सत्यमेव जयते

Ministry of Electronics & Information Technology

सुस्वागतम्
நல்வரவு
ಸುಸ್ವಾಗತ
സുസ്വാഗതം
ସୁସ୍ୱାଗତମ
ਸੁਆਗਤਮ
خوش آمدید

NPSF User Workshop 2023

24 July – 24 Aug

Presentation On

Training in pyTorch

Swaroop Lokesh – Quantiphi

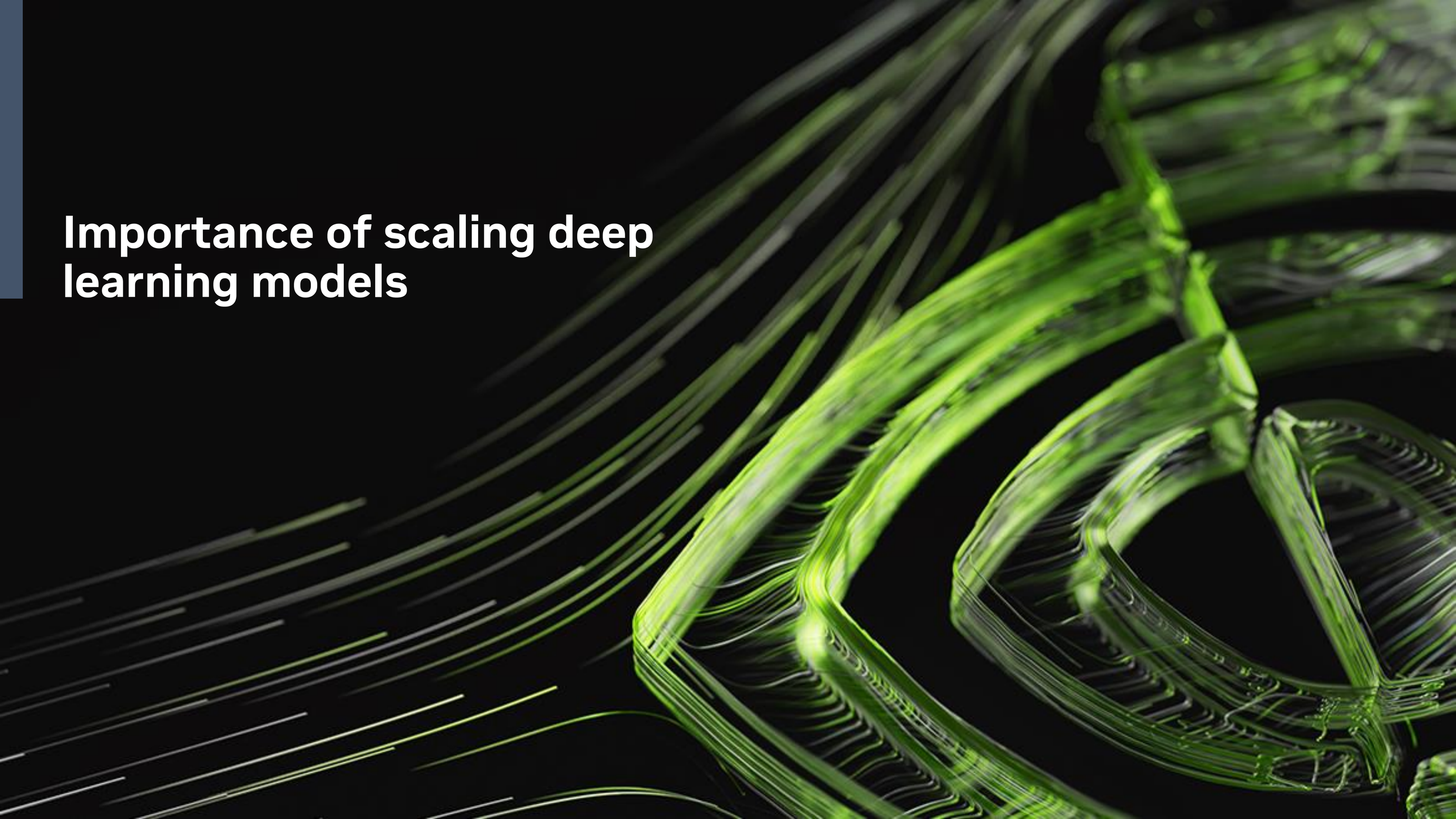
31st July

National PARAM Supercomputing Facility

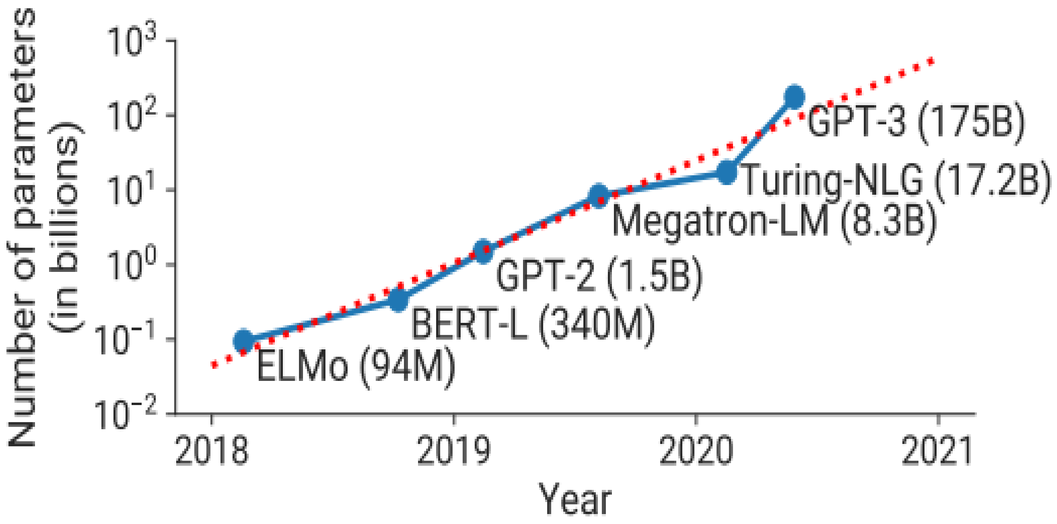
Agenda

- Importance of scaling deep learning models
- Benefits and Challenges of Multi-GPU and Multi-Node Training
- Parallelization Overview
  - Data Parallelism
  - Model Parallelism
- Pytorch Distributed Overview
- Handling communication and Synchronization between nodes and GPUs
- Performance optimization techniques
- Best Practices
- Demo

Importance of scaling deep learning models



Importance of scaling deep learning models

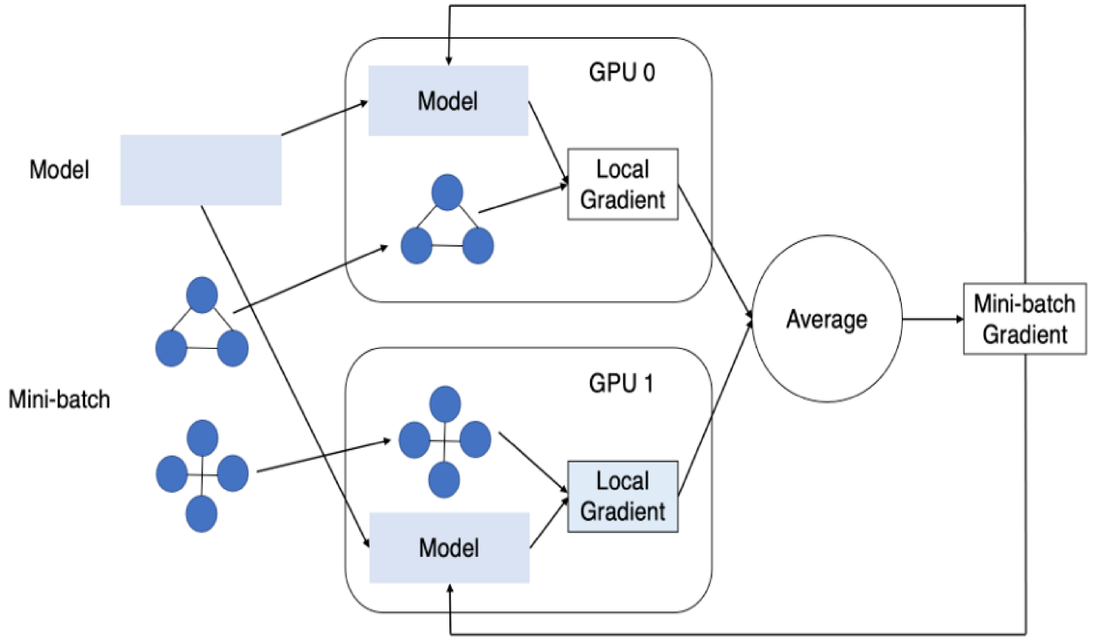


- Large language models are effective zero- or few-shot learners, with high accuracy on many NLP tasks and datasets.
- The number of parameters in state-of-the-art NLP models have grown at an exponential rate
- Training such models, however, is challenging for two reasons:
  - It is no longer possible to fit the parameters of these models in the main memory of even the largest GPU (NVIDIA recently released 80GB-A100 cards)
  - Even if we were able to fit the model in a single GPU (e.g., by swapping parameters between host and device memory), the high number of compute operations required can result in unrealistically long training times (e.g., training GPT-3 with 175 billion parameters would require approximately 288 years with a single V100 NVIDIA GPU).

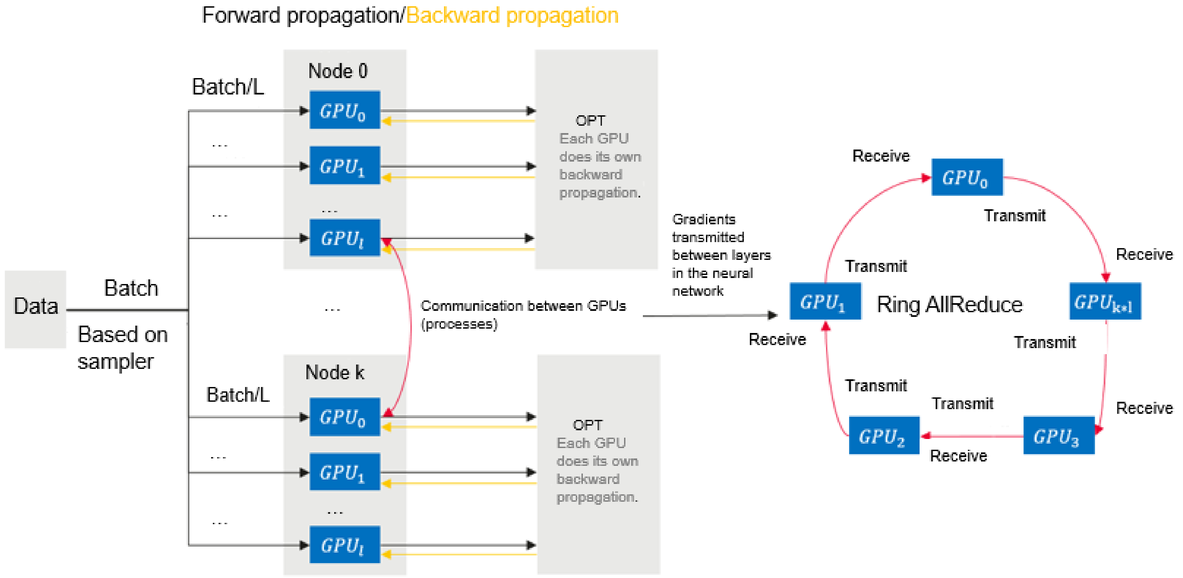
Benefits and Challenges of Multi-GPU and Multi-Node Training

Multi-GPU and Multi-node Training

Overview



Multi-GPU, single-node.



Multi-GPU,Multi-nodeTraining

- When training on a single GPU is too slow or the model weights don't fit in a single GPU's memory, we use a multi-GPU setup
- Multi-Node training refers to training deep learning models across multiple nodes or machines. It is particularly useful for training large-scale models on massive datasets, achieving even faster training times and handling computational workloads that exceed the capabilities of a single machine.

Multi-GPU and Multi-Node Training

Benefits

- **Faster Training:**
  - With multi-GPU and multi-node training, developers can distribute the workload across multiple GPUs and nodes
  - Multiple GPUs/multiple nodes work in parallel, processing different batches of data simultaneously. This reduces the time required to complete one training epoch.
- **Larger Batch Sizes:**
  - Multi-node and multi-gpu training enables the distribution of large datasets across multiple nodes.
- **Handling Larger Models:**
  - You can train larger models that may not fit into the memory of a single GPU or single node.
  - Each GPU and each node can handle a portion of the model and the data, enabling the training of models with more parameters.
- **Better Performance:**
  - improves the performance of the model by allowing it to process more data in parallel.
- **Efficient Resource Utilization:** .
  - With multi-GPU training, the model and data are distributed across GPUs, making better use of available memory.
  - Each node can make use of its own GPUs, CPUs, and memory resources
- **Parallelism at a Larger Scale:**
  - Each node or each gpu (in case of muti-gpu, single node) processes a subset of the data and computes gradients for its part of the model.
  - Then, the gradients are synchronized among the GPUs and used to update the shared model parameters.

Multi-GPU and Multi-node Training

Deciding between multi-gpu and multi-node training

The decision between multi-node and multi-GPU training depends on the scale of your model, the size of your dataset, the available resources (number of nodes and GPUs), and the desired training speed.

The case for multi-node:

- Multi-node training allows you to scale your training to larger models and datasets that cannot fit within the memory of a single node.
- When dealing with massive datasets that cannot fit in the memory of a single machine, multi-node training enables you to distribute the data across nodes for parallel processing.
- In case where the model architecture fits within the memory of a single node, but the required batch size exceeds the GPU's memory capacity, multi-GPU training allows for larger batch sizes.

When to use multi-gpu training only:

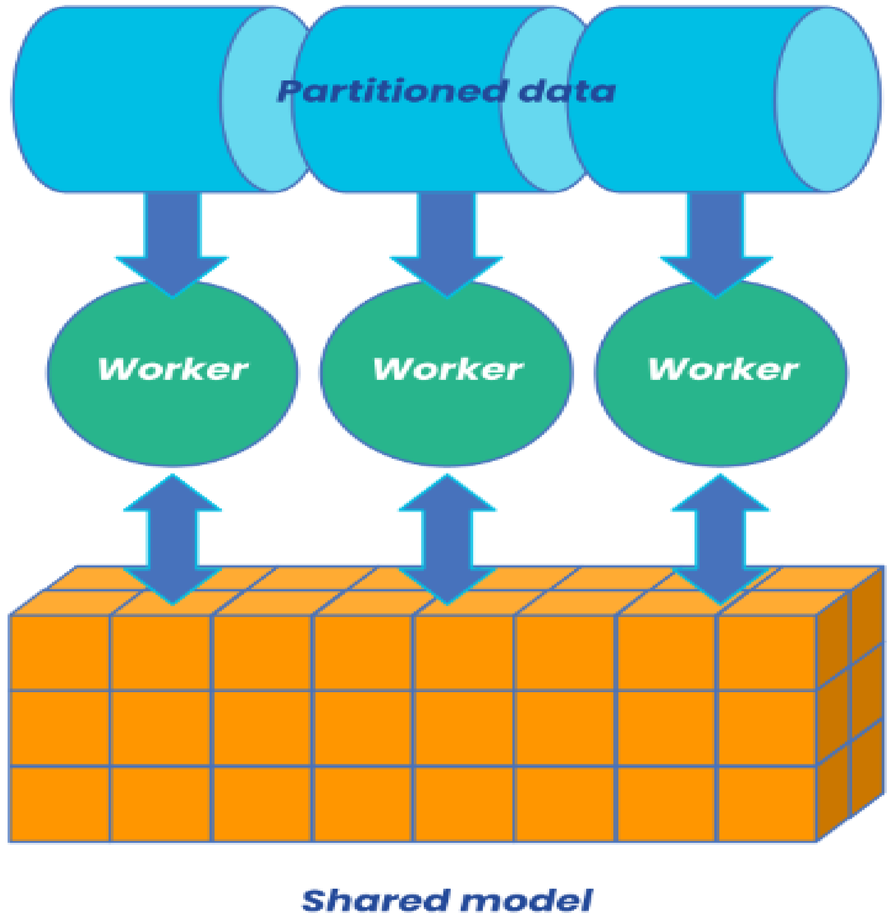
- multi-GPU training on a single node is advantageous when you need faster training on medium-sized models, want to iterate quickly during development, or require larger batch sizes while staying within the memory limits of each GPU.

Parallelization Overview - Data Parallelism, Model Parallelism

Parallelization Overview

Data Parallelization

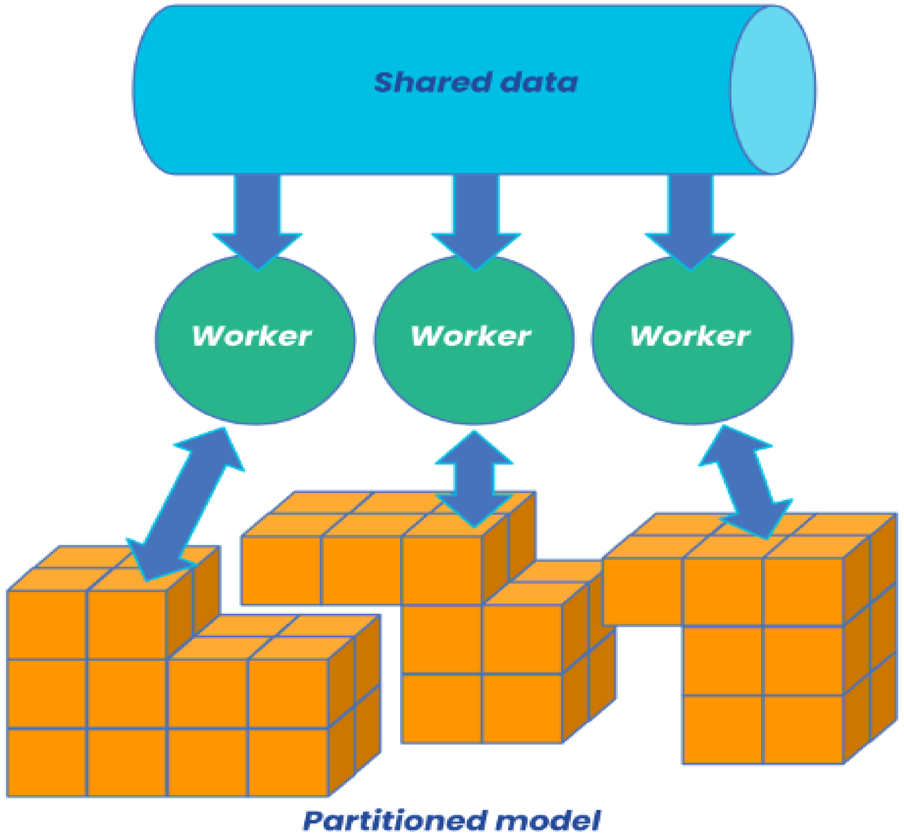
- Data parallelism shards data across all cores with the same model
- A data parallelism framework like PyTorch Distributed Data Parallel, SageMaker Distributed, and Horovod mainly accomplishes the following three tasks:
  - First, it creates and dispatches copies of the model, one copy per each accelerator.
  - It shards the data and then distributes it to the corresponding devices.
  - It finally aggregates all results together in the backpropagation step.
- Data parallelism has several benefits:
  - Faster Training
  - Increased Model Capacity
  - Better Generalization
- PyTorch Distributed Data Parallel (DDP) implements data parallelism at the module level for running across multiple machines



Parallelization Overview

Model Parallelization

- Model parallelism shards a model (i.e., its layers or tensors) across multiple cores, unlike data parallelism, replicating the same model for all training cores.
- Enables the training and inference of extremely large and complex models by distributing the computational workload across multiple processing units.
- Model parallelism has several benefits:
  - Memory Efficiency
  - Scalability
  - Flexibility
  - Can be combined with Data Parallelism
- Model parallelism techniques:
  - Pipeline Parallelism
    - The model is split up vertically (layer-level) across multiple GPUs, so that only one or several layers of the model are places on a single gpu.
  - Tensor Parallelism
    - Each GPU processes only a slice of a tensor and only aggregates the full tensor for operations that require the whole thing.

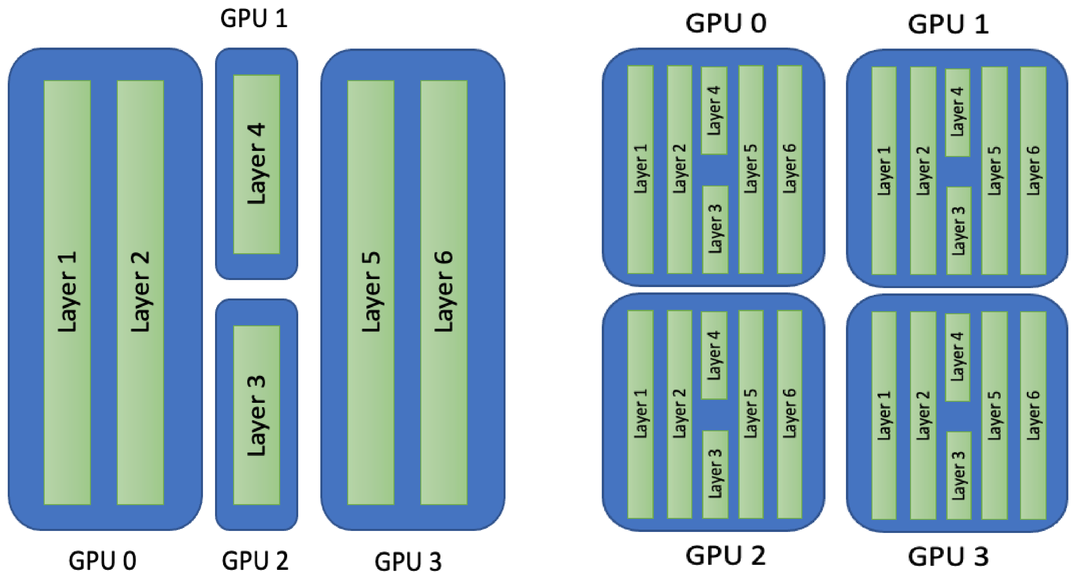


Pytorch Distributed Overview

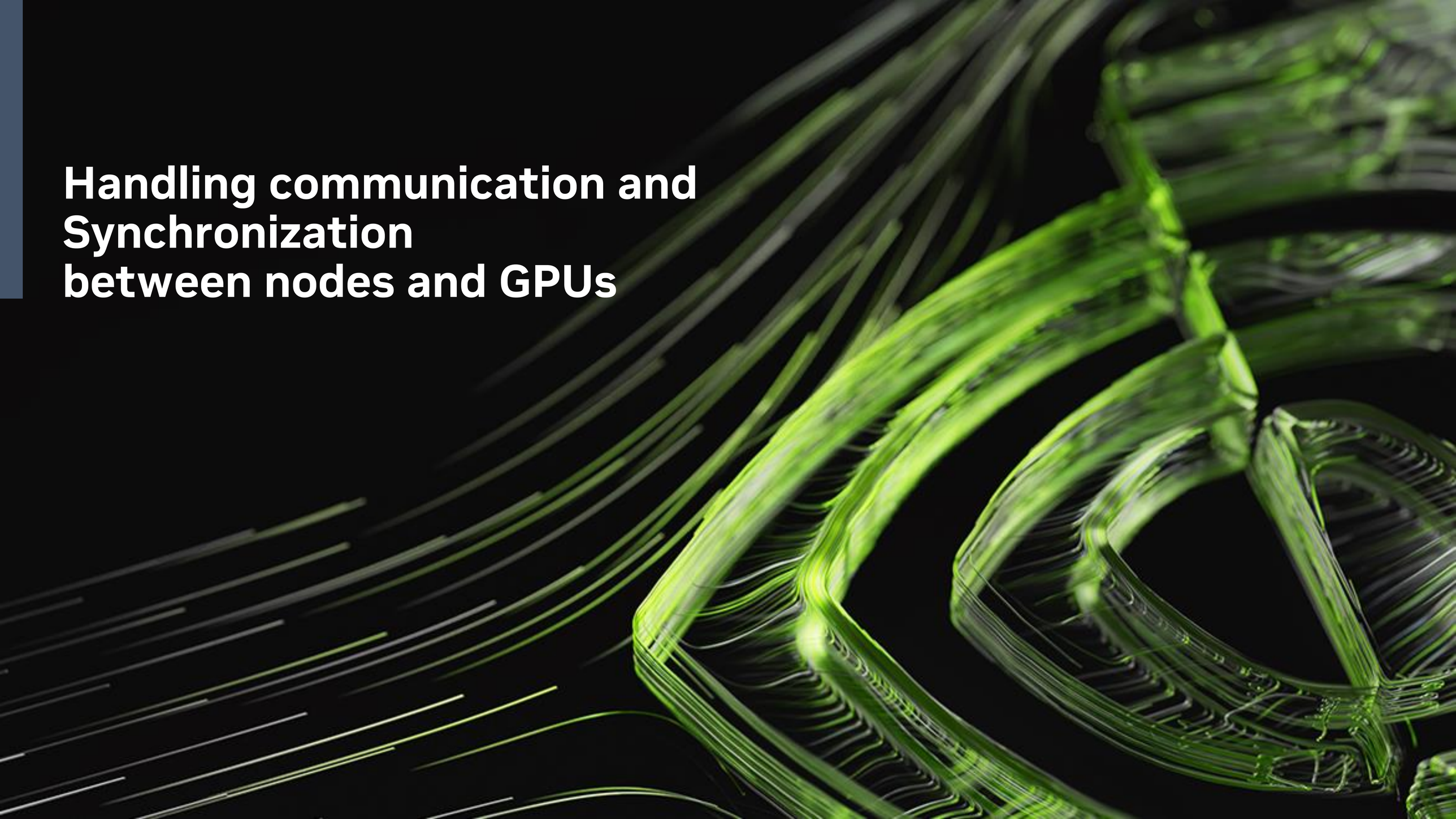
Pytorch Distributed

A powerful framework for distributed deep learning training and inference

- It can be categorized into three main components:
  - Distributed Data-Parallel Training (DDP)
    - Single-program multiple-data training paradigm
    - With DDP the model is replicated on every process.
    - Therefore, every model replica will be fed a different set of input data samples.
    - In addition, DDP takes care of gradient communication to keep the different model replicas on separate processes in sync and overlaps it with the gradient computations to speed up training.
  - RPC-Based Distributed Training (RPC)
    - Supports general training structures that cannot fit into data-parallel training, such as distributed pipeline parallelism, parameter server paradigm, or combinations of DDP with other such paradigms.
  - Collective Communication (c10d) library
    - Offers to send tensors across processes within a group. It offers collective communication APIs and P2P communication APIs.

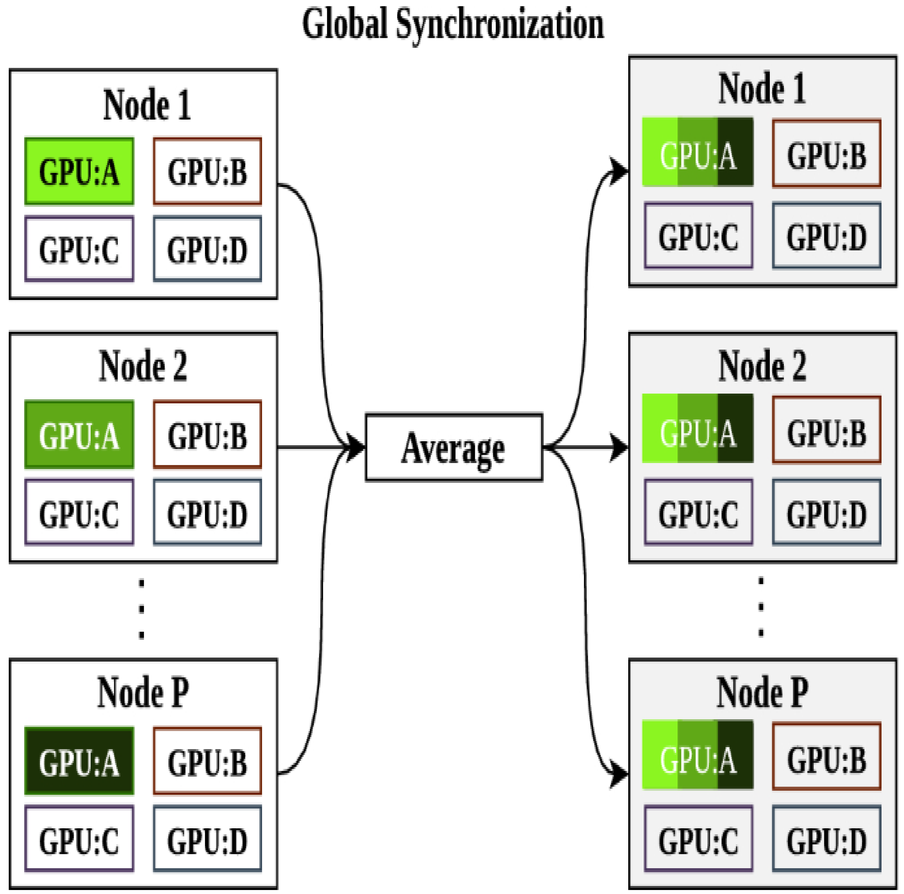


Handling communication and Synchronization between nodes and GPUs

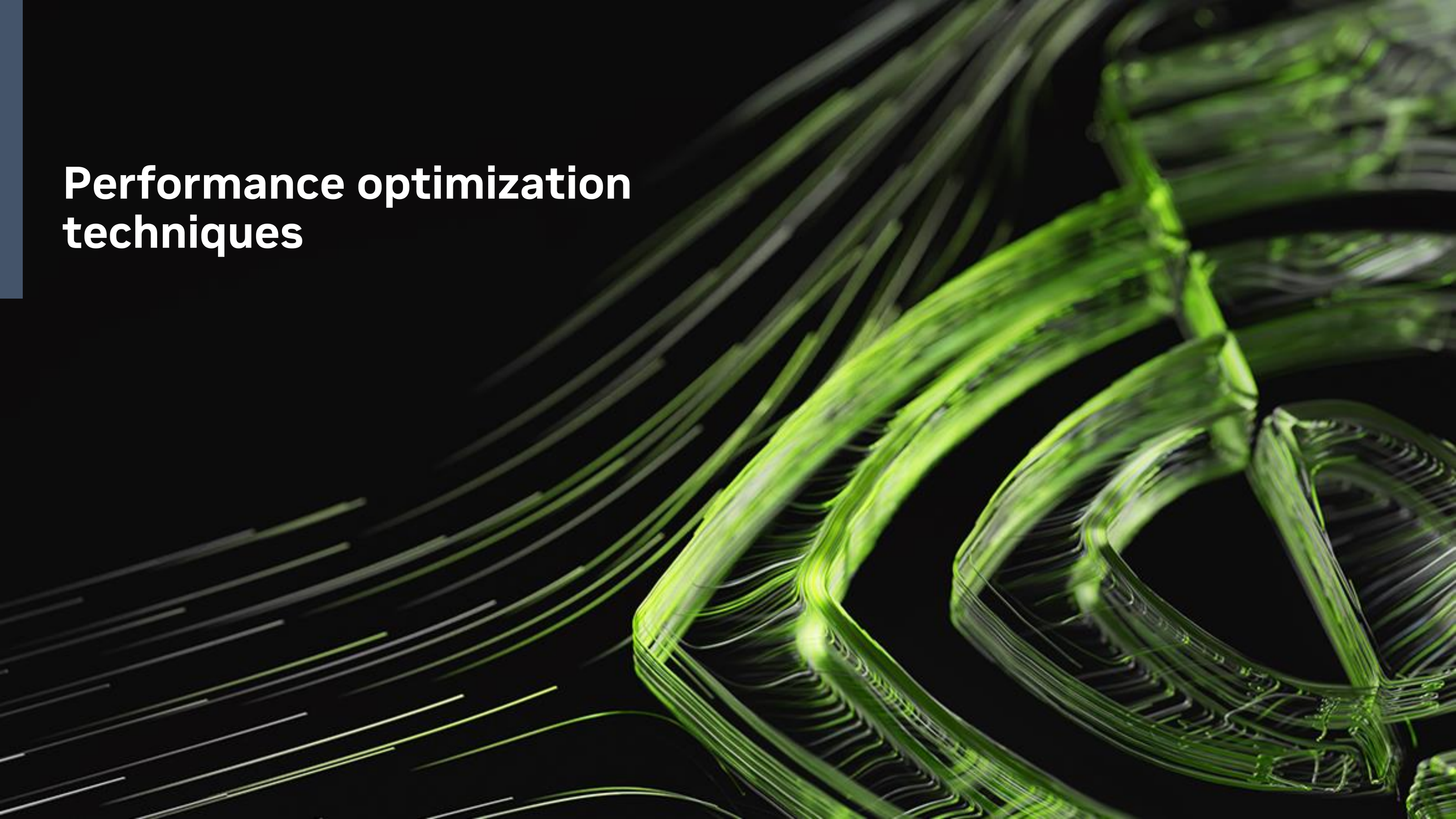


Handling Communication & Synchronization between Nodes & GPUs

- When using data parallelism or model parallelism across multiple nodes and GPUs, handling communication and synchronization becomes crucial.
- In data parallelism,
  - Each node/gpu has a complete copy of the model. During training, the input data is divided into batches, and each node/gpu processes a different batch using its local model.
  - PyTorch's torch.nn.DataParallel and torch.nn.parallel.DistributedDataParallel handle the communication of gradients automatically by performing all-reduce operations across the nodes/gpu's.
  - Synchronization is managed internally by the data parallelism wrappers.
- In model parallelism,
  - The model is partitioned across multiple GPUs or nodes, with each handling a specific portion of the model.
  - The data or intermediate activations that need to be passed between GPUs are explicitly sent and received using the torch.distributed package's communication primitives, such as send, recv, or all_to_all.
  - Synchronization can be achieved using torch.distributed.barrier, which acts as a synchronization point for all nodes in the distributed setup.



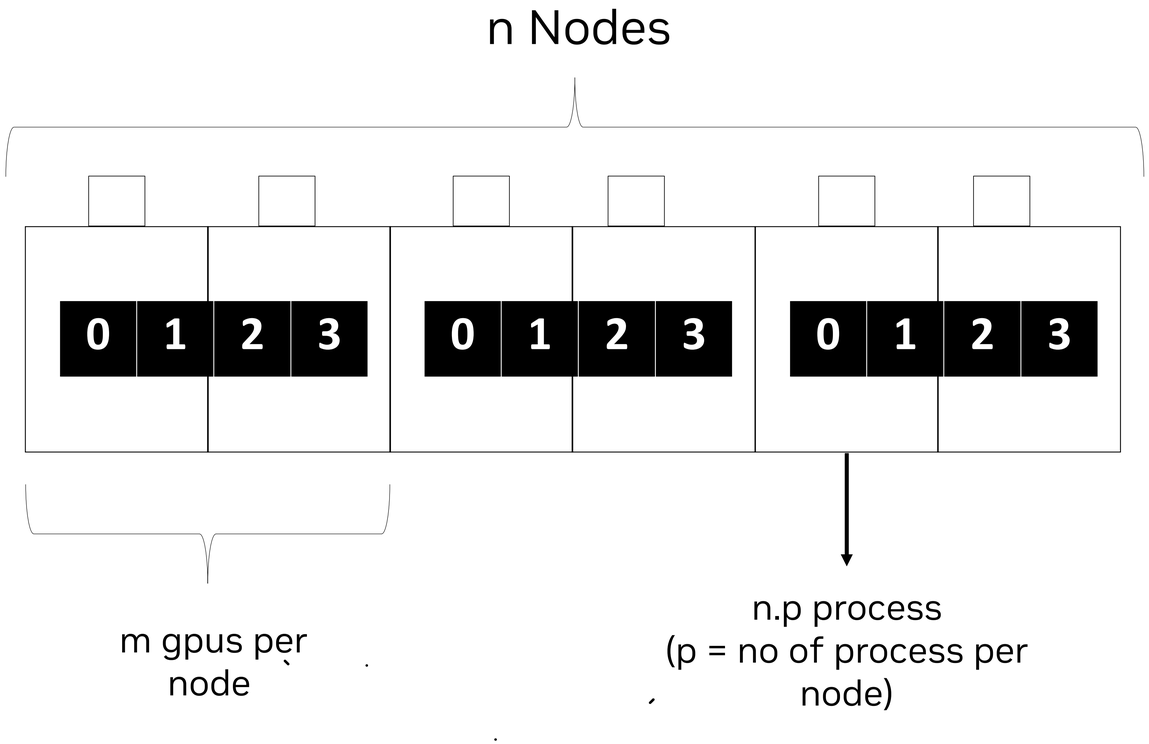
Performance optimization techniques



Performance Optimization Techniques

- **Use efficient data-parallel backend:**
  - PyTorch has two ways to implement data-parallel training:
    - torch.nn.DataParallel
    - Torch.nn.parallel.DistributedDataParallel
  - DistributedDataParallel offers much better performance and scaling to multiple-GPUs.
- **Skip unnecessary all-reduce if training with DistributedDataParallel and gradient accumulation.**
  - DistributedDataParallel executes gradient all-reduce by default after every backward pass.
  - If training uses gradient accumulation over N steps, then all-reduce is not necessary after every training step.
  - DistributedDataParallel provides no_sync() which should be applied to first N-1 iterations of gradient accumulation, the last iteration should follow the default execution
- **Match the order of layers in constructors and during the execution if using DistributedDataParallel (find_unused_parameters=True)**
  - find_unused_parameters=True can build buckets for DistributedDataParallel gradient all-reduce using the order of layers and parameters from model constructors
  - with find_unused_parameters=False (which is the default setting) relies on automatic bucket formation based on order of operations encountered during the backward pass.
- **Load-balance workload in a distributed setting**
  - If batch of data received are not same, then all devices wait for the worker which finishes last.
  - Backward pass functions as an implicit synchronization point in a distributed setting with DistributedDataParallel backend.
  - Transformer solves imbalance by forming batches with approximately constant number of tokens while other models solve imbalance by bucketing samples with similar sequence length or even by sorting dataset by sequence length.

Selecting Devices for Hybrid Data/Model Parallelism

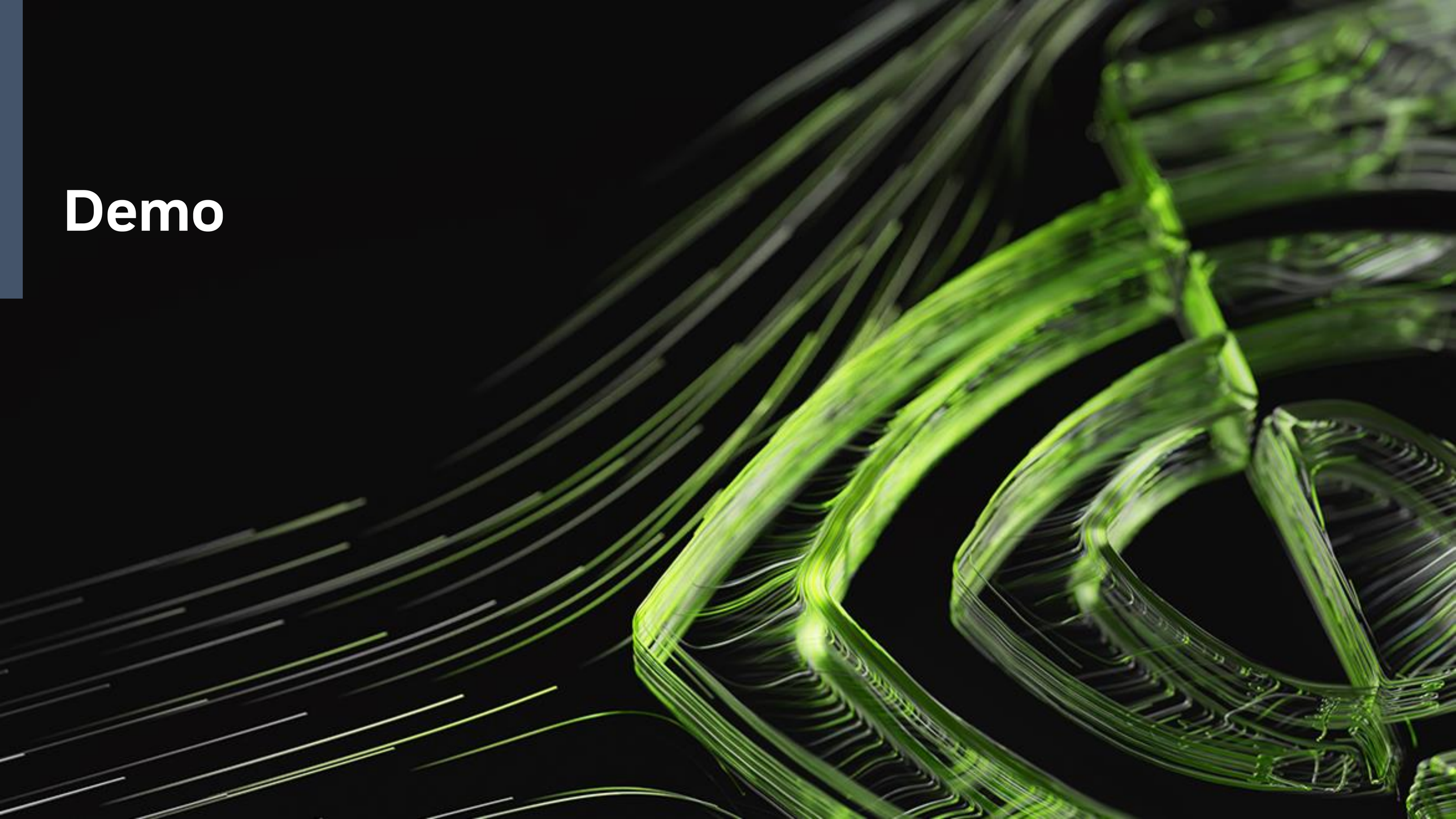



```
rank = dist.get_rank()

n = int(os.environ["NUM_NODES"])          # Number of nodes
p = int(os.environ["NUM_PROC_PER_NODE"])  # Number of processes per node
m = torch.cuda.device_count()             # Number of gpus per node
g = m // p                                # Number of gpus per process

devices = [(rank % p) * g + i for i in range(g)]
```

Demo



Exascale Computing
Microprocessor and Quantum Computing
AI and Language Computing
IoE, Dependable and Secure Computing
GenNext Applied Computing
10010111001100
Advanced Computing For Human Advancement
Thank You !