Name: *Chiming Ni*
NetID: *chiming2*
Section: *AL2*

ECE 408/CS483 Milestone 3 Report

0. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. Note: **Do not** use batch size of 10k when you profile in *--queue rai_amd64_exclusive*. We have limited resources, so any tasks longer than 3 minutes will be killed. Your baseline M2 implementation should comfortably finish in 3 minutes with a batch size of 5k (About 1m35 seconds, with nv-nsight).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.184033ms*	*0.650947ms*	*0m1.551s*	*0.86*
1000	*1.7141ms*	*6.39048ms*	*0m10.070s*	*0.886*
5000	*8.49622ms*	*32.1391ms*	*0m48.479s*	*0.871*

1. **Optimization 1: *Shared memory matrix multiplication and input matrix unrolling (2pts)***
   Code in: opt_archive/new-forward_opt1_matrix.cu
   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      I chose shared memory matrix multiplication and input matrix unrolling technique.

      I chose it because it's the one mentioned in lecture slides and it's not very hard to implement.

   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 better reusing tiles in convolution. As mentioned in the slides, each tile was loaded into shared memory M times in the baseline implementation. But by transforming it into a matmul we can avoid reloading each tile M times. This should reduce overhead in loading data from memory.

In theory, the matmul kernel itself should be faster than baseline. However, unrolling kernel imposes extra overhead into the system, which remains unknown until we profile it.

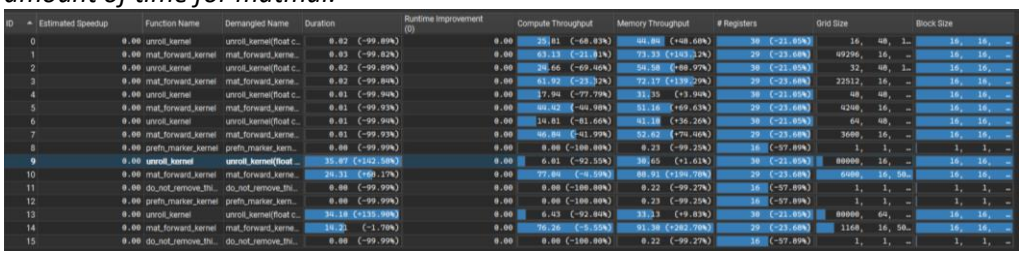
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.45912ms*	*1.26407ms*	*0m1.590s*	*0.86*
1000	*13.691ms*	*10.8761ms*	*0m10.150s*	*0.886*
5000	*67.6519ms*	*55.032ms*	*0m48.332s*	*0.871*

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

No, this optimization greatly slows down the forward convolution. As we can see from Op Times, the OP Time 1 increases almost 8 times and OP Time 2 doubled.

This is mostly because unrolling kernel takes up too much time. As we can see from the results from Nsight-Compute below, unrolling matrix takes up almost twice the amount of time for matmul.

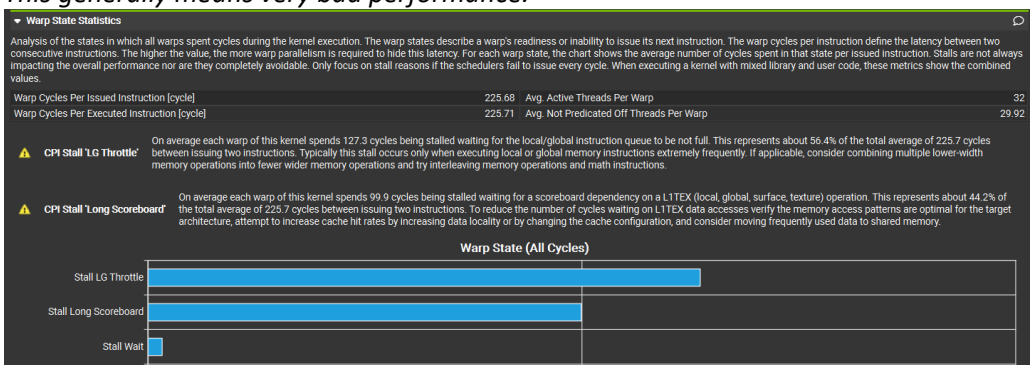


ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement (0)	Compute Throughput	Memory Throughput	# Registers	Grid Size	Block Size
0	0.00	unroll_kernel	unroll_kernel(float c...	0.02 (-99.89%)	0.00	25.81 (-68.93%)	44.84 (+48.60%)	30 (-21.05%)	16, 48, 1...	16, 16, ...
1	0.00	mat_forward_kernel	mat_forward_kerne...	0.03 (-99.82%)	0.00	63.13 (-21.01%)	73.33 (+143.12%)	29 (-23.68%)	49296, 16, ...	16, 16, ...
2	0.00	unroll_kernel	unroll_kernel(float c...	0.02 (-99.89%)	0.00	26.66 (-69.46%)	54.58 (+80.97%)	30 (-21.05%)	32, 48, 1...	16, 16, ...
3	0.00	mat_forward_kernel	mat_forward_kerne...	0.02 (-99.84%)	0.00	61.92 (-23.32%)	72.17 (+139.29%)	29 (-23.68%)	22512, 16, ...	16, 16, ...
4	0.00	unroll_kernel	unroll_kernel(float c...	0.01 (-99.94%)	0.00	17.94 (-77.79%)	31.35 (+3.94%)	30 (-21.05%)	48, 48, ...	16, 16, ...
5	0.00	mat_forward_kernel	mat_forward_kerne...	0.01 (-99.93%)	0.00	60.02 (-60.98%)	51.16 (+69.63%)	29 (-23.68%)	4248, 16, ...	16, 16, ...
6	0.00	unroll_kernel	unroll_kernel(float c...	0.01 (-99.94%)	0.00	14.81 (-81.66%)	41.18 (+36.26%)	30 (-21.05%)	64, 48, ...	16, 16, ...
7	0.00	mat_forward_kernel	mat_forward_kerne...	0.01 (-99.93%)	0.00	40.84 (-41.99%)	52.62 (+76.66%)	29 (-23.68%)	3600, 16, ...	16, 16, ...
8	0.00	prefn_marker_kernel	prefn_marker_kern...	0.00 (-99.99%)	0.00	0.00 (-100.00%)	0.23 (-99.25%)	16 (-57.89%)	1, 1, ...	1, 1, ...
9	0.00	**unroll_kernel**	**unroll_kernel(float ...**	35.07 (+142.50%)	0.00	6.01 (-92.55%)	30.65 (+1.61%)	30 (-21.05%)	80000, 16, ...	16, 16, ...
10	0.00	mat_forward_kernel	mat_forward_kerne...	24.31 (+0.17%)	0.00	77.84 (-6.59%)	80.91 (+196.70%)	29 (-23.68%)	6400, 16, 50...	16, 16, ...
11	0.00	do_not_remove_thi...	do_not_remove_thi...	0.00 (-99.99%)	0.00	0.00 (-100.00%)	0.22 (-99.27%)	16 (-57.89%)	1, 1, ...	1, 1, ...
12	0.00	prefn_marker_kernel	prefn_marker_kern...	0.00 (-99.99%)	0.00	0.00 (-100.00%)	0.23 (-99.25%)	16 (-57.89%)	1, 1, ...	1, 1, ...
13	0.00	unroll_kernel	unroll_kernel(float c...	34.10 (+135.90%)	0.00	6.43 (-92.84%)	33.13 (+9.83%)	30 (-21.05%)	80000, 64, ...	16, 16, ...
14	0.00	mat_forward_kernel	mat_forward_kerne...	14.2 (-1.70%)	0.00	76.26 (-5.55%)	91.30 (+282.70%)	29 (-23.68%)	1160, 16, 50...	16, 16, ...
15	0.00	do_not_remove_thi...	do_not_remove_thi...	0.00 (-99.99%)	0.00	0.00 (-100.00%)	0.22 (-99.27%)	16 (-57.89%)	1, 1, ...	1, 1, ...

This is expected as global memory accesses are very slow, especially when accessed in a uncoalesced way. As shown below, access to X_unroll is coalesced, but I can't find a way to access X in a coalesced way.

```
if (h < H_out && w < W_out) {
  for (int p = 0; p < K; p++) {
    for (int q = 0; q < K; q++) {
      const int h_unroll = w_base + p * K + q;
      const int w_unroll = h * W_out + w;
      X_unroll_3d(b, h_unroll, w_unroll) = X_4d(b, c, h * S + p, w * S + q);
```

This can be confirmed by profiling results from Nsight-Compute. As we can see, warps in this kernel spend significant amount of time waiting for the global instruction queue to be not full or waiting for a global instruction required for dependency to complete. This generally means very bad performance.



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

Lecture 12 Slides from Prof. Patel.

2. **Optimization 2:** ***Kernel fusion for unrolling and matrix-multiplication (3pts)***
   Code in: opt_archive/new-forward_opt2_matrix_fusion.cu
   a. Which optimization did you choose to implement and why did you choose that optimization technique.

      I implemented Kernel fusion for unrolling and matrix-multiplication.

      This is a direct improvement from the Optimization 1 since we saw how slow unrolling kernel was.

   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 loading elements from input only when needed. Compared with 1 Global Read (in unrolling), 1 Global Write (in unrolling) and 1 Global Read (in matmul) in Optimization 1, this technique only requires 1 Global Read in matmul.

      I think it should greatly improve the performance of the forward convolution as global memory access is reduced.

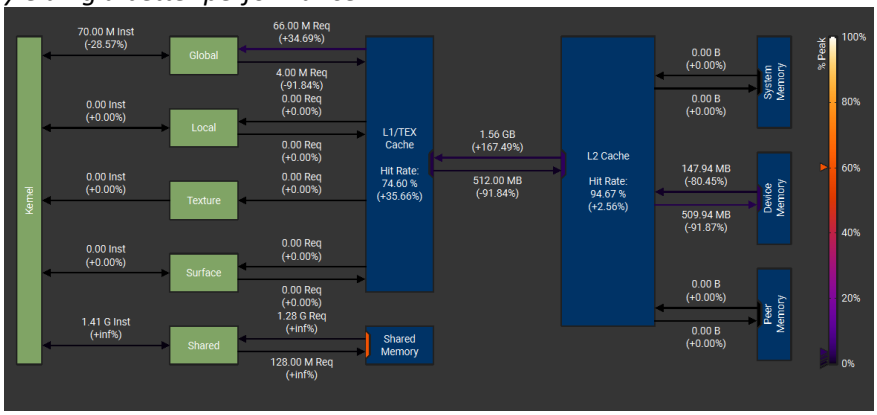
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.632654ms*	*0.35613ms*	*0m1.546s*	*0.86*
1000	*6.20707ms*	*3.28314ms*	*0m10.067s*	*0.886*
5000	*30.9783ms*	*16.4376ms*	*0m48.232s*	*0.871*

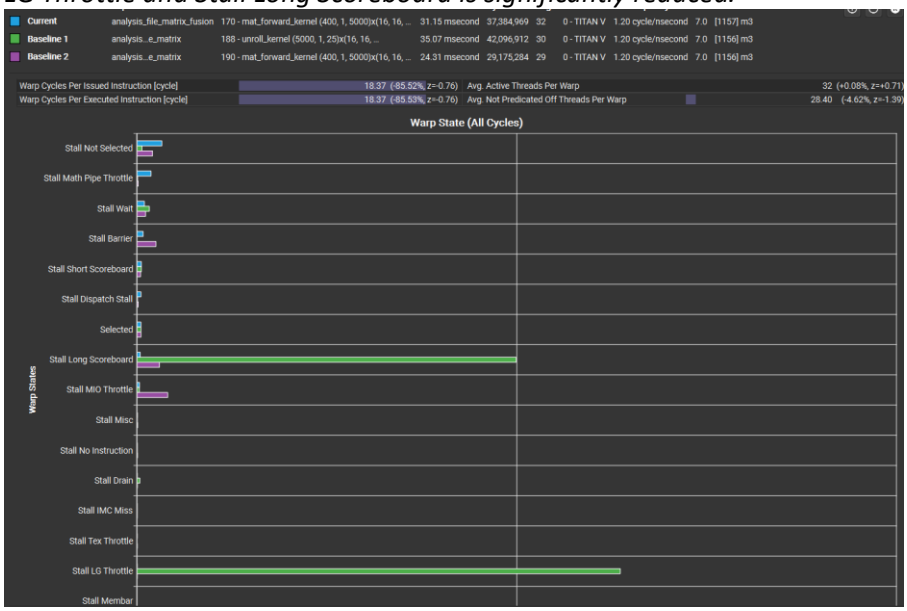
d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Yes, this optimization is much better than Optimization 1.

This should be expected as we greatly reduces the amount of direct global memory accesses (-80.45% for load and -91.87% for store). L2 accesses are much faster than global access and therefore the average warp waiting time is significantly reduced, yielding a better performance.

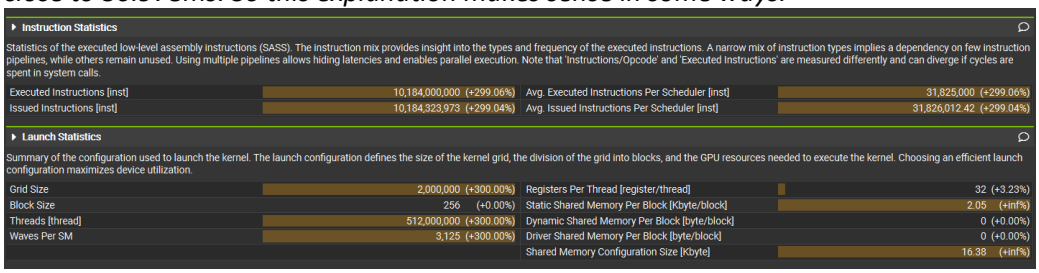


We can further verify this by looking at the warp state diagram. As we can see, Stall LG Throttle and Stall Long Scoreboard is significantly reduced.



*However, it still remains unclear why this implementation is worse than the baseline implementation (before any optimization). My guess is that this layer only has M=4, while our tiled matmul is 16*16, which means that 75% of the threads are idle, causing a large waste of compute resource.*

*As we can see from the profiling result below, compared with baseline, our new kernel almost has 4 times the instructions executed. Coincidentally, 4 * 8.496ms is pretty close to 30.978ms. So this explanation makes sense in some ways.*



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]	10,184,000,000 (+299.06%)	Avg. Executed Instructions Per Scheduler [inst]	31,825,000 (+299.06%)
Issued Instructions [inst]	10,184,323,973 (+299.04%)	Avg. Issued Instructions Per Scheduler [inst]	31,826,012.42 (+299.04%)

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	2,000,000 (+300.00%)	Registers Per Thread [register/thread]	32 (+3.23%)
Block Size	256 (+0.00%)	Static Shared Memory Per Block [Kbyte/block]	2.05 (+inf%)
Threads [thread]	512,000,000 (+300.00%)	Dynamic Shared Memory Per Block [byte/block]	0 (+0.00%)
Waves Per SM	3,125 (+300.00%)	Driver Shared Memory Per Block [byte/block]	0 (+0.00%)
		Shared Memory Configuration Size [Kbyte]	16.38 (+inf%)

We'll address this issue later in Optimization 5, where we implement a rectangular tiled matmul using Tensor Cores' 8-32-16 mode.

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

Lecture 12 Slides from Prof. Patel.

3. **Optimization 3:** ***Using Tensor Cores to speed up matrix multiplication (5pts)***
   Code in: opt_archive/new-forward_opt3_matrix_fusion_tensor.cu

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

      I implemented the Tensor Cores optimization.

      It was chosen because we're doing matmul, why not utilize some free lunch from nVidia?

   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 because nVidia designed some dedicated hardware for matrix multiplication, the Tensor Cores. These cores compute much faster than traditional CUDA matmul kernels in fixed-size matrix multiplications.

      It should improve the performance of the forward convolution, as this is supposed to be a "free lunch", there is no extra overhead imposed to the kernel.

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

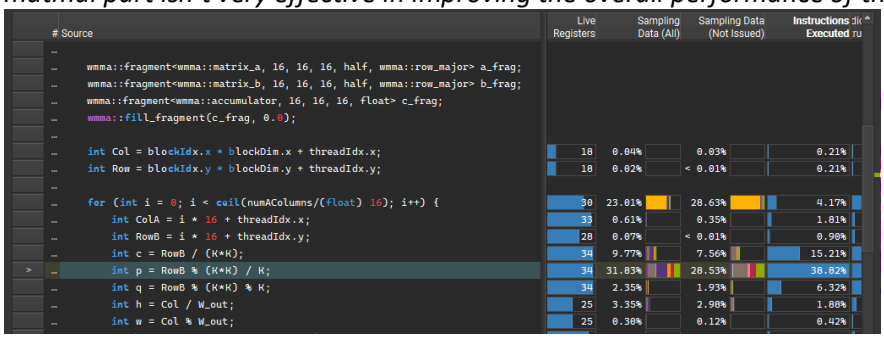
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.582063ms*	*0.307894ms*	*0m1.536s*	*0.86*
1000	*5.71425ms*	*2.91198ms*	*0m10.137s*	*0.886*
5000	*28.4993ms*	*14.5584ms*	*0m48.208s*	*0.871*

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

Yes, it improves the performance compared with Optimization 2, as we can see from comparison from Nsight-Compute below. However, the performance increase is slight.

	Report	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	analysis_file_m...x_fusion_tensor	170 - mat_forward_kernel_tensor (400, 1, 5000)x(16, 16, ...	28.48 msecond	34,392,714	38	0 - TITAN V	1.21 cycle/nsecond	7.0	[1158] m3
Baseline 1	analysis_file_matrix_fusion	170 - mat_forward_kernel (400, 1, 5000)x(16, 16, ...	31.15 msecond	37,384,969	32	0 - TITAN V	1.20 cycle/nsecond	7.0	[1157] m3

The main reason why performance improvement is so slight is possibly because computing the matrix multiplication of tiles is not the bottleneck of the kernel. As shown in profiling from Nsight-Compute below, a large portion of the execution time (40%ish) was spent on computing values of c, p, q. So, improving the performance of matmul part isn't very effective in improving the overall performance of this kernel.



This is somewhat expected since GPUs, unlike CPUs, don't have significant advantage on integer arithmetic compared with float-point arithmetic.

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

Lecture 23 Slides from Prof. Patel
CUDA C++ Programming Guide (https://docs.nvidia.com/cuda/archive/12.0.1/cuda-c-programming-guide/#warp-matrix-functions)

4. **Optimization 4:** ***Index Reusing in Fusion Kernel (Not on the list, worth 1pts, https://campuswire.com/c/G184FB646/feed/622)***
   Code in: opt_archive/new-forward_opt4_matrix_fusion_tensor_index(rowmajor).cu

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

      I implemented an optimization not on the list (I call it "index reusing"), it's verified by the TA (see link above).

      My inspiration comes from the fact that Tensor Core doesn't improve much performance. This means that the "actual" matmul is not the bottleneck, the bottleneck is somewhere else.

   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?

      *As mentioned in Optimization 3, most of the time was spent on computing c, p, q. Therefore a direct solution to this problem would be trying to reuse these values. We can observe that c, p, q values are the same for an entire row in the tile. This means that if we load elements on a same line in a thread, we only need to compute c, p, q once. Combined with the fact that Tensor Core only requires one warp (32 threads) to perform 16*16*16 matmul, we can design a loading pattern where each thread loads 8 values from a same line.*

      In theory, this should improve the performance of this kernel as one of the major bottlenecks of this kernel is addressed.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.213969ms*	*0.143532ms*	*0m1.535s*	*0.86*
1000	*2.00769ms*	*1.24454ms*	*0m10.112s*	*0.886*
5000	*10.0396ms*	*6.15833ms*	*0m48.003s*	*0.871*

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

Yes, it greatly improves the performance. We can see from the comparison with Optimization 3 that this optimization reduces more than 50% of the kernel runtime.

	Report	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	analysis_file_matri...fusion_tensor_index	170 - mat_forward_kernel_tensor (400, 1, 5000)x(16, 2, 1)	9.97 msecond	12,027,415	72	0 - TITAN V	1.21 cycle/nsecond	7.0	[1157] m3
Baseline 1	analysis_file_m...x_fusion_tensor	170 - mat_forward_kernel_tensor (400, 1, 5000)x(16, 16, ...	28.48 msecond	34,392,714	38	0 - TITAN V	1.21 cycle/nsecond	7.0	[1158] m3

This should be expected as we solve a major bottleneck of the kernel. As shown in profiling results from Nsight-Compute below, computing c, p, q now only takes around 15% of the whole execution time.

# Source	Live Registers	Sampling Data (All)	Sampling Data (Not Issued)	Instruction Execut
…				
… for (int i = 0; i < ceil(numAColumns/(float) 16); i++) {	67	2.40%	2.89%	2.11
… int ColA = i * 16 + threadIdx.x;	64	0.59%	0.47%	0.48
… int RowB = i * 16 + threadIdx.x;				
… int c = RowB / (K*K);	68	4.09%	2.42%	7.57
> … int p = RowB % (K*K) / K;	69	11.80%	7.36%	19.78
… int q = RowB % K;	70	1.31%	0.67%	3.15
… for (int j = 0; j < 8; j++) {				
… int RowA = blockIdx.y * 16 + threadIdx.y * 8 + j;	27	0.62%	0.67%	0.24
… int ColB = blockIdx.x * 16 + threadIdx.y * 8 + j;	27	0.06%	0.04%	0.31
… int h = ColB / W_out;	53	1.38%	0.46%	3.39
… int w = ColB % W_out;	58	0.53%	0.08%	2.25
… if (RowA < M && ColA < numAColumns)	68	0.18%		1.35
… tileA[threadIdx.y * 8 + j][threadIdx.x] = __float2half(mask[R	70	21.87%	23.43%	16.42
… else				
… tileA[threadIdx.y * 8 + j][threadIdx.x] = 0.0;				
… if (ColB < numBColumns && RowB < numAColumns)	70	0.82%	0.44%	3.66
… tileB[threadIdx.x][threadIdx.y * 8 + j] = __float2half(in_4d(	70	32.97%	36.33%	23.17
… else				
… tileB[threadIdx.x][threadIdx.y * 8 + j] = 0.0;				
… }				
… __syncthreads();	59	1.00%	1.41%	0.96
…				
… wmma::load_matrix_sync(a_frag, (half *) tileA, 16);	65	1.82%	2.51%	1.38
… wmma::load_matrix_sync(b_frag, (half *) tileB, 16);	67	5.21%	7.10%	1.97
… wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);	67	9.55%	9.96%	7.19
… }				

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

No references

5. **Optimization 5:** ***Multiple kernel implementations for different layer sizes (1pts)***
   Code in: opt_archive/new-forward_opt5_matrix_fusion_tensor_index_separate.cu

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

      I implemented multiple kernel implementations for different layer sizes.

      *As mentioned in Optimization 2, 16*16 is too big for M=4 (Layer 1), that's why I'm trying to solve this problem by launching different kernels.*

   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 by creating another kernel that utilizes the 8-32-16 mode provided by Tensor Core. In this way, only 50% of the threads are idle compared with 75% previously (it can't be smaller due to limitation of Tensor Core).

      It's complicated to implement the row-major loading trick in Optimization 4 when we're in 8-32-16, so I implemented a shared memory version of the index reusing trick (sounds unusual to put indices in shared memory but it actually works).

      I think this should theoretically improve the performance as less blocks are created and more of the threads get used.

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

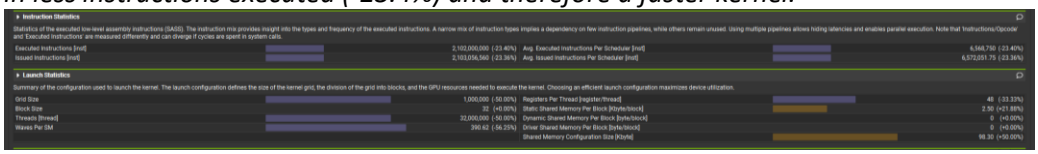
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.158265ms*	*0.147308ms*	*0m1.558s*	*0.86*
1000	*1.42254ms*	*1.24668ms*	*0m10.163s*	*0.886*
5000	*7.01001ms*	*6.16044ms*	*0m48.103s*	*0.871*

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

Yes, it successfully improves the performance in the first layer. And finally, the matmul implementation is faster than the baseline in the first layer :)

	Report	Result	Time	Cycles	Regs	GPU	SM Frequency	CC	Process
Current	analysis_file_matrix_fusion_tensor_index_separate	171 - mat_forward_kernel_tensor_8_32_16 (200, 1, 5000)x(32, 1, 1)	7.01 msecond	8,454,582	48	0 - TITAN V	1.21 cycle/nsecond	7.0	[1158] m3
Baseline 1	analysis_file_matri_fusion_tensor_index	170 - mat_forward_kernel_tensor (400, 1, 5000)x(16, 2, 1)	9.97 msecond	12,027,415	72	0 - TITAN V	1.21 cycle/nsecond	7.0	[1157] m3

This is expected as we halve the number of blocks (grid size) created, and this results in less instructions executed (-23.4%) and therefore a faster kernel.



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

No references

6. **Optimization 6:** ***Using Streams to overlap computation with data transfer (4pts)***
   Code in: opt_archive/new-forward_opt6_stream.cu

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

      I implemented the streams to overlap computation with data transfer.

      I chose this because I think up to now the computation is already fast enough, and there's little room for improvement in kernels. Therefore I decided to exploit the task parallelism provided by CUDA to further improve the performance of the entire system.

   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 by dividing the entire task into smaller chunks and then we'll be able to overlap the computation of some chunks with data transfer of other chunks.

      Though the speed of computation hasn't been improved, the entire system will run faster because computation and data transfer can happen at the same time now.

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

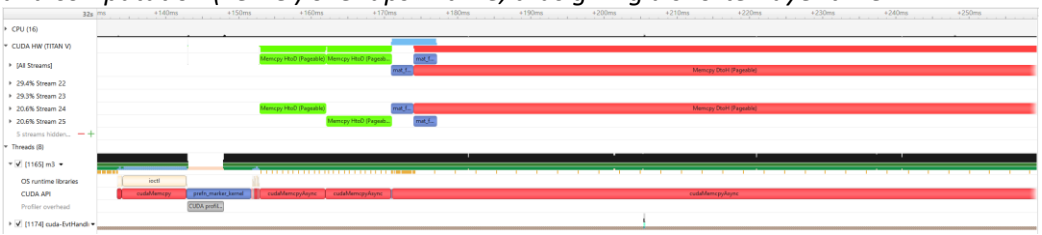
Batch Size	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	*7.14026 ms*	*5.42162 ms*	*0m1.504s*	*0.86*
1000	*118.642 ms*	*48.5748 ms*	*0m10.867s*	*0.886*
5000	*297.281 ms*	*226.066 ms*	*0m51.017s*	*0.871*
For comparison, I list layer time for Optimization 5 below (note this run is separate from previous run in Optimization 5 so total execution time will be different)				
Batch Size	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	*6.82445 ms*	*5.27564 ms*	*0m1.693s*	*0.86*
1000	*63.8219 ms*	*46.5222 ms*	*0m10.063s*	*0.886*
5000	*333.497 ms*	*232.319 ms*	*0m51.298s*	*0.871*

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

Yes, it successfully reduces the layer time for B=5000.

It doesn't optimize the performance for B=1000 and B=100. This is expected as I set seg_size to 2500, this means that no concurrency happens for B=1000 and B=100. Additionally, the program has to deal with the additional overhead imposed by launching a stream, so we may get even worse performance.

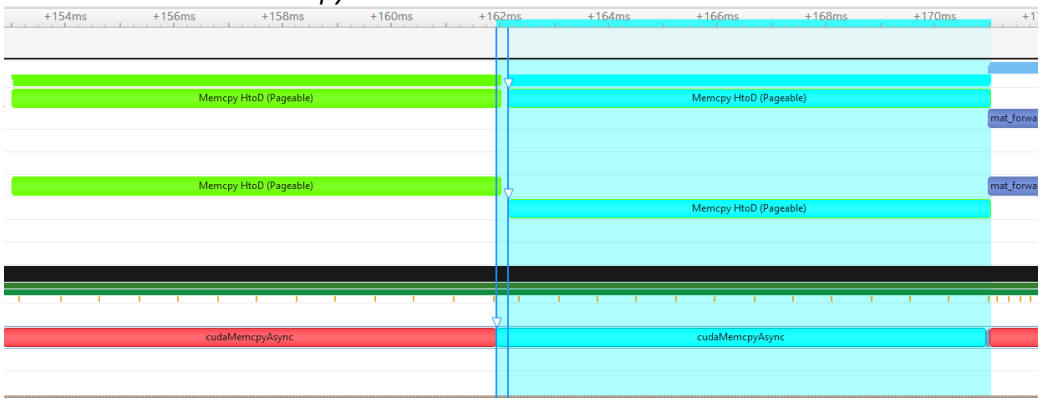
As shown in nsys profiling result below, we can see that the execution of data transfer and computation (kernel) overlaps in time, thus giving a shorter layer time.



The performance gain is pretty slight, this can also be explained by profiling results from nsys.

First, MemcpyDtoH is taking too long, much longer than MemcpyHtoD and the kernel. There's no parallelism in two copy operations in the same direction, so we still need to take tremendous amount of time to wait for MemcpyDtoH to complete.

Second, the kernel and MemcpyHtoD doesn't have much time overlap. This seems pretty strange as Stream 24 kernel is supposed to run parallelly with Stream 25 MemcpyHtoD, at least that's what we should expect from lecture slides. This unexpected behavior can be explained with the screenshot below. As we can see the cudaMemcpyAsync call on host takes more than 95% of the time, so 95% time of the copy operation is forced to be synchronous. This causes very limited time overlap between kernel and MemcpyHtoD.



Why would a cudaMemcpyAsync be forced to be synchronous? This is actually well-documented in CUDA Documentation. The CUDA Documentation states that "If pageable memory must first be staged to pinned memory, the driver may synchronize with the stream and stage the copy into pinned memory." (https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html#api-sync-behavior__memcpy-async)

Our current implementation didn't use pinned memory to store host data, therefore it has to be first copied to a pinned memory buffer, and this process has to be synchronous.

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

Lecture 22 Slides from Prof. Patel
CUDA Toolkit Documentation – API Synchronization Behavior (https://docs.nvidia.com/cuda/cuda-runtime-api/api-sync-behavior.html#api-sync-behavior__memcpy-async)

f. Code for Streaming

```
#include <cmath>
#include <iostream>
#include "gpu-new-forward.h"
#include <cuda_fp16.h>
#include <mma.h>
using namespace nvcuda;

#define NUM_STREAMS 2
constexpr int seg_size = 2500;

cudaStream_t streams[NUM_STREAMS];

__global__ void mat_forward_kernel_tensor(float *output, const float *input, const float *mask, const
int B, const int M, const int C, const int H, const int W, const int K,const int S)
```

```
{
    #define out_4d(i3, i2, i1, i0) output[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) *
(W_out) + i0]
    #define in_4d(i3, i2, i1, i0) input[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
    #define mask_4d(i3, i2, i1, i0) mask[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    const int H_out = (H - K)/S + 1;
    const int W_out = (W - K)/S + 1;
    int b = blockIdx.z;
    int numAColumns = C*K*K;
    int numBColumns = H_out*W_out;

    __shared__ half tileA[16][16];
    __shared__ half tileB[16][16];
    __shared__ float tileC[16][16];

    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;
    wmma::fill_fragment(c_frag, 0.0);

    for (int i = 0; i < ceil(numAColumns/(float) 16); i++) {
        int ColA = i * 16 + threadIdx.x;
        int RowB = i * 16 + threadIdx.x;
        int c = RowB / (K*K);
        int p = RowB % (K*K) / K;
        int q = RowB % K;
        for (int j = 0; j < 8; j++) {
            int RowA = blockIdx.y * 16 + threadIdx.y * 8 + j;
            int ColB = blockIdx.x * 16 + threadIdx.y * 8 + j;
            int h = ColB / W_out;
            int w = ColB % W_out;
            if (RowA < M && ColA < numAColumns)
                tileA[threadIdx.y * 8 + j][threadIdx.x] = __float2half(mask[RowA * numAColumns +
ColA]);
            else
                tileA[threadIdx.y * 8 + j][threadIdx.x] = __float2half(0.0f);
            if (ColB < numBColumns && RowB < numAColumns)
```

```
                tileB[threadIdx.x][threadIdx.y * 8 + j] = __float2half(in_4d(b, c, h * S + p, w * S + q));
            else
                tileB[threadIdx.x][threadIdx.y * 8 + j] = __float2half(0.0f);
        }

        wmma::load_matrix_sync(a_frag, (half *) tileA, 16);
        wmma::load_matrix_sync(b_frag, (half *) tileB, 16);
        wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
    }
    wmma::store_matrix_sync((float *) tileC, c_frag, 16, wmma::mem_row_major);
    for (int i = 0; i < 8; i++) {
        int RowC = blockIdx.y * 16 + threadIdx.y * 8 + i;
        int ColC = blockIdx.x * 16 + threadIdx.x;
        if (RowC < M && ColC < numBColumns)
            output[b * (M * numBColumns) + RowC * numBColumns + ColC] = tileC[threadIdx.y * 8 + i][threadIdx.x];
    }

    #undef out_4d
    #undef in_4d
    #undef mask_4d
}

__global__ void mat_forward_kernel_tensor_8_32_16(float *output, const float *input, const float *mask, const int B, const int M, const int C, const int H, const int W, const int K,const int S)
{
    #define out_4d(i3, i2, i1, i0) output[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) * (W_out) + i0]
    #define in_4d(i3, i2, i1, i0) input[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]
    #define mask_4d(i3, i2, i1, i0) mask[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    const int H_out = (H - K)/S + 1;
    const int W_out = (W - K)/S + 1;
    int b = blockIdx.z;
    int numAColumns = C*K*K;
    int numBColumns = H_out*W_out;

    __shared__ half tileA[8][16];
```

```
    __shared__ half tileB[16][32];
    __shared__ float tileC[8][32];
    __shared__ int c_arr[16];
    __shared__ int p_arr[16];
    __shared__ int q_arr[16];

    wmma::fragment<wmma::matrix_a, 8, 32, 16, half, wmma::row_major> a_frag;
    wmma::fragment<wmma::matrix_b, 8, 32, 16, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, 8, 32, 16, float> c_frag;
    wmma::fill_fragment(c_frag, 0.0);



    for (int i = 0; i < ceil(numAColumns/(float) 16); i++) {
        if (threadIdx.x < 16) {
            for (int j = 0; j < 8; j++) {
                int ColA = i * 16 + threadIdx.x;
                int RowA = blockIdx.y * 8 + j;
                if (RowA < M && ColA < numAColumns)
                    tileA[j][threadIdx.x] = __float2half(mask[RowA * numAColumns + ColA]);
                else
                    tileA[j][threadIdx.x] = __float2half(0.0f);
            }
            int RowB = i * 16 + threadIdx.x;
            c_arr[threadIdx.x] = RowB / (K*K);
            p_arr[threadIdx.x] = RowB % (K*K) / K;
            q_arr[threadIdx.x] = RowB % K;
        }
        int ColB = blockIdx.x * 32 + threadIdx.x;
        int h = ColB / W_out;
        int w = ColB % W_out;
        for (int j = 0; j < 16; j++) {
            int RowB = i * 16 + j;
            if (ColB < numBColumns && RowB < numAColumns)
                tileB[j][threadIdx.x] = __float2half(in_4d(b, c_arr[j], h * S + p_arr[j], w * S +
q_arr[j]));
            else
                tileB[j][threadIdx.x] = __float2half(0.0f);
        }
```

```
        wmma::load_matrix_sync(a_frag, (half *) tileA, 16);
        wmma::load_matrix_sync(b_frag, (half *) tileB, 32);
        wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
    }
    wmma::store_matrix_sync((float *) tileC, c_frag, 32, wmma::mem_row_major);
    for (int i = 0; i < 8; i++) {
        int RowC = blockIdx.y * 8 + i;
        int ColC = blockIdx.x * 32 + threadIdx.x;
        if (RowC < M && ColC < numBColumns)
            output[b * (M * numBColumns) + RowC * numBColumns + ColC] = tileC[i][threadIdx.x];
    }

    #undef out_4d
    #undef in_4d
    #undef mask_4d
}

__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output, const float *host_input,
const float *host_mask, float **device_output_ptr, float **device_input_ptr, float **device_mask_ptr,
const int B, const int M, const int C, const int H, const int W, const int K, const int S)
{
    // Allocate memory and copy over the relevant data structures to the GPU

    // We pass double pointers for you to initialize the relevant device pointers,
    //  which are passed to the other two functions.
    for (int i = 0; i < NUM_STREAMS; i++) {
        cudaStreamCreate(&streams[i]);
    }

    float *host_output_nonconst = const_cast<float *>(host_output); // This is a hack to avoid having
to change the function signature to non-const

    const int H_out = (H - K)/S + 1;
    const int W_out = (W - K)/S + 1;
    cudaMalloc((void **)device_input_ptr, B*C*H*W*sizeof(float));
    cudaMalloc((void **)device_output_ptr, B*M*H_out*W_out*sizeof(float));
    cudaMalloc((void **)device_mask_ptr, M*C*K*K*sizeof(float));

    cudaMemcpy(*device_mask_ptr, host_mask, M*C*K*K*sizeof(float), cudaMemcpyHostToDevice);
```

```
    for (int i = 0; i < B; i += seg_size * NUM_STREAMS) {
        for (int j = 0; j < NUM_STREAMS && i + j * seg_size < B; j++) {
            int size = min(seg_size, B - i - j * seg_size);
            cudaMemcpyAsync(*device_input_ptr + (i + j * seg_size) * C * H * W, host_input + (i + j *
seg_size) * C * H * W, size * C * H * W * sizeof(float), cudaMemcpyHostToDevice, streams[j]);
        }

        for (int j = 0; j < NUM_STREAMS && i + j * seg_size < B; j++) {
            int size = min(seg_size, B - i - j * seg_size);
            if (M < 16) {
                dim3 dimBlock = dim3(32, 1, 1);
                dim3 dimGrid = dim3(ceil((H_out*W_out)/(float)32), ceil(M/(float)8), size);
                mat_forward_kernel_tensor_8_32_16<<<dimGrid, dimBlock, 0,
streams[j]>>>(*device_output_ptr + (i + j * seg_size) * M * H_out * W_out, *device_input_ptr + (i + j *
seg_size) * C * H * W, *device_mask_ptr, size, M, C, H, W, K, S);
            } else {
                dim3 dimBlock = dim3(16, 2, 1);
                dim3 dimGrid = dim3(ceil((H_out*W_out)/(float)16), ceil(M/(float)16), size);
                mat_forward_kernel_tensor<<<dimGrid, dimBlock, 0, streams[j]>>>(*device_output_ptr + (i
+ j * seg_size) * M * H_out * W_out, *device_input_ptr + (i + j * seg_size) * C * H * W,
*device_mask_ptr, size, M, C, H, W, K, S);
            }
        }

        for (int j = 0; j < NUM_STREAMS && i + j * seg_size < B; j++) {
            int size = min(seg_size, B - i - j * seg_size);
            cudaMemcpyAsync(host_output_nonconst + (i + j * seg_size) * M * H_out * W_out,
*device_output_ptr + (i + j * seg_size) * M * H_out * W_out, size * M * H_out * W_out * sizeof(float),
cudaMemcpyDeviceToHost, streams[j]);
        }
    }
}


__host__ void GPUInterface::conv_forward_gpu(float *device_output, const float *device_input, const
float *device_mask, const int B, const int M, const int C, const int H, const int W, const int K, const
int S)
{
    // Set the kernel dimensions and call the kernel
```

```
}
__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, float
*device_input, float *device_mask, const int B, const int M, const int C, const int H, const int W,
const int K, const int S)
{
    cudaDeviceSynchronize();

    // Free device memory
    cudaFree(device_output);
    cudaFree(device_input);
    cudaFree(device_mask);

    for (int i = 0; i < NUM_STREAMS; i++) {
        cudaStreamDestroy(streams[i]);
    }
}
__host__ void GPUInterface::get_device_properties()
{
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);

    for(int dev = 0; dev < deviceCount; dev++)
    {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);

        std::cout<<"Device "<<dev<<" name: "<<deviceProp.name<<std::endl;
        std::cout<<"Computational capabilities: "<<deviceProp.major<<"."<<deviceProp.minor<<std::endl;
        std::cout<<"Max Global memory size: "<<deviceProp.totalGlobalMem<<std::endl;
        std::cout<<"Max Constant memory size: "<<deviceProp.totalConstMem<<std::endl;
        std::cout<<"Max Shared memory size per block: "<<deviceProp.sharedMemPerBlock<<std::endl;
        std::cout<<"Max threads per block: "<<deviceProp.maxThreadsPerBlock<<std::endl;
        std::cout<<"Max block dimensions: "<<deviceProp.maxThreadsDim[0]<<" x,
"<<deviceProp.maxThreadsDim[1]<<" y, "<<deviceProp.maxThreadsDim[2]<<" z"<<std::endl;
        std::cout<<"Max grid dimensions: "<<deviceProp.maxGridSize[0]<<" x,
"<<deviceProp.maxGridSize[1]<<" y, "<<deviceProp.maxGridSize[2]<<" z"<<std::endl;
        std::cout<<"Warp Size: "<<deviceProp.warpSize<<std::endl;
    }
}
```