ECE408 - Applied Parallel Programming
Final Report

Neil Singh (ngsingh2)
Patrick McMahon (pfmcmah2)
Matthew Krikorian (krikorn2)

December 18, 2017

Milestone 1

Milestone 1 of this project was mostly setting up our test environment and preparing it to start developing the later parts of this project.

Running Time of m1.1.py

For m1.1.py, we had an accuracy of 0.8673, and an elapsed running time of 9.76 seconds.

Running Time of m1.2.py

For m1.2.py, we had an accuracy of 0.8673, and an elapsed running time of 3.19 seconds.

Most Time Consuming Kernels

Some of the most time consuming kernels when running the nvprof profile were the*implicit_convolve_sgemm* kernel, taking up 36.96% of the execution time, the *activation_fw_4d_kernel* kernel, taking up 14.33% of the execution time, and the *pooling_fw_4d_kernel* kernel, which took up 10.69% of the programs execution time.

Milestone 2

In this milestone of the project, we were tasked with writing sequential convolution code. Using the skeleton from our textbook, and the corresponding data accesses, we successfully implemented the function below.

```
{
    const int B = x.shape_[0];
    const int M = y.shape_[1];
    const int C = x.shape_[1];
    const int H = x.shape_[2];
    const int W = x.shape_[3];
    const int K = k.shape_[3];

    int H_out = H - K + 1;
    int W_out = W - K + 1;

    for (int b = 0; b < B; ++b)
    {
      for (int m = 0; m < M; m++)
      {
        for (int h = 0; h < H_out; h++)
        {
          for (int w = 0; w < W_out; w++)
          {
            y[b][m][h][w] = 0;

            for (int c = 0; c < C; c++)
            {
              for (int p = 0; p < K; p++)
              {
                for (int q = 0; q < K; q++)
                {
                  y[b][m][h][w] += x[b][c][h + p][w + q] * k[m][c][p][q];
                }
              }
            }
          }
        }
      }
    }
}
```

Figure 1: Our CPU implementation of the convolutional layer in MxNet

Our Results For Respective Models

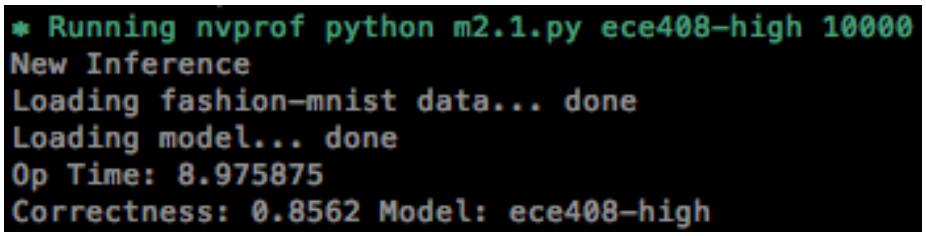

Figure 2: Accuracy and elapsed time for high correctness and sample size 10000

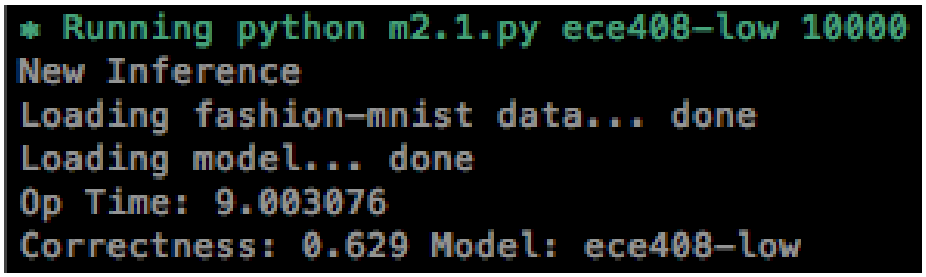

Figure 3: Accuracy and elapsed time for low correctness and sample size 10000

Milestone 3

In this milestone we were tasked with writing a baseline paralllel convolution kernel that had to pass several baseline tests. This implementation is very sub-optimal, as it uses a TILE_WIDTH of 1.

```
{
    const int H_out = H - K + 1;
    const int W_out = W - K + 1;

    #define y4d(i3,i2,i1,i0) y[(i3) * (M * H_out * W_out) + (i2)*(H_out
        * W_out) + (i1)*(W_out) + i0]
    #define x4d(i3,i2,i1,i0) x[(i3) * (C * H * W) + (i2)*(H * W) +
        (i1)*(W) + i0]
    #define k4d(i3,i2,i1,i0) k[(i3) * (C * K * K) + (i2)*(K * K) +
        (i1)*(K) + i0]

    int W_grid = ceil(W_out / (float)TILE_WIDTH);
    int H_grid = ceil(H_out / (float)TILE_WIDTH);
    int n, m, h, w, c, p, q;
    n = blockIdx.x;
    m = blockIdx.y;
    h = blockIdx.z / W_grid + threadIdx.y;
    w = blockIdx.z % W_grid + threadIdx.x;

    float acc = 0;
    for (c = 0; c < C; c++) {
        for (p = 0; p < K; p++) {
            for (q = 0; q < K; q++) {
                if (h+p < H && w+q < W)
                    acc += x4d(n, c, h+p, w+q) * k4d(m, c, p, q);
            }
        }
    }
    y4d(n, m, h, w) = acc;

    #undef y4d
    #undef x4d
    #undef k4d
}
```

Figure 4: Our GPU implementation of the convolutional layer in MxNet

Nvprof GPU Profile

```
* Running nvprof python m3.1.py
Loading fashion-mnist data... done
==311== NVPROF is profiling process 311, command: python m3.1.py
Loading model... done
Op Time: 21.654475
Correctness: 0.8562 Model: ece408-high
==311== Profiling application: python m3.1.py
==311== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
99.60%  21.6544s         1  21.6544s  21.6544s  21.6544s  void mxnet::op::forward_kernel
```

Figure 5: Nvprof GPU profile for our forward kernel

Optimized Layer

Initial Optimization

We knew right off the bat that the first thing we were going to need to optimize was the way our implementation handled tiling. Our current implementation was giving us a slightly incorrect accuracy for any TILE_SIZE above 1, so we first had to correct that to speedup our kernel. Below is the corrected kernel from part 2, which led to an average Op time of about 450ms instead of the 14s we got with our implementation where TILE_SIZE = 1. This new implementation had TILE_SIZE = 8. This was due in part to the correction of all the floating point operations we had.

```
{
    int W_grid = (int) ceil(W_out / TILE_WIDTH * 1.0);
//  int H_grid = ceil(H_out / TILE_WIDTH * 1.0);
    int n, m, h, w, c, p, q;
    n = blockIdx.x;
    m = blockIdx.y;
    h = (blockIdx.z / W_grid) * TILE_WIDTH + threadIdx.y;
    w = (blockIdx.z % W_grid) * TILE_WIDTH + threadIdx.x;
    float acc = 0.0;
    for (c = 0; c < C; c++) {
      for (p = 0; p < K; p++) {
        for (q = 0; q < K; q++) {
//        if (h+p < H && w+q < W)
            acc += x4d(n, c, h+p, w+q) * k4d(m, c, p, q);
        }
      }
    }
    y4d(n, m, h, w) = acc;}
```

Figure 6: Our first improved GPU implementation with TILE_SIZE = 8

Further Optimization

After we made this initial optimization, we knew that a running time of about 450ms was still too poor to really say we made any real optimizations to the kernel. After this point, we chose to optimize the shared memory and tiling implementation of our kernel, to improve on the DRAM accesses our code was making. Rather than taking variables from shared memory, our implementation was making calls to global memory every time it wanted to make an operation, which is highly inefficient. To correct this, we allocated shared memory for both the inputs and the weight matrix, to allow for much faster accesses to speed up our implementation. In our new implementation, we load the filter W into shared memory, then all the threads work together to load the input into a shared memory array. Then part of the sum is computed and put into Y, and then the calculation moves onto the next channel where the threads perform the next subset of convolution. We thought this optimization would be fruitful because we were now going to use shared memory to handle our operations, which would severely decrease our running time by reducing the latency to global memory accesses. With this new and improved implementation, we were able to reduce our running time all the way down to 151ms (200ms when we run Nvprof). Below is our implementation.

Optimization Code

```
for(c = 0; c < C; c++)
    {
      if((h0 < K) && (w0 < K))
      {
        W_shared[h0*K+w0] = k4d(m, c, h0, w0);
      }

      //__syncthreads();

      int itemp = h0temp;
      for(i = h; i < h_base + X_tile_width; i += TILE_WIDTH, itemp +=
          x_tile_tile_width)
      {
        for(j = w; j < w_base + X_tile_width; j += TILE_WIDTH)
        {
            X_shared[itemp + (j - w_base)] = x4d(n, c, i, j);
        }
      }

      __syncthreads();

      int ptemp = h0temp;
      for(p = 0; p < k2; p += K, ptemp += X_tile_width)
      {
        for(q = 0; q < K; q++)
        {
            acc += X_shared[ptemp + (w0 + q)] * W_shared[p+q];
        }
      }
      __syncthreads();

    }

y4d(n, m, h, w) = acc;
}
```

Figure 7: Our second improved GPU implementation with TILE_SIZE = 8

Nvprof GPU profile of Optimized Kernel

```
==317== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
 44.28%  1.88130s        18  104.52ms  16.944us  940.32ms  cudaStreamCreateWithFlags
 27.17%  1.15452s        10  115.45ms     852ns  329.56ms  cudaFree
 21.24%  902.56ms        23  39.242ms  236.93us  895.81ms  cudaMemGetInfo
  4.86%  206.68ms         1  206.68ms  206.68ms  206.68ms  cudaDeviceSynchronize
  1.85%  78.562ms        25  3.1425ms  5.2420us  42.421ms  cudaStreamSynchronize
  0.31%  12.986ms         8  1.6233ms  17.672us  4.5404ms  cudaMemcpy2DAsync
  0.15%  6.5768ms        41  160.41us  9.6770us  1.1189ms  cudaMalloc
  0.03%  1.4116ms         4  352.89us  337.57us  368.23us  cuDeviceTotalMem
  0.03%  1.3409ms         4  335.22us  52.578us  1.0484ms  cudaStreamCreate
  0.02%  845.02us       114  7.4120us     634ns  290.39us  cudaEventCreateWithFlags
  0.02%  843.11us       352  2.3950us     248ns  63.338us  cuDeviceGetAttribute
  0.01%  552.32us        24  23.013us  11.030us  58.201us  cudaLaunch
  0.01%  238.10us         6  39.683us  22.698us  75.956us  cudaMemcpy
  0.00%  98.102us         4  24.525us  18.421us  29.756us  cuDeviceGetName
  0.00%  77.039us        30  2.5670us     614ns  8.4120us  cudaSetDevice
  0.00%  68.728us       104     660ns     416ns  1.9050us  cudaDeviceGetAttribute
  0.00%  65.114us       145     449ns     244ns  1.3380us  cudaSetupArgument
  0.00%  35.856us         2  17.928us  17.517us  18.339us  cudaStreamCreateWithPriority
  0.00%  27.921us        24  1.1630us     448ns  2.8410us  cudaConfigureCall
  0.00%  16.601us        10  1.6600us  1.3160us  2.1260us  cudaGetDevice
  0.00%  9.2960us        17     546ns     319ns     930ns  cudaPeekAtLastError
  0.00%  4.7640us         6     794ns     268ns  1.8080us  cuDeviceGetCount
  0.00%  4.0080us         2  2.0040us  1.3730us  2.6350us  cudaEventRecord
  0.00%  3.8340us         1  3.8340us  3.8340us  3.8340us  cudaStreamGetPriority
  0.00%  3.6930us         2  1.8460us  1.5870us  2.1060us  cudaStreamWaitEvent
  0.00%  3.4670us         2  1.7330us  1.5390us  1.9280us  cudaDeviceGetStreamPriorityRange
  0.00%  3.3950us         6     565ns     349ns     785ns  cuDeviceGet
  0.00%  3.1780us         3  1.0590us     768ns  1.2740us  cuInit
  0.00%  2.5550us         5     511ns     405ns     711ns  cudaGetLastError
  0.00%  2.3020us         3     767ns     532ns  1.0950us  cuDriverGetVersion
  0.00%  1.3320us         1  1.3320us  1.3320us  1.3320us  cudaGetDeviceCount
```

Figure 8: Nvprof GPU profile for our optimized forward kernel

Summary

From the profiler, we can see a heavy increase in runtime speed from the running time of our previous implementation in milestone 3, showing that the shared memory and tiling improvements we implemented were truly worthwhile, and effective in making our forward convolution kernel much faster than it previously was. This implementation was influenced by the implementation in the textbook, which served as a great model for implementing a tiling kernel that utilized the benefits of shared memory to achieve peak performance.

Team Contributions

We all met up together and finished the sequential code for the convolutional layer in person. We also met up and wrote the parallel code together after meeting up in person, as well as crafting this report together for submission. Then for the final optimizations, we met up in person to speak about the implementation we were going to pursue, and used an online repository to assist each other in the implementation of our final kernel. We then documented our respective contributions in the report.