HW3 Report

Name: Haixu Song CWID:10446032

Objective

Implement a kernel to perform an inclusive parallel scan on a 1D list (with randomly generated integers). The scan operator will be the addition (plus) operator. You should implement the work efficient kernel in our lecture. Your kernel should be able to handle input lists of arbitrary length. To simplify the lab, the student can assume that the input list will be at most of length 2048*65535 elements. This means that the computation can be performed using only one kernel launch. The boundary condition can be handled by filling “identity value (0 for sum)” into the shared memory of the last block when the length is not a multiple of the thread block size.

How to run this code

Open src directory as a project in Visual Studio. Change project property > debugging > Command Arguments with `-i 50000` such things. Then `Ctrl` + `F5` to run the code without debugging to see the result.

Or you can also compile the source code using nvcc the run with command. Don't forget the input parameters. You can use -h for help.

Implementation Details

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

Kernel Part:

I used Brent-Kung Algorithm as the scan algorithm in each kernel. And also used hierarchical approach to process arbitrary sized input. Then I used streaming to process multiple steps in one kernel.

Result Analyzing

Parameters Bounds and Hardware Resources

My device shared memory each block is 49152 bytes, which can maximum holds 12288 unsigned int numbers. My BLOCK_SIZE is 1024(which is the max threads of each block), which needs 2048 unsigned integers stored in shared memory. In bound CHECKED.

The max grid dimension is 2048*65535, which means there's maximum 65535 blocks in a grid. My device can hold maximum of 2147483647. In bound CHECKED.

2048 * 65536 unsigned int needs 2048 * 65536 * 4 bytes = 536870912 bytes = 512M memory (both host and device). In bound CHECKED.

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.

```
Bin numbers: 5

[   3,   7,   8,   2,   4]
Done initializing data array with length 5.

cumSum_CPU:
[   3,  10,  18,  20,  24]
Done histogrm calculation 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 cumSum with GPU in 0 miliseconds.
hist_GPU:
[   3,  10,  18,  20,  24]

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
Bin numbers: 500000

[   5,   7,   9,   1,   2,   4,   5,   3,   8,   9...]
Done initializing data array with length 500000.

cumSum_CPU:
[   5,  12,  21,  22,  24,  28,  33,  36,  44,  53...]
Done histogrm calculation with CPU in 1 miliseconds.

Done allocating space in device.
Done copying memory from host to device.
Done initializing block dimention and grid dimention.
Done cumSum with GPU in 2 miliseconds.
hist_GPU:
[   5,  12,  21,  22,  24,  28,  33,  36,  44,  53...]

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. Let's see the performance when the size of data is about to be the boundary.

```
...\yourDirectory>conv -i 3920 2160 99
Bin numbers: 100000000

[   7,   7,   5,   1,   1,   9,   6,   1,   0,   1...]
```

```
Done initializing data array with length 100000000.

cumSum_CPU:
[   7,  14,  19,  20,  21,  30,  36,  37,  37,  38...]
Done histogrm calculation with CPU in 196 miliseconds.

Done allocating space in device.
Done copying memory from host to device.
Done initializing block dimention and grid dimention.
Done cumSum with GPU in 284 miliseconds.
hist_GPU:
[   7,  14,  19,  20,  21,  30,  36,  37,  37,  38...]

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

We see that still GPU takes more time the CPU sequential code does. I think it's because my kernel didn't do iteration hierarchical approach. Which means when the size of input is going large, there's still 3 hierarchies and in the second step, those threads streams goes like sequential.

Answering Questions

(1) Name 3 applications of parallel scan.

> Stream Compaction
>
> Summed-Area Tables
>
> Radix Sort
>
> Referred from NVIDA Developer

(2) How many floating operations are being performed in your reduction kernel? Explain.

> There are 2047 in reduction in each block, 2047 - 10 = 2037 in distribution in each block. 1 addition in each block, 2048 addition in step 3 for each block. There's INPUT_SIZE / BLOCK_SIZE blocks, so there's `6133 * INPUT_SIZE / (2 * BLOCK_SIZE)` calculations in kernel.

(3) How many global memory reads are being performed by your kernel? Explain.

> Read each input into shared memory: INPUT_SIZE times.
>
> Read partial sum into each first thread in block: INPUT_SIZE / BLOCK_SIZE times.(this part is actually read 2 times, however the second time should be automatically processed by L2 cache.)
>
> Read DCounter as block index INPUT_SIZE / BLOCK_SIZE times.
>
> Total: INPUT_SIZE + 2 * INPUT_SIZE / BLOCK_SIZE times.

(4) How many global memory writes are being performed by your kernel? Explain.

> Write each final result into global mem INPUT_SIZE times.
>
> Write each partial result into global mem INPUT_SIZE / BLOCK_SIZE times.
>
> Atomic write each DCounter into global mem. INPUT_SIZE / BLOCK_SIZE times.
>
> Total: INPUT_SIZE + 2 * INPUT_SIZE / BLOCK_SIZE times.

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

> Maximum, Log2(INPUT_SIZE)
>
> Minimum, 1
>
> Average, 2

(6) How many times does a single thread block synchronize to reduce its portion of the array to a single value?

> This should be logBLOCK_SIZE(INPUT_SIZE), however in my kernel, it's 1.

(7) Describe what optimizations were performed to your kernel to achieve a performance speedup.

> I used memory accessing coalesce for threads to get values from global memory. I made continuously do the same operations so that reduce the warp threads divergence.

(8) Describe what further optimizations can be implemented to your kernel and what would be the expected performance behavior?

> I should use iterations to do my hierarchical approach. So that when the input size rises, the performance will be far more better.

(9) Suppose the input is greater than 2048*65535, what modifications are needed to your kernel?

> We should actually use several kernels to calculate each partial result. The just follow the hierarchical approach.

(10)Suppose you want to scan using a binary operator that’s not commutative, can you use a parallel scan for that?

> I think it depends. If this operation can transform into commutative ones, then that's OK. Like if we change + in this algo into -, we can't get the result directly, but we can get it easily by just reversing each element. But if the binary operation is x^2+y, then the algo will be really hard. This algorithm doesn't work anymore.