|  |  |
| --- | --- |
| **Name:** | Ander Liu |
| **NetID:** | anderdl2 |
| **Section:** | AL1 |

**ECE 408/CS483 Milestone 3 Report**

|  |
| --- |
| 1. List Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | 0.21207 ms | 0.618524 ms | 0m1.158s | 0.86 | | 1000 | 1.92372 ms | 5.83225 ms | 0m11.235s | 0.886 | | 10000 | 18.7866 ms | 57.072 ms | 1m39.627s | 0.8714 | |
| 1. **Optimization 1: Weight matrix (kernel values) in constant memory (1 point)** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| Weight matrix in constant memory, because using constant memory can reduce global memory access. |
| * 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?   In host, we put the weight matrix into constant memory first. Then, whenever we need access to the weight matrix, we can directly get it from constant memory instead of global memory. I think the optimization could reduce OP time because the global memory bandwidth would decrease. |
|  |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | 0.16978 ms | 0.509486 ms | 0m1.484s | 0.86 | | 1000 | 1.62142 ms | 5.23384 ms | 0m9.797s | 0.886 | | 10000 | 15.9062 ms | 51.8366 ms | 1m36.829s | 0.8714 | |
| * 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 slightly improves the performance. Before the optimization, the total time of conv\_forward\_kernel is 846ms. After the optimization, the total time of conv\_forward\_kernel is 732ms   * **Before (baseline)**   CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  93.8 185336005 8 23167000.6 5015 184778945 cudaMalloc  5.4 10657368 8 1332171.0 17636 5635700 cudaMemcpy  0.4 876302 6 146050.3 2918 633192 cudaDeviceSynchronize  0.3 506914 6 84485.7 5696 144198 cudaFree  0.1 117269 6 19544.8 14623 24389 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.4 845946 2 422973.0 213822 632124 conv\_forward\_kernel  0.3 2688 2 1344.0 1280 1408 prefn\_marker\_kernel  0.3 2592 2 1296.0 1280 1312 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  93.3 8178665 2 4089332.5 3461417 4717248 [CUDA memcpy DtoH]  6.7 588541 6 98090.2 1376 337726 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  35.1 1134820547 26 43646944.1 23278 100152941 sem\_timedwait  31.0 1000917110 25 40036684.4 35079 100219931 poll  31.0 1000222676 2 500111338.0 500093718 500128958 pthread\_cond\_timedwait     * **After (Weight matrix in constant memory)** |
| CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  92.8 170451291 6 28408548.5 74487 169917476 cudaMalloc  6.2 11376831 6 1896138.5 11753 6135749 cudaMemcpy  0.4 748391 6 124731.8 2980 559688 cudaDeviceSynchronize  0.3 629120 6 104853.3 57668 179366 cudaFree  0.2 407238 2 203619.0 202829 204409 cudaMemcpyToSymbol  0.1 115539 6 19256.5 13755 23838 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.3 732190 2 366095.0 174175 558015 conv\_forward\_kernel  0.4 2720 2 1360.0 1344 1376 prefn\_marker\_kernel  0.4 2592 2 1296.0 1248 1344 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  90.8 8956523 2 4478261.5 3792471 5164052 [CUDA memcpy DtoH]  9.2 904446 6 150741.0 1216 480479 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  34.6 1099201304 25 43968052.2 30586 100166372 sem\_timedwait  31.5 1001974516 25 40078980.6 38452 100217631 poll  31.5 1000252958 2 500126479.0 500105654 500147304 pthread\_cond\_timedwait |
| * 1. What references did you use when implementing this technique? |
| Lab4 from UIUC ECE408 |
| 1. **Optimization 2: Shared memory matrix multiplication and input matrix unrolling + Shared memory matrix multiplication and input matrix unrolling (5 points)** |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| Shared memory matrix multiplication and input matrix unrolling, because the baseline algorithm is not efficient in global memory bandwidth. Each input tile will be loaded M times, where M is number of output features. |
| 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? |
| We unroll the input matrix so that the convolution has been transform into matrix multiplication, which could reduce more global memory bandwidth. I think the optimization could reduce OP time because the global memory bandwidth would decrease. I synergize this optimization with Weight matrix in constant memory. |
| 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | 0.687337 ms | 1.8069 ms | 0m1.218s | 0.86 | | 1000 | 5.27139 ms | 13.8351 ms | 0m9.790s | 0.886 | | 10000 | 51.212 ms | 595.715 ms | 1m36.738s | 0.8714 | |
| 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 decreases the performance instead. Before the optimization, the total time of conv\_forward\_kernel is 732ms. After the optimization, the total time of conv\_forward\_kernel is 2.486s. I think the reason is memory coalescing. Although we reduce the total access of memory, the implementation does not fit with memory burst, which is important when memory optimization as well. |
| * **Before (Weight matrix in constant memory)**   CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  92.8 170451291 6 28408548.5 74487 169917476 cudaMalloc  6.2 11376831 6 1896138.5 11753 6135749 cudaMemcpy  0.4 748391 6 124731.8 2980 559688 cudaDeviceSynchronize  0.3 629120 6 104853.3 57668 179366 cudaFree  0.2 407238 2 203619.0 202829 204409 cudaMemcpyToSymbol  0.1 115539 6 19256.5 13755 23838 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.3 732190 2 366095.0 174175 558015 conv\_forward\_kernel  0.4 2720 2 1360.0 1344 1376 prefn\_marker\_kernel  0.4 2592 2 1296.0 1248 1344 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  90.8 8956523 2 4478261.5 3792471 5164052 [CUDA memcpy DtoH]  9.2 904446 6 150741.0 1216 480479 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  34.6 1099201304 25 43968052.2 30586 100166372 sem\_timedwait  31.5 1001974516 25 40078980.6 38452 100217631 poll  31.5 1000252958 2 500126479.0 500105654 500147304 pthread\_cond\_timedwait   * **After (Weight matrix in constant memory + Shared memory matrix multiplication and input matrix unrolling)**   CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  92.9 191246907 6 31874484.5 75812 190662214 cudaMalloc  5.3 10953041 6 1825506.8 12708 5902541 cudaMemcpy  1.2 2502647 6 417107.8 3089 1804983 cudaDeviceSynchronize  0.3 621546 6 103591.0 61793 163830 cudaFree  0.2 397966 2 198983.0 198455 199511 cudaMemcpyToSymbol  0.1 117895 6 19649.2 14743 23014 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.8 2486478 2 1243239.0 682779 1803699 conv\_forward\_kernel  0.1 2528 2 1264.0 1184 1344 prefn\_marker\_kernel  0.1 2400 2 1200.0 1184 1216 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  90.5 8604030 2 4302015.0 3616356 4987674 [CUDA memcpy DtoH]  9.5 904409 6 150734.8 1184 480764 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  34.8 1119405357 26 43054052.2 38820 100151786 sem\_timedwait  31.2 1002479586 25 40099183.4 52837 100227092 poll  31.1 1000268842 2 500134421.0 500107029 500161813 pthread\_cond\_timedwait |
| 1. What references did you use when implementing this technique? |
| Slides from UIUC ECE408 Lecture 12 |

|  |
| --- |
| 1. **Optimization 3: FP16 arithmetic (4 points)** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| FP16 arithmetic, because using half-precision floating-point might reduce the time of calculation. |
| * 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? |
| There are two types of FP16 in CUDA, half and half2. half is single 16-bit floating point quantity/type. half2 is a vector type, consisting of two 16-bit floating point quantities packed into a single 32-bit type. In kernel, I transform two floats into half2 first. And then, multiple half2 and transform back into floats again.  I think the optimization could reduce OP time because the calculation reduces.  I synergize this optimization with Weight matrix in constant memory. |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | 1.88773 ms | 2.45859 ms | 0m1.205s | 0.86 | | 1000 | 2.40648 ms | 7.88971 ms | 0m9.994s | 0.886 | | 10000 | 23.8918 ms | 76.3115 ms | 1m39.224s | 0.8714 | |

|  |
| --- |
| * 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 does not improve the performance as well. Before the optimization, the total time of conv\_forward\_kernel is 732ms. After the optimization, the total time of conv\_forward\_kernel is 1.056s. I think the reason is that the conversion time between float and FP16 is larger than the reduced time of calculation   * **Before (Weight matrix in constant memory)** |
| CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  92.8 170451291 6 28408548.5 74487 169917476 cudaMalloc  6.2 11376831 6 1896138.5 11753 6135749 cudaMemcpy  0.4 748391 6 124731.8 2980 559688 cudaDeviceSynchronize  0.3 629120 6 104853.3 57668 179366 cudaFree  0.2 407238 2 203619.0 202829 204409 cudaMemcpyToSymbol  0.1 115539 6 19256.5 13755 23838 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.3 732190 2 366095.0 174175 558015 conv\_forward\_kernel  0.4 2720 2 1360.0 1344 1376 prefn\_marker\_kernel  0.4 2592 2 1296.0 1248 1344 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  90.8 8956523 2 4478261.5 3792471 5164052 [CUDA memcpy DtoH]  9.2 904446 6 150741.0 1216 480479 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  34.6 1099201304 25 43968052.2 30586 100166372 sem\_timedwait  31.5 1001974516 25 40078980.6 38452 100217631 poll  31.5 1000252958 2 500126479.0 500105654 500147304 pthread\_cond\_timedwait   * **After (Weight matrix in constant memory + FP16)**   CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  93.1 176129476 6 29354912.7 75957 175497364 cudaMalloc  5.7 10815388 6 1802564.7 12235 5799626 cudaMemcpy  0.6 1072154 6 178692.3 3162 810449 cudaDeviceSynchronize  0.3 592857 6 98809.5 61519 149900 cudaFree  0.2 398766 2 199383.0 194448 204318 cudaMemcpyToSymbol  0.1 127542 6 21257.0 15305 27471 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.5 1056024 2 528012.0 247358 808666 conv\_forward\_kernel  0.2 2528 2 1264.0 1184 1344 prefn\_marker\_kernel  0.2 2432 2 1216.0 1216 1216 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  90.3 8458909 2 4229454.5 3545988 4912921 [CUDA memcpy DtoH]  9.7 904824 6 150804.0 1216 480988 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  34.7 1161505490 26 44673288.1 27492 100162685 sem\_timedwait  32.9 1101419643 26 42362294.0 32943 100219958 poll  29.9 1000268776 2 500134388.0 500115683 500153093 pthread\_cond\_timedwait |
| * 1. What references did you use when implementing this technique? |
| <https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__HALF.html> |
| 1. **Optimization 4: Tuning with restrict and loop unrolling (3 points)** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| Restrict pointer and loop unrolling, because they can optimize the compiler more. |
| * 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?   Restrict pointer means that the pointer is the only way to access the object pointed by it. By utilizing restrict pointer, compiler can produce better optimized code.  The first benefit of loop unrolling is that it reduces the calculation of index.  The second benefit of loop unrolling is the enhancement of Instruction-Level Parallelism. In the unrolled version, there would possibly be more operations for the processor to push into processing pipeline without being worried about the for loop condition in every iteration.  I think the optimization could reduce OP time because of the complier optimization.  I synergize this optimization with Weight matrix in constant memory. |
|  |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | 0.194829 ms | 0.615394 ms | 0m1.274s | 0.86 | | 1000 | 1.75325 ms | 5.76556 ms | 0m9.707s | 0.886 | | 10000 | 17.2728 ms | 57.4893 ms | 1m39.408s | 0.8714 | |
| * 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 does not affect the performance. Before the optimization, the total time of conv\_forward\_kernel is 732ms. After the optimization, the total time of conv\_forward\_kernel is 790ms. I think the reason is that the complier is advanced enough that it already implements Tuning with restrict and loop unrolling.   * **Before (Weight matrix in constant memory)**   CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  92.8 170451291 6 28408548.5 74487 169917476 cudaMalloc  6.2 11376831 6 1896138.5 11753 6135749 cudaMemcpy  0.4 748391 6 124731.8 2980 559688 cudaDeviceSynchronize  0.3 629120 6 104853.3 57668 179366 cudaFree  0.2 407238 2 203619.0 202829 204409 cudaMemcpyToSymbol  0.1 115539 6 19256.5 13755 23838 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.3 732190 2 366095.0 174175 558015 conv\_forward\_kernel  0.4 2720 2 1360.0 1344 1376 prefn\_marker\_kernel  0.4 2592 2 1296.0 1248 1344 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  90.8 8956523 2 4478261.5 3792471 5164052 [CUDA memcpy DtoH]  9.2 904446 6 150741.0 1216 480479 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  34.6 1099201304 25 43968052.2 30586 100166372 sem\_timedwait  31.5 1001974516 25 40078980.6 38452 100217631 poll  31.5 1000252958 2 500126479.0 500105654 500147304 pthread\_cond\_timedwait |
| CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  93.2 195806043 6 32634340.5 85007 195167318 cudaMalloc  5.9 12480644 6 2080107.3 13348 7255342 cudaMemcpy  0.4 806017 6 134336.2 2709 607979 cudaDeviceSynchronize  0.3 572078 6 95346.3 57150 129742 cudaFree  0.2 399348 2 199674.0 198119 201229 cudaMemcpyToSymbol  0.1 121053 6 20175.5 14292 24991 cudaLaunchKernel   * **After (Weight matrix in constant memory + Tuning with restrict and loop unrolling)**   CUDA API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  93.2 195806043 6 32634340.5 85007 195167318 cudaMalloc  5.9 12480644 6 2080107.3 13348 7255342 cudaMemcpy  0.4 806017 6 134336.2 2709 607979 cudaDeviceSynchronize  0.3 572078 6 95346.3 57150 129742 cudaFree  0.2 399348 2 199674.0 198119 201229 cudaMemcpyToSymbol  0.1 121053 6 20175.5 14292 24991 cudaLaunchKernel  CUDA Kernel Statistics (nanoseconds)  Time(%) Total Time Instances Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  99.3 790493 2 395246.5 184767 605726 conv\_forward\_kernel  0.3 2624 2 1312.0 1248 1376 prefn\_marker\_kernel  0.3 2592 2 1296.0 1280 1312 do\_not\_remove\_this\_kernel  CUDA Memory Operation Statistics (nanoseconds)  Time(%) Total Time Operations Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  91.6 9861593 2 4930796.5 3771185 6090408 [CUDA memcpy DtoH]  8.4 903996 6 150666.0 1216 480446 [CUDA memcpy HtoD]  CUDA Memory Operation Statistics (KiB)  Total Operations Average Minimum Maximum Name  ----------------- -------------- ----------------- ----------------- ----------------- --------------------------------------------------------------------------------  17225.0 2 8612.0 7225.000 10000.0 [CUDA memcpy DtoH]  5402.0 6 900.0 0.004 2889.0 [CUDA memcpy HtoD]  Operating System Runtime API Statistics (nanoseconds)  Time(%) Total Time Calls Average Minimum Maximum Name  ------- -------------- ---------- -------------- -------------- -------------- --------------------------------------------------------------------------------  35.0 1123566464 26 43214094.8 24007 100160391 sem\_timedwait  31.2 1001225507 25 40049020.3 35192 100218490 poll  31.2 1000270362 2 500135181.0 500120288 500150074 pthread\_cond\_timedwait |
| * 1. What references did you use when implementing this technique? |
| <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#restrict>  <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#pragma-unroll> |
| 1. **Optimization 5: Sweeping various parameters to find best values(1 point)** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| Sweeping various parameters, because finding correct parameters might improve the performance 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?   I test with different parameter, such as (4, 4, 64), (8, 8, 16), (16, 16, 4), and find that (8, 8, 16) is the best. However, in PM2, I already tune the parameter into (8, 8, 16), so it will not change the performance. |
|  |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 10k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | 0.194829 ms | 0.615394 ms | 0m1.274s | 0.86 | | 1000 | 1.75325 ms | 5.76556 ms | 0m9.707s | 0.886 | | 10000 | 17.2728 ms | 57.4893 ms | 1m39.408s | 0.8714 | |
| * 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). |
| In PM2, I already tune the parameter into (8, 8, 16), so it will not change the performance. |
| * 1. What references did you use when implementing this technique?   PM3 github of UIUC ECE408 |