Ministry of Electronics and Information Technology
Government of India

Best Practice and Methodologies for DL Development on Supercomputing Platform

One Vision. One Goal... Advanced Computing for Human Advancement…

Outline

- **Introduction to the Workshop & Deep Learning on HPC**
  - Workshop Overview
  - Presenters & Contact Information
  - Exponential Growth of AI Models & Compute Power
  - Hands-on Learning Resources (GitHub, Cluster Access)
- **High-Performance Computing (HPC) for AI Workloads**
  - HPC System Architecture & PARAM RUDRA Overview
  - HPC Containerization
  - Job Scheduling & Resource Management with Slurm & Pyxis
  - Running AI Workloads Using NVIDIA NGC & Enroot
- **Deep Learning Training: Single GPU to Distributed Systems**
  - AI Frameworks: PyTorch vs TensorFlow
  - Single GPU Training: Data Loading, Optimization, and AMP
  - Multi-GPU Training: DDP and FSDP
  - Synchronization & Communication in Distributed AI Training
  - Hands-on Session on Distributed Training
- **Advanced Distributed AI Training & Model Optimization**
  - DeepSpeed: Optimizing Large Model Training
  - Enhancing HPC with Containerization
  - Hands-on Session on DeepSpeed and Containers
  - Demonstration of Multi-node Training with and without containers

Welcome to the SCA25, Best Practice and Methodologies for DL Development

In this tutorial, you will learn how to effectively utilize HPC clusters for deep learning model training, including:

- Configuration and optimization for AI workloads
- SLURM scheduler for job management
- Virtual environments and container technologies for HPC
- Distributed training using PyTorch and DeepSpeed

Key Topics Covered:

- Deep learning training workflows on single and multi-GPU setups
- Hands-on implementation of distributed training with DDP and FSDP
- Optimization of LLMs using DeepSpeed
- Running AI workloads using NVIDIA NGC, Enroot and Slurm

Note:

- This tutorial does not cover deep learning basics but will introduce key concepts relevant to HPC-based AI training.
- Hands-on code and resources will be provided for practical learning on an HPC cluster.

Who are we ?

We are CDAC (Centre for Development of Advanced Computing)

A premier R&D organization of the Government of India, driving technological advancements in computing and digital infrastructure. Our mission is to address India’s growing needs in areas like high-performance computing, artificial intelligence, digital communication, and more. We aim to create innovative solutions that support national initiatives and contribute to India's technological leadership.

Presenters:

Mr. Shashank Sharma
shashank.sharma@cdac.in

www.linkedin.com/in/shashank-sharma-93119543/

Ms. Sowmya Shree N
ssowmya@cdac.in

www.linkedin.com/in/sowmya-shree-n-466b07136/

Mr. Kishor Y D
kishoryd@cdac.in

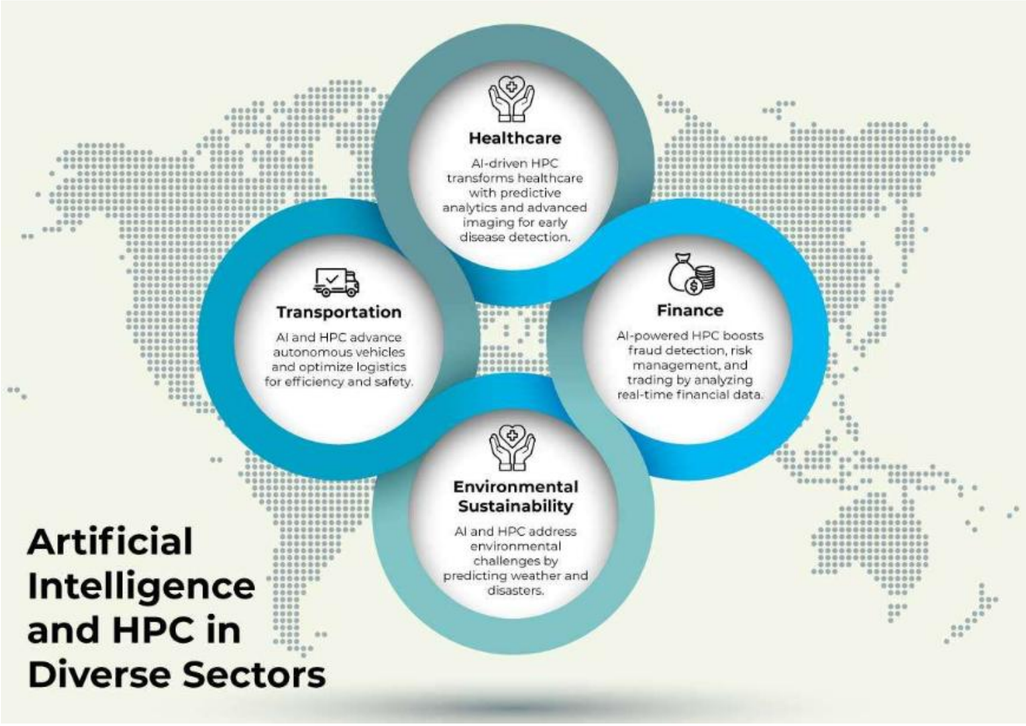
www.linkedin.com/in/kishor-yd

Mr. Anandhu Nair
anandhun@cdac.in

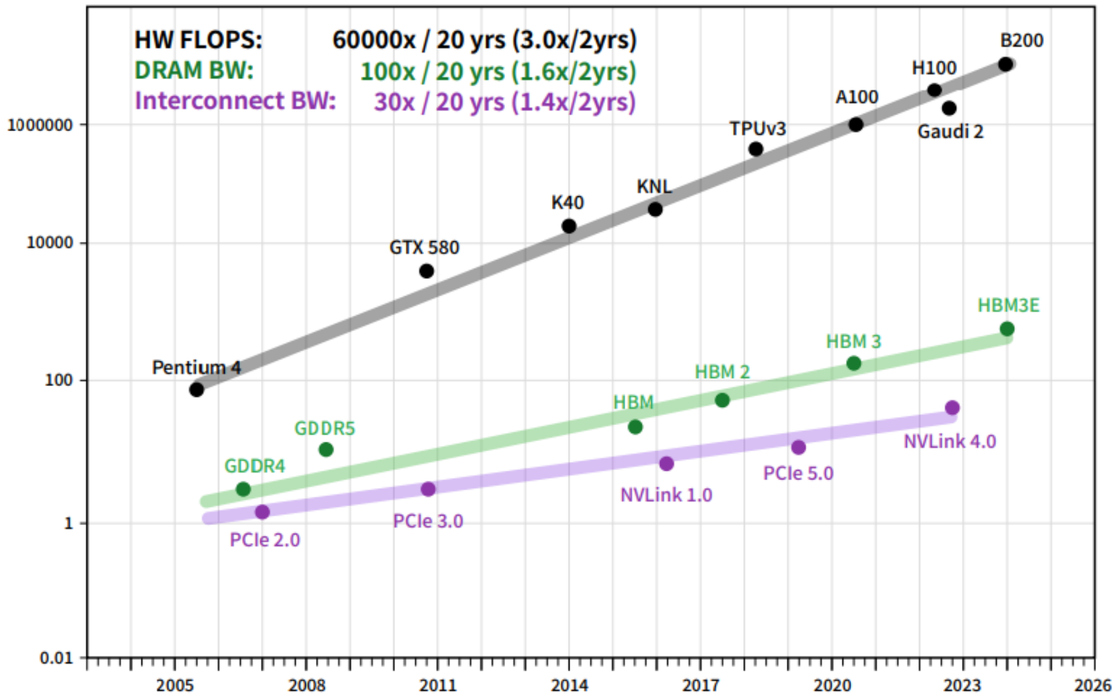
www.linkedin.com/in/anandhunair

Why are we here today?

- Deep Learning (DL) is changing industries, science, and everyday life.
- Training DL models needs a lot of computing power and works well with HPC systems.
- Using modern HPC systems can speed up DL tasks significantly.
- Efficient resource management in HPC can reduce training time and costs.



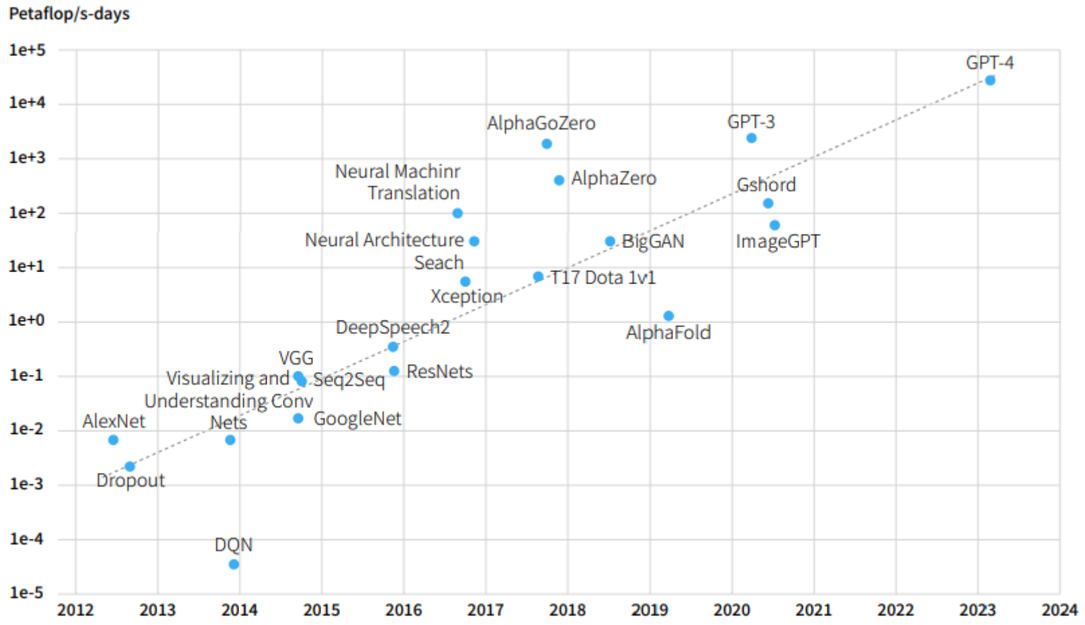
Courtesy: https://www.zinfi.com/blog/artificial-intelligence-in-high-performance-computing/



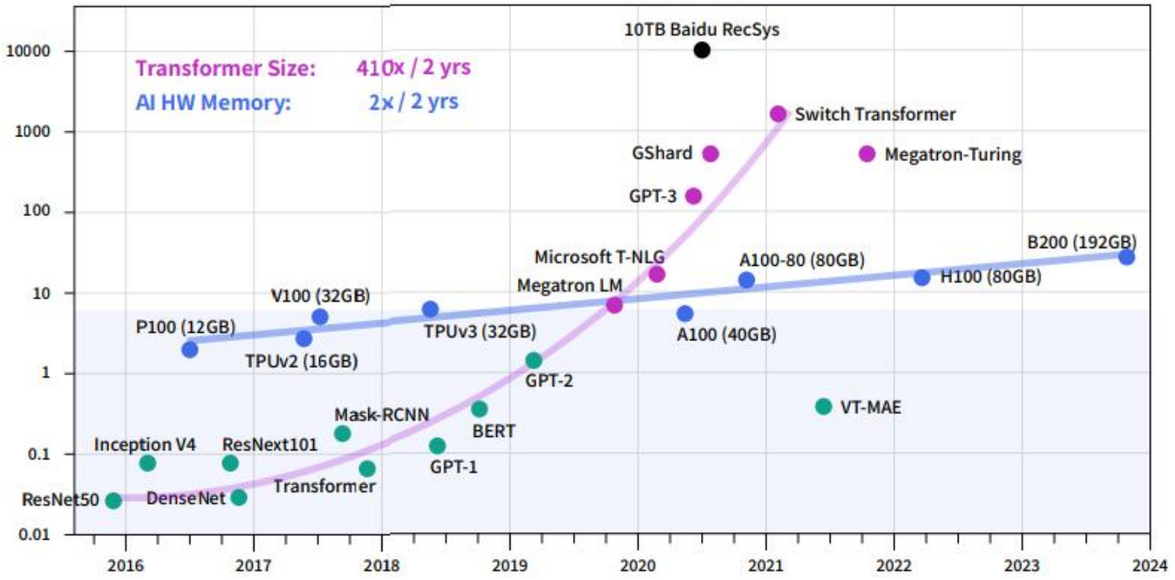
Courtesy: https://arxiv.org/pdf/2408.14158v1 Peak Hardware FLOPs and memory bandwidth

Exponential Growth of model size and Computational Power for Deep Learning

- **Increasing Computational Demand –** AI model training now requires petaflop-scale computing, making HPC systems essential.
- **Rapid Growth of AI Models –** Deep learning models have expanded exponentially, requiring higher computational power.
- **Scaling AI Hardware –** Transformer model sizes have increased **410x in two years,** while AI hardware memory has doubled every **two years.**



Courtesy: https://arxiv.org/pdf/2408.14158v1 Exponential Growth of Computational Power for Deep Learning



Courtesy: https://arxiv.org/pdf/2408.14158v1 Size of Model Parameter and Accelerator Memory

Time Series, From Regression to LLMs

1805 — **Regression**: Utilizes 10 to 60 input features with an equivalent number of parameters (weights) optimized during training.

1975 — **Backpropagation**: Core technique for training neural networks, connecting layers with 500 to 500,000 parameters.

1980 — **ARIMA**: Auto-Regressive Integrated Moving Average models rely on past data and averages for time-series forecasting.

1982 — **RNN**: reuse output as input, suitable for sequential data but struggle with long dependencies due to vanishing gradient issues.

1997 — **LSTM**: Long Short-Term Memory networks introduce memory cells, managing dependencies effectively and typically containing ~20 million parameters.

2013 — **Word Embeddings**: Techniques like Word2Vec represent words or entities as vectors, enabling efficient Natural Language Processing

2017 — **Attention Mechanisms**: Transformers revolutionize speech and language tasks by prioritizing relevant data within input sequences.

2018 — **BERT**: Bidirectional Transformers for NLP feature ~100 million parameters, enabling deep contextual understanding.

2022 — **ChatGPT 3.5**: Leveraging 175 billion parameters, it sets a new standard for conversational AI and text generation.

Hands-on Materials

- To claim the training account:

  https://forms.gle/QhtYJFmxrsWfWMjNA

All tutorial materials are available at
https://github.com/snsharma1311/SCA-2025-DistributedTraining

Outline

- **Introduction to the Workshop & Deep Learning on HPC**
  - Workshop Overview
  - Presenters & Contact Information
  - Exponential Growth of AI Models & Compute Power
  - Hands-on Learning Resources (GitHub, Cluster Access)
- **High-Performance Computing (HPC) for AI Workloads**
  - HPC System Architecture & PARAM RUDRA Overview
  - HPC Containerization
  - Job Scheduling & Resource Management with Slurm & Pyxis
  - Running AI Workloads Using NVIDIA NGC & Enroot
- **Deep Learning Training: Single GPU to Distributed Systems**
  - AI Frameworks: PyTorch vs TensorFlow
  - Single GPU Training: Data Loading, Optimization, and AMP
  - Multi-GPU Training: DDP and FSDP
  - Synchronization & Communication in Distributed AI Training
  - Hands-on Session on Distributed Training
- **Advanced Distributed AI Training & Model Optimization**
  - DeepSpeed: Optimizing Large Model Training
  - Enhancing HPC with Containerization
  - Hands-on Session on DeepSpeed and Containers
  - Demonstration of Multi-node Training with and without containers

HPC resources for Today: Rudra IUAC

PARAM Rudra System Specifications

Theoretical Peak Floating-point Performance Total (Rpeak)	3 PFLOPS
Base Specifications (Compute Nodes)	2X Intel Xeon GOLD 6240R, 24 Cores, 2.4 GHz Processors per node, 192 GB Memory, 800 GB SSD
Master/Service/Login Nodes	20 nos.
CPU only Compute Nodes (Memory)	473 nos. (192 GB)
High Memory Nodes (Memory)	64 nos. (768GB)
GPU Compute Nodes (GPU Cards)	27 nos. (54 Nvidia A100 PCIe)
GPU Ready Compute Nodes (Memory)	35 nos. (192GB)
Total Memory	148.312 TB
Storage	4.4 PiB
Interconnect	Primary: 100Gbps Mellanox Infiniband Interconnect Network 100% non blocking, fat tree topology Secondary: 10G/1G Ethernet Network Management Network: 1G Ethernet

CPU Only Compute Nodes

- 473 Nodes
- 22704 Cores
- Compute power of Rpeak 1743.4 TFLOPS
- Each Node with
  - 2 X Intel Xeon GOLD 6240R, 24 Cores, 2.4 GHz
  - 192 GB memory
  - 800 GB SSD

High Memory Compute Nodes

- 64 Nodes
- 3072 Cores
- Compute power of Rpeak 235.90 TFLOPS
- Each Node with
  - 2 X Intel Xeon GOLD 6240R, 24 cores, 2.4 GHz
  - 768 GB Memory
  - 800 GB SSD

GPU Compute Nodes

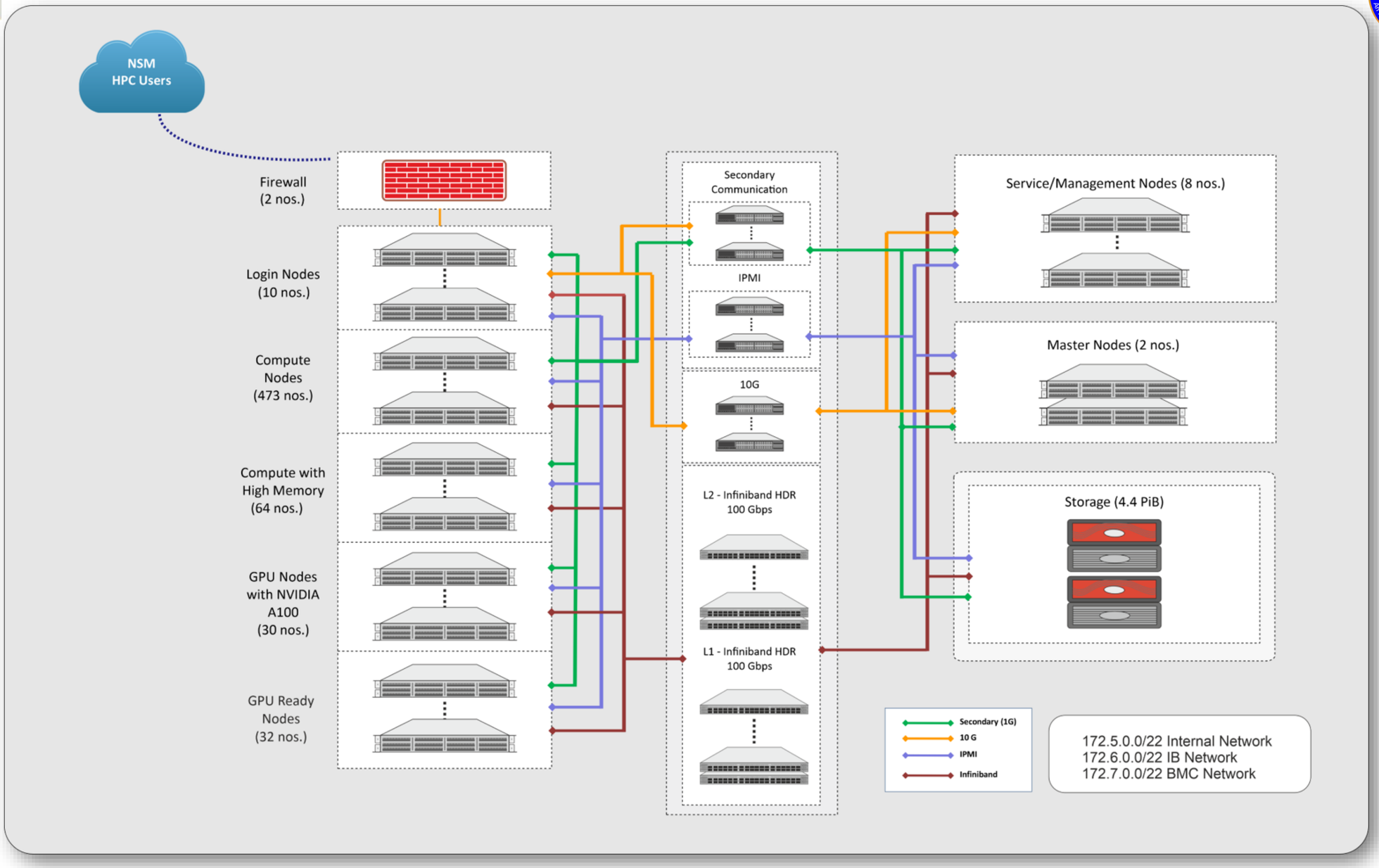
- 27 Nodes
- 1296 CPU Cores
- Compute power of Rpeak 931.12 TFLOPS
- Each Node with
  - 2 X Intel Xeon GOLD 6240R, 24 cores, 2.4 GHz
  - 192 GB Memory
  - 2 x NVIDIA A100
  - 800 GB SSD

GPU Ready Compute Nodes

- 35 Nodes
- 1680 CPU Cores
- Compute power of Rpeak 129.01 TFLOPS
- Each Node with
  - 2 X Intel Xeon GOLD 6240R, 24 cores, 2.4 GHz
  - 192 GB Memory
  - 800 GB SSD

Courtesy: https://nsmindia.in/

PARAM RUDRA - Architecture



Reference: https://nsmindia.in/

Essentials for HPC cluster Architecture

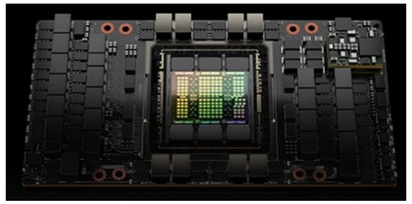
- HPC workloads require CPUs with **high core counts** and **optimized parallel processing.**
- *Examples:* Intel Xeon Platinum 8592 (up to 64 cores), AMD EPYC 9754 (128 Zen 5 cores,) and NVIDIA Grace Superchip (ARM-based, 144 cores)

Courtesy: https://www.nvidia.com/en-in/data-center/grace-cpu/

- HPC and AI workloads require **low-latency, high-bandwidth networking, low CPU overhead** to handle distributed training across thousands of GPUs. HPC has adopted Infiniband, Omni-path, RDMA over converged enhanced ethernet (RoCE)
- *Examples:* NVIDIA BlueField-3 DPU , NVIDIA Quantum-2 InfiniBand

Courtesy: https://www.nvidia.com/en-in/networking/infiniband-adapters/

- AI model training relies on **tensor cores, high memory bandwidth, and parallel processing.**
- *Examples:* NVIDIA H100 Tensor Core GPU (80GB HBM3), AMD Instinct MI300X (192GB HBM3), Intel Gaudi 2 AI Accelerator (96GB HBM2E)



Courtesy: https://www.nvidia.com/en-sg/data-center/h100/

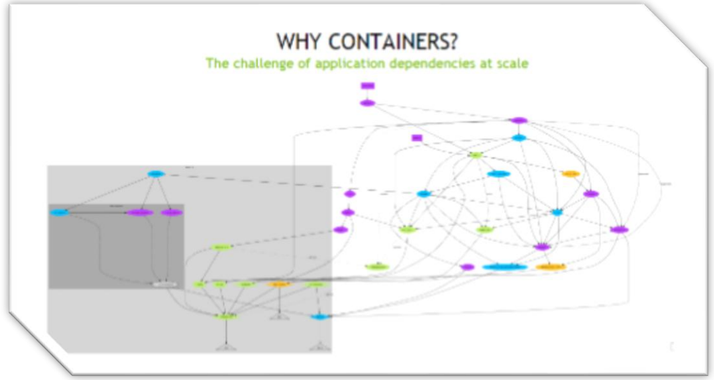
- **Solid-State Drives (SSDs)** provide **low-latency, high-speed storage for AI training, caching**, and **fast data retrieval**.
- *Examples:* Samsung PM1743 (14GB/s Read, 7GB/s Write), Micron 9400 Pro(7GB/s Read, 6GB/s Write)
- **Parallel file systems** are optimized **for large-scale data processing, distributing data** across multiple storage nodes.
- *Examples:* Lustre (Open-source, high-performance, parallel I/O), DAOS (Distributed Asynchronous Object Storage) used in Exascale supercomputers, NVMe-optimized.

Courtesy: https://www.miphi.in/pages/aidaptiv.html

Courtesy: https://www.ddn.com/products/a3i-accelerated-any-scale-ai/

HPC-focused container software

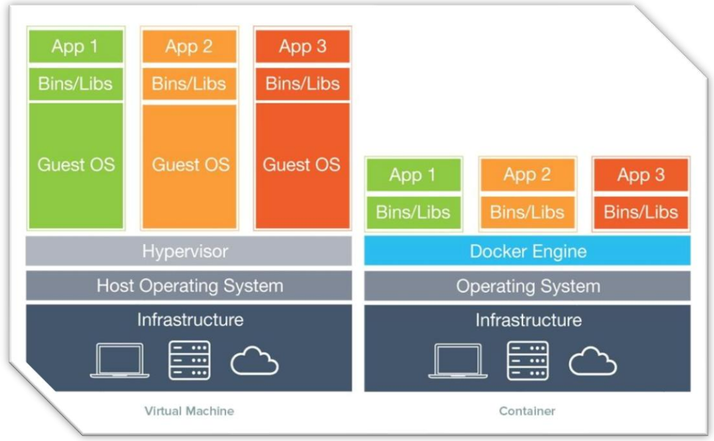
- **Portability:** Applications packaged in containers can run on any HPC system, regardless of the underlying OS.
- **Isolation:** Provide environment isolation, avoiding conflicts between different users' dependencies.
- **Ease of Deployment:** Simplify the deployment of complex applications by bundling dependencies with the application.



Courtesy: https://en.wikipedia.org/wiki/Docker,_Inc.

Linux containers ecosystem

- Linux containers rely on abstraction features (namespaces[1]) provided by the kernel
- Different design decisions and use cases gave rise to several solutions



Courtesy: https://en.wikipedia.org/wiki/Docker,_Inc.

Why Docker and HPC is not a good fit ?

- Security model assumes root privileges
- No integration with workload managers
- Missing support for diskless nodes
- Very limited support for kernel bypassing devices (e.g. accelerators and NICs)
- No adequate parallel storage driver

HPC focused: Singularity, APPTAINER, Charliecloud, SHIFTER

https://sylabs.io/singularity/
https://apptainer.org/
https://hpc.github.io/charliecloud/
https://github.com/NERSC/shifter

[1] "Namespaces in operation,namespaces overview" at https://lwn.net/Articles/531114/

Slurm and Pyxis: Efficient Containerized Workflows for HPC

Slurm (Simple Linux Utility for Resource Management) is an **open-source, fault-tolerant system** for managing and scheduling **Linux clusters**. It efficiently handles **job allocation**, **execution**, and **queuing** in HPC environments.

Key Features:

- **Resource Allocation:** Assigns compute nodes exclusively or shared among users.
- **Job Scheduling & Monitoring:** Manages job queues and runs parallel jobs across multiple nodes.
- **Extensibility:** Supports plugins for authentication, logging, security, energy management, and topology-based scheduling.

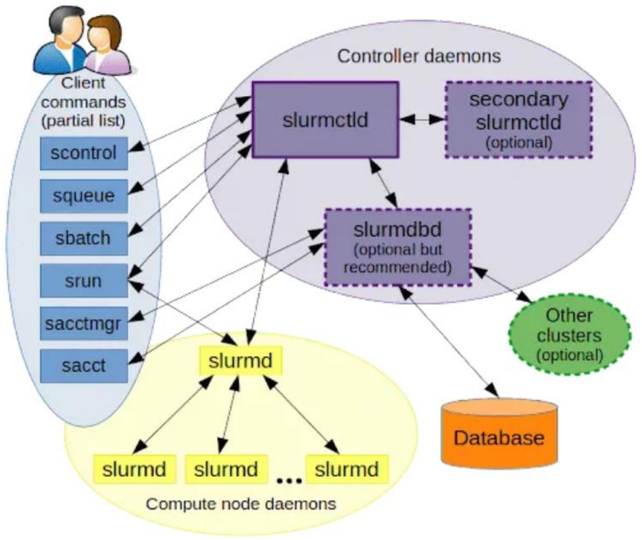
Resource Managers	Scheduler
ALPS (Cray)	Maui
Torque	Moab
LoadLeveler (IBM)	
LSF	
SLURM	
PBS Pro	

https://github.com/SchedMD/slurm
https://github.com/NVIDIA/pyxis

SPANK plugin: Pyxis
Allows unprivileged cluster users to run containerized tasks through the srun command.

Benefits:

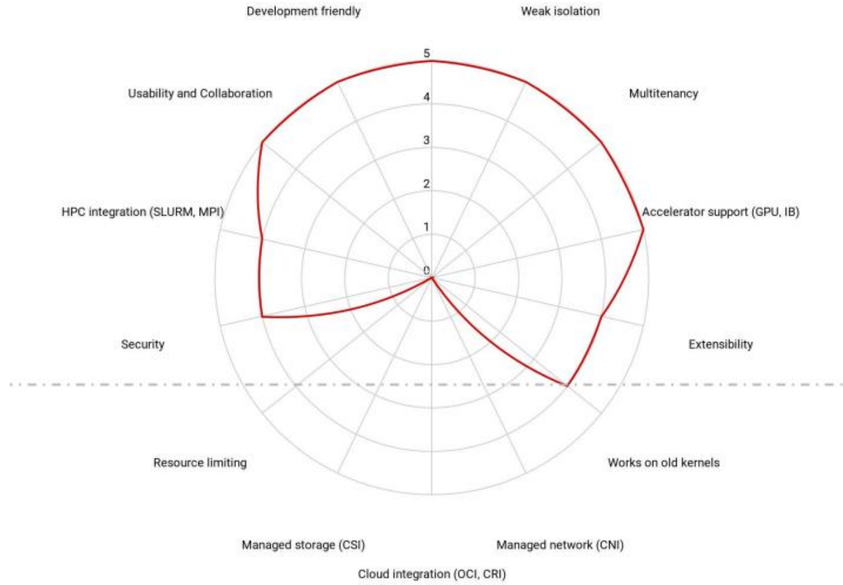
- Seamlessly execute the user's task in an **unprivileged container**.
- Simple command-line interface.
- Fast Docker image download with support for **layers caching** and **layers sharing** across users.
- Supports **multi-node MPI** jobs through PMI2 or PMIx (requires Slurm support).
- Allows users to install packages inside the container.
- Works with shared filesystems.
- Does not require cluster-wide management of subordinate user/group ids.



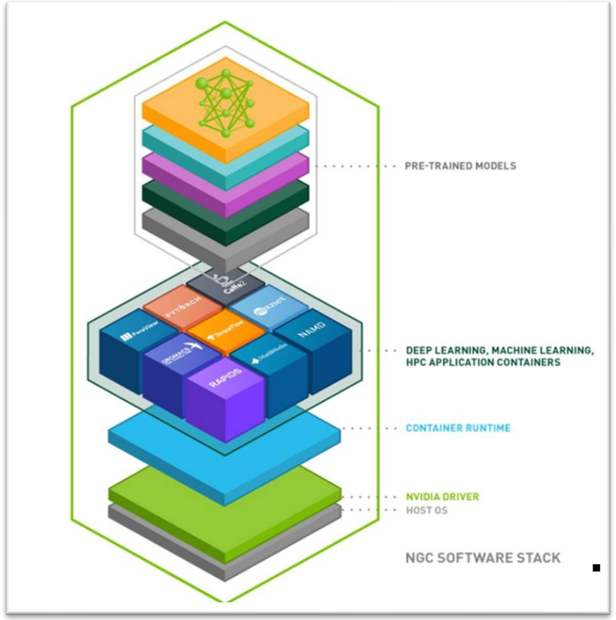
Courtesy: https://www.run.ai/guides/slurm

Accelerate AI & HPC with **Enroot** and NVIDIA **NGC**

- Simple, Lightweight and Open Source - : https://github.com/NVIDIA/enroot
- Runs as an **Unprivileged User-Space Application**
- U**ser-Owned** File Systems
- **Built-in GPU Support** with libnvidia-container
- Works Well with Resource Managers
- Leverages NVIDIA’s Container Resources on NGC
- Compatibility with Docker Images



Courtesy: https://slurm.schedmd.com/SLUG19/NVIDIA_Containers.pdf



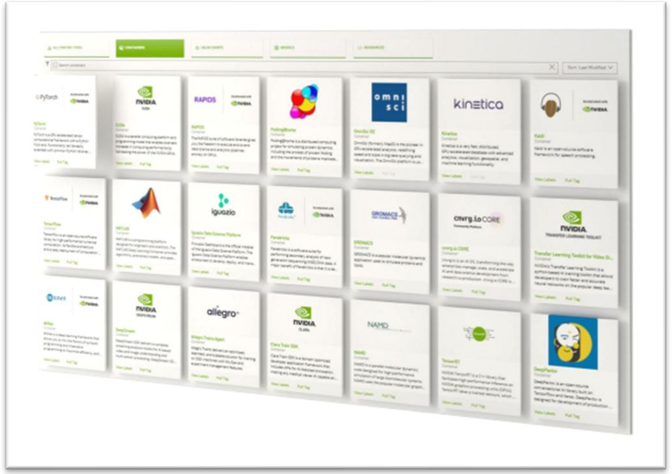
Courtesy: https://developer.nvidia.com/blog/jump-start-ai-training-with-ngc-pretrained-models-on-premises-and-in-the-cloud/

The NVIDIA NGC catalog contains a host of GPU-optimized containers for **deep learning, machine learning, visualization, and high-performance computing (HPC) applications** that are tested for performance, security, and scalability.

Benefits:

- Deploy Easily
- Train Faster
- Run Anywhere
- Deploy with Confidence

- https://catalog.ngc.nvidia.com



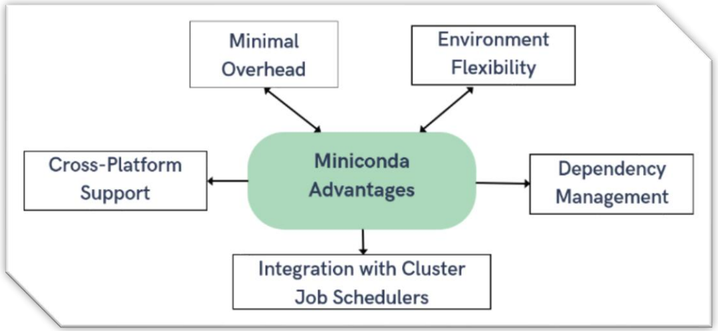
Courtesy: https://developer.nvidia.com/blog/new-on-ngc-new-and-updated-hpc-containers-on-the-ngc-catalog/

Managing **Miniconda** in HPC with **Modules**

Miniconda is a minimalistic distribution of the Conda package manager, specifically designed for users who need a lightweight, flexible tool to manage software environments and dependencies.

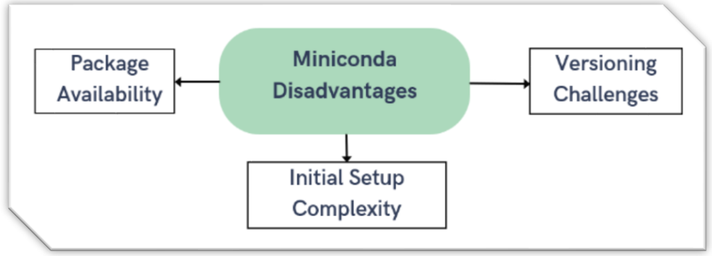
Package Managers:

- **Miniconda** (Minimal Installer)
- Anaconda Distribution (Full Package)
- Miniforge (Minimal Installer)
- **Mamba** (Fast Conda Alternative)



Why Use Mamba?

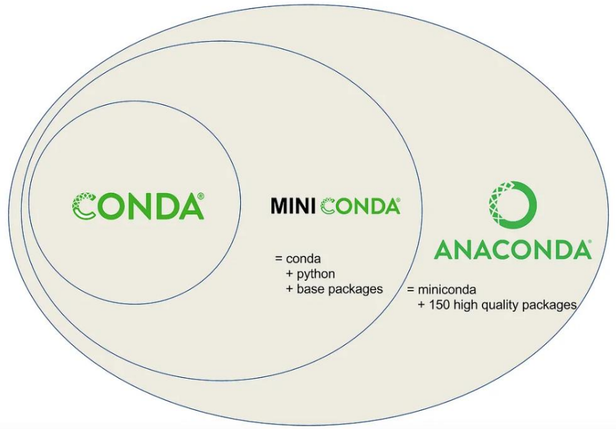
- A faster alternative to Conda.
- Uses libmamba for improved dependency resolution speed.
- Compatible with Conda environments and packages.
- Efficient for HPC (High-Performance Computing) workloads.



The **lmod** modules system on the HPC system enables users to easily set their environments for selected software and to choose versions if appropriate.

Best Practices for Using Modules in HPC

- **Purge before starting** – Run module purge to clear loaded modules.
- **Specify versions** – Avoid using defaults; check versions with module spider.
- **Don't use .bashrc** – Prevent conflicts by not loading modules in .bashrc.
- **Use bash scripts** – Create scripts to load necessary modules.
- **Update regularly** – Keep scripts up to date with new module versions.



Courtesy: https://medium.com/towards-data-science/managing-project-specific-environments-with-conda-b8b50aa8be0e

Hands-on Materials

- To claim the training account:

  https://forms.gle/QhtYJFmxrsWfWMjNA

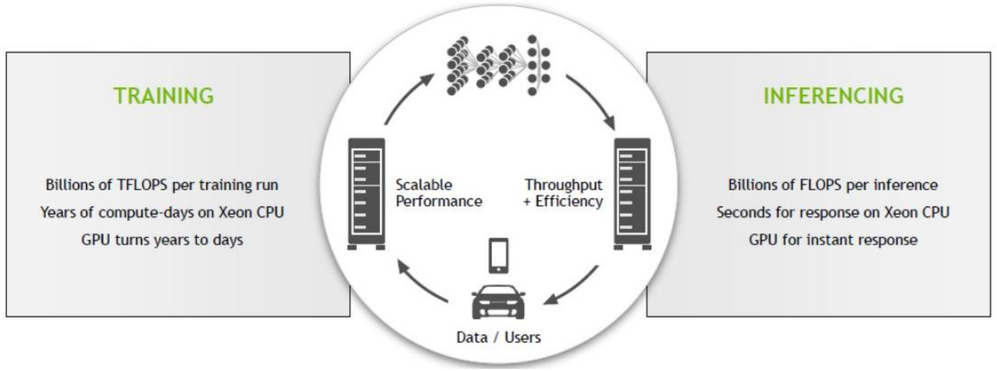
All tutorial materials are available at
https://github.com/snsharma1311/SCA-2025-DistributedTraining

Outline

- **Introduction to the Workshop & Deep Learning on HPC**
  - Workshop Overview
  - Presenters & Contact Information
  - Exponential Growth of AI Models & Compute Power
  - Hands-on Learning Resources (GitHub, Cluster Access)
- **High-Performance Computing (HPC) for AI Workloads**
  - HPC System Architecture & PARAM RUDRA Overview
  - HPC Containerization
  - Job Scheduling & Resource Management with Slurm & Pyxis
  - Running AI Workloads Using NVIDIA NGC & Enroot
- **Deep Learning Training: Single GPU to Distributed Systems**
  - AI Frameworks: PyTorch vs TensorFlow
  - Single GPU Training: Data Loading, Optimization, and AMP
  - Multi-GPU Training: DDP and FSDP
  - Synchronization & Communication in Distributed AI Training
  - Hands-on Session on Distributed Training
- **Advanced Distributed AI Training & Model Optimization**
  - DeepSpeed: Optimizing Large Model Training
  - Enhancing HPC with Containerization
  - Hands-on Session on DeepSpeed and Containers
  - Demonstration of Multi-node Training with and without containers

Why is HPC Essential for Deep Learning ?

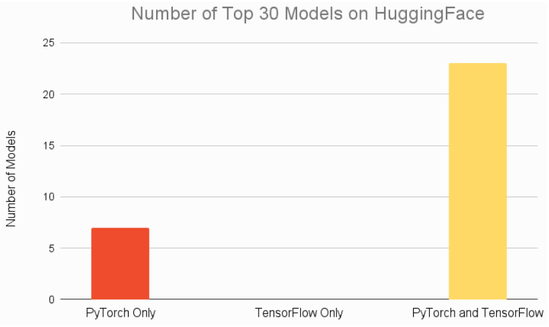
- **High Computational Intensity**
  - State-of-the-art models often have billions of parameters requires significant computational resources.
- **Massive Data Processing**
  - High-dimensional data (e.g., images, videos, text) requires substantial computational resources.
- **Memory and Bandwidth Demands**
  - Large models require significant memory for parameters, intermediate activations, and gradients.
- **Parallelism Requirements**
  - Data parallelism splits data across devices and synchronizes gradients.
  - Model parallelism splits large models across devices to fit into memory.
- **Scalability Challenges**
  - Fault tolerance is essential for handling failures in large-scale training jobs.
  - Cluster management tools like SLURM handle job scheduling and resource allocation.
- **Precision and Numerical Stability**
  - Mixed precision computing (e.g., FP16, BF16) balances performance and accuracy.
  - Specialized hardware like Tensor Cores accelerates mixed precision computations.
- **Need for Robust Software Ecosystems**
  - Distributed deep learning frameworks like PyTorch and TensorFlow support multi-node training.



Courtesy: https://developer.nvidia.com/blog/new-pascal-gpus-accelerate-inference-in-the-data-center/

Deep Learning Frameworks for GPU Acceleration

- **Bridges the Gap Between Hardware & AI Models**
  - Provides high-level APIs that simplify model development while efficiently utilizing GPU resources for accelerated training.
- **Optimizes GPU Performance**
  - Leverages hardware acceleration (CUDA, ROCm, XLA) for optimized computations, reduces memory consumption with advanced techniques.
- **Simplifies Model Implementations**
  - Eliminates the need for manual GPU programming, automates backpropagation and optimization.



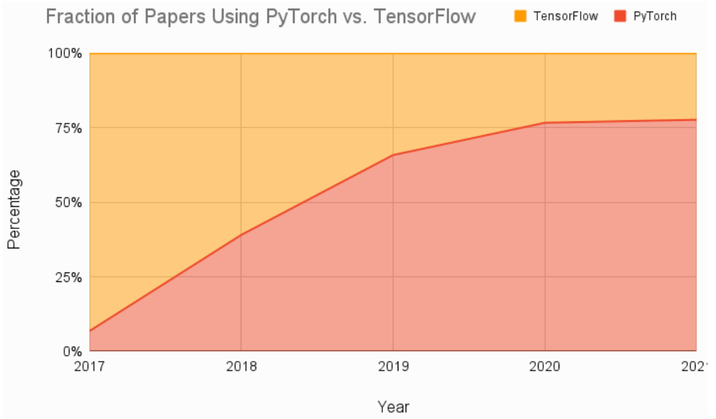
"PyTorch vs. TensorFlow: Which One Should You Use?"

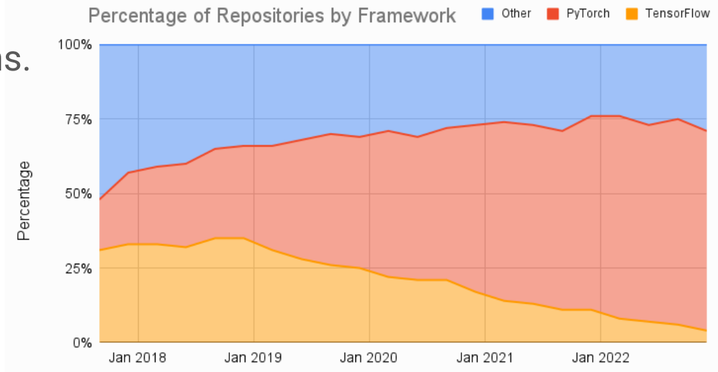
PyTorch (Best for Research & Rapid Prototyping)

- ✅ Better for AI research and experimentation.
- ✅ Dynamic computation graph for flexibility.
- ✅ Easier to use & feels more native to Python.
- ✅ Optimized for GPUs (CUDA support).
- ✅ Fast-growing community with active usage.

TensorFlow (Best for Large-Scale AI & Deployment)

- ✅ Better for large-scale AI deployments & production.
- ✅ Works well for cross-platform & embedded AI applications.
- ✅ Includes a built-in visualizer (TensorBoard).
- ✅ Highly flexible with layered components.
- ✅ Backed by Google, widely used in enterprise AI.

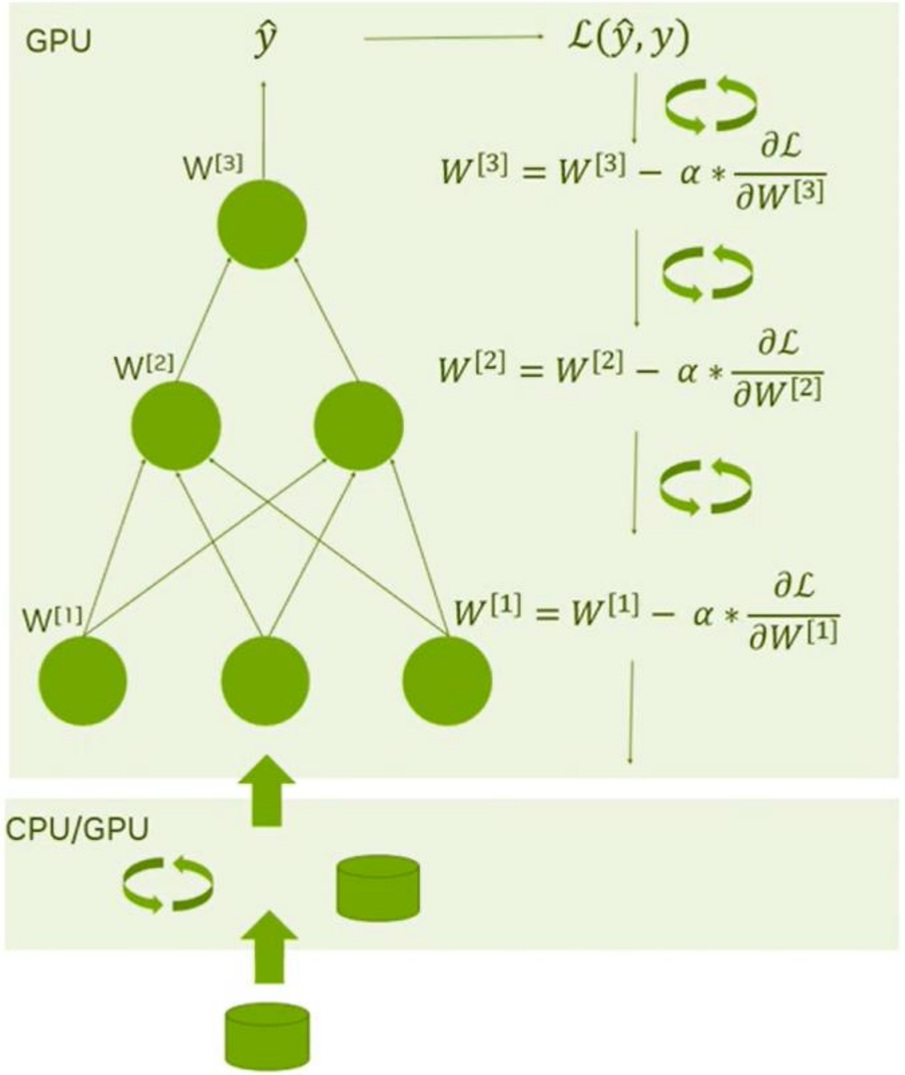




RAPIDS TensorFlow PaddlePaddle deepspeed mxnet PyTorch

Courtesy: PyTorch vs TensorFlow in 2023

Single GPU Training



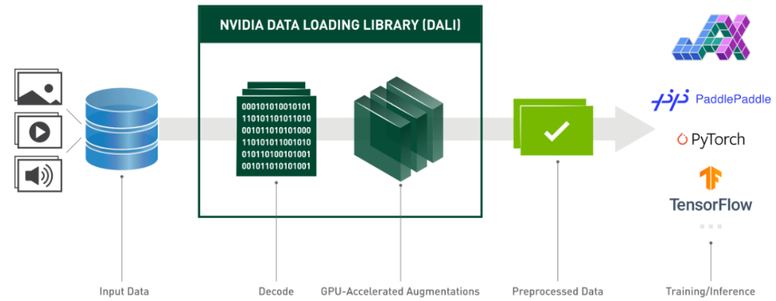
Reference: Data Parallelism Using PyTorch DDP | NVAITC Webinar

1. **Load and Preprocess Data**
   - Preprocess data by applying resizing, normalization, and augmentations.
   - Use a data loader to efficiently load and batch data for training.
2. **Transfer Model and Data to GPU**
   - Ensure that both the model and each batch of data are transferred to the same device to enable GPU acceleration.
3. **Forward Pass**
   - The input batch is passed through the model and predictions are generated.
4. **Compute Loss**
   - The loss function measures the difference between predictions and actual labels, determining how well the model performs.
5. **Backpropagation (Gradient Computation)**
   - The gradients of the loss with respect to each model parameter are computed using automatic differentiation.
6. **Update Model Weights**
   - The optimizer updates the model parameters using the computed gradients to improve performance.

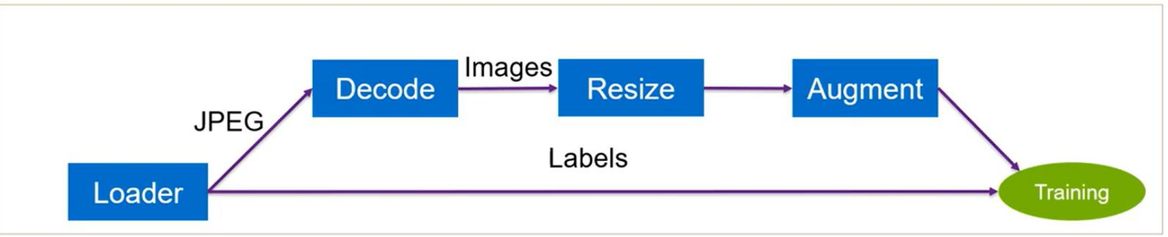
NVIDIA DALI – Efficient Data Loading

High-performance library designed to accelerate **data loading and preprocessing** for deep learning. It replaces CPU-based data pipelines by offloading preprocessing to the GPU, **reducing bottlenecks and improving training speed.**

- **GPU-Accelerated Preprocessing** – Handles loading, decoding, cropping, resizing, and augmentations on the GPU.
- **Faster Training & Inference** – Removes CPU bottlenecks, improving overall performance.
- **Optimized Execution** – Uses parallel execution, prefetching, and batch processing to maximize efficiency.



Courtesy: https://github.com/NVIDIA/DALI

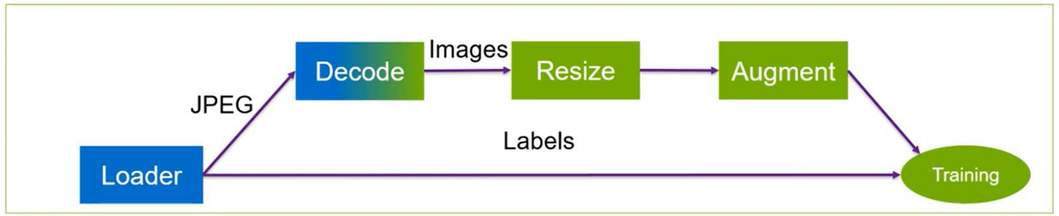


Common pipelines runs on CPU only

Before

CPU

GPU



After

Optimized pipelines can run on CPU and GPU

PyTorch DataLoader

```
data_loader = torch.utils.data.DataLoader(...)
data_fetcher = DataPrefetcher(data_loader)

index = 0
inputs, labels = data_fetcher.next()
while inputs is not None:
   (...)
   index += 1
   inputs, labels = data_fetcher.next()
```

DALI Iterator

```
dali_pipeline = TrainingPipeline(...)
data_loader = DALIClassificationIterator(dali_pipeline)

for index, batch in enumerate(data_loader):
inputs = batch[0]["data"]
labels = batch[0]["label"].squeeze()
(...)
```

Reference: Efficient Data Loading using DALI | NVAITC Webinar

Single GPU Training with AMP

Standard way to represent real numbers on a computer,

- Double precision (FP64), single precision (FP32), half precision (FP16/BF16), Quarter Precision (FP8)

Cannot store numbers with infinite precision, trade-off between range and precision

Benefits of Mixed Precision Training

Faster Computation on Specialized Hardware (GPU Tensor Cores)

- FP16/BF16 provides up to 16x higher throughput than FP32 on GPUs with Tensor Cores.

Efficient Memory Utilization

- 16-bit formats require half the memory bandwidth, reducing memory traffic.

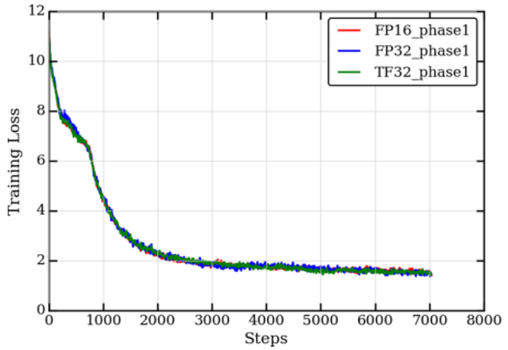
Lower Memory Requirements

- Reduces storage of activations and gradients, freeing up GPU memory.
- Enables training of larger models, bigger mini-batches, and larger input sizes.

```python
import torch

# Creates once at the beginning of training
scaler = torch.cuda.amp.GradScaler()

for data, label in data_iter:
    optimizer.zero_grad()

    # Casts operations to mixed precision
    with torch.cuda.amp.autocast():
        loss = model(data)

    # Scales the loss, and calls backward()
    # to create scaled gradients
    scaler.scale(loss).backward()

    # Unscales gradients and calls
    # or skips optimizer.step()
    scaler.step(optimizer)

    # Updates the scale for next iteration
    scaler.update()
```

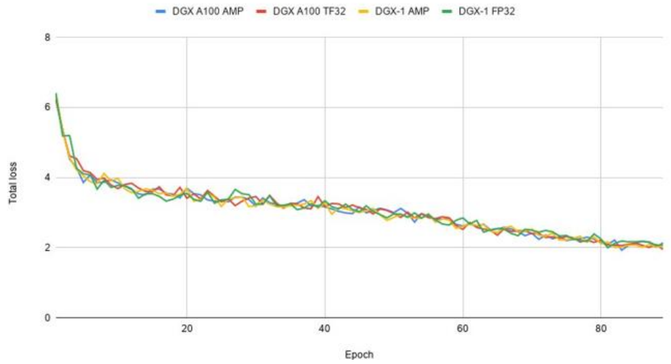
NATIVE AMP FOR PYTORCH

BERT Large Pre-training



Model can be found at: https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT

ResNet50v1.5

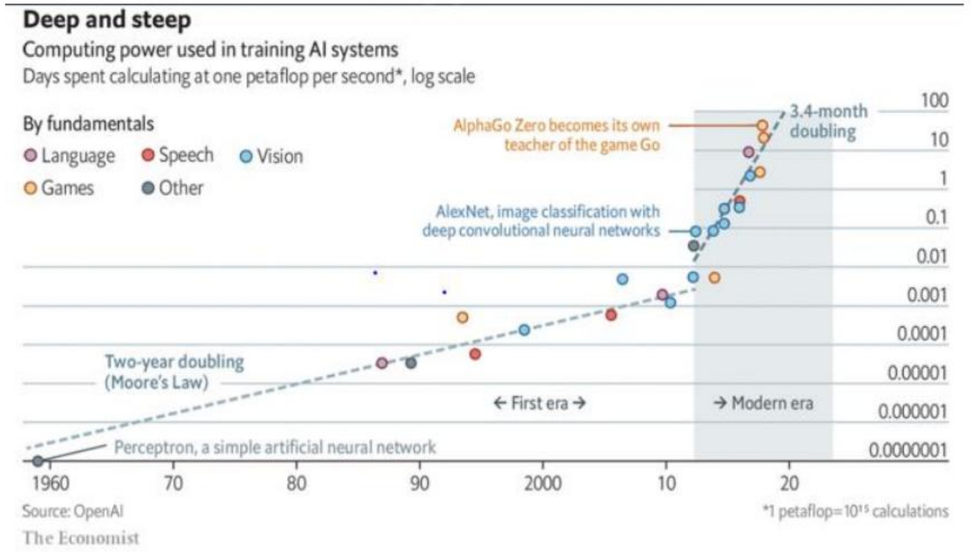


Model can be found at: https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow/Classification/ConvNets/resnet50v1.5 Data collected on NVIDIA A100

Need for distributed Computing

"Since 2012, the amount of compute used in the largest AI training runs has been increasing exponentially with a 3.5 month doubling time (by comparison, Moore's Law had an 18 months doubling period)."

- **Growing model complexity** significantly increases computational demands.
- **Larger datasets** make sequential scanning impractical and inefficient.
- **Advancements in computational power** are driven primarily by **parallel computing** and will continue in this direction.
- **Integrating deep learning with traditional HPC simulations** may necessitate **distributed inference** for efficient processing.



Courtesy: https://openai.com/index/ai-and-compute/?utm_source=chatgpt.com

Example: Training GPT-3

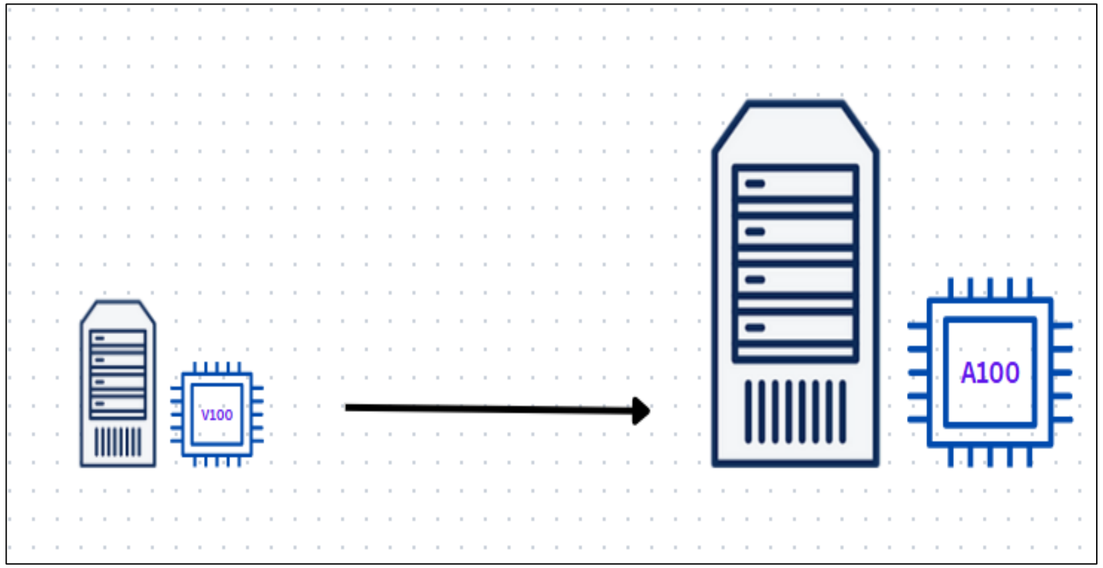
- On a single NVIDIA Tesla V100 GPU → **~355 years**
- On a cluster of 1,024 NVIDIA A100 GPUs → **~34 days**
- Total Compute Time for Training GPT-3 → **~3,640 GPU-days (~9.97 years on a single GPU)**
- Estimated cost of training GPT-3: **$4M – $5M in compute costs.**

Model	Total train compute (PF-days)	Total train compute (flops)	Params (M)	Training tokens (billions)	Flops per param per token	Mult for bwd pass	Fwd-pass flops per active param per token	Frac of params active for each token
T5-Small	2.08E+00	1.80E+20	60	1,000	3	3	1	0.5
T5-Base	7.64E+00	6.60E+20	220	1,000	3	3	1	0.5
T5-Large	2.67E+01	2.31E+21	770	1,000	3	3	1	0.5
T5-3B	1.04E+02	9.00E+21	3,000	1,000	3	3	1	0.5
T5-11B	3.82E+02	3.30E+22	11,000	1,000	3	3	1	0.5
BERT-Base	1.89E+00	1.64E+20	109	250	6	3	2	1.0
BERT-Large	6.16E+00	5.33E+20	355	250	6	3	2	1.0
RoBERTa-Base	1.74E+01	1.50E+21	125	2,000	6	3	2	1.0
RoBERTa-Large	4.93E+01	4.26E+21	355	2,000	6	3	2	1.0
GPT-3 Small	2.60E+00	2.25E+20	125	300	6	3	2	1.0
GPT-3 Medium	7.42E+00	6.41E+20	356	300	6	3	2	1.0
GPT-3 Large	1.58E+01	1.37E+21	760	300	6	3	2	1.0
GPT-3 XL	2.75E+01	2.38E+21	1,320	300	6	3	2	1.0
GPT-3 2.7B	5.52E+01	4.77E+21	2,650	300	6	3	2	1.0
GPT-3 6.7B	1.39E+02	1.20E+22	6,660	300	6	3	2	1.0
GPT-3 13B	2.68E+02	2.31E+22	12,850	300	6	3	2	1.0
GPT-3 175B	3.64E+03	3.14E+23	174,600	300	6	3	2	1.0

Courtesy: https://arxiv.org/pdf/2005.14165

Different Types of Scaling

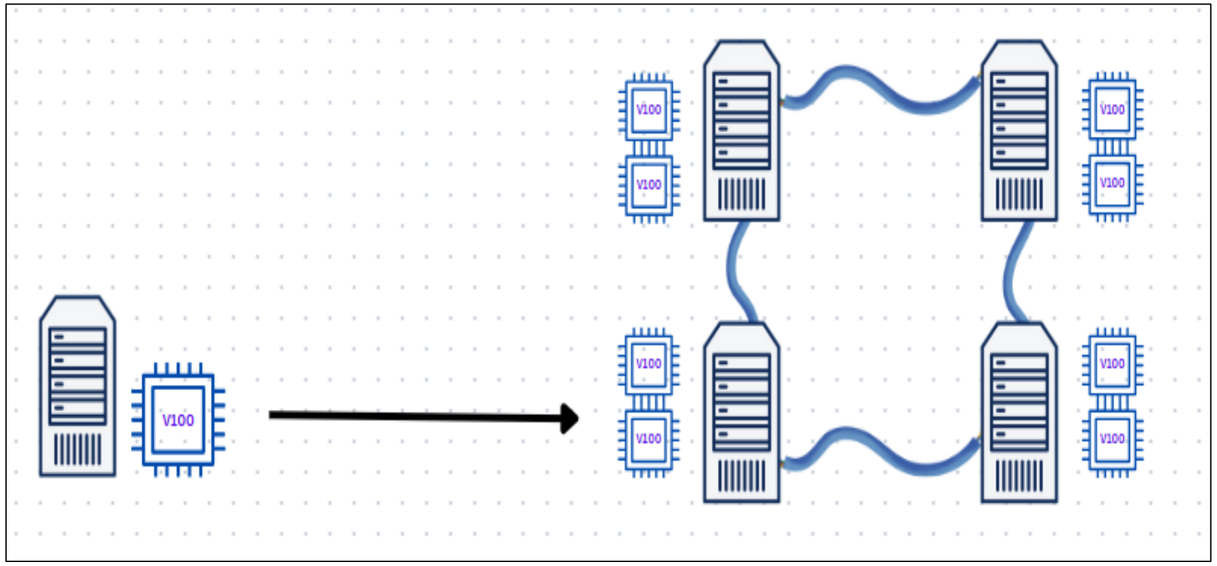
Vertical Scaling



1 x Server
16GB GPU memory

1 x Server
80GB GPU memory

Horizontal Scaling



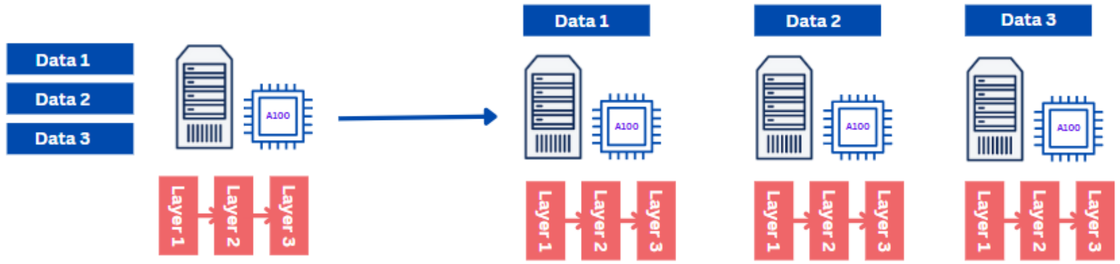
1 x Server
16GB GPU memory

4 x Server
16GB x 2 GPU memory

Parallelization Schemes for Distributed Training

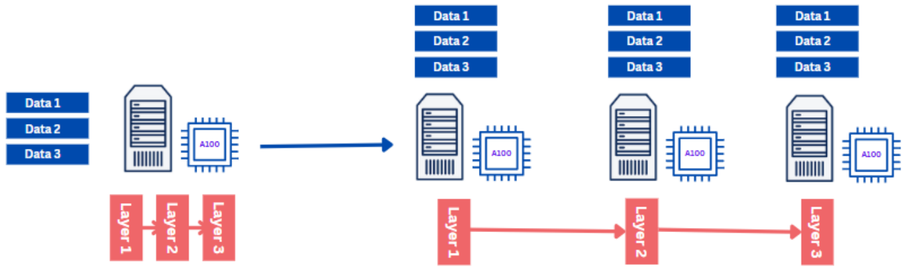
Data Parallelism

If the model can fit within a single GPU, then we can distribute the training on multiple servers (each containing one or multiple GPUs), with each GPU processing a subset of the entire dataset in parallel and synchronizing the gradients during backpropagation. This option is known as Data Parallelism.

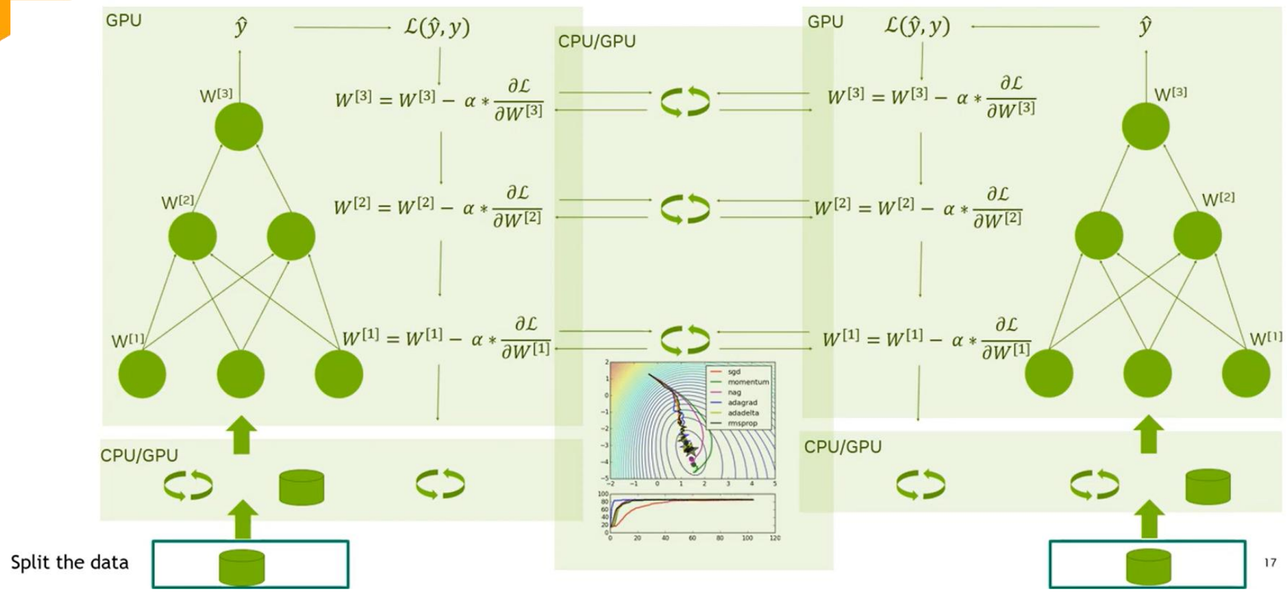


Model Parallelism

If the model cannot fit within a single GPU, then we need to “break” the model into smaller layers and let each GPU process a part of the forward/backward step during gradient descent. This option is known as Model Parallelism.



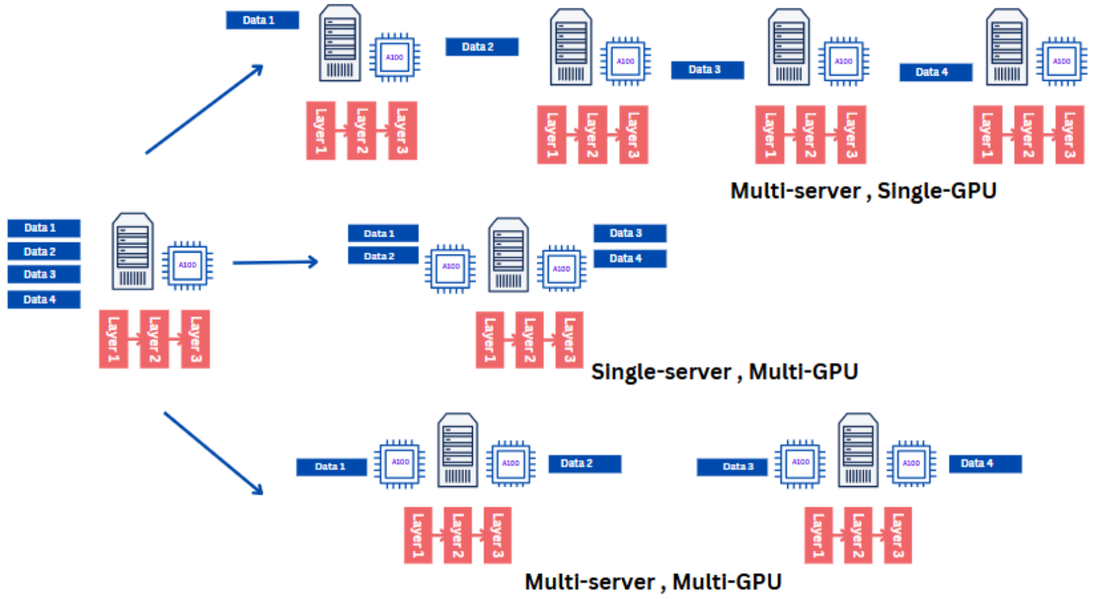
Multi- GPU training



Reference: Data Parallelism Using PyTorch DDP | NVAITC Webinar

Train DL models through many iterations of 3 stages

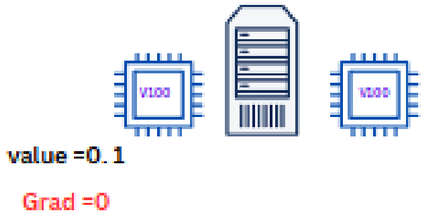
1. **Forward propagation**: apply model to a batch of input samples and run calculation through operators to produce a prediction
2. **Backward propagation**: run the model in reverse to produce a gradient for each trainable weight
3. **Weight update**: use the gradients to update model weights



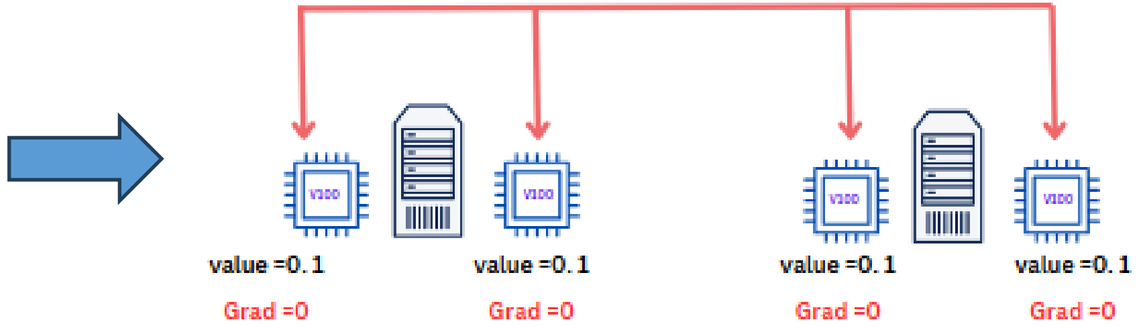
Scaling Deep Learning: Understanding Distributed Data Parallel (DDP)

Distributed Data Parallel: step 1

At the beginning of the training, the model's weights are initialized on one node and sent to all the other nodes (Broadcast)



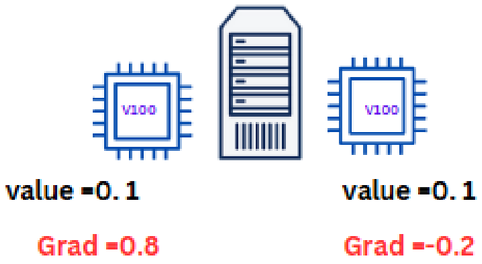
Model weights are initialized here

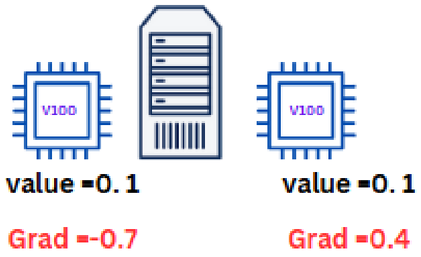


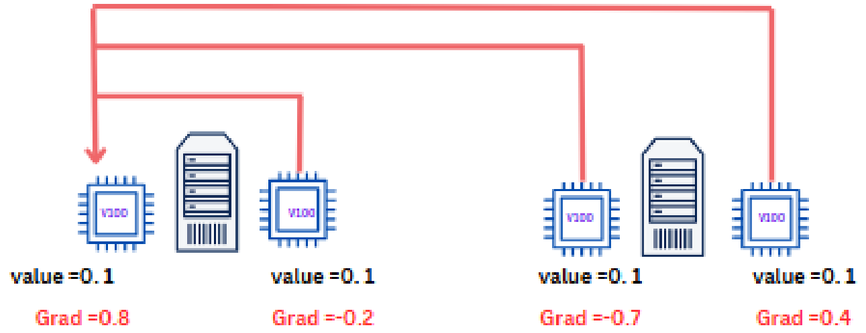
Initial weights are sent to all the other nodes (Broadcast)

Distributed Data Parallel: step 2

Each node runs a forward and backward step on one or more batch of data. This will result in a local gradient. The local gradient may be the accumulation of one or more batches





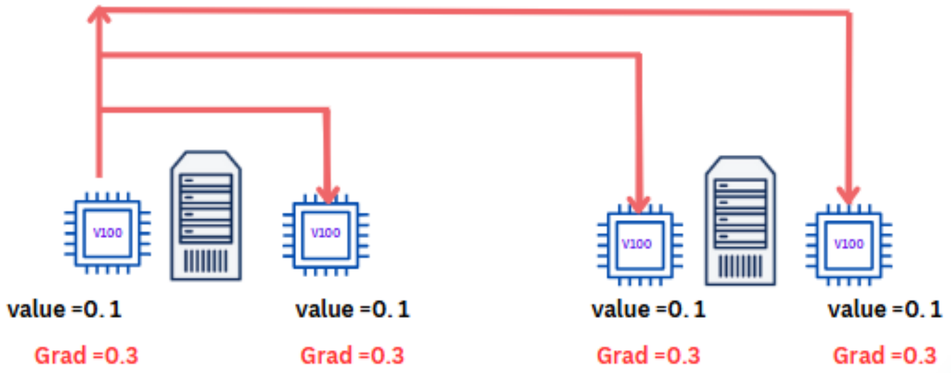


The sum of all gradients is cumulated on one node (Reduce)

Scaling Deep Learning: Understanding Distributed Data Parallel (DDP)

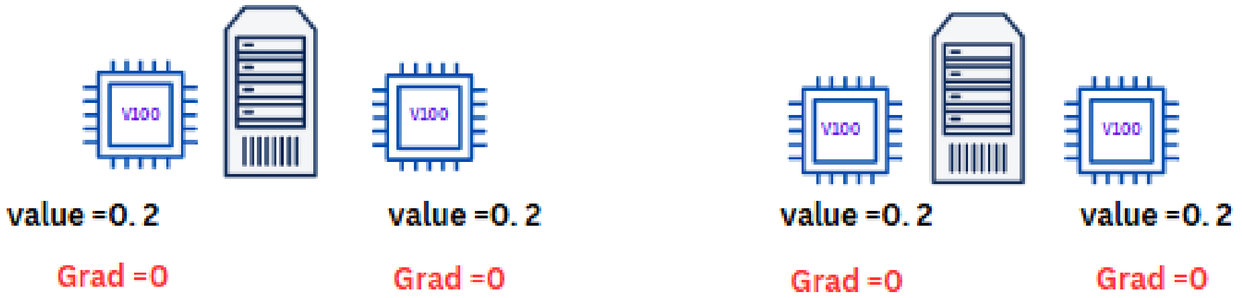
Distributed Data Parallel: step 3

The cumulative gradient is sent to all the other nodes (Broadcast).The sequence of Reduce and Broadcast are implemented as a single operation (All-Reduce).



Distributed Data Parallel: step 4

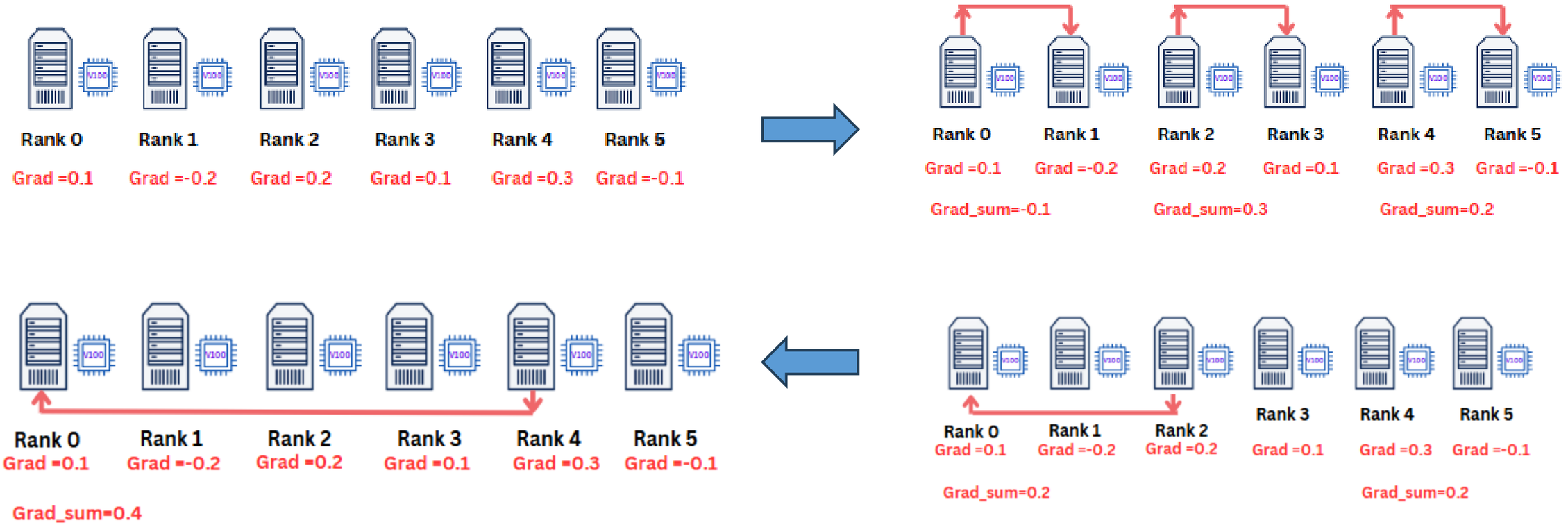
Each node updates the parameters of its local model using the gradient received. After the update, the gradients are reset to zero and we can start another loop



Broadcast

The Broadcast operator is used to send the initial weights to all the other nodes when we start the training loop.

At every few batches of data processed by each node, the gradients of all nodes need to be sent to one node and accumulated (summed up). This operation is known as Reduce.



With only 3 steps we accumulated the gradient of all nodes into one node. It can be proven that the communication time is logarithmic w.r.t the number of nodes .

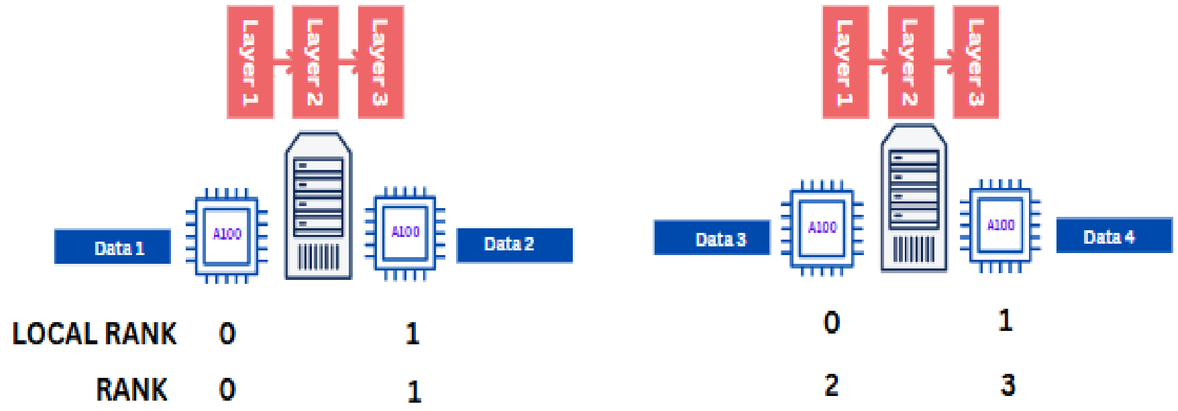
Collective Communication: All-Reduce

Having accumulated the gradients of all the nodes into a single node, we need to send the cumulative gradient to all the nodes. This operation can be done using a Broadcast operator. The sequence of Reduce-Broadcast is implemented by another operator known as All-Reduce, whose runtime is generally lower than the sequence of Reduce followed by a Broadcast.

LOCAL_RANK vs RANK

The environment variable LOCAL_RANK indicates the ID of the GPU on the local computer, while the RANK variable indicates the a globally unique ID among all the nodes in the cluster.

Please note that ranks are not stable, meaning that if you restart the entire cluster, a different node may be assigned the rank number 0.

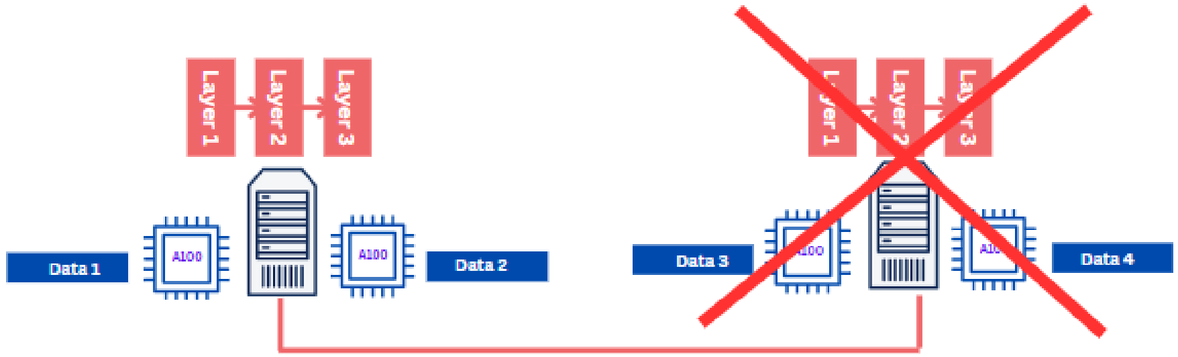


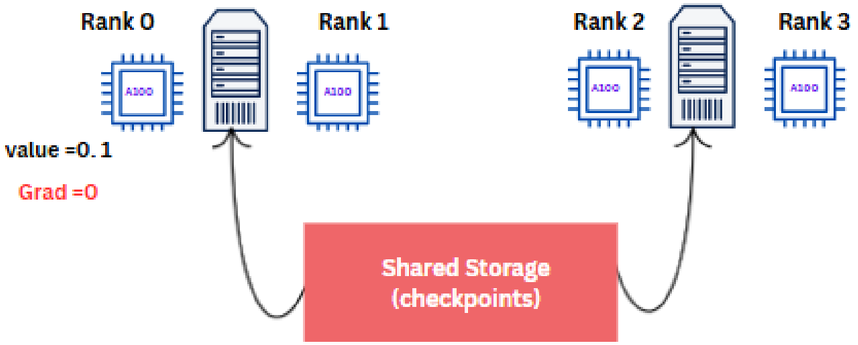
Failover: what happens if one node crashes?

Imagine you're training in a distributed scenario like the one shown below and one of the nodes suddenly crashes.
One way, would be to restart the entire cluster and that's easy. However, by restarting the cluster, the training would restart from zero, and we would lose all the parameters and computation done so far. A better approach is to use checkpointing.

We need a shared storage because PyTorch will decide which node will initialize the weights and we should make no assumption on which one will it be. So, every node should have access to the shared storage.

When we start the cluster, PyTorch will assign a unique ID (RANK) to each GPU. We will write our code in such a way that whichever node is assigned the RANK 0 will be responsible for saving the checkpoint, so that the other nodes do not overwrite each other's files. So only one node will be responsible for writing the checkpoints and all the other files we need for training.

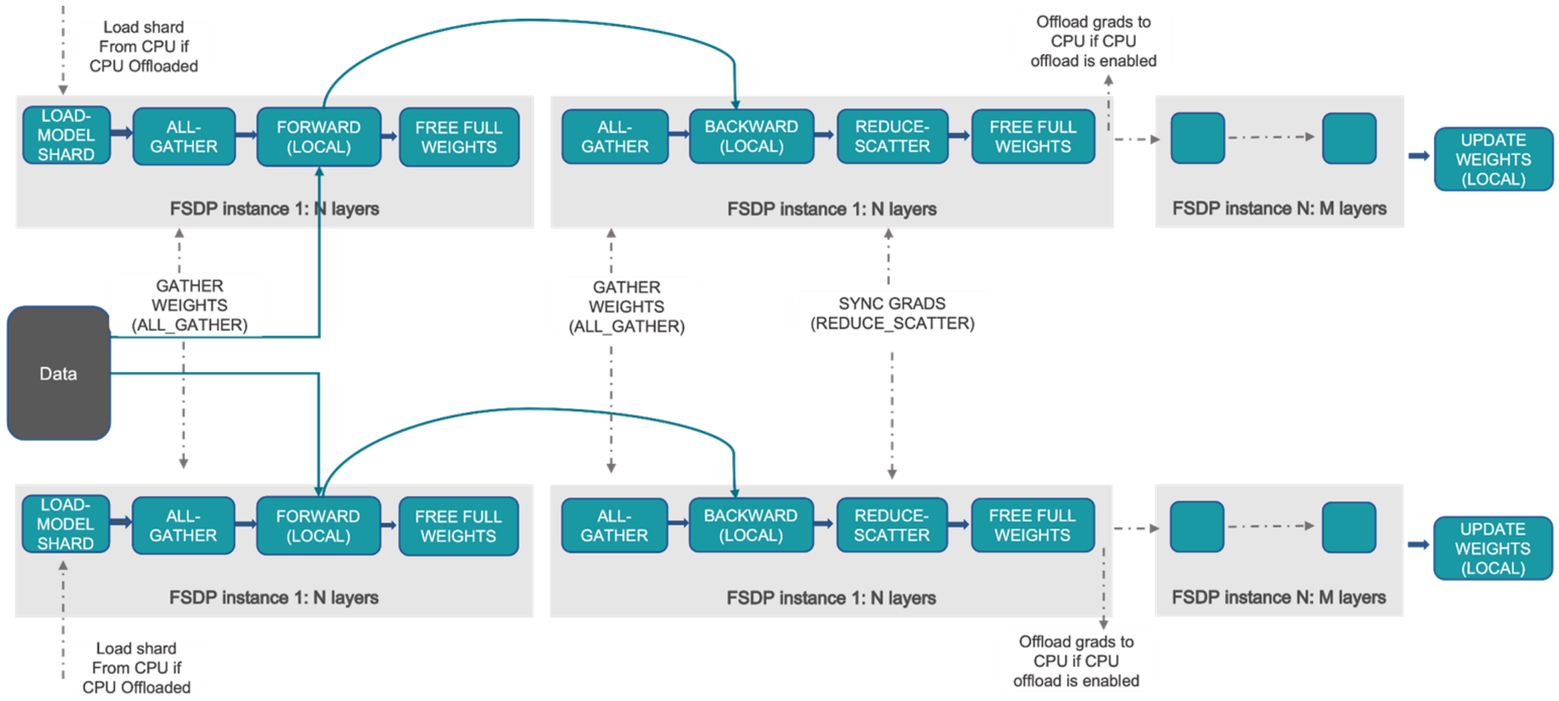




Sharded Data Parallelism

Fully Sharded Data Parallel (FSDP) is a feature in PyTorch that addresses the limitations of data parallelism by allowing more efficient utilization of memory and compute resources. FSDP shards (i.e., splits) both the model parameters and optimizer states across all available GPUs, significantly reducing memory usage and enabling the training of very large models that otherwise wouldn't fit into GPU memory.

Fully Sharded Data Parallel (FSDP) is primarily a form of model parallelism. However, it incorporates elements of data parallelism as well.



Courtesy: https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

- Communicate parameters only when needed
- Simpler than model parallelism
- Beneficial when weights & optimizer states are a significant portion of memory footprint, but less effective when memory usage is dominated by data + activations!

Can enable trillion parameter models without model-parallelism!

https://arxiv.org/abs/1910.02054

Performance Considerations for Data Parallel Training

- **Select the Appropriate Libraries and Backends**

  Choose the best communication library for your hardware, e.g., NCCL is well-suited for NVIDIA GPUs.

- **Optimize Communication Efficiency**

  Use bucketized communication to minimize the number of collective operations, such as allreduce.

  Adjust the bucket size (bucket_cap_mb) for better performance—default is 25MB, but tuning it can improve efficiency.

- **Improve I/O Performance at Scale**

  High-performance computing (HPC) file systems, such as Lustre, may not provide optimal performance for random small reads at large scale, especially under heavy workloads.

  Use data staging and partitioning, storing datasets on local SSDs or RAM to reduce I/O bottlenecks.

Summary

- **Distributed training** is essential for scaling up large and complex models or datasets efficiently.
- Data parallelism is an effective and straightforward way to accelerate training across multiple GPUs.
- **Fully Sharded Data Parallel (FSDP)** enables efficient training by sharding model parameters, gradients, and optimizer states across devices, reducing memory consumption.
- **Large batch sizes** can lead to instability and reduced generalization if hyperparameters are not properly optimized.
- **Mixed-precision training** improves efficiency by reducing memory usage and speeding up computations.
- **Learning rate warm-up** and **linear scaling** are useful techniques for scaling training to moderate levels. but results may vary by model.
- **Checkpointing** and recovery strategies are critical for fault tolerance in long-running distributed training.
- **Efficient data loading** and preprocessing help prevent I/O bottlenecks in large-scale training.
- State-of-the-art methods and best practices continue to evolve, requiring continuous adaptation and experimentation.

Outline

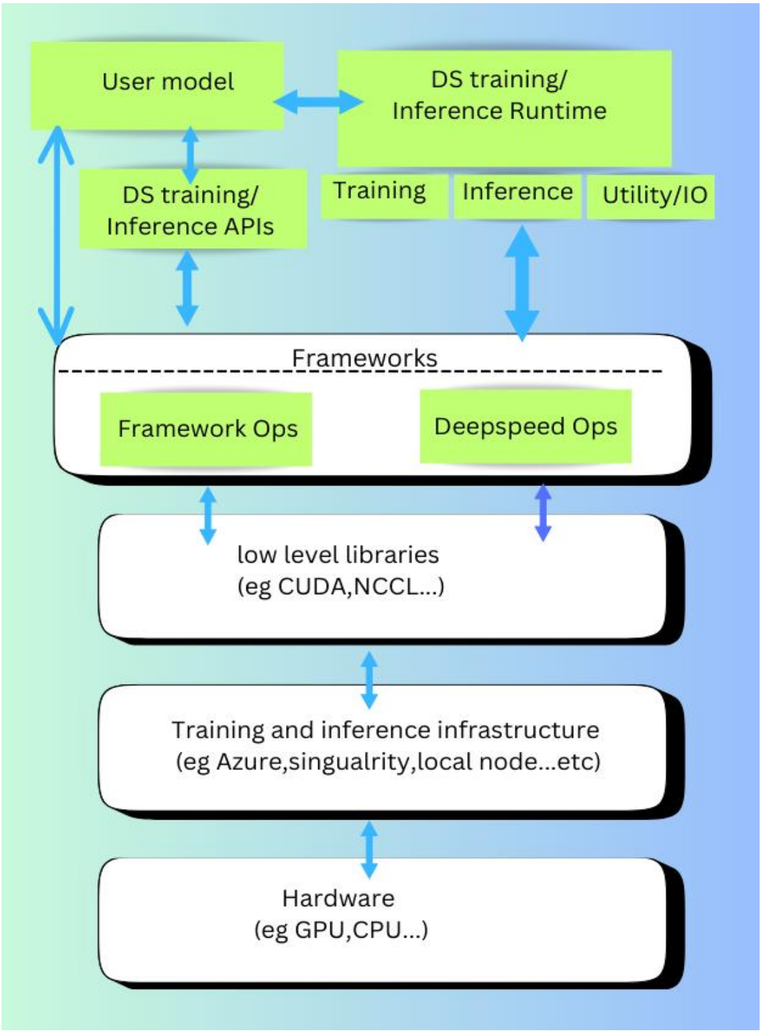
- **Introduction to the Workshop & Deep Learning on HPC**
  - Workshop Overview
  - Presenters & Contact Information
  - Exponential Growth of AI Models & Compute Power
  - Hands-on Learning Resources (GitHub, Cluster Access)
- **High-Performance Computing (HPC) for AI Workloads**
  - HPC System Architecture & PARAM RUDRA Overview
  - HPC Containerization
  - Job Scheduling & Resource Management with Slurm & Pyxis
  - Running AI Workloads Using NVIDIA NGC & Enroot
- **Deep Learning Training: Single GPU to Distributed Systems**
  - AI Frameworks: PyTorch vs TensorFlow
  - Single GPU Training: Data Loading, Optimization, and AMP
  - Multi-GPU Training: DDP and FSDP
  - Synchronization & Communication in Distributed AI Training
  - Hands-on Session on Distributed Training
- **Advanced Distributed AI Training & Model Optimization**
  - DeepSpeed: Optimizing Large Model Training
  - Enhancing HPC with Containerization
  - Hands-on Session on DeepSpeed and Containers
  - Demonstration of Multi-node Training with and without containers

What is DeepSpeed ?

- Easy-to-use deep learning optimization software suite for both training and inference for large-scale deep learning models.
- Developed by Microsoft
- Focuses on scalability, speed, and resource efficiency.
- It solves challenges like memory bottlenecks, long training times, and inefficient GPU utilization.

DeepSpeed Website : https://www.deepspeed.ai/

DeepSpeed GitHub : https://github.com/deepspeedai

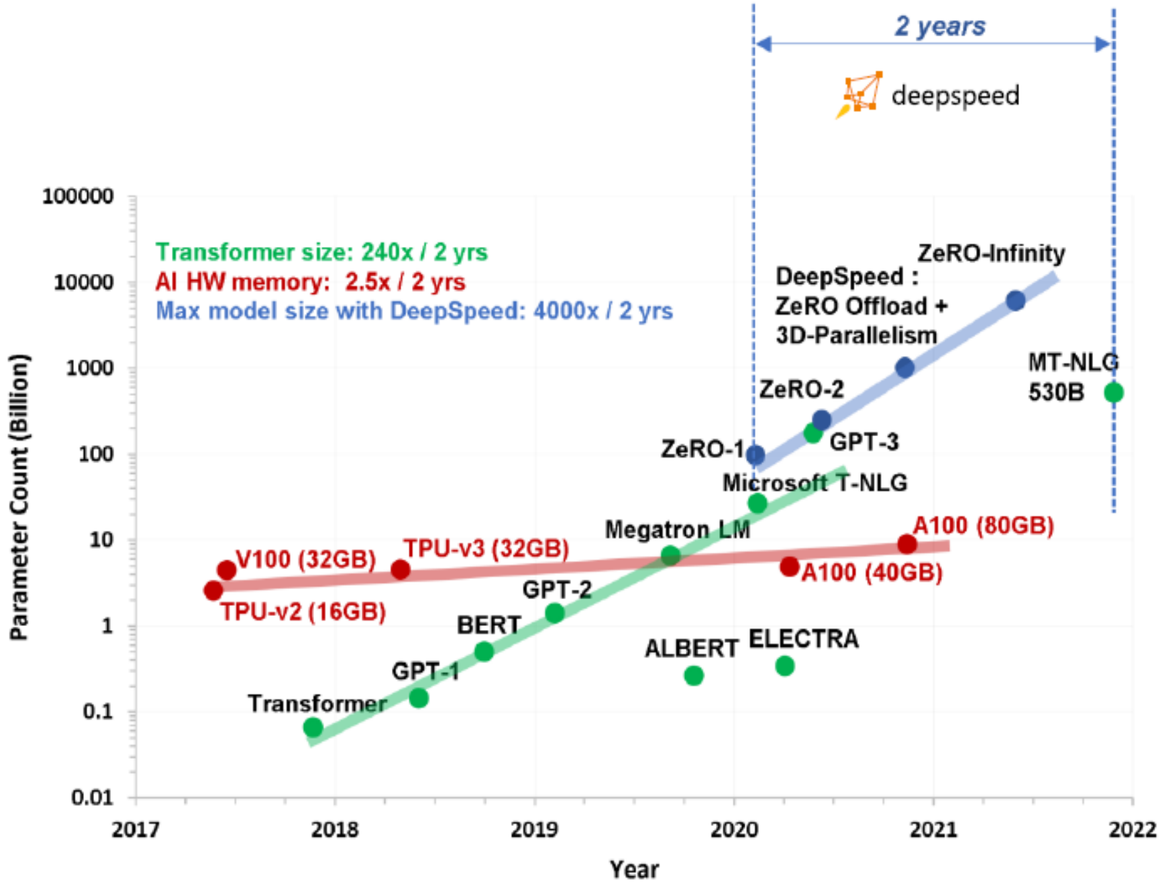


Reference: https://github.com/deepspeedai/DeepSpeed/blob/master/docs/assets/files/presentation-mlops.pdf

DeepSpeed: Model Scaling

Key Training Technologies

- Zero Redundancy Optimizer
- ZeRO - Infinity
- 3D Parallelism
- Distributed Training with mixed precision
- Sparse Attention
- Progressive layer drop



Courtesy: https://github.com/deepspeedai/DeepSpeed/blob/master/docs/assets/files/presentation-mlops.pdf

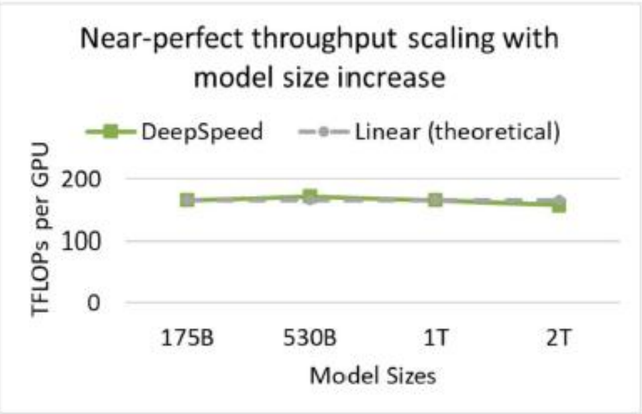
DeepSpeed : Speed

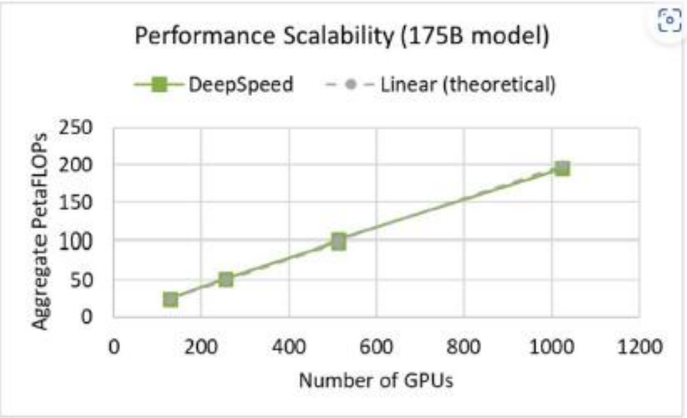
- DeepSpeed was the fastest to train the BERT model in 44 minutes on a cluster of 1024 Nvidia V100 GPUs
- Linear speedup with Increasing GPUS
- Currently DeepSpeed empowers ChatGPT-like model training with a single click, offering 15x speedup over SOTA RLHF systems

World's fastest Bert training

#Devices	Source	Training Time
256 V100 GPUs	Nvidia	236 mins
256 V100 GPUs	DeepSpeed	144 mins
1024 TPU3 chips	Google	76 mins
1024 V100 GPUs	Nvidia	67 mins
1024 V100 GPUs	DeepSpeed	44 mins

Throughput Scaling on 1024 A100 Cluster





Courtesy: https://github.com/deepspeedai/DeepSpeed/blob/master/docs/assets/files/presentation-mlops.pdf

DeepSpeed : Democratize AI

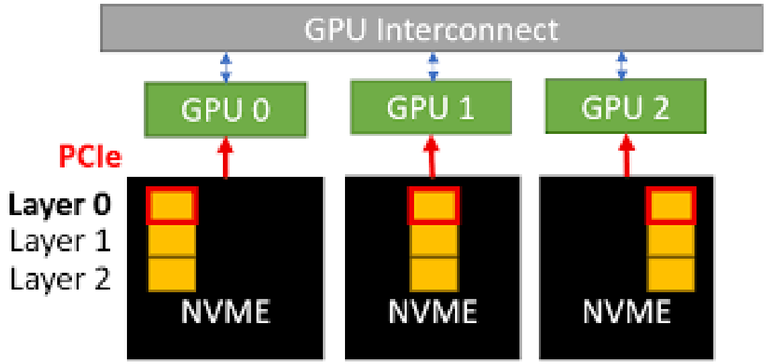
Zero Redundancy Optimizer

- Memory efficient form of Data Parallelism
- Each GPU stores a mutually exclusive subset of the parameters
- Broadcast parameters from owner to all the other GPUS as needed

Courtesy: https://oracle-oci-ocas.medium.com/zero-redundancy-optimizers-a-method-for-training-machine-learning-models-with-billion-parameter-472e8f4e7a5b

Zero Infinity

- A technology that leverages GPU, CPU, and NVMe memory to allow for unprecedented model scale on limited resources without requiring model code refactoring.



Courtesy: https://github.com/deepspeedai/DeepSpeed/blob/master/docs/assets/files/presentation-mlops.pdf

DeepSpeed : Usability

- Only few lines of code changes to enable DeepSpeed on PyTorch models
- Scalable and convenient data parallelism
- HuggingFace and PyTorch Lightning has integrated DeepSpeed as a performance-optimized backend

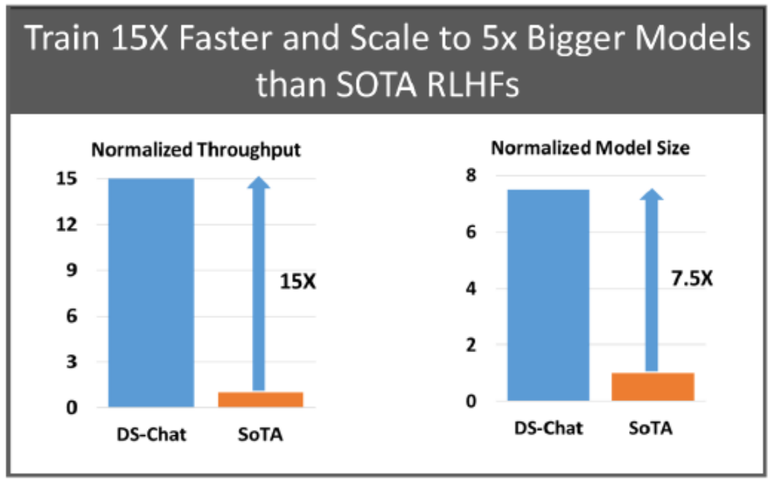
```python
# construct torch.nn.Module
model = MyModel()

# wrap w. DeepSpeed engine
engine, *_ = deepspeed.initialize(
    model=model,
    config=ds_config

# training-loop w.r.t. engine
for batch in data_loader:
  loss = engine(batch)
  engine.backward(loss)
  engine.step()
```

```python
ds_config = {
  "optimizer": {
    "type": "Adam",
    "params": {"lr": 0.001}
  },
  "zero": {
    "stage": 3,
    "offload_optimizer": {
        "device": "[cpu|nvme]"
    },
    "offload_param": {
        "device": "[cpu|nvme]"
    }
  }
}
```

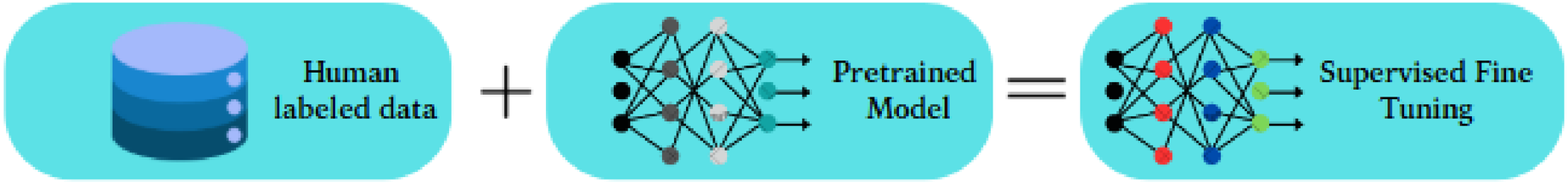
Courtesy: https://github.com/deepspeedai/DeepSpeed/blob/master/docs/assets/files/presentation-mlops.pdf

DeepSpeed-Chat

- Microsoft’s DeepSpeed-Chat provides an end-to-end framework for training ChatGPT-style models
- It can train models with hundreds of billions of parameters in record time using only commodity GPUs.
- DeepSpeed-Chat’s optimized DeepSpeed-RLHF training system combines innovations in memory optimization ,parallelism and other techniques to achieve unparalleled efficiency and scalability.



Courtesy : https://github.com/deepspeedai/DeepSpeed/blob/master/docs/assets/files/presentation-mlops.pdf



Enhancing HPC with Containerization

Why do we need containers ?

- **Portability & Reproducibility** – Run applications consistently across different HPC clusters.
- **Dependency Management** – Avoid software conflicts by bundling required libraries and tools.
- **Lightweight & Fast** – Minimal overhead, faster startup than VMs.
- **Secure Multi-User Execution** – Isolates environments without requiring root access.
- **Scalability & Parallelism** – Easily deploy workloads across thousands of nodes.

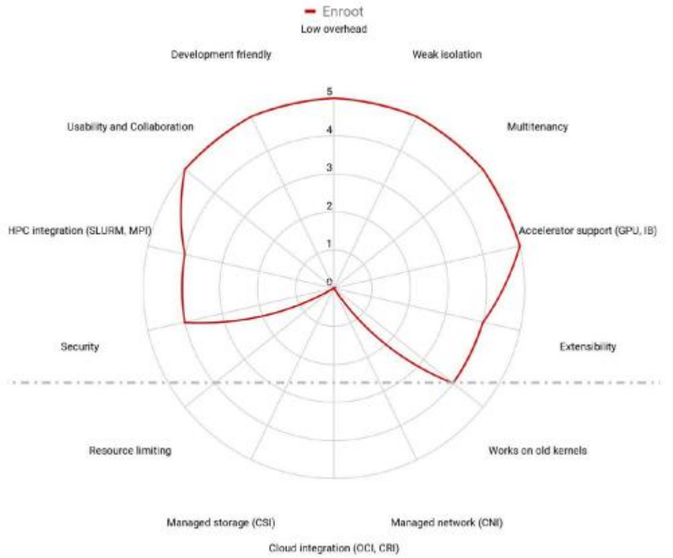
Courtesy : https://www.snappyflow.io/blog/containerization-when-do-you-need-it-and-why

What do we need in containers for HPC ?

- Support for Docker images
- Exposing NVIDIA GPUs and Mellanox InfiniBand cards inside containers
- Resources (CPU/Mem/GPUs/HCAs) isolation
- Launching multi-node jobs
- Development workflow: interactive jobs, installing packages, debugging

Enroot: Lightweight and Secure Containerization for HPC

- **Unprivileged Sandboxing** : Runs applications in a separate environment without root privileges.
- **Filesystem Separation** : Uses chroot-like behavior to isolate files but doesn’t enforce strict process isolation.
- **Lightweight & Fast** : Avoids the overhead of full-fledged containers like Docker, making it ideal for HPC and virtualized environments.
- **Supports Container Formats** : Can import Docker images and convert them into enroot environments.
- **Flexible plugins** : including support for NVIDIA and Mellanox devices



Courtesy: https://slurm.schedmd.com/SLUG19/NVIDIA_Containers.pdf

Enroot : http://github.com/nvidia/enroot
Slurm plugin : http://github.com/nvidia/pyxis

thank you

https://x.com/cdacindia

https://www.linkedin.com/company/cdacindia

{Shashank.sharma, ssowmya, kishoryd, anandhun}@cdac.in

MINISTRY OF
ELECTRONICS &
INFORMATION TECHNOLOGY
GOVERNMENT OF INDIA

सत्यमेव जयते

NATIONAL SUPERCOMPUTING MISSION
INFRASTRUCTURE | APPLICATIONS | R&D | HRD