Name: *<Jiahan Chang>*
NetID: *<jiahanc2>*
Section: *<AL1 >*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0. 176354 ms	0.634392 ms	0m1.201s	0.86
1000	1.62789 ms	6.26231 ms	0m10.174s	0.886
10000	16.017 ms	63.1262 ms	1m37.002s	0.8714

1. **Optimization 1: *<Tiled shared memory convolution>***

    a. Which optimization did you choose to implement and why did you choose that optimization technique.

    I choose the tiled shared memory convolution because, with the tiled shared memory, it can save time when accessing data. The device does not need to access from global memory each time so it can save time.

    b. How does the optimization work? Did you think the optimization would increase the performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

    When the data is first accessed from global memory, it will be stored in shared memory, and when it is accessed. The device can only access shared memory so it can save time in accessing global memory. I did not synergize this optimization with my previous one.

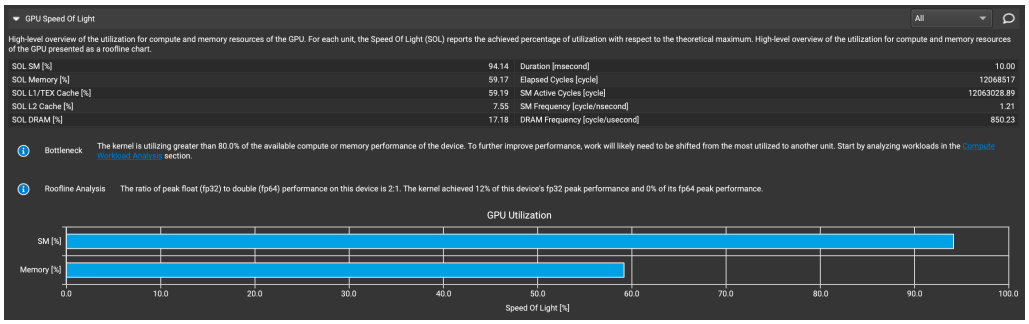
    c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.207617 ms*	*1.01881 ms*	*0m1.191s*	*0.86*
1000	*1.95519ms*	*10.035ms*	*0m10.423s*	*0.886*
10000	*19.4687 ms*	*100.676 ms*	*1m35.341s*	*0.8714*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

After the optimization, it is wired that the time is slower than the normal one in milestone 2 in both layers. So I use the nsys and Nsight-Compute to analyze the reason.

```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  -----------  --------------  --------------  --------------  ----------------------
   49.1      1326678106           8     165834763.3          216582      1294749495  cudaMalloc
   42.7      1155147859           8     144393482.4           39478       607191209  cudaMemcpy
```



From the profile, the utilization of the shared memory has increased significantly. Also, both the memory and SM have a much larger utilization shows that using shared memory improves memory utilization. As for the speed decrease, I think it is due to the control divergence. There are so many places to have the control divergence such as judging the input is in size so this divergence may make the speed slow down because the device needs time to judge.

e. What references did you use when implementing this technique?

Chapter 16 page 15 of the book.

2. **Optimization 2: *< Shared memory matrix multiplication and input matrix unrolling >***

a. Which optimization did you choose to implement and why did you choose that optimization technique.

Shared memory matrix multiplication and input matrix unrolling. As we have learned, making the X to X_unroll will make it much easier for GPU to access.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

I first make the input X to X_unroll and then do the multiplication with shared memory optimization. I think this would help because GPU can access much easier than normal X. Then, shared memory can same time from accessing from global memory. I synergized this with optimization one.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*2.52296 ms*	*3.08717 ms*	*0m1.211s*	*0.86*
1000	*23.5422 ms*	*24.9347 ms*	*0m9.741s*	*0.886*
10000	*251.434 ms*	*295.296 ms*	*1m38.805s*	*0.8714*

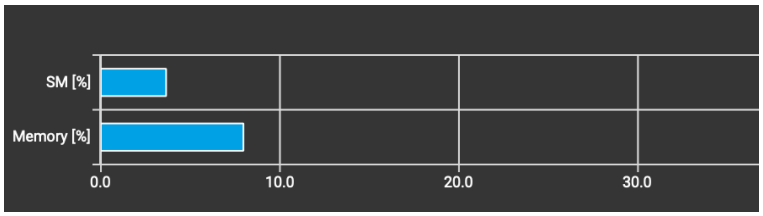
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

As the result shows, this optimization is not successful because it is much slower than the basic one.
For this optimization, I use the dataset size 1000 instead of 10000 to save time for the nsys and Nsight output.

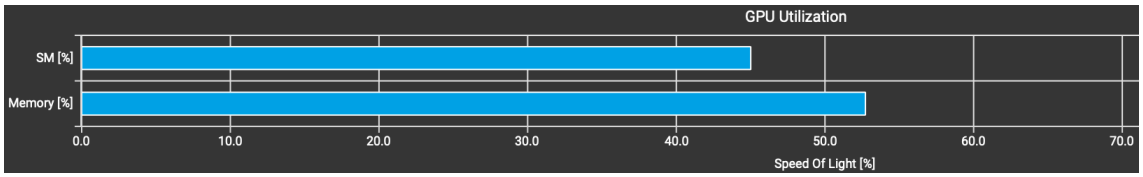
Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
99.2	23095654154	10	2309565415.4	4219	23094001717	cudaMalloc
0.5	106319084	8	13289885.5	17866	56637711	cudaMemcpy

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
54.0	19665071	2000	9832.5	8160	12384	unroll
46.0	16763777	2000	8381.9	7392	10176	forward_kernel_unroll

Unroll kernel



Forward kernel



In the unroll kernel, there are a lot of divide and mod operations because we want to remap the matrix X to X_unroll, which can cost a lot of time for GPU to handle. The second foeward kernel also needs much time to load and sotre. That is why this optimization can not save time.l

e. What references did you use when implementing this technique?

Chapter 16 page 20.

3. **Optimization 3: < *Weight matrix (kernel values) in constant memory* >**

   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      I choose the constant memory. This is because, in this project, the weighted matrix is never changed. So I think changing it to a constant memory may help speed up.

   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

      Because the weighted matrix is never changed, moving it to the constant will cause no error. Moreover, if it is moved to constant, it can speed up the calculation and increase the utilization of bandwidth. I did not synergize it with previous optimization.

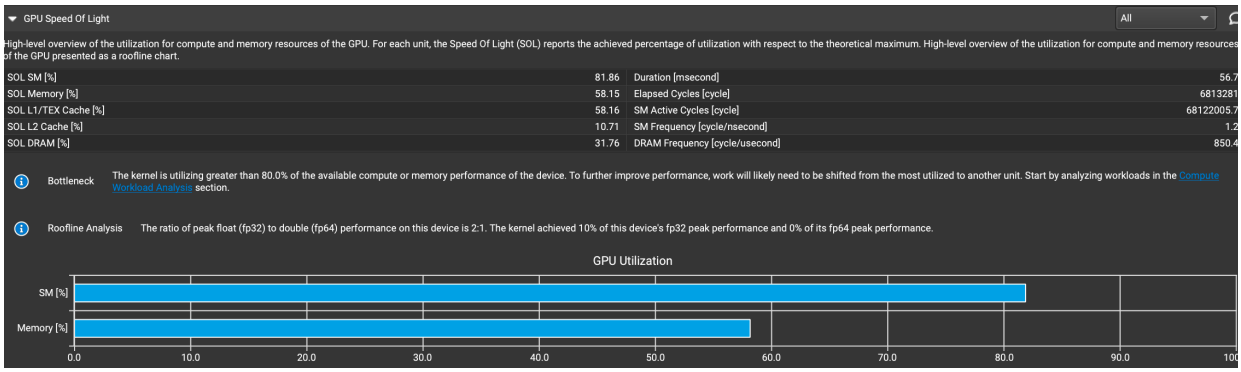
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.16784 ms*	*0.585554 ms*	*0m1.150s*	*0.86*
1000	*1.48527 ms*	*5.65395 ms*	*0m10.870s*	*0.886*
10000	*14.6028 ms*	*56.7984 ms*	*1m41.857s*	*0.8714*

   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

This optimization is successful because it reduce the OP times for both layers in all the three datasets which means that this optimization increases the efficiency of the algorithm in memory loading.

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
79.7	1084429131	8	135553641.4	19895	588935006	cudaMemcpy
13.6	185356921	8	23169615.1	68603	182010598	cudaMalloc



The memory utilization is reduced because the weight matrix is constant, so it can save a lot of cost of memory access.

e. What references did you use when implementing this technique?

Lecture 8 and Chapter 7.

4. **Optimization 4: *< Fixed point (FP16) arithmetic. >***

a. Which optimization did you choose to implement and why did you choose that optimization technique.

Fixed point (FP16) arithmetic. Because in the readMe file it is the only one that may change the accuracy so I want to learn what is it.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

When doing floating point operations, GPUs can see 2X-8X speedup on FP16 over FP32 and it uses half the space as a normal float (source linked in references). I synergize this FP16 with constant memory of the weight kernels and Shared memory matrix multiplication and input matrix unrolling.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

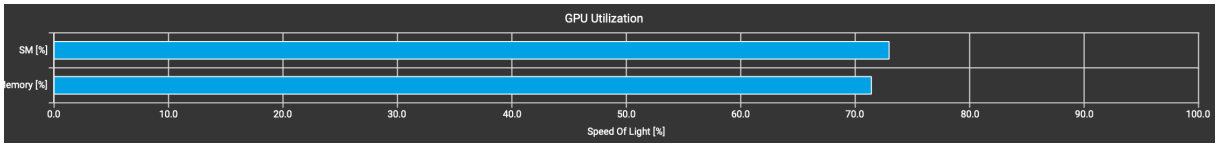
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.741727 ms*	*1.28054 ms*	*0m1.158s*	*0.86*
1000	*7.09713 ms*	*12.9886 ms*	*0m9.703s*	*0.887*
10000	*70.3486 ms*	*130.289 ms*	*1m44.196s*	*0.8716*

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of)

The implementation is successful because combined with the optimization 2, it directly shorten the time almost the same as the basic one. Compared with the only Shared memory matrix multiplication and input matrix unrolling, this performance is much better, showing the success of FP16. Moreover, it can increase the accuracy which is the only optimization among my practice.

```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ----------------
-----------------------------------
   59.6       209354336           8      26169292.0           72456       208313831  cudaMalloc
   34.3       120717964           8      15089745.5           17848        64714184  cudaMemcpy
```

```
Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
-----------------------------------
  100.0        20110475           2      10055237.5         7062871        13047604  matrixMultiplyShared
```



From the profile, the effect is not large. So maybe the floating points operations are not the bottleneck and there should be somethings else that limit the factor.

e. What references did you use when implementing this technique?

https://ion-thruster.medium.com/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f