CME 213 Final Report

Jeremy P. Binagia

November 16, 2021

1 Introduction

In this project[1], we will use multiple graphics processing units (GPUs) to accelerate the training of a deep neural network on the MNIST dataset, i.e. images of handwritten digits. To accomplish this, we will utilize Google Cloud to access a virtual environment where we can run our program concurrently on up to four Tesla K80 GPUs. The CUDA API will be used to facilitate GPU programming, and we will leverage MPI to dictate how each of the devices communicate with one another. We will begin by discussing how and what computation is implemented on the GPU as well as how we have structured our program in regards to MPI.

2 How Training was Implemented on Multiple GPUs

2.1 CUDA implementation of GEMM

Perhaps the computation that stands to gain the most in terms of speed when running on the GPU is the general matrix multiply operation, otherwise known as "GEMM". GEMM defines a general matrix multiplication for $\mathbf{A} \in \mathbb{R}^{M\times K}$, $\mathbf{B} \in \mathbb{R}^{K\times N}$, $\mathbf{C} \in \mathbb{R}^{M\times N}$, $\alpha \in \mathbb{R}$, and $\beta \in \mathbb{R}$ as follows:

$$\mathbf{C} = \alpha\mathbf{AB} + \beta\mathbf{C} \tag{1}$$

It is important that we perform this operation as efficiently as possible since this fundamental computation is repeatedly used during both the feedforward and backpropogation sections of each training step.

The "naive" implementation involves assigning each thread in a CUDA warp a specific element of $\mathbf{C}$ to compute. While simple to design, this implementation should be inferior to more well-thought algorithms since there is a great deal of inefficiency in this approach; for example, threads in a warp in this case do not share any information such as common elements of $\mathbf{A}$ or $\mathbf{B}$ when computing their entry of $\mathbf{C}$.

Better implementations typically leverage CUDA shared memory to minimize costly reads and writes to global memory. The algorithm I implemented, based on section 4.1 in the first final project handout, falls into this category. In this method, a square block of threads of size `BLOCK_SIZE` is responsible for calculating a sub-matrix of $\mathbf{C}$, $\mathbf{C}_{sub}$. Similar to before, each thread is responsible for calculating a specific element of $\mathbf{C}_{sub}$. The main computation involves looping over the sub-matrices of $\mathbf{A}$ and $\mathbf{B}$ required to compute $\mathbf{C}_{sub}$ For each iteration, sub-matrices $\mathbf{A}_{sub}$ and $\mathbf{B}_{sub}$ are loaded into shared memory so that multiple threads can efficiently access their values. Each thread can then compute $\alpha\mathbf{A}_{sub}\mathbf{B}_{sub}$ and accumulate this value into a local variable. Once the main loop is completed, each thread writes its value to global memory.[2]

2.2 Additional functions implemented on the GPU

In addition to GEMM, all other matrix-vector computations that are required for the training process are implemented using CUDA. These include matrix addition/subtraction, the transpose operation, Armadillo's `repmat` function (which replicates a vector to form a matrix), summation of a specified matrix dimension, and the following element-wise operations: multiplication, exponentiation, sigmoid, and softmax. The implementations of each of these functions may be found in the file `gpu_func.cu`.

[1] The repository for this project may be found at `https://github.com/jbinagia/cme213-final-project`.

[2] I also implemented the algorithm discussed in section 4.2 (`gpuGEMM4d2` in `gpu_func.cu`). While it performs the GEMM calculation correctly, I found the implementation discussed here to more efficient, and so I did not explore this method further.

2.3 MPI implementation

In this following we will discuss how MPI is used to distribute data and computation amongst the N processes. Initially, only the rank 0 process has access to the entire training set. Thus, we begin by looping over each batch of data and distributing an equal amount of this data to each of the processes via a call to `MPI_Scatter`. Each process stores each of its "minibatches" of data that it receives from process 0 in a `std::vector` container. Note that for the final batch `MPI_Scatterv` is used instead to take care of the case in which the training set size is not divisible by the batch size (and thus the last batch is smaller than usual).

The training process proceeds as it would for the case of a single process, with each process calculating gradients with which to update the neural network coefficients based on its minibatch for that iteration. At this point, however, the processes have only computed "local" contributions to the total gradients required to update the network weights and biases; hence, a call to `MPI_Allreduce` is used to combine these values and to distribute the result to each process, who can then update its network coefficients. Each of these MPI calls may be found in the `neural_network.cpp` file, specifically within the funciton `parallel_train`.

3 Correctness of the Implemented Algorithm

3.1 Verifying the GEMM calculation that is performed on the GPU

To verify our implementation, we begin by checking our GEMM calculation against an established benchmark (i.e. that provided by `cublasDgemm`). To perform this test, we use grading mode 4, which performs the GEMM operation using both the cuBLAS and the user-provided algorithm for two different matrix sizes. For both cases, $\alpha = 2$ and $\beta = 5$. For the first test, the matrix multiplication is carried out for matrix dimensions $M = 800$, $N = 1000$, and $K = 784$ (recall $\mathbf{A} \in \mathbb{R}^{M\times K}$, $\mathbf{B} \in \mathbb{R}^{K\times N}$, and $\mathbf{C} \in \mathbb{R}^{M\times N}$). In this case, the relative difference between my GEMM implementation and that of the reference is $1.857\,46 \times 10^{-16}$. For the second test, we consider $M = 800$, $N = 10$, and $K = 1000$. In this case, the relative difference between the two solutions is 3.2505×10^{-16}. Because these values are both on the order of magnitude of working precision, we conclude that the user-defined GEMM has been implemented correctly.

3.2 Verifying the MPI implementation used in network training

We now proceed to verify the correctness of our entire implementation (both MPI and CUDA together). To do this, we run the program using grading modes 1, 2, and 3, which differ in their learning rate (0.001, 0.01, and 0.025 respectively) and their number of epochs (40, 10, and 1 respectively).[3] The results of these tests are shown in table 1. Here, for each grading mode, we list the error between the parallel and sequential implementation for each of the neural network coefficients (measured using both the max and L2 norm). Each test is run for a number of processes ranging from oen to four to ensure the correctness of the MPI implementation. We see that in all cases the errors are below the desired tolerance of 10^{-7}, thereby verifying that our parallel implementation is correct.

4 Profiling and Analysis

4.1 Tuning block size to optimize GEMM

The first step I took in optimizing my code prior to profiling was selecting the optimal CUDA block size for my GEMM implementation. This involved re-running grading mode 4 for a range of block sizes (referred to here and in `gpu_func.cu` as `BLOCK_SIZE`). Note that `BLOCK_SIZE` sets the size of both of the dimensions of each 2D block; this is a result of the fact that the algorithm described in section 2.1 only works for square thread blocks. The results of these tests may be found in table 2. We see that for both sets of matrix sizes (i.e. test 1 and 2) that a block size of 16 gives the fastest execution time. For this reason, this block size will be used for the remainder of the report.

[3]Unless otherwise stated, the other hyperparameters take on their default values; i.e. the the batch size is 800 and the L2 regularization is 0.0001. The exception is the number of neurons, whose default value is 1000 but is set to 100 for grading.

Grading mode	num_procs	W[0]		b[0]		W[1]		b[1]		Time (s)		Precision
		max norm	L2 norm	max norm	L2 norm	max norm	L2 norm	max norm	L2 norm	Parallel	Sequential	
1	1	1.40E-15	1.26E-15	3.75E-15	1.75E-15	3.64E-16	8.40E-16	1.53E-15	8.59E-16	18.4831	540.135	0.894333
	2	1.61E-15	1.45E-15	4.64E-15	2.04E-15	4.75E-16	1.01E-15	1.69E-15	9.68E-16	11.6833		
	3	1.68E-15	1.52E-15	4.92E-15	2.14E-15	5.11E-16	1.14E-15	1.84E-15	9.96E-16	14.883		
	4	1.51E-15	1.35E-15	4.22E-15	1.82E-15	4.62E-16	9.21E-16	1.23E-15	7.98E-16	17.1794		
2	1	9.82E-09	7.11E-09	1.48E-08	4.67E-09	6.11E-11	1.27E-10	4.98E-10	4.65E-10	5.41098	78.1479	0.924167
	2	7.89E-09	5.72E-09	1.19E-08	3.78E-09	5.05E-11	9.89E-11	4.10E-10	3.47E-10	4.05076		
	3	2.85E-08	2.06E-08	4.32E-08	1.36E-08	1.82E-10	3.60E-10	1.48E-09	1.28E-09	3.88052		
	4	6.57E-09	4.79E-09	9.97E-09	3.15E-09	4.29E-11	8.34E-11	3.49E-10	3.04E-10	4.97615		
3	1	6.93E-12	6.64E-12	2.28E-11	7.74E-12	3.83E-14	1.43E-13	4.47E-13	4.90E-13	6.56215	12.7516	0.845167
	2	4.13E-11	3.93E-11	1.06E-10	2.99E-11	2.01E-13	8.46E-13	2.52E-12	2.77E-12	6.28473		
	3	3.06E-12	3.23E-12	2.65E-11	7.17E-12	1.10E-14	7.16E-14	6.77E-14	8.18E-14	11.5126		
	4	8.81E-12	8.40E-12	2.27E-11	6.19E-12	4.28E-14	1.86E-13	5.41E-13	5.78E-13	11.4637		

Table 1: Measured error for the parallel neural network coefficients (relative to that computed using the sequential implementation) for each grading mode. The final two columns list the execution times and the precision obtained on the validation set.

Test \BLOCK_SIZE	4	8	16	32
Test 1	0.654227	0.140203	0.128672	0.132084
Test 2	0.009971	0.00299	0.002807	0.004545

Table 2: Execution time in seconds for grading mode 4 as a function of the CUDA block size, `BLOCK_SIZE`. Each row (i.e. each of the constituent tests for grading mode 4) are colored according to their relative ordering in terms of speed, with green being the fastest and red being the slowest.

4.2 Optimizing the training loop: reducing memory allocations and transfers

Initially, the execution time for the entire program running on 4 GPUs was much slower than its current value (for example, commit `48f2053` ran in 92.1804 s while the current implementation takes 26.6605 s). The first optimization to enable this speedup was significantly reducing the number of calls to `cudaMalloc` and `cudaMemcpy` by allocating device memory only once outside of the entire training loop and keeping data on the device for as long as possible. This change alone, made in commit `3056174`, decreased the execution time to 56.596 s. After this, reducing redundant calls to `cudaMalloc` and `cudaFree` by moving such statements outside of the training loop increased the speed further, to a runtime of 48.0565 s (commit `8046165`). By commit `32d6f40`, the remaining calls to `cudaMalloc` and `cudaFree` in the `GPUbackprop` function were removed and costly array copies were avoided by retrieving (and subsequently passing to functions) raw pointers for the minibatch `arma` matrices as opposed to creating local `arma` matrices to represent them. This lead to the mostly optimized time of 29.35 s as compared to the original parallel runtime.

4.3 Profiling the entire program while using 4 GPUs

We will profile our program using the `nvprof` command line profiler. Here, we consider the default command line arguments of 1000 neurons, a batch size of 800, an L2 regularization of 0.0001, and 40 epochs. From viewing the timeline of events in the NVIDIA Visual Profiler (c.f. fig. 1), we immediately see that much of the runtime is dominated by repeated `cudaMemcpy`'s (specifically from the device to the host) immediately followed by "blank spaces" in the profiler. The former memory transfers correspond to the point within each iteration at which the local gradients described in section 2.3 must be moved from the GPU to the CPU before being combined via `MPI_Allreduce`. The latter blank space then likely corresponds to the `MPI_Allreduce` operation itself since we know it will not be explicitly represented in the profiler. This suggests that the best way to further improve performance is to minimize the amount of data that needs to be offloaded from the GPU and the amount of communication between the different processes. One way to accomplish this is to move from the data parallelism described here to the more advanced "model parallelism" approach. Here, data transfer and communication can be kept to a minimum by broadcasting the input images only at the beginning of training and by intelligently splitting the model itself (i.e. the network coefficients) amongst the processes depending upon the dimensions of the relevant weight matrices.

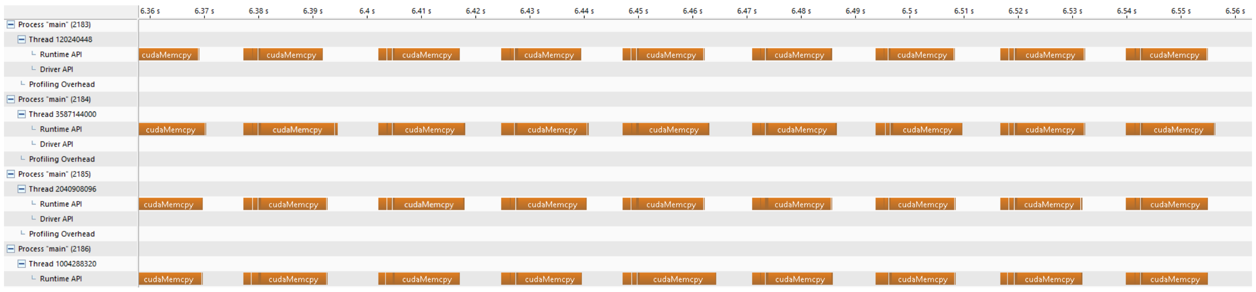


Figure 1: Event timeline from the NVIDIA Visual Profiler, zoomed-in to illustrate relative duration of the `cudaMemcpy` events.

Kernel	Avg. Duration (ns)	% of Total Execution Time
gpuGEMM4d1()	7860726	93.2
elemwise()	328472	1.6
sum()	314124	2.2
sigmoidKernel()	146940	0.3
addmat()	127352	1.8
transpose()	87592	0.6
repmatKernel()	33154	0.2

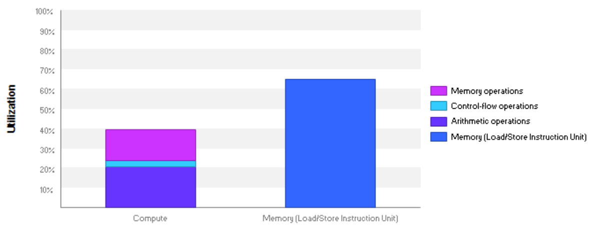
Table 3: GPU kernels ordered by average duration in nanoseconds. Also listed is the percentage of time of all instances of the given kernel relative to the total execution time of all kernels. Kernels whose percent of total execution time is less than 0.1% are omitted in this table.

4.4 In-depth profiling of the GEMM implementation

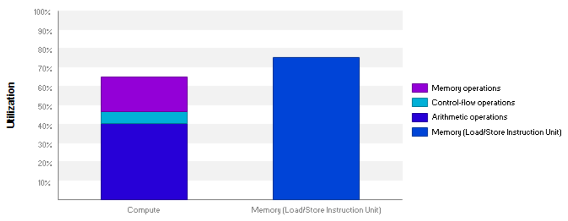
We would now like to examine the kernels in more depth to see what optimizations could be made in regards to computations done on the GPU. Running `nvprof` with the `-analysis-metrics` option allows us to do just this (now running for only one epoch to save time). The first thing we take note of is the relative importance of the kernels we have implemented. This can be seen in section 4.4, where the kernels are listed in order of decreasing average duration. Clearly the `gpuGEMM4d1()` is the slowest function run on the GPU, with calls to this function making up approximately 93.2% of the total execution time of all kernels. It is for this reason that all subsequent analysis will focus on this kernel.

By analyzing `gpuGEMM4d1()` in particular, we find that this kernel is limited by memory bandwidth (c.f. fig. 2a). Because computation or instruction / memory latency are not the limiting factors, we will focus our attention on increasing memory bandwidth for this kernel. In fig. 3, one thing that sticks out is that "ECC Overhead" is highlighted by the profiler. The CUDA C Best Practices Guide mentions that "accessing memory in a coalesced way is even more important when ECC is turned on" and that "scattered accesses increase ECC memory transfer overhead, especially when writing data to the global memory." Indeed, when we examine the "global memory access pattern" section of the profiler while re-running the analysis for a range of block sizes, we find that the global load/store L2 transactions per memory access is always equal to the chosen block size. Note that the ideal ratio is 8; this is because for devices of compute capability 3.x (the Tesla K80 is 3.7), accesses to global memory are only cached in the L2 cache, which uses 32 byte cache lines. There are 32 threads per warp meaning that a warp would need to access 256 bytes of data when each thread is accessing a double; hence, the minimum number of transactions to service this request is 8.

Why does the kernel deviate from this ideal ratio? In my unoptimized kernel (prior to commit `0c6490a`), the row and column of A_{sub} and B_{sub} that each thread is responsible for reading into shared memory is given by `row = threadIdx.y` and `col = threadIdx.x` respectively. The problem with this is that CUDA numbers threads in a thread block with `threadIdx.x` varying the fastest (in essence row-major ordering); in contrast, the matrices for our problem are stored in memory in column-major order. The result of this inconsistency is exactly the behavior described in the previous section; `BLOCK_SIZE` memory transactions will be needed per



(a) Prior compute utilization



(b) Current compute utilization

Figure 2: Compute utilization before and after making the edits described in section 4.4, which were made to ensure global memory coalescence and minimize shared memory bank conflicts.

memory access since warps are trying to read from `BLOCK_SIZE` number of discontiguous memory addresses. This issue can easily be solved by reversing the assignments, i.e. setting `row = threadIdx.x` and `col = threadIdx.y`, which creates a nicely coalesced memory access pattern.

Initially, however, this change did not lead to an increase in performance as one might expect. This is because making this change in isolation leads to a great number of shared memory bank conflicts (as revealed by re-running `nvprof`). This can be seen by noting how elements are nominally loaded into shared memory: `As[row][col] = GetElement(Asub, row, col)`. As written, we are writing to shared memory in columns and hence `BLOCK_SIZE` threads are all trying to access the same memory bank. The simplest fix is a slight modification in the creation of the shared memory: for example, changing `As[BLOCK_SIZE][BLOCK_SIZE]` to `As[BLOCK_SIZE][BLOCK_SIZE+1]`. Considering a block size of 32 (i.e. the number of threads in a warp) for illustration, this simple padding will now map every thread to a distinct bank due to the modulo arithmetic involved in how bank indices are determined (because the stride is 33 and there are only 32 banks).[4]

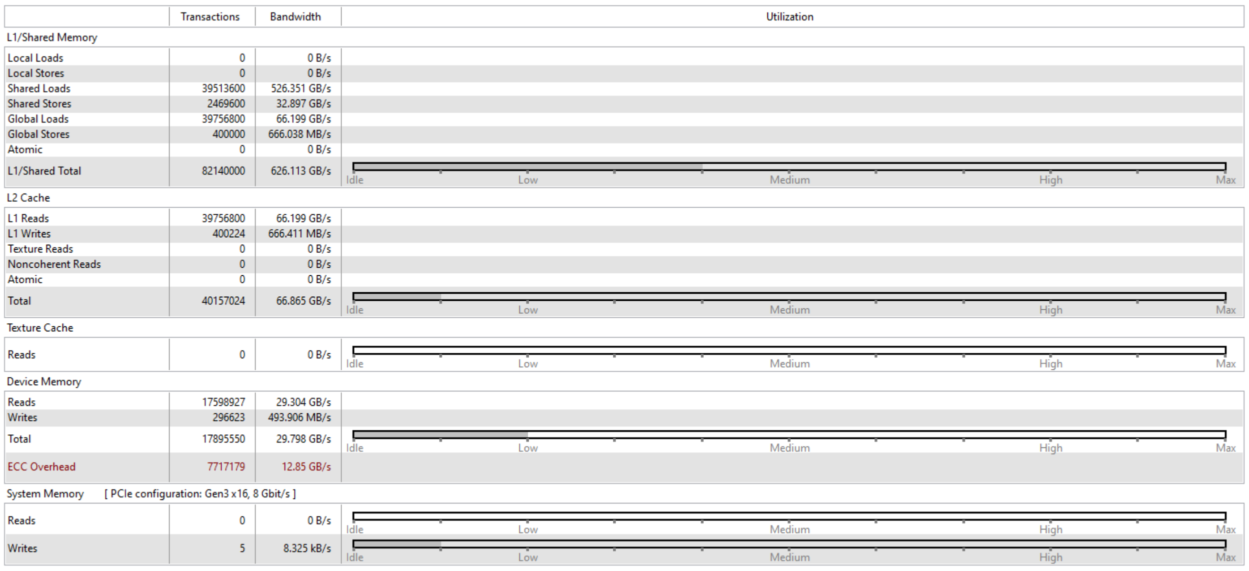
With this pair of changes, the execution time for this kernel increases significantly. By considering again the range of block sizes given in table 2, a block size of 16 was again found to be optimal. For this block size, the NVIDIA visual profiler indicates that there are now 2 shared memory transaction per access as opposed to the previous `BLOCK_SIZE` per access. Note that this is the best that can be achieved for a block size of 16, because 16 is half the warp size (meaning there will be 2 conflicts at each bank).[5]

For this block size, the runtimes for the two test cases given by grading mode 4 are 0.0652142 and 0.00172805 respectively, which are 2.0x and 1.6x faster than my unoptimized GEMM implementation (itself roughly comparable in speed to the naive GEMM implementation). The increase in performance can also be seen by comparing the compute utilization before and after the change (figs. 2a and 2b) and the prior and current memory profiles (fig. 3 and fig. 4). In regards to the former, while we are still bound by memory bandwidth, the percent utilization for memory has increased by 10% (from 65% to 75%) and that for compute has increased from 40% to 65%. The optimization is also reflected in the memory profiles; comparing fig. 4 to fig. 3 shows us that the utilization of shared memory has increased substantially (with bandwidth 2x as much as it was previously), ECC overhead transactions have decreased, and the bandwidth for reads from device memory has increased by about a factor of 1.3x.

Unfortunately, with this new optimization, the execution time for the program as a whole (running on 4 GPUs) only decreases by a few seconds (from 29.35 s to 26.6605 s when run using the default configuration). This marginal increase in performance is a classic instance of Amdahl's law. Suppose for the sake of argument that the GEMM computations make up 10% of the original runtime. Then Amdahl's law predicts that a 2x speedup of GEMM will only lead to a 1.05x speedup of the overall program, which is essentially what we observe. Hence, to significantly increase the execution speed, one should focus their efforts on the items discussed in section 4.3 (namely, moving to model rather than data parallelism).

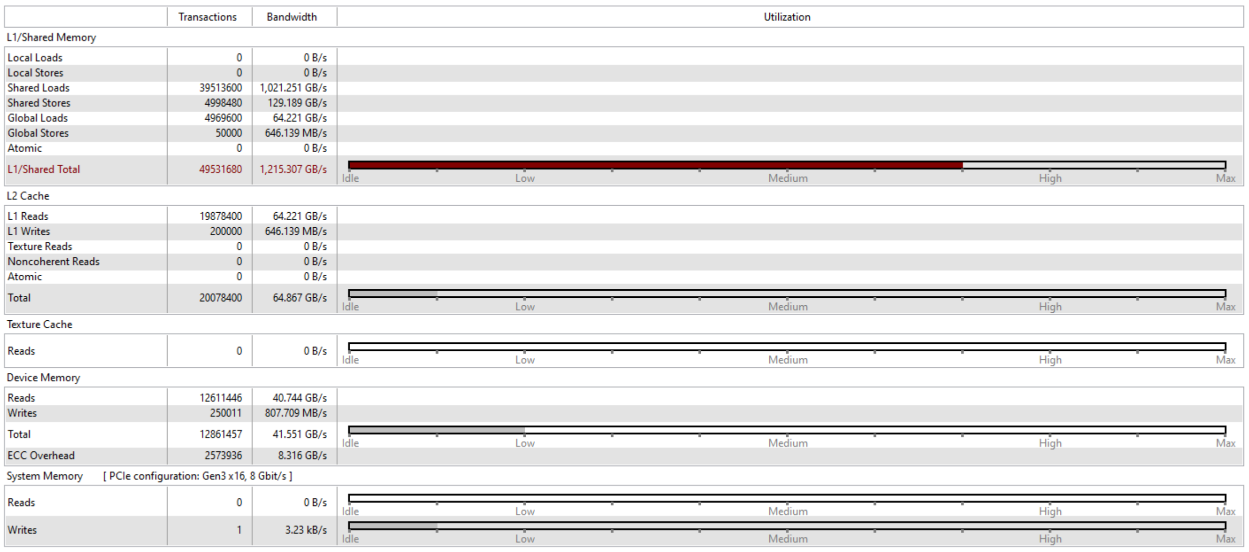
[4]This analysis assumes 64 bit banks are being used, which can be configured via `cudaDeviceSetSharedMemConfig()`.

[5]I also tested this by running the profiler with 64 bit banks and a block size of 32; as expected, the ideal memory transaction per access is achieved in this case.



	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	39513600	526.351 GB/s	
Shared Stores	2469600	32.897 GB/s	
Global Loads	39756800	66.199 GB/s	
Global Stores	400000	666.038 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	82140000	626.113 GB/s	
L2 Cache			
L1 Reads	39756800	66.199 GB/s	
L1 Writes	400224	666.411 MB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	40157024	66.865 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	17598927	29.304 GB/s	
Writes	296623	493.906 MB/s	
Total	17895550	29.798 GB/s	
ECC Overhead	7717179	12.85 GB/s	
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	5	8.325 kB/s	

Figure 3: Device memory profile before making the edits described in section 4.4.



	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	39513600	1,021.251 GB/s	
Shared Stores	4998480	129.189 GB/s	
Global Loads	4969600	64.221 GB/s	
Global Stores	50000	646.139 MB/s	
Atomic	0	0 B/s	
L1/Shared Total	49531680	1,215.307 GB/s	
L2 Cache			
L1 Reads	19878400	64.221 GB/s	
L1 Writes	200000	646.139 MB/s	
Texture Reads	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Atomic	0	0 B/s	
Total	20078400	64.867 GB/s	
Texture Cache			
Reads	0	0 B/s	
Device Memory			
Reads	12611446	40.744 GB/s	
Writes	250011	807.709 MB/s	
Total	12861457	41.551 GB/s	
ECC Overhead	2573936	8.316 GB/s	
System Memory [PCIe configuration: Gen3 x16, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	1	3.23 kB/s	

Figure 4: Device memory profile after making the edits described in section 4.4.

5 Conclusion

In conclusion, we have successfully used CUDA and MPI to accelerate training of a deep neural network on the MNIST dataset. The execution of the optimized program using the default hyperparameter values was 26.6605 s, which is two orders of magnitude faster than that for sequential execution (1215.93 s). Key in obtaining this speedup was minimizing the amount of GPU memory allocations and data transfer between the GPU and CPU (discussed in section 4.2) as well optimization of the GEMM operation (section 4.4). Profiling the entire program (section 4.3) (in addition to custom timings in the source code) suggest that further speedup can be obtained by reducing communication between the processes and the amount of data that needs to be transferred from the device to host at each iteration. Perhaps the best way to achieve is to distribute the neural network weights between the processes, as opposed to distributing the input data (i.e. transitioning from data to model parallelism); this is left for future work.