Team Name: Fast

ECE408 Final Project Report: Milestone 4

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

November 21, 2019

1 Milestone 4

1.1 Optimization 1

1.1.1 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:

1.1.2 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
```

1.1.3 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. 1. This is a 10% improvement of the previous compute utilization of 70%.

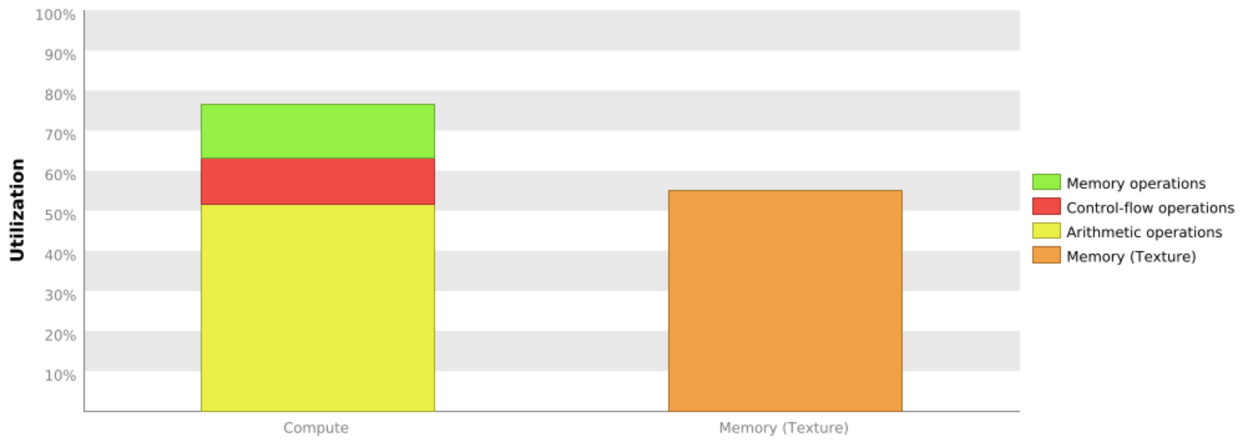

Figure 1: Program execution timeline for Dataset 3, N=1000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

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.

1.2 Optimization 2

1.2.1 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:

1.2.2 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
```

1.2.3 NVVP Analysis

The execution timeline for dataset 3 is shown in Fig. 2. 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.

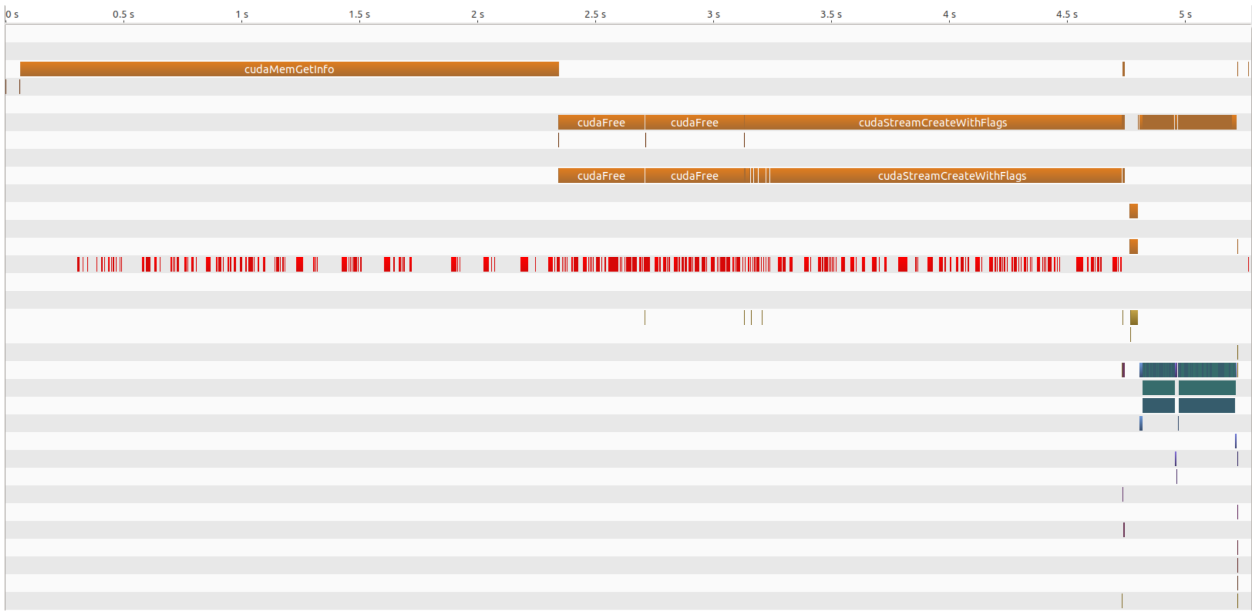

Figure 2: Program execution timeline for Dataset 3, N=1000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

Compute and memory bandwidth utilizations are shown for the unroll and tiled matrix multiplication kernels in Figs. 3 and 4, respectively.

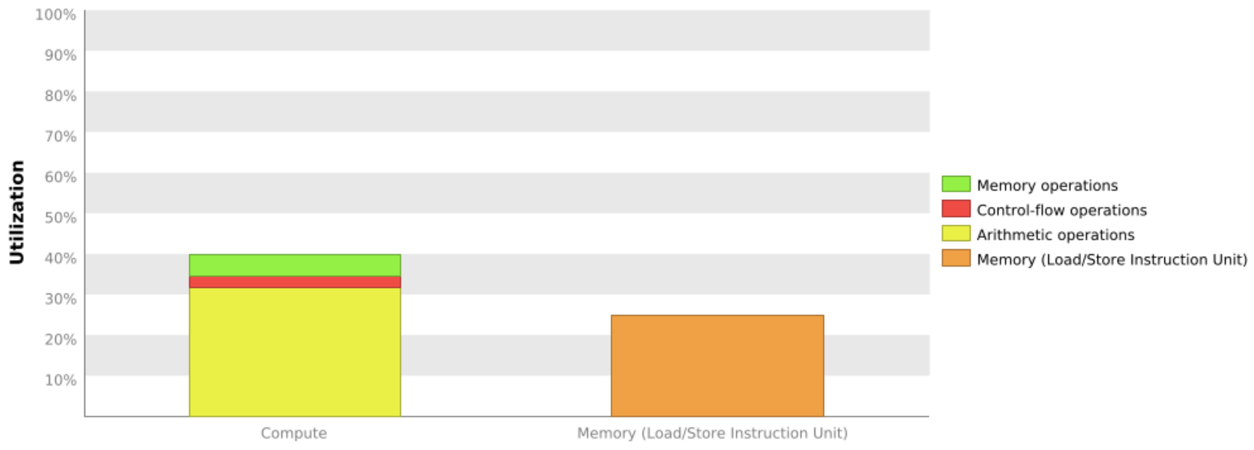

Figure 3: Compute and memory bandwith utilization for the unroll kernel. Data shown for Dataset 3, N=10000.

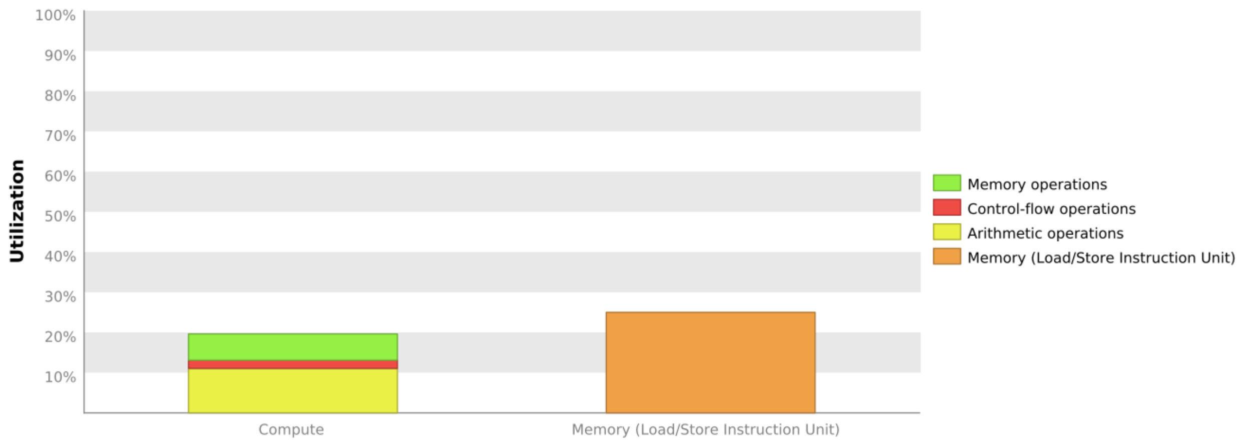

Figure 4: Compute and memory bandwith utilization for the tiled matrix multiplication kernel. Data shown for Dataset 3, N=10000.

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.

1.3 Optimization 3

1.3.1 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.

1.3.2 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
```

1.3.3 NVVP Analysis

The execution timeline for dataset 3 is shown in Fig. 5. 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.

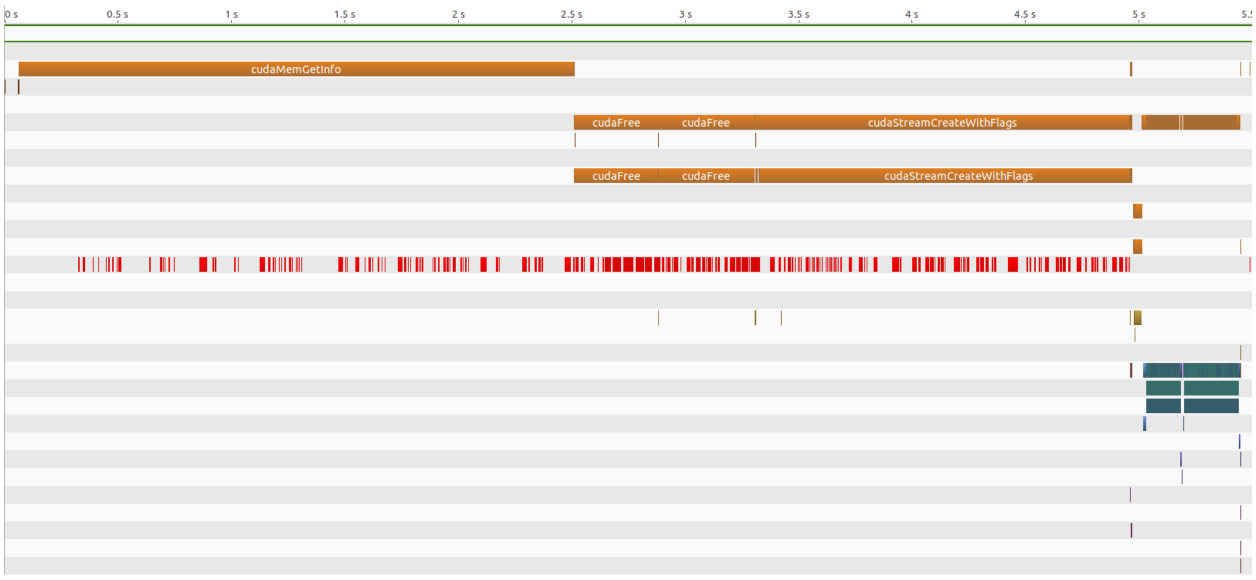

Figure 5: Program execution timeline for Dataset 3, N=1000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

Compute and memory bandwidth utilization for the unroll kernel and the matrix multiplication kernel in Figs. 6 and 7, respectively.

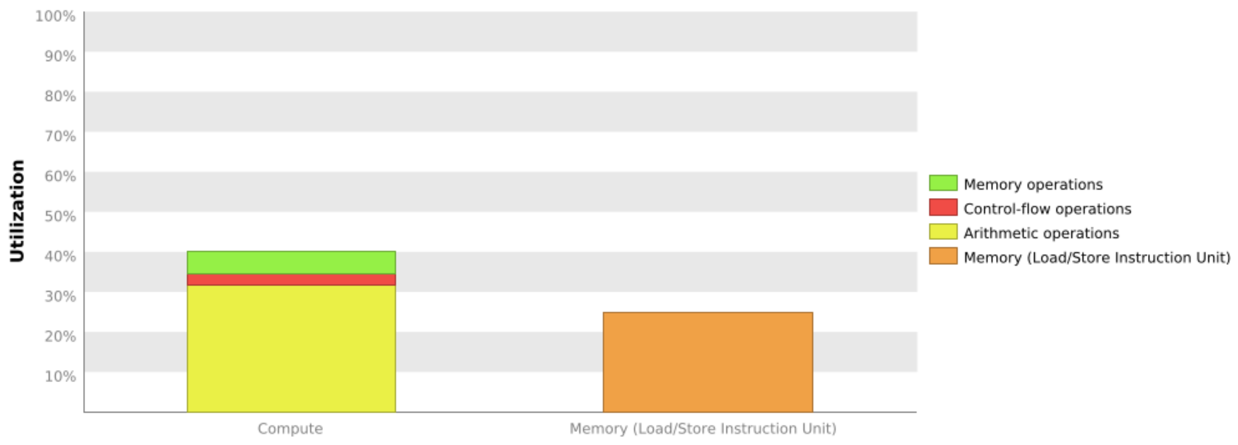

Figure 6: Compute and memory bandwith utilization for the unroll kernel. Data shown for Dataset 3, N=10000.

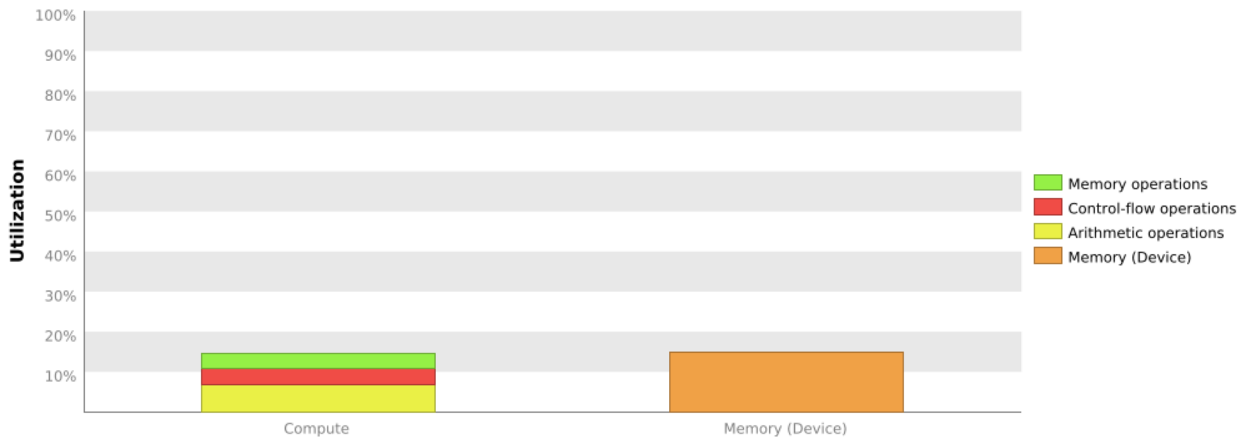

Figure 7: Compute and memory bandwith utilization for the tiled matrix multiplication kernel. Data shown for Dataset 3, N=10000.

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. 4 and 7, 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.

2 Milestone 3

2.1 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
```

2.2 Profiling with nvprof and NVVP

Overall program execution timelines are shown in Figs. 8-10. 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 ∼2.5% of program run time.

2.2.1 Timeline

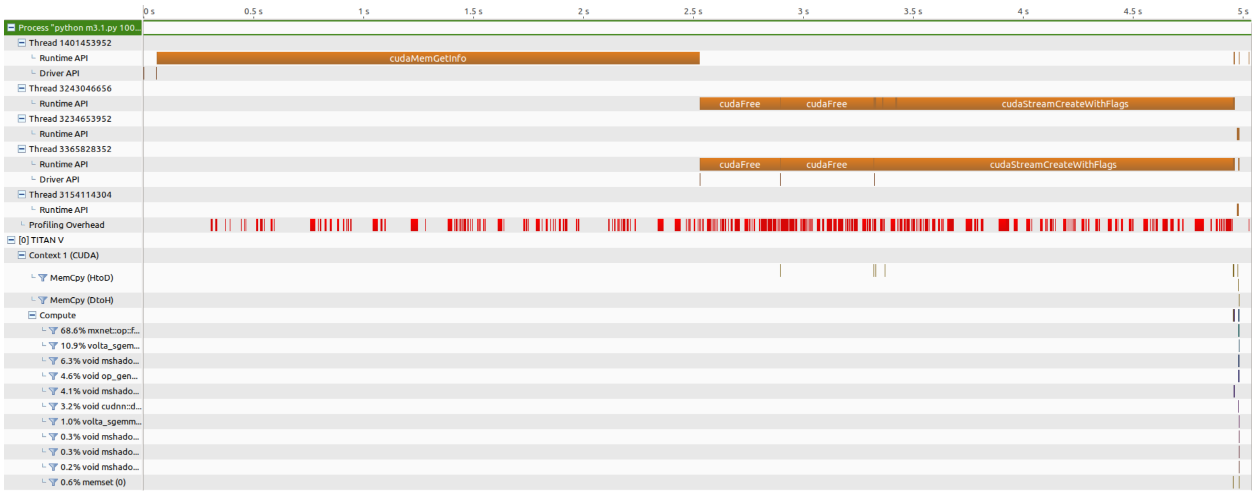

Figure 8: Program execution timeline for Dataset 1, N=100. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

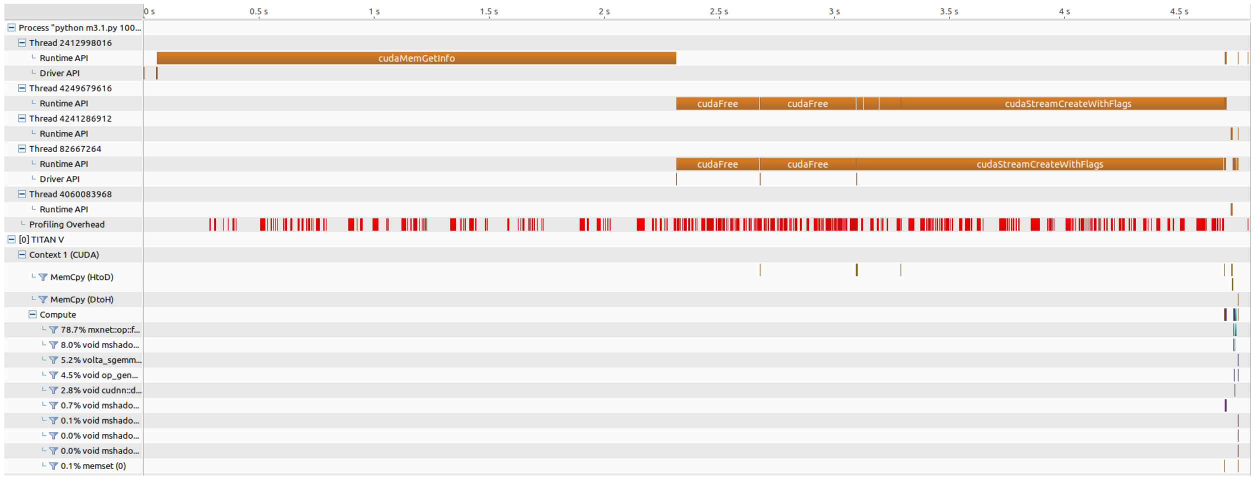

Figure 9: Program execution timeline for Dataset 2, N=1000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

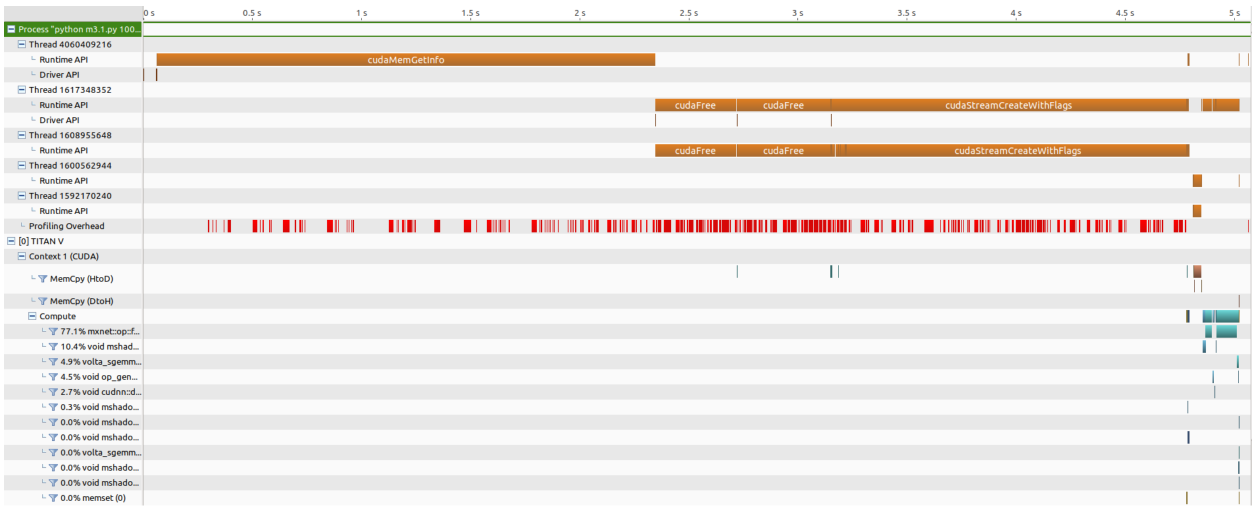

Figure 10: Program execution timeline for Dataset 3, N=10000. Orange bars show API calls, red bars show profiling overhead, and cyan bars show kernel calls.

2.2.2 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. 11, 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.

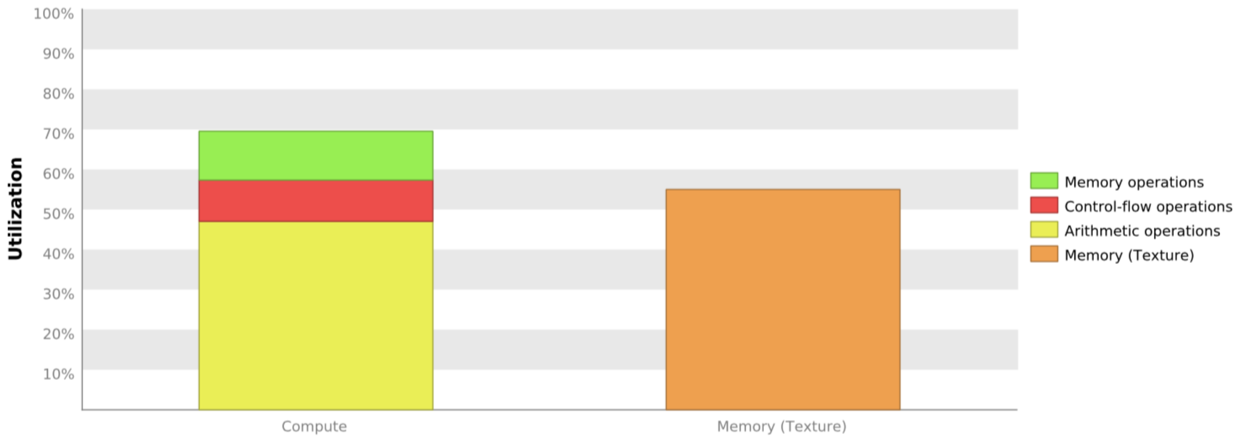

Figure 11: Utilization for forward kernel 2 and Dataset 3, N=10000.

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.

3 Milestone 2

3.1 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%)

3.2 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%)

3.3 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.

3.4 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
```

3.5 Program run time on CPU

17.09user 4.49system 0:08.92elapsed 241%CPU

3.6 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
```

3.7 Program run time on GPU

4.91user 3.48system 0:04.71elapsed 177%CPU

3.8 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

3.9 Op times

Op Time: 10.801294
Op Time: 59.181779
Correctness: 0.7653

4 Milestone 1

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