

# Parallel and Distributed Deep Learning

*Shaohao Chen*

*ORCD at MIT*

# Outline

- Basics of deep learning
- Deep learning on a single GPU
- Distributed deep learning on multiple GPUs
  - Data parallelism
  - Model parallelism
    - Pipeline parallelism
    - Tensor parallelism

# Machine learning and deep learning

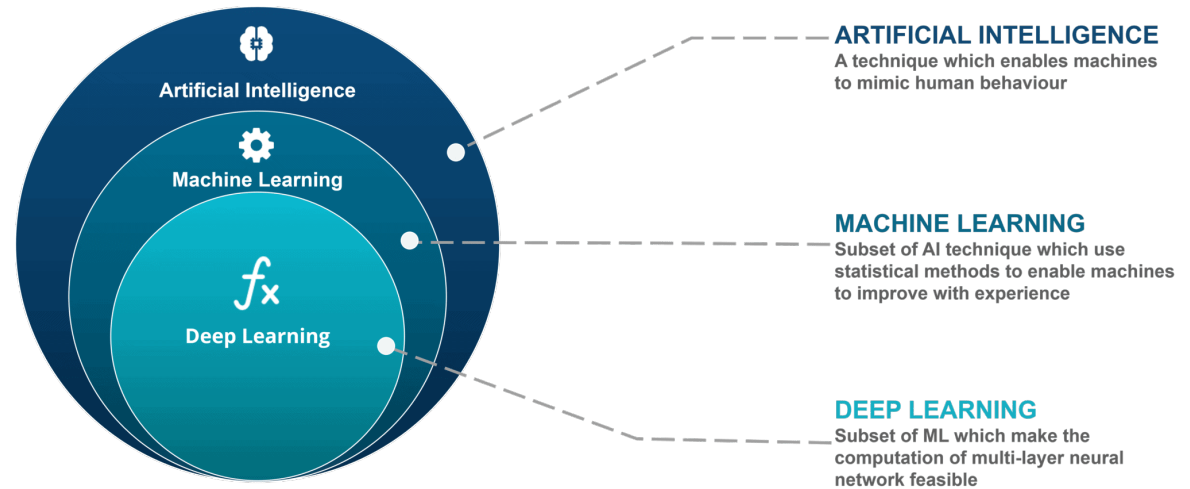
- Artificial intelligence

- Machine learning

Statistical algorithms

Learn from data

- Supervised learning: classification, regression
- Unsupervised learning: clustering



- 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

# Access to ORCD clusters

- Get started

<https://orcd-docs.mit.edu/getting-started/>

- Log in Engaging

```
ssh <user>@eofe10.mit.edu
```

- Get an interactive session and set up environment

```
srun -t 120 -n 4 --gres=gpu:h100:2 -p mit_normal_gpu --pty bash  
module load miniforge/23.11.0-0
```

# Install PyTorch and Deepspeed

- Install PyTorch, Deepspeed, and dependencies.

```
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
```

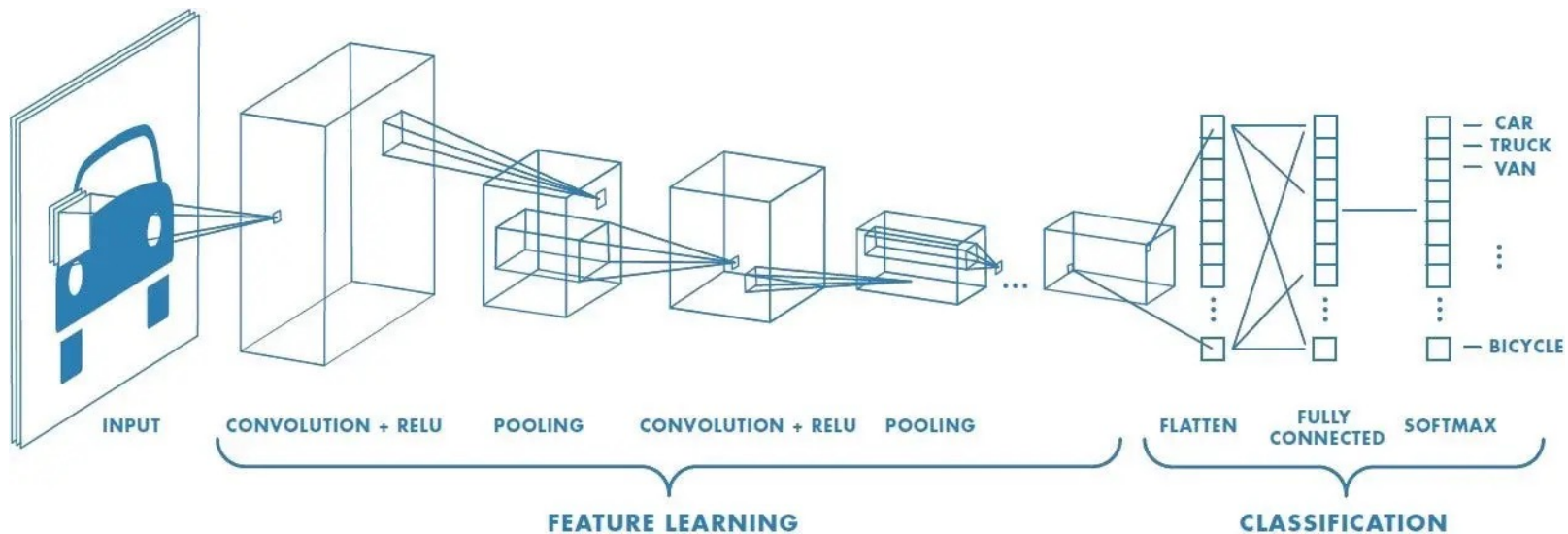
```
pip install deepspeed
```

```
pip install datasets tensorboard transformers
```

```
pip install fire loguru sh matplotlib
```

# Convolutional Neural Network (CNN)

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



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

# Train a neural network

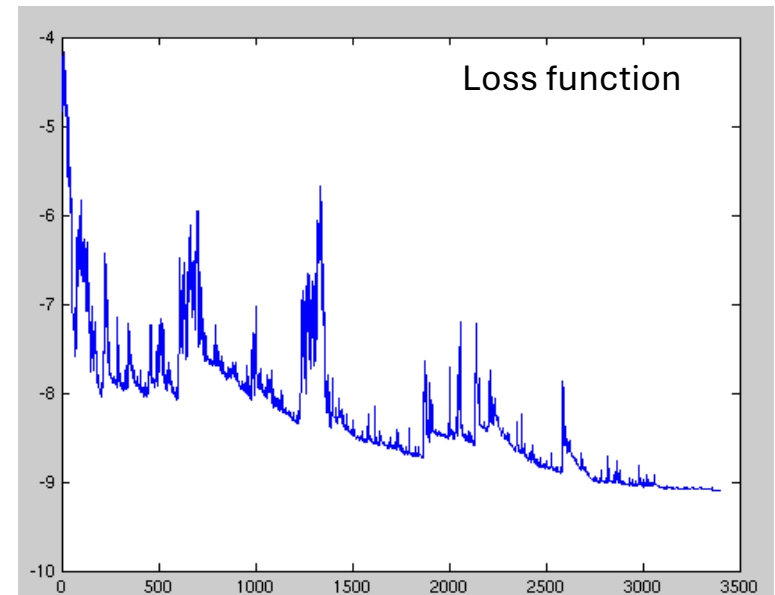
- **Training:** adjust the model to minimize a loss function.
- **Loss function:** cross entropy

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

- **Optimizer:** Stochastic Gradient Descent (SGD)
- **Training data:** batch (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 of the network, 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](https://en.wikipedia.org/wiki/Stochastic_gradient_descent)

# Run 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)
```

- Send a batch of data to the GPU at every step

```
inputs, labels = data[0].to(device), data[1].to(device)
```

- Run the program

```
source activate ds  
python cnn_cifar10_gpu.py
```



## 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 can be accelerated on GPUs.

# Distributed Parallelism in Deep Learning

- Distributed on multiple GPUs.

- Data Parallelism

Each GPU gets a different batch of data

Process more data at the same time period.

Universal to different models. The model must fit within GPU memory.

- Model Parallelism

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

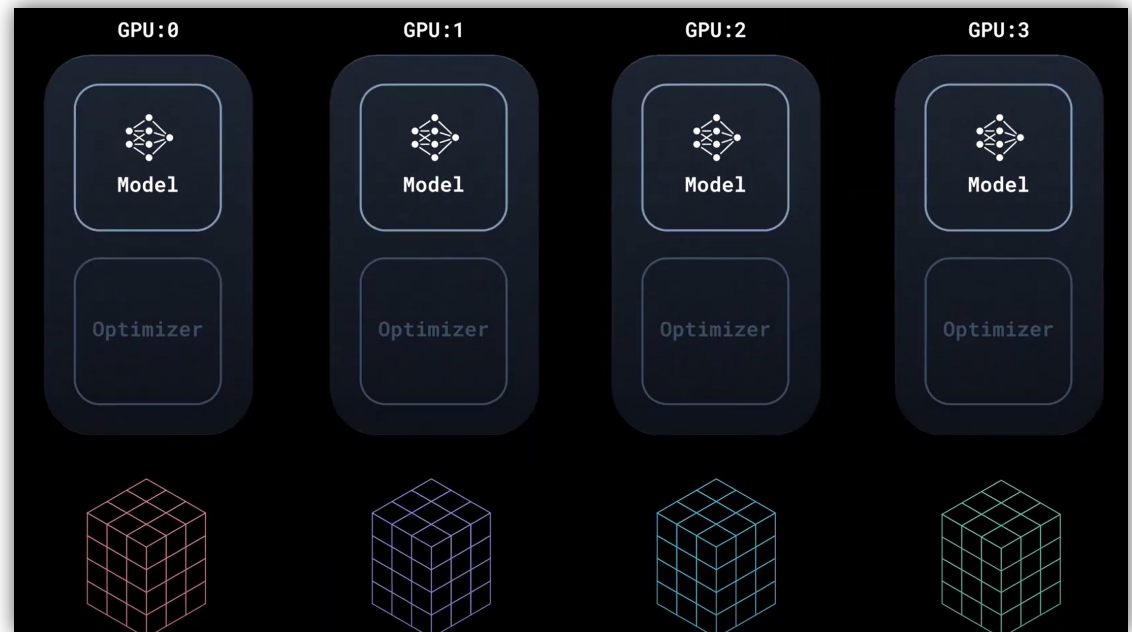
Split the model on multiple GPUs.

Tricky to design and implement.

# Data Parallelism

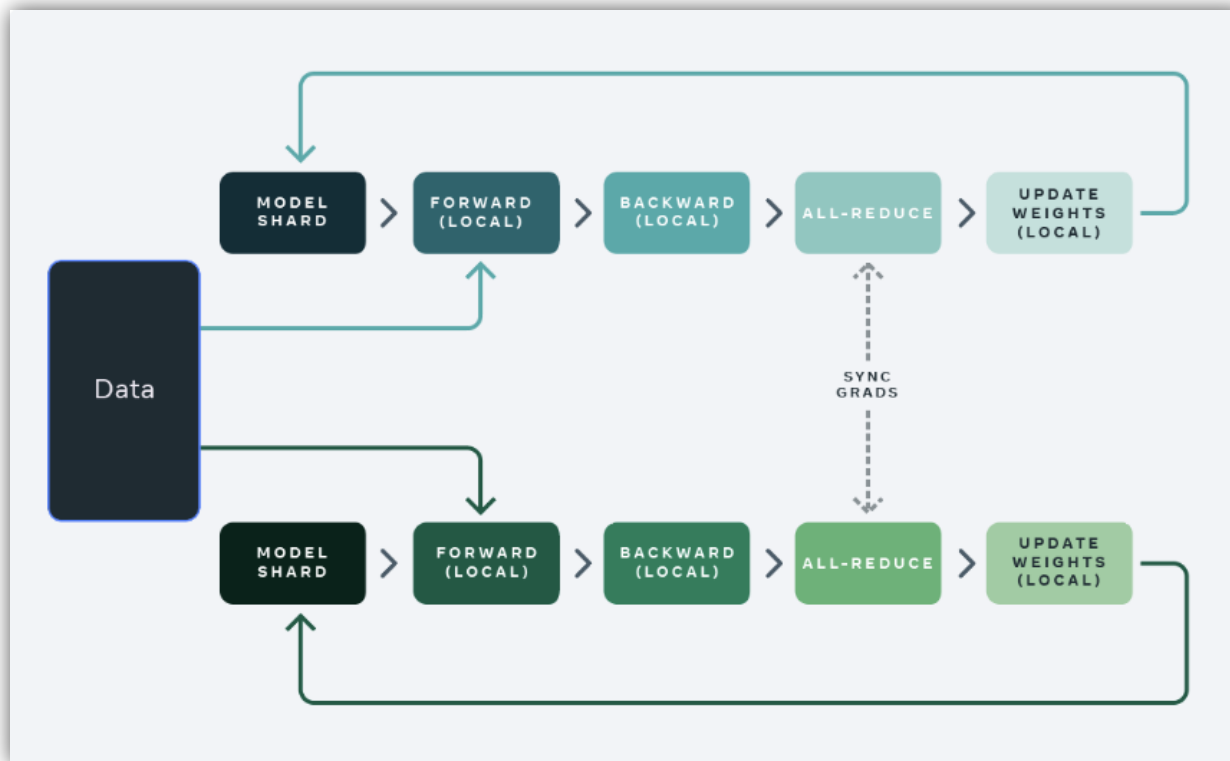
- **Data Parallelism (DP)** - the same model is replicated multiple times, and each being fed a slice of the data. The processing is done in parallel and all models are synchronized at the end of each training step.

- Each GPU has a copy of the model
- Optimizer states and random seeds are the same
- Each GPU gets a different batch of data
- Concurrently processing 4x the data of a single GPU
- Data sampling is handled by a Distributed Sampler



Data Parallel training at large scale may affect model quality

# Communication and Synchronization



- Each model is different because it is trained on different batches of data.
- Gradients from each GPU are synchronized before the update step, ensuring model replicas stay consistent across iterations.
- Synchronization is done with a bucketed Ring-AllReduce algorithm. Each GPU gets the averaged gradient.
- The algorithm overlaps gradient computation with communication so GPU is utilized efficiently.

Scaling with data parallel introduces communication overhead when syncing gradients

# Distributed Data Parallel with Pytorch

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

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

- Set up GPU ID
- Apply DDP
- Spawn training processes on multiple GPUs

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

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

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

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

- Data communication is executed by DDP under the hood.

[Ref\[1\]](#), [Ref\[2\]](#)

# Why big models?

- Transformer architecture
  - Removed the sequential processing dependency of RNNs, such as Long Short-term Memory (LSTM).
  - Enabled language models to be trained with parallelism
- A dramatic increase in model sizes after the birth of Transformer.

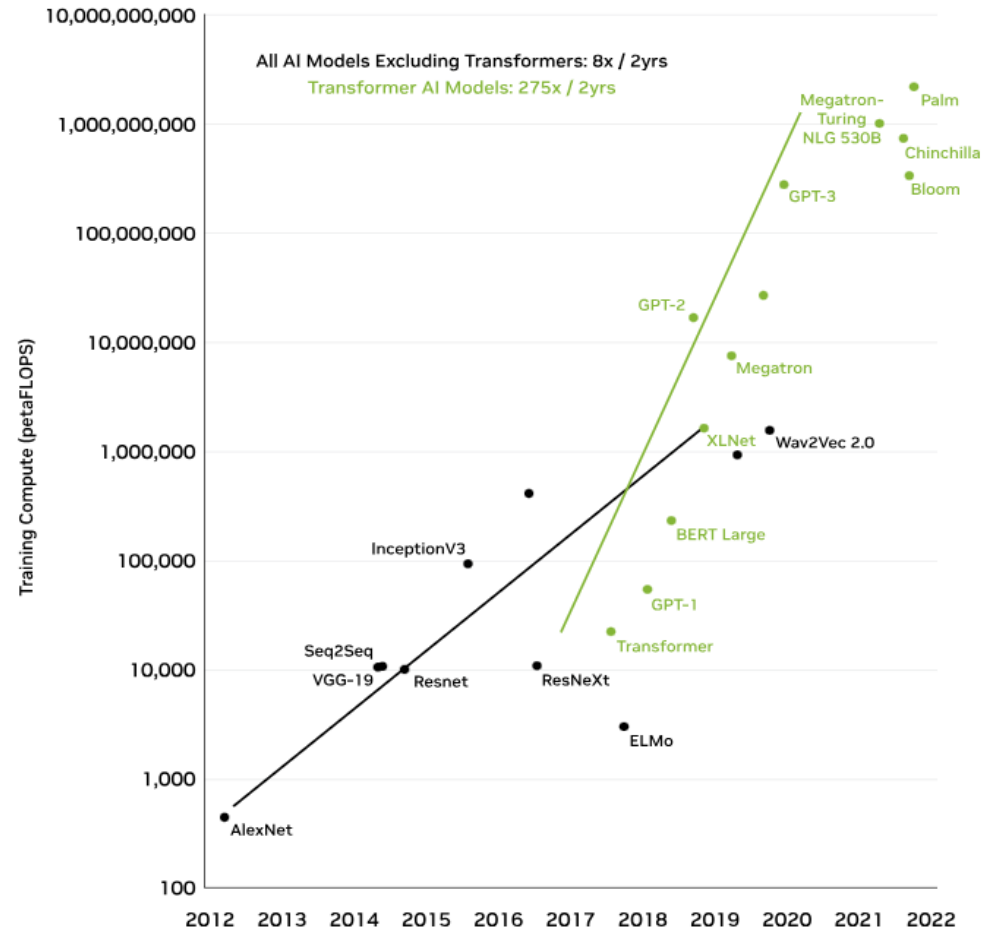


Figure 3. Compute required for training transformer models.

## 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

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

## 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

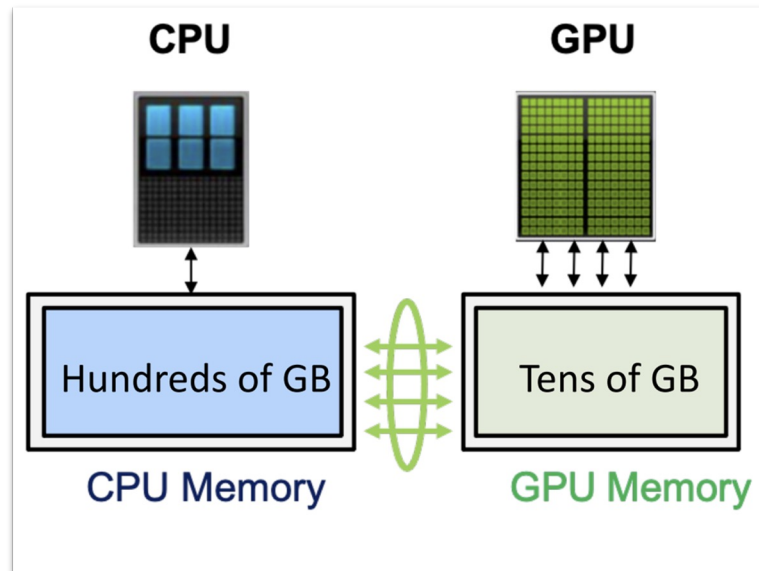
~65 weeks on 1 x DGX A100  
~16 weeks on 4 x DGX A100

~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



## Offloading to the host



Offload CPU tensors not used in computation from GPU to CPU

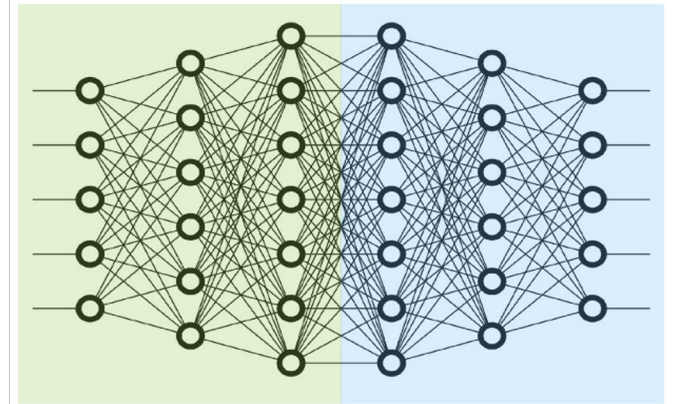
- Training times will be slower due to slow data movement.
- Overlap communication with computation.

# Model Parallelism

- Pipeline (Inter-Layer) Parallelism

Split contiguous sets of layers across multiple GPUs

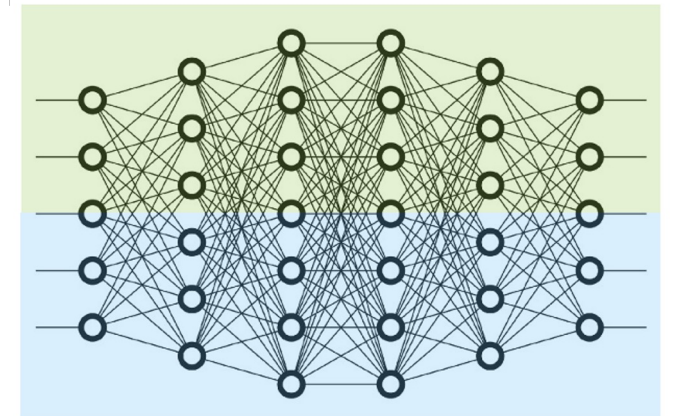
Layers 0,1,2 and layers 3,4,5 are on different GPUs



- Tensor (Intra-Layer) Parallelism

Split individual layers across multiple GPUs

GPUs compute different parts of Layers 0,1,2,3,4,5



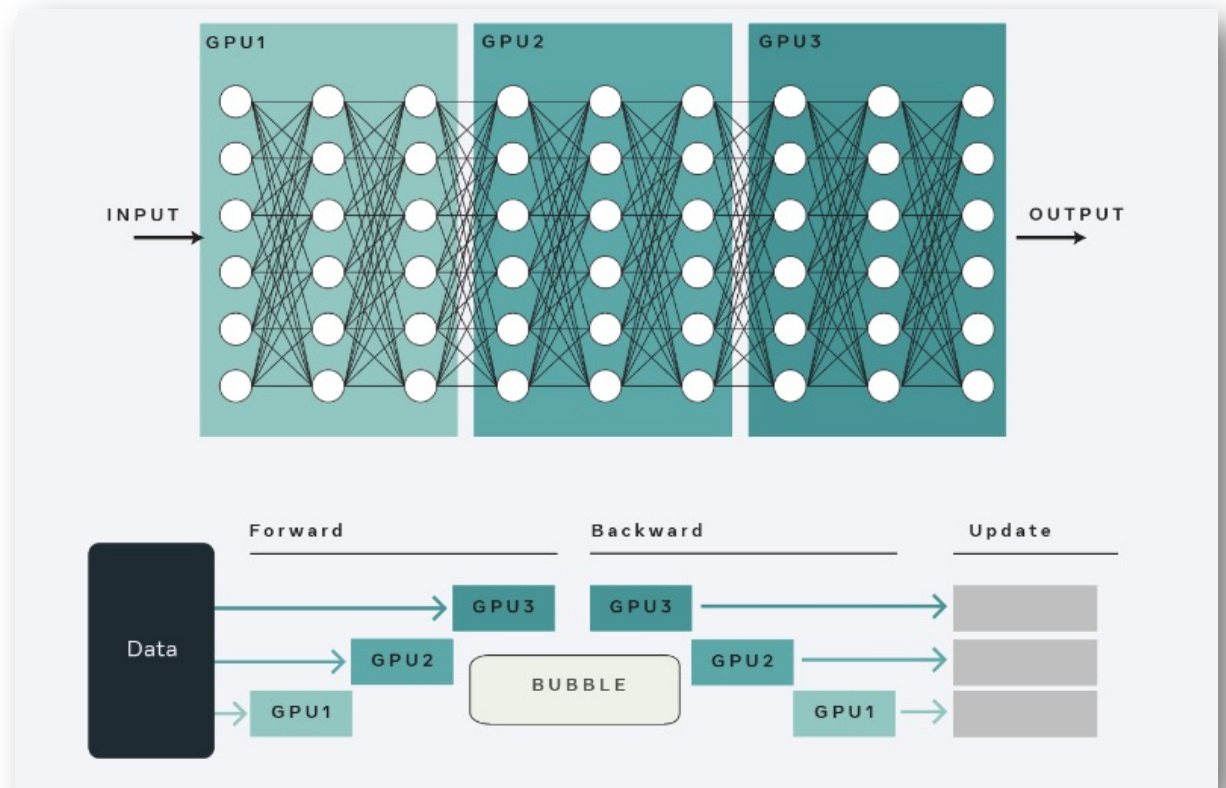
# Model Parallelism

- **Pipeline Parallel (PP)** - The model is split vertically (layer-level) across multiple GPUs, so that 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 Parallel (TP)** - Each tensor is split into multiple shards, and each shard of the tensor resides on its designated GPU. Each shard gets processed in parallel on different GPUs and the results are synced at the end of the step. This is what one may call horizontal parallelism.
- **Zero Redundancy Optimizer (ZeRO)** (also named **Sharded DDP**): Also performs sharding of the tensors somewhat similar to TP, except the whole tensor gets reconstructed in time for a forward or backward computation, therefore the model doesn't need to be modified. It also supports various offloading techniques to compensate for limited GPU memory.

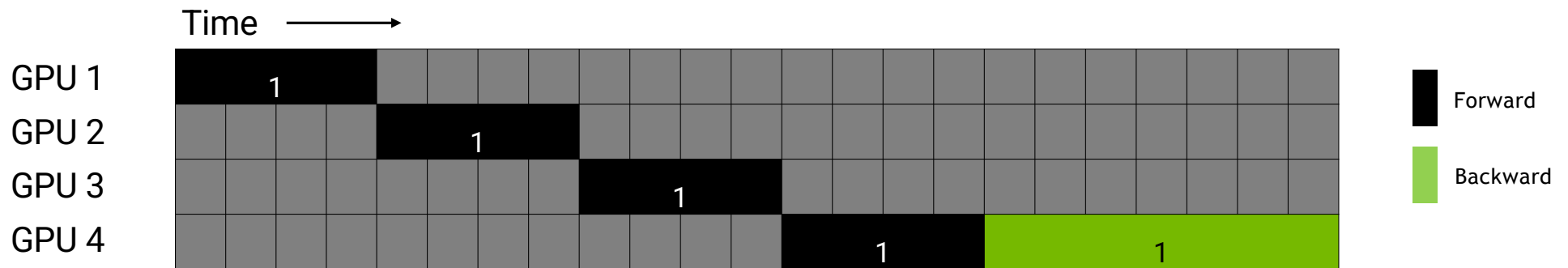
Ref: <https://huggingface.co/docs/transformers/v4.13.0/en/parallelism>

# Pipeline Parallelism

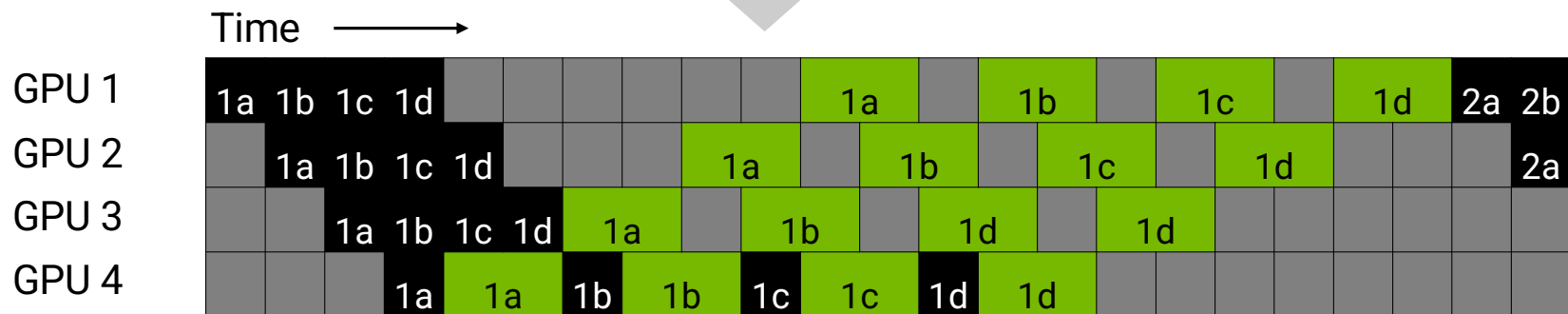
- Naive pipeline parallel is sequentially processed.
- Leads to GPU underutilization.



# Pipeline Parallelism

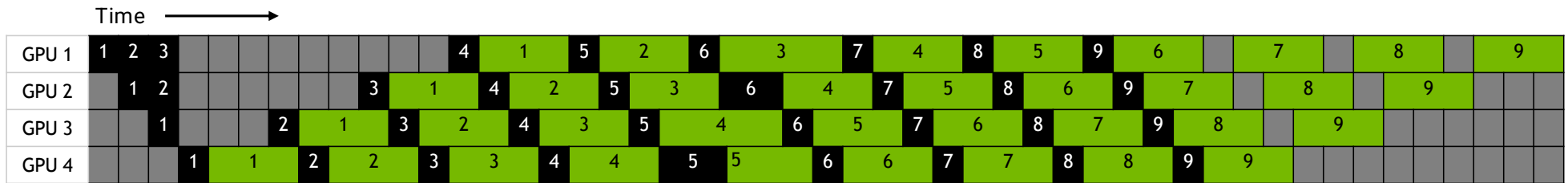


Split batch into micro batches and pipeline executions



# Pipeline Parallelism

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

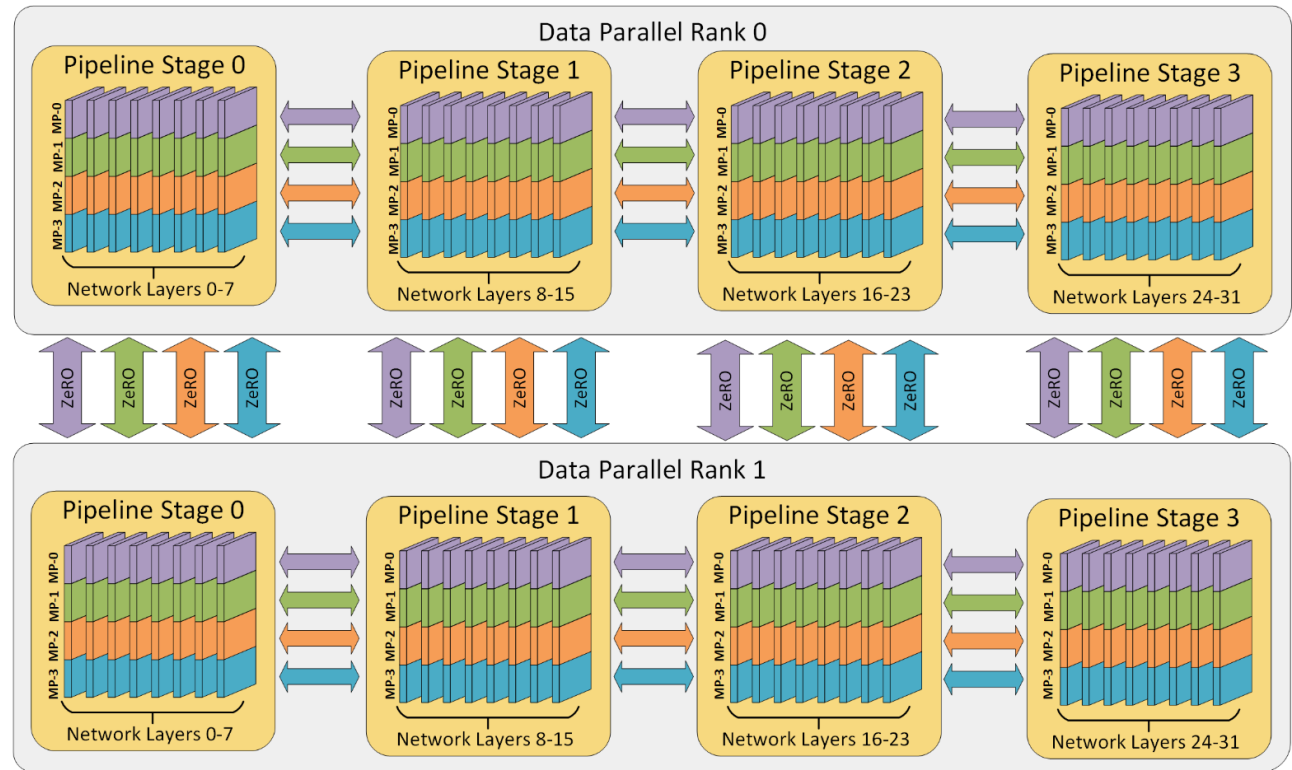
$t_b$  : backward step time

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

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

# Data and Pipeline Parallel with Deepspeed

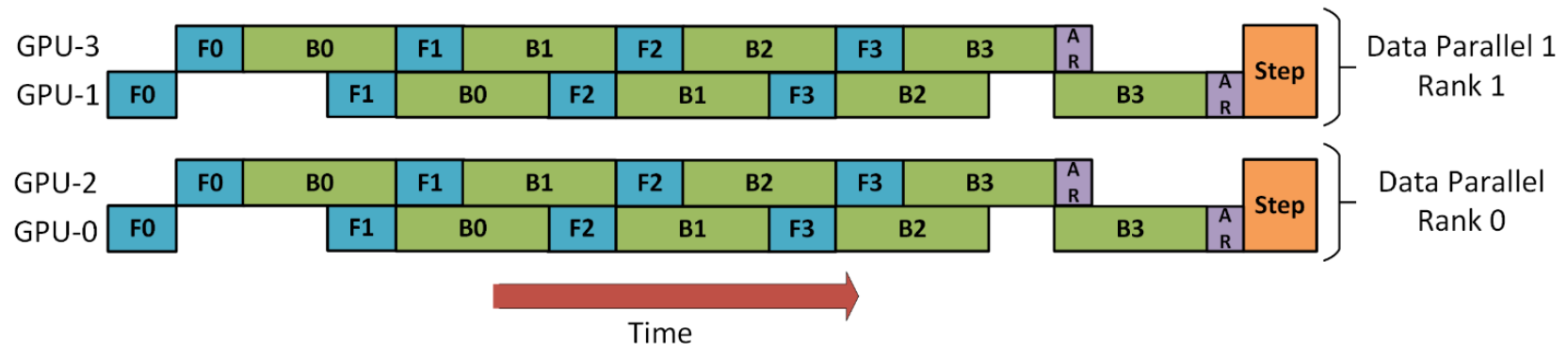
- Hybrid parallelism
- two-way data parallel
- two-stage pipeline parallel



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

# Data and Pipeline Parallel with Deepspeed

- Four GPUs, 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.
- Once a stage completes the forward pass for a micro-batch, the activation memory is communicated to the next stage in the pipeline.
- As the next stage completes its backward pass on a micro-batch, the gradient with respect to the activation is communicated backward through the pipeline.
- 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.





## 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 in Deepspeed

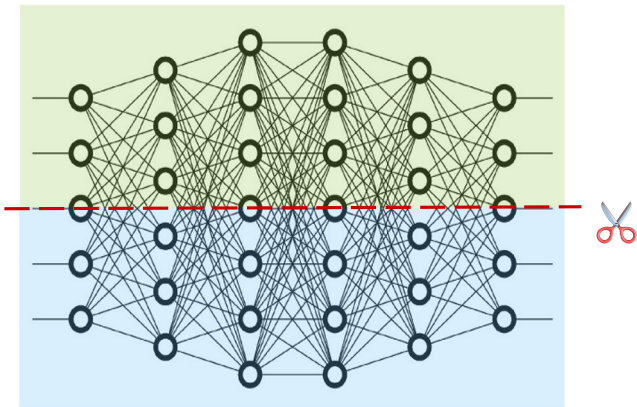
```
net = PipelineModule(layers=join_layers(net),  
                    loss_fn=torch.nn.CrossEntropyLoss(),  
                    num_stages=args.pipeline_parallel_size,  
                    partition_method=parameters,  
                    activation_checkpoint_interval=0)
```

- 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

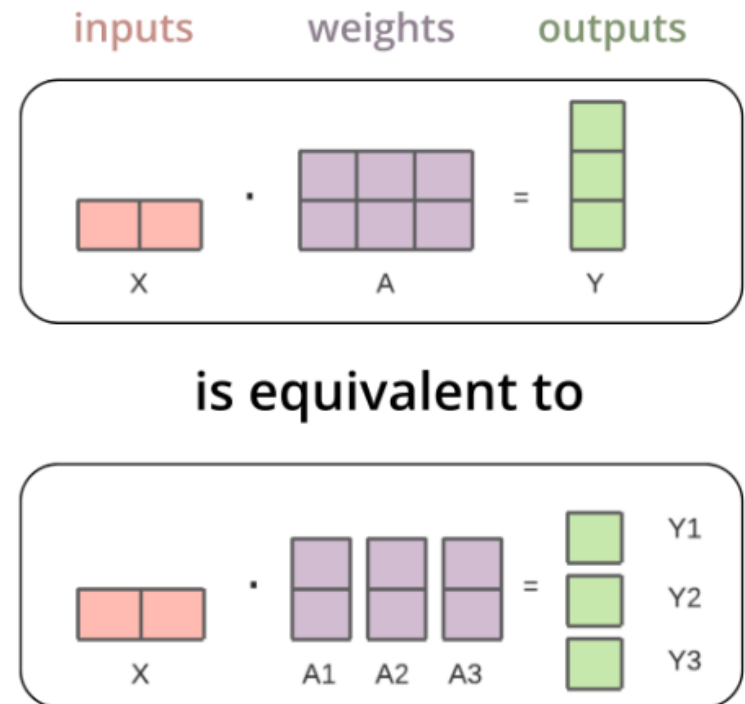


- Use to scale beyond data parallelism
  - Typically used when the model can't fit on a single GPU
  - Reduces memory proportional to the number of workers (model dependent)
  - Requires high compute-bandwidth to overcome communication overhead
  - Less restrictive on the batch size (avoids bubble issue in pipelining)
  - Tensor parallelism works well for large matrices (e.g. Transformers)
  - Does not scale well beyond the node boundary
- 
- The implementation of TP depends on the neural network architecture.

[Ref: Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, A Large-Scale Generative Language Model](#)

## A simple example of tensor parallelism

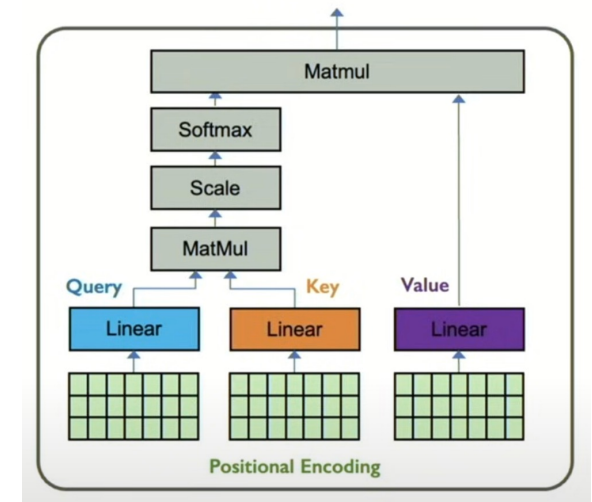
- When multiplying the input tensors with the first weight tensor, the matrix multiplication is equivalent to splitting the weight tensor column-wise, multiplying each column with the input separately, and then concatenating the separate outputs.
- The outputs are then transferred from the GPUs and concatenated together to get the final result.



# Transformer architecture

## *Attention is all you need*

- Positional embedding. A word → A vector in a high-dimensional space.
- Extract **Query**, **Key**, and **Value** for search.
- 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.



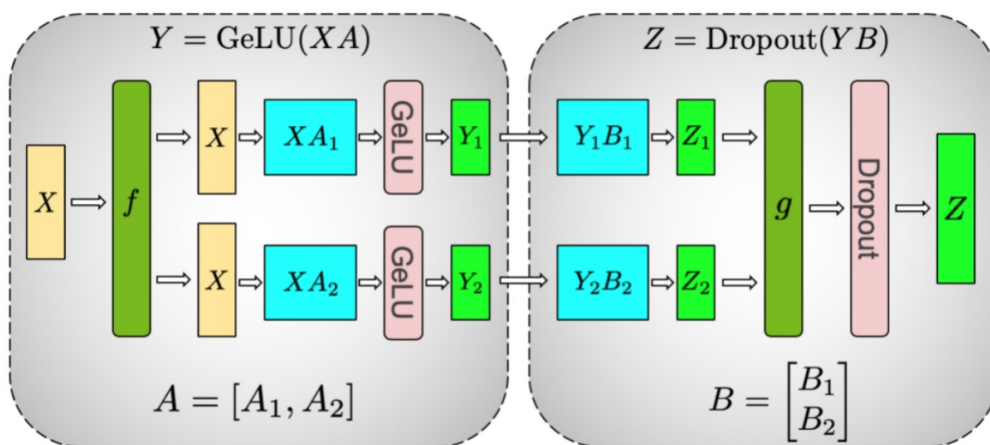
## Good at sequence modeling problems

- 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

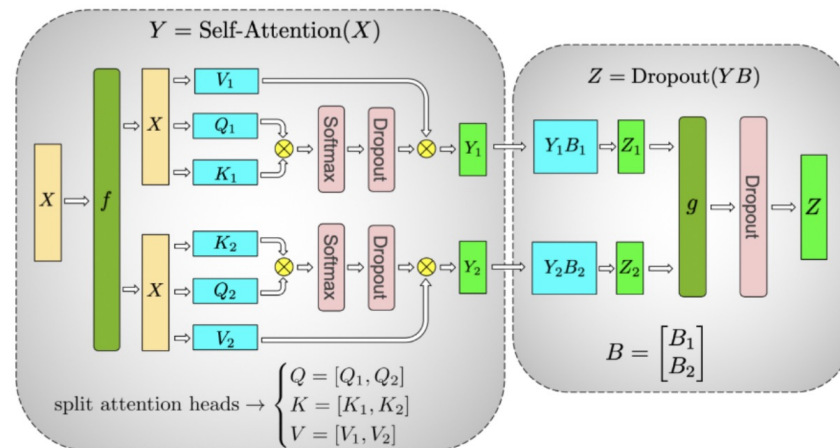
<https://www.youtube.com/watch?v=dqoEU9Ac3ek>

# Tensor Parallel for Transformer Model

- [Llama2](#) (Large Language Model Meta AI): built on transformer architecture.
- A transformer block consists of a **feed-forward (MLP) layer** and a **self-attention layer**.
- Split MLP and self-attention layers into multiple shards.
- The matrix multiplications in both attention and MLP happen through sharded computations.



Multilayer perceptron (MLP)



Self-attention

# Tensor Parallel with Pytorch

- Set up a tensor parallel module in Pytorch

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

Specify how to shard feed-forward and self-attention layers, **column-wise** or **row-wise**.

**Communications** (allreduce) will happen under the hood.

- **Sequence parallel:** a variant of Tensor Parallel that performs sharded computations on layer normalization.

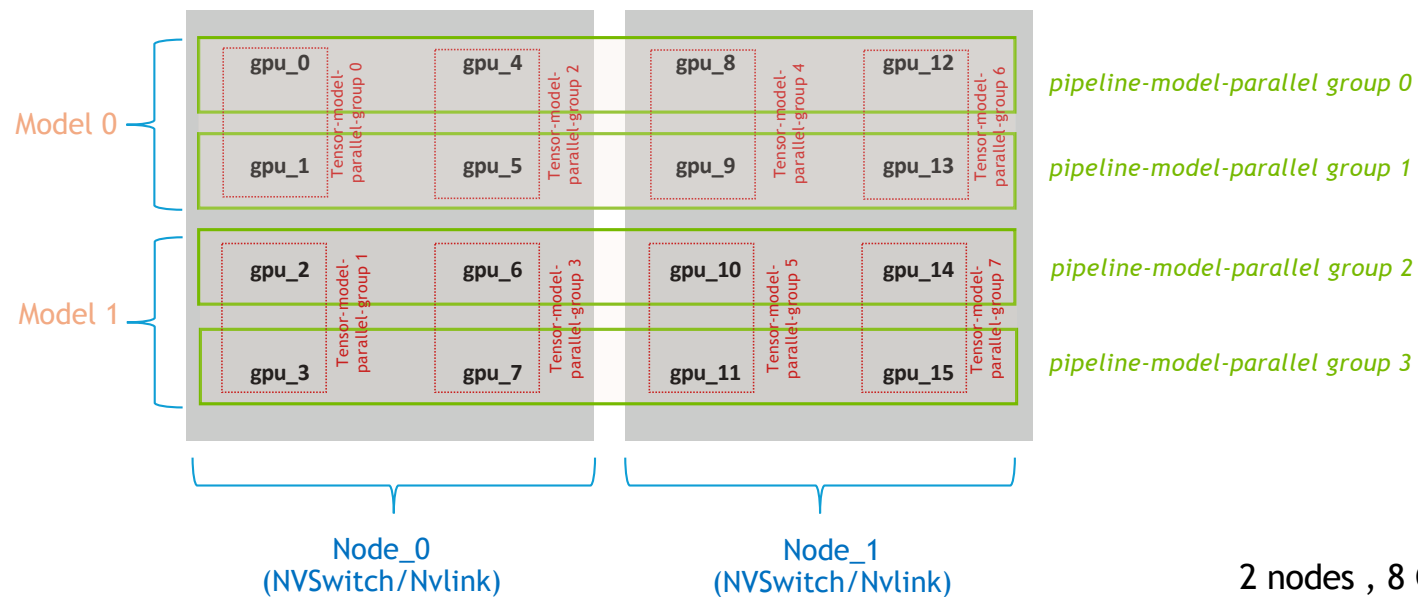
Keeps sharded activations in the forward pass and sharded gradients in the backward pass.

- Apply Tensor Parallel within each node (intra-node) and apply **Fully Sharded Data Parallel (FSDP)** across nodes (inter-nodes).

[https://pytorch.org/tutorials/intermediate/TP\\_tutorial.html](https://pytorch.org/tutorials/intermediate/TP_tutorial.html)

# Hybrid model parallelism

GPU Affinity grouping example for PP + TP + DP



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

2 nodes , 8 GPUs per node

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

# Which Strategy To Use When

## ❑ Single-node Multi-GPU

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

## ❑ Multi-node Multi-GPU

- PP + DP
- TP + DP (or FSDP)
- PP + TP + DP

➤ Best to experiment to find the winner on your particular setup.

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



## What is not covered ...

- Zero Redundancy Optimizer (ZeRO)
- Fully Sharded Data Parallel (FSDP)
- Sequence parallelism
- Hybrid model parallelism: PP + TP + DP
- Activation Checkpointing
- Save memory by offloading to CPU
- Mixed precision training
- Distributed inference