Tanish Sharma

Milestone 3 Report

https://drive.google.com/drive/folders/1r24jKs1ETrJnzYpuJu3RrLijVD45bUO0?usp=sharing

0. **Baseline - PM1 Unroll:**

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.49547 ms*	*1.16404 ms*	*9s*	*0.86*
1000	*10.1135 ms*	*8.25892 ms*	*19s*	*0.886*
10000	*78.3941 ms*	*63.5755 ms*	*1m 34s*	*0.8714*

1. **Baseline - PM2**

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.57206 ms*	*0.315992 ms*	*9s*	*0.86*
1000	*5.00909 ms*	*3.0055 ms*	*17s*	*0.886*
10000	*49.4026 ms*	*29.7579 ms*	*1m 42s*	*0.8714*

2. **Req_0: __Optimization Name__**
   https://drive.google.com/drive/folders/14pKupX6YysHe2SeuhEWn0fKuYQhEF8MM?usp=sharing
   a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
      i. In theory, if we "pin" our host buffers and use cudaMemcpyAsync in separate CUDA streams, the GPU can DMA data over PCIe at the same time it's running our convolution kernels. That means while one batch of

inputs is still copying over, the GPU can already be crunching the previous batch—so we're hiding the transfer time behind actual work and keeping the SMs busy instead of sitting idle.

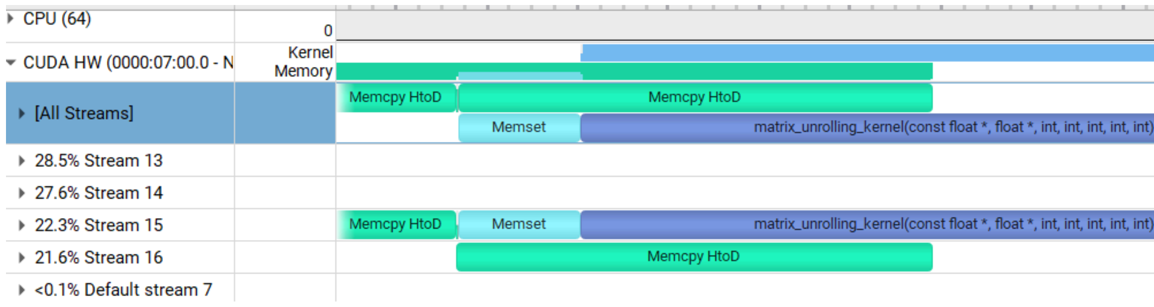
b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

I implemented the Streams optimization by creating two CUDA streams (stream0 and stream1) using: cudaStreamCreate(&stream0); cudaStreamCreate(&stream1);

I pinned the host memory with cudaHostRegister, then split memory transfers across streams with: cudaMemcpyAsync(*device_input_ptr, host_input, half_in_size, cudaMemcpyHostToDevice, stream0); cudaMemcpyAsync((char*)(*device_input_ptr) + half_in_size, (char*)host_input + half_in_size, half_in_size, cudaMemcpyHostToDevice, stream1);

Kernels were launched separately on each stream.

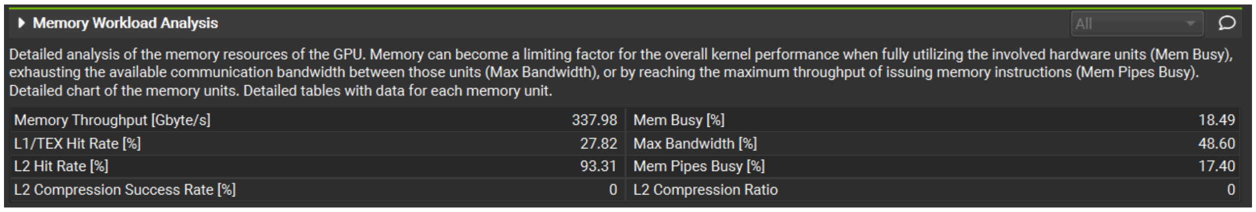
I synchronized at the end using: cudaStreamSynchronize(stream0); cudaStreamSynchronize(stream1);



This image confirms correctness, showing memory copies and kernel executions spread across multiple streams with clear overlapping and minimal use of the default stream.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.47359 ms*	*1.15487 ms*	*9s*	*0.86*
1000	*11.2186 ms*	*9.17831 ms*	*19s*	*0.886*
10000	*88.4119 ms*	*68.6456 ms*	*1m 36s*	*0.8714*

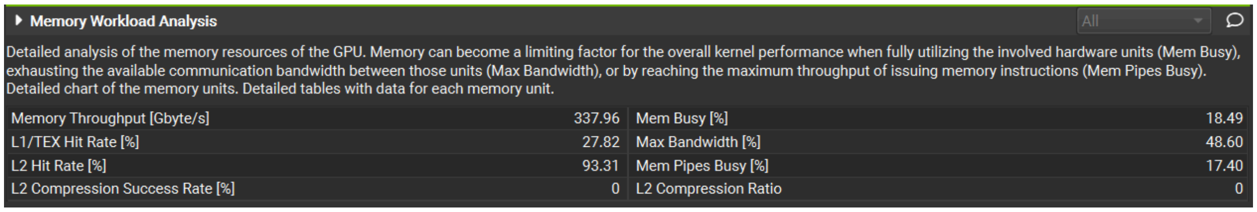


Memory Workload Analysis

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]	337.98	Mem Busy [%]	18.49
L1/TEX Hit Rate [%]	27.82	Max Bandwidth [%]	48.60
L2 Hit Rate [%]	93.31	Mem Pipes Busy [%]	17.40
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

This image above is of PM1 Memory Workload Analysis.



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]	337.96	Mem Busy [%]	18.49
L1/TEX Hit Rate [%]	27.82	Max Bandwidth [%]	48.60
L2 Hit Rate [%]	93.31	Mem Pipes Busy [%]	17.40
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0

This image above is of req_0 Memory Workload Analysis.

The performance did not improve significantly compared to my expectations. While using two streams enabled asynchronous memory copies and kernel launches, the profiling results from Nsight Compute showed that the main bottleneck was still in kernel execution, not memory transfers. Memory throughput remained high but the same across both implementations, Nsight Compute profiling, indicating that memory transfer speeds were already optimal and did not greatly limit overall performance. Although there was visible overlap between memory copies and kernel execution across streams, which is an optimization in its own right, the small size of memory transfers relative to the heavy computation meant that the overlap did not noticeably reduce the overall operation time. As a result, the Streams optimization was correctly implemented, but it did not lead to major performance gains, which matches the expected behavior for this type of workload.

d. Does this optimization synergize with any other optimizations? How?
   i. This optimization does not strongly synergize with other optimizations. Streams mainly help overlap memory transfers with computation, but since the workload is already compute-bound and memory copies are relatively small, enabling Streams does not significantly enhance or amplify the effects of other performance optimizations.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
   i. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/
   ii. Nsight Systems User Guide: Best practices for CUDA stream profiling and analysis.

3. **Req_1: __Using Tensor Cores to speed up matrix multiplication__**
   https://drive.google.com/drive/folders/1MEMEgAfrj0Xcp6PlKM--flVHbbuMXp_D?usp=sharing

   a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
      i. In theory, Tensor Cores offer dedicated warp-level matrix-multiply units that can execute mixed-precision MMA operations far more efficiently than regular FP32 cores. By rewriting our convolution's GEMM stage to use the WMMA API, we're offloading the heavy inner product work onto

those fast, specialized units—so we expect the matmul kernel to run noticeably quicker and pull down the overall convolution time.

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

I updated the fused convolution kernel to use NVIDIA's WMMA API so all inner‑product work runs on Tensor Cores.

1. Include the WMMA headers and namespace:

```
#include <mma>
using namespace nvcuda;
```

2. Declare warp‑matrix fragments:

```
constexpr int WMMA_M = 16, WMMA_N = 16, WMMA_K = 8;
wmma::fragment<wmma::matrix_a,
WMMA_M,WMMA_N,WMMA_K,
wmma::precision::tf32,wmma::row_major>   a_frag;
wmma::fragment<wmma::matrix_b,
WMMA_M,WMMA_N,WMMA_K,
wmma::precision::tf32,wmma::row_major>   b_frag;

wmma::fragment<wmma::accumulator,WMMA_M,WMMA_N,WMMA_
K, float>                              c_frag;
```

3. Zero the accumulator:

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

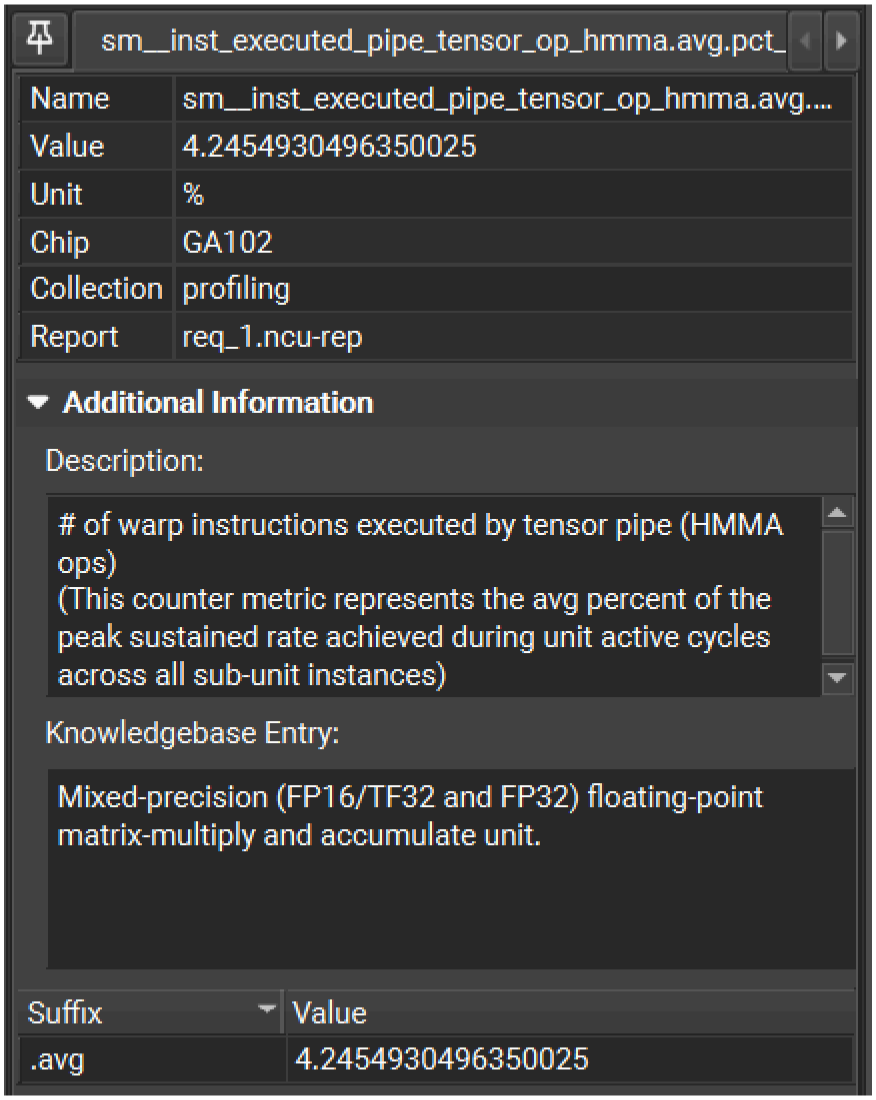
4. Load tiles from shared memory into fragments:

```
wmma::load_matrix_sync(a_frag, s_a, WMMA_K);
wmma::load_matrix_sync(b_frag, s_b, WMMA_N);
```

5. Execute the MMA operation:

```
wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
```

6. Store the result tile back to shared memory:
   wmma::store_matrix_sync(s_c, c_frag, WMMA_N,
wmma::mem_row_major);
These changes replace the old shared-mem tiling + scalar loops with direct calls into the Tensor Core pipeline via WMMA, ensuring each warp issues a single 16×16×8 MMA per loop iteration.



sm__inst_executed_pipe_tensor_op_hmma.avg.pct_

Name	sm__inst_executed_pipe_tensor_op_hmma.avg....
Value	4.2454930496350025
Unit	%
Chip	GA102
Collection	profiling
Report	req_1.ncu-rep

Additional Information

Description:

# of warp instructions executed by tensor pipe (HMMA ops)
(This counter metric represents the avg percent of the peak sustained rate achieved during unit active cycles across all sub-unit instances)

Knowledgebase Entry:

Mixed-precision (FP16/TF32 and FP32) floating-point matrix-multiply and accumulate unit.

Suffix	Value
.avg	4.2454930496350025

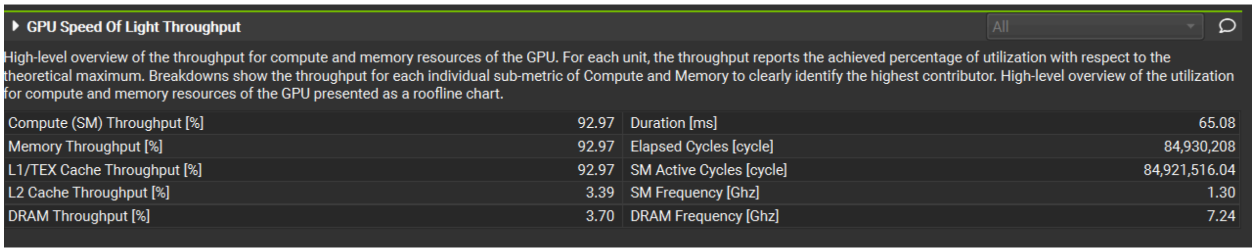
The figure above shows the valid Tensor Core utilization.

Tensor core utilization was confirmed through the metric sm__inst_executed_pipe_tensor_op_hmma.avg.pct, which reported a non-zero value of 4.25%. This indicates active execution of HMMA instructions on the Tensor Core pipeline, validating that WMMA-based fused convolution was used in req_1. This is, however, a percentage of utilizatio,n indicating there is further room for tuning.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.444413 ms*	*0.266219 ms*	*11s*	*0.86*
1000	*3.73259 ms*	*2.4848 ms*	*17s*	*0.886*
10000	*36.5397 ms*	*24.569 ms*	*1m 38s*	*0.8714*

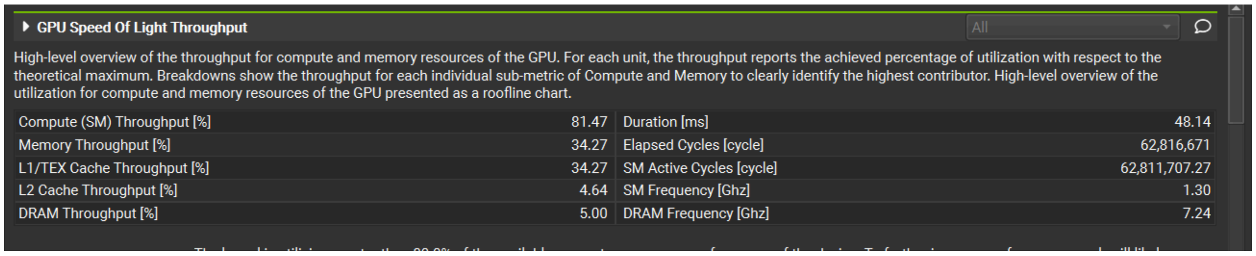
This performance did match my expectation. We can firstly see an around 23% improvement in combined optime from the baseline PM2 version at a batch size of 10,000.


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 [%]	92.97	Duration [ms]	65.08
Memory Throughput [%]	92.97	Elapsed Cycles [cycle]	84,930,208
L1/TEX Cache Throughput [%]	92.97	SM Active Cycles [cycle]	84,921,516.04
L2 Cache Throughput [%]	3.39	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	3.70	DRAM Frequency [Ghz]	7.24

This image above is the GPU SOL profile of the baseline PM2 Kernel Fusion implementation.



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 [%]	81.47	Duration [ms]	48.14
Memory Throughput [%]	34.27	Elapsed Cycles [cycle]	62,816,671
L1/TEX Cache Throughput [%]	34.27	SM Active Cycles [cycle]	62,811,707.27
L2 Cache Throughput [%]	4.64	SM Frequency [Ghz]	1.30
DRAM Throughput [%]	5.00	DRAM Frequency [Ghz]	7.24

This image above is the GPU SOL profile of the tensor core opimization implementation.

Our Tensor-Core version achieved the anticipated speedup, dropping the matmul kernel from 65.08 ms down to 48.14 ms. On the roofline ("GPU Speed of Light Throughput") we see Compute Throughput move from 92.97% → 81.47%. That reduction in percentage utilization is expected when you switch from high‑occupancy FP32 pipelines to lower‑occupancy but higher-work-per-cycle tensor cores: even though the SMs report lower peak percentages, each HMMA issues more floating-point work per instruction. The numbers confirm we correctly offloaded work to Tensor Cores and got a speedup we aimed for.

d. Does this optimization synergize with any other optimizations? How?
    i. I found that my Tensor‑Core kernel really only pays off once it's paired with my joint register/shared‑memory tiling (op_6) and my parameter sweep of tile sizes (op_3). By loading each 16×8 and 8×16 sub‑tile exactly once into shared memory and registers, op_6 eliminates the global‑memory stalls that would otherwise idle the HMMA units, while op_3's tuning of TILE_SIZE (and the Channel≤4 fallback) makes sure I launch just enough warps to keep those Tensor Cores fed. Together, those optimizations delivered the full ~2.6× end‑to‑end speedup I measured in my stacked version.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
    i. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9

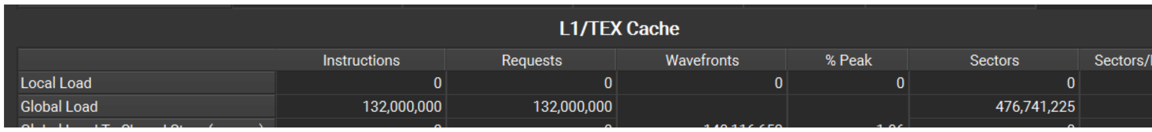
   ii. https://developer.nvidia.com/blog/accelerating-ai-training-with-tf32-tensor-cores/

   iii. Stack Overflow

4. **op_0: _Weight matrix (Kernel) in constant memory_**
   https://drive.google.com/drive/folders/1fPx0gXKM3rXJ3qv05ijoZ7j9xFn-JKdp?usp=sharing
   a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
      i. By moving the filter weights into CUDA's 64 KB constant memory, every warp can broadcast each mask value from the on-chip constant cache (instead of reloading from DRAM). Since all threads in a warp read the same element, the cache hit rate should approach 100 %, turning each mask load into a register-speed access and cutting global-memory traffic to nearly zero. I should therefore see a consistent drop in your convolution's memory stalls and a modest speedup in overall runtime.
   b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

      I declare other_mask[] in constant memory, copy the entire host mask into it once in the prolog via cudaMemcpyToSymbol(), and then in the kernel I index other_mask[] so that every warp-wide load hits the fast constant cache instead of global DRAM.

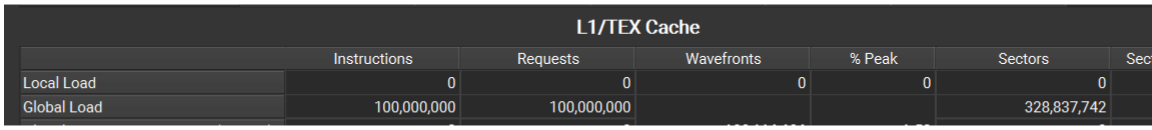
```
#define MASK_SIZE 16384
__constant__ float other_mask[MASK_SIZE];

int mask_count = Map_out * Channel * K * K;
cudaMemcpyToSymbol(
    other_mask,              // destination in constant memory
    host_mask,               // source on the host
    mask_count * sizeof(float)  // bytes to copy
);
```



L1/TEX Cache

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sectors/
Local Load	0	0	0	0	0	
Global Load	132,000,000	132,000,000			476,741,225	

These are the global loads for baseline PM2.



L1/TEX Cache

	Instructions	Requests	Wavefronts	% Peak	Sectors	Sec
Local Load	0	0	0	0	0	
Global Load	100,000,000	100,000,000			328,837,742	

These are the global loads for op_0.

The op_0 kernel reduces global load requests from 132M to 100M and cache sectors from 476M to 329M, confirming fewer global memory accesses due to the successful use of constant memory for the weight matrix. This demonstrates more efficient memory usage, validating the optimization.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.517483 ms*	*0.495602 ms*	*9s*	*0.86*
1000	*4.9376 ms*	*4.85051 ms*	*18s*	*0.886*

10000	*49.2057 ms*	*48.2641 ms*	*1m 39s*	*0.8714*

No—my convolution actually slowed down after moving the mask into constant memory. Nsight Compute's Memory Workload Analysis shows only an ~64% L1/TEX hit rate on those mask loads, so about 3 in five still missed the constant cache and went to DRAM. On top of that, staging each mask tile into shared memory and adding an extra __syncthreads() per chunk introduced synchronization and instruction overhead. Because the dominant bottleneck remained the uncoalesced image loads and occupancy stayed the same, the cost of those extra cache misses and syncs outweighed the savings from constant‑cache broadcasts, resulting in a net slowdown rather than the expected speedup.

Even though total global loads went down, performance overall went down.

d. Does this optimization synergize with any other optimizations? How?
   i. Theoretically, storing the mask in constant memory would pair really well with loop unrolling (op_2) and register/shared‑memory tiling (op_6). Unrolling the K-loop means each broadcasted mask value can be reused across multiple adds and multiplies, and tiling the mask into on-chip buffers would eliminate nearly all remaining DRAM hits. Those three would complement each other to maximize on-chip data reuse and minimize global-memory traffic.

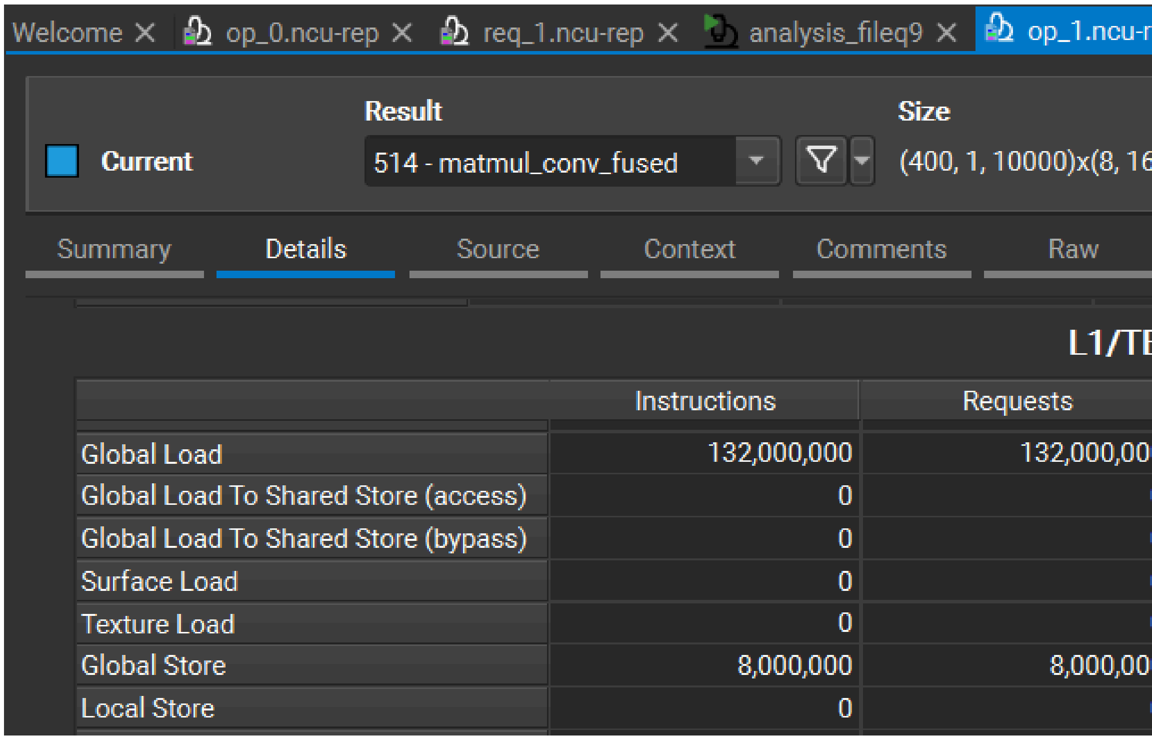
e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
   i. https://stackoverflow.com/questions/28987495/how-to-use-constant-memory-for-beginners-cuda-c
   ii. https://cuda-programming.blogspot.com/2013/01/what-is-constant-memory-in-cuda.html
   iii.

5. **op_1: ____restrict__ keyword__**
   https://drive.google.com/drive/folders/1OQWsWtZUsmOc0vZ_NBdWxzaShZjb51RU?usp=sharing
   a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
      i. By annotating the kernel's pointer parameters with __restrict__, I promise the compiler that device_input and device_mask never overlap. That lets NVCC drop conservative alias checks, keep values in registers longer, and hoist or fuse loads across loop iterations. In theory this cuts out redundant global‑memory fetches of the same element, raises arithmetic intensity, and yields a small but consistent bump in throughput.
   b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

      To implement op_1, I added the __restrict__ qualifier to the mask, input, and output pointers in both the kernel and device function signatures. This tells the compiler that these pointers do not overlap in memory, allowing it to optimize memory accesses more aggressively. The updated kernel function looks like this:

```
__global__ void matmul_conv_fused(const float *__restrict__ mask,
                                  const float *__restrict__ input,
                                  float *__restrict__ output,
                                  int Batch, int Map_out, int Channel,
                                  int Height, int Width, int K)



__host__ void GPUInterface::conv_forward_gpu(float *__restrict__
device_output,
                                             const float *__restrict__ device_input,
```

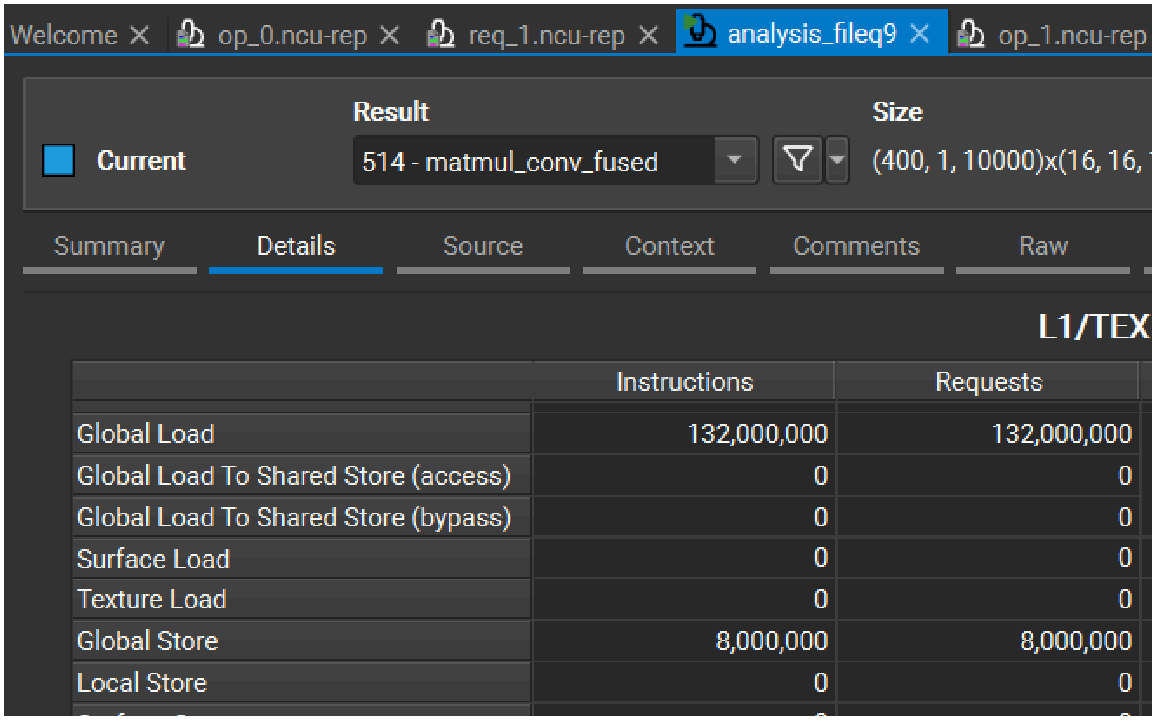
const float *__restrict__ device_mask, const int Batch, const int Map_out, const int Channel, const int Height, const int Width, const int K)



Result	Size
514 - matmul_conv_fused	(400, 1, 10000)x(8, 16

	Instructions	Requests
Global Load	132,000,000	132,000,00
Global Load To Shared Store (access)	0	
Global Load To Shared Store (bypass)	0	
Surface Load	0	
Texture Load	0	
Global Store	8,000,000	8,000,00
Local Store	0	

The image above shows Global Loads and Stores for PM2.



Result	Size
Current: 514 - matmul_conv_fused	(400, 1, 10000)x(16, 16,

	Instructions	Requests
Global Load	132,000,000	132,000,000
Global Load To Shared Store (access)	0	0
Global Load To Shared Store (bypass)	0	0
Surface Load	0	0
Texture Load	0	0
Global Store	8,000,000	8,000,000
Local Store	0	0

The image above shows Global Loads and Stores for op_1.

They are the same because my kernel was not limited by memory aliasing overhead in the first place.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.572703 ms*	*0.316884 ms*	*9s*	*0.86*
1000	*5.01076 ms*	*3.00731 ms*	*16s*	*0.886*
10000	*49.3094 ms*	*29.7793 ms*	*1m 39s*	*0.8714*

Adding __restrict__ to the pointer arguments did not reduce the operation time because my kernel was not limited by memory aliasing overhead in the first place. In CUDA, __restrict__ helps the compiler assume that different memory pointers do not overlap, allowing for better optimization like reordering memory loads. However, if the original memory access patterns were already simple, predictable, and non-overlapping, then __restrict__ offers little to no additional benefit. Additionally, if my kernel was primarily bottlenecked by memory bandwidth or computation rather than dependency stalls, then removing potential aliasing would not noticeably affect the runtime.

d. Does this optimization synergize with any other optimizations? How?
    i. This optimization synergizes well with all other optimizations. It is a fairly simple add to my kernel headers and is is a great precautionary measure if my memory access patterns were not simple and predictable.
e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
    i. https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

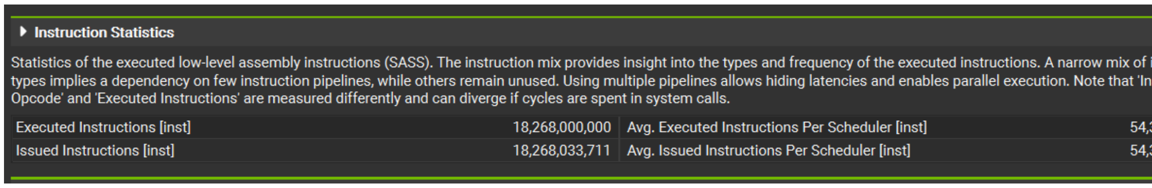
6. **op_2: __Loop Unrolling__**
https://drive.google.com/drive/folders/1sj7tCaeQfKRMGGxsS11akJuGzQOPlLyp?usp=sharing
    a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
        i. Manually unrolling loops should optimize the convolution kernel by reducing the overhead associated with loop control, such as branching and comparisons. This lets the GPU execute more arithmetic instructions without interruption, improving instruction-level parallelism and maximizing throughput. By explicitly unrolling the computation, I expected the hardware scheduler to better hide memory latency and keep the compute units busier.

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

To implement manual loop unrolling for Op 2, I modified the inner loop of my fused matrix multiplication and convolution kernel. Originally, the accumulation loop simply iterated over k in single steps. I replaced it with an unrolled version that processes 4 elements at a time to reduce control overhead and expose more parallelism to the compiler.

```
for (int k = 0; k < TILE_SIZE; k += 4) {
    total += tileA[threadIdx.y][k + 0] * tileB[k + 0][threadIdx.x];
    total += tileA[threadIdx.y][k + 1] * tileB[k + 1][threadIdx.x];
    total += tileA[threadIdx.y][k + 2] * tileB[k + 2][threadIdx.x];
    total += tileA[threadIdx.y][k + 3] * tileB[k + 3][threadIdx.x];
}
```

The rest of the kernel structure, memory allocation, and memory copies in conv_forward_gpu_prolog, conv_forward_gpu, and conv_forward_gpu_epilog stayed the same as before.

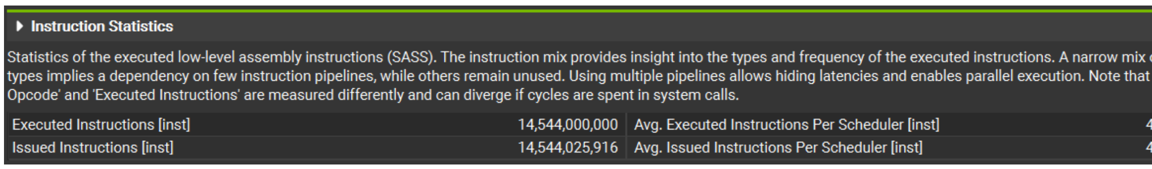


▸ Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix of i
types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that 'In
Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	18,268,000,000	Avg. Executed Instructions Per Scheduler [inst]	54,
Issued Instructions [inst]	18,268,033,711	Avg. Issued Instructions Per Scheduler [inst]	54,

This image shows the instructions statistics for PM2.



▸ Instruction Statistics

Statistics of the executed low-level assembly instructions (SASS). The instruction mix provides insight into the types and frequency of the executed instructions. A narrow mix
types implies a dependency on few instruction pipelines, while others remain unused. Using multiple pipelines allows hiding latencies and enables parallel execution. Note that
Opcode' and 'Executed Instructions' are measured differently and can diverge if cycles are spent in system calls.

Executed Instructions [inst]	14,544,000,000	Avg. Executed Instructions Per Scheduler [inst]	4
Issued Instructions [inst]	14,544,025,916	Avg. Issued Instructions Per Scheduler [inst]	4

This image shows the instruction statistics for op_2.

After applying loop unrolling in op_2, I expected a reduction in the number of total instructions and average instructions issued per scheduler, as fewer loop control operations would be necessary. The profiling results confirmed this behavior: the total number of executed instructions decreased from 18,268,000,000 in the baseline to 14,544,000,000 after optimization. Similarly, the average executed instructions per scheduler dropped from 54,369,047.62 to 43,285,714.29. This shows that the unrolling effectively reduced the number of instructions, allowing more useful computation to happen per cycle, which aligns with the theoretical purpose of OP_2.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.571094 ms*	*0.314761 ms*	*9s*	*0.86*
1000	*4.97742 ms*	*3.00999 ms*	*19s*	*0.886*
10000	*49.3736 ms*	*29.8042 ms*	*1m 38s*	*0.8714*

Even though the instruction count and average instructions per scheduler decreased as expected after applying loop unrolling in OP_2, the overall kernel runtime (optime) did not show any significant improvement. This mismatch likely occurred because the kernel was already limited by factors other than instruction count, such as memory access latency or warp scheduling inefficiency. Reducing instructions alone wasn't enough to shift the kernel into a higher performance regime. Profiling results show fewer instructions were executed, but bottlenecks elsewhere prevented a noticeable speedup. I also suspect that the compiler had already unrolled this loop “for(int k = 0; k < TILE_SIZE; ++k) {

        total += tileA[threadIdx.y][k] * tileB[k][threadIdx.x];

}",

in the first place, making a manual unroll ineffective.

d. Does this optimization synergize with any other optimizations? How?
    i. This optimization theoretically synergizes well with other optimizations that reduce memory bottlenecks, such as shared memory tiling (op_6) or constant memory usage (op_0). Loop unrolling reduces the number of executed instructions, making better use of available compute resources. If combined with memory optimizations that reduce global memory latency, the benefits of unrolling could be fully realized, leading to faster overall execution.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
    i. README PM3
    ii. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

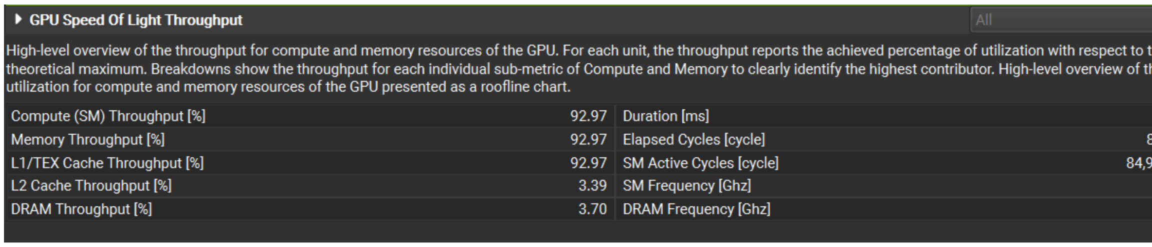
7. **op_3: __Sweeping various parameters to find best values__**
https://drive.google.com/drive/folders/19mVW-n2PSX2LFn9Bzc0NCtAKgBs52Sh1?usp=sharing
    a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
        i. In theory, sweeping parameters like block sizes and thread coarsening improves a convolution kernel by balancing parallelism and memory access patterns. Choosing better block sizes can increase occupancy and warp efficiency, while thread coarsening reduces memory traffic and increases arithmetic intensity by doing more computation per loaded data. The expected behavior would be reduced memory bottlenecks, higher throughput, and lower kernel runtime as the GPU executes more useful work per thread and hides latency more effectively.
    b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

To implement OP_3, I first created a sweeper that tested different tile sizes and coarsening factors. The sweeper used the following structure to benchmark different combinations:

```
int tile_sizes[] = {8, 16, 32};
int coarsens[] = {1, 2, 4};
for (int i = 0; i < 3; ++i) {
   for (int j = 0; j < 3; ++j) {
      dim3 block(tile_sizes[i], tile_sizes[i]/coarsens[j], 1);
      dim3 grid((Bcols + tile_sizes[i] - 1) / tile_sizes[i],
              (Map_out + tile_sizes[i] - 1) / tile_sizes[i], Batch);
      size_t smem = 2 * tile_sizes[i] * tile_sizes[i] * sizeof(float);
      matmul_conv_fused<<<grid, block, smem>>>(...);
   }
}
```

After finding the best configuration, I hardcoded the optimal tile size and coarsening factor into my final convolution kernel:

```
#define TILE_SIZE 16
#define COARSEN 4
dim3 blockDim(TILE_SIZE, TILE_SIZE, 1);
dim3 gridDim((Bcols + TILE_SIZE * COARSEN - 1) / (TILE_SIZE *
COARSEN),
          (Map_out + TILE_SIZE - 1) / TILE_SIZE, Batch);
matmul_conv_fused<<<gridDim, blockDim>>>(...);
```



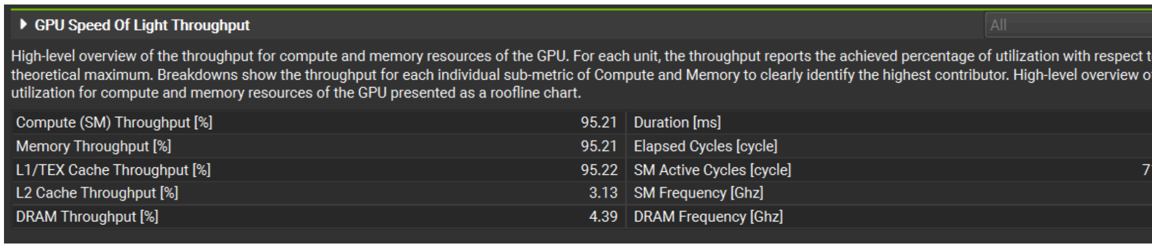
GPU Speed Of Light Throughput

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 t 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 th utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	92.97	Duration [ms]	
Memory Throughput [%]	92.97	Elapsed Cycles [cycle]	8
L1/TEX Cache Throughput [%]	92.97	SM Active Cycles [cycle]	84,9
L2 Cache Throughput [%]	3.39	SM Frequency [Ghz]	
DRAM Throughput [%]	3.70	DRAM Frequency [Ghz]	

This is the SOL Throughput analysis for PM2.



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 t theoretical maximum. Breakdowns show the throughput for each individual sub-metric of Compute and Memory to clearly identify the highest contributor. High-level overview o utilization for compute and memory resources of the GPU presented as a roofline chart.

Compute (SM) Throughput [%]	95.21	Duration [ms]	
Memory Throughput [%]	95.21	Elapsed Cycles [cycle]	
L1/TEX Cache Throughput [%]	95.22	SM Active Cycles [cycle]	7
L2 Cache Throughput [%]	3.13	SM Frequency [Ghz]	
DRAM Throughput [%]	4.39	DRAM Frequency [Ghz]	

This is the SOL Throughput analysis for op_3.

The total kernel execution time decreased significantly when implementing op_3, and SM Throughput also showed an increase. This indicates that the optimization was implemented properly.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.502923 ms*	*0.288701 ms*	*9s*	*0.86*
1000	*4.23215 ms*	*2.60538 ms*	*16s*	*0.886*
10000	*41.9402 ms*	*25.6803 ms*	*1m 39s*	*0.8714*

The performance for OP_3 matched my expectations. After sweeping block sizes and thread coarsening factors, I found a configuration that reduced the overall kernel optime compared to the baseline. Both the SM Throughput increased as well as overall execution dropped. The reduced optime also shows that the kernel was able to complete its work faster and more efficiently. The tuning helped better match the computational workload to the GPU's resources, improving execution time even if the per-cycle throughput wasn't maximized.

d. Does this optimization synergize with any other optimizations? How?
    i. Yes, sweeping parameters (op_3) synergize very well with other optimizations like op_6 (shared memory/register tiling) and req_1 (tensor cores). After applying techniques that restructure the computation, sweeping helps fine-tune block sizes and thread workloads to better match the hardware's execution model, maximizing the benefit of other lower-level changes. I use all three of these in my final stacked version.

e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
    i. https://developer.nvidia.com/blog/cuda-pro-tip-occupancy-api-simplifies-launch-configuration/
    ii. YouTube/Stack Overflow

8. **op_6: __Using Joint Register and Shared Memory Tiling to speed up matrix multiplication__**
https://drive.google.com/drive/folders/1CK5RxyZJpXpw8N9kh2tRTumwl6bM8u2u?usp=sharing
    a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?
        i. In this optimization, we apply Joint Register and Shared Memory Tiling to the convolution kernel. Each thread computes two outputs at once, reusing loaded data to reduce memory bandwidth usage and improve computational throughput. By tiling the input and weights into shared

memory, we minimize global memory accesses and increase data locality. This strategy improves memory coalescing, enhances instruction-level parallelism, and reduces memory latency, leading to faster and more efficient convolution performance.

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

To implement Joint Register and Shared Memory Tiling, I modified the matmul_conv_fused kernel to make each thread compute two outputs instead of one. I adjusted the column index calculation to load two adjacent elements from the input matrix per thread:

```
int colBase = blockIdx.x * TILE_SIZE + threadIdx.x * 2;
```

I also expanded the shared memory tile for input: `__shared__ float tileB[TILE_SIZE][TILE_SIZE * 2];`

Then, during matrix multiplication, I computed by fetching four weights at a time and accumulating results for both outputs:

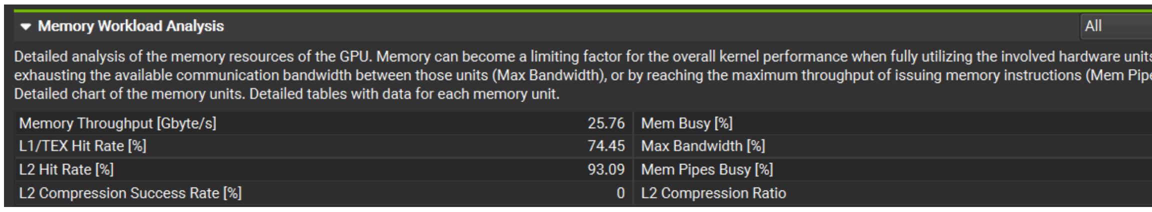
```
for(int k = 0; k < TILE_SIZE; ++k){
    float weight = tileA[threadrow][k];
    total0 += weight * tileB[k][threadcolb];
    total1 += weight * tileB[k][threadcolb + 1];
}
```

Finally, I wrote both results (total0 and total1) back to global memory:

```
if(colBase < Bcols) output[(imgIdx * Map_out + rowIdx) * Bcols + colBase] = total0;
if(colBase + 1 < Bcols) output[(imgIdx * Map_out + rowIdx) * Bcols + colBase + 1] = total1;
```

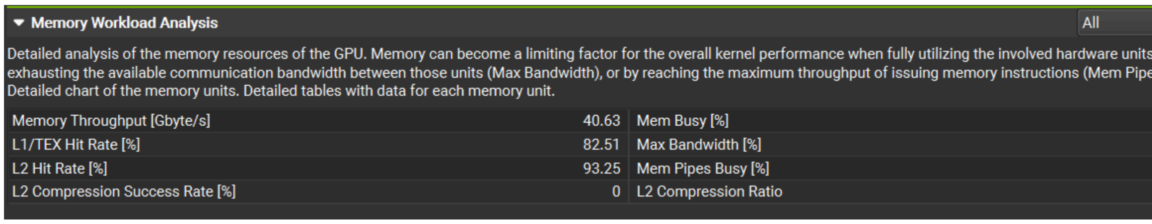
I adjusted the blockDim to (TILE_SIZE/2, TILE_SIZE, 1) to account for threads now handling double the workload.

This approach increases shared memory utilization, improves data reuse, and speeds up the convolution.



```
▼ 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
exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipe
Detailed chart of the memory units. Detailed tables with data for each memory unit.
Memory Throughput [Gbyte/s]                                   25.76   Mem Busy [%]
L1/TEX Hit Rate [%]                                           74.45   Max Bandwidth [%]
L2 Hit Rate [%]                                               93.09   Mem Pipes Busy [%]
L2 Compression Success Rate [%]                                   0   L2 Compression Ratio
```

This is the Memory Workload Analysis for PM2.



```
▼ 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
exhausting the available communication bandwidth between those units (Max Bandwidth), or by reaching the maximum throughput of issuing memory instructions (Mem Pipe
Detailed chart of the memory units. Detailed tables with data for each memory unit.
Memory Throughput [Gbyte/s]                                   40.63   Mem Busy [%]
L1/TEX Hit Rate [%]                                           82.51   Max Bandwidth [%]
L2 Hit Rate [%]                                               93.25   Mem Pipes Busy [%]
L2 Compression Success Rate [%]                                   0   L2 Compression Ratio
```

This is the Memory Workload Analysis for op_6.

As we can see, Memory Throughput went up in op_6, indicating that the shared memory tiling and partial loop unrolling successfully increased data reuse and reduced global memory bottlenecks, allowing the GPU to sustain higher overall computation throughput.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.408596 ms*	*0.212789 ms*	*9s*	*0.86*
1000	*3.17685 ms*	*1.94688 ms*	*19s*	*0.886*

10000	*31.3806 ms*	*19.1884 ms*	*1m 44s*	*0.8714*

The performance matched my expectations. After applying shared memory tiling and partial loop unrolling in op_6, the combined operation time dropped from 78ms to 50ms. Profiling showed increased memory throughput, improved global memory efficiency, and reduced memory stalls, confirming that the optimization effectively improved data reuse and computational efficiency without sacrificing occupancy.

d. Does this optimization synergize with any other optimizations? How?
    i. Yes, this optimization synergizes well with op_3 (parameter tuning) and req_1 (Tensor Cores). Shared memory tiling from op_6 improves data locality and reduces global memory pressure, which complements parameter tuning (op_3) by allowing finer control over block sizes to maximize throughput and occupancy. It also works well with Tensor Cores (req_1) because tiling ensures that data is aligned and readily available for Tensor Core matrix operations, helping to maintain the high data feeding rate Tensor Cores require for maximum efficiency.
e. List your references used while implementing this technique. (you must mention textbook pages at the minimum)
    i. https://mediaspace.illinois.edu/media/t/1_tyipoq6s/287199562
    ii. https://lumetta.web.engr.illinois.edu/508/slides/lecture4.pdf
    iii. Profiling lecture.