Vanya Zhadovich

Milestone 3 Report

0. Baseline:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.189199 ms*	*0.515635 ms*	*0m4.078s*	*0.86*
1000	*1.20908 ms*	*5.04132 ms*	*0m36.842s*	*0.886*
10000	*11.5888 ms*	*53.6658 ms*	*6m10.192s*	*0.8714*

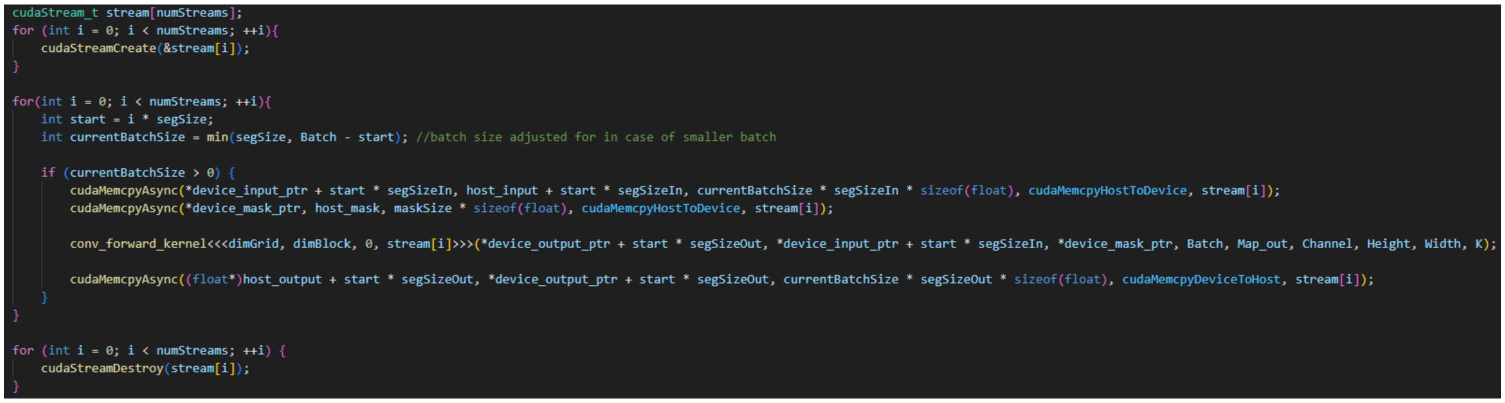
1. Optimization Number #0: Streams

a. How does this optimization work in theory? Expected behavior?

This optimization (streams) allows for concurrent execution of kernel computations and memory transfers. By using streams, data transfers can be overlapped during kernel executions which can reduce op times. The streams can operate in separate queues at once. Op times may improve with streams as data transfers overlap with kernel execution reducing memory latency, but it may not be significant.

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

Multiple streams are created to manage parallel execution of data transfers and kernel launches using cudaStreamCreate. The input data, mask, and outputs are managed in segments to facilitate batch processing. Each stream handles a segment of the batch, and this is done by using offsets for the input and output pointers. In this code, there are 4 streams that divide the batch up. Asynchronous memory transfers (cudaMemcpyAsync) and kernel executions are scheduled on different streams. After all operations are scheduled, the streams are destroyed.



```
cudaStream_t stream[numStreams];
for (int i = 0; i < numStreams; ++i){
    cudaStreamCreate(&stream[i]);
}

for(int i = 0; i < numStreams; ++i){
    int start = i * segSize;
    int currentBatchSize = min(segSize, Batch - start); //batch size adjusted for in case of smaller batch

    if (currentBatchSize > 0) {
        cudaMemcpyAsync(*device_input_ptr + start * segSizeIn, host_input + start * segSizeIn, currentBatchSize * segSizeIn * sizeof(float), cudaMemcpyHostToDevice, stream[i]);
        cudaMemcpyAsync(*device_mask_ptr, host_mask, maskSize * sizeof(float), cudaMemcpyHostToDevice, stream[i]);

        conv_forward_kernel<<<dimGrid, dimBlock, 0, stream[i]>>>(*device_output_ptr + start * segSizeOut, *device_input_ptr + start * segSizeIn, *device_mask_ptr, Batch, Map_out, Channel, Height, Width, K);

        cudaMemcpyAsync((float*)host_output + start * segSizeOut, *device_output_ptr + start * segSizeOut, currentBatchSize * segSizeOut * sizeof(float), cudaMemcpyDeviceToHost, stream[i]);
    }
}

for (int i = 0; i < numStreams; ++i) {
    cudaStreamDestroy(stream[i]);
}
```

Fig. – Code snippet of stream usage for copying memory and kernel launches

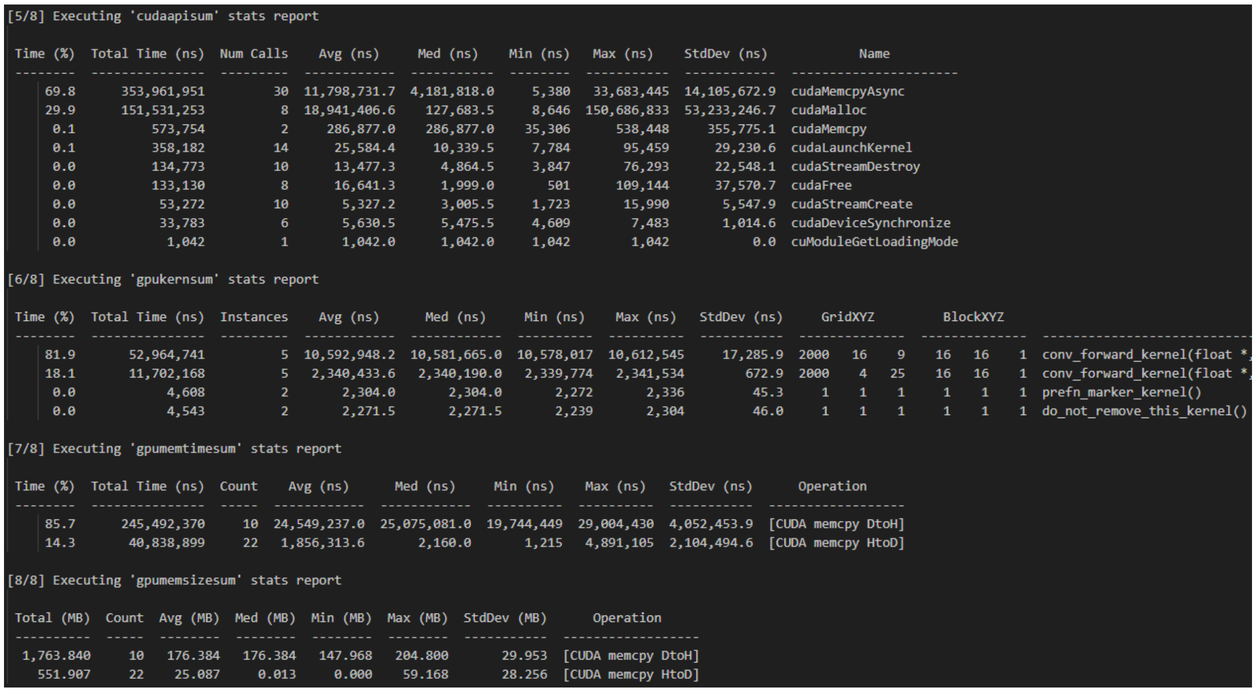
c. Did the performance match your expectation? Show your analysis results using profiling tools.

*Layer Times are recorded here due to streams being written in conv_forward_gpu_prolog

Batch Size	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	*2.89909 ms*	*2.96913 ms*	*0m4.062s*	*0.86*
1000	*20.6266 ms*	*18.9936 ms*	*0m37.565s*	*0.886*
10000	*177.632 ms*	*181.944 ms*	*6m5.448s*	*0.8714*

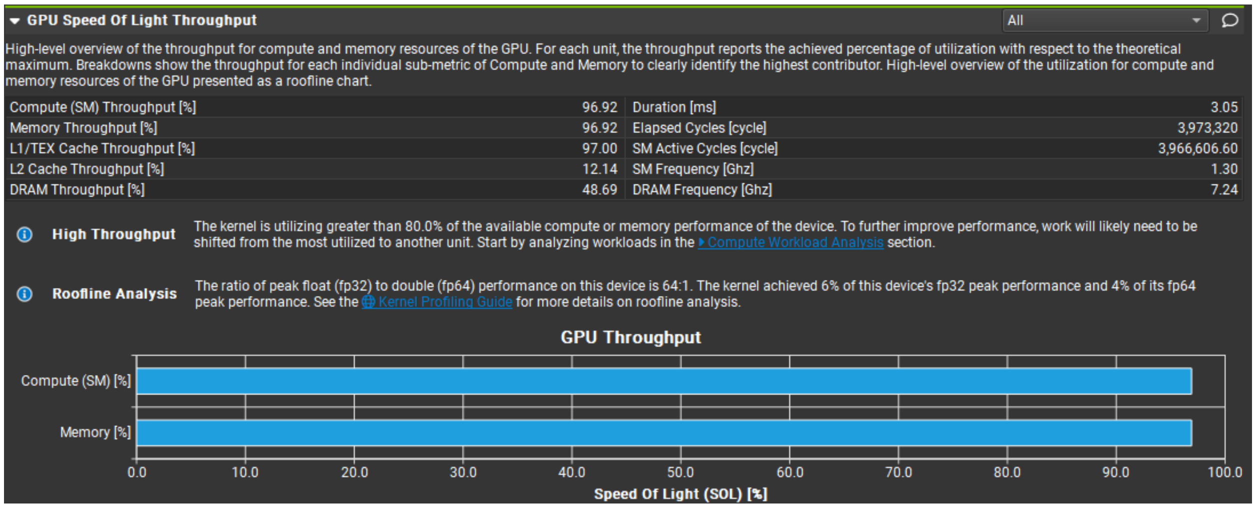
While it is difficult to identify if streams improve the code, we can identify op times through looking at the total kernal time in the nsys profiling below. For a batch of 10,000, the first layer the op time is 52,964,741ns and for the second layer, the op time is 11,702,168ns. Comparing these results closely to the op times for the baseline, it can be noted that there is no improvement. The lack of improvement can be explained by the baseline code being the same, and the only time is saved on copying memory, which is negligible in this case, The noticeable improvement in the nsys profiling is the large reduction in cudaapisum where cudaDeviceSynchronize nearly consumes no time. Compared to other profiles, cudaDeviceSynchronize was always in the top three of largest times in cudaapisum. Also, in thegpumemtimesum. The overall time is lower compared to other optimizations.

Nsight compute shows that the GPU throughput is very high and balanced. This is expected as the forward kernel code does not change. The code can improve with its memory utilization as the L2 load and store access pattern is not optimal show in the Nsight memory workload analysis below. Making sure that memory accesses take advantage of coalescing can improve performance.



```
[5/8] Executing 'cudaapisum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)     Med (ns)    Min (ns)   Max (ns)    StdDev (ns)           Name
 --------  ---------------  ---------  ------------  -----------  --------  -----------  ------------  ----------------------
     69.8      353,961,951         30  11,798,731.7  4,181,818.0     5,380   33,683,445  14,105,672.9  cudaMemcpyAsync
     29.9      151,531,253          8  18,941,406.6    127,683.5     8,646  150,686,833  53,233,246.7  cudaMalloc
      0.1          573,754          2     286,877.0    286,877.0    35,306      538,448     355,775.1  cudaMemcpy
      0.1          358,182         14      25,584.4     10,339.5     7,784       95,459      29,230.6  cudaLaunchKernel
      0.0          134,773         10      13,477.3      4,864.5     3,847       76,293      22,548.1  cudaStreamDestroy
      0.0          133,130          8      16,641.3      1,999.0       501      109,144      37,570.7  cudaFree
      0.0           53,272         10       5,327.2      3,005.5     1,723       15,990       5,547.9  cudaStreamCreate
      0.0           33,783          6       5,630.5      5,475.5     4,609        7,483       1,014.6  cudaDeviceSynchronize
      0.0            1,042          1       1,042.0      1,042.0     1,042        1,042           0.0  cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)     GridXYZ         BlockXYZ
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  --------------  --------------  ----------------------------
     81.9       52,964,741          5  10,592,948.2  10,581,665.0  10,578,017  10,612,545     17,285.9  2000    16     9    16    16     1  conv_forward_kernel(float *,
     18.1       11,702,168          5   2,340,433.6   2,340,190.0   2,339,774   2,341,534        672.9  2000     4    25    16    16     1  conv_forward_kernel(float *,
      0.0            4,608          2       2,304.0       2,304.0       2,272       2,336         45.3     1     1     1     1     1     1  prefn_marker_kernel()
      0.0            4,543          2       2,271.5       2,271.5       2,239       2,304         46.0     1     1     1     1     1     1  do_not_remove_this_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)      Operation
 --------  ---------------  -----  ------------  ------------  ----------  ----------  -----------  ------------------
     85.7      245,492,370     10  24,549,237.0  25,075,081.0  19,744,449  29,004,430  4,052,453.9  [CUDA memcpy DtoH]
     14.3       40,838,899     22   1,856,313.6       2,160.0       1,215   4,891,105  2,104,494.6  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  --------  -----------  ------------------
  1,763.840     10   176.384   176.384   147.968   204.800       29.953  [CUDA memcpy DtoH]
    551.907     22    25.087     0.013     0.000    59.168       28.256  [CUDA memcpy HtoD]
```

Fig. – nsys result 5-8 for Streams


GPU Speed Of Light Throughput

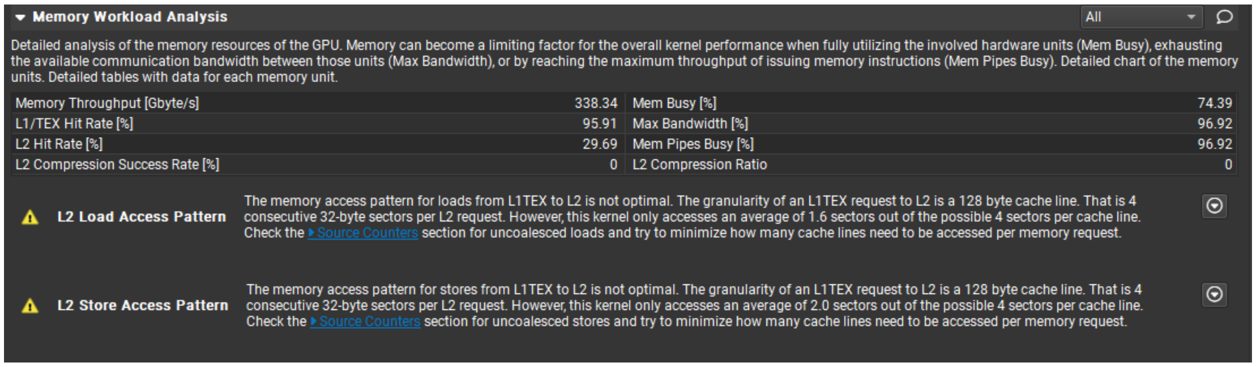
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	96.92	Duration [ms]	3.05
Memory Throughput [%]	96.92	Elapsed Cycles [cycle]	3,973,320
L1/TEX Cache Throughput [%]	97.00	SM Active Cycles [cycle]	3,966,606.60
L2 Cache Throughput [%]	12.14	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	48.69	DRAM Frequency [Ghz]	7.24

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 6% of this device's fp32 peak performance and 4% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. – Streams GPU SOL Throughput


Memory Workload Analysis

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]	338.34	Mem Busy [%]	74.39
L1/TEX Hit Rate [%]	95.91	Max Bandwidth [%]	96.92
L2 Hit Rate [%]	29.69	Mem Pipes Busy [%]	96.92
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L2 Load Access Pattern The memory access pattern for loads from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.6 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced loads and try to minimize how many cache lines need to be accessed per memory request.

L2 Store Access Pattern The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 2.0 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

Fig. – Memory workload analysis for Streams

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

Using constant memory with streams optimizes performance by reducing memory bandwidth and improving data access speed. Constant memory provides fast access to data across multiple streams, enhancing efficiency and throughput in parallel kernel executions. Using constant memory can eliminate the need for multiple copies of the mask, streamlining memory usage. Additionally, employing streams can further optimize performance by reducing data transfer times for optimizations that may need multiple kernel launches such as input matrix unrolling & tiled multiplication using shared memory.

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

i. Slide Deck 20: GPU Data Transfer

ii. Cuda C++ Programming Guide, Release 12.3 – p. 48

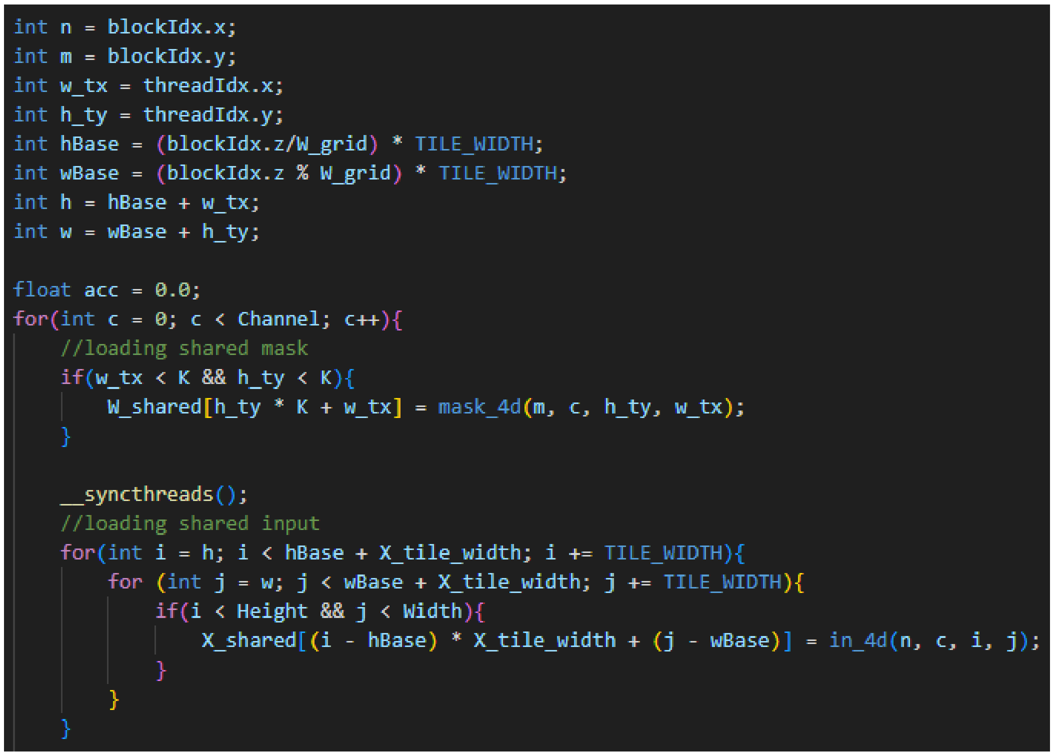
2. Optimization Number #1: Tiled (Shared Memory) Convolution

a. How does this optimization work in theory? Expected behavior?

Tiling, or using shared memory for convolution, is an optimization technique to reduce global memory accesses by loading chunks of the input data and kernel into faster shared memory. Shared memory is much faster than global memory, but its size is limited, so it's used to store smaller tiles of the input and kernel that are then used to compute a portion of the output. By loading data into shared memory once and reusing it for multiple computations, we minimize the number of slow global memory accesses. With the use of shared memory, reduced global memory access and increased throughput can be expected.

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

The implementation leverages shared memory to store input tiles and convolution masks, reducing the number of global memory accesses. Below is an image of the code where the tiles in shared memory are loading the kernel and input. After loading the necessary input elements, the convolution is performed and written back to global memory.



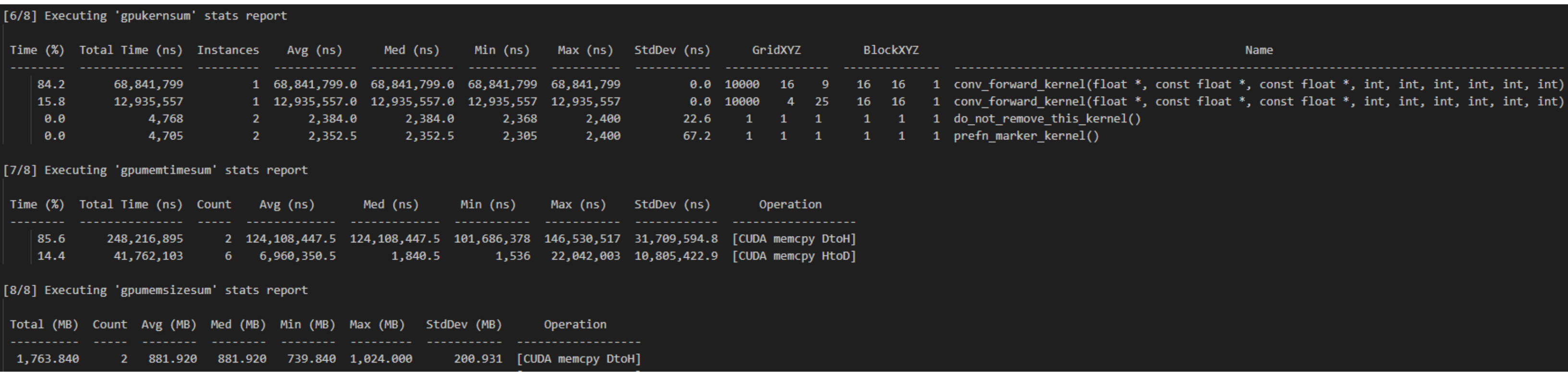
```
int n = blockIdx.x;
int m = blockIdx.y;
int w_tx = threadIdx.x;
int h_ty = threadIdx.y;
int hBase = (blockIdx.z/W_grid) * TILE_WIDTH;
int wBase = (blockIdx.z % W_grid) * TILE_WIDTH;
int h = hBase + w_tx;
int w = wBase + h_ty;

float acc = 0.0;
for(int c = 0; c < Channel; c++){
    //loading shared mask
    if(w_tx < K && h_ty < K){
        W_shared[h_ty * K + w_tx] = mask_4d(m, c, h_ty, w_tx);
    }

    __syncthreads();
    //loading shared input
    for(int i = h; i < hBase + X_tile_width; i += TILE_WIDTH){
        for (int j = w; j < wBase + X_tile_width; j += TILE_WIDTH){
            if(i < Height && j < Width){
                X_shared[(i - hBase) * X_tile_width + (j - wBase)] = in_4d(n, c, i, j);
            }
        }
    }
```

Fig. – Code snippet of loading shared tiles with inputs

c. Did the performance match your expectation? Show your analysis results using profiling tools.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.191784 ms*	*0.707698 ms*	*0m4.096s*	*0.86*
1000	*1.33759 ms*	*6.90021 ms*	*0m37.074s*	*0.886*
10000	*13.0311 ms*	*68.7664 ms*	*6m3.390s*	*0.8714*

As seen by the op times, this optimization increases the run time. This is the opposite of the expected behavior. The overall run time increases as time is spent copying over the data to shared memory and bank conflicts. Looking at the nsys profiling results below, it can be noticed that the conv_forward_kernel runs for a greater amount of time, compared to successful optimizations.



```
[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)      GridXYZ         BlockXYZ                                          Name
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ---------------  --------------  ------------------------------------------------------------------------------------------------
     84.2       68,841,799          1  68,841,799.0  68,841,799.0  68,841,799  68,841,799          0.0  10000    16     9    16    16     1  conv_forward_kernel(float *, const float *, const float *, int, int, int, int, int, int)
     15.8       12,935,557          1  12,935,557.0  12,935,557.0  12,935,557  12,935,557          0.0  10000     4    25    16    16     1  conv_forward_kernel(float *, const float *, const float *, int, int, int, int, int, int)
      0.0            4,768          2       2,384.0       2,384.0       2,368       2,400         22.6      1     1     1     1     1     1  do_not_remove_this_kernel()
      0.0            4,705          2       2,352.5       2,352.5       2,305       2,400         67.2      1     1     1     1     1     1  prefn_marker_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)       Med (ns)      Min (ns)     Max (ns)    StdDev (ns)       Operation
 --------  ---------------  -----  -------------  -------------  -----------  -----------  ------------  ------------------
     85.6      248,216,895      2  124,108,447.5  124,108,447.5  101,686,378  146,530,517  31,709,594.8  [CUDA memcpy DtoH]
     14.4       41,762,103      6    6,960,350.5        1,840.5        1,536   22,042,003  10,805,422.9  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)   StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  ---------  -----------  ------------------
  1,763.840      2   881.920   881.920   739.840  1,024.000      200.931  [CUDA memcpy DtoH]
```

Fig. – nsys results 6-8 for Tiled (Shared Memory) Convolution

When analyzing in Nsight Compute, GPU throughput slightly decreases from the baseline as seen below.

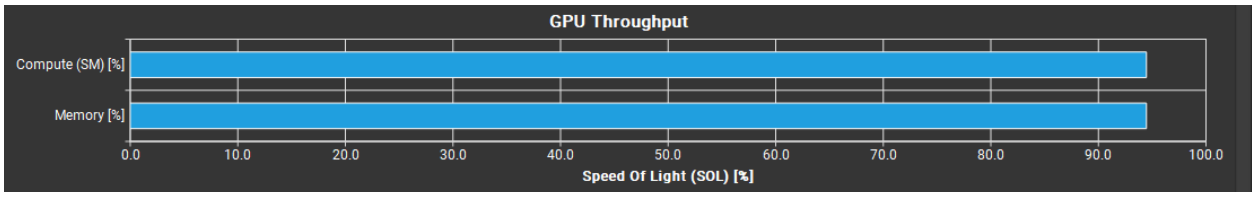


Fig. – Tiled (Shared Memory) Convolution GPU SOL Throughput

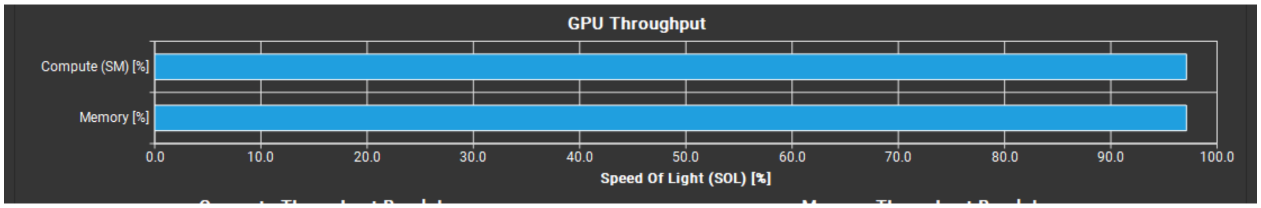
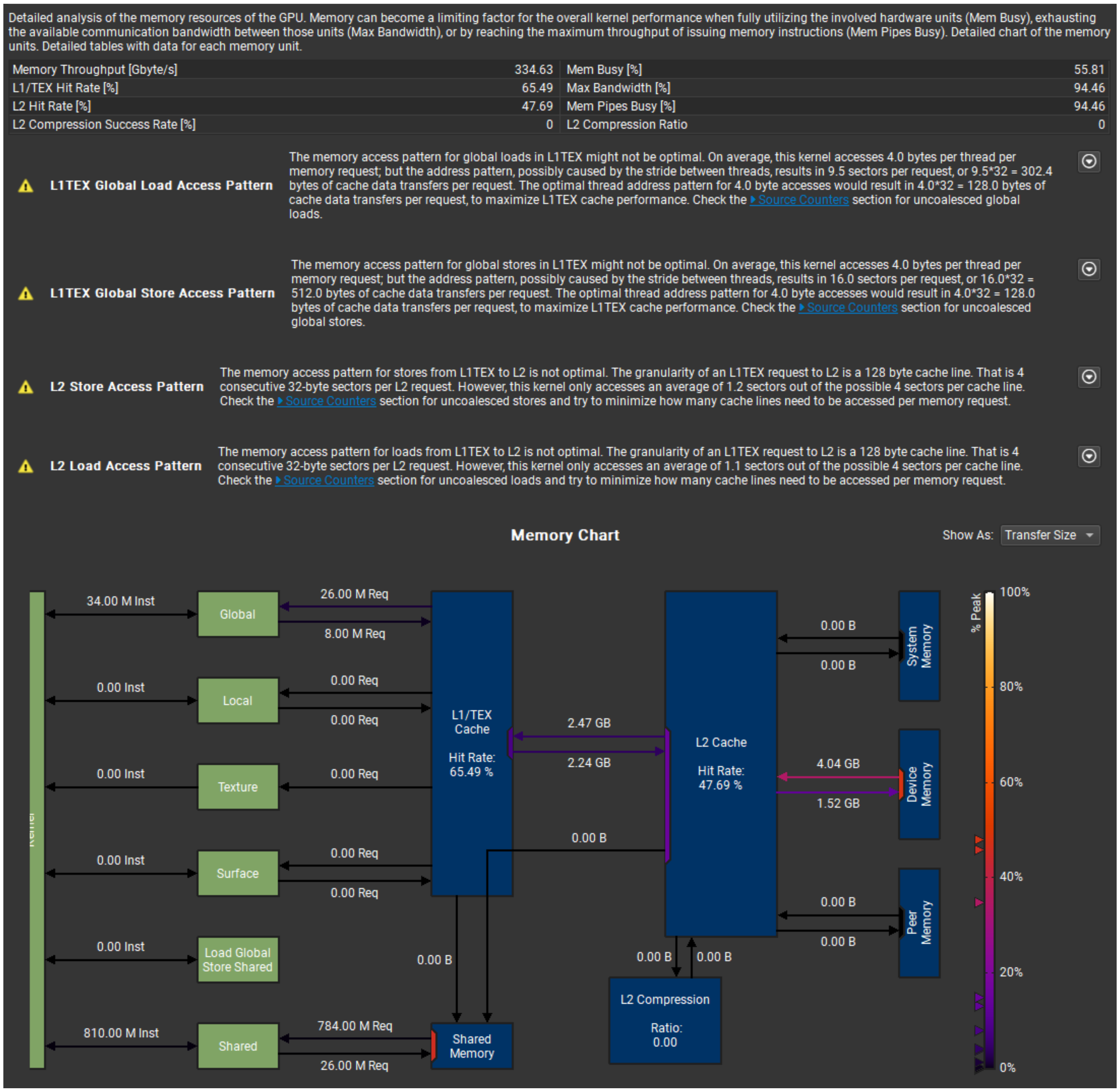


Fig. – Baseline GPU SOL Throughput

Another issue can be found in the L1TEX global load/store access pattern and the L2 load/store access pattern as seen below. Looking at the prompts next to the yellow icons, is the explanation for why there are problems with the access patterns for the L1TEX and L2. To improve functionality, global memory accesses should accommodate to the number of bytes in a burst. Additionally, in the figure below showing data on shared memory and L1/TEX Cache, it is visible that there are 7,224,461 bank conflicts and misses to the L2 cache. By optimizing global memory accesses, these values can be reduced. With the large count of bank conflicts within the shared memory, run time slows down which eliminates the benefit of using shared memory. Improving memory access patterns in this optimization could possibly lead to a benefit.



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]	334.63	Mem Busy [%]	55.81
L1/TEX Hit Rate [%]	65.49	Max Bandwidth [%]	94.46
L2 Hit Rate [%]	47.69	Mem Pipes Busy [%]	94.46
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

⚠ **L1TEX Global Load Access Pattern**

The memory access pattern for global loads in L1TEX might not be optimal. On average, this kernel accesses 4.0 bytes per thread per memory request; but the address pattern, possibly caused by the stride between threads, results in 9.5 sectors per request, or 9.5*32 = 302.4 bytes of cache data transfers per request. The optimal thread address pattern for 4.0 byte accesses would result in 4.0*32 = 128.0 bytes of cache data transfers per request, to maximize L1TEX cache performance. Check the Source Counters section for uncoalesced global loads.

⚠ **L1TEX Global Store Access Pattern**

The memory access pattern for global stores in L1TEX might not be optimal. On average, this kernel accesses 4.0 bytes per thread per memory request; but the address pattern, possibly caused by the stride between threads, results in 16.0 sectors per request, or 16.0*32 = 512.0 bytes of cache data transfers per request. The optimal thread address pattern for 4.0 byte accesses would result in 4.0*32 = 128.0 bytes of cache data transfers per request, to maximize L1TEX cache performance. Check the Source Counters section for uncoalesced global stores.

⚠ **L2 Store Access Pattern**

The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.2 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

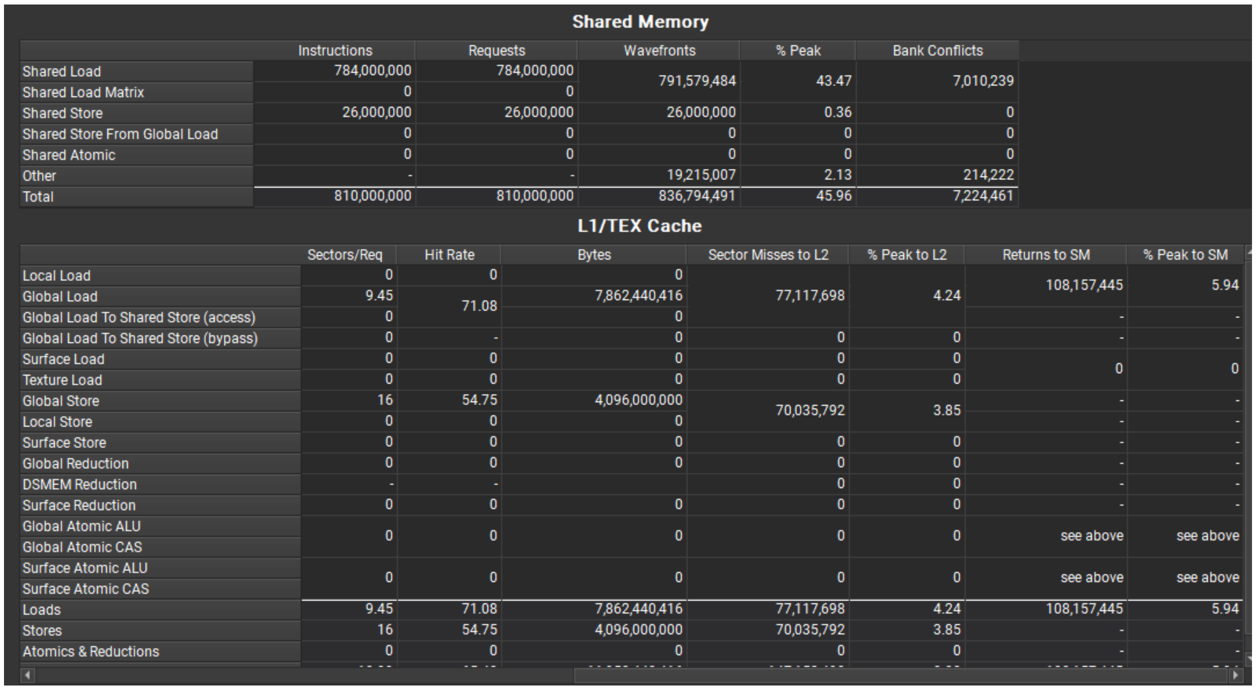
⚠ **L2 Load Access Pattern**

The memory access pattern for loads from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.1 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced loads and try to minimize how many cache lines need to be accessed per memory request.

Memory Chart

Show As: Transfer Size

Fig. – Nsight Compute Memory Chart for Tiled (Shared Memory) Convolution



Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	784,000,000	784,000,000	791,579,484	43.47	7,010,239
Shared Load Matrix	0	0			
Shared Store	26,000,000	26,000,000	26,000,000	0.36	0
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	19,215,007	2.13	214,222
Total	810,000,000	810,000,000	836,794,491	45.96	7,224,461

L1/TEX Cache

	Sectors/Req	Hit Rate	Bytes	Sector Misses to L2	% Peak to L2	Returns to SM	% Peak to SM
Local Load	0	0	0	77,117,698	4.24	108,157,445	5.94
Global Load	9.45	71.08	7,862,440,416				
Global Load To Shared Store (access)	0		0			-	-
Global Load To Shared Store (bypass)	0	-	0	0	0	-	-
Surface Load	0	0	0	0	0	0	0
Texture Load	0	0	0	0	0		
Global Store	16	54.75	4,096,000,000	70,035,792	3.85	-	-
Local Store	0	0	0			-	-
Surface Store	0	0	0	0	0	-	-
Global Reduction	0	0	0	0	0	-	-
DSMEM Reduction	-	-		0	0	-	-
Surface Reduction	0	0	0	0	0	-	-
Global Atomic ALU	0	0	0	0	0	see above	see above
Global Atomic CAS							
Surface Atomic ALU	0	0	0	0	0	see above	see above
Surface Atomic CAS							
Loads	9.45	71.08	7,862,440,416	77,117,698	4.24	108,157,445	5.94
Stores	16	54.75	4,096,000,000	70,035,792	3.85	-	-
Atomics & Reductions	0	0	0	0	0	-	-

Fig. – Nsight compute data chart for shared memory and L1/TEX cache

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

Yes, this optimization can synergize with constant memory usage by storing frequently accessed kernel data in constant memory, we can further reduce global memory accesses. To improve the global memory access pattern, focusing on memory coalescing can reduce the flaws of this optimization. Additionally, loop unrolling within the shared memory kernel can reduce the overhead of loop control and increase instruction-level parallelism.

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

i. Course Textbook p. 358

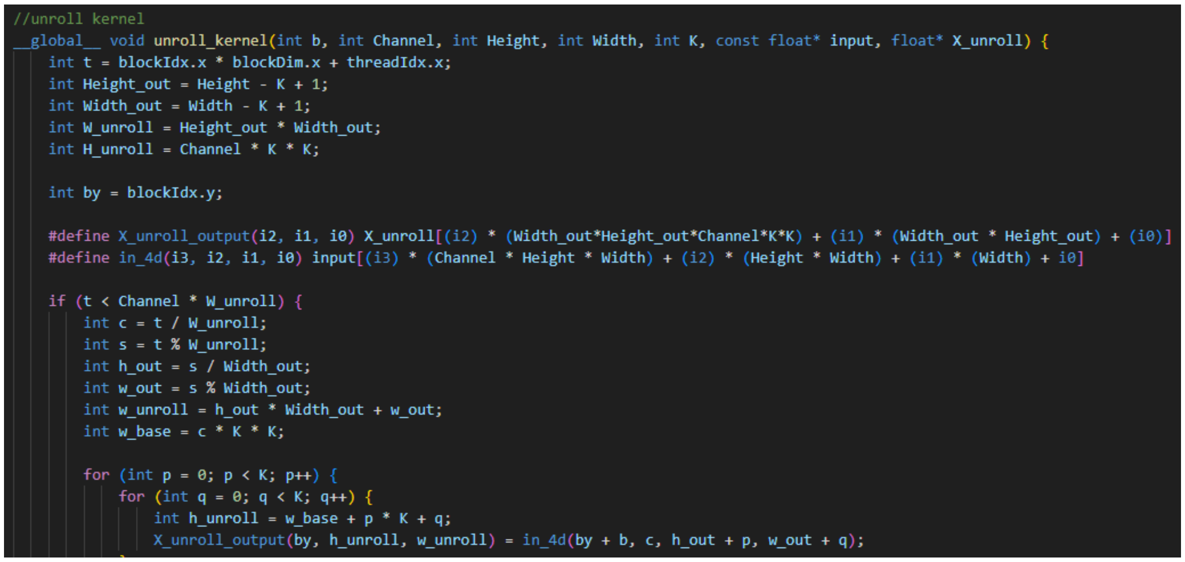
3. Optimization Number #2: Input Matrix Unrolling & Tiled Multiplication using Shared Memory

a. How does this optimization work in theory? Expected behavior?

The optimization combines two techniques, Input Matrix Unrolling and Tiled Multiplication using Shared Memory. Input matrix unrolling involves transforming the input matrix into a different representation to optimize memory access patterns. Tiled matrix multiplication is like the tiled convolution optimization, this approach loads tiles of the input matrix and kernel into shared memory to minimize global memory accesses and utilize the faster shared memory for computations. Since the matrix is unrolled explicitly (in a separate kernel), the run time is expected to slow down because the data is being replicated and transferred redundantly.

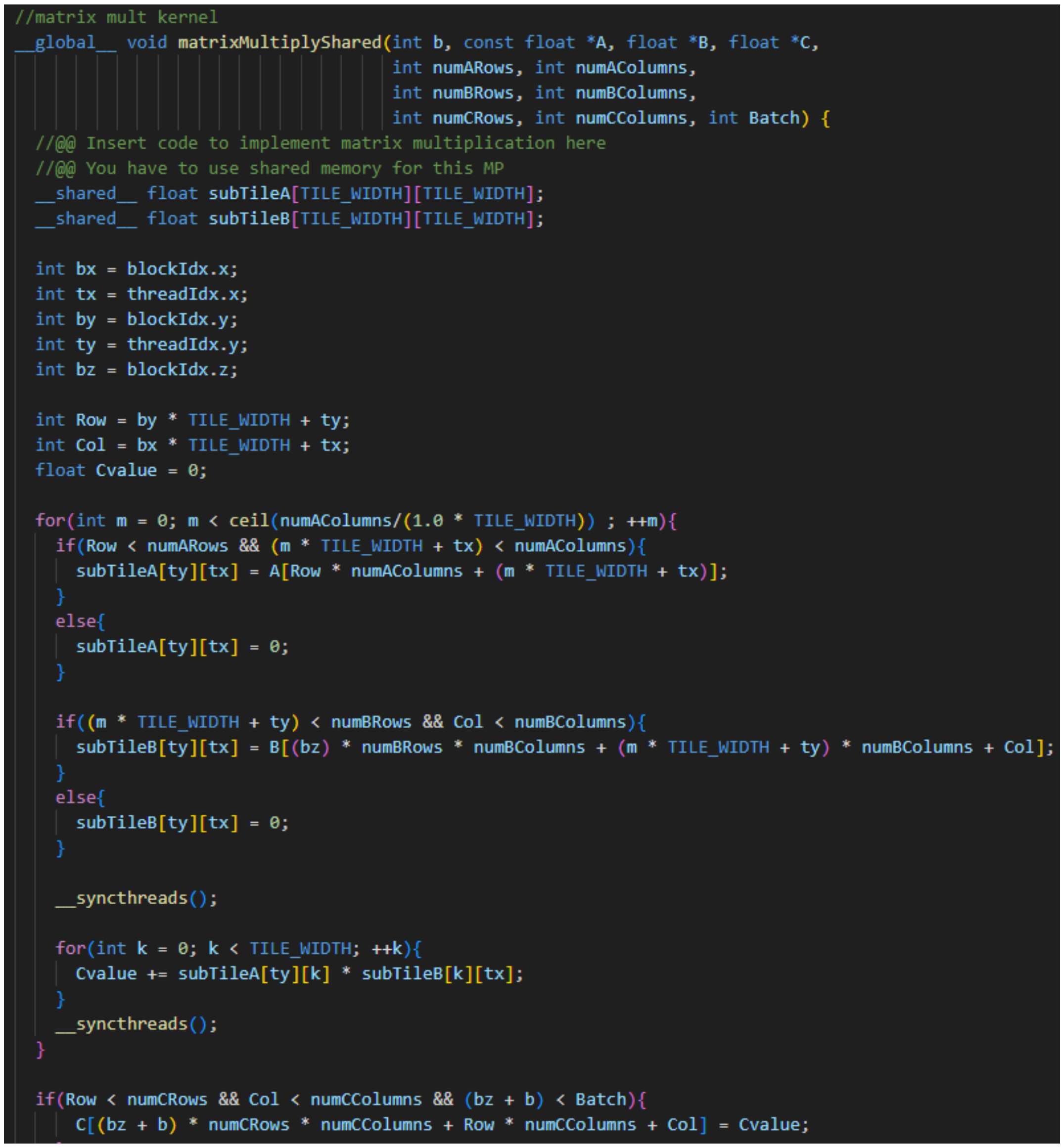
b. How did you implement your code? Explain thoroughly and show code snippets.

The implementation includes input matrix unrolling and tiled matrix multiplication using shared memory in two separate kernels. The unrolling kernel transforms the input matrix into a more optimized format for subsequent processing. This is done by rearranging the input data into a linear array that facilitates coalesced memory accesses and efficient computation in the next stage. The tiled matrix multiplication kernel performs the matrix multiplication using tiles of the input and kernel data loaded into shared memory. Lastly, the kernels are launched in "mini batches" using a loop, since running a large batch (above ~5000 images) leads to code failure which could be a result of global memory limitations. Snippets of the code for each kernel and kernel launches can be seen below.



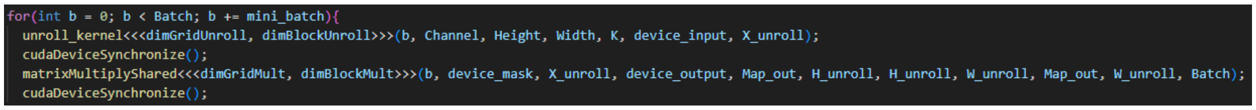
```
//unroll kernel
__global__ void unroll_kernel(int b, int Channel, int Height, int Width, int K, const float* input, float* X_unroll) {
    int t = blockIdx.x * blockDim.x + threadIdx.x;
    int Height_out = Height - K + 1;
    int Width_out = Width - K + 1;
    int W_unroll = Height_out * Width_out;
    int H_unroll = Channel * K * K;

    int by = blockIdx.y;

    #define X_unroll_output(i2, i1, i0) X_unroll[(i2) * (Width_out*Height_out*Channel*K*K) + (i1) * (Width_out * Height_out) + (i0)]
    #define in_4d(i3, i2, i1, i0) input[(i3) * (Channel * Height * Width) + (i2) * (Height * Width) + (i1) * (Width) + i0]

    if (t < Channel * W_unroll) {
        int c = t / W_unroll;
        int s = t % W_unroll;
        int h_out = s / Width_out;
        int w_out = s % Width_out;
        int w_unroll = h_out * Width_out + w_out;
        int w_base = c * K * K;

        for (int p = 0; p < K; p++) {
            for (int q = 0; q < K; q++) {
                int h_unroll = w_base + p * K + q;
                X_unroll_output(by, h_unroll, w_unroll) = in_4d(by + b, c, h_out + p, w_out + q);
```

Fig. – Matrix unrolling kernel



```
//matrix mult kernel
__global__ void matrixMultiplyShared(int b, const float *A, float *B, float *C,
                                     int numARows, int numAColumns,
                                     int numBRows, int numBColumns,
                                     int numCRows, int numCColumns, int Batch) {
  //@@ Insert code to implement matrix multiplication here
  //@@ You have to use shared memory for this MP
  __shared__ float subTileA[TILE_WIDTH][TILE_WIDTH];
  __shared__ float subTileB[TILE_WIDTH][TILE_WIDTH];

  int bx = blockIdx.x;
  int tx = threadIdx.x;
  int by = blockIdx.y;
  int ty = threadIdx.y;
  int bz = blockIdx.z;

  int Row = by * TILE_WIDTH + ty;
  int Col = bx * TILE_WIDTH + tx;
  float Cvalue = 0;

  for(int m = 0; m < ceil(numAColumns/(1.0 * TILE_WIDTH)) ; ++m){
    if(Row < numARows && (m * TILE_WIDTH + tx) < numAColumns){
      subTileA[ty][tx] = A[Row * numAColumns + (m * TILE_WIDTH + tx)];
    }
    else{
      subTileA[ty][tx] = 0;
    }

    if((m * TILE_WIDTH + ty) < numBRows && Col < numBColumns){
      subTileB[ty][tx] = B[(bz) * numBRows * numBColumns + (m * TILE_WIDTH + ty) * numBColumns + Col];
    }
    else{
      subTileB[ty][tx] = 0;
    }

    __syncthreads();

    for(int k = 0; k < TILE_WIDTH; ++k){
      Cvalue += subTileA[ty][k] * subTileB[k][tx];
    }
    __syncthreads();
  }

  if(Row < numCRows && Col < numCColumns && (bz + b) < Batch){
    C[(bz + b) * numCRows * numCColumns + Row * numCColumns + Col] = Cvalue;
```

Fig. – Code snippet of tiled matrix multiplication kernel



```
for(int b = 0; b < Batch; b += mini_batch){
  unroll_kernel<<<dimGridUnroll, dimBlockUnroll>>>(b, Channel, Height, Width, K, device_input, X_unroll);
  cudaDeviceSynchronize();
  matrixMultiplyShared<<<dimGridMult, dimBlockMult>>>(b, device_mask, X_unroll, device_output, Map_out, H_unroll, H_unroll, W_unroll, Map_out, W_unroll, Batch);
  cudaDeviceSynchronize();
```

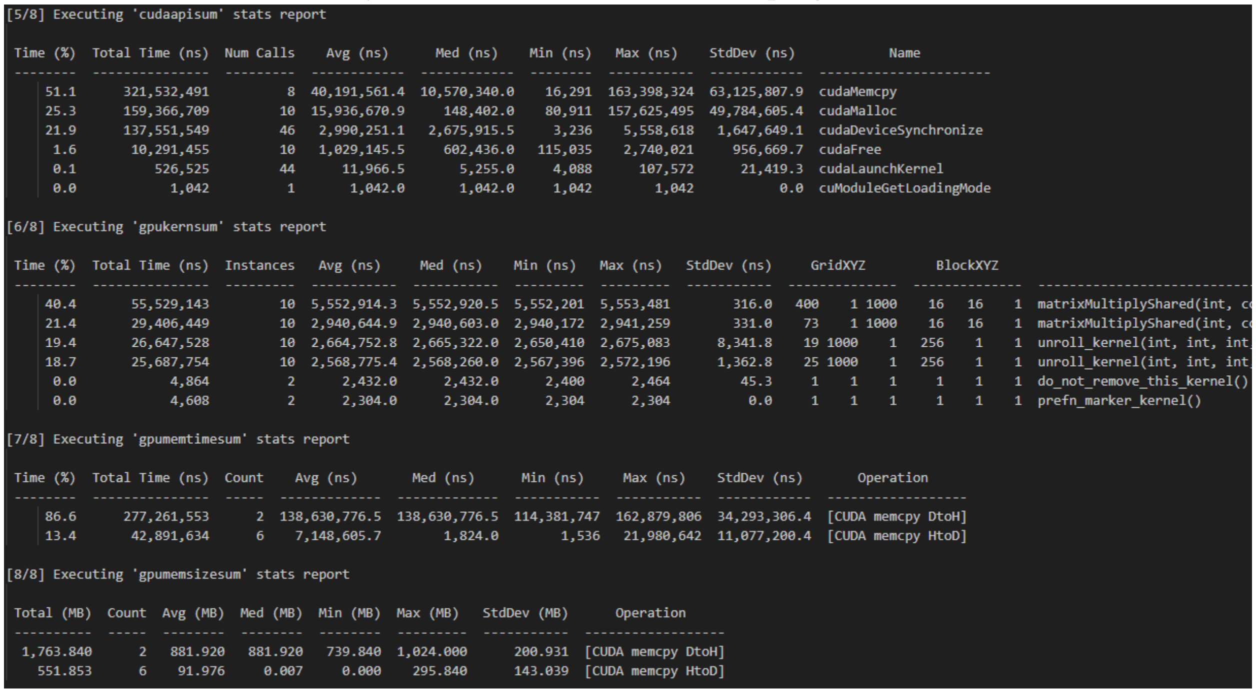
Fig. - Code snippet of "mini batch" launches

c. Did the performance match your expectation? Show your analysis results using profiling tools.

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

100	*1.17789 ms*	*0.961721 ms*	*0m4.086s*	*0.86*
1000	*8.52241 ms*	*5.80802 ms*	*0m37.380s*	*0.886*
10000	*82.477 ms*	*56.9284 ms*	*6m4.185s*	*0.8714*

Compared to the baseline, this optimization did not improve op times. This is due to the replication of the data. Having separate kernels requires the unrolled matrix to be computed separately and then transferred to the matrix multiplication. Such redundancies are nonoptimal for speed. Based on the nsys results below, there are 10 instances of each kernel launch, and each isn't very fast. In cudaapisum, it is also visible that there are 46 instances of cudaDeviceSynchronize since it needs to be executed after every kernel. Such factors slow the program down.

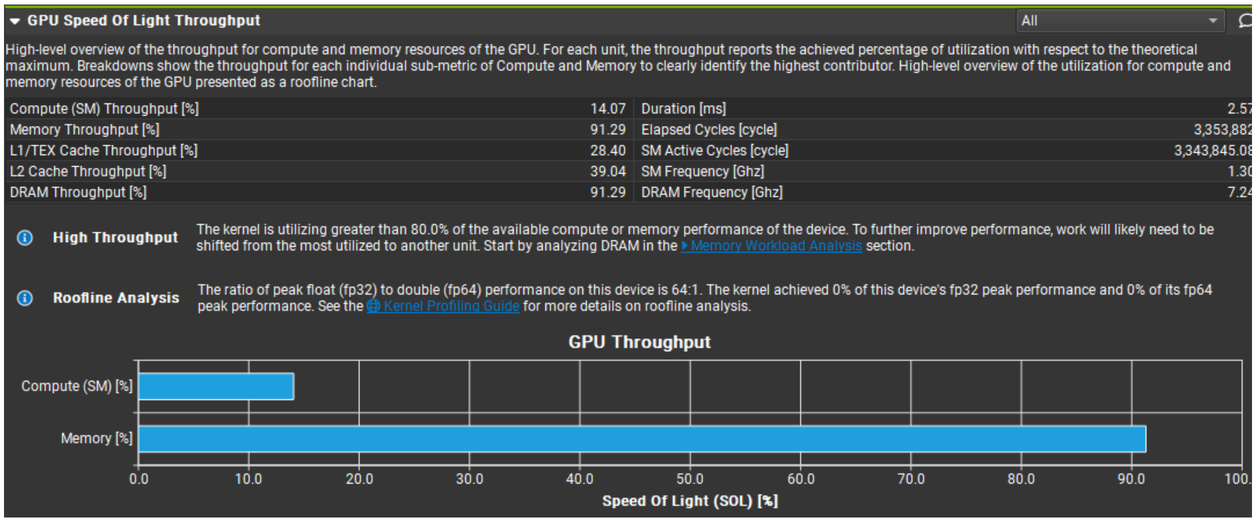


```
[5/8] Executing 'cudaapisum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)     Min (ns)    Max (ns)     StdDev (ns)           Name
 --------  ---------------  ---------  ------------  ------------  --------  -----------  ------------  ---------------------
     51.1      321,532,491          8  40,191,561.4  10,570,340.0    16,291  163,398,324  63,125,807.9  cudaMemcpy
     25.3      159,366,709         10  15,936,670.9     148,402.0    80,911  157,625,495  49,784,605.4  cudaMalloc
     21.9      137,551,549         46   2,990,251.1   2,675,915.5     3,236    5,558,618   1,647,649.1  cudaDeviceSynchronize
      1.6       10,291,455         10   1,029,145.5     602,436.0   115,035    2,740,021     956,669.7  cudaFree
      0.1          526,525         44      11,966.5       5,255.0     4,088      107,572      21,419.3  cudaLaunchKernel
      0.0            1,042          1       1,042.0       1,042.0     1,042        1,042           0.0  cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)     Med (ns)     Min (ns)    Max (ns)    StdDev (ns)     GridXYZ         BlockXYZ
 --------  ---------------  ---------  -----------  -----------  ---------  ---------  -----------  --------------  --------------  ----------------------------
     40.4       55,529,143         10  5,552,914.3  5,552,920.5  5,552,201  5,553,481        316.0   400     1 1000    16    16     1  matrixMultiplyShared(int, c
     21.4       29,406,449         10  2,940,644.9  2,940,603.0  2,940,172  2,941,259        331.0    73     1 1000    16    16     1  matrixMultiplyShared(int, c
     19.4       26,647,528         10  2,664,752.8  2,665,322.0  2,650,410  2,675,083      8,341.8    19 1000     1   256     1     1  unroll_kernel(int, int, int
     18.7       25,687,754         10  2,568,775.4  2,568,260.0  2,567,396  2,572,196      1,362.8    25 1000     1   256     1     1  unroll_kernel(int, int, int
      0.0            4,864          2      2,432.0      2,432.0      2,400      2,464         45.3     1     1     1     1     1     1  do_not_remove_this_kernel()
      0.0            4,608          2      2,304.0      2,304.0      2,304      2,304          0.0     1     1     1     1     1     1  prefn_marker_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)       Med (ns)      Min (ns)     Max (ns)     StdDev (ns)       Operation
 --------  ---------------  -----  -------------  -------------  -----------  -----------  ------------  ------------------
     86.6      277,261,553      2  138,630,776.5  138,630,776.5  114,381,747  162,879,806  34,293,306.4  [CUDA memcpy DtoH]
     13.4       42,891,634      6    7,148,605.7        1,824.0        1,536   21,980,642  11,077,200.4  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)   StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  ---------  -----------  ------------------
  1,763.840      2   881.920   881.920   739.840  1,024.000      200.931  [CUDA memcpy DtoH]
    551.853      6    91.976     0.007     0.000    295.840      143.039  [CUDA memcpy HtoD]
```

Fig. – nsys results 5-8 for Input Matrix Unrolling & Tiled Multiplication using Shared Memory

Looking at the Nsight Compute output below, it is realized that the GPU throughput for the matrix unrolling is nonoptimal. The compute percentage is very low while the memory usage is very high, underutilizing compute resources. In the figure containing the warp statistics for the matrix unroll kernel, the errors can be seen next to the yellow icon. In the unroll kernel, time is being wasted waiting to receive data resulting in stalls and under utilization of the possible warp output shown in the warp statistics.

On the other hand, GPU throughput for the matrix multiplication kernel is much better and is balanced. The matrix multiplication kernel can improve in its L2 hit rate as it is at 7.76%. Loads and store from the L1/TEX to the L2 are not optimal as seen in the figure containing the memory workload analysis. The matrix multiplication could benefit from better memory access patterns.



GPU Speed Of Light Throughput

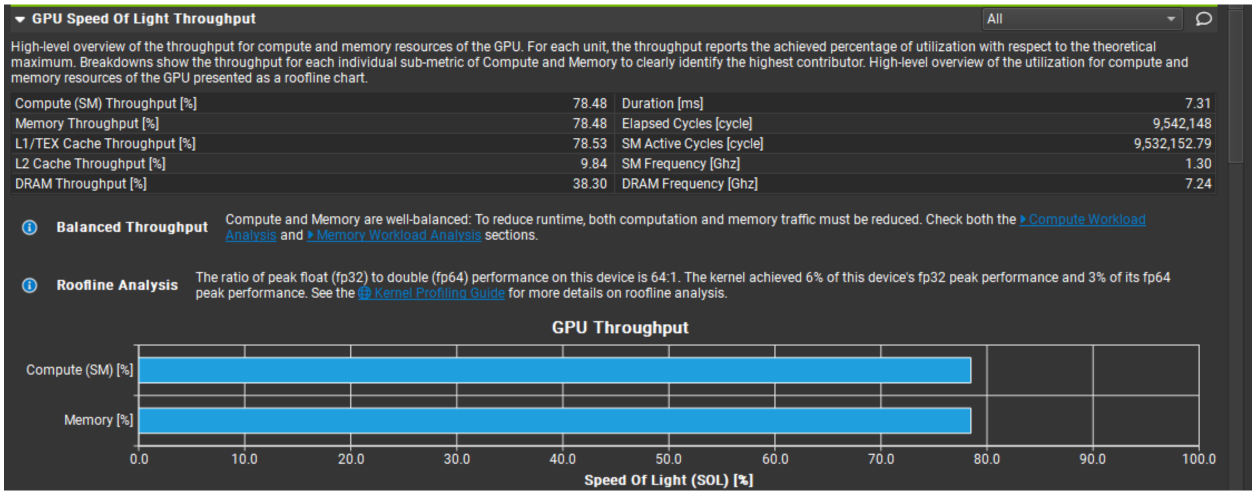
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	14.07	Duration [ms]	2.57
Memory Throughput [%]	91.29	Elapsed Cycles [cycle]	3,353,882
L1/TEX Cache Throughput [%]	28.40	SM Active Cycles [cycle]	3,343,845.08
L2 Cache Throughput [%]	39.04	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	91.29	DRAM Frequency [Ghz]	7.24

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing DRAM in the Memory Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 0% of this device's fp32 peak performance and 0% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. – Matrix Unroll Kernel GPU SOL Throughput



GPU Speed Of Light Throughput

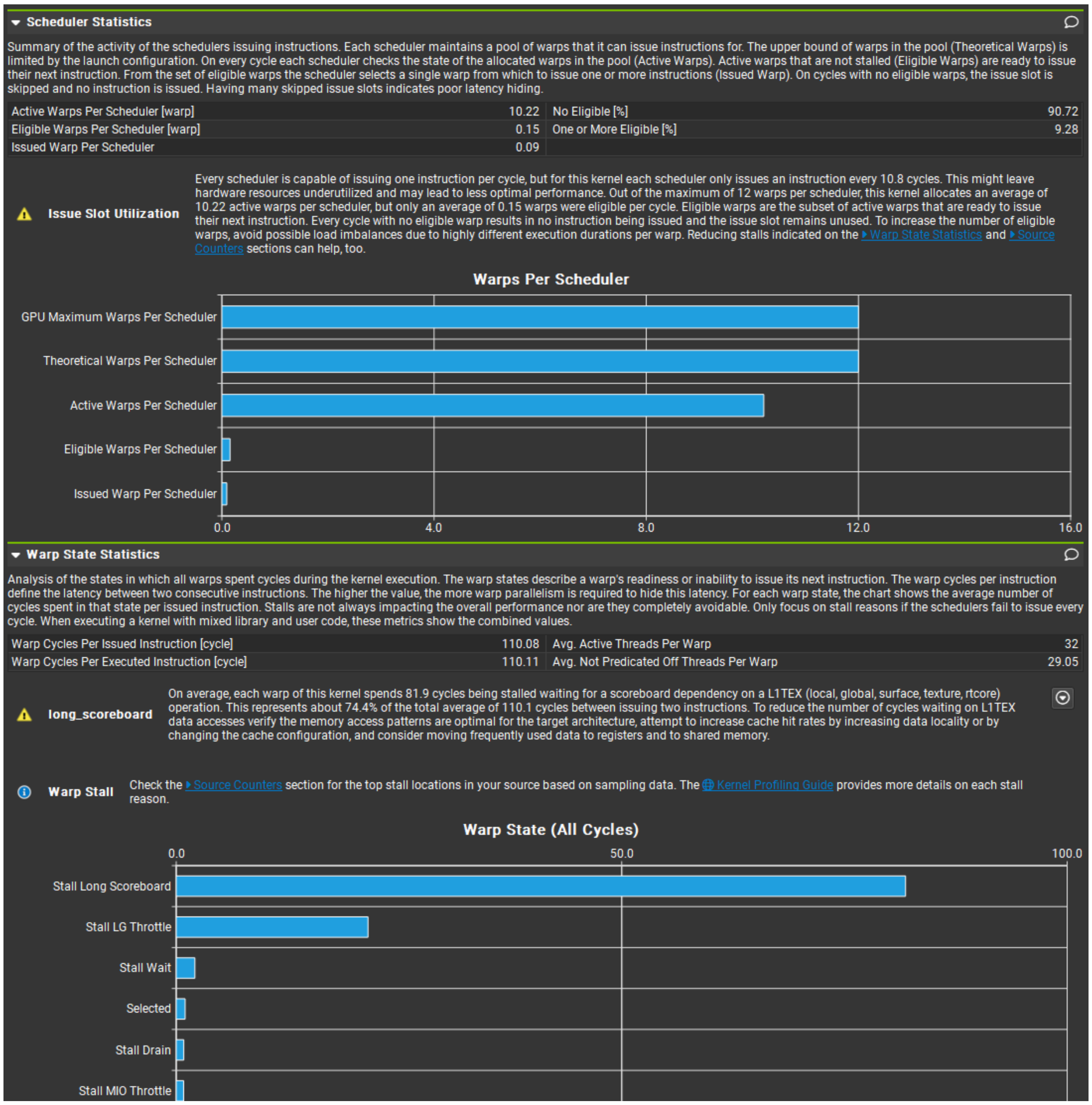
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	78.48	Duration [ms]	7.31
Memory Throughput [%]	78.48	Elapsed Cycles [cycle]	9,542,148
L1/TEX Cache Throughput [%]	78.53	SM Active Cycles [cycle]	9,532,152.79
L2 Cache Throughput [%]	9.84	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	38.30	DRAM Frequency [Ghz]	7.24

Balanced Throughput Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis sections.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 6% of this device's fp32 peak performance and 3% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. –Tiled Matrix Multiply GPU SOL Throughput



Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	10.22	No Eligible [%]	90.72
Eligible Warps Per Scheduler [warp]	0.15	One or More Eligible [%]	9.28
Issued Warp Per Scheduler	0.09		

Issue Slot Utilization Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 10.8 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 12 warps per scheduler, this kernel allocates an average of 10.22 active warps per scheduler, but only an average of 0.15 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the Warp State Statistics and Source Counters sections can help, too.

Warps Per Scheduler
GPU Maximum Warps Per Scheduler
Theoretical Warps Per Scheduler
Active Warps Per Scheduler
Eligible Warps Per Scheduler
Issued Warp Per Scheduler
0.0
4.0
8.0
12.0
16.0

Warp State Statistics

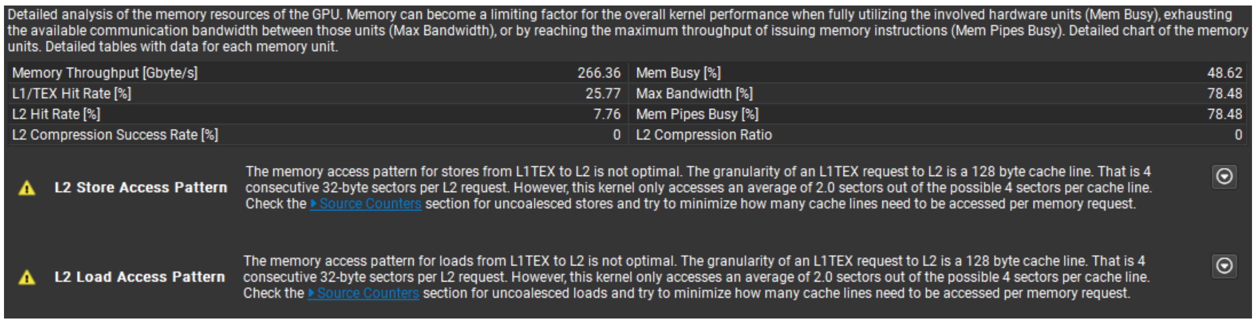
Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	110.08	Avg. Active Threads Per Warp	32
Warp Cycles Per Executed Instruction [cycle]	110.11	Avg. Not Predicated Off Threads Per Warp	29.05

long_scoreboard On average, each warp of this kernel spends 81.9 cycles being stalled waiting for a scoreboard dependency on a L1TEX (local, global, surface, texture, rtcore) operation. This represents about 74.4% of the total average of 110.1 cycles between issuing two instructions. To reduce the number of cycles waiting on L1TEX data accesses verify the memory access patterns are optimal for the target architecture, attempt to increase cache hit rates by increasing data locality or by changing the cache configuration, and consider moving frequently used data to registers and to shared memory.

Warp Stall Check the Source Counters section for the top stall locations in your source based on sampling data. The Kernel Profiling Guide provides more details on each stall reason.

Fig. – Warp Statistics for Matrix Unroll Kernel



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]	266.36	Mem Busy [%]	48.62
L1/TEX Hit Rate [%]	25.77	Max Bandwidth [%]	78.48
L2 Hit Rate [%]	7.76	Mem Pipes Busy [%]	78.48
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L2 Store Access Pattern The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 2.0 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

L2 Load Access Pattern The memory access pattern for loads from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 2.0 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced loads and try to minimize how many cache lines need to be accessed per memory request.

Fig. – Memory Workload Analysis for Tiled Matrix Multiplication

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

Constant memory can be used to store immutable data such as kernel weights that do not change during execution and are shared across threads. When combined with unrolling and tiling, using constant memory reduces memory latency and increases cache efficiency. Furthermore, the mini batch computations can be distributed across different streams to use memory efficiently. Loop unrolling could be implemented to exploit instruction level parallelism in the loops of the unrolling and matrix multiplication. Lastly, this optimization can benefit from kernel fusion which is discussed in the next section.

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

   i. Course Textbook p. 359-366
   ii. Slide Deck 18: Computation in Deep Neural Networks

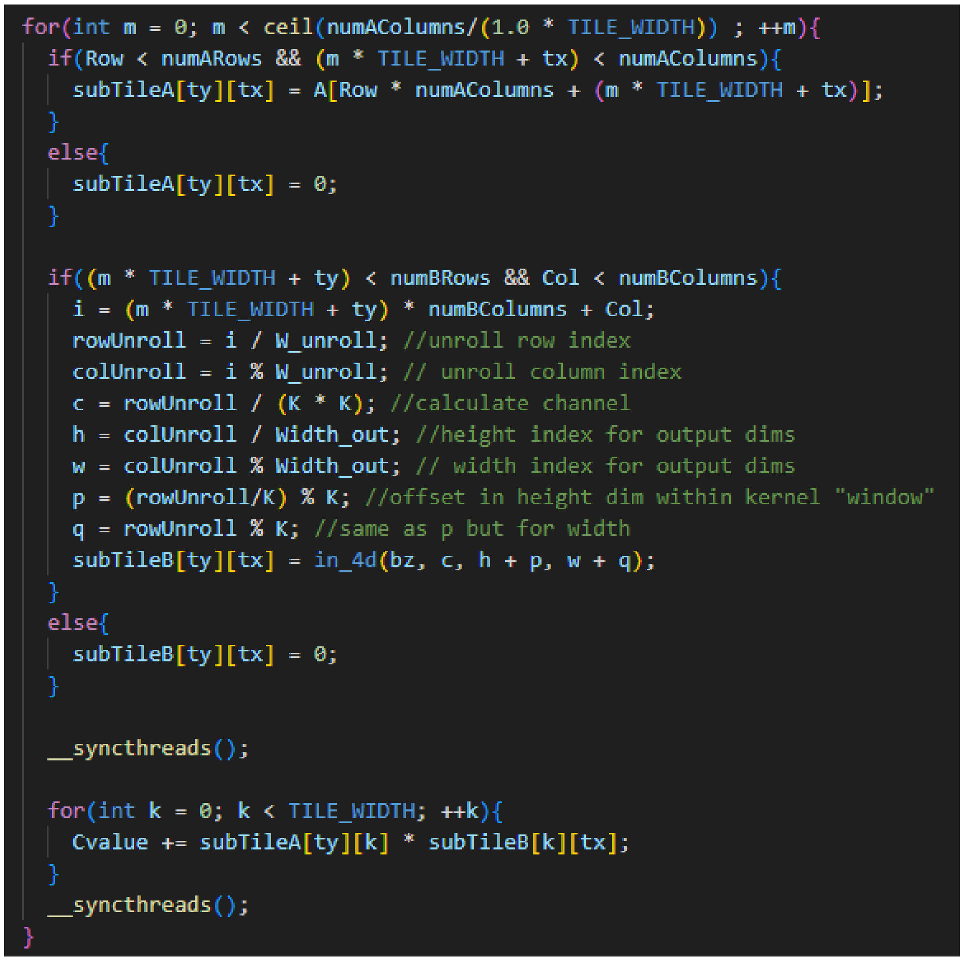
4. Optimization Number #3: Kernel Fusion for Unrolling and Matrix-Multiplication

a. How does this optimization work in theory? Expected behavior?

Kernel fusion combines multiple computational steps into a single kernel. This approach reduces the overhead of launching multiple kernels and minimizes data transfers between the device and memory. By fusing the unrolling and matrix multiplication steps from optimization #2, the data is easier to access. Less time is wasted replicating, which significantly speeds up the process by reducing global memory access and improving locality.

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

The fused kernel below performs both unrolling and matrix multiplication using shared memory. This fusion of operations is done within a single kernel launch to ensure data remains on-chip, enhancing computational speed and efficiency. The code snippet below demonstrates how the matrix is unrolled and loaded into shared memory. The comments in the snippet explicitly state how the unrolling occurs. Furthermore, the kernel is configured to handle dimensions appropriately for the input and output sizes, ensuring that the tiling covers the entire operation. 3D grid dimensions are used to accommodate batch processing and tiling within the matrix multiplication.



```
for(int m = 0; m < ceil(numAColumns/(1.0 * TILE_WIDTH)) ; ++m){
  if(Row < numARows && (m * TILE_WIDTH + tx) < numAColumns){
    subTileA[ty][tx] = A[Row * numAColumns + (m * TILE_WIDTH + tx)];
  }
  else{
    subTileA[ty][tx] = 0;
  }

  if((m * TILE_WIDTH + ty) < numBRows && Col < numBColumns){
    i = (m * TILE_WIDTH + ty) * numBColumns + Col;
    rowUnroll = i / W_unroll; //unroll row index
    colUnroll = i % W_unroll; // unroll column index
    c = rowUnroll / (K * K); //calculate channel
    h = colUnroll / Width_out; //height index for output dims
    w = colUnroll % Width_out; // width index for output dims
    p = (rowUnroll/K) % K; //offset in height dim within kernel "window"
    q = rowUnroll % K; //same as p but for width
    subTileB[ty][tx] = in_4d(bz, c, h + p, w + q);
  }
  else{
    subTileB[ty][tx] = 0;
  }

  __syncthreads();

  for(int k = 0; k < TILE_WIDTH; ++k){
    Cvalue += subTileA[ty][k] * subTileB[k][tx];
  }
  __syncthreads();
}
```

Fig. – Code snippet of fused kernel for unrolling and matrix multiplication

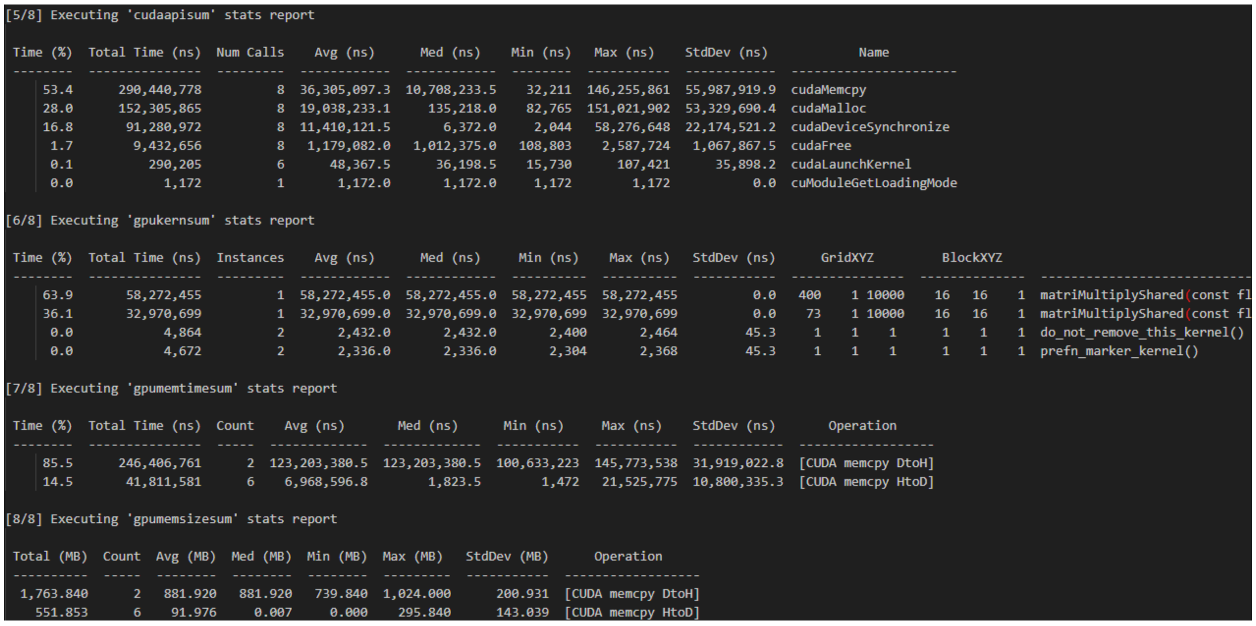
c. Did the performance match your expectation? Show your analysis results using profiling tools.

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

100	*0.637177 ms*	*0.356009 ms*	*0m4.044s*	*0.86*
1000	*5.81798 ms*	*3.32418 ms*	*0m36.556s*	*0.886*
10000	*57.4412 ms*	*33.0359 ms*	*6m2.886s*	*0.8714*

Compared to optimization #2, this optimization is much faster as it demonstrates the benefits of fusion, but the op times are still slower compared to the baseline. The overall execution time is slightly improved, but this could just vary with each execution of the program. The slow down in op time could be due to how the memory is being accessed and loaded into shared memory causing op times to slow down.

The nsys profiling shows how many fewer kernel launches and cudaDeviceSnychronizes compared to two separate kernels in the previous optimization. With fusion, only one kernel needs to be launched per layer in the convolution. It can also be noticed that fewer time is spent in gpumemtimesum compared to other optimizations.

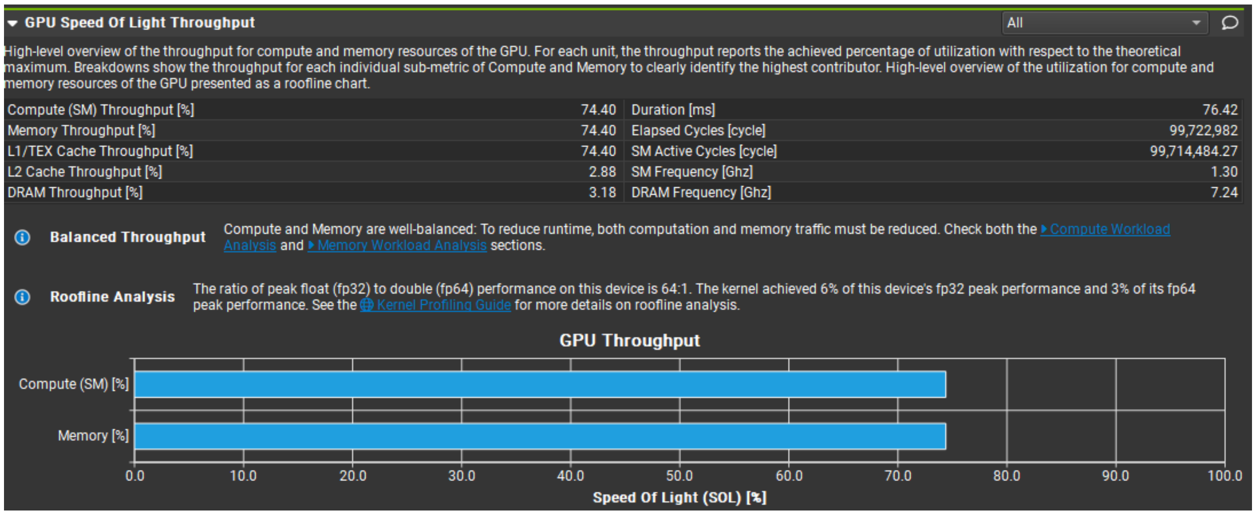


```
[5/8] Executing 'cudaapisum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)    Min (ns)   Max (ns)     StdDev (ns)           Name
 --------  ---------------  ---------  ------------  ------------  --------  -----------  ------------  ---------------------
     53.4      290,440,778          8  36,305,097.3  10,708,233.5    32,211  146,255,861  55,987,919.9  cudaMemcpy
     28.0      152,305,865          8  19,038,233.1     135,218.0    82,765  151,021,902  53,329,690.4  cudaMalloc
     16.8       91,280,972          8  11,410,121.5       6,372.0     2,044   58,276,648  22,174,521.2  cudaDeviceSynchronize
      1.7        9,432,656          8   1,179,082.0   1,012,375.0   108,803    2,587,724   1,067,867.5  cudaFree
      0.1          290,205          6      48,367.5      36,198.5    15,730      107,421      35,898.2  cudaLaunchKernel
      0.0            1,172          1       1,172.0       1,172.0     1,172        1,172           0.0  cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)    StdDev (ns)      GridXYZ         BlockXYZ
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  --------------  --------------  ----------------------------
     63.9       58,272,455          1  58,272,455.0  58,272,455.0  58,272,455  58,272,455          0.0   400    1 10000    16   16    1  matriMultiplyShared(const fl
     36.1       32,970,699          1  32,970,699.0  32,970,699.0  32,970,699  32,970,699          0.0    73    1 10000    16   16    1  matriMultiplyShared(const fl
      0.0            4,864          2       2,432.0       2,432.0       2,400       2,464         45.3     1    1    1     1    1    1  do_not_remove_this_kernel()
      0.0            4,672          2       2,336.0       2,336.0       2,304       2,368         45.3     1    1    1     1    1    1  prefn_marker_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)       Med (ns)       Min (ns)     Max (ns)     StdDev (ns)       Operation
 --------  ---------------  -----  -------------  -------------  -----------  -----------  ------------  ------------------
     85.5      246,406,761      2  123,203,380.5  123,203,380.5  100,633,223  145,773,538  31,919,022.8  [CUDA memcpy DtoH]
     14.5       41,811,581      6    6,968,596.8        1,823.5        1,472   21,525,775  10,800,335.3  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)   StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  ---------  -----------  ------------------
  1,763.840      2   881.920   881.920   739.840  1,024.000      200.931  [CUDA memcpy DtoH]
    551.853      6    91.976     0.007     0.000    295.840      143.039  [CUDA memcpy HtoD]
```

Fig. – nsys results 5-8 for Kernel Fusion for Unrolling and Matrix-Multiplication

In Nsight Compute, the GPU throughput is well balanced and to reduce runtime, both computation and memory traffic must be reduced as said in the figure below. This throughput is a significant improvement to the separated optimizations. The compute workload analysis demonstrates that FP64 is a heavily utilized pipeline at

73.4%. The LSU pipeline is used heavily to handle low latency instructions. In addition, the L1/TEX and L2 cache are well utilized as they both have a high hit rate, but the L2 store and load access pattern could improve as seen in the memory workload analysis figure. Lastly, the shared memory is utilized shown by the shared memory data, but there still exist 2,896,371 bank conflicts which is slowing down the program. These bank conflicts may be the reason why the op times are still slower


GPU Speed Of Light Throughput

All

High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

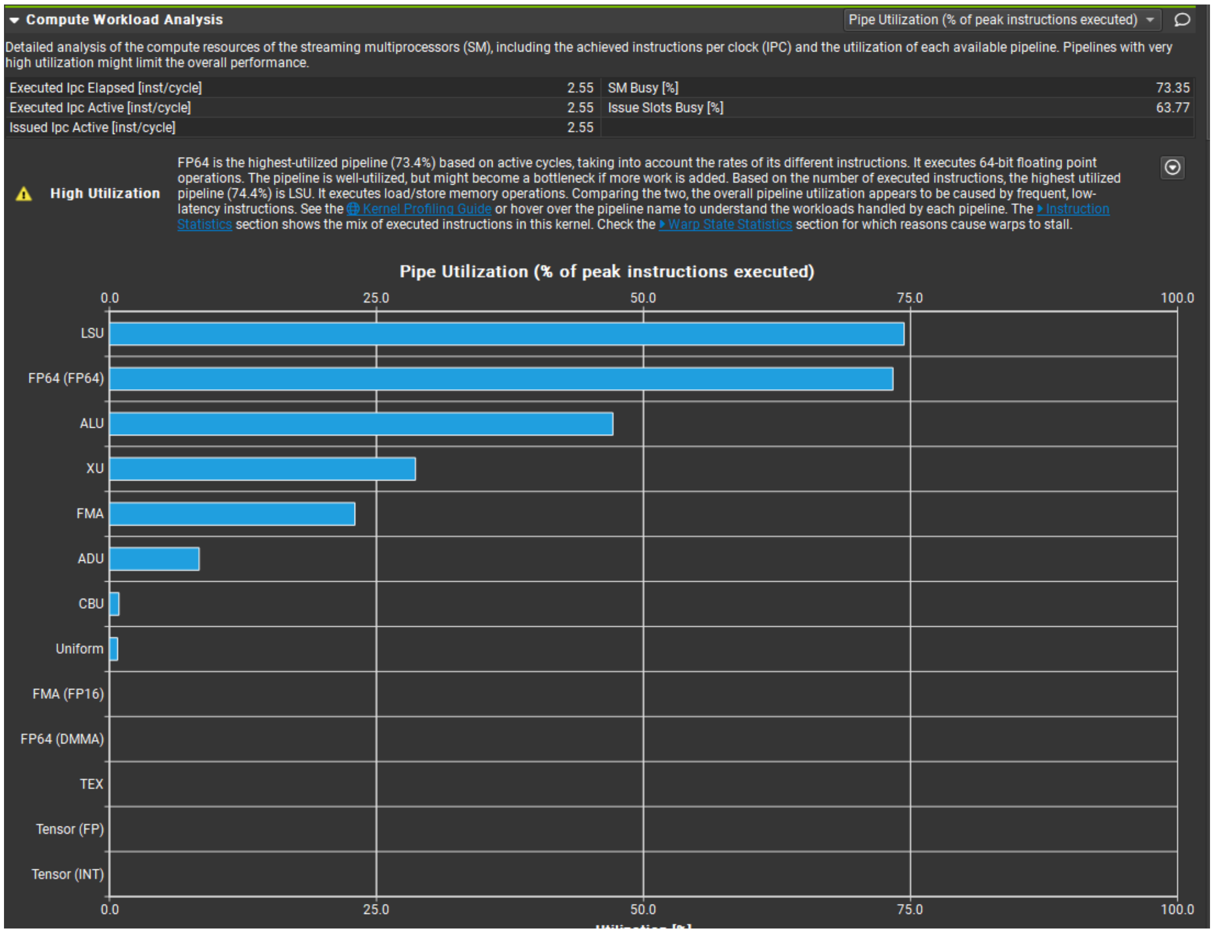
Compute (SM) Throughput [%]	74.40	Duration [ms]	76.42
Memory Throughput [%]	74.40	Elapsed Cycles [cycle]	99,722,982
L1/TEX Cache Throughput [%]	74.40	SM Active Cycles [cycle]	99,714,484.27
L2 Cache Throughput [%]	2.88	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	3.18	DRAM Frequency [Ghz]	7.24

Balanced Throughput Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis sections.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 6% of this device's fp32 peak performance and 3% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

than the baseline.

Fig. – Kernel Fusion for Unrolling and Matrix-Multiplication GPU SOL Throughput



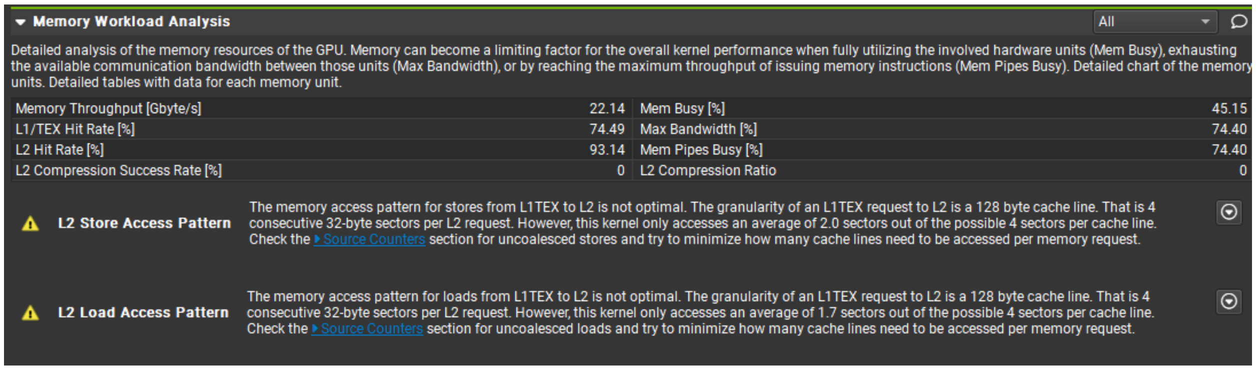
Compute Workload Analysis — Pipe Utilization (% of peak instructions executed)

Detailed analysis of the compute resources of the streaming multiprocessors (SM), including the achieved instructions per clock (IPC) and the utilization of each available pipeline. Pipelines with very high utilization might limit the overall performance.

Executed Ipc Elapsed [inst/cycle]	2.55	SM Busy [%]	73.35
Executed Ipc Active [inst/cycle]	2.55	Issue Slots Busy [%]	63.77
Issued Ipc Active [inst/cycle]	2.55		

High Utilization FP64 is the highest-utilized pipeline (73.4%) based on active cycles, taking into account the rates of its different instructions. It executes 64-bit floating point operations. The pipeline is well-utilized, but might become a bottleneck if more work is added. Based on the number of executed instructions, the highest utilized pipeline (74.4%) is LSU. It executes load/store memory operations. Comparing the two, the overall pipeline utilization appears to be caused by frequent, low-latency instructions. See the Kernel Profiling Guide or hover over the pipeline name to understand the workloads handled by each pipeline. The Instruction Statistics section shows the mix of executed instructions in this kernel. Check the Warp State Statistics section for which reasons cause warps to stall.

Fig. - Compute Workload Analysis for Kernel Fusion for Unrolling and Matrix-Multiplication



Memory Workload Analysis — All

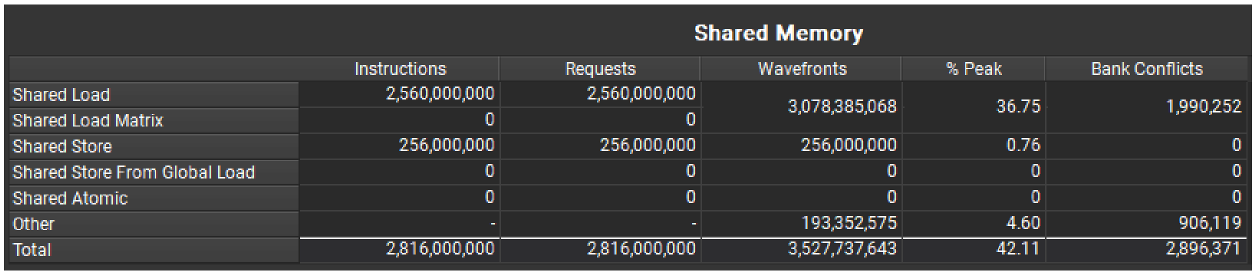
Detailed analysis of the memory resources of the GPU. Memory can become a limiting factor for the overall kernel performance when fully utilizing the involved hardware units (Mem Busy), exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput 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]	22.14	Mem Busy [%]	45.15
L1/TEX Hit Rate [%]	74.49	Max Bandwidth [%]	74.40
L2 Hit Rate [%]	93.14	Mem Pipes Busy [%]	74.40
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L2 Store Access Pattern The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 2.0 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

L2 Load Access Pattern The memory access pattern for loads from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.7 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced loads and try to minimize how many cache lines need to be accessed per memory request.

Fig. - Memory Workload Analysis for Kernel Fusion for Unrolling and Matrix-Multiplication



Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	2,560,000,000	2,560,000,000	3,078,385,068	36.75	1,990,252
Shared Load Matrix	0	0			
Shared Store	256,000,000	256,000,000	256,000,000	0.76	0
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	193,352,575	4.60	906,119
Total	2,816,000,000	2,816,000,000	3,527,737,643	42.11	2,896,371

Fig. – Shared Memory Data for Kernel Fusion for Unrolling and Matrix-Multiplication

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

Kernel fusion, when combined with loop unrolling and constant memory usage, could lead to performance improvement. Loop unrolling within the fused kernel reduces the overhead of loop control and increases instruction throughput by executing multiple iterations in a single loop cycle. This enhances the efficiency of the computation within the kernel. Additionally, using constant memory in fused kernels provides fast access to frequently used data (the mask), minimizing memory latency. This synergy improves data throughput and computational speed, as constant data is accessed faster, and loops are executed more efficiently.

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

   i. Course Textbook p. 359-366
   ii. Slide Deck 18: Computation in Deep Neural Networks

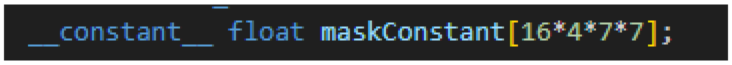
5. **Optimization Number #4: Weight Matrix (Kernel) in constant memory**

a. How does this optimization work in theory? Expected behavior?

This optimization places the weight matrix in constant memory, which has significantly faster access times than global memory. The weight matrix is placed in constant memory since its values do not change and it is frequently reused. This reduces memory latency, leading to faster convolution operations. By significantly eliminating global memory accesses for the mask, op times can be expected to improve.

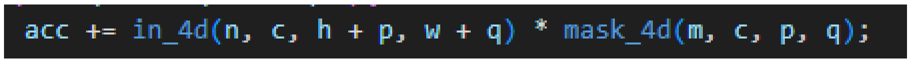
b. How did you implement your code? Explain thoroughly and show code snippets.

The kernel weights are placed in constant memory using __constant__ and accessed via the mask_4d macro. The constant memory size calculation comes from the number of inputs and channels there, this can be identified through the output feature map sizes in the kernel launches. The cudaMemcpyToSymbol function is used to copy the weights from host memory to constant memory.



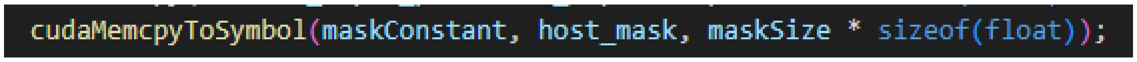
```
__constant__ float maskConstant[16*4*7*7];
```

Fig. - Code snippet of constant memory allocation



```
acc += in_4d(n, c, h + p, w + q) * mask_4d(m, c, p, q);
```

Fig. – Code snippet of mask_4d in convolution computation



```
cudaMemcpyToSymbol(maskConstant, host_mask, maskSize * sizeof(float));
```

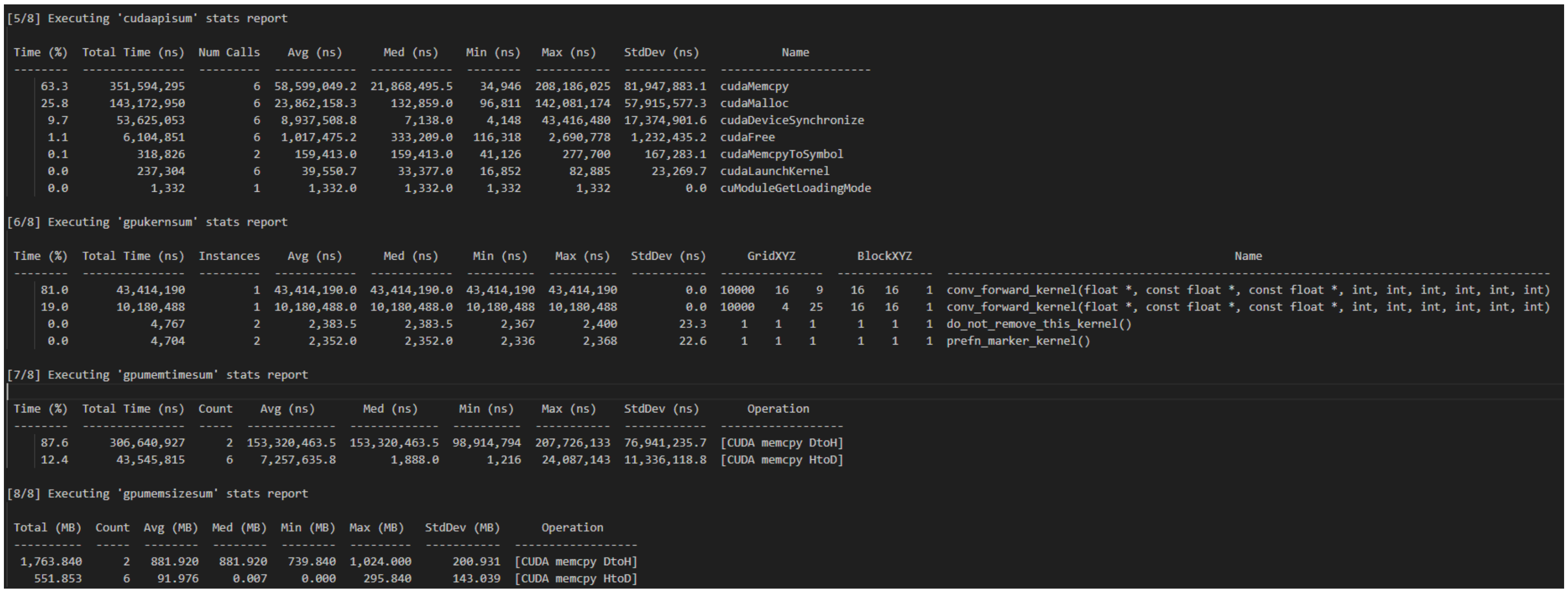
Fig. – Code snippet of copying mask form host to constant memory

c. Did the performance match your expectation? Show your analysis results using profiling tools.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.137679 ms*	*0.386495 ms*	*0m4.087s*	*0.86*
1000	*1.00568 ms*	*4.0301 ms*	*0m36.836s*	*0.886*
10000	*10.0485 ms*	*43.2677 ms*	*6m3.901s*	*0.8714*

The op times demonstrate that constant memory optimization significantly improves performance. The program's run time improves as there are significantly fewer cycles when accessing constant memory versus global, and with repetitive use of the mask, this is extremely beneficial.

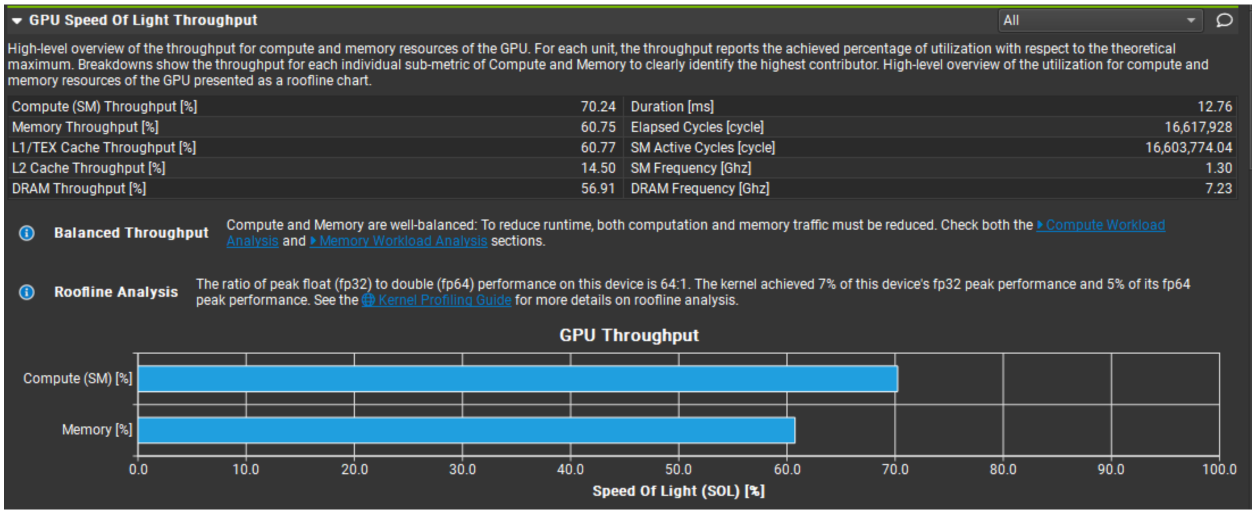
Firstly, in the nsys results below, kernel time has reduced compared to other optimizations and the baseline. Second, there are fewer cudaMemcpys, fewer cudaMallocs, and fewer cudaFrees due to the single cudaMemcpyToSymbol. These reductions have led to an increase in performance.



```
[5/8] Executing 'cudaapisum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)    Min (ns)   Max (ns)    StdDev (ns)            Name
 --------  ---------------  ---------  ------------  ------------  --------  -----------  ------------  ----------------------
     63.3      351,594,295          6  58,599,049.2  21,868,495.5    34,946  208,186,025  81,947,883.1  cudaMemcpy
     25.8      143,172,950          6  23,862,158.3     132,859.0    96,811  142,081,174  57,915,577.3  cudaMalloc
      9.7       53,625,053          6   8,937,508.8       7,138.0     4,148   43,416,480  17,374,901.6  cudaDeviceSynchronize
      1.1        6,104,851          6   1,017,475.2     333,209.0   116,318    2,690,778   1,232,435.2  cudaFree
      0.1          318,826          2     159,413.0     159,413.0    41,126      277,700     167,283.1  cudaMemcpyToSymbol
      0.0          237,304          6      39,550.7      33,377.0    16,852       82,885      23,269.7  cudaLaunchKernel
      0.0            1,332          1       1,332.0       1,332.0     1,332        1,332           0.0  cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)      GridXYZ         BlockXYZ                                          Name
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ---------------  --------------  ------------------------------------------------------------------------------------------
     81.0       43,414,190          1  43,414,190.0  43,414,190.0  43,414,190  43,414,190          0.0  10000   16    9    16   16    1  conv_forward_kernel(float *, const float *, const float *, int, int, int, int, int, int)
     19.0       10,180,488          1  10,180,488.0  10,180,488.0  10,180,488  10,180,488          0.0  10000    4   25    16   16    1  conv_forward_kernel(float *, const float *, const float *, int, int, int, int, int, int)
      0.0            4,767          2       2,383.5       2,383.5       2,367       2,400         23.3     1    1    1     1    1    1  do_not_remove_this_kernel()
      0.0            4,704          2       2,352.0       2,352.0       2,336       2,368         22.6     1    1    1     1    1    1  prefn_marker_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)       Med (ns)      Min (ns)    Max (ns)    StdDev (ns)       Operation
 --------  ---------------  -----  -------------  -------------  ----------  -----------  ------------  ------------------
     87.6      306,640,927      2  153,320,463.5  153,320,463.5  98,914,794  207,726,133  76,941,235.7  [CUDA memcpy DtoH]
     12.4       43,545,815      6    7,257,635.8        1,888.0       1,216   24,087,143  11,336,118.8  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)   StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  ---------  -----------  ------------------
  1,763.840      2   881.920   881.920   739.840  1,024.000      200.931  [CUDA memcpy DtoH]
    551.853      6    91.976     0.007     0.000    295.840      143.039  [CUDA memcpy HtoD]
```

Fig. – nsys results 5-8 for Weight Matrix (Kernel) in Constant Memory

In Nsight Compute, the GPU SOL throughput has decreased, but this can be realized since the kernel is no longer being accessed globally. Even with a decreased throughput, it is still well balanced, and op times have still significantly improved because constant memory is faster. The lower throughput gives space for other optimizations as well. Additionally, looking at the memory workload analysis in the memory chart image, there is a higher memory throughput, 394.99 GB/S, compared to other optimizations that maximize overall throughput. Also, in the memory chart diagram, there is 95.20% hit rate on the L1/TEX Cache (because of the constant memory allocation) which compared to other optimizations is a significant improvement. The yellow icons state that there could be better memory coalescing to take advantage of a memory burst. To improve, the global access pattern would have to change so that the L1/TEX and L2 Cache are utilized effectively.



GPU Speed Of Light Throughput

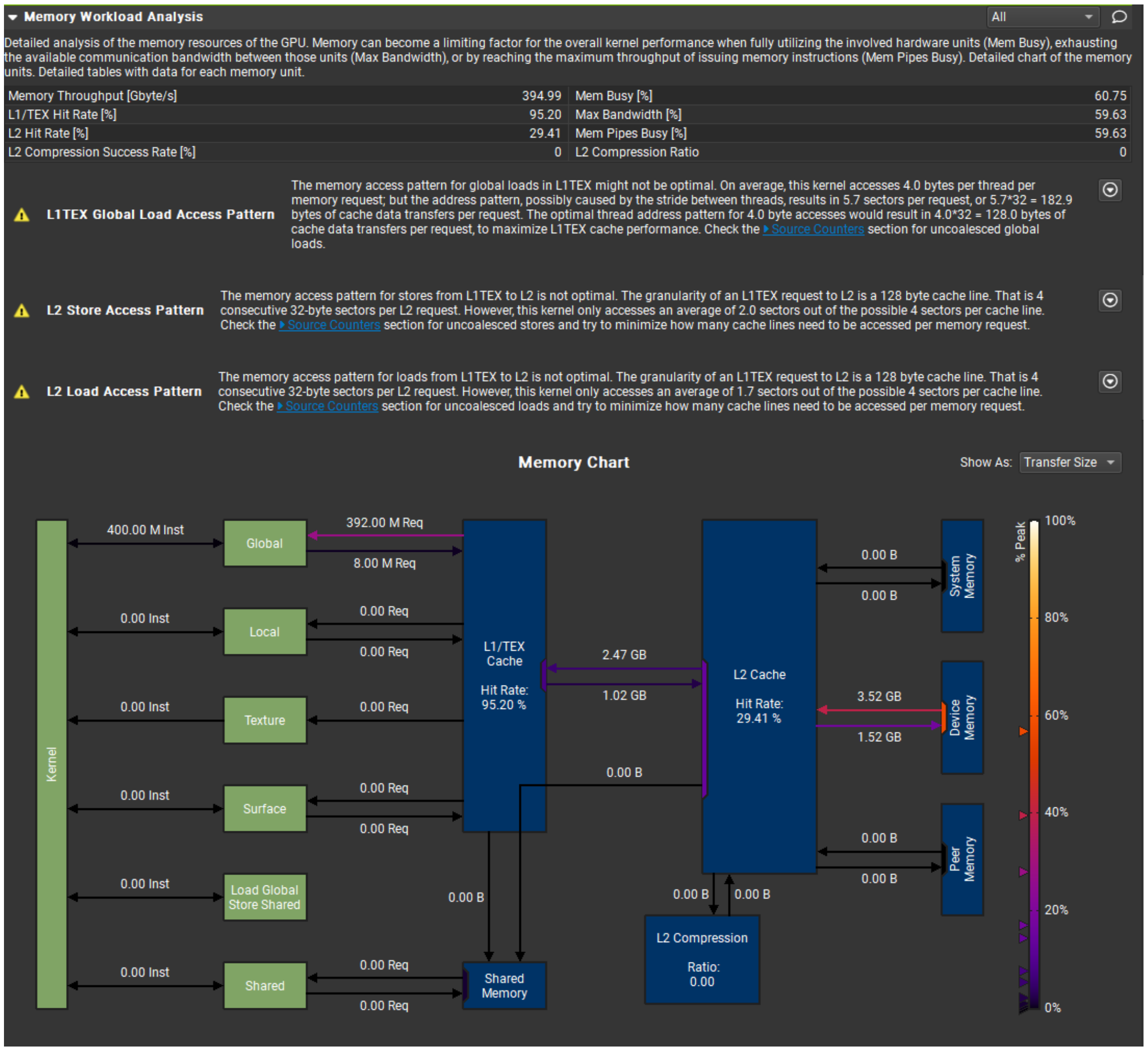
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	70.24	Duration [ms]	12.76
Memory Throughput [%]	60.75	Elapsed Cycles [cycle]	16,617,928
L1/TEX Cache Throughput [%]	60.77	SM Active Cycles [cycle]	16,603,774.04
L2 Cache Throughput [%]	14.50	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	56.91	DRAM Frequency [Ghz]	7.23

Balanced Throughput Compute and Memory are well-balanced: To reduce runtime, both computation and memory traffic must be reduced. Check both the Compute Workload Analysis and Memory Workload Analysis sections.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 7% of this device's fp32 peak performance and 5% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. - Weight Matrix (Kernel) in constant memory GPU SOL Throughput



Memory Workload Analysis

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]	394.99	Mem Busy [%]	60.75
L1/TEX Hit Rate [%]	95.20	Max Bandwidth [%]	59.63
L2 Hit Rate [%]	29.41	Mem Pipes Busy [%]	59.63
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L1TEX Global Load Access Pattern The memory access pattern for global loads in L1TEX might not be optimal. On average, this kernel accesses 4.0 bytes per thread per memory request; but the address pattern, possibly caused by the stride between threads, results in 5.7 sectors per request, or 5.7*32 = 182.9 bytes of cache data transfers per request. The optimal thread address pattern for 4.0 byte accesses would result in 4.0*32 = 128.0 bytes of cache data transfers per request, to maximize L1TEX cache performance. Check the Source Counters section for uncoalesced global loads.

L2 Store Access Pattern The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 2.0 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

L2 Load Access Pattern The memory access pattern for loads from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.7 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced loads and try to minimize how many cache lines need to be accessed per memory request.

Fig – Nsight Compute Memory Chart for Weight Matrix (Kernel) in constant memory

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

Yes, it synergizes with optimizations like using shared memory for input tiles, unrolling and matrix multiplication, loop unrolling, kernel fusion, etc. All these optimizations still require the use of the kernel, and since the values in the kernel do not change, using constant memory can reduce global memory accesses and improving data reuse, further enhancing performance. Taking advantage of the fast constant memory for all these optimizations is a clear decision.

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

    i. Slide Deck 8: Convolution and Constant Memory

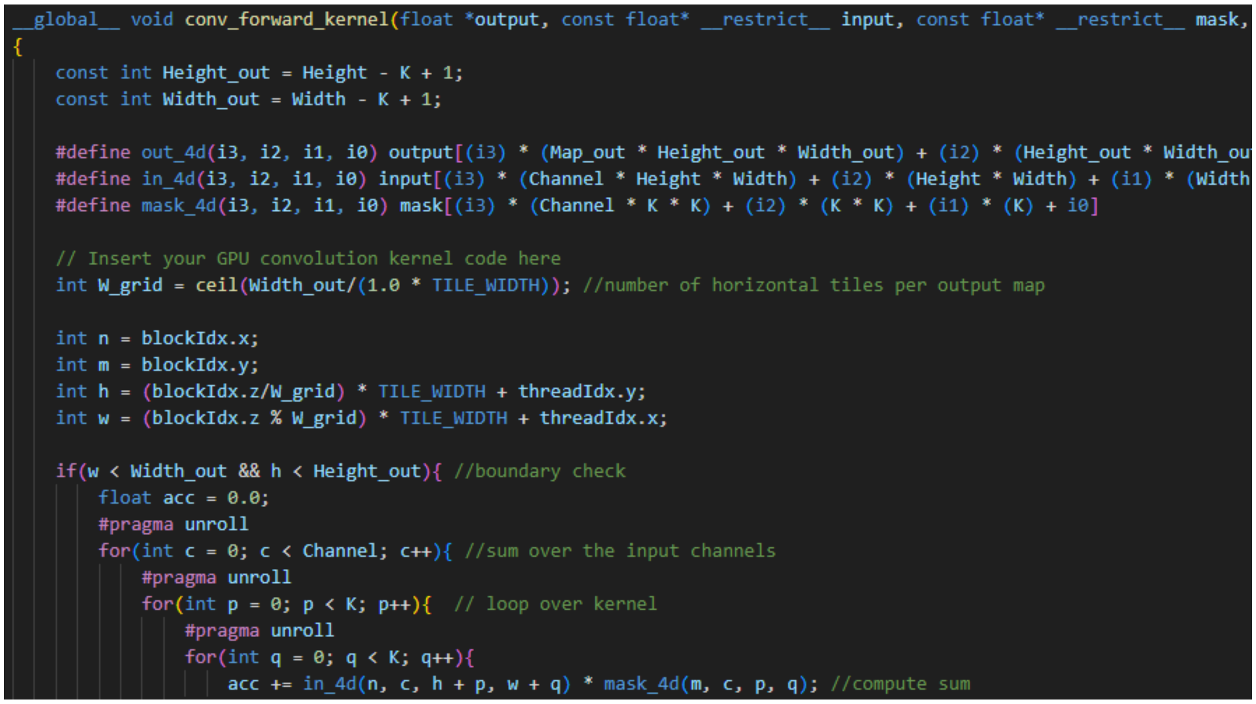
6. Optimization Number #5: Tuning with Restrict and Loop Unrolling

a. How does this optimization work in theory? Expected behavior?

For tuning with restrict and loop unrolling, the __restrict__ keyword informs the compiler that the object pointed to by the pointer will only be accessed by that pointer, ensuring no memory aliasing occurs. This is useful where memory bandwidth and access patterns impact performance. Loop unrolling reduces the overhead of loop control, decreasing conditional branching and increasing instruction throughput. Together, these optimizations are expected to enhance memory access efficiency and computational speed, leading to reduced execution times and increased throughput.

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

The implementation involves marking the input and mask pointers with the restrict qualifier to suggest that these pointers will not alias, allowing the compiler to generate more efficient memory access instructions. __restrtict__ is placed in front of the input and mask that are passed in as parameters to take advantage of the read-only cache. Loop unrolling is employed using #pragma unroll before loops. The compiler will unroll the loops reducing the need to check the conditional in the loop. #pragma unroll is placed before the loops that handle the convolution computation, effectively reducing the number of iterations and the overhead. Below is a code snippet demonstrating the use of restrict in the parameters and #pragma unroll before the loops.



```
__global__ void conv_forward_kernel(float *output, const float* __restrict__ input, const float* __restrict__ mask,
{
    const int Height_out = Height - K + 1;
    const int Width_out = Width - K + 1;

    #define out_4d(i3, i2, i1, i0) output[(i3) * (Map_out * Height_out * Width_out) + (i2) * (Height_out * Width_ou
    #define in_4d(i3, i2, i1, i0) input[(i3) * (Channel * Height * Width) + (i2) * (Height * Width) + (i1) * (Width
    #define mask_4d(i3, i2, i1, i0) mask[(i3) * (Channel * K * K) + (i2) * (K * K) + (i1) * (K) + i0]

    // Insert your GPU convolution kernel code here
    int W_grid = ceil(Width_out/(1.0 * TILE_WIDTH)); //number of horizontal tiles per output map

    int n = blockIdx.x;
    int m = blockIdx.y;
    int h = (blockIdx.z/W_grid) * TILE_WIDTH + threadIdx.y;
    int w = (blockIdx.z % W_grid) * TILE_WIDTH + threadIdx.x;

    if(w < Width_out && h < Height_out){ //boundary check
        float acc = 0.0;
        #pragma unroll
        for(int c = 0; c < Channel; c++){ //sum over the input channels
            #pragma unroll
            for(int p = 0; p < K; p++){  // loop over kernel
                #pragma unroll
                for(int q = 0; q < K; q++){
                    acc += in_4d(n, c, h + p, w + q) * mask_4d(m, c, p, q); //compute sum
```

Fig. – Code snippet of Tuning with restrict and loop unrolling

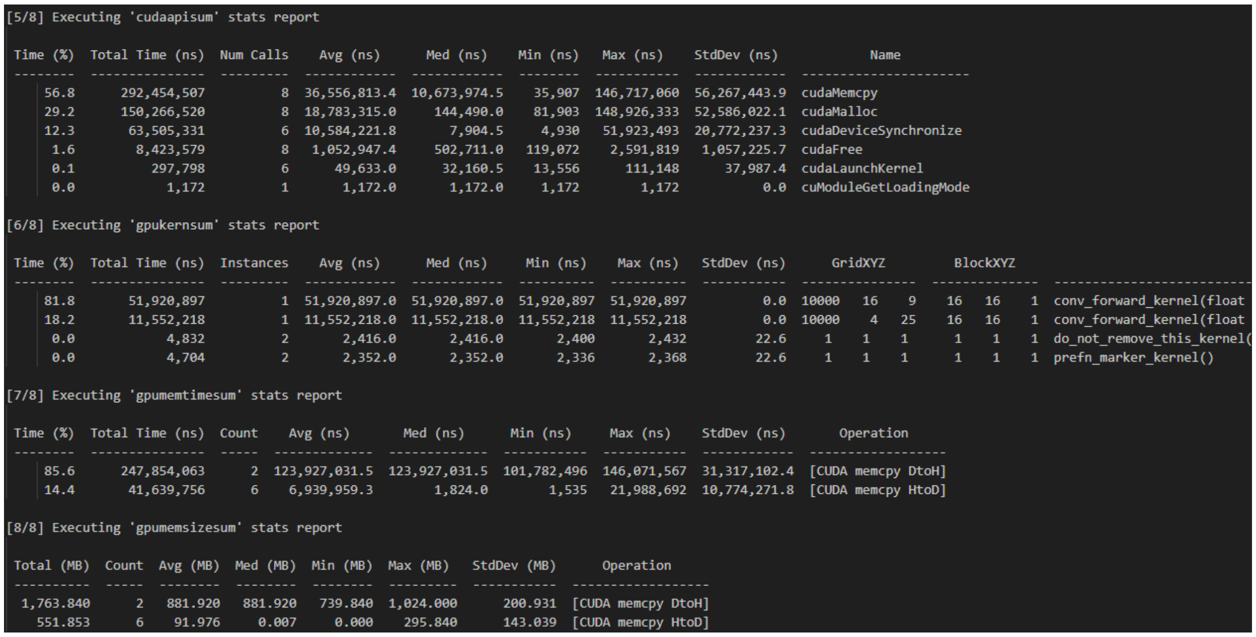
c. Did the performance match your expectation? Show your analysis results using profiling tools.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.173955 ms*	*0.50287 ms*	*0m4.024s*	*0.86*
1000	*1.20073 ms*	*4.93621 ms*	*0m36.389s*	*0.886*

10000	*11.5396 ms*	*51.7928 ms*	*5m59.084s*	*0.8714*

Based on the op times and total execution, there is a slight improvement in performance when tuning with restrict and loop unrolling compared to the baseline. This slight improvement was to be expected as the code hadn't changed and only restricts and unrolls were added which add slight performance benefits at a lower level.

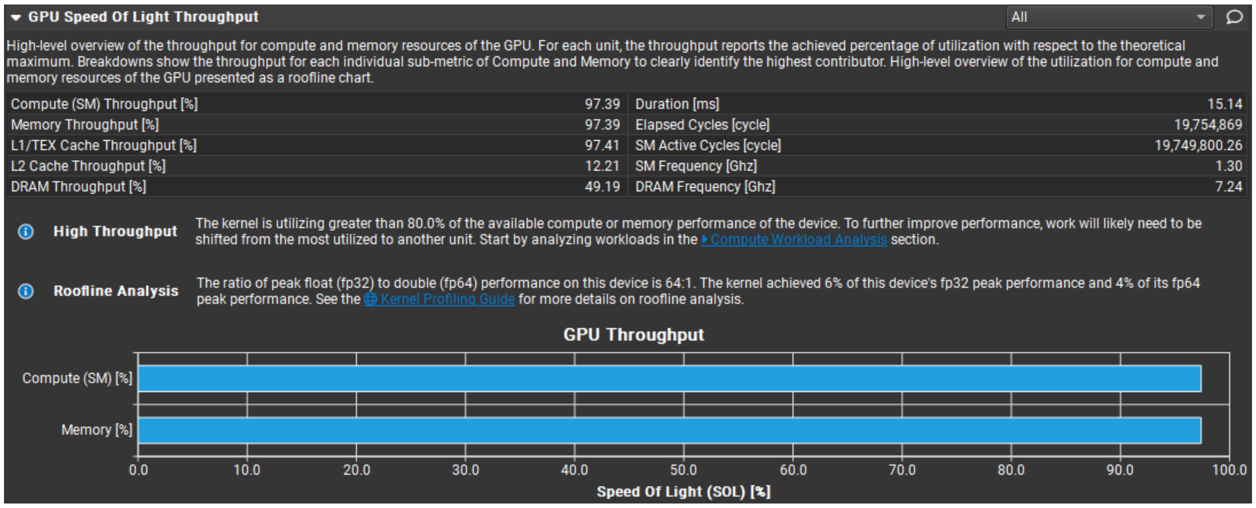
The nsys results show that overall kernel times slightly decrease. There is a decrease in gpumemtimesum compared to most optimizations. Memory time is expected to see a decrease as we restrict the inputsand unroll the loops to reduce overhead and utilize the read only cache.



```
[5/8] Executing 'cudaapisum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)     Min (ns)   Max (ns)     StdDev (ns)            Name
 --------  ---------------  ---------  ------------  ------------  --------  -----------  ------------  ----------------------
     56.8      292,454,507          8  36,556,813.4  10,673,974.5    35,907  146,717,060  56,267,443.9  cudaMemcpy
     29.2      150,266,520          8  18,783,315.0     144,490.0    81,903  148,926,333  52,586,022.1  cudaMalloc
     12.3       63,505,331          6  10,584,221.8       7,904.5     4,930   51,923,493  20,772,237.3  cudaDeviceSynchronize
      1.6        8,423,579          8   1,052,947.4     502,711.0   119,072    2,591,819   1,057,225.7  cudaFree
      0.1          297,798          6      49,633.0      32,160.5    13,556      111,148      37,987.4  cudaLaunchKernel
      0.0            1,172          1       1,172.0       1,172.0     1,172        1,172           0.0  cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)      Min (ns)    Max (ns)    StdDev (ns)       GridXYZ         BlockXYZ
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ---------------  --------------  --------------------------
     81.8       51,920,897          1  51,920,897.0  51,920,897.0  51,920,897  51,920,897          0.0  10000   16    9    16   16    1  conv_forward_kernel(float
     18.2       11,552,218          1  11,552,218.0  11,552,218.0  11,552,218  11,552,218          0.0  10000    4   25    16   16    1  conv_forward_kernel(float
      0.0            4,832          2       2,416.0       2,416.0       2,400       2,432         22.6      1    1    1     1    1    1  do_not_remove_this_kernel(
      0.0            4,704          2       2,352.0       2,352.0       2,336       2,368         22.6      1    1    1     1    1    1  prefn_marker_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)       Med (ns)       Min (ns)     Max (ns)     StdDev (ns)       Operation
 --------  ---------------  -----  -------------  -------------  -----------  -----------  ------------  ------------------
     85.6      247,854,063      2  123,927,031.5  123,927,031.5  101,782,496  146,071,567  31,317,102.4  [CUDA memcpy DtoH]
     14.4       41,639,756      6    6,939,959.3        1,824.0        1,535   21,988,692  10,774,271.8  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)   StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  ---------  -----------  ------------------
  1,763.840      2   881.920   881.920   739.840  1,024.000      200.931  [CUDA memcpy DtoH]
    551.853      6    91.976     0.007     0.000    295.840      143.039  [CUDA memcpy HtoD]
```

Fig. – nsys results 5-8 for Tuning with Restrict and Loop Unrolling

In the Nsight Compute analysis below, due to the similarity of the code to the baseline, the GPU throughput is very similar and is to be expected. The overall change isn't large, but there is a well balance high throughput. There could be improvement in the L2 load and store access pattern as seen in the memory workload analysis. The L2 cache has a 29.44% hit rate. Overall, the performance boost for this optimization simply comes from loop unrolling as there are fewer checks for the loop condition and the usage of the read only cache using restrict.



GPU Speed Of Light Throughput

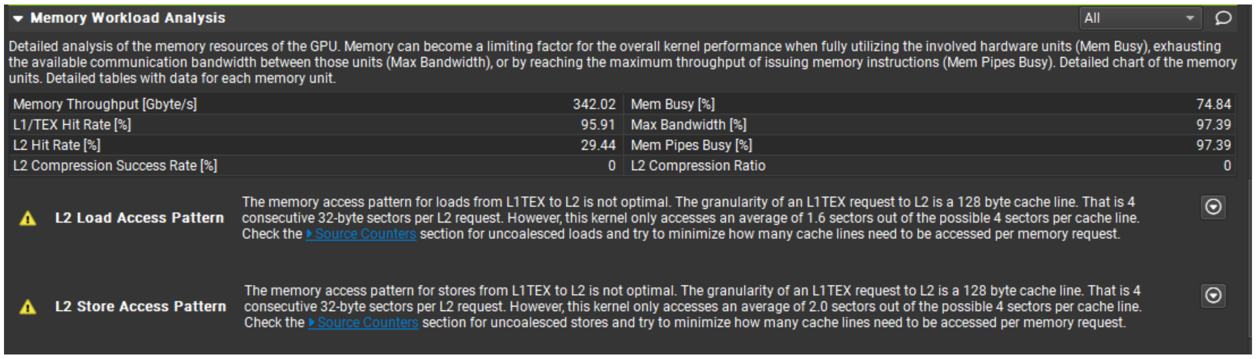
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	97.39	Duration [ms]	15.14
Memory Throughput [%]	97.39	Elapsed Cycles [cycle]	19,754,869
L1/TEX Cache Throughput [%]	97.41	SM Active Cycles [cycle]	19,749,800.26
L2 Cache Throughput [%]	12.21	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	49.19	DRAM Frequency [Ghz]	7.24

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 6% of this device's fp32 peak performance and 4% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. - Tuning with Restrict and Loop Unrolling GPU SOL Throughput



Memory Workload Analysis

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]	342.02	Mem Busy [%]	74.84
L1/TEX Hit Rate [%]	95.91	Max Bandwidth [%]	97.39
L2 Hit Rate [%]	29.44	Mem Pipes Busy [%]	97.39
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

L2 Load Access Pattern The memory access pattern for loads from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 1.6 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced loads and try to minimize how many cache lines need to be accessed per memory request.

L2 Store Access Pattern The memory access pattern for stores from L1TEX to L2 is not optimal. The granularity of an L1TEX request to L2 is a 128 byte cache line. That is 4 consecutive 32-byte sectors per L2 request. However, this kernel only accesses an average of 2.0 sectors out of the possible 4 sectors per cache line. Check the Source Counters section for uncoalesced stores and try to minimize how many cache lines need to be accessed per memory request.

Fig. – Memory Workload Analysis for Tuning with Restrict and Loop Unrolling

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

This optimization naturally synergizes with all the optimizations in the paper because unrolls could be added in front of most loops and restricts could easily be added to parameters that are only read. Furthermore, shared memory usage becomes more efficient when combined with restrict and loop unrolling, as the predictable access patterns and reduced loop overhead allow for faster and more consistent data handling.

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

   i. Slide Deck 22: Alternatives to CUDA

ii. CUDA C++ Programming Guide, Version 12.5, section 7.40 #pragma unroll

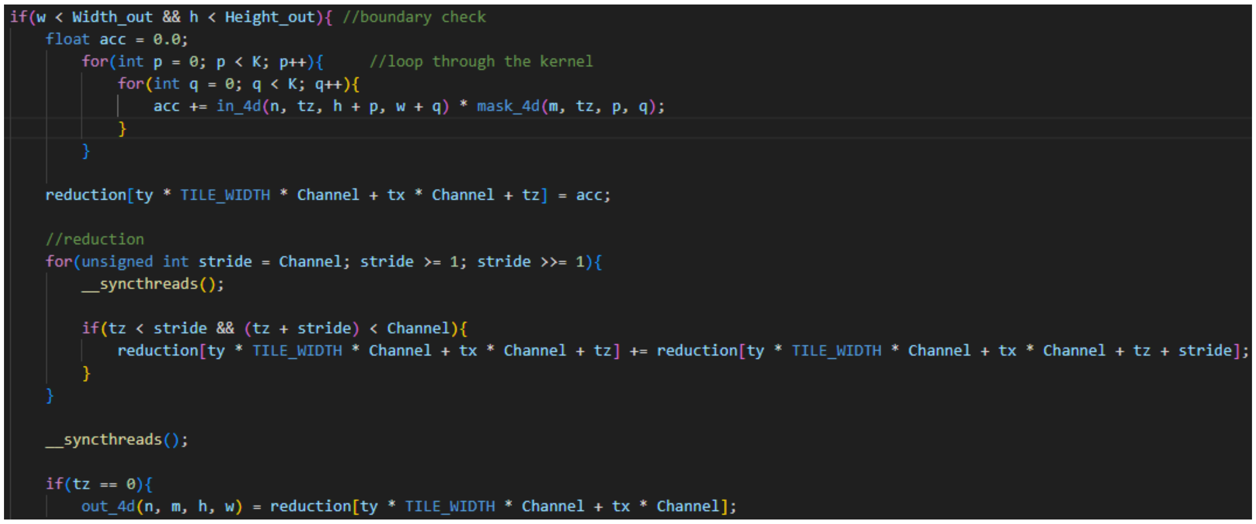
7. Optimization Number #8: Input Channel Reduction: Tree

a. How does this optimization work in theory? Expected behavior?

This optimization uses a reduction tree for collecting computation results across input channels. This method aims to quickly summarize the necessary data before finishing the calculation, making the entire process more efficient. The goal is to take the sum across all channels quickly to produce the output features faster. The expected benefits include a reduction in computational overhead. This optimization aims to minimize global memory accesses and accelerate the overall computation within the convolution process by efficiently summarizing the data before final accumulation.

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

This code performs a convolution and uses shared memory for efficiency when using the tree. It initializes shared memory, calculates the convolution sum for each thread, and stores it in shared memory. Then, it uses a tree-based reduction to combine these results, halving the contributors in each step until one result per block remains. This result is written to the output by the thread of index zero.



```
if(w < Width_out && h < Height_out){ //boundary check
    float acc = 0.0;
        for(int p = 0; p < K; p++){     //loop through the kernel
            for(int q = 0; q < K; q++){
                acc += in_4d(n, tz, h + p, w + q) * mask_4d(m, tz, p, q);
            }
        }

    reduction[ty * TILE_WIDTH * Channel + tx * Channel + tz] = acc;

    //reduction
    for(unsigned int stride = Channel; stride >= 1; stride >>= 1){
        __syncthreads();

        if(tz < stride && (tz + stride) < Channel){
            reduction[ty * TILE_WIDTH * Channel + tx * Channel + tz] += reduction[ty * TILE_WIDTH * Channel + tx * Channel + tz + stride];
        }
    }

    __syncthreads();

    if(tz == 0){
        out_4d(n, m, h, w) = reduction[ty * TILE_WIDTH * Channel + tx * Channel];
```

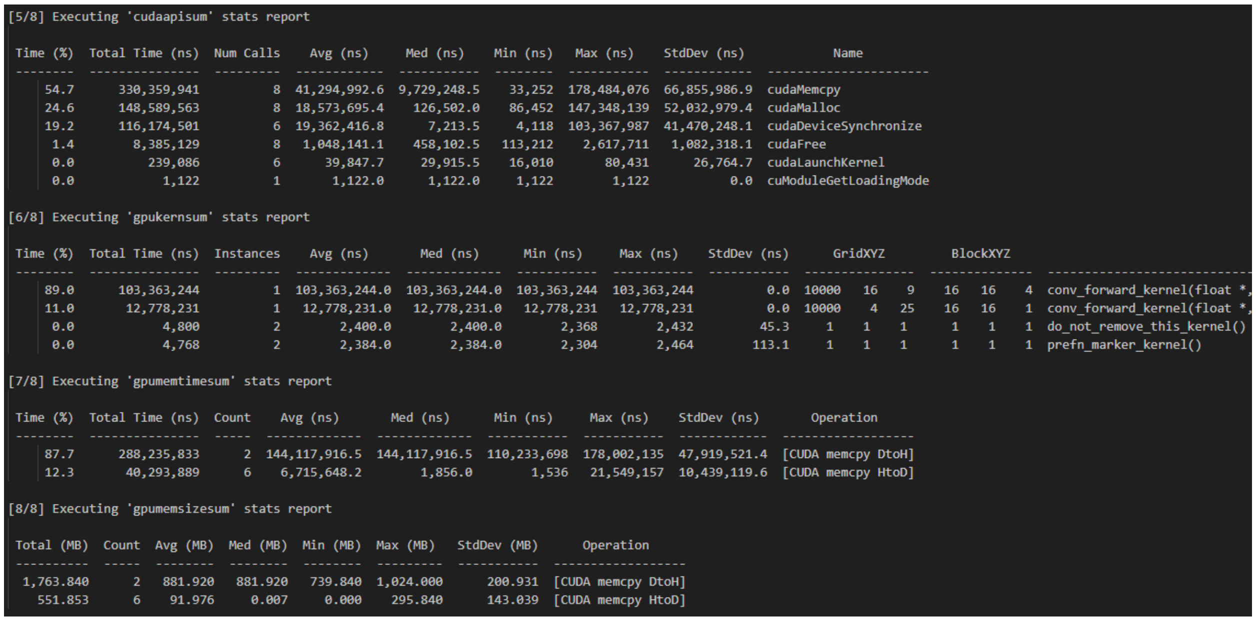
Fig. – Code snippet for Input Channel Reduction: Tree

c. Did the performance match your expectation? Show your analysis results using profiling tools.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.204883 ms*	*0.871011 ms*	*0m4.064s*	*0.86*
1000	*1.30605 ms*	*9.79366 ms*	*0m36.672s*	*0.886*
10000	*12.8414 ms*	*101.913 ms*	*6m5.675s*	*0.8714*

Despite its theoretical advantages, this optimization technique showed a slowdown in practice compared to the baseline. This could be due to frequent synchronization delays in the reduction steps, inefficient memory access patterns, and underutilization of device resources, especially since the number of channels is 4 in the second layer which isn't very optimal.

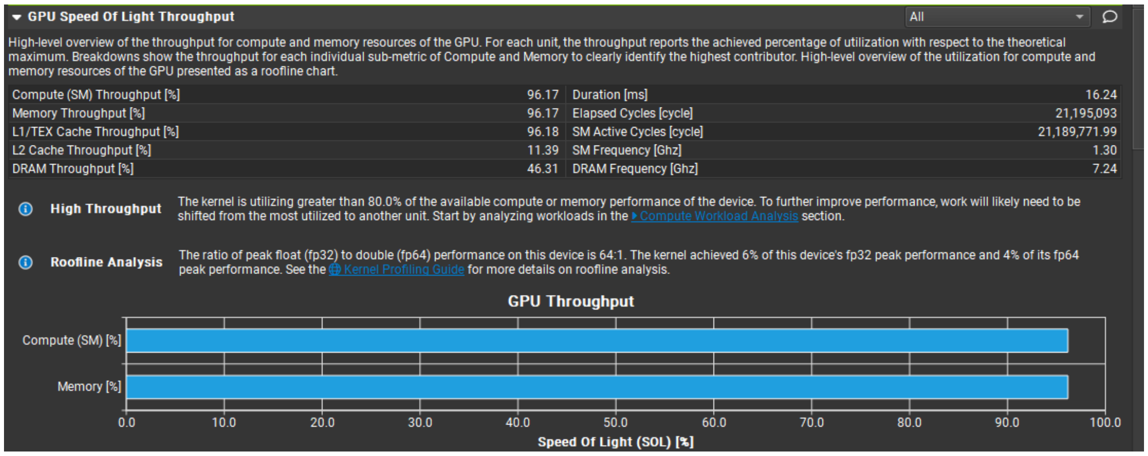
Looking at the nsys profiling, it can be noticed that the second layer in the CNN takes up most of the run time, while the first layer is quite fast. The first layer is faster as there is only one input channel so there is no need for a reduction amongst the input channels. The second layer is slow as there are only four input channels per image, so performing a reduction tree with four channels isn't worth it.



```
[5/8] Executing 'cudaapisum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)     Med (ns)    Min (ns)   Max (ns)    StdDev (ns)           Name
 --------  ---------------  ---------  ------------  -----------  --------  -----------  ------------  ----------------------
     54.7      330,359,941          8  41,294,992.6  9,729,248.5    33,252  178,484,076  66,855,986.9  cudaMemcpy
     24.6      148,589,563          8  18,573,695.4    126,502.0    86,452  147,348,139  52,032,979.4  cudaMalloc
     19.2      116,174,501          6  19,362,416.8      7,213.5     4,118  103,367,987  41,470,248.1  cudaDeviceSynchronize
      1.4        8,385,129          8   1,048,141.1    458,102.5   113,212    2,617,711   1,082,318.1  cudaFree
      0.0          239,086          6      39,847.7     29,915.5    16,010       80,431      26,764.7  cudaLaunchKernel
      0.0            1,122          1       1,122.0      1,122.0     1,122        1,122           0.0  cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)       Med (ns)      Min (ns)     Max (ns)    StdDev (ns)      GridXYZ         BlockXYZ
 --------  ---------------  ---------  -------------  -------------  -----------  -----------  -----------  --------------  --------------  ----------------------------
     89.0      103,363,244          1  103,363,244.0  103,363,244.0  103,363,244  103,363,244          0.0  10000   16    9    16   16    4  conv_forward_kernel(float *,
     11.0       12,778,231          1   12,778,231.0   12,778,231.0   12,778,231   12,778,231          0.0  10000    4   25    16   16    1  conv_forward_kernel(float *,
      0.0            4,800          2        2,400.0        2,400.0        2,368        2,432         45.3      1    1    1     1    1    1  do_not_remove_this_kernel()
      0.0            4,768          2        2,384.0        2,384.0        2,304        2,464        113.1      1    1    1     1    1    1  prefn_marker_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)       Med (ns)      Min (ns)     Max (ns)    StdDev (ns)       Operation
 --------  ---------------  -----  -------------  -------------  -----------  -----------  ------------  ------------------
     87.7      288,235,833      2  144,117,916.5  144,117,916.5  110,233,698  178,002,135  47,919,521.4  [CUDA memcpy DtoH]
     12.3       40,293,889      6    6,715,648.2        1,856.0        1,536   21,549,157  10,439,119.6  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)   StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  ---------  -----------  ------------------
  1,763.840      2   881.920   881.920   739.840  1,024.000      200.931  [CUDA memcpy DtoH]
    551.853      6    91.976     0.007     0.000    295.840      143.039  [CUDA memcpy HtoD]
```

Fig. – nsys profiling sections 5-8 for Input Channel Reduction: Tree

In Nsight Compute, it is first noticed that in the first layer, there is a high throughput, while in the second layer the throughput is quite low as there is a latency issue utilizing below 60% throughput. This low throughput could be due to the extremely large number of bank conflicts, 267,002,659, seen in the shared memory data for the second kernel launch (second layer). Additionally, in the second layer, the L2 cache has a low hit rate at 12.44% as shown in the memory workload analysis. It seems that when the reduction is carried out, there are many other complications especially with the use of shared memory and the large count of bank conflicts.



GPU Speed Of Light Throughput

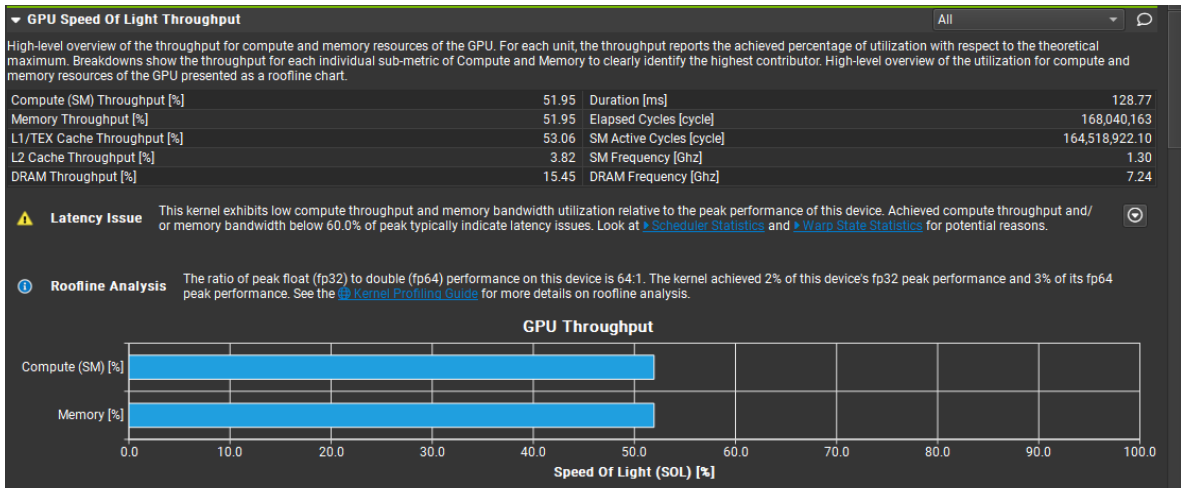
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	96.17	Duration [ms]	16.24
Memory Throughput [%]	96.17	Elapsed Cycles [cycle]	21,195,093
L1/TEX Cache Throughput [%]	96.18	SM Active Cycles [cycle]	21,189,771.99
L2 Cache Throughput [%]	11.39	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	46.31	DRAM Frequency [Ghz]	7.24

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 6% of this device's fp32 peak performance and 4% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. – GPU SOL Throughput for first layer of Input Channel Reduction: Tree



GPU Speed Of Light Throughput

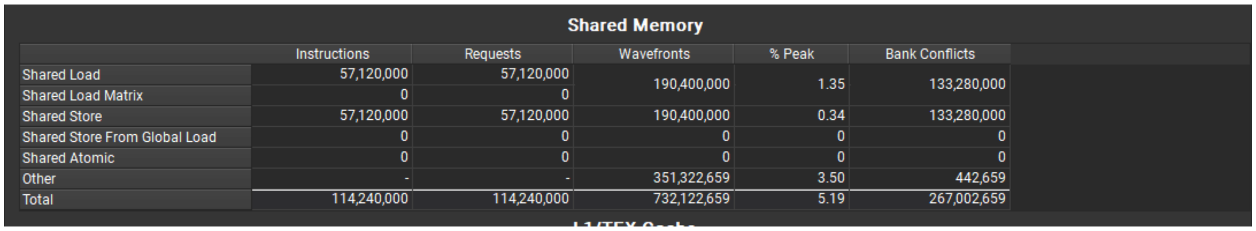
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	51.95	Duration [ms]	128.77
Memory Throughput [%]	51.95	Elapsed Cycles [cycle]	168,040,163
L1/TEX Cache Throughput [%]	53.06	SM Active Cycles [cycle]	164,518,922.10
L2 Cache Throughput [%]	3.82	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	15.45	DRAM Frequency [Ghz]	7.24

Latency Issue This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 2% of this device's fp32 peak performance and 3% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

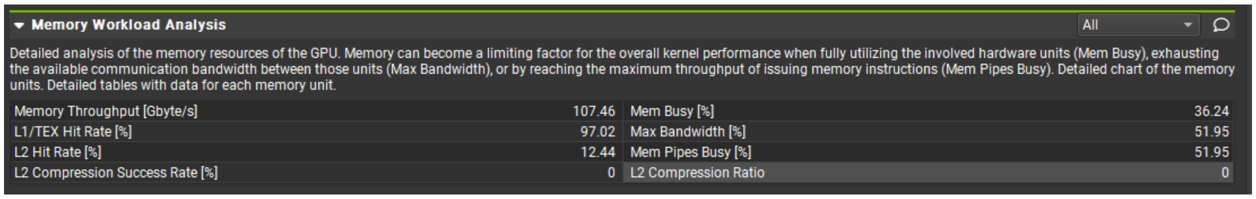
Fig. – GPU SOL Throughput for second layer of Input Channel Reduction: Tree



Shared Memory

	Instructions	Requests	Wavefronts	% Peak	Bank Conflicts
Shared Load	57,120,000	57,120,000	190,400,000	1.35	133,280,000
Shared Load Matrix	0	0			
Shared Store	57,120,000	57,120,000	190,400,000	0.34	133,280,000
Shared Store From Global Load	0	0	0	0	0
Shared Atomic	0	0	0	0	0
Other	-	-	351,322,659	3.50	442,659
Total	114,240,000	114,240,000	732,122,659	5.19	267,002,659

Fig. – Shared Memory Data for Input Channel Reduction: Tree



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]	107.46	Mem Busy [%]	36.24
L1/TEX Hit Rate [%]	97.02	Max Bandwidth [%]	51.95
L2 Hit Rate [%]	12.44	Mem Pipes Busy [%]	51.95
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

Fig. – Memory Workload Analysis for Input Channel Reduction: Tree

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

This reduction optimization complements optimizations such as shared memory optimization, which it already uses, and loop unrolling, which can further minimize loop overhead and maximize arithmetic operations per thread. These enhancements can collectively amplify the computational efficiency and reduce runtime. Additionally, with proper memory coalescing and reducing the number of bank conflicts, if there are more input channels, there could be performance benefits to using a tree.

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

i. Slide Deck 11: Reduction Trees

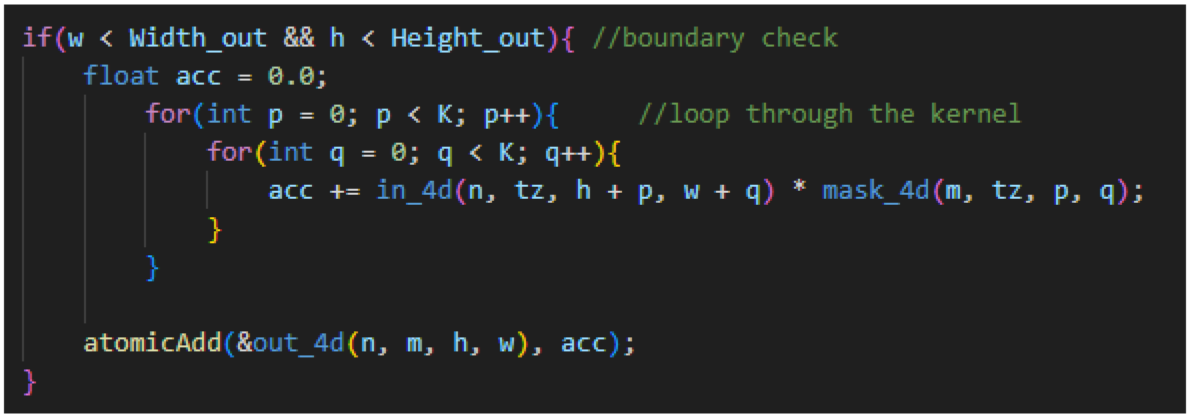
8. Optimization Number #9: *Input Channel Reduction: Atomics*

a. How does this optimization work in theory? Expected behavior?

The atomic operations optimization is designed to manage data contention effectively when multiple threads simultaneously attempt to write to the same location in memory. This approach is particularly useful for accumulating contributions from multiple channels to the output feature maps without needing a __syncthreads(). The expected behavior is a more efficient handling of writes to the output array where each thread updates the sum independently, reducing the overhead typically associated with thread synchronization.

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

The implementation utilizes atomic addition function (atomicAdd) to safely accumulate the results of the convolution from different threads into the output array. The code is very similar to optimization #8 except instead of using shared memory and a reduction, the reduction is replaced with an atomic add. This is crucial in scenarios where multiple threads compute results that contribute to the same output element, as it prevents race conditions.



```
if(w < Width_out && h < Height_out){ //boundary check
    float acc = 0.0;
        for(int p = 0; p < K; p++){     //loop through the kernel
            for(int q = 0; q < K; q++){
                acc += in_4d(n, tz, h + p, w + q) * mask_4d(m, tz, p, q);
            }
        }

    atomicAdd(&out_4d(n, m, h, w), acc);
}
```

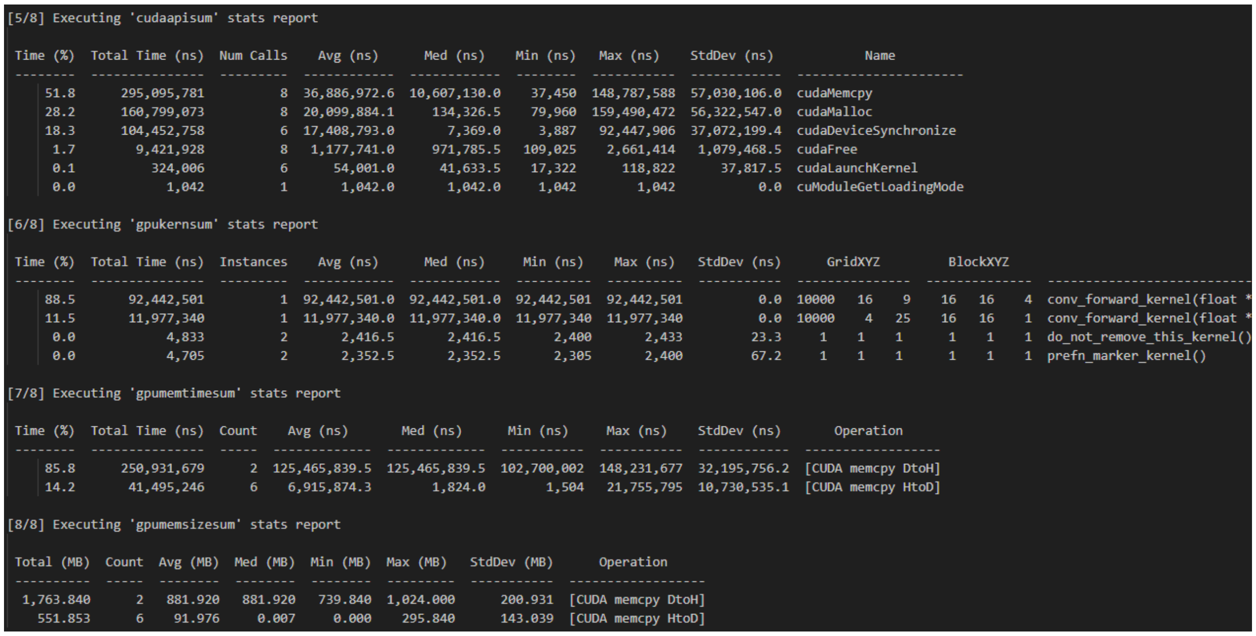
Fig. – Code snippet of Input Channel Reduction: Atomics

c. Did the performance match your expectation? Show your analysis results using profiling tools.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.177893 ms*	*0.789646 ms*	*0m4.011s*	*0.86*
1000	*1.23126 ms*	*8.65766 ms*	*0m36.655s*	*0.886*
10000	*11.9356 ms*	*92.0994 ms*	*6m3.722s*	*0.8714*

The use of atomic operations slows down overall op times. There is an increase in time in the second layer as there is a greater number of channels resulting in a larger number of atomic operations needed. The first layer only has one channel so there is only one atomic operation per output. This could be due to increased memory latency and the serialization of accesses.

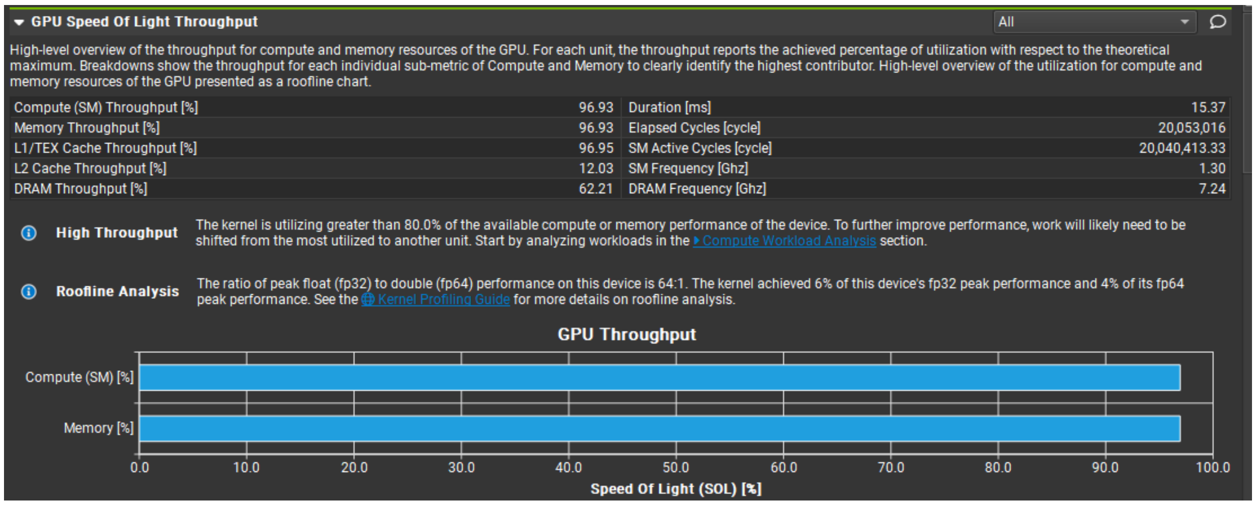
Viewing the nsys profiling below, there aren't many differences apart from the op times for the kernel, and the overall gpumemtimesum. The gpumemtimesumhas has decreased compared to other optimizations, and this could be due to fewer memory operations in separate location and the reduction of intermediate storage.



```
[5/8] Executing 'cudaapisum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)     Min (ns)   Max (ns)     StdDev (ns)           Name
 --------  ---------------  ---------  ------------  ------------  --------  -----------  ------------  ---------------------
     51.8      295,095,781          8  36,886,972.6  10,607,130.0    37,450  148,787,588  57,030,106.0  cudaMemcpy
     28.2      160,799,073          8  20,099,884.1     134,326.5    79,960  159,490,472  56,322,547.0  cudaMalloc
     18.3      104,452,758          6  17,408,793.0       7,369.0     3,887   92,447,906  37,072,199.4  cudaDeviceSynchronize
      1.7        9,421,928          8   1,177,741.0     971,785.5   109,025    2,661,414   1,079,468.5  cudaFree
      0.1          324,006          6      54,001.0      41,633.5    17,322      118,822      37,817.5  cudaLaunchKernel
      0.0            1,042          1       1,042.0       1,042.0     1,042        1,042           0.0  cuModuleGetLoadingMode

[6/8] Executing 'gpukernsum' stats report

 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)    StdDev (ns)      GridXYZ         BlockXYZ
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  --------------  --------------  ---------------------------
     88.5       92,442,501          1  92,442,501.0  92,442,501.0  92,442,501  92,442,501          0.0  10000    16    9    16    16    4  conv_forward_kernel(float *
     11.5       11,977,340          1  11,977,340.0  11,977,340.0  11,977,340  11,977,340          0.0  10000     4   25    16    16    1  conv_forward_kernel(float *
      0.0            4,833          2       2,416.5       2,416.5       2,400       2,433         23.3      1     1    1     1     1    1  do_not_remove_this_kernel()
      0.0            4,705          2       2,352.5       2,352.5       2,305       2,400         67.2      1     1    1     1     1    1  prefn_marker_kernel()

[7/8] Executing 'gpumemtimesum' stats report

 Time (%)  Total Time (ns)  Count    Avg (ns)       Med (ns)      Min (ns)     Max (ns)    StdDev (ns)       Operation
 --------  ---------------  -----  -------------  -------------  -----------  -----------  ------------  ------------------
     85.8      250,931,679      2  125,465,839.5  125,465,839.5  102,700,002  148,231,677  32,195,756.2  [CUDA memcpy DtoH]
     14.2       41,495,246      6    6,915,874.3        1,824.0        1,504   21,755,795  10,730,535.1  [CUDA memcpy HtoD]

[8/8] Executing 'gpumemsizesum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)   StdDev (MB)      Operation
 ----------  -----  --------  --------  --------  ---------  -----------  ------------------
  1,763.840      2   881.920   881.920   739.840  1,024.000      200.931  [CUDA memcpy DtoH]
    551.853      6    91.976     0.007     0.000    295.840      143.039  [CUDA memcpy HtoD]
```

Fig. – nsys profiling for Input Channel Reduction: Atomics

In Nsight Compute, the first layer has a high throughput while the second layer's throughput is nearly halved as shown below. These results are very similar to the previous optimization as they operate in a similar manner. While the first layer functions well, the second layer has several issues. As shown by the yellow icon in the throughput, there is a latency issue which could be due to the slower atomic operations. The L2 load access pattern is not optimal as well as the code could have a better use of memory coalescing. Additionally, in the scheduler statistics next to the yellow icon, it is shown that the scheduler only issues an instruction every 2.2 cycles, underutilizing device resources. It also mentions how there is a maximum of 12 warps per scheduler, but the kernel allocates 7.13 warps per scheduler, and only 1.11 warps were eligible within that cycle. Lastly, in the warp state statistics, thread divergence is shown to be an issue as the kernel only achieves 23.5 active threads per cycle. With many issues resulting in the use of atomic operations, atomic operations must only be used when absolutely needed and very carefully.



GPU Speed Of Light Throughput

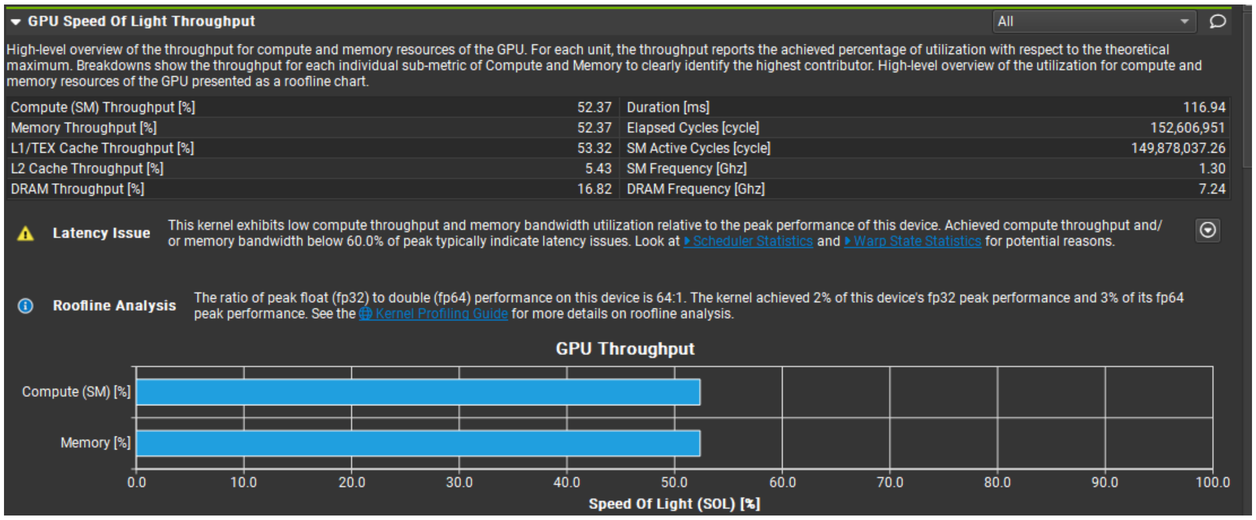
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	96.93	Duration [ms]	15.37
Memory Throughput [%]	96.93	Elapsed Cycles [cycle]	20,053,016
L1/TEX Cache Throughput [%]	96.95	SM Active Cycles [cycle]	20,040,413.33
L2 Cache Throughput [%]	12.03	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	62.21	DRAM Frequency [Ghz]	7.24

High Throughput The kernel is utilizing greater than 80.0% of the available compute or memory performance of the device. To further improve performance, work will likely need to be shifted from the most utilized to another unit. Start by analyzing workloads in the Compute Workload Analysis section.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 6% of this device's fp32 peak performance and 4% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. – GPU SOL Throughput for first layer of Input Channel Reduction: Atomics



GPU Speed Of Light Throughput

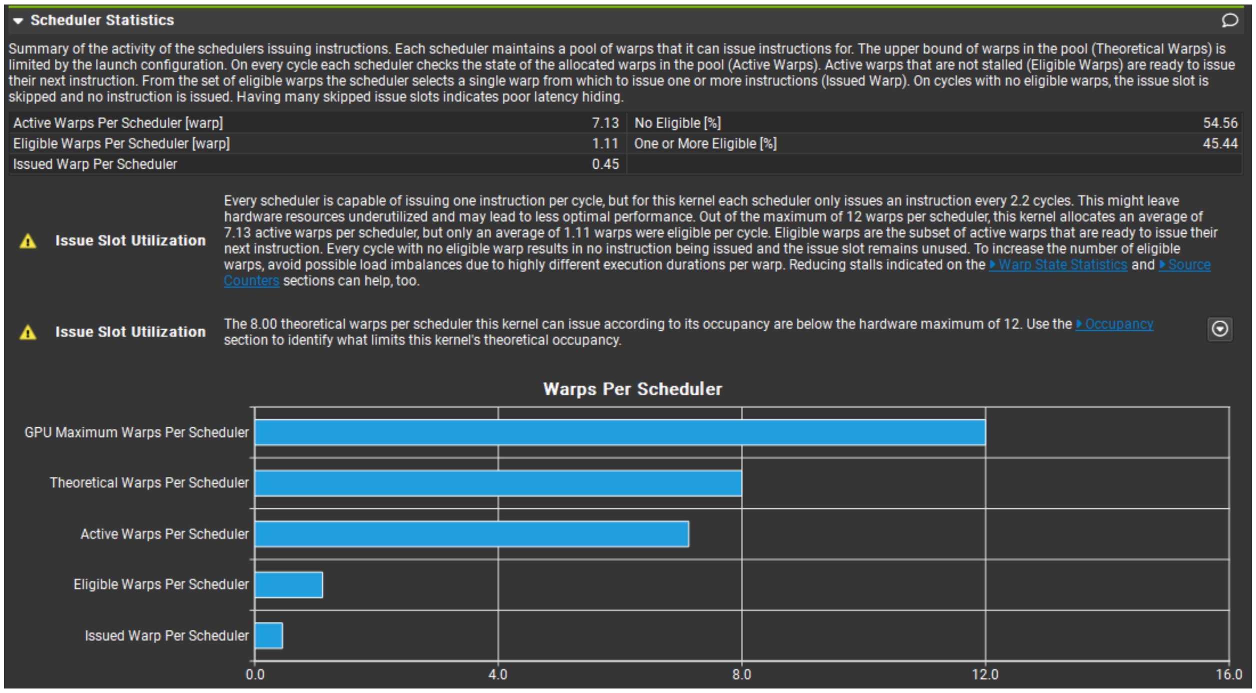
High-level overview of the throughput for compute and memory resources of the GPU. For each unit, the throughput reports the achieved percentage of utilization with respect to the theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview of the utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	52.37	Duration [ms]	116.94
Memory Throughput [%]	52.37	Elapsed Cycles [cycle]	152,606,951
L1/TEX Cache Throughput [%]	53.32	SM Active Cycles [cycle]	149,878,037.26
L2 Cache Throughput [%]	5.43	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	16.82	DRAM Frequency [Ghz]	7.24

Latency Issue This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of this device. Achieved compute throughput and/or memory bandwidth below 60.0% of peak typically indicate latency issues. Look at Scheduler Statistics and Warp State Statistics for potential reasons.

Roofline Analysis The ratio of peak float (fp32) to double (fp64) performance on this device is 64:1. The kernel achieved 2% of this device's fp32 peak performance and 3% of its fp64 peak performance. See the Kernel Profiling Guide for more details on roofline analysis.

Fig. – GPU SOL Throughput for second layer of Input Channel Reduction: Atomics


Scheduler Statistics

Summary of the activity of the schedulers issuing instructions. Each scheduler maintains a pool of warps that it can issue instructions for. The upper bound of warps in the pool (Theoretical Warps) is limited by the launch configuration. On every cycle each scheduler checks the state of the allocated warps in the pool (Active Warps). Active warps that are not stalled (Eligible Warps) are ready to issue their next instruction. From the set of eligible warps the scheduler selects a single warp from which to issue one or more instructions (Issued Warp). On cycles with no eligible warps, the issue slot is skipped and no instruction is issued. Having many skipped issue slots indicates poor latency hiding.

Active Warps Per Scheduler [warp]	7.13	No Eligible [%]	54.56
Eligible Warps Per Scheduler [warp]	1.11	One or More Eligible [%]	45.44
Issued Warp Per Scheduler	0.45		

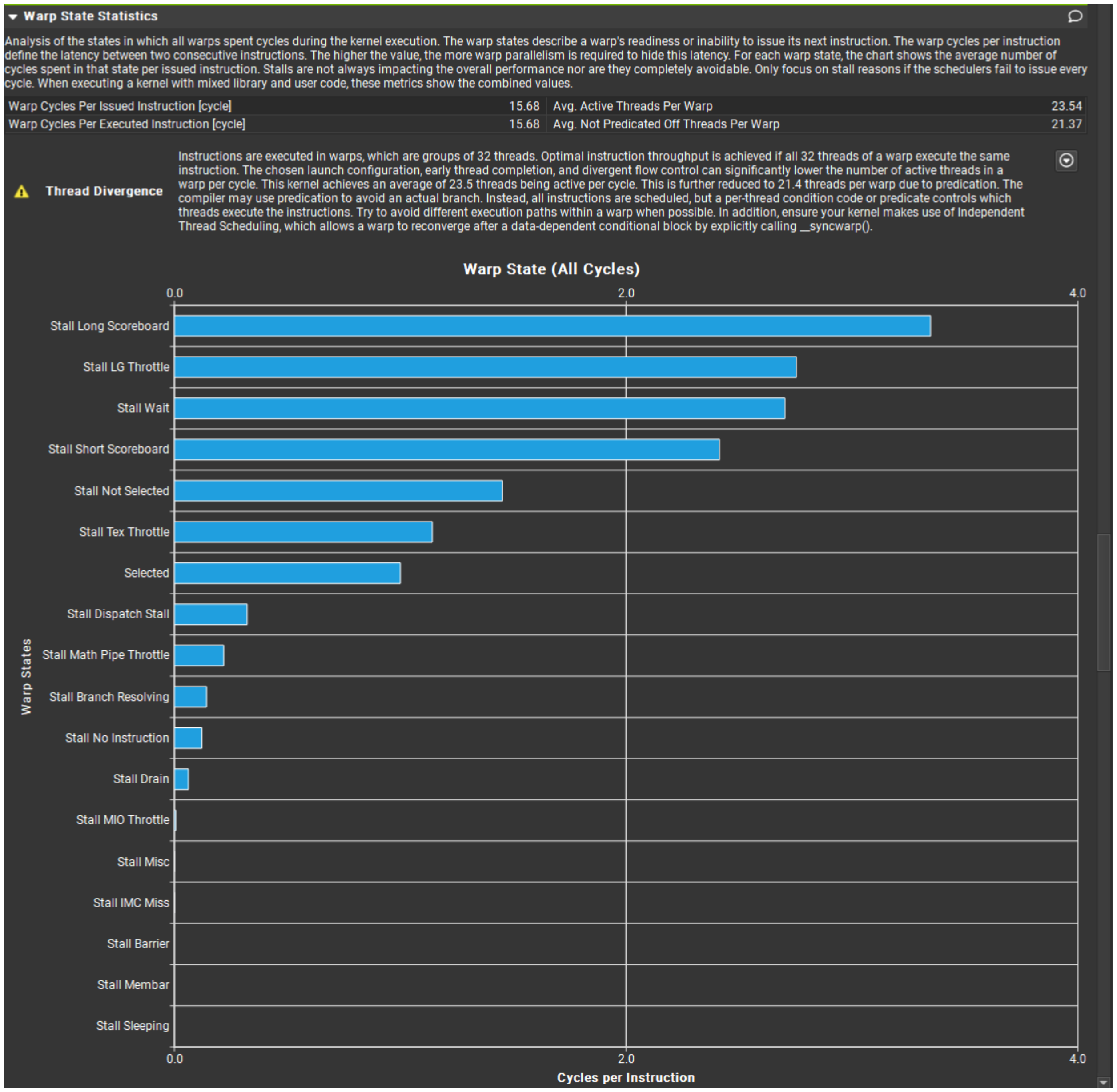
Issue Slot Utilization Every scheduler is capable of issuing one instruction per cycle, but for this kernel each scheduler only issues an instruction every 2.2 cycles. This might leave hardware resources underutilized and may lead to less optimal performance. Out of the maximum of 12 warps per scheduler, this kernel allocates an average of 7.13 active warps per scheduler, but only an average of 1.11 warps were eligible per cycle. Eligible warps are the subset of active warps that are ready to issue their next instruction. Every cycle with no eligible warp results in no instruction being issued and the issue slot remains unused. To increase the number of eligible warps, avoid possible load imbalances due to highly different execution durations per warp. Reducing stalls indicated on the Warp State Statistics and Source Counters sections can help, too.

Issue Slot Utilization The 8.00 theoretical warps per scheduler this kernel can issue according to its occupancy are below the hardware maximum of 12. Use the Occupancy section to identify what limits this kernel's theoretical occupancy.

Warps Per Scheduler
GPU Maximum Warps Per Scheduler
Theoretical Warps Per Scheduler
Active Warps Per Scheduler
Eligible Warps Per Scheduler
Issued Warp Per Scheduler
0.0
4.0
8.0
12.0
16.0

Fig. – Scheduler Statistics for second layer of Input Channel Reduction: Atomics

Image on next page



Warp State Statistics

Analysis of the states in which all warps spent cycles during the kernel execution. The warp states describe a warp's readiness or inability to issue its next instruction. The warp cycles per instruction define the latency between two consecutive instructions. The higher the value, the more warp parallelism is required to hide this latency. For each warp state, the chart shows the average number of cycles spent in that state per issued instruction. Stalls are not always impacting the overall performance nor are they completely avoidable. Only focus on stall reasons if the schedulers fail to issue every cycle. When executing a kernel with mixed library and user code, these metrics show the combined values.

Warp Cycles Per Issued Instruction [cycle]	15.68	Avg. Active Threads Per Warp	23.54
Warp Cycles Per Executed Instruction [cycle]	15.68	Avg. Not Predicated Off Threads Per Warp	21.37

Thread Divergence

Instructions are executed in warps, which are groups of 32 threads. Optimal instruction throughput is achieved if all 32 threads of a warp execute the same instruction. The chosen launch configuration, early thread completion, and divergent flow control can significantly lower the number of active threads in a warp per cycle. This kernel achieves an average of 23.5 threads being active per cycle. This is further reduced to 21.4 threads per warp due to predication. The compiler may use predication to avoid an actual branch. Instead, all instructions are scheduled, but a per-thread condition code or predicate controls which threads execute the instructions. Try to avoid different execution paths within a warp when possible. In addition, ensure your kernel makes use of Independent Thread Scheduling, which allows a warp to reconverge after a data-dependent conditional block by explicitly calling __syncwarp().

Fig. – Warp State Statistics for second layer of Input Channel Reduction: Atomics

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

Yes, this optimization synergizes well with optimizations that involve segmenting work among multiple threads, such as tiling or using shared memory. By reducing the granularity of the data each thread works on and then combining results via atomics, the overall memory coherence and execution efficiency can be enhanced. It also pairs well with optimizations that aim to balance the load among threads, ensuring that the atomic operations do not become a bottleneck. Since many of the optimizations such as tiling and tuning with restrict and loop unrolling require threads to write to output features simultaneously, atomic operations could be used. It is important to use atomic operations only when needed as they have many drawbacks.

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

i. Slide Deck 13: Atomic Operations and Histograms