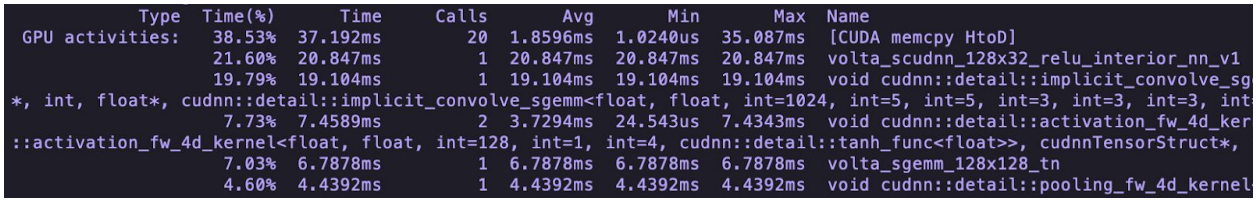
Milestone 1
Team Name: mallard
Names: Gauri Konanoor, Hamilton Silberg, Sankruth Kota
NetIDs: gmk2, hfs2, srkota2

Introduction

In milestone 1 of the final project, we run a batch forward pass on test data using the MXNet framework. We compare the performance of this operation on an AMD64 CPU as opposed to a Pascal GPU. We then checked which kernels consumed 90% of the computation time.

Kernel Calls



```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   38.53%  37.192ms        20  1.8596ms  1.0240us  35.087ms  [CUDA memcpy HtoD]
                   21.60%  20.847ms         1  20.847ms  20.847ms  20.847ms  volta_scudnn_128x32_relu_interior_nn_v1
                   19.79%  19.104ms         1  19.104ms  19.104ms  19.104ms  void cudnn::detail::implicit_convolve_sg
*, int, float*, cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int
                    7.73%  7.4589ms         2  3.7294ms  24.543us  7.4343ms  void cudnn::detail::activation_fw_4d_ker
::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>, cudnnTensorStruct*,
                    7.03%  6.7878ms         1  6.7878ms  6.7878ms  6.7878ms  volta_sgemm_128x128_tn
                    4.60%  4.4392ms         1  4.4392ms  4.4392ms  4.4392ms  void cudnn::detail::pooling_fw_4d_kernel
```

The kernel calls adding up to greater than 90% include the combination of:

[CUDA memcpy HtoD]
volta_scuddnn_128x32_relu_interior_nn_v1
cudnn::detail::implicit_convolve_sgemm
cudnn::detail::activation_fw_4d_kernel
volta_sgemm_128x128_tn

and,

[CUDA memcpy HtoD]
Volta_scuddnn_128x32_relu_interior_nn_v1
cudnn::detail::implicit_convolve_sgemm
cudnn::detail::activation_fw_4d_kernel
cudn::detail::pooling_fw_4d_kernel.

API Calls

```
      API calls:   39.88%  2.72075s        22  123.67ms  12.913us
1.41769s  cudaStreamCreateWithFlags
                   34.21%  2.33356s        24  97.232ms  65.529us  2.32859s  cudaMemGetInfo
                   21.89%  1.49314s        19  78.586ms     288ns  389.22ms  cudaFree
```

The API calls covering greater than 90 percent consumption by time is the combination of:

cudaStreamCreateWithFlags
cudaMemGetInfo
cudaFree

Kernel Calls v. API Calls

Kernels are C functions completed repeated N times in parallel by N different CUDA threads within the GPU device. API calls are specific actions called for the GPU to enact by the CPU, whether it be to allocate or transfer memory, or even to run an instance of a designated kernel with specific thread dimensions. Overall, the main difference is that the kernel deals with GPU tasks that are much more repeatable than those dealt with in API calls, with API calls being more generalized to GPU operation.

In general, it is beneficial to analyse the optimisations from parallelising computations using kernels, which is what we do in this milestone.

Test Results

First, we tan our forward pass test on a CPU. As we can see below, our runtime duration was **13.87 seconds.**

```
Successfully installed mxnet
+ Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
19.79user 4.22system 0:13.87elapsed 173%CPU (0avgtext+0avgdata 5954848maxresident)k
0inputs+2856outputs (0major+1578349minor)pagefaults 0swaps
```

Upon running the same computation on a GPU, we can see that out runtime duration is **4.68 seconds**.

```
Successfully installed mxnet
+ Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
4.11user 2.52system 0:04.68elapsed 141%CPU (0avgtext+0avgdata 2847636maxresident)k
8inputs+4568outputs (0major+707070minor)page
faults 0swaps
```

Parallelising our simple machine learning computation optimised our runtime by 66%.

Operation Time One:

Milestone 2

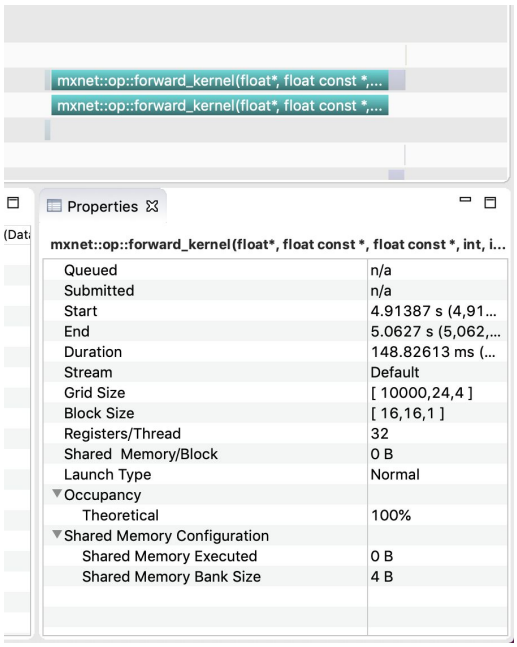
In this milestone, we implemented a sequential CPU version of a forward-propagation path convolutional layer.

```
Loading model... done
New Inference
Op Time: 24.896752
Op Time: 105.804736
Correctness: 0.8152 Model: ece408
142.33user 9.44system 2:15.91elapsed 111%CPU (0avgtext+0avgdata 5950984maxresident)k
```

The operation times as shown above is **24.896 seconds** for the first time that layer is invoked, and approximately **105.8 seconds** for the second time the layer is invoked. The total runtime of our program is about **2 minutes and 15 seconds.**

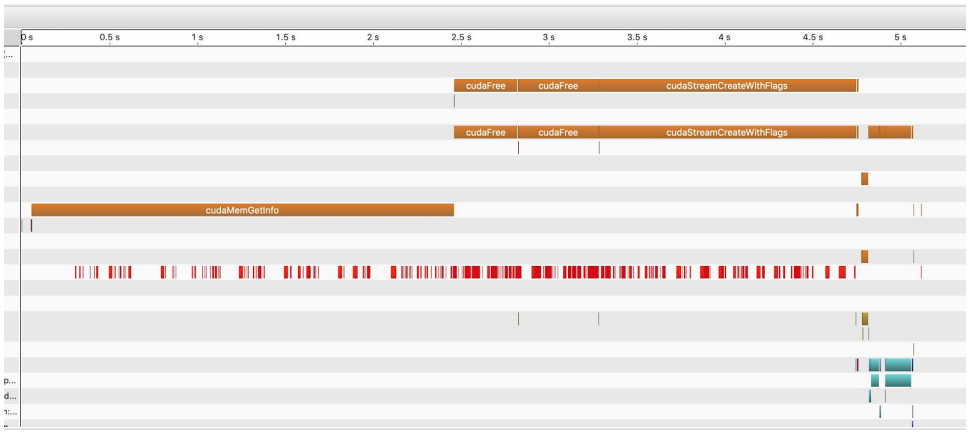
Milestone 3

In this milestone, our team worked on developing a GPU convolution. Our convolution for this milestone had a correctness of **0.85 for 100 image**s, a correctness of **0.827 for 1000 image**s, and a correctness of **0.8171 for 10000 images** which aligned with the expected output.



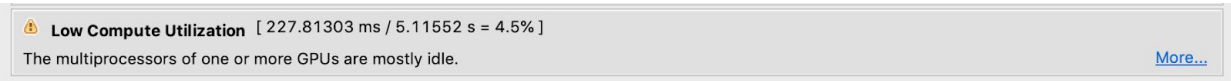
After performing some analysis using nvvp and nvprof, we can better understand our kernel. We can see that the total time it took to execute the kernel was approximately **0.14883 seconds**. We can also see that we performed **no shared memory optimizations** since our shared memory usage was 0.

The visual profiler also provided key data to how our kernel executed. After taking a more in-depth look we can see that we launched our kernel with a **grid size of [10,000, 24, 4]** and a **block size of [16, 16, 1].** With a **theoretical occupancy of 100 percent**, our team was fairly certain that our kernel utilized our hardware SM's well, and that we had the correct block and grid sizes.



Zooming out a little, the fact that we have no shared memory usage tells us that we have **poor bandwidth utilization**. The way our team determined this was by taking a look at the higher level graphical data shown above. As you can see, **cudaMemGetInfo** is the single cuda API call that takes the longest time to perform, and we're hoping to implement shared memory and some of the tiling strategies that we learned about in class.

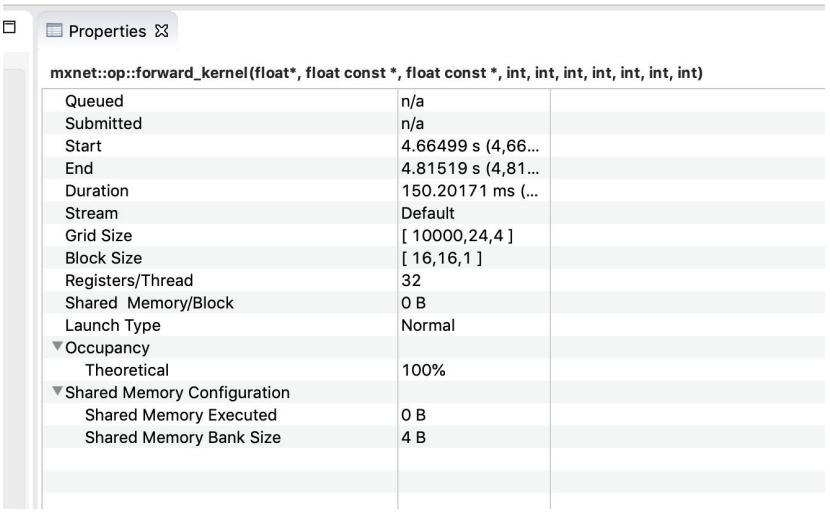
Also, it seems as though our implementation was **not as parallel** as we had thought since we did receive an indication from nvprof informing us that we could have had stronger GPU usage to speed up our kernel, as shown in the image below.



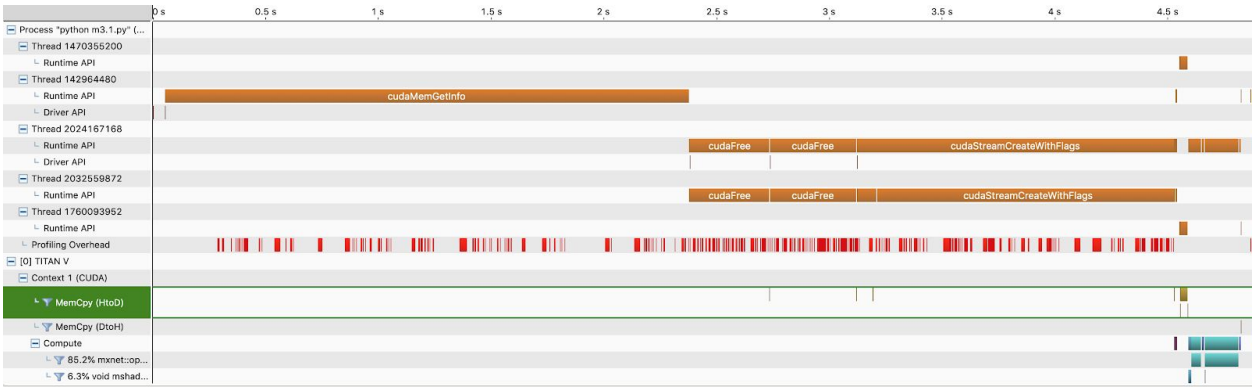
Milestone 4

Optimisation 1:

The first optimization that was done was the tuning with restrict and loop unrolling. After performing this optimization, we did not notice any significant change in our op-time. The op time after this optimization was approximately 150 milliseconds, while our kernel with 0 optimizations was 148.83 milliseconds.



Our shared memory usage is still at 0, which is accurate, and our block size and grid size did not change from our previous implementation. By parallelizing the for loop, we don't achieve much optimization as each new thread still needs to access each element of information individually from global memory. **Our overall conclusion is that loop unrolling and tuning with restrict was an unnecessary optimization within our naive strictly convolutional kernel.**



Optimisation 2:

The second optimization was the sweeping of tile size, which primarily led to changes in the block sizes.

Our shared memory usage is still at 0, which is accurate, however our block size and grid size changed as we swept the TILE_WIDTH parameter. We tested several values between 8 and 32 (with 32 being the max size due to the total amount of threads being equivalent to block size squared, capped at 1024).

A block width of 8 resulted in an op time of 228.37 ms, drastically increasing the time required as compared to 148.82 ms taken by the base tile width of 16. This likely results from how with smaller tile widths, consecutive memory reads are limited and causes less burst read efficiency.

Properties

mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int, int)

Queued	n/a
Submitted	n/a
Start	4.67408 s (4,674,081,8...
End	4.89481 s (4,894,806,6...
Duration	220.72484 ms (220,724...
Stream	Default
Grid Size	[10000,24,16]
Block Size	[8,8,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Occupancy	
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

A block width of 32 resulted in an op time of 151.65 ms, a similar length as when block width was 16. While potentially larger burst memory reads are available, larger blocks can cause more threads to have control divergence.

Properties

mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int, int)

Queued	n/a
Submitted	n/a
Start	4.70654 s (4,706,541,1...
End	4.97384 s (4,973,842,7...
Duration	267.30159 ms (267,301...
Stream	Default
Grid Size	[10000,24,4]
Block Size	[26,26,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Occupancy	
Theoretical	68.8%
Shared Memory Configuration	
Shared Memory Executed	
Shared Memory Bank Size	4 B

The peak occupancy based on block size, register usage, shared memory usage, etc.

A block width of 26 resulted in an op time of 267.30159 ms, which most likely due to increased control divergence, similar to why a block width of 32 takes a longer time as well.

A block width of 30 resulted in an op time of 145.579 ms which is much slower than our original of 16. Again, the reasoning it behind this seems to come from an increased control divergence.

A block width of 28 resulted in a op time of 129.31 ms, a significantly faster time compared to the original width. We believe this to find the ideal balance between limiting control divergence and optimizing burst memory reads. **Our end conclusion for this optimization is that it is a suitable optimization and worth including in the final kernel. We recognize that if significant changes happen to the algorithm, we will have to sweep the values again as tile width of 28 was specific to this kernel, and may not be the ideal number for future kernel implementations.**

mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)

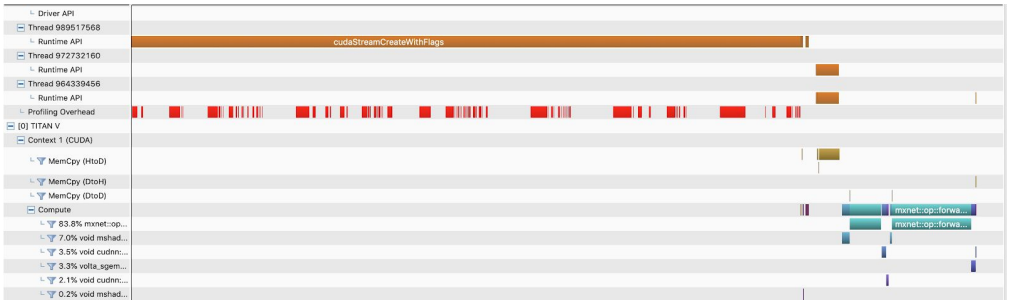
Queued	n/a
Submitted	n/a
Start	4.82843 s (4,828,431,3...
End	4.95775 s (4,957,748,7...
Duration	129.31737 ms (129,317...
Stream	Default
Grid Size	[10000,24,1]
Block Size	[28,28,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Occupancy	
Theoretical	78.1%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B

Optimisation 3:

For our third optimisation, we put the convolution masks in constant memory. The total size of the masks is M * C * K * K (unique mask for each channel of each input feature map). This led to a total time of 123.75 ms which is much faster than our original. Through printf statements, we figured out the values of the M, C and K and hard-coded these values to allocate constant memory. Depending on which convolutional layer it is (the first layer has C = 1, and the second layer has C = 12), we filled up the constant memory and used it accordingly in our convolution. This optimisation sped up our code because it decreased the number of global memory accesses, instead using the much shorter latency memory of constant memory for the frequently used data. We also split the

mxnet::op::forward_kernel(float*, float const *, int, int, int, int, int, int)

Queued	n/a
Submitted	n/a
Start	4.69177 s (4,691,768,517 ns)
End	4.81552 s (4,815,519,939 ns)
Duration	123.75142 ms (123,751,42...
Stream	Default
Grid Size	[10000,24,1]
Block Size	[28,28,1]
Registers/Thread	32
Shared Memory/Block	0 B
Launch Type	Normal
Occupancy	
Theoretical	78.1%
Shared Memory Configuration	
Shared Memory Executed	0 B
Shared Memory Bank Size	4 B



operations into specialized kernels, which also optimized as it allowed for precisely the necessary constant memory to be instantiated, getting all of the benefits of constant memory access while not using time setting up extra memory for the smaller set of masks. **Our overall conclusion was that constant memory lead to a considerable increase in speed and will be included in our final kernel.**

NOTE: When trying to pursue deeper analysis our team encountered this issue and even after asking on Piazza and after trying on EWS and on local on with a batch size of 10,000 images there was still no fix.