ECE 408 Final Project Milestone 3

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

Milestone 2 Timing Results

Training iterations	5 Iterations	10 Iterations	20 Iterations
Training time (s)	119.51	237.96	476.62
Test accuracy (%)	96.11	96.54	96.86

Optimization Analysis.

Shared Memory Use

When we only implemented shared memory optimization, the performance varied depending on the function and memory size. As we can see in the above chart, 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. 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 due to 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

	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%

Constant Memory Use

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. 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. We plan to experiment with adding the usage of constant memory to only our backwards-propagation kernels. This is because our backwards-propagation kernels run many times during a training epoch with similar/identical input data. We can investigate this optimization before our final report.

	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

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 back propagation.

	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 atomic adds 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.

Temporary register

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

Unrolling for loops

By unrolling our for-loops in several of our kernels, we were able to increase parallelization. This is because we use one thread per calculation. As we can see, not all kernels benefitted from this optimization. Our first convolution layer kernel for forward-propagation saw no benefit in performance:

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

Overall, however, our training time for our digit classifier was greatly decreased.

	Milestone 2	Atomic Add + Unrolling	Speed up
Training time (s)	119.51	24.11	4.59x
Accuracy for 5 iterations	96.11	96.11	--

Conclusion

For milestone 3, we optimized the training performance of our handwritten digit classifier by 79%. 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. Additionally, flattening/unrolling the for-loops in our convolution kernels also yielded a significant performance gain over our basic implementation for milestone 2. Moving forward, we aim to investigate and quantify the effects of each of these optimizations in our final report. Notably, we will discuss the reasoning for any performance gains or losses introduced by each implementation.

Reference:

[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/