ece408 Milestone 4

Andrew Smith, Ann Thomas, Nick Cebry

October 12^{th} 2019

1 Info

Table 1: Team Information

Name	Andrew Smith	Ann Thomas	Nick Cebry
NetID	atsmith3	ahthoma2	ncebry2
RAI ID	5d97b20e88a5ec28f9cb94e0	5d97b2190fb81f0e26d4f412	5d97b1a788a5ec28f9cb9421
Team Names	little_computer_parallel		
School Affiliation	UIUC Campus		

2 Results (Milestone 2)

2.1 Top Kernels by Execution Time

1. `volta_scudnn_128x64_relu_interior_nn_v1`
2. `volta_sgemm_128x128_tn`
3. `volta_gcgemm_64x32_nt`
4. `void cudnn::detail::pooling_fw_4d_kernel`
5. `void op_generic_tensor_kernel`
6. `void fft2d_c2r_32x32`
7. `void fft2d_r2c_32x32`
8. `void mshadow::cuda::MapPlanLargeKernel`
9. `void mshadow::cuda::SoftmaxKernel`

2.2 Top API Calls by Execution Time

1. `cudaStreamCreateWithFlags`
2. `cudaMemGetInfo`
3. `cudaFree`
4. `cudaMemcpy2DAsync`
5. `cudaStreamSynchronize`
6. `cudaEventCreate`
7. `cudaMemcpy`
8. `cudaHostAlloc`
9. `cuDeviceGetName`

2.3 Kernel and API difference

The difference between an API call and a Kernel is that an Kenerls are launched on the GPU and execute on the Stream Multiprocessors (SMs). API calls are something like cudaMemcpy(...) where it interfaces with the cuda libraries and even the hardware but doesn't directly run work on the SMs.

2.4 MXNet CPU

2.4.1 Output

```
Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
16.97user 4.74system 0:08.92elapsed 243%CPU (0avgtext+0avgdata 6044568maxres
ident)k
0inputs+2616outputs (0major+1601016minor)pagefaults 0swaps
```

2.4.2 Run Time

According to the output of the CPU run above it took `0:08.92` seconds of system time to complete with an accuracy of `0.8154`.

2.5 MXNet GPU

2.5.1 Output

```
Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
5.04user 3.22system 0:04.73elapsed 174%CPU (0avgtext+0avgdata 2954432maxresident)k
0inputs+4536outputs (0major+732074minor)pagefaults 0swaps
```

2.5.2 Run Time

According to the output of the GPU run above it took `0:04.73` seconds of system time to complete with an accuracy of `0.8154`.

2.6 CPU Implementation

See `ece408_src/new_forward.h` for the implementation.

2.6.1 Output

```
Running /usr/bin/time python m2.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.107684
Op Time: 0.587626
Correctness: 0.76 Model: ece408
4.55user 2.90system 0:01.86elapsed 399%CPU (0avgtext+0avgdata 309340maxresident)k
0inputs+2824outputs (0major+111706minor)pagefaults 0swaps

Running /usr/bin/time python m2.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 1.054678
Op Time: 5.946515
Correctness: 0.767 Model: ece408
12.07user 3.23system 0:08.43elapsed 181%CPU (0avgtext+0avgdata 829884maxresident)k
0inputs+0outputs (0major+305347minor)pagefaults 0swaps

Running /usr/bin/time python m2.1.py 10000
Loading fashion-mnist data... done
Loading model... done
```

```
New Inference
Op Time: 10.827225
Op Time: 62.795590
Correctness: 0.7653 Model: ece408
89.85user 11.82system 1:17.98elapsed 130%CPU (0avgtext+0avgdata 6043320maxresident)k
0inputs+0outputs (0major+2299977minor)pagefaults 0swaps
```

2.6.2 Program Execution Time

Table 2: Program Execution Time

Batch Size (Images)	User Time	System Time	Elapsed Time
100	4.55	2.90	1.86
1000	12.07	3.23	8.43
10000	89.85	11.82	17.98

2.6.3 Operation Times

Table 3: Operation Execution Time

Batch Size (Images)	Operation 1	Operation 2
100	0.107684	0.587626
1000	1.054678	5.946515
10000	10.827225	62.795590

3 Milestone 3

3.1 Accuracy

Table 4: Accuracy

Batch Size (Images)	Accuracy
100	0.76
1000	0.767
10000	0.7653

3.2 Operation Times

Table 5: Operation Execution Time

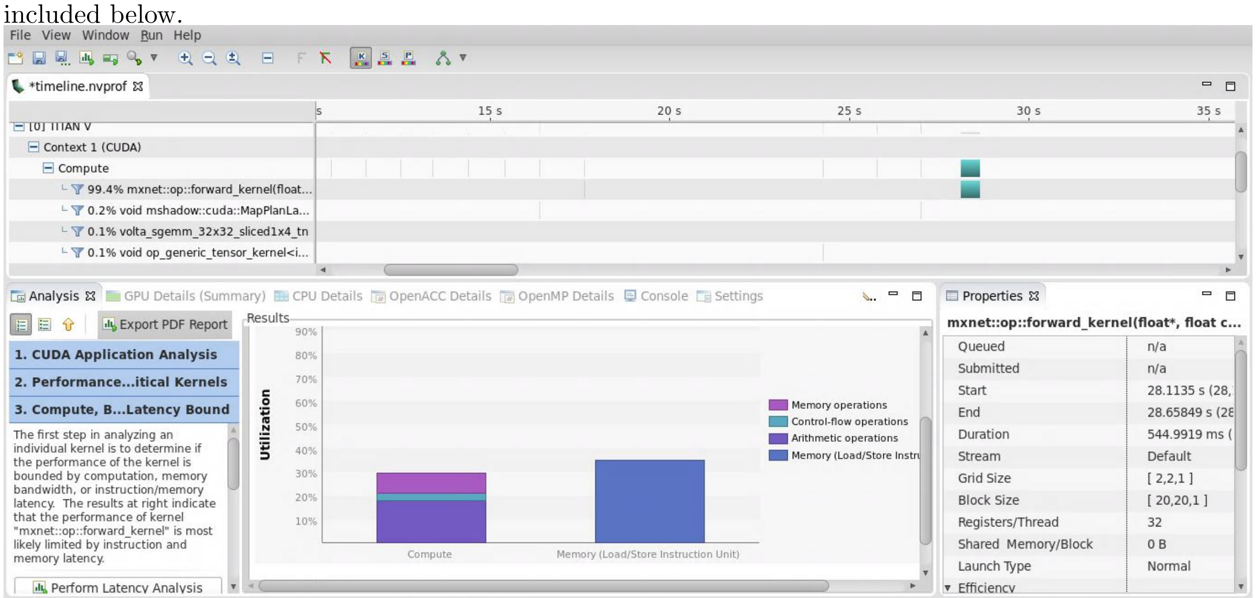
Batch Size (Images)	Operation 1	Operation 2
100	0.002426	0.060505
1000	0.023602	0.561961
10000	0.221924	5.527549

Table 6: Program Execution Time

Batch Size (Images)	User Time	System Time	Elapsed Time
100	4.86	2.88	4.79
1000	5.12	2.48	5.08
10000	9.06	4.31	10.23

3.3 NVPROF (1000)

A screenshot of the NVVP program showing the execution times of the kernel for batch size of 1000 is included below.



4 Milestone 4

4.1 Overview

For milestone 4 we had to take our baseline GPU implementation from the previous milestone and add three optimizations to the convolution kernel. The three optimizations we decided to implement were 1. *loop unrolling* 2. *constant memory* and 3, *shared memory.* In each of the following sections we will describe the optimization's implementation details and provide and analysis of the performance impact with respect to the baseline.

4.2 Optimization: Loop Unroll

Loop unrolling is reducing the number of iterations of the loop and then doing more sequential operations per loop body. This improves the threads throughput. We applied the loop unrolling to the two inner loops of the convolution which do the multiplication and accumulation of the kernel w with the input tensor x. We found that this optimization was provided the greatest increase in performance.

4.2.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.001475	0.004636	0.76
1000	0.016734	0.048412	0.767
10000	0.153228	0.456949	0.7653

4.3 Optimization: Storing Kernel in Constant Memory

We chose to store the kernel in constant memory. Constant memory has an access time of 5 cycles, whereas global memory has an access time of 500 cycles. Although loading into constant memory has an associated overhead time, it is still less than the 100x increase we get from using constant memory. We used constant memory over shared memory because it is read-only, and we never want to write to the kernel.

4.3.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.002029	0.005533	0.76
1000	0.020372	0.055014	0.767
10000	0.203387	0.500582	0.7653

4.4 Optimization: Shared Memory Input Tensor

We used shared memory on the input tensor as an optimization. We loaded the input tensor into shared memory from global memory. That way, the thread block would only have to perform a shared memory access instead of a global memory access. The access to shared memory is about 100x faster than the access to global memory.

4.4.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.001876	0.005113	0.76
1000	0.018391	0.050340	0.767
10000	0.177534	0.456385	0.7653

4.5 All 3 Optimizations

Finally, we combined the loop unrolling, shared memory and constant memory optimizations into one kernel. Combining the three optimizations resulted in the fastest op1 and op2 times and was significantly faster than the baseline.

4.5.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.001353	0.003711	0.76
1000	0.013015	0.036563	0.767
10000	0.129487	0.331177	0.7653

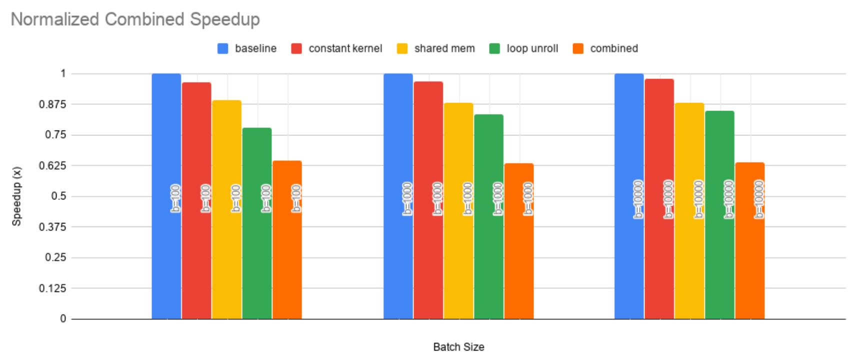


Figure 1: Normalized Speedup for Individual and Combined Optimizations

4.5.2 NVVP Milestone 4

All of the following NVVP figures were generated with all 3 of the milestone 4 optimizations enabled. The Forward pass layer was run with a batch size of 10000 on a Nvidia Titan V GPU. For the final set of optimizations there are some things we need to focus on improving. We have mediocre Load-Store Efficiency and mediocre Warp Execution Efficiency (Figure 2). After analyzing the memory utilization we see that a majority of our bandwidth goes to the shared memory (Figure 4) verifying our Shared-Memory optimization. Anoother area to be addressed is there is a vast majority of stalls due to Synchronization. If we can figure out how to relax the synchronization requirement between threads, then we could dramatically improve the execution time (Figures 6, 7).

mxnet::op::forward_kernel(float*, float ...

Queued	n/a
Submitted	n/a
Start	3.627 s (3,626,646,397 ns)
End	3.744 s (3,744,213,993 ns)
Duration	117.568 ms (117,567,596 ns)
Stream	Default
Grid Size	[9,9,1]
Block Size	[12,12,7]
Registers/Thread	28
Shared Memory/Block	3.938 KiB
Launch Type	Normal
Efficiency	
Global Load Efficiency	62.5%
Global Store Efficiency	n/a
Shared Efficiency	60.6%
Warp Execution Efficiency	71.1%
Not-Predicated-Off Warp Exe	61.7%
Occupancy	
Achieved	50.7%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Figure 2: NVVP Overall Stats

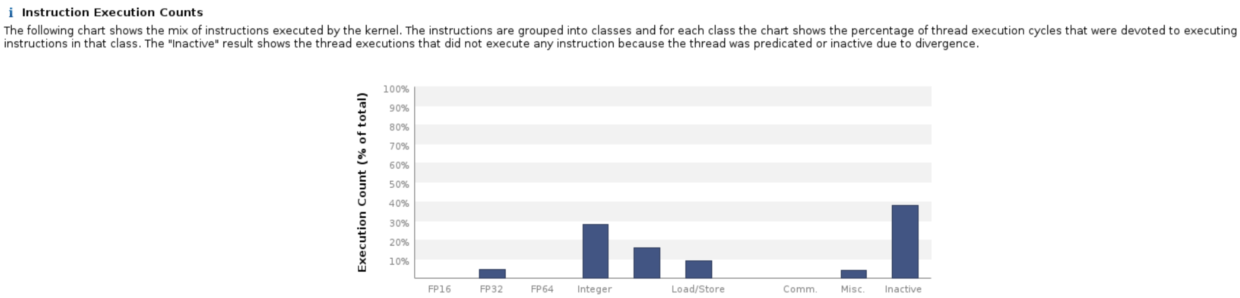


Figure 3: Instruction Execution Counts

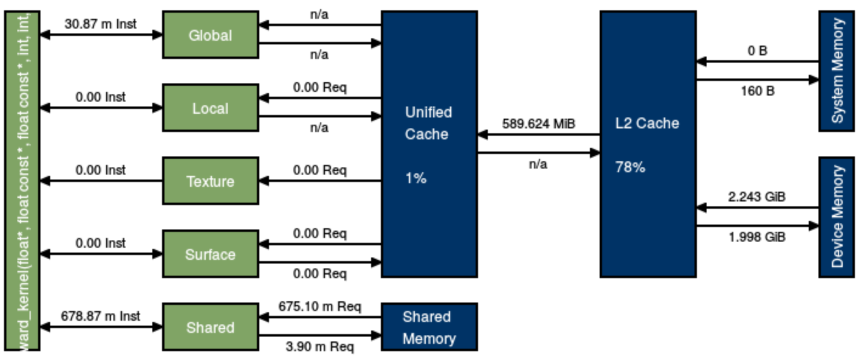
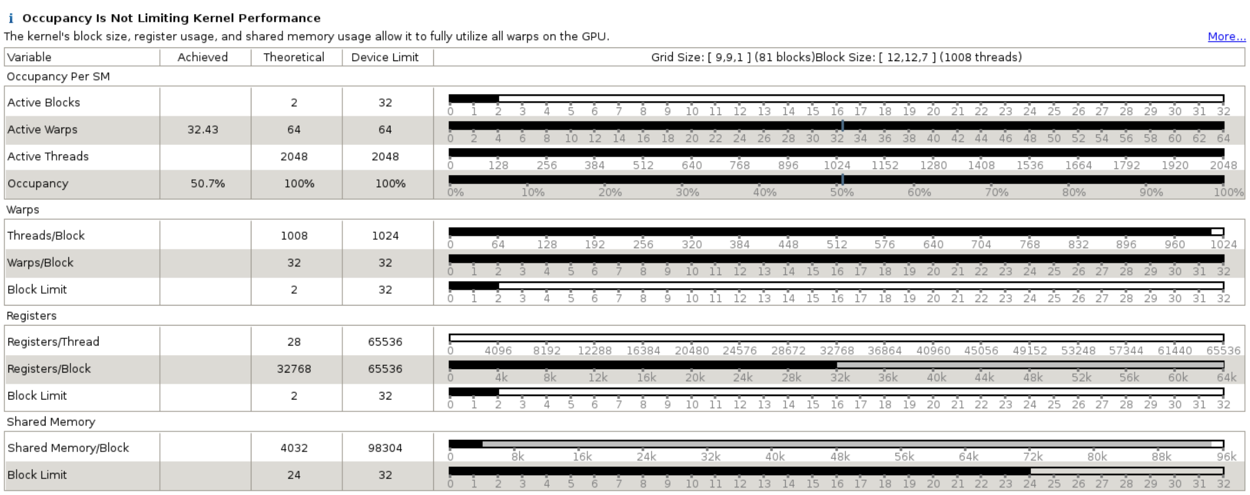


Figure 4: Memory Stats



Variable	Achieved	Theoretical	Device Limit	Grid Size: [9,9,1] (81 blocks)Block Size: [12,12,7] (1008 threads)
Occupancy Per SM				
Active Blocks		2	32	
Active Warps	32.43	64	64	
Active Threads		2048	2048	
Occupancy	50.7%	100%	100%	
Warps				
Threads/Block		1008	1024	
Warps/Block		32	32	
Block Limit		2	32	
Registers				
Registers/Thread		28	65536	
Registers/Block		32768	65536	
Block Limit		2	32	
Shared Memory				
Shared Memory/Block		4032	98304	
Block Limit		24	32	

Figure 5: Occupancy

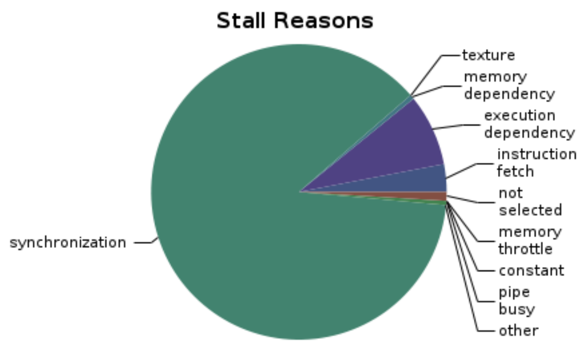


Figure 6: Stall Reasons

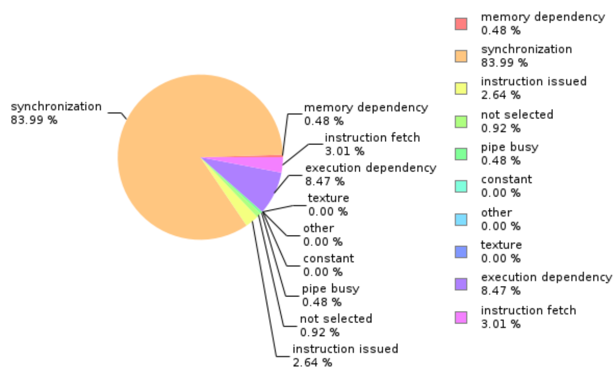


Figure 7: PC Distribution

Line / File	new-forward.cuh - /mxnet/src/operator/custom
38	Divergence = 4.4% [1150000 divergent executions out of 25920000 total executions]
43	Divergence = 8.7% [27000000 divergent executions out of 311040000 total executions]

Figure 8: Divergence