HPC Project

Section CS-HPC-A, Spring 2025

Team Members:
- Ibrahim Awais 22i-0878
- Muhammad Taha 22i-0983
- Rahat Shafi 22i-1061

Accelerating a Neural Network on the MNIST Dataset Using CUDA

Abstract

This report documents the journey of converting a CPU-based neural network for MNIST digit classification into a GPU-accelerated version using CUDA. The MNIST Digit Classification benchmark is a series of 60,000 training and 10,000 evaluation images to judge a Machine Learning algorithm on its speed and accuracy. Four solutions for this algorithm were explored:

- V1 (sequential CPU implementation)
- V2 (naive GPU implementation)
- V3 (optimized GPU version)
- V4 (Tensor Core-accelerated version)

The aim was to analyze the performance benefits, understand GPU programming, and explore the CUDA memory hierarchy and optimization strategies.

V1: Sequential CPU Implementation

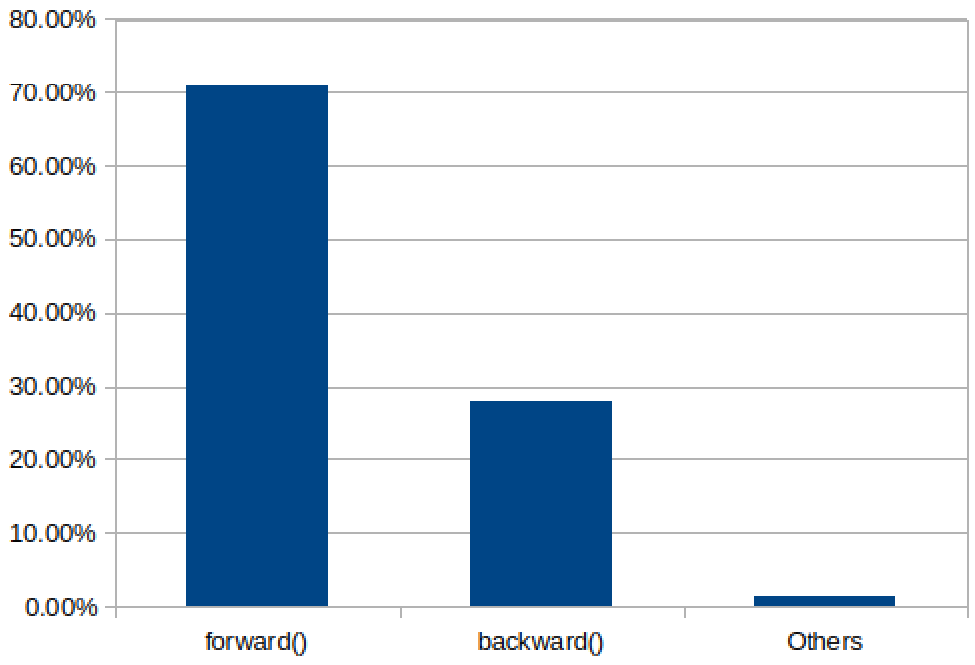
The initial implementation was a neural network written in C, designed to classify the MNIST dataset. It used a fully connected architecture. The dimensions were 784 nodes (because the images were 28x28) for the input layer, 128 for the hidden layer, and 10 (because the values could be 0 to 9) for the output layer. The hidden layer was trained using backpropagation.

This version served as a baseline for benchmarks and the Profiling tools to figure out which parts of the code needed improvement or optimization. gprof was run multiple times, and the results were as follows:

- ~ 71% of the time was taken by forward()
- ~ 28% of the time was taken by backward()
- ~ 1% was by all other functions (loading images, creating memory, etc.)

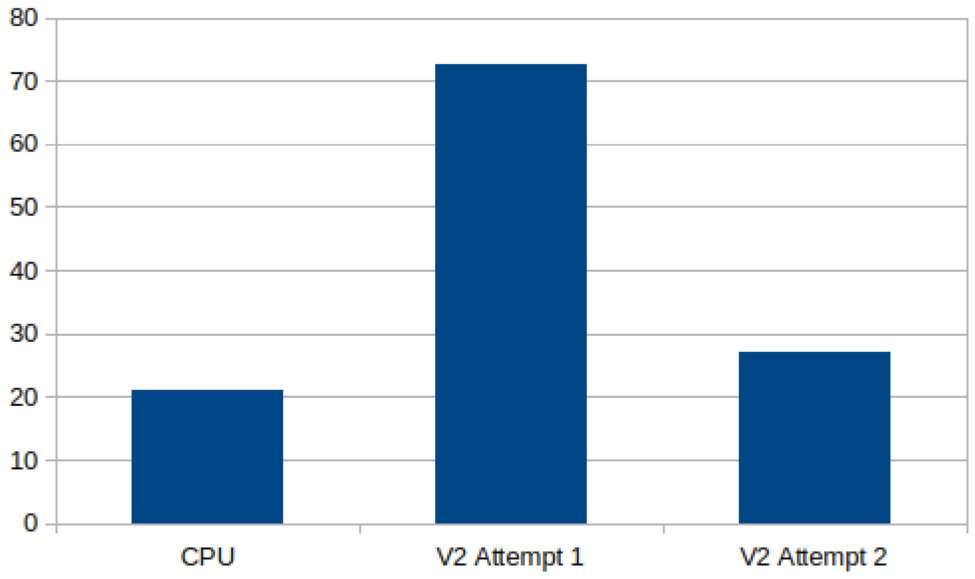
On an AMD Ryzen 7 5800H CPU the execution time per epoch was 7 seconds, and with 3 epochs ended up being 21 total seconds. Details of the gprof analysis can be found in the Deliverable 1 document.

There is, another thing to note here, however, which is something that will become very prevalent in V2. The forward() function taking up so much computation isn't because it's a slow function, but rather because it's called so often. It took 13.1 seconds of total CPU time and was called 190,000 times, meaning per function call it only occupied 70 microseconds of CPU time. This is not something which can necessarily be improved upon, and V2 proved this.



V2: Naive GPU Implementation

There were two implementations done for this. The first exists as V2_Attempt1.cu and the second exists as V2_Attempt2.cu. Both of these were implemented with the same goal in mind: attempt to parallelize within existing function loops and operations, with the aim of speeding up the function call to be as fast as possible. No major changes were made to the train() function, and in both of these, each image is still trained one at a time. Running both of these, the results were as follows:



Y axis: Time in seconds

From the chart we can observe that our original strategy of speeding up existing functions will not be applicable in this case, as instead of achieving a speedup, we have instead caused our code to slow down instead. Many attempts were made to improve it, such as using float instead of double, and optimizing memory transfers, however ultimately there was no major improvement to be had with these values. A different strategy was required, which is where V3 comes in.

V3: Optimized GPU Implementation

The primary focus with this version was to speed up our original code and optimize it, by the strategy of Batch Processing. In the program we observe slowdowns not because of how long it takes for the function to execute, but because of how often it has to be executed. Each function call requires the previous one to finish, and if this happens 190,000 times it makes sense as to why a GPU could not enhance its speeds.

The strategy, then, was to try and parallelize the loop within the train() function, so that instead of each image being processed one at a time, multiple were processed simultaneously. Adjustments had to be made for each array and to the code itself, so as a proof of concept, V3_CPU.c was made first. Consultation of many articles and existing codes from research papers was done to analyze how to properly utilize batch processing and rewrite the code to implement the algorithm as such, and finally after multiple hours of debugging, the CPU version of V3 was complete. It was fundamentally the same as the existing V1 version except now adjusting the BATCH_SIZE would give different results.

Once this was done, it was overhauled again to be written as CUDA instructions. This also took many hours of debugging but finally when completed and run, it provided a time of 0.5 seconds per epoch on average, which is a speedup of 14x our original V1 baseline. However, there was a major caveat:

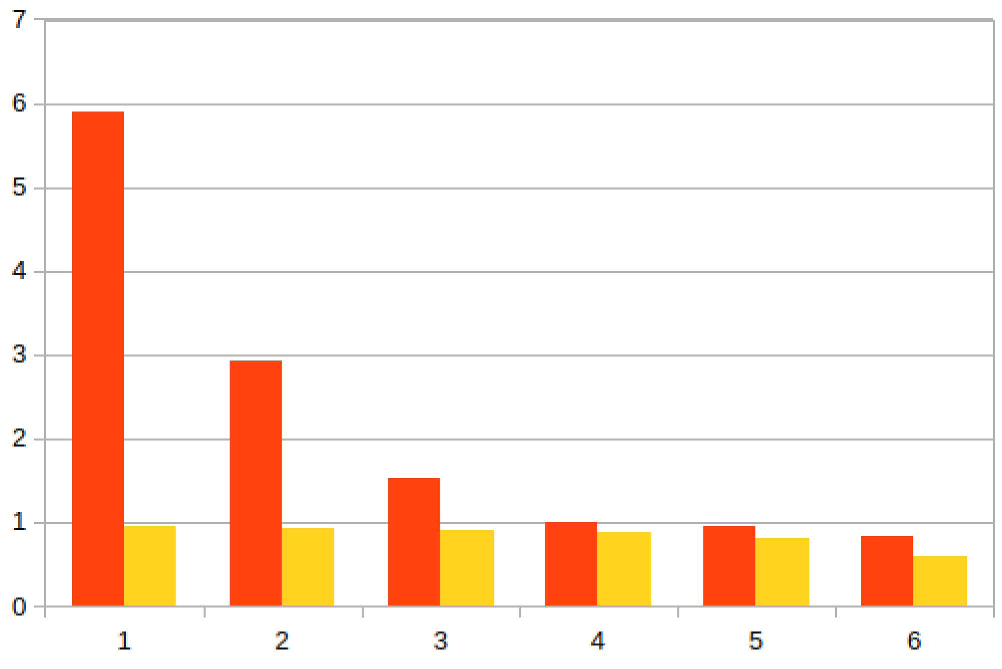
The speedup came at the cost of accuracy. The higher the batch size, the faster it got processed, however it became less and less accurate.

A table and bar chart is shown below to emphasize on how different values for BATCH_SIZE, EPOCHS and LEARNING_RATE affect the time and accuracy of the model. Here is how the values affect the code:

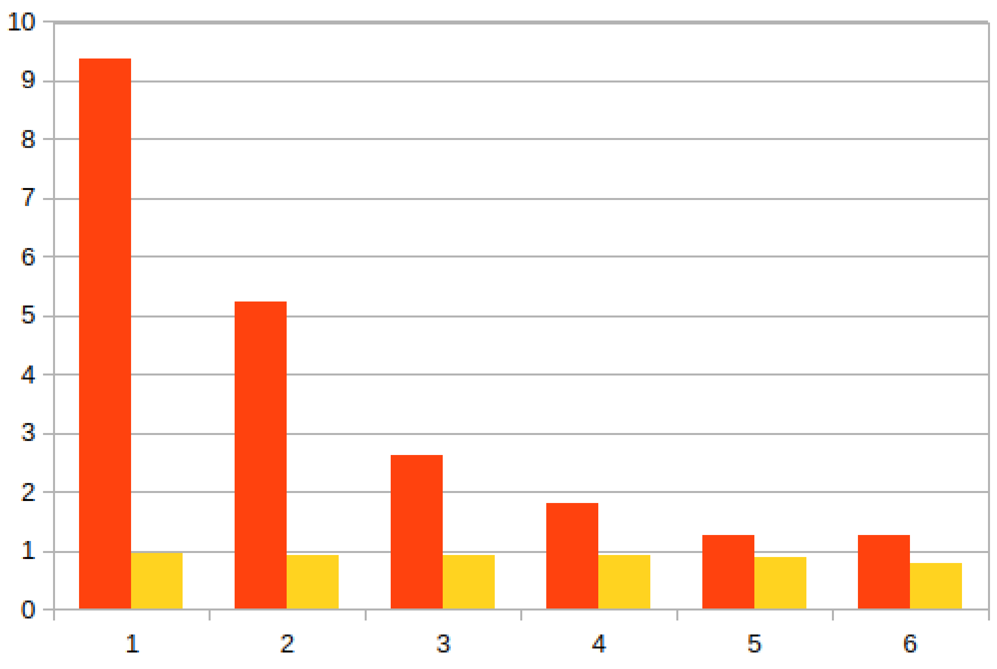
- EPOCHS tells how many times to run the data through. Setting this too low will mean a very small amount of training, and setting too high will mean overfitting of data.
- LEARNING_RATE tells how quickly or slowly the model learns. Setting too high will mean the quality of the learning will be lower, but setting too low will cause stagnation.
- BATCH_SIZE tells how many images to process at the same time. The higher the value, the faster the execution because of parallelism, however this will come at a cost of accuracy, as the neural network ends up averaging the results based on the batch size.

Although LEARNING_RATE does affect accuracy, our only concern is when that accuracy is linked with times, which only EPOCHS and BATCH_SIZE affect, so it’s kept at 0.01 for all tests.

Cases	BATCH_SIZE	EPOCHS	LEARNING_RATE	Total Time Taken	Accuracy
1	8	3	0.01	5.896s	94.08%
2	8	5	0.01	9.378s	95.91%
3	16	3	0.01	2.926s	91.98%
4	16	5	0.01	5.235s	93.43%
5	32	3	0.01	1.515s	90.41%
6	32	5	0.01	2.617s	91.84%
7	64	3	0.01	0.989s	88.11%
8	64	5	0.01	1.803s	90.35%
9	128	3	0.01	0.962s	81.32%
10	128	5	0.01	1.268s	87.57%
11	256	3	0.01	0.821s	60.00%
12	256	5	0.01	1.243s	78.37%



Y Axis: Time in seconds (3 Epochs comparison, these are cases 1, 3, 5, 7, 9, 11 respectively)



Y Axis: Time in seconds (5 Epochs Comparison, these are cases 2, 4, 6, 8, 10, 12 respectively)

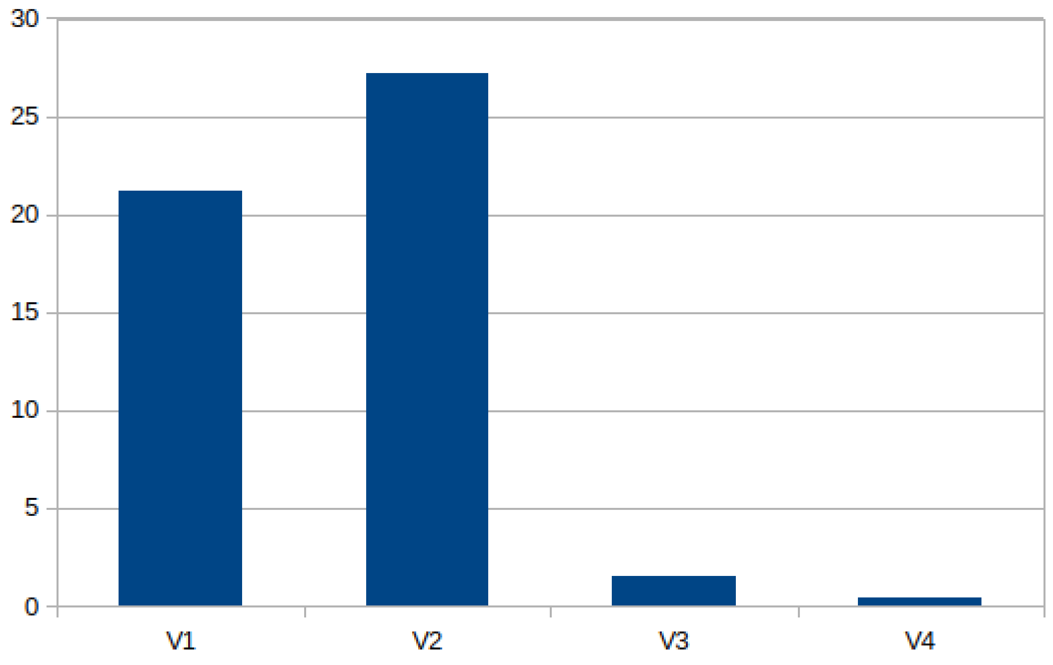
V4: Tensor-Core Accelerated Version

In the final version, we leveraged NVIDIA Tensor Cores (available in Volta and later architectures) to accelerate matrix multiplications, which are central to neural network training. Using batch sizes was the correct approach for V3, as it allowed V4 to properly be implemented as a Matrix Multiplication scenario. It did present complications, however, such as:

- Precision management: converting float32 to float16 required careful handling to avoid underflows, as Tensor Cores use half-precision floating-point (__half).
- Having to use WMMA (Warp Matrix Multiply Accumulate) APIs is significantly less straight-forward then CUDA syntax such as blockIdx.x and threadIdx.x
- Having to connect to and utilize a GPU which actually has Tensor Core compatibility.

Due to restrictions and complications with times and 7 other assignments and projects being due at the same time, the implementation of V4 at the time of writing is incomplete. However, if it does end up being complete before this group's presentation is due, an up to date report with the appropriate values and graphs will be present on the Github repository.

In theory, with a Batch Size of 32, Tensor Cores should offer a major improvement by ~ 3x or more, and with higher Batch Sizes this could increase even more.



Conclusion

This project provided valuable hands-on experience with CUDA, GPU programming, and neural network optimization. We progressed from a simple CPU-bound neural net to a high-performance GPU-accelerated version that utilized CUDA's full potential. By understanding memory hierarchies, optimizing kernel execution, and leveraging specialized hardware like Tensor Cores, we achieved significant performance gains. The process highlighted the trade-offs between performance and precision and demonstrated the importance of architectural awareness in high-performance computing.