|  |  |
| --- | --- |
| **Name:** | *Jiawei Zhang* |
| **NetID:** | *673400538* |
| **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 | *0.202112 ms* | *0.670636 ms* | *0m1.922s* | *0.86* | | 1000 | *1.88994 ms* | *6.57501 ms* | *0m10.786s* | *0.886* | | 5000 | *9.38285 ms* | *33.1992 ms* | *0m51.317s* | *0.871* | |
| 1. **Optimization 1: Weight matrix (kernel values) in constant memory (0.5 points)** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *Using constant memory to store convolution kernel values. I choose this because convolutional kernels are constant during calculation and, in addition, relatively small so that can be accommodated into constant memory. In this case, we can copy the values into constant memory where the memory access speed is much higher than in global memory.* |
| * 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? |
| *Constant memory is a memory type in CUDA architecture, where data cannot be changed through the running but the access speed is much faster than global memory. Because the kernel data is unchangeable through convolution process, so it is ideally to use constant memory to load kernel data. I thought it should increase the performance, because the constant memory access speed is much higher than the global memory access speed. This optimization doesn’t synergize with 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 | *0.203602 ms* | *0.62055 ms* | *0m2.752s* | *0.86* | | 1000 | *1.57822 ms* | *5.78779 ms* | *0m10.182s* | *0.886* | | 5000 | *7.77181 ms* | *29.1623 ms* | *0m51.109s* | *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). |
| *According to the results, it improved the performance on batch 5000 from 9.38+33.19 ms to 7.77+29.16 ms. We compared our Nsight and Nsight-Compute results with the original ones.*  *I think this improvement is basically from the lower memory access time.*  *Original profiling results:*      *After optimization results:* |
| *From the system level profiling, we can see that the kernel operation time is reduced. From the kernel level profiling, we can further know that this reduction is mainly from the higher instruction execution speed, which is because of the fast data access due to constant memory usage.* |
| * 1. What references did you use when implementing this technique? |
| *Lecture slides for convolutions and MP code.* |
| 1. **Optimization 2: Tiled shared memory convolution (2 points)** |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chosen tiled shared memory as an optimization point. I choose this because there are a lot of reuses during the convolution process. Using shared memory can reduce the time to revisit some data.* |
| 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? |
| *To use shared memory, I first split the output data into some little ‘tile’s, each tile is loaded by a single block into a block-wise shared memory. Every thread in the block will share the same data during process. When calculating 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 | *1.19127 ms* | *4.3784 ms* | *0m1.592s* | *0.86* | | 1000 | *12.3704 ms* | *44.8784 ms* | *0m10.690s* | *0.886* | | 5000 | *65.2961 ms* | *220.938 ms* | *1m0.081s* | *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). |
| *According to the results, we can see that the Op time has largely increased. I think this result comes from two aspects: 1. We can only use tiles whose widths are much smaller because of the shared memory restriction. 2. When loading data into shared memory, each thread needs to load a square of K \* K data which decreased the coalescence.*  *After optimization profiling:*        *From the kernel level profiling, we can see the compute and memory throughput is extremely lower than before (only use constant memory). I believe this result comes from the very low warp occupancy according to the kernel level profiling results. But I don’t know what causes that. Even though my implementation may cause some performance degradation, but I didn't expect the warp occupancy to decrease by so much.* |
| 1. What references did you use when implementing this technique? |
| *Lecture slides and MP code.* |

|  |
| --- |
| 1. **Optimization 3: *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. |
| *I chosen the input unrolling and shared matrix multiplication as an optimization. Since the convolution function can be regarded as a matrix multiplication. To transform the convolution method to matrix multiplication, we need to do input unroll first, which make the overlapping elements to expand.* |
| * 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, the input data will be transformed by a unroll function, which will map all data into a new matrix. The new matrix has W\_out \* H\_out columns and K \* K \* C rows. For each output data, the corresponding K \* K input data from C different channels will be placed in its column. After the unrolling, we can easily use shared memory matrix multiplication to conduct convolution process.* |
| * 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 | *1.34161 ms* | *1.16739 ms* | *0m1.539s* | *0.86* | | 1000 | *12.9572 ms* | *110.815 ms* | *0m12.438s* | *0.886* | | 5000 | *63.9139 ms* | *54.0466 ms* | *0m51.404s* | *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). |
| *According to the results, the optimization did not improve the performance. I think this is mostly because of two aspects: 1. I use two kernel calls: one for input unroll and one for matrix multiplication. The interval time between two kernel calls is very time consuming compared to computation.*  *2. The unroll function need to revisit data in global memory too many times.*  *After optimization profiling:*    *From the profiling results, we can know that the compute and memory throughput of unroll kernel is too low.* |
| *Further, from the compute workload analysis, we can know the instruction execution speed is very low due to frequent global memory access.*  *An interesting fact is Op time 2 on batch size 1000 is very high (over 110 ms), which surprised and confused me.* |
| * 1. What references did you use when implementing this technique? |
| *Lecture slides and MP code.* |
| 1. **Optimization 4: *Input Channel Reduction-Atomics (2 points)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose channel reduction using atomic operation. I chose this because I found that the channel dimension for convolution is potentially parallel. We can compute convolution in each channel independently and then sum them up to take advantage of such independency. Atomic operation is the most straight forward way 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? |
| *To make the channel level computation parallel, I add another dimension in blockDim. The new dimension in blockDim.z is to compute convolution in each channel independently. After computation, each thread add its result into the corresponding output address. During the write back, we use AtomicAdd function to avoid each thread conflicting resources.*  *I think it might not increase the performance because the channel number is relatively small in our case, so this parallelism may not bring positive effects.*  *This optimization does not synergize with previous optimization. I implement this om top of constant memory 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 | *0.198991 ms* | *0.853776 ms* | *0m1.496s* | *0.86* | | 1000 | *1.89801 ms* | *8.99917 ms* | *0m10.631s* | *0.886* | | 5000 | *9.41942 ms* | *45.6833 ms* | *0m51.775s* | *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). |
| *According to the results, it doesn’t improve performance.*  *Profiling after optimization:* |
| *From the profiling results, we can know there aren’t much differences between the baseline and optimization. But the SOL SM and Executed Ipc is indeed decreased. I guess this is just because the synchronization wait due to atomic operations.* |
|  |
| * 1. What references did you use when implementing this technique? |
| *Lecture slides about scan.* |
| **Optimization 5: *Input Channel Reduction-Tree (3 points)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose channel reduction using tree reduction. Different from the atomic reduction method, tree reduction does not require the program to run only one thread at each atomic operation. Thus, tree method could reduce the overhead in atomic reduction method.* |
| * 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 tree reduction method, each thread will iteratively add two number into one address. After each iteration, the number of active threads will reduce to half. Finally, only the first thread adds two number into the first place which turns out to be the final sum result.*  *I think it probably cannot increase the performance, because although it releases more parallelism, it still introduces more overhead since it needs more loops to do the reduction.*  *It doesn’t synergize with previous optimizations but I implemented it on top of constant 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 | *0.211212 ms* | *0.94025 ms* | *0m1.524s* | *0.86* | | 1000 | *2.08956 ms* | *9.91084 ms* | *0m10.604s* | *0.886* | | 5000 | *10.3921 ms* | *50.1881 ms* | *0m51.667s* | *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). |
| *According to the results, it does not improve the performance.*  *Profiling after optimization:*          *As the results from atomic reduction, there are not much differences between tree reduction and constant memory optimization. But the SOL SM and Executed lpc are further slightly decreased compared with atomic reduction. I think this is because the introduced overhead (more loops and more shared memory access) slows down the instruction execution speed.* |
| * 1. What references did you use when implementing this technique? |
| *Lecture slides about scan and MP code.* |
| **Optimization 6: *Tuning with restrict and loop unrolling (3 points)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *I chose the Tuning with restrict and loop unroll as an optimization. I found that the convolution operation needs several loops and the some of their iteration times are predictable: the kernel size is smaller than 5, the channel number is smaller than 6, etc. Thus, it is easily to use loop unroll to increase performance. In addition, convolution needs to operate different float pointers, which is suitable to use restrict word to increase compiling performance.* |
| * 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\_\_ keyword is used to tell to the compiler that the different pointers are pointed to different objects, which can help the compiler to optimize the instruction arrangement and in turn improve the program performance. The unroll directive is used to unroll a specific loop with a given integer. By unrolling the loop, the compiler could also optimize the directions for better performance.*  *I think this optimization could improve the performance. Because there are many loops and address visit operation in the convolution process, the improvement room for compiler is relatively large.* |
| * 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 | *0.163474 ms* | *0.527474 ms* | *0m1.598s* | *0.86* | | 1000 | *1.39799 ms* | *5.03846 ms* | *0m10.549s* | *0.886* | | 5000 | *6.87317 ms* | *25.6123 ms* | *0m51.265s* | *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 improved the performance. I think the optimizations by the compiler took effects. There are 3 unroll parameters that need to be tuned: loop for channel C, loop for kernel row p and loop for kernel column q.  I have tried some different sets of parameters, the experiments are as follows:  *Unroll parameter experiments: (results are Op time on batch size 5000)*  *c 6 p 5 q 5 7.71642 ms 26.167 ms*  *c 3 p 5 q 5 7.508 ms 30.6715 ms*  *c 3 p 3 q 3 7.41564 ms 26.4705 ms*  *C 1 p 5 q 5 6.9425 ms 26.0933 ms*  *C 1 p 3 q 3 6.90675 ms 25.7142 ms*  I chose the parameter setting: 1 for C unrolling, 3 for p unrolling, 3 for q unrolling.  Profiling results:      From the system level profiling results, we can see the kernel time is smaller than every previous implementation. The GPU Speed of Light results show that both the memory throughput and L1 or L2 cache hitting rate are higher, which I believe gets benefits from the compiler to optimize the arrangement of instructions. |
|  |
| * 1. What references did you use when implementing this technique? |
| *CUDA C++ Programming Guide*  *\_\_restrict\_\_:* [CUDA C++ Programming Guide (nvidia.com)](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=restrict#restrict)  *#pragma unroll:* [CUDA C++ Programming Guide (nvidia.com)](https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=restrict#pragma-unroll) |