ECE408 Final Project Report (up to Milestone 4)

Haoxiang Li(hl11)
Haichuan Xu(hxu52)
Chicago Scholar/UIUC
Team: Wardrick

MileStone 1:

1. List kernel calls that collectively consumes 90% of program run time

- void fermiPlusCgemmLDS128_batched<bool=0, bool=1, bool=0, bool=0, int=4, int=4, int=4, int=3, int=3, bool=1, bool=1>(float2**, float2**, float2**, float2*, float2 const *, float2 const *, int, int, int, int, int, int, __int64, __int64, __int64, float2 const *, float2 const *, float2, float2, int)

Usage: 34.08%

- void cudnn::detail::implicit_convolve_sgemm<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, cudnn::detail::implicit_convolve_sgemm<float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, float const *, kernel_conv_params, int, float, float, int, float const *, float const *, int, int)

Usage: 27.02%

- void fft2d_c2r_32x32<float, bool=0, unsigned int=0, bool=0, bool=0>(float*, float2 const *, int, int, int, int, int, int, int, int, int, float, float, cudnn::reduced_divisor, bool, float*, float*)

Usage: 12.67%

- sgemm_sm35_ldg_tn_128x8x256x16x32

Usage: 8.2%

- [CUDA memcpy HtoD]

Usage: 6.42%

- void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *, cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>, cudnnTensorStruct*, float, cudnnTensorStruct*, int, cudnnTensorStruct*)

Usage: 4.07%

2. List CUDA API calls that collectively consumes 90% of program time.

- cudaStreamCreateWithFlags

Usage: 43.91%

- cudaFree

Usage: 27.04%

- cudaMemGetInfo

Usage: 20.45%

3. Explain the difference between kernels and API calls.

Kernel calls are actual functions that do the computational works in the GPU, while API calls are calls from the host to setup the interface between host and device.

4. Show output of rai running MXNet on the CPU and list program run time.

✱ Running /usr/bin/time python m1.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
EvalMetric: {'accuracy': 0.8444}
12.85user 6.12system 0:08.40elapsed 225%CPU (0avgtext+0avgdata 2828652maxresident)k
0inputs+2624outputs (0major+39228minor)pagefaults 0swaps

Program run time: 8.40s.

5. Show output of rai running MXNet on the GPU and list program run time.

✱ Running /usr/bin/time python m1.2.py
Loading fashion-mnist data...
done
Loading model...
[22:46:19] 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}
2.18user 1.10system 0:02.81elapsed 116%CPU (0avgtext+0avgdata 1140180m axresident)k
0inputs+3136outputs (0major+157211minor)pagefaults 0swaps

Program run time: 2.81s.

MileStone 2:

1. **List full program runtime of the CPU implementation of CNN forward propagation layer.**

30.60user 1.44system 0:30.03elapsed 106%CPU (0avgtext+0avgdata 2819932maxresident)k
0inputs+2624outputs (0major+38686minor)pagefaults 0swaps

Program runtime is: 30.03s.

2. List Op runtime.

✱ Running /usr/bin/time python m2.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 6.567796
Op Time: 19.518647
Correctness: 0.8451 Model: ece408

Op 1 runtime is: 6.567796s.
Op 2 runtime is: 19.518647s.

MileStone 3:

1. **Demonstrate nvprof profiling the execution.**

✱ Running nvprof /usr/bin/time python m3.1.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.340871
Op Time: 0.502004
Correctness: 0.8451 Model: ece408
2.66user 1.38system 0:03.61elapsed 112%CPU (0avgtext+0avgdata 1145256maxresident)

MileStone 4:

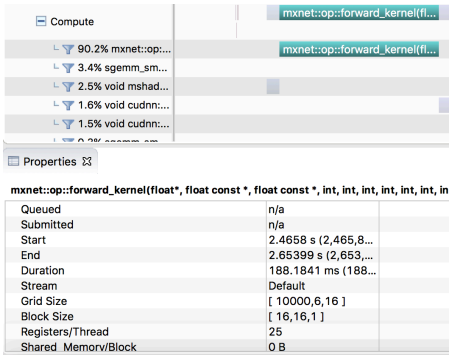
Optimization 1:

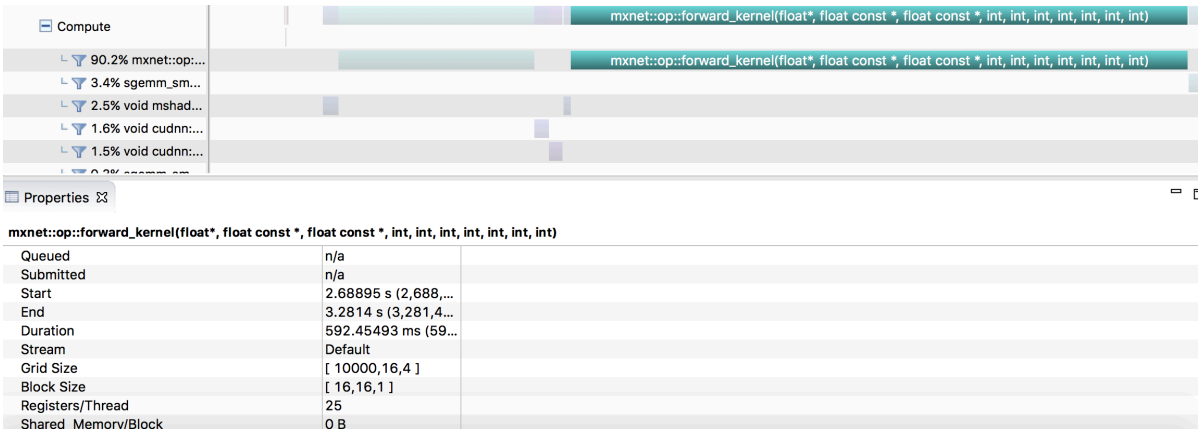
Placed the convolution mask(kernel) into constant memory. Since two different input mask sizes exist, we implemented two constant memory array and uses the corresponding one in kernel call with respect to the layer size.

Profiling with nvprof result:

==314== NVPROF is profiling process 314, command: python m4.1.py
done
New Inference
Op Time: 0.188060
Op Time: 0.591159
Correctness: 0.8451 Model: ece408

NVVP result for two kernel calls:





Performance:

Reduced Op1 runtime from 341ms to 188ms.
Increases Op2 runtime from 502ms to 581ms

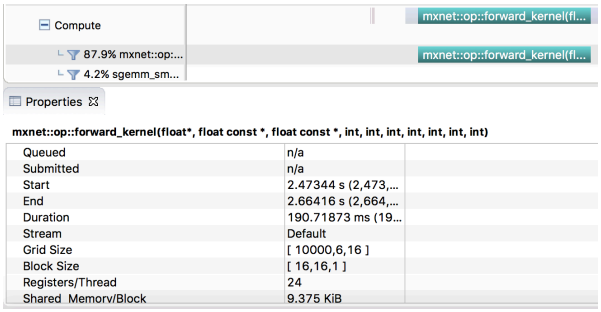
Optimization 2:

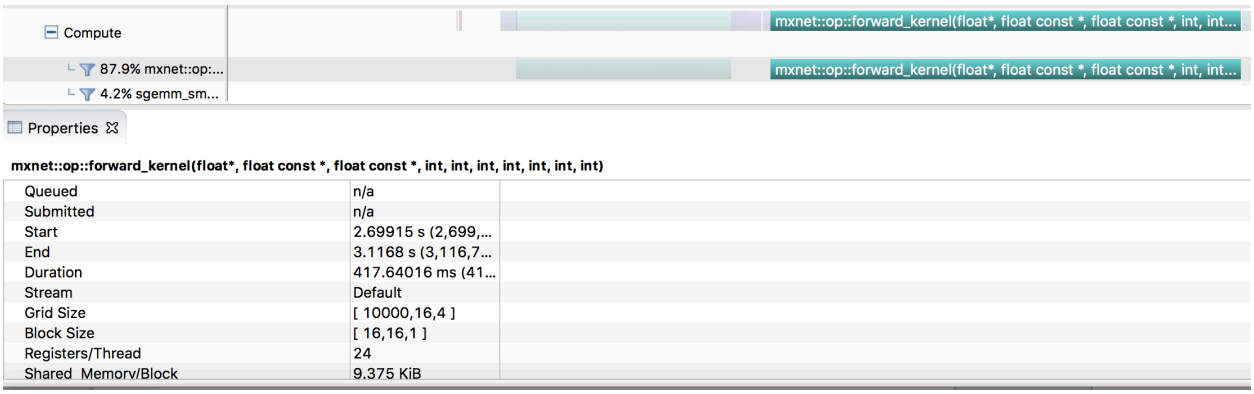
Shared memory convolution for the larger layer kernel.
We loaded the input feature data needed for the output pixels in one block to process into the shared memory and the computation retrieves data from the shared memory.
At first we implemented this method for both small and large layer operation, but realized that doing so for the smaller kernel increases the runtime, so we only preserved the implementation for the larger layer kernel call.

Profiling with nvprof result:

==315== NVPROF is profiling process 315, command: python m4.1.py
done
New Inference
Op Time: 0.190928
Op Time: 0.417818
Correctness: 0.8451 Model: ece408

NVVP result for two kernel calls:





Performance:

Slightly increases Op1 runtime from 188ms to 191ms.
Reduced Op2 runtime from 581ms to 418ms.

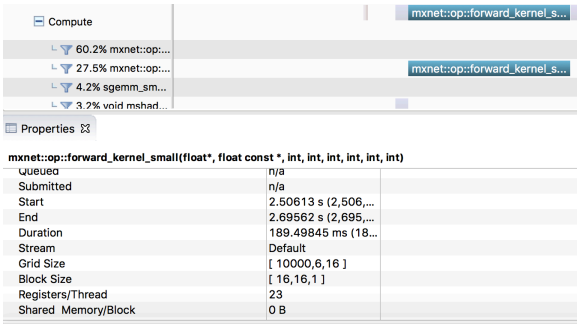
Optimization 3:

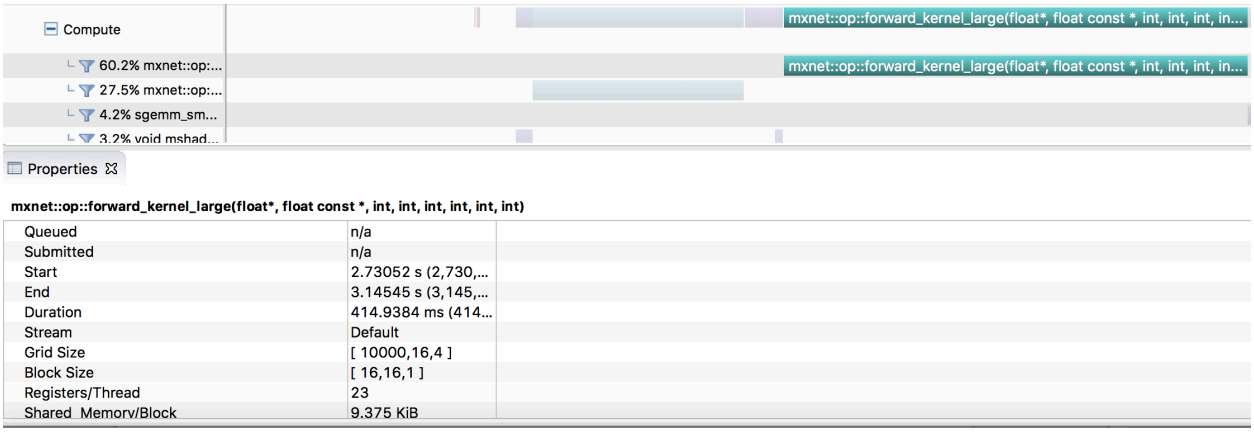
Separated condition checks and implemented two separate kernels for two different layer sizes.

Profiling with nvprof result:

==315== NVPROF is profiling process 315, command: python m4.1.py
done
New Inference
Op Time: 0.189507
Op Time: 0.414641
Correctness: 0.8451 Model: ece408

NVVP result for two kernel calls:





Performance:

Slightly reduced Op1 runtime from 191 to 190ms.
Slightly reduced Op2 runtime from 418ms to 415ms.

MileStone 5:

Optimization 4:

Loop unrolling and tuning in kernel code.

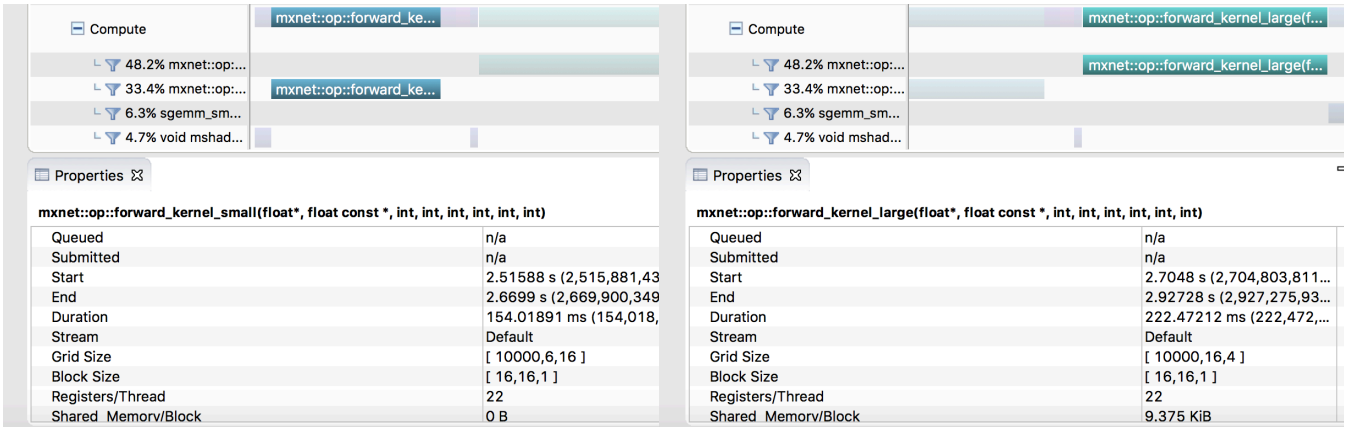
Since for every for loop, the kernel will do one condition check for every iteration before executing the real code. Thus, there are a lot of overhead in executing for loops. Since for our project, we already know the size for every for loop, unrolling the for loops should increase performance. As it turns out, the loop unrolling optimization did increase performances, having a greater speed up on the larger kernel.

Profiling with nvprof result:

✱ Running nvprof -o timeline.nvprof python m4.1.py
Loading fashion-mnist data...
done
Loading model...
==313== NVPROF is profiling process 313, command: python m4.1.py
done
New Inference
Op Time: 0.154115
Op Time: 0.222569
Correctness: 0.8451 Model: ece408
==313== Profiling application: python m4.1.py
==313== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
46.05%	224.91ms	1	224.91ms	224.91ms	224.91ms	mxnet::op::forward_kernel_large(float*, float const *, int, int, int, int, int, int)
31.95%	156.02ms	1	156.02ms	156.02ms	156.02ms	mxnet::op::forward_kernel_small(float*, float const *, int, int, int, int, int, int)

NVVP result for two kernel calls:



Performance:

Reduced Op1 runtime from 190ms to 154ms.
Reduced Op2 runtime from 415ms to 223ms.

Optimization 5:

Sweeping TILE_WIDTH value and block sizes.

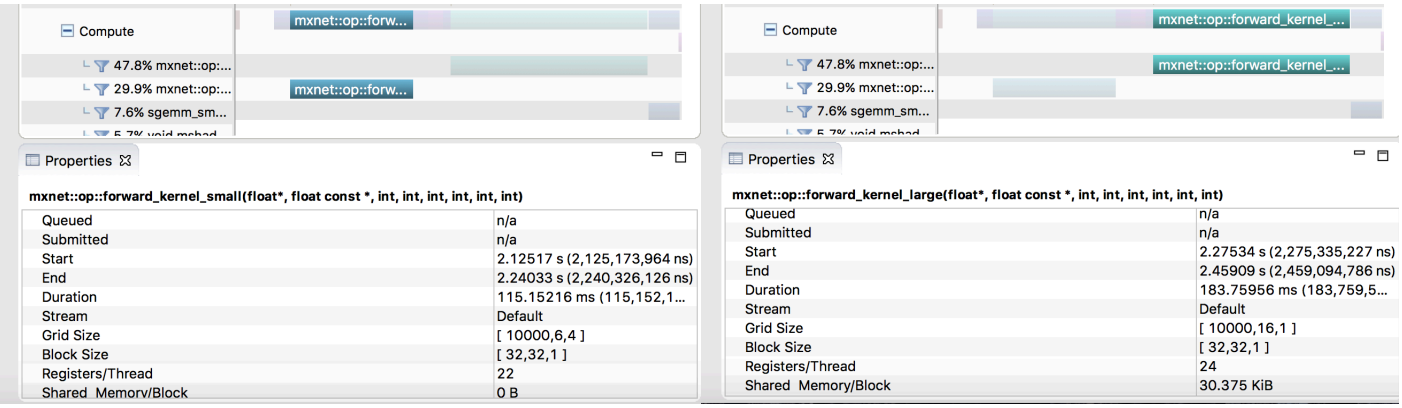
Since for different output feature map sizes, the best TILE_WIDTH definition in order to minimize control divergence among threads in the same block and to maximize shared memory reuse within block is different. Sweeping for different TILE_WIDTH values and thus sweeping the kernel launch parameters may be helpful to increase kernel performance. Indeed, with experimenting of TILE_WIDTH of size 8, 16, 30 and 32. We found that the TILE_WIDTH of 32, which maximizes the available threads in a block and minimizes control divergence for the edge block yields the best performance.

Profiling with nvprof result:

✱ Running nvprof python m4.1.py
Loading fashion-mnist data...
done
Loading model...
==316== NVPROF is profiling process 316, command: python m4.1.py
done
New Inference
Op Time: 0.115179
Op Time: 0.184124
Correctness: 0.8451 Model: ece408
==316== Profiling application: python m4.1.py
==316== Profiling result:

Time(%)	Time	Calls	Avg	Min	Max	Name
45.19%	183.99ms	1	183.99ms	183.99ms	183.99ms	mxnet::op::forward_kernel_large(float*, float const *, int, int, int, int, int, int)
28.26%	115.06ms	1	115.06ms	115.06ms	115.06ms	mxnet::op::forward_kernel_small(float*, float const *, int, int, int, int, int, int)

NVVP result for two kernel calls:



Performance:

Reduced Op1 runtime from 154ms to 115ms.
Reduced Op2 runtime from 223ms to 184ms.

Optimization 6:

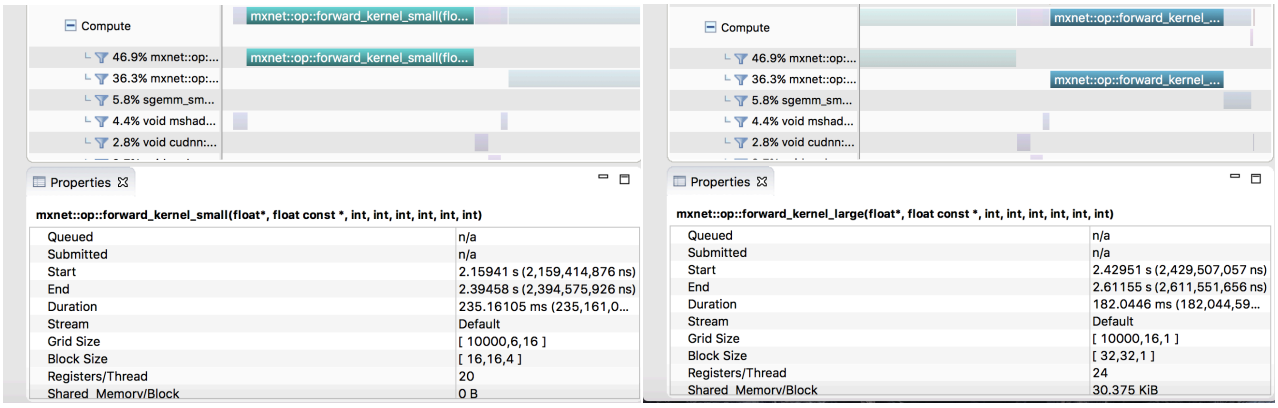
Input channel reduction: atomic add operations on small kernel.

In the design without atomic add operations for calculating the value of one output feature pixel, 25 floating point operations need to be done within one thread. In order to reduce the amount of work every thread need to be done, we thought of allocating more threads within each thread block and perform atomic add operations to split the work of one thread. As it turns out, although the amount of work for each thread decreases, the total runtime increases. I suggested that there may be three reasons behind this. First reason is that the overhead of atomicAdd() function outrages the time we saved for each thread. Second, the __syncthread() operation before adding the result to the same memory location may be waiting for too long before continuing because we do not have control over the behavior of thread execution sequences. Third reason is that the previous optimized version is utilizing 32*32 thread block, to implement atomic add operation, we had to scale down to an unoptimized 16*16*4 thread block, the overhead of the unoptimized thread block dimension may cause the program to run slower.

Profiling with nvprof result:

✱ Running nvprof python m4.1.py
Loading fashion-mnist data...
done
Loading model...
==315== NVPROF is profiling process 315, command: python m4.1.py
done
New Inference
Op Time: 0.235150
Op Time: 0.182346
Correctness: 0.8451 Model: ece408
==315== Profiling application: python m4.1.py
==315== Profiling result:
Time(%) Time Calls Avg Min Max Name
44.77% 235.04ms 1 235.04ms 235.04ms 235.04ms
mxnet::op::forward_kernel_small(float*, float const *, int, int, int, int, int, int)
34.70% 182.22ms 1 182.22ms 182.22ms 182.22ms
mxnet::op::forward_kernel_large(float*, float const *, int, int, int, int, int, int)

NVVP result for two kernel calls:



Performance:

Increased Op1 runtime from 115ms to 235ms.