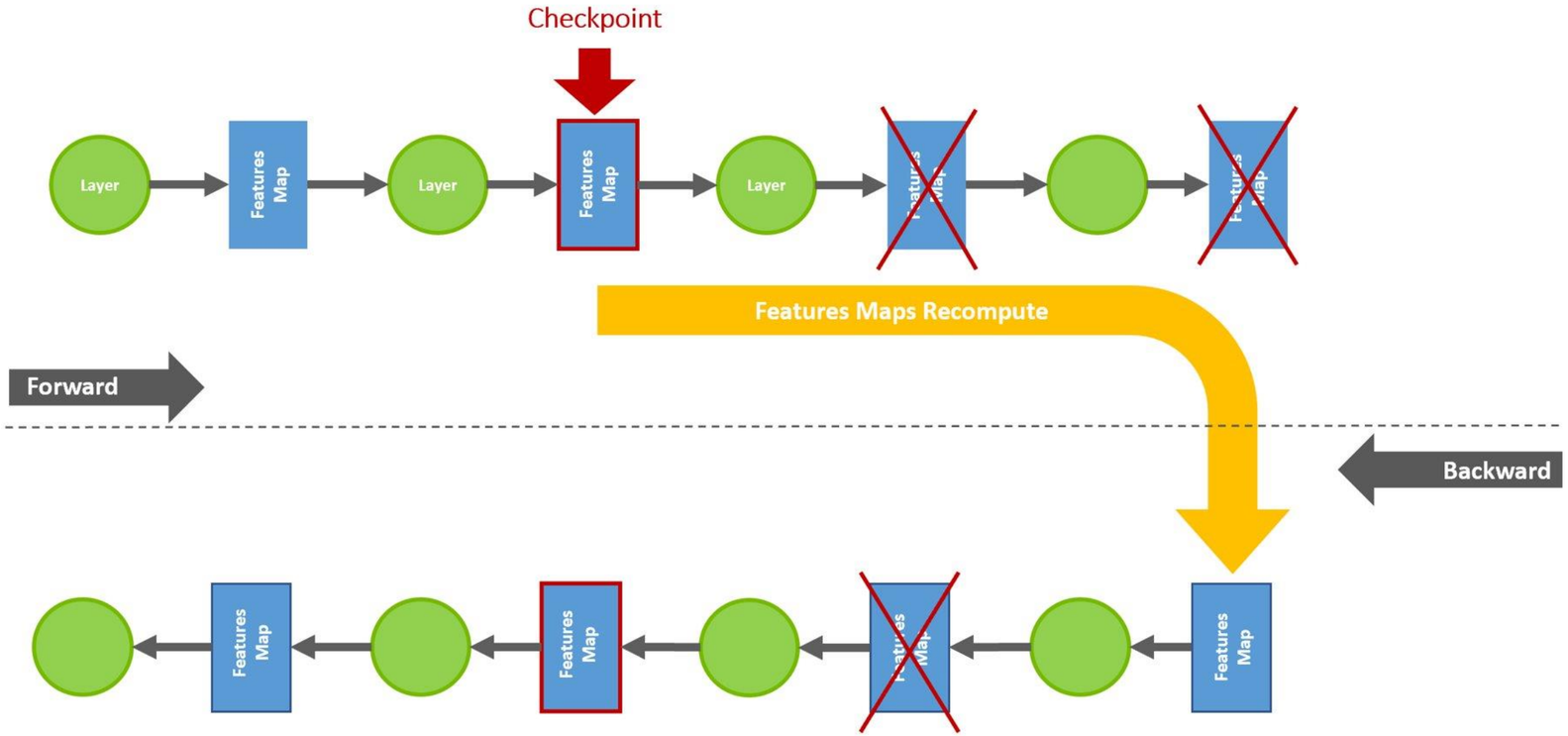
Figure 3: Scaling with sequence/tensor parallelism

Sequence Parallelism: Long Sequence Training from System Perspective Shenggui Li, Fuzhao Xue1 Chaitanya Baranwal Yongbin Li Yang You. 2022

# ACTIVATIONS CHECKPOINTING

# ACTIVATION RECOMPUTE CHALLENGES

## Activation Recompute




Balance the memory savings and computational overhead?

# ACTIVATION RECOMPUTE CHALLENGES

## Microbatch Level Activation Recomputation

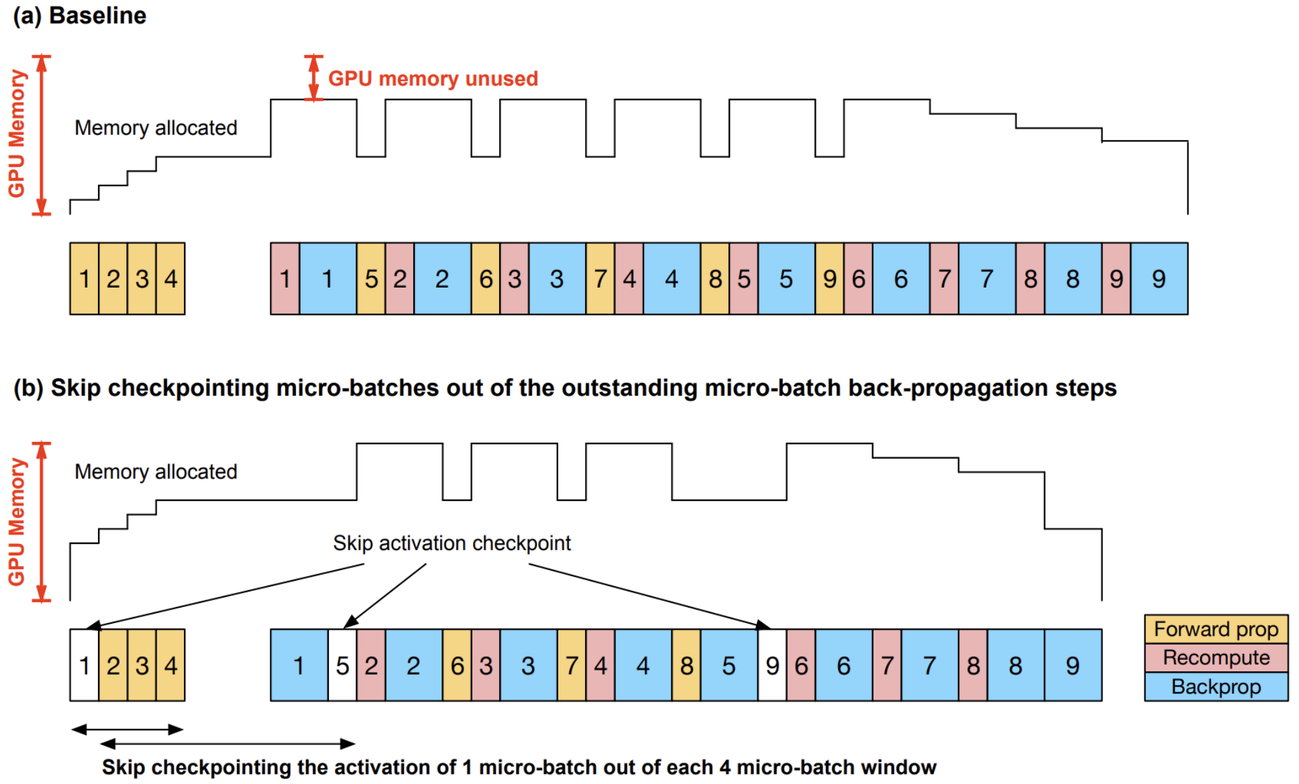



Figure 10: Computation and memory usage patterns of the baseline activation recomputation and microbatch level activation recomputation. Yellow boxes are a forward pass with activations checkpointed (i.e. only some activations are saved), red boxes are activation recomputation, blue boxes are backpropagation, and white boxes are a foward pass with all activations saved.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# ACTIVATION RECOMPUTE CHALLENGES

## Selective Activation Recompute with Megatron-LM

**Selective recomputation:**

- Saves the activations that take less space and are expensive to recompute
- Recompute activations that take a lot of space but are relatively cheap to recompute.

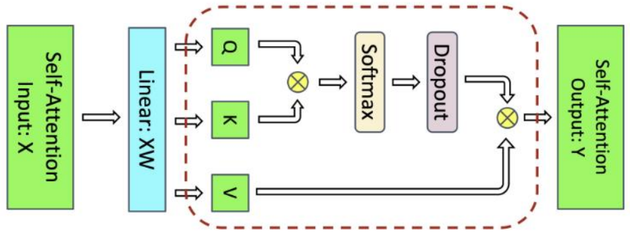



Figure 3: Self-attention block. The red dashed line shows the regions to which selective activation recomputation is applied (see Section 5 for more details on selective activation recomputation).

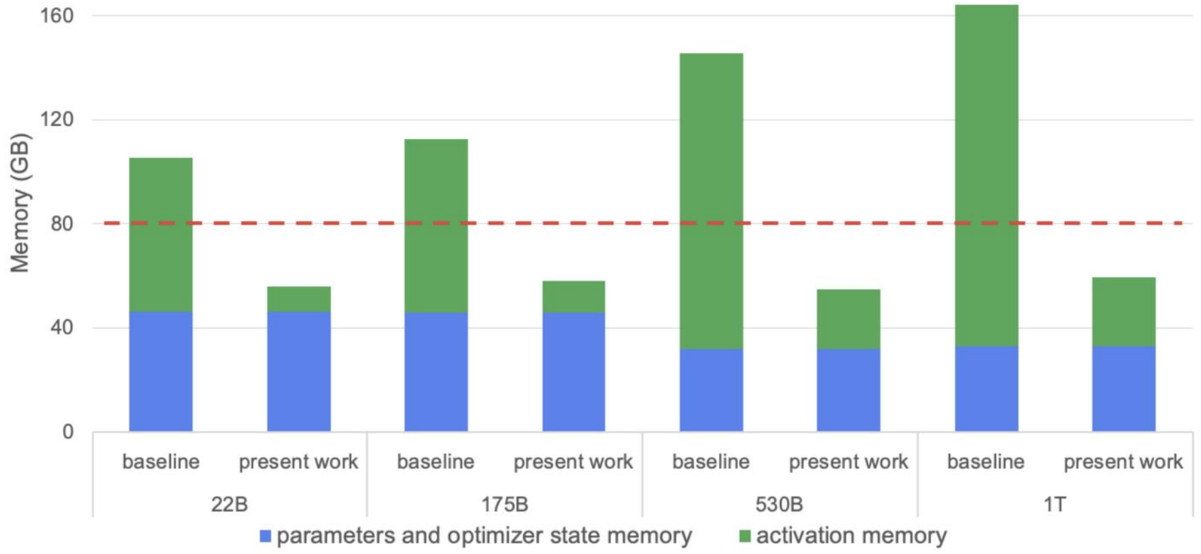
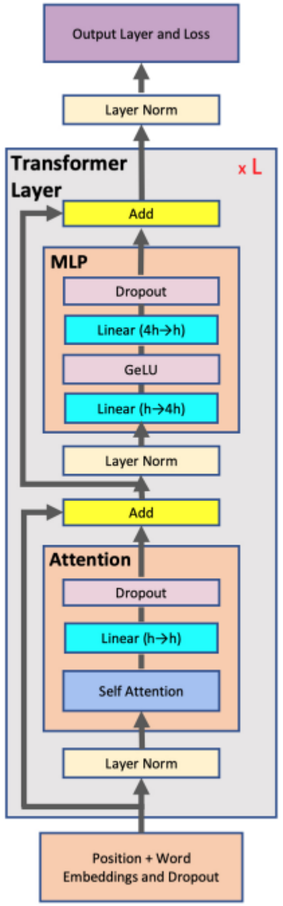



Figure 1: Parameters, optimizer state, and activations memory. The dashed red line represents the memory capacity of an NVIDIA A100 GPU. Present work reduces the activation memory required to fit the model. Details of the model configurations are provided in Table 3.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# ACTIVATION RECOMPUTE CHALLENGES

## Selective Activation Recompute with Megatron-LM




**Attention block:** which includes self attention followed by a linear projection and an attention dropout. The linear projection stores its input activations with size $2sbh$ and the attention dropout requires a mask with size $sbh$. The self attention shown in Figure 3 consists of several elements:

- **Query ($Q$), Key ($K$), and Value ($V$) matrix multiplies:** We only need to store their shared input with size $2sbh$.
- **$QK^T$ matrix multiply:** It requires storage of both $Q$ and $K$ with total size $4sbh$.
- **Softmax:** Softmax output with size $2as^2b$ is required for back-propagation.
- **Softmax dropout:** Only a mask with size $as^2b$ is needed.
- **Attention over Values ($V$):** We need to store the dropout output ($2as^2b$) and the Values ($2sbh$) and therefore need $2as^2b + 2sbh$ of storage.

Summing the above values, in total, the attention block requires $11sbh + 5as^2b$ bytes of storage.

**MLP:** The two linear layers store their inputs with size $2sbh$ and $8sbh$. The GeLU non-linearity also needs its input with size $8sbh$ for back-propagation. Finally, dropout stores its mask with size $sbh$. In total, MLP block requires $19sbh$ bytes of storage.

**Layer norm:** Each layer norm stores its input with size $2sbh$ and therefore in total, we will need $4sbh$ of storage.

Summing the memory required for attention, MLP, and the layer-norms, the memory required to store the activations for a single layer of a transformer network is:

$$\text{Activations memory per layer} = sbh\left(34 + 5\frac{as}{h}\right). \quad (1)$$

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# ACTIVATION RECOMPUTE CHALLENGES

## Selective Activation Recompute with Megatron-LM

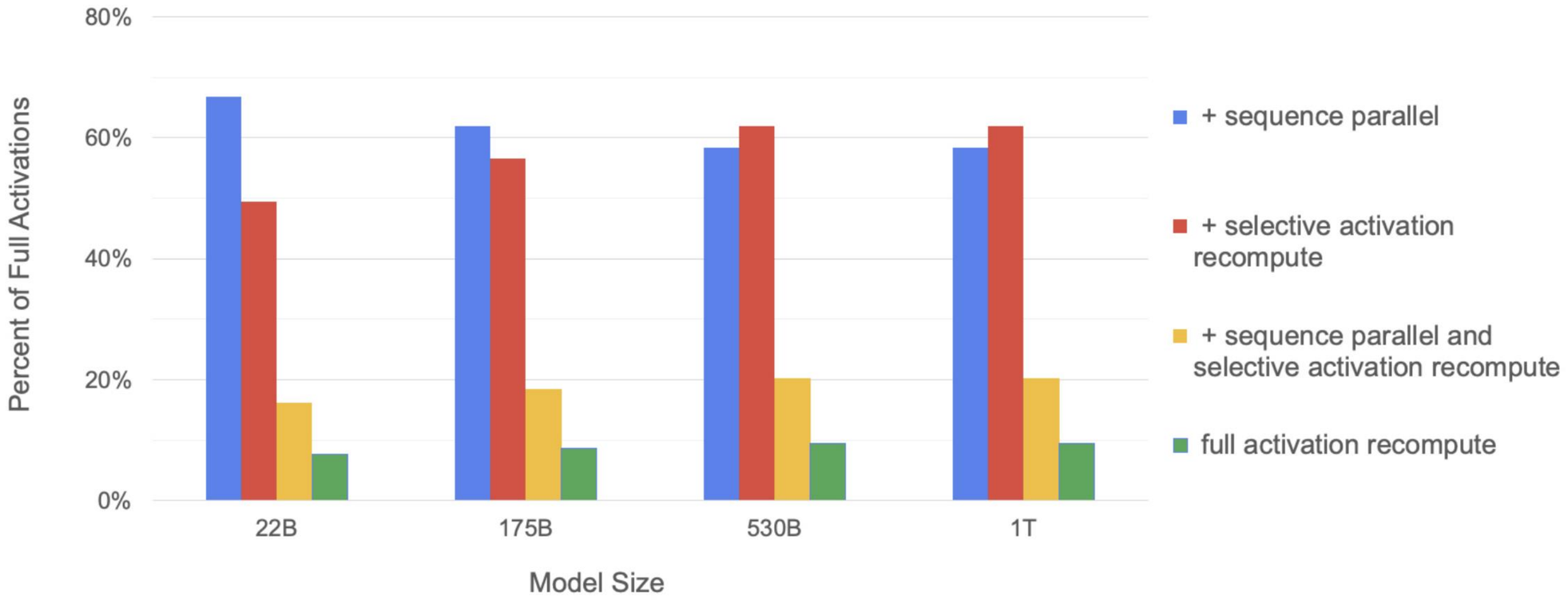



Figure 7: Percentage of required memory compared to the tensor-level parallel baseline. As the model size increases, both sequence parallelism and selective activation recomputation have similar memory savings and together they reduce the memory required by $\sim 5\times$.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# ACTIVATION RECOMPUTE CHALLENGES

## Selective Activation Recompute with Megatron-LM

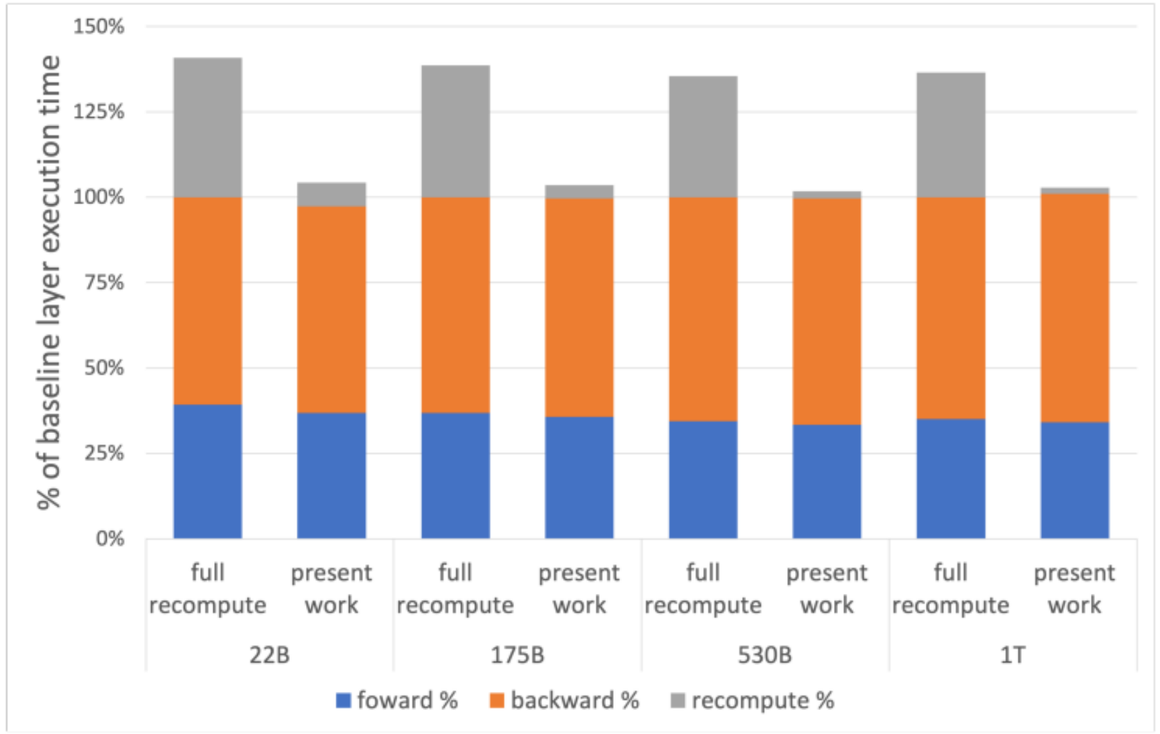



Figure 8: Per layer breakdown of forward, backward, and recompute times. Baseline is the case with no recomputation and no sequence parallelism. Present work includes both sequence parallelism and selective activation recomputation.

| Experiment | Forward (ms) | Backward (ms) | Combined (ms) | Overhead (%) |
|---|---|---|---|---|
| Baseline no recompute | 7.7 | 11.9 | 19.6 | – |
| Sequence Parallelism | 7.2 | 11.8 | 19.0 | −3% |
| Baseline with recompute | 7.7 | 19.5 | 27.2 | 39% |
| Selective Recompute | 7.7 | 13.2 | 20.9 | 7% |
| Selective + Sequence | 7.2 | 13.1 | 20.3 | 4% |

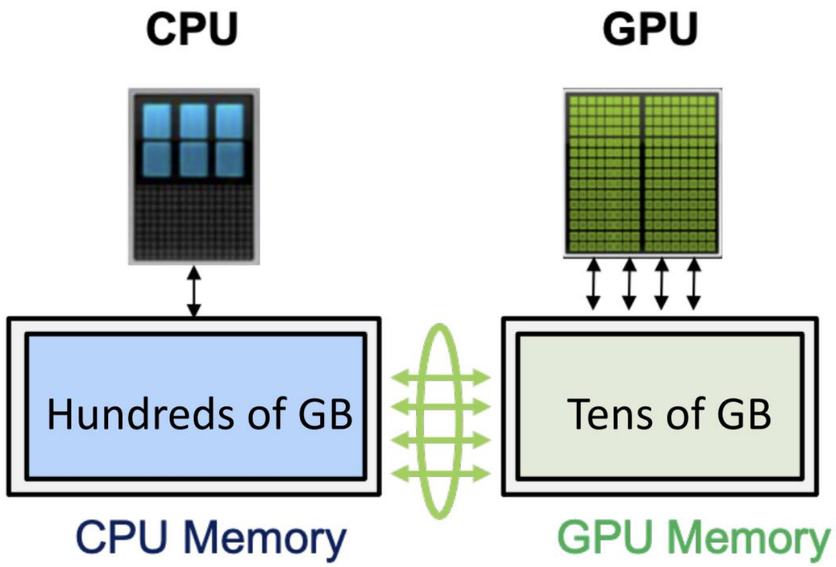
Table 4: Time to complete the forward and backward pass of a single transformer layer of the 22B model.

Reducing Activation Recomputation in Large Transformer Models Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro

# OFFLOADING TO THE CPU

# DEALING WITH MEMORY CONSTRAINTS

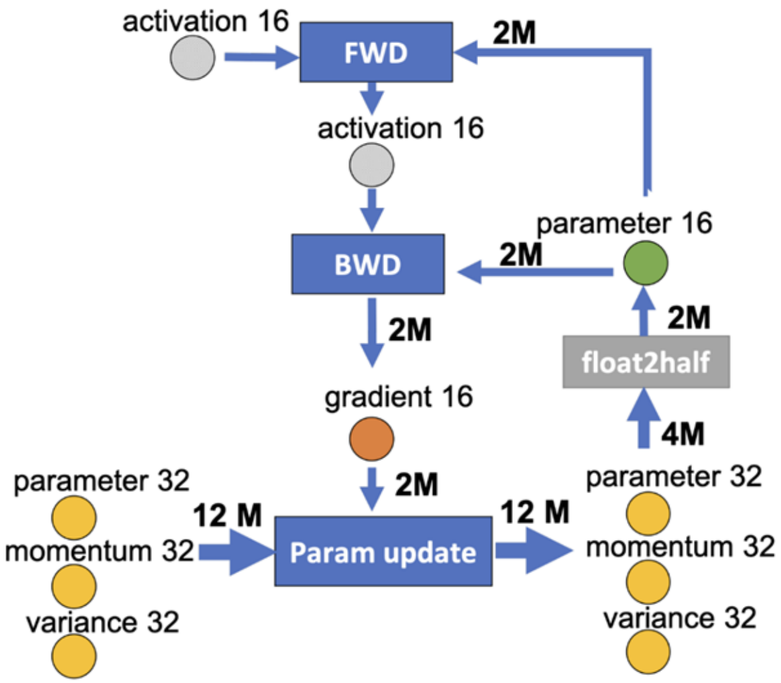
## Offloading data to CPU memory




Offload CPU tensors not used in computation form GPU to CPU

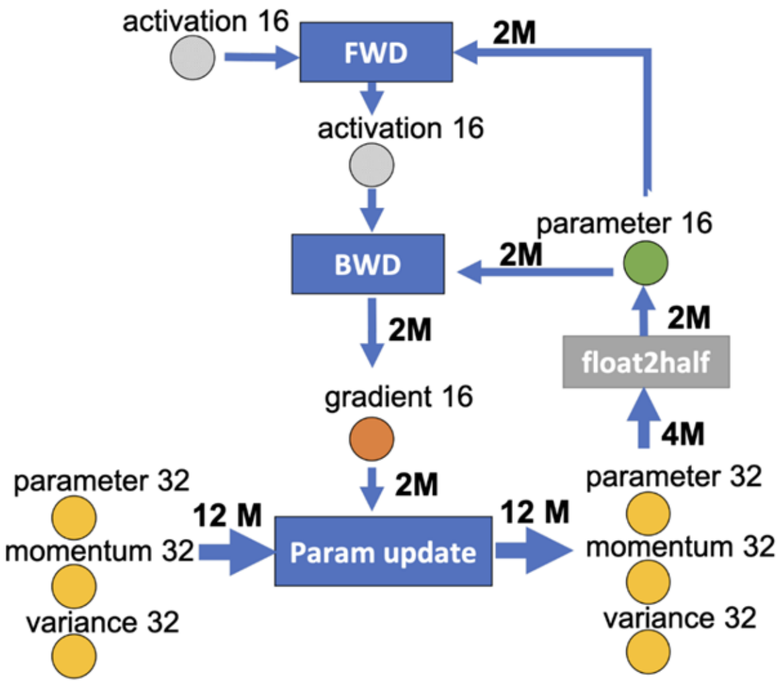
# DEALING WITH MEMORY CONSTRAINTS

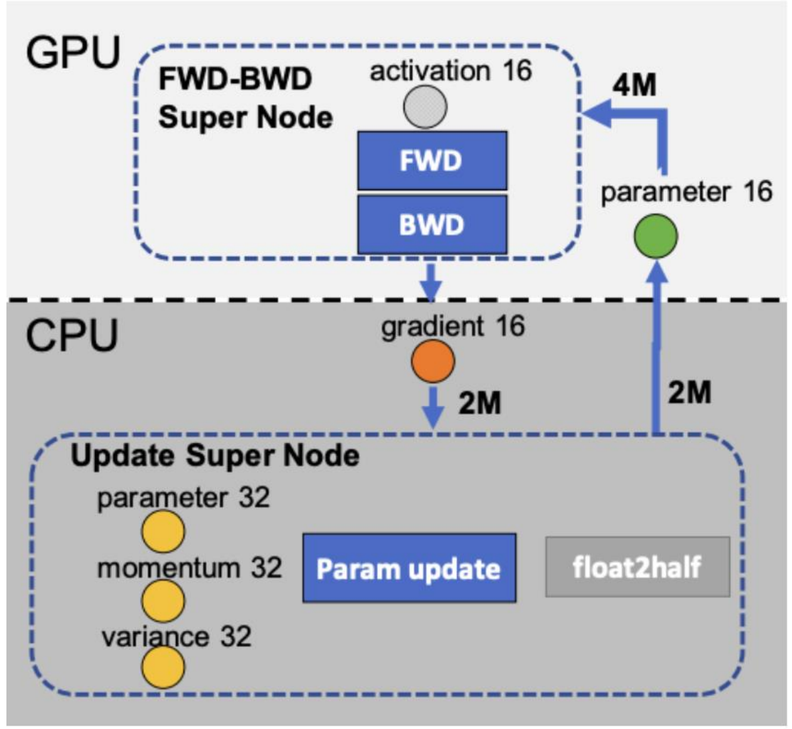
## Offloading data to CPU memory



# DEALING WITH MEMORY CONSTRAINTS

## Offloading data to CPU memory

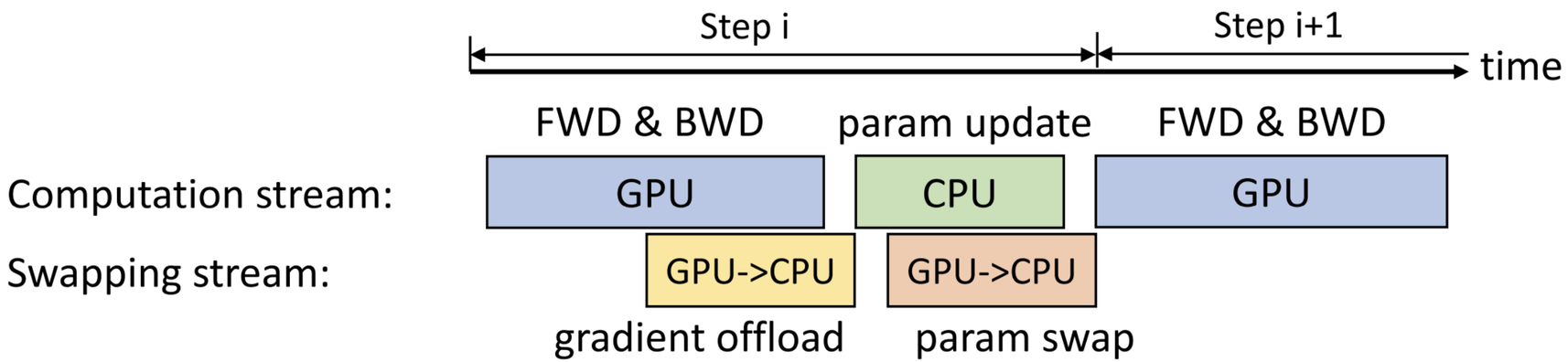



- Keep the fp16 parameters and forward and backward computation on GPU.
- Offloading to CPU:
  - Gradients
  - Optimizer states
  - Optimizer computation

# DEALING WITH MEMORY CONSTRAINTS

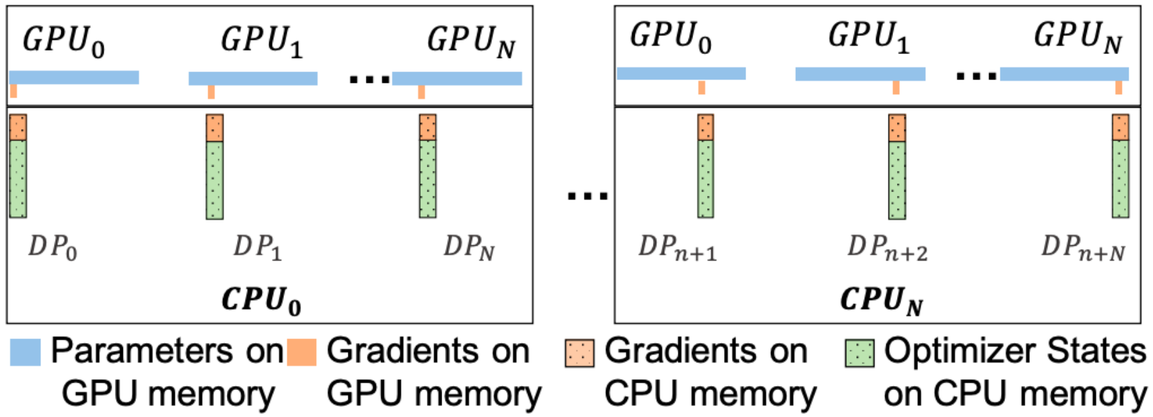
## Offloading data to CPU memory



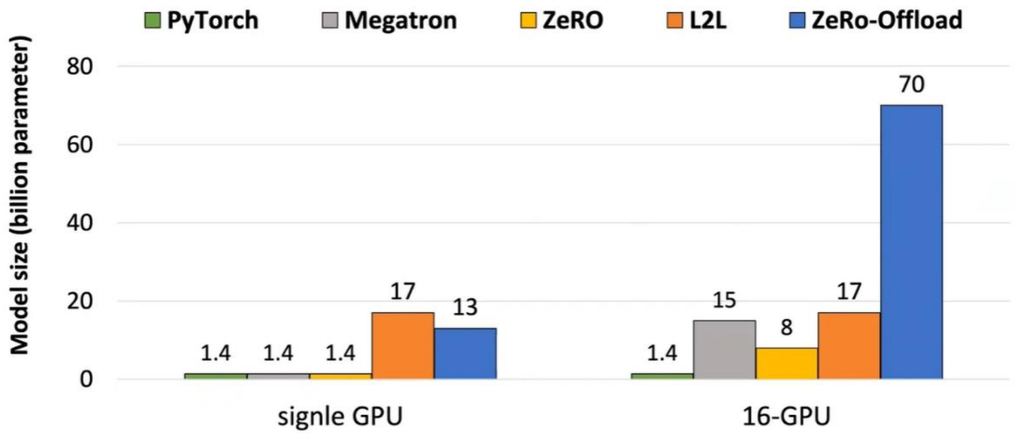

| FWD-BWD | param16 | gradient16 | Update | Memory | Reduction |
|---|---|---|---|---|---|
| GPU | GPU | GPU | GPU | 16M | 1x(baseline) |
| GPU | GPU | CPU | GPU | 14M | 1.14x |
| GPU | GPU | GPU | CPU | 4M | 4x |
| GPU | GPU | CPU | CPU | 4M | 8x |

# DEALING WITH MEMORY CONSTRAINTS

## DeepSpeed ZeRO-Offload + with Data Parallelism




| DGX-2 node | |
|---|---|
| GPU | 16 NVIDIA Tesla V100 Tensor Core GPUs |
| GPU Memory | 32GB HBM2 on each GPU |
| CPU | 2 Intel Xeon Platinum 8168 Processors |
| CPU Memory | 1.5TB 2666MHz DDR4 |
| CPU cache | L1, L2, and L3 are 32K, 1M, and 33M, respectively |
| PCIe | bidirectional 32 GBps PCIe |




ZeRO-Offload enables 13B-model training on a single GPU, and easily enables training of up to 70B parameter with 16 GPUs.

# MIXTURE OF EXPERTS

# MIXTURE OF EXPERTS

## Not a new Idea

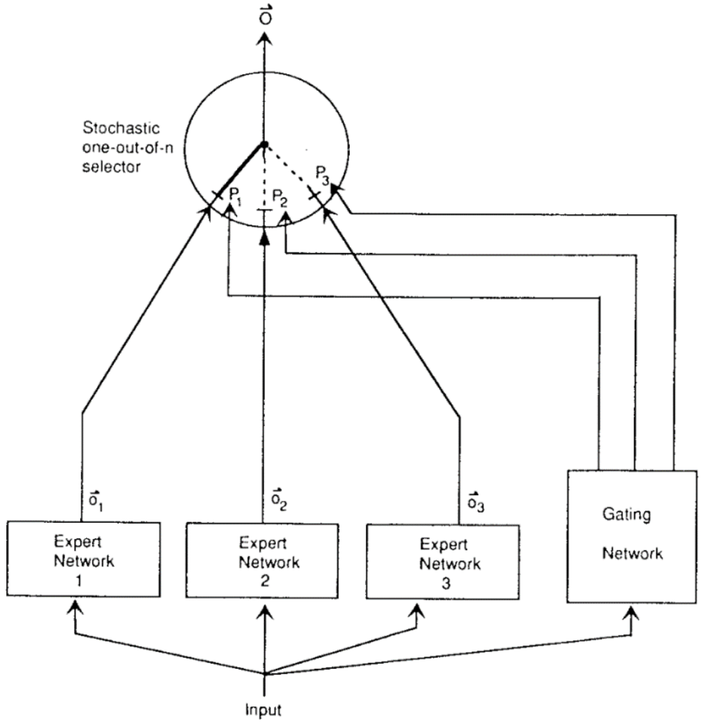



Figure 1: A system of expert and gating networks. Each expert is a feedforward network and all experts receive the same input and have the same number of outputs. The gating network is also feedforward, and typically receives the same input as the expert networks. It has normalized outputs $p_j = \exp(x_j)/\sum_i \exp(x_i)$, where $x_j$ is the total weighted input received by output unit $j$ of the gating network. The selector acts like a multiple input, single output stochastic switch; the probability that the switch will select the output from expert $j$ is $p_j$.

R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. Neural Computation, 3(1):79–87, 1991.

# MIXTURE OF EXPERTS

## MoE in Neural Networks

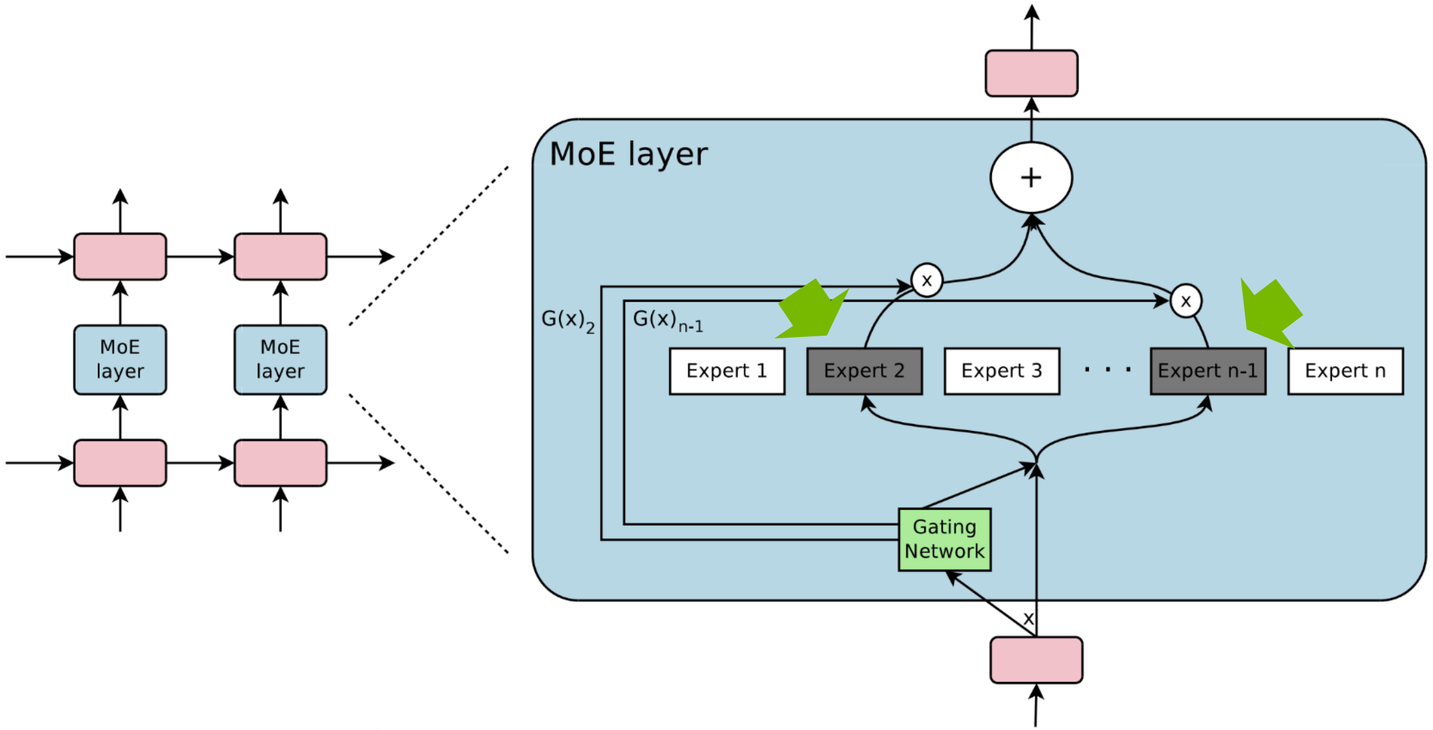



Figure 1: A Mixture of Experts (MoE) layer embedded within a recurrent language model. In this case, the sparse gating function selects two experts to perform computations. Their outputs are modulated by the outputs of the gating network.

Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, Jeff Dean. ICLR 2017

# MIXTURE OF EXPERTS

## Switch Transformers

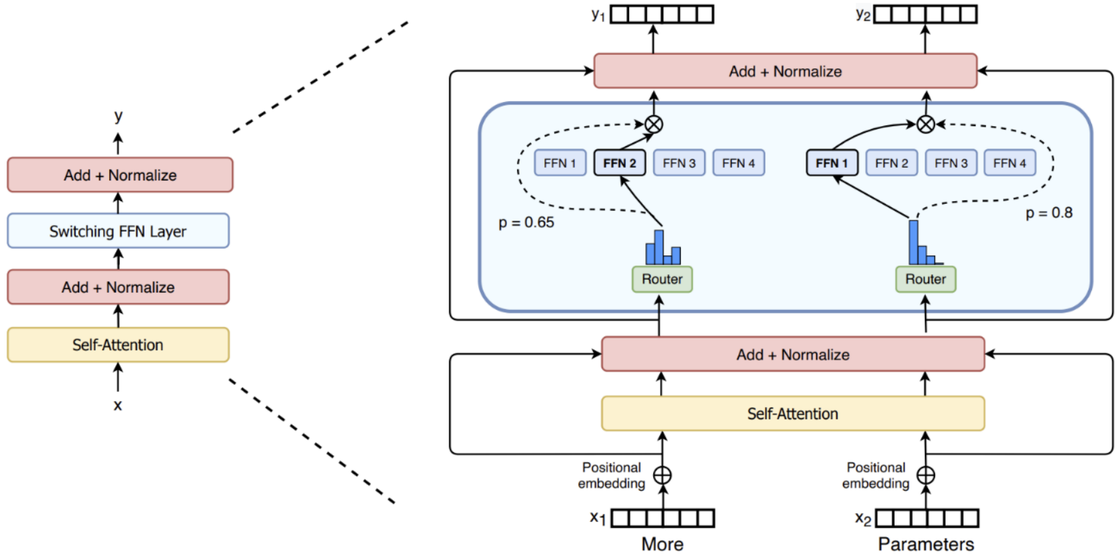



Figure 2: Illustration of a Switch Transformer encoder block. We replace the dense feed forward network (FFN) layer present in the Transformer with a sparse Switch FFN layer (light blue). The layer operates independently on the tokens in the sequence. We diagram two tokens ($x_1$ = "More" and $x_2$ = "Parameters" below) being routed (solid lines) across four FFN experts, where the router independently routes each token. The switch FFN layer returns the output of the selected FFN multiplied by the router gate value (dotted-line).

# MIXTURE OF EXPERTS

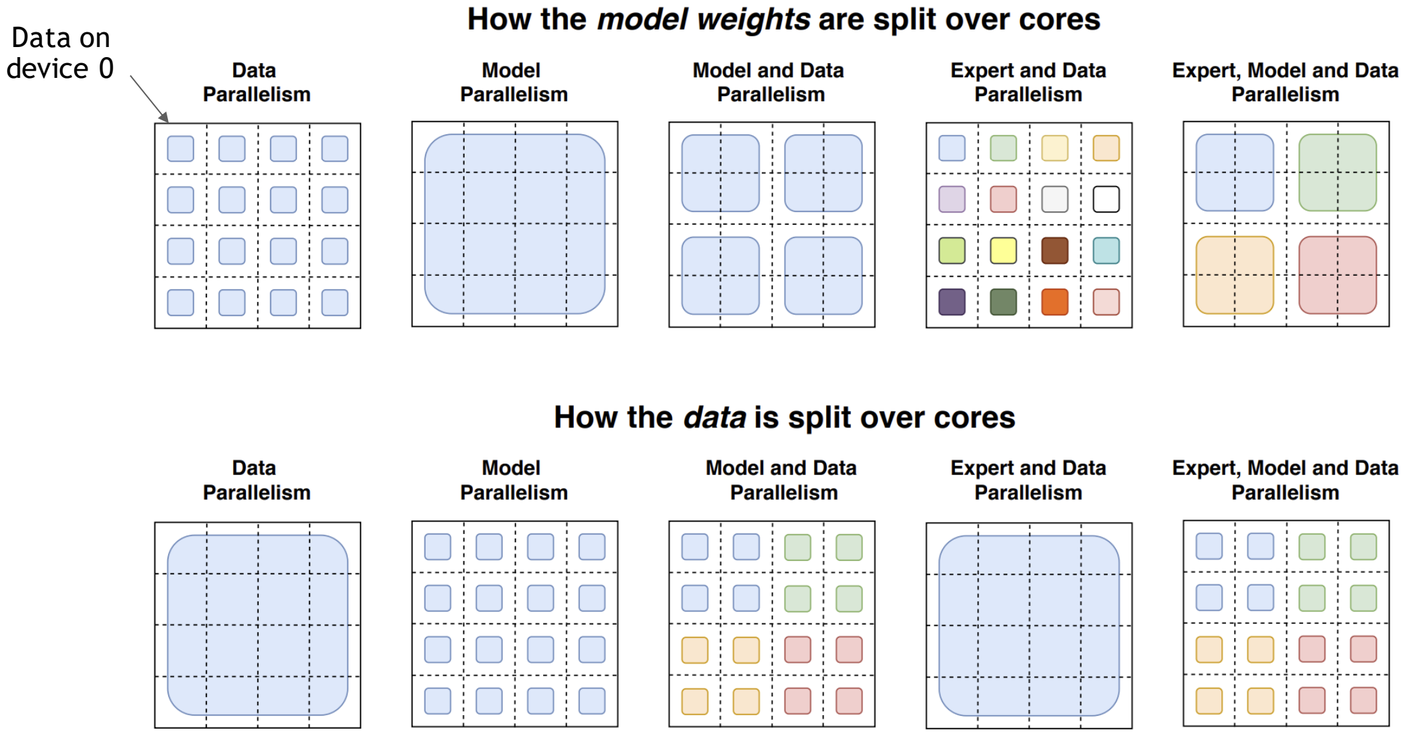
## Switch Transformers - Data and weight partitioning strategies

- Model weights partitions across cores:

Shapes of different sizes in this row represent larger weight matrices in the Feed Forward Network (FFN) layers. Each color of the shaded squares identifies a unique weight matrix. The number of parameters per core is fixed, but larger weight matrices will apply more computation to each token.

- Data partitions across cores:

Each core holds the same number of tokens which maintains a fixed memory usage across all strategies. The partitioning strategies have different properties of allowing each core to either have the same tokens or different tokens across cores, which is what the different colors symbolize.




Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, Jeff Dean. ICLR 2017

# MIXTURE OF EXPERTS

## Switch Transformers

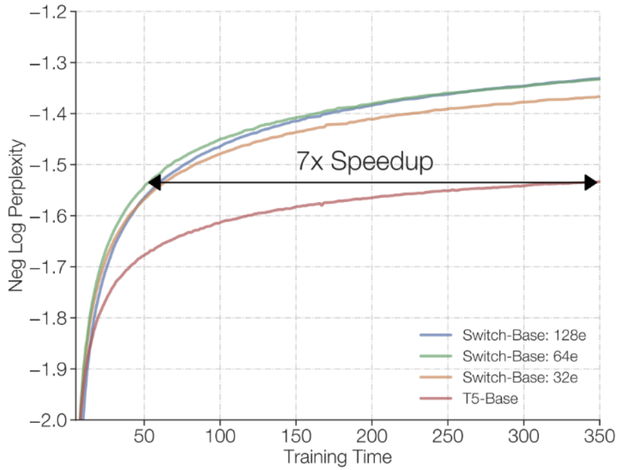



Figure 5: Speed advantage of Switch Transformer. All models trained on 32 TPUv3 cores with equal FLOPs per example. For a fixed amount of computation and training time, Switch Transformers significantly outperform the dense Transformer baseline. Our 64 expert Switch-Base model achieves the same quality in *one-seventh* the time of the T5-Base and continues to improve.

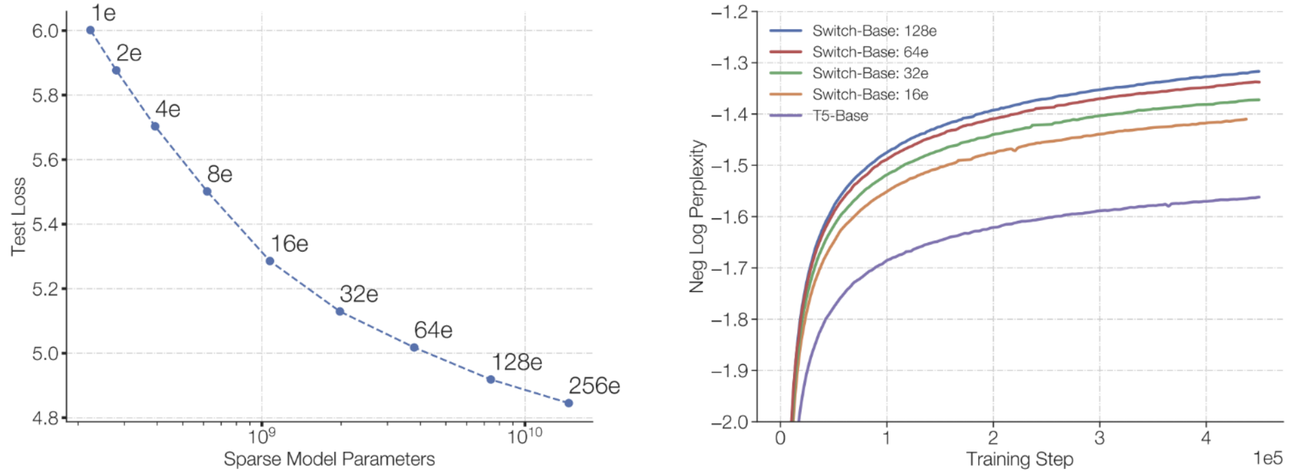



Figure 1: Scaling and sample efficiency of Switch Transformers. Left Plot: Scaling properties for increasingly sparse (more experts) Switch Transformers. Right Plot: Negative log perplexity comparing Switch Transformers to T5 (Raffel et al., 2019) models using the same compute budget.

Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, Jeff Dean. ICLR 2017

# MIXTURE OF EXPERTS

## GShard: with device placement

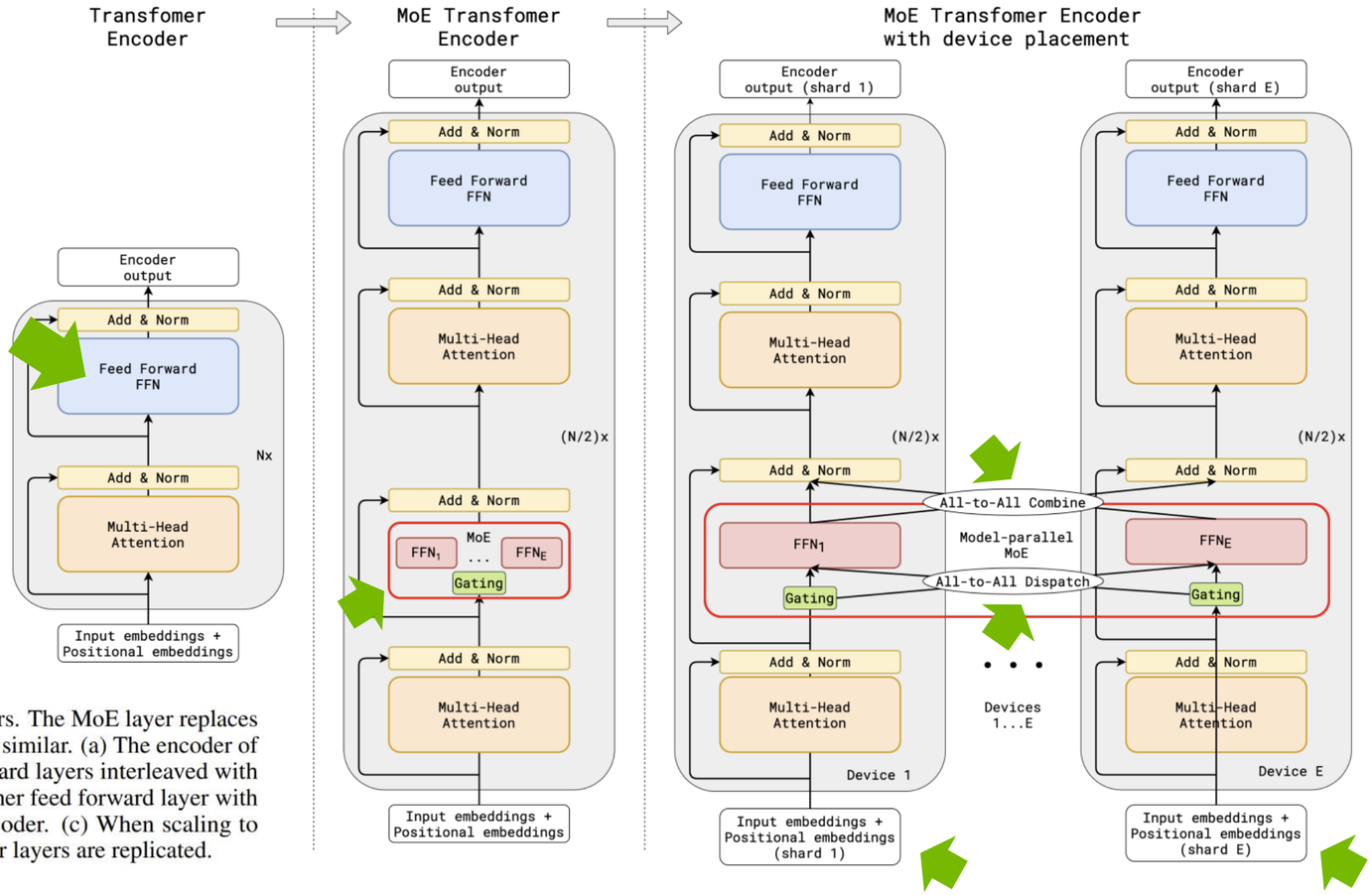



Figure 3: Illustration of scaling of Transformer Encoder with MoE Layers. The MoE layer replaces the every other Transformer feed-forward layer. Decoder modification is similar. (a) The encoder of a standard Transformer model is a stack of self-attention and feed forward layers interleaved with residual connections and layer normalization. (b) By replacing every other feed forward layer with a MoE layer, we get the model structure of the MoE Transformer Encoder. (c) When scaling to multiple devices, the MoE layer is sharded across devices, while all other layers are replicated.

D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In ICLR, 2021

# MIXTURE OF EXPERTS

## GShard: with device placement

| Id | Model | BLEU avg. | ΔBLEU avg. | Weights |
|---|---|---|---|---|
| (1) | MoE(2048E, 36L) | **44.3** | 13.5 | 600B |
| (2) | MoE(2048E, 12L) | 41.3 | 10.5 | 200B |
| (3) | MoE(512E, 36L) | 43.7 | 12.9 | 150B |
| (4) | MoE(512E, 12L) | 40.0 | 9.2 | 50B |
| (5) | MoE(128E, 36L) | 39.0 | 8.2 | 37B |
| (6) | MoE(128E, 12L) | 36.7 | 5.9 | 12.5B |
| * | T(96L) | 36.9 | 6.1 | 2.3B |
| * | Baselines | 30.8 | - | 100×0.4B |

Figure 6: Translation quality comparison of multilingual MoE Transformer models trained with GShard and monolingual baselines. Positions along the $x$-axis represent languages, raging from high- to low-resource. ΔBLEU represents the quality gain of a single multilingual model compared to a monolingual Transformer model trained and tuned for a specific language. MoE Transformer models trained with GShard are reported with solid trend-lines. Dashed trend-line represents a single 96 layer multilingual Transformer model T(96L) trained with GPipe on same dataset. Each trend-line is smoothed by a sliding window of 10 for clarity. (Best seen in color)

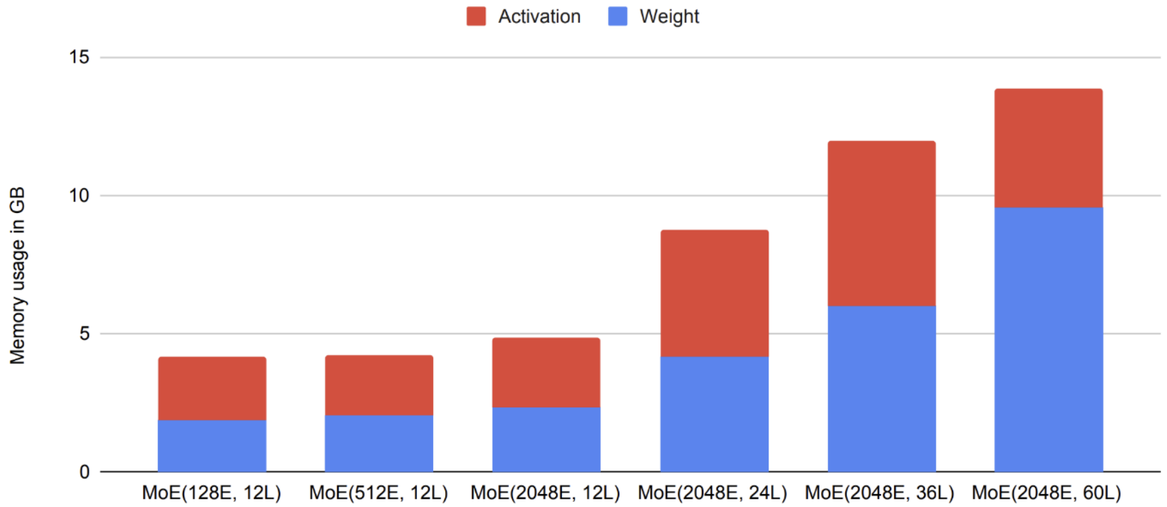



Figure 7: Per-device memory consumption in gigabytes.

D. Lepikhin, H. Lee, Y. Xu, D. Chen, O. Firat, Y. Huang, M. Krikun, N. Shazeer, and Z. Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In ICLR, 2021

# LAYERED GRADIENT ACCUMULATE

# GRADIENT ACCUMULATE

## Layered Gradient Accumulate

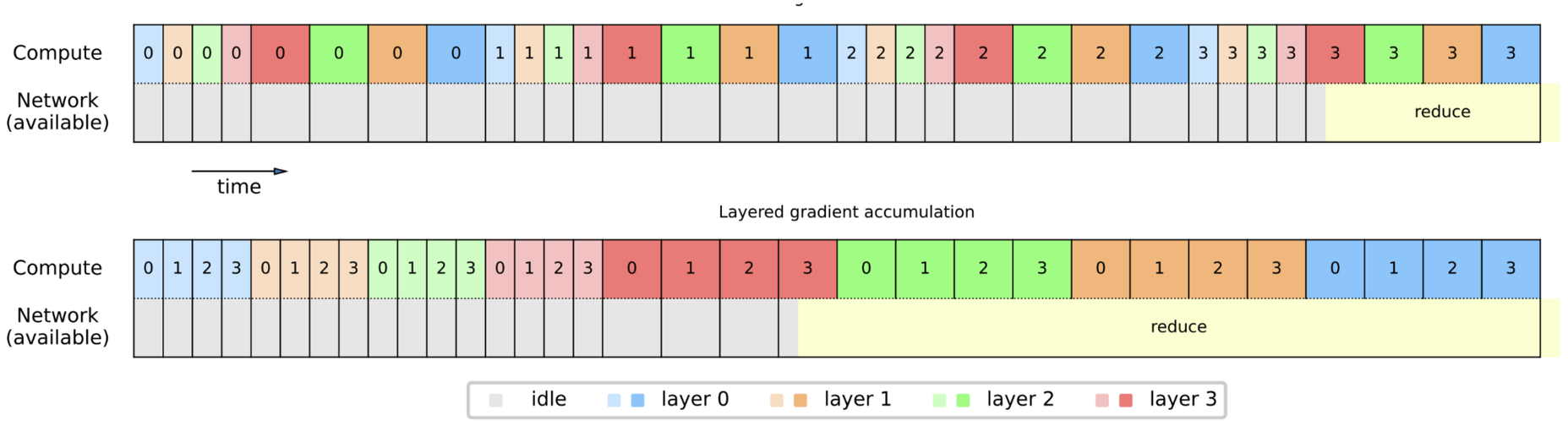



Figure 1: Computation and network scheduling example with data parallelism for standard gradient accumulation (top) and layered gradient accumulation (bottom). The colors represent the different layers, with different shades for the forward and backward pass (lengths not to scale), and the numbers indicate the micro-batch index. The layered version reduces the network requirement by spreading the gradient reduction over most of the backward pass.

# OPTIMIZE THE OPTIMIZER

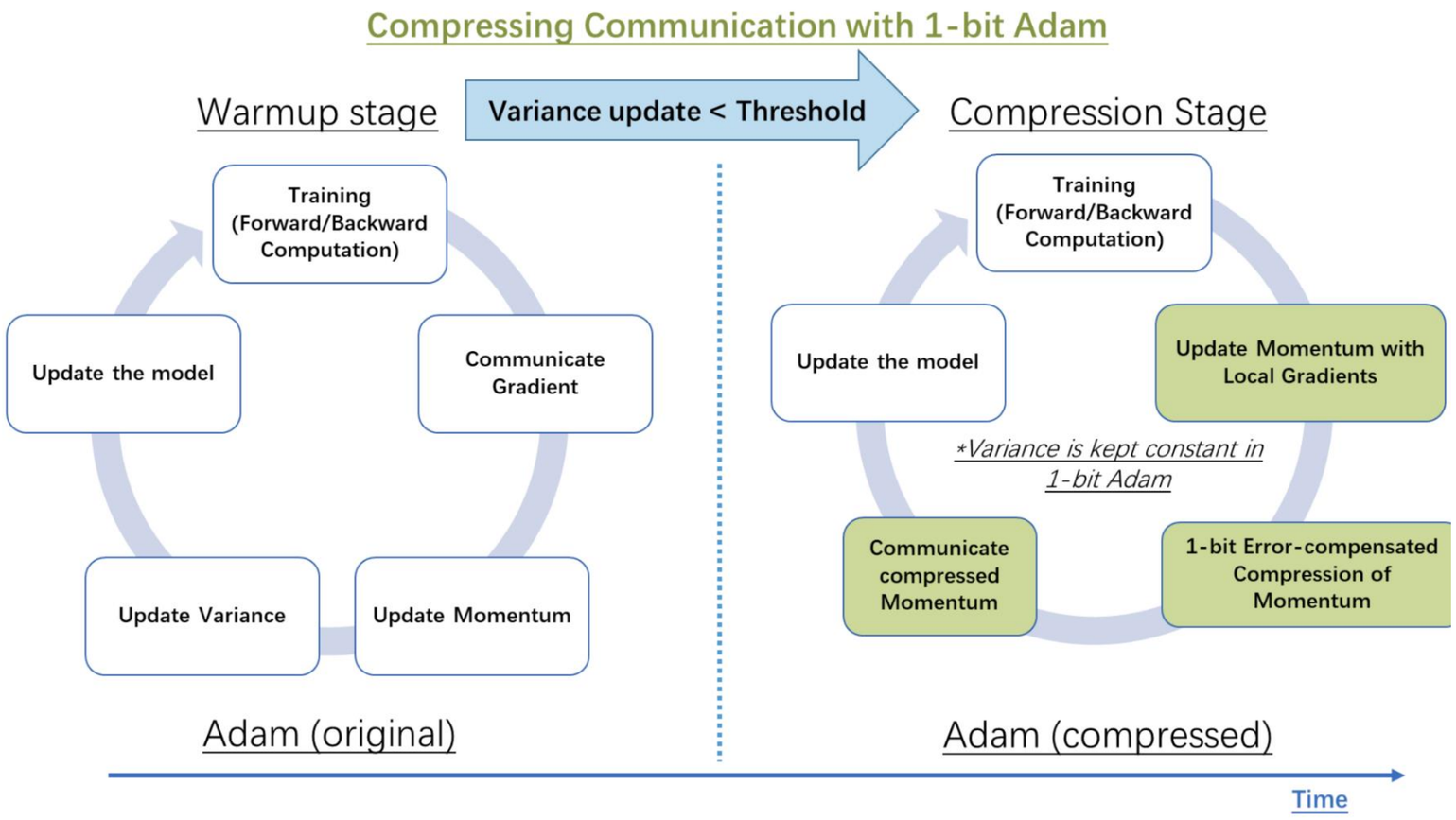
# 1-BIT ADAM

## 1-bit Adam

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t$$

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(g_t)^2$$

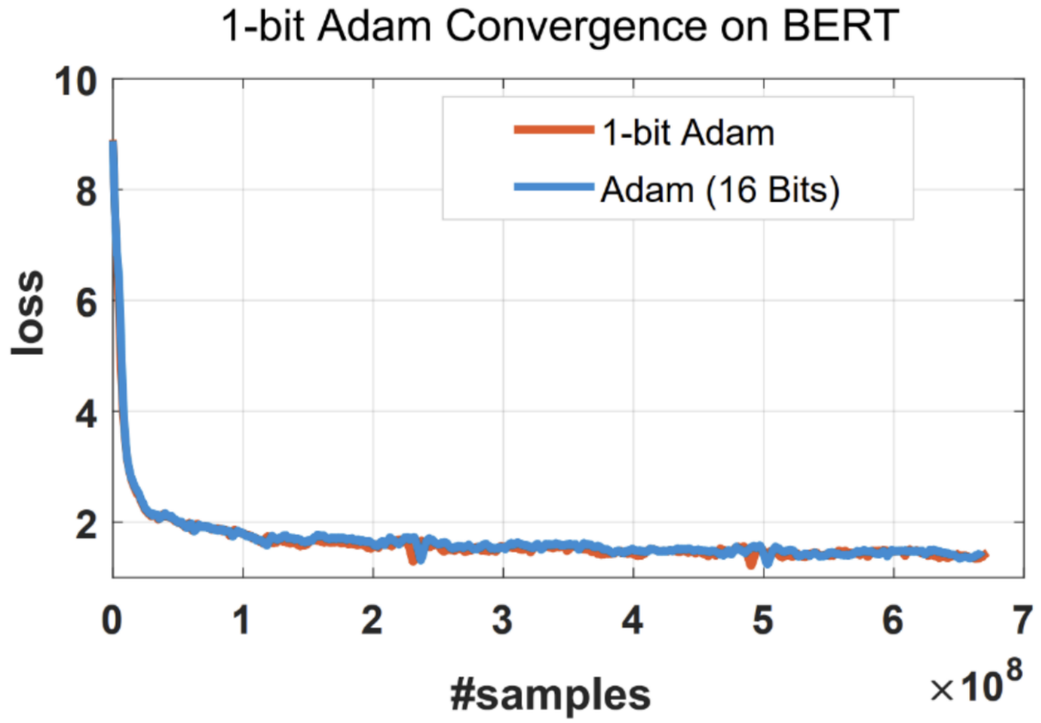
$$x_{t+1} = x_t - \gamma \frac{m_{t+1}}{\sqrt{v_{t+1}} + \eta}$$




1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, Yuxiong He

# 1-BIT ADAM

## Convergence and Throughput



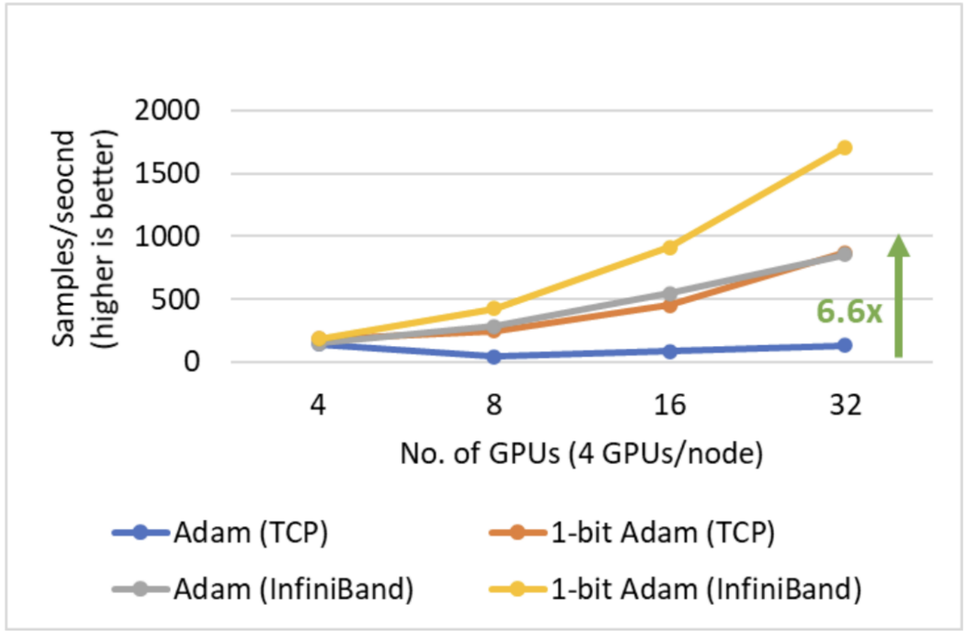




Figure 3: Scalability of 1-bit Adam for BERT-Large Pretraining on V100 GPUs with batch size of 16/GPU.

1-bit Adam: Communication Efficient Large-Scale Training with Adam's Convergence Speed. Hanlin Tang, Shaoduo Gan, Ammar Ahmad Awan, Samyam Rajbhandari, Conglong Li, Xiangru Lian, Ji Liu, Ce Zhang, Yuxiong He

# NCCL: NVIDIA COLLECTIVE COMMUNICATION LIBRARY

# MULTI-GPU TRAINING

## NCCL: NVIDIA Collective Communication Library

### NVIDIA Collective Communications Library

1. Automatic topology detection
2. Graph search for the optimal set of rings and trees with the highest bandwidth and lowest latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes
3. Provide routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter, point-to-point send and receive
4. Integrated within several Deep Learning frameworks such as Caffe2, MxNet, PyTorch
5. Optimized for all platforms, from desktop to DGX Superpod.

PCI

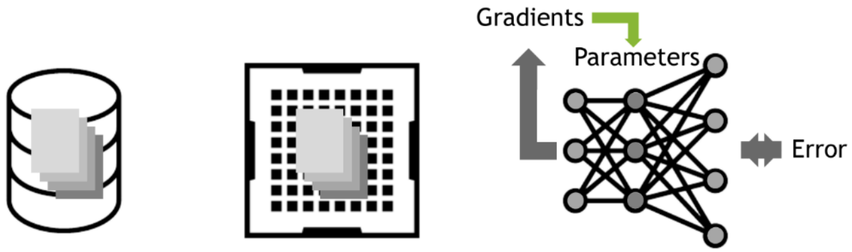
NVLink

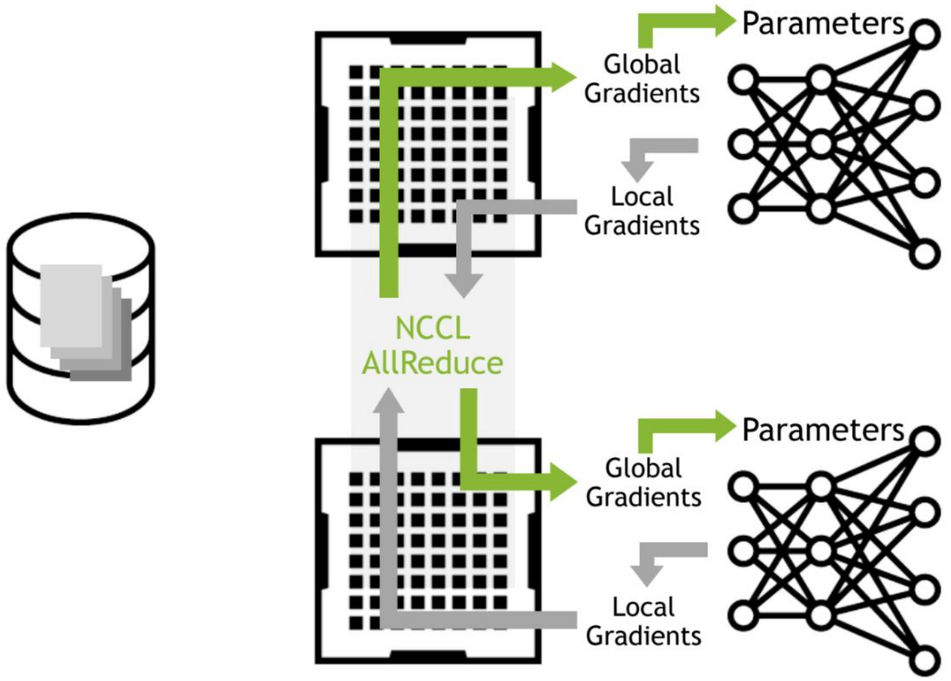
NVLink + Infiniband

https://developer.nvidia.com/nccl

# MULTI-GPU TRAINING

## NCCL : NVIDIA Collective Communication Library

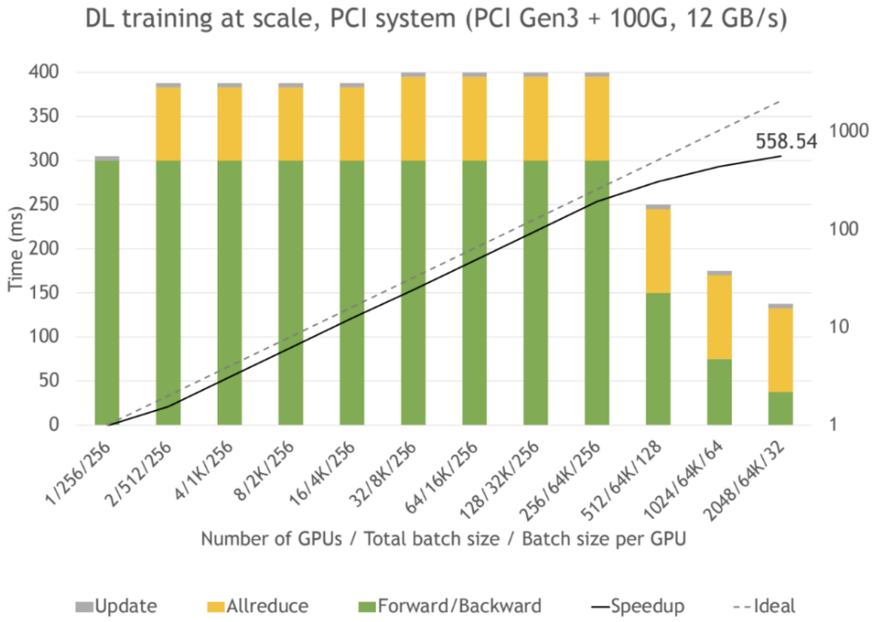


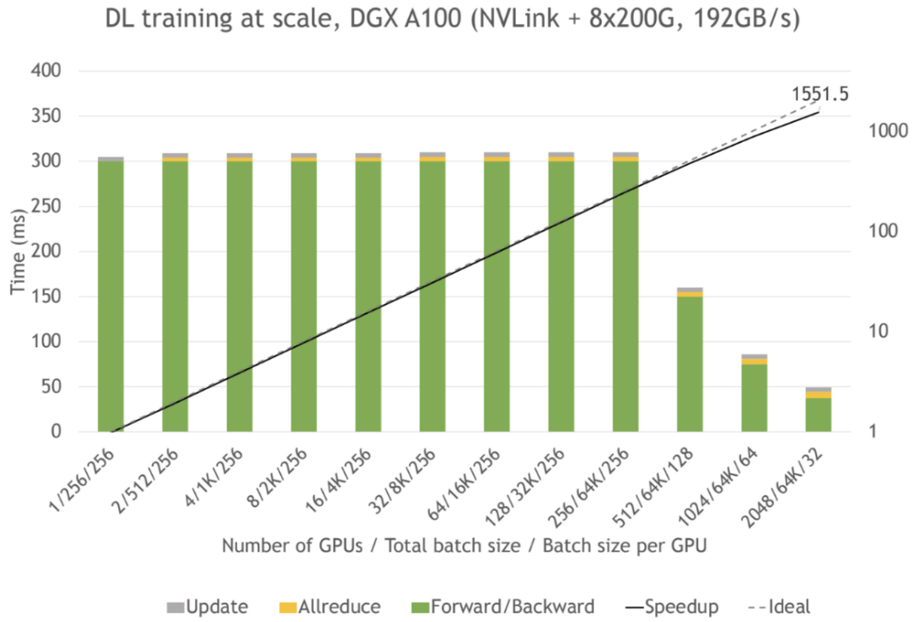




In Deep Learning, AllReduce operations used to sum gradients over multiple GPUs

# MULTI-GPU TRAINING

## Why NCCL performance is critical?




Weak scaling

Strong scaling

Efficiency: 77%

Max speedup: 640x

*Theoretical model of size 500MB, compute time 300ms / 256 samples*

Weak scaling

Strong scaling

Efficiency: 97%

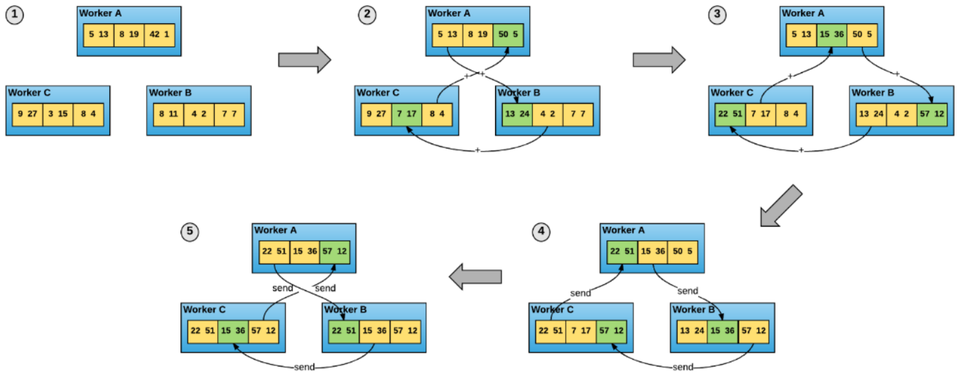
Max speedup: 4800x

NVIDIA

# MULTI-GPU TRAINING

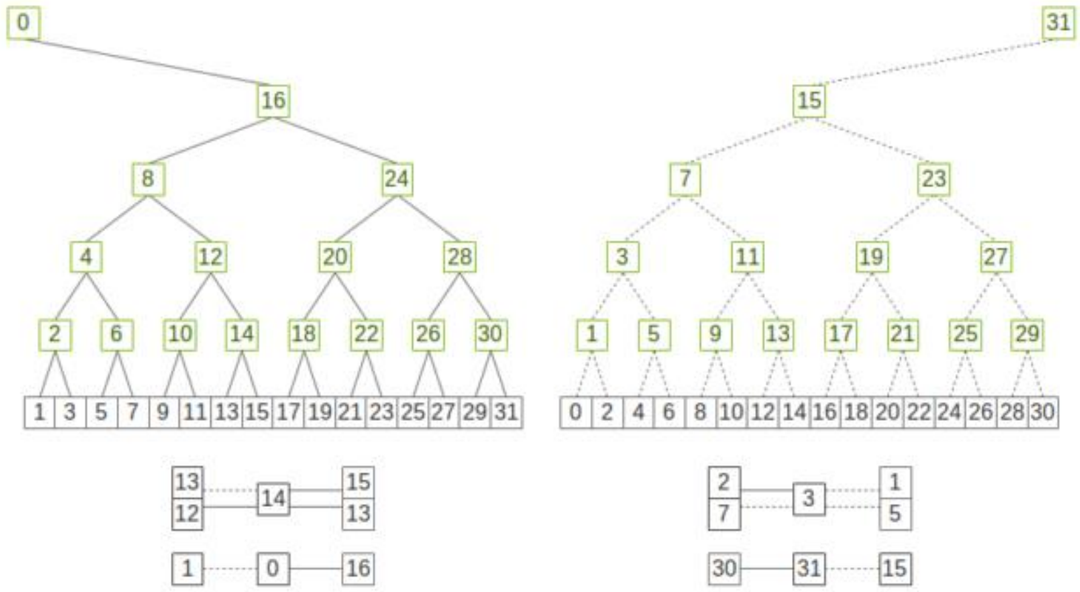
## Rings and Trees

### Rings




### Double binary Trees

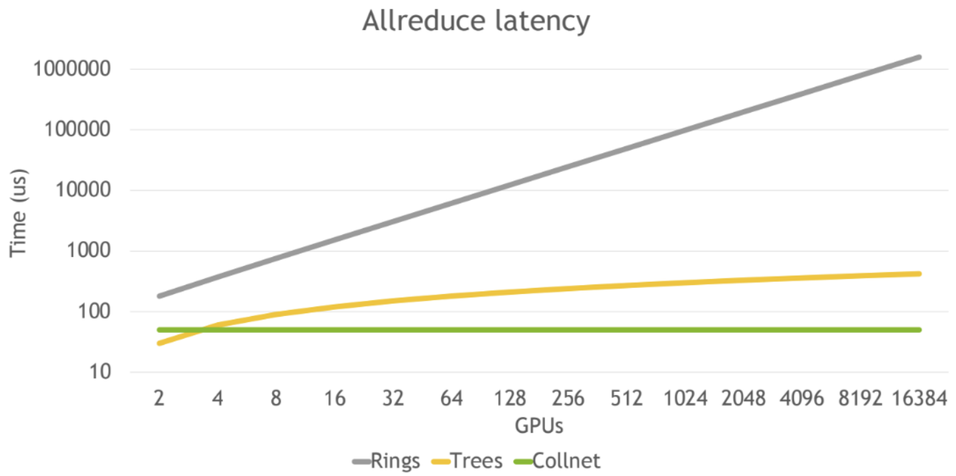
Two complementary binary trees where each rank is at most a node in one tree and a leaf in the other.

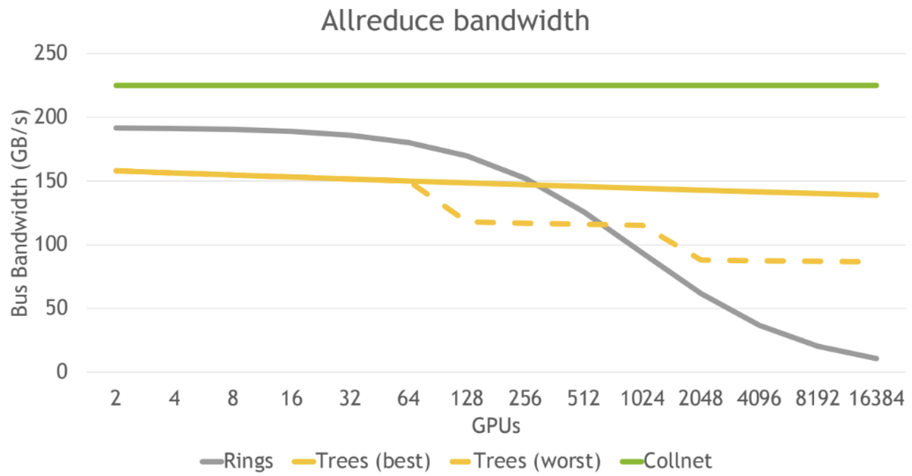



https://eng.uber.com/horovod/
https://developer.nvidia.com/blog/massively-scale-deep-learning-training-nccl-2-4/

# MULTI-GPU TRAINING

| Algorithm | Latency | Bandwidth | Computing | Network Pattern |
|---|---|---|---|---|
| Rings | Linear | Perfect | Uniform | Few flows |
| Trees | Log | Close to perfect | Imbalanced | Many flows |
| Collnet | Constant | Close to 2x (may be limited by NVLink) | Almost uniform | Minimal flows |

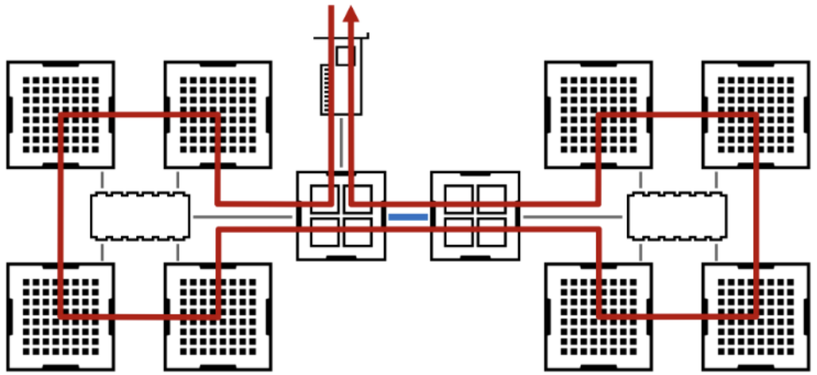


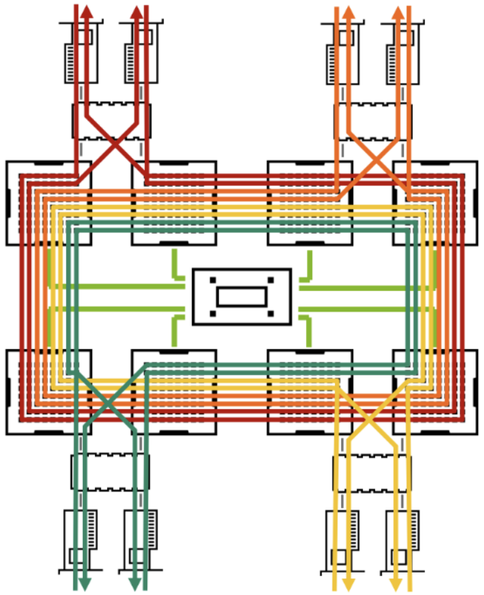

*Theoretical models on DGX A100*

# MULTI-GPU TRAINING

## Topology Detection- Mapping rings to the hardware



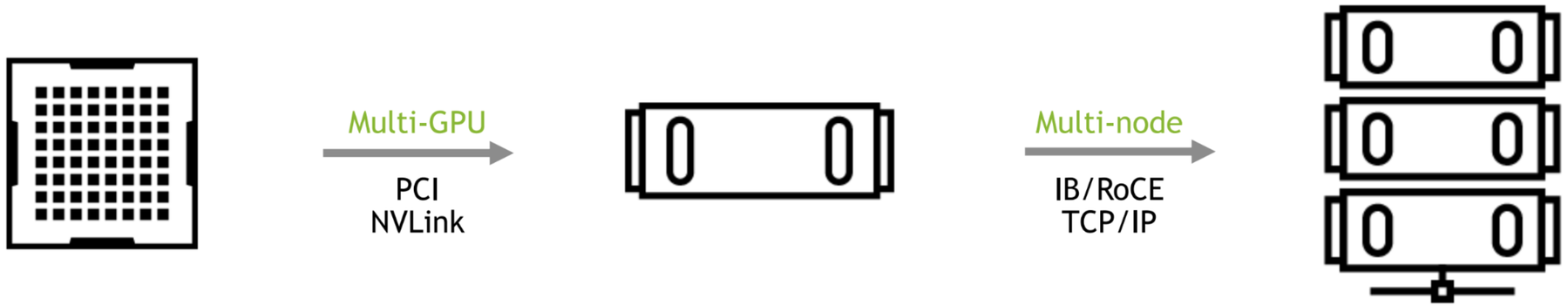
PCI platform

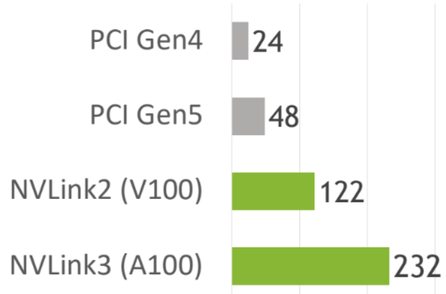


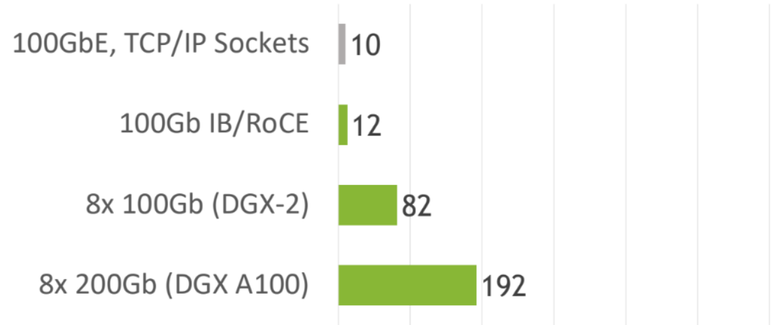
NVSwitch
(DGX A100)

# MULTI-GPU TRAINING

## Example: ALLREDUCE PERFORMANCE



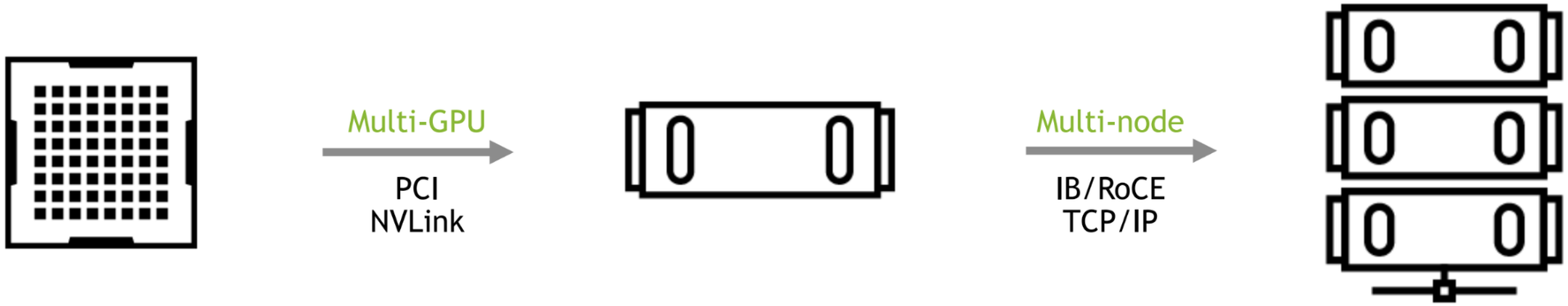



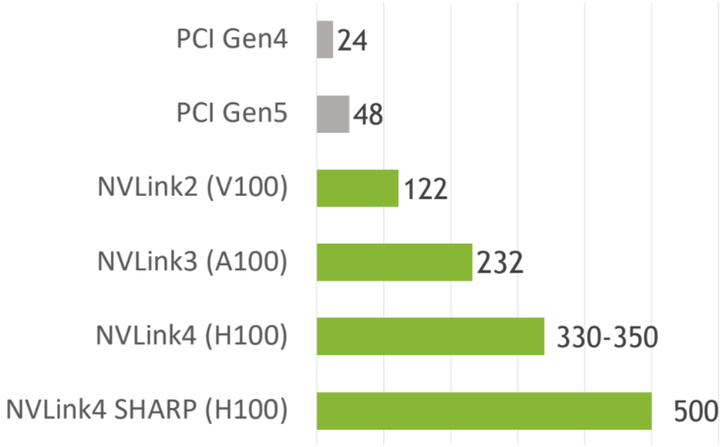



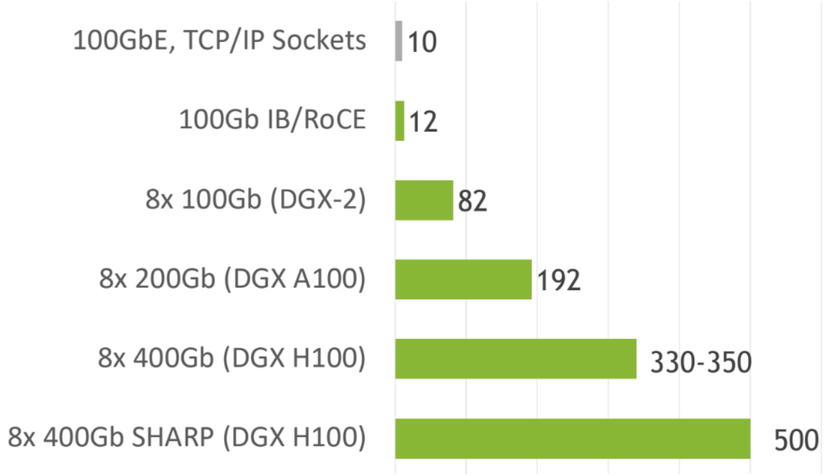

NCCL tests Allreduce Bus Bandwidth in GB/s.

# MULTI-GPU TRAINING

## ALLREDUCE PERFORMANCE



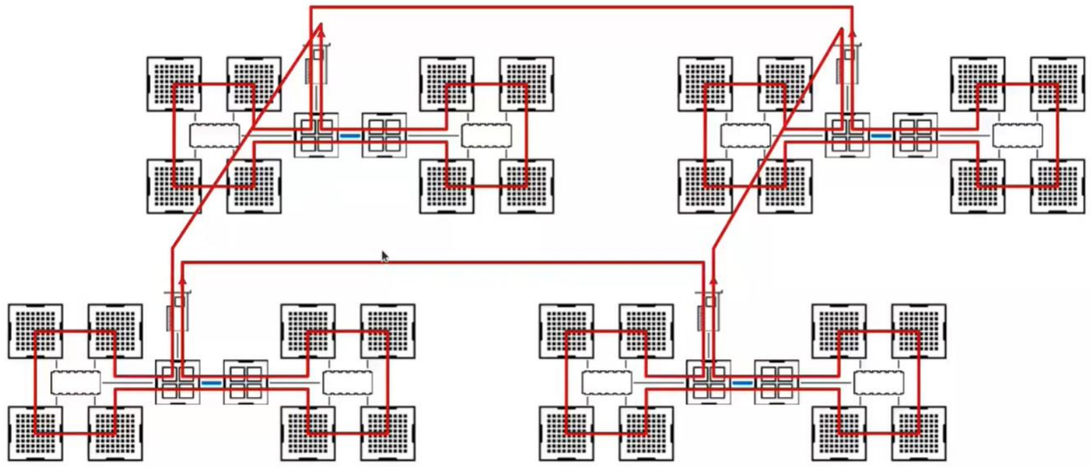



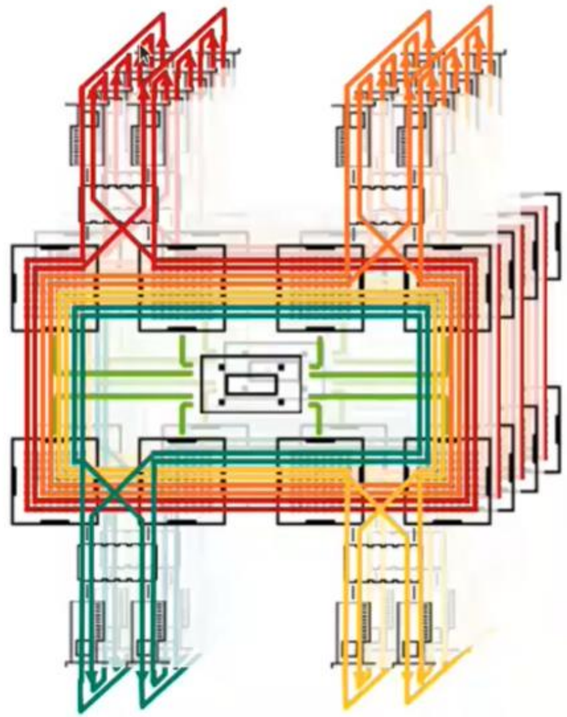

NCCL tests Allreduce Bus Bandwidth in GB/s.

# MULTI-GPU TRAINING

## Internodes Communication



PCI platform

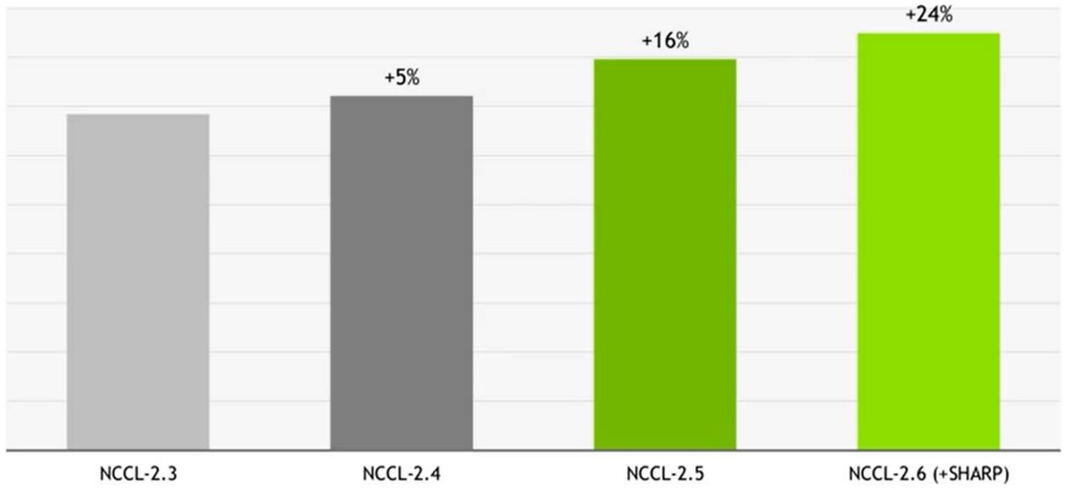


NVSwitch
(DGX A100)

# MULTI-GPU TRAINING

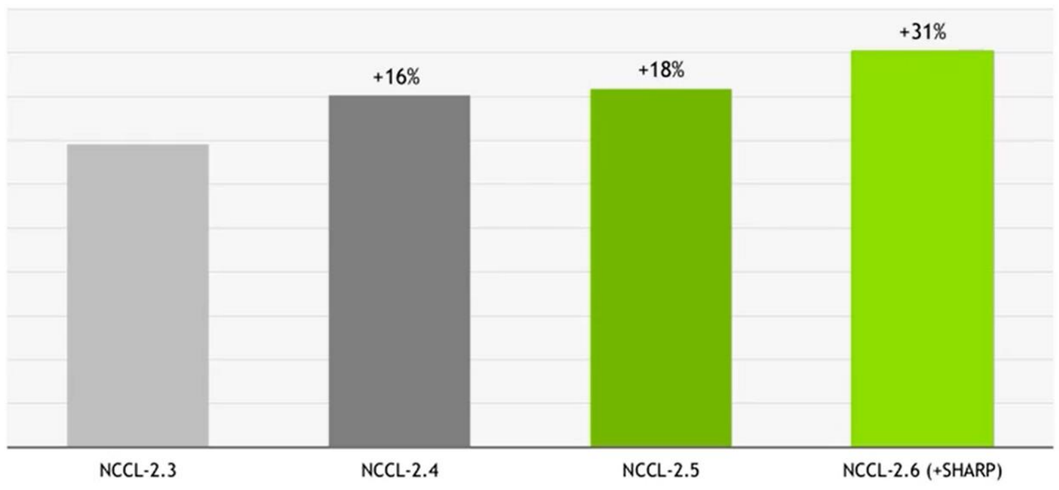
## NCCL

GNMT




*24xDGX1V + 4xMellanox CX-6, GNMT benchmark: Batch Size=32, Overlap=0.15*

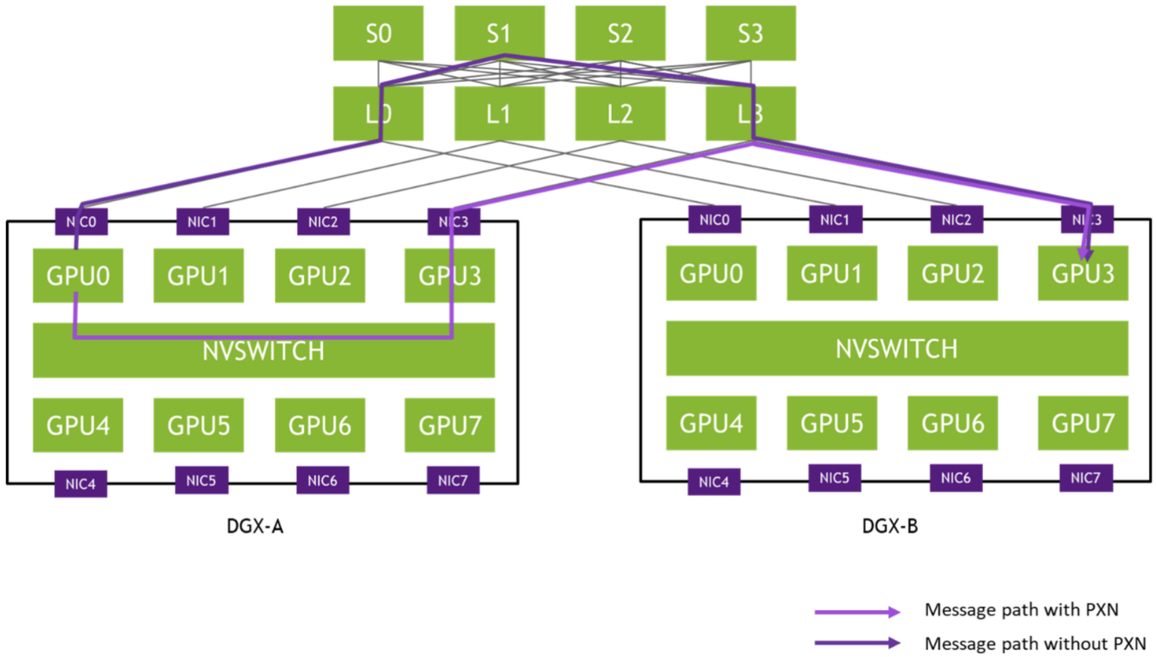
Transformer




*32xDGX1V + 4xMellanox CX-6, Transformer benchmark: Batch Size=640, Overlap=0.20*

# MULTI-GPU TRAINING

## NCCL PXN (v2.12)



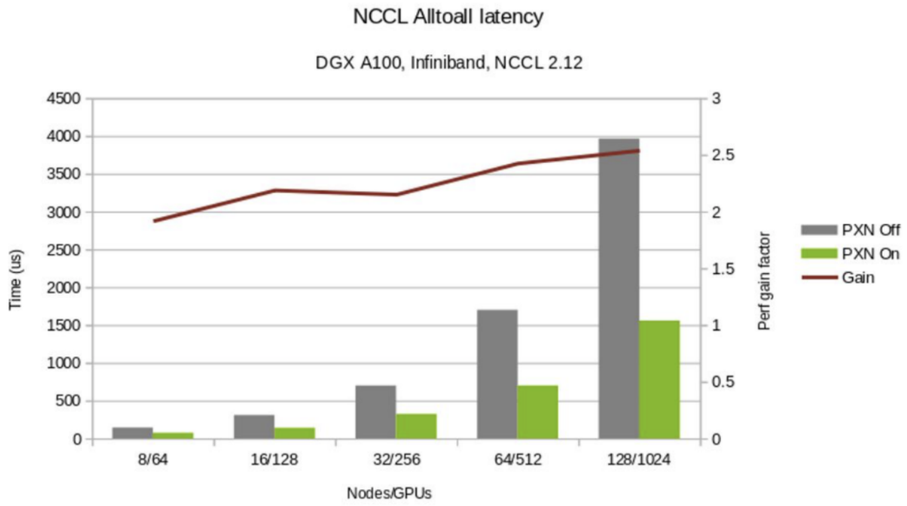




*Figure 3. NCCL 2.12 PXN performance improvements*

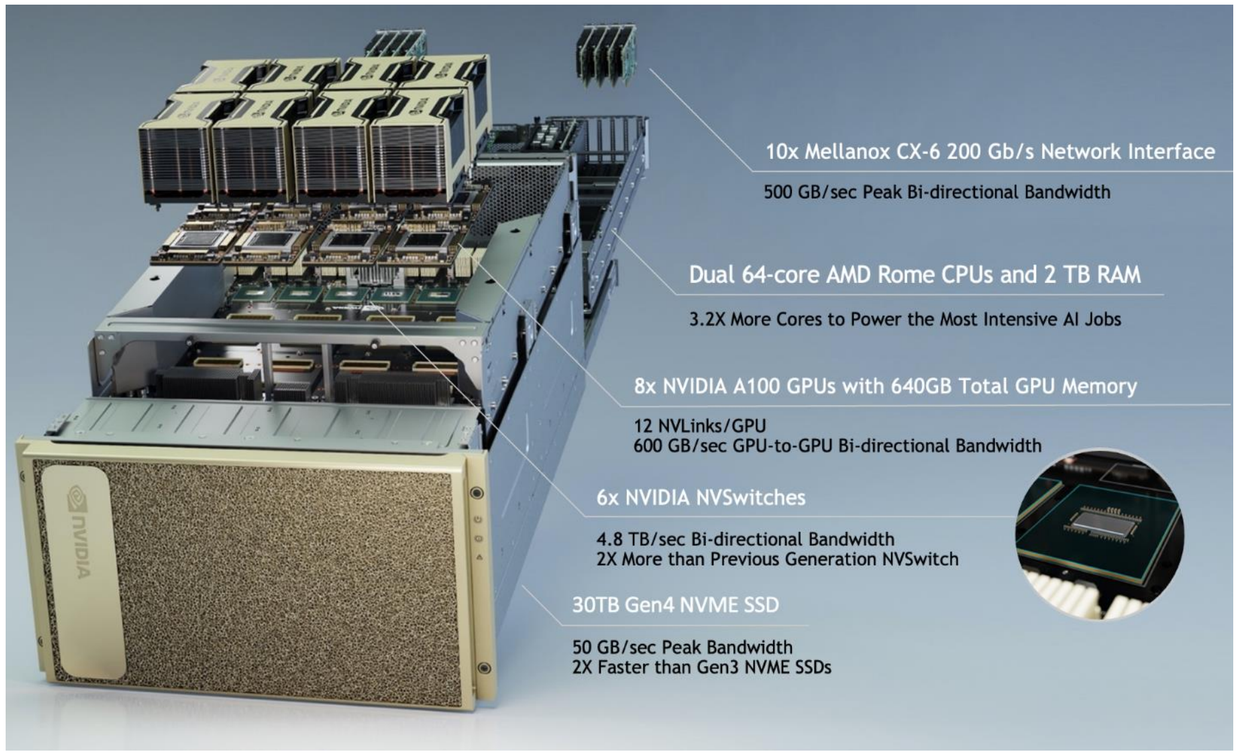
https://developer.nvidia.com/blog/doubling-all2all-performance-with-nvidia-collective-communication-library-2-12/

# HARDWARE DESIGN

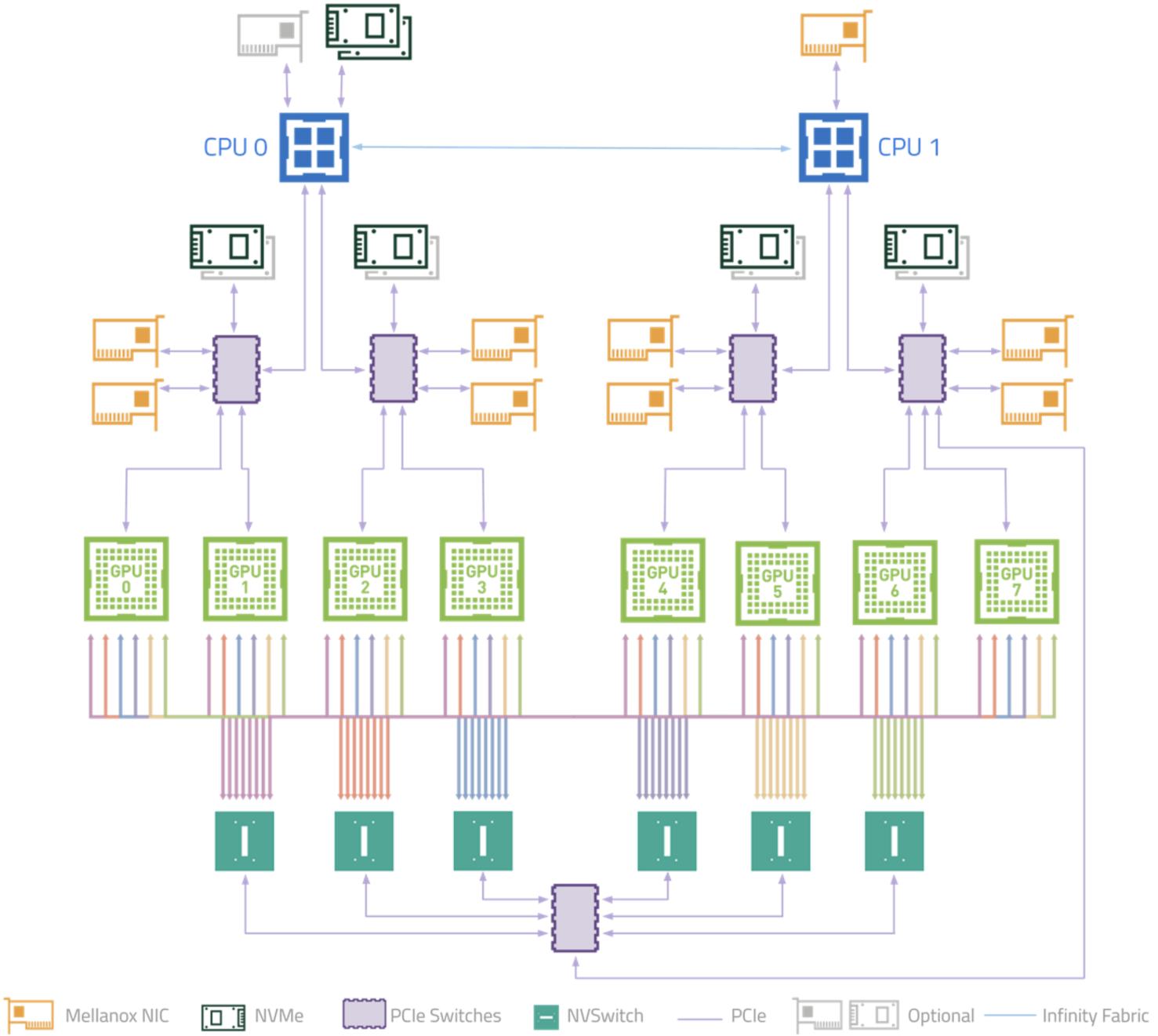
# A NEW GENERATION OF SYSTEMS

## NVIDIA DGX A100 640G

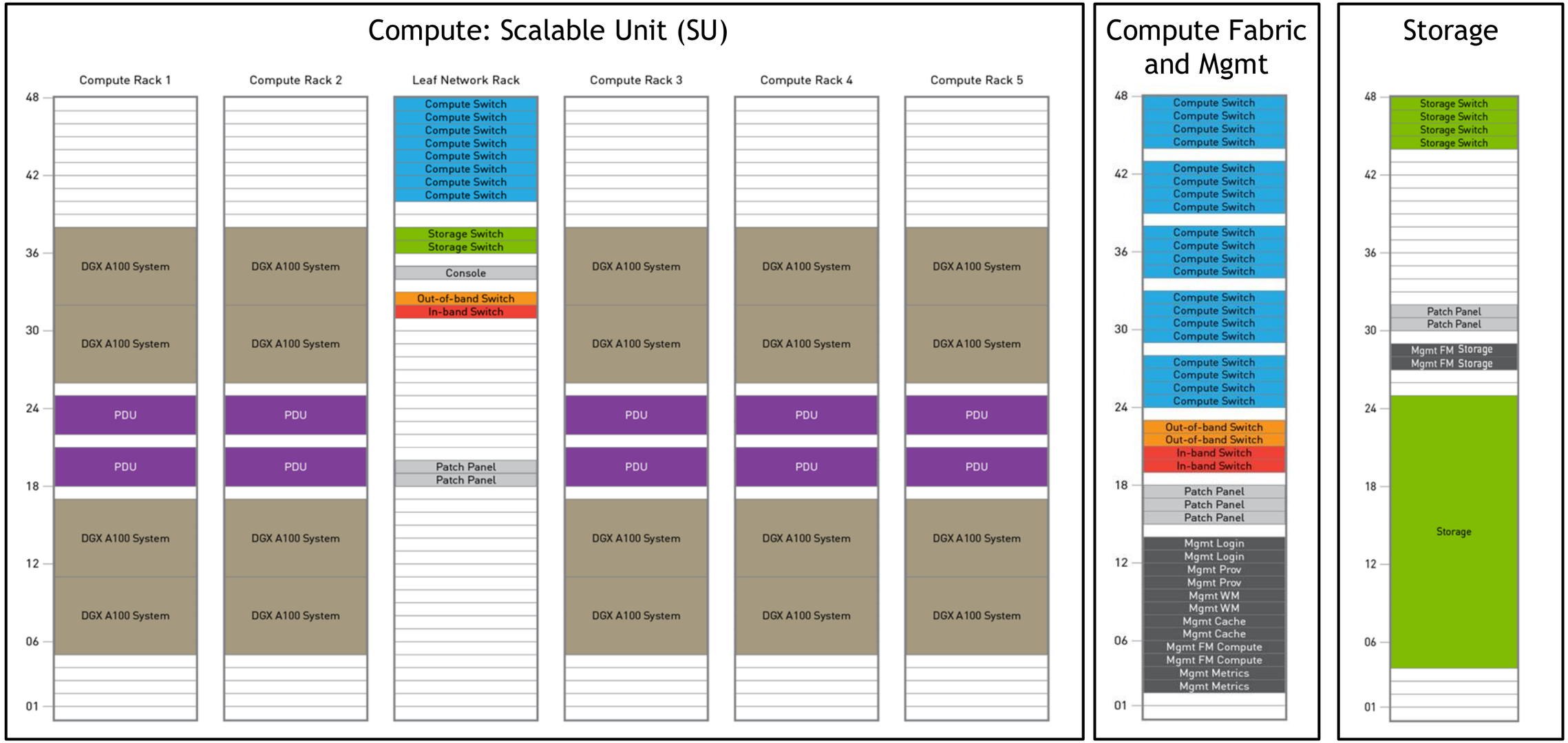
| | |
|---|---|
| GPUs | **8x NVIDIA A100 80GB GPUs** |
| GPU Memory | 640 GB total |
| Peak performance | 5 petaFLOPS AI \| 10 petaOPS INT8 |
| NVSwitches | 6 |
| System Power Usage | **6.5kW max** |
| CPU | **Dual AMD Rome 7742**<br>128 cores total, 2.25 GHz(base), 3.4GHz (max boost) |
| System Memory | **2TB** |
| Networking | **8x Single-Port Mellanox ConnectX-6** 200Gb/s HDR InfiniBand (**Compute Network**)<br>**2x Dual-Port Mellanox ConnectX-6** 200Gb/s HDR InfiniBand (**Storage Network** also used for Eth*) |
| Storage | OS: 2x 1.92TB M.2 NVME drives<br>Internal Storage: 30TB (8x 3.84TB) U.2 NVME drives |
| Software | Ubuntu Linux OS (**5.3+ kernel**) |
| System Weight | 271 lbs (123 kgs) |
| Packaged System Weight | 360 lbs (163 kgs) |
| Height | **6U** |
| Operating temp range | **5°C to 30°C** (41°F to 86°F) |



# NVIDIA DGX A100 System



# MODULARITY: RAPID DEPLOYMENT



# DGX SUPERPOD

## Modular Architecture

### 1K GPU SuperPOD Cluster

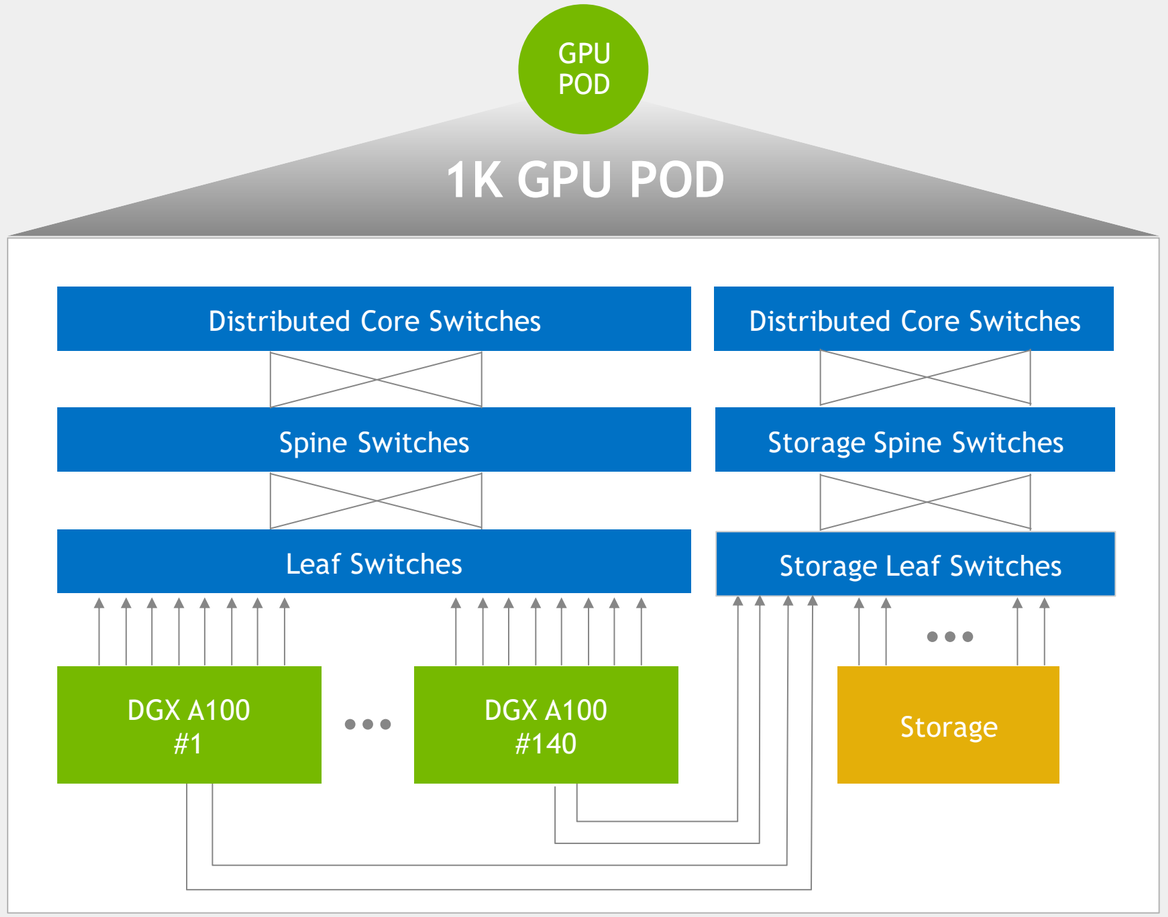
- 140 DGX A100 nodes (1,120 GPUs) in a GPU POD
- 1st tier fast storage - DDN AI400x with Lustre
- Mellanox HDR 200Gb/s InfiniBand - Full Fat-tree
- Network optimized for AI and HPC

### DGX A100 Nodes

- 2x AMD 7742 EPYC CPUs + 8x A100 GPUs
- NVLINK 3.0 Fully Connected Switch
- 8 Compute + 2 Storage HDR IB Ports

### A Fast Interconnect

- Modular IB Fat-tree
- Separate network for Compute vs Storage
- Adaptive routing and SharpV2 support for offload

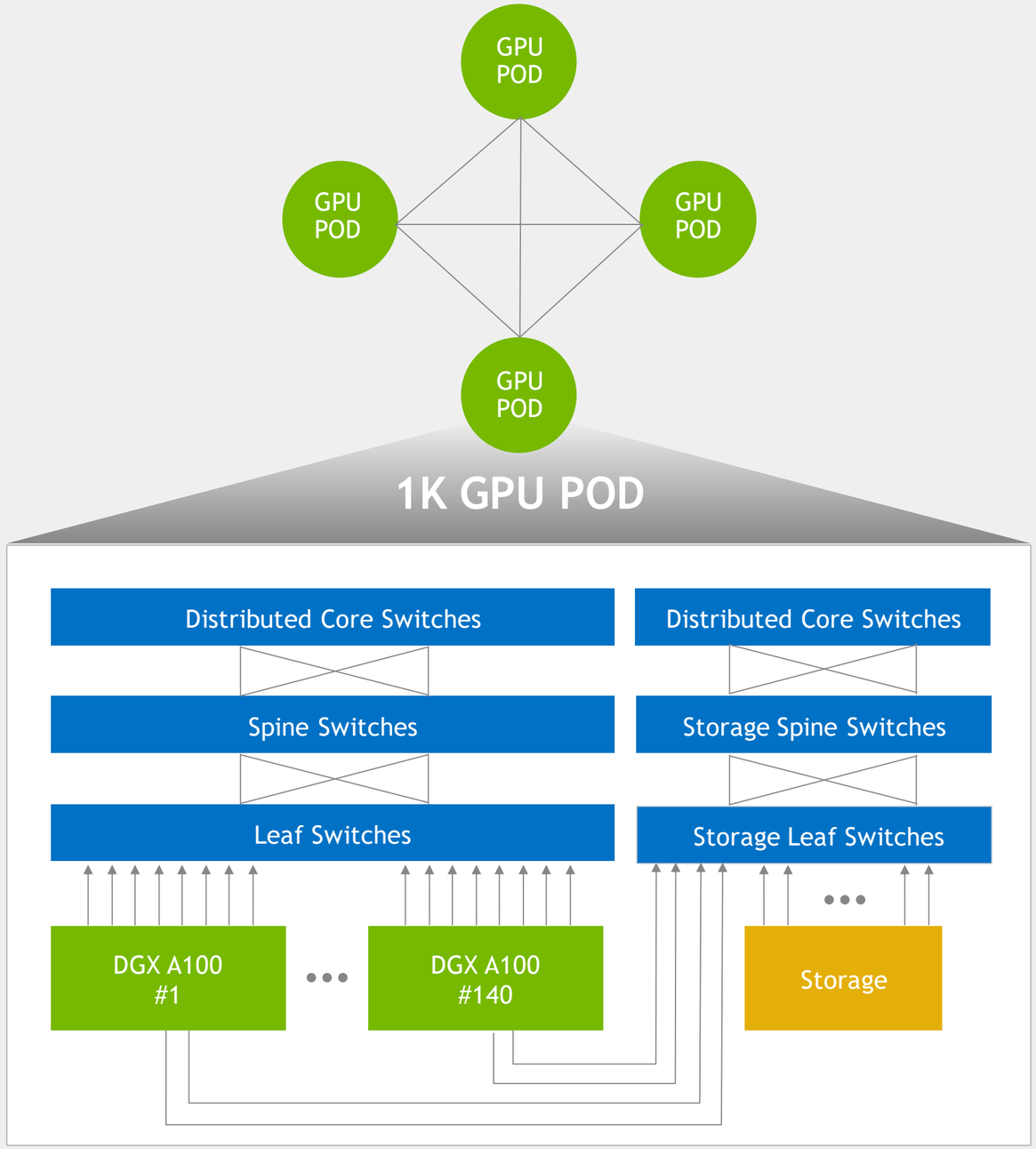


# DGX SUPERPOD

## Extensible Architecture

POD to POD

- Modular IB Fat-tree or DragonFly+
  - Core IB Switches Distributed Between PODs
  - Direct connect POD to POD



# NEMO-MEGATRON

# NEMO-MEGATRON WITH DGX SUPERPOD

## Train what was once impossible

### Algorithmic innovation
Train the world's largest transformer-based language models using Megatron's advanced optimizations and parallelization algorithms.

### Direct access to world-class NLP experts
Access dedicated expertise from install to infrastructure management to scaling workloads to streamlined production AI.

### Optimized Topology for Multi-Node Training
Train the largest models using model parallelism, with NVLINK and InfiniBand for fast cross-node communication.

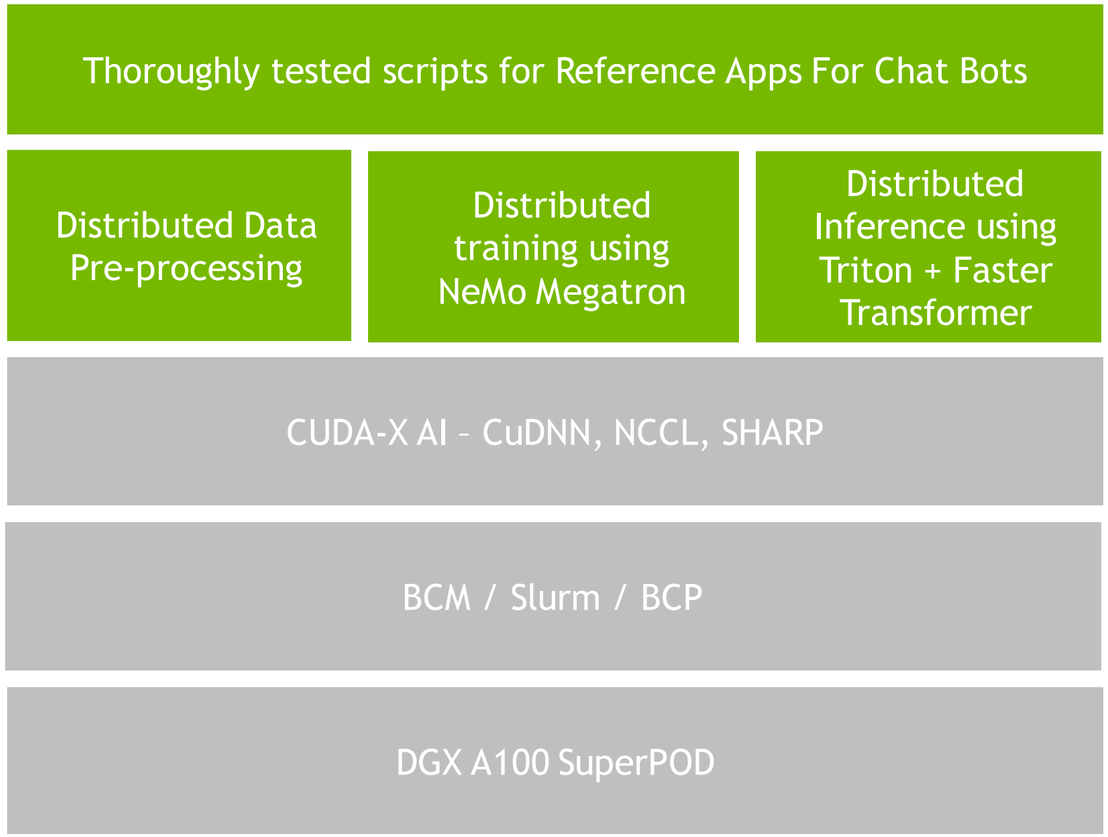
### Turnkey Experience for Rapid Deployment
A full-stack data center platform that includes industry-leading computing, storage, networking, software, and management tools.

### Efficiency at Extreme Scale
Training GPT-3 175B takes 355 years on a V100, 14.8 years on 1 DGX A100 and about 1 month on a 140-node DGX SuperPOD

# NEMO MEGATRON - EA

**Early access components:**

1. **Thoroughly tested:**
   - Pre-training recipes for GPT-3 (up to 1T parameters) and T5/mT5 (up to 50B models)
   - Evaluation scripts trained model on LM harness containing 8+ standard industry benchmarks
   - Inferencing scripts for GPT-3/T5 models
2. **Smart Tools:**
   - Hyperparameter tuning tool to automagically create training configuration with high probability of convergence given training constraints
   - Model navigator to automagically generate optimized inference configurations
3. Sample Chat Bot Application the demonstrates use of GPT-3 dialogue generation capability
4. Direct Access to NVIDIA NLP Experts through NVPS




**Updates since fall 2021**

- Added support for pipeline parallelism
- Ready to use recipes for 40B and 175B GPT-3 models
- Support for base command and available for trial on NVIDIA LaunchPad
- Sample downstream application - Chatbot
- W&B visualizations for training

**Things in pipeline:**

- Recipes for T5 and mT5
- Support for distributed data preprocessing

**Early Access Limitations:**

- Recipes and software stack is tuned for DGX A100 SuperPOD

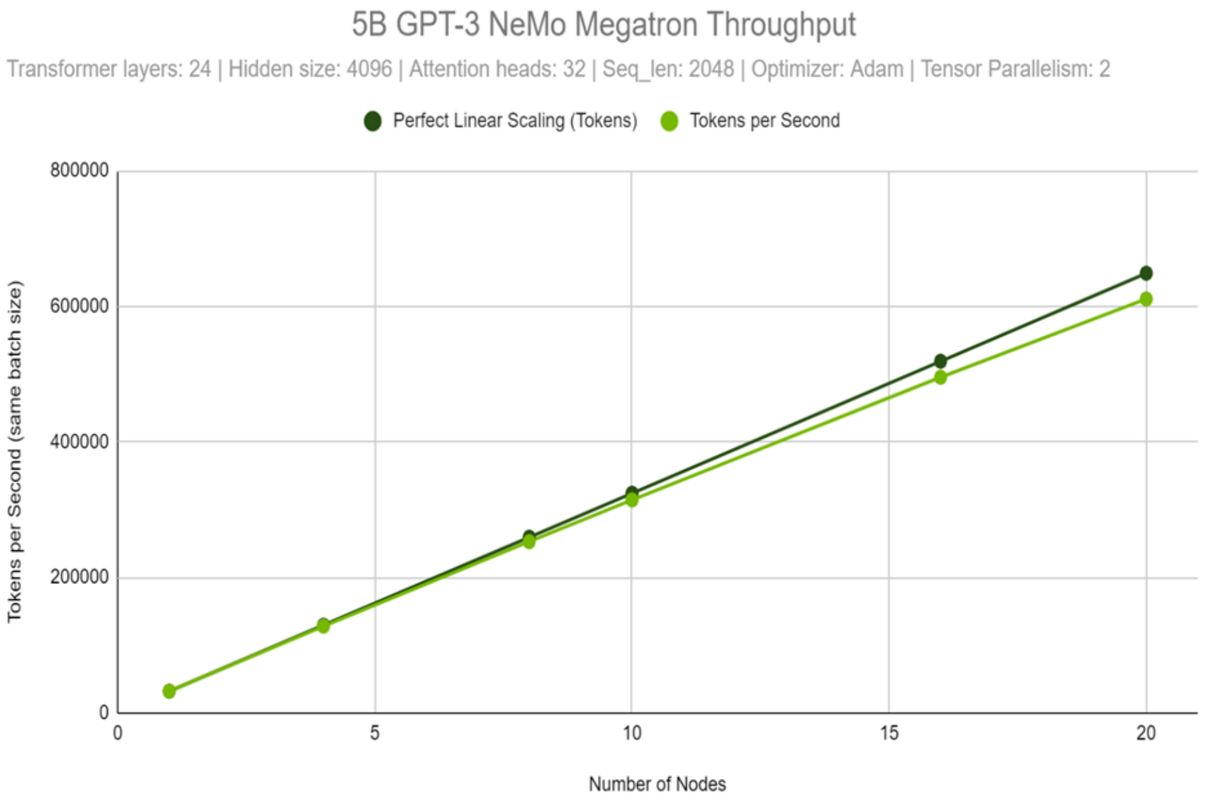
# NEMO-MEGATRON

## Training

**Goal:** Train transformer models with billions of parameters

**Current Capabilities:**

- Parallel layer implementation supported:
  - ColumnParallelLinear
  - RowParallelLinear
  - ParallelMLP
  - ParallelSelfAttention
- Optimizations supported:
  - Native AMP support for FP16 and BF16
  - NVFuser: bias-dropout-add and bias-gelu fusion
  - Partial Activation Checkpointing
  - Custom fused layernorm
  - Overlap all-reduces with WGRAD GEMM
  - Rank allocation optimization




**Exceptions/Limitations:**

- Support for memory footprint reduction
- Support for model compression

https://developer.nvidia.com/nemo-megatron-early-access

# THE LAB

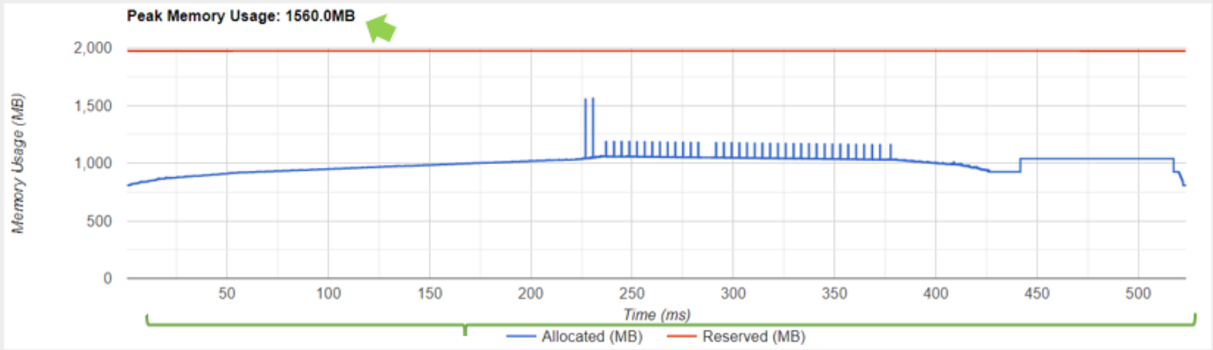
# LAB 2

## Overview

**Distributed Training for Computer Vision**

- Learn how to train a simple Image Classifier
- Implement a vanilla pipeline parallel distribution
- Port the code to DeepSpeed library
- Scale training using Data parallel distribution
- Optimize training with DeepSpeed autotuning and Zero Redundancy Optimizer
- Mixture of Experts

***No ZeRO***

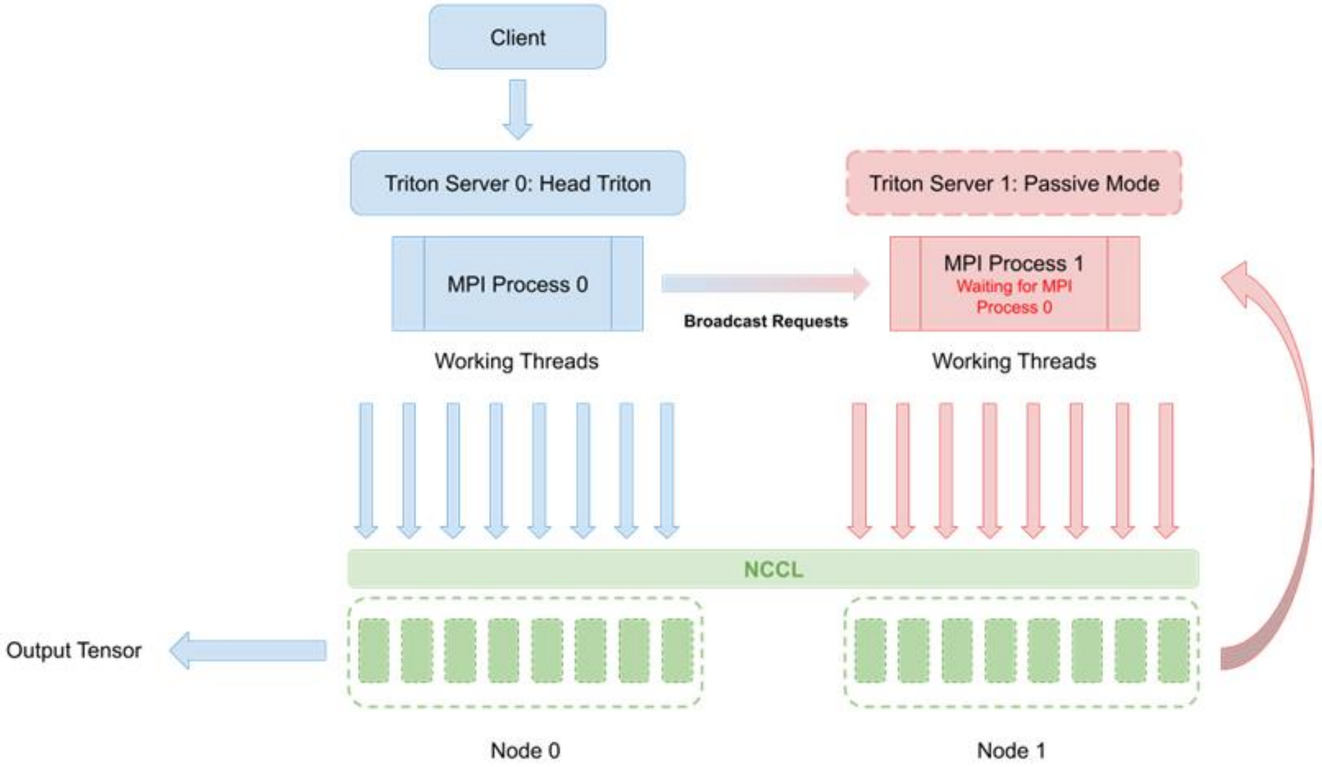



***ZeRO Stage 3 + Offload***

# IN THE NEXT CLASS

# NEXT CLASS

## Discuss large model deployment challenges and solutions




FasterTransformer and Triton Inference Server for Multi-Node Inference for large scale Transformer Model