Evaluating the Performance of Containerized ML and LLM Applications on the Odo and Frontier Supercomputers

Bishwo Dahal
dahalbi@warhakws.ulm.edu
University of Louisiana Monroe &
Oak Ridge National Laboratory
USA

Elijah MacCarthy
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
maccarthyea@ornl.gov

Subil Abraham
Oak Ridge National Laboratory
Oak Ridge, Tennessee, USA
abrahams@ornl.gov

Abstract

Containers are transforming scientific computing by simplifying the packaging and distribution of applications. This enables researchers to create and deploy their applications in isolated environments with all necessary dependencies, enhancing portability and deployment flexibility. These advantages make containers especially suitable for High Performance Computing (HPC) facilities like the Oak Ridge Leadership Computing Facility (OLCF), where complex scientific applications are being developed and deployed. In this work, we investigate the performance of containerized machine learning (ML) applications in comparison to bare-metal execution on the Frontier Exascale supercomputer. Specifically, we aim to determine whether ML models, when trained and tested within containers on Frontier using Apptainer, exhibit performance similar to that of bare-metal implementations. To achieve this, we use containers to package and run Convolutional Neural Network (CNN)-based ML applications on the OLCF Frontier and Odo supercomputers and assess their performance against bare-metal runs. After conducting scalability tests across up to 30 nodes with 1680 AMD EPYC CPU cores and 240 GPUs, we find that the performance of the containerized ML applications is at par with that of bare-metal runs. We apply the lessons learned from our containerized ML model to containerizing and evaluating performance of LLMs like AstroLLaMA, and CodeLLaMA on Frontier.

CCS Concepts

• **Computer systems organization → Parallel architectures**; *Distributed architectures*; • **Computing methodologies →** *Machine learning*; *Distributed algorithms*; • **Theory of computation → Distributed algorithms**.

Keywords

HPC, Containers, Machine Learning, Large Language Models, Pytorch, Ollama, AstroLLaMA, CodeLLaMA

ACM Reference Format:
Bishwo Dahal, Elijah MacCarthy, and Subil Abraham. 2025. Evaluating the Performance of Containerized ML and LLM Applications on the Odo and Frontier Supercomputers. In *Proceedings of Cray User Group Conference (CUG '25)*. ACM, New York, NY, USA, 9 pages. https://doi.org/XXXXXXX.XXXXXXX

CUG '25, Jersey City, NY
2025. ACM ISBN 978-1-4503-XXXX-X/2025/05
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The Internet Data Center (IDC) projected that the total volume of collected data will reach an estimated 175 zettabytes, equivalent to 1.92×10^{14} gigabytes, by 2025.[2]. With this growing global data market, technological advancements in computer hardware, and development of efficient algorithms and data analysis techniques have attracted huge attention in the global economy. It requires a lot of advanced computational resources and efficient algorithms to effectively manage such a massive amount of data. High-Performance Computing (HPC) provides an avenue to tackle some of these issues as it addresses data and computationally intensive operations at very high speed by utilizing its multiple nodes to tackle problems in parallel.

Running these data and computationally intensive applications at large scale requires efficient optimization and data management. This can be achieved effectively through container technology. Containers are lightweight software packages that bundle applications along with their dependencies into a single unit. These unique features provide a huge advantage when managing complex workloads in HPC facilities. Recently, containers have gained great popularity because of their lightweight and portable nature. In a portworx and Aqua security survey conducted, 87% respondents among 500 IT professionals mentioned that they were running containers in their companies[16]. This growing usage of containers has impacted the field of machine learning (ML) by providing isolated, scalable, and reproducible environments for the evaluation of ML models.

Machine learning models can be trained using various techniques including Linear Regression, Logistic Regression, Binary Classification, Random Forests and Decision Trees, depending on the type of dataset and requirements of the model. These algorithms can be used in image recognition, traffic prediction, autonomous vehicles, fraud detection, and academia. Due to the significance of these applications, machine learning research has been growing significantly in recent years. HPC systems support distributed workloads, making them optimal for research in these topics.

Training large models requires hardware such as central processing units (CPUs), graphics processing units (GPUs), and field programmable gate arrays (FPGAs)[23]. While these hardware are fundamental for machine learning algorithms, their architecture plays a crucial role in enhancing efficiency. In particular, evolution of multi-core and many-core architectures in CPUs has been groundbreaking in parallel computing. Although CPUs are considered as core components of computing, a platform built with a mix of CPUs, GPUs, and FPGAs or many core CPUs are preferred for performing these complex operations due to their efficiency and support for parallel computing required for handling compute intensive operations.

Key among most powerful computers in the world are those deployed by the Department of Energy's (DOE) Office of Science User Facilities (UF). Among the User Facilities are the Leadership Computing Facilities (LCF), Argonne Leadership Computing Facility (ALCF), and Oak Ridge Leadership Computing Facility (OLCF). OLCF deployed Frontier, the first supercomputer to break the exascale barrier on the Top500 list with a High-Performance Linpack (HPL) of 1.102 quintillion floating-point operations per second (FLOPs) in 2022.

In this work, we use Frontier to investigate the performance of containerized machine learning (ML) applications compared to bare-metal implementations. We draw from lessons learned in the containerization of ML models to containerize and test LLMs on Frontier. We assess the performance of the models in each environment and provide recommendations to users in containerizing such workflows on Frontier.

Our text is organized as follows. In Section 2, we describe container technology in brief. We then present and discuss recent containerization technologies including Podman, Docker and Apptainer. We compare the pros and cons of some of these technologies and present the reasoning behind using Apptainer as our containerization technology. In Section 3, we discuss work related to containerizing scientific applications and narrow down to container usage in ML. Section 4 provides details on the architecture of the systems used in this work with a deeper dive into the ML model and data-set used. We present our results in Sections 5 and 6. Specifically in Section 5, we discuss performance from our containerized ML model compared to native implementations. In Section 6 we apply containers to LLMs on Frontier and evaluate the performance. We provide an overview of next steps in Section 7 and summarize our findings and recommendations for users of containers within HPC in Section 8.

2 Background

In this section, we provide an overview of containers, including different container build and orchestration tools. We further highlight the portability and scalability of these container technologies and their usage within different fields of science such as ML/AI.

2.1 Container Technology

With the growing need of cloud applications, it has become important to make applications portable. Moreover, it is also crucial to run the same application on a single computer encapsulating the data to that specific application.

Virtualization and containerization are two commonly used techniques for hosting multiple applications on a single computer. Virtualization uses a hypervisor, which sits on top of the operating system, to divide the system into independent virtual machines. Containers, on the other hand, use the host system's kernel to isolate multiple instances of containers. By sharing the host kernel and operating system, containers avoid the overhead of virtualization in terms of memory, speed, and disk resources, thereby providing higher performance with fewer resources. It is also worth mentioning that containers can bind to host-specific libraries and files at runtime (a process known as *bind-mode*), potentially offering performance comparable to native implementations. If the container strictly uses libraries and files installed within it at runtime, it is said to be running in *hybrid-mode*.

We discuss a couple of container build and runtime tools in the subsections below:

2.1.1 Podman. Podman is used to develop, manage and run container ecosystems using the libpod library[3]. Podman is daemonless, which makes it secure since it does not require root privileges to run containers. It also supports Open Container Initiative (OCI) and Docker images, thus, making it possible to run docker images without requiring docker in a rootless environment.

2.1.2 Docker. Docker is a lightweight, easily deployable containerization technology that isolates applications and its dependencies inside a virtual container. Initially, Docker heavily relied on Linux Containers (os-level virtualization technology) but moved to its own created module called libcontainer [6]. These virtual containers can be built, shipped and run inside various environments by users, offering huge advantages in terms of portability, scalability, speed, delivery and maintenance[20]. Docker made containerization technology famous by focusing on scalability and flexibility.

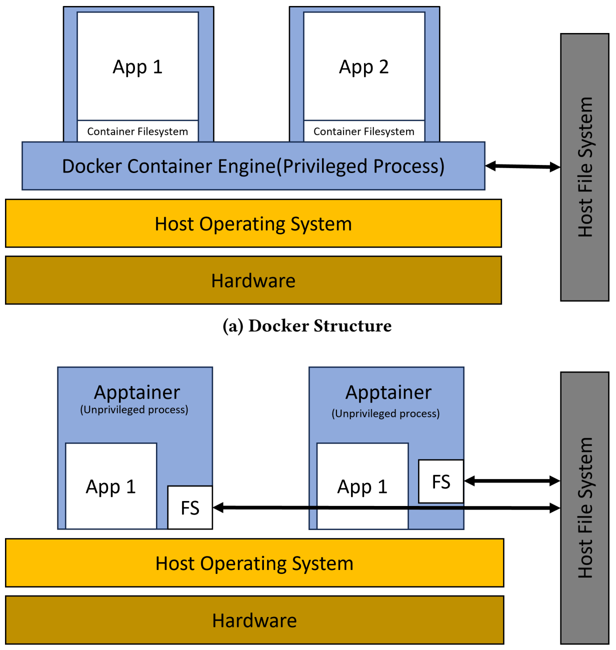
2.1.3 Apptainer. Singularity was developed by Greg Kurtzer while working as HPC systems architect at Lawrence Berkeley National Laboratory (LBNL). It was later renamed as Apptainer and added as a Linux Foundation project in 2022. Apptainer focuses on performance, reproducibility, and security. It is more suitable for HPC environments and makes integration easier for multi-node applications by providing support for services like MPI.

2.2 Comparing Apptainer and Docker in HPC

Docker runtime in some cases requires elevated privileges to run containers. It provides applications a means to gain root access of the system that they are running on. Apptainer filled this security gap in container technology by allowing users to create and run containers without root access. Figure 1 compares the architecture of Docker and Apptainer. In Apptainer, a privilege separation model called non-setuid mode is used to prevent users from escalating privileges once they are inside a container. Admins can limit Apptainer's user on the system to binding only specific directories. Apptainer can also generate cryptographically signed images for additional security. The difference in performance for the most common operations is virtually same for both setuid-root and non-setuid mode[4]. These features of Apptainer make it secure to build and run containers on multi-user systems like HPC environments.

2.3 Machine Learning

Containers have been used in various fields of science including Bioinformatics, Meteorology, Astronomy and Machine Learning [15] [25] [7]. While its popularity varies across different fields, Machine Learning is one field where it has gained significant presence in the last couple of years.



(a) Docker Structure

(b) Apptainer Structure

Figure 1: Comparison of internal structure of Apptainer and Docker

Machine learning provides computers the ability to mimic human behavior, gradually improving its accuracy. Some early footprints of machine learning were seen by different researchers, in statistical methods by *Nilsson (1965)* and in neural networks by *Rosenblatt (1962)*[8]. Over the years, many advanced methods have been developed, including statistical and pattern recognition techniques(k-nearest neighbors, discriminant analysis, Bayesian classifiers), inductive learning of symbolic rules(decision i, decision rules, logic programs), and artificial neural networks(multilayered network with backpropagation, Kohonen's self organizing network)[8].

Sahu et al. [19] emphasized how machine learning (ML), deep learning (DL), reinforcement learning (RL), and deep reinforcement learning (DRL) are used to develop accurate predictive models. With the help of containers, these models can be packaged and deployed into several environments, making their usage for predictive modeling less stressful and efficient.

As a result, we apply the concept of containers in this work in packaging and testing the performance of a Convolutional neural network(CNN) based model for image recognition. The Convolutional neural network (CNN) model is an artificial intelligence (AI) system based on multi-layer neural networks that can identify, recognize, and classify objects as well as detect and segment objects in images[21]. CNN is best known for hiding complex layers and requiring less human supervision, making it one of the most popular algorithms[21]. Generally, CNN is developed by combining Convolutional layers, Pooling layers, activation function, and fully connected layers[21]. They are very popular specifically in binary classification algorithms, hence, the motivation for their usage in this study.

2.4 Parallelism in Pytorch

Training image recognition models, such as Convolutional Neural Networks(CNNs), on a single CPU is feasible but often time consuming. A single CPU system can process the data and perform necessary computations, but the limited resources (CPUs and memory capacity) makes the training process significantly slower. Unlike these kinds of systems, supercomputers like Frontier comprise several nodes with many-core CPUs and efficient GPUs which can be used in implementing and optimizing models like these.

Within Pytorch are different parallel training paradigms that can be utilized for faster model training and in essence benefit from the computational prowess of a system like Frontier. We itemize and provide details about these parallel training paradigms in PyTorch below.

2.4.1 Data-Parallel (DP). With Data-Parallel, input is split across the specified devices by chunking in the batch dimension[17]. During the forward pass, the module is replicated across devices, with each replica handling a segment of the input, while in the backward pass, gradients from all replica are summed back into the original module[17].

2.4.2 Distributed Data-Parallel (DDP). Distributed Data-Parallel is a feature that provides data parallelism by synchronizing gradients across each model replica[18]. As a result, it does not chunk input like in DP. Data-Parallel (DP) is usually slower than Distributed Data-Parallel (DDP) even on a single machine due to GIL contention across threads, per-iteration replicated model, and additional overhead introduced by scattering inputs and gathering outputs[10].

3 Related Work

Though computer scientists and developers appreciate the isolation provided by containers and containerized applications, there is still skepticism in running compute intensive algorithms like machine learning models within containers. However, a great deal of work has been done to analyze the performance of various applications within container frameworks.

In recent work, researchers investigated features of apptainer and its effect on the runtime of computations within a container [15] [13]. *Younge et al.* [26] also examined the application of Singularity on Cray systems and benchmarked the performance overhead on HPC systems. *Abraham et al.* [1] on the other-hand, compared the use of Singularity, Docker, Podman, and Charlie-cloud on HPC systems, focusing on performance with the Lustre file-system.

With the increasing adoption of containerization tools in High-Performance Computing (HPC), it is crucial to evaluate their performance on various applications, including different fields of machine learning. In particular, image recognition which has been widely studied using different machine learning approaches such as convolutional neural networks (CNN) and deep belief networks (DBN). *Wu et al.* [24] proposed that classification accuracy rate of CNNs are comparably higher than that of DBNs. *Taye* [21], also, further explained in-depth the underlying layers of CNN along with its importance in the classification problem in a recent work.

Numerous studies have been conducted by researchers to highlight the role and importance of CNNs within machine learning. *Q. Liu et al.* [12] argued that CNNs represent the state-of-the-art model for image recognition, while *Li et al.* [11] analyzed their building blocks and various applications. In this paper, we evaluate the performance of this *state-of-the-art* model for image recognition. We will assess its performance from within a container compared to native implementation.

4 Experimental Setup

This study used two Oak Ridge Leadership Computing Facility (OLCF) resources, Frontier and Odo. In the following subsections, we provide details on the architecture of these OLCF resources. We applied the concept of containers to Artificial Intelligence approaches by using CNN based machine learning algorithms in building models on Frontier and Odo. These kinds of model are resource intensive, requiring intensive vector manipulation and I/O operations to process large amounts of data efficiently. We further applied lessons learned from our ML model containerization to containerizing and testing LLMs on Frontier.

We setup our machine learning experiments in Section 5 such that the same PyTorch model, running natively on Frontier and Odo is containerized and executed from within a container. This is to ensure a leveled ground for testing. Our LLM experiments are also setup in Section 6 and follow a similar setup as 5. Apptainer v1.3.2-1 is used as the container build and runtime tool in building container images on Frontier and Odo. The container images from Apptainer are in Singularity Image Format which are OCI artifacts. Definition files for building these images follow the Apptainer definition file format and consists of compiler installs, python interpreter and PyTorch needed to run the application. We installed miniforge within the container to activate the conda environment used for PyTorch. Specifically, we installed ROCM/5.7.1, Python/3.9.12, PyTorch/2.2.2, and torchvision/0.17.2 within the container.

4.1 Architecture of Frontier and Odo

Frontier consists of 74 Olympus rack HPE cabinets, each with 128 AMD compute nodes, and a total of 9,408 AMD compute nodes[9]. The AMD compute nodes have 64-core Optimized 3rd Gen EPYC CPUs (with 2 hardware threads per physical core). Each node also contains [4x] AMD MI250X, each with 2 Graphics Compute Dies (GCDs) for a total of 8 GCDs per node. The 8 GCDs can be thought of as 8 separate GPUs each having 64 GB of high-bandwidth memory (HBM2E). The Frontier interconnect is HPE Cray Slingshot and it also mounts a Lustre file system for storage. Unlike Frontier, Odo mounts a GPFS file system but has the same architecture as frontier. Since it is for open enclave training events, it has 32 nodes dedicated for training purposes.

Figure 2 shows the structure of Frontier Compute node which has 64 core CPU and 8 GPUs.

4.2 Datasets

The dataset used in this experiment was obtained from UC Berkeley's official directory of CycleGAN Datasets [5]. It comprised 2418 images which we split into training and testing sets following a

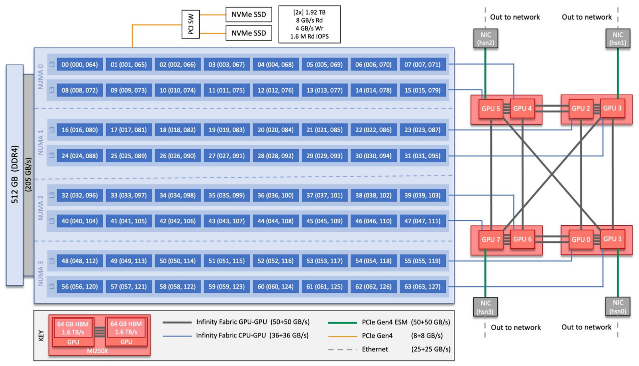
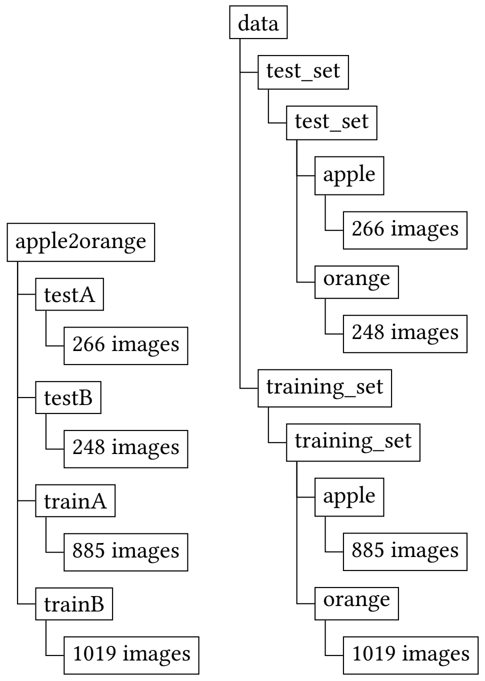


Figure 2: Frontier Compute Node Diagram from OLCF docs. (https://docs.olcf.ornl.gov/systems/frontier_user_guide.html)

[9]

20:80 ratio respectively. We modified the layout of the datasets such that it follows the pattern shown below in Fig.4.2.



4.3 Our Model

We use convolutional neural network designed for binary classification of images. It is implemented using PyTorch and uses several convolutional layers, activation functions, batch normalization, and pooling layers to analyze the image. In our model,

```
nn.Conv2d(3, 64, 3, padding=1)
```

acts as convolutional layer, 3 input channels and 64 output channels with size of 3. `nn.ReLU()` is a Rectified Linear Unit which is an activation function. `nn.BatchNorm2d(64)` is a batch normalization layer which takes 64 input features. `nn.MaxPool2d(2)` is a max pooling layer with size of 2. `nn.Linear(512*3*3,2)` is a fully connected layer that takes flattened input from the convolutional layers and outputs 2 features for binary classification. When we tested our model, we found the accuracy of the model to be near 94% as shown in Figure 3.

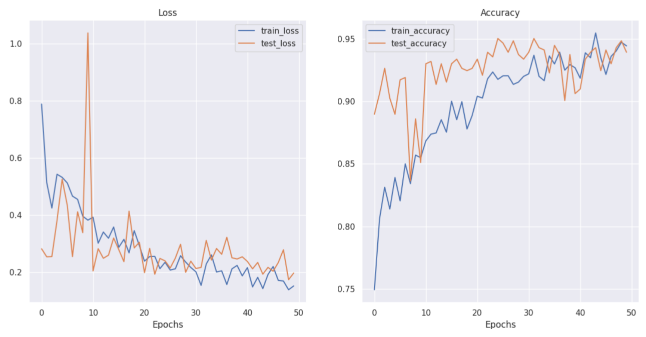


Figure 3: Accuracy and Loss of our model

Listing 1: CNN Model

```
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
```

We began our experiments by training the CNN model using multiple CPU cores and a single GPU on a node. We also implemented the PyTorch Data-Parallel feature, which is a single process with support for multithreads and capable of utilizing multiple GPUs within a single node[10]. We further assessed the model performance by using Distributed Data-Parallel (DDP), which is multiprocess and works for both single and multi-node training[10]. DDP works with model parallel which splits data to multiple GPUs, while Data-Parallel does not. When DDP is combined with model parallel, each DDP process would use model parallel, and all processes collectively would use data parallel with a possibility of better performance.

We test our application using single node and multiple GPUs and CPUs in both containerized and bare-metal environments. We also utilized rocm-smi (library that provides interface to monitor and control AMD GPU applications) to confirm the number of GPUs being used. In addition, we tested our application using multiple nodes and multiple GPUs using DDP.

5 Performance of Containerized ML Model

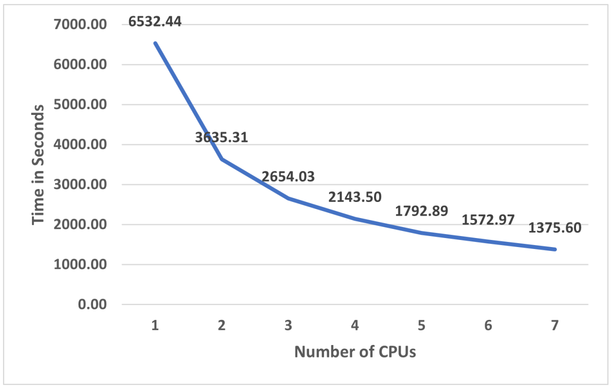
In this section, we discuss the varying tests performed in order to evaluate the performance of our ML model. Sections 5.1 through 5.2.1 discuss results from the Odo implementation, while 5.3 discusses results from Frontier for our ML model. The evaluations are considered on two fronts:

5.1 Single Node

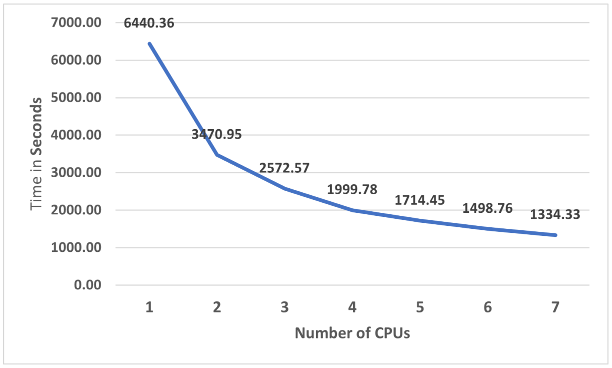
We initially developed a model capable of training on a single node utilizing multiple CPUs and a single GPU. This model did not use the Data-Parallel feature within PyTorch. Figure 4 shows line graphs of training time of this model. We evaluate the performance of the model on a single node utilizing multiple CPU cores and a single GPU. We observed the average training time for the model was 2815.25 seconds within a container compared to 2718.74 seconds on bare-metal. This is slightly beyond a one (1) minute difference in training time between the containerized model compared to bare-metal, thus, we can say a comparable performance using the two approaches.

For experiments conducted using multiple GPUs on a single node, we implemented the PyTorch DP feature. This resulted in an average training time of 659.59 seconds for containerized model compared to 598.89 seconds for native runs as shown in Figure 5. This is a minute of training time difference between containerized and bare-metal runs and about 4*x* speedup compared to non DP runs discussed earlier. We observed overall that as we increased the number of CPUs for the containerized model, the training time decreased rapidly but started to get slightly stable after 32 CPUs within the container. For native runs on the other-hand, the training time dropped more sharply from 8 to 16 CPUs and then stabilized after 16 CPUs.

For increased number of GPUs, the model training time significantly reduced, going from 1 to 2 GPUs. After 2 GPUs, the pattern of training time followed a similar trend, depending on the number of CPUs used for the containerized model. On bare metal, however, we noticed a more substantial difference in performance between using 2 GPUs and anything more than 2 GPUs.

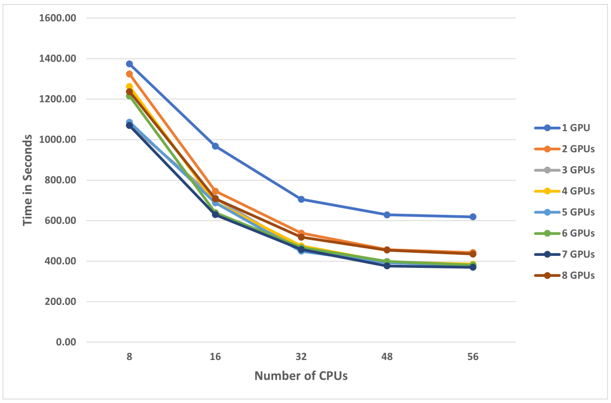


(a) Performance of Model in Container

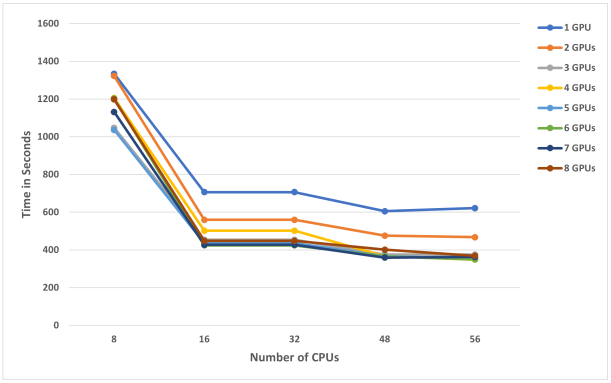


(b) Performance of Model on Bare metal

Figure 4: Performance of model without Data Parallel feature



(a) Performance of Model in Container



(b) Performance of Model on Bare metal

Figure 5: Performance of model with Data Parallel mode

Overall, the model consistently performed better when using multiple CPU cores and 6 or 7 GPUs in both the containerized and bare-metal environments as shown in Figure 4.

5.2 Scalability

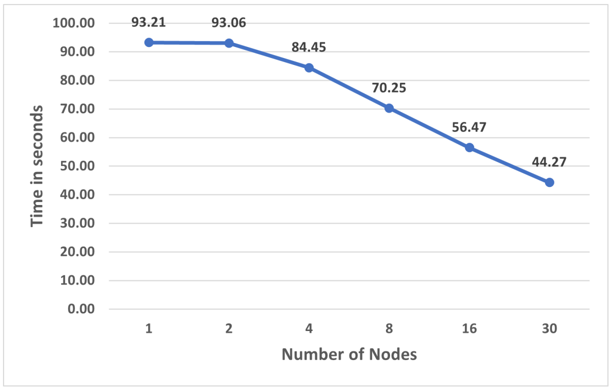
For this section, we examine how the CNN model training scales across multiple nodes. We start off with a single node with multiple GPU runs and observe the performance as we gradually scale across multiple nodes using MPI and multiple GPUs. For single node model training, we measured the execution time from beginning of the training iterations, evaluation at checkpoints, and exporting our final model as a file. For multi-node on the other hand, we measured the runtime from the start to the end of MPI initialization, training iterations, saving model at checkpoints, and exporting our final model as a file.

5.2.1 Multiple Nodes. When we moved to multiple nodes with multiple CPUs and GPUs using Distributed Data-Parallel, we observed a significant reduction in model training time across both native and container runs. We observed an average runtime of 76.61 seconds across bare-metal runs and 73.62 across containerized runs as shown in figure 6. This makes for less than 3 seconds difference in runtime between containerized and bare metal implementations with an almost 9*x* speedup (659.59 to 73.62 seconds) in model training compared to the DP approach.

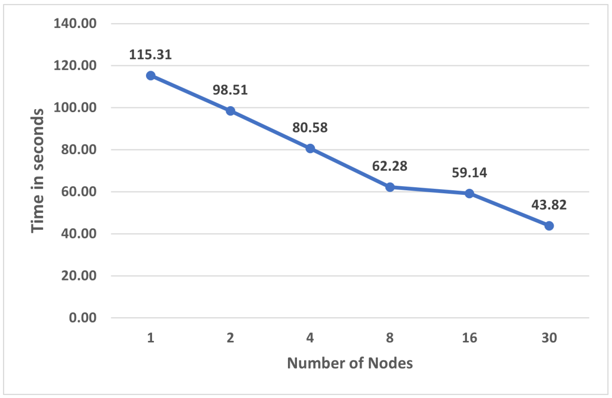
5.3 Portability

In this section, we assess the portability of our containerized model by moving it between Odo and Frontier. We port our container to Frontier, where we test our model on multiple nodes using DDP.

5.3.1 Frontier. Since Odo is the same architecture and just a training version of Frontier, porting from Odo to Frontier is more of `plug-and-play`. We needed to update the versions of some modules, including cray-mpich-abi and ROCM, to ensure compatibility with Frontier's environment. Additionally, we also adjusted the paths to match Frontier's file system. The performance of the model on Odo can be found in 6, while that for Frontier is shown in 7. The performance difference of the model on both Odo and Frontier in the containerized environment is negligible for up to two nodes. However, we see a higher performance difference as we increase the number of nodes. The average performance of our model on Frontier is marginally faster, with 1.97% less training time on Frontier than Odo when running the model in the native environment. In the containerized environment, on the other-hand, we see an average of 2.88% less training time on Frontier compared to Odo.

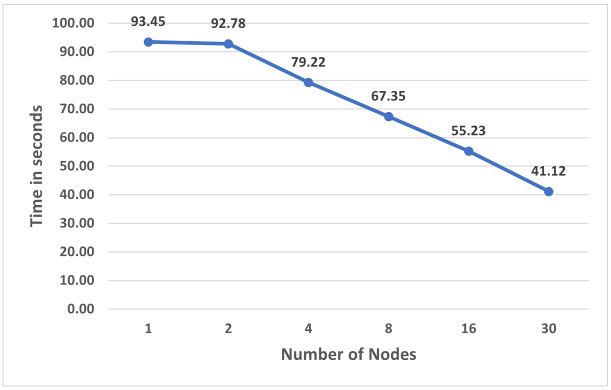


(a) Performance of Model in the Container on Odo

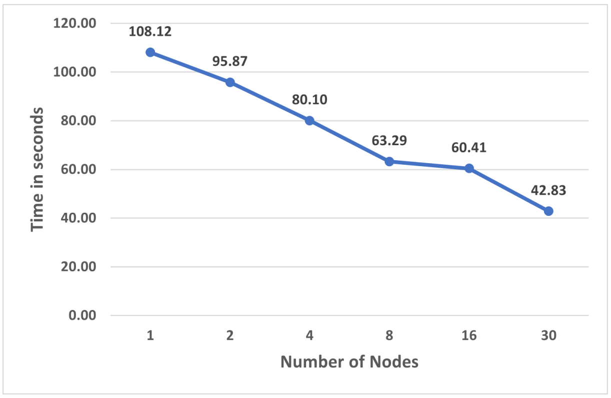


(b) Performance of Model on Bare metal on Odo

Figure 6: Performance of model with Distributed Data Parallel in Multi-node environment on Odo



(a) Performance of Model in the Container on Frontier



(b) Performance of Model on Bare metal on Frontier

Figure 7: Performance of model with Distributed Data Parallel in Multi-node environment on Frontier

6 Applications to Containerization of Large Language Models

Having been successful in containerizing machine learning models and verifying their conformity in performance to native implementations on Frontier and Odo, we apply our lessons learned to containerizing and evaluating performance of LLMs on Frontier. Two LLMs we consider for containerization and performance evaluations are CodeLLaMA and AstroLLaMA.

We provide details about these two models and their containerization process in the following subsections. It is worth noting that these LLMs were decided on as a way to demonstrate to the LLM user-base on Frontier and other User Facilities (UF) the possibility of running LLMs from within containers on these Exascale systems and the ability for the models to utilize multiple GPUs. The performance of the models as they are implemented and scaled across multiple GPUs is also assessed.

6.1 Containerizing LLMs

Our containerization process for LLMs is similar to the process described above for our ML model. We write Apptainer definition files that are used in building a SIF file of the container image. A notable difference between the definition file for the ML model and our LLMs is the building of Ollama and its AMD ROCM related libraries within the LLM definition file. Ollama is an open-source tool that allows creating, running and sharing LLMs.

CodeLLaMA is a Large Language Model built on top of LLaMA 2, and is capable of generating and understanding code across various programming languages, like, Python, C++ and Java. LLaMA 2 is an open-source LLM developed by Meta. For more on LLaMA 2 which is the base image for CodeLLaMA, refer to [22]. We used the 70 billion (B) parameter version of CodeLLaMA, which is approximately 39GB in size. Our CodeLLaMA LLM is built from ollama using the definition file in Listing 1.

Listing 2: Apptainer Definition file for Building CodeLLaMA

```
Bootstrap: localimage
From:ollama.sif

%post
ollama serve&

%runscript
```

```
ollama pull codellama:70b
ollama run codellama:70b
```

AstroLLaMA is our other LLM for containerization and performance evaluation. It is specifically related to astronomy research in that it can summarize astronomy research articles and provide domain-specific literature [14]. LLaMA 2 is the base image used in the development of the AstroLLaMA model. We used the 6.74B parameter version of AstroLLaMA, which is approximately 7.2GB in size. We show the definition file used for the AstroLLaMA model in Listing 2.

Listing 3: Apptainer Definition file for Building AstroLLaMA

```
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
```

6.2 Performance Evaluation of LLMs on Frontier

We run our LLMs from within containers on Frontier and track the time to response when requests are sent to the model. We assess the runtime from a CPU only model as well as a GPU implementation. We vary the number of GPUs available to the LLM using `ROCR_VISIBLE_DEVICES` and assess its impact on the performance of the model. We used `OLLAMA_SCHED_SPREAD=1` flag to enable multi-GPU usage and evaluated the performance from 0 GPUs(CPU only) to 8 GPUs.

6.2.1 Evaluation of LLMs. Table 1 represents the runtime data of two LLMs, AstroLLama and CodeLLama. We initially experimented with a CPU only model and then a GPU aware model, scaling from 1 to 8 GPUs. The same question is passed to the models using interactively allocated compute nodes on Frontier.

For CPU based (0 GPUs in table) AstroLLama model, we observe a runtime of 11.02 seconds for the containerized model compared to 12.20 seconds from bare metal. As GPUs are introduced to the test, both environments demonstrate performance gains with the containerized model recording a 3.58 seconds runtime compared to 5.96 seconds on bare metal. It is important to note however, that as the GPU count increases, the runtime rather increases. This can be best explained by the fact that the problem size is too small to benefit from the additional resources (increased GPU counts) beyond a certain threshold.

We see a similar trend with CodeLLaMA with the introduction of a single GPU recording about $9x$ and $8.4x$ speedup for the containerized and native implementations respectively. This is from a bare-metal runtime of 121.39 seconds for CPU only and 14.51 seconds for a single GPU addition. For the containerized model, 131.42 seconds against a 14.66 seconds using CPU only and a single GPU respectively.

7 Future Work

In this work, we conducted experiments using *bind mode (–bind)* to run our PyTorch model in containerized environments. This means that all host libraries and files (including mpi, specifically CRAY MPICH) needed at runtime by the application were mounted into the container, and therefore visible and accessible to the application. It would be worth examining how *hybrid mode* runs impact the performance of the containerized model and containerized ML applications in general. With *hybrid mode*, libraries and software installed within the container are used at runtime without mounting host-specific libraries into the container at runtime.

Also, we plan to utilize multiple nodes for our LLM experiments in the future as our current experiments only used single node with multiple GPUs on the node. This will help in evaluating whether multi-node runs result in any additional performance gains or result in expensive communication overheads.

Finally, we plan to further explore the potential benefits of a reduced container size by using multi-stage image builds, thereby reducing the final image size. This is to investigate whether the size of the container at startup has any bearing on overall performance of the application within the container.

8 Conclusion

We have successfully demonstrated that ML and LLM applications can be containerized and run on OLCF Frontier and Odo supercomputers with performance metrics comparable to native implementations.

Using a CNN-based image recognition model within PyTorch framework, we have shown that ML based containers can scale multiple nodes with performance equal to native implementations on OLCF Frontier and Odo supercomputers. Specifically, we have shown that when a containerized CNN model is scaled across up to 30 nodes with 240 AMD MI250X GPUs and 1680 AMD EPYC CPU cores using DDP, a performance improvement of about $9x$ is realized compared to DP implementations on a single node.

Additionally, based on lessons learned in containerizing and running ML models at scale, we have been able to containerize CodeLLaMA and AstroLLaMA large language models on Frontier. This demonstrates to the LLM user base on Frontier (and other Cray HPC systems) that LLMs can be containerized from the ground up and run successfully on Frontier without any performance penalty.

Acknowledgments

This research was supported in part by an appointment to the Oak Ridge National Laboratory Pathways to Computing Internship Program, sponsored by the U.S. Department of Energy and administered by the Oak Ridge Institute for Science and Education (ORISE).

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725.

Table 1: Runtime of AstroLLama and CodeLLama on varying GPUs in Seconds

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

Finally, we would like to thank Tuguldur Odbadrakh, John Holmen, Asa Rentschler and Veronica Melessa Vergara for their invaluable assistance throughout this work. We are also thankful to the University of Louisiana at Monroe and William Barnett for providing guidance on joining the internship.

References

[1] Subil Abraham, Arnab K. Paul, Redwan Ibne Seraj Khan, and Ali R. Butt. 2020. On the Use of Containers in High Performance Computing Environments. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. Institute of Electrical and Electronics Engineers(IEEE), 284–293. doi:10.1109/CLOUD49709.2020.00048

[2] Internet Data Center. 2018. IDC: Expect 175 zettabytes of data worldwide by 2025. Accessed 2024-07-22.

[3] Podman Docs. 2025. What is Podman? https://www.redhat.com/en/topics/containers/what-is-podman. Accessed 2025-04-02.

[4] Dave Dykstra. 2024. Apptainer Without Setuid. In *EPJ Web of Conferences*, Vol. 295. EDP Sciences, 07005.

[5] Berkeley EECS. 2021. Cyclegan datasets of apple and oranges. https://efrosgans.eecs.berkeley.edu/cyclegan/datasets/apple2orange.zip. Accessed 2024-08-26.

[6] Alessandro Fantini. 2016. *Virtualization technologies from hypervisors to containers: overview, security considerations, and performance comparisons.* Ph. D. Dissertation. University of Bologna. https://amslaurea.unibo.it/id/eprint/12846/

[7] Sabah Kadri, Andrea Sboner, Alexandros Sigaras, and Somak Roy. 2022. Containers in bioinformatics: applications, practical considerations, and best practices in molecular pathology. *The Journal of molecular diagnostics* 24, 5 (2022), 442–454.

[8] Igor Kononenko. 2001. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine* 23, 1 (2001), 89–109.

[9] Oak Ridge National Laboratory. 2022. Frontier User Guide. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html. Accessed 2024-07-17.

[10] Shen Li and Joe Zhu. 2024. Getting Started with Distributed Data Parallel. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html. Accessed 2024-07-29.

[11] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. 2022. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems* 33, 12 (2022), 6999–7019. doi:10.1109/TNNLS.2021.3084827

[12] Qing Liu, Ningyu Zhang, Wenzhu Yang, Sile Wang, Zhenchao Cui, Xiangyang Chen, and Liping Chen. 2017. A review of image recognition with deep convolutional neural network. In *Intelligent Computing Theories and Application: 13th International Conference, ICIC 2017, Liverpool, UK, August 7-10, 2017, Proceedings, Part I 13*. Springer, 69–80.

[13] Nina Mujkanovic, Juan J Durillo, Nicolay Hammer, and Tiziano Müller. 2023. Survey of adaptive containerization architectures for HPC. In *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 165–176.

[14] Tuan Dung Nguyen, Yuan-Sen Ting, Ioana Ciucă, Charlie O'Neill, Ze-Chang Sun, Maja Jabłońska, Sandor Kruk, Ernest Perkowski, Jack Miller, Jason Li, et al. 2023. Astrollama: Towards specialized foundation models in astronomy. *arXiv preprint arXiv:2309.06126* (2023).

[15] Péter Polgár, Tamás Menyhárt, Csanád Bátor Baksay, Gergely Kocsis, Tibor Gábor Tajti, and Zoltán Gál. 2023. Three level benchmarking of Singularity containers for scientific calculations. In *Annales Mathematicae et Informaticae*, Vol. 58. 133–146.

[16] Portworx and Aqua Security. 2019. 2019 Container Adoption Survey. https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf. Accessed 2024-07-22.

[17] pytorch docs. 2023. DataParallel. https://pytorch.org/docs/stable/generated/torch.nn.DataParallel.html. Accessed 2024-08-12.

[18] pytorch docs. 2023. DistributedDataParallel. https://pytorch.org/docs/stable/generated/torch.nn.parallel.DistributedDataParallel.html. Accessed 2024-08-12.

[19] Santosh Kumar Sahu, Anil Mokhade, and Neeraj Dhanraj Bokde. 2023. An overview of machine learning, deep learning, and reinforcement learning-based techniques in quantitative finance: recent progress and challenges. *Applied Sciences* 13, 3 (2023), 1956.

[20] Sachchidanand Singh and Nirmala Singh. 2016. Containers & Docker: Emerging roles & future of Cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*. 804–807. doi:10.1109/ICATCCT.2016.7912109

[21] Mohammad Mustafa Taye. 2023. Theoretical understanding of convolutional neural network: Concepts, architectures, applications, future directions. *Computation* 11, 3 (2023), 52.

[22] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[23] Mario Vestias and Horácio Neto. 2014. Trends of CPU, GPU and FPGA for high-performance computing. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 1–6.

[24] Meiyin Wu and Li Chen. 2015. Image recognition based on deep learning. In *2015 Chinese automation congress (CAC)*. IEEE, 542–546.

[25] Kejiang Ye, Yanmin Kou, Chengzhi Lu, Yang Wang, and Cheng-Zhong Xu. 2018. Modeling application performance in docker containers using machine learning techniques. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 1–6.

[26] Andrew J Younge, Kevin Pedretti, Ryan E Grant, and Ron Brightwell. 2017. A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 74–81.

Received 08 April 2025; revised 12 March 2009; accepted 5 June 2009