# ECE 408 - Final Project Report

School: UIUC

Team: wen_mei_we_clutch_that_a

Jacob Brown, jlbrown5 ◇ Brian Yan, yyan18 ◇ Brandon Yu, byu23

## Milestone 1

- Kernels that collectively consume more than 90% of the program time.

| Time (%) | Time | Name |
|---|---|---|
| 34.07 | 118.49ms | fermiPlusCgemmLDS128_batched |
| 27.00 | 93.897ms | cudnn::detail::implicit_convolve_sgemm |
| 12.70 | 44.164ms | fft2d_c2r_32x32 |
| 8.20 | 28.518ms | sgemm_sm35_ldg_tn_128x8x256x16x32 |
| 6.43 | 22.368ms | [CUDA memcpy HtoD] |
| 4.08 | 14.180ms | cudnn::detail::activation_fw_4d_kernel |

- API calls that collectively consume more than 90% of the program time.

| Time (%) | Time | Name |
|---|---|---|
| 43.66 | 1.93s | cudaStreamCreateWithFlags |
| 26.90 | 1.19s | cudaFree |
| 20.615 | 911.5ms | cudaMemGetInfo |

- Difference between kernels and API calls

  API calls are calls to the cuda api defined in cuda.h. These functions are usually called by the host to set up the kernel and get information about the GPU. All the API calls listed in the table above are high level CUDA runtime API calls, which are built on lower level CUDA driver APIs. Those CUDA runtime API calls make it easier for us to compile our kernels into executables.

  Kernels are user programs that run on the GPU. These are more traditional functions that do the work needed by the user, such as vector addition. However, unlike traditional C functions, CUDA kernels run N (number of threads) times per invocations instead of only once. Which is essentially what makes parallel computing possible.

- Output of rai running MXNet on the CPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8444}
```

- Program run time (On CPU)

```
12.77user 6.12system 0:08.42elapsed 224%CPU (0avgtext+0avgdata 2826068maxresident)k
0inputs+2624outputs (0major+39593minor)pagefaults 0swaps
```

- Output of rai running MXNet on the GPU

```
Loading fashion-mnist data... done
Loading model...
[05:18:37] src/operator/././cudnn_algoreg-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
New Inference
EvalMetric: {'accuracy': 0.8444}
```

- Program run time (On GPU)

```
2.12user 1.11system 0:02.71elapsed 119%CPU (0avgtext+0avgdata 1134872maxresident)k
0inputs+512outputs (0major+154708minor)pagefaults 0swaps
```

## Milestone 2

- Full Program Run time

| Number of images | User Time (s) | System Time (s) | Elapsed Time (s) |
|---|---|---|---|
| 10000 | 30.39 | 1.41 | 29.80 |
| 100 | 1.04 | 0.49 | 1.01 |
| 10 | 0.75 | 0.52 | 0.74 |

- Op Times

| Number of images | Convolutional Layer 1 Op Time (s) | Convolutional Layer 2 Op Time (s) |
|---|---|---|
| 10000 | 6.499919 | 19.373818 |
| 100 | 0.064483 | 0.193488 |
| 10 | 0.006535 | 0.019293 |

# Milestone 3

For the GPU implementation of forward convolution, we decided to unroll the input images and use matrix multiplication to get the output data. We run a sequential loop for every image in the batch that first unrolls the data via the "unroll_Kernel" and then multiplies the unrolled matrix with the filter matrix via the "matrixMultiply" kernel.

Below we present the overall run times as well as an analysis of which kernels and API calls take up the most during execution for each batch size.

## Full Program Run Times

| Number of images | User Time (s) | System Time (s) | Elapsed Time (s) |
|---|---|---|---|
| 10000 | 3.34 | 1.80 | 4.61 |
| 100 | 1.55 | 0.98 | 2.01 |
| 10 | 1.58 | 0.83 | 1.95 |

## NVProfs

### BATCH SIZE = 10

- Kernels that take more than 1% of the program run time.

| Time (%) | Time | Name |
|---|---|---|
| 54.95 | 2.33ms | matrixMultiply |
| 20.04 | 850us | [CUDA memcpy HtoD] |
| 10.92 | 463us | unroll_kernel |
| 5.98 | 253us | void sgemm_largek_lds64 |
| 4.26 | 181us | MapPlanKernel |

- API calls that takes more than 1% of the program run time.

| Time (%) | Time | Name |
|---|---|---|
| 41.18 | 1.43s | cudaStreamCreateWithFlags |
| 32.14 | 1.12s | cudaFree |
| 26.05 | 906.67ms | cudaMemGetInfo |

## BATCH SIZE = 100

- Kernels that take more than 1% of the program run time.

| Time (%) | Time | Name |
|---|---|---|
| 76.10 | 23.36ms | matrixMultiply |
| 15.28 | 4.69ms | unroll_kernel |
| 3.56 | 1.09ms | [CUDA memcpy HtoD] |
| 2.29 | 704.25us | void sgemm_largek_lds64 |

- API calls that takes more than 1 of the program run time.

| Time (%) | Time | Name |
|---|---|---|
| 40.8 | 1.44s | cudaStreamCreateWithFlags |
| 32.24 | 1.14s | cudaFree |
| 25.56 | 906.37ms | cudaMemGetInfo |

## BATCH SIZE = 10000

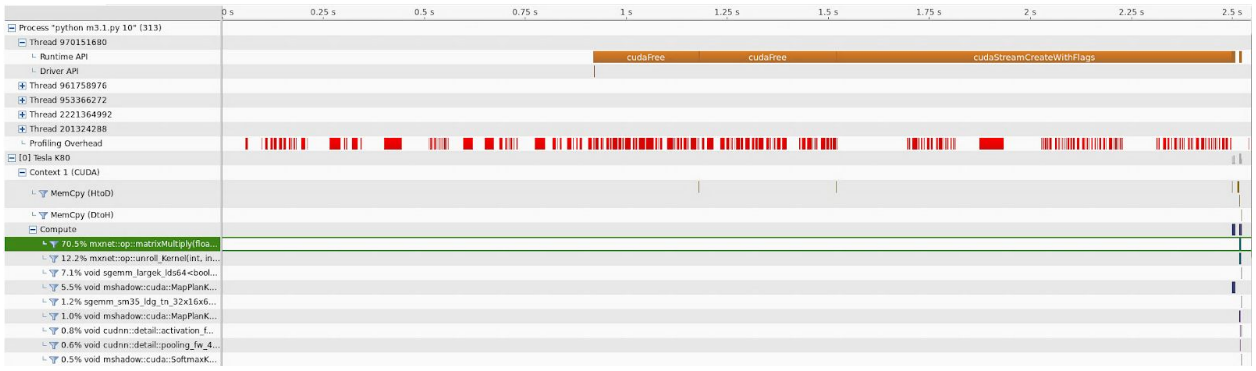
- Kernels that take more than 1% of the program run time.

| Time (%) | Time | Name |
|---|---|---|
| 80.17 | 2.33s | matrixMultiply |
| 16.12 | 469.65ms | unroll_kernel |

- API calls that takes more than 1% of the program run time.

| Time (%) | Time | Name |
|---|---|---|
| 43.25 | 3.15s | cudaDeviceSynchronize |
| 19.72 | 1.14s | cudaStreamCreateWithFlags |
| 15.28 | 1.11s | cudaFree |
| 12.50 | 910.83ms | cudaMemGetInfo |
| 4.27 | 310.69ms | cudaLaunch |

NVVP for Datasize 10


0 s
0.25 s
0.5 s
0.75 s
1 s
1.25 s
1.5 s
1.75 s
2 s
2.25 s
2.5 s
Process "python m3.1.py 10" (313)
Thread 970151680
Runtime API
Driver API
Thread 961758976
Thread 953366272
Thread 2221364992
Thread 201324288
Profiling Overhead
[0] Tesla K80
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
70.5% mxnet::op::matrixMultiply(floa...
12.2% mxnet::op::unroll_Kernel(int, in...
7.1% void sgemm_largek_lds64<bool...
5.5% void mshadow::cuda::MapPlanK...
1.2% sgemm_sm35_ldg_tn_32x16x6...
1.0% void mshadow::cuda::MapPlanK...
0.8% void cudnn::detail::activation_f...
0.6% void cudnn::detail::pooling_fw_4...
0.5% void mshadow::cuda::SoftmaxK...
cudaFree
cudaFree
cudaStreamCreateWithFlags

For small datasets, a large majority of the compute time is spent on CUDA API calls instead of doing useful work in the kernel.

## Next Steps

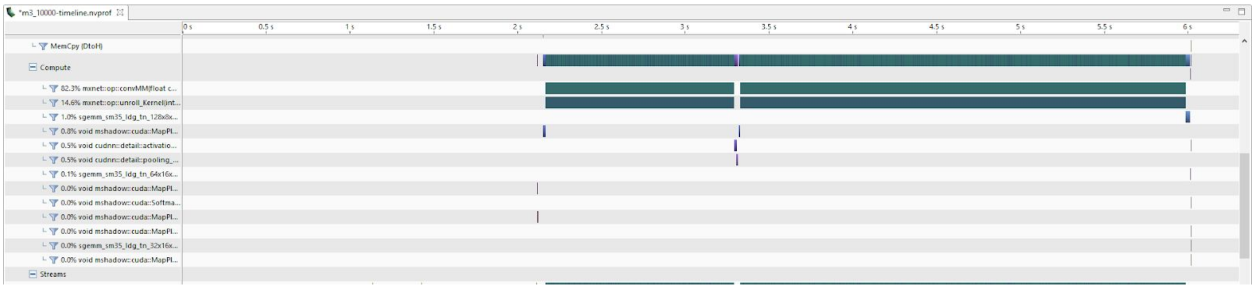
Concerning future optimizations, there is much room for improvement with our current design.

One optimization we plan to implement is putting the filters into constant memory. Since the filter data is constant throughout the entire computation and currently stored in global memory, utilizing constant memory can give a significant performance boost for a sequential loop that executes thousands of times.

Additionally, we can look into exploiting the parallelism in unrolling the input images. Right now we sequentially unroll all of the input images, even though they're independent of one another. Clearly there is room to more efficiently handle this process, and doing so would be highly beneficial to our model.

# Milestone 4

## Baseline - A look at Milestone 3 Performance


*m3_10000-timeline.nvprof
0 s
0.5 s
1 s
1.5 s
2 s
2.5 s
3 s
3.5 s
4 s
4.5 s
5 s
5.5 s
6 s
MemCpy (DtoH)
Compute
82.3% mxnet::op::convMM(float c...
14.6% mxnet::op::unroll_Kernel(int...
1.0% sgemm_sm35_ldg_tn_128x8x...
0.8% void mshadow::cuda::MapPl...
0.5% void cudnn::detail::activatio...
0.5% void cudnn::detail::pooling_...
0.1% sgemm_sm35_ldg_tn_64x16x...
0.0% void mshadow::cuda::MapPl...
0.0% void mshadow::cuda::Softma...
0.0% void mshadow::cuda::MapPl...
0.0% void mshadow::cuda::MapPl...
0.0% sgemm_sm35_ldg_tn_32x16x...
0.0% void mshadow::cuda::MapPl...
Streams

Above is our unmodified code from milestone 3 ran on the default batch size (10000 images). The NVVP window has be scrolled to focus on the computation kernels and their respective runtimes. Our two kernels consists of the `convMM` kernel for matrix multiplication and the `unroll_Kernel` for unrolling the image data.

Our current convolution method, as stated above, unrolls each image to perform matrix multiplication with the filter bank data. We can see from NVVP that it's the matrix multiplication kernel that is taking up the majority of the computation time (about 80%), with the second time hog being the unrolling kernel, which takes about 15% of the total computation time.

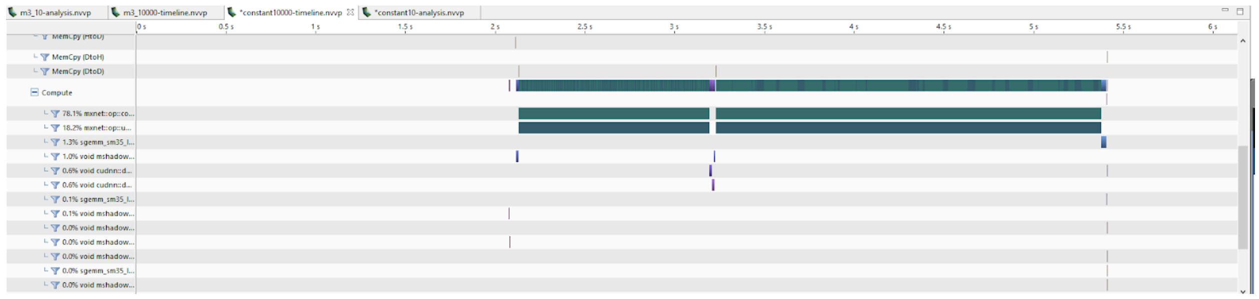
Additionally, from `nvprof` we see that our `cudaDeviceSynchronize` calls take up 42.82% of the total "API Calls" time.

In total, our implementation takes about 2.3 seconds, with .7 seconds spent on the first convolution layer and 1.6 seconds spent on the second.

We use these observations to optimize our forward convolution methods, as we discuss below.

## Optimization 1 - Filter Banks in Constant Memory

The following data and screenshots result from running on the default batch size (10000 images).



- Op Times (using `/usr/bin/time`)
  - First Layer: 0.675961s
  - Second Layer: 1.286228s
- Our kernels in nvprof

| Time (%) | Time | Name |
|---|---|---|
| 77.22 | 1.748 s | convMM |
| 17.99 | 407.18 ms | unroll_Kernel |

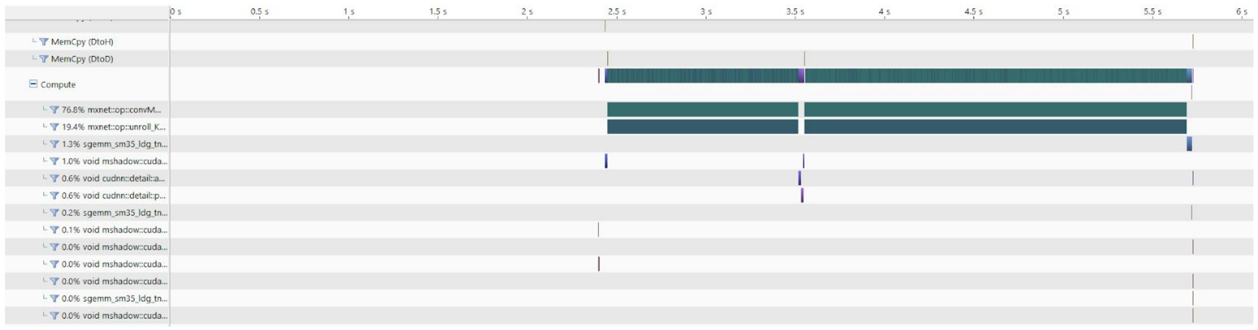
For this optimization, we initialized constant memory on the device that was large enough to hold the maximum amount of filter data (2400 elements in the second convolution layer). We then replaced the `A` parameter to the matrix multiply kernel (now `convMM`) with calls to the constant memory data `Kc`.

Before the filter data would just be passed like the unrolled input data through global memory. In the case of the second convolution layer, this would result in 2400 global memory reads per image just to load the filter data, which is the same for every image, into shared memory. With constant memory we avoid this, saving millions of cycles across the entire computation.

As seen in the NVVP window above, the total time for computation was reduced by about 600 ms. This is more explicitly clear in the nvprof output, where the matrix multiply kernel takes about 1.7 seconds, when it used to take 2.3 seconds.

To acknowledge our failures for this optimization, we had mistakenly attempted to load the constant memory into shared memory within the matrix multiply kernel. This essentially was just replacing the single global read from `A` with a constant memory read from `Kc`. This step did not offer a significant performance boost, and after further consideration we realized we could skip that step entirely, and instead access `Kc` directly when calculating the output value. This resembles more of the naïve matrix multiply strategy, but works in our favor because it removes overhead and unnecessary resource usage.

## Optimization 2 - Shared Memory in Unroll Kernel


0 s
0.5 s
1 s
1.5 s
2 s
2.5 s
3 s
3.5 s
4 s
4.5 s
5 s
5.5 s
6 s
MemCpy (DtoH)
MemCpy (DtoD)
Compute
76.8% mxnet::op::convM...
19.4% mxnet::op::unroll_K...
1.3% sgemm_sm35_ldg_tn...
1.0% void mshadow::cuda...
0.6% void cudnn::detail::a...
0.6% void cudnn::detail::p...
0.2% sgemm_sm35_ldg_tn...
0.1% void mshadow::cuda...
0.0% void mshadow::cuda...
0.0% void mshadow::cuda...
0.0% void mshadow::cuda...
0.0% sgemm_sm35_ldg_tn...
0.0% void mshadow::cuda...

- Op Times (using `/usr/bin/time`)
  - First Layer: 0.711s
  - Second Layer: 1.30s
- Our kernels in nvprof

| Time (%) | Time | Name |
|---|---|---|
| 76.03 | 1.749 s | convMM |
| 19.26 | 443.13 ms | unroll_Kernel |

This optimization uses shared memory in our unroll kernel. In our design, each block has 1024 threads, and each thread loads one element into shared memory.

During the unrolling process, if the element a thread is trying to access is not loaded into shared memory, the thread grabs the element from global memory.

As discussed in lecture, the unrolled matrix will have (H_out * W_out) * (K * K) elements, whereas the input image has (H_in * W_in) elements. With large enough images, this means that the average re-use of input elements is about (K * K) times. This arithmetic is what lead us to believe that shared memory could benefit our implementation, since we were pulling each element from global memory each time we needed it.

Sadly, we currently see no improvement in performance with shared memory usage within the unroll kernel. We hypothesize that the overhead of copying the data into shared memory and the fact that our accesses to shared memory are not coalesced are some drawbacks preventing an decrease in computation time.

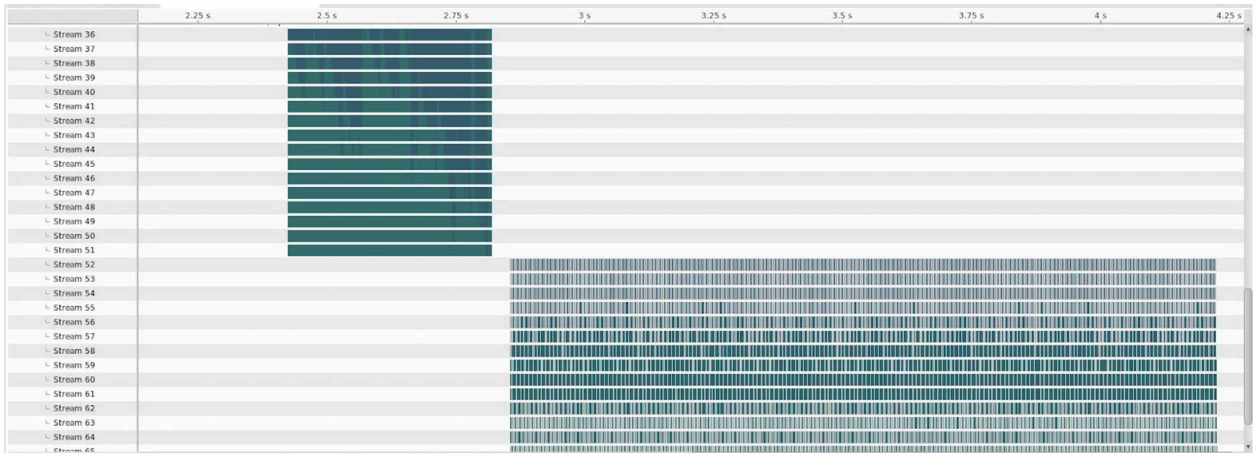
## Optimization 3 - Multiple Streams

- Op Times (using `/usr/bin/time`)
    - First Layer: 0.343s
    - Second Layer: 0.922s
- Our kernels in nvprof

| Time (%) | Time | Name |
|---|---|---|
| 71.61 | 2.00s | convMM |
| 24.56 | 0.688s | unroll_Kernel |

This optimization involves using multiple CUDA streams to work on each batch of images. Because each image is used independently of the others, each stream can take a image and perform our convolution method without issue. This should offer a performance increase because we are now doing more tasks concurrently, rather than sequentially.

There are two kernel calls in our convolution but they cannot be parallelized because the matrix multiplication kernel requires an unrolled matrix as input, so it needs to be serialized. With this in mind, we parallelize unrolling multiple images and then parallelize the matrix multiplication in the same manner.



With streaming we were able to save 0.6s on full program run time even though the execution time of the kernels stayed the same. Streaming also has another benefit, it decreases the amount of time synchronization by 2 seconds, which was a big part of the baseline time. However, there was extra overhead in creating and using the streams, which means the time saved wasn't exactly 2 seconds. The NVVP screencap above shows that the streams work as expected, 16 streams doing the unroll kernel and then 16 streams doing the matrix multiply kernel.

For future optimizations, we note that sixteen streams are only half of the maximum amount available with a Kepler GPU, so we expect to have greater performance once we increase the number of streams.