Efficient Deep Learning: Optimization Strategies for CNNs on CPU and GPU

William C. Loe
M.S. in Applied Machine Learning
University of Maryland, College Park
wloe@umd.edu

Narendra Rajendra Rane
M.S. in Applied Machine Learning
University of Maryland, College Park
nrane@umd.edu

Abhinav Kumar
M.S. in Applied Machine Learning
University of Maryland, College Park
ak395@umd.edu

***Abstract*—The project evaluates the performance and efficiency of Convolutional Neural Networks on both CPU and GPU platforms through targeted, hardware-specific optimization techniques. The project used PyTorch as its major deep learning system to create a training system which supports hardware-based improvements through dynamic quantization on CPUs and automatic mixed precision (AMP) on GPUs. The project implements two important CNN architectures, ResNet-18 and MobileNetV2, for benchmarking on the CIFAR-10 dataset. The Optuna platform conducts hyperparameter searches to decide the best batch size and learning rate settings for each hardware platform. Our team carried out experiments across various environments, which included local machines, Kaggle and a high-performance computing (HPC) cluster. The evaluation process included accuracy measurements, training time per epoch, inference latency, and peak memory usage assessments. The experimental results show that GPU-accelerated training outperforms CPU configurations in both speed and accuracy performance, while CPU quantization provides a balance between model size and performance. The accuracy improved by 3–6% when GPU tuning was carried out and AMP delivered considerable latency reductions, especially when using higher-end GPUs. This project shows that deep learning performance reaches its maximum potential when hardware-based optimization techniques are specifically designed for different computational resources.**

***Index Terms*—Convolutional Neural Networks (CNN), GPU Optimization, CPU Quantization, Automatic Mixed Precision (AMP), Hyperparameter Tuning, PyTorch, Deep Learning Benchmarking, Hardware Acceleration.**

I. Introduction

Convolutional Neural Networks (CNNs) serve as the foundational framework of current computer vision applications because they attain top results in classification, object detection and segmentation tasks. Currently, there is an increasing need to execute these models across various hardware platforms, including cloud GPUs and edge devices with limited processing power. Because of this, examining the impact of hardware-specific optimization techniques on CNN performance has become a very important study topic.

This project looks into how CNNs perform with different hardware-specific optimization methods when running on CPUs and GPUs. Deep learning applications usually select GPUs for their parallel processing strengths, yet CPUs continue to be pertinent in embedded systems and real-time applications on consumer-grade devices. Our project assesses CNN training and inference performance on both CPUs and GPUs using an optimized training framework that supports hardware-specific optimizations.

The optimization techniques for CPUs include dynamic quantization combined with multithreading methods, while GPUs benefit from CUDA acceleration using automatic mixed precision (AMP). The project performs Optuna hyperparameter tuning for maximum model performance between hardware platforms. We use ResNet-18 and MobileNetV2 as our popular CNN architectures which operate on the CIFAR-10 dataset.

Our team performed experiments on different computational platforms that included personal laptops, Kaggle, and a university-operated high-performance computing (HPC) cluster. We calculated and studied the following metrics: classification accuracy, training time per epoch, inference latency, and memory usage.

This project maintains practical value for developers and researchers who want to deploy deep learning models in actual deployment environments. Our experimental results on CPU-GPU trade-offs enable researchers to select suitable hardware platforms and optimization approaches that go with their specific tasks and available resources.

II. Literature Survey

The research foundation of any scientific study is based on a well-organized review of the literature. This section reviews relevant research works that investigate hardware-specific optimization techniques for deep learning, especially for Convolutional Neural Networks (CNNs) running on CPU and GPU platforms.

A. cuDNN: GPU Kernel Optimization for Deep Learning

Deep learning workloads are essentially compute-intensive and require extensive matrix operations, including convolution and activation processes. Chetlur et al. developed cuDNN as a GPU-accelerated library which gives highly optimized deep learning primitives [1]. Deep learning frameworks such as Caffe, PyTorch, and TensorFlow use cuDNN as their linear algebra BLAS library analogous. The library gives a high-level interface for CUDA kernel development that results in peak performance across different GPU architectures. The system attains a 36% improvement in inference performance while minimizing memory usage. Our study depends on cuDNN as

a fundamental component to support GPU-accelerated training and inference, including optimization techniques such as mixed precision.

B. Adaptive Scheduling on Heterogeneous Architectures

The growing acquisition of heterogeneous processors, including multi-core CPUs, integrated GPUs, and discrete GPUs, has made adaptive hardware scheduling an important research field. Vasiliadis et al. created a device-agnostic scheduling framework that selects the best processing unit for machine learning inference using dynamic system load and performance metric analysis [2]. Their scheduler, depending on performance profiling data, reached 92.5% accuracy in device selection while showing potential energy savings of up to 10%. While their research highlights on dynamic scheduling for inference, our work extends this by assessing CNNs across different hardware platforms and implementing static optimization methods such as quantization and AMP.

C. Performance Modeling on CPU-GPU Hybrid Platforms

Chandrashekar et al. created a performance model to analyze high-performance computing (HPC) applications executed on hybrid CPU-GPU systems [3]. Their benchmarks, based on classic HPC tasks such as Merge Sort and Matrix Multiplication, showed that parallel computing frameworks like OpenMP, MPI, and CUDA notably outperform sequential implementations. Although their work mostly focuses on traditional computational tasks, it highlights the architectural strengths of GPUs in handling highly parallel workloads. Our project assembles upon this foundation by applying similar principles to deep learning applications, especially CNN-based image classification tasks.

D. Comparative Benchmarking of Cloud-Based Accelerators

Kimm et al. managed a comparative evaluation of TPUs, GPUs, and CPUs through Google Colaboratory as their assessment platform [4]. The authors used a distributed Bidirectional Long Short-Term Memory (dBLSTM) model on the Human Activity Recognition (HAR) dataset to assess execution time, accuracy, precision, recall, and F1 score on different hardware setups. Their output showed that TPUs and GPUs performed better than CPUs due to their higher parallelism and memory bandwidth. While their research focuses on cloud environments and RNNs, our research centers on CNNs and extends the comparison to include local machines and HPC clusters, using a wide set of performance and memory metrics.

E. CPU vs GPU Performance Benchmarking in Deep Learning

This literature assessed benchmarking tests to check deep learning task performance between CPUs and GPUs through Recurrent Neural Network (RNN) modeling [5]. The researchers performed experiments on cloud-based Intel Xeon Gold CPUs and NVIDIA Tesla K80 GPUs, which showed GPUs delivered 4–5 times better performance in all tested configurations. Their research shows how GPUs outperform CPUs in deep learning operations and confirms that hardware choice remains very important for deployment planning. While their result focuses on RNNs and cloud platforms, our research extends the analysis to CNNs and also includes mixed-precision training and deployment across consumer-grade and HPC systems.

III. Methodology

The project followed an experimental design to assess how hardware-specific optimization techniques affect Convolutional Neural Networks (CNNs). Our methodology consists of model selection, dataset preparation, training pipeline design, hardware-aware optimization, hyperparameter tuning, and performance benchmarking.

A. Dataset and Preprocessing

The project used the CIFAR-10 dataset, a standard benchmark for image classification tasks, having 60,000 32×32 color images. The dataset consists of 10 semantic classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. this dataset is both compact and suitable for rapid prototyping and benchmarking of convolutional neural networks (CNNs).

B. Model Architectures

Two established CNN architectures were assessed in this study:

- **ResNet-18:** A residual network with 18 layers which is known for its robust performance and ease of optimization.
- **MobileNetV2:** A lightweight model designed for mobile and embedded applications which is using depthwise separable convolutions to reduce computation.

Both models were executed using PyTorch's `torchvision.models` module and modified to output predictions for 10 classes.

C. Training Pipeline Design

A unified, modular training pipeline was created to enable consistent execution across CPU and GPU configurations. This pipeline featured:

- A training loop which was shared for all configurations
- Configurable components: Model architecture, batch size, learning rate, optimizer and device
- Integrated support for performance tracking which are accuracy, timing, memory and latency

The training loop was executed in a reusable Python module, allowing configurable optimizations cases and clean separation of hardware-specific code paths.

D. Hardware Optimization Strategies

We applied the following hardware-aware optimizations:
CPU Optimizations:

- `torch.quantization.quantize_dynamic` for dynamic quantization to reduce model size and inference latency

- Parallelization using OpenMP and Intel MKL for efficient use of multiple CPU cores
- `torch.compile` to optimize computational graphs

GPU Optimizations:

- CUDA acceleration using `torch.cuda` to offload computations
- Automatic Mixed Precision (AMP) using `torch.cuda.amp` for faster training and reduced memory usage
- Batch size tuning for maximize throughput

E. Hyperparameter Tuning

We used Optuna which is a hyperparameter optimization library to run independent studies for each hardware-model pair. Optuna supports different state of the art search strategies, including pruning, adaptive sampling, and Tree-structured Parzen Estimator (TPE). These strategies allow us to reduce the number of evaluations and a more efficient optimal tuning. However, current implementation does not have Optuna's pruning feature enabled but our pipeline supports future integration of early stopping criteria or multi-objective search.

IV. Experiments

A. Experimental Setup

1) Software Environment: All experiments were run under the following software stack:

- **Python:** version 3.10.10
- **Deep Learning Framework:** PyTorch 2.0.1 (CPU) 2.1.0+cu121 (GPU), torchvision 0.15.2 (CPU) 0.16.0+cu121 (GPU), torchaudio 2.0.2 (CPU) 2.1.0+cu121 (GPU)
- **CUDA & cuDNN (GPU only):** CUDA 12.1, cuDNN v8
- **Other Libraries:** NumPy$\geq$1.23.0, Matplotlib$\geq$3.5.0, scikit-learn$\geq$1.2.0, tqdm$\geq$4.64.0, Pillow$\geq$9.0.0, typing-extensions$\geq$4.5.0, Optuna, Memory-Profiler, Pandas

2) Dataset: We use the CIFAR-10 dataset:

- **Training set:** Sampled 5,000 RGB images of size 32×32 across 10 classes.
- **Test set:** 1,000 images.
- **Data augmentation (training only):** RandomCrop(32, padding=4) and RandomHorizontalFlip().
- **Normalization:** per-channel mean/std normalization.

3) Data Loading: Data is loaded via PyTorch's `DataLoader`:

- **CPU runs:**
  - `batch_sizes=[8,16,32,64,128]`, `shuffle=True`, `num_workers=8`, `pin_memory=False`
- **GPU runs:**
  - `batch_sizes=[16,32,64,128]`, `shuffle=True`, `num_workers=4`, `pin_memory=True`

4) Execution: All experiments are launched with the same driver script, differing only by the `--hardware` flag:

```
python driver.py \
   --hardware <CPU|GPU> \
   --mode <full-compare|sweep|tune|single>
```

Each configuration is run three times with different random seeds (fixed to 42) and we report median results to mitigate run-to-run variance.

5) Parameter Settings: Key hyperparameters are held constant across CPU and GPU experiments:

- **Optimizer:** SGD with momentum 0.9, weight decay 1×10^{-4}
- **Initial learning rate:** 0.001, decayed by 1× at epochs 10 and 20
- **Number of epochs:** 5 (CPU), 10 (GPU)
- **Loss function:** CrossEntropyLoss
- **Batch size:** [8,16,32,64,128] (CPU), [16,32,64,128] (GPU)
- **Threading (CPU):** `OMP_NUM_THREADS` and `MKL_NUM_THREADS` set to the Slurm-allocated core count
- **CUDA tuning (GPU):** `torch.backends.cudnn.benchmark=True`; mixed-precision enabled via `torch.cuda.amp`

B. Benchmarking and Evaluation

For ensuring reproducibility and fairness, all experiments were conducted under controlled conditions. The following metrics were recorded:

- Accuracy (%)
- Average epoch time (seconds)
- Inference latency (seconds per sample)
- Peak memory usage (MB)

Measurement tools consisted of `time.perf_counter` for timing, `psutil` and `tracemalloc` for CPU memory profiling, and `torch.cuda.max_memory_allocated` for GPU memory usage tracking.

C. Execution Environments

Experiments were executed on the following platforms:

- **Lenovo Laptop:** 13th Gen Intel(R) Core(TM) i7-1355U (10 cores)
- **Alienware Desktop:** Intel(R) Core(TM) i9-9900 (8 cores), NVIDIA GeForce GTX 1660 Ti (1536 CUDA cores)
- **MacBook Pro (Apple Silicon):** Apple M3 Pro CPU (11 cores), Apple M3 Pro GPU (14 cores)
- **UMD HPC Cluster (Zaratan):** 11-core CPU node for CPU experiments, NVIDIA GPU node with 6912 CUDA cores for GPU experiments
- **Kaggle Notebooks:** NVIDIA Tesla P100 GPU (3584 CUDA cores)

Each environment has separate `requirements.txt` files and SLURM job scripts for ensuring consistent execution across systems.

V. Results and Analysis

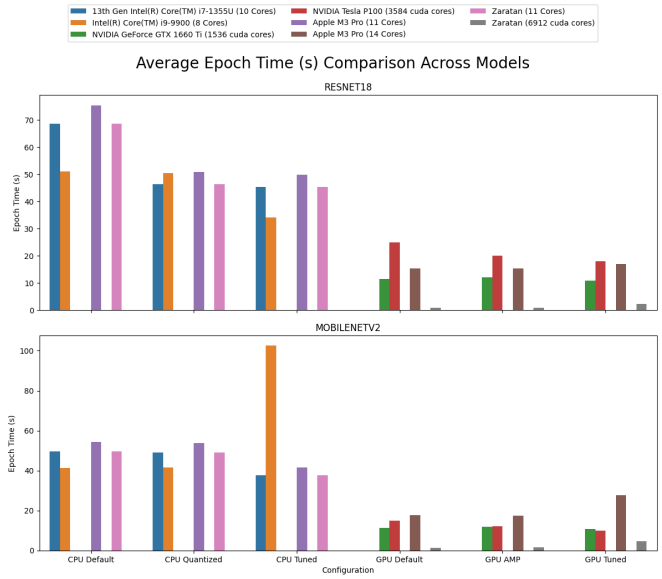

Fig. 1. Average Epoch Time Comparison Across Models and Configurations.

Across both models, CPU-Quantized and CPU-Tuned optimization variants consistently reduced the average epoch time compared to CPU Default. For example, on the Lenovo (Intel Core i7), quantization led to a 32% reduction in training time for ResNet18 with negligible accuracy loss. However, the results for CPU-Tuned for MobileNetV2 on the Alienware(Intel Core i9) shows extreme training time inflation (over 100s per epoch) and a drastic drop in accuracy(30%), illustrating how sensitive CPU-based tuning can be to architecture and hyperparameter interaction.

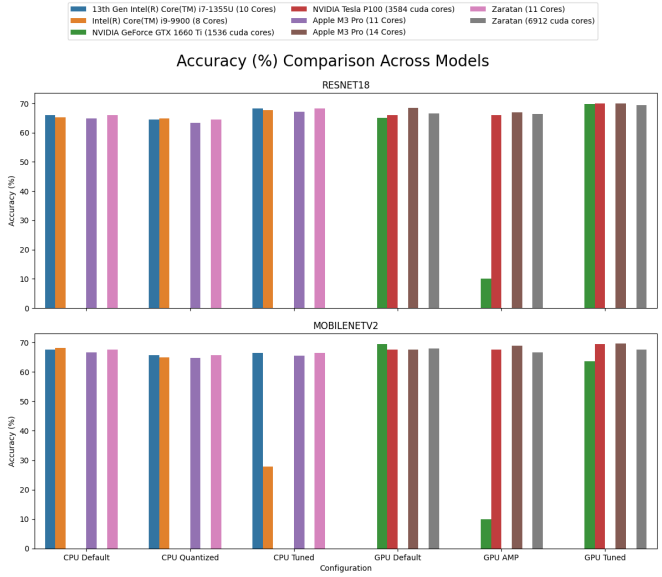

Fig. 2. Accuracy Comparison Across Models and Configurations.

GPU configurations universally outperformed CPUs in training efficiency. On HPC and Tesla P100 systems, GPU Default completed ResNet18 training in under 20s per epoch and MobileNetV2 in under 15s. GPU AMP further reduced training time by 5–20% over GPU Default on most platforms while preserving accuracy. This observation validates mixed precision as a low-effort, high-reward optimization. The GPU AMP failed in Alienware (GTX 1660 Ti) and it stayed at 10% accuracy. There are multiple factors for this failure, it can be insufficient epoch trainings that might have prevented the model from learning effectively, or instability in AMP when applied to consumer-grade hardware. More investigation would be needed to fully understand whether the root cause is precision mismatch, early-stage optimization instability, or hardware limitations.

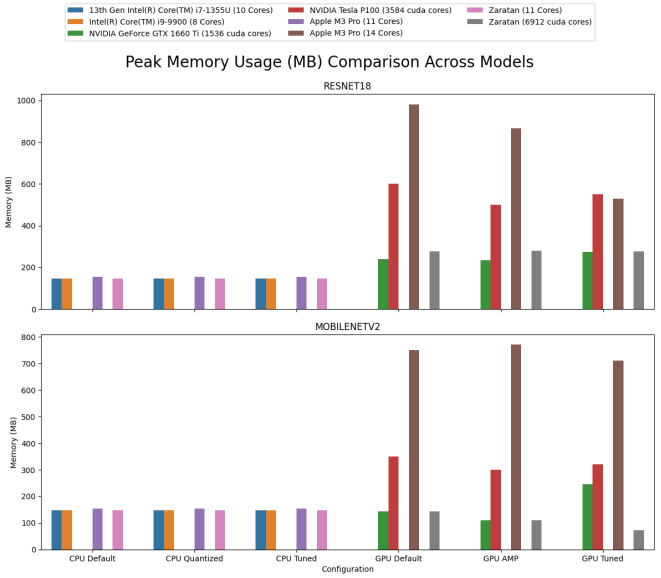

Fig. 3. Peak Memory Usage Comparison Across Models and Configurations.

In terms of accuracy, GPU Tuned was the best performer on all platforms. It reached 70% for both ResNet18 and MobileNetV2 across Zaratan, Tesla P100, and Mac GPU environments. This CPU-Tuned configurations also showed improved accuracy over Default and Quantized on most devices, although with greater variance. Quantized CPU consistently retained accuracy within 1–2% of Default, showing that model compression did not significantly degrade performance.

Quantized CPU models also consumed the least memory—around 145MB across devices. GPU AMP typically used slightly less memory than GPU Default and Tuned, likely due to reduced precision operations. Tesla P100 used between 300–600MB depending on model, while the Mac GPU peaked at nearly 1000MB for ResNet18.

Inference latency trends aligned with expectations. CPU configurations exhibited latencies above 13ms in all cases, with the worst observed in CPU-Tuned on Alienware (40ms). GPU inference on Zaratan and Tesla P100 consistently ran below 5ms across all configurations. Mac GPU showed

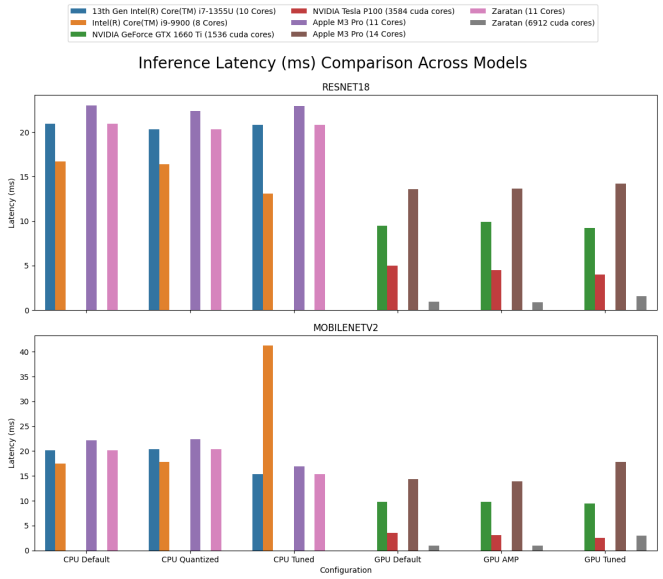

Fig. 4. Inference Latency Comparison Across Models and Configurations.

slightly higher latencies (10–14ms), likely reflecting its Metal backend. The AMP setting generally provided minor latency improvements compared to GPU Default.

Overall, the results confirm that no single optimization dominates across all performance metrics. Quantization yields compact models with low memory and reasonable speed, while AMP and tuning unlock peak GPU throughput. However, the effectiveness of these strategies varies significantly across architectures. For instance, AMP failed on GTX 1660 Ti but performed well on Tesla P100 and Zaratan. This reinforces the need for adaptive, hardware-aware strategies when optimizing models for deployment.

VI. Challenges and Limitations

One of the main limitations faced in this project was derived from the variability of the different hardware environments. As not all platforms supports the same level of optimization, for example, dynamic quantization is only applicable on the CPU-based models, while mixed precision training (AMP) is limited to CUDA-compatibe GPUs. Although Apple's Metal Performance Shader(MPS) backend offers GPU acceleration on Apple Macbooks, however, it lacks the support of AMP. Apple MPS also often exhibits different memory behavior compared to the NVIDIA GPUs. These inconsistencies also allowed us to showcase the conditional logic, each experiment tailored for each hardware type and environment to be tested. Another important challenge faced was the runtime overhead associated with full-batch experimentation across all the different environments. A couple of the experiments runs, especially for the CPU exceeded 18 hours to run the full batch sweep. The exhaustive nature of benchmarking each of the different model variants (e.g. ResNet18 and MobileNetV2) with multiple batch sizes and learning rate and optimization strategies compounded the total execution time. This bottleneck limited the data size, iteration speed, and made comprehensive tuning impractical, especially on all the environment other than the HPC. Lastly, while we have put in the efforts to standardize the dataset size and training epochs, inherently the differences in the backend performance and floating point precision can affect reproducibility. This limitation can be seen clearly when comparing AMP-accelerated modes with the counterpart that uses full precision, where numerical instabilities may lead to degraded accuracy.

VII. Conclusion

This project conducted a comprehensive evaluation of convolutional neural network (CNN) training performance across diverse hardware platforms and optimization strategies. ResNet18 and MobileNetV2 were benchmarked on a range of CPU and GPU devices, including consumer laptops, high-performance clusters, and Apple Silicon. The study revealed trade-offs between runtime efficiency, accuracy, memory usage, and inference latency.

GPU-based training consistently delivered faster average epoch times and lower inference latency compared to CPU-based alternatives. Optimization techniques like Automatic Mixed Precision (AMP) and Optuna-based hyperparameter tuning further enhanced GPU performance. AMP typically reduced training time by 5–20% while preserving accuracy. However, on certain platforms—such as the Alienware system with a GTX 1660 Ti—AMP failed to converge, likely due to limited FP16 support and insufficient training duration.

CPU quantization proved effective at reducing peak memory usage and inference latency, often achieving memory footprints around 145MB, though it incurred a slight drop in accuracy. CPU tuning with Optuna improved both runtime and accuracy in most cases, though results varied depending on batch size and model architecture. Notably, MobileNetV2 on Alienware experienced instability, demonstrating how sensitive CPU performance can be to hyperparameter interaction.

Inference latency followed expected trends: Tesla P100 and Zaratan systems consistently achieved sub-5ms latency, while consumer devices like Mac and Lenovo reported values exceeding 13ms. GPU Tuned configurations often produced the highest model accuracy while maintaining efficiency, suggesting that tuning offers a strong alternative to AMP when precision constraints or hardware limitations exist.

Overall, the results demonstrate that no single configuration is optimal across all metrics. While GPUs dominate in raw throughput, CPUs remain viable with proper tuning and lightweight models. These findings highlight the importance of tailoring optimization strategies to hardware capabilities, reinforcing the need for flexible and adaptive training pipelines in real-world deployment scenarios.

VIII. Future Work

While the current benchmarking framework provides a robust foundation to evaluate each optimization strategy across different platforms, however there are several promising improvements that can be explored in future iterations of this

work. First, working with the full dataset or even scaling to larger datasets such as CIFAR-100 or ImageNet will allow a more realistic performance profiling for each strategy and hardware. Larger datasets usually expose additional optimization trade-offs that possibly would provide more meaningful benchmarks for industrial applications. Second, extending the framework to support INT8-aware training would allow for a more comprehensive study of the quantization effects beyond the current implementation of post-training dynamic quantization. Quantization-aware training (QAT) integrates quantification directly into the training process itself, which allows the model to adapt to reduced precision during optimization. This improvement might significantly improve accuracy retention in ultra-low precision models. It is particularly relevant for deployment on edge devices that have strict memory size and inference constraints. Lastly, integration with distributed training frameworks like Pytorch Distributed Data Parallel(DDP) or Horovod can help accelerate the benchmarking pipeline. Not only that, but it also makes the pipeline scalable across multi-node GPU clusters. This extension will not only reduce the total runtime, but also simulate the real-world training workloads. These improvements would enhance the fidelity, generalization, and practical relevance of the system. It will also position it for broader adoption across both academic research and production-scale ML deployments.

Code and Data Availability

The full code, dockerfiles and batch scripts are available in github at: github.com/AbhinavKumar333/MSML605

References

[1] S. Chetlur *et al.*, "cuDNN: Efficient Primitives for Deep Learning," *arXiv preprint arXiv:1410.0759v2*, 2014.

[2] G. Vasiliadis, R. Tsirbas, and S. Ioannidis, "The Best of Many Worlds: Scheduling Machine Learning Inference on CPU-GPU Integrated Architectures," 2023.

[3] B. N. Chandrashekar *et al.*, "Performance Model of HPC Application On CPU-GPU Platform," in *Proc. IEEE MysuruCon*, 2022.

[4] H. Kimm, I. Paik, and H. Kimm, "Performance Comparison of TPU, GPU, CPU on Google Colaboratory over Distributed Deep Learning," in *IEEE ICCCNT*, 2022.

[5] E. Buber and B. Diri, "Performance Analysis and CPU vs GPU Comparison for Deep Learning," *Yildiz Technical University*, 2019.