|  |  |
| --- | --- |
| **Name:** | *Xuanbo Jin* |
| **NetID:** | *xuanboj2* |
| **Section:** | *ZJUI VK* |

**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 | *0.236463 ms* | *0.677674 ms* | *0m0.215s* | *0.86* | | 1000 | *3.58915 ms* | *6.46691 ms* | *0m0.320s* | *0.887* | | 5000 | *9.2407 ms* | *31.8312 ms* | *0m1.060s* | *0.8712* |   **CUDA API Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Calls** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 59.4 | 174599088 | 20 | 8729954.4 | 2631 | 172410862 | cudaMalloc | | 37.0 | 108744592 | 20 | 5437229.6 | 31812 | 57407516 | cudaMemcpy | | 2.9 | 8387028 | 10 | 838702.8 | 3525 | 6551902 | cudaDeviceSynchronize | | 0.6 | 1889911 | 20 | 94495.6 | 3432 | 285744 | cudaFree | | 0.1 | 257650 | 10 | 25765.0 | 18170 | 31050 | cudaLaunchKernel |   **CUDA Kernel Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Instances** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 99.9 | 8367826 | 6 | 1394637.7 | 7584 | 6549955 | conv\_forward\_kernel | | 0.0 | 2848 | 2 | 1424.0 | 1376 | 1472 | do\_not\_remove\_this\_kernel | | 0.0 | 2528 | 2 | 1264.0 | 1216 | 1312 | prefn\_marker\_kernel |   **CUDA Memory Operation Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Operations** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 92.2 | 97175440 | 6 | 16195906.7 | 12800 | 56493742 | [CUDA memcpy DtoH] | | 7.8 | 8222288 | 14 | 587306.3 | 1120 | 4015386 | [CUDA memcpy HtoD] | |
| 1. **Optimization 1: *Shared memory matrix multiplication and input matrix unrolling*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose to implement shared memory matrix multiplication and input matrix unrolling for the forward convolution on the GPU. Shared memory reduces global memory transfers, enhancing efficiency. Input matrix unrolling optimizes memory access patterns, aligning with GPU SIMD capabilities.* |
| * 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? |
| *HOW: Shared memory matrix multiplication partitions input matrices into smaller tiles stored in shared memory, reducing global memory access. Input matrix unrolling reshapes the input, improving vectorized operations and memory access. These optimizations are expected to increase forward convolution performance by minimizing data movement and leveraging GPU parallelism.*  *MY OPINION: I don’t think it will be effective, because it would cost a lot of time to allocate memory for shared mem, which could be even longer than potential optimization time.*  *They synergize by using shared memory to cache unrolled input portions, combining reduced global memory transfers with optimized memory access patterns.* |
| * 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 | *0.332411 ms* | *1.55134 ms* | *0m0.164s* | *0.86* | | 1000 | *3.4167 ms* | *16.862 ms* | *0m0.420s* | *0.886* | | 5000 | *17.138 ms* | *84.5166 ms* | *0m1.088s* | *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). |
| **CUDA API Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Calls** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 55.7 | 170905225 | 20 | 8545261.2 | 3233 | 168563388 | cudaMalloc | | 36.9 | 113046742 | 20 | 5652337.1 | 27913 | 61370271 | cudaMemcpy | | 6.7 | 20521385 | 10 | 2052138.5 | 3531 | 16945365 | cudaDeviceSynchronize | | 0.6 | 1925119 | 20 | 96255.9 | 3937 | 227184 | cudaFree | | 0.1 | 259157 | 10 | 25915.7 | 17141 | 30348 | cudaLaunchKernel |   **CUDA Kernel Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Instances** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 100.0 | 20496247 | 6 | 3416041.2 | 22559 | 16939727 | conv\_forward\_kernel | | 0.0 | 2848 | 2 | 1424.0 | 1408 | 1440 | 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** | | --- | --- | --- | --- | --- | --- | --- | | 92.8 | 101510304 | 6 | 16918384.0 | 12640 | 60293073 | [CUDA memcpy DtoH] | | 7.2 | 7906891 | 14 | 564777.9 | 1152 | 3863430 | [CUDA memcpy HtoD] |   **CUDA Memory Operation Statistics (KiB)**   | **Total** | **Operations** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | | 173672.0 | 6 | 28945.4 | 148.535 | 100000.0 | [CUDA memcpy DtoH] | | 60644.0 | 14 | 4331.7 | 0.004 | 28890.0 | [CUDA memcpy HtoD] |   *Unsuccessful.*  *WHY: In this case, the memory access patterns of the algorithm may not align well with the advantages that shared memory offers. If the algorithm doesn't exhibit spatial locality or if the required data doesn't fit into shared memory efficiently, the benefits may be limited.* |
| * 1. What references did you use when implementing this technique? |
| *None* |
| 1. **Optimization 2:** ***FP16 arithmetic & Multiple kernel implementations for different layer sizes & Weight matrix (kernel values) in constant memory*** |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *FP16 Arithmetic: Employing FP16 arithmetic reduces memory requirements by using half-precision, beneficial for large-scale data processing and neural networks. This choice prioritizes faster computation and lower memory usage.*  *Multiple Kernel Implementations: Implementing different kernels for various layer sizes optimizes resource utilization, parallelism, and memory access patterns. This approach recognizes that efficient algorithms may vary based on layer dimensions, enhancing overall GPU performance.*  *Constant memory provides fast, read-only access for all threads in a block, minimizing global memory transfers and enhancing memory coalescing. This proves beneficial in convolution operations where the same kernel is applied across data points.* |
| 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? |
| *HOW:*   1. *FP16 Arithmetic: Utilizing 16-bit floating-point precision reduces memory bandwidth demands. While sacrificing some precision, it often suffices for tasks like deep learning, where faster computation and reduced memory usage are critical. Synergizes with multiple kernels by further minimizing memory overhead.* 2. *Multiple Kernel Implementations: Tailoring kernels for different layer sizes optimizes resource usage, parallelism, and memory access patterns. This flexibility accommodates diverse optimization needs. Synergizes with FP16 arithmetic by aligning with the goal of memory efficiency.* 3. *Constant memory provides fast, read-only access for all threads in a block*   *MY OPINION: probably not so good optimization, since we were told in READ.ME that the change could be slight*  *It doesn’t synergize with any of my previous optimizations. Const mem can.* |
| 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 | *0.223472 ms* | *0.673919 ms* | *0m0.196s* | *0.86* | | 1000 | *2.05063 ms* | *6.5197 ms* | *0m0.340s* | *0.887* | | 5000 | *9.58134 ms* | *37.3775 ms* | *0m1.199s* | *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). |
| **CUDA API Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Calls** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 60.2 | 188602698 | 32 | 5893834.3 | 2218 | 185924031 | cudaMalloc | | 36.4 | 113980568 | 20 | 5699028.4 | 28994 | 58679818 | cudaMemcpy | | 2.6 | 8092819 | 28 | 289029.3 | 768 | 6123583 | cudaDeviceSynchronize | | 0.7 | 2056592 | 32 | 64268.5 | 2802 | 260033 | cudaFree | | 0.1 | 317056 | 22 | 14411.6 | 4570 | 32124 | cudaLaunchKernel | | 0.0 | 73185 | 6 | 12197.5 | 11463 | 13216 | cudaMemcpyToSymbol |   **CUDA Kernel Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Instances** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 97.8 | 7863492 | 6 | 1310582.0 | 6848 | 6121137 | conv\_forward\_kernel | | 2.1 | 168574 | 12 | 14047.8 | 1440 | 76576 | Float2HalfKernel | | 0.0 | 2688 | 2 | 1344.0 | 1344 | 1344 | do\_not\_remove\_this\_kernel | | 0.0 | 2496 | 2 | 1248.0 | 1216 | 1280 | prefn\_marker\_kernel |   **CUDA Memory Operation Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Operations** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 90.9 | 100472758 | 6 | 16745459.7 | 23552 | 57792065 | [CUDA memcpy DtoH] | | 9.1 | 10104530 | 14 | 721752.1 | 1120 | 4813371 | [CUDA memcpy HtoD] | | 0.0 | 10048 | 6 | 1674.7 | 1376 | 2144 | [CUDA memcpy DtoD] |   **CUDA Memory Operation Statistics (KiB)**   | **Total** | **Operations** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | | 173672.0 | 6 | 28945.4 | 148.535 | 100000.0 | [CUDA memcpy DtoH] | | 60644.0 | 14 | 4331.7 | 0.004 | 28890.0 | [CUDA memcpy HtoD] | | 7.0 | 6 | 1.2 | 0.158 | 6.0 | [CUDA memcpy DtoD] |   *Unsuccessful.*  *WHY:*  *Loss of Precision:*  *FP16 has lower precision compared to FP32, and this reduction might lead to numerical instability or loss of accuracy in certain computations. If the specific application demands higher precision, sacrificing precision for speed may not be suitable.*  *Limited Dynamic Range:*  *FP16 has a more limited dynamic range, which can result in numerical issues, especially when dealing with values outside its representable range. This may impact the accuracy of certain computations and limit the applicability of FP16 in scenarios requiring a broader range.* |
| 1. What references did you use when implementing this technique? |
| *https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html* |

|  |
| --- |
| 1. **Optimization 3: *Tuning with restrict and loop unrolling*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *Tuning with restrict and loop unrolling.*  *WHY: Utilizing restrict in GPU programming informs the compiler about non-overlapping pointers, aiding in aggressive optimizations for reduced memory reloads.* |
| * 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? |
| *HOW: Loop unrolling enhances performance by reducing loop overhead and increasing instruction-level parallelism. Together, these techniques improve data locality, minimize memory latency, and boost instruction throughput, optimizing GPU kernels. This is particularly beneficial in scenarios where precise control over memory access patterns and loop structures is essential for leveraging the parallel processing capabilities of GPUs efficiently.*  *MY OPINION: I think I would increase the performance of the forward convolution. Because in this case, K could only be chosen from 3 or 7, which means we can unrolling loops efficiently.*  *It synergizes with all of my previous optimizations, because all optimizations have same loops about K.* |
| * 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 | *0.163001 ms* | *0.585012 ms* | *0m0.185s* | *0.86* | | 1000 | *1.37721 ms* | *5.31537 ms* | *0m0.332s* | *0.887* | | 5000 | *7.54042 ms* | *29.7535 ms* | *0m1.048s* | *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). |
| **CUDA API Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Calls** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 61.6 | 455614159 | 32 | 14237942.5 | 2673 | 184401964 | cudaMalloc | | 19.6 | 145049487 | 32 | 4532796.5 | 2844 | 62349649 | cudaFree | | 17.6 | 129802393 | 20 | 6490119.7 | 31900 | 61728434 | cudaMemcpy | | 1.1 | 8115294 | 28 | 289831.9 | 787 | 6120256 | cudaDeviceSynchronize | | 0.1 | 445172 | 22 | 20235.1 | 4479 | 56002 | cudaLaunchKernel | | 0.0 | 96272 | 6 | 16045.3 | 11551 | 34544 | cudaMemcpyToSymbol |   **CUDA Kernel Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Instances** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 97.8 | 7854776 | 6 | 1309129.3 | 6592 | 6113601 | conv\_forward\_kernel | | 2.1 | 167614 | 12 | 13967.8 | 1600 | 76031 | Float2HalfKernel | | 0.0 | 2816 | 2 | 1408.0 | 1408 | 1408 | do\_not\_remove\_this\_kernel | | 0.0 | 2624 | 2 | 1312.0 | 1312 | 1312 | prefn\_marker\_kernel |   **CUDA Memory Operation Statistics (nanoseconds)**   | **Time(%)** | **Total Time** | **Operations** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | --- | | 91.4 | 115262966 | 6 | 19210494.3 | 23519 | 60786859 | [CUDA memcpy DtoH] | | 8.6 | 10817546 | 14 | 772681.9 | 1120 | 4872871 | [CUDA memcpy HtoD] | | 0.0 | 10400 | 6 | 1733.3 | 1440 | 2208 | [CUDA memcpy DtoD] |   **CUDA Memory Operation Statistics (KiB)**   | **Total** | **Operations** | **Average** | **Minimum** | **Maximum** | **Name** | | --- | --- | --- | --- | --- | --- | | 173672.0 | 6 | 28945.4 | 148.535 | 100000.0 | [CUDA memcpy DtoH] | | 60644.0 | 14 | 4331.7 | 0.004 | 28890.0 | [CUDA memcpy HtoD] | | 7.0 | 6 | 1.2 | 0.158 | 6.0 | [CUDA memcpy DtoD] |   *Yes, it is successful.*  *WHY: K could only be chosen from 3 or 7. Loop unrolling increases instruction-level parallelism and enhances performance by reducing loop overhead* | |
| * 1. What references did you use when implementing this technique? |
| *None* |