ECE 408 Milestone 3 Report

Baoming Wang(baoming2), **Jun Luo**(junluo2), **Peiyuan Yang**(py6)

10/10/2019

Team Name: hmachine

On-campus

Final Submission: Due October 19, 2019

Introduction:

This report is to introduce the work for ECE 408 milestone 3. In this milestone, we used GPU kernel to build CNN convolution layers. In m3.1, we use GPU kernel to build convolution layers instead of CPU in m2.1.

Output running MXNet:

- Running python m3.1.py

```
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.024381
Op Time: 0.088384
Correctness: 0.7653 Model: ece408
```

- Running python m3.1.py 1000

```
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.002463
Op Time: 0.008877
Correctness: 0.767 Model: ece408
```

- Running python m3.1.py 100

```
Loading fashion-mnist data... done
Loading model... done
```

```
New Inference
Op Time: 0.000278
Op Time: 0.000929
Correctness: 0.76 Model: ece408
```

CUDA Code:

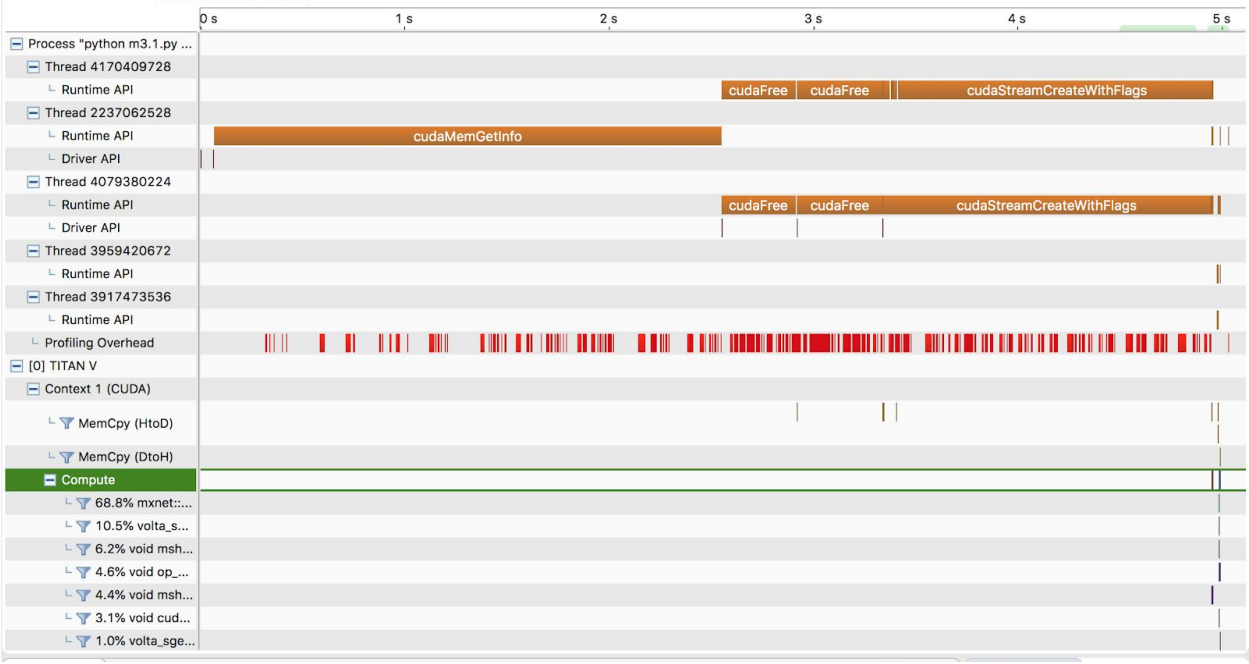
```
#ifndef MXNET_OPERATOR_NEW_FORWARD_CUH_
#define MXNET_OPERATOR_NEW_FORWARD_CUH_

#define TILE_WIDTH 16

#include <mxnet/base.h>

namespace mxnet
{
namespace op
{

__global__ void forward_kernel( float *y, const float *x, const float
*k, const int B, const int M, const int C, const int H, const int W,
const int K, const int TILE_H, const int TILE_W) {

    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 b = blockIdx.z; int m = blockIdx.y;
    int tile_h = blockIdx.x/TILE_W;
    int tile_w = blockIdx.x%TILE_W;
    int h = tile_h*TILE_WIDTH + threadIdx.y;
```

```
    int w = tile_w*TILE_WIDTH + threadIdx.x;
    float acc = 0;
    if (h < H_out && w < W_out) {
        for (int c = 0; c < C; c++) {
            for (int p = 0; p < K; p++) {
                for (int q = 0; q < K; q++) {
                    acc += x4d(b, c, h + p, w + q) * k4d(m, c, p, q);
                }
            }
        }
        y4d(b, m, h, w) = acc;
    }

#undef y4d
#undef x4d
#undef k4d
}

template <>
void forward<gpu, float>(mshadow::Tensor<gpu, 4, float> &y, const
mshadow::Tensor<gpu, 4, float> &x, const mshadow::Tensor<gpu, 4,
float> &w) {
    // Extract the tensor dimensions into B,M,C,H,W,K
    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 = w.shape_[3];

    const int H_out = H - K + 1;
    const int W_out = W - K + 1;
    // Set the kernel dimensions
    int TILE_CNT_H = ceil(H_out/(TILE_WIDTH*1.0));
    int TILE_CNT_W = ceil(W_out/(TILE_WIDTH*1.0));
    dim3 gridDim(TILE_CNT_H*TILE_CNT_W, M, B);
    dim3 blockDim(TILE_WIDTH, TILE_WIDTH, 1);
```

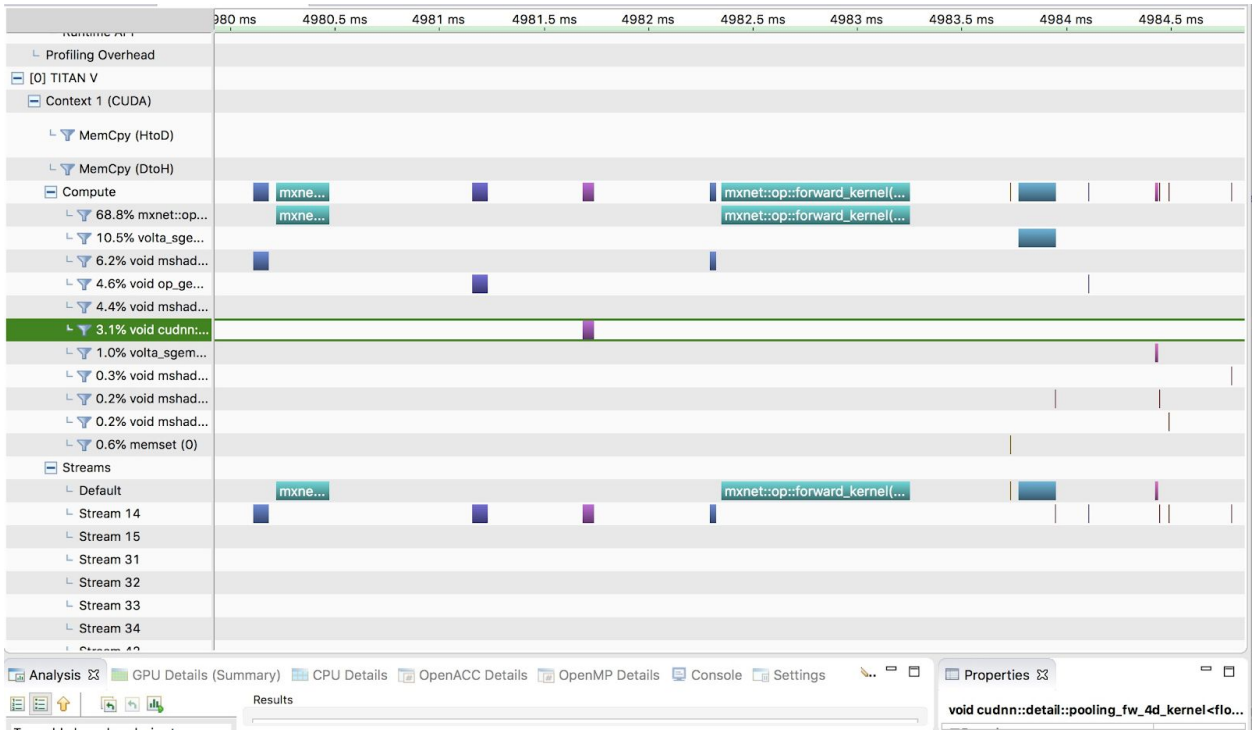
```
    // Call the kernel
    forward_kernel<<<gridDim, blockDim>>>(y.dptr_,x.dptr_,w.dptr_,
B,M,C,H,W,K,TILE_CNT_H,TILE_CNT_W);

    // Use MSHADOW_CUDA_CALL to check for CUDA runtime errors.
    MSHADOW_CUDA_CALL(cudaDeviceSynchronize());

}
}
}

#endif
```

NVPROF Analyzation:

This is the kernel analyzation we used (forward_kernel). In this graph, we use convolution layers for 100 pictures. As what we see, in total, most of the time are used on cuda memory setting and free. Because we used GPU kernel, the time cost of computation is about 1%.



For the computation, we can find that the mxnet function used about 68.8% of time because we use that for convolution calculating.



However, in the following picture, we can find that there are some control divergence happens. It costs about 11% of computation time for dealing with control divergence. We have some space to optimize this function, such as decreasing control divergences.

The following kernels are ordered by optimization importance based on execution time and achieved occupancy.

Rank	Description
100	[1 kernel instances] mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int,
28	[1 kernel instances] mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int,
19	[1 kernel instances] volta_sgemm_32x32_sliced1x4_tn
8	[1 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::e
8	[1 kernel instances] void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGeneric
5	[1 kernel instances] void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling
5	[2 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::e
3	[1 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::e
1	[1 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::e
1	[1 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::plusto, int=8, mshadow::e
1	[1 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::e
1	[1 kernel instances] void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshac
1	[1 kernel instances] void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGeneric
1	[1 kernel instances] volta_sgemm_128x32_tn
1	[10 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow:
1	[2 kernel instances] void mshadow::cuda::MapPlanKernel<mshadow::sv::saveto, int=8, mshadow::e

Conclusion:

From the result analyzation, we have a deep understanding for convolution layers in GPU. Because CPU is serialized, GPU has a much better performance in convolution layers.

ECE 408 Milestone 2 Report

Baoming Wang(baoming2), **Jun Luo**(junluo2), **Peiyuan Yang**(py6)

10/10/2019

Team Name: hmachine

On-campus

Final Submission: Due October 12, 2019

Introduction:
This report is to introduce the work for ECE 408 milestone 2. In this milestone, we installed RAI system and tried different code on RAI system. In m1.1 and m1.2, we compared and contrasted performances for the same code in GPU and CPU. In m2.1, we made a convolution code and tested its performance for different layer numbers (100, 1000, 10000).

1. Include a list of all kernels that collectively consume more than 90% of the program time.

	Time(%)	Name
1	**31.72%**	**[CUDA memcpy HtoD]**
2	**17.56%**	**volta_scudnn_128x64_relu_interior_nn_v1**
3	**16.98%**	**volta_gcgemm_64x32_nt**
4	**8.63%**	**void fft2d_c2r_32x32**
5	**7.71%**	**volta_sgemm_128x128_tn**
6	**6.49%**	**void op_generic_tensor_kernel**
7	**6.40%**	**void fft2d_r2c_32x32**
Total	**95.49%**	

2. **Include a list of all CUDA API calls that collectively consume more than 90% of the program time.**

	Time(%)	Name
1	41.83%	cudaStreamCreateWithFlags
2	33.40%	cudaMemGetInfo
3	21.02%	cudaFree
Total	96.25%	

3. **Include an explanation of the difference between kernels and API calls.**
   The API calls are all about memory set and control for the kernel variables. The kernels are the parallel portions of an application are executed on the device. The API calls only happen once, but the kernel calls are made by different threads in parallel.

   For the two different sections generated by *nvprof*:

   "GPU activities" is the timing information that represents the execution time on the **GPU.**

   "API calls" section is the timing that represents the execution time on the **host**.

4. **Show output of rai running MXNet on the CPU**
   Terminal output:

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
17.02user 4.48system 0:08.85elapsed 242%CPU (0avgtext+0avgdata
6046980maxresident)k
0inputs+2824outputs (0major+
1601620minor)pagefaults 0swaps
```

5. **List program run time(CPU Version)**
   17.02user 4.48system 0:08.85elapsed

6. **Show output of rai running MXNet on the GPU**
   Terminal Output:

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
5.14user 3.22system 0:04.72elapsed 177%CPU (0avgtext+0av
gdata 2981512maxresident)k
0inputs+1712outputs (0major+733933minor)pagefaults 0swaps
```

7. **List program run time(GPU Version)**
   5.14user 3.22system 0:04.72elapsed

8. **Create a CPU implementation**
   CUDA Code:

```
template <typename cpu, typename DType>
void forward(mshadow::Tensor<cpu, 4, DType> &y, const
mshadow::Tensor<cpu, 4, DType> &x, const mshadow::Tensor<cpu, 4,
DType> &k)
{
    /*
    Modify this function to implement the forward pass described in
Chapter 16.
    The code in 16 is for a single image.
    We have added an additional dimension to the tensors to support
an entire mini-batch
```

```
    The goal here is to be correct, not fast (this is the CPU
implementation.)
    */

    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];
    const int H_out = H-K+1;
    const 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];
                }
              }
            }
          }
        }
      }
    }
}
```

9. List whole program execution time

10000: 82.49user 8.20system 1:13.41elapsed
1000: 12.15user 3.02system 0:08.42elapsed
100: 4.64user 2.34system 0:01.96elapsed

10. List Op Times

10000: Op Time: 10.848974 Op Time: 59.013833
1000: Op Time: 1.081818 Op Time: 5.914613
100: Op Time: 0.117280 Op Time: 0.621784

Conclusion:
From this project milestone, we learned the differences between CPU and GPU while running the CUDA code. Furthermore, we had a deep understanding of the profit of GPU in CV calculation and the function of CUDA for convolution layers.