Aalto University

Scaling PyTorch Models: Single vs Multi-GPU Training and Techniques

Hossein Firooz

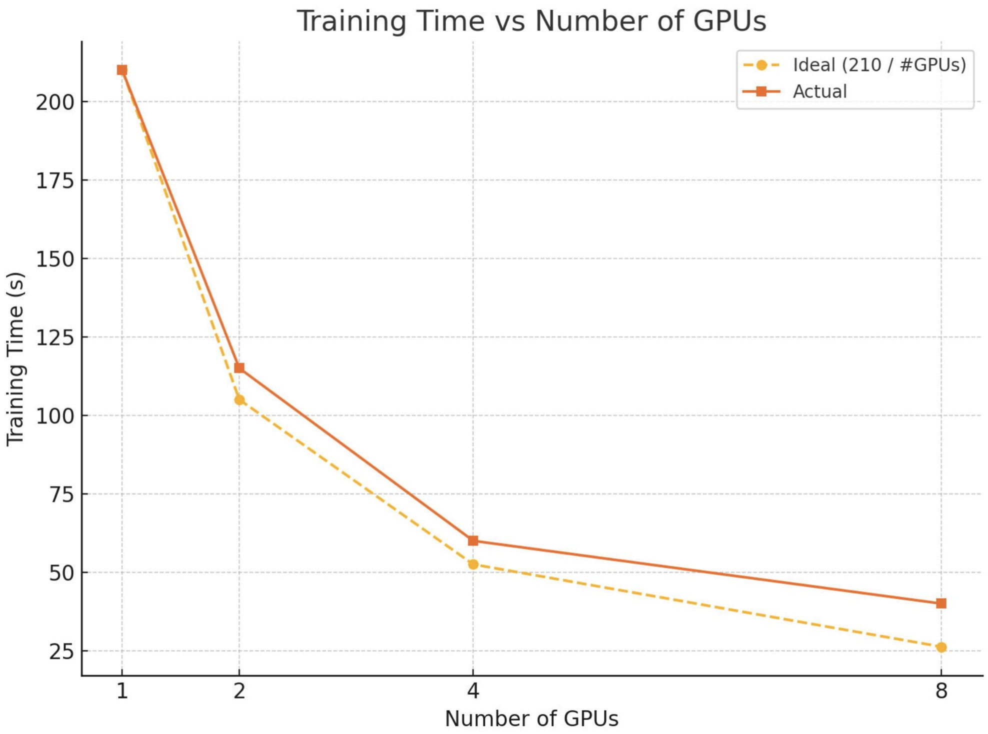
Single GPU vs Multi-GPU training

Aalto University

- Training ML models could be intense
  - Heavy computations
  - Large Model
- That’s why we might need use multiple GPUs to train
  - GPUs could be across multiple nodes
- Multi-GPU or Multi-Node training has overhead
  - Communication costs
  - Underutilization
  - Distribution of the data

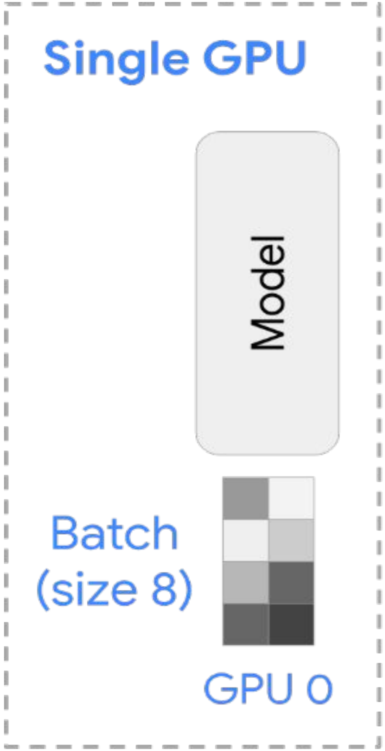
Multi-GPU performance

Aalto University



ResNet152 with CIFAR100 multi-gpu performance

Single-GPU Training



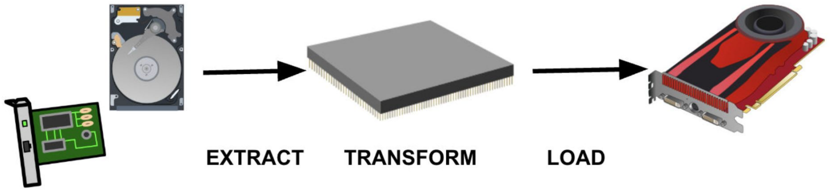
- Entire model & data on one GPU
- Pros: Simple, fast for small models
- Cons: Not scalable to large models/dataset

Most Common Bottleneck: DataLoader

Aalto University

- Most common bottleneck in workflows
- Causes the underutilization issue
- In Python process, the Global Interpreter Lock (GIL) prevents true parallelization across threads
- Moving data blocks computation
- `Nvidia-smi` and `rocm-smi` show high utilization for waiting kernels
- Solution: Use multiple workers (processes) in PyTorch `DataLoader`

```
train_loader = torch.utils.data.DataLoader(data, ..., num_workers=N)
```



DataLoader Best Practices

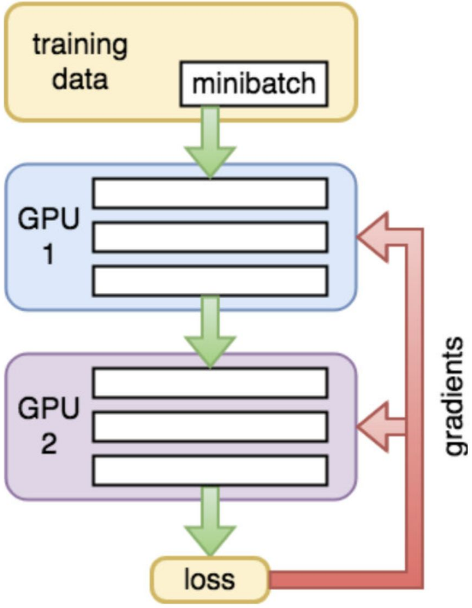
Aalto University

- PyTorch DataLoader uses single-process data loading by default.
  - Good for small datasets / Limited resources
  - Easier debugging
- Multi-process data loading
  - Each process will consume as much memory as the parent process
  - Total required memory = `num_workers * size of parent process`
  - Return CPU tensors only. Returning CUDA tensors could lead to deadlocks and mem corruption
  - Using `pin_memory=True` will increase the memory transfer speed
  - Using `non_blocking=True` will do asynchronous GPU copies.

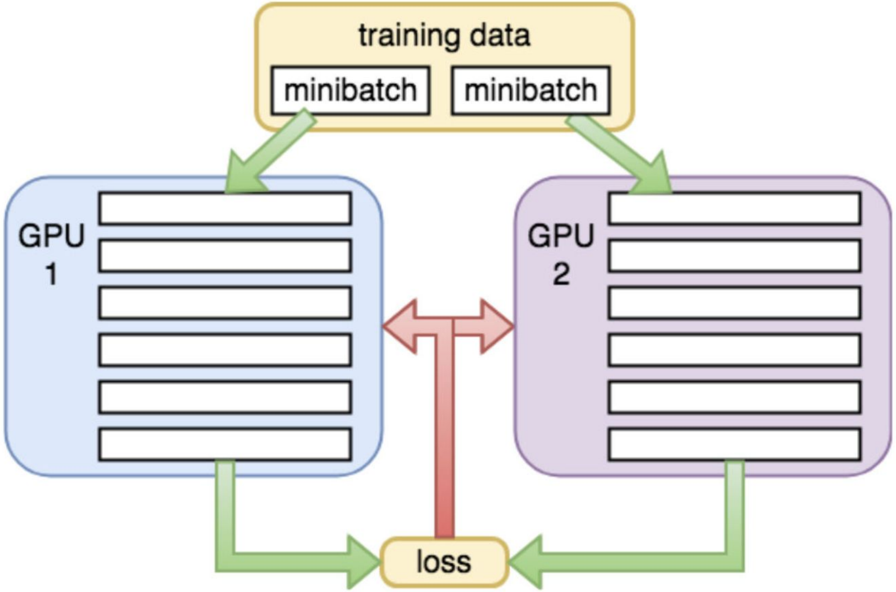
Multi-GPU Techniques

Aalto University

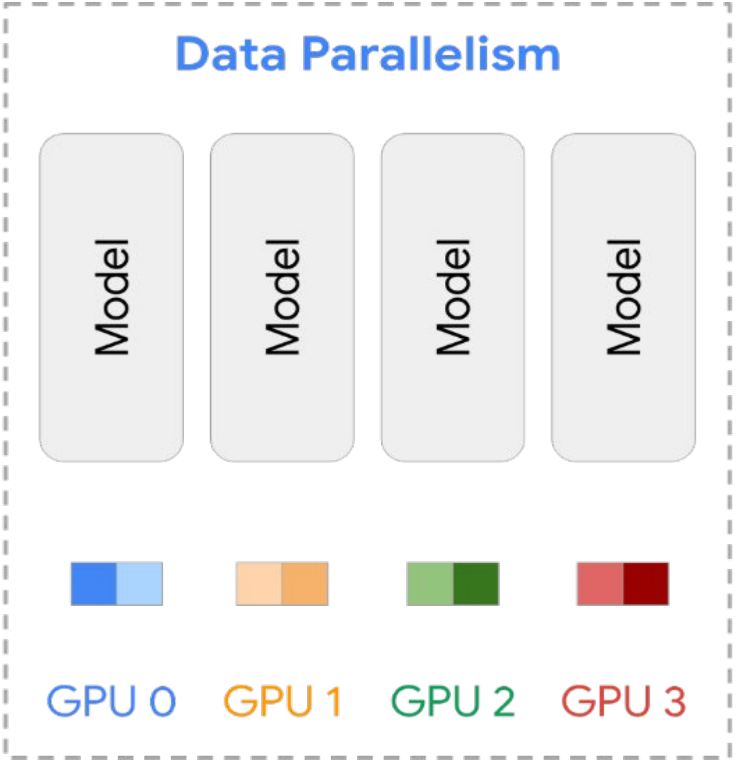
Model Parallelism



Data Parallelism



Data Parallelism



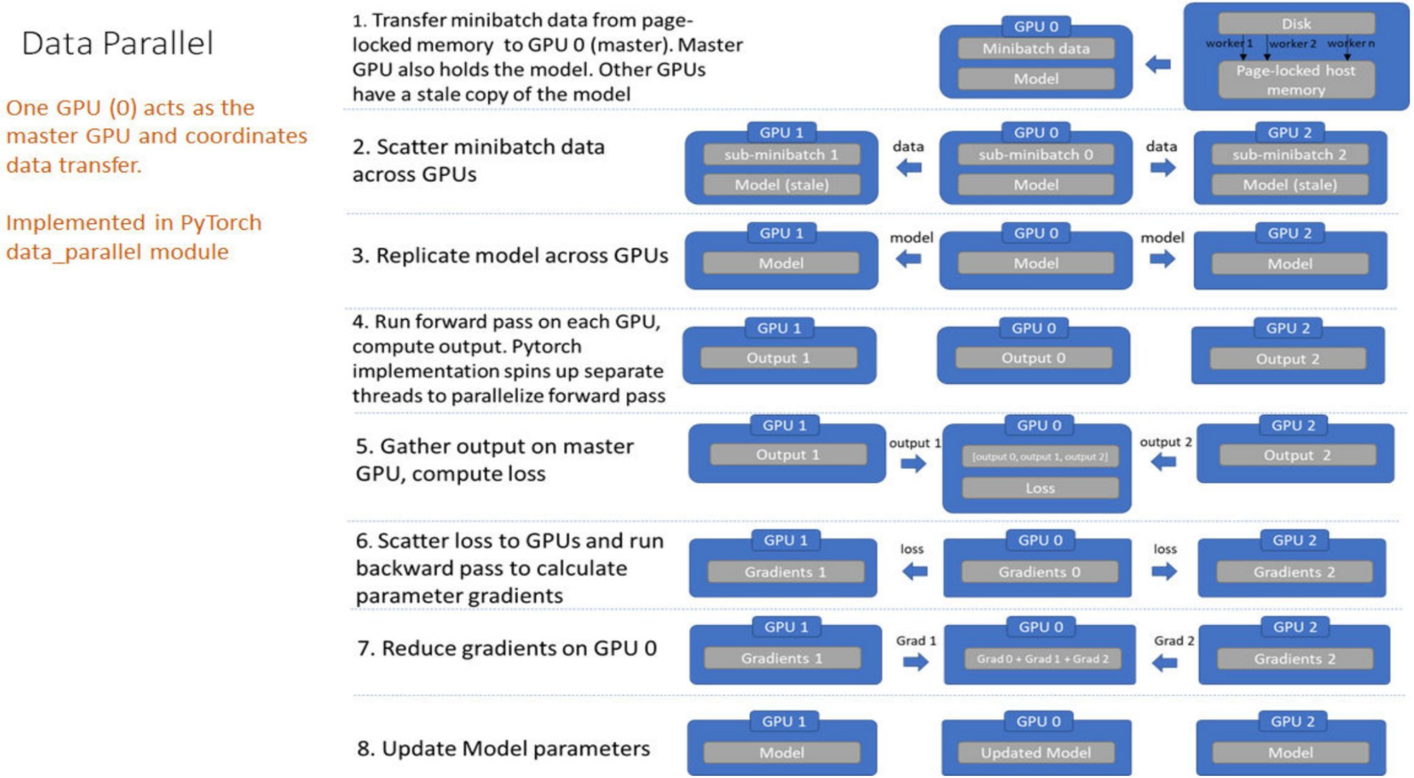
- Copy model to each GPU
- Split Data across GPUs
- Compute forward/backward
- Aggregate gradients

Overheads

Type	Description
Communication Overhead	High
Partial distribution	Possible
Underutilization	Possible

Naive PyTorch Data Parallelism (DP)

Aalto University



PyTorch Distributed Data Parallelism (DDP)

Aalto University

Distributed Data Parallel

No master GPUs

Implemented in PyTorch DistributedDataParallel module

1. Load data from disk into page-locked memory on the host. Use multiple worker processes to parallelize data load. Distributed minibatch sampler ensures that each process loads non-overlapping data

2. Transfer minibatch data from page-locked memory to each GPU concurrently. No data broadcast is needed. Each GPU has an identical copy of the model and no model broadcast is needed either

3. Run forward pass on each GPU, compute output

4. Compute loss, run backward pass to compute gradients. Perform gradient all-reduce in parallel with gradient computation

5. Update Model parameters. Because each GPU started with an identical copy of the model and gradients were all-reduced, weights updates on all GPUs are identical. Thus no model sync is required

Data Parallelism: DDP vs DP

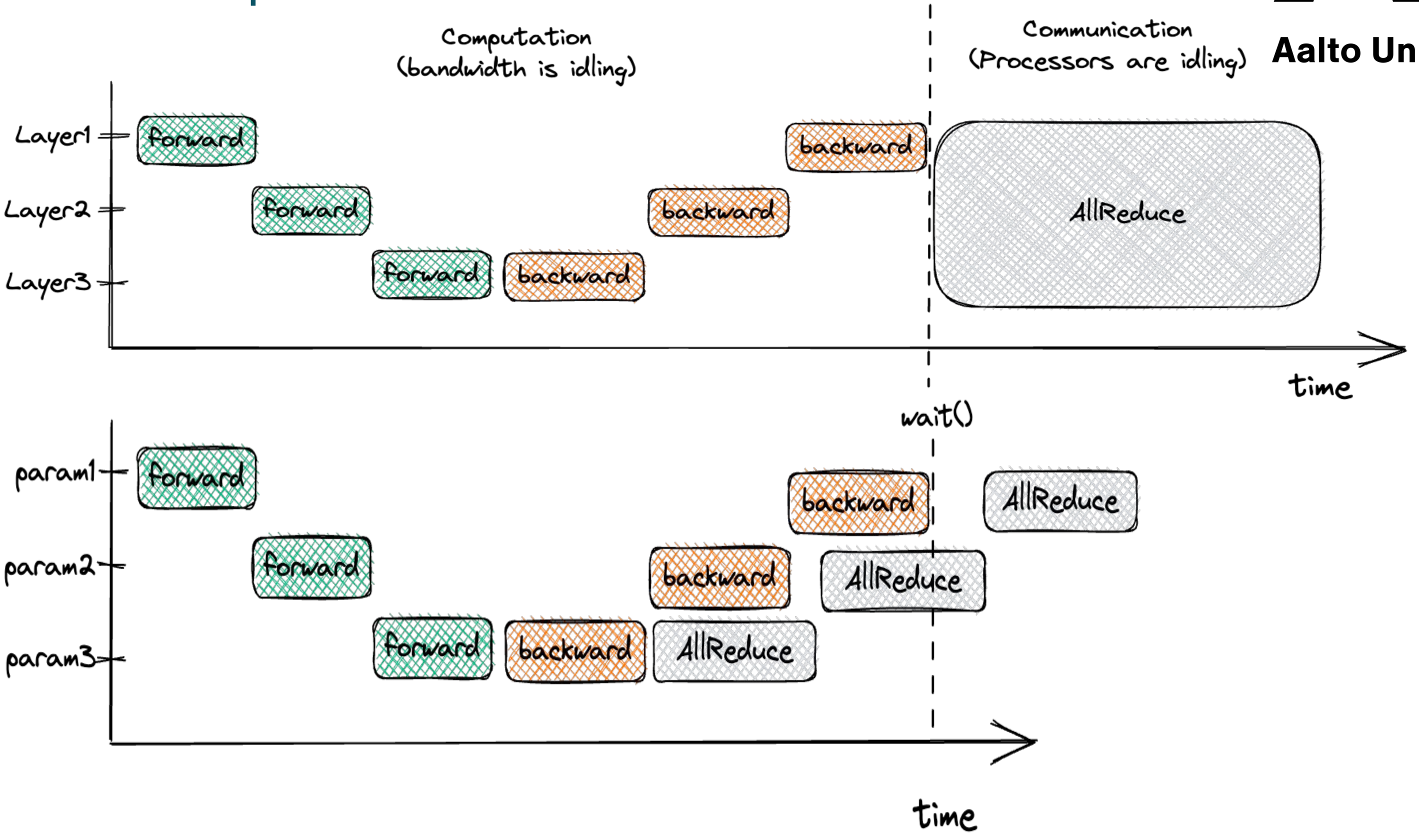
Aalto University

- DP is Python threads-based, DDP is multiprocess-based
  - No Python threads limitations, such as GIL
  - Simpler data flow
- Both have high inter-GPU communication overhead (all-reduce)
- DDP has a lower overhead, but still high
  - Overlapping pipeline of gradient all-reduce with layer gradient computation
- DDP is generally the recommended approach

DDP AllReduce overlap

No overlap:

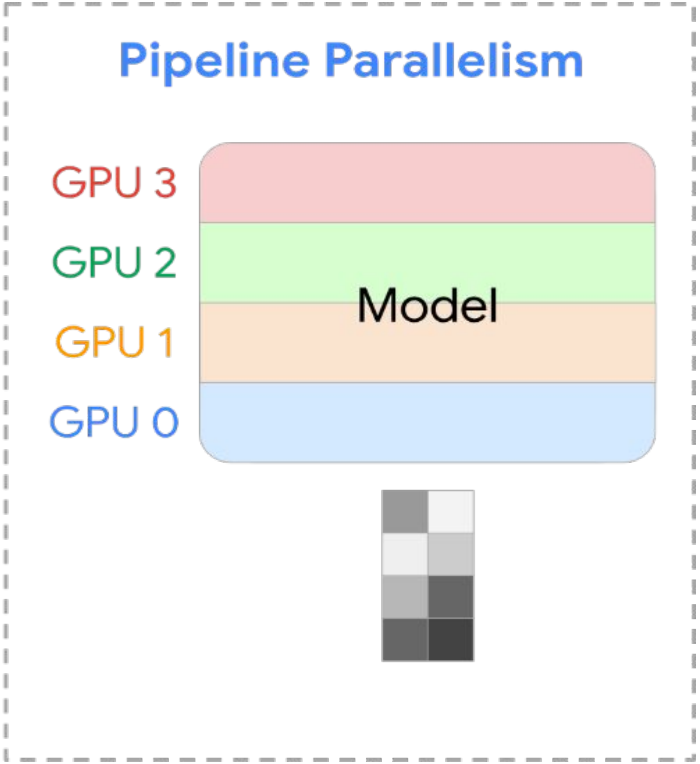
With overlap:



Reproducibility of DP / DDP

- Randomness in multi-process DataLoader
  - Each process uses `base_seed + worker_id` for the seed.
  - For reproducibility:
    - `Base_seed` needs to be set
      - `toch.Generator().manual_seed()`
  - When using multiple GPUs:
    - Use `DistributedSampler` to divide the data across GPUs
- Data Distribution of DDP:
  - Training on Mini-batchs and gradient accumulation
  - Each GPU is exposed to part of the data
  - More epochs to converge

Model Parallelism (1): Pipeline Parallelism



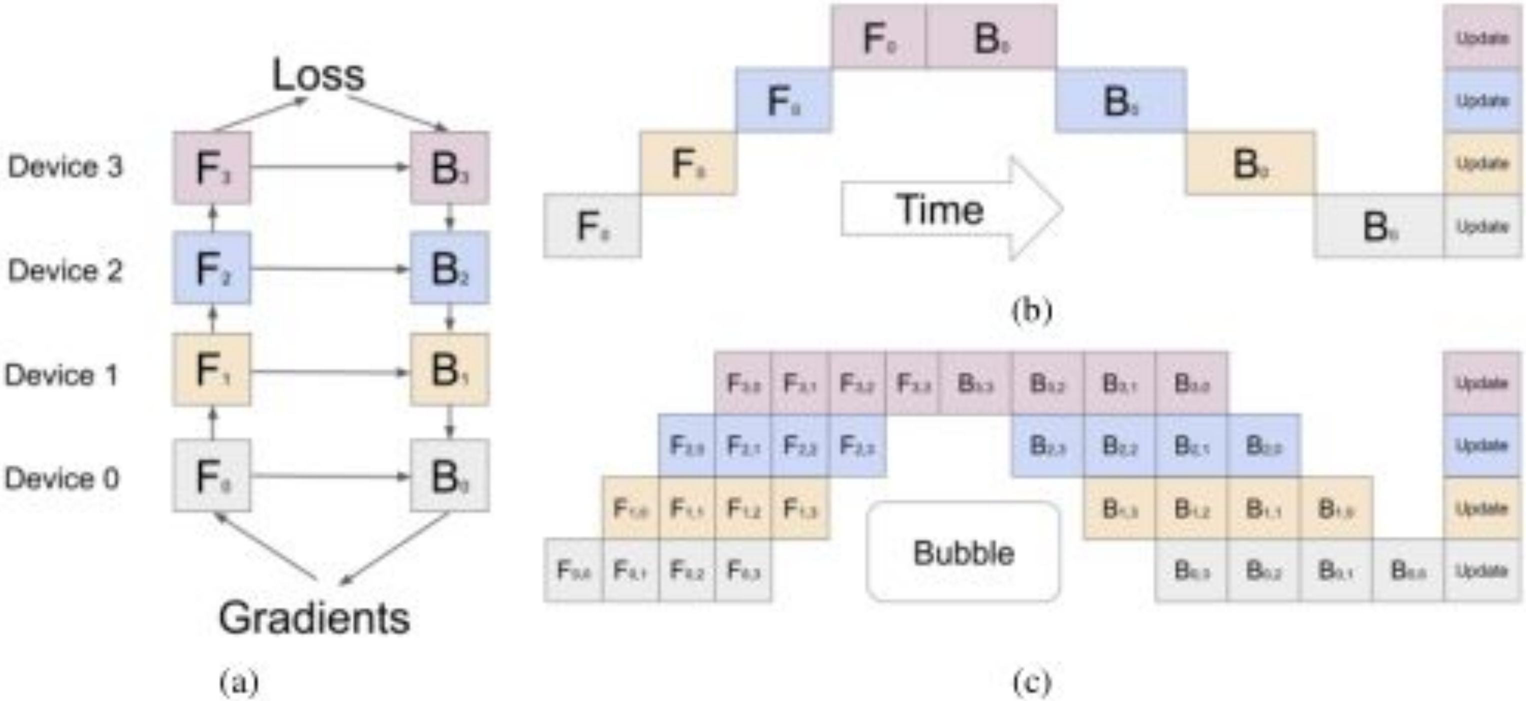
- Vertical Parallelism
- Split layer-wise across GPUs
- Each GPU processes part of the model sequentially
- Chain of dependencies

Overheads

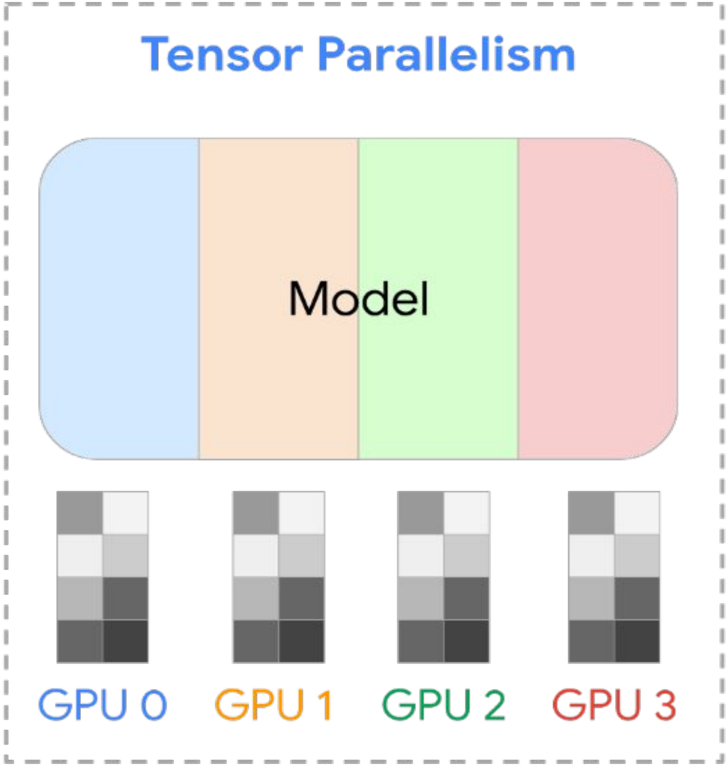
Type	Description
Communication Overhead	Low
Partial distribution	No
Underutilization	High

Chain of dependencies and bubble issue

Aalto University



Model Parallelism (2): Tensor Parallelism

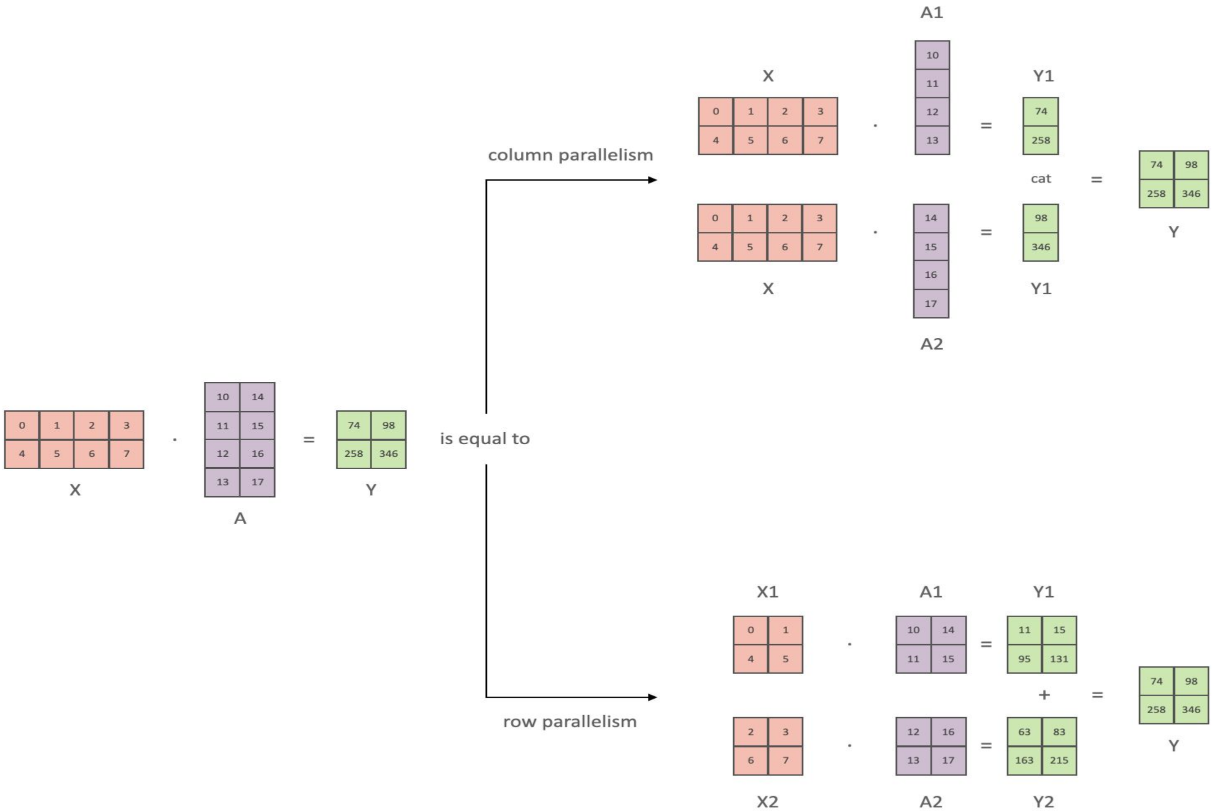


- Horizontal Parallelism
- Divide tensors horizontally
- Store part of the layers or blocks on different GPUs
- Concat outputs between GPUs manually

Overheads

Type	Description
Communication Overhead	Low
Partial distribution	No
Underutilization	No

How MP works?



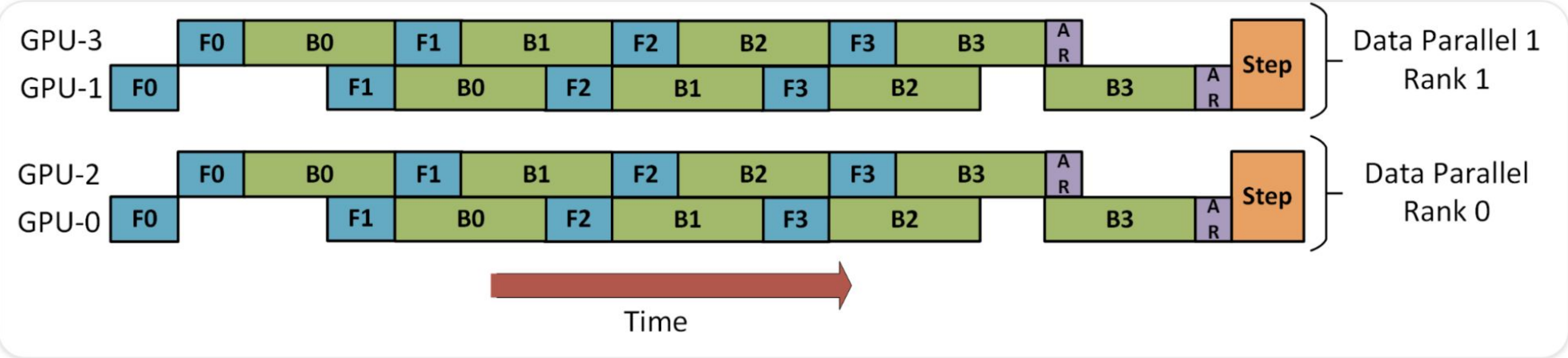
X
A
Y
is equal to
column parallelism
row parallelism
A1
A2
Y1
cat
X1
X2
Y2

Reproducibility of PP / TP

Aalto University

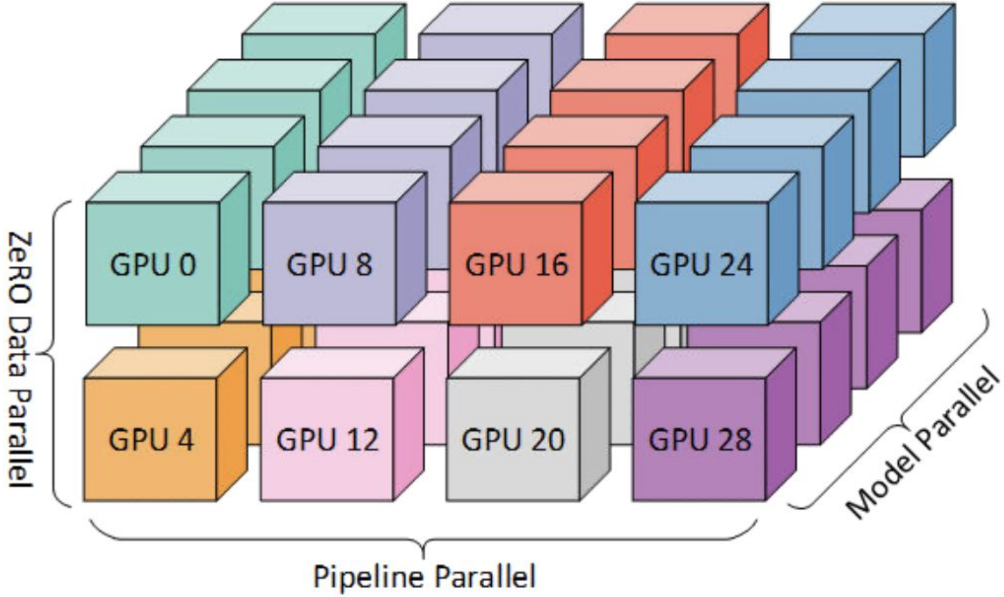
- Cross-GPU communication are asynchronous
  - small floating-point rounding differences between runs
- PP depends on micro-batches
  - Therefore data exposure is slightly different
- Device dependent random operations
  - Dropout etc will result different on each device.
- Generally speaking, the effect of PP / TP on reproducibility is <u>small</u>.

Mix and Match: DP + PP!



- It reduces the bubble issue
- For DP, there are two GPUs: GPU0 and GPU1
- Inside each DP rank, there is a PP

Reality: 3D Parallelism



- In real world: Data Parallel + Tensor Parallel + Pipeline Parallel are combined
- Example: Training GPT-3 used all three

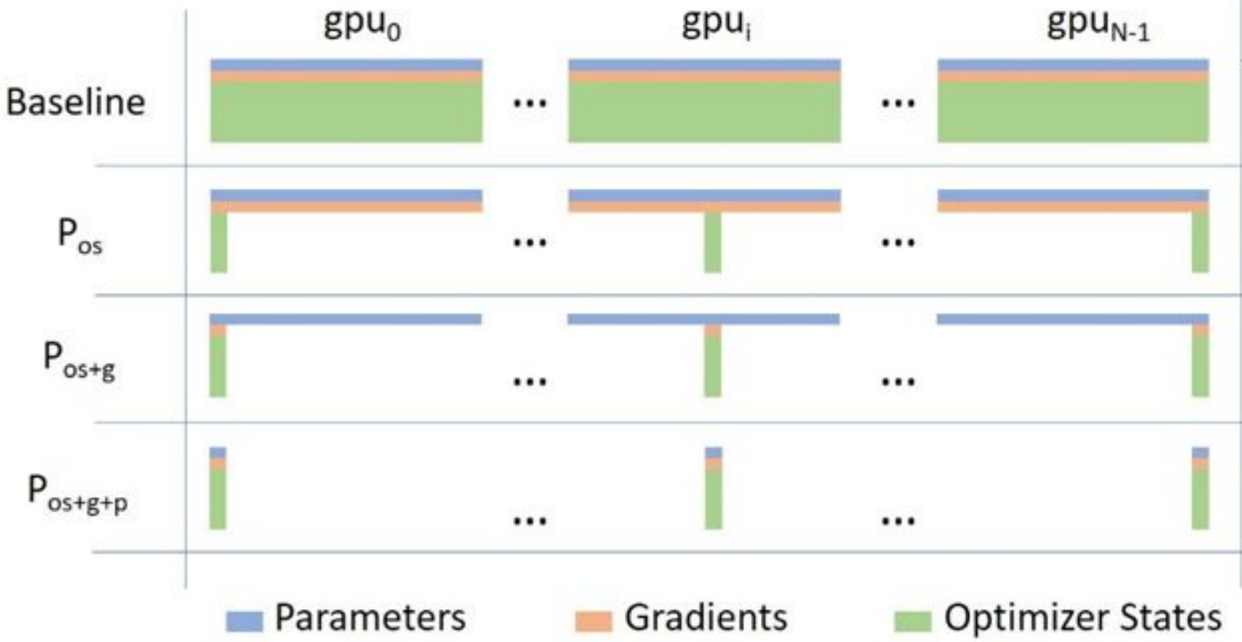
ZeRO: Advanced Data Parallelism

Aalto University

- Issue with DP: Full optimizer states and gradients duplicated on every GPU
  - Not efficient with VRAM
- ZeRO Idea: Partition optimizer states, gradients, and parameters across GPUs
- Result: Efficient use of VRAM
  - Train MUCH larger models without running out of memory

ZeRO

Aalto University



	Memory Consumption: Formulation	Memory Consumption: Specific Example K=12 Ψ=7.5B N_d=64	Comm Volume
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

ZeRO Stages

Aalto University

- Zero-1: Optimizer State Partitioning
  - Up to 4x memory reduction, same communication volume as DP
- Zero-2: Optimizer + Gradient Partitioning
  - Up to 8x memory reduction, same communication volume as DP
- Zero-3: Optimizer + Gradient + Parameter Partitioning
  - Memory reduction is linear with DP degree
  - For example, with 64 GPUs will yield a 64x memory reduction
  - There is a modest 50% increase in communication volume

ZeRO Reproducibility

Aalto University

- Issues with DDP reproducibility
  - Micro-batch setting
  - Random seed
  - Async reductions

Scaling is hard, Reproducibility is harder!

Aalto University

- Use standard libraries.
- Do checkpointing
- Monitor training
-
- Model Fits onto a single GPU → DDP or ZeRO
- Model doesn’t fit into a single GPU
    - Fast intra-node/GPU connection → PP, ZeRO, TP
    - Without intra-node/GPU connection → PP
- Largest layer not fitting into a single GPU → TP
- Multi-Node / Multi-GPU:
- ZeRO - as it requires close to no modifications to the model
- PP+TP+DDP: less communications, but requires massive changes to the
- model
- PP+TP+ZeRO-1: when you have slow inter-node connectivity and still low on
- GPU memory