
# Neural Network Acceleration on GPUs: MNIST Classification Case Study

**Report By:**

Ubaida Tariq (22i-1155)

Muhammad Khan (22i-1040)

Rihab Rabbani (22i-1345)

# 1. Introduction

This report documents our implementation and optimization of a neural network for MNIST digit classification using GPU acceleration. The MNIST dataset consists of 70,000 grayscale images (28×28 pixels) of handwritten digits (0-9). Our goal was to progressively optimize the neural network implementation from a baseline CPU version to highly optimized GPU versions using various CUDA techniques.

The neural network architecture consists of:

**Input layer:** 784 neurons (one per pixel)

**Hidden layer:** 128 neurons (256 in later versions)

**Output layer:** 10 neurons (one per digit class)

# 2. Baseline Implementation (V1)

### Overview

The baseline implementation (V1) is a sequential CPU implementation of a fully connected neural network with one hidden layer. Key characteristics:

**Network Architecture:** 784-128-10 (input-hidden-output)

**Activation Functions:** ReLU for hidden layer, Softmax for output

**Training:** Stochastic Gradient Descent with backpropagation

**Learning Rate:** 0.01

**Epochs**: 3

**Batch Size:** 64

**Implementation Details:**

- Uses standard C with no parallelization
- Matrix operations implemented with nested loops
- Memory allocated dynamically for weights and biases
- Forward and backward passes computed sequentially

**Performance**

- Training Time: 18 seconds
- Test Accuracy: ~92-94% (varies with random initialization)

**Limitations**

- No utilization of parallel processing capabilities
- All computations performed sequentially on CPU
- Memory operations not optimized
- No hardware-specific optimizations

## 3. Naive GPU Implementation (V2)

**Overview**

V2 represents our initial port of the neural network to GPU using CUDA. This version maintains the same network architecture but parallelism key operations.

**Key Modifications**

**CUDA Kernel Implementation:**

- Parallelized matrix operations (matrix-vector products)
- Separate kernels for forward/backward passes
- Element-wise operations (ReLU, Softmax) parallelized

**Memory Management:**

- Host-device memory transfers for weights and activations
- Flattened matrix storage for coalesced memory access

**Basic Optimization:**

- Xavier initialization for better convergence
- Error checking with CHECK_CUDA_ERROR macro

**Implementation Challenges**

- Initial difficulty with memory management between host and device
- Ensuring correct synchronization between kernels
- Debugging race conditions in parallel implementations

**Performance**

- Training Time: 35.23 seconds
- Slowdown Factor: ~1.96x slower than V1

**Analysis**

The naive implementation was actually slower than the CPU version due to:

- Excessive host-device memory transfers
- Suboptimal kernel launch configurations
- Lack of memory hierarchy utilization
- Overhead of small kernel launches

## 4. Optimized GPU Implementation (V3)

**Overview**

V3 introduces significant optimizations to address the shortcomings of V2 while maintaining numerical accuracy.

**Key Optimizations**

**Memory Optimizations:**

- Pinned memory for faster host-device transfers
- Batch processing to reduce memory transfer overhead
- Shared memory utilization in key kernels

**Kernel Optimization:**

- Tiled matrix multiplication (TILE_WIDTH = 16)
- Optimized launch configurations (block/grid dimensions)
- Parallel reduction for Softmax computation
- Fused operations where possible

**Training Improvements:**

- Momentum (0.9) for faster convergence
- Weight decay (0.0001) for regularization
- Learning rate scheduling

**Numerical Stability:**

- Improved Softmax implementation with max subtraction
- Careful handling of floating-point operations

**Implementation Details**

- Network Architecture: 784-256-10 (larger hidden layer)
- Batch Processing: More efficient memory access patterns
- Streams and Events: For asynchronous operations and timing
- Occupancy Optimization: Adjusted block sizes for better GPU utilization

**Performance**

- Training Time: 8.62 seconds (V3), 21.1 seconds (V3-accurate)
- Speedup: ~2.09x faster than V1 (V3), ~0.85x (V3-accurate)

**V3 vs V3-accurate**

The faster V3 version makes some numerical approximations that **slightly reduce accuracy**, while **V3-accurate maintains full precision at the cost of speed**.

## 5. Tensor Core Implementation (V4)

**Overview**

V4 builds upon V3's optimizations while leveraging Tensor Cores for mixed-precision computation.

**Key Enhancements**

**FP16 Computation:**

- Weights and activations stored as FP16
- Mixed-precision training with FP32 accumulation
- Custom FP16 arithmetic operations

**Tensor Core Utilization:**

- Matrix operations optimized for Tensor Cores
- Specialized kernels for FP16 computation
- Efficient memory access patterns for Tensor Cores

**Additional Optimizations:**

- Enhanced memory hierarchy usage
- Improved occupancy with better thread block configurations
- Reduced memory footprint with FP16 storage

**Implementation Details**

Performance

- **Training Time**: 4.23 seconds
- **Speedup:** ~4.25x faster than V1

## 5. OpenACC-Enhanced Implementation (V5)

**Overview**

V5 builds upon V3's CUDA optimizations while incorporating OpenACC directives to further accelerate data preparation and host-side computations. This hybrid approach combines the best of both programming models.

**Key Enhancements**

**Hybrid Parallelization:**

- CUDA for compute-intensive neural network operations
- OpenACC for parallelizing data preprocessing and memory operations

**Memory Optimizations:**

- Pinned memory for both CUDA and OpenACC access
- Optimized data transfers between host and device
- Unified memory management between CUDA and OpenACC regions

**Compute Optimizations:**

- OpenACC parallelization of data loading and batch preparation
- SIMD vectorization for host-side computations

- Asynchronous execution of data preparation and GPU computations

**Implementation Details**

- **Network Architecture:** 784-256-10 (same as V3/V4)
- **Batch Size:** 128 (increased from 64 for better throughput)

**Parallel Regions:**

- Data loading and normalization
- Batch preparation
- Shuffling and index management
- Accuracy evaluation

**Performance**

- **Training Time:** 6.11 seconds
- **Speedup:** ~2.95x faster than V1, 1.42x faster than V3
- **Accuracy:** ~93% (maintained full precision)

**6. Performance Analysis**

**Training Time Comparison**

| Version | Time (seconds) | Speedup vs V1 | Accuracy |
|---|---|---|---|
| V1 (CPU) | 18.00 | 1.00x | ~94% |
| V2 (Naive GPU) | 35.23 | 0.51x | ~94% |
| V3 | 8.62 | 2.09x | ~94% |
| V3-accurate | 21.10 | 0.85x | ~98% |
| V4 (FP16) | 4.23 | 4.25x | ~94% |
| V5 (OpenACC) | 6.11 | 2.95x | ~94% |

**Key Findings**

1. The naive GPU implementation (V2) was slower due to overheads
2. Proper optimizations (V3) yielded significant speedups
3. Tensor Cores (V4) provided the best performance
4. There's a trade-off between speed and accuracy in some versions
5. The hybrid OpenACC/CUDA approach (V5) provided a good balance between performance and accuracy
6. OpenACC effectively accelerated data preparation and host-side operations
7. V5 outperformed V3 while maintaining full precision (unlike V4's FP16)
8. The implementation showed that mixed-paradigm approaches can be effective

**Optimization Impact**

- **Kernel Optimization**: Proper launch configurations improved throughput
- **FP16 Computation**: Enabled Tensor Core usage for maximum speed
- **Hybrid Parallelization:** OpenACC complemented CUDA by accelerating host-side operations
- **Memory Optimizations:** Unified memory management reduced overhead
- **Batch Processing**: Larger batches improved throughput
- **Numerical Stability:** Maintained FP32 precision throughout

**7. Conclusion**

Through this project, we successfully accelerated a neural network through multiple optimization stages. The V5 implementation demonstrated that:

**Hybrid Approaches Work:** Combining CUDA and OpenACC can yield performance benefits while maintaining code simplicity for certain operations.
**Host-Side Optimization Matters:** Accelerating data preparation and memory operations provided measurable improvements.
**Precision-Speed Trade-offs:** V5 showed that near-TensorCore performance could be achieved while maintaining FP32 precision.

Future work could explore:

- Deeper integration of OpenACC and CUDA
- OpenACC for additional operations like evaluation
- Multi-GPU support through OpenACC directives
- Automated tuning of gang/vector parameters

This project provided valuable hands-on experience with GPU programming, performance optimization, and the practical challenges of accelerating machine learning workloads.