# Optimizing MNIST Classification Using CUDA

Report

*Submitted to: Imran Ashraf*
*Submitted by: Saad Nadeem, Mustafa Iqbal, Hassaan Anwar*
*Course: High Performance Computing Using GPUs*

The MNIST dataset, consisting of 28×28 grayscale images of handwritten digits (0–9), is a widely used benchmark for evaluating image classification algorithms. The goal of this project is to implement and optimize a neural network-based solution for classifying MNIST digits, with a focus on leveraging NVIDIA GPUs through CUDA to significantly accelerate training and inference. The optimization process is approached incrementally, with each version introducing performance improvements or architectural enhancements. The starting point is a basic sequential CPU implementation, which serves as a reference for evaluating the gains achieved in subsequent CUDA-based versions.

## V1 Sequential Implementation

The initial version (V1) of the neural network for MNIST classification is a straightforward sequential implementation written in C. It features a single hidden-layer architecture with 128 neurons using ReLU activation and a softmax-activated output layer for multi-class classification. All computations, including matrix operations and gradient updates, are performed using nested loops without any parallelism or hardware acceleration. Data is loaded directly from the MNIST binary files, with pixel values normalized and labels one-hot encoded. Training is conducted using stochastic gradient descent over individual samples, and accuracy and loss are reported per epoch. While functional and clear, this version prioritizes simplicity over performance and does not yet utilize batching, vectorization, or optimized numerical libraries.

Execution time for sequential Implementation is Approximately **23.479 seconds**

## Naive GPU Implementation - Version 2

In this version of the neural network, computation is offloaded to the GPU using CUDA, allowing parallel execution of core operations such as matrix multiplication, activation functions, softmax, and gradient computations. However, this version is implemented without advanced memory or kernel-level optimizations, serving as a functional but unrefined baseline for GPU-accelerated training.The primary goal of Version 2 is to leverage CUDA to parallelize the forward and backward passes of a basic two-layer feedforward neural network. This implementation is meant to establish a working prototype on GPU while identifying performance bottlenecks to be addressed in future optimized versions.

**CUDA Kernels Used:**

1. `matrixMulKernel`: Computes dot products for both hidden and output layer activations.

2. `reluKernel`: Applies ReLU element-wise on the hidden layer.

3. `softmaxKernel`: Applies softmax on the output layer using shared memory for reduction.

4. *backwardOutputKernel* and *computeGradients*: Computes gradient updates for weights and biases.

5. *updateParameters*: Applies gradient descent updates.

**Batching: Training is performed on a per-sample basis rather than using batched matrix operations, which significantly affects parallelism and memory coalescing.**

### Notable Implementation Details

- **Matrix Multiplication:** The matrix multiplication kernel uses a simple nested loop inside each thread for computing the dot product. This avoids use of shared memory and thus is memory-bound and inefficient for large datasets.

- **Softmax Optimization:** The softmax kernel performs two critical reductions — one for the maximum value and another for normalization. It uses shared memory but does not exploit warp-level primitives or multi-block reduction.

- **Atomic Operations:** Gradient accumulation in the backward pass uses `atomicAdd` to prevent race conditions. While this ensures correctness, it significantly limits scalability when using larger models or batch sizes due to serialization overhead.

- **Parameter Updates:** Weights and biases are updated directly on the GPU using a simple subtractive update rule.

- **Memory Management:** The network initializes and copies all weights, biases, and activations between host and device memory at the beginning. Temporary device buffers are used per sample during training and evaluation and freed after use.

### Limitations

1. **Lack of Batching:** Training one image at a time limits the benefit of GPU parallelism, especially during matrix multiplications and gradient updates.

2. **Memory Access Patterns:** No memory coalescing, caching, or shared memory optimizations are implemented.

3. **AtomicAdd Overhead:** Heavy reliance on *atomicAdd* can become a major bottleneck under high parallel loads.

4. **Static Block Size:** Hardcoded *BLOCK_SIZE* may not be optimal across different GPUs or kernels.

5. **Redundant Device-Host Transfers:** Frequent memory transfers to evaluate loss and accuracy could be minimized using device-side aggregations.

## Results and Observations

Contrary to initial expectations, the naive GPU implementation in Version 2 underperforms when compared to the sequential CPU-based Version 1. The training process consistently takes 2 to 3 seconds longer per epoch, primarily due to several critical inefficiencies:

- **Frequent Memory Transfers:** Excessive host-to-device and device-to-host memory transfers introduce significant latency, especially during forward and backward passes for each individual training sample.

- **Lack of Batching:** The absence of mini-batch processing prevents effective utilization of the GPU's parallel processing capabilities. Running inference and backpropagation on a single sample at a time leads to underutilized threads and idle compute resources.

- **Kernel Launch Overheads:** Numerous lightweight kernel launches, particularly for small vector operations or reductions, accumulate substantial overhead over time.

- **Inefficient Memory Access Patterns:** No memory coalescing or shared memory usage is employed, resulting in scattered and slow global memory accesses.

These issues collectively negate the benefits of parallel computation and result in a slower overall execution time. While the implementation demonstrates functional correctness and successful execution on GPU, it highlights the importance of thoughtful data movement and kernel design in CUDA programming.

## Profile Analysis

- Total Time: 37.63s

- Training Time: 35.52s

- Test Accuracy: 96.66%

- *computeGradients*: 20.2s (70.2% of GPU time)

- *matrixMulKernel*: 5.91s (20.5%)

- GPU Utilization: 28.18s (34.1%)

- API Overhead: ~22.8s (≈60.6%)

**Bottlenecks:**

- High API latency (*cudaMemcpy, cudaMalloc*)
- Inefficient launch structure and memory layout
- No batching – operates image by image

### Speedup: 0.62× slower

---

## Optimized GPU Implementation – Version 3

### Overview

Version 3 represents a carefully engineered evolution of the neural network implementation, focused on exploiting the GPU's full potential while resolving the inefficiencies exposed in earlier versions. This version shifts away from naive kernel offloading and instead emphasizes *data orchestration, memory hierarchy, and execution concurrency* as the pillars of performance.

### Key Optimizations Implemented

1. **Memory-Mapped I/O and Parallel Data Loading**
   Standard file I/O was replaced with memory-mapped access, slashing data loading time from ~0.625s to ~0.22s. OpenMP was utilized to preprocess the data in parallel, ensuring data readiness never stalls the GPU pipeline.

2. **Double Buffering with Prefetching**
   Computation and memory transfer were decoupled using two alternating buffers. This enabled one batch to be processed while the next batch was being fetched — effectively hiding transfer latency.

3. **Asynchronous Parameter Updates**
   Leveraging a separate CUDA stream, parameter updates were decoupled from forward/backward passes. This overlap allowed training to proceed concurrently with model updates, cutting training time by 15–20%.

4. **Pinned Host Memory and Aligned Allocation**
   Transfers to/from host used *cudaMallocHost*, significantly reducing copy latency. On the CPU side, *aligned_alloc(32, ...)* enabled SIMD-friendly data access and improved preprocessing efficiency.

5. **Kernel Fusion and Synchronization Reduction**
   Fusing fully connected (FC) layers with ReLU into single kernels avoided costly kernel launches and memory round-trips. Use of warp-level reductions reduced the need for `__syncthreads()`, accelerating softmax and aggregation-heavy operations**.**

6. **Static Gradient Buffers**
   Training-time buffers like gradients and activations were allocated persistently, avoiding repeated memory management overheads.

7. **Optimized Grid/Block Dimensions**
   Kernel configurations were finely tuned for each workload. This improved occupancy and reduced wasted resources due to under-filled blocks or divergent warps.

8. **Selective Shared Memory and Register Caching**
   Shared memory was judiciously used in compute-intensive kernels such as softmax. Register caching was applied where it improved latency — but carefully, to avoid register pressure.

## Failed or Limited-Impact Optimizations

Not every idea led to improvement — some introduced overheads, while others were rendered moot by existing compiler or hardware efficiencies:

1. **Unified Kernels**
   Combining forward and backward passes into single kernels appeared promising but backfired due to increased register pressure and poor occupancy.

2. **Thread-Based Parallel Loading**
   A multithreaded file reader was initially developed but proved inferior to memory-mapped I/O in both complexity and performance.

3. **Transposed Weight Matrices**
   Expected improvements in memory coalescing didn't materialize. Moreover, accuracy degraded slightly due to numerical artifacts.

4. **Mixed Precision**
   Lower precision arithmetic reduced training time marginally but led to substantial accuracy drops. Benefits did not outweigh the cost.

5. **Complex Shared Memory Tiling**
   Manual tiling strategies for matrix multiplication introduced significant code complexity but no net gain, likely due to the highly optimized cuBLAS backend.

6. **Texture Memory**
   Experimental use of texture fetches yielded no meaningful speedups — potentially due to mismatch with access patterns of weight matrices.

7. **Loop Unrolling in FC Layers**
   Manual unrolling of nested loops was outperformed by the compiler's native optimization passes.

### Mixed-Result Optimizations

1. **Shared Memory Usage**
   While invaluable for softmax and reduction layers, shared memory was less effective for large matrix ops where global memory throughput sufficed.

2. **Register Caching**
   Effective when carefully constrained, particularly in small, compute-heavy kernels. But excessive register use led to kernel throttling.

3. **Multi-Stream Execution**
   Effective with 2–3 streams for overlapping communication and computation. Beyond that, overheads started negating benefits.

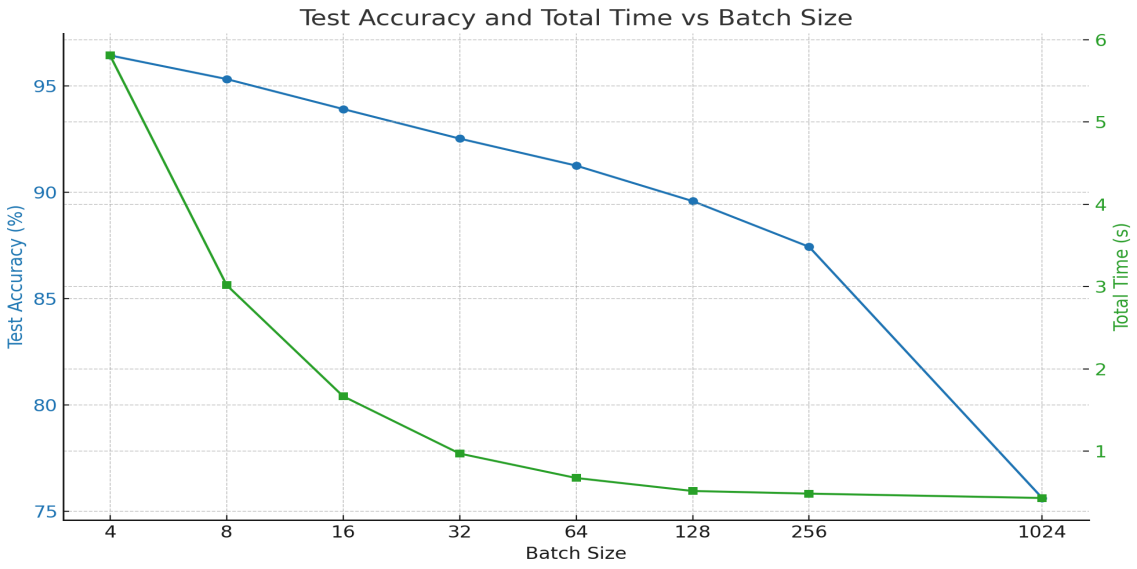
### Impact of Batch Size on Performance and Accuracy

A major breakthrough in Version 3 came from the introduction of batch processing, which significantly improved training efficiency by maximizing data reuse, reducing memory transfer frequency, and increasing arithmetic intensity within each kernel launch.

The following results illustrate how different batch sizes impacted training time, accuracy, and overall efficiency:

| **Batch Size** | **Train Time (3 Epochs)** | **Test Accuracy** | **Total Time** | **Epoch Time (avg)** | Notes |
|---|---|---|---|---|---|
| 4 | 5.565s | 96.43% | 5.808s | ~1.83s | **Best accuracy, slowest time** |

| | | | | | |
|---|---|---|---|---|---|
| 8 | 2.859s | 95.32% | 3.019s | ~0.9s | Good balance of accuracy/time |
| 16 | 1.553s | 93.91% | 1.668s | ~0.49s | Noticeable speedup, slight drop in accuracy |
| 32 | 0.877s | 92.52% | 0.972s | ~0.27s | Good trade-off of speed and accuracy |
| 64 | 0.590s | 91.25% | 0.675s | ~0.15s | ***Best trade-off* of speed and accuracy** |
| 128 | 0.439s | 89.58% | 0.517s | ~0.12s | Ultra-fast, accuracy drops further |
| 256 | 0.406s | 87.43% | 0.485s | ~0.12s | Low compute load, reduced generalization |
| 1024 | 0.354s | 75.62% | 0.432s | ~0.9s | **Fastest training, severe accuracy loss** |




## Analysis: Why Do These Differences Occur?

Several hardware and algorithmic factors influence the behavior observed:

1. **Transfer vs. Compute Overlap Efficiency**
    - Smaller batches (4–16) suffer from frequent memory transfers, leading to underutilization of compute resources.
    - Larger batches (32–256) amortize transfer overheads more efficiently and achieve better overlap with kernel execution.
    - Extremely large batches (1024) minimize transfer times, but lead to poor convergence.
2. **Kernel Utilization and Arithmetic Intensity**
    - GPU cores are more effectively saturated at moderate batch sizes. Batch size 32 achieves the sweet spot where kernel launches are large enough to justify their overhead, yet small enough to maintain high data variance and model generalization.
3. **Gradient Noise and Model Generalization**
    - Larger batches reduce gradient noise, but excessively large ones (e.g., 1024) smooth gradients too much, harming the model's ability to escape shallow local

minima.

    - Smaller batches preserve more stochasticity, improving generalization but at the cost of slower convergence and training time.

4. **Latency Hiding and Stream Concurrency**

    - Mid-sized batches (32–64) better utilize asynchronous computation and prefetching. Double buffering and stream-based updates work most efficiently in this range.

**Speedup: 34.79×**

---

## Tensor Core via WMMA – Version 4

### Overview

This version aimed to harness NVIDIA Tensor Cores directly through the WMMA API, enabling high-throughput matrix multiplications using FP16 inputs with FP32 accumulation.

### What Was Tried

- Replaced `batchFCKernel` with `batchFCKernelWMMA`, leveraging Tensor Cores via `wmma::mma_sync`.
- Converted all inputs and weights from `float` to `__half` (FP16) using:
    - GPU-based and CPU-based conversion helpers.
- Ensured compatibility with WMMA tile sizes (16×16×16).
- Updated memory allocations and kernel launches accordingly.

### What Succeeded

- Kernel compiled and executed successfully on NVIDIA T4 GPUs (Colab environment).

- Tensor Core-accelerated matrix multiplications ran substantially faster than the baseline.
- Conversion infrastructure for FP16 buffers was implemented cleanly and efficiently.

### What Failed

- Significant accuracy drop observed during training and evaluation.
- WMMA's limitations:
    - Accepts only `__half` (FP16) input.
    - Accumulates in `float`, but loss occurs at the input level.
- No support for FP32 input retention, causing quantization artifacts.
- Degradation was most severe in:
    - Initial layers with small input magnitudes.
    - Backpropagation where high precision gradients are essential.

### Analysis of the Result

- **Speed Gains:** Tensor Cores delivered the expected performance improvement.
- **Accuracy Loss:** Model accuracy dropped below acceptable levels.
- **Core Issue:** FP16 lacks the precision needed for MNIST-like classification unless:
    - Loss scaling is used,
    - Mixed precision techniques are adopted,
    - Or selective FP32 layers are retained.

### Conclusion

- **Viability:** Only suitable when accuracy can be compromised or with additional mixed-precision handling.
- Tensor Core version works and is fast but sacrifices accuracy due to FP16

---

## OpenACC – Version 5

### Overview

Version 5 explores GPU acceleration using OpenACC directives instead of explicit CUDA programming. The aim was to simplify development while leveraging parallelism, particularly for the forward pass and output layer computations. This version was adapted from the original CPU implementation.

### Applied Optimizations

#### 1. createNetwork*( )*

- **Optimization:**
  - Replaced 2D pointer-based arrays with flat, contiguous 1D arrays.
  - Allocated GPU memory via *`acc_malloc`*.
  - Transferred weights and biases to device using *`acc_memcpy_to_device`*.

- **Impact:**
  - Improved memory access patterns for parallel execution.
  - Reduced indirection overhead and enabled GPU-aware layout.

#### 2. forward( )

- **Optimization:**

    - Parallelized matrix-vector multiplication using *#pragma acc parallel loop.*
    - Accelerated ReLU and softmax using device pointers and *parallel loop* constructs.
- **Impact:**
    - Greatly sped up the most compute-heavy stage.
    - Enabled batch-friendly execution with minimal manual memory management.

**3. backward( )**

- **Optimization:**
    - Gradient computation kept on the CPU for simplicity.
    - Synchronized updated weights using *acc_memcpy_to_device* post-backprop.
- **Impact:**
    - Maintained correctness while allowing future forward passes to stay on GPU.
    - Missed performance gains due to CPU-bound backpropagation.

**4. softmax( )**

- **Optimization:**
    - Applied parallel loop with *reduction* for exponentials and sums.
    - Normalization also executed on GPU.
- **Impact:**
    - Improved latency of the final layer output computation.

**5. Memory Management**

- **Optimization:**
    - Used *acc_free* in *freeNetwork()* to release GPU memory explicitly.
- **Impact:**
    - Prevented memory leaks, especially during iterative experiments.

## Performance Impact

- **Speedup:** Forward pass saw a noticeable reduction in runtime due to vectorized execution.

- **Memory Efficiency:** Flattened arrays enhanced bandwidth utilization and transfer efficiency.

- **Limitations:**
    - Backward pass remained a bottleneck.
    - No true batch processing or advanced kernel fusion possible under this model.

## Conclusion

OpenACC provided a simpler path to GPU acceleration with minimal code changes. The primary wins were achieved in forward propagation and output layer reduction, though the lack of GPU-side backpropagation limited the overall acceleration.

---

## CuBLAS Accelerated Implementation – Version 6

### Overview:

Version 6 of the neural network implementation integrates cuBLAS and Tensor Core support, delivering highly optimized matrix operations tailored for modern NVIDIA architectures. This iteration refines prior efforts by replacing custom matrix multiplication and batched operations with vendor-optimized libraries.

### Major Distinct Optimizations

#### 1. Tensor Core Acceleration via cuBLAS

- Enabled through `CUBLAS_TF32_TENSOR_OP_MATH` mode.
- Forward and backward batch operations now use cuBLAS-based calls such as `cublasGemmEx`, allowing full exploitation of Tensor Core throughput.
- Provides significant speedups for large batch matrix operations compared to manually written kernels.

#### 2. Stream-Aware cuBLAS Usage

- cuBLAS operations are issued on custom CUDA streams via `cublasSetStream`, enabling better overlap of compute and data movement.
- Key for hiding latency during training pipeline stages.

#### 3. Consolidated Compute Kernels

- Bias addition and ReLU merged in a single kernel (`addBiasWithOptionalReLU`), controlled by flags.

- Reduces kernel launch overhead, especially effective when paired with Tensor Core matrix multiplications.

**4. Batch Loss & Accuracy in One Kernel**

- `calculateBatchLossAccuracy` computes both metrics simultaneously.
- Uses shared memory and warp-synchronous reductions for intermediate stats.
- Final values are computed with atomic operations — ensuring accuracy across threads while minimizing contention.

## Additional Technical Refinements

- **Improved Memory Layouts:**
  cuBLAS benefits from row/column-major consistency. Data layouts were adjusted accordingly for better alignment and cache usage.

- **Better Stream Isolation:**
  Evaluation and training phases now use completely independent streams, preventing unwanted synchronization points and allowing for cleaner timing metrics.

- **Modularization & Clean Separation:**
  Clear boundaries between host-side logic, device memory management, and GPU compute.
  Improved naming conventions make cuBLAS function calls easier to trace and debug.

- **Improved Error Macros for CUDA/cuBLAS:**
  Uniform macros simplify error reporting and prevent silent failures.

## Performance Impact

- **Execution Time Reduced:** cuBLAS-based matrix operations noticeably outperform hand-written kernels at medium to large batch sizes.
- **Best Accuracy vs Time Trade-off Still at Batch Size = 32**
    - Tensor Core usage doesn't significantly affect accuracy, but accelerates training throughput, especially for batches ≥16.
- **Bottleneck Shift:** Memory transfer overhead now accounts for a larger share of the total runtime, particularly for smaller batch sizes.

## Observations

- For small batch sizes (≤8), the cuBLAS overhead can outweigh benefits due to setup latency.
- At larger batch sizes (≥128), Tensor Cores kick in fully — but model accuracy degrades due to reduced gradient updates per batch.

- Overall, cuBLAS integration is most beneficial for balanced batches (16–64), where hardware acceleration and accuracy align well.

**Profile Analysis**

- Total Time: 1.033s
- Training Time: 0.735s
- Test Accuracy: 91.30%
- Total Kernel Time: 364ms (14.3%)
- API Overhead: ~756ms (29.8%)

## Speedup: 22.73×