ECE 408 REPORT
Project Milestone 1

team_foveon
Zuodong Liang / zuodong2

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

Name	Time(%)
[CUDA memcpy HtoD]	38.97
void cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int, float*, cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, kernel_conv_params, int, float, float, int, float, float, int, int)	20.81
volta_cgemm_64x32_tn	12.2
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)	7.21
volta_sgemm_128x128_tn	5.76
void fft2d_c2r_32x32<float, bool=0, bool=0, unsigned int=1, bool=0, bool=0>(float*, float2 const *, int, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*, int2, int, int)	5.74
Total Time %	**90.69**

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

Name	Time(%)
cudaStreamCreateWithFlags	41.7
cudaMemGetInfo	34.01
cudaFree	21.27
Total Time %	**96.98**

III. Explanation of the difference between kernels and API calls

API, as a programming interface, allow the host to interface with the GPU. APIs are executed on the host. The kernels, on the other hand, are the code that actually running on the GPU.
CUDA APIs are defined by NVIDIA, kernels are codes that written by programmers.

IV. Output of rai running MXNet on the CPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8236}
8.81user 3.67system 0:05.11elapsed 244%CPU (0avgtext+0avgdata
2469544maxresident)k
0inputs+2824outputs (0major+669134minor)pagefaults 0swaps
```

Run time: 5.11 s

V. Output of rai running MXNet on the GPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8236}
4.35user 3.41system 0:04.30elapsed 180%CPU (0avgtext+0avgdata
2836344maxresident)k
0inputs+4552outputs (0major+661560minor)pagefaults 0swaps
```

Run time: 4.30 s

Project Milestone 2

```
* Running /usr/bin/time python m2.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 2.446374
Op Time: 7.487101
Correctness: 0.8397 Model: ece408
15.36user 4.47system 0:11.54elapsed 171%CPU (0avgtext+0avgdata
1619204maxresident)k
0inputs+2824outputs (0major+618017minor)pagefaults 0
swaps
```

```
* Running /usr/bin/time python m2.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.247094
Op Time: 0.747553
Correctness: 0.852 Model: ece408
4.44user 2.77system 0:01.98elapsed 362%CPU (0avgtext+0avgdata
330676maxresident)k
0inputs+2824outputs (0major+110770minor)pagefaults 0swaps
```

```
* Running /usr/bin/time python m2.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.032041
Op Time: 0.073482
Correctness: 0.84 Model: ece408
3.10user 2.87system 0:01.05elapsed 564%CPU (0avgtext+0avgdata
203712maxresident)k
0inputs+2824outputs (0major+62399minor)pagefaults 0swaps
```

I. Whole program execution time

When number of images is 10,000, from the execution result, we can see that the whole program execution time is 11.54 s.
When number of images is 1,000, the whole program execution time is 1.98 s.
When number of images is 100, the whole program execution time is 1.05 s.

II. Op Times

When number of images is 10,000, the Op Time for layer 1 is 2.446374 s, the Op Time for layer 2 is 7.487101 s.
When number of images is 1,000, the Op Time for layer 1 is 0.247094 s, the Op Time for layer 2 is 0.747553 s.
When number of images is 100, the Op Time for layer 1 is 0.032041s, the Op Time for layer 2 is 0.073482s.

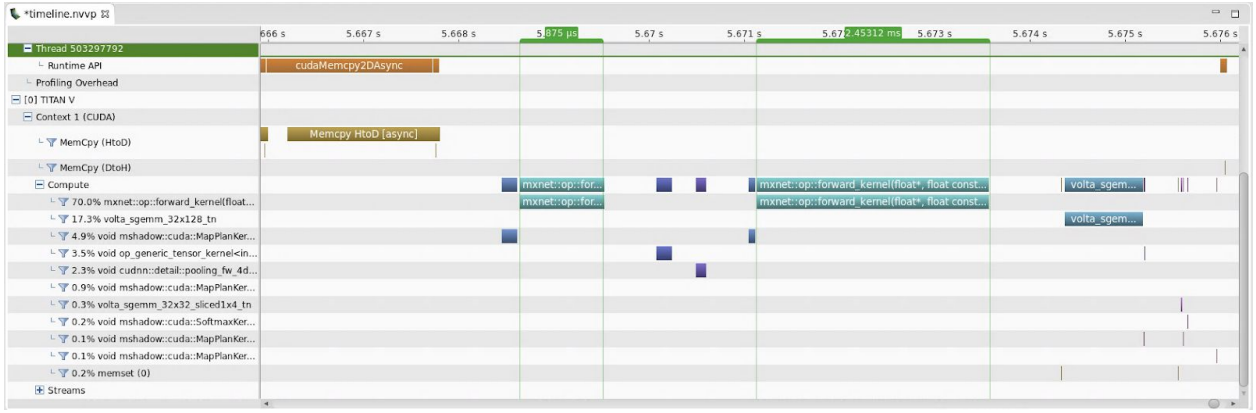
Project Milestone 3

I. Whole Program Execution Time

```
✱ Running /usr/bin/time python m3.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.009143
Op Time: 0.024142
Correctness: 0.8397 Model: ece408
4.29user 3.20system 0:04.25elapsed 176%CPU (0avgtext+0avg
data 2834916maxresident)k
0inputs+4576outputs (0major+660785minor)pagefaults 0swaps
```

When we use CPU forward pass, the total time was 11.54s. When we use the GPU forward pass, the total time is 4.25s. The total op time for the CPU pass was 9.933475s, while the total op time for the unoptimized GPU pass is 33.285ms.

II. NVPROF Timeline



When the number of images is 1000, from the NVPROF result, we can notice that the first kernel run took 875us, the second kernel run took 2.45312ms. The mxnet kernel took 70% of the execution time.

Project Milestone 4

I. Optimizations Performed

By the end of milestone 4, I have tried three different optimizations. The first optimization is put input image in shared memory. The second optimization is put input image in shared memory, and put kernel values into constant memory. The third optimization is use different kernel for different layer.

The result of each optimization is in the following table:

Optimization	**Layer 1 (ms)**	**Layer 2 (ms)**	**Total (ms)**
No Optimization	9.143	24.142	33.285
Shared memory convolution	6.533	39.588	46.121
Shared memory convolution with kernel in constant memory	6.239	36.158	42.397
Shared memory + constant memory, different kernel for different layer, adjusted TILE_WIDTH	4.463	10.436	14.899

From the table above I noticed that the result of shared memory convolution and constant memory kernel did not improve my original implementation as I expected. Look back at my code I noticed that I did the multiplication on one channel right after I put one channel of input image into the shared memory. So we can notice that for layer 1, which only have one channel, the performance improved. However, for the second layer, the performance significantly decreased.

These result tell me that the shared memory convolution and constant memory kernel do have a positive impact on my performance, but the actual implementation will also impact my performance.

Thus, in the next optimization I wrote two convolution kernel, one for each layer. For the second layer kernel, I also changed the order to load input of *all layers* to the shared memory before perform any multiplication. This optimization give me a significantly better result: the total op time is within 20ms.

I further tuned the TILE_WIDTH of each kernel. After several test I eventually find the optimal TILE_WIDTH for each layer, and my op time decreased to 14.899ms.

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

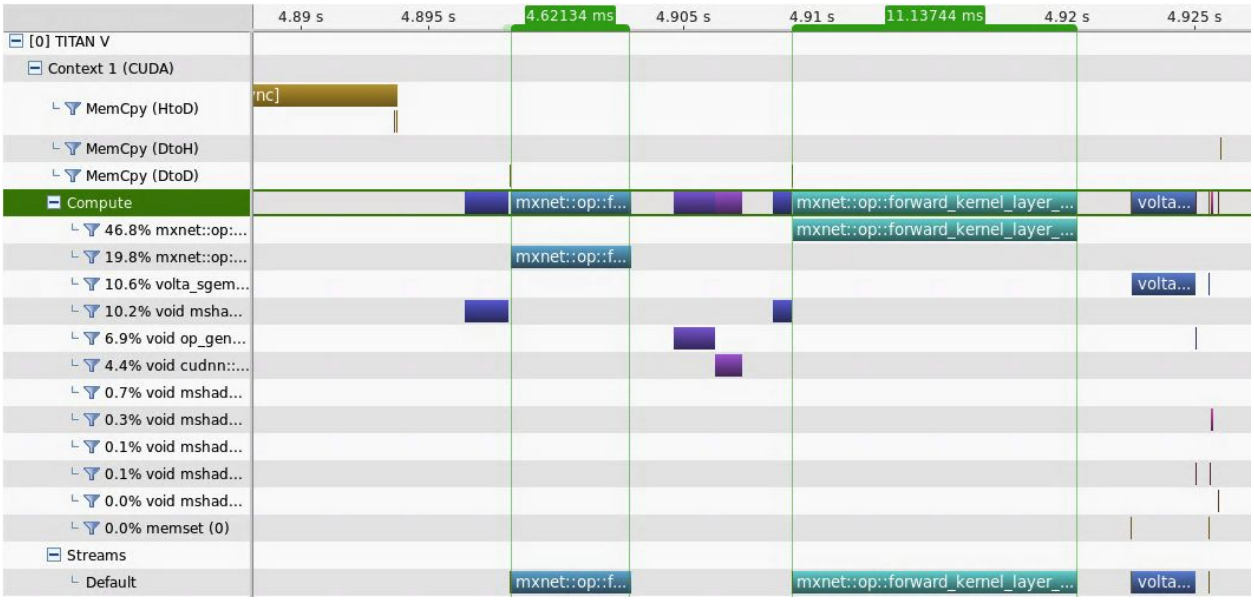
These results are obtained by running nvprof on the last optimization.

Name	Time(%)
[CUDA memcpy HtoD]	42.75
mxnet::op::forward_kernel_layer_2(float*, float const *, int)	27.12
mxnet::op::forward_kernel_layer_1(float*, float const *, int)	11.44
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)	5.82
volta_sgemm_32x128_tn	5.63
Total Time %	**91.76**

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

Name	Time(%)
cudaStreamCreateWithFlags	42.36
cudaMemGetInfo	34.40
cudaFree	20.81
Total Time %	**97.56**

IV. NVPROF Timeline



The visual profiler also suggests that the forward_kernel_layer_2 has the highest optimization priority. So I will focus on optimize kernel 2 before our final submission.

Final Milestone

I. Optimizations Performed

By the end of milestone 5, I have tried several different optimizations. The first optimization is put input image in shared memory. The second optimization is put input image in shared memory, and put kernel values into constant memory. The third optimization is use different kernel for different layer.

The result of each optimization is in the following table:

Optimization	**Layer 1 (ms)**	**Layer 2 (ms)**	**Total (ms)**
No Optimization	9.143	24.142	33.285
Shared memory convolution	6.533	39.588	46.121
Shared memory convolution with kernel in constant memory	6.239	36.158	42.397
Shared memory + constant memory, different kernel for different layer, adjusted TILE_WIDTH	4.463	10.436	14.899
Unroll + shared-memory matrix multiply	125.996	259.194	385.19
Tuning unroll + shared-memory matrix multiply with restrict and loop unrolling (best time)	99.169	153.775	252.944
Sweeping various parameters to find best values	See table below		

From the table above, I noticed that the performance of the popular unroll + shared-memory matrix multiply did not perform very well. This might because of my own implementation, since I loop around the batch to make my code easier. There are several optimizations came into my mind: unroll the for loop, since there are several of them in the gemm kernel; try kernel fusion, which will reduce the memcpy time between two kernel. I first tried to tune the kernel with __restrict__ and #pragma unroll macros. These macros actually significantly increase the running speed of my gemm kernel, it helps me to cut almost half the time. After that I also tried to put the unroll kernel and matrix multiply kernel together, perform a kernel fusion and see if I can get further optimization from there. However, after several hours of debugging I decided to put it on the side and try other optimizations.

I also tried to use __restrict__ and unroll for my best-performance kernel. However, after several different unroll and different combination, I couldn't make my kernel perform any better.

I also tried to sweep different parameters to see if I can make my kernel run faster. Specifically, I want see if the block size can make a big difference on my run time. I think in my implementation I

don't need to worry too much about memory coarsening issue. So I test my kernel with several different TILE_WIDTH combination. Here's the result:

TILE_WIDTH_1	TILE_WIDTH_2	Layer 1 (ms)	Layer 2 (ms)	Total (ms)
16	16	4.625	21.508	26.133
22	18	4.516	11.757	16.273
22	9	4.463	10.436	14.899
22	6	4.527	16.748	21.275
11	9	6.059	10.83	16.889

I first begin with the standard tile width of 16. The result wasn't very good. After that I went through my code and noticed that since the block size is determined by ceil(_H1_OUT / (float)(TILE_WIDTH_1)) and ceil(_W1_OUT / (float)(TILE_WIDTH_1)), integer divisor as the tile width might provide me a better result. So I tried to fix the TILE_WIDTH_1 to 22 and find the optimal TILE_WIDTH_2. From the table above I noticed that TILE_WIDTH_2 = 9 works the best for my kernel. I also tried several different TILE_WIDTH_1 value, and it turns out 22 works the best.

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

These results are obtained by running nvprof on my gemm kernel. I tried to find some clue from the nvprof result.

Name	Time(%)
mxnet::op::matrixMultiplyShared	62.73
mxnet::op::gemm_unroll	25.01
[CUDA memcpy HtoD]	8.65
Total Time %	**96.39**

It is pretty clear that the long optime of my gemm kernel is due to the inefficient shared matrix multiplication. Maybe register-tiled or other advanced matrix multiplication algorithm can make my gemm kernel run faster.

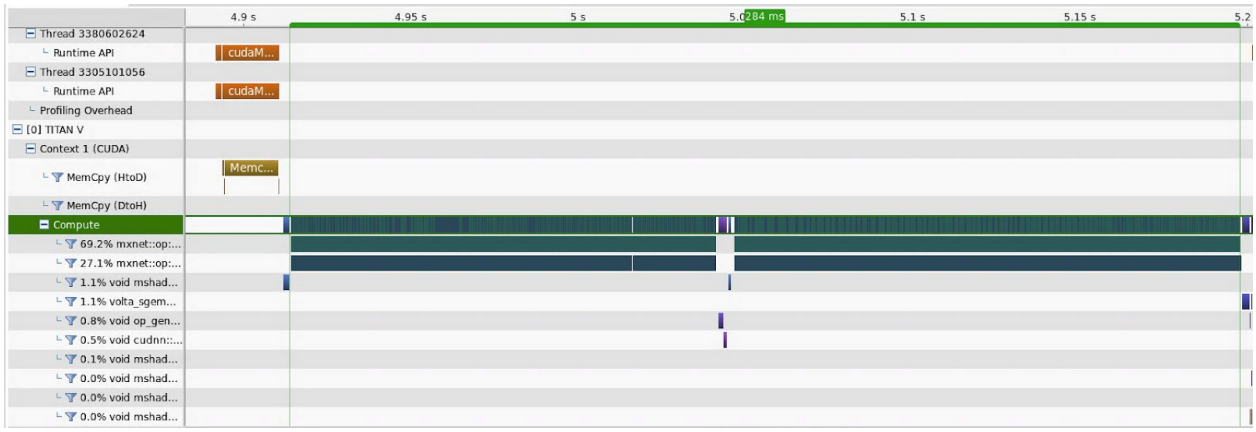
III. CUDA API calls that collectively consume more than 90% of the program time

Name	Time(%)

cudaStreamCreateWithFlags	41.21
cudaMemGetInfo	33.3
cudaFree	19.79
Total Time %	**94.3**

Compare with my best performance separate kernel shared-memory implementation, the overall API calls are pretty similar. I think this can tell us that after unroll and restrict, the overall cuda api usage for these two different kernel are roughly the same.

IV. NVVP Timeline



The kernel are divided into many small segments, which indicates that my kernel are running through a for loop that go over the mini-batches. I think I can address this issue with task parallelism in the future.
The visual profiler also suggests that the matrix multiplication has the highest optimization priority, which agree with my earlier observation from nvprof result.

V. Teamwork & Course Suggestion

Since my partner dropped this course right after the project start, I end up doing this project solo. It is nice to be able to think about all the possible optimizations and implement them, but sometimes it is quite desperate to debug along.
All in all, I really enjoy this course and I'm actually applying what I learned in this course in my research project. So I really appreciate this course!