Ziye Tang

Milestone 3 Report

0. Baseline:

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.207399 ms	0.519773 ms	0m1.759s	0.86
1000	1.29363 ms	5.1869 ms	0m9.330s	0.886
10000	12.2446 ms	55.4134 ms	1m31.221s	0.8714

1. Optimization Number #0: __Using Streams to overlap computation with data transfer__

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

In stream optimization, large datasets are divided into smaller segments that are processed in parallel by multiple streams. Specifically, I divide the data along the batch dimension, enabling each stream to handle a smaller batch of images. This approach allows for the overlapping of data transfer and computation across adjacent segments. By concurrently executing these tasks, we can significantly reduce the total running time required for the convolutional forward pass, enhancing the efficiency and throughput of the processing pipeline.

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

1. To apply streams for optimization, I only need to modify host codes. I moved all host codes into the function “conv_forward_gpu_prolog”, so that I can run data transferring and computing for multiple times.

2. Here is the initialization of streams. I created 10 streams, and the data will be evenly assigned to all stream

```
int num_stream = 10;
cudaStream_t stream[num_stream];
for (int i = 0; i < num_stream; i++)
    cudaStreamCreate(&stream[i]);

int seg_size = Batch / num_stream;
int seg_input_size = seg_size * Channel * Height * Width * sizeof(float);
int seg_output_size = seg_size * Map_out * Height_out * Width_out * sizeof(float);
```

3. Since I will need to read and write to host data for multiple times, I will need pinned memory for the host data to avoid being paged out.

```
float *host_input_pinned, *host_output_pinned;
cudaMallocHost(&host_input_pinned, Batch * Channel * Height * Width * sizeof(float));
cudaMallocHost(&host_output_pinned, Batch * Map_out * Height_out * Width_out * sizeof(float));
cudaMemcpy(host_input_pinned, host_input, Batch * Channel * Height * Width * sizeof(float), cudaMemcpyHostToHost);
```

4. I give each stream distinct memory space for device input and output, they can share the same mask

```
float *device_input[num_stream];
float *device_output[num_stream];
float * device_mask;
for (int i = 0; i < num_stream; i++) {
    cudaMalloc((void**) &(device_output[i]), seg_output_size);
    cudaMalloc((void**) &(device_input[i]), seg_input_size);
}
cudaMalloc((void**) &device_mask, kernel_size);
```

5. Here is the modified grid and block configuration. I the x dimention of grid is now the segment size, which is Batch / num_stream:

```
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);
dim3 dimGrid(seg_size, Map_out, y_dim_grid);
```

6. The following is the code for data transfer and compute. I transfer input data for all streams first, and then run the kernel. And finally, I transfer the output data from all streams to the host.

```
cudaMemcpyAsync(device_mask, host_mask, kernel_size, cudaMemcpyHostToDevice, stream[0]);
for (int i=0; i<Batch; i+=seg_size * num_stream) {
    for (int j = 0; j < num_stream; j++) {
        cudaMemcpyAsync(device_input[j], host_input + (i+j*seg_size) * Channel * Height * Width, seg_input_size, cudaMemcpyHostToDevice, s
    }

    for (int j = 0; j < num_stream; j++) {
        conv_forward_kernel<<<dimGrid, dimBlock, 0, stream[j]>>>(device_output[j], device_input[j], device_mask, Batch, Map_out, Channel,
    }

    for (int j = 0; j < num_stream; j++) {
        cudaMemcpyAsync(host_output_pinned + (i+j*seg_size) * Map_out * Height_out * Width_out, device_output[j], seg_output_size, cudaMem
    }
}
```

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

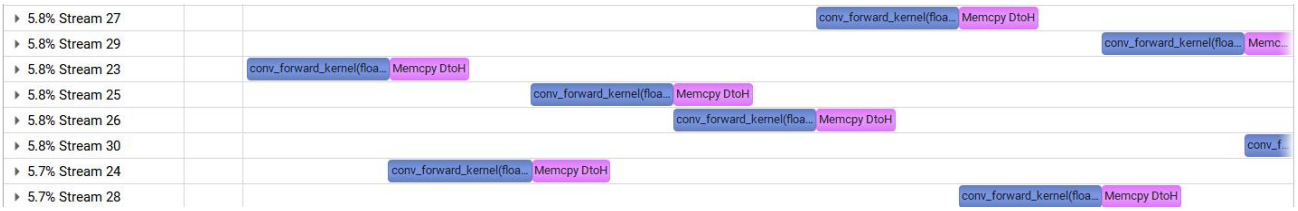
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.315 ms	0.494 ms	0m2.377s	0.86
1000	1.247 ms	4.754 ms	0m9.914s	0.886
10000	12.099ms	52.058 ms	1m31.760s	0.8714

Although, we original way of obtaining op time doesn't work, I can still find them in the profiling out put. Here's the kernel execution time:

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
81.1	52,058,582	10	5,205,858.2	5,206,649.0	5,197,176	5,211,961	4,109.1	1000	16	9	16	16	1	conv_forwa
18.9	12,099,780	10	1,209,978.0	1,210,045.5	1,205,694	1,211,549	1,652.9	1000	4	25	16	16	1	conv_forwa

From this, we can find that the two Op Time are 12.099ms and 52.058ms. Compared with baseline, the the Op Time running the second (the one that takes longer) conv_forward_kernel decreases by around 3ms.

According to the results shown from Nsignt System, the computing and data transfering are overlapped. Therefore, the performance of my codes matches my expectation.



The lack of improvement in total execution time despite using stream optimization could be attributed to several factors. Firstly, additional time is required for allocating pinned memory and transferring data from the host to this pinned memory. As shown below, memory allocation, copy and free for the pinned memory takes musch longer time compared with other Cuda API calls.

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
46.5	726,677,661	4	181,669,415.3	196,261,496.5	68,724,266	265,430,402	86,782,215.7	cudaHostAlloc
22.5	351,873,077	6	58,645,512.8	17,072,202.5	36,749	169,351,022	78,270,899.7	cudaMemcpy
15.1	235,099,349	4	58,774,837.3	51,860,840.5	30,418,117	100,959,551	33,546,897.5	cudaFreeHost
13.2	206,322,877	44	4,689,156.3	77,275.0	59,471	202,535,858	30,520,189.1	cudaMalloc
2.6	39,870,125	42	949,288.7	25,237.0	3,106	2,281,279	1,010,909.7	cudaMemcpyAsync

Secondly, there is inherent overhead associated with launching kernels. As stream optimization necessitates multiple kernel launches, these overheads accumulate, which can obscure any potential gains from parallel processing, resulting in less noticeable improvements in execution time.

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

Since the optimization using stream only involve modification to the host code, any other optimizations that only modify the kernel code can synergize with stream optimization. For example, it stream optimization can synergize with matrix unrolling, kernel fusion, constant memory optimization, loop unrolling, atomics, and FP 16.

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

1. ece408-lecture19-GPU-in-PC-Architecture-vk-SP24.pdf, https://canvas.illinois.edu/courses/43562/files/folder/lectures
2. ece408-lecture20-GPU-Data-Transfer-vk-SP24.pdf, https://canvas.illinois.edu/courses/43562/files/folder/lectures

2. **Optimization Number #1: __Tiled (shared memory) convolution__**

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

   In standard convolution operations, input data is frequently reused to calculate various output entries, typically involving multiple accesses to slow global memory. To optimize this, shared memory is employed for each processing block, storing necessary input data locally. This approach reduces the frequency of global memory access, as the data once loaded into shared memory can be reused multiple times with much faster access speeds. In theory, the bandwidth reduction would be (TILE_WIDTH * MASK_WIDTH)^2 / (TILE_WIDTH + MASK_WIDTH - 1)^2. Consequently, this reduces the kernel's runtime by limiting the expensive global memory fetch operations and enhancing overall computational efficiency, effectively speeding up the entire process.

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

   I implement the strategy 2 tiled convolution, which is mentioned in Lecture 8.

   From the host side, I modifed the block size from (TILE_WIDTH, TILE_WIDTH, 1) to (TILE_WIDTH - K + 1, TILE_WIDTH - K + 1, 1), so that all threads participate in loading the input data to shared memory, but only TILE_WIDTH* TILE_WIDTH threads will be responsible for the computing.

   Since the kernel size is unknown at compile time, I have to use dynamic shared memory. In host code, when invoking the kernel, I need to specify shared memory allocation size per thread block, as shown below:

```
size_t shared_size = (Channel * (TILE_WIDTH + K - 1) * (TILE_WIDTH + K - 1)) * sizeof(float);
conv_forward_kernel<<<dimGrid, dimBlock, shared_size>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width, K)
```

And when declaring the shared variable, I will need to add “extern”:

```
extern __shared__ float tile[];
```

The following are kernel codes for storing input data in shared memory and computing:

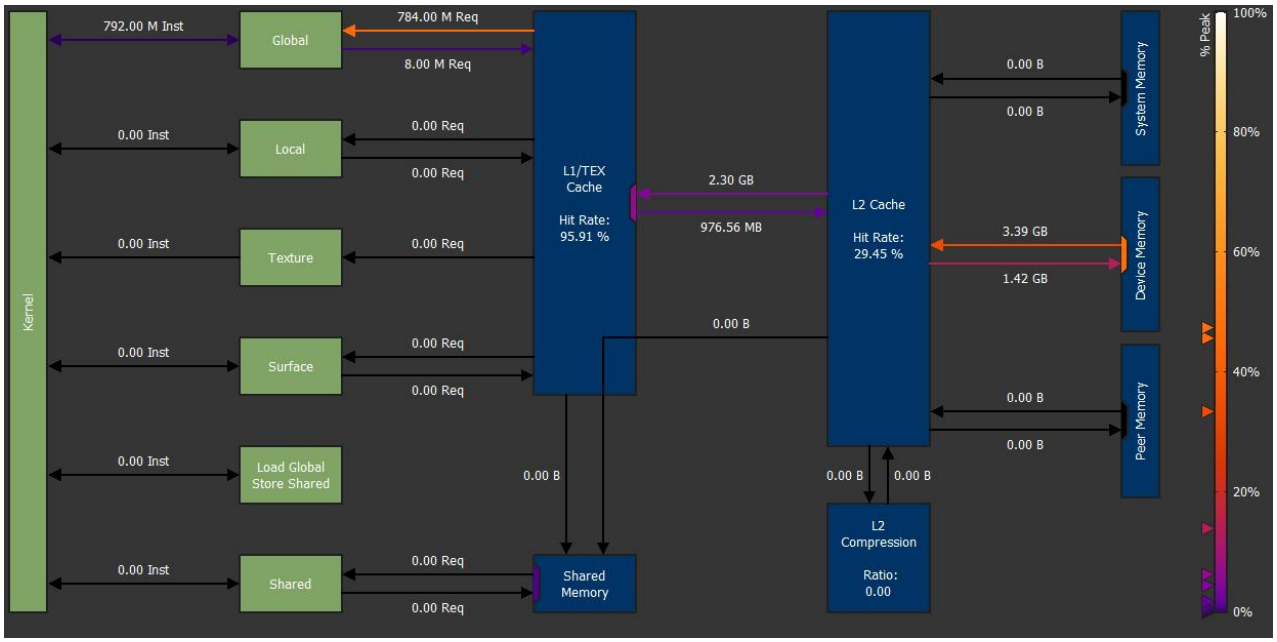
```
extern __shared__ float tile[];
if (h < Height && w < Width) {
    for (int c = 0; c < Channel; c++) {
        tile[c*tile_size*tile_size + ty * tile_size + tx] = in_4d(b, c, h, w);
    }
} else {
    for (int c = 0; c < Channel; c++) {
        tile[c*tile_size*tile_size + ty * tile_size + tx] = 0.0f;
    }
}

__syncthreads();

if (h < Height_out && w < Width_out && ty < TILE_WIDTH && tx < TILE_WIDTH)  {
    float acc = 0.0f;
    for (int c = 0; c < Channel; c++) {
        for (int p = 0; p < K; p++) {
            for (int q = 0; q < K; q++) {
                acc += tile[c*tile_size*tile_size + (p+ty) * tile_size + q+tx] * mask_4d(m, c, p, q);
            }
        }
    }
    out_4d(b, m, h, w) = acc;
}
```

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

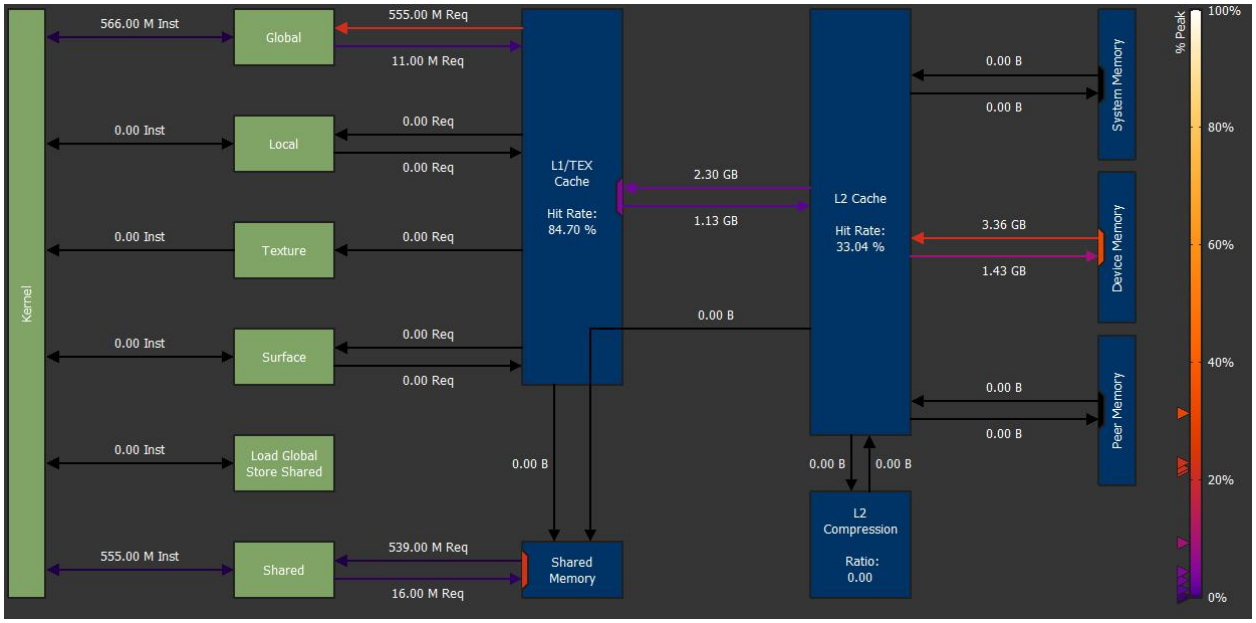
Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.279774 ms	0.765633 ms	0m1.238s	0.86
1000	1.88783 ms	7.48267 ms	0m9.595s	0.886
10000	18.0552 ms	73.4623 ms	1m32.676s	0.8714

The following are memory charts for baseline and op#1

Baseline:



Op#1:



As shown in the char, in op#1, the kernel requested 539 M of data from the shared memory. However, in baseline, the kernel requested no data from the shared memory. And we can find that, compared with baseline, op#1 requested less data from the global memory, which is what we expected.

However, compared with the baseline, the kernel running time of this optimization is even worse. In my opinion, one possible reason can be that the chosen tile size may not be optimal. Also, implementing tiling introduces additional complexity such as managing shared memory, synchronizing threads, and handling tile boundaries. This overhead can sometimes negate the benefits.

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

This optimization can synergize with stream as mentioned previously because stream optimization only modifies how the host transfer data and invoke the kernel. Also, this optimization can synergize with input matrix unrolling & matrix multiplication, which is exactly op#2. This optimization synergizes with op#4, because there's no conflicts between these two optimization. I can store the mask in constant memory, whils still storing input data into shared memory. By stakcing with op#4, the kernel running time will be further improved because the constant memory access is much faster than global memory. This optimization can synergize with op#5, because loop unrolling makes the computation faster, while the optimization with shared memory makes the data reading faster. So there is not conflict on stacking this optimization with op#5.

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

1. ece408-lecture8-tiled-convolution-vk-SP24.pdf，
https://canvas.illinois.edu/courses/43562/files/folder/lectures
2. Harris, Mark (2013). Using Shared Memory in CUDA C/C++. Nvidia Developer.
https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/

3. **Optimization Number #2: __Input matrix unrolling & tiled matrix multiplication using shared memory__**

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

   Matrix unrolling will convert each input image (with all channels) in to a matrix who has (C * K * K) rows, and has (H_out * W_out) columns. Each column of the matrix corresponds to an output entry. Each (K * K) entries in a column corresponds to all the input data needed for each channel. The mask will also be considered as a matrix, where each row represents the mask for each output feature map. Therefore, performing the convolution is equivalent to calculating matrix multiplication between the mask matrix and the unrolled input.

   Matrix unrolling enhances computational efficiency by leveraging optimized routines available in modern CPUs and GPUs. This technique enables the use of highly parallel matrix multiplication algorithms, reducing the overhead associated with iterative convolution computations. Although matrix unrolling generally results in faster execution times due to improved hardware utilization and parallelism, it does increase memory usage, as it requires large intermediate matrices.

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

   When implementing this optimization, I found that the kernel cannot perform unrolling and matrix multiplication at once when there are 10k batches of images. So, I decided to cut the data into smaller pieces, and call the kernel iteratively on each piece of data ("i" can be used as the starting batch index in each kernel):

```
cudaMalloc((void**) &unrolled_matrix,  seg_size * numRow * numCol * sizeof(float));
for (int i = 0; i < Batch; i+=seg_size) {
    unroll_kernel<<<dimGrid_unroll, dimBlock_unroll>>>(unrolled_matrix, device_input, Channel, Height, Width, K, i);
    conv_forward_kernel<<<dimGrid_forward, dimBlock_forward>>>(device_mask, unrolled_matrix, device_output, Map_out, numRow, numCol, i);
}
cudaFree(unrolled_matrix);
```

   For the unroll_kernel, I set the grid dimension to be dimGrid_unroll(W_grid, H_grid, seg_size). Here is how I obtain each index of the current output data.

```
int b = blockIdx.z;
int w = blockIdx.x * TILE_WIDTH + threadIdx.x;
int h = blockIdx.y * TILE_WIDTH + threadIdx.y;
```

Since each thread corresponds to a dinstinct output entry, it is in charge with building one whole column of the unrolled data. I will need to traverse through all the input entries that will be used to compute the current output. For each input data with location (c, h+p, w+q), the corresponiding row index of unrolled data is c*K*K+p*K+q, and the column index is h*Width_out+w:

```
if (h < Height_out && w < Width_out) {
    for (int c = 0; c < Channel; c++) {
        int w_base = c * (K*K);
        for (int p = 0; p < K; p++) {
            for (int q = 0; q < K; q++) {
                out_3d(b, w_base + p * K + q, h * Width_out + w) = in_4d(b+start, c, h + p, w + q);
            }
        }
    }
}
```

For the tiled matrix multiplication kernel, I did exactly the same as my lab3.

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.915745 ms	1.08946 ms	0m1.849s	0.86
1000	6.47774 ms	10.9756 ms	0m9.897s	0.886
10000	59.4673 ms	107.046 ms	1m30.566s	0.8714

The op times are much longer than the baseline. One big reason is that unrolling takes a lot of time. As shown in profile.out, majority of the time was taken for data unrolling.

Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ			BlockXYZ			
76,470,744	2	38,235,372.0	38,235,372.0	38,213,260	38,257,484	31,271.1	3	3	5000	16	16	1	unroll_kernel(f
28,463,606	2	14,231,803.0	14,231,803.0	14,225,275	14,238,331	9,232.0	5	5	5000	16	16	1	unroll_kernel(f
28,409,450	2	14,204,725.0	14,204,725.0	14,203,845	14,205,605	1,244.5	73	1	5000	16	16	1	conv_forward_ke
27,965,174	2	13,982,587.0	13,982,587.0	13,981,115	13,984,059	2,081.7	400	1	5000	16	16	1	conv_forward_ke
5,184	2	2,592.0	2,592.0	2,464	2,720	181.0	1	1	1	1	1	1	do_not_remove_t
4,607	2	2,303.5	2,303.5	2,303	2,304	0.7	1	1	1	1	1	1	prefn_marker_ke

If we only look at the matrix multiplication part (in my code, I keep the name conv_forward_kernel), we can see the kernel execution times are around 28 ms for both convolution. The larger convolution get improved, but the smaller one gets worse. The reason might be that more complex convolution can make better use of the parallelism and memory throughput advantages of matrix multiplication. In contrast, simpler convolutions might not have enough computational workload to outweigh the overhead introduced by the matrix setup and transformation process.

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

As mentioned previously, matrix unrolling can synergize with stream optimization, and tiled convolution. And I believe that storing mask into constant memory can still take effect in matrix multiplication because it simply make it faster reading mask data. I don't think atomics can be applied in this optimization because there is no contention. Fixed point arithmetic can be used in this optimization, as all we need to do is to cast the data type into __half when doing the computation.

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

1. ece408-lecture12-CNN-vk-SP24.pdf, https://canvas.illinois.edu/courses/43562/files/folder/lectures

4. Optimization Number #3: __Kernel fusion for unrolling and matrix-multiplication__

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

In this optimization, matrix unrolling and matrix multiplication are integrated into a single kernel, reducing the number of kernel launches and thus minimizing the associated overheads, which in turn enhances the running time.

b. How did you implement your code? Explain thoroughly and show code snippets. In order to put unrolling and matrix multiplication into one kernel. I changed the gird configuration of unrolling to fit matrix multiplication. So, in the fused kernel, I will only need to modify the implementation of unrolling. Now, the grid x dimension becomes (Height_out * Width_out / TILE_WIDTH), and the y dimension becomes (Map_out / TILE_WIDTH), so we cannot directly get h and w index for the input data. We first calculate the row and col index in the unrolled data:

```
int Row = TILE_WIDTH * by + ty;
int Col = TILE_WIDTH * bx + tx;
```

Then, we can calculate h and w by:

```
int h = Col/Width_out;
int w = Col%Width_out;
```

In the new gird configuration, each thread corresponds to a unrolled data entry. among threads that correspond to the same column index, I only need one of them to perform the unrolling:

```
if (Col < Height_out * Width_out) {
    int h = Col/Width_out;
    int w = Col%Width_out;
    if(Row == 0) {
        for (int c = 0; c < Channel; c++) {
            int w_base = c * (K*K);
            for (int p = 0; p < K; p++) {
                for (int q = 0; q < K; q++) {
                    B_3d(b, w_base + p * K + q, h * Width_out + w) = in_4d(b+start, c, h + p, w + q);
                }
            }
        }
    }
}
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution	Accuracy

			Time	
100	0.76285 ms	0.689673 ms	0m1.338s	0.86
1000	4.94758 ms	5.20124 ms	0m9.635s	0.886
10000	45.3212 ms	48.079 ms	1m33.846s	0.8714

Compared with op#2, the kernel fusion halved the total kernel running time by around 70 ms. Therefore, this optimization matches my expectation

As shown in profile.out, only one kernel is called in each convolutoin:

Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	GridXYZ	BlockXYZ	
45,631,341	2	22,815,670.5	22,815,670.5	22,807,991	22,823,350	10,860.5	73 1 5000	16 16 1	conv_forward_k
41,511,324	2	20,755,662.0	20,755,662.0	20,753,646	20,757,678	2,851.1	400 1 5000	16 16 1	conv_forward_k

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

This optimization can synergize with any operations that synergizes with op#2 because the only difference in this optimization is that I put matrix unrolling and matrix multiplication into one kernel.

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

1. ece408-lecture12-CNN-vk-SP24.pdf, https://canvas.illinois.edu/courses/43562/files/folder/lectures

5. Optimization Number #4: __Weight matrix (Kernel) in constant memory__

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

In this optimization, mask data is stored in constant memory instead of global memory, so that the kernel can read mask data faster. The total execution time of the kernel is expected to be lower than the baseline.

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

The mask variable is declared globally at the beginning. I gave it a fixed size of 6000 so that it can store all mask data:

```
__constant__ float dk[6000];
```

The memory copy function for constant memory is a bit different:

```
cudaMemcpyToSymbol(dk, host_mask, kernel_size);
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.137527 ms	0.446907 ms	0m1.286s	0.86
1000	1.12761 ms	4.59084 ms	0m9.500s	0.886
10000	11.2314 ms	48.7555 ms	1m32.255s	0.8714

Compared with baseline, the first convolution op time gets improved by around 1 ms. And the second convolution op time gets improved by around 7 ms.

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

This optimization can synergize with any optimizations because it simply changes where the mask data is stored.

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

1. ece408-lecture7-convolution-constant-memory-vk-SP24.pdf, https://canvas.illinois.edu/courses/43562/files/folder/lectures

6. **Optimization Number #5: __Tuning with restrict and loop unrolling__**

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

Loop unrolling can speed up the execution time by reducing loop control instructions.

When a compiler can't determine whether pointers alias, it has to assume that they do, which makes the compling process inefficient. Tuning with restrict can prevent the compiler from assuming pointers alias, improving the performance of the code.

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

In conv_forward_kernel, I manually unrolled the inner double loop for traversing mask entries:

```
float acc = 0.0f;
for (int c = 0; c < Channel; c++) {
    acc += in_4d(b, c, h + 0, w + 0) * mask_4d(m, c, 0, 0) + in_4d(b, c, h + 0, w + 1) * mask_4d(m, c, 0, 1) + in_4d(b, c, h + 0, w + 2) *
           in_4d(b, c, h + 1, w + 0) * mask_4d(m, c, 1, 0) + in_4d(b, c, h + 1, w + 1) * mask_4d(m, c, 1, 1) + in_4d(b, c, h + 1, w + 2) *
           in_4d(b, c, h + 2, w + 0) * mask_4d(m, c, 2, 0) + in_4d(b, c, h + 2, w + 1) * mask_4d(m, c, 2, 1) + in_4d(b, c, h + 2, w + 2) *
           in_4d(b, c, h + 3, w + 0) * mask_4d(m, c, 3, 0) + in_4d(b, c, h + 3, w + 1) * mask_4d(m, c, 3, 1) + in_4d(b, c, h + 3, w + 2) *
           in_4d(b, c, h + 4, w + 0) * mask_4d(m, c, 4, 0) + in_4d(b, c, h + 4, w + 1) * mask_4d(m, c, 4, 1) + in_4d(b, c, h + 4, w + 2) *
           in_4d(b, c, h + 5, w + 0) * mask_4d(m, c, 5, 0) + in_4d(b, c, h + 5, w + 1) * mask_4d(m, c, 5, 1) + in_4d(b, c, h + 5, w + 2) *
           in_4d(b, c, h + 6, w + 0) * mask_4d(m, c, 6, 0) + in_4d(b, c, h + 6, w + 1) * mask_4d(m, c, 6, 1) + in_4d(b, c, h + 6, w + 2) *
}
```

I added the keyword "__restrict__" to the parameter data type of conv_forward_kernel:

```
__global__ void conv_forward_kernel(float * __restrict__ output, const float * __restrict__ input, const float * __restrict__ mask,
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.208059 ms	0.468938 ms	0m1.260s	0.86
1000	1.23368 ms	4.6468 ms	0m9.582s	0.886
10000	11.44 ms	48.1193	1m33.051s	0.8714

		ms		

Compared with baseline, the second convolution time decreased by around 7 ms. It shows that the performance of the code does get inproved.

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

This optimization can synergize with stream optimization, tiled convolution, storing mask data in constant memory, and FP16 implementation. That's because the four optimizations all keeps the triple loop in their kernel, which could have been unrolled. Also, in our case, there's no harm to mark parameters as restrict because we always know that no pointers alias.

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

1. Appleyard, Jeremy (2014). CUDA Pro Tip: Optimize for Pointer Aliasing. Nvidia Developer. https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/

7. Optimization Number #9: __Input channel reduction: atomics__

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

Atomics is used when there might be contention, i.e. multiple threads try to write to the same memory address.This technique can prevent inaccurate results caused by contention.

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

I modified the grid configuration and the kernel, so that each threads corresponds to an input entry instead of an output entry. For each input entry (or thread), I loop through all possible locations in the mask (i.e. from (0,0) to (K-1, K-1). If the input location is (h,w), and its multiplying with a mask data entry in (p,q), then we know that it is calculating the output at location (h-p,w-q). In this implementation, each threads participating in calculates multiple output entry, and so each output entry is calculated by adding up the

value from multiple threads. Therefore, contention might exists in this midification. Here is the kernel code for writing to output:

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

if (h < Height && w < Width) {
    for (int c = 0; c < Channel; c++) {
        for (int p = 0; p < K; p++) {
            for (int q = 0; q < K; q++) {
                if (h-p >= 0 && w-q >= 0 && h-p < Height_out && w-q < Width_out) {
                    atomicAdd(&out_4d(b, m, h-p, w-q), in_4d(b, c, h, w) * mask_4d(m, c, p, q));
                }
            }
        }
    }

}
```

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

Batch Size	Op Time 1	Op Time 2	Total Execution Time	Accuracy
100	0.566391 ms	1.86871 ms	0m1.377s	0.86
1000	4.89143 ms	18.0471 ms	0m9.181s	0.882
10000	48.8095 ms	182.196 ms	1m28.955s	0.8705

The reason why the op time is much longer than the baseline could be that having multiple threads to writing to one memory address will damage the parallelism, i.e. threads might have to wait for others finish writing to an address before itself can write.

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

This synergize with stream optimization as mentioned previously. It also synergizes constant memory optimization. Because it will still need to read mask data, for sure. It

also synergize with tiled convolution, there is no conflicts between these two optimization: tiled convolution tries to speed up the reading of input data, while atomics prevent contention by modifying the way kernel write value to output.

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

1. ece408-lecture16-histogram-vk-SP24.pdf, https://canvas.illinois.edu/courses/43562/files/folder/lectures

8. Optimization Number #10: __Fixed point (FP16) arithmetic implementation__

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

In our convolution, 32 bits is more than enough to store the data. We can store each data point with only 16 bits to save space and also speed up the computation because FP16 computation is faster than FP32.

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

When performing arithmetic operation, I cast the data into __half type using __float2half. And I use __hadd and __hmul for arithmetic operation between __half types. Before writing the results to output array, I cast the accumulative sum into FP32 with __half2float since the output should still be in float type.

```
if (h < Height_out && w < Width_out) {
    __half acc = __float2half(0.0f);
    for (int c = 0; c < Channel; c++) {
        for (int p = 0; p < K; p++) {
            for (int q = 0; q < K; q+=1) {
                if (h+p < Height && w+q < Width) {
                    acc = __hadd(acc, __hmul( __float2half(in_4d(b, c, h + p, w + q)),  __float2half(mask_4d(m, c, p, q))));
                }
            }
        }
    }
    out_4d(b, m, h, w) =  __half2float(acc);
}
```

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

Batch	Op Time	Op Time	Total	Accuracy

Size	1	2	Execution Time	
100	0.208631 ms	0.516407 ms	0m1.241s	0.86
1000	1.3053 ms	5.14762 ms	0m9.426s	0.887
10000	12.2794 ms	54.3637 ms	1m33.199s	0.8716

One reason this optimization does not improve the performance could be that the convolution operation is compute-bound rather than memory-bound, meaning that the number of arithmetic operations per memory access is high, switching to __half may not improve performance significantly.

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

This optimization can synergize with any optimizations because it only affects the space needed to store data, and how the arithmatic computation is performed.

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

1. Cuda Toolkit, https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__ARITHMETIC.html#group__CUDA__MATH____HALF__ARITHMETIC