Name: *Ge Yuhao*
NetID: *Yuhaoge2@illinois.edu*
Section: *ZJ1/ZJ2*

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	*0.174333ms*	*0.630749ms*	*1.216s*	*0.86*
1000	*1.63886ms*	*6.25089ms*	*9.673s*	*0.886*
10000	*16.0759ms*	*62.9824ms*	*1m34.672s*	*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.

   Optimize through using tiled shared memory convolution. Because the original method read image data and weight data from the global memory, the speed can be slow, and the total time may be limited by the memory bandwidth. By first storing the image data and weight data into the shared memory, the later access time will be smaller.

   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?

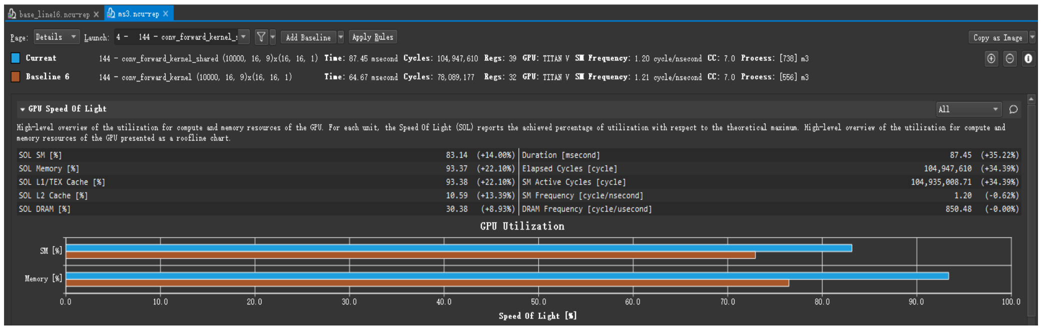
   Divide each image into many tiles, and each block will calculate one tile. First copy the image date from the global memory into the shared memory, so each thread in a block can use the same shared memory to do calculation. Then the result will write back to the global memory. I think this optimization would increase performance of the forward convolution because the shared memory reading is much faster than the global memory. This is the first optimization I try.

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 Layer Time	Total Execution Time	Accuracy
100	*0.2387ms*	*0.6342ms*	*11.3232ms*	*1.324s*	*0.86*
1000	*2.4932ms*	*6.3892ms*	*114.1242ms*	*9.523s*	*0.886*
10000	*24.007ms*	*63.5762ms*	*1120.903ms*	*1m28.123s*	*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).

The performance is not improved, and the speed is even slower. From the Nsight-Compute, the utilization of SM and memory does increase while using shared memory, but the total time needed is increased. I think this is because the input images' size is not that large, and the shared memory is not fully used, or the original op time is not limited by the global memory bandwidth. Also, copying data from global memory to shared memory also need time.


```
base_line16.ncu-rep   ms3.ncu-rep
Page: Details   Launch: 4 - 144 - conv_forward_kernel_s   Add Baseline   Apply Rules   Copy as Image
Current      144 - conv_forward_kernel_shared (10000, 16, 9)x(16, 16, 1) Time: 87.45 msecond Cycles: 104,947,610 Regs: 39 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [738] m3
Baseline 6   144 - conv_forward_kernel (10000, 16, 9)x(16, 16, 1)        Time: 64.67 msecond Cycles: 78,089,177 Regs: 32 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [556] m3

GPU Speed Of Light                                                                                     All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%]             83.14  (+14.00%) | Duration [msecond]                   87.45  (+35.22%)
SOL Memory [%]         93.37  (+22.10%) | Elapsed Cycles [cycle]         104,947,610  (+34.39%)
SOL L1/TEX Cache [%]   93.38  (+22.10%) | SM Active Cycles [cycle]    104,935,008.71  (+34.39%)
SOL L2 Cache [%]       10.59  (+13.39%) | SM Frequency [cycle/nsecond]          1.20   (-0.62%)
SOL DRAM [%]           30.38   (+8.93%) | DRAM Frequency [cycle/usecond]      850.48   (-0.00%)
```

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

I refer to the chapter16 of the textbook for this technique

2. **Optimization 2: <*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 to implement a new version with Weight matrix (kernel values) in constant memory. Because the kernel values are frequently used in the convolution, and if it is in constant memory the total speed may be faster.

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?

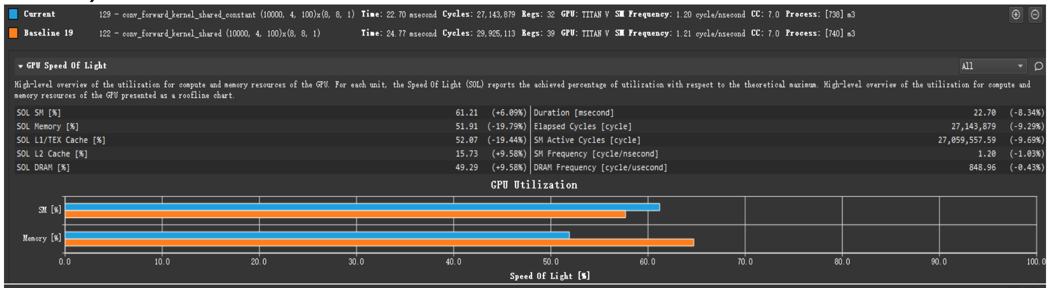
   In the code, I copy the weight matrix data into the constant memory and use this to do calculation in the kernel. I think this would increase performance, as we only need to copy the weight matric data once and can reach it quickly in later use. I think this will synergize with the former shared memory method.

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).

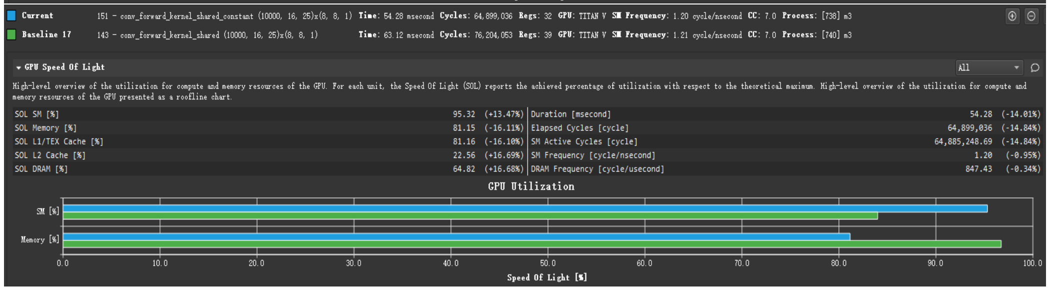
Batch Size	Op Time 1	Op Time 2	Total Layer Time	Total Execution Time	Accuracy
100	*0.2121ms*	*0.5421ms*	*10.782ms*	*1.258s*	*0.86*
1000	*2.2791ms*	*5.4101ms*	*113.217ms*	*9.781s*	*0.886*
10000	*22.6564ms*	*54.0805ms*	*1121.443ms*	*1m25.421s*	*0.8714*

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 successfully improved the performance. Because the optime become smaller than the former shared memory method. From the Nsight-Compute, the utilization of SM is increased and the use of memory is decreased, and this is corresponding to my theoretical analysis. Also, as the copying is conducted in the host, no more kernel time is needed for memory copying, so this is faster than the shared memory method.



```
🟦 Current      129 - conv_forward_kernel_shared_constant (10000, 4, 100)x(8, 8, 1) Time: 22.70 msecond Cycles: 27,143,879 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [738] m3
🟧 Baseline 19  122 - conv_forward_kernel_shared (10000, 4, 100)x(8, 8, 1)          Time: 24.77 msecond Cycles: 29,925,113 Regs: 39 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [740] m3

GPU Speed Of Light
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%]              61.21   (+6.09%) | Duration [msecond]                      22.70   (-8.34%)
SOL Memory [%]          51.91  (-19.79%) | Elapsed Cycles [cycle]             27,143,879   (-9.29%)
SOL L1/TEX Cache [%]    52.07  (-19.44%) | SM Active Cycles [cycle]        27,059,557.59   (-9.69%)
SOL L2 Cache [%]        15.73   (+9.58%) | SM Frequency [cycle/nsecond]             1.20   (-1.03%)
SOL DRAM [%]            49.29   (+9.58%) | DRAM Frequency [cycle/usecond]         848.96   (-0.43%)
```



```
🟦 Current      151 - conv_forward_kernel_shared_constant (10000, 16, 25)x(8, 8, 1) Time: 54.28 msecond Cycles: 64,899,036 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [738] m3
🟩 Baseline 17  143 - conv_forward_kernel_shared (10000, 16, 25)x(8, 8, 1)          Time: 63.12 msecond Cycles: 76,204,053 Regs: 39 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [740] m3

GPU Speed Of Light
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%]              95.32  (+13.47%) | Duration [msecond]                      54.28  (-14.01%)
SOL Memory [%]          81.15  (-16.11%) | Elapsed Cycles [cycle]             64,899,036  (-14.84%)
SOL L1/TEX Cache [%]    81.16  (-16.10%) | SM Active Cycles [cycle]        64,885,248.69  (-14.84%)
SOL L2 Cache [%]        22.56  (+16.69%) | SM Frequency [cycle/nsecond]             1.20   (-0.95%)
SOL DRAM [%]            64.82  (+16.68%) | DRAM Frequency [cycle/usecond]         847.43   (-0.34%)
```

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

I refer to the lecture talking about constant memory.

3. **Optimization 3: *<Shared memory matrix multiplication and input matrix unrolling>***

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

   I use the shared memory matrix multiplication and input matrix unrolling method. Because I think the former shared memory method is not that good, maybe a brand-new algorithm can help.

   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?

   This method uses a brand-new algorithm, it first transfers the original image data into new arrangement. Then a matrix multiplication between the kernel and the newly rearranged data can give the result. I think this may increase the performance, because we can do many other optimizations basing on the matrix multiplication such as shared memory tiling. This method has nothing to do with the former methods.

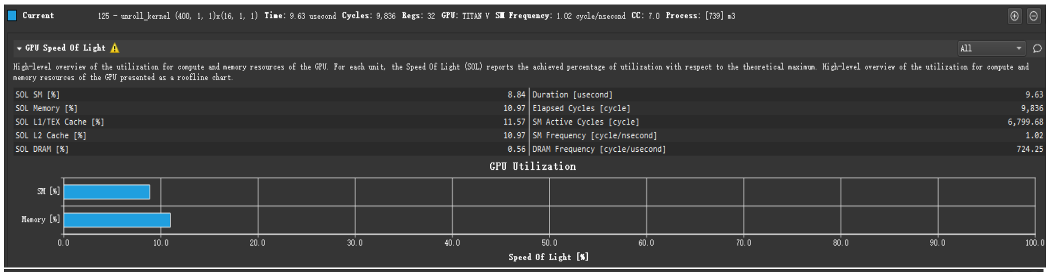
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Layer Time	Total Execution Time	Accuracy
100	*1.5268ms*	*2.291ms*	*13.291ms*	*1.316s*	*0.86*
1000	*14.3975ms*	*23.101ms*	*136.712ms*	*9.671s*	*0.886*
10000	*144.495ms*	*224.985ms*	*1353.618ms*	*1m31.671s*	*0.8714*

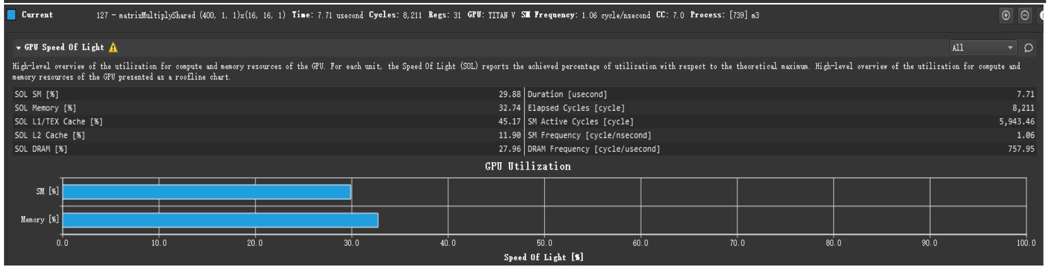
   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

   The result is not that well. The op time increases a lot. I think the reason is that when we rearrange the data from the input image data into the matrix operands, the order of the data is messed up and the merit of memory burst may have gone.

   From the Nsight-Compute, we can see that the utilization of SM and Memory is extremely low for both two kernels. And there are many loops to do all the jobs which means the program is more serially rather than parallelly.



```
Current     125 - unroll_kernel (400, 1, 1)x(16, 1, 1) Time: 9.63 usecond Cycles: 9,836 Regs: 32 GPU: TITAN V SM Frequency: 1.02 cycle/nsecond CC: 7.0 Process: [739] m3

GPU Speed Of Light                                                                                      All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%]                    8.84 | Duration [usecond]                  9.63
SOL Memory [%]               10.97 | Elapsed Cycles [cycle]             9,836
SOL L1/TEX Cache [%]         11.57 | SM Active Cycles [cycle]        6,799.68
SOL L2 Cache [%]             10.97 | SM Frequency [cycle/nsecond]        1.02
SOL DRAM [%]                  0.56 | DRAM Frequency [cycle/usecond]    724.25
```



```
Current     127 - matrixMultiplyShared (400, 1, 1)x(16, 16, 1) Time: 7.71 usecond Cycles: 8,211 Regs: 31 GPU: TITAN V SM Frequency: 1.06 cycle/nsecond CC: 7.0 Process: [739] m3

GPU Speed Of Light                                                                                      All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%]                   29.88 | Duration [usecond]                  7.71
SOL Memory [%]               32.74 | Elapsed Cycles [cycle]             8,211
SOL L1/TEX Cache [%]         45.17 | SM Active Cycles [cycle]        5,943.46
SOL L2 Cache [%]             11.90 | SM Frequency [cycle/nsecond]        1.06
SOL DRAM [%]                 27.96 | DRAM Frequency [cycle/usecond]    757.95
```

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

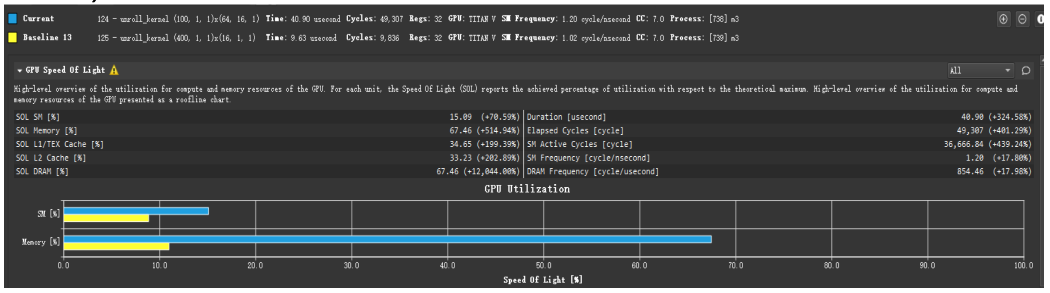
I refer to the lecture and the textbook talking about this technique.

f. Further modification (additional, not sure if this can count as points. Plz check it)

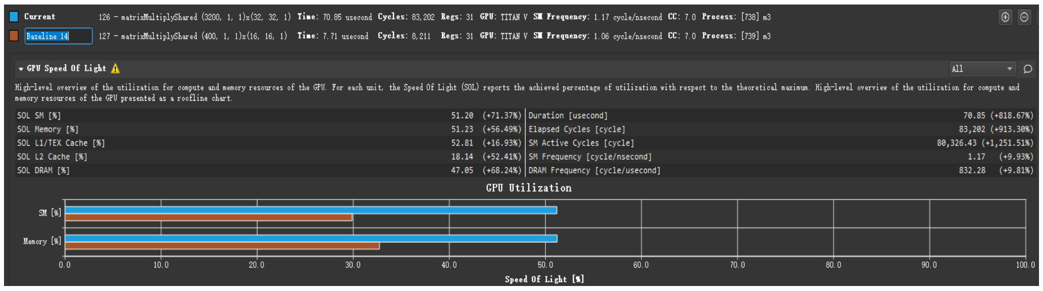
As the performance is extremely bad, I think I can do more modification on the data rearrangement. Originally, the method rearranges one image's data into a new matrix, but as our original data is considerably small, this rearrangement is not sufficient. So, I changed the algorithm a little bit, to make the rearrangement happen on several image together to produce one big matrix and do matrix multiplication. And do some index modification when storing the data back to the global memory (The code is in new-forward-mod-unroll.cu)

Batch Size	Op Time 1	Op Time 2	Total Layer Time	Total Execution Time	Accuracy
100	*0.6667ms*	*0.5234ms*	*1.1002ms*	*1.429s*	*0.86*
1000	*6.5232ms*	*5.2612ms*	*12.3212ms*	*9.582s*	*0.886*
10000	*66.7534ms*	*58.247ms*	*1107.559ms*	*1m29.431s*	*0.8714*

It's a surprise that the op time is much smaller than the original unrolling method. Although the time is still larger than the base line, this modification is a success. This can also be proved be the Nsight-Compute. For both kernel, there is a boost in SM and Memory utilization.



```
Current       124 - unroll_kernel (100, 1, 1)x(64, 16, 1) Time: 40.90 usecond Cycles: 49,307 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [738] m3
Baseline 13   125 - unroll_kernel (400, 1, 1)x(16, 1, 1)  Time: 9.63 usecond  Cycles: 9,836  Regs: 32 GPU: TITAN V SM Frequency: 1.02 cycle/nsecond CC: 7.0 Process: [739] m3

GPU Speed Of Light                                                                                      All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.
SOL SM [%]                  15.09  (+70.59%) | Duration [usecond]                  40.90 (+324.58%)
SOL Memory [%]              67.46 (+514.94%) | Elapsed Cycles [cycle]             49,307 (+401.29%)
SOL L1/TEX Cache [%]        34.65 (+199.39%) | SM Active Cycles [cycle]        36,666.84 (+439.24%)
SOL L2 Cache [%]            33.23 (+202.89%) | SM Frequency [cycle/nsecond]         1.20  (+17.80%)
SOL DRAM [%]             67.46 (+12,044.00%) | DRAM Frequency [cycle/usecond]     854.46  (+17.98%)
```



```
Current      126 - matrixMultiplyShared (3200, 1, 1)x(32, 32, 1) Time: 70.85 usecond Cycles: 83,202 Regs: 31 GPU: TITAN V SM Frequency: 1.17 cycle/nsecond CC: 7.0 Process: [738] m3
Baseline 14  127 - matrixMultiplyShared (400, 1, 1)x(16, 16, 1) Time: 7.71 usecond Cycles: 8,211 Regs: 31 GPU: TITAN V SM Frequency: 1.06 cycle/nsecond CC: 7.0 Process: [739] m3

GPU Speed Of Light
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and
memory resources of the GPU presented as a roofline chart.
SOL SM [%]                 51.20  (+71.37%) | Duration [usecond]                  70.85 (+818.67%)
SOL Memory [%]             51.23  (+56.49%) | Elapsed Cycles [cycle]             83,202 (+913.30%)
SOL L1/TEX Cache [%]       52.81  (+16.93%) | SM Active Cycles [cycle]        80,326.43 (+1,251.51%)
SOL L2 Cache [%]           18.14  (+52.41%) | SM Frequency [cycle/nsecond]         1.17   (+9.93%)
SOL DRAM [%]               47.05  (+68.24%) | DRAM Frequency [cycle/usecond]     832.28   (+9.81%)
```

4. **Optimization 4: *<Multiple kernel implementations for different layer sizes>***

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

   I use multiple kernel implementations for different layer sizes to optimize. When I am testing the former methods and trying with different TILE_WIDTH, I found that a big tile width is better for the first layer and a small tile width is better for the second layer. So, I think I can use different kernel configuration for different layer.

   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 add an "if" logic in my host code. If the size of the image is larger than 64, I will use 16 as the TILE_WIDTH and block dimension, or I will use 8. I think this will increase performance because if the input image is large, a larger tile may have better use of the threads, if the input image is small, a smaller tile may cause less control divergence. I think this method can add to any other methods because it only changes the size of kernel. I tested this method in many former implementations and the speed is always faster. When profiling, I use this method together with the shared memory method and constant weight method.

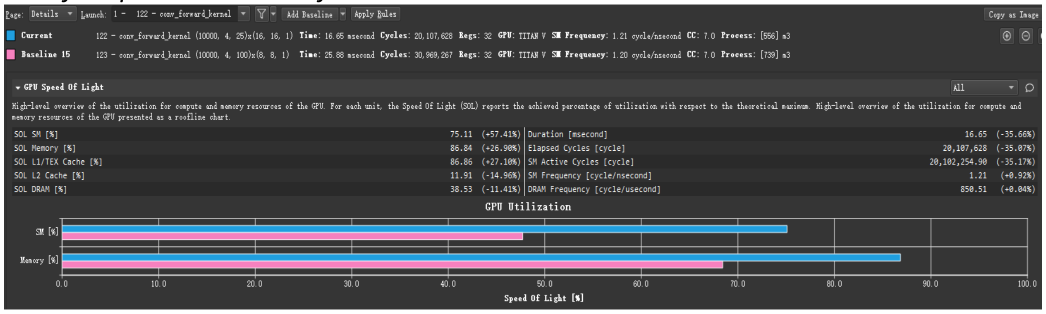
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Layer Time	Total Execution Time	Accuracy
100	*0.1454ms*	*0.5541ms*	*11.2353ms*	*1.361s*	*0.86*
1000	*1.5394ms*	*5.6181ms*	*113.131ms*	*9.212s*	*0.886*
10000	*15.1784ms*	*56.6811ms*	*1132.293ms*	*1m29.119s*	*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).

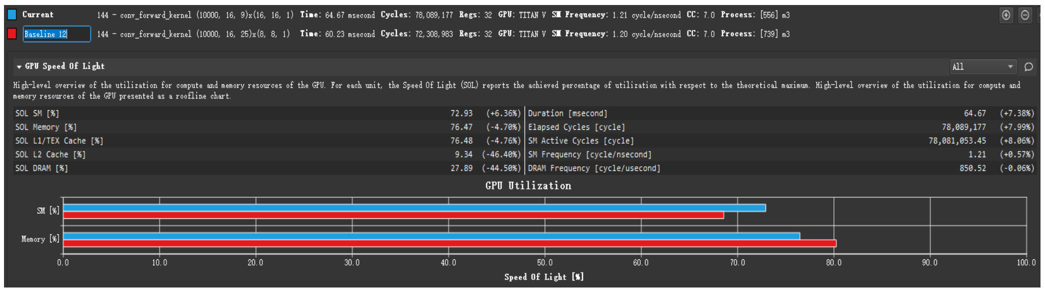
This optimization is successful as the total op time is smaller. Below is the profiling data I get when testing the different configurations for different layers.

*For the first layer, when using the block size of 16*16*1, the utilization of SM is 57% larger and the memory use is 27% larger comparing with the size of 8*8*1, and the time for op time1 reduced from 25.8 ms to 16.65 ms.*



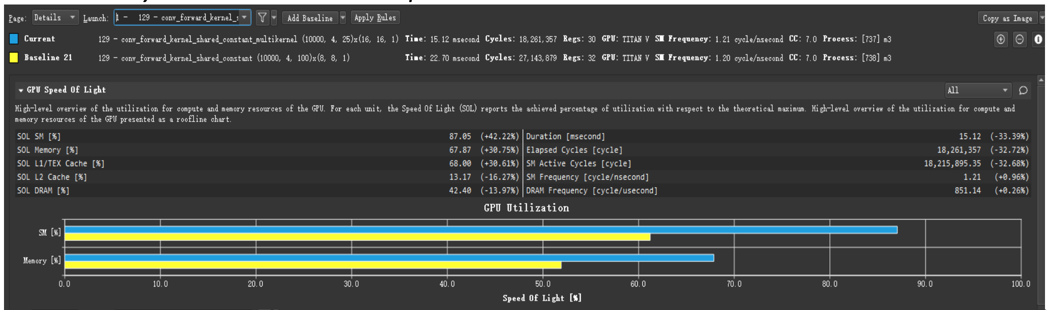
```
Page: Details   Launch: 1 - 122 - conv_forward_kernel   Add Baseline   Apply Rules                                        Copy as Image
Current      122 - conv_forward_kernel (10000, 4, 25)x(16, 16, 1) Time: 16.65 msecond Cycles: 20,107,628 Regs: 32 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [556] m3
Baseline 15  123 - conv_forward_kernel (10000, 4, 100)x(8, 8, 1)  Time: 25.88 msecond Cycles: 30,969,267 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [739] m3

GPU Speed Of Light                                                                                                        All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and
memory resources of the GPU presented as a roofline chart.
SOL SM [%]              75.11  (+57.41%) | Duration [msecond]                   16.65  (-35.66%)
SOL Memory [%]          86.84  (+26.90%) | Elapsed Cycles [cycle]          20,107,628  (-35.07%)
SOL L1/TEX Cache [%]    86.86  (+27.10%) | SM Active Cycles [cycle]     20,102,254.90  (-35.17%)
SOL L2 Cache [%]        11.91  (-14.96%) | SM Frequency [cycle/nsecond]          1.21   (+0.92%)
SOL DRAM [%]            38.53  (-11.41%) | DRAM Frequency [cycle/usecond]      850.51   (+0.04%)
```

*For the second layer, the utilization of SM and memory is almost the same for bock size of 16*16*1 and 8*8*1, but the op time 2 is 4ms smaller when using the size of 8*8*1.*

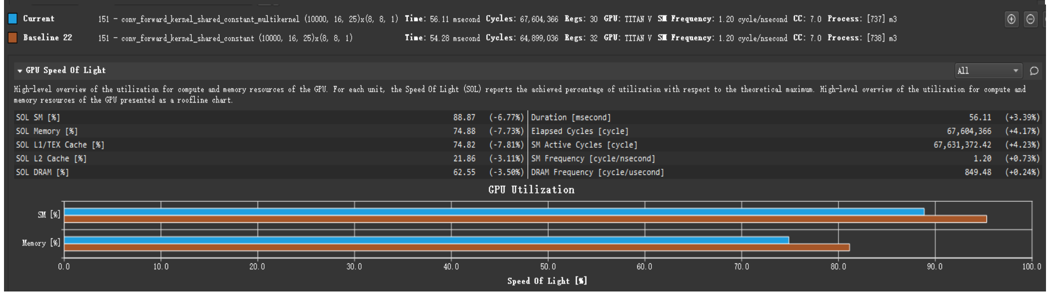


```
Current      144 - conv_forward_kernel (10000, 16, 9)x(16, 16, 1) Time: 64.67 msecond Cycles: 78,089,177 Regs: 32 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [556] m3
Baseline 12  144 - conv_forward_kernel (10000, 16, 25)x(8, 8, 1)  Time: 60.23 msecond Cycles: 72,308,983 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [739] m3

GPU Speed Of Light                                                                                                        All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and
memory resources of the GPU presented as a roofline chart.
SOL SM [%]              72.93   (+6.36%) | Duration [msecond]                   64.67   (+7.38%)
SOL Memory [%]          76.47   (-4.70%) | Elapsed Cycles [cycle]          78,089,177   (+7.99%)
SOL L1/TEX Cache [%]    76.48   (-4.76%) | SM Active Cycles [cycle]     78,081,053.45   (+8.06%)
SOL L2 Cache [%]         9.34  (-46.40%) | SM Frequency [cycle/nsecond]          1.21   (+0.57%)
SOL DRAM [%]            27.89  (-44.50%) | DRAM Frequency [cycle/usecond]      850.52   (-0.06%)
```

*So using block size of 16*16*1 in the first layer and 8*8*1 in the second will improve the performance. So I try to add this method to the former shared memory and constant weight optimization, and the result is good. The total op time is 71.3ms which is very close to the 70ms requirement.*



```
Page: Details   Launch: 1 - 129 - conv_forward_kernel_s   Add Baseline   Apply Rules                                      Copy as Image
Current      129 - conv_forward_kernel_shared_constant_multikernel (10000, 4, 25)x(16, 16, 1) Time: 15.12 msecond Cycles: 18,261,357 Regs: 30 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [737] m3
Baseline 21  129 - conv_forward_kernel_shared_constant (10000, 4, 100)x(8, 8, 1)              Time: 22.70 msecond Cycles: 27,143,879 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [738] m3

GPU Speed Of Light                                                                                                        All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and
memory resources of the GPU presented as a roofline chart.
SOL SM [%]              87.05  (+42.22%) | Duration [msecond]                   15.12  (-33.39%)
SOL Memory [%]          67.87  (+30.75%) | Elapsed Cycles [cycle]          18,261,357  (-32.72%)
SOL L1/TEX Cache [%]    68.00  (+30.61%) | SM Active Cycles [cycle]     18,215,895.35  (-32.68%)
SOL L2 Cache [%]        13.17  (-16.27%) | SM Frequency [cycle/nsecond]          1.21   (+0.96%)
SOL DRAM [%]            42.40  (-13.97%) | DRAM Frequency [cycle/usecond]      851.14   (+0.26%)
```



```
Current      151 - conv_forward_kernel_shared_constant_multikernel (10000, 16, 25)x(8, 8, 1) Time: 56.11 msecond Cycles: 67,604,366 Regs: 30 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [737] m3
Baseline 22  151 - conv_forward_kernel_shared_constant (10000, 16, 25)x(8, 8, 1)             Time: 54.28 msecond Cycles: 64,899,036 Regs: 32 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [738] m3

GPU Speed Of Light                                                                                                        All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and
memory resources of the GPU presented as a roofline chart.
SOL SM [%]              88.87   (-6.77%) | Duration [msecond]                   56.11   (+3.39%)
SOL Memory [%]          74.88   (-7.73%) | Elapsed Cycles [cycle]          67,604,366   (+4.17%)
SOL L1/TEX Cache [%]    74.82   (-7.81%) | SM Active Cycles [cycle]     67,631,372.42   (+4.23%)
SOL L2 Cache [%]        21.86   (-3.11%) | SM Frequency [cycle/nsecond]          1.20   (+0.73%)
SOL DRAM [%]            62.55   (-3.50%) | DRAM Frequency [cycle/usecond]      849.48   (+0.24%)
```

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

I refer to the data of my testing to raise up this method.

5. **Optimization 5: *<Input channel reduction: atomics>***

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

   I use input channel reduction method, because the dimension of the kernel is not fully used in the original code, and I think I can use z-dimension threads to make the computation in different channel parallelized. And I also need to use atomic addition to get rid of confliction.

   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?

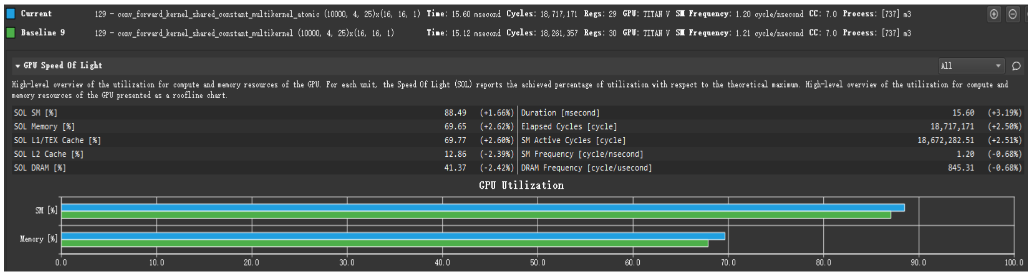
   I change the configuration of the kernel and add new z dimension threads to make the computation in different channel parallelized. When adding the result back to the global memory, I use atomic addition to get rid of confliction. I think this would increase performance because it makes the calculation more paralleled. This optimization can build on any former method with a channel loop. As the performance for the optimization "shared memory, constant weight, Multiple kernel for different layer" is good, I add this new optimization onto the former 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	*0.281ms*	*0.661ms*	*1.419s*	*0.86*
1000	*2.792ms*	*6.819ms*	*9.172s*	*0.886*
10000	*28.218ms*	*69.0906ms*	*1m31.291s*	*0.8714*

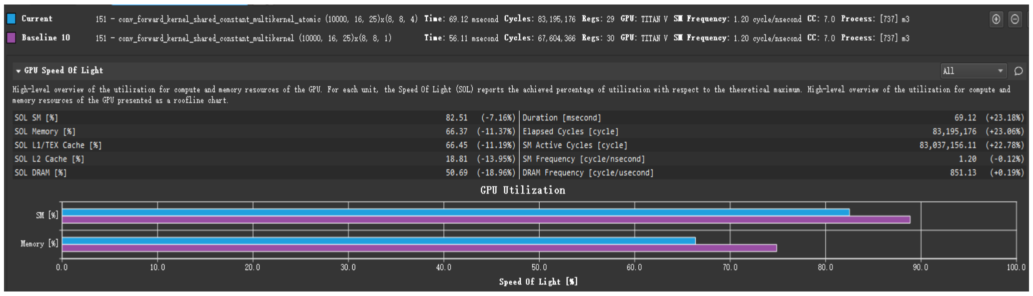
   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

   As for the first layer, the input channel is 1, so there is no difference when using input channel reduction



```
Current     129 - conv_forward_kernel_shared_constant_multikernel_atomic (10000, 4, 25)x(16, 16, 1) Time: 15.60 msecond Cycles: 18,717,171 Regs: 29 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [737] m3
Baseline 9  129 - conv_forward_kernel_shared_constant_multikernel (10000, 4, 25)x(16, 16, 1)        Time: 15.12 msecond Cycles: 18,261,357 Regs: 30 GPU: TITAN V SM Frequency: 1.21 cycle/nsecond CC: 7.0 Process: [737] m3

▾ GPU Speed Of Light                                                                                   All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and
memory resources of the GPU presented as a roofline chart.
SOL SM [%]                 88.49   (+1.66%) | Duration [msecond]                      15.60  (+3.19%)
SOL Memory [%]             69.65   (+2.62%) | Elapsed Cycles [cycle]             18,717,171  (+2.50%)
SOL L1/TEX Cache [%]       69.77   (+2.60%) | SM Active Cycles [cycle]        18,672,282.51  (+2.51%)
SOL L2 Cache [%]           12.86   (-2.39%) | SM Frequency [cycle/nsecond]             1.20  (-0.68%)
SOL DRAM [%]               41.37   (-2.42%) | DRAM Frequency [cycle/usecond]         845.31  (-0.68%)
```

As for the second layer, the input channel is 4, so the dimension of z will increase to 4. But from the Nsight-Compute, we can see that the performance becomes worse, and more time is needed.



```
Current      151 - conv_forward_kernel_shared_constant_multikernel_atomic (10000, 16, 25)x(8, 8, 4) Time: 69.12 msecond Cycles: 83,195,176 Regs: 29 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [737] m3
Baseline 10  151 - conv_forward_kernel_shared_constant_multikernel (10000, 16, 25)x(8, 8, 1)        Time: 56.11 msecond Cycles: 67,604,366 Regs: 30 GPU: TITAN V SM Frequency: 1.20 cycle/nsecond CC: 7.0 Process: [737] m3

▾ GPU Speed Of Light                                                                                   All
High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and
memory resources of the GPU presented as a roofline chart.
SOL SM [%]                 82.51   (-7.16%) | Duration [msecond]                      69.12 (+23.18%)
SOL Memory [%]             66.37  (-11.37%) | Elapsed Cycles [cycle]             83,195,176 (+23.06%)
SOL L1/TEX Cache [%]       66.45  (-11.19%) | SM Active Cycles [cycle]        83,037,156.11 (+22.78%)
SOL L2 Cache [%]           18.81  (-13.95%) | SM Frequency [cycle/nsecond]             1.20  (-0.12%)
SOL DRAM [%]               50.69  (-18.96%) | DRAM Frequency [cycle/usecond]         851.13  (+0.19%)
```

I think this is because although the input channel is computed parallelly, the threads also need to wait in line when doing the atomic addition. Also, the total number of threads in one block is limited by 1024, if more are allocated in the z dimension, less will be allocated in the x-y plate.

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

   I refer to the lecture talking about reduction and histogram.

6. **Optimization 6: *<Using Streams to overlap computation with data transfer>***

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

      I use streams to overlap computation with data transfer. I use this method because I think continue to optimize op time is kind of hard and there are some optimization spaces for the data transferring time in host code.

   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?

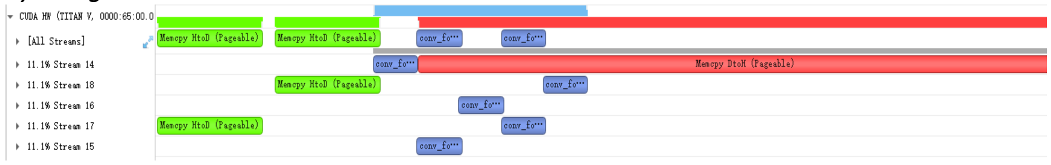
      I use many streams to make the memory copy and calculation asynchronized, and this will make the computation and data transferring of different streams occur at the same time. I think this optimization can add to any former methods. When profiling, I use this method above the constant weight method.

   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	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	*5.719ms*	*4.499ms*	*1.398s*	*0.86*
1000	*58.129ms*	*43.142ms*	*9.019s*	*0.886*
10000	*588.248ms*	*429.997ms*	*1m28.111s*	*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).

I run the test many times and find that this method will increase the speed and synergize with any of previous optimizations. Also, I find that the performance is the best when the stream number is 5. From the Nsight-system, we can see that by using the method of overlap, different streams can do the computation and copying at the same time. From the layer time data we can find that the total layer time is reduced by using this method.



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

I use this technique basing on the knowledge from the class.

7. **Optimization 7: *<Combine different methods together (Pin memory method is added)>***

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

I use constant memory for weight, Multiple kernel implementations for different layer sizes, Streams to overlap computation with data transfer, sweeping various parameters to find best values, and pin memory to implement the fastest version. I choose this combination as my final result basing on tons of running test, the log is in my code folder.

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?

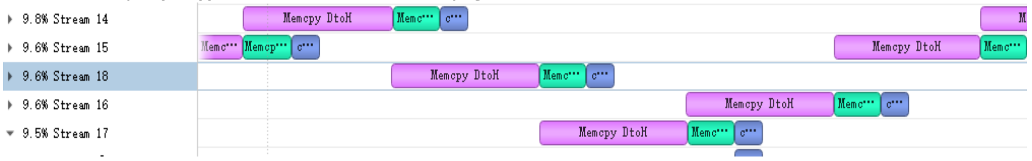
First, the weight data is stored into the constant memory, so each time I need the weight data, it can be fetched from the fast constant memory. Then, the kernel size of each layer is specially chosen (16 for big picture, 8 for small picture). Next, use five streams to do overlapping, and loop 10 times, so each kernel will calculate 200 images at one time (this configuration is determined by many tests, logs are also in the ./log folder). Lastly, pin the device memory to avoid further memory copying.

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).

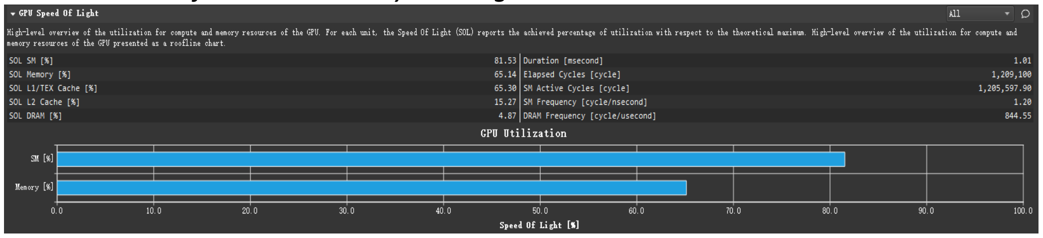
Batch Size	Layer time 1	Layer Time 2	Total Execution Time	Accuracy
100	*2.518ms*	*1.841ms*	*1.419s*	*0.86*
1000	*26.234ms*	*18.192ms*	*9.121s*	*0.886*
10000	*260.575ms*	*187.767ms*	*1m26.341s*	*0.8714*

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).

The optimization is extremely successful in improving performance. The total layer time is in top three of the class. Basing on all the former optimization trails, each method I use in this compound optimization can somehow optimize the speed. No wonder this combination can boost the final speed. By looking at the nsys, I find that originally, the time is mostly consumed on the memory copying (97% for first layer, 89% for second layer). By using the pin memory method, this time is reduced visibly (87% for first layer, 62% for second layer).
The overlap of different streams is very good.



The utilization of SM and Memory is also good.


GPU Speed Of Light

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

SOL SM [%]	81.53	Duration [msecond]	1.01
SOL Memory [%]	65.14	Elapsed Cycles [cycle]	1,209,100
SOL L1/TEX Cache [%]	65.30	SM Active Cycles [cycle]	1,205,597.90
SOL L2 Cache [%]	15.27	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	4.87	DRAM Frequency [cycle/usecond]	844.55

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

Experience from all other optimizations above.