ECE 408 Final Report

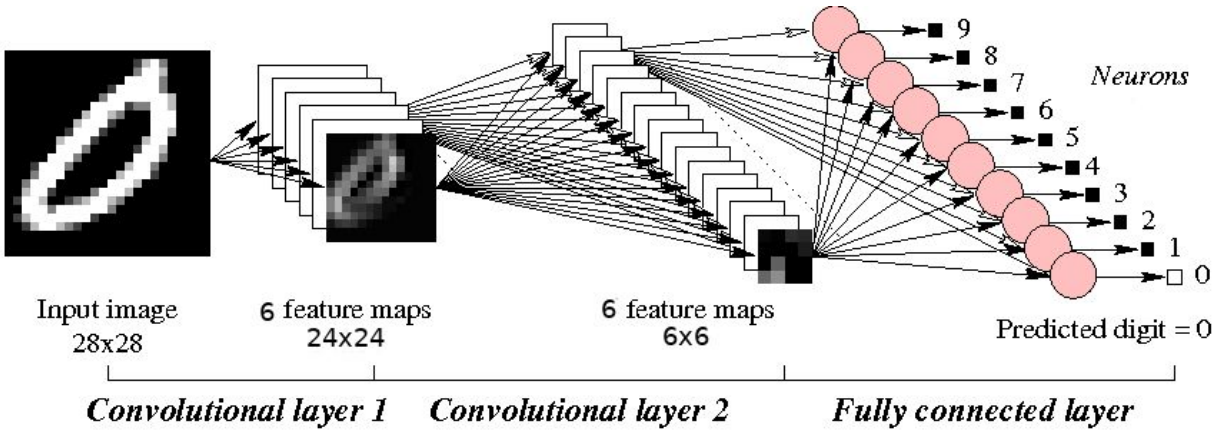
Team 7: Peiming Wu, Xiuyi Qin, Evan Chang, Rio Martinez

1. Resources

We used private GPU resources to analyze running time with reduced variance.
Individual kernel profiles were run on local GPUs, but total running times were run on RAI.

2. Application:

We solved the problem of handwritten digit classification (Topic 4) for our final project. This problem required us to develop a neural network with several hidden layers to enable our algorithm to learn how to classify written digits. To aid our training, we used a database of handwritten digits sourced from the MNIST dataset for training and validation of our algorithm. For training, we used 60,000 uniquely labeled digits, and for testing, we used a smaller set of 10,000 images.



The overall architecture of the implemented CNN[3]

We selected this topic for several reasons. Firstly, this problem is most efficiently solved by using a neural network. This attribute makes the solution a good candidate for parallelization on

a GPU since neural networks perform many iterations of multiplication, addition and convolution for gradient descent (a process by which the neural network learns how to adjust its own weights and biases to converge to a correct solution). Secondly, our team found personal interest in the topic of digit classification, as it is a small example of using neural networks, which can easily be generalized to larger problems that are traditionally difficult to design algorithms for.

3. Milestones

Through development of our final project, we summarized our progress into three checkpoints/milestones. This section provides an overview of our aggregate process from all milestones.

3.1 Background

In this project, we developed a convolutional neural network to recognize handwritten digits on a CUDA GPU. The basic idea is to take a large number of the dataset from the Modified National Institute of Standard and Technology (MNIST), and develop a system that can learn from those training datasets. The neural network we built consists of an input layer, a hidden layer, and an output layer. The neural network is able to determine weights and biases using a gradient descent algorithm. We used backpropagation to compute the gradients of each layer in our network to minimize our network's error. We built a CNN with two convolutional layers and a fully connected layer. The two convolutional layers consist of the one with 6 24*24 filter maps and the one with 6 6*6 filter maps. For the backward propagation, we used gradient descent method to reach the minimum value of the cost function. We used the common sigmoid function:

$$s(x) = \frac{1}{1+e^{-x}}$$

to work as the activation function. The final output of our CNN to each input image is a set of probabilities corresponding to each label (i.e. from 0~9). Our error function takes the error between the expected output probabilities and the actual probabilities we got.

3.2 Implementation

Our overall parallel strategies focus on reducing the running time of convolution since most of our CUDA kernels perform convolution. We created four kernels to implement a sigmoid function and to compute the error for each output vector. Additionally, we created six kernels to implement two convolution layers and one fully connected layer with different weights and bias

in forward propagation. For the back propagation process, we made 10 kernels. We used shared memory to store our input image and weights to speed up the reading processes. However, some kernels result in slower running time, which will be mentioned in more detail in following sections. Hence, some of our kernels' inputs in shared memory and some are just in global memory. The output of our kernels are writing into global memory. The work for each thread varies among kernels too. We made some kernels to do only one multiplication for one thread. But for some other kernels, we decided to let each thread compute the one output pixel in the convolution.

3.3 Optimization

3.3.1 Shared Memory

When we only implemented shared memory optimization, the performance varied depending on the function and memory size. As we can see in the charts below, when we put 6*24*24 masks from forwarding propagation in shared memory other than global memory, we gain 1 second faster than the original code on average of 5 runs. However, If we try to do the same thing to 6*6*6 masks in hidden layers, the training time will increase and accuracy will decrease. The increasing time is due to the overhead for assigning space for comparatively smaller masks in shared memory. The decreasing accuracy is due to the accuracy lost during data transfer.

Size of mask in shared memory	Original code	6*24*24 mask	6*6*6 mask
Training time (s)	119.51	118.90	124.68
Accuracy for 5 iterations	96.11	96.11	95.23
Speedup	1	1.005	0.958

As mentioned above, using shared memory does not always result in a speedup. The table below shows more details about how using shared memory can slow down kernels. This might be because modern GPUs have global memory caches that can quickly respond to global read requests.

	Kernel Time	SM Usage	Memory Usage
No shared memory	4.70us	8.35%	9.20%

Shared convolution mask	5.31us	3.97%	10.05%
Shared convolution mask + input	6.69us	4.81%	8.06%

3.3.2 Constant Memory

The input data, which is the image, can be stored in constant memory on the device since we don't need to modify it. The result varies a lot as we run for dozens of tests. We take the average training time to mark the performance of our code.

As we can see from the table, putting input data in constant memory did not decrease the running time of our forward propagation kernel overall. In fact, it significantly decreased our runtime performance when unilaterally applied to all kernels in our neural network. This could be due to extra copies required between each layers' inputs and outputs in our launch code, or due to the global memory cache in modern GPUs.

Because each iteration of our forward and backward propagation kernels required new input, we have concluded that our project is not a good candidate for constant memory usage. If our kernels performed multiple runs with identical inputs, constant memory would have provided more substantial performance gains.

	Code optimized with atomic adding and shared memory	Code adding constant memory
Training time (s)	24.11	24.37
Accuracy for 5 iterations	96.11	96.11

3.3.3 Atomic operations

In our milestone 2 code for convolution, each thread calculated one output element. By using one thread for each calculation and atomic add operations, we can increase parallelism. This optimization would likely be beneficial for convolutions where the mask size is large relative to the input size, as using one thread per output element would result in very few threads being launched and very large for loops.

The table below shows the execution times of two convolution kernels, `fp_preact_c1` for forward propagation, and `bp_weight_c1` for backpropagation.

	`fp_preact_c1`	`bp_weight_c1`
for loops	8.26us	271.81us
w/ atomic operations	12.16us	36.06us

With atomic operations, the forward convolution kernel showed a small increase in execution time, but the backprop convolution kernel had a drastic 7.5x speedup. By using Atomicadd for the `bp_weight_c1` kernel, we were able to reduce our total 5 iteration training time from 95s to 27s, a 2.8x speedup.

Network accuracy was unaffected by this optimization.

3.3.4 Temporary registers

In several kernels, we performed several sequential global memory accesses in succession. Since these accesses were to identical locations in global memory, we moved these accesses into local variables within the kernels so that they could be stored in temporary registers within the SM. This decreased the running time of our forward-propagation first convolution layer.

Global writes	8.26us
w/ temporary register	4.70us

3.3.5 Unrolling loops

By unrolling our for-loops in several of our kernels, we can reduce loop calculation overhead. Our first convolution layer kernel for forward-propagation saw no benefit in performance. It's possible that the NVCC compiler had already unrolled the for loops automatically as an optimization, or that the overhead of calculating the loops was negligible.

w/ for loops	4.70us
w/ unrolled for loops	4.70us

4. Results

We kept our accuracy results consistent from milestone 1 to the end, while improving the running time. Below are detailed performance results for the project on RAI.

Training iterations	5	10	20
Accuracy (%)	96.11	96.54	96.86
Training time per iteration before optimizations (s)	23.90	23.76	23.83
Training time per iteration after optimizations (s)	4.87	4.84	4.83

5. Conclusion

In this project, we optimized the training performance of our handwritten digit classifier by 79% by using resources learned from ECE 408 course curriculum. This was accomplished by evaluating the usage of several CUDA constructs including *shared memory*, *constant memory,* and *atomicAdd*. Our evaluation led to several unexpected discoveries, including the decrease in performance observed by implementing constant memory in our forward propagation kernel. Across all of our forward and backward propagation kernels, using *atomicAdd* for our convolution operations yielded the highest performance gains. Through the optimization of this final project, we have garnered a deeper understanding of neural networks, parallel processing performance optimization and computer architecture.

6 References

[1] Texture Memory in CUDA: What is Texture Memory in CUDA programming. (n.d.). Retrieved from http://cuda-programming.blogspot.com/2013/02/texture-memory-in-cuda-what-is-texture.html

[2] Kirk, D., & Hwu, W.-mei W. (2017). Programming massively parallel processors: a hands-on approach. Amsterdam; Boston; Heidelberg: Elsevier.

[3] Rajas Chavadekar, rvc-mnist-cnn-gpu, Retrieved from https://github.com/rvcgeeks/rvc-mnist-cnn-gpu

[4] LeNet-5 - A Classic CNN Architecture, Muhammad Rizwan. (2018) Retrived from: https://engmrk.com/lenet-5-a-classic-cnn-architecture/