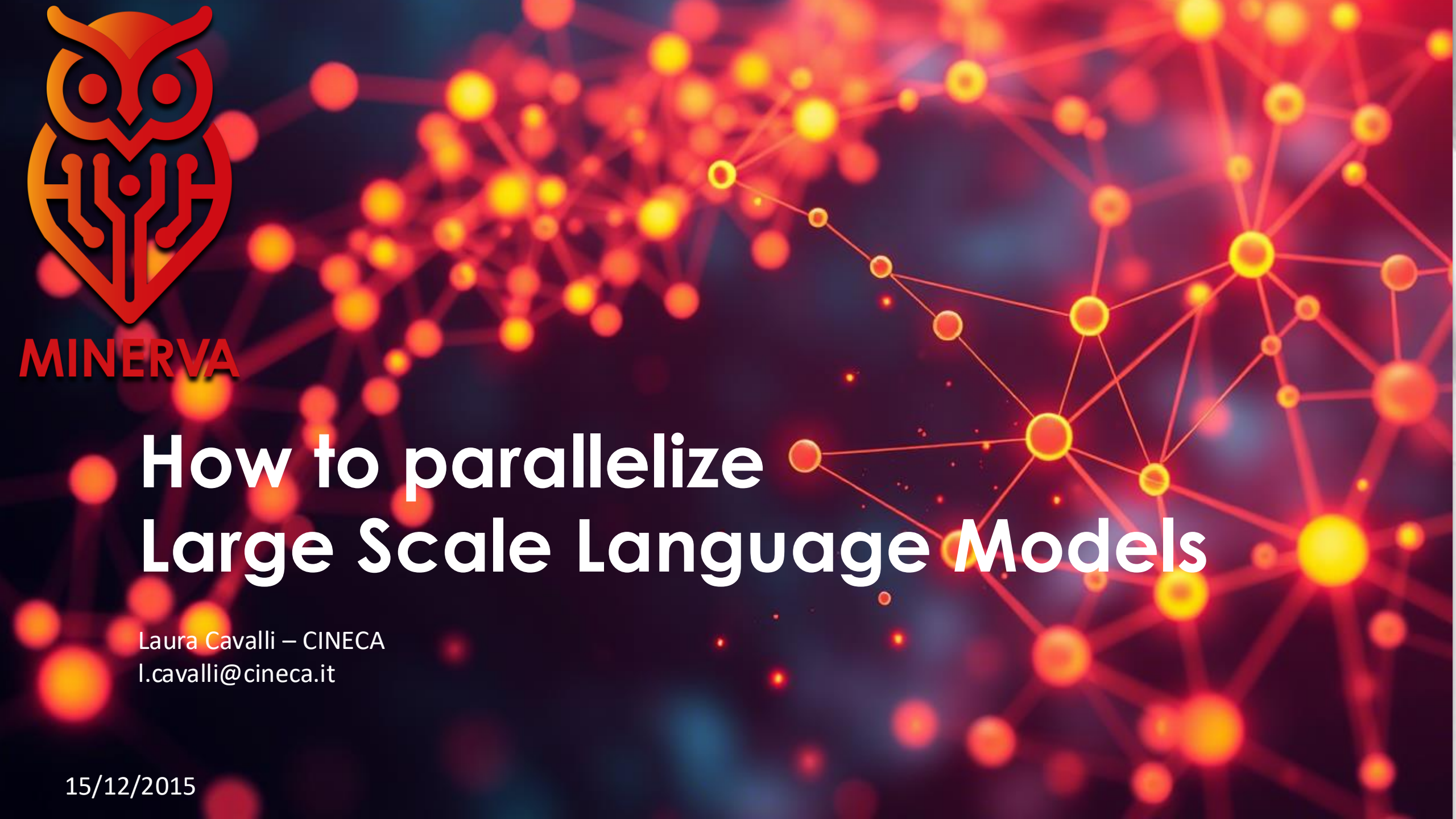
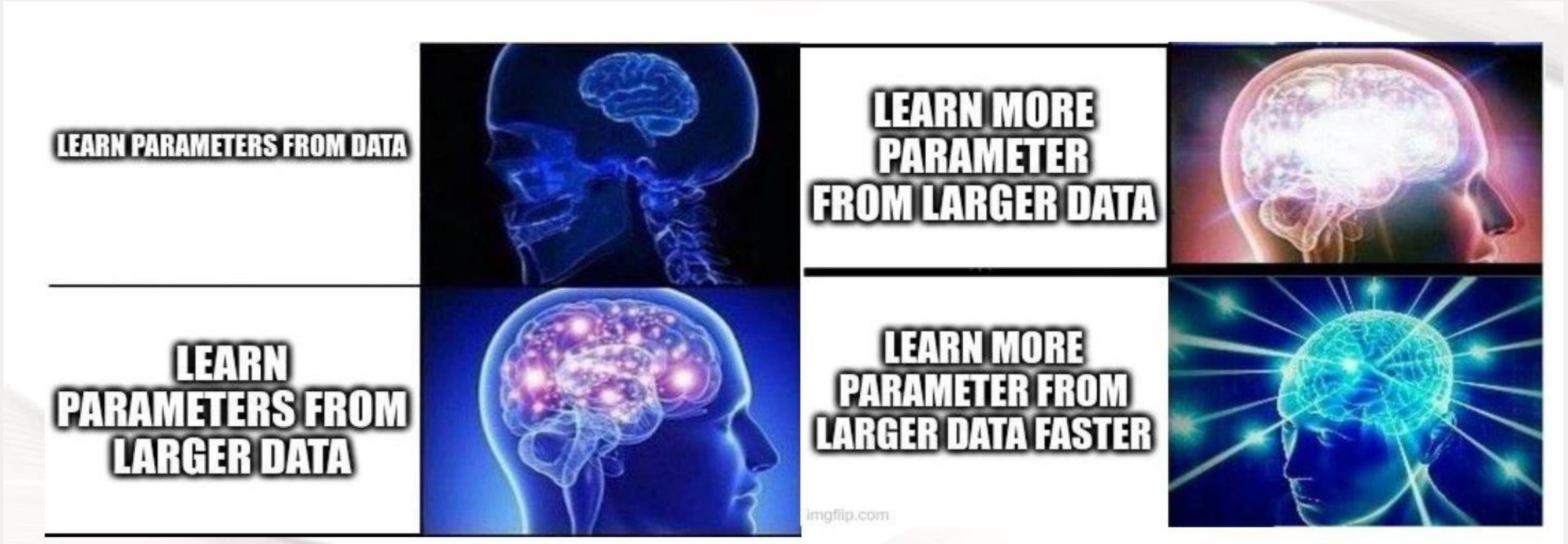


How to parallelize Large Scale Language Models

Laura Cavalli – CINECA
l.cavalli@cineca.it

15/12/2015

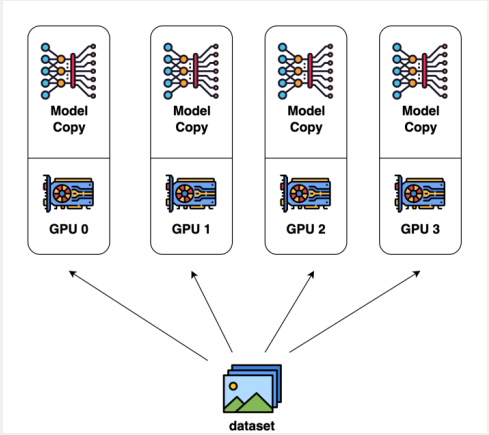
Training AI models evolution during time



DRAWBACKS + WHY PARALLEL TOOLS IN AI?

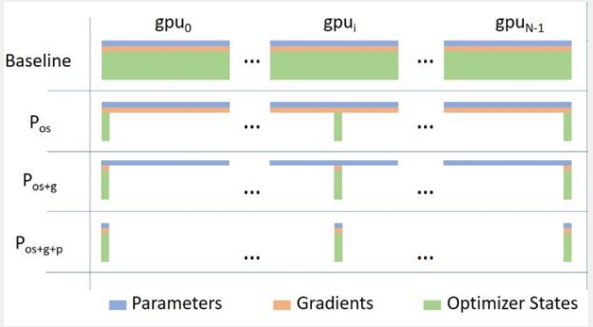
DIFFERENT TYPES OF PARALLELISM

Data Parallelism

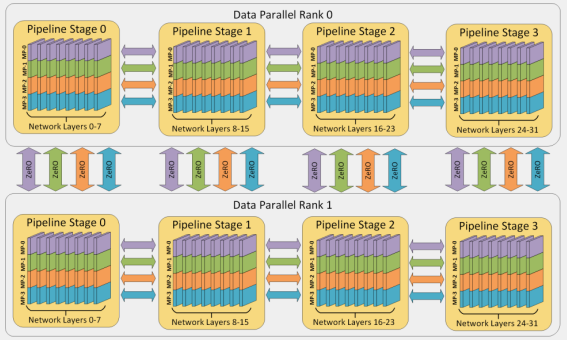


https://colossalai.org/docs/concepts/paradigms_of_parallelism/

Model Sharding



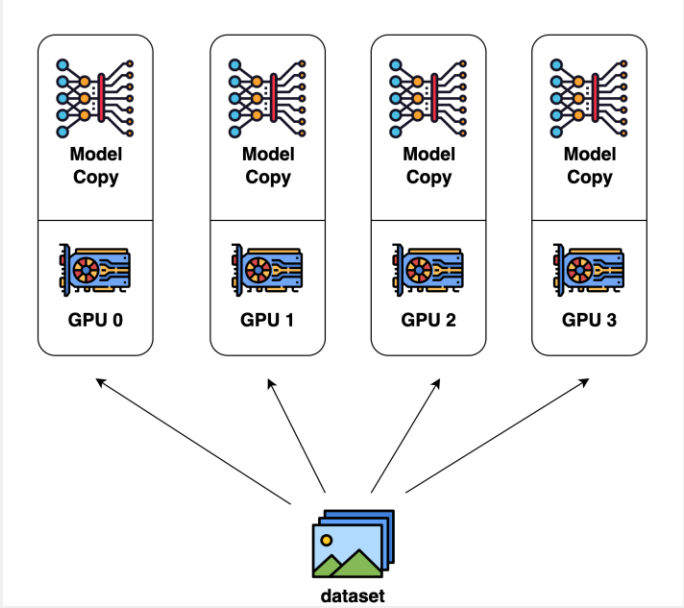
Model Parallelism



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

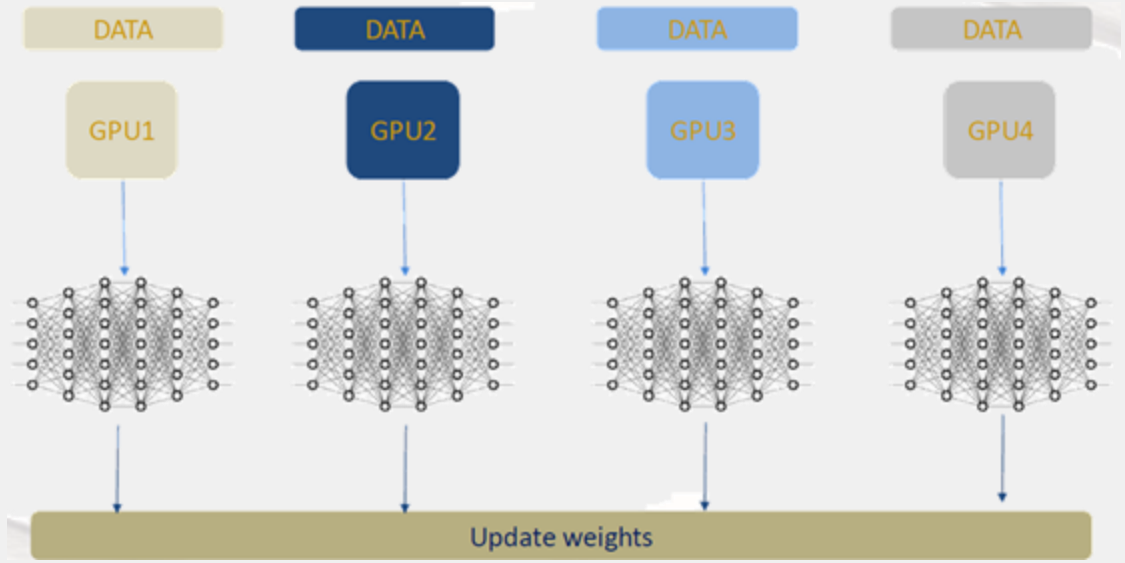
DATA PARALLELISM

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.

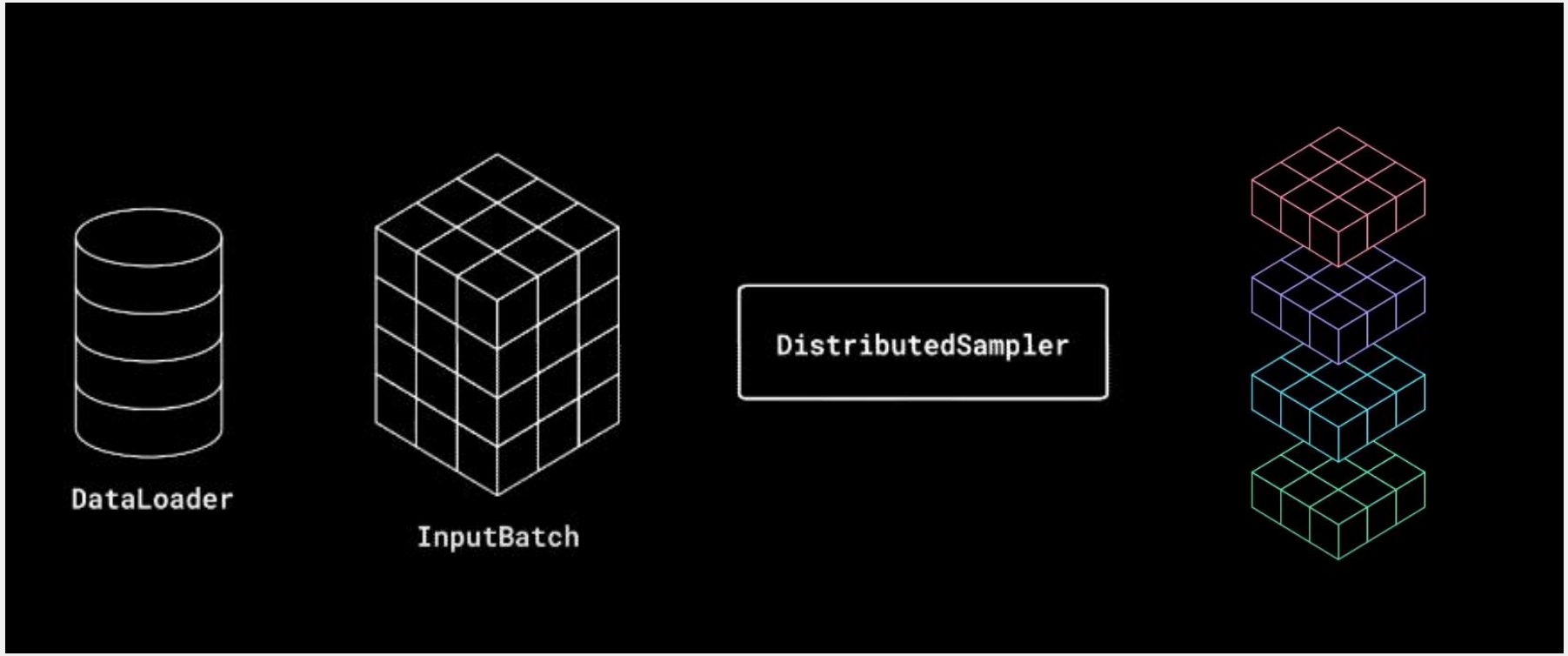


Data Parallelism – How it works

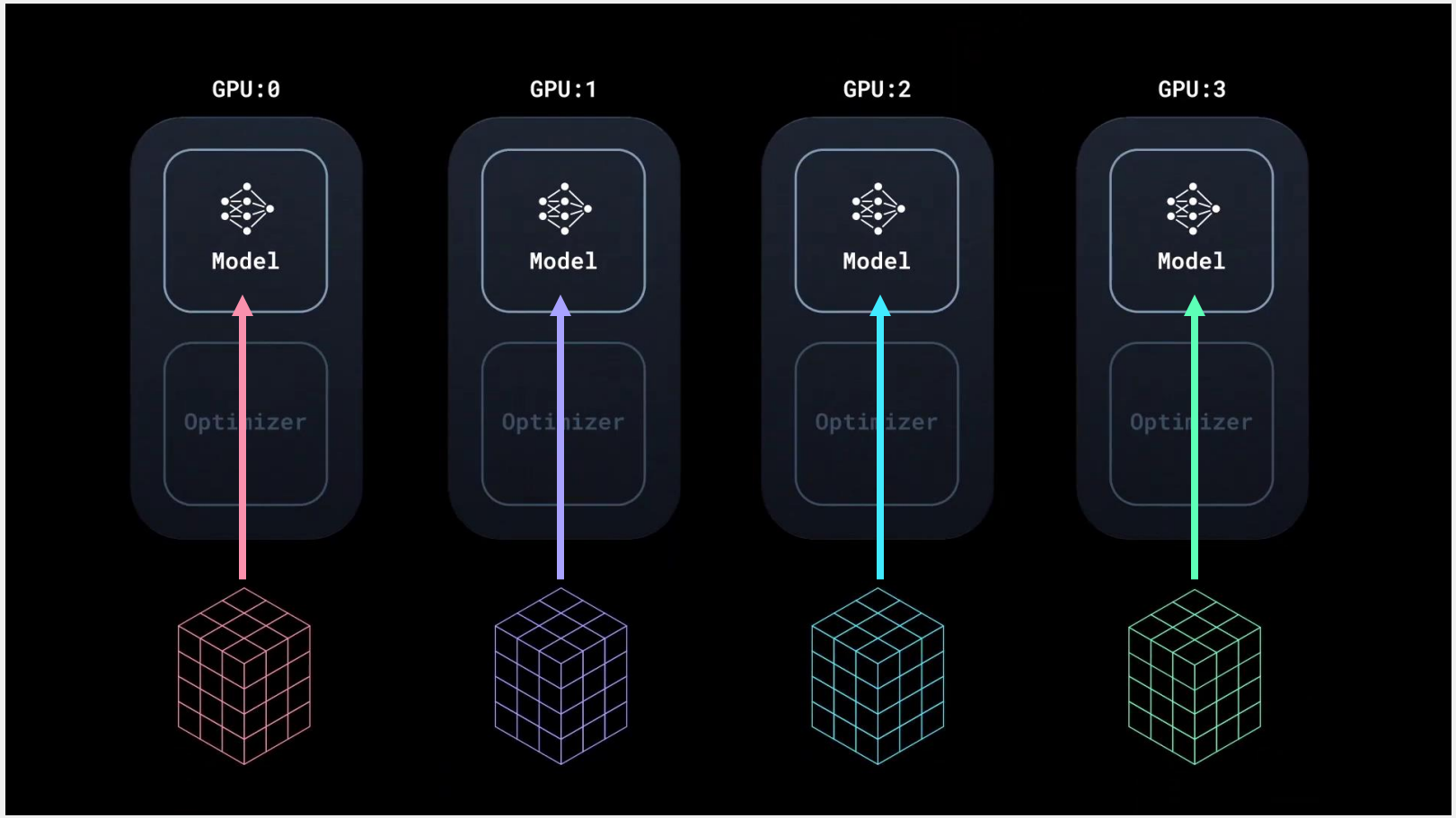
1. Each device holds a full copy of the model
2. Data are split on multiple GPUs (single or multi-node)
3. After some backpropagation steps, the results are are synchronized to avoid convergence issues



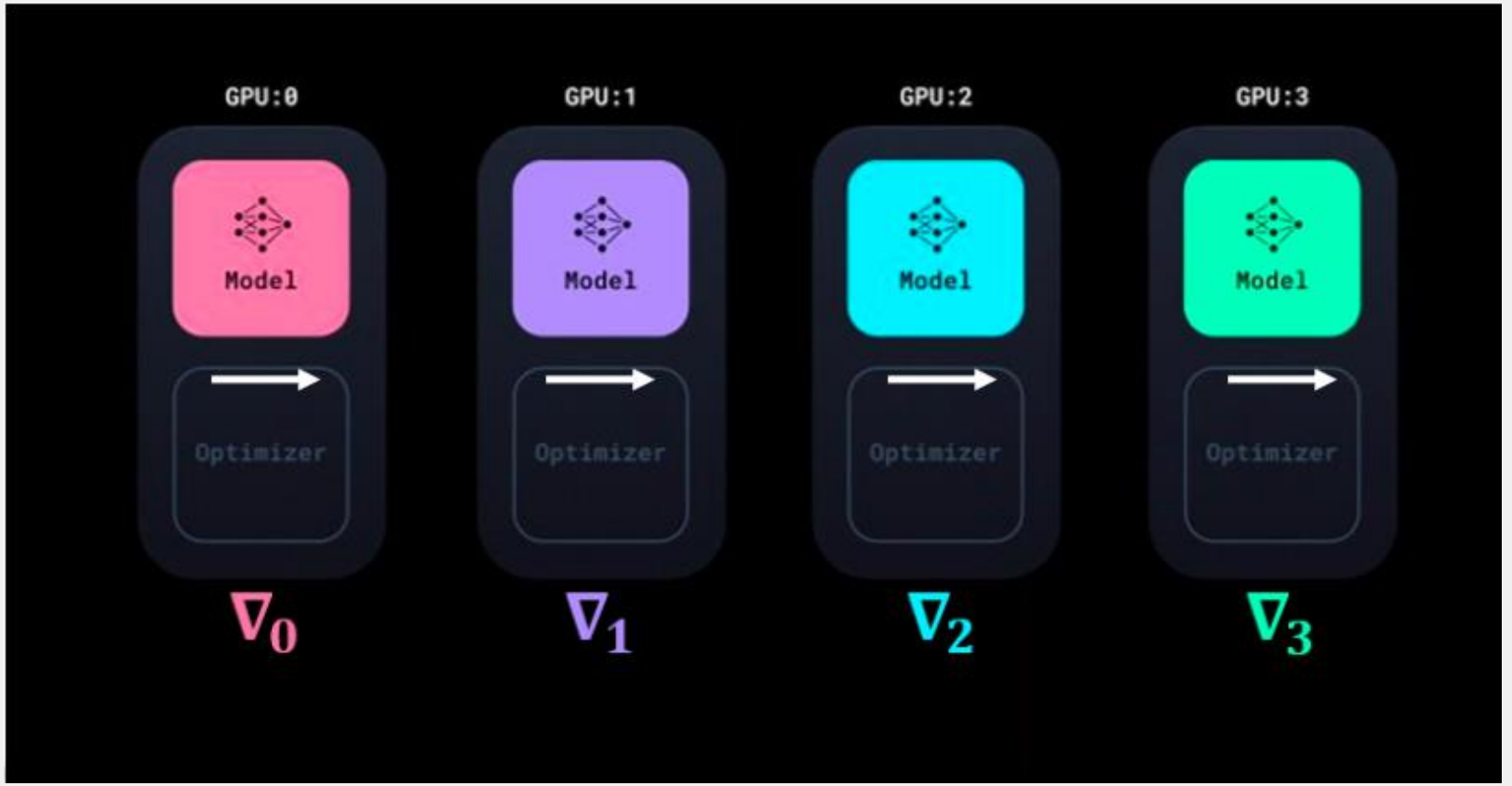
PYTORCH DDP (Distributed Data Parallel)



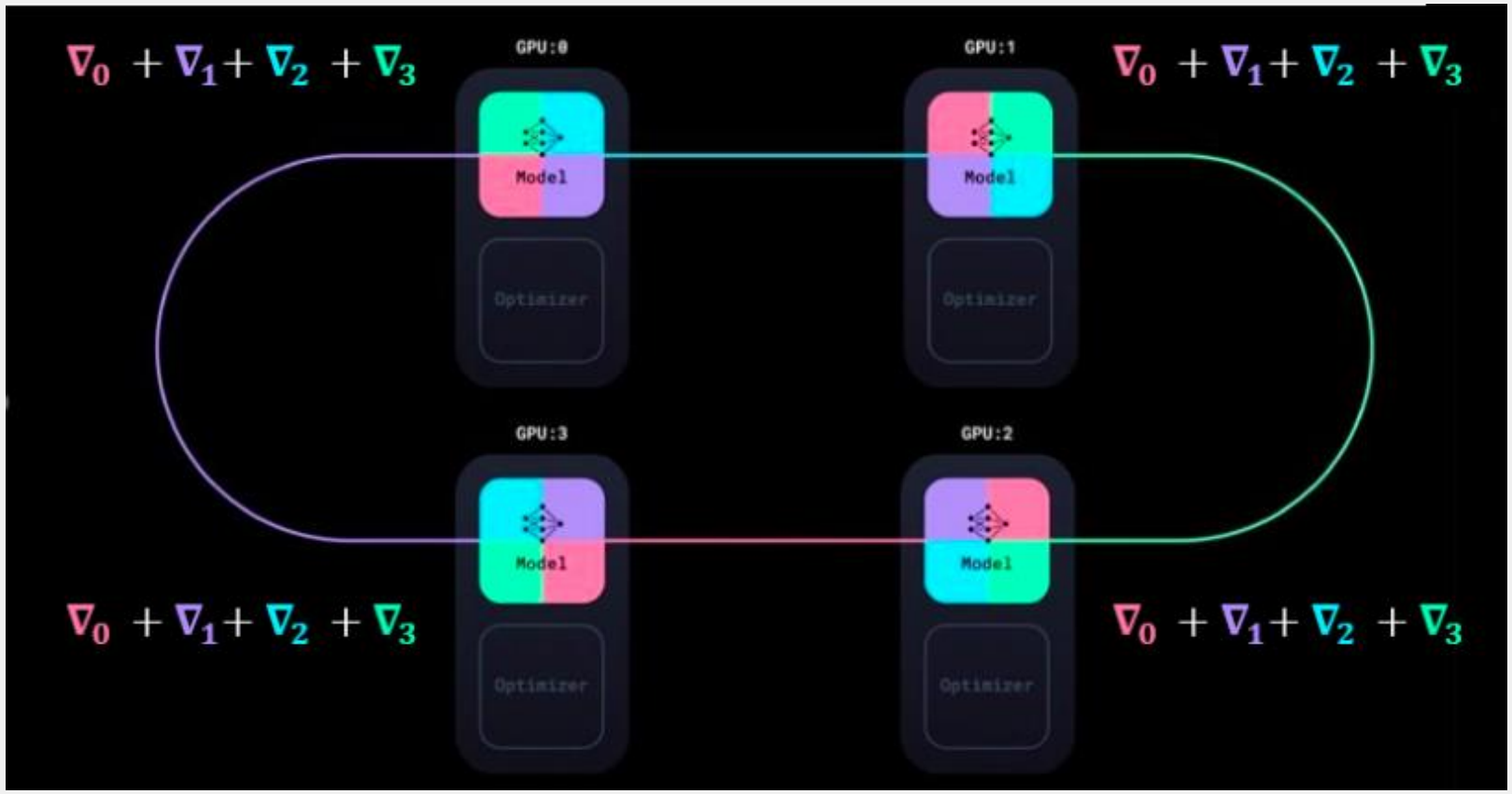
PYTORCH DDP (Distributed Data Parallel)



PYTORCH DDP (Distributed Data Parallel)



PYTORCH DDP (Distributed Data Parallel)



Ring ALL-REDUCE

synchronize gradients across model replicas

PYTORCH

How to implement it with PYTORCH DDP

```
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.utils.data.distributed import DistributedSampler

# Initialize the process group
rank, local_rank, world_size, local_size, num_workers = get_resources()
dist.init_process_group("nccl", rank=rank, world_size=world_size)

...

# Data Loading
train_sampler = DistributedSampler(dataset=train_dataset, rank = rank, num_replicas = world_size)
train_loader = torch.utils.data.DataLoader(train_dataset, sampler = train_sampler,
batch_size=batch_size)

...

# Model preparation for Training
device = torch.device("cuda:{}".format(local_rank))
torch.cuda.set_device(local_rank)

model = model.to(device)
model = DDP(model, device_ids=[local_rank], output_device=local_rank)
```

How to implement it with PYTORCH DDP

How to launch PyTorch DDP code

```
torchrun --nnodes=2 \
         --nproc_per_node=8 \
         --rdzv_id=100 \
         --rdzv_backend=c10d \
         --rdzv_endpoint=$MASTER_ADDR:29400\
         my_ddp_training.py
```

Here torchrun will launch 8 processes and invoke my_ddp_training.py on each process on the node it is launched on

bonus : SEQUENCE PARALLELISM

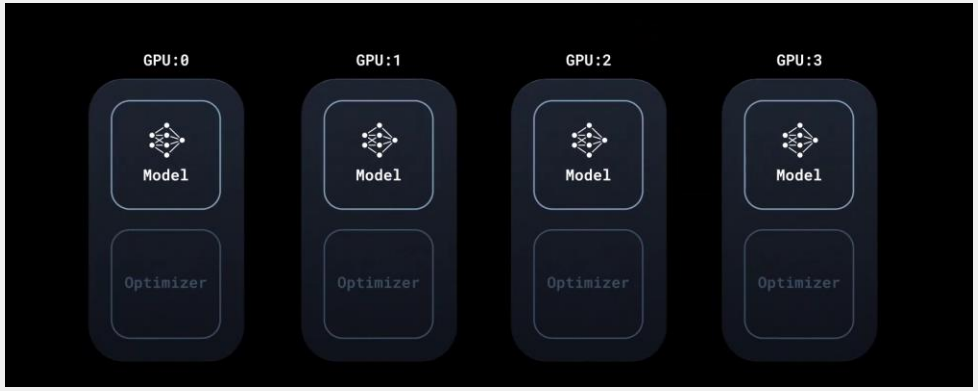
Sequence Parallelism (SP) is a distributed training technique used to **split long input sequences across multiple GPUs**, when the sequence length is too long for a single GPU's memory.

For example, if we have 4 GPUs, instead of putting all 4096 tokens on one single GPU, SP splits the sequence into 1024 tokens per GPU.

Can be seen as a sort of data parallelism specific for LLMs use case.

Main drawback of DATA PARALLELISM

With DDP each device has
a copy of the entire model



when the model exceed the single
device memory we need something
more sophisticated

LLM memory footprint determined by:

MODEL STATES

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

RESIDUAL STATES

- Activations
- Temporary buffers
- Unusable fragmented memory

+ precision, batch size,...

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

LLM MEMORY FOOTPRINT EXTIMATIONS

Consider Llama 7B :

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!

NAIVE SOLUTIONS to Out of Memory problem

1. Lower the precision :

   double (64 bit) → single (32bit) → half (16bit)

2. Quantization :

   int8 (Integer Quantization)
   4-bit Quantization (QLoRA, …)

BUT it impacts the performance, loosing information during the training

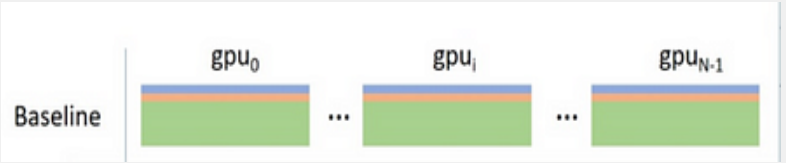
MODEL SHARDING

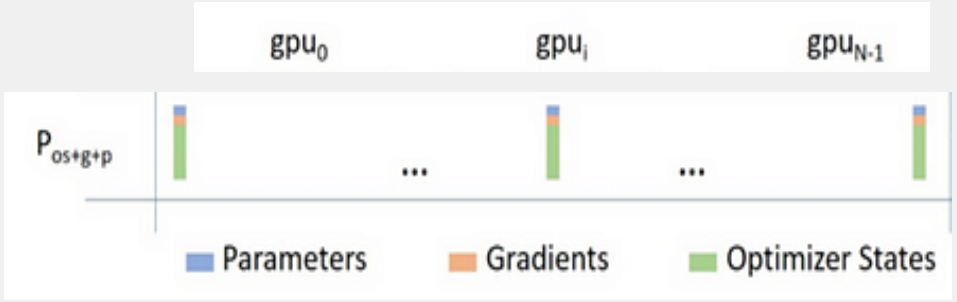
Model sharding is a technique for distributed training that **divides a model's parameters, gradients, and optimizer states across multiple GPUs** or nodes. This approach is particularly beneficial for training large models that cannot fit entirely into the memory of a single GPU.

FSDP and ZeRO Stage 3

DDP training :

- each worker (GPU) owns a replica of the entire model.
- it uses all-reduce to sum up only the gradients over different workers, (but the model weights and optimizer states are replicated across all workers)

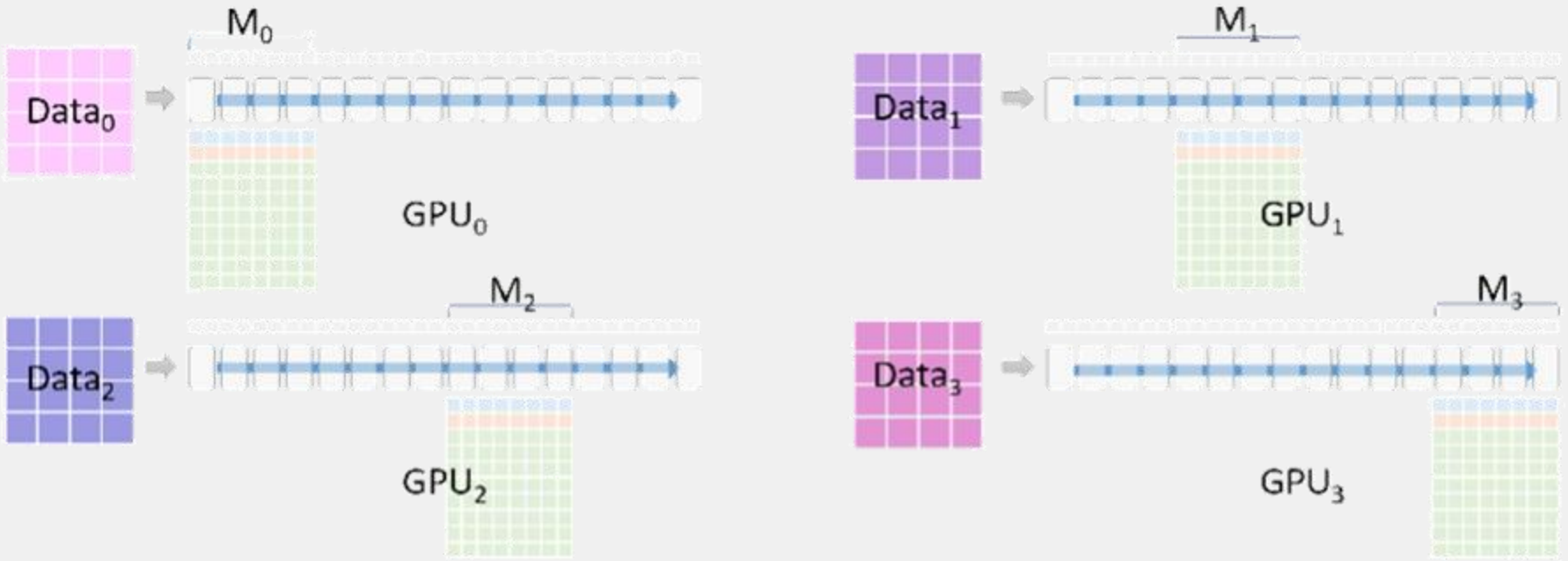




Model Sharding, across DDP ranks, shard either:

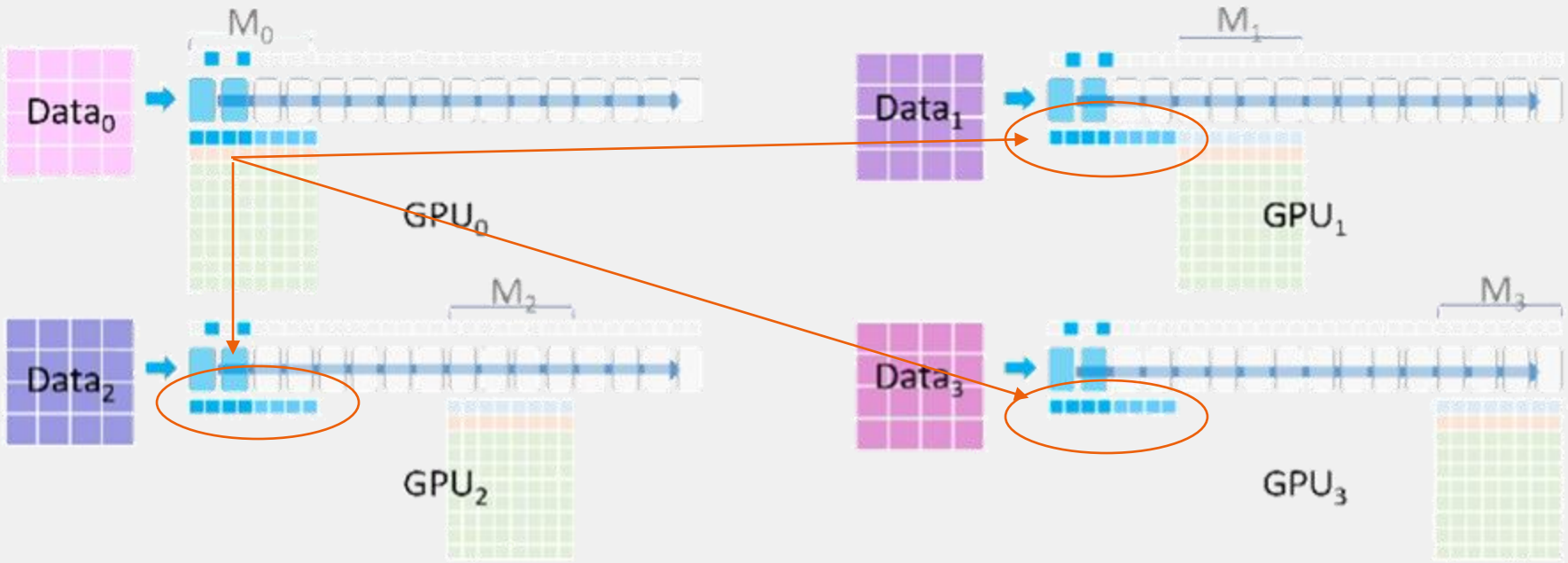
- model parameters,
- optimizer states
- gradients

FSDP and ZeRO Stage 3

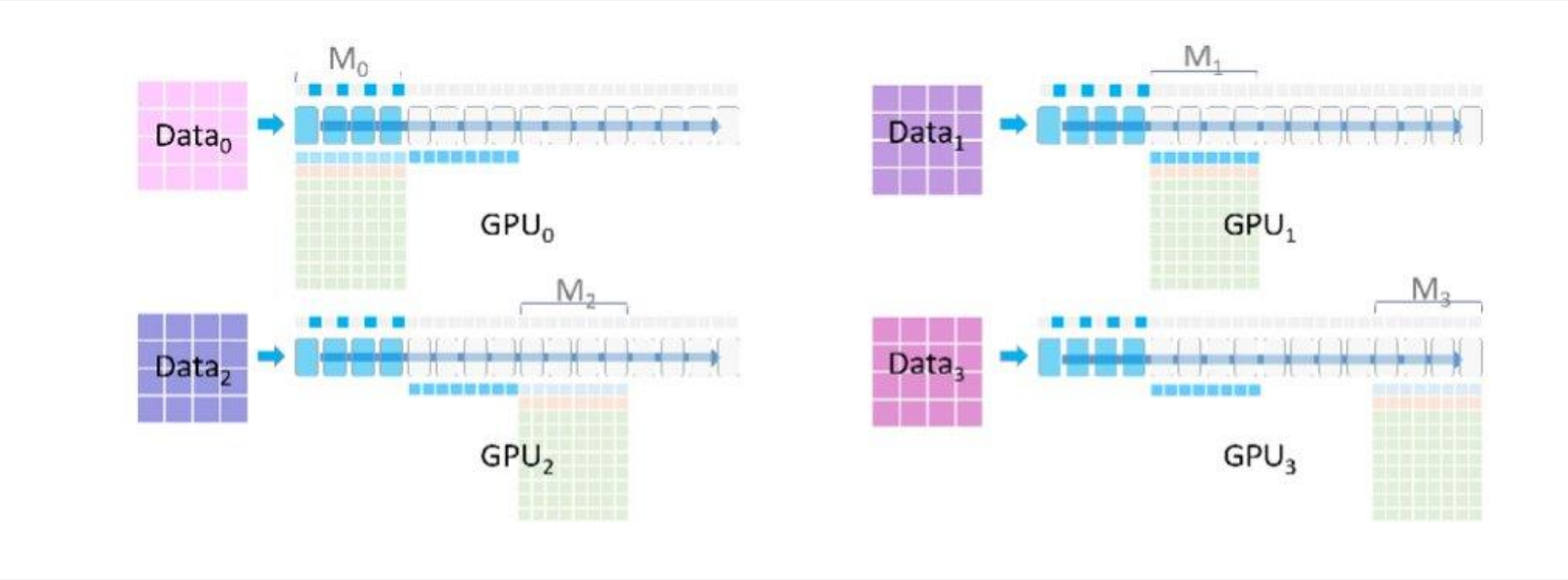


FSDP and ZeRO Stage 3

All - Gather



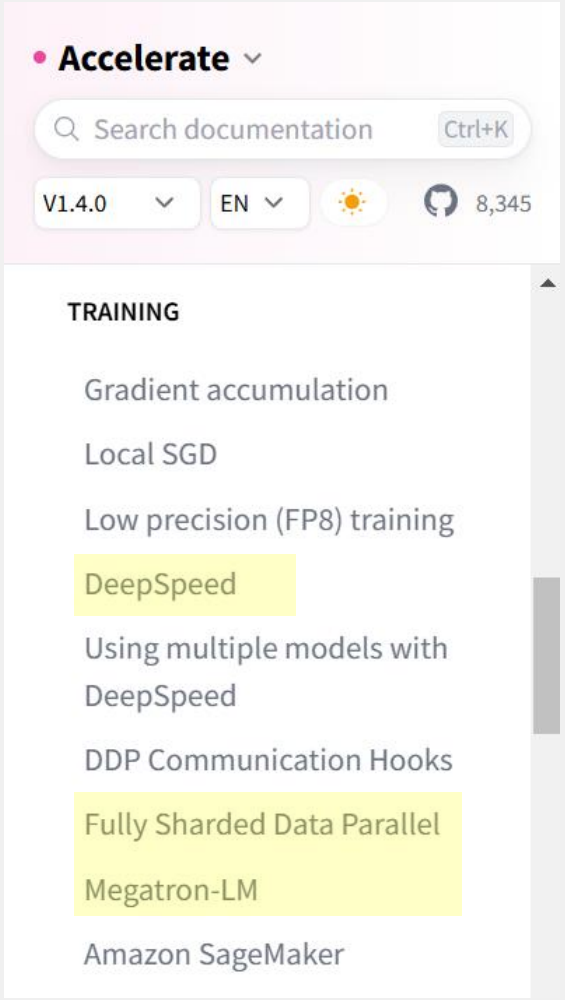
FSDP and ZeRO Stage 3



TOOLS FOR MODEL SHARDING

1. Accelerate
2. DeepSpeed
3. Olmo-core

ACCELERATE by Hugging Face



Accelerate is an HF library that enables running the same PyTorch code across any distributed configuration by adding just few lines of code.

“In short, training and inference at scale made simple, efficient and adaptable.”

https://huggingface.co/docs/accelerate/index

Examples of **config.yaml** files

```yaml
compute_environment: LOCAL_MACHINE
debug: false
distributed_type: FSDP
downcast_bf16: 'no'
enable_cpu_affinity: false
fsdp_config:
  fsdp_auto_wrap_policy: TRANSFORMER_BASED_WRAP
  fsdp_backward_prefetch: BACKWARD_PRE
  fsdp_cpu_ram_efficient_loading: false
  fsdp_forward_prefetch: true
  fsdp_offload_params: false
  fsdp_sharding_strategy: FULL_SHARD
  fsdp_state_dict_type: SHARDED_STATE_DICT
  fsdp_sync_module_states: false
  fsdp_use_orig_params: true
machine_rank: 0
main_training_function: main
mixed_precision: bf16
num_machines: 1
num_processes: 4
rdzv_backend: static
same_network: true
tpu_env: []
tpu_use_cluster: false
tpu_use_sudo: false
use_cpu: false
```

```yaml
compute_environment: LOCAL_MACHINE
debug: false
distributed_type: DEEPSPEED
downcast_bf16: 'no'
deepspeed_config:
  gradient_clipping: 0.3
  gradient_accumulation_steps: 1
  offload_optimizer_device: none
  offload_param_device: none
  zero3_init_flag: true
  zero3_save_16bit_model: true
  zero_stage: 3
machine_rank: 0
main_training_function: main
mixed_precision: bf16
num_machines: 2 # 1
num_processes: 8 # 4
rdzv_backend: static
same_network: true
tpu_env: []
tpu_use_cluster: false
tpu_use_sudo: false
use_cpu: false
```

How to launch it

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 \
    --multi_gpu \
    --num_machines $NNODES \
    --num_processes $WORLD_SIZE \
    --main_process_ip "$MASTER_ADDR" \
    --main_process_port $MASTER_PORT \
    --machine_rank $SLURM_PROCID \
    --rdzv_backend c10d \
    --config_file config_accelerate.yaml
    my_script.py <py_args>
```

Results

Fine-Tuning of Llama3.1-7B

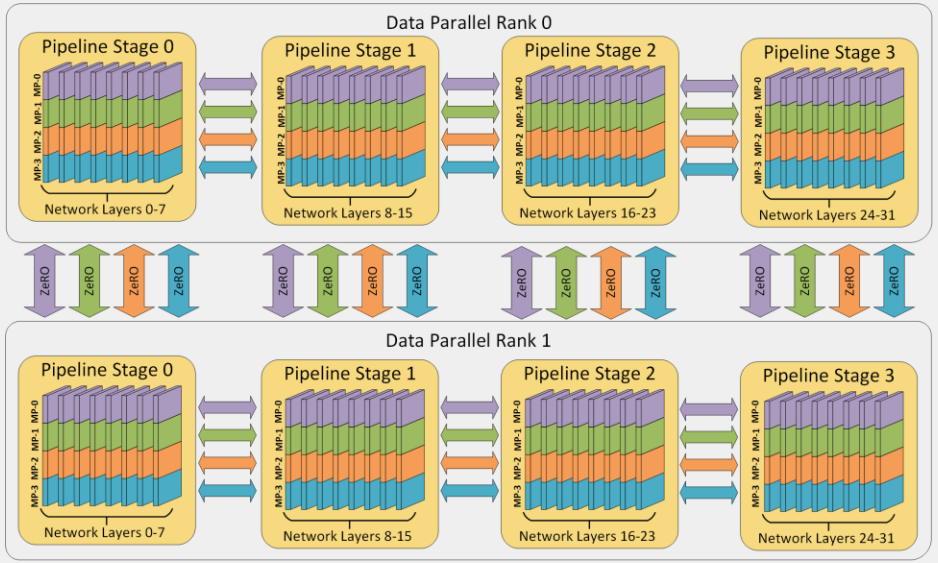
NUM NODES	NUM GPUS	TRAINING TIME	SPEED UP
1	1	3884 s	1
1	2	1925 s	2
1	4	954 s	4
2	8	602 s	6.45

Fine-Tuning of Llama2-70B

NUM NODES	NUM GPUS	TRAINING TIME	SPEED UP
1	3	OoM	_
1	4	246,2 min	1
2	8	136 min	1.8

MODEL PARALLELISM

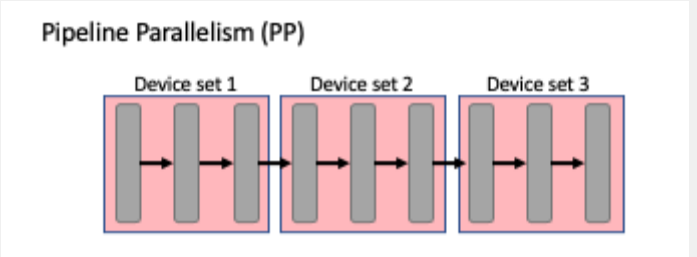
Model Parallelism is a technique used to split a neural network across multiple GPUs or nodes where **different GPUs handle different parts of the model.**



Model Parallelism Techniques

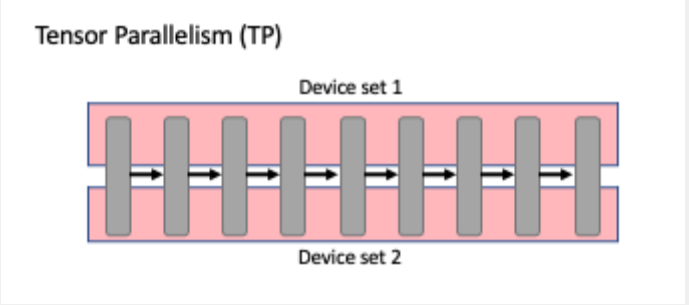
Pipeline Parallelism - Vertical

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)

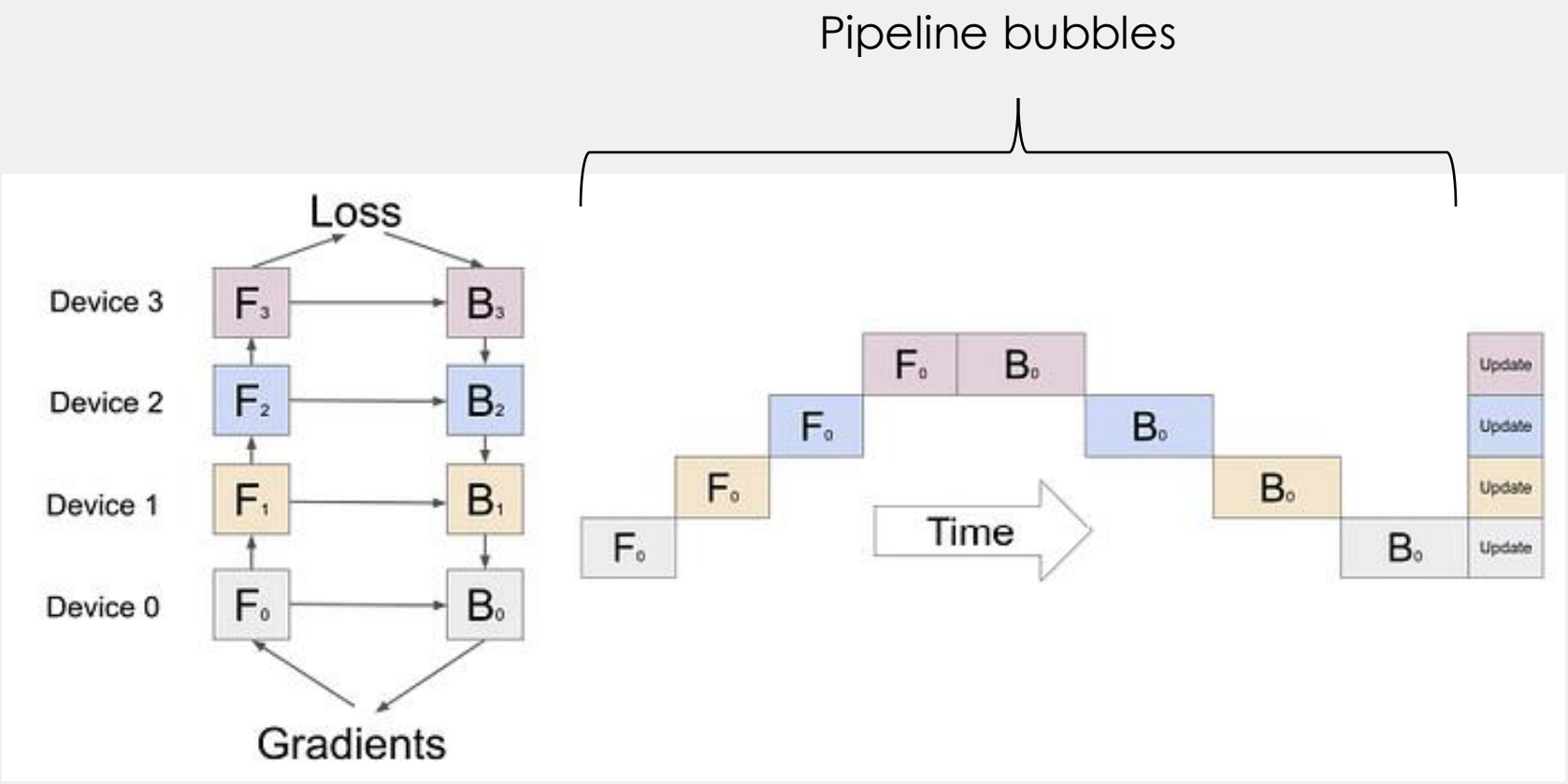


Tensor Parallelism - Horizontal

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.



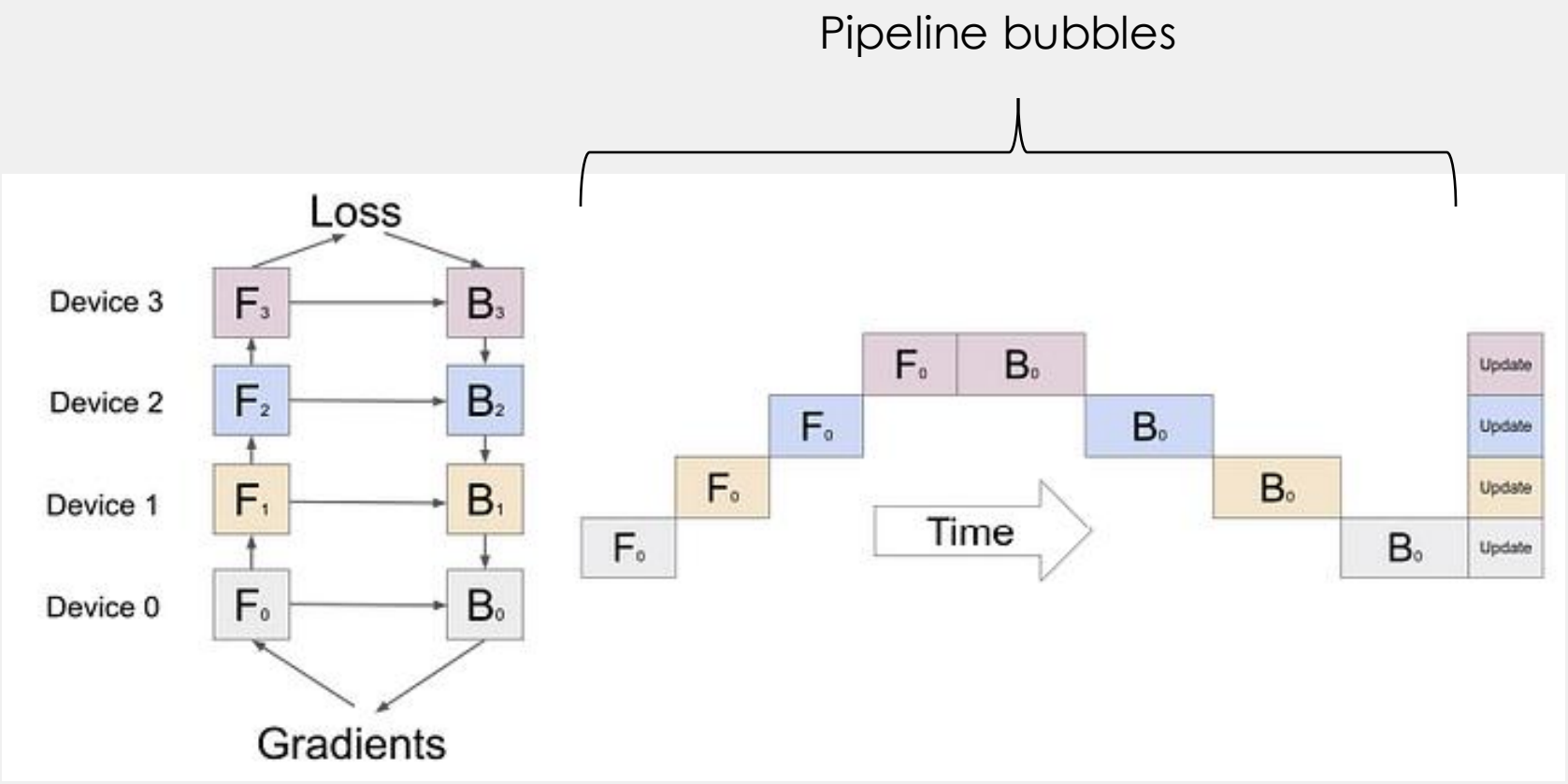
Naïve PIPELINE PARALLELISM



Only ONE GPU running at time!!

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

Naïve PIPELINE PARALLELISM

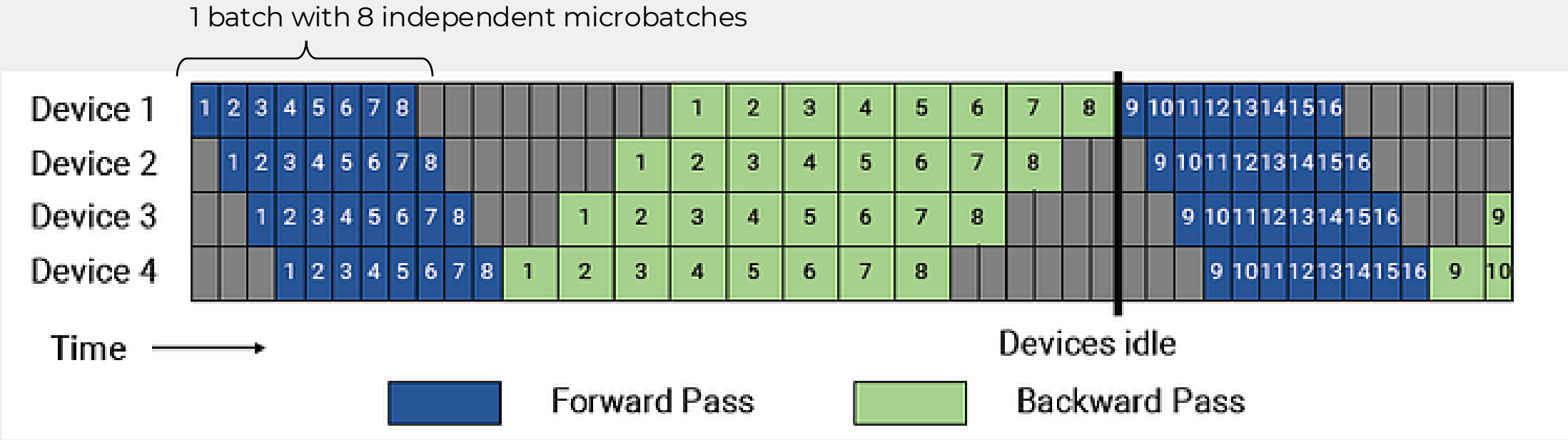


Only ONE GPU running at time!!

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

PIPELINE PARALLELISM

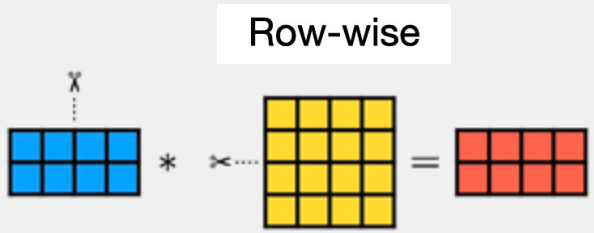
They introduce the idea of microbatches

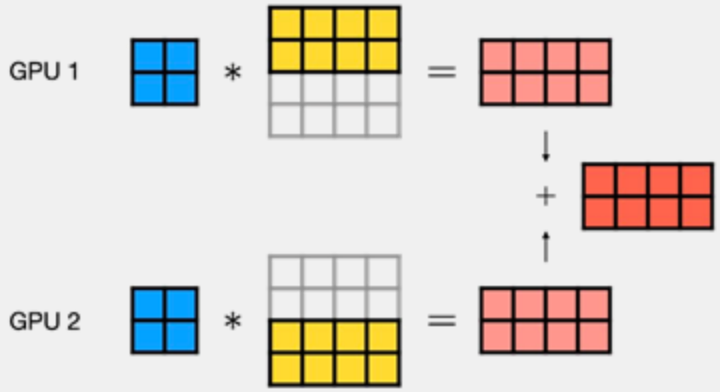


... and there is much more literature on how to reduce pipeline bubbles

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

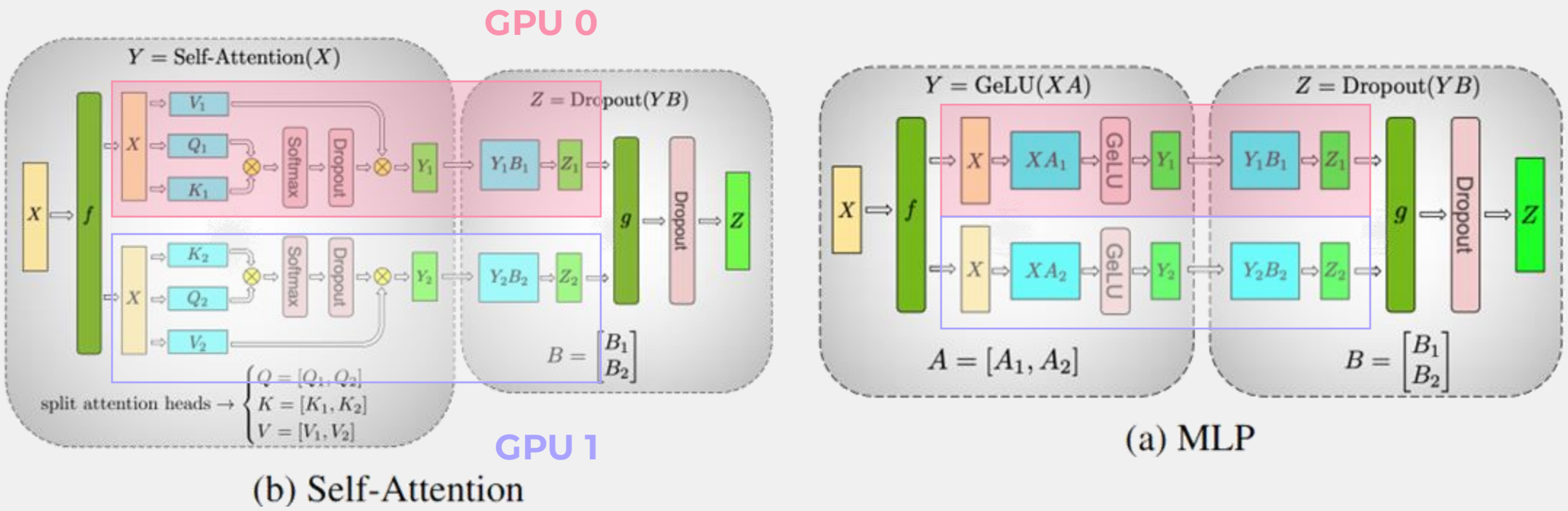
Idea behind TENSOR PARALLELISM





TENSOR PARALLELISM

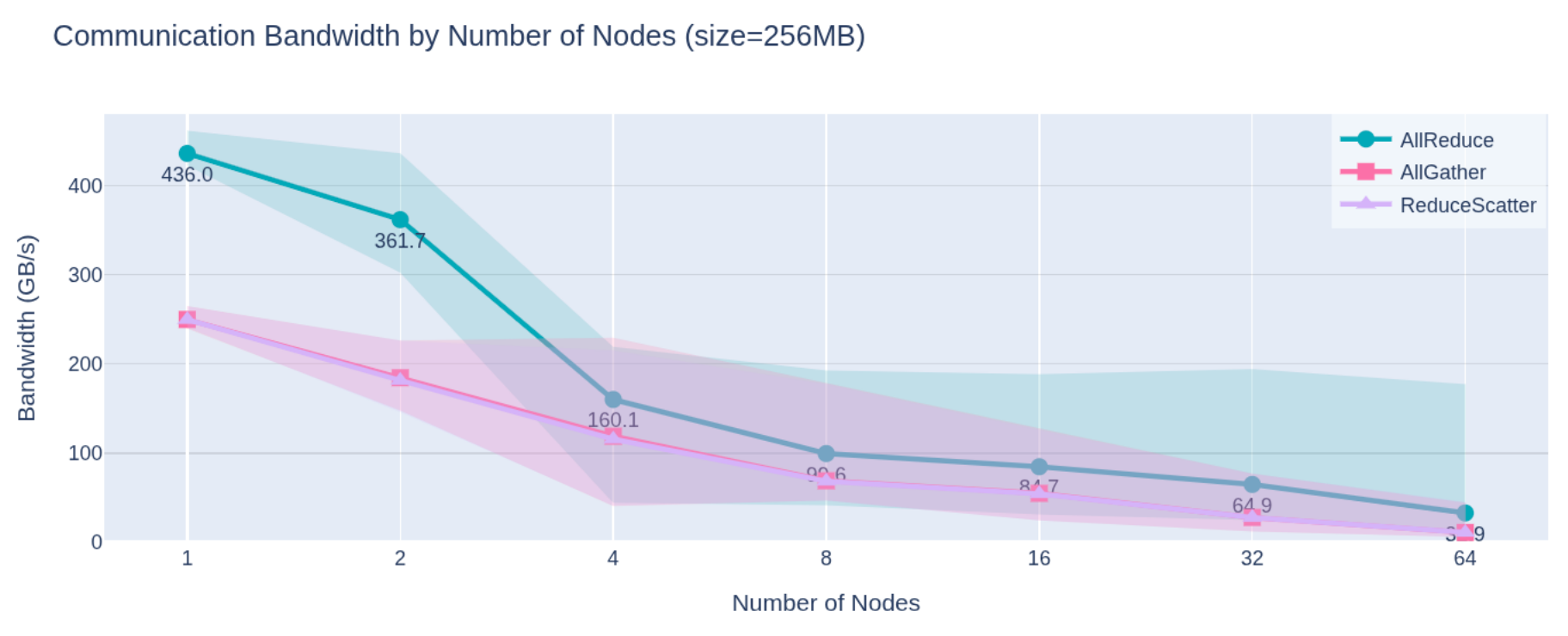
Transformer block



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

Drawback of TP

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.



TENSOR or PIPELINE PARALLELISM

https://arxiv.org/pdf/2104.04473

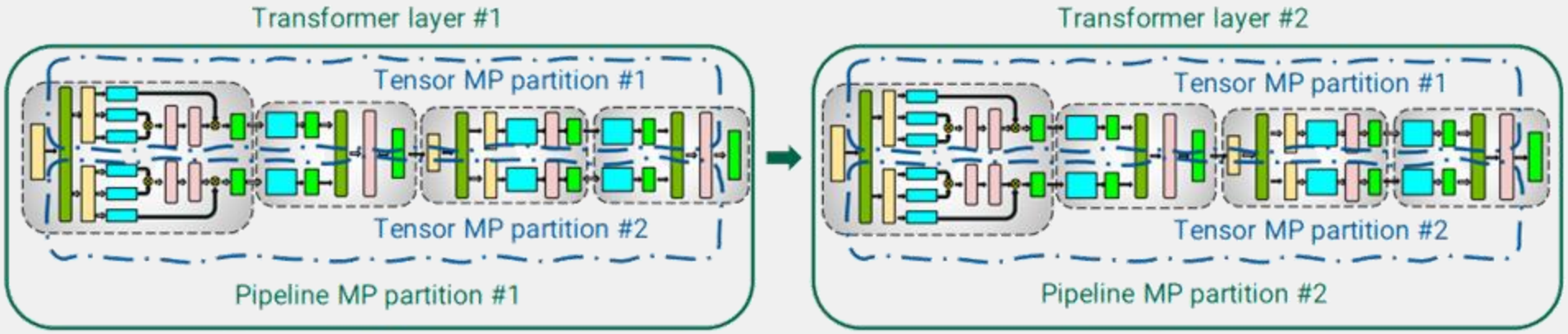


Figure 2: Combination of tensor and pipeline model parallelism (MP) used in this work for transformer-based models.

Communications :

TP :
What : All-Gather (forward pass for activations) + All-Reduce (backward pass for gradients).
When : Every layers
TP <= num_gpus_per_node!!

PP :
What : Send/Recv
When : Once per forward/backward pass per pipeline stage per mini-batch.

Model Parallelism Libraries

1. Megatron-LM / Nemo (NVIDIA) — https://github.com/NVIDIA/Megatron-LM https://github.com/NVIDIA/NeMo
2. DeepSpeed — https://huggingface.co/docs/transformers/v4.19.2/en/parallelism
3. ColossalAI — https://colossalai.org/
4. Nanotron — https://github.com/huggingface/nanotron
5. vLLM — https://docs.vllm.ai/en/latest/

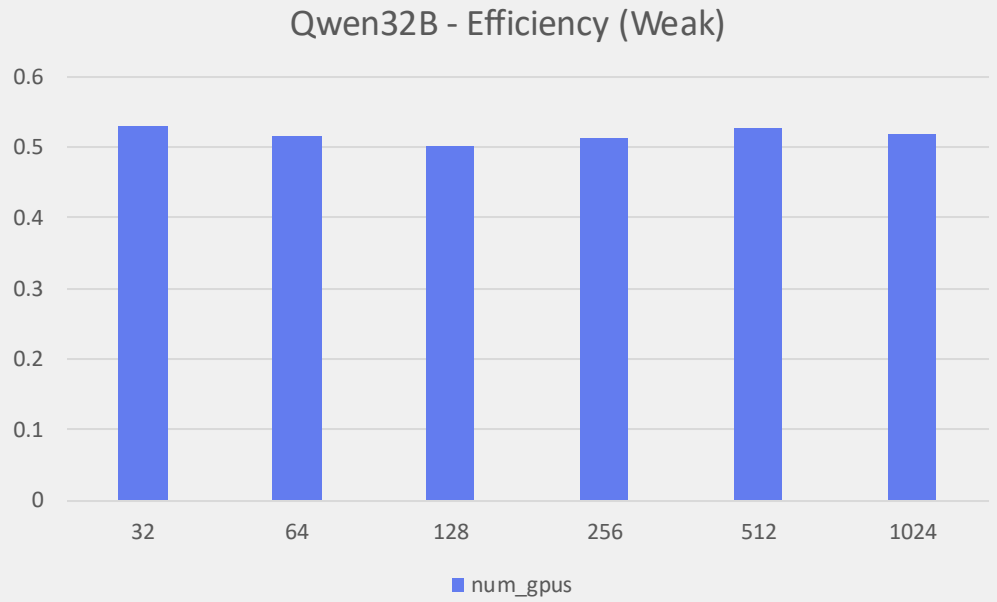
Some Results with Megatron-LM

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

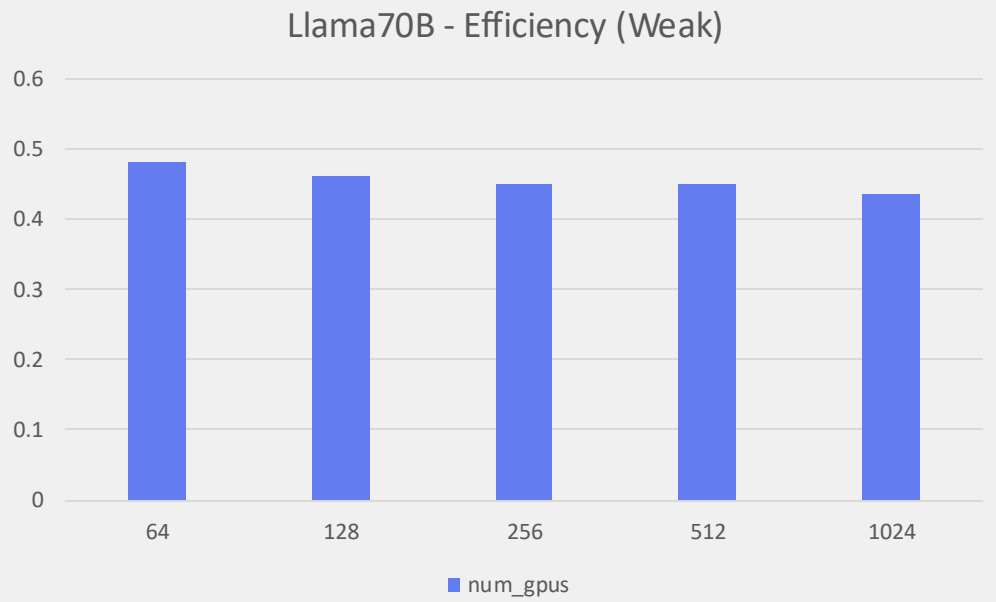
Some Results with Megatron-LM

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

Some Results with Megatron-LM

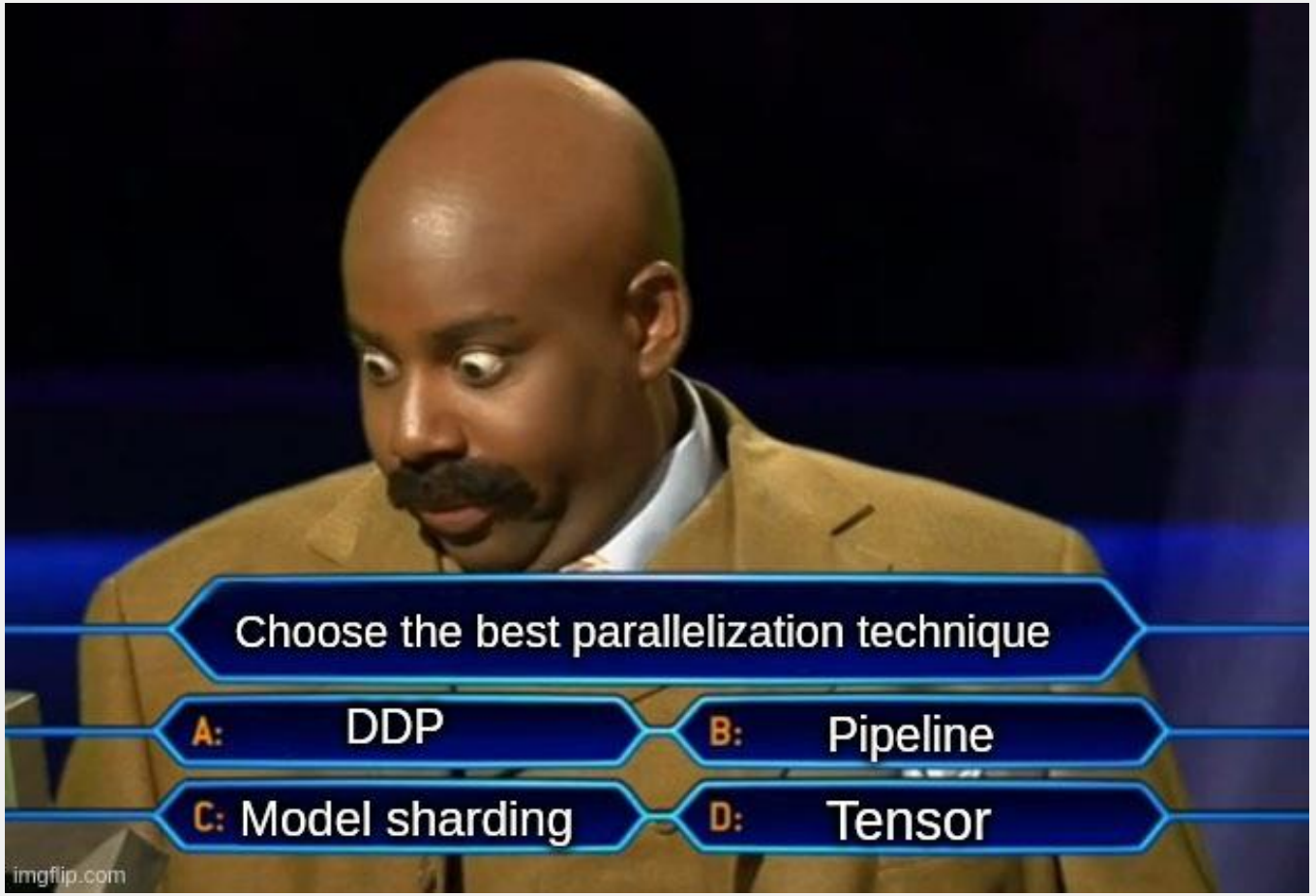


(TP=4, PP=8)



(TP=4, PP=16)

TAKE-HOME MESSAGE



Useful Resources

- **HF UltraScale Playbook :**
  https://huggingface.co/spaces/nanotron/ultrascale-playbook
- **Megatron-LM papers:**
  https://arxiv.org/pdf/2104.04473
- The **official websites/documentations** of Pytorch DDP, HF Accelerate, ecc …

Thank you

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.

