|  |  |
| --- | --- |
| **Name:** | **Rohan Sreenivasan** |
| **NetID:** | *Rohanjs3* |
| **Section:** | *AL1 - CS 483 - Kindratenko*  **\*\*\*I have submitted my Project Code with a folder called optimizations with 3 files: tilingwithconstmem.cu, floatingpointconstmem.cu, streaming.cu, atomics.cu. These correspond to Optimization 1, 2, 3, 4 respectively. TOTAL POINTS = 12** |

**ECE 408/CS483 Milestone 3 Report**

|  |
| --- |
| 1. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. Note: **Do not** use batch size of 10k when you profile in *--queue rai\_amd64\_exclusive*. We have limited resources, so any tasks longer than 3 minutes will be killed. Your baseline M2 implementation should comfortably finish in 3 minutes with a batch size of 5k (About 1m35 seconds, with nv-nsight). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.183774 ms* | *0.656389 ms* | *0m1.618s* | *0.86* | | 1000 | *1.73373 ms* | *6.42681 ms* | *0m10.644s* | *0.886* | | 5000 | *8.54613 ms* | *32.545 ms* | *0m50.733s* | *0.871* | |
| 1. **Optimization 1: Tiled and Shared memory convolution** |
| * 1. **Which optimization did you choose to implement and why did you choose that optimization technique.** |
| I chose Tiled and shared memory convolution so that not all accesses come from global memory. I thought that this would improve our memory throughput |
| * 1. **How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?**   Tiling is a technique to efficiently utilize the shared memory in CUDA by breaking down large arrays into smaller 'tiles'. This code implements tiling for the convolution operation in the following way:  It divides the input and mask data into smaller blocks or 'tiles' (as indicated by TILE\_W1).  Each block of threads processes a small portion of the input and mask data at a time, storing it in the fast shared memory (sharedTile).  The convolution computation is then performed on these tiles, reducing global memory accesses and improving memory bandwidth utilization.  After the computation, results are written back to the output array in global memory. This should limit global memory accesses and therefore reduce memory bandwidth load. |
|  |
| * 1. **List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.286888 ms* | *1.01882 ms* | 0m1.773s | *0.86* | | 1000 | *2.79965 ms* | *10.1827 ms* | 0m10.361s | *0.886* | | 5000 | *13.9361 ms* | *51.1966 ms* | 0m51.615s | *0.871* | |  |  |  |  |  | |
| * 1. **Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).**   While the performance time or accuracy did not change much, we can see that memory utilization was much lower and more efficient. We can clearly see that we have around 12 FLOPS / S for our memory performance. This is much better than our initial baseline as seen below. |
| * 1. What references did you use when implementing this technique?  Used these developer docs to learn more about tiling for convolution:  <https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf> |
| 1. **Optimization 2: *Fixed Point*** |
| 1. **Which optimization did you choose to implement and why did you choose that optimization technique.** |
| I chose to implement the fixed-point technique to that we can optimize the memory bandwidth usage by loading halves to constant memory. Loading halves instead of full 32 bit floats into memory should take up less space as well. |
| 1. **How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations?**   The optimization works by converting the input and mask values from single-precision (FP32) to half-precision (FP16) floating points. This is from the use of the \_\_float2half and \_\_half2float functions and the \_\_half data type in CUDA, which represents half-precision floating point numbers. This does not work very well with the other optimizations (tiling with shared memory) since we are now using constant memory rather than shared. |
|  |
| 1. **List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used).** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.407011 ms* | *0.888872 ms* | 0m1.604s | *0.86* | | 1000 | *1.95907 ms* | *8.83124 ms* | 0m10.516s | *0.887* | | 5000 | *8.77902 ms* | *31.1816 ms* | 0m49.988s | *0.8712* | |
| 1. **Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).**   The optimization did not change the OP times much but it did change the memory bandwidth a lot because now we are only loading fixed point 16 bits instead of 32 bits. We can see that our memory bandwidth decreased and is now only 100 FLOPS/S as seen in the graph below: |
| 1. What references did you use when implementing this technique? |
| Used cuda documentation relating to the \_\_half data type, the \_\_hfma function, and the conversion functions \_\_half2float and \_\_float2half. I used the following link:  <https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__ARITHMETIC.html> |

|  |
| --- |
| 1. **Optimization 3: Streaming**   ***(Delete this section blank if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| I added this optimization to utilize asynchronous memory copies and kernel launches with multiple streams to overlap data transfer and computation. This should decrease kernel execution time by allowing us to pipeline the memory transfer with kernel execution. |
| * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations? |
| I chose this optimization because I thought it was increase total execution time performance of the convolution kernel by taking advantage of pipelining on the host side. While we are doing the convolution in the kernel we are also transferring data to save time. |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *1.65506 ms* | *3.00758 ms* | 0m1.616s | *0.86* | | 1000 | *22.1546 ms* | *27.3709 ms* | 0m10.510s | *0.886* | | 5000 | *208.485 ms* | *148.336 ms* | 0m53.130s | *0.871* | |
| * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).   I did not see the performance I was expecting since the overall execution time actually increased rather than decreased. This is because I implemented streaming by having Batch number of streams and often times the batch number was fairly small in the testcases. Therefore, including streaming did not make a huge impact on the kernel performance. Below is a graph of our FLOPS /s which shows a major difference from the baseline. |
|  |
| * 1. What references did you use when implementing this technique?   I used the following cuda docs to create my code and I used methods like cudaStreamCreate to create our streams:  <https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__STREAM.html>  I also used the NSIGHT SYSTEMS Tool to verify my streaming optimization:  <https://developer.nvidia.com/nsight-systems/get-started>  After downloading this tool and running the profiling for just streaming we can see that my Cuda calls and MemCopys are being interleaved: |
| 1. **Optimization 4: *Input channel reduction: atomics***   ***(Delete this section blank if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| I implemented the atomics optimization to optimize memory usage in the convolution process. I wanted to make sure that the adds we do are single and indivisible operations, which is necessary since multiple threads write to the same values in memory. |
| * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations? |
| Instead of processing input channels sequentially, I wanted to paralleize over the channel dimension. I removed the loop where we iterate over channel and instead launched the kernel with number of channels in the z dimension Each thread computes a part of the convolution and accumulates the results using atomicAdd, ensuring data consistency and avoiding race conditions. I expected this to increase performance due to the more efficient use of the GPU. This also works well with other optimizations since we can easily change the way we can do adds with one line. |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.186928 ms* | *2.38933 ms* | *0m1.652s* | *0.86* | | 1000 | *1.54596 ms* | *7.97833 ms* | *0m13.293s* | *0.886* | | 5000 | *0m55.805s* | *0m54.740s* | *0m1.068s* | *0.871* | |
| * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).   I saw an increase in memory throughput performance. The main point of atomic adds was to improve the data integrity not necessarily improve overall kernel execution performance. I did hope that we could improve the kernel execution time by parallelizing over the channel dimension, but this was not the case. I observed a major improvement in FLOPS / S as seen below: |
| * 1. What references did you use when implementing this technique? |
| I used this link to learn more about atomicAdd and how to implement it:  <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> |