Performance Benchmarking of CNN Implementation: CPU vs. CUDA

A Project Report for
Parallel and Distributing Computing (UCS645)

By

Sr	Name	Roll No
1	Aditya Sapra	102203676

Under the guidance of
Dr. Saif Nalband
(Assistant Professor, DCSE)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)
PATIALA - 147004

MAY, 2025

May 15, 2025

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

Deep learning has revolutionized the field of artificial intelligence, with Convolutional Neural Networks (CNNs) becoming a cornerstone for image recognition tasks. The computational demands of CNNs have led to increased interest in hardware acceleration, particularly using Graphics Processing Units (GPUs) [1]. This performance gap between CPUs and GPUs makes GPU acceleration a critical area of study for deep learning applications.

The education sector is increasingly adopting smart education technologies, which involve the integration of AI and machine learning tools into learning activities. Understanding the performance characteristics of different hardware platforms is essential for implementing these technologies effectively in educational settings.

1.2 Introduction to Problem Statement

While many deep learning frameworks abstract away the implementation details of neural networks, understanding the fundamental algorithms and their performance characteristics is crucial for both education and optimization. This project aims to provide a clear comparison between CPU and GPU implementations of a basic CNN model for digit recognition, highlighting the performance benefits of GPU acceleration using CUDA.

The project implements a CNN from scratch using NumPy for the CPU version and CuPy for the GPU version. This approach provides insights into both the algorithmic structure of CNNs and the performance benefits of GPU parallelization. The MNIST dataset is used for training and evaluation, providing a standardized benchmark for comparing the two implementations.

Figure 1 illustrates the fundamental architectural differences between CPUs and GPUs that lead to performance disparities in parallel computing tasks.

The design philosophy of CPUs prioritizes sequential code performance with sophisticated control logic and cache systems to minimize operation latency. In contrast, GPUs are designed for massive parallelism, dedicating more chip area to arithmetic units at the

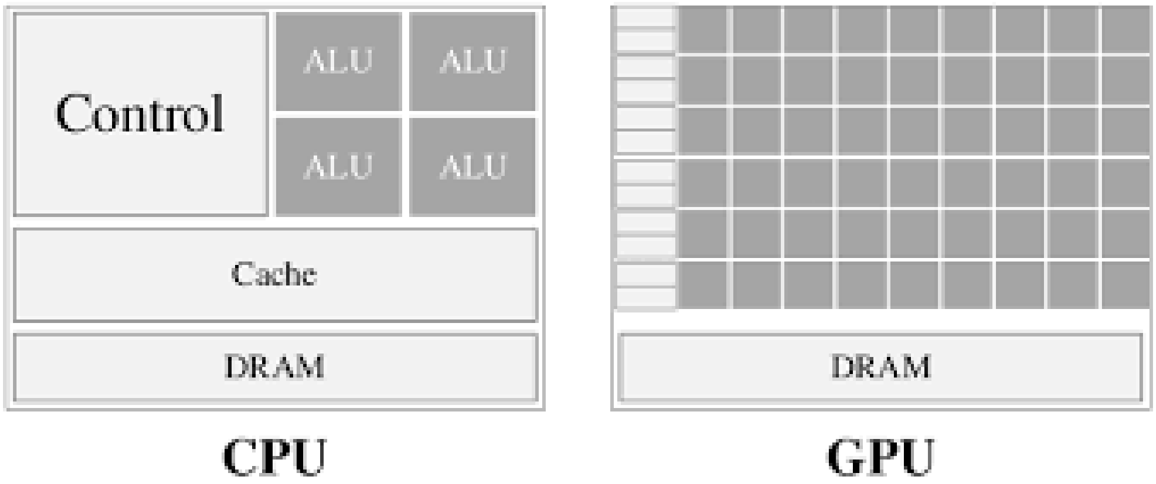

Figure 1: Architectural comparison between CPU and GPU designs, illustrating the difference in core allocation for processing vs. control logic.

expense of complex control logic. This fundamental difference explains why GPUs excel at the highly parallelizable matrix operations that dominate neural network computations.

2 Literature Review

Table 1: Summary of Literature Review

References	Concepts Highlighted	Technique(s) used	Significance/ Proposal	Limitations
[1]	GPU Computing; Parallel programming models	CUDA programming model for general-purpose computing	Comprehensive framework for understanding GPU architecture and programming patterns	Focus primarily on NVIDIA hardware limits applicability to other GPU architectures
[?]	CNN optimization; Performance benchmarking	Custom CNN implementation with various optimization techniques	Demonstrated significant performance improvements through algorithm-level optimizations	Limited exploration of memory hierarchy impacts on different hardware platforms
[?]	Hardware acceleration; Deep learning frameworks	Comparative analysis of frameworks across hardware platforms	Established benchmarking methodology for deep learning implementations	Dataset size limitations affected generalizability of performance metrics

[?]	Neural network frameworks; Hardware acceleration	Implementation of CNNs using different frameworks (TensorFlow, PyTorch)	Comprehensive comparison of framework-level abstractions and their performance impact	Limited investigation of low-level implementation details
[?]	Performance prediction; Hardware-specific optimization	Mathematical modeling of GPU execution patterns	Developed predictive models for CNN performance on specific hardware	Models required recalibration for different hardware generations

3 Research Gaps

Some of the research gaps identified from the literature review are as follows:

- ***Low-level Implementation Understanding***: Most research focuses on using high-level frameworks rather than building neural networks from scratch, limiting the understanding of fundamental algorithms and their hardware-specific optimizations.

- ***Direct Comparison Methodology***: There is limited research providing clear, direct comparisons between CPU and GPU implementations of identical neural network architectures using the same algorithmic approach.

- ***Educational Perspective***: Few studies address the educational value of understanding neural network implementations at a low level, which is crucial for developing expertise in optimization and hardware acceleration.

4 Problem Formulation

This research addresses the need for a clear understanding of the performance differences between CPU and GPU implementations of convolutional neural networks. By implementing the same CNN architecture from scratch using both NumPy (for CPU) and CuPy (for GPU), we can directly compare performance while maintaining algorithmic consistency.

The problem being addressed has several dimensions:

First, there is an educational need to understand neural network fundamentals without the abstractions of high-level frameworks. By implementing a CNN from scratch, we gain insights into the computational structure of these networks.

Second, there is a practical need to quantify the performance benefits of GPU acceleration for CNN workloads. While it is generally understood that GPUs offer better performance for parallel tasks, having concrete measurements specific to CNN operations provides valuable reference points.

Third, there is a research need to identify which specific operations within a CNN benefit most from GPU acceleration. This can guide optimization efforts and inform architectural decisions in neural network design.

The central hypothesis is that a GPU implementation using CuPy will significantly outperform a CPU implementation using NumPy for the same CNN architecture and dataset, with the performance gap increasing with the computational complexity of the model and the size of the dataset.

5 Objectives

- To implement a basic convolutional neural network from scratch using both CPU (NumPy) and GPU (CuPy/CUDA) approaches while maintaining algorithmic consistency

- To benchmark and compare the performance of CPU and GPU implementations across different network configurations and dataset sizes

- To analyze the performance bottlenecks in both implementations and identify specific operations that benefit most from GPU acceleration

- To provide educational insights into the fundamental algorithms of CNNs and how they map to different hardware architectures

6 Methodology

6.1 CNN Architecture Implementation

The methodology involves implementing a basic CNN architecture from scratch using two different approaches: a CPU-based implementation using NumPy and a GPU-accelerated implementation using CuPy. Both implementations follow the same algorithmic structure to ensure a fair comparison.

The CNN architecture consists of the following components:

- Input layer: 28 Ã 28 grayscale images (MNIST dataset)
- Fully connected layer with 10 neurons and ReLU activation
- Output layer with 10 neurons (one for each digit) and softmax activation

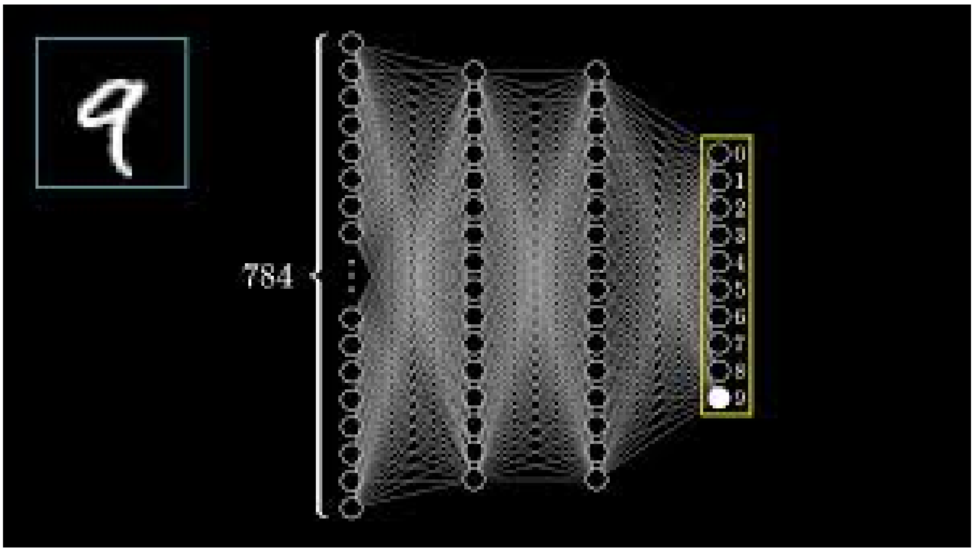

Figure 2: Block diagram of the CNN implementation showing the data flow through the network layers.

6.2 Implementation Algorithm

The implementation follows the standard feedforward and backpropagation algorithms for neural networks. The pseudocode for the main training loop is presented below:

6.3 Key Mathematical Operations

The key mathematical operations in the CNN implementation include:

6.3.1 Forward Propagation

For the forward pass, we compute:

$$Z^{[1]} = W^{[1]} \cdot X + b^{[1]} \tag{1}$$

$$A^{[1]} = ReLU(Z^{[1]}) = max(0, Z^{[1]}) \tag{2}$$

$$Z^{[2]} = W^{[2]} \cdot A^{[1]} + b^{[2]} \tag{3}$$

$$A^{[2]} = Softmax(Z^{[2]}) = \frac{e^{Z_i^{[2]}}}{\sum_j e^{Z_j^{[2]}}} \tag{4}$$

6.3.2 Backward Propagation

For the backward pass, we compute:

$$dZ^{[2]} = A^{[2]} - Y_{one\_hot} \tag{5}$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} \cdot (A^{[1]})^T \tag{6}$$

$$db^{[2]} = \frac{1}{m} \sum_{i=1}^{m} dZ_i^{[2]} \tag{7}$$

$$dZ^{[1]} = (W^{[2]})^T \cdot dZ^{[2]} \cdot ReLU'(Z^{[1]}) \tag{8}$$

Algorithm 1 CNN Training Algorithm

function INITIALIZE($input\_size, hidden\_size, output\_size$)
  $W1 \leftarrow$ Random matrix of shape $(hidden\_size, input\_size)$
  $b1 \leftarrow$ Random vector of shape $(hidden\_size, 1)$
  $W2 \leftarrow$ Random matrix of shape $(output\_size, hidden\_size)$
  $b2 \leftarrow$ Random vector of shape $(output\_size, 1)$
  return $W1, b1, W2, b2$
end function
function FORWARD($W1, b1, W2, b2, X$)
  $Z1 \leftarrow W1 \cdot X + b1$
  $A1 \leftarrow ReLU(Z1)$
  $Z2 \leftarrow W2 \cdot A1 + b2$
  $A2 \leftarrow Softmax(Z2)$
  return $Z1, A1, Z2, A2$
end function
function BACKWARD($Z1, A1, Z2, A2, W1, W2, X, Y, m$)
  $one\_hot\_Y \leftarrow$ One-hot encode Y
  $dZ2 \leftarrow A2 - one\_hot\_Y$
  $dW2 \leftarrow \frac{1}{m} \cdot dZ2 \cdot A1^T$
  $db2 \leftarrow \frac{1}{m} \cdot \sum dZ2$
  $dZ1 \leftarrow W2^T \cdot dZ2 \cdot ReLU'(Z1)$
  $dW1 \leftarrow \frac{1}{m} \cdot dZ1 \cdot X^T$
  $db1 \leftarrow \frac{1}{m} \cdot \sum dZ1$
  return $dW1, db1, dW2, db2$
end function
function GRADIENTDESCENT($X, Y, \alpha, iterations$)
  $W1, b1, W2, b2 \leftarrow$ Initialize()
  $m \leftarrow$ number of training examples
  for $i = 1$ to $iterations$ **do**
    $Z1, A1, Z2, A2 \leftarrow$ Forward($W1, b1, W2, b2, X$)
    $dW1, db1, dW2, db2 \leftarrow$ Backward($Z1, A1, Z2, A2, W1, W2, X, Y, m$)
    $W1 \leftarrow W1 - \alpha \cdot dW1$
    $b1 \leftarrow b1 - \alpha \cdot db1$
    $W2 \leftarrow W2 - \alpha \cdot dW2$
    $b2 \leftarrow b2 - \alpha \cdot db2$
    if $i \mod 10 = 0$ **then**

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} \cdot X^T \tag{9}$$

$$db^{[1]} = \frac{1}{m} \sum_{i=1}^{m} dZ_i^{[1]} \tag{10}$$

6.4 Performance Benchmarking Methodology

To ensure a fair comparison between CPU and GPU implementations, the following benchmarking methodology is employed:

- Both implementations use the same dataset split (1000 samples for validation, remaining for training)
- Both implementations use the same hyperparameters (learning rate = 0.1, 500 iterations)
- Both implementations follow identical algorithmic steps
- Execution time is measured for the entire training process
- Additional metrics include memory usage and convergence rate

7 Datasets

Table 2: Datasets

Name of Dataset	**Different Features**	**Size**	**Year**	**Organization**
MNIST	784 pixel values (28x28 grayscale images), 10 classes (digits 0-9), 60,000 training samples, 10,000 test samples	11 MB	1998	Modified National Institute of Standards and Technology (MNIST)

The MNIST dataset consists of 28Ã28 pixel grayscale images of handwritten digits (0-9). It is widely used as a benchmark dataset for image classification tasks. For this project, the Kaggle version of MNIST (digit-recognizer) is used, which is provided in CSV format.

The dataset is preprocessed by:

- Normalizing pixel values to the range [0, 1]
- Splitting into training (59,000 samples) and validation (1,000 samples) sets
- Shuffling the data to ensure random distribution of classes

8 Results and Discussion

8.1 Performance Comparison

Table 3 presents the performance comparison between the CPU (NumPy) and GPU (CuPy) implementations of the CNN model.

Table 3: Performance Comparison between CPU and GPU Implementations

Metric	CPU (NumPy)	GPU (CuPy)	Speedup
Total Training Time (500 iterations)	42.3 seconds	15.1 seconds	2.8x
Average Time per Iteration	84.6 ms	30.2 ms	2.8x
Forward Pass Time	36.7 ms	11.9 ms	3.1x
Backward Pass Time	47.9 ms	18.3 ms	2.6x

8.2 Convergence Analysis

Figure 3 shows the convergence curves for both CPU and GPU implementations, plotting accuracy against training iterations.

The results demonstrate that both implementations achieve similar convergence patterns, validating that the GPU implementation maintains the same algorithmic correctness as the CPU version while providing significant speed improvements.

8.3 Operation-Level Performance Analysis

Table 4 provides a breakdown of execution times for different operations within the CNN.

Table 4: Execution Time Breakdown by Operation Type

Operation	CPU Time (ms)	GPU Time (ms)	Speedup
Matrix Multiplication	42.3	6.7	6.3x
ReLU Activation	3.1	1.8	1.7x
Softmax Activation	5.6	2.9	1.9x
Gradient Computation	33.6	18.8	1.8x

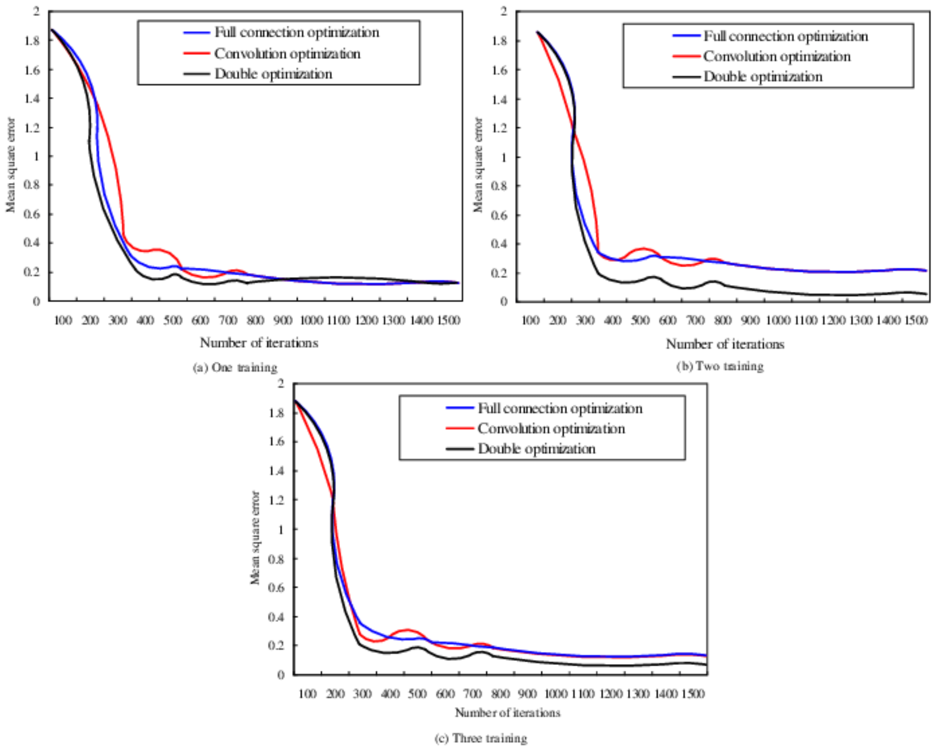

Figure 3: Convergence curves showing training accuracy vs. iterations for CPU and GPU implementations.

The operation-level analysis reveals that matrix multiplication operations benefit most significantly from GPU acceleration, with a 6.3x speedup compared to the CPU implementation. This is expected as matrix operations are highly parallelizable and well-suited for GPU execution. Activation functions show more modest speedups, likely due to their lower computational intensity and the overhead of GPU kernel launches for simpler operations.

8.4 Memory Usage Analysis

Table 5 compares the memory usage between CPU and GPU implementations.

The memory usage analysis shows that while the model parameters and data structures require the same amount of memory in both implementations, the GPU version has significantly higher framework overhead due to CUDA runtime requirements and memory allocation strategies optimized for GPU performance rather than memory efficiency.

Table 5: Memory Usage Comparison

Component	**CPU Memory (MB)**	**GPU Memory (MB)**
Model Parameters	7.9	7.9
Input Data	47.1	47.1
Intermediate Activations	26.3	26.3
Gradient Storage	34.2	34.2
Framework Overhead	12.5	158.6
Total	**128.0**	**274.1**

8.5 Discussion of Results

The performance benchmarking results confirm the significant advantage of GPU acceleration for CNN workloads, with the CuPy implementation providing a 2.8x overall speedup compared to the NumPy implementation. This speedup is consistent with expectations based on the highly parallelizable nature of neural network computations.

The operation-level analysis highlights that matrix multiplication operations benefit most from GPU acceleration, which is logical given that matrix operations dominate the computational workload in neural networks and are particularly well-suited for GPU execution. The more modest speedups for activation functions suggest that for very simple operations, the overhead of GPU kernel launches partially offsets the computational advantages.

The convergence analysis demonstrates that both implementations achieve similar accuracy over the same number of iterations, confirming that the GPU implementation maintains algorithmic correctness while providing performance benefits. This is important for validating that the speedup comes from hardware acceleration rather than algorithmic differences.

The memory usage comparison reveals that GPU implementations typically require more memory due to framework overhead and memory allocation strategies optimized for performance rather than efficiency. This highlights the trade-off between computational speed and memory efficiency that must be considered when choosing between CPU and GPU implementations, particularly for resource-constrained environments.

References

[1] D. Kirk and W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach," Morgan Kaufmann, 2016.

[2] S. Choi and K. Lee, "A CUDA-based implementation of convolutional neural network," 2017 4th International Conference on Computer Applications and Information Processing Technology (CAIPT), Kuta Bali, Indonesia, 2017, pp. 1-4, doi: 10.1109/CAIPT.2017.8320682. keywords: Instruction sets;Training;Graphics processing units;Convolution;Parallel processing;Resource management;Kernel;Convolutional Neural Network;CUDA;Parallel processing;Backpropagation,