HW3 Report

Name: Haixu Song CWID:10446032

Objective

Implement a convolution routine using texture and shared memory in CUDA. Your code should be able handle arbitrary 2D input sizes (1D kernel and 2D image pixel values should be randomly generated between 0~15). You only need to use a 1D kernel (mask) for both row and column convolutions that can be performed separately.

How to run this code

This part is the same with HW1. There's no verbose explaining here.

Implementation Details

This part is the same with HW2. There's no verbose explaining here.

Kernel Part:

Firstly, I have to make sure the largest constant memory of my device so that when input mask width is too large, just stop running. Then I used the algorithm called "a simpler tiled convolution algorithm" introduced in Ch7.5. Each tile just load the elements inside the tile and visit the skirt elements directly to global memory. This process may be accelerated by hardware's L2 cache since these elements are once loaded by other threads.

Result Analyzing

Parameters Bounds and Hardware Resources

My device's Total constant memory is 65536 bytes, which means at most 16384 float numbers. 16384 = 128 * 128, which means for a 2D mask, its width is at most 128. This is defined as macro in my code.

Other parameters are far from bounds and the analyzing process are similar with previous reports. No verbose here.

Performance

This is the result when data length is small.

```
...\yourDirectory>conv -i 3 2 1
Total constant memory: 65536
x: 3
y: 2
k: 1
data:
[   5,   2,   5,  13,   2,  11]
data:
[1.000000]
Done initializing data array.
data:
[5.000000,2.000000,5.000000,13.000000,2.000000,11.000000]
Done convolution with CPU in 0 miliseconds.
Done allocating space in device.
Done copying memory from host to device.
Done initializing block dimention and grid dimention.
Done matrix multiplication with GPU in 0 miliseconds.
conv_GPU:data:
[5.000000,2.000000,5.000000,13.000000,2.000000,11.000000]

Result check: ---PASS---.
```

We can easily see that GPU is similar with CPU since there's no much calculation needed while GPU did a lot of data transferring and there's significant threads divergence happening. Let's see what's the result will be like when the data length is a medium number.

```
...\yourDirectory>conv -i 300 250 5
Total constant memory: 65536
x: 300
y: 250
k: 5
data:
[  14,   9,   5,  11,   9,   1,   2,   4,  14,  11...]
data:
[0.015274,0.040431,0.035939,0.088050,0.079964,0.002695,0.047619,0.062893,0.044924,0.01
1680...]
Done initializing data array.
data:
[2.100629,3.197664,3.820305,3.891284,4.174304,4.226415,4.806829,4.400719,3.495058,5.77
7179...]
Done convolution with CPU in 12 miliseconds.
Done allocating space in device.
Done copying memory from host to device.
Done initializing block dimention and grid dimention.
Done matrix multiplication with GPU in 0 miliseconds.
conv_GPU:data:
[2.100629,3.197664,3.820305,3.891285,4.174304,4.226415,4.806829,4.400719,3.495058,5.77
7178...]

Result check: ---PASS---.
```

We see that GPU stills getting few milliseconds, but CPU times for running sequential code is rising. In this case, it's a very tiny picture. Let's see the performance when the size of data is about to be the biggest picture of today's computer.

```
...\yourDirectory>conv -i 3920 2160 99
Total constant memory: 65536
x: 3920
y: 2160
```

```
k: 99
data:
[  13,   1,  14,   5,   7,   1,   4,   4,  15,  10...]
data:
[0.000004,0.000165,0.000116,0.000200,0.000120,0.000064,0.000087,0.000002,0.000163,0.00
0130...]
Done initializing data array.
data:
[1.882835,1.899540,1.950500,1.992527,2.019738,2.053959,2.107800,2.152441,2.188421,2.24
0226...]
Done convolution with CPU in 546116 miliseconds.
Done allocating space in device.
Done copying memory from host to device.
Done initializing block dimention and grid dimention.
Done matrix multiplication with GPU in 497 miliseconds.
conv_GPU:data:
[1.882835,1.899540,1.950499,1.992527,2.019738,2.053959,2.107799,2.152441,2.188421,2.24
0226...]

Result check: ---PASS---.
```

We see that sequential code takes about 1000 times the time GPU needed. The advantage of parallel algorithm is huge when the data size is huge. In this case, total integer operations are x * y * (2 * k * k). So GFLOPS is about 3920 * 2160 * 99 * 99 * 2 / 0.497 = 333,951,819,718.31 = 334GFLOPS.

Answering Questions

(1) Name 3 applications of convolution.

> It is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision. In high-performance computing, the convolution pattern is often referred to as stencil computation, which appears widely in numerical methods for solving dif-ferential equations. It also forms the basis of many force calculation algorithms in simulation models.
>
> ------- Quote from out text book Ch7

(2) How many floating-point operations are being performed in your convolution kernel (expressed in dimX, dimY and dimK)? Explain.

> Each output thread calculates k * k multiplications and k * k - 1 additions. There's x * y threads like this. So the total operations is x * y * (2k^2 - 1).

(3) How many global memory reads are being performed by your kernel (expressed in dimX, dimY and dimK)? Explain.

> In my kernel, there's x * y global memory reads. That's because the halo elements are loaded in L2 caches according to what the textbook Ch7.5.

(4) How many global memory writes are being performed by your kernel (expressed in dimX, dimY and dimK)? Explain.

> Only x * y writes. Each thread writes one. There's x * y threads, so ....

(5) What is the minimum, maximum, and average number of real operations that a thread will perform (expressed in dimX, dimY and dimK)? Real operations are those that directly contribute to the final output value.

> max: k * k * 2 - 1 (k * k multiplications and k * k - 1 additions)
>
> min: (k/2 + 1)^2 * 2 - 1 (comparing to max, there's only k/2+1 on each width)

(6) What is the measured floating-point computation rate for the CPU and GPU kernels in this application? How do they each scale with the size of the input?

> CPU: It's 1:1, each 2 data got from memory, 2 operations done (1 multi 1 addition).
>
> GPU: TILE_SIZE^2 * 2 * k^2 / TILE_SIZE^2 = 2k^2. We can see that with a greater mask size, the higher this rate is.

(7) What will happens if the mask (convolution kernel) size is too large (e.g. 1024)? What do you end up spending most of your time doing? Does that put other constraints on the way you'd write your algorithm (think of the shared/constant memory size)?

> Firstly, when k = 1024, the mask size is too big for constant memory. So each time we will load mask into shared memory in order to have a faster speed.
>
> It have to change my code for this, because k >=128 can't pass the UI in my code, since my constant memory is just 65536 bytes (analyzed before).
>
> I think most of the time will be spent in loading global memory into shared memory.