CONVOLUTION: A STUDY OF CONVOLUTION IN SEQUENTIAL AND PARALLEL ENVIRONMENT

TECHNICAL REPORT

Author 1*
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712
author1@utexas.edu

Author 2
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712
author2@utexas.edu

November 18, 2025

ABSTRACT

The 2D convolution operation is a foundational yet computationally expensive bottleneck in modern deep learning models, particularly in Convolutional Neural Networks (CNNs). Optimizing this primitive is essential for achieving practical training and inference speeds.

This projects implements, benchmarks, and compares the performance of 2D convolution by contrasting a sequential, single-core implementation against two parallel paradigms: multi-core CPU using OpenMP and general-purpose GPU using OpenCL.

These implementations were evaluated against four distinct workloads, including memory-bound (e.g., 2048x2048 Box Blur) and compute-bound (e.g., 1024x1024 Big Kernel) tasks. Additionally, a high-level MNIST CNN application was developed to compare framework-level performance (PyTorch CPU vs. PyTorch OpenCL) and the impact of data precision (FP32 vs. INT8 quantization).

Performance results from the low-level C++ benchmark demonstrated that GPGPU parallelism offers a dramatic and scalable advantage over CPU parallelism. For the 2048x2048 blur task, the OpenCL implementation was $\sim$131 times faster than the 16-thread vectorized OpenMP implementation and $\sim$1601 times faster than the sequential baseline. At the high-level framework level, post-training quantization of the MNIST model from FP32 to INT8 provided a $\sim$74% reduction in model size and a 1.20x inference speedup with negligible loss in accuracy.

All source code are available on GitHub.

Keywords Convolution · Parallel Algorithm · Performance Benchmark · GPU Programming

1 Introduction

In the past decade, Deep Learning (DL) has become a dominant force in technology, driving advancements in fields from natural language processing to scientific discovery. A cornerstone of this revolution is the Convolutional Neural Network (CNN), an architecture inspired by the human visual cortex Hubel and Wiesel [1959], LeCun et al. [1998]. CNNs are the industry standard for computer vision (CV) tasks, powering applications from medical image analysis to autonomous driving Krizhevsky et al. [2012].

The power of a CNN stems from its foundational operation: the 2D convolution. This is a mathematical process where a small matrix of weights, known as a filter or kernel, "slides" across an input image, performing element-wise multiplications and sums at each position.

$$O(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} I(x+m, y+n) \cdot K(m,n) \tag{1}$$

*Use footnote for providing further information about author (webpage, alternative address)—*not* for acknowledging funding agencies.

- $O(x, y)$ is the output pixel at coordinates (x, y).
- I represents the input image intensity.
- K is the kernel of size $M \times N$.
- m, n are the kernel indices.

This operation is exceptionally effective at detecting hierarchical patterns like edges, textures, and shapes. While powerful, 2D convolution is computationally intensive, requiring billions of floating-point operations (FLOPs) to process a single image in a modern, "deep" network. A sequential, single-threaded implementation of this operation is a critical bottleneck, rendering these powerful models unusable for practical, real-time applications.

The solution to this bottleneck is parallelism. This project investigates and quantifies the performance of the two primary parallel computing strategies:

- **Shared-Memory CPU Parallelism:** This approach utilizes the multiple cores available in modern CPUs. We chose OpenMP for this task, as it is the industry-standard, pragma-based framework for easily and effectively parallelizing computationally heavy loops across CPU threads OpenMP Architecture Review Board [2018].
- **GPGPU (Accelerator) Parallelism:** This approach offloads the computation to the thousands of smaller, specialized cores on a Graphics Processing Unit (GPU). We selected OpenCL as our framework, a powerful and vendor-agnostic standard that allows compute kernels to run on GPUs from any manufacturer, including AMD, NVIDIA, and Intel Munshi et al. [2020].

While models like GPT (Transformers) are driven by 1D convolution and massive matrix multiplication Vaswani et al. [2017], this report will focus on the 2D convolution primitive, which is equally, if not more, foundational to the broader field of deep learning.

To provide a comprehensive analysis, this investigation was split into two parts. First, a low-level C++ benchmark was developed to directly compare the performance of Sequential, OpenMP, and OpenCL implementations under various workloads. Second, a high-level application—a PyTorch CNN to classify handwritten digits based on MNIST dataset LeCun et al. [1998]—was built. This practical demo allows us to explore further design alternatives, such as the performance of framework-level backends (PyTorch-OpenCL) and the impact of critical optimization techniques like INT8 quantization.

This report will present the findings from these investigations. Section 3 describes the components of the C++ benchmark and PyTorch application. Section 4 analyzes the various design alternatives considered. Section 5 presents the quantitative performance results from all benchmarks. Finally, Section 6 provides a conclusion and suggests areas for future work.

2 Project Description

To effectively evaluate the performance of parallel convolution, this project was divided into two main components: a low-level C++ benchmark designed for raw performance analysis, and a high-level PyTorch application built to explore these concepts in a real-world context.

2.1 C++ Benchmark

The core of the investigation is a custom C++ benchmark harness. This application is designed to provide a controlled, "bare-metal" comparison of different convolution implementations, free from the overhead of high-level frameworks.

2.1.1 Test Harness

The main program acts as a robust test harness responsible for managing the entire benchmark suite. It probes the system for OpenCL-capable devices and initializes the OpenCL context, loading and building the custom kernels we wrote. Simple checks on OpenMP status are done too. It iterates through a series of predefined test cases. For each case, it runs each convolution implementation 10 times to gather reliable, averaged performance data. It uses the output of the sequential implementation as the "golden" reference. The output of every parallel implementation is compared against this reference to ensure computational correctness. It collects all timing and pass/fail results into a final summary table.

2.1.2 Test Cases

To provide a comprehensive analysis of performance scaling, we moved away from arbitrary static test cases. Instead, we implemented a programmatic test suite designed to stress and compare memory bandwidth versus computational intensity.

The benchmark suite now consists of three distinct scenarios:

1. **Baseline Correctness (5x5 Edge Detect):** A trivial 5×5 input with a 3×3 kernel. This ensures that all implementations (Sequential, OpenMP, OpenCL) produce mathematically identical results before running heavy workloads. It also serves to measure the "launch overhead" of parallel APIs.

2. **Memory Bandwidth Stress Test (Scaling Image Size):** With a fixed 3×3 kernel over various Image Dimension ($N \times N$), scaling from 256×256 up to 4096×4096. The idea is that with a small kernel, the Arithmetic Intensity (FLOPS per byte fetched) is low. As the image size grows from fitting in the L2 cache (256^2) to exceeding L3 and requiring heavy RAM access (4096^2), performance becomes strictly bound by how fast data can be moved from main memory to the cores.

3. **Compute Bound Stress Test (Scaling Kernel Size):** With a fixed Image Dimension (1024×1024) and various kernel size ($K \times K$), scaling from 3×3 up to 15×15. Convolution complexity grows quadratically with kernel size ($O(K^2)$). A 15×15 kernel requires 225 multiplications per pixel, compared to just 9 for a 3×3 kernel. This test keeps memory usage constant while dramatically increasing the number of floating-point operations, shifting the bottleneck to the CPU/GPU's Arithmetic Logic Units (ALUs).

2.1.3 Implementations

The harness executes three functionally identical implementations of 2D convolution:

- **Sequential CPU:** A standard, single-threaded C++ implementation using nested for loops. This serves as the performance baseline.
- **Parallel CPU:** A shared-memory parallel implementation using OpenMP. A '`#pragma omp parallel for`' directive is applied to the outer y loop, automatically distributing the rows of the output image across all available CPU threads.
- **Parallel GPU:** A GPGPU implementation. The C++ host program manages memory buffers and enqueues the custom 'convolve_fp32' kernel, which is written in C-like OpenCL programming language. This kernel executes on the GPU, with one thread being launched for each pixel in the output image.

2.2 CNN-based Handwriting Recognizer Demo Application

To connect the low-level benchmark to a practical application, a second project component was developed to train, optimize, and serve real-world CNN models. The models are then served via a custom Flask web application to enable users to draw and classify numbers via a web browser.

2.2.1 CNN Model and Training

A LeNet-style CNN was built using PyTorch LeCun et al. [1998], Paszke et al. [2019]. The architecture consists of two convolutional blocks (Conv-ReLU-Pool) followed by two fully-connected layers. This model was trained on the MNIST dataset to achieve high accuracy ($\sim$99.33%) in classifying handwritten digits+.

2.2.2 Web Application

A simple Python web application was created using the Flask framework. This application serves as a practical interface for the project by providing a "Digit Drawer" web page where a user can draw a number on an HTML canvas. When the user clicks "Predict," the drawing is sent to the Flask backend. The backend runs inference on the user's drawing using two different, pre-trained models: the standard FP32 model and an INT8-quantized model. The application returns the predictions, confidence scores, and inference times for both models, allowing for a direct, real-world comparison of the two.

3 Design Alternatives Considered

The central goal of accelerating 2D convolution can be achieved through multiple distinct strategies. During this project, several key design alternatives were considered, implemented, and analyzed. These alternatives fall into three main categories: CPU parallelization strategy, the level of GPU abstraction, and the choice of data precision.

3.0.1 Alternative 1: CPU Parallelization Strategy (OpenMP & SIMD)

Optimizing convolution on the CPU requires utilizing two levels of parallelism: multi-threading (scaling across cores) and vectorization (scaling within a core). We explored two implementations:

- **Naive Parallelism (Baseline):** The initial approach involves adding a single '`#pragma omp parallel for`' directive to the outermost loop (y coordinate).

  Pros: This is the simplest possible implementation, requiring only one line of code. It is highly effective for "embarrassingly parallel" tasks and provides a clear demonstration of scalability as thread count increases.

  Cons: It leaves significant performance on the table. Modern CPUs support Single Instruction, Multiple Data (SIMD) instructions (e.g., AVX2) that can process 8 floating-point numbers in a single cycle Intel Corporation [2023]. The naive implementation processes pixels sequentially (scalar), utilizing only a fraction of the core's theoretical throughput.

- **Vectorized Parallelism (Implemented):** We implemented a robust Vectorization strategy targeting AVX2/AVX-512 instruction sets. This involved fundamentally restructuring the inner loops to enable the compiler's auto-vectorizer. Memory Alisasing proved to be a major challenge, as the compiler cannot prove that input and output pointers do not overlap, forcing it to serialize writes for safety.

  We implemented a "Vectorization-First" kernel using three key techniques:

  Loops need to start from zero: Inner loops were rewritten to strictly iterate from 0 to kernel size, removing complex boundary checks from the hot path.

  Explicit Directives: We employed '`#pragma omp simd reduction(+:sum)`' directive and the `__restrict` keyword to explicitly grant the compiler permission to ignore aliasing and reorder floating-point reductions.

  Fast Floating Point: We have to explicitly add compiler flag to permit the generation of fused multiply-add (FMA) instructions.

  Vectorising convolution allowed the CPU to process 8 pixels simultaneously per thread, yielding a massive speedup over the naive OpenMP implementation, bringing CPU performance a step closer to hardware accelerator.

3.0.2 Alternative 2: GPU Acceleration (Low-Level vs. High-Level)

A primary goal was to compare CPU performance to GPGPU acceleration. This project explored both ends of the abstraction spectrum for this task.

- **Low-Level (C++/OpenCL):** This is the method used in the C++ benchmark. It involves writing a custom compute kernel and managing its execution from a C++ host program. **Pros:** This provides maximum control and performance. The developer manages all memory buffers ('clCreateBuffer'), kernel arguments ('clSetKernelArg'), and work-group dispatching ('clEnqueueNDRangeKernel'). This allows for fine-grained optimizations, such as the 'convolve_int8' kernel also developed for this project. **Cons:** This method has a very high implementation complexity. It requires significant boilerplate code for context setup, error checking, and kernel compilation, and it requires writing in a separate kernel language.

- **High-Level (Framework Backend):** This method was explored in the CNN model building part of the project. Instead of writing a kernel, we used a high-level framework (PyTorch) with a third-party OpenCL backend '`pytorch_dlprim`' Belevich [2024]. **Pros:** This approach offers extreme simplicity and productivity. The code for training the model on the GPU is identical to the CPU code, with the only change being the device specification (e.g., 'device=torch.device("ocl:0")'). The backend handles all kernel compilation and memory management automatically. **Cons:** The developer is entirely dependent on the quality of the third-party backend. It acts as a "black box," and GPU utilization appears generally lackluster compared to hand-tuned implementations in CUDA or ROCm.

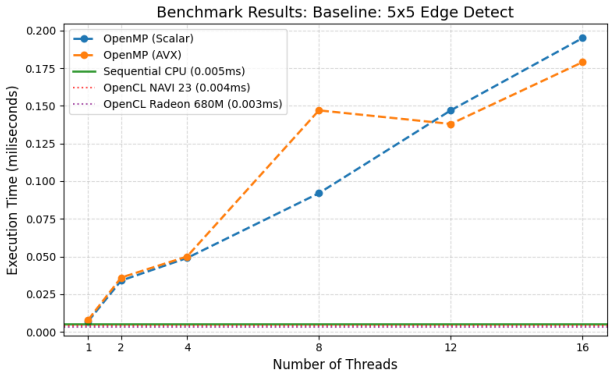


Figure 1: Enter Caption

3.0.3 Alternative 3: Data Precision (FP32 vs. INT8 Quantization)

The final design alternative explored was the trade-off between 32-bit floating-point precision and 8-bit integer quantization, a critical optimization in modern deep learning Jacob et al. [2018].

- **FP32 (32-bits Floating Point)** This is the standard data type for training and default inference. **Pros:** High precision, ensuring no loss of model accuracy. **Cons:** Requires 4 bytes per number, leading to larger model sizes, higher memory bandwidth usage, and reliance on slower floating-point ALUs.
- **INT8 (Quantized 8-bit Integer):** This is an optimization where 32-bit floats are mapped to an 8-bit integer range for inferencing. **Pros:** 4x reduction in model size and memory bandwidth. Integer arithmetic is significantly faster on modern CPUs and GPUs, leading to faster inference. **Cons:** Requires a "calibration" step to determine the mapping ranges. This can introduce a small, though often negligible, loss in accuracy. Also requires additional quantization stubs in the CNN model that OpenCL PyTorch backend does not support.

This alternative was also explored at both the high and low levels:

1. **Framework Level:** We demonstrated a full post-training quantization (PTQ) workflow. The FP32 model is trained, and then a calibrated INT8 version is created. The results (presented in Section 5) show a $\sim$74% model size reduction with negligible accuracy drops, with close to 20% speedup in inference.
2. **Kernel Level:** The low-level logic for this was implemented directly in our OpenCL kernel file. The ‘convolve_int8‘ kernel was written to accept 8-bit (char) inputs and weights. To prevent overflow, it correctly casts these to 32-bit integers for multiplication and accumulation. The final 32-bit sum is then re-quantized back to an 8-bit output. While this kernel was successfully written, it was not integrated into the final C++ benchmark harness and remains a clear avenue for future work.

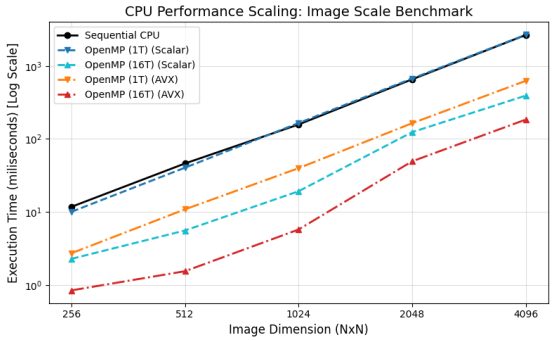
4 Performance Results and Analysis

The performance of each implementation was benchmarked, with all tests (excluding the "5x5" overhead test) averaged over 10 runs. The results are divided into two sections: the low-level C++ benchmark and the high-level PyTorch application analysis.

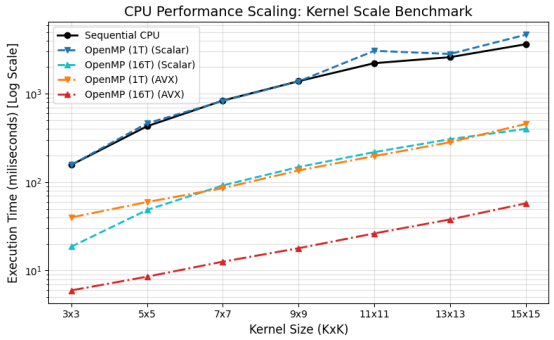
4.1 C++ Benchmark: CPU (OpenMP) Scalability

First, the scalability of the OpenMP implementation was measured by running it with 1, 2, 4, 8, 12, and 16 threads. The results, plotted on a logarithmic scale, are shown in Figure 2.

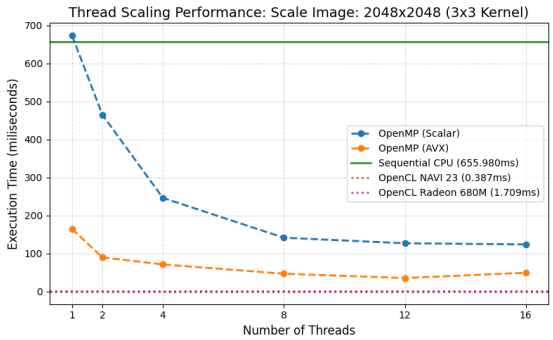
As we can observe in Figure 1 for the overhead-dominated 5x5 test, performance became worse as threads were added (0.005 ms for Sequential vs. 0.195 ms for 16T). This is because the overhead of creating and synchronizing threads was far greater than the negligible computation time. In Figure 2(c), the memory-bounded 2048x2048 test scaled well up to 12 threads (126.860 ms) but no improvements at 16 threads (123.877 ms). This plateau indicates the task became memory-bandwidth bound; the CPU cores were stalling, waiting for data from RAM, and adding more cores provided no benefit. In Table 1, the 1024x1024 Big Kernel test, which is compute-bound, scaled the best, showing a 9.4x speedup on 16 threads (148.152 ms) compared to the sequential run (1386.488 ms).



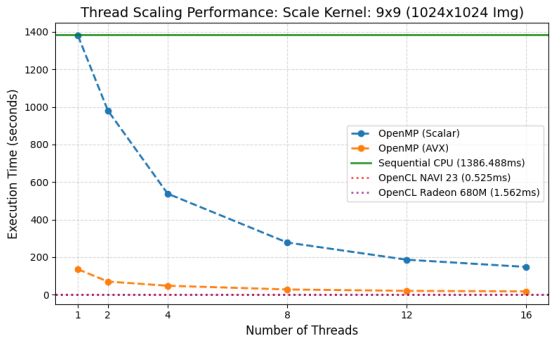
(a) Image Scale CPU



(b) Kernel Scale CPU



(c) Image Scale Speed vs Threads



(d) Kernel Scale Speed

Figure 2: Comparison of CPU performance in various thread counts

4.2 C++ Benchmark: Final Implementation Comparison

Next, the best-performing parallel implementations were compared against the sequential baseline. Table 1 summarizes the average execution times for four "signature" test cases that each stress a different part of the hardware-software stack.

Table 1: C++ Benchmark Results for Sequential, 16-Thread OpenMP, and OpenCL on two GPUs.

Test Case	Seq	OMP-16T	OMP-16T-AVX	OCL (NAVI 23)	OCL (680M)
5x5 Edge Detect	0.003	0.233	0.179	0.007	0.018
1024x1024 Identity	157.714	19.179	5.960	0.138	0.367
2048x2048 Box Blur	539.065	123.877	49.339	0.390	1.735
1024x1024 Big Kernel	1386.488	148.152	17.874	0.528	1.385

As seen in Table 1, the GPGPU implementation on the NAVI 23 GPU provided a transformative performance increase. On the memory-bound 2048x2048 box blur test, the OpenCL kernel was ∼127 times faster than the 16-thread OpenMP vectorized implementation (0.390 ms vs. 49.339 ms). On the compute-bound 1024x1024 9x9 Kernel test, this advantage was maintained, with the OpenCL kernel proving 34 times faster than the 16-thread vectorized OpenMP implementation (0.528 ms vs. 17.874 ms). The comparison between the two GPUs (NAVI 23 vs. Radeon 680M) also highlights the significant performance difference between a discrete, high-performance GPU and an integrated mobile GPU which have very different memory layouts, despite both using RNDA2 microarchitecture Advanced Micro Devices, Inc. [2020].

Additional, as we can observe in Figure 3, even the fastest CPU implementation that parallelize and vectorize still is magnitude slower than both GPU tested, in both image dimension scaling and kernel size scaling benchmarks. This shows the massive parallel advantage of GPU when facing the right algorithms.

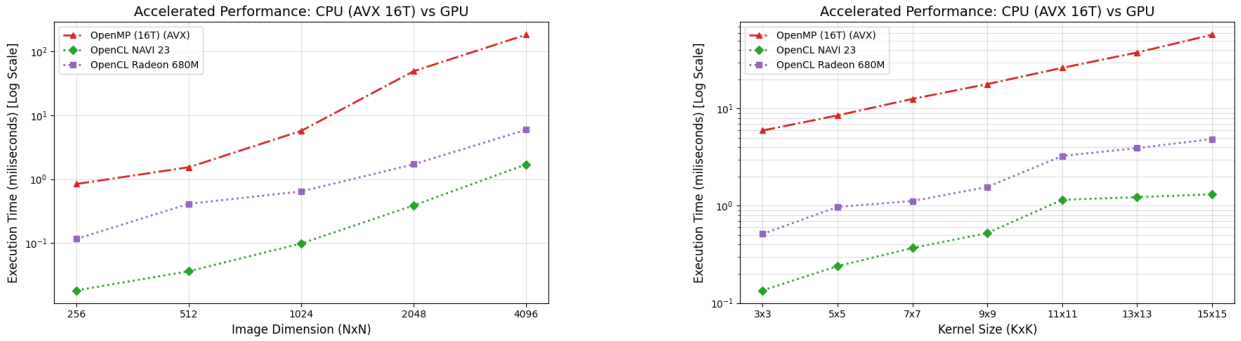


Figure 3: Vectorized CPU vs GPU performance scaling across different Image and Kernel sizes

4.3 High-Level Application: Framework and Quantization Results

The results from the high-level PyTorch application complement the low-level C++ findings. The PyTorch CPU backend used here is Facebook GEneral Matrix Multiplication (FBGEMM), which uses OpenMP for CPU parallelization Meta Platforms [2024].

Table 2: PyTorch Training Time (CPU versus OpenCL Backend) of MNIST CNN for 10 epochs.

Implementation	Training Time	Hardware Utilization
PyTorch OpenCL GPU0	3.13 minutes	15%
PyTorch OpenCL GPU1	4.75 minutes	30%
PyTorch CPU (16T)	7.40 minutes	50%

Table 3: Quantization impact across various thread counts, Inference Speed measured in samples per second

Metric	FP32	INT8	Change
Model Size	1.74 MB	0.45 MB	-74.1%
Accuracy	99.17%	99.16%	-0.01%
Inf. Speed (16T)	2141.32	2570.53	20.0%
Inf. Speed (14T)	2330.19	2659.39	14.1%
Inf. Speed (12T)	2301.10	2708.73	17.7%
Inf. Speed (8T)	2170.47	2570.72	18.4%
Inf. Speed (4T)	1781.53	2232.06	25.3%
Inf. Speed (2T)	1429.40	1938.85	35.6%
Inf. Speed (1T)	942.51	1453.89	54.2%

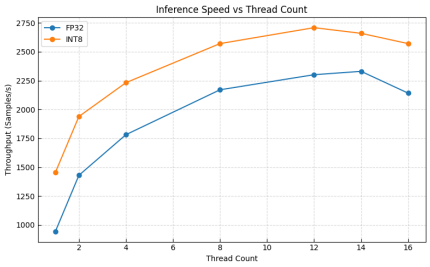


Figure 4: CPU inference speed in number of samples per second across different thread count

Framework Parallelism: As seen in Table 2, simply switching the PyTorch from CPU to GPU0 reduced the total model training time from 7.40 to 3.13 minutes, a 2.36x speedup when training the same model. Even upgrading to the much less powerful GPU1 yields 1.56x speedup. These are all under the condition that the GPU hardware is severely underutilized due to small model size, showing that GPU is the go-to way to train CNN models. An interesting observation for CPU inferencing as observed in Figure 4 is that the speed started to decrease as the number of threads assigned approach maximum. It is entirely unclear why this occurs and may be a feature of PyTorch.

Quantization Optimization: As shown in Table 3, post-training quantization provided a massive 74.1% reduction in model size while delivering a 1.20x inference speedup, all for a margin-of-error 0.01% drop in accuracy. It should be noted that Pytorch OpenCL does not currently support quantized models so we only have comparisons for CPU.

5 Future Work

The results of this project open up several clear avenues for future investigation:

Test Larger CNN Models: The MNIST model is trained to classify simple, small 28x28 grayscale images instead of large RGB images. It would be great addition to benchmark the performance of large scale CNN models such as ResNet50 both in quantized CPU mode and OpenCL mode to push the computing units harder on the hardware He et al. [2016].

Test on big.LITTLE CPU: Modern consumer CPU often utilizes a mix of small and big cores, where small cores are power efficient and the big cores have very high compute power. It would be interesting to see the parallelization performance of convolution across various combinations of big and little cores.

Compare OpenCL against Vendor-Specific Backends: This project used OpenCL as a vendor-agnostic standard. A valuable comparison would be to benchmark these results against a highly-optimized, vendor-specific library such as NVIDIA's CUDA and cuDNN, or AMD's ROCm. This would help quantify the performance difference between a general-purpose standard and a proprietary, hardware-specific solution.

6 Conclusion

This project successfully implemented and evaluated 2D convolution across sequential, parallel CPU, and GPGPU paradigms, confirming that performance is a product of both parallelization strategy and design-level choices. The investigation yielded three primary conclusions:

GPGPU parallelism is decisively superior for compute-bound tasks. The low-level C++ benchmark provided the clearest evidence. While OpenMP vectorized scaled well on multi-core CPUs (achieving a 12.20x speedup on a 16-thread CPU over non-vectorized sequential for the 2048x2048 task), it was completely outmatched by the OpenCL implementation. On the same 2048x2048 task, the "NAVI 23" GPU was $\sim$131 times faster than the 16-thread AVX2 OpenMP implementation (0.376 ms vs 49.339 ms) and a jaw-dropping $\sim$1601 times faster than the sequential baseline (601.955 ms).

High-level frameworks effectively abstract parallel complexity. The PyTorch application demonstrated that developers do not need to write low-level kernels to leverage GPGPU power. By using the `pytorch_dlprim` backend, our model's training time was reduced from 7.40 minutes to 3.13 minutes.

Quantization is a critical, low-cost optimization. The comparison of FP32 and INT8 PyTorch models proved the value of data precision optimization. Post-training quantization yielded a $\sim$74% reduction in model file size and a 1.20x inference speedup. This was achieved with no drop in model accuracy, making it a clear and highly effective strategy for deploying models on resource-constrained devices.

References

David H Hubel and Torsten N Wiesel. Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574–591, 1959.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 5.0*, November 2018. URL `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf`.

Aaftab Munshi et al. *The OpenCL Specification, Version 3.0*. Khronos OpenCL Working Group, 2020. URL `https://www.khronos.org/registry/OpenCL/specs/3.0-unified/pdf/OpenCL_API.pdf`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8024–8035, 2019.

Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 2023. Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.

Artyom Belevich. pytorch_dlprim: Opencl backend for pytorch. `https://github.com/11111111/pytorch_dlprim`, 2024.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2704–2713, 2018.

Advanced Micro Devices, Inc. *"RDNA 2" Instruction Set Architecture: Reference Guide*, November 2020. URL `https://developer.amd.com/wp-content/resources/RDNA2_Shader_ISA_November2020.pdf`.

Meta Platforms. Fbgemm: Facebook general matrix multiplication. `https://github.com/pytorch/FBGEMM`, 2024.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

A Supplemental Screenshot

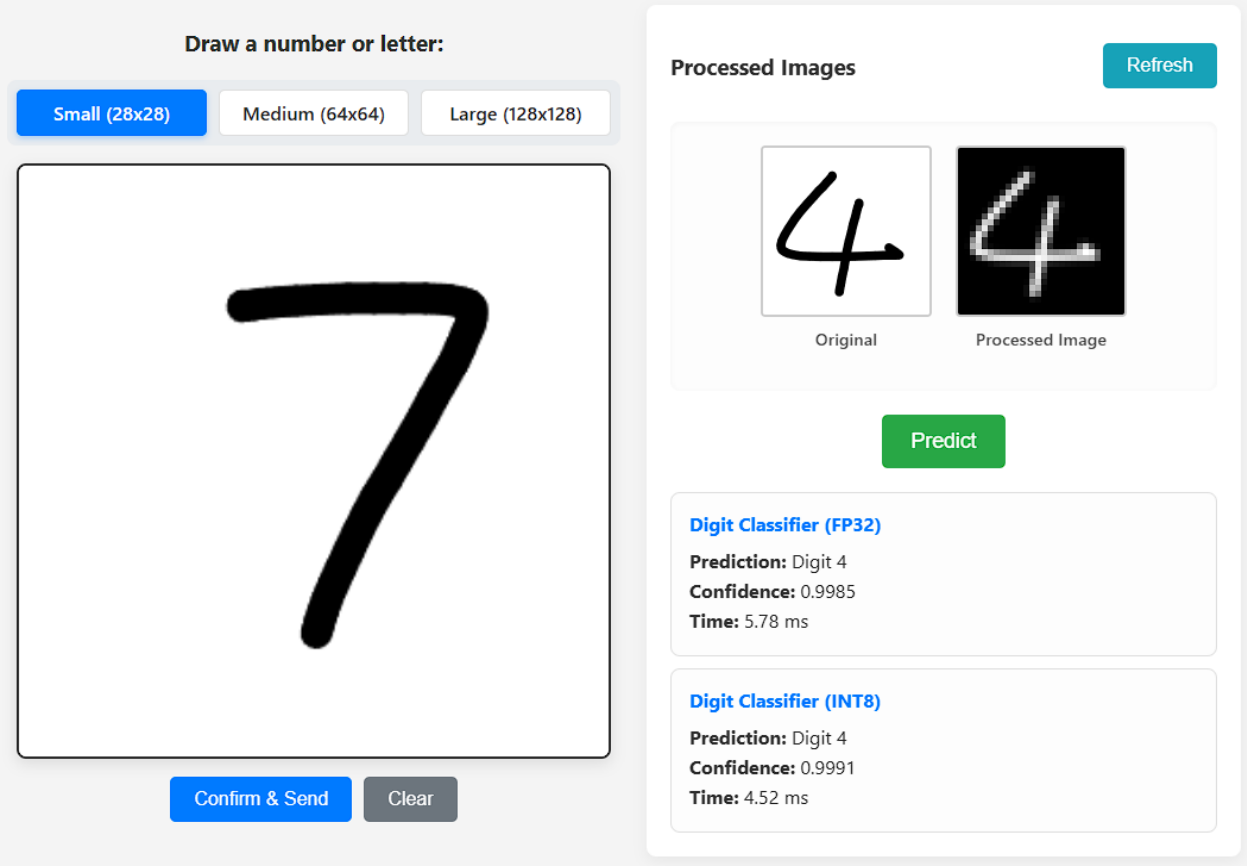


Figure 5: Screenshot of the web application hosting the CNN models running.