Name: <Chun-Yi Lee>
NetID: <chunyil3>
Section: <AL2>

ECE 408/CS483 Milestone 3 Report

List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. Note: **Do not** use batch size of 10k when you profile in --queue rai_amd64_exclusive. We have limited resources, so any tasks longer than 3 minutes will be killed. Your baseline M2 implementation should comfortably finish in 3 minutes with a batch size of 5k (About 1m35 seconds, with nv-nsight).
File path: Project/m3_file/m3_baseline.cu

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.263 ms	1.554 ms	0.201 s	0.86
1000	2.669 ms	15.746 ms	0.332 s	0.886
5000	13.266 ms	79.045 ms	0.864 s	0.871

Optimization 1: FP16

File path: Project/m3_file/m3_FP16_inKernel.cu

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

I chose **FP16** since it is an easy-to-implement technique, since we only need to convert the data type we used to half precision(16 bits) and use the corresponding function to do addition and multiplication. Also, using FP16 is expected to bring 2x faster computation on the kernel.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

Using FP16, we store a value using half the amount of bits compared to the baseline. It will enhance the memory utilization and speed of computation, so it should enhance the performance on forward convolution by 2x.

No, I use baseline code and convert data type into __half and use __half related function on calculation.

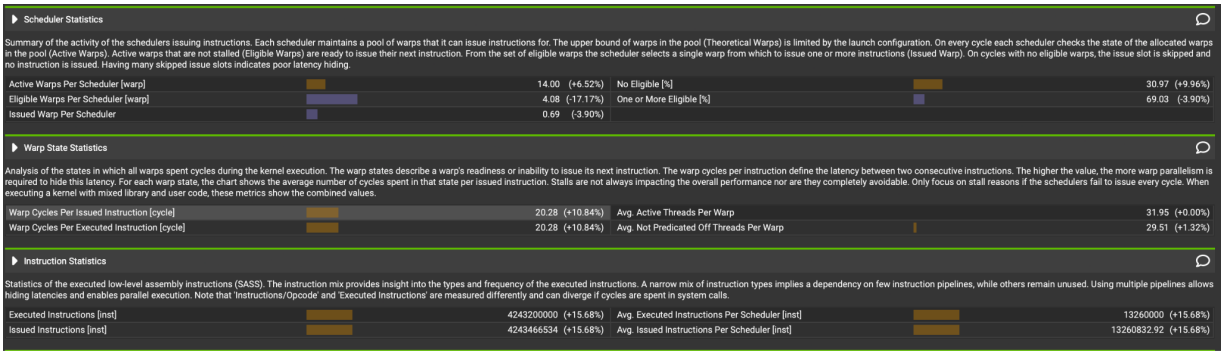
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.327 ms	1.997 ms	0.181 s	0.86
1000	3.219 ms	19.881 ms	0.324 s	0.887
5000	15.984 ms	99.217 ms	1.028 s	0.8712

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from nsys and Nsight-Compute to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

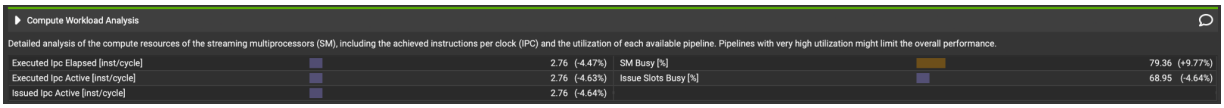
No, it does not improve performance. Based on the information in the NVIDIA Nsight Compute, I think the reason is the overhead of type conversion in the kernel.

First, FP16 spent 16.04 ms on kernel 5. However, the baseline spent only 13.24 ms.



In NVIDIA Nsight Compute, there is an increase in warp cycles per issued instruction. It indicates that for each instruction, it takes more time to be issued, possibly due to increased waiting time on type

conversion(stall). Also, we can see we execute more instructions, at the same code structure, the program spent more time to finish all the job.



There we can see although the memory usage is lower, the SM busy rate is higher than baseline. Since they have the same code structure except for the pValue mul-add line, the overhead of type conversion may be the reason why SM is busier than before.

Another factor is that a large portion of the kernel execution time is dealing with global memory access. Therefore, this may make the improvement on the computing aspect seem small, as the memory access time is much longer than an addition and a multiplication.

Note that FP16 makes the accuracy slightly changed due to a loss of precision.

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

1. Campuswire
2. https://ion-thruster.medium.com/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f
3. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html

Optimization 2: Tiled Share Memory

File path: Project/m3_file/m3_tiledSharedMemory.cu

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

I chose tiled share memory since I was already familiar with this method since we spent a lot of time in it for the half semester. Also, the technique is supposed to have better performance in high probability. (Others such as FP16 may not improve the performance due to overhead)

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

It works because it reduces the time on accessing data on the kernel during calculation by putting data onto shared memory. Assume we want to compute 2 adjacent outputs, if the mask size is 3 and stride is 1, then we would have about 6 input elements overlapped for the two masks. In the baseline version, we access

global memory 18 times, but only 12 in the tiled version. Since the access time of global memory is way more larger than that of shared memory, tiling makes the kernel run faster.

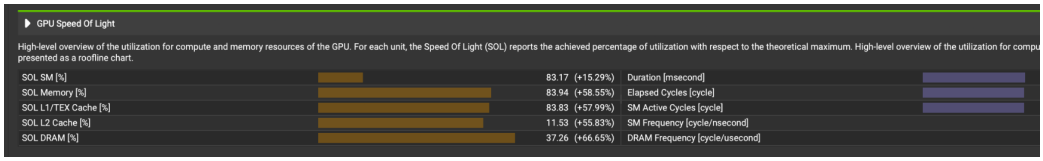
I use baseline code and convert it to the tiled version, so it does not synergize with previous optimizations.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

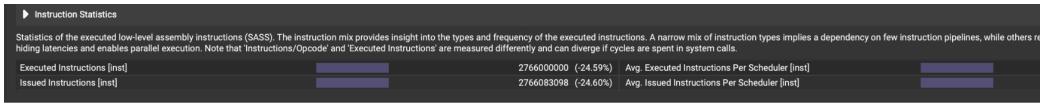
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.165 ms	0.784 ms	0.168 s	0.86
1000	1.592 ms	7.852 ms	0.357 s	0.886
5000	7.878 ms	43.697 ms	1.088 s	0.871

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from nsys and Nsight-Compute to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

It successfully improves the performance. The reason is stated in part b.



In the GPU SOL part, we can see utilization rate of the resources in GPU are way more better than the baseline. Especially on memory, cache, and SM. Therefore, it is reasonable that the execution time of the tiled version is lower.



In the Instruction Statistics part, there are fewer instructions executed in the tiled version. It indicates the tiled version is more effective. Each data can be used multiple times in shared memory, thus reducing the number of memory instructions.

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

MP3, MP4

Optimization 3: Restrict + Loop Unrolling

File path: Project/m3_file/m3_loopUnrolling_restrict.cu

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

I use it since it is easy to implement. Also, for most optimization method, we still need to go through the loop of calculating “input * mask”. Therefore, the optimization can combine with others so we can achieve a better result.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

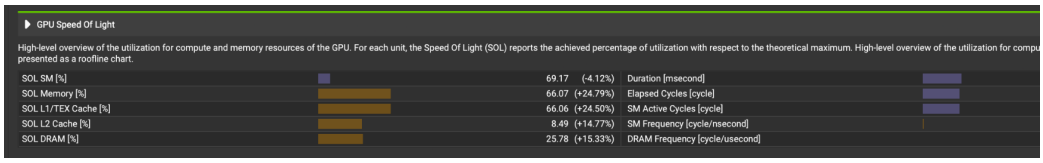
I think the optimization will increase performance of the forward convolution. Loop unrolling means we replicate code in the loop and calculate related variables in advance, then we can get rid of the loop. In that case, we save the loop overhead, including condition checking and increment.

In this optimization, I use baseline and unroll the kernel only for analyzing purposes. Therefore, it does not synergize with previous optimizations.

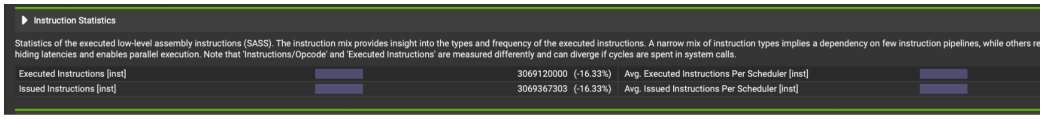
c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.243 ms	1.500 ms	0.177 s	0.86
1000	2.336 ms	14.956 ms	0.332 s	0.886
5000	11.572 ms	74.634 ms	0.912 s	0.871

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from nsys and Nsight-Compute to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).



Through GPU SOL, we have more memory, L1/L2 cache, and DRAM utilization. By unrolling the loop, in the instruction level, the processor may be able to do the computation in parallel. Also, since we are calculating data near each other. Therefore, we may have data locality by loop unrolling, increasing the cache and memory utilization.



Instruction Statistics show a reduction of “number of instructions executed” by 16.33%. It means that the loop unrolling kernel is more efficient than the baseline kernel since the former completes the same job as the latter, but the former uses less instruction. The reduction may be due to the elimination of nested for loop time by 4 times(k1+=2, k2+=2), so the overhead of loop checking, loop increment, and non-coalesced memory access in baseline code will be reduced.

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

https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf

Optimization 4:
File path: Project/m3_file/m3_stream.cu

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

I chose stream since it can parallelize the memory copy and kernel computation.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

Theoretically, the more streams are created, the more parallelized the system would be. Since our program is memory bound, it can be possible that all the kernel launches can overlap with cudaMemcpy. Therefore, by using stream optimization, the kernel execution time can be ignored in this program.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Layer Time 1	Layer Time 2	Total Execution Time	Accuracy
100	9.296 ms	7.252 ms	0.124 s	0.86
1000	98.457 ms	79.617 ms	0.232 s	0.886
5000	466.228 ms	428.168 ms	0.844 s	0.871

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from nsys and Nsight-Compute to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

   No, the optimization does not improve the performance. The layer time of 2 convolution is longer than baseline (~300ms for each layer). Since we cannot measure the op time. Therefore, let's analyze the other aspects.

CUDA API Statistics (nanoseconds)

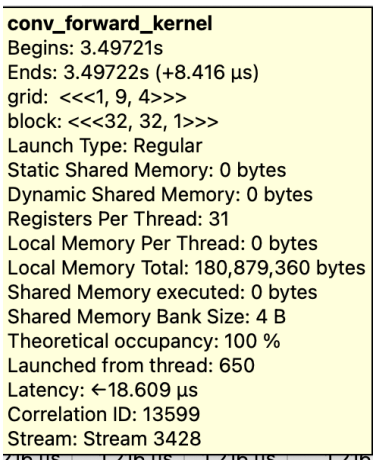
Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
65.8	864139955	20026	43150.9	4753	525637	cudaMemcpyAsync
21.7	285328688	10010	28504.4	1198	179711833	cudaStreamCreate
10.2	133677716	10014	13349.1	3408	43018402	cudaLaunchKernel

CUDA API Statistics (nanoseconds)

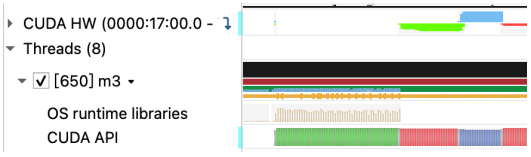
Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
63.9	551845732	20	27592286.6	33979	288851721	cudaMemcpy
23.1	199674290	20	9983714.5	2595	195406333	cudaMalloc
10.7	92691346	16	5793209.1	873	79197655	cudaDeviceSynchronize

As the above cuda api Statistics, the left side is Stream, and the right side is Baseline.

We can see increasing cuda api overhead in Stream. We spent about 61% more time on cudaMemcpy(Async), compared to the baseline.



Second, in Nsight Systems, streams use a really small grid size to launch the kernel. (given number of stream is B) Therefore, the overhead on cudaAPI calls may exceed the benefit of streams executing in parallel. We would have about 5000 streams if the batch size = 5000. Since there are limited resources (memory copy, kernel computing…) in the whole system, the streams may compete for resources with each other. The code ends up performing like a serialized version of code with excessive overhead (stream creation, stream deletion, cuda api call).



Another evidence is that, in CUDA HW in Nsight Systems, we see the kernel time(blue) has only a little overlap with cudaMemcpy time(green and red).

Ideally, we want green and red parts to stick together, so that the kernel time can fully parallelize with memory copy time. However, since there are too many streams executing on the system, we cannot see the parallel effect on this optimization. That indicates if we use a smaller number of streams, it can reduce the overhead and reach a better parallel level.

e. What references did you use when implementing this technique?
    1. Campuswire
    2. https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async.cu

f. stream code

```
#include <cmath>

#include <iostream>

#include "gpu-new-forward.h"

#include <iostream>

using namespace std;
```

```
#define TILE_WIDTH 32


__global__ void conv_forward_kernel(float *output, const float *input, const float *mask, const int B, const int M, const int C, const int H, const int W, const int K,const int S)

{

    const int H_out = (H - K)/S + 1;

    const int W_out = (W - K)/S + 1;


    #define out_4d(i3, i2, i1, i0) output[(i3) * (M * H_out * W_out) + (i2) * (H_out * W_out) + (i1) * (W_out) + i0]

    #define in_4d(i3, i2, i1, i0) input[(i3) * (C * H * W) + (i2) * (H * W) + (i1) * (W) + i0]

    #define mask_4d(i3, i2, i1, i0) mask[(i3) * (C * K * K) + (i2) * (K * K) + (i1) * (K) + i0]


    const int W_Grid = ceil(float(W_out) / TILE_WIDTH); // IMPORTANT: for grid, we see an output element as a block unit.

    int b = blockIdx.x;

    int m = blockIdx.z;

    int h = (blockIdx.y / W_Grid) * TILE_WIDTH + threadIdx.y;

    int w = (blockIdx.y % W_Grid) * TILE_WIDTH + threadIdx.x;


    // Insert your GPU convolution kernel code here

    float pValue = 0.0;

    for (int c=0; c<C; c++) {

        for (int kr=0; kr<K; kr++) {

            for (int kc=0; kc<K; kc++) {

                pValue += in_4d(b, c, h*S+kr, w*S+kc) * mask_4d(m, c, kr, kc);

            }

        }
```

```
    }

    if (h < H_out && w < W_out) {
        out_4d(b, m, h, w) = pValue;
    }

    #undef out_4d
    #undef in_4d
    #undef mask_4d
}

__host__ void GPUInterface::conv_forward_gpu_prolog(const float *host_output, const float
*host_input, const float *host_mask, float **device_output_ptr, float **device_input_ptr, float
**device_mask_ptr, const int B, const int M, const int C, const int H, const int W, const int K, const int
S)
{
    // Allocate memory and copy over the relevant data structures to the GPU
    const int H_out = (H - K)/S + 1;
    const int W_out = (W - K)/S + 1;

    int numOfStream = B;

    cudaStream_t stream[numOfStream];

    for (int i=0; i<numOfStream; i++)
        cudaStreamCreate(&stream[i]);
```

```
    float* host_out_cast = (float*) host_output;

    int stream_outSize = H_out * W_out * M * B / numOfStream;
    int stream_inSize = H * W * C * B / numOfStream;
    int remainingInputFloats = (H * W * C * B) % numOfStream;
    int remainingOutputFloats = (H_out * W_out * M * B) % numOfStream;

    cudaMalloc((void**) device_output_ptr, H_out * W_out * M * B * sizeof(float));
    cudaMalloc((void**) device_input_ptr, H * W * C * B * sizeof(float));
    cudaMalloc((void**) device_mask_ptr, M * C * K * K * sizeof(float));

    const int H_Grid = ceil(H_out / float(TILE_WIDTH));
    const int W_Grid = ceil(W_out / float(TILE_WIDTH));
    const int G = H_Grid*W_Grid;

    dim3 DimBlock(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 DimGrid(B/numOfStream, G, M); // batch_size, GridSize, # of mask
    cudaMemcpyAsync(*device_mask_ptr, host_mask, M * C * K * K * sizeof(float),
cudaMemcpyHostToDevice, stream[0]);

    for (int i=0; i<numOfStream; i++) {
        int inputOffset = i * stream_inSize;
        int inputSize = (i == numOfStream - 1) ? (stream_inSize + remainingInputFloats) : stream_inSize;
        // int inputSize = stream_inSize;
        cudaMemcpyAsync((*device_input_ptr) + inputOffset, host_input + inputOffset, inputSize *
sizeof(float), cudaMemcpyHostToDevice, stream[i]);
    }
    for (int i=0; i<numOfStream; i++) {
```

```
        conv_forward_kernel<<<DimGrid,DimBlock,0,stream[i]>>>((*device_output_ptr) + i *
stream_outSize, (*device_input_ptr) + i * stream_inSize, *device_mask_ptr, B, M, C, H, W, K, S);
    }
    for (int i=0; i<numOfStream; i++) {
        int outputOffset = i * stream_outSize;
        int outputSize = (i == numOfStream - 1) ? (stream_outSize + remainingOutputFloats) :
stream_outSize;
        // int outputSize = stream_outSize;
        // Copy the output back to host
        cudaMemcpyAsync(host_out_cast + outputOffset, (*device_output_ptr)+ outputOffset,
outputSize * sizeof(float), cudaMemcpyDeviceToHost, stream[i]);
    }
    cudaDeviceSynchronize(); // not sure if I need this.


    for (int i=0; i<numOfStream; i++)
        cudaStreamDestroy(stream[i]);


    // Free device memory
    cudaFree(device_output_ptr);
    cudaFree(device_input_ptr);
    cudaFree(device_mask_ptr);
}


__host__ void GPUInterface::conv_forward_gpu(float *device_output, const float *device_input,
const float *device_mask, const int B, const int M, const int C, const int H, const int W, const int K,
const int S)
{
    return;
}
```

```
__host__ void GPUInterface::conv_forward_gpu_epilog(float *host_output, float *device_output, float *device_input, float *device_mask, const int B, const int M, const int C, const int H, const int W, const int K, const int S)
{
    return;
}



__host__ void GPUInterface::get_device_properties()
{
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);

    for(int dev = 0; dev < deviceCount; dev++)
    {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, dev);

        std::cout<<"Device "<<dev<<" name: "<<deviceProp.name<<std::endl;
        std::cout<<"Computational capabilities: "<<deviceProp.major<<"."<<deviceProp.minor<<std::endl;
        std::cout<<"Max Global memory size: "<<deviceProp.totalGlobalMem<<std::endl;
        std::cout<<"Max Constant memory size: "<<deviceProp.totalConstMem<<std::endl;
        std::cout<<"Max Shared memory size per block: "<<deviceProp.sharedMemPerBlock<<std::endl;
        std::cout<<"Max threads per block: "<<deviceProp.maxThreadsPerBlock<<std::endl;
        std::cout<<"Max block dimensions: "<<deviceProp.maxThreadsDim[0]<<" x, "<<deviceProp.maxThreadsDim[1]<<" y, "<<deviceProp.maxThreadsDim[2]<<" z"<<std::endl;
        std::cout<<"Max grid dimensions: "<<deviceProp.maxGridSize[0]<<" x, "<<deviceProp.maxGridSize[1]<<" y, "<<deviceProp.maxGridSize[2]<<" z"<<std::endl;
```

```
        std::cout<<"Warp Size: "<<deviceProp.warpSize<<std::endl;
    }
}
```

Optimization 5: TiledShareMemory + Constant Memory on Mask + Loop Unrolling & Restrict
File path: Project/m3_file/m3_tiled_loopUnrolling_restrict_constantMask.cu
**This is the code I submitted as m3 optimization! (i.e. this code = Project/custom/new-forward.cu)

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

   Based on the previous result of the tiled version. I could have other compatible optimizations on it. Therefore, I let the mask be in the constant memory, avoiding the data transfer time on the kernel if we use shared memory to store the mask. Also, I unrolled the loop on mask multiplication (K by K), it could further reduce the overhead of the nested loop and improve performance.

b. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?

   Loop Unrolling & Restrict: Stated in optimization 3

   Constant Mask: It reduces the global memory access time in the kernel by copying the mask into constant memory before launching the kernel. It will reduce the memory copy time on the mask, thus improving the kernel performance.

   This optimization synergizes tiled shared memory, loop unrolling & restrict, and constant memory on mask.

c. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.132 ms	0.572 ms	0.189 s	0.86
1000	1.339 ms	5.899 ms	0.316 s	0.886
5000	**6.647 ms**	**29.619 ms**	**0.884 s**	**0.871**

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.243 ms	1.500 ms	0.177 s	0.86
1000	2.336 ms	14.956 ms	0.332 s	0.886
5000	11.572 ms	74.634 ms	0.912 s	0.871

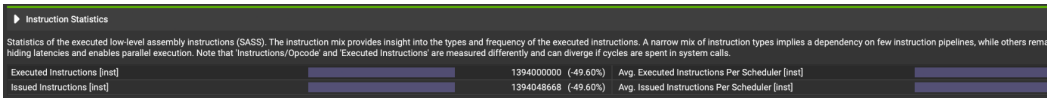
loop unrolling & restrict

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.165 ms	0.784 ms	0.168 s	0.86
1000	1.592 ms	7.852 ms	0.357 s	0.886
5000	7.878 ms	43.697 ms	1.088 s	0.871

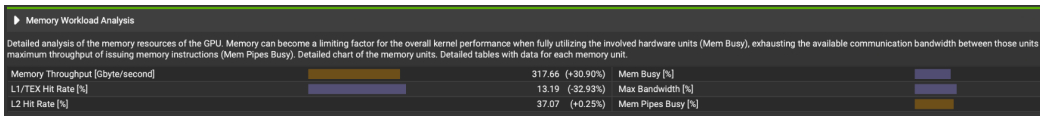
tiled shared memory

d. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from nsys and Nsight-Compute to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).

The total op time is below 40 ms. Compared to the other 2, this optimization has better performance. Next, I'll analyze in what aspect it is more efficient.



The amount of instruction is half compared to the merely tiled version. It means the kernel finishes the same job with only half the number of instructions. By using loop unrolling and constant mask, we reduce the overhead of global memory accessing and mask shared memory copy.



The Memory throughput is about 31% higher, meaning the kernel of this optimization is more efficient in using memory. This is probably because we use loop unrolling and constant memory, so we get memory coalescing.

CUDA API Statistics (nanoseconds)

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
71.0	537236318	14	38374022.7	26834	292569012	cudaMemcpy
22.6	170936081	14	12209720.1	3853	167553625	cudaMalloc
4.8	36106970	16	2256685.6	829	29415280	cudaDeviceSynchronize

tiled share memory + loop unrolling + mask

CUDA API Statistics (nanoseconds)

Time(%)	Total Time	Calls	Average	Minimum	Maximum	Name
70.6	567254596	20	28362729.8	30130	308679922	cudaMemcpy
22.5	181118122	20	9055906.1	2189	177783893	cudaMalloc
6.6	52699297	16	3293706.1	841	43894509	cudaDeviceSynchronize

tiled shared memory only

In the statistics, we can see the cudaMemcpy execution time of this optimization is slightly lower than merely using the tiled share memory due to constant memory.

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

optimization 2,3.