MINERVA

PARALLEL AI

Minerva AI Winter School, February 2026

WHY PARALLEL TOOLS IN AI?

MEMORY OCCUPANCY

MODEL STATES

- Parameters
- Gradients
- Optimizer states (such as momentum and variances in Adam)

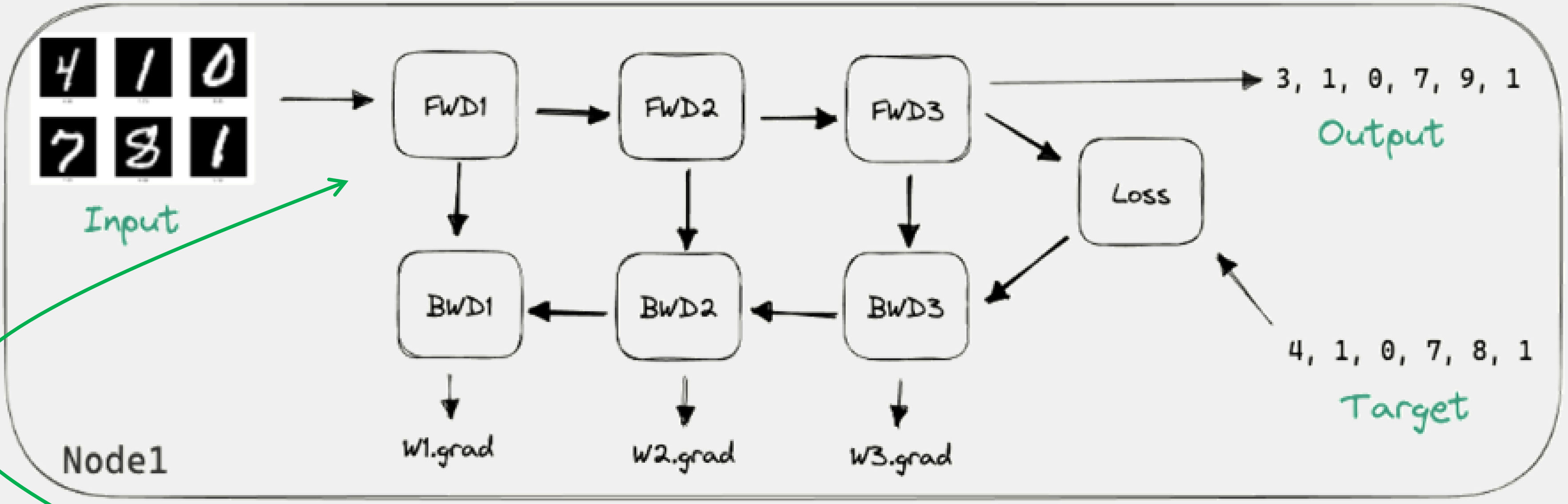
RESIDUAL STATES

- Activations
- Temporary buffers
- Unusable fragmented memory

[1] https://arxiv.org/pdf/2101.06840

BASICS OF DNN TRAINING

$$h^{(l)} = s(W^{(l)}h^{(l-1)} + b^{(l)}))$$



loss computation

$$R_i = loss(\hat{y}_i - y_i)$$

gradient computation

$$\frac{\partial R_i}{\partial b^{(l)}(j)} = \delta^{(l)}(j)$$

$$\frac{\partial R_i}{\partial W^{(l)}(j,k)} = \delta^{(l)}(j)h_i^{(l)}$$

optimizer step

$$W^{(l)}(j,k) = W^{(l)}(j,k) - \gamma \sum_{i=k}^{k+t} \frac{\partial R_i}{\partial W^{(l)}(j,k)}$$

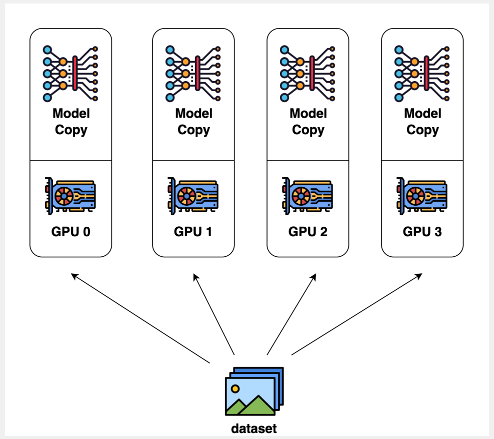
$$b^{(l)}(j) = b^{(l)}(j) - \gamma \sum_{i=k}^{k+t} \frac{\partial R_i}{\partial b^{(l)}(j)}$$

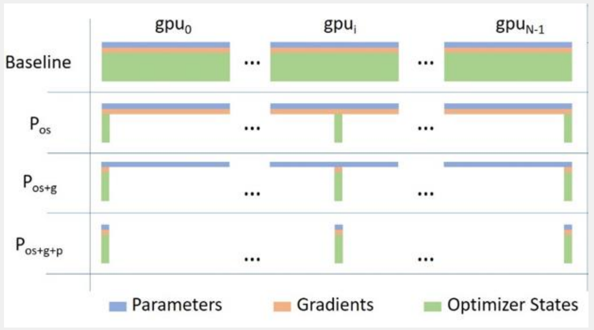
DIFFERENT TYPES OF PARALLELISM

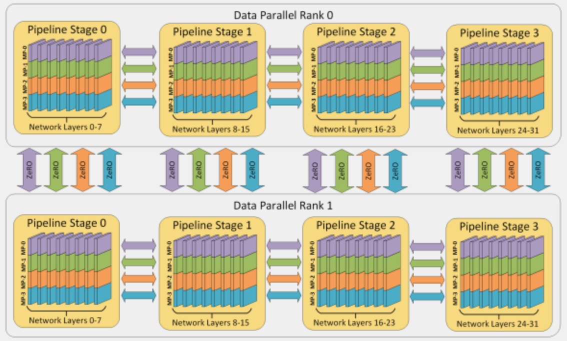
Data Parallelism

Model Sharding

Model Parallelism







https://colossalai.org/docs/concepts/paradigms_of_parallelism/

https://www.deepspeed.ai/tutorials/pipeline/

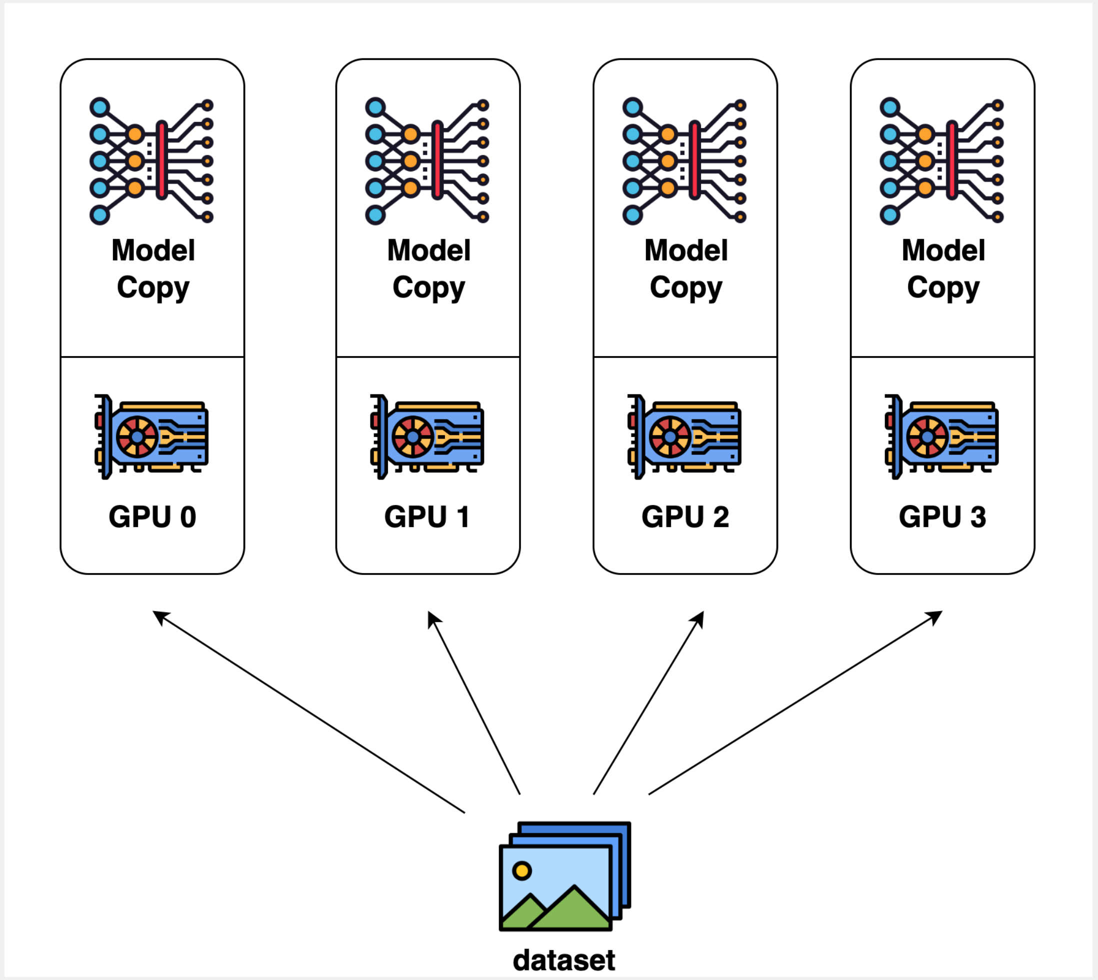
MINERVA

DATA PARALLELISM

HOW IT WORKS

- Each device holds a **full copy** of the model

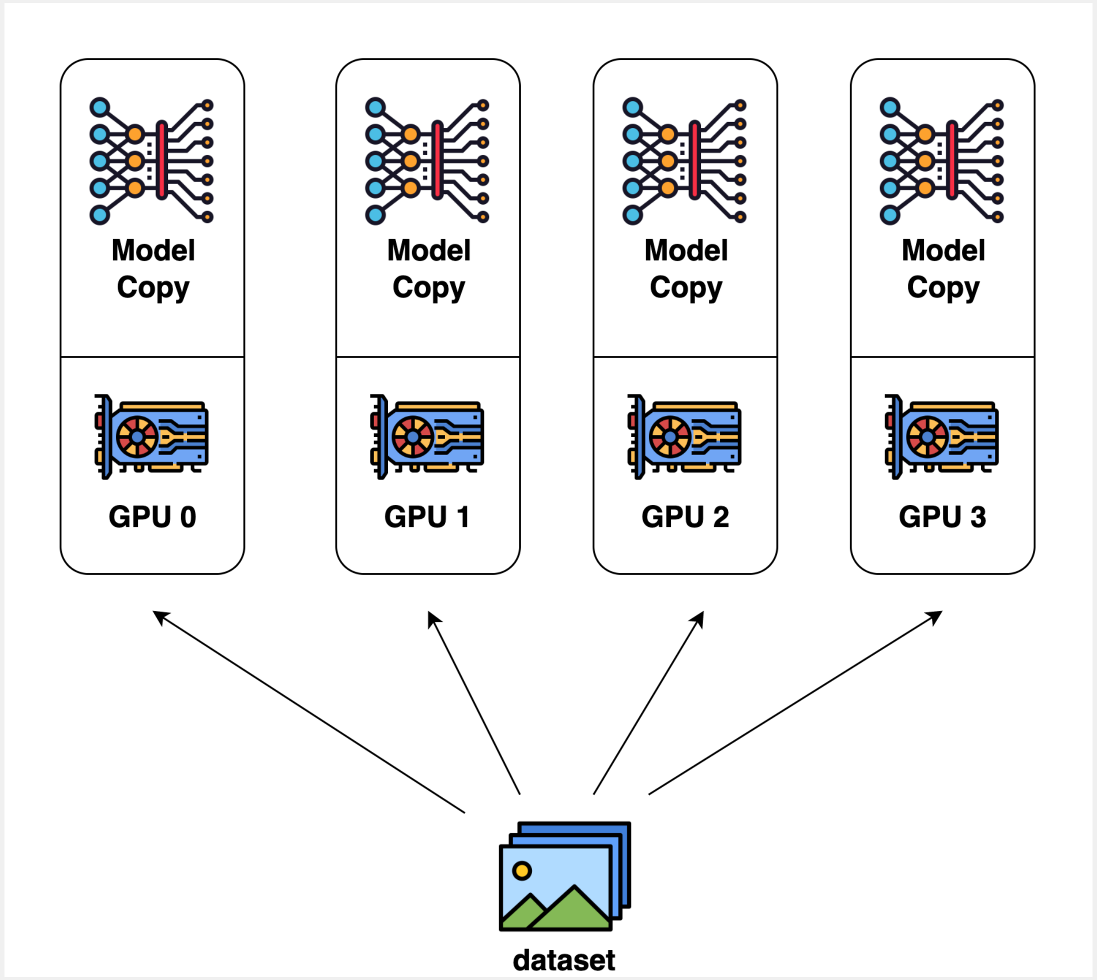
https://colossalai.org/docs/concepts/paradigms_of_parallelism/



HOW IT WORKS

- Each device holds a **full copy** of the model
- Data are split on multiple GPUs (single or multi-node)

https://colossalai.org/docs/concepts/paradigms_of_parallelism/

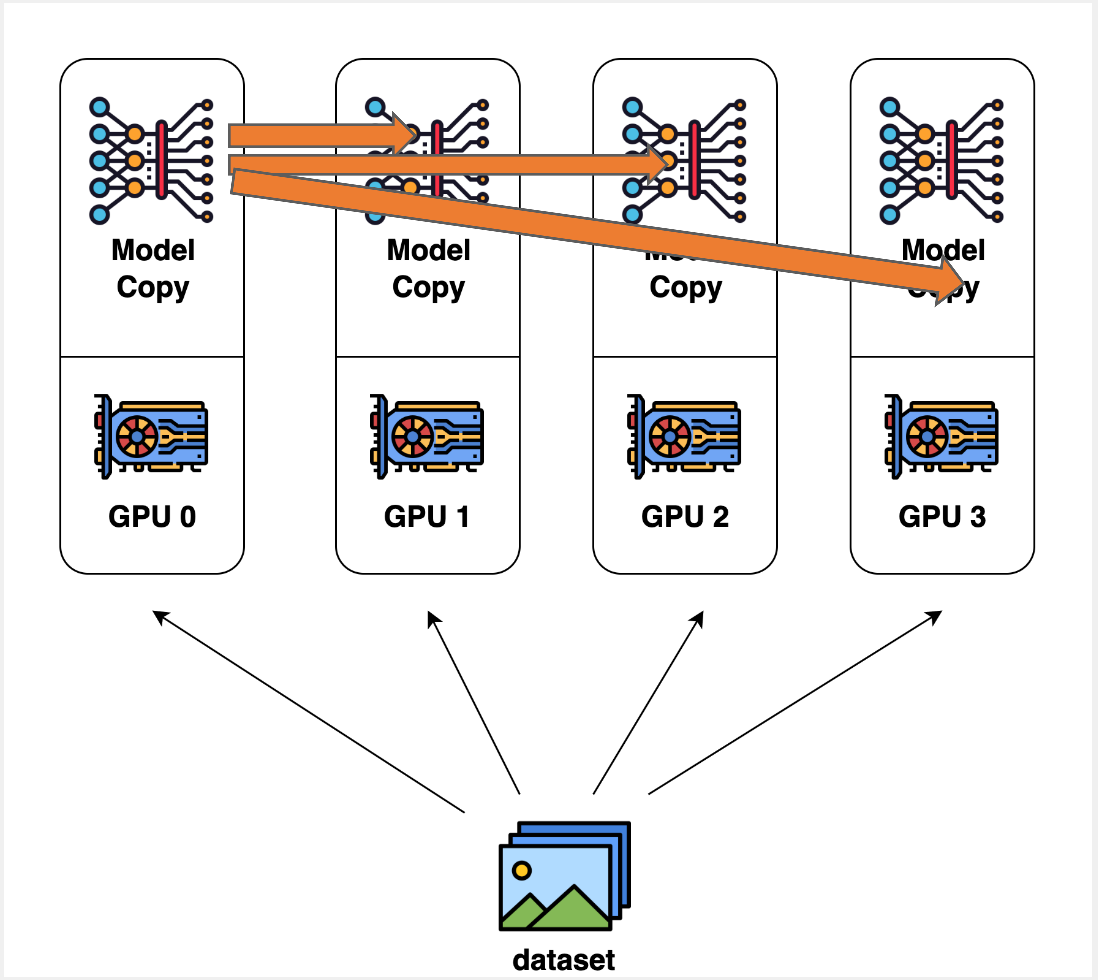


HOW IT WORKS

- Each device holds a **full copy** of the model
- Data are split on multiple GPUs (single or multi-node)
- After some backpropagation steps, **gradients** are **synchronized** to avoid convergence issues

https://colossalai.org/docs/concepts/paradigms_of_parallelism/

Ring ALL-REDUCE



HOW IT WORKS

Data parallelism enables distributed training by **communicating gradients before the optimizer step** to make sure that parameters of all model replicas are updated using exactly the same set of gradients, and hence model replicas can stay consistent across iterations.

https://docs.nvidia.com/nemo-framework/user-guide/24.07/nemotoolkit/features/parallelisms.html

PYTORCH DDP

- **DataParallel** works on a **single** multi-GPU **node**, slower than DistributedDataParallel due to GIL contention

- **DistributedDataParallel** works on a **multi-node** multi-GPU setting, it can be combined also with **model parallelism**

[1] https://pytorch.org/tutorials/intermediate/ddp_tutorial.html
[2] https://towardsdatascience.com/distributed-parallel-training-data-parallelism-and-model-parallelism-ec2d234e3214
[3] https://medium.com/codex/a-comprehensive-tutorial-to-pytorch-distributeddataparallel-1f4b42bb1b51

DISTRIBUTEDDATAPARALLEL (DDP)

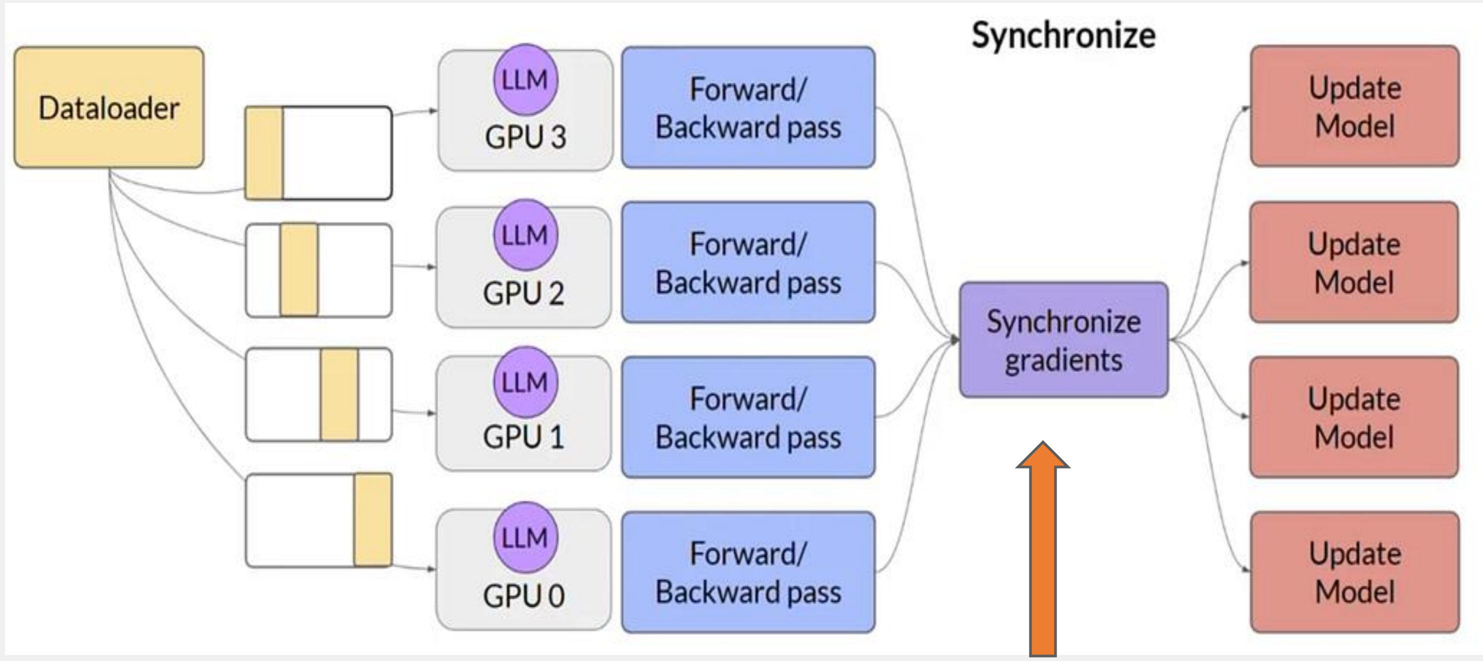
Faster and more flexible **than DP**, but does **not split the data** across multiple GPUs

Setup the process group
(always the same)

Split the data across multiple GPUs
(torch.utils.data.**DistributedSampler**)

Wrap the model with DDP

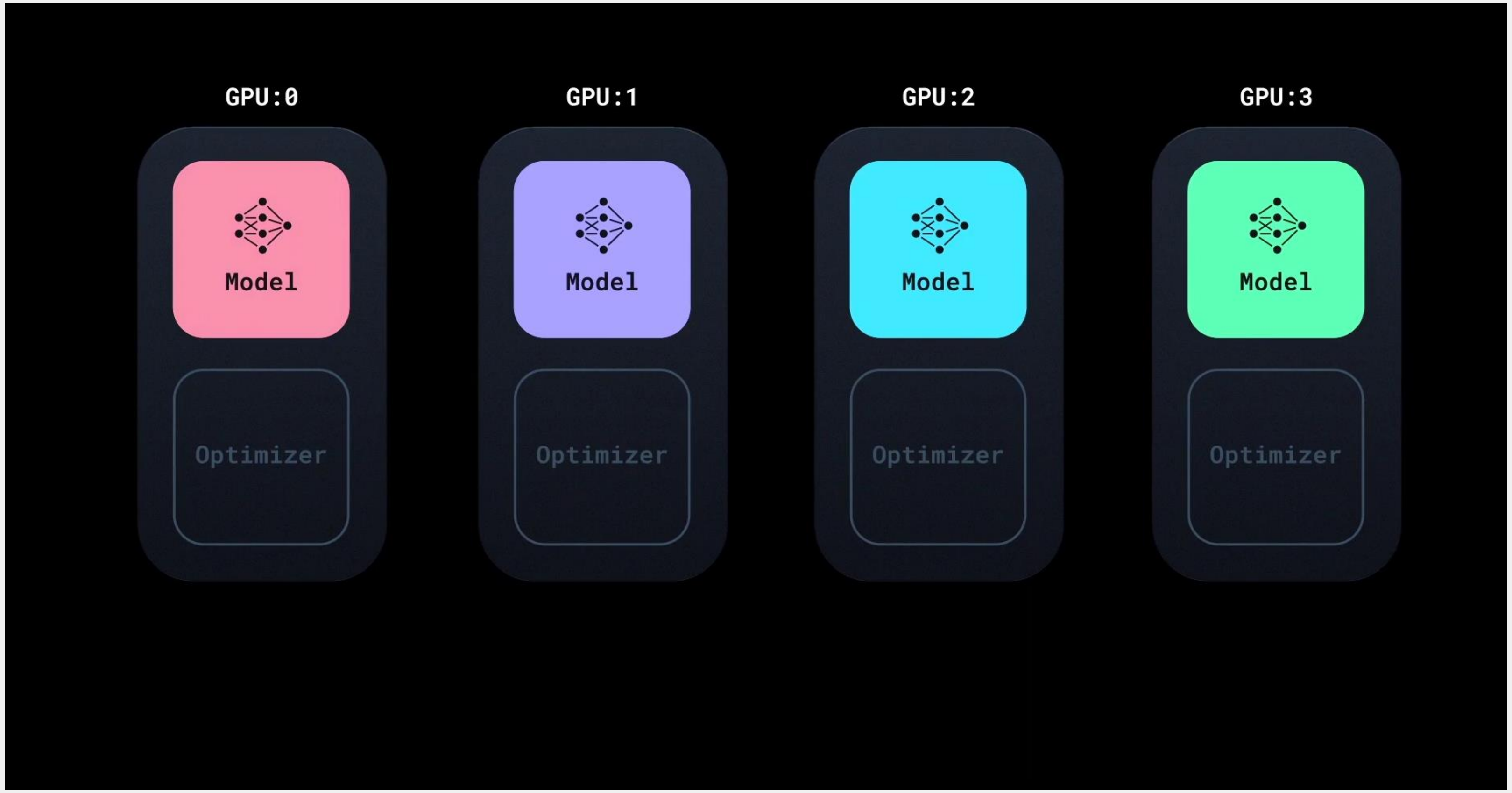
Nccl backend is highly reccommended
(both for single-node and multi-node)



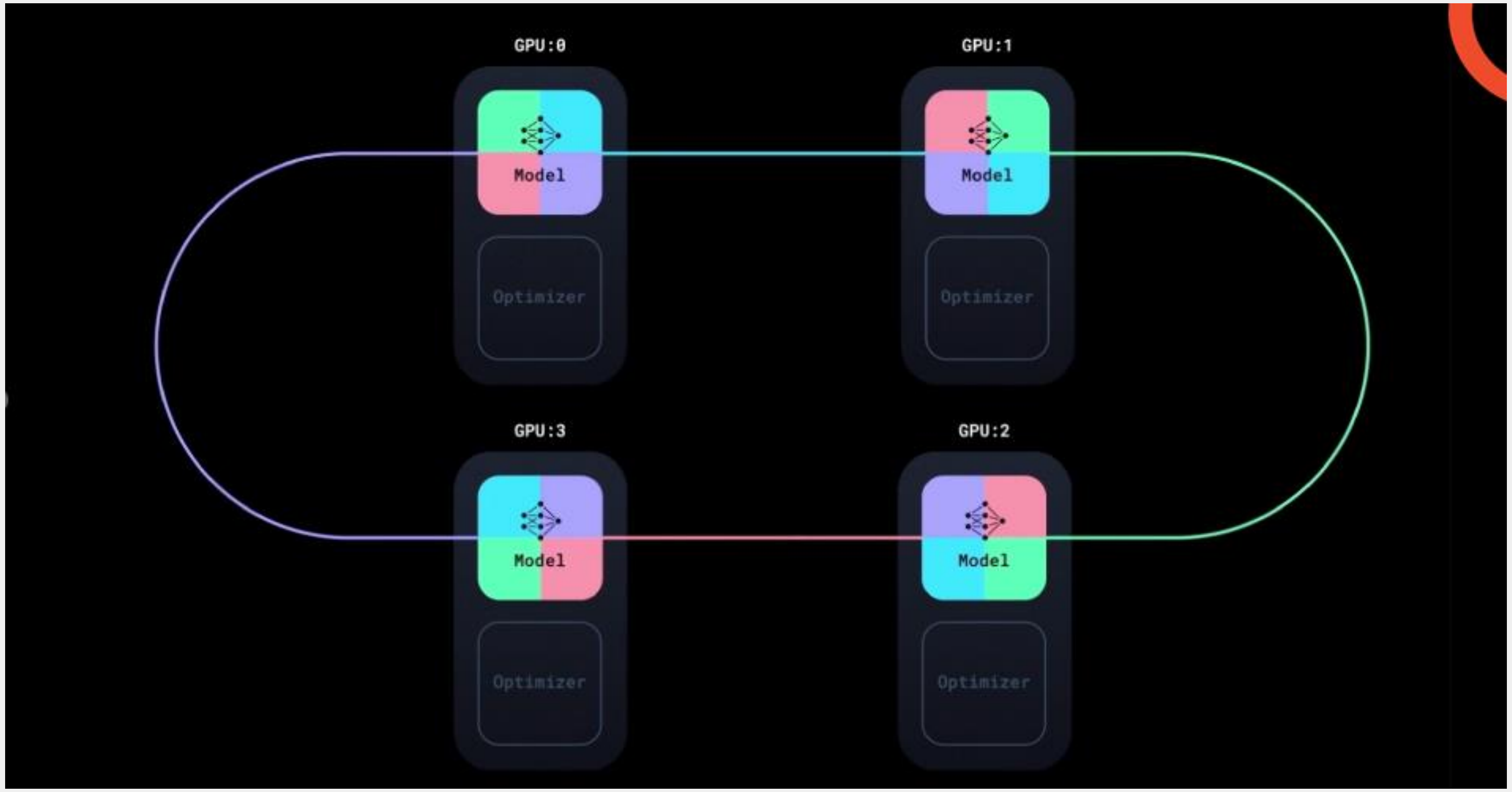
Ring ALL-REDUCE
synchronize gradients across model replicas

[1] https://pytorch.org/tutorials/intermediate/ddp_tutorial.html
[2] https://towardsdatascience.com/distributed-parallel-training-data-parallelism-and-model-parallelism-ec2d234e3214
[3] https://medium.com/codex/a-comprehensive-tutorial-to-pytorch-distributeddataparallel-1f4b42bb1b51
[4] https://arxiv.org/pdf/2006.15704

DISTRIBUTEDDATAPARALLEL (DDP)



DISTRIBUTEDDATAPARALLEL (DDP)



PYTORCH

Ring ALL-REDUCE synchronize gradients across model replicas

ACCELERATE – HOW TO LAUNCH LLM TRAINING

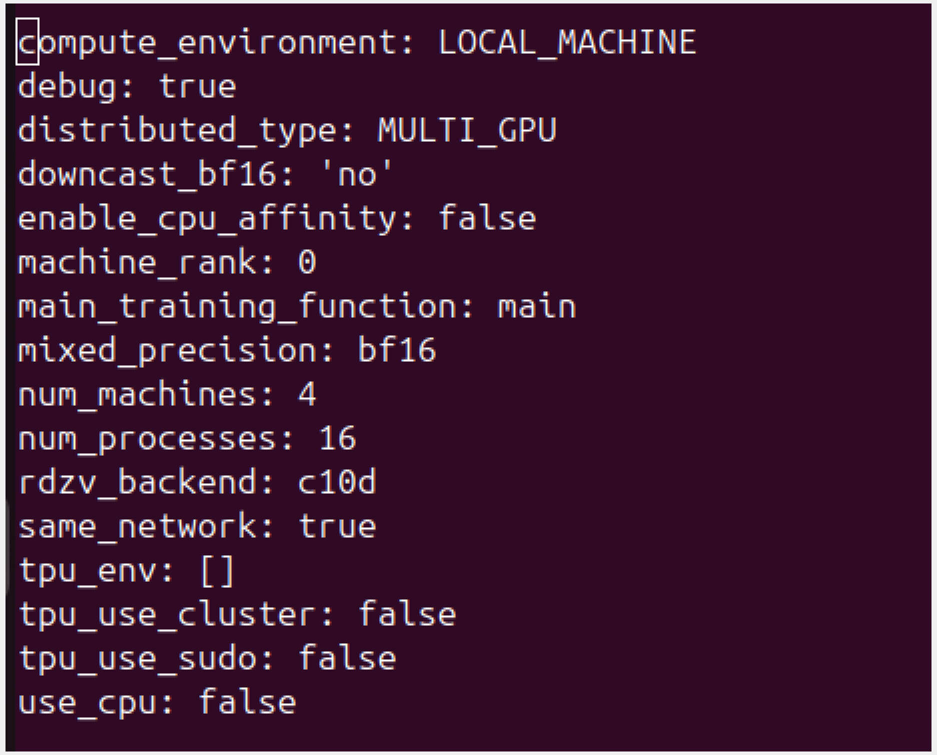
To implement DDP using HF Accelerate, you just need to follow these 3 simple steps:

1. Initialize Accelerate processes in our training
2. Define a .yaml config file
3. Launch it with accelerate launch

ACCELERATE

- Distributed training across multiple nodes with a simple configuration files and few changes in the python script

- Several level of parallelism: **data parallelism**, model sharding, hybrid sharding. Handles both multi CPUs and multi GPUs using
Only a config file!



```
compute_environment: LOCAL_MACHINE
debug: true
distributed_type: MULTI_GPU
downcast_bf16: 'no'
enable_cpu_affinity: false
machine_rank: 0
main_training_function: main
mixed_precision: bf16
num_machines: 4
num_processes: 16
rdzv_backend: c10d
same_network: true
tpu_env: []
tpu_use_cluster: false
tpu_use_sudo: false
use_cpu: false
```

ACCELERATE

To launch the training with the new Accelerate configuration performing distributed training it will be enough to run

```
accelerate launch --config_file config_accelerate.yaml my_script.py <py_args>
```

In SLURM it becomes

```
accelerate launch \
    --main_process_ip "$MASTER_ADDR" \
    --main_process_port $MASTER_PORT \
    --machine_rank $SLURM_PROCID \
    --rdzv_backend c10d \
    --config_file config_accelerate.yaml
    my_script.py <py_args>
```

READY TO SCALE YOUR LLM TRAINING

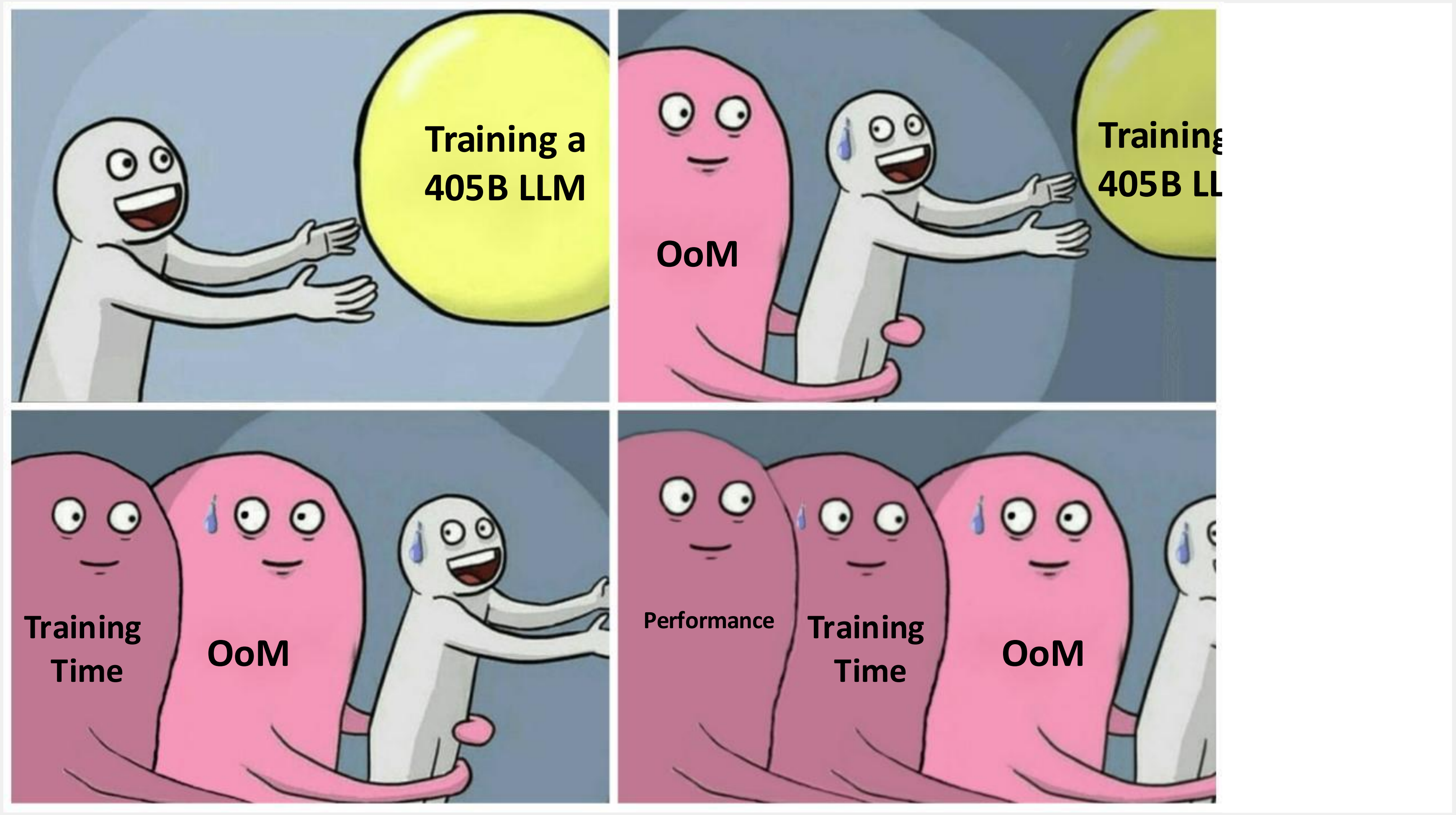
DISTRIBUTEDDATAPARALLEL (DDP)

- Copy the parallelAI folder in your $SCRATCH
- Go in the DDP subfolder
- Complete "FFT_DDP.py" exercise using :
  - "/leonardo_work/tra26_minwinsc/DATA//Bitext-customer-support-llm-chatbot-training-dataset" dataset
  - "/leonardo_work/tra26_minwinsc/models/Llama-3.2-1B-Instruct" model
- Run the code with 1 node ("job_FFT_DDP.sh")
- Run the code also with 2 nodes and check the training times
- Set **#SBATCH --reservation=s_tra_minwinsc** for accessing the reservation

MINERVA

MODEL SHARDING

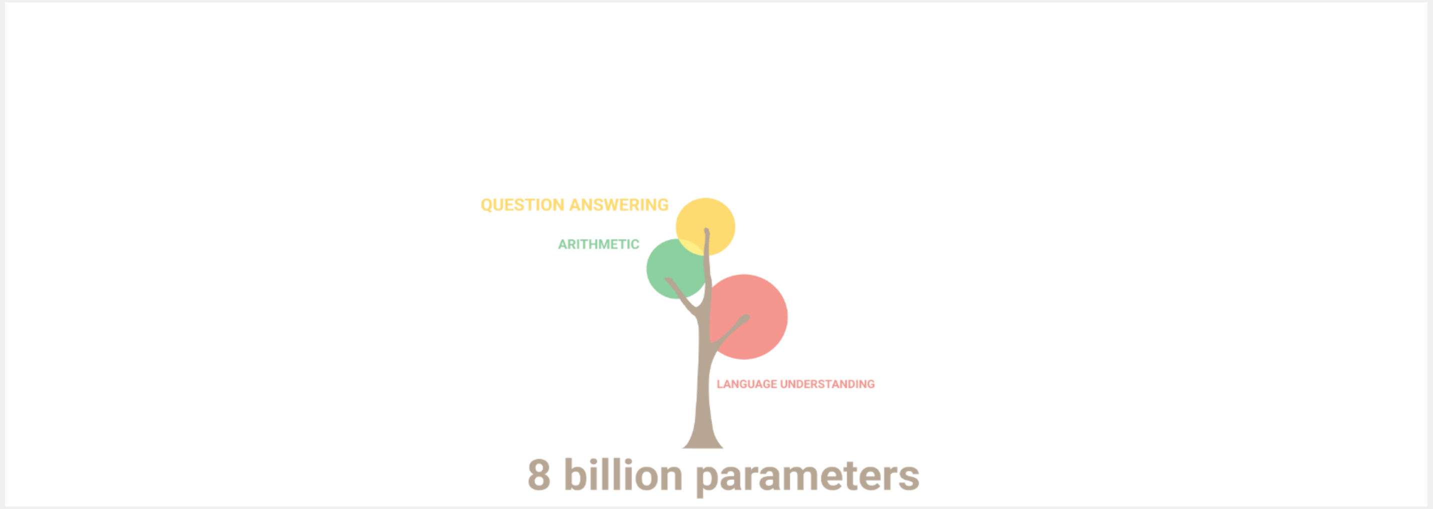
WHY MODEL SHARDING?



WHY MODEL SHARDING?

Full-Precision : fp32 (4 bytes)		
Params	7B x 4	28 GB
Activation	7B x 4	28 GB
Gradients	7B x 4	28 GB
Optimizer	7B x 4 x 2	56 GB
TOT		**140 GB**

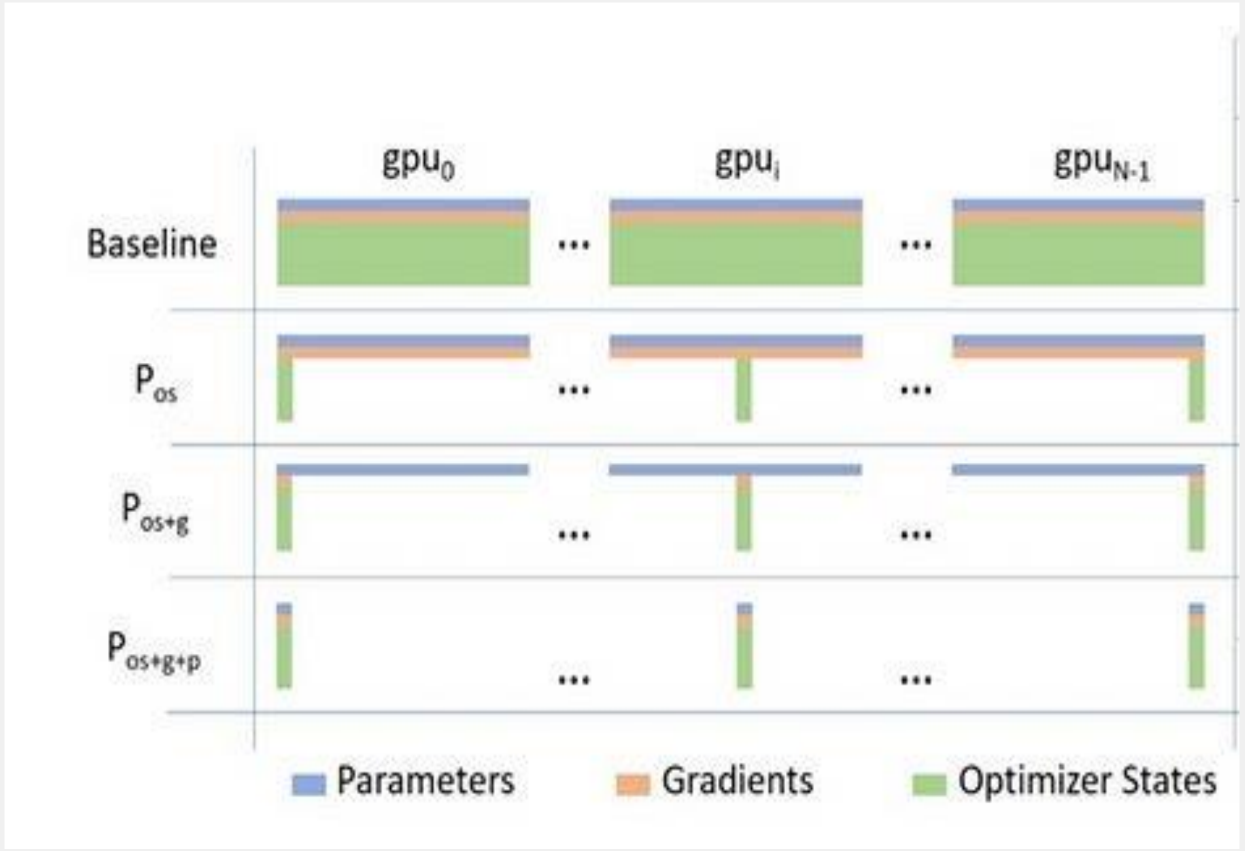
Half-Precision : fp16 or bf16 (2 bytes)		
Params	7B x 2	14 GB
Activation	7B x 2	14 GB
Gradients	7B x 2	14 GB
Optimizer	7B x 2 x 2	28 GB
TOT		**70 GB**



In Leonardo we have A100 GPUs with 64 GB each! → OoM error!

SHARDING: HOW IT WORKS

- Each device holds a **"shard"** of the model **parameters**, **optimizer states** and **gradients**
- Data are **split** on multiple GPUs (single or multi-node)
- Enables **larger** models **training** (the model Does not need to fit in a single gpu)
- **More communications** needed wrt DDP
- Replace **All Reduce** communication With **Reduce Scatter+ All Gather**



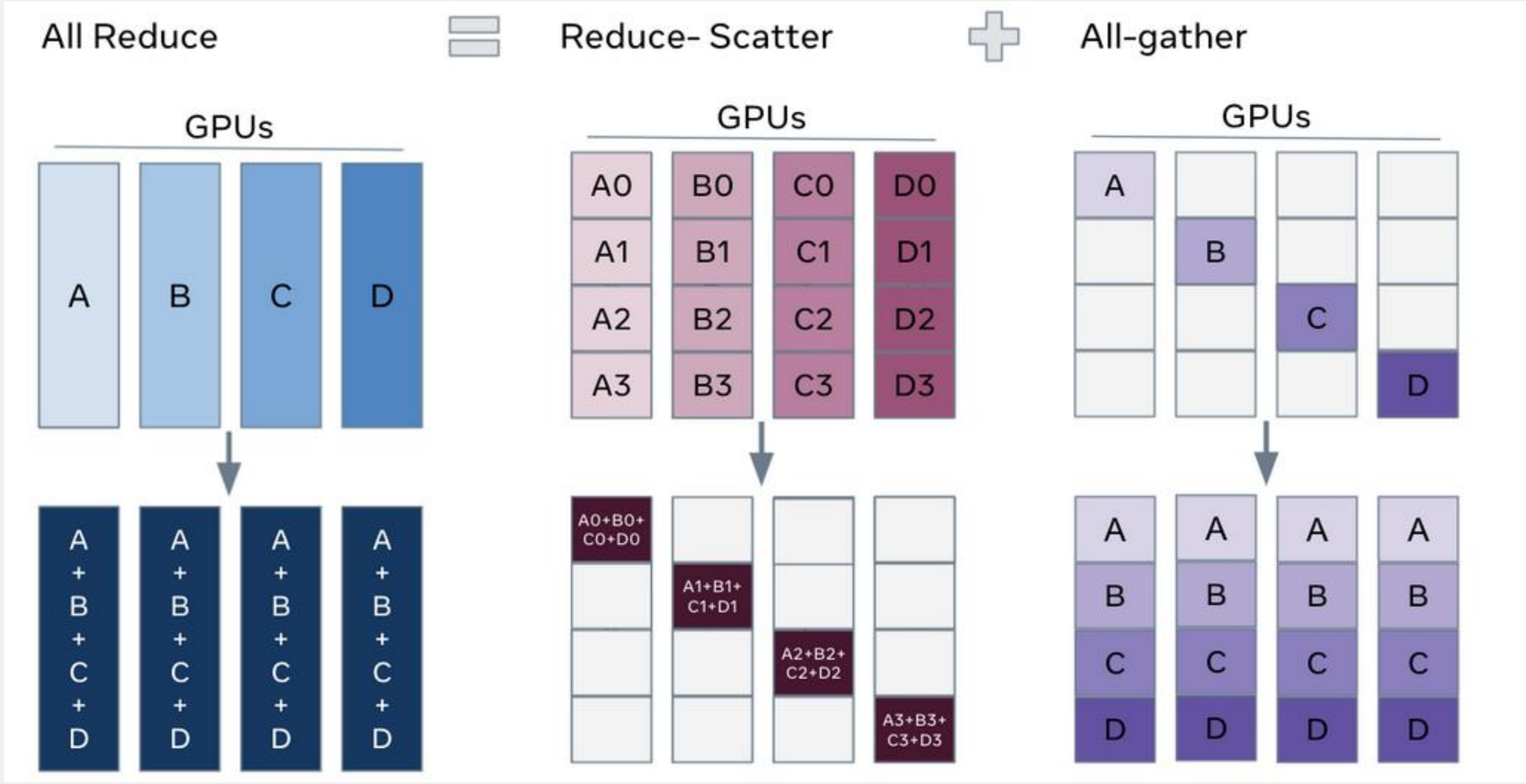
[1] https://medium.com/@pranay.janupalli/understanding-model-sharding-and-model-parallelism-scaling-large-language-models-dee6144d0591#:~:text=%E2%80%94%20Model%20Sharding:%20Usually%20involves%20a,types%20of%20operations%20across%20devices

COMMUNICATIONS

- **Reduce-Scatter**:
  one rank receive
  the result T

- **All-Gather**:
  each rank receives
  the contributions of other ranks
  (**but not the final result T**)

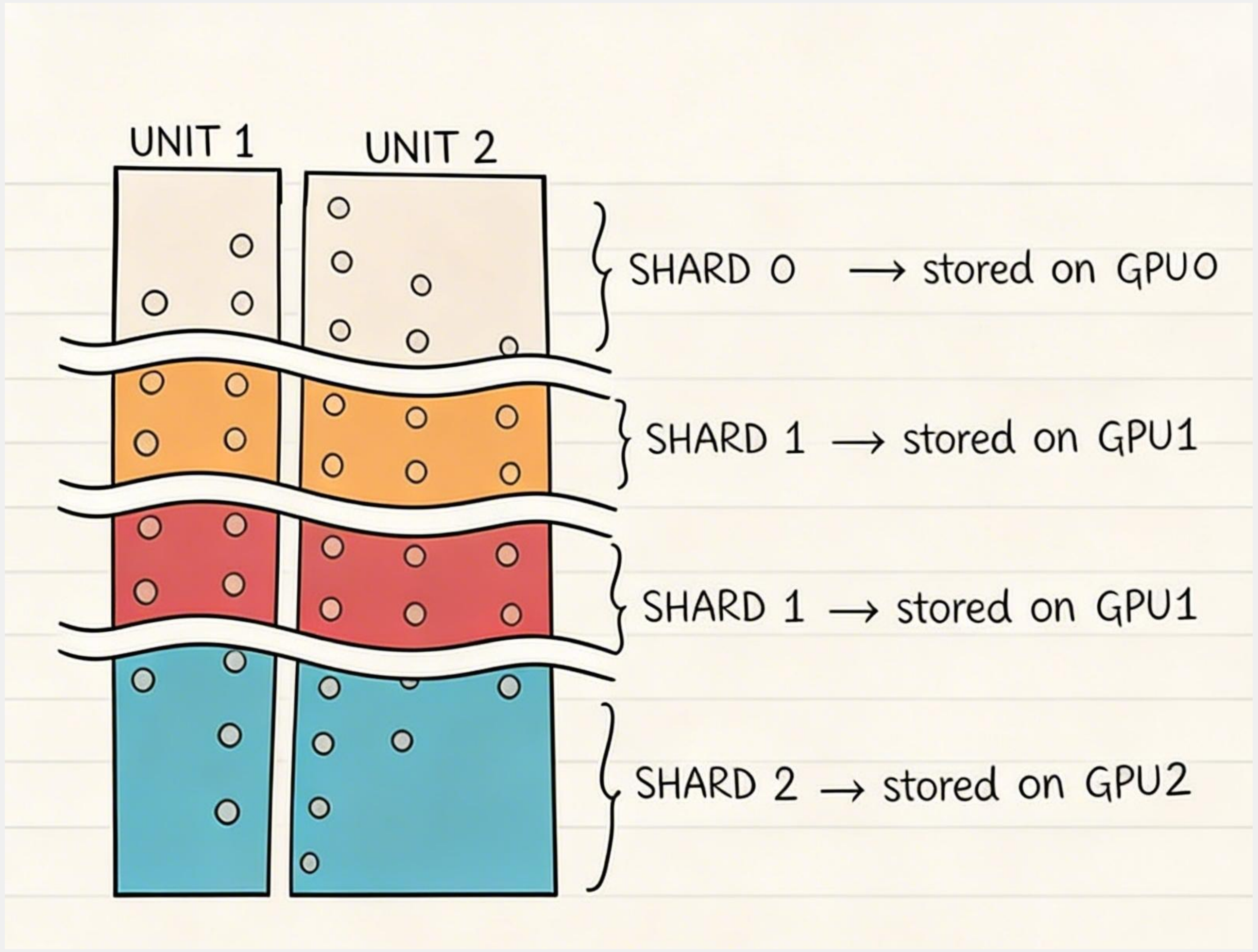
- **All-Reduce**:
  all ranks receive the result T



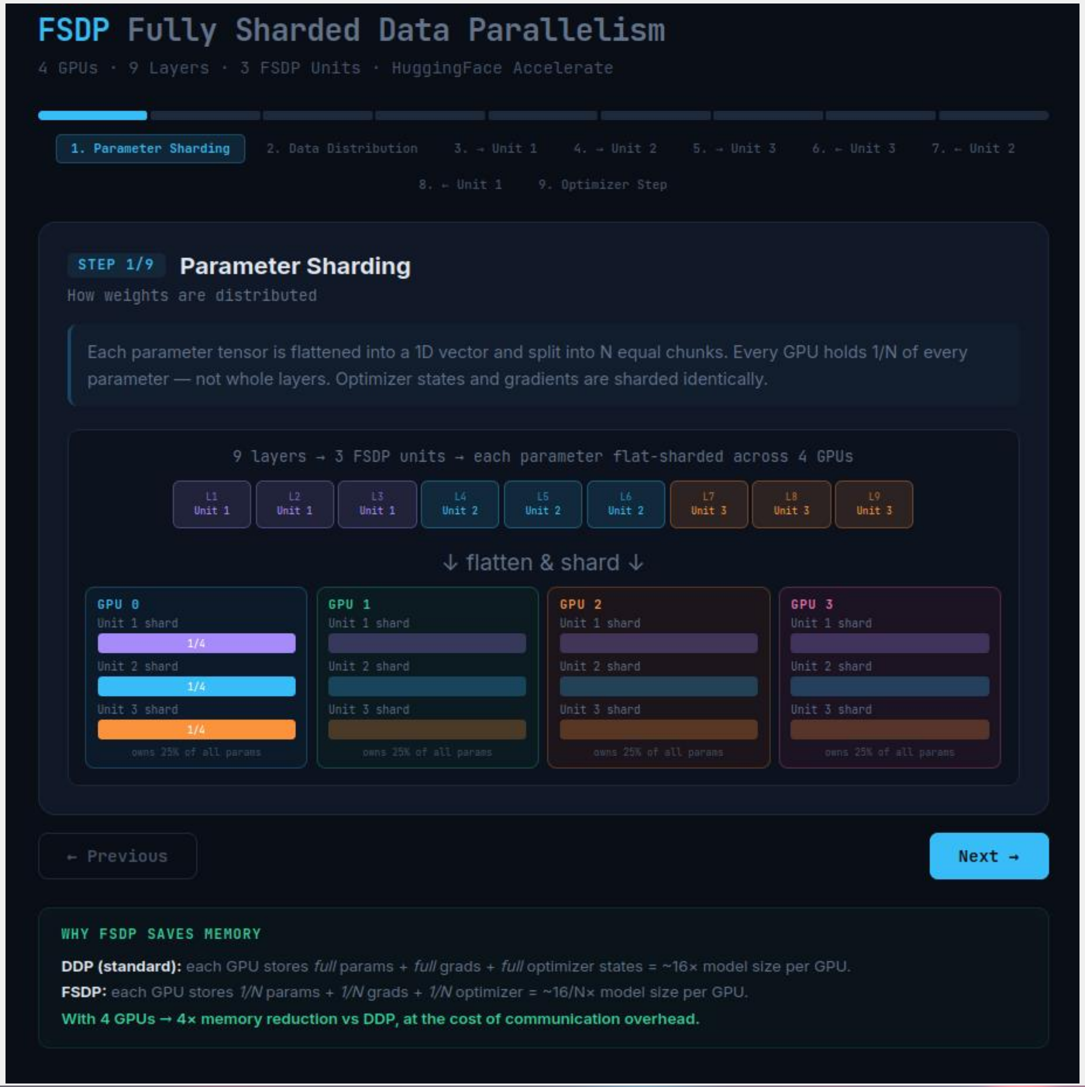
[1] https://docs.pytorch.org/tutorials/intermediate/FSDP_tutorial.html

FSDP

- 4 GPUS
- 5 layers
- 2 units -> 2 all gather in the fwd



FSDP



FSDP Fully Sharded Data Parallelism

4 GPUs · 9 Layers · 3 FSDP Units · HuggingFace Accelerate

STEP 1/9 **Parameter Sharding**

How weights are distributed

Each parameter tensor is flattened into a 1D vector and split into N equal chunks. Every GPU holds 1/N of every parameter — not whole layers. Optimizer states and gradients are sharded identically.

9 layers → 3 FSDP units → each parameter flat-sharded across 4 GPUs

WHY FSDP SAVES MEMORY

DDP (standard): each GPU stores *full* params + *full* grads + *full* optimizer states = ~16× model size per GPU.

FSDP: each GPU stores *1/N* params + *1/N* grads + *1/N* optimizer = ~16/N× model size per GPU.

With 4 GPUs → 4× memory reduction vs DDP, at the cost of communication overhead.

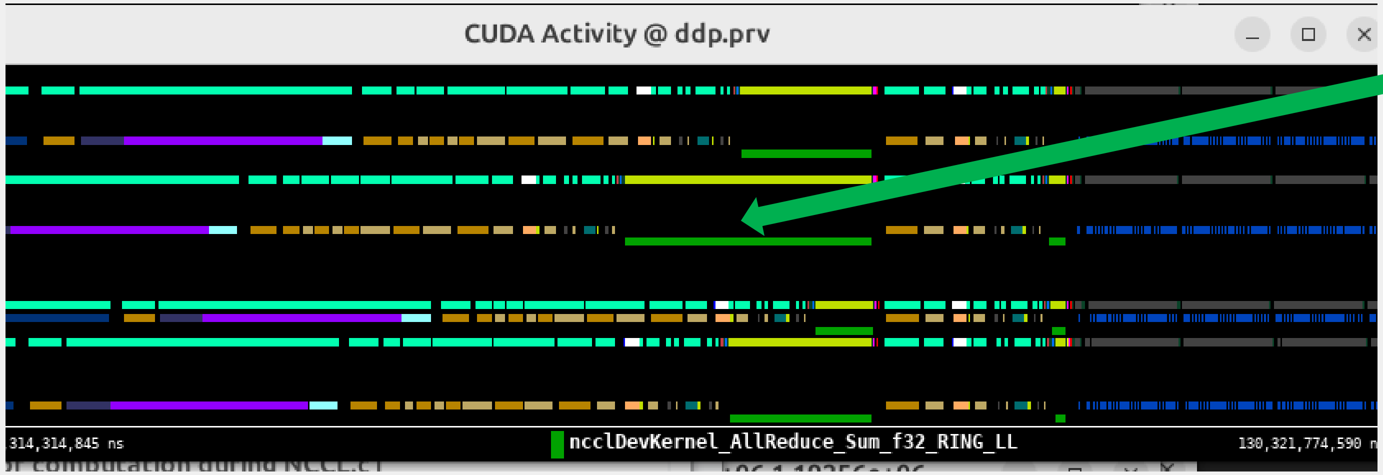
FSDP pseudocode

```
FSDP forward pass:
    for layer_i in layers:
        all-gather full weights for layer_i
        forward pass for layer_i
        discard full weights for layer_i

FSDP backward pass:
    for layer_i in layers:
        all-gather full weights for layer_i
        backward pass for layer_i
        discard full weights for layer_i
        reduce-scatter gradients for layer_i
```

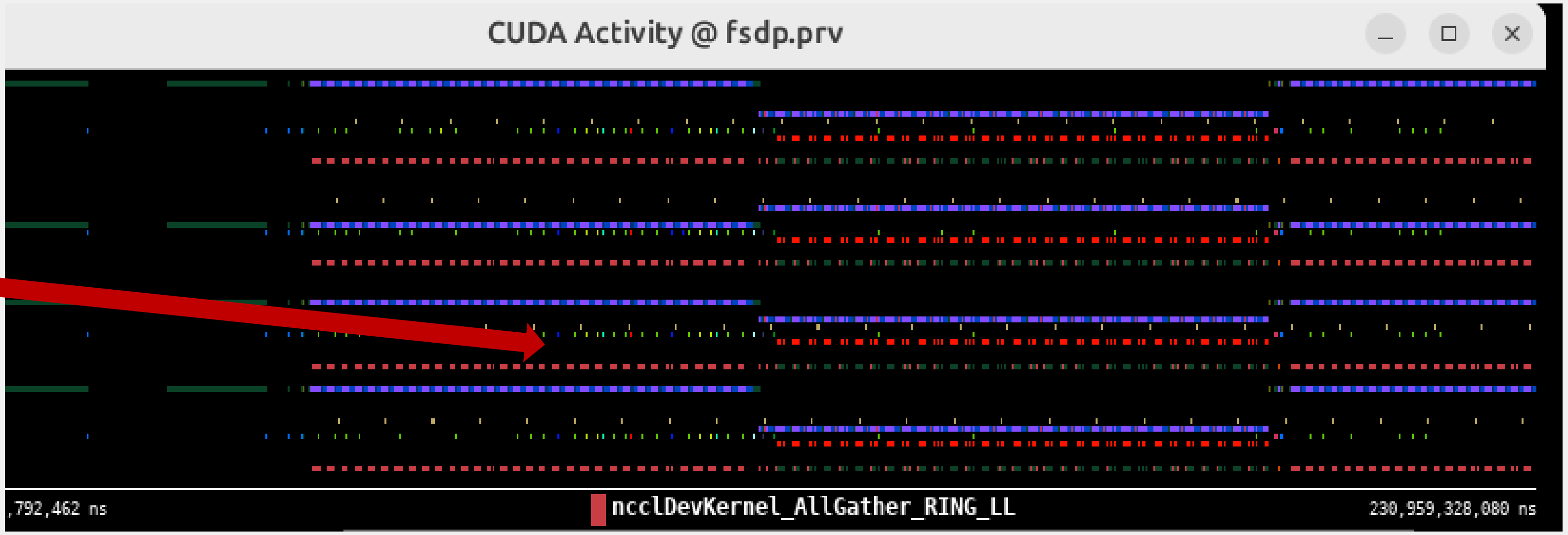
https://engineering.fb.com/2021/07/15/open-source/fsdp/

DDP VS FSDP


CUDA Activity @ ddp.prv

DDP

Bwd: All_reduce


CUDA Activity @ fsdp.prv

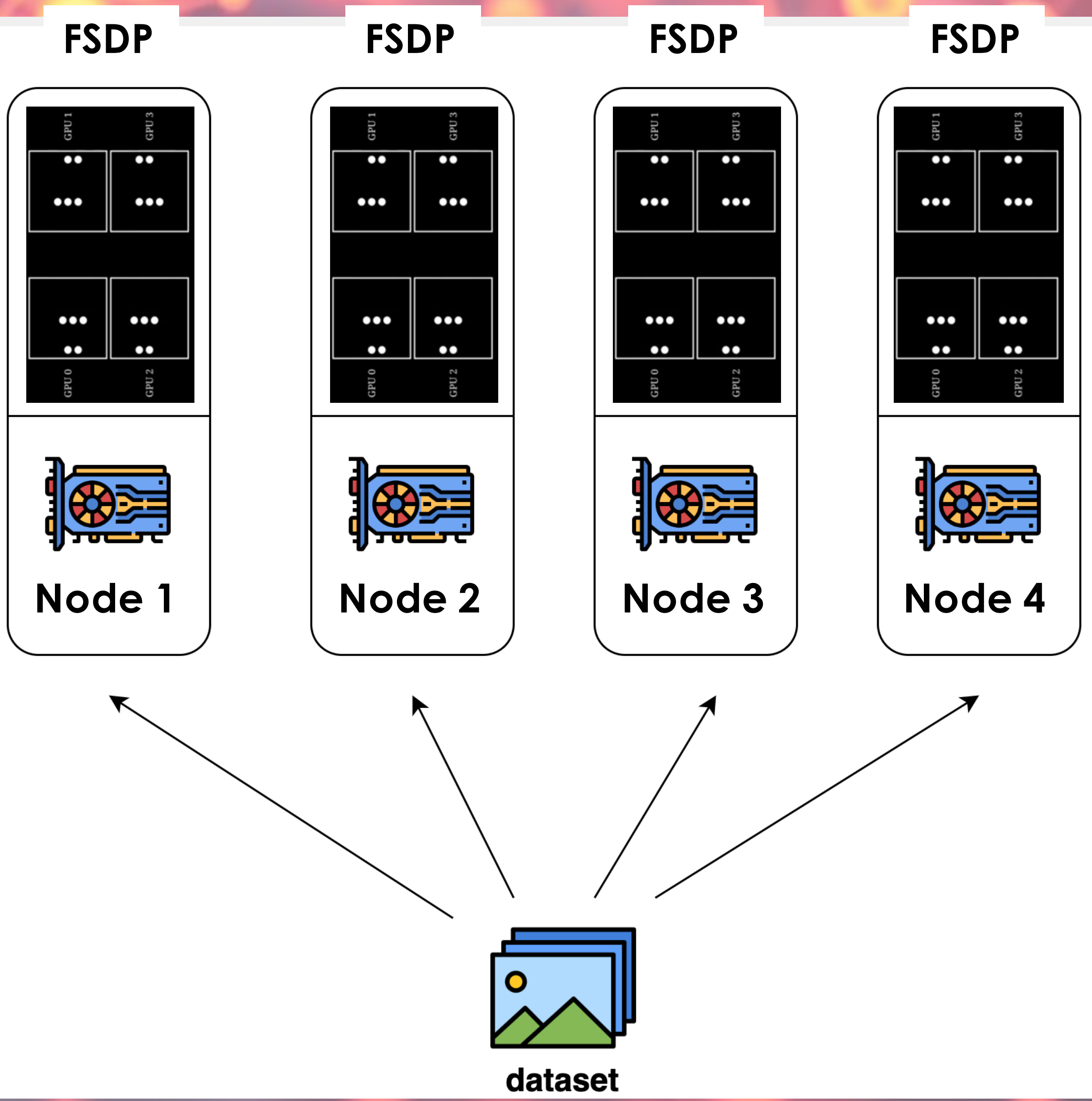
FSDP

Fwd: All_Gather

Bwd: All_Gather + Reduce Scatter

Hybrid shared data parallelism (HSDP)

- **HYBRID SHARD**: Apply FULL SHARD **within one node**, and **replicate** parameters across nodes.



Pro:
- **Reduced** communication volume

Cons:
- **More** GPU memory needed w.r.t. FSDP

TOOLS FOR MODEL SHARDING

1 Accelerate

2 DeepSpeed

3 Olmo-core

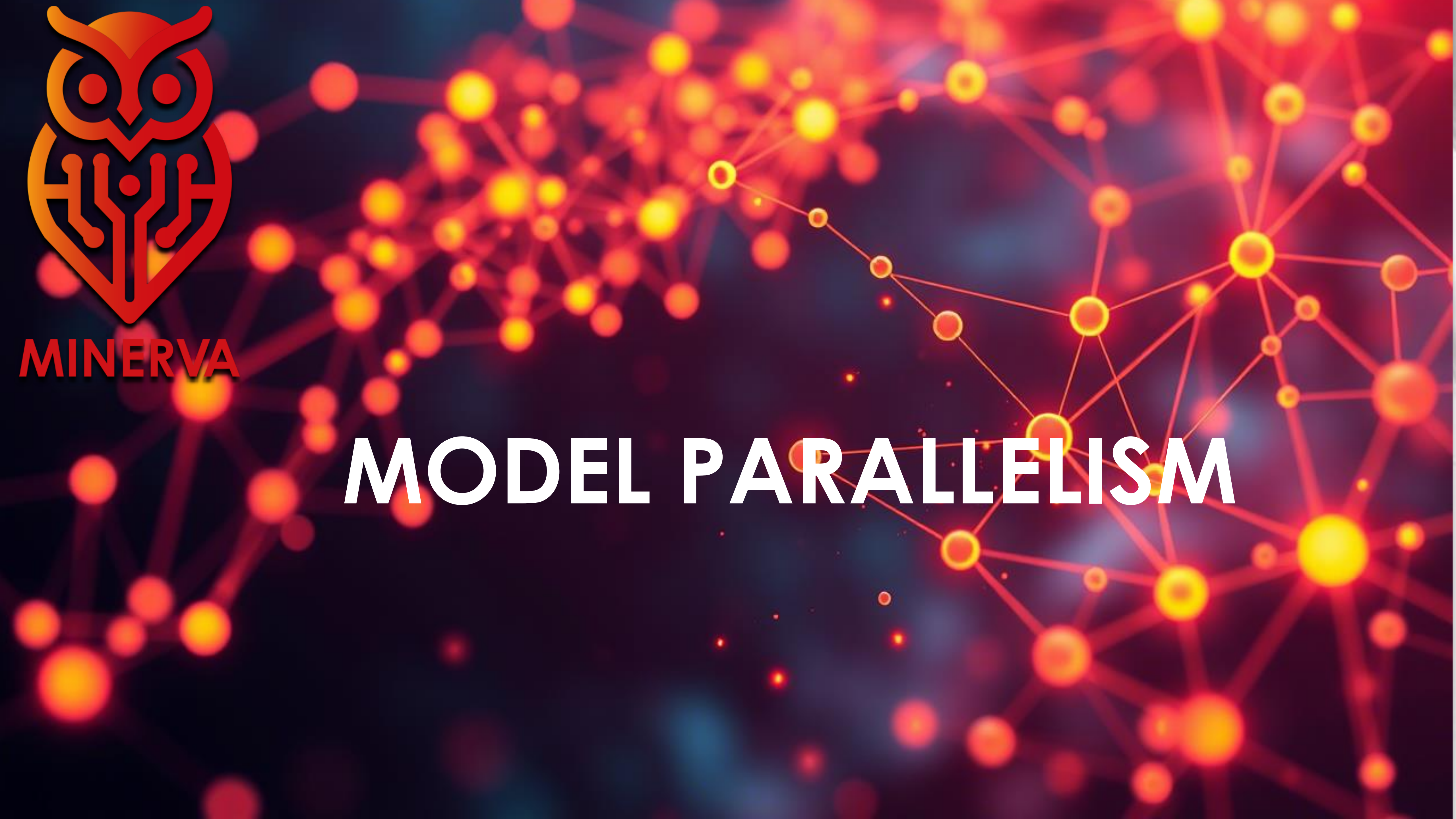
4 Modalities

FSDP EXERCISE

- Go in the FSDP subfolder
- Complete "FFT.py" exercise
- Complete "config_FFT_FSDP.yaml"
- Complete the jobscript "job_FFT.sh"
- Run the code with 2, 4,8 GPU
- Set **#SBATCH --reservation=s_tra_minwinsc** for accessing the reservation

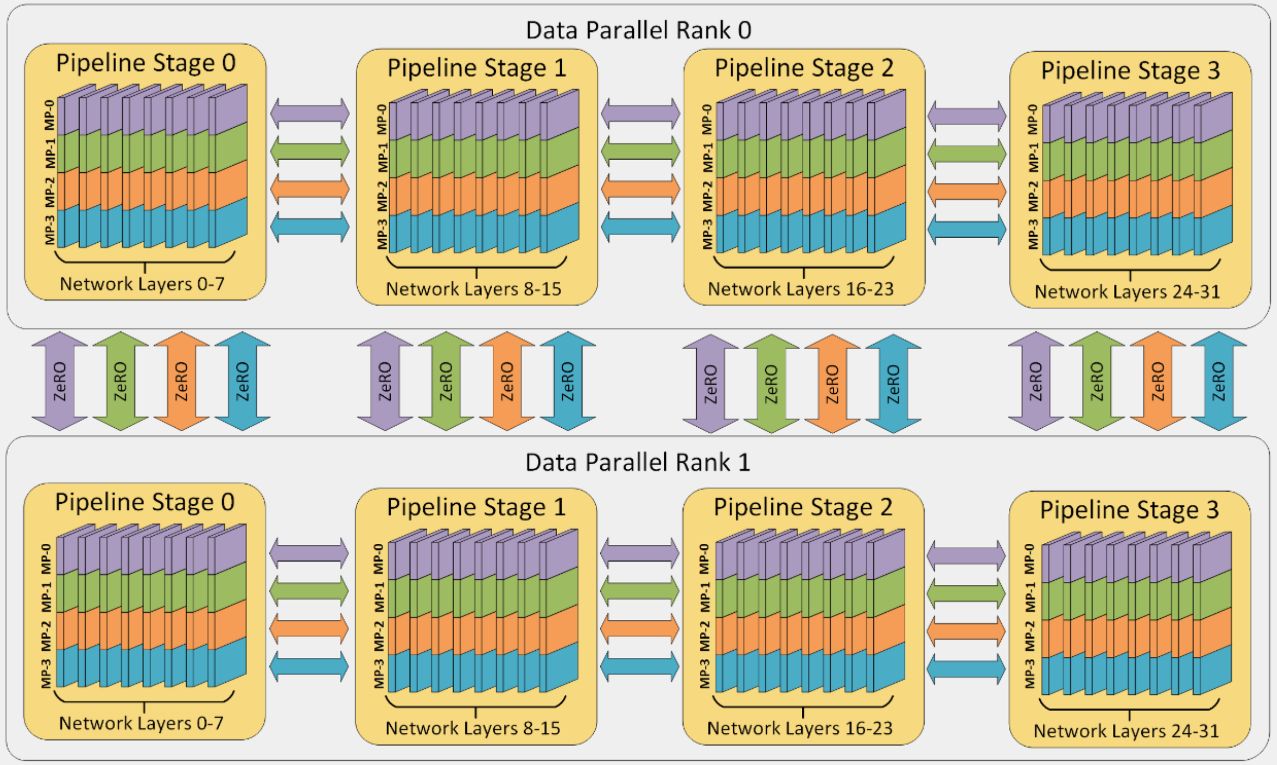
MINERVA

MODEL PARALLELISM



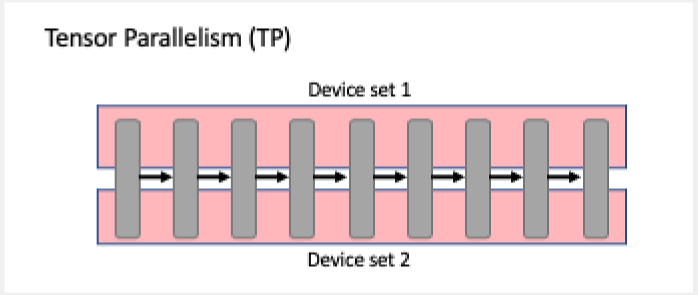
MODEL PARALLELISM

Model Parallelism is a technique used to split a neural network across multiple GPUs or nodes where **different GPUs handle different parts of the model.**

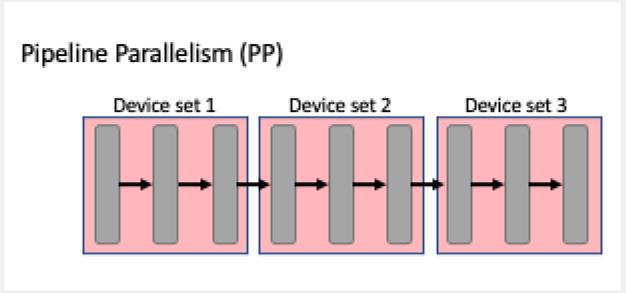


MODEL PARALLELISM

HORIZONTAL SPLIT – TENSOR PARALLELISM
Each tensor is split up into multiple chunks saved on different gpus. During processing each shard gets processed separately. Then, the results are synchronized at the end of the step.



VERTICAL SPLIT - PIPELINE PARALLELISM
The model is split at layer level on multiple gpus (i.e. only one or several layers are stored in the same GPU).
To avoid big pipeline bubbles each gpu process a portion of the batch (micro batch)

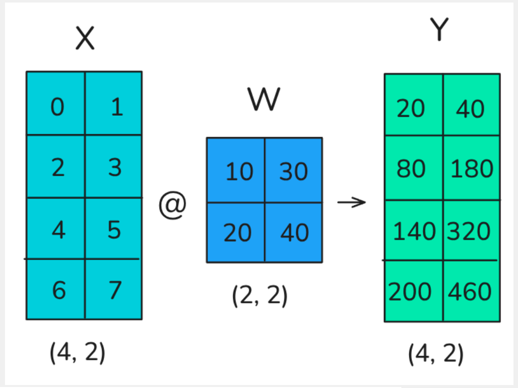


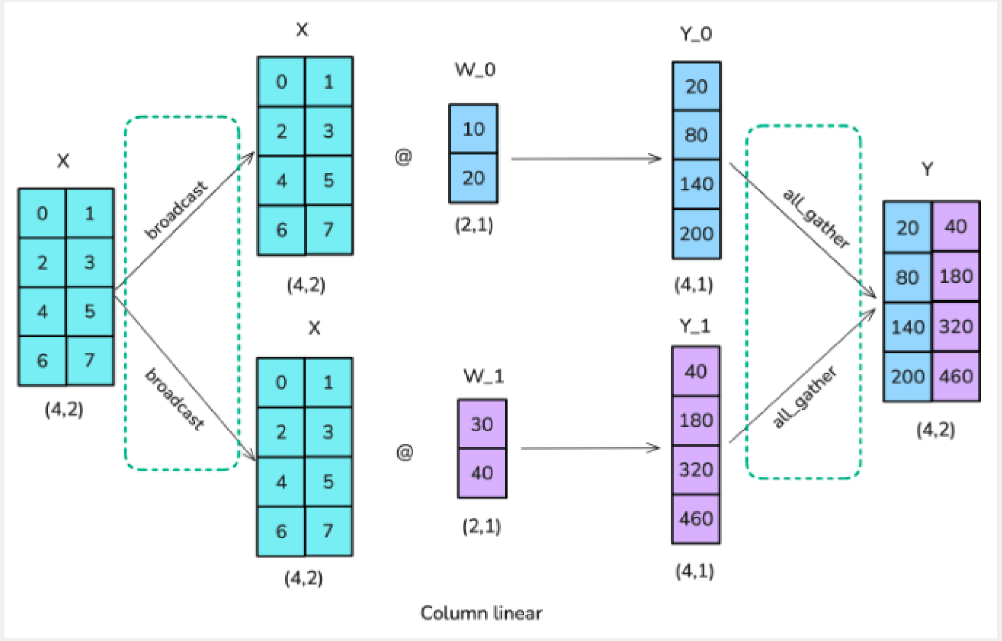
TENSOR PARALLELISM – HOW IT WORKS

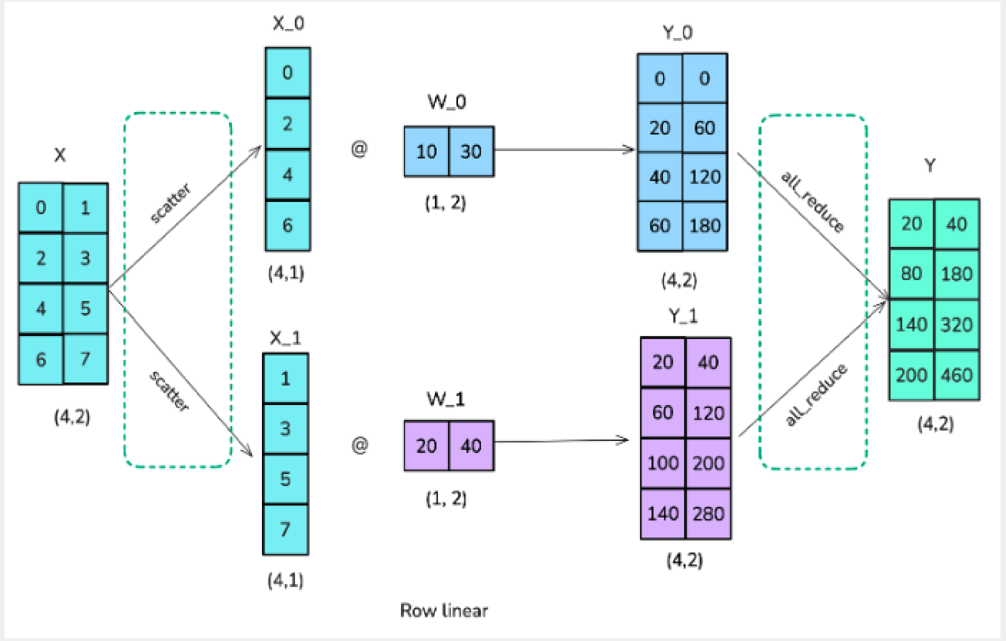
Given two matrices X and W, we can compute the matrix product XW by either :

- multiplying each **column** of W individually
- multiplying each **row** of W individually

and combining the results

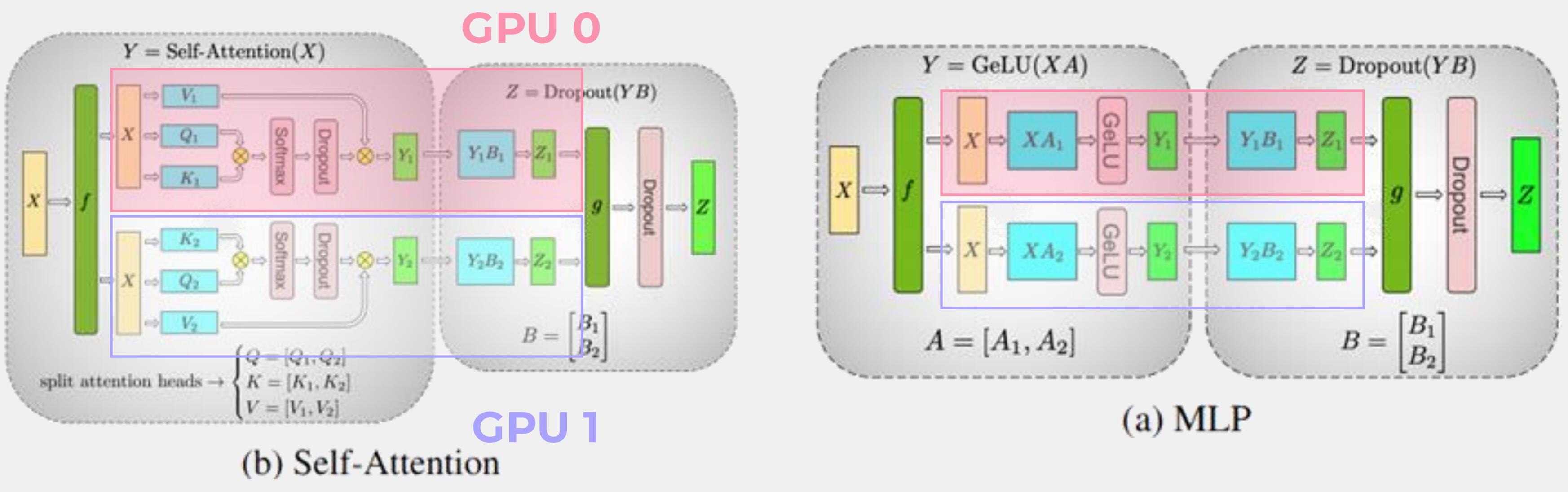






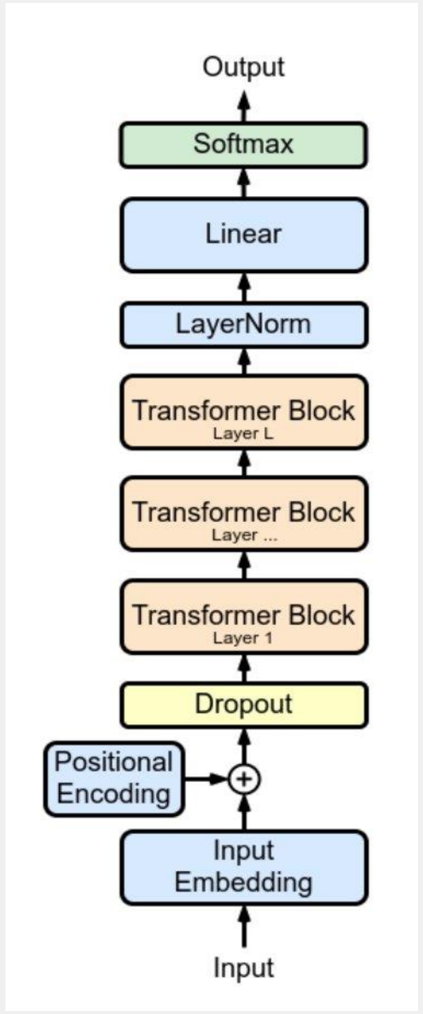
TENSOR PARALLELISM – HOW IT WORKS

A Transformer model is made of two main building blocks:
a feedforward multi-layer perceptron (MLP) block and a multi-head attention block.
We can apply tensor parallelism to both.

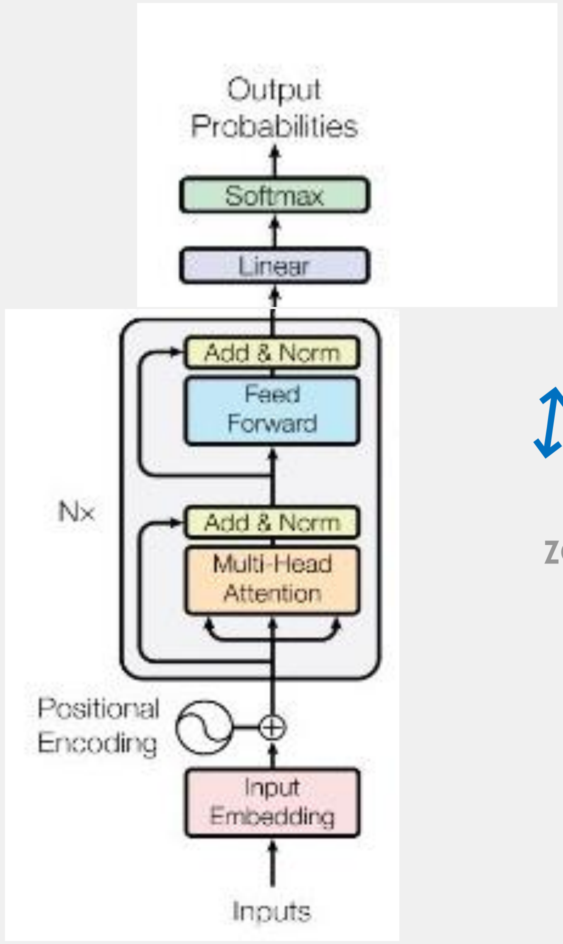


[1] https://github.com/huggingface/transformers/issues/10321#issuecomment-783543530
[2] https://arxiv.org/pdf/2104.04473

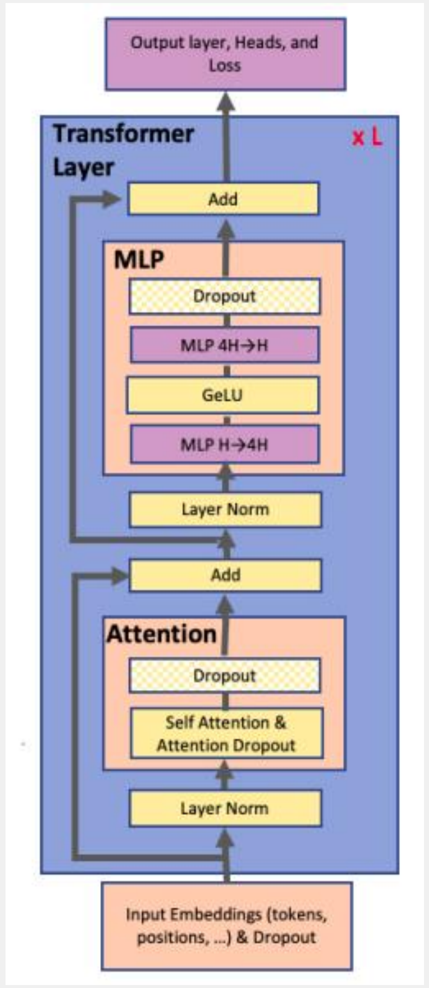
RECAP ON LLM ARCHITECTURE – TRANSFORMER BLOCK



ZOOM IN

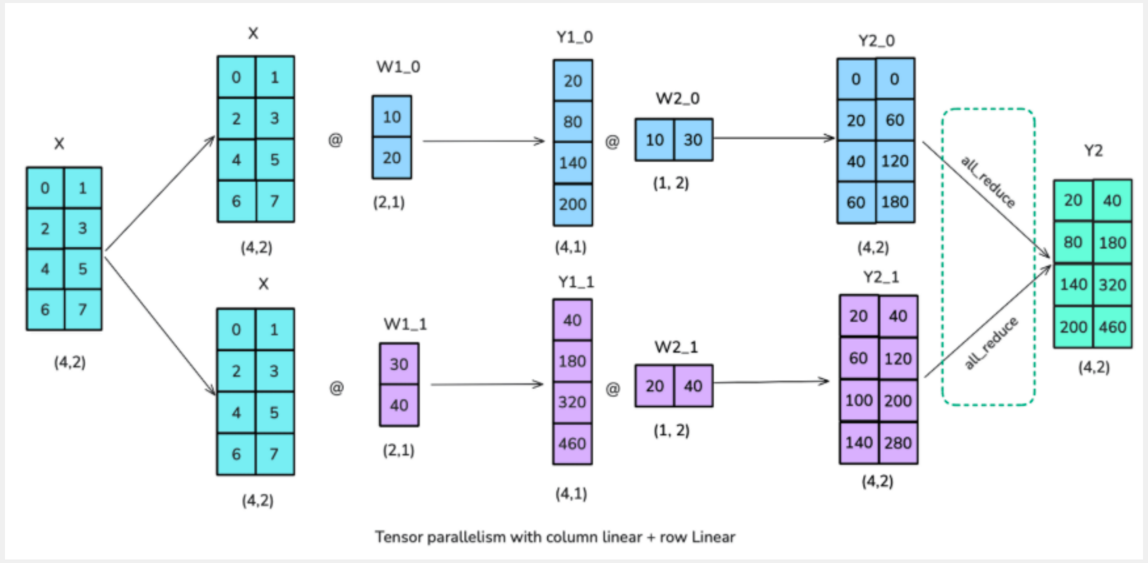


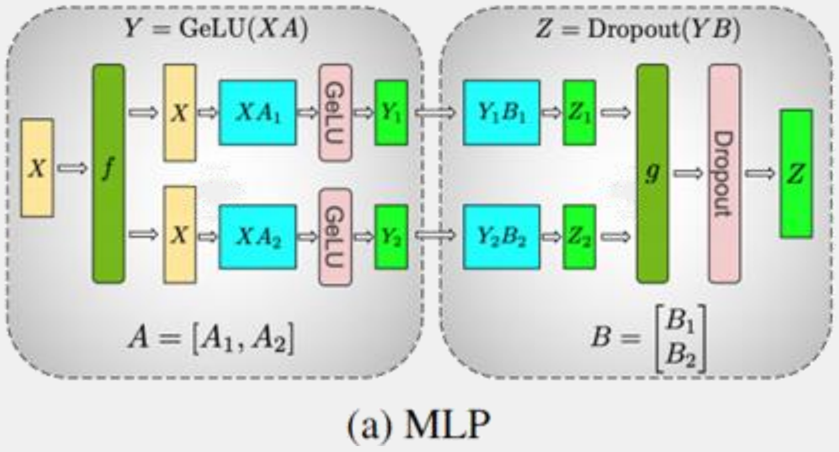
ZOOM IN


Output layer, Heads, and Loss
Transformer Layer
x L
Add
MLP
Dropout
MLP 4H→H
GeLU
MLP H→4H
Layer Norm
Add
Attention
Dropout
Self Attention & Attention Dropout
Layer Norm
Input Embeddings (tokens, positions, ...) & Dropout

TENSOR PARALLELISM IN MLP

The **feedforward** part (2-layer MLP) can be parallelized by having a column-linear followed by a row-linear split. This setup is more efficient than starting with a row-linear followed by column-linear split, as we can skip the intermediate all-reduce between the split operations.

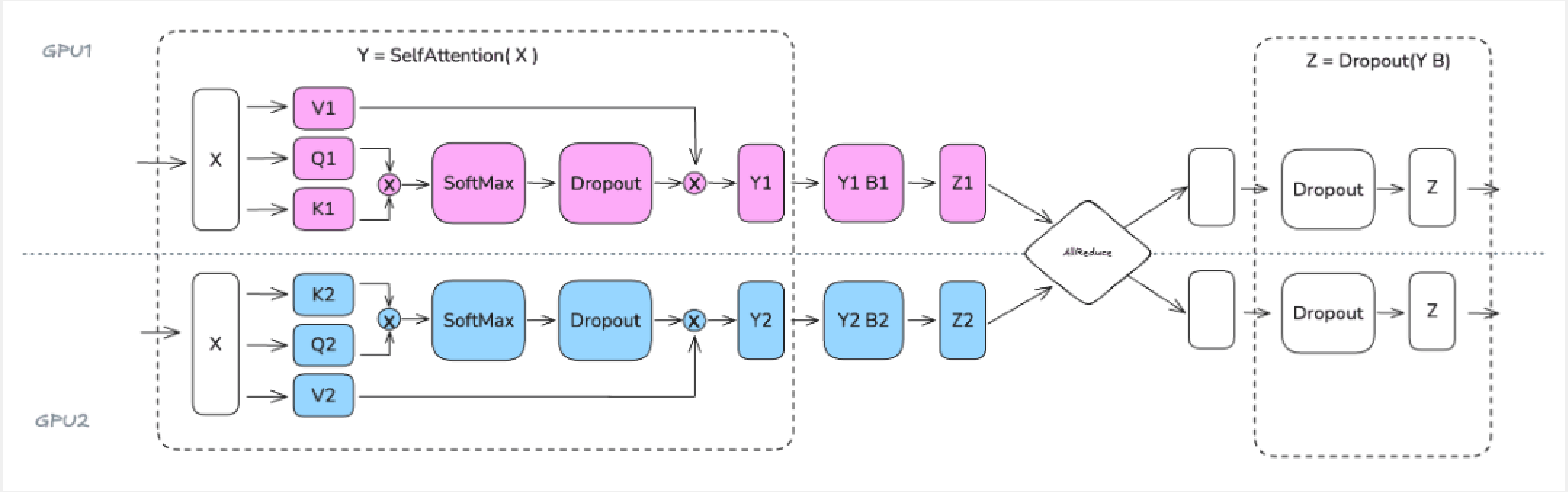




TENSOR PARALLELISM IN SELF ATTENTION

A similar approach can be used here, where

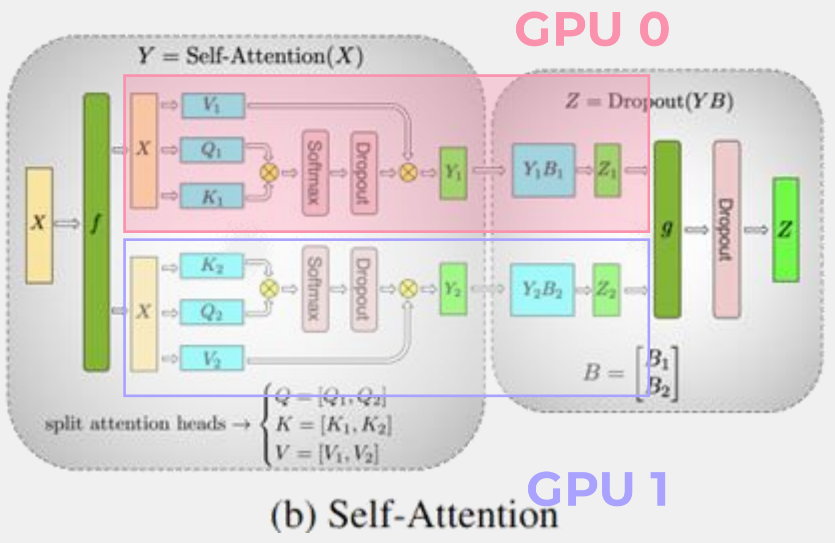
- Query (Q), Key (K), and Value (V) matrices are split in a **column**-parallel fashion *
- the output projection can be considered a **row**-linear in the GEMM after attention **



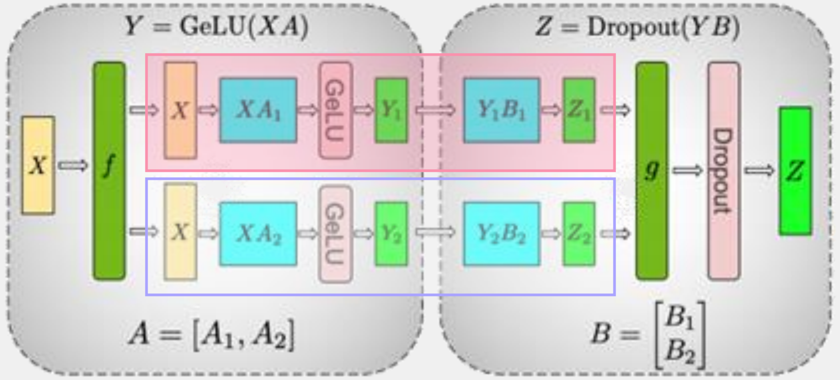
* in multihead attention operation, partitioning K,Q,W in a column parallel fashion results into each attention head is done locally on one GPU.
** this linear layer projects the combined features coming from the attention into a useful representation.

TENSOR PARALLELISM IN A TRANSFORMER BLOCK

Transformer block


GPU 0
$Y = \text{Self-Attention}(X)$
$Z = \text{Dropout}(YB)$
Softmax
Dropout
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$
split attention heads → $Q = [Q_1, Q_2]$, $K = [K_1, K_2]$, $V = [V_1, V_2]$
GPU 1

(b) Self-Attention



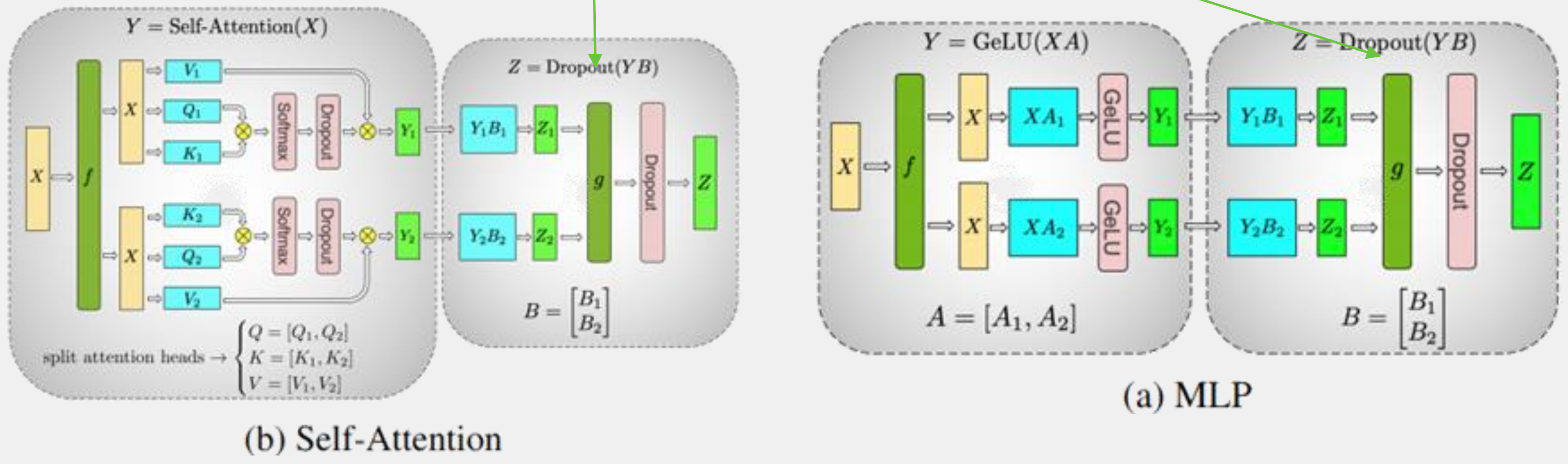
(a) MLP

[1] https://github.com/huggingface/transformers/issues/10321#issuecomment-783543530
[2] https://arxiv.org/pdf/2104.04473

TENSOR PARALLELISM COMM

2 all-reduce in the forward pass

Transformer block



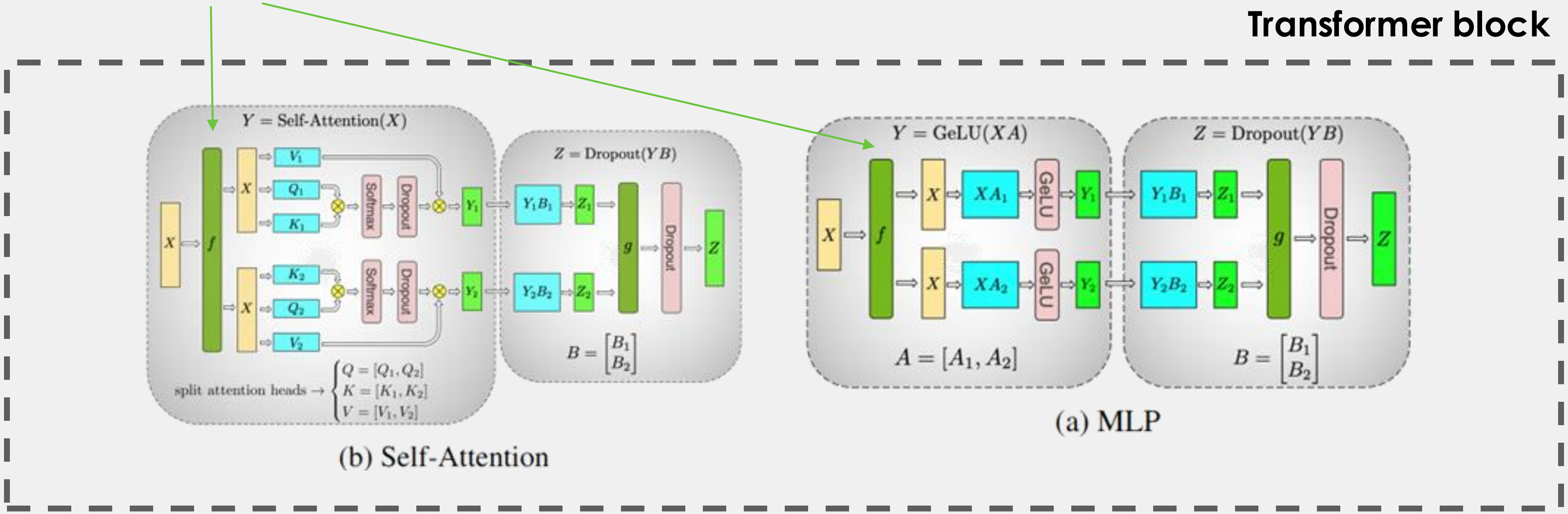
$Y = \text{Self-Attention}(X)$
$Z = \text{Dropout}(YB)$
$B = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}$
split attention heads → $Q = [Q_1, Q_2]$, $K = [K_1, K_2]$, $V = [V_1, V_2]$
Softmax
Dropout
f
g
(b) Self-Attention
$Y = \text{GeLU}(XA)$
$A = [A_1, A_2]$
GeLU
(a) MLP

[1] https://github.com/huggingface/transformers/issues/10321#issuecomment-783543530
[2] https://arxiv.org/pdf/2104.04473

TENSOR PARALLELISM COMM

2 all-reduce in the backward pass

Transformer block



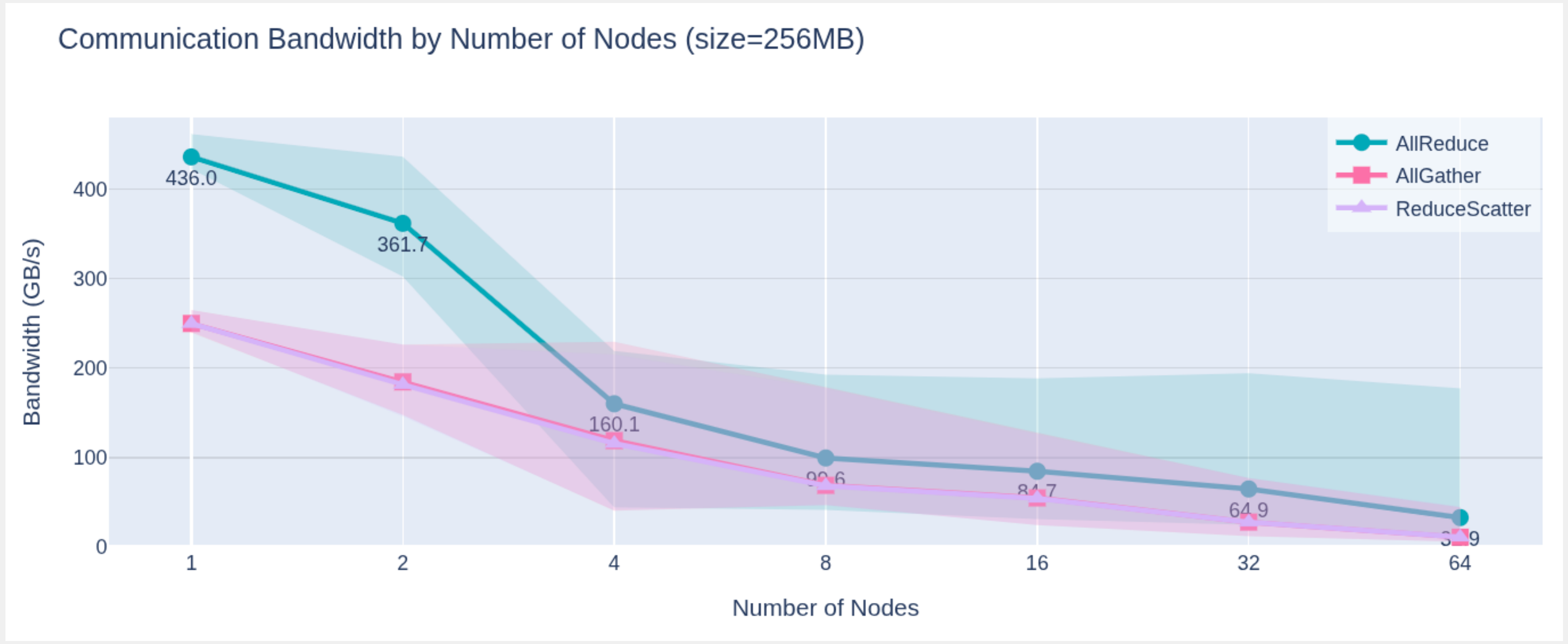
(b) Self-Attention

(a) MLP

[1] https://github.com/huggingface/transformers/issues/10321#issuecomment-783543530
[2] https://arxiv.org/pdf/2104.04473

DRAWBACK OF TENSOR PARALLELISM

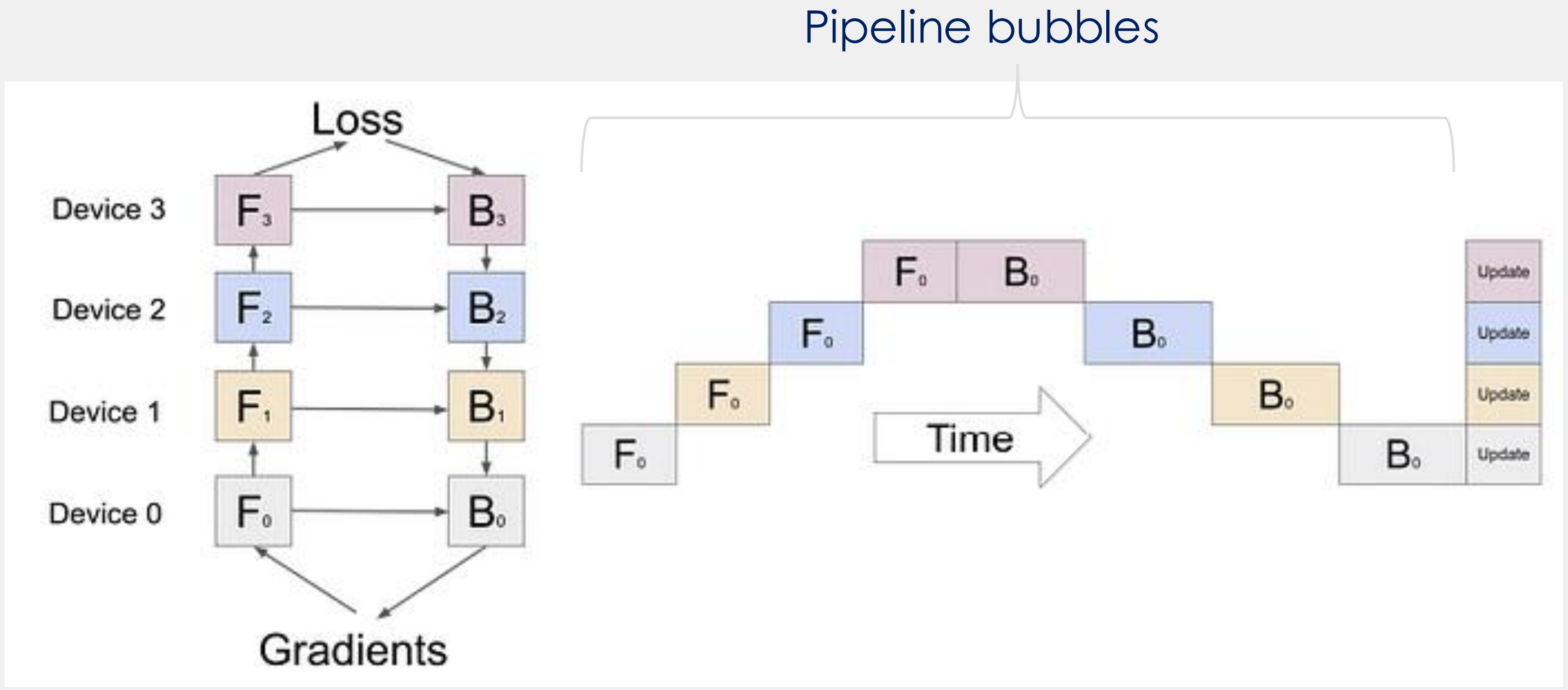
Trying to scale TP past the number of GPUs on a single node (TP>4 in our case) forces to use lower-bandwidth network communication (intra-node), which can significantly impair performance.



We can solve this issue by adding another parallelism dimension: **pipeline parallelism (PP).**

PIPELINE PARALLELISM

The model is split at layer level on multiple gpus.



[1] https://medium.com/byte-sized-ai/pipeline-parallelism-explained-in-2-mins-6bdf1ab29053
[2] https://arxiv.org/pdf/2104.04473

PIPELINE PARALLELISM

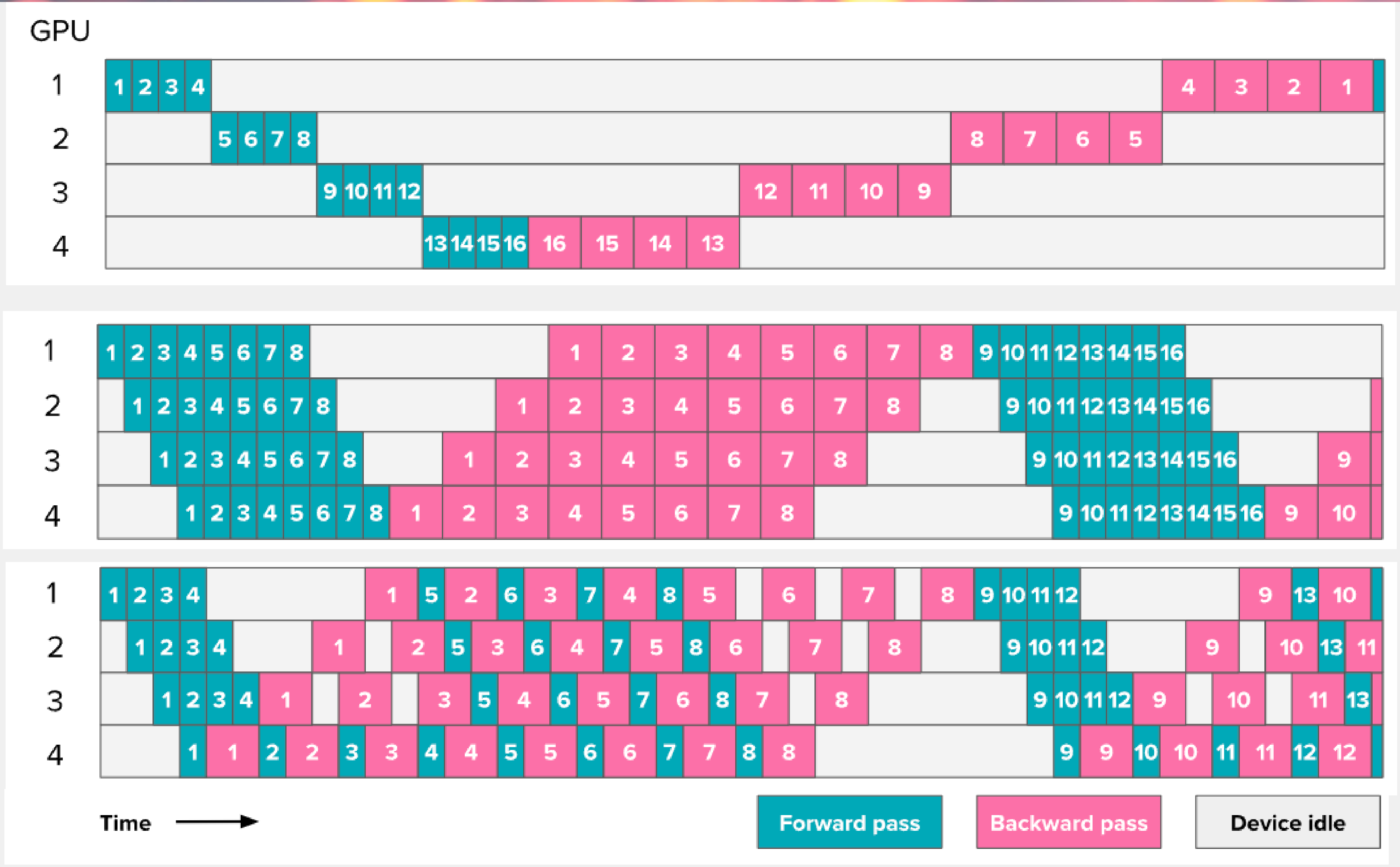
[1] https://docs.nvidia.com/nemo/megatron-bridge/0.2.0/parallelisms.html

PIPELINE PARALLELISM

Vanilla

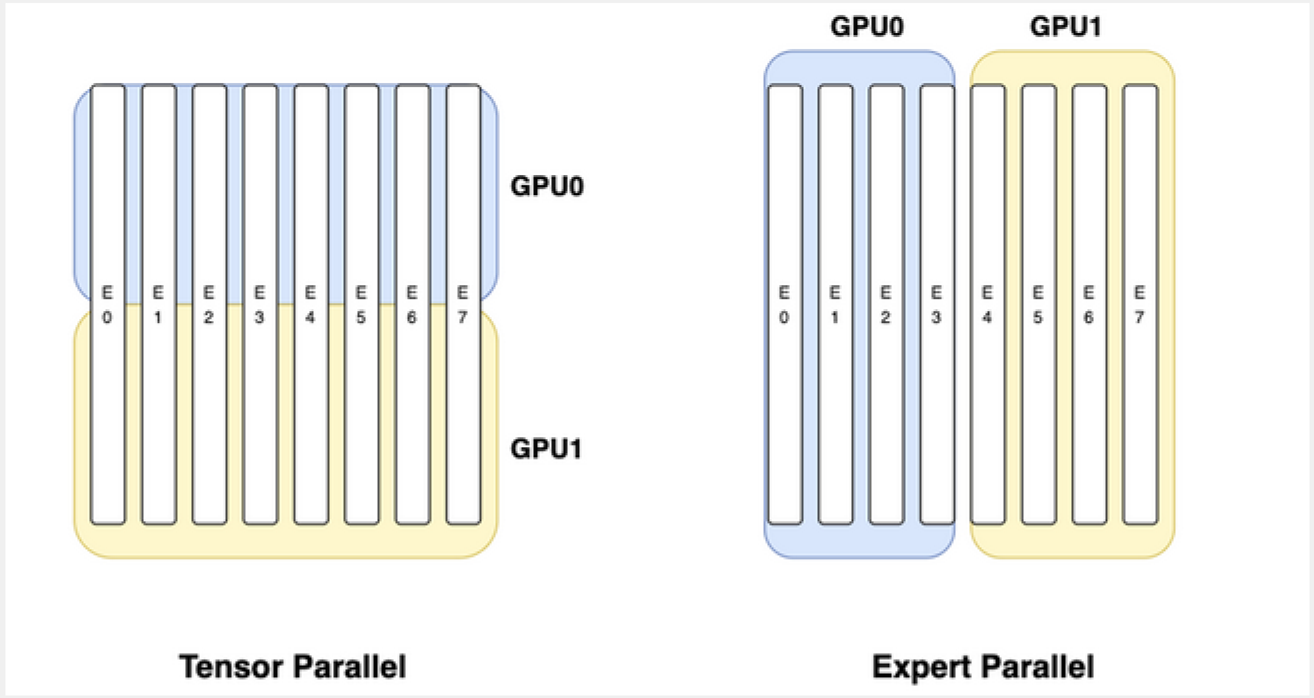
AFAB

1F1B



fwd: rcv/snd activ.
bwd: rcv/snd grad.

EXPERT PARALLELISM

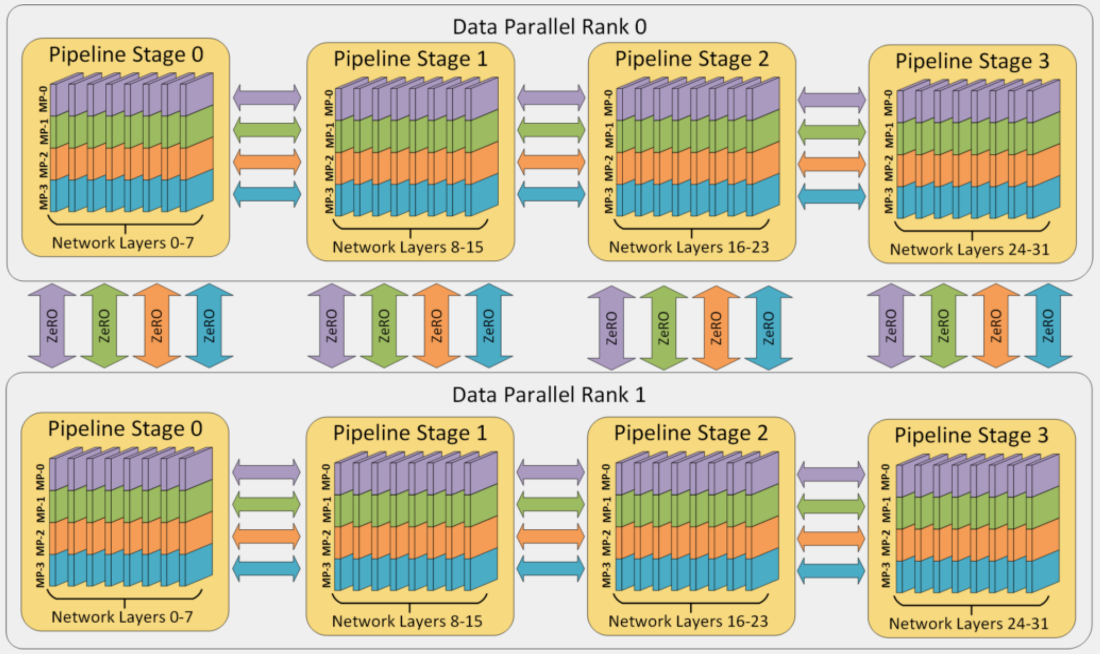


https://nvidia.github.io/TensorRT-LLM/advanced/expert-parallelism.html

HYBRID PARALLELISM

- **DP + PP (2D parallelism)**
- **DP+PP+TP (3D parallelism)**
- **DP+PP+TP+ZeRO1**
- **PP + TP + EP**
- **DP + PP + TP + SP/CP + EP (5D parallelism)**

3D parallelism



https://www.microsoft.com/en-us/research/blog/deepspeed-extreme-scale-model-training-for-everyone /

Interesting article on 5D parallelism :
https://huggingface.co/spaces/nanotron/ultrascale-playbook?section=high_level_overview

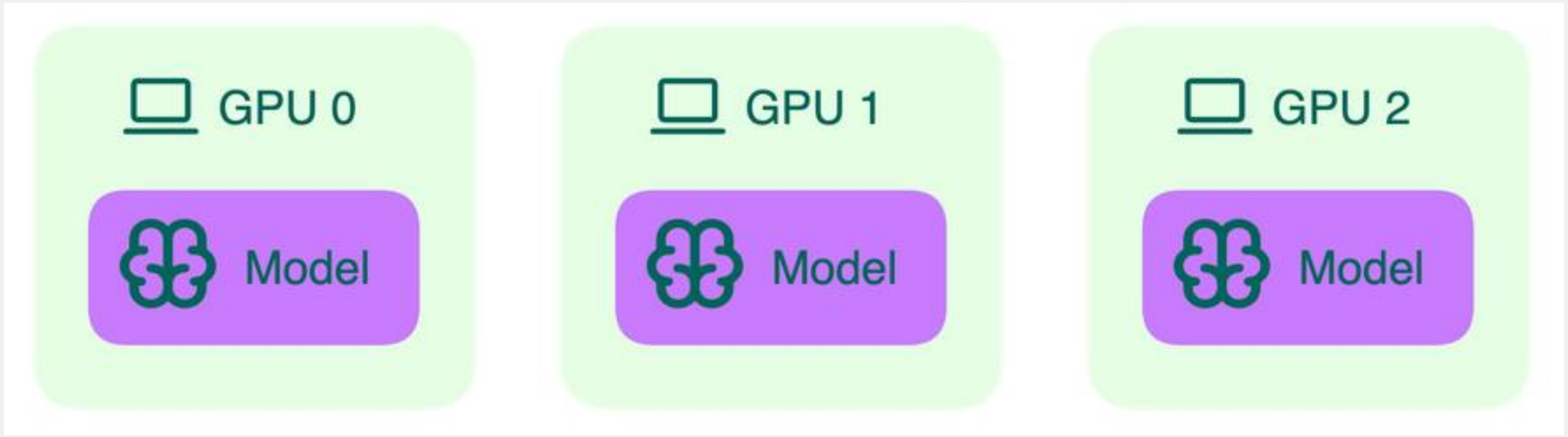
MEGATRON-LM

- Create singularity container
- Convert a model checkpoint in Megatron-LM format (not needed if you train from scratch)
- Tokenize the dataset and create a .pbin file

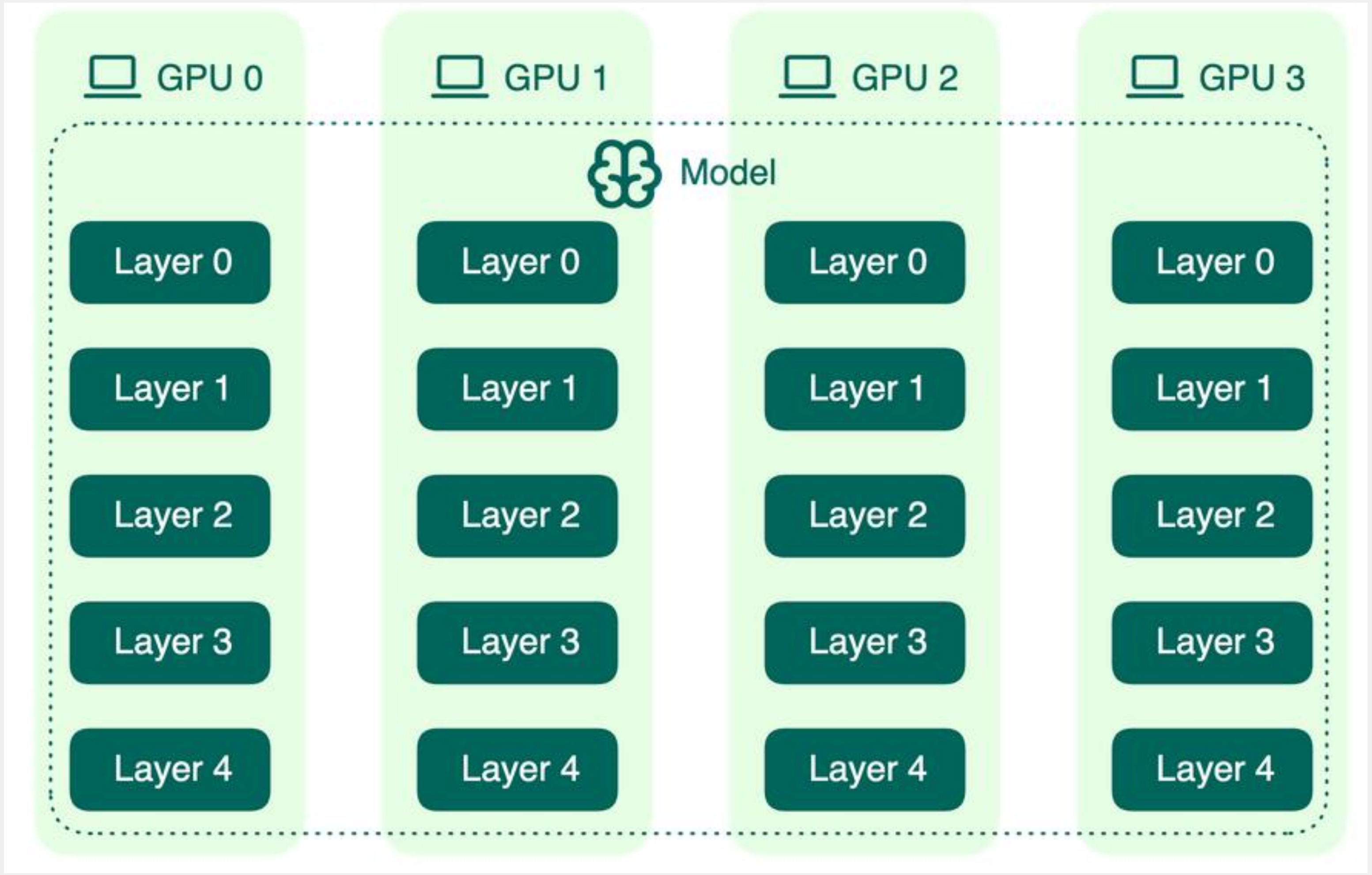
MEGATRON-LM EXERCISE

- Create your mode architecture (less than 8B params!)
- Complete the jobscript launch_megatron_LMM.sh
- Do some tests with different hyperparams
- How did you choose TP and PP?

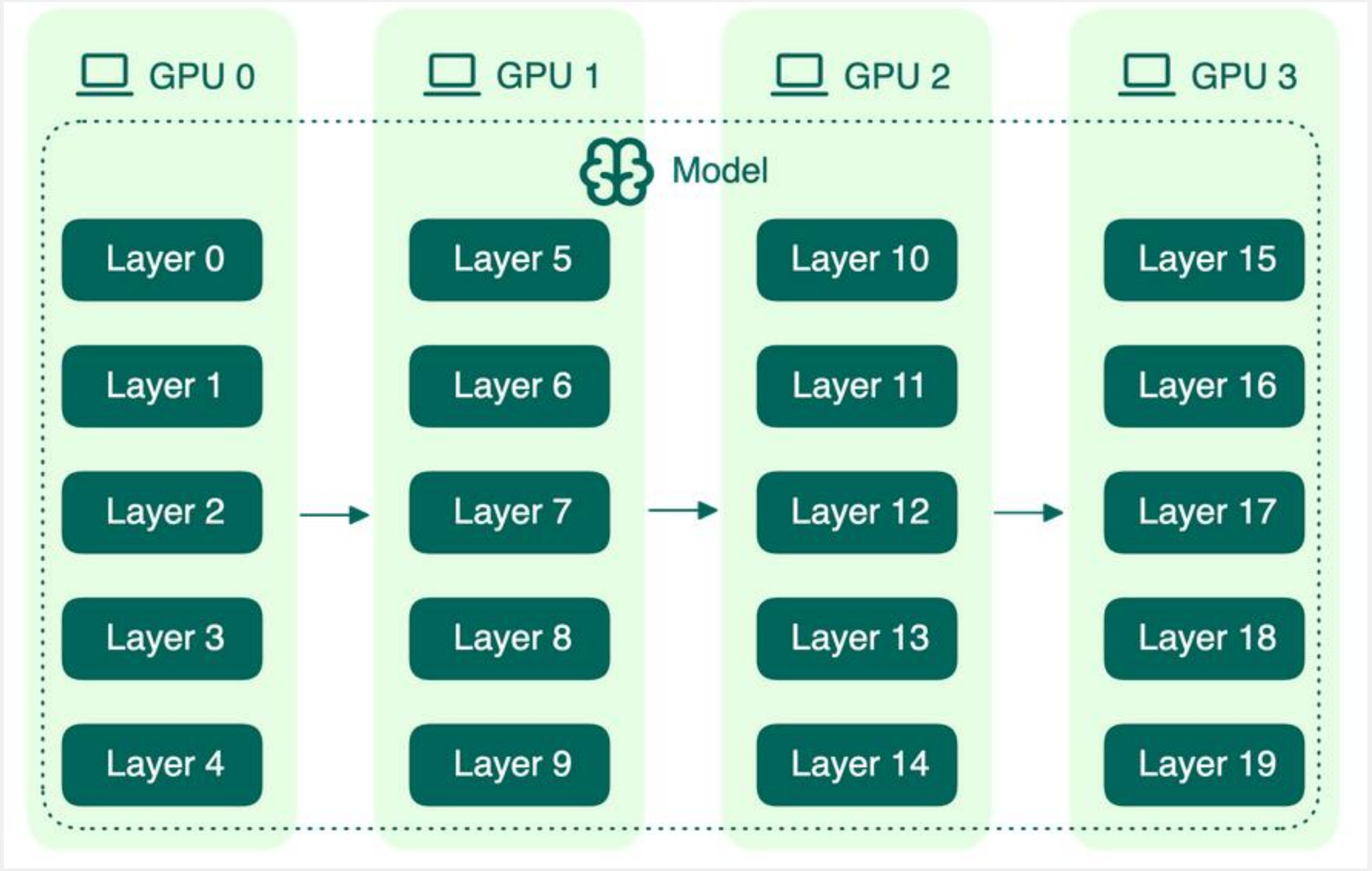
TEST YOUR KNOWLEDGE



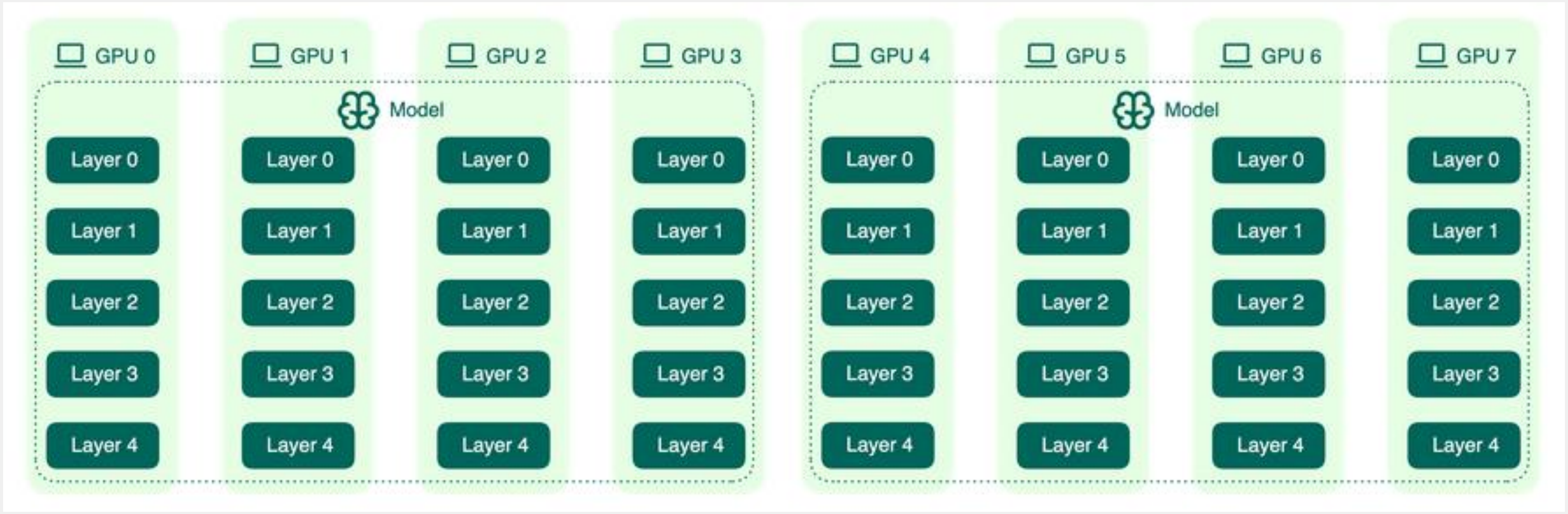
TEST YOUR KNOWLEDGE



TEST YOUR KNOWLEDGE



TEST YOUR KNOWLEDGE



RESULTS – OLMO-CORE

Size	#gpus	#replicas	GBS	MBS*	parallelism	Avg Tflops	Max Tflops
1B	8	2	256	4	HSDP	174	186
1B	16	2	512	4	HSDP	142	190
1B	32	2	1024	4	HSDP	140	190
8B	8	1	512	2	FSDP	118	119
8B	16	1	1024	2	FSDP	117	118
8B	32	1	2048	2	FSDP	112	113
8B	8	2	512	1	HSDP	-----	----
8B	16	2	1024	1	HSDP	82	81
8B	32	2	2048	1	HSDP	80	80
8B	32	2	512	2	HSDP	101	104

*This must evenly divide into the global batch size by a factor of the data parallel world size. If this is less than the global batch divided by the data parallel world size then gradient accumulation is used.

MINERVA

SCALING HSDP

Size	#gpus	#replicas	GBS	MBS	parallelism	Avg Tflops	Max Tflops
1B	8	2	256	4	HSDP	174	186
1B	16	2	512	4	HSDP	142	190
1B	32	2	1024	4	HSDP	140	190
8B	8	1	512	2	FSDP	118	119
8B	16	1	1024	2	FSDP	117	118
8B	32	1	2048	2	FSDP	112	113
8B	8	2	512	1	HSDP	-----	----
8B	16	2	1024	1	HSDP	82	81
8B	32	2	2048	1	HSDP	80	80
8B	32	2	512	2	HSDP	101	104

HSDP MEMORY ISSUES

Size	#gpus	#replicas	GBS	MBS	parallelism	Avg Tflops	Max Tflops
1B	8	2	256	4	HSDP	174	186
1B	16	2	512	4	HSDP	142	190
1B	32	2	1024	4	HSDP	140	190
8B	8	1	512	2	FSDP	118	119
8B	16	1	1024	2	FSDP	117	118
8B	32	1	2048	2	FSDP	112	113
8B	8	2	512	1	HSDP	-----	----
8B	16	2	1024	1	HSDP	82	81
8B	32	2	2048	1	HSDP	80	80
8B	32	2	512	2	HSDP	101	104

MEGATRON-LM : How to keep constant Tflops

Size	#gpus	TP	PP	GBS	MBS	Avg Tflops	Max Tflops
8B	8	4	2	512	1	160	162
8B	16	4	2	512	1	155	158
8B	16	4	2	512	2	160	163
8B	32	4	2	512	1	144	147
8B	32	4	2	512	2	149	152
8B	64	4	2	512	2	121	136
8B	128	4	2	512	2	109	113
8B	256	4	2	512	1	79	83
8B	256	4	2	1024	1	100	109

MEGATRON-LM : How to keep constant Tflops

Size	#gpus	TP	PP	GBS	MBS	Avg Tflops	Max Tflops
8B	8	4	2	512	1	160	162
8B	16	4	2	512	1	155	158
8B	16	4	2	512	2	160	163
8B	32	4	2	512	1	144	147
8B	32	4	2	512	2	149	152
8B	64	4	2	512	2	121	136
8B	128	4	2	512	2	109	113
8B	256	4	2	512	1	79	83
8B	256	4	2	1024	1	100	109

MEGATRON-LM : Microbatch size matters

Size	#gpus	TP	PP	GBS	MBS	Avg Tflops	Max Tflops
8B	8	4	2	512	1	160	162
8B	16	4	2	512	1	155	158
8B	16	4	2	512	2	160	163
8B	32	4	2	512	1	144	147
8B	32	4	2	512	2	149	152
8B	64	4	2	512	2	121	136
8B	128	4	2	512	2	109	113
8B	256	4	2	512	1	79	83
8B	256	4	2	1024	1	100	109

MEGATRON-LM : Microbatch size matters

Size	#gpus	TP	PP	GBS	MBS	Avg Tflops	Max Tflops
8B	8	4	2	512	1	160	162
8B	16	4	2	512	1	155	158
8B	16	4	2	512	2	160	163
8B	32	4	2	512	1	144	147
8B	32	4	2	512	2	149	152
8B	64	4	2	512	2	121	136
8B	128	4	2	512	2	109	113
8B	256	4	2	512	1	79	83
8B	256	4	2	1024	1	100	109

MINERVA

MEGATRON-LM : Larger model, higher PP

Size	#gpus	TP	PP	GBS	MBS	Avg Tflops	Max Tflops
8B	8	4	2	512	1	160	162
8B	16	4	2	512	1	155	158
8B	16	4	2	512	2	160	163
8B	32	4	2	512	2	149	152
32B	64	4	16	512	1	110	115
32B	64	4	16	1024	1	114	116
32B	128	4	16	1024	1	110	111
70B	128	4	16	1024	1	96	114

MEGATRON-LM : Larger models, lower tflops

Size	#gpus	TP	PP	GBS	MBS	Avg Tflops	Max Tflops
8B	8	4	2	512	1	160	162
8B	16	4	2	512	1	155	158
8B	16	4	2	512	2	160	163
8B	32	4	2	512	2	149	152
32B	64	4	16	512	1	110	115
32B	64	4	16	1024	1	114	116
32B	128	4	16	1024	1	110	111
70B	128	4	16	1024	1	96	114

Reach out to us @ *info@minerva4ai.eu*

Thank you

Co-funded by the European Union

EuroHPC Joint Undertaking

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 101182737. The JU receives support from the Digital Europe Programme.