|  |  |
| --- | --- |
| **Name:** | *Wesley Wu* |
| **NetID:** | *wwu70* |
| **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 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.248519 ms* | *0.903248 ms* | *1.281s* | *0.86* | | 1000 | *2.37707 ms* | *8.86233 ms* | *11.111s* | *0.886* | | 10000 | *23.5532 ms* | *86.059 ms* | *1m42.330s* | *0.8714* | |
| 1. **Optimization 1: *Fixed point (FP16) arithmetic*** |
| * 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| I chose to implement the FP16 optimization because it peaked my interest that changing the arithmetic type could potentially speed up execution. |
| * 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 reducing the precision of floating point numbers by storing the same float number (typically 32 bits) in half the number of bits (16 bits), which speeds up arithmetic as well as saves memory since less bitsare needed for each float.  I thought this optimization would increase performance of the forward convolution because I would think that doing binary arithmetic, especially operations like multiplication, would take less cycles on the lower precision FP16 types based on what I’ve learned about multipliers from ECE 385 and ECE 411 – many multipliers typically use an add-shift algorithm, which takes longer the more bits there are in an operand. |
| * 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 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.310123 ms* | *1.15521 ms* | *1.221s* | *0.86* | | 1000 | *2.96759 ms* | *11.3375 ms* | *10.266s* | *0.887* | | 10000 | *29.4949 ms* | *112.365 ms* | *1m40.249s* | *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).  Compared to the baseline kernel, this optimization actually didn’t help improve performance in terms of time. From the table above compared to the baseline kernel, the op times actually increased for each batch size, but the total execution time decreased.  Baseline kernel:     After FP16 optimization: |
| Based on these analyses from Nsight-compute, we can see that the theoretical reduction in memory usage was observable, but it was not to the extent that I would’ve expected based on how the optimization works. Both the % of mem busy and % memory usage were cut down by a good amount, but not nearly the half I was expecting, maybe because of the conversion overhead. Also, the max bandwidth was much lower, most likely because the fp16 takes up less space in memory. And also, the amount of SMs used on the GPU increased (i.e. greater hardware utilization). So, overall, while it didn’t improve the op times, this optimization certainly did improve memory usage. |
| * 1. What references did you use when implementing this technique? |
| CUDA Documentation on FP16 for conversions This blog: <https://ion-thruster.medium.com/an-introduction-to-writing-fp16-code-for-nvidias-gpus-da8ac000c17f> |
| 1. **Optimization 2:** Weight matrix (kernel values) in constant memory |
| 1. Which optimization did you choose to implement and why did you choose that optimization technique. |
| I chose to put the weight matrix in constant memory becauseI expected the caching aspect of it to help even if just a little bit. |
| 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 the GPU, since constant cachelines are read-only, and the weight matrix is read-only, this enables higher thoroughput accesses than L1 caches since the weight matrix values are accessed very frequently in the multipliy-adds.  I thought the optimization would increase the performance by reducing the latency of memory accesses, and when I tested this with the FP16 optimization, it did not synergize well (in fact, it made the already worse op times even worse), so I will be comparing the op times listed below to the baseline op times. |
| 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 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.227844 ms* | *0.799336 ms* | *1.260s* | *0.86* | | 1000 | *2.15439 ms* | *7.8373 ms* | *10.141s* | *0.886* | | 10000 | *21.2987 ms* | *78.1107 ms* | *1m37.631s* | *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).  Compared to the baseline kernel, this optimization did improve performance, as the op times in the tables are slightly less than those in the baseline kernel table, and the total execution times are also slightly less than those from the kernel baseline table.   Baseline kernel:     After constant memory weight matrix optimization:     From these graphs, we can see that the memory usage decreased, as well as the % of mem busy. In addition, there was also a significant boost in the memory throughput, and the memory also had a lower max bandwidth since less information was loaded once the kernel was copied to constant memory. So, there was some benefit to the memory side of the kernel from copying the mask to constant memory, as we would’ve expected. And, since there was less stress on the memory, there was a greater % of SMs busy. Plus, without the reads from global memory, we are able to execute more IPC, as shown in the Nsight-compute graphs. So, this optimization improved both op times as well as memory utilization. |
|  |
| 1. What references did you use when implementing this technique? |
| I followed the example on the lecture 7 slides to implement this technique. |

|  |
| --- |
| 1. **Optimization 3:** Tuning with restrict and loop unrolling   ***(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 chose to implement tuning because it sounded fairly simple while also providing great benefit based on what I read online about it. |
| * 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? |
| Loop unrolling works a lot like a synthesized hardware for loop – it’s basically replicated logic, so then the GPU doesn’t need to evaluate an assembly branch instruction and add to a counter every time it runs the for loop, drastically reducing the overhead on the GPU for executing the for loop.  The restrict keyword makes sure that any pointers don’t accidentally alias, or point to the same address, so that any writes to an address given by pointer will not cause and bad data reads in subsequent instructions that read from that address.  It synergized really well with the constant memory mask optimization, showing greater improvements to op times on the larger batch sizes. |
| * 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 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.330522 ms* | *0.607221 ms* | *1.321s* | *0.86* | | 1000 | *1.94942 ms* | *4.49842 ms* | *11.388s* | *0.886* | | 10000 | *17.9008 ms* | *43.2705 ms* | *1m44.644s* | *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). |
| (This one is built off of the constant memory kernel values optimization) Compared to the constant memory kernel from the section above, this provided great improvement to performance – in fact, it helped me to achieve sub-70ms total op times and also improved the total op times.  Previous (constant memory weight matrix) kernel:    Kernel after restrict and unrolling:    The main reason for the significant speedup with unrolling and restrict is the significant reduction in number of instructions executed (but mainly unrolling) – since unrolling significantly reduces the number of add (IADD and IMAD) instructions, and especially cuts down on the number of branch (BRA) instructions, there are overall much less instructions that are executed across all of the GPU, which is why there was such a sharp decrease in the op time. Also, there are much less load instructions overall, as expected with the \_\_restrict\_\_ keyword, as the keyword notifies the compiler that the pointers for the input and output do not overlap (and mask, however since I’m stacking this on top of the constant memory weight values optimization, the \_\_restrict\_\_ keyword has no effect on the mask parameter), thus less overall reads are needed. From the assembly instruction breakdown, we can easily see why the op time was greatly reduced with this two-part optimization.  On the memory side, as we can see from the comparison of the kernel with and without \_\_restrict\_\_, the memory has much higher thoroughput and is closer to achieving its max bandwidth since the kernel doesn’t have to assume pointer aliasing. |
| * 1. What references did you use when implementing this technique? |
| <https://developer.nvidia.com/blog/cuda-pro-tip-optimize-pointer-aliasing/>  <https://www.nvidia.com/docs/IO/116711/sc11-unrolling-parallel-loops.pdf> |
| 1. **Optimization 4: Using Streams to overlap computation with data transfer**   ***(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 chose to implement streams for software pipelining because I wanted to appreciate the software parallel to what I am currently implementing in 411, as I really enjoyed listening in on the lecture on streams for that reason as well. |
| * 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? |
| By breaking up the computation and data transfer into streams, what happens is that when the GPU is performing the calculation, it uses the available memory resources to transfer input and output data at the same time, allowing for even greater parallel execution of instructions than before. It essentially makes greater use of the available hardware at any moment. I thought it would increase performance of the forward convolution because in theory it would increase hardware utilization and no hardware on the GPU would be idiling. It synergizes well at larger batch sizes, as shown in the table below. |
| * 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 | Op Time 2 | Total Execution Time | Accuracy | | 100 | *0.184878 ms* | *0.452997 ms* | *1.215s* | *0.86* | | 1000 | *1.63703 ms* | *3.81498 ms* | *10.466s* | *0.886* | | 10000 | *16.9798 ms* | *43.1093 ms* | *1m42.647s* | *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). |
| (I combined this optimization with constant memory weight matrix values and tuning optimizations)  Based on the table in the previous section, this actually barely edged out an improvement over the combination of constant memory mask and tuning with restrict and unroll for the larger batch sizes (1000, 10000).    (note: these runs are when I put all 3 necessary API calls and kernel call into the prologue function, whereas I recorded the op times for the table when I separated the API calls among the 3 host functions)  We can also see from the side-by-side comparison of the runs (left = with streams, right = without streams) that adding in streams to overlap data transfer and computation really helped the layer times, and this is because the layer times. We can also see why this is in the comparison of memcpy calls and kernel calls between a version of the dnn with streams and a version without the streams:  Without streams:  With streams: |
| As we can see, there is clearly overlap between kernel calls and memcpy calls, and sometimes even between memcpy calls, which is the reason why we see a sharp decrease in layer times. So, while this optimization did not improve op times, it clearly helped the layer times by making as much utilization of the GPU as possible at any point in time. |
| * 1. What references did you use when implementing this technique? |
| I used the lecture 22 slides as well as this blog from NVIDIA: <https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/> |