Name: *Haozhe Chen*
NetID: *Haozhe3*
Section: *ZJ1*

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.

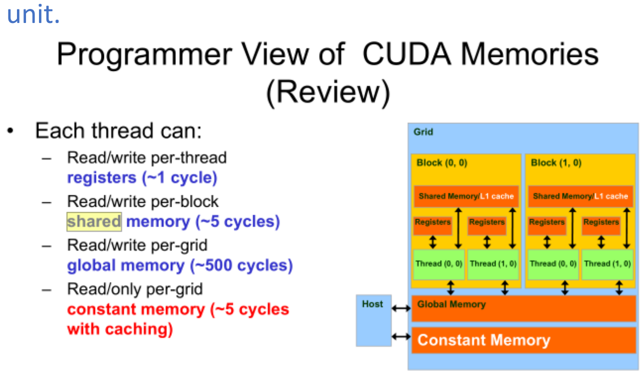
Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.247644*	*1.00156*	*0m1.026s*	*0.86*
1000	*2.21488*	*8.90458*	*0m9.797s*	*0.886*
10000	*21.7086*	*86.3012*	*1m37.050s*	*0.8714*

1. **Optimization 1:** ***Weight matrix (kernel values) in constant memory (1 point)***

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

   Based on the baseline, I use constant memory to store weight matrix in this optimization. I choose it since all the thread need to use weight matrix and it is read-only. By storing it to constant memory, global memory IO can be reduced.

   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?
   In this optimization, I copy the weight matrix to constant memory by using ***cudaMemcpyToSymbol***. All the thread can quickly fetch the data from the constant memory instead of global memory, since the former is "closer" to the GPU computing unit.



   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).

baseline

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.247644*	*1.00156*	*0m1.026s*	*0.86*
1000	*2.21488*	*8.90458*	*0m9.797s*	*0.886*
10000	*21.7086*	*86.3012*	*1m37.050s*	*0.8714*

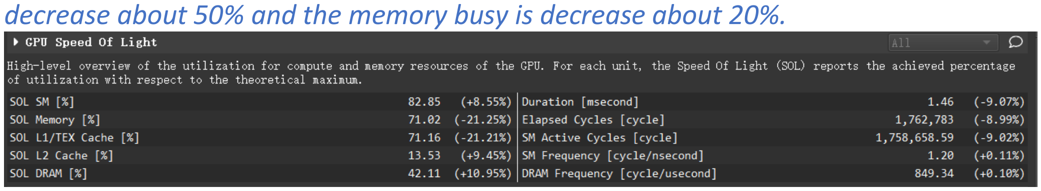
Weight matrix (kernel values) in constant memory (1 point)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.180341*	*0.586121*	*0m1.007s*	*0.86*
1000	*1.50042*	*5.66141*	*0m9.542s*	*0.886*
10000	*14.5644*	*57.1704*	*1m35.073s*	*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, both the op-time and the "memory busy" decreased, since the less global memory data IO is required during running threads.

The op-time is show above in the table and the memory usage is showed form analysis-file of nv-nsight-cu-cli. The data IO between unified cache and global is decrease about 50% and the memory busy is decrease about 20%.

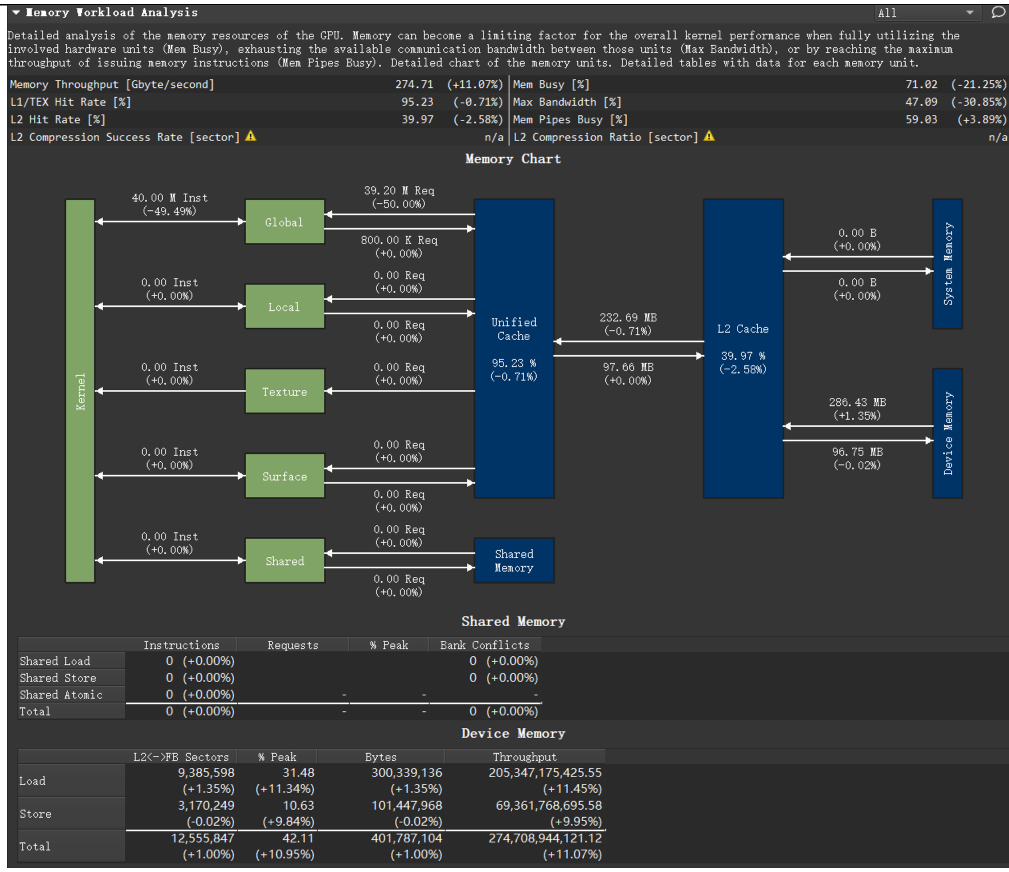


GPU Speed Of Light

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.

SOL SM [%]	82.85	(+8.55%)	Duration [msecond]	1.46	(-9.07%)
SOL Memory [%]	71.02	(-21.25%)	Elapsed Cycles [cycle]	1,762,783	(-8.99%)
SOL L1/TEX Cache [%]	71.16	(-21.21%)	SM Active Cycles [cycle]	1,758,658.59	(-9.02%)
SOL L2 Cache [%]	13.53	(+9.45%)	SM Frequency [cycle/nsecond]	1.20	(+0.11%)
SOL DRAM [%]	42.11	(+10.95%)	DRAM Frequency [cycle/usecond]	849.34	(+0.10%)

GPU SOL



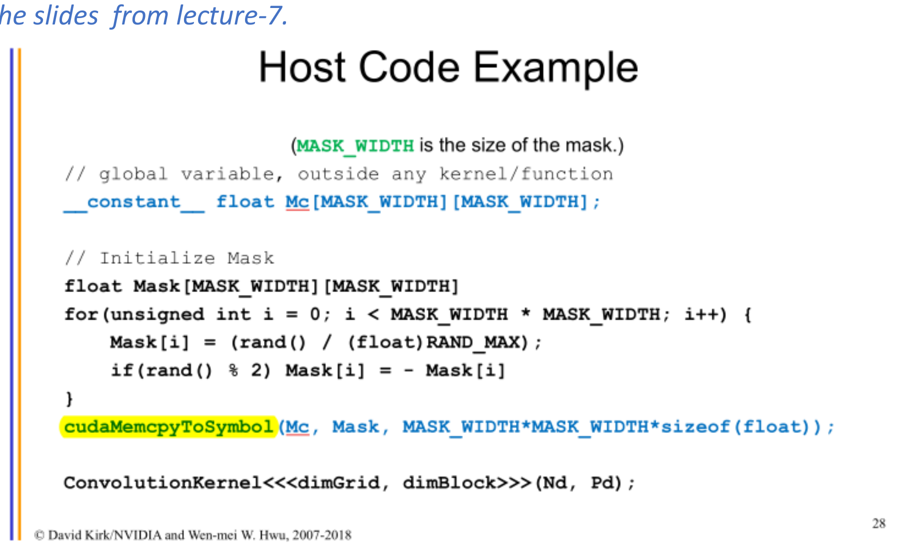
	Instructions	Requests	% Peak	Bank Conflicts
Shared Load	0 (+0.00%)			0 (+0.00%)
Shared Store	0 (+0.00%)			0 (+0.00%)
Shared Atomic	0 (+0.00%)	-	-	-
Total	0 (+0.00%)	-	-	0 (+0.00%)

	L2<->FB Sectors	% Peak	Bytes	Throughput
Load	9,385,598 (+1.35%)	31.48 (+11.34%)	300,339,136 (+1.35%)	205,347,175,425.55 (+11.45%)
Store	3,170,249 (-0.02%)	10.63 (+9.84%)	101,447,968 (-0.02%)	69,361,768,695.58 (+9.95%)
Total	12,555,847 (+1.00%)	42.11 (+10.95%)	401,787,104 (+1.00%)	274,708,944,121.12 (+11.07%)

Memory workload analysis

e. references did you use when implementing this technique?

The slides from lecture-7.



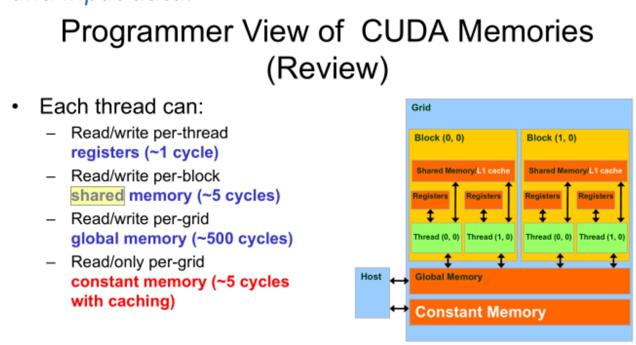
2. **Optimization 2: *Tiled shared memory convolution (2 points)***

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

Tiled shared memory is chosen in this optimization. It can also reduce the time of data transfer.

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?

Data in tiled shared memory can be fetch quicker than the global data as shown in slide below. Therefore, store data into shared memory and then fetch will reduce the data transfer cost. Considering the convolution computation condition, the input data X is more suitable for this optimization. Along with the constant memory optimization on weight matrix. The result will be better. But to get more perceptual intuition, this optimization is based on baseline and shared tiled memory is applied on both weight and input data.



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).

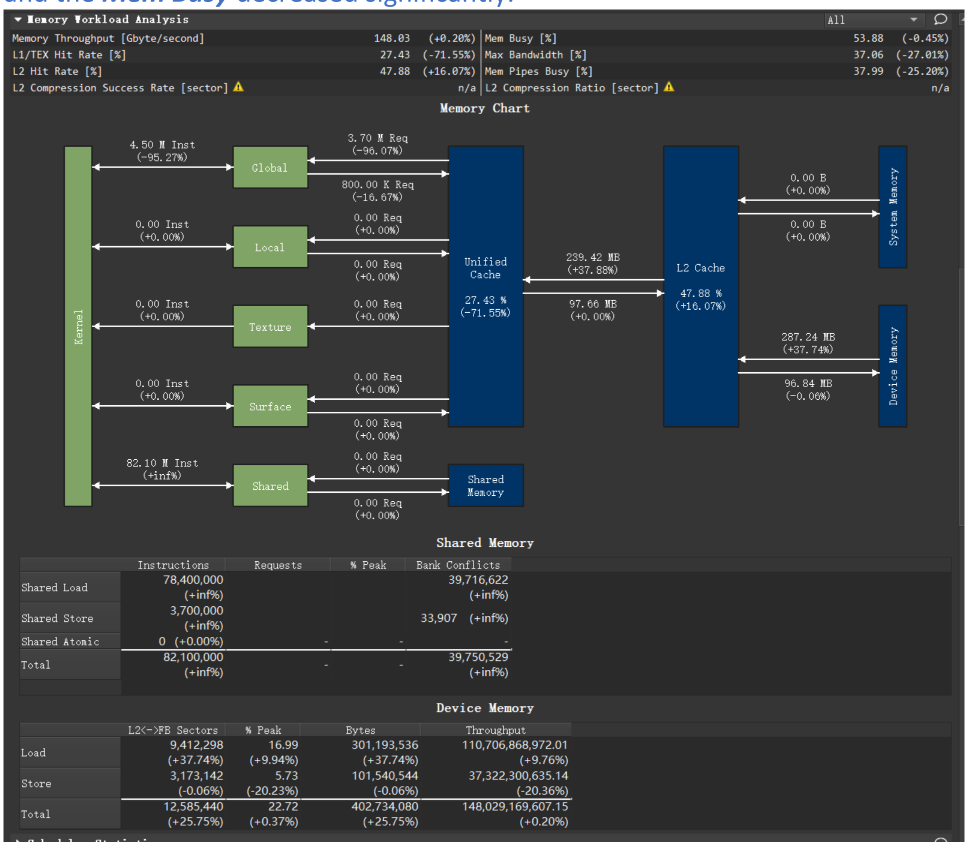
baseline

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.247644*	*1.00156*	*0m1.026s*	*0.86*
1000	*2.21488*	*8.90458*	*0m9.797s*	*0.886*
10000	*21.7086*	*86.3012*	*1m37.050s*	*0.8714*

Tiled shared memory convolution

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.281618*	*1.32705*	*0m1.017s*	*0.86*
1000	*2.82856*	*13.5985*	*0m9.559s*	*0.886*
10000	*26.9448*	*135.485*	*1m35.518s*	*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).
	From the optime shown in table above, it did not successful improve the time cost, for both layers, the kernel spend more time to do the computation. However, from the Nsight-Compute visualization it can be found that shared memory does be used and the ***Mem Busy*** decreased significantly.



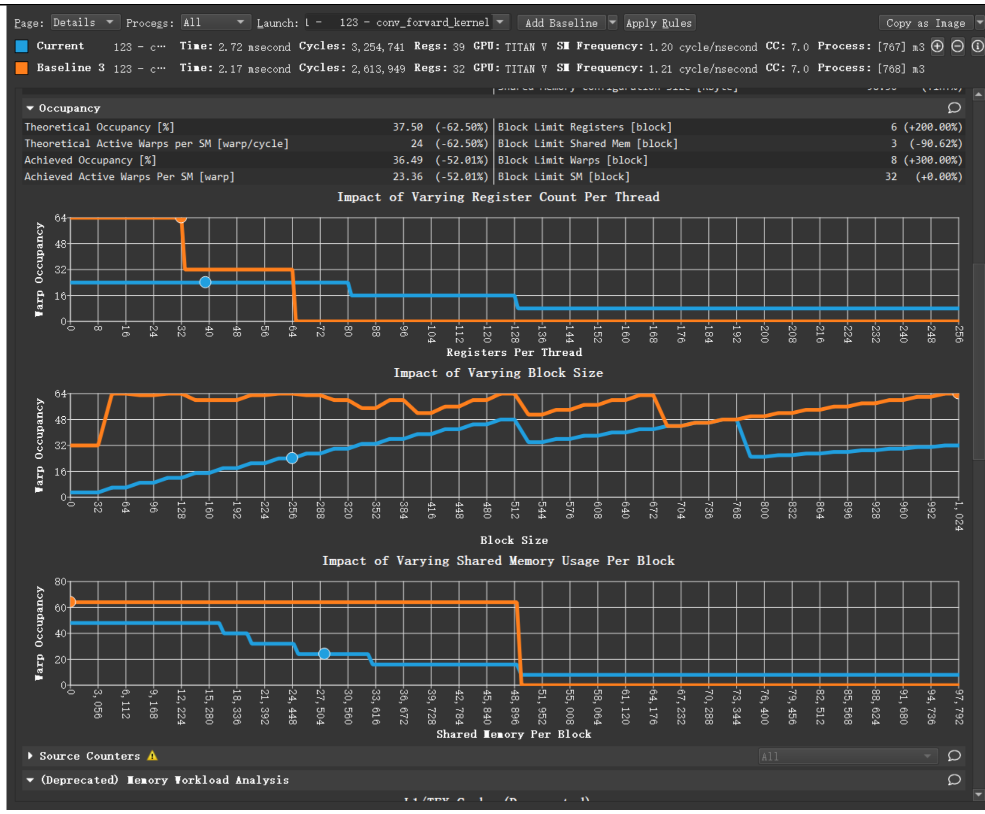
	Instructions	Requests	% Peak	Bank Conflicts
Shared Load	78,400,000 (+inf%)			39,716,622 (+inf%)
Shared Store	3,700,000 (+inf%)			33,907 (+inf%)
Shared Atomic	0 (+0.00%)	-	-	-
Total	82,100,000 (+inf%)	-	-	39,750,529 (+inf%)

Device Memory

	L2<->FB Sectors	% Peak	Bytes	Throughput
Load	9,412,298 (+37.74%)	16.99 (+9.94%)	301,193,536 (+37.74%)	110,706,868,972.01 (+9.76%)
Store	3,173,142 (-0.06%)	5.73 (-20.23%)	101,540,544 (-0.06%)	37,322,300,635.14 (-20.36%)
Total	12,585,440 (+25.75%)	22.72 (+0.37%)	402,734,080 (+25.75%)	148,029,169,607.15 (+0.20%)

Memory workload analysis

One of the possible reasons for the time cost is that, each kernel need to copy data into shared memory first and then do the computation. There is a __syncthreads between the memory copies and the computation. The tile with cannot just fit the size of the input and the weight. As a result, more divergence happen in copy memory, especially the coping input step in the second layer.



Occupancy, which indicates the divergence

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

Slides from lecture-8.

```
__global__ void convolution_1D_tiled_kernel float *N, float *P, int Width) {

  int I = blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ float N_ds[TILE_SIZE + MASK_WIDTH - 1];
  int radius = MASK_WIDTH / 2;
  int start = i - radius;

  if (0 <= start && Width > start) {       // all threads
    N_ds[threadIdx.x] = N[start];
  else
    N_ds[threadIdx.x] = 0.0f;

  if (MASK_WIDTH - 1 > threadIdx.x) {      // some threads
    start += TILE_SIZE;
    if (Width > start) {
      N_ds[threadIdx.x + TILE_SIZE] = N[start];
    else
      N_ds[threadIdx.x + TILE_SIZE] = 0.0f;
  }

  __syncthreads();

  float Pvalue = 0.0f;
  for (int j = 0; MASK_WIDTH > j; j++) {
    Pvalue += N_ds[threadIdx.x + j] * Mc[j];
  }
  P[i] = Pvalue;
}
```

Alt. Strategy 1

3. **Optimization 3:** ***Fixed point (FP16) arithmetic. (4 point)***

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

Fixed point (FP16) arithmetic is chosen to improve the FP throughput of SM.

Problem Solving:

- If FP32 is high, consider using FP16 instead.
  - In theory, a 2X speedup is possible due solely to pipeline width.
  - In theory, a 3X speedup is possible by perfectly balancing FP32 and FP16.

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?

Comparing to the float type data, the time of memory IO (transfer) of FP16 type data is smaller. The memory size the FP16 token is also smaller. Hence, it would be helpful to change computed data type from float to FP16.

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).

baseline

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.247644*	*1.00156*	*0m1.026s*	*0.86*
1000	*2.21488*	*8.90458*	*0m9.797s*	*0.886*
10000	*21.7086*	*86.3012*	*1m37.050s*	*0.8714*

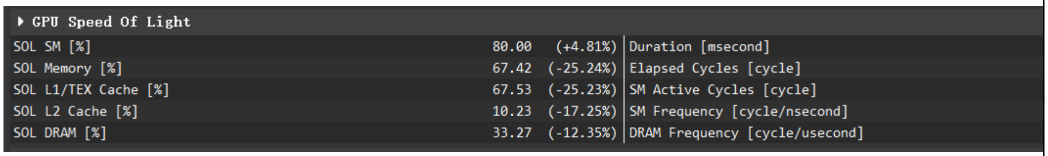
Fixed point

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.226148*	*0.834954*	*0m0.985s*	*0.86*
1000	*2.09082*	*8.1657*	*0m9.327s*	*0.887*
10000	*20.596*	*81.7839*	*1m32.731s*	*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).

Yes, it the op-time is shown in the table above. This optimization reduces the op-time for all data size tests. From the Nsight-computer visualization, it can be figured out that the FP16 compute unit was utilized, which was not in the baseline.

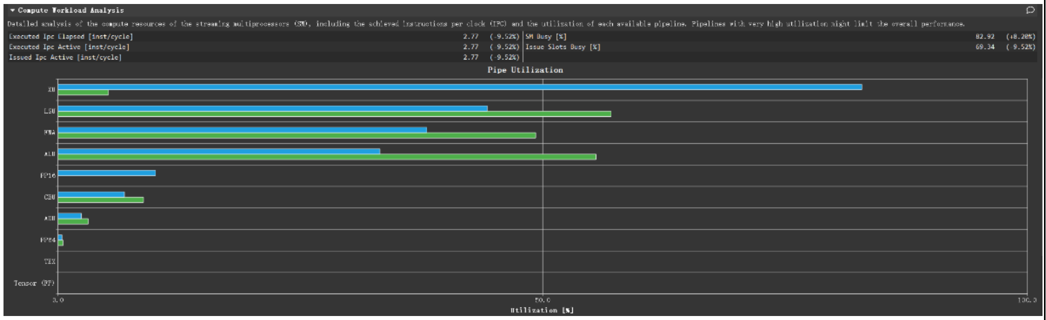
GPU SOL



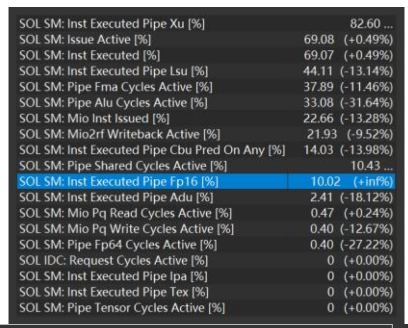
GPU Speed Of Light

SOL SM [%]	80.00	(+4.81%)	Duration [msecond]
SOL Memory [%]	67.42	(-25.24%)	Elapsed Cycles [cycle]
SOL L1/TEX Cache [%]	67.53	(-25.23%)	SM Active Cycles [cycle]
SOL L2 Cache [%]	10.23	(-17.25%)	SM Frequency [cycle/nsecond]
SOL DRAM [%]	33.27	(-12.35%)	DRAM Frequency [cycle/usecond]

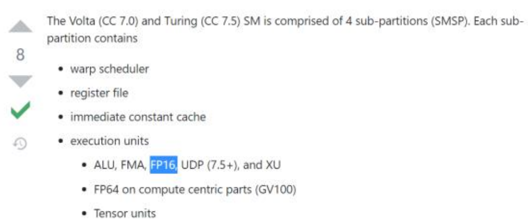
FP16 kernel is used



It worth mentioning that when I try to run the ranking, this method did not invoke FP16 but use XU(execution unit) instead, even though the code is not modified. The stackoverflow pointed that PF16 may be contained in XU unit. The evidence of using FP16 is showed in GPU SOL. However, due to its unstablity, the optimization is not used in the final version.



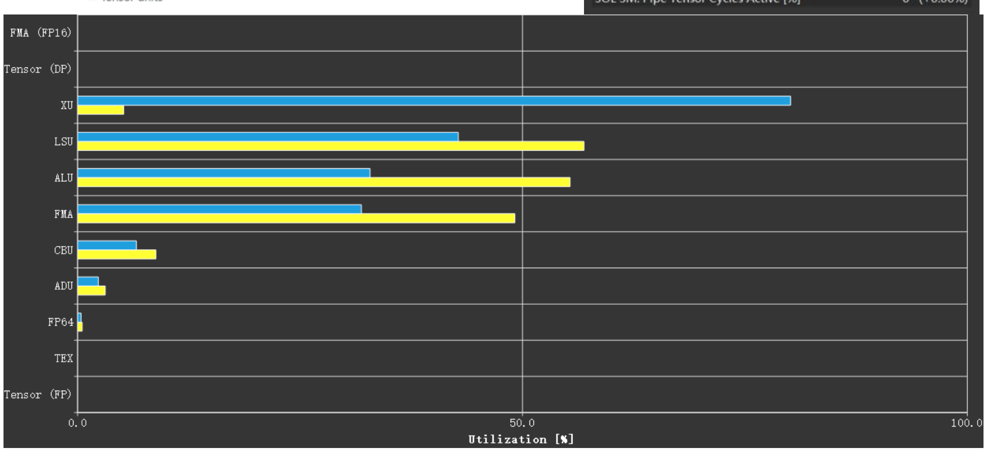
SOL SM: Inst Executed Pipe Xu [%]	82.60 ...	
SOL SM: Issue Active [%]	69.08	(+0.49%)
SOL SM: Inst Executed [%]	69.07	(+0.49%)
SOL SM: Inst Executed Pipe Lsu [%]	44.11	(-13.14%)
SOL SM: Pipe Fma Cycles Active [%]	37.89	(-11.46%)
SOL SM: Pipe Alu Cycles Active [%]	33.08	(-31.64%)
SOL SM: Mio Inst Issued [%]	22.66	(-13.28%)
SOL SM: Mio2rf Writeback Active [%]	21.93	(-9.52%)
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	14.03	(-13.98%)
SOL SM: Pipe Shared Cycles Active [%]	10.43 ...	
SOL SM: Inst Executed Pipe Fp16 [%]	10.02	(+inf%)
SOL SM: Inst Executed Pipe Adu [%]	2.41	(-18.12%)
SOL SM: Mio Pq Read Cycles Active [%]	0.47	(+0.24%)
SOL SM: Mio Pq Write Cycles Active [%]	0.40	(-12.67%)
SOL SM: Pipe Fp64 Cycles Active [%]	0.40	(-27.22%)
SOL IDC: Request Cycles Active [%]	0	(+0.00%)
SOL SM: Inst Executed Pipe Ipa [%]	0	(+0.00%)
SOL SM: Inst Executed Pipe Tex [%]	0	(+0.00%)
SOL SM: Pipe Tensor Cycles Active [%]	0	(+0.00%)



8

The Volta (CC 7.0) and Turing (CC 7.5) SM is comprised of 4 sub-partitions (SMSP). Each sub-partition contains

- warp scheduler
- register file
- immediate constant cache
- execution units
  - ALU, FMA, FP16, UDP (7.5+), and XU
  - FP64 on compute centric parts (GV100)
  - Tensor units



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

https://docs.nvidia.com/cuda/cuda-math-
api/group__CUDA__MATH____HALF__ARITHMETIC.html#group__CUDA__MATH___
_HALF__ARITHMETIC
https://stackoverflow.com/questions/61413176/interpreting-compute-workload-
analysis-in-nsight-compute

4. **Optimization 4:** ***Using Streams to overlap computation with data transfer (4 point)*** ***(Delete this section blank if you did not implement this many optimizations.)***

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

Streams is chosen to make the computation event overlap with each other. It can increase the performance of data transfer.

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?

Stream can divide a computation task into some event and execute them in a specific order. For tasks with large data size, it takes time to copy data from CPU memory to GPU memory. By using stream, it was possible to send part of data first and start computation for those arrived data. As a result, the transfer and the computation can run simultaneously. The time is therefore saved. I think it will work since from the nsys visualization, the time of data copy is an important part of kernel.

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).

baseline

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.247644*	*1.00156*	*0m1.026s*	*0.86*
1000	*2.21488*	*8.90458*	*0m9.797s*	*0.886*
10000	*21.7086*	*86.3012*	*1m37.050s*	*0.8714*

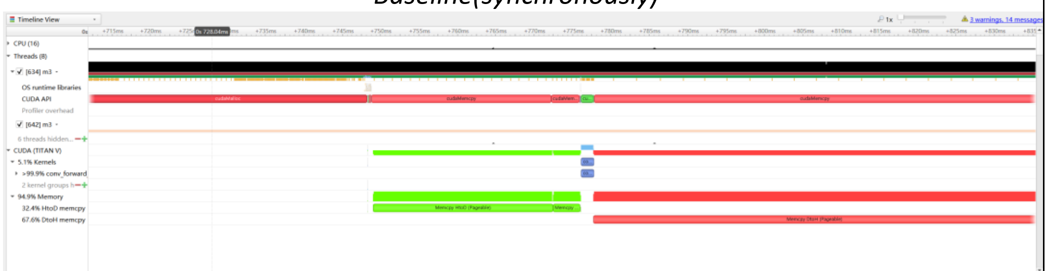
Streams

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time	Accuracy (%)
100	*0.197063*	*0.647642*	*0m1.007s*	*0.86*
1000	*1.6191*	*6.14623*	*0m9.745s*	*0.886*
10000	*16.0476*	*62.3129*	*1m37.263s*	*0.8716*

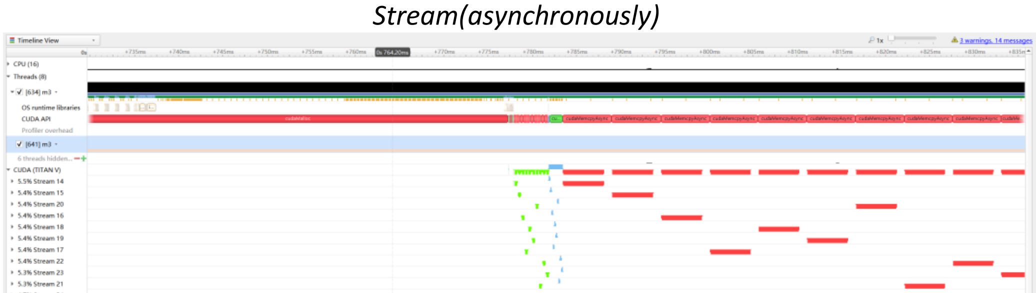
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Yes, the stream optimization reduces the op-time as shown in the table above. The data transferred between CPU and GPU just as I described in part.b. I set 10 stream for each layer and data was divided into 10 batch. The first arrived batch begins to compute first and then the second. Since this optimization is based on baseline and to special work required, it copy the data back to host asynchronously, after a "cudaDeviceSynchronize". The SOL SM and SOL memory of

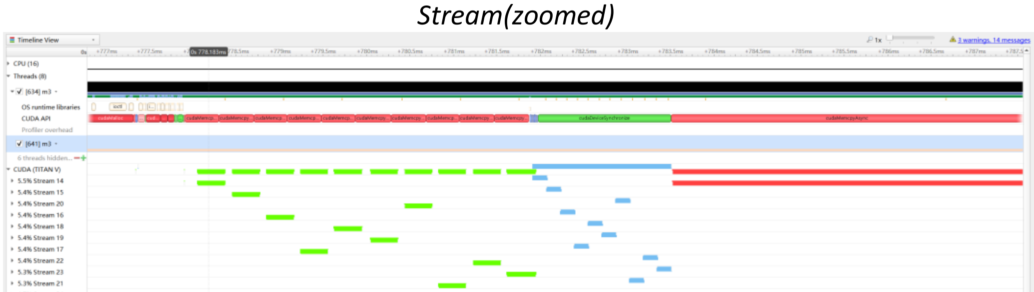
Baseline(synchronously)



Stream(asynchronously)



Stream(zoomed)



e. What references did you use when implementing this technique?
https://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf