Name: *Ruiqi Li*
NetID: *Ruiqili4*
Section: *AL1*

ECE 408/CS483 Milestone 3 Report

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.338023 ms*	*1.21262 ms*	*0m1.141s*	*0.86*
1000	*3.08233 ms*	*11.9752 ms*	*0m9.566s*	*0.886*
10000	*30.435 ms*	*120.535 ms*	*1m38.909s*	*0.8714*

1. **Optimization 1:** ***Tiled shared memory convolution***

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

      I chose Tiled shared memory convolution. Because this is the most intuitive way to accelerate computation.

   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 optimization works very well. I think the optimization would increase performance of the forward convolution, because it replaces the global memory traffic with shared memory, therefore increases the execution speed. The execution speed of the kernel will be limited by the global memory bandwidth. But if we use shared memory tiling to temporarily store data in each block, the computation time can be significantly reduced. The optimization doesn't synergize with previous optimizations.

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	*0.226206 ms*	*1.01411 ms*	*0m1.601s*	*0.86*
1000	*1.99117 ms*	*9.94768 ms*	*0m9.905s*	*0.886*
10000	*19.2631 ms*	*98.5901 ms*	*1m34.265s*	*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).

Yes, it successfully improved performance. The execution time decreased with this optimization.
Since I only modify the memory usage of each tile, namely, replace global memory access with shared memory, the memory copy and allocation times are roughly identical. Here is the comparison between nsys results of this optimization and that of the baseline.

```
Time(%)  Total Time   Calls    Average      Minimum     Maximum   Name
-------  -----------  -------  -----------  ----------  ----------  --------------------
 81.2    1546321026     10    154632102.6     19306     598762736  cudaMemcpy
  9.6     181871156      8     22733894.5     70365     176797902  cudaMalloc
  7.9     151134746      8     18891843.2      1424     120623695  cudaDeviceSynchronize
  1.1      21709731      6      3618288.5     23201      21577572  cudaLaunchKernel
  0.1       2851037      8       356379.6     67387       1030270  cudaFree

Generating CUDA Kernel Statistics...

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

Time(%)  Total Time  Instances   Average     Minimum     Maximum   Name
-------  ----------  ---------  ----------  ----------  ----------  --------------------
100.0    151096614       2      75548307.0   30474316   120622298  conv_forward_kernel
  0.0         2784       2          1392.0       1376        1408  do_not_remove_this_kernel
  0.0         2656       2          1328.0       1312        1344  prefn_marker_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)  Total Time  Operations   Average     Minimum     Maximum   Name
-------  ----------  ----------  ----------  ----------  ----------  --------------------
 66.9    1029784465       2      514892232.5  431950981   597833484  [CUDA memcpy DtoH]
 33.1     510150122       8       63768765.2       1184   247256301  [CUDA memcpy HtoD]

CUDA Memory Operation Statistics (KiB)

     Total    Operations    Average     Minimum      Maximum   Name
 -----------  ----------  ----------  ----------  ----------  --------------------
  2261419.0        8        282677.0       0.004   1000000.0  [CUDA memcpy HtoD]
  1722500.0        2        861250.0  722500.000   1000000.0  [CUDA memcpy DtoH]
```

Baseline ↑

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
81.8	1615060821	10	161506082.1	22541	653661406	cudaMemcpy
10.9	215090157	8	26886269.6	95378	209962132	cudaMalloc
6.0	118675057	8	14834382.1	1139	99182254	cudaDeviceSynchronize
1.2	22807227	6	3801204.5	26244	22660964	cudaLaunchKernel
0.1	2794053	8	349256.6	67514	960841	cudaFree

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	118644791	2	59322395.5	19464881	99179910	conv_forward_kernel
0.0	2912	2	1456.0	1408	1504	do_not_remove_this_kernel
0.0	2688	2	1344.0	1344	1344	prefn_marker_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
68.2	1093096865	2	546548432.5	440407757	652689108	[CUDA memcpy DtoH]
31.8	509079359	8	63634919.9	1184	249950584	[CUDA memcpy HtoD]

CUDA Memory Operation Statistics (KiB)

Total	Operations	Average	Minimum	Maximum	Name
2261419.0	8	282677.0	0.004	1000000.0	[CUDA memcpy HtoD]
1722500.0	2	861250.0	722500.000	1000000.0	[CUDA memcpy DtoH]

Shared Memory Optimization ↑

Apparently, they are similar. However, if we take a look at results from Nsight-Compute:

Page: Details | Launch: 1 - 123 - conv_forward_kernel | Add Baseline | Apply Rules | Copy as Image

Current	123 - conv_forward_kernel (10000, 4, 25)x(16, 16, 1)	**Time:** 19.32 msecond	**Cycles:** 23,321,066	**Regs:** 35	**GPU:** TITAN V	**SM Frequency:** 1.21 cycle/nsecond	**CC:** 7.0	**Process:** [699] m3
Baseline	123 - conv_forward_kernel (10000, 4, 25)x(16, 16, 1)	**Time:** 15.97 msecond	**Cycles:** 19,283,452	**Regs:** 32	**GPU:** TITAN V	**SM Frequency:** 1.21 cycle/nsecond	**CC:** 7.0	**Process:** [558] m3

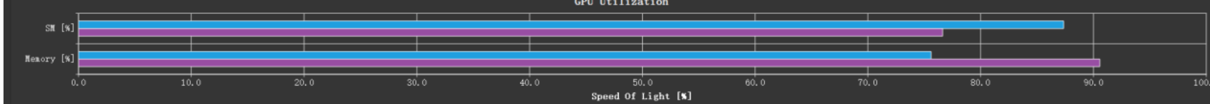
GPU Speed Of Light

All

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

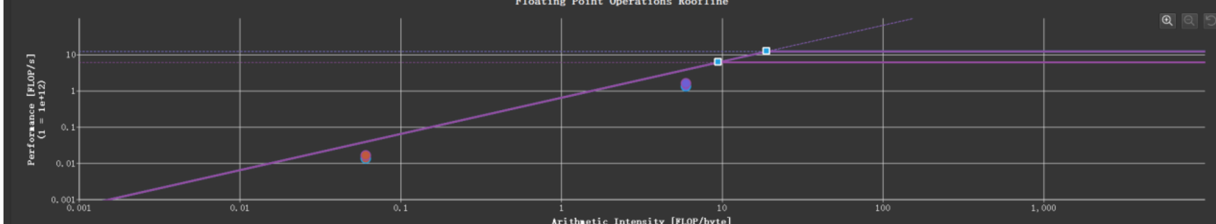
SOL SM [%]	87.36 (+14.00%)	Duration [msecond]	19.32 (+20.94%)
SOL Memory [%]	75.60 (-16.52%)	Elapsed Cycles [cycle]	23,321,066 (+20.94%)
SOL L1/TEX Cache [%]	75.60 (-16.54%)	SM Active Cycles [cycle]	23,320,259.94 (+20.96%)
SOL L2 Cache [%]	10.29 (-17.22%)	SM Frequency [cycle/nsecond]	1.21 (-0.00%)
SOL DRAM [%]	33.22 (-17.31%)	DRAM Frequency [cycle/usecond]	850.30 (-0.00%)

GPU Utilization



SOL SM Breakdown		SOL Memory Breakdown	
SOL SM: Issue Active [%]	87.36 (+14.00%)	SOL L1: Data Pipe Lsu Wavefronts [%]	75.60 (-16.52%)
SOL SM: Inst Executed [%]	87.36 (+14.01%)	SOL L1: Lsuin Requests [%]	51.14 (-10.36%)
SOL SM: Pipe Fma Cycles Active [%]	56.59 (+14.87%)	SOL L1: Lsu Writeback Active [%]	45.08 (-34.12%)
SOL SM: Pipe Alu Cycles Active [%]	52.96 (-4.56%)	SOL GPU: Dram Throughput [%]	33.22 (-17.31%)
SOL SM: Inst Executed Pipe Lsu [%]	52.42 (-8.11%)	SOL L1: Data Bank Reads [%]	11.50 (-13.97%)
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	47.81 (+163.41%)	SOL L2: T Sectors [%]	10.29 (-17.22%)
SOL SM: Mio Inst Issued [%]	26.85 (-8.35%)	SOL L2: D Sectors Fill Device [%]	9.34 (-17.06%)
SOL SM: Mio2rf Writeback Active [%]	22.96 (-19.16%)	SOL L2: Xbar2lts Cycles Active [%]	7.56 (-17.21%)
SOL SM: Inst Executed Pipe Xu [%]	4.29 (-17.31%)	SOL L2: Lts2xbar Cycles Active [%]	7.27 (-17.28%)
SOL SM: Inst Executed Pipe Adu [%]	2.57 (-17.31%)	SOL L2: T Tag Requests [%]	6.05 (-17.24%)
SOL SM: Mio Pq Read Cycles Active [%]	1.90 (+211.72%)	SOL L2: D Sectors [%]	5.66 (-17.13%)
SOL SM: Mio Pq Write Cycles Active [%]	1.82 (+251.42%)	SOL L1: M L1tex2xbar Req Cycles Active [%]	4.30 (-17.47%)
SOL SM: Pipe Fp64 Cycles Active [%]	0.43 (-17.31%)	SOL L1: M Xbar2l1tex Read Sectors [%]	4.13 (-17.55%)
SOL SM: Pipe Shared Cycles Active [%]	0.43 (-17.31%)	SOL L1: Data Bank Writes [%]	1.18 (+32.92%)
SOL IDC: Request Cycles Active [%]	0 (+0.00%)	SOL L1: Texin Sm2tex Req Cycles Active [%]	0.16 (+1,033,557.51%)
SOL SM: Inst Executed Pipe Fp16 [%]	0 (+0.00%)	SOL L1: F Wavefronts [%]	0.00 (-17.31%)
SOL SM: Inst Executed Pipe Ipa [%]	0 (+0.00%)	SOL L1: Data Pipe Tex Wavefronts [%]	0 (+0.00%)
SOL SM: Inst Executed Pipe Tex [%]	0 (+0.00%)	SOL L1: Tex Writeback Active [%]	0 (+0.00%)
SOL SM: Pipe Tensor Cycles Active [%]	0 (+0.00%)	SOL L2: D Atomic Input Cycles Active [%]	0 (+0.00%)
		SOL L2: D Sectors Fill Sysmem [%]	0 (+0.00%)

Floating Point Operations Roofline



Recommendations

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

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 close to 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]	3.49 (+14.01%)	SM Busy [%]	87.36 (+13.98%)
Executed Ipc Active [inst/cycle]	3.49 (+13.99%)	Issue Slots Busy [%]	87.36 (+13.98%)
Issued Ipc Active [inst/cycle]	3.49 (+13.98%)		

Memory Workload Analysis

All

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. Deprecated UI elements for backwards compatibility.

Memory Throughput [Gbyte/second]	216.96 (-17.32%)	Mem Busy [%]	75.60 (-16.52%)
L1/TEX Hit Rate [%]	19.66 (-79.50%)	Max Bandwidth [%]	51.14 (-25.27%)
L2 Hit Rate [%]	36.96 (-0.35%)	Mem Pipes Busy [%]	52.42 (-8.11%)

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]	11.74 (-15.12%)	No Eligible [%]	12.63 (-45.92%)
Eligible Warps Per Scheduler [warp]	2.87 (-28.18%)	One or More Eligible [%]	87.37 (+14.00%)
Issued Warp Per Scheduler	0.87 (+14.00%)		

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]	13.44 (-25.54%)	Avg. Active Threads Per Warp	31.72 (-0.88%)
Warp Cycles Per Executed Instruction [cycle]	13.44 (-25.55%)	Avg. Not Predicated Off Threads Per Warp	28.87 (-0.95%)

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]	6,519,000,000 (+37.88%)	Avg. Executed Instructions Per Scheduler [inst]	20,371,875 (+37.88%)
Issued Instructions [inst]	6,519,117,296 (+37.87%)	Avg. Issued Instructions Per Scheduler [inst]	20,372,241.55 (+37.87%)

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	1,000,000 (+0.00%)	Registers Per Thread [register/thread]	35 (+9.38%)
Block Size	256 (+0.00%)	Static Shared Memory Per Block [Kbyte/block]	2.13 (+inf%)
Threads [thread]	256,000,000 (+0.00%)	Dynamic Shared Memory Per Block [byte/block]	0 (+0.00%)
Waves Per SM	2,083.33 (+33.33%)	Driver Shared Memory Per Block [byte/block]	0 (+0.00%)
		Shared Memory Configuration Size [Kbyte]	16.38 (+inf%)

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 [%]	75 (-25.00%)	Block Limit Registers [block]	6 (-25.00%)
Theoretical Active Warps per SM [warp]	48 (-25.00%)	Block Limit Shared Mem [block]	42 (+31.25%)
Achieved Occupancy [%]	73.53 (-15.12%)	Block Limit Warps [block]	8 (+0.00%)
Achieved Active Warps Per SM [warp]	47.06 (-15.12%)	Block Limit SM [block]	32 (+0.00%)

Source Counters ⚠

All

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.

It's obvious that global memory throughput has been reduced about 17.32%, and correspondingly SM throughput has been increased. This change can certainly improve GPU performance, since the global memory bandwidth is limited.

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

   Mainly Textbook chapter 16.

2. **Optimization 2: *Streams***

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

      I chose Streams to overlap computation with data transfer. I choose this because as far as I know, this is the most potential way to reduce execution time significantly, since memory copy operations takes a lot of time.

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

I leveraged cudaStream_t classes to create nStreams cuda streams. This hyper parameter nStreams can be tuned in practice. Then I use cudaMemcpyAsync() to copy memory contents asynchronously and merge it with nStreams parts of kernel execution. All these merges can be done with cuda streams. Beyond that, I also overlap kernel execution with output memory copy asynchronously. All the code reference can be found on [How to Overlap Data Transfers in CUDA C/C++ | NVIDIA Developer Blog]. *The optimization doesn't synergize with previous optimizations.*

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	*0.001077 ms*	*0.001201 ms*	*0m1.154s*	*0.86*
1000	*0.001233 ms*	*0.001186 ms*	*0m10.337s*	*0.886*
10000	*0.003007 ms*	*0.001301 ms*	*1m45.721s*	*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).

Yes, it improved performance a lot. Let's look at the results from nsys.

CUDA API Statistics (nanoseconds)

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
85.1	1168393870	66	17702937.4	8895	43442016	cudaMemcpyAsync
13.6	186729720	8	23341215.0	77242	183299057	cudaMalloc
0.7	10286849	36	285745.8	4147	9926407	cudaLaunchKernel
0.3	3667023	2	1833511.5	14492	3652531	cudaMemcpy
0.3	3520463	8	440057.9	65264	1079808	cudaFree
0.0	169053	32	5282.9	1316	75246	cudaStreamCreate
0.0	158677	8	19834.6	1893	115068	cudaDeviceSynchronize
0.0	124683	32	3896.3	1799	20814	cudaStreamDestroy

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	78703226	32	2459475.8	1018081	3908840	conv_forward_kernel
0.0	2976	2	1488.0	1440	1536	do_not_remove_this_kernel
0.0	2784	2	1392.0	1376	1408	prefn_marker_kernel

CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
93.3	1061203476	32	33162608.6	27101751	42764135	[CUDA memcpy DtoH]
6.7	76237228	36	2117700.8	1536	2530404	[CUDA memcpy HtoD]

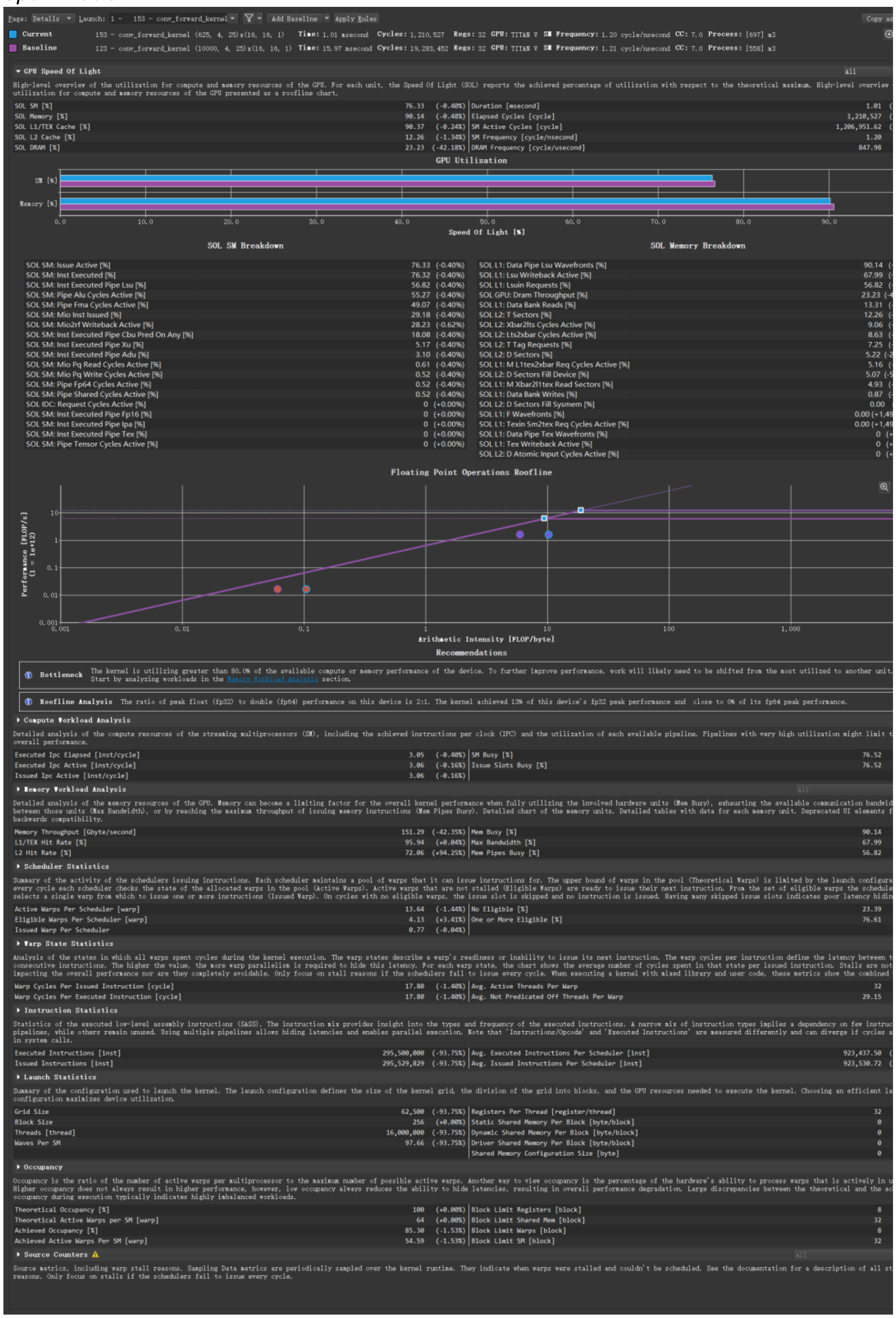
CUDA Memory Operation Statistics (KiB)

Total	Operations	Average	Minimum	Maximum	Name
1722500.0	32	53828.0	45156.250	62500.0	[CUDA memcpy DtoH]
538919.0	36	14970.0	0.004	18056.0	[CUDA memcpy HtoD]

Results that used Stream ↑

Compared with baseline, cudaMemcpyAsync() dominates the overall time consuming since we replace the normal cudaMemcpy() with it. Also, it's not hard to find out that the time used by CUDA memcpy HtoD is reduced dramatically while DtoH remains *roughly the same. It’s largely because we overlap the output with input and kernel execution.*

This is the result of Nsight-Compute of forward kernel call one of the Streams of the optimization:



Since the kernel part remains still, the SM memory condition remains the same. However, the global memory throughput has been reduced dramatically by about 42.35%. This is because the bulk memory copy operations are now distributed over time by cuda streams. From the very top of this picture, we can also find that one of cuda streams only process a small part of data set, not 10000 images anymore. The scheduling of data transportation and kernel execution indeed improves performance.

Nonetheless, I think the OP time measure in the output is not accurate. From the document, Op Time is time between the last cudaMemcpy call before the first kernel call and the first cudaMemcpy after the last kernel call. However, it measures this time by simply measure the execution time of new-forward.cu->conv_forward_gpu() function. So, it assumes that we all implement this function properly. Unfortunately, to implement CUDA Stream boosted optimization, we have to write the memory copy code and the kernel calls together to merge them. This makes new-forward.cu->conv_forward_gpu() useless and I simply returns once the program enters it. This will make the OP time measure meaningless. Yet, comparing Layer times still proves that Streams method improved performance.

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

   CUDA stream · CUDA Little Book (gitbooks.io)
   How to Overlap Data Transfers in CUDA C/C++ | NVIDIA Developer Blog
   GPU Pro Tip: CUDA 7 Streams Simplify Concurrency | NVIDIA Developer Blog

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

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

      I chose 16-bit floating point (FP16) data arithmetic. 16-bit "half-precision" floating point types are useful in applications that can process larger datasets or gain performance by choosing to store and operate on lower-precision data.

   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 optimization works well. I only leveraged FP16 data arithmetic but not storage. So based on the baseline method, I only cast the input X values and kernel values into half-precision representation, then do the computation, then cast them back to 32-bit floating point values and stored them back. The optimization doesn't synergize with previous optimizations.

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	*0.239754 ms*	*0.850697 ms*	*0m1.169s*	*0.86*
1000	*2.10516 ms*	*8.22387 ms*	*0m10.344s*	*0.887*
10000	*20.7124 ms*	*81.9974 ms*	*1m37.546s*	*0.8716*

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 nsys results of this optimization is similar to the baseline.

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

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
78.5	1575841657	10	157584165.7	15022	603009293	cudaMemcpy
9.8	196049067	8	24506133.4	74468	193653203	cudaMalloc
5.8	115602581	8	14450322.6	69378	66851279	cudaFree
5.2	103575808	8	12946976.0	1552	81840786	cudaDeviceSynchronize
0.8	15617410	6	2602901.7	23104	15456344	cudaLaunchKernel

Generating CUDA Kernel Statistics...

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

Time(%)	Total Time	Instances	Average	Minimum	Maximum	Name
100.0	102474157	2	51237078.5	20643336	81830821	conv_forward_kernel
0.0	2784	2	1392.0	1376	1408	do_not_remove_this_kernel
0.0	2687	2	1343.5	1311	1376	prefn_marker_kernel

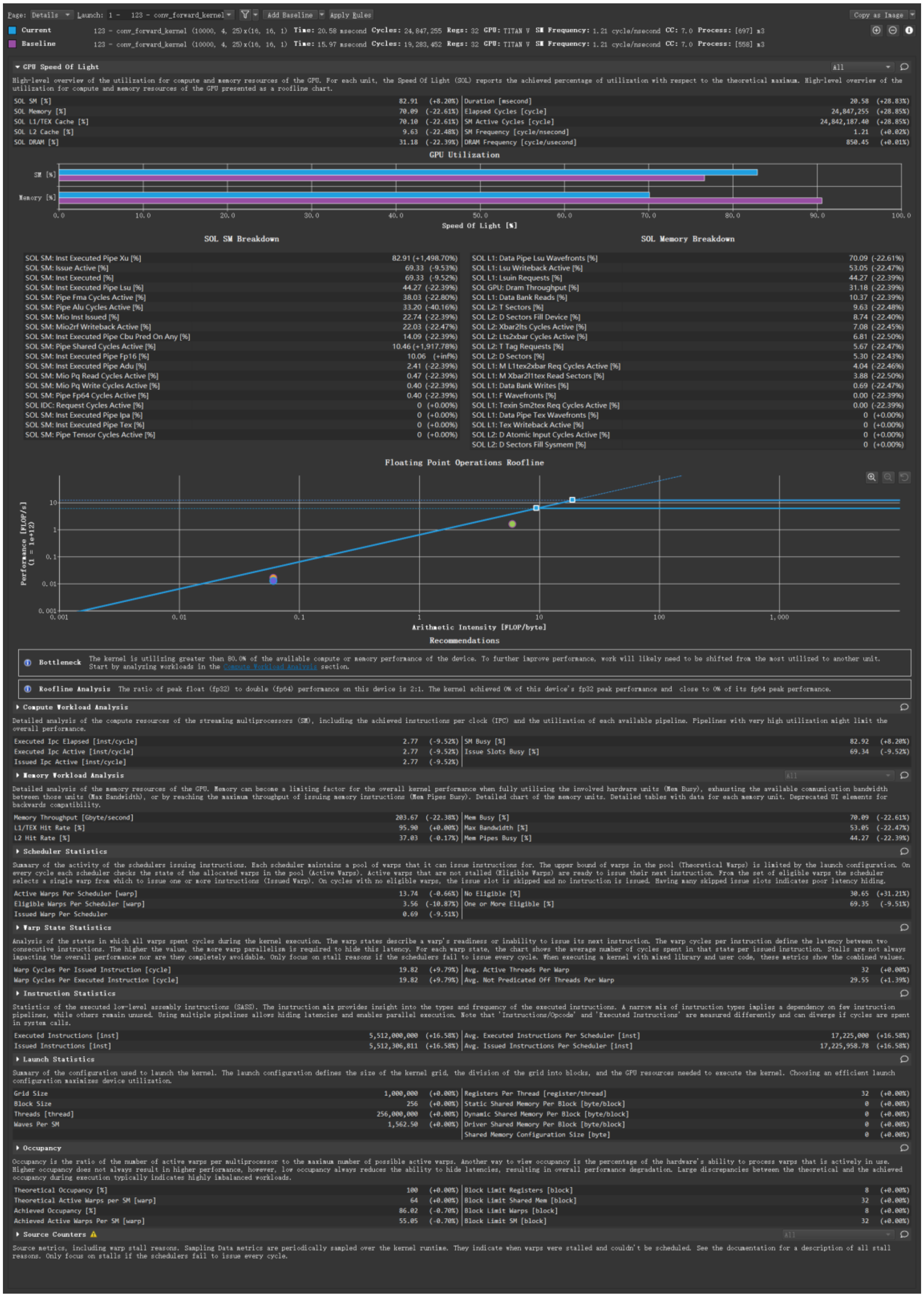
CUDA Memory Operation Statistics (nanoseconds)

Time(%)	Total Time	Operations	Average	Minimum	Maximum	Name
66.7	1047597210	2	523798605.0	445479694	602117516	[CUDA memcpy DtoH]
33.3	522153882	8	65269235.2	1184	250909320	[CUDA memcpy HtoD]

CUDA Memory Operation Statistics (KiB)

Total	Operations	Average	Minimum	Maximum	Name
2261419.0	8	282677.0	0.004	1000000.0	[CUDA memcpy HtoD]
1722500.0	2	861250.0	722500.000	1000000.0	[CUDA memcpy DtoH]

This is the results comparison of Nsight-Compute ↓.



The biggest difference is that the global memory throughput is decreased about 22.38%. In fact, I'm not 100% sure why this would happen, maybe casting from 16-bit floating point to 32-bit can save some space or time when assign the results back to output pointer. Also, in the middle it says "The kernel achieved 0% of this device's fp32

peak performance and close to 0% of its fp64 peak performance", which means we were indeed 100% utilizing FP16 arithmetic. Beyond that, there is a measure called "Compute Workload Analysis" showing that the computation overhead is actually reduced by a little.
In conclusion, this optimization surely improves computation performance.

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

New Features in CUDA 7.5 | NVIDIA Developer Blog
Support FP16 in CUDA · Issue #699 · apache/tvm (github.com)
c++ - How to allocate cuda half-precision arrays correctly? - Stack Overflow