Umich EECS498 Applied Parallel Programming with GPUs

Final Project Report

Dec 2019

Team 8

Wentao Zhang

Haojie Ye

Contents

Chapter 1

Implemented Optimizations

1.1 Shared Memory Convolution

1.1.1 Reason

We use a 2-dimensional shared memory to map to each output image pixels. Each block of shared memory fetches the input pixel of the corresponding image and output the output features. Using shared memory implementation in CUDA programming usually achieves 2 benefits based on what we have learned in class. They are (a) Increase the compute/memory access ratio. (b) Memory coalescing (when fetching from global memory to shared memory). In the convolution kernel, we use shared memory mainly to achieve better compute/memory access ratio. We initially used a shared memory size (32x32), in this case the shared memory implementation has 1000x compute/memory access ratio compared with fetching each data from the global memory. Later we modified the shared memory size to 11x11 and 24x24 to avoid divergence within the block. This heuristic tuning creates a trade off between the compute/memory access ratio and the amount of thread divergence in the thread block, but the overall principle is that the shared memory implementation will greatly improve the compute/memory access ratio, reduce the amount of the global memory accesses, and achieves better kernel performance.

1.1.2 Implementation Method and Effect

The CUDA code of implementing shared memory is as follows. Note that this piece of code demo (Fig. 1.1) captures the idea of how we implement the shared memory, but more details are open to change to fit in more optimizations in our work.

```
    for(int c = 0; c < C; c++){
        //load the feature map to shared memory
        if(h_out < H && w_out < W){
            featuremap[ty][tx] = x4d(b, c, h_out, w_out);
        }else{
            featuremap[ty][tx] = 0;
        }
        __syncthreads();
        //do convolution
        if(tx < out_Dim && ty < out_Dim && h_out < H_out && w_out < W_out){
            float result = 0;
            for(int p = 0; p < K; p++){
                for(int q = 0; q < K; q++){
                    result += featuremap[ty + p][tx + q] * filter4d(m, c, p, q);
                }
            }
            y4d(b, m, h_out, w_out) += result;
        }
        __syncthreads();
    }
}
```

Figure 1.1: Shared Memory Implementation (Final version differs from this piece of code)

Because the convolution kernel will shrink the size of input, we fetch more elements as the input feature than output features, just as we did in 3D convolution kernel implementation in homework. For example, if we set the shared memory size is 32x32, the dimension of the output block size will be 32 - KernelWidth + 1. We then compute each output features with the kernel information along with the input features in the shared memory (shown in Fig. 1.1). The shared memory implementation (along with the parallelism in output images) has a large improvement on the performance, as it reduces the kernel execution time of the second case from 25s to about 0.7s.

1.2 Weight Matrix in Constant Memory

1.2.1 Reason

CUDA constant memory enables a faster scratchpad structure and interface to efficiently read from the read-only memory. In the convolution kernel, we observed that the kernel information remains unchanged during the kernel execution time. We exploit this observation and put the kernel data in the read-only constant memory, which gives better performance than fetching the kernel data from the global memory or the shared memory (putting kernel data in shared memory may generate unnecessary bank conflict).

```
__constant__ float filter[64000 / sizeof(float)];
```

Figure 1.2: Constant memory initialization

```
cudaMemcpyToSymbol(filter, w.dptr_, M * C * K * K * sizeof(float));
```

Figure 1.3: Constant memory copy from host to device

1.2.2 Implementation Method and Effect

The CUDA code of implementing constant memory is as follows. Constant memory initialization example is shown in Fig. 1.2, and constant memory copy from host to device example is shown in Fig. 1.3. The constant memory implementation has little effect on the improvement on the performance, as it reduces the kernel execution time of the second case for less than 0.1s.

1.3 Parallelism in Output Images

1.3.1 Reason

The starter code shows the example of assigning each thread for each single image. This implementation lacks parallelism in output images because threads in each output images execute in sequence. Our purposed layout of the threads assign each thread for each output image pixels. Different threads can collaborate in computing the output features and construct the features for each image in parallel, thus achieving parallelism in input images. The layout of the thread blocks mapping to output features of the image set is illustrated in Fig. 1.4. The M output features for each images are expanded so that the output features can be computed in parallel. The layout of the thread blocks enables computation of the output features in parallel, compared with the baseline design where M output features are computed sequentially. This parallelism in output images gives M times better performance (if neglecting the launching overhead of the additional thread blocks).

1.3.2 Implementation Method and Effect

The CUDA code of implementing parallelism in output image is as follows. As shown in Fig. 1.5, these parameters are initialized in order to index the output features for each thread block in parallel. Overall, assume that the dimension of the thread block is (Blocksize, Blocksize). Thus, the dimension of grid is (number of images x M x ceil(ImageWidth / Blocksize), ceil(ImageHeight /

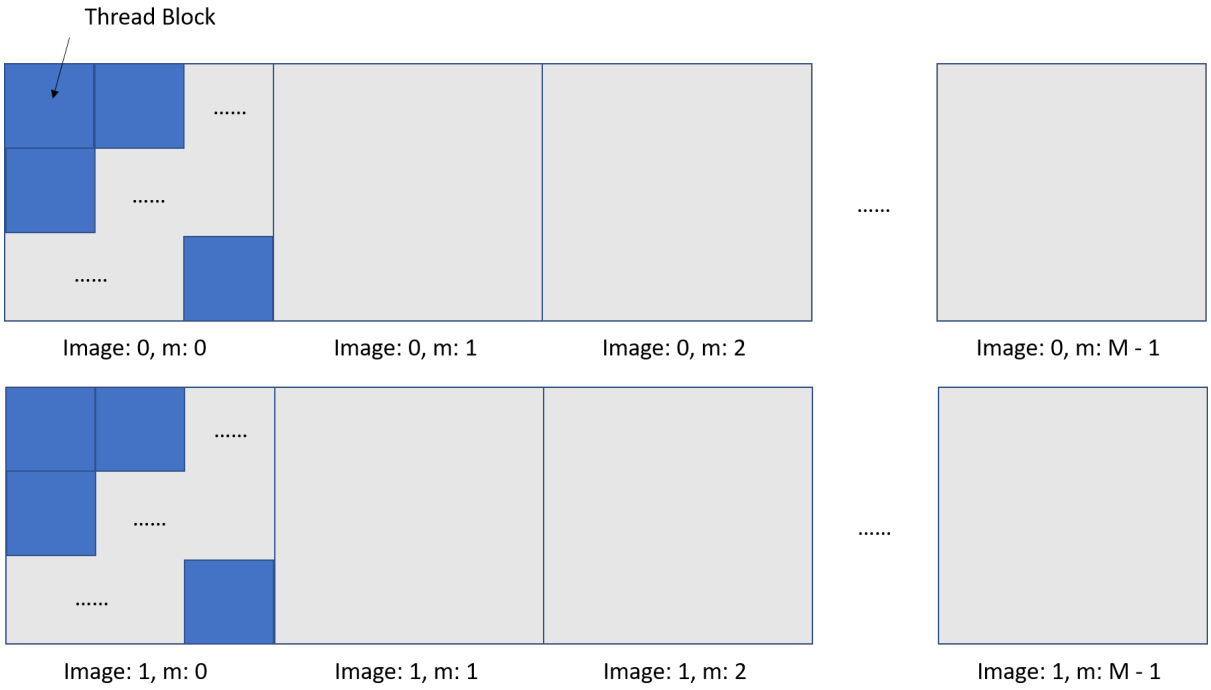

Figure 1.4: Constant memory initialization

```
int out_Dim = blockDim.x - K + 1; // assume x_size = y_size

int size_blk_x_per_image = (W_out + out_Dim - 1) / out_Dim;
int blockxNum_per_image = M * size_blk_x_per_image; //M * ceil(W_out/out_DIm) :::ceil(W_out/out_Dim)-> block per result

int m = (blockIdx.x / size_blk_x_per_image) % M; // (blockIdx.x / ceil(W_out/out_Dim)) % M

int h_out = blockIdx.y * out_Dim + ty;
int w_out = (blockIdx.x % size_blk_x_per_image) * out_Dim + tx;

int b = blockIdx.x / blockxNum_per_image;
```

Figure 1.5: Constant memory initialization

Blocksize). The shared memory implementation (along with the parallelism in output images) has a large improvement on the performance, as it reduces the kernel execution time of the second case from 25s to about 0.7s.

1.4 Sweeping Various Parameters

1.4.1 Reason

The width and height of the image differs in the two test cases that are used to test the performance of the convolution kernel. Specifically, they are 33 and 72. If we use a unified block size to be 32x32, the number of block allocated for each output image and feature will be 2x2 and 3x3. What is more, 3 out of 4, 5 out of 9 of the thread blocks will have a control divergence because these blocks have thread size that exceeds the image size. Under this observation, we

```
int out_Dim1 = 17 - K + 1;
int out_Dim2 = 30 - K + 1;
int out_Dim3 = 32 - K + 1;

dim3 gridDim1(B * M * ((H_out + out_Dim1 - 1) / out_Dim1), (H_out + out_Dim1 - 1) / out_Dim1);
dim3 blockDim1(17, 17);

dim3 gridDim2(B * M * ((H_out + out_Dim2 - 1) / out_Dim2), (H_out + out_Dim2 - 1) / out_Dim2);
dim3 blockDim2(30, 30);

dim3 gridDim3(B * M * ((H_out + out_Dim3 - 1) / out_Dim3), (H_out + out_Dim3 - 1) / out_Dim3);
dim3 blockDim3(32, 32);

MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

cudaMemcpyToSymbol(filter, w.dptr_, M * C * K * K * sizeof(float));
// std::cout<<"B:"<<B<<" M:"<<M<<" C:"<<C<<" H:"<<H<<" W:"<<W<<" K:"<<K<<'\n';
if(H == 33){
    forward_kernel<<<gridDim1, blockDim1>>>(y.dptr_, x.dptr_, B, M, C, H, W, K);
}
else if (H == 72){
    forward_kernel<<<gridDim2, blockDim2>>>(y.dptr_, x.dptr_, B, M, C, H, W, K);
}
else{
    forward_kernel<<<gridDim3, blockDim3>>>(y.dptr_, x.dptr_, B, M, C, H, W, K);
}
MSHADOW_CUDA_CALL(cudaDeviceSynchronize());
```

Figure 1.6: Constant memory initialization

purpose to sweep various size of thread block and size of grid to reduce the control divergence in the thread blocks. For example if we set the thread block size to be 11 to tackle the test case where the image size is 33, we lose some level of compute/memory access ratio because the reduction in shared memory size, but we have less control divergence in the kernel code because no thread size exceeds the image size when fetching image data with thread blocks.

1.4.2 Implementation Method and Effect

The CUDA code of implementing sweeping Various parameters is as follows. As shown in Fig. 2.1, we tailor different thread block sizes for the two test cases, and for the rest of the cases, we apply the default thread block size 32x32. In this way, we examined the trade off between the compute/memory access ratio and the amount of control divergence within the thread block. The sweeping various parameters implementation has a medium improvement on the performance, as it reduces the kernel execution time of the second case from 0.7s further to about 0.4s.

1.5 Unroll Loop

1.5.1 Reason

We have observed that the kernel size has remained unchanged in different test cases. This gives us a motivation to unroll the loop that iterate the kernel data. In a conventional way (as in the starter code), there are two temporary variables that are used to iterate the kernel. We purpose to unroll the loop that iterates the constant kernel. This will potentially save the ALU operation that increment and bound checking of the two temporary variables.

1.5.2 Implementation Method and Effect

The CUDA code of implementing loop unrolling is as follows. The Fig. 1.7 shows in part that the kernel read loop is unrolled into 49 consecutive lines. This unrolling of the loop saves the redundant instructions that are used to loop through the kernel data, since we know that the kernel has a fixed size of 7. Unrolling to 49 lines allows the thread to progress without frequent checking of the bounding conditions and increment of the temporary variables. The unrolling loop implementation has a medium improvement on the performance, as it reduces the kernel execution time of the second case from 0.4s further to about 0.3s.

```
__syncthreads();
//do convolution
if(tx < out_Dim && ty < out_Dim && h_out < H_out && w_out < W_out){
    result += featuremap[ty + 0][tx + 0] * filter4d(m, c, 0, 0);
    result += featuremap[ty + 0][tx + 1] * filter4d(m, c, 0, 1);
    result += featuremap[ty + 0][tx + 2] * filter4d(m, c, 0, 2);
    result += featuremap[ty + 0][tx + 3] * filter4d(m, c, 0, 3);
    result += featuremap[ty + 0][tx + 4] * filter4d(m, c, 0, 4);
    result += featuremap[ty + 0][tx + 5] * filter4d(m, c, 0, 5);
    result += featuremap[ty + 0][tx + 6] * filter4d(m, c, 0, 6);

    result += featuremap[ty + 1][tx + 0] * filter4d(m, c, 1, 0);
    result += featuremap[ty + 1][tx + 1] * filter4d(m, c, 1, 1);
    result += featuremap[ty + 1][tx + 2] * filter4d(m, c, 1, 2);
    result += featuremap[ty + 1][tx + 3] * filter4d(m, c, 1, 3);
    result += featuremap[ty + 1][tx + 4] * filter4d(m, c, 1, 4);
    result += featuremap[ty + 1][tx + 5] * filter4d(m, c, 1, 5);
    result += featuremap[ty + 1][tx + 6] * filter4d(m, c, 1, 6);

    result += featuremap[ty + 2][tx + 0] * filter4d(m, c, 2, 0);
    result += featuremap[ty + 2][tx + 1] * filter4d(m, c, 2, 1);
    result += featuremap[ty + 2][tx + 2] * filter4d(m, c, 2, 2);
    result += featuremap[ty + 2][tx + 3] * filter4d(m, c, 2, 3);
    result += featuremap[ty + 2][tx + 4] * filter4d(m, c, 2, 4);
    result += featuremap[ty + 2][tx + 5] * filter4d(m, c, 2, 5);
    result += featuremap[ty + 2][tx + 6] * filter4d(m, c, 2, 6);
```

Figure 1.7: Constant memory initialization

Chapter 2

Output and Profiler Screenshot

Using the nvprof, we get the following profiler result. As we can see, comparing with the original version, which takes about 20s to do the 2 forward convolution layer, our final version takes 0.09s and 0.345s for two layers, with the same correctness as the original one.

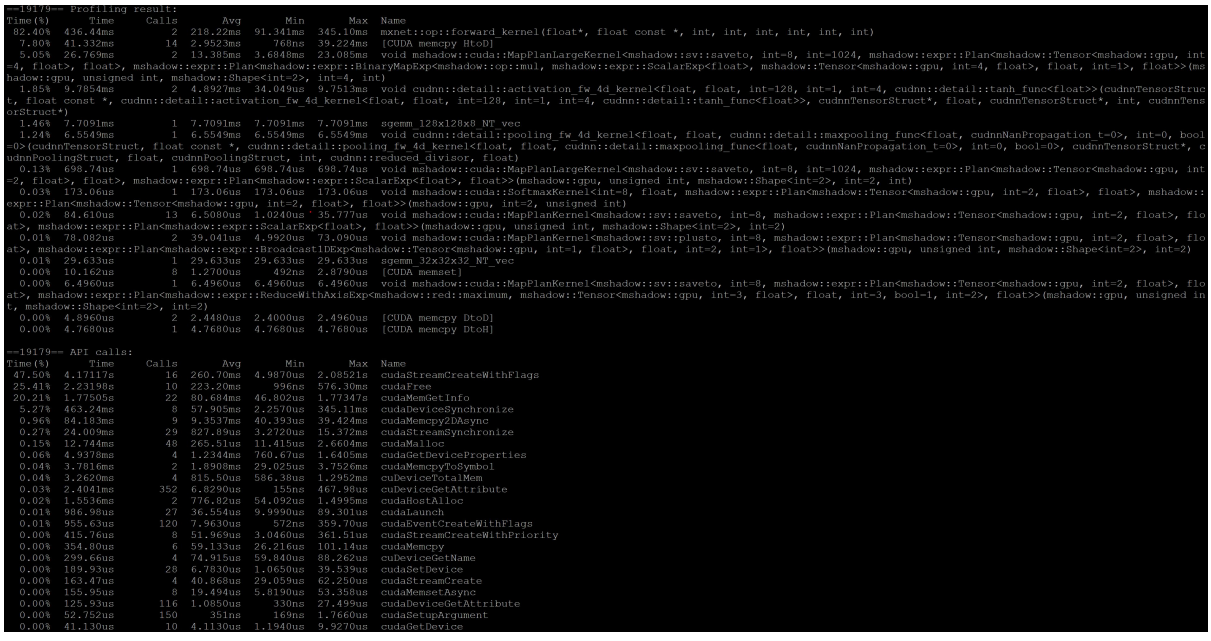

Figure 2.1: Profiler Screenshot

Chapter 3

Work Distribution

Wentao Zhang:

Shared Memory Convolution, Weight Matrix in Constant Memory, Parallelism in Output Images

Haojie Ye:

Shared Memory Convolution, Weight Matrix in Constant Memory, Sweeping Various Parameters, Unroll Loop.