Comparison of ML Workloads in Velox, Tensorflow and Pytorch

Group 11 - Shrey Malvi, Misha Jarsaniya, Himali Gajare

Problem

In the domain of machine learning and data management systems, the challenge lies in developing efficient and scalable frameworks that can handle increasingly large datasets while maintaining high performance standards. Existing solutions such as TensorFlow and PyTorch offer robust capabilities for model training and inference, but they may lack in terms of unified execution and data management. This fragmentation can lead to inefficiencies, inconsistency in latency, and difficulties in resource allocation. Therefore, there is a pressing need for a unified platform that seamlessly integrates data management and machine learning tasks, ensuring efficiency, consistency, and reusability across the entire workflow. The primary objective of this project is to compare the latency of ML workloads between Velox and TensorFlow/PyTorch for an Image classification pipeline based on Convolutional Neural Networks architecture. Understanding and quantifying the latency differences between these platforms provide insights into their performance characteristics and help in making informed decisions when choosing a platform for ML tasks. Additionally, this project will also explore the efficiency gains offered by depthwise separable convolutions, considering their potential for reducing computational costs and memory footprint in resource-constrained environments.

Literature Survey

The integration of machine learning (ML) with database systems has been an active area of research, with various approaches and frameworks proposed to address the challenges of efficiently combining these two domains. One prevalent method is the utilization of end-to-end ML systems that provide a seamless interface for integrating ML models with database systems. In the paper [1], the authors described the design of the ML-to-SQL Python framework, which offers a simple API to automatically generate model tables and SQL queries for performing model inference using a pre-trained neural network model and a database connection.
While some database systems have expanded their SQL dialects to accommodate ML functionalities, other approaches have adopted a more decoupled approach. The paper [2] introduces an open-source solution called SQLFlow, which acts as a bridge between different database systems and ML engines. The authors designed an extension and created a collaborative parsing algorithm that integrates ML syntax with various SQL dialects. In contrast to traditional practices, SQLFlow shifts the paradigm by compiling SQL programs into Kubernetes-compatible workflows, enabling efficient deployment and execution across both public clouds and on-premise clusters.
Tool developers also play a crucial role in the integration of ML with database systems by creating ML engines and database platforms, primarily utilizing languages such as C++ and Go. One such tool is Velox [3], developed by Meta, which addresses the challenge of integrating data analytics and ML due to fragmented infrastructure. Velox provides a unified platform for seamless data preprocessing, offering

core features that integrate seamlessly with existing ML frameworks. Real-world case studies have highlighted its effectiveness in diverse ML pipelines.

In addition to these approaches, several research efforts have focused on developing specialized database systems and frameworks for efficient ML workloads. OmniSciDB [4] is a GPU-accelerated database system designed for interactive analytics and machine learning on large-scale datasets. BlazingSQL [5] is another GPU-accelerated SQL engine that aims to accelerate ML workloads by leveraging the computational power of GPUs. SamsaRA [6] is a distributed in-memory database system that supports efficient execution of ML workloads by exploiting the parallelism and scalability of modern hardware architectures.

Furthermore, the integration of hardware acceleration, such as GPUs and TPUs, with database systems for ML workloads has been an active area of research. The paper "GPU-Accelerated Database Systems: A Survey" [7] provides a comprehensive overview of the challenges and opportunities in leveraging hardware acceleration for database systems, including ML workloads.

To ensure a comprehensive and rigorous comparative analysis, it is essential to consider established benchmarking methodologies and performance evaluation techniques specifically designed for ML workloads in database systems. MLPerf [8] is an industry-standard benchmark suite for evaluating the performance of ML systems, providing a standardized way to measure and compare the performance of various ML workloads across different platforms and hardware configurations.

Environment and Experimental Setup

We followed a standard convolution+dense layer pipeline for image classification. The input images of MNIST are of size 28x28. We applied two convolutional layers of kernel size 7x7 with 24 filters followed by a dense layer with 32 hidden units and the output layer with 10 neurons (10 classes) with softmax activation. All hidden layers are employed with the ReLU activation function. The total number of trainable parameters are 78,962. The exact same models have been used across all three frameworks to ensure consistency and an unbiased comparison in latency. Models in TensorFlow and PyTorch are trained with Adam optimizer (default parameters) for 10 epochs and we attained 99.71% accuracy in TensorFlow and 99.61% accuracy in PyTorch for the train set. Apart from this, the pipeline with depth-wise separable convolutions contains 51,387 trainable parameters. For comparing the latency, we took the average over 10 iterations for each of the framework.

We built the Velox using a docker file on a GCP VM with Ubuntu 22.04. The experimental configuration are as follows:

Ubuntu 4-core 16GB RAM CPU

Libraries: ConnectorX, Velox, PyTorch and TensorFlow

Languages: C++ and Python

Dataset

For this project, we are using the MNIST dataset, a widely recognized benchmark in the field of machine learning. The MNIST dataset consists of 70,000 handwritten digit images divided into a training set of 60,000 examples and a test set of 10,000 examples. Each image is 28x28 pixels in grayscale. The dataset is commonly used for benchmarking machine learning algorithms in the task of digit recognition. It comprises four files: training set images (9,912,422 bytes), training set labels (28,881 bytes), test set images (1,648,877 bytes), and test set labels (4,542 bytes).

Methodology

We have proposed a standard pipeline for processing the images for the classification task. A ConvBlock consists of three layers: Convolution operation, ReLU activation, and a MaxPooling layer. The model architecture and training details are mentioned in the experimental setup section. We trained the model with the exact same number of parameters on PyTorch and TensorFlow and saved the corresponding weights and biases. These are the weights that will be used for inference in Velox, PyTorch, and TensorFlow. After this, we have defined two distinct pipelines for inference (figure 1): one utilizing Velox and the other TensorFlow/PyTorch. For Velox evaluation, we begin by reading image data, weights, and biases from txt files. We then register functions and create a plan for the machine learning workload. Evaluation entails recording a single forward pass of the workload. Conversely, for TensorFlow/PyTorch evaluation, data is stored in PostgreSQL, and the database is loaded using the Connectorx library. For ingesting the input images in a database, we flattened the 2D matrix into a 1D vector and stored it in a columnar format.

In implementing the image classification pipeline using Velox, we leverage functions from the functions.h and NNBuilder.h header files to define the workflow. The pre-trained weights are converted into arrays and saved to a text file for easy retrieval. Subsequently, in C++, inputs and weights are loaded from the text file. The pipeline execution in Velox involves a single forward pass. We also created a new workload using Torch bindings and added them to functions.h header file which incurred significant improvement in latency over C++ implementation of convolution using Eigen that is discussed in more detail in the results section.

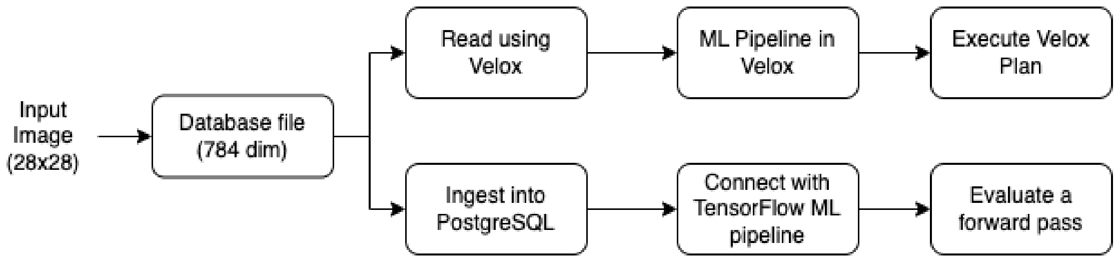

Figure 1: Pipeline for Velox and TensorFlow

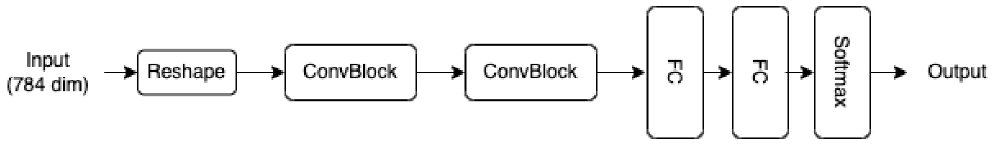

Figure 2: ML (image Classification) pipeline to be created

Alternatively, we have also compared the performance of depth-wise separable convolution operation to regular Conv2d Depth-wise separable convolutions that offer reduced computational cost and memory footprint compared to standard convolutions, leading to lower resource requirements. They improve generalization by independently filtering spatial features across input channels while preserving channel-wise information. The details of the architecture leveraging depth-wise convolutional operations are mentioned in the experimental setup section.

Results

For the defined ML workload of image classification, we compared the latency using Velox, PyTorch, and TensorFlow. For Velox, we leveraged two implementations: conventional convolution operations using the Eigen library, and convolution operations from Torch. Figure 3 shows the latency comparison for all implementations for various numbers of samples. The conventional conv implementation resulted in exponential increase in latency with an increase in number of samples due to nested for loops whereas the Torch implementation of Velox resulted in comparable results actually 30% better latency than PyTorch. TensorFlow took slightly more time than PyTorch. We also compared the data load time in all three frameworks. Velox being an in-memory database execution engine, it doesn't make sense to calculate the data load time but for the sake of comparison, we evaluated the data loading time for all three frameworks. Figure 4 represents the comparison between data load times between Velox, Pytorch and Tensorflow.

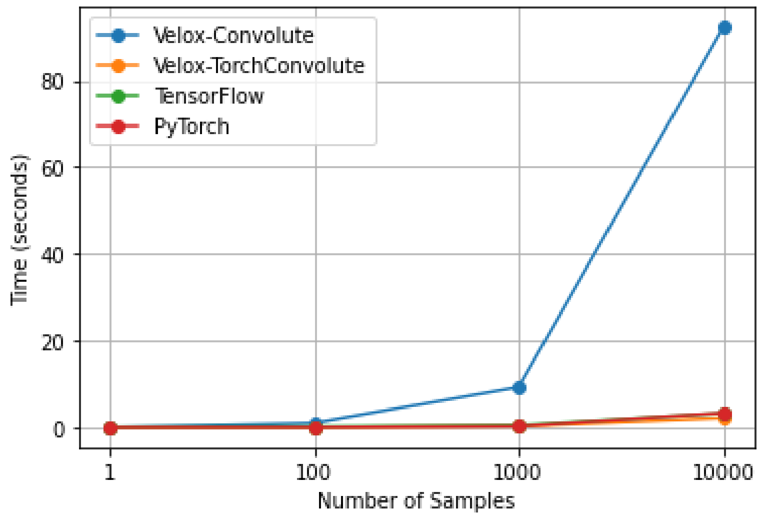

Figure 3: Latency comparison between Velox (Convolute, torchConvolute); Tensorflow, Pytorch

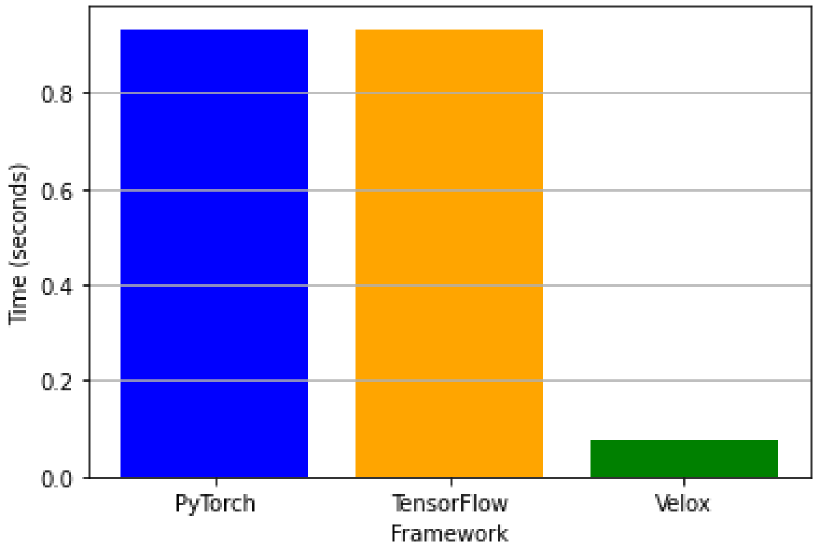

Figure 4: Data Load time comparison

Moreover, on a deeper investigation, we also monitored the latency of each component of velox i.e. data load, data inference and data transfer. Figure 5 depicts the time taken by each component.

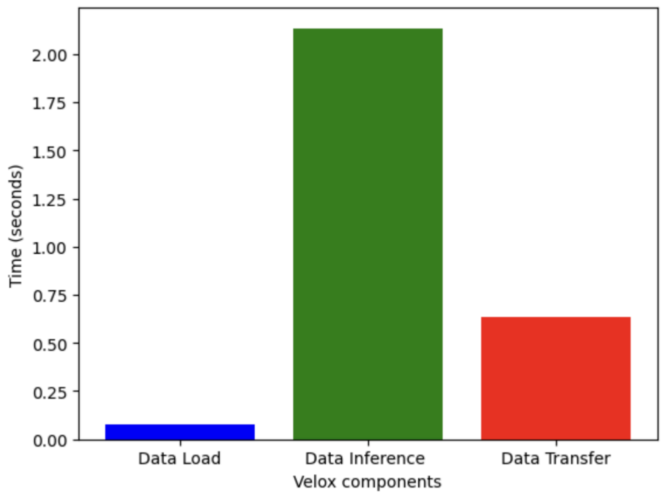

Figure 5: Time taken by each component of Velox (Data Load, Data Transfer, Data Inference)

For the second part, we replaced the vanilla convolutional operation with depth-wise separable convolution operation and trained the model which resulted in 99.35% accuracy on train set in TensorFlow and 99.12% accuracy in PyTorch. There is a 93.67% reduction in the number of parameters from the vanilla convolution operators to the depth-wise separable convolutions. Table 1 shows the inference time between vanilla convolution and depthwise convolution operations for each implementation for num samples 1, 100 and 500. The depthwise convolution operations took slightly longer than vanilla convolution because each depth-wise conv contains a pointwise operation followed by vanilla conv. These resulted in higher latency but significant reduction in the number of parameters.

No. of Samples	PyTorch	TensorFlow	TorchConvolute	Torch Depthwise Convolute
1	0.02 sec	0.15 sec	0.09 sec	0.12 sec
100	0.01 sec	0.37 sec	0.097107 sec	0.13 sec
500	0.08 sec	0.4 sec	0.17797 sec	0.2 sec

Table 1: Inference time between vanilla convolution and depthwise separable convolution

Lastly, we also experimented with the HiveConnector splits functionality of the Velox for multithreading by setting num splits = 3 and STRIP_SIZE = 100 KB but didn't witness any improvement over single split implementation.

Discussions

Velox performed better in terms of inference than TensorFlow and PyTorch. The data transfer time in Velox puts an overhead in the entire pipeline. The initial implementation of the Velox consisted of separate compute_string for each convolutional as well as dense layer. We optimized this pipeline by creating a single compute_string which performs all ML layers at once. This eliminated the data transfer required in each compute_string and resulted in a significant reduction in the overall latency for the inference. Additionally, the Torch bindings of Velox turns out to be way faster than Eigen implementation due to a combination of optimized CPU implementations, parallelization, library integration, algorithmic optimizations, and framework-specific optimizations. The depth-wise separable convolutional are slower than convolutional operations but have low memory footprint and hence are beneficial for resource-limited environments.

For multithreading, our input is float data type and for num samples = 3000, the total data size is 784x3000x4 (784 dimensional vector and 4 bytes for float). The STRIP_SIZE is utilized to denote the quantity of data written to a disk within a single stripe. Opting for a large strip size, where, for instance, all data resides within one strip, enables the reading of all data from a file simultaneously, thereby potentially minimizing data loading overhead. However, this approach sacrifices parallelism capability within UDF kernels as all data is loaded as one batch. Conversely, selecting a smaller strip size facilitates loading data in multiple batches, thus achieving parallelism within UDF kernels. The ideal STRIP_SIZE <= total data size/num_splits. Hence, we set it num_splits = 3 and STRIPE_SIZE = 100 KB which obtained the best results. For other parameters, we didn't witness any improvement in latency and hence further experiments and in-depth analysis is required to make full use of multithreading capabilities.

Conclusion

We analyzed that Velox demonstrated superior performance in terms of inference latency compared to TensorFlow and PyTorch. Leveraging Torch bindings in Velox significantly improved latency, outperforming both TensorFlow and PyTorch by approximately 30%. Furthermore, we explored the potential of depth-wise separable convolutions to reduce computational costs and memory footprint. While these operations exhibited slightly higher latency compared to regular convolutions due to additional pointwise convolution, they offered a significant reduction in the number of parameters, making them beneficial for resource-constrained environments. Additionally, we investigated multithreading capabilities in Velox using the HiveConnector splits functionality. However, we did not observe significant improvements over single split implementations, suggesting the need for further experimentation and analysis to fully leverage multithreading capabilities. In conclusion, our study highlights the efficiency gains and performance advantages of Velox over traditional ML frameworks like TensorFlow and PyTorch. By optimizing data processing pipelines, leveraging efficient convolution operations, and exploring multithreading techniques, Velox demonstrates potential for enhancing the scalability, efficiency, and performance of machine learning workloads in various applications.

Future works

Building upon the insights gained from the comparison between Velox, TensorFlow, and PyTorch, several opportunities for future work emerge, all aimed at supporting the capabilities and performance of machine learning workloads. Firstly, there is a need to explore direct data reading from databases within Velox to enable unbiased comparisons of data loading efficiency. Additionally, attention must be given to assessing the efficiency of attention layers within Velox to enhance its functionality further. Integrating transformer architectures into Velox stands as another crucial step to broaden its capabilities and accommodate advanced machine learning tasks. Furthermore, investigating multithreaded techniques for parameter optimization in Velox could significantly accelerate model training and enhance overall performance. By addressing these areas of future work, we aspire to refine and expand the Velox framework, ultimately leading to improved efficiency, scalability, and performance in various machine learning applications.

References

[1] Kläbe, S., Hagedorn, S., & Sattler, K. U. (2023). Exploration of Approaches for In-Database ML. In Proceedings of the 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-March 31.
[2] Wang, Y., Yang, Y., Zhu, W., Wu, Y., Yan, X., Liu, Y., ... & Tang, Y. (2020). SQLFlow: A Bridge between SQL and Machine Learning.
[3] Pedreira, P., Erling, O., Basmanova, M., Wilfong, K., Sakka, L., Pai, K., ... & Chattopadhyay, B. (2022). Velox: Meta's unified execution engine. Proceedings of the VLDB Endowment, 15(12), 3372-3384.
[4]渡世, L., Mostak, T., Ain, R., Sundararaman, R., Nishtala, R., Papakonstantinou, Y., ... & Marathe, A. P. (2019). OmniSciDB: A Semi-Structured Path-Based Scientific Database for Interactive Extreme Data Analytics. Proceedings of the VLDB Endowment, 12(12), 1903-1906.
[5] Ramirez, A., Kiefer, J., Baudel, T., Pauloski, G., & Childs, H. (2020). Massive Data Acceleration With BlazingSQL. In GPU Technology Conference.
[6] Elgohary, A., Boehm, M., Hadjis, P. J., Rahman, M. Z., Lin, P. Á., El Nei, N. G., ... & Freire, J. (2020). Compressed Linear Algebra for Large-Scale Machine Learning. Proceedings of the VLDB Endowment, 14(5), 913-925.
[7] Shi, L., Malik, A., & Weng, J. T. (2021). GPU-Accelerated Database Systems: A Survey. IEEE Transactions on Knowledge and Data Engineering.
[8] Mattson, P., Cheng, C., Coleman, C., Diamos, G., Micikevicius, P., Patterson, D., ... & Wu, Y. (2020). MLPerf: An industry standard benchmark suite for machine learning performance. IEEE Micro, 40(2), 8-16.