Qingyuan Liu

Milestone 3 Report

[https://drive.google.com/drive/u/1/folders/1gfFlVh5kqfgnh6ROY53tlT-XuSdhPTzm]

0. **Baseline:**

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.413585 ms*	*0.323176 ms*	*1.5346 s*	*0.86*
1000	*3.43879 ms*	*3.07593 ms*	*9.7743 s*	*0.886*
10000	*33.4551ms*	*30.5477 ms*	*1m28.996s*	*0.8714*

1. **Req_0: Using Streams to overlap computation with data transfer**
   [https://drive.google.com/drive/u/1/folders/1b-XCiQDM0o-zgI9pWDWrX-YEyIV6o1OG]
   a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

The streams optimization should overlap the data transfer when calling three different kernels. The operations in different streams can be interleaved and they can run concurrently so that our total execution time will decrease. Originally, three kernels are called sequentially; however, since we have streams now, we expect that the three kernels can do the parallel works.

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

For streams, the modifications are majorly on the conv_forward_gpu_prolog kernel. The first thing we need to define is the stream number and the list for stream to store the values – show in the Figure 1. We will then allocate the device memory by doing cudaMemcpyAsync as shown in the reference website to make sure that during the stream processing, the data will not have conflicts – show in the Figure 2. After that, we will call three kernels as usual, but we will reshape the input values for each kernel calls by setting a offset for each one(depending on how

many streams we want to implement). The final part is to copy the device values to host output and according to the reference page, we need to do the cudaStreamSynchronize and cudaStreamDestroy so that we can block the host until all preceding commands in the stream have completed. To check the correctness, the accuracy for batch size equals to 10000 is 0.8714, which is exactly the same value as the unmodified unroll version. In addition to that, we see a dramatic decrease in the Op time comparing with the original one, which means our implementation is correct. For the further analysis on streams, I will put that in the section C including the profiling results.

```
//Stream creatation
int steam_nums = 4;
cudaStream_t streams[steam_nums];
for (int i = 0; i < steam_nums; i++) {
    cudaStreamCreate(&streams[i]);
}
```

Fig 1

```
// Allocate device memory for each stream correspondign to the input size
for (int i = 0; i < steam_nums; i++) {
    size_t offset = i * Batch * Channel * Height * Width / steam_nums;
    cudaMemcpyAsync(*device_input_ptr + offset, host_input + offset, Batch * Channel * Height * Width / steam_nums * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
}
```

Fig 2

```
// Call the unrolling kernel for each stream
for (int i = 0; i < steam_nums; i++) {
    size_t input_offset = (size_t)i * Batch * Channel * Height * Width / steam_nums;
    size_t unroll_offset = (size_t)i * Height_unrolled * Width_unrolled / steam_nums;
    dim3 block_unroll(TILE_WIDTH, TILE_WIDTH, 1);
    dim3 grid_unroll((Height_out + TILE_WIDTH - 1)/TILE_WIDTH * (Width_out + TILE_WIDTH - 1)/TILE_WIDTH, Batch / steam_nums, 1);
    matrix_unrolling_kernel<<<grid_unroll, block_unroll, 0, streams[i]>>>(*device_input_ptr + input_offset, unrolled_matrix + unroll_offset, Batch / steam_nums, Channel, H
}

// Call the matrix multiplication kernel
// Multiply the mask with the unrolled matrix
for (int i = 0; i < steam_nums; i++) {

    int Height_unrolled = Channel * K * K;
    int Width_unrolled = Batch / steam_nums * Height_out * Width_out ;

    size_t unroll_offset = (size_t) i * Batch * Channel * K * K * Height_out * Width_out / steam_nums;
    size_t matmul_offset = (size_t) i * Batch * Map_out * Height_out * Width_out / steam_nums;

    dim3 matmul_grid_dim((Width_unrolled + TILE_WIDTH - 1)/ TILE_WIDTH, (Map_out + TILE_WIDTH - 1) / TILE_WIDTH, 1);
    dim3 matmul_block_dim(TILE_WIDTH, TILE_WIDTH, 1);
    matrixMultiplyShared<<<matmul_grid_dim, matmul_block_dim, 0, streams[i]>>>(*device_mask_ptr, unrolled_matrix + unroll_offset, matmul_output + matmul_offset, Map_out, H
}


// Call the permutation kernel for each stream
for (int i = 0; i < steam_nums; i++) {
    size_t permute_offset = i * (Batch * Map_out * Height_out * Width_out / steam_nums);
    size_t image_size = Height_out * Width_out;
    dim3 permute_kernel_grid_dim((image_size - 1) / PERMUTE_BLOCK_SIZE + 1, Batch / steam_nums, 1);
    matrix_permute_kernel<<<permute_kernel_grid_dim, PERMUTE_BLOCK_SIZE, 0, streams[i]>>>(
        matmul_output + permute_offset, *device_output_ptr + permute_offset, Map_out, Batch / steam_nums, image_size
    );
}
```

Fig 3

```
// Copy the output back to host
for (int i = 0; i < steam_nums; i++) {
    size_t offset_output = i * (Batch * Map_out * Height_out * Width_out / steam_nums);
    cudaMemcpyAsync((void*)(host_output + offset_output), *device_output_ptr + offset_output, Batch * Map_out * Height_out * Width_out / steam_nums * sizeof(float), cudaMe
}


// Stream synchronization
for (int i = 0; i < steam_nums; i++) {
    cudaStreamSynchronize(streams[i]);
    cudaStreamDestroy(streams[i]);
}
```

Fig 4

```
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Layer Time: 365.556 ms
Op Time: 0.004929 ms
Conv-GPU==
Layer Time: 333.04 ms
Op Time: 0.004459 ms

Test Accuracy: 0.8714

```

Fig 5

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	*0.002154 ms*	*0.002014 ms*	*1.424*	*0.86*
1000	*0.003025 ms*	*0.003106 ms*	*9.675*	*0.886*
10000	*0.002835 ms*	*0.003046 ms*	*1m32.261s*	*0.8714*

I think the performance matches my expectations. First, from the total duration time shown in the figure 6, all three kernels – unrolling, matrix multiplication and permutation – decreases approximately 4 times compared to the original design, which is due to the fact that we picked stream numbers equal to 4. This can parallelize the data execution and accelerate the data transfer. In addition to that, from the cache and memory perspective, we see that the total transfer

data between device and the caches decreases to 3.27 GB from 13 GB, which dramatically decrease the total time of global transferring time that can be the source of stalling time for the system. Finally, the accuracy maintains the same and the Op time and execution time decreases, meaning that our optimization does its job.

Fig 6

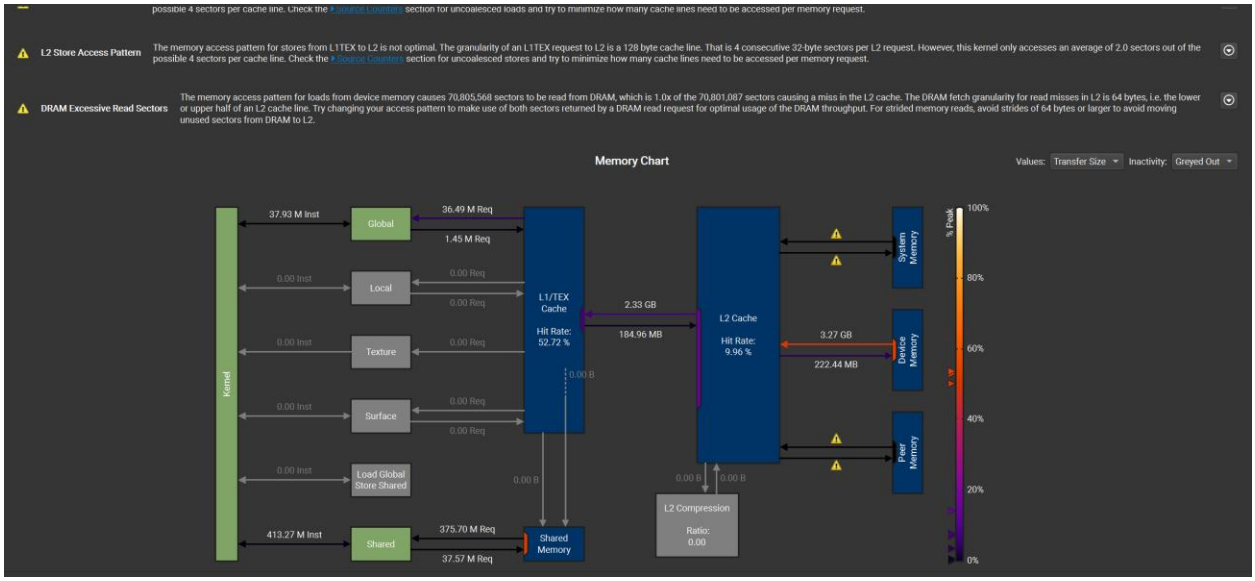

Fig 7

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

For the streams specifically, it can't synergize with other optimizations since we called three different kernels while other optimizations are based on the fusion kernel – combining all three kernels together. Streams should be designed specifically to reduce the overhead between different kernel calls

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

1. Harris, M. (2012, December 13). How to overlap data transfers in CUDA C/C++. NVIDIA Technical Blog. https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/

2. Rennich, S. (2011). CUDA C/C++ Streams and Concurrency [Webinar slides]. NVIDIA. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf

3. Harris, M. (2015, January 22). GPU Pro Tip: CUDA 7 Streams Simplify Concurrency. NVIDIA Technical Blog. https://developer.nvidia.com/blog/gpu-pro-tip-cuda-7-streams-simplify-concurrency/

2. **Req_1: Using Tensor Cores to speed up matrix multiplication**
[https://drive.google.com/drive/u/1/folders/1SjhN-PQrz4oNVINxMnqYide_2tW4h-x1]

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

The Tensor Cores TF32 can help to improve the arithmetic granularity by using 16 × 16 × K fused MMA tile per cycle (warp-level) and hence we can have more FLOPs issued for each instruction. We expect to see the faster duration time of operation for he fusion kernels and better memory access time or hit rates for the cache.

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

For Tensor Core TF32, similar to Tensor Core FP16, we will first include “nvcuda” and define new block sizes for the tiles. In addition to that, new tile size should be defined as A: 32 * 8, B: 8*32 and C:32*32(In Fig 8) since we want to make fragments have 16 * 8 size(in Fig 9). After that, we will modify the loading processes from milestone 2 fusion kernels. We will need to modify the index as twice of the original value(2*idx) to traverse all the elements correctly in new fragments and also modify the Row and Col to redefine the coordinates based on the M, N

and K values we define for the tiles(Fig 10 and Fig 11). We will then load all the data to TileC and make it ready for the store process(Fig 12). Finally, we will output the results from TileC back to the output matrix with the modified coordinates.

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

#define BLOCK_SIZE 256
#define TILE_WIDTH 16
#define MATMUL_TILE_WIDTH 16

#define M 16
#define N 16
#define K 8

__global__ void matmul_conv_fused(const float *mask, const float *input, float *output,
                                  int Batch, int Map_out, int Channel, int Height, int Width, int K)
{
    __shared__ float tileA[2*M][K];
    __shared__ float tileB[K][2*N];
    __shared__ float tileC[2*M][2*N];
```

Fig 8

```
// Initialize the output fragment
wmma::fragment<wmma::matrix_a, 16, 16, 8, wmma::precision::tf32, wmma::row_major> a_frag;
wmma::fragment<wmma::matrix_b, 16, 16, 8, wmma::precision::tf32, wmma::row_major> b_frag;
wmma::fragment<wmma::accumulator, 16, 16, 8, float> c_frag;
wmma::fill_fragment(c_frag, 0.0f);
```

Fig 9

```
// load data into shared memory
for (int i = 0; i < (mask_unrolled_width + TF_K - 1) / TF_K; i++){

    //Load the mask into shared memory
    int col_m = i * TILE_WIDTH + tx;
    int idx = 2*tx;
    for (int j = 0; j < 2; j++){
        int Row = (idx + j) / TF_K;
        int Col = (idx + j) % TF_K;
        int col_m = TF_K * i + Col;
        int row_m = Row_G + Row;
        if ((row_m < mask_unrolled_height) && (col_m < mask_unrolled_width)){
            tileA[Row][Col] = wmma::__float_to_tf32(mask[row_m * mask_unrolled_width + col_m]);
        } else {
            tileA[Row][Col] = wmma::__float_to_tf32(0.0f);
        }
    }
    __syncthreads();
```

Fig 10

```
    // Load the input into shared memory
    int idxA = 2*tx;
    for (int j = 0; j < 2; j++){
        int Row = (idxA + j) / (2*N);
        int Col = (idxA + j) % (2*N);
        int col_i = Col_G + Col;
        int row_i = i*TF_K + Row;
        if ((row_i < input_unrolled_height) && (col_i < input_unrolled_width)) {
            int c = row_i / (K * K);

            int h = col_i / Width_out;
            int p = (row_i - c * K * K) / K;

            int w = col_i % Width_out;
            int q = (row_i - c * K * K) % K;
            tileB[Row][Col] = wmma::__float_to_tf32(in_4d(bz, c, h + p, w + q));
        }
        else {
            tileB[Row][Col] = wmma::__float_to_tf32(0.0f);
        }
    }
    __syncthreads();
```

Fig 11

```
    wmma::load_matrix_sync(a_frag, &tileA[M * Row_W][0], TF_K);
    wmma::load_matrix_sync(b_frag, &tileB[0][N * Col_W], 2 * N);
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
    __syncthreads();
}

wmma::store_matrix_sync(&tileC[M * Row_W][N * Col_W], c_frag, 2 * N, wmma::mem_row_major);
__syncthreads();
```

Fig 12

```
// store the result in the output matrix
int idx = 8*tx;
for (int j = 0; j < 8; j++){
    int Row = (idx + j) / (2*N);
    int Col = (idx + j) % (2*N);
    int col_o = Col_G + Col;
    int row_o = Row_G + Row;
    if ((row_o < mask_unrolled_height) && (col_o < input_unrolled_width)) {
        out_4d(bz, row_o, col_o/Width_out, col_o%Width_out) = tileC[Row][Col];
    }
}
```

Fig13

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	*0.4342 ms*	*0.2389 ms*	*3.314s*	*0.86*
1000	*3.2434 ms*	*2.3234 ms*	*9.372s*	*0.886*
10000	*33.3653 ms*	*21.5523 ms*	*1m31.231s*	*0.8714*

I think the performance matches my expectations. First, from the table above, we see that the Op time for batch size equal to 10000 has decreased from total 63 ms to about 55 ms – approximately 8 ms improvement while maintaining the accuracy of 0.8714. From the ncu report, we can see that the duration of the Tensor Core TF32 has about 44 ms while the original one is only 90 ms, which is a huge improvement(Fig 14). In addition to that, we also see the increase in the hit rate of both L1 and L2 caches due to the fact that the Tensor Core TF32 utilizes the resource better with the fragments than just using the tiling methods.

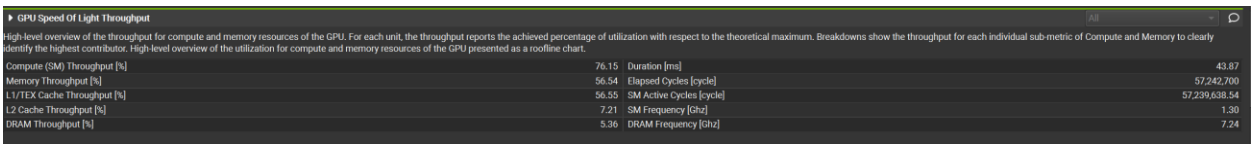

Fig 14

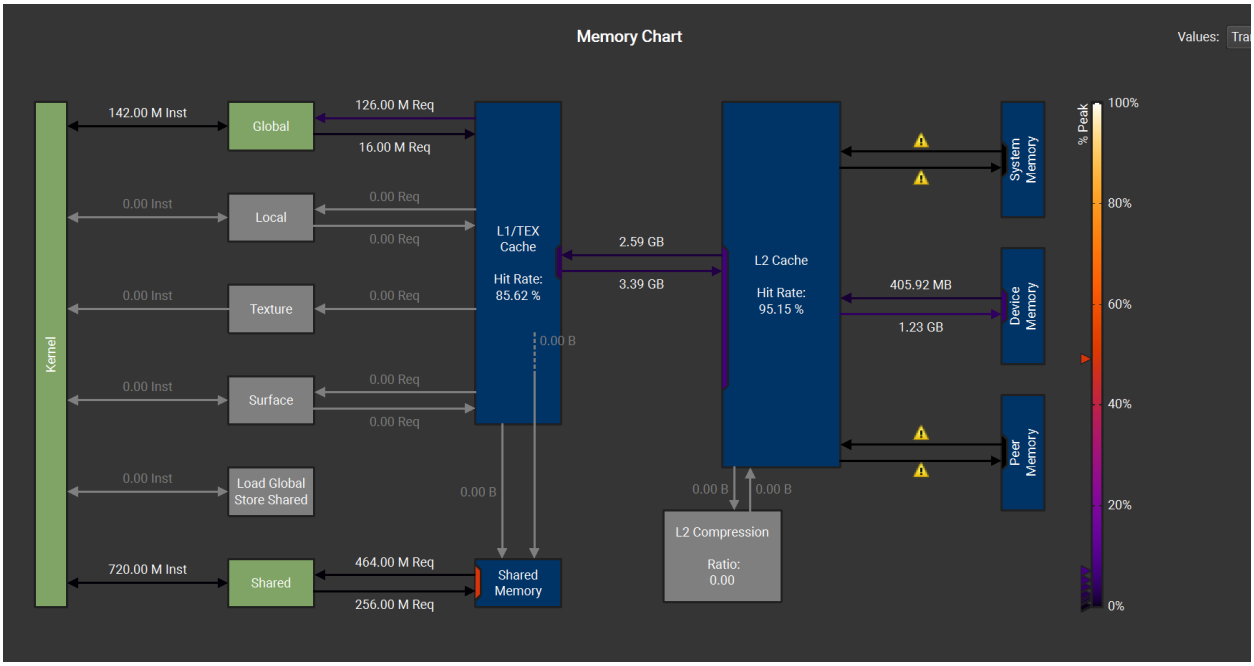

Fig 15

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

Tensor Cores TF32 can synergize with other optimization. For example, it is compatible with the share memory that reduces global memory latency and ensures that tiles of matrices are available with low latency for Tensor Core execution. In addition to that, the unrolling matrix can be a good suit for TF32 since it provides the GEMM (matrix-multiplication) form to boost the computation.

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

1. NVIDIA. (2017, October 12). *Programming Tensor Cores in CUDA 9*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/

2. NVIDIA. (2025). *CUDA C++ Programming Guide: Warp Matrix Functions*. NVIDIA Developer Documentation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-matrix-functions

3. **Op_1: ____restrict__ keyword__**

   [https://drive.google.com/drive/u/1/folders/1stNg7da49BPJKWvvBNqYp078frUf-_KT]

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

The optimization 1 is adding the __restrict__ keyword to the arguments in the fusion kernels. It tells the compiler that pointers marked with __restrict__ can't alias, which means that they are non-overlapping memory regions. We expect that the memory usage for this one will improve after we implement that.

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

The optimization for this one is to simply add __restrict__ keyword to the mask, input and output arguments in the kernel. The correctness is shown in the table, which is that the accuracy of the batch equal to 10000 is 0.8714 and the Op time is slightly increases.

```
__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)
```

Fig 16

   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	*0.5843 ms*	*0.3481 ms*	*3.5134s*	*0.86*
1000	*3.9453*	*3.5341*	*9.4123s*	*0.886*
10000	*33.3471 ms*	*30.5303 ms*	*1m31.421s*	*0.8714*

We can see that the performance matches my expectations since the Op time and total execution time has decreased a little from the baseline while also maintaining the accuracy for batch size equals to 10000. In addition to that, from ncu report, we see that the memory throughput is also increasing a little, even though the hit rate seems to decrease a little.

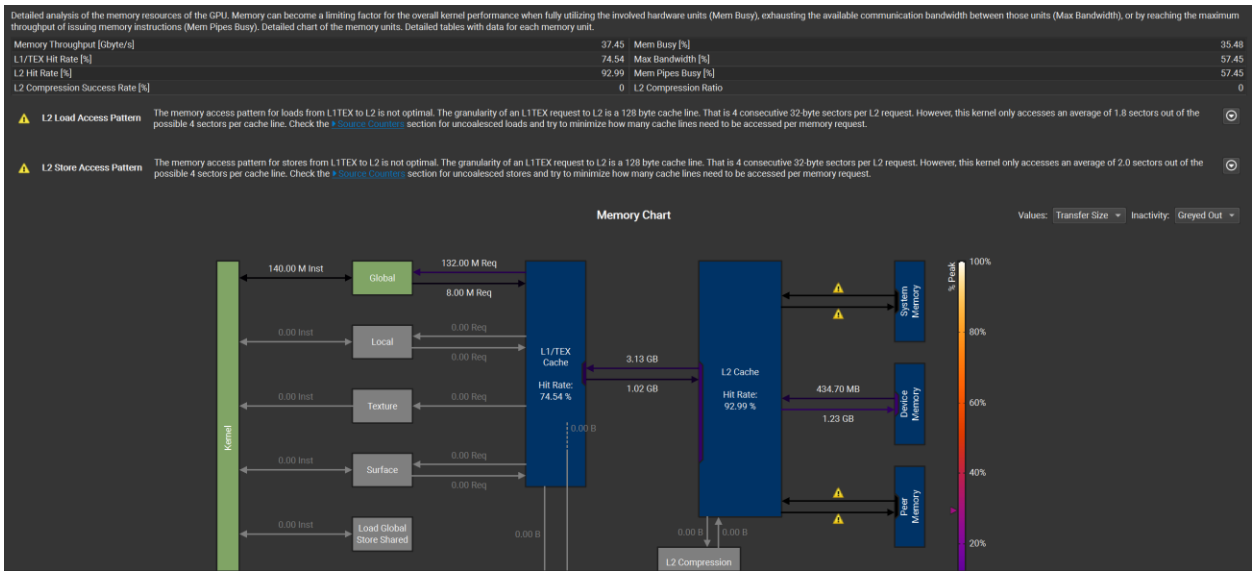

Fig 17

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

I think the __restrict__ keyword can synergize with other optimizations. For example, if we tell the system that the memory is not guaranteed to alias, the hardware can do the memory coalesce more easily.

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

1. Harris, M. (2013, August 12). *CUDA Pro Tip: Optimize pointer aliasing*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

4. Op_0: Constant Memory

[https://drive.google.com/drive/u/1/folders/1Kn08hdCELTY0GS_0BSYd42h0bo1Mzvwc]

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

The optimization 0 is adding the constant memory for the mask and using the constant_mask to implement CNN with fusion kernel. It will theoretically decrease the access to the mask memory times dramatically, so we expect that the ncu report can show us the decrease in the L1/L2 cache access to the device time.

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

We will declare a constant_mask for the mask to access the constant memory. The size is hard to find in this case, so we just pick a large value for now to make sure that all the information can be included – shown in the Fig 18, we use 10000 as the size of constant memory. After that, we need to replace "mask" with "constant_mask" we just defined. (Fig 19). We will also do memory copy from host to constant memory(Fig 20).

```
__constant__ float constant_mask [10000];

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

Fig 18

```
for (int i = 0; i < (mask_unrolled_width - 1) / TILE_WIDTH + 1; i++){

    // Load the mask into shared memory
    int col_m = i * TILE_WIDTH + tx;
    if ((row < mask_unrolled_height) && (col_m < mask_unrolled_width)){
        tileB[ty][tx] = constant_mask[row * mask_unrolled_width + col_m];
    } else {
        tileB[ty][tx] = 0;
    }
    // __syncthreads();
```

Fig 19

```
cudaMemcpy(*device_input_ptr, host_input, Batch * Channel * Height * Width * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(*device_mask_ptr, host_mask, Map_out * Channel * K * K * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpyToSymbol(constant_mask, host_mask, Map_out * Channel * K * K * sizeof(float));
```

Fig 20

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	*0.9341 ms*	*1.3241 ms*	*3.5134s*	*0.86*
1000	*3.8813 ms*	*5.3256 ms*	*14.3412s*	*0.886*
10000	*32.4589 ms*	*48.3032 ms*	*1m50.341s*	*0.8714*

We can see that the performance actually does not really match our expectations. From table, we see that for batch size equals 10000, the Op time 2 is increasing approximately 15 ms compared to the original one. From ncu report, we do see a decrease in the total access data values (approximately 430 MB in total from Fig 21), but the hit rate decreases. Hence, the total execution time might increase due to the fact of lot of missing elements in the process.

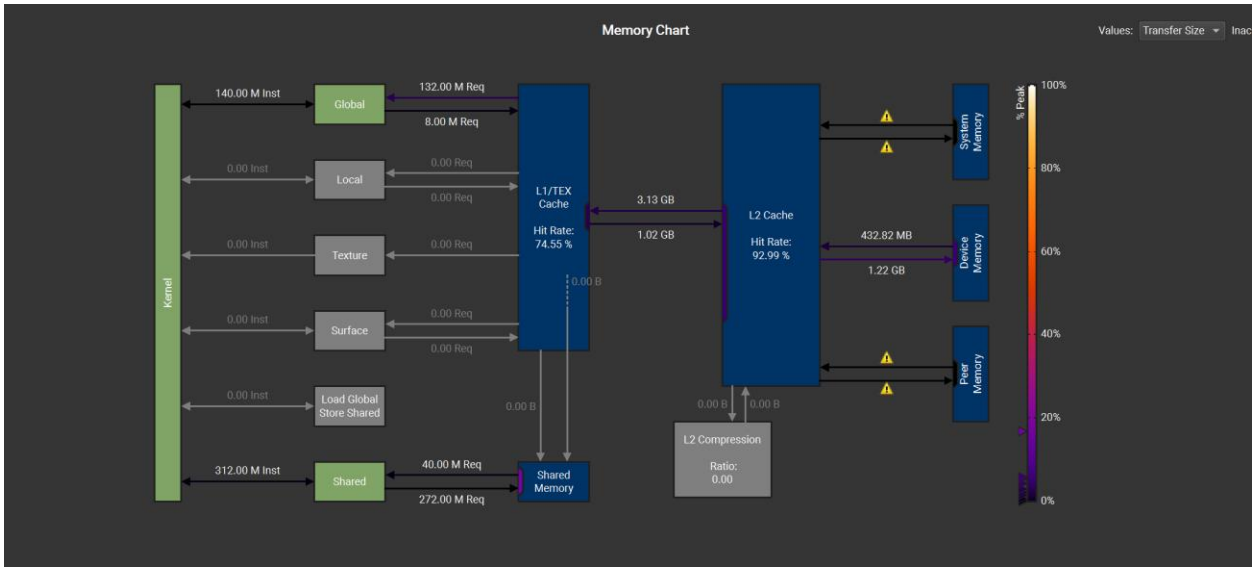

Fig 21

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

I think the constant memory can synergize with other optimizations, especially with memory coalescing. In other words, when threads in a warp access the same constant address, the data is fetched once from the constant cache, decreasing global memory transactions and improving memory throughput.

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

1. Harris, M. (2013, August 12). *CUDA Pro Tip: Optimize pointer aliasing*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/
2. NVIDIA Developer Forums. (2023, September 6). *Cuda constant memory*. https://forums.developer.nvidia.com/t/cuda-constant-memory/265584

5. **Op_2: Unrolling Loop**
   [https://drive.google.com/drive/u/1/folders/1oHNR1XezVKziNv7gFXwR4dCynNtRZHjO]
   a. How does this optimization theoretically optimize your convolution kernel? Expected behavior?

The optimization 2 is to unroll the loops in the fusion kernels so that the thread can be used more efficiently. In addition to that, loop unrolling can reduce loop control overhead and increasing instruction-level parallelism (ILP). We expect that the total duration of the kernel execution will drop and the instruction throughput will increase.

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

We will modify the inner loop in the fusion kernel to unroll the loop for 16 iterations. In Fig 21, when doing the addition of tile A and tile B values, I manually add each iterations in one shot so that we can get rid of the for loop.

```
    if (row < mask_unrolled_height && col < input_unrolled_width){
        // Unrolled version of the inner loop
        val += tileB[ty][0] * tileA[0][tx];
        val += tileB[ty][1] * tileA[1][tx];
        val += tileB[ty][2] * tileA[2][tx];
        val += tileB[ty][3] * tileA[3][tx];
        val += tileB[ty][4] * tileA[4][tx];
        val += tileB[ty][5] * tileA[5][tx];
        val += tileB[ty][6] * tileA[6][tx];
        val += tileB[ty][7] * tileA[7][tx];
        val += tileB[ty][8] * tileA[8][tx];
        val += tileB[ty][9] * tileA[9][tx];
        val += tileB[ty][10] * tileA[10][tx];
        val += tileB[ty][11] * tileA[11][tx];
        val += tileB[ty][12] * tileA[12][tx];
        val += tileB[ty][13] * tileA[13][tx];
        val += tileB[ty][14] * tileA[14][tx];
        val += tileB[ty][15] * tileA[15][tx];
    }
    __syncthreads();
}
```

Fig 22

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	*0.8342 ms*	*1.1264 ms*	*3.4854s*	*0.86*
1000	*3.7433 ms*	*5.1253 ms*	*14.1352s*	*0.886*
10000	*32.3437 ms*	*29.722 ms*	*1m30.441s*	*0.8714*

We can see that the performance matches the expectation as the execution time and operation time actually decreases a little while maintaining the accuracy. Also, from ncu report, we see that the total duration of the execution decreases(Fig 23) and the occupancy for the threads increases to 95.37%

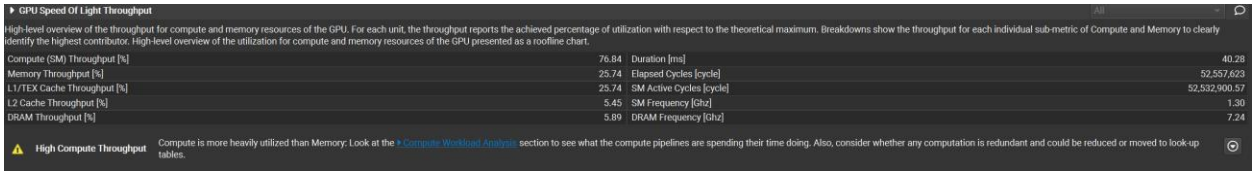

Fig 23

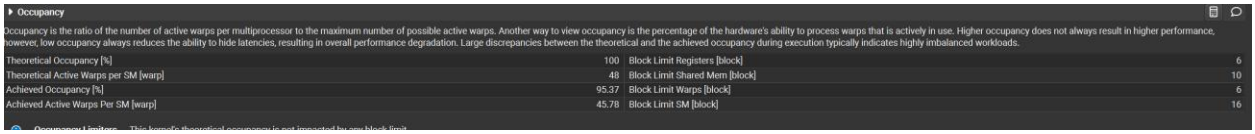

Fig 24

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

I think the loop unrolling does synergize with other optimizations. For example, it can work together with tiled share memory to boost the better memory access patterns and reuse. It can also work together with Tensor Cores to further improve the utilization of the threads

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

1. Volkov, V. (2011, November 14). *Unrolling parallel loops*. NVIDIA. https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf

6. Op_5: FP16 Implementation

[https://drive.google.com/drive/u/1/folders/1puqQq1QEmu8-365kSvx5VpcOF2aNyA-3]

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

The optimization 5 is to implement FP 32/16 into the fusion kernel. FP16 should theoretically allow GPUs with Tensor Cores to perform mixed-precision matrix operations at much higher throughput. Therefore, we expect to see the decrease in Op time and execution time from the output file. Also, we can see the decrease in the duration and better memory access time for cache part.

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

The implementation of FP16 will be similar to the TF32 implementation for requirement 1. First, we need to define several fragments(Fig 22) like TF32. However, we need to modify the type to half for FP16. Then, we will make sure that the tile A and tile B contains the correct data types, so we do __float2half for the inputs in each iteration.(Fig 23), Finally, we store the output values to tile C and do the synchronization.

```
wmma::fragment<wmma::matrix_a, TILE_WIDTH, TILE_WIDTH, TILE_WIDTH, half,  wmma::row_major> a_frag;
wmma::fragment<wmma::matrix_b, TILE_WIDTH, TILE_WIDTH, TILE_WIDTH, half,  wmma::row_major> b_frag;
wmma::fragment<wmma::accumulator, TILE_WIDTH, TILE_WIDTH, TILE_WIDTH, float>        c_frag;
wmma::fill_fragment(c_frag, 0.0f);
```

Fig 22

```
for (int i = 0; i < iteration; i++){

    #pragma unroll
    for (int j = 0; j < loop_unroll_factor; j++){
        int idx = i * loop_unroll_factor + j;
        int col_m = idx * TILE_WIDTH + tx;
        if ((row < mask_unrolled_height) && (col_m < mask_unrolled_width)){
            tileB[ty][tx] = __float2half(mask[row * mask_unrolled_width + col_m]
        } else {
            tileB[ty][tx] = __float2half(0);
        }
        int rowb = idx * TILE_WIDTH + ty;
        if ((rowb < input_unrolled_height) && (col < input_unrolled_width)) {
            int c = rowb / (K * K);

            int h = col / Width_out;
            int p = (rowb-c*K*K) / K;

            int w = col % Width_out;
            int q = (rowb-c*K*K) % K;

            tileA[ty][tx] = __float2half(in_4d(bz, c, h + p, w + q));
        } else {
            tileA[ty][tx] = __float2half(0);
            }
        __syncthreads();
        if (threadIdx.y < 2) {
            wmma::load_matrix_sync(a_frag, (half*)tileB, TILE_WIDTH);
            wmma::load_matrix_sync(b_frag, (half*)tileA, TILE_WIDTH);
            wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);
        }
        __syncthreads();
    }
```

Fig 23

```
if (threadIdx.y < 2) {
    wmma::store_matrix_sync((float*)tileC, c_frag, TILE_WIDTH, wmma::mem_row_major);
}
__syncthreads();

if (row < mask_unrolled_height && col < input_unrolled_width){
    // Calculate the output index
    output[bz * (mask_unrolled_height * input_unrolled_width) + row * input_unrolled_width + col] = tileC[threadIdx.y][threadIdx.x];
}

#undef out_4d
#undef in_4d
```

Fig 24

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	*0.6853 ms*	*0.8973 ms*	*2.9423s*	*0.86*
1000	*3.1423 ms*	*4.9325 ms*	*13.2452s*	*0.886*
10000	*30.3615 ms*	*19.1316 ms*	*1m29.241s*	*0.8714*

We can see that the performance matches the expectation as the execution time and operation time actually decreases a lot after we implement the FP16. From ncu report, we can see that FP16 utilizes memory better than the original design with silghtly higher hit rate in L2 cache(Fig 25). In addition, The total duration of the kernel has decreases to 40ms(Fig 26). Another notice is that the occupancy of the threads also improves to 95.38%. With all those improvements, we can achieve the higher operation and execution time.

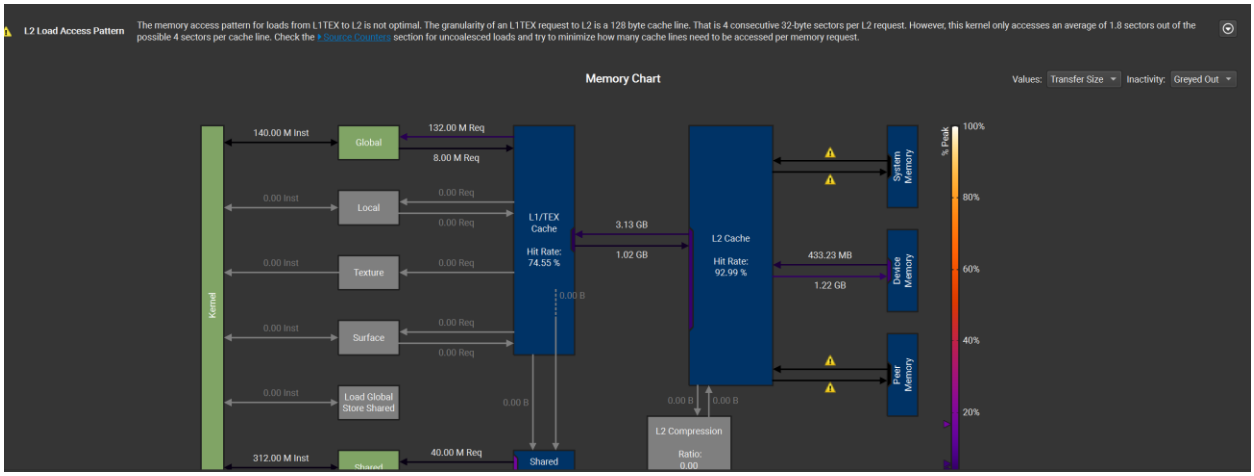

Fig 25

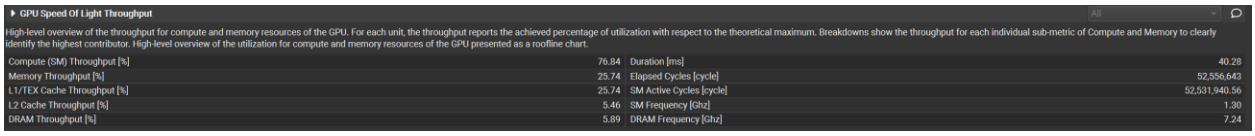

Fig 26

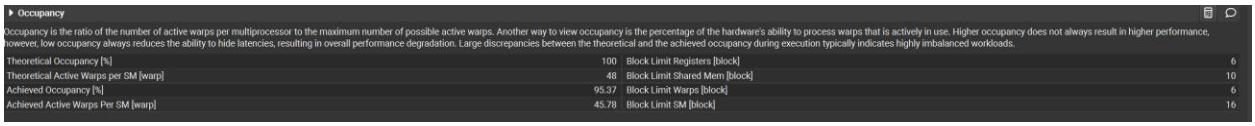

Fig 27

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

I think the FP16 optimization can synergize with other optimizations, such as loop unrolling, tiled share memory and so on. Since FP16 modifies the data types to half, it would be easier for the threads to do the allocation and if we have coalesce memory, it can process much faster comparing with the normal float type data.

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

1. Harris, M. (2016, October 19). *Mixed-precision programming with CUDA 8*. NVIDIA Technical Blog. https://developer.nvidia.com/blog/mixed-precision-programming-cuda-8/
2. NVIDIA. (n.d.). *__half structure — CUDA Math API documentation*. NVIDIA Developer. https://docs.nvidia.com/cuda/cuda-math-api/cuda_math_api/struct____half.html#_CPPv46__half

3. NVIDIA. (n.d.). *CUDA Math API: Half Precision Arithmetic Functions*. NVIDIA Developer. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__ARITHMETIC.html
4. NVIDIA. (n.d.). CUDA Math API: Half Precision Miscellaneous Functions. NVIDIA Developer. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html

7. Op_3: Sweeping various parameters to find best values

[https://drive.google.com/drive/u/1/folders/1B22vsXoXcgaqZvmvxLBxpJfrAGR4iW7E]

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

The optimization 3 is to sweep several block size and tile width values to find the optimized value. The size will affect the performance of fusion kernel a lot. For example, if the tile width is too large, we might require larger share memory and the loading time will be the problem; if we have width too small, higher latency will be introduced since we will require more global data transfer.

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

For optimization specifically, we only modify the block size and tile width in this section to sweep different values and find the optimized one. We have the following tables showing the different Op times and execution times for tile width ranging from 8 to 32.

```
#define BLOCK_SIZE 256
#define TILE_WIDTH 16
#define MATMUL_TILE_WIDTH 16 //8,  16,  32

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

Fig 28

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.8342 ms*	*1.1264 ms*	*3.4854s*	*0.86*
1000	*3.7433 ms*	*5.1253 ms*	*14.1352s*	*0.886*
10000	*32.3437 ms*	*29.722 ms*	*1m30.441s*	*0.8714*

Table 1: TILE_WIDTH = 16

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.9043 ms*	*1.1843 ms*	*3.4393s*	*0.86*
1000	*3.8634 ms*	*6.1345 ms*	*13.6342s*	*0.886*
10000	*33.2889 ms*	*42.9808 ms*	*1m40.214s*	*0.8714*

Table 2: TILE_WIDTH = 8

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.3042 ms*	*1.8943 ms*	*4.2921s*	*0.86*
1000	*5.3402 ms*	*7.4372 ms*	*20.0372s*	*0.886*
10000	*61.0035 ms*	*54.4061 ms*	*2m10.316s*	*0.8714*

Table 3: TILE_WIDTH = 32

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	*0.8342 ms*	*1.1264 ms*	*3.4854s*	*0.86*
1000	*3.7433 ms*	*5.1253 ms*	*14.1352s*	*0.886*
10000	*32.3437 ms*	*29.722 ms*	*1m30.441s*	*0.8714*

We can see that the best performance happens when TILE_WIDTH equals to 16. The proper tile size gives us slightly higher throughput values for the kernel – with IPC equals to 1.81 cycle/instruction vs 1.80 cycle/instruction(Fig 29 and Fig 30)

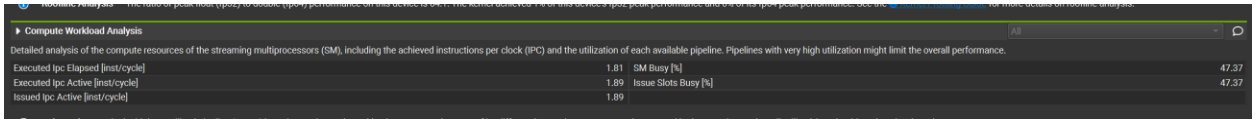

Fig 29

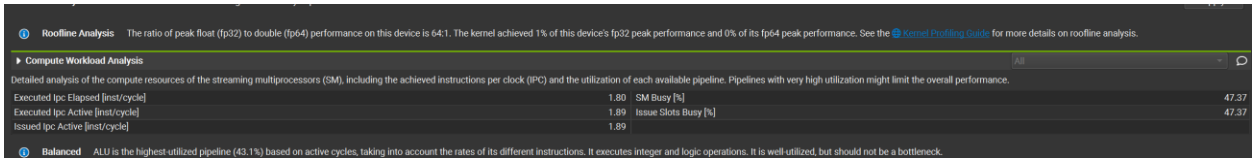

Fig 30

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

I think the best choice of tile width can synergize with other optimizations since it serves as the baseline for the share memory to access the data. If the width is the best, it can improve the throughput hugely for higher memory bandwidth

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

Kirk, D. B., & Hwu, W.-m. W. (2016). *Programming massively parallel processors: A hands-on approach* (3rd ed.). Morgan Kaufmann. [Chapter 4]