Parallel and Distributed Deep Learning

Shaohao Chen

ORCD at MIT

Outline

- ❑ Basics of deep learning
- ❑ Parallel deep learning on a GPU
- ❑ Distributed deep learning on multiple GPUs
- Data parallelism

  -- Zero Redundancy Optimizer, Fully Sharded Data Parallel
- Model parallelism

  -- Pipeline parallelism

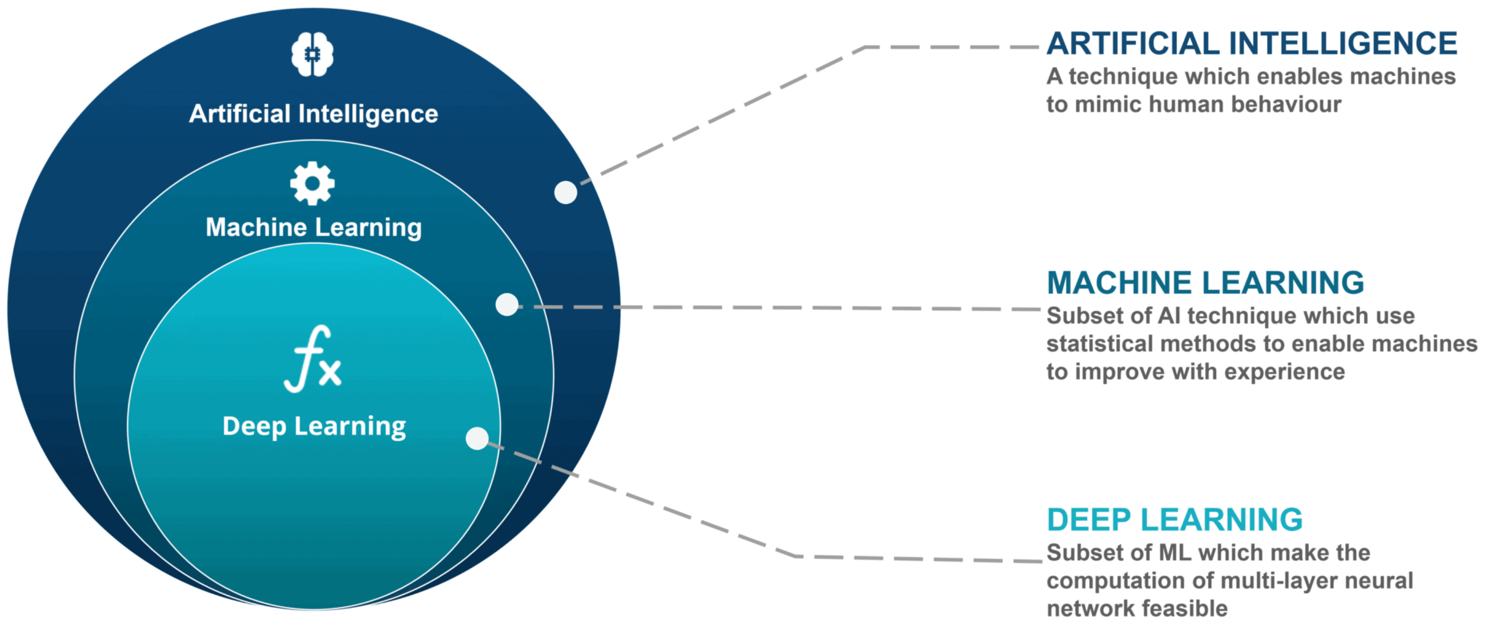
  -- Tensor parallelism

Machine learning and deep learning

- Artificial intelligence
- Machine learning

  Statistical methods

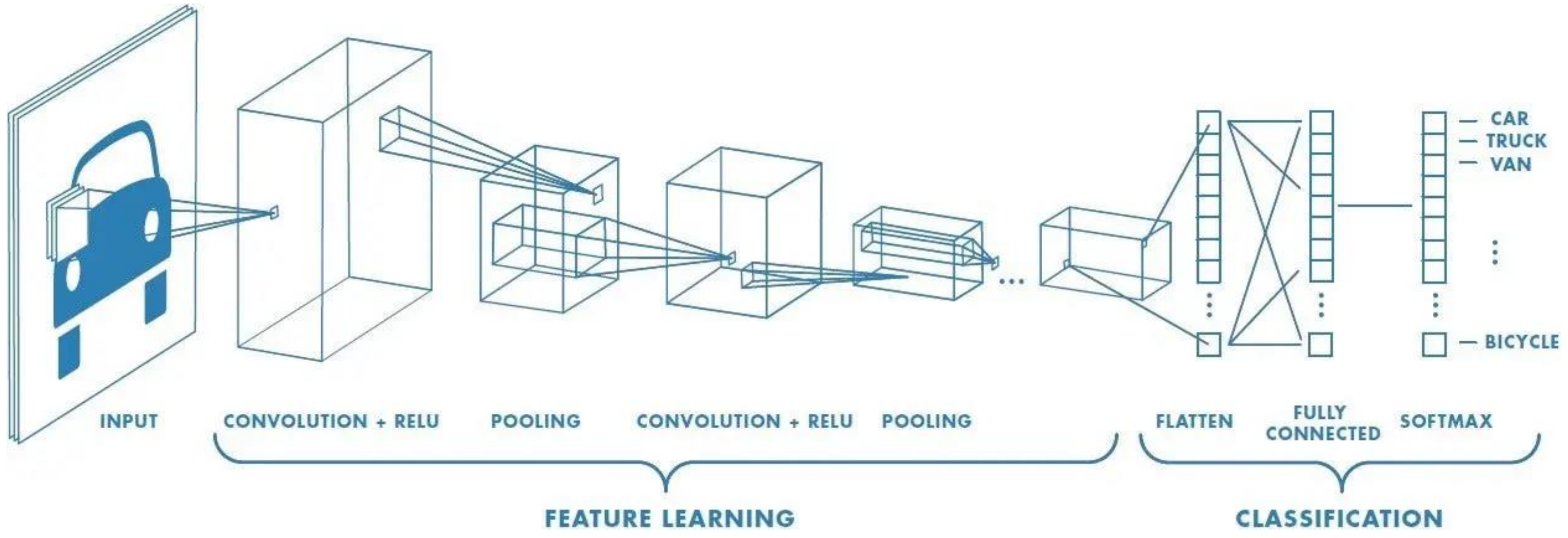
  Learn from data
- Supervised learning: classification, regression
- Unsupervised learning: clustering


Artificial Intelligence
Machine Learning
Deep Learning
ARTIFICIAL INTELLIGENCE
A technique which enables machines to mimic human behaviour
MACHINE LEARNING
Subset of AI technique which use statistical methods to enable machines to improve with experience
DEEP LEARNING
Subset of ML which make the computation of multi-layer neural network feasible

- Deep learning: deep neural network
- Cornerstones of DL: learning algorithms, big data, and high-performance computing.
- Computer vision: Convolutional Neural Network (CNN)
- Natural Language Processing (NLP): Large Language Model (LLM), transformer architecture

Convolutional Neural Network (CNN)

- [CNN for CIFAR10 in PyTorch]
- Prepare training and test datasets: load and normalize using torchvision
- Define a CNN: convolutional layers, nonlinear ReLU activation, pooling, fully connected layers, softmax



https://saturncloud.io/blog/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way/

Train a neural network

- Train: adjust the model to minimize a loss function.
- Loss function: cross entropy

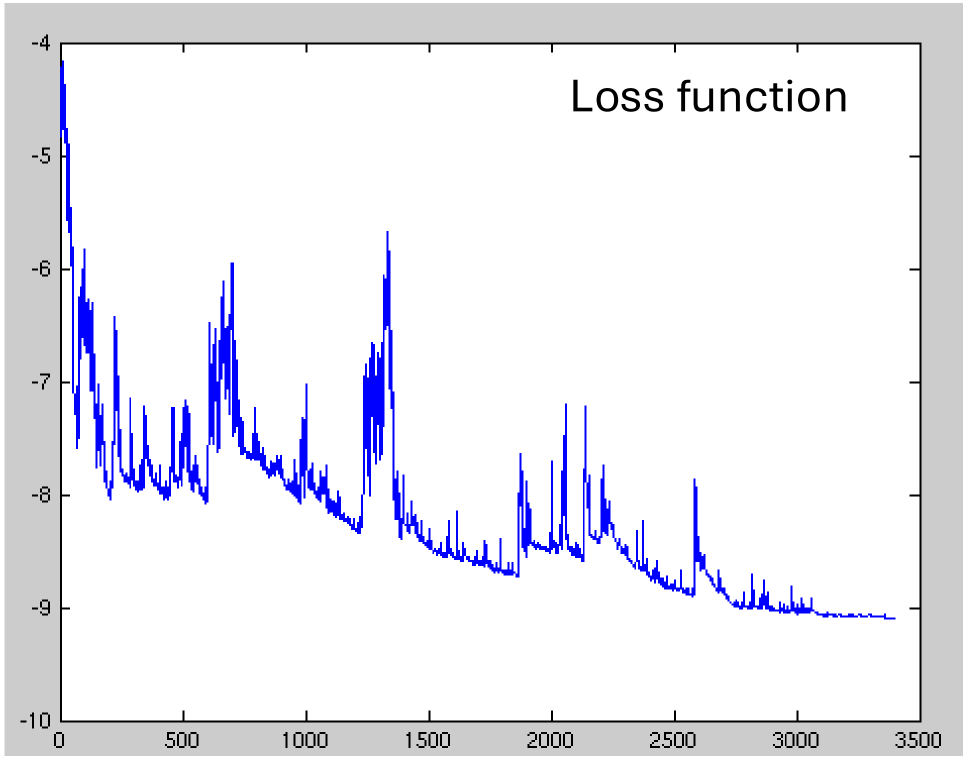
$$-\frac{1}{N}\sum_{n=1}^{N}\left[y_n \log \hat{y}_n + (1-y_n)\log(1-\hat{y}_n)\right]$$

- Optimizer: Stochastic Gradient Descent (SGD)
- Training data: batch or mini-batch (a randomly-picked subset of data), epoch (loop over all data).
- Train the network on the training data:

-- forward + backward + optimize

-- Backpropagation: computes the gradient of the loss function with respect to the weights, one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule of derivatives.

- Test the network on the test data



https://en.wikipedia.org/wiki/Stochastic_gradient_descent

Training on a GPU with PyTorch

- Define a CUDA device

```
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
```

- Send the model to the GPU

```
net.to(device)
```

- Training process:

```
for epoch in range(2):  # loop over the dataset multiple times
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):   # get a batch of data

        # Send a batch of data to the GPU at every step
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()     # initialize gradients
        outputs = net(inputs)     # forward pass
        loss = nn.CrossEntropyLoss(outputs, labels) # calculate loss
        loss.backward()           # backward pass
        optimizer.step()          # optimize
```

Prepare on ORCD clusters

- Get started: https://orcd-docs.mit.edu/getting-started/
- Log in Engaging

```
ssh <user>@orcd-login002.mit.edu
```

- Install PyTorch, Deepspeed, dependencies, and related tools.

```
module load  miniforge/23.11.0-0
conda create -n ds
source activate ds
conda install PyTorch==2.4.1 torchvision==0.19.1 torchaudio==2.4.1 pytorch-cuda=12.4 -c PyTorch -c nvidia
pip install deepspeed
pip install datasets  tensorboard transformers
pip install fire loguru sh matplotlib
```

Submit a job to a GPU

- Prepare a job script named job.sh

```
#!/bin/bash
#SBATCH -p mit_normal_gpu
#SBATCH -t 30
#SBATCH -N 1
#SBATCH -n 1
#SBATCH --mem=10GB
#SBATCH --gres=gpu:1

module load miniforge/23.11.0-0
source activate ds
python cnn_cifar10_gpu.py
```

- Submit the job

```
sbatch job.sh
```

https://orcd-docs.mit.edu/recipes/torch-gpu/

What happens under the hood?

What about parallel?

- Training a neural network involves large-scale linear algebra computations.
- When PyTorch is built with CUDA support, it dynamically links to cuDNN and cuBLAS libraries.
- Linear algebra computations are optimized and parallelized in cuBLAS and thus accelerated on GPUs.

What about other platforms or libraries?

- cuDNN: C API, a bridge between deep-learning platforms and GPUs.
- Tensorflow: Python or C API, a steeper learning curve, less friendly to researchers, easier with Keras integration, better performance optimizations, better for developers.

Distributed Parallelism for Deep Learning

- Distributed on multiple GPUs.

- Data Parallelism

Each GPU gets a different batch of data.

Process more data at the same time

Universal to different models

- Model Parallelism

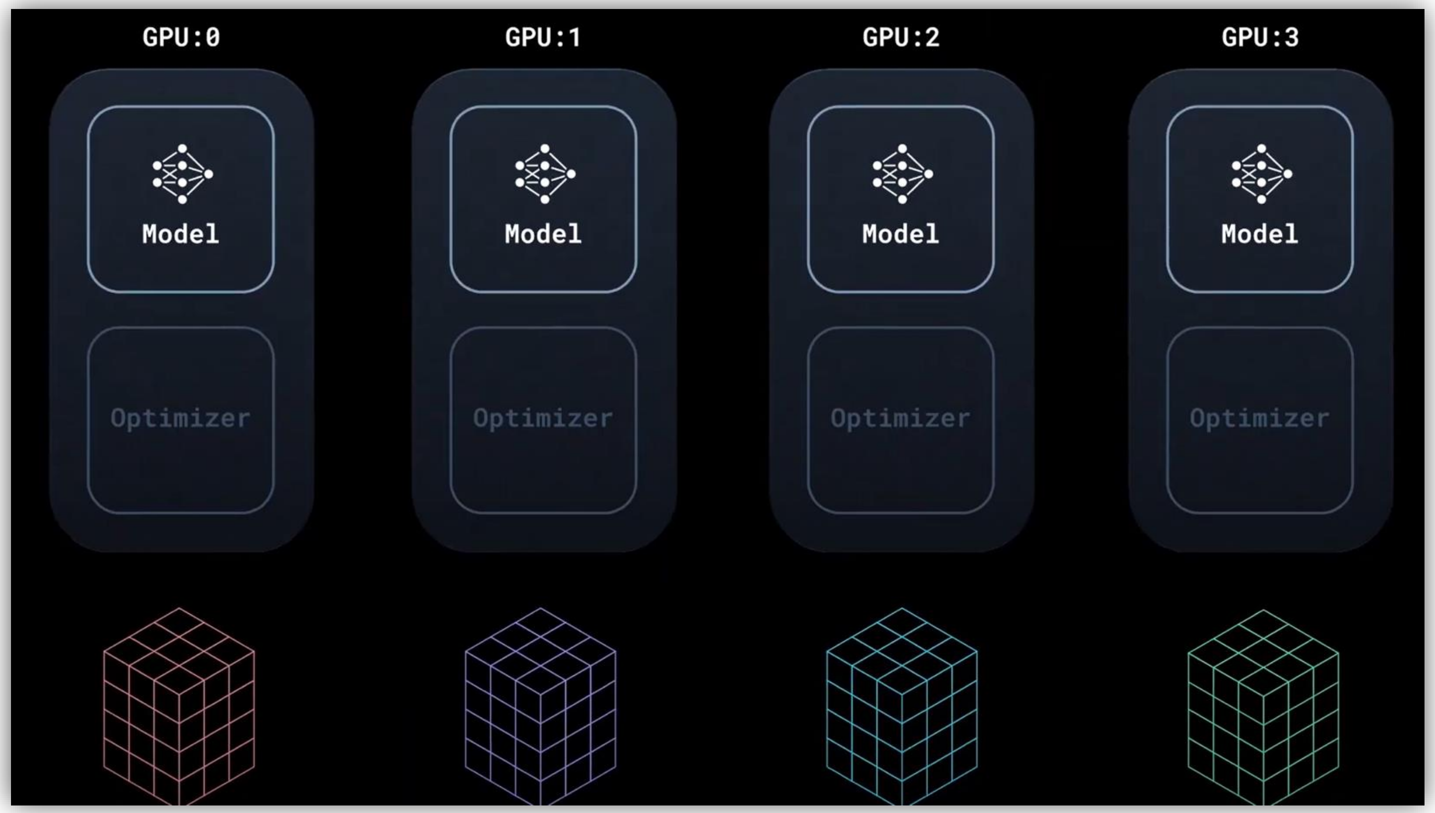
A model is too big to be stored on a GPU.

Partition the model on multiple GPUs.

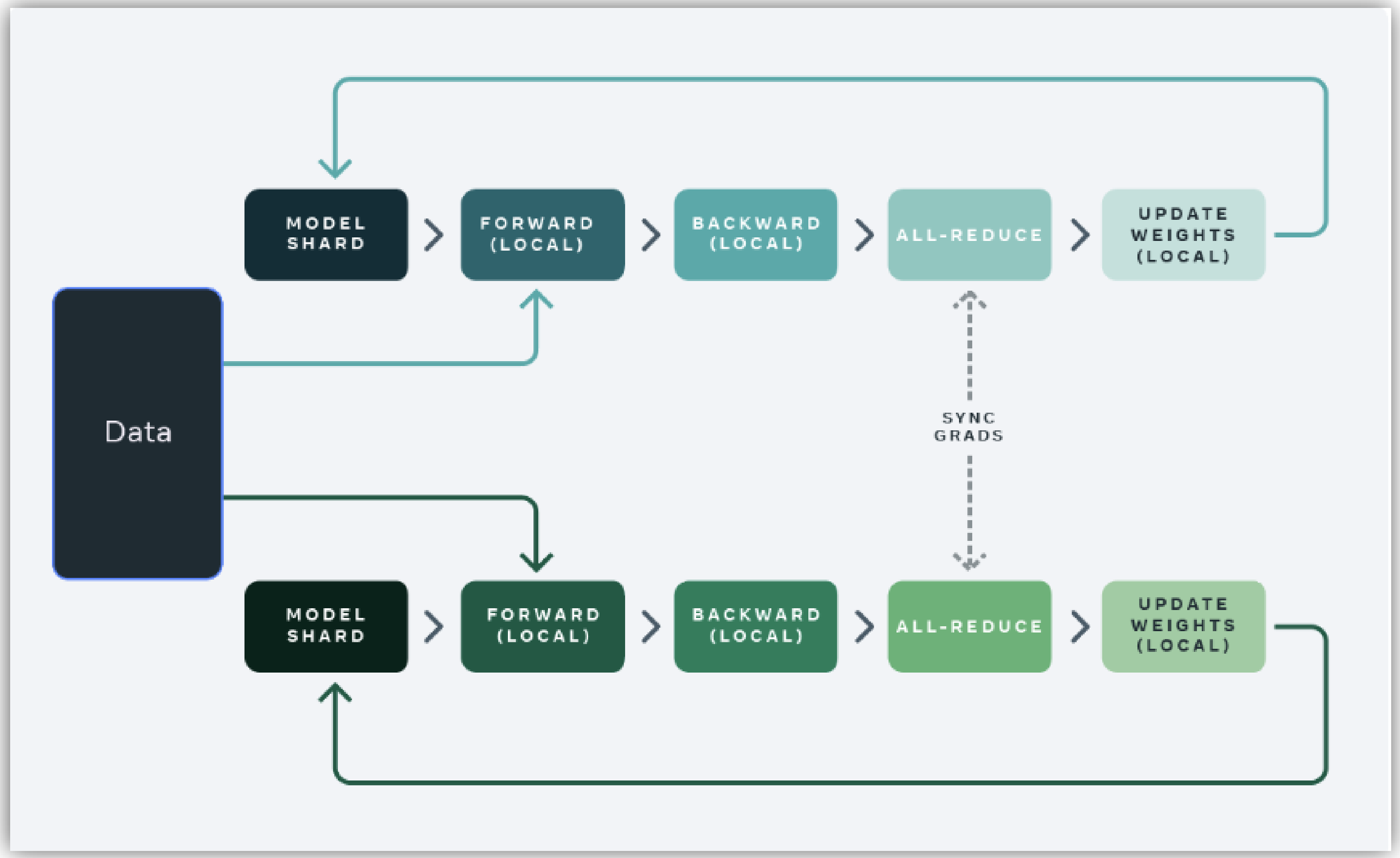
Tricky to design and implement.

Data Parallelism

- Each GPU has a copy of the model
- Each GPU gets a different batch of data
- Data sampling is handled by a Distributed Sampler
- Concurrently processing multiple batches of data



Communication in data parallel



- Gradients on each GPU are different because the input data is different.
- Gradients from each GPU are synchronized before the update.
- Synchronization is done with a bucketed Ring-AllReduce algorithm.
- Each GPU gets the averaged gradient, then models are updated locally.
- Overlap gradient computation with communication so GPUs are utilized efficiently.

Distributed Data Parallel with PyTorch

- Linear neural network $y = xA^T + b$

```
model = torch.nn.Linear(20, 1)
```

- Set up GPU ID

```
torch.cuda.set_device(rank)
```

- Apply DDP

```
self.model = DistributedDataParallel(model, device_ids=[gpu_id])
```

- Spawn training processes on multiple GPUs

```
world_size = torch.cuda.device_count()

mp.spawn(main, args=(world_size, args.save_every, args.total_epochs, args.batch_size), nprocs=world_size)
```

- Communication is under the hood. PyTorch calls NCCL.

https://PyTorch.org/tutorials/beginner/ddp_series_multigpu.html , https://PyTorch.org/docs/stable/notes/ddp.html

Data Parallel Jobs on Multiple GPUs

- Single-node Multi-GPU

```
#!/bin/bash
#SBATCH -p mit_preemptable
#SBATCH -N 1
#SBATCH -n 2
#SBATCH --mem=20GB
#SBATCH --gres=gpu:2
module load miniforge/23.11.0-0
source activate ds
# Run with python
python multigpu.py --batch_size=1024 100 20
# or, run with torchrun
torchrun --nnodes=$SLURM_NNODES --nproc_per_node=$SLURM_NTASKS   \\
         --rdzv_id=$SLURM_JOB_ID --rdzv_endpoint="localhost:1234"    \\
         multigpu_torchrun.py --batch_size=1024 100 20
```

- Multi-node Multi-GPU: https://orcd-docs.mit.edu/recipes/torch-gpu/

Drawbacks of Data Parallel

- Training on more data in a shorter time can lead to faster convergence, but it is not a guaranteed.
- Inefficient convergence when scaled up to a large number of GPUs
- Other ways to obtain faster convergence: tuning hyperparameters like learning rate and batch size.
- Scaling with data parallel introduces communication overhead when syncing gradients.
- The model must fit within GPU memory.

Why big models?

- Transformer architecture

  -- Remove the sequential processing dependency of RNNs, such as Long Short-term Memory (LSTM).

  -- Enable language models to be trained with parallelism

- A dramatic increase in model sizes after the birth of Transformer.

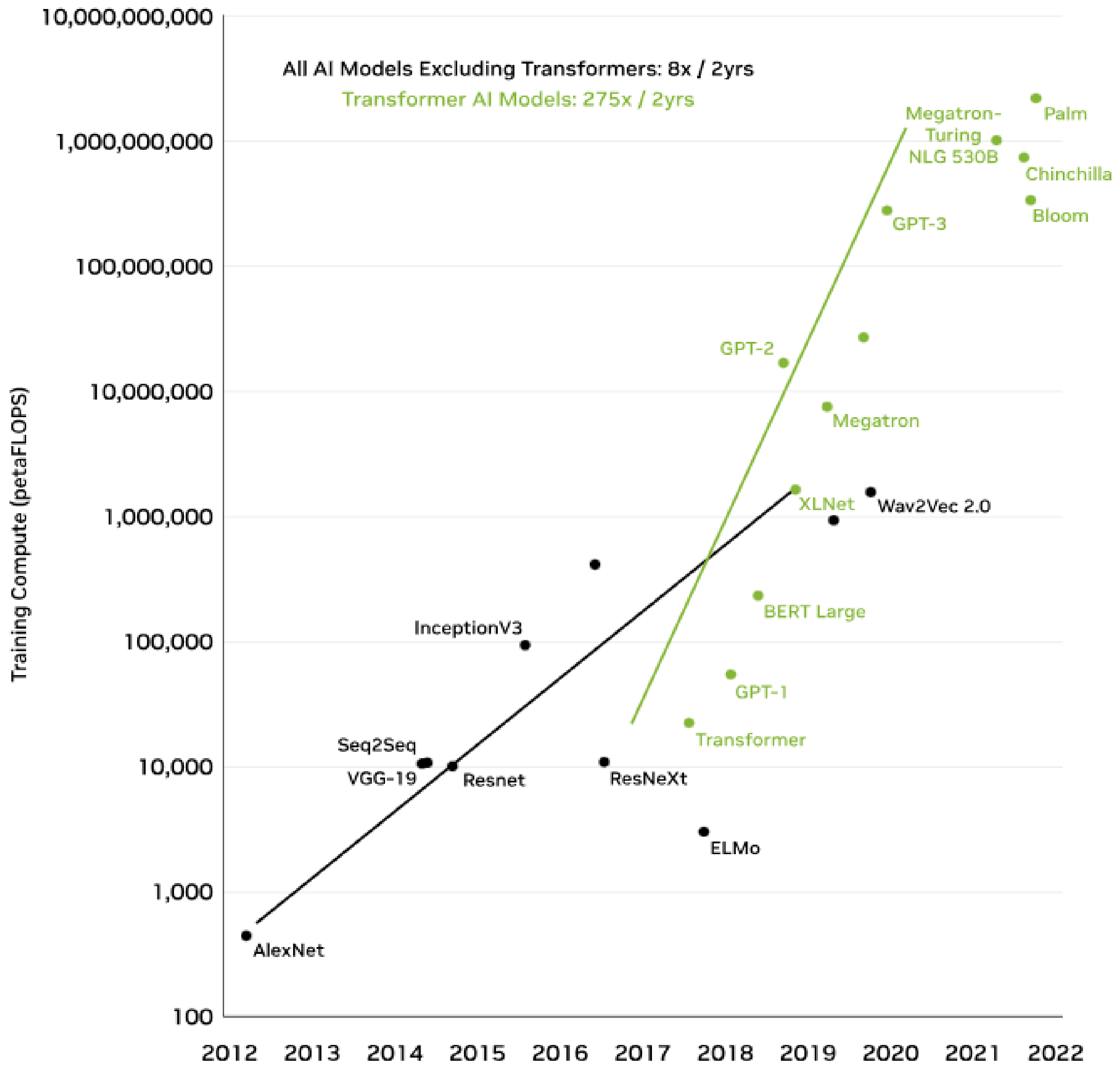
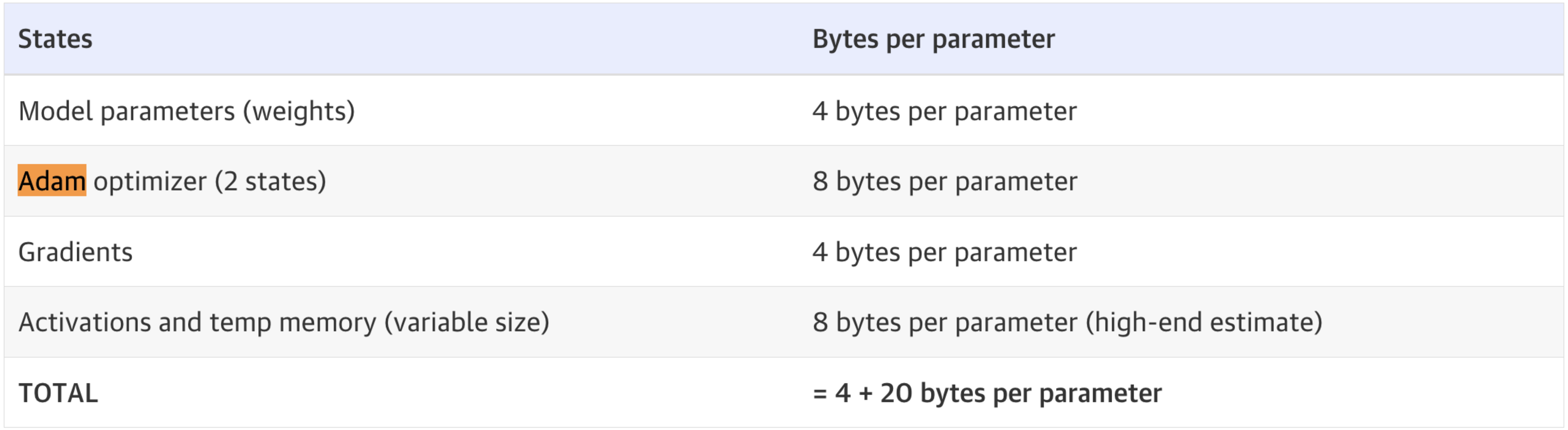


Figure 3. Compute required for training transformer models.

Memory requirements by big models

- Adam optimizer: 24 bytes per parameter for FP32



States	Bytes per parameter
Model parameters (weights)	4 bytes per parameter
Adam optimizer (2 states)	8 bytes per parameter
Gradients	4 bytes per parameter
Activations and temp memory (variable size)	8 bytes per parameter (high-end estimate)
TOTAL	**= 4 + 20 bytes per parameter**

- 1 billion parameters:

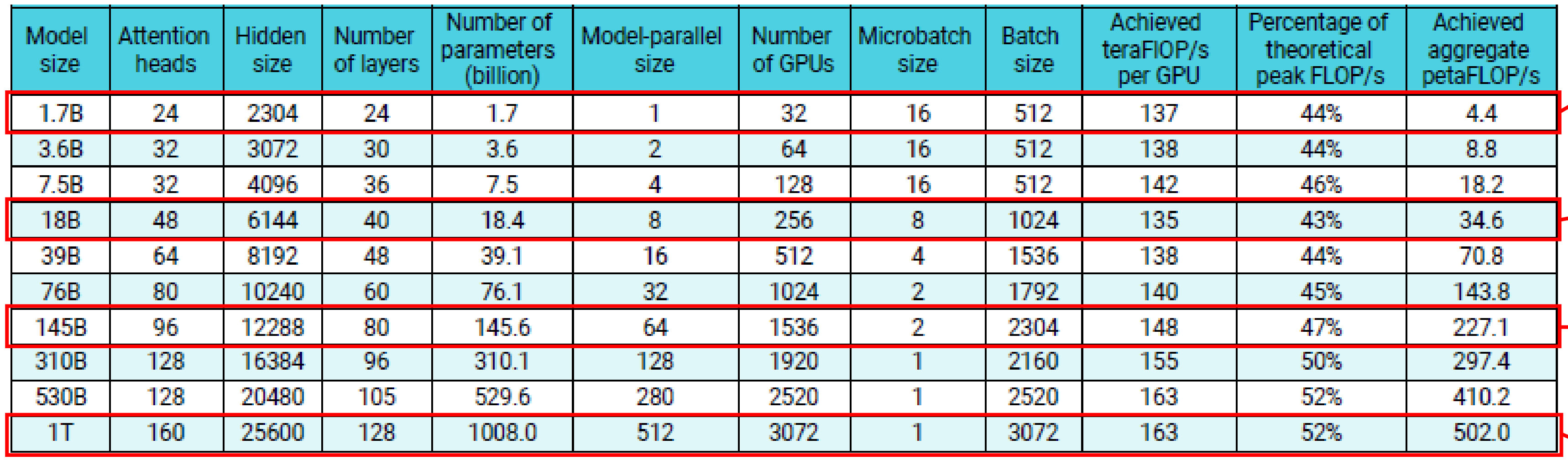
  24 GB for FP32, 12 GB for FP16, 16 GB for mixed-precision (FP32 for optimizer states, FP16 for the rest)

https://www.oreilly.com/library/view/generative-ai-on/9781098159214/ch04.html#:~:text=Quantization%20from%20fp32%20to%20fp16,shown%20in%20Figure%204%2D6

https://www.geeksforgeeks.org/adam-optimizer/

https://www.determined.ai/blog/act-mem-2

Scale of compute with big models



Model size	Attention heads	Hidden size	Number of layers	Number of parameters (billion)	Model-parallel size	Number of GPUs	Microbatch size	Batch size	Achieved teraFlOP/s per GPU	Percentage of theoretical peak FLOP/s	Achieved aggregate petaFLOP/s
1.7B	24	2304	24	1.7	1	32	16	512	137	44%	4.4
3.6B	32	3072	30	3.6	2	64	16	512	138	44%	8.8
7.5B	32	4096	36	7.5	4	128	16	512	142	46%	18.2
18B	48	6144	40	18.4	8	256	8	1024	135	43%	34.6
39B	64	8192	48	39.1	16	512	4	1536	138	44%	70.8
76B	80	10240	60	76.1	32	1024	2	1792	140	45%	143.8
145B	96	12288	80	145.6	64	1536	2	2304	148	47%	227.1
310B	128	16384	96	310.1	128	1920	1	2160	155	50%	297.4
530B	128	20480	105	529.6	280	2520	1	2520	163	52%	410.2
1T	160	25600	128	1008.0	512	3072	1	3072	163	52%	502.0

Weak scaling throughput for GPT models ranging from 1 billion to 1 trillion parameters.

~6 weeks on 1 x DGX A100
~2 weeks on 4 x DGX A100

~65 weeks on 1 x DGX A100
~16 weeks on 4 x DGX A100

~5 years on 1 x DGX A100
~1 year on 4 x DGX A100

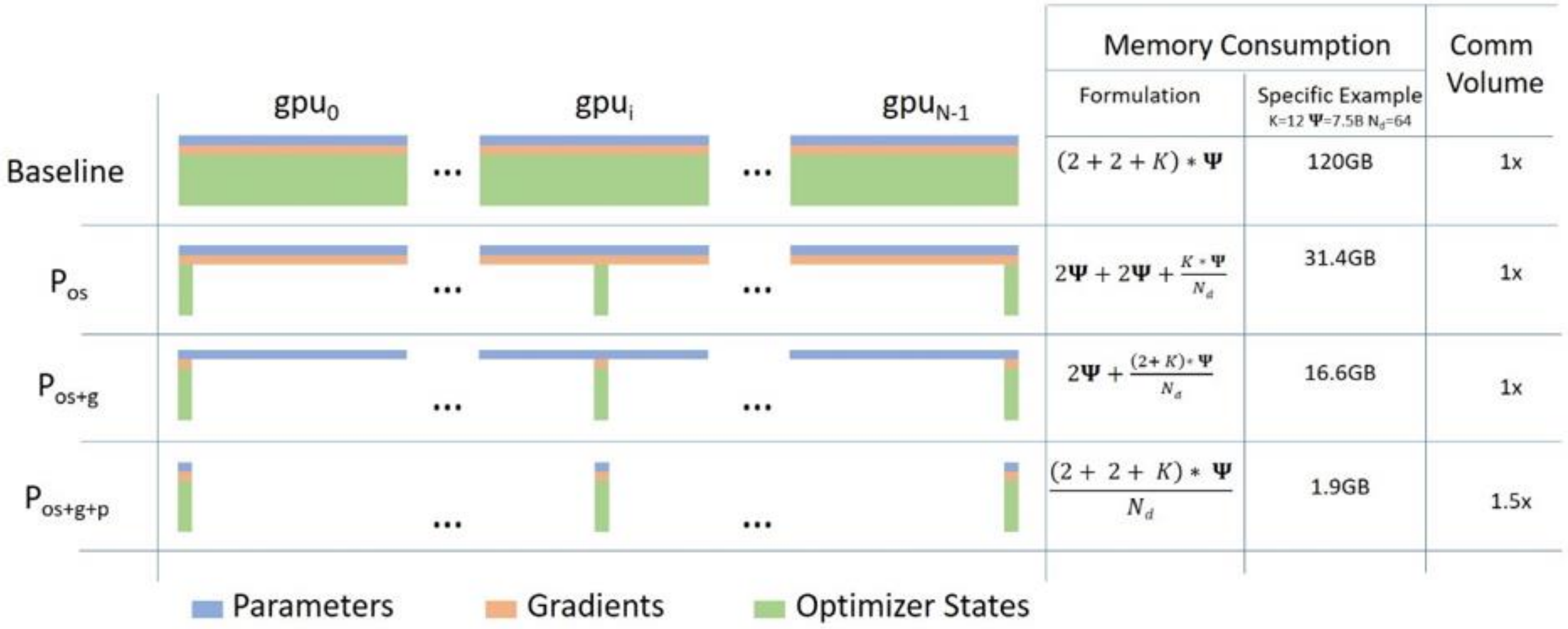
~69 years on 1 x DGX A100
~17 year on 4 x DGX A100

Scaling up of training enables faster time to convergence.

Narayanan, Deepak, et al. "**Efficient large-scale language model training on GPU clusters using megatron-LM**." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2021.

Zero Redundancy Optimizer (1)

- Operate within the data parallel framework, optimizing memory usage by distributing model states across data parallel workers.
- Each GPU stores only a slice of model parameters, gradients, and/or optimizer states.
- Communication: Each GPU receives missing slices of parameters from other GPUs when needed.



	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=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

https://www.microsoft.com/en-us/research/blog/zero-deepspeed-new-system-optimizations-enable-training-models-with-over-100-billion-parameters/

Zero Redundancy Optimizer (2)

- Quick and easy: only need to change a few configurations in the configuration JSON.

  Does not require a code redesign or model refactoring.

- ZeRO may or may not be faster than regular DP.

- ZeRO is implemented in Deepspeed.

- Fully Sharded Data Parallel (FSDP): an implementation of the ZeRO concept in PyTorch.

- Save more GPU memory

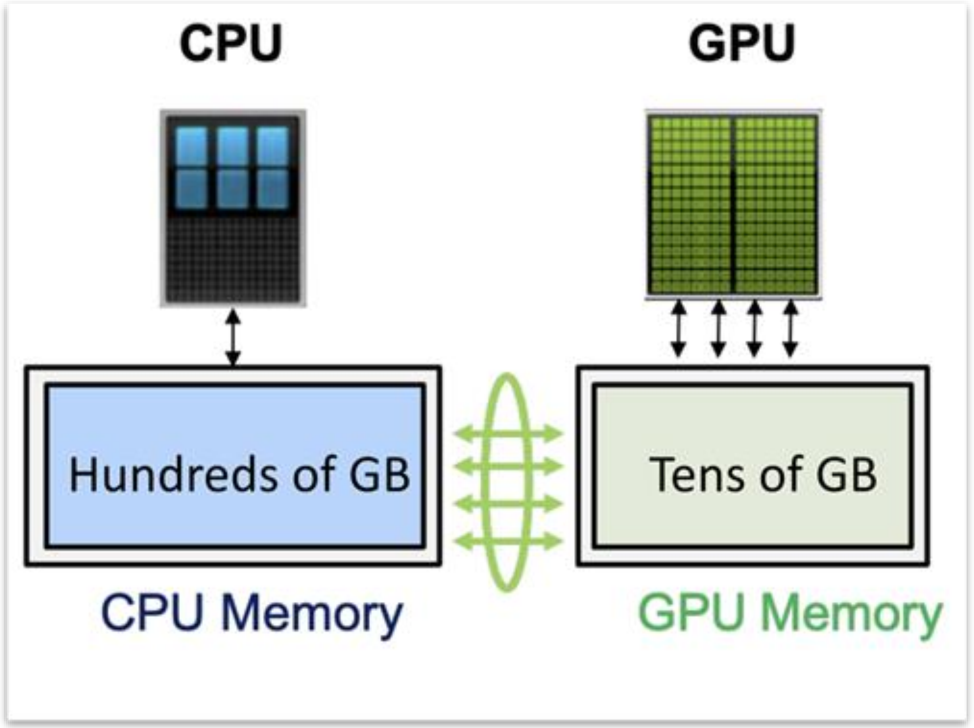
  -- Mixed precision: weights and gradients stored in FP16, optimizer states stored in FP32

  -- Checkpointing activations

  -- Offloading to CPU

https://www.deepspeed.ai/tutorials/zero/ https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Offloading to CPU



Offload CPU tensors not used in computation form GPU to CPU

- Training times will be slower due to slow data movement.
- Overlap communication with computation.

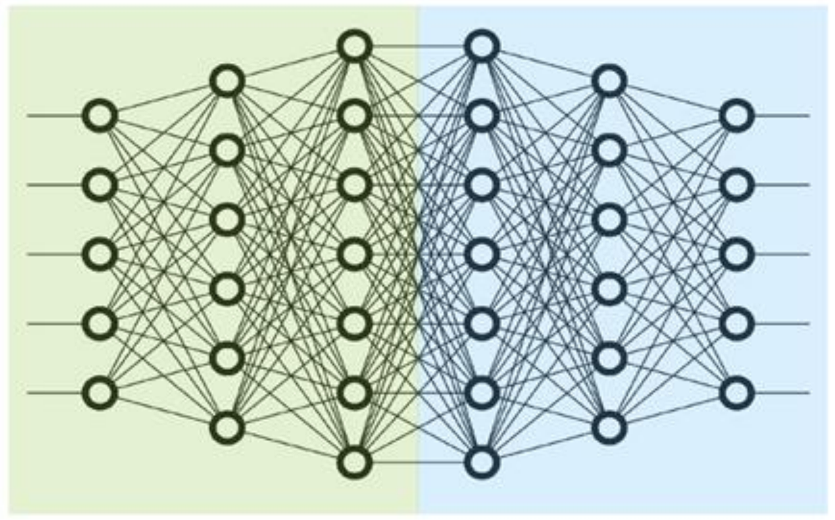
Model Parallelism

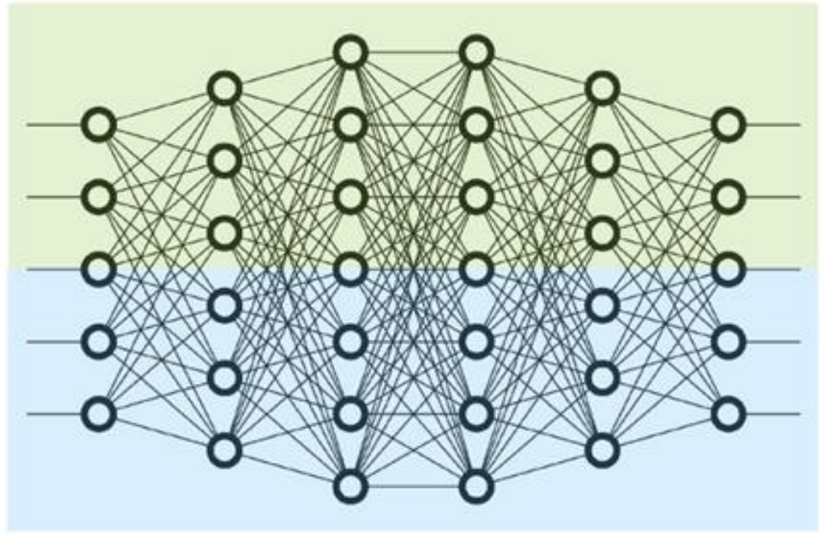
❑ Pipeline (Inter-Layer) Parallelism

- Split the model vertically
- Only one or several layers of the model are placed on a single GPU.
- Each GPU processes in parallel different stages of the pipeline and works on a small chunk of the batch.

❑ Tensor (Intra-Layer) Parallelism

- Split the model horizontally
- Each tensor is split into multiple shards, and each shard resides on its designated GPU.
- Each shard is computed in parallel on different GPUs and the results are synced at the end of the step.

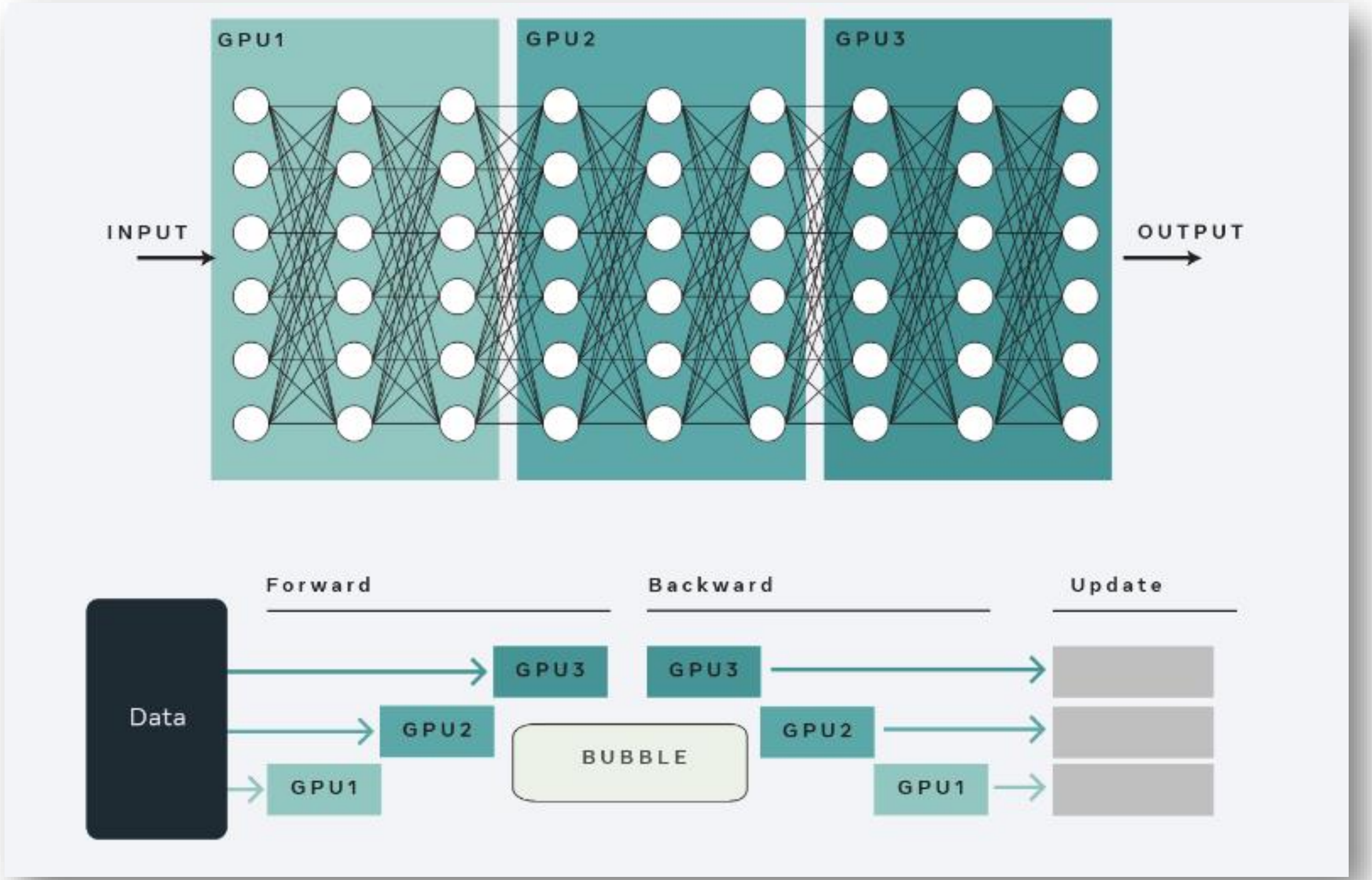




https://huggingface.co/docs/transformers/v4.13.0/en/parallelism

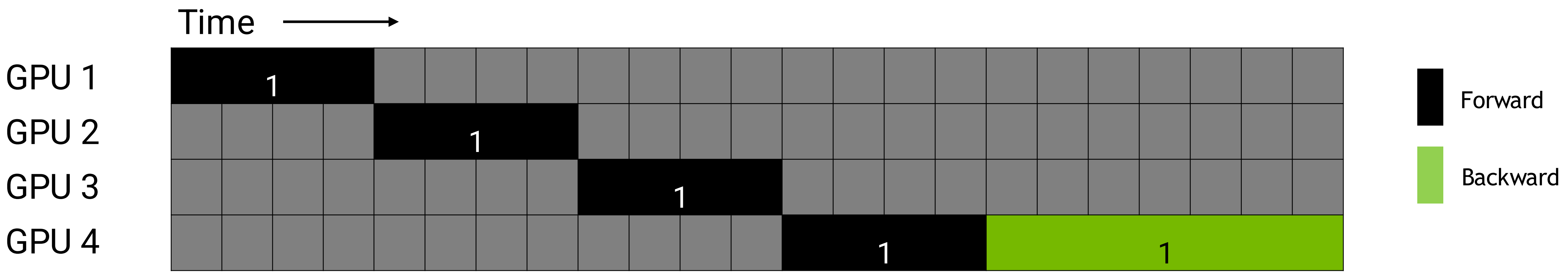
Pipeline Parallelism (1)

- Naive pipeline parallel is sequentially processed.
- Leads to GPU underutilization.

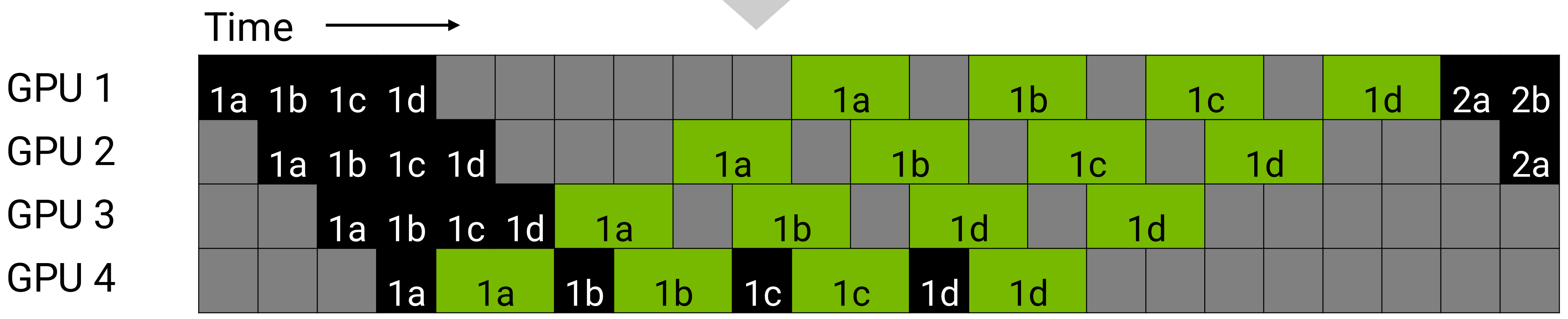


https://fairscale.readthedocs.io/en/latest/deep_dive/pipeline_parallelism.html

Pipeline Parallelism (2)

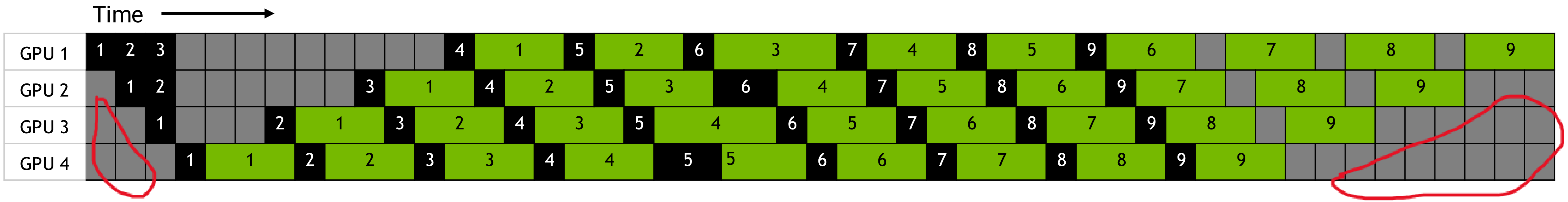


Split batch into micro batches and pipeline executions



Pipeline Parallelism (3)

Split batch into micro batches and pipeline executions to increase GPU utilization.



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

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

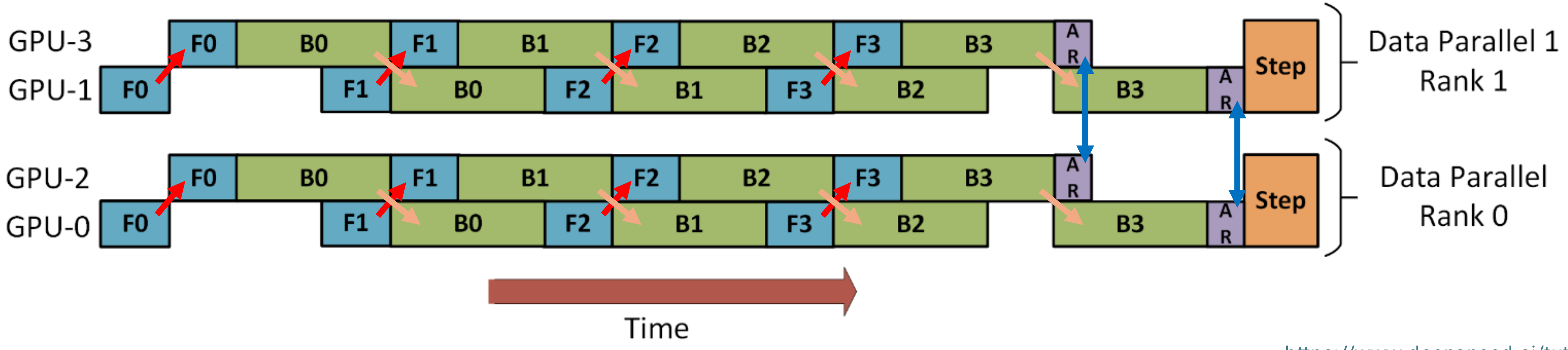
p : number of pipeline stages
m : number of micro batches
t_f : forward step time
t_b : backward step time

$$speedup = \frac{t_1}{t_p} = \frac{m*p*(t_f+t_b)}{(m+p-1)(t_f+t_b)} = \frac{m*p}{m+p-1}$$

3 times speedup with 4 pipeline stages and 9 micro batches.

Data and Pipeline Parallel

- Hybrid parallel: two-way data parallel, two pipeline stages, and eight micro-batches.
- GPUs 0 and 2 are arranged in a pipeline and alternate forward (F) and backward (B) passes — the same for GPUs 1 and 3.
- In the forward pass on a micro-batch, the activation is communicated to the next pipeline stage.
- In the backward pass on a micro-batch, the gradient with respect to the activation is communicated to the next pipeline stage.
- Each backward pass accumulates gradients locally, then a GPU will all-reduce (AR) gradients with its data-parallel counterpart (0 - 1, 2 - 3).
- Finally, the two pipeline stages update their model weights.



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

Data and Pipeline Parallel with Deepspeed

- Alexnet: 5 convolutional layers + 2 fully connected hidden layers + 1 fully connected output layer.

```
net = AlexNet(num_classes=10)
```

- Set up a pipeline module

```
net = PipelineModule(layers=join_layers(net),
            loss_fn=torch.nn.CrossEntropyLoss(),
            num_stages=args.pipeline_parallel_size,
            partition_method=parameters,
            activation_checkpoint_interval=0)
```

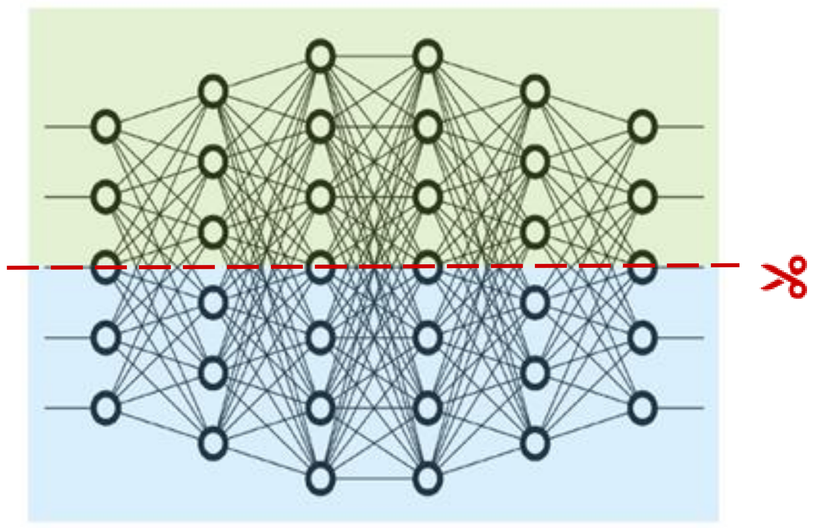
- Set the micro batch size in the configuration JSON

```
"train_micro_batch_size_per_gpu" : 8,
```

- Run the program. The total number of GPUs must be divisible by the number of pipeline stages.

```
deepspeed train.py --deepspeed_config=ds_config.json -p 2 --steps=200
```

Tensor Parallelism

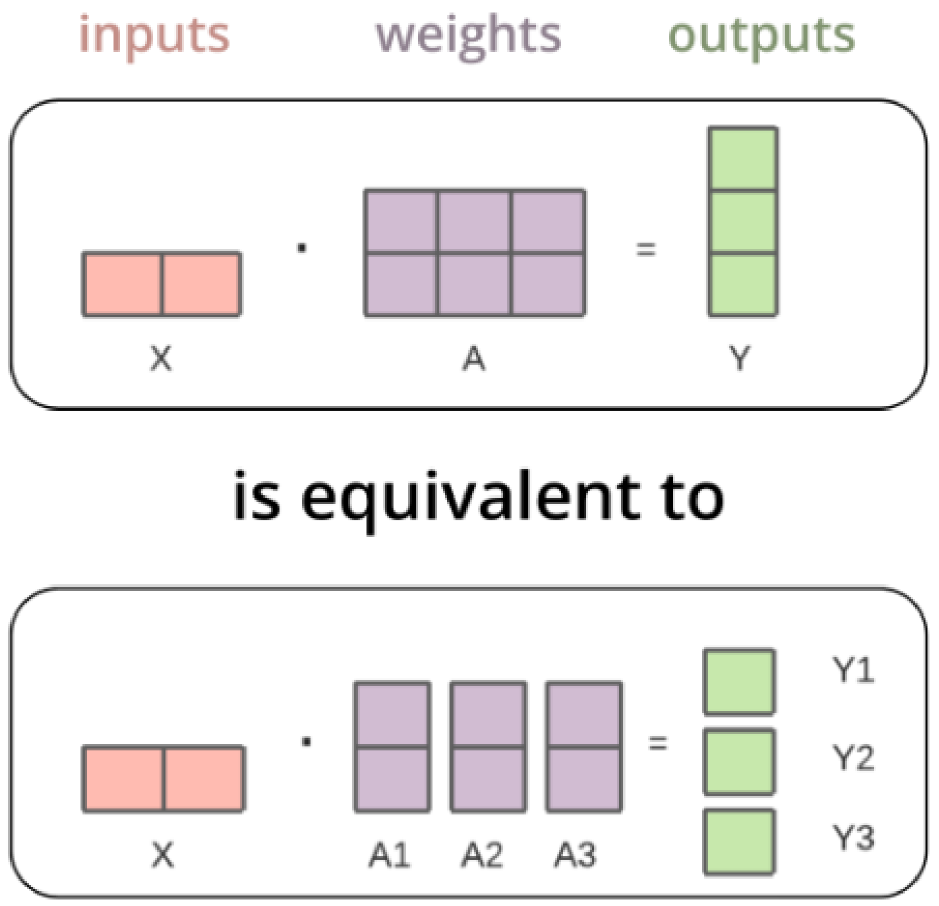


- Use to scale beyond data parallelism
- Less restrictive on the batch size (avoids bubble issue in pipelining)
- Reduces memory proportional to the number of workers (model dependent)
- Sharded computations work well for large matrices (e.g. Transformers)
- Large communication overhead. Does not scale well beyond the node boundary.

The implementation of TP depends on the neural network architecture.

A simple example of tensor parallelism

- Linear neural network
- Sharded matrix-vector multiplication on multiple GPUs
- The local results are transferred between GPUs and combined to get the final result.

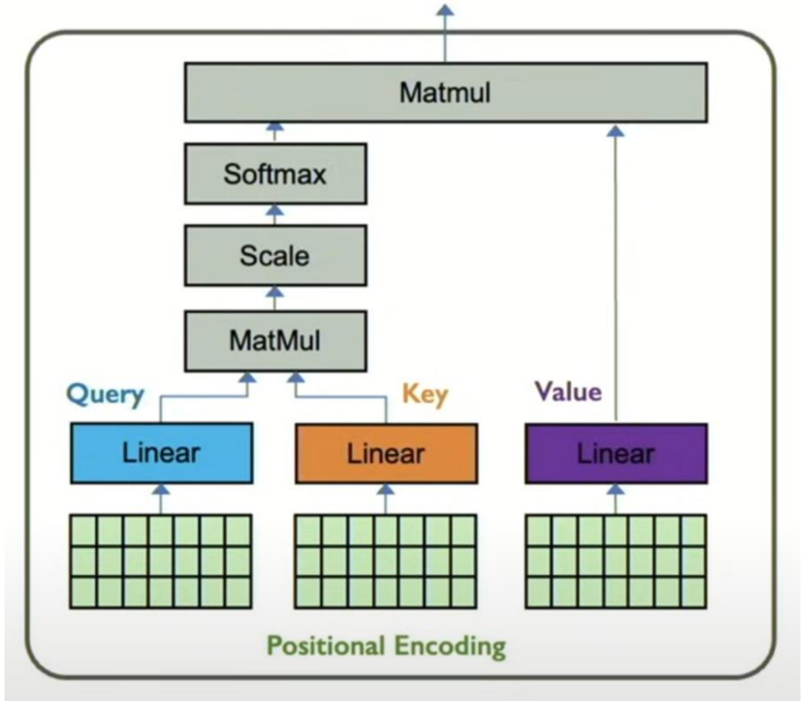


https://huggingface.co/docs/text-generation-inference/en/conceptual/tensor_parallelism

Transformer architecture

Attention is all you need

- Positional embedding. token → vector in high-dimensional space.
- Query, Key, and Value vectors.
- Attention weighting/mask: cosine similarity between query and key
- Extract features with high attention: multiply attention mask and value.
- A self-attention head.
- Transformer: a neuro network built on multiple self-attention heads.



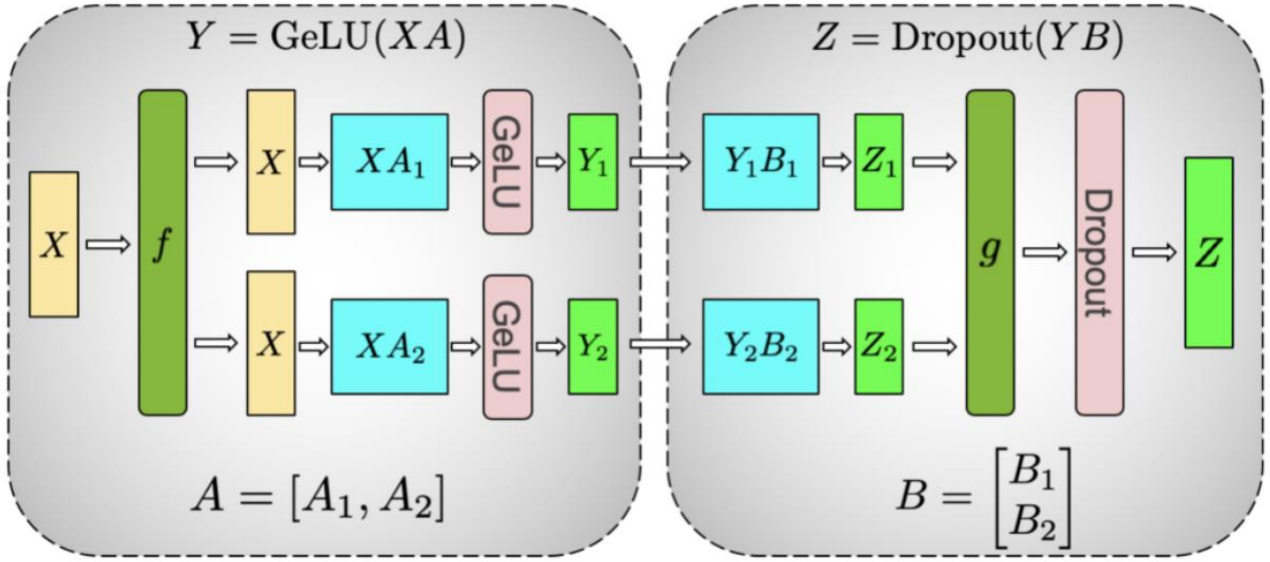
Successful in sequence modeling problems

- LLM: predict the next word. Bidirectional Encoder Representations from Transformers (BERT), Generative pre-trained transformer (GPT)
- Predict protein structure from DNA sequence (AlphaFold)
- Video/audio production

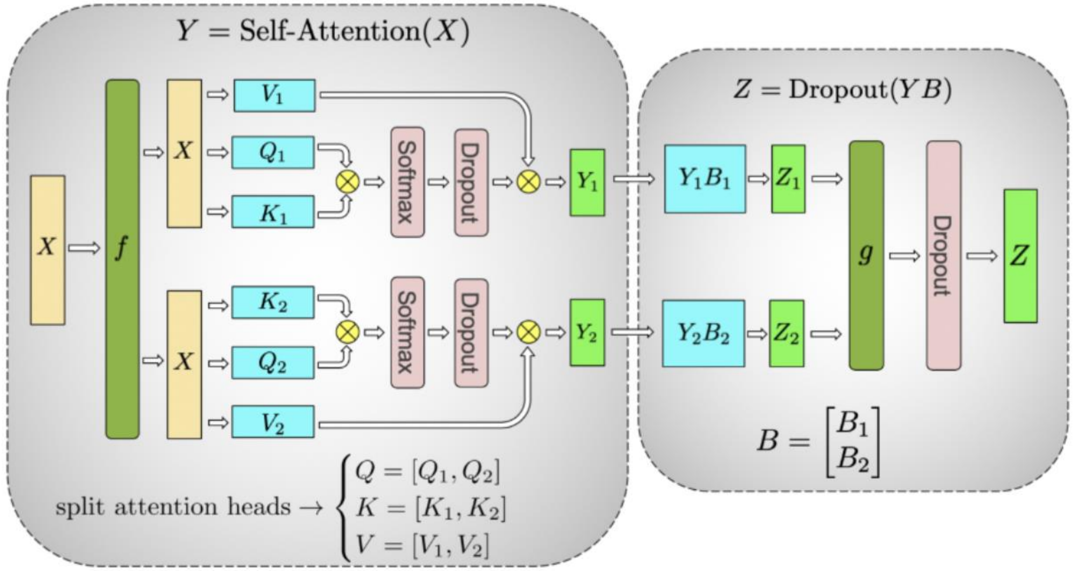
MIT course: https://www.youtube.com/watch?v=dqoEU9Ac3ek

Tensor Parallel for Transformer (1)

- A transformer block consists of a feed-forward (MLP) layer and a self-attention layer.
- Split matrices in the MLP and self-attention layers.
- The matrix multiplications in both attention and MLP happen through sharded computations.



Multilayer perceptron (MLP)

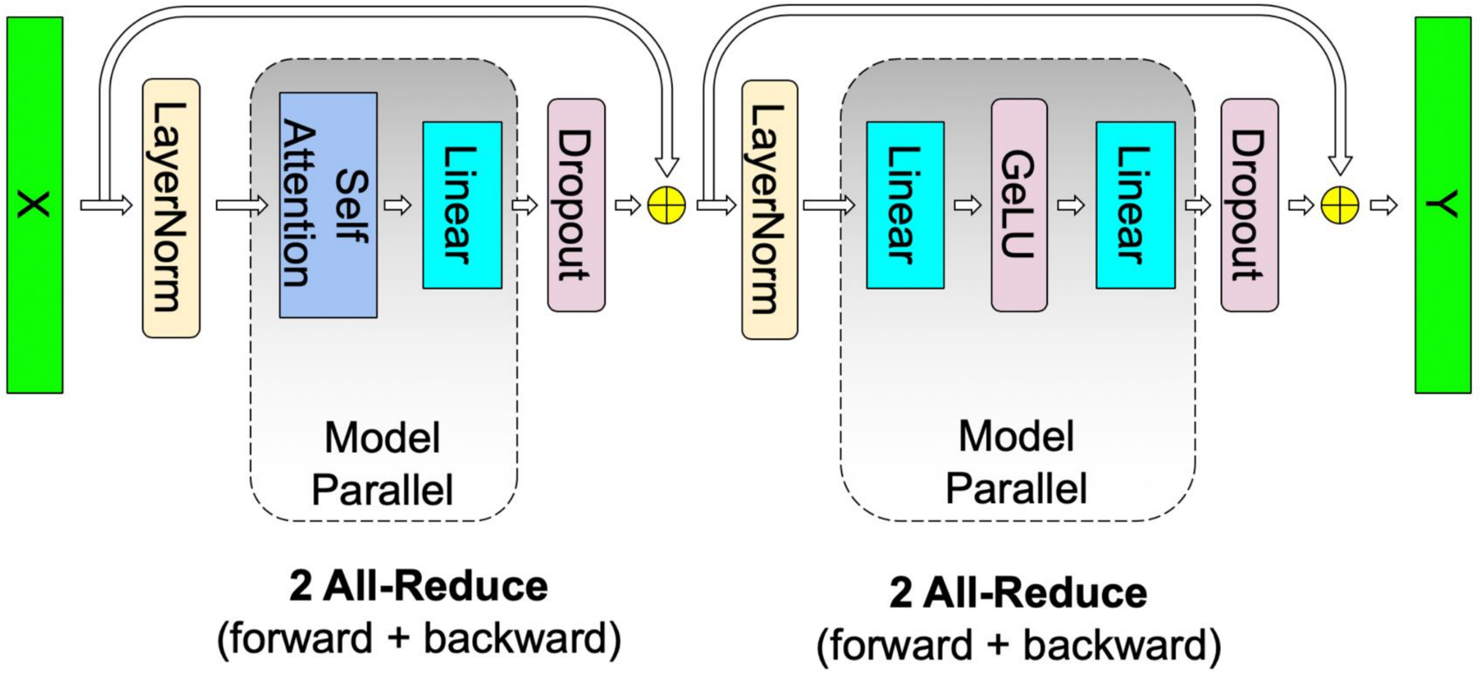


Self-attention

https://arxiv.org/abs/1909.08053

Tensor Parallel for Transformer (2)

- Minimal communication: 4 x all-reduce in the forward and backward passes of a single tensor parallel transformer layer.



- Larger communication overhead than DP or PP: more frequently.

https://arxiv.org/abs/1909.08053

Data and Tensor Parallel with PyTorch (1)

- Llama2 (Large Language Model Meta AI): built on transformer architecture.
- Hybrid parallel: Tensor Parallel within each node + Fully Sharded Data Parallel (FSDP) across nodes.
- Group GPUs for TP and DP

```
device_mesh = init_device_mesh("cuda", (dp_size, tp_size), mesh_dim_names=("dp", "tp"))
```

- Create the model and send it to GPUs

```
model = Transformer.from_model_args(simple_llama2_config).to("cuda")
```

- Set up a tensor parallel module

```
Parallelize_module(
   module=transformer_block,
   device_mesh=tp_mesh,
   parallelize_plan=layer_tp_plan
)
```

- Apply FSDP to the model

```
sharded_model = FSDP(model, device_mesh=dp_mesh, use_orig_params=True)
```

https://PyTorch.org/tutorials/intermediate/TP_tutorial.html

Data and Tensor Parallel with PyTorch (2)

- TP plan: specify how to shard feed-forward and self-attention layers, column-wise or row-wise.

```
layer_tp_plan = {
    "attention_norm": SequenceParallel(),
    "attention": PrepareModuleInput(
        input_layouts=(Shard(1), None),
        desired_input_layouts=(Replicate(), None),
    ),
    "attention.wq": ColwiseParallel(),
    "attention.wk": ColwiseParallel(),
    "attention.wv": ColwiseParallel(),
    "attention.wo": RowwiseParallel(output_layouts=Shard(1)),
    "ffn_norm": SequenceParallel(),
    "feed_forward": PrepareModuleInput(
        input_layouts=(Shard(1),),
        desired_input_layouts=(Replicate(),),
    ),
    "feed_forward.w1": ColwiseParallel(),
    "feed_forward.w2": RowwiseParallel(output_layouts=Shard(1)),
    "feed_forward.w3": ColwiseParallel(),
}
```

- Sequence parallel: a variant of TP that performs sharded computations on layer normalization.

- Communications (e.g. allreduce) happen under the hood.

https://PyTorch.org/tutorials/intermediate/TP_tutorial.html

FSDP + TP Jobs on Multiple GPUs

- Single-node Multi-GPU

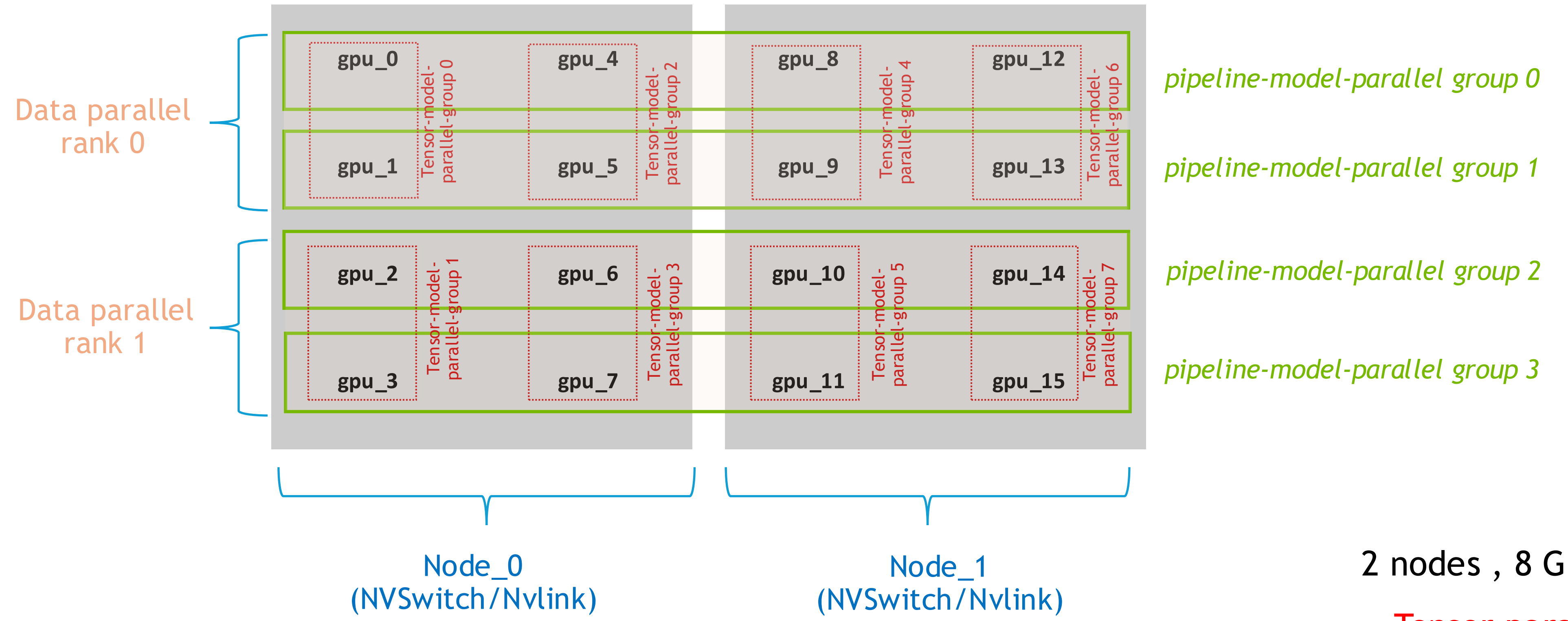
```
#!/bin/bash
#SBATCH -p mit_preemptable
#SBATCH -t 60
#SBATCH -N 1
#SBATCH -n 4
#SBATCH --mem=30GB
#SBATCH --gres=gpu:h200:4
module load miniforge/23.11.0-0
source activate ds

torchrun --nnodes=1 --nproc_per_node=4 \
        --rdzv_id=$SLURM_JOB_ID \
        --rdzv_endpoint="localhost:1234" \
        fsdp_tp_example.py
```

- Multi-node Multi-GPU: https://orcd-docs.mit.edu/recipes/torch-gpu-intermediate/

Hybrid model parallelism

GPU Affinity grouping example for PP + TP + DP



Data parallel rank 0
Data parallel rank 1
gpu_0
gpu_1
gpu_2
gpu_3
gpu_4
gpu_5
gpu_6
gpu_7
gpu_8
gpu_9
gpu_10
gpu_11
gpu_12
gpu_13
gpu_14
gpu_15
Tensor-model-parallel-group 0
Tensor-model-parallel-group 1
Tensor-model-parallel-group 2
Tensor-model-parallel-group 3
Tensor-model-parallel-group 4
Tensor-model-parallel-group 5
Tensor-model-parallel-group 6
Tensor-model-parallel-group 7
pipeline-model-parallel group 0
pipeline-model-parallel group 1
pipeline-model-parallel group 2
pipeline-model-parallel group 3
Node_0 (NVSwitch/Nvlink)
Node_1 (NVSwitch/Nvlink)

2 nodes , 8 GPUs per node

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

- Communication overhead: PP < DP < TP
- Network: fast Nvlinks within a node, Infiniband across nodes

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

Which Strategy To Use When

❑ Single-node Multi-GPU

- The model fits into a single GPU: DP (distributed DP)
- The model doesn’t fit into a single GPU: PP, TP, ZeRO/FSDP, PP + DP, or TP + DP
- The largest layer does not fit into a single GPU: TP or ZeRO/FSDP.

❑ Multi-node Multi-GPU

- ZeRO/FSDP (easy)
- PP + TP + DP (tricky but faster)

➢ Best to experiment to find the winner on the computing platform.

https://huggingface.co/docs/transformers/v4.13.0/en/parallelism

What is more?

- Hybrid model parallelism: PP + TP + DP
- Mixture of experts
- Distributed inference