Heng An Cheng

Milestone 3 Report

[Google Drive link to the top-level m3 folder]
https://drive.google.com/drive/folders/1gSlyrFQ7bd9TMtDJ8_ofL5btDZ7ZaGxc?usp=sharing

0. Baseline:

a. M2 kernel fusion

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.4630 ms*	*0.3548 ms*	*1.458 s*	*0.86*
1000	*3.89628 ms*	*3.37797 ms*	*9.913 s*	*0.886*
10000	*38.2504 ms*	*33.5331 ms*	*1 m 36.021 s*	*0.8714*

b. M1 unroll

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.4688 ms*	*1.53357 ms*	*1.411s*	*0.86*
1000	*9.2552 ms*	*13.9012 ms*	*9.443s*	*0.886*
10000	*69.494 ms*	*119.248 ms*	*1m30.953*	*0.8714*

1. Req_0: Using Streams to overlap computation with data transfer

[Google Drive link to subfolder req_0]
https://drive.google.com/drive/folders/1LsqC0ZBKrhW5z09hsBkKnx51vg7d9DT9?usp=sharing

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

   i. The core idea of using CUDA streams is to overlap data transfer and kernel execution to maximize GPU utilization. By dividing the total batch of image data into smaller chunks (e.g., one image per chunk) and assigning each chunk to a different stream, the GPU can simultaneously process one chunk while transferring another. This overlapping hides memory transfer latency and ensures that the GPU is kept busy, which ideally improves overall throughput.

   ii. For smaller batch sizes (e.g., 100), this optimization may not yield noticeable performance gains—or may even degrade performance—due to

the overhead associated with stream creation and management. In contrast, for larger batch sizes (e.g., 10,000), the benefits of overlapping become more significant, leading to better GPU utilization and faster execution. However, using too many streams can saturate GPU resources and introduce contention, so it's important to strike a balance.

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

We can break down the implementation into several parts below:

i. Stream & Buffer initialization: Create 8 streams and allocate per-batch GPU buffers for unrolled inputs and outputs

```
const int NUM_STREAMS = 8; // Allocate per-batch buffers
float *unrolled_per_batch[NUM_STREAMS] ;
float *output_per_batch[NUM_STREAMS] ;


cudaStream_t streams[NUM_STREAMS];
// 8 streams
for (int i = 0; i < NUM_STREAMS; i++){
    cudaStreamCreate(&streams[i]);
    cudaMalloc(&unrolled_per_batch[i], Height_unrolled * Width_unrolled * sizeof
    (float));
    cudaMalloc(&output_per_batch[i], Map_out * Width_unrolled * sizeof(float));
}
```

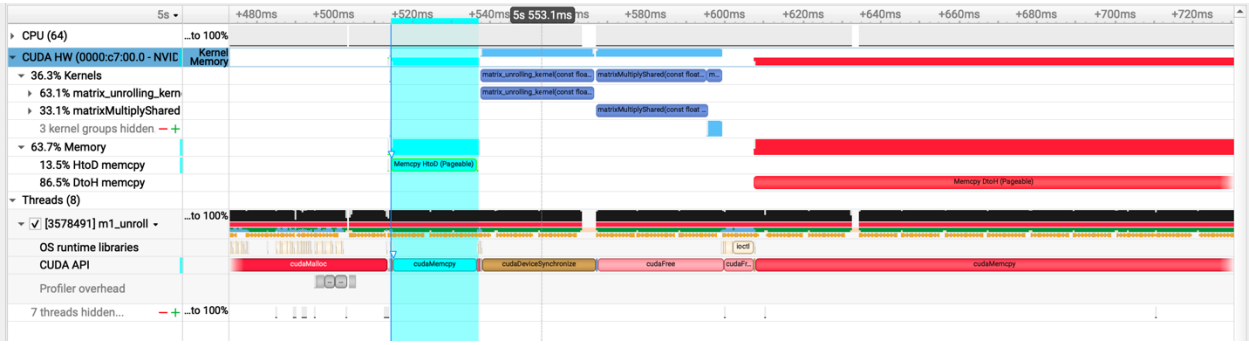
ii. Streaming Each Batch: Each batch is assigned to a stream (cyclically), and we prefetch the next batch while executing kernels for the current one

```
int streamid;
for (int b = 0; b < Batch; ++b) {
    streamid = b % NUM_STREAMS;

    float *input_b = *device_input_ptr + b * input_offset;
    float *output_b = *device_output_ptr + b * output_offset;
    const float *host_input_b = host_input + b * input_offset;

    // Prefetch next batch
    if (b < Batch - 1) {
        int next_streamid = (b + 1) % NUM_STREAMS;
        float *input_next_b = *device_input_ptr + (b + 1) * input_offset;
        const float *host_input_next_b = host_input + (b + 1) * input_offset;

        cudaMemcpyAsync(input_next_b, host_input_next_b,
                        Channel * Height * Width * sizeof(float),
                        cudaMemcpyHostToDevice, streams[next_streamid]);
    }

    // Compute kernels
    matrix_unrolling_kernel_stream_with_batch<<<DimGrid, DimBlock, 0, streams[streamid]>>>(
        input_b, unrolled_per_batch[streamid], 1,
        Channel, Height, Width, K
    );

    matrixMultiplyShared<<<matmul_grid_dim, matmul_block_dim, 0, streams[streamid]>>>(
        *device_mask_ptr, unrolled_per_batch[streamid], output_per_batch[streamid],
        Map_out, Height_unrolled,
        Height_unrolled, Width_unrolled,
        Map_out, Width_unrolled
    );

    matrix_permute_kernel_stream_with_batch<<<permute_kernel_grid_dim, PERMUTE_BLOCK_SIZE, 0, streams[streamid]>>>(
        output_per_batch[streamid], output_b,
        Map_out, 1, Width_unrolled
    );
}
```

iii. Synchronization & Clean up Streams: Wait for all streams to complete and clean them up

```
cudaDeviceSynchronize();

// Cleanup
for (int i = 0; i < NUM_STREAMS; ++i) {
    cudaStreamDestroy(streams[i]);
    cudaFree(unrolled_per_batch[i]);
    cudaFree(output_per_batch[i]);
}
```

Baseline:



Req-0:

From the profiler output, we can clearly see that the streams version achieves the intended **overlap** between memory transfers and kernel execution. Specifically, while one batch is being copied from host to device, the GPU is already computing the previous batch.

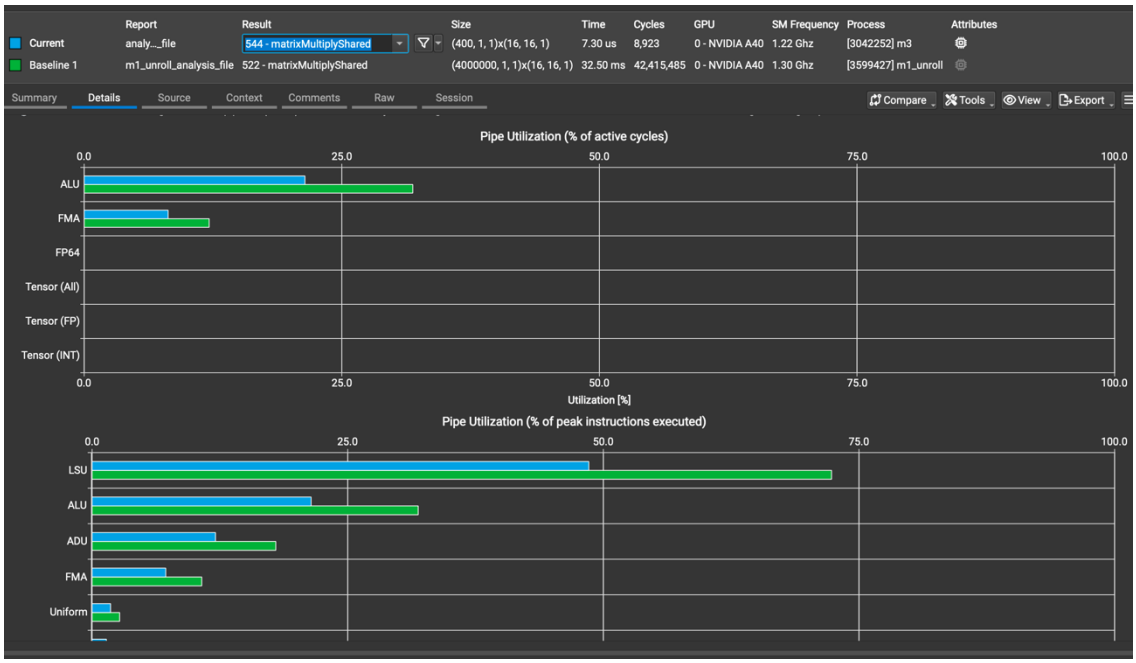
In contrast, the **baseline version** performs these steps sequentially—copying from host to device, running the kernels, and then copying back—without any overlap.

This overlapping behavior in the streams version reduces idle GPU time and improves overall throughput.

c. Did the performance match your expectation? Explain why or why not, by analyzing profiling results.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.0011 ms*	*0.0012 ms*	*1.423s*	*0.86*
1000	*0.0012 ms*	*0.00132 ms*	*9.216s*	*0.886*
10000	*0.0022 ms*	*0.0014 ms*	*1m28.688s*	*0.8714*

Yes, the performance generally matched our expectations. By using CUDA streams for parallelization, we were able to achieve overlap between memory transfers and kernel execution, which reduced overall operation time across different batch sizes. However, profiling results revealed a limitation: since we launched only one batch per stream, this led to a high number of kernel launches with relatively small workloads per kernel. As a result, Streaming Multiprocessors (SMs) were underutilized, leading to inefficient use of memory throughput and reduced performance scalability.



From the charts:

- Pipe Utilization (% of active cycles): The ALU and FMA units show relatively low utilization, indicating that the computational resources are underutilized. This aligns with our earlier observation that launching one batch per stream results in small workloads per kernel, failing to saturate the GPU's execution pipelines.
- Pipe Utilization (% of peak instructions executed): The LSU (Load/Store Unit) has significantly higher utilization compared to ALU and FMA, meaning the program is more memory-bound than compute-bound. This suggests a bottleneck due to frequent memory accesses with relatively little arithmetic intensity. It also reinforces that the SMs are not being used efficiently, especially for compute-heavy operations.

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

   Yes, CUDA streams synergize well with other optimizations because they focus on overlapping memory transfers and computation to keep the GPU busy. Techniques like loop unrolling, FP16 (__half2), __restrict__, and Tensor Cores reduce compute or memory time, which complements streams by making each kernel faster. As long as memory is handled correctly, these optimizations can be combined effectively.

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

   i. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/
   ii. Slides from Lecture 20
   iii. Chapter 13 in textbook

2. **Req_1: Using Tensor Cores to speed up matrix multiplication**

   [Google Drive link to subfolder req_1]
   https://drive.google.com/drive/folders/1ppGhUBvYC0jfOFYG9uDxGj7pbFIC7Yfm?usp=sharing

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

      i. This optimization uses Tensor Cores through the WMMA API to speed up matrix multiplication in the convolution kernel. Since Tensor Cores are designed for fast matrix operations, using them should make the kernel run faster.
      ii. By offloading matrix multiplication to Tensor Cores, we expect better performance, especially for large inputs where the GPU can fully take advantage of this hardware.

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

      i. We used CUDA Tensor Cores

         via wmma::fragment and wmma::mma_sync to accelerate the convolution. Shared memory tiles were loaded with TF32 inputs, and each 16×16 matrix multiplication was split into two 16×8 × 8×16 operations due to the 256-byte load limit of Tensor Cores.

ii. Shared memory tiles & Tensor Core fragments

```
// Shared memory tiles

extern __shared__ float share
float* tile_mask = shared;                                  // 16x8 = 128 floats
float* tile_input = tile_mask                                   // 8*16 = 128 floats
float* tile_C = tile_input + WMMA_K * WMMA_N;                  // 16*16 = 256 floats

// ==== Tensor Core multiply ====
wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, wmma::precision::tf32, wmma::row_major> a_frag;
wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, wmma::precision::tf32, wmma::row_major> b_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

wmma::fill_fragment(c_frag, 0.0f);
```

```
#define WMMA_K 8
Expands to:
8
```

iii. Each tile load–compute–accumulate cycle is done in a loop with synchronization:

```
for (int tileId = 0; tileId < (numAColumns - 1) / TILE_WIDTH + 1; tileId++) {
    for(int t=0;t<2;t++){
        // ==== Load A and B into shared memory ====
        size_t tiledIdx = tileId * TILE_WIDTH+WMMA_K*t; //tile_A*tile_B = 16*16 matrix
        dx_a = tx%8;
        dy_a = tx/8; // 0, 1, 2, 3, 4
        dx_b = tx%16;
        dy_b = tx/16; // 0, 1
        for(int i=0;i<4;i++){…
        __syncthreads();

        // tensor core matmul
        wmma::load_matrix_sync(a_frag, tile_mask, WMMA_K); // stride is full shared row size
        wmma::load_matrix_sync(b_frag, tile_input, WMMA_N);
        wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
        __syncthreads();
    }

}
wmma::store_matrix_sync(tile_C, c_frag, WMMA_N, wmma::mem_row_major);
__syncthreads();
```

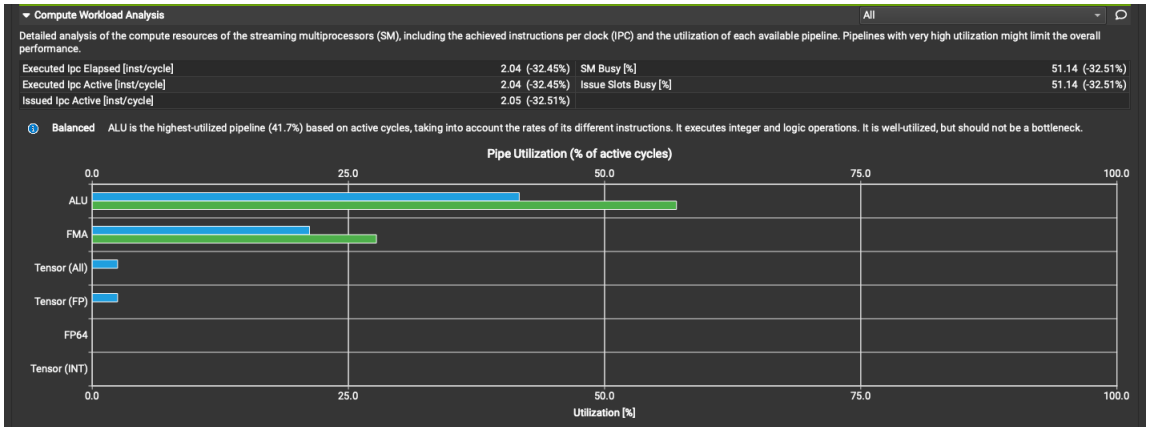
iv. Write back to global memory:

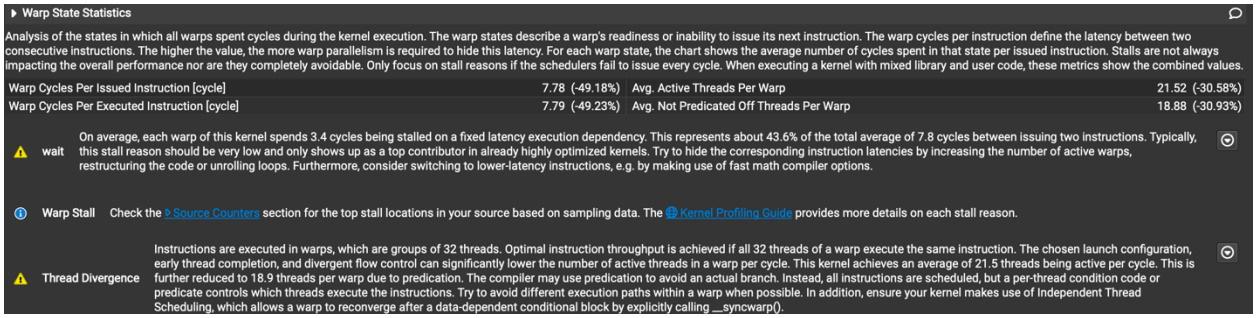
```
int dx = tx%16;
int dy = tx/16;
// 16*16/32 = 8
// each thread will need to store 8 elements to C
for(int i=0;i<8;i++){
    int row = block_row+dy*8+i;// represents m
    int col = block_col+dx;
    int b = col/image_size;
    int hw = col%image_size; // x in the unroll c
    int h = hw/Width_out;
    int w = hw%Width_out;
    if (row < numCRows && col < numCColumns) {
        output[b * Map_out * image_size + row * image_size + hw] = tile_C[(dy*8+i)*TILE_WIDTH+dx];
    }
}
```

v. Each thread splits the tile block into two parts and performs two matrix multiplications to compute the final result. At each step, fragments for A and B are set up, and an accumulator fragment (C) is initialized to zero. We use wmma::load_matrix_sync to load the fragments, perform the multiplication with wmma::mma_sync, and write the result back to shared memory using wmma::store_matrix_sync. Finally, the results are copied from shared memory to global memory, and at the end of the kernel, we copy the output back to the host.

- Tensor cores accelerate computation by handling mixed-precision FP16 operations, as shown by **0% FP32/FP64 usage**, confirming that they offload work from traditional ALUs.
- With **3.98 active warps per scheduler**, the kernel efficiently utilizes warps, maximizing parallelism and throughput.

Tensor cores reduce strain on ALUs and LSUs, optimizing memory and compute tasks, as seen in reduced LSU activity and balanced ALU utilization. This, along with high warp efficiency, confirms that tensor cores are essential for accelerating the workload.





c. Did the performance match your expectation? Explain why or why not, by analyzing profiling results.

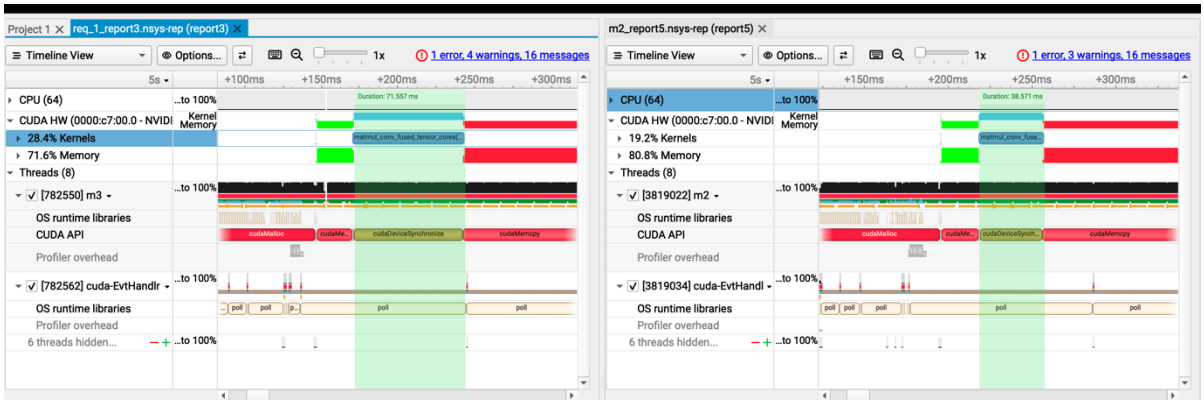
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.8250 ms*	*0.5465 ms*	*1.489s*	*0.86*
1000	*7.195 ms*	*4.7431 ms*	*9.597s*	*0.886*
10000	*70.99 ms*	*46.65 ms*	*1m33.193s*	*0.8714*

No, the performance only partially met our expectations. Although the individual operation time (OP Time) increased when using Tensor Cores — for example, at a batch size of 10,000, OP Time 1 and 2 totaled over 117 ms compared to the faster baseline kernel times — the total execution time was actually shorter, completing in 1m33.193s versus the slower original version.

This counterintuitive result can be explained by the more effective utilization of GPU resources in the Tensor Core version. Despite each operation taking longer — primarily due to

the need to split 16×16 tile matrix multiplications into two steps because of the 256-byte load limitation — overall throughput improved. This improvement stems from having more parallel work in flight and better occupancy of GPU pipelines across the larger workload.

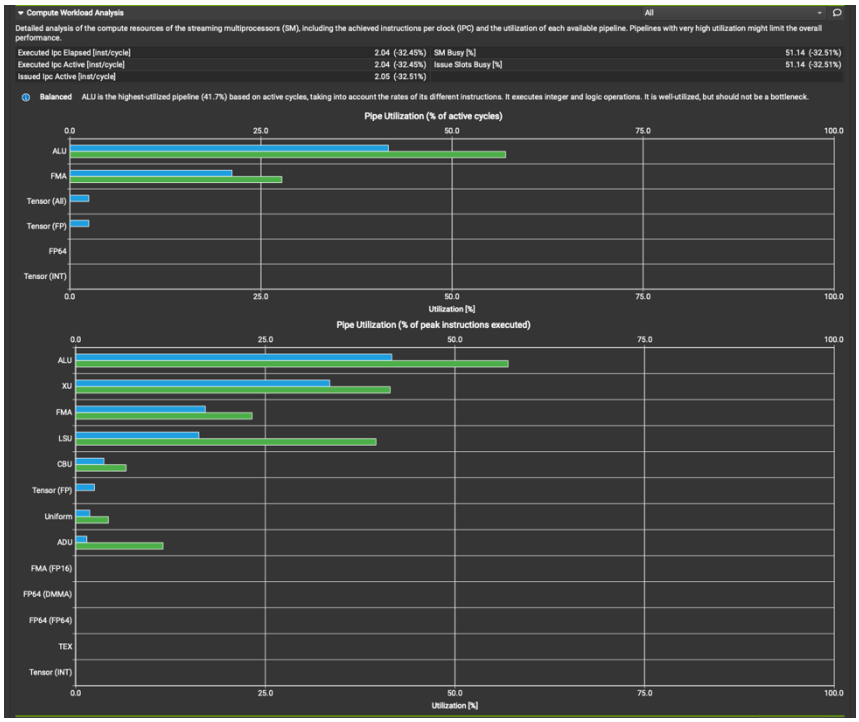
As shown, the kernel runtime for the Tensor Core version (70 ms) is nearly twice that of the original version (38 ms). This increase is due to the tile block being split into two parts, requiring two matrix multiplications to compute the final result.



From the profiling data:

- Tensor Core Activity: The kernel is confirmed to be using tensor cores, contributing to improved compute density.
- Lower LSU Utilization: Memory access overhead was reduced slightly, possibly because of fewer memory instructions or better-coalesced accesses, though some inefficiencies still

exist.



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

Yes, using tensor cores synergizes well with several other optimizations, including CUDA streams, FP16 precision, loop unrolling, and constant memory.

Most notably, combining tensor cores with FP16 (half precision) can significantly improve performance. Since FP16 values are half the size of FP32, we can load more data into shared memory and registers, and tensor cores can operate more efficiently — enabling direct 16×16 matrix multiplications without the need to split the computation, as we had to with FP32 due to the 256-byte load limit. This not only reduces kernel launch overhead but also increases compute throughput.

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

i. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9

ii. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html - warp-matrix-functions

iii. Lecture 22

3. Op_0: Weight matrix (Kernel) in constant memory

[Google Drive link to subfolder op_0]
https://drive.google.com/drive/folders/1aTwTXwR1h_MGK6CSJ5VA9HUFq-H5fjoi?usp=share_link

a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
    i. In this optimization, since the convolution mask remains the same every time, we can load it into constant memory instead of reading it from global memory repeatedly.
    ii. This reduces memory access time because constant memory is faster than global memory, especially when all threads access the same data.
b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.
    i. Because I find that the maximum mask size (bytes) will be 12544 bytes, I set MAX_CONST_MASK_SIZE to be 4000 > 12544/4 = 3136.

```
#define TILE_WIDTH 16
#define MAX_CONST_MASK_SIZE 4000
__constant__ float const_mask[MAX_CONST_MASK_SIZE];
```

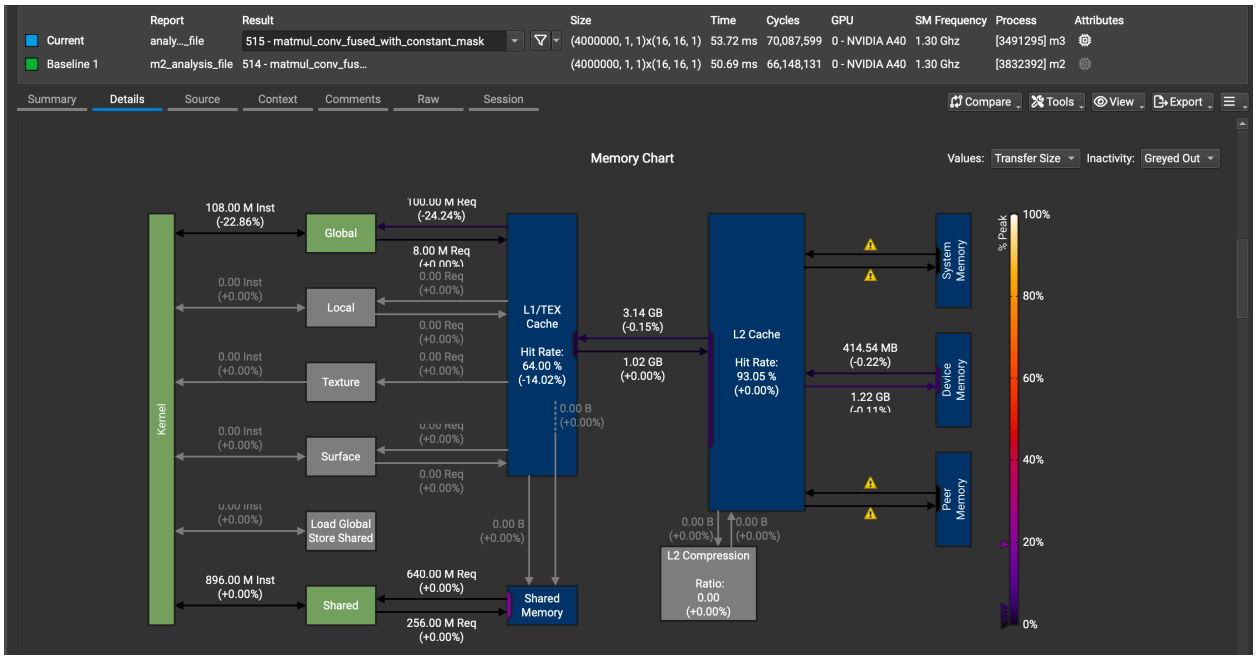
```
cudaError_t err = cudaMemcpyToSymbol(const_mask, host_mask, mask_size, 0,
cudaMemcpyHostToDevice);
```

```
#define mask_4d(i3, i2, i1, i0) const_mask[(i3) * (Channel * K * K) + (i2)
* (K * K) + (i1) * (K) + i0] // mask(m,c,p,q)
```

    ii. Load the mask into constant memory, and access the mask data through constant memory in kernel code.

From the memory chart, we can observe that the use of constant memory helped reduce the overall memory requests by 24.24% and global memory access by 22.86%. Additionally, it led to a reduction in global load requests handled by the L1 cache, indicating improved memory

efficiency and reduced pressure on global memory bandwidth.



L1/TEX Cache

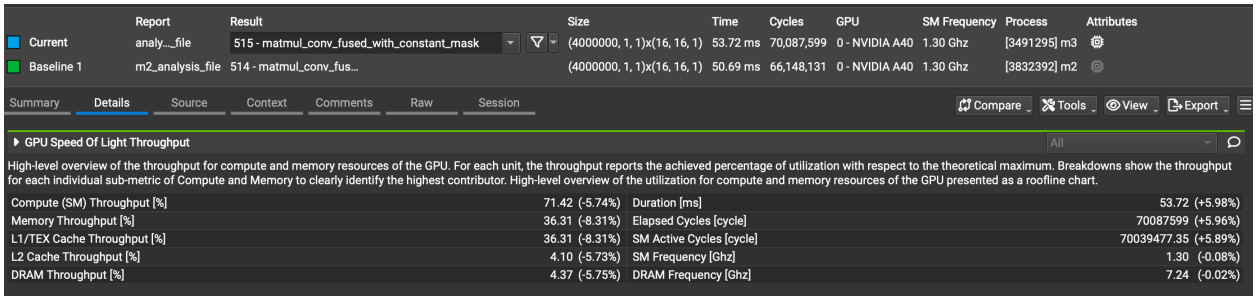
	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/Req	Hit Rate	Bytes
Local Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Global Load	100000000 (-24.24%)	100000000 (-24.24%)	108116648 (-22.84%)	1.84 (-27.13%)	328902958 (-31.04%)	3.29 (-8.97%)	70.23 (-11.59%)	10524894656 (-31.04%)
Global Load To Shared Store (access)	0 (+0.00%)	0 (+0.00%)			0 (+0.00%)	0 (+0.00%)		0 (+0.00%)
Global Load To Shared Store (bypass)	0 (+0.00%)	0 (+0.00%)			0 (+0.00%)	0 (+0.00%)	-	0 (+0.00%)
Surface Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Texture Load	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Global Store	8000000 (+0.00%)	8000000 (+0.00%)	10325312 (-0.00%)	0.18 (-5.56%)	32000000 (+0.00%)	4 (+0.00%)	0.01 (-0.37%)	1024000000 (+0.00%)
Local Store	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Surface Store	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Global Reduction	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
DSMEM Reduction						-	-	
Surface Reduction	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Global Atomic ALU	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Global Atomic CAS	0 (+0.00%)							
Surface Atomic ALU	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)	0 (+0.00%)
Surface Atomic CAS	0 (+0.00%)							

c. Did the performance match your expectation? Explain why or why not, by analyzing profiling results.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.4336 ms*	*0.5090 ms*	*3.349s*	*0.86*
1000	*4.064 ms*	*4.961 ms*	*9.899s*	*0.886*
10000	*40.33 ms*	*49.44 ms*	*1m37.076s*	*0.8714*

No, the performance did not fully match our expectations. We anticipated improvements in both operation time and total execution time, but the results were comparable to the original

version. Based on the profiling results, this appears to be due to compute throughput remaining the primary bottleneck, while the use of constant memory primarily benefits memory throughput. Since constant memory optimizes data access patterns but does not significantly impact computational intensity, the overall speedup was limited.



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

Yes, again we can use streams. Mixed precision and tensor cores are also not impacted from using restrict is also possible. As long as you make sure to constant memory is large enough.

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

i. Lecture 7

4. Op_1: __restrict__ keyword

[Google Drive link to subfolder op_1]
https://drive.google.com/drive/folders/1VCch_HLZwn38m-2vJPvpt-KNroTcw8FI?usp=share_link

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

i. By using the __restrict__ keyword, we tell the compiler that pointers do not point to overlapping memory. This allows the compiler to perform more aggressive optimizations, especially in memory access and instruction scheduling. Without __restrict__, the compiler must assume pointers might alias, which limits optimization and can slow down performance.

ii. We might not see any significant improvement after using __restrict__ keyword.

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

We simply add the restrict keyword to certain pointers in our function calls and when declaring them:

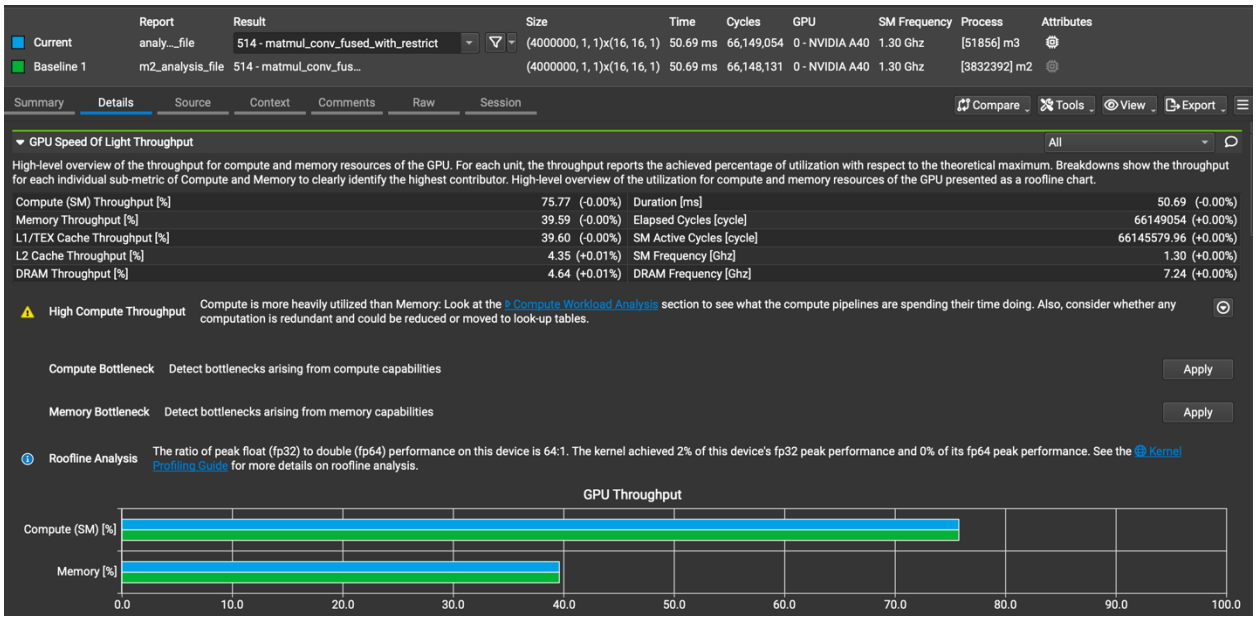
```
#define task "op_1"
#define TILE_WIDTH 16
__global__ void matmul_conv_fused_with_restrict(const float *__restrict__ mask,
                                                const float *__restrict__ input,
                                                float *__restrict__ output,
                                                int Batch, int Map_out, int Channel, int Height, int Width, int K)
{
```

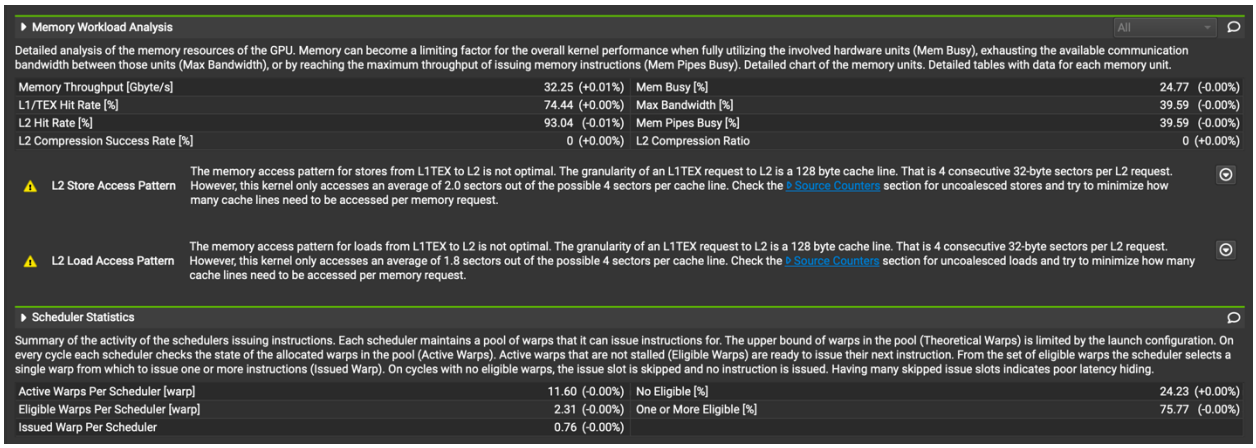
From profiling we see that nothing changed, however __restrict__ was being used. We know this because despite everything being near identical there are subtle differences in our optimes, and memory analysis values as shown in below screenshots. We know our implementation is correct because our duration actually went down negligibly but it does and our accuracy is maintained.

c. Did the performance match your expectation? Explain why or why not, by analyzing profiling results.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.4666 ms*	*0.3539 ms*	*1.434s*	*0.86*
1000	*3.8809 ms*	*3.37621 ms*	*9.403s*	*0.886*
10000	*38.502 ms*	*33.4809 ms*	*1m30.579s*	*0.8714*

As expected, we didn't observe any significant change in operation times. The number of registers used remained the same, and there was no noticeable difference in either Nsight System or Compute metrics. This might suggest that the compiler applied some low-level optimizations — but the impact appears negligible. Overall, any improvement in memory utilization is minimal. One possible explanation is that each major buffer was allocated separately using cudaMalloc, resulting in distinct memory regions with no aliasing. The compiler likely already recognizes this during compilation, which would render the use of __restrict__ redundant in this context.





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

Yes, again we can use streams. Mixed precision and tensor cores are also not impacted from using restrict is also possible. As long as you make sure to add the keyword into the function calls and in the declaration of your pointers.

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

i. https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

5. **Op_2: Loop unrolling**

   [Google Drive link to subfolder op_2]
   https://drive.google.com/drive/folders/1VJCmGILynCSRoJmOs08_f1Zt4_gU1b-K?usp=share_link

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

      i. Loop unrolling is an optimization where we manually expand the loop body to reduce loop control overhead and potentially increase instruction-level parallelism.

      ii. It can improve memory access patterns and reduce branch divergence in some cases. In theory, unrolling combined with shared memory could help reduce redundant loads. However, in our implementation, the practical benefits are minimal, and we only expect a slight or negligible improvement.

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

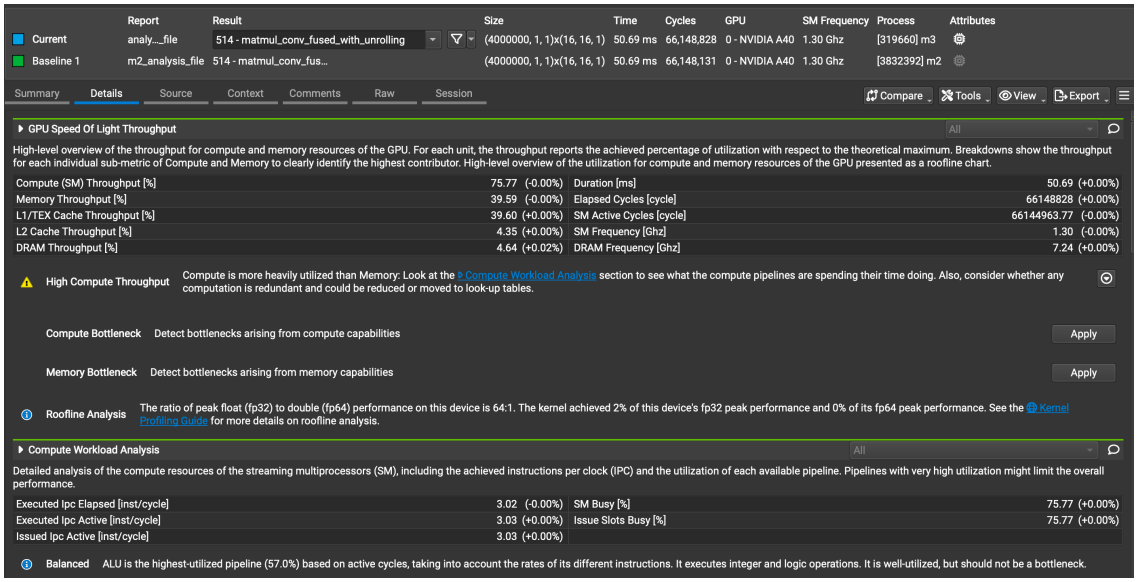
      i. We found every loop we had, unrolled it and then changed the increment of our loop accordingly:

```
if (row < numCRows && col < numCColumns) {
    for (int i = 0; i < TILE_WIDTH; i+=4) {
        val += tile_mask[ty][i] * tile_input[i][tx];
        val += tile_mask[ty][i+1] * tile_input[i+1][tx];
        val += tile_mask[ty][i+2] * tile_input[i+2][tx];
        val += tile_mask[ty][i+3] * tile_input[i+3][tx];
    }
}
__syncthreads();
```

We tried manual loop unrolling to optimize the inner tile computation loop, but profiling showed no improvement in execution time, operation time, or register usage.

This likely happened because:

- The loop is simple, so the compiler probably already unrolled it automatically.
- The loop bounds (TILE_WIDTH) are known at compile time, making manual unrolling redundant.
- Register usage stayed constant, and execution time didn't change, suggesting the bottleneck is elsewhere, like memory bandwidth.



c. Did the performance match your expectation? Explain why or why not, by analyzing profiling results.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.4666 ms*	*0.3539 ms*	*1.434s*	*0.86*
1000	*3.8809 ms*	*3.37621 ms*	*9.403s*	*0.886*
10000	*38.502 ms*	*33.4809 ms*	*1m30.579s*	*0.8714*

Yes, the performance matched our expectations. Based on the profiling results, the operation time, total execution time, and accuracy showed minimal changes. This aligns with our expectation because the optimizations we applied, like manual loop unrolling, did not provide significant improvements due to factors such as the simplicity of the loop and the compiler's internal optimizations. Consequently, the performance remained largely the same.

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

Yes, this optimization synergizes well with other optimizations like **streaming**, as it doesn't alter the distribution of work within a chunk. Additionally, since the data types remain unchanged, and also it is compatible with **FP16 (half-precision) computation**, which opens the door to leveraging **Tensor Cores** for further acceleration and improved memory efficiency.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
    i. The loop unrolling optimization section of the github project readme for milestone 3.
    ii. https://forums.developer.nvidia.com/t/understanding-unrolling-and-concurrent-memory-operations/38617

6. **Op_4: Using cuBLAS for matrix multiplication**

[Google Drive link to subfolder op_4]
https://drive.google.com/drive/folders/1lEn2sI6kt01GTBZP8OQuAbEJCHMG9vH4?usp=share_link

a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
    i. The CUDA Basic Linear Algebra Subprograms (cuBLAS) library is an optimized GPU-accelerated library for standard linear algebra operations like matrix multiplication. It's specifically tuned for NVIDIA GPUs to deliver high performance.
    ii. By using cuBLAS for our matrix multiplication (instead of a simple matmul version), we expect better computational efficiency and faster runtime due to its highly optimized implementation.
b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.
    i. To perform efficient matrix multiplication on the GPU, we utilized the cuBLAS library's cublasSgemm function and breaks into several steps.

ii. cuBLAS Setup

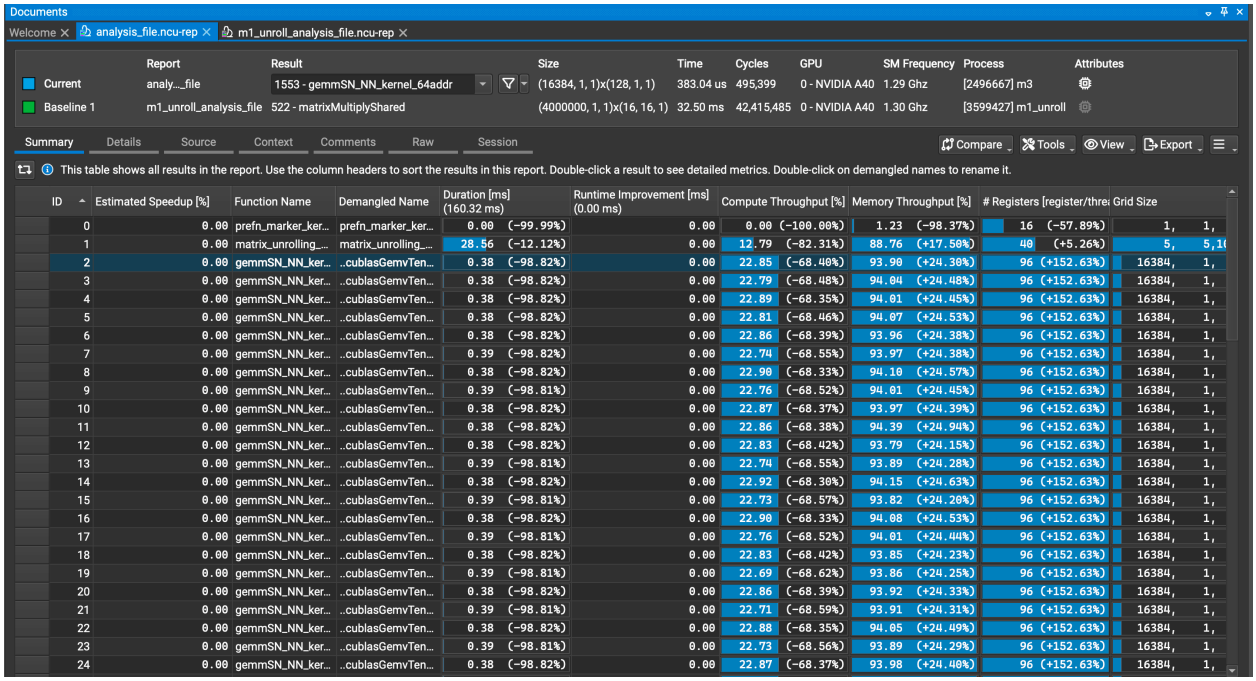
```
// Matrix multiplication and permutation.
// Using cuBLAS to do matmul
// Multiply the mask with the unrolled matrix

// cuBLAS variables
cublasHandle_t handle;
cublasCreate(&handle);

// A=Mask(row-major): Map_out*Height_unrolled
// B=Unrolled_Input(row-major): Height_unrolled*Width_unrolled
// C=Output: Map_out*Width_unrolled
// M = Map_out, N = Width_unrolled, K = Height_unrolled
```

```
float alpha = 1.0f;
float beta = 0.0f;
// C = alpha*A*B+beta*C
// C = A * B in row-major
// Matrices are passed in row-major format, but cuBLAS treats them as column-major.
// TC = TB*TA
```

iii. Matrix Multiplication Call

```
// Leading dimensions correspond to row widths in memory.
cublasSgemm(handle,
        CUBLAS_OP_N, CUBLAS_OP_N, // CUBLAS_OP_N indicates no transpose.
        Width_unrolled, Map_out, Height_unrolled,
        &alpha,
        unrolled_matrix, Width_unrolled,  // actually B, row-major
        device_mask, Height_unrolled,      // actually A, row-major
        &beta,
        matmul_output, Width_unrolled);    // C, stored row-major
```

The screenshot shows multiple invocations of gemmSN_NN_kernel_64addr, which is indeed the cuBLAS internal kernel being called for our matrix multiplication operation. This confirms our cuBLAS implementation is working correctly.



Documents

Welcome × analysis_file.ncu-rep × m1_unroll_analysis_file.ncu-rep ×

	Report	Result	Size	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	analy..._file	1553 - gemmSN_NN_kernel_64addr	(16384, 1, 1)x(128, 1, 1)	383.04 us	495,399	0 - NVIDIA A40	1.29 Ghz	[2496667] m3	
Baseline 1	m1_unroll_analysis_file	522 - matrixMultiplyShared	(4000000, 1, 1)x(16, 16, 1)	32.50 ms	42,415,485	0 - NVIDIA A40	1.30 Ghz	[3599427] m1_unroll	

Summary Details Source Context Comments Raw Session — Compare Tools View Export

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics. Double-click on demangled names to rename it.

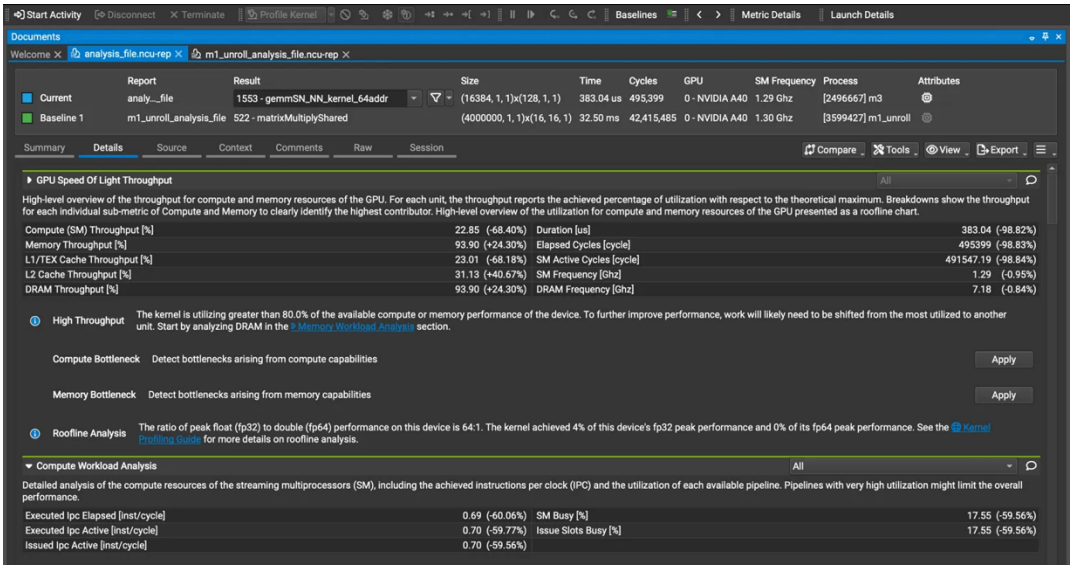
ID	Estimated Speedup [%]	Function Name	Demangled Name	Duration [ms] (160.32 ms)	Runtime Improvement [ms] (0.00 ms)	Compute Throughput [%]	Memory Throughput [%]	# Registers [register/thre	Grid Size
0	0.00	prefn_marker_ker...	prefn_marker_ker...	0.00 (−99.99%)	0.00	0.00 (−100.00%)	1.23 (−98.37%)	16 (−57.89%)	1, 1,
1	0.00	matrix_unrolling_...	matrix_unrolling_...	28.56 (−12.12%)	0.00	12.79 (−82.31%)	88.76 (+17.50%)	40 (+5.26%)	5, 5,1
2	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.85 (−68.40%)	93.90 (+24.30%)	96 (+152.63%)	16384, 1,
3	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.79 (−68.48%)	94.04 (+24.48%)	96 (+152.63%)	16384, 1,
4	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.89 (−68.35%)	94.01 (+24.45%)	96 (+152.63%)	16384, 1,
5	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.81 (−68.46%)	94.07 (+24.53%)	96 (+152.63%)	16384, 1,
6	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.86 (−68.39%)	93.96 (+24.38%)	96 (+152.63%)	16384, 1,
7	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.82%)	0.00	22.74 (−68.55%)	93.97 (+24.38%)	96 (+152.63%)	16384, 1,
8	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.90 (−68.33%)	94.10 (+24.57%)	96 (+152.63%)	16384, 1,
9	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.81%)	0.00	22.76 (−68.52%)	94.01 (+24.45%)	96 (+152.63%)	16384, 1,
10	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.87 (−68.37%)	93.97 (+24.39%)	96 (+152.63%)	16384, 1,
11	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.86 (−68.38%)	94.39 (+24.94%)	96 (+152.63%)	16384, 1,
12	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.83 (−68.42%)	93.79 (+24.15%)	96 (+152.63%)	16384, 1,
13	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.81%)	0.00	22.74 (−68.55%)	93.89 (+24.28%)	96 (+152.63%)	16384, 1,
14	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.92 (−68.30%)	94.15 (+24.63%)	96 (+152.63%)	16384, 1,
15	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.81%)	0.00	22.73 (−68.57%)	93.82 (+24.20%)	96 (+152.63%)	16384, 1,
16	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.90 (−68.33%)	94.08 (+24.53%)	96 (+152.63%)	16384, 1,
17	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.81%)	0.00	22.76 (−68.52%)	94.01 (+24.44%)	96 (+152.63%)	16384, 1,
18	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.83 (−68.42%)	93.85 (+24.23%)	96 (+152.63%)	16384, 1,
19	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.81%)	0.00	22.69 (−68.62%)	93.86 (+24.25%)	96 (+152.63%)	16384, 1,
20	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.86 (−68.39%)	93.92 (+24.33%)	96 (+152.63%)	16384, 1,
21	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.81%)	0.00	22.71 (−68.59%)	93.91 (+24.31%)	96 (+152.63%)	16384, 1,
22	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.88 (−68.35%)	94.05 (+24.49%)	96 (+152.63%)	16384, 1,
23	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.39 (−98.81%)	0.00	22.73 (−68.56%)	93.89 (+24.29%)	96 (+152.63%)	16384, 1,
24	0.00	gemmSN_NN_ker...	..cublasGemvTen...	0.38 (−98.82%)	0.00	22.87 (−68.37%)	93.98 (+24.40%)	96 (+152.63%)	16384, 1,

c. Did the performance match your expectation? Explain why or why not, by analyzing profiling results.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*19.47 ms*	*1.3960 ms*	*3.434s*	*0.86*
1000	*25.21 ms*	*13.2489 ms*	*11.040s*	*0.886*
10000	*67.514 ms*	*108.119 ms*	*1m35.435s*	*0.8714*

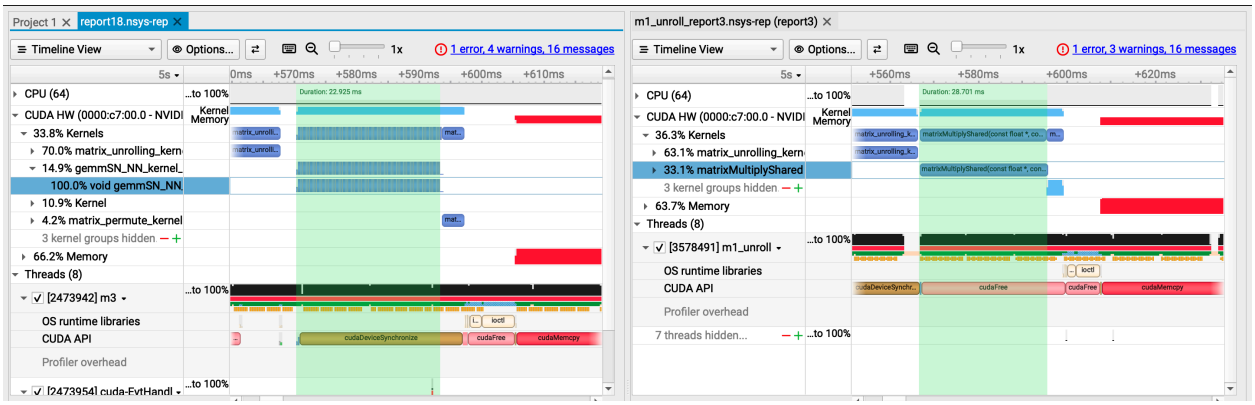
Not really. While I expected all batch sizes to show improved performance with cuBLAS, the results reveal a more nuanced picture. For smaller batches (100, 1000), the cuBLAS implementation actually performed worse than the baseline (19.47ms vs 1.47ms for batch size 100). Only at larger batch sizes (10000) did cuBLAS begin to show modest improvements.

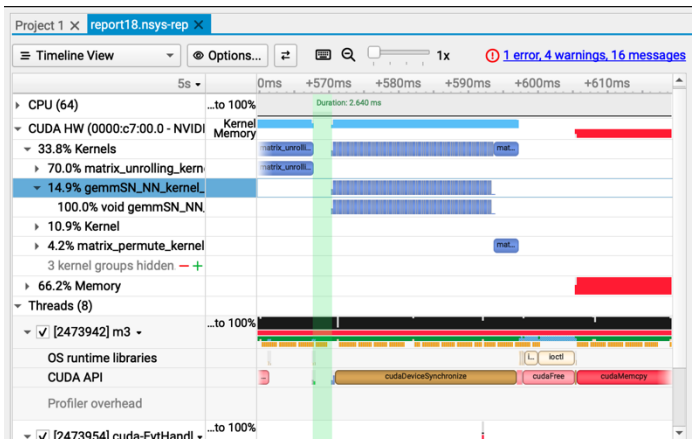
The profiling data explains this discrepancy. The cuBLAS implementation shows high memory throughput (93.90%, +24.30%) but low compute throughput (22.85%, -68.40%), indicating the operation is memory-bound rather than compute-bound. While the core matrix multiplication kernel shows significant speed improvement (383.04µs vs 32.50ms), this advantage is offset by initialization overhead and memory transfer costs that dominate with smaller workloads.



The timeline view confirms significant time(2ms) spent in API calls and setup for cuBLAS, which becomes proportionally less significant as batch size increases. This explains why only at batch size 10000 does cuBLAS begin to outperform the

baseline, with the core multiplication taking 22ms compared to the baseline's 28ms.





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

Yes, this optimization synergizes well with other optimizations. For example, by using FP16 to reduce memory throughput requirements, it complements cuBLAS, which enhances compute throughput by efficiently handling mixed-precision operations. The profiling data shows our implementation is memory-bound (93.90% memory throughput) rather than compute-bound (22.85% compute throughput), making this combination particularly effective.

Together, these optimizations address both sides of the performance equation - FP16 reduces memory bandwidth pressure while cuBLAS maximizes computational efficiency through vendor-optimized kernels.

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

i. All readings from the project github milestone 3 section op_4 cuBLAS

ii. Peers report slides about cuBLAS.

7. **Op_5: FP16(using __half2)**

   [Google Drive link to subfolder op_5]
   https://drive.google.com/drive/folders/1UewvodqdPwvmnDzklYl8R2SgeBzwhTpN?usp=share_link

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

      i. Since FP16 is half the size of FP32, it allows for better memory utilization and faster data movement. Using __half2, which packs two FP16 values into a single 32-bit register, further improves data transfer efficiency. This helps reduce memory bandwidth usage, a common bottleneck in CUDA kernels, and potentially doubles bandwidth utilization.

      ii. We also expect faster kernel execution due to reduced precision, better memory throughput, and improved compute utilization.

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

      i. Using __half2 Data Type

```
// Shared memory tiles
__shared__ __half2 tile_mask[TILE_WIDTH][TILE_WIDTH]; // A
__shared__ __half2 tile_input[TILE_WIDTH][TILE_WIDTH]; // B
```

ii. Loading Data into __half2 Format

```
size_t tiledIdx = tileId * TILE_WIDTH * 2; // Each __half2 handles 2 values
// Load tile_mask
if (row < numARows && tiledIdx + tx * 2 < numAColumns) {
    size_t idx = tiledIdx + tx * 2;
    int c = idx / KK;
    int pq = idx % KK;
    int p = pq / K;
    int q = pq % K;

    // Load two consecutive values into __half2
    float val0, val1;
    if (idx < numAColumns) val0 = mask_4d(row, c, p, q);
    else val0 = 0.0f;

    if (idx + 1 < numAColumns) val1 = mask_4d(row, c + (pq + 1) / KK, (pq + 1) % KK / K, (pq + 1) % K);
    else val1 = 0.0f;

    tile_mask[ty][tx] = __floats2half2_rn(val0, val1);
} else {
    tile_mask[ty][tx] = __floats2half2_rn(0.0f, 0.0f);
}
```

iii. Using FP16 Arithmetic for Computation

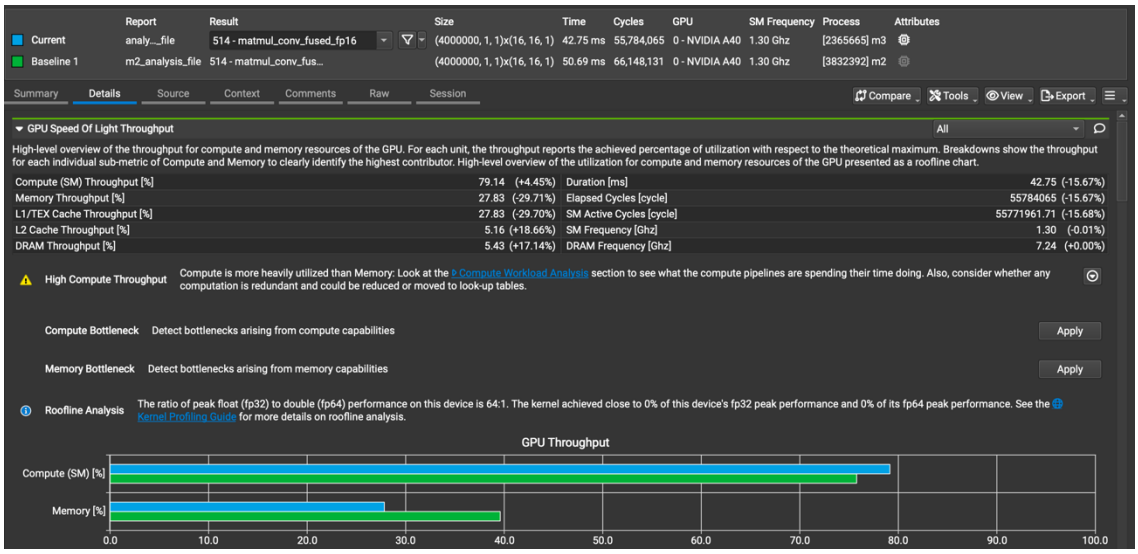
```
// Compute with __half2 operations
if (row < numCRows && col < numCColumns) {
    for (int i = 0; i < TILE_WIDTH; i++) {
        val = __hadd2(val, __hmul2(tile_mask[ty][i], tile_input[i][tx]));
    }
}
__syncthreads();
```

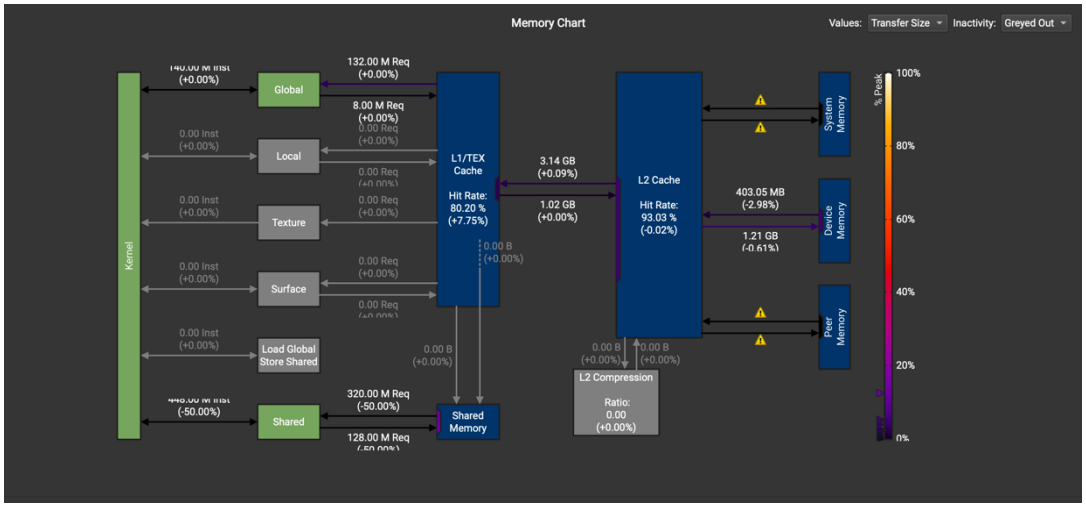
iv. Storing Results

```
// Write output
if (row < numCRows && col < numCColumns) {
    // Sum the two __half values in val to get the final result
    float result = __half2float(val.x) + __half2float(val.y);
    output[b * Map_out * image_size + row * image_size + hw] = result;
}
```

The profiling results validate the correctness of FP16 optimization:

- **Memory Throughput (-29.71%)**: Using __half2 reduces memory usage by packing two FP16 values into one register, cutting memory throughput by nearly 30%, leading to faster memory access.
- **L1 Cache Hit Rate (+7.75%)**: The smaller size of FP16 values improves cache efficiency, resulting in a 7.75% increase in L1 cache hits and reducing the need for slower global memory accesses.
- **Shared Memory Loading (-50%)**: Loading FP16 values (half the size of FP32) into shared memory reduces data transfer by 50%, improving efficiency.



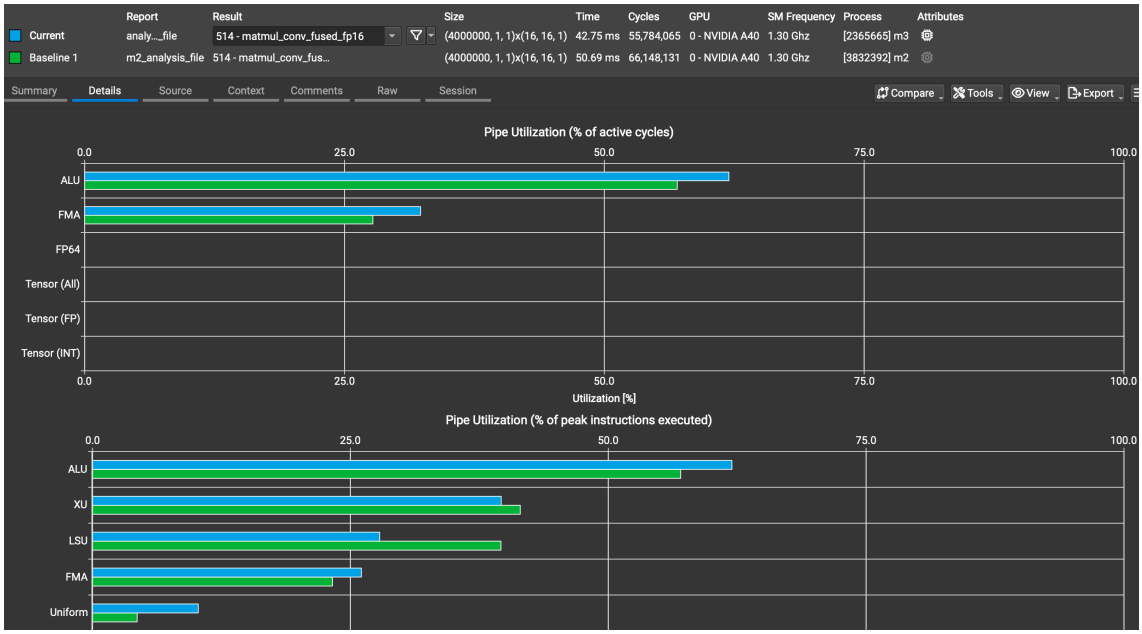


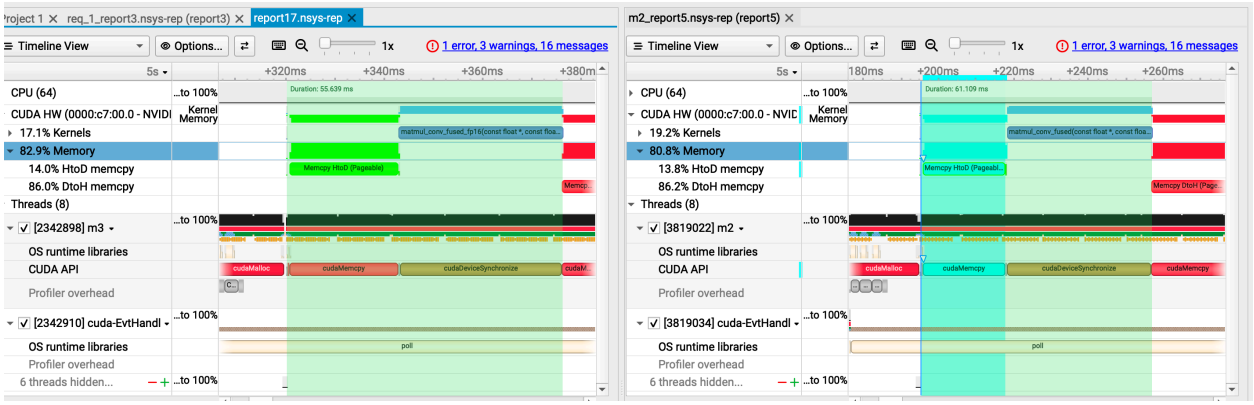
c. Did the performance match your expectation? Explain why or why not, by analyzing profiling results.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.3996 ms*	*0.305523 ms*	*3.573s*	*0.86*
1000	*3.2896 ms*	*2.89146 ms*	*10.158s*	*0.886*
10000	*32.484 ms*	*28.6796 ms*	*1m30.572s*	*0.8713*

Yes, the performance met our expectations. Both **operation time** and **total execution time** are faster than the baseline. Additionally, we see **better ALU utilization** and **lower LSU utilization**, confirming the expectation of **improved memory throughput** and **better compute utilization**.

The **runtime (55ms)** for **memcpy and kernel computing** is also faster than the baseline's **61ms**, further indicating that the optimizations are effectively reducing overhead and enhancing performance.





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

Yes, this optimization synergizes well with other optimizations. By using FP16 to reduce memory throughput, it complements tensor cores, which enhance compute throughput by efficiently handling mixed-precision operations.

Together, they address memory bottlenecks while improving computational efficiency. Additionally, optimizations like loop unrolling, constant memory, the restrict keyword, and CUDA streams can further enhance performance.

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

i. All readings from the project github milestone 3 section op_5 FP16