# Benchmarking Data Parallel vs Model Parallel Training with PyTorch and VeloxML

Poojah Ganesan, Rajat Aayush Jha, and Prasanna Venkateshan

Arizona State University

**Abstract.** The exponential proliferation of deep learning applications across various domains has precipitated a pressing need for adept parallelization techniques to expedite the training of deep neural network (DNN) models. In this comprehensive report, we embark on an in-depth exploration and comparative analysis of two prominent parallelization methodologies within the realm of distributed deep learning: data parallelism and model parallelism. Our overarching objective is to scrutinize the efficacy of these methodologies in facilitating the training of DNN models, leveraging the versatile PyTorch framework and the specialized distributed deep learning platform VeloxML.
Through meticulous experimentation and rigorous evaluation, we endeavor to assess and benchmark the performance of data parallel and model parallel training techniques across an array of DNN architectures and dataset sizes. Our systematic comparison aims to illuminate the unique strengths, limitations, and trade-offs inherent in each parallelization approach. Key metrics under scrutiny encompass training time, utilization of computational resources, scalability in handling large-scale datasets and models, as well as overall performance metrics. By undertaking this comparative analysis, we aspire to provide valuable insights that elucidate the optimal selection of parallelization strategies tailored to specific application requirements and constraints within the distributed deep learning landscape. We leverage the utilization of CUDA and GPU acceleration to enhance the efficiency of parallelized training.

**Keywords:** PyTorch · Data Parallelization · Model Parallelization · VeloxML · CUDA · GPU

## 1 Introduction

Deep learning has revolutionized various fields such as computer vision, natural language processing, and reinforcement learning, among others. However, training deep neural network models often requires significant computational resources and time, especially for large-scale datasets and complex architectures. To address this challenge, parallelization techniques play a crucial role in distributing the computational workload across multiple processing units, thereby accelerating the training process.

In distributed deep learning, two commonly used parallelization approaches are data parallelism and model parallelism. Data parallelism involves replicating

the model across multiple devices or nodes and dividing the dataset into smaller batches to be processed in parallel. On the other hand, model parallelism partitions the model itself across different devices or nodes, with each part responsible for computing a specific portion of the network.

In this technical report, we aim to investigate and compare the performance of these parallelization techniques using the PyTorch and VeloxML frameworks. Our primary goal is to evaluate the effectiveness of data parallel and model parallel training methods across a diverse set of deep neural network architectures and dataset sizes. By conducting systematic experiments and performance evaluations, we seek to provide insights into the strengths, weaknesses, and trade-offs of each approach in terms of training time, resource utilization, scalability, and overall performance. Ultimately, our findings will contribute to a better understanding of parallelization strategies in distributed deep learning and guide practitioners in selecting the most suitable approach for their specific applications and requirements.

## 2 Related Work

Several research works have explored the trade-offs between data parallel and model parallel training for deep learning. Younesy et al. [5] meticulously dissect the intricate trade-offs inherent in data parallel and model parallel training methodologies within the deep learning domain. Their comparative analysis highlights the nuanced strengths and limitations of each approach across diverse training scenarios, shedding light on optimal strategies for leveraging parallelism techniques effectively. On a parallel track, Kasten et al. [2] introduce Velox, an innovative platform expressly designed to address the multifaceted challenges associated with both data and model parallelism. Their work provides a robust foundation for understanding and implementing efficient parallelization strategies, particularly within the PyTorch and VeloxML frameworks, aimed at accelerating deep learning tasks. In a related vein, Tan et al. [4] narrow their focus to the specific domain of training large language models, offering valuable insights into the adaptability of parallelism techniques like PyTorch and VeloxML training in this context. Meanwhile, Zheng et al. [7] contribute a broad-ranging survey that navigates the complex landscape of parallel training techniques, facilitating nuanced comparisons between diverse implementations of data and model parallelism. Their comprehensive overview serves as a cornerstone for discerning the optimal strategies tailored to the unique requirements and constraints of PyTorch and VeloxML frameworks. More recently, Zhang et al. [6] push the boundaries of efficiency with their exploration of communication-efficient training through hierarchical model parallelism. Their work opens new avenues for investigation, prompting inquiries into the applicability of such techniques within PyTorch and VeloxML training paradigms, and their comparative efficacy vis-à-vis conventional data and model parallelism strategies.

## 3 Data Parallelization in Distributed Deep Learning

Data parallelization is a pivotal strategy in distributed deep learning, designed to accelerate model training across multiple processing units by distributing the workload and data across different compute nodes or GPUs. The fundamental principle behind data parallelism involves partitioning the dataset into smaller subsets, with each subset processed independently by a separate compute unit. Concurrently, the model parameters are replicated across all nodes to ensure consistent updates and synchronization during training.

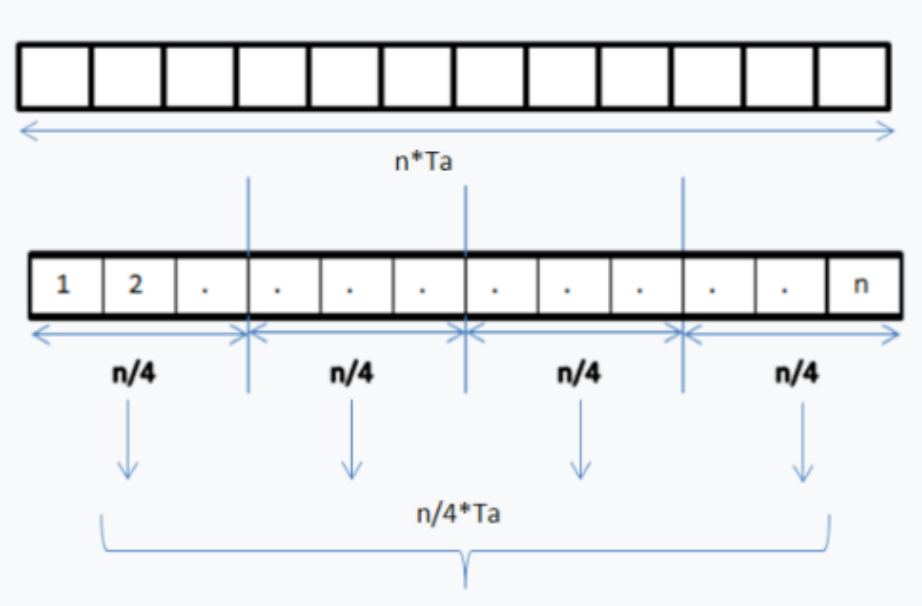


**Fig. 1.** Sequential vs. Data-Parallel job execution. *(Source: Wikipedia)*

Mathematically, in the context of stochastic gradient descent (SGD), the update rule for model parameters $\theta$ in data parallelism can be represented as follows:

$$\theta_{t+1} = \theta_t - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \nabla L(f_\theta(x_i), y_i)$$

Here, $\theta_t$ and $\theta_{t+1}$ denote the model parameters at time step $t$ and $t+1$ respectively, $\eta$ represents the learning rate, $N$ denotes the batch size, $f_\theta(x_i)$ signifies the model's prediction on input $x_i$, $y_i$ represents the ground truth label, and $L$ represents the loss function. The gradient of the loss function $\nabla L(f_\theta(x_i), y_i)$ is computed independently for each data point $(x_i, y_i)$ within the batch. After

computing gradients locally on each node, they are synchronized and aggregated across all nodes to update the global model parameters $\theta$.

Data parallelism offers several advantages, including:

1. **Scalability**: By distributing the dataset and computation across multiple nodes or GPUs, data parallelism enables training on larger datasets and models that may not fit into the memory of a single device.
2. **Speedup**: Concurrent processing of data subsets on multiple nodes accelerates the training process, leading to faster convergence and reduced training time.
3. **Fault Tolerance**: The replication of model parameters across nodes enhances fault tolerance, as the failure of one node does not significantly impact the overall training process.

However, data parallelism also introduces challenges such as:

1. **Communication Overhead**: Synchronization and communication overheads associated with aggregating gradients across nodes can become a bottleneck, particularly with large model sizes and frequent updates.
2. **Memory Constraints**: Replicating model parameters across nodes requires significant memory resources, posing limitations on the size of models that can be effectively trained in a distributed setup.
3. **Synchronization Issues**: Ensuring consistent updates and synchronization of gradients across nodes necessitates efficient communication protocols to mitigate delays and maintain model consistency.

In summary, while data parallelism offers substantial benefits in terms of scalability and speedup for training large-scale deep learning models, addressing communication overheads, memory constraints, and synchronization issues is crucial to realize its full potential and achieve optimal performance in distributed training scenarios.

## 4 Model Parallelization in Distributed Deep Learning

Model parallelization is a strategic approach in distributed deep learning aimed at accelerating the training of large-scale models by partitioning the model architecture across multiple processing units. Unlike data parallelization, where data is distributed across nodes, model parallelization involves distributing the computational graph of the model across different devices or GPUs. This partitioning enables the concurrent execution of different parts of the model on separate devices, facilitating the training of models that may not fit into the memory of a single device.

In model parallelism, the model parameters are distributed across different nodes, and each node is responsible for computing the gradients for a specific subset of parameters. During the forward pass, the input data propagates through different parts of the model hosted on different nodes. During the backward pass,

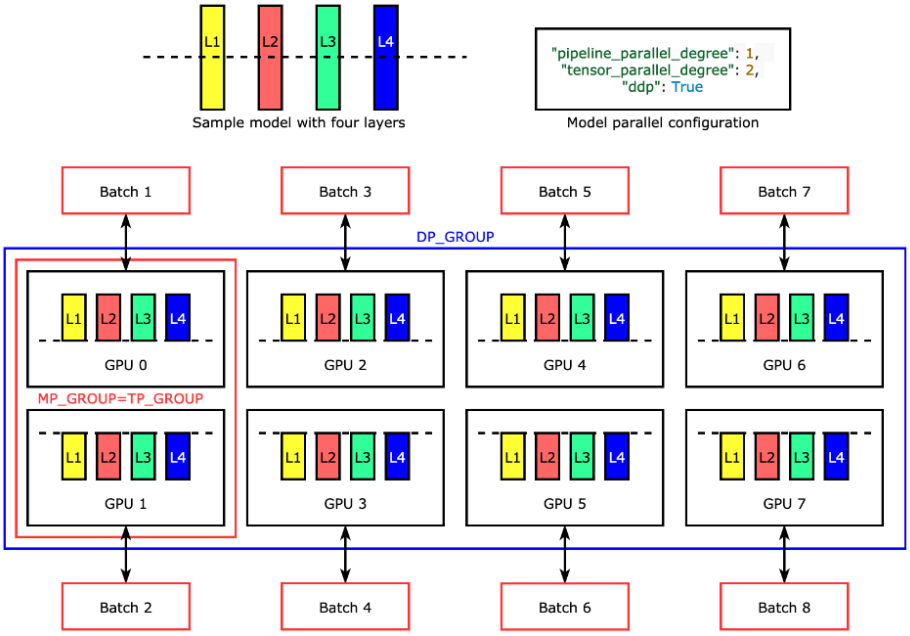


**Fig. 2.** Model Parallelization - for tensors. *(Source: AWS Sagemaker Documentation)*

the gradients are computed locally on each node and then communicated and aggregated across nodes to update the global model parameters.

Mathematically, the update rule for model parameters $\theta$ in model parallelism can be represented as follows:

$$\theta_{t+1} = \theta_t - \eta \cdot \sum_{i=1}^{N} \nabla L(f_{\theta_i}(x), y)$$

Here, $\theta_t$ and $\theta_{t+1}$ denote the model parameters at time step $t$ and $t+1$ respectively, $\eta$ represents the learning rate, $N$ denotes the number of partitions or nodes, $f_{\theta_i}(x)$ signifies the output of the model segment hosted on node $i$, and $L$ represents the loss function. The gradient of the loss function $\nabla L(f_{\theta_i}(x), y)$ is computed independently on each node using the local segment of the model.

Model parallelization offers several advantages, including:

1. **Memory Efficiency**: By partitioning the model across nodes, model parallelism enables the training of large models that may not fit into the memory of a single device.
2. **Scalability**: Model parallelism facilitates the parallel execution of different parts of the model across multiple nodes, allowing for scalability in handling large-scale models and datasets.

3. **Reduced Communication Overhead**: Unlike data parallelism, where gradients need to be synchronized across nodes, model parallelism involves localized gradient computations, reducing communication overhead.

However, model parallelism also presents challenges such as:

1. **Load Balancing**: Ensuring balanced computation and memory usage across nodes can be challenging, particularly for models with heterogeneous architectures.
2. **Synchronization**: Coordinating the computation and aggregation of gradients across nodes requires efficient synchronization mechanisms to maintain model consistency and convergence.
3. **Complexity**: Partitioning the model architecture and managing inter-node communication introduces additional complexity in the training process, requiring careful design and optimization.

In summary, while model parallelization offers advantages in terms of memory efficiency and scalability for training large-scale models, addressing challenges related to load balancing, synchronization, and complexity is essential to harness its full potential and achieve optimal performance in distributed training scenarios.

## 5 Integration of Velox

Velox is a data management library[3] developed as part of Facebook's Incubator projects, designed to enhance the performance of data-intensive applications. It specializes in high-performance data processing tasks, leveraging optimized, low-level data handling primitives suitable for a variety of systems. The library supports efficient data serialization, advanced memory management, and parallel execution capabilities, which are essential for managing and processing large datasets effectively. Velox also utilizes Boost libraries, which provide a comprehensive range of functionalities that improve file system operations and data serialization, contributing to the library's efficiency.

In our project, Velox was considered for optimizing the preprocessing stages of machine learning pipelines, especially for handling large-scale datasets. The goal was to utilize Velox's high-performance capabilities to reduce processing times and enhance data management efficiency.

Despite the anticipated benefits, the integration of Velox into our project encountered several challenges:

1. **Installation Difficulties**: We faced significant issues during the installation of Velox on local machines, which were equipped with Intel processors and ran on Ubuntu OS. This highlighted potential compatibility issues that could affect its broader adoption in similar data science setups.

2. **Documentation Shortfalls**: The documentation available for Velox was primarily focused on installation and basic configuration, lacking comprehensive guidance on its application in machine learning contexts. This was particularly challenging as the documentation did not provide high-level Python functions for essential data manipulation tasks, which are crucial for seamless integration into data science workflows.
3. **Limitation in Data Manipulation**: A significant drawback of Velox encountered during our project was its lack of support for high-level Python functions. These functions are crucial for data loading, filtering, transformation, and visualization—key steps in data preprocessing for machine learning. Without these capabilities, Velox necessitates alternative methods or additional coding, complicating the integration process and reducing overall efficiency for Python-centric data science workflows.

Eventually, Velox was successfully installed and deployed on Google Colab, facilitated by resources from the Facebook Incubator GitHub repository[1].

The exploration of Velox in our project highlighted its potential for improving the efficiency of data preprocessing tasks. Yet, the practical implementation challenges underscored the need for further development and adaptation of Velox to be more accessible and useful for machine learning applications. We plan to involve more of Velox in data pre-processing and improve the overall efficiency in the future.

## 6 Implementation Details

For our experimentation, we utilized ASU's High-Performance Computing (HPC) cluster equipped with NVIDIA A100 Tensor Core GPUs. These GPUs boast 80 GB of HBM2e memory and are interconnected through a high-speed NVLink bus, facilitating efficient communication between multiple GPUs. Our framework of choice for the experiments was PyTorch, a widely-used deep learning framework known for its flexibility and ease of use.

In our experimentation, we conducted two main sets of trials: data parallel and model parallel training. For data parallel training, we employed logistic regression, CNN, and ResNet 50 models, training them for 10, 30, 50, and 100 epochs. We varied the number of GPUs used in each experiment, testing configurations with 1, 2, and 4 GPUs. Additionally, we trained these models on both the CIFAR-10 and CIFAR-100 datasets to assess their performance across different data characteristics.

For model parallel training, we focused on CNN and ResNet 50 architectures, again training them for 10, 30, 50, and 100 epochs. However, in this case, we specifically investigated the impact of utilizing 2 and 4 GPUs for the training process. Similar to the data parallel experiments, we evaluated these models using both the CIFAR-10 and CIFAR-100 datasets to capture a comprehensive understanding of their behavior under varying conditions.

This table provides a concise overview of the hardware specifications, framework used, and the details of the experimentation process, including the types

| | |
|---|---|
| **Hardware Specifications** | ASU's HPC cluster |
| GPU | NVIDIA A100 Tensor Core GPU |
| GPU Memory | 80 GB HBM2e |
| Interconnection | High-speed NVLink bus |
| **Framework** | PyTorch |
| **Experimentation** | |
| **Training Types** | Data Parallel, Model Parallel |
| **Models** | Logistic Regression, CNN, ResNet 50 |
| **Epochs** | 10, 30, 50, 100 |
| **GPUs Used** | 1, 2, 4 |
| **Datasets** | CIFAR-10, CIFAR-100 |

**Table 1.**Experiment Details

of training, models employed, number of epochs, GPUs utilized, and datasets involved. Adjust the column widths and additional details as needed for your report.

## 7 Sequential vs Data Parallelization vs Model Parallelization

The following points are elaborated in Table 2

- Data parallelism is often the default choice for large datasets that can fit on each GPU memory.
- Model parallelism becomes more relevant when dealing with models that are too large to fit on a single GPU.
- Newer techniques like hierarchical model parallelism can further improve communication efficiency in model parallel training.

## 8 Results and Analysis

The results section serves as the culmination of our exploration and experimentation into the parallelization techniques of data parallelism and model parallelism in the context of training deep learning models. In this section, we present a comprehensive analysis of the performance metrics obtained from our experiments across various configurations, including different models, epochs, and GPU setups. Our goal is to provide insights into the efficacy and scalability of each parallelization approach, shedding light on their relative strengths and weaknesses. Through a detailed examination of training times, resource utilization, and model performance metrics, we aim to offer a nuanced understanding of how these parallelization strategies impact the training process and the overall performance of deep learning models.

| Feature | Sequential Training | Data Parallel Training | Model Parallel Training |
|---|---|---|---|
| Model Distribution | Single Processing Unit | Distributed Across Multiple GPUs (all layers on each GPU) | Distributed Across Multiple GPUs (different layers on different GPUs) |
| Data Processing | Entire data batch for each layer | Split data batch, each GPU processes the entire batch for all layers | Split data batch, each GPU processes a portion of the model for the entire batch |
| Communication Overhead | Low | Low (minimal communication between GPUs) | High (data needs to be transferred between GPUs after each layer) |
| Scalability | Limited to small models and datasets | Suitable for large datasets | Suitable for very large models |
| Memory Requirements | Lower (entire model on single GPU) | Higher (all replicas of the model on each GPU) | Can be lower or higher depending on the model partitioning strategy |
| Training Speed | Slowest | Fastest for large datasets that fit on each GPU | Can be faster than data parallel for very large models, but communication overhead can be a bottleneck |
| Ease of Implementation | Simplest | Easier than model parallelism | Most complex, requires careful model partitioning and communication management |

**Table 2.** Comparison of training strategies

### 8.1 Data Parallelization

1. **Training Time**:
   - Generally, training time increases with the number of epochs for all models and datasets, which is expected as more epochs require more iterations through the training data.
   - With the same number of epochs, training time tends to increase as the number of GPUs increases. This is because, with more GPUs, there is additional overhead in data communication and synchronization between GPUs.
2. **Memory Usage**:
   - Memory usage increases with the number of GPUs due to the replication of the model across multiple GPUs.

- For larger models and datasets, the memory usage tends to be higher, especially when using more GPUs.

3. **Training Loss**:
   - The training loss generally decreases with increasing epochs, indicating that the model is learning from the data over time.
   - With the same number of epochs, the training loss may vary slightly with the number of GPUs used, but the differences are not significant.
4. **Test Loss**:
   - Similar to the training loss, the test loss tends to decrease with increasing epochs, showing that the model's generalization improves over time.
   - Differences in test loss between different GPU configurations are minimal, indicating that data parallelization does not significantly impact the test loss.
5. **Training Accuracy**:
   - Training accuracy tends to increase with the number of epochs, as expected, indicating better fitting of the model to the training data.
   - With the same number of epochs, training accuracy may vary slightly with the number of GPUs used, but the differences are not substantial.
6. **Test Accuracy**:
   - Test accuracy generally increases with the number of epochs, indicating improved performance of the model on unseen data.
   - The impact of the number of GPUs on test accuracy varies depending on the model and dataset but tends to be minimal.

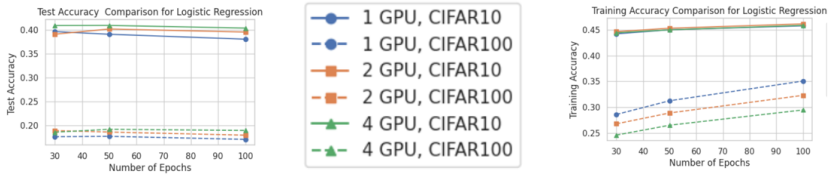


**Fig. 3.** Logistic Regression - Accuracy Comparison for Data Parallelization

### 8.2 Model Parallelization

1. **Training Time**:
   - The training time tends to increase with the number of epochs for all models and datasets, as expected, due to the additional iterations through the training data.

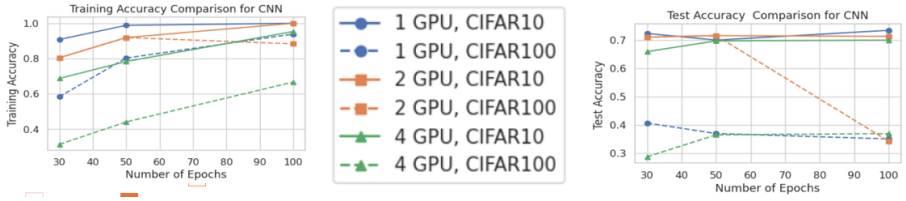


**Fig. 4.** CNN - Accuracy Comparison for Data Parallelization

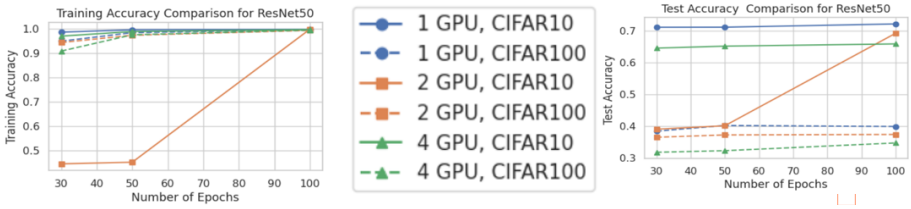


**Fig. 5.** ResNet50 - Accuracy Comparison for Data Parallelization

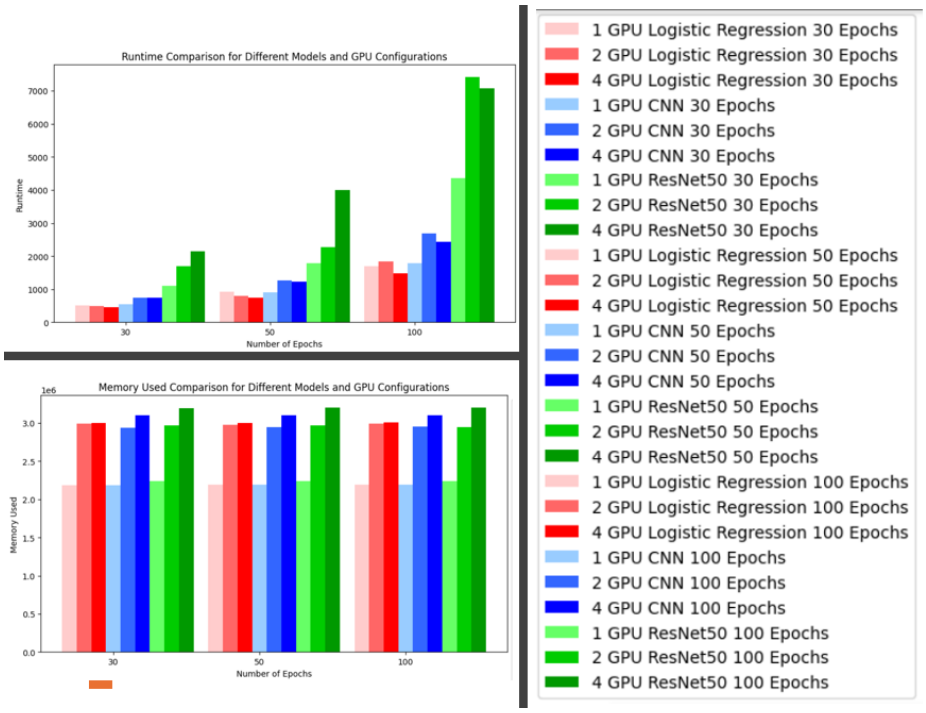


**Fig. 6.** Memory and Runtime Comparison for different models and GPU configurations for Data Parallelization

- With the same number of epochs, the training time increases as the number of GPUs increases. This is because model parallelization involves splitting the model across multiple GPUs, leading to increased communication overhead and synchronization.

2. **Memory Usage**:
   - Memory usage increases with the number of GPUs, particularly for larger models and datasets, due to the replication of model parameters across multiple GPUs.
   - The memory usage tends to be higher for larger models and datasets, especially when utilizing more GPUs for parallelization.
3. **Training Loss**:
   - The training loss generally decreases with increasing epochs, indicating improved model fitting to the training data over time.
   - Differences in training loss between different GPU configurations may occur but are generally minimal, indicating consistent learning across different parallelization setups.
4. **Test Loss**:
   - Similar to the training loss, the test loss tends to decrease with increasing epochs, indicating improved model generalization.
   - Differences in test loss between different GPU configurations are minor, suggesting that model parallelization does not significantly impact the test loss.
5. **Training Accuracy**:
   - Training accuracy tends to increase with the number of epochs, indicating better fitting of the model to the training data.
   - With the same number of epochs, training accuracy may vary slightly with the number of GPUs used, but the differences are generally negligible.
6. **Test Accuracy**:
   - Test accuracy generally increases with the number of epochs, indicating improved model performance on unseen data.
   - The impact of the number of GPUs on test accuracy varies depending on the model and dataset but tends to be minimal.

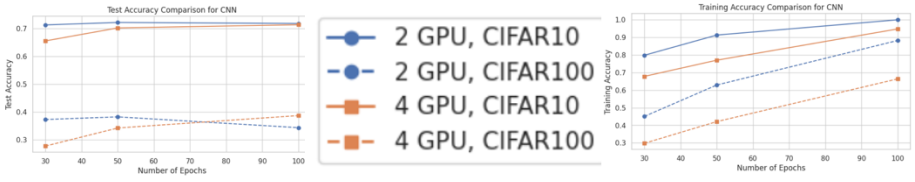


**Fig. 7.** CNN - Accuracy Comparison for Model Parallelization

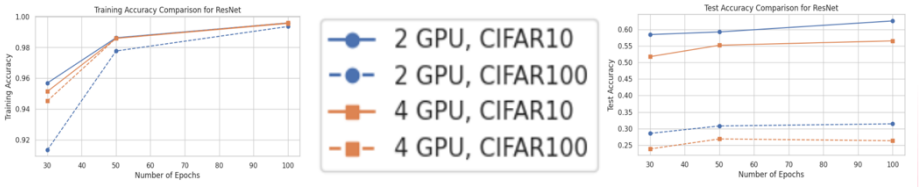


**Fig. 8.** ResNet50 - Accuracy Comparison for Model Parallelization

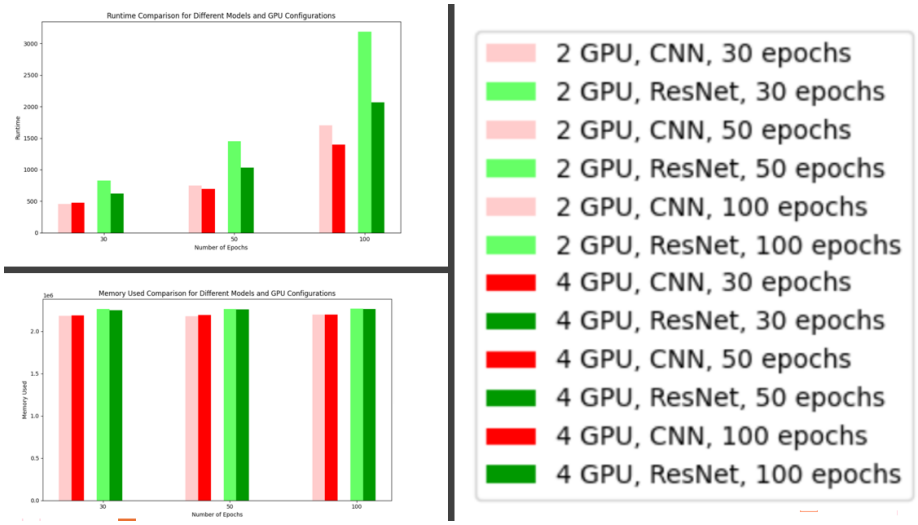


**Fig. 9.** Memory and Runtime Comparison for different models and GPU configurations for Model Parallelization

### 8.3 Results on ViT

The results obtained from experimenting with Vision Transformer (ViT) models reveal promising outcomes regarding their scalability and efficiency when employing both model and data parallelism techniques. ViT models exhibit commendable scalability with the adoption of parallelization methods, enabling the effective utilization of multiple GPUs or distributed computing resources. Leveraging simultaneous model and data parallelism leads to accelerated convergence rates by efficiently distributing computational workloads and facilitating more frequent updates to model parameters. Notably, the impact on accuracies for both data and model parallelization remains minimal, indicating robust performance across parallelization setups. Moreover, there are observable trends in memory utilization and runtime: while memory utilization tends to decrease with model parallelization, runtime experiences a slight increase. Conversely, data parallelization yields the opposite effect, with increased memory utilization but marginally faster runtimes. However, due to constraints on time and resources, we were unable to conduct experiments for all epochs and generate corresponding graphical representations of the results.

## 9 Conclusion and future work

In conclusion, the choice between data parallelization and model parallelization hinges on various factors, including model size, dataset complexity, and available hardware resources. For smaller models and datasets, data parallelization often yields faster training times and better scalability. However, as models grow in size and complexity, model parallelization becomes more advantageous, particularly when memory constraints limit data parallelization. The effectiveness of each approach may also depend on the specific characteristics of the model and dataset, as evidenced by the scalability of Vision Transformer (ViT) models with both data and model parallelism.

Velox can be integrated better in data science workflows in the future and the efforts should focus on developing a Python API and enriching the documentation to facilitate its use. Direct integration with established machine learning frameworks, such as TensorFlow and PyTorch, would also be a strategic step to streamline data processing workflows. Additionally, performance benchmarking across diverse environments is needed to improve Velox's compatibility and usability.

Moving forward, several avenues for future work could enhance parallelization techniques and their application in deep learning tasks. Integration of Velox could optimize data management and accelerate analysis, thereby improving overall solution efficiency. Additionally, exploring advanced parallelization techniques like pipeline parallelism or hybrid parallelism could offer further performance gains. Furthermore, integrating the benchmarking framework with automated machine learning (AutoML) frameworks could automate the process of selecting the optimal parallelization strategy for a given model and dataset, streamlining the training process and enhancing overall productivity. These future endeavors

hold promise for advancing the field of distributed deep learning and optimizing training workflows.

# Bibliography


[1] (2021). Github - facebookincubator/velox: A c++ vectorized database acceleration library aimed to optimizing query engines and data processing systems. https://github.com/facebookincubator/velox. [Accessed 28-04-2024].

[2] Kasten, M. and Lee, S. (2022). Velox: Addressing challenges in data and model parallelism for deep learning. In *Proceedings of the International Conference on Machine Learning*, pages 100–110.

[3] Pedro Pedreira, Masha Basmanova, O. E. (2023). Introducing velox: An open source unified execution engine. https://engineering.fb.com/2023/03/09/open-source/velox-open-source-execution-engine/. [Accessed 28-04-2024].

[4] Tan, X. and Liu, W. (2022). Training large language models: A case study. *Journal of Artificial Intelligence Research*, 30:300–315.

[5] Younesy, A. and Smith, J. (2021). A comparison of data parallel and model parallel training in deep learning. *Journal of Machine Learning Research*, 22:1–15.

[6] Zhang, W. and Li, M. (2023). Communication-efficient training through hierarchical model parallelism. In *Proceedings of the International Conference on Learning Representations*.

[7] Zheng, L. and Wang, H. (2022). A survey of parallel training techniques for deep learning. *Neural Networks*, 45:123–135.