Name: *<Yicong Mo>*
NetID: *<yicongm2>*
Section: *<class section>*

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	0.180575 ms	0.65874 ms	0m1.284s	0.86
1000	1.63858 ms	6.29151 ms	0m11.591s	0.886
10000	14.7092 ms	62.9892 ms	1m43.325s	0.8714

1. *Optimization 1:* **Tiled shared memory convolution (2 points**)
student_code/others/tile shared memory.cu
   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      I used tiled shared memory convolution. In computing, a large amount of data will be reused. By storing data in shared memor, data can be accessed with lower latency.

   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?

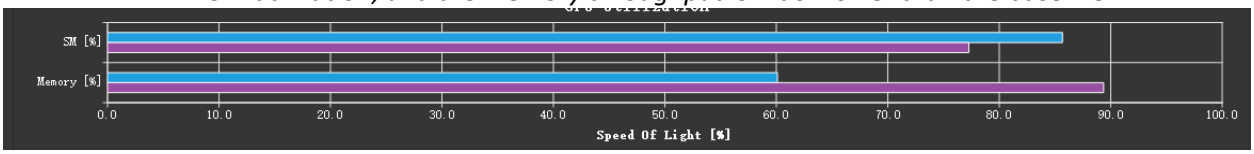
      Put input data into shared memory. Yes, I think the optimization would increase performance of the forward convolution. Because the shared memory means lower access latency. And the optimization doesn't synergize with any other previous optimization.

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	0.217275 ms	0.84988 ms	0m1.164s	0.86
1000	1.99648 ms	8.2324 ms	0m10.087s	0.886
10000	19.7838 ms	82.0799 ms	1m38.869s	0.8714

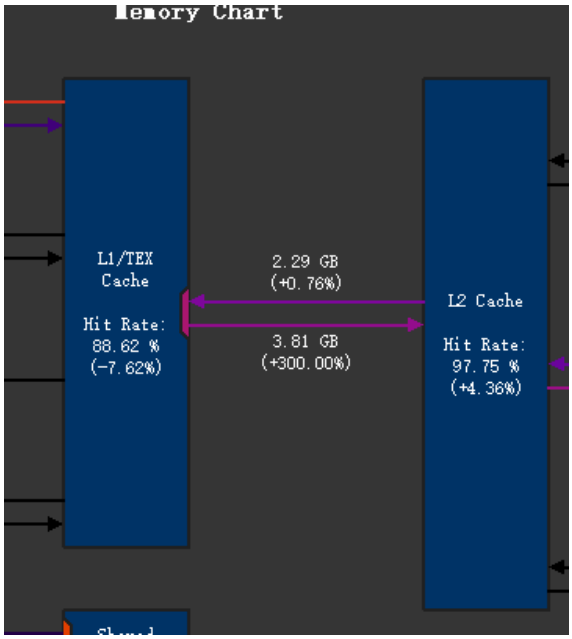
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

It didn't improve the performance compared to my baseline.
Purple is the baseline and blue is the optimization. We can see the blue one has higher SM utilization, and the memory throughput is much lower than the baseline.



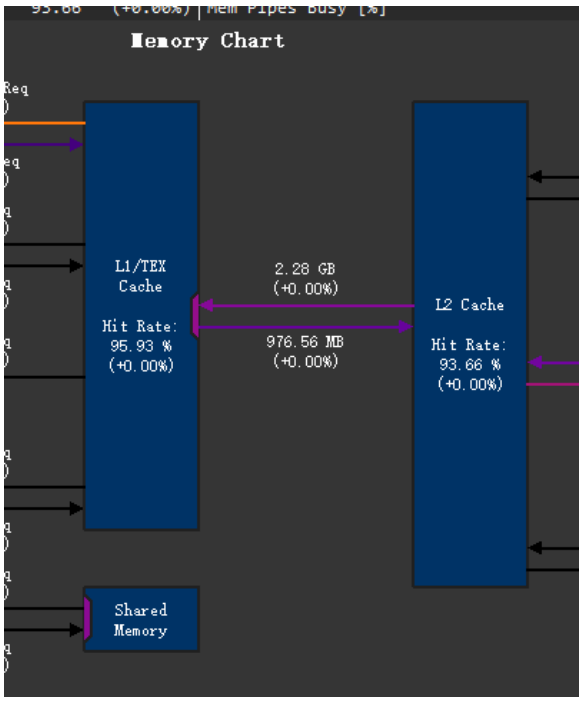
However, based on the nsys profile below, it shows the performance is not better than baseline, but even worse. I think it is because the time saved by using shared memory is not enough to make up for the time spent on loading input data into shared memory.

Optimization 1:



Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	101720305	2	50860152.5	19780762	81939543	conv_forward_kernel

Baseline:



Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	79533806	2	39766903.0	16375179	63158627	conv_forward_kernel

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

Lab4

2. **Optimization 2: Weight matrix (kernel values) in constant memory (1 point)**
student_code/others/tile shared memory+constant.cu

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

I chose weight matrix in constant memory optimization. Because the weight matrix is constant, putting it in constant memory can help other parts get higher bandwidth.

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?

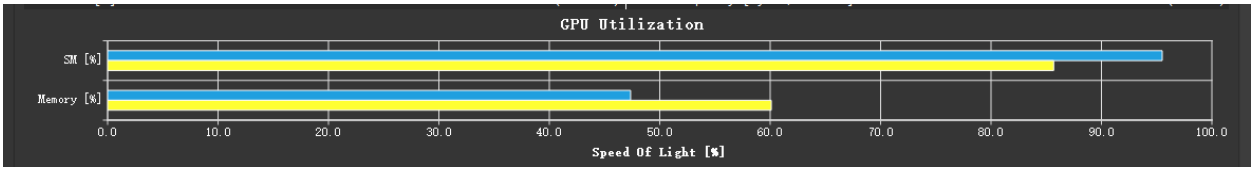
I put the weight matrix into constant memory. Compared with not using constant memory, the performance has been improved. Because weight matrix is same during the running time, put it into constant memory can save some time on transferring data. This optimization synergizes with the first optimization (Tiled shared memory convolution).

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).

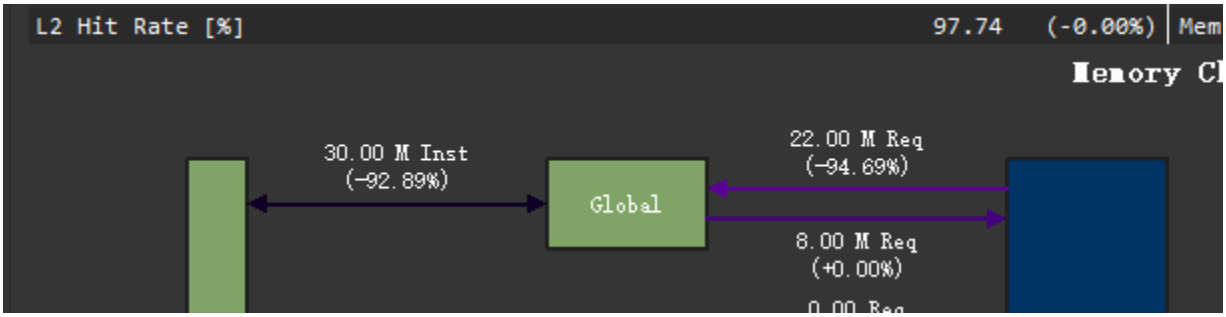
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.190013 ms*	*0.728978 ms*	*0m1.408s*	*0.86*
1000	*1.68759 ms*	*7.03213 ms*	*0m11.408s*	*0.886*
10000	*14.9969 ms*	*63.6819 ms*	*1m44.633s*	*0.8714*

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).

Yes, it significantly improves performance.
In the following image chart(yellow is the the kernel without constant memory, and blue is the kernel with constant), we can see the blue one has higher SM utilization with lower memory pipeline throughput.

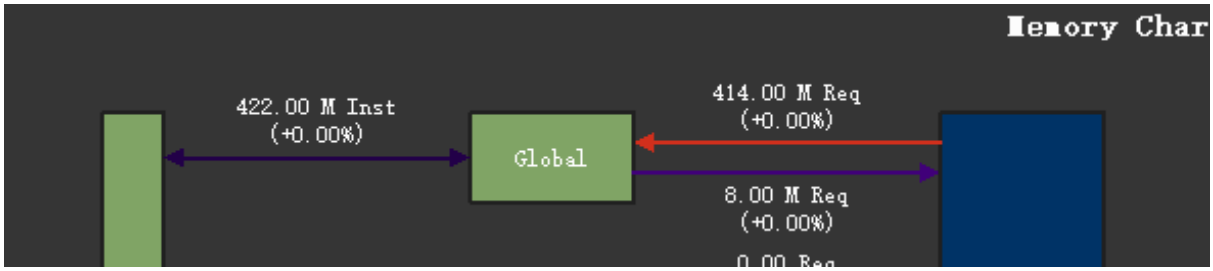


With constant memory:



Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	86966679	2	43483339.5	16683141	70283538	conv_forward_kernel

without constant memory:



Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	101720305	2	50860152.5	19780762	81939543	conv_forward_kernel

The kernel using constant memory just has 22 M Req, but the optimization 1 has 414 M.
Also the total time of kernel using constant memory is greatly reduced relative to the another one.

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

Lab4

3. **Optimization 3: Shared memory matrix multiplication and input matrix unrolling + Kernel fusion for unrolling and matrix-multiplication (requires previous optimization) (5 points)** **student_code/others/fusion.cu**

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

I used Kernel fusion for unrolling and matrix-multiplication based on shared memory matrix multiplication and input matrix unrolling.
Compare with the shared memory matrix multiplication and input matrix unrolling optimization, the data don't need to be transfer from unroll kernel to matrix multiplication kernel. I think it will greatly reduce the OP time.

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 put the input unroll function into the shared memory matrix multiplication kernel. In this case, the unrolled input data can be loaded directly into the shared memory instead of being transferred. I think the optimization would increase performance, because of the elimination of the transfer. And this optimization synergizes with the 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	*0.621291 ms*	*0.352736 ms*	*0m1.179s*	*0.86*
1000	*6.07391 ms*	*3.35128 ms*	*0m10.042s*	*0.886*
10000	*60.5924 ms*	*33.3365 ms*	*1m40.072s*	*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 improve performance a lot, for 10000 Batch size, the total Op Time reduced from about 115ms to 90ms. The speed is increased by about 20% based on the nsys profile's total time.

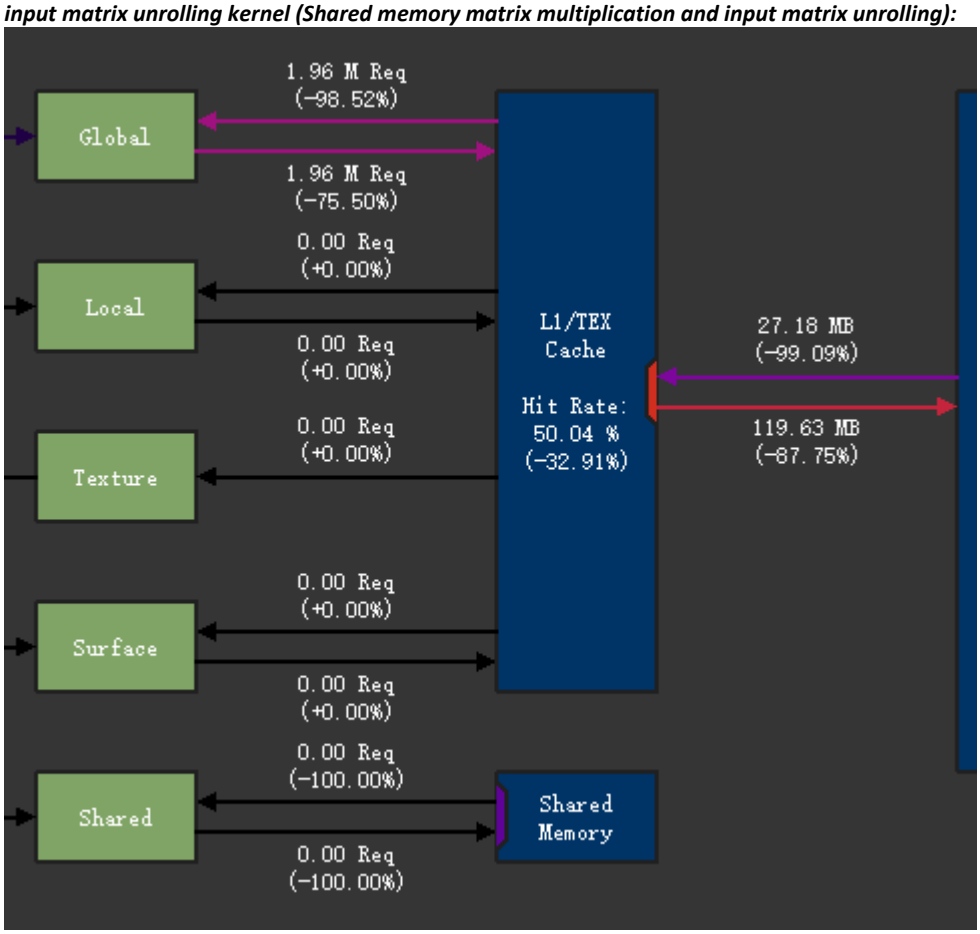
Shared memory matrix multiplication and input matrix unrolling:

```
Time(%)     Total Time   Instances       Average       Minimum       Maximum  Name
-------  --------------  ----------  --------------  ------------  ------------  ----------------------------------------------------------------------------------------------------
   62.6        72864843         200        364324.2        271359        456638  conv_forward_kernel
   37.4        43446246         200        217231.2        198111        240063  input_unroll1
```

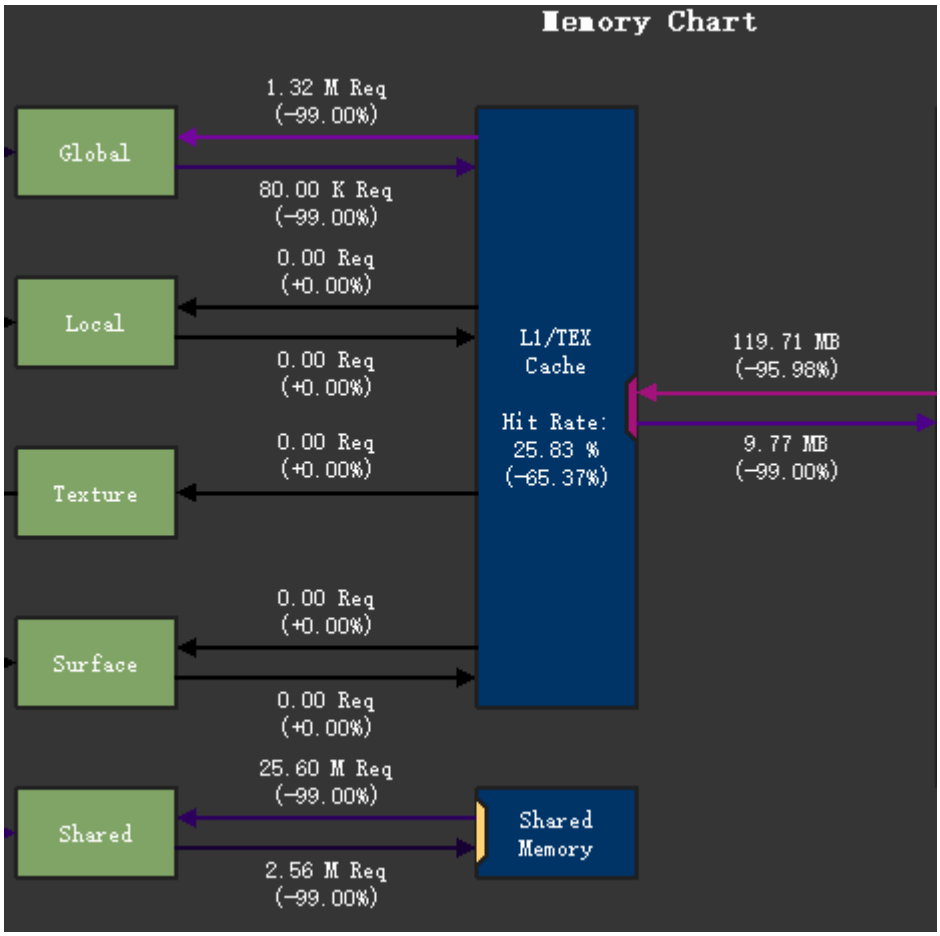
Kernel fusion for unrolling and matrix-multiplication:

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	94542470	2	47271235.0	33347392	61195078	conv_forward_kernel

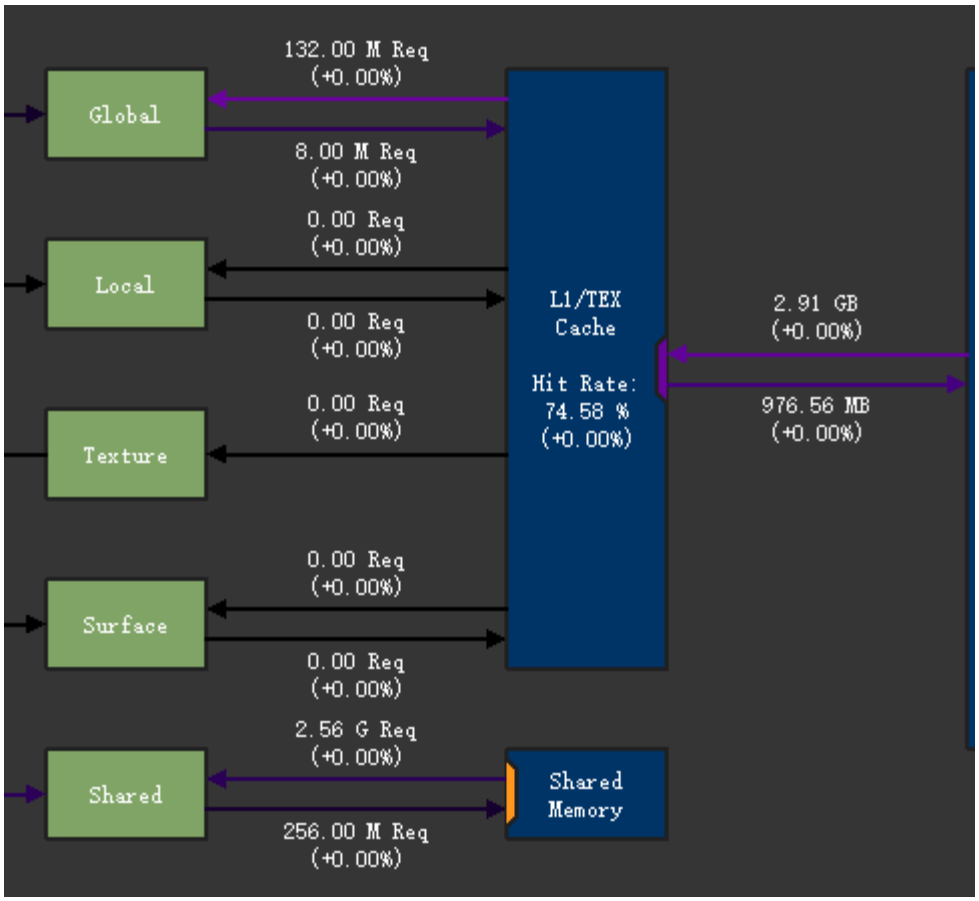
input matrix unrolling kernel (Shared memory matrix multiplication and input matrix unrolling):



Shared memory matrix multiplication kernel (Shared memory matrix multiplication and input matrix unrolling):



Fusion kernel (Kernel fusion for unrolling and matrix-multiplication):



*For **shared memory matrix multiplication and input matrix unrolling** under 10000 Batch size, Since I feed the kernel 100 Batch each loop, and there is 2 layers, 100 instances of conv_forward_kernel and 100 instances of input_unroll1 for each layer. Based on the memory workload analysis shown above, the total Req between global and L1 in Shared memory matrix multiplication kernel are 1.32M * 100 and 80K * 100, which are equal to the Req between global and L1 in Fusion kernel. It is also true for Req between Shared and Shared Memory. However, the fusion kernel of **Kernel fusion for unrolling and matrix-multiplication** doesn't produce extra Req generated by unrolling kernel of **the shared memory matrix multiplication and input matrix unrolling**. That is why the performance can be improved.*

e. What references did you use when implementing this technique?
Reading: Kirk & Hwu Chapter 16
Lab3

4. **Optimization 4: Fixed point (FP16) arithmetic. (note this can modify model accuracy slightly) (4 points) student_code/others/fp16.cu**

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

      I used FP16 arithmetic. FP16 has lower precision, the cost of computing and data transmission will be lower. I think this will improve performance.

   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?

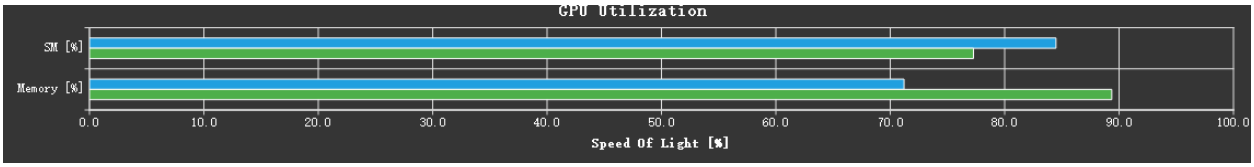
      Convert float to half2 before doing multiplication and addition, and convert half2 back to float before output. I think the optimization would improve performance. Because the FP16 is half-precision, which means the cost on computing and transfer would be lower compared to the float. And this optimization synergizes with the baseline.

   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	0.2208 ms	0.845338 ms	0m1.132s	0.86
1000	2.04824 ms	8.0746 ms	0m10.135s	0.887
10000	20.3332 ms	78.4819 ms	1m39.386s	0.8716

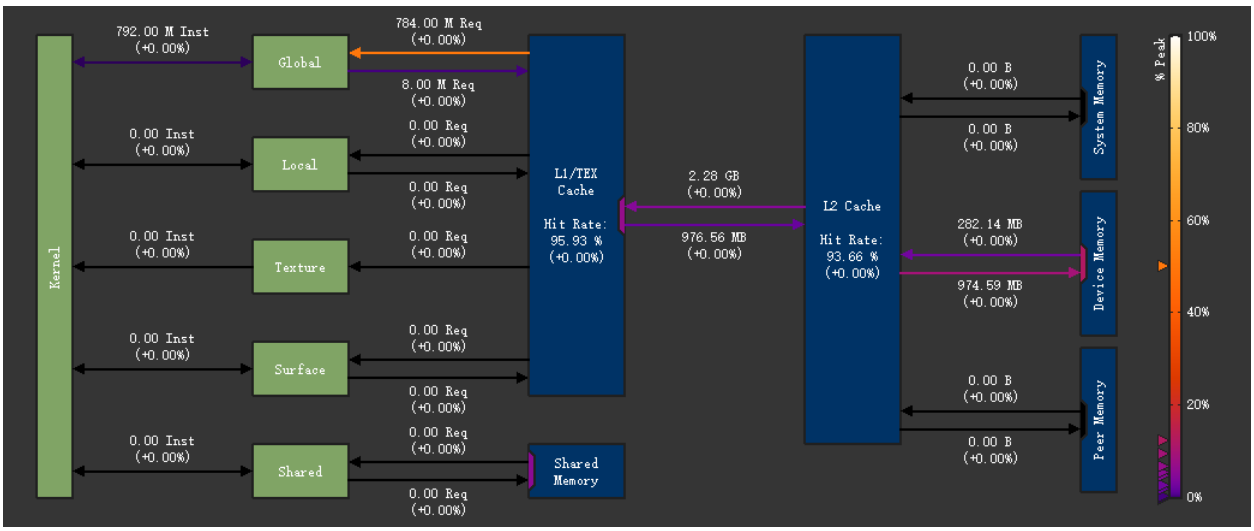
   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

This optimization didn't improve performance.
Based on the following image, we can see the blue one (FP16) has higher SM throughput and lower memory throughput. (Baseline is represented by the green)



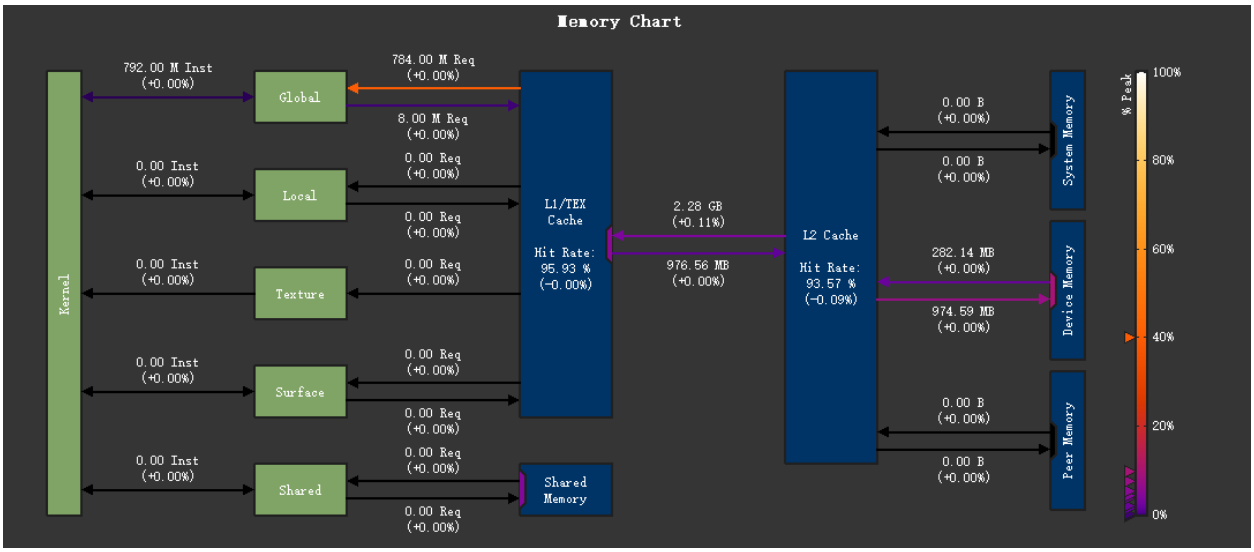
Based on the nsys profile, we can see the performance of FP16 kernel is worse than baseline. Also, their memory workload analysis is pretty much same. I think the reason for the same workload analysis is that the input and mask passed into the kernel are both float types, I just convert the float type is converted to the half2 type before multiplication and addition, so there is no efficiency of data transmission. In addition, I don't think the speedup of calculation brought by the reduction of precision is enough to make up for the time consumed by type conversion.

Baseline:



Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	79533806	2	39766903.0	16375179	63158627	conv_forward_kernel

FP16:



Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	100535873	2	50267936.5	20252018	80283855	conv_forward_kernel

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

[CUDA Math API :: CUDA Toolkit Documentation (nvidia.com)]

5. **Optimization 5: Multiple kernel implementations for different layer sizes (1 point)**
new_forward.cu Fastest one (under "custom", not under "student_code/others" folder)

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

I used different kernel for different layer size. I noticed that for different sizes of layers, the same kernel has different performance.

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 used 2 kernels, the first one is the kernel of **Tiled shared memory convolution** the second kernel is fusion kernel from Optimization 3. For the layer with a Map_out of 16, I load data into fusion kernel, and for the layer with a Map_out less than 16, I use **Tiled shared memory convolution**'s kernel. I think this optimization would improve performance. During the analysis of the previous optimizations, I found that the performance of the same kernel for different layer sizes is different. Therefore, I compared the performance of several optimizations that have been implemented under the same layer size, and find the kernel with the best performance in each size, let them work together to achieve better performance. This optimization synergizes with the Optimization 1, 2 and 3.*

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	0.182958 ms	0.349832 ms	0m1.146s	0.86
1000	1.67578 ms	3.33303 ms	0m10.264s	0.886
10000	16.6989 ms	33.3439 ms	1m40.068s	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 improve performance a lot. Based on the following 3 nsys profile, we can see the total time of **Multiple kernel implementations for different layer sizes** is much less than the **Tiled shared memory convolution (constant memory)** and **Kernel fusion for unrolling and matrix-multiplication.***

Multiple kernel implementations for different layer sizes:

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
66.6	33537055	1	33537055.0	33537055	33537055	fusion_kernel
33.4	16785282	1	16785282.0	16785282	16785282	conv_forward_kernel_shared

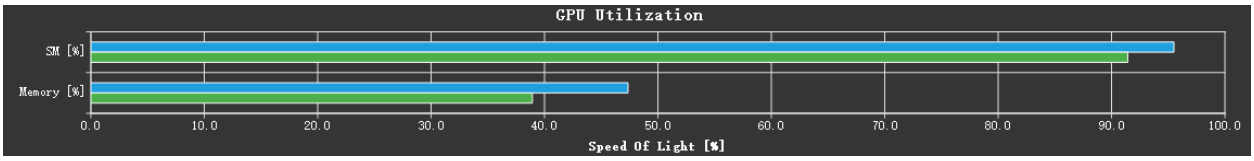
Tiled shared memory convolution (constant memory):

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	86966679	2	43483339.5	16683141	70283538	conv_forward_kernel

Kernel fusion for unrolling and matrix-multiplication:

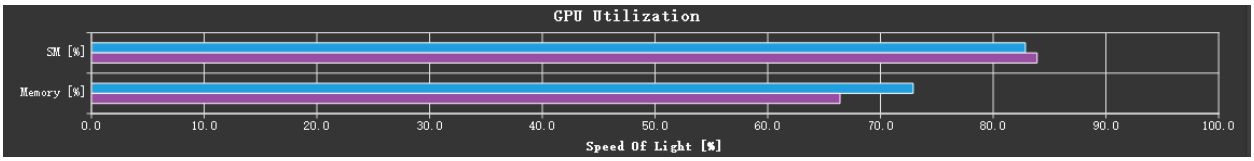
Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	94542470	2	47271235.0	33347392	61195078	conv_forward_kernel

Tiled shared memory convolution + constant memory 2 layers (Blue is layer with Map_out of 4, green is layer with Map_out of 16):



*Based on the above GPU Utilization of **Tiled shared memory convolution + constant memory**, we can see the performance on the first layer is better than the second layer.*

Kernel fusion for unrolling and matrix-multiplication (Blue is layer with Map_out of 4, purple is layer with Map_out of 16):



*Based on the above GPU Utilization of **Kernel fusion for unrolling and matrix-multiplication**, we can see GPU Utilizations of 2 layers are similar. And if we compare the second layer of **fusion kernel** with the second layer of **shared memory + constant memory kernel**, we notice the fusion kernel is much better. If we use **shared memory + constant memory kernel** on first layer and use **fusion kernel** on second layer, the performance improvement would be produced.*

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

Optimization 1, 2 and 3