Names: Brian Tang (bt3), Joseph Adamo (jdadamo2), Kyle Jew (kjew2)

Team Name: thread_beast

Affiliation: On-Campus students

List of kernels consuming more than 90% of program time:

None, highest is [CUDA memcpy HtoD] at 30.04% of time.

List of CUDA API calls consuming more than 90% of program time:

None, highest is cudaStreamCreateWithFlags at 41.19% of time.

Kernels and API calls difference:

Kernels are programmer-defined C functions and when launched, are executed N times in parallel by N different threads. However, CUDA API calls are pre-defined extensions to the C language and meant for easing the experience for programmers to set up programs for execution by the device.

Output of rai running MXNET on CPU:

“Loading fashion-mnist data... done

Loading model... done

New Inference

EvalMetric: {'accuracy': 0.8154}

18.26user 4.46system 0:09.56elapsed 237%CPU (0avgtext+0avgdata 6047060maxresident)k

0inputs+2824outputs (0major+1601873minor)pagefaults 0swaps”

Program run time: 9.56 seconds

Output of rai running MXNET on GPU:

“Loading fashion-mnist data... done

Loading model... done

New Inference

EvalMetric: {'accuracy': 0.8154}

4.97user 3.25system 0:04.59elapsed 179%CPU (0avgtext+0avgdata 2968484maxresident)k

0inputs+4536outputs (0major+733238minor)pagefaults 0swaps”

Program run time: 4.59 seconds

Whole Program Execution Time: 1 minute 16.47 seconds

Op Times:

Python m2.1.py:

Op Time: 12.307846

Op Time: 59.309954

Correctness: 0.7653

At 100 images:

Op Time: 1.082469

Op Time: 5.923644

Correctness: 0.767

At 1000 images:

Op Time: 0.108870

Op Time: 0.590093

Correctness: 0.76

At 10000 images:

Op Time: 10.855807

Op Time: 60.478481

Correctness: 0.7653

Milestone 3:

Correctness and Timing at 100 images:

Op Time: 0.000282

Op Time: 0.000924

Correctness: 0.76 Model: ece408

4.84user 2.65system 0:06.72elapsed 111%CPU (0avgtext+0avgdata 2783704maxresi

dent)k

0inputs+4560outputs (0major+636682minor)pagefaults 0swaps

At 1000 images:

Op Time: 0.002764

Op Time: 0.009408

Correctness: 0.767 Model: ece408

4.82user 2.77system 0:04.38elapsed 173%CPU (0avgtex

t+0avgdata 2811072maxresident)k

0inputs+4560outputs (0major+641440minor)pagefaults 0swaps

At 10000 images:

Op Time: 0.027439

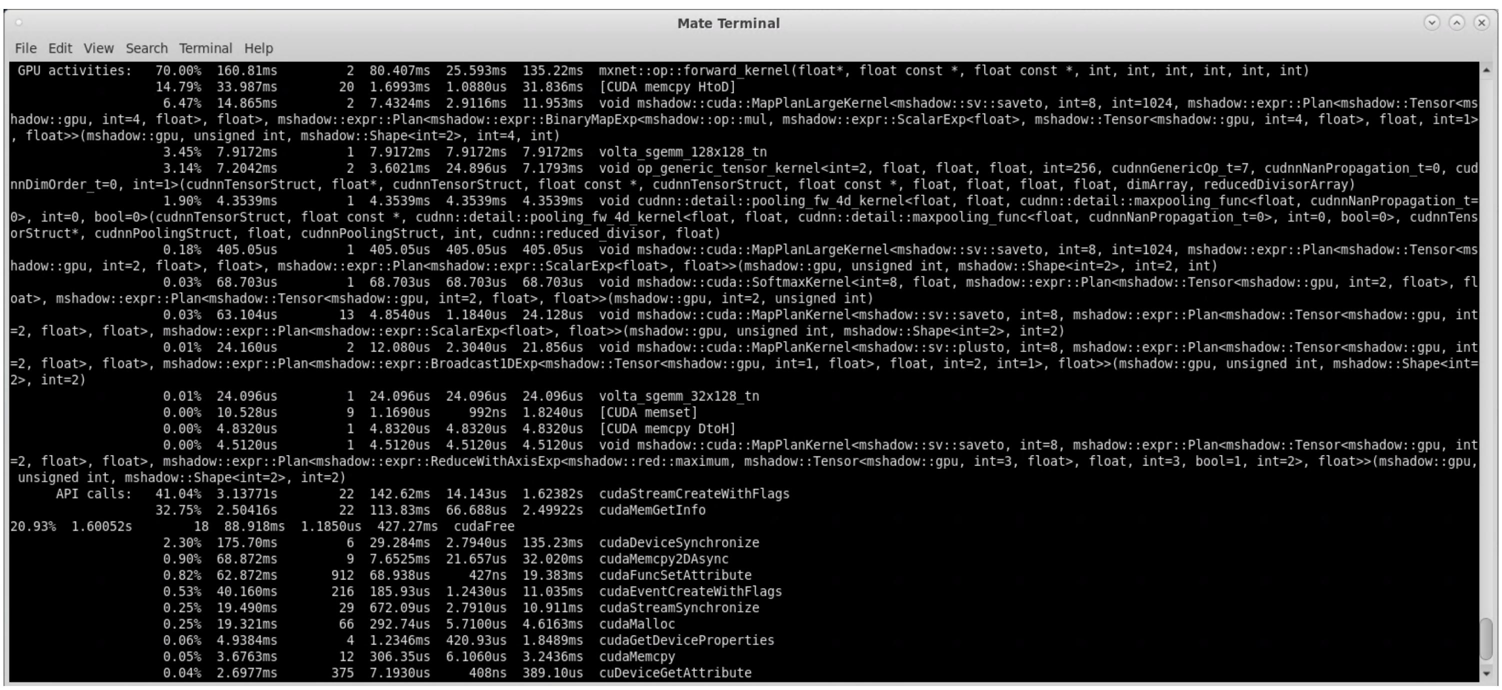
Op Time: 0.093477

Correctness: 0.7653 Model: ece408

5.19user 3.16system 0:04.87elapsed 171%CPU (0avgtext+0avgdata 2981280maxresident)k

0inputs+4560outputs (0major+734975minor)pagefaults 0swaps

NVPROF Execution:



```
Mate Terminal
File  Edit  View  Search  Terminal  Help
 GPU activities:   70.00%  160.81ms         2  80.407ms  25.593ms  135.22ms  mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
                   14.79%  33.987ms        20  1.6993ms  1.0880us  31.836ms  [CUDA memcpy HtoD]
                    6.47%  14.865ms         2  7.4324ms  2.9116ms  11.953ms  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)
                    3.45%  7.9172ms         1  7.9172ms  7.9172ms  7.9172ms  volta_sgemm_128x128_tn
                    3.14%  7.2042ms         2  3.6021ms  24.896us  7.1793ms  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)
                    1.90%  4.3539ms         1  4.3539ms  4.3539ms  4.3539ms  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)
                    0.18%  405.05us         1  405.05us  405.05us  405.05us  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)
                    0.03%  68.703us         1  68.703us  68.703us  68.703us  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)
                    0.03%  63.104us        13  4.8540us  1.1840us  24.128us  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)
                    0.01%  24.160us         2  12.080us  2.3040us  21.856us  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)
                    0.01%  24.096us         1  24.096us  24.096us  24.096us  volta_sgemm_32x128_tn
                    0.00%  10.528us         9  1.1690us     992ns  1.8240us  [CUDA memset]
                    0.00%  4.8320us         1  4.8320us  4.8320us  4.8320us  [CUDA memcpy DtoH]
                    0.00%  4.5120us         1  4.5120us  4.5120us  4.5120us  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:   41.04%  3.13771s        22  142.62ms  14.143us  1.62382s  cudaStreamCreateWithFlags
                   32.75%  2.50416s        22  113.83ms  66.688us  2.49922s  cudaMemGetInfo
20.93%  1.60052s        18  88.918ms  1.1850us  427.27ms  cudaFree
                    2.30%  175.70ms         6  29.284ms  2.7940us  135.23ms  cudaDeviceSynchronize
                    0.90%  68.872ms         9  7.6525ms  21.657us  32.020ms  cudaMemcpy2DAsync
                    0.82%  62.872ms       912  68.938us     427ns  19.383ms  cudaFuncSetAttribute
                    0.53%  40.160ms       216  185.93us  1.2430us  11.035ms  cudaEventCreateWithFlags
                    0.25%  19.490ms        29  672.09us  2.7910us  10.911ms  cudaStreamSynchronize
                    0.25%  19.321ms        66  292.74us  5.7100us  4.6163ms  cudaMalloc
                    0.06%  4.9384ms         4  1.2346ms  420.93us  1.8489ms  cudaGetDeviceProperties
                    0.05%  3.6763ms        12  306.35us  6.1060us  3.2436ms  cudaMemcpy
                    0.04%  2.6977ms       375  7.1930us     408ns  389.10us  cuDeviceGetAttribute
```

Many issues with trying to install NVVP. We had the disk space failure problem and then referred to the Instructor’s answer in the Piazza Post @352. The steps detailed by Ayush were not successful for us, as we were being denied access to install the runfile from CUDA download page. “Access Denied. The username you have entered cannot authenticate with Duo Security. Please contact system administrator”.

As it seems there are no Office Hours until Monday earliest, please excuse us our allow a late submission for this Nvidia profiling portion. We have been successful in performing everything else required in this milestone but have run into logistic problems with NVVP (it seems many other groups have the same problems).

Milestone 4:

Optimization 1: Shared Memory Convolution

Due to the exclusive use of global memory, the basic convolution kernel is extremely limited by the memory bandwidth. This is a lot slower than the peak performance speeds of GPUs, so a good place to start when it comes to optimizations is to reduce global memory reads. For this optimization we achieve this by placing x and k into blocks of shared memory. We allocate a block of size (TILE_WIDTH + K - 1) * (TILE_WIDTH + K - 1) of memory for x and a block of size K * K for k. For each input feature map c, we then use the 1st K * K threads of a given block to load in k, then had all threads load in x. Once k and x are loaded into shared memory, we do the standard convolution where each thread within the acceptable domain preforms K * K operations before moving onto the next input feature map c and repeating the process of loading and calculating. (Results and conclusions provided with NVPROF evidence below)

Optimization 2: Unrolling + Matrix Multiplication

We also set out to try to fundamentally alter the method of forward-propagation calculation and see if it had any impact on performance. As described in Chapter 16 of the book, it's possible to unroll the inputs k and x into large matrices to that the convolution step becomes a simple matrix multiplication. We unrolled k from a 4D M * C * K * K tensor to a 2D M * (C*K*K) matrix. This only had to be done once due to k being the same for all batch elements. For x, we iterated over batch elements B on the CPU and for each element b, unrolled it form a 3D array of size H * C * W to a 2D array of size ((W-K+1)*(H-K+1)) * (C*K*K), after which we performed a shared-memory matrix multiplication in a separate kernel. We iterated over b sequentially due to the fact that unrolling x duplicated a lot of elements, and for large x inputs unrolling the entire 4D tensor might cause x_unrolled to be too large for our memory. Thus, we expect this method to not work as effectively when B is large as that will require more kernel calls. Despite this, switching to this method might prove highly beneficial, as even if the base unroll + matrix multiply method is comparable in speed to convolution, there are many known ways to improve upon the matrix multiplication kernel to speed it up further. (Results and conclusions provided with NVPROF evidence below)

Optimization 3: Sweeping Parameters for Best Values

As our third optimization attempt, we went over our original convolution implementation and our unrolling + matrix multiplication optimization to see if various parameter tweaks could improve performance even more. Specifically, we experimented with changing block sizes and number of threads until we found a value that gave the best performance. Generally, we noticed that tweaking block sizes were not as flexible because of resulting memory access problems. However, we found success in experimenting with thread counts because with certain block dim parameters, we can find ways to minimize control divergence. (Results and conclusions provided with NVPROF evidence below)

NVPROF Performance Analysis and Comparison

Original Implementation (No Optimizations)

```
* Running nvprof python m4.1.py
Loading fashion-mnist data... done
==268== NVPROF is profiling process 268, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.025635
Op Time: 0.135242
Correctness: 0.7653 Model: ece408
==268== Profiling application: python m4.1.py
==268== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   70.00%  160.81ms         2  80.407ms  25.593ms  135.22ms  mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
                   14.79%  33.987ms        20  1.6993ms  1.0880us  31.836ms  [CUDA memcpy HtoD]
                    6.47%  14.865ms         2  7.4324ms  2.9116ms  11.953ms  void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<msha
dow::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)
                    3.45%  7.9172ms         1  7.9172ms  7.9172ms  7.9172ms  volta_sgemm_128x128_tn
                    3.14%  7.2042ms         2  3.6021ms  24.896us  7.1793ms  void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropag
ation_t=0, cudnnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedD
ivisorArray)
                    1.90%  4.3539ms         1  4.3539ms  4.3539ms  4.3539ms  void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNan
Propagation_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)
                    0.18%  405.05us         1  405.05us  405.05us  405.05us  void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<msha
dow::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)
                    0.03%  68.703us         1  68.703us  68.703us  68.703us  void mshadow::cuda::SoftmaxKernel<int=8, float, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int
```

Best Parameters for Original Implementation (Optimization 3)

```
* Running nvprof python m4.1.py
Loading fashion-mnist data... done
==267== NVPROF is profiling process 267, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.030198
Op Time: 0.093225
Correctness: 0.7653 Model: ece408
==267== Profiling application: python m4.1.py
==267== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   60.97%  123.30ms         2  61.651ms  30.128ms  93.175ms  mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
                   20.76%  41.974ms        20  2.0987ms  1.0560us  39.441ms  [CUDA memcpy HtoD]
                    8.30%  16.782ms         2  8.3908ms  3.0264ms  13.755ms  void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<ms
hadow::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)
                    3.87%  7.8284ms         1  7.8284ms  7.8284ms  7.8284ms  volta_sgemm_128x128_tn
                    3.59%  7.2686ms         2  3.6343ms  25.408us  7.2432ms  void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cud
nnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
                    2.19%  4.4386ms         1  4.4386ms  4.4386ms  4.4386ms  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>, cudnnTens
orStruct*, cudnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
                    0.21%  429.44us         1  429.44us  429.44us  429.44us  void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<ms
hadow::gpu, int=2, float>, float>, mshadow::expr::Plan<mshadow::expr::ScalarExp<float>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=2, int)
```

When looking for parameter optimizations for our original milestone 3 implementation of convolution, we discovered a large room for improvement. We believe that our original block dimensions were causing lots of control divergence which resulted in more GPU usage and longer op time. By using blocks with 256 threads instead of 1024, we found a 10% boost in kernel GPU efficiency and 0.04s speedup.

Shared Memory Convolution (Optimization 1)

```
* Running nvprof python m4.1.py
Loading fashion-mnist data... done
==267== NVPROF is profiling process 267, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.045119
Op Time: 0.129944
Correctness: 0.7653 Model: ece408
==267== Profiling application: python m4.1.py
==267== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   71.19%  175.01ms         2  87.506ms  45.087ms  129.93ms  mxnet::op::forward_kernel(float*, float const *, float const *, int, int, int, int, int, int)
                   13.66%  33.590ms        20  1.6795ms  1.1200us  31.432ms  [CUDA memcpy HtoD]
                    6.90%  16.956ms         2  8.4778ms  3.0478ms  13.908ms  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>>(m
hadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                    3.23%  7.9517ms         1  7.9517ms  7.9517ms  7.9517ms  volta_sgemm_128x128_tn
                    2.96%  7.2666ms         2  3.6333ms  24.960us  7.2416ms  void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cudnnDim
rder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
                    1.79%  4.4004ms         1  4.4004ms  4.4004ms  4.4004ms  void cudnn::detail::pooling_fw_4d_kernel<float, float, cudnn::detail::maxpooling_func<float, cudnnNanPropagation_t=0>, i
t=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*,
udnnPoolingStruct, float, cudnnPoolingStruct, int, cudnn::reduced_divisor, float)
```

Even though our functionality for shared memory convolution was sound and conceptually it made sense as an optimization, we were surprised to more GPU activity by our forward kernel compared to the original convolution implementation (by about 1.19%). We believe an explanation for this is that since our shared memory implementation required much more boundary checks, we created more room for control divergence However, we did see a small amount of speedup. This makes sense because using shared memory optimizes the speed, considering that accesses to shared memory is much faster than global memory accesses.

Unrolling + Matrix Multiplication (Optimization 2)

```
* Running nvprof python m4.1.py
Loading fashion-mnist data... done
==267== NVPROF is profiling process 267, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.161680
Op Time: 0.242704
Correctness: 0.7653 Model: ece408
==267== Profiling application: python m4.1.py
==267== Profiling result:
Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   57.54%  232.76ms     20000  11.637us  6.7200us  23.360us  mxnet::op::matrix_multiply(float*, float*, float*, int, int, int)
                   25.33%  102.48ms     20000  5.1240us  3.5830us  17.888us  mxnet::op::unroll_x_kernel(int, int, int, int, float*, int, float*)
                    8.51%  34.441ms        20  1.7220ms  1.0880us  32.305ms  [CUDA memcpy HtoD]
                    3.67%  14.842ms         2  7.4211ms  2.9354ms  11.907ms  void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<ms
hadow::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)
                    1.93%  7.8273ms         1  7.8273ms  7.8273ms  7.8273ms  volta_sgemm_128x128_tn
                    1.78%  7.2106ms         2  3.6053ms  25.056us  7.1856ms  void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cud
nnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
                    1.08%  4.3740ms         1  4.3740ms  4.3740ms  4.3740ms  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>, cudnnTens
```

With unrolling and matrix multiplication, we saw our optimization pay off with improvements in kernel GPU usage. Each individual kernel involved in this optimization (unroll and matrix multiply) was more efficient in terms of GPU usage compared to our original convolution implementation (70% kernel GPU activity). However, one downside is that we noticed this optimization affected the total op time, taking about 0.11 seconds longer than the original convolution. This may be attributed to the fact that we are looping through batch elements and launching two kernels.

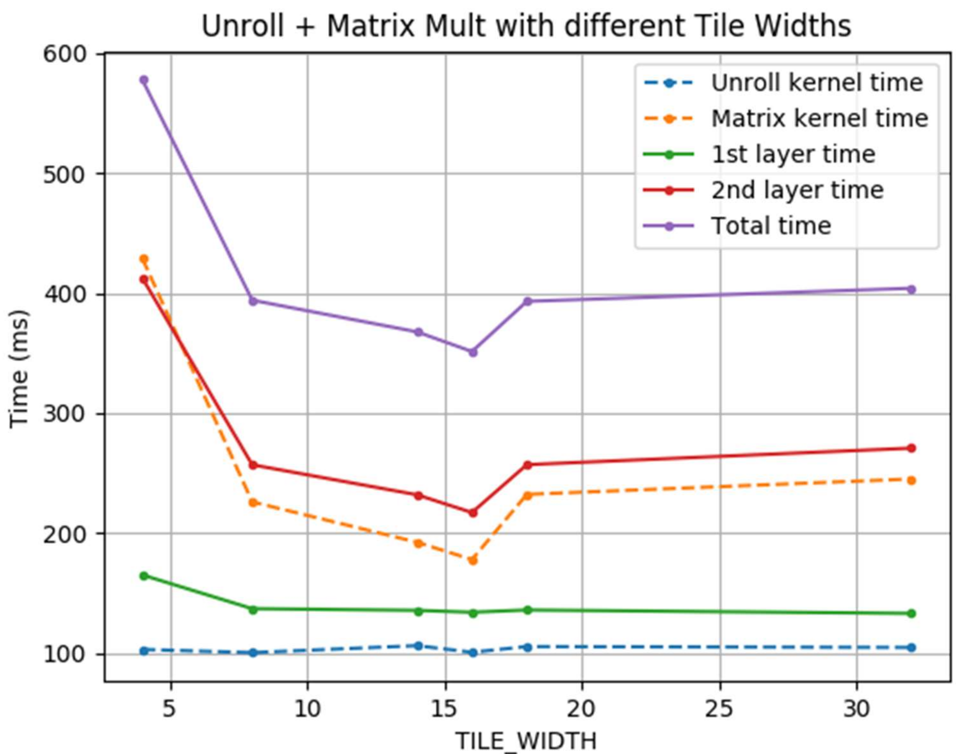
Best Parameters for Unrolling + Matrix Multiplication (Optimization 3)

```
* Running nvprof python m4.1.py
Loading fashion-mnist data... done
==269== NVPROF is profiling process 269, command: python m4.1.py
Loading model... done
New Inference
Op Time: 0.134484
Op Time: 0.240742
Correctness: 0.7653 Model: ece408
==269== Profiling application: python m4.1.py
==269== Profiling result:
Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   54.38%  205.55ms     20000  10.277us  4.6710us  18.207us  mxnet::op::matrix_multiply(float*, float*, float*, int, int, int)
                   27.05%  102.24ms     20000  5.1120us  3.8720us  20.800us  mxnet::op::unroll_x_kernel(int, int, int, int, float*, int, float*)
                    9.35%  35.342ms        20  1.7671ms  1.1200us  32.947ms  [CUDA memcpy HtoD]
                    3.91%  14.774ms         2  7.3871ms  2.9435ms  11.831ms  void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<ms
hadow::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)
                    2.07%  7.8267ms         1  7.8267ms  7.8267ms  7.8267ms  volta_sgemm_128x128_tn
                    1.92%  7.2630ms         2  3.6315ms  25.119us  7.2379ms  void op_generic_tensor_kernel<int=2, float, float, float, int=256, cudnnGenericOp_t=7, cudnnNanPropagation_t=0, cud
nnDimOrder_t=0, int=1>(cudnnTensorStruct, float*, cudnnTensorStruct, float const *, cudnnTensorStruct, float const *, float, float, float, float, dimArray, reducedDivisorArray)
```

When sweeping parameters to improve performance for unrolling+matrix multiplication, we looked into the grid and block dimensions that we used to launch the unroll kernel and matrix multiply kernel. We did not have much success changing the block size used to launch the unroll kernel because any size other than 1024 would cause an illegal memory access. However, tweaking block dimensions for the matrix multiply kernel helped. By changed the number of threads in the block from 576 (blockDim(24,24)) to 256 (blockDim(16,16)), we noticed a 3% more efficient GPU activity (from 57.54% to 54.38%). Additionally, the op time dropped by .03 seconds (0.134484 s to 0.161680 s). This success can be attributed to the fact we found the most optimal thread count in a block such that there is less control divergence happening in the kernel.

Final Milestone Report

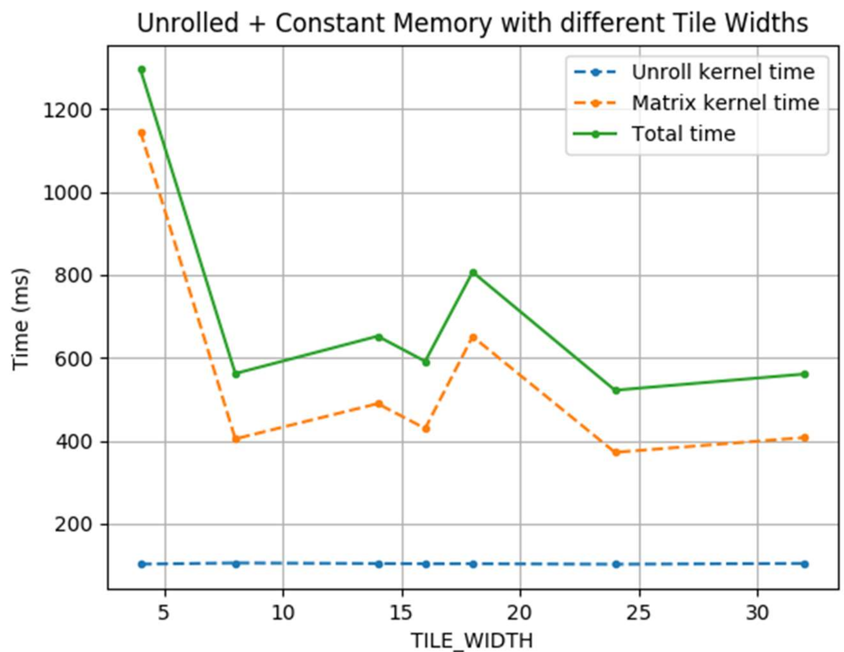
All the optimizations that we tried in this part in some way build off the implementation within file new-forward4-2.cuh, which holds an unroll + matrix multiplication kernel. The following graph shows a parameter sweep of TILE_WIDTH for new-forward4-2.cuh. All analysis shown uses a data size of 10,000.



From this graph we can see that for the time to undergo the matrix multiplication kernel is minimized with TILE_WIDTH = 16, while the unroll kernel is mostly unaffected by the tile size changing. This makes sense, as our implementation of the unroll kernel doesn't use any sort of tiling, and thus changing the block size should have no effect on performance. The fact that an increased TILE_WIDTH doesn't necessarily cause an increase in performance is interesting. Our thinking is that, while an increased TILE_WIDTH allows for more efficient use of global memory, it also puts a higher strain on the shared memory both in regard to the size of the block and how many threads are accessing it at a given time. Also, from this graph it's clear that the matrix multiplication takes up a significant portion of the total time of execution, so a logical conclusion seems to be that focusing on the matrix kernel will substantially increase performance.

Optimization 4 – w tensor in constant memory

Based on this conclusion, we implemented **placing w in constant memory** as an optimization to reduce dependence on the memory bandwidth. Our code for this implementation is in new-forward5-1.cuh. In our regular unroll + matrix multiply implementation, w and x are loaded into shared memory on a tile-by-tile basis, so each thread loads 2 * W_unroll / TILE_WIDTH elements from global memory. Our reasoning was that by loading w into constant memory this would be reduced to W_unroll / TILE_WIDTH reads, leading to a reduced dependence on memory bandwidth. After implementing this, we ran another sweep over TILE_WIDTH like for the new-forward4-2, and from that sweep produced the following graph.



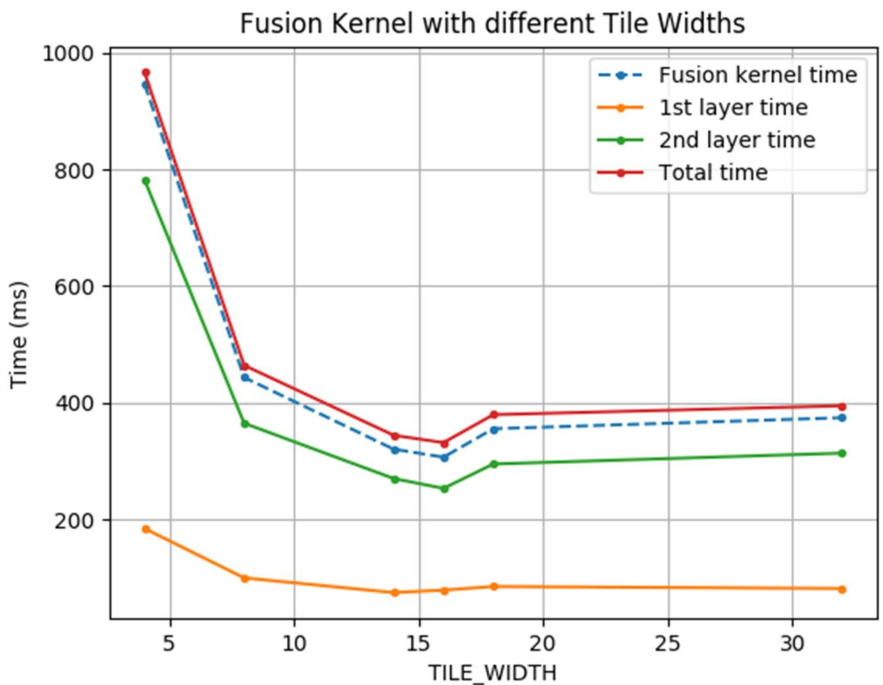
This graph is much harder to interpret. It appears that there are several dips in runtime at TILE_WIDTH = 16, 24, and that once again there is no effect on the unroll kernel by changing TILE_WIDTH. More concerning, however, is that the total time for all values of TILE_WIDTH is larger than the analogous times for new-forward4-2.cuh. This might be happening because, while each memory read from constant memory is faster than a global read, our implementation reads those values directly into the calculation of Y_val, and so there isn't any reduction in constant memory reads. After trying to load the constant memory into a shared tile, we did see a performance increase when compared to loading from global memory directly, as can be seen by the following output from nvprof.

```
New Inference
Op Time: 0.151861
Op Time: 0.364859
Correctness: 0.1 Model: ece408
==272== Profiling application: python final.py
==272== Profiling result:
Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   67.16%  358.63ms     20000  17.931us  7.3920us  28.736us  mxn
et::op::matrix_multiply(float const *, float const *, float*, int, int, int)
                   19.60%  104.64ms     20000  5.2320us  3.9990us  14.816us  mxn
et::op::unroll_x_kernel(int, int, int, int, float*, int, float const *)
                    6.73%  35.931ms        20  1.7966ms  1.0880us  33.653ms  [CU
DA memcpy HtoD]
                    2.77%  14.792ms         2  7.3958ms  2.9226ms  11.869ms  voi
d mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshado
w::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>, mshadow::expr
::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul, mshadow::expr::ScalarExp<fl
oat>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>, float>>(mshado
w::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)
                    1.47%  7.8285ms         1  7.8285ms  7.8285ms  7.8285ms  vol
ta_sgemm_128x128_tn
```

This run was done with TILE_WIDTH=16 loading const memory into shared memory. If we compare this output to the above graphs, it's clear that this matrix_multiply kernel has a better performance compared to graph 2 but comparing to graph 1 shows there is still a major performance decrease. For this reason, we do not use constant memory in our final version of our code. This optimization was primarily written by Brian and Kyle, with Joe doing the parameter sweep and further debugging to increase our runtimes with this implementation.

Optimization 5 – Fused unroll + matrix multiply kernel

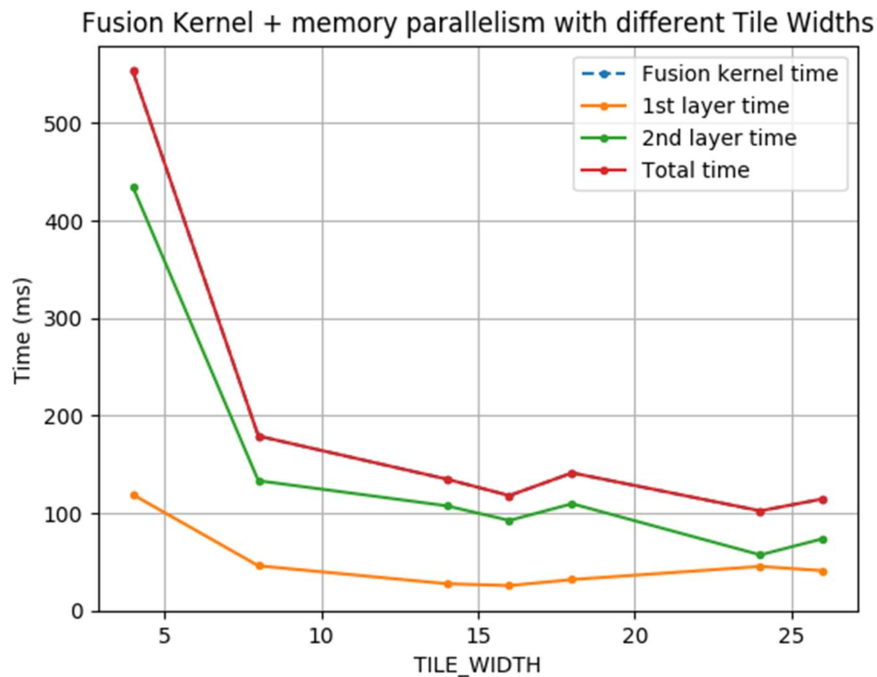
Shifting focus slightly, from the above analysis we can see that the unroll kernel, while taking less time than matrix multiplication, spends a significant amount of time unrolling the x tensor. Because the act of unrolling can be achieved purely conceptually, by assigning x within the matrix multiply kernel such that the result is the same as if you unrolled x explicitly, one can expect to save time by **fusing the unroll and multiplication kernels**. This would also reduce overhead regarding memory, as you are no longer assigning, filling, and freeing a separate unrolled x matrix, which both saves time and memory space. Optimistically, we expected an increase in total performance on the order of 50 – 100 ms, as while additional indexing is required in matrix multiplication, the time of around 100 – 110 ms by the unroll kernel is removed. The following graph shows another sweep of TILE_WIDTH for our implementation of the fusion kernel.



Once again there appears to be a dip where TILE_WIDTH = 16 and comparing to graph 1 the total time is comparable to slightly better than for the unroll + matrix multiply case. (317.865 ms vs 351.302 ms). While not a major improvement, this is significant enough that our final version implements a version of this fused kernel. More importantly, we no longer utilize a separate x_unroll matrix, freeing up a block of memory of size W_unrolled*H_unrolled. This optimization was written by Brian and Kyle, while Joe swept through the parameters and debugged to further improve our runtimes.

Optimization 6 – Exploiting parallelism and multiple implementations

With the fusion kernel freeing up a significant amount of memory, it was then possible for us to take further advantage of the parallelism of the input data. Previously, due to the memory expansion used by x_unrolled, it became impractical to load the entire input tensor into global memory, as once it got expanded it would more than likely exceed the total memory storage. This meant that we had to loop through the batch elements and do a separate unroll + matrix multiply kernel execution for each batch. Now however, exceeding the total memory is no longer as large of a concern, and we can thus once again launch with the entire input data in global memory, **further exploiting parallelism in the input data**. We accomplished this by extending our grid dimensions by B in the z dimension, keeping each blockDim.z equal to 1. This allows us to execute each batch in parallel, which means that we should see a significant increase in performance. Once again, we swept through several TILE_WIDTH values, resulting in the following graph.

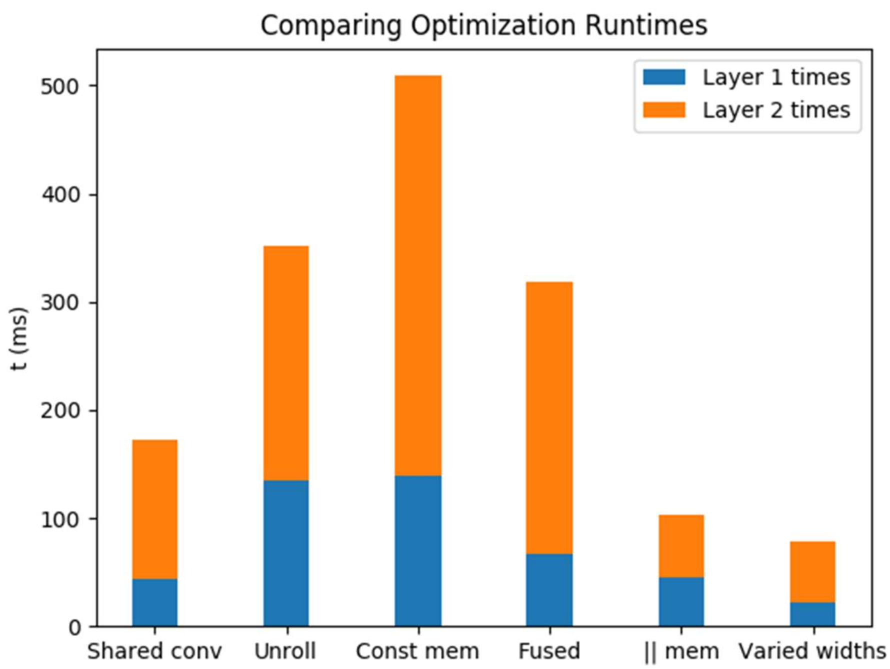


In addition to a significant increase in performance when compared to graph 1, there appears to once again be a dip where TILE_WIDTH = 16. Now however, there is a further dip where TILE_WIDTH = 24 for the second, larger layer, suggesting that after further parallelizing our input data, the most effective value for TILE_WIDTH changes as the convolution layer increases. Given this information, we implemented **multiple kernel implementations** depending on the size of the input layer, using the fused kernel with input parallelism with different values for TILE_WIDTH based on the above graph (we used TILE_WIDTH = 16 and 24). A screenshot of the output form nvprof is given below.

```
New Inference
Op Time: 0.025708
Op Time: 0.062750
Correctness: 0.7653 Model: ece408
==272== Profiling application: python final.py
==272== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   39.36%  62.730ms         1  62.730ms  62.730ms  62.730ms  mxnet::op::ma
trix_multiply_big(float const *, float const *, float*, int, int, int, int, int)
                   22.59%  36.005ms        20  1.8002ms  1.0880us  33.684ms  [CUDA memcpy
HtoD]
                   16.11%  25.677ms         1  25.677ms  25.677ms  25.677ms  mxnet::op::ma
trix_multiply_small(float const *, float const *, float*, int, int, int, int, int)
                    9.31%  14.835ms         2  7.4176ms  2.9097ms  11.926ms  void mshadow:
:cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshado
w::Tensor<mshadow::gpu, int=4, float>, float>, mshadow::expr::Plan<mshadow::expr::BinaryMa
pExp<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)
                    5.00%  7.9633ms         1  7.9633ms  7.9633ms  7.9633ms  volta_sgemm_1
28x128_tn
```

Notice now how there are two instances of matrix_multiply – one for each TILE_WIDTH size – and that, comparing to graph 4 – these times correspond to the optimal times for each layer. Also, of note is that our times are now comparable to the time required for memcpy to execute. This indicates that future optimizations that focus on reducing the impact of this step, perhaps by impending streaming in the host code to simultaneously, may help further increase performance. For our group, Joe focused on the multiple-kernel implementation, while Brian and Kyle handled the parallelization.

The following graph is given as a summary of each optimization we tried for this milestone, and the best times we achieved with each. We include the shared-memory convolution and base unroll + matrix multiply from the previous milestone as a reference.



All times shown on the bar graph were determined from running /user/local/time. The final time we achieved according to user/local/time was 72.938 ms, which was achieved by the final bar as shown here. It's clear from this graph that we were able to achieve a substantial performance increase by implementing the above optimizations, specifically 5 and 6. These optimizations achieved roughly a factor of 5 performance improvement when compared to just the unroll + matrix multiply As stated in each individual optimization section, Brian and Kyle were the ones that mainly wrote the first 2 and half of the 3rd optimizations, while Joe focused on debugging and analysis of each, as well as writing this report and the 2nd half of optimization 6.