CPU vs GPU optimized CNN algorithms

*

1st Javad Baghirov
UMD CMNS
UMD
College Park, US
jbaghiro@umd.edu

***Abstract*—This project implements and evaluates CNN inference on the MNIST dataset using both CPU and GPU platforms. On the GPU side, we develop two self-contained CUDA programs—in NCHW (channels-first) and NHWC (channels-last) layouts—that load pre-trained CNN weights, perform a single convolution-ReLU-pool layer followed by a fully-connected classification layer, and report throughput (images/second) and accuracy. For the CPU, we build an optimized PyTorch-based inference pipeline: weights are loaded from binary files, a TorchScript script enables JIT compilation, and inference is parallelized across all available cores. Benchmarks on 10,000 MNIST test images show that GPU inference in NCHW layout achieves up to an order-of-magnitude higher throughput than the optimized CPU implementation. These results underscore the importance of hardware-aware layout choices, thread-level parallelism, and JIT optimizations for CNN inference.**

I. Introduction

Convolutional Neural Networks (CNNs) have become the standard for image classification tasks thanks to their high accuracy. However, achieving low-latency, high-throughput inference remains a challenge, especially when deploying on resource-constrained devices or in cost-sensitive cloud environments. While GPUs offer massive parallelism, CPUs may be used in certain cases where the GPU costs outweigh their benefits. Moreover, the choice of tensor layout—channels-first (NCHW) versus channels-last (NHWC)—can critically impact memory-access patterns and thus performance on specifically the GPU architecture.

This project addresses two core questions:

How does inference throughput and latency for a simple CNN algorithm compare between CPU and GPU implementations?

What is the performance impact of NCHW versus NHWC memory layouts on GPU inference?

Our objectives are to (a) implement end-to-end inference pipelines on both platforms, (b) benchmark throughput, accuracy and memory under similar conditions, and (c) analyze the role of memory layout and parallelization strategies. By doing so, we aim to provide concrete guidelines for people choosing between CPU and GPU deployments and selecting appropriate data formats for CNN algorithms.

II. Literature Survey

Prior work has demonstrated the critical role of low-level optimization techniques and data layouts in CNN inference performance. We summarize key findings in those fields below:

A. Memory Layouts: NCHW vs. NHWC

The memory layout (ordering of dimensions in linear memory) significantly impacts performance by affecting data access patterns and cache behavior [1]. Two common layouts are NCHW (batch, channels, height, width) and NHWC (batch, height, width, channels).

PyTorch and cuDNN use NCHW (channels-first), whereas TensorFlow uses NHWC (channels-last) by default [2]. The choice influences memory access patterns and cache behavior. In NCHW format, all elements of each channel are stored contiguously, whereas NHWC stores elements of all channels for each spatial location contiguously.

For CPU platforms, research shows that a channels-last layout (NHWC) yields much better performance than plain NCHW for CNNs. However, specialized blocked formats like nChw16c which pads channels to 16 for SIMD operations performs best. [3] [4]

B. CPU vs. GPU Performance for CNNs

GPUs are specialized for parallel numerical computing: a single GPU often contains thousands of cores that can execute many operations at the same time, whereas a typical CPU might have 4–16 cores optimized for sequential task performance. This architectural difference means GPUs can achieve 10× to 1000× more operations per second than a CPU. [5] [6]

Training Performance: GPUs substantially outperform CPUs for training deep CNN models. Early work by Raina et al. showed that training deep belief networks on a single GPU ran up to 70× faster than on a multicore CPU server, making large-scale unsupervised learning feasible in hours rather than days . Another study by Bahrampour et al showed that across LeNet, AlexNet, autoencoder, and LSTM benchmarks, GPU-based training and inference ran approximately 11×, 25–30×, 7–8×, and 20× faster respectively than optimized multi-threaded CPU versions. Moreover, Torch led CPU performance in all cases, Theano often topped GPU speed for smaller models, and Neon was most competitive on large convolutional networks. [7] [8].

Framework optimizations: Deep learning frameworks use optimized low-level libraries to maximize CPU performance. PyTorch both integrates Intel's oneDNN library for x86 CPUs. These libraries use vectorized instructions, cache-blocking and threading to accelerate key matrix operations. In this paper by Li et al., the authors introduce the oneDNN Graph Compiler, a hybrid tensor compiler that blends compiler-driven graph optimizations with expert-tuned microkernel templates to generate high-performance code for DNN computation graphs. Their experiments demonstrate 2–6× speedups over state-of-the-art compilers like TVM on key subgraphs (MLP and multi-head attention) and up to a 20% end-to-end throughput improvement on full BERT and DLRM inference. [9]

Inference Performance: For inference, the CPU vs. GPU decision depends on the scenario. GPUs excel at high-throughput inference and heavy models. CPUs can be competitive for low-batch or real-time inference when using optimized libraries.

C. Just-In-Time (JIT) Compilation

JIT compilation refers to runtime or ahead-of-time code generation that optimizes model execution. In CNNs, JIT compilers like PyTorch's TorchScript and TensorFlow transform high-level model definitions into efficient low-level code, performing graph-level optimizations.

Vasilache et al. created a polyhedral JIT compiler that takes high-level tensor expressions and automatically generates fused, size-specialized CUDA kernels. Their system achieves up to 4× speedup over NVIDIA libraries (e.g. cuDNN) [10]

The primary benefits of JIT compilation are reducing memory I/O, fewer kernel launches, better cache locality, and targeting of specialized hardware units.

III. Methodology

A. Hardware and Software Environment

The project was done on an AWS EC2 instance (g4dn.xlarge). We both ssh'd into it via the root user and used the github repository as the middle man to transfer code.

GPU: NVIDIA T4 (16 GiB GDDR6)

CPU: 4 vCPUs, which corresponds to 2 physical CPU cores

Dataset: MNIST test set (10,000 × 28×28 grayscale images).

The weights and parameters of the CNN were pretrained using PyTorch. For each implementation, weights were exported in the memory layout matching the target system: NCHW (channels-first) and NHWC (channels-last). The pretrained weights were then loaded into the respective CPU and GPU inference pipelines to ensure consistent evaluation and layout alignment. We made sure the networks were correctly implemented by checking the accuracy which was always in 96% range.

Profiling and benchmarking were automated using a shell script (detailed_profiling_fixed.sh) that:

- Runs both GPU (NCHW and NHWC) and CPU (PyTorch, optimized PyTorch) implementations.
- Collects throughput and timing for each implementation using /usr/bin/time and Python time.time().
- Profiles GPU kernel activity and memory usage with NVIDIA nvprof.
- Monitors memory usage for all implementations by polling /proc/<pid>/status and saving results to CSV.
- Aggregates all results and logs in the profiling_results/ directory.
- Analyzes output and generates summary plots and tables with a Python script (analyze_results.py).

IV. Experiments

A. GPU Implementation

We develop two standalone programs, `gpu_mnist_nchw.cu` and `gpu_mnist.cu`. Both follow the same flow:

CUDA Kernels:

- Convolution (3×3, padding=1): A single pass over each output pixel, accumulating bias plus weighted sums from the 3×3 window.
- ReLU: Element-wise activation.
- MaxPool (2×2, stride=2): Each thread computes one pooled output by scanning a 2×2 window. Reduces each 28×28 feature map to 14×14
- Fully Connected layer

Benchmarking:

- Run 10 forward passes end-to-end, measure average elapsed GPU time, compute throughput = 10,000 images / avg sec.

B. CPU Implementation

We use PyTorch (with MKL and OpenMP) and TorchScript JIT to accelerate the same TinyCNN:

JIT Compilation:

- Script the module with `torch.jit.script`, enabling graph optimizations and fusion.

Parallel Data Loading & Inference:

- Use DataLoader with `pin_memory=True` and `num_workers=min(4, n_cpu)` for prefetching.
- Set `OMP_NUM_THREADS` and `MKL_NUM_THREADS` to total logical cores.
- Measure inference time and throughput across the MNIST test set while recording accuracy.

V. Results

A. Evaluation Metrics

Throughput: images processed per second.

Memory Usage: amount of MB used during inference.

Accuracy: classification accuracy on the 10,000-image test set.

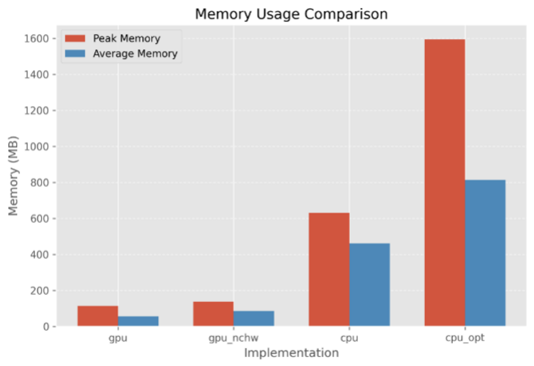

Fig. 1. Memory usage comparison across all four implementation approaches. Both CUDA GPU implementations (NCHW and NHWC) demonstrate excellent memory efficiency with peak usage under 150 MB, while CPU implementations require substantially more memory. The Optimized CPU (JIT) implementation consumes approximately 1,600 MB at peak compared to around 630 MB for the basic CPU (PyTorch) implementation.

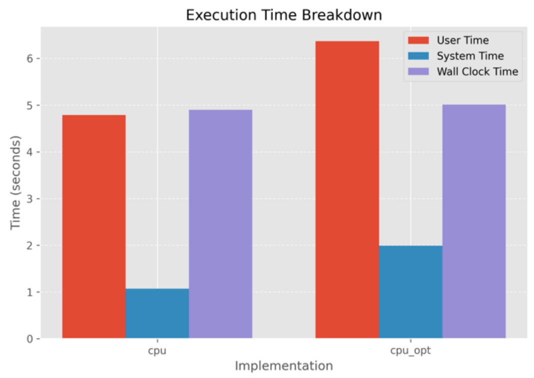

Fig. 2. Execution time breakdown showing user time, system time, and wall clock time across implementations. Both CPU implementations show similar user time, but the Optimized CPU (JIT) version demonstrates higher system time due to compilation overhead. CUDA GPU implementations offer significantly lower overall execution times.

B. Performance Analysis

Our experimental results reveal significant performance differences across implementations while maintaining consistent accuracy. The comprehensive analysis covers throughput, accuracy, memory usage, and execution time patterns.

In particular, the NCHW (channels-first) layout enabled better memory coalescing since all 32 threads in a GPU warp operate on the same channel group, requiring a single memory transaction per channel group. By contrast, the NHWC (channels-last) layout maps each thread to a distinct output channel, resulting in multiple memory trips and less efficient access. On the CPU side, the optimized implementation spawns multiple background workers for parallel data loading and employs TorchScript JIT compilation, which accelerates inference at the cost of increased memory and CPU usage. The baseline CPU implementation uses a single process with lower memory overhead. The execution time breakdown reveals higher total user time for the optimized CPU (due to summing CPU time across all threads) but only a marginal increase in wall clock time thanks to parallelism. Additionally, the NCHW format incurs slightly higher memory usage due to channel dimension padding (e.g., 32-byte alignment). This is because the Channel layer is 2nd so every layer below the channel layer gets padded ending up with slightly higher memory usage.

1) Throughput Performance: Our performance measurements show dramatic throughput differences between implementations:

Implementation	Throughput (img/s)	Accuracy (%)
CUDA GPU (NCHW)	521,462	96.3
CUDA GPU (NHWC)	253,271	96.1
CPU (PyTorch)	6,997	96.2
Optimized CPU (JIT)	10,000	96.3

TABLE I
THROUGHPUT AND ACCURACY SUMMARY FOR TINYCNN ON MNIST ACROSS FOUR IMPLEMENTATION APPROACHES.

The CUDA GPU implementation with NCHW layout achieved the highest throughput at approximately 521,462 images per second, more than doubling the performance of the NHWC layout implementation (253,271 img/s). This dramatic difference demonstrates the significant impact of memory layout optimization on our particular GPU architecture (NVIDIA T4).

Both CPU implementations showed substantially lower throughput. The optimized CPU model with multithreading and JIT compilation processed approximately 10,000 images per second compared to 6,997 images per second for the basic PyTorch CPU implementation.

2) Accuracy Analysis: All implementations maintained nearly identical accuracy (96.1 – 96.3 %), since the CNN was pretrained and the same exported weights were loaded into every variant. This confirms that our optimization strategies preserved model integrity.

3) Memory Usage Patterns: As illustrated in Figure 1, memory utilization varied significantly across implementations:

- **CUDA GPU Implementations:** Both CUDA GPU versions (NCHW and NHWC) demonstrated excellent memory efficiency, with peak memory usage under 150 MB. The NCHW layout variant showed only slightly higher memory consumption than the NHWC implementation.
- **CPU Implementations:** CPU-based processing required substantially more memory, with the Optimized CPU (JIT) version consuming approximately 1,600 MB at peak compared to around 630 MB for the basic CPU (PyTorch) implementation. This 2.5$\times$ increase in memory footprint represents a significant tradeoff for the 1.43$\times$ throughput gain.

4) Execution Time Breakdown: Figure 2 provides a detailed breakdown of execution time:

- **CPU (User/System) Time:** The JIT-optimized build incurred roughly twice the system time (2 s vs. 1 s), while its user time remained on par with the standard PyTorch implementation.

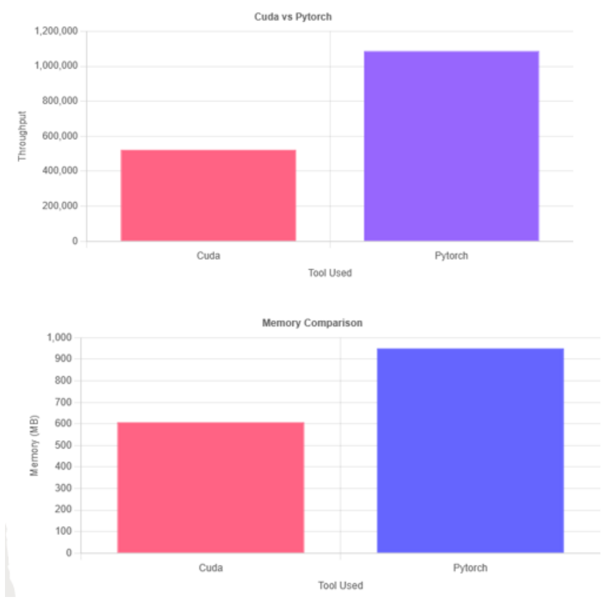

Fig. 3. Framework comparison between custom CUDA implementation and PyTorch. Top: PyTorch achieves substantially higher throughput. Bottom: PyTorch requires significantly more memory to achieve its performance advantage.

- **Wall-Clock Time:** As expected, wall clock time (which is the actual elapsed time) increased slightly for the non-optimized version.

5) Additional Framework Comparison: We conducted additional experiments comparing the performance of different framework approaches, with results visualized in Figure 3.

- **Throughput:** As shown in Figure 3 (top), a separate PyTorch implementation achieved approximately twice the throughput of our custom CUDA implementations (1,100,000 vs. 550,000 img/s), highlighting the benefits of PyTorch's highly optimized backend operations when utilizing its GPU capabilities.
- **Memory Utilization:** This performance advantage comes at a cost—as illustrated in Figure 3 (bottom), the optimized PyTorch implementation consumed approximately 950 MB compared to 600 MB for our custom CUDA implementations, a 58% increase in memory usage.

VI. Conclusion

A. Memory Layout Selection

Our results clearly demonstrate that for our particular CNN architecture on the NVIDIA T4 GPU, NCHW provides substantially better performance. The CUDA GPU (NCHW) implementation achieved over 2× higher throughput (521,462 vs. 253,271 img/s) compared to the CUDA GPU (NHWC) implementation. This dramatic difference demonstrates the significant impact of memory layout optimization on our particular GPU architecture.

Both CPU implementations showed substantially lower throughput. The optimized CPU model with multithreading and JIT compilation processed approximately 10,000 images per second compared to 6,997 images per second for the basic PyTorch CPU implementation. The optimizations (JIT compilation and multithreading) provided a small speedup, demonstrating the benefit of eliminating Python interpreter overhead.

B. CPU vs. GPU Deployment Tradeoffs

Both CUDA GPU implementations dramatically outperformed CPU-based inference in terms of raw throughput, with even the NHWC implementation processing images at 36× the rate of the basic CPU (PyTorch) implementation.

C. Framework Selection Considerations

Additional experiments comparing our custom CUDA implementations with framework-optimized approaches revealed that despite offering greater control, our custom implementations achieved only half the throughput of highly optimized framework-based approaches (Pytorch). This is shown in Figure 3.

References

[1] X. Fu, X. Zhang, J. Ma, P. Zhao, S. Lu, and X. T. Liu, "High Performance Im2win and Direct Convolutions Using Three Tensor Layouts on SIMD Architectures," IEEE High Performance Extreme Computing Conference (HPEC), 2024.

[2] Intel Corporation, "oneDNN Documentation: Memory Format Propagation," Intel GitHub, 2023.

[3] A. Heinecke, E. Georganas, K. Banerjee, D. Kalamkar, N. Sundaram, A. Venkat, G. Henry, and H. Pabst,"Understanding the Performance of Small Convolution Operations for CNN on Intel Architecture,"in *Proceedings of the ACM/IEEE Supercomputing Conference (SC'17)*, Denver, CO, Nov. 2017.

[4] X. Fu, X. Zhang, J. Ma, P. Zhao, S. Lu, and X. T. Liu, "High Performance Im2win and Direct Convolutions Using Three Tensor Layouts on SIMD Architectures,"*arXiv preprint arXiv:2408.00278*, Aug. 2024.

[5] V. W. Lee *et al.*,"Debunking the 100X GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU,"in *Proceedings of the 37th International Symposium on Computer Architecture (ISCA)*, Saint-Malo, France, Jun. 2010.

[6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips,"GPU Computing,"*Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[7] R. Raina, A. Madhavan, and A. Y. Ng,"Large-scale Deep Unsupervised Learning Using Graphics Processors," in *Proceedings of the 26th International Conference on Machine Learning (ICML '09)*, Montreal, Canada, Jun. 2009.

[8] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative Study of Deep Learning Software Frameworks,"*arXiv preprint arXiv:1511.06435*, Nov. 2015. :contentReference[oaicite:2]index=2

[9] J. Li, Z. Qin, Y. Mei, J. Cui, Y. Song, C. Chen, Y. Zhang, et al.,"oneDNN Graph Compiler: A Hybrid Approach for High-Performance Deep Learning Compilation," arXiv preprint arXiv:2301.01333 [cs.LG], Jan. 2023.

[10] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions," arXiv preprint arXiv:1802.04730 [cs.LG], Feb. 2018.

Project Repository

For all code, scripts, and reproducibility instructions, see the project GitHub repository:
https://github.com/Javad228/MSML605Proj