Report for Milestone 1

Team name: teamparallel
UIN: 668641631, 675929143,
Team member names: Dong Liu, Hengzhe Ding, Yijian Duan
Net ID: dongl3, hengzhe2, yijiand2

Include a list of all kernels that collectively consume more than 90% of the program time：

Time	Name
36.73%	[CUDA memcpy HtoD]
22.71%	volta_scudnn_128x32_relu_interior_nn_v1
20.84%	void cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>(int, int, int, float const *, int, float*, cudnn::detail::implicit_convolve_sgemm<float, float, int=1024, int=5, int=5, int=3, int=3, int=3, int=1, bool=1, bool=0, bool=1>*, kernel_conv_params, int, float, float, int, float, float, int, int)
7.40%	volta_sgemm_128x128_tn
7.27%	void cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>(cudnnTensorStruct, float const *, cudnn::detail::activation_fw_4d_kernel<float, float, int=128, int=1, int=4, cudnn::detail::tanh_func<float>>, cudnnTensorStruct*, float, cudnnTensorStruct*, int, cudnnTensorStruct*)

Include a list of all CUDA API calls that collectively consume more than 90% of the program time.

Time	Name
39.72%	cudaStreamCreateWithFlags
34.14%	cudaMemGetInfo
21.63%	cudaFree

Include an explanation of the difference between kernels and API calls:
Kernels are something programmers write to reach the function they want.
API calls are functions NVIDIA write for programmers to simplify some basic use in cuda environment.

Show output of rai running MXNet on the CPU:

```
* Running /usr/bin/time python m1.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
21.08user 6.18system 0:14.36elapsed 189%CPU (0avgtext+0avgdata 5955796maxre
sident)k
0inputs+2856outputs (0major+1583127minor)pagefaults 0swaps
```

List program run time: 14.36s

Show output of rai running MXNet on the GPU:

```
* Running /usr/bin/time python m1.2.py
Loading fashion-mnist data... done
Loading model... done
New Inference
EvalMetric: {'accuracy': 0.8177}
4.44user 2.55system 0:04.81elapsed 145%CPU (0avgtext+0avgdata 2846644maxr
esident)k
0inputs+1712outputs (0major+704905minor)pagefaults 0swaps
```

List program run time: 4.81s

Report for Milestone 2

Team name: teamparallel
UIN: 668641631, 675929143,
Team member names: Dong Liu, Hengzhe Ding, Yijian Duan
Net ID: dongl3, hengzhe2, yijiand2

```
* Running /usr/bin/time python m2.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 21.373073
Op Time: 101.454568
Correctness: 0.8171 Model: ece408
132.98user 4.53system 2:07.14elapsed 108%CPU (0avgtext+0avgdata 5952360maxreside
nt)k
0inputs+2856outputs (0major+2266811minor)pagefaults 0swaps
```

Whole program execution time: 2:07.14

Op time: 21.373073s + 101.454568s = 122.827641s

Report for Milestone 3

Team name: teamparallel
UIN: 668641631, 675929143,
Team member names: Dong Liu, Hengzhe Ding, Yijian Duan
Net ID: dongl3, hengzhe2, yijiand2

Correctness and timing with 3 different dataset sizes:

Datasize = 100:

```
* Running /usr/bin/time python m3.1.py 100
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.000447
Op Time: 0.001608
Correctness: 0.85 Model: ece408
```

Datasize = 1000:

```
* Running /usr/bin/time python m3.1.py 1000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.004266
Op Time: 0.016017
Correctness: 0.827 Model: ece408
```

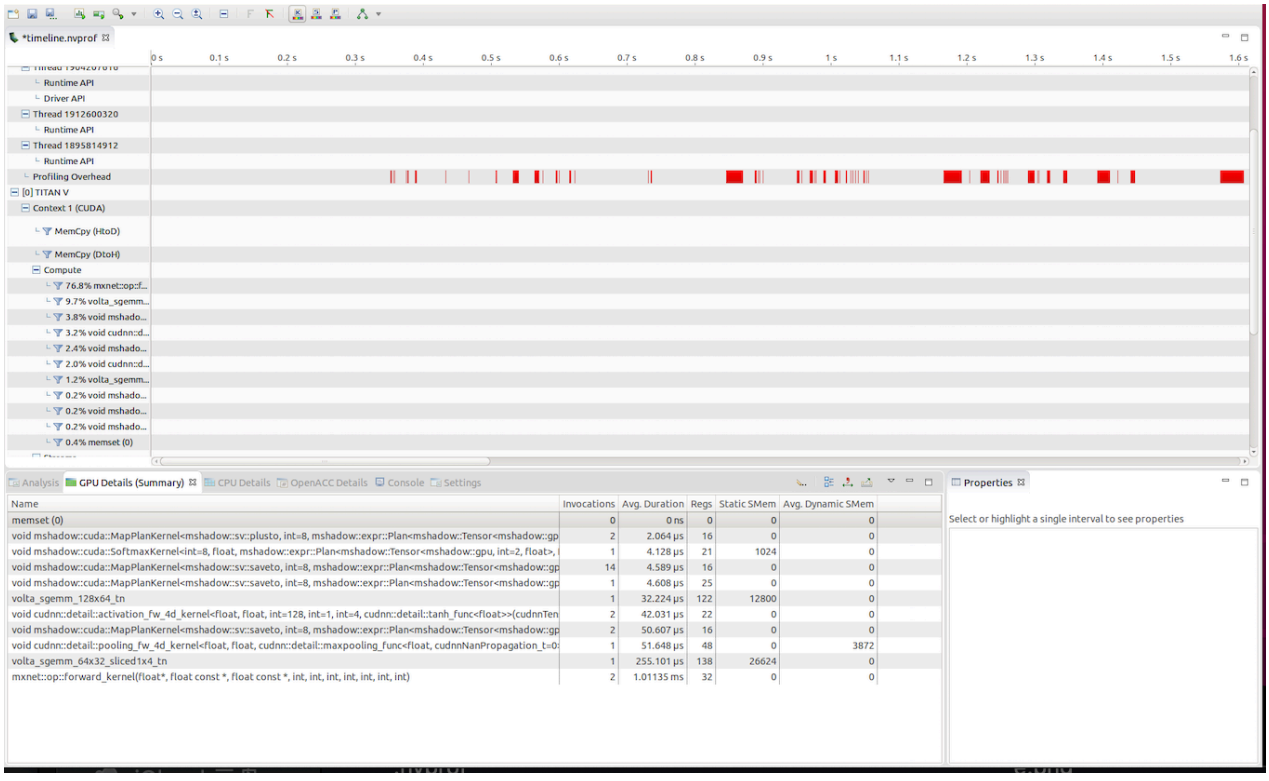
Datasize = 10000:

```
* Running /usr/bin/time python m3.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.042506
Op Time: 0.150724
Correctness: 0.8171 Model: ece408
```

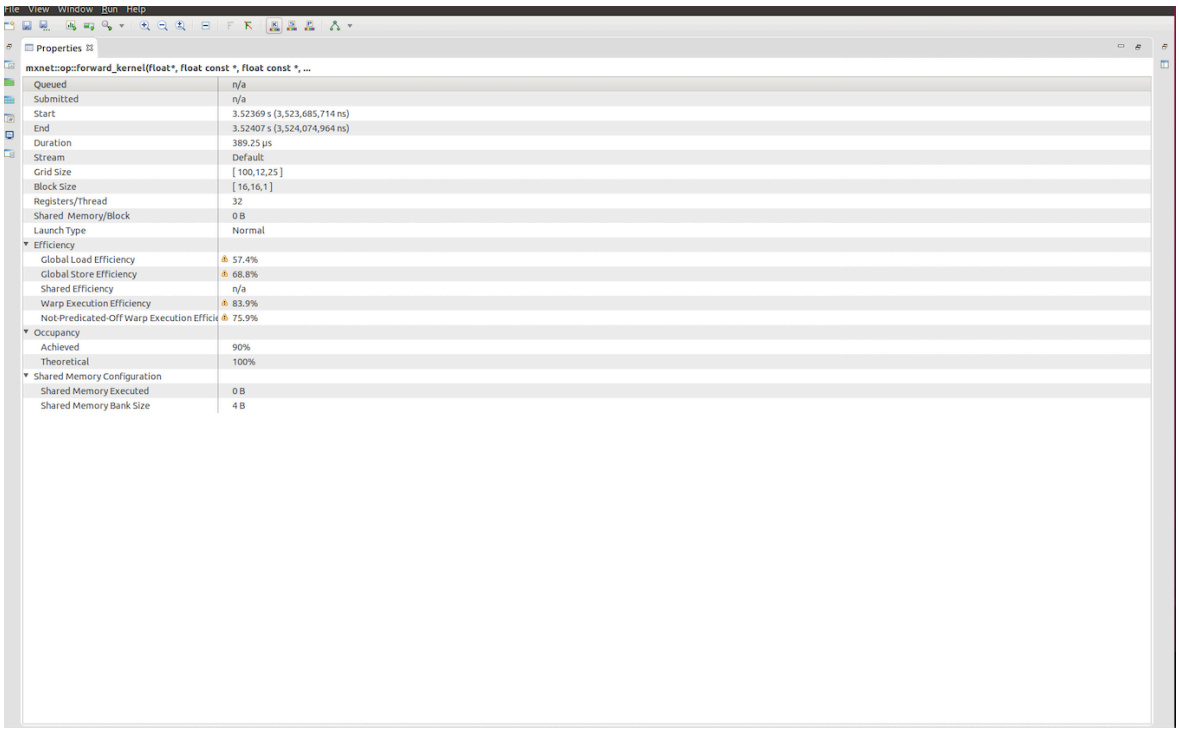
Demonstrate nvprof profiling the execution:

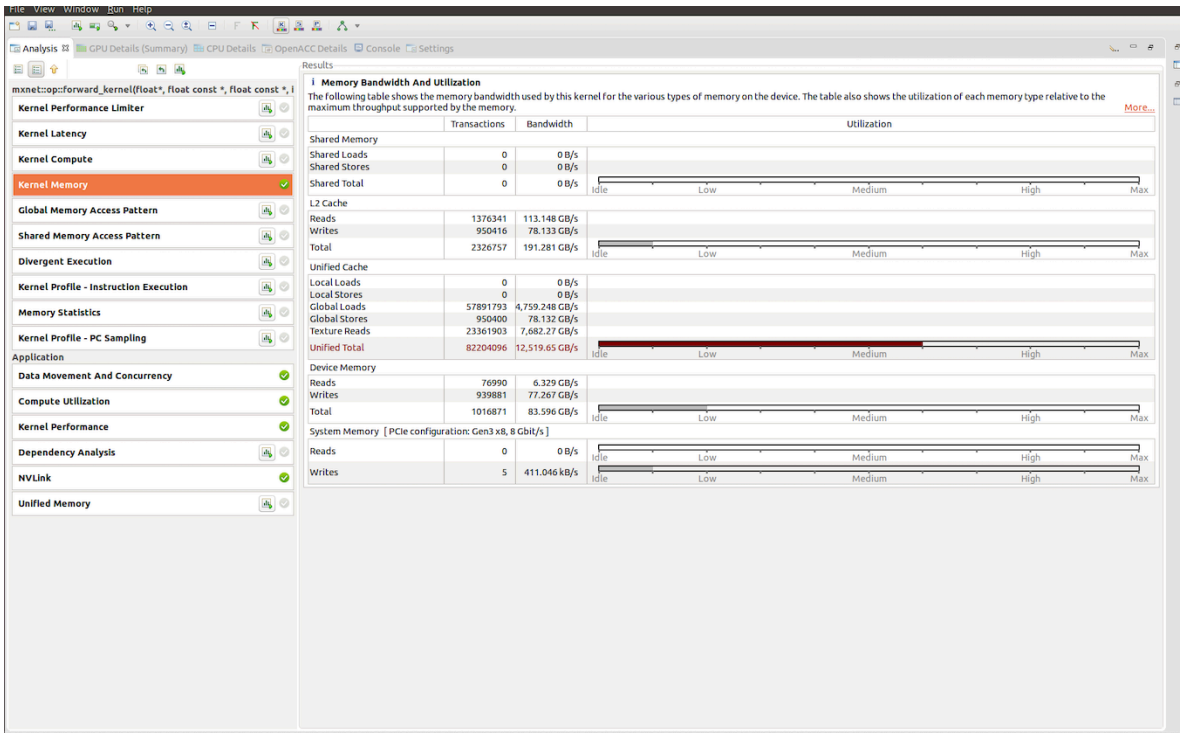
NVVP results:

timeline.nvprof:

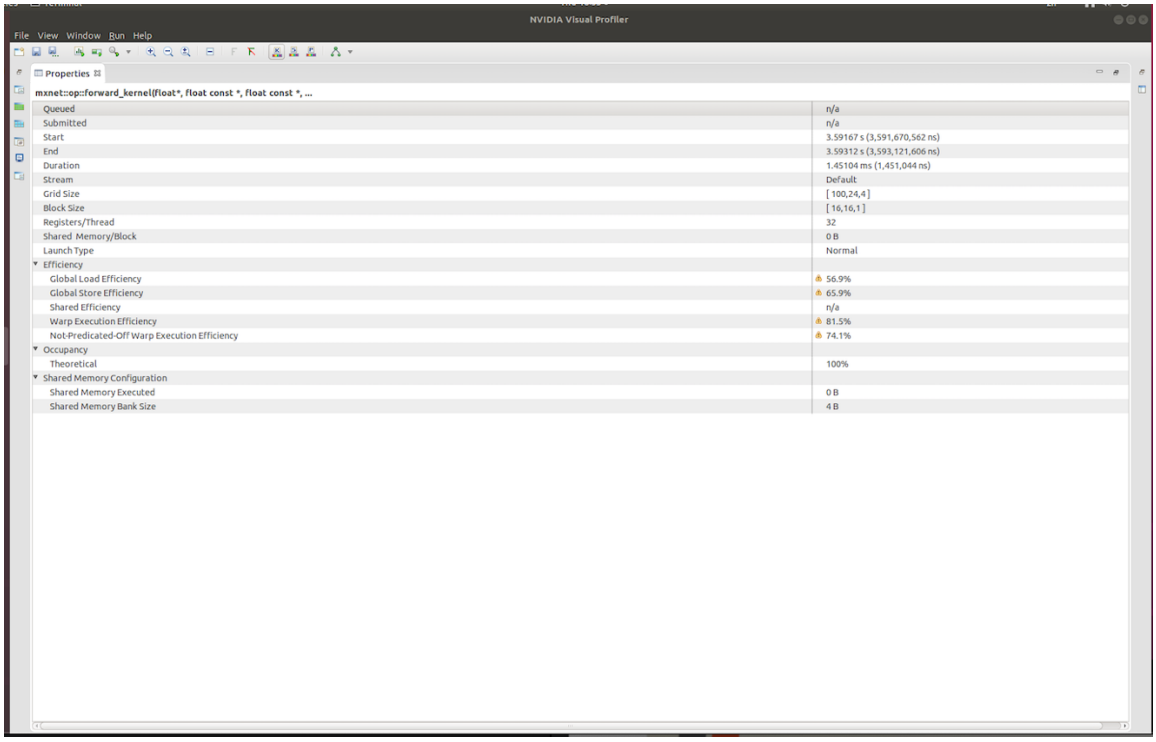


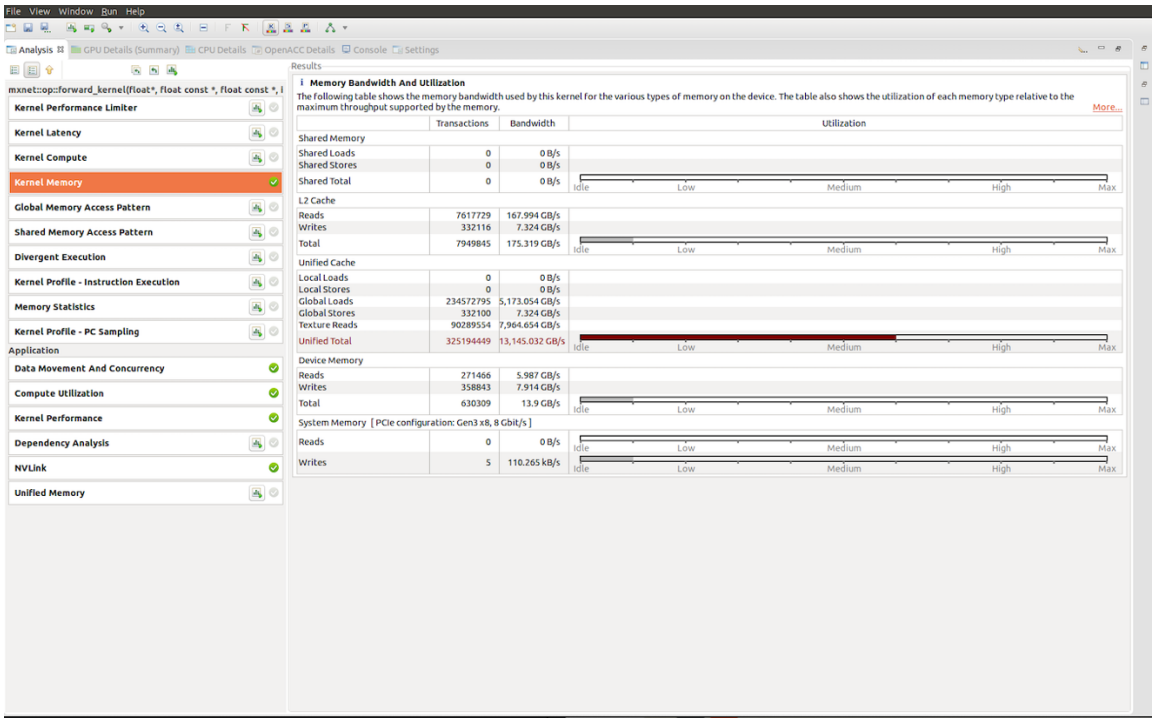
forward1_analysis.nvprof:





forward2_anaylsis.nvprof:





Analysis:

Shared memory:

Since we did not use shared memory, so in *analysis.nvprof, shared memory/block is 0 B.

Warp execution:

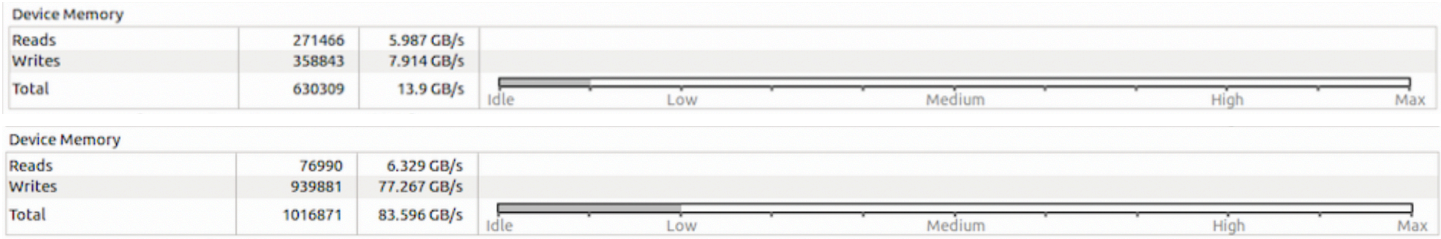
From the screenshots above we can see that the forward kernel’ s warp execution is around 80 %, which is not 100% because of the divergence in the kernel. This will lead to low efficiency of execution. Therefore, in our next optimization process, we need to improve warp execution efficiency and make full use of each warp in the kernel;

Also, in our kernel the compute utilization is low since there are a lot of control divergence. By reducing the control divergence in our kernel, we could get higher performance during the kernel' s execution.

In addition, there are for-loop add in our kernel which do harm to our kernel performance and efficiency. In our future optimization, we will try a parallel scan way to solve this problem in order to improve the compute utilization.

Memory bandwidth:

Since we did not use shared memory, all data in kernel are read/written directly from/to global memory, making utilization of bandwidth low due to large latency.



Report for Milestone 4

Team name: teamparallel
UIN: 668641631, 675929143,
Team member names: Dong Liu, Hengzhe Ding, Yijian Duan
Net ID: dongl3, hengzhe2, yijiand2

We complete 4 optimizations in this milestone:

1. Unroll/shared-memory matrix multiplication (Yijian Duan)
2. Shared memory convolution (Hengzhe Ding, Dong Liu)
3. Weight matrix (kernel values) in constant memory (Hengzhe Ding)
4. Tuning with restrict, loop unrolling (Hengzhe Ding, Dong Liu)

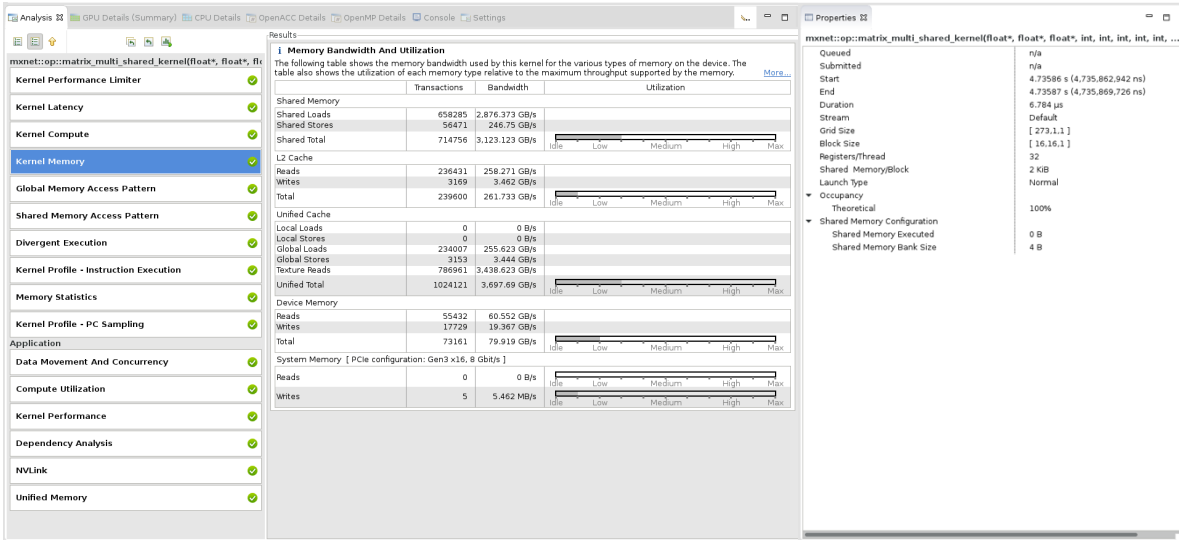
Since 1 (Unroll/shared-memory matrix multiplication) and 2 (Shared memory convolution) are more like approaches than optimizations, we implemented both and chose a better one.

We first implemented 1 (Unroll/shared-memory matrix multiplication), which include two kernels, unroll_kernel and matrix_multiplication. The unroll_kernel is to, same as discussed in text book, map data from input matrix into an unrolled version. The matrix_multiplication kernel is to perform matrix multiplication of two input matrixes and write result to one output matrix, same as MP3 in this course. Shared memory is used in matrix_multiplication kernel to optimize memory bandwidth. The test result is shown below, note that operation time is measured based on 10000 batches and nvprof results are measured based on 100 batches.

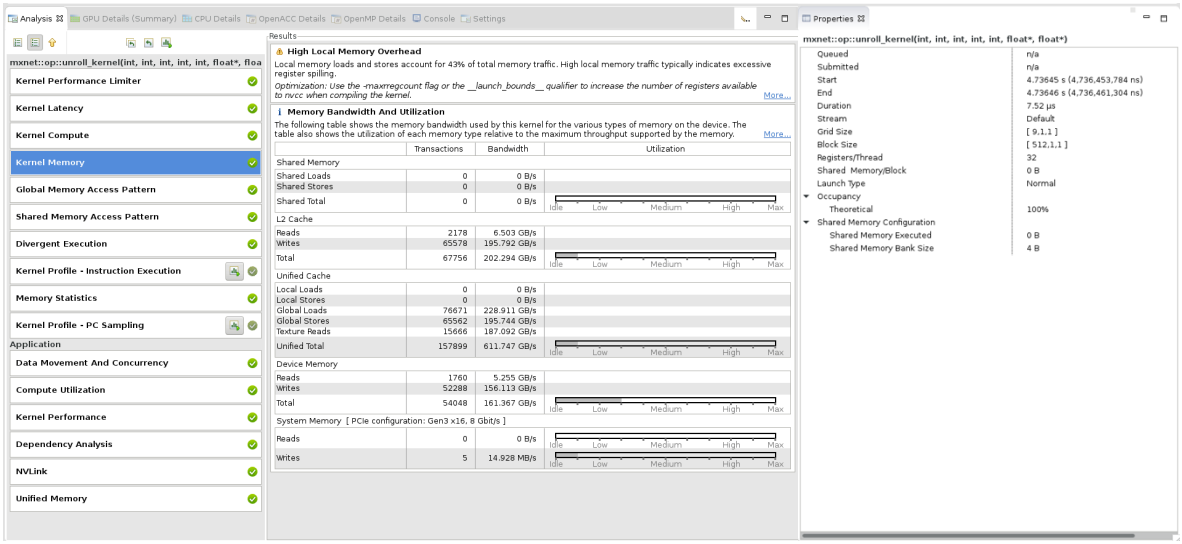
Operation Time:

```
* Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.156181
Op Time: 0.321393
Correctness: 0.8171 Model: ece408
4.61user 2.86system 0:05.13elapsed 145%CPU (0avgtext+0avgdata 2848664maxresident)k
0inputs+4648outputs (0major+708008minor)pagefaults 0swaps
```

Matrix multiplication kernel:



Matrix unroll kernel:

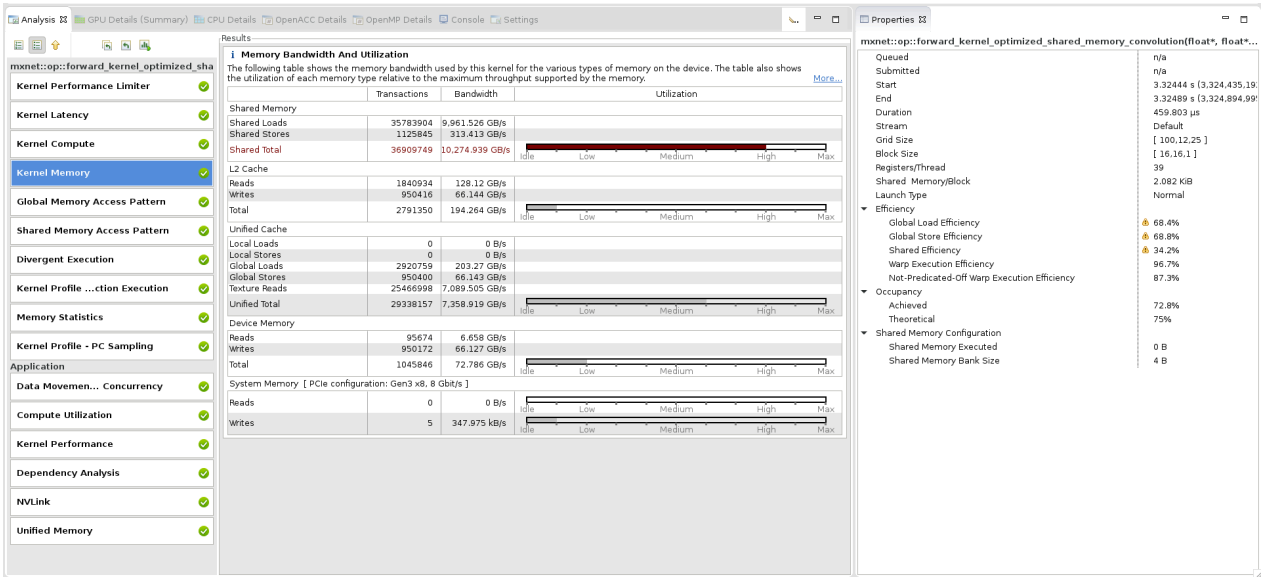


We then implement 2 (shared memory convolution), which is an approach that enables shared memory in previous GPU implementation. It first loads data from global memory to shared memory, then perform convolution using data in shared memory. We can see there is a huge improvement in Warp Execution Efficiency (from 81.5% to 96.7%), compared with GPU implementation of previous milestone and shared memory utilization is also improved a lot. The test result is shown below, note that operation time is measured based on 10000 batches and nvprof results are measured based on 100 batches.

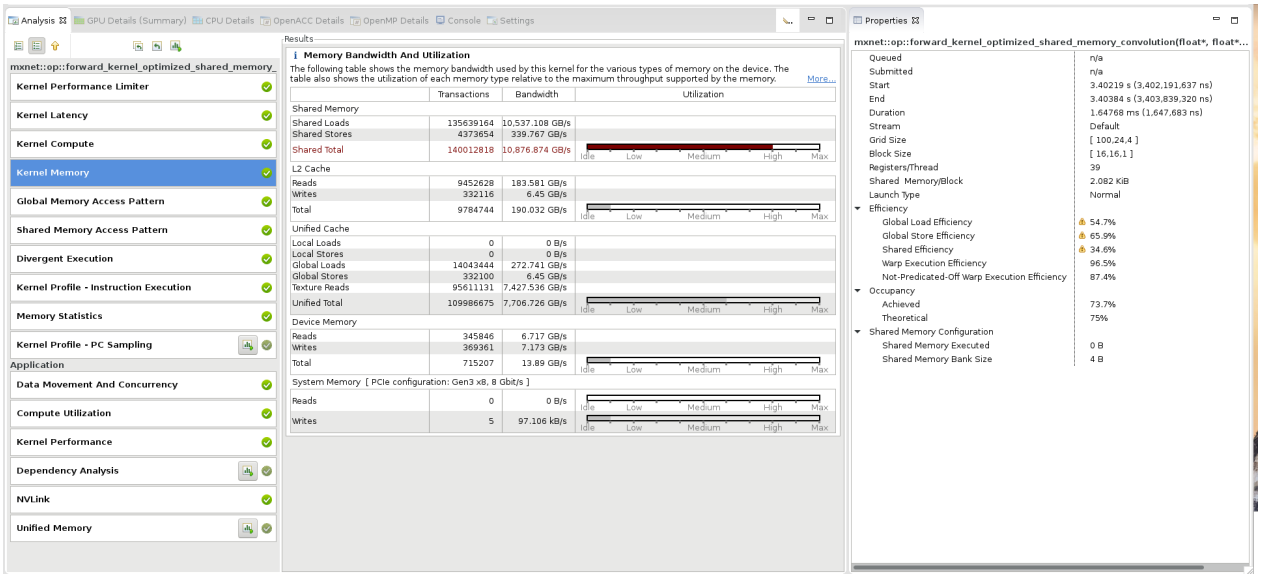
Operation time:

```
* Running /usr/bin/time -f "%Uuser %Ssystem %eelapsed" python /eval-scripts/m4.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.051793
Op Time: 0.176992
Correctness: 0.8171 Model: ece408
4.41user 2.39system 4.81elapsed
```

First layer:



Second layer:

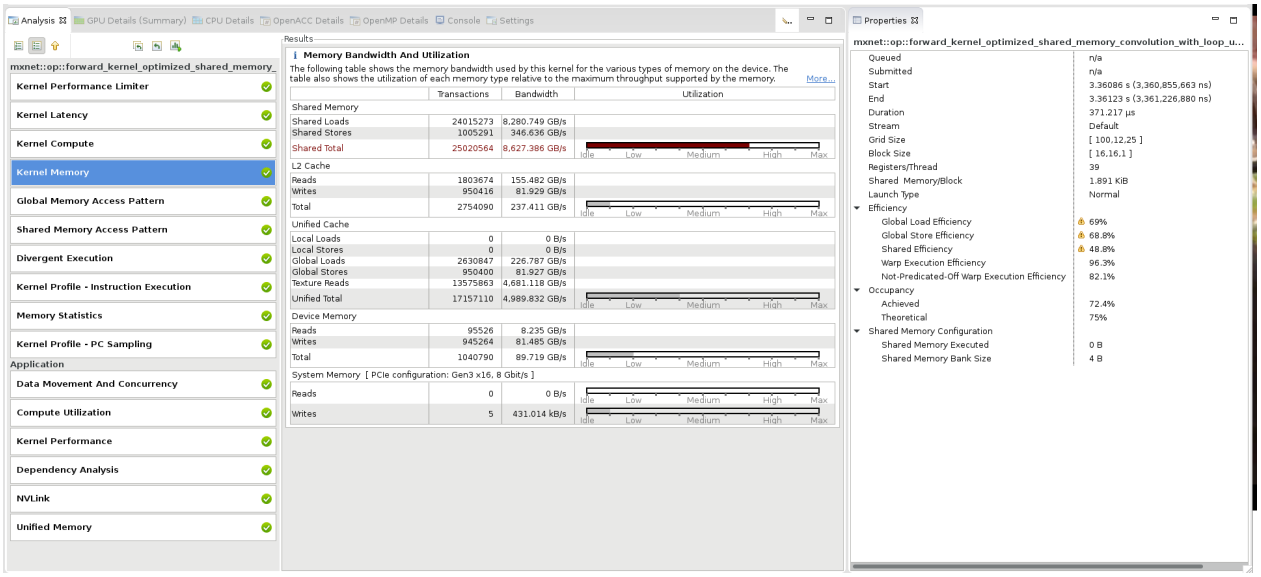


After comparison, we find 2 is better than 1, in terms of operating time. So we choose 2 to do some further optimizations, integrating 3 and 4 (constant memory and tuning restrict, loop unrolling). The test result is shown below, note that operation time is measured based on 10000 batches and nvprof results are measured based on 100 batches.

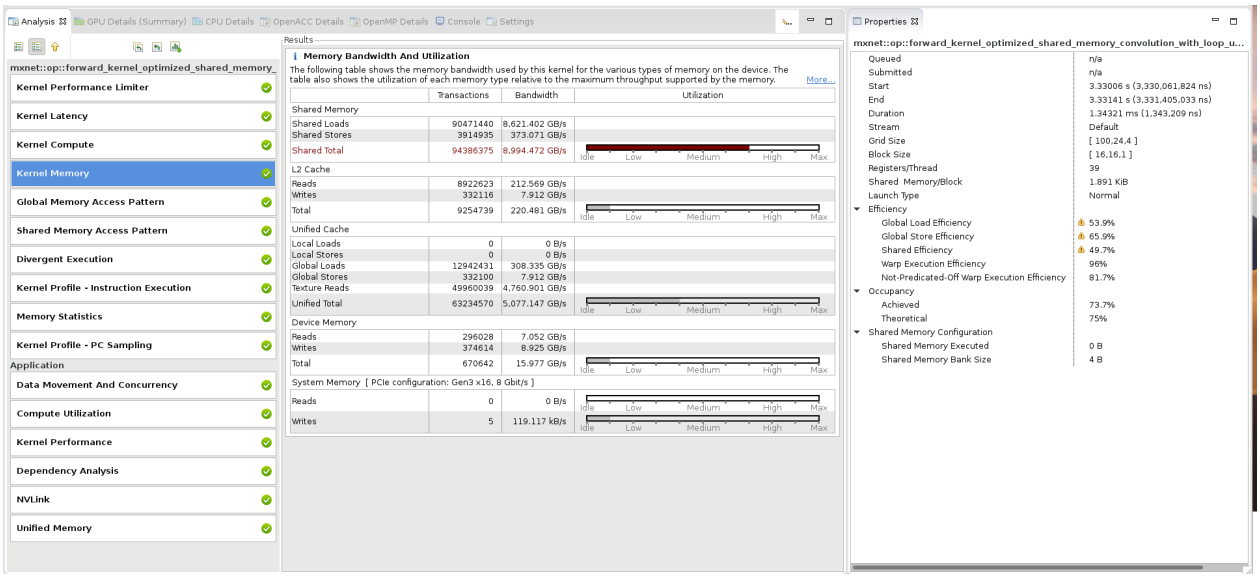
Operation time:

```
* Running /usr/bin/time -f "%Uuser %Ssystem %eelapsed" python /eval-scripts/m4.1.py
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.043898
Op Time: 0.140297
Correctness: 0.8171 Model: ece408
4.40user 2.85system 4.77elapsed
```

First layer:



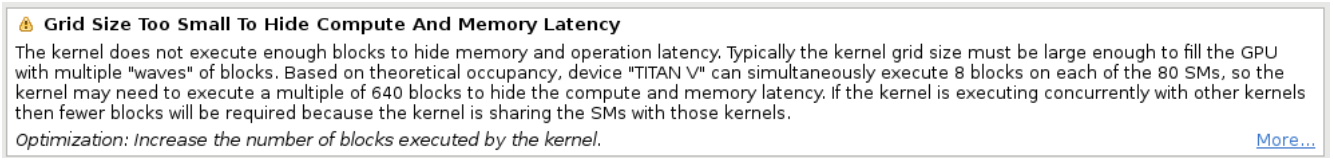
Second layer:



Analysis:

We find that both 1 and 2 are worse than unoptimized version of GPU implementation, in terms of operating time. There are various reasons for this phenomenon.

For 1 (Unroll/shared-memory matrix multiplication), we need to invoke two kernels every time in batch, in test case, we invoke kernel 200 times, which means the number of data processed by each kernel is small. This leads to a warning in Nvidia Visual Profiler called “Grid size too small to hide compute and memory latency”, shown in image below. If we use larger data, say, images with more pixels, or choosing a smaller thread block, using this approach may be a better choice. However, implementing a too small thread block will have smaller shared memory, therefore lead to worse performance.



For 2 (shared memory convolution), the reason lies in that, according to analysis, shared memory efficiency is low, we think it is because we load shared memory from global memory, which is very costly. After loading weight matrix from constant memory, Shared Efficiency get improved from around 35% to around 50%.

Adding optimization 3, 4 (constant memory and restrict, loop unrolling) is effective on kernel performances.

For 3 (constant memory), the optimization is effective because reading/writing from/to constant memory is 1000X faster than reading/writing from/to global memory. Since CUDA constant memory is limited to 64KB, which is 16000 floats in this case, constant memory array size should be within this range.

For 4 (tuning with restrict, loop unrolling), this optimization is done by adding two keywords, “__restrict__” and “#pragma unroll”. The “__restrict__” keyword tells complier that pointers with this keyword do not have memory overlapping, meaning changing one pointer’s value does not affect other pointers, so that some unnecessary assembly instructions can be saved, making operation time faster. The “#pragma unroll” keyword tells complier to expand for-loop to hard-coded version when executing.

Report for Final

Team name: teamparallel
UIN: 668641631, 675929143,
Team member names: Dong Liu, Hengzhe Ding, Yijian Duan
Net ID: dongl3, hengzhe2, yijiand2

We complete 4 more optimizations in final stage:

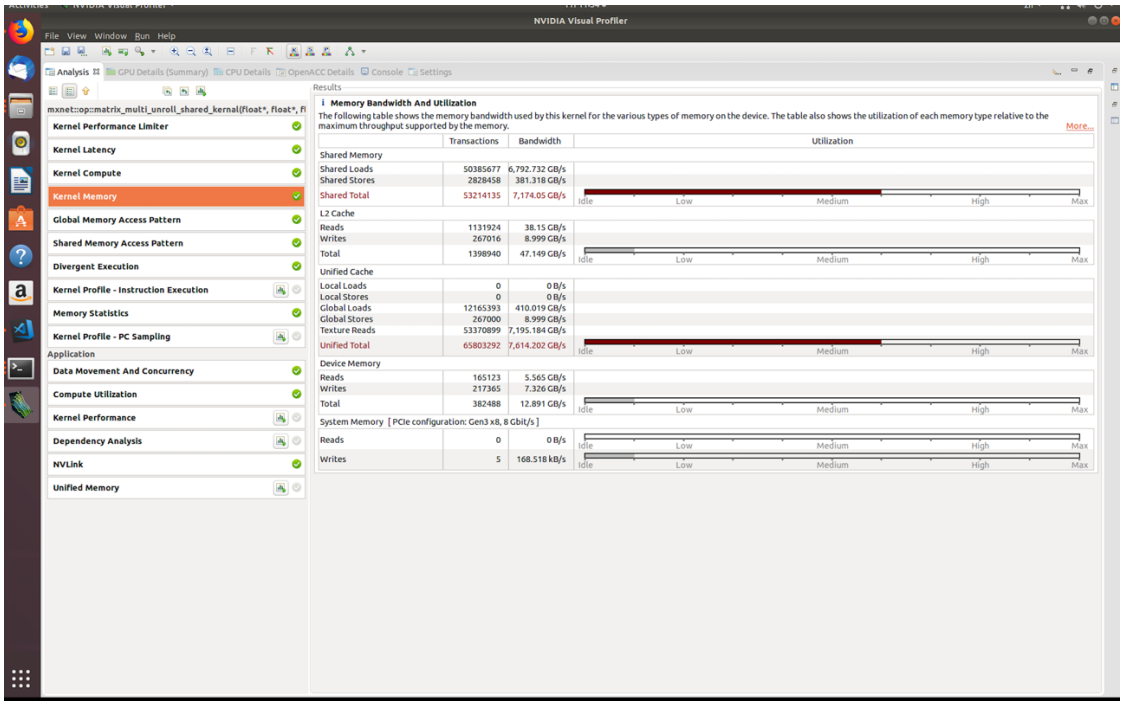
5. Kernel fusion for unrolling and matrix-multiplication (Yijian Duan)
6. Input channel reduction: tree (Dong Liu)
7. Input channel reduction: atomics (Hengzhe Ding)
8. Sweeping various parameters to find best values (block sizes, thread coarsening) (Yijian Duan, Hengzhe Ding)

We first implemented 5 (Kernel fusion for unrolling and matrix-multiplication), which is a direct optimization to optimization 1 (Unroll/shared-memory matrix multiplication). During the experiment, the kernel is implemented by doing unrolling operations and took the advantage of tiled matrix multiplication. We assume that the width of input feature maps is a multiple of tiled width and then reading them into tiled matrix using shared memory. This can improve the efficiency of our kernel and also avoid the slow global memory access. Also, in our tiled matrix multiplication, the loads to both the subtile M and subtile N is coalesced. Therefore, the DRAM bandwidth utilization is further improved. In our optimization, such kernel fusion did well in improving the efficiency. The test result is shown below, note that operation time is measured based on 10000 batches and nvprof results are measured based on 100 batches.

Operation time:

```
* Running /usr/bin/time python m4.1.py 10000
Loading fashion-mnist data... done
Loading model... done
New Inference
Op Time: 0.053622
Op Time: 0.097029
Correctness: 0.8171 Model: ece408
```

Kernel analysis:

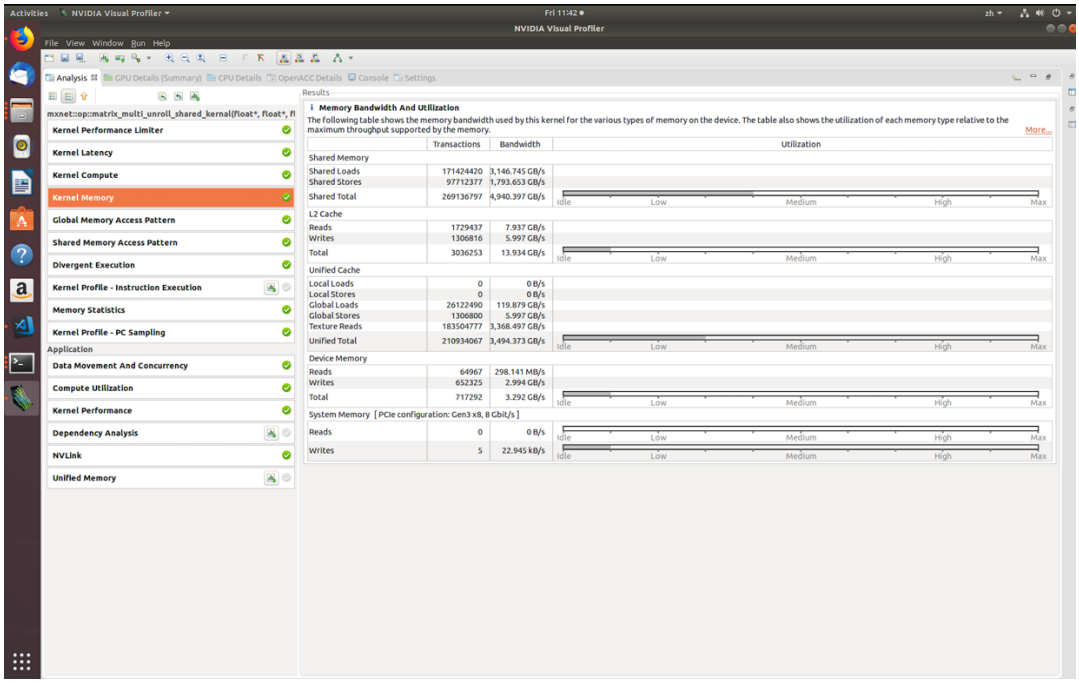


We then implemented 6 (Input channel reduction: tree), considering that the tiled matrix convolution needs to do the for-loop addition, we decided to use scan to replace of the original design of for-loop operation. Here firstly we decided to use the dimension x in our block to do the scan operation and took the method of Brent-Kung parallel scan as the implementation. We stored each output of tiled matrix multiplication in the shared memory and then use it to do the parallel scan. This is great because parallel scan is better than just using a for-loop addition when meeting a scan question. The test result is shown below, note that operation time is measured based on 10000 batches and nvprof results are measured based on 100 batches.

Operation time:

```
Op Time: 0.722005
Op Time: 1.957412
Correctness: 0.8171 Model: ece408
6.19user 3.31system 0:07.36elapsed 129%CPU (0avgtext+0avgdata
ent)k
0inputs+4608outputs (0major+703934minor)pagefaults 0swaps
* The build folder has been uploaded to http://s3.amazonaws.c
```

Kernel analysis:

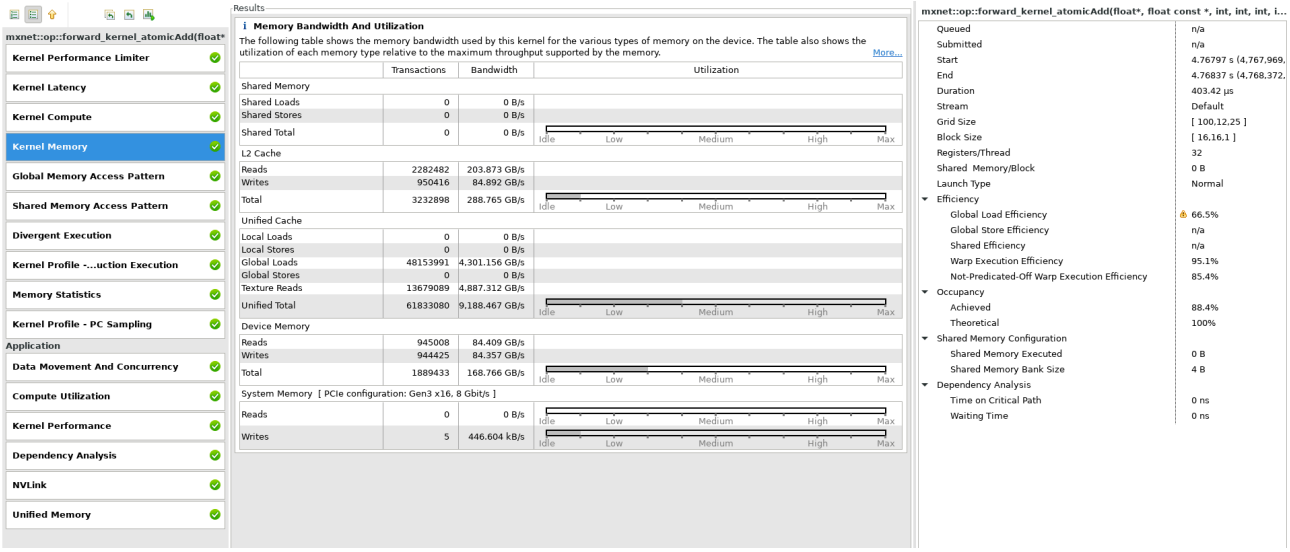


We then implemented 7 (Input channel reduction: atomics) and combined it with the original GPU implementation, this is to replace outer loop (the one loops from 0 to C) with atomic operation, as a way to reduce the amount of computation in each kernel. The test result is shown below, note that operation time is measured based on 10000 batches and nvprof results are measured based on 100 batches.

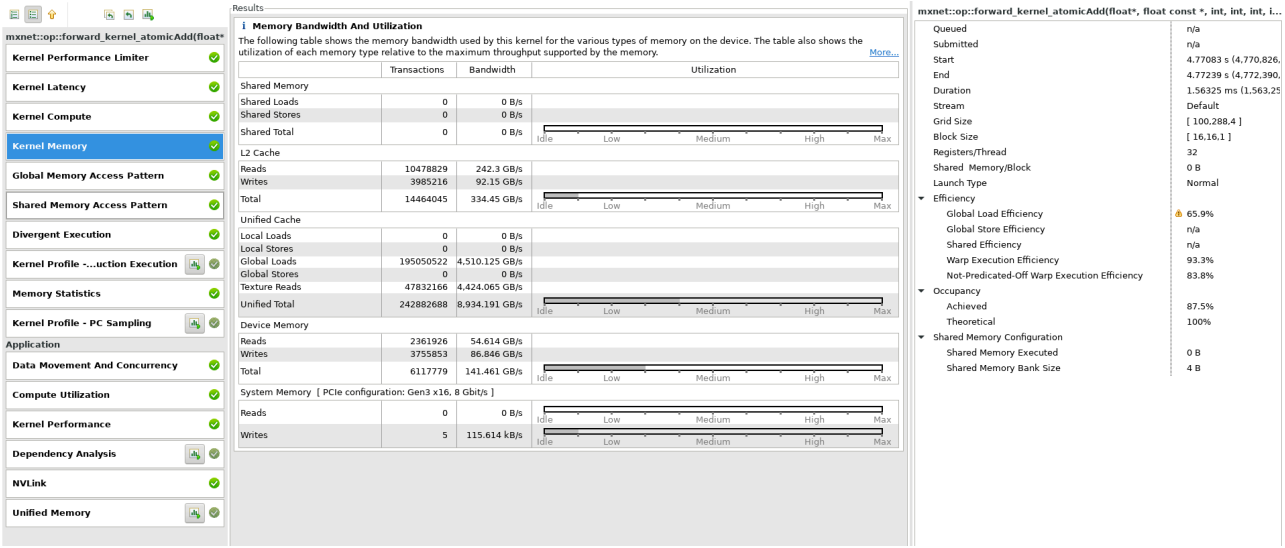
Operation time:

```
* Running /usr/bin/time python final.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.042121
Op Time: 0.150010
Correctness: 0.8171 Model: ece408
4.29user 2.35system 0:04.77elapsed 139%CPU (0avgtext+0avgdata 2850596maxresident)k
0inputs+4768outputs (0major+708691minor)pagefaults 0swaps
```

First layer:



Second layer:

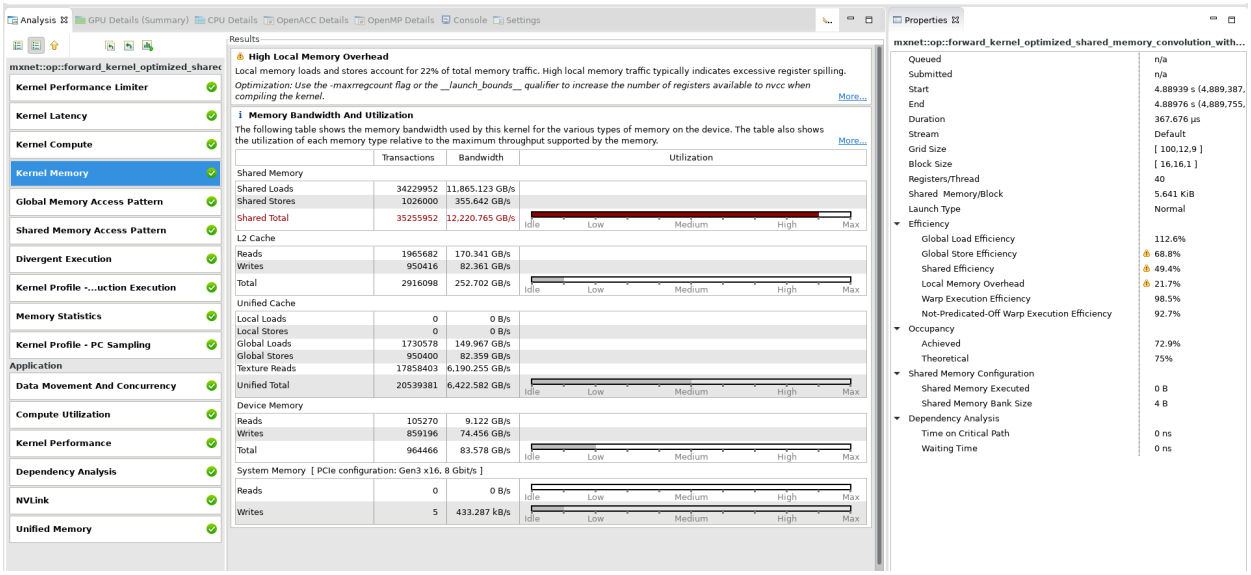


We then implemented 8 (thread coarsening) and combined it with 2 (shared memory convolution), because we thought there may exist many redundant work in each thread, for example most part of data restored in shared memory can be reused to compute multiple element as long as we expand properly the size of shared memory. After several experiments, we found that block size being 256 and one thread doing for threads' work gave us the best performance. The test result is shown below, note that operation time is measured based on 10000 batches and nvprof results are measured based on 100 batches.

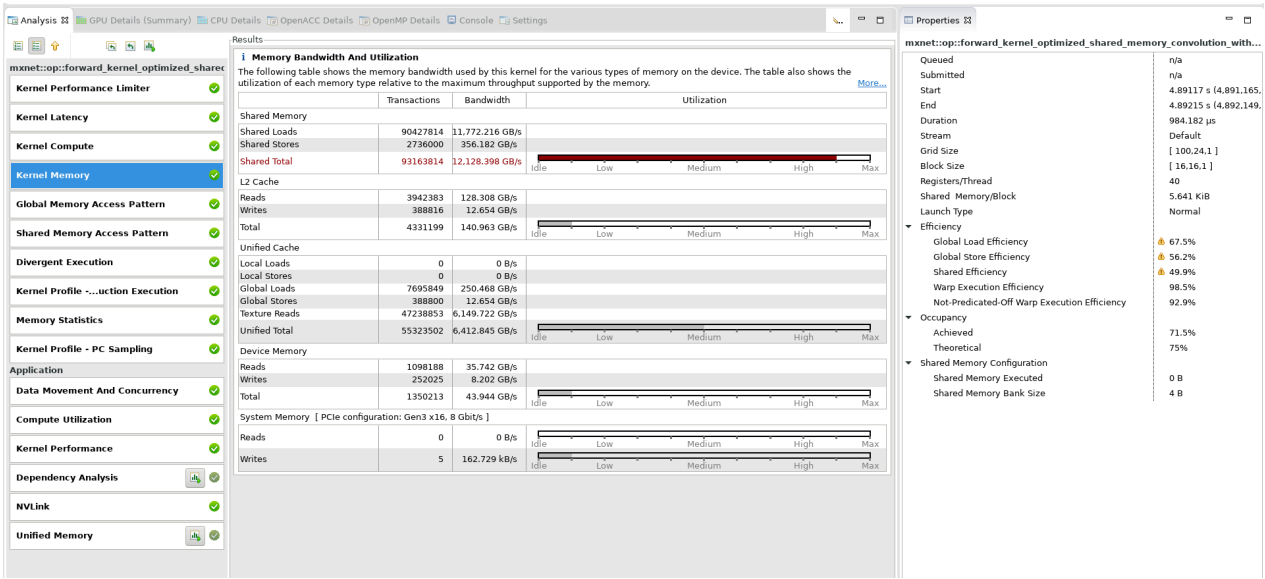
Operation time:

```
* Running /usr/bin/time python final.py
Loading fashion-mnist data...
done
Loading model...
done
New Inference
Op Time: 0.040558
Op Time: 0.103785
Correctness: 0.8171 Model: ece408
4.23user 2.45system 0:04.80elapsed 139%CPU (0avgtext+0
avgdata 2836552maxresident)k
0inputs+4752outputs (0major+705436minor)pagefaults 0swaps
```

First layer:



Second layer:

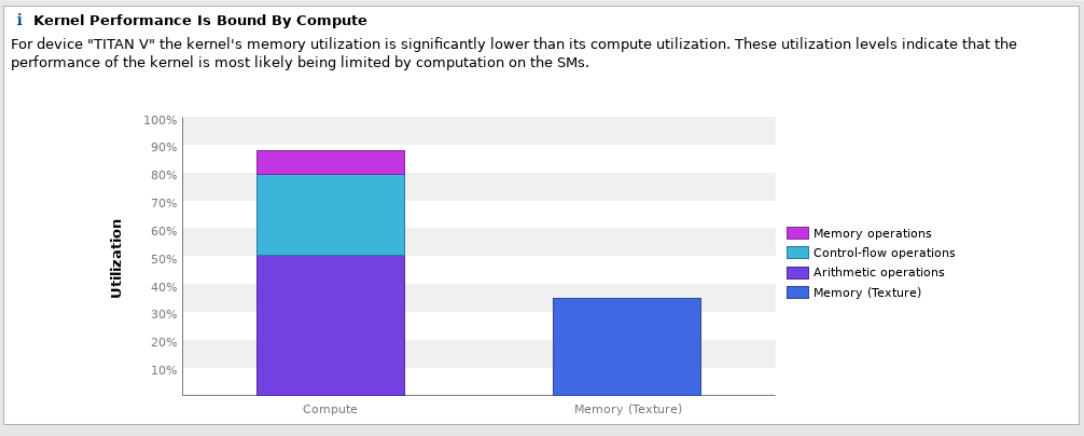


Analysis:

Kernel fusion: From the Kernel analysis part we find that by improve the kernel with unroll and tiled matrix multiplication, the shared memory usage in this kernel is better compared to Unroll/shared-memory matrix multiplication. Apparently by taking advantage of the shared memory throughput, we can achieve high performance.

Tree: During the experiment we found that some factors may cause the delay of the program. Firstly, the __syncthread() operation in the Brent-Kung parallel scan has limited the performance of the operation. Secondly, the active thread in the block is limited. Since we need to deal with tiled convolution, we cannot do large scan operation due to the hardware limit. Therefore, the final performance using Brent-Kung parallel scan in our program is not ideal. From result given by nvvp, we can see that after taking scan operation into our kernel, the shared efficiency performance did not improve simultaneously. It is considered that the limitation of thread block has restricted the performance since we cannot achieve high concurrency.

Atomics: Performance of Kernel using atomic operation is similar to the un-optimized version of kernel, we think it' s because the process of atomic operation, read-compute-write have larger latency compared with normal operation, test result from Nvidia Visual Profiler also shows that kernel performance is bounded by computation.



Thread coarsening: Thread coarsening is an effective optimization, compared with the original shared memory convolution, operation time is faster by 40%, which shows that we can take advantage of reuse element in shared memory. From Nvidia Visual Profiler, we can see that shared memory utilization is higher and the efficiency of shared memory improved from around 35% to around 50%, we think this increase in efficiency is reason why we can achieve better performance.