Large Language Model Distributed Training

October 18, 2024

Objectives

By the end of this workshop, you will be able to:

- Outline reasons to train models using more than one GPU.
- Understand different GPU collective communication primitives and their role in each parallel technique.
- Understand different parallelization techniques for distributed LLM training using GPUs.
- Train different transformers using OLMo (AI2 Open Large Language Model) in a distributed fashion on the HPC cluster.

Agenda

1. Why Going Distributed?
2. Intro to Distributed GPU Computing
3. Different Distributed LLM Training Techniques
4. Training a Large LLM on the Cluster

Why Distributed?

Data Size, Model Size or Both?

- Speedup Training
  - Model fits into a single GPU
  - Have a huge dataset to process
  - More GPUs - More computing resources
- Train Larger Models
  - We need to divide the model across multiple GPUs to be able to train it.

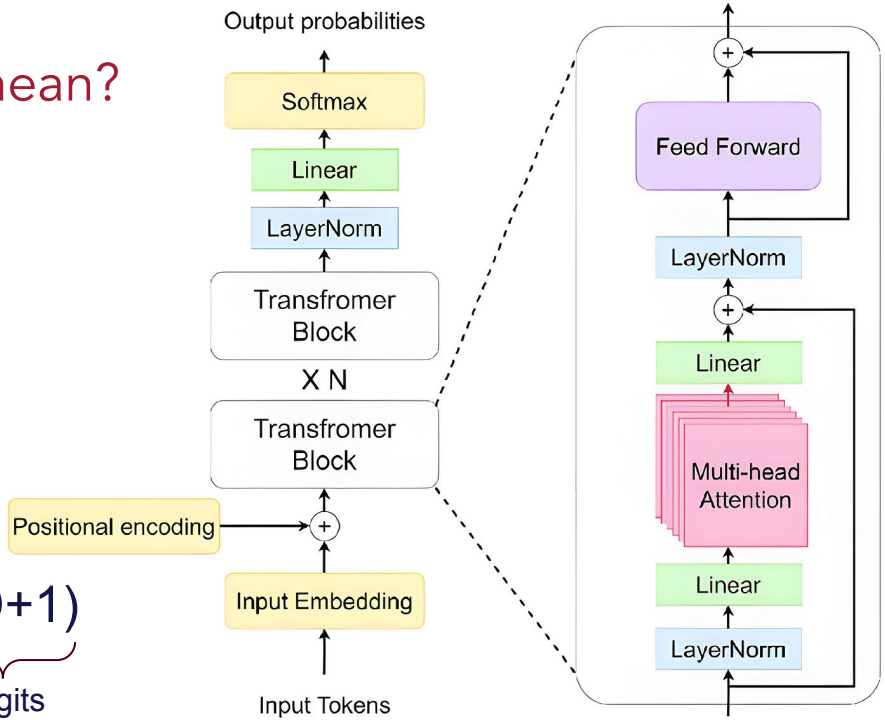
Decoder-only LLM Architecture

What do 1B, 7B, 70B, … LLM sizes mean?

- V: vocabulary size
- B: batch size
- D: model dimension
- T: sequence length
- N: number of transformer blocks
- N_h: number of attention heads

$$P = N(12D^2 + 10D) + D(V+T) + 2D + V(D+1)$$

Transformer Block | Embedding | Last Norm | Logits



Required Memory Estimation

Memory Contributors:

- Parameters
- Optimizer states
- Activations
- Others (Input/Target)

$$M_{parameters} = P$$

$$M_{optimizer} = 3P$$

$$M_{activations} = \underbrace{5NBTD}_{\text{FF}} + \underbrace{BNN_h T^2}_{\text{attention}} + \underbrace{2BVT}_{\text{embed}} + \underbrace{2BTD}_{\text{LNs, head}}$$

$$M_{other} = 2BT$$

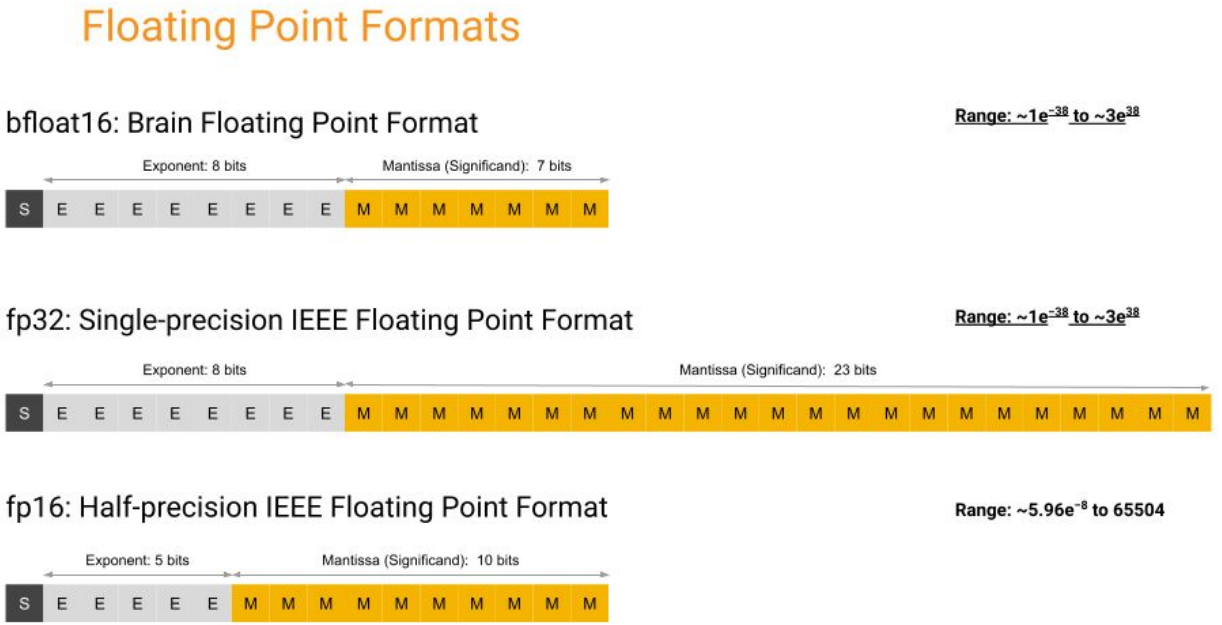
Total Memory Requirement:

$$M = M_{parameters} + M_{optimizer} + M_{activations} + M_{other}$$

- V: vocabulary size
- B: batch size
- D: model dimension
- T: sequence length
- N: number of transformer blocks
- N_h: number of attention heads

$$P = N(12D^2 + 10D) + D(V+T) + 2D + V(D+1)$$

Floating-point Formats



https://www.tensorops.ai/post/what-are-quantized-llms

Required Memory Estimation

Just Loading Weights

- Parameters in bfloat16 (2 Bytes)
- Optimizer in float32 (4 Bytes)

$M_{model} = P * 2 + 3P * 4 = 14\ P$ (Bytes)

GPUs:

H100 ⇒ 80 G
A100 ⇒ 40 G

Exercise: Assessing Memory Needs

One of the OLMo models we will be working with has the hyperparameters shown on the right.

- V (vocabulary size): 50280
- B (batch size): 256
- D (model dimension): 8192
- T (sequence length): 4096
- N (number of transformer blocks): 80
- N_h (number of attention heads): 64

1) Compute the number of parameters
2) Compute the memory that the model weights will take
3) Will this fit on one GPU? If not, how many A100 GPUs would you need? How many H100s?

GPUs:

H100 ⇒ 80 G
A100 ⇒ 40 G

Exercise Solution

$P = N(12D^2 + 10D) + D(V+T) + 2D + V(D+1)$

P = 80*(12*8192**2 + 10 *8192) +
8192*(50280 + 4096) + 2*8192 +
50280*(8192+1)

P = 65288471656

P = 65 billion

M = 14P = 14*65288471656

M = 910 billion bytes = 910 GB

- V (vocabulary size): 50280
- B (batch size): 256
- D (model dimension): 8192
- T (sequence length): 4096
- N (number of transformer blocks): 80
- N_h(number of attention heads): 64

GPUs:

H100 ⇒ 80 G
A100 ⇒ 40 G

Required Memory Estimation

Just Loading Weights

- Parameters in bfloat16 (2 Bytes)
- Optimizer in float32 (4 Bytes)

$M_{model} = P * 2 + 3P * 4 = 14\ P$ (Bytes)

Model Size (P)	Approx. memory used to train model (GB)
300M	4
1B	14
7B	98
13B	182
70B	980

GPUs:

H100 ⇒ 80 G
A100 ⇒ 40 G

Parameter Comparisons

	Vocabulary size (V)	Model dimension (D)	Sequence length (T)	Number of transformer blocks (N)	Number of attention heads (N_h)
OLMo 1B	32100	2046	2048	16	16
OLMo 7B	32100	4096	2048	32	32
OLMo 70B	50280	8192	4096	80	64

OLMo

Open Language Model

- A highly performant, truly open LLM and framework
  - 100% of ingredients are available to public including code, weights, checkpoints, training data and system logs.
  - To advance AI and study language models collectively
  - Decoder-only Architecture

Let's setup and install it on the cluster:
https://github.com/KempnerInstitute/OLMo/blob/main/README_KempnerInstitute.md

To request Kempner GPUs if you don't have access to Kempner cluster:
https://handbook.eng.kempnerinstitute.harvard.edu/s1_high_performance_computing/kempner_cluster/accessing_gpu_by_fasrc_users.html

Agenda

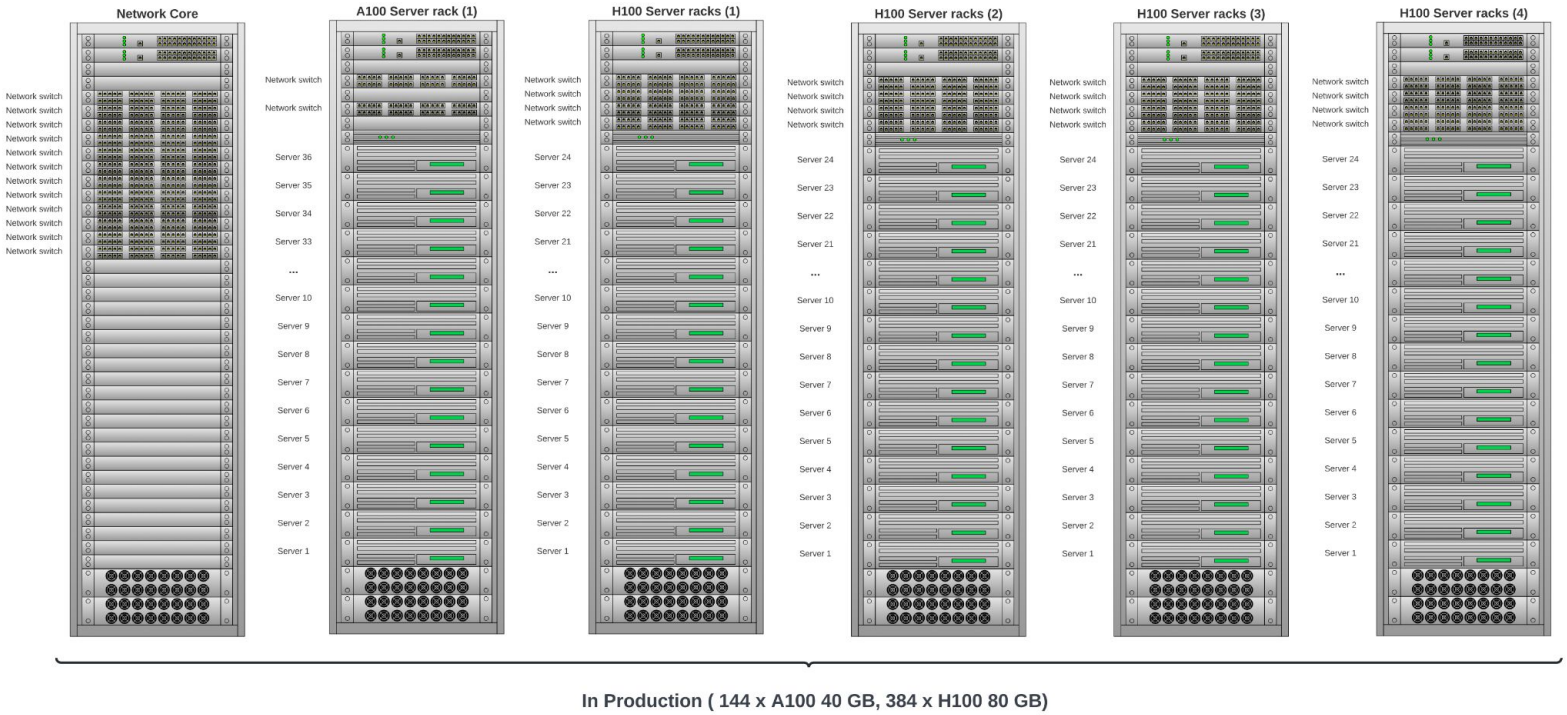
1 Why Going Distributed?

2 Intro to Distributed GPU Computing

3 Different Distributed LLM Training Techniques

4 Training a Large LLM on the Cluster

HPC Cluster


Network Core
A100 Server rack (1)
H100 Server racks (1)
H100 Server racks (2)
H100 Server racks (3)
H100 Server racks (4)
Network switch
Server 36
Server 35
Server 34
Server 33
Server 24
Server 23
Server 22
Server 21
Server 10
Server 9
Server 8
Server 7
Server 6
Server 5
Server 4
Server 3
Server 2
Server 1
In Production (144 x A100 40 GB, 384 x H100 80 GB)

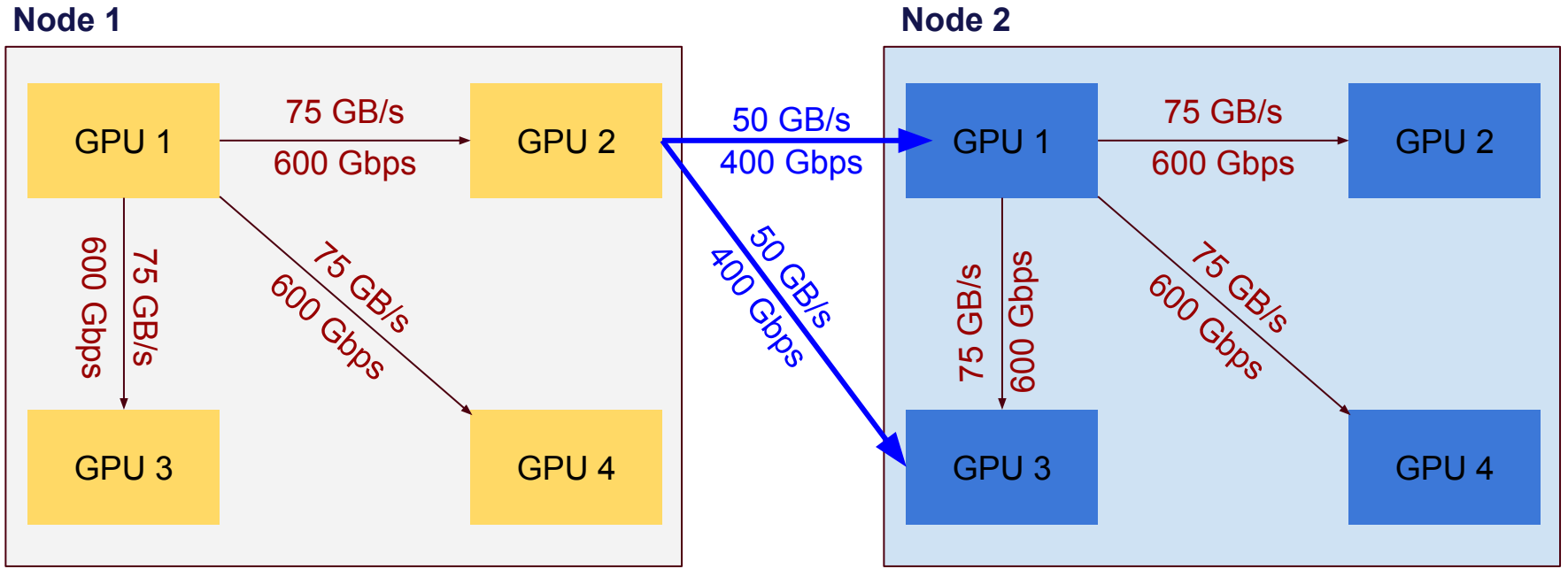
HPC Cluster - Computational Power

Compute Power (H100 GPUs)

- Number of Racks (N_{Rack}) = 4
- Number of Nodes per Rack (N_{Node}) = 24
- Number of GPUs per Node (N_{GPU}) = 4
- Total Number of GPUs = N_{Rack} x N_{Node} x N_{GPU} = 384 H100 GPUs
- Total Computational Power in FLOPs,
  - Total FLOPs (BFLOAT16 Tensor Core) = 384 GPU x 1979 TFLOPs / GPU = 759,936 TFLOPs = 759 PFLOPs
  - Total FLOPs (FP32) = 384 GPU x 67 TFLOPs / GPU = 25,728 TFLOPs = 25 PFLOPs

Technical Specifications	
	H100 SXM
FP64	34 teraFLOPS
FP64 Tensor Core	67 teraFLOPS
FP32	67 teraFLOPS
TF32 Tensor Core*	989 teraFLOPS
BFLOAT16 Tensor Core*	1,979 teraFLOPS
FP16 Tensor Core*	1,979 teraFLOPS
FP8 Tensor Core*	3,958 teraFLOPS
INT8 Tensor Core*	3,958 TOPS

GPU-to-GPU Communication



Inside Node (**NVLINK**): Each GPU talks to other three GPUs at 75 GB/s (single direction). This sums up to 900 GB/s all GPU-GPU bidirectional speed. 75 GB/s * 6 * 2 = 900 GB/s

Outside Node (**InfiniBand Network NDR**): Each GPU communicates to other GPUs in another node at 400 Gbps (50 GB/s).

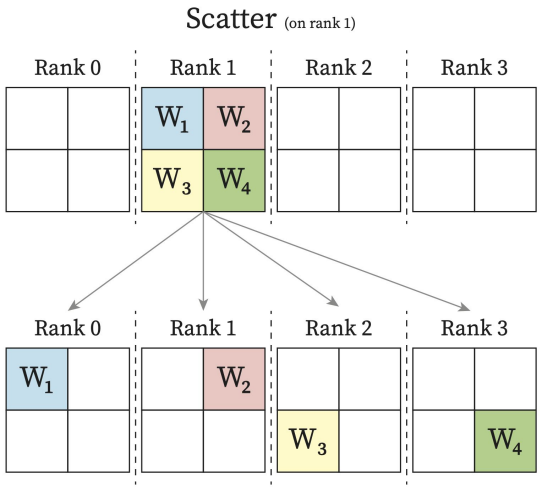
Inter-GPU Communication

NCCL

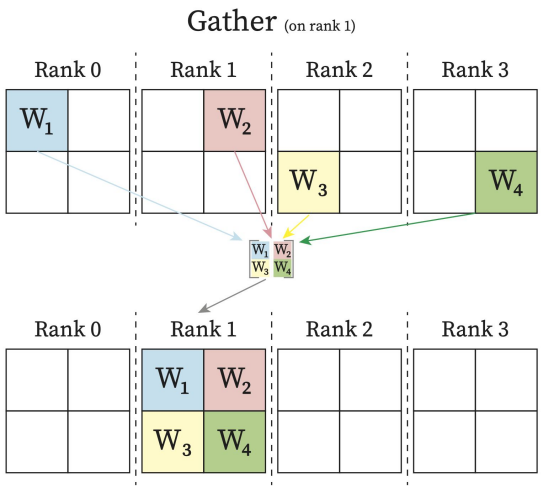
NVIDIA Collective Communication Library (NCCL, pronounced “NICKEL”) is used as backend in distributed strategies for NVIDIA GPUs

NCCL offers various collective communication primitives

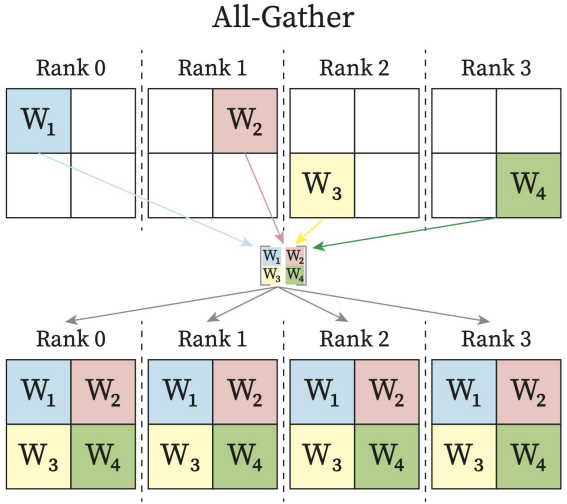
Other NCCL Collective Primitives



Scatter: From one rank data will be distributed across ranks.

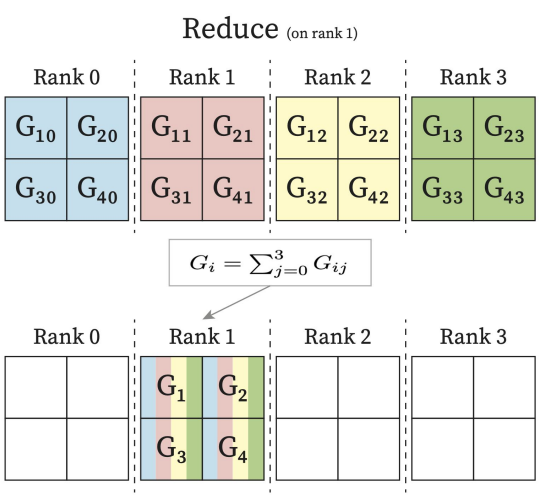


Gather: One rank will receive the aggregation of data from all ranks.

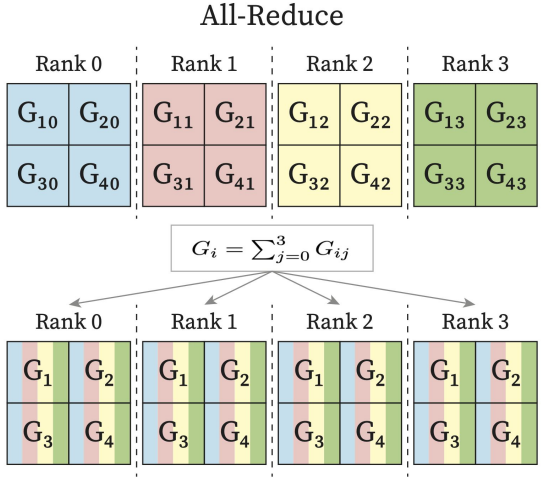


All-Gather: Each rank receives the aggregation of data from all ranks in the order of the ranks.

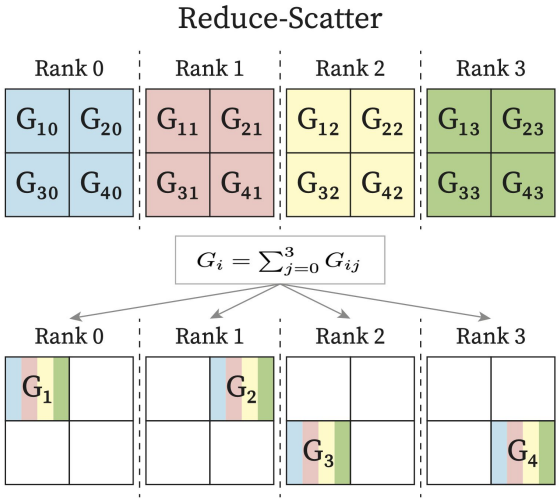
Other NCCL Collective Primitives



Reduce: One rank receives the reduction of input values across ranks.



All-Reduce: Each rank receives the reduction of input values across ranks.



Reduce-Scatter: Input values are reduced across ranks, with each rank receiving a subpart of the result.

Agenda

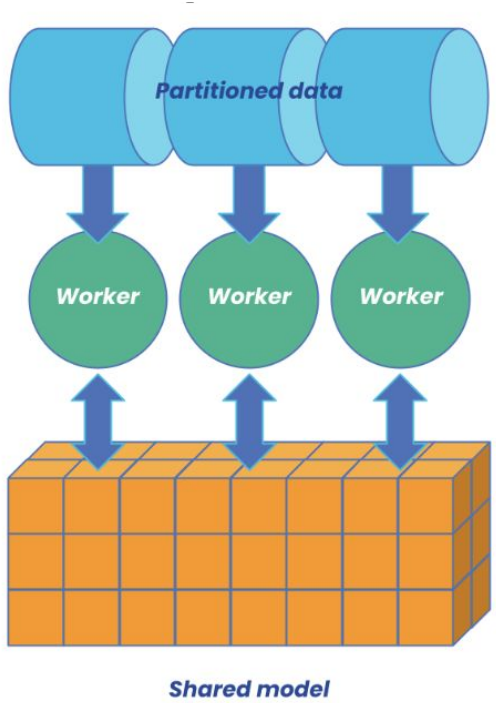
1 Why Going Distributed?

2 Intro to Distributed GPU Computing

3 Different Distributed LLM Training Techniques

4 Training a Large LLM on the Cluster

Distributed Data Parallel Processing

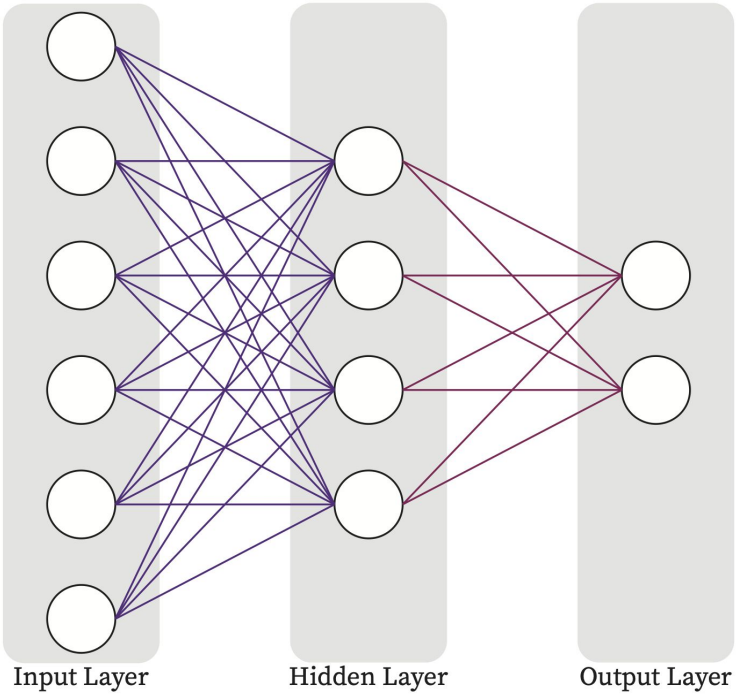


Most common approach to distributed training in machine learning

Each GPU trains a copy of the model. Dataset is split into different batches of data on each GPU

https://www.anyscale.com/blog/what-is-distributed-training

Multi-layer Perceptron



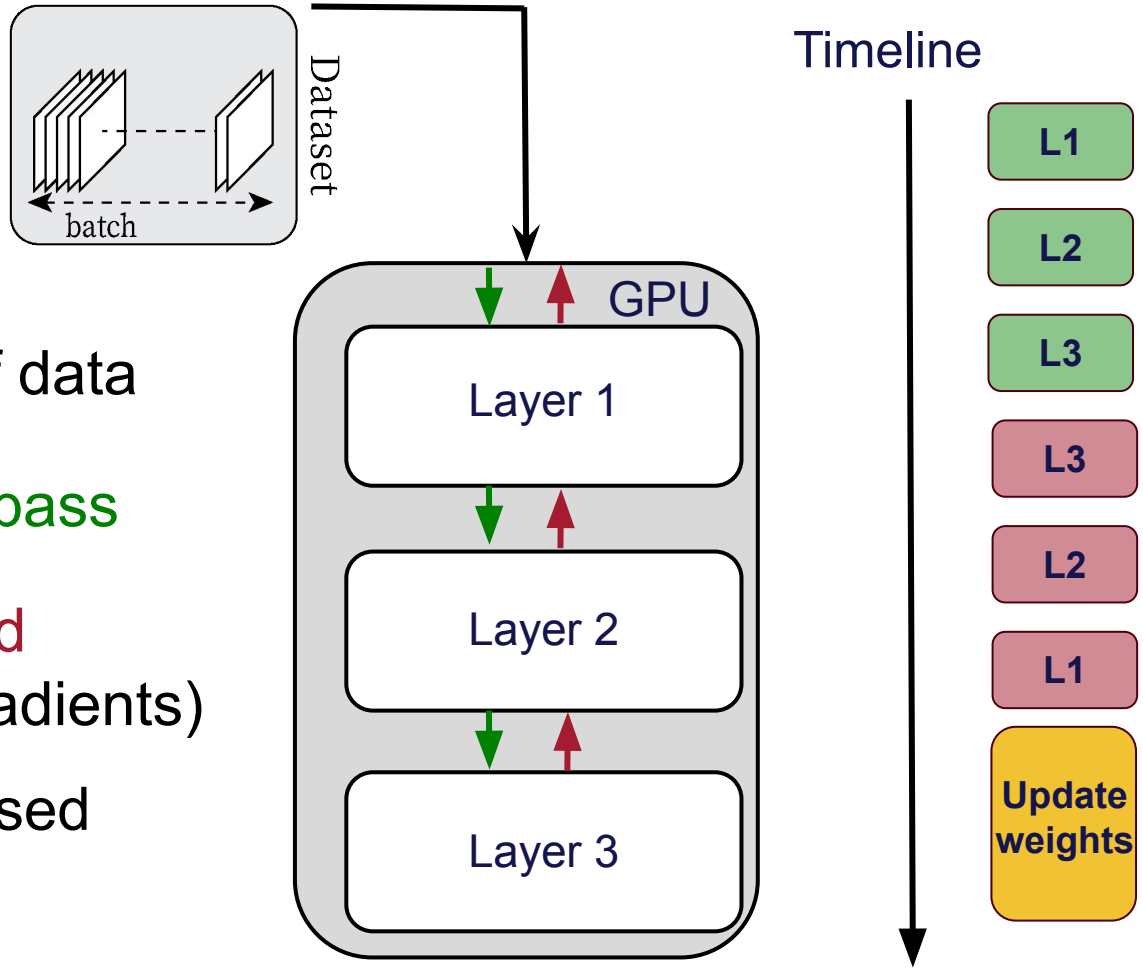

```python
class MLP(nn.Module):
  def __init__(self, in_feature, hidden_units, out_feature):
    super().__init__()

    self.hidden_layer = nn.Linear(in_feature, hidden_units)
    self.output_layer = nn.Linear(hidden_units, out_feature)

  def forward(self, x):
    x = self.hidden_layer(x)
    x = self.output_layer(x)
    return x
```

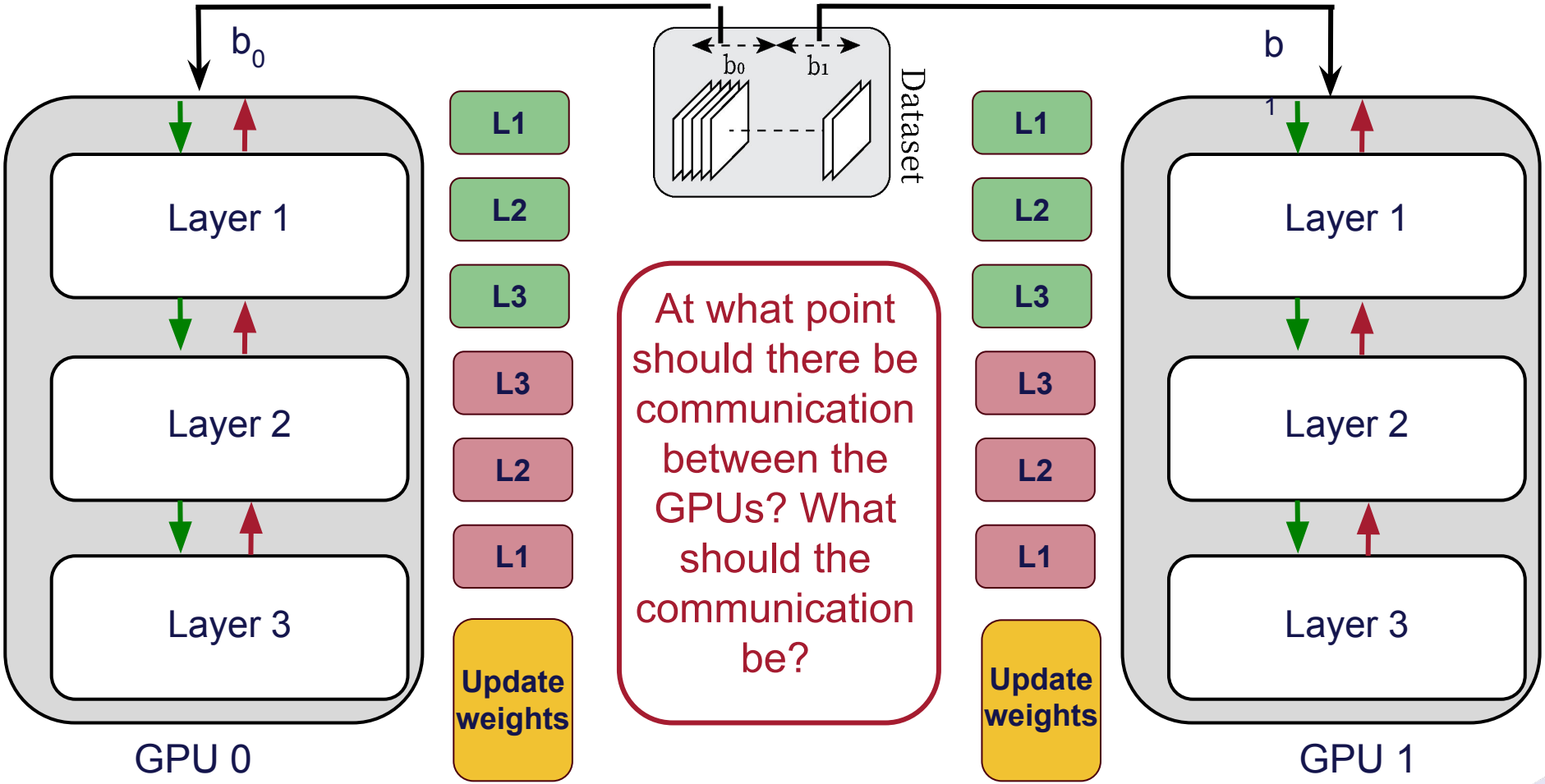
$$x_i = [x_1\ x_2\ x_3\ x_4\ x_5\ x_6] \qquad W_1 = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \\ w_5 & w_6 & w_7 & w_8 \\ w_9 & w_{10} & w_{11} & w_{12} \\ w_{13} & w_{14} & w_{15} & w_{16} \\ w_{17} & w_{18} & w_{19} & w_{20} \\ w_{21} & w_{22} & w_{23} & w_{24} \end{bmatrix} \qquad W_2 = \begin{bmatrix} z_1 & z_2 \\ z_3 & z_4 \\ z_5 & z_6 \\ z_7 & z_8 \end{bmatrix}$$

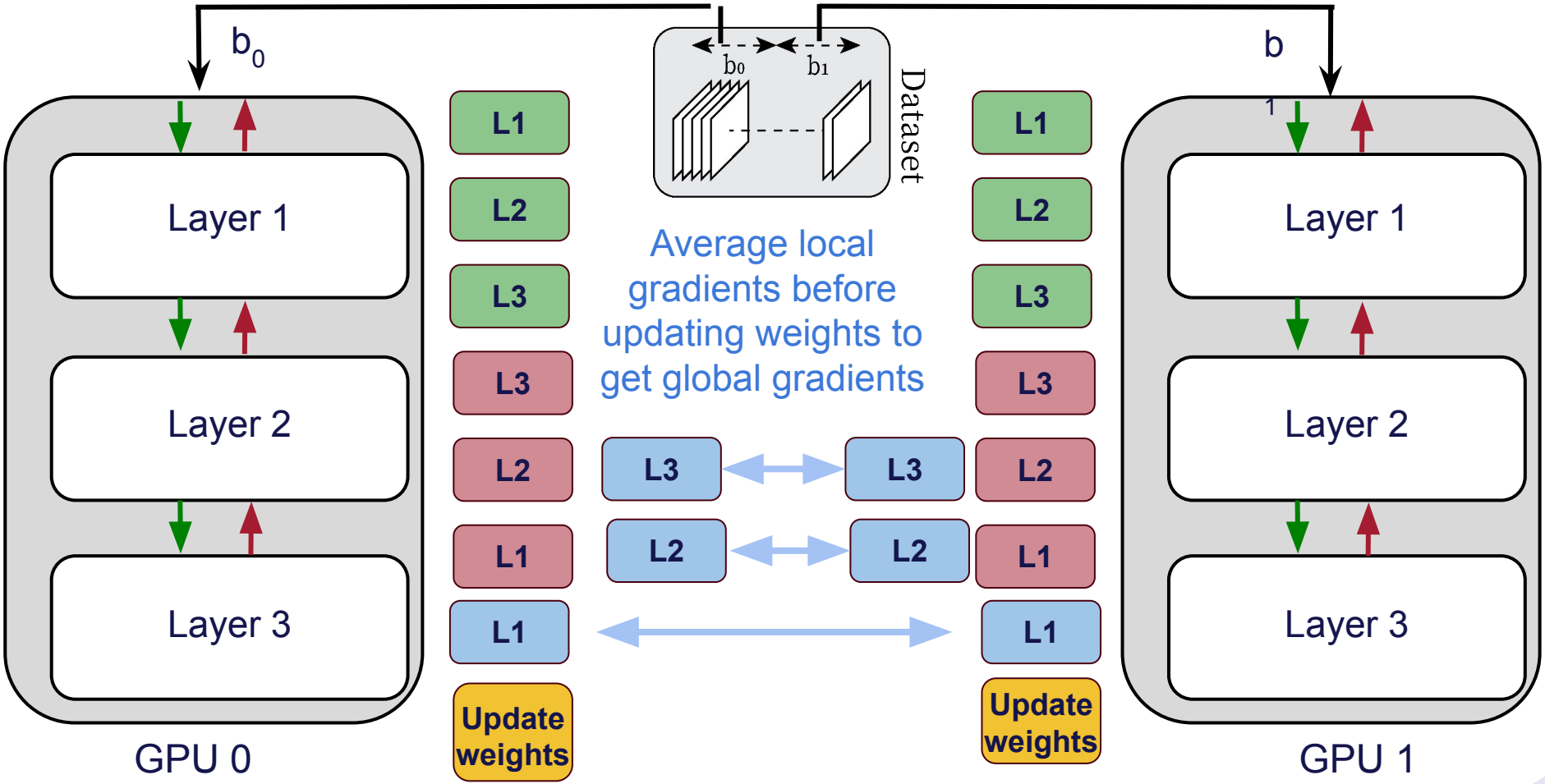
$$y_i = [y_1\ y_2] \qquad b_1 = [b_1\ b_2\ b_3\ b_4] \qquad b_2 = [c_1\ c_2]$$

Single GPU MLP Training

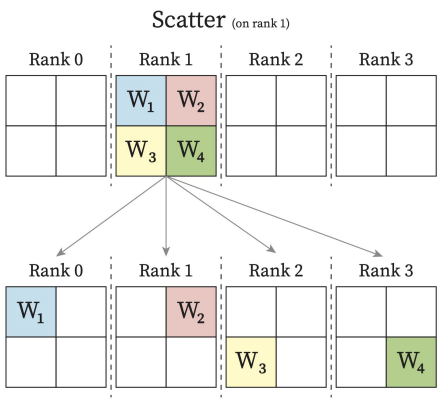
1) Model gets batch of data
2) Computes forward pass
3) Computes backward pass (computing gradients)
4) Updates weights based on gradients

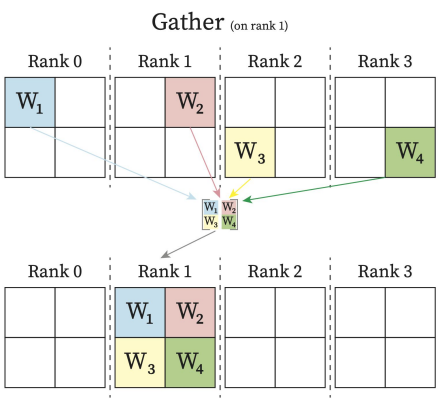


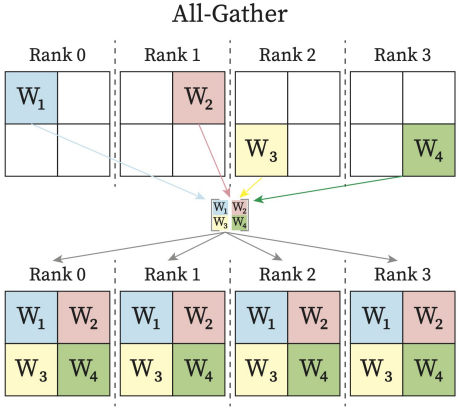

b0
b1
Dataset
b
1
Layer 1
Layer 2
Layer 3
L1
L2
L3
L3
L2
L1
Update weights
At what point should there be communication between the GPUs? What should the communication be?
GPU 0
GPU 1

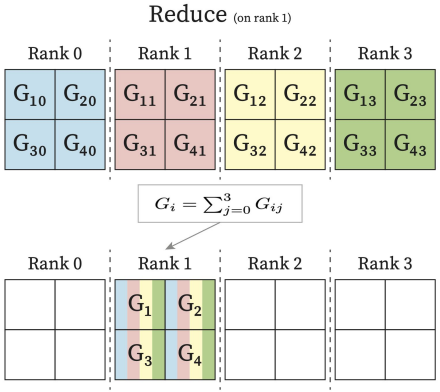

b0
b1
Dataset
b
1
Layer 1
Layer 2
Layer 3
L1
L2
L3
L3
L2
L1
L1
Average local gradients before updating weights to get global gradients
L3
L3
L2
L2
L1
L1
Update weights
Update weights
GPU 0
GPU 1

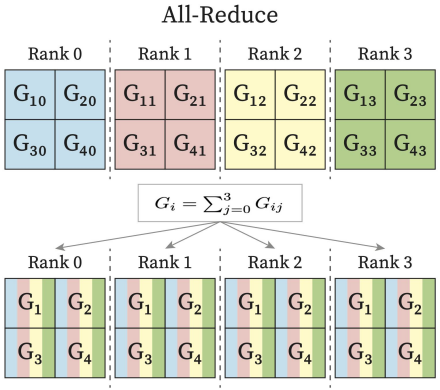
Which NCCL Collective Primitive?

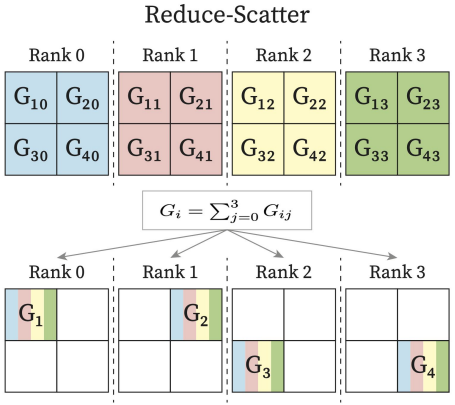


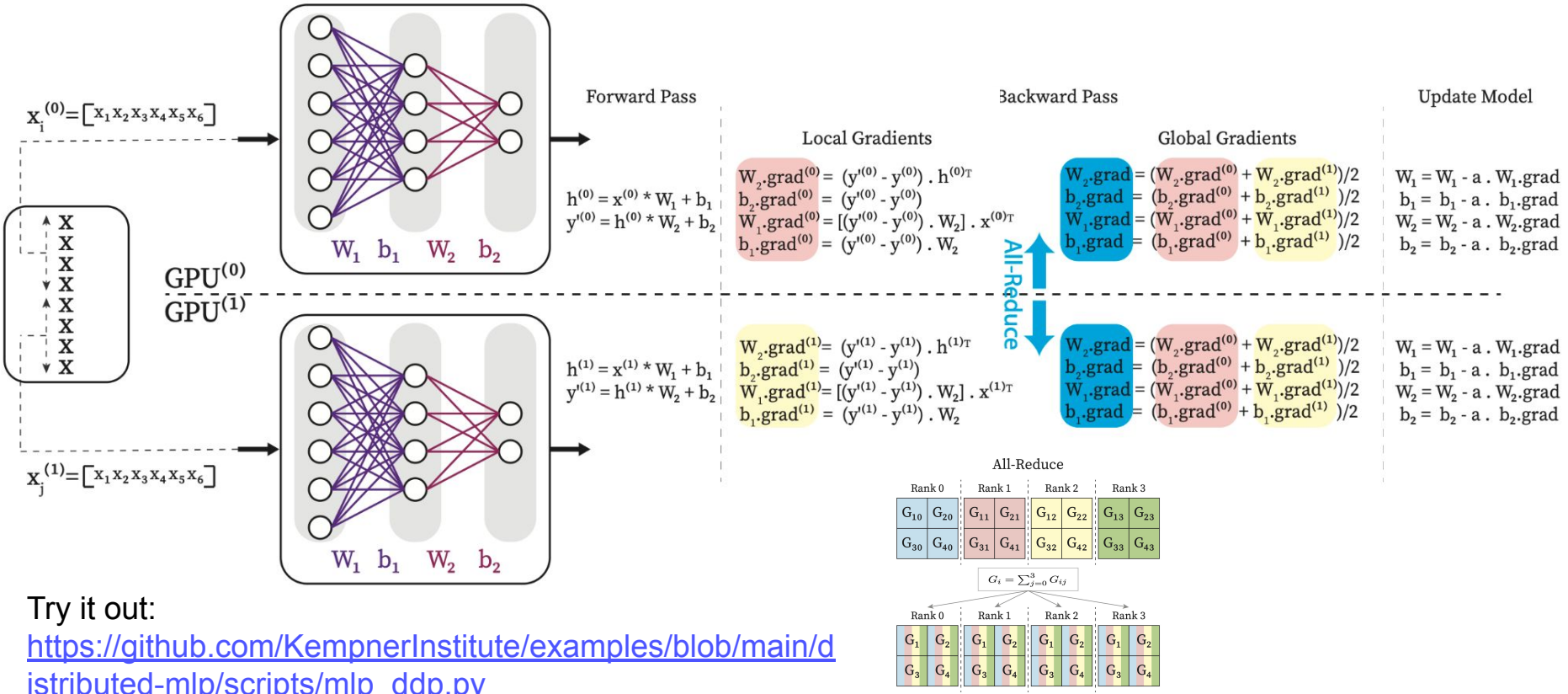




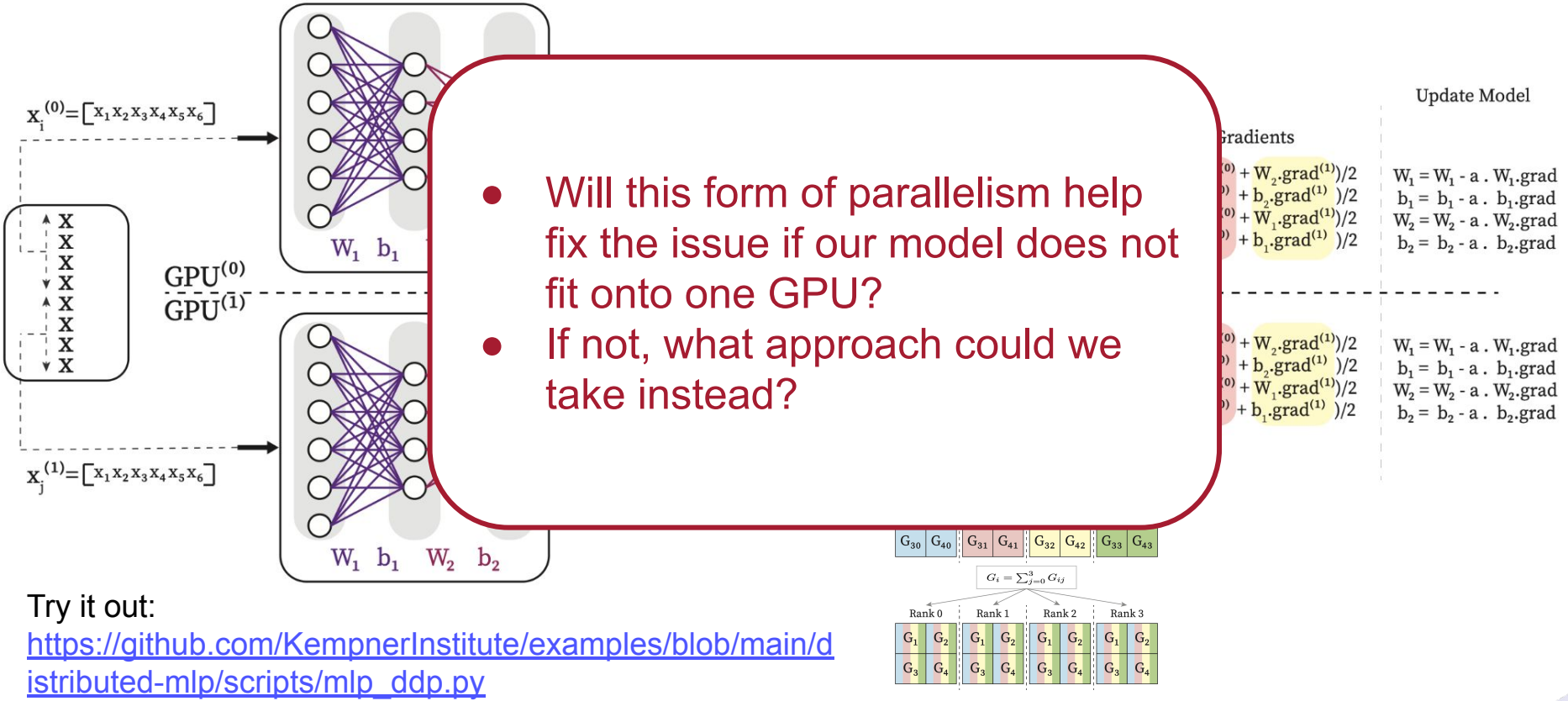








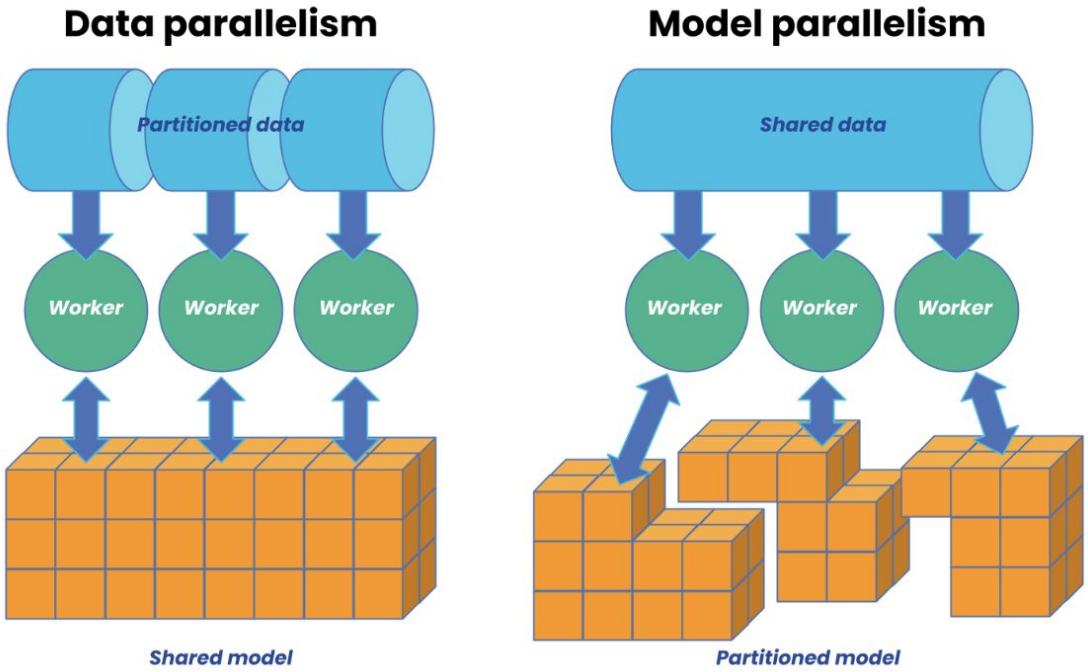
Try it out:
https://github.com/KempnerInstitute/examples/blob/main/distributed-mlp/scripts/mlp_ddp.py



- Will this form of parallelism help fix the issue if our model does not fit onto one GPU?
- If not, what approach could we take instead?

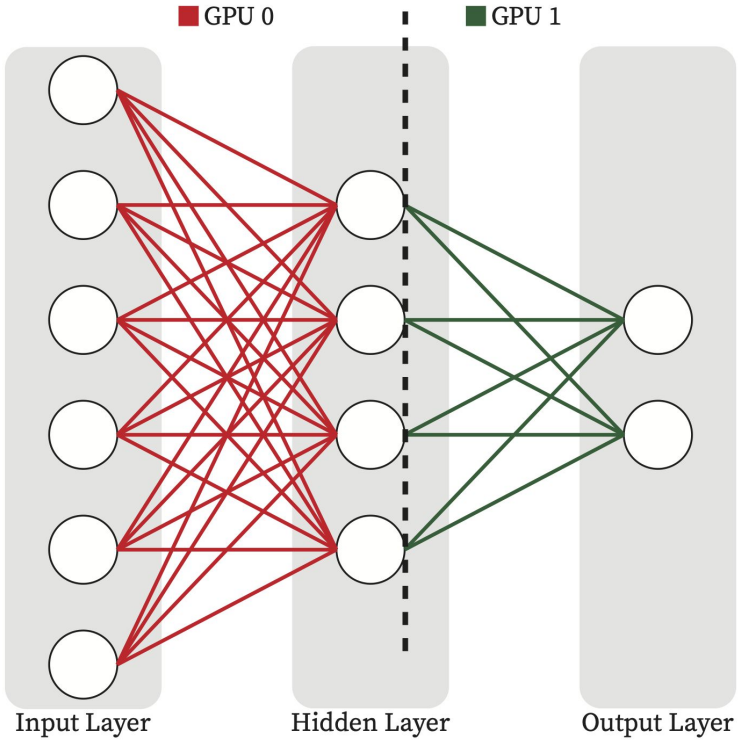
Try it out:
https://github.com/KempnerInstitute/examples/blob/main/distributed-mlp/scripts/mlp_ddp.py

Model Parallelism



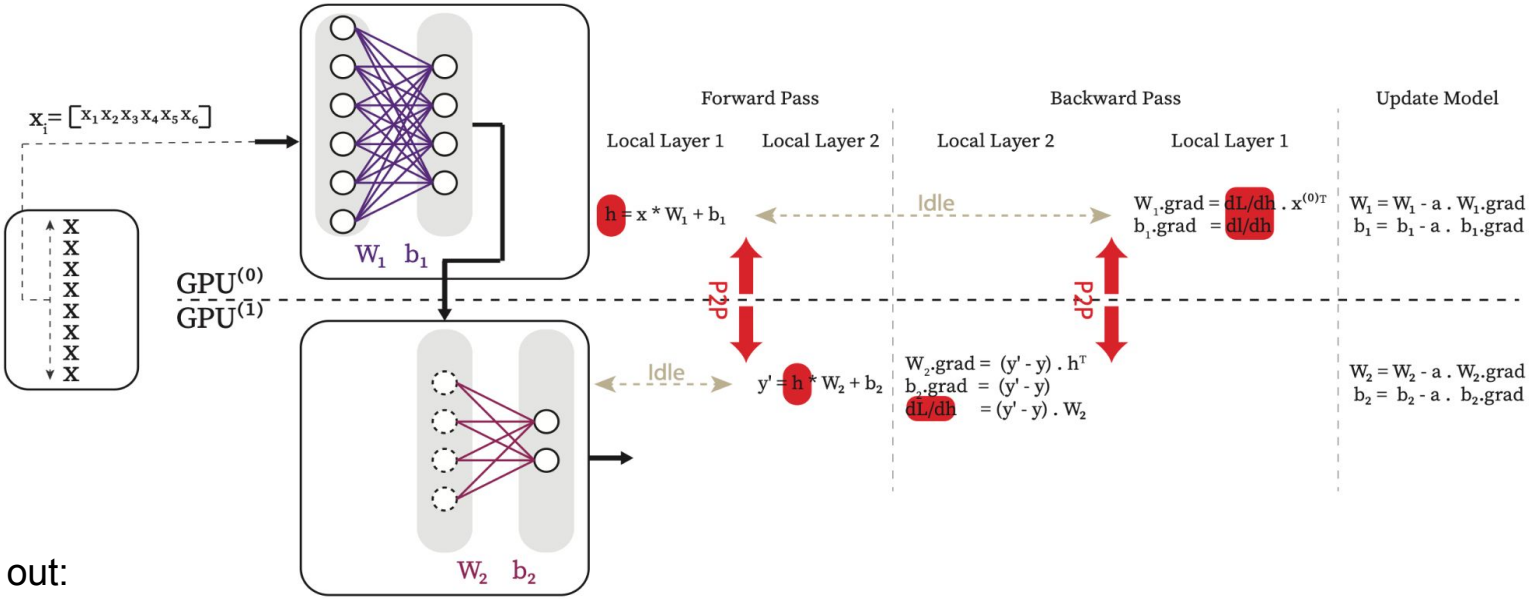
https://www.anyscale.com/blog/what-is-distributed-training

Model Parallelism



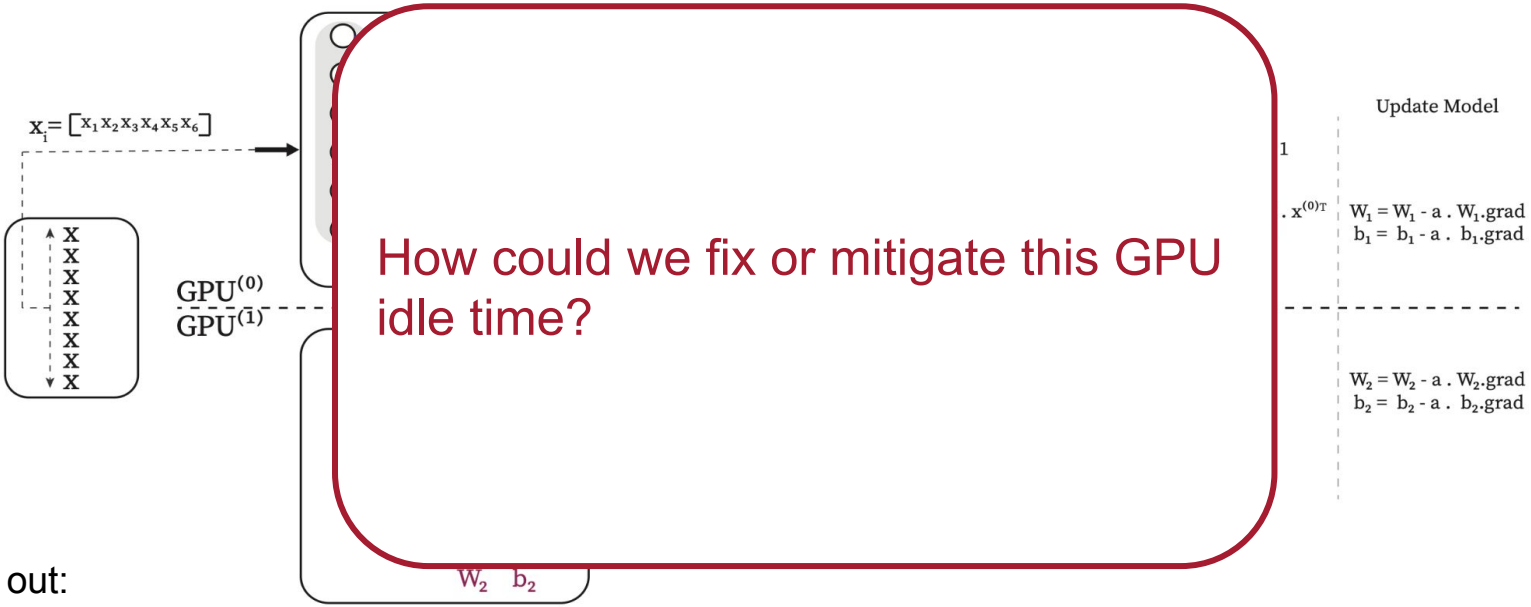

```python
class MLP(nn.Module):
  def __init__(self, in_feature, hidden_units, out_feature):
    super().__init__()

    self.hidden_layer = nn.Linear(in_feature, hidden_units).to(0)
    self.output_layer = nn.Linear(hidden_units, out_feature).to(1)

  def forward(self, x):
    x = self.hidden_layer(x.to(0))
    x = self.output_layer(x.to(1))
    return x
```

Model Parallelism and its Drawback



Try it out:
https://github.com/KempnerInstitute/examples/blob/main/distributed-mlp/scripts/mlp_model_parallel.py

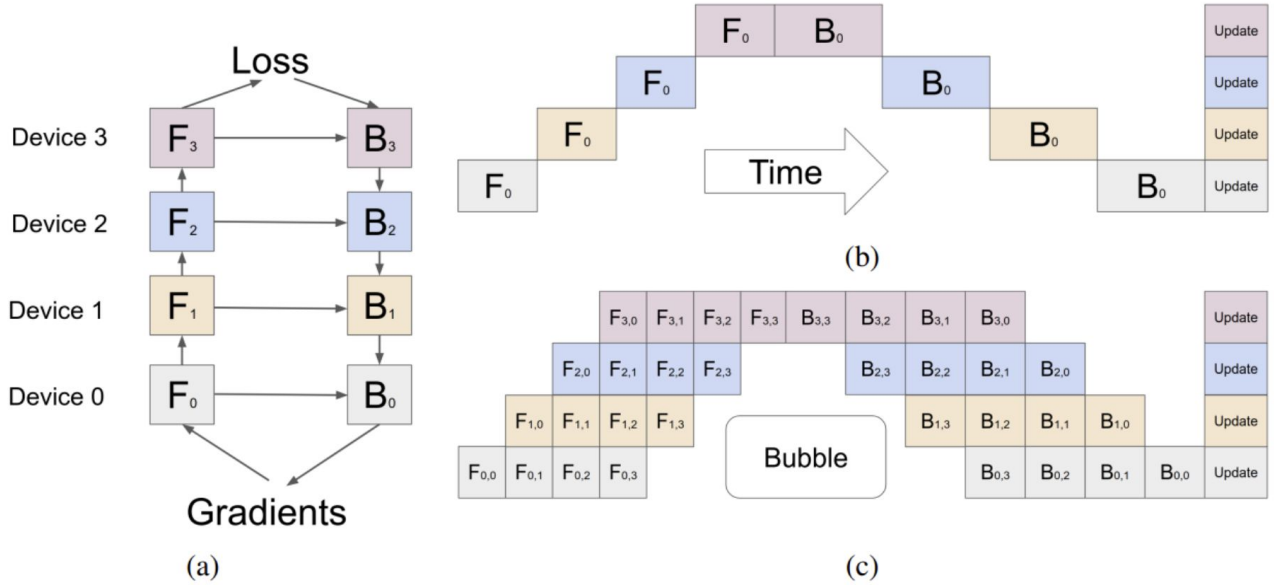
Model Parallelism and its Drawback



Try it out:
https://github.com/KempnerInstitute/examples/blob/main/distributed-mlp/scripts/mlp_model_parallel.py

Pipeline Parallelism

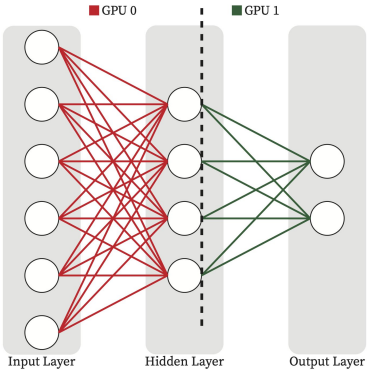
Pipeline Parallelism uses micro batches to reduce the idle time by adding overlaps

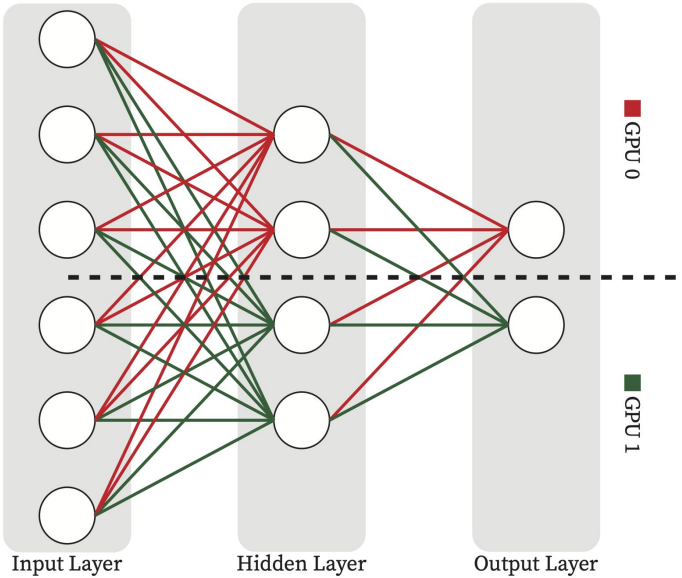


https://medium.com/nerd-for-tech/an-overview-of-pipeline-parallelism-and-its-research-progress-7934e5e6d5b8

Tensor Parallelism

Model/Pipeline Parallelism vs Tensor Parallelism

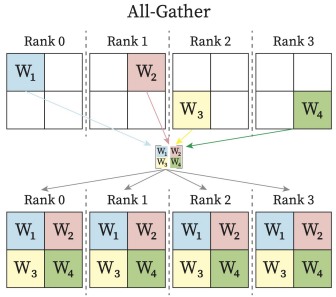


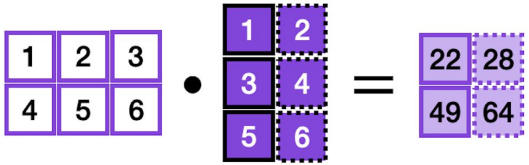


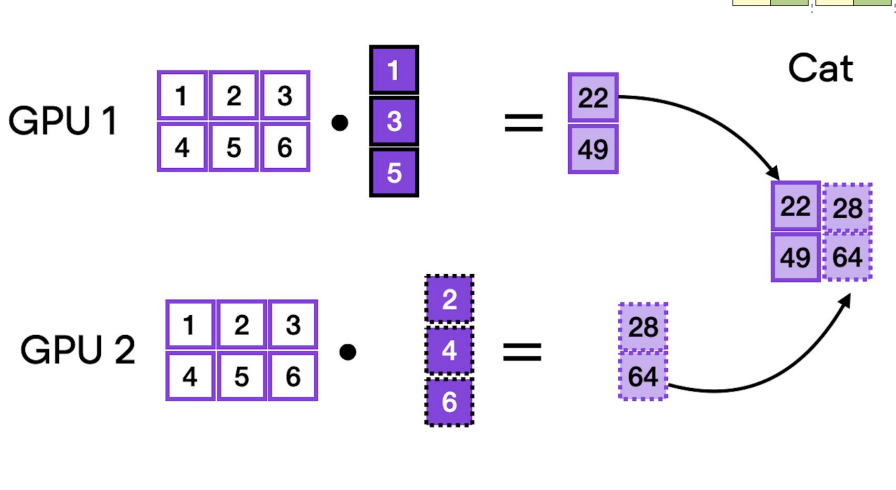
- All GPUs contribute in each layer computation
- Remove the GPU Idle time

Tensor Parallelism

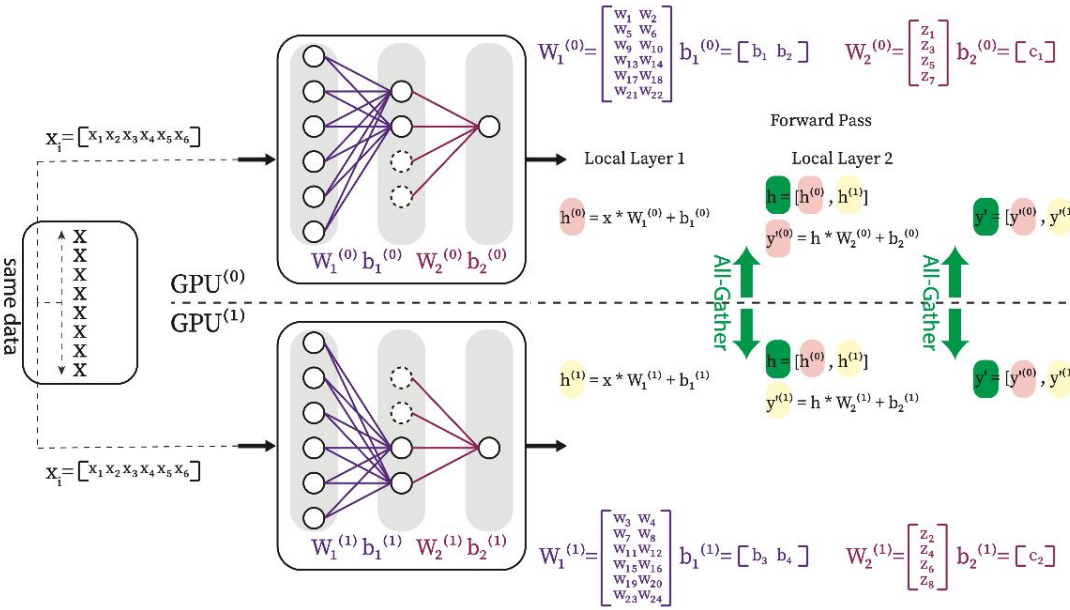
Splitting the weights column-wise between GPUs



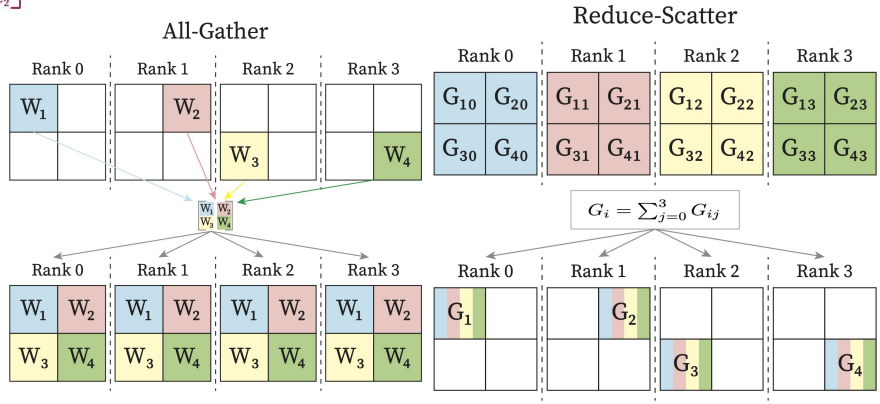




https://magazine.sebastianraschka.com/p/accelerating-pytorch-model-training



Try it out:
https://github.com/KempnerInstitute/examples/blob/main/distributed-mlp/scripts/mlp_tensor_parallel.py
More insight into what happens behind the scene:
https://github.com/KempnerInstitute/examples/blob/main/distributed-mlp/notebooks/tensor_parallelism_insight.ipynb



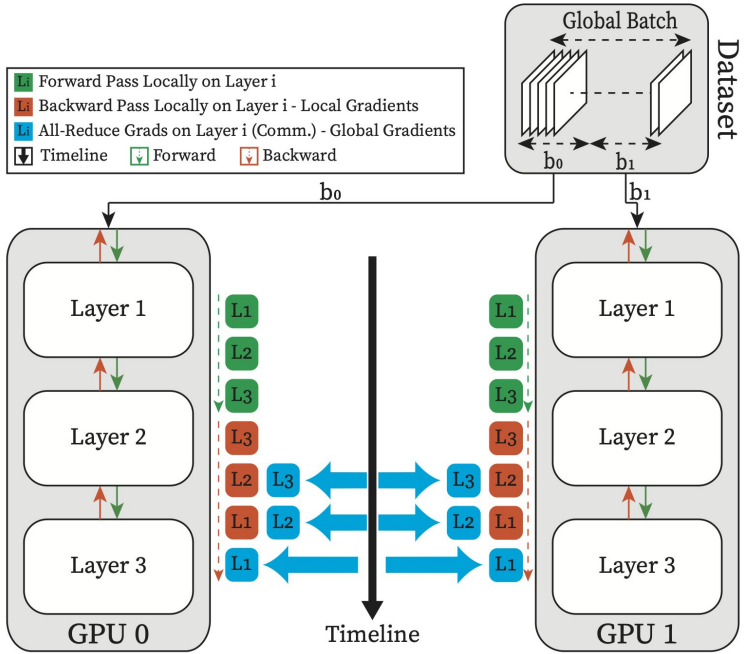
DDP vs FSDP: DDP

Each GPU has a copy of the model

1. Perform forward and backward passes locally

2. All-reduce gradients across GPUs (NCCL operation)

3. Update optimizer states and weights locally



FSDP vs DDP: Model Size

DDP

FSDP

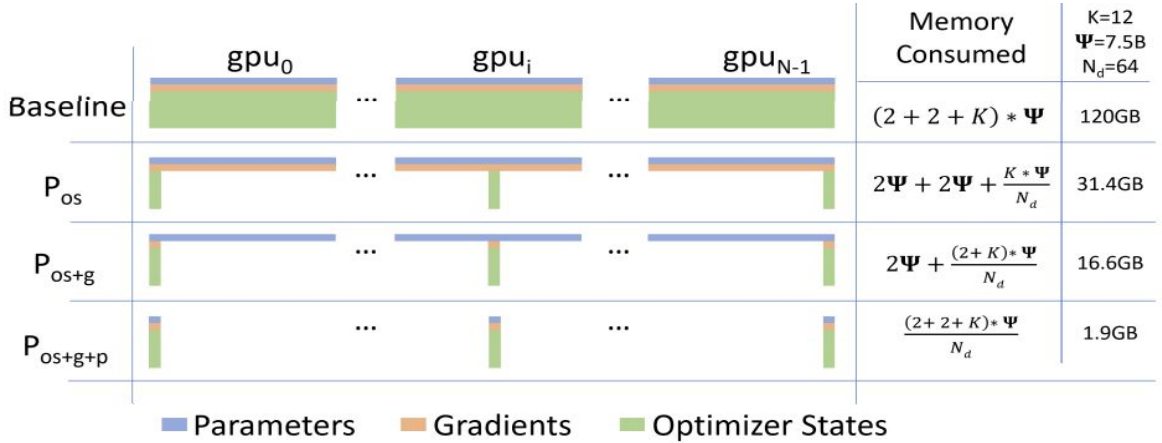


Figure 1: Comparing the per-device memory consumption of model states, with three stages of *ZeRO*-DP optimizations. Ψ denotes model size (number of parameters), K denotes the memory multiplier of optimizer states, and N_d denotes DP degree. In the example, we assume a model size of $\Psi = 7.5B$ and DP of $N_d = 64$ with $K = 12$ based on mixed-precision training with Adam optimizer.

Samyam R. et al, ZeRO: Memory Optimizations Toward Training Trillion Parameter Models. arxiv

DDP vs FSDP: FSDP

Each GPU has a shard of the model - 1D flatten parameters divided between GPUs

Forward:

1. All-gather all the weights across GPUs (NCCL operation)
2. Perform the forward pass locally
3. Release the collected weights to free memory

Backward:

4. All-gather all the weights across GPUs (NCCL operation)
5. Perform the backward pass locally
6. Release the collected weights to free memory

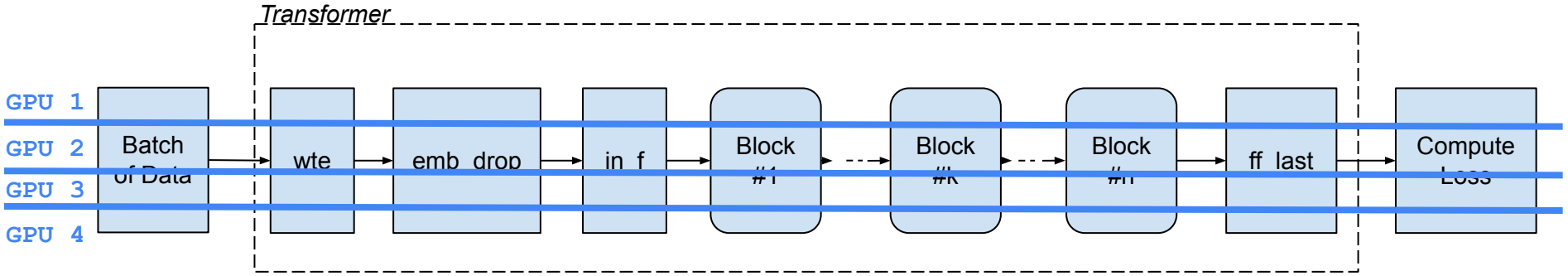
Optimizer step and weight update:

7. Reduce-scatter gradients across GPUs (NCCL operation)

   (Each GPUs will only compute their own partial of the gradient)

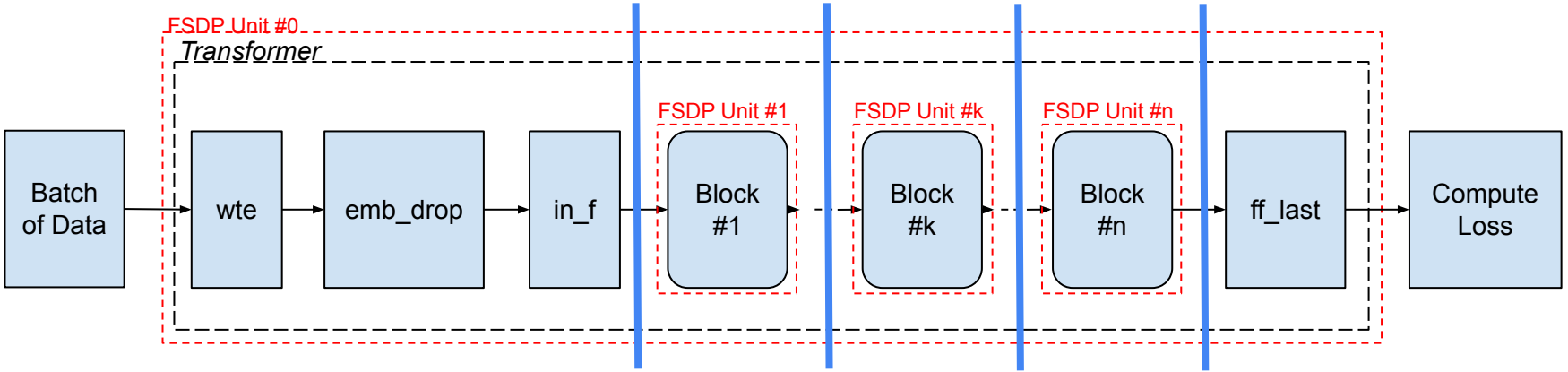
8. Update optimizer states and weights locally

FSDP: Overlap Computation and Communication



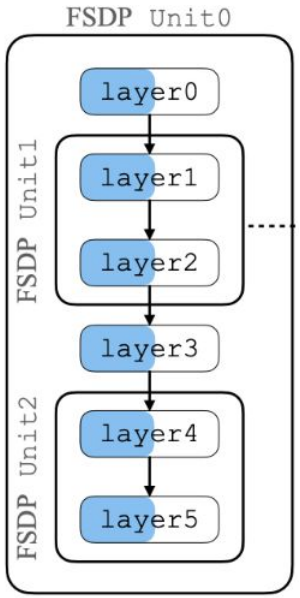
- FSDP fully shards all Parameters, Gradients and Optimizer states across the GPUs.
- Each All-gather, Forward pass, All-gather, Backward pass, Reduce-scatter, Optimizer and Weights update needs to be done sequentially.
  - No opportunity for overlapping computation and communication
  - Needs sort of dividing the model vertically into multiple subsets (aka units) to make this overlap possible

FSDP: Overlap Computation and Communication



- All-gathers and perform forward/backward pass is performed unit by unit
  - Helps with memory
    - Loads parameters only for the current unit
    - Needs to have enough memory to load the largest FSDP unit
  - Provide the computation and communication overlap
    - While unit #1 is performing forward pass, unit #2 all-gathers it's parameters

FSDP: Overlap Computation and Communication



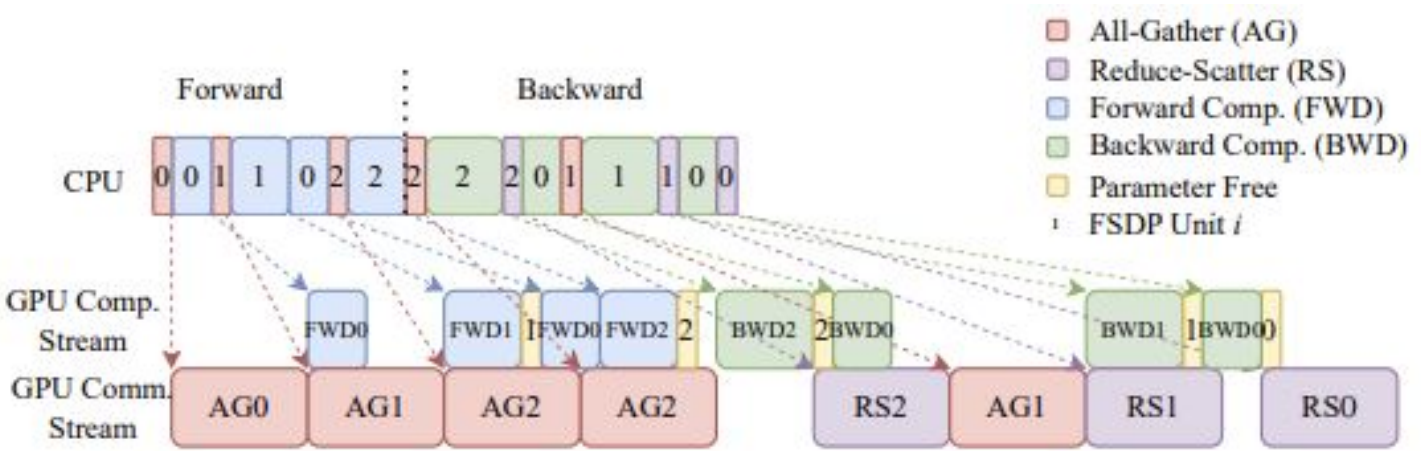
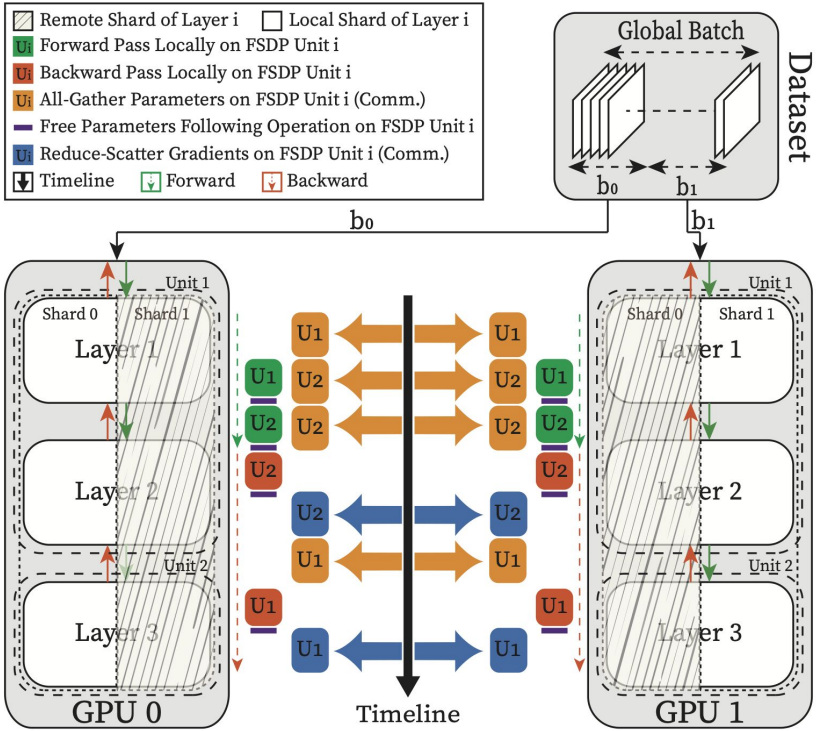


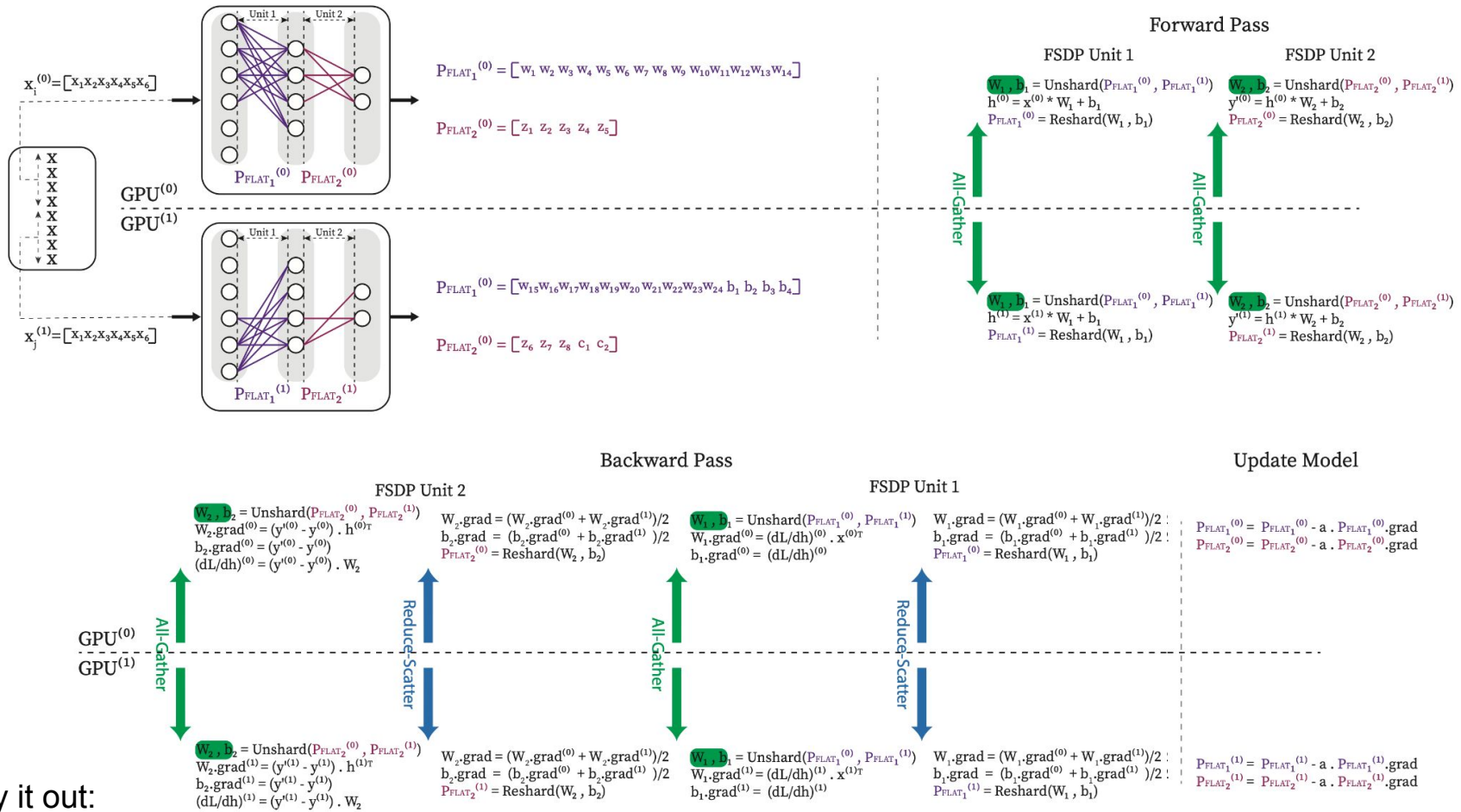
Figure 5: Overlap Communication and Computation

Yanli Z., et al, PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel. arxiv

FSDP: Overlap Computation and Communication



https://handbook.eng.kempnerinstitute.harvard.edu/s5_ai_scaling_and_engineering/scalability/gpu_computing.html



Try it out:
https://github.com/KempnerInstitute/examples/blob/main/distributed-mlp/scripts/mlp_fsdp.py

Agenda

1 Why Going Distributed?

2 Intro to Distributed GPU Computing

3 Different Distributed LLM Training Techniques

4 Training a Large LLM on the Cluster

OLMo: 1B - DDP

Distributed Data Parallelism

```
model:
  d_model (D): 2048
  n_layers (N): 16
  n_heads (N_h): 16
  max_sequence_length (T): 2048
  vocab_size (V): 32100
```

```
ddp:
  grad_sync_mode:      batch
  find_unused_params: false

distributed_strategy: ddp
```

$P = N(12D^2 + 10D) + D(V+T) + 2D + V(D+1)$

$P = 0.944\ B$

https://github.com/KempnerInstitute/OLMo/blob/main/configs/kempner_institute/1b_Olmo.yaml

Instruction:
https://github.com/KempnerInstitute/OLMo/blob/main/README_KempnerInstitute.md

OLMo: 7B - FSDP

Fully Sharded Data Parallelism

model:
- d_model (D): 4096
- n_layers (N): 32
- n_heads (N_h): 32
- max_sequence_length (T): 2048
- vocab_size (V): 32100

fsdp:
- wrapping_strategy: by_block
- precision: mixed
- sharding_strategy: FULL_SHARD

distributed_strategy: fsdp

$P = N(12D^2 + 10D) + D(V+T) + 2D + V(D+1)$

$P = 6.715$ B

https://github.com/KempnerInstitute/OLMo/blob/main/configs/kempner_institute/7b_Olmo.yaml

Instruction:
https://github.com/KempnerInstitute/OLMo/blob/main/README_KempnerInstitute.md

Exercise: Try out running OLMo (Steps 1 and 2)

1) Install OLMo if you haven't already

(instructions: http://github.com/KempnerInstitute/OLMo/blob/main/README_KempnerInstitute.md)

2) Look through the config file for 7B FSDP

(Section 2.1: https://github.com/KempnerInstitute/OLMo/blob/main/README_KempnerInstitute.md)

Different Wrapping Policy

How to wrap modules into FSDP units

- by_block: put each OLMo block into its own FSDP unit.
- by_block_and_size: same as by_block but `wte` and `ff_out` will be wrapped separately.
- by_block_group: put each m OLMo blocks into its own FSDP unit.
- size_based: uses PyTorch's default size-based auto wrap policy. (Wraps any module above 100M size in its own FSDP unit)

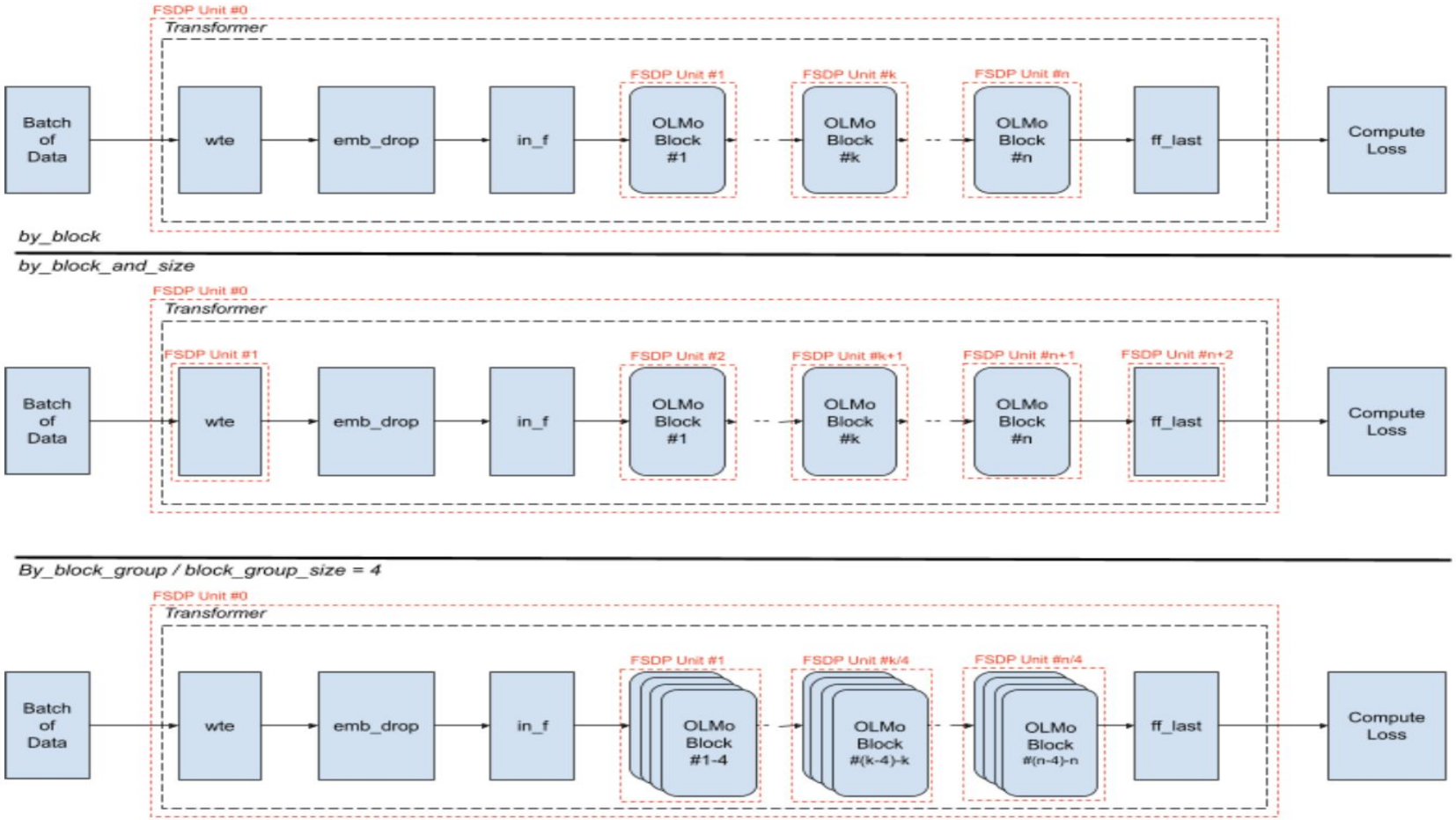


Figure 3: This shows how OLMo's FSDP wrapping policies form the FSDP units.

Different Sharding Strategies

May trades off memory saving and communication overhead

- NO_SHARD: as same as DDP
- FULL_SHARD: Parameters, gradients, and optimizer states are sharded.
- SHARD_GRAD_OP: Gradients and optimizer states are sharded during computation, and additionally, parameters are sharded outside computation meaning it keeps parameters unshareded throughout the forward and backward computation.
- HYBRID_SHARD: Apply ``FULL_SHARD`` within a node, and replicate parameters across nodes. This results in reduced communication volume as expensive all-gathers and reduce-scatters are only done within a node, which can be more performant for medium-sized models.

Exercise: Try out running OLMo (Steps 3 and 4)

3) Update the SLURM script. Pair up and run OLMo with FSDP! Check out the output logs

- You can use `tail -f <path-to-output-file>` to monitor the logs.

4) Try out different wrapping policies and sharding strategies.

Thank you