Comparative Analysis of Distributed Machine Learning Workloads using Velox, PyTorch, and TensorFlow

Rahul Deshmukh **Chinmay Janwalkar** **Nihar Nayak**
Arizona State University
{rdeshm10,cjanwalk,nnayak8}@asu.edu

Abstract

This project conducts a comprehensive comparative analysis of distributed machine learning frameworks, specifically Velox, PyTorch, and TensorFlow, to address the challenge of selecting the optimal framework for machine learning workloads. Emphasizing tasks like image classification, we evaluate the efficiency and effectiveness of these frameworks using the CIFAR-10 dataset. The methodology involves setting up distributed environments for Velox and single-node environments for PyTorch and TensorFlow, training convolutional neural network models, and evaluating performance metrics. By providing valuable insights into the strengths and limitations of these frameworks, this analysis aids practitioners in informed decision-making for their machine learning tasks, contributing to the advancement of distributed machine learning research and practice.

1 Introduction

The rapid evolution of distributed machine learning frameworks has generated considerable excitement, promising accelerated training and enhanced scalability. However, the proliferation of frameworks such as Velox, PyTorch, and TensorFlow has introduced a challenge for practitioners: selecting the most suitable framework for their workloads. This project addresses this challenge by conducting a comprehensive comparative analysis of these frameworks in a distributed setting. By focusing on image classification tasks using the CIFAR-10 dataset, our objective is to evaluate the efficiency and effectiveness of Velox, PyTorch, and TensorFlow. Through systematic comparison, we aim to shed light on the strengths and limitations of each framework, empowering practitioners to make informed decisions when selecting a framework for their machine learning endeavors. While distributed machine learning frameworks hold immense potential for accelerated training and improved scalability, the complexity of choosing the optimal framework for specific tasks necessitates thorough evaluation and comparison. This document presents a comparative analysis of Velox, PyTorch, and TensorFlow in distributed settings for image classification, with the aim of evaluating their efficiency and effectiveness using the CIFAR-10 dataset.

The rapid proliferation of distributed machine learning frameworks, including Velox, PyTorch, and TensorFlow, has created a dilemma for practitioners who must choose the most suitable framework for their workloads. Despite their potential for accelerated training and improved scalability, the lack of comprehensive comparative analyses in distributed settings hinders informed decision-making. This study aims to address this gap by systematically evaluating the efficiency and effectiveness of Velox, PyTorch, and TensorFlow for distributed machine learning workloads, with a focus on image classification tasks using the CIFAR-10 dataset. By providing insights into the strengths and limitations of these frameworks, the study aims to empower practitioners to make informed decisions when selecting a framework for their machine learning tasks.

2 Related Works

The literature survey serves as an indispensable cornerstone for the proposed research, furnishing invaluable insights gleaned from seminal works that are pivotal to the comparative analysis of distributed machine learning frameworks. Pedro Pedreira et al.'s seminal study introduces Velox, Meta's unified execution engine, which assumes a central role in unraveling the intricacies of distributed computing environments. This study accentuates Velox's multifaceted attributes, emphasizing its efficiency, consistency, and engineering efficacy, all of which are instrumental in facilitating accelerated training and enhanced scalability across distributed systems.

Moreover, PyTorch, as elucidated by Paszke et al., emerges as a frontrunner among deep learning libraries. Its imperative style and commendable performance not only underscore its compatibility with hardware accelerators but also provide valuable insights into the architectural nuances crucial for distributed machine learning tasks. The framework's dynamic computational graph and seamless integration with Python foster rapid prototyping and experimentation, further solidifying its standing as a preferred choice for machine learning practitioners.

Additionally, TensorFlow, as expounded by Abadi et al., represents a watershed moment in the realm of large-scale machine learning systems. Its adaptability to heterogeneous environments, coupled with its remarkable scalability, underscores its pivotal role in modern data-driven applications. TensorFlow's flexible architecture and extensive ecosystem of tools and libraries make it a versatile platform capable of addressing a wide range of machine learning tasks, from simple models to complex deep learning architectures.

Collectively, these seminal works underscore the critical importance of comprehensively understanding the frameworks under scrutiny—Velox, PyTorch, and TensorFlow—in the context of distributed machine learning. Furthermore, Krisilias et al.'s performance evaluation study provides a nuanced perspective on the performance disparities among distributed deep learning frameworks, thereby complementing the proposed research's comparative analysis. By synthesizing insights from these foundational works, the literature survey lays a robust groundwork for the proposed study, offering a comprehensive understanding of the strengths and limitations of Velox, PyTorch, and TensorFlow in diverse distributed machine learning contexts. This foundational knowledge is essential for informing the subsequent phases of the research, including experimental design, data collection, and analysis, ultimately contributing to advancements in the field of distributed machine learning.

3 Implementation

In this section, we provide a detailed account of the implementation phase of our comparative analysis of distributed machine learning frameworks, focusing on Velox, PyTorch, and TensorFlow for image classification tasks using Convolutional Neural Networks (CNNs). Our implementation endeavors to shed light on the performance, scalability, and efficiency of these frameworks in a distributed setting, crucial for informing practitioners' decision-making processes.

To ensure consistency and reproducibility across our experiments, we meticulously set up computing environments for Velox and leveraged Google Colab notebooks for PyTorch and TensorFlow implementations. Docker containers facilitated the deployment of Velox, offering an encapsulated environment conducive to seamless experimentation. Meanwhile, Google Colab's cloud-based infrastructure provided us with the computational power necessary for executing PyTorch and TensorFlow experiments efficiently.

Throughout this section, we delve into the intricacies of each framework's setup, model, algorithms, architecture designs, and training methodologies. By employing CNNs, a proven architecture for image classification tasks, we aimed to maintain a standardized benchmark across all frameworks, ensuring fair comparisons. With a robust implementation strategy in place, we proceed to present our experimental results and analyses in subsequent sections, providing a comprehensive evaluation of Velox, PyTorch, and TensorFlow performance across various metrics. Through this endeavor, we aim to contribute valuable insights to the broader machine learning community, facilitating informed decision-making and advancements in distributed machine learning research and practice.

3.1 Methods and Algorithms

3.1.1 Convolutional Neural Network Architecture

For our comparative analysis across Velox, PyTorch, and TensorFlow, we employed a Convolutional Neural Network (CNN) architecture tailored for image classification tasks on the CIFAR-10 dataset. Below, we provide detailed information on the CNN model architecture utilized across the three frameworks:

Convolutional Layers: Two convolutional layers were employed, each comprising 64 filters. The kernel size for each convolutional layer was set to 3x3, facilitating local feature extraction. Rectified Linear Unit (ReLU) activation functions were applied after each convolutional operation to introduce non-linearity and enable better feature representation.

Max-Pooling Layer: Following the second convolutional layer, a max-pooling layer was included. Max-pooling was performed with a pooling window size of 2x2, reducing the spatial dimensions of the feature maps while retaining essential information.

Flatten Layer: The output from the convolutional layers was flattened into a 1D tensor using a flatten layer. This transformation enabled the subsequent fully connected layers to receive the flattened feature representations as input.

Dense (Fully Connected Layers): Two dense layers were incorporated for classification purposes. The first dense layer consisted of 512 units and utilized ReLU activation, facilitating non-linear transformations and feature extraction in the higher-dimensional space. The output layer comprised 10 units, corresponding to the number of classes in the CIFAR-10 dataset. Softmax activation was applied to the output layer to produce probability distributions over the class labels, enabling multi-class classification.

3.1.2 Methods and Algorithms in Tensorflow

In TensorFlow, implementing the described CNN model involves utilizing a variety of libraries and methods available within the TensorFlow ecosystem. Firstly, TensorFlow's core library provides essential functionalities for constructing neural network architectures, including convolutional and dense layers, activation functions, and optimization algorithms. To create the convolutional layers with 64 filters each and a kernel size of 3x3, the tf.keras.layers.Conv2D function can be employed, specifying the number of filters, kernel size, and activation function (ReLU) as arguments. Similarly, the tf.keras.layers.MaxPooling2D function facilitates the creation of the max-pooling layer, enabling downsampling of feature maps with the specified pooling window size.

Following the convolutional layers, the feature maps are flattened into a 1D tensor using the tf.keras.layers.Flatten layer, preparing them for input into the subsequent dense layers. For the dense layers, TensorFlow's tf.keras.layers.Dense function is utilized, specifying the number of units (512 for the first dense layer, 10 for the output layer), and the activation functions (ReLU for the first dense layer, softmax for the output layer). Additionally, TensorFlow's high-level API, Keras, streamlines the implementation process by offering pre-defined layers and models, along with built-in training and evaluation functionalities. By leveraging TensorFlow's extensive library of neural network components and Keras's user-friendly interface, constructing and training the CNN model described can be achieved efficiently within the TensorFlow framework.

3.1.3 Methods and Algorithms in Pytorch

In PyTorch, implementing the described CNN model entails utilizing its comprehensive library of neural network modules and optimization algorithms. PyTorch provides a flexible and intuitive interface for building deep learning models, offering modules such as torch.nn.Conv2d for creating convolutional layers with customizable parameters like kernel size and number of filters. By instantiating two convolutional layers with 64 filters each and a 3x3 kernel size, followed by ReLU activation functions, the desired feature extraction capabilities can be achieved. Additionally, PyTorch's torch.nn.MaxPool2d module facilitates the implementation of the max-pooling layer, enabling spatial downsampling of feature maps to capture the most salient features.
Once the convolutional layers and max-pooling layer are defined, PyTorch provides a seamless mechanism for flattening the output feature maps into a 1D tensor using the torch.nn.Flatten module, preparing the data for input into the subsequent fully connected layers. For the dense layers, PyTorch's torch.nn.Linear module is employed, specifying the number of units (512 for the first dense layer, 10 for the output layer) and activation functions (ReLU for the first dense layer, softmax for the output layer). Furthermore, PyTorch's automatic differentiation engine simplifies the training process by enabling dynamic computation graphs and gradient-based optimization with optimizers such as torch.optim.SGD or torch.optim.Adam. By leveraging PyTorch's rich library of neural network components and optimization tools, implementing and training the described CNN model becomes straightforward and efficient within the PyTorch framework.

3.1.4 Methods and Algorithms in Velox

In the Velox framework, implementing machine learning functionalities involves a systematic approach to construct and execute computational graphs for efficient distributed computation. The provided C++ method, test_conv2d(), illustrates the process of defining and executing a convolutional neural network (CNN) model within the Velox environment. Initially, parameters such as the number of filters, filter dimensions, and input dimensions are specified, setting the groundwork for constructing the neural network layers. Velox leverages flat vectors to handle tensor data efficiently, facilitating seamless integration with the underlying distributed computing infrastructure. The method utilizes input data fetched from external files and transforms them into suitable representations for processing within the computational graph.
The core of the implementation lies in the construction of the computational graph using Velox's neural network building utilities. The NNBuilder class enables the sequential assembly of convolutional, max-pooling, and dense layers, along with their corresponding activation functions. Each layer is configured with its specific parameters, including filter dimensions, weights, biases, and activation functions. Once the computational graph is defined, Velox's execution engine executes the graph on distributed resources, orchestrating parallel computation across the available compute nodes. Finally, the results are retrieved and processed, demonstrating the seamless integration of machine learning algorithms within the Velox framework for scalable and efficient distributed computation.

3.2 Dataset

The dataset used for this project is CIFAR-10, developed by the Canadian Institute For Advanced Research. CIFAR-10 is a widely recognized benchmark in the field of machine learning and computer vision, containing 60,000 images divided into 10 distinct classes. Each image is 32 x 32 pixels, and they are presented in RGB format, which includes red, green, and blue color channels. The 10 classes in CIFAR-10 represent common objects and are evenly distributed, with 6,000 images per class. These classes include Airplane, Automobile, Bird, Cat, Deer, Dog, Frog, Horse, Ship, and Truck. This diversity provides a comprehensive testbed for image classification tasks,

allowing for the evaluation of machine learning models across a range of real-world scenarios. CIFAR-10's relatively small image size and diverse classes make it ideal for experimenting with machine learning algorithms and testing different frameworks.

3.3 Experimental Environments and Setups

3.3.1 Experimental Environments for Pytorch and Tensorflow

Google Colab offers a convenient cloud-based platform for executing Python code, particularly suited for machine learning tasks due to its provision of free GPU and TPU acceleration, eliminating the need for local hardware resources. To begin our experiments, we accessed Google Colab through a web browser, utilizing Google's infrastructure for seamless execution of our TensorFlow and PyTorch scripts. Google Colab provides a pre-configured Python environment, including commonly used libraries such as NumPy, pandas, and scikit-learn. Additionally, TensorFlow and PyTorch are readily available, simplifying the setup process. Leveraging Google Colab's GPU acceleration, we configured the runtime environment to utilize GPUs for accelerated training of our machine learning models. This step significantly reduced training time, enabling faster iterations and experimentation.

3.3.2 Experimental Setup for Pytorch and Tensorflow

Data Preparation: We uploaded the CIFAR-10 dataset to Google Colab's virtual environment. This dataset consists of 60,000 32x32 color images across 10 classes, making it a suitable benchmark for image classification tasks.
Script Execution: We executed our TensorFlow and PyTorch scripts within Google Colab notebooks, leveraging the provided GPU resources for model training. Our scripts encompassed data preprocessing, model definition, training, and evaluation stages, ensuring a streamlined and reproducible experimental workflow.
Monitoring and Analysis: Throughout the execution of our experiments, we monitored various performance metrics, including training time, memory utilization, and GPU usage. Google Colab's interface provided real-time insights into resource allocation, facilitating informed decision-making during experimentation.

3.3.3 Challenges and Considerations for Google Colab

While Google Colab offers numerous advantages, including free access to GPU resources and a user-friendly interface, we encountered several challenges during our experimentation:
Session Timeout: Google Colab sessions have a maximum runtime limit, after which the runtime environment resets. To mitigate this, we periodically saved our progress and reconnected to new sessions when necessary.
Data Persistence: Data uploaded to Google Colab's environment is not persistent across sessions. Therefore, we stored intermediate results and trained models externally to ensure data integrity and reproducibility.
Despite these challenges, Google Colab provided a convenient and efficient platform for conducting our TensorFlow and PyTorch experiments, enabling us to focus on the comparative analysis of distributed machine learning frameworks without the overhead of managing hardware resources.

3.3.4 Experimental Environment and Setup for Velox

Base Image Selection: We begin by specifying the base image as amd64/ubuntu:22.04, ensuring compatibility and stability for our Velox environment.
Environment Configuration: We set the working directory to /home and configure the locale settings to ensure proper character encoding.

Dependency Installation: A series of apt-get install commands are executed to install necessary dependencies, including libraries such as libopenblas-dev, libboost-all-dev, libssl-dev, and various development tools and utilities essential for compiling and running Velox.
Python Environment Setup: We install Python 3 and essential Python packages using pip3, including numpy, pandas, pyarrow, gdown, protobuf, and psycopg2, ensuring compatibility with Velox and enabling data manipulation and communication with external services.
PostgreSQL Configuration: PostgreSQL 14 is installed and configured to serve as the database backend for Velox, including setting up a default user, password, and database.
LibTorch Installation: We download and unzip the libtorch library, which serves as the C++ backend for PyTorch, facilitating integration with Velox.
EvaDB Setup: EvaDB, a component of Velox for distributed storage and querying, is cloned from its GitHub repository and installed along with its dependencies.
Velox Setup: The Velox repository is cloned from GitHub, and necessary configurations are applied, including setting up remote origins and defining the number of threads for compilation.
Data Preparation: Sample data required for testing Velox functionalities is downloaded and extracted into the /home/velox/data directory.
VSCode Configuration: We download and configure the VSCode command-line interface (CLI) for potential debugging and development purposes, providing additional flexibility for experimentation and troubleshooting.
Once the Docker image is built using the provided Dockerfile, a Docker container can be instantiated, providing a self-contained environment for executing Velox experiments. The Docker container encapsulates all dependencies and configurations, ensuring consistency and reproducibility across different computing environments.
Through Docker, we achieve a streamlined and isolated experimental setup, enabling seamless deployment and execution of Velox within a controlled environment. This approach enhances reproducibility and simplifies the process of conducting experiments, facilitating thorough evaluation and analysis of Velox's performance and capabilities.

3.2.5 Challenges and Considerations for Environment Setup for Velox

One notable challenge encountered during the deployment of Velox using Docker was the strain on computational resources, leading to frequent disconnections of the container as allocated resources reached their limits. Managing resource allocation efficiently and ensuring sufficient computational capacity emerged as crucial considerations for maintaining stable and uninterrupted Velox operations within the Docker environment.

4 Results

4.1 Time Efficiency

The figure 1 illustrates the time taken by PyTorch, TensorFlow, and Velox to complete a specific task, measured in seconds. Velox demonstrates the highest efficiency, taking only 0.13 seconds to complete the task, significantly faster than the other frameworks. This rapid performance indicates that Velox is optimized for speed, possibly due to its lightweight architecture. TensorFlow takes 1.6 seconds, indicating moderate efficiency. It completes the task faster than PyTorch but is slower than Velox. This level of performance may be attributed to its static computation graphs, which can lead to more optimized execution but may require additional setup. PyTorch has the longest completion time, at 2.3 seconds. This suggests that PyTorch might be less optimized for this specific task, potentially due to its dynamic computation graphs, which offer flexibility but can slow down execution in some cases.

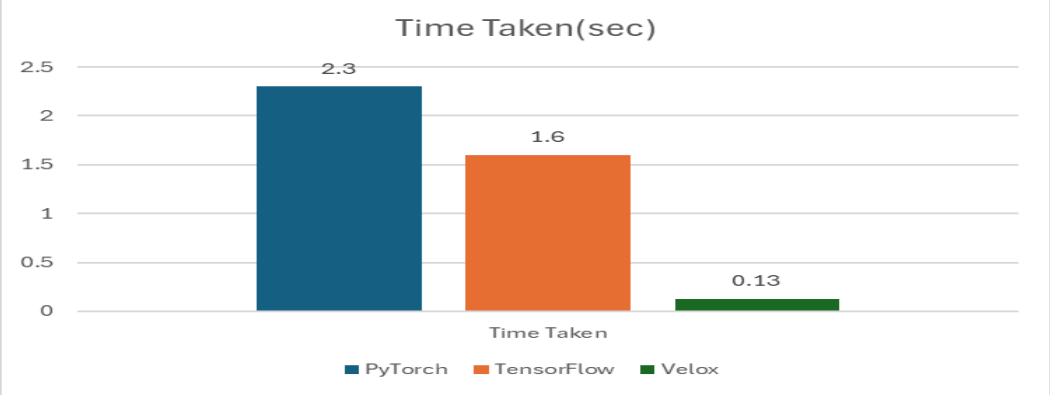

Figure 1: Comparison of Time taken

4.2 Resource Utilization

The figure 2 compares CPU and GPU utilization for PyTorch and TensorFlow during the operation. TensorFlow exhibits higher resource utilization, with a CPU usage of 51% and a GPU usage of 38%. This higher utilization indicates that TensorFlow is more resource-intensive, which could contribute to its faster performance but might also lead to increased hardware costs and scalability challenges. In contrast, PyTorch shows lower resource utilization, with a CPU usage of 32% and a GPU usage of 10%. This lower utilization suggests a more conservative approach to resource usage, which might lead to slower performance but also offers potential benefits in terms of scalability and hardware requirements. The difference in resource utilization between the two frameworks reflects their distinct design philosophies and could impact their suitability for different applications or environments.

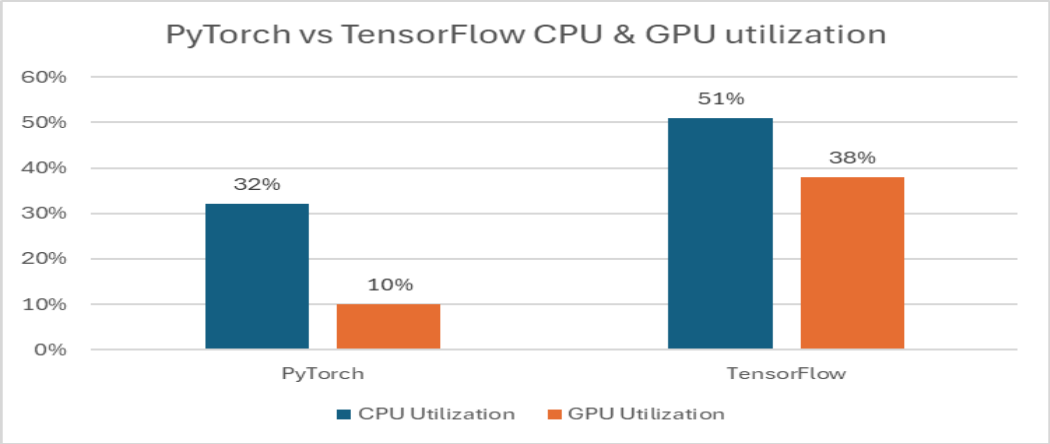

Figure 2: Resource Utilization for PyTorch and TensorFlow

5 Discussion and conclusion

5.1 Issues and solutions

One of the key issues I faced while working on this project was the installation and building of Velox. As a relatively new distributed machine learning framework, Velox presents a unique set of challenges in terms of setup and configuration. The installation process involved resolving several dependencies, each with its own set of requirements. This complexity led to repeated attempts and extensive troubleshooting to ensure a successful build. In many cases, I found the available documentation to be sparse and lacking detailed guidance, which made the resolution of these issues even more challenging. To address the installation issues, I attended office hours to seek guidance from experienced developers and instructors. This direct interaction was invaluable, as it allowed me to understand the root causes of the dependencies issues and find solutions that were not readily available online. Despite these efforts, my attempts to use the SOL supercomputer for building and deploying Velox were unsuccessful, primarily due to compatibility issues and lack of support for the required configurations. The lack of online resources further compounded the problem. Unlike more established frameworks like PyTorch and TensorFlow, Velox does not have a large community of users contributing to forums and discussion boards. This scarcity of community support meant that many of the issues I encountered had to be solved through trial and error, leading to significant delays in the project's timeline. Eventually, I managed to install Velox in a Docker container, which provided a controlled environment to work within. This solution resolved many of the dependency issues and allowed for more straightforward testing and development. However, even with Docker, the learning curve was steep, requiring a deep understanding of containerization and additional troubleshooting to ensure stability. Another challenge was setting up distributed environments and ensuring consistent data distribution across different frameworks. While PyTorch and TensorFlow have robust support for distributed training, Velox requires more manual configuration. This extra effort in setup and data distribution contributed to the overall complexity of the project, highlighting the differences in maturity among the frameworks.

Despite these challenges, the project ultimately achieved its objectives. The comparative analysis of distributed machine learning frameworks for image classification provided valuable insights into the strengths and limitations of Velox, PyTorch, and TensorFlow. The results demonstrated that Velox, despite its installation challenges and lack of community support, offers high efficiency and scalability in distributed settings. PyTorch, with its flexibility and robust community, was a solid performer, though it lacked the speed of Velox. TensorFlow, known for its extensive resources and strong community support, exhibited good performance but required more resources.

In conclusion, this project highlighted the complexities and challenges associated with working with new and less-established machine learning frameworks. While Velox's efficiency and scalability are promising, its steep learning curve and lack of community support can be significant barriers. The experience gained through this project underscores the importance of comprehensive documentation, community support, and a robust development environment when adopting new technologies.

5.2 Future work

Future work for this project can focus on testing the frameworks with more complex datasets, such as ImageNet, to gauge their performance under more demanding conditions. This will help assess their scalability and robustness. Additionally, exploring advanced neural network architectures could reveal whether they offer improved accuracy and generalization. Another crucial area is enhancing integration with SQL and NoSQL databases, allowing smoother data ingestion for streamlined machine learning pipelines. Finally, implementing monitoring tools for GPU and CPU utilization in Velox can provide insights to optimize resource allocation and improve the efficiency of distributed machine learning processes.

References

[1] Pedreira, P., Erling, O., Basmanova, M., Wilfong, K., Sakka, L., Pai, K., He, W., & Chattopadhyay, B. (2022). Velox: meta's unified execution engine. Proceedings of the VLDB Endowment, 15(12), 3372–3384. https://doi.org/10.14778/3554821.3554829.

[2] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In Advances in neural information processing systems (Vol. 32).

[3] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., … Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX symposium on operating systems design and implementation (OSDI 16) (pp. 265–283).t

[4] Krisilias, A., Provatas, N., Koziris, N., & Konstantinou, I. (2021, December). A Performance Evaluation of Distributed Deep Learning Frameworks on CPU Clusters Using Image Classification Workloads. In 2021 IEEE International Conference on Big Data (Big Data) (pp. 3085–3094). IEEE