Team Name: Fast
ECE408 Final Project Report: Final Submission

Jiming Chen (jimingc2)
Jiangyan Feng (jf8)
Maxim Korolkov (korolko2)
University of Illinois at Urbana-Champaign
December 17, 2019

1 Final Milestone

1.1 Optimization 4

1.1.1 Rationale

For our fourth optimization, we combined the unroll and matrix mulitply kernels into a single kernel by performing unrolling when loading tiles of the input feature matrices into shared memory. This implementation also performs matrix multiplications for each image in the layer in parallel rather than iterating over batch size. Thus, we hypothesized that our new matrix multiplication kernel would be a significant improvement over our previous implementation since the previous implementation was primarily latency bound.

1.1.2 Correctness and Timing

- Dataset 1, N=100:

```
Op Time: 0.000464
Op Time: 0.000716
Correctness: 0.76
```

- Dataset 2, N=1000:

```
Op Time: 0.004265
Op Time: 0.006836
Correctness: 0.767
```

- Dataset 3, N=10000:

```
Op Time: 0.042196
Op Time: 0.063959
Correctness: 0.7653
```

1.1.3 NVVP Analysis

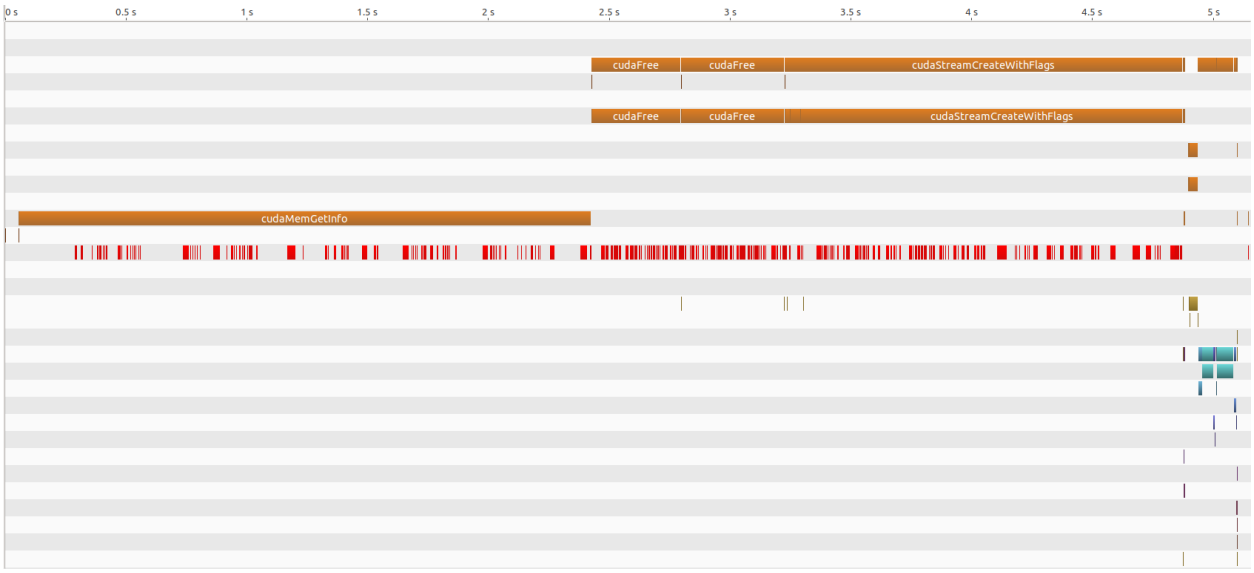

Figure 1: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

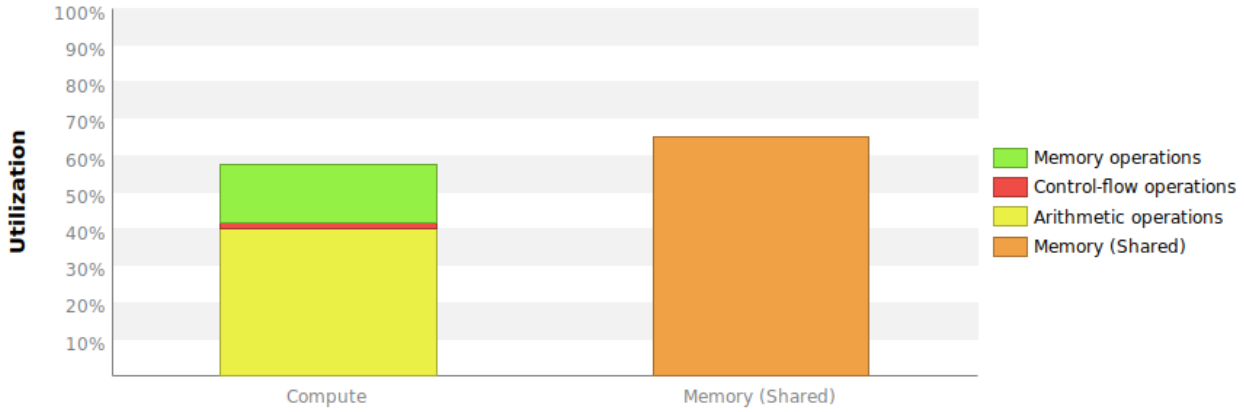

Figure 2: Compute and memory utilization.

1.1.4 Discussion

Based on our NVVP results, this kernel is now bound by memory bandwidth rather than latency, which indicates it uses available compute resources more efficiently than the previous implementation. In addition, this implementation eliminates the unroll kernel and instead performs unrolling while loading into shared memory. This also helps reduce memory bandwidth. Using a separate kernel to unroll the input features requires writing to another global memory variable, x_unroll, and then reading from the x_unroll variable during matrix multiplication. Unrolling while loading the input matrix into shared memory

eliminates these global memory reads and writes. According to NVVP, the kernel performance is still bound by shared memory bandwidth, so further optimizations will be the most effective if they focus on reducing memory bandwidth.

1.2 Optimization 5

1.2.1 Rationale

For our next optimization, we reversed the first and third dimensions of our block grid such that the batch size is the third dimension rather than the first. We hypothesized that this would improve performance by allowing for coalesced access to the input matrices and thus reduce the amount of time required to load tiles of the inputs from global to shared memory.

1.2.2 Correctness and Timing

- Dataset 1, N=100:

```
Op Time: 0.000464
Op Time: 0.000714
Correctness: 0.76
```

- Dataset 2, N=1000:

```
Op Time: 0.004334
Op Time: 0.006695
Correctness: 0.767
```

- Dataset 3, N=10000:

```
Op Time: 0.043072
Op Time: 0.066534
Correctness: 0.7653
```

1.2.3 NVVP Analysis

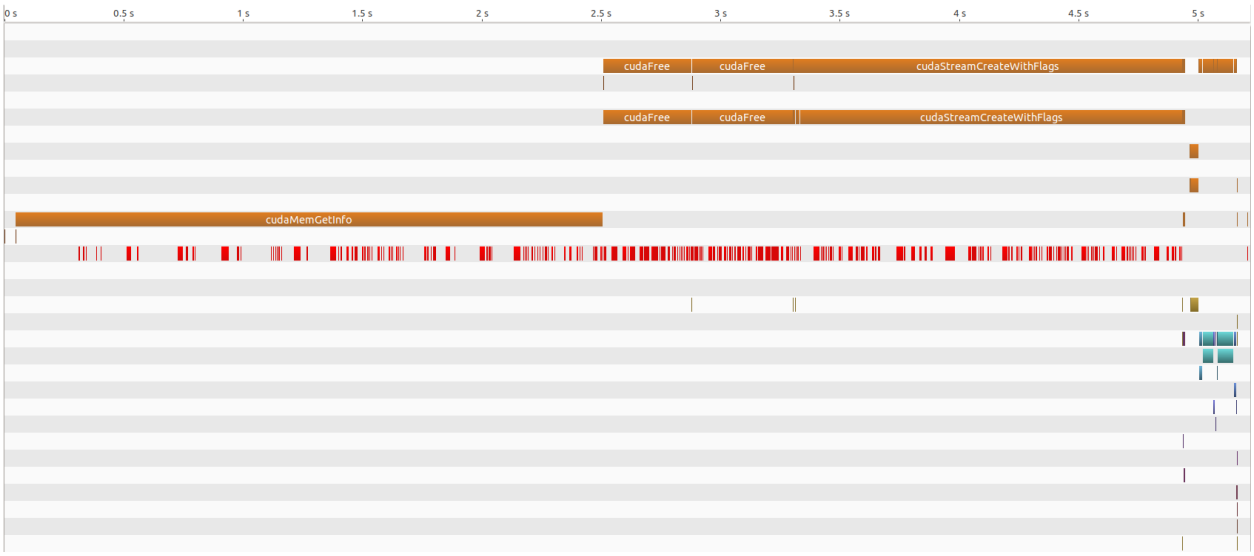

Figure 3: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

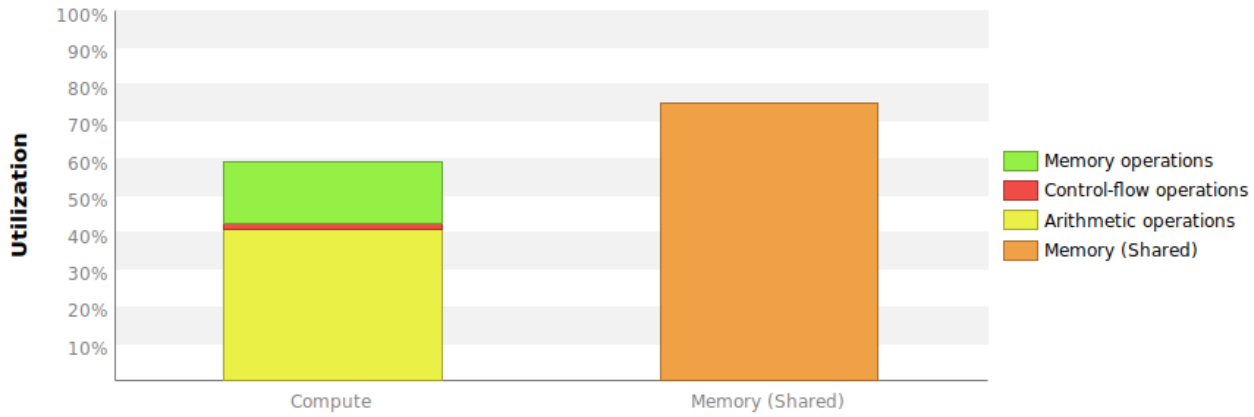

Figure 4: Compute and memory utilization.

1.2.4 Discussion

Based on the resulting Op times and NVVP analysis, transposing the x- and z- dimensions of the input had little effect on the performance. NVVP indicates that the kernel is still bound by memory bandwidth, specifically shared memory bandwidth. This, transposing input dimensions to allow for coalesced access likely did not lead to a significant speedup since the primary bottleneck is reading and writing shared memory data.

1.3 Optimization 6

1.3.1 Rationale

For optimization 6, we unrolled the for-loop in the kernel using restrict and pragma unroll. We hypothesized that this would improve our timing by parallelizing compute tasks that were previously done serially, which would increase compute efficiency.

1.3.2 Correctness and Timing

- Dataset 1, N=100:

```
Op Time: 0.000496
Op Time: 0.000723
Correctness: 0.76
```

- Dataset 2, N=1000:

```
Op Time: 0.004374
Op Time: 0.006733
Correctness: 0.767
```

- Dataset 3, N=10000:

```
Op Time: 0.043064
Op Time: 0.060709
Correctness: 0.7653
```

1.3.3 NVVP Analysis

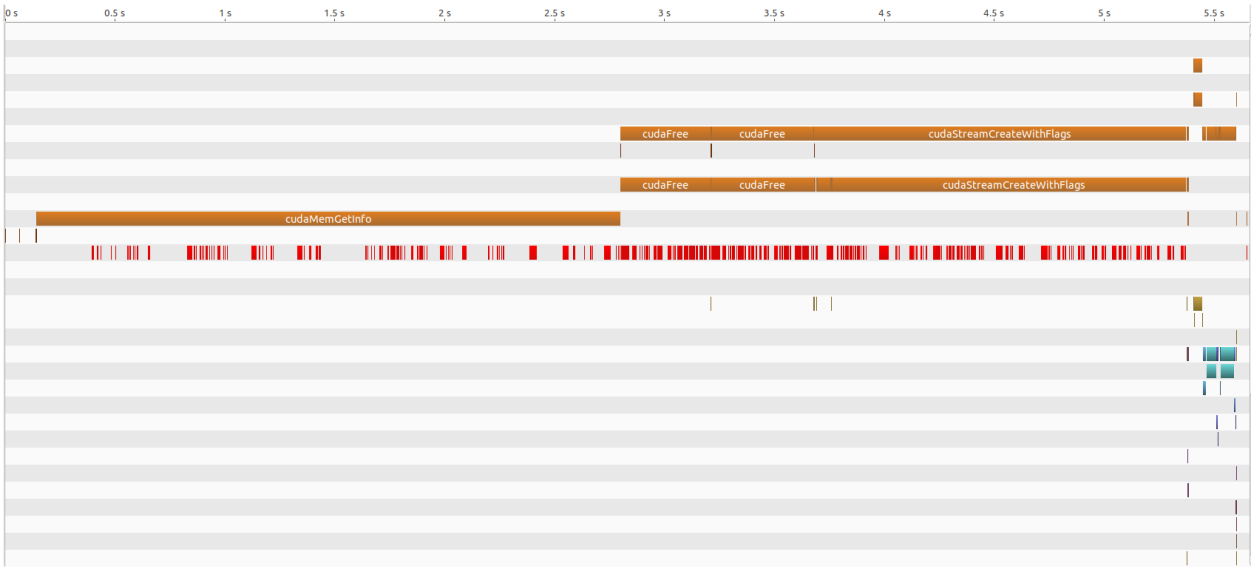

Figure 5: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

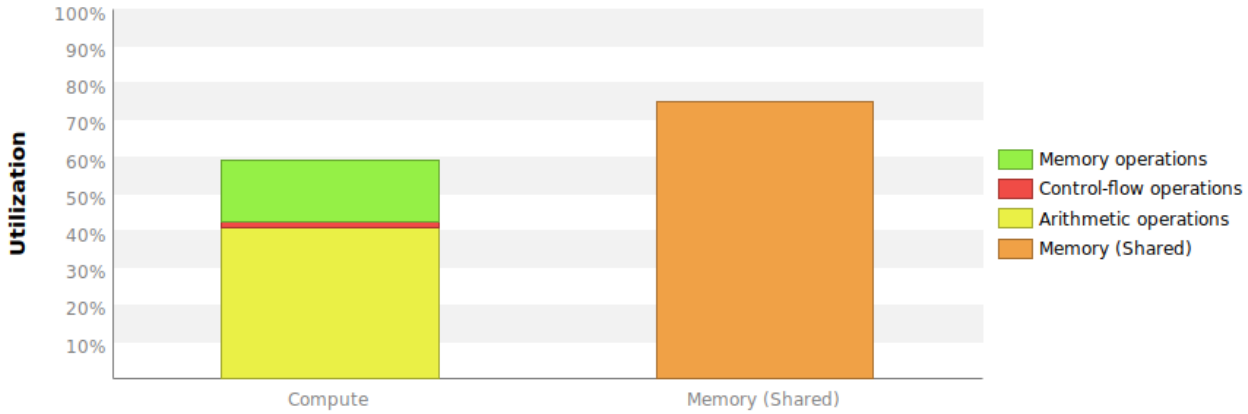

Figure 6: Compute and memory utilization for Dataset 3, N=10000.

1.3.4 Discussion

Our analysis of this kernel indicates that with pragma unroll, while the Op time decreases slightly, the performance is still memory bound, so further optimizations should primarily focus on decreasing memory bandwidth. The number of iterations of this loop is inversely proportional to the tile width. Since the tile width was larger than both input matrix dimensions for the first pass and one of the matrix dimensions for the second pass, there were either only one or a very small number of iterations of this loop, which explains the low performance impact. Additionally, the primary limitation is still memory bandwidth, so improving compute efficiency would not be expected to have a significant impact on overall performance.

1.4 Optimization 7

1.4.1 Rationale

Another optimization we tried was varying tile widths for tiled matrix multiplication. Using block sizes 8, 24, and 32, found that a tile width of 24 gave us the optimal overall performance, with the majority of the improvement coming from the second layer.

1.4.2 Correctness and Timing, Tile Width = 8

- Dataset 1, N=100:

```
Op Time: 0.000360
Op Time: 0.000885
Correctness: 0.76
```

- Dataset 2, N=1000:

```
Op Time: 0.003278
Op Time: 0.008451
Correctness: 0.767
```

- Dataset 3, N=10000:

```
Op Time: 0.032318
Op Time: 0.083950
Correctness: 0.7653
```

1.4.3 Correctness and Timing, Tile Width = 24

- Dataset 1, N=100:

```
Op Time: 0.000250
Op Time: 0.000554
Correctness: 0.76
```

- Dataset 2, N=1000:

```
Op Time: 0.002299
Op Time: 0.005244
Correctness: 0.767
```

- Dataset 3, N=10000:

```
Op Time: 0.022693
Op Time: 0.052049
Correctness: 0.7653
```

1.4.4 Correctness and Timing, Tile Width = 32

- Dataset 1, N=100:

```
Op Time: 0.000464
Op Time: 0.000714
Correctness: 0.76
```

- Dataset 2, N=1000:

```
Op Time: 0.004334
Op Time: 0.006695
Correctness: 0.767
```

- Dataset 3, N=10000:

```
Op Time: 0.043072
Op Time: 0.066534
Correctness: 0.7653
```

1.4.5 NVVP Analysis

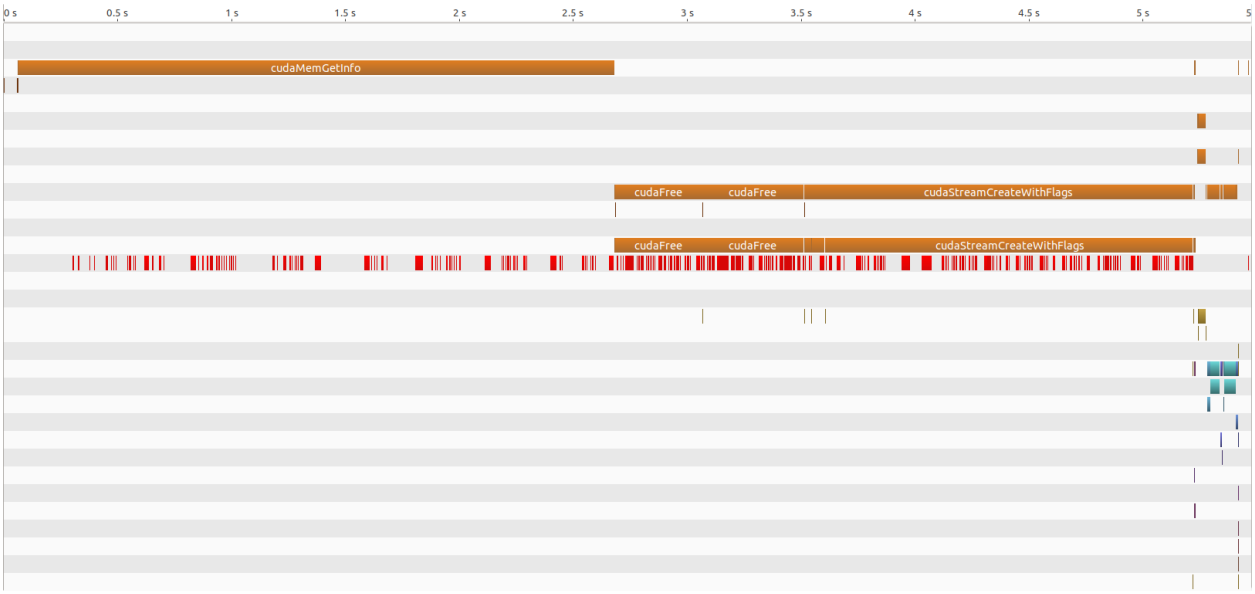

Figure 7: Representative program execution timeline for Dataset 3, N=10000, tile width 24. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

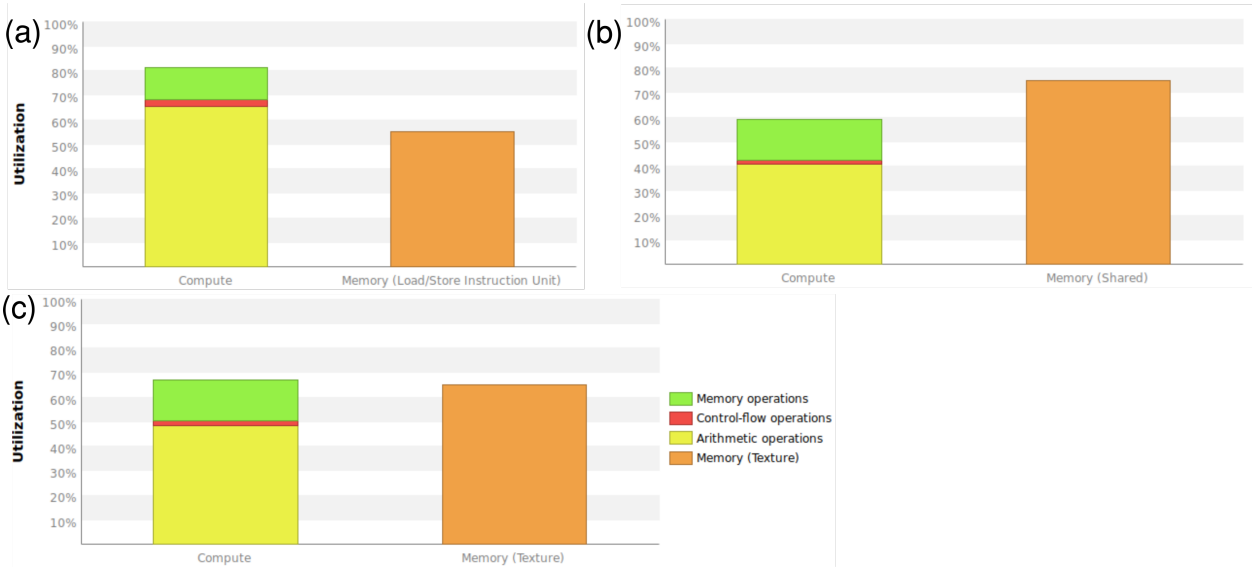

Figure 8: Compute and memory utilization for Dataset 3, N=10000 using tile widths (a) 8, (b) 24, and (c) 32.

1.4.6 Discussion

Decreasing our tile width from 32 to 24 to match the smaller input matrix input dimension of the second pass, likely due to decreased shared memory bandwidth. Based on

our NVVP results at different tile widths, there is a tradeoff between memory and compute efficiency. Using tile width 8, our kernel memory bandwidth was not a limiting factor, however, the compute efficiency decreased. This resulted in the kernel becoming compute limited. Using a tile width of 24 provided the best balance of compute and memory efficiency, at least for the second pass of convolution.

1.5 Optimization 8

1.5.1 Rationale

For optimization 8, we used different kernel implementations for the two different layers. In our matrix-matrix multiplication formulation, the matrix dimensions are 12×25 and 25×4356 for the first layer and 24×300 and 300×841 for the second layer. Since the matrix dimensions differ between the two layers, we hypothesized that using different tile widths for the two layers would improve performance since the optimal tile width is likely different for each layer.

1.5.2 Correctness and Timing

- Dataset 1, N=100:

```
Op Time: 0.000250
Op Time: 0.000554
Correctness: 0.76
```

- Dataset 2, N=1000:

```
Op Time: 0.002299
Op Time: 0.005244
Correctness: 0.767
```

- Dataset 3, N=10000:

```
Op Time: 0.022693
Op Time: 0.052049
Correctness: 0.7653
```

1.6 NVVP Analysis

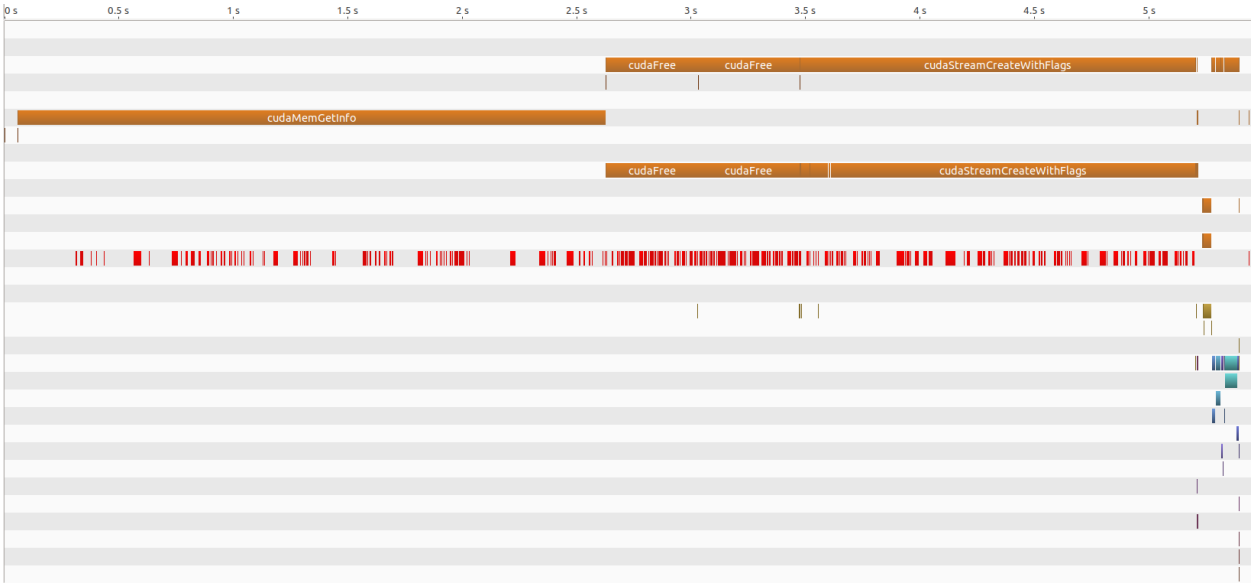

Figure 9: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls. This timeline demonstrates that there were two distinct kernels called for each of the two passes of the convolution layer.

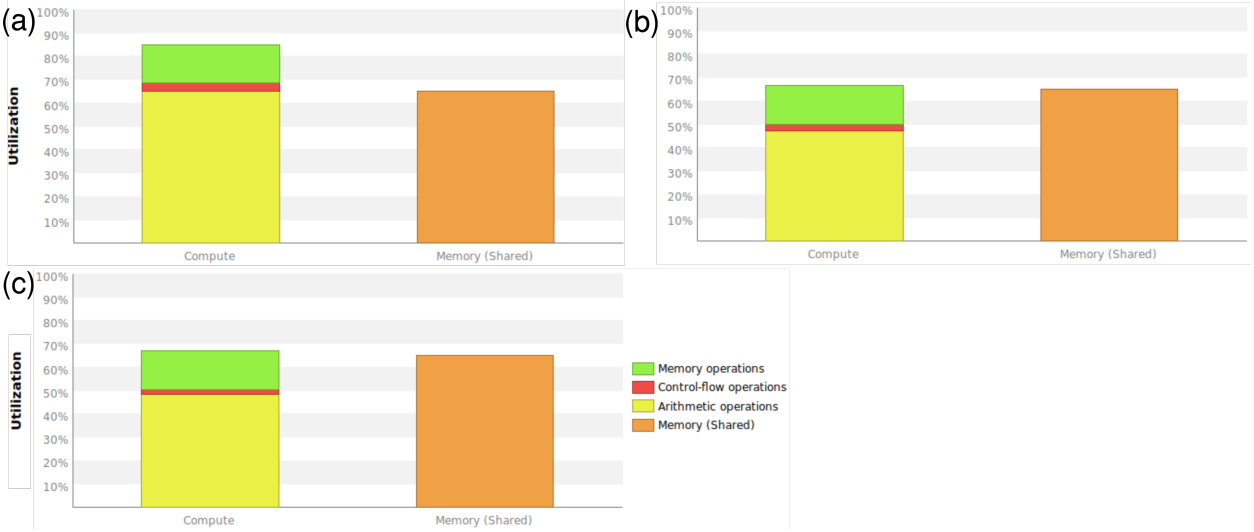

Figure 10: Compute and memory utilization for Dataset 3, N=10000 for (a) the first pass with tile width 16, (b) the first pass with tile width 24, and (c) the second pass of the convolution layer with tile width 24.

1.6.1 Discussion

Using separate matrix multiplication kernels improved our performance significantly primarily by decreasing the Op Time for the first convolution layer. Since the input matrices

for the first layer have smaller dimensions than the input matrices for the second layer, different tile widths are required for good balance of compute efficiency and memory efficiency. In the old implementation (Fig. 10b), NVVP analysis indicated that the kernel had significant memory bandwidth limitation in the first layer. Decreasing the tile width to 16 enabled lower memory bandwidth for the first layer without causing compute limitations in the second layer.

2 Milestone 4

2.1 Optimization 1

2.1.1 Rationale

For optimization 1, we varied the block size parameter in our original convolution implementation to determine its effect on overall performance. This follows logically from the results of the NVVP analysis of our baseline implementation, which indicated that high latency was a limiting factor to our performance. We increased the block size to determine if utilization would improve. Using a block size of 32, we obtained these timings:

2.1.2 Correctness and Timing

- Dataset 1, N=100:

```
Op Time: 0.000384
Op Time: 0.000943
Correctness: 0.76
4.98user 4.32system 0:05.92elapsed 157%CPU
```

- Dataset 2, N=1000:

```
Op Time: 0.004080
Op Time: 0.009161
Correctness: 0.767
4.65user 4.47system 0:05.73elapsed 159%CPU
```

- Dataset 3, N=10000:

```
Op Time: 0.040393
Op Time: 0.087113
Correctness: 0.7653
4.87user 4.66system 0:06.23elapsed 153%CPU
```

2.1.3 NVVP Analysis

Using NVVP to compare our previous implementation with block size 16 with our first optimization with block size 32, we found that a block size of 32 results in a compute utilization of 77%, as shown in Fig. 11. This is a 10% improvement of the previous compute utilization of 70%.

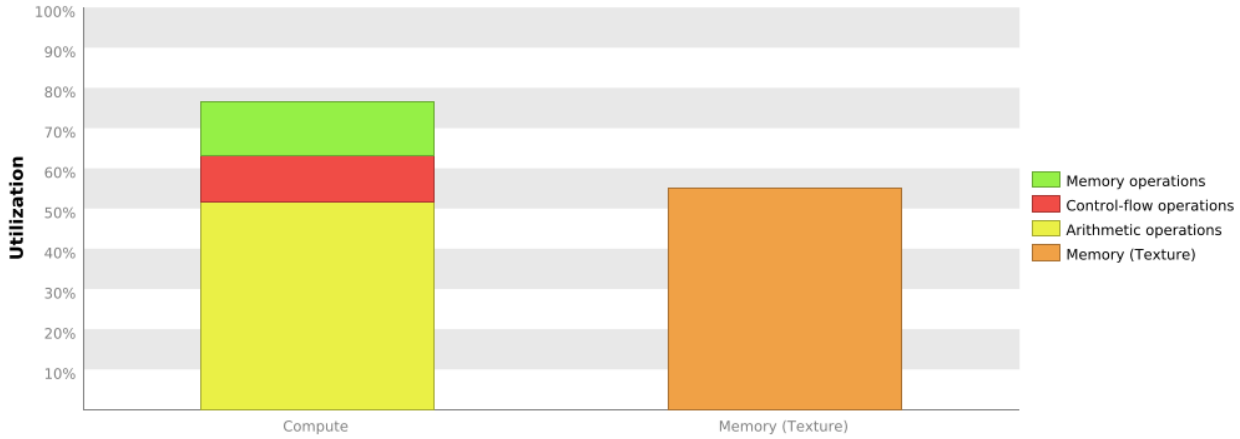

Figure 11: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

A drawback to this optimization was that the control divergence as a result of boundary checking increased from 50% to 90% of executions. Based on comparison of timings with our baseline, this increase in control divergence appears to have offset the performance improvement gained from increasing the block size.

2.2 Optimization 2

2.2.1 Rationale

For optimization 2, we recast the convolution as a tiled matrix multiplication. Our reasoning for using tiled matrix multiplication rather than tiled convolution is that while both methods reduce global memory reads, tiling provides more performance improvement over the untiled implementation for matrix multiplication than for convolution. Using a tiled matrix multiplication recasting of convolution, we obtained these timings:

2.2.2 Correctness and Timing

- Dataset 1, N=100:

```
Op Time: 0.001922
Op Time: 0.002711
Correctness: 0.76
5.06user 3.16system 0:04.42elapsed 185%CPU
```

- Dataset 2, N=1000:

```
Op Time: 0.014818
Op Time: 0.026517
Correctness: 0.767
4.87user 3.01system 0:04.22elapsed 186%CPU
```

- Dataset 3, N=10000:

```
Op Time: 0.145230
Op Time: 0.243535
Correctness: 0.7653
5.42user 3.37system 0:04.99elapsed 176%CPU
```

2.2.3 NVVP Analysis

The execution timeline for dataset 3 is shown in Fig. 12. The majority of the program runtime is spent on API calls. The unroll and matrix multiply kernels occupy $\sim$2.5% and $\sim$3.7% of total program runtime, respectively.

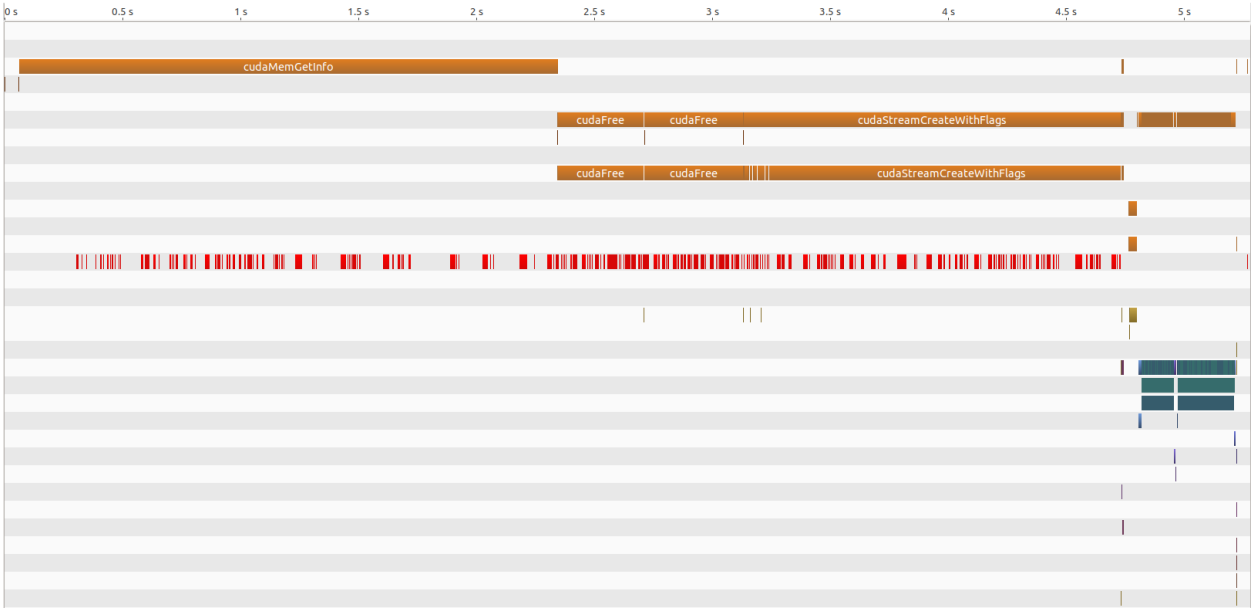

Figure 12: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

Compute and memory bandwidth utilizations are shown for the unroll and tiled matrix multiplication kernels in Figs. 13 and 14, respectively.

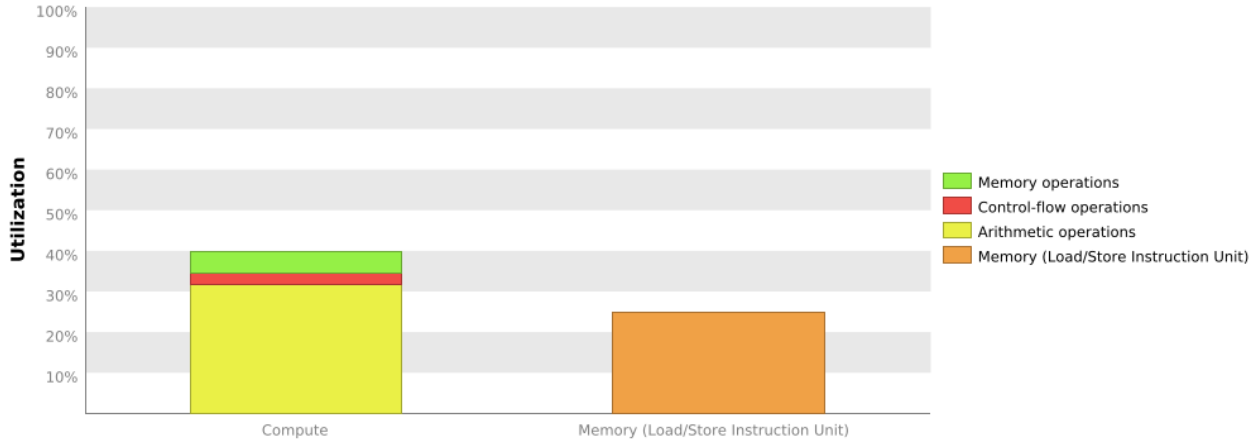

Figure 13: Compute and memory bandwith utilization for the unroll kernel. Data shown for Dataset 3, N=10000.

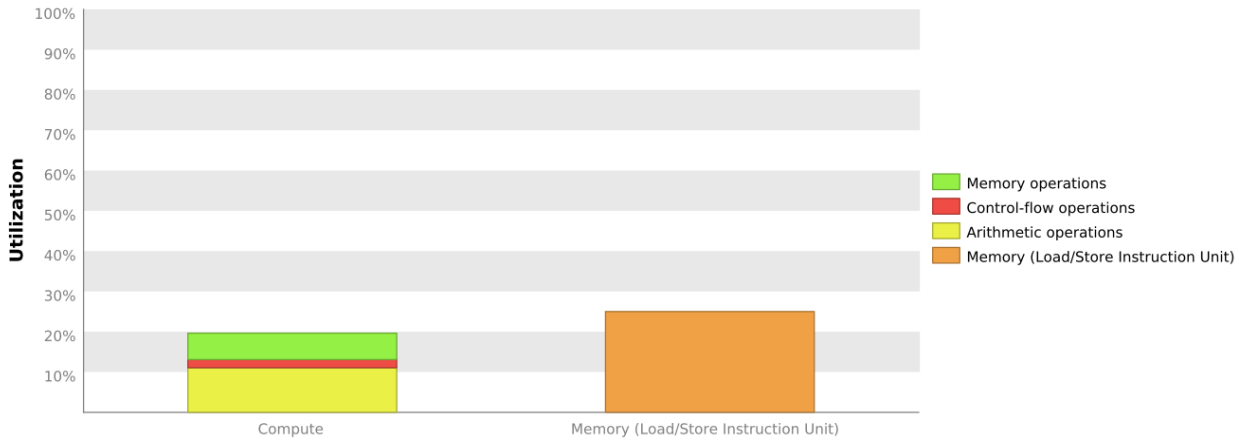

Figure 14: Compute and memory bandwith utilization for the tiled matrix multiplication kernel. Data shown for Dataset 3, N=10000.

Based on the NVVP results, the performance of this implementation is limited both by low memory bandwidth usage and low compute usage relative to the compute capacity of the GPU. A possible reason for this is that for each convolution layer, our code calls the unroll and matrix multiply kernels B times, where B is the batch size. For further optimizations, parallelizing rather than iterating over batch size to unroll and multiply matrices could improve our kernel performance by a factor of B.

2.3 Optimization 3

2.3.1 Rationale

For optimization 3, we stored the kernels in constant memory. Our reasoning for implementing this optimization is that in the implementation of tiled matrix multiplication, each element of the kernel matrix still needs to be read from global memory and stored into shared memory, whereas when using constant memory, no global memory reads are required to access elements of the kernel matrix.

2.3.2 Correctness and Timing

- Dataset 1, N=100:

```
Op Time: 0.002148
Op Time: 0.005546
Correctness: 0.76
4.80user 3.06system 0:04.45elapsed 176%CPU
```

- Dataset 2, N=1000:

```
Op Time: 0.018123
Op Time: 0.054892
Correctness: 0.767
4.77user 3.25system 0:04.25elapsed 188%CPU
```

- Dataset 3, N=10000:

```
Op Time: 0.163737
Op Time: 0.495567
Correctness: 0.7653
5.21user 3.42system 0:05.07elapsed 170%CPU
```

2.3.3 NVVP Analysis

The execution timeline for dataset 3 is shown in Fig. 15. The majority of the program runtime is spent on API calls. The unroll and matrix multiply kernels occupy $\sim$2.5% and $\sim$3.6% of total program runtime, respectively.

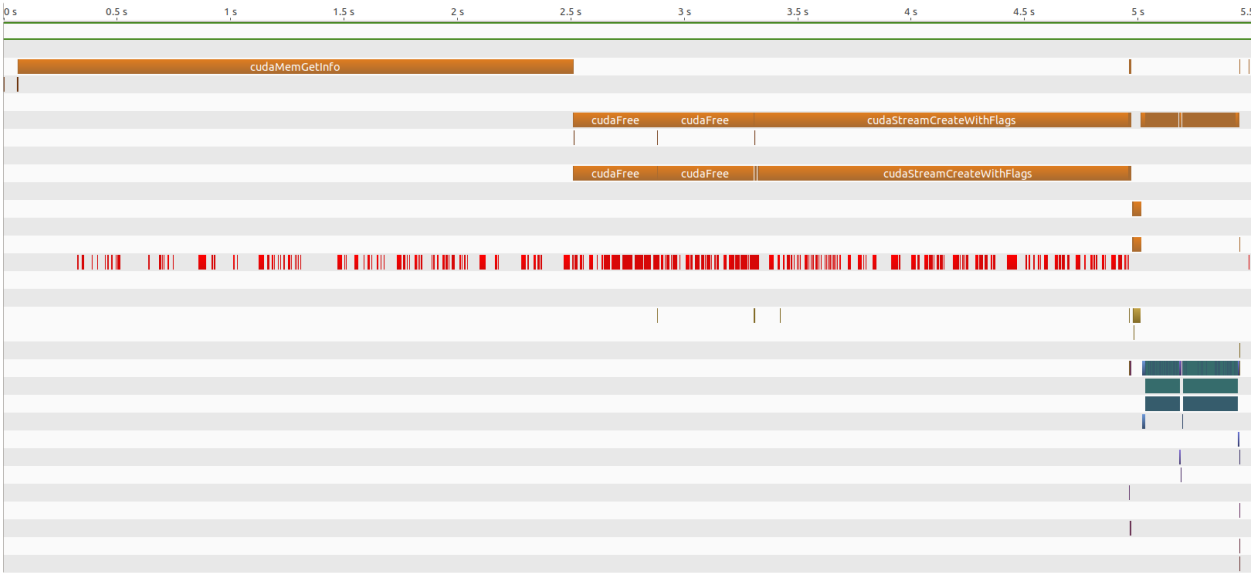

Figure 15: Program execution timeline for Dataset 3, N=1000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

Compute and memory bandwidth utilization for the unroll kernel and the matrix multiplication kernel in Figs. 16 and 17, respectively.

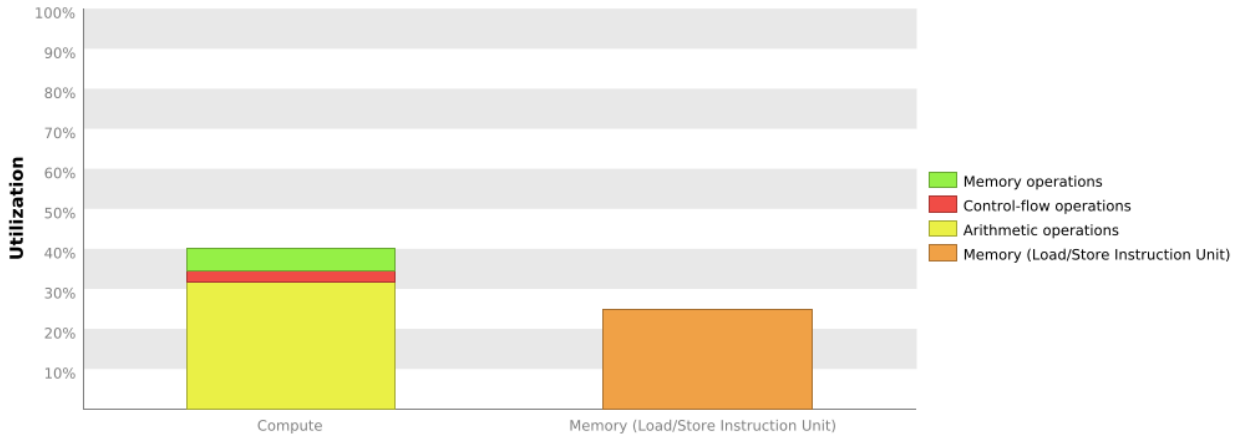

Figure 16: Compute and memory bandwith utilization for the unroll kernel. Data shown for Dataset 3, N=10000.

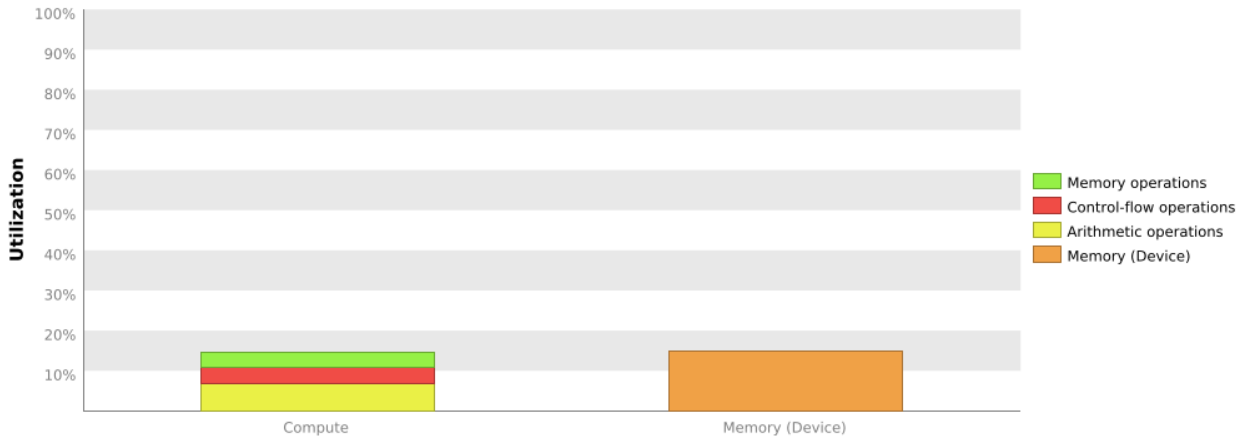

Figure 17: Compute and memory bandwith utilization for the tiled matrix multiplication kernel. Data shown for Dataset 3, N=10000.

Based on the NVVP results, the unroll kernel has the same performance for both optimizations 2 and 3, as is expected since the change in optimization 3 only affects the tiled matrix multiplication kernel. The tiled matrix multiply kernel shows even lower compute utilization than the tiled multplication implementation where the kernel is not stored in constant memory. Based on the memory operations contributions to overall utilization shown in Figs. 14 and 17, this can attributed primarily to reduction in memory bandwidth requirement that results from not needing to load the kernel matrix from global into shared memory. As previously noted, a possible path for future optimization is to parallelize over the B (batch size) variable rather than call the unroll and matrix multply kernels B times to improve resource utilization.

3 Milestone 3

3.1 Correctness and Timing of GPU Implementation

- Dataset 1, N=100:

```
Op Time: 0.000265
Op Time: 0.000910
Correctness: 0.76
4.91user 3.08system 0:04.42elapsed 180%CPU
```

- Dataset 2, N=1000:

```
Op Time: 0.003134
Op Time: 0.009863
Correctness: 0.767
4.82user 2.99system 0:04.21elapsed 185%CPU
```

- Dataset 3, N=10000:

```
Op Time: 0.032332
Op Time: 0.096688
Correctness: 0.7653
5.07user 3.29system 0:04.54elapsed 184%CPU
```

3.2 Profiling with nvprof and NVVP

Overall program execution timelines are shown in Figs. 18-20. For all three datasets, the majority of program execution time is spent on API calls, specifically, cudaMemGetInfo, cudaFree, and cudaStreamCreateWithFlags. The forward convolution kernel is called twice, once for each layer of the model. For the first two datasets (N=100, N=1000), the time spent executing the forward convolution kernel is nearly negligible in relation to the time spent executing API calls. For the third dataset (N=10000), the time spent executing the kernel is still significantly less than the time spent executing API calls. However, the kernel run time for the third dataset is now a more appreciable portion of total program run time, occupying $\sim$2.5% of program run time.

3.2.1 Timeline

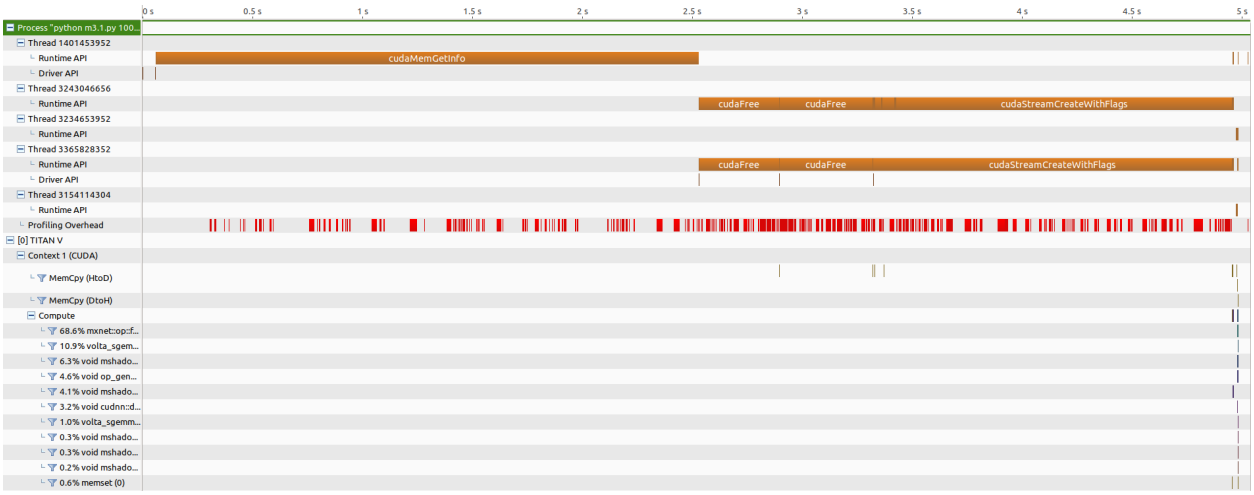

Figure 18: Program execution timeline for Dataset 1, N=100. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

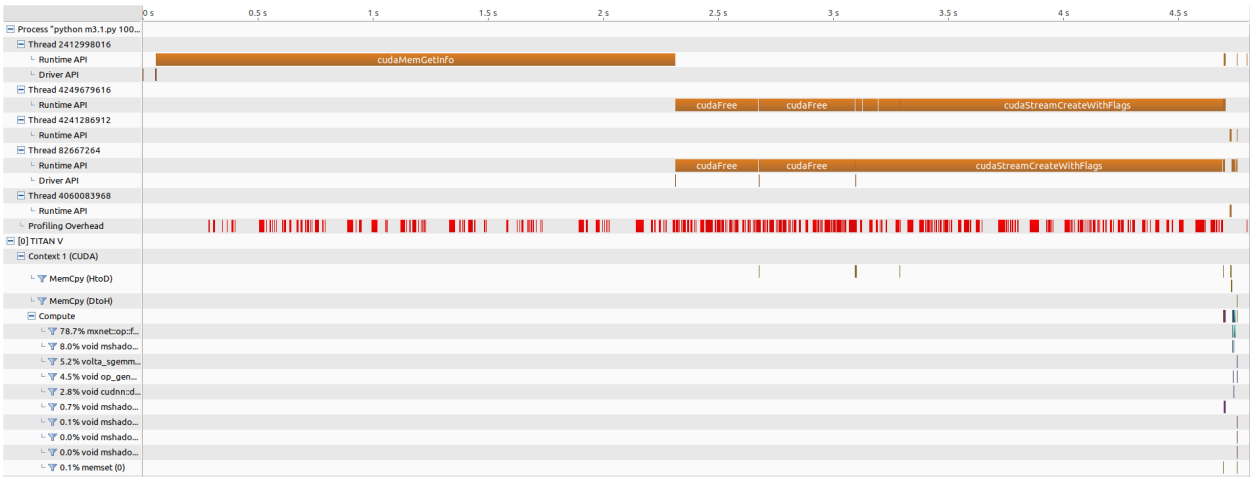

Figure 19: Program execution timeline for Dataset 2, N=1000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

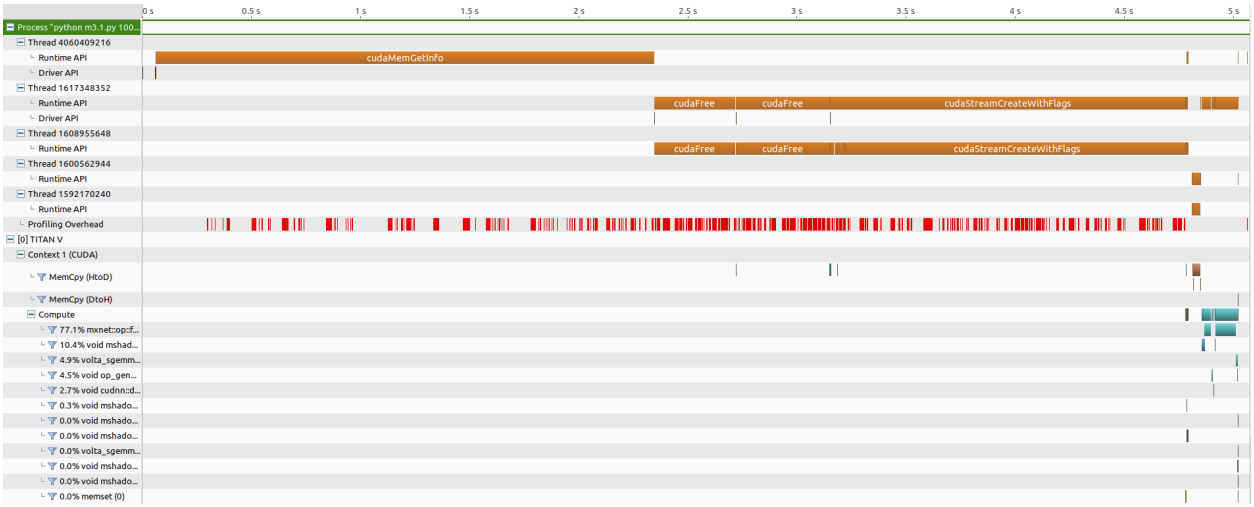

Figure 20: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

3.2.2 Analysis

Program Time Consumption by Different Kernels

The top kernels consuming time on the GPU are listed below. Reported values are for dataset 3 (N=10000), however, the time percentages were fairly consistent across all three datasets.

1. mxnet::op::forward_kernel (77%)
2. mshadow::cuda::MapPlanLargeKernel (10.4%)
3. volta_sgemm_128x128_tn (4.9%)
4. op_generic_tensor_kernel (4.5%)
5. cudnn::detail::pooling_fw_4d_kernel (2.7%)

All other kernels collectively consume less than 1% of GPU time.

According to the Performance-Critical Kernels Utility within NVVP, the optimization importance of the different kernels is:

1. mxnet::op::forward_kernel (Rank 100)
2. mxnet::op::forward_kernel (Rank 32)
3. mshadow::cuda::MapPlanLargeKernel (Rank 14)
4. volta_sgemm_128x128_tn (Rank 8)
5. op_generic_tensor_kernel (Rank 7)
6. cudnn::detail::pooling_fw_4d_kernel (Rank 4)

Higher rank score indicates higher potential for optimization, which indicates that the forward convolution kernel has the highest potential for optimization.

Performance Analysis
Since "mxnet::op::forward_kernel" consumes the most of program time and has the highest potential of optimization, we focus on analyzing the performance of this kernel.

Overall, the performance is similar for the 3 different datasets. Here, we will take the forward kernel 2 and Dataset 3 (N=10000) as an example to illustrate the performance.

1. **"mxnet::op::forward_kernel" performance is bound by instruction and memory latency.** As shown in Fig. 21, the current kernel exhibits low compute throughput and memory bandwidth utilization. This suggests the performance of this kernel could be limited by arithmetic and memory operations. Further optimization should focus on enhancing the utilization to achieve the peak performance.

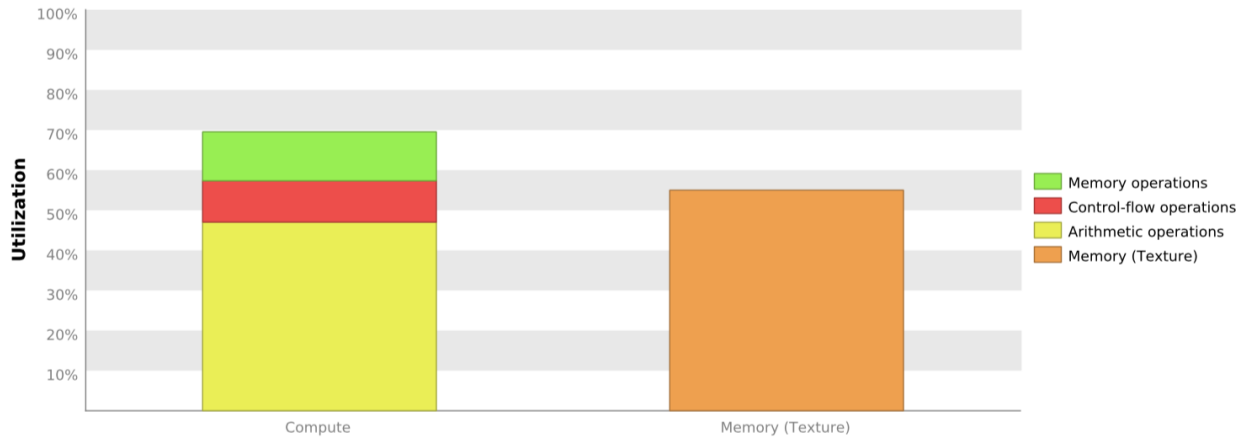

Figure 21: Utilization for forward kernel 2 and Dataset 3, N=10000.

2. **Low warp execution efficiency and divergent branches lead to poor computational resources utilization.** According to NVVP report, the warp execution efficiency is 76.6% ($<$ 100%) due to divergent branches and predicted instructions. Specifically, Line 39 in new-forward.cuh file has the divergence = 16.5%. This suggests that the divergence in branch and prediction behavior wastes a large fraction of available compute resources and thus limits the performance. Further optimizations should focus on reducing the divergence in each warp to achieve better warp execution efficiency.

4 Milestone 2

4.1 Kernels that collectively consume more than 90% of the program time

1. CUDA memcpy HtoD (30.43%)
2. volta_scudnn_128x64_relu_interior_nn_v1 (17.97%)
3. volta_gcgemm_64x32_nt (17.29%)
4. fft2d_c2r_32x32 (8.80%)
5. volta_sgemm_128x128_tn (7.85%)
6. op_generic_tensor_kernel (6.60%)
7. fft2d_r2c_32x32 (6.48%)

4.2 CUDA API calls that collectively consume more than 90% of the program time

1. cudaStreamCreateWithFlags (41.94%)
2. cudaMemGetInfo (32.96%)
3. cudaFree (21.22%)

4.3 Explanation of the difference between kernels and API calls

Kernels are code that is called from the CPU but is executed on the GPU. API calls on the other hand are called from the CPU and are executed on the CPU but are able to send the required information to the GPU, such as memcpy information.

4.4 Output of rai running MXNet on the CPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
17.09user 4.49system 0:08.92elapsed 241%CPU (0avgtext+0avgdata 6045876maxre
sident)k
0inputs+2824outputs (0major+1602890minor)pagefaults 0swaps
```

4.5 Program run time on CPU

17.09user 4.49system 0:08.92elapsed 241%CPU

4.6 Output of rai running MXNet on the GPU

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8154}
4.91user 3.48system 0:04.71elapsed 177%CPU
(0avgtext+0avgdata 2981120maxresident)k
0inputs+1712outputs (0major+732138minor)pagefaults 0swaps
```

4.7 Program run time on GPU

4.91user 3.48system 0:04.71elapsed 177%CPU

4.8 Whole program execution time

m1.1.py: 17.09user 4.49system 0:08.92elapsed
m1.2.py: 4.91user 3.48system 0:04.71elapsed
m2.1.py: 83.12user 8.17system 1:13.74elapsed
Total: 105.12user, 16.14system, 1:27.37elapsed

4.9 Op times

Op Time: 10.801294
Op Time: 59.181779
Correctness: 0.7653

5 Milestone 1

Team Name: fast
Team Members: Jiming Chen, Jiangyan Feng, Maxim Korolkov