|  |  |
| --- | --- |
| **Name:** | *Hao Ren* |
| **NetID:** | *haor2* |
| **Section:** | *ZJ1* |

**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 | *0.246515 ms* | *0.83401 ms* | *0m1.542s* | *0.86* | | 1000 | *2.24226 ms* | *15.3607 ms* | *0m10.669s* | *0.886* | | 5000 | *11.5922 ms* | *40.737 ms* | *0m0.740s* | *0.871* | |
| 1. **Optimization 1:** IMPL\_INPUT\_UNROLLING **Shared memory matrix multiplication and input matrix unrolling (\*\*3 points\*\*)** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *The current access pattern and techniques are convolution which isn’t optimized and doesn’t have special hardware support. To further optimize the performance, we can first convert the whole computing pattern to matrix multiplication, hence a conv2mul kernel is needed to unroll the input matrix.*  *The expected improvement might not be that much (or degrades the performance),*  *but it leaves room for off-load the workload to tensor core.* |
| * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations? |
| *The optimization works by changing the computing method from convolution to matrix multiplication. NO, I don’t think it will improve the performance at this point since the unrolling can be expensive at this point and memory coalescing also happens in baseline when set properly. Since this is the first optimization, I will talk about the synergize in next optmization.* |
| * 1. List the Op Times, whole program execution time, and accuracy for batch size of 100, 1k, and 5k images using this optimization (including any previous optimizations also used). |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *1.45413 ms* | *0.939222 ms* | *0m0.297s* | *0.86* | | 1000 | *14.3275 ms* | *9.03383 ms* | *0m1.064s* | *0.886* | | 5000 | *71.0609 ms* | *44.6977 ms* | *0m3.832s* | *0.871* | |
| * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of).   NSYS |
| *No. Because the convolution kernel, as expected, takes longer time due to the fact that we have to load the data into shared memory first, and the copying between shared memory and global memory takes time even longer than pure global memory access due to the fact that we have better memory coalescing in pure global memory access pattern because of my design. The exec time will also be larger because the input unroll makes the memory usage about K\*K times larger than baseline. (As we can see from nsys, cudaMalloc and cudaMemcpy takes significantly longer time, conv\_forward\_kernel also takes longer time).*  *When we compare nsight compute GUI, we can see this optimization uses more memory bandwidth because we have much larger input because of input unroll (K\*K larger, around 49 times)* |
| * 1. What references did you use when implementing this technique? |
| *The lecture and MP4. Copilot also helps.* |
| 1. **Optimization 2:** IMPL\_UNROLLING\_KERNEL\_FUSION **Kernel fusion for unrolling and matrix-multiplication (requires previous optimization) (\*\*2 points\*\*)** |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I use kernel fusion for the original CPU unrolling to kernel input unrolling and I optimized the original matrix multiplication.* |
| 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations? |
| *The optimization works by running input unrolling and finding a better way to do kernel multiplication. I think this will improve the op time as well as the exec time as the work was done in CPU in the previous optimization now it’s done in GPU, and I tried the way with better memory coalescing. This optimization is built on previous optimization which gives a baseline of input unroll version implementation. In this optimization, the performance should be significantly improved.* |
| 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 | *0.234637 ms* | *1.08158 ms* | *0m0.181s* | *0.86* | | 1000 | *2.24452 ms* | *10.7243 ms* | *0m0.332s* | *0.886* | | 5000 | *11.0265 ms* | *53.3184 ms* | *0m0.925s* | *0.871* | |
| 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of). |
| *Yes. It significantly improves both exec time and Op time. For exec time it’s easy because we move CPU exec to GPU kernel exec. For Op time it’s actually quite dependent on batch size. When the batch size isn’t big enough, using shared memory would add extra overhead since the global memory isn’t fully exploited or the over-utilized global memory bandwidth isn’t great enough to degrade the performance of kernel fusion in this optimization, from the statistics, it could be seen that although the memory usage (global) in this optimization is greater than the previous optimization, the peformance is still better.* |
| 1. What references did you use when implementing this technique? |
| *Asked chatgpt and copilot for some help.* |

|  |
| --- |
| 1. **Optimization 3:** IMPL\_LOOP\_UNROLL **Tuning with restrict and loop unrolling (considered as one optimization only if you do both) (\*\*3 points\*\*)**   ***(Delete this section blank if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I added \_\_restrict\_\_ keyword to pointer and I also enumerate several cases of K (K=1,2,3,4,7) for unrolling purpose. The restrict pointer can help compiler improve the performance, and the unrolling can significantly improve the baseline approach by improving memory coalescing ability and helping compiler and reduce the use of extra registers which would cause many overheads.* |
| * 1. How does the optimization work? Did you think the optimization would increase performance of the forward convolution? Why? Does the optimization synergize with any of your previous optimizations? |
| *The restrict pointer can help compiler improve the performance by informing the compiler that this memory region is exclusive to this pointer and will not be referenced by other pointer in this kernel, allowing it to make optimization if available, and the unrolling can significantly improve the baseline approach by improving memory coalescing ability and helping compiler and reduce the use of extra registers which would cause many overheads. This would definitely improve the performance for forward convolution for previously stated reasons. This optimization is independent from previous 2 optimizations and should be compared with baseline implementation.* |
| * 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 | *0.197214 ms* | *0.540495 ms* | *0m0.173s* | *0.86* | | 1000 | *1.89108 ms* | *5.41518 ms* | *0m0.344s* | *0.886* | | 5000 | *10.1522 ms* | *26.9867 ms* | *0m1.024s* | *0.871* | |
| * 1. Was implementing this optimization successful in improving performance? Why or why not? Include profiling results from *nsys* and *Nsight-Compute* to justify your answer, directly comparing to your baseline (or the previous optimization this one is built off of). |
| The implementation is very successful in terms of performance (Op time) because the memory is better coalesced which can be shown that SOL memory cycles is less in current optimization. However, the total data transferred from memory should (trivially) be unchanged as shown. |
| * 1. What references did you use when implementing this technique? |
| *Chatgpt.* |
| 1. **Optimization 4:** IMPL\_FP16 **FP16 arithmetic. (note this can modify model accuracy slightly) (\*\*4 points\*\*)**   ***(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. |
| *FP16 arithmetic with restrict and loop unrolling because this set of optimizations should be most promising since all the operations can operate on 16bit width operands which will significantly reduce overhead. Although it’s note-worthy that the conversion of input and mask and output can introduce overhead.* |
| * 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? |
| *It works by transforming input matrix and mask matrix to FP16 accuracy. I think it will reduce the overhead of arithmetic calculation and reduce the memory throughput. This optimization is built from the loop unrolling optimization and is expected to reach the best performance.* |
| * 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 | *0.238143 ms* | *0.519883 ms* | *0m0.228s* | *0.86* | | 1000 | *2.22948 ms* | *5.2218 ms* | *0m0.316s* | *0.887* | | 5000 | *10.6534 ms* | *25.0527 ms* | *0m0.904s* | *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). |
| From result from NSYS and nsight compute we can see FP16 reduce the overhead of arithmetic operations as well as memory throughput as expected. This becomes my best implementation and is submitted with this version. |
| * 1. What references did you use when implementing this technique? |
| *GPT* |