Name: *Yen-Ting Liu*
NetID: *ytliu2*
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 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	*2.07946 ms*	*31.8139 ms*	*1.117 s*	*0.86*
1000	*20.7417 ms*	*322.405 ms*	*9.517 s*	*0.886*
10000	*207.877 ms*	*3242.94 ms*	*1 m 33.543 s*	*0.8714*

(build-fbba3)

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

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

   Calculation for all outputs require kernel, and it does not change during grid execution. These characteristics make kernel array a nice candidate to move into constant memory with minimal code change.

   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?

   Constant memory resides in device memory and caches in the constant cache. In case of cache hit, the resulting memory requests are served at the throughput of the constant cache. Since kernel weight will not change throughout the inference, I believe constant memory can provide some improvements, and can work well with other optimization (this is my first optimization).

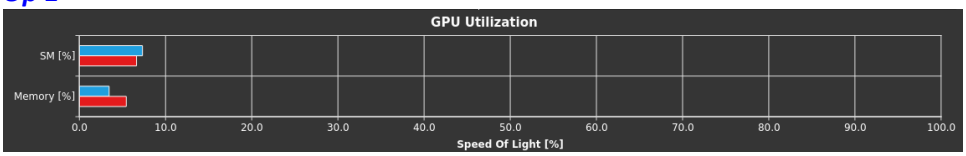
   c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.86608 ms*	*30.5911 ms*	*1.093 s*	*0.86*
1000	*18.702 ms*	*314.776 ms*	*9.420 s*	*0.886*
10000	*186.951 ms*	*3183.48 ms*	*1 m 32.670 s*	*0.8714*

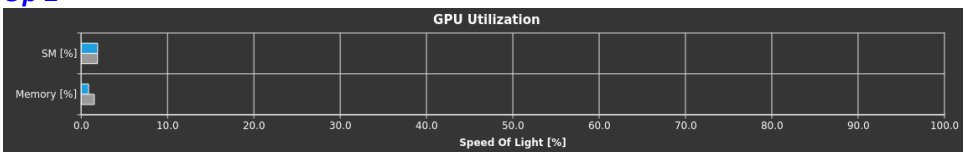
   d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your

answer, directly comparing to your baseline (or the previous optimization this one is built off of).

Op 1



Op 2



In all following sections, I will use numbers directly instead of screenshot SOL.

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*+10.4%*	*-36.58%*	*-37.8%*	*-10.36%*
Op 2	*+0.81%*	*-43.65%*	*-45.45%*	*-3.11%*

The statistics shows marginal improvement when we move the kernel to constant memory. Memory bandwidth drastically decreases as expected, but L1 cache utilization also decreases for some reason.

(build-08beb)

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

CUDA Toolkit v11.5.0 Programming Guide

2. **Optimization 2:** ***Sweeping various parameters to find best values (block sizes, amount of thread coarsening)***

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

Nsight Compute shows my kernel launch only execute 4 blocks, which is less than the GPU's 80 MPs. It also shows on average, each warp spends 3.3 cycles stalled on a fixed latency execution dependency. This calls for adjust block size.

Originally, I go for block size (32, 32), since maximum per thread block is 1024. However, dumping out the data dimension shows
Op 1 *(B=100, M=4, C=1, H=86, W=86, K=7)*
Op 2 *(B=100, M=16, C=4, H=40, W=40, K=7)*
Plug these numbers into kernel size calculation W-K+1 shows 80 and 34 respectively.

Therefore, I decided to do the following tests

- *(4, 4), since GCD(80, 36)=4.*
- *(8, 8), which leaves some warps unused for Op2.*
- *(17, 17), perfect match for Op2, but not so much for Op1.*
- *(20, 20), similar to (8, 8) with some idle warps for Op2, but larger block.*

This implementation stacks on top of the constant memory one, but asides from comparing with previous optimization, we will compare with respect to (4, 4).

- *(4, 4) vs (8, 8), (20, 20), to figure out if increase block size works.*
- *(4, 4) vs (17, 17), (20, 20), whether we should optimize for individual ops.*

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?

GPU has limited SMs, shared memory, and memory bandwidth. Therefore, it is necessary to tweak around different launch size to ensure resources utilization is maximized across these two different OPs.

I would also like to point out that in later optimizations, I will continuously run with different block size, essentially keep redoing this optimization in later optimizations. This section is used to satisfied grading requirements.

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).

Block size (4, 4)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.19454 ms*	*16.3916 ms*	*1.084 s*	*0.86*
1000	*11.8864 ms*	*163.821 ms*	*9.314 s*	*0.886*
10000	*119.357 ms*	*1695.41 ms*	*1 m 31.348 s*	*0.8714*

Block size (8, 8)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.15893 ms*	*15.9704 ms*	*1.131 s*	*0.86*
1000	*11.5606 ms*	*160.342 ms*	*9.702 s*	*0.886*
10000	*114.973 ms*	*1603.01 ms*	*1 m 34.981 s*	*0.8714*

Block size (17, 17)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.20952 ms*	*18.0999 ms*	*1.165 s*	*0.86*
1000	*11.8804 ms*	*187.275 ms*	*10.205 s*	*0.886*
10000	*120.622 ms*	*1909.28 ms*	*1 m 40.053 s*	*0.8714*

Block size (20, 20)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy

100	*1.26736 ms*	*19.9817 ms*	*1.097 s*	*0.86*
1000	*12.6605 ms*	*204.028 ms*	*9.310 s*	*0.886*
10000	*125.81 ms*	*2071.2 ms*	*1 m 31.560 s*	*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).

In this section, we will first compare performance with respect to block (4, 4), which can accommodate both ops without idle warps, but has utilize computation resource much worse.

Increase block size

Block size (8, 8), optimize for op 1, op 2 has half warps left over.

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*-48.61%*	*-46.94%*	*-46.9%*	*-2.65%*
Op 2	*-43.22%*	*-40.48%*	*+86.15%*	*-2.01%*

Block size (20, 20), similar to previous block condition.

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*-51.2%*	*-60.03%*	*+100.11%*	*+6.69%*
Op 2	*-56.36%*	*-46.17%*	*+993.35%*	*+24.64%*

Optimize for different ops

Block (20, 20), optimize for op 1

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*-51.2%*	*-60.03%*	*+100.11%*	*+6.69%*
Op 2	*-56.36%*	*-46.17%*	*+993.35%*	*+24.64%*

Block (17, 17), optimize for op 2

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*-42.09%*	*-50.17%*	*+59.72%*	*-0.20%*
Op 2	*-56.83%*	*-35.56%*	*+1181.68%*	*+13.44%*

Analysis shows that optimizing for op 2 (17, 17) indeed increases occupancy,

	(4, 4)	***(8, 8)***	***(17, 17)***	***(20, 20)***
Op 1	*7.62%*	*3.9%*	*14.44%*	*20.3%*
Op 2	*1.58%*	*2.81%*	*15.02%*	*16.11%*

and average active threads per warp indicates resources are utilized

	(4, 4)	***(8, 8)***	***(17, 17)***	***(20, 20)***
Op 1	*16*	*32*	*27.71*	*30.77*
Op 2	*14.29*	*25.69*	*28.9*	*26.27*

From these results, we can see that while larger block size can hide latency effect and better utilize our cache, they are not particularly beneficial at current stage, since large amount of uncoalesced global will only hinder the performance.

To improve this, we can batch across the B (batch) or M (output channel) dimension, however, this will need shared memory in the kernel in order to hold the calculation results before writing it back.

In next stage, we should start looking into tweaking the kernel itself, instead of lingering with the naïve implementation.

(build-7649f; build-9c79b; build-df87b; build-aa107)

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

How do I choose grid and block dimensions for CUDA kernels?

3. **Optimization 3: *Tiled shared memory convolution***

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

From previous optimization, we already know naïve implementation will cause lots of uncoalesced global memory access. A straightforward way to solve this is to use shared memory, similar to what we have done in MP4.

This also has an added benefit of batch process across other independent dimensions (B and M) mentioned in previous optimization.

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?

Since shared memory is on-chip, it has a much higher bandwidth and lower latency than global memory. From previous optimization, we know that the device is heavily under utilized due to inefficient memory access pattern, and lack of complete utilization of CUDA cores.

We should be able to build on-top of constant memory improvements. In this optimization, we will also slightly go over parameter sweep again.

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).

Basic implementation of tiled convolution kernel.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.185767 ms*	*1.01014 ms*	*1.06 s*	*0.86*
1000	*1.57127 ms*	*10.2723 ms*	*9.11 s*	*0.886*
10000	*15.5011 ms*	*111.34 ms*	*1 m 30.141 s*	*0.8714*

With block (8, 8, 4), where z implies batched across the B dimension.

Batch Size	Op Time 1	Op Time 2	Total Execution	Accuracy

			Time	
100	*0.189615 ms*	*0.814923 ms*	*1.117 s*	*0.86*
1000	*1.58099 ms*	*6.20755 ms*	*9.123 s*	*0.886*
10000	*15.4214 ms*	*60.4442 ms*	*1 m 29.363 s*	*0.8714*

With block (8, 8, 8)

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.197114 ms*	*0.851227 ms*	*1.113 s*	*0.86*
1000	*1.60071 ms*	*6.38676 ms*	*9.544 s*	*0.886*
10000	*15.682 ms*	*61.8753 ms*	*1 m 33.212 s*	*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, very successful.

Increase B block size

*Op 1, block size (8, 8, *), compare with constant memory implementation*

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
1	*+1155.91%*	*+1949.8%*	*+110.5%*	*-91.65%*
4	*+1154.11%*	*+1920.96%*	*+107.84%*	*-91.64%*
8	*+1137.85%*	*+1889%*	*+105.33%*	*-91.52%*

Op 2

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
1	*+4086.97%*	*+4369.92%*	*+72.17%*	*-96.56%*
4	*+4549.94%*	*+7602.29%*	*+195.45%*	*-98.01%*
8	*+4422.48%*	*+7385.23%*	*+189.69%*	*-97.96%*

Re-using the conclusion from previous optimization, with (8, 8) seems to be the balancing point, we can see that to full utilize all SMs, increase B block size to 4 is the sweet spot.

(build-e1787; build-4bde7)

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

My "MP4: 3D Convolution" on WebGPU.

4. **Optimization 4: *Tuning with restrict and loop unrolling (considered as one optimization only if you do both)***

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

After finish tuning the tiled shared memory convolution, we can see the actual convolution is performed on a fixed size kernel (after all, we already stores the kernel in constant memory).

We can do the low hanging fruit by adding `#pragma unroll` to the nested loop that performs kernel multiplication. We also know we are not working with in-place convolution, so we can simply drop in the `__restrict__` keyword.

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?

Compiler generally cannot determine whether points will overlap each other or not. However, we programmer may know they never will, and can therefore use the restrict keyword to promise compiler that pointer is not read by any other pointer in the process.

In general, the more sequential the code, the better the performance on GPU. Therefore, by unrolling loops, we can not only reduce overhead by those loops, but also allowing GPU to access more instructions, and potentially optimize some variables to registers.

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).

Restrict

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.139424 ms*	*0.530685 ms*	*1.169 s*	*0.86*
1000	*1.13016 ms*	*4.16747 ms*	*9.991 s*	*0.886*
10000	*10.9497 ms*	*40.5972 ms*	*1 m 38.343 s*	*0.8714*

Unroll

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.138641 ms*	*0.521346 ms*	*1.08 s*	*0.86*
1000	*1.12127 ms*	*4.15797 ms*	*9.127 s*	*0.886*
10000	*10.9651 ms*	*40.5293 ms*	*1 m 37.837 s*	*0.8714*

Restrict + unroll

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.137104 ms*	*0.556045 ms*	*1.088 s*	*0.86*
1000	*1.1236 ms*	*4.17904 ms*	*9.113 s*	*0.886*
10000	*10.9645 ms*	*40.5441 ms*	*1 m 29.418 s*	*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).

Compare with the tiled convolution in previous optimization, using block size (8, 8, 4) as baseline.

Op 1

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Restrict	*-18.35%*	*+32.37%*	*+32.48%*	*-28.89%*
Unroll	*-17.93%*	*+33.04%*	*+33.01%*	*-29.32%*
Restrict Unroll	*-17.88%*	*+33.1%*	*+32.97%*	*-29.25%*

Op 2

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Restrict	*-9.52%*	*+38.42%*	*+39.95%*	*-33.07%*
Unroll	*-8.13%*	*+40.13%*	*+40.35%*	*-33.09%*
Restrict Unroll	*-8.4%*	*+39.72%*	*+40.22%*	*-33.3%*

When adding the `__restrict__` keyword, we can find the compiler unroll the inner most loop by examining the PTX.

Baseline	***Restrict***

`mul.wide.s32 %rd10, %r32, 4;`
`mov.u64 %rd11, kernel;`
`add.s64 %rd12, %rd11, %rd10;`
`ld.const.f32 %f21, [%rd12];`
`ld.shared.f32 %f22, [%r96];`
`fma.rn.f32 %f46, %f22, %f21, %f46;`
	`mov.u64 %rd8, kernel;`
	`add.s64 %rd9, %rd8, %rd7;`
	`ld.const.f32 %f10, [%rd9];`
	`ld.shared.f32 %f11, [%r78];`
	`fma.rn.f32 %f12, %f11, %f10, %f32;`
	`ld.const.f32 %f10, `**`[%rd9+4];`**
	`ld.shared.f32 %f11, `**`[%r78+4];`**
	`fma.rn.f32 %f15, %f14, %f13, %f12;`
	…

However, with #pragma unroll, we can force the compiler to further unroll the nested loop entirely

```
for (int p = 0; p < K; p++) {
  for (int q = 0; q < K; q++) {
  }
}
```

(build-f2beb; build-5c5e7; build-3361b)

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

CUDA Pro Tip: Optimize for Pointer Aliasing

5. **Optimization 5: *Using Streams to overlap computation with data transfer***

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

As of time of writing, I saw someone points out that leaderboard currently counts layer time instead of OP time as the README shows. While I don't think streaming should be my top priority since there are still a few tricks I can work with, I still want to

get put on the leaderboard after putting these efforts in optimizing my code. After going through my source code, I believe I can integrate streaming easily, so here I am.

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?

By using stream, one can overlap memory transfer between host-device with kernel execution. This can allow one to better utilize idle SMs (assuming these SMs have enough resource to take on more kernels).

Working with stream is also orthogonal with previous optimizations, however, TA's clarification on Canvas shows there is an additional `cudaDeviceSynchronize` *after prolog, therefore, I choose to pin host side memory in prolog, and do the actual asynchronous copy in kernel call function* `conv_forward_gpu`*, which may make the separation between OP time and layer time less meaningful.*

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).

1 stream *(same kernel code as previous optimization)*

Batch Size	Op Time 1 (Layer Time 1)	Op Time 2 (Layer Time 2)	Total Execution Time	Accuracy
100	*2.37819 ms (5.2166 ms)*	*2.28425 ms (4.51786 ms)*	*1.067 s*	*0.86*
1000	*21.7951 ms (43.5798 ms)*	*19.8404 ms (35.7771 ms)*	*9.284 s*	*0.886*
10000	*215.609 ms (411.711 ms)*	*195.173 ms (334.709 ms)*	*1 m 29.143 s*	*0.8714*

4 streams

Batch Size	Op Time 1 (Layer Time 1)	Op Time 2 (Layer Time 2)	Total Execution Time	Accuracy
100	*2.05837 ms (5.00366 ms)*	*1.71561 ms (3.90921 ms)*	*1.091 s*	*0.86*
1000	*18.0272 ms (39.769 ms)*	*14.142 ms (29.4891 ms)*	*9.101 s*	*0.886*
10000	*177.284 ms (371.931 ms)*	*137.748 ms (290.513 ms)*	*1 m 33.399 s*	*0.8714*

8 streams

Batch Size	Op Time 1 (Layer Time 1)	Op Time 2 (Layer Time 2)	Total Execution Time	Accuracy
100	*1.13036 ms (3.95326 ms)*	*1.14952 ms (3.36664 ms)*	*1.074 s*	*0.86*
1000	*8.97379 ms*	*7.19614 ms*	*9.051 s*	*0.886*

	(30.1844 ms)	*(22.3229 ms)*		
10000	*86.6162 ms (285.025 ms)*	*67.347 ms (203.777 ms)*	*1 m 28.838 s*	*0.8714*

For some reason, I am unable to get back to 16 streams on the exclusive queue, and the time spend in pinning host memory suddenly spiked after Nov 16th 18:30. As of time of writing, my best score (490 ms) uses 16 streams.

On the side note, as I asked in #728, CUDA 11.2 supports asynchronous `malloc` *and* `free`*, which can be beneficial in our test set, since 10k images is not a small number.*

(build-1edc7; build-19cc3; build-5aa87)

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 is very successful.

Since adding stream requires some move around with the `cudaMalloc`*, we will use 1 stream as baseline. The more the streams are, the more kernel launch there is, and smaller the kernel itself, parenthesis below includes normalized result (divide by number of streams).*

Op 1

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
4	*-2.14% (-0.54%)*	*-2.31% (-0.58%)*	*-0.96% (-0.24%)*	*-74.05% (-18.51%)*
8	*-5.63% (-0.7%)*	*-5.95% (-0.74%)*	*-2.27% (-0.28%)*	*-86.26% (-10.78%)*

Op 2

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
4	*-5.45% (-1.36%)*	*-5.17% (-1.29%)*	*-0.79% (-0.2%)*	*-73.38% (-18.35%)*
8	*-10.1% (-1.2625%)*	*-9.7% (-1.21%)*	*-4.71% (-0.59%)*	*-85% (-10.63%)*

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

- *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*
- *Enhancing Memory Allocation with New NVIDIA CUDA 11.2 Features*

6. **Optimization 6: *Shared memory matrix multiplication and input matrix unrolling***

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

It is known from ECE558 (Digital Imaging) that 2D convolution can be implemented using matrix multiplication by converting an input matrix to a Toeplitz matrix. This is also the foundation of most modern neural network libraries.

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 my implementation, we first launch a kernel called `im2col`*, which flatten the input matrix* `x` *into column matrix. For a naïve implementation, we simply replicate the convolution kernel, but instead of doing the actual FMA operation, we first save it out to a temporary array* `xc`*, and feed kernel weights and* `xc` *to a generic matrix multiplication, recycled from my MP3.*

This optimization can easily work with restrict-unroll, and the first constant memory optimization, without any modification, since we are unrolling the `x` *instead 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).

Batch Size	Op Time 1 (Layer Time 1)	Op Time 2 (Layer Time 2)	Total Execution Time	Accuracy
100	*0.495665 ms (14.1678 ms)*	*1.22994 ms (12.2753 ms)*	*1.202 s*	*0.86*
1000	*4.79164 ms (140.557 ms)*	*12.2056 ms (121.93 ms)*	*10.165 s*	*0.886*
10000	-	-	-	-

*Under current condition, 10k images leads to out-of-memory exception. Taking op 1 as an example, the unrolled column matrix requires 10000 * (80 * 80) * (1 * 7 * 7) * 4 = 11.962 GiB.*

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).

Not really, since this implementation currently cannot run the full benchmark (10k) due to the enormous memory requirements.

We compare this implementation with the restrict-unroll optimization, using the same block size (8, 8, 4) as before.

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*+18.54%*	*-65.76%*	*-65.86%*	*+505.93%*
Op 2	*+16.14%*	*-86.27%*	*-86.59%*	*+406.34%*

While computation-wise, GEMM indeed increases the utilization, nsight shows that all pipelines are under-utilized. This is not hard to imagine, consider the original grid

*dimension is based on output (height, width), current optimization grid size is based on (height*width, output channels).*

Per Campus Wire discussion, we will continue the discussion in next optimization, where we can run the entire dataset.

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

`im2col` – *Rearrange image blocks into columns*

7. **Optimization 7:** ***Kernel fusion for unrolling and matrix-multiplication (requires "Shared memory matrix multiplication and input matrix unrolling")***

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

From previous optimization, it is clear that without kernel fusion, the unrolled column matrix is too large to hold in the device memory. We _need_ this optimization for GEMM-based convolution to work in this milestone.

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?

Instead of output the unrolled matrix somewhere, when loading data to shared memory in matrix multiplication kernel, we directly calculate the unrolled index, and grab the value from global memory.

However, in my implementation, I have played around with numerous implementations, this is the final implementation details.

- *Region of interest is first loaded from global memory (`x`) to shared memory (`tile_x`).*
- *Unrolled index is pre-calculated in `prolog` and saved in a constant array.*
- *Unrolled result is converted from the shared memory copy to another array in shared memory (`tile_xc`).*
- *I implement a matrix multiplication kernel that allow asymmetric tiles.*

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).

The kernel is launched with block size (64, 4, 4) where grid is calculated based on (`H_out * W_out`, `M`, `B`). This uses about 29 KiB of shared memory.

Batch Size	Op Time 1 (Layer Time 1)	Op Time 2 (Layer Time 2)	Total Execution Time	Accuracy
100	*0.428028 ms (6.97454 ms)*	*1.25891 ms (6.17669 ms)*	*1.111 s*	*0.86*
1000	*3.88637 ms (61.7753 ms)*	*12.0608 ms (55.4058 ms)*	*9.540 s*	*0.886*
10000	*38.4248 ms (604.261 ms)*	*119.629 ms (540.379ms)*	*1 m 33.434s*	*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).

If we are comparing with separate kernel, then this optimization indeed helps. From 100 and 1k dataset, the time improves a bit, and the layer time improves significantly due to reduced amount of needed device memory.

However, when we compare with the last equivalent kernel, restrict-unroll.

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*-38.86%*	*-54.18%*	*-53.65%*	*+247.88%*
Op 2	*-44.28%*	*-53.06%*	*-53.95%*	*+188.39%*

From the statistics, executed instructions increased +117.75%, but memory throughput decrease 74.47%. It is apparent that current GEMM strategy does more useless computation, which lowers the computation throughput.

(build-241ff; build-5af93; build-4fbdb)

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

Parallel Multi Channel Convolution using General Matrix Multiplication

8. **Optimization 8:** ***Fixed point (FP16) arithmetic. (note this can modify model accuracy slightly)***

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

This is a low hanging fruit after working through the matrix multiplication. We simply change stored data type to half and uses intrinsic functions for 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?

GPU with compute capability over 5.3 has native FP16 support, whose hardware arithmetic instructions can operate on 2 FFP16 values at a time. Therefore, theoretical peak throughput can directly double if we can formulate our arrays in `half2` *operations.*

I am directly using matrix multiplication with this optimization. At the final step, each thread works over a column-row of values and accumulate them together, which is an awesome use case to force cast `half*` *to* `half2*`*, and utilize* `__hfma2` *to do FP16 multiply-and-add.*

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).

In order to compare with previous optimization, we use the same block size block size (64, 4, 4).

Batch Size	Op Time 1 (Layer Time 1)	Op Time 2 (Layer Time 2)	Total Execution Time	Accuracy
100	*0.448387 ms (8.10683 ms)*	*1.3412 ms (6.94776 ms)*	*1.175 s*	*0.86*
1000	*4.04071 ms (72.0852 ms)*	*12.9155 ms (60.8562 ms)*	*10.367 s*	*0.886*
10000	*40.0566 ms (682.635 ms)*	*228.24 ms (778.147ms)*	*1 m 50.975 s*	*0.8713*

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).

In terms of execution time, no improvement, but other statistics do hint FP16 helps.

We first look at the gross overview.

	SM [%]	***Memory [%]***	***L1 Cache [%]***	***Duration [%]***
Op 1	*-20.43%*	*+26.21%*	*+26.05%*	*+4.7%*
Op 2	*-23.59%*	*+28.02%*	*+27.98%*	*+7.77%*

The computation turn into a memory bottleneck problem. This is expected when switching to FP16, where computational throughput should double. This is further confirmed by compute workload analysis

- ***LSU** -56.77%*
- ***FP16** (from 0 to 3.42%)*

where packing our array into `half2` *FP16 operations indeed reduce the number of load-store needed (packed), and the FP16 units chew through them (utilized). To show improvements with FP16, we need to further test with block size in order to figure out the best launch size, with limited time on hand right now, I am going to stop at here.*

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

Mixed-Precision Programming with CUDA 8