ECE408 Project
Final Submission

Yutong Xie(yutongx6); Yunyi Zhang(yunyiz3); Pu Jian(pujian2)
Team names: hhh
UIUC Campus students
Instructor: Professor Wen-mei Hwu

December 17, 2019

1 Introduction

This is final submission report for ECE408 group project. In this project, we will get practical experience by using, profiling, and modifying MXNet, a standard open-source neural-network framework. Also, we need to demonstrate command of CUDA and optimization approaches by designing and implementing an optimized neural-network convolution layer forward pass.

For this milestone, we implement four different optimization methods and we will analyze them in the following report. First, we use the unroll and shared-memory matrix multiply method to optimize the kernel. Then, we tried to conduct kernel fusion for unrolling and multiply to speed up more. Third, we sweep the CUDA parameters to find the best values. Finally, we realize different implementation in kernels with different layer sizes.

In our code base, there are totally 8 .cuh file. The new-forward.cuh file is our best performance optimization and is also the optimization 7. New-forward1,2,3,4,5 have the code for five different optimizations. For optimization 6, we change the TILE_WIDTH manually and run the code based on optimization 5, so there is no single cuh file for this optimization. Also, in milestone 4, we implemented three different optimizations in three different cuh file, and we combine them into new-forward123.cuh in the code folder.

The operation time for these four optimizations is shown blow.

2 Optimization 4: Unroll + shared-memory Matrix multiply

2.1 Describe the optimization

In this part, we review the chapter 16 of the textbook. Like the code proposed in the textbook, we first used one kernel to make the input matrix X to X_unroll, which makes it will be much easier to access for the GPU. Secondly, we launched one kernel to do the matrix multiplication like previous optimization.

The op time results can be seen in the Fig 1. Compared with the baseline running time in the milestone3, using this "optimization" method the running time becomes about 10 times slower. We used the nvprof file to find the reason.

Optimization	data set size = 100		data set size = 1000		data set size = 10000	
	layer1	layer2	layer1	layer2	layer1	layer2
No optimization(milestone3)	0.000263	0.000903	0.002959	0.009841	0.03072	0.09796
Unroll + shared-memory Matrix multiply	0.002774	0.001791	0.015791	0.024462	0.130555	0.218353
Kernel Fusion for Unroll and Matrix Multiplication	0.000236	0.000739	0.002184	0.007172	0.02155	0.071317
Sweeping Parameters	0.000257	0.000846	0.002324	0.008654	0.021397	0.067132
Multiple kernel implementations for different layer sizes	0.000247	0.000569	0.002182	0.005318	0.019266	0.047179

Figure 1: Operation time for different optimization.

2.2 Demonstrate nvprof profiling and analyze optimization

Like previous optimizations we used the nvprof [U+FB01]le of this optimization to get more detailed data of the performance of this method. We add nvprof command to the yml [U+FB01]le and run the program. The result of the running commands is shown above.

In this optimization, we used 2 kernel functions in total, so we have 2 analysis files. First of all, we analyzed the kernel to unroll the matrix X to X_unroll.

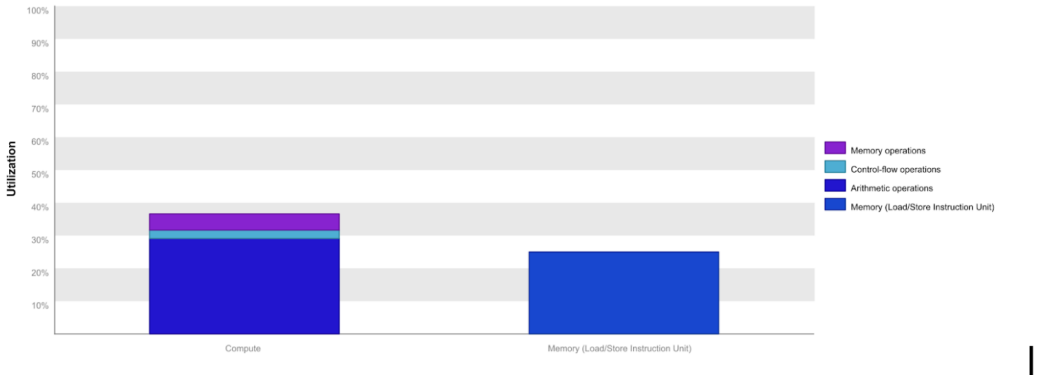


Figure 2: Utilization chart for unroll matrix.

These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations.

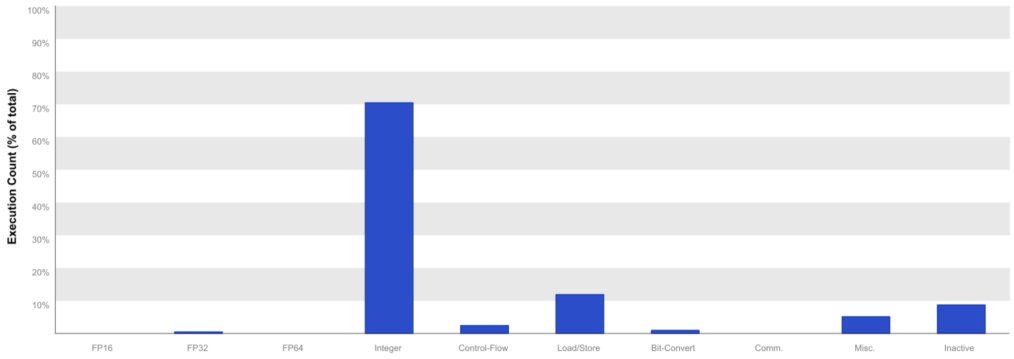


Figure 3: Execution count for kernel 1.

In this kernel, since we need to re-mapping the input matrix X to X_unroll, we implemented a lot of divide and mod operations to the original data in the X matrix, which may not be the optimal operation for the GPU to handle. This may be one possible reason why the kernel performance is slower.

In the second kernel of this optimization, we do the basic matrix multiplication based on the unrolled input matrix X_unroll. Like the first kernel, the load/store part takes a lot of time, which makes us come up one idea that doing all these things in only one kernel, which will help us save the loading and storing time.

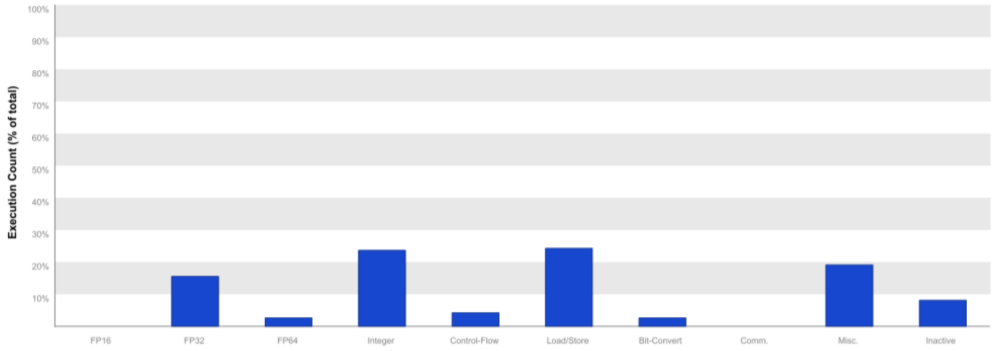


Figure 4: Execution count for multiply kernel.

3 Optimization 5: Kernel fusion for unrolling and matrix-multiplication

3.1 Describe the optimization

In this optimization, we merge the unrolling kernel and matrix multiply kernel in the previous optimization into one single kernel. In the kernel, we set up two shared memory, one for weighted matrix and the other for input feature. Then, we load data into shared memory and do the matrix multiplication. Finally, we write the results into output.

3.2 Correctness and timing with 3 different dataset sizes

The Op Time for different data set is shown in Fig 1 ,and our implementation of the optimization 3 has achieved the expected correctness values. There are different degrees of reduction in the running time of each data set (10000, 1000, 100) which means our implementation of kernel fusion increases the performance of forward convolution.

3.3 Demonstrate nvprof profiling and analyze optimization

Compared the Op time between this optimization and optimization 4, we can see that there is significant improvement in both convolution layer. In optimization 4, the input and weighted matrix should be loaded by multiple kernels and then the results would be written back to global memory multiple times. The redundant operation causes the increasing of operation time. However, in the kernel fusion version of code, there is only one kernel, which means that the input and output are read and written once. Hence, the Op time decreases a lot.

Also, we used the nvprof file of this optimization to get more detailed data of the performance of this method. We add **nvprof** command to the yml file and run the program.

```
- nvprof -o timeline.nvprof python m3.1.py 10000
- nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python
    m3.1.py 10000
```

By comparing the multiprocessor utilization for optimization 4 and this optimization, we found that the utilization for kernel fusion optimization has significant promotion as shown in the following two diagram. This can explain why the performance of this optimization becomes better.

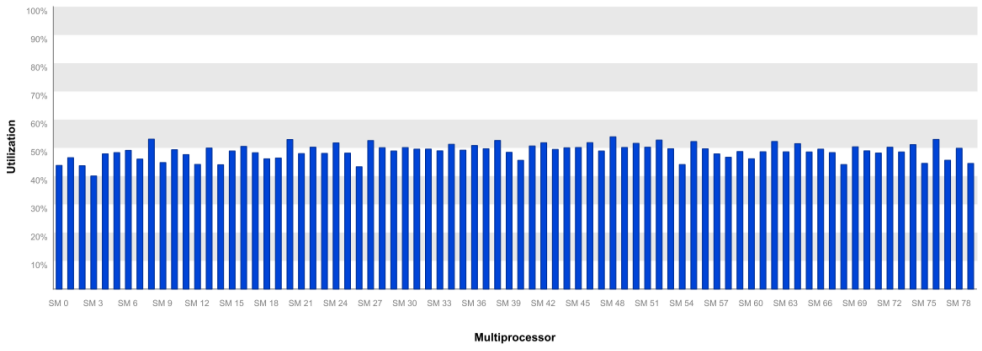


Figure 5: Multiprocessor utilization chart for optimization 4.

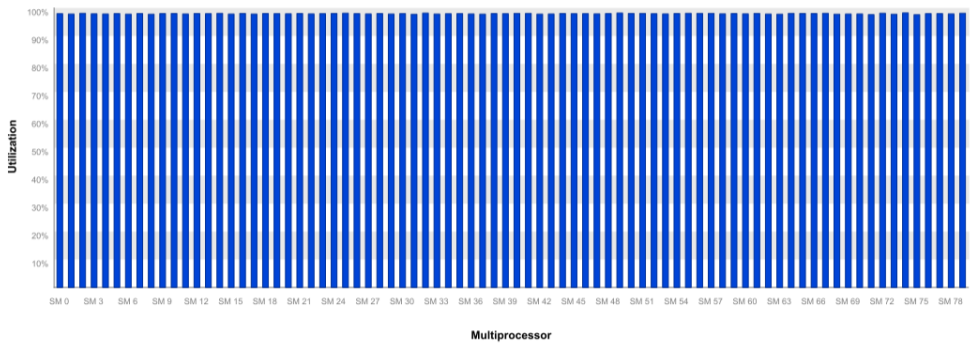


Figure 6: Multiprocessor utilization chart for optimization 5.

4 Optimization 6: Sweeping various parameters to find best values

4.1 Describe the optimization

In this version of optimization, we sweep various parameters to find the best values that give the best performance. Since the optimization 5 performs best among all previous optimizations, we sweep parameters to find the optimal solution based on the kernel fusion for unrolling and matrix multiplication.

We used different block sizes and shared memory sizes by change the value of TILE_SIZE of our code for different data set size (100, 1000, 10000).

4.2 Correctness and timing with 3 different dataset sizes

The best Op Time for this optimization is shown in Fig 1. All the correctness keep same with the result in milestone3 milestone4.

4.3 Demonstrate nvprof profiling and analyze optimization

Here, we analyze the performance based on the experiment using data set with size of 10000. Due to the max thread in one block is 1024, we set the TILE_WIDTH from 4 to 32 with interval of 2. And we plot the total time with TILE_WIDTH as shown in following figure. From the figure, we can see that when TILE_WIDTH equals to 16, the convolution achieves the best performance.

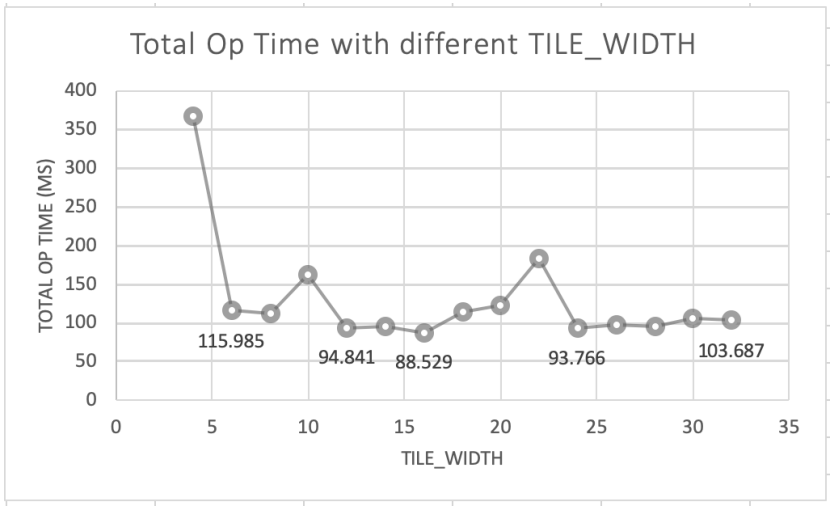


Figure 7: Total Op time for different TILE WIDTH.

Except for the total Op time, we also test the time for two different layers and plot the result.

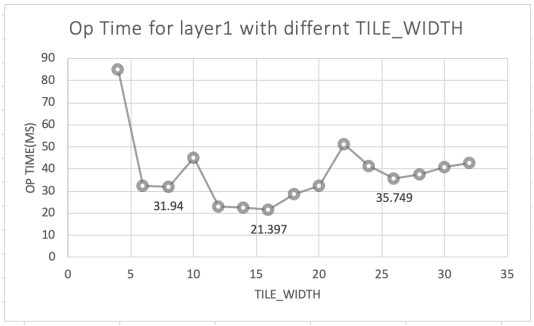


Figure 8: Op Time for layer 1 under different tile width

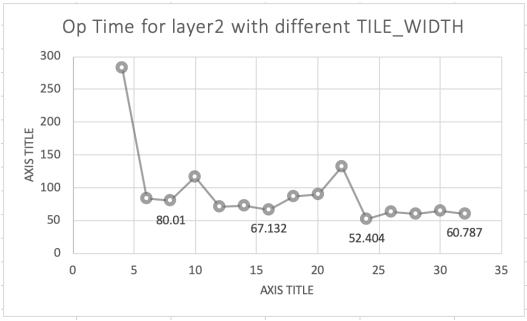


Figure 9: Op Time for layer 2 under different tile width

From the figure above, we can easily find that the TILE_WIDTH should be set to 16 for the first kernel and 24 for the second kernel. For the first layer, the output channel is 12. However if we set the TILE WIDTH as 12, there will be total $144/32 = 4.5$ warps in each block, which is not very efficient. Hence, 16 should be the best choice. For the second layer, M is 24 and there are total 18 warps in one block, which can be fully utilized. And the occupancy approximates to 100% for this kernel when tile width is 24 as shown in following figure.

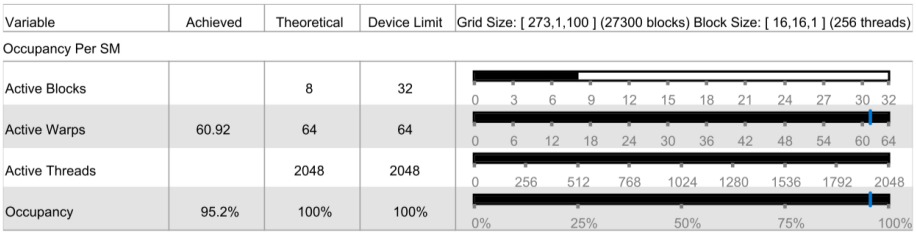


Figure 10: Execution status for the second kernel.

5 Optimization 7: Multiple kernel implementations for different layer sizes

5.1 Describe the optimization

In previous optimizations, we found that the M size of the 2 layers is different. So we implemented 2 different kernels with different TILE_SIZE to help us improve the performance. We tried to print out the M size of each layer, the results is listed below.

```
M,C,H,W,K is: 12,1,70,70,5
M,C,H,W,K is: 24,12,33,33,5
```

So we implemented 2 different kernels to handle the different layers. For the first layer, the M is 12, so we use 16 as the TILE_WIDTH; for the second layer, we firstly designed using 32 as the TILE_WIDTH, but in practice, we found that 24 have better performance.

5.2 Correctness and timing with 3 different dataset sizes

All the correctness keep same with the result in milestone3 milestone4. The Op time is shown in Fig 1.

5.3 Demonstrate nvprof profiling and analyze optimization

In this optimization, we designed 2 kernels, one with tile size is 16 and 24 in the other. In the first kernel, the nvprof analysis is shown below:

Duration	202.074 μs
Grid Size	[273,1,100]
Block Size	[16,16,1]
Registers/Thread	32
Shared Memory/Block	2 KiB
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Figure 11: Basic Information for kernel 1.

This part is the same as the analysis of the optimization 5, since we didn't change the kernel code for this part. So the detailed analysis is the same as the analysis in the optimization.

In the second kernel, we use the 24 as the TILE_WIDTH, which enables the performance to be improved.

Duration	490.268 μs
Grid Size	[36,1,100]
Block Size	[24,24,1]
Registers/Thread	32
Shared Memory/Block	4.5 KiB
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Figure 12: Basic Information for kernel 2.

Compared with the first kernel in this optimization, the utilization of the GPU resource is slightly decreased. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system.

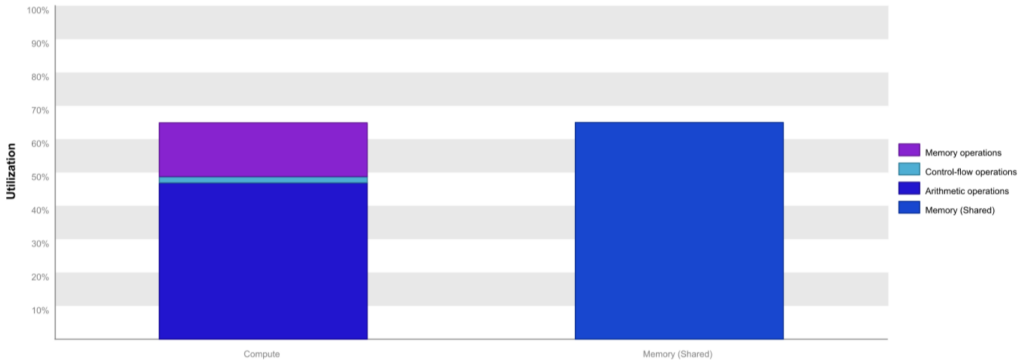


Figure 13: Utilization chart for optimization 7.

In this following, the data indicates that the shared memory and the cache is the main memory in this kernel.

Transactions	Bandwidth	Utilization	
Shared Memory			
Shared Loads	30432352	7,945.33 GB/s	
Shared Stores	3398064	887.172 GB/s	
Shared Total	33830416	8,832.502 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	981586	64.069 GB/s	
Writes	369916	24.145 GB/s	
Total	1351502	88.213 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	7574318	494.379 GB/s	
Global Stores	369900	24.144 GB/s	
Texture Reads	32681471	8,532.534 GB/s	
Unified Total	40625689	9,051.056 GB/s	Idle Low Medium High Max
Device Memory			
Reads	164408	10.731 GB/s	
Writes	252897	16.507 GB/s	
Total	417305	27.238 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	326.352 kB/s	Idle Low Medium High Max

Figure 14: Memory bandwidth used by this kernel.

In the control divergence part, since we utilized the larger tile size for the larger M size, the control

divergence becomes very small in this part. And the code causing the control divergence is shown below.

/mxnet/src/operator/custom/./new-forward.cuh

Line 127	Divergence = 0% [0 divergent executions out of 64800 total executions]
Line 129	Divergence = 0% [0 divergent executions out of 64800 total executions]
Line 129	Divergence = 0% [0 divergent executions out of 842400 total executions]
Line 155	Divergence = 2.7% [22500 divergent executions out of 842400 total executions]
Line 173	Divergence = 2.8% [1800 divergent executions out of 64800 total executions]
Line 175	Divergence = 0% [0 divergent executions out of 64800 total executions]

Figure 15: Control Divergence this kernel.

Transactions	Bandwidth	Utilization	
Shared Memory			
Shared Loads	30432352	7,945.33 GB/s	
Shared Stores	3398064	887.172 GB/s	
Shared Total	33830416	8,832.502 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	981586	64.069 GB/s	
Writes	369916	24.145 GB/s	
Total	1351502	88.213 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	7574318	494.379 GB/s	
Global Stores	369900	24.144 GB/s	
Texture Reads	32681471	8,532.534 GB/s	
Unified Total	40625689	9,051.056 GB/s	Idle Low Medium High Max
Device Memory			
Reads	164408	10.731 GB/s	
Writes	252897	16.507 GB/s	
Total	417305	27.238 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	326.352 kB/s	Idle Low Medium High Max

Figure 16: Memory bandwidth used by this kernel.

ECE408 Project
Milestone4

Yutong Xie(yutongx6); Yunyi Zhang(yunyiz3); Pu Jian(pujian2)
Team names: hhh
UIUC Campus students
Instructor: Professor Wen-mei Hwu

November 21, 2019

1 Introduction

This is milestone4 report for ECE408 group project. In this project, we will get practical experience by using, profiling, and modifying MXNet, a standard open-source neural-network framework. Also, we need to demonstrate command of CUDA and optimization approaches by designing and implementing an optimized neural-network convolution layer forward pass.

For these milestone, we implement three different optimization methods and we will analyze them in the following report. First, we put the input image and weight matrix into the shared memory to reduce the access to the global memory. Second, we try to move the weight matrix to the constant memory, because they won't change during computation. Finally, we implement the loop unrolling to improve the efficiency of computation.

2 Optimization 1: convolution using shared memory

2.1 Describe the optimization

In this part we review the chapter 16 to recall the case study about this field. We design to put the input data and the weighted matrix to the shared memory. And also modified the code provided in the chapter16 to get all the task done.

2.2 Correctness and timing with 3 different dataset sizes

In the following sheet, we can see the op time and correctness for different data set. All the correctness keep same with the result in milestone3.

Compared with the baseline of the optimizations (result in milestone3) we can get that after using the shared memory to handle the convolution in this milestone, the correctness keeps the same. While the time of the operation will be slower than what we got in milestone 3 both in the 1st layer and the second layer.

data set size	OP Time (with the optimization)	Op Time(milestone3)	Correctness
10000	0.042111	0.030720	0.76
	0.122195	0.097960	
1000	0.004204	0.002959	0.767
	0.013026	0.009841	
100	0.000395	0.000263	0.7653
	0.001295	0.000903	

Figure 1: Test Result of shared optimization1.

2.3 Demonstrate nvprof profiling and analyze optimization

Also, we used the nvprof file of this optimization to get more detailed data of the performance of this method. We add **nvprof** command to the yml file and run the program.

```
- nvprof -o timeline.nvprof python m3.1.py 100
- nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python
    m3.1.py 100
```

The command will generate timeline.nvprof and *analysis.nvprof. –analysis-metrics significantly slows the run time, so we use 100 datasets during profiling.

```
* Running nvprof -o timeline.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==526== NVPROF is profiling process 526, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.000409
Op Time: 0.001299
Correctness: 0.76 Model: ece408
==526== Generated result file: /build/timeline.nvprof
* Running nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==621== NVPROF is profiling process 621, command: python m3.1.py 100
Loading model... done
New Inference
==621== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)" (done)
Op Time: 1.425146
Op Time: 0.001374
Correctness: 0.76 Model: ece408
```

Figure 2: Result of nvprof profiling for optimization1.

We import nvprof files into nvvp to analyze our optimization method. From the utilization chart (Fig 18), we can easily notice that the utilization of memory is significantly lower than the utilization of compute, which means the limitation of memory access is of less importance after using shared memory.

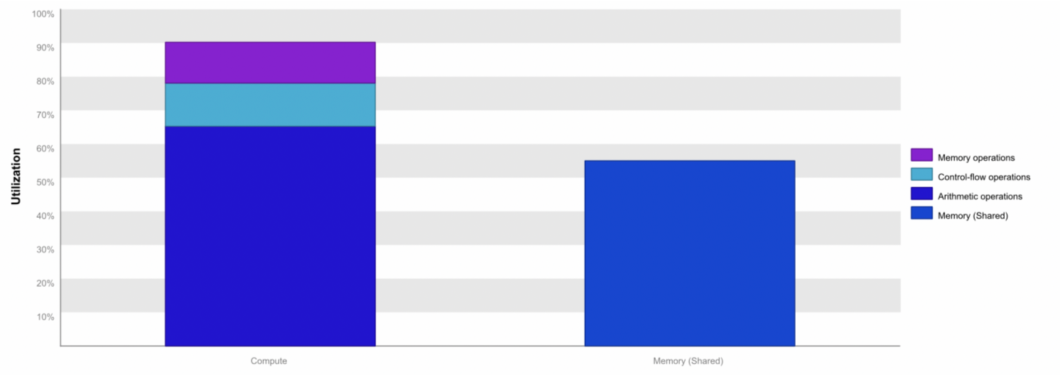


Figure 3: Utilization chart for optimization1.

For the memory statics, the emphasis of bandwidth and utilization shift from global memory to shared memory. And the decrease of bandwidth and utilization of global memory is significant. (from 11430 GB/s to 5347 GB/s) A great improvement in efficiency of memory access is provided by using shared memory.

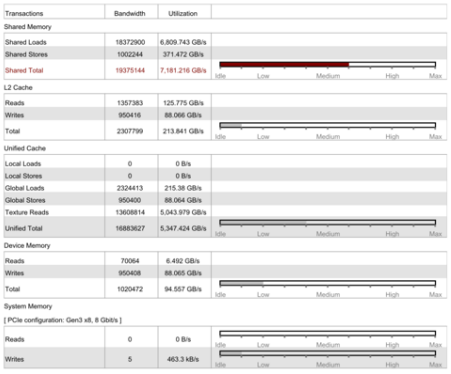


Figure 4: Memory usage highlighting with shared memory.

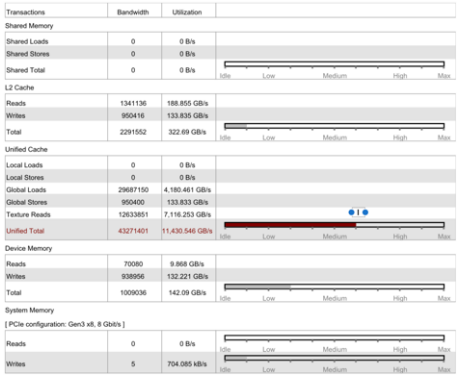


Figure 5: Memory usage highlighting without shared memory.

According to the running time in 2.2, it seems strange that, after implemented shared memory, our kernel runs slower than previous one. The reason here is we introduce much divergence, according to the control divergence report from nvprof, some line of algorithm with shared memory have 100% control divergence, whereas the traditional approach only has one line with 16.5% control divergence.

/mxnet/src/operator/custom/./new-forward.cuh

Line 45	Divergence = 0% [0 divergent executions out of 240000 total executions]
Line 45	Divergence = 0% [0 divergent executions out of 240000 total executions]
Line 46	Divergence = 37.5% [90000 divergent executions out of 240000 total executions]
Line 51	Divergence = 0% [0 divergent executions out of 240000 total executions]
Line 51	Divergence = 0% [0 divergent executions out of 300000 total executions]
Line 52	Divergence = 0% [0 divergent executions out of 300000 total executions]
Line 52	Divergence = 100% [300000 divergent executions out of 300000 total executions]
Line 52	Divergence = 0% [0 divergent executions out of 300000 total executions]
Line 52	Divergence = 100% [300000 divergent executions out of 300000 total executions]
Line 52	Divergence = 0% [0 divergent executions out of 300000 total executions]
Line 62	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 62	Divergence = 0% [0 divergent executions out of 240000 total executions]
Line 63	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 63	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 63	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 63	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 63	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 63	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 65	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 65	Divergence = 0% [0 divergent executions out of 1200000 total executions]
Line 71	Divergence = 16.5% [39600 divergent executions out of 240000 total executions]
Line 78	Divergence = 0% [0 divergent executions out of 198000 total executions]

Figure 6: Control divergence with shared memory for optimization1.

/mxnet/src/operator/custom/./new-forward.cuh

Line 30	Divergence = 16.5% [39600 divergent executions out of 240000 total executions]
Line 32	Divergence = 0% [0 divergent executions out of 198000 total executions]
Line 32	Divergence = 0% [0 divergent executions out of 198000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 198000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 45	Divergence = 0% [0 divergent executions out of 198000 total executions]

Figure 7: Control divergence without shared memory for optimization1.

The first control divergence occurs when we try to make sure the input is in the range. While in the line 52 its also introduce the extremely high control divergence, which will hurt the program performance.

```
46      if ((h0<K) && (w0<K)){
47      W_shared[h0*K+w0]=k4d(m,c,h0,w0);
48      }
49      __syncthreads();
50
51      for (int i=h; i<h_base+X_out_width; i+=TILE_WIDTH){
52          for (int j=w; j<w_base+X_out_width; j+=TILE_WIDTH){
53              if (i<H && j<W){
54                  X_shared[(i-h_base)*(X_out_width)+(j-w_base)]=x4d(n,c,i,j);
55              }
56              else{
57                  X_shared[(i-h_base)*(X_out_width)+(j-w_base)]=0;
58              }
59          }
60      }
```

Figure 8: Code of shared memory for optimization1.

In conclusion, although using shared memory can reduce large amount of global memory access, it introduces more control divergence which will slow down the running speed.

3 Optimization 2: Weight matrix in constant memory

3.1 Describe the optimization

During the process of doing this coding work, we found that the weighted matrix of this project is never change. So we try to move it to the constant memory to help speed up the operation time and the help get full use of the bandwidth.

3.2 Correctness and timing with 3 different dataset sizes

Our implementation of the optimization 2 has achieved the expected correctness values. There are different degrees of reduction in the running time of each data set (10000, 1000, 100) which means our implementation of constant memory increase the efficiency of the algorithm in memory loading.

data set size	OP Time (with the optimization)	Op Time(milestone3)	Correctness
10000	0.030513	0.030720	0.76
	0.094252	0.097960	
1000	0.002972	0.002959	0.767
	0.009492	0.009841	
100	0.000272	0.000263	0.7653
	0.000856	0.000903	

Figure 9: Test Result of shared optimization2.

3.3 Demonstrate nvprof profiling and analyze optimization

Also, we used the nvprof file of this optimization to get more detailed data of the performance of this method. We add **nvprof** command to the yml file and run the program.

```
- nvprof -o timeline.nvprof python m3.1.py 100
- nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python
    m3.1.py 100
```

The command will generate timeline.nvprof and *analysis.nvprof. –analysis-metrics significantly slows the run time, so we use 100 datasets during profiling.

```
* Running nvprof -o timeline.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==525== NVPROF is profiling process 525, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.000287
Op Time: 0.000861
Correctness: 0.76 Model: ece408
==525== Generated result file: /build/timeline.nvprof
* Running nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==620== NVPROF is profiling process 620, command: python m3.1.py 100
Loading model... done
New Inference
==620== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int, int)" (done)
Op Time: 1.265488
Op Time: 0.000998
Correctness: 0.76 Model: ece408
```

Figure 10: Result of nvprof profiling for optimization2.

The compute utilization of memory is largely reduced. And the utilization of memory cut down significantly, because the weight matrix is constant. Using constant memory to store it will save much cost of memory access.

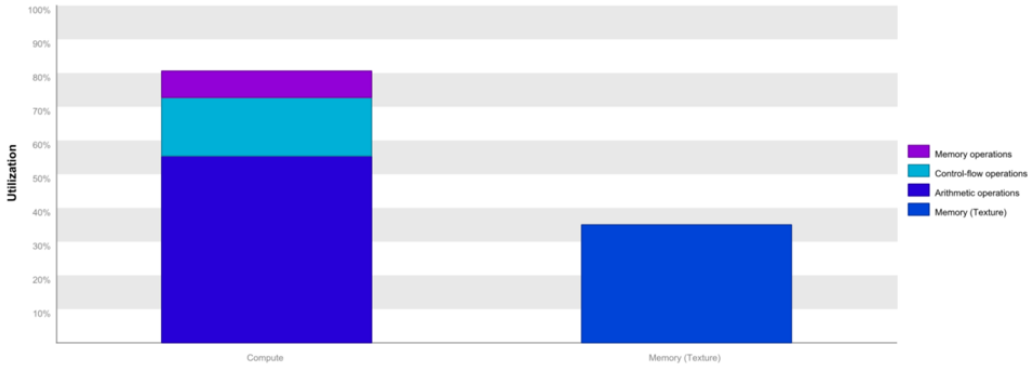


Figure 11: Utilization chart with constant memory.

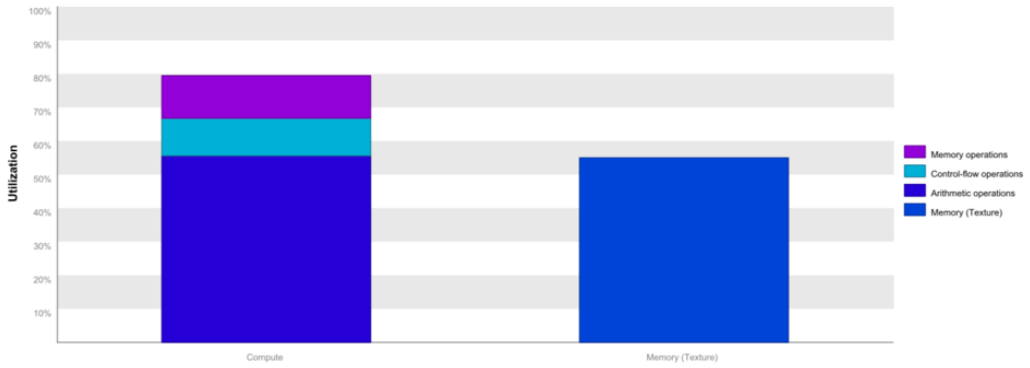


Figure 12: Utilization chart without constant memory.

From the point of memory statics, the constant memory help the kernel accessing much less global memory (from 11430GB/s to 8480GB/s), since elements in weight matrix can be accessed in constant memory. The time of accessing will be largely cut down.

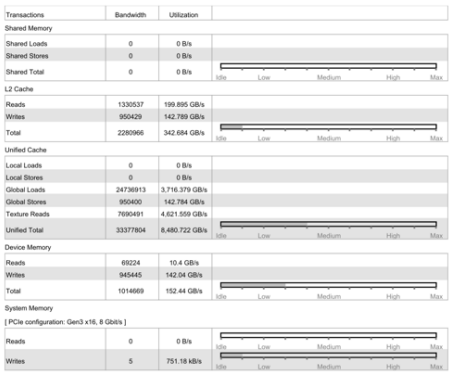


Figure 13: Memory usage highlighting with constant memory.

Figure 14: Memory usage highlighting without constant memory.

4 Optimization 3: loop unrolling

4.1 Describe the optimization

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions. Here we substitute a for loop in order to realize loop unrolling. The original approach is shown below.

```
float acc = 0;
  for (c = 0; c < C; c++) // sum over all input channels
    for (p = 0; p < K; p++) // loop over KxK filter
      for (q = 0; q < K; q++)
        acc += x4d(b, c, h + p, w + q) * k4d(m, c, p, q);

```

After using loop unrolling for inner two for loop, the method was rewritten.

```
float acc=0;
      for(int c=0;c<C;c++)
        acc+=x4d(n,c,h,w+0)*k4d(m,c,0,0)
            +x4d(n,c,h,w+1)*k4d(m,c,0,1)
            +x4d(n,c,h,w+2)*k4d(m,c,0,2)
            +x4d(n,c,h,w+3)*k4d(m,c,0,3)
            +x4d(n,c,h,w+4)*k4d(m,c,0,4)
             ......
            +x4d(n,c,h+4,w+4)*k4d(m,c,4,4);
```

4.2 Correctness and timing with 3 different dataset sizes

Our implementation of the optimization 2 has achieved the expected correctness values. There are reductions in the running time of each data set (10000, 1000, 100). The relative shorter time proof effectiveness of our loop unrolling algorithm.

data set size	OP Time (with the optimization)	Op Time(milestone3)	Correctness
10000	0.024599	0.030720	0.76
	0.082069	0.097960	
1000	0.002466	0.002959	0.767
	0.008270	0.009841	
100	0.000230	0.000263	0.7653
	0.000819	0.000903	

Figure 15: Test Result of shared optimization3.

4.3 Demonstrate nvprof profiling and analyze optimization

Also, we used the nvprof file of this optimization to get more detailed data of the performance of this method. We add **nvprof** command to the yml file and run the program.

```
- nvprof -o timeline.nvprof python m3.1.py 100
- nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python
    m3.1.py 100
```

The command will generate timeline.nvprof and *analysis.nvprof. –analysis-metrics significantly slows the run time, so we use 100 datasets during profiling.

```
* Running nvprof -o timeline.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==528== NVPROF is profiling process 528, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.000255
Op Time: 0.000836
Correctness: 0.76 Model: ece408
==528== Generated result file: /build/timeline.nvprof
* Running nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==623== NVPROF is profiling process 623, command: python m3.1.py 100
Loading model... done
New Inference
==623== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int, int)" (done)
Op Time: 1.002996al events
Op Time: 0.000946
Correctness: 0.76 Model: ece408
```

Figure 16: Result of nvprof profiling for optimization3.

For the baseline approach, all the variables generated in the for loop will be put into local memory. However, for the loop unrolling, they will be placed in registers, which increased the speed for using. When you unroll more loop, the increase will be more prominent.The utilization chart is quite different after applying loop unrolling. It cut down the utilization of computer but raise the utilization of memory. This kind of operation increase the efficiency of computer execution.

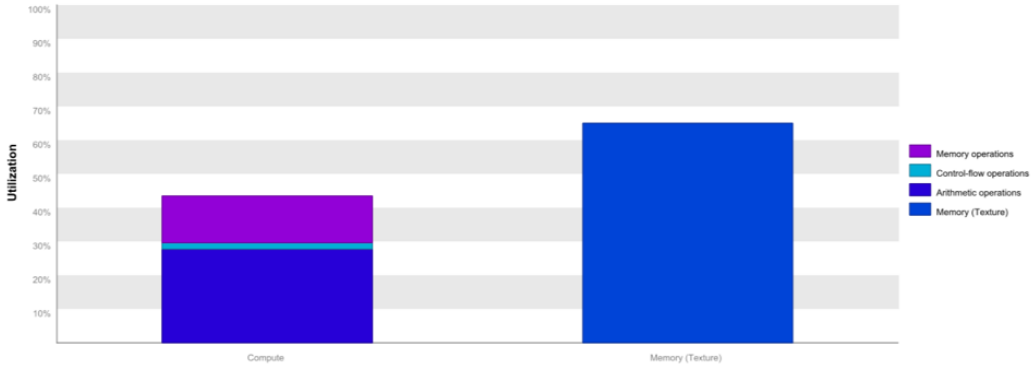


Figure 17: Utilization chart with constant memory.

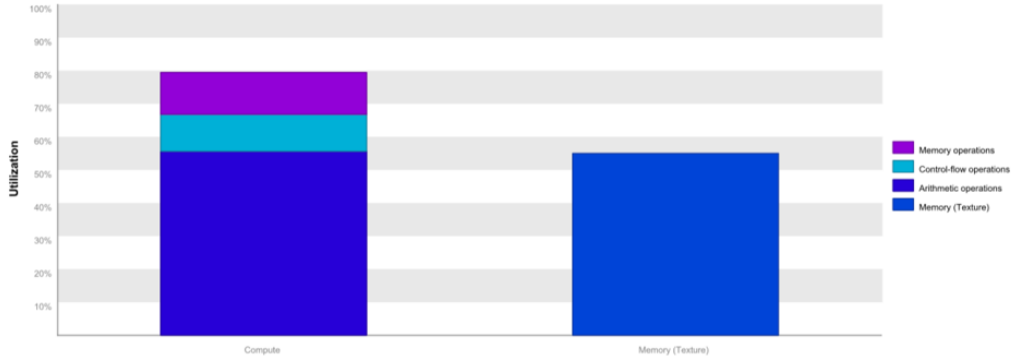


Figure 18: Utilization chart without constant memory.

ECE408 Project
Milestone3

Yutong Xie(yutongx6); Yunyi Zhang(yunyiz3); Pu Jian(pujian2)
Team names: hhh
UIUC Campus students
Instructor: Professor Wen-mei Hwu

October 19, 2019

1 Introduction

This is milestone3 report for ECE408 group project. In this project, we will get practical experience by using, profiling, and modifying MXNet, a standard open-source neural-network framework. Also, we need to emonstrate command of CUDA and optimization approaches by designing and implementing an optimized neural-network convolution layer forward pass.

2 Deliverables

2.1 Implement a GPU Convolution

We modify the file **new-forward.cuh** and realize the GPU convolution. Then, we run the m3.1.py program with three different data size and the result is shown below.

```
* Running python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.030720
Op Time: 0.097960
Correctness: 0.7653 Model: ece408
* Running python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.002959
Op Time: 0.009841
Correctness: 0.767 Model: ece408
* Running python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000263
Op Time: 0.000903
Correctness: 0.76 Model: ece408
```

Figure 1: Correctness and timing with 3 different data size.

2.2 Demonstrate nvprof profiling the execution

We add **nvprof** command to the yml file and run the program.

```
- nvprof -o timeline.nvprof python m3.1.py 100
- nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python
    m3.1.py 100
```

The command will generate timeline.nvprof and *analysis.nvprof. –analysis-metrics significantly slows the run time, so we use 100 datasets during profiling.

```
* Running nvprof -o timeline.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==264== NVPROF is profiling process 264, command: python m3.1.py 100
Loading model... done
New Inference
Op Time: 0.000289
Op Time: 0.000922
Correctness: 0.76 Model: ece408
==264== Generated result file: /build/timeline.nvprof
* Running nvprof --kernels "::forward:1" --analysis-metrics -o forward1_analysis.nvprof python m3.1.py 100
Loading fashion-mnist data... done
==359== NVPROF is profiling process 359, command: python m3.1.py 100
Loading model... done
New Inference
==359== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
Replaying kernel "mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int, int)" (done)
Op Time: 1.1976741 events
Op Time: 0.001029
Correctness: 0.76 Model: ece408
==359== Generated result file: /build/forward1_analysis.nvprof
```

Figure 2: Result of nvprof profiling for the program

2.3 Analysis using NVIDIA Visual Profiler

We import time.nvprof and forward1_analysis.nvprof file into nvvp to analyze our program. Fig 3 demonstrates the control divergence within the kernel. The line 30 in our program has control divergence. Line 30 is :

```
    if ((w < (W_out)) && (h < (H_out))) {
```

This if statement here is to check whether the current element is out of bound or not. Hence, it will result in the occurance of control divergence.

/mxnet/src/operator/custom/./new-forward.cuh

Line 30	Divergence = 16.5% [39600 divergent executions out of 240000 total executions]
Line 32	Divergence = 0% [0 divergent executions out of 198000 total executions]
Line 32	Divergence = 0% [0 divergent executions out of 198000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 198000 total executions]
Line 33	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 34	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 35	Divergence = 0% [0 divergent executions out of 990000 total executions]
Line 45	Divergence = 0% [0 divergent executions out of 198000 total executions]

Figure 3: Divergent branch within the kernel

2.3.1 Kernel Performance Is Bound By Compute

For device "TITAN V" the kernel's memory utilization is significantly lower than its compute utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by computation on the SMs.

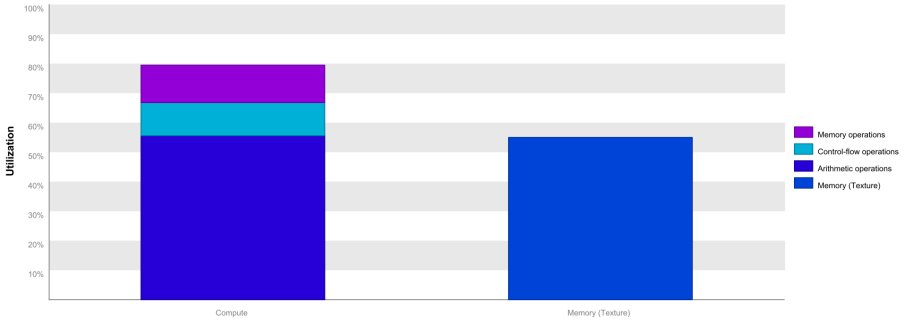


Figure 4: Utilization levels

2.3.2 Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit. Load/Store - Load and store instructions for shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Half - Half-precision floating-point arithmetic instructions.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.
Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.

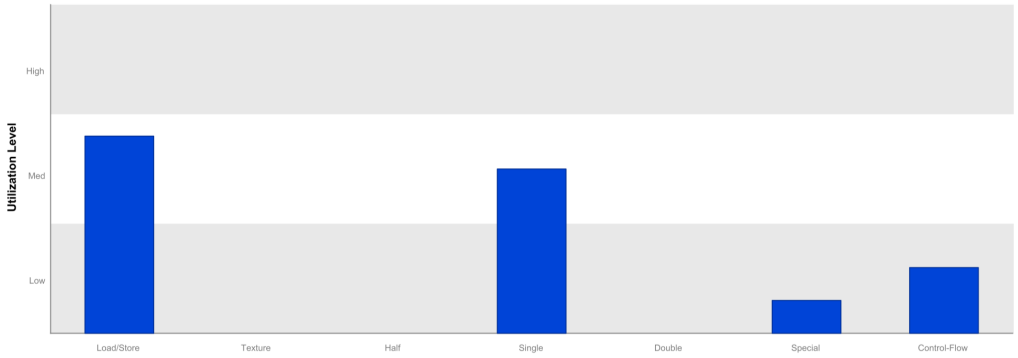


Figure 5: Utilization levels for different types of instructions

2.3.3 Instruction Execution Counts

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were

devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.

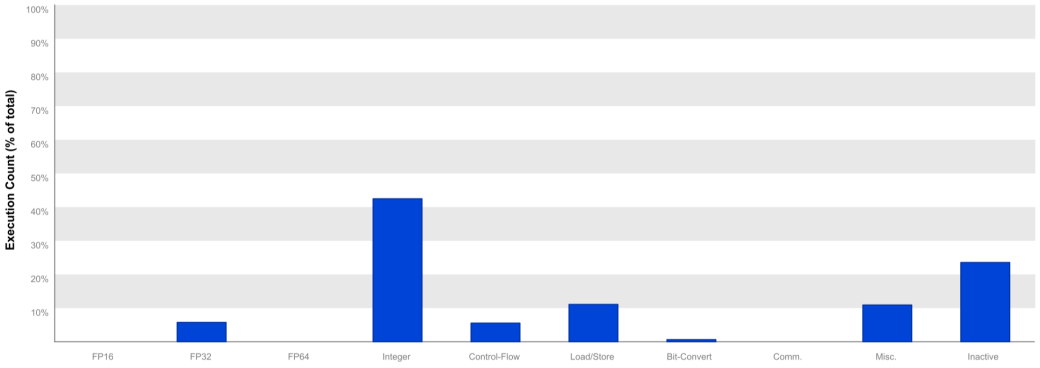


Figure 6: Execution count of different instructions

2.3.4 Floating-Point Operation Counts

The following chart shows the mix of floating-point operations executed by the kernel. The operations are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing operations in that class. The results do not sum to 100

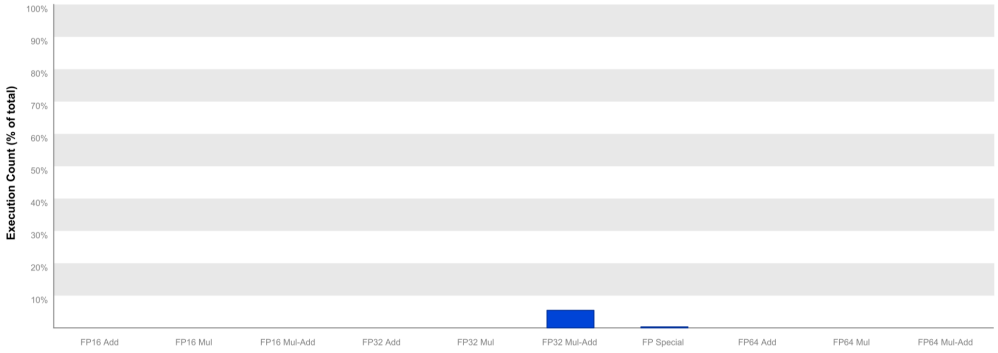
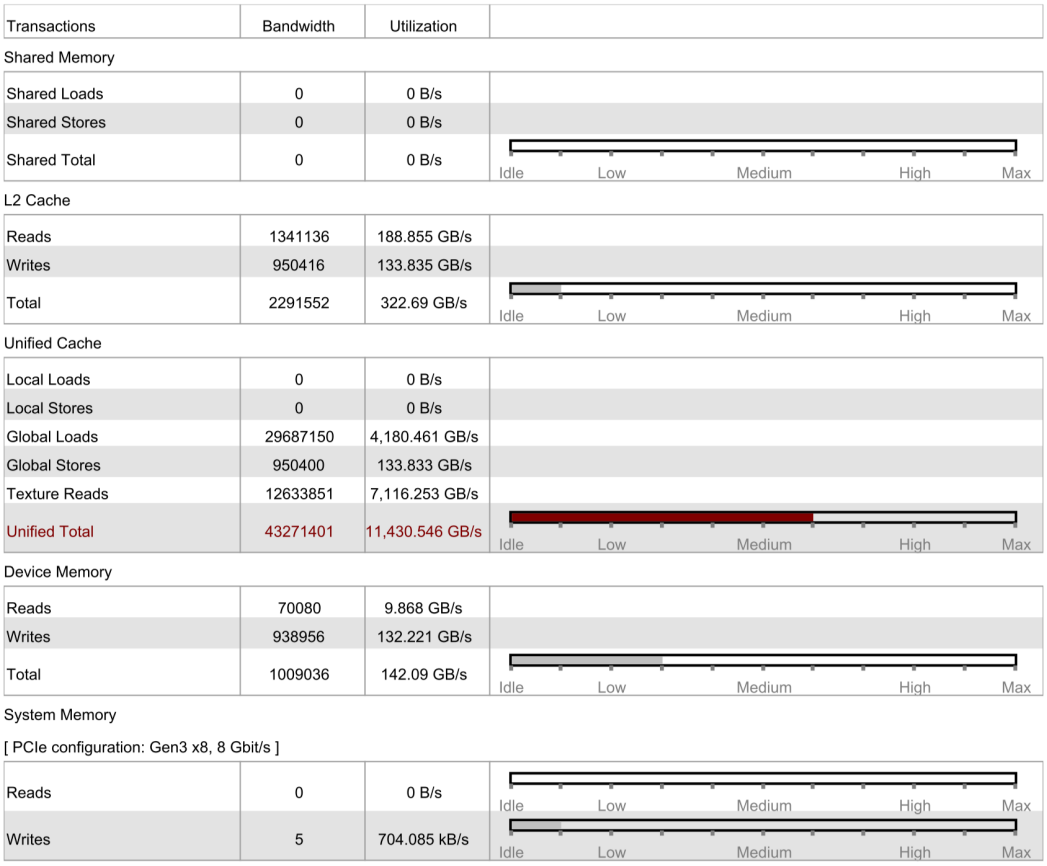


Figure 7: Floating-Point Operation Counts

Fig 8 shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory.



Transactions	Bandwidth	Utilization	
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	Idle Low Medium High Max
L2 Cache			
Reads	1341136	188.855 GB/s	
Writes	950416	133.835 GB/s	
Total	2291552	322.69 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	29687150	4,180.461 GB/s	
Global Stores	950400	133.833 GB/s	
Texture Reads	12633851	7,116.253 GB/s	
Unified Total	43271401	11,430.546 GB/s	Idle Low Medium High Max
Device Memory			
Reads	70080	9.868 GB/s	
Writes	938956	132.221 GB/s	
Total	1009036	142.09 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	704.085 kB/s	Idle Low Medium High Max

Figure 8: Memory bandwidth used by this kernel

Fig 9 demonstrates the occupancy per SM, the register usage, warps usage and shared memory usage.

Variable	Achieved	Theoretical	Device Limit	Grid Size: [100,12,25] (30000 blocks) Block Size: [16,16,1] (256 threads)
Occupancy Per SM				
Active Blocks		8	32	0 3 6 9 12 15 18 21 24 27 30 32
Active Warps	53.65	64	64	0 6 12 18 24 30 36 42 48 54 60 64
Active Threads		2048	2048	0 256 512 768 1024 1280 1536 1792 2048
Occupancy	83.8%	100%	100%	0% 25% 50% 75% 100%
Warps				
Threads/Block		256	1024	0 128 256 384 512 640 768 896 1024
Warps/Block		8	32	0 3 6 9 12 15 18 21 24 27 30 32
Block Limit		8	32	0 3 6 9 12 15 18 21 24 27 30 32
Registers				
Registers/Thread		32	65536	0 8192 16384 24576 32768 40960 49152 57344 65536
Registers/Block		8192	65536	0 16k 32k 48k 64k
Block Limit		8	32	0 3 6 9 12 15 18 21 24 27 30 32
Shared Memory				
Shared Memory/Block		0	98304	0 16k 32k 48k 64k 80k 96k
Block Limit		0	32	0 3 6 9 12 15 18 21 24 27 30 32

Figure 9: Occupancy and some usages.

2.3.5 Occupancy Charts

The following charts show how varying different components of the kernel will impact theoretical occupancy.

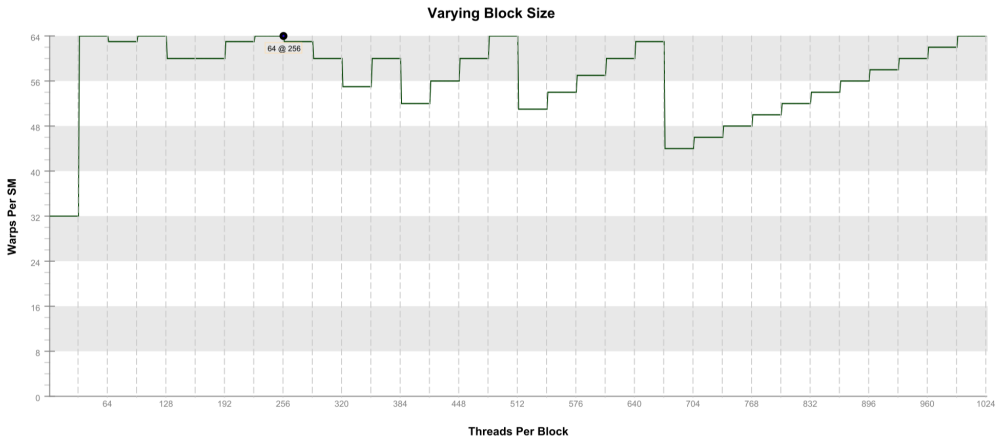


Figure 10: Varying block count.

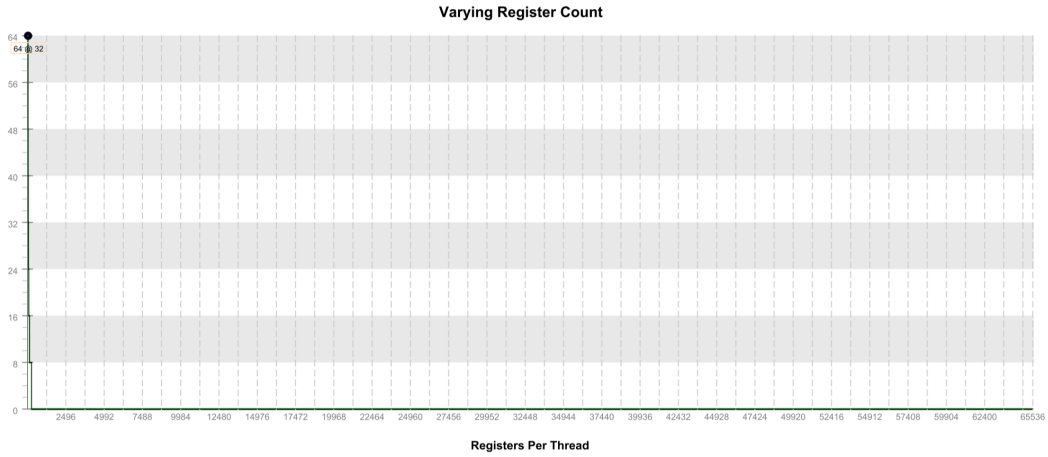


Figure 11: Varying register count.

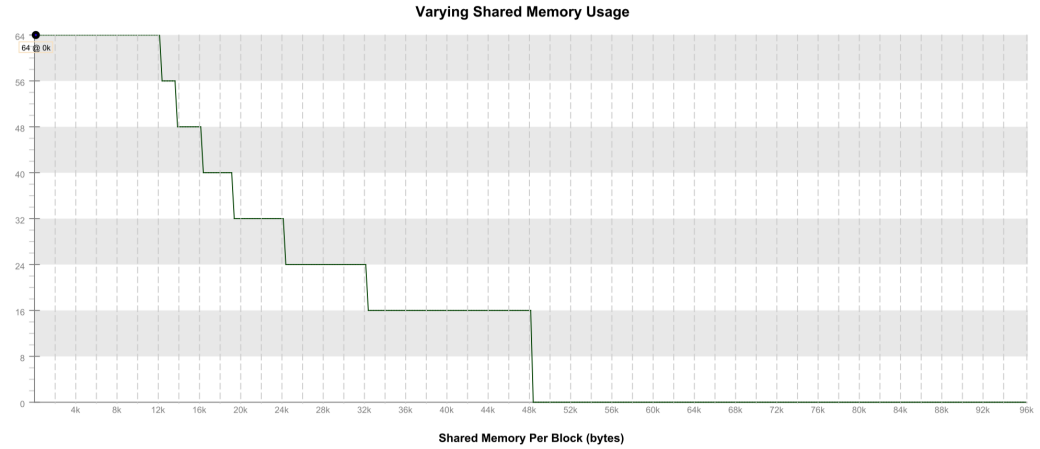


Figure 12: Varying shared memory count.

2.3.6 Multiprocessor Utilization

The kernel's blocks are distributed across the GPU's multiprocessors for execution. Depending on the number of blocks and the execution duration of each block some multiprocessors may be more highly utilized than others during execution of the kernel. The following chart shows the utilization of each multiprocessor during execution of the kernel.

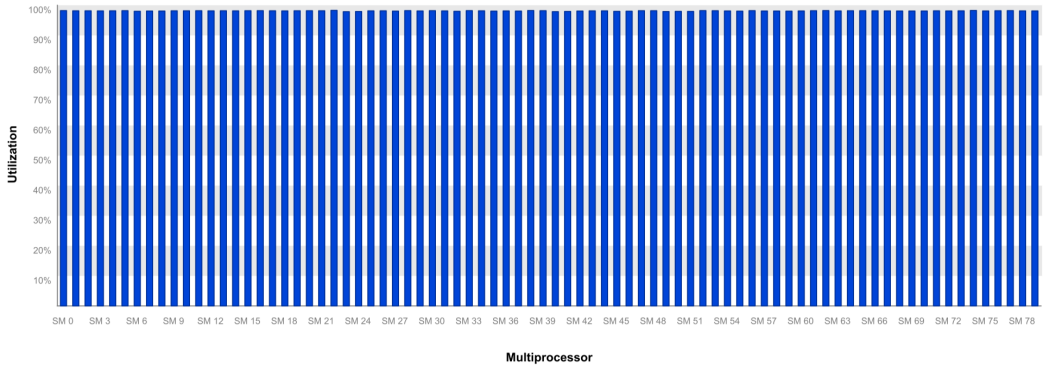


Figure 13: Multiprocessor Utilization.

ECE408 Project
Milestone2

Yutong Xie(yutongx6); Yunyi Zhang(yunyiz3); Pu Jian(pujian2)
Team names: hhh
UIUC Campus students
Instructor: Professor Wen-mei Hwu

October 19, 2019

1 Introduction

This is milestone2 report for ECE408 group project. In this project, we will get practical experience by using, profiling, and modifying MXNet, a standard open-source neural-network framework. Also, we need to emonstrate command of CUDA and optimization approaches by designing and implementing an optimized neural-network convolution layer forward pass.

2 Deliverables

2.1 Report: Show output of rai running MXNet on the CPU and list run time

The file m1.1py is used to run MXNet on the CPU. The output is shown below.
The user time is 18.02s, system time is 4.67s, and elapsed time is 9.56s.

```
* Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
18.02user 4.67system 0:09.56elapsed 237%CPU (0avgtext+0avgdata 6044980maxresident)k
0inputs+2824outputs (0major+1603601minor)pagefaul
ts 0swaps
```

Figure 1: Output of MXNet on the CPU.

2.2 Report: Show output of rai running MXNet on the GPU and list run time

The file m1.2py is used to run MXNet on the GPU. The output is shown below.
The user time is 5.02s, system time is 3.27s, and elapsed time is 4.63s.

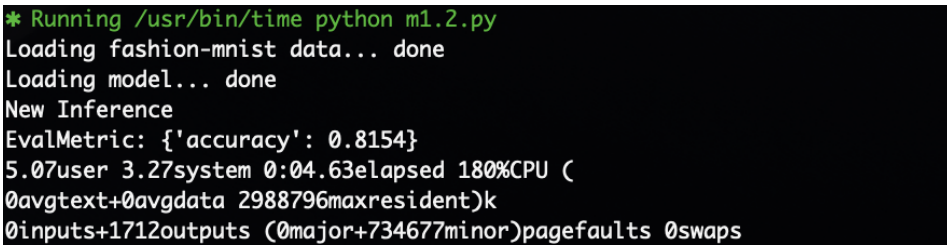


Figure 2: Output of MXNet on the GPU.

2.3 Comparision between kernels and API calls

We modify the **rai_build.yml** file by adding *nvprof python m1.2.py*. Then, we submit the job to rai. The result is following.
From the ouput, we can get a list of all kernels that collectively consume more than 90% of the program time:

[CUDA memcpy HtoD],
volta_scudnn_128x64_relu_interior_nn_v1,
volta_gcgemm_64x32_nt,
void fft2d_c2r_32x32<float, bool=0, bool=0, unsigned int=0, bool=0, bool=0 > (float*, float2 const *, int, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int),
volta_sgemm_128x128_tn,
void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray),
void fft2d_r2c_32x32<float, bool=0, unsigned int=0, bool=0>(float2*, float const *, int, int, int, int, int, int, int, int, int, cudnn::reduced_divisor, bool, int2, int, int)

Also, we can get a a list of all CUDA API calls that collectively consume more than 90% of the program time:

cudaStreamCreateWithFlags,
cudaMemGetInfo,
cudaFree

Figure 3: Output of using nvprof.

2.4 Report: Include an explanation of the difference between kernels and API calls

The API calls part is the section contains the time of CPU using, while the profiling result is the time that the GPUs taking. The total time of the API call is from the moment it is launched to the moment it completes, so will overlap with executing kernels.

2.5 Create a CPU implementation

We modify **new-forward.h** file to implement the forward convolution on CPU. When we use command */usr/bin/time python m2.1.py*, we can get the whole execution time. The output is following.
From the output, we can see that he user time is 83.56s, system time is 8.14s, and elapsed time is 1:13.83min.

```
* Running /usr/bin/time python m2.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 10.849182
Op Time: 59.401043
Correctness: 0.7653 Model: ece408
83.56user 8.14system 1:13.83elapsed 124%CPU (0avgtex
t+0avgdata 6044128maxresident)k
0inputs+0outputs (0major+2310702minor)pagefaults 0swaps
```

Figure 4: Ouput of whole program.

Then, we use command *python m2.1.py [data size]* to check the correctness and get the op time. Here, we tried data size with 100, 1000, and 10000. Because we have to convolution layer, there are two different op time.

- **Data size:** 100 **Op time:** 0.112125s 0.590948s **Correctness:** 0.76

- **Data size:** 1000 **Op time:** 1.084841s 6.087901s **Correctness:** 0.767
- **Data size:** 10000 **Op time:** 10.820261s 60.201280s **Correctness:** 0.7653

The output is shown below.

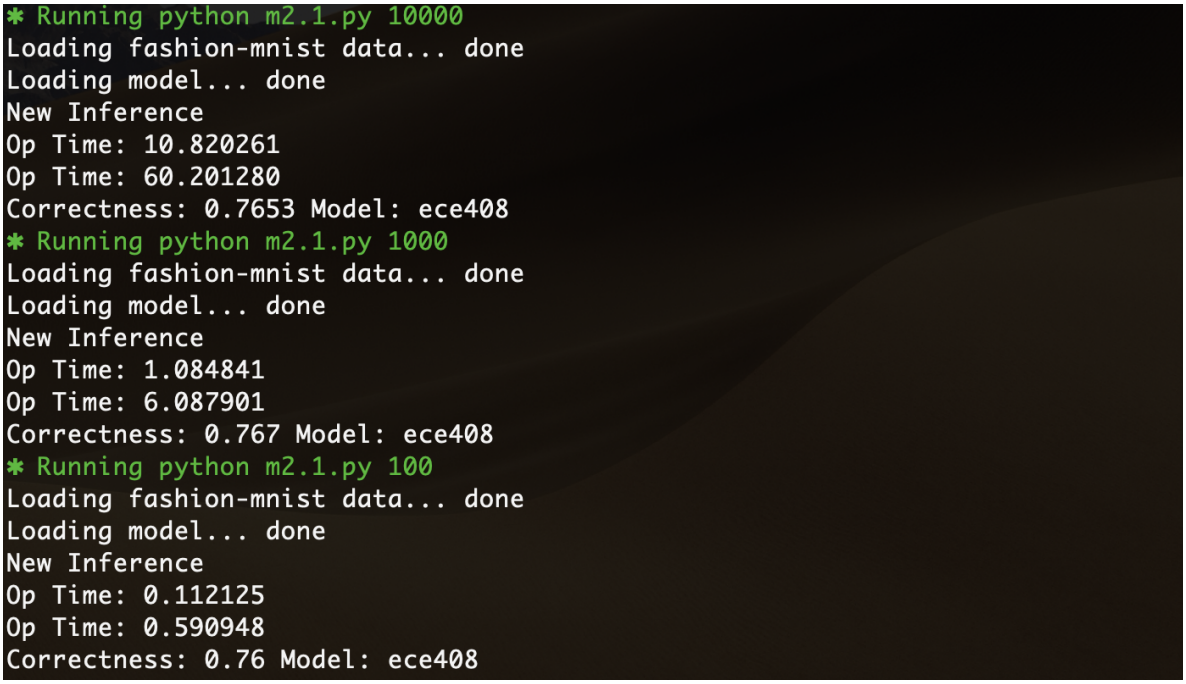


Figure 5: Ouput for different data size.