OAK RIDGE National Laboratory | LEADERSHIP COMPUTING FACILITY

CUG 2025 | May 8, 2025

Performance Evaluation of Containerized ML & LLM Applications on Odo & Frontier

Bishwo Dahal, Elijah MacCarthy, Subil Abraham

U.S. DEPARTMENT OF ENERGY

ORNL IS MANAGED BY UT-BATTELLE LLC
FOR THE US DEPARTMENT OF ENERGY

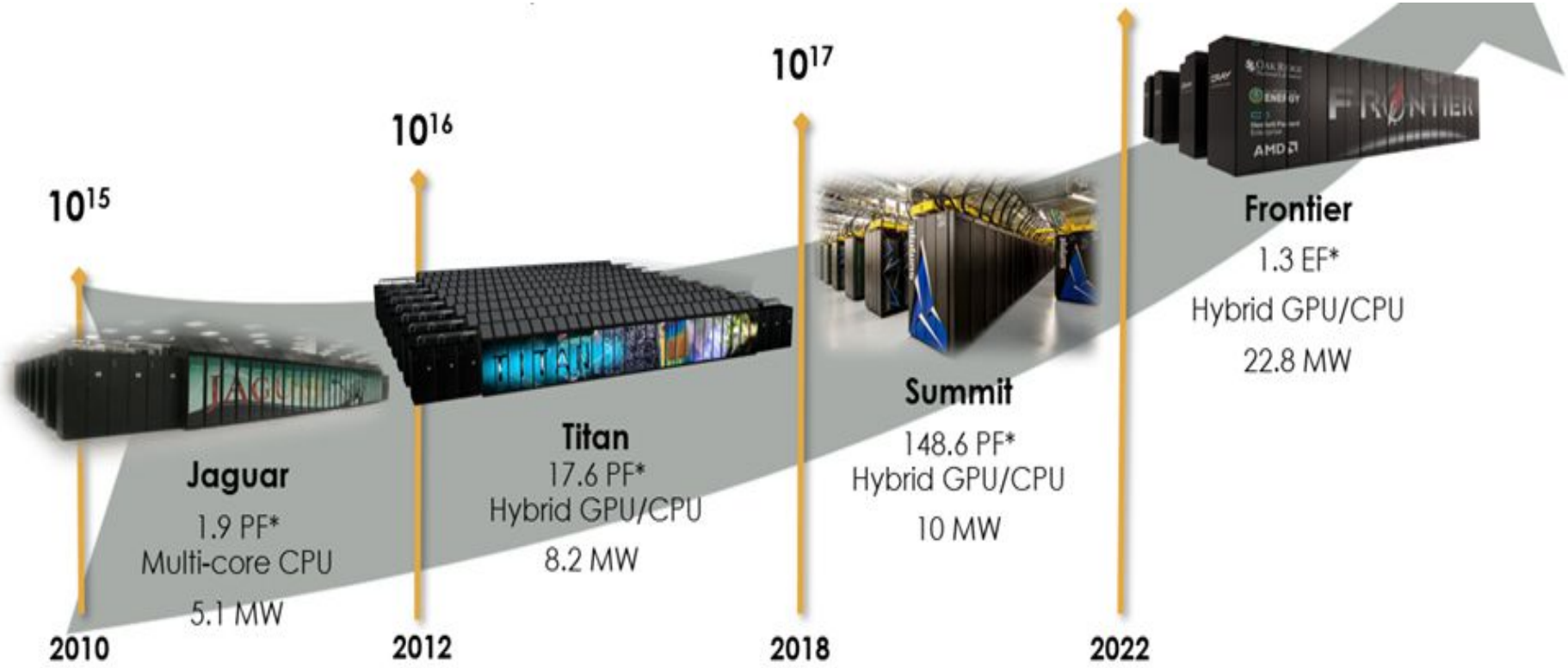
Outline

- Overview of Container Technology
- Applications of Containers to ML and the Sciences
- Performance Evaluation of Containers in Machine Learning
- Performance Evaluation of Containers in Large Language Modeling
- Conclusion and Future Work

Oak Ridge Leadership Computing Facility (OLCF)

- One of two Department of Energy LCF's
- Based in Oak Ridge, TN at the Oak Ridge National Laboratory (ORNL)
- Deployed leadership class HPC systems
- Largest center for open science research in the US
- First supercomputer to break the exascale barrier on the Top500 list.

OLCF Deployed Systems (Top500 Spots)



Overview of Container Technology and Its Applications

- Packages applications with all their dependencies into lightweight portable units.
- Much lighter and resource efficient compared to Virtual Machines.
- Isolated especially for multi-user systems like HPC
- Fast, portable, reproducible and scalable with tools like Kubernetes.
- Could be integrated with CI/CD
- Strong community and support

Motivation

- Containerized ML/LLM workflows are still at the early stages on Frontier.
- A guide is needed for containerizing ML and LLM applications on Frontier.
- Performance evaluation of containerized applications and native applications is essential.
- Growing demand of scalable ML and LLM in the market.

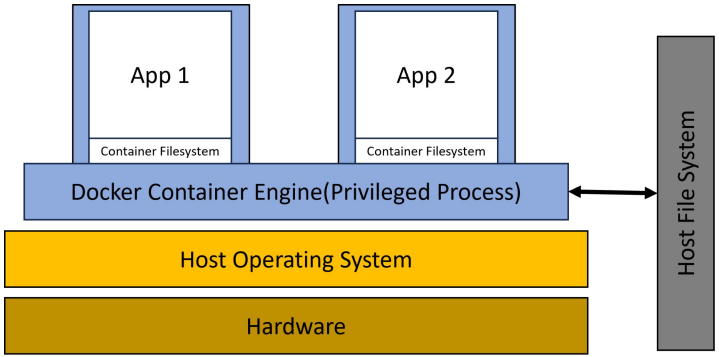
Docker and Apptainer in HPC

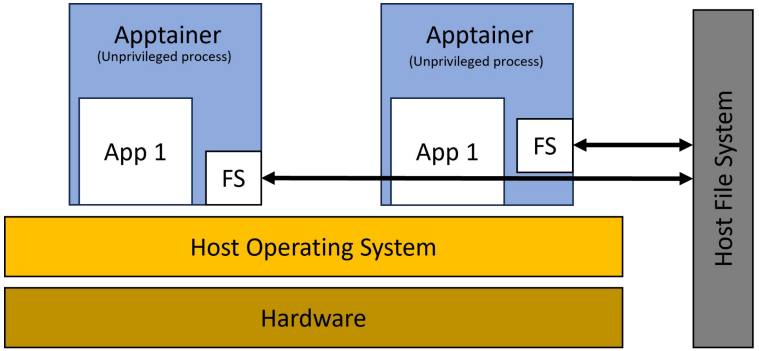
Docker

- Daemon with root privileges for certain features
- Can be less secure due to elevated privileges on large systems with shared resources
- General containerization

Apptainer

- Daemonless and rootless
- Secure with rootless
- Cryptographically signed images
- HPC, Scientific Computing





Architecture of Odo and Frontier

Frontier

- 77 Olympus rack HPE cabinets
- Each rack has 128 AMD compute nodes
- Each node has 64-Core Optimized 3rd Gen EPYC CPUs
- Each node has 4 AMD MI250X GPU, each with 2 GCDs
- Total of 9856 AMD compute nodes(nodes)
- Lustre File System

Odo

- Training focused cluster with same architecture as Frontier
- Only 32 nodes
- GPFS file system

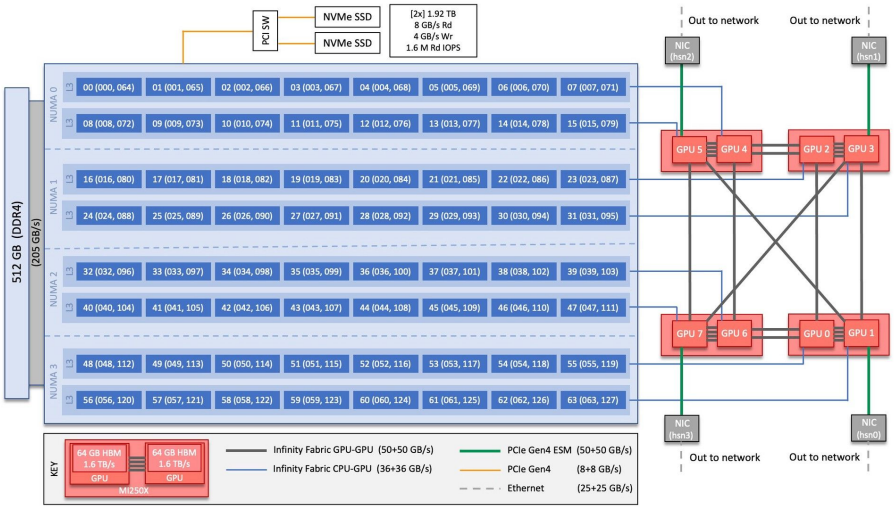


Image obtained from: https://docs.olcf.ornl.gov/systems/frontier_user_guide.html

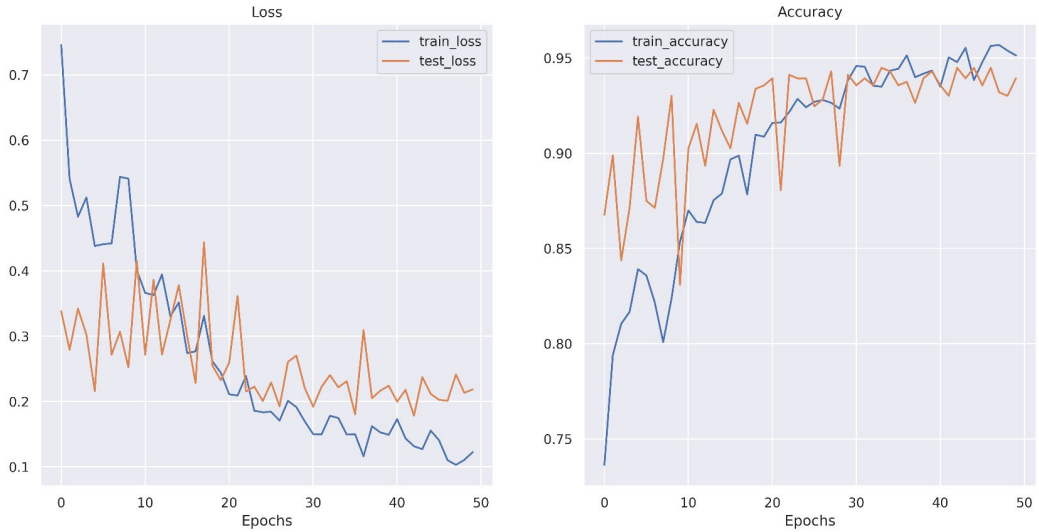
Our Model

- Convolutional Neural Network for binary classification
- Implemented in PyTorch and uses Convolutional layers, activation functions, batch normalization, and pooling layers to analyze the images

```
class ImageClassifier(nn.Module):
        def __init__(self):
        super().__init__()
        self.conv_layer_1 = nn.Sequential(
        nn.Conv2d(3, 64, 3, padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(64),
        nn.MaxPool2d(2))
        self.conv_layer_2 = nn.Sequential(
        nn.Conv2d(64, 512, 3, padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(512),
        nn.MaxPool2d(2))
        self.conv_layer_3 = nn.Sequential(
        nn.Conv2d(512, 512, kernel_size=3,
        padding=1),
        nn.ReLU(),
        nn.BatchNorm2d(512),
        nn.MaxPool2d(2))
        self.classifier = nn.Sequential(
        nn.Flatten(),
        nn.Linear(in_features=512*3*,
        out_features=2))
        def forward(self, x: torch.Tensor):
        x = self.conv_layer_1(x)
        x = self.conv_layer_2(x)
        x = self.conv_layer_3(x)
        x = self.conv_layer_3(x)
        x = self.conv_layer_3(x)
        x = self.conv_layer_3(x)
        x = self.classifier(x)
        return x
```

Training our Model

- Dataset of 2418 images obtained from Cyclegan database of UC Berkeley.
- Initially built model used single node with multiple CPUs and a single GPU.



Dataset obtained from: https://efrosgans.eecs.berkeley.edu/cyclegan/datasets/apple2orange.zip

Containerizing of Our ML Model

- Definition File for building Miniforge, MPI4py and Pytorch

```
Bootstrap: localimage
From: opensusempich342rocm571.sif

%post
set -e
zypper install -y bzip2 libtool

# installing miniforge and creating a conda environment for pytorch
wget "https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-$(uname)-$(uname -m).sh"
bash Miniforge3-$(uname)-$(uname -m).sh -b -p /opt/miniforge
rm Miniforge3-$(uname)-$(uname -m).sh
source /opt/miniforge/bin/activate
pip install torch==2.2.2 torchvision==0.17.2 torchaudio==2.2.2 --index-url https://download.pytorch.org/whl/rocm5.7

# install mpi4py
MPICC="mpicc -shared" pip install --no-cache-dir matplotlib seaborn tqdm --no-binary=mpi4py mpi4py
```

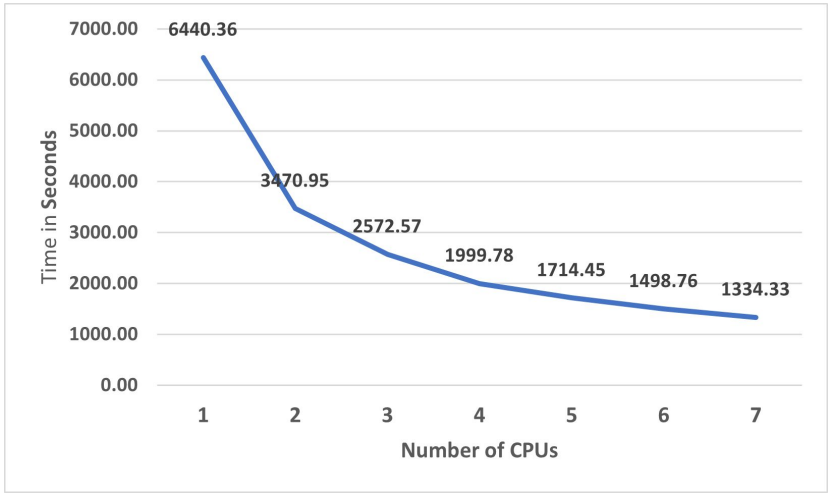
$ apptainer build opensusempich342rocm571torch222.sif opensusempich342rocm571torch222.def

Running Containerized Model

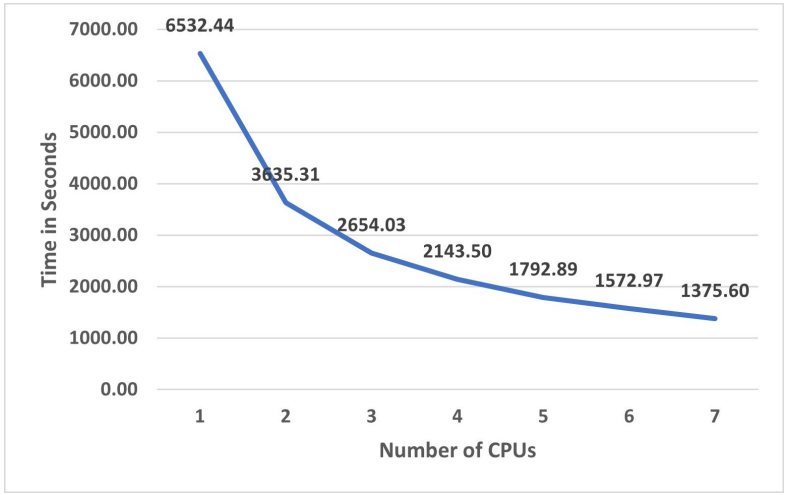
```
#!/bin/bash
#SBATCH -J ML_CONTAINER
#SBATCH -t 35:00
#SBATCH -N 1
                    .
                    .
                    .
# Needed to bypass MIOpen, Disk I/O Errors
export MIOPEN_USER_DB_PATH="/tmp/my-miopen-cache"
export MIOPEN_CUSTOM_CACHE_DIR=${MIOPEN_USER_DB_PATH}
rm -rf ${MIOPEN_USER_DB_PATH}
mkdir -p ${MIOPEN_USER_DB_PATH}
                    .
                    .
                    .
export MPICH_GPU_SUPPORT_ENABLED=1
export BINDS=/usr/share/libdrm,/var/spool/slurm,/opt/cray,${PWD}
export APPTAINERENV_LD_LIBRARY_PATH="/opt/cray/pe/mpich/8.1.28/ofi/crayclang/14.0/lib-abi-mpich:/
opt/cray/pe/mpich/8.1.28/gtl/lib:/opt/rocm/lib:/opt/rocm/lib64:$CRAY_LD_LIBRARY_PATH:$LD_LIBRARY_PATH:/opt/c
ray/pe/lib64"

srun -N1 --tasks-per-node=8   -c7 --gpus-per-task=1 --gpu-bind=closest  apptainer exec --bind $BINDS --workdir `pwd`
--rocm opensusempich342rocm571torch222.sif ./pyrun.sh
```

Performance of our ML Model (Single Node, Single GPU) on Odo

- Training time on Bare Metal is 2718.74 seconds on average
- Training time in Container is 2815.25 seconds on average
- Difference in training time is less than 60 seconds



Bare Metal Model



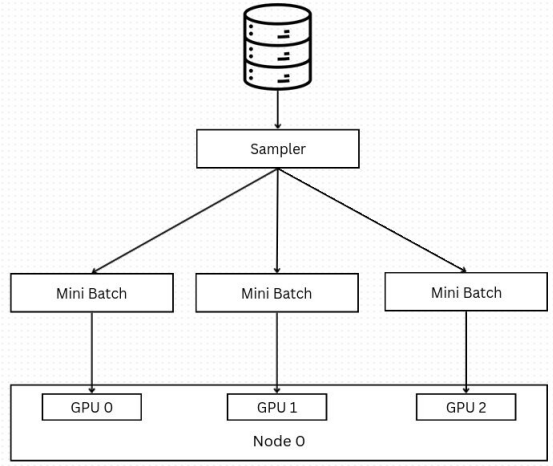
Containerized Model

Parallelism in Pytorch

- Utilized PyTorch DP and DDP in scaling to multiple GPUs and nodes respectively

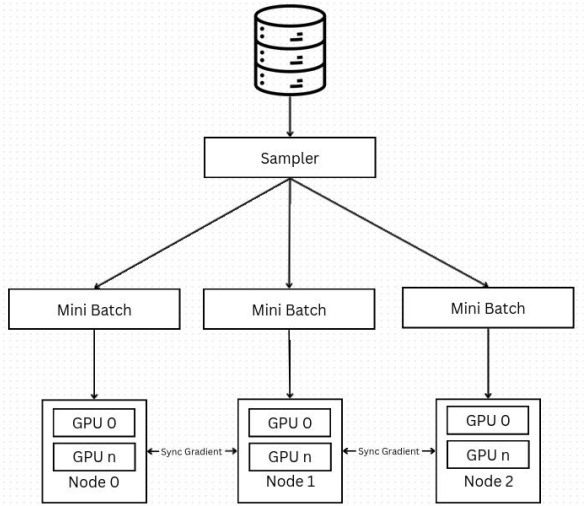
Data Parallel

- Single Process, multi-thread
- Input is split across GPUs by chunking in the batch dimensions
- During forward pass, module is replicated across devices
- During backward pass, gradients from all replicas are combined



Distributed Data Parallel

- Multiple process and creates a single instance of DDP per process.
- Each process will have their own copy of the model
- Input is not chunked but utilizes gradient synchronization and buffer to provide parallelism
- Generally, faster than Data Parallel



Code Comparison for Data Parallel and Distributed Data Parallel

Data Parallel

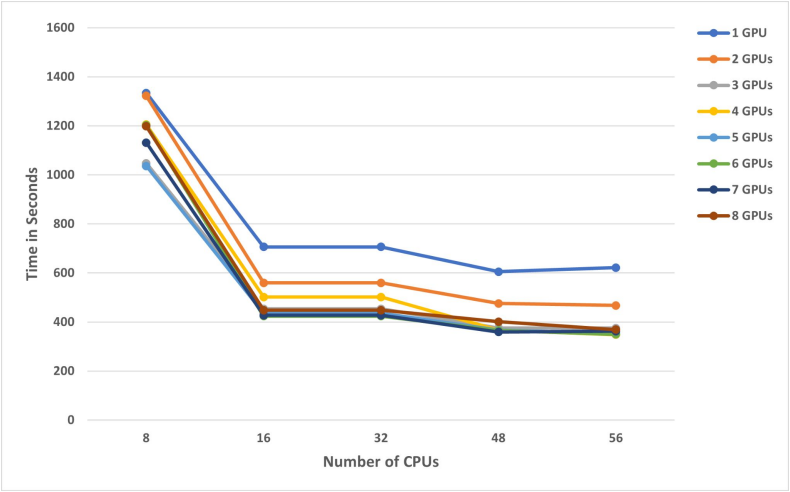
```
from torch import nn
from ImageClassifier import ImageClassifier
        :
        :
        :
device = "cuda" if torch.cuda.is_available() else "cpu"
        :
        :
        :
# Instantiate an object.
model = ImageClassifier()
model = nn.DataParallel(model)
model.to(device)
```

Distributed Data Parallel

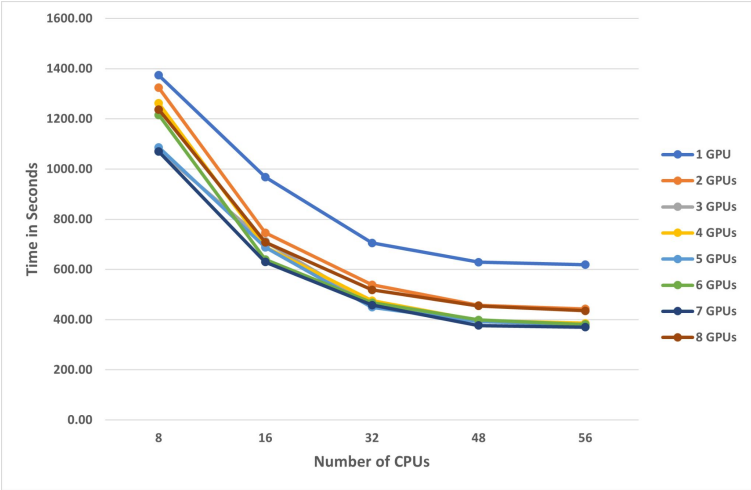
```
from torch.nn.parallel import DistributedDataParallel as DDP

class Trainer:
        def __init__(
        self,
        model: torch.nn.Module,
        save_every: int,
        snapshot_path: str,
        local_rank: int,
        world_rank: int,

        ) -> None:
        self.local_rank = local_rank
        self.global_rank = global_rank

        self.model = model.to(self.local_rank)

        self.model = DDP(self.model, device_ids=[self.local_rank])
```

Performance of our ML model (Single Node with varying CPUs and GPUs)

- Used **Data Parallel** with 4x speedup compared to initial single GPU run.
- Training time on Bare Metal averaged 598.89 seconds
- Training time on Container averaged 659.59 seconds
- Difference in training time is less than 60 seconds



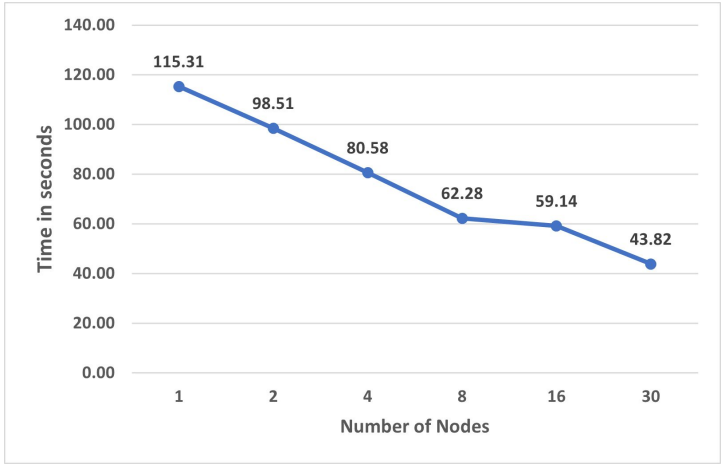
Bare Metal Model



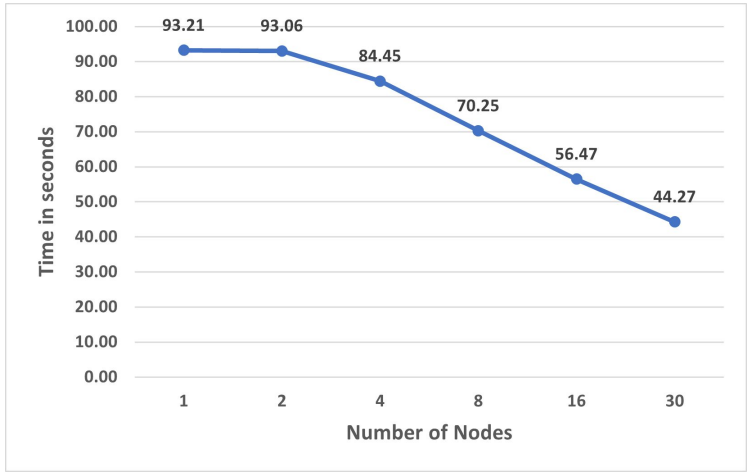
Containerized Model

Performance of our ML model (Scalability)

- Used **Distributed Data Parallel** to scale to 30 nodes, 1680 CPU cores and 240 GPUs.
- 9x faster compared to Data Parallel
- Training time on Bare Metal was averaged 76.61 seconds
- Training time from Container was averaged 73.62 seconds
- Difference in training time less than 3 seconds on average



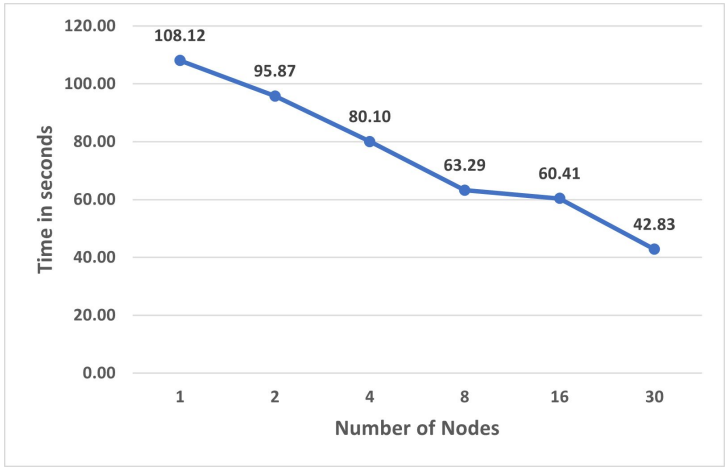
Bare Metal Model



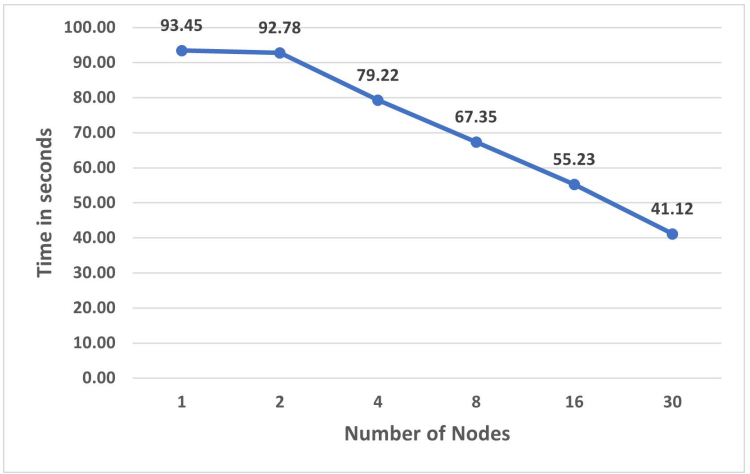
Containerized Model

Porting our Containerized ML Model to Frontier

- More of plug-and-play as Odo is just a mini Frontier
- Updated file-paths and versions of modules like cray-mpich-abi and rocm.
- Training time reduced by 1.97% on Frontier compared to Odo on Bare Metal
- Training time reduced by 2.88% on Frontier compared to Odo on Container
- Similar performance between native and container implementation on Frontier.



Bare Metal Model



Containerized Model

Containerizing LLMs On Frontier

- Definition File for building Ollama for LLM

```
Bootstrap: docker
From: opensuse/leap:15.4
%post
set -e
export DEBIAN_FRONTEND=noninteractive

zypper install -y wget sudo gzip gcc-c++  gcc-fortran tar make autoconf automake binutils cpp glibc-devel m4 makeinfo zlib-devel gcc-info git glibc-info patch pkg-config bzip2 libtool
# installing miniforge and creating a conda environment for pytorch
wget "https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-$(uname)-$(uname -m).sh"
bash Miniforge3-$(uname)-$(uname -m).sh -b -p /opt/miniforge
rm Miniforge3-$(uname)-$(uname -m).sh
source /opt/miniforge/bin/activate


# downloading ollama pre-released model with rocm
wget https://github.com/ollama/ollama/releases/download/v0.6.3/ollama-linux-amd64.tgz
wget https://github.com/ollama/ollama/releases/download/v0.6.3/ollama-linux-amd64-rocm.tgz


# extracting ollama
tar -xzvf ollama-linux-amd64.tgz
tar -xzvf ollama-linux-amd64-rocm.tgz
# installing ollama via pip to use it in python
pip install ollama
```

Applying Containers to LLMs on Frontier

- Decided on AstroLlama(6.74 Billion Parameters) with 7.2GB size and CodeLlama(70 Billion Parameters) with 39GB size.

```
Bootstrap: localimage
From:ollama.sif

%post
ollama serve&

%runscript
echo "building astrollama from ollama"
ollama pull hf.co/UniverseTBD/astrollama.gguf
ollama run hf.co/UniverseTBD/astrollama.gguf
```

AstroLlama

```
Bootstrap: localimage
From:ollama.sif

%post
ollama serve&

%runscript
echo "building codellama from ollama"
ollama pull codellama:70b
ollama run codellama:70b
```

CodeLlama

Performance of Containerized Large Language Model

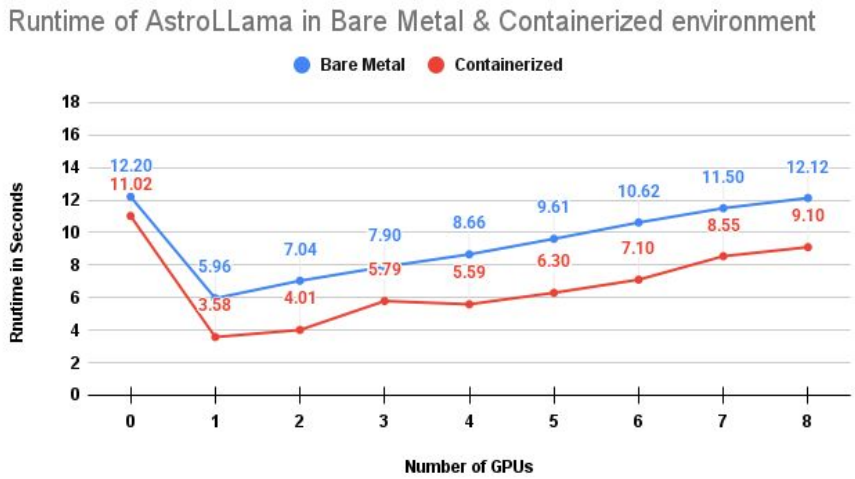
- Utilized *OLLAMA_SCHED_SPREAD=1* flag to enable multi-GPU usage for the model.
- Used *ROCR_VISIBLE_DEVICES* to vary the number of GPUs available to the model.
- Containerized environment had lower runtime consistently compared to Bare Metal for smaller size LLM, AstroLLama

AstroLLama Runtime		
Number of GPUs	Bare Metal	Containerized
0	12.2	11.02
1	5.96	3.58
2	7.04	4.01
3	7.90	5.79
4	8.66	5.59
5	9.61	6.30
6	10.62	7.10
7	11.50	8.55
8	12.12	9.10

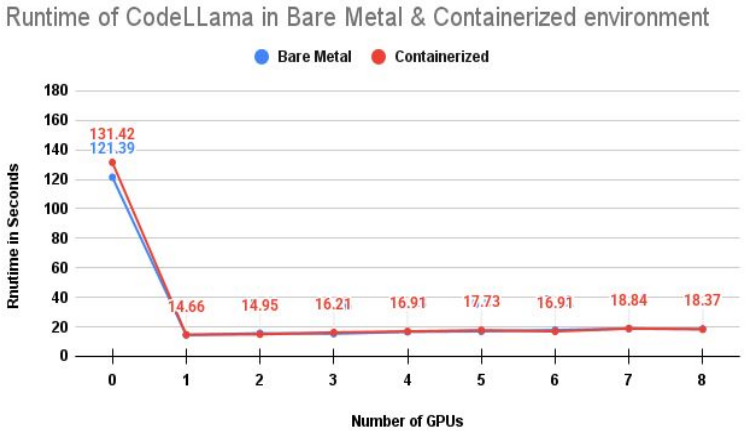
CodeLLama Runtime		
Number of GPUs	Bare Metal	Containerized
0	121.39	131.42
1	14.51	14.66
2	15.63	14.95
3	15.36	16.21
4	16.63	16.91
5	16.97	17.73
6	17.82	16.91
7	18.86	18.84
8	18.51	18.37

Performance of containerized LLM

- Performance of LLM within container comparable to native in some cases
- On average, containerized LLM performed up to 30% better than Bare Metal implementation.
- Performance comparable for both smaller and larger size model in containerized and Bare Metal environment



AstroLLama



CodeLLama

Conclusion

- From our ML model, it is possible to containerize ML applications and run on OLCF Frontier and Odo supercomputers.
- Performance of containerized PyTorch model using DDP found to be 9X faster compared to containerized model using DP on Frontier.
- Successfully containerized LLMs like CodeLLama and AstroLLama on Frontier.
- No performance penalty observed for containerized LLMs on Frontier.

Future Work

- Explore how *hybrid mode* implementations impact the performance of containerized model and containerized ML applications in general.
- Scaling LLM implementation to multiple nodes, and evaluating the performance impact.
- Experimenting with reduced container image size by using multi-stage image builds.

Questions?

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.