Jiyu Hu jiyuhu2

rai id: 5d97b1cc88a5ec28f9cb9464

Leihao Chen leihaoc2

rai id: 5d97b1b088a5ec28f9cb9430

Anthony Nguyen alnguyn2

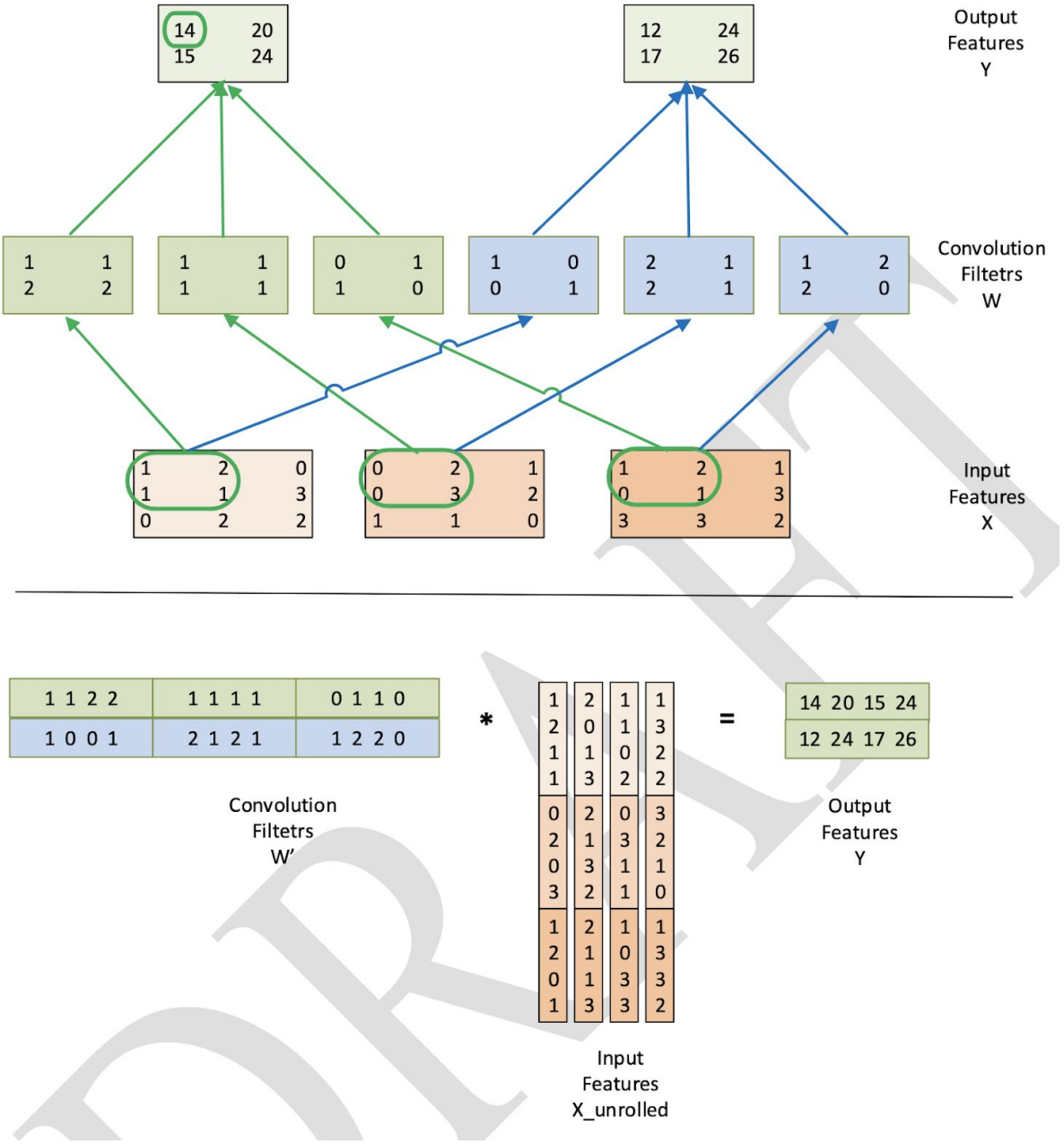
rai id: 5d97b1f288a5ec28f9cb94ab

Unroll and Matrix Multiplication

Description:

The optimization unrolls both input data matrices and weight matrices so that we can perform a simple shared memory matrix multiplication on the data to generate the output. The unrolling of weight matrices needs no additional execution since the row-major layout, we only need to change the indexing when we access it as unrolled matrices. For the input data, we run an additional kernel to perform the unroll. The unrolled matrix exceeds the assigned CUDA memory size when we perform op2 for 10000 dataset. So we partition the input data into small sections and deal with each section sequentially. This slows down the performance of this optimization for the last dataset.

This optimization is based on Chapter 16:



GPU Activities:

mxnet::op::matrixMultiplyShared(float*, float*, float*, int, int, int, int, int, int)

Time(%)	Time	Calls	Avg	Min	Max
34.31%	67.957ms	5	13.591ms	3.5026ms	17.328ms

mxnet::op::unroll_Kernel(int, int, int, int, int, float*, float*)

Time(%)	Time	Calls	Avg	Min	Max
22.23%	39.680ms	5	7.9360ms	2.1032ms	10.816ms

[CUDA memcpy HOTD]

Time(%)	Time	Calls	Avg	Min	Max
19.72%	35.198ms	20	1.7599ms	1.0880us	32.926ms

API Calls:

cudaStreamCreateWithFlags

Time(%)	Time	Calls	Avg	Min	Max
41.65%	3.37645s	22	153.48ms	15.188us	1.68743s

cudaMemGetInfo

Time(%)	Time	Calls	Avg	Min	Max
30.50%	2.47277s	24	103.03ms	72.464us	2.46793s

cudaFree

Time(%)	Time	Calls	Avg	Min	Max
20.58%	1.73656s	20	86.828ms	1.2690us	454.89ms

cudaMalloc

Time(%)	Time	Calls	Avg	Min	Max
4.40%	356.67ms	71	5.0236ms	7.7960us	121.33ms

Output of Rai running MXnet on the GPU:

*Running nvprof python m4.1.py with new-forward1.cuh
Op Time: 0.038605
Op Time: 0.419100
Correctness: 0.7653 **Model:** ece408

NVVP Analysis

Kernel Performance:

We utilize this optimization because we can transfer a time-complex matrix convolution operation into a faster matrix multiplication operation, but the runtime is about the same for the datasets. According to our understanding, the reason is mainly that the memory utilization grows a lot for this optimization compared to the basic approach. There are much more global and shared memory reads and writes, so even our code becomes more parallel, the overall performance is not better.

We uses two kernels for this optimization, one matrix unroll and one matrix multiplication. Matrix multiplication kernel takes more time to finish than the other.

Matrix Multiplication:

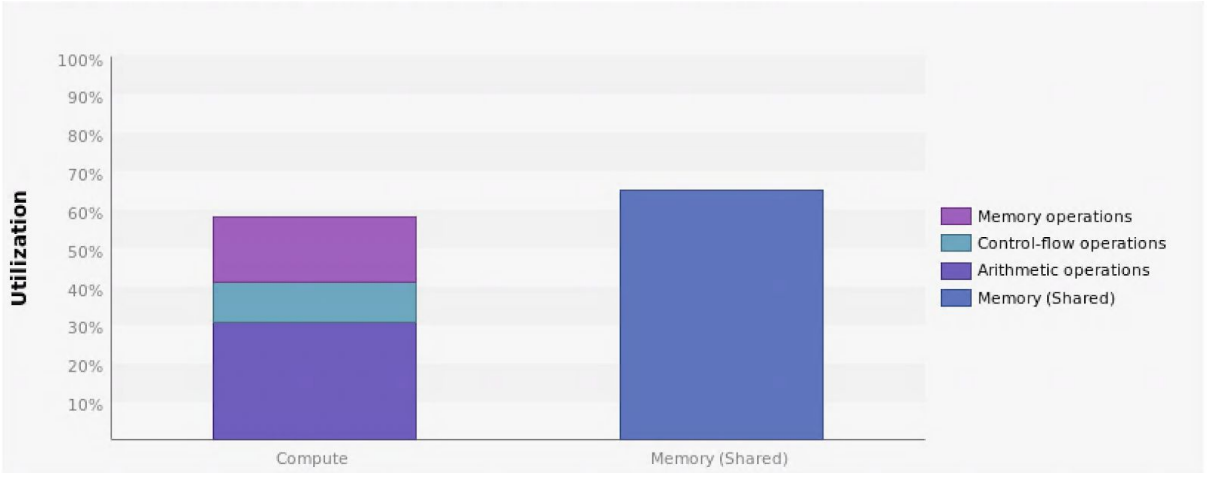


figure 1.1

This kernel is Bounded by Memory Bandwidth

As we can see from **figure 1.1**, which is kernel performance chart for matrix multiplication, the memory utilization is higher than compute utilization. This is obvious with our implementation since the matrix multiplication kernel accesses shared memory a lot.

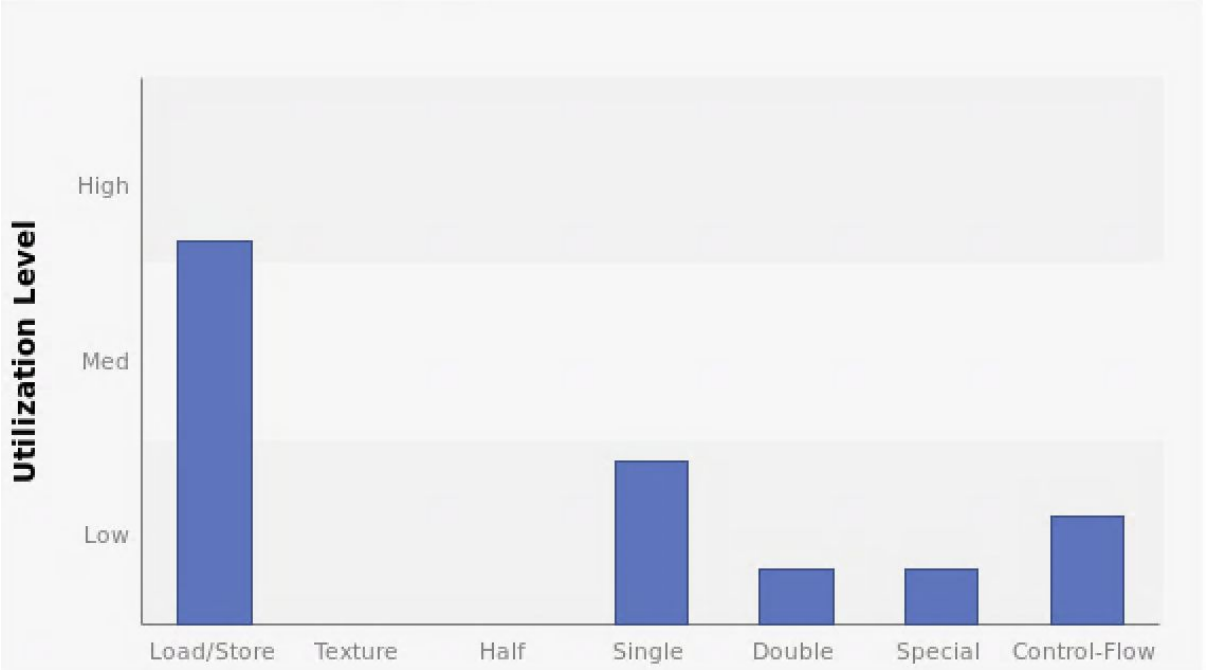


figure 1.2

Figure 1.2 is function unit utilization chart, this also indicates that the matrix multiplication kernel mainly spends time on memory load and store.

i Memory Bandwidth And Utilization

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. More...

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	300795071	7,643.729 GB/s	
Shared Stores	34423445	874.76 GB/s	
Shared Total	335218516	8,518.489 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	98525435	625.926 GB/s	
Writes	69084016	438.886 GB/s	
Total	167609451	1,064.812 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	149554945	950.113 GB/s	
Global Stores	69084000	438.886 GB/s	
Texture Reads	337500213	8,576.471 GB/s	
Unified Total	556139158	9,965.47 GB/s	Idle Low Medium High Max
Device Memory			
Reads	31549698	200.433 GB/s	
Writes	2579777	16.389 GB/s	
Total	34129475	216.822 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	31.764 kB/s	Idle Low Medium High Max

figure 1.3

Figure 1.3 is the memory usage of Matrix multiplication kernel. Both shared memory and unified cache memory is accessed frequently. We are utilizing the memory bandwidth pretty well.

Control Divergence:
No control divergence issue is detected in this kenel

Matrix Unroll:

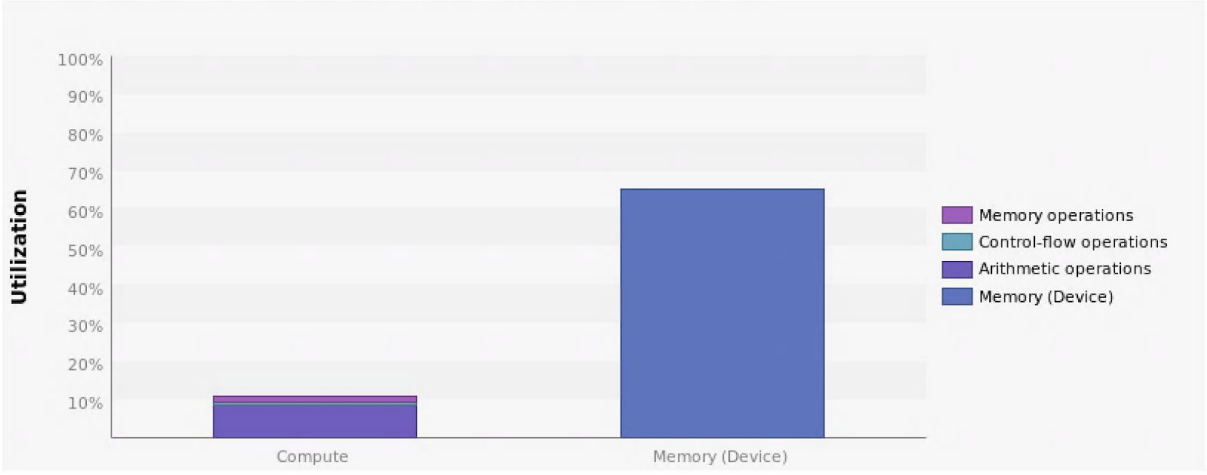


figure 1.4

This kernel is Bounded by Memory Bandwidth
As we conclude from **figure 1.4**, this kernel's utilization distribution is very unbalanced. This is consistent with our design since we unroll the matrix, which mainly moves data from one place in memory to another. For further improvement to our optimization, we can abandon this kernel and instead access data from original matrix by changing the indexing in matrix multiplication kernel.

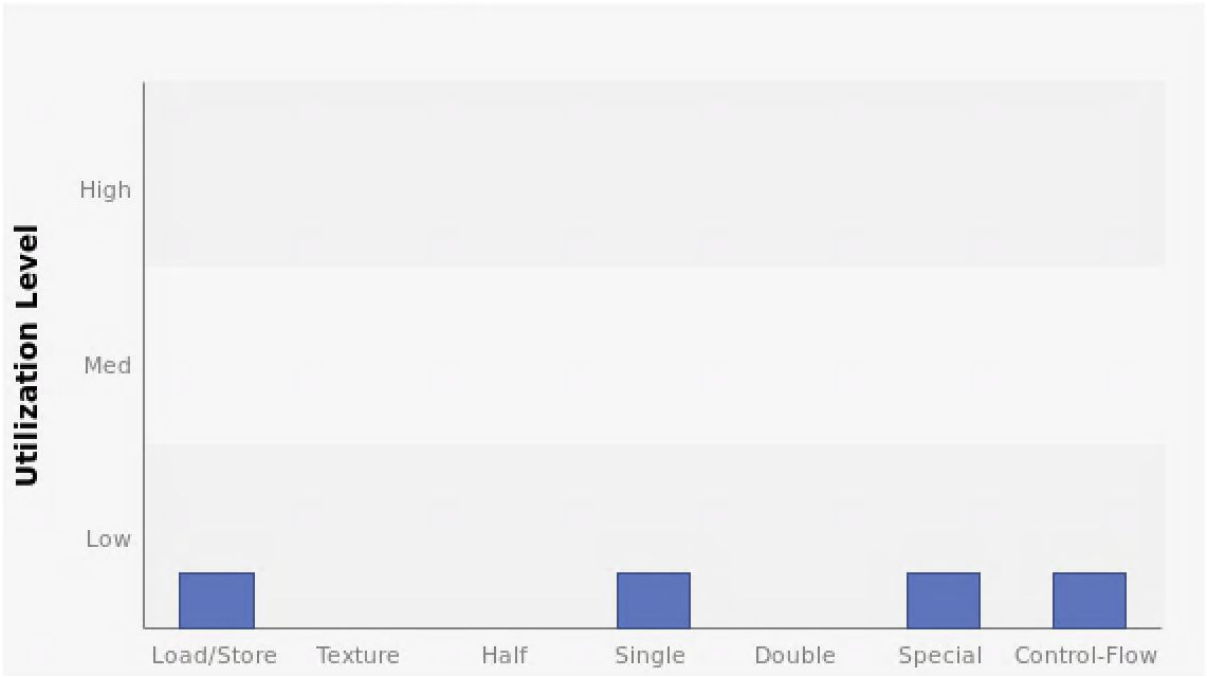


figure 1.5

Figure 1.5 shows that utilization level is low for all performances in the kernel. The same reason applies here.

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	Idle Low Medium High Max
L2 Cache			
Reads	5050372	56.362 GB/s	
Writes	38572516	430.464 GB/s	
Total	43622888	486.826 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	42220986	471.181 GB/s	
Global Stores	38572500	430.464 GB/s	
Texture Reads	10085356	450.205 GB/s	
Unified Total	90878842	1,351.85 GB/s	Idle Low Medium High Max
Device Memory			
Reads	3468952	38.713 GB/s	
Writes	34770087	388.03 GB/s	
Total	38239039	426.743 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	55.799 kB/s	Idle Low Medium High Max

figure 1.6

Memory access is exactly as expected: we perform a lot of global memory access in this kernel.

Control Divergence:
No control divergence issue has been found in this kernel

Shared Memory Convolution

Description:

Kernel performance in checkpoint 3 is bounded by memory. Since all data read and write are global, there is a high demand on memory bandwidth. This optimization use shared memory to reduce memory dependency on convolution kernel execution. Similar to lab: 3D convolution, we preload the data and weight matrices on shared memory.

GPU Activities:

mxnet::op::forward_kernel_shared(float*, float const *, float const *, int, int, int, int, int, int)

Time(%)	Time	Calls	Avg	Min	Max
70.57%	170.98ms	2	85.489ms	39.489ms	131.49ms

[CUDA memcpy HtoD]

Time(%)	Time	Calls	Avg	Min	Max
14.03%	33.995ms	20	1.6998ms	1.0880us	31.782ms

void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024, mshadow::expr::Plan<mshadow::Tensor<mshadow::gpu, int=4, float>, float>, mshadow::expr::Plan<mshadow::expr::BinaryMapExp<mshadow::op::mul, mshadow::expr::ScalarExp<float>, mshadow::Tensor<mshadow::gpu, int=4, float>, float, int=1>, float>>(mshadow::gpu, unsigned int, mshadow::Shape<int=2>, int=4, int)

Time(%)	Time	Calls	Avg	Min	Max
7.03%	17.040ms	2	8.5200ms	3.0484ms	13.992ms

API Calls:

cudaStreamCreateWithFlags

Time(%)	Time	Calls	Avg	Min	Max
40.84%	3.22616s	22	146.64ms	14.968us	1.71205s

cudaMemGetInfo

Time(%)	Time	Calls	Avg	Min	Max
32.04%	2.53134s	22	115.06ms	71.350us	2.52673s

Output of Rai running MXnet on the GPU:

*Running nvprof python m4.1.py with new-forward2.cuh
Op Time: 0.039538
Op Time: 0.131531
Correctness: 0.7653 **Model:** ece408

NVVP Analysis

Kernel Performance:

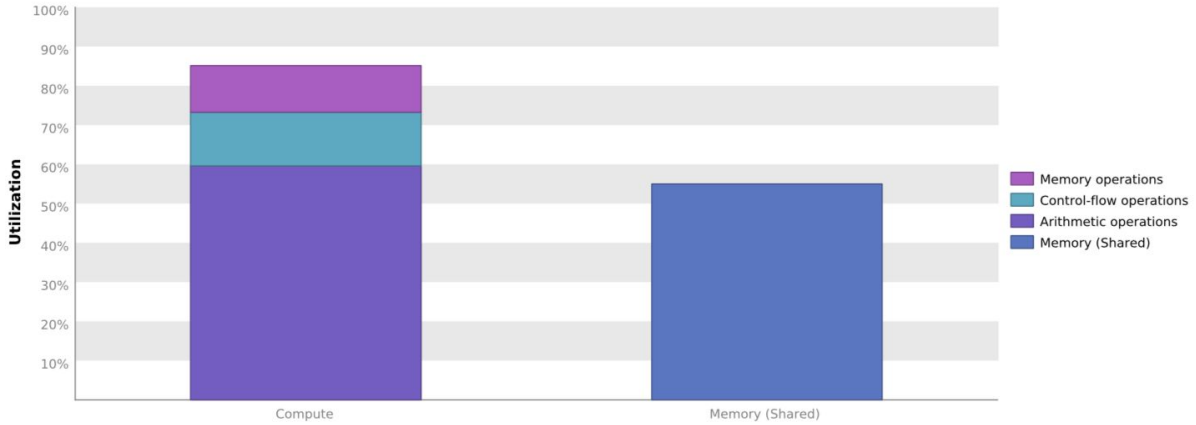


figure 2.1

As we can see from **figure 2.1**, which is kernel performance chart for shared memory convolution, the compute utilization is higher than memory utilization. It kernel is bounded by computation. As further proved in **figure 2.3** resource analysis. TITIAN V supports 65536 registers for each block and 32 blocks in a SM. The kernel used 10240 registers. Therefore, only 6 block is simultaneously executing on a SM.

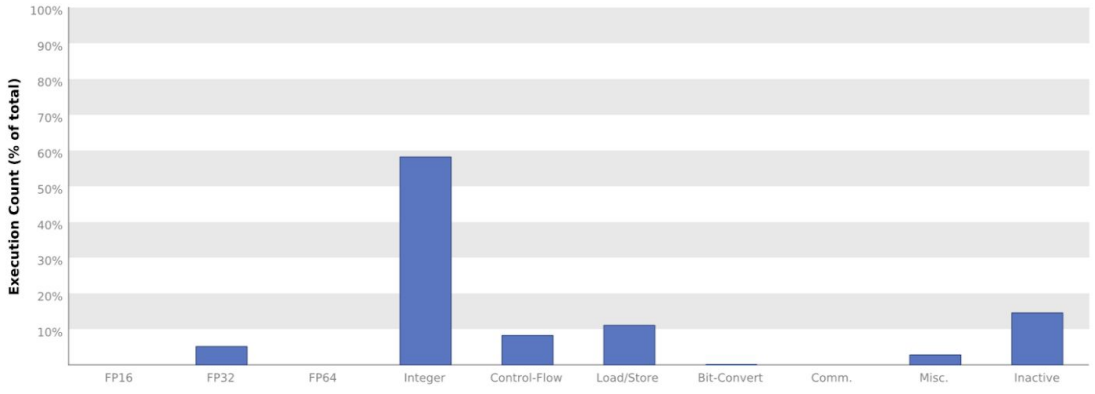


figure 2.2

Figure 1.2 is function unit utilization chart, this also indicates that the matrix multiplication kernel mainly spends time on memory load and store.

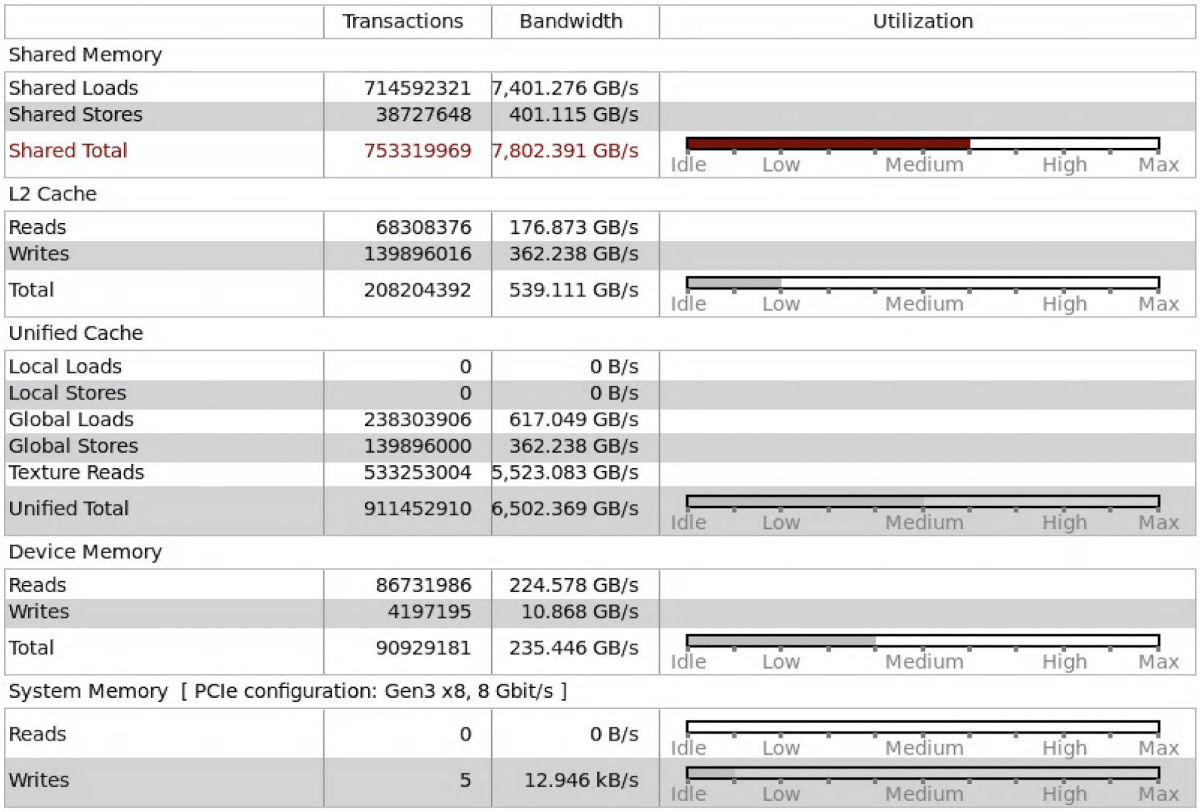
Line 43	Divergence = 37.5% [3456000 divergent executions out of 9216000 total executions]

```
if(t < C * W_unroll && b < B) // when loading shared memory
```

One major disadvantage of share memory method is the control divergence when transferring data from global memory to shared memory.

Line 47	Divergence = 50% [9216000 divergent executions out of 18432000 total executions]

```
for(p = 0; p < K; p++){
    for(q = 0; q < K; q++){ // for loop for convolution
```



	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	714592321	7,401.276 GB/s	
Shared Stores	38727648	401.115 GB/s	
Shared Total	753319969	7,802.391 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	68308376	176.873 GB/s	
Writes	139896016	362.238 GB/s	
Total	208204392	539.111 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	238303906	617.049 GB/s	
Global Stores	139896000	362.238 GB/s	
Texture Reads	533253004	5,523.083 GB/s	
Unified Total	911452910	6,502.369 GB/s	Idle Low Medium High Max
Device Memory			
Reads	86731986	224.578 GB/s	
Writes	4197195	10.868 GB/s	
Total	90929181	235.446 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	12.946 kB/s	Idle Low Medium High Max

figure 2.3

Figure 2.3 is the memory usage of Shared Memory Convolution kernel. Share Memory is properly utilized.

Constant Cache Optimization

Description:

The optimization was to use constant cache memory to load the weighted matrix, then perform the convolution by reading through the constant Cache. The reason for this was because the cache reduces memory traffic by being read-only thus making memory access much quicker. By loading our weighted matrix into a constant cache, it is expected that most of the reads will be accessed from the Cache during convolution execution thus improving performance.

GPU Activities:

mxnet::op::forward_kernel(float *, float const *, float const *, int, int, int, int, int, int);

Time(%)	Time	Calls	Avg	Min	Max
81.75%	41.815ms	2	20.957ms	4.065ms	37.849ms

[CUDA memcpy HOTD]

Time(%)	Time	Calls	Avg	Min	Max
11.41%	5.8487ms	20	293.44us	1.088us	3.557ms

Void mshadow::cuda::MapPlanLargeKernel<mshadow::sv::saveto, int=8, int=1024,.....)

Time(%)	Time	Calls	Avg	Min	Max
2.55%	1.3098ms	2	654.92us	282.43us	1.0274ms

Volta_sgemm_32x32_slicedlx4_tn

Time(%)	Time	Calls	Avg	Min	Max
1.69%	864.54us	2	432.27us	10.047us	854.49us

API Calls:

cudaStreamCreateWithFlags

Time(%)	Time	Calls	Avg	Min	Max
43.77%	3.20445s	22	145.66ms	14.999us	1.65972s

cudaMemGetInfo

Time(%)	Time	Calls	Avg	Min	Max
31.57%	2.31112s	22	105.05ms	93.281us	2.30600s

cudaFree

Time(%)	Time	Calls	Avg	Min	Max
21.96%	1.60793s	18	89.329ms	1.2550us	431.47ms

cudaMemcpy

Time(%)	Time	Calls	Avg	Min	Max
0.01%	497.0us	12	41.466us	6.405us	89.563us

cudaMemcpyToSymbol

Time(%)	Time	Calls	Avg	Min	Max
0.00%	81.937us	2	40.968us	38.458us	43.479us

Output of Rai running MXnet on the GPU:

*Running nvprof python m4.1.py with new_forward3.cuh
Op Time: 0.040644
Op Time: 0.354736
Correctness: 0.7653 **Model: ece408**
5.47user 2.33system 0:05.32elapsed 146%CPU (0avgtext+0avgdata 2975148maxresident) k

NVVP Analysis

Kernel Performance:

The optimization resulted in a similar performance as it were to the baseline kernel (milestone 3.1). The correctness was the same; however, the computational time was also the same. A major factor to this result is the control divergence seen in line 54 of the kernel. There was 50% control divergence, meaning half the threads in the same warp were executing the same branching behavior. Due to high divergence, the compute utilization of the kernel was low to the point where most of the GPU is idle, as seen in the far right bar in **figure 3.1.** We can see that the code reaches 20% execution count for Inactive instructions. Meaning that these threads did not execute any code due to control divergence. As we can see from **figure 3.2**, the kernel is also bounded by instruction and memory latency. This shows that the kernel is not fully utilizing the TITAN V to achieve the optimal throughput.

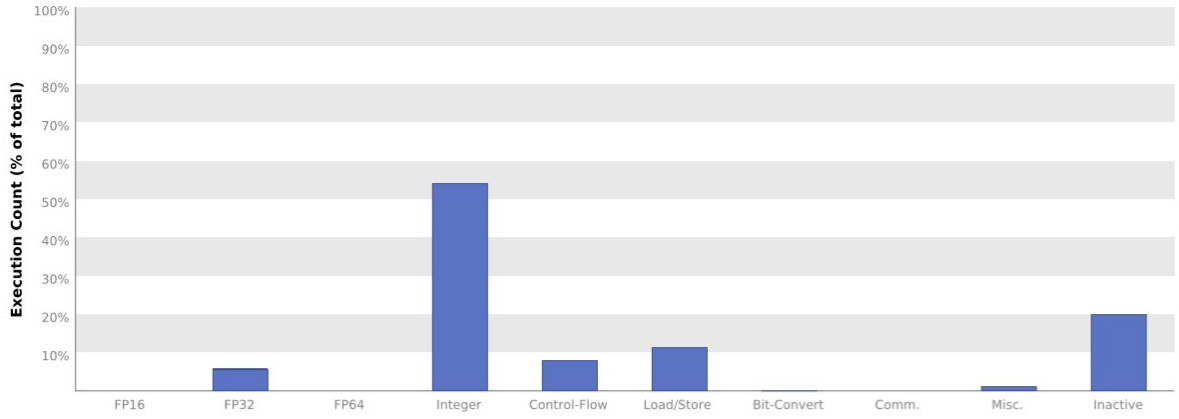


Figure 3.1: Constant Cache Execution Count

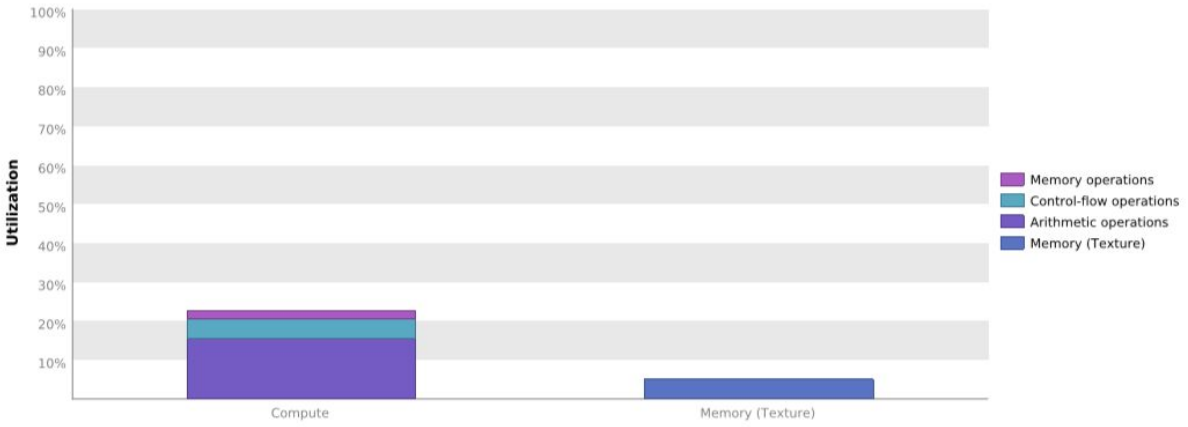


Figure 3.2: Kernel Computational Performance

Reason for optimization:

The reason for trying this optimization was because the constant cache is faster than global memory. By placing the weighted matrix into constant memory, it can be accessed quickly. This is ideal since the weighted matrix is used to calculate every value of the output, so by making these reads quicker, it should cause the overall kernel to be quicker. We believe that the idea is correct; however, the implementation will need to work on optimizing the number of blocks being executed by the kernel to reduce the memory and instruction latency. Along with that, the 50% control divergence causes drastic slow down during execution. We believe by optimizing the control flow branches and finding the optimal grid and block sizes, this optimization can be quicker.

Summary of GPU resource utilization:

	Matrix Unroll	Matrix Multiplication	Shared Memory Convolution	Constant Weight Matrix Convolution	Base line kernel (3.1)
Duration	2.86742 ms	5.03704 ms	12.4 ms	34.68ms	35.0527ms
Grid Size	[158, 1000, 1]	[53, 2, 1000]	[1000, 24, 4]	[24,4,1]	[24,4,1]
Block Size	[64, 1, 1]	[16, 16, 1]	[16, 16, 1]	[16,16,1]	[16,16,1]
Global Load Efficiency	74.7%	74.1%	22.2%	67.5%	58.6%
Global Store Efficiency	81.8%	69.4%	21.6%	67.4%	67.4%
Shared Efficiency	n/a	41.8%	33.6%	n/a	n/a
Local Memory Overhead	n/a	38.1%	66.6%	n/a	n/a
Warp Execution Efficiency	99.8%	99.5%	92.8%	87.6%	87.6%
Not-Predicted-Off Warp Execution Efficiency	92.1%	93.5%	85.5%	80.1%	80.4%
Achieved Occupancy	92.1%	99.2%	74.6%	15%	14.1%