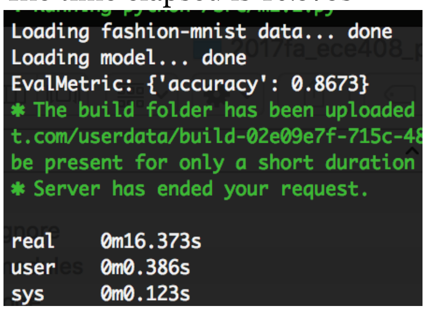
ECE408: Applied Parallel Programming	Fall 2017

Project Report

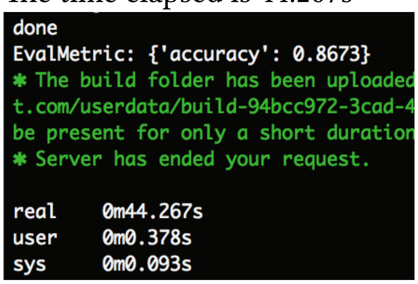
teamname: q80 — *Handed In: December 19, 2017*

1. **Milestone 1**

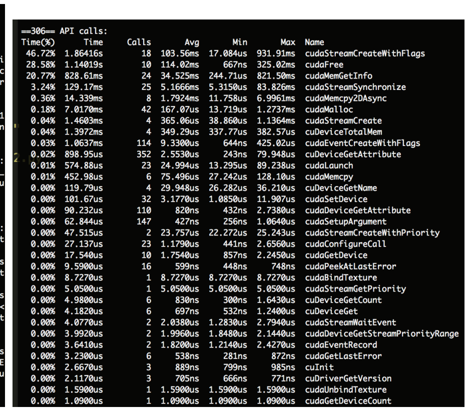
   (a) The time elapsed is 16.373s

   (b) The time elapsed is 44.267s

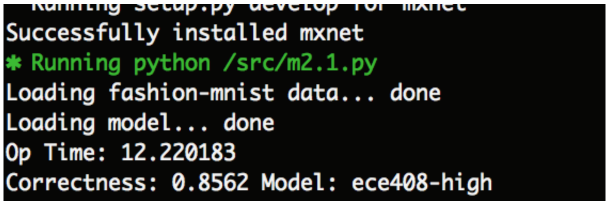
   (c) The result for nvprof is shown below

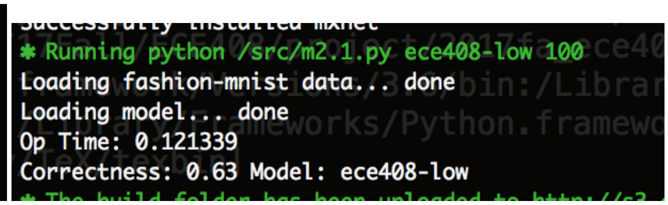
    

   We can see from the result that the most time consuming kernel in profile part in *implicit_convolve_sgemm*, *sgemm_sm35_ldg_tn_128x8x256x16x32* and *activation_fw_4d_kernel*. For the API call, the most consuming kernel is *cudaStreamCreateWithFlags*, *cudaFree* and *cudaMemGetInfo*.

2. **Milestone 2**

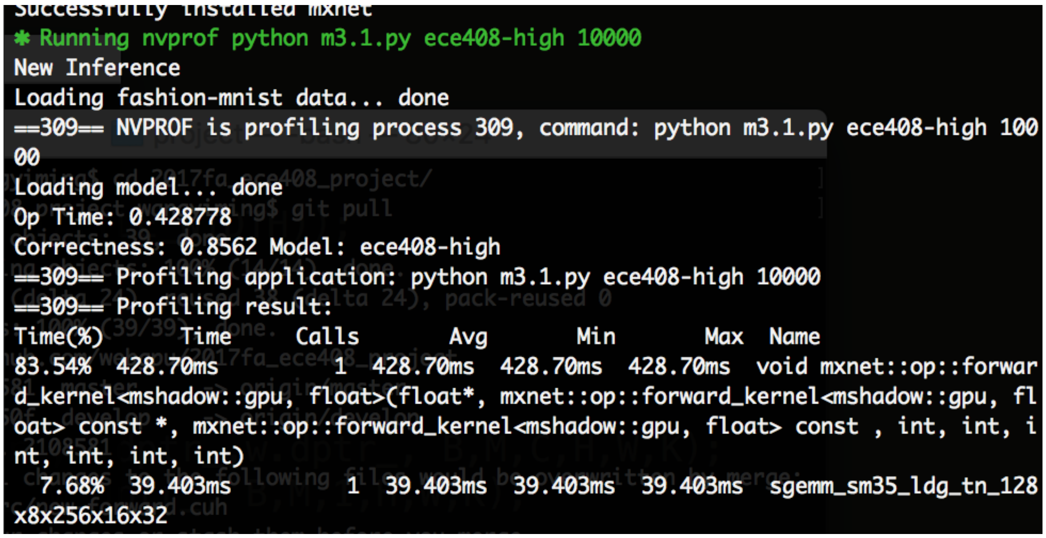
   The result is shown below.





3. **Milestone 3**

   The result is shown below.



   In this example, the forward layer took 0.4288 seconds, and the forward_kernel took 0.4287 seconds.

4. **Final step**

   (a) **Methods implemented for optimization**

   Step 1: Use shared memory to improve tile convolution

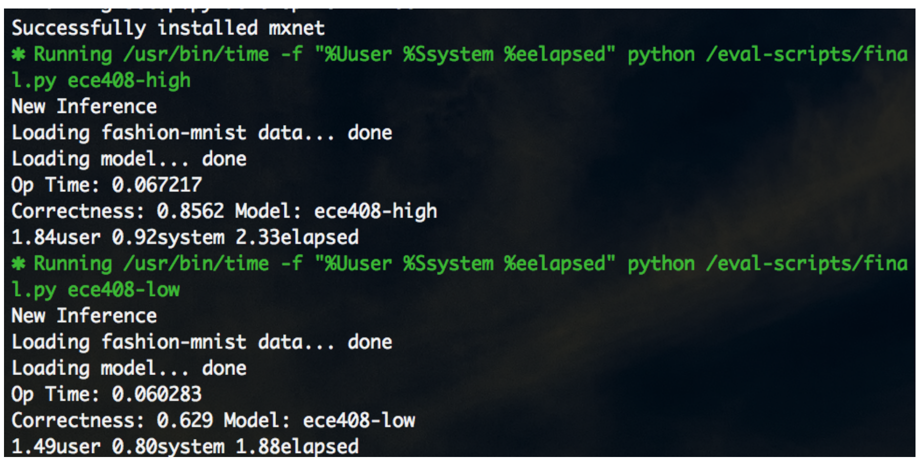
   In milestone 3, we implemented a naive version of tile convolution but without using any shared memory. To improve the performance, we defined a dynamic shared memory to store input tile and kernel. The size of input tile is (H_out + K -1) x (W_out + K - 1) + K_SIZE X K_SIZE. This will increase the performance a lot since we reduce the number of global memory access and the access to shared memory will be way faster. The Op time is about 140ns.

   Step 2: Add unroll and mapping to previous step

   We then consider all the optimization methods introduced in the course including using the shared memory to reduce the number of global read and write, use the constant memory to reduce the number of global read and write, avoid the control divergence, unroll the matrix to do the convolution operation so that the memory coalescing can be achieved, try to use as many threads in a thread block as possible.

Our second approach simply consists of two steps. First unroll the input matrix X, then perform tiled matrix multiplication and generate the result to the output matrix. In this approach we first unroll the input matrix of size [batch, num_filter, y, x] into [k*k,(y-k+1)*(x-k+1)*batch]. Then we do tiled matrix multiplication using the tile width of [25*32]. the reason of choosing this shared memory size is because the output of the unrolled matrix multiplication is [num_kernel,(y-k+1)*(x-k+1)*batch]. The number of rows of the matrix and the number of columns of the matrix are the integer multiples of the shared memory dimensions . 32 is also the number of threads in a warp so the control divergence is minimized. Lastly we map the result matrix [num_kernel,(y-k+1)*(x-k+1)*batch] back to the [batch, num_filter, (y-k+1), (x-k+1)] and write the result to output pointer.

(b) **Result for optimization**



```
Successfully installed mxnet
* Running /usr/bin/time -f "%Uuser %Ssystem %eelapsed" python /eval-scripts/fina
l.py ece408-high
New Inference
Loading fashion-mnist data... done
Loading model... done
Op Time: 0.067217
Correctness: 0.8562 Model: ece408-high
1.84user 0.92system 2.33elapsed
* Running /usr/bin/time -f "%Uuser %Ssystem %eelapsed" python /eval-scripts/fina
l.py ece408-low
New Inference
Loading fashion-mnist data... done
Loading model... done
Op Time: 0.060283
Correctness: 0.629 Model: ece408-low
1.49user 0.80system 1.88elapsed
```

(c) **Work distribution**

Our team meet every week to optimize the system. We use the Codepad to share our code and did the coding part together.