

Name: A. Kadri Turker

Netid: aturker2

Team Name: Mango Mad Lads

Team Members Names: Michal Kalita, Caleb Chow

Team Netids: mkalita2, calebyc2

School Affiliation: UIUC

ECE 408 Final Project Report

Milestone 5:

Optimization 4: Unroll + shared-memory Matrix multiply

How we identified the optimization:

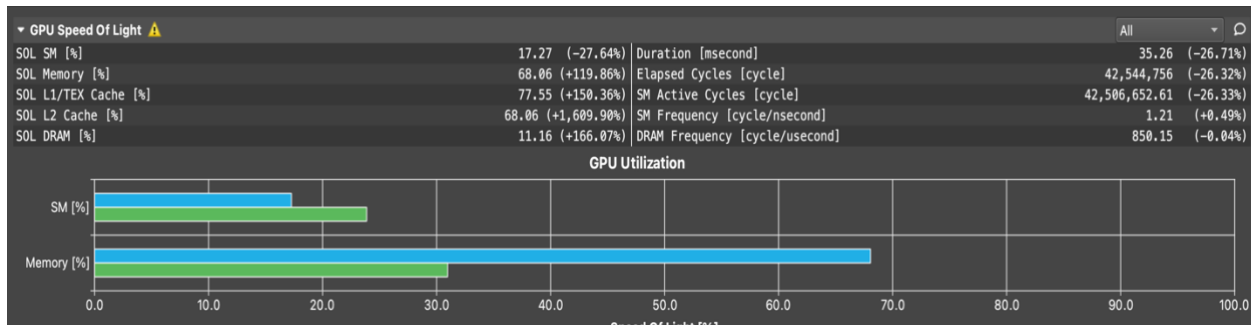
- Many people in the class were using unrolling and it was constantly mentioned on piazza.
- We also felt that the only algorithmic improvement we could do was unrolling + shared memory matrix multiply, because there were so many possible improvements to matrix multiply that could be done.

Why we thought that the optimization would be helpful:

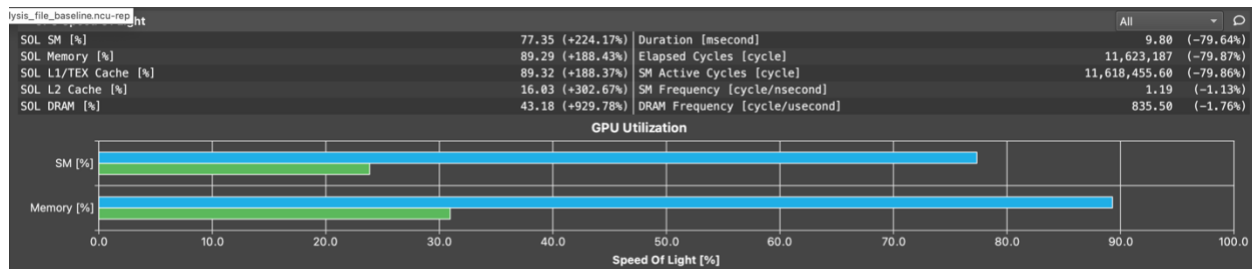
- Matrix multiplication is more embarrassingly parallel than convolution and has the potential to be implemented by register tiling and such advanced algorithms.

The effect of the optimization:

- The optimization was largely unsuccessful, the timings were around 10 times worse than our PM4 code. We used mini batches of size 2000, so each layer did 10 kernel calls, 5 for unrolling and 5 for matrix mult. It's in "custom/new-forward_unroll+matrixMult.cu" in our directory.
- The unroll kernel, one of 5 in a single layer, took almost the same time as the base line, in addition to that, the memory usage in unrolling became too much and made unrolling kernel the bottleneck, as can be seen below.



- The performance of matrix multiply, on the other hand, was pretty good. As can be seen.



- Overall, unrolling + shared memory matrix multiply was a disaster and was mostly memory bound thanks to unrolling, therefore we will not be using this optimization in our final implementation.

How we organized and divided the work:

- aturker2: Implemented the code, tested and made sure it worked.
- mkalita2: Did the nsight analysis, wrote the report for the optimization.
- calebyc2: Did testing, helped come up with the algorithm for modified matrix multiply.

Optimization 5: Tuning with restrict and loop unrolling

How we identified the optimization:

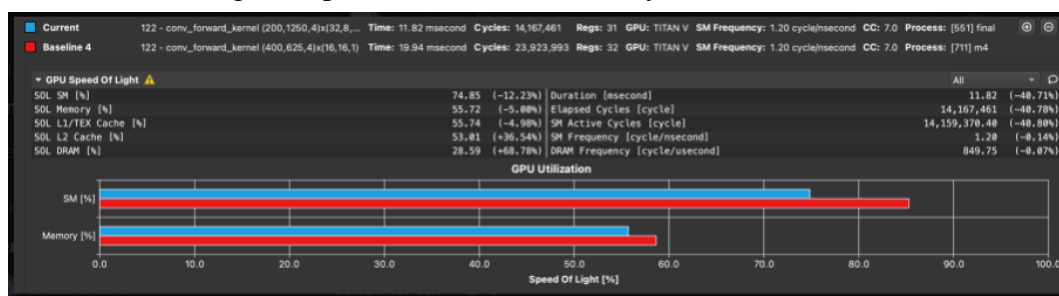
- The only part of our code that we couldn't parallelize was the for loop that did the convolution, therefore we decided to try loop unrolling as it was listed as an option on the final project documentation.

Why we thought that the optimization would be helpful:

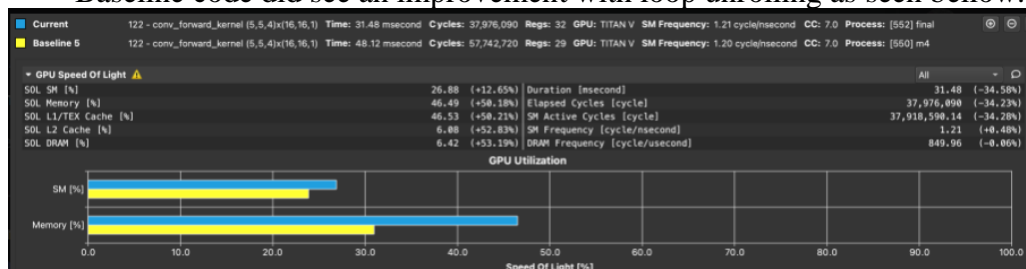
- We thought this optimization would be useful because `#pragma unroll` reduces the processing load for the processor and increases instruction-Level Parallelism. For example, in a for loop, instead of having the compiler do the code overhead for the loop, it unrolls the loop into simple accesses and loads. This would hypothetically speed up the code since there is less work the processor has to do.
- We also thought implementing the `restrict` keyword would disable pointer aliasing and because the compiler must conservatively assume pointer aliasing (two pointers if memory pointed to overlaps/same address) which slows down the code processing because the compiler must generate code to reload the value which was thought to be aliased. If we implement the `restrict` keyword where we know this explicitly does not occur, we can improve processing speed.

The effect of the optimization:

- When added to our fastest implementation. the effects of this optimization were a significant speedup, about 6x speedup for layer 1. We see much more GPU utilization in the streaming multiprocessor and the memory.



- Baseline code did see an improvement with loop unrolling as seen bellow.



How we organized and divided the work:

- aturker2: Did the nsight analysis, helped debugging.
- mkalita2: Wrote the report for the optimization, fixed algorithmic issues.
- calebyc2: Implemented the code, test and made sure it worked.

Optimization 6: Fixed point (FP16) arithmetic

How we identified the optimization:

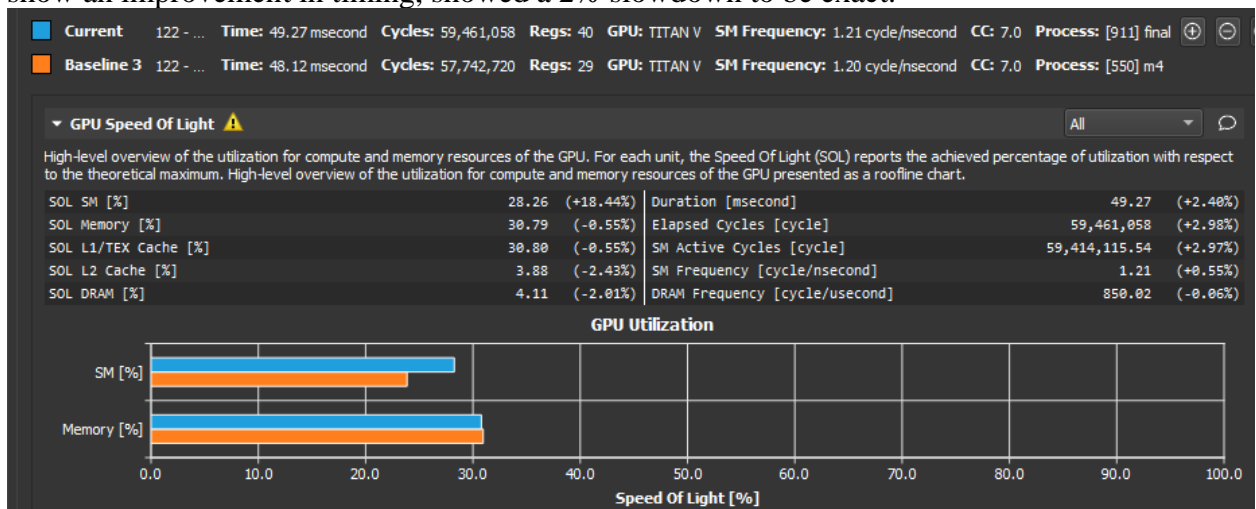
- We identified the optimization through the list of given optimizations on the project page and thought that it could help our implementation achieve better FP throughput.

Why we thought that the optimization would be helpful:

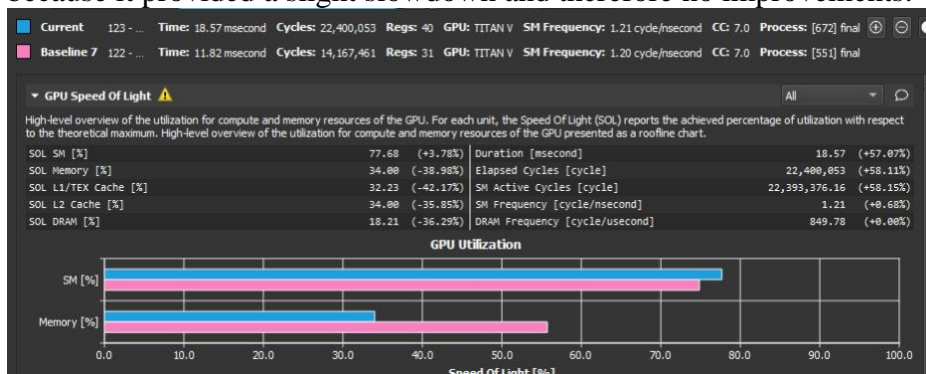
- CUDA documentation under “Mixed Precision Training” states that FP16 data takes up less memory and the transfer of FP16 data takes less time. Given this info, we thought it would be worthwhile to implement FP16.
- This optimization allows us to have less loop iterations by storing to two 16-bit float values inside a 32-bit float value, we could save up on the computational weight of the kernel.

The effect of the optimization:

- This optimization implemented along the baseline increased the SM usage and didn’t show an improvement in timing, showed a 2% slowdown to be exact.



- Here is a snapshot of fp16 with all the fastest optimizations v.s all the fastest optimizations. We decided not to include this optimization in our final submission because it provided a slight slowdown and therefore no improvements.



How we organized and divided the work:

- aturker2: Came up with the algorithm, looked up library functions.
- mkalita2: Implemented the code.
- calebyc2: Did the nsight analysis, wrote the report for the optimization.

EXTRA Optimization 7: Atomic Add and more exploitation of parallelism:

How we identified the optimization:

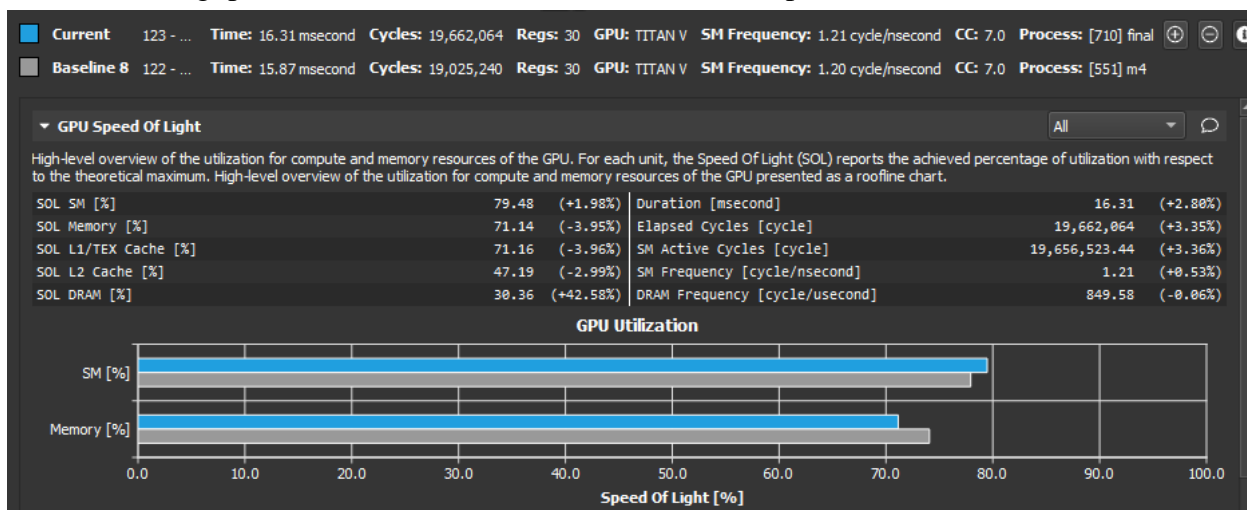
- After the first optimization we wanted to improve the parallelism by parallelizing the for loop that did the convolution and decided to use atomic operations to achieve that.

Why we thought that the optimization would be helpful:

- Exploitation of parallelism provided a 3x boost to our timings, so we thought more parallelism, even with atomic operations, would be beneficial

The effect of the optimization:

- The atomic add operations optimization alongside the first two optimizations was one of the optimizations we considered, however, it turned out to be a little slower than the first two optimizations.
- Since atomic operations halt some threads and make them do nothing, this increases the total amount of cycles in our code, as can be seen in the top right, where we have 600,000 more cycles than the code for optimization 1 and 2. All other statistics such as memory throughput and GPU utilization for the SM were superior.




How we organized and divided the work:

- aturker2: Implemented the code.
- mkalita2: Did the testing, generated nsight files, concluded usefulness.
- calebyc2: Came up with the idea, designed the indexing.

Milestone 4:

Optimization 1: Exploitation of Parallelism

How we identified the optimization:

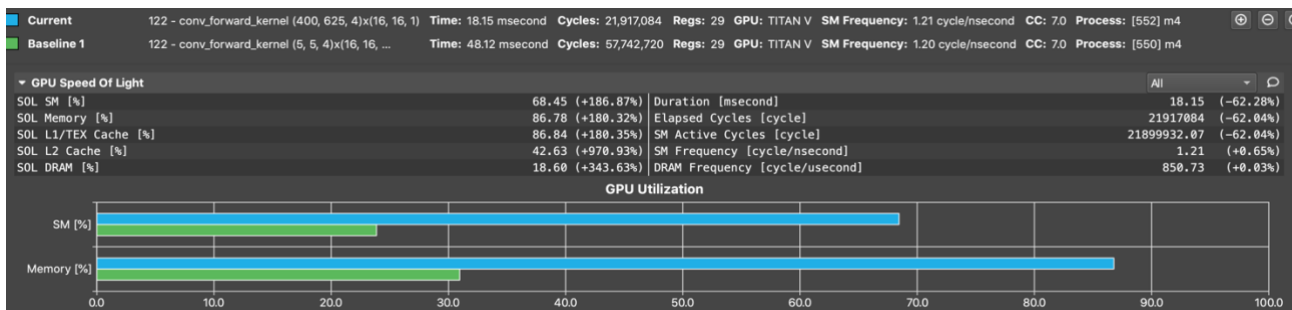
- Nsight gave the following optimization recommendation:
- 
- So, we decided that making the batch number the y-dimension of the block and combining height and width into the x-dimension of the block would allow the increase of number of threads.

Why we thought that the optimization would be helpful:

- The optimization, we thought, would allow us to use more of the GPU resources and use the inherent parallelism to speed our program up.

The effect of the optimization:

- The optimization allowed us to speed up our code around 3 times compared to its baseline.
- As can be seen in the screenshot below, SM and Memory utilization increased by a significant margin as we desired, and our program got around %62 compared to our initial timings.



How we organized and divided the work:

- aturker2: Looked at Nsight and came up with the possibility of the optimization.
- mkalita2: Did the implementation of the code.
- calebyc2: Did the testing, generated nsight files, concluded usefulness.

Optimization 2: Constant memory usage for the kernel

How we identified the optimization:

- After the previous optimization, our memory usage increased a lot and nsight recommended that we should try to reduce the usage to around 70%.
- Since the same kernel values are used in parallel, the constant memory is more beneficial to us in accessing the kernel memory rather than the DRAM burst since we're using the same value of the kernel in parallel.

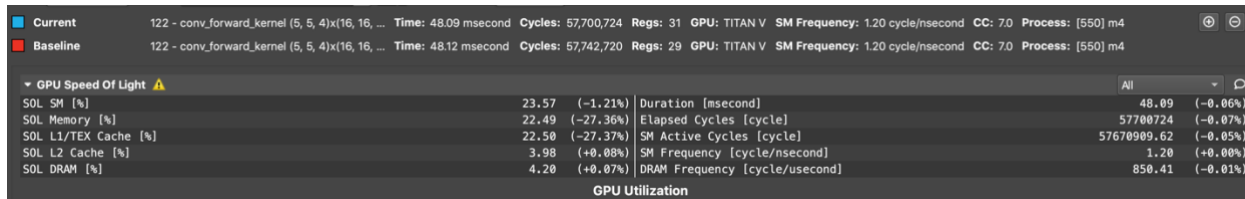
Why we thought that the optimization would be helpful:

- The memory usage (%86.78) was too much compared to the SM utilization (%68.45), so we decided that something that would ease up the memory usage would make our program faster.

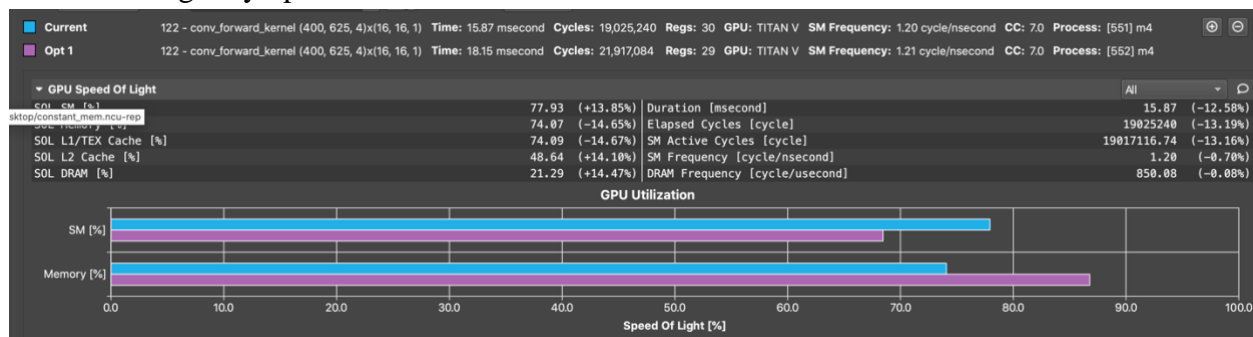
The effect of the optimization:

- The optimization, by itself without the first optimization, offered a really insignificant speedup, which is to be expected as our memory utilization in the baseline wasn't too much, but when we looked at the combination of both optimizations, we saw a more significant speedup compared to having just optimization 1.

- The following screenshot shows the speedup of the optimization by itself compared to the baseline.



- The next screenshot shows how the combination of both optimizations were compared to having only optimization 1.



- It can be seen on the 2nd screenshot that the usage of constant memory did successfully lower the memory % and provided us with a more balanced SM and Memory utilization.

How we organized and divided the work:

- aturker2: Implemented the code, tested and made sure it worked.
- mkalita2: Did the research on nsight, found the possibility of optimization.
- calebyc2: Did more testing, generated nsight files, concluded usefulness.

Optimization 3: Sweeping the TILE WIDTH to find the best value

How we identified the optimization:

- We had never thought of what effect the tile width would have on our program, so we decided to explore those values further.
- We decided that we could use different tile widths for the x and y dimensions, so we decided to further explore that idea.

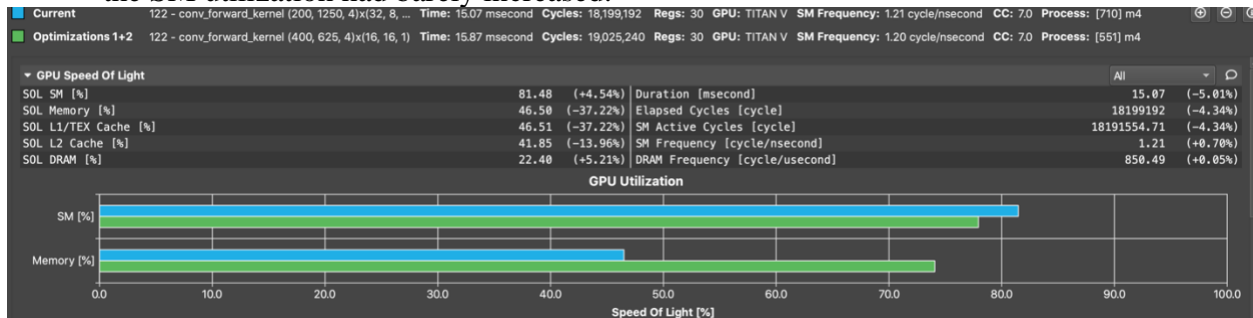
Why we thought that the optimization would be helpful:

- We knew that exploration of this idea could help us with the memory coalescence since each block would be composed of more warps.
- Sweeping the tile width value could allow us to have significant improvements with minimal algorithmic changes, which is always a plus.

The effect of the optimization:

- We decided NOT to try the optimization by itself on the baseline code since our first optimization did increase the number of threads by exploiting the parallelism of the code, so we used the version of our code that included the first two optimizations as the baseline.
- Our final best tile widths were 32 for the x dimension and 8 for the y dimension while our initial tile width was 16 for both dimensions.
- We were able to observe slight improvements in the timings, around %5 to be precise.

- When we profiled the kernel, we saw some interesting results. We observed that the memory utilization had decreased substantially compared to before the optimization and the SM utilization had barely increased.



- This decrease in Memory utilization is helpful as we'll be able to implement optimizations that are taxing on the memory usage.

How we organized and divided the work:

- aturker2: Realized the chance for optimization and setup the framework for research.
- mkalita2: Did the research to find the optimal dimension combination.
- calebyc2: Helped with research to find optimal dimension combination.

Milestone 3:

Output of rai running my GPU implementation of convolution for all 3 batch sizes:

```
* Running /bin/bash -c "./m3 100"
Test batch size: 100
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Op Time: 128.587 ms
Conv-GPU==
Op Time: 6.74768 ms
Test Accuracy: 0.86
```

```
* Running /bin/bash -c "./m3 1000"
Test batch size: 1000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Op Time: 288.263 ms
Conv-GPU==
Op Time: 61.5025 ms
Test Accuracy: 0.886
```

```
* Running /bin/bash -c "./m3"
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-GPU==
Op Time: 882.683 ms
Conv-GPU==
Op Time: 607.39 ms
Test Accuracy: 0.8714
```

Demonstration of nsys profiling the GPU execution:

```
Collecting data...
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
```

```

Generating CUDA API Statistics...
CUDA API Statistics (nanoseconds)
Time(%)      Total Time      Calls      Average      Minimum      Maximum      Name
-----
83.7         1219766367         6      203294394.5      82263      590361123      cudaMemcpy
16.1         234824592         6      39137432.0      62693      232545388      cudaMalloc
0.2          2196353         6      366058.8      58243      910401      cudaFree
0.0          244894         2      122447.0      26146      218748      cudaLaunchKernel
Generating CUDA Kernel Statistics...

Generating CUDA Memory Operation Statistics...
CUDA Kernel Statistics (nanoseconds)
Time(%)      Total Time      Instances      Average      Minimum      Maximum      Name
-----
100.0        214188393         2      107094196.5      48251972      165936421      conv_forward_kernel

CUDA Memory Operation Statistics (nanoseconds)
Time(%)      Total Time      Operations      Average      Minimum      Maximum      Name
-----
92.4         922755227         2      461377613.5      381454529      541300698      [CUDA memcpy DtoH]
7.6          76386630         4      19096657.5      1184      41167323      [CUDA memcpy HtoD]

CUDA Memory Operation Statistics (KiB)
Total      Operations      Average      Minimum      Maximum      Name
-----
1722500.0      2      861250.0      722500.000      1000000.0      [CUDA memcpy DtoH]
538919.0      4      134729.0      0.766      288906.0      [CUDA memcpy HtoD]

```

List of all kernels that collectively consume more than 90% of the program time:

There is only one kernel, conv_forward_kernel, and it's being called twice, so:

- conv_forward_kernel

List of all CUDA API calls that collectively consume more than 90% of the program time:

There are 4 different CUDA API calls, the ones, combined, that consume 90% of the program time are:

- cudaMemcpy (called 6 times)
- cudaMalloc (called 6 times)

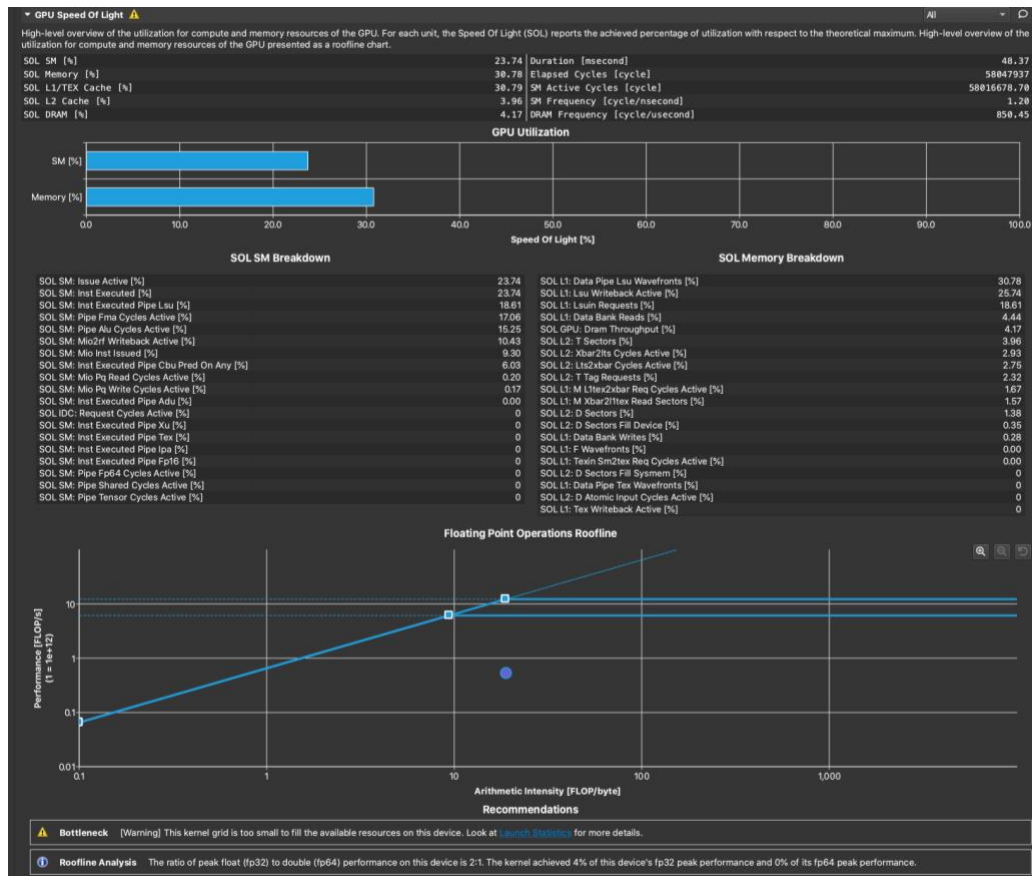
The difference between kernels and API calls:

A kernel, in our programs, is a C++ function that starts with the “__global__” keyword. When called, “are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C++ functions” (From the CUDA documentation of kernels).

An API, on the other hand, is a set of definitions, protocols and tools for building software. In CUDA, “it provides C and C++ functions that execute on the host to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc.” (From the CUDA documentation on Programming Interface).

In Kernel's functionality is defined, but in API calls we are not implementing functionality, we just pass in parameters and use an already existing resource.

Screenshot of the GPU SOL utilization in Nsight-Compute GUI for my kernel profiling data:



Milestone 2:

Output of rai running Mini-DNN on the CPU for batch size of 10k images:

```
* Running /bin/bash -c "time ./m2 10000"
Test batch size: 10000
Loading fashion-mnist data...Done
Loading model...Done
Conv-CPU==
Op Time: 54885.1 ms
Conv-CPU==
Op Time: 158249 ms
Test Accuracy: 0.8714
real    5m8.435s
user    5m7.431s
sys     0m1.000s
```

Op Times (CPU convolution implemented) for batch size of 10k images:

- 54885.1 ms
- 158249 ms

Whole program execution time (CPU convolution implemented) for batch size of 10k images:

Whole program execution time = sys + user = 5m(7.431 + 1.000)s = 5m8.431s

Whole program execution time: 5 minutes and 8.431 seconds