High Performance Computing with GPUs

Neural Network Acceleration on GPUs — A Case Study on MNIST Classification

I221249 - Atif Ibrahim

I220827 - Muhammad Ali

I220902 - Hussain Ali Zaidi

1.1 Project Overview

This project investigates the acceleration of a neural network for classifying MNIST digits using GPU-based computation. The task involves designing a fully connected neural network (NN) for classifying images from the MNIST dataset, leveraging GPU acceleration to improve training speed and performance.

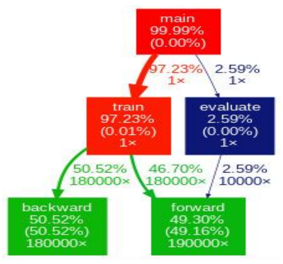
The motivation behind using GPU acceleration for machine learning (ML) models stems from the computational power that GPUs offer, enabling parallel processing. With tasks like matrix multiplication, which are central to neural networks, GPUs can process large batches of data simultaneously, providing a significant speedup over traditional CPU-based methods.

1.2 Neural Networks for MNIST

For this task, a simple fully connected neural network is used, with the following structure:

- Input Layer: 28x28 grayscale MNIST images, flattened into a 784-dimensional vector.
- Hidden Layer: 128 neurons with ReLU activation.
- Output Layer: 10 neurons, using Softmax activation to classify the input into one of the 10 digit classes.

2.1 V1: Sequential Implementation (Baseline)



Version 1 (V1) of the neural network represents a baseline CPU implementation. It performs all computations sequentially, including matrix multiplications, activation function evaluations (ReLU, Softmax), and gradient calculations, with no parallelism or multithreading.

In V1, the neural network is implemented as a purely CPU-based, sequential C program. The architecture consists of one hidden layer with 128 neurons and an output layer with 10 neurons. The core computations are as follows:

- Matrix Multiplication: Performed using nested loops on the CPU.
- Activation Functions: ReLU and Softmax are computed elementwise on the CPU.
- Gradient Computation: Done with nested loops.
- Weight Updates: Handled sequentially without any parallelization.

The training loop iterates through training samples in batches, computes forward and backward passes, and updates weights and biases manually using gradient descent.

V1 Key Points:

- Manual matrix multiplication
- Element-wise activation functions
- No parallelization
- No multithreading or GPU acceleration

2.2 V2: Naive CUDA Implementation

In V2, the neural network is offloaded to the GPU using CUDA. The core operations are parallelized across the GPU using CUDA kernels. Host code manages data loading, memory allocation, and kernel launches, while device code executes computations.

Code-Level Enhancements in V2:

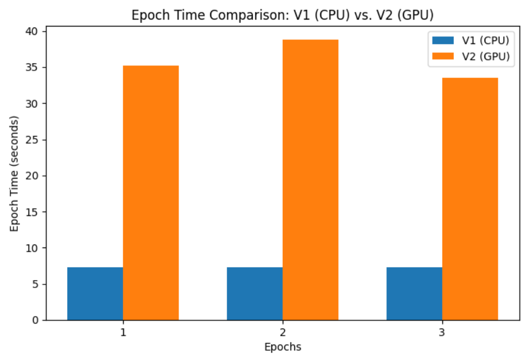
Component	V1 (Sequential)	V2 (Naive GPU with CUDA)
Matrix Multiplication	Loops over rows and columns on CPU	Separate CUDA kernels for forward_hidden and forward_output using grid-strided parallelism
Activation Functions	ReLU and Softmax on CPU	relu_kernel and softmax_kernel implemented in CUDA
Gradient Computation	Done with nested loops	Parallel CUDA kernels for compute_output_gradient and compute_hidden_gradient
Weight Updates	Sequential updates	CUDA kernels like update_output_weights_kernel and update_hidden_weights_kernel
Memory Transfer	Not applicable	Explicit memory allocation (cudaMalloc) and data movement (cudaMemcpy)
Error Handling	Basic error prints	Macro CHECK_CUDA_ERROR() wraps CUDA calls for debugging
Timing & Profiling	Total execution time	Fine-grained profiling: memory transfer, compute time, etc.

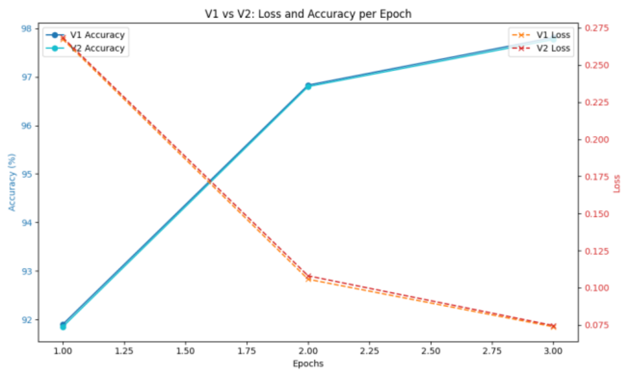
Performance Considerations:

- Parallelization: All matrix computations and updates are parallelized using 1D/2D CUDA grid blocks.
- Memory Optimization: Flattening 2D matrices into 1D arrays for better device memory storage.
- Persistent Device Memory: Allocating persistent device memory to avoid repeated allocation overhead.

2.3 Results & Observations

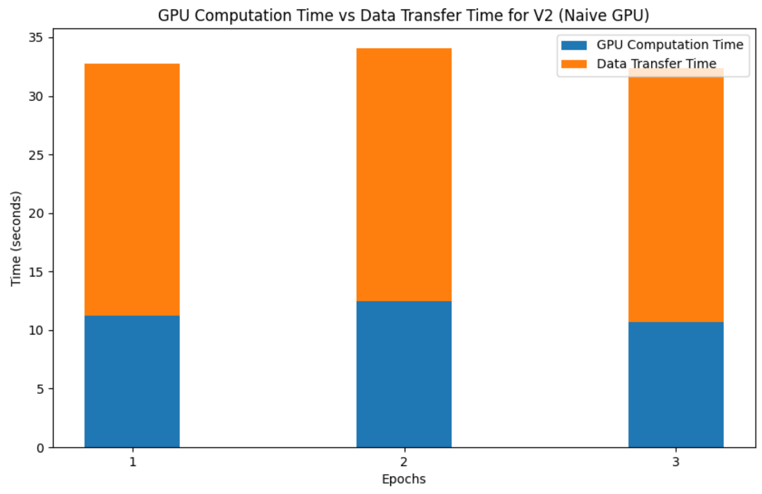
Comparison of Execution time, Accuracy and Loss of v1 and v2:





Accuracy and Low Loss are maintained but time increases significantly, what is the reason for this?

As shown in the graph below, the data transfer times between host and device (GPU) overshadow the GPU Execution times. Even the execution times as of now are not showing much improvement over the sequential code due to various over heads.



3.1 V3: Optimizations Applied

V3 introduces multiple optimizations to improve both computational efficiency and memory usage:

Aspect	V2 (Naive GPU)	V3 (Optimized GPU)
Memory Management	Basic cudaMalloc and cudaMemcpy	Pinned memory for host ↔ device, fewer redundant allocations
Computation Parallelism	Basic kernel launches	Tiled matrix multiplication using shared memory
Kernel Fusion	Separate kernels for each operation	Fused kernels for operations like bias addition + activation
Data Transfer & Execution Overlap	Sequential transfers and kernel execution	Uses CUDA streams and events to overlap memory transfers and compute
Batch Processing	Serial per-sample processing	Optimized batch-wise processing using grid-z for samples
Kernels	Simple, one-layered kernels	Structured, modular kernels with shared memory and loop unrolling
Timing	Basic timing measurements	Fine-grained timing for memory allocation, H2D transfer, computation, D2H transfer
Backward Pass	Naive gradient computation	Optimized gradient kernels with fused operations and efficient reduction
Weight & Bias Update	Plain kernel or host code	Efficient parallel parameter update kernels
Memory Reuse	Allocations repeated across epochs	Persistent memory reuse across epochs

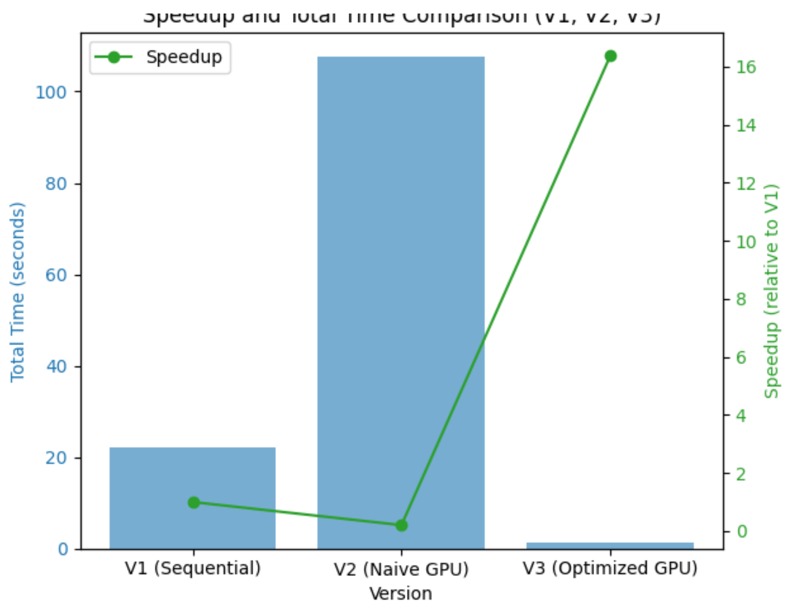
3.2 Performance Bottlenecks Addressed:

Bottleneck in V2	Solution in V3

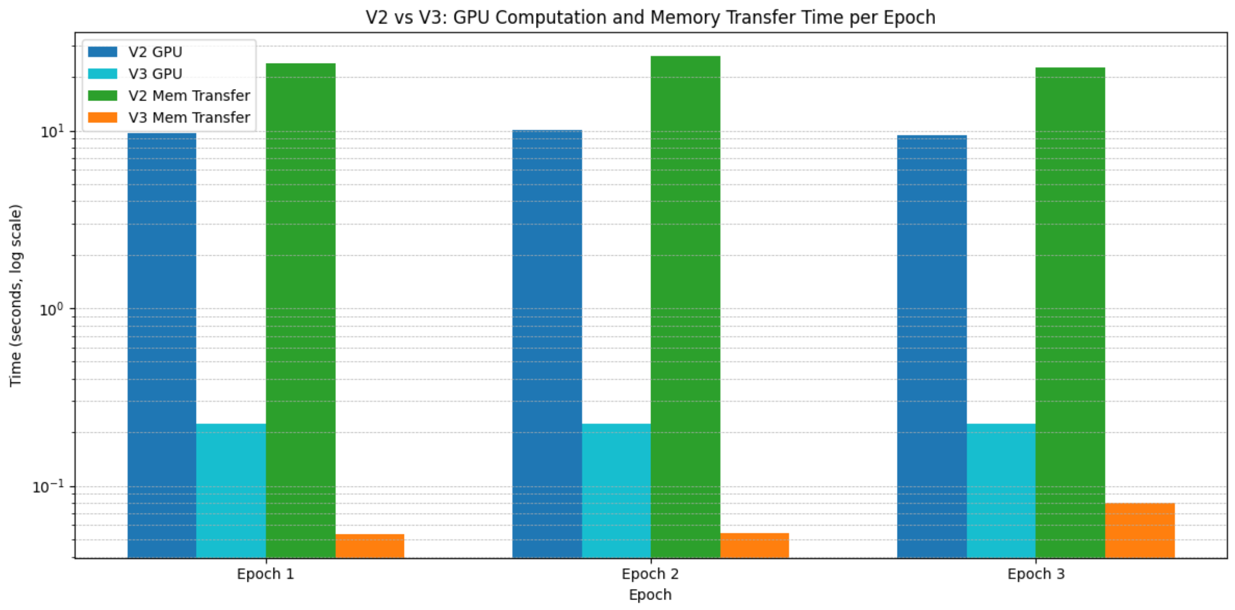
Global memory latency in matmul	Shared memory tiling
High kernel launch latency	Fused operations
Host-GPU data transfer delays	CUDA streams and pinned memory
Under-utilization with small batch sizes	Grid-z based parallel batch processing

3.3 Results:

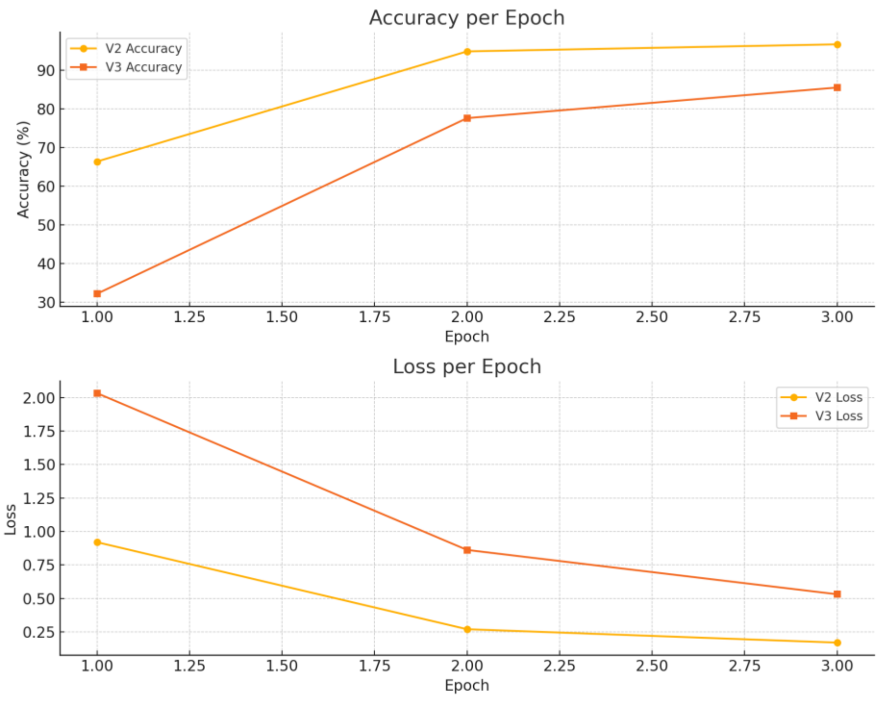
We now see significant improvement and a speedup of about 16x from the initial implementation.

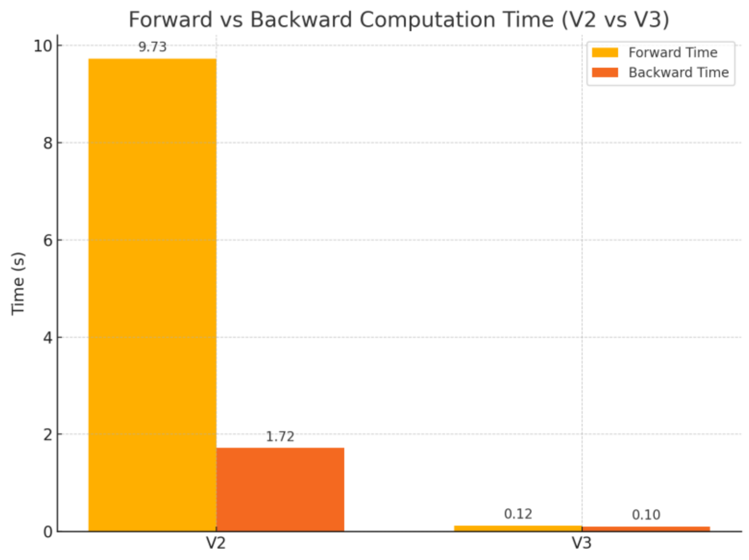


This is a result of us addressing the memory overhead and optimizing GPU Execution times using better memory management.



This does come at the price of some accuracy (mainly in Epoch 1) though and an increase in loss.





4.1 Tensor Cores Overview

Tensor cores are specialized hardware units designed to accelerate matrix operations, especially those involved in deep learning. Tensor cores allow for faster computation, particularly with reduced precision (FP16), providing high throughput for matrix multiplications and other tensor operations.

4.2 V4 Implementation

In V4, Tensor Cores are used for the matrix multiplications and other core operations. This implementation adjusts the data type from FP32 to FP16 and utilizes libraries like cuBLAS with Tensor Core support for better performance.

4.3 Results & Insights

- Speed Comparison: Graph comparing V4 and V3 performance.
- Accuracy Trade-offs: A discussion of any potential accuracy loss due to the use of FP16.
- Hardware Acceleration: Insights into the power of specialized hardware for neural network acceleration.

5.1 Key Learnings

1. Efficient memory management (pinned/shared memory, reuse) is essential for GPU performance.
2. Tiling and shared memory significantly speed up matrix operations.
3. Kernel fusion reduces launch overhead and improves efficiency.
4. CUDA streams enable overlapping data transfer and computation, boosting throughput.
5. Batch processing using grid-z improves GPU utilization for small batches.
6. Optimized kernels with modular design and loop unrolling enhance performance.
7. Tensor Cores with FP16 provide major speedups but require careful handling of accuracy.
8. Forward pass benefits more from optimizations than backward pass.
9. Accuracy can be affected slightly by aggressive optimization.
10. Profiling is crucial to identify and fix performance bottlenecks.

5.2 Conclusion

This project successfully accelerated MNIST digit classification using GPU-based neural network implementations. Starting from a sequential baseline, we achieved up to **16x speedup** through CUDA optimizations like shared memory, kernel fusion, and CUDA streams. Finally, leveraging **Tensor Cores** further improved performance with minor accuracy trade-offs. Overall, the study highlights how effective GPU optimization significantly boosts neural network training efficiency.