Neural Network Optimization Report

MNIST Classification Performance Analysis

Team Members:

- Muhammad Owais (22i-1123)
- Muhammad Mahad Khan (22i-1028)
- Hassan Waqar (22i-1192)

Table of Contents

---

1. Introduction

This report evaluates four implementations of a neural network for MNIST classification, focusing on GPU-based optimizations. The versions analyzed are:

- **Version 1 (V1)**: Sequential CPU implementation.
- **Version 2 (V2)**: Naive GPU implementation using CUDA.
- **Version 3 (V3)**: Optimized GPU implementation with batch processing.
- **Version 4 (V4)**: Advanced GPU implementation with tensor cores and cuBLAS.

We compare execution time, accuracy, and optimization strategies, with speedups calculated relative to the previous version.

---

2. Neural Network Architecture

- **Input Layer**: 784 neurons (28×28 pixel images).
- **Hidden Layer**: 128 neurons (ReLU activation).
- **Output Layer**: 10 neurons (softmax activation).
- **Training Parameters**:
  - Learning rate: 0.01 (V1-V3); 0.1 (V4).
  - Epochs: 3.
  - Batch size: 64 (V1, V2, per-sample); 128 (V3); 256 (V4).

---

3. Version 1: Sequential Implementation

3.1 Implementation Details

A CPU-based implementation in C:

- **Matrix Operations**: Sequential nested loops.
- **Memory**: Dynamically allocated 2D arrays for weights, 1D for biases.
- **Forward/Backward Pass**: Sequential matrix multiplication, ReLU, softmax, and gradient updates.

3.2 Workflow and Issues

Workflow:

1. Load MNIST data.
2. Initialize weights/biases.
3. Per-sample: Forward pass, compute loss, backward pass, update weights.
4. Evaluate on test set.

Issues:

- Slow execution (48.023s for 3 epochs).
- High memory allocation overhead.
- No parallelism.

3.3 Bottlenecks

- No parallelism, leading to high latency.
- Inefficient memory access patterns.
- Poor scalability for larger datasets.

---

4. Version 2: Naive GPU Implementation

4.1 Optimization Approach

Objective: Transition computations to GPU.

Techniques:

- **Kernel Fusion**: Two kernels (fused_forward_kernel, fused_backward_kernel) for matrix operations, activations, and updates.
- **CUDA Parallelism**: GPU threads for matrix operations.
- **Device Storage**: Weights/biases on GPU.

4.2 Implementation Details

A CUDA-based implementation:

- **Kernels**:
  - fused_forward_kernel: Matrix multiplication, ReLU, softmax (1D blocks, 128 threads).
  - fused_backward_kernel: Gradient computation, weight updates (2D blocks, 32×32).
- **Memory**: Parameters on host and device; per-sample input transfers.
- **Synchronization**: __syncthreads() within blocks, cudaDeviceSynchronize() for host-device.

4.3 Workflow and Issues

Workflow:

1. Load MNIST data.
2. Initialize weights/biases, copy to device.
3. Per-sample: Transfer input/target, run forward/backward kernels, synchronize.
4. Evaluate on test set.

Issues:

- Per-sample transfers cause high overhead (~6.26s/epoch).
- Limited GPU utilization.
- Non-coalesced memory access.
- Frequent synchronization.

4.4 Bottlenecks

- **Data Transfers**: Per-sample host-device transfers dominate (18.789s total).
- **GPU Underutilization**: Single-sample processing limits parallelism.
- **Double Precision**: Slows computation.
- **Synchronization Overhead**: Frequent cudaDeviceSynchronize().
- **Memory Access**: Non-coalesced accesses reduce efficiency.

Resolution: V3 introduces batch processing and memory optimizations.

---

5. Version 3: Optimized GPU Implementation

5.1 Optimization Approach

Objective: Address V2's data transfer and memory issues.

Techniques:

- **Batch Processing**: 128 samples/batch.
- **Memory Coalescing**: Row-major data for coalesced access.
- **Shared Memory**: Cache intermediate results.
- **Optimized Threads**: 2D blocks (16×16).
- **Reduced Synchronization**: Sync at batch boundaries.

5.2 Implementation Details

A CUDA implementation with batch processing:

- **Batch Processing**: 128 samples, reducing communication overhead.
- **Memory**: Weights/biases on device, row-major; pre-allocated batch buffers.
- **Kernels**:
    - Matrix multiplication with shared memory (2D blocks, 16×16).
    - relu_kernel, softmax_kernel, output_delta_kernel, hidden_delta_kernel, weight_update_kernel, bias_update_kernel.
- **Synchronization**: __syncthreads() in kernels, reduced cudaDeviceSynchronize().

5.3 Workflow and Issues

Workflow:

1. Load MNIST data.
2. Initialize weights/biases, allocate batch buffers.
3. Per epoch:
    - Shuffle data.
    - Process batches (128 samples): Copy inputs/labels, run forward/backward kernels, update weights.
    - Synchronize at epoch end.

4. Evaluate in batches.

Issues:

- Double precision slows computation.
- Batch size (128) requires frequent transfers.
- No tensor cores.
- Shared memory adds complexity.

5.4 Bottlenecks

- **Double Precision**: Limits speed.
- **Batch Size**: 128 samples still requires transfers (0.432s total).
- **No Tensor Cores**: Custom kernels lack tensor core acceleration.
- **Synchronization**: Required at batch boundaries.
- **Shared Memory Overhead**: Adds complexity and minor latency.

Resolution: V4 adopts single precision, tensor cores, and larger batches.

5.5 Improvements Over V2

- Batch processing reduces transfer overhead.
- Coalesced memory access improves bandwidth.
- Shared memory lowers global memory latency.
- Reduced synchronization via batch processing.

---

6. Version 4: Tensor Core Optimized Implementation

6.1 Optimization Approach

Objective: Maximize GPU performance.

Techniques:

- **Batch Processing**: 256 samples/batch.
- **Tensor Cores**: cuBLAS with TF32 precision.
- **Single Precision**: Reduces computation time.
- **CUDA Streams**: Four streams for overlapping operations.
- **Memory Coalescing**: Row-major aligned data.
- **Xavier Initialization**: Improves convergence.
- **Asynchronous Transfers**: cudaMemcpyAsync.

6.2 Implementation Details

An advanced CUDA implementation:

- **Batch Processing**: 256 samples.
- **cuBLAS**: Matrix multiplications with tensor cores (TF32).

- **Single Precision**: Faster computation.
- **Streams**: Four non-blocking streams.
- **Memory**: Flattened row-major weights/biases; pre-allocated buffers.
- **Kernels**:
    - cublasSgemm for matrix multiplications.
    - Custom: add_bias_kernel, apply_relu_inplace_kernel, calculate_output_delta_kernel, calculate_hidden_delta_relu_deriv_kernel, adjust_all_biases_kernel (1D blocks, 256 threads).
- **Synchronization**: Streams sync at epoch end.
- **Softmax**: Host-based with asynchronous transfers.

6.3 Workflow and Issues

Workflow:

1. Load MNIST data.
2. Initialize with Xavier weights, allocate buffers.
3. Per epoch:
    - Shuffle data.
    - Process batches (256 samples): Async copy, run forward/backward on streams, host-based softmax, update weights.
    - Synchronize streams at epoch end.
4. Evaluate in batches.

Issues:

- Host-based softmax adds overhead.
- Large batch buffers strain memory.
- Limited stream overlap.
- TF32 precision affects accuracy.

6.4 Bottlenecks

- **Host-Based Softmax**: Device-host transfers add overhead (0.188s total).
- **Stream Underutilization**: Limited computation-communication overlap.
- **Memory Allocation**: Large buffers (256 × 784) strain lower-end GPUs.
- **Custom Kernel Latency**: Minor overhead vs. full cuBLAS.
- **TF32 Tradeoff**: Lower accuracy (93.11%) due to precision and higher learning rate (0.1).

Resolution: Future versions could use device-based softmax and optimize streams.

6.5 Improvements Over V3

- Larger batch size (256) reduces transfers.
- Tensor cores via cuBLAS boost performance.
- Single precision lowers computation time.
- Streams enable concurrency.

- Coalesced memory access improves efficiency.

---

7. Performance Analysis

7.1 Execution Time Comparison

Version	Total Training Time (s)	Evaluation Time (s)	Speedup (Relative to Previous)	Test Accuracy (%)
V1 (Sequential)	48.023	~0.094	-	96.75
V2 (Naive GPU)	18.789	~0.037	~2.56x (vs. V1)	95.64
V3 (Optimized GPU)	0.432	~0.001	~43.49x (vs. V2)	95.01
V4 (Tensor Core Optimized)	0.188	~0.0004	~2.30x (vs. V3)	93.11

Analysis:

- V2: ~2.56x speedup over V1 via GPU parallelism, limited by per-sample transfers.
- V3: ~43.49x faster than V2 with batch processing and memory optimizations.
- V4: ~2.30x faster than V3 using tensor cores, single precision, and streams.
- Accuracy: V1 (96.75%), V2 (95.64%), V3 (95.01%), V4 (93.11%)—lower in V4 due to TF32 and higher learning rate.

---

8. Conclusion

This project evolved a neural network for MNIST classification from a sequential CPU baseline (V1, 48.023s) to advanced GPU implementations:

- **V2**: Introduced GPU parallelism (18.789s, ~2.56x speedup vs. V1), limited by data transfers.
- **V3**: Added batch processing and memory optimizations (0.432s, ~43.49x vs. V2).
- **V4**: Leveraged tensor cores, single precision, and streams (0.188s, ~2.30x vs. V3), but with lower accuracy (93.11%).

The progression highlights the impact of GPU optimizations, though V4's accuracy trade-off suggests a need to balance speed and stability.

Github Link: https://github.com/Mahad811/HPC_Project.git