ece408 Final Project

Andrew Smith, Ann Thomas, Nick Cebry

December 17^{th} 2019

1 Info

Table 1: Team Information

Name	Andrew Smith	Ann Thomas	Nick Cebry
NetID	atsmith3	ahthoma2	ncebry2
RAI ID	5d97b20e88a5ec28f9cb94e0	5d97b2190fb81f0e26d4f412	5d97b1a788a5ec28f9cb9421
Team Names	little_computer_parallel		
School Affiliation	UIUC Campus		

2 Results (Milestone 2)

2.1 Top Kernels by Execution Time

1. `volta_scudnn_128x64_relu_interior_nn_v1`
2. `volta_sgemm_128x128_tn`
3. `volta_gcgemm_64x32_nt`
4. `void cudnn::detail::pooling_fw_4d_kernel`
5. `void op_generic_tensor_kernel`
6. `void fft2d_c2r_32x32`
7. `void fft2d_r2c_32x32`
8. `void mshadow::cuda::MapPlanLargeKernel`
9. `void mshadow::cuda::SoftmaxKernel`

2.2 Top API Calls by Execution Time

1. `cudaStreamCreateWithFlags`
2. `cudaMemGetInfo`
3. `cudaFree`
4. `cudaMemcpy2DAsync`
5. `cudaStreamSynchronize`
6. `cudaEventCreate`
7. `cudaMemcpy`
8. `cudaHostAlloc`
9. `cuDeviceGetName`

2.3 Kernel and API difference

The difference between an API call and a Kernel is that an Kenerls are launched on the GPU and execute on the Stream Multiprocessors (SMs). API calls are something like cudaMemcpy(...) where it interfaces with the cuda libraries and even the hardware but doesn't directly run work on the SMs.

2.4 MXNet CPU

2.4.1 Output

```
Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
16.97user 4.74system 0:08.92elapsed 243%CPU (0avgtext+0avgdata 6044568maxres
ident)k
0inputs+2616outputs (0major+1601016minor)pagefaults 0swaps
```

2.4.2 Run Time

According to the output of the CPU run above it took `0:08.92` seconds of system time to complete with an accuracy of `0.8154`.

2.5 MXNet GPU

2.5.1 Output

```
Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
5.04user 3.22system 0:04.73elapsed 174%CPU (0avgtext+0avgdata 2954432maxresident)k
0inputs+4536outputs (0major+732074minor)pagefaults 0swaps
```

2.5.2 Run Time

According to the output of the GPU run above it took `0:04.73` seconds of system time to complete with an accuracy of `0.8154`.

2.6 CPU Implementation

See `ece408_src/new_forward.h` for the implementation.

2.6.1 Output

```
Running /usr/bin/time python m2.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.107684
Op Time: 0.587626
Correctness: 0.76 Model: ece408
4.55user 2.90system 0:01.86elapsed 399%CPU (0avgtext+0avgdata 309340maxresident)k
0inputs+2824outputs (0major+111706minor)pagefaults 0swaps

Running /usr/bin/time python m2.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 1.054678
Op Time: 5.946515
Correctness: 0.767 Model: ece408
12.07user 3.23system 0:08.43elapsed 181%CPU (0avgtext+0avgdata 829884maxresident)k
0inputs+0outputs (0major+305347minor)pagefaults 0swaps

Running /usr/bin/time python m2.1.py 10000
Loading fashion-mnist data... done
Loading model... done
```

```
New Inference
Op Time: 10.827225
Op Time: 62.795590
Correctness: 0.7653 Model: ece408
89.85user 11.82system 1:17.98elapsed 130%CPU (0avgtext+0avgdata 6043320maxresident)k
0inputs+0outputs (0major+2299977minor)pagefaults 0swaps
```

2.6.2 Program Execution Time

Table 2: Program Execution Time

Batch Size (Images)	User Time	System Time	Elapsed Time
100	4.55	2.90	1.86
1000	12.07	3.23	8.43
10000	89.85	11.82	17.98

2.6.3 Operation Times

Table 3: Operation Execution Time

Batch Size (Images)	Operation 1	Operation 2
100	0.107684	0.587626
1000	1.054678	5.946515
10000	10.827225	62.795590

3 Milestone 3

3.1 Accuracy

Table 4: Accuracy

Batch Size (Images)	Accuracy
100	0.76
1000	0.767
10000	0.7653

3.2 Operation Times

Table 5: Operation Execution Time

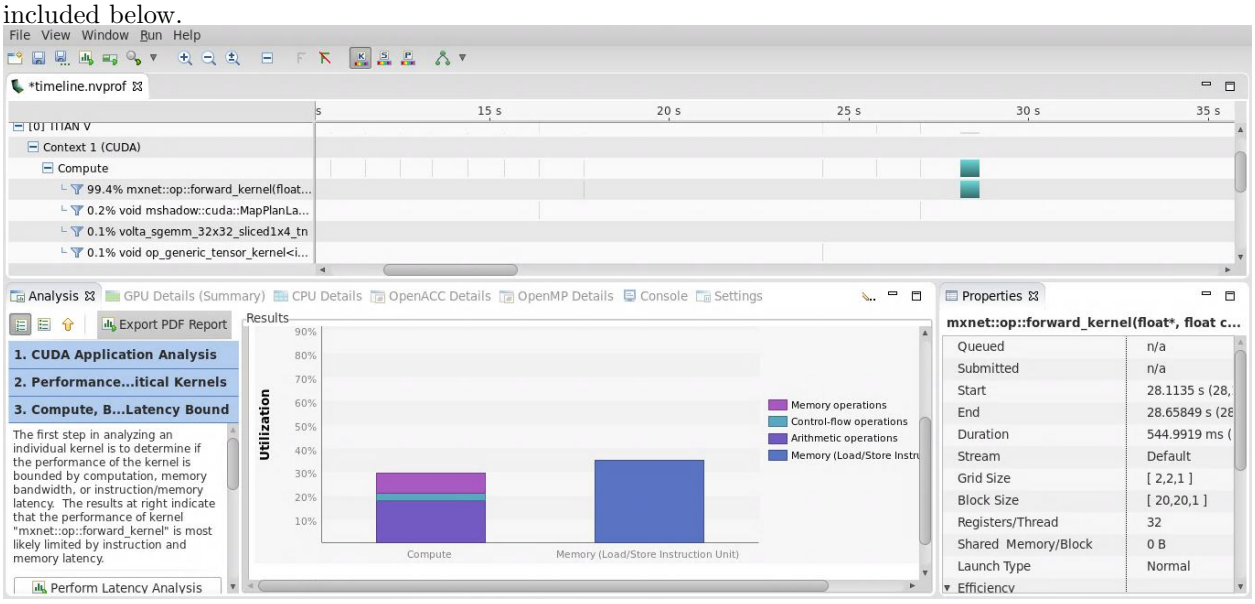
Batch Size (Images)	Operation 1	Operation 2
100	0.002426	0.060505
1000	0.023602	0.561961
10000	0.221924	5.527549

Table 6: Program Execution Time

Batch Size (Images)	User Time	System Time	Elapsed Time
100	4.86	2.88	4.79
1000	5.12	2.48	5.08
10000	9.06	4.31	10.23

3.3 NVPROF (1000)

A screenshot of the NVVP program showing the execution times of the kernel for batch size of 1000 is included below.



4 Milestone 4

4.1 Overview

For milestone 4 we had to take our baseline GPU implementation from the previous milestone and add three optimizations to the convolution kernel. The three optimizations we decided to implement were 1. *loop unrolling* 2. *constant memory* and 3, *shared memory*. In each of the following sections we will describe the optimization's implementation details and provide and analysis of the performance impact with respect to the baseline.

4.2 Optimization: Loop Unroll

Loop unrolling is reducing the number of iterations of the loop and then doing more sequential operations per loop body. This improves the threads throughput. We applied the loop unrolling to the two inner loops of the convolution which do the multiplication and accumulation of the kernel w with the input tensor x. We found that this optimization was provided the greatest increase in performance.

4.2.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.001475	0.004636	0.76
1000	0.016734	0.048412	0.767
10000	0.153228	0.456949	0.7653

4.3 Optimization: Storing Kernel in Constant Memory

We chose to store the kernel in constant memory. Constant memory has an access time of 5 cycles, whereas global memory has an access time of 500 cycles. Although loading into constant memory has an associated overhead time, it is still less than the 100x increase we get from using constant memory. We used constant memory over shared memory because it is read-only, and we never want to write to the kernel.

4.3.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.002029	0.005533	0.76
1000	0.020372	0.055014	0.767
10000	0.203387	0.500582	0.7653

4.4 Optimization: Shared Memory Input Tensor

We used shared memory on the input tensor as an optimization. We loaded the input tensor into shared memory from global memory. That way, the thread block would only have to perform a shared memory access instead of a global memory access. The access to shared memory is about 100x faster than the access to global memory.

4.4.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.001876	0.005113	0.76
1000	0.018391	0.050340	0.767
10000	0.177534	0.456385	0.7653

4.5 All 3 Optimizations

Finally, we combined the loop unrolling, shared memory and constant memory optimizations into one kernel. Combining the three optimizations resulted in the fastest op1 and op2 times and was significantly faster than the baseline.

4.5.1 Results

Batch Size	Operation 1	Operation 2	Accuracy
100	0.001353	0.003711	0.76
1000	0.013015	0.036563	0.767
10000	0.129487	0.331177	0.7653

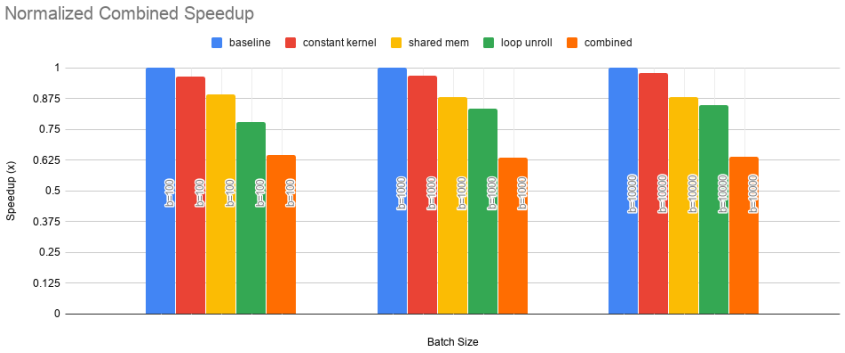


Figure 1: Normalized Speedup for Individual and Combined Optimizations

4.5.2 NVVP Milestone 4

All of the following NVVP figures were generated with all 3 of the milestone 4 optimizations enabled. The Forward pass layer was run with a batch size of 10000 on a Nvidia Titan V GPU. For the final set of optimizations there are some things we need to focus on improving. We have mediocre Load-Store Efficiency and mediocre Warp Execution Efficiency (Figure 2). After analyzing the memory utilization we see that a majority of our bandwidth goes to the shared memory (Figure 4) verifying our Shared-Memory optimization. Anoother area to be addressed is there is a vast majority of stalls due to Synchronization. If we can figure out how to relax the synchronization requirement between threads, then we could dramatically improve the execution time (Figures 6, 7).

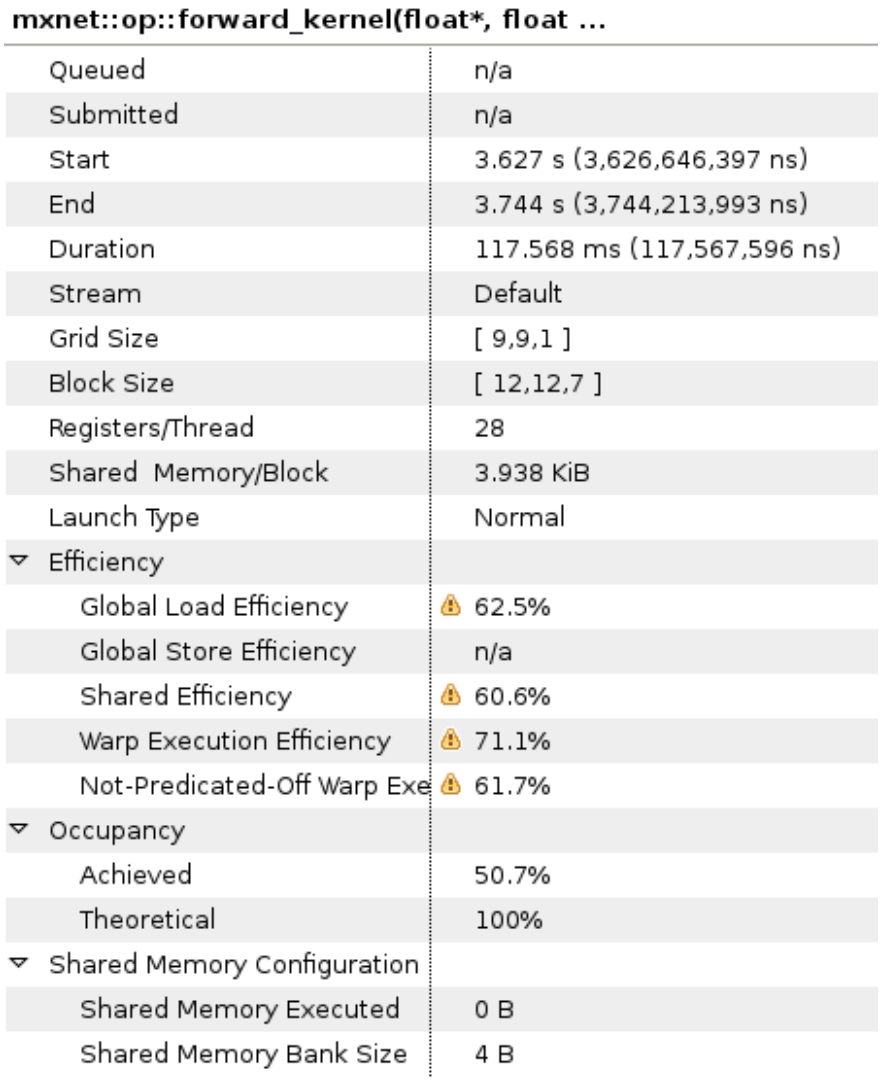


Figure 2: NVVP Overall Stats

Instruction Execution Counts

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.

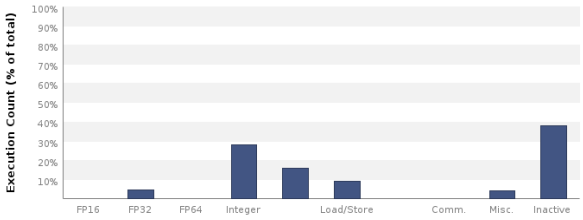


Figure 3: Instruction Execution Counts

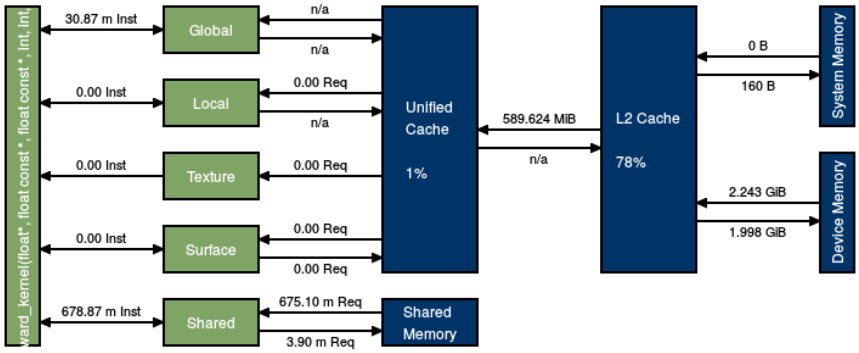


Figure 4: Memory Stats

Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [9,9,1] (81 blocks)Block Size: [12,12,7] (1008 threads)
Occupancy Per SM				
Active Blocks		2	32	
Active Warps	32.43	64	64	
Active Threads		2048	2048	
Occupancy	50.7%	100%	100%	
Warps				
Threads/Block		1008	1024	
Warps/Block		32	32	
Block Limit		2	32	
Registers				
Registers/Thread		28	65536	
Registers/Block		32768	65536	
Block Limit		2	32	
Shared Memory				
Shared Memory/Block		4032	98304	
Block Limit		24	32	

Figure 5: Occupancy

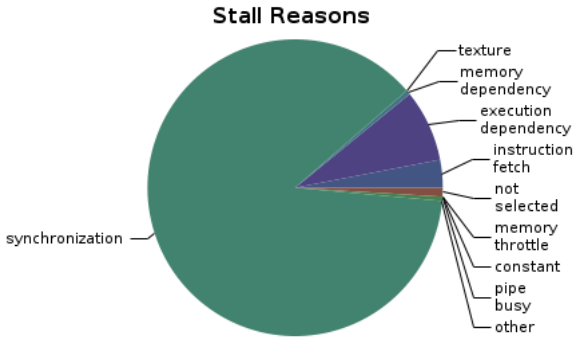


Figure 6: Stall Reasons

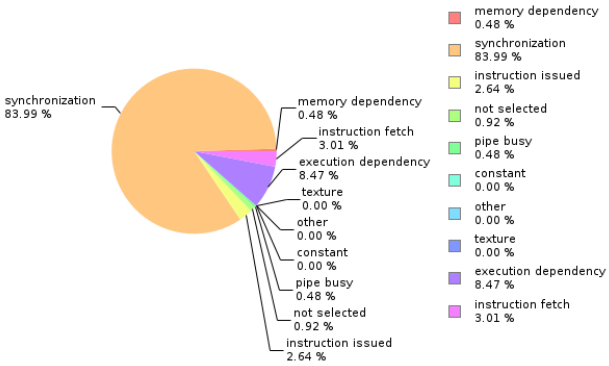


Figure 7: PC Distribution

Line / File	new-forward.cuh - /mxnet/src/operator/custom
38	Divergence = 4.4% [1150000 divergent executions out of 25920000 total executions]
43	Divergence = 8.7% [27000000 divergent executions out of 311040000 total executions]

Figure 8: Divergence

5 Milestone 5

5.1 Overview

For the final milestone, we continued to add optimizations to our baseline GPU implementation from milestone 3 in addition to the optimizations from milestone 4. We added optimizations for 1. *separate kernels*, 2. *tuning with restrict and loop unrolling*, 3. *thread coarsening*, and 4.*parameter optimization*. In each of the following sections we will describe the optimization's implementation details and provide and analysis of the performance impact with respect to the previous version.

5.2 Optimization: Convolution as a Matrix Multiply

Our first optimization was to re-interpret the convolution as a matrix multiply. Instead of unrolling the 3D input images into a matrix and re-rolling the output matrix into 3D images, the matrix multiplication simply calculates the appropriate indices when loading into shared memory and storing the output.

Table 7: Experimental Results of Implementing Matrix Multiply and Shared Memory (MM, SM)

	Op 1 (s)	Op 2 (s)	Combined (s)
Average μ	0.036530	0.058041	0.094571
Standard Deviation σ	0.001708	0.002113	0.003772
max	0.039776	0.063269	0.103036
min	0.035705	0.057066	0.092789

5.3 Optimization: Separate Kernels

After examining the performance of the two different layers we found that the unrolling as a matrix multiply kernel performed poorly during the execution of Op 1 but that our traditional convolution kernel from milestone 4 performed better for Op 1. We expect this is due to the single input channel in Op 1. Because there is only a single input channel, our atomic add at the end of our convolution kernel should never have to stall, because there are no other input channels affecting the given output map. So we decided to launch the traditional convolution kernel for Op 1 and then launch the matrix multiplication kernel for Op 2. This reduced the execution time of the Op 1 by a substantial amount. This particular optimization is referred to as *MM, SM, CD, CONV_1* in figures 10 and 11. Below is a table of execution times for this particular set of optimizations over 10 trials.

Table 8: Experimental results of implementing Matrix Multiply, Shared Memory, Compiler Directives, and Conv Op 1 (MM, SM, CD, CONV1)

	Op 1 (s)	Op 2 (s)	Combined (s)
Average μ	0.021145	0.058277	0.079422
Standard Deviation σ	0.000998	0.002694	0.003692
max	0.023117	0.063563	0.08668
min	0.020665	0.056966	0.077632

5.4 Optimization: Tuning with Restrict and Loop Unrolling

Similar to milestone 4 the the matrix multiply has loops within the body of the kernel function. These loops can be easily optimized with the use of the `#pragma unroll` compiler directive. We added these pragmas to the inner loop that calculates the partial sum of an output element. We also took advantage of the `__restrict__` keyword. This is a keyword you can attach to pointers in the function interface. This tells the compiler that there will be no aliasing of the memory regions of the input and output data. This makes the compilers optimization job easier and it yielded a substantial performance improvement.

Table 9: Experimental Results of Implementing Matrix Multiply, Shared Memory, and Compiler Directives (MM, SM, CD)

	Op 1 (s)	Op 2 (s)	Combined (s)
Average μ	0.036593	0.053636	0.090229
Standard Deviation σ	0.001669	0.008459	0.008245
max	0.039764	0.060664	0.008245
min	0.035714	0.03587	0.07174

5.5 Optimization: Constant Kernel

In milestone 4 we found that storing the weight matrix kernel into the device's constant memory section improved the GPU kernel's execution time. We thought that because it gave a performance improvement to the traditional convolution that it could benefit the matrix multiply kernel as well. We thought that instead of having to load the blocks of the kernel and the input from global memory, that if the kernel was kept in faster constant memory then the load from constant to shared would be faster than the load from global to shared. However, we found that the run time of the op 2 matrix multiply was dramatically worse than before adding the optimization. We are not exactly sure why this is the case however it might be due to caching of the global memory accesses. Alternatively, the time needed to copy the weights from global to constant memory may have outweighed any speedup gained by having the weights in constant memory.

Table 10: Experimental Results of Implementing Matrix Multiply, Shared Memory, Compiler Directives, and Constant Memory Kernel (MM, SM, CD, CK)

	Op 1 (s)	Op 2 (s)	Combined (s)
Average μ	0.054595	0.115396	0.169991
Standard Deviation σ	0.000016	0.003229	0.003230
max	0.054634	0.119383	0.173968
min	0.05458	0.108125	0.162705

5.6 Optimization: Thread Coarsening

Thread Coarsening is the practice of giving each thread more work to do by processing a larger block of input and output block per thread. Thread Coarsening allows for better reuse of of memory and calculations. It reduces the am mount of redundant memory accesses and redundant work done by the thread block. Each coarsened thread performs the work of several uncoarsened threads, suffering a slight serialization penalty. But this is outweighed by the increase in amount of inputs covered by a single block (larger shared array per block), and ability for a coarsened thread to store some intermediate results in its registers, which are even faster to access than shared memory. We found that both the Matrix Multiply in Op2 and the Convolution in Op 1 benefited from thread coarsening. The thread coarsening almost doubled the speedup from the kernels without coarsening. Ultimately thread coarsening along with the other other optimizations allowed us to attain a speedup of 11x over the implementation in milestone 4.

Table 11: Experimental results of implementing Matrix Multiply, Shared Memory, Compiler Directives, Conv Op 1, Thread Coarsening and Parameter Tuning (MM, SM, CD, CONV1, PT)

	Op 1 (s)	Op 2 (s)	Combined (s)
Average μ	0.011925	0.029756	0.041680
Standard Deviation σ	0.000029	0.001348	0.001373
max	0.011996	0.032316	0.044312
min	0.011898	0.029067	0.040966

5.7 Optimization: Parameter Tuning

After adding the thread coarsening our forward pass implementation now has 4 parameters we can tune to minimize the execution time of each operation. The parameters are Coarsening Factor (CF) for Op 1,

Block Size (BS) for Op 1, Coarsening Factor for Op 2, and Block Size for Op 2. This was a straightforward process. We swept the parameter values and searched for the the ones that yielded the fastest execution time for that operation. Because we are limited by a maximum number of threads and maximum amount of shared memory per block, the range for the parameters is relatively limited. We settled on the parameter values of $CF_1 = 2$, $BS_1 = 16$, $CF_2 = 4$, $BS_2 = 8$. Some of the other parameter values we explored are shown below in figure 9.

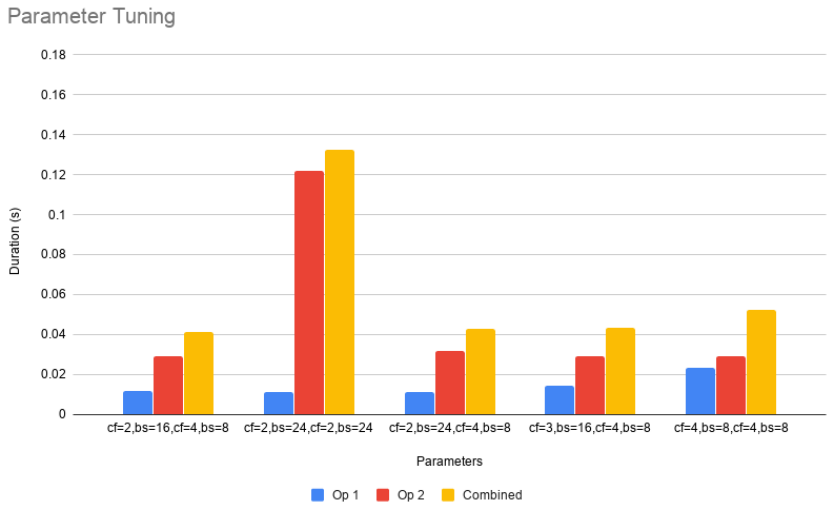


Figure 9: Execution time of operations with different block size (BS) and thread coarsening (CF) parameters

5.8 Results

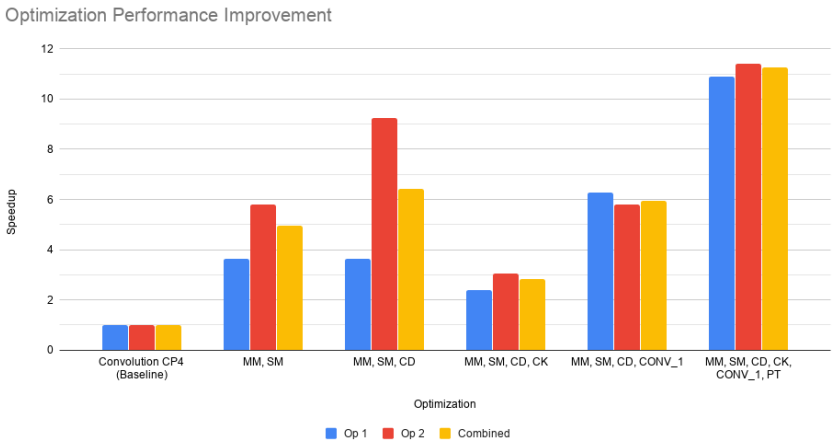


Figure 10: Normalized Speedup for milestone 5 optimizations

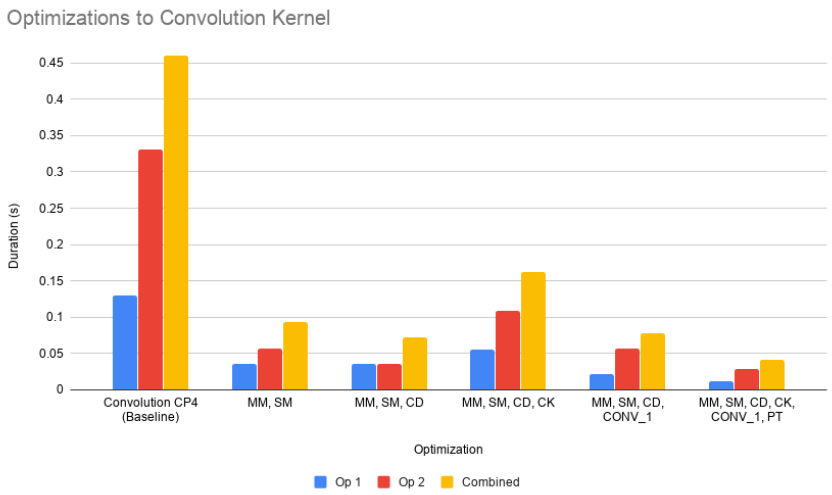


Figure 11: Execution time of operations for milestone 5 optimizations

5.9 NVPROF & NVVP Analysis of the Fully Optimized Kernel

For the standard convolution kernel we use for Op 1, we find that performance is memory bound.

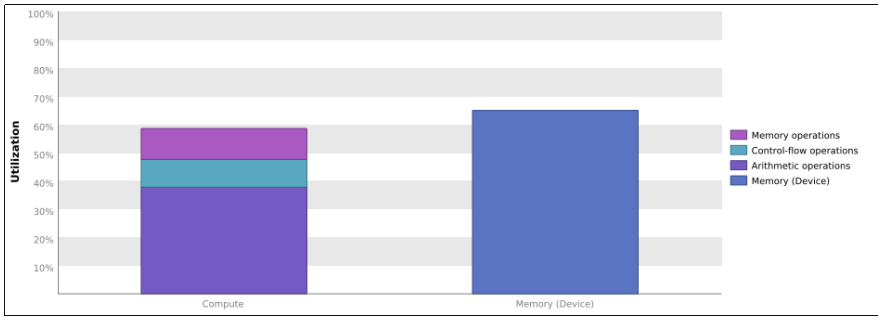
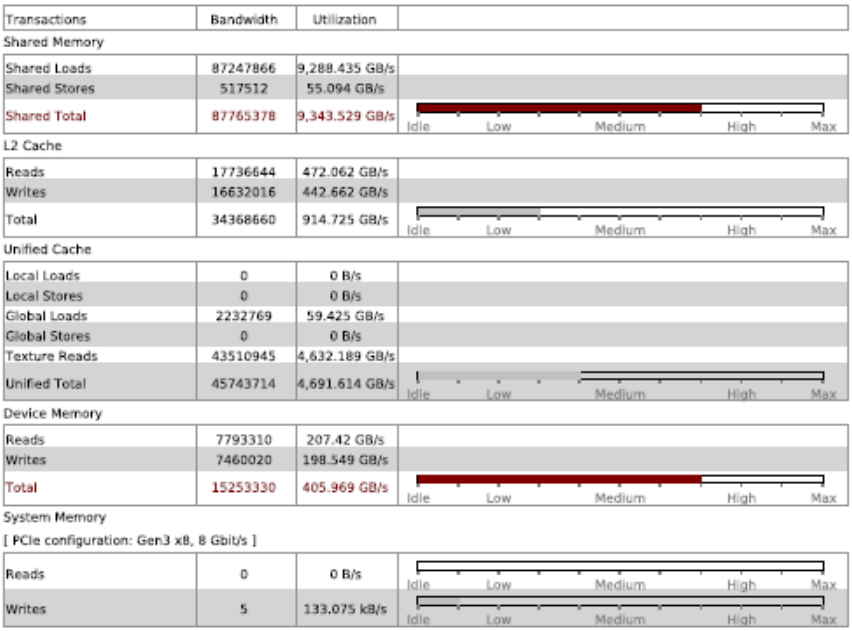


Figure 12: Memory and compute utilization for Op1

For this kernel, both shared and global memory bandwidth are both high.



Transactions	Bandwidth	Utilization	
Shared Memory			
Shared Loads	87247866	9,288.435 GB/s	
Shared Stores	517512	55.094 GB/s	
Shared Total	87765378	9,343.529 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	17736644	472.062 GB/s	
Writes	16632016	442.662 GB/s	
Total	34368660	914.725 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	2232769	59.425 GB/s	
Global Stores	0	0 B/s	
Texture Reads	43510945	4,632.189 GB/s	
Unified Total	45743714	4,691.614 GB/s	Idle Low Medium High Max
Device Memory			
Reads	7793310	207.42 GB/s	
Writes	7460020	198.549 GB/s	
Total	15253330	405.969 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	133.075 kB/s	Idle Low Medium High Max

Figure 13: Memory utilization for Op1

For the matrix multiply kernel used in Op 2, we see that the performance is also likely memory bound, but possibly also limited by compute.

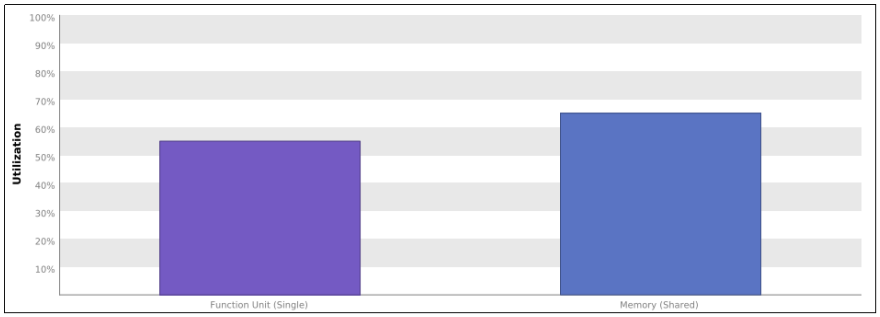


Figure 14: Memory and compute utilization for Op2

The memory bandwidth utilization for shared memory is particularly high. This is probably why thread coarsening was effective. The uncoarsened threads were probably having their memory accesses serialized due to bandwidth limitations, so coarsening likely did not come with a large memory bandwidth penalty.

Transactions	Bandwidth	Utilization	
Shared Memory			
Shared Loads	140759189	6,146.455 GB/s	
Shared Stores	71072477	3,103.483 GB/s	
Shared Total	211831666	9,249.938 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	41110086	448.783 GB/s	
Writes	11388016	124.319 GB/s	
Total	52498102	573.101 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	269540486	2,942.469 GB/s	
Global Stores	11388000	124.318 GB/s	
Texture Reads	154015490	6,725.31 GB/s	
Unified Total	434943976	9,792.098 GB/s	Idle Low Medium High Max
Device Memory			
Reads	1636423	17.864 GB/s	
Writes	2532071	27.642 GB/s	
Total	4168494	45.506 GB/s	Idle Low Medium High Max
System Memory			
[PCIe configuration: Gen3 x8, 8 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	54.583 kB/s	Idle Low Medium High Max

Figure 15: Memory bandwidth utilization for Op2