**ECE 408 Final Project Report**

**Andrew Betbadal (betbadl2), Michael Beaudin (mbeaudi2), Atin Ganti (ganti3)**

**big\_baller\_brand\_123**

**Milestone 1:**

**Kernels that take 90% of program time**

Cuda memcpy HtoD

volta\_scudnn\_128x32\_relu\_interior\_nn\_v1

implicit\_convolve\_sgemm

volta\_sgemm\_128x128\_tn

activation\_fw\_4d\_kernel

**API calls that take 90 % of program time**

cudaStreamCreateWithFlags

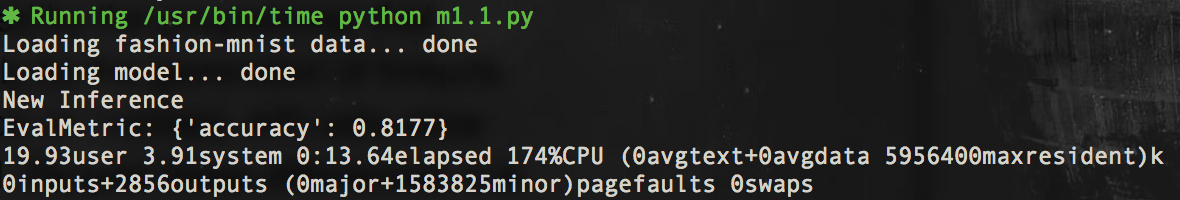
cudaMemGetInfo

cudaFree

**Include an explanation of the difference between kernels and API calls**

In Cuda, kernels are C functions defined by the programmer that allow a programmer to execute code parallely by the generation of threads instead of behaving like a standard C function. They must be signified by the global keyword. In contrast, API calls are a set of subroutine definitions, protocols, and tools that allow building software. API calls like cudaFree are required in order to build the software effectively.

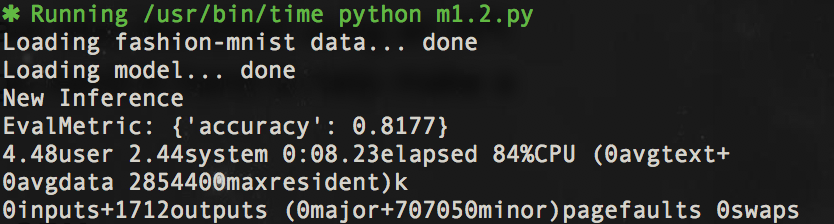
**Show output of rai running MXNet on the CPU**



**List program run time**

Run Time : 13.64

**Show output of rai running MXNet on the GPU**

****

**List program run time**

Run Time : 08.23

**Milestone 2:**

**List Whole Program Execution Time :**

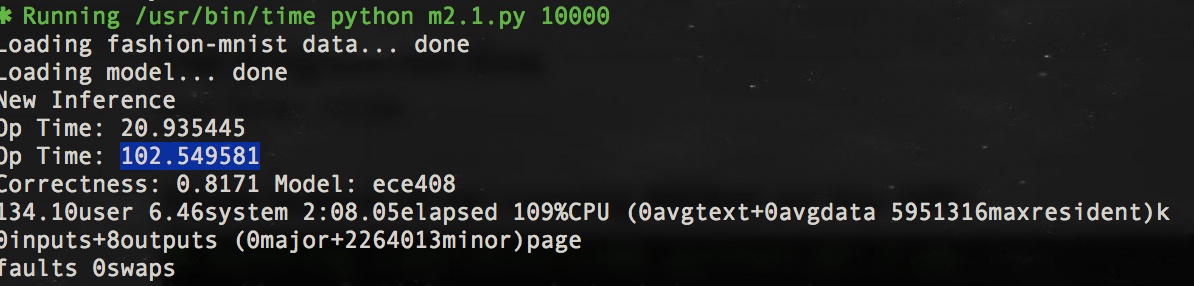
Run time : 2:08.05

**List OP Times :**

OP Time 1 : 20.935445

OP Time 2 : 102.549581

**Output:**

****

**Milestone 3:**

**Data Set Size 100:**

Op Time 1 :0.000648

Op Time 2: 0.002359

Correctness : 0.85

**Data Set Size 1000:**

Op Time 1: 0.006191

Op Time 2: 0.023141

Correctness: 0.827

**Data Set Size 10000:**

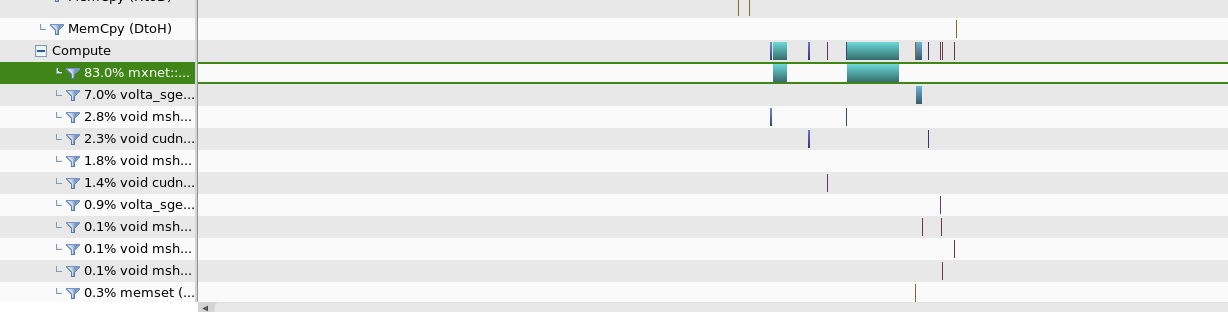
Op Time 1: 0.061674

Op Time 2: 0.212911

Correctness: 0.8171

**NVprof analysis**

Our primary slowdown was global memory reads. Because we were not loading into shared memory, the majority of our GPU usage was the overhead required to load in data from global memory.

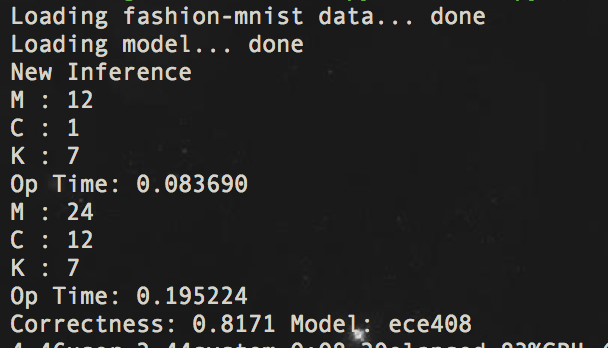


**Milestone 4:**

**Optimizations :**

**Kernel in Constant Memory :**This was one of the easiest, and also most obvious implementations. Because the kernel values were read only, we had simply had to create the kernel in constant memory. We made the kernel size 24\*12\*7\*7, as that was the largest kernel we would expect, and the constant memory size had to be hardcoded. Since the constant memory has a much quicker read time, the time improvement is noticeable.

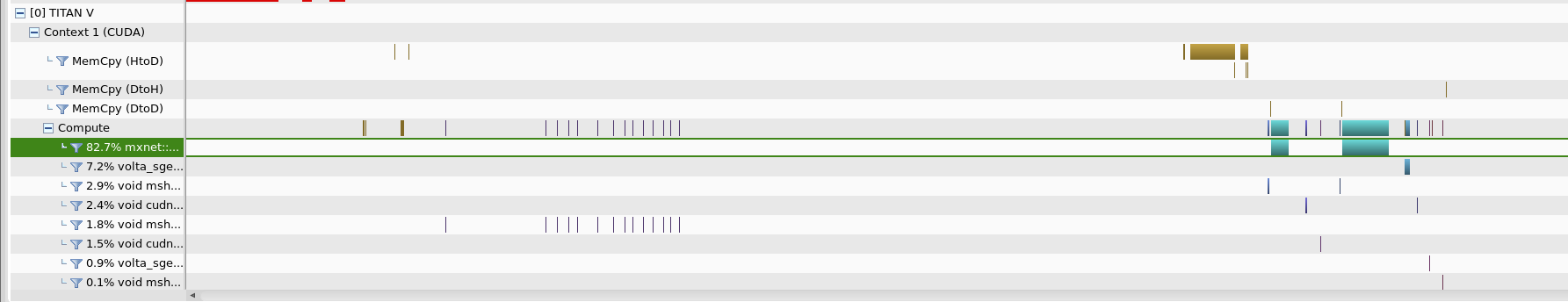
**Time :**

****

**Analysis:**

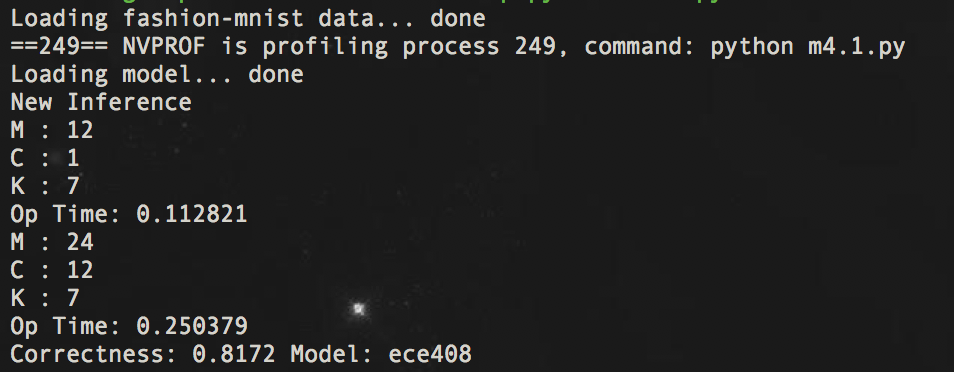
One interesting point is that for the smaller M and C valued model, there is a decrease in time efficiency, with an OP time of .083690 compared to .061674 of the non optimized code. However, with the larger M and C values, there is an OP time of .195224, compared to 0.212911.

**NVVP:**



**Shared Memory :**

Shared memory was our second optimization. Because the kernel was already stored in the constant memory, we only had to write the input tiles into shared memory. Therefore, the size was (TILE\_WIDTH-K+1)^2. We had to load in parallel the shared memory values, but noticed a significant improvement in time. One concern was boundary conditions, so any tiles that were beyond the bounds of the tile were stored as zero in shared memory.

****

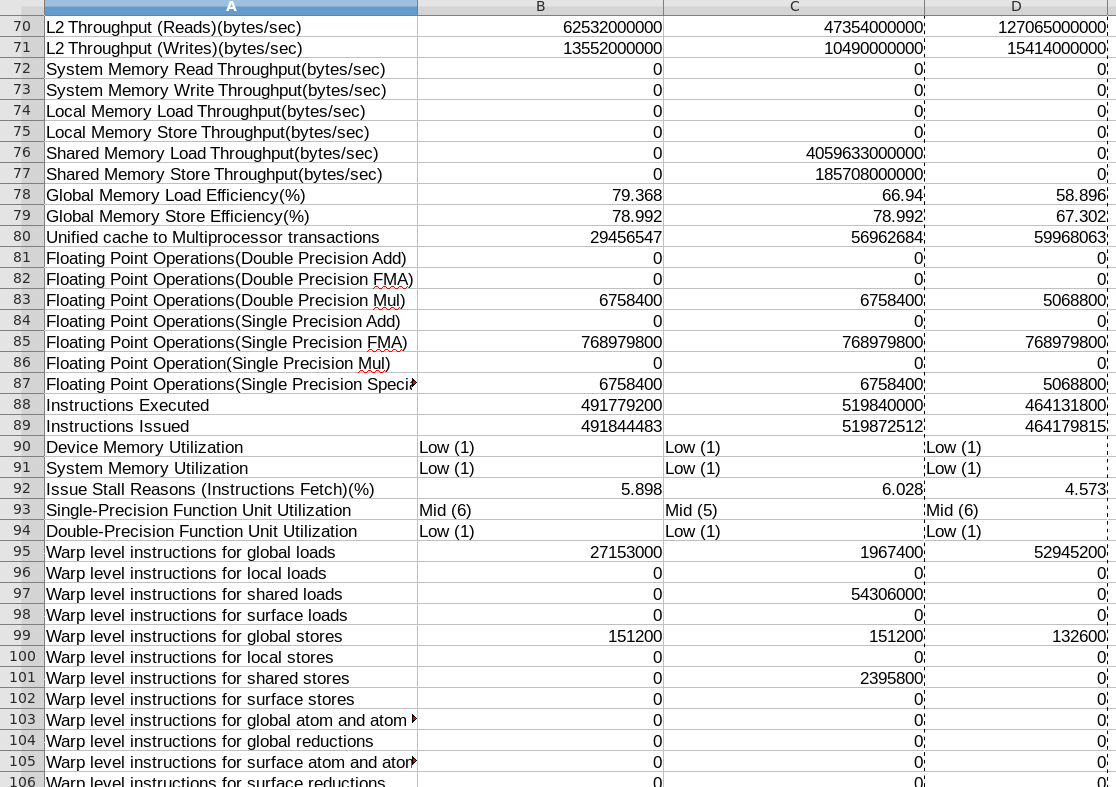
**Analysis:**

We noticed a significant increase when using the shared memory. This may have been due to the amount of overhead required to utilize the shared memory appropriately, however we were still very surprised by how much longer this optimization took, compared to the unoptimized code. This is particularly noticeable for small M and C values. This makes sense, however, as the overhead required to write to shared memory for smaller values is a much greater percentage of the total work required.

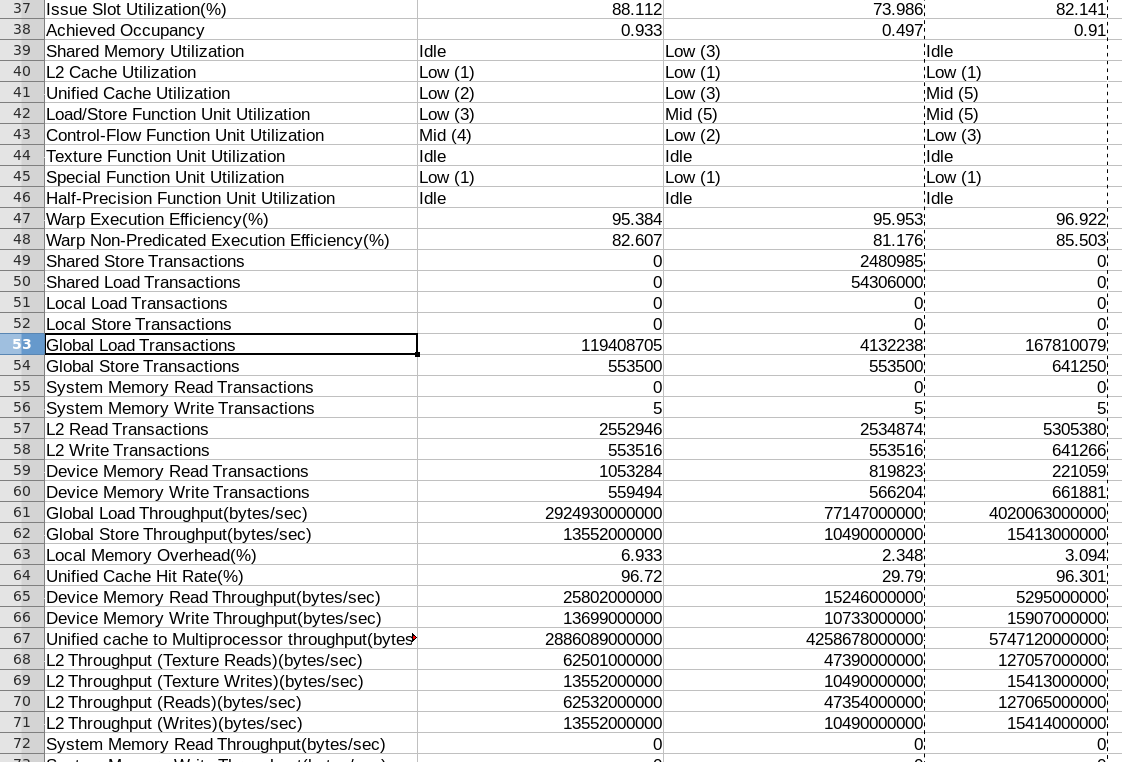
**NVVP**



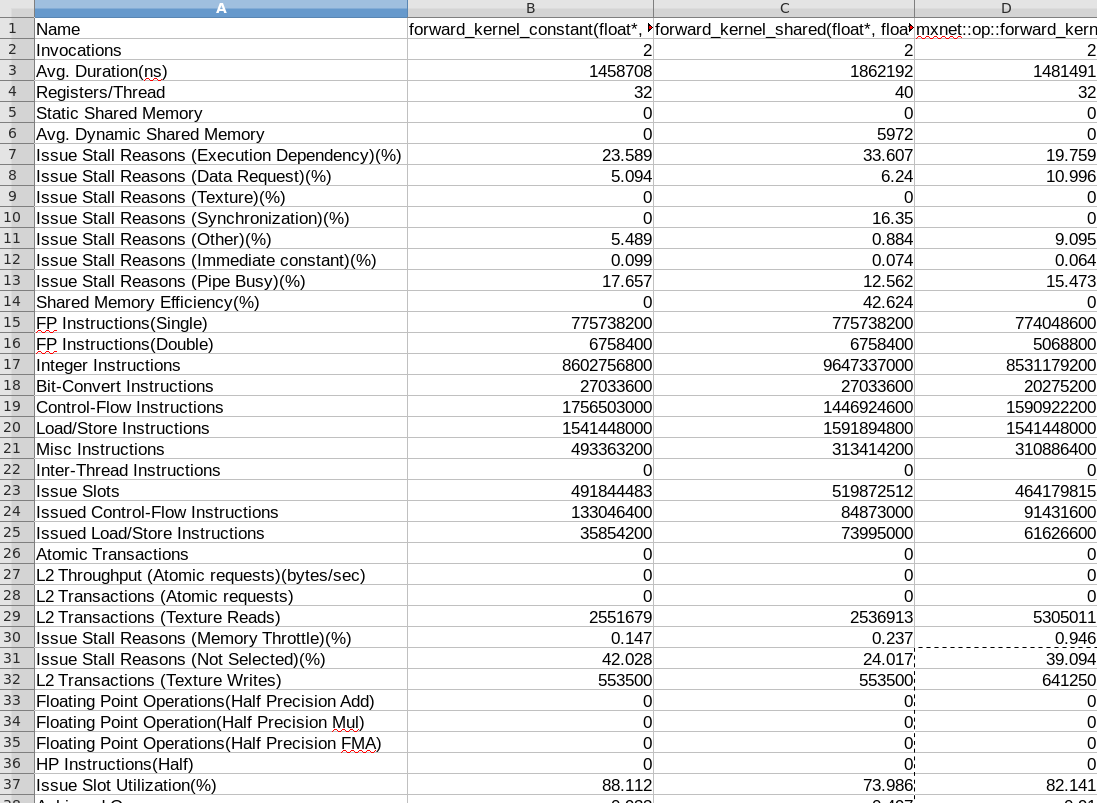
**Analysis Overall:**



In the image above, it can be seen that the Warp level instructions for the shared loads and for shared stores is significantly higher for the column C, which corresponds to the shared memory optimization. This is to be expected, as we are only utilizing shared memory for the shared memory kernel. Column B corresponds to the constant memory optimization, and column D corresponds to the unoptimized code.



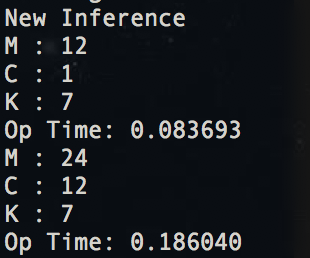
We can see above that the global load transactions is significantly lower for the shared memory optimization, as we are reading from the global memory much less once we store values into the shared memory. We can also see, however, that the Device memroy read transactions is much lower for the constant memory in Column B. This is because, once we load the constant memory, we are no longer required to read from the global memory, significantly reducing the total amount of transactions required.



The image above depicts the advantages of using the shared memory optimization. As you can see, when compared to the unoptimized and constant memory model, the shared memory optimization increased the Average Dynamic Shared Memory usage from 0 to 5972 bytes and the Shared Memory Efficiency from 0 to 42.694%. This is expected as we are not utilizing the shared memory capabilities in the constant memory optimization or unoptimized versions.

**Final Milestone:**

As we were adding more optimizations, we used the combination of constant memory for the kernel with using shared memory for the input matrix as our base, as that was the best runtime we had after Milestone 4. Seen below is the runtime for our program after Milestone 4.

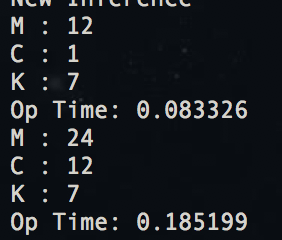


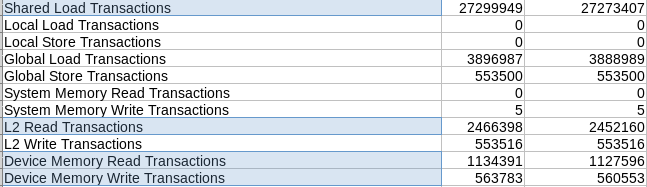
**Optimizations:**

As a note, in order to save time on runtime, all NVVP profiles for the following optimizations will be run using 100 as the parameter, while all timing images, such as above, will be run using 1000 as the parameter.

**Tuning with restrict, loop unrolling**

For this optimization, we chose to use the \_\_restrict\_\_ indicators for the READ ONLY (MAYBE CHANGE) arrays that we passed in, and also used the #pragma unroll indicator on outside most loop. We saw fairly positive returns, with the overall time decreasing noticeably.



As we profiled this optimization, a few key points stuck out. While some of the improvements we saw were marginal by adding just this optimization, there were also a few statistics that stood out. For reference, the right column is after the optimization.

Highlighted in blue are points of interest. Our shared load transactions are significantly lower, as well as as our Device Memory Read/Write transactions. This was slightly unexpected, as the benefit of this optimization was primarily during compile time, as there would be less time spent resolving pointer aliasing. There was a slight increase in speed, but not as great as we would have expected. After analysing the code, however, we realized that unrolling the for loop resulted in a race condition that occasionally outputted an incorrect accuracy. In order to resolve this, we had to add an additional sync threads, and the effect of this can be seen in the increase in percentage of stall issues due to synchronization.



**Parameter Change / Coarsening**

Our next optimization was changing several parameters within the forward kernel. Our first, and rather simple change, was removing K as a parameter passed into the function and defining it as a global variable because it remained at 7. While this was a simple change, it reduced the amount of data required to pass from CPU to GPU, saving bandwidth that could be used elsewhere. After we made this change, we proceeded to sweep across multiple TILE\_WIDTH values. Shown below are the time values for given TILE\_WIDTH values.

|  |  |  |
| --- | --- | --- |
| TILE\_WIDTH | OP Time 1 | OP Time 2 |
| 8 | .05826 | .139762 |
| 16 | .078293 | .195365 |
| 32 | .082845 | .186868 |

It is apparent that a TILE\_WIDTH value of 8 performed the best, so that’s what we went with. Our next parameter change involved utilizing the calculation independence of each image batch in B. Prior to this change, gridDim.x was equal to B, so that each image was given its own block in the x dimension. We modified this, so that we would batch the images, and every thread would cover B/BATCH\_SIZE images, so that we would only have to launch BATCH\_SIZE blocks in the x dimension. We saw great improvements from this as well. Shown below are the time values for given BATCH\_SIZE values.

|  |  |  |
| --- | --- | --- |
| BATCH\_SIZE | OP Time 1 | OP Time 2 |
| 10 | .0433 | .207498 |
| 100 | .045852 | .176942 |
| 1000 | .04722 | .133552 |

It is important to distinguish OP Time 1 and 2 here, because we can see that smaller batching value is better for the first operation, while larger batch value is better for the second operation. So, we chose to distinguish the two, so that the first operation ran on a batch size of 10, while the second operation ran on a batch size of 1000, giving us a much better run time than one or the other independently. For this optimization we ran on a larger data set than before, a set of 1000, so comparisons to previous optimizations will take that into account.

On the left is the Shared Constant Base, and in the middle is after utilizing restrict and pragma unrolling, and on the right is the parameter changed kernel.

We can see below that we achieved slightly higher warp execution efficiency, as we fewer blocks, so less warps that could be executed inefficiently.



Scaling the right most column by 10 (since we launched 1000 flag instead of 100), we can see that the issued load/store instructions and control-flow instructions are lower, which is to be expected, because we relying more on registers since we chose to batch.



We see much lower Synchronization stalling, again due to the fact that we are launching 1/BATCH\_SIZE fewer blocks in the X dimension, so there are less blocks that require synchronization. We also see greater shared memory utilization and cache utilization, as we are using block shared memory BATCH\_SIZE times more.





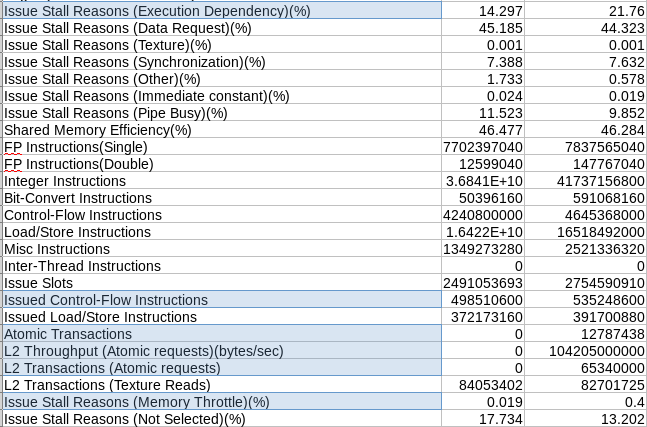
**Input Reduction Using Atomics**

Our next optimization was reducing the input by using Atomic addition. One latency that we noticed was that the outside loop in our original shared memory convolution, iterating over C, was inefficient. We chose to view it similar to a histogram, where each accumulated value of C was calculated independently, and then added to each “bin”, where the bin was y(bx, by, h\_out, w\_out). So, rather than just launching (TILE\_WIDTH,TILE\_WIDTH,1) sized blocks, we chose to launch (TILE\_WIDTH, TILE\_WIDTH, C) sized blocks. In doing this, we could remove the outermost iteration over C, and instead use atomicAdd once each value was calculated. So, threadIdx.z determined which input map we were using to compute per image. Our results were below.

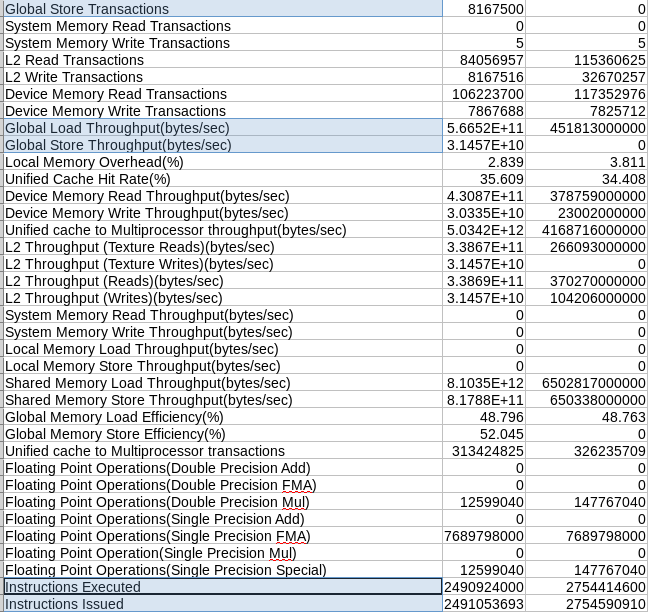
|  |  |
| --- | --- |
| OP Time 1 | OP Time 2 |
| .050817 | .152171 |

These values were not what we expected, as we had assumed that reducing the input would increase the efficiency. However, when looking at the NVPROF result, it makes sense. For reference, the Optimization described directly above, parameter change, is the left column, and this C reduction optimization is the right column.

Seen below, we can see that Execution dependency took up a much larger proportion of stall reasons, which makes sense as atomic adds are not able to complete until the previous atomic add completed. We see this also in memory throttle, as we were limited on the speed we could write to the output. We also see much high issued control-flow instructions as well, which again makes sense for the same reasoning as above.



We also see, at the bottom below, much higher levels of instructions executed and instructions issued. We do, however, see much fewer global store transactions. We also see much lower global load throughput, which may be due to the fact that we are attempting to load C times more memory in each block, thereby reducing the number of bytes we can access per second.

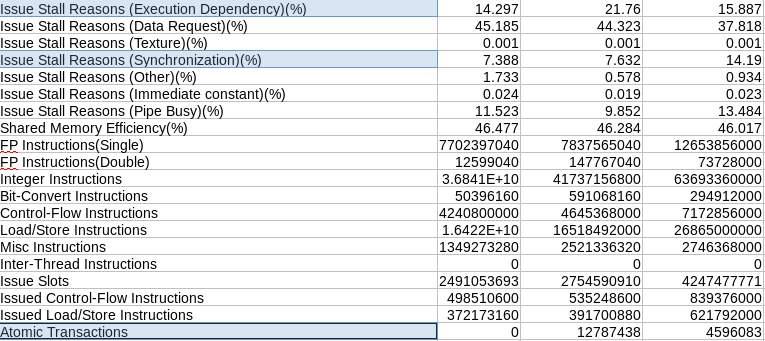


We tried improving this by batching C values to improve coarsening, similar to how we did for each image in B. For example, with a C batch size of 2, we launched blocks with z dimensions of 2 instead of 12, so each thread was responsible for calculating 6 values and atomic adding each. We tested several values, with their time values shown below. We got the following time results.

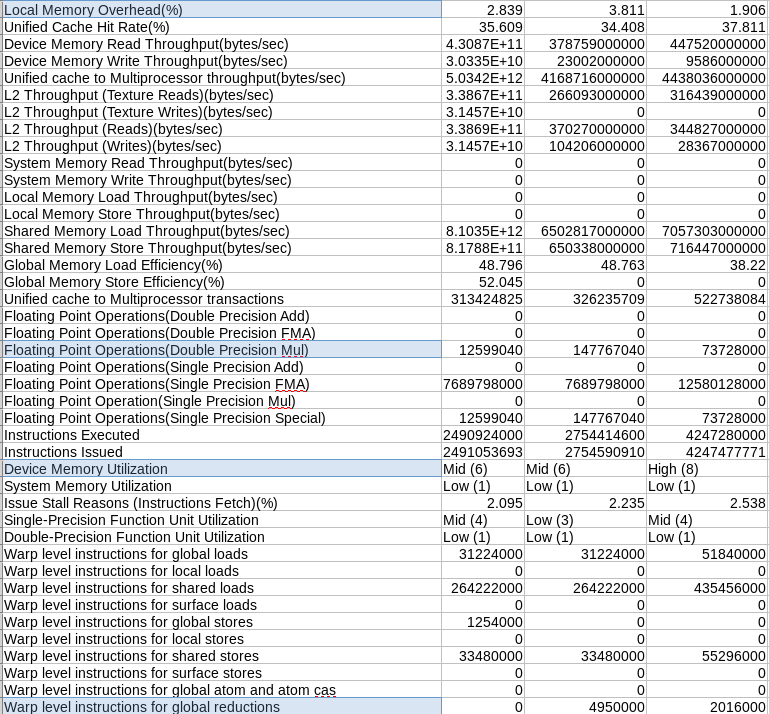
|  |  |  |
| --- | --- | --- |
| C Batch Size | OP 1 Time | OP 2 Time |
| 2 | .060194 | .143798 |
| 3 | .059298 | .146284 |
| 4 | .052028 | .138999 |
| 6 | .060065 | .145763 |

Again, we were surprised at the lack of improvement in time. Once more, we had to look at the NVProfile results. Again, leftmost column is using just parameter change optimization, middle column is using C reduction optimization, right column is both previous optimizations and C batching.

We can see some improvements, as we see that execution dependency was a lower percentage of stall issues, since we were only doing C/C\_BATCH\_SIZE atomic adds at a time. We do see greater percentage of synchronization issues, which is to be expected, as we were running the kernel C\_BATCH\_SIZE times more.



One positive we see is significantly lower local memory overhead %, since launching blocks that were 1/C\_BATCH\_SIZE times smaller reduced the amount of local memory required to launch the kernel.



**Unrolled Matrix Multiplication**

For our final optimization, we chose to implement an unroll kernel and our matrix multiplication code from the MPs. Unfortunately, we did not have time to fuse the kernels together, so we chose to run them independently. Additionally, unlike previous implementations, we were unable to figure out how to launch one kernel for the entire batch of images, or to implement mini batching to improve coarsening, so we had to launch each of the kernels B times, for each image. This resulted in really terrible run times, which can be seen below.



These run times are remarkably bad, but again are understandable as we are launching a kernel upwards of 10000 times. Had we been able to mitigate the issue of launching one kernel and handling all of the images in the batch, this issue would have been resolved. However, looking at NVPROF, we do see that the code was as efficient as promise.

The unroll kernel is on the far right, and the matrix multiplication kernel is second to the right. We see vast improvement in Global Memory load and store efficiency, which is a huge plus, and makes sense as the tiling method we implemented in the WebGPU MP was designed to be incredibly fast.



We also see remarkable improvements in the warp execution efficiency, as we had implemented the same intelligent indexing in the matrix multiplication kernel that we used in the WebGPU MP.



Our main issue in resolving the issue of launching the kernel pair for every image was the shared memory load. It was incredibly difficult to try and load the images intelligently into shared memory using the adjusted indexing, at least from what we tried.

**Files**New-forward.cuh : Most optimized

**In Folder all\_optimizations:**

New\_forward\_cidx.cuh : Using atomic add for the C Index

New\_forward\_cidx\_coarsened.cuh : Atomic add for C index with attempts to use thread coarsening by changing batch C sizes.

New\_forward\_just\_constant.cuh : Just using constant memory

New\_forward\_just\_shared.cuh : Just using shared memory

New\_forward\_matrixunroll.cuh : using the shared memory matrix multiplication

New\_forward\_thread\_restrict.cuh : Using Pragma and Restrict alongside shared memory