|  |  |
| --- | --- |
| **Name:** | *Wei CHEN* |
| **NetID:** | *weic6* |
| **Section:** | *AL2* |

**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 | *0.198946 ms* | *0.579089 ms* | *0m0.184s* | *0.86* | | 1000 | *1.81063 ms* | *5.53631 ms* | *0m0.352s* | *0.886* | | 5000 | *8.97226 ms* | *27.5535 ms* | *0m0.940s* | *0.871* | |
| 1. **Optimization 1: *Weight matrix (kernel values) in constant memory*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *Weight matrix (kernel values) in constant memory.*  *Mask is not modified during the convolution, and therefore it can be saved in constant memory. Furthermore, accessing mask value saved in constant memory cost less cycles than accessing mask values saved in global memory in device. I expect using this strategy will make the convolution faster. Third, this optimization is one of the easy methods 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? |
| *The optimization used enables accessing mask value saved in constant memory, instead of global memory.*  *Yes, because accessing mask value saved in constant memory costs less cycles than accessing mask values saved in global memory in device. Furthermore, mask values will be reused when computing output feature maps, it obviates repetitive global memory access of mask values.*  *This optimization does not synergize with any of my previous optimizations.* |
| * 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 | *0.1707 ms* | *1.16038 ms* | *0m0.220s* | *0.86* | | 1000 | *1.5373 ms* | *11.8234 ms* | *0m0.300s* | *0.886* | | 5000 | *7.60822 ms* | *59.5705 ms* | *0m0.888s* | *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).   *This optimization is not successful in improving the overall performance in terms of total execution time.*  *Result of nsys:* |
|  |
| *From the picture, we can see that conv\_forward\_kernel takes 66.5 ms to run. It longer than the case without this optimization in the baseline, where conv\_forward\_kernel takes ~36.3 ms (see the result of nsys for baseline in the appendix) to run.*  *Result of Nsight:*  *Before optimization:*    *After optimization:*    *From the picture, we can see although the utilization of SM is higher, the memory utlization is lower than without optimization, which could be the reason why the total execution time is longer.* |
| * 1. What references did you use when implementing this technique? |
| *Lecture 7* |
| 1. **Optimization 2: *FP16 arithmetic (4 points)*** |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *FP16 arithmetic. \_\_half type takes less space in memory, and therefore transferring \_\_half type will require less time in the kernel. Therefore I expect this strategy will make the kernel run faster. FP16 is also easy 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? |
| *First, I convert the float input, mask to \_\_half type, then copy them to device global memory.*  *I think the optimization would increase the performance of the kernel, since \_\_half takes less space, and therefore I expect them to transfer faster in GPU.*  *This optimization does not* synergize with any of my previous optimizations. |
| 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 | *0.199584 ms* | *1.15215 ms* | *0m0.166s* | *0.86* | | 1000 | *2.32835 ms* | *8.96509 ms* | *0m0.337s* | *0.887* | | 5000 | *8.91745 ms* | *29.1677 ms* | *0m0.964s* | *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). |
| *This Implementation does not improves the performance, in terms of total execution time.*  *Result of nsys:*    *From the picture, we can see that conv\_forward\_kernel uses 38.3ms, which is slightly longger than the baseline (*takes ~36.7 ms*).*  *Result of Nsight:*  *Before optimization:*    *After Optimization:*  *From the picture, we can see that the memory optimization after optimization drops from 97.38% to 88.86%, which can explain the longer total execution time after optimization.* |
| 1. What references did you use when implementing this technique? |
| *CUDA ToolKit Documentation* |

|  |
| --- |
| 1. **Optimization 3: *Input channel reduction: atomics  (2 point)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I choose to implement Input channel reduction: atomics because it is easy to implement and I expect this strategy to work because now multiple threads can work on the same output element.* |
| * 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 enables C threads handle one output element collectively. Each threads use attomicAdd to operate on the same output array element. I expect this optimization to improve the performance because this optimization leverages multiple threads to handle one output element and also unnecessitates the most outer for-loop in the forward convolution kernel.*  *This* optimization does not synergize with any of my previous optimizations. |
| * 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 | *0.907786 ms* | *4.44102 ms* | *0m0.276s* | *0.86* | | 1000 | *8.91858 ms* | *43.8966 ms* | *0m0.363s* | *0.886* | | 5000 | *44.3847 ms* | *218.811 ms* | *0m0.992s* | *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 of this optimization is not very successful, as the* Total Execution Time is slightly bigger than the baseline.  *Result of nsys:*    *However, this optimization optimization does speed up the execution of conv\_forward\_kernel, which can be explained by the reasons I mentioned in the subquesiton (b).*  *Result of Nsight:*  *Results before optimization: Baseline*        *Results after optimization: adding Atomics*        *From the picture, we can see that after optimization, the utilization of SM and Memory decrease. Pipes are also underutilized. Memory workload after optimization shows that the L1 hit rate (65.5%) is lower than before (95.35%). These may be due to the introduction of a new dimention threadIdx.z. making the memory access in my code even less efficient, with the global Memory access more uncoalesced.* |
| * 1. What references did you use when implementing this technique? |
| *Lecture 18 about atomics.* |
| 1. **Optimization 4: *Input channel reduction: tree (3 point)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I choose to implement Input channel reduction: tree. In my implementation, I let c threads calculate the partial sum for one output element. Using tree reduction, the final sum can be calculated for that output element.* |
| * 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? |
| *For a specific output element, C threads are responsible for calculating. Using tree reduction, the final sum can be calculated efficiently for that output element. I think the optimization would increase the performance of forward convolution, since it leverages the parallelism by using multiple threads when calculating the sum of an array.*  *The implementation does not synergize with any previous 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 | *0.201159 ms* | *0.714096 ms* | *0m0.185s* | *0.86* | | 1000 | *1.84839 ms* | *6.96857 ms* | *0m0.328s* | *0.886* | | 5000 | *9.17274 ms* | *34.7842 ms* | *0m0.888s* | *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).   *In terms of total execution time, this optimization does not improve, compared to the baseline where no optimization is applied.*  Result of nsys: |
| *From the picture, we can see conv\_forward\_kernel takes ~56 ms, which is slower than the baseline which takes* ~36.7 ms to run.  *Result of Nsight*    *From the picture, we see that the memory utilization is 84.55%, lower than the baseline (memoery utilization is 97.38%). The drop in memory utilization. Also, tree reduction introduces warp divergence which leads to a longer kernel execution time, as it can be seen from the Warp Cycles Per Executed Instruction= 20.43, as shown in the picture below.*    *Without optimization, the Warp Cycles Per Executed Instruction = 19.80, as shown blow.* |
| * 1. What references did you use when implementing this technique? |
| *Tree reduction: Lec 15;*  *dynamically array allocation: https://stackoverflow.com/questions/5531247/allocating-shared-memory%5B/url%5D* |
| 1. **Optimization 5: *Sweeping various parameters to find best values (block sizes, amount of thread coarsening) (0.5 point)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *This one is the easiest one to do, where I can vary the amount of thread in one block* |
| * 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? |
| *My baseline use TILE\_WIDTH = 8. I decided to try other values as well and see if there is any better performance, since the size of output feature maps is not necessarily divisible by 8. Upon testing TILE\_WIDTH = 4,6,7,9,16,18,20,21,22,24,32 [results are listed in the appendix], I find that TILE\_WIDTH=18 shows a better result as total execution time is slightly lower than the baseline.*  *This optimization does not synergize with any of my previous optimization.* |
| * 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 | *0.234712 ms* | *0.53655 ms* | *0m0.198s* | *0.86* | | 1000 | *2.2143 ms* | *5.12418 ms* | *0m0.301s* | *0.886* | | 5000 | *10.9948 ms* | *25.499 ms* | *0m0.876s* | *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 optimization successful in improving performance, as the total execution time is lower than the baseline.*  *Result of nsys:*    *From the pic, we can see that the time for conv\_forward\_kernel is 36.6ms, which is slightly faster than the baseline.* |
| * 1. What references did you use when implementing this technique?   ***Lecture*** |

|  |
| --- |
|  |

**Appendix:**

nsys Result of the baseline:

*A screenshot of a computer screen

Description automatically generated*

Sweeping various *TILE\_WIDTH:*

*TILE\_WIDTH = 4*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 6*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 7*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 9*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 10*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 16*

*A screenshot of a computer program

Description automatically generated­*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 18*

*A screenshot of a computer program

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 20*

*A screenshot of a computer program

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 21*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer

Description automatically generated*

*TILE\_WIDTH = 22*

*A screen shot of a black screen

Description automatically generated*

*A screenshot of a computer

Description automatically generated*

*TILE\_WIDTH = 24*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*

*TILE\_WIDTH = 32*

*A screenshot of a computer

Description automatically generated*

*A screenshot of a computer screen

Description automatically generated*