Name: *Kaushik Indukuri*
NetID: *kvi3*
Section: *ECE408 AL/AB*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.73674 ms*	*2.34653 ms*	*1.231 secs*	*0.86*
1000	*1.79239 ms*	*6.92718 ms*	*10.162 secs*	*0.886*
10000	*17.738 ms*	*69.3869 ms*	*1 min 43.661 secs*	*0.8714*

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

I choose this optimization because I thought it was really cool how you are able to divide the trivial tasks of copy memory and executing a kernel even further into streams of execution queues

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

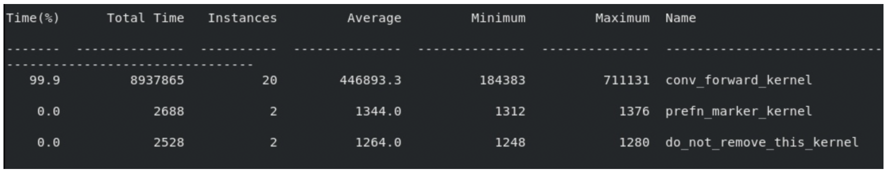
This optimization works by allowing the device overlap which simultaneously executes a kernel while performing a copy between device and host memory. These streams divide the data sets into batches and assign each batch to a stream. Within a stream, there are three major operations that are added to its queue of execution: Memcpy to device, execute kernel function, and Memcpy back to host. I believed this would increase performance significantly by allowing even further parallelism between redundant and independent tasks. This optimization synergizes with additional optimizations as most optimizations require two memcpy's and a kernel call.

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

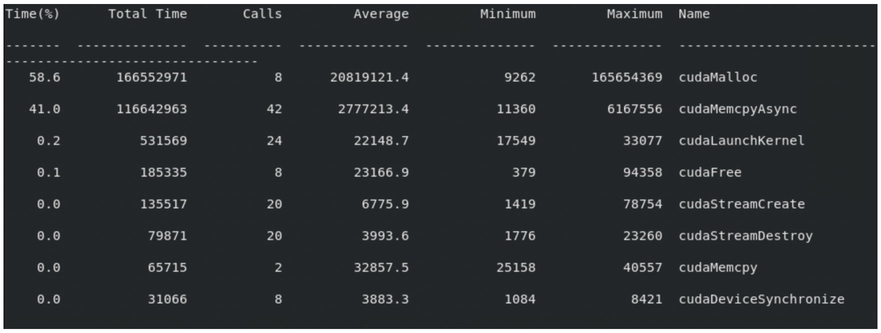
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.001047 ms*	*0.001274ms*	*1.224 secs*	*0.86*
1000	*0.000816 ms*	*0.000689ms*	*10.702 sec*	*0.886*
10000	*0.000741 ms*	*0.00086 ms*	*1min42 secs*	*0.8714*

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

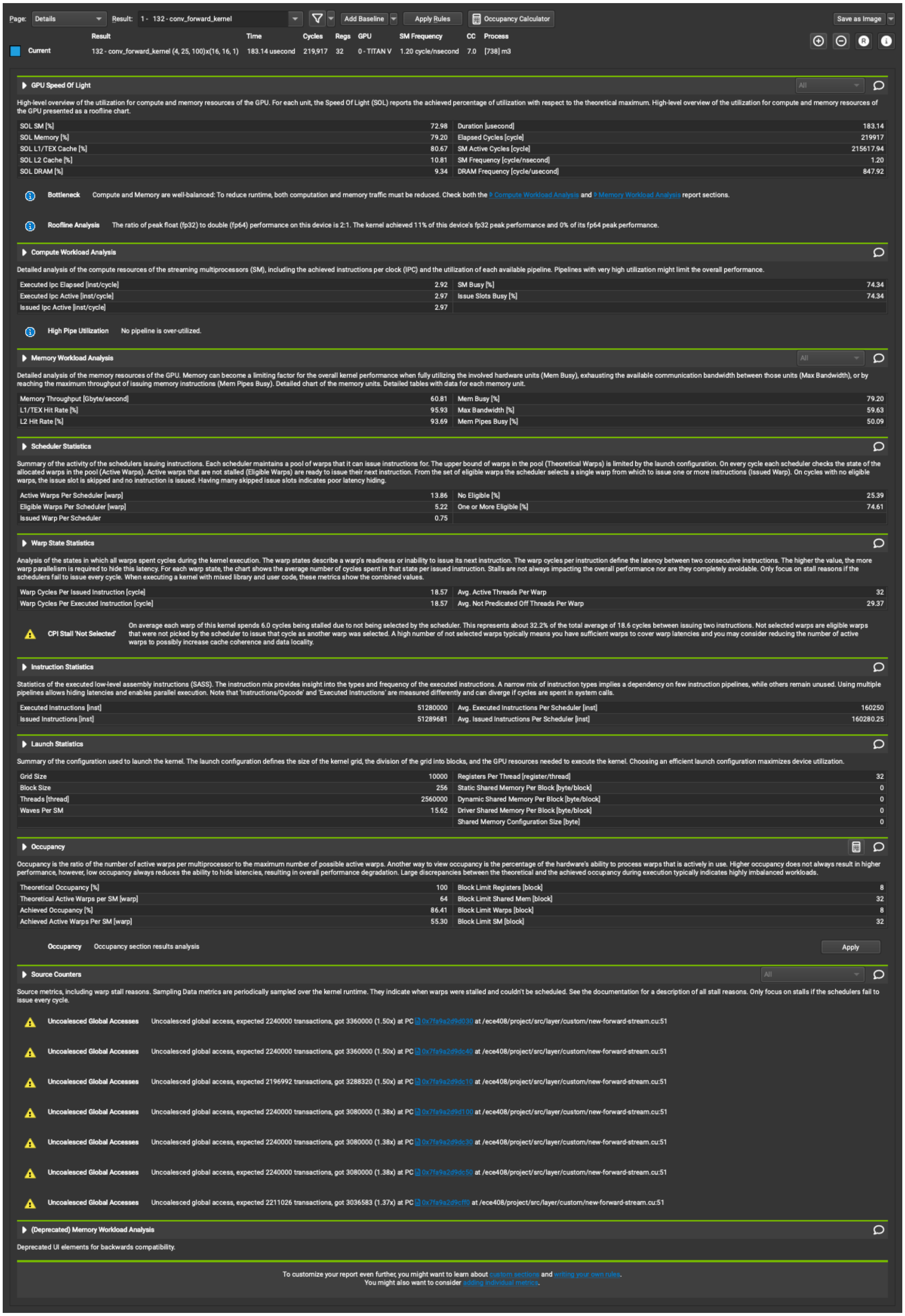
The streams were successful in that they allowed for more parallelism within the GPU, making the usage of the device more efficient. This can be shown through the screenshots below in that they show that the device and streaming multiprocessor use was much more efficient.



```
Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name

-------  -------------  ----------  -------------  -------------  -------------  ----------------------------
-------------------------------
   99.9        8937865          20        446893.3          184383          711131  conv_forward_kernel

    0.0           2688           2          1344.0            1312            1376  prefn_marker_kernel

    0.0           2528           2          1264.0            1248            1280  do_not_remove_this_kernel
```



```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name

-------  -------------  ----------  -------------  -------------  -------------  ------------------------
-------------------------------
   58.6      166552971           8      20819121.4            9262       165654369  cudaMalloc

   41.0      116642963          42       2777213.4           11360         6167556  cudaMemcpyAsync

    0.2         531569          24         22148.7           17549           33077  cudaLaunchKernel

    0.1         185335           8         23166.9             379           94358  cudaFree

    0.0         135517          20          6775.9            1419           78754  cudaStreamCreate

    0.0          79871          20          3993.6            1776           23260  cudaStreamDestroy

    0.0          65715           2         32857.5           25158           40557  cudaMemcpy

    0.0          31066           8          3883.3            1084            8421  cudaDeviceSynchronize
```



Page: Details   Result: 1 - 132 - conv_forward_kernel

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	132 - conv_forward_kernel (4, 25, 100)x(16, 16, 1)	183.14 usecond	219,917	32	0 - TITAN V	1.20 cycle/nsecond	7.0	[738] 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 [%]	72.98	Duration [usecond]	183.14
SOL Memory [%]	79.20	Elapsed Cycles [cycle]	219917
SOL L1/TEX Cache [%]	80.67	SM Active Cycles [cycle]	215617.94
SOL L2 Cache [%]	10.81	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	9.34	DRAM Frequency [cycle/usecond]	847.92

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.

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]	2.92	SM Busy [%]	74.34
Executed Ipc Active [inst/cycle]	2.97	Issue Slots Busy [%]	74.34
Issued Ipc Active [inst/cycle]	2.97		

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]	60.81	Mem Busy [%]	79.20
L1/TEX Hit Rate [%]	95.93	Max Bandwidth [%]	59.63
L2 Hit Rate [%]	93.69	Mem Pipes Busy [%]	50.09

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]	13.86	No Eligible [%]	25.39
Eligible Warps Per Scheduler [warp]	5.22	One or More Eligible [%]	74.61
Issued Warp Per Scheduler	0.75		

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]	18.57	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	18.57	Avg. Not Predicated Off Threads Per Warp	29.37

⚠ **CPI Stall 'Not Selected'** On average each warp of this kernel spends 6.0 cycles being stalled due to not being selected by the scheduler. This represents about 32.2% of the total average of 18.6 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.

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]	51280000	Avg. Executed Instructions Per Scheduler [inst]	160250
Issued Instructions [inst]	51289681	Avg. Issued Instructions Per Scheduler [inst]	160280.25

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	10000	Registers Per Thread [register/thread]	32
Block Size	256	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	2560000	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	15.62	Driver Shared Memory Per Block [byte/block]	0
		Shared Memory Configuration Size [byte]	0

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 [%]	100	Block Limit Registers [block]	8
Theoretical Active Warps per SM [warp]	64	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	86.41	Block Limit Warps [block]	8
Achieved Active Warps Per SM [warp]	55.30	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.

⚠ **Uncoalesced Global Accesses** Uncoalesced global access, expected 2240000 transactions, got 3360000 (1.50x) at PC 0x7fa9a2d9d030 at /ece408/project/src/layer/custom/new-forward-stream.cu:51

⚠ **Uncoalesced Global Accesses** Uncoalesced global access, expected 2240000 transactions, got 3360000 (1.50x) at PC 0x7fa9a2d9dc40 at /ece408/project/src/layer/custom/new-forward-stream.cu:51

⚠ **Uncoalesced Global Accesses** Uncoalesced global access, expected 2196992 transactions, got 3288320 (1.50x) at PC 0x7fa9a2d9dc10 at /ece408/project/src/layer/custom/new-forward-stream.cu:51

⚠ **Uncoalesced Global Accesses** Uncoalesced global access, expected 2240000 transactions, got 3080000 (1.38x) at PC 0x7fa9a2d9d100 at /ece408/project/src/layer/custom/new-forward-stream.cu:51

⚠ **Uncoalesced Global Accesses** Uncoalesced global access, expected 2240000 transactions, got 3080000 (1.38x) at PC 0x7fa9a2d9dc30 at /ece408/project/src/layer/custom/new-forward-stream.cu:51

⚠ **Uncoalesced Global Accesses** Uncoalesced global access, expected 2240000 transactions, got 3080000 (1.38x) at PC 0x7fa9a2d9dc50 at /ece408/project/src/layer/custom/new-forward-stream.cu:51

⚠ **Uncoalesced Global Accesses** Uncoalesced global access, expected 2211026 transactions, got 3036583 (1.37x) at PC 0x7fa9a2d9cff0 at /ece408/project/src/layer/custom/new-forward-stream.cu:51

(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.

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

I used lecture 22 from class as well as this NVIDIA blog post for an additional example with multiple streams: https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/.

f. 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

```
  const int Height_out = Height - K + 1;
  const int Width_out = Width - K + 1;
  int H_grid = ceil(1.0 * Height_out / TILE_WIDTH);
  int W_grid = ceil(1.0 * Width_out / TILE_WIDTH);
  int Y = H_grid * W_grid;

  dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
  dim3 gridDim(Map_out, Y, Batch/numstreams);

  cudaMalloc((void**)device_output_ptr, Batch * Map_out * Height_out * Width_out * sizeof(float));
  cudaMalloc((void**)device_mask_ptr, Channel * Map_out * K * K * sizeof(float));
  cudaMalloc((void**)device_input_ptr, Batch * Channel * Height * Width * sizeof(float));

  cudaStream_t streams[numstreams];
  int i;
  for (i = 0; i < numstreams; i++) {
    cudaStreamCreate(&streams[i]);
  }
  cudaMemcpyAsync(*device_mask_ptr, host_mask, Channel * Map_out * K * K * sizeof(float),
cudaMemcpyHostToDevice, streams[0]);

  int instreamdiv = Batch * Channel * Height * Width;
  instreamdiv /= numstreams;
  int outstreamdiv = Batch * Map_out * Height_out * Width_out;
  outstreamdiv /= numstreams;
  float * device_input = *device_input_ptr;
  float * device_output = *device_output_ptr;
  float * device_mask = *device_mask_ptr;
  for (i = 0; i < numstreams; i++) {
        cudaMemcpyAsync(device_input + instreamdiv * i, host_input + instreamdiv * i, instreamdiv *
sizeof(float), cudaMemcpyHostToDevice, streams[i]);
        conv_forward_kernel<<<gridDim, blockDim, 0, streams[i]>>>(device_output + outstreamdiv * i,
device_input + instreamdiv * i, device_mask, Batch, Map_out, Channel, Height, Width, K);
        cudaMemcpyAsync((float*)host_output + outstreamdiv * i, device_output + outstreamdiv * i,
outstreamdiv * sizeof(float), cudaMemcpyDeviceToHost, streams[i]);
    }
    cudaDeviceSynchronize();

    for (i = 0; i < numstreams; i++) {
        cudaStreamDestroy(streams[i]);
    }

    cudaFree(device_input_ptr);
    cudaFree(device_mask_ptr);
    cudaFree(device_output_ptr);
```

2. **Optimization 2: Input channel reduction: tree**

    a. 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**)
    ☐ 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

    I chose this optimization because I was really comfortable with applying the reduction tree to other concepts as I felt I learned it well in lecture and the readings.

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

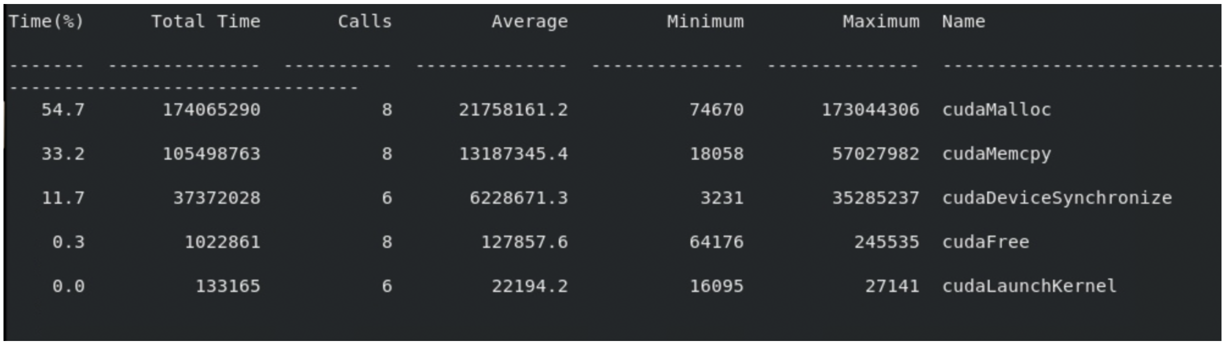
This optimization works by using the reduction tree method we learned and applying it to the channel dimension by reducing the samples based on their channels as we are adding them eventually. I thought it would increase performance by reducing the amount of overhead as well as making it work efficient. This optimization synergizes with only the optimizations that preserve the convolution kernel code we used in milestone 2.

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

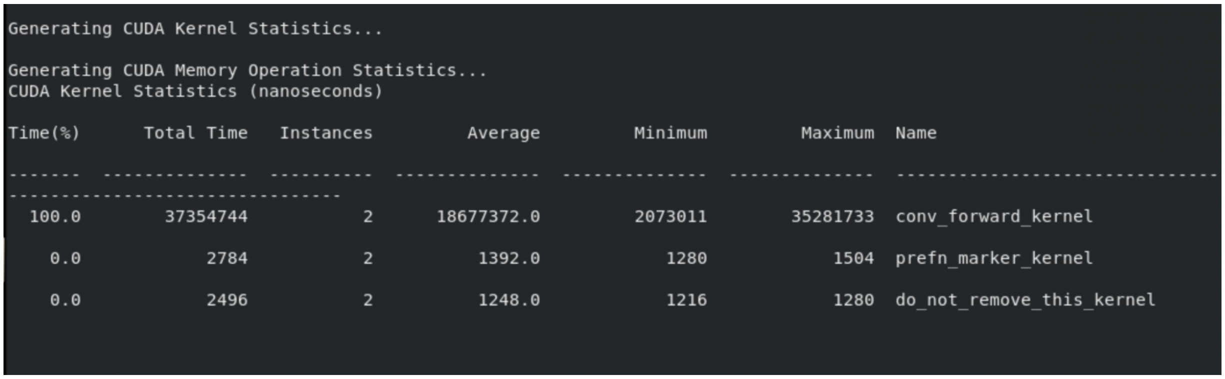
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.225ms*	*1.05ms*	*1.497sec*	*0.86*
1000	*2.06ms*	*10.93ms*	*9.636sec*	*0.886*
10000	*20.44ms*	*109.3ms*	*1min34sec*	*0.8714*

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

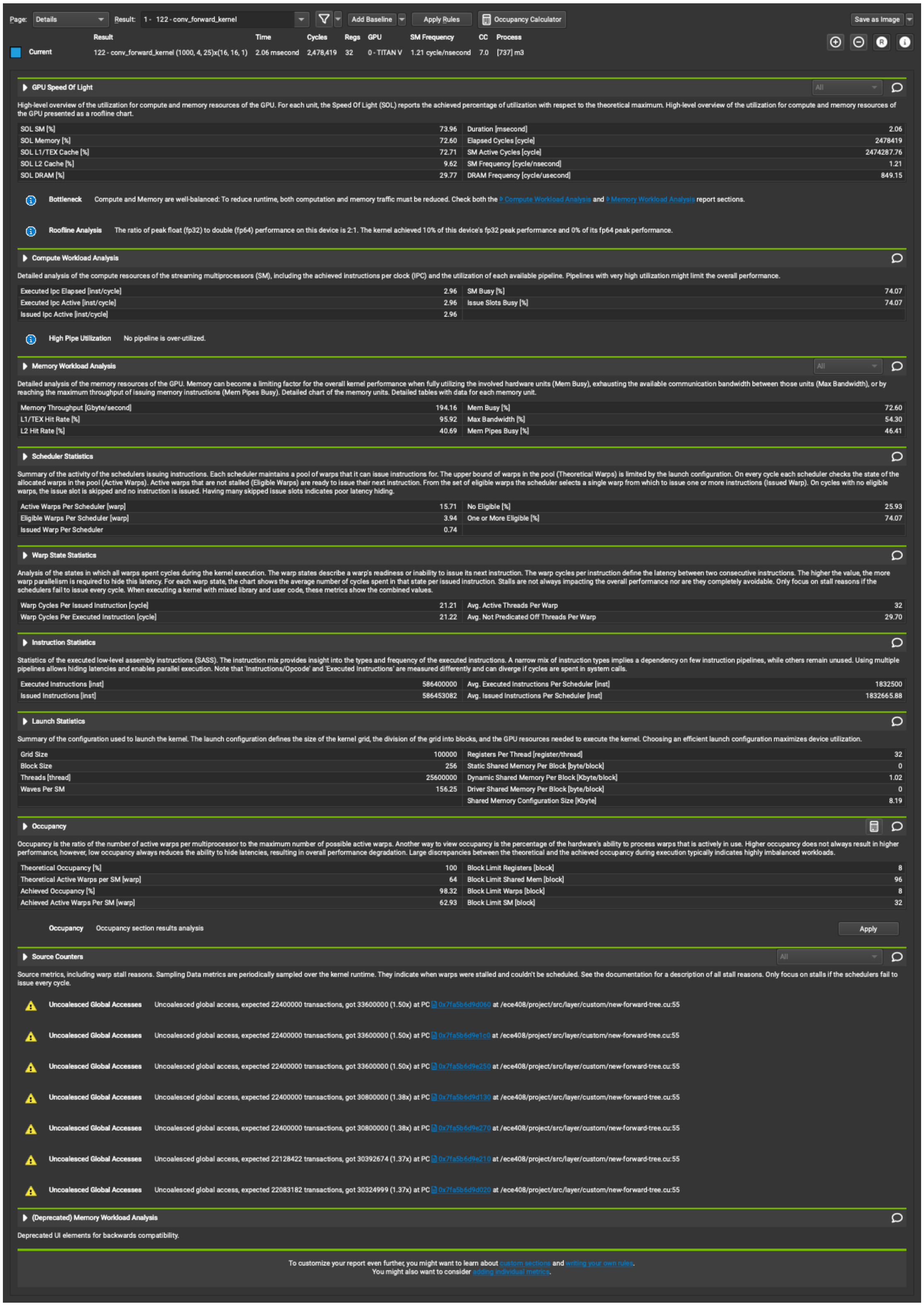
The tree reduction was not as successful as I hoped as the operation time increased from 1.73 seconds to 2.06. Additionally, the memory accesses were inefficient as the hit rates for the cashes decreased significantly. I had hoped that the reduction would allow for a much faster speed of data movement from the accumulated variable to the output, but the results show that the tradeoffs are not effective.



```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name
-------  -------------  ----------  -------------  -------------  -------------  -------------------------
--------------------------------
   54.7       174065290           8      21758161.2           74670       173044306  cudaMalloc
   33.2       105498763           8      13187345.4           18058        57027982  cudaMemcpy
   11.7        37372028           6       6228671.3            3231        35285237  cudaDeviceSynchronize
    0.3         1022861           8        127857.6           64176          245535  cudaFree
    0.0          133165           6         22194.2           16095           27141  cudaLaunchKernel
```



```
Generating CUDA Kernel Statistics...

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

Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name
-------  -------------  ----------  -------------  -------------  -------------  ------------------------------
-------------------------------
  100.0        37354744           2      18677372.0         2073011        35281733  conv_forward_kernel
    0.0            2784           2          1392.0            1280            1504  prefn_marker_kernel
    0.0            2496           2          1248.0            1216            1280  do_not_remove_this_kernel
```



Page: Details Result: 1 - 122 - conv_forward_kernel

	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (1000, 4, 25)x(16, 16, 1)	2.06 msecond	2,478,419	32	0 - TITAN V	1.21 cycle/nsecond	7.0	[737] 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 [%]	73.96	Duration [msecond]	2.06
SOL Memory [%]	72.60	Elapsed Cycles [cycle]	2478419
SOL L1/TEX Cache [%]	72.71	SM Active Cycles [cycle]	2474287.76
SOL L2 Cache [%]	9.62	SM Frequency [cycle/nsecond]	1.21
SOL DRAM [%]	29.77	DRAM Frequency [cycle/usecond]	849.15

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.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 2:1. The kernel achieved 10% 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]	2.96	SM Busy [%]	74.07
Executed Ipc Active [inst/cycle]	2.96	Issue Slots Busy [%]	74.07
Issued Ipc Active [inst/cycle]	2.96		

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]	194.16	Mem Busy [%]	72.60
L1/TEX Hit Rate [%]	95.92	Max Bandwidth [%]	54.30
L2 Hit Rate [%]	40.69	Mem Pipes Busy [%]	46.41

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]	15.71	No Eligible [%]	25.93
Eligible Warps Per Scheduler [warp]	3.94	One or More Eligible [%]	74.07
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]	21.21	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	21.22	Avg. Not Predicated Off Threads Per Warp	29.70

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]	586400000	Avg. Executed Instructions Per Scheduler [inst]	1832500
Issued Instructions [inst]	586453082	Avg. Issued Instructions Per Scheduler [inst]	1832665.88

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	100000	Registers Per Thread [register/thread]	32
Block Size	256	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	25600000	Dynamic Shared Memory Per Block [Kbyte/block]	1.02
Waves Per SM	156.25	Driver Shared Memory Per Block [byte/block]	0
		Shared Memory Configuration Size [Kbyte]	8.19

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 [%]	100	Block Limit Registers [block]	8
Theoretical Active Warps per SM [warp]	64	Block Limit Shared Mem [block]	96
Achieved Occupancy [%]	98.32	Block Limit Warps [block]	8
Achieved Active Warps Per SM [warp]	62.93	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.

Uncoalesced Global Accesses Uncoalesced global access, expected 22400000 transactions, got 33600000 (1.50x) at PC 0x7fa5b6d9d060 at /ece408/project/src/layer/custom/new-forward-tree.cu:55

Uncoalesced Global Accesses Uncoalesced global access, expected 22400000 transactions, got 33600000 (1.50x) at PC 0x7fa5b6d9e1c0 at /ece408/project/src/layer/custom/new-forward-tree.cu:55

Uncoalesced Global Accesses Uncoalesced global access, expected 22400000 transactions, got 33600000 (1.50x) at PC 0x7fa5b6d9e250 at /ece408/project/src/layer/custom/new-forward-tree.cu:55

Uncoalesced Global Accesses Uncoalesced global access, expected 22400000 transactions, got 30800000 (1.38x) at PC 0x7fa5b6d9d130 at /ece408/project/src/layer/custom/new-forward-tree.cu:55

Uncoalesced Global Accesses Uncoalesced global access, expected 22400000 transactions, got 30800000 (1.38x) at PC 0x7fa5b6d9e270 at /ece408/project/src/layer/custom/new-forward-tree.cu:55

Uncoalesced Global Accesses Uncoalesced global access, expected 22128422 transactions, got 30392674 (1.37x) at PC 0x7fa5b6d9e210 at /ece408/project/src/layer/custom/new-forward-tree.cu:55

Uncoalesced Global Accesses Uncoalesced global access, expected 22083182 transactions, got 30324999 (1.37x) at PC 0x7fa5b6d9d020 at /ece408/project/src/layer/custom/new-forward-tree.cu:55

(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.

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

I used the lecture slides and the reading going over tree reduction.

f. 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

```
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;
    int H_grid = (Height_out + TILE_WIDTH - 1) / TILE_WIDTH;
    int W_grid = (Width_out + TILE_WIDTH - 1) / TILE_WIDTH;
    //(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

    extern __shared__ float tree[];

    #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]
    #define tree_3d(i2, i1, i0) tree[(i2) * (Channel * TILE_WIDTH) + (i1) * (Channel) + i0]



    // Insert your GPU convolution kernel code here
    int m = blockIdx.x;
    int h = (blockIdx.z / W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.z % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0.0;
    if (h < Height_out && w < Width_out) {
        for (int p = 0; p < K; p++) {
          for (int q = 0; q < K; q++) {
            acc += in_4d(blockIdx.x, threadIdx.z, h + p, w + q) * mask_4d(blockIdx.y, threadIdx.z, p, q);
         }
      }
      tree_3d(threadIdx.y, threadIdx.x, threadIdx.z) = acc;
      for (unsigned int stride = 1; stride < Channel; stride <<= 1) {
        __syncthreads();
        if ((threadIdx.z + stride < Channel) && (threadIdx.z % (2 * stride) == 0)) {
          tree_3d(threadIdx.y, threadIdx.x, threadIdx.z) += tree_3d(threadIdx.y, threadIdx.x,
  threadIdx.z + stride);
        }
      }
      __syncthreads();
      out_4d(blockIdx.x, blockIdx.y, h, w) = tree_3d(threadIdx.y, threadIdx.x, 0);
    }
```

3. **Optimization 3: Fixed point (FP16) arithmetic**

    a. 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**)
    ☐ 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

    I chose this optimization as I come from a hardware background so it was really interesting to me how we are tampering with the amount of data stored in a float and data structure.

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?

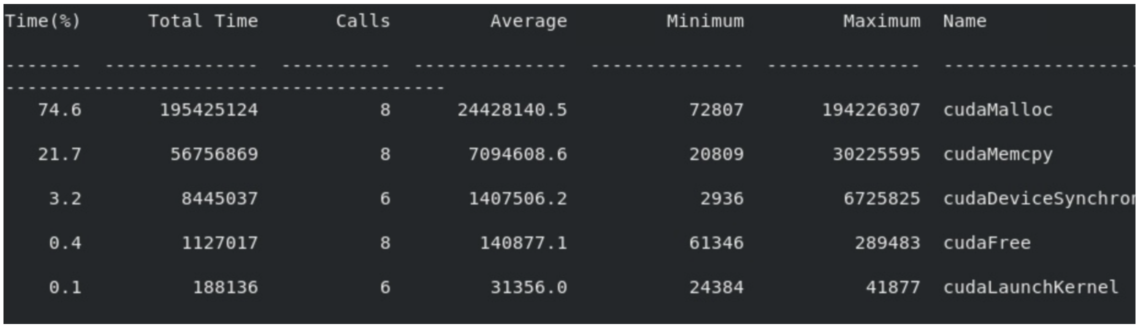
The fp16 performance optimization works by reducing the time needed for each floating point operation. Typically, floats are represented by 32 bits but fp16 operations use the __half data type which are 16 bits. The floats fed in are converted to __half as they are passed in. Special operations are used to perform multiplication and addition on this data type. This makes computations faster along with making memory transfers more efficient. This optimization will increase the performance of forward convolution at the cost of a slight decrease in accuracy.

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

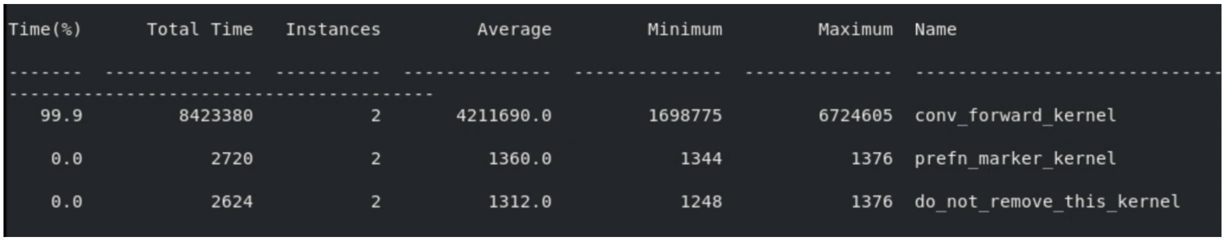
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.184ms*	*0.696ms*	*1.365sec*	*0.77*
1000	*1.704ms*	*6.722ms*	*10.767sec*	*0.829*
10000	*16.884ms*	*66.922ms*	*1min45sec*	*0.84*

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

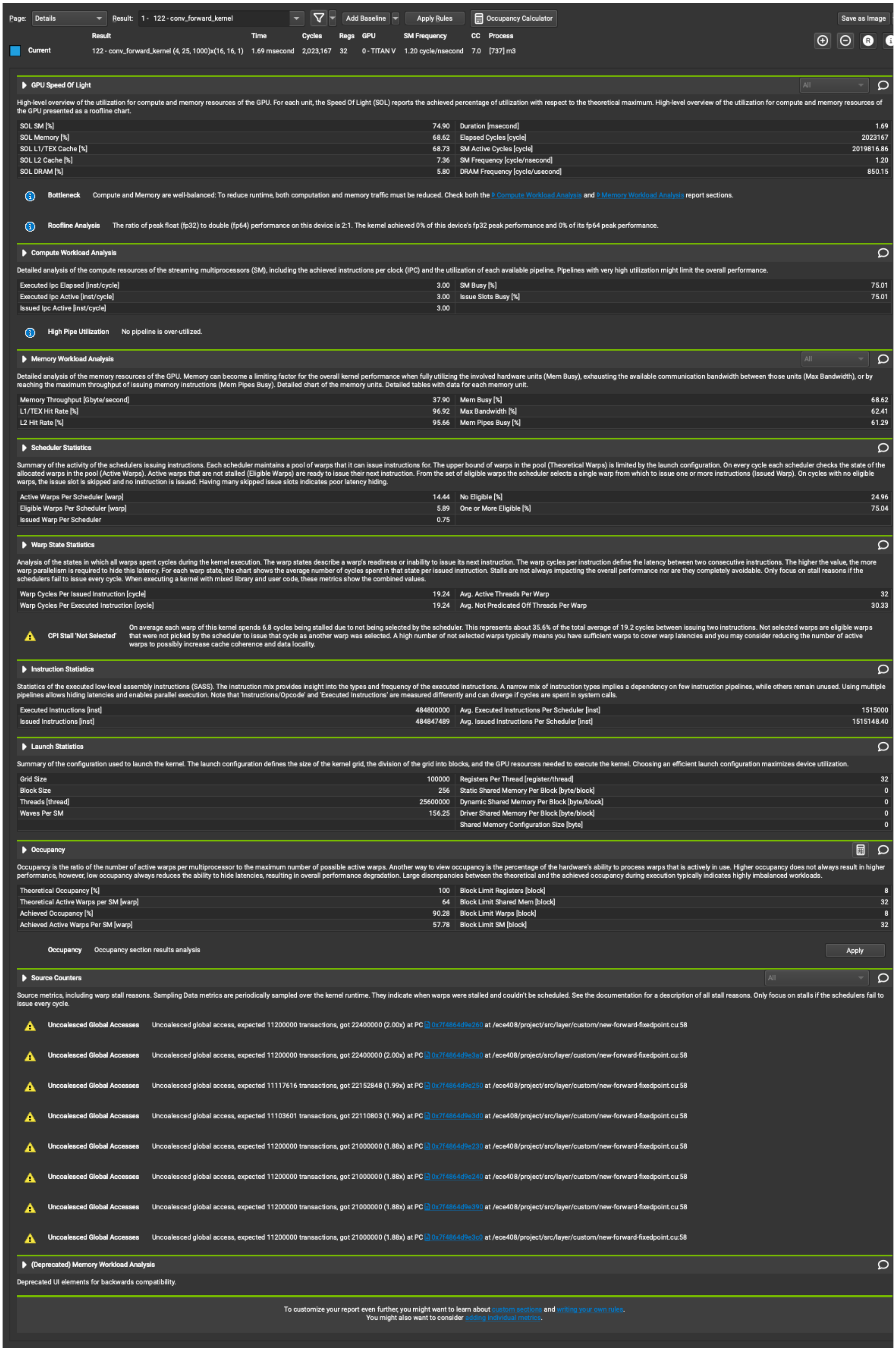
Yes, this performance optimization was successful. The operation time decreased from 1.77 seconds to 1.69 seconds after the optimization. Memory usage was also greatly decreased. This is likely a result of the 16 bit half data type which reduced memory transfers and space allocated.



```
Time(%)      Total Time       Calls         Average         Minimum         Maximum  Name

-------  --------------  ----------  --------------  --------------  --------------  ------------------
----------------------------------------
   74.6       195425124           8      24428140.5           72807       194226307  cudaMalloc

   21.7        56756869           8       7094608.6           20809        30225595  cudaMemcpy

    3.2         8445037           6       1407506.2            2936         6725825  cudaDeviceSynchron

    0.4         1127017           8        140877.1           61346          289483  cudaFree

    0.1          188136           6         31356.0           24384           41877  cudaLaunchKernel
```



```
Time(%)      Total Time   Instances         Average         Minimum         Maximum  Name

-------  -------------  ----------  --------------  --------------  --------------  ---------------------------
---------------------------------------
   99.9         8423380           2       4211690.0         1698775         6724605  conv_forward_kernel

    0.0            2720           2          1360.0            1344            1376  prefn_marker_kernel

    0.0            2624           2          1312.0            1248            1376  do_not_remove_this_kernel
```



	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	122 - conv_forward_kernel (4, 25, 1000)x(16, 16, 1)	1.69 msecond	2,023,167	32	0 - TITAN V	1.20 cycle/nsecond	7.0	[737] 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 [%]	74.90	Duration [msecond]	1.69
SOL Memory [%]	68.62	Elapsed Cycles [cycle]	2023167
SOL L1/TEX Cache [%]	68.73	SM Active Cycles [cycle]	2019816.86
SOL L2 Cache [%]	7.36	SM Frequency [cycle/nsecond]	1.20
SOL DRAM [%]	5.80	DRAM Frequency [cycle/usecond]	850.15

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.

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]	3.00	SM Busy [%]	75.01
Executed Ipc Active [inst/cycle]	3.00	Issue Slots Busy [%]	75.01
Issued Ipc Active [inst/cycle]	3.00		

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]	37.90	Mem Busy [%]	68.62
L1/TEX Hit Rate [%]	96.92	Max Bandwidth [%]	62.41
L2 Hit Rate [%]	95.66	Mem Pipes Busy [%]	61.29

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]	14.44	No Eligible [%]	24.96
Eligible Warps Per Scheduler [warp]	5.89	One or More Eligible [%]	75.04
Issued Warp Per Scheduler	0.75		

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]	19.24	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	19.24	Avg. Not Predicated Off Threads Per Warp	30.33

⚠ CPI Stall 'Not Selected' On average each warp of this kernel spends 6.8 cycles being stalled due to not being selected by the scheduler. This represents about 35.6% of the total average of 19.2 cycles between issuing two instructions. Not selected warps are eligible warps that were not picked by the scheduler to issue that cycle as another warp was selected. A high number of not selected warps typically means you have sufficient warps to cover warp latencies and you may consider reducing the number of active warps to possibly increase cache coherence and data locality.

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]	484800000	Avg. Executed Instructions Per Scheduler [inst]	1515000
Issued Instructions [inst]	484847489	Avg. Issued Instructions Per Scheduler [inst]	1515148.40

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	100000	Registers Per Thread [register/thread]	32
Block Size	256	Static Shared Memory Per Block [byte/block]	0
Threads [thread]	25600000	Dynamic Shared Memory Per Block [byte/block]	0
Waves Per SM	156.25	Driver Shared Memory Per Block [byte/block]	0
		Shared Memory Configuration Size [byte]	0

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 [%]	100	Block Limit Registers [block]	8
Theoretical Active Warps per SM [warp]	64	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	90.28	Block Limit Warps [block]	8
Achieved Active Warps Per SM [warp]	57.78	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.

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11200000 transactions, got 22400000 (2.00x) at PC 0x7f4864d9e260 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11200000 transactions, got 22400000 (2.00x) at PC 0x7f4864d9e3a0 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11117616 transactions, got 22152848 (1.99x) at PC 0x7f4864d9e250 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11103601 transactions, got 22110803 (1.99x) at PC 0x7f4864d9e3d0 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11200000 transactions, got 21000000 (1.88x) at PC 0x7f4864d9e230 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11200000 transactions, got 21000000 (1.88x) at PC 0x7f4864d9e240 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11200000 transactions, got 21000000 (1.88x) at PC 0x7f4864d9e390 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

⚠ Uncoalesced Global Accesses Uncoalesced global access, expected 11200000 transactions, got 21000000 (1.88x) at PC 0x7f4864d9e3c0 at /ece408/project/src/layer/custom/new-forward-fixedpoint.cu:58

(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.

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

I used this link to learn more about the data types that NVIDIA commonly uses

https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html

f. 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

```
__half2 load1;
__half2 load2;
__half2 acc = __half2half2(0);
__half2 halfmul;

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+=2) {
        load2 = __halves2half2(mask_4d(m,c,p,q),mask_4d(m,c,p,q+1));
        load1 = __halves2half2(in_4d(blockIdx.z,c,h+p,w+q),in_4d(blockIdx.z,c,h+p,w+q+1));
        halfmul = __hmul2(load1,load2);
        acc = __hadd2(acc,halfmul);
      }
    }
  }
  out_4d(blockIdx.z, m, h, w) = __hadd(__high2half(acc),__low2half(acc));
}
```