EECS 471 Final Project Report

Yan Cheng Poon

Optimization Strategy 1:

Shared Memory Tiling for Convolution (2D Output Tiling) + Shared Memory Filter Weight

In the baseline kernel, the input tensor and filter weight data are accessed from global memory, which is really slow because each data point is accessed many times in this kernel, and each time, data is fetched from slow global memory accesses. To optimize this, I decided to first load both the input tensor data and the filter weight data into shared memory, so that whenever the data is needed, it can be loaded from shared memory. This is because shared memory access is way faster than global memory access due to the GPU architecture of shared memory being close to each block.

To do this, I initialized a dynamic shared memory array to store both the input tensor and the filter weight. I also calculated the size of the input tensor data needed for the block, which is $KERNEL\_WIDTH^2$, where KERNEL_WIDTH = TILE_WIDTH + K - 1. Hence, the allocation of shared memory for the filter weight starts at index $KERNEL\_WIDTH^2$.

```
int KERNEL_WIDTH = TILE_WIDTH + K - 1;
extern __shared__ float shared[];
float* sharedmem_x = &shared[0];
float* sharedmem_w = &shared[KERNEL_WIDTH*KERNEL_WIDTH];
```

Figure 1. Shared Memory Initialization

Each block then loads in the necessary portion of the input tensor data and the filter weight for the computation. The threads are synchronized before computation. After this optimization, data is now loaded from shared memory instead of global memory, achieving a faster kernel.

```
for (int i = ty; i < KERNEL_WIDTH; i += TILE_WIDTH) {
  int h_idx = h_currBlock + i;
  for (int j = tx; j < KERNEL_WIDTH; j += TILE_WIDTH) {
      int w_idx = w_currBlock + j;
      float val = (h_idx < H && w_idx < W) ? x4d(b, c, h_idx, w_idx) : 0.0f;
      sharedmem_x[i * SHARED_X_STRIDE + j] = val;
  }
}
```

Figure 2. Loading Input Tensor Data "X" to Shared Memory

```
if ((tx < K) && (ty < K)) {
  sharedmem_w[ty*K+tx] = k4d(m, c, ty, tx);
}
```

Figure 3. Loading Filter Weight "W" to Shared Memory

Performance:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.05197414398193359
Op Time: 0.17363558959960937
Correctness: 0.7955 Model: eecs471

Performance was definitely improved, however, it still fails to meet the 0.2-second requirement.

Optimization Strategy 2:
Shared Memory Tiling for Convolution + Shared Memory Filter Weight from Constant Memory

At this point, I was playing around with constant memory, I noticed how the filter weights are constant across all blocks and are reused many times across different blocks, which led me to store the filter weights in constant memory. This is because constant memory is generally faster than global memory due to its read-only properties, and hence, will be suitable for data like filter weights, which are constant. However, at this point, I was still in the belief that in each block, loading the necessary filter weights into shared memory from the constant memory and using shared memory is faster than directly using constant memory. This led to me keeping the shared memory loading phase of filter weights.

To initialize constant memory, I need to calculate the size of my constant memory. The size of constant memory is M*C*K*K. K is always constant at 7, M and C, however, can be different depending on layers, hence, we used the largest M and largest C value, which are 24 and 12, respectively. That gives us 7*7*24*12 = 14112 constant memory size. Knowing the size, we just need to initialize the memory and use cudaMemcpyToSymbol to copy weights to constant memory.

```
__constant__ float constMem[14112];
```

```
cudaMemcpyToSymbol(constMem, w.data_ptr<float>(), 14112*sizeof(float), 0, cudaMemcpyDeviceToDevice);
```

Figure 4. Initializing Constant Memory for Filter Weights

Performance:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.05039820861816406
Op Time: 0.1716510467529297
Correctness: 0.7955 Model: eecs471

Performance was barely improved from the first one, with ~0.003-second improvement. It is clear that we need a different strategy to meet the requirement.

Optimization Strategy 3:
Shared Memory Tiling for Convolution + Direct Constant Memory Filter Weight

Optimization Strategy 2 got me thinking if shared memory for filter weight is even needed at all. It is perhaps faster to access constant memory directly whenever filter weights are needed. This is because the time taken to load filter weights into shared memory is actually nontrivial in the total execution time, and accessing from shared memory for filter weights doesn't seem to benefit at all compared to accessing from constant memory.

Hence, I removed the code where filter weights are loaded into shared memory and made my convolution code access the constant memory directly.

Performance:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.04413849639892578
Op Time: 0.14239945983886718
Correctness: 0.7955 Model: eecs471

I was finally able to meet the 0.2-second requirement. It seems that in this assignment, accessing filter weights directly from constant memory is much more beneficial than loading the filter weights to shared memory, as we can see significant improvements (~0.05s reduction) when shared memory is completely removed for filter weights. We can also conclude from this

experiment that combining shared memory and constant memory is not a good idea, as the benefits of constant memory are not prevalent when shared memory is combined.

Final Strategy: Optimization Strategy 4:

Shared Memory Tiling + Constant Memory Filter Weight + 1D Loop Unrolling

To further improve my kernel, I was exploring loop unrolling techniques. It was to my understanding that unrolling a constant iteration loop can reduce warp divergence because the loop introduces branching in assembly code, which will result in warps taking different paths and consequently result in more stalls. If a constant iteration loop is unrolled, it ensures that all warps execute the same instructions, reducing divergence and improving warp efficiency. Additionally, unrolling loops removes various overhead instructions that come with a loop function (ie, branching, counter, condition check). Removing these overheads means reducing the instructions executed in every thread, which will increase the overall efficiency.

Hence, with K being a constant at 7, we can unroll the convolution loop to 7 lines of instructions. See Figure 5 for the convolution loop before unrolling, and see Figure 6 for the convolution loop after 1D unrolling.

```
for (int p = 0; p < K; ++p) {
    for (int q = 0; q < K; ++q) {
        res += sharedmem[(ty + p) * SHARED_X_STRIDE + (tx + q)] * k4d(m, c, p, q);
    }
}
```

Figure 5. Convolution Loop Before Unrolling

```
for (int p = 0; p < K; ++p) {
    int y_idx = ty + p;
    res += sharedmem[y_idx * SHARED_X_STRIDE + (tx + 0)] * k4d(m, c, p, 0);
    res += sharedmem[y_idx * SHARED_X_STRIDE + (tx + 1)] * k4d(m, c, p, 1);
    res += sharedmem[y_idx * SHARED_X_STRIDE + (tx + 2)] * k4d(m, c, p, 2);
    res += sharedmem[y_idx * SHARED_X_STRIDE + (tx + 3)] * k4d(m, c, p, 3);
    res += sharedmem[y_idx * SHARED_X_STRIDE + (tx + 4)] * k4d(m, c, p, 4);
    res += sharedmem[y_idx * SHARED_X_STRIDE + (tx + 5)] * k4d(m, c, p, 5);
    res += sharedmem[y_idx * SHARED_X_STRIDE + (tx + 6)] * k4d(m, c, p, 6);
}
```

Figure 6. Convolution Loop After 1D Unrolling

Performance:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.04063334274291992
Op Time: 0.10697727966308594
Correctness: 0.7955 Model: eecs471

Unrolling just the 1D Loop makes a huge difference by improving the execution time by ~0.05 seconds, making my kernel well meet the requirements of 0.2 seconds. It is honestly a huge surprise to me to find out that a simple loop unrolling would result in such a big improvement. To further analyze the factor behind this huge improvement, I ran the NSight Profiler to analyze the difference between the code in Figure 5 and Figure 6. The profiler screenshot for code before unrolling is in Figure 7, while the profiler screenshot for code after unrolling is in Figure 8.

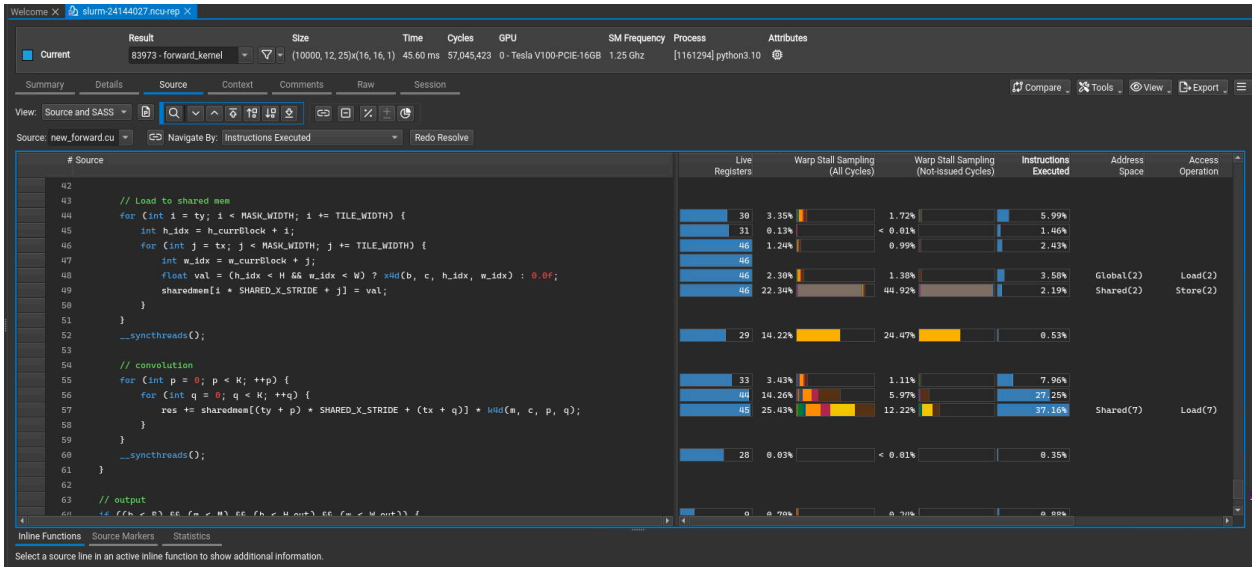

Figure 7. Profiler for Code Before Unrolling

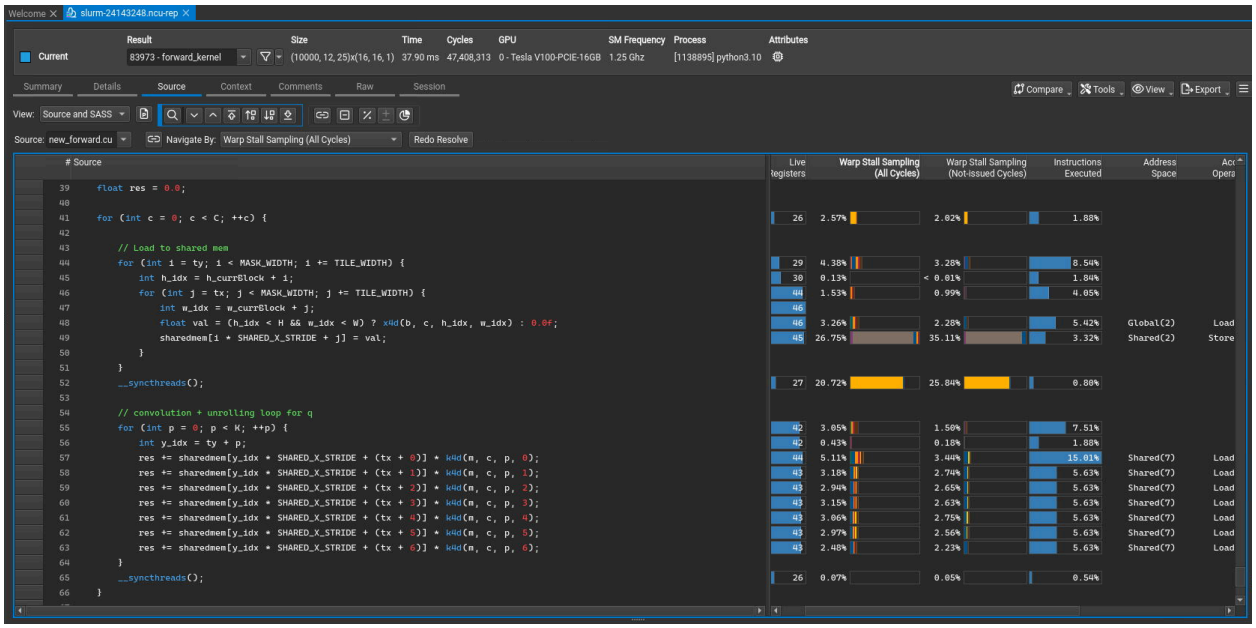

Figure 8. Profiler for Code After 1D Unrolling

Reading from this profiler, we can see that the total instructions executed in the inner loop of Figure 8 add up to 50.67% of the program, which is significantly less than the inner loop of Figure 7 at 64.41%. This further proves that unrolling loops reduces the overhead instructions executed, leading to more efficient execution.

Additionally, the warp stalls of the inner loop in Figure 8 add up to 23.32%, while the inner loop in Figure 7 adds up to 39.69% of total warp stalls. This further proves that warp stalls significantly reduce when the loop is unrolled, possibly due to the reduction of warp divergence that is caused by the inherent branching in loops.

Other Optimization Strategies Used and Experimented

Tiling and Block/Grid Dimensions

I used the output tiling strategy where my blockDim is 2D with TILE_WIDTH size in the x dimension and TILE_WIDTH size in the y dimension. For my grid dimensions, I made it 3D, with batch (B) size as my x dimension size, number of output channels (M) as my y dimension size, and finally, total width * total height / $TILE\_WIDTH^2$ as my z dimension size. With this, I can account for sufficient blocks to compute all outputs for all output channels in the entire batch.

```
const int W_numBlocks = ceil((float) W_out / TILE_WIDTH);
const int H_numBlocks = ceil((float) H_out / TILE_WIDTH);
dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
dim3 gridDim(B, M, W_numBlocks * H_numBlocks);
```

Figure 9. Instantiation of Dimensions

Padding +1 to reduce bank conflicts

Without padding, shared memory is indexed by sharedmem[y_idx * KERNEL_WIDTH + tx] where KERNEL_WIDTH = TILE_WIDTH + K - 1. Due to K being 7, KERNEL_WIDTH ends up being a multiple of 2, and hence, has an increased chance of bank conflicts (multiple threads may hit the same bank). A potential solution to this that I found from research was to pad my shared memory stride by 1 index and have this new variable SHARED_X_STRIDE.

```
const int SHARED_X_STRIDE = KERNEL_WIDTH + 1;
```

Figure 10. SHARED_X_STRIDE

Using this new variable, I essentially stride through different threadIdx.y by SHARED_X_STRIDE amount (which is 1 more than it is supposed to be), in an attempt to reduce bank conflicts, as each thread would fall into different banks.

Performance:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.04044800186157226
Op Time: 0.10601369476318359
Correctness: 0.7955 Model: eecs471

It turned out to have very minimal improvements, if at all. This might be due to the fact that K is 7 and hence, TILE_WIDTH + K - 1 isn't really a multiple of 16 or 32 to begin with. The padding probably doesn't make much sense because of that. However, if TILE_WIDTH = 16 and K = 17, it is pretty likely that we will see a huge performance improvement by padding 1 index and reducing bank conflicts.

TILE_WIDTH Sweep

Lastly, I attempted to sweep the TILE_WIDTH size by 8, 16, and 32 to find out which has the best performance. The performance comparison is shown below:

Performance for TILE_WIDTH = 8:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.045175807952880856
Op Time: 0.12555980682373047
Correctness: 0.7955 Model: eecs471

Performance for TILE_WIDTH = 16:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.04063334274291992
Op Time: 0.10697727966308594
Correctness: 0.7955 Model: eecs471

Performance for TILE_WIDTH = 32:
Loading fashion-mnist data...
Warming up CUDA kernels...
Loading model weights...
Running New Inference
Op Time: 0.07672422027587891
Op Time: 0.1567150115966797
Correctness: 0.7955 Model: eecs471

TILE_WIDTH=16 seems to yield the best performance. It is surprising that TILE_WIDTH=32 yields significantly worse performance than 8 or 16, as it was my guess that TILE_WIDTH=32 would have the best performance out of all. More research and profiling are needed to understand the underlying behavior behind this odd phenomenon.

Future Improvements

2D Unrolling

I have only unrolled the inner loop for this project. Since the outer loop for the convolution is also constant at K=7 iterations, we could potentially unroll the outer loop as well. It would be extremely tedious, however, as there are 49 lines of code to write if we choose to unroll both loops. Some performance boost could be expected in return due to the reduction of warp divergence and the reduction of loop overhead instructions.

Matrix Multiplication

Using the matrix multiplication strategy could potentially improve the performance of this kernel due to more perfectly coalesced memory and the reduction of redundant loads. Using matrix multiplication would also potentially have better parallelism than the current convolution method. That is worth trying out after this project to push for better execution time.

Multiple Kernel Implementation

Using multiple specialized kernels can improve performance by allowing each kernel to focus on a specific task or data pattern, optimizing resource usage and parallelism. By splitting kernels, threads have more uniform tasks and hence, have less divergence and better memory access patterns. This is also something that is worth trying out after this project.