Name:
<Arnav Mehta>
NetID:
<arnavm7>
Section:
<AL>

ECE 408/CS483 Milestone 3 Report

- 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	*<0.17832 ms>*	*<0.636396 ms>*	*<1.293 s>*	*<0.86>*
1000	*<1.60053 ms>*	*<6.11565 ms>*	*<10.143 s>*	*<0.886>*
10000	*<15.8478 ms>*	*<61.2404 ms>*	*<1m39.470 s>*	*<0.8714>*

- **Optimization 1:** ***<Using Streams to Overlap Computation with Data Transfer>***
    - Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique.

☐ Tiled shared memory convolution (**2 points**)
☐ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
☐ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
☐ Weight matrix in constant memory (**1 point**)
☐ Tuning with restrict and loop unrolling (**3 points**)

□ Sweeping various parameters to find best values (**1 point**)
□ Multiple kernel implementations for different layer sizes (**1 point**)
□ Input channel reduction: tree (**3 point**)
□ Input channel reduction: atomics (**2 point**)
□ Fixed point (FP16) arithmetic. (**4 points**)
□ Using Streams to overlap computation with data transfer (**4 points**)
I CHOSE TO IMPLEMENT THE STREAMS OPTIMIZATION
□ An advanced matrix multiplication algorithm (**5 points**)
□ Using Tensor Cores to speed up matrix multiplication (**5 points**)
□ Overlap-Add method for FFT-based convolution (**8 points**)
□ Other optimizations: please explain
<*answer here*>

- 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 works by overlapping tasks to reduce the bottleneck of data transfer. Unlike the traditional flow of first sending data to the GPU, having the kernel perform computations, and then sending it back, streams allow for these actions to happen concurrently. A stream is a queue of actions that keeps track of dependencies.
As a result of streams input data can be split up, and some can be sent to the GPU while the GPU's computation on another piece of data gets sent back to the host. In this optimization I had 10 different streams, with each stream processing a batch size of 10 images. In a loop structure I first had the streams send data to the GPU, which are independent activities. These go to the data transfer engine/driver. Then, the streams all launched the kernel via the kernel engine. Then, at the end of an iteration, the streams got the data back from the GPU.

There is an issue where subsequent transfers from host to device are hindered by the queue and the dependency of transfers from the device to the host on the kernel execution. I tried to increase the number of streams so that this would not be a problem at the beginning, which seemed like it could reduce overhead for small batch sizes. I do not think this optimization would synergies particularly well with some other operations, like constant memory for example, due to the large number of kernel calls and thus loading from global memory to constant memory.>

- 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	*<0.001097 ms>*	*<0.001024 ms>*	*<0m1.181s>*	*<0.86>*
1000	*<0.001076 ms>*	*<0.001056 ms>*	*<0m9.747 s>*	*<0.886>*
10000	*<0.001166 ms>*	*<0.001587 ms>*	*<1m37.128 s>*	*<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 implementation was partially successful in improving performance. Total execution times were less than the baseline (for example in 1000 batch case the streams had a total time of 9.747 seconds which was less than 10.143 seconds in the baseline case. The op times here do not accurately measure the speed because memcpy is happening so frequently. The profiling is as follows:

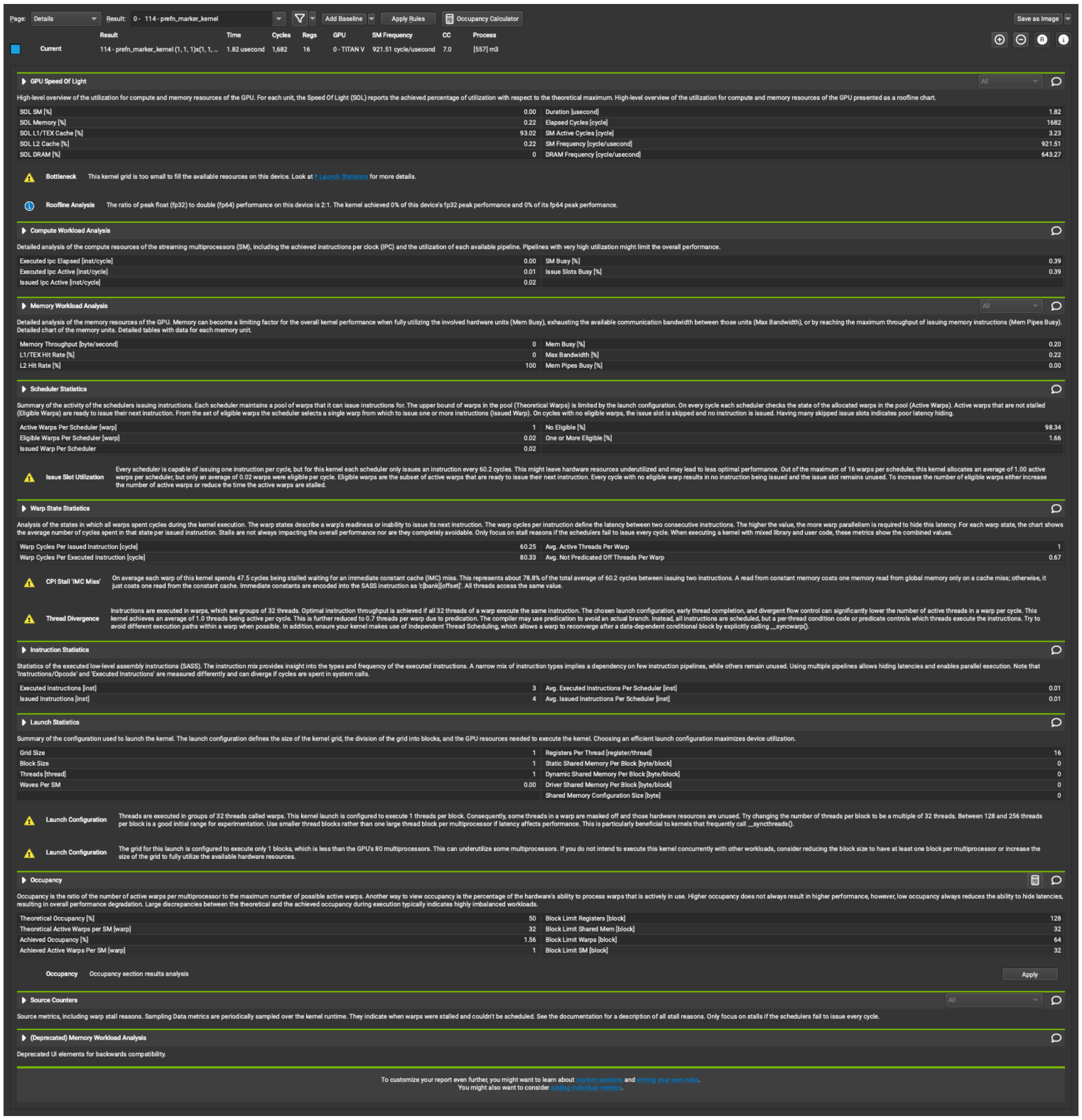
```
Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)     Total Time      Calls       Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
--------------------------------------------------------------------------------
   85.3     1254657823       4000       313664.5          29993        1192353
cudaMemcpyAsync
   12.6      185983604          8     23247950.5          73613      182909037
cudaMalloc
    1.6       23948217       2004        11950.2           3123       12175404
cudaLaunchKernel
    0.2        3209323          4       802330.7          10016        3160489
cudaMemcpy
    0.2        2891419          8       361427.4          68036         986158
cudaFree
```

```
   0.0         292814          4        73203.5          21035         123284
cudaHostRegister
   0.0         145866         20         7293.3           1394          86917
cudaStreamCreate
   0.0         121925         20         6096.2           1831          38486
cudaStreamDestroy
   0.0          39828          8        4978.5           1948           8350
cudaDeviceSynchronize




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
---------------------------------------------------------------------------------
  100.0       134937686        2000         67468.8          22912          124767
conv_forward_kernel
    0.0            2816           2          1408.0           1376            1440
do_not_remove_this_kernel
    0.0            2720           2          1360.0           1312            1408
prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
---------------------------------------------------------------------------------
   73.2       136441620        2000         68220.8          57056           91104  [CUDA memcpy
DtoH]
   26.8        49922461        2004         24911.4           1536           27967  [CUDA memcpy
HtoD]


CUDA Memory Operation Statistics (KiB)

          Total      Operations           Average          Minimum          Maximum
Name
-----------------  --------------  -----------------  -----------------  -----------------
---------------------------------------------------------------------------------
```

```
      1722500.0          2000           861.0          722.500          1000.0  [CUDA memcpy
DtoH]
       538919.0          2004           268.9            0.004           288.0  [CUDA memcpy
HtoD]




Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time      Calls       Average        Minimum        Maximum
Name
-------  --------------  ----------  -------------  -------------  --------------
--------------------------------------------------------------------------------
   33.3     96197588197        976     98563102.7          23513       100413091
sem_timedwait
   33.3     96176948240        975     98643023.8          35416       100553552
poll
   22.2     64147271031          2  32073635515.5     23453954467     40693316564
pthread_cond_wait
   11.1     32008316231         64    500129941.1       500049524       500182970
pthread_cond_timedwait
    0.0        92081956        923        99763.8           1036        18818041
ioctl
    0.0        22738425       9072         2506.4           1270           37548
read
    0.0        17088224         26       657239.4           1532        17015224
fopen
    0.0         3636000         98        37102.0           1178         1591611
mmap
    0.0         1137756        101        11264.9           4138           26753
open64
    0.0          466864          1       466864.0         466864          466864
pthread_mutex_lock
    0.0          308047          5        61609.4          40480           76885
pthread_create
    0.0          134849          3        44949.7          41962           48999
fgets
    0.0           84047         19         4423.5           1374           14813
munmap
    0.0           83913          3        27971.0           3896           55330
fopen64
    0.0           64916         15         4327.7           2182            7925
write
    0.0           56669          8         7083.6           2311            9958
fflush
```

```
    0.0       30387       5      6077.4       2845       7864
open
    0.0       26328      10      2632.8       1014      10360
fclose
    0.0       14297       2      7148.5       4631       9666
pthread_cond_signal
    0.0       13538       2      6769.0       5924       7614
socket
    0.0       12296       8      1537.0       1216       1865
fcntl
    0.0        7903       1      7903.0       7903       7903
pipe2
    0.0        7613       1      7613.0       7613       7613
connect
    0.0        6270       3      2090.0       1057       3556
fwrite
    0.0        2224       1      2224.0       2224       2224
bind
```

>



Page: Details Result: 0 - 114 - prefn_marker_kernel

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	114 - prefn_marker_kernel (1, 1, 1)x(1, 1, …	1.82 usecond	1,682	16	0 - TITAN V	921.51 cycle/usecond	7.0	[557] 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 [%]	0.00	Duration [usecond]	1.82
SOL Memory [%]	0.22	Elapsed Cycles [cycle]	1682
SOL L1/TEX Cache [%]	93.02	SM Active Cycles [cycle]	3.23
SOL L2 Cache [%]	0.22	SM Frequency [cycle/usecond]	921.51
SOL DRAM [%]	0	DRAM Frequency [cycle/usecond]	643.27

Bottleneck This kernel grid is too small to fill the available resources on this device. Look at Launch Statistics for more details.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% 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]	0.00	SM Busy [%]	0.39
Executed Ipc Active [inst/cycle]	0.01	Issue Slots Busy [%]	0.39
Issued Ipc Active [inst/cycle]	0.02		

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 [byte/second]	0	Mem Busy [%]	0.20
L1/TEX Hit Rate [%]	0	Max Bandwidth [%]	0.22
L2 Hit Rate [%]	100	Mem Pipes Busy [%]	0.00

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]	1	No Eligible [%]	98.34
Eligible Warps Per Scheduler [warp]	0.02	One or More Eligible [%]	1.66
Issued Warp Per Scheduler	0.02		

Issue Slot Utilization Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 60.2 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 1.00 active warps per scheduler, but only an average of 0.02 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.

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]	60.25	Avg. Active Threads Per Warp	1
Warp Cycles Per Executed Instruction [cycle]	80.33	Avg. Not Predicated Off Threads Per Warp	0.67

CPI Stall 'IMC Miss' On average each warp of this kernel spends 47.5 cycles being stalled waiting for an immediate constant cache (IMC) miss. This represents about 78.8% of the total average of 60.2 cycles between issuing two instructions. A read from constant memory costs one memory read from global memory only on a cache miss; otherwise, it just costs one read from the constant cache. Immediate constants are encoded into the SASS instruction as 'c[bank][offset]'. All threads access the same value.

Thread Divergence Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 1.0 threads being active per cycle. This is further reduced to 0.7 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().

Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	3	Avg. Executed Instructions Per Scheduler [inst]	0.01
Issued Instructions [inst]	4	Avg. Issued Instructions Per Scheduler [inst]	0.01

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	1	Registers Per Thread [register/thread]	16
Block Size	1	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	1	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	0.00	Driver Shared Memory Per Block [byte/block]	0
		Shared Memory Configuration Size [byte]	0

Launch Configuration Threads are executed in groups of 32 threads called warps. This kernel launch is configured to execute 1 threads per block. Consequently, some threads in a warp are masked off and those hardware resources are unused. Try changing the number of threads per block to be a multiple of 32 threads. Between 128 and 256 threads per block is a good initial range for experimentation. Use smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call __syncthreads().

Launch Configuration The grid for this launch is configured to execute only 1 blocks, which is less than the GPU's 80 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel concurrently with other workloads, consider reducing the block size to have at least one block per multiprocessor or increase the size of the grid to fully utilize the available hardware resources.

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	128
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	1.56	Block Limit Warps [block]	64
Achieved Active Warps Per SM [warp]	1	Block Limit SM [block]	32

Occupancy Occupancy section results analysis

Source Counters

Source metrics, including warp stall reasons. Sampling Data metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

(Deprecated) Memory Workload Analysis

Deprecated UI elements for backwards compatibility.

To customize your report even further, you might want to learn about custom sections and writing your own rules.
You might also want to consider adding individual metrics.

- What references did you use when implementing this technique? <I used course lecture 22 and the CUDA API programming guide for syntax for the streams.>

- Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization. For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the "for" loop for loop unrolling

```
<cudaStream_t cudaArr[10];
   for(int i = 0; i < 10; i++){
       cudaStreamCreate(&cudaArr[i]);
   }

   int inter_batch = Channel * Height * Width;
   int inter_out = Map_out * (Height - K + 1) * (Width
- K + 1);
   float * temp = (float*) *device_output_ptr;
   float * temp2 = (float*) host_output;

   cudaMemcpy(*device_mask_ptr, host_mask,
sizeof(float) * Channel * Map_out * K * K,
cudaMemcpyHostToDevice);


   cudaHostRegister(&temp2, Batch * inter_out,
cudaHostRegisterDefault);
   cudaHostRegister(&host_input, Batch * inter_batch,
cudaHostRegisterDefault);
  for(int i = 0; i < Batch; i+= seg_size * 10){

   for(int s = 0; s < 10; s++){
       cudaMemcpyAsync(*device_input_ptr + (i + s *
seg_size) * inter_batch, host_input + (i + s *
seg_size) * inter_batch, seg_size * sizeof(float) *
inter_batch, cudaMemcpyHostToDevice, cudaArr[s]);
   }


   for(int s = 0; s < 10; s++){
       conv_forward_kernel<<<dimGrid, dimBlock, 0,
cudaArr[s]>>>(*device_output_ptr + (i + s * seg_size) *
inter_out,*device_input_ptr + (i + s * seg_size) *
inter_batch, *device_mask_ptr, seg_size, Map_out,
Channel, Height, Width, K);
   }
```

```
    for(int s = 0; s < 10; s++){
        cudaMemcpyAsync(temp2 + (i + s * seg_size) *
inter_out, temp + (i + s * seg_size) * inter_out,
seg_size * inter_out * sizeof(float),
cudaMemcpyDeviceToHost, cudaArr[s]);
    }


   }
```

>

- **Optimization 2: <Input channel reduction: atomics>**
  - Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique.

☐ Tiled shared memory convolution (**2 points**)
☐ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
☐ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
☐ Weight matrix in constant memory (**1 point**)
☐ Tuning with restrict and loop unrolling (**3 points**)
☐ Sweeping various parameters to find best values (**1 point**)
☐ Multiple kernel implementations for different layer sizes (**1 point**)
☐ Input channel reduction: tree (**3 point**)
☐ Input channel reduction: atomics (**2 point**)
INPUT CHANNEL REDUCTION: ATOMICS
☐ Fixed point (FP16) arithmetic. (**4 points**)
☐ Using Streams to overlap computation with data transfer (**4 points**)
☐ An advanced matrix multiplication algorithm (**5 points**)
☐ Using Tensor Cores to speed up matrix multiplication (**5 points**)
☐ Overlap-Add method for FFT-based convolution (**8 points**)
☐ Other optimizations: please explain
<answer here>

- 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 works by getting rid of the need for a for loop by allowing multiple threads to write to the same global memory location at around the same times using atomic operations. Atomic operations ensure that if any thread at a moment writes a value to the global memory, that value will not accidentally be overwritten
by another thread. As a result input channels can do work in parallel, which should increase the speed of the convolution by getting rid of the sequential for loop. This modification would synergies well with something like constant memory, as it remains relatively independent of the constant memory optimization.>

- 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	*<0.376371 ms>*	*<0.941186 ms>*	*<0m1.279s>*	*<0.86>*
1000	*<1.65392 ms>*	*<7.57432 ms>*	*<0m10.815s>*	*<0.886>*
10000	*<15.7643 ms>*	*<75.4195 ms>*	*<1m47.923s>*	*<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

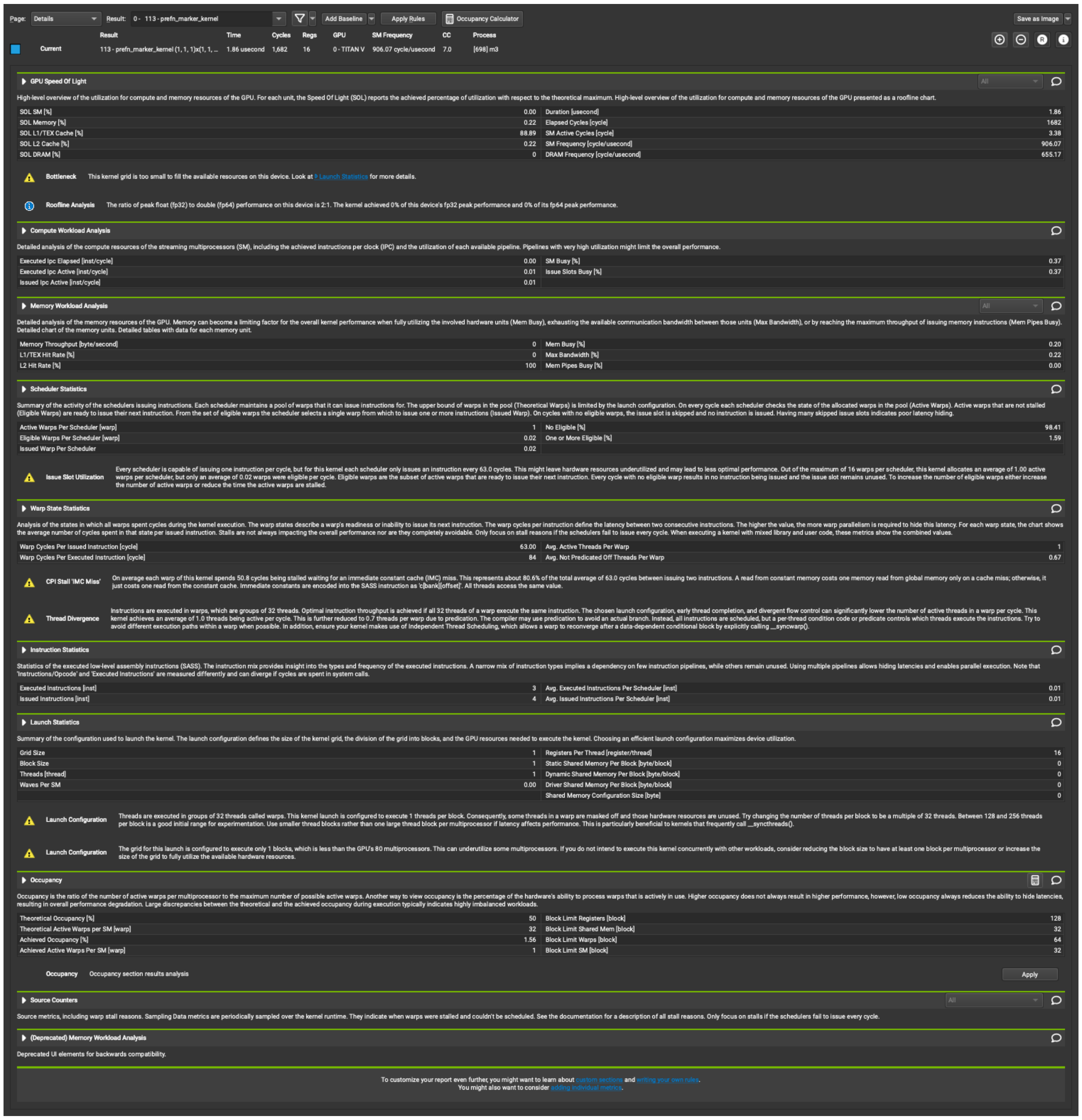
<The optimization was not successful in improving performance, as both the total

execution times and op times went up. There is likely some overhead in increasing the size of each block (since we added a third dimension for input channel parallelism). Also, there were a small number of input channels anyway, so the optimization was not
very noticeable. >

```
0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************

Exported successfully to
/build/report1.sqlite

Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)

Time(%)      Total Time      Calls        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
------------------------------------------------------------------------------
   77.1      1098639241          8     137329905.1           8135      603996490
cudaMemcpy
   15.3       217419189          8      27177398.6          70188      214037695
cudaMalloc
    6.5        91957447          6      15326241.2           3002       76048638
cudaDeviceSynchronize
    1.0        14245586          6       2374264.3          22817       14103181
cudaLaunchKernel
    0.2         2792429          8        349053.6          69434         974655
cudaFree




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
------------------------------------------------------------------------------
  100.0        91718101          2      45859050.5       15728854       75989247
conv_forward_kernel
    0.0            2848          2          1424.0           1376           1472
do_not_remove_this_kernel
    0.0            2720          2          1360.0           1344           1376
```

```
prefn_marker_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
--------------------------------------------------------------------------------
   93.2      1019488638           2     509744319.0      416403012      603085626  [CUDA memcpy
DtoH]
    6.8        73805870           6      12300978.3           1504       39059544  [CUDA memcpy
HtoD]


CUDA Memory Operation Statistics (KiB)

          Total      Operations          Average          Minimum          Maximum
Name
-----------------  --------------  -----------------  -----------------  -----------------
--------------------------------------------------------------------------------
       1722500.0             2          861250.0        722500.000         1000000.0  [CUDA memcpy
DtoH]
        538919.0             6           89819.0             0.004          288906.0  [CUDA memcpy
HtoD]




Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time       Calls        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
--------------------------------------------------------------------------------
   33.3     94409383418         958      98548416.9          44928      100314577
sem_timedwait
   33.3     94381346070         957      98622096.2          76138      100395926
poll
   22.0     62333209484           2   31166604742.0    22724892940    39608316544
pthread_cond_wait
   11.3     32008534773          64     500133355.8      500044444      500174930
pthread_cond_timedwait
    0.0       120450582         911        132218.0           1059       20093878
ioctl
    0.0        22826252        9072          2516.1           1220          18704
```

```
read
   0.0      18408261        26       708010.0          1128       18334532
fopen
   0.0       3913684        98        39935.6          1155        1650623
mmap
   0.0       1515003       101        15000.0          4472          27967
open64
   0.0        326197         5       65239.4         38423          93773
pthread_create
   0.0        141657         3       47219.0         42867          53421
fgets
   0.0        102195        15        6813.0          2415          14183
write
   0.0         98205         2       49102.5         42746          55459
pthread_mutex_lock
   0.0         80713        17        4747.8          1145          13709
munmap
   0.0         74374         3       24791.3          3043          46997
fopen64
   0.0         58180        39        1491.8          1073           1700
fcntl
   0.0         50180         8        6272.5          1165          10100
fflush
   0.0         30593         5        6118.6          2900           7671
open
   0.0         24572        10        2457.2          1058           9256
fclose
   0.0         14247         2        7123.5          4301           9946
pthread_cond_signal
   0.0         13459         2        6729.5          5736           7723
socket
   0.0          8108         1        8108.0          8108           8108
pipe2
   0.0          7846         1        7846.0          7846           7846
connect
   0.0          3638         2        1819.0          1243           2395
fwrite
   0.0          2023         1        2023.0          2023           2023
bind
   0.0          1032         1        1032.0          1032           1032  listen
```



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	113 - prefn_marker_kernel (1, 1, 1)x(1, 1, ...	1.86 usecond	1,682	16	0 - TITAN V	906.07 cycle/usecond	7.0	[698] 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 [%]	0.00	Duration [usecond]	1.86
SOL Memory [%]	0.22	Elapsed Cycles [cycle]	1682
SOL L1/TEX Cache [%]	88.89	SM Active Cycles [cycle]	3.38
SOL L2 Cache [%]	0.22	SM Frequency [cycle/usecond]	906.07
SOL DRAM [%]	0	DRAM Frequency [cycle/usecond]	655.17

Bottleneck This kernel grid is too small to fill the available resources on this device. Look at Launch Statistics for more details.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% 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]	0.00	SM Busy [%]	0.37
Executed Ipc Active [inst/cycle]	0.01	Issue Slots Busy [%]	0.37
Issued Ipc Active [inst/cycle]	0.01		

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 [byte/second]	0	Mem Busy [%]	0.20
L1/TEX Hit Rate [%]	0	Max Bandwidth [%]	0.22
L2 Hit Rate [%]	100	Mem Pipes Busy [%]	0.00

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]	1	No Eligible [%]	98.41
Eligible Warps Per Scheduler [warp]	0.02	One or More Eligible [%]	1.59
Issued Warp Per Scheduler	0.02		

Issue Slot Utilization Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 63.0 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 1.00 active warps per scheduler, but only an average of 0.02 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.

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]	63.00	Avg. Active Threads Per Warp	1
Warp Cycles Per Executed Instruction [cycle]	84	Avg. Not Predicated Off Threads Per Warp	0.67

CPI Stall 'IMC Miss' On average each warp of this kernel spends 50.8 cycles being stalled waiting for an immediate constant cache (IMC) miss. This represents about 80.6% of the total average of 63.0 cycles between issuing two instructions. A read from constant memory costs one memory read from global memory only on a cache miss; otherwise, it just costs one read from the constant cache. Immediate constants are encoded into the SASS instruction as 'c[bank][offset]'. All threads access the same value.

Thread Divergence Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 1.0 threads being active per cycle. This is further reduced to 0.7 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().

Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	3	Avg. Executed Instructions Per Scheduler [inst]	0.01
Issued Instructions [inst]	4	Avg. Issued Instructions Per Scheduler [inst]	0.01

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	1	Registers Per Thread [register/thread]	16
Block Size	1	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	1	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	0.00	Driver Shared Memory Per Block [byte/block]	0
		Shared Memory Configuration Size [byte]	0

Launch Configuration Threads are executed in groups of 32 threads called warps. This kernel launch is configured to execute 1 threads per block. Consequently, some threads in a warp are masked off and those hardware resources are unused. Try changing the number of threads per block to be a multiple of 32 threads. Between 128 and 256 threads per block is a good initial range for experimentation. Use smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call __syncthreads().

Launch Configuration The grid for this launch is configured to execute only 1 blocks, which is less than the GPU's 80 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel concurrently with other workloads, consider reducing the block size to have at least one block per multiprocessor or increase the size of the grid to fully utilize the available hardware resources.

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	128
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	1.56	Block Limit Warps [block]	64
Achieved Active Warps Per SM [warp]	1	Block Limit SM [block]	32

Occupancy Occupancy section results analysis

Source Counters

Source metrics, including warp stall reasons. Sampling Data metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

(Deprecated) Memory Workload Analysis

Deprecated UI elements for backwards compatibility.

To customize your report even further, you might want to learn about custom sections and writing your own rules.
You might also want to consider adding individual metrics.

- What references did you use when implementing this technique?

<I used lecture 18, which was histogramming and atomic operations.>

- Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.

For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the "for" loop for loop unrolling
<answer here>

```
   // Insert your GPU convolution kernel code here
    int W_grid = ceil((float)Width_out / TILE_WIDTH);
    int m = blockIdx.x;
    int h = (blockIdx.y / W_grid) * TILE_WIDTH + 
threadIdx.y;
    int w = (blockIdx.y % W_grid) * TILE_WIDTH + 
threadIdx.x;
    int z = blockIdx.z;
    float acc = 0.0f;

    int c = threadIdx.z;

     if(h < Height_out && w < Width_out){
    //for(int c = 0; c < Channel; c++){
        for(int p = 0; p < K; p++){
            for(int q = 0; q < K; q++){
                acc += in_4d(z, c, h + p, w + q) * 
mask_4d(m, c, p, q);

            }
        }
    //}

        atomicAdd(&out_4d(z, m, h, w), acc);
    }
```

In the host code a modification was made to add a z dimension to each block, to represent the input channel.

- **Optimization 3: <*Weight Matrix in Constant Memory*>**
  - Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose that optimization technique.

☐Tiled shared memory convolution (**2 points**)
☐ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
☐ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
☐ Weight matrix in constant memory (**1 point**)
WEIGHT MATRIX IN CONSTANT MEMORY
☐ Tuning with restrict and loop unrolling (**3 points**)
☐ Sweeping various parameters to find best values (**1 point**)
☐ Multiple kernel implementations for different layer sizes (**1 point**)
☐ Input channel reduction: tree (**3 point**)
☐ Input channel reduction: atomics (**2 point**)
☐ Fixed point (FP16) arithmetic. (**4 points**)
☐ Using Streams to overlap computation with data transfer (**4 points**)
☐ An advanced matrix multiplication algorithm (**5 points**)
☐ Using Tensor Cores to speed up matrix multiplication (**5 points**)
☐ Overlap-Add method for FFT-based convolution (**8 points**)
☐ Other optimizations: please explain
<*answer here*>

  - 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?

<Our optimization works by reducing the number of access to global memory by loading the convolution kernels as constant memory, which takes fewer clock cycles to access. I think the optimization will increase performance due to reuse of some elements of every single convolutional filter. I think this becomes Significant for large convolution filter sizes. This optimization would synergize nicely with the atomic operations operations optimization due to its reliance on accessing global memory.>

- 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	*<0.3387 ms>*	*<0.783927 ms>*	*<0m1.198s>*	*<0.86>*
1000	*<1.52063 ms>*	*<6.48102 ms>*	*<0m10.422 s>*	*<0.886>*
10000	*<14.5867 ms>*	*<64.2765 ms>*	*<1m41.936s>*	*<0.8714>*

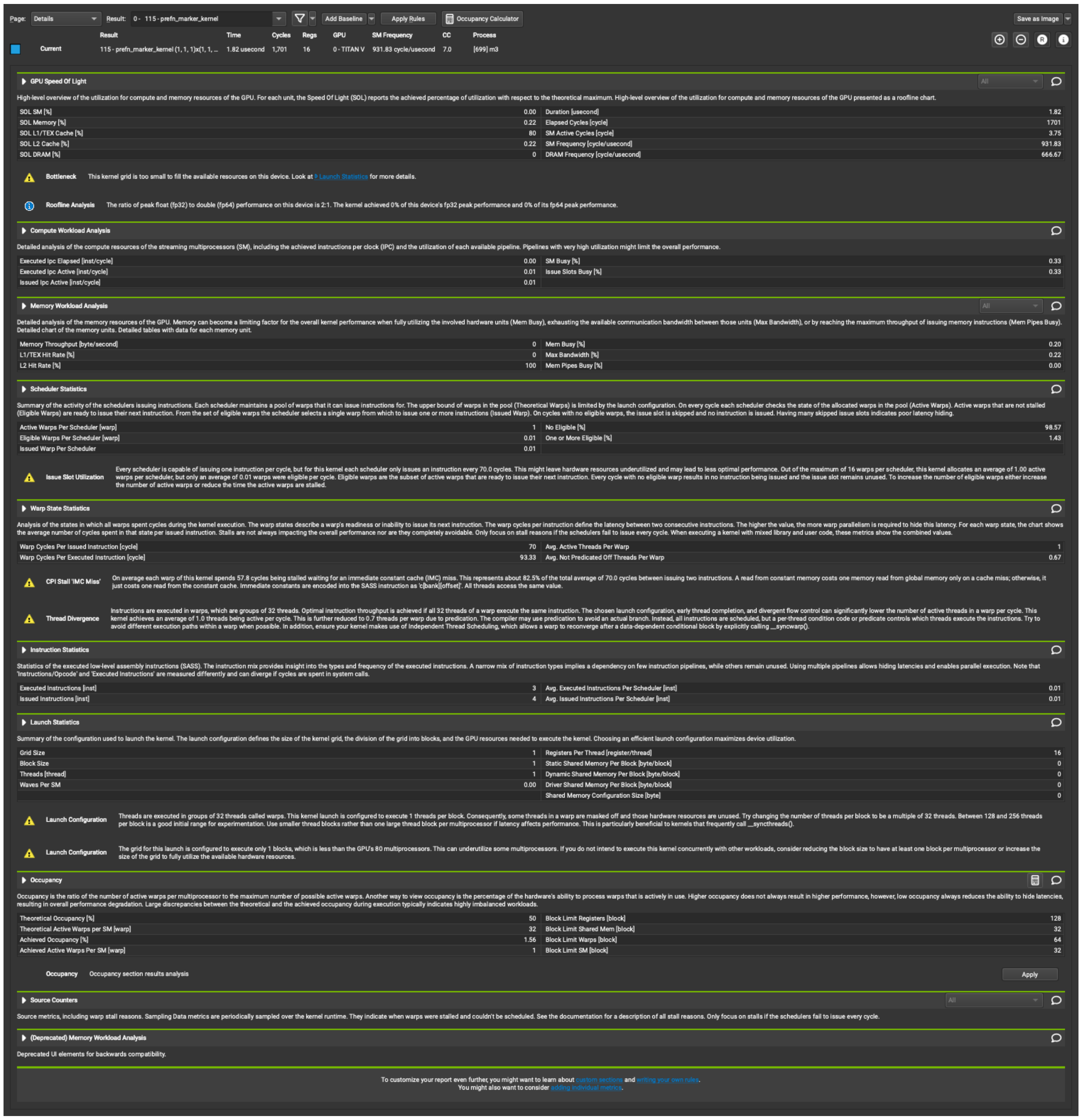
- Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of

<This kernel performed slightly worse than the baseline in op time and total execution time. I suspect the kernel size was not big enough for the advantages of reuse to come into play, and instead I suspect there was a greater overhead from loading the entire convolution kernel from global memory to constant memory.>

```
                Time(%)    Total Time    Calls     Average      Minimum      Maximum
Name
-------  --------------  ----------  -------------  -------------  --------------
---------------------------------------------------------------------------
   75.8      1064133872          6     177355645.3          18050       575181643
cudaMemcpy
   17.2       241291910          6      40215318.3         283589       237961736
cudaMalloc
    5.5        77550874          6      12925145.7           2611        62968824
```

```
cudaDeviceSynchronize
   1.2       16964279         6      2827379.8         22897       16794763
cudaLaunchKernel
   0.2        3385717         8       423214.6          1739        1505472
cudaFree
   0.0          37472         2        18736.0         10065          27407
cudaMemcpyToSymbol




Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
---------------------------------------------------------------------------
  100.0       77475471          2     38737735.5       14521035       62954436
conv_forward_kernel
   0.0           2656          2        1328.0           1280           1376
prefn_marker_kernel
   0.0           2624          2        1312.0           1248           1376
do_not_remove_this_kernel


CUDA Memory Operation Statistics (nanoseconds)

Time(%)      Total Time  Operations        Average        Minimum        Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
---------------------------------------------------------------------------
   92.7      980928707          2    490464353.5      406643722      574284985  [CUDA memcpy
DtoH]
    7.3       77535661          6     12922610.2           1472       41242162  [CUDA memcpy
HtoD]


CUDA Memory Operation Statistics (KiB)

         Total      Operations          Average          Minimum          Maximum
Name
-----------------  --------------  -----------------  -----------------  -----------------
---------------------------------------------------------------------------
      1722500.0             2         861250.0        722500.000        1000000.0  [CUDA memcpy
DtoH]
       538919.0             6          89819.0           0.004         288906.0  [CUDA memcpy
HtoD]



Generating Operating System Runtime API Statistics...
Operating System Runtime API Statistics (nanoseconds)

Time(%)      Total Time      Calls       Average       Minimum       Maximum
Name
-------  --------------  ----------  --------------  --------------  --------------
-------------------------------------------------------------------------------
   33.3     95465670242        968     98621560.2         52360      100356230
sem_timedwait
   33.3     95378665141        967     98633573.1         65796      100365652
poll
   22.1     63418425309          2   31709212654.5    22962143204    40456282105
pthread_cond_wait
   11.2     32006595905         64     500103061.0      500071669      500159032
pthread_cond_timedwait
    0.0       138726493        905       153288.9          1038       18824794
ioctl
    0.0        22186182       9072         2445.6          1223          19315
read
    0.0        19719675         26       758449.0          1291       19641771
fopen
    0.0         3776971         98        38540.5          1059        1573992
mmap
    0.0         1410426        101        13964.6          4201          46988
open64
    0.0          353836          2       176918.0         82371         271465
pthread_mutex_lock
    0.0          306135          5        61227.0         44522          77096
pthread_create
    0.0          158506          3        52835.3         48859          58949
fgets
    0.0           91279         17         5369.4          1152          12959
munmap
    0.0           77109         15         5140.6          2540          12087
write
    0.0           74376          3        24792.0          3355          49946
fopen64
    0.0           52792          7         7541.7          3990           9904
fflush
    0.0           31643          5         6328.6          3149           8684
```

```
open
   0.0        28221        12        2351.8          1042          8345
fclose
   0.0        16090         2        8045.0          6678          9412
socket
   0.0        11214         8        1401.8          1025          2086
fcntl
   0.0         8210         1        8210.0          8210          8210
pipe2
   0.0         7982         2        3991.0          3127          4855
pthread_cond_signal
   0.0         7460         1        7460.0          7460          7460
connect
   0.0         6667         4        1666.8          1001          2582
fwrite
   0.0         2060         1        2060.0          2060          2060
bind
   0.0         1006         1        1006.0          1006          1006
putc
```


Page: Details Result: 0 - 115 - prefn_marker_kernel Add Baseline Apply Rules Occupancy Calculator Save as Image

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	115 - prefn_marker_kernel (1, 1, 1)x(1, 1, ...	1.82 usecond	1,701	16	0 - TITAN V	931.83 cycle/usecond	7.0	[699] 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 [%]	0.00	Duration [usecond]	1.82
SOL Memory [%]	0.22	Elapsed Cycles [cycle]	1701
SOL L1/TEX Cache [%]	80	SM Active Cycles [cycle]	3.75
SOL L2 Cache [%]	0.22	SM Frequency [cycle/usecond]	931.83
SOL DRAM [%]	0	DRAM Frequency [cycle/usecond]	666.67

Bottleneck This kernel grid is too small to fill the available resources on this device. Look at Launch Statistics for more details.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 0% 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]	0.00	SM Busy [%]	0.33
Executed Ipc Active [inst/cycle]	0.01	Issue Slots Busy [%]	0.33
Issued Ipc Active [inst/cycle]	0.01		

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 [byte/second]	0	Mem Busy [%]	0.20
L1/TEX Hit Rate [%]	0	Max Bandwidth [%]	0.22
L2 Hit Rate [%]	100	Mem Pipes Busy [%]	0.00

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]	1	No Eligible [%]	98.57
Eligible Warps Per Scheduler [warp]	0.01	One or More Eligible [%]	1.43
Issued Warp Per Scheduler	0.01		

Issue Slot Utilization Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 70.0 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 1.00 active warps per scheduler, but only an average of 0.01 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.

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]	70	Avg. Active Threads Per Warp	1
Warp Cycles Per Executed Instruction [cycle]	93.33	Avg. Not Predicated Off Threads Per Warp	0.67

CPI Stall 'IMC Miss' On average each warp of this kernel spends 57.8 cycles being stalled waiting for an immediate constant cache (IMC) miss. This represents about 82.5% of the total average of 70.0 cycles between issuing two instructions. A read from constant memory costs one memory read from global memory only on a cache miss; otherwise, it just costs one read from the constant cache. Immediate constants are encoded into the SASS instruction as 'c[bank][offset]'. All threads access the same value.

Thread Divergence Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 1.0 threads being active per cycle. This is further reduced to 0.7 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().

Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of instruction types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'Instructions/Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	3	Avg. Executed Instructions Per Scheduler [inst]	0.01
Issued Instructions [inst]	4	Avg. Issued Instructions Per Scheduler [inst]	0.01

Launch Statistics

Summary of the configuration used to launch the kernel. The launch configuration defines the size of the kernel grid, the division of the grid into blocks, and the GPU resources needed to execute the kernel. Choosing an efficient launch configuration maximizes device utilization.

Grid Size	1	Registers Per Thread [register/thread]	16
Block Size	1	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	1	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	0.00	Driver Shared Memory Per Block [byte/block]	0
		Shared Memory Configuration Size [byte]	0

Launch Configuration Threads are executed in groups of 32 threads called warps. This kernel launch is configured to execute 1 threads per block. Consequently, some threads in a warp are masked off and those hardware resources are unused. Try changing the number of threads per block to be a multiple of 32 threads. Between 128 and 256 threads per block is a good initial range for experimentation. Use smaller thread blocks rather than one large thread block per multiprocessor if latency affects performance. This is particularly beneficial to kernels that frequently call __syncthreads().

Launch Configuration The grid for this launch is configured to execute only 1 blocks, which is less than the GPU's 80 multiprocessors. This can underutilize some multiprocessors. If you do not intend to execute this kernel concurrently with other workloads, consider reducing the block size to have at least one block per multiprocessor or increase the size of the grid to fully utilize the available hardware resources.

Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	128
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	1.56	Block Limit Warps [block]	64
Achieved Active Warps Per SM [warp]	1	Block Limit SM [block]	32

Occupancy Occupancy section results analysis Apply

Source Counters

Source metrics, including warp stall reasons. Sampling Data metrics are periodically sampled over the kernel runtime. They indicate when warps were stalled and couldn't be scheduled. See the documentation for a description of all stall reasons. Only focus on stalls if the schedulers fail to issue every cycle.

(Deprecated) Memory Workload Analysis

Deprecated UI elements for backwards compatibility.

To customize your report even further, you might want to learn about custom sections and writing your own rules.
You might also want to consider adding individual metrics.

- What references did you use when implementing this technique?

<I used one of the earlier lectures on DRAM bursting, L2 cache, and memory architecture.>

- Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization.

For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the “for” loop for loop unrolling
<answer here>

```
__constant__ float MASK[4000];

// #include <cmath>
// #include <iostream>
// #include "gpu-new-forward.h"

// #define TILE_WIDTH 16

// __global__ void conv_forward_kernel(float *output,
const float *input, const float *mask, const int Batch,
const int Map_out, const int Channel, const int Height,
const int Width, const int K)
// {
//     /*
//     Modify this function to implement the forward
pass described in Chapter 16.
//     We have added an additional dimension to the
tensors to support an entire mini-batch
//     The goal here is to be correct AND fast.

//     Function paramter definitions:
//     output - output
//     input - input
//     mask - convolution kernel
//     Batch - batch_size (number of images in x)
//     Map_out - number of output feature maps
//     Channel - number of input feature maps
//     Height - input height dimension
//     Width - input width dimension
//     K - kernel height and width (K x K)
//     */

//     const int Height_out = Height - K + 1;
//     const int Width_out = Width - K + 1;
//     //(void)Height_out; // silence declared but
never referenced warning. remove this line when you
start working
```

```
//      //(void)Width_out; // silence declared but never
referenced warning. remove this line when you start
working

//      // We have some nice #defs for you below to
simplify indexing. Feel free to use them, or create
your own.
//      // An example use of these macros:
//      // float a = in_4d(0,0,0,0)
//      // out_4d(0,0,0,0) = a

//      #define out_4d(i3, i2, i1, i0) output[(i3) *
(Map_out * Height_out * Width_out) + (i2) * (Height_out
* Width_out) + (i1) * (Width_out) + i0]
//      #define in_4d(i3, i2, i1, i0) input[(i3) *
(Channel * Height * Width) + (i2) * (Height * Width) +
(i1) * (Width) + i0]
//      #define mask_4d(i3, i2, i1, i0) MASK[(i3) *
(Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]
```

- **Optimization 4: <*optimization name*>**
  - Which optimization did you choose to implement? Chose from the optimization below by clicking on the check box and explain why did you choose

that optimization technique.

□Tiled shared memory convolution (**2 points**) THIS ONE
□ Shared memory matrix multiplication and input matrix unrolling (**3 points**)
□ Kernel fusion for unrolling and matrix-multiplication (**2 points**)
□ Weight matrix in constant memory (**1 point**)
□ Tuning with restrict and loop unrolling (**3 points**)
□ Sweeping various parameters to find best values (**1 point**)
□ Multiple kernel implementations for different layer sizes (**1 point**)
□ Input channel reduction: tree (**3 point**)
□ Input channel reduction: atomics (**2 point**)
□ Fixed point (FP16) arithmetic. (**4 points**)
□ Using Streams to overlap computation with data transfer (**4 points**)
□ An advanced matrix multiplication algorithm (**5 points**)
□ Using Tensor Cores to speed up matrix multiplication (**5 points**)
□ Overlap-Add method for FFT-based convolution (**8 points**)
□ Other optimizations: please explain
<answer here>

- 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?
<your answer here>

- 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
<op_time>
<op_time>
<exec_time>
<accuracy>
1000
<op_time>
<op_time>
<exec_time>
<accuracy>

10000
<op_time>
<op_time>
<exec_time>
<accuracy>

- 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

- What references did you use when implementing this technique?

- Please Paste your kernel code for this optimization. Your code should include the non-trivial code that you have changed for this optimization. For example, it can be the complete kernel code for Tiled shared memory convolution several lines of code for Weight matrix in constant memory, or the "for" loop for loop unrolling
<answer here>