|  |  |
| --- | --- |
| **Name:** | *<Leelanarasimha Konatham>* |
| **NetID:** | *<lrk4>* |
| **Section:** | *<AL1>*  *ALL OPTIMIZATIONS ARE IN THE SAME FILE, COMMENTED* |

**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 | *<0.1856>* | *<0.65086>* | *<0m1.612s >* | *<.86>* | | 1000 | *<1.7162 ms>* | *<6.38038 ms>* | *<0m10.508s>* | *<.866>* | | 5000 | *<8.52429 ms>* | *<33.2263 ms>* | *<0m50.345s>* | *<.871>* | |
| 1. **Optimization 1: *<Weight matrix (kernel values) in constant memory (0.5 point)>*** |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose to implement the weight matrix (kernel values) in constant memory optimization, and I chose this optimization because we have gone over it in class and applied it previously.* |
| 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?   This optimization works by loading the convolution kernels as constant memory which takes fewer clock cycles to access than normal. This effectively reduces the number of accesses to global memory, and this should increase the performance of the forward convolution because we are now able to reuse some of the elements. This would help when you are using larger convolution filter sizes. This was one of my first optimizations implemented, but I think it would synergize with the atomic optimization since they both deal with global memory. |
|  |
| 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 | *<0.168839 ms>* | *<0.581316 ms>* | *<0m1.573s>* | *<.86>* | | 1000 | *<1.56246 ms>* | *<5.72464 ms>* | *<0m13.772s>* | *<.886>* | | 5000 | *<7.75067 ms>* | *<28.5194 ms>* | *<0m52.968s>* | *<.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).   This optimization was successful in improving performance because we can clearly see that the Op times reduced by a significant amount compared to the baseline Op times. The baseline Op times were around 41.5 ms, but the Op times after this optimization was implemented is around 36-37 ms. This is a clear difference and improvement so we can conclude that this optimization was successful in improving performance. In terms of the profiling and statistics, they are fairly similar to the baseline. However, the total execution time slightly increased. |
| *Running test case 1*  *B = 1 M = 3 C = 3 H = 224 W = 224 K = 3 S = 1*  *Running test case 2*  *B = 2 M = 3 C = 3 H = 301 W = 301 K = 3 S = 2*  *Running test case 3*  *B = 3 M = 3 C = 3 H = 196 W = 196 K = 3 S = 3*  *Running test case 4*  *B = 4 M = 3 C = 3 H = 239 W = 239 K = 3 S = 4*  *All test cases passed*  *Test batch size: 5000*  *Loading fashion-mnist data...Done*  *Loading model...Done*  *Conv-GPU==*  *Layer Time: 348.736 ms*  *Op Time: 7.86987 ms*  *Conv-GPU==*  *Layer Time: 255.551 ms*  *Op Time: 29.1105 ms*  *Test Accuracy: 0.871*  *Generating the /build/report1.qdstrm file.*  *Capturing raw events...*  *334648 total events collected.*  *Capturing symbol files...*  *Saving diagnostics...*  *Saving qdstrm file to disk...*  *Finished saving file.*  *Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.*  *Importing...*  *Importing [==================================================100%]*  *Saving report to file "/build/report1.qdrep"*  *Report file saved.*  *Please discard the qdstrm file and use the qdrep file instead.*  *Removed /build/report1.qdstrm as it was successfully imported.*  *Please use the qdrep file instead.*  *Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.*  *Exporting 334621 events:*  *0% 10 20 30 40 50 60 70 80 90 100%*  *|----|----|----|----|----|----|----|----|----|----|*  *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *Exported successfully to*  */build/report1.sqlite*  *Generating CUDA API Statistics...*  *CUDA API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *72.2 569205243 14 40657517.4 29129 315207142 cudaMemcpy*  *22.6 178287528 14 12734823.4 3425 175264459 cudaMalloc*  *4.7 36977037 10 3697703.7 3135 29084912 cudaDeviceSynchronize*  *0.3 2539107 20 126955.4 1897 454306 cudaFree*  *0.1 988783 6 164797.2 105733 186956 cudaMemcpyToSymbol*  *0.0 383975 10 38397.5 22355 130078 cudaLaunchKernel*  *Generating CUDA Kernel Statistics...*  *Generating CUDA Memory Operation Statistics...*  *CUDA Kernel Statistics (nanoseconds)*  *Time(%) Total Time Instances Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *100.0 36960316 6 6160052.7 6912 29081434 conv\_forward\_kernel*  *0.0 2944 2 1472.0 1376 1568 do\_not\_remove\_this\_kernel*  *0.0 2592 2 1296.0 1280 1312 prefn\_marker\_kernel*  *CUDA Memory Operation Statistics (nanoseconds)*  *Time(%) Total Time Operations Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *91.9 518056106 6 86342684.3 23200 314468242 [CUDA memcpy DtoH]*  *8.1 45927733 14 3280552.4 1152 24023938 [CUDA memcpy HtoD]*  *CUDA Memory Operation Statistics (KiB)*  *Total Operations Average Minimum Maximum Name*  *----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------*  *862672.0 6 143778.7 148.535 500000.0 [CUDA memcpy DtoH]*  *276206.0 14 19729.0 0.004 144453.0 [CUDA memcpy HtoD]*  *Generating Operating System Runtime API Statistics...*  *Operating System Runtime API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *33.3 52119553266 535 97419725.7 18620 100408243 sem\_timedwait*  *33.3 52082414845 535 97350308.1 27338 100445480 poll*  *22.1 34507040069 69 500102030.0 500012632 500383271 pthread\_cond\_timedwait*  *11.2 17563232315 2 8781616157.5 4416446752 13146785563 pthread\_cond\_wait*  *0.1 84338979 933 90395.5 1024 18051726 ioctl*  *0.0 21354335 9427 2265.2 1053 19067 read*  *0.0 3615966 98 36897.6 1263 1738085 mmap*  *0.0 1139451 101 11281.7 4123 25240 open64*  *0.0 321848 19 16939.4 4190 49013 fopen64*  *0.0 289363 5 57872.6 40420 72073 pthread\_create*  *0.0 135473 3 45157.7 41732 50454 fgets*  *0.0 116111 26 4465.8 1388 38296 fopen*  *0.0 99232 15 6615.5 3289 12435 fflush*  *0.0 78829 18 4379.4 1032 12553 munmap*  *0.0 77713 26 2989.0 1067 6075 fclose*  *0.0 69405 15 4627.0 2273 7586 write*  *0.0 30758 5 6151.6 4247 8608 open*  *0.0 15549 2 7774.5 5845 9704 pthread\_cond\_signal*  *0.0 13377 2 6688.5 5207 8170 socket*  *0.0 10372 1 10372.0 10372 10372 pipe2*  *0.0 7064 1 7064.0 7064 7064 connect*  *0.0 5471 2 2735.5 1558 3913 fwrite*  *0.0 4373 4 1093.3 1039 1134 fcntl*  *0.0 1993 1 1993.0 1993 1993 bind*  *0.0 1333 1 1333.0 1333 1333 listen*  *Generating NVTX Push-Pop Range Statistics...*  *NVTX Push-Pop Range Statistics (nanoseconds)*  *real 1m16.022s*  *user 1m22.322s*  *sys 0m18.437s*  *A screenshot of a computer  Description automatically generated*  *A screenshot of a graph  Description automatically generated* |
| 1. What references did you use when implementing this technique? |
| *I used one of the earlier lectures that talked about some of these topics such as memory architecture and also looked at some of the previous MPs.* |

|  |
| --- |
| 1. **Optimization 2: *<Sweeping various parameters to find best values (block sizes, amount of thread coarsening)> .5 points Stacked on Optimization 1***   ***(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 chose to implement the sweeping various parameters to find best values, I chose to implement this optimization because I knew it would easily improve the overall performance of the forward convolution once you found values that were optimal for your kernel.* |
| * 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? |
| *This optimization works by testing different values for things such as the tile sizes and the amount of thread coarsening. Different combinations of the parameters leads to different performance results, so finding an optimal number is crucial and would be an easy way to improve the overall performance of the forward convolution. The number of threads in a block and the number of blocks in the grid affected the times the most so I decided to test out different values for this since other parameters I changed did not really have a significant impact on the performance. This does synergize with the previous optimization of the weight matrix (kernel values) in constant memory and both of these optimizations together can reduce the memory access and lead to a better throughput.* |
| * 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 | *<0.183546 ms>* | *<0.510442 ms>* | *<0m1.610s>* | *<.86>* | | 1000 | *<1.73735 ms>* | *<5.16881 ms>* | *<0m12.377s>* | *<.886>* | | 5000 | *<8.61244 ms>* | *<25.2708 ms>* | *<0m56.506s>* | *<.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). |
| *Implementing this optimization was successful in improving the overall performance of the forward convolution because the Op times decreased from the previous optimization of around 36 ms to 33-34 ms. Although it is not a major significant decrease, it is still an improvement in terms of overall performance. I was able to test out different values and concluded that using a TILE\_WIDTH of 20 was the most optimal and I will be using this for the rest of the project along with the previous optimization with the weight matrix. We can see that the compute throughput and the memory throughput both increased after implementing this optimization compared to the previous one. However, the total execution time increased by a little bit again.*  *Running test case 1*  *B = 1 M = 3 C = 3 H = 224 W = 224 K = 3 S = 1*  *Running test case 2*  *B = 2 M = 3 C = 3 H = 301 W = 301 K = 3 S = 2*  *Running test case 3*  *B = 3 M = 3 C = 3 H = 196 W = 196 K = 3 S = 3*  *Running test case 4*  *B = 4 M = 3 C = 3 H = 239 W = 239 K = 3 S = 4*  *All test cases passed*  *Test batch size: 5000*  *Loading fashion-mnist data...Done*  *Loading model...Done*  *Conv-GPU==*  *Layer Time: 314.404 ms*  *Op Time: 8.70131 ms*  *Conv-GPU==*  *Layer Time: 247.96 ms*  *Op Time: 25.4823 ms*  *Test Accuracy: 0.871*  *Generating the /build/report1.qdstrm file.*  *Capturing raw events...*  *334476 total events collected.*  *Capturing symbol files...*  *Saving diagnostics...*  *Saving qdstrm file to disk...*  *Finished saving file.*  *Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.*  *Importing...*  *Importing [==================================================100%]*  *Saving report to file "/build/report1.qdrep"*  *Report file saved.*  *Please discard the qdstrm file and use the qdrep file instead.*  *Removed /build/report1.qdstrm as it was successfully imported.*  *Please use the qdrep file instead.*  *Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.*  *Exporting 334409 events:*  *0% 10 20 30 40 50 60 70 80 90 100%*  *|----|----|----|----|----|----|----|----|----|----|*  *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *Exported successfully to*  */build/report1.sqlite*  *Generating CUDA API Statistics...*  *CUDA API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *70.4 527796410 14 37699743.6 37251 283627257 cudaMemcpy*  *22.4 168083110 14 12005936.4 3523 164900799 cudaMalloc*  *4.6 34194079 10 3419407.9 2912 25452998 cudaDeviceSynchronize*  *2.2 16331694 10 1633169.4 21812 16087596 cudaLaunchKernel*  *0.3 2560799 20 128039.9 1876 443482 cudaFree*  *0.1 455697 6 75949.5 59699 87207 cudaMemcpyToSymbol*  *Generating CUDA Kernel Statistics...*  *Generating CUDA Memory Operation Statistics...*  *CUDA Kernel Statistics (nanoseconds)*  *Time(%) Total Time Instances Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *100.0 34167035 6 5694505.8 7616 25449981 conv\_forward\_kernel*  *0.0 2848 2 1424.0 1408 1440 do\_not\_remove\_this\_kernel*  *0.0 2688 2 1344.0 1344 1344 prefn\_marker\_kernel*  *CUDA Memory Operation Statistics (nanoseconds)*  *Time(%) Total Time Operations Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *92.5 484583139 6 80763856.5 12640 282881484 [CUDA memcpy DtoH]*  *7.5 39400273 14 2814305.2 1152 20449155 [CUDA memcpy HtoD]*  *CUDA Memory Operation Statistics (KiB)*  *Total Operations Average Minimum Maximum Name*  *----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------*  *862672.0 6 143778.7 148.535 500000.0 [CUDA memcpy DtoH]*  *276206.0 14 19729.0 0.004 144453.0 [CUDA memcpy HtoD]*  *Generating Operating System Runtime API Statistics...*  *Operating System Runtime API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *33.3 51854542849 533 97288072.9 20541 100573210 sem\_timedwait*  *33.3 51789504174 532 97348692.1 35110 100569448 poll*  *22.2 34508917134 69 500129233.8 500071409 500377772 pthread\_cond\_timedwait*  *11.1 17313192902 2 8656596451.0 4468691370 12844501532 pthread\_cond\_wait*  *0.1 78419292 934 83960.7 1001 15592427 ioctl*  *0.0 21511729 9424 2282.7 1129 19548 read*  *0.0 2743460 98 27994.5 1097 934919 mmap*  *0.0 1142564 101 11312.5 4750 26917 open64*  *0.0 361773 19 19040.7 4036 55358 fopen64*  *0.0 303042 5 60608.4 36296 83581 pthread\_create*  *0.0 244381 26 9399.3 1136 169320 fopen*  *0.0 136655 3 45551.7 41775 50916 fgets*  *0.0 104839 17 6167.0 1002 10671 fflush*  *0.0 94712 27 3507.9 1019 9438 fclose*  *0.0 83836 17 4931.5 1657 15352 munmap*  *0.0 71593 15 4772.9 1965 8350 write*  *0.0 32530 5 6506.0 4321 8849 open*  *0.0 18262 2 9131.0 5258 13004 socket*  *0.0 9105 2 4552.5 4131 4974 pthread\_cond\_signal*  *0.0 8947 3 2982.3 1082 6634 fwrite*  *0.0 7923 1 7923.0 7923 7923 connect*  *0.0 6592 1 6592.0 6592 6592 pipe2*  *0.0 3599 3 1199.7 1169 1245 fcntl*  *0.0 1479 1 1479.0 1479 1479 bind*  *Generating NVTX Push-Pop Range Statistics...*  *NVTX Push-Pop Range Statistics (nanoseconds)*  *real 1m16.534s*  *user 1m23.258s*  *sys 0m19.153s*  *A screenshot of a computer  Description automatically generated*  *A screenshot of a graph  Description automatically generated* |
| * 1. What references did you use when implementing this technique?   I used the previous lectures and MPs as well as basic knowledge that different values will result in different performance results, so testing for the best one is important. |
| 1. **Optimization 3: *<Input channel reduction: atomics (2 point)> Stacked on Optimization 1, 2*** |
|  |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
|  |
| *I chose to implement the Input channel reduction: atomics optimization. I chose to implement this optimization because we have recently used it in one of the MPs as well as learned about it in lecture. It is also a very easy optimization to implement for our kernel.*   * 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?   This optimization works by allowing for multiple threads to write to the same memory location at nearly the same time using the atomic operation. It allows you to make sure that a value is not getting overwritten by another thread when any thread writes a value to global memory. This means that the input channels can perform parallelly in a sense. Hence, I believe that this optimization will increase the performance of the forward convolution. This optimization synergizes well with the previous two optimizations, the weight matrix (kernel values) in constant memory and sweeping optimization because they are independent and they have different benefits to the overall performance of the forward convolution. |
| * 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 | *<0.186657 ms>* | *<0.517039 ms>* | *<0m1.833s>* | *<.86>* | | 1000 | *<1.76218 ms>* | *<5.20718 ms>* | *<0m12.734s>* | *<.886>* | | 5000 | *<8.64758 ms>* | *<25.3075 ms>* | *<0m56.376s>* | *<.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).   Implementing this optimization was not very helpful in improving the performance of the forward convolution because we can see that most of the statistics are the same. For example, you can look at the nsys statistics and nsight compute image and compare them to see that there is not much of a difference. Furthermore, looking at the Optimes, we can see that they are around the same at 33-34 ms. The performance did not really change because there was a small number of input channels, so the effects of the atomic optimization were not very noticeable. In terms of the statistics and profiling, we can see that the they stayed fairly similar.  Running test case 1  B = 1 M = 3 C = 3 H = 224 W = 224 K = 3 S = 1  Running test case 2  B = 2 M = 3 C = 3 H = 301 W = 301 K = 3 S = 2  Running test case 3  B = 3 M = 3 C = 3 H = 196 W = 196 K = 3 S = 3  Running test case 4  B = 4 M = 3 C = 3 H = 239 W = 239 K = 3 S = 4  All test cases passed  Test batch size: 5000  Loading fashion-mnist data...Done  Loading model...Done  Conv-GPU==  Layer Time: 326.153 ms  Op Time: 8.75162 ms  Conv-GPU==  Layer Time: 277.471 ms  Op Time: 25.3991 ms  Test Accuracy: 0.871  Generating the /build/report1.qdstrm file.  Capturing raw events...  334854 total events collected.  Capturing symbol files...  Saving diagnostics...  Saving qdstrm file to disk...  Finished saving file.  Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.  Importing...  Importing [==================================================100%]  Saving report to file "/build/report1.qdrep"  Report file saved.  Please discard the qdstrm file and use the qdrep file instead.  Removed /build/report1.qdstrm as it was successfully imported.  Please use the qdrep file instead.  Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.  Exporting 334788 events:  0% 10 20 30 40 50 60 70 80 90 100%  |----|----|----|----|----|----|----|----|----|----|  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Exported successfully to  /build/report1.sqlite  Generating CUDA API Statistics...  CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  64.6 575572083 14 41112291.6 31654 291694188 cudaMemcpy  27.2 242554935 14 17325352.5 3200 238445349 cudaMalloc  3.9 34351668 10 3435166.8 23859 34093224 cudaLaunchKernel  3.8 34154672 10 3415467.2 3342 25363995 cudaDeviceSynchronize  0.3 2876765 20 143838.3 1734 472944 cudaFree  0.1 1006440 6 167740.0 115851 192980 cudaMemcpyToSymbol  Generating CUDA Kernel Statistics...  Generating CUDA Memory Operation Statistics...  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  100.0 34122857 6 5687142.8 7744 25359640 conv\_forward\_kernel  0.0 2848 2 1424.0 1408 1440 do\_not\_remove\_this\_kernel  0.0 2624 2 1312.0 1280 1344 prefn\_marker\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  91.7 518015500 6 86335916.7 23200 290953517 [CUDA memcpy DtoH]  8.3 46615496 14 3329678.3 1152 23993376 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  862672.0 6 143778.7 148.535 500000.0 [CUDA memcpy DtoH]  276206.0 14 19729.0 0.004 144453.0 [CUDA memcpy HtoD]  Generating Operating System Runtime API Statistics...  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  33.3 52303573941 537 97399579.0 32241 100566243 sem\_timedwait  33.3 52197444061 536 97383291.2 39105 100646970 poll  22.0 34509316488 69 500135021.6 500046937 501035201 pthread\_cond\_timedwait  11.3 17742862412 2 8871431206.0 4502866824 13239995588 pthread\_cond\_wait  0.1 126858814 9429 13454.1 1038 16470199 read  0.1 86472926 935 92484.4 1059 17116389 ioctl  0.0 3171519 98 32362.4 1476 1226290 mmap  0.0 1321514 101 13084.3 3917 50576 open64  0.0 389974 19 20524.9 3969 68072 fopen64  0.0 352638 5 70527.6 39623 123914 pthread\_create  0.0 202681 26 7795.4 1483 122952 fopen  0.0 183580 3 61193.3 41795 70955 fgets  0.0 120955 19 6366.1 1094 20580 munmap  0.0 108669 15 7244.6 3795 10430 fflush  0.0 95604 28 3414.4 1126 8278 fclose  0.0 77845 15 5189.7 2464 11400 write  0.0 38104 1 38104.0 38104 38104 pthread\_mutex\_lock  0.0 30777 5 6155.4 3290 8581 open  0.0 15354 2 7677.0 7144 8210 socket  0.0 14232 2 7116.0 6286 7946 pthread\_cond\_signal  0.0 9180 5 1836.0 1109 4265 fwrite  0.0 8026 6 1337.7 1008 1976 fcntl  0.0 7834 1 7834.0 7834 7834 connect  0.0 6378 1 6378.0 6378 6378 pipe2  0.0 2569 1 2569.0 2569 2569 bind  0.0 1178 1 1178.0 1178 1178 putc  0.0 1145 1 1145.0 1145 1145 listen  Generating NVTX Push-Pop Range Statistics...  NVTX Push-Pop Range Statistics (nanoseconds)  real 1m11.162s  user 1m16.065s  sys 0m14.053s  A screenshot of a computer  Description automatically generated  A screenshot of a graph  Description automatically generated |
|  |
| 1. What references did you use when implementing this technique?   I used histogram part of the class, the previous MP where we implemented the atomic operation and also the lecture in which we went over this topic, and the textbook |
|  |
|  |
| 1. **Optimization 4: *<Tuning with restrict and loop unrolling (considered as one optimization only if you do both)> 3 points Stacked on Optimization 1,2***   ***(Delete this section 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 chose to implement the tuning with restrict and loop unrolling optimization. I chose this optimization because it seemed like an interesting optimization to implement.* |
| * 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? |
| *This optimization works by using the restrict keyword in the parameters for the kernel which allows the compiler to understand the pointers are not aliased which should allow it to perform faster. The loop unrolling allows for multiple iterations of a loop to be consolidated into a single iteration, which should reduce the total amount of loop overhead, which should increase the performance as well since it allows for more parallelism in the computations so the compiler can use the resources it has better. To find the best value for unrolling, I simply tested values until one set of values resulted in a lower Op time. This optimization does synergize with the previous optimizations because it allows for further restriction of the number of computations or memory accesses that happen in the forward convolution.* |
| * 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 | *<0.160412 ms >* | *<0.409067 ms >* | *<0m1.585s >* | *<.86>* | | 1000 | *<1.54411 ms>* | *<4.12461 ms >* | *<0m10.422s >* | *<.886>* | | 5000 | *<6.92641 ms >* | *<18.3425 ms >* | *<0m56.935s >* | *<.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).   Implementing this optimization was successful in improving the performance of the forward convolution. We can see that the overall Op times decreased by a very large amount, the previous optimization of atomic+sweeping+constant memory resulting in an overall Op time of around around 34 ms, while adding the tuning optimization on top reduces the Op time to around 25 ms which is almost a 10 ms decrease. The total execution time remained around the same, but the Op time decreased significantly which shows how this optimization was very successful in improving the performance of the forward convolution. Furthermore, when comparing the statistics returned from the nsys profiling and the nsight compute, we can see that the compute throughput decreased, and the memory throughput increased. Overall, this was a successful optimization. |
| *Running test case 1*  *B = 1 M = 3 C = 3 H = 224 W = 224 K = 3 S = 1*  *Running test case 2*  *B = 2 M = 3 C = 3 H = 301 W = 301 K = 3 S = 2*  *Running test case 3*  *B = 3 M = 3 C = 3 H = 196 W = 196 K = 3 S = 3*  *Running test case 4*  *B = 4 M = 3 C = 3 H = 239 W = 239 K = 3 S = 4*  *All test cases passed*  *Test batch size: 5000*  *Loading fashion-mnist data...Done*  *Loading model...Done*  *Conv-GPU==*  *Layer Time: 309.956 ms*  *Op Time: 7.81181 ms*  *Conv-GPU==*  *Layer Time: 247.381 ms*  *Op Time: 20.6612 ms*  *Test Accuracy: 0.871*  *Generating the /build/report1.qdstrm file.*  *Capturing raw events...*  *334353 total events collected.*  *Capturing symbol files...*  *Saving diagnostics...*  *Saving qdstrm file to disk...*  *Finished saving file.*  *Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.*  *Importing...*  *Importing [==================================================100%]*  *Saving report to file "/build/report1.qdrep"*  *Report file saved.*  *Please discard the qdstrm file and use the qdrep file instead.*  *Removed /build/report1.qdstrm as it was successfully imported.*  *Please use the qdrep file instead.*  *Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.*  *Exporting 334287 events:*  *0% 10 20 30 40 50 60 70 80 90 100%*  *|----|----|----|----|----|----|----|----|----|----|*  *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *Exported successfully to*  */build/report1.sqlite*  *Generating CUDA API Statistics...*  *CUDA API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *70.8 530090068 14 37863576.3 29947 276359650 cudaMemcpy*  *23.5 176077836 14 12576988.3 3843 172442476 cudaMalloc*  *3.8 28476583 10 2847658.3 3435 20622238 cudaDeviceSynchronize*  *1.4 10344330 10 1034433.0 24742 10092300 cudaLaunchKernel*  *0.3 2486750 20 124337.5 1669 437445 cudaFree*  *0.1 999771 6 166628.5 105960 189329 cudaMemcpyToSymbol*  *Generating CUDA Kernel Statistics...*  *Generating CUDA Memory Operation Statistics...*  *CUDA Kernel Statistics (nanoseconds)*  *Time(%) Total Time Instances Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *100.0 28456950 6 4742825.0 8096 20622198 conv\_forward\_kernel*  *0.0 2880 2 1440.0 1376 1504 do\_not\_remove\_this\_kernel*  *0.0 2464 2 1232.0 1184 1280 prefn\_marker\_kernel*  *CUDA Memory Operation Statistics (nanoseconds)*  *Time(%) Total Time Operations Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *90.9 477176833 6 79529472.2 23136 275665888 [CUDA memcpy DtoH]*  *9.1 47798390 14 3414170.7 1152 24175161 [CUDA memcpy HtoD]*  *CUDA Memory Operation Statistics (KiB)*  *Total Operations Average Minimum Maximum Name*  *----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------*  *862672.0 6 143778.7 148.535 500000.0 [CUDA memcpy DtoH]*  *276206.0 14 19729.0 0.004 144453.0 [CUDA memcpy HtoD]*  *Generating Operating System Runtime API Statistics...*  *Operating System Runtime API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *33.3 51498711390 530 97167380.0 20384 100283083 sem\_timedwait*  *33.3 51395475794 528 97339916.3 27098 100386565 poll*  *22.0 34006561749 68 500096496.3 500073236 500198829 pthread\_cond\_timedwait*  *11.3 17452961903 2 8726480951.5 4160189729 13292772174 pthread\_cond\_wait*  *0.1 86873818 934 93012.7 1026 16854881 ioctl*  *0.0 20148527 9426 2137.5 1011 18675 read*  *0.0 3077952 98 31407.7 1260 1239202 mmap*  *0.0 1070618 101 10600.2 4341 26531 open64*  *0.0 319434 19 16812.3 3008 47507 fopen64*  *0.0 278037 5 55607.4 35771 65159 pthread\_create*  *0.0 269280 26 10356.9 1119 202313 fopen*  *0.0 137375 3 45791.7 41811 51386 fgets*  *0.0 93915 16 5869.7 1042 8358 fflush*  *0.0 84828 17 4989.9 1676 14615 munmap*  *0.0 79785 24 3324.4 1017 8405 fclose*  *0.0 65451 15 4363.4 2223 6173 write*  *0.0 44357 1 44357.0 44357 44357 pthread\_mutex\_lock*  *0.0 30833 5 6166.6 4209 8118 open*  *0.0 15936 2 7968.0 5463 10473 socket*  *0.0 9904 2 4952.0 4855 5049 pthread\_cond\_signal*  *0.0 7564 1 7564.0 7564 7564 connect*  *0.0 6432 1 6432.0 6432 6432 pipe2*  *0.0 5386 3 1795.3 1060 3208 fwrite*  *0.0 2504 2 1252.0 1246 1258 fcntl*  *0.0 1563 1 1563.0 1563 1563 bind*  *Generating NVTX Push-Pop Range Statistics...*  *NVTX Push-Pop Range Statistics (nanoseconds)*  *real 1m15.503s*  *user 1m22.742s*  *sys 0m18.176s*  *A screenshot of a computer  Description automatically generated*  *A screenshot of a graph  Description automatically generated* |
| * 1. What references did you use when implementing this technique? |
| *I used the textbook: D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016, ISBN 978-0123814722 as well as the NVIDIA, CUDA C Programming Guide (CUDA PG), NVIDIA, CUDA C++ Best Practices Guide (CUDA BPG), and NVIDIA Developer Blog (NVIDIA DB)* |
| 1. **Optimization 5: *<FP16 arithmetic. (note this can modify model accuracy slightly)> 4 points***   ***(Delete this section if you did not implement this many optimizations.) Stacked on Optimizations 1,2*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose to implement the optimization for FP16 arithmetic, and I chose this optimization because I thought it would be helpful in reducing the computations that occur.* |
| * 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? |
| *This optimization works by using only the first half of the bits, so the first 16 bits of a number during the computations. The first 16 bits are the most significant bits but ignoring the other 16 bits might lead to a lower accuracy, but it should be faster in terms of the performance of the forward convolution. This optimization synergizes with the other optimizations as you can all stack them on top of each other. However, it may not result in an optimal solution.* |
| * 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 | *<0.236819 ms>* | *<0.701653 ms>* | *<* *0m1.722s>* | *<.86>* | | 1000 | *<2.25716 ms>* | *<* *6.8612 ms>* | *<0m12.541s>* | *<.886>* | | 5000 | *<11.1225 ms>* | *<34.1384 ms>* | *<0m58.574s>* | *<.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). |
| *This optimization did not improve the performance in terms of the Op time, it increased it by a little. In terms of the statistics and profiling we can see that the memory throughput decreased.*  *Running test case 1*  *B = 1 M = 3 C = 3 H = 224 W = 224 K = 3 S = 1*  *Running test case 2*  *B = 2 M = 3 C = 3 H = 301 W = 301 K = 3 S = 2*  *Running test case 3*  *B = 3 M = 3 C = 3 H = 196 W = 196 K = 3 S = 3*  *Running test case 4*  *B = 4 M = 3 C = 3 H = 239 W = 239 K = 3 S = 4*  *All test cases passed*  *Test batch size: 5000*  *Loading fashion-mnist data...Done*  *Loading model...Done*  *Conv-GPU==*  *Layer Time: 374.522 ms*  *Op Time: 11.799 ms*  *Conv-GPU==*  *Layer Time: 290.389 ms*  *Op Time: 34.5048 ms*  *Test Accuracy: 0.871*  *Generating the /build/report1.qdstrm file.*  *Capturing raw events...*  *352057 total events collected.*  *Capturing symbol files...*  *Saving diagnostics...*  *Saving qdstrm file to disk...*  *Finished saving file.*  *Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.*  *Importing...*  *Importing [==================================================100%]*  *Saving report to file "/build/report1.qdrep"*  *Report file saved.*  *Please discard the qdstrm file and use the qdrep file instead.*  *Removed /build/report1.qdstrm as it was successfully imported.*  *Please use the qdrep file instead.*  *Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.*  *Exporting 352030 events:*  *0% 10 20 30 40 50 60 70 80 90 100%*  *|----|----|----|----|----|----|----|----|----|----|*  *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *Exported successfully to*  */build/report1.sqlite*  *Generating CUDA API Statistics...*  *CUDA API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *58.9 2256611005 20 112830550.3 1697 2123420200 cudaFree*  *24.7 947163705 14 67654550.4 4045 642639081 cudaMalloc*  *15.0 576408124 14 41172008.9 33379 294922070 cudaMemcpy*  *1.3 50421099 10 5042109.9 4417 34472878 cudaDeviceSynchronize*  *0.0 1003973 6 167328.8 105380 238995 cudaMemcpyToSymbol*  *0.0 646758 10 64675.8 23370 346408 cudaLaunchKernel*  *Generating CUDA Kernel Statistics...*  *Generating CUDA Memory Operation Statistics...*  *CUDA Kernel Statistics (nanoseconds)*  *Time(%) Total Time Instances Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *100.0 45657209 6 7609534.8 9152 34368410 conv\_forward\_kernel*  *0.0 2816 2 1408.0 1344 1472 prefn\_marker\_kernel*  *0.0 2816 2 1408.0 1408 1408 do\_not\_remove\_this\_kernel*  *CUDA Memory Operation Statistics (nanoseconds)*  *Time(%) Total Time Operations Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *91.9 526347360 6 87724560.0 25632 294223609 [CUDA memcpy DtoH]*  *8.1 46643825 14 3331701.8 1152 24008405 [CUDA memcpy HtoD]*  *CUDA Memory Operation Statistics (KiB)*  *Total Operations Average Minimum Maximum Name*  *----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------*  *862672.0 6 143778.7 148.535 500000.0 [CUDA memcpy DtoH]*  *276206.0 14 19729.0 0.004 144453.0 [CUDA memcpy HtoD]*  *Generating Operating System Runtime API Statistics...*  *Operating System Runtime API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *33.3 55612429599 570 97565666.0 42148 100489966 sem\_timedwait*  *33.2 55489084465 569 97520359.3 53708 100454874 poll*  *22.4 37509680833 75 500129077.8 500056069 500192473 pthread\_cond\_timedwait*  *10.7 17971773663 2 8985886831.5 4781089670 13190683993 pthread\_cond\_wait*  *0.3 554113367 937 591369.7 1033 63233822 ioctl*  *0.0 83013851 9429 8804.1 1161 15532078 read*  *0.0 2992570 98 30536.4 1281 1082174 mmap*  *0.0 1191128 101 11793.3 4785 27138 open64*  *0.0 485067 19 25529.8 4598 94371 fopen64*  *0.0 368547 5 73709.4 40749 137506 pthread\_create*  *0.0 211167 26 8121.8 1476 125676 fopen*  *0.0 189333 3 63111.0 59545 68415 fgets*  *0.0 131757 17 7750.4 1228 14965 fflush*  *0.0 123518 18 6862.1 1232 18723 munmap*  *0.0 109755 1 109755.0 109755 109755 pthread\_mutex\_lock*  *0.0 108570 36 3015.8 1033 8551 fclose*  *0.0 61161 15 4077.4 2185 6550 write*  *0.0 32243 5 6448.6 4574 9063 open*  *0.0 17480 10 1748.0 1085 5939 fwrite*  *0.0 15933 2 7966.5 6817 9116 socket*  *0.0 11555 2 5777.5 5256 6299 pthread\_cond\_signal*  *0.0 8585 1 8585.0 8585 8585 connect*  *0.0 7256 1 7256.0 7256 7256 pipe2*  *0.0 2561 2 1280.5 1217 1344 fcntl*  *0.0 1700 1 1700.0 1700 1700 bind*  *0.0 1041 1 1041.0 1041 1041 listen*  *Generating NVTX Push-Pop Range Statistics...*  *NVTX Push-Pop Range Statistics (nanoseconds)*  *real 1m24.731s*  *user 1m29.149s*  *sys 0m20.143s* |
| * 1. What references did you use when implementing this technique? |
| *I used the textbook: D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016, ISBN 978-0123814722 as well as the NVIDIA, CUDA C Programming Guide (CUDA PG), NVIDIA, CUDA C++ Best Practices Guide (CUDA BPG), and NVIDIA Developer Blog (NVIDIA DB)*   1. *Optimization 6: Using Streams to overlap computation with data transfer 4 points*    1. Which optimization did you choose to implement and why did you choose that optimization technique.   *I chose to implement the streams optimization because it would drastically decrease the Op times and I thought it would be a very useful optimization to learn how to implement.*   * 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?   *This optimization works by allowing for async execution of tasks on the GPU so you can do computations and data transfers at the same time so that the CPU can do its tasks while the GPU does it tasks respectively. The overlapping of computation with data transfer allows the GPU to be much more efficient so it should definitely improve the performance of the convolution kernel. This optimization can synergize with the other ones, but since it was mentioned to not submit it, I just included it separately.*   * 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 | *<* *0.003976 ms>* | *<* *0.003556 ms>* | *<* *0m1.707s>* | *<.86>* | | 1000 | *<* *0.006747 ms>* | *<* 0.004536 ms*>* | *<0m12.135s>* | *<.886>* | | 5000 | *<* *0.00462 ms>* | *<* *0.006053 ms>* | *<0m51.925s>* | *<.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)   *This optimization was definitely successful in improving the performance. When looking at the total execution time, we can see that it decreased. The Op times for this optimization are not valid as mentioned by the instructors. When looking at the other statistics on computation and memory throughput provided by the nsight and nsys profiling, we can further reaffirm that this optimization was successful in improving the performance of the forward convolution.*  *Running test case 1*  *B = 1 M = 3 C = 3 H = 224 W = 224 K = 3 S = 1*  *Running test case 2*  *B = 2 M = 3 C = 3 H = 301 W = 301 K = 3 S = 2*  *Running test case 3*  *B = 3 M = 3 C = 3 H = 196 W = 196 K = 3 S = 3*  *Running test case 4*  *B = 4 M = 3 C = 3 H = 239 W = 239 K = 3 S = 4*  *All test cases passed*  *Test batch size: 5000*  *Loading fashion-mnist data...Done*  *Loading model...Done*  *Conv-GPU==*  *Layer Time: 589.064 ms*  *Op Time: 0.005324 ms*  *Conv-GPU==*  *Layer Time: 495.851 ms*  *Op Time: 0.004863 ms*  *Test Accuracy: 0.871*  *Generating the /build/report1.qdstrm file.*  *Capturing raw events...*  *452893 total events collected.*  *Capturing symbol files...*  *Saving diagnostics...*  *Saving qdstrm file to disk...*  *Finished saving file.*  *Importing the qdstrm file using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/QdstrmImporter.*  *Importing...*  *Importing [==================================================100%]*  *Saving report to file "/build/report1.qdrep"*  *Report file saved.*  *Please discard the qdstrm file and use the qdrep file instead.*  *Removed /build/report1.qdstrm as it was successfully imported.*  *Please use the qdrep file instead.*  *Exporting the qdrep file to SQLite database using /opt/nvidia/nsight-systems/2019.5.2/host-linux-x64/nsys-exporter.*  *Exporting 442818 events:*  *0% 10 20 30 40 50 60 70 80 90 100%*  *|----|----|----|----|----|----|----|----|----|----|*  *\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *Exported successfully to*  */build/report1.sqlite*  *Generating CUDA API Statistics...*  *CUDA API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *67.9 892841184 20026 44584.1 5411 445475 cudaMemcpyAsync*  *14.2 186293018 20 9314650.9 1995 182588375 cudaMalloc*  *8.2 108444580 10010 10833.6 1274 1366220 cudaStreamCreate*  *7.0 92370290 10014 9224.1 3361 50657293 cudaLaunchKernel*  *1.6 21370474 10010 2134.9 1460 363178 cudaStreamDestroy*  *0.6 8147842 10010 814.0 591 5800 cudaStreamSynchronize*  *0.2 3169686 2 1584843.0 30871 3138815 cudaMemcpy*  *0.2 2552298 38 67165.7 376 409281 cudaFree*  *0.0 44670 10 4467.0 2150 12780 cudaDeviceSynchronize*  *Generating CUDA Kernel Statistics...*  *Generating CUDA Memory Operation Statistics...*  *CUDA Kernel Statistics (nanoseconds)*  *Time(%) Total Time Instances Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *100.0 124606738 10010 12448.2 6016 20192 conv\_forward\_kernel*  *0.0 2656 2 1328.0 1312 1344 prefn\_marker\_kernel*  *0.0 2496 2 1248.0 1248 1248 do\_not\_remove\_this\_kernel*  *CUDA Memory Operation Statistics (nanoseconds)*  *Time(%) Total Time Operations Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *62.1 73424741 10010 7335.1 3840 46016 [CUDA memcpy DtoH]*  *37.9 44857370 10018 4477.7 1408 94048 [CUDA memcpy HtoD]*  *CUDA Memory Operation Statistics (KiB)*  *Total Operations Average Minimum Maximum Name*  *----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------*  *862672.0 10010 86.2 42.188 577.0 [CUDA memcpy DtoH]*  *276206.0 10018 27.6 0.004 1061.0 [CUDA memcpy HtoD]*  *Generating Operating System Runtime API Statistics...*  *Operating System Runtime API Statistics (nanoseconds)*  *Time(%) Total Time Calls Average Minimum Maximum Name*  *------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------*  *33.3 49400587687 508 97245251.4 38961 100241733 poll*  *33.3 49358647852 507 97354335.0 24749 100233438 sem\_timedwait*  *23.6 35009943342 70 500142047.7 500093601 500157297 pthread\_cond\_timedwait*  *9.7 14340920819 2 7170460409.5 3080197602 11260723217 pthread\_cond\_wait*  *0.1 159643864 6125 26064.3 1000 17921467 ioctl*  *0.0 20722657 9426 2198.5 1121 18457 read*  *0.0 4638351 155 29924.8 1397 1250221 mmap*  *0.0 1776724 141 12600.9 3813 27669 open64*  *0.0 346402 26 13323.2 1025 288214 fopen*  *0.0 287853 19 15150.2 3048 56645 fopen64*  *0.0 246038 5 49207.6 38282 64779 pthread\_create*  *0.0 134425 3 44808.3 41059 50347 fgets*  *0.0 105055 31 3388.9 1388 15642 munmap*  *0.0 80549 15 5369.9 2920 8442 fflush*  *0.0 63227 23 2749.0 1010 7070 fclose*  *0.0 60639 15 4042.6 2284 7014 write*  *0.0 41740 1 41740.0 41740 41740 pthread\_mutex\_lock*  *0.0 25590 5 5118.0 3412 6929 open*  *0.0 9277 2 4638.5 4464 4813 socket*  *0.0 9212 2 4606.0 3767 5445 pthread\_cond\_signal*  *0.0 6351 2 3175.5 1162 5189 fwrite*  *0.0 6209 1 6209.0 6209 6209 pipe2*  *0.0 6022 1 6022.0 6022 6022 connect*  *0.0 1365 1 1365.0 1365 1365 bind*  *0.0 1194 1 1194.0 1194 1194 fcntl*  *Generating NVTX Push-Pop Range Statistics...*  *NVTX Push-Pop Range Statistics (nanoseconds)*  *real 1m11.854s*  *user 1m18.221s*  *sys 0m16.984s*  *A screenshot of a computer  Description automatically generated*  *A screenshot of a graph  Description automatically generated*   * 1. What references did you use when implementing this technique?   *I used the lecture, textbook: D. Kirk and W. Hwu, “Programming Massively Parallel Processors – A Hands-on Approach,” Morgan Kaufman Publisher, 3rd edition, 2016, ISBN 978-0123814722 as well as the NVIDIA, CUDA C Programming Guide (CUDA PG), NVIDIA, CUDA C++ Best Practices Guide (CUDA BPG), and NVIDIA Developer Blog (NVIDIA DB)*  *//STREAMS OPTIMIZATION*  *#include <cmath>*  *#include <iostream>*  *#include "gpu-new-forward.h"*  *#define TILE\_WIDTH 16*  *\_\_global\_\_ void conv\_forward\_kernel(float \* output, const float \* input, const float \* mask, const int B, const int M, const int C, const int H, const int W, const int K,const int S)*  *{*  */\**  *Modify this function to implement the forward pass described in Chapter 16.*  *We have added an additional dimension to the tensors to support an entire mini-batch*  *The goal here is to be correct AND fast.*  *Function paramter definitions:*  *output - output*  *input - input*  *mask - convolution kernel*  *B - batch\_size (number of images in x)*  *M - number of output feature maps*  *C - number of input feature maps*  *H - input height dimension*  *W - input width dimension*  *K - kernel height and width (K x K)*  *S - stride step length*  *\*/*  *const int H\_out = (H - K)/S + 1;*  *const int W\_out = (W - K)/S + 1;*  *(void)H\_out; // silence declared but never referenced warning. remove this line when you start working*  *(void)W\_out; // silence declared but never referenced warning. remove this line when you start working*  *// We have some nice #defs for you below to simplify indexing. Feel free to use them, or create your own.*  *// An example use of these macros:*  *// float a = in\_4d(0,0,0,0)*  *// out\_4d(0,0,0,0) = a*  *#define out\_4d(i3, i2, i1, i0) output[(i3) \* (M \* H\_out \* W\_out) + (i2) \* (H\_out \* W\_out) + (i1) \* (W\_out) + i0]*  *#define in\_4d(i3, i2, i1, i0) input[(i3) \* (C \* H \* W) + (i2) \* (H \* W) + (i1) \* (W) + i0]*  *#define mask\_4d(i3, i2, i1, i0) mask[(i3) \* (C \* K \* K) + (i2) \* (K \* K) + (i1) \* (K) + i0]*  *// Insert your GPU convolution kernel code here*  *int temp = (ceil(W\_out/(1.0\*TILE\_WIDTH)));*  *int bx = blockIdx.x;*  *int by = blockIdx.y;*  *int h = (blockIdx.z/temp)\*TILE\_WIDTH+threadIdx.y;*  *int w = (blockIdx.z%temp)\*TILE\_WIDTH+threadIdx.x;*  *if (w<W\_out)*  *{*  *if(h<H\_out)*  *{*  *if(by<M)*  *{*  *float f = 0.0f;*  *for (int channel=0; channel<C; channel++)*  *{*  *for (int p = 0; p < K; p++)*  *{*  *for (int q = 0; q < K; q++)*  *{*  *f+=in\_4d(bx, channel, h\*S+p, w\*S+q)\*mask\_4d(by,channel,p,q);*  *}*  *}*  *}*  *out\_4d(blockIdx.x, by, h, w) = f;*  *}*  *}*  *}*  *#undef out\_4d*  *#undef in\_4d*  *#undef mask\_4d*  *}*    *\_\_host\_\_ void GPUInterface::conv\_forward\_gpu\_prolog(const float \*host\_output, const float \*host\_input, const float \*host\_mask, float \*\*device\_output\_ptr, float \*\*device\_input\_ptr, float \*\*device\_mask\_ptr, const int B, const int M, const int C, const int H, const int W, const int K, const int S)*  *{*  *// Allocate memory and copy over the relevant data structures to the GPU*  *// We pass double pointers for you to initialize the relevant device pointers,*  *// which are passed to the other two functions.*  *// Useful snippet for error checking*  *// cudaError\_t error = cudaGetLastError();*  *// if(error != cudaSuccess)*  *// {*  *// std::cout<<"CUDA error: "<<cudaGetErrorString(error)<<std::endl;*  *// exit(-1);*  *// }*  *const int H\_out = (H - K)/S + 1;*  *const int W\_out = (W - K)/S + 1;*  *size\_t val1 = (B \* M \* H\_out \* W\_out)\*sizeof(float);*  *size\_t val2 = (B \* C \* H \* W)\*sizeof(float);*  *size\_t val3 = (M \* C \* K \* K)\*sizeof(float);*  *cudaMalloc((void \*\*) device\_output\_ptr, val1);*  *cudaMalloc((void \*\*) device\_input\_ptr, val2);*  *cudaMalloc((void \*\*) device\_mask\_ptr, val3);*  *cudaStream\_t streams[B];*  *for (int i = 0; i < B; ++i)*  *{*  *cudaStreamCreate(&streams[i]);*  *}*  *cudaMemcpyAsync(\*device\_mask\_ptr, host\_mask, val3, cudaMemcpyHostToDevice, streams[0]);*  *dim3 dimGrid(1, M, ceil((float)W\_out/TILE\_WIDTH) \* ceil((float)H\_out/TILE\_WIDTH));*  *dim3 dimBlock(TILE\_WIDTH, TILE\_WIDTH, 1);*    *for(int i = 0; i < B; ++i)*  *{*  *cudaMemcpyAsync(\*device\_input\_ptr + i \* (C\*H\*W), host\_input + i \* (C\*H\*W), (C\*H\*W) \* sizeof(float), cudaMemcpyHostToDevice, streams[i]);*  *conv\_forward\_kernel<<<dimGrid, dimBlock, 0, streams[i]>>>(\*device\_output\_ptr + i \* (M\*H\_out\*W\_out), \*device\_input\_ptr + i \* (C\*H\*W), \*device\_mask\_ptr, B, M, C, H, W, K, S);*  *cudaMemcpyAsync((float\*)host\_output + i \* (M\*H\_out\*W\_out), (\*device\_output\_ptr) + i \* (M\*H\_out\*W\_out), (M\*H\_out\*W\_out) \* sizeof(float), cudaMemcpyDeviceToHost, streams[i]);*  *}*  *for (int i = 0; i < B; ++i)*  *{*  *cudaStreamSynchronize(streams[i]);*  *cudaStreamDestroy(streams[i]);*  *}*  *cudaFree(device\_mask\_ptr);*  *cudaFree(device\_input\_ptr);*  *cudaFree(device\_output\_ptr);*    *}*  *\_\_host\_\_ void GPUInterface::conv\_forward\_gpu(float \*device\_output, const float \*device\_input, const float \*device\_mask, const int B, const int M, const int C, const int H, const int W, const int K, const int S)*  *{*  *return;*  *}*  *\_\_host\_\_ void GPUInterface::conv\_forward\_gpu\_epilog(float \*host\_output, float \*device\_output, float \*device\_input, float \*device\_mask, const int B, const int M, const int C, const int H, const int W, const int K, const int S)*  *{*  *cudaFree(device\_mask);*  *cudaFree(device\_input);*  *cudaFree(device\_output);*  *}* |
|  |
|  |