ECE408 Final Project Report

Shriyaa Mittal (smittal6, 653819262), **Tiancheng Wu** (twu54, 674564428), and **Chuankai Zhao** (czhao37, 660893745)

Team Name: smartconvolutionteam

Final Submission: Due December 14, 2018

5.1 Optimization 4: Exploiting parallelism in input images

5.1.1 Describe the optimization

Following the optimization 3 (kernel fusion for unrolling and matrix multiplication on shared memory), we seek to exploit parallelism in input images. Briefly, we assign input images to the third grid dimension, and assign output feature maps and the values of each output feature map to the second and the first grid dimensions. In this way, we increase the level of kernel parallelism, thereby achieving a faster implementation compared to our base code.

In this optimization, we have also adjusted the block size from previous 32, 32, 1 to 24, 24, 1, which further improves kernel performance. We observe that the numbers of output feature maps are 12 and 24 for the two forward pass layers. It is clear that if we use a block size larger than 24, many threads in the block will not be used. We vary the sizes of block x, y dimensions, and finally determine as 24 for the optimal overall performance.

5.1.2 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
Op Time: 0.000636
Op Time: 0.000949
Correctness: 0.85 Model: ece408

4.04user 2.36system 0:04.55elapsed 140%CPU
```

Dataset Size 1000

```
Op Time: 0.006175
Op Time: 0.008948
Correctness: 0.827 Model: ece408

4.19user 2.67system 0:04.32elapsed 158%CPU
```

Dataset Size 10,000

```
Op Time: 0.061277
Op Time: 0.088465
Correctness: 0.8171 Model: ece408

4.16user 2.47system 0:04.73elapsed 140%CPU
```

Our implementation of the optimization 4 has achieved the expected correctness values. Compared to the based code and the previous optimizations, the optimization 4 gives us the fastest kernel executions due to parallelism in input images. For example, it took 195.45 ms to compute the two forward passes on dataset 10,000 with the based code (the fastest kernel by milestone 4), while it took only 149.74 ms after this optimization. Furthermore, the execution time for the second forward pass layer is significantly reduced (152.91 versus 88.47 ms), while the execution time for the first layer is still large compared to the base code (42.53 versus 61.28 ms). Similar results are observed on all three datasets.

5.1.3 Demonstrate nvprof profiling and analyze optimization

After we exploit parallelism in input images, the kernel *matrixMultiplyShared* is executed exactly once in each forward pass layer (Figure 1). This is in contrast with the implementation in the optimization 3, where each input image is loaded one by one and the kernel is executed multiple times in each layer. Due to the increased parallelism, the total running time decreases.

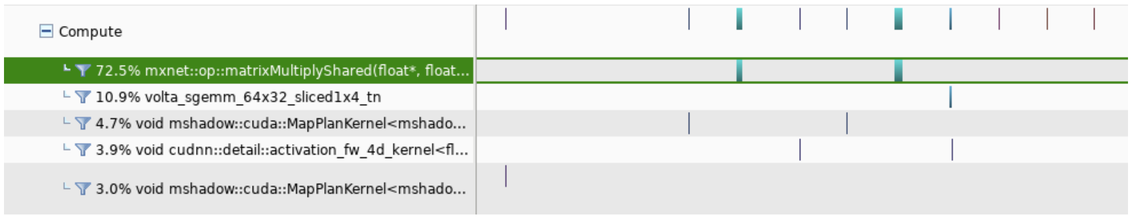

Compute
72.5% mxnet::op::matrixMultiplyShared(float*, float...
10.9% volta_sgemm_64x32_sliced1x4_tn
4.7% void mshadow::cuda::MapPlanKernel<mshado...
3.9% void cudnn::detail::activation_fw_4d_kernel<fl...
3.0% void mshadow::cuda::MapPlanKernel<mshado...

Figure 1: Kernel execution timeline on dataset size 100 based on the optimization 4. Only part of the timeline is shown for clarity.

In this optimization, the total number of threads within each block are 576 (24*24*1). The maximum number of threads in each SM is 2048 in the GPU, which limits the number of blocks to 3 and the theoretical occupancy 84.4%. In the first and the second layers, the achieved occupancy are 81.6% and 82.6%, respectively.

To achieve 100% theoretical occupancy, we can potentially vary block sizes for the kernel, such as 32, 32, 1 or 16, 16, 1. However, the numbers of output feature maps in both layers are 12 and 24. If we choose a large block size, many threads in each block will be wasted, which can limit resource utilization efficiency. If we choose a small block size, the shared memory data reuse efficiency will decrease. In addition, branch divergence can limit kernel performance if the total number of threads in each block is not divisible by 32. For these reasons, we find that a block size of 24 gives us the best performance. We observe that the block sizes for two layers can be optimized independently, which will be addressed in the optimization 5.

As shown in Figure 2, the *matrixMultiplyShared* kernel is bound by memory bandwith in the first layer and by both compute and memory bandwidth in the second layer. In both layers, the kernel is

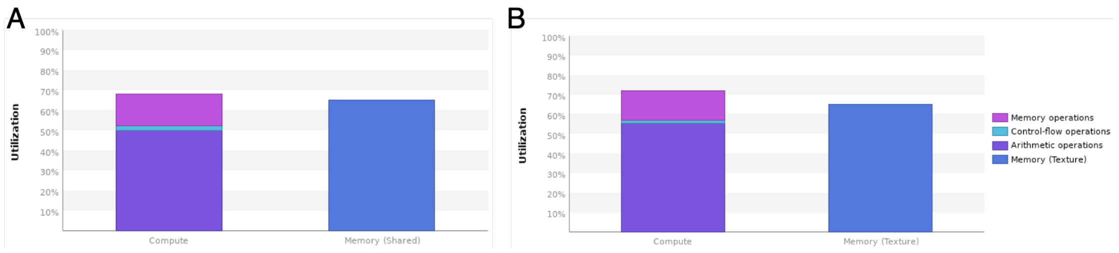
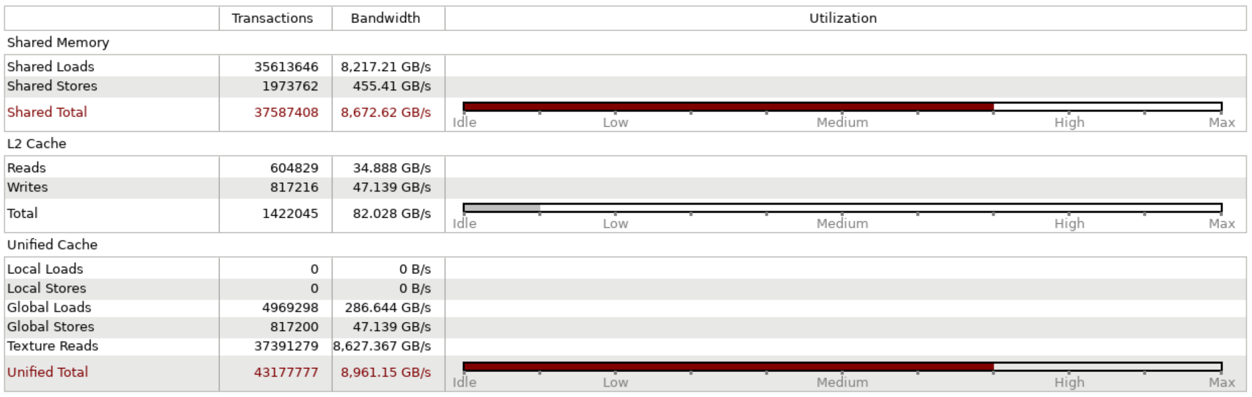


Figure 2: Kernel performances in (A) the first and (B) the second layers running on dataset 100 with the optimization 4.

limited by the bandwidth available to the shared memory (Figure 3). This is resulted from massive read and store operations on shared memory. In terms of global load or store, the bandwidth is effectively reduced by utilizing shared memory. It is suggested that proper alignment and access pattern of global memory load and store can potentially further improve the kernel performance.

The divergences are 2.2% and 3.2% in the first and the second layers. The branch divergence is resulted from the number of elements in each output feature map. Since our block size is 576 which is divisible by 32, we have eliminated potential divergence caused by the block size. We believe that this kernel has achieved relatively low divergence.



	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	35613646	8,217.21 GB/s	
Shared Stores	1973762	455.41 GB/s	
Shared Total	37587408	8,672.62 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	604829	34.888 GB/s	
Writes	817216	47.139 GB/s	
Total	1422045	82.028 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	4969298	286.644 GB/s	
Global Stores	817200	47.139 GB/s	
Texture Reads	37391279	8,627.367 GB/s	
Unified Total	43177777	8,961.15 GB/s	Idle Low Medium High Max

Figure 3: Memory utilization relative to the maximum throughput supported by the memory on the GPU device. Shared memory utilization is highlighted in red.

From the latency analysis, the kernel uses 31 registers for each thread and 17856 registers for each block. Because of this, each SM is limited to simultaneously executing 3 blocks. However, the maximum number of threads in each SM is 2048, which also limits the number of blocks to 3. We believe that register usage is not limiting the kernel performance.

5.2 Optimization 5: Multiple kernel implementations for different layer sizes

5.2.1 Describe the optimization

Following the optimization 4, we exploit separate kernel implementations for the first and the second forward pass layers. First, we seek to use the same kernel as in the optimization 4 but choose different block sizes for the two layers. We vary the sizes of block x, y dimensions, and determine as 16,16 and 24,24 for the optimal performances of both layers. We note that the numbers are chosen such that the total number of threads in each block is divisible by 32 (the warp size), in order to reduce control divergence.

We also seek to utilize the kernel in the base code for the first layer, and the kernel in the optimization 4 for the second layer. We compare the overall performances using these two strategies and find that the implementation with the first strategy gives us better kernel performance. The following results and analysis are based on the first strategy.

5.2.2 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
Op Time: 0.000413
Op Time: 0.000934
Correctness: 0.85 Model: ece408

4.25user 2.60system 0:04.48elapsed 152%CPU
```

Dataset Size 1000

```
Op Time: 0.003942
Op Time: 0.008910
Correctness: 0.827 Model: ece408

4.13user 2.19system 0:04.23elapsed 149%CPU
```

Dataset Size 10,000

```
Op Time: 0.039160
Op Time: 0.083881
Correctness: 0.8171 Model: ece408

4.27user 2.50system 0:04.61elapsed 146%CPU
```

In the case of using the same kernel (as in the optimization 4) but different block sizes in the two forward pass layers (16, 16, 1 and 24, 24, 1), our implementation has achieved the expected correctness values. We have improved kernel performance in the first layer even further. For example, the first kernel execution on dataset 10,000 took 61.28 ms in the optimization 4, and it took only 39.16 ms with this optimization. Now, the total running time including both layers decreases to 123.04 ms. Compared to our base code, the first kernel execution in this optimization is faster. Overall, this optimization has significantly improved kernel performance.

5.2.3 Demonstrate nvprof profiling and analyze optimization

In this optimization, we have only changed the kernel execution for the first layer, while the second kernel execution remains the same as in the optimization 4. Following results and analysis are entirely based on the kernel execution in the first layer.

As expected, the kernel theoretical occupancy in the first layer is 100% after the block size is chosen as 16,16,1 and the achieved occupancy is 96.9%. Even though the number of output feature maps is 12 (less than 16), current block size can eliminate divergence within most blocks (i.e. divergence will exist in every block if block size is 12, 12, 1) and maximize share memory data reuse efficiency. Therefore, the running time in the first layer has greatly reduced.

Compute and memory utilization are balanced in the first kernel execution (Figure 4). Utilization levels suggest that kernel performance is good, and bound by compute and memory bandwidth. The divergence is 3.4%, which is caused at the end of x grid dimension. We believe that this is a relatively low divergence too, given the constraint on dataset size. As similar in the optimization 5, shared memory bandwidth utilization is high relative to the maximum throughput, due to massive operations on shared memory. Kernel is limited by the bandwidth available to the unified cache. Global memory alignment and access pattern can be improved to further enhance kernel performance.

In contrast with the optimization 4, the smaller block size reduces the register uses to 8192 registers per block, allowing each SM executing 8 blocks simultaneously. This is equivalent to the number of blocks supported by each SM.

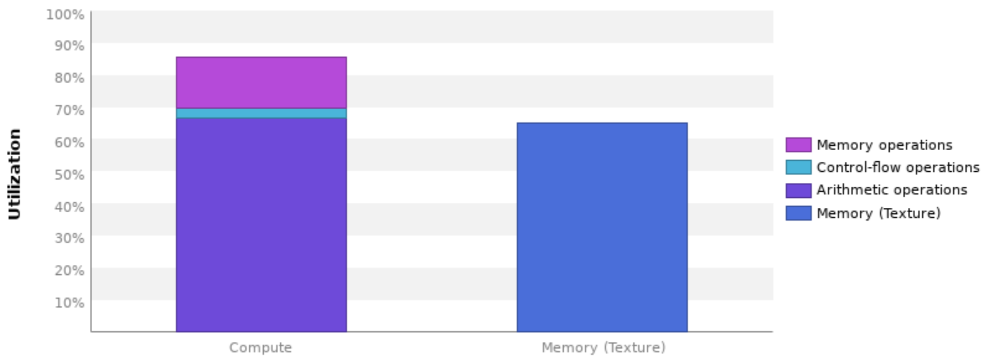


Figure 4: Kernel performance in the first layer with the optimization 5.

5.3 Optimization 6: Tuning with restrict, loop unrolling

5.3.1 Describe the optimization

To further improve the kernel performances in both layers, we seek to tune the kernels in the optimization 5 with restrict and loop unrolling. Briefly, we add both *const* and *__restrict__* before the pointers which point to input images and weight matrices, and add *__restrict__* to the pointer which points to output feature maps. In this way, the kernel can utilize the read-only data cache on the GPUs. In addition, we add *#pragma unroll* before the for loops in the kernels to control loop unrolling by the GPU compiler.

5.3.2 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
    Op Time: 0.000389
    Op Time: 0.000873
    Correctness: 0.85 Model: ece408

    4.33user 2.51system 0:04.44elapsed 154%CPU
```

Dataset Size 1000

```
    Op Time: 0.003673
    Op Time: 0.008403
    Correctness: 0.827 Model: ece408

    3.73user 2.45system 0:04.08elapsed 151%CPU
```

Dataset Size 10,000

```
    Op Time: 0.036532
    Op Time: 0.076987
    Correctness: 0.8171 Model: ece408

    4.12user 2.64system 0:04.41elapsed 153%CPU
```

Our implementation for the optimization 6 has achieved the expected correctness values. Remarkably, this optimization has further enhanced the kernel execution speeds in both layers. Now, the running times in the first and the second kernel executions are 36.53 and 76.99 ms on dataset 10,000, with a total of 113.52 ms.

5.3.3 Demonstrate nvprof profiling and analyze optimization

This optimization does not significantly change compute and memory utilization. The kernel performances in both layers are bound by compute and memory bandwidth. Utilization of both compute and memory are balanced. The kernel is limited by the bandwidth available to the shared memory.

The enhancement of kernel performance is driven by the loop unrolling. In this way, the device can continuously execute the operations without the need to check the *for* loop condition in every iteration. Utilizing the read-only data cache also improves the efficiency in data operation.

5.4 Optimization 7: Shared Memory Convolution

5.4.1 Describe the optimization

Here, we have implemented convolution using tiling such that the input and the kernel are in shared memory. This code is similar to what we did in MP4, except that we did not use constant memory for the mask/kernel. Unfortunately, this method is not fast, we believe loading kernel in the constant memory would be a good choice.

We also experimented with different ways of declaring shared memory, such as with and without dynamic allocation. It seems that the dynamic allocation is faster, we do this using the keyword *extern* in the *forward_kernel* function and the kernel call specifies the shared memory to be allocated. We also tried out multiple tile lengths, and the value 16 gives the best results. With this value, internal tile size is 16 and there are $K - 1$ halo or ghost elements. Ghost elements are treated as 0. The kernel is implemented such that neighbouring threads access consecutive memory elements to ensure memory coalescing.

5.4.2 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
    Op Time: 0.000607
    Op Time: 0.002130
    Correctness: 0.85 Model: ece408
```

```
4.06user 2.45system 0:04.28elapsed 152%CPU
```

Dataset Size 1000

```
    Op Time: 0.005966
    Op Time: 0.021024
    Correctness: 0.827 Model: ece408

    4.30user 2.60system 0:04.45elapsed 155%CPU
```

Dataset Size 10,000

```
    Op Time: 0.058280
    Op Time: 0.191002
    Correctness: 0.8171 Model: ece408

    4.33user 2.68system 0:04.78elapsed 146%CPU
```

Our implementation for optimization 7 has achieved the expected correctness values. This optimization does not enhance the running time of the convolution code as compared to the other optimizations we have implemented. It took 249.28 ms overall to compute the two forward pass layers on dataset with 10,000 images.

5.4.3 Demonstrate nvprof profiling and analyze optimization

We generated the profiling data for dataset with 100 images (Figure 5). Two clear blocks executing the code are seen corresponding to each of the forward pass layers. Achieved occupancy is about 73% whereas the theoretical occupancy is 75%.

The kernel performance is bound by compute. The kernel's memory utilization is significantly lower than the compute utilization as seen in Figure 6A. Further, since both input and the convolution mask are in the shared memory, utilization of shared memory is the highest (Figure 6B).

We use barrier synchronization, *__syncthreads*() call, in three different places - (i) after kernel values are loaded into shared memory is completed, (ii) after input values are loaded into shared memory, (iii) after the convolution sum is collected into a register. Then the final result is written

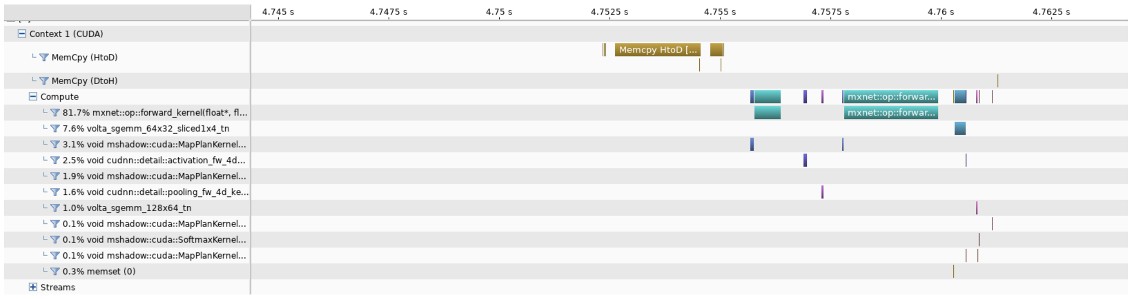
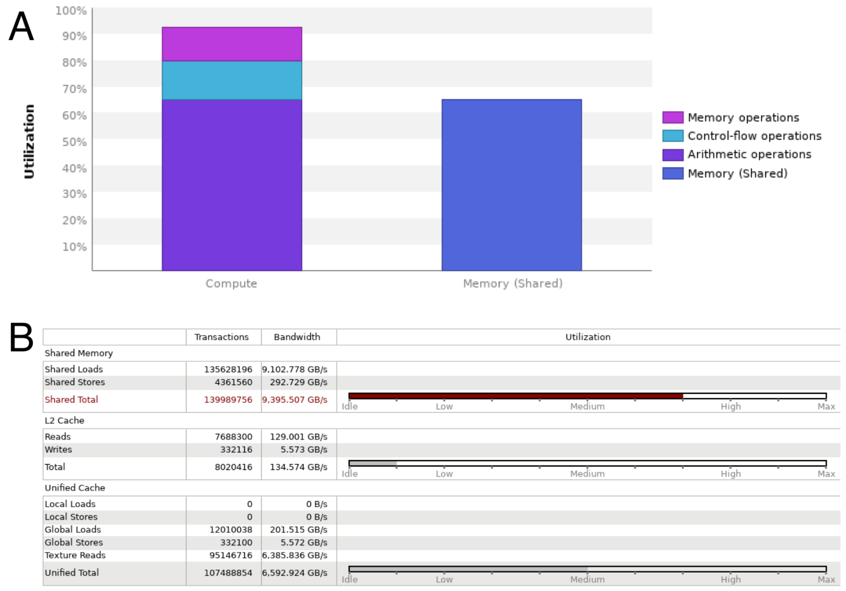

4.745 s
4.7475 s
4.75 s
4.7525 s
4.755 s
4.7575 s
4.76 s
4.7625 s
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
81.7% mxnet::op::forward_kernel(float*, fl...
7.6% volta_sgemm_64x32_sliced1x4_tn
3.1% void mshadow::cuda::MapPlanKernel...
2.5% void cudnn::detail::activation_fw_4d...
1.9% void mshadow::cuda::MapPlanKernel...
1.6% void cudnn::detail::pooling_fw_4d_ke...
1.0% volta_sgemm_128x64_tn
0.1% void mshadow::cuda::MapPlanKernel...
0.1% void mshadow::cuda::SoftmaxKernel...
0.1% void mshadow::cuda::MapPlanKernel...
0.3% memset (0)
Streams
Memcpy HtoD [...
mxnet::op::forwar...
mxnet::op::forwar...

Figure 5: Kernel execution timeline on dataset size 100 based on the optimization 7. Only part of the timeline is shown for clarity.



	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	135628196	9,102.778 GB/s	
Shared Stores	4361560	292.729 GB/s	
Shared Total	139989756	9,395.507 GB/s	
L2 Cache			
Reads	7688300	129.001 GB/s	
Writes	332116	5.573 GB/s	
Total	8020416	134.574 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	12010038	201.515 GB/s	
Global Stores	332100	5.572 GB/s	
Texture Reads	95146716	6,385.836 GB/s	
Unified Total	107488854	6,592.924 GB/s	

Figure 6: (A) Kernel performances in the optimization 7. (B) Memory usage highlighting high usage of shared memory.

to the global memory location. We see synchronization as a major stall reason in our profiling analysis.

Also, there is significant overhead due to calculation of the indices of the memory locations in the shared memory. Hence, we would expect this code to be slow. During our profiling, we observe high utilization of arithmetic instructions among the many types of instructions executed within each SM.

To improve the execution time, we loaded the kernel in the constant memory and the input tile in the shared memory. We then implemented the strategy 3 of the shared memory convolution. Using this also we are able to obtain the expected correctness values, eliminate one instance of $\_\_syncthreads()$, increase usage of constant memory, decrease usage of shared memory. However, the execution time dropped further to 496.8 ms. There is still significant overhead for index calculations which leads to high register usage per thread and less number of blocks that can execute on each SM.

Divergence and consequently warp efficiency are low in both implementations of this optimizations since there are multiple loops that require boundary condition checks. The divergence is different for the two forward kernel pass. Hence, the divergence can be optimized by firstly modulating the block dimension and secondly using two separate block dimensions for the two kernel calls.

5.4 Summary of final submission

In this submission, we have performed four additional optimizations on top of the base code in the milestone 3 and the three optimizations in the milestone 4. The first three optimizations (the optimizations 4-6) are built on top of the optimization 3 (kernel fusion for unrolling and matrix multiplication on shared memory). Specifically, we exploit parallelism in input images, implement seperate kernels for different layer sizes, and utilize loop unrolling and the read-only data cache. Each optimization has contributed to the enhanced kernel performances, leading to a significant improvement compared to our previous milestones. The fastest kernel executions on dataset 10,000 including both forward pass layers is 107.52 ms, which is ranked 32/85 at the time of submission. In addition, we have also explored shared memory convolution as the optimization 7. However, we do not observe performance enhancement with shared memory convolution.

5.5 Team member contributions

All members in the *smartconvolutionteam* team contributed equally in implementing code, profiling optimizations and writing the report.

MileStone 4: Due December 2, 2018

4.1 Optimization 1: Unroll / shared-memory Matrix multiply

4.1.1 Describe the optimization

For each input image, we first produce an unrolled matrix and then do a simple matrix multiplication with kernel to compute the output. Shared memory is used to reduce the memory bandwidth.

4.1.2 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
Op Time: 0.001758
Op Time: 0.005427
Correctness: 0.85 Model: ece408

4.03user 2.72system 0:04.26elapsed 158%CPU
```

Dataset Size 1000

```
Op Time: 0.017142
Op Time: 0.049663
Correctness: 0.827 Model: ece408
```

```
    3.82user 2.47system 0:04.10elapsed 153%CPU
```

Dataset Size 10,000

```
    Op Time: 0.159199
    Op Time: 0.443849
    Correctness: 0.8171 Model: ece408

    4.48user 2.80system 0:04.91elapsed 148%CPU
```

Our implementation of the optimization 1 has achieved the expected correctness values. Compared to the base code, the running times for both forward pass layers on all three datasets increased by three to five times. For example, it takes 0.04 and 0.15 seconds to compute the first and the second forward pass layers on dataset 10,000 using the base code, while it takes 0.16 and 0.44 seconds using the optimized code. This suggests that simple unrolling of input matrix could not improve the computing efficiency.

4.1.3 Demonstrate nvprof profiling and analyze optimization

There are 2 kernels in the optimization, one is matrix multiplication *matrixMultiplyShared* and the other is unrolling *unrollKernel*. From Figure 7 we can see that the nvprof profiling exhibits an interleaving behaviour, where one kernel has to wait for another interchangeably.

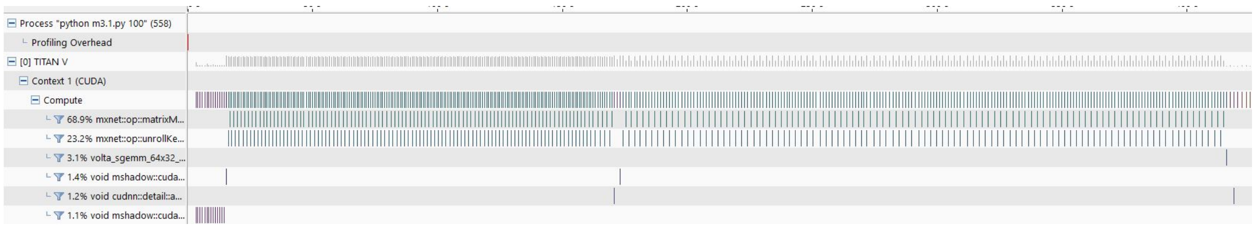

Process "python m3.1.py 100" (558)
Profiling Overhead
[0] TITAN V
Context 1 (CUDA)
Compute
68.9% mxnet::op::matrixM...
23.2% mxnet::op::unrollKe...
3.1% volta_sgemm_64x32_...
1.4% void mshadow::cuda...
1.2% void cudnn::detail::a...
1.1% void mshadow::cuda...

Figure 7: Kernel execution timeline on dataset size 100 based on the optimization 1. Only part of the timeline and the top 6 kernels are shown for clarity.

It is also evident that the kernel performances are limited by the low utilization of all memory types - shared memory, L2 cache, unified cache, device memory and system memory.

The occupation is quite low as well, for the *matrixMultiplyShared* kernel it is 77% and 49.9% for the first and second call respectively, and for the *unrollKernel* it is 79% and 84%. These values are for a single instance of the kernel calls, however, all instances report approximately same observations. These observations can be fixed by exploiting parallelism for the input images as opposed to simply looping through images.

With respect to memory access, there is some inefficiency that can be improved. The *unrollKernel* is bound by compute (Figure 8A) and the *matrixMultiplyShared* kernel, the performance is bound by memory (Figure 8B). This is in contrast to the previous observation that the convolution code without any optimization involving matrix multiplication (Figure 8C) where memory utilization was higher. We aim to improve memory access by loading the kernel/mask into constant memory to use the benefit of caching or optimize global memory accesses.

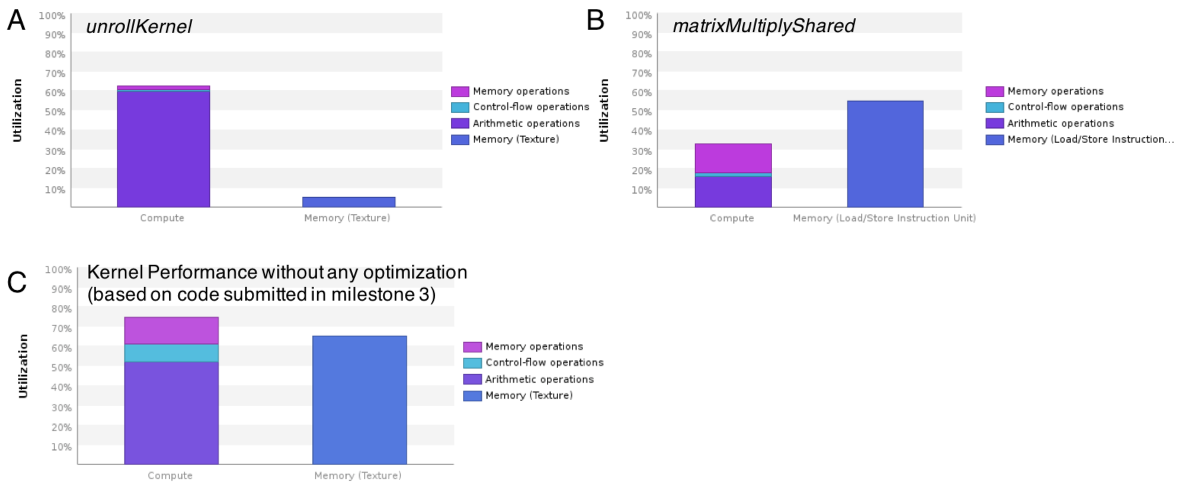


Figure 8: The performances of (A) the *unrollKernel* kernel in the optimization 1, (B) the *matrixMultiplyShared* kernel in the optimization 1 and (C) *matrixMultiplyShared* kernel in the base code.

The *matrixMultiplyShared* kernel has slight divergence, 5.3% in line 32 and 3.3% in line 51 respectively. This is expected, since both these lines check for boundary conditions in the kernel.

In conclusion, the matrix multiplication kernel takes more time than the unrolling kernel. During the first layer, each instance spends around 5 μs to unroll and about 8 μs in matrix multiplication; for the second layer, these values are about 9 μs and 33 μs, each.

4.2 Optimization 2: Weight matrix in constant memory

4.2.1 Describe the optimization

From the previous matrix multiplication code, we try to push the kernel to constant memory of size 14112. Also, we eliminates all shared memory access for tileA in matrix multiplication since constant memory access is faster.

4.2.2 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
Op Time: 0.002389
Op Time: 0.006688
Correctness: 0.85 Model: ece408

4.10user 2.39system 0:04.29elapsed 151%CPU
```

Dataset Size 1000

```
Op Time: 0.023550
Op Time: 0.063755
```

```
    Correctness: 0.827 Model: ece408

    3.99user 2.53system 0:04.13elapsed 157%CPU
```

Dataset Size 10,000

```
    Op Time: 0.216136
    Op Time: 0.567764
    Correctness: 0.8171 Model: ece408

    4.70user 2.88system 0:05.11elapsed 148%CPU
```

Our implementation of the optimization 2 has achieved the expected correctness values. However, compared to the base code, the running times for both forward pass layers on all three datasets increase by four to five times. For example, it takes 0.04 and 0.15 seconds to compute the first and the second forward pass layers on dataset 10,000 using the base code, while it takes 0.22 and 0.57 seconds using the optimized code. Compared to the optimization 1, this optimization decreases the performance even further.

4.2.3 Demonstrate nvprof profiling and analyze optimization

We believe that the performance has gone down because of the long copy and access time to constant memory becoming a dominating factor. The execution timeline is still segmented as before, because we are still looping through kernels several times.

The occupation is quite low again for this optimization. For the *matrixMultiplyShared* kernel it is 75% and 49.9% for the first and second forward pass layer calls respectively, and for the *unrollKernel* it is 79% and 84%. We observe that these values are similar to the optimization 1 where constant memory was not used. This is because we have not exploited parallelism yet.

Looking at the computation-memory tradeoff (Figure 9) memory utilization of the *unrollKernel* improves, which is what we intended. Also, the compute utilization is still same as the optimization 1. This could be the second factor for the performance drop and lack of execution time improvement, first being the long constant memory copy time.

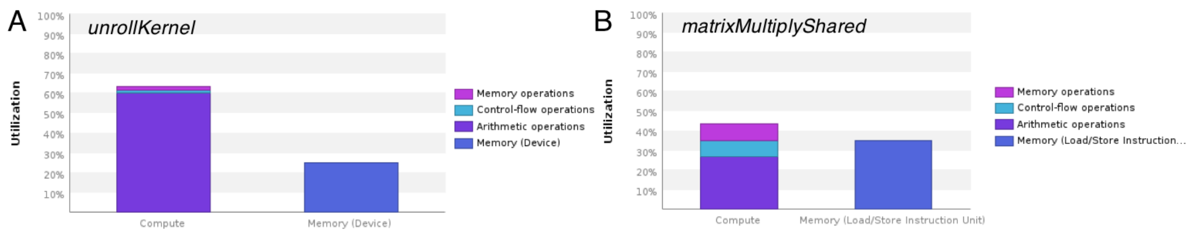


Figure 9: The performances of (A) the *unrollKernel* kernel and (B) the *matrixMultiplyShared* kernel in the optimization 2.
The divergence is 3.3%, which is better than optimization 1 because we eliminated one of the boundary condition checks at the first share memory.

As seen for the optimization 1, the *matrixMultiplyShared* kernel takes more time than the *unrollKernel* kernel. During the first layer, each instance spends around 9 μs to unroll and about

13 μs in matrix multiplication; for the second layer, these values are about 9 μs and 45 μs, each. These values are a significant increase from optimization 1 values for execution of an instance of these kernels.

In summary, although constant memory avoids loading into shared memory, it does not fix the main problem here - low parallelism. Another most important limitation until now in our code has been low utilization of all the memory resources.

4.3 Optimization 3: Kernel fusion for unrolling and matrix-multiplication

4.3.1 Describe the optimization

Here we eliminated the unrolling kernel entirely and the code consists of a single kernel.

4.3.2 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
Op Time: 0.001725
Op Time: 0.006300
Correctness: 0.85 Model: ece408

3.98user 2.42system 0:04.22elapsed 151%CPU
```

Dataset Size 1000

```
Op Time: 0.017180
Op Time: 0.063257
Correctness: 0.827 Model: ece408

4.12user 2.49system 0:04.12elapsed 160%CPU
```

Dataset Size 10,000

```
Op Time: 0.165171
Op Time: 0.566342
Correctness: 0.8171 Model: ece408

4.59user 2.71system 0:05.00elapsed 146%CPU
```

Our implementation of the optimization 3 has achieved the expected correctness values. However, compared to the base code, the running times for both forward pass layers on all three datasets increased by three to four times. For example, it took 0.04 and 0.15 seconds to compute the first and the second forward pass layers on dataset 10,000 using the base code, while it took 0.17 and 0.57 seconds using the optimized code. This suggests that combining unrolling and matrix-multiplication kernel could not improve the computing efficiency.

4.3.3 Demonstrate nvprof profiling and analyze optimization

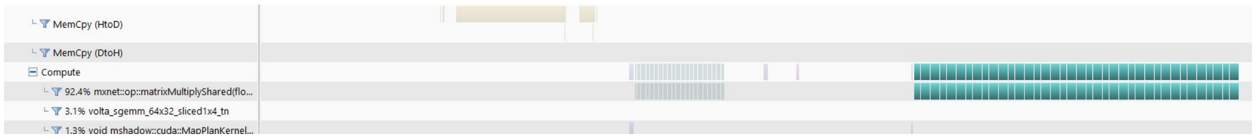

MemCpy (HtoD)
MemCpy (DtoH)
Compute
92.4% mxnet::op::matrixMultiplyShared(flo...
3.1% volta_sgemm_64x32_sliced1x4_tn
1.3% void mshadow::cuda::MapPlanKernel...

Figure 10: Kernel execution timeline on dataset size 100 based on the optimization 3. Only part of the timeline and the top few kernels are shown for clarity.

The timeline for a combined kernel is much more consolidated (Figure 10) as compared to the code with two separated kernels (Figure 7).

The $matrixMultiplyShared$ kernel takes about 4.7 μs and 55 μs to execute each instance in the first layer and the second layer call, respectively.

Interestingly, the compute versus memory graph is very similar to the optimization 2. In the kernel, memory bandwidth and utilization of the different memory types, shared memory and unified cache utilizations, have increased a lot from our previous optimizations.

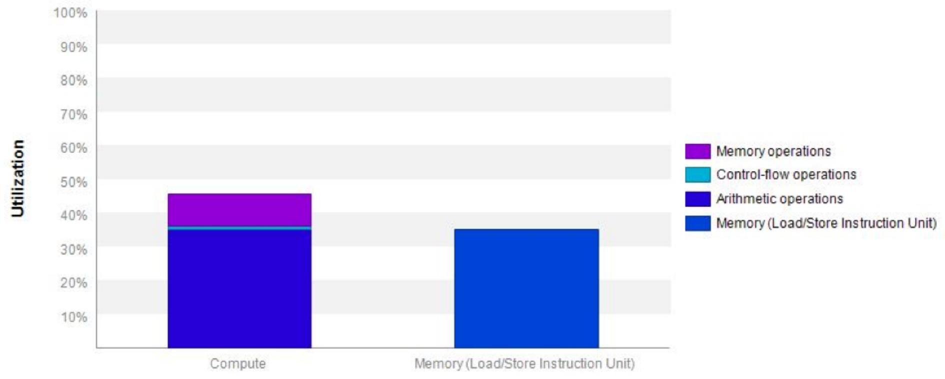


Figure 11: The performances of the $matrixMultiplyShared$ kernel in the optimization 3.

The Occupancy is 85% and 49.9%, similar to the previous 2 optimizations. We did not expect this to change with this optimization, only changing the structure of the code to employ parallelism instead of looping over images will change this value. The divergence is also smaller, only 2.1%.

With optimization 3, we see an interesting SM utilization behaviour (Figure 12), which we did not see before. Previously when the kernels were separate, many more SMs were not used at all during matrix multiplication. This chart is for the multiprocessor utilization in the second layer call. For the first forward layer call, all SMs are utilized about 55% except 2 SMs which are 80% utilized. Previously, about half SMs were utilised 50% and half were utilised 70% in the first forward layer call.

We conclude that this optimization is simple and clean, however hurts performance for all dataset sizes.

4.4 Summary of milestone 4

In summary, we have explored three optimizations in this milestone, including unrolling the input matrix, utilizing constant memory for weight matrix, and combining kernels for unrolling and matrix multiplication. In general, we are unable to observe increase in kernel performance using

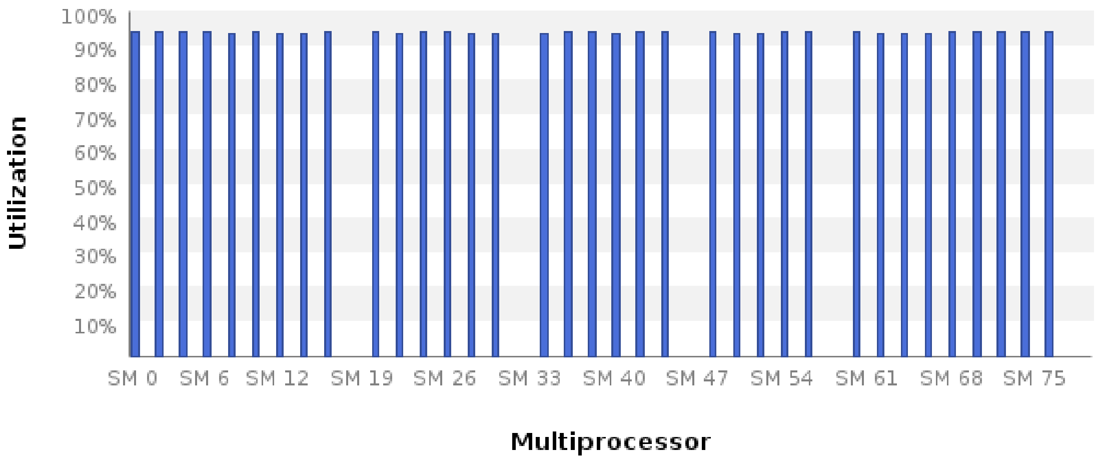


Figure 12: Multiprocessor utilization with the optimization 3.

each strategy separately, compared to the base performance. Based on our profiling results, we can explore parallelism in processing input images, input channels and output channels later. Also, we will try out other optimization methods as suggested in the list.

MileStone 3: Due November 16, 2018

3.1 Correctness and timing with 3 different dataset sizes

Dataset Size 100

```
Op Time: 0.000446
Op Time: 0.001617
Correctness: 0.85 Model: ece408

3.99user 2.37system 0:04.20elapsed 151%CPU
```

Dataset Size 1000

```
Op Time: 0.004267
Op Time: 0.016125
Correctness: 0.827 Model: ece408

4.03user 2.41system 0:03.99elapsed 161%CPU
```

Dataset Size 10,000

```
Op Time: 0.042534
Op Time: 0.152912
Correctness: 0.8171 Model: ece408

4.03user 2.84system 0:04.41elapsed 155%CPU
```

Table 1: Kernel execution properties for dataset size 1000.

Property	**1st Call**	**2nd Call**
Grid Size	[1000,12,25]	[1000,24,4]
Block Size	[16,16,1]	[16,16,1]
Global Load Efficiency	57.4%	56.9%
Global Store Efficiency	68.6%	65.9%
Warp Execution Efficiency	83.9%	81.5%
Occupancy Acheived	84.9%	90.6%

3.2 Demonstrate nvprof profiling the execution

Figure 13 shows a timeline for the execution of the code on dataset size 1,000 using NVIDIA Visual Profiler. The profiling overhead is highlighted in red.

In particular, the *forward_kernel* is called twice in the current execution. The first call takes shorter time and the the second takes longer time (for all dataset sizes). The exact start and end time can be seen in the Properties View of the profiler. For both calls to *forward_kernel*, some important properties are summarized in Table 1.

Overall, both compute and memory are likely bottlenecks to performance for the *forward_kernel*, and latency is likely not the primary performance bottleneck for this kernel.

3.2.1 Compute analysis

3.2.1.1 Low warp execution efficiency

The warp execution efficiency for the *forward_kernel* is 83.9% during the first call and 81.5% during the second call, if predicted instructions are not taken into account. The kernel's not predicted off warp execution efficiency is less than 100% due to divergent branches and predicted instructions. It is suggested to reduce the amount of intra-warp divergence and prediction in the kernel.

3.2.1.2 Divergent Branches

In the first *forward_kernel* call, the divergence rate (line 42 in new-forward.cuh) is 16.5% (396,000 divergent executions out of 2,400,000 total executions). In the second *forward_kernel* call, the divergence rate (line 42 in new-forward.cuh) is 46.9% (360,000 divergent executions out of 768,000 total executions. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

3.2.2 Memory bandwidth analysis

3.2.2.1 Inefficient global memory alignment and access pattern

It is suggested that the *forward_kernel* kernel has an efficient global memory alignment and access pattern (lines 48 and 49 in new-forward.cuh). As a result, memory bandwidth is not used most efficiently.

3.2.2.2 GPU utilization limited by memory bandwidth

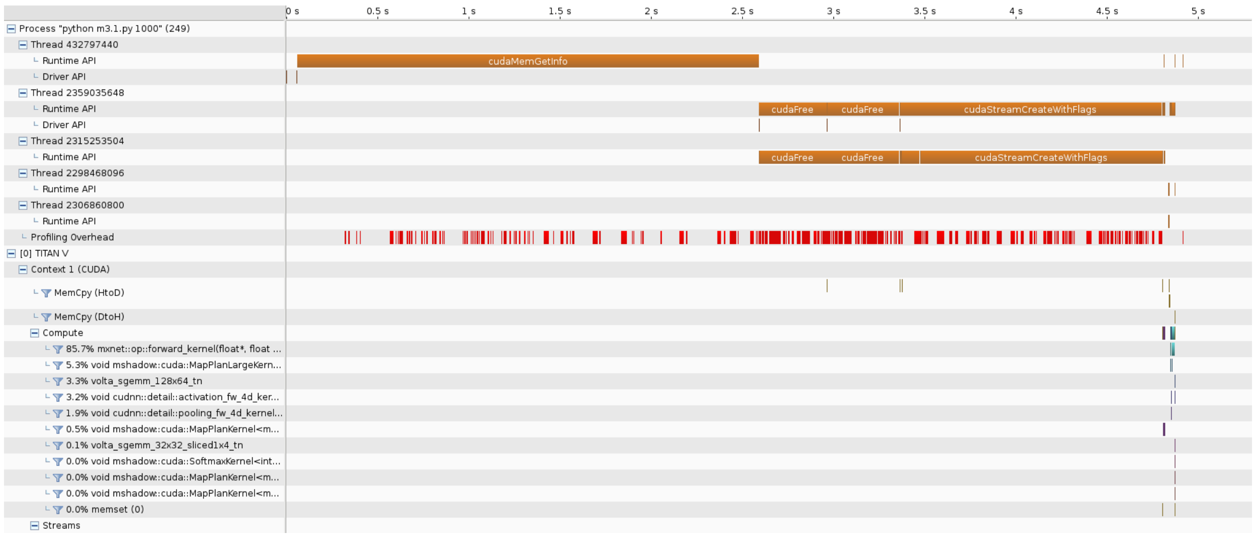

0 s
0.5 s
1 s
1.5 s
2 s
2.5 s
3 s
3.5 s
4 s
4.5 s
5 s
Process "python m3.1.py 1000" (249)
Thread 432797440
Runtime API
cudaMemGetInfo
Driver API
Thread 2359035648
Runtime API
cudaFree
cudaFree
cudaStreamCreateWithFlags
Driver API
Thread 2315253504
Runtime API
cudaFree
cudaFree
cudaStreamCreateWithFlags
Thread 2298468096
Runtime API
Thread 2306860800
Runtime API
Profiling Overhead
[0] TITAN V
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
85.7% mxnet::op::forward_kernel(float*, float ...
5.3% void mshadow::cuda::MapPlanLargeKern...
3.3% volta_sgemm_128x64_tn
3.2% void cudnn::detail::activation_fw_4d_ker...
1.9% void cudnn::detail::pooling_fw_4d_kernel...
0.5% void mshadow::cuda::MapPlanKernel<m...
0.1% volta_sgemm_32x32_sliced1x4_tn
0.0% void mshadow::cuda::SoftmaxKernel<int...
0.0% void mshadow::cuda::MapPlanKernel<m...
0.0% void mshadow::cuda::MapPlanKernel<m...
0.0% memset (0)
Streams

Figure 13: Execution timeline on dataset size 1000 as seen with NVIDIA Visual Profiler.

Figure 14 shows that the performance of the *forward_kernel* is potentially limited by the bandwidth available from one or more of the memories on the device. In particular, the bandwidth available to the unified cache that holds texture, global, and local data limits the kernel performance, highlighted in red. Accordingly, a series of optimizations are suggested for the memory with high bandwidth utilization, which will be explored in the future.

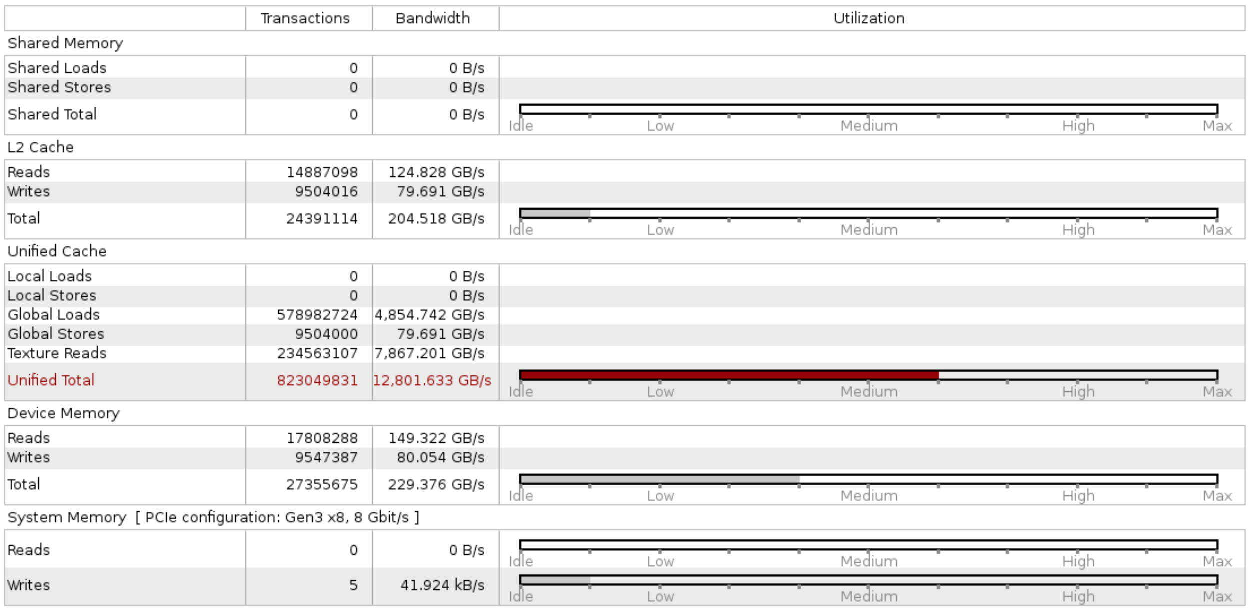
1. Shared memory: If possible use 64-bit accesses to shared memory and 8-byte bank mode to achieve 2 times throughput.
2. L2 cache: Align and block kernel data to maximize L2 cache efficiency.
3. Unified cache: Reallocate texture data to shared or global memory.
4. Device memory: Resolve alignment and access pattern issues for global loads and stores.
5. System memory: Make sure performance critical data is placed in device or shared memory.

3.2.3 Latency analysis

From the profiling analysis, the kernel's block size, register usage, and shared memory usage allow the kernel to fully utilize all warps on the GPU.

3.2.4 Comparing GPU implementation to CPU implementation

We also observed that the performance using GPU is better as compared to the CPU implementation (from Milestone 2). Previously, the execution time was 2 min 38 sec whereas, with the current GPU kernel implementation the execution time is 4.03 sec. This is a 97.5% increase. We hope to improve the kernel code with the suggested optimizations in future milestone deliverables.



	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	
L2 Cache			
Reads	14887098	124.828 GB/s	
Writes	9504016	79.691 GB/s	
Total	24391114	204.518 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	578982724	4,854.742 GB/s	
Global Stores	9504000	79.691 GB/s	
Texture Reads	234563107	7,867.201 GB/s	
Unified Total	823049831	12,801.633 GB/s	
Device Memory			
Reads	17808288	149.322 GB/s	
Writes	9547387	80.054 GB/s	
Total	27355675	229.376 GB/s	
System Memory [PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	
Writes	5	41.924 kB/s	

Figure 14: Memory bandwidth used by the *forward_kernel* for the various types of device memory.

MileStone 2: Due October 29, 2018

2.1 List whole program execution time

```
157.56user 4.42system 2:31.72elapsed 106%CPU
```

2.2 List Op times

```
Op Time: 28.737308
Op Time: 118.727850
Correctness: 0.8171 Model: ece408
```

Milestone 1: Due October 24, 2018

1.1 Include a list of all kernels that collectively consume more than 90% of the program time

Top 10 kernels are as below:

1. volta_scudnn_128x32_relu_interior_nn_v1
2. Implicit_convolve_sgemm
3. volta_sgemm_128x128_tn
4. activation_fw_4d_kernel
5. pooling_fw_4d_kernel

6. MapPlanLargeKernel
7. SoftmaxKernel
8. MapPlanKernel
9. volta_sgemm_32x32_sliced1x4_tn
10. computeOffsetsKernel

1.2 Include a list of all CUDA API calls that collectively consume more than 90% of the program time

Top 10 CUDA API calls are as below:

1. cudaStreamCreateWithFlags
2. cudaMemGetInfo
3. cudaFree
4. cudaEventCreateWithFlags
5. cudaMemcpy2DAsync
6. cudaFuncSetAttribute
7. cudaStreamSynchronize
8. cudaMalloc
9. cudaGetDeviceProperties
10. cudaMemcpy

1.3 Include an explanation of the difference between kernels and API calls

Kernels are programmer defined functions, while API calls are built-in.

1.4 Show output of rai running MXNet on the CPU

```
* Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
```

1.5 List program run time

```
19.48user 4.09system 0:13.30elapsed 177%CPU
```

1.6 Show output of rai running MXNet on the GPU

```
* Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
```

1.7 List program run time

```
4.05user 2.67system 0:04.63elapsed 145%CPU
```