Kai-Jie Lin

Milestone 3 Report

[https://drive.google.com/drive/folders/1hYXiHKtpWlK4YVfhRnYvekSCt8O8p2rj?usp=sharing]

0. Baseline:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.53732ms*	*1.20928ms*	*4.151s*	*0.86*
1000	*11.862 ms*	*9.1758 ms*	*36.606s*	*0.886*
10000	*95.6191ms*	*72.8522ms*	*5m58.921s*	*0.8714*

1. Req_0: __Streams__

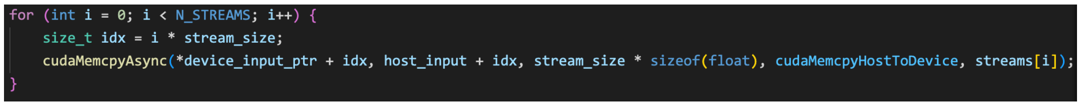
[https://drive.google.com/drive/folders/146rut8HuSy_XQAbXKVTYEV6fRNa5W4Q1?usp=drive_link]

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

Data transfers and kernel execution run concurrently in different CUDA streams. This optimization reduced end-to-end latency, caused higher GPU utilization and the memory transfer overhead is partially hidden.

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

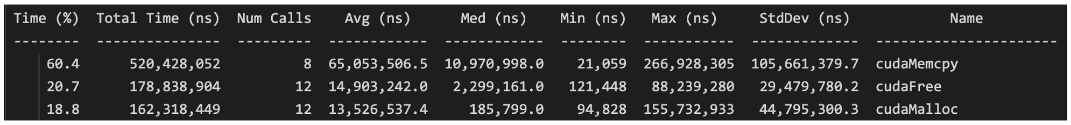
I implemented my work in conv_forward_gpu_prolog function. I also define a global constant named N_STREAMS = 2, which specify how many streams we want to use. The implementation is like the original one, except that we use streams to transfer the data between device and host. For example:



```
for (int i = 0; i < N_STREAMS; i++) {
    size_t idx = i * stream_size;
    cudaMemcpyAsync(*device_input_ptr + idx, host_input + idx, stream_size * sizeof(float), cudaMemcpyHostToDevice, streams[i]);
}
```

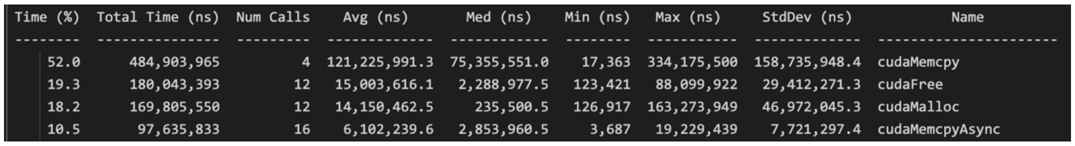
For the profiling results, we compare the time cost of data transfer part with milestone2:

M2:



```
Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)     Min (ns)   Max (ns)     StdDev (ns)            Name
--------  ---------------  ---------  ------------  ------------  --------  -----------  -------------  ----------------------
    60.4      520,428,052          8  65,053,506.5  10,970,998.0    21,059  266,928,305  105,661,379.7  cudaMemcpy
    20.7      178,838,904         12  14,903,242.0   2,299,161.0   121,448   88,239,280   29,479,780.2  cudaFree
    18.8      162,318,449         12  13,526,537.4     185,799.0    94,828  155,732,933   44,795,300.3  cudaMalloc
```

M3 streams:



```
Time (%)  Total Time (ns)  Num Calls    Avg (ns)        Med (ns)     Min (ns)   Max (ns)     StdDev (ns)            Name
--------  ---------------  ---------  -------------  ------------  --------  -----------  -------------  ----------------------
    52.0      484,903,965          4  121,225,991.3  75,355,551.0    17,363  334,175,500  158,735,948.4  cudaMemcpy
    19.3      180,043,393         12   15,003,616.1   2,288,977.5   123,421   88,099,922   29,412,271.3  cudaFree
    18.2      169,805,550         12   14,150,462.5     235,500.5   126,917  163,273,949   46,972,045.3  cudaMalloc
    10.5       97,635,833         16    6,102,239.6   2,853,960.5     3,687   19,229,439    7,721,297.4  cudaMemcpyAsync
```

We compare the cudaMemcpy and cudaMemcpyAsync part, the second picture shows that stream optimization cost less time since it transfers the data concurrently.

c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.72517ms*	*1.27065ms*	*4.147s*	*0.86*
1000	*11.7004ms*	*9.0554 ms*	*36.516s*	*0.886*
10000	*93.9101ms*	*71.0694ms*	*5m57.382s*	*0.8714*

The results have slightly improved. We can see the result from b question.

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

Yes, since this optimization is implemented in conv_forward_gpu_prolog function, other optimizations that implemented in kernel can synergize with it. For example, we implemented stream optimization in conv_forward_gpu_prolog function and tensor core optimization in matrix multiplication kernel.

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

How to Overlap Data Transfers in CUDA C/C++

David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016 Chapter 18 p.15

2. Req_1: __ Tensor Cores__

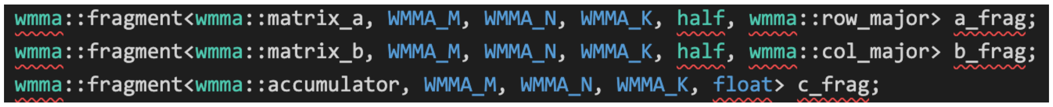
[https://drive.google.com/drive/folders/1du3TL1qOtechMtHYxhryAB-O6ie8jg6b?usp=drive_link]

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

Tensor Cores optimize convolutions by transforming them into 4x4x4 matrix multiplications that can be executed in a single clock cycle. They use mixed-precision computation with FP16 inputs and FP32 accumulation, balancing efficiency with accuracy. The im2col transformation reshapes convolution operations into matrix multiplications, enabling parallel processing across multiple cores. This typically yields 3-8x speedup versus traditional CUDA cores, with efficient memory access patterns through coalesced reads and cache utilization. While using FP16 could theoretically impact precision, the FP32 accumulation helps maintain accuracy while reducing memory bandwidth requirements and computational load.

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

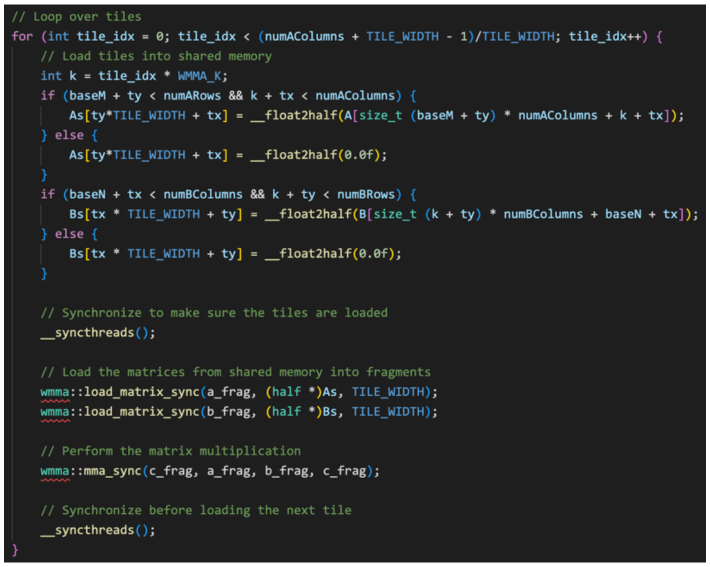
I implement matrix multiplication using NVIDIA's Tensor Cores through WMMA (Warp Matrix Multiply Accumulate) API in CUDA. The implementation uses shared memory and tiling for better performance.



```
wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::row_major> a_frag;
wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
```

These fragments store matrix data in Tensor Core-compatible format. Matrix A is row-major, B is column-major, and C accumulates results.

The main computation happens in the tile loop:



```
// Loop over tiles
for (int tile_idx = 0; tile_idx < (numAColumns + TILE_WIDTH - 1)/TILE_WIDTH; tile_idx++) {
    // Load tiles into shared memory
    int k = tile_idx * WMMA_K;
    if (baseM + ty < numARows && k + tx < numAColumns) {
        As[ty*TILE_WIDTH + tx] = __float2half(A[size_t (baseM + ty) * numAColumns + k + tx]);
    } else {
        As[ty*TILE_WIDTH + tx] = __float2half(0.0f);
    }
    if (baseN + tx < numBColumns && k + ty < numBRows) {
        Bs[tx * TILE_WIDTH + ty] = __float2half(B[size_t (k + ty) * numBColumns + baseN + tx]);
    } else {
        Bs[tx * TILE_WIDTH + ty] = __float2half(0.0f);
    }

    // Synchronize to make sure the tiles are loaded
    __syncthreads();

    // Load the matrices from shared memory into fragments
    wmma::load_matrix_sync(a_frag, (half *)As, TILE_WIDTH);
    wmma::load_matrix_sync(b_frag, (half *)Bs, TILE_WIDTH);

    // Perform the matrix multiplication
    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    // Synchronize before loading the next tile
    __syncthreads();
}
```

Important functions used:

__float2half(): Converts float to half-precision

wmma::load_matrix_sync(): Loads data into Tensor Core fragments

wmma::mma_sync(): Performs matrix multiplication using Tensor Cores

wmma::store_matrix_sync(): Stores results back to memory

__syncthreads(): Synchronizes threads within a block

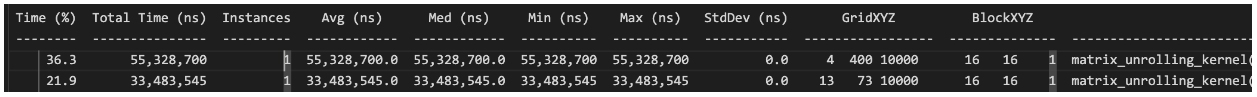
The implementation handles boundary conditions and uses proper indexing to ensure correct matrix multiplication even when dimensions aren't perfectly aligned with tile sizes.

c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.62383ms*	*1.25506ms*	*4.130s*	*0.86*
1000	*11.6713ms*	*9.04399ms*	*36.047s*	*0.886*
10000	*93.961 ms*	*71.3142ms*	*5m56.462s*	*0.8714*

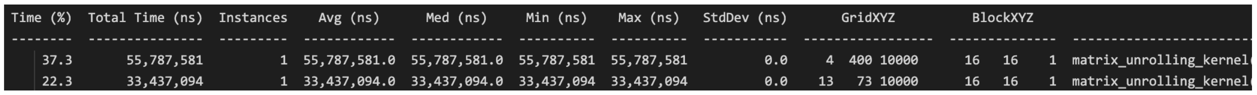
The result seems less improvements. Here I provided the time used in gpu kernels:

M2:



```
Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)       GridXYZ          BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  -------------  ----------------------
    36.3       55,328,700          1  55,328,700.0  55,328,700.0  55,328,700  55,328,700          0.0     4  400 10000    16   16    1  matrix_unrolling_kernel
    21.9       33,483,545          1  33,483,545.0  33,483,545.0  33,483,545  33,483,545          0.0    13   73 10000    16   16    1  matrix_unrolling_kernel
```

M3 tensor core:



```
Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)      Min (ns)    Max (ns)    StdDev (ns)      GridXYZ         BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  -------------  -----------------------
    37.3       55,787,581          1  55,787,581.0  55,787,581.0  55,787,581  55,787,581          0.0     4  400 10000     16   16     1  matrix_unrolling_kernel
    22.3       33,437,094          1  33,437,094.0  33,437,094.0  33,437,094  33,437,094          0.0    13   73 10000     16   16     1  matrix_unrolling_kernel
```

The second kernel has slightly improved.

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

   Yes, this optimization can synergize with other optimizations like streams and kernel fusion.

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

   Programming Tensor Cores in CUDA 9

   Warp Matrix Functions

   Lecture Slides: tensor operation

3. **Req_2: __ Kernel Fusion__**

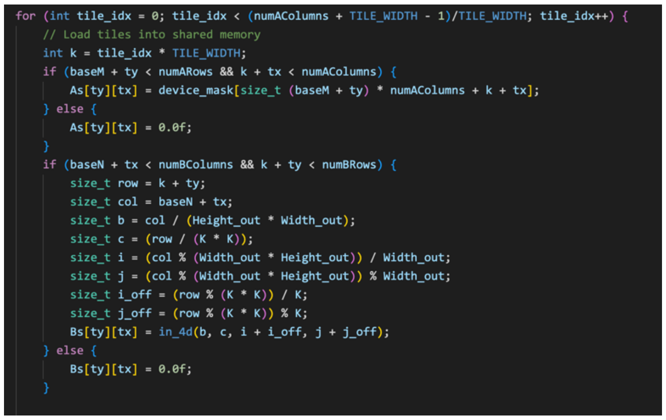
   [https://drive.google.com/drive/folders/1JPi15vb3jMjH41_6rm04_dxZ254cEHab?usp=drive_link]

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

      I fused three CUDA kernels (matrix unrolling, matrix multiplication, and permutation) to improve performance by reducing memory transactions and kernel launch overhead. We eliminate the overhead of launching three separate kernels and the need for intermediate storage buffers. This implementation should provide better performance through: 1. Reduced memory bandwidth usage by eliminating intermediate data structures 2. Better cache utilization through shared memory 3. Reduced kernel launch overhead 4. Improved memory access patterns.

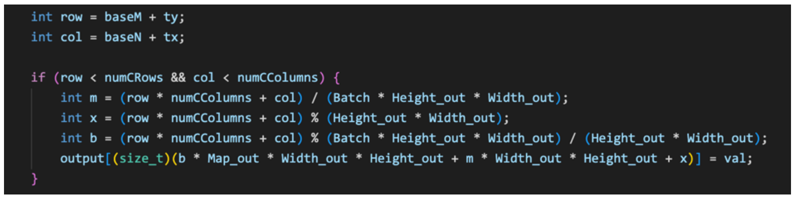
   b. How did you implement your code? Explain thoroughly and show code snippets. Justify the correctness of your implementation with proper profiling results.

      The implementation directly merges three CUDA kernels. In the loading tile part, instead loading value from intermediate input, we load data directly from original image input:



```
for (int tile_idx = 0; tile_idx < (numAColumns + TILE_WIDTH - 1)/TILE_WIDTH; tile_idx++) {
    // Load tiles into shared memory
    int k = tile_idx * TILE_WIDTH;
    if (baseM + ty < numARows && k + tx < numAColumns) {
        As[ty][tx] = device_mask[size_t (baseM + ty) * numAColumns + k + tx];
    } else {
        As[ty][tx] = 0.0f;
    }
    if (baseN + tx < numBColumns && k + ty < numBRows) {
        size_t row = k + ty;
        size_t col = baseN + tx;
        size_t b = col / (Height_out * Width_out);
        size_t c = (row / (K * K));
        size_t i = (col % (Width_out * Height_out)) / Width_out;
        size_t j = (col % (Width_out * Height_out)) % Width_out;
        size_t i_off = (row % (K * K)) / K;
        size_t j_off = (row % (K * K)) % K;
        Bs[ty][tx] = in_4d(b, c, i + i_off, j + j_off);
    } else {
        Bs[ty][tx] = 0.0f;
    }
```

Changing from the permute kernel, instead of launching whole kernel, we directly permute the output from matrix multiplication results:



```
  int row = baseM + ty;
  int col = baseN + tx;

  if (row < numCRows && col < numCColumns) {
      int m = (row * numCColumns + col) / (Batch * Height_out * Width_out);
      int x = (row * numCColumns + col) % (Height_out * Width_out);
      int b = (row * numCColumns + col) % (Batch * Height_out * Width_out) / (Height_out * Width_out);
      output[(size_t)(b * Map_out * Width_out * Height_out + m * Width_out * Height_out + x)] = val;
  }
```

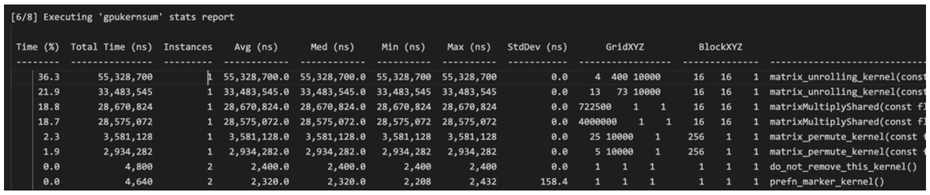
The result shows that my implementation is correct since it has a lot of improvement. We can see part c.

c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*0.508583ms*	*0.346769ms*	*4.183s*	*0.86*
1000	*4.30243 ms*	*3.24765 ms*	*36.595s*	*<accuracy>*
10000	*42.3195 ms*	*32.2163 ms*	*6m0.423s*	*<accuracy>*

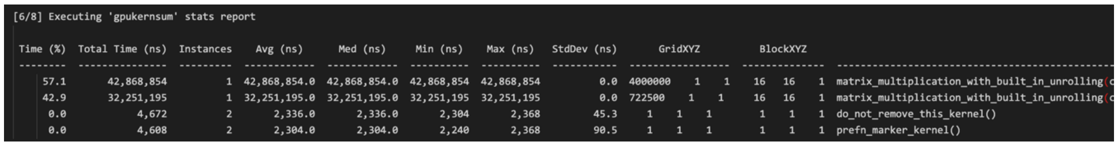
Kernel time cost:

M2:



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

 Time (%)  Total Time (ns)  Instances   Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)      GridXYZ         BlockXYZ
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  ------------  ----------------------------
     36.3       55,328,700          1  55,328,700.0  55,328,700.0  55,328,700  55,328,700          0.0     4  400 10000    16   16    1  matrix_unrolling_kernel(const
     21.9       33,483,545          1  33,483,545.0  33,483,545.0  33,483,545  33,483,545          0.0    13   73 10000    16   16    1  matrix_unrolling_kernel(const
     18.8       28,670,824          1  28,670,824.0  28,670,824.0  28,670,824  28,670,824          0.0  722500     1     1    16   16    1  matrixMultiplyShared(const fl
     18.7       28,575,072          1  28,575,072.0  28,575,072.0  28,575,072  28,575,072          0.0  4000000     1     1    16   16    1  matrixMultiplyShared(const fl
      2.3        3,581,128          1   3,581,128.0   3,581,128.0   3,581,128   3,581,128          0.0    25 10000     1   256    1    1  matrix_permute_kernel(const f
      1.9        2,934,282          1   2,934,282.0   2,934,282.0   2,934,282   2,934,282          0.0     5 10000     1   256    1    1  matrix_permute_kernel(const f
      0.0            4,800          2       2,400.0       2,400.0       2,400       2,400          0.0     1    1     1     1    1    1  do_not_remove_this_kernel()
      0.0            4,640          2       2,320.0       2,320.0       2,208       2,432        158.4     1    1     1     1    1    1  prefn_marker_kernel()
```

M3 kernel fusion:



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

 Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)       GridXYZ          BlockXYZ
 --------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  --------------  ------------------------------------------
     57.1       42,868,854          1  42,868,854.0  42,868,854.0  42,868,854  42,868,854          0.0  4000000    1    1    16   16    1  matrix_multiplication_with_built_in_unrolling(c
     42.9       32,251,195          1  32,251,195.0  32,251,195.0  32,251,195  32,251,195          0.0  722500    1    1     16   16    1  matrix_multiplication_with_built_in_unrolling(c
      0.0            4,672          2       2,336.0       2,336.0       2,304       2,368         45.3     1    1    1      1    1    1  do_not_remove_this_kernel()
      0.0            4,608          2       2,304.0       2,304.0       2,240       2,368         90.5     1    1    1      1    1    1  prefn_marker_kernel()
```

See from the pictures and results above. We can see a lot of improve from baseline. Especially the kernel time, instead of launching three kernel, launching one kernel cost less time.

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

We can cooperate other optimizations like streams and tensor core. We can implement stream optimization in conv_forward_gpu_prolog function and tensor core optimization in kernel.

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

David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016 Chapter 2 p.15

Kernel Fusion in CUDA

4. Op_1: __restrict__

[https://drive.google.com/drive/folders/1bh47z3ibjlPevUfPnh6fRLj2WdT4LFhm?usp=drive_link]

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

Two pointers alias if the memory to which they point overlaps. When a compiler can't determine whether pointers alias, it has to assume that they do. C/C++ compilers offer a way for the programmer to give the compiler information about pointer aliasing. By giving a pointer the restrict property, the programmer is promising the compiler that any data written to through that pointer is not read by any other pointer with the restrict property. In other words, the compiler doesn't have to worry that a write to a restrict pointer will cause a value read from another restrict pointer to change. This greatly helps the compiler optimize code.

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

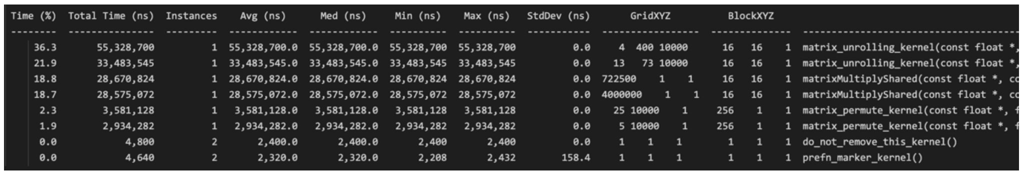
In every kernel, I add the __restrict__ argument to every pointer variable.

c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.66096ms*	*1.45994ms*	*4.093s*	*0.86*
1000	*11.7749ms*	*9.06719ms*	*36.101s*	*0.886*
10000	*94.153 ms*	*70.3902ms*	*5m56.413s*	*0.8714*

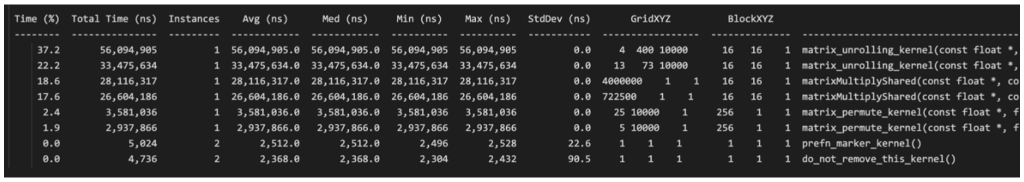
The second optime improve about 2ms.

M2:



```
Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)      GridXYZ          BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  --------------  ----------------------------------------
    36.3       55,328,700          1  55,328,700.0  55,328,700.0  55,328,700  55,328,700          0.0     4  400 10000    16   16     1  matrix_unrolling_kernel(const float *,
    21.9       33,483,545          1  33,483,545.0  33,483,545.0  33,483,545  33,483,545          0.0    13   73 10000    16   16     1  matrix_unrolling_kernel(const float *,
    18.8       28,670,824          1  28,670,824.0  28,670,824.0  28,670,824  28,670,824          0.0  722500     1     1    16   16     1  matrixMultiplyShared(const float *, co
    18.7       28,575,072          1  28,575,072.0  28,575,072.0  28,575,072  28,575,072          0.0  4000000     1     1    16   16     1  matrixMultiplyShared(const float *, co
     2.3        3,581,128          1   3,581,128.0   3,581,128.0   3,581,128   3,581,128          0.0    25 10000     1   256    1     1  matrix_permute_kernel(const float *, f
     1.9        2,934,282          1   2,934,282.0   2,934,282.0   2,934,282   2,934,282          0.0     5 10000     1   256    1     1  matrix_permute_kernel(const float *, f
     0.0            4,800          2       2,400.0       2,400.0       2,400       2,400          0.0     1     1     1     1    1     1  do_not_remove_this_kernel()
     0.0            4,640          2       2,320.0       2,320.0       2,208       2,432        158.4     1     1     1     1    1     1  prefn_marker_kernel()
```

M3_restrict:



```
Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)      GridXYZ          BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  --------------  ----------------------------------------
    37.2       56,094,905          1  56,094,905.0  56,094,905.0  56,094,905  56,094,905          0.0     4  400 10000    16   16     1  matrix_unrolling_kernel(const float *,
    22.2       33,475,634          1  33,475,634.0  33,475,634.0  33,475,634  33,475,634          0.0    13   73 10000    16   16     1  matrix_unrolling_kernel(const float *,
    18.6       28,116,317          1  28,116,317.0  28,116,317.0  28,116,317  28,116,317          0.0  4000000     1     1    16   16     1  matrixMultiplyShared(const float *, co
    17.6       26,604,186          1  26,604,186.0  26,604,186.0  26,604,186  26,604,186          0.0  722500     1     1    16   16     1  matrixMultiplyShared(const float *, co
     2.4        3,581,036          1   3,581,036.0   3,581,036.0   3,581,036   3,581,036          0.0    25 10000     1   256    1     1  matrix_permute_kernel(const float *, f
     1.9        2,937,866          1   2,937,866.0   2,937,866.0   2,937,866   2,937,866          0.0     5 10000     1   256    1     1  matrix_permute_kernel(const float *, f
     0.0            5,024          2       2,512.0       2,512.0       2,496       2,528         22.6     1     1     1     1    1     1  prefn_marker_kernel()
     0.0            4,736          2       2,368.0       2,368.0       2,304       2,432         90.5     1     1     1     1    1     1  do_not_remove_this_kernel()
```

We can observe the time cost of matrix multiplication kernel. This provide about 2ms improvement from M2 baseline.

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

Yes, this optimization can synergize with other optimizations. Since its implementation is adding __restrict__ to every pointer. It should work for any other optimizations.

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

CUDA Pro Tip: Optimize for Pointer Aliasing

5. Op_2 Loop Unrolling:

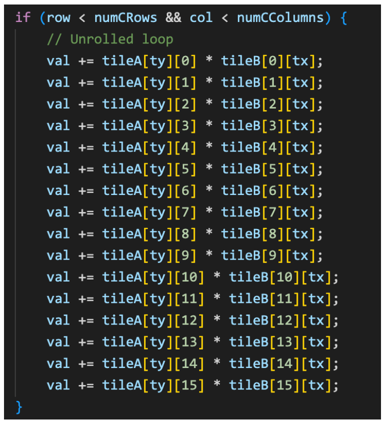
[https://drive.google.com/drive/folders/1zjqG9kbVJjs4xgdCl7tpDK7QjicCohos?usp=drive_link]

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

   Loop unrolling is an optimization technique in which a loop's iterations are expanded to reduce the overhead of loop control and potentially increase parallelism. In the unrolled version, each loop iteration now processes four elements, reducing the number of loop control operations. This can improve performance by minimizing branching overhead and increasing instruction-level parallelism.

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

   Unrolling the summation part in matrix multiplication:



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

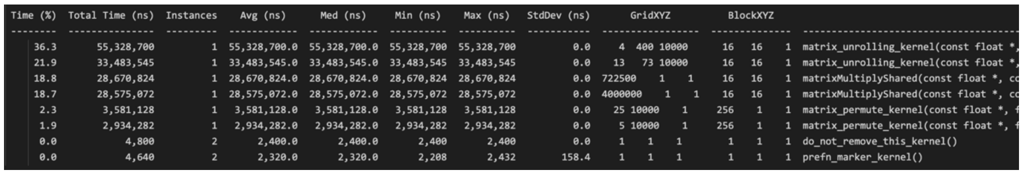
   It indeed improve some performance.

c. Did the performance match your expectation? Analyze the profiling results as a scientist.

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.6328 ms*	*1.25477ms*	*4.344s*	*0.86*
1000	*11.7051ms*	*8.89664ms*	*36.415s*	*0.886*
10000	*93.5056ms*	*68.8384ms*	*5m57.448s*	*0.8714*

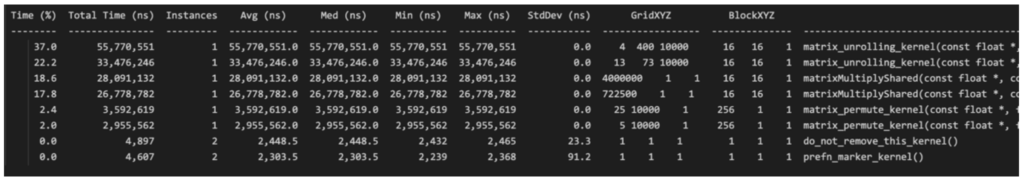
   We can see the 4ms improvement from baseline.

   M2:



```
Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)       GridXYZ         BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  --------------  ----------------------------------------
    36.3       55,328,700          1  55,328,700.0  55,328,700.0  55,328,700  55,328,700          0.0     4  400 10000     16   16     1  matrix_unrolling_kernel(const float *,
    21.9       33,483,545          1  33,483,545.0  33,483,545.0  33,483,545  33,483,545          0.0    13   73 10000     16   16     1  matrix_unrolling_kernel(const float *,
    18.8       28,670,824          1  28,670,824.0  28,670,824.0  28,670,824  28,670,824          0.0  722500     1     1     16   16     1  matrixMultiplyShared(const float *, co
    18.7       28,575,072          1  28,575,072.0  28,575,072.0  28,575,072  28,575,072          0.0  4000000     1     1     16   16     1  matrixMultiplyShared(const float *, co
     2.3        3,581,128          1   3,581,128.0   3,581,128.0   3,581,128   3,581,128          0.0    25 10000     1    256    1     1  matrix_permute_kernel(const float *, f
     1.9        2,934,282          1   2,934,282.0   2,934,282.0   2,934,282   2,934,282          0.0     5 10000     1    256    1     1  matrix_permute_kernel(const float *, f
     0.0            4,800          2       2,400.0       2,400.0       2,400       2,400          0.0     1     1     1      1    1     1  do_not_remove_this_kernel()
     0.0            4,640          2       2,320.0       2,320.0       2,208       2,432        158.4     1     1     1      1    1     1  prefn_marker_kernel()
```

M3_loop_unroll:



```
Time (%)  Total Time (ns)  Instances    Avg (ns)      Med (ns)     Min (ns)    Max (ns)   StdDev (ns)       GridXYZ         BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  --------------  ----------------------------------------
    37.0       55,770,551          1  55,770,551.0  55,770,551.0  55,770,551  55,770,551          0.0     4  400 10000     16   16     1  matrix_unrolling_kernel(const float *,
    22.2       33,476,246          1  33,476,246.0  33,476,246.0  33,476,246  33,476,246          0.0    13   73 10000     16   16     1  matrix_unrolling_kernel(const float *,
    18.6       28,091,132          1  28,091,132.0  28,091,132.0  28,091,132  28,091,132          0.0  4000000     1     1     16   16     1  matrixMultiplyShared(const float *, co
    17.8       26,778,782          1  26,778,782.0  26,778,782.0  26,778,782  26,778,782          0.0  722500     1     1     16   16     1  matrixMultiplyShared(const float *, co
     2.4        3,592,619          1   3,592,619.0   3,592,619.0   3,592,619   3,592,619          0.0    25 10000     1    256    1     1  matrix_permute_kernel(const float *, f
     2.0        2,955,562          1   2,955,562.0   2,955,562.0   2,955,562   2,955,562          0.0     5 10000     1    256    1     1  matrix_permute_kernel(const float *, f
     0.0            4,897          2       2,448.5       2,448.5       2,432       2,465         23.3     1     1     1      1    1     1  do_not_remove_this_kernel()
     0.0            4,607          2       2,303.5       2,303.5       2,239       2,368         91.2     1     1     1      1    1     1  prefn_marker_kernel()
```

Since the loop unrolling optimization is implemented in matrix multiplication kernel. The time cost of matrix multiplication kernel gets a lot of improvement.

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

Yes, it can synergize with some optimizations except tensor core. Since tensor core do the matrix multiplication in WMMA functions, we can not unroll it.

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

David Kirk/NVIDIA and Wen-mei Hwu, 2006-2016 Chapter 16 p.5

6. Op_5: FP16

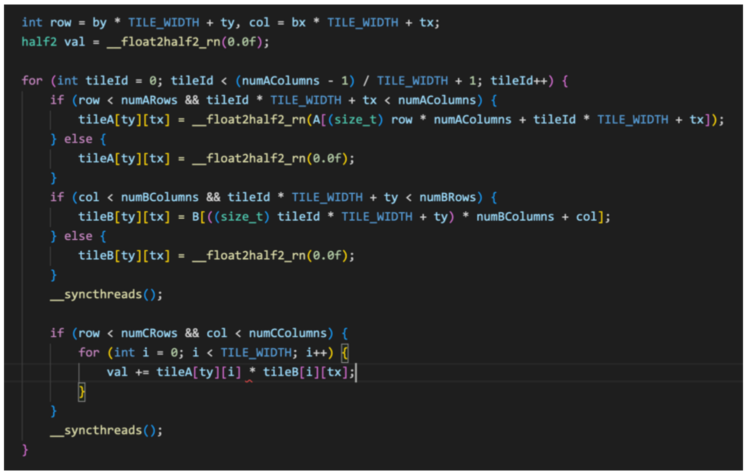
[https://drive.google.com/drive/folders/1Gom9QCb6WzW_D1tveQQSy7SWBtZWrvqM?usp=drive_link]

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

Storing FP16 (half precision) data compared to higher precision FP32 or FP64 reduces memory usage of the neural network, allowing training and deployment of larger networks, and FP16 data transfers take less time than FP32 or FP64 transfers.

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

For the input and mask, I use float type. For intermediate calculation results like unrolled matrix and matrix multiplication results, I use half type. I used half precision conversion function to convert float to half. For example:

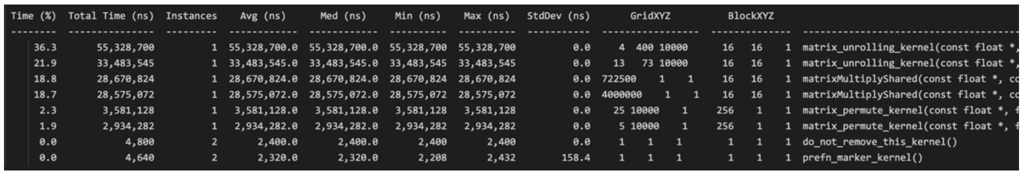


```
int row = by * TILE_WIDTH + ty, col = bx * TILE_WIDTH + tx;
half2 val = __float2half2_rn(0.0f);

for (int tileId = 0; tileId < (numAColumns - 1) / TILE_WIDTH + 1; tileId++) {
    if (row < numARows && tileId * TILE_WIDTH + tx < numAColumns) {
        tileA[ty][tx] = __float2half2_rn(A[(size_t) row * numAColumns + tileId * TILE_WIDTH + tx]);
    } else {
        tileA[ty][tx] = __float2half2_rn(0.0f);
    }
    if (col < numBColumns && tileId * TILE_WIDTH + ty < numBRows) {
        tileB[ty][tx] = B[((size_t) tileId * TILE_WIDTH + ty) * numBColumns + col];
    } else {
        tileB[ty][tx] = __float2half2_rn(0.0f);
    }
    __syncthreads();

    if (row < numCRows && col < numCColumns) {
        for (int i = 0; i < TILE_WIDTH; i++) {
            val += tileA[ty][i] * tileB[i][tx];
        }
    }
    __syncthreads();
}
```

c. Did the performance match your expectation? Analyze the profiling results as a scientist.

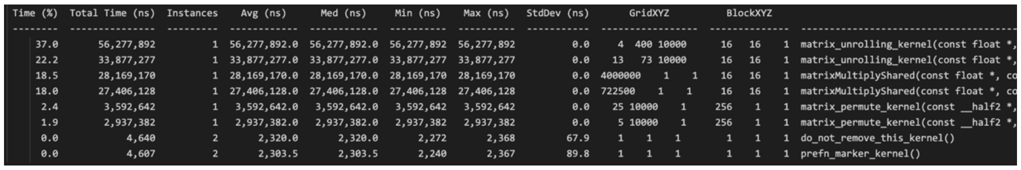
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	*1.70509ms*	*1.26151ms*	*4.124s*	*0.86*
1000	*38.9588ms*	*8.92879ms*	*36.695s*	*0.887*
10000	*94.1657ms*	*69.522ms*	*6m0.133s*	*0.8716*

M2:



```
Time (%)  Total Time (ns)  Instances   Avg (ns)      Med (ns)      Min (ns)    Max (ns)    StdDev (ns)      GridXYZ          BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  --------------  ----------------------------------------
    36.3       55,328,700          1  55,328,700.0  55,328,700.0  55,328,700  55,328,700          0.0     4  400 10000      16    16     1  matrix_unrolling_kernel(const float *,
    21.9       33,483,545          1  33,483,545.0  33,483,545.0  33,483,545  33,483,545          0.0    13   73 10000      16    16     1  matrix_unrolling_kernel(const float *,
    18.8       28,670,824          1  28,670,824.0  28,670,824.0  28,670,824  28,670,824          0.0  722500     1     1      16    16     1  matrixMultiplyShared(const float *, co
    18.7       28,575,072          1  28,575,072.0  28,575,072.0  28,575,072  28,575,072          0.0  4000000     1     1      16    16     1  matrixMultiplyShared(const float *, co
     2.3        3,581,128          1   3,581,128.0   3,581,128.0   3,581,128   3,581,128          0.0    25 10000     1     256     1     1  matrix_permute_kernel(const float *, f
     1.9        2,934,282          1   2,934,282.0   2,934,282.0   2,934,282   2,934,282          0.0     5 10000     1     256     1     1  matrix_permute_kernel(const float *, f
     0.0            4,800          2       2,400.0       2,400.0       2,400       2,400          0.0     1     1     1       1     1     1  do_not_remove_this_kernel()
     0.0            4,640          2       2,320.0       2,320.0       2,208       2,432        158.4     1     1     1       1     1     1  prefn_marker_kernel()
```

M3_FP16:



```
Time (%)  Total Time (ns)  Instances   Avg (ns)      Med (ns)      Min (ns)    Max (ns)    StdDev (ns)      GridXYZ          BlockXYZ
--------  ---------------  ---------  ------------  ------------  ----------  ----------  -----------  ----------------  --------------  ----------------------------------------
    37.0       56,277,892          1  56,277,892.0  56,277,892.0  56,277,892  56,277,892          0.0     4  400 10000      16    16     1  matrix_unrolling_kernel(const float *,
    22.2       33,877,277          1  33,877,277.0  33,877,277.0  33,877,277  33,877,277          0.0    13   73 10000      16    16     1  matrix_unrolling_kernel(const float *,
    18.5       28,169,170          1  28,169,170.0  28,169,170.0  28,169,170  28,169,170          0.0  4000000     1     1      16    16     1  matrixMultiplyShared(const float *, co
    18.0       27,406,128          1  27,406,128.0  27,406,128.0  27,406,128  27,406,128          0.0  722500     1     1      16    16     1  matrixMultiplyShared(const float *, co
     2.4        3,592,642          1   3,592,642.0   3,592,642.0   3,592,642   3,592,642          0.0    25 10000     1     256     1     1  matrix_permute_kernel(const __half2 *,
     1.9        2,937,382          1   2,937,382.0   2,937,382.0   2,937,382   2,937,382          0.0     5 10000     1     256     1     1  matrix_permute_kernel(const __half2 *,
     0.0            4,640          2       2,320.0       2,320.0       2,272       2,368         67.9     1     1     1       1     1     1  do_not_remove_this_kernel()
     0.0            4,607          2       2,303.5       2,303.5       2,240       2,367         89.8     1     1     1       1     1     1  prefn_marker_kernel()
```

We can see that the unrolling kernel of M3 performs slightly worse than M2. I think the reason is float2half conversion overhead. While there is overhead in float2half conversion, we can see the improvement of matrix multiplication kernel.

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

Yes, this optimization can synergize with other, since half precision only affect types. We can only modify the data type part.

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

   Mixed-Precision Programming with CUDA 8