Hongyi Li

Milestone 3 Report

0. Baseline (Milestone 2):

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.231835*	*0.384682*	*0.616517*	*3.51*	*2.81*	*6.32*	*0.86*
1000	*1.54232*	*3.71468*	*5.257*	*20.9*	*19.2*	*40.1*	*0.886*
10000	*14.5131*	*36.8421*	*51.3552*	*199*	*188*	*387*	*0.8714*

Table 1: Performance statistics of Milestone 2

CUDA API	Total Time (ms)
cudaMemcpy	*294.8*
cudaMalloc	*198.9*
cudaFree	*9.9*
Total	*503.6*

Table 2: Running time of CUDA APIs in Milestone 2

1. Optimization Number #: 4 Optimization Name: Weight matrix (Kernel) in constant memory

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

The algorithm pseudo-code for the two convolution layers is as below.

```
for b = 0 .. Batch                    // for each image in the batch
    for m = 0 .. Map_out              // for each output feature maps
        for h = 0 .. Height_out       // for each output element
            for w = 0 .. Width_out
            {
                output[b][m][h][w] = 0;
                for c = 0 .. Channel   // sum over all input feature maps
                    for p = 0 .. K // KxK filter
                        for q = 0 .. K
                            output[b][m][h][w] += input[b][c][h + p][w + q] * k[m][c][p][q]
            }
```

Figure 1: Pseudo-code for a convolution layer

As we can see, we need to do two memory reads, at every time when we increment an output element. As Professor Lumetta shows on Page 15 of Slide Deck 8, each global memory read takes about 500 cycles, while each constant memory read takes about 5 cycles. In Milestone 2, we get both the input element and the kernel weight from the global memory, which takes about 1,000 cycles in total. Instead, if we store the kernel weights in constant memory, reading the input element and the kernel weight can take around 505 cycles in total. In this way, the time consumed by memory reads is expected to be approximately halved.

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

As Page 22 of Slide Deck 8, we can first establish the constant memory space for the kernels of the two layers, respectively.

```
7   // Define constant memory for device kernel
8   #define Map_out_L1 4
9   #define Channel_L1 1
10  #define Map_out_L2 16
11  #define Channel_L2 4
12  #define MASK_WIDTH 7
13  static __constant__ float Mc1[Map_out_L1][Channel_L1][MASK_WIDTH][MASK_WIDTH];
14  static __constant__ float Mc2[Map_out_L2][Channel_L2][MASK_WIDTH][MASK_WIDTH];
```

Figure 2: Weight matrices in constant memory

Before launching the convolution kernel, we need to copy the weight matrix to the corresponding constant memory space, as shown below.

```
102         // Copy Layer 1 or 2's kernel to constant memory
103         int masksize = Map_out * Channel * K * K * sizeof(float);
104         if (Channel > 1) {
105           cudaMemcpyToSymbol(Mc2, host_mask, masksize);
106         }
107         else {
108           cudaMemcpyToSymbol(Mc1, host_mask, masksize);
109         }
```

Figure 3: Copying weight matrix to constant memory

Eventually, when doing calculations in the GPU, we need to get the kernel weights from the constant memory as below.

```
59          float r = 0.0;
60
61          for (int c = 0; c < Channel; c++) {
62            for (int p = 0; p < K; p++) {
63              for (int q = 0; q < K; q++) {
64                float k = (Channel > 1) ? Mc2[m][c][p][q] : Mc1[m][c][p][q];
65                r += (in_4d(b, c, h+p, w+q) * k);
66              }
67            }
68          }
69
70          out_4d(b, m, h, w) = r;
```

Figure 4: Inner loop to calculate an output element

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

Compared to Milestone 2, this optimization will change how CUDA APIs are called as well. It's obvious that this optimization calls cudaMemcpyToSymbol two more times. Correspondingly, each of cudaMemcpy, cudaMalloc and cudaFree is called two fewer times, as the convolution kernels are no longer stored in the global memory. The running time statistics of these CUDA APIs are shown in Table 3 below.

CUDA API	Total Time (ms)
cudaMemcpy	*303.2*
cudaMalloc	*152.0*
cudaFree	*5.9*
cudaMemcpyToSymbol	*0.3*
Total	*461.4*

Table 3: Running time of CUDA APIs in Op #4

By comparing Tables 2 and 3, we can see this optimzation brings down the total running time of these CUDA APIs, by 8.4%. The performance of the two layers is displayed in Table 4 below.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.162224*	*0.32605*	*0.488274*	*3.06*	*2.89*	*5.95*	*0.86*
1000	*1.31219*	*3.29556*	*4.60775*	*21.6*	*19.8*	*41.4*	*0.886*
10000	*12.7247*	*32.6812*	*45.4059*	*188*	*162*	*350*	*0.8714*

Table 4: Performance statistics of Op #4

By comparing Tables 4 and 1, we can see this optimization brings down the operation time by 11.58%.

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

This is the first optimization I implement, and it's directly added to Milestone 2.

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

I mainly rely on Slide Deck 8 to implement this optimization.

2. Optimization Number #: 1 Optimization Name: Tiled (shared memory) convolution

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

For this optimization, I set TILE_WIDTH to be 17, and thus each block contains (TILE_WIDTH + MASK_WIDTH - 1) x (TILE_WIDTH + MASK_WIDTH - 1) = 23 x 23 = 529 threads. Like Page 25 of Slide Deck 9, each thread will load input elements

into shared memory, and TILE_WIDTH x TILE_WIDTH = 289 threads will compute output elements.

To maximize input element reuses, I decide to use one thread block to process four output tile squares, as below.

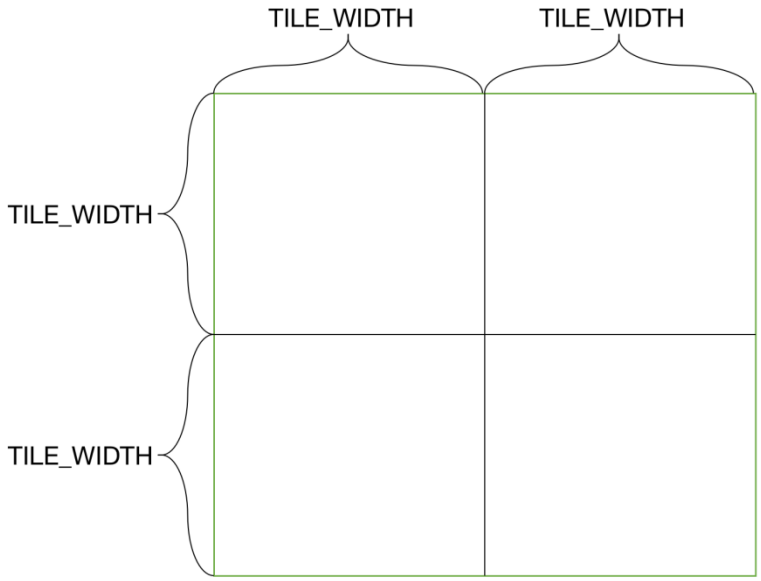


Figure 5: Tiled convolution output elements

The threads in each block will collaboratively load (2*TILE_WIDTH + MASK_WIDTH - 1) x (2*TILE_WIDTH + MASK_WIDTH - 1) input elements into shared memory. The specific implementation details will be discussed later.

We can then calculate the bandwidth reduction as $\frac{(2\times\text{TILE\_WIDTH})^2\times\text{MASK\_WIDTH}^2}{(2\times\text{TILE\_WIDTH} + \text{MASK\_WIDTH} - 1)^2} = 35.4025$. As a result, running time of the two convolution layers can be saved from fewer global memory reads.

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

We can design our kernel according to Page 21 of Slide Deck 18, as shown in Figure 6. This design is realized by the code in Figure 7.

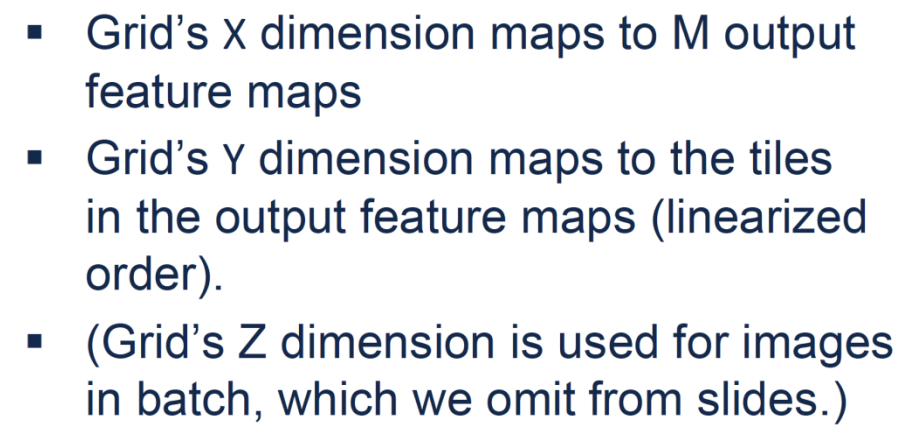
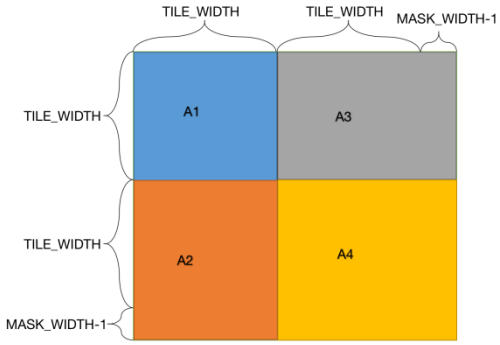


Figure 6: Tiled convolution kernel design for CNN

```
169         int W_grid = ceil(Width_out / (2.0 * TILE_WIDTH));
170         int H_grid = ceil(Height_out / (2.0 * TILE_WIDTH));
171         int Y = H_grid * W_grid;
172
173         // Initialize grid and block dimensions here
174         dim3 DimGrid(Map_out, Y, Batch);
175         dim3 DimBlock(TILE_WIDTH + K - 1, TILE_WIDTH + K - 1, 1);
176
177         conv_forward_kernel<<<DimGrid,DimBlock>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width, K);
178         cudaDeviceSynchronize();
```

Figure 7: Tiled convolution kernel setup code

As for loading the input elements into the shared memory, all threads in each block collaboratively load the four subareas in the input image (Figure 8a) for each input channel, with the code snippet in Figure 8b. Especially, we need a __syncthreads() call at the end of the loading process to make sure all input elements are in place and ready for computations.



(a)

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

  if (tx < TILE_WIDTH) {
    if (h + TILE_WIDTH < Height && w < Width) {
      tile[c][ty+TILE_WIDTH][tx] = in_4d(b, c, h+TILE_WIDTH, w);
    }
    else {
      tile[c][ty+TILE_WIDTH][tx] = 0.0;
    }
  }

  if (ty < TILE_WIDTH) {
    if (h < Height && w + TILE_WIDTH < Width) {
      tile[c][ty][tx+TILE_WIDTH] = in_4d(b, c, h, w+TILE_WIDTH);
    }
    else {
      tile[c][ty][tx+TILE_WIDTH] = 0.0;
    }
  }

  if (h + TILE_WIDTH < Height && w + TILE_WIDTH < Width) {
    tile[c][ty+TILE_WIDTH][tx+TILE_WIDTH] = in_4d(b, c, h+TILE_WIDTH, w+TILE_WIDTH);
  }
  else {
    tile[c][ty+TILE_WIDTH][tx+TILE_WIDTH] = 0.0;
  }
}

__syncthreads();
```

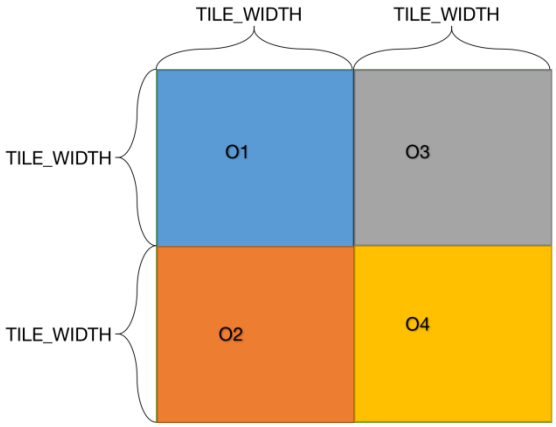
(b)

Figure 8: Tiled convolution kernel shared memory. (a): General scheme to load input elements. (b): Code snippet to load input elements.

By doing a device query, we can see the maximum shared memory size per block is 49,152 bytes. The first convolutional layer has one input channel, while the second layer has four. Hence, in our design, the size of shared memory for each block is:

$$4 \text{ channels} \times (2 \times \text{TILE\_WIDTH} + \text{MASK\_WIDTH} - 1)^2 \text{ floats/channel} \times 4 \text{ bytes/float} = 25600 \text{ bytes}$$

As we can see, this is under the device limit. For the output computation, there are TILE_WIDTH^2 threads doing this. As shown in Figure 9, these threads will calculate the four subareas in the output image respectively. That is to say, each of these computing threads will handle four output elements.




```
97        if (ty < TILE_WIDTH && tx < TILE_WIDTH) {
98          float r1 = 0.0;
99          float r2 = 0.0;
100         float r3 = 0.0;
101         float r4 = 0.0;
102
103         for (int c = 0; c < Channel; c++) {
104           for (int p = 0; p < K; p++) {
105             for (int q = 0; q < K; q++) {
106               r1 += tile[c][ty+p][tx+q] * mask_4d(m, c, p, q);
107                     r2 += tile[c][ty+p][tx+q+TILE_WIDTH] * mask_4d(m, c, p, q);
108               r3 += tile[c][ty+p+TILE_WIDTH][tx+q] * mask_4d(m, c, p, q);
109               r4 += tile[c][ty+p+TILE_WIDTH][tx+q+TILE_WIDTH] * mask_4d(m, c, p, q);
110             }
111           }
112         }
113
114         if (h < Height_out && w < Width_out) {
115           out_4d(b, m, h, w) = r1;
116         }
117         if (h < Height_out && w+TILE_WIDTH < Width_out) {
118           out_4d(b, m, h, w+TILE_WIDTH) = r2;
119         }
120         if (h+TILE_WIDTH < Height_out && w < Width_out) {
121           out_4d(b, m, h+TILE_WIDTH, w) = r3;
122         }
123         if (h+TILE_WIDTH < Height_out && w+TILE_WIDTH < Width_out) {
124           out_4d(b, m, h+TILE_WIDTH, w+TILE_WIDTH) = r4;
125         }
126       }
```

(a) (b)

Figure 9: Tiled convolution kernel output computation. (a): General scheme to compute output elements. (b): Code snippet to compute output elements.

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

By comparing Tables 5 and 1, we can see this optimization brings down the total execution time by 0.7%-0.9%, compared to Milestone 2.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.270908*	*0.348624*	*0.619532*	*3.31*	*3.55*	*6.86*	*0.86*
1000	*2.02356*	*3.18368*	*5.20724*	*21.4*	*19.2*	*40.6*	*0.886*
10000	*19.5404*	*31.4202*	*50.9606*	*194*	*156*	*350*	*0.8714*

Table 5: Performance statistics of Op #1

After getting detailed performance statistics from Nsight-Compute, we can make the chart below, for a comparation between this optimization and Milestone 2.

Metric	Milestone 2	Optimization #1

Layer 1 Duration (ms)	*19.04*	*25.46*
Layer 1 Throughput (%)	*76.77*	*87.97*
Layer 2 Duration (ms)	*47.28*	*41.67*
Layer 2 Throughput (%)	*87.98*	*94.43*

Table 6: Comparisons between Milestone 2 and Optimization #1, for batch size 10,000

From Table 6, we can see that Optimization 1 improves the throughput for both layers, by bandwidth reduction. In Milestone 2, each block has 1,024 threads and each thread handles a unique output element. In this way, warps don't have control divergence. In contrast, for Optimization #1, the if statements in input loading and output computation can result in lots of control divergence. This can counteract some runtime advantages from tiled convolution.

From Table 6, we can also see Optimization #1 actually results in a longer runtime for Layer 1, and a shorter runtime for Layer 2. Hence, larger input datasets tend to benefit more from tiled convolution. This insight may be helpful for designing different kernels for different layer sizes.

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

This optimization can synergize with Optimization #4, which puts the convolution matrices into constant memory. As a result, fewer global memory reads will be done. The performance statistics of this combination is in Table 7 below. By comparing this with Table 5, we can see the running time goes down by about 15%.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.220764*	*0.2882*	*0.508964*	*3.08*	*2.75*	*5.83*	*0.86*
1000	*1.813*	*2.57455*	*4.38755*	*22.6*	*732*	*755*	*0.886*
10000	*17.6603*	*25.4611*	*43.1214*	*183*	*151*	*334*	*0.8714*

Table 7: Performance statistics of Op #1+#4

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

I mainly rely on Slide Deck 9 and 18 for implementations, and Slide Deck 10 for theoretical analysis.

3. **Optimization Number #: 2 Optimization Name: Input matrix unrolling & tiled matrix multiplication using shared memory**

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

In the original convolution algorithm, we need to do $C \cdot K^2$ multiplications to compute every output element. This process usually involves double for loops. To reduce latency from this process, we can rearrange input elements, and thus turn the convolution into a matrix-vector multiplication. Once this is achieved, we can then use the shared memory and tiled matrix multiplication techniques for acceleration. Specifically, we set TILE_WIDTH to be 16 in our implementation. Therefore, each input element/convolution weight is reused 16 times.

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

First, we need to rearrange the input elements in the way shown by Page 37 and 38 of Slide Deck 18. To do so, we set up a kernel, in which each thread handles one output element and its associated $C \cdot K^2$ input elements, as shown in Figure 10 below.

```
__global__ void unroll_kernel(float *output, const float *input, const int Batch,
                              const int Channel, const int Height, const int Width,
                              const int K, const int start)
{
  const int Height_out = Height - K + 1;
  const int Width_out = Width - K + 1;

  int w = (blockIdx.x * blockDim.x) + threadIdx.x;

  #define in_4d(i3, i2, i1, i0) input[(i3) * (Channel * Height * Width) + (i2) * (Height * Width) + (i1) * (Width) + i0]
  #define unroll_3d(i2, i1, i0) output[(i2) * (Channel * K * K * Height_out * Width_out) + (i1) * (Height_out * Width_out) + i0]

  if (w < Batch * Height_out * Width_out) {
    int b = w / (Height_out * Width_out);
    w -= (b * Height_out * Width_out);
    int h = w / Width_out;
    w -= (h * Width_out);

    for (int c = 0; c < Channel; c++) {
      for (int p = 0; p < K; p++) {
        for (int q = 0; q < K; q++) {
          int col = (h * Width_out) + w;
          int row = (c * K * K) + (p * K) + q;
          unroll_3d(b, row, col) = in_4d(start+b, c, h + p, w + q);
        }
      }
    }
  }

  #undef in_4d
  #undef unroll_3d
}
```

Figure 10: Kernel for input matrix unrolling

After the unrolling kernel, we launch another kernel to carry out the tiled matrix multiplication in the shared memory. When the input data size becomes very large, like a batch size of 10,000, our GPU device may not have enough memory space to hold the unrolled input matrix. To resolve this issue, we can divide the whole input dataset into mini batches of 1,000. That is to say, we unroll the input matrix for the first 1,000 images, carry out tiled matrix multiplication, and then do the same things for the next 1,000 images until all images are processed. This is realized by the for loop in Figure 11.

```
      for (int i = 0; i < ceil(Batch/1000.0); i++) {
        int start = i * 1000;
        int mini_b = min(1000, Batch - start);

        // Initialize grid and block dimensions here
        int numPixels = mini_b * Height_out * Width_out;
        unroll_kernel<<<ceil(numPixels/96.0),96>>>(unroll, device_input, Batch, Channel, Height, Width, K, start);
        cudaDeviceSynchronize();

        dim3 DimGrid(ceil(Height_out * Width_out * 1.0 / TILE_WIDTH),
                     ceil(Map_out * 1.0 / TILE_WIDTH),
                     mini_b);
        dim3 DimBlock(TILE_WIDTH, TILE_WIDTH, 1);
        matrixMultiplyShared<<<DimGrid,DimBlock>>>(device_output, unroll, Batch,
                                                   Map_out, Channel, Height, Width,
                                                   K, start);
        cudaDeviceSynchronize();
      }
```

Figure 11: Program flow of Optimization #2

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

The performance statitstics of this optimization is in Table 8 below. As we can see, the program uses longer execution time than other optimizations. This meets my expectations as this optimization needs to copy much more data to global memory and launch two separate kernels for each mini batch.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*1.61619*	*1.44808*	*3.06427*	*5.40*	*5.51*	*10.9*	*0.86*
1000	*8.98411*	*8.87045*	*17.85456*	*29.6*	*22.9*	*52.5*	*0.886*
10000	*81.8882*	*82.3085*	*164.1967*	*256*	*219*	*475*	*0.8714*

Table 8: Performance statistics of Op #2+#4

Furthermore, by looking at the profiling result from Nsight System, we can see that the input matrix unrolling kernel takes up 35.5% of the CUDA hardware running time, while the tiled matrix multiplication kernel takes up the rest. This may indicate a performance boost if we fuse these two kernels and avoid massive writing to the global memory for the unrolled input matrix.

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

This optimization can synergize with Optimization 4, and thus store the convolution weights in the constant memory. In fact, our implementation already includes this combination.

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

I mainly rely on Slide Deck 18 to implement this technique.

4. **Optimization Number #: 3 Optimization Name: Kernel fusion for unrolling and matrix-multiplication (requires op #2)**

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

This optimization starts from Optimization 2. Instead of creating and launching a separate kernel just to unroll the input matrix, we'll have the tiled matrix multiplication kernel do the unrolling work in-place. That is to say, we won't allocate any global memory space for the unrolled matrix, which only exists conceptually in this case. As a result, some execution time can be saved from loading the unrolled matrix into the global memory.

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

The original unrolled matrix has $C \cdot K^2$ rows. If we fuse the unrolling kernel into the matrix multiplication kernel, we then need to do some indexing work to load the correct input element into the shared memory for the tile. This can be done by the code snippet in Figure 12. We also need a __syncthreads() call at the end to make sure all elements are in place before computations start.

```
50        if (subRow < inputSize && Col < outputSize) {
51          int c = subRow / (K * K);
52          int p = subRow % (K * K);
53          int q = p % K;
54          p /= K;
55          int h = Col / Width_out;
56          int w = Col % Width_out;
57          subTileB[ty][tx] = in_4d(b, c, h+p, w+q);
58        }
59        else {subTileB[ty][tx] = 0.0;}
60
61        __syncthreads();
```

Figure 12: Conceptually load the unrolled input matrix element

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

The performance statistics is in Table 9 below. By comparing this with Table 8, we can see that the kernel fusion action brings down the execution time by about 37.5%.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.593912*	*0.515806*	*1.109718*	*3.48*	*3.09*	*6.57*	*0.86*
1000	*5.64658*	*4.96938*	*10.61596*	*24.8*	*19.9*	*44.7*	*0.886*
10000	*55.6937*	*49.4612*	*105.1549*	*229*	*177*	*406*	*0.8714*

Table 9: Performance statistics of Op #2+#3+#4

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

The implementation of this technique starts from Optimization #2, and stores the convolution weights in the constant memory by Optimization #4.

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

I mainly rely on Page 42 of Slide Deck 18 to implement this technique.

5. Optimization Number #: 9 Optimization Name: Input channel reduction -- atomics

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

In our Milestone 2 implementation, we assign each output element a thread, which goes through every input element in every input channel. This results in a three-level for loop shown in Figure 13.

```
49          float r = 0.0;
50
51          for (int c = 0; c < Channel; c++) {
52            for (int p = 0; p < K; p++) {
53              for (int q = 0; q < K; q++) {
54                r += (in_4d(b, c, h+p, w+q) * mask_4d(m, c, p, q));
55              }
56            }
57          }
58
59          out_4d(b, m, h, w) = r;
```

Figure 13: Thread-level implementation in Milestone 2

Of course, this code structure can increase the execution time of each single thread. In fact, we can also utilize parallelism on input channels, and thus spread the work across more threads. Theoretically, this can reduce the single thread execution time to $\frac{1}{Channel}$ of that in Milestone 2.

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

In Milestone 2, our grids and blocks are just one-dimension that corresponds to every output element in every output channel. For this optimization, we set up two-dimension blocks, where the Y dimension corresponds to the input channels.

```
// Set the kernel dimensions and call the kernel
const int Height_out = Height - K + 1;
const int Width_out = Width - K + 1;
int numPixels = Batch * Map_out * Height_out * Width_out;

conv_forward_kernel<<<ceil(numPixels/1024.0),1024>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width, K);
cudaDeviceSynchronize();
```

(a)

```
int t = 96 / Channel;

dim3 DimGrid(ceil(numPixels * 1.0 / t), 1, 1);
dim3 DimBlock(t, Channel, 1);

conv_forward_kernel<<<DimGrid,DimBlock>>>(device_output, device_input, device_mask, Batch, Map_out, Channel, Height, Width, K);
cudaDeviceSynchronize();
```

(b)

Figure 14: Kernel setup comparisons. (a): Milestone 2 (b): Optimization #9 + #4

By doing a device query, we find out each SM can hold at most 16 blocks and 1,536 threads. Therefore, the optimal number of threads in a block is $\frac{1536}{16} = 96$. The computation of each output element requires *Channel* threads, each of which computes for one input channel. Hence, we can set up our blocks as (96/*Channel*, *Channel,* 1).

In Figure 15a, the program will go through each pixel in the corresponding input channel, and use atomicAdd to accumulate the result for the output elements, as other threads may be computing for the same output element. In addition, we need to set all output to be zero so that the accumulations can yield correct results (Figure 15b).

```
60          float r = 0.0;
61
62          for (int p = 0; p < K; p++) {
63            for (int q = 0; q < K; q++) {
64              float k = (Channel > 1) ? Mc2[m][c][p][q] : Mc1[m][c][p][q];
65              r += (in_4d(b, c, h+p, w+q) * k);
66            }
67          }
68
69          atomicAdd( &out_4d(b, m, h, w), r );
```

(a)

```
91          const int Height_out = Height - K + 1;
92          const int Width_out = Width - K + 1;
93          int outputsize = Batch * Map_out * Height_out * Width_out * sizeof(float);
94          cudaMalloc((void **) device_output_ptr, outputsize);
95          cudaMemset(device_output_ptr, 0, outputsize);
```

(b)

Figure 15: Optimization #9 details. (a): Thread computation (b): Output memory setup

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

The performance statistics is in Table 10 below. By comparing Table 10 with Table 4, we can see the parallelism on input channels and associated atomicAdd calls still add some latency to memory access and executioin time.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.127289*	*1.16571*	*1.292999*	*2.97*	*6.57*	*9.54*	*0.86*
1000	*0.947316*	*12.2314*	*13.178716*	*21.4*	*30.7*	*52.1*	*0.886*
10000	*9.10043*	*120.794*	*129.89443*	*182*	*250*	*432*	*0.8714*

Table 10: Performance statistics of Op #9+#4

By looking at the profiling results from Nsight Compute, we can get the comparison results in Table 11 below. For Layer 1 with fewer input channels, the parallelism on input channels actually decreases the execution time, without affecting the memory throughput too much. In

contrast, the memory throughput is largely decreased by the aforementioned latency in Layer 2, which contains much more input channels.

Metric	Milestone 2	Optimization #9
Layer 1 Duration (ms)	*19.04*	*12.02*
Layer 1 Memory Throughput (%)	*76.77*	*63.17*
Layer 2 Duration (ms)	*47.28*	*160.81*
Layer 2 Memory Throughput (%)	*87.98*	*14.69*

Table 11: Comparisons between Milestone 2 and Optimization #9, for batch size 10,000

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

The implementation of this technique already includes Optimization #4. We can also combine this technique with tiled convolution (Optimization #1). As shown by Figure 16a, each block is 2D and corresponds to an output tile, in Optimization #1. To combine Optimization #9 with Optimization #1, we can make each block 3D, where the third dimension corresponds to the input channels (Figure 16b).

```
174        // Set the kernel dimensions and call the kernel
175        const int Height_out = Height - K + 1;
176        const int Width_out = Width - K + 1;
177
178        int W_grid = ceil(Width_out / (2.0 * TILE_WIDTH));
179        int H_grid = ceil(Height_out / (2.0 * TILE_WIDTH));
180        int Y = H_grid * W_grid;
181
182        // Initialize grid and block dimensions here
183        dim3 DimGrid(Map_out, Y, Batch);
184        dim3 DimBlock(TILE_WIDTH + K - 1, TILE_WIDTH + K - 1, 1);
```

(a)

```
170        // Set the kernel dimensions and call the kernel
171        const int Height_out = Height - K + 1;
172        const int Width_out = Width - K + 1;
173
174        int W_grid = ceil(Width_out / (2.0 * TILE_WIDTH));
175        int H_grid = ceil(Height_out / (2.0 * TILE_WIDTH));
176        int Y = H_grid * W_grid;
177
178        // Initialize grid and block dimensions here
179        dim3 DimGrid(Map_out, Y, Batch);
180        dim3 DimBlock(TILE_WIDTH + K - 1, TILE_WIDTH + K - 1, Channel);
```

(b)

Figure 16: Kernel setup comparisons. (a): Optimization #1+#4 (b): Optimization #1+#4+#9

Theoretically, this combination is possible, as the addition of input channels into the output is associative and commutative. Instead of directly assigning to the output element, we again need to use atomicAdd calls to add the result from a specific input channel into the final output (Figure 17). The performance statistics of this combination is in Table 12 below. By comparing Table 12 with Table 7, we can again see the input channel parallelism creates some latency.

```
118         if (h < Height_out && w < Width_out) {
119           out_4d(b, m, h, w) = r1;
120         }
121         if (h < Height_out && w+TILE_WIDTH < Width_out) {
122           out_4d(b, m, h, w+TILE_WIDTH) = r2;
123         }
124         if (h+TILE_WIDTH < Height_out && w < Width_out) {
125           out_4d(b, m, h+TILE_WIDTH, w) = r3;
126         }
127         if (h+TILE_WIDTH < Height_out && w+TILE_WIDTH < Width_out) {
128           out_4d(b, m, h+TILE_WIDTH, w+TILE_WIDTH) = r4;
129         }
```

(a)

```
115         if (h < Height_out && w < Width_out) {
116           atomicAdd( &out_4d(b, m, h, w), r1 );
117         }
118         if (h < Height_out && w+TILE_WIDTH < Width_out) {
119           atomicAdd( &out_4d(b, m, h, w+TILE_WIDTH), r2 );
120         }
121         if (h+TILE_WIDTH < Height_out && w < Width_out) {
122           atomicAdd( &out_4d(b, m, h+TILE_WIDTH, w), r3 );
123         }
124         if (h+TILE_WIDTH < Height_out && w+TILE_WIDTH < Width_out) {
125           atomicAdd( &out_4d(b, m, h+TILE_WIDTH, w+TILE_WIDTH), r4 );
126         }
```

(b)

Figure 17: Output storage comparisons. (a): Optimization #1+#4 (b): Optimization #1+#4+#9

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.144541*	*0.820568*	*0.965109*	*2.99*	*3.59*	*6.58*	*0.86*
1000	*1.07141*	*8.01211*	*9.08352*	*19.6*	*22.8*	*42.4*	*0.886*
10000	*10.2218*	*79.8767*	*90.0985*	*187*	*213*	*400*	*0.8714*

Table 12: Performance statistics of Op #1+#4+#9

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

I use Slide Deck 13 to add atomic operations to my code.

6. Optimization Number #: 12 Optimization Name: An advanced matrix multiplication algorithm -- joint register-shared memory tiling

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

On CUDA devices, registers can offer low latency and high throughput, but are private to each thread. The shared memory can offer comparable latency and lower throughput, and are visible to all threads in a block. In the original matrix multiplication algorithm, inputs from both matrices are loaded into the shared memory. Nevertheless, as discussed in Professor Lumetta's 508 slide, the full tile is not necessarily needed all the time when the input elements are reused. Hence, this can be the space to save more on-chip memory. Instead of loading all input tiles into

the shared memory, we can save some of them in per-thread registers. In this way, every block can use less on-chip memory, and thus may help boost the overall performance.

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

Borrowing ideas from Page 36 of Professor Lumetta's 508 slide, we can set $T = 128$, $U = 16$, and $S = 8$. The meanings of these varibales and the overall layout of this algorithm are shown in Figure 18 below.

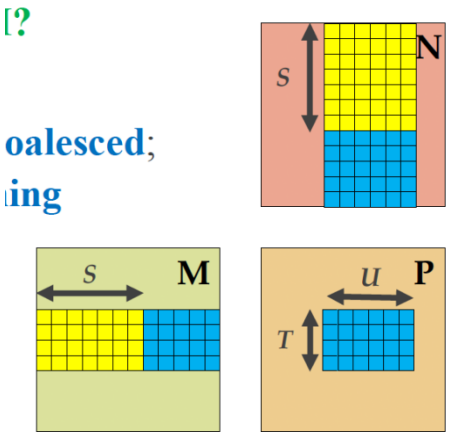


Figure 18: Layout of joint register-shared memory tiling. **M** and **N** are input matrices, and **P** is the output matrix.

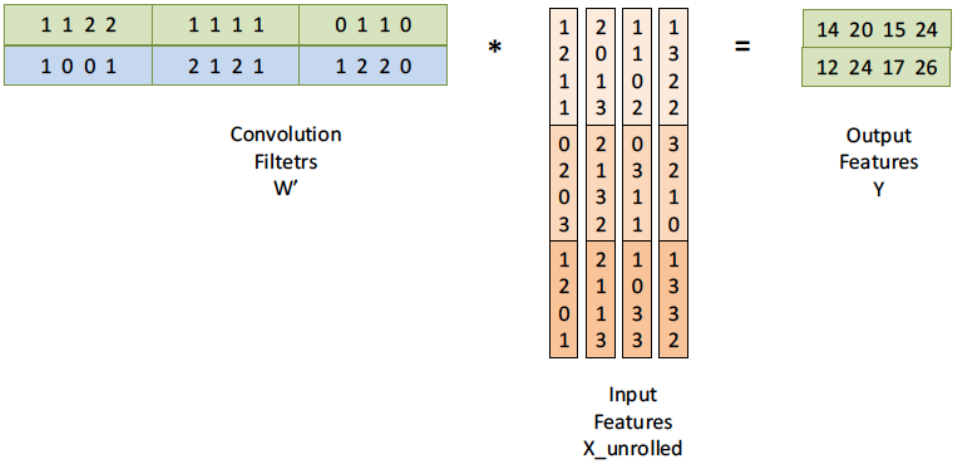


Figure 19: Unrolled input matrix illustration.

As displayed in Figure 19, in the original unrolled matrix multiplication algorithm, the convolution filters W' correspond to **M** in Figure 18, while the input features $X_{unrolled}$ correspond to **N**. Each row of W' corresponds to an output channel, and our Layer 2 has four output channels. With $\mathsf{T} = 128$, lots of threads can be idle, which creates work imbalance. To resolve this issue, we can transpose both input matrices, and thus calculate the transposed version of the output matrix. In fact, the transposing work happens conceptually, and we can rely on careful indexing work to load correct elements from the two input matrices.

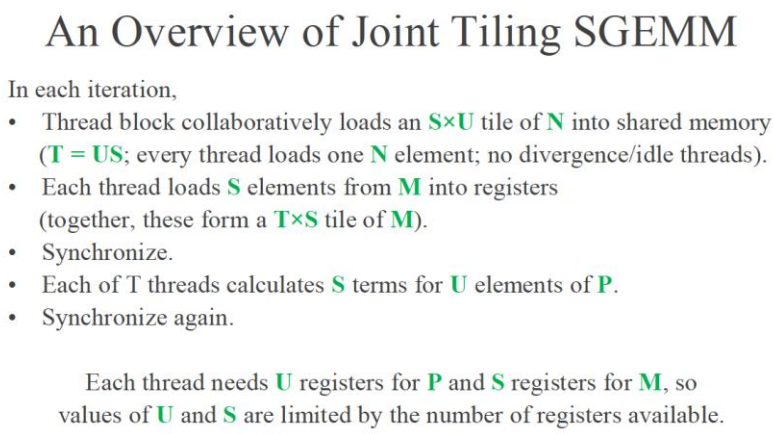


Figure 20: Overall flow of Optimization #12.

In our implementation, each block has T threads, and the overall flow of the joint tiling algorithm is described in Figure 20 above. Specifically, Figure 21 shows how each of the T threads handles the elements from **M** and **N**.

```
44      int subCol = Col + (tx % U);
45      int subRow = (m * S) + (tx / U);
46
47      if (subCol < Map_out && subRow < inputSize) {
48        subTileN[tx / U][tx % U] = (Channel > 1) ? Mc2[subCol][subRow] : Mc1[subCol][subRow];
49      }
50      else {subTileN[tx / U][tx % U] = 0.0;}
51
52      __syncthreads();
```

(a)

```
54      for (int s = 0; s < S; s++) {
55        subRow = Row + tx;
56        subCol = (m * S) + s;
57
58        float cw;
59
60        if (subRow < outputSize && subCol < inputSize) {
61          int c = subCol / (K * K);
62          int p = subCol % (K * K);
63          int q = p % K;
64          p /= K;
65          int h = subRow / Width_out;
66          int w = subRow % Width_out;
67          cw = in_4d(b, c, h+p, w+q);
68        }
69        else {cw = 0.0;}
70
71        for (int i = 0; i < U; i++) {Pvalue[i] += cw * subTileN[s][i];}
72      }
73
74      __syncthreads();
```

(b)

Figure 21: Code snippets in Optimization #12. (a): Collaboratively loading **N** elements (b): Each thread loads S elements from **M**

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

The performance of this technique is in Table 13 below. By comparing it with Table 9, we can see the total execution time is cut down by over 50%.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.294722*	*0.26147*	*0.556192*	*3.31*	*3.02*	*6.33*	*0.86*
1000	*2.53571*	*2.05095*	*4.58666*	*21.1*	*16.1*	*37.2*	*0.886*
10000	*24.9103*	*19.9673*	*44.8776*	*190*	*146*	*336*	*0.8714*

Table 13: Performance statistics of Op #2+#3+#4+#12

By comparing the profiling results from Nsight Compute, we can get Table 14 below. As we can see, each thread in Optimization #12 uses slightly more registers, but much less shared memory in terms of the whole block. As a result, Optimization #12 has larger block limits.

Metric	Optimization #3	Optimization #12
Registers per Thread	*37*	*48*
Shared Memory per Block	*2.05 Kbyte*	*512 byte*
Block Limit by Registers	*6*	*10*
Block Limit by Shared Memory	*10*	*21*

Table 14: Comparisons between Optimization #3 and Optimization #12, for batch size 10,000

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

This joint tiling technique aims to accelerate matrix multiplications. Therefore, the implementation of Optimization #12 is added on top of Optimizations #2, #3 and #4.

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

I use Professor Lumetta's 508 slide (posted on Piazza) to implement this technique.

7. Optimization Number #: 0 Optimization Name: Using Streams to overlap computation with data transfer (required)

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

In our previous implementations, we always follow the procedures, in which we first load the input data from CPU to GPU, launch kernels on GPU, and then copy the output data from GPU back to CPU. As a result, when we transfer data into and out of GPU, the actual GPU devices are idle. Similarly, when we carry out computations, the PCIe links, which transfer data between CPU and GPU, are idle as well. Furthermore, we're actually only using one direction of the PCIe links during either of the two transferring events. Therefore, it makes sense for us to make use of CUDA streams, which are more advanced features of the CUDA APIs. This can allow us to overlap data transferring and kernel launching, and thus make many of such events happen concurrently. In this way, the overall execution time can be reduced.

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

We set up the CUDA streams based on the architecture shown in Figure 22 below. We first divide the input data into several segments by images. In our implementation, every 60 input images form a segment, and each segment will be assigned to a stream. We have three streams in total. For the majority of the program, there are always one stream loading its input image data into GPU, one stream doing computations on GPU, and one stream loading its computed output image data into CPU. Once a stream finishes porting its output data, it will go ahead to handle the next segment of input data.

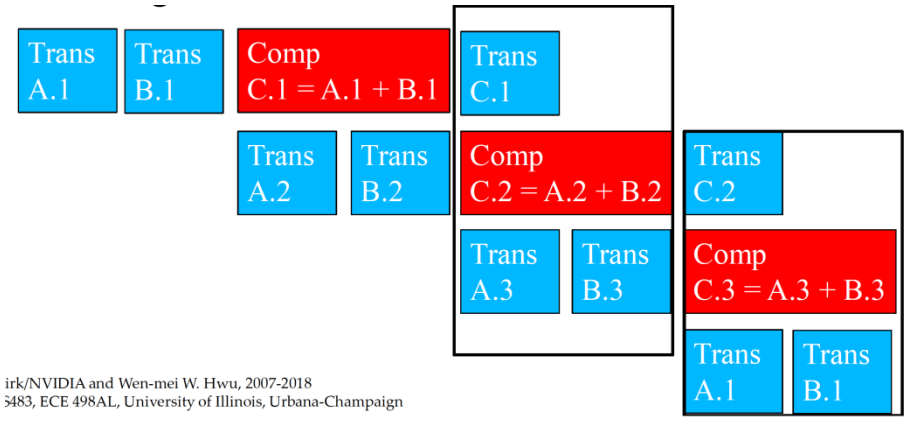


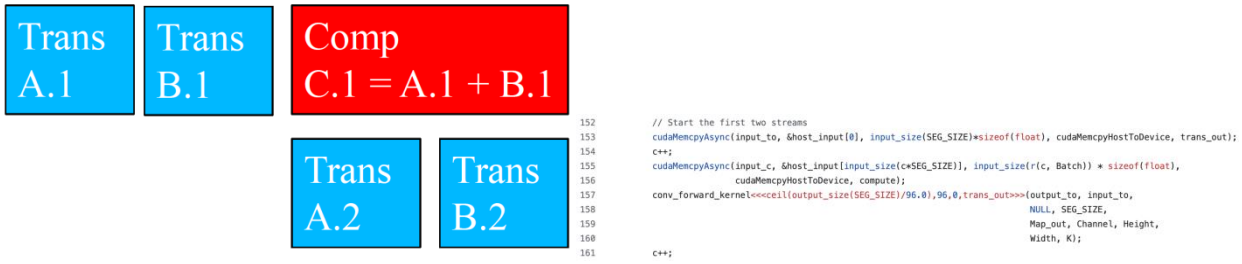
Figure 22: Overall structure of CUDA streams

If all of the input images are not enough to form a segment, we can then handle it straightforwardly and without parallelism, as shown in Figure 23 below.

```
143         // When the dataset is small
144         cudaMemcpyAsync(input_to, &host_input[0], input_size(Batch)*sizeof(float), cudaMemcpyHostToDevice, trans_out);
145         conv_forward_kernel<<<ceil(output_size(Batch)/96.0),96,0,trans_out>>>(output_to, input_to,
146                                                                               NULL, Batch,
147                                                                               Map_out, Channel, Height,
148                                                                               Width, K);
149         cudaMemcpyAsync(((void*) &host_output[0]), output_to, output_size(Batch)*sizeof(float), cudaMemcpyDeviceToHost, trans_out);
```

Figure 23: Code to put all input data into a single segment

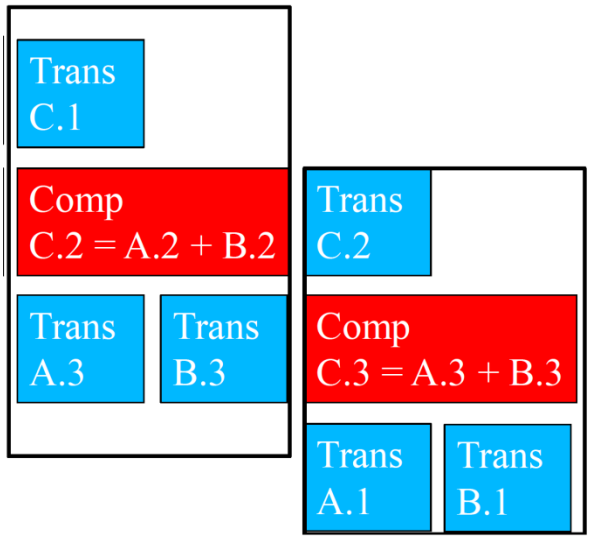
Otherwise, we need to set up multiple streams. As shown in Figure 24a, we need to first initiate two streams to handle the first two input data segments, and this can be done by the code in Figure 24b.



(a) (b)

Figure 24: Start the first two streams. (a): Overlapping scheme (b): Code implementation

Then, we need to overlap three streams to handle the majority of the input data segments, as shown in Figure 25a. To do so, we implement the code in Figure 25b. Specifically, in each round, we always have one stream loading its input data, one stream doing computations, and one stream porting its output data. From Figure 25a, in the next overlapping block, the original computation stream becomes the transferring-out stream, the original transferring-in stream becomes the computation stream, and the original transferring-out stream becomes the transferring-in stream. Hence, at the end of each round, we write some codes to swap the three streams and their associated memory pointers. We repeat this loop until all input data segments have been loaded into the global memory.




```
163         while (c < total) {
164           // One stream transfers its data into GPU
165           // One stream does its computation work
166           // One stream transfers its data out of GPU
167           cudaMemcpyAsync(input_ti, &host_input[input_size(c*SEG_SIZE)], input_size(r(c, Batch)) * sizeof(float),
168                           cudaMemcpyHostToDevice, trans_in);
169           conv_forward_kernel<<<ceil(output_size(SEG_SIZE)/96.0),96,0,compute>>>(output_c, input_c,
170                                                                                  NULL, SEG_SIZE,
171                                                                                  Map_out, Channel, Height,
172                                                                                  Width, K);
173           cudaMemcpyAsync(((void*) &host_output[output_size((c-2)*SEG_SIZE)]), output_to, output_size(SEG_SIZE)*sizeof(float),
174                           cudaMemcpyDeviceToHost, trans_out);
175
176           // Swap the three streams and their memory pointers
177           tmp = compute;
178           compute = trans_in;
179           trans_in = trans_out;
180           trans_out = tmp;
181
182           dtmp = input_c;
183           input_c = input_ti;
184           input_ti = input_to;
185           input_to = dtmp;
186
187           dtmp = output_c;
188           output_c = output_ti;
189           output_ti = output_to;
190           output_to = dtmp;
191
192           c++;
193         }
```

(a) (b)

Figure 25: Overlap three streams. (a): Overlapping scheme (b): Code implementation

After all input data segments have been loaded into the global memory, the original transferring-out stream won't add any more work to its queue, as there is no data segment to be loaded. We can then use the overlapping scheme (Figure 26a) and its corresponding implementation (Figure 26b) to wrap up the remaining two streams.


```
195         // Wrap up the computation and transferring work for two streams
196         conv_forward_kernel<<<ceil(output_size(r(c-1,Batch))/96.0),96,0,compute>>>(output_c, input_c,
197                                                                                    NULL, r(c-1, Batch),
198                                                                                    Map_out, Channel, Height,
199                                                                                    Width, K);
200         cudaMemcpyAsync(((void*) &host_output[output_size((c-2)*SEG_SIZE)]), output_to, output_size(SEG_SIZE)*sizeof(float), cudaMemcpyDeviceToHost, trans_out);
201         cudaMemcpyAsync(((void*) &host_output[output_size((c-1)*SEG_SIZE)]), output_c, output_size(r(c-1, Batch)) * sizeof(float),
202                         cudaMemcpyDeviceToHost, compute);
```

(a) (b)

Figure 26: Finish computing and transferring out the last two segments. (a): Overlapping scheme (b): Code implementation

```
205         // Make sure all of the three streams have finished all their work in the queue
206         cudaStreamSynchronize(trans_out);
207         cudaStreamSynchronize(compute);
208         cudaStreamSynchronize(trans_in);
209
210         cudaStreamDestroy(trans_out);
211         cudaStreamDestroy(compute);
212         cudaStreamDestroy(trans_in);
```

Figure 27: Code snippet used to synchronize and destroy all streams

At the end, we can use the code snippet in Figure 27 to make sure all streams have finished their work.

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

The performance statistics is in Table 15 below. By comparing it with M2's performance statistics in Table 1, we can see the total Layer times are reduced by over 10%. By taking a closer look at the profiling results from Nsight-Systems, we can get Table 16 below. As we can see, the computing work load is evenly distributed among the three streams, for both CNN layers. In this way, the three streams are overlapped with each other, and both of PCIe links and GPU kernels spend less time being idle.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.002863*	*0.002933*	*0.005796*	*3.72*	*3.43*	*7.15*	*0.86*
1000	*0.002144*	*0.002505*	*0.004649*	*18.9*	*15.1*	*34.0*	*0.886*
10000	*0.00468*	*0.005168*	*0.009848*	*180*	*132*	*312*	*0.8714*

Table 15: Performance statistics of Op #0

Layer	Stream 1	Stream 2	Stream 3
1	*18.6%*	*18.2%*	*18.0%*
2	*15.2%*	*15.0%*	*15.0%*

Table 16: CUDA hardware running time percentages of the three streams, for batch size 10,000

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

The implementation of this technique already includes Optimization #4. Furthermore, since we divide input data segments by images, this streaming technique can also synergize with optimizations that are associated with tiled convolution, like #1 and #9. In fact, I implement this combination (Please check out the folder project/m3/op_0_1_4_9 in Github). After some tuning, I find out 125-image is the best input data segment size for this combination. In addition, since this implementation involves atomicAdd calls (Optimization #9), we need to reset the output memory to zero, whenever we're going to use it for a kernel launch, as illustrated in Figure 28.

```
207         // Set up the kernel dimensions and zero the output memory (for atomicAdd)
208         g = dim3(Map_out, Y, Batch);
209         cudaMemset(output_to, 0, output_size(SEG_SIZE)*sizeof(float));
210         conv_forward_kernel<<<g,b,0,trans_out>>>(output_to, input_to, NULL, Batch, Map_out, Channel, Height, Width, K);
```

Figure 28: Reset output memory for streaming kernels

The performance statistics of this combination is in Table 17 below. By comparing it with Table 12, we can see the total Layer time is cut down by about 10% for the batch size of 1,000, and by about 20% for the batch size of 10,000.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.003423*	*0.003912*	*0.007335*	*4.30*	*4.64*	*8.94*	*0.86*
1000	*0.003075*	*0.003236*	*0.006311*	*21.1*	*16.8*	*37.9*	*0.886*
10000	*0.003587*	*0.002455*	*0.006042*	*180*	*133*	*313*	*0.8714*

Table 17: Performance statistics of Op #0+#1+#4+#9

For similar reasons, we can also combine this streaming technique with optimizations that are related to unrolling input matrix multiplication, like #2, #3 and #12. I implement this combination (Please check out the folder project/m3/op_0_2_3_4_12 in Github). After some tuning, I find out 100-image is the best input data segment size for this combination. The performance statistics of this combination is in Table 18 below. By comparing it with Table 13, we can see the total Layer time is cut down by about 4.8% for the batch size of 1,000, and by about 8.6% for the batch size of 10,000.

Batch Size	Op Time 1 (ms)	Op Time 2 (ms)	Total Execution Time / Op1+Op2 (ms)	Layer Time 1 (ms)	Layer Time 2 (ms)	Sum of Layer Times (ms)	Accuracy
100	*0.001983*	*0.001853*	*0.003836*	*4.03*	*3.69*	*7.72*	*0.86*
1000	*0.002225*	*0.002776*	*0.005001*	*20.6*	*14.8*	*35.4*	*0.886*
10000	*0.003476*	*0.003507*	*0.006983*	*179*	*128*	*307*	*0.8714*

Table 18: Performance statistics of Op #0+#2+#3+#4+#12

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

I use Slide Deck 20 to implement this technique.