ECE 408 Final Project Report

Team Name: tbd

Yiqing Zhou (yiqing2)
Nuocheng Pan(np9)

April 17th, 2019

Milestone 1: RAI Setup

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

```
40.45%:   [CUDA memcpy HtoD]
20.32%:   implicit_convolve_sgemm
11.88%:   volta_cgemm_64x32_tn
7.07%:    op_generic_tensor_kernel
5.62%:    volta_sgemm_128x128_tn
5.61%:    fft2d_c2r_32x32
4.52%:    pooling_fw_4d_kernel
3.70% :   fft2d_r2c_32x32
```

CUDA API calls that collectively consume more than 90% of the program time:

```
42.61%    cudaStreamCreateWithFlags
34.35%    cudaMemGetInfo
21.02%    cudaFree
```

Explanation of difference between kernels and API calls:

Kernels are functions programed by users. Kernels are launched by host and run on devices. APIs are provided by CUDA runtime system and could be directly called by users.

CPU output and runtime: (runtime is bolded)

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8236}
8.98user 3.57system 0:05.07elapsed 247%CPU (0avgtext+0avgdata
2470144maxresid
ent)k
0inputs+2824outputs (0major+668695minor)pagefaults 0swaps
```

GPU output and runtime: (runtime is bolded)

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8236}
4.40user 3.12system 0:04.38elapsed 171%CPU (0avgtext+0avgdata
2840696maxresident)k
0inputs+4552outputs (0major+660254minor)pagefaults 0swaps
```

Milestone 2: CPU Convolution Implementation

OP and Exec Time for different input data size:

- Running /usr/bin/time python m2.1.py 100

```
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.034094
Op Time: 0.075474
Correctness: 0.84 Model: ece408
2.87user 2.76system 0:01.00elapsed 562%CPU (0avgtext+0avgdata
203620maxresident)k
0inputs+8outputs (0major+61034minor)pagefaults 0swaps
```

* Running /usr/bin/time python m2.1.py 1000

```
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.245769
Op Time: 0.749210
Correctness: 0.852 Model: ece408
4.29user 3.00system 0:02.00elapsed 363%CPU (0avgtext+0avgdata
331980maxresident)k
0inputs+2824outputs (0major+110686minor)pagefaults 0swaps
```

* Running /usr/bin/time python m2.1.py 10000

```
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 2.446601
Op Time: 7.594124
Correctness: 0.8397 Model: ece408
15.54user 4.46system 0:11.65elapsed 171%CPU (0avgtext+0avgdata
1617164maxresident)k
0input
s+2824outputs (0major+617305minor)pagefaults 0swaps
```

Milestone 3: GPU Forward Convolution

Nvprof result:

```
==278== NVPROF is profiling process 278, command: python m3.1.py
Loading model... done
New Inference
```

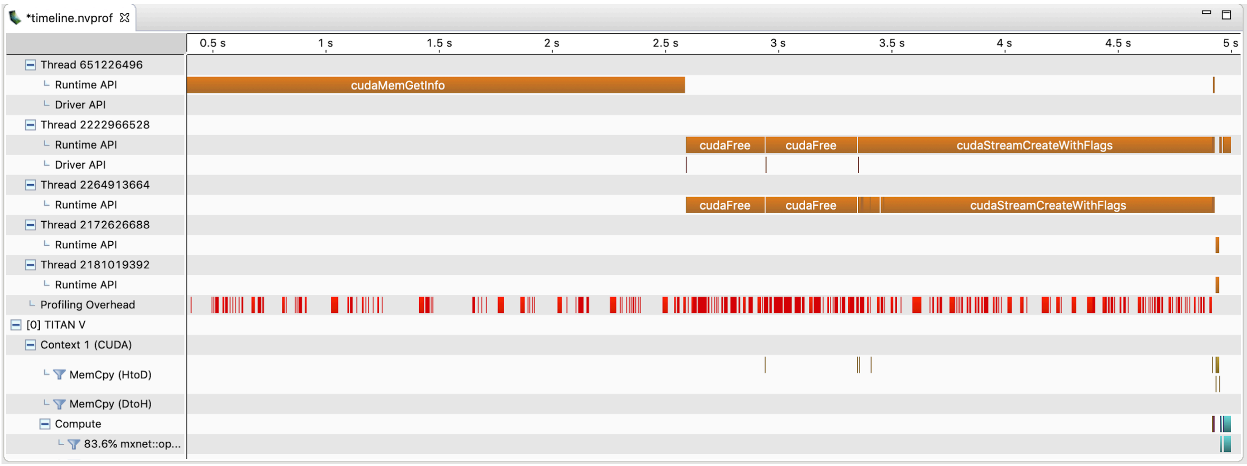
```
Op Time: 0.005826
Op Time: 0.031620
Correctness: 0.8397 Model: ece408
==278== Profiling application: python m3.1.py
==278== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.09%  37.378ms         2  18.689ms  5.7804ms  31.598ms
mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int,
int)
                   27.21%  16.927ms        20  846.34us  1.1200us  16.409ms  [CUDA memcpy
HtoD]
                    3.96%  2.4646ms         2  1.2323ms  20.864us  2.4438ms
volta_sgemm_32x128_tn
                    3.93%  2.4415ms         2  1.2207ms  737.50us  1.7040ms  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>,
mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul,
mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float,
int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                    2.62%  1.6307ms         2  815.34us  22.368us  1.6083ms  void
op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7,
cudnnNanPropagation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*,
cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float,
float, dimArray, reducedDivisorArray)
                    1.70%  1.0576ms         1  1.0576ms  1.0576ms  1.0576ms  void
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>(cudnnTensorStruct, float const *,
cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float,
cudnnNanPropagation_t=0>, int=0, bool=0>, cudnnTensorStruct*, cudnnPoolingStruct, float,
cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                    0.25%  152.89us         1  152.89us  152.89us  152.89us  void
mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,
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, int)
                    0.12%  72.416us         1  72.416us  72.416us  72.416us  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)
                    0.04%  27.614us        13  2.1240us  1.1520us  6.5280us  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)
                    0.04%  23.711us         2  11.855us  2.4000us  21.311us  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)
                    0.02%  11.712us        10  1.1710us     992ns  1.5680us  [CUDA memset]
                    0.01%  7.6160us         1  7.6160us  7.6160us  7.6160us  [CUDA memcpy
DtoH]
                    0.01%  4.9920us         1  4.9920us  4.9920us  4.9920us  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>,
float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2)
      API calls:   41.86%  2.98971s        22  135.90ms  13.172us  1.56053s
cudaStreamCreateWithFlags
                   33.80%  2.41345s        22  109.70ms  95.187us  2.40157s  cudaMemGetInfo
                   21.03%  1.50207s        18  83.449ms
841ns  399.12ms  cudaFree
                    1.13%  80.808ms       912  88.604us     305ns  23.436ms
cudaFuncSetAttribute
                    0.62%  43.958ms       216  203.51us     855ns  25.633ms
cudaEventCreateWithFlags
                    0.56%  39.842ms         6  6.6404ms  2.2730us  31.600ms
cudaDeviceSynchronize
                    0.48%  34.204ms         9  3.8005ms  27.692us  16.468ms
cudaMemcpy2DAsync
                    0.14%  9.9121ms        66  150.18us  5.5080us  2.0783ms  cudaMalloc
                    0.11%  8.1087ms         4  2.0272ms  499.17us  2.6772ms
cudaGetDeviceProperties
                    0.09%  6.1239ms        12  510.33us  6.6460us  5.1190ms  cudaMemcpy
                    0.07%  4.7508ms        29  163.82us  2.1090us  2.2024ms
cudaStreamSynchronize
                    0.04%  2.6436ms       375  7.0490us     272ns  334.18us
cuDeviceGetAttribute
                    0.01%  909.36us         4  227.34us  45.866us  691.55us
cuDeviceGetName
                    0.01%  887.76us         8  110.97us  13.114us  688.13us
cudaStreamCreateWithPriority
                    0.01%  726.15us         2  363.08us  52.207us  673.94us  cudaHostAlloc
                    0.01%  713.23us        10  71.323us  7.5440us  480.63us
cudaMemsetAsync
                    0.01%  646.22us         4  161.55us  92.941us  275.56us
cuDeviceTotalMem
                    0.01%  617.82us         4  154.46us  74.072us  259.14us
cudaStreamCreate
                    0.01%  540.37us        27  20.013us  8.2670us  52.173us
cudaLaunchKernel
                    0.00%  292.81us       202  1.4490us     545ns  4.5470us
cudaDeviceGetAttribute
                    0.00%  150.39us        29  5.1850us  1.0190us  16.214us  cudaSetDevice
                    0.00%  133.47us         6  22.244us  1.1630us  86.376us
cudaEventCreate
                    0.00%  113.29us       557     203ns      75ns     812ns
cudaGetLastError
                    0.00%  44.747us        18  2.4850us     581ns  4.4910us  cudaGetDevice
                    0.00%  26.923us         2  13.461us  5.0500us  21.873us
cudaHostGetDevicePointer
                    0.00%  16.492us         4  4.1230us  1.7760us  7.1760us
cudaEventRecord
                    0.00%  7.4790us         2  3.7390us  2.7320us  4.7470us  cudaEventQuery
                    0.00%  6.1870us        20     309ns     140ns     599ns
cudaPeekAtLastError
                    0.00%  5.9340us         2  2.9670us  1.7720us  4.1620us
cudaDeviceGetStreamPriorityRange
                    0.00%  5.4820us         6     913ns     441ns  1.8620us
cuDeviceGetCount
                    0.00%  5.2120us         5  1.0420us     505ns  1.5460us  cuDeviceGet
                    0.00%  4.4920us         3  1.4970us     793ns  2.6370us  cuInit
```

```
                    0.00%  3.6930us         1  3.6930us  3.6930us  3.6930us
cuDeviceGetPCIBusId
                    0.00%  2.7100us         4     677ns     328ns  1.3210us
cuDeviceGetUuid
                    0.00%  2.0830us         3     694ns     338ns  1.3080us
cuDriverGetVersion
                    0.00%  1.9460us         4     486ns     239ns     806ns
cudaGetDeviceCount
```

Milestone 4: GPU Forward Convolution Optimizations and Analysis

NVVP Timeline:



Optimization 1: Weight matrix (kernel values) in constant memory

Description: We observed that elements in kernel k are accessed by different threads in grid multiple times. By loading the kernel into the constant memory, we could decrease the number of global reads.

Code Snippet:

```
__constant__ float k_const[4096];
…
cudaMemcpyToSymbol(k_const,w.dptr_, sizeof(float)*C*K*K*M, 0);
// access k from k_const in kernel function
```

Analysis:

Before this optimization, our kernel performance was dominantly limited by memory bandwidth (as shown in Figure 1.1). Before optimization, the op time data was (0.006040,0.031286). After optimization, we reduce it to (0.010241, 0.024736).

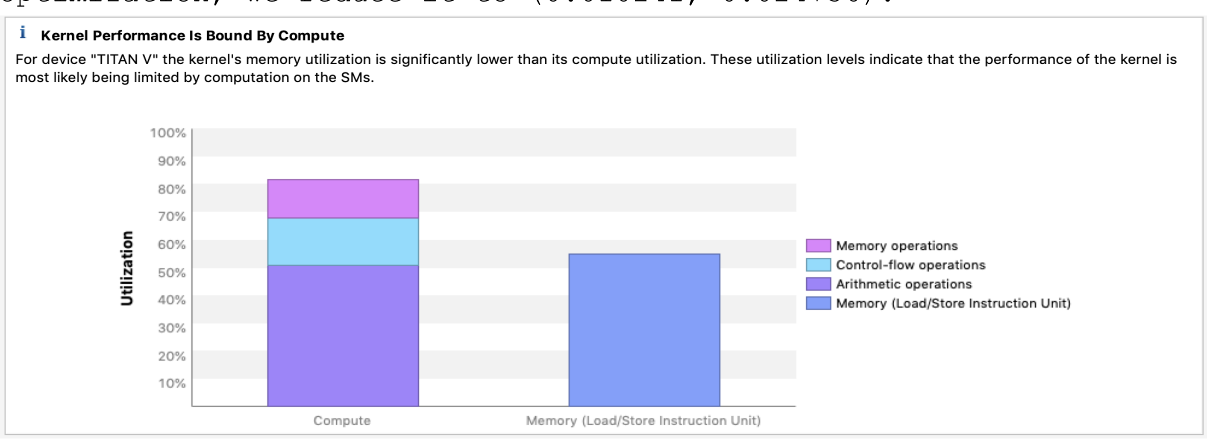


Figure 1.1 No optimization

Optimization 2: Shared Memory convolution

Description: In this optimization, we loaded the input x into shared memory. Asymptotically, this optimization should decrease global memory read by a factor of TILE_WIDTH. We tried two version of shared memory convolution.

Version 1 Code Snippet:

```
for (int c = 0; c < C; ++c){
       int s1 = threadIdx.y;
       while( s1 < TWK){
           int s2 = threadIdx.x;
           while( s2 < TWK){
               x_shared3d(c,s1,s2) = x4d(b,c, h + s1 - ty, w + s2 -
tx);
               s2 += TILE_WIDTH;
           }
           s1 += TILE_WIDTH;
       }
   }
```

Analysis:

We use threads to put the input elements needed in to the shared memory of current block. We observed that in the same block, threads with same Idx x tends to repeatedly load some of elements that overlaps and is consecutive. Using shared memory may save some time by reducing global read and may utilize some of coalescing.

Version 2 Snippet:

```
    for(int c=0; c<C; ++c){
        int tid = ty*TILE_WIDTH+tx;
        while(tid<TWK*TWK){
            // if(h+(tid/TWK)-ty<H && w+tid%TWK-tx<W)
                x_shared[c*TWK*TWK+tid] =       x4d(b,c,h+(tid/TWK)-
ty,w+(tid%TWK)-tx);
            tid+=TILE_WIDTH*TILE_WIDTH;
        }
    }
```

Analysis: Version 1 involves lots of control divergence due to the way we map threads onto the 2d array we want to load. Version 2 aims at reducing control divergence and taking advantage of memory coalescing.

From nvvp profile, we could see control divergence is drastically decreased after this modification. Following are some graph from NVVP.

Comparing 2 Versions:

Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. More...

Line / File	new-forward.cuh - /mxnet/src/operator/custom
53	Divergence = 25% [8640000 divergent executions out of 34560000 total executions]
55	Divergence = 100% [34560000 divergent executions out of 34560000 total executions]
55	Divergence = 100% [34560000 divergent executions out of 34560000 total executions]
78	Divergence = 38.9% [1120000 divergent executions out of 2880000 total executions]

Figure 2.1: Optimization 1 and Version 1 of Optimization 2

Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. More...

Line / File	new-forward.cuh - /mxnet/src/operator/custom
69	Divergence = 50% [8640000 divergent executions out of 17280000 total executions]
78	Divergence = 38.9% [1120000 divergent executions out of 2880000 total executions]

Figure 2.2: Optimization 1 and Version 2 of Optimization 2

Comment: Loading k and x from global memory to shared memory involves boundary checks, which will lead to control divergence. As shown in figure 2 and 3, adding version 1 of optimization 2 leads to a high control divergence rate.Comparing figure 3 and 4, Version 2 effectively decreases control divergence rate.

Optimization 3: Sweeping various parameters to find best values for block size

Description:

Based on the previous two optimizations, we tried different block sizes to get even better performance. With optimization 1 and 2 implemented, we noticed that our kernel performance was limited by computation on the SMs, which means when memory are read, SMs do not have enough computational resources. Thus we decrease number of threads in each block to resolve this problem. As shown in Figure 3.1, when using block dimension 16*16, the performance of the kernel suffer from insufficient computational resources on SMs. For block dimension of 8*8, it is no longer the case.

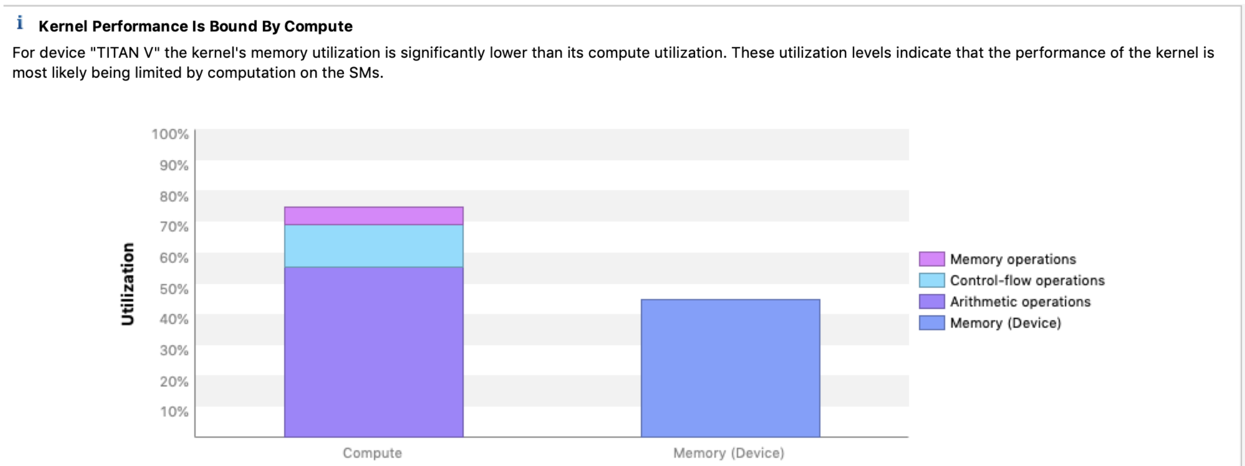


Figure 3.1 Block dimension 16*16

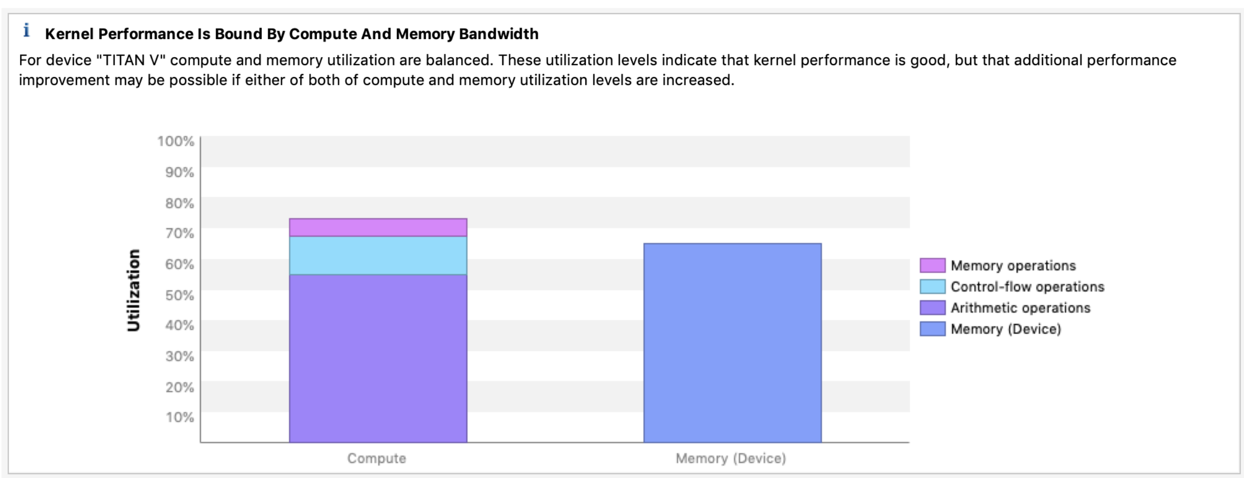


Figure 3.2 Block dimension 8*8

We could also see the improvement in performance simply from the decrease in the bolded operation time.

Output of Size of 32*32 in 10000 data size:

New Inference
Op Time: 0.014388
Op Time: **0.034988**
Correctness: 0.8397 Model: ece408

4.40user 3.34system 0:04.44elapsed 174%CPU (0avgtext+0avg
data 2856860maxresident)k
0inputs+4640outputs (0major+666082minor)pagefaults 0swaps

Output of Size of 16*16 in 10000 data size:
New Inference
Op Time: 0.006355
Op Time: **0.031710**
Correctness: 0.8397 Model: ece408
4.43user 3.32system 0:04.43elapsed 175%CPU (0avgtext+0avgdata
2844352maxresident)k
0inputs+4640outputs (0major+662779minor)pagefaults 0swaps

Output of Size of 8*8 in 10000 data size:
New Inference
Op Time: 0.010416
Op Time: **0.026588**
Correctness: 0.8397 Model: ece408
4.54user 2.97system 0:04.59elapsed 163%CPU (0avgtext+0avgdata
2858084maxresident)k
0inputs+4640outputs (0major+666778minor)pagefault
s 0swaps

Analysis:
Potential
Running in different block size, we found for current optimization, size of 8*8 is optimal. In most obvious way, it uses least time in completing second layer, so it is the best, but this may change subject to different opt implemented in final run. From figure 3.1 and 3.2 we can see the reason: Both has high computational power utilization but 16*16 has a lower memory usage, the performance is most limited by compute. Thus in 8*8, by decreasing the size of block, we increase computation and memory usage ratio so none of them remain idle while another is doing work.