

Milestone 3 Report

Google Drive Link:

https://drive.google.com/drive/folders/1OdfaJfkpnaOxyoqwVSeVKv_BDdKz-h2?usp=drive_link

0. Baseline:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	<i>1.418ms</i>	<i>1.522ms</i>	<i>0m1.507s</i>	<i>0.86</i>
1000	<i>10.095ms</i>	<i>14.240ms</i>	<i>0m9.164s</i>	<i>0.886</i>
10000	<i>77.781ms</i>	<i>124.33ms</i>	<i>1m29.972s</i>	<i>0.8714</i>

1. Req_0: __Stream__

https://drive.google.com/drive/folders/1Vu8muV2rKgBkq4FgLE4YME9ZF740mFAI?usp=drive_link

- a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

By streaming, I can parallelize the large task. Streaming divides large vectors into segments and overlaps the data transferring and computing phrases for adjacent segments. Therefore, when one stream is performing memory I/O operations, another stream can launch the kernel functions.

- b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

I can parallelize the processing of a batch of images into several smaller batch segments. I defined a variable of the number of streams and separates the batch segments accordingly.

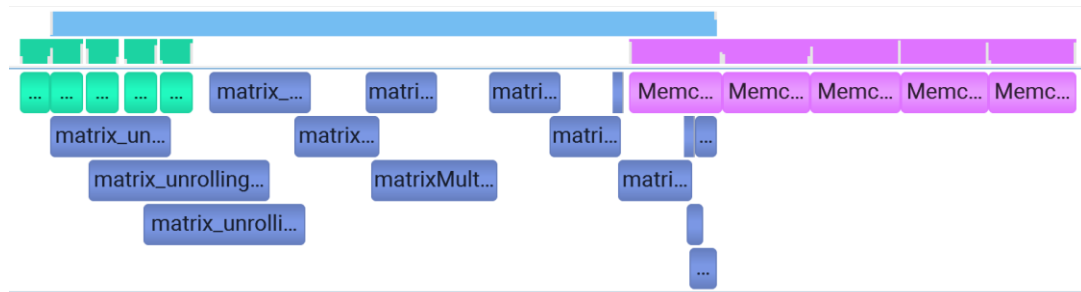
```
int num_streams = 5;
int seg_batch = 20;
```

Also, unroll_matrix and unroll_output used in kernel function are irrelevant to stream, so I declare these objects inside each iteration of stream execution. To facilitate implementation, I moved the functionality of conv_forward_gpu() to conv_forward_gpu_prolog().

- c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.00204ms	0.00208ms	0m1.417s	0.86
1000	0.00247ms	0.00265ms	0m9.260s	0.886
10000	0.00408ms	0.003327ms	1m35.143s	0.8714

It didn't match my expectations. The total execution time was even slower than the baseline. I tried several settings for `num_streams` and `seg_batch`, and the results were consistent. The reason for a lower performance can be seen in profile.



This is the profile of [`num_streams` = 5, `seg_batch` = 20]. Green parts were `MemcpyHtoD`, and pink parts were `MemcpyHtoD`. We can see overlaps did exist, but were very small compared to the whole execution. The reason is that the kernel function is relatively simple, and memory copy occupied most of the running time. Since `DtoH` and `HtoD` can only be done sequentially among streams, memory transferring speed became the bottleneck for this implementation. The time saved by stream was less than the overhead of multiple streams.

- d. Does this optimization synergize with any other optimizations? How?

Since this is just parallelism and kernel is unchanged, it doesn't synergize with other optimizations.

- e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Lecture Slide 20: Data Transfer and CUDA Stream (Task Parallelism)

CUDA Technical Blog <https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9>

2. Req_1: __Tensor Core__

https://drive.google.com/drive/folders/1rBF45831Lzmi0cGEB2rxOqr9FQti_9Yo?usp=drive_link

- a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

Tensor core is a hardware specified to matrix multiplication. It can compute the matrix multiplication result much faster than the original shared memory matmul kernel. Therefore, it is expected that tensor core can accelerate the matmul kernel in our convolution kernel.

- b. How did you implement your code? Explain thoroughly and show code snippets.

Justify the correctness of your implementation with proper profiling results.

I set the tensor core to compute in half datatype. The general idea is to use tiling for type casting, and then copy the tiles to tensor cores and do the computation.

```
if (row < numRows && tileId * TILE_WIDTH + tx < numAColumns) {
    tileA[ty][tx] = __float2half(A[(size_t) row * numAColumns + tileId * TILE_WIDTH + tx]);
} else {
    tileA[ty][tx] = __float2half(0.0f);
}
if (col < numBColumns && tileId * TILE_WIDTH + ty < numBRows) {
    tileB[ty][tx] = __float2half(B[(size_t) (tileId * TILE_WIDTH + ty) * numBColumns + col]);
} else {
    tileB[ty][tx] = __float2half(0.0f);
}
```

Then, I can load the tiles into the tensor core. It's worth mentioning that since tensor cores operation is done in warp-level, I only used the first warp to do the work.

```
if (ty == 0 || ty == 1){ // use one warp
    wmma::load_matrix_sync(a_frag, (half*)tileA, TILE_WIDTH);
    wmma::load_matrix_sync(b_frag, (half*)tileB, TILE_WIDTH);
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
}
__syncthreads();
```

- c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	1.47ms	1.41ms	0m1.490s	0.86
1000	9.97ms	13.24ms	0m9.606s	0.886
10000	76.69ms	114.88ms	1m32.156s	0.8714

As shown in the table, the op time is reduced, so my implementation is correct.

This can also be seen in the Nsight Compute:

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	analysis_file_tensor_core	825 - matrixMultiplyShared	(4000000, 1, 1)x(16, 16, 1)	30.01 ms	39,161,854	0 - NVIDIA A40	1.31 Ghz	[1752073] m3	
Baseline 1	analysis_file_baseline	519 - matrixMultiplyShared	(4000000, 1, 1)x(16, 16, 1)	32.48 ms	42,373,256	0 - NVIDIA A40	1.30 Ghz	[2745841] m2	

However, I noticed that the improvement is very limited. This is because originally the matrix multiplication didn't occupy the main running time. Therefore, the improvement is small compared to the entire performance. In file "profile.outs," one can see that matmul kernel only takes 1/4 of total kernel runtime.

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	
48.8	85,638,174	1	85,638,174.0	85,638,174.0	85,638,174	85,638,174	0.0	10000 9 4	16 16	1 matrix_unrolling_kernel(constant)
21.0	36,784,283	1	36,784,283.0	36,784,283.0	36,784,283	36,784,283	0.0	10000 25 1	16 16	1 matrix_unrolling_kernel(constant)
15.7	27,479,532	1	27,479,532.0	27,479,532.0	27,479,532	27,479,532	0.0	4000000 1 1	16 16	1 matrixMultiplyShared(constant)
10.8	18,885,108	1	18,885,108.0	18,885,108.0	18,885,108	18,885,108	0.0	722500 1 1	16 16	1 matrixMultiplyShared(constant)
2.0	3,580,403	1	3,580,403.0	3,580,403.0	3,580,403	3,580,403	0.0	25 10000 1	256 1	1 matrix_permute_kernel(constant)
1.7	2,933,965	1	2,933,965.0	2,933,965.0	2,933,965	2,933,965	0.0	5 10000 1	256 1	1 matrix_permute_kernel(constant)
0.0	4,736	2	2,368.0	2,368.0	2,304	2,432	90.5	1 1 1	1 1	1 do_not_remove_this_kernel()
0.0	4,609	2	2,304.5	2,304.5	2,209	2,400	135.1	1 1 1	1 1	1 prefn_marker_kernel()

d. Does this optimization synergize with any other optimizations? How?

No. This optimization only modified the matrix multiplication method, so it has no impact on other optimizations.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

CUDA Docs for Warp Matrix Function <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-matrix-functions>

CUDA Technical Blog <https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9>

Lecture Slide 22: Accelerating Matrix Operations

3. Req_2: __Kernel Fusion__

https://drive.google.com/drive/folders/1T4yYWqrumS3sy8NuteSj1GBuJHch34PC?usp=drive_link

a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

First, the three kernels will be fused into one, reducing the overhead for multiple kernel launches. Also, after kernel fusion, the unrolling matrix doesn't need actual

load and storage. We can save global memory access by directly loading the element in the correct position into the shared memory. It is expected that this optimization will greatly reduce the runtime, as we already realized that memory operation took much time in Req_0.

- b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

Set the kernel configuration according to the original three kernels. I tried several configurations like putting batch in grid's x-dimension, or not tiling Map_out dimension. However, the former design performed poorly due to insufficient use of memory burst, the latter design faced difficulties in finding the row index for unrolling.

```
dim3 grid_dim(ceil((Height_out * Width_out) / (1.0 * TILE_WIDTH)), ceil(Map_out / (TILE_WIDTH * 1.0)), Batch);
dim3 block_dim(TILE_WIDTH, TILE_WIDTH, 1);
conv_forward_kernel<<<grid_dim, block_dim>>>(device_input, device_mask, device_output, Batch, Map_out, Channel,
```

The hardest part is to find the corresponding index between matmul and unrolling.

Below is a snippet of index settings.








```
int batch_idx = blockIdx.z;
int col = blockIdx.x * TILE_WIDTH + threadIdx.x; // pixel
int row = blockIdx.y * TILE_WIDTH + threadIdx.y; // map
int numAColumns = Channel * K * K;
int numBColumns = Height_out * Width_out;
int h = col / Width_out;
int w = col % Width_out;
int mask_size = K * K;

float Cvalue = 0;
for (int tileId = 0; tileId < ceil(numAColumns / (TILE_WIDTH * 1.0)); tileId++) {
    int colA = tileId * TILE_WIDTH + threadIdx.x;
    int rowB = tileId * TILE_WIDTH + threadIdx.y;
    int c = rowB / mask_size;
    int p = (rowB % mask_size) / K;
    int q = (rowB % mask_size) % K;
```

- c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.628ms	0.329ms	0m1.574s	0.86
1000	5.679ms	3.047ms	0m10.476s	0.886
10000	55.014ms	30.106ms	1m37.738s	0.8714

The operation time for two layers is greatly reduced by kernel fusion. Memory workload Analysis in profile also justified my expectation. The fused kernel used much more shared memory and L1 cache.

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
 Current	fu_on	820 - conv_forward_kernel	(400, 1, 10000)x(16, 16, 1)	72.86 ms	95,084,387	0 - NVIDIA A40	1.30 Ghz	[721800] m3	
 Baseline 2	analysis_file_baseline	516 - matrix_unrolling_kern...	(10000, 25, 1)x(16, 16, 1)	38.06 ms	49,629,676	0 - NVIDIA A40	1.30 Ghz	[2745841] m2	
 Baseline 3	analysis_file_baseline	519 - matrixMultiplyShared	(4000000, 1, 1)x(16, 16, 1)	32.48 ms	42,373,256	0 - NVIDIA A40	1.30 Ghz	[2745841] m2	
 Baseline 4									

SummaryDetailsSourceContextCommentsRawSession

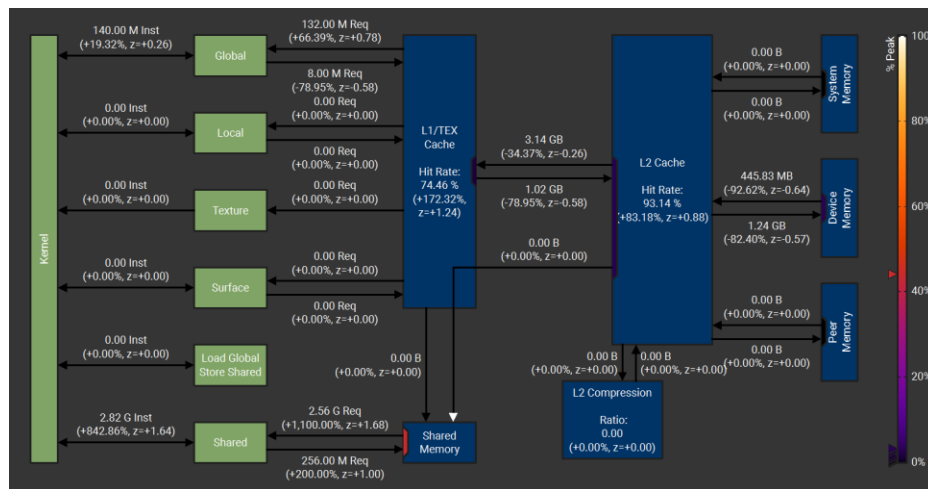
CompareToolsViewExport

▼ 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.

Memory Throughput [Gbyte/s]	23.12	(-95.90%, z=-1.70)	Mem Busy [%]	47.37	(+18.68%, z=+0.80)
L1/TEX Hit Rate [%]	74.46	(+172.32%, z=+1.24)	Max Bandwidth [%]	78.03	(-3.73%, z=-0.32)
L2 Hit Rate [%]	93.14	(+83.18%, z=+0.88)	Mem Pipes Busy [%]	78.03	(+156.78%, z=+1.08)
L2 Compression Success Rate [%]	0	(+0.00%, z=+0.00)	L2 Compression Ratio	0	(+0.00%, z=+0.00)



In file “profile.out,” one can also see that the kernel launching time has been reduced. However, since originally this time didn’t take long, the optimization effect can be ignored.

Baseline:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
43.7	296,699,356	8	37,087,419.5	9,973,505.0	25,558	149,177,541	57,651,732.5	cudaMemcpy
31.2	212,243,838	12	17,686,986.5	2,150,258.5	121,597	117,620,501	36,978,460.7	cudaFree
25.0	170,036,877	12	14,169,739.8	196,693.0	106,219	163,379,234	46,999,705.6	cudaMalloc
0.1	420,808	10	42,080.8	23,494.0	7,985	175,939	52,124.3	cudaLaunchKernel
0.0	37,238	6	6,206.3	5,630.0	3,857	10,480	2,245.5	cudaDeviceSynchronize
0.0	1,252	1	1,252.0	1,252.0	1,252	1,252	0.0	cuModuleGetLoadingMode

Fusion:

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
54.6	306,530,714	8	38,316,339.3	9,659,792.0	23,514	158,651,550	60,522,137.2	cudaMemcpy
28.1	157,794,895	8	19,724,361.9	175,799.5	112,200	156,150,912	55,124,953.2	cudaMalloc
15.1	84,933,483	6	14,155,580.5	7,854.5	3,888	54,872,562	23,284,725.0	cudaDeviceSynchronize
2.0	11,320,215	8	1,415,026.9	1,222,001.0	130,274	3,905,914	1,234,847.6	cudaFree
0.1	325,972	6	54,328.7	38,672.5	17,122	132,599	44,047.0	cudaLaunchKernel
0.0	1,313	1	1,313.0	1,313.0	1,313	1,313	0.0	cuModuleGetLoadingMode

d. Does this optimization synergize with any other optimizations? How?

This optimization can be stacked to tensor core operation, to further optimize the loop matrix multiplication to tensor operation.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Lecture Slide 12: Computation in Deep Neural Network.

4. Op_1: __restrict__

https://drive.google.com/drive/folders/1OEU0FiWBGun8cUBiStHM-s-gxhq430BJ?usp=drive_link

a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

By using __restrict__ keywords, I told the compilers that there would be no aliasing between pointers, and it can optimize the compilation whatever it likes.

b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

Just add keywords __restrict__ to every pointer in function declaration.

```
__global__ void matrixMultiplyShared(const float * __restrict__ A, const float * __restrict__ B, float * __restrict__ C,
                                     int numARows, int numAColumns,
                                     int numBRows, int numBColumns,
                                     int numCRows, int numCColumns)
{
```

c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	1.441ms	1.673ms	0m1.408s	0.86
1000	10.075ms	14.199ms	0m9.473s	0.886
10000	77.529ms	124.012ms	1m30.772s	0.8714

There was no obvious optimization, which aligned with my expectation. In the baseline, I had already guaranteed that there were no overlaps between pointers, so the compiler wasn't able to make any further optimizations. There was also no significant change in profiling results.

- d. Does this optimization synergize with any other optimizations? How?

Since this is just a keyword, it doesn't synergize with other optimizations.

- e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

CUDA Technical Blog <https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/>

5. Op_2: __Loop Unrolling__

https://drive.google.com/drive/folders/1cATpvnx42C2jcOLi559F19-y1c8fVR-v?usp=drive_link

- a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

There are many loops in the baseline. By asking the compiler to unroll these loops, I can potentially increase the execution parallelism.

- b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

Since every loop in the baseline implementation has a controlled number of iterations, I can directly add the keyword “#pragma unroll” for every loop.

```
int w_out = (blockIdx.y % num_tile_w) * TILE_WIDTH + threadIdx.x;
if (h_out < Height_out && w_out < Width_out) {
    int out_col = b * Height_out * Width_out + h_out * Width_out + w_out;
    #pragma unroll
    for (int p = 0; p < K; ++p) {
        #pragma unroll
        for (int q = 0; q < K; ++q) {
            int in_row = h_out + q;
```

- c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	1.501ms	1.535ms	0m1.391s	0.86
1000	10.106ms	14.244ms	0m9.198s	0.886
10000	77.633ms	124.295ms	1m28.105s	0.8714

The performance slightly improved. This is also shown in the profile. However, since the loops in the code were all quite simple with not so many iterations, there was no significant improvement.

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	analysis...e_unroll	825 - matrixMultiplyShared	(4000000, 1, 1)x(16, 16, 1)	32.47 ms	42,360,421	0 - NVIDIA A40	1.30 Ghz	[3001377] m3	
Baseline 3	analysis_file_baseline	519 - matrixMultiplyShared	(4000000, 1, 1)x(16, 16, 1)	32.48 ms	42,373,256	0 - NVIDIA A40	1.30 Ghz	[2745841] m2	

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	analysis...e_unroll	822 - matrix_unrolling_kernel	(10000, 25, 1)x(16, 16, 1)	38.06 ms	49,636,104	0 - NVIDIA A40	1.30 Ghz	[3001377] m3	
Baseline 2	analysis_file_baseline	516 - matrix_unrolling_kern...	(10000, 25, 1)x(16, 16, 1)	38.06 ms	49,629,676	0 - NVIDIA A40	1.30 Ghz	[2745841] m2	

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	analysis...e_unroll	828 - matrix_permute_kernel	(25, 10000, 1)x(256, 1, 1)	3.58 ms	4,668,068	0 - NVIDIA A40	1.30 Ghz	[3001377] m3	
Baseline 4	analysis_file_baseline	522 - matrix_permute_kernel	(25, 10000, 1)x(256, 1, 1)	3.59 ms	4,676,483	0 - NVIDIA A40	1.30 Ghz	[2745841] m2	

d. Does this optimization synergize with any other optimizations? How?

It may synergize with streaming, as I used a loop to control streams. However, this part falls in the host code, so there will be no effect to the kernel layer time.

```

dim3 block_dim(FILL_WIDTH, FILL_WIDTH, 1);
for (int j = 0; j < num_streams; j++) {
    int start = i * seg_batch * num_streams + j * seg_batch;
    matrix_unrolling_kernel<<<grid_unroll_dim, block_dim, 0, str
}

```

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

CUDA Developer Forum <https://forums.developer.nvidia.com/t/loop-unroll-remainder-perf/209443>, <https://forums.developer.nvidia.com/t/understanding-unrolling-and-concurrent-memory-operations/38617/3>

6. Op_5: __FP16__

<https://drive.google.com/drive/folders/1W0WsuipIMdzzieOUBFGAqKniZEiEAvvS?usp=sharing>

- How does this optimization theoretically optimize your convolution kernel? Expected behavior?

FP16 represents a floating-point number using 16 bits rather than 32 bits. In such a way it can reduce memory consumption. Given the same memory bandwidth, it can double the memory throughput theoretically.

- How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

The most memory-demanding operation in the code lies in matmul. Therefore, I want to turn the input unrolled matrix and mask from float to half. For the unrolled matrix, since there is already an one-to-one mapping in the unrolling kernel, I can just add `__float2half()` there.

```
int in_col = w_out + q;
int out_row = c * K * K + p * K + q;
out_2d(out_row, out_col) = __float2half(in_4d(b, c, in_row, in_col));
}
```

For mask, I launched an extra kernel to do the conversion and stored the half mask in a new area.

```
cudaMalloc((void**)&device_mask_half, mask_size * sizeof(half));

dim3 grid_mask_dim(ceil(mask_size / (BLOCK_SIZE*1.0)), 1, 1);
dim3 block_mask_dim(BLOCK_SIZE, 1, 1);
mask2half<<<grid_mask_dim, block_mask_dim>>>(device_mask, mask_size, device_mask_half);
```

Finally, in the matmul kernel, do the half type multiplication and convert the result back to float.

```

    if (row < numCRows && col < numCColumns) {
        for (int i = 0; i < TILE_WIDTH; i++) {
            val = __hadd(val, hmul(tileA[ty][i], tileB[i][tx]));
        }
        __syncthreads();
    }

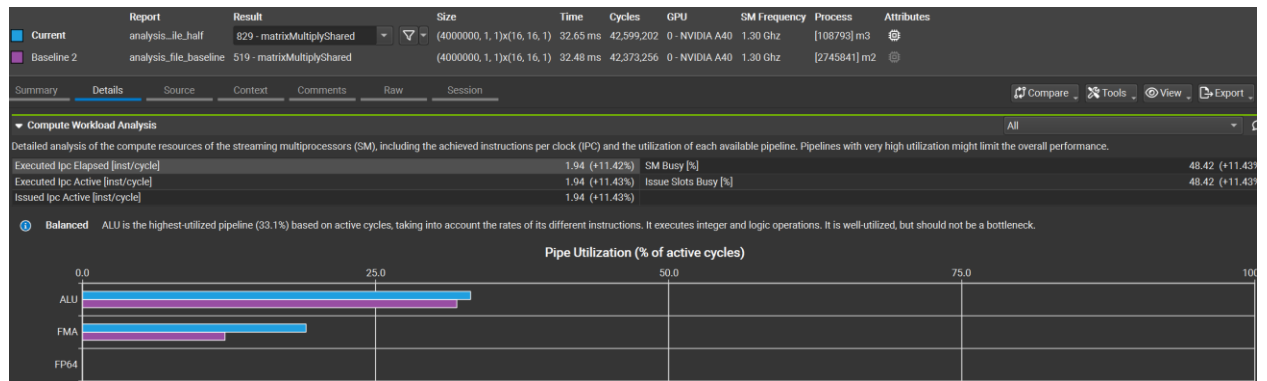
    if (row < numCRows && col < numCColumns) {
        C[row * numCColumns + col] = __half2float(val);
    }
}

```

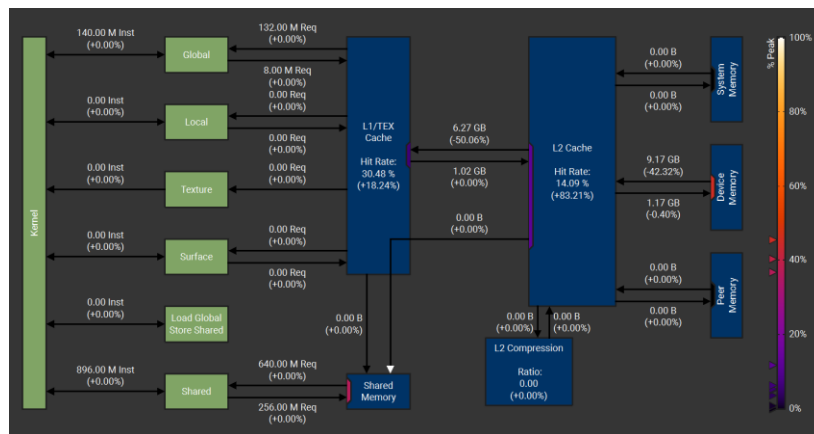
c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	1.359ms	1.303ms	0m1.415s	0.86
1000	9.733ms	10.779ms	0m9.411s	0.887
10000	74.645ms	90.274ms	1m30.293s	0.8716

The performance is improved. In the profiling files, one can also see an increase in the utilization of the compute resources of SM.



Additionally, L1 and L2 cache hit rates both increased.



- d. Does this optimization synergize with any other optimizations? How?

This optimization can be adopted to tensor cores, as tensor cores mainly support fp16 operations.

- e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

CUDA Technical Blog <https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/>

Cuda Docs https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/group_CUDA_MATH_HALF_ARITHMETIC.html,
https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/struct_half.html#_CPPv46_half