High-Performance Computing with GPUs

Project: Accelerated MNIST Classification

Aalyan Raza Kazmi, 22i0833 - Sahrish Mustafa, 22i0977

Table of Contents

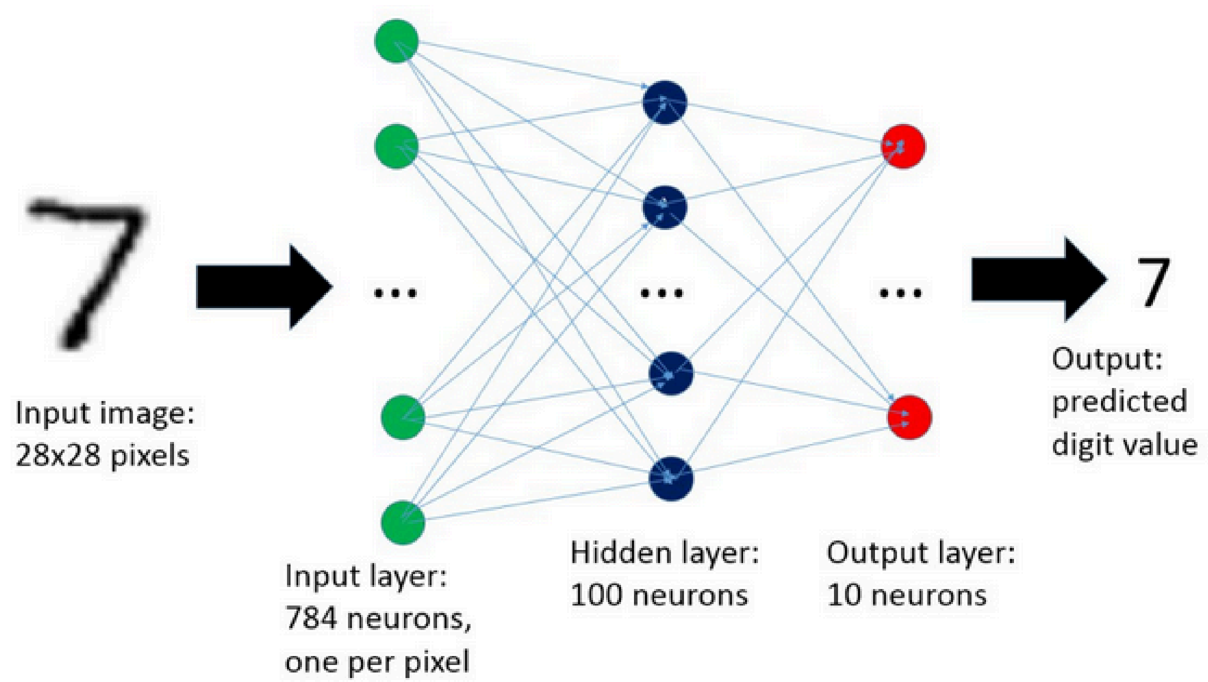
Introduction

MNIST digit classification, involving 28x28 grayscale images of handwritten digits (0–9), is a standard benchmark in machine learning. This project applies CUDA GPU programming to speed up the training and inference of a neural network on MNIST, showcasing how HPC techniques enhance computational performance in real-world AI tasks.

Problem

The neural network takes each image as a flattened array of pixel intensity values and passes it through multiple layers of interconnected nodes. Through repeated exposure to training data, the network adjusts its configurations. Over time, the model "learns" to recognize the underlying patterns in the pixel data that correspond to each digit.

A demonstration of this is given below.



Implementation Versions

V1: Baseline

Neural Network Structure:

- **Input layer**: 784 nodes (28×28 pixels) - each input.
- **Hidden layer**: 128 nodes with **ReLU** activation function.
- **Output layer**: 10 nodes with **Softmax** (representing digits 0–9).

Forward propagation multiplies input by weights → adds bias → applies ReLU → feeds to output layer → applies Softmax.

```
hidden = ReLU(W1 * input + b1)
output = Softmax(W2 * hidden + b2)
```

Backward propagation computes gradients by:

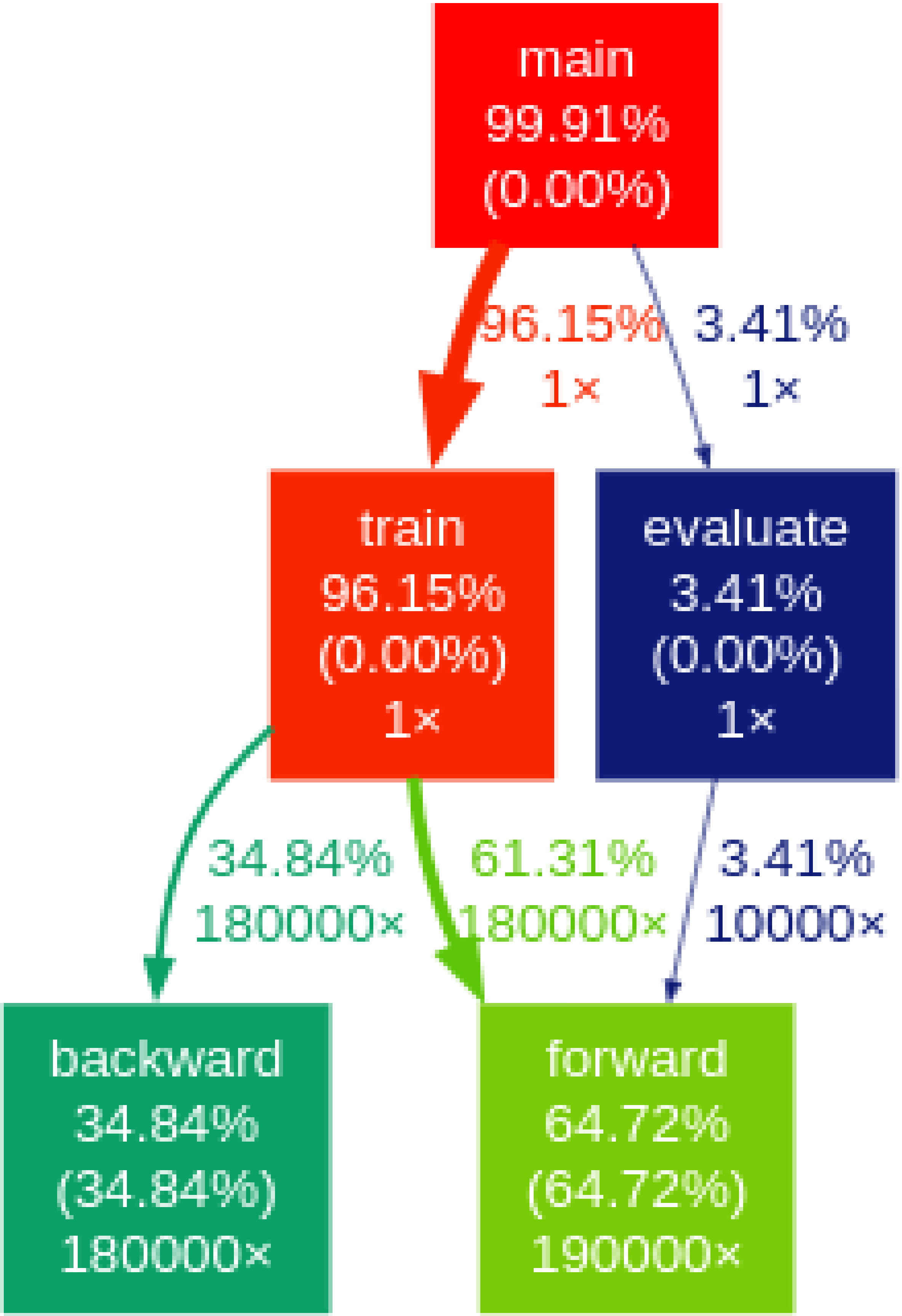
- Output error: output - target
- Hidden error via chain rule and ReLU derivative

Updates weights and biases using **gradient descent**:

```
W2 -= LR * d_output * hidden^T
W1 -= LR * d_hidden * input^T
```

The training loop repeats the process for 3 epochs on all 60,000 training images, measures loss (cross-entropy) and accuracy per epoch.

The serial implementation takes an average of `77.101` second training time, with most of the computation focussed on forward and backward propagation functions.


main
99.91%
(0.00%)
96.15%
1×
3.41%
1×
train
96.15%
(0.00%)
1×
evaluate
3.41%
(0.00%)
1×
34.84%
180000×
61.31%
180000×
3.41%
10000×
backward
34.84%
(34.84%)
180000×
forward
64.72%
(64.72%)
190000×

V2: Naive GPU Implementation

This implementation introduces CUDA. By integrating CUDA, we shift from the serial CPU-based implementation to a parallelized approach that can take advantage of the thousands of threads available on a GPU.

The serial implementation helped identify computational bottlenecks. These are now targeted for parallel execution to improve performance.

CUDA kernels

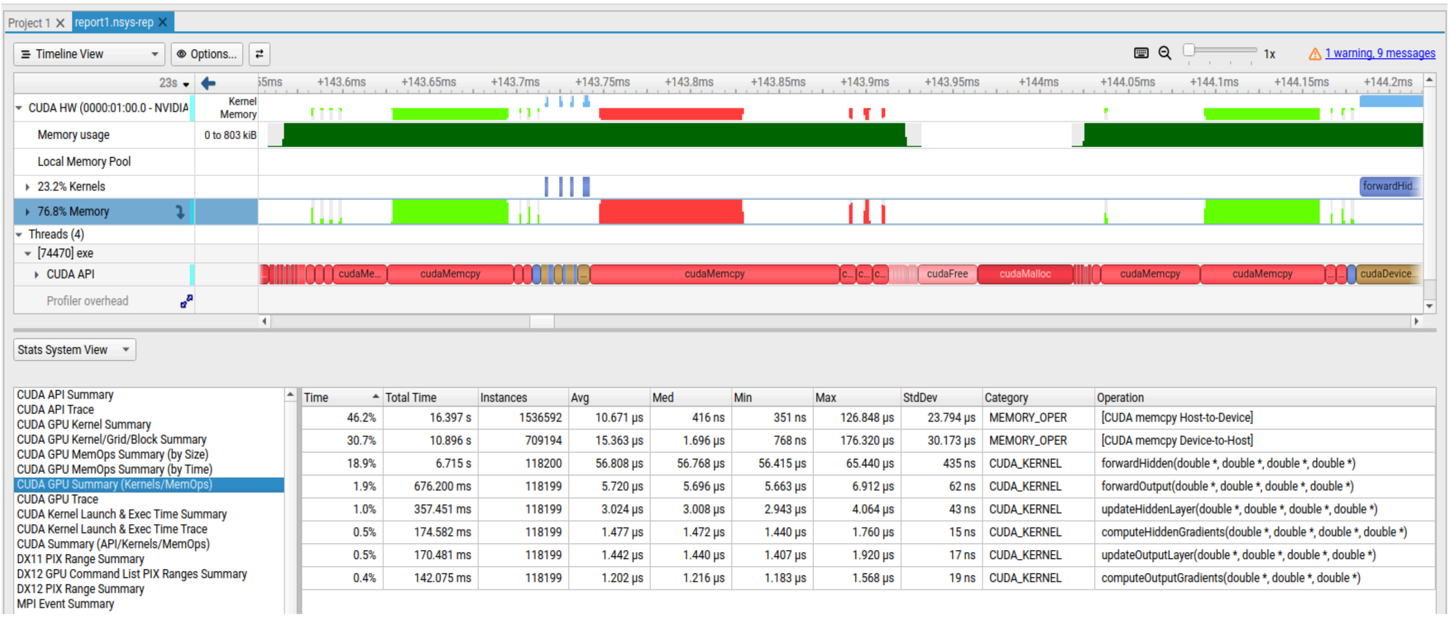
The forward propagation functions, **(`forwardHidden`, `forwardOutput`)** and the backward propagation functions **(`updateOutputLayer`,`updateHiddenLayer`)** have been implemented in the form of **CUDA kernels**.

Device-Side Memory Optimization

A key design decision in this version was to keep all weights and biases on the GPU (device memory) throughout training. By allocating them once at the beginning and updating them in-place on the GPU, we reduce memory transfer overhead and maintain consistency in computation.

Additionally, the entire dataset is also copied once to the device at the start of training and remains there for the duration of the process. This ensures that we do not need to perform repeated cudaMemcpy() calls for each image, which would otherwise become a major bottleneck.

Mainly, this communicational optimization insight was given to us by using the profiling tool, which showed that most of our time was being spent in copying data to and fro between the device and the host.



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
46.2%	16.397 s	1536592	10.671 µs	416 ns	351 ns	126.848 µs	23.794 µs	MEMORY_OPER	[CUDA memcpy Host-to-Device]
30.7%	10.896 s	709194	15.363 µs	1.696 µs	768 ns	176.320 µs	30.173 µs	MEMORY_OPER	[CUDA memcpy Device-to-Host]
18.9%	6.715 s	118200	56.808 µs	56.768 µs	56.415 µs	65.440 µs	435 ns	CUDA_KERNEL	forwardHidden(double *, double *, double *, double *)
1.9%	676.200 ms	118199	5.720 µs	5.696 µs	5.663 µs	6.912 µs	62 ns	CUDA_KERNEL	forwardOutput(double *, double *, double *, double *)
1.0%	357.451 ms	118199	3.024 µs	3.008 µs	2.943 µs	4.064 µs	43 ns	CUDA_KERNEL	updateHiddenLayer(double *, double *, double *, double *)
0.5%	174.582 ms	118199	1.477 µs	1.472 µs	1.440 µs	1.760 µs	15 ns	CUDA_KERNEL	computeHiddenGradients(double *, double *, double *, double *)
0.5%	170.481 ms	118199	1.442 µs	1.440 µs	1.407 µs	1.920 µs	17 ns	CUDA_KERNEL	updateOutputLayer(double *, double *, double *, double *)
0.4%	142.075 ms	118199	1.202 µs	1.216 µs	1.183 µs	1.568 µs	19 ns	CUDA_KERNEL	computeOutputGradients(double *, double *, double *)

Profiler Results

The NVIDIA Nsight Systems Profiling tools gives us accurate measurements of time for each kernel and memory transfer. The optimized version, showing lesser memory transfers is shown below.

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
79.8%	11.203 s	190000	58.963 µs	58.912 µs	57.568 µs	67.649 µs	684 ns	CUDA_KERNEL	forwardHidden(double *, double *, double *, double *)
7.7%	1.082 s	190000	5.694 µs	5.696 µs	5.600 µs	6.881 µs	66 ns	CUDA_KERNEL	forwardOutput(double *, double *, double *, double *)
4.1%	581.479 ms	380000	1.530 µs	1.536 µs	512 ns	22.465 µs	345 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
3.9%	544.478 ms	180000	3.024 µs	2.976 µs	2.784 µs	4.289 µs	182 ns	CUDA_KERNEL	updateHiddenLayer(double *, double *, double *, double *)
1.9%	266.865 ms	180000	1.482 µs	1.472 µs	1.440 µs	1.792 µs	22 ns	CUDA_KERNEL	computeHiddenGradients(double *, double *, double *, double *)
1.8%	246.253 ms	180000	1.368 µs	1.376 µs	1.312 µs	1.856 µs	22 ns	CUDA_KERNEL	updateOutputLayer(double *, double *, double *, double *)
0.8%	111.976 ms	180006	622 ns	352 ns	320 ns	40.013 ms	95.556 µs	MEMORY_OPER	[CUDA memcpy Host-to-Device]

The speed-ups comparing every version to V1 (serial) can be found at the end [here.]

V3: Optimized GPU Implementation

Since most of the memory transfer time has now been greatly reduced, we will now be aiming to make the kernels faster.

Launch Configuration

The launch configurations on the naive implementation were set to use 1 block only which for obvious reasons did not yield the best results. We tried various combinations of multiple threads and blocks to see which one yields the best result (in terms of time for each epoch). As a result of this hit and trial method, We concluded that with these combinations the time consumed was lowest.

```
forwardHidden<<<4, 32>>>
forwardOutput<<<1, 64>>>
computeHiddenGradients<<<16, 4>>>
updateOutputLayer<<<OUTPUT_SIZE, HIDDEN_SIZE>>>
updateHiddenLayer<<<HIDDEN_SIZE, INPUT_SIZE>>>
```

The code was also changed a bit to manage multiple blocks instead of just one.

Shared Memory Usage

To further improve performance, **shared memory** was introduced in performance-critical kernels to reduce redundant global memory access. `The forwardHidden` and `forwardOutput` kernels seem to be taking the most amount of time, and therefore we will be aiming to reduce the time it takes by using shared memory.

Once the launch configurations changed, and shared memory was implemented, the following times were noted on the profiler.

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
69.3%	6.607 s	190000	34.773 μs	32.994 μs	32.194 μs	48.707 μs	2.114 μs	CUDA_KERNEL	forwardHidden(double *, double *, double *, double *)
10.8%	1.031 s	190000	5.424 μs	5.152 μs	4.993 μs	6.241 μs	340 ns	CUDA_KERNEL	forwardOutput(double *, double *, double *, double *)
6.5%	618.806 ms	380000	1.628 μs	1.600 μs	480 ns	24.001 μs	271 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
5.4%	519.306 ms	180000	2.885 μs	2.816 μs	2.304 μs	4.320 μs	209 ns	CUDA_KERNEL	updateHiddenLayer(double *, double *, double *, double *)
3.8%	358.952 ms	180000	1.994 μs	2.080 μs	1.824 μs	2.369 μs	112 ns	CUDA_KERNEL	computeHiddenGradients(double *, double *, double *, double *)
3.0%	286.495 ms	180000	1.591 μs	1.632 μs	1.408 μs	2.176 μs	53 ns	CUDA_KERNEL	updateOutputLayer(double *, double *, double *, double *)
1.2%	116.710 ms	180006	648 ns	416 ns	320 ns	39.812 ms	95.095 μs	MEMORY_OPER	[CUDA memcpy Host-to-Device]

When we compare these times to the ones in the naive implementation, we see that the time is greatly reduced.

Note: We tried shared memory on the other two functions too but they did not seem to be affecting the performance a lot. Since accessing global memory was not a bottleneck in these kernels, using shared memory gave us little leverage.

Half-Precision (FP16) Data Type Usage

FP16 (half) uses 2 bytes (16 bits) per number instead of 8 bytes which we were previously using. This means for every calculation we are doing (weights, activations and gradients), the data is ¼ times smaller in size making the calculations quicker. This helps reduce bottlenecks caused by limited memory bandwidth.

Although we did have concerns about compromising the accuracy/precision of our model's predictions, no significant loss in accuracy was observed. It improved our time greatly.

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
70.1%	5.133 s	190000	27.017 µs	27.008 µs	26.464 µs	36.896 µs	248 ns	CUDA_KERNEL	forwardHidden(__half *, __half *, __half *, __half *)
8.4%	613.448 ms	380000	1.614 µs	1.600 µs	832 ns	21.025 µs	285 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
7.9%	578.068 ms	190000	3.042 µs	3.040 µs	3.008 µs	4.608 µs	17 ns	CUDA_KERNEL	forwardOutput(__half *, __half *, __half *, __half *)
5.1%	373.844 ms	180000	2.076 µs	2.080 µs	1.984 µs	3.264 µs	49 ns	CUDA_KERNEL	updateHiddenLayer(__half *, __half *, __half *, __half *)
3.8%	278.374 ms	180000	1.546 µs	1.536 µs	1.536 µs	2.176 µs	16 ns	CUDA_KERNEL	computeHiddenGradients(__half *, __half *, __half *, __half *)
3.5%	257.601 ms	180000	1.431 µs	1.440 µs	1.408 µs	2.080 µs	16 ns	CUDA_KERNEL	updateOutputLayer(__half *, __half *, __half *, __half *)
1.2%	86.796 ms	180006	482 ns	416 ns	384 ns	10.077 ms	24.081 µs	MEMORY_OPER	[CUDA memcpy Host-to-Device]

Coalesced Memory Access

Memory being accessed in column-major format seemed to be taking more time, so we changed it and stored it in row-major format. This allowed performance improvement by almost 2 seconds per epoch.

All memory is accessed as contiguously as possible, meaning threads in a warp read memory addresses adjacent to each other. This makes sure that minimal bank conflicts occur in the shared memory.

Pinned Memory Usage

Although we did try using pinned memory, the result was way worse than our previous versions, so we dumped that idea.

Profiler Results

The most optimized version reported the following times:

Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
61.4%	3.269 s	190000	17.206 µs	17.184 µs	16.639 µs	19.872 µs	310 ns	CUDA_KERNEL	forwardHidden(__half *, __half *, __half *, __half *)
9.9%	527.456 ms	380000	1.388 µs	1.535 µs	480 ns	20.160 µs	265 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
9.5%	504.786 ms	190000	2.656 µs	2.656 µs	2.623 µs	3.520 µs	38 ns	CUDA_KERNEL	forwardOutput(__half *, __half *, __half *, __half *)
5.4%	290.296 ms	180000	1.612 µs	1.600 µs	1.535 µs	1.920 µs	31 ns	CUDA_KERNEL	updateHiddenLayer(__half *, __half *, __half *, __half *)
5.0%	264.852 ms	180006	1.471 µs	1.408 µs	320 ns	10.096 ms	24.052 µs	MEMORY_OPER	[CUDA memcpy Host-to-Device]
4.6%	244.794 ms	180000	1.360 µs	1.344 µs	1.311 µs	1.696 µs	24 ns	CUDA_KERNEL	computeHiddenGradients(__half *, __half *, __half *, __half *)
4.3%	227.049 ms	180000	1.261 µs	1.248 µs	1.215 µs	1.472 µs	23 ns	CUDA_KERNEL	updateOutputLayer(__half *, __half *, __half *, __half *)

The speed-ups comparing every version to V1 (serial) can be found at the end [here.]

V4: Tensor Core Utilization

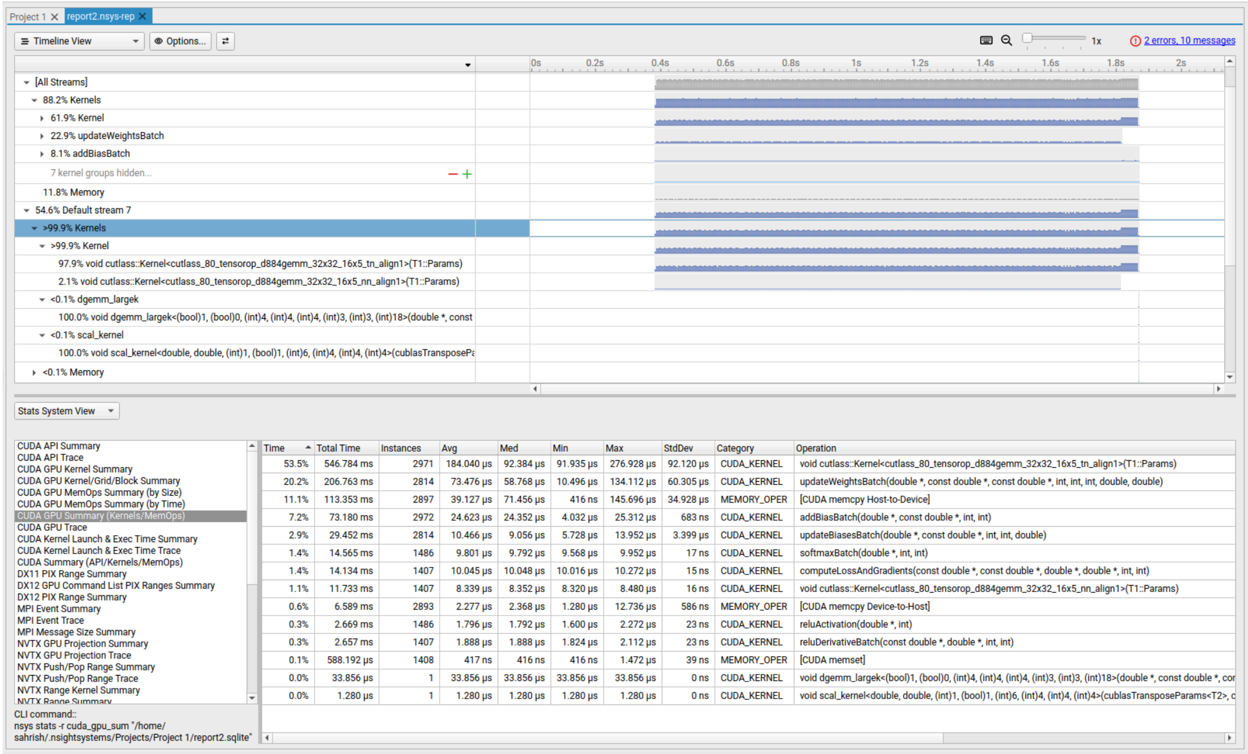
This version largely focused on batches and matrix multiplication. This led to the updation of the forward and back propagation in the following ways:

Forward Propagation

1. Input → Hidden Layer:
   - Matrix Multiply: Uses cublasDgemm (cuBLAS) for W1 @ input_batch.
   - Bias Add: Custom kernel addBiasBatch adds biases to all samples in the batch.
   - Activation: Leaky ReLU applied via reluActivation kernel.
2. Hidden → Output Layer:
   - Matrix Multiply: cublasDgemm for W2 @ hidden_activations.
   - Bias Add: Same as above.
   - Softmax: Batched softmax via softmaxBatch kernel for numerical stability.

Backward Propagation

1. Loss & Output Gradients:
   - Kernel: computeLossAndGradients computes cross-entropy loss and gradients (output - target) in one pass.
2. Hidden Gradients:
   - Matrix Multiply: cublasDgemm computes W2^T @ output_gradients.
   - ReLU Derivative: reluDerivativeBatch applies gradients based on hidden layer activations.
3. Weight Updates:
   - Kernels: updateWeightsBatch (for W1/W2) and updateBiasesBatch (for b1/b2).



Time	Total Time	Instances	Avg	Med	Min	Max	StdDev	Category	Operation
53.5%	546.784 ms	2971	184.040 μs	92.384 μs	91.935 μs	276.928 μs	92.120 μs	CUDA_KERNEL	void cutlass::Kernel<cutlass_80_tensorop_d884gemm_32x32_16x5_tn_align1>(T1::Params)
20.2%	206.763 ms	2814	73.476 μs	58.768 μs	10.496 μs	134.112 μs	60.305 μs	CUDA_KERNEL	updateWeightsBatch(double *, const double *, const double *, int, int, int, double, double)
11.1%	113.353 ms	2897	39.127 μs	71.456 μs	416 ns	145.696 μs	34.928 μs	MEMORY_OPER	[CUDA memcpy Host-to-Device]
7.2%	73.180 ms	2972	24.623 μs	24.352 μs	4.032 μs	25.312 μs	683 ns	CUDA_KERNEL	addBiasBatch(double *, const double *, int, int)
2.9%	29.452 ms	2814	10.466 μs	9.056 μs	5.728 μs	13.952 μs	3.399 μs	CUDA_KERNEL	updateBiasesBatch(double *, const double *, int, int, double)
1.4%	14.565 ms	1486	9.801 μs	9.792 μs	9.568 μs	9.952 μs	17 ns	CUDA_KERNEL	softmaxBatch(double *, int, int)
1.4%	14.134 ms	1407	10.045 μs	10.048 μs	10.016 μs	10.272 μs	15 ns	CUDA_KERNEL	computeLossAndGradients(const double *, const double *, double *, double *, int, int)
1.1%	11.733 ms	1407	8.339 μs	8.352 μs	8.320 μs	8.480 μs	16 ns	CUDA_KERNEL	void cutlass::Kernel<cutlass_80_tensorop_d884gemm_32x32_16x5_nn_align1>(T1::Params)
0.6%	6.589 ms	2893	2.277 μs	2.368 μs	1.280 μs	12.736 μs	586 ns	MEMORY_OPER	[CUDA memcpy Device-to-Host]
0.3%	2.669 ms	1486	1.796 μs	1.792 μs	1.600 μs	2.272 μs	23 ns	CUDA_KERNEL	reluActivation(double *, int)
0.3%	2.657 ms	1407	1.888 μs	1.888 μs	1.824 μs	2.112 μs	23 ns	CUDA_KERNEL	reluDerivativeBatch(const double *, double *, int, int)
0.1%	588.192 μs	1408	417 ns	416 ns	416 ns	1.472 μs	39 ns	MEMORY_OPER	[CUDA memset]
0.0%	33.856 μs	1	33.856 μs	33.856 μs	33.856 μs	33.856 μs	0 ns	CUDA_KERNEL	void dgemm_largek<(bool)1, (bool)0, (int)4, (int)4, (int)4, (int)3, (int)3, (int)18>(double *, const double *, cor
0.0%	1.280 μs	1	1.280 μs	1.280 μs	1.280 μs	1.280 μs	0 ns	CUDA_KERNEL	void scal_kernel<double, double, (int)1, (bool)1, (int)6, (int)4, (int)4, (int)4>(cublasTransposeParams<T2>, c

The speed-ups comparing every version to V1 (serial) can be found at the end [here.](#)

Performance Analysis

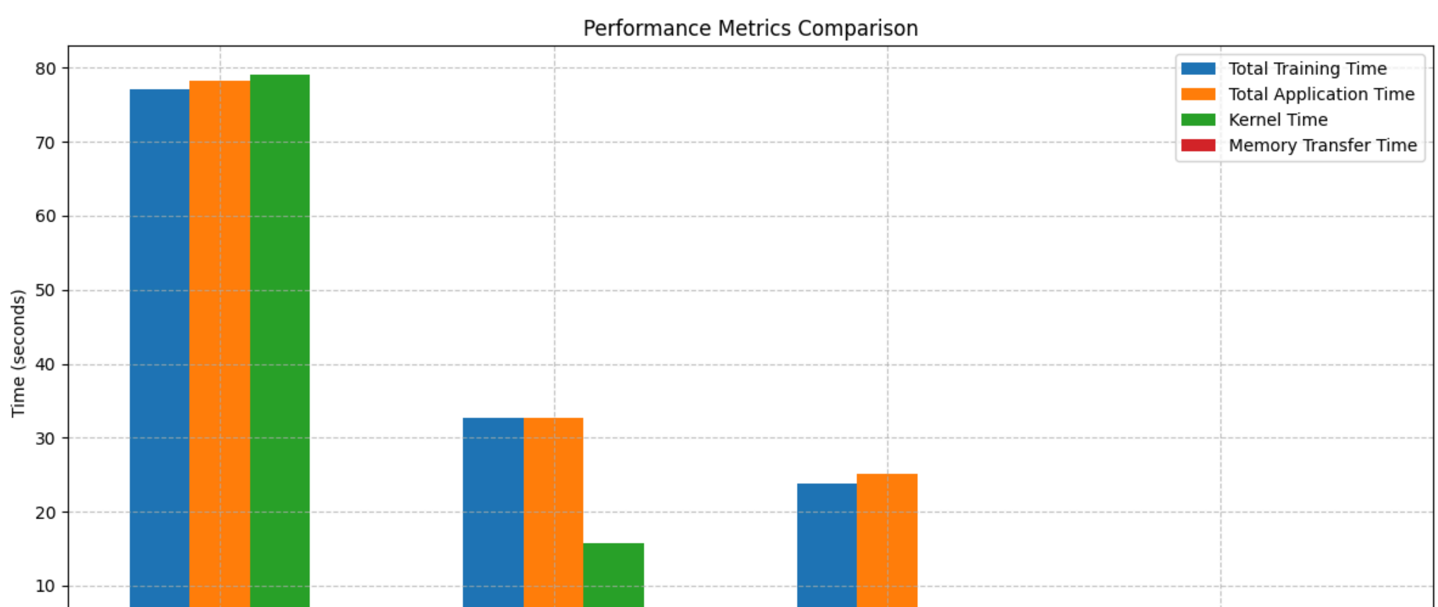
> Note: The serial implementation does not include any hardware optimization instructions to the compiler. This means that the -o2 flag included in the original makefile has been removed so that the impact of GPU parallelization can be clearly demonstrated.

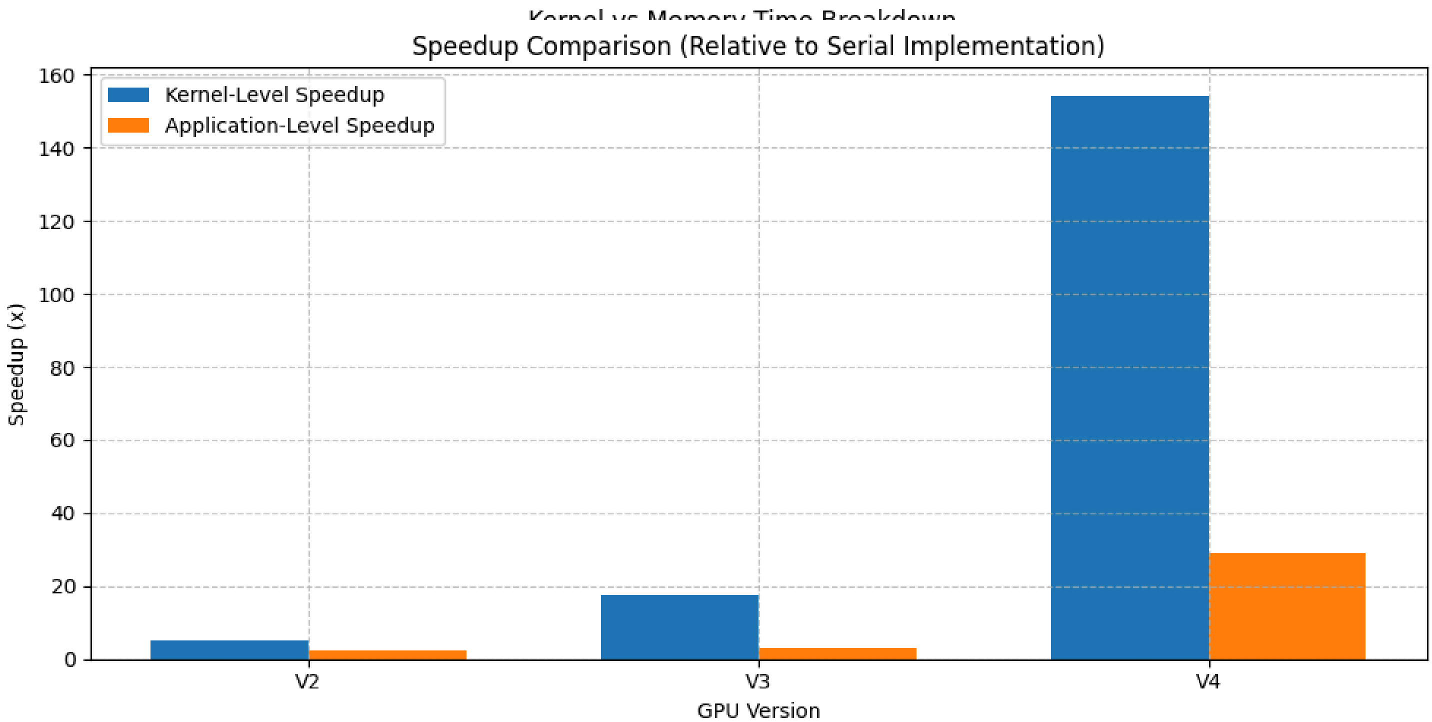
Although outputs have been generated in each version of code for calculating speed-ups, using the NVIDIA Nsight Systems profiling tool gives us better, more accurate measurements of time.

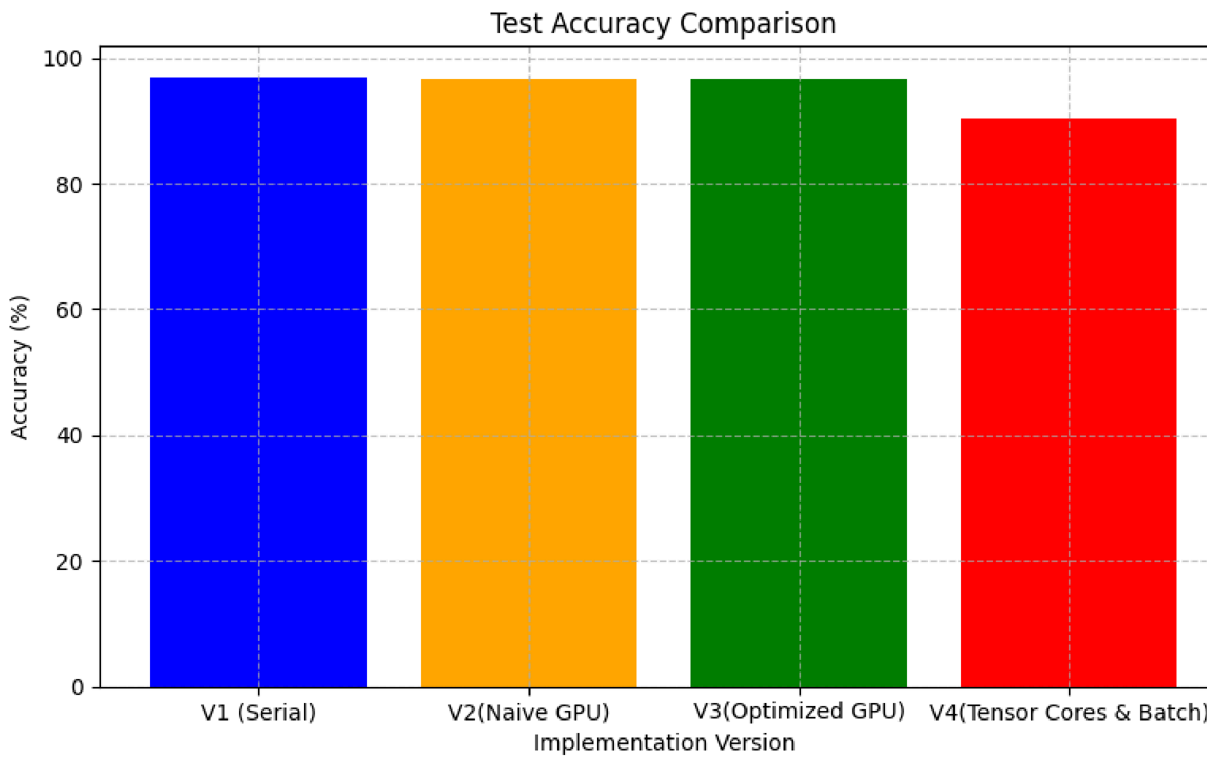
These timings have been reported for each version above, and used to calculate speed-ups below.

Number of Epochs	3			
Timing (s)	**V1 (Serial)**	**V2(Naive GPU)**	**V3(Optimized GPU)**	**V4(Tensor Cores)**
Total Training Time	77.101	32.671	23.841	1.396
Total Application Time (train + evaluate)	78.311	32.763	25.1168	2.699
Kernel Time (forward + back prop)	79.009	15.744381	4.556925	0.512
Memory Transfer Time	-	0.693455	0.792308	0.18
Kernel + Memory Transfer	79.097	16.437836	5.349233	0.692
Test Accuracy (%)	97	96.72	96.55	90.29
Speed Up (Kernel-Level)		5.018234759	17.33822698	154.3144531
Speed Up (Application-Level)		2.39022678	3.117873296	29.0148203

The results are presented in tabular format below.




Speedup Comparison (Relative to Serial Implementation)
Kernel-Level Speedup
Application-Level Speedup
Speedup (x)
0
20
40
60
80
100
120
140
160
V2
V3
V4
GPU Version



Conclusion

This project showcased how GPU acceleration using CUDA can significantly improve neural network training times for MNIST classification. By progressively optimizing memory access, kernel configuration, and precision, we reduced training time from 77 seconds to just 1.4 seconds. Although minor accuracy trade-offs occurred, the performance gains were substantial. Overall, the project highlights the power of parallel computing in real-world machine learning tasks.