

ECE408 Project Report

Team: sthBigBig

Members: Guanchen He(ghe10), Guxin Jin(gjin7), Yingyi Zhang(yingyiz2)

1.3 NVPROF Profile

```
—307— NVPROF is profiling process 307, command: python /src/m1.2.py
Loading model...[05:51:06] src/operator/././cudnn_algorreg-inl.h:112: Running performance tests to find the best convolution algorithm, this can take a while... (setting env variable MXNET_CUDNN_AUTOTUNE_DEFAULT to 0 to disable)
done
EvalMetric: {'accuracy': 0.8673}
—307— Profiling application: python /src/m1.2.py
—307— Profiling result:
Time(%)   Time      Calls      Avg      Min      Max      Name
36.46% 49.296ms      1 49.296ms 49.296ms 49.296ms void cudnn::detail::implicit_convolve_sgemm<float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, float const *, int, cudnn::detail::implicit_convolve_sgemm<float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, float const *, kernel_conv_params, int, float, float, int, float const *, float const *, int, int)
28.20% 38.131ms      1 38.131ms 38.131ms 38.131ms sgemm_sm35_ldg_tn.128x8x256x16x32
14.33% 19.384ms      2 9.6920ms 455.22us 18.929ms void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *, cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>, cudnnTensorStruct*, float, cudnnTensorStruct*, int, cudnnTensorStruct*)
10.64% 14.392ms      1 14.392ms 14.392ms 14.392ms void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0>(cudnnTensorStruct, float const *, cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, int=0>, cudnnTensorStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
5.70% 7.7031ms     13 592.55us 1.6000us 5.6581ms [CUDA memcopy HtoD]
2.68% 3.6252ms      1 3.6252ms 3.6252ms 3.6252ms sgemm_sm35_ldg_tn.64x16x128x8x32
0.81% 1.0991ms      1 1.0991ms 1.0991ms 1.0991ms void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan-mshadow::Tensor-mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan-mshadow::Tensor-mshadow::gpu, int=2, float>, float>(mshadow::gpu, int=2, unsigned int)
0.55% 738.45us     12 61.537us 2.0480us 372.57us void mshadow::cuda::MapPlanKernel-mshadow::sv::saveto, int=8, mshadow::expr::Plan-mshadow::Tensor-mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan-mshadow::expr::ScalarExp<float>, float>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)
0.32% 430.55us      2 215.28us 17.119us 413.43us void mshadow::cuda::MapPlanKernel-mshadow::sv::plusto, int=8, mshadow::expr::Plan-mshadow::Tensor-mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan-mshadow::expr::Broadcast1DExp-mshadow::Tensor-mshadow::gpu, int=1, float>, float, int=2, int=1>, float>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)
0.29% 391.51us      1 391.51us 391.51us 391.51us sgemm_sm35_ldg_tn.32x16x64x8x16
0.02% 22.911us      1 22.911us 22.911us 22.911us void mshadow::cuda::MapPlanKernel-mshadow::sv::saveto, int=8, mshadow::expr::Plan-mshadow::Tensor-mshadow::gpu, int=2, float>, float>, mshadow::expr::Plan-mshadow::expr::ReduceWithAxisExp-mshadow::red::maximum, mshadow::Tensor-mshadow::gpu, int=3, float>, float, int=3, bool=1, int=2>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)
0.01% 9.6000us      1 9.6000us 9.6000us 9.6000us [CUDA memcopy DtoH]
```

Figure 1 screenshot of profile application

Table 1 several time-consuming kernels

Time(%)	Time	Name
36.46%	49.296ms	implicit_convolve_sgemm
28.20%	38.131ms	sgemm_sm35_ldg_tn
14.33%	19.384ms	activation_fw_4d_kernel
10.64%	14.392ms	pooling_fw_4d_kernel
5.70%	7.7031ms	cuda memcopy HtoD
2.68%	3.6252ms	sgemm_sm35_ldg_tn

From the table we find that the forward activation and pooling part could be optimized by leveraging parallel algorithm or techniques discussed in this lecture.

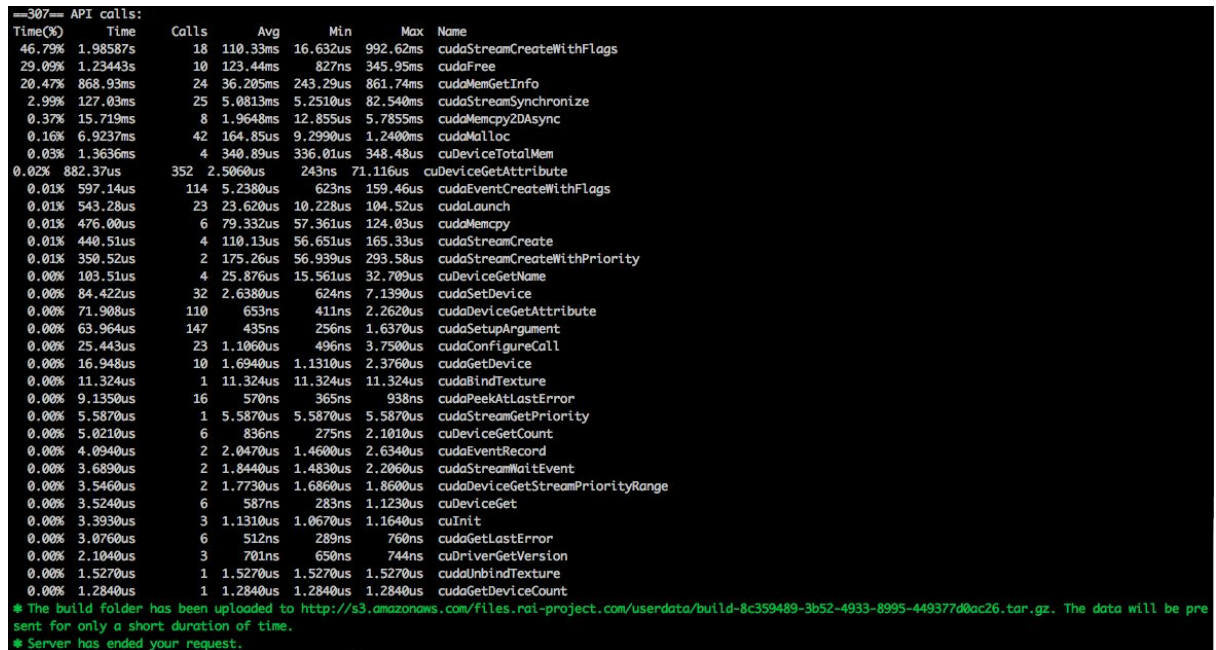


Figure 2 screenshot of API calls.

Table 2 some time-consuming API calls

Time(%)	Time	Name
46.79%	1.9858s	cudaStreamCreateWithFlag
29.09%	1.2344s	cudaFree
20.47%	868.93ms	cudaMemGetInfo
2.99%	127.03ms	cudaStreamSynchronize

2.1 Simple CPU implementation

In this step, we implemented a CPU convolutional kernel. We flowed the forward convolution described in Chapter 16 of the textbook. The implementation is a for-loop that loops through all the computation positions and produces the convolution result. The classification results with our convolution kernel are presented in the following figures.

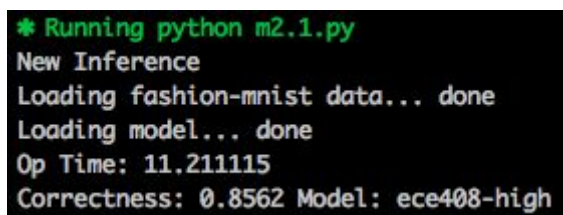


Figure 3 ece408-high model execution time and accuracy

```
* Running python m2.1.py ece408-low
New Inference
Loading fashion-mnist data... done
Loading model... done
Op Time: 11.220047
Correctness: 0.629 Model: ece408-low
```

Figure 4 ece408-low model execution time and accuracy

3 Simple GPU implementation and optimization plan

3.1 Simple GPU implementation

In this step, we implemented a simple GPU forward implementation. This implementation applies the convolution strategy discussed in ECE408 classes, i.e. tiled convolution. The implementation uses shared memory to apply convolution operation without extensive global memory loads. Figure 5 presents the performance of this version of implementation printed with nvprof. The kernel function tasks about 160ms to finish.

```
* Running nvprof python m3.1.py
New Inference
Loading fashion-mnist data... done
==311== NVPROF is profiling process 311, command: python m3.1.py
Loading model... done
Op Time: 0.163358
Correctness: 0.8562 Model: ece408-high
```

Figure 5 performance of our simple GPU forward implementation.

During the forward pass of the convolutional neural network, the weights in the kernel won't change. A straightforward optimization strategy is putting the kernel into constant memory to reduce the time consumption of accessing global memory. Since the kernel weights are already in global memory, we need to use **cudaMemcpyToSymbol()** function with copy type **cudaMemcpyDeviceToDevice** to copy kernel weights into constant memory. The performance of tiled convolution with constant memory is presented in Figure 6.

```
* Running nvprof python m3.1.py
New Inference
Loading fashion-mnist data... done
==311== NVPROF is profiling process 311, command: python m3.1.py
Loading model... done
Op Time: 0.146766
Correctness: 0.8562 Model: ece408-high
```

Figure 6 performance of tiled convolution with constant memory.

According to the result produced by nvprof, this simple optimization reduce the forward time from ~160ms to ~147ms. Further optimization plans are discussed in next section.

3.2 Optimization plan

Besides the basic optimization with constant memory, we also discussed about the further optimization strategies. Chapter 16 mainly presents two kinds of optimization techniques:

reduce the convolution to matrix multiplication and use FFT for convolution. We print the size of our kernel in our experiment. Since the kernel size is only 5X5, FFT may not provide great performance improvement in our convolution task. The reduction strategy doesn't have this kind of restriction. Thus we decide to apply this strategy and reduce our convolution to matrix multiplication.

Reducing convolution to matrix multiplication requires two steps to produce the convolution result. First, the kernels and the input data batch are converted into two large matrix in unroll step. Then a matrix multiplication is conducted to produce the final result. Figure 7 exhibits the basic process of this strategy.

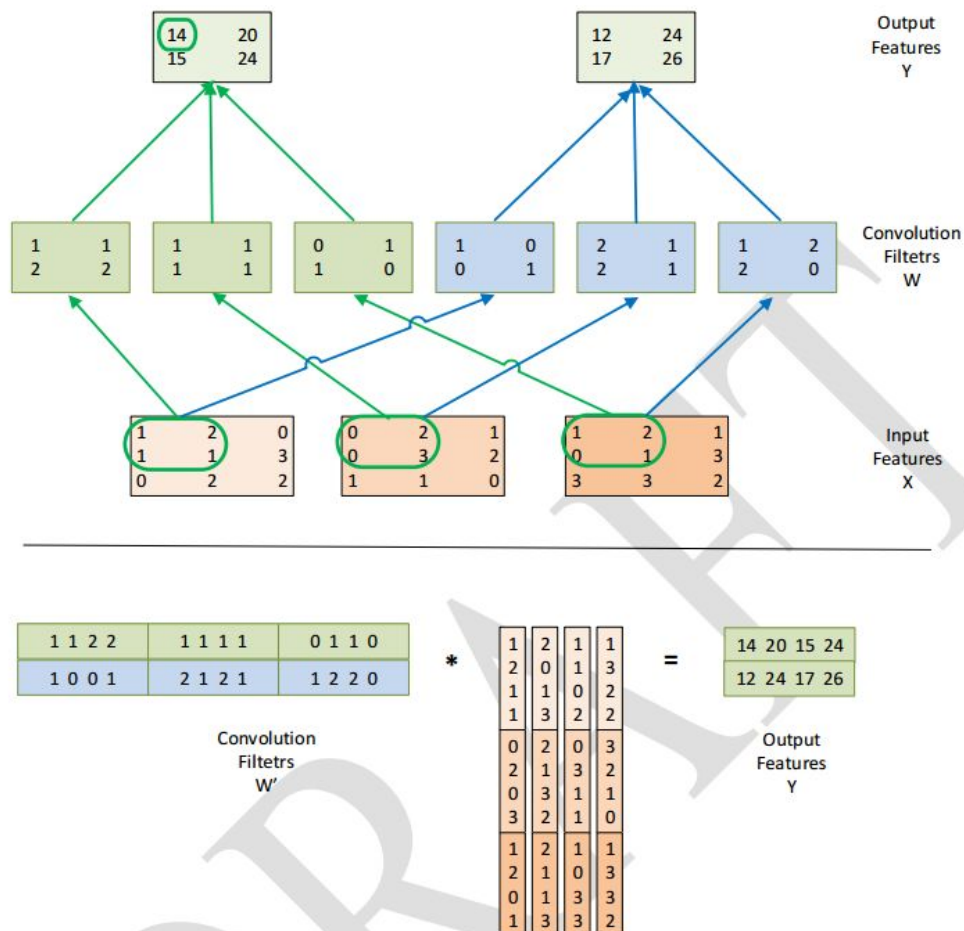


Figure 7. basic process of reducing convolution to matrix multiplication [1]

Our next optimization step is adding a new kernel to unroll the convolution kernels and the input data into two large matrices. The forward kernel will be responsible for the matrix multiplication. The matrix multiplication will be computed with tiled matrix multiplication discussed in our course.

We also discussed about some implementation details of the matrix multiplication strategy. According to our computation, the unrolled matrix can't be put into shared memory due to its large size. This might requires multiple extra accesses on global memory. A possible solution to this problem is combining the two steps in one kernel and create unrolled matrix on the fly and do the computation without writing the unrolled matrix to global memory. Besides, we try to use streams to perform concurrent computation in this combined kernel

(since the computation of each unrolled sub-matrix doesn't rely on the results of others). In the multiplication part, we also use transpose to make adjacent threads are coalesced. This part might be further optimized with non-square tile size. If the tile size is well designed, we may avoid divergence in the computation.

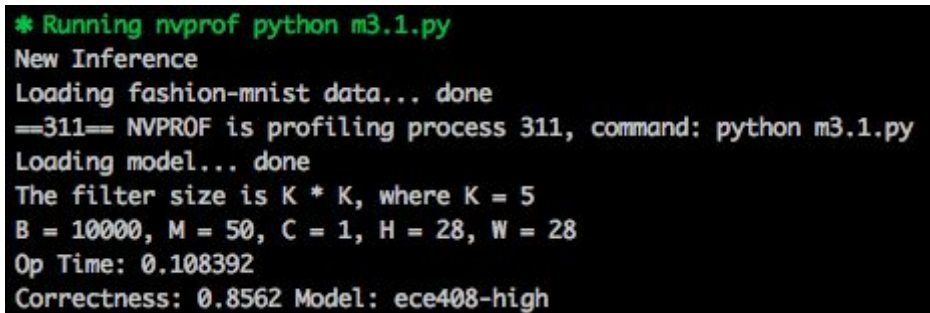
Note that this is just a plan of optimization. Some details might be changed during implementation.

4 Optimization

4.1 Unroll

Unrolling will reduce the forward operation of the convolutional layer to one large general matrix-matrix multiplication. For the unroll process, it benefits in that when the image is too big for shared memory, there are input elements which will be loaded repeatedly inevitably.

We unroll the kernels and the input data into two big matrix. Since the kernel size is $5 * 5$, with 50 kernels in total, and the input size is $28 * 28 * 10000$ (batch size), the dimension of unrolled kernel matrix would be 50×25 , and dimension of unrolled input matrix for each image is $25 * 576$ (where $576 = 24 * 24$ according to dimension of output image). The issue of the implementation in the textbook is that it writes the unrolled kernel matrix and input matrix into global memory, and then performs matrix multiplication using the unrolled matrices, which leads to **multiple times of global memory reads and writes** and would increase the running time dramatically. The result of naive unroll implementation is shown in Figure 8. **The time is 108.3 ms.**



```
* Running nvprof python m3.1.py
New Inference
Loading fashion-mnist data... done
==311== NVPROF is profiling process 311, command: python m3.1.py
Loading model... done
The filter size is K * K, where K = 5
B = 10000, M = 50, C = 1, H = 28, W = 28
Op Time: 0.108392
Correctness: 0.8562 Model: ece408-high
```

Figure 8. Result of naive unroll implementation

Here we propose an unroll-based matrix multiplication which guarantees **reading from input and writing into global memory result only once**.

To unroll kernel, the strategy is similar to that in the textbook chapter, with the only difference that the kernel matrix is loaded into shared memory. To unroll input data (image), the position of each element in the unrolled matrix is decided by the position of the element in the original data, as well as the relative position within the $5 * 5$ kernel.

For a single element in the original input matrix, it can appear multiple times in the unrolled input matrix, hence, we only need to load it once and write it into multiple positions in unrolled input matrix. When the kernel slides, we record the relative position of the element inside the kernel as (i, j) and the position of upper-left element inside the kernel as $(topRow, topCol)$. Then such element can be uniquely decided in the unrolled input matrix. $(topRow,$

topCol) is used to determine which column this element should be placed and (i, j) is used to determine which row this element should be placed in the unrolled input matrix.

Take Figure 9 as an example: for the same element with blue color, when the kernel slides, we show two pairs of (i, j) and (topRow, topCol). We assume the coordinates of the element in the original matrix is (x,y), then $\text{topR} = x - 2$, $\text{topCol} = y - 1$. Thus we put the element in the unrolled matrix, with column index $x_{\text{unroll}} = (x - 2) * 24 + (y - 1)$ and row index $y_{\text{unroll}} = 4 * 5 + 2 = 22$. In the second pair, $\text{topRow} = x$, $\text{topCol} = y$. We put the element in the unrolled matrix, with column index $x_{\text{unroll}} = (x) * 24 + (y - 1)$ and row index $y_{\text{unroll}} = 1 * 5 + 1 = 6$. In this way, we find that each element-kernel pair is uniquely put into the position in the unrolled input matrix with only once global memory read.

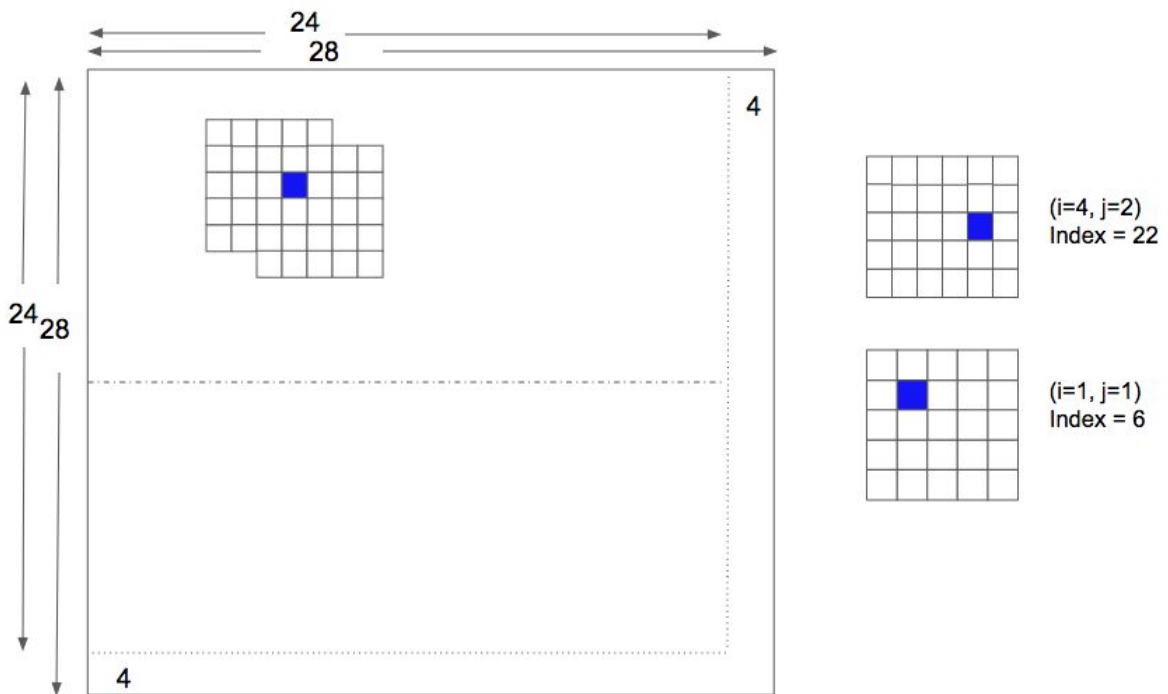


Figure 9 Put the unrolled element into the correct position uniquely

After kernels and part of input matrix are unrolled, we can perform general matrix multiplication immediately and write the partial result into output. Here, each thread is required to sequentially calculate several output elements which is different from unroll approach in textbook chapter 16.

The unrolled matrices and multiplication process are displayed in Figure 10.

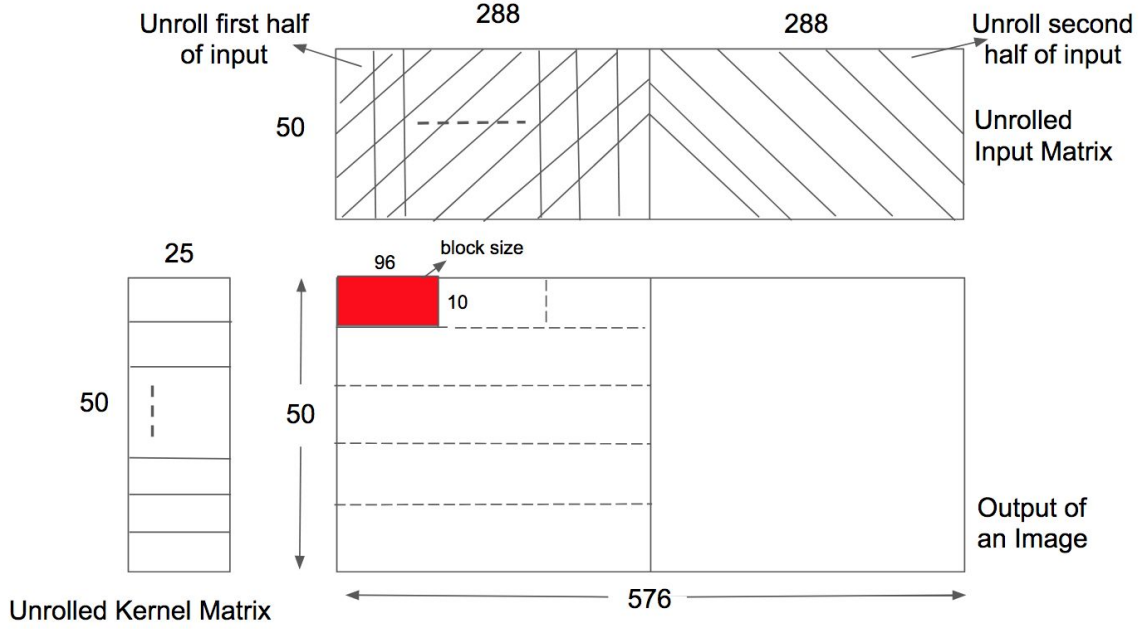


Figure 10 Unrolled matrix multiplication

Due to the limitation of shared memory and block size, we can not compute single image at a time. We load entire unrolled kernel matrix but half of the unrolled input matrix into share memory, with `__shared__ float tileA[1250]` and `__shared__ float tileB[7200]` (tileA stores unrolled kernel matrix, tileB stores half of unrolled input matrix). The maximum thread number within a block is 1024. Thus, we set the block size as 10X96. The reason for choosing 10X96 is: (a) 288 is multiple of 96. (b) The memory interface of Pascal GPU is 4096-bit HBM2, which is equivalent to 128 float elements. With dimension of 10 * 96, we can improve utilization of global memory interface from 25% to 75%. In our implementation, each thread is required to sequentially compute $3 * 5 = 15$ elements for first half of unrolled input matrix. The result would be written into output tensor immediately. Based on modified unroll algorithm the operation time could be reduced significantly as shown in Figure 11. **The shortest time we can achieve for the modified unroll algorithm is 22.8ms.**

```
* Running nvprof python m3.1.py
New Inference
Loading fashion-mnist data... done
==311== NVPROF is profiling process 311, command: python m3.1.py
Loading model... done
Op Time: 0.022810
Correctness: 0.8562 Model: ece408-high
```

Figure 11. The result of unroll algorithm we actually implemented

To further optimize the performance of our convolution, we also consider occupancy of resources. The definition of occupancy is presented in equation (1). According to reference [2], resources allocated for entire block are finite. If a thread uses too many resources, then

$$\text{Occupancy} = \text{Active Warps} / \text{Maximum Active Warps} \quad (1)$$

the occupancy is limited. The resource limit includes register usage, shared memory usage and block size. Reference [2] suggests that an occupancy of 66% is a good choice. We used [3] provided by Nvidia to estimate the occupancy of our cuda kernel.

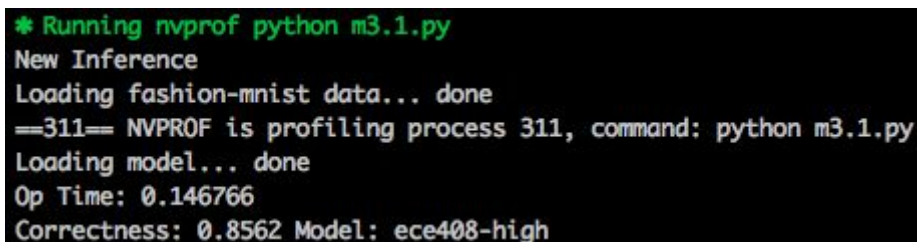
4.2 Optimization with “true” convolution

According to reference [4], some of the GPU convolution libraries still use real convolution instead of unroll. This strategy is also being studied by different research groups. This exhibits the possibility of achieving faster speed beyond tiled convolution discussed in class. In this section, we present our optimization strategy and experiment results in optimizing “true” convolution.

First, we use comment to determine the time consumption of each operation in our kernel code. We find out that the global memory read and write consumes a large portion of execution time. Since the input image size is relatively small, a possible optimization is further reduce the global access by loading whole images into shared memory and do all the convolution related to these images in a block. This will reduce the input load to exactly once. This idea is also inspired by one of the strategies proposed in [4] where load is also reduced to about exactly once but in a more complicated manner.

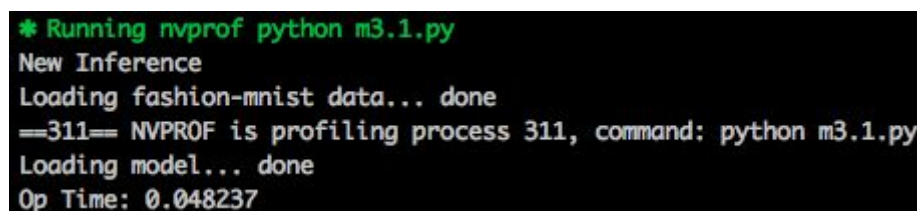
Our first step is implementing this strategy directly. In this implementation, each block loads 10 images to fully utilize the shared memory. The convolution filter weights are loaded into constant memory. In our first execution, we got an error saying that we are running out of registers. According to our search online, this is the result of cuda compiler optimization. All the for loop in kernel code are optimized (unroll loop, i.e. serialized). The unrolled for loop requires about iteration-number times more registers. So we remove this optimization in the output channel level for loop (50 output channels) with **#pragma unroll 1**.

The following nvprof result shows the performance before and after this optimization. Note that in the optimized kernel, each block will load 10 input images. This is almost the maximum number of input images possible in shared memory.



```
* Running nvprof python m3.1.py
New Inference
Loading fashion-mnist data... done
==311== NVPROF is profiling process 311, command: python m3.1.py
Loading model... done
Op Time: 0.146766
Correctness: 0.8562 Model: ece408-high
```

Figure 12. Convolution kernel with constant memory performance before optimization



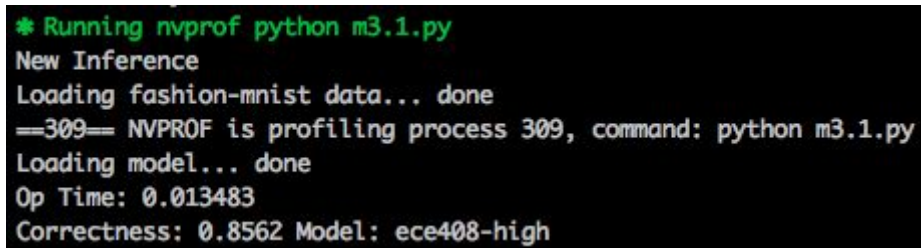
```
* Running nvprof python m3.1.py
New Inference
Loading fashion-mnist data... done
==311== NVPROF is profiling process 311, command: python m3.1.py
Loading model... done
Op Time: 0.048237
```

Figure 13. Convolution kernel performance after optimization, unroll = 1

According to Figure 12 and Figure 13, the kernel performance is improved from **146.7ms** to **48.2ms**. This is the result of reduced global memory access. In tiled matrix multiplication, the

load part of different tiles are overlapped at the halo part. These halo values cannot be shared among different blocks. In our previous tiled convolution, we use `tile_width = 12`. Thus we need four tile load for each image. We have to load $4 * (12 + 5 - 1) * (12 + 5 - 1) = 1024$ floats. Each value is loaded $1024 / 28 * 28$ times. This value is about 1.3. Besides, the original kernel has an occupancy of more than 80%, which also have a bad impact on the performance. (details about occupancy is discussed in previous section)

Next we tried more unroll values. It seems that for a relatively large range (we tried most values from 12 to 25), the performance could be further improved to about **13.4ms**! Figure 14 presents the nvprof result when **unroll = 15**.



```
* Running nvprof python m3.1.py
New Inference
Loading fashion-mnist data... done
==309== NVPROF is profiling process 309, command: python m3.1.py
Loading model... done
Op Time: 0.013483
Correctness: 0.8562 Model: ece408-high
```

Figure 14. Convolution kernel performance after optimization, unroll = 15

This large improvement step should be the combined result of occupancy and compiler optimization. We analyze this by looking into our kernel resource usage. Each block uses $1000 * 28 * 28 * 4$ byte shared memory. This value is close to the maximum shared memory available in each block. Thus only one block could be fitted into one SM. Since each SM has a large number of registers, the register resources will be wasted with `unroll = 1`. A suitable unroll value will improve the resource usage. Besides, a suitable unroll value might help to make the occupancy close 66.6%, which is the suggested value by Nvidia. Due to time limitation, we didn't look into the assembly code for the exact improved part of assembly code with more register usage.

Finally, we tweak the block size and number of input feature map per block based on aforementioned convolution design. Since each warp has 32 threads and the dimension of output is $24 * 24 = 576$, we choose $9 * 64$ block dimension instead of original size to ensure coalesced access without divergence when writing into global memory. Also to further reduce the occupancy, we let each thread process 25 input feature maps in two sequential stages. First, kernel function loads 13 input feature maps into shared memory, performs convolution and writes into global memory. Second, kernel function loads extra 12 input feature images overriding shared memory, performs convolution and write operation. In this implementation, the **#pragma unroll 15** instruction is adopted to achieve best performance. The result is listed in Figure 15. The operation time is slightly optimized.

```

* Running nvprof python final.py
New Inference
Loading fashion-mnist data... done
==312== NVPROF is profiling process 312, command: python final.py
Loading model... done
Op Time: 0.010970
Correctness: 0.8562 Model: ece408-high
==312== Profiling application: python final.py
==312== Profiling result:

```

Figure 15. Convolution kernel with $9 * 64$ block dimension, unroll = 15

4.3 Summary of the optimization performance

In this section, we summarize the optimization results. The performance of our kernels is presented in Table 1. We consider the following aspects during optimization: (1) exact once read and write of global memory; (2) resources usage and occupancy; (3) cuda compiler optimization; (4) advanced convolution algorithm.

Table 1. Kernel performance summary

Strategy	Performance
Basic tiled convolution	163.3 ms
Tiled convolution with constant memroy	146.7 ms
Basic unrolled kernel	108.3ms
Unrolled kernel with exactly once load	22.8 ms
Convolution with exactly once load, unroll = 1	48.2 ms
Convolution with exactly once load, unroll = 15 (parameter chosen from experiment)	9.4 ms (Fastest)

Contribution: All the members thoroughly discussed the problem and distributed the task reasonably. The tasks are distributed as following

Milestone 1 : experiments done by Yingyi Zhang(yingyiz2), Guxin Jin(gjin7) and Guanchen He(ghe10).

Milestone 2: CPU code by Guanchen He(ghe10), experiment done by Yingyi Zhang(yingyiz2), report wrote by Guxin Jin(gjin7).

Milestone 3: GPU code by Yingyi Zhang(yingyiz2), optimization with constant memory done by Guanchen He(ghe10), report wrote by Guxin Jin(gjin7) and Guanchen He(ghe10).

Milestone final: Basic unroll kernel by Guanchen He(ghe10) and Yingyi Zhang(yingyiz2) and Guxin Jin(gjin7). Optimized unrolled kernel code by Yingyi Zhang (yingyiz2). Optimized convolution kernel with exactly once load and optimization with true convolution by Guanchen He (ghe10). The optimization plan was discussed and proposed by Yingyi

Zhang(yingyiz2), Guxin Jin(gjin7) and Guanchen He(ghe10). Report is written by Guanchen He(ghe10) and Guxin Jin(gjin7).

Reference

- [1] Textbook Chapter 16 from course website. URL:<https://wiki.illinois.edu/wiki/display/ECE408Fall2017/Textbook+Chapters>
- [2] URL:http://on-demand.gputechconf.com/gtc-express/2011/presentations/cuda_webinars_WarpsAndOccupancy.pdf
- [3] URL: https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&cad=rja&uact=8&ved=0ahUKEwjW_e7ygljYAhWNQ98KHdJvAoEQFggxMAE&url=https%3A%2F%2Fdeveloper.download.nvidia.com%2Fcompute%2Fcuda%2FCUDA_Occupancy_calculator.xls&usg=AOvVaw3C1_WHEkOfxeH1sjzxGYB5
- [4] Chen X, Chen J, Chen D Z, et al. Optimizing Memory Efficiency for Convolution Kernels on Kepler GPUs[J]. arXiv preprint arXiv:1705.10591, 2017.