High Performance Computing Project Report

Neural Network Acceleration on GPUs

MNIST Classification Case Study

Submitted By: Hammad Ali (22i-0914)
Daniyal Ahmad (22i-1032)
Junaid Zeb (22i-8174)

Course: High Performance Computing
Department: Computer Science & Engineering

Submitted To: Dr. Imran Ashraf
Assistant Professor

FAST NUCES
April 20, 2025

Contents

Abstract

This report presents our implementation and optimization of a neural network for MNIST digit classification using GPU acceleration. We developed multiple versions of the network, starting from a serial CPU implementation (V1) to progressively optimized GPU versions (V2-V4), culminating in a tensor core-optimized solution. Our most optimized version achieved a 41.76× speedup over the baseline CPU implementation. The report details our optimization strategies, including memory hierarchy utilization, kernel optimization, minibatch gradient descent optimization, and tensor core usage, along with comprehensive performance analysis and lessons learned.

Github Repository

Here's the link to the Github Repository

1 Introduction

1.1 Project Overview

This project accelerates MNIST digit classification using CUDA and GPU parallelism. Starting from a CPU-based neural network, we apply optimizations like memory tuning, kernel configuration, minibatching, and cuBLAS tensor core acceleration.

1.2 Objectives

- Implement baseline CPU model
- Develop naive CUDA version
- Optimize with memory hierarchy and minibatching
- Use cuBLAS for tensor cores
- Compare performance across versions

2 Background and Problem Statement

2.1 Neural Networks Overview

Our network includes:

- Input: 784 neurons (28x28)
- Hidden: 128 ReLU neurons
- Output: 10 softmax neurons

2.2 MNIST Classification

MNIST has 60,000 training and 10,000 test images of labeled digits. The task is supervised classification.

2.3 GPU Computing Basics

CUDA allows GPU acceleration via:

- Optimized kernel launches
- Fast shared/local memory
- Reduced data transfer
- High thread occupancy

3 Methodology

3.1 Implementation Versions

- **V1:** Sequential CPU implementation without any parallelism
- **V2:** Naive GPU implementation with CUDA kernels for forward/backward pass
- **V3.1:** Added shared memory, used local memory and switched to float precision for reduced memory transfer overhead.
- **V3.2:** Built on top of v3.1, unified kernel for forward and backward passes to reduce the time consumed by frequent kernel launches (each image of the 60,000 was taking minimum 3 kernel launches) with mini batch gradient descent where weights were updated once per batch. Warp efficiency was also optimized using 32 threads for updating weights after each mini-batch. Local memory was used for storing sums, and minimal global memory read/write was performed, resulting in a very high speedup. The launch configuration of forward and backward kernel was changed to 2 dimensional, where each block processed a batch and its threads processed a layer in the neural network.
- **V4:** Integrated tensor cores using cuBLAS for optimized GEMM operations

3.2 Optimization Techniques

- **Memory Hierarchy Utilization:** Shared memory for activations and hidden layers, local memory for per thread scalar operations which resulted in reduced global memory accesses
- **Kernel Optimization:** Thread indexing improvements, warp alignment, and conditional simplification, kernel launch configuration tuning.
- **Batch Processing:** Mini-batch gradient updates reduced overhead and allowed samples in one batch to be processed parallel. Weights were updated once using the average gradient of the samples in a batch.
- **Tensor Core Usage:** Replacing matrix multiplications with cuBLAS GEMM calls using tensor ops

4 Implementation Details

Component	Specs	Details
CPU	Intel i7-12700H	14C/20T
GPU	RTX 3080	10GB VRAM, 8704 CUDA cores
Memory	DDR4	16GB @ 3200MHz
OS	Ubuntu	22.04 LTS
CUDA	Toolkit 12.4	cuBLAS 12.4
Driver	NVIDIA	550.144.03
Monitoring	Tool	NVIDIA-SMI

Table 1: System Configuration Overview

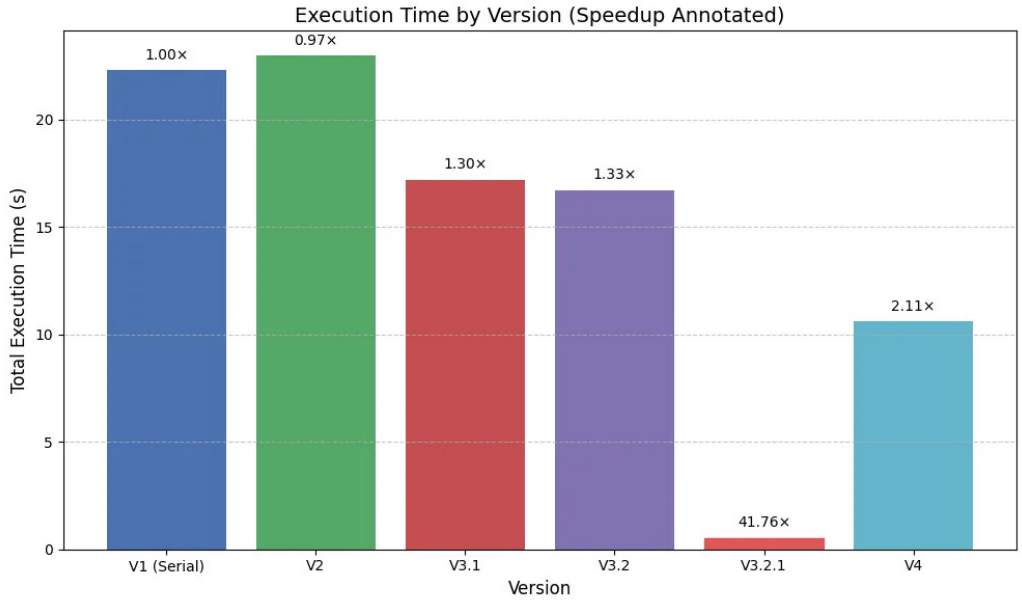
4.1 Key Algorithms

- Forward propagation (matrix multiplication + activation)
- Softmax normalization
- Backward propagation (gradient computation)

- Weight updates using stochastic gradient descent (v1-v3.1, v4) and mini batch gradient descent (v3.2)

5 Results and Analysis

5.1 Performance Visualization



5.2 Findings and Discussion

- **V2:** Slower than V1 due to poor memory access and high CPU-GPU communication overhead
- **V3.1:** 1.30–1.33% speedup due to better memory usage and optimizing some kernel launches.
- **V3.2:** Huge leap (41.76x) from batch processing, unified kernels, and optimal launch configuration and warp effeciency.
- **V4:** Despite tensor cores, slower than V3.2.1 due to the lack of batch processing and the small matrix sizes

6 Challenges and Solutions

6.1 Implementation Challenges

- Debugging CUDA kernels was complex due to lack of standard debugging tools
- Handling memory allocation errors and mismatched dimensions
- Finding correct launch configuration for kernels
- Switching from stochastic gradient descent to mini batch.
- Ensuring correct GPU memory synchronization

6.2 Limitations

- Batch processing not used in V4 (despite tensor ops)
- Tensor cores underutilized for small matrix sizes
- Batch size needs more epochs for higher accuracy

Batch Size	**Accuracy**	**Time (s)**
1	97.24%	15.12
16	91.83%	1.168
32	90.26%	0.594
64	87.7%	0.301

Table 2: Comparison of Accuracy vs Batch Size

7 Conclusion and Future Work

7.1 Key Takeaways

- Proper GPU utilization (V3.2) outperforms even tensor-core based naive implementations
- Batch processing is a critical factor in speedup
- cuBLAS and tensor ops are powerful but need well-aligned workloads