ECE 408, Fall 2023 **ECE 408/CS 483 Milestone 3 Report** Dec. 1, 2023

Name: Shurong Wang
NetID: shurong3
Section: ZJ1

Baseline

List Op Times, whole program execution time, and accuracy for batch sizes of 100, 1k, and 5k images from your basic forward convolution kernel in milestone 2. This will act as your baseline for 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	0.32139 ms	2.09556 ms	0 m 1.536 s	.86
1000	3.05063 ms	20.7067 ms	0 m 10.629 s	.886
5000	15.173 ms	103.406 ms	0 m 51.339 s	.871

Code can be found in `optimization/baseline.cu`.

1 Optimization: Overlap-Add method for FFT-based convolution

a. Which optimization did you choose to implement and why did you choose that?

I chose to implement a 2D FFT-based overlap-add convolution method. The reason I chose it is that this method has a better theoretical time complexity, i.e. $\mathcal{O}(nm \log k)$ (FFT-overlap-add) v.s. $\mathcal{O}(nmk^2)$ (brute-force) in serial case. And $\mathcal{O}(\log k)$ v.s. $\mathcal{O}(k^2)$ when having sufficiently many threads available.

I have implemented 1-D FFT before for polynomial multiplication, so it wouldn't be so hard for me to implement a parallelized 2-D FFT and the overlap-add method.

b. How does the optimization work? Did you think the optimization would increase the performance of the forward convolution? Why? Does the optimization synergize with any previous optimizations?

The optimization takes advantage of the efficient Fast Fourier Transform, which can complete 1-D convolution with a time complexity of $\mathcal{O}(n \log n)$, and a 2-D convolution in $\mathcal{O}(nm \log(n+m))$. The overlap-add method is involved to further improve the time complexity to $\mathcal{O}(nm \log k)$, where k is the length of the mask. When $k \ll n + m$, the FFT-based overlap-add method will have a great advantage on time complexity over the brute-force method $\mathcal{O}(nmk^2)$.

The 2-D FFT is merely performing 1-D FFT on the grid in the row manner first, then the column manner. For a $n \times m$ grid, we need to perform n row manner FFT $\mathcal{O}(m \log m)$, and m column manner FFT $\mathcal{O}(n \log n)$. And the 1-D FFT is about evaluating the polynomial with $\omega_n^0, \omega_n^1, \cdots, \omega_n^{n-1}$ (the n-th root of unity) by divide-and-conquer method:

$$\mathrm{DFT}(a_0, a_1, \cdots, a_{n-1}) = \mathrm{DFT}(a_0, a_2, \cdots, a_{n-2}) + \left\{(-1)^{[k \geq n/2]} \omega_n^k\right\} \times \mathrm{DFT}(a_1, a_3, \cdots, a_{n-1})$$

and performing an IDFT to the product of the two grids' DFT will give you the convolution result of those two grids.

The overlap-add method divides the whole grid into several (actually there will be $\lceil n/(2k-1) \rceil \times \lceil m/(2k-1) \rceil$ of them) small $(2k-1) \times (2k-1)$ grids. We perform a 2-D convolution on each small grid, which yields a $\mathcal{O}(n/k \times m/k \times k^2 \log k) = \mathcal{O}(nm \log k)$ times complexity. Denote $R_{r,c,x,y}$ as a result of the convolution for the element (x, y) in small grid (r, c). For an arbitrary element (i, j), we can calculate the

result of the convolution by, (let $d = 2k - 1$ and suppose it is in small grid $(r = i/d, c = j/d)$)

$$R_{i,j} = R_{r,c,i+d-1,j+d-1} + R_{r+1,c,i-1,j+d-1} + R_{r,c+1,i+d-1,j-1} + R_{r,c,i-1,j-1}$$

However, I don't think this optimization would increase the performance of the forward convolution. Since there are too many operations in this method. We need to launch 1-D FFT kernel 6 times, grid transpose kernel 4 times, 2 kernels for expanding the grid and mask, 1 for multiplying and adding the DFTs, and 1 for the overlap-add. And in the 1-D FFT kernel, there will be the butterfly transformation, and a bunch of complex number multiplication $\mathcal{O}(2 \log k)$ and addition $\mathcal{O}(4 \log k)$, which are quite time-consuming compares to `float` or even `__half`.

This optimization does not synergize with any other optimization.

c. List the Op Times, whole program execution time, and accuracy for batch sizes 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	10.2103 ms	11.4907 ms	0 m 1.515 s	.86
1000	101.591 ms	113.796 ms	0 m 10.785 s	.886
5000	507.595 ms	568.959 ms	0 m 51.899 s	.871

Code can be found in `optimization/overlap-add-fft.cu`.

d. 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 of).

From the result, it is obvious that the FFT-based overlap-add method is not successful in improving performance, though it might not be the case when the mask length k is much larger (e.g. about 10^3).

The main reason is that FFT needs more complicated kernels, more thread synchronization, and more expensive complex number arithmetic. Also, the block size is relatively fixed for FFT (Block $\langle 16, 16, 1 \rangle$ for convolutional forward network).

e. What references did you use when implementing this technique?

https://en.wikipedia.org/wiki/Fast_Fourier_transform

2 Optimization: FP16 arithmetic

a. Which optimization did you choose to implement and why did you choose that?

I try to place the whole weight matrix ($M \times C \times K \times K$ tensor) into the constant read-only memory, which has a larger read/write bandwidth compared to global memory.

However, I spotted a performance reduction for a large mask size (the second layer). So, I choose to use the normal way (Optimization 2) to deal with a large mask while storing the mask in constant memory when it is small.

This method is promising as constant read-only memory has faster I/O in most cases.

b. How does the optimization work? Did you think the optimization would increase the performance of the forward convolution? Why? Does the optimization synergize with any previous optimizations?

Instead of using `float` in the convolution kernel, I use `__half` for all computations. The input grid and mask are transformed into FP16 on the host before the convolution kernel is launched, and the result is transformed back into FP32 in the kernel.

For the grid and block dimensions, they are changed from Grid $\langle BM, \lceil H_{out}/16 \rceil, \lceil W_{out}/16 \rceil \rangle$, Block $\langle 16, 16, 1 \rangle$ to Grid $\langle H_{out}, B, 1 \rangle$, Block $\langle W_{out}, M, 1 \rangle$. The new thread block will have no control divergence, and no halo cells, which makes the efficiency of each block higher.

The optimization does not synergize with the previous optimization.

c. List the Op Times, whole program execution time, and accuracy for batch sizes 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	0.16737 ms	0.43685 ms	0 m 10.518 s	.86
1000	1.45131 ms	4.06647 ms	0 m 10.845 s	.887
5000	7.15763 ms	20.2066 ms	0 m 52.224 s	.8712

Code can be found as part of `custom/new-forward.cu`.

d. 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 of).

From the result, it is obvious that this optimization has greatly improved the performance, which makes the total Op time reduce from above 100 ms to around 27 ms.

FP16 makes the floating point arithmetic much faster, and the conversion between FP16 and FP32 takes almost no time as they are completed on the host (CPU).

The new grid and block dimension enable more efficient thread use, more coalesced memory access, no control divergence, and, no halo cells. All of them contribute to the boost in performance.

e. What references did you use when implementing this technique?

https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH____HALF__MISC.html

3 Optimization: Weight matrix in constant memory for small mask

a. Which optimization did you choose to implement and why did you choose that?

I chose to use 16-bit floating point number data type `__half` instead of 32-bit `float` and 64-bit `double`. smaller data types have faster floating-point multiplication and addition speed, and smaller size which will make it hit the register/cache with a higher chance and also result in faster data transfer speed.

I also changed the grid and block shape so that there will be more coalesced memory access, and will also take full advantage of the device. This will definitely speed up the computation.

b. How does the optimization work? Did you think the optimization would increase the performance of the forward convolution? Why? Does the optimization synergize with any previous optimizations?

The mask is defined in the global scope with `__constant__` when putting the weight matrix in constant read-only memory, and this happens only when $M \times C \times K \times K \leq 1024$.

When the mask size is too large, a performance reduction happens for some reason. In the experiment, the total Op time becomes larger than 50 ms (6 ms + 45 ms). So, I adopt the constant memory method when the mask size is small, while the normal method when it is large.

I think it would increase the performance as putting the mask into constant memory works fine when it is small, as the result shows.

The optimization synergizes with the last optimization (Optimization 2: FP 16 arithmetic).

c. List the Op Times, whole program execution time, and accuracy for batch sizes 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	0.15128 ms	0.42657 ms	0 m 1.673 s	.86
1000	1.32708 ms	4.0433 ms	0 m 10.749 s	.887
5000	6.54637 ms	20.086 ms	0 m 51.307 s	.8712

Code can be found in `custom/new-forward.cu`.

d. 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 of).

From the result, a performance improvement can be found, especially for the first layer (Op Time 1). It seems that the weight matrix in constant memory outperforms its counterpart when the mask size is small, and yields around 26.6 total Op Time.

e. What references did you use when implementing this technique?

None.