|  |  |
| --- | --- |
| **Name:** | *Shao-Chian Chen* |
| **NetID:** | *scchen4* |
| **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 10k images from your basic forward convolution kernel in milestone 2. This will act as your baseline this milestone. |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 (ms) | Op Time 2 (ms) | Total Execution Time | Accuracy | | 100 | *0.161288* | *0.580886* | *0m1.172s* | *0.86* | | 1000 | *1.6204* | *6.28107* | *0m10.421s* | *0.886* | | 10000 | *14.4788* | *57.9115* | *1m46.132s* | *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 (kernel values) in constant memory (1 point)*  *This is the first thing to come in mind when doing CUDA optimization. And is probably the easiest to do in the list.* |
| * 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? |
| *Put mask on constant memory.*  *It did increase performance for batch size 10000. Since batch is being parallelized as well, device\_mask is being called 10000 times from global memory at any moment. By putting it in the constant memory significantly decreased op time. No, it is the first thing to do so no previous optimizations.* |
| * 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 (ms) | Op Time 2 (ms) | Total Execution Time | Accuracy | | 100 | *0.167836* | *0.573577* | *0m1.276s* | *0.86* | | 1000 | *1.47797* | *5.62968* | *0m10.176s* | *0.886* | | 10000 | *14.5508* | *52.1175* | *1m38.353s* | *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). |
| *Baseline:*    *Optimization 1:*    *A huge difference in memory usage, which leads to less bandwidth bottleneck.* |
| * 1. What references did you use when implementing this technique? |
| *MP4, which is the only MP we used constant memory.* |
| 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. |
| *Tiled shared memory convolution (2 points)*  *It’s an experiment between taking time to put input data into shared memory vs. reading inputs from global memory everytime.* |
| 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? |
| *Since taking input values from global memory may cause bandwidth bottleneck, so the easiest method is to put everything into shared memory. But there are downsides for this method. First is that it takes extra space. Second is that moving data from global memory to shared memory also takes time. It is important to see whether keeping them in the global memory would be faster, or do the extra work for faster data read gives us better result.*  *It turns out it did not do as well as I thought. Total execution time dropped a little, but op time is significantly larger than before.* |
| 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 (ms) | Op Time 2 (ms) | Total Execution Time | Accuracy | | 100 | *0.32255* | *0.931614* | *0m1.215s* | *0.86* | | 1000 | *3.1046* | *61.6793* | *0m9.908s* | *0.886* | | 10000 | *30.8797* | *91.578* | *1m35.862s* | *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). |
| *Not really. The reasons were stated above. Apparently, this trade off isn’t good enough. It didn’t make a big difference after implementation.*  *Optimization 1:*    *Optimization 2:*    *Memory usage again dropped significantly, but SM utilization reached almost 90%.* |
| 1. What references did you use when implementing this technique? |
| *MP4, since it was where we did tiled convolution. However, MP4 is slightly different than this time. We kept the image the same size then, but shrunk our image this time.* |

|  |
| --- |
| 1. **Optimization 3: *Input channel reduction: tree (3 point)***   ***(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. |
| *Input channel reduction: tree*  *I thought we have to do reduction for a number of* ***(TILE\_WIDTH \* TILEWIDTH \* CHANNEL)*** *elements. What I read from Campuswire was that I only had to do this across channels. Meaning that we only have to do reduction on a maximum of* ***CHANNEL*** *elements, which is 4, for each thread.* |
| * 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? |
| *No it didn’t do well at all. We have a very small* ***Channel*** *size of 4, which means we are only performing this amazing summation trick on 4 elements. It wouldn’t take long if we just sum them up together.*  *Also, I’m building this on top of the previous two optimizations, and leads to a huge increase in time, which is awful. I guess it comes from increasing the number of threads in a block, which makes SM does a lot more work than it used to.* |
| * 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 (ms) | Op Time 2 (ms) | Total Execution Time | Accuracy | | 100 | *1.98987* | *4.03512* | *0m1.149s* | *0.86* | | 1000 | *20.3215* | *41.0832* | *0m10.551s* | *0.886* | | 10000 | *204.345* | *413.095* | *1m42.630s* | *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). |
| *Strangely, I got it running on batch size 10000 on default queue, but couldn’t get it working on the exclusive queue. Couldn’t complete this part.* |
| * 1. What references did you use when implementing this technique? |
| *MP5.1, that was where we did reduction tree. Although we are only performing it on 4 elements, I still looked that up.* |
| 1. **Optimization 4: *Input channel reduction: atomics (2 point)***   ***(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. |
| *Same reason as* ***Optimization 3****. However, doing* ***atomicAdd*** *only assures that the addition wouldn’t be interrupted, it doesn’t speed up the sequential add process. The only slight change I did was to atomically add them to another shared memory first, then write the whole thing back from shared memory to global memory at a time. This way I wouldn’t have to write to global memory too many times and cause memory bandwidth bottleneck.* |
| * 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 should be even worse than using tree for reduction. In tree, if completely parallelized, we only have to do two sequential add operations for 4 elements, but has to do 3 in atomic add.*  *This optimization only synergizes with the first two optimizations.* |
| * 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 (ms) | Op Time 2 (ms) | Total Execution Time | Accuracy | | 100 | *2.02818* | *4.16258* | *0m1.171s* | *0.86* | | 1000 | *18.1811* | *88.8141* | *0m10.926s* | *0.886* | | 10000 | *195.871* | *383.485* | *1m44.547s* | *0.8704* | |
| * 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). |
| *Same thing as tree, I got it running on batch size 10000 on default queue, but couldn’t get it working on the exclusive queue. Couldn’t complete this part.* |
| * 1. What references did you use when implementing this technique? |
| *MP6, because I have to look up how* ***atomicAdd*** *works.* |
| 1. **Optimization 5:** ***Fixed point (FP16) arithmetic. (4 points)***   ***(Delete this section if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *Fixed point (FP16) arithmetic.*  *Ever since I learned to turn a picture from float to char for smaller memory usage from MP5.2, I knew this is what I should do in the Project. Although I don’t know whether it is going to improve in time.* |
| * 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 turns all floating point variables into the \_\_half type (FP16, or floating point 16), so that it takes only half of the memory than floating point, which is 32 bits. It also provides a reduction in memory bandwidth requirements. I think it should help on storing data, so it should work well on shared memory tiling, but won’t do as much for non-shared memory convolution.*  *I implemented this on optimizations 1 and 2.* |
| * 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 (ms) | Op Time 2 (ms) | Total Execution Time | Accuracy | | 100 | *2.09582* | *2.80601* | *0m1.225s* | *0.86* | | 1000 | *3.56342* | *10.6752* | *0m10.858s* | *0.887* | | 10000 | *34.046* | *228.9* | *1m43.515s* | *0.8716* | |
| * 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). |
| *It didn’t do well on the tiled convolution as I thought. It even increased both op time. Apparently turning all input data from 32 bits floating point to 16 bit takes time. The time increased from it is more than getting a reduction in bandwidth requirement.*  *Without FP16:*    *With FP16:* |
| * 1. What references did you use when implementing this technique? |
| *Campuswire and from the CUDA Toolkit Documentation:*  [*https://docs.nvidia.com/cuda/cuda-math-api/group\_\_CUDA\_\_MATH\_\_\_\_HALF\_\_MISC.html*](https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html)  *Also* [*https://forums.developer.nvidia.com/t/speed-of-fixed-versus-floating-point/31260*](https://forums.developer.nvidia.com/t/speed-of-fixed-versus-floating-point/31260) |
| 1. **Optimization 6:** ***Sweeping various parameters to find best values (block sizes, amount of thread coarsening) (1 point)***   ***(Delete this section if you did not implement this many optimizations.)*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| *Sweeping various parameters to find best values*  *Sometimes the little things are what matters. No one’s going to know what block size is going to fit best without trial and error.* |
| * 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? |
| *Basically I went through all possible combinations between choosing tiling, FP16, and different block sizes. (Since there might be something wrong about reduction, so I did not consider it here). We know that the first layer has 86\*86 elements in one channel, and the second layer has 40\*40. So I came up with TILE\_WIDTH 22 so that not a lot of threads are stalling for not being chosen for both layers (22 \* 2 = 44, 22 \* 4 = 88).* |
| * 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). **(The following is time for running TILE\_WIDTH 22 on FP16, Tiled Shared Memory Convolution, and Weight matrix (kernel values) in constant memory)** |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | | Batch Size | Op Time 1 (ms) | Op Time 2 (ms) | Total Execution Time | Accuracy | | 100 | *0.285044* | *0.792678* | *0m1.125s* | *0.86* | | 1000 | *2.74269* | *7.80027* | *0m9.769s* | *0.887* | | 10000 | *27.171* | *78.1154* | *1m36.683s* | *0.8716* | |
| * 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). |
| *It did speed up with different TILE\_WIDTH. However, using 20 was actually the fastest among 8, 16, 20, and 22. Also, doing convolution without tiling and shared memory still holds its record for being fastest. For the non-shared memory version, TILE\_WIDTH 20 works better than all other TILE\_WIDTHs as well.*  *I did not run Nsight-Compute for this, but I had several testing Op times listed below.*  *I also tested out different TILE\_WIDTHs for the* ***Constant*** *version.*  *TILE\_WIDTH comparison between* ***FP16*** *+* ***Tiled*** *+* ***Constant****:*   |  |  |  |  | | --- | --- | --- | --- | |  | *C:\Users\jacky\AppData\Local\Microsoft\Windows\INetCache\Content.Word\fp16 tile 16.png* |  | *C:\Users\jacky\AppData\Local\Microsoft\Windows\INetCache\Content.Word\fp16 tile 22.png* | | ***FP16*** *+* ***Tiled*** *+* ***Constant***  *TILE\_WIDTH 8*  *Total Op time:*  *137.1419* | ***FP16*** *+* ***Tiled*** *+* ***Constant***  *TILE\_WIDTH 16*  *Total Op time:*  *107.1676* | ***FP16*** *+* ***Tiled*** *+* ***Constant***  *TILE\_WIDTH 20*  *Total Op time:*  *105.2864* | ***FP16*** *+* ***Tiled*** *+* ***Constant***  *TILE\_WIDTH 22*  *Total Op time:*  *105.2864* |   *We can see from the above results, for* ***FP16*** *+* ***Tiled*** *+* ***Constant*** *optimizations, TILE\_WIDTH 16 and TILE\_WIDTH 22 are both better than TILE\_WIDTH 8, TILE\_WIDTH 20 is actually the best in total op time.*  *TILE\_WIDTH with only* ***Constant*** *optimization:*   |  |  |  |  | | --- | --- | --- | --- | |  |  |  |  | | ***Constant***  *TILE\_WIDTH 8*  *Total Op time: 93.9219* | ***Constant***  *TILE\_WIDTH 16*  *Total Op time: 82.0369* | ***Constant***  *TILE\_WIDTH 20*  *Total Op time: 70.6014* | ***Constant***  *TILE\_WIDTH 22*  *Total Op time: 72.9997* |   *TILE\_WIDTH 20 is still the best even under this condition.*  ***In Optimization 1, the op time I wrote was in TILE\_WIDTH 8. Since Op time isn’t consistent enough, I couldn’t recreate total op time under 70ms. But if I used TILE\_WIDTH 20 that time, I can conclude that it should run even faster.*** |
| * 1. What references did you use when implementing this technique? |
| *None*   1. *Honorable mention:* ***Maximize thread usage when doing K\*K addition in shared memory version***    1. *How does this optimization work?*   *For shared memory, I chose* ***TILE\_WIDTH + MASK\_ WIDTH - 1*** *as block width. This is strategy 2 from lecture 8. After storing input into shared memory, the additional threads then start to stall. I tried reorganizing them so that they also help the addition for the K\*K loop. However, it did not work at last.* |