**Milestone 3 Report**

1. **Baseline:**

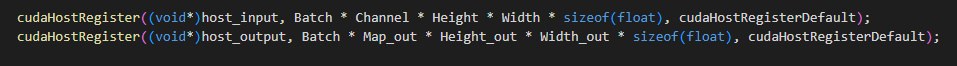
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.183975ms* | *0.530455ms* | *0m1.447s* | *86%* |
| 1000 | *1.29562ms* | *5.70121ms* | *0m9.371s* | *88.6%* |
| 10000 | *12.5814ms* | *62.1256ms* | *1m32.111s* | *87.14%* |

1. **Optimization Number #0: Streams (4 points)**
   1. How does this optimization work in theory? Expected behavior?

Creating multiple streams can optimize my original design by pipelining the loading/execution/outputting stage of my entire design. This means that while I am loading data to be processed by the kernel function, there should be a portion of input data that had already been memcpy-ed over to device memory that is being executed. And all while this is happening, another previous execution should be outputting its value and copying its data back to host. Since execution is happening simultaneously with loading/outputting data, I would expect faster optimes.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I first initialized pinned memory. Pinned memory is necessary for streams so that the pipelined effect which I explained previously can take place. Then I set the value of total streams I would have in my process. I set this value to be 4 because I thought this was the optimal number of streams. Since having 3 streams gave me errors, I put 4 because there could only be 3 operations happening simultaneously. These operations include loading the data via memcpy, executing the data in the kernel, and outputting the data via memcpy again. After creating my streams, I iterated through each of my streams and used cudamemcpyasync to copy data over the device memory. I also called the kernel function and copied the output back to host all in the same for loop. I also had to account for the input and output offset to make sure data was being copied to/from where it should be. After the for loop, I freed memory and destroyed the streams. To further mention, I also accounted for pinned memory to make sure the pipelining process will work.



A black screen with white text

Description automatically generated

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

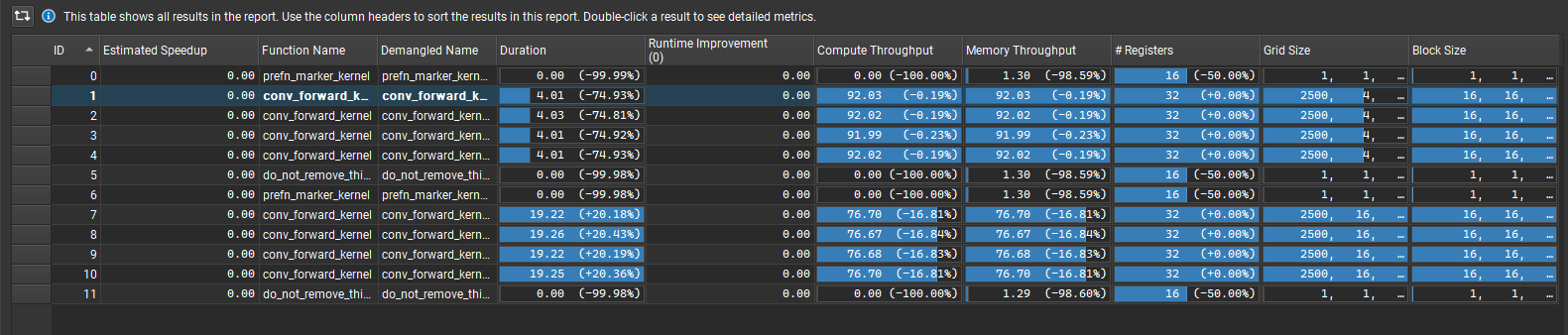
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.001733ms* | *0.002365ms* | *0m1.260s* | *86%* |
| 1000 | *0.002204ms* | *0.002204ms* | *0m9.610s* | *88.6%* |
| 10000 | *0.002024ms* | *0.002024ms* | *1m32.583s* | *87.14%* |

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



Although I am unable to see the optimes due to everything being done inside the prolog function, I am able to prove that my implementation works by looking ad Nvidia Nsight Systems where once can easily see the memcpy to device, the calculations, and memcpy back to host being done in the same cycle and these calls are being pipelined. The GPU throughput and memory usage have gone down slightly from the original Milestone 2 code (Looking at the baseline) but this can be seen as negligible due to the benefits which are seen from the pipelined effect thanks to using pinned memory and async memcpy API calls. So yes, This performance matched my expectation.

* 1. Does this optimization synergize with any other optimizations? How and why?

Streams pair well with Tiled shared memory convolution since streams do not affect the overall calculations done within the kernel itself, it only splits up the input data to be pipelined to the kernel. It will also synergize with input matrix unrolling and tiled matrix multiplication for the same reasons stated above. It will go well with constant memory as this will be loaded in even before the streams are used and tuning with restrict and loop unrolling will also benefit from this optimization because one is done inside the kernel while the other is done on the host side. FP16 will also synergize well since this only requires a declaration change.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Textbook Chapter 18 page 18.

1. **Optimization Number 1: Tiled (Shared Memory) Convolution (2 points)**
   1. How does this optimization work in theory? Expected behavior?

By loading shared memory first and reading from shared memory during computations, I can effectively reduce the number of times I am accessing device memory to read data. It takes less time to read from shared memory compared to device memory. As a result, I should see optimes going down.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I first set the size of shared memory (as seen in the textbook) inside my conv\_forward\_gpu function. Once that size was set, I iterated through each Channel, and the width/height of the tile to load the input data to shared memory. Once loaded, I was able to iterate through the mask dimensions and multiply the corresponding input data with the mask. After summing up the products, I wrote the final value (acc) to the output only if its coordinates were inside the output height/width dimensions.

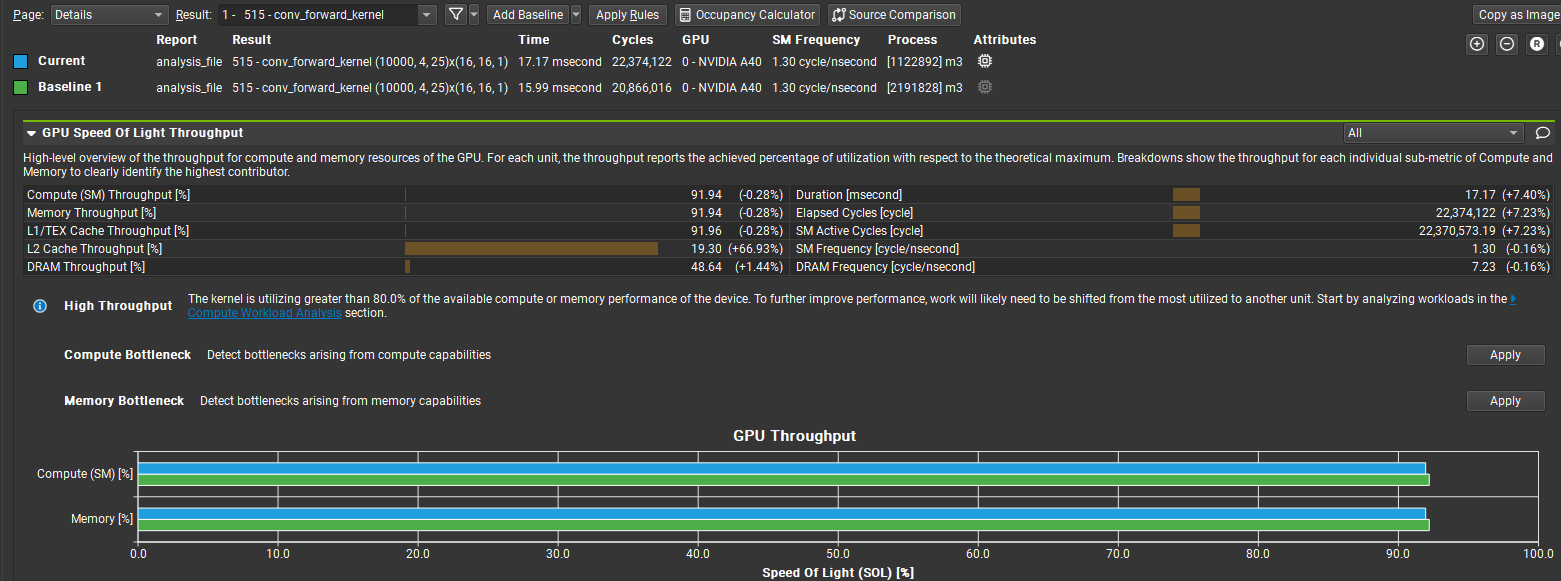


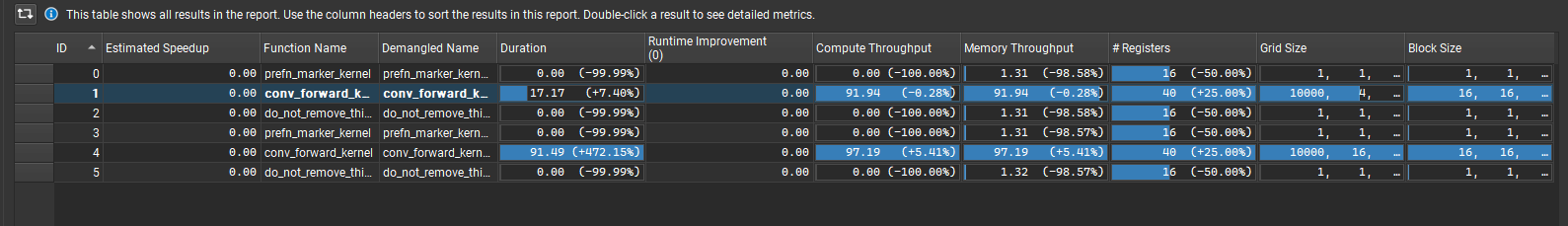
A screen shot of a computer program

Description automatically generated

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.198392ms* | *0.704841ms* | *0m1.257s* | *86%* |
| 1000 | *1.37182ms* | *6.93111ms* | *0m9.392s* | *88.6%* |
| 10000 | *13.1363ms* | *69.2941ms* | *1m32.397s* | *87.14%* |





This optimization did not live up to the expectations that I had for it. Despite the GPU throughput and compute throughput matching that of the Milestone 2 code, I believe that the optimes had gone up due to my placement of the if statement for checking if the specific thread was inside the H\_out and W\_out bounds. If I checked earlier before the loading of shared memory, I believe the optimes would have been more promising as no computation would be done unless that specific thread was linked to an output cell that was part of the output matrix. So, by moving the conditional to before the start of the for loops, I believe the optimes would have seen an improvement, but from looking at the raw data above, the current optimes did not meet my expectations.

* 1. Does this optimization synergize with any other optimizations? How and why?

This optimization pairs well will streams as this is taken care of before the calling of the kernel, Input matrix unrolling and tiled matrix multiplication because this optimization already used shared memory, constant memory as this is loaded before the kernel call, tuning with restrict and loop unrolling because this will help unroll the loops for loading in shared memory, and floating point because this only requires a change in declaration (\_\_half).

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Textbook Chapter 4 Page 23.

1. **Optimization Number 2: Input Matrix Unrolling & Tiled Matrix Multiplication using Shared Memory (3 points)**
   1. How does this optimization work in theory? Expected behavior?

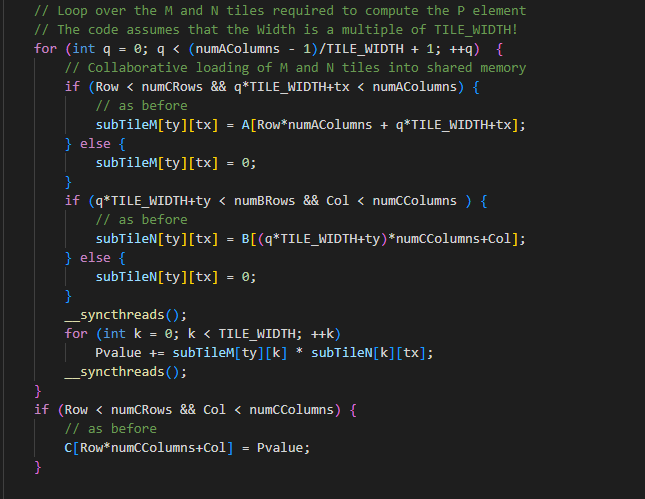
This optimization works by completely unroll the input matrix so tiled, shared memory multiplication could be using to make optimes lower. The Mask is done multiplied by an unrolled matrix of size K\*K\*Channels by H\_out\*W\_out where H\_out = Height - K + 1 and W\_out = Width – K +1. This optimization should make optimes lower as the use of shared memory and tiling results in less reads to Device memory, however, because I have done CPU unrolling and I had to move input data/mask data from Host and Device and back to Host again to measure optimes and having to create new variables to cudaMalloc, the optime will be significantly worse.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

First, I created an unrolling matrix CPU function that runs on host, so that I could properly unroll the input data to prepare for matrix multiplication. I unrolled once per batch to reduce the amount of memory I had to cudaMalloc in device Memory. Once unrolled, I copied this new preprocessed data to device memory each iteration of the for loop. Inside that same for loop, I called the matrix multiplication kernel function that has the exact same functionality as the one I had from lab 3. After that kernel function, I copied data back to host while making sure to account for the offset of each output. Unfortunately, to get my output data to the epilog function, I had to make another memcpy call to get it back to device memory and retrieve it again in the epilog.

A screen shot of a computer code

Description automatically generated

A screen shot of a computer code

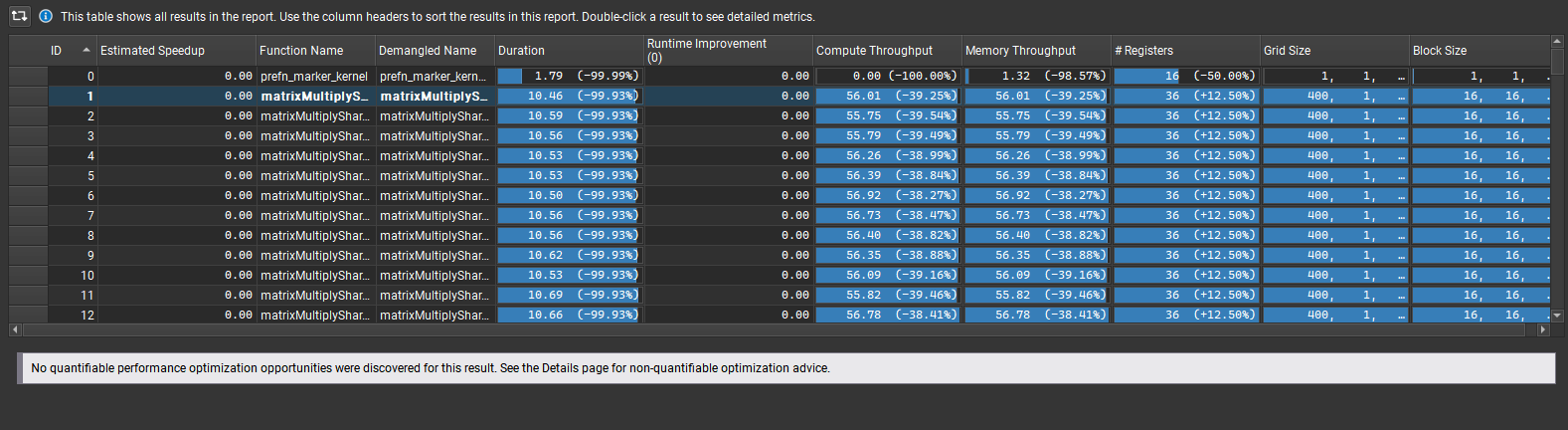
Description automatically generated

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *219.156ms* | *161.186ms* | *0m2.266s* | *86%* |
| 1000 | *2196.89ms* | *1604.81ms* | *0m13.617s* | *88.6%* |
| 10000 | *22023.9ms* | *16017.8ms* | *2m12.301s* | *87.14%* |

A screenshot of a video

Description automatically generated



Even before running this program, I knew that the optimes would be terrible. I had to make several memcpy calls to move data from host to device and another memcpy call to move the data back to host in order to have optimes measured properly. Unfortunately, I was not able to put all my code inside prolog or else the optimes would not have been measured. But, even if all my code was inside prolog and the optimes were measured properly from there would not have resulted in good optime results. I am partitioning data by Batch which means I will have a call to kernel every single batch and because I’m using CPU to unroll my input data, I have to use memcpy to copy my host data over to the device every time I am iterating into a new batch. The plethora of memcpy calls is what is resulting my low optime. In addition to these many memcpy calls, I am calling the matrixmultiply kernel on very small partitions of input data which can also result in slower optimes. But this was completely expected so I can say that these optimes met my expectations. Fortunately, I know that my implementation is still correct because I am getting the same accuracy results as before.

* 1. Does this optimization synergize with any other optimizations? How and why?

This optimization synergies well with streaming, constant memory, floating point, and tiled shared memory convolution. This is because all 4 of these optimizations do not effect the main calculation or program execution. They are either done before the calling of the kernel or inside the kernel before the execution of the main part of the program. It will not synergize well with Tuning and restrictions because as I have said in later optimizations, the inner for loop can be unrolled, however, there will not be a significant change as only one for loop will see the benefits of the pragma unroll opposed to other optimizations.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Textbook Chapter 4 page 16

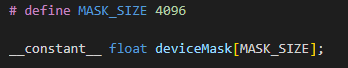
1. **Optimization Number 4: Weight Matrix (Kernel) in Constant Memory (1 point)**
   1. How does this optimization work in theory? Expected behavior?

This Optimization should work by extracting the mask from device memory all at once and only accessing constant memory for the duration of the execution. Reading data from device memory can be very taxing in terms of optime, but by reading it all at once and writing to constant memory will mean that future reads will all be directed to constant memory and not device memory. The optime should go down slightly.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

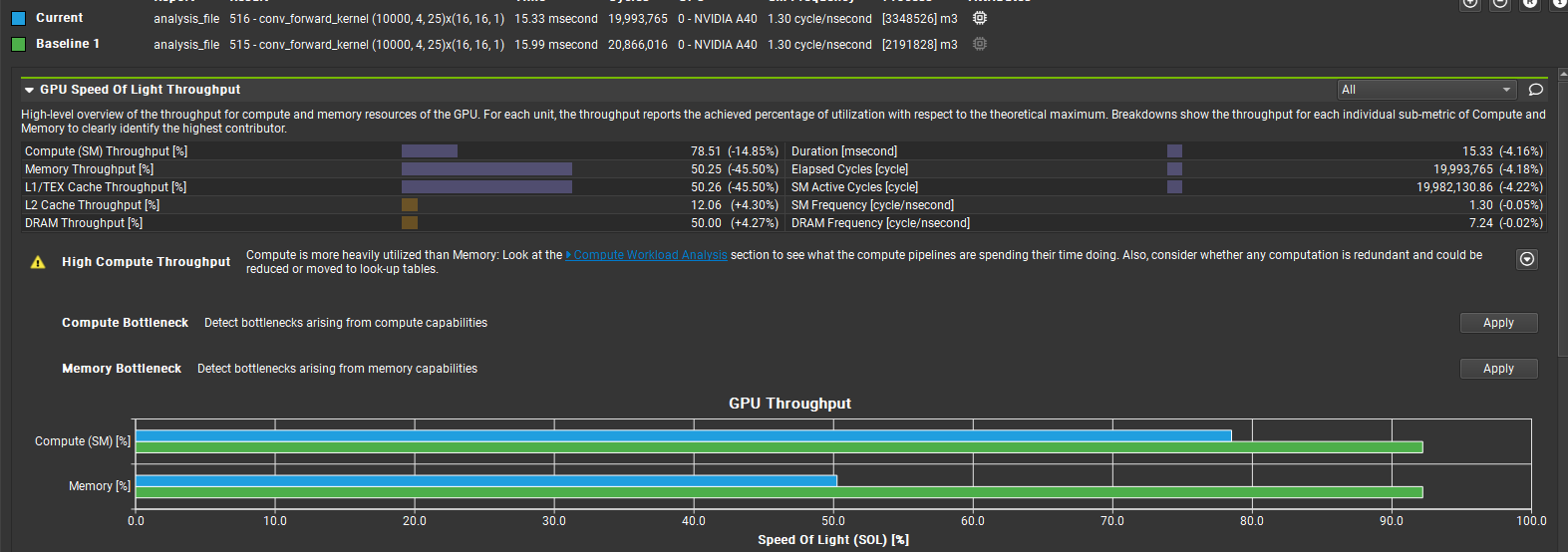
I implemented constant memory by first setting the size of the mask to be a big number (which I chose to be 4096) because I needed the size of constant memory to be set at runtime. Then I used cudamemcpytosymbol in order to populate constant memory and ran the rest of the program as the base was run for milestone 2.

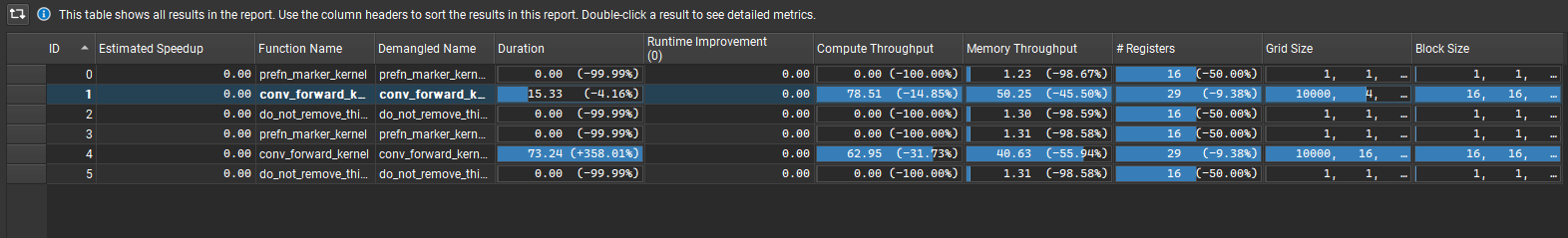




* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.145213ms* | *0.492415ms* | *0m1.254s* | *86%* |
| 1000 | *1.20572ms* | *5.46756ms* | *0m9.529s* | *88.6%* |
| 10000 | *12.1421 ms* | *59.882 ms* | *1m31.439s* | *87.14%* |





The Constant memory optimization lived up to my expectations because compared to the Baseline 1 (Milestone 2 implementation), the times shown in Nsight compute is faster and the optimes reflect. The GPU memory bandwidth is lacking however due to the usage of constant memory, but that was to be expected since there are less reads to device memory by using constant memory which was loaded in straight from host\_mask.

* 1. Does this optimization synergize with any other optimizations? How and why?

To keep it short, this optimization can synergize with every other optimization that I did because it does not effect the overall execution flow of the program. The mask can simply be loaded in on the host side and the execution inside the Kernel will not change. It will simply access constant memory as opposed to device memory when it is trying to multiply the inputs with the mask.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Textbook Chapter 4 page 9

1. **Optimization Number 5: Tuning with Restrict and Loop Unrolling (2 points)**
   1. How does this optimization work in theory? Expected behavior?

By performing loop unrolling, I’m able to tell the compiler to manually unroll the inner loops by a factor of X. X is the number that I can choose. After bouncing between 3, 5, and 7 based on the size of the Mask, I found 7 to have the best performance. Loop unrolling will result in an increase in code size, but it will relieve the loop overhead and prevent loop control instructions. Also, by using the \_\_restrict\_\_ keyword on the input pointers, I can tell the compiler that I will be accessing the arrays only through the input parameters. This can result in optimizations in that the compiler will be able to avoid redundant memory loads and stores. This is useful since many loads are done during convolution to device memory. This will also result in a lower optime.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I implemented Loop unrolling by adding a pragma unroll 7. As I have stated before, I chose this number based off the size of the masks and I realized 7 was the best out of the 3 given choices. I placed this line right above the loops that iterate through the Width of the Mask. I also added \_\_restrict\_\_ before the 3 device input pointers to tell the compiler that I will be accessing this array only through this pointer. The rest of the code is identical to that of milestone 2.

A computer screen shot of text

Description automatically generated

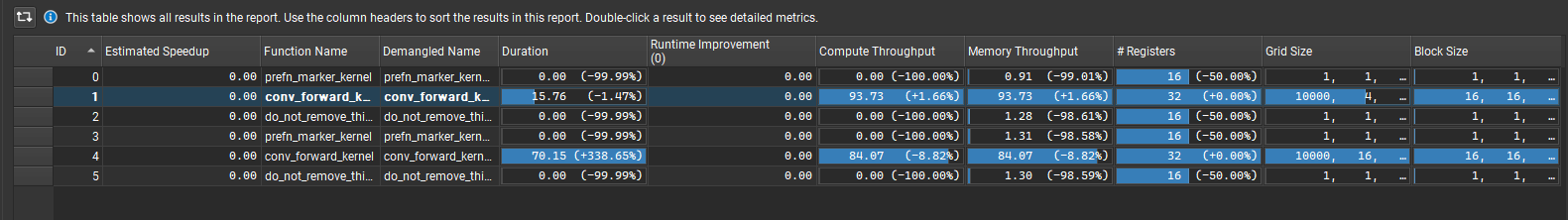


* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.204874ms* | *0.522428ms* | *0m1.294s* | *86%* |
| 1000 | *1.29384ms* | *5.16032ms* | *0m9.570s* | *88.6%* |
| 10000 | *12.2954ms* | *54.6668ms* | *1m32.693s* | *87.14%* |

A screen shot of a computer

Description automatically generated



This optimization lived up to my expectations because the optimes did go down significantly. This can be seen through Nsight Compute because the GPU is computing more efficiently and the GPU memory is also being used efficiently. This can be seen by how both bars are closer to the 100% marker compared to the baseline or M2 implementation. By unrolling the inner 2 for loops by a factor of 7, I did decrease the overall optimes, but I did this at the cost of increasing the total code area. However, in this scenario, we can see that increasing the area in this manner was beneficial as our optime went down by a total of 12ms. Yes, this optimization lived up to my expectations.

* 1. Does this optimization synergize with any other optimizations? How and why?

This optimization pairs well with Streams because it does not effect the overall execution flow of the program, Tiled Shared memory since loading the shared memory can be unrolled and optimized, constant memory because this also does not affect the execution flow of the program, and floating point precision because this only requires a change in declaration of the input, but it will not pair well with optimization #2/6 (Input matrix unrolling and tiled matrix multiplication) because there aren’t that many loops that can be unrolled optimally. One loop that holds the multiplication aspect can be unrolled but it won’t give optimal results as shown in optimization #5.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Nvidia Developer forums regarding Loop unrolling + Restrict keyword.

1. **Optimization Number 6: Sweeping Various Parameters to find best values (2 points)**
   1. How does this optimization work in theory? Expected behavior?

This optimization works by using trial and error to find the best TILE\_WIDTH for Tiled Matrix Multiplication. This optimization will lower optimes for sure because by scanning through ~10 different TILE sizes I will be able to find one that is divisible by the Width/Height of the input data to make sure less unnecessary computation is done to get the final result. I expect the optimes to decrease further then after my previous optimization 2.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

The code itself is the same as optimization 2 except I am now changing the TILE\_WIDTH size every test of 10k batches.

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

Batch size will be kept at 10K and profiling data will be taken from the best TILE\_WIDTH result and compared to Optimization 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| TILE\_WIDTH | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 4 | *22820.4 ms* | *16602.1 ms* | *2m10.190s* | *87.14%* |
| 8 | *22643.2 ms* | *16530.5 ms* | *2m9.692s* | *87.14%* |
| 11 | *22453.1ms* | *16340.4ms* | *2m8.325s* | *87.14%* |
| 13 | *22313.5ms* | *16213.8ms* | *2m9.724s* | *87.14%* |
| 15 | *22089.9 ms* | *16137.3 ms* | *2m10.444s* | *87.14%* |
| 16 | *22086.9ms* | *16128.8ms* | *2m12.301s* | *87.14%* |
| 21 | *22080.1 ms* | *16117.7 ms* | *2m7.233s* | *87.14%* |
| 25 | *22060.2ms* | *16135.3ms* | *2m8.123s* | *87.14%* |
| 29 | *22034.4ms* | *16124.6 ms* | *2m10.907s* | *87.14%* |
| 32 | *21993.7 ms* | *16104.6 ms* | *2m9.044s* | *87.14%* |

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

A graph of different colored bars

Description automatically generated

These statistics that one can see above is what I predicted. I predicted greater the shared memory, the more optimal the results will be which will result in a less optime. The optimes will be much less because as the size of shared memory increases per block, less calls will be made to device memory to extract/read data if the data is not already residing inside shared memory. This is why there is a linear downward trend when it comes to optime vs TILE\_WIDTH. Although I have already stated this inside OP#2, I would like to clarify that although these optimes are not optimal by any means, these results are correct as they produce the correct accuracy. The optimes are extremely high because there is a memcpy done for each iteration done per batch and more memcpy API calls were made to move the data to the correct function before executing,

* 1. Does this optimization synergize with any other optimizations? How and why?

Because the default size of the TILE\_WIDTH (16), which I had previously in optimization 2 happened to be the most optimal TILE\_WIDTH in my implementation, this question can be answered in the same manner. This optimization can pair well with streams since this does not disrupt the flow of the execution, Tiled shared memory because shared memory has already been incorporated, constant memory because having the mask loaded into constant memory does not affect the execution flow, and floating point precision since this only requires a change in declaration. It will probably not go well with loop unrolling with restriction because there is not an effective location that would benefit from having its loop unrolled. One could have it right before the syncThreads() call right before the multiplication but that would not result in much optimization compared to the 2 loops which were unrolled inside my optimization #5.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Textbook Chapter 4 page 16

1. **Optimization Number 10: Fixed Point (FP16) Arithmetic Implementation (1 point)**
   1. How does this optimization work in theory? Expected behavior?

This optimization works by reducing the precision of the input data to half (16 bits). By reducing the precision, accuracy should go down slightly while making the overall program run faster. I would expect the total optime to decrease because less calculations are being done.

* 1. How did you implement your code? Explain thoroughly and show code snippets.

I implemented my code by lowering the precision of the input data coming into the kernel function using \_\_half and \_\_float2half() API calls. Once I reduced the precision of the input values, I multiplied the inputs together and summed them to acc. The rest of the code is as same as milestone 2.

A computer screen with text

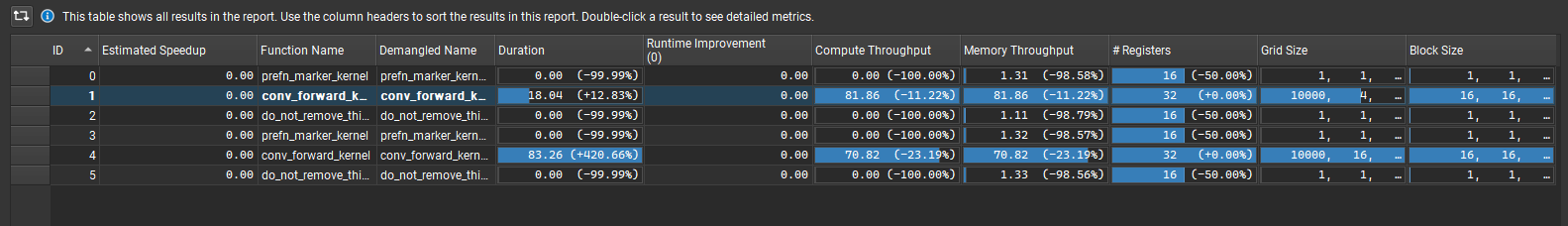
Description automatically generated

* 1. Did the performance match your expectation? Show your analysis results using profiling tools.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy |
| 100 | *0.200065ms* | *0.584314ms* | *0m1.683s* | *86%* |
| 1000 | *1.43338 ms* | *6.08926 ms* | *0m9.650s* | *88.7%* |
| 10000 | *13.9363 ms* | *66.4284 ms* | *1m30.011s* | *87.16%* |

A screenshot of a computer

Description automatically generated



FP16 did not meet my expectations as I expected much lower optimes after decreasing the precision of my input. Looking at the Nsight Compute, it does not seem like the Compute resources were being used efficiently, and this could be due to our input data being reduced by half. As a result, some compute resources which were partitioned to execute on data is no longer being utilized. The same goes for the memory bandwidth. The optimes are still low, however, it is higher then what I had for milestone 2.

* 1. Does this optimization synergize with any other optimizations? How and why?

This optimization can synergize with all my other optimizations because I am simply reducing the precision of the input values. Reducing the precision of the inputs does not disrupt the execution flow of streams, tiled shared memory, unrolling with matrix multiplication, constant memory, or tuning with loop unrolling.

* 1. List your references used while implementing this technique. (you must mention textbook pages at the minimum)

Nvidia CUDA Toolkit Documentation 1.2.6 Half Precision Conversion and Data Movement

Total Points = 4 + 2 + 3 + 1 + 2 + 2 + 1 = 15 points