University of Illinois at Urbana Champaign

Applied Parallel Programming

Team: wandering-gpu

Final Project

Author	*Net ID*
Alvin Sun	yixiaos3
Yuqi Xue	yuqixue2
Yan Miao	yanmiao2

April 18, 2019

Milestone 1

1 Kernel Statistics

Time(%)	Time	Calls	Avg	Min	Max	Name
40.10%	16.788ms	20	839.42us	1.1200us	16.155ms	[CUDA memcpy HtoD]
20.18%	8.4497ms	1	8.4497ms	8.4497ms	8.4497ms	void cudnn::detail::implicit_convolve_sgemm
11.81%	4.9434ms	1	4.9434ms	4.9434ms	4.9434ms	volta_cgemm_64x32_tn
7.05%	2.9497ms	2	1.4748ms	25.568us	2.9241ms	void op_generic_tensor_kernel
5.69%	2.3830ms	1	2.3830ms	2.3830ms	2.3830ms	void fft2d_c2r_32x32
5.59%	2.3404ms	1	2.3404ms	2.3404ms	2.3404ms	volta_sgemm_128x128_tn
4.55%	1.9059ms	1	1.9059ms	1.9059ms	1.9059ms	void cudnn::detail::pooling_fw_4d_kernel
4.18%	1.7480ms	1	1.7480ms	1.7480ms	1.7480ms	void fft2d_r2c_32x32

2 CUDA API Statistics

Time(%)	Time	Calls	Avg	Min	Max	Name
41.94%	2.94373s	22	133.81ms	13.721us	1.52482s	cudaStreamCreateWithFlags
34.43%	2.41664s	24	100.69ms	97.853us	2.41057s	cudaMemGetInfo
20.93%	1.46898s	19	77.315ms	817ns	393.98ms	cudaFree

3 Differences Between Kernels & API Calls

A CUDA kernel is an extended C function that, when called, are executed multiple times in parallel by different CUDA threads on the GPU. The CUDA APIs are programming interfaces that allow the programmer to use the CUDA device, i.e. the GPU. Kernel functions are written by the programmer and are meant to execute specific (mostly computation-intensive) tasks on the GPU, while API calls are provided by the CUDA library and are to manage the CUDA runtime environment and mostly prepare for the execution of kernels. While kernels are always executed by CUDA cores, CUDA APIs do not necessarily involve the execution of CUDA cores.

4 MXNet CPU Execution

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8236}
```

Run Time. 5.06s

5 MXNet GPU Execution

```
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8236}
```

Run Time: 4.40s

Milestone 2

Full CPU Time	11.32s
First Layer Time	2.405296s
Second Layer Time	7.342860

10000 images

Full CPU Time	0.250576s
First Layer Time	0.756567s
Second Layer Time	2.03s

1000 images

Full CPU Time	1.08s
First Layer Time	0.035050s
Second Layer Time	0.075312s

100 images

Table 1: CPU Run Time Statistics

Milestone 3

1 Execution Summary

Accuracy	0.8397
First Layer Time	9.252ms
Second Layer Time	19.331ms

10000 images

Accuracy	0.852
First Layer Time	0.677ms
Second Layer Time	1.968ms

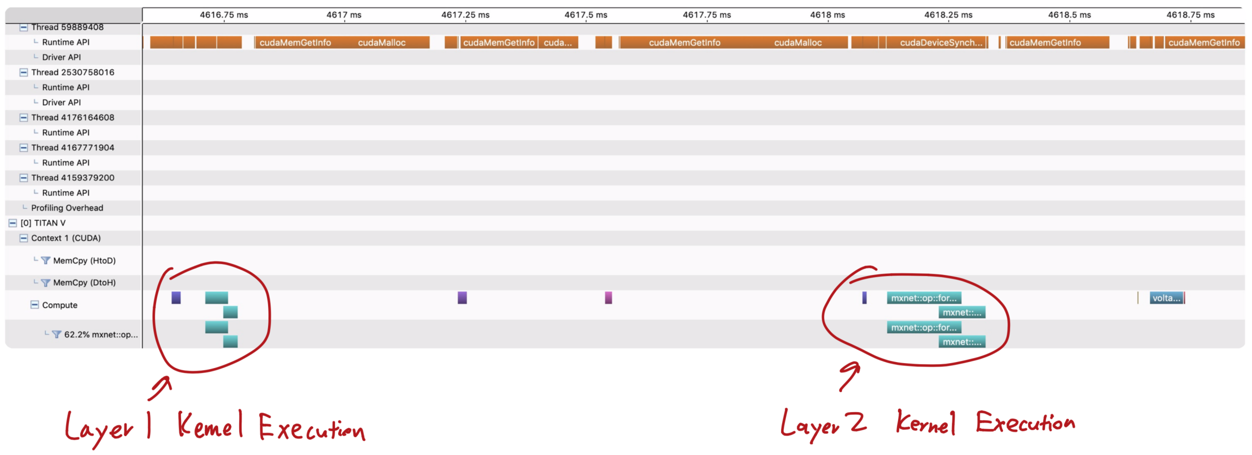
1000 images

Accuracy	0.84
First Layer Time	0.121ms
Second Layer Time	0.248ms

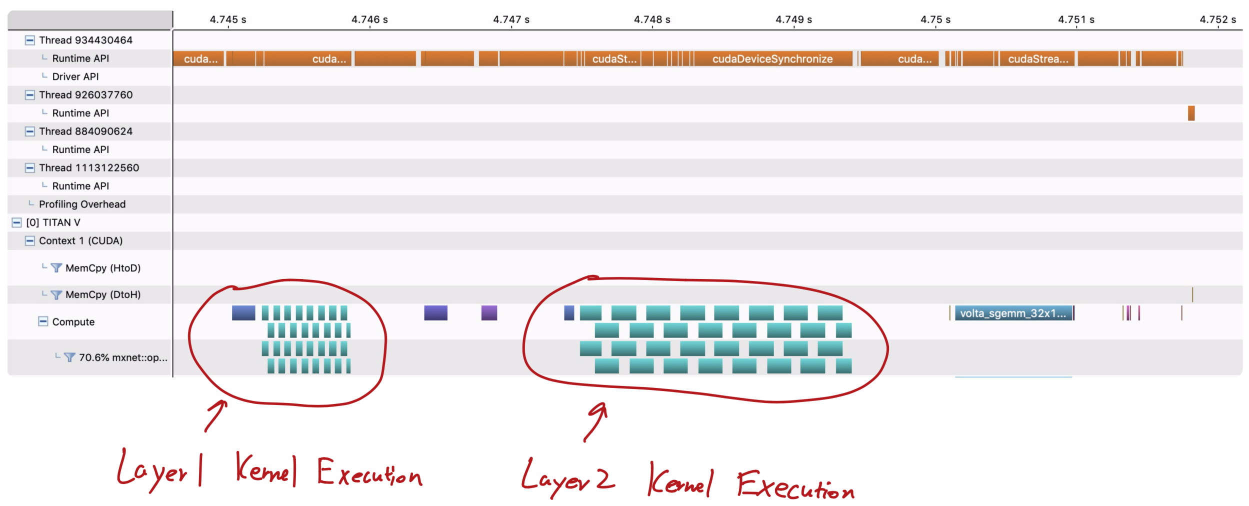
100 images

Table 2: GPU Run Time Statistics

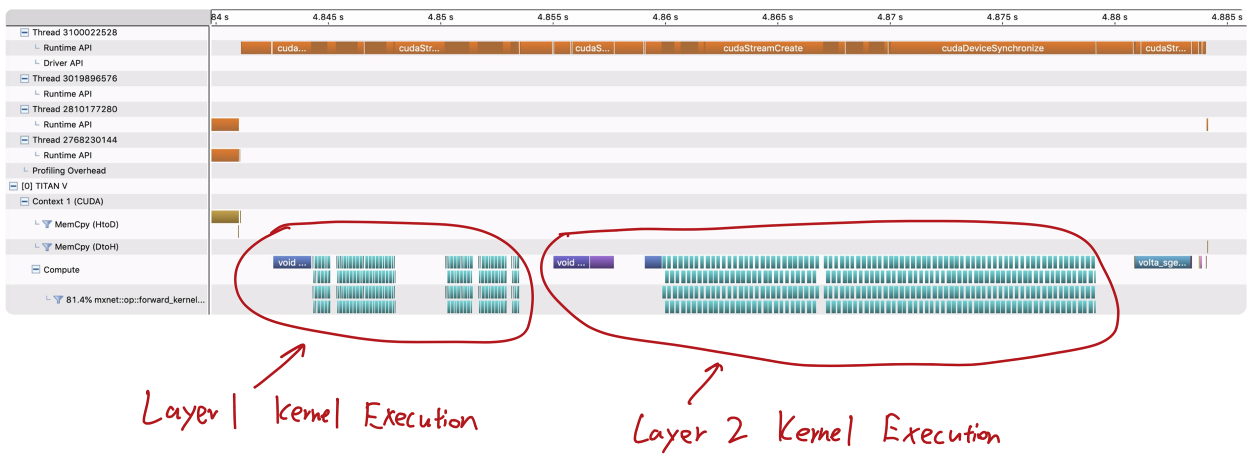
2 Execution Timeline



(a) Dataset Size 100



(b) Dataset Size 1000



(c) Datset Size 10000

Figure 1: Kernel Execution Timelines generated by NVVP.

Milestone 4

1 Execution Summary

Upon Milestone 4, we have implemented three optimizations to our kernel function, including having different implementations and tuning for different layers, using shared memory convolution, and putting convolution kernels in constant memory. The overall execution (on server) involving all three optimizations yields the following results.

Accuracy	0.8397
First Layer Time	3.560ms
Second Layer Time	12.878ms

10000 images

Accuracy	0.852
First Layer Time	0.366ms
Second Layer Time	1.180ms

1000 images

Accuracy	0.84
First Layer Time	0.074ms
Second Layer Time	0.149ms

100 images

Table 3: Run time statistics after applying all three optimizations.

We will discuss each optimization in the following section.

2 Optimizations

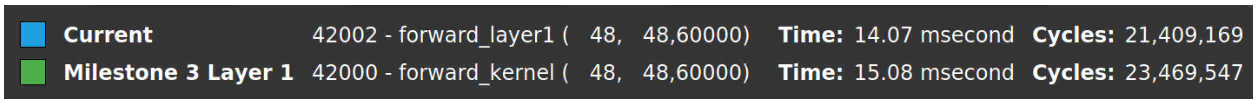
Note As a preliminary note, we would like to mention that, in order to avoid unexpected server fluctuations and speedup our development, we have setup a local development environment using an NVIDIA GeForce RTX 2080 running CUDA 10.0. Our discussions below will be based on execution results and statistics generated in our local environment. Despite this, we have ensured that our optimizations work as intended on the server. Also, for simplicity, we will only benchmark data size 10000.

2.1 Seperate Kernel Implementations for Different Layers

As far as we observe, the original kernel function, although general enough to be used for both layers, has a function signiture

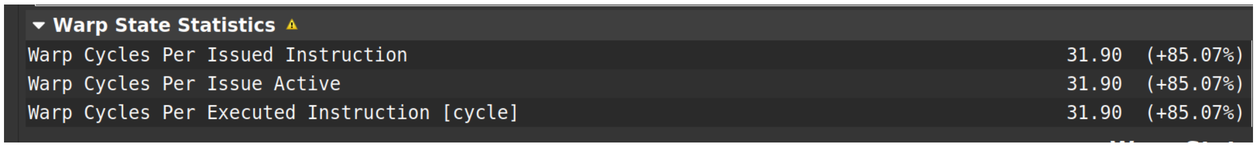
```
__global__ void forward_kernel(float *y, const float *x, const float *k,
        const int B, const int M, const int C, const int H, const int W, const int K)
```

that is too long and many of the parameters are unnecessary as we already know them for each layer. Hence, one optimization involves implementing two different kernel functions for these two different layers. In other words, this optimization is barely finding and hardcoding the best possible parameters for each layer, including layer input and output channel numbers, input and output dimensions, and, most importantly, kernel launch parameters that best avoid warp control divergences.
By profiling our optimized version using NVIDIA Nsight Compute, we observe a great deal of increase in execution time and decrease in warp divergences.



🟦	**Current**	42002 - forward_layer1 (48, 48,60000)	**Time:** 14.07 msecond	**Cycles:** 21,409,169
🟩	**Milestone 3 Layer 1**	42000 - forward_kernel (48, 48,60000)	**Time:** 15.08 msecond	**Cycles:** 23,469,547

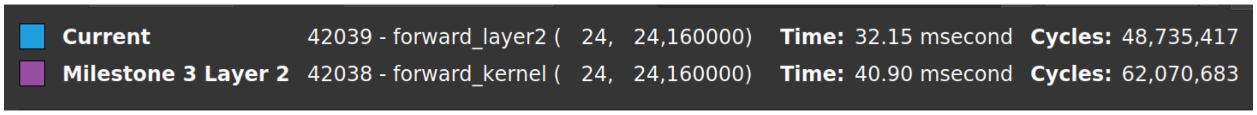
(a) Execution Time



```
▾ Warp State Statistics ⚠
Warp Cycles Per Issued Instruction                31.90  (+85.07%)
Warp Cycles Per Issue Active                      31.90  (+85.07%)
Warp Cycles Per Executed Instruction [cycle]      31.90  (+85.07%)
```

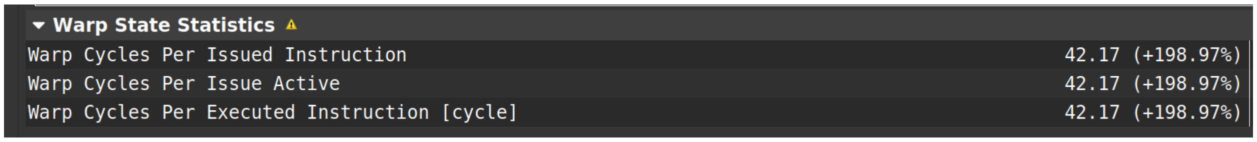
(b) Warp Statistics

Figure 2: Layer 1 Statistics



🟦	**Current**	42039 - forward_layer2 (24, 24,160000)	**Time:** 32.15 msecond	**Cycles:** 48,735,417
🟪	**Milestone 3 Layer 2**	42038 - forward_kernel (24, 24,160000)	**Time:** 40.90 msecond	**Cycles:** 62,070,683

(a) Execution Time



```
▾ Warp State Statistics ⚠
Warp Cycles Per Issued Instruction                42.17 (+198.97%)
Warp Cycles Per Issue Active                      42.17 (+198.97%)
Warp Cycles Per Executed Instruction [cycle]      42.17 (+198.97%)
```

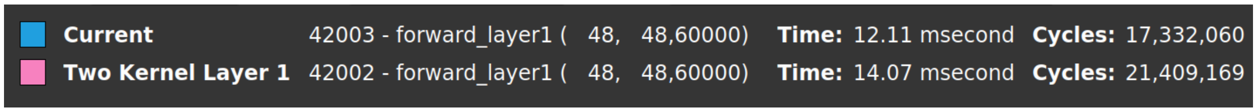
(b) Warp Statistics

Figure 3: Layer 2 Statistics

Note Now, we have already implemented different kernel functions for the two layers. Since optimizations in Section 2.1 are merely parameter tuning for each layer, we would like to directly implement further optimizations based upon the two-kernel version in Section 2.1 and analyze the results by comparing to that version instead of the original unoptimized version.

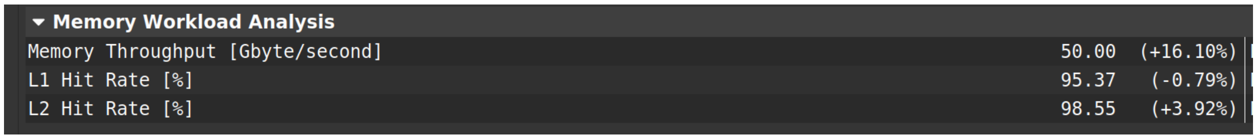
2.2 Kernels in Constant Memory

One optimization is to put kernels in constant memory by using cudaMemcpyToSymbol() API call. This optimization is possible because convolution kernels are effectively read-only for each CUDA kernel launch. The contribution by this optimization alone results in a relatively small amount of improvement in runtime due to increasing efficiency of memory accessing throughput. The constant memory kernel alone did not boost up too much the performance due to the bottleneck of global memory fetching induced by the input element accesses.



```
Current            42003 - forward_layer1 (  48,  48,60000)  Time: 12.11 msecond  Cycles: 17,332,060
Two Kernel Layer 1 42002 - forward_layer1 (  48,  48,60000)  Time: 14.07 msecond  Cycles: 21,409,169
```

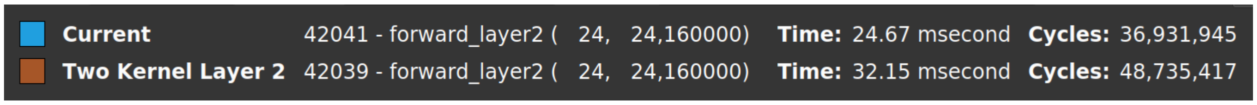
(a) Execution Time



```
Memory Workload Analysis
Memory Throughput [Gbyte/second]    50.00  (+16.10%)
L1 Hit Rate [%]                     95.37   (-0.79%)
L2 Hit Rate [%]                     98.55   (+3.92%)
```

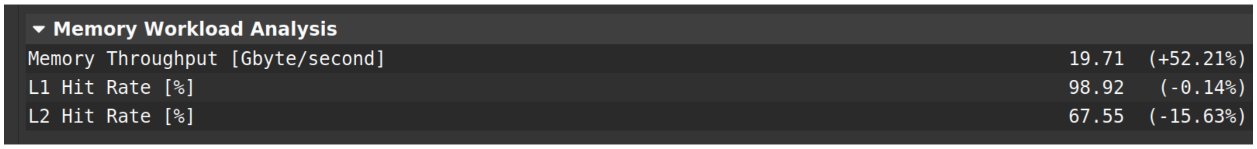
(b) Memory Throughput

Figure 4: Layer 1 Statistics



```
Current            42041 - forward_layer2 (  24,  24,160000)  Time: 24.67 msecond  Cycles: 36,931,945
Two Kernel Layer 2 42039 - forward_layer2 (  24,  24,160000)  Time: 32.15 msecond  Cycles: 48,735,417
```

(a) Execution Time



```
Memory Workload Analysis
Memory Throughput [Gbyte/second]    19.71  (+52.21%)
L1 Hit Rate [%]                     98.92   (-0.14%)
L2 Hit Rate [%]                     67.55  (-15.63%)
```

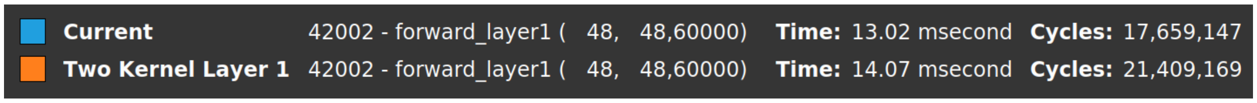
(b) Memory Throughput

Figure 5: Layer 2 Statistics

2.3 Tiled Convolution using Shared Memory

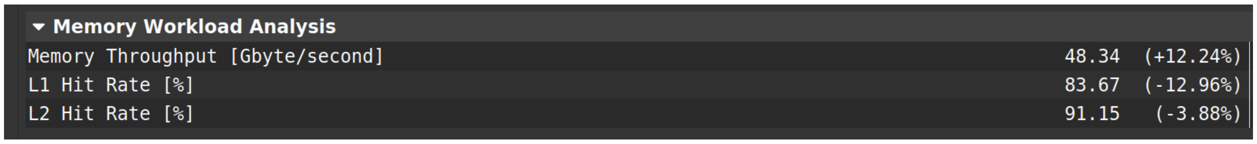
We can also optimize memory access pattern and reduce global memory access by caching elements that are highly reused into shared memory for each thread block. This optimization also implies that we break the input image into smaller tiles and each thread block processes a tile of elements at one time.

We chose loading strategy one, where all threads are utilized in multiple loop cycles to load the slightly bigger shared tile of data. The reason for not using strategy 2 is that we realize the finite computing resources is one of the biggest limiting factor of performance, so we don't want to waste the extra threads that are active only in loading stage but idle in computing stage. From the runtime report below, we can see that this optimization alone did not improve too much of the overall runtime, but even slightly hurt the performance. This is because of the bottleneck presented in reading the kernel data from global memory. Since the kernel is quite small, many threads are competing for the same addresses which causes a lot of serialization in the calculation loop. The tiled strategy, however, does introduce some overhead in the loading stage. Despite all of these, the overall memory throughput is still increased by this optimization.



🟦 Current	42002 - forward_layer1 (48, 48,60000)	Time: 13.02 msecond	Cycles: 17,659,147
🟧 Two Kernel Layer 1	42002 - forward_layer1 (48, 48,60000)	Time: 14.07 msecond	Cycles: 21,409,169

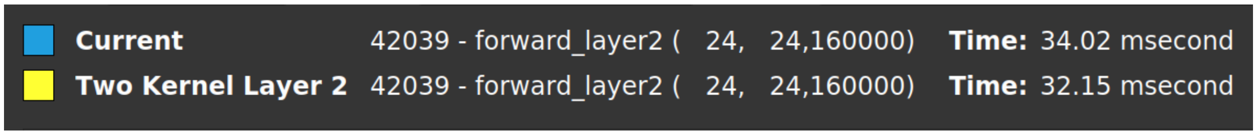
(a) Execution Time



```
▾ Memory Workload Analysis
Memory Throughput [Gbyte/second]        48.34  (+12.24%)
L1 Hit Rate [%]                         83.67  (-12.96%)
L2 Hit Rate [%]                         91.15   (-3.88%)
```

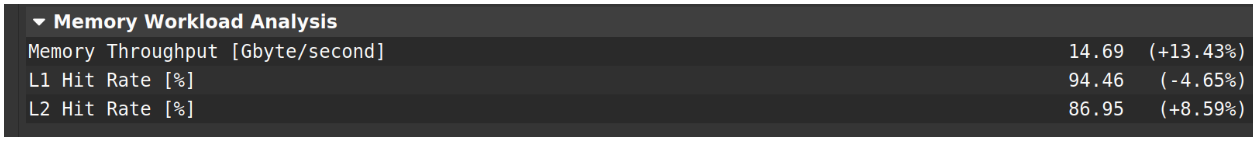
(b) Memory Throughput

Figure 6: Layer 1 Statistics



🟦 Current	42039 - forward_layer2 (24, 24,160000)	Time: 34.02 msecond
🟨 Two Kernel Layer 2	42039 - forward_layer2 (24, 24,160000)	Time: 32.15 msecond

(a) Execution Time



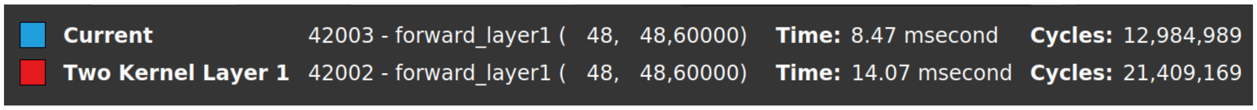
```
▾ Memory Workload Analysis
Memory Throughput [Gbyte/second]        14.69  (+13.43%)
L1 Hit Rate [%]                         94.46   (-4.65%)
L2 Hit Rate [%]                         86.95   (+8.59%)
```

(b) Memory Throughput

Figure 7: Layer 2 Statistics

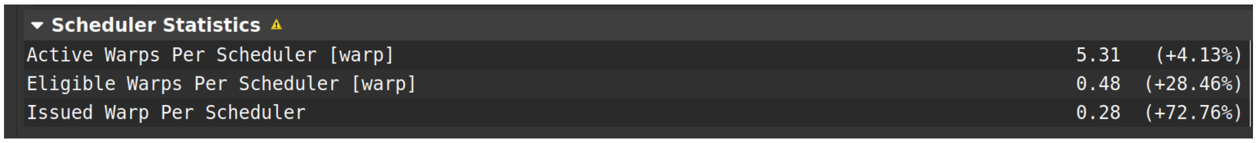
2.4 Combined Optimization

Now we combine all the optimizations to benchmark against our Two Kernel baseline implementation, we can see a great deal of improvement in runtime, memory throughput, and SM occupancy. From the statistics below we can see that a lot less threads are stalling for memory operations and more time are spent in floating point computation. The combination of constant cached kernel and tiled convolution completely takes away global memory access in the computing stage. And since shared tile loading is almost perfectly coealesed with this linearized strategy, the runtime is dramatically reduced.



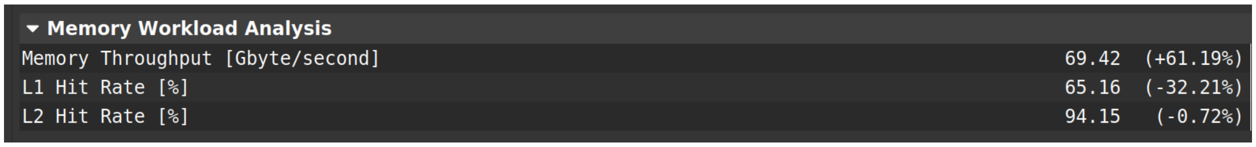
```
🟦 Current           42003 - forward_layer1 (  48,  48,60000)  Time: 8.47 msecond   Cycles: 12,984,989
🟥 Two Kernel Layer 1 42002 - forward_layer1 (  48,  48,60000)  Time: 14.07 msecond  Cycles: 21,409,169
```

(a) Execution Time



```
▾ Scheduler Statistics ⚠
Active Warps Per Scheduler [warp]                5.31   (+4.13%)
Eligible Warps Per Scheduler [warp]              0.48  (+28.46%)
Issued Warp Per Scheduler                        0.28  (+72.76%)
```

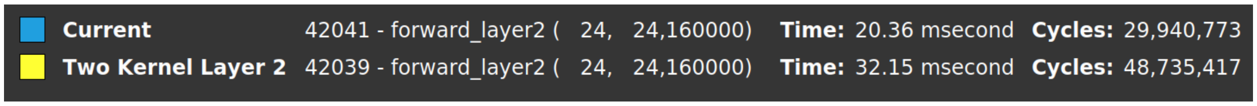
(b) Warp Statistics



```
▾ Memory Workload Analysis
Memory Throughput [Gbyte/second]                69.42  (+61.19%)
L1 Hit Rate [%]                                 65.16  (-32.21%)
L2 Hit Rate [%]                                 94.15   (-0.72%)
```

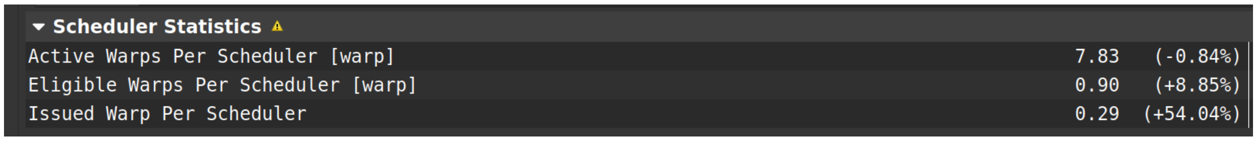
(c) Memory Throughput

Figure 8: Layer 1 Statistics



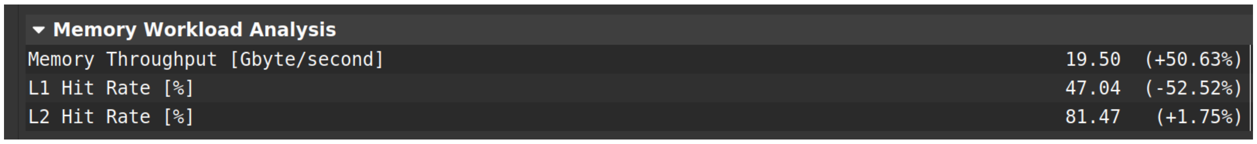
```
🟦 Current           42041 - forward_layer2 (  24,  24,160000)  Time: 20.36 msecond  Cycles: 29,940,773
🟨 Two Kernel Layer 2 42039 - forward_layer2 (  24,  24,160000)  Time: 32.15 msecond  Cycles: 48,735,417
```

(a) Execution Time



```
▾ Scheduler Statistics ⚠
Active Warps Per Scheduler [warp]                7.83   (-0.84%)
Eligible Warps Per Scheduler [warp]              0.90   (+8.85%)
Issued Warp Per Scheduler                        0.29  (+54.04%)
```

(b) Warp Statistics



```
▾ Memory Workload Analysis
Memory Throughput [Gbyte/second]                19.50  (+50.63%)
L1 Hit Rate [%]                                 47.04  (-52.52%)
L2 Hit Rate [%]                                 81.47   (+1.75%)
```

(c) Memory Throughput

Figure 9: Layer 2 Statistics