Name: *Qiqiain Fu*
NetID: *qiqianf2*
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 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. Note: **Do not** use batch size of 10k when you profile in *--queue rai_amd64_exclusive*. We have limited resources, so any tasks longer than 3 minutes will be killed. Your baseline M2 implementation should comfortably finish in 3 minutes with a batch size of 5k (About 1m35 seconds, with nv-nsight).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.17711ms*	*0.662171ms*	*0m5.974s*	*0.86*
1000	*1.68443ms*	*6.83812ms*	*0m54.411s*	*0.886*
5000	*8.33635ms*	*27.0691ms*	*4m24.008s*	*0.871*

1. **Optimization 1: Using Streams to overlap computation with data transfer**

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

   The optimization I choose is to use stream because it allows concurrent execution, which can overlap data transfers between the host and device with kernel execution. Therefore, I think it can reduce the time of execution greatly.

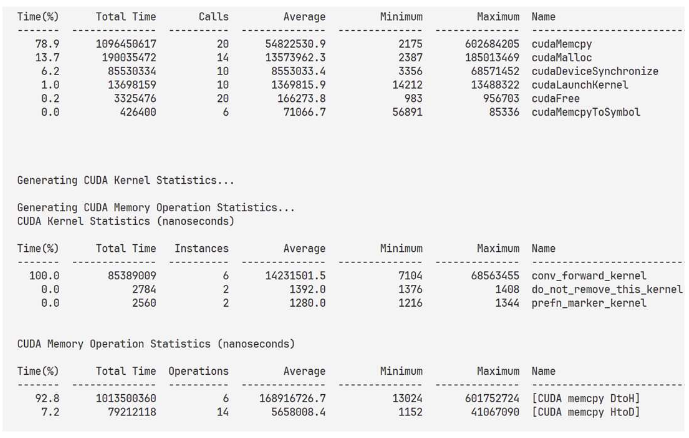
   b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

I think it should work and increase the performance greatly. As I mentioned, it can overlap data transfers between the host and device with kernel execution, which is quite time consuming in the baseline. It doesn't synergize with any of my previous optimizations.

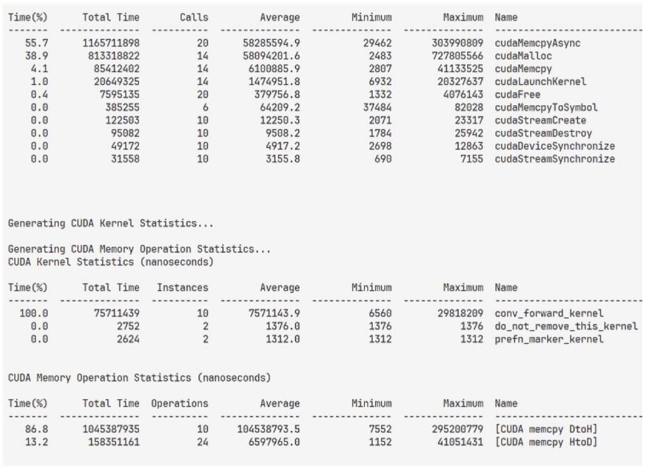
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	/	/	*0m5.764s*	*0.86*
1000	/	/	*52.655s*	*0.886*
5000	/	/	*4m4.630s*	*0.871*

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).



```
Time(%)      Total Time       Calls          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   78.9      1096450617          20      54822530.9            2175       602684205  cudaMemcpy
   13.7       190035472          14      13573962.3            2387       185013469  cudaMalloc
    6.2        85530334          10       8553033.4            3356        68571452  cudaDeviceSynchronize
    1.0        13698159          10       1369815.9           14212        13488322  cudaLaunchKernel
    0.2         3325476          20        166273.8             983          956703  cudaFree
    0.0          426400           6         71066.7           56891           85336  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
  100.0        85389009           6      14231501.5            7104        68563455  conv_forward_kernel
    0.0            2784           2          1392.0            1376            1408  do_not_remove_this_kernel
    0.0            2560           2          1280.0            1216            1344  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   92.8      1013500360           6     168916726.7           13024       601752724  [CUDA memcpy DtoH]
    7.2        79212118          14       5658008.4            1152        41067090  [CUDA memcpy HtoD]
```

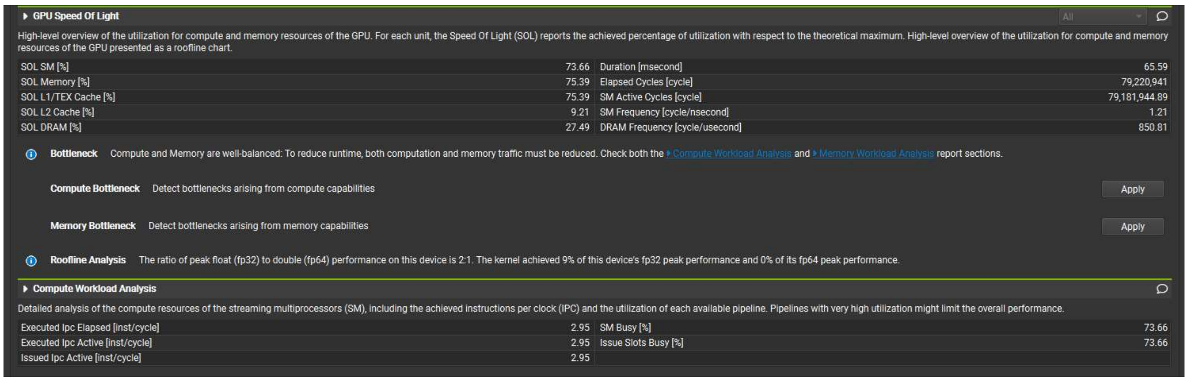


```
Time(%)      Total Time       Calls          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   55.7      1165711898          20      58285594.9           29462       303990809  cudaMemcpyAsync
   38.9       813318822          14      58094201.6            2483       727805566  cudaMalloc
    4.1        85412402          14       6100885.9            2807        41133525  cudaMemcpy
    1.0        20649325          14       1474951.8            6932        20327637  cudaLaunchKernel
    0.4         7595135          20        379756.8            1332         4076143  cudaFree
    0.0          385255           6         64209.2           37484           82028  cudaMemcpyToSymbol
    0.0          122503          10         12250.3            2071           23317  cudaStreamCreate
    0.0           95082          10          9508.2            1784           25942  cudaStreamDestroy
    0.0           49172          10          4917.2            2698           12863  cudaDeviceSynchronize
    0.0           31558          10          3155.8             690            7155  cudaStreamSynchronize



Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)      Total Time   Instances          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
  100.0        75711439          10       7571143.9            6560        29818209  conv_forward_kernel
    0.0            2752           2          1376.0            1376            1376  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1312            1312  prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations          Average          Minimum          Maximum  Name
-------  --------------  ----------  --------------  --------------  --------------  ------------------------
   86.8      1045387935          10     104538793.5            7552       295200779  [CUDA memcpy DtoH]
   13.2       158351161          24       6597965.0            1152        41051431  [CUDA memcpy HtoD]
```

Fig 1

Baseline: Basic Convolution   *Optimization: Stream*

Yes, the implementation did improve the performance. The profiling result using nsys in Fig1 shows that the total time spending on 'conv_forward_kernel' does decrease a lot (from 85389009 to 75711439). Except for the operation time, I compared the profiling result using Nsight-Compute, and their differences have become even more pronounced which is shown in Fig2.



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 [%]	73.66	Duration [msecond]	65.59
SOL Memory [%]	75.39	Elapsed Cycles [cycle]	79,220,941
SOL L1/TEX Cache [%]	75.39	SM Active Cycles [cycle]	79,181,944.89
SOL L2 Cache [%]	9.21	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	27.49	DRAM Frequency [cycle/usecond]	850.81

Bottleneck Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis report sections.

Compute Bottleneck Detect bottlenecks arising from compute capabilities

Memory Bottleneck Detect bottlenecks arising from memory capabilities

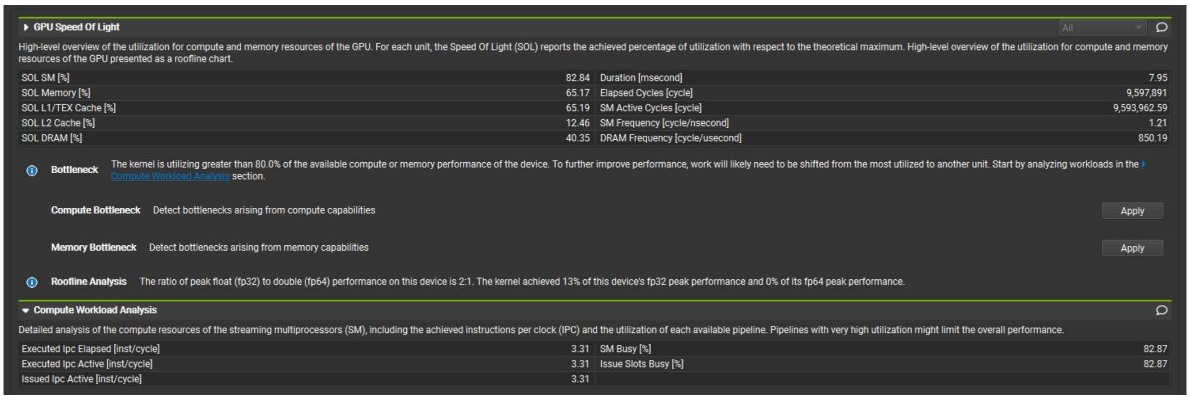
Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 9% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	2.95	SM Busy [%]	73.66
Executed Ipc Active [inst/cycle]	2.95	Issue Slots Busy [%]	73.66
Issued Ipc Active [inst/cycle]	2.95		

Baseline: Basic Convolution



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 [%]	82.84	Duration [msecond]	7.95
SOL Memory [%]	65.17	Elapsed Cycles [cycle]	9,597,891
SOL L1/TEX Cache [%]	65.19	SM Active Cycles [cycle]	9,593,962.59
SOL L2 Cache [%]	12.46	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	40.35	DRAM Frequency [cycle/usecond]	850.19

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Compute Bottleneck Detect bottlenecks arising from compute capabilities

Memory Bottleneck Detect bottlenecks arising from memory capabilities

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 13% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	3.31	SM Busy [%]	82.87
Executed Ipc Active [inst/cycle]	3.31	Issue Slots Busy [%]	82.87
Issued Ipc Active [inst/cycle]	3.31		

Optimization: Stream

Fig 2

We can find that stream has approximately 9% higher streaming multiprocessor efficiency compared to baseline, indicating more effective use of the GPU's compute resources. Except for higher memory efficiency, stream has higher cache hit rates, better DRAM utilization, and increased IPC and Issue Slot utilization, indicating overall more effective memory handling and compute performance than baseline.

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

1. Nvidia official documentation: cuda-c-programming-guide chapter 11
2. Nvidia slice named CUDA C/C++ Streams and Concurrency
3. CSDN
4. Stack overflow

2. Optimization 2: Tiled Shared Memory Convolution and Shared Memory Matrix Multiplication and Input Matrix Unrolling

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

The second optimization I chose is Tiled Shared Memory Convolution and Shared Memory Matrix Multiplication and Input Matrix Unrolling. I think tiled shared memory convolution is a good way to reduce the times for global memory access and therefore decrease the bandwidth. However, it's not that easy to do tiled shared memory convolution in convolution compared with matrix multiplication because the tiled input size is normally larger than the mask size, not to mentioned that if there's stride, the size would be much larger. So the only solution is to unroll the input first then do the tiled shared memory convolution.

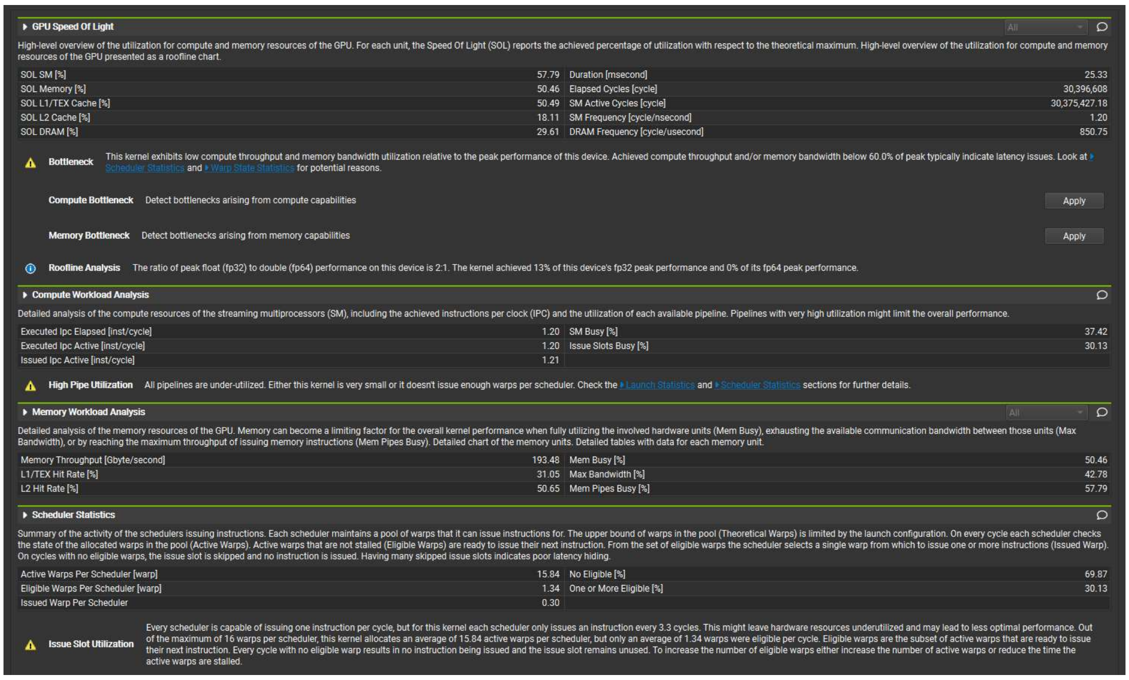
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

Because it can reduce the times for global memory access and decrease the bandwidth. Additionally, it can make use of all the threads better. The optimization would definitely increase the performance of the forward convolution, because each thread will spend less time for data accessing. Logically, unroll synergizes with tiled shared memory, but I have represented them as a joint optimization because I believe they can only work in synergy, not independently.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.29814ms*	*0.42601ms*	*5.803s*	*0.86*
1000	*2.8649 ms*	*3.82974 ms*	*51.930s*	*0.886*
5000	*15.9825*	*22.5951*	*4m22.025s*	*0.871*

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).



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 [%]	57.79	Duration [msecond]	25.33
SOL Memory [%]	50.46	Elapsed Cycles [cycle]	30,396,608
SOL L1/TEX Cache [%]	50.49	SM Active Cycles [cycle]	30,375,427.18
SOL L2 Cache [%]	18.11	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	29.61	DRAM Frequency [cycle/usecond]	850.75

Bottleneck This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

Compute Bottleneck Detect bottlenecks arising from compute capabilities

Memory Bottleneck Detect bottlenecks arising from memory capabilities

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 13% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	1.20	SM Busy [%]	37.42
Executed Ipc Active [inst/cycle]	1.20	Issue Slots Busy [%]	30.13
Issued Ipc Active [inst/cycle]	1.21		

High Pipe Utilization All pipelines are under-utilized. Either this kernel is very small or it doesn't issue enough warps per scheduler. Check the Launch Statistics and Scheduler Statistics sections for further details.

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	193.48	Mem Busy [%]	50.46
L1/TEX Hit Rate [%]	31.05	Max Bandwidth [%]	42.78
L2 Hit Rate [%]	50.65	Mem Pipes Busy [%]	57.79

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	15.84	No Eligible [%]	69.87
Eligible Warps Per Scheduler [warp]	1.34	One or More Eligible [%]	30.13
Issued Warp Per Scheduler	0.30		

Issue Slot Utilization Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 3.3 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 16 warps per scheduler, this kernel allocates an average of 15.84 active warps per scheduler, but only an average of 1.34 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps either increase the number of active warps or reduce the time the active warps are stalled.

Optimization: Unrolling and Tiling

Fig 4

The optimization is not that good as my expected. Comparing the detailed information in Nsight-Computer Fig 4, we can find that the active warps per scheduler is much lower compared with the baseline in Fig 2. The main reason I think is that the if judgement in tiling occurs warp divergence. That's why the total execution time become even longer. But the duration does decrease, which means that the tiling does makes the calculation faster than baseline. And its memory throughput is also higher. That makes sense because the share memory of tiling can reduce the time for global memory access and thus help the kernel function to read the data faster than the baseline.

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

   1. *CSDN*
   2. *Stack overflow*

3. **Optimization 3: Tuning with restrict and loop unrolling.**

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

I choose to use tuning with restrict and loop unrolling because restrict can be used as a keyword to toll the compiler that the no pointer will point to the same place thus allowing the compiler not to worry data overlap or pointer aliasing. What's more, loop unrolling can help increase the efficiency of each thread's work since the mask size is fixed in this project.

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?

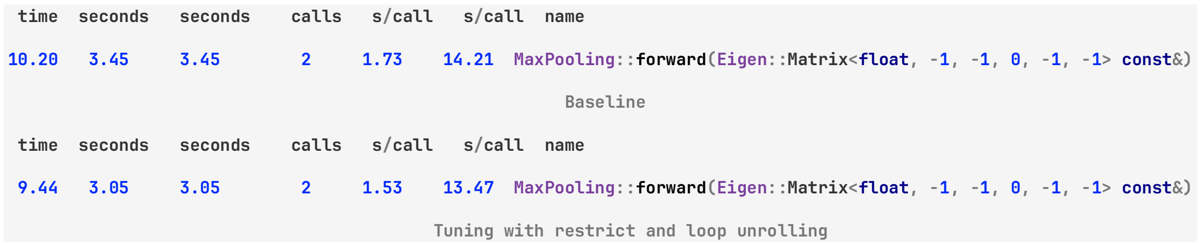
I think it will help the thread working more quickly and increase the performance of the forward convolution. No, it's a separate optimization and if it works, it should increase the performance of the base-line implementation.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.2761ms*	*0.3929ms*	*5.603s*	*0.86*
1000	*1.3397 ms*	*5.6390 ms*	*0m52.411s*	*0.886*
5000	*6.6417 ms*	*27.0089 ms*	*4m11.939s*	*0.871*

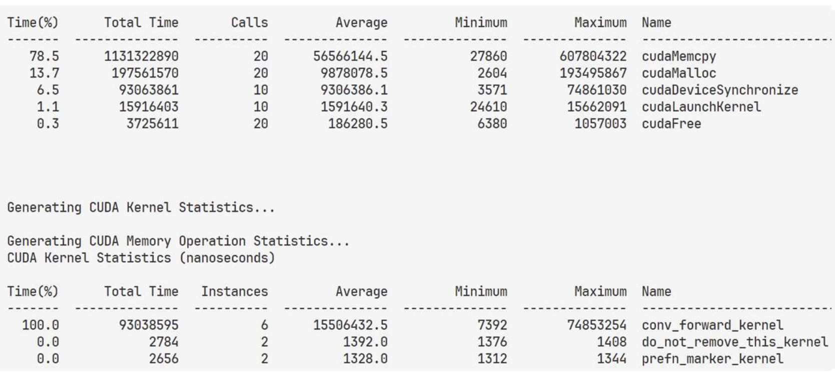
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Yes, it was successful.

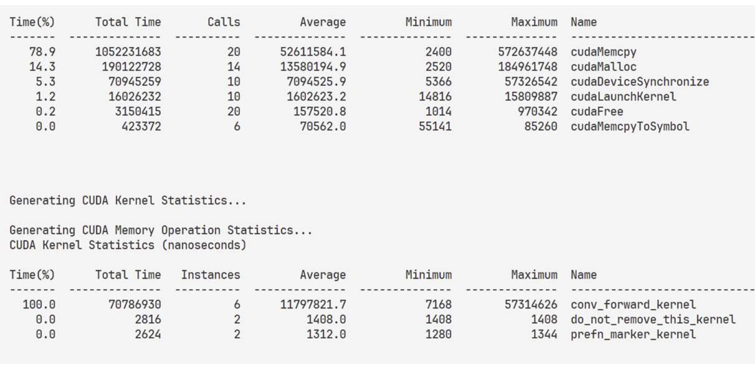


```
 time   seconds    seconds     calls    s/call    s/call   name
10.20    3.45       3.45         2       1.73     14.21   MaxPooling::forward(Eigen::Matrix<float, -1, -1, 0, -1, -1> const&)
                                                Baseline
 time   seconds    seconds     calls    s/call    s/call   name
 9.44    3.05       3.05         2       1.53     13.47   MaxPooling::forward(Eigen::Matrix<float, -1, -1, 0, -1, -1> const&)
                                   Tuning with restrict and loop unrolling
```

Comparing the time spending on forward with nsys, we can see that it takes less time using restrict and unrolling.



```
Time(%)    Total Time      Calls       Average         Minimum         Maximum     Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
   78.5      1131322890          20      56566144.5           27860       607804322  cudaMemcpy
   13.7       197561570          20       9878078.5            2604       193495867  cudaMalloc
    6.5        93063861          10       9306386.1            3571        74861030  cudaDeviceSynchronize
    1.1        15916403          10       1591640.3           24610        15662091  cudaLaunchKernel
    0.3         3725611          20        186280.5            6380         1057003  cudaFree



Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)    Total Time   Instances       Average         Minimum         Maximum     Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
  100.0        93038595           6      15506432.5            7392        74853254  conv_forward_kernel
    0.0            2784           2          1392.0            1376            1408  do_not_remove_this_kernel
    0.0            2656           2          1328.0            1312            1344  prefn_marker_kernel
```

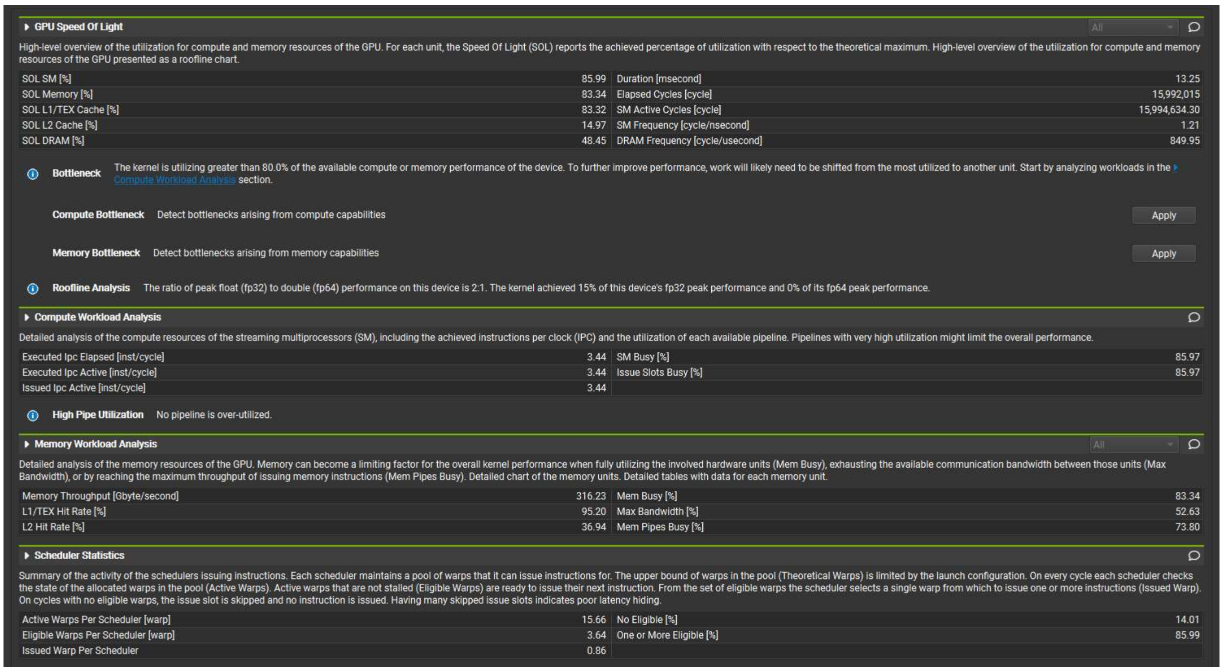


```
Time(%)    Total Time      Calls       Average         Minimum         Maximum     Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
   78.9      1052231683          20      52611584.1            2400       572637448  cudaMemcpy
   14.3       190122728          14      13580194.9            2520       184961748  cudaMalloc
    5.3        70945259          10       7094525.9            5366        57326542  cudaDeviceSynchronize
    1.2        16026232          10       1602623.2           14816        15809887  cudaLaunchKernel
    0.2         3150415          20        157520.8            1014          970342  cudaFree
    0.0          423372           6         70562.0           55141           85260  cudaMemcpyToSymbol



Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)

Time(%)    Total Time   Instances       Average         Minimum         Maximum     Name
-------  --------------  ----------  --------------  --------------  --------------  -------------------------
  100.0        70786930           6      11797821.7            7168        57314626  conv_forward_kernel
    0.0            2816           2          1408.0            1408            1408  do_not_remove_this_kernel
    0.0            2624           2          1312.0            1280            1344  prefn_marker_kernel
```

Baseline: Basic Convolution　　　　*Optimization: Tuning and Restrict*

Fig 4

And in Fig 4 we can see that the total time spending on conv_forward_kernel is decreased tremendously. In Fig 5, we show the detailed information of the optimization using Nsight-Computer. Comparing with the Basic Convolution in Fig 2, we can find that the SOL SM increase from 74% to 86%, and memory throughput increase from 179% to 316%. I think SOL SM shows that the tunning does help streaming multiprocessor to effectively use of the GPU's compute resources, and memory throughput means the rate at which data is being read from or written to memory by the GPU , which demonstrates that the reading data speed is much higher with the restrict keyword.



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 [%]	85.99	Duration [msecond]	13.25
SOL Memory [%]	83.34	Elapsed Cycles [cycle]	15,992,015
SOL L1/TEX Cache [%]	83.32	SM Active Cycles [cycle]	15,994,634.30
SOL L2 Cache [%]	14.97	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	48.45	DRAM Frequency [cycle/usecond]	849.95

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Compute Bottleneck Detect bottlenecks arising from compute capabilities

Memory Bottleneck Detect bottlenecks arising from memory capabilities

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 15% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	3.44	SM Busy [%]	85.97
Executed Ipc Active [inst/cycle]	3.44	Issue Slots Busy [%]	85.97
Issued Ipc Active [inst/cycle]	3.44		

High Pipe Utilization No pipeline is over-utilized.

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

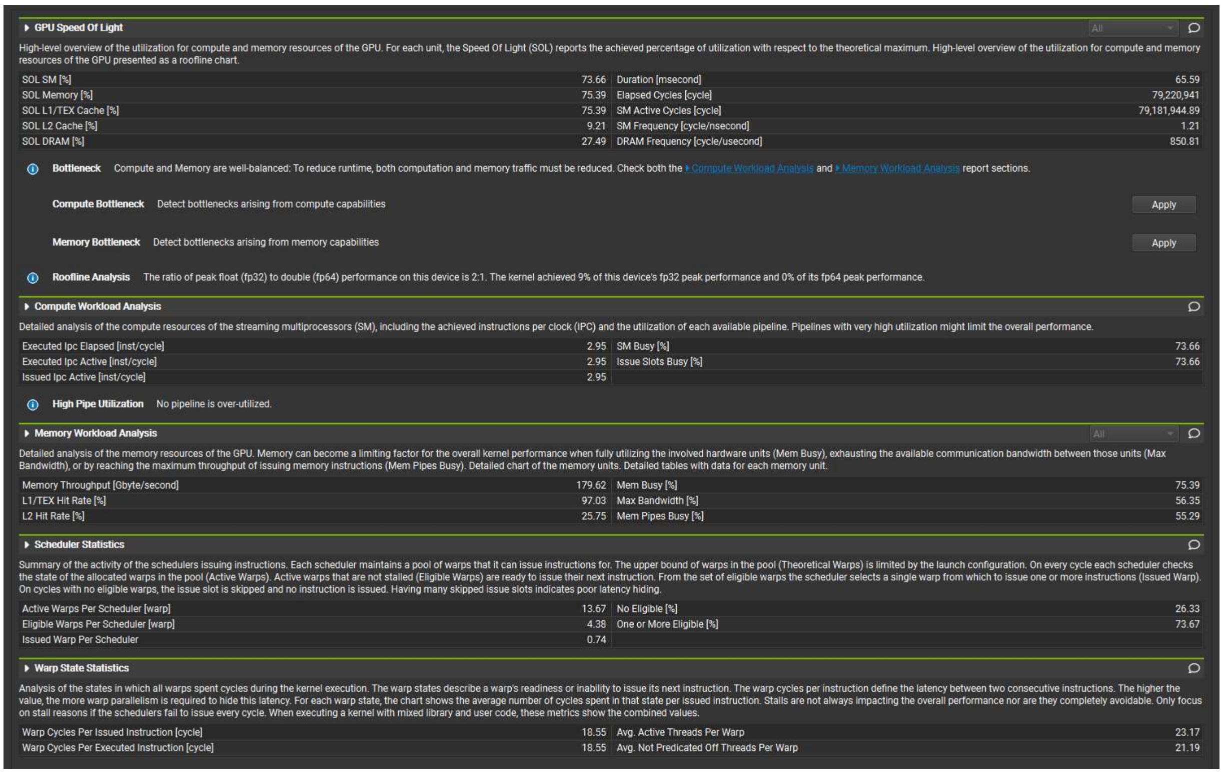
Memory Throughput [Gbyte/second]	316.23	Mem Busy [%]	83.34
L1/TEX Hit Rate [%]	95.20	Max Bandwidth [%]	52.63
L2 Hit Rate [%]	36.94	Mem Pipes Busy [%]	73.80

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	15.66	No Eligible [%]	14.01
Eligible Warps Per Scheduler [warp]	3.64	One or More Eligible [%]	85.99
Issued Warp Per Scheduler	0.86		

Optimization: Tuning and Restrict



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 [%]	73.66	Duration [msecond]	65.59
SOL Memory [%]	75.39	Elapsed Cycles [cycle]	79,220,941
SOL L1/TEX Cache [%]	75.39	SM Active Cycles [cycle]	79,181,944.89
SOL L2 Cache [%]	9.21	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	27.49	DRAM Frequency [cycle/usecond]	850.81

Bottleneck Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis report sections.

Compute Bottleneck Detect bottlenecks arising from compute capabilities

Memory Bottleneck Detect bottlenecks arising from memory capabilities

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 9% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	2.95	SM Busy [%]	73.66
Executed Ipc Active [inst/cycle]	2.95	Issue Slots Busy [%]	73.66
Issued Ipc Active [inst/cycle]	2.95		

High Pipe Utilization No pipeline is over-utilized.

Memory Workload Analysis

Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipes Busy). Detailed chart of the memory units. Detailed tables with data for each memory unit.

Memory Throughput [Gbyte/second]	179.62	Mem Busy [%]	75.39
L1/TEX Hit Rate [%]	97.03	Max Bandwidth [%]	56.35
L2 Hit Rate [%]	25.75	Mem Pipes Busy [%]	55.29

Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	13.67	No Eligible [%]	26.33
Eligible Warps Per Scheduler [warp]	4.38	One or More Eligible [%]	73.67
Issued Warp Per Scheduler	0.74		

Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	18.55	Avg. Active Threads Per Warp	23.17
Warp Cycles Per Executed Instruction [cycle]	18.55	Avg. Not Predicated Off Threads Per Warp	21.19

Baseline: Basic Convolution

Fig 5

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

   1. *CSDN*
   2. *Open Liberty*
   3. *Stack Overflow*

4. Optimization 4: Weight matrix (kernel values) in constant memory

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

   I chose to use weight matrix (kernel values) in constant memory because constant memory access takes about 5 times longer than local memory access which is much less than 20 times for global memory access.

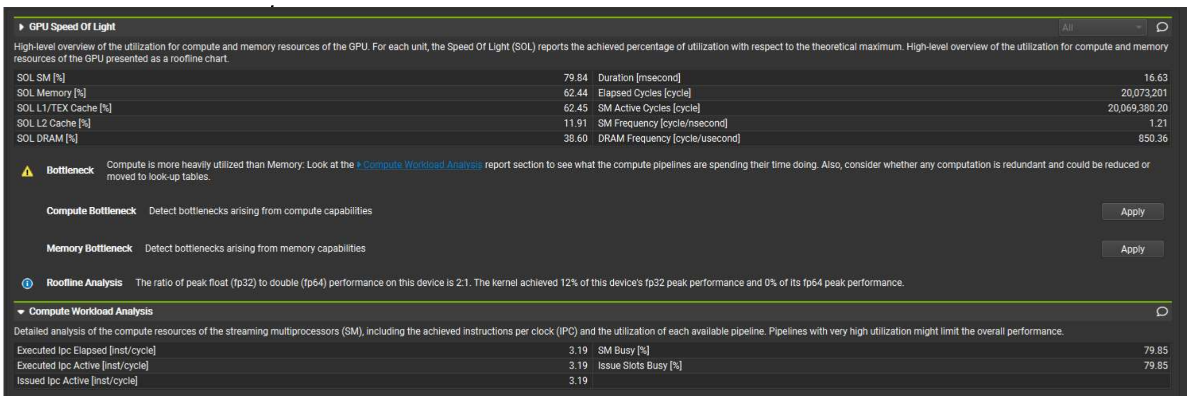
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

   I think it will reduce the operation time because like I said, constant memory access is faster than global memory access, especially for mask data which will be read many times by threads. No, it doesn't synergize with any of your previous optimizations.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.1859ms*	*0.6201ms*	*0m5.848s*	*0.86*
1000	*1.5341ms*	*6.6709ms*	*0m53.190s*	*0.886*
5000	*8.0699ms*	*25.0621ms*	*4m20.179s*	*0.871*

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).



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 [%]	79.84	Duration [msecond]	16.63
SOL Memory [%]	62.44	Elapsed Cycles [cycle]	20,073,201
SOL L1/TEX Cache [%]	62.45	SM Active Cycles [cycle]	20,069,380.20
SOL L2 Cache [%]	11.91	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	38.60	DRAM Frequency [cycle/usecond]	850.36

Bottleneck Compute is more heavily utilized than Memory: Look at the Compute Workload Analysis report section to see what the compute pipelines are spending their time doing. Also, consider whether any computation is redundant and could be reduced or moved to look-up tables.

Compute Bottleneck Detect bottlenecks arising from compute capabilities

Memory Bottleneck Detect bottlenecks arising from memory capabilities

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 12% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	3.19	SM Busy [%]	79.85
Executed Ipc Active [inst/cycle]	3.19	Issue Slots Busy [%]	79.85
Issued Ipc Active [inst/cycle]	3.19		

Optimization: Constant Memory
Fig 6

Compared with the baseline in Fig 2 using Nsight-Compute, the SOL SM, SOL Memory, SOL DRAM increases slightly. That's due to the constant memory. And the SM busy, Executed Ipc Elapsed are the same, which means constant memory doesn't help reduce the computation workload, it only helps each thread reads data faster.

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

1. *Textbook Programming Massively Parallel Processors A Hands-On Approach (David B. Kirk, Wen-Mei W Hwu) (Z-Library).pdf*
2. *MP5*

5. **Optimization 5: Sweeping various parameters to find best values**

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

I chose Sweeping various parameters to find best values because I think good parameters helps the streaming multiprocessor to be more rational in planning of computation resources

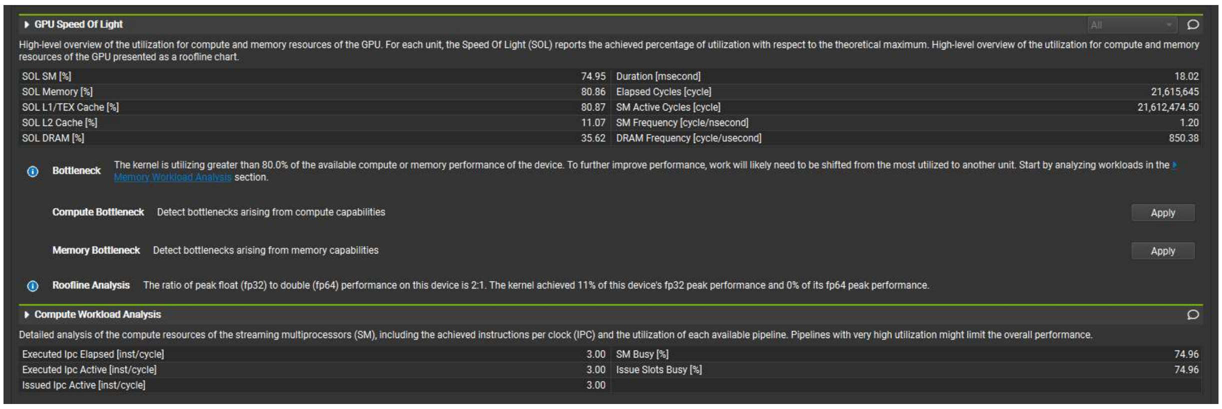
b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

I think it will have some help because if the block size is 32 or times of 32, then the SM can better make use of the computation resources. No, it doesn't synergize with any of your previous optimizations.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.1894ms*	*0.6452ms*	*0m5.990s*	*0.86*
1000	*1.5418ms*	*6.7988ms*	*0m55.891s*	*0.886*
5000	*8.7167ms*	*24.9920ms*	*4m27.653s*	*0.871*

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).


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 [%]	74.95	Duration [msecond]	18.02
SOL Memory [%]	80.86	Elapsed Cycles [cycle]	21,615,645
SOL L1/TEX Cache [%]	80.87	SM Active Cycles [cycle]	21,612,474.50
SOL L2 Cache [%]	11.07	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	35.62	DRAM Frequency [cycle/usecond]	850.38

Bottleneck The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Memory Workload Analysis section.

Compute Bottleneck Detect bottlenecks arising from compute capabilities

Memory Bottleneck Detect bottlenecks arising from memory capabilities

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 11% of this device's fp32 peak performance and 0% of its fp64 peak performance.

Compute Workload Analysis

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	3.00	SM Busy [%]	74.96
Executed Ipc Active [inst/cycle]	3.00	Issue Slots Busy [%]	74.96
Issued Ipc Active [inst/cycle]	3.00		

Optimization: Sweeping Various

Fig 7

The implementation is successful in improving performance. As I expected, setting blockdim to 32 helps the model to increase executed Ipc Elapsed and SM Busy. Because it avoid warp divergence, makes threads to run concurrently.

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

1. *Textbook Programming Massively Parallel Processors A Hands-On Approach (David B. Kirk, Wen-Mei W Hwu) (Z-Library).pdf*
2. *MP2*