Name: *<Hongqing Liu>*
NetID: *<hl85>*
Section: *<AL2>*

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	*<0.176106 ms>*	*<0.638516>*	*<0m1.198s>*	*<0.86>*
1000	*<1.71601 ms>*	*<11.4767 ms>*	*<0m10.256s>*	*<0.886>*
10000	*<16.1471 ms>*	*<63.5251 ms>*	*<1m41.293s>*	*<0.8714>*

1. **Optimization 1: <Weight matrix (kernel values) in constant memory> *4***

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

    The first optimization I use is **Weight matrix (kernel values) in constant memory**.

    Weight matrix is using in the convolution section, it will be frequently read by GPU. If we set the weight matrix in shared memory or constant memory, it will cost some time to read it. However, if we set it in constant memory, it can be read 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?

This optimization load weight matrix into shared memory. I think this optimization will have very good performance since constant memory is read-only and it supports boardcasting a single value to each thread in the thread bundle. It is the fastest way to read constant variables to GPU, as far as I learn. This optimization is the first optimization I choose, so it is independent.

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).

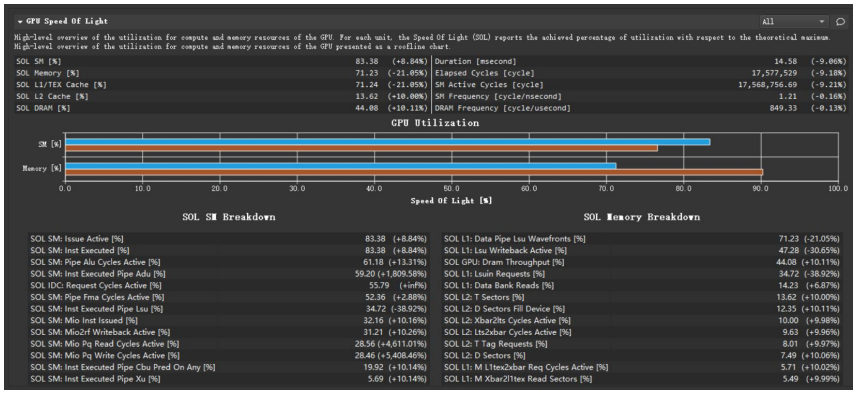
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.160633 ms*	*0.575336 ms*	*0m1.135s*	*0.86*
1000	*1.48362 ms*	*5.64619 ms*	*0m10.781s*	*0.886*
10000	*14.6362 ms*	*56.7935 ms*	*1m45.129s*	*0.8714*

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).

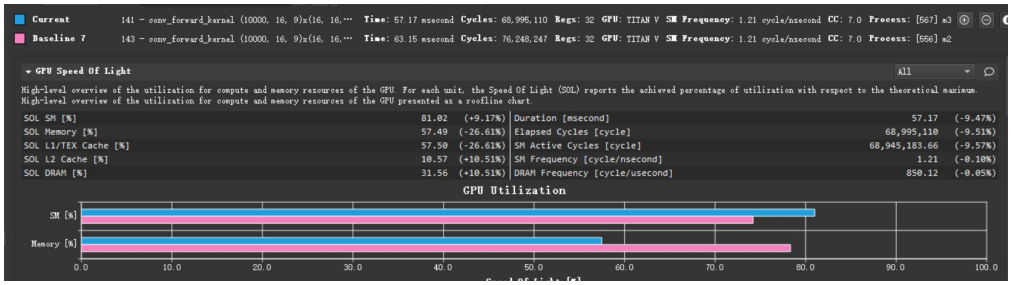
I think this optimization is successful in improving performance, both in OP time for layer 1 and layer 2. This optimization is compared with baseline.

GPU Speed of Light: We can see that for layer 1 and layer 2, the memory decrease while SM increase. This is because we use constant memory to store weight matrix.

Layer1:

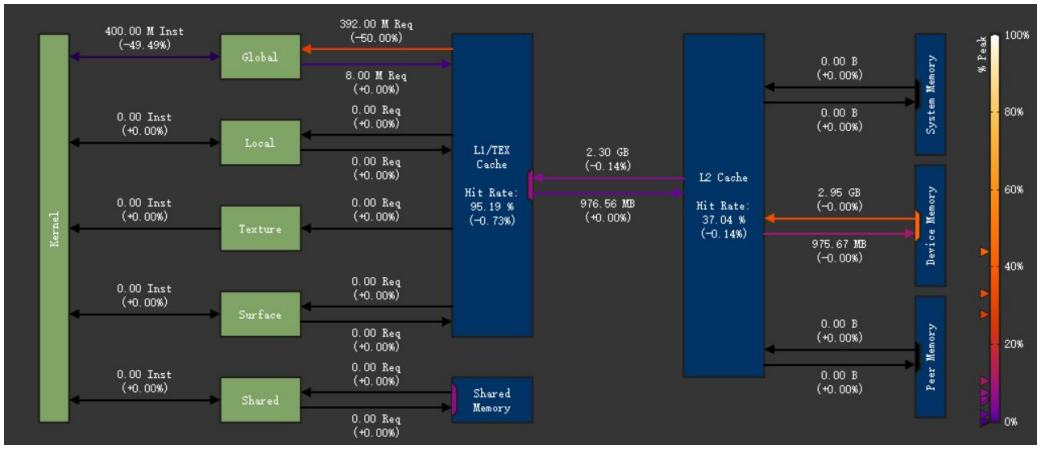


Layer2:

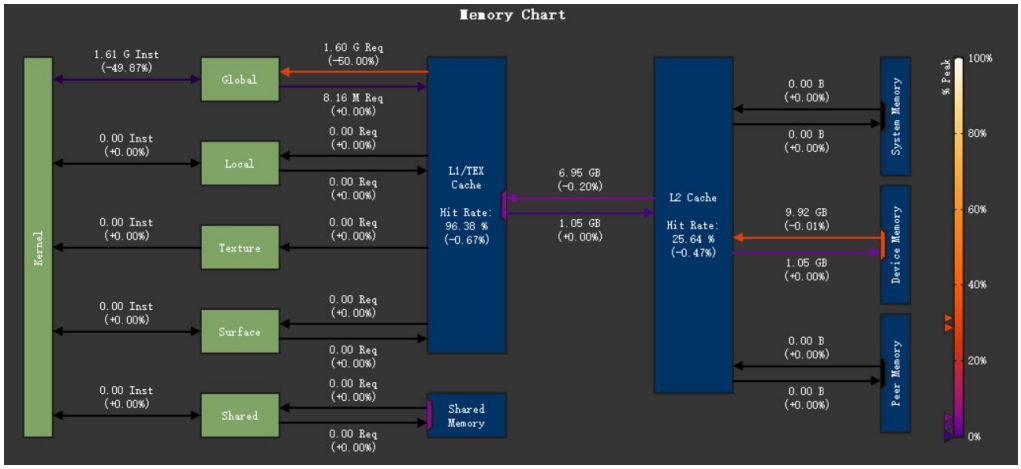


Memory Workload Analysis: We can see that the memory from L1 Cache to Global decrease, this can be easily explained since we use constant memory to store weight matrix instead of using shared memory.

Layer1:



Layer2:



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

Text book.

2. **Optimization 2: <Tiled shared memory convolution>** ***1 (1 + 4)***

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

      I use Tiled shared memory convolution with Weight matrix (kernel values) in constant memory in this section. When we do convolution computation, GPU need to read a single element in input matrix for many times. If we store input matrix in global memory, which takes more time to read in than shared memory, it may take a lot of time to read it. In that case, I decide to use shared memory convolution to improve the 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?

      This optimization use shared memory to store input array in order to speed up the input reads. Shared memory can be shared in each block and it is much faster to read than global memory. In each kernel, we load relevant global memory into shared memory before we do the convolution part. I use this optimization in the basic of optimization of constant memory.

   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).

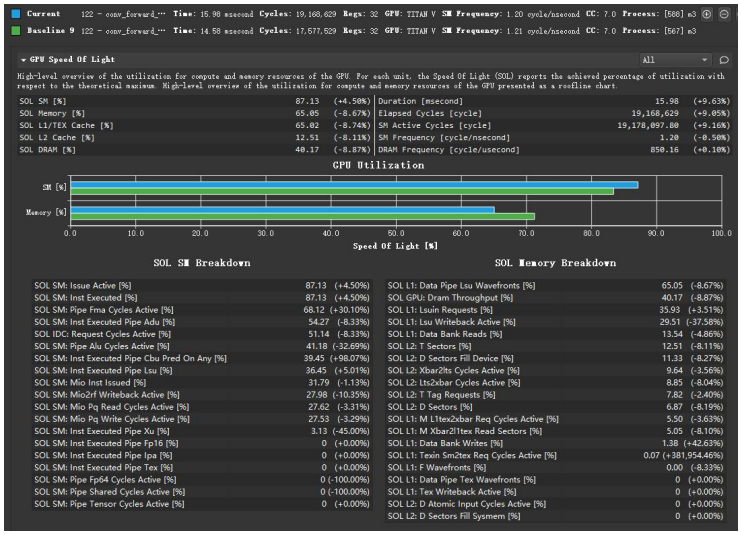
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.162511 ms*	*0.64941 ms*	*0m1.164s*	*0.86*
1000	*1.53752 ms*	*6.43267 ms*	*0m10.040s*	*0.886*
10000	*15.1768 ms*	*64.1261 ms*	*1m39.354s*	*0.8714*

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).

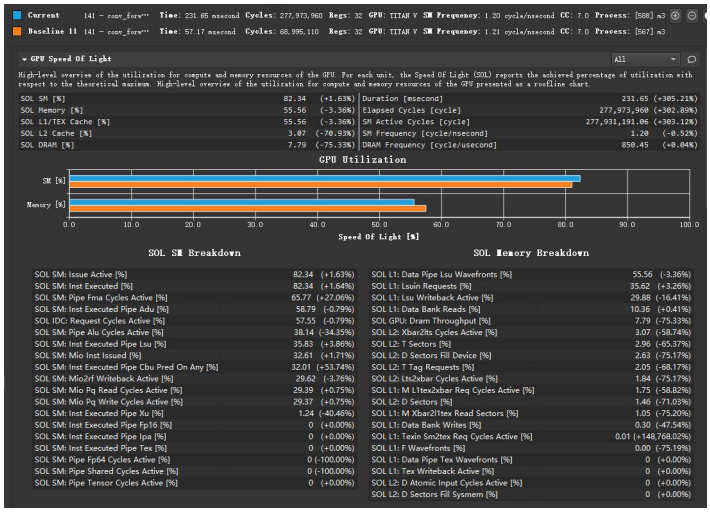
I do not think that this optimization is successful since it does not have a significnat performance in Layer 1 and have a worse performance in Layer2.

GPU Speed of Light: For layer 2, the duration increase 305.21%. The reason why is happened is that although using shared memory can reduce time that GPU load data from global memory, which can save some time, this optimization add a section to load data from global memory to shared memory. The __syncthreads() after this section may increase a lot of latency.

Layer1:

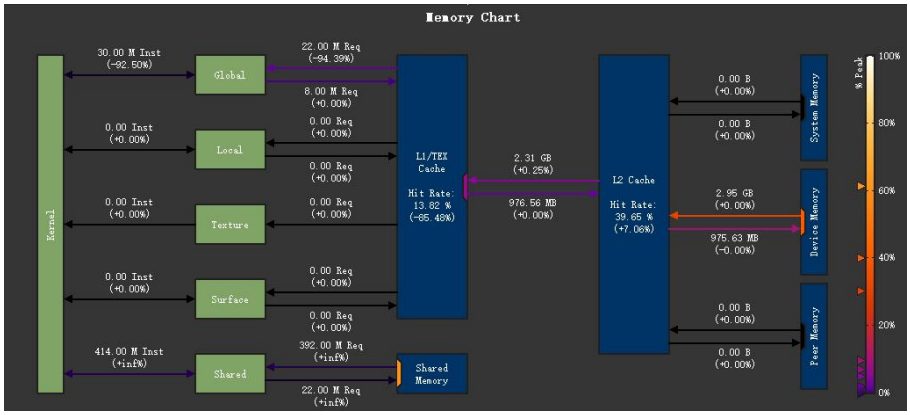


Layer2:

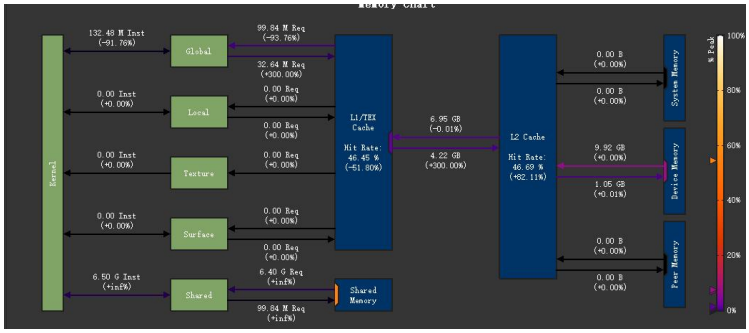


Memory Workload Analysis: since it use shared memory, the use of global memory decreases while the use of shared memory increases.

Layer1:



Layer2:



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

Text book

3. **Optimization 3: <Input channel reduction: tree>**
(Delete this section blank if you did not implement this many optimizations.)

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

I use Weight matrix (kernel values) in constant memory and Input channel reduction tree in this section.

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?

Since shared memory does not have a good performance, I decide to use input channel reduction tree on the basis of constant memory for weight matrix.

The basic idea is that using different threads to calculate different channel and add them together, this way can reduce time complexity in this part form O(n) to O(log n).

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).

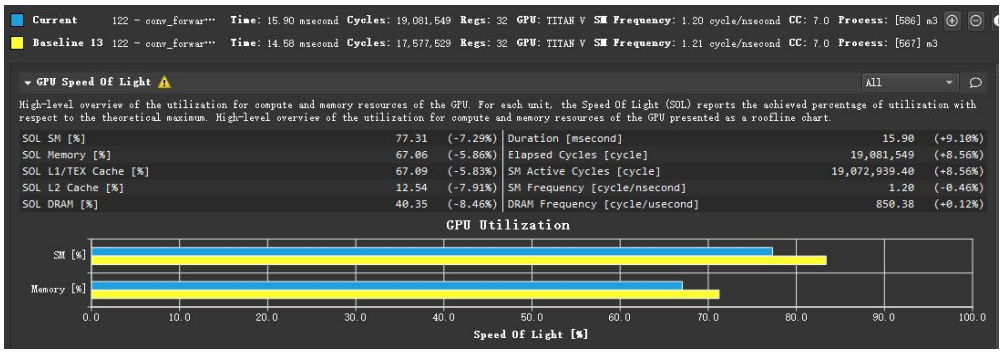
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.162819 ms*	*0.818721 ms*	*0m1.183s*	*0.86*
1000	*1.6055 ms*	*8.75642 ms*	*0m10.068s*	*0.886*
10000	*15.9255 ms*	*87.909 ms*	*1m39.388s*	*0.8714*

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).

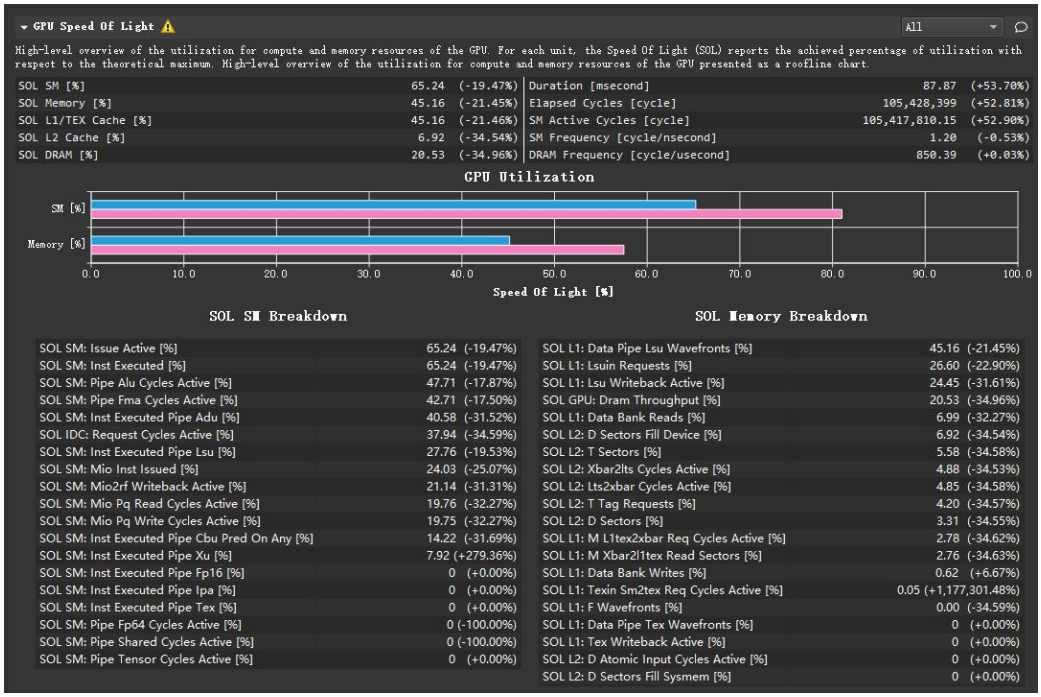
This optimization does not have a good performance, the reason of this is that there are 2 __syncthreads() there which brings it a lot of latency. What's more, the channel is three in this situation. The original times to add between channel is 3 and the optimized times to add between channel is 2, which doe not have a large improve. In that case, this optimization have a worse performance.

GPU speed of Light: both SM and memory decrease while duration of increase a little bit.

Layer1:

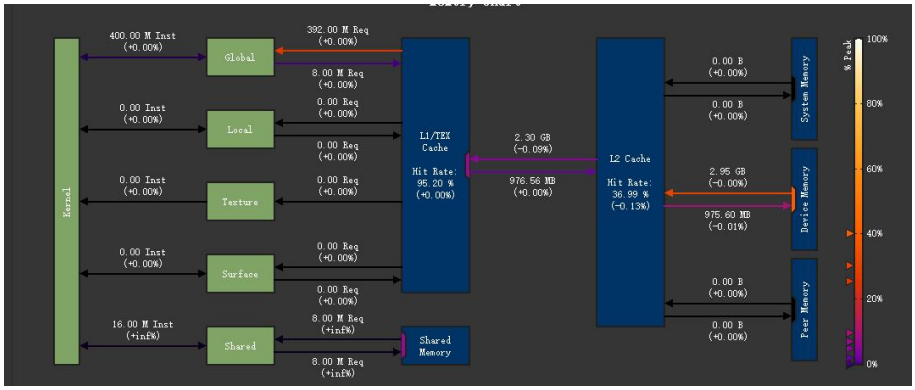


Layer2:

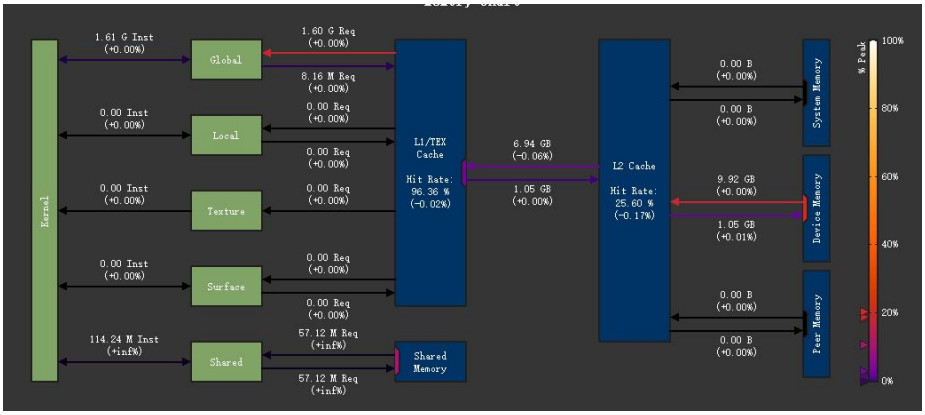


Memory workload Analyse:

Layer1:



Layer2:



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

Text book

4. **Optimization 4: <Input channel reduction: atomics>** ***9 (4+9)***

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

I use Weight matrix (kernel values) in constant memory and Input channel reduction atomics in this section.

Atmoics can improve system's performance by replacing __syncthreads();

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?

The atomic operation of CUDA can be understood as the execution process of a minimum unit of the three operations of "read-modify-write" on a variable. This execution process cannot be decomposed into smaller parts. During its execution , does not allow other parallel threads to read and write to this variable. Based on this mechanism, atomic operations implement mutual exclusion protection for variables shared among multiple threads, ensuring the correctness of the results of any operation on variables.

I do not think this optimization would have a better performance in this situation since I compare it in basic algorithm with constant memory, which does not have __syncthreads().

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).

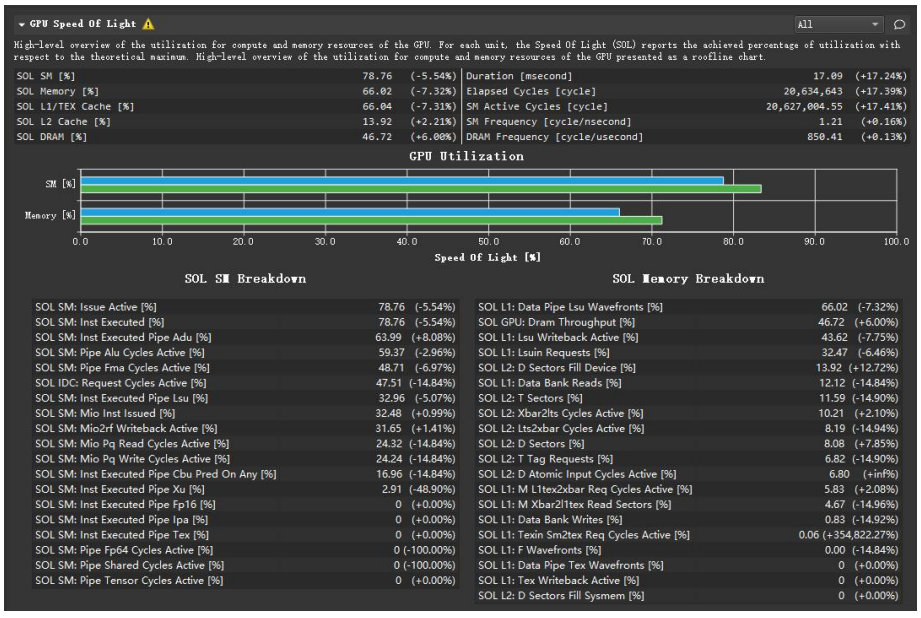
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.174494 ms*	*0.785904 ms*	*0m1.191s*	*0.86*
1000	*1.70954 ms*	*8.40849 ms*	*0m10.019s*	*0.886*
10000	*17.2063 ms*	*85.5182 ms*	*1m38.691s*	*0.8714*

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).

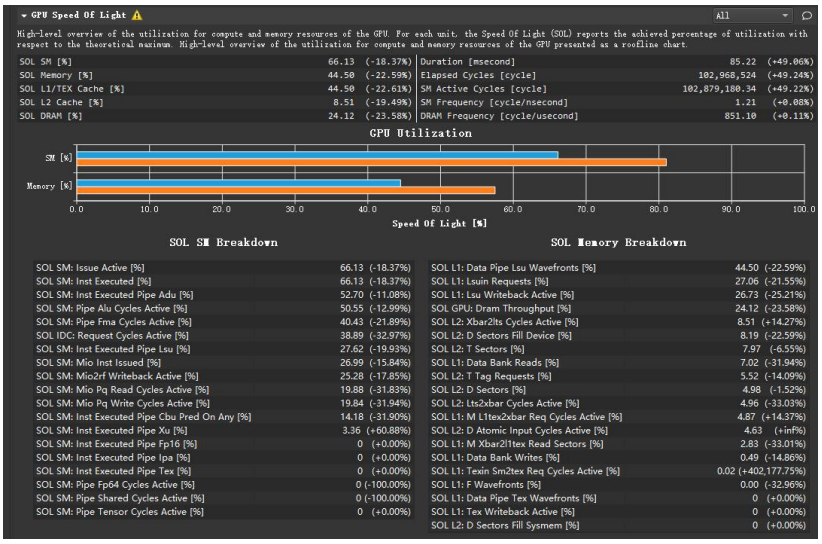
This optimization does not have a good performance, the reason is explained above.

GPU speed of Light: both SM and memory decrease while duration of increase a little bit.

Layer1:

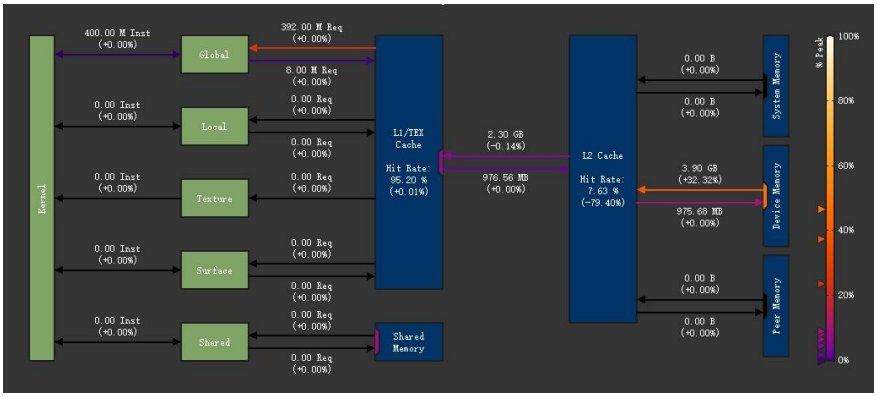


Layer2:

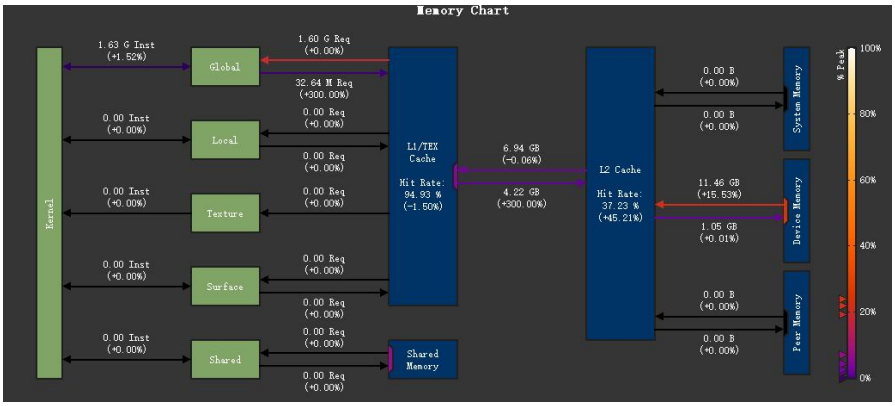


Memory workload Analyse: no memory change between L1 and kernel but have huge increase between L2 and device memory.

Layer1:



Layer2:



What references did you use when implementing this technique?

1. https://blog.csdn.net/dcrmg/article/details/54959306

2. Text book

5. **Optimization 5: <Shared memory matrix multiplication and input matrix unrolling> 2**

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

   I use **Shared memory matrix multiplication and input matrix unrolling** *in this section.*

   This method can reshape the input matrix and increase element replication.

   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?

   Firstly I use a kernel to reshape the matrix into an unroll form, this kernel would output the unroll matrix. Then I use another kernel to do the matrix multiplication. This method will use shared memory so that it can decrease the time that data transport from global memory to GPU.

   I think this optimization will have a good performance, since it increase the time each element in input matrix replicates.

   This method does not synergize with any of other previous optimization so that it will be compared with the original project.

   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	1.64247 ms	*2.32911 ms*	*0m1.192s*	*0.86*
1000	*16.3857 ms*	*22.2854 ms*	*0m10.117s*	*0.886*
10000	*162.93 ms*	*221.627 ms*	*1m39.075s*	*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).

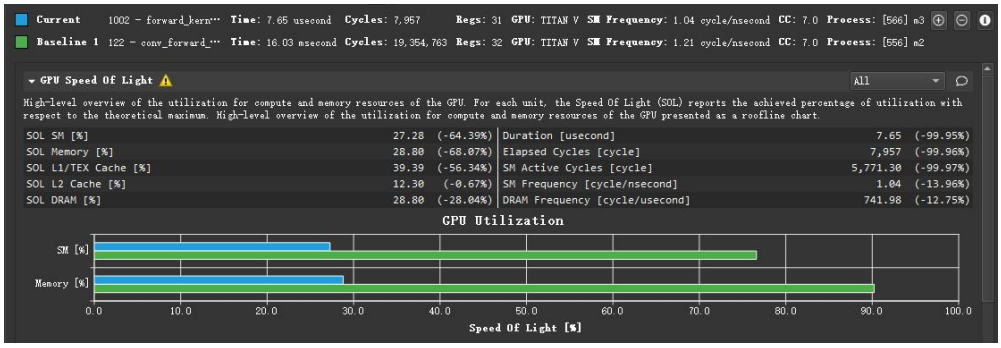
This algorithm has a very bad performance, it has multiple reasons:

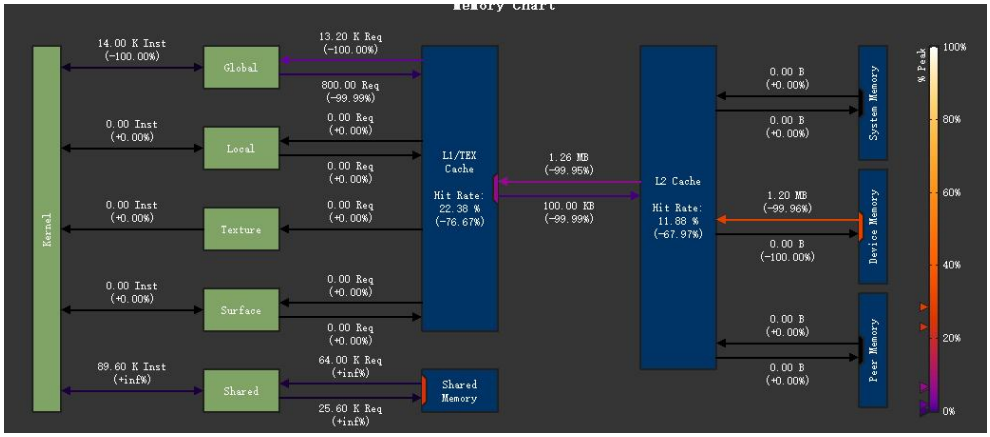
*Firstly, unroll_launch kernel needs a lot of time to unroll the input matrix not only because the input data is very large but also because it use loop to unroll input, which needs to loop for channel *K*K/TILE_WIDTH times*

Another reason is that the since the system use two kernels and they are both launched for Batch times, so there will cost a lot time for information transport and system latency between CPU and GPU.

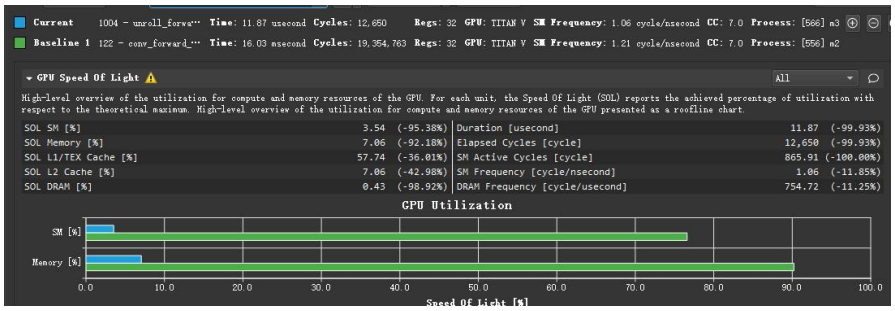
GPU speed of Light: Since it use 2 kernels for many times, it is hard to complete with baseline kernel for its performance.

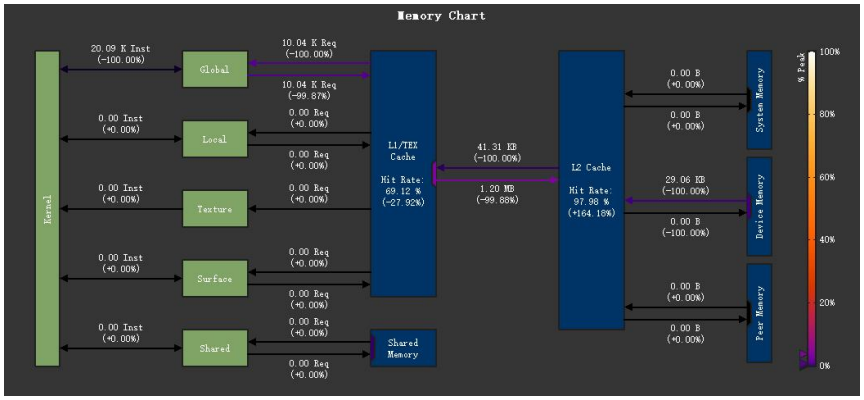
Matrix multiple kernel:





Unrolled kernel:





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

Text book

6. **Optimization 6: <kernel fusion for unrolling and matrix multiplication> 3**

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

I use **kernel fusion for unrolling and matrix multiplication** *in this part.*

This method can combine two kernel into one so that the time spend for information transport between kernel and host will reduce to 1 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 combine the unrolling kernel and matrix multiplication into one kernel and it will only launched for one time for each grid.

This one will have a good performance since it can reduce the time spend for information transport between kernel and host.

This method does not synergize with any of other previous optimization so that it will be compared with the original project.

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).

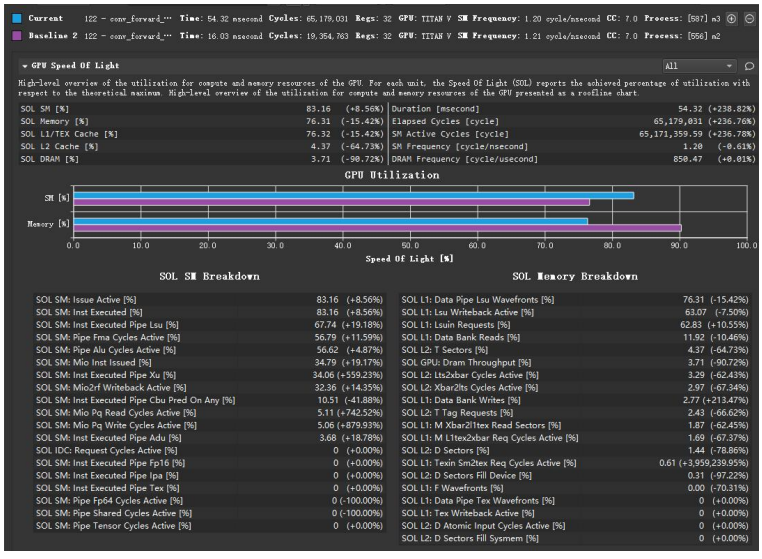
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.557903 ms*	*0.349553 ms*	*0m2.367s*	*0.86*
1000	*5.44795 ms*	*3.29176 ms*	*0m10.461s*	*0.886*
10000	*54.3284 ms*	*32.74 ms*	*1m38.915s*	*0.8714*

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).

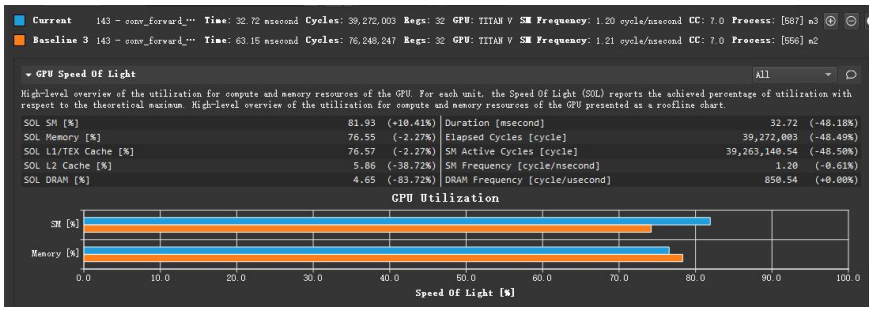
The performance is improved for the second layer while decrease for the first layer, as shown below. The reason of this might be that since the first layer has less data,unroll multiply may not have a significant improvement influence while unroll section cost some computing time. On the contrast, for the second layer, unroll multiply have saved a lot of time.

GPU speed of Light: for layer 1, SM increase while memory decrease, duration increase significantly; for layer 2 duration decrease significantly.

Layer1:

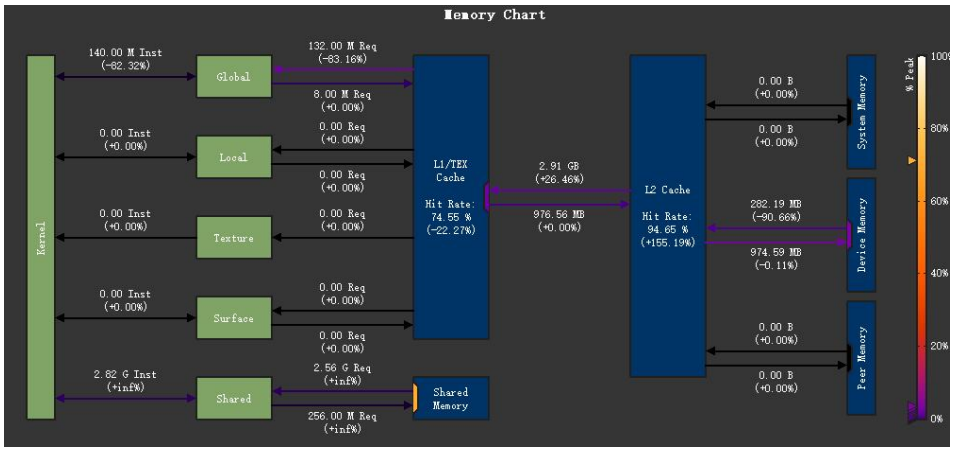


Layer2:

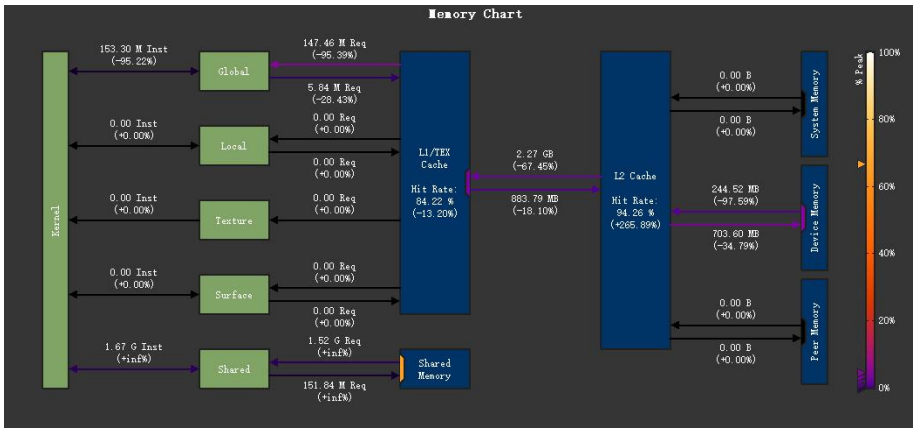


Memory workload Analyse: since it use share memory, memory from L1 to global reduced while shared memory increase.

Layer1:



Layer2:



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

Text book

https://github.com/dmlc/nnvm-fusion

7. **Optimization 7: <Fixed point FP16> 3 + 10**

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

I use FP16 with kernel fusion here.

FP16 has less bits so it occupies less memory and has faster computing time, in that case, it can improve the performance.

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 float point 16 to do the calculate instead of using float 32. this might have some accuracy problem, but it can make the system speed up.

I use this optimization on the basic of optimization 6, **<kernel fusion for unrolling and matrix multiplication> .**

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).

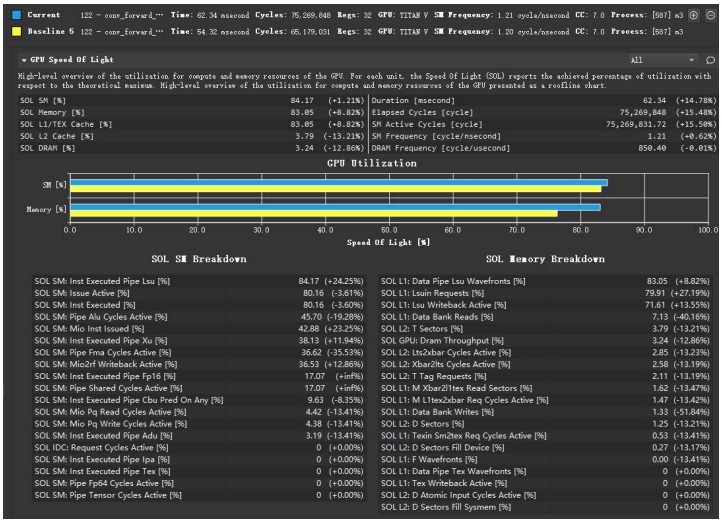
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.64411 ms*	*0.409405 ms*	*0m2.643s*	*0.86*
1000	*6.29102 ms*	*3.79444 ms*	*0m13.219s*	*0.887*
10000	*62.3529 ms*	*37.7198 ms*	*1m40.003s*	*0.8716*

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).

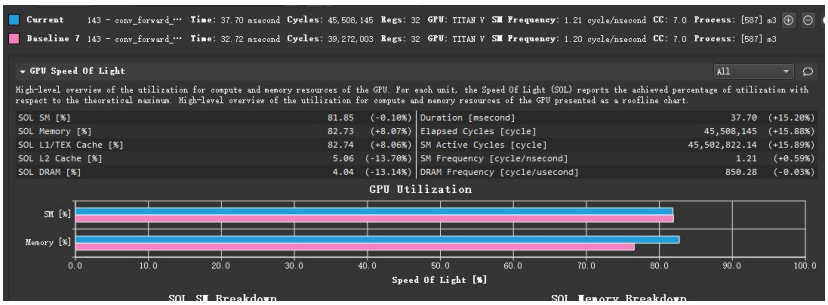
The performance decrease for both the first layer and second layer, as shown below.

GPU speed of Light:SM and memory increase, duration increase significantly

Layer1:

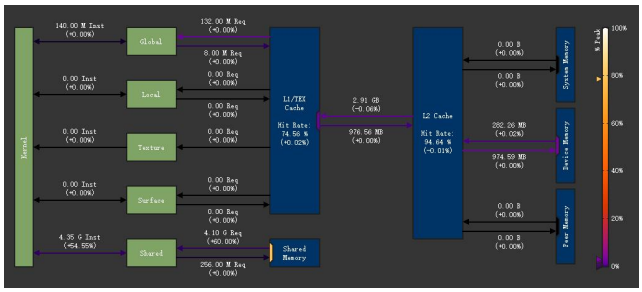


Layer2:



Memory workload Analyse: memory allocate does not change.

Layer1:



Layer2:

What references did you use when implementing this technique?

https://datascience.stackexchange.com/questions/73107/fp16-fp32-what-is-it-all-about-or-is-it-just-bitsize-for-float-values-pytho

8. **Optimization 8: <Multiple kernel implementations for different layer sizes > 7**

   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 since the layer 1 has best performance with constant memory while layer2 has best performance with kernel fusion for unrolled and matrix multiplication.

   In this way, different layer use different kernel, the system can has a best 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?

   This optimization must have a better performance, since the layer 1 has best performance with constant memory while layer2 has best performance with kernel fusion for unrolled and matrix multiplication.

   In this way, different layer use different kernel, the system can has a best performance.

   This codes are synergize with constant memory and kernel fusion for unrolled and matrix multiplication.

   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	*0.159577 ms*	*0.339416 ms*	*0m1.174s*	*0.86*
1000	*1.57053 ms*	*3.20037 ms*	*0m10.219s*	*0.886*
10000	*15.5764 ms*	*29.0843 ms*	*1m42.581s*	*0.8714*

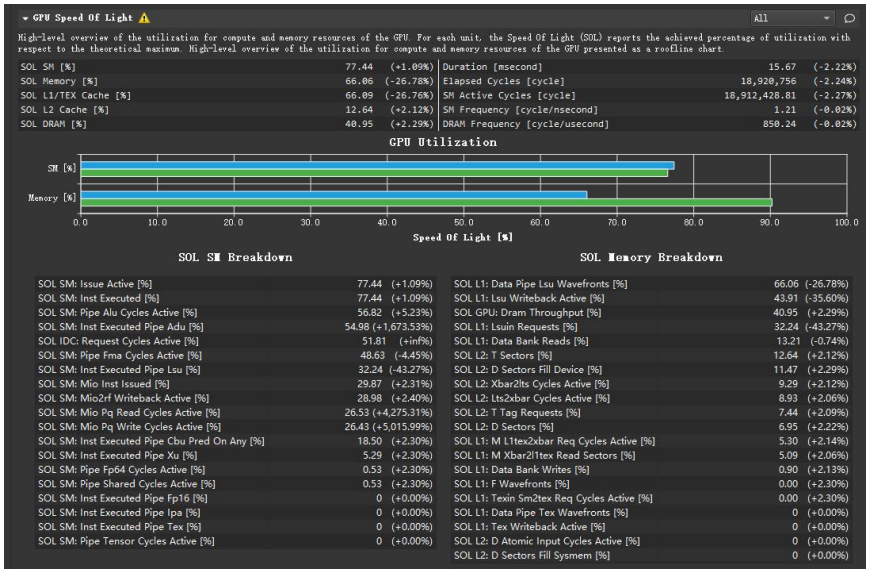
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

The performance of this optimization is perfect.

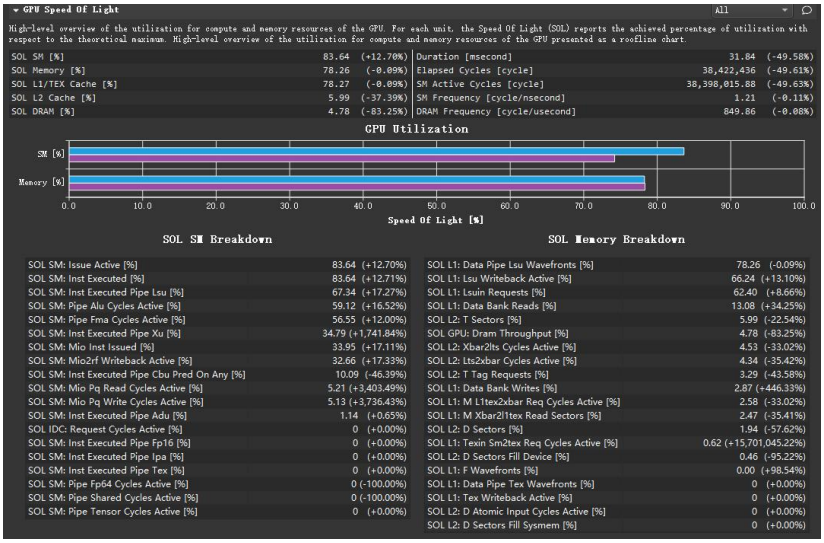
GPU speed of Light: for layer1, it has same performance with constant memory optimization.

For layer2 it has same performance with kernel fusion optimization

Layer1:

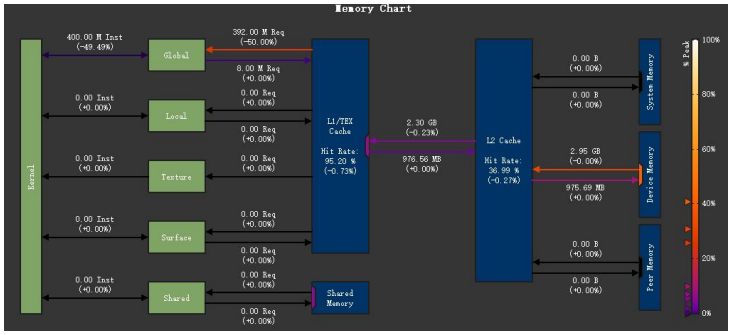


Layer2:

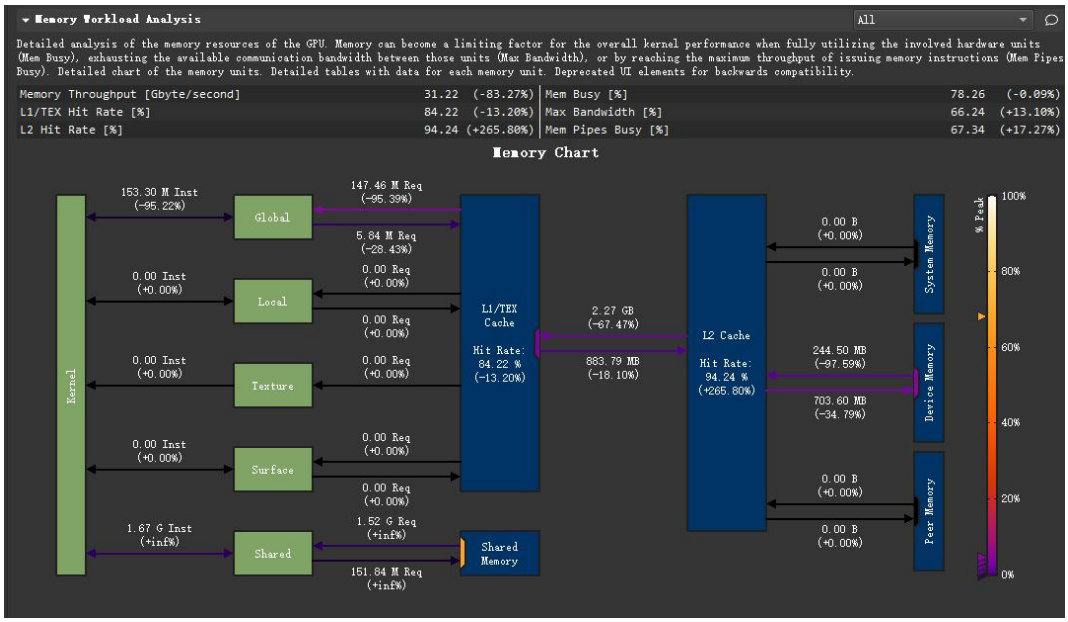


Memory workload Analyse: memory allocate does not change.

Layer1:



Layer2:



What references did you use when implementing this technique?

Text book.