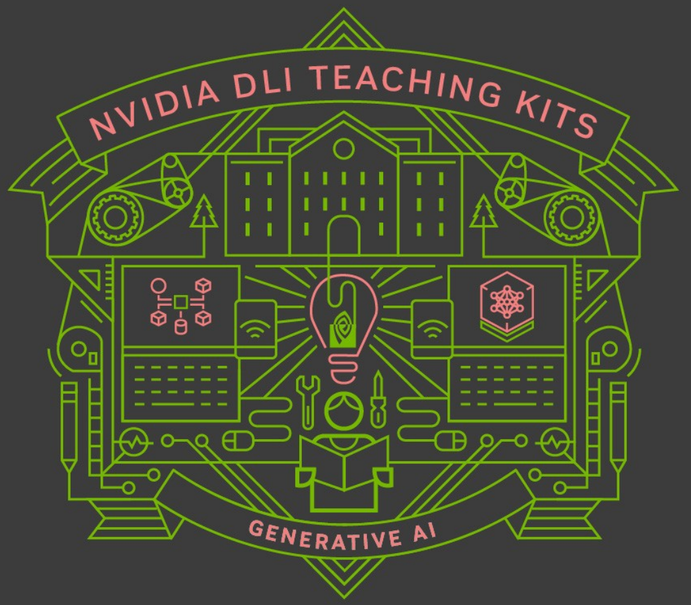
Lecture 9.1 - Distributed Training with Data and Model Parallelism Strategies

Generative AI Teaching Kit



This lecture

- Introduction to Parallelism: Data Parallelism
- Introduction to Parallelism: Model Parallelism
- Single Node Multi GPU Parallelism
- Multi Node Multi GPU Parallelism

Fundamentals of Data Parallelism

What is Parallelism? Why do we need it?

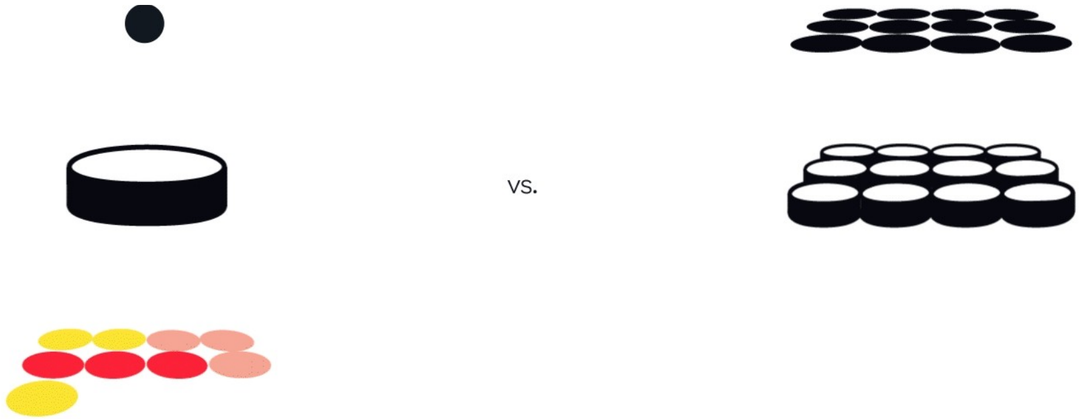
Until now we have largely assumed that we have access to a GPU with a limitless amount of memory to hold a model, and that we are happy to wait as long as it takes to train. This is not reality.

GPUs have limited:

- Memory within which to hold models and data to train on.
- Processing speed which means training takes time!

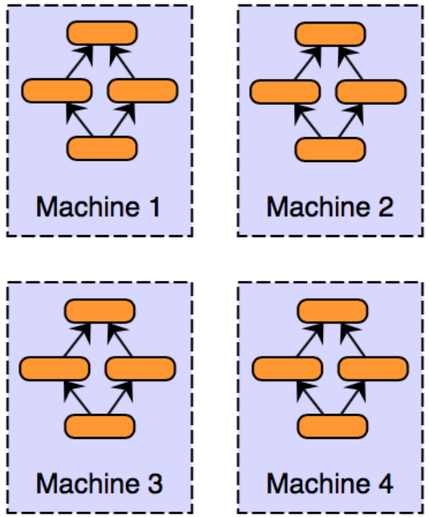
Time to train matters, we need to ***iterate*** to do science and discovery.

Parallelism is the idea that we can split the tasks we need to complete and complete them on separate computing units simultaneously to solve these limitations.



Parallelism in GenAI

There are two main families of parallelism when it comes to model training and inference:

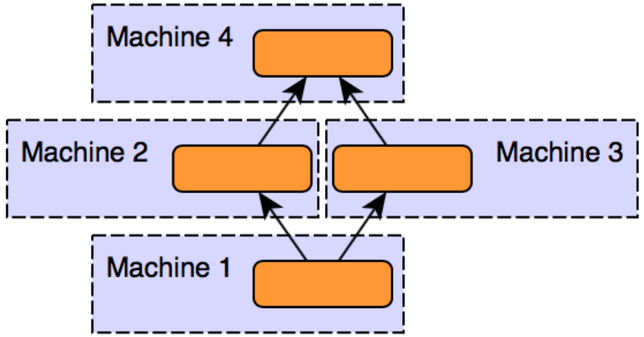


Data Parallelism

- The training data is split into equal pieces and sent to different GPUs
- A replica of the model is on each GPU and they update in parallel to stay synchronized
- **Trains a model faster**

Model Parallelism

- Models that are too big to fit on one GPU are split across GPUs
- A single training batch is sent through all GPUs to be processed
- **Trains very large models**



Training faster with Data Parallelism

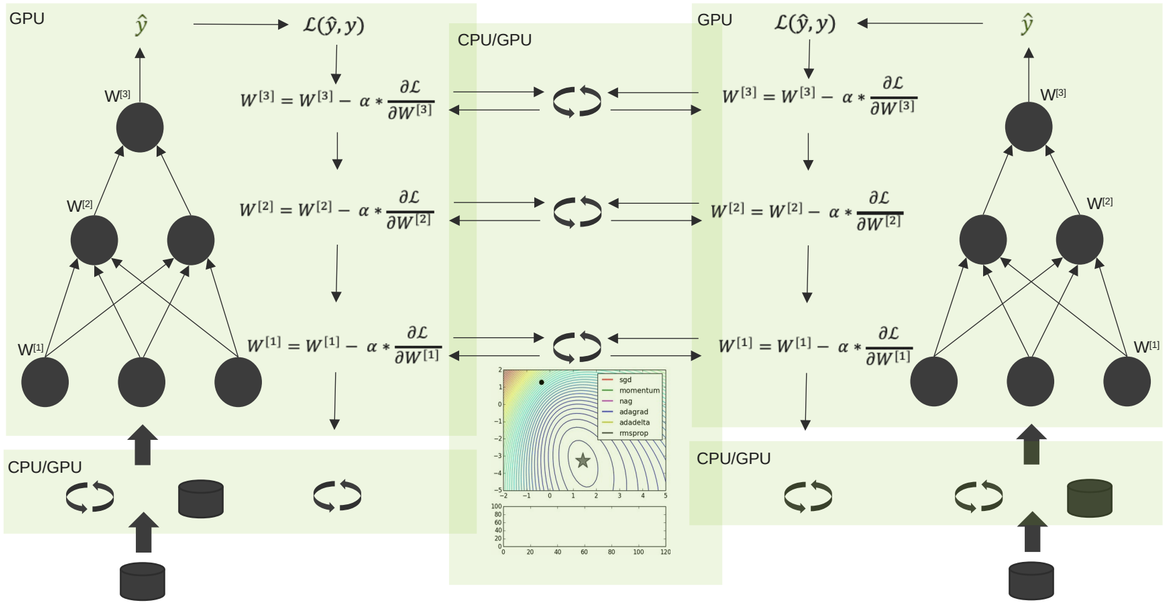
How does data parallelism work?

Recap how we train with backpropagation:

- A forward pass is made through the network
- At the output a loss is calculated based on how correct the model's prediction was to true values
- The loss is used to update the weights of the network by using the gradients with respect to the loss value.

Data parallelism does the exact same thing, **except**:

- At the end of the forward pass, all of the gradients for each copy of the model across the GPUs are averaged together and used to update the different model copies.
- This keeps the models in sync so they are all identical for the forward passes.
- You can think of this as averaging the training data to cycle through each epoch faster.



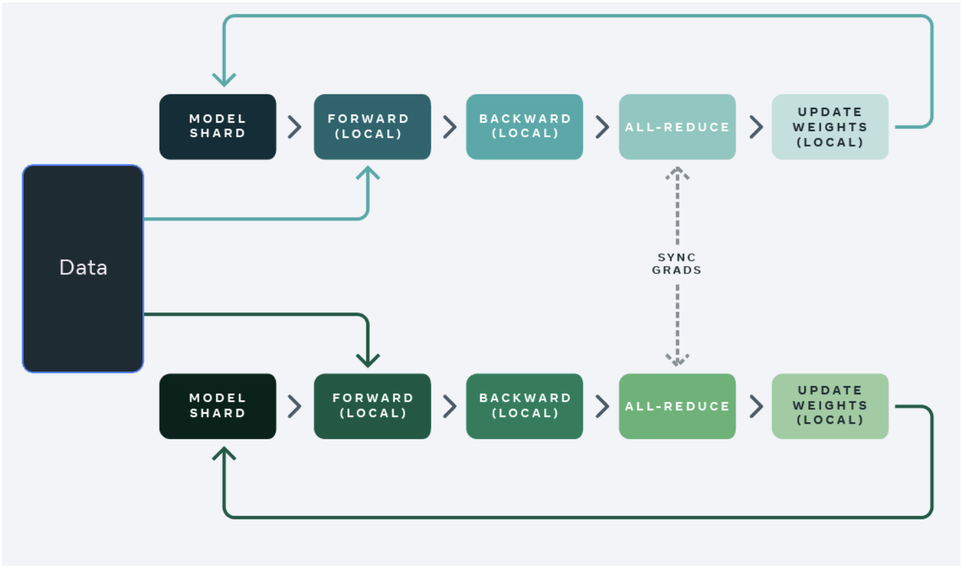
The Data Parallelism Algorithm

Before training loop

1. Setup a model with randomized weights
2. Split the dataset into subsets to be trained on into equal amounts based on the numbers of GPUs available
3. Copy the model onto each of the GPU devices

During training loop

4. Load baches of each dataset subset onto each GPU
5. Run a regular forward pass of each model on each GPU
6. Calculate loss and gradients for each model on each GPU
7. Communicate and aggregate gradients across all GPUs
8. Update each model with the aggregated (averaged) gradient values
9. Continue until stopping criterion



After training loop:

Since all copies of the model are kept identical, any one of the distributed models can be used as the final model

- Transfer and store weights of the final model back to the CPU and storage

Learning Rate and Batch Size Considerations

One important consideration with data parallelism is the batch size.

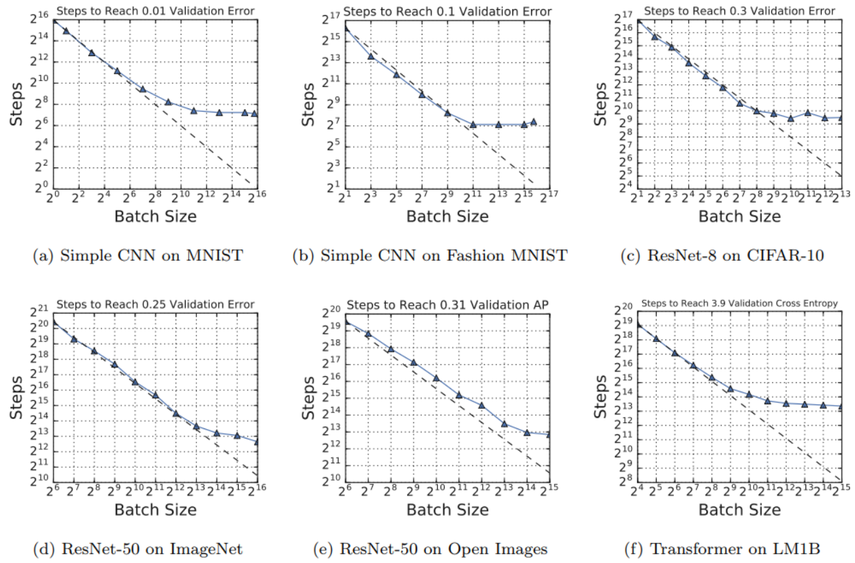
Since multiple batches are sent to identical copies of the model at the same time, and the model updates using the average effect, data parallelism effectively uses a larger batch size by default.

Global batch size = mini batch size x no. model instances

The mini batch size is what would be the regular batch size if we weren't doing data parallelism.

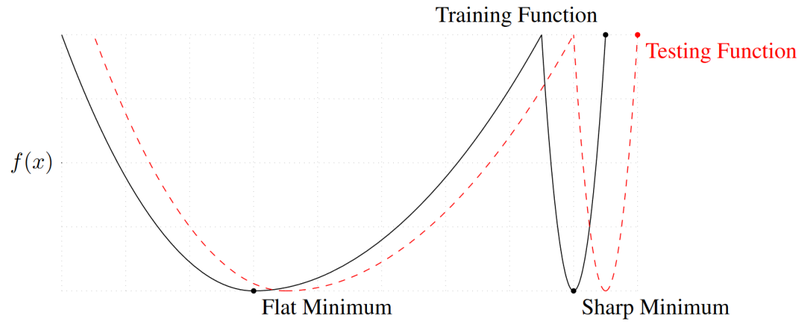
Why this matters?

- Larger global batch sizes decrease training iteration time
- Larger global batch sizes can mean training is unstable (sharp minima etc.)



(a) Simple CNN on MNIST (b) Simple CNN on Fashion MNIST (c) ResNet-8 on CIFAR-10

(d) ResNet-50 on ImageNet (e) ResNet-50 on Open Images (f) Transformer on LM1B

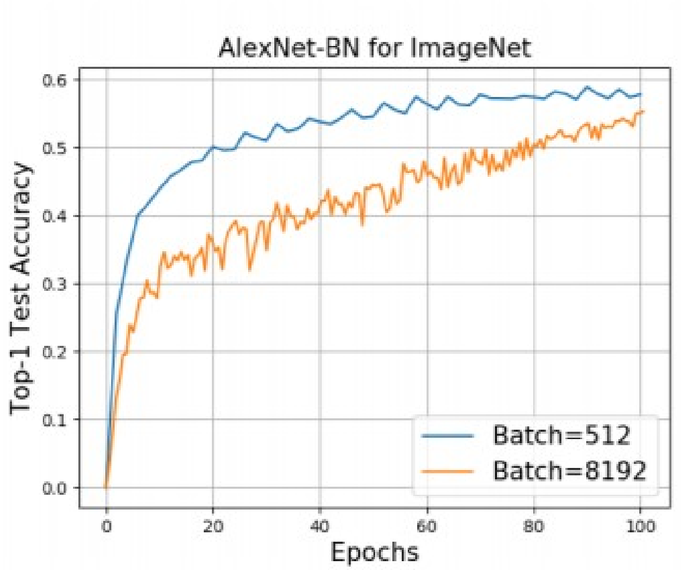


Learning Rate and Batch Size Considerations

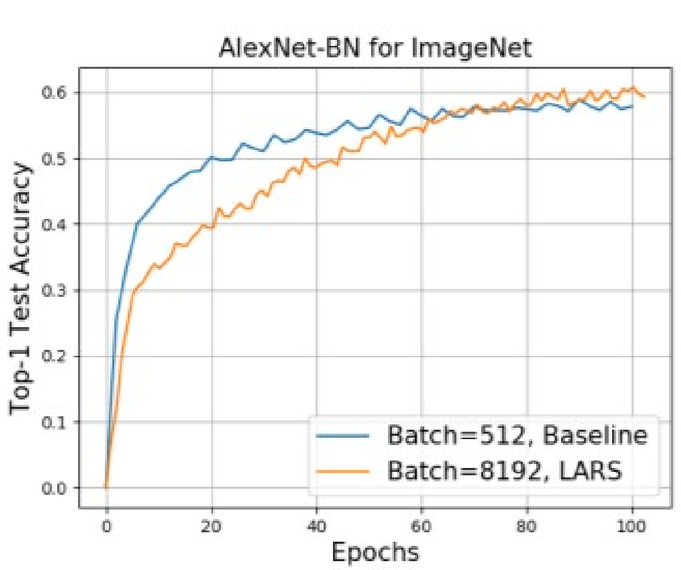
Large batch sizes resulting from naïve implementations of data parallelism degrade training performance.

Solutions to this include:

- Learning Rate Manipulation:
  - Recall that the optimizer is run each micro batch and uses a learning rate value to scale the change in weights
  - With a large batch size, a smaller learning rate, proportional to the batch size increase often yields good results
- Batch Normalization
  - This reduces the effect of variances in the different batches by requiring the batch values to be statistically normalized
- More Advanced Optimizers
  - Layer-adaptive learning rate optimizers
  - Adam/momentum-based optimizers



(a) Training without LARS



(b) Training with LARS

FSDP: Going extreme

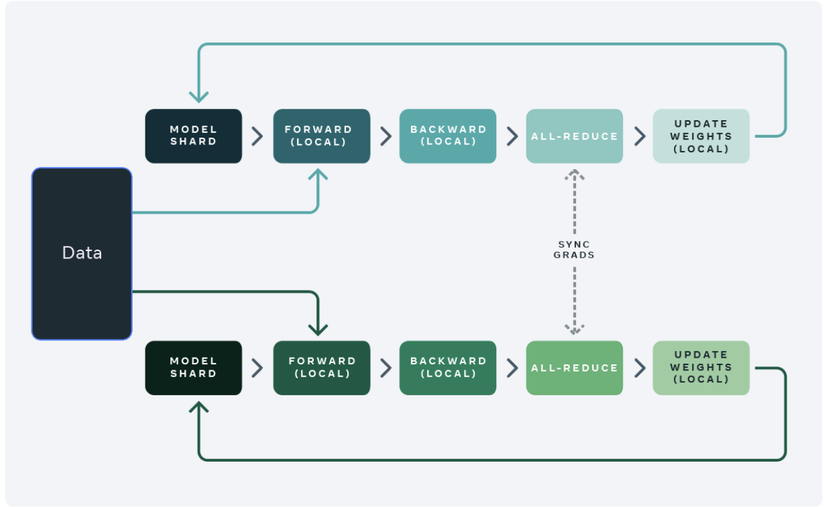
If the model is also too large to fit on a single GPU, meaning copies cannot be made, then a more extreme option can be used:

Fully Sharded Data Parallelism (FSDP)

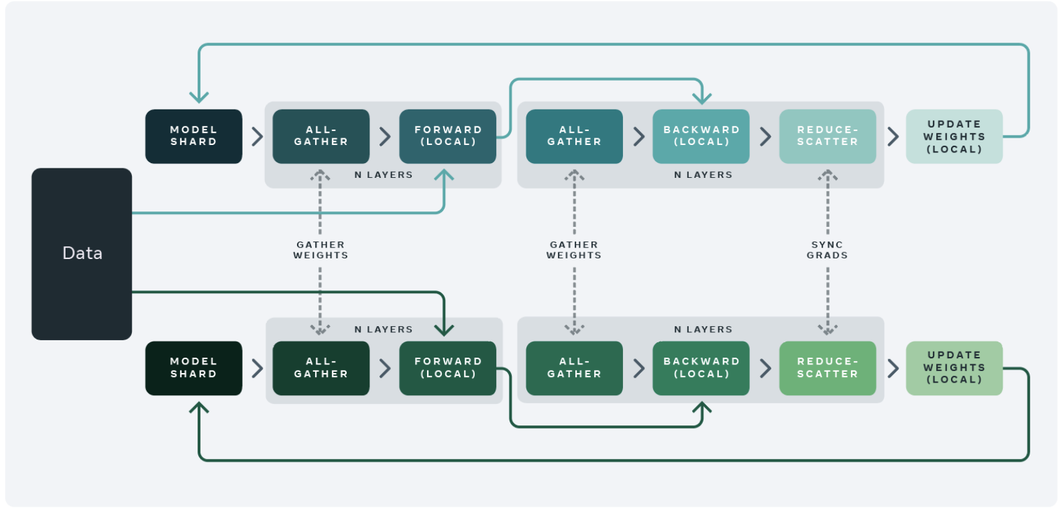
FSDP extends the idea of data parallelism where the gradients are synced across devices but goes a step further to introduce full parameter sharding. Here, where only a subset of the model parameters, gradients, and optimizers needed for a local computation is made available.
As one shard is completed, the data is communicated to the other GPUs where the shards of the next layer of the model are then updated.

FSDP only holds one copy of the model in memory since the motivation to sharding the model was that we were memory constrained.

Standard data parallel training



Fully sharded data parallel training



DARTMOUTH ENGINEERING | NVIDIA

Principles of Model Parallelism

Extreme Model Sizes and Distributed Training

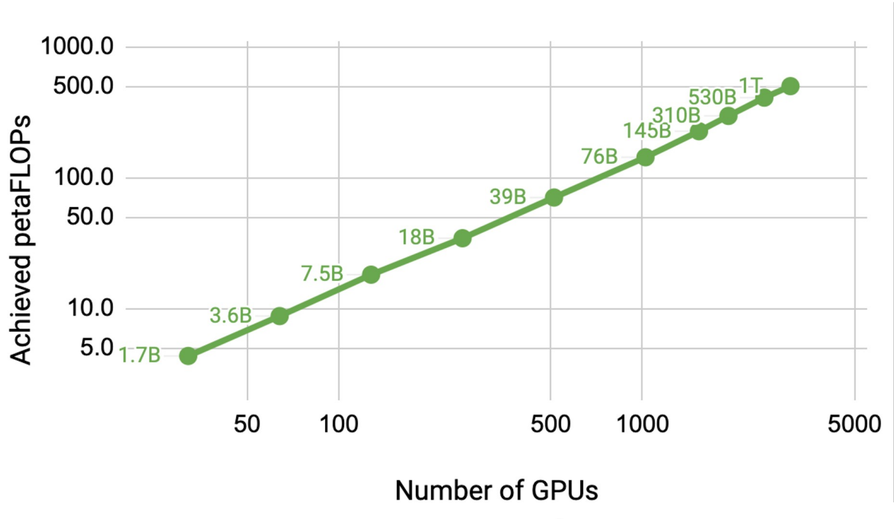
As model sizes continue to increase at an exponential pace, managing the training and inference of these giant neural networks becomes an engineering challenge.

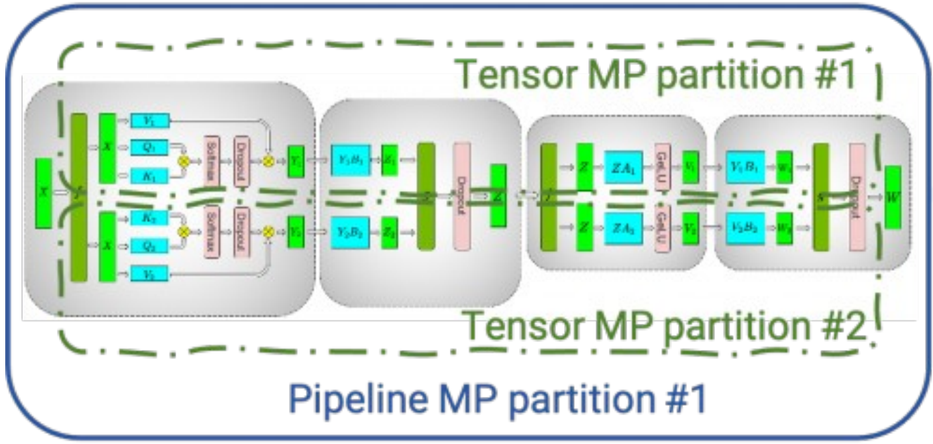
Data Parallelism focused on being able to train a model faster using copies of the model on each GPU and splitting up the training data.

FSDP took this further to allow the parameters of larger models to be sharded along with data, gradients, and optimizer states and sent to the GPUs.

As a model continues to grow in size though, these shards become too large for a single GPU to manage and requires us to rethink how we perform calculations of the internal structures of these models across devices.

This is model parallelism.

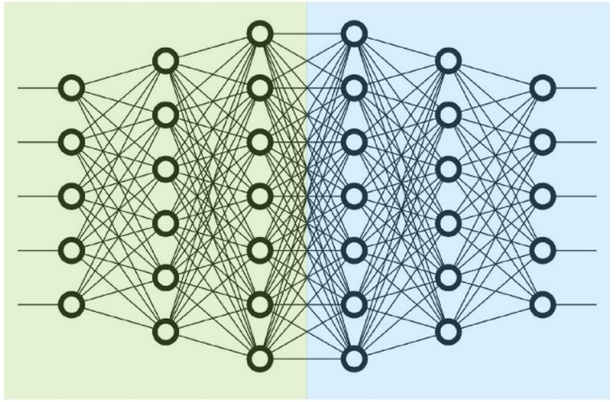




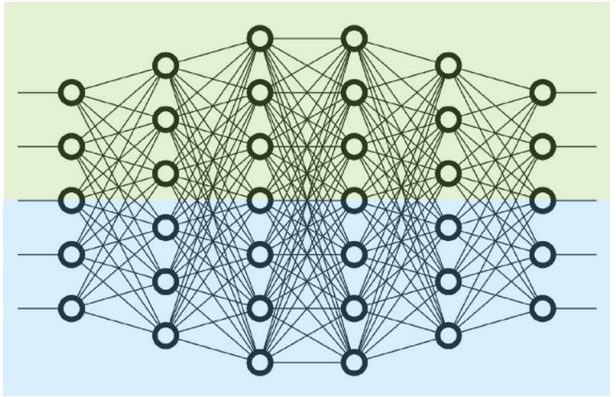
How to split a model: pipeline, tensor

There are two main types of model parallelism:

Pipeline Parallelism
Description
The model is split logically across layers with each layer/group of layers assigned to a GPU. During backpropagation, information is sent sequentially from one layer to the next.
Pros: Easy to understand conceptually, maximizes GPU utilization and memory usage
Cons: Suffers from latency issues with GPUs waiting for their turn to be used



Tensor Parallelism
Description
Large operations like matrix multiplications are split across GPUs for the entire model, or set of layers
Pros: Maximizes throughput and minimizes latency as all GPUs are used for each pass.
Cons: Can be more complex to program, suffers communication overhead when implemented across multiple nodes



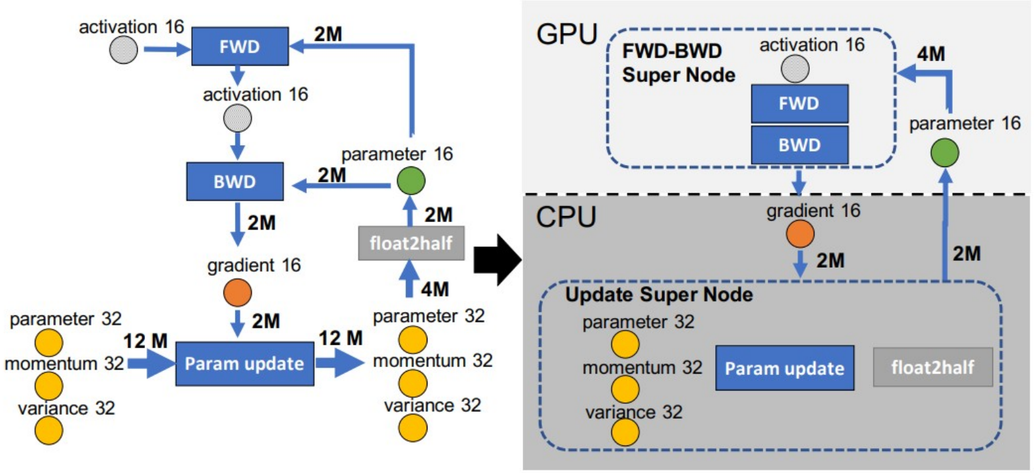
CPU Offloading

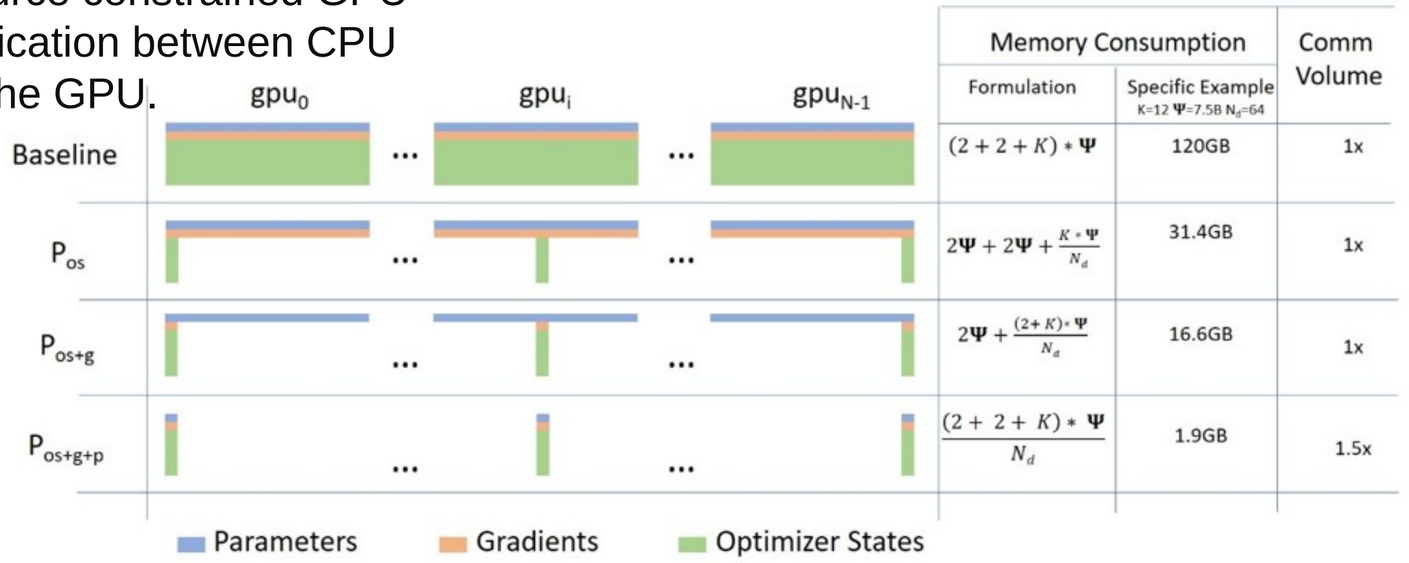
While the logic behind “get more GPUs and you can build bigger models” makes sense, it does have some practical limits.

Many cannot access extra GPUs to continue to scale their models, requiring another approach.

In CPU Offloading, only the parts of the model that can fit on the GPU/s are loaded at a time, with the rest of the data ”offloaded” to the CPU and main memory.

This enables larger models to be trained on resource constrained GPU environments, but sacrifices time as the communication between CPU and GPU is orders of magnitude slower than on the GPU.





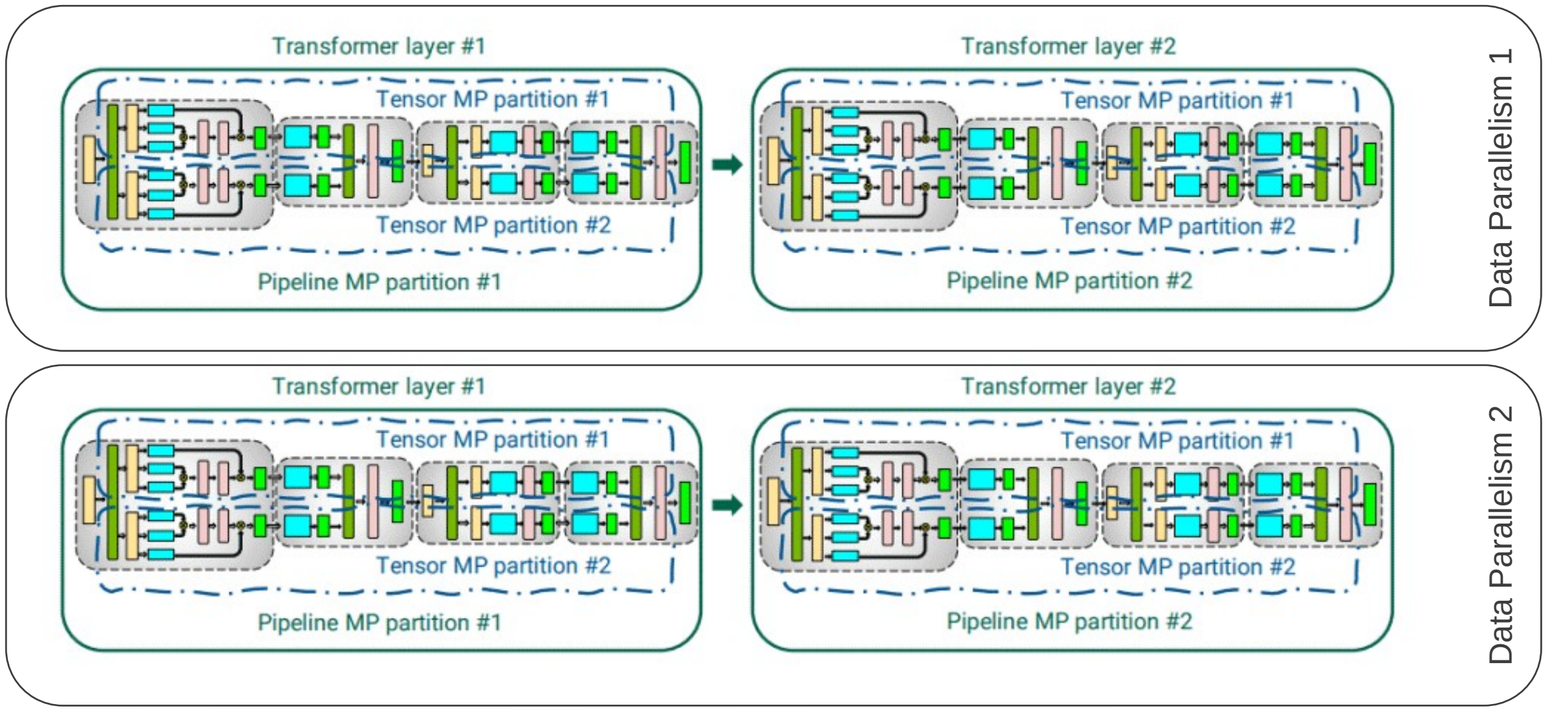
	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

FSDP and Model Parallelism

In practice it is common to see a combination of FSDP and Model Parallelism

FSDP is utilized up until the point that the individual shards are no longer able to fit on a single GPU, like a single transfer block or layer of neurons in giant models.

This hybrid approach typically makes use of some replicas of the model, like in DDP, but fewer, 2-8.



Using FSDP with Tensor Parallelism can be helpful when the model doesn't have sufficient parallelism to deploy on a large-scale training system with the data-parallel mapping. For example, running a model with the global batch size of 1024 on 2048 GPUs. Also, Tensor Parallelism enables FSDP feasibility by reducing the model state size and the activation size per GPU, thus lower the FSDP communication overhead and the activation memory overhead.

Which approach to take?

Depending on your compute budget, here are some guidelines:

Small Models Fits in a single GPU. Single-GPU Training Simple and efficient. No need for distributed strategies.

Medium Models Fits in a single GPU, but distributed training speeds up training. Normal Data Parallelism (DDP) Dataset split across GPUs; model replicated on each GPU.

Large Models (Fits with Sharding) Too large for a single GPU, but sharding allows it to fit across multiple GPUs. FSDP Shards parameters, gradients, and optimizer states; maintains data-parallel training.

Massive Models Layers/tensors too large for single GPU even with sharding. Model Parallelism (Tensor/Pipeline) Splits model computation (e.g., tensors or layers) across GPUs to handle extreme sizes.

Extreme-Scale Models Hundreds of billions of parameters; training across thousands of GPUs. Hybrid Parallelism (FSDP + Model) Combines FSDP for memory savings and model parallelism for splitting computation.

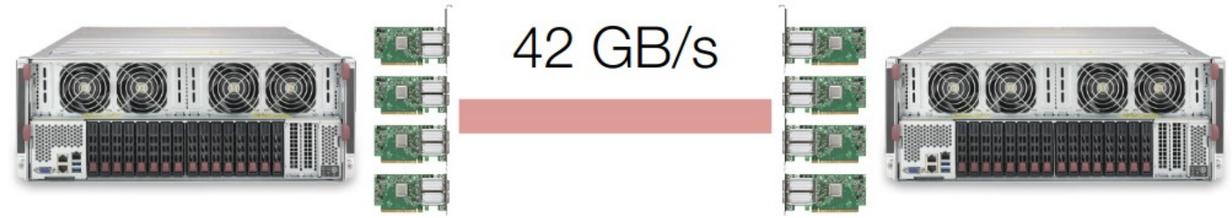
Training Across Single vs. Multi-node Clusters

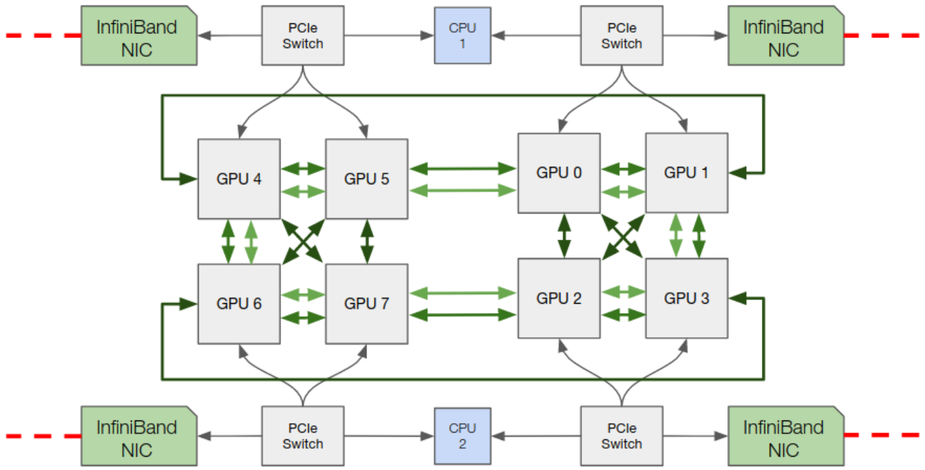
Hardware Considerations: Latency and Networking refresh

A single motherboard using in a training server, or node, will typically contain up to 8xGPUs. If more nodes are used for training, then inter-node communications and considerations are required.

These are grouped into:

- Resource management
- Network communication
- Data processing and storage





Resource Management: Creating Jobs

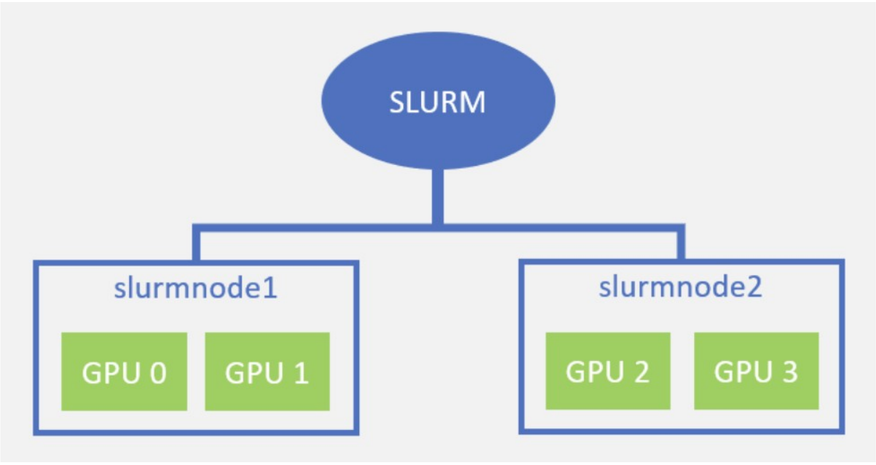
There are now several libraries that can be used to deploy multi-node training runs, some of which we will see later.

On the hardware level these rely on layers of compute infrastructure that enables communications. These include libraries like the NVIDIA Common Communications Library (NCCL).

SLURM is an open source job scheduling tool that you can use with Linux-based clusters. It is designed to be highly-scalable, fault-tolerant, and self-contained. SLURM does not require any kernel modifications for use.

Custer manager libraries like SLURM are used to allocate resources for generic multi-node applications that run on top of it and are used to launch batch jobs over the network

```
#!/bin/bash
#SBATCH --job-name=pyg-multinode-tutorial # identifier for the job listings
#SBATCH --output=pyg-multinode.log        # outputfile
#SBATCH --partition=gpucloud              # ADJUST this to your system
#SBATCH -N 2                              # number of nodes you want to use
#SBATCH --ntasks=4                        # number of processes to be run
#SBATCH --gpus-per-task=1                 # every process wants one GPU!
#SBATCH --gpu-bind=none                   # NCCL can't deal with task-binding...
```

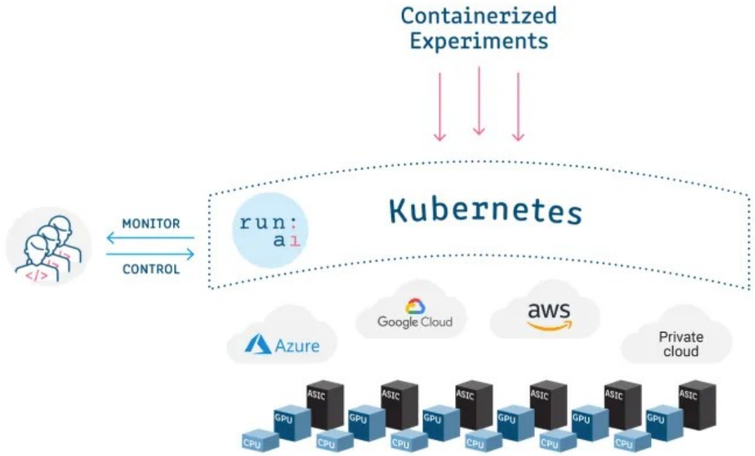


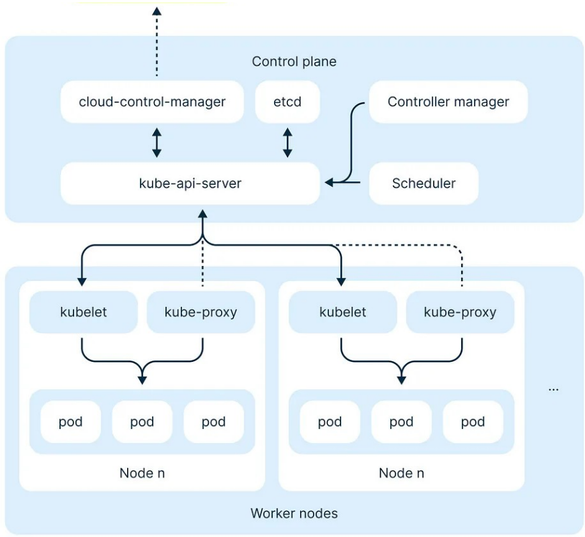
Kubernetes and common software stacks

A commonly used software stack for distributed training involves Docker containers and Kubernetes.

Kubernetes allows for the distribution of containers across the cloud into nodes. These nodes hosts a group of pods with their own resources, such as GPU nodes. This network enables pods to communicate with each other while still containing isolated workloads or applications.

For many modern GenAI training workflows, Kubernetes is used since it is very well aligned with the rest of the cloud-native ecosystem. However, for topology and scheduling features it often require additional plugins.





Data Management: Checkpointing, Restarting, and Cloud storage

During training on multi-node systems, it is important to guard against hardware failures as they are an eventuality in modern computing.

With the nodes all connected with high-speed internet, cloud storage has become the de-facto standard from which to retrieve and store parameters from training.

Checkpointing during training runs is also extremely valuable and can be stored as shards to improve I/O speed and reliability.

Certain training libraries, such as Mosaic Composer and Hugging Face Accelerate, also support elastic checkpointing where the number of current GPUs in one training session need not match the former checkpointed files' environment

```python
from composer.trainer import Trainer

# Save checkpoints every epoch to oci://my_bucket/checkpoints
trainer = Trainer(
    model=model,
    train_dataloader=train_dataloader,
    max_duration='10ep',
    save_folder='oci://my_bucket/checkpoints',
    save_interval='1ep',
    save_overwrite=True,
    save_filename='ep{epoch}.pt',
    save_num_checkpoints_to_keep=0,  # delete all checkpoints locally
)
```

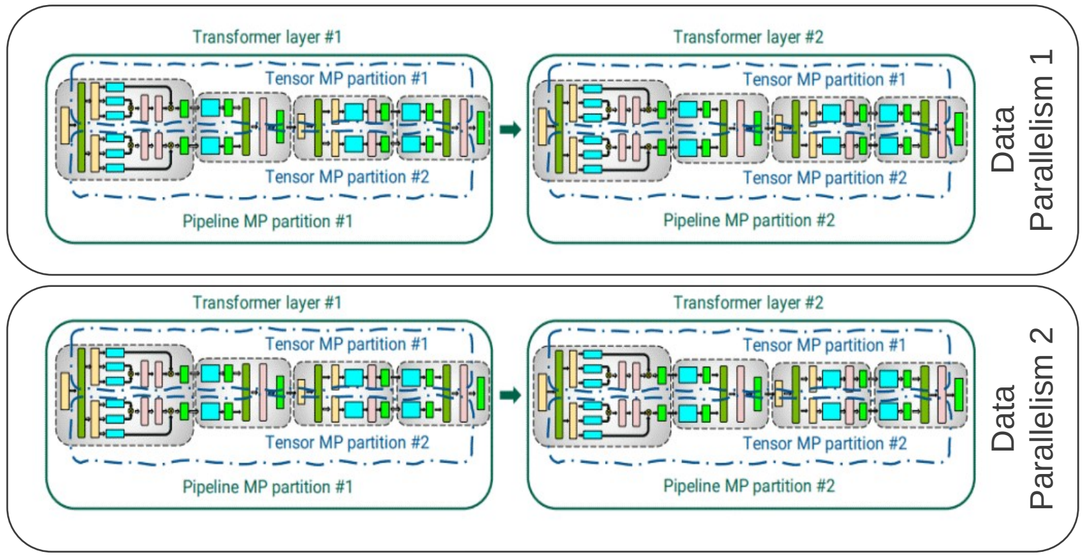
Attribute	Description
model	The model under training.
optimizers	The optimizers being used to train the model.
schedulers	The learning rate schedulers.
algorithms	The algorithms used for training.
callbacks	The callbacks used for training.
scaler	The gradient scaler in use for mixed precision training.
timestamp	The timestamp that tracks training loop progress.
rank_zero_seed	The seed of the rank zero process.
train_metrics	The current training metrics.
eval_metrics	The current validation metrics.
run_name	The run name for training.
dataset_state	The dataset iteration state.

Wrap Up

Distributed Training with Data and Model Parallelism Strategies

- Today we introduced several topics in data and model parallelism and distributed computing.
- We saw the different approaches to dealing with larger and larger models to train them using several GPUs and multi-node systems.
- Based on the size of the model and the available hardware we discussed guidelines on the best approach to train each model
- We covered some of the important distributed libraries like SLURM and discussed important aspects like checkpointing to ensure our training runs are reliably maintained and can be re-used.

In the next class we will explore some of the distributed training libraries in more detail and investigate more practical considerations of the challenges in distributed training.



Thank you!