Lab 8: 2D Exclusive Scan

CMPUT 398

1. Objective

Implement an **exclusive** scan program on a 2D arrays.

The scan operator will be the addition (plus) operator, so in other words you are performing a Summed-Area Table.

You should try to implement the work efficient kernel (Blelloch) shown in the lectures, but it is fine if you implement the naive scan (Hillis and Steele). You will get 80% for correct outputs and 20% for having good optimization.

One thing to be aware of is that most implementations only handle arrays the same size as a block; however, your kernel should be able to handle input arrays of arbitrary dimension. More instruction about this below.

This lab will be submitted as one zipped file through eclass. Details for submission are at the end of the lab.

2. Instructions

First, you need to implement a 1D scan kernel and from this 1D case you will be able to extend to 2D case.

2.1. Handling 1D list with arbitrary length:

First, make sure you carefully read the instructions after reading through the following NVIDIA article:

http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Especially examples 39-1 and 39-2.
These examples show you instructions on how to implement a good 1D prefix sum kernel that can handle list with length exactly equal to the block size or block size * 2 depend on the implementation but it's a good place to start. You can do a little bit of extension to the kernel mentioned in the article to make it work on arrays with length less than block size as well as greater than block size.

To make the kernel work on arrays with length less than block size, you just need to add two things to your kernel:

- When loading the input into shared memory, check if the index is out of bound (more than the length of the array – 1). If it is out of bound, simply load a 0 into shared memory. This is like padding the 1D list with 0 at the end.
- When you write the output, also check if the index is out of bound. If it is, do not write anything

After this step, you have a kernel that can handle lists with length less than or equal to block size (or block size * 2)

<u>To make the kernel work on arrays with length greater than block size:</u>
Let's assume we are running an **inclusive** scan with blocks of size 4 and we have the following input array of length 12. We are doing **exclusive** scan but the logic is the very same except we are shifting our output by 1 to the right.

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9	X_{10}	X_{11}

For an inclusive scan, we want the following:

X_0	$\sum(X_0..X_1)$	$\sum(X_0..X_2)$	$\sum(X_0..X_3)$	...	$\sum(X_0..X_9)$	$\sum(X_0..X_{10})$	$\sum(X_0..X_{11})$

Let say we **fix our block size to a constant** and divide the bigger list into smaller lists of equal size that are handled by each block. Each block will do a scan on each smaller list (of length block size * 2 or block size in this example).
After we do scan on smaller list, we will have:
Block 0:

X_0	$\sum(X_0..X_1)$	$\sum(X_0..X_2)$	$\sum(X_0..X_3)$

Block 1:

X_4	$\sum(X_4..X_5)$	$\sum(X_4..X_6)$	$\sum(X_4..X_7)$

Block 2:

X_8	$\sum(X_8..X_9)$	$\sum(X_8..X_{10})$	$\sum(X_8..X_{11})$

Now, if we add **0** to all the output of block 0, $\sum(X_0..X_3)$ to all the output of block 1 and $\sum(X_0..X_3) + \sum(X_4..X_7)$ to the output of block two we will have
Block 1:

X_4	$\sum(X_0..X_5)$	$\sum(X_0..X_6)$	$\sum(X_0..X_7)$

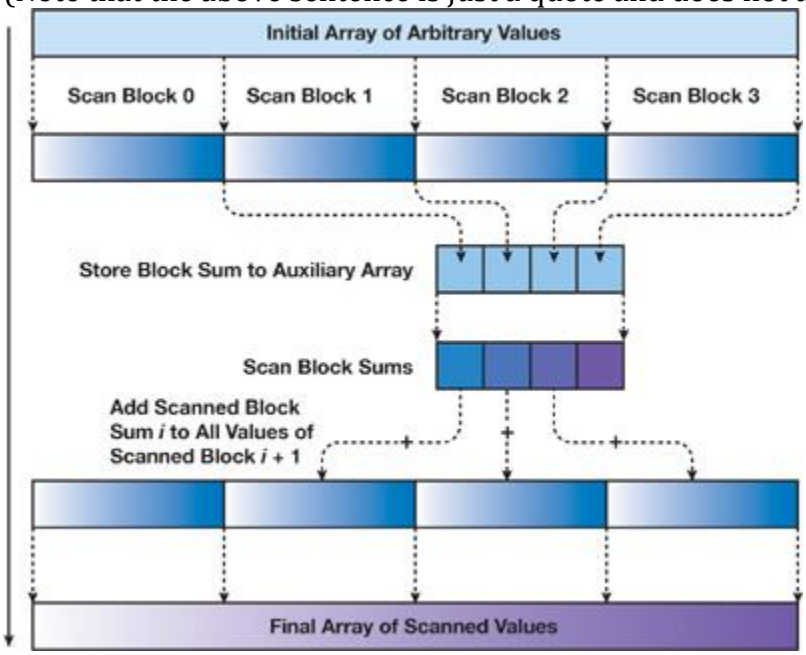
Block 2:

X_8	$\sum(X_0..X_9)$	$\sum(X_0..X_{10})$	$\sum(X_0..X_{11})$

This is exactly what we wanted.
So the trick is to have an additional array that can somehow contains the sum of elements from the beginning of the list to the elements that are at index of multiples of the block size. How do we get this? From the above example, let say we have $\sum(X_0..X_3)$ and $\sum(X_4..X_7)$ and

we want **0, $\sum(X_0..X_3)$** and **$\sum(X_0..X_3) + \sum(X_4..X_7)$**. This is exactly equal to doing a scan on a list with two elements **$\sum(X_0..X_3)$** and **$\sum(X_4..X_7)$.**

This bring us to the idea in section 39.2.4 Arrays of Arbitrary Size of the link http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html :
"Before zeroing the last element of block i (the block of code labeled B in Listing 39-2), we store the value (the total sum of block i) to an auxiliary array SUMS. We then scan SUMS in the same manner, writing the result to an array INCR. We then add INCR[i] to all elements of block i using a simple uniform add kernel invoked on N/B thread blocks of B/2 threads each."
(Note that the above sentence is just a quote and does not apply to this example)



Come back to the example, we allocate an array which is the same length as the number of blocks (in this case 3). Then each block stores the max value in the aux array. If the entire array fits into a single block, then we don't need the aux array and can pass NULL as the argument. To use the aux array, we add the following line at the end of the kernel code.

```
1.  if (aux && threadIdx.x == 0)
2.      aux[blockIdx.x] = temp[BLOCK_SIZE - 1]; // where temp is the shared memory array
```

Note that BLOCK_SIZE -1 is supposed to point to the last element in the block or the max sum, but this might be different for your implementation. For example in the second block aux[blockIdx.x] will be **$\sum(X_4..X_7)$**. With the example above the aux array should be:

$\sum(X_0..X_3)$	$\sum(X_4..X_7)$	$\sum(X_8..X_{11})$

Now we can perform scan on the aux array. In this case the array fits into one block and we do not need to pass anything to the aux argument. By performing scan on the aux array, we get:

$\sum(X_0..X_3)$	$\sum(X_0..X_7)$	$\sum(X_0..X_{11})$

Now we can perform uniform sum on the output array from the first scan:

Block 0:

X_0	$\sum(X_0..X_1)$	$\sum(X_0..X_2)$	$\sum(X_0..X_3)$

Block 1 + aux[0]:

$\sum(X_0..X_4)$	$\sum(X_0..X_5)$	$\sum(X_0..X_6)$	$\sum(X_0..X_7)$

Block 2 + aux[1]:

$\sum(X_0..X_8)$	$\sum(X_0..X_9)$	$\sum(X_0..X_{10})$	$\sum(X_0..X_{11})$

We need to change the signature of our scan function from (this is in the cuda scan page above):

```
__global__ void scan(float *g_odata, float *g_idata, int n)
```

The arguments are as follows:

- g_odata – output array which is the same size as the input array
- g_idata – input array
- n – block size

To:

```
__global__ void scan(float *g_odata, float *g_idata, float *g_aux, int len)
```

Where:

- g_odata – output array which is the same size as the input array
- g_idata – input array
- g_aux – is an array the size of the number of blocks we will see why later
- len – is the size of the array

We removed n and additionally predefine the block size:

```
#define BLOCK_SIZE 512
```

The one issue we must still solve is when the length of aux array is larger than block size (i.e. when the input length is greater than block size * block size or (block size * 2) * (block size * 2)). In that case, we need another aux array to add the sums from the previous blocks. A simple solution is to write a recursive function on the **CPU** as a wrapper of the actual scan kernel.

```
void recursive_scan(float *g_odata, float *g_idata, int len)
```

The pseudo code will look something like this:

1. FUNCTION recursive_scan(output, input, len)
2. Calculate the number of blocks needed
3. IF only one block is needed perform scan by calling the kernel and exit function
4. ELSE
5. Allocate memory for aux array (contains the sum of elements of each block)
6. Allocate memory for scanned aux array (contain the prefix sum of aux)
7. Perform scan passing in the arguments from the function and the aux array
8. Call recursive_scan (scanned_aux, aux, num_of_blocks)
9. Perform uniform addition on output with the scanned aux array
10. Free aux array and scanned aux array
11. ENDIF

For the pseudo code above, red lines are CUDA kernel calls. Line 7 being the actual scan kernel for 1D list.

<u>The exclusive scan is like the inclusive scan; however, everything is shifted to the right by one.</u> For example, using the example array above, we will get the following:

Block 0:

0	**X_0**	**$\sum(X_0..X_1)$**	**$\sum(X_0..X_2)$**

Block 1:

0	**X_4**	**$\sum(X_4..X_5)$**	**$\sum(X_4..X_6)$**

Block 2:

0	**X_8**	**$\sum(X_8..X_9)$**	**$\sum(X_8..X_{10})$**

Now we need to add $\sum(X_0..X_2) + X_3$ to Block 1 and $\sum(X_0..X_2) + X_3 + \sum(X_4..X_6) + X_7$ to Block 2. There is a simple trick for the exclusive algorithm if you implement the work efficient kernel (Blelloch). After the Up-Sweep or Reduce phase we have the value we need.

Block 0:

X_0	**$\sum(X_0..X_1)$**	**X_2**	**$\sum(X_0..X_3)$**

Block 1:

X_4	**$\sum(X_4..X_5)$**	**X_6**	**$\sum(X_4..X_7)$**

Block 2:

X_8	**$\sum(X_8..X_9)$**	**X_{10}**	**$\sum(X_8..X_{11})$**

```
1.  // Up-Sweep
2.  …
3.
4.  if (threadIdx.x == 0) {
5.      if (aux)
6.          aux[blockIdx.x] = temp[BLOCK_SIZE - 1]; // save the last element
7.      temp[BLOCK_SIZE - 1] = 0; // clear the last element
8.  }
9.
```

```
10. // Down-Sweep
11.
12. …
```

2.2. Extend 1D to 2D scan:

Extending 1D scan to 2D scan is relatively easy. First, you do a 1D scan on each row of the matrix. Then do another scan on the **transpose** of what you get from the previous step and then transpose back to get the final result. (See section 39.3.2 Summed-Area Tables of the CUDA scan page)

For this lab, you need to use a transpose kernel. A good place to start is in this link: https://devblogs.nvidia.com/parallelforall/efficient-matrix-transpose-cuda-cc/
But any other code may also be fine.

The pseudo code for all the steps we mentioned:

1. for each row j in deviceInput array:
2. recursive_scan(deviceTmpOutput[j,:] , deviceInput[j,:], numInputColumns)
3. deviceOutput = transpose(deviceTmpOutput)
4. for each row j in deviceOutput array:
5. recursive_scan(deviceTmpOutput[j,:] , deviceOutput[j,:], numInputColumns)
6. deviceOutput = transpose(deviceTmpOutput)

Note that in the above pseudo code we are using another array called deviceTmpOutput to store the temporary output (there a variable with same name in the code) and the assignment deviceOutput = transpose(deviceTmpOutput) may not be what the code actually really looks like (the transpose should be a kernel with input, output array in its argument list and should not return anything). The notation X[j,:] denotes the row j of the 2D array X (actually notation in your code should be different).

Important: after launching a kernel in the pseudo code make sure you call:

```
wbCheck(cudaDeviceSynchronize());
```

Or

```
cudaDeviceSynchronize();
```

3. Local Setup Instructions

Steps:

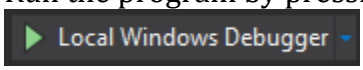
1. Download “Lab8.zip”.
2. Unzip the file.
3. Open the Visual Studios Solution in Visual Studios 2013.
4. Build the project. Note the project has two configurations.

a. Debug

b. Submission

But make sure you have the “Submission” configuration selected when you finally submit.

5. Run the program by pressing the following button:



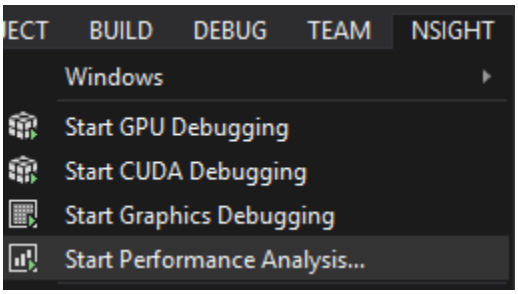
Make sure the “Debug” configuration is selected.

4. Testing

To run all tests located in “Dataset/ Test”, first build the project with the “Submission” configuration selected. Make sure you see the “Submission” folder and the folder contains the executables.

To run the tests, click on “Testing_Script.bat”. This will take a couple of seconds to run and the terminal should close when finished. The output is saved in “Marks.js”, but to view the calculated grade open “Grade.html” in a browser. If you make changes and rerun the tests, then make sure you reload “Grade.html”. You can double check with the timestamp at the top of the page.

You can test your performance using NSight. First build with the “Test” configuration selected. Then select NSIGHT -> Start Performance Analysis…



In “Application Settings” make sure the “Application:” is set to the executable you wish to run and the “Arguments:” should be:

-e output.raw -i input.raw -o myOutput.raw -t vector

The “Working Directory:” should be the path to the test. For example:

PathToLab8\Dataset\Test\11

Remember to replace PathToLab8 with the actual path to lab 8.

To make it easier for you to test the 1D scan kernel (the first and important step of this lab), you will be given another Visual Studio solution that contains the skeleton code that you can write your **exclusive** scan kernel in and test it. There will be two project in the solution, 1 for inclusive and 1 for exclusive scan, but you just need to write the exclusive

scan. There will be test script provided for this solution too. Moving the scan kernel from this 1D solution to the 2D solution should be a matter of copy paste.

Also, there will a reference file provided for you to compare the speed of your kernel.

5. Submission

After you build your project with the "Submission" configuration selected (see above) you will see a new folder called "Submission" in the project directory. If you open the folder you will see your executable. Finally zip the folder "Submission" and upload the zipped folder to eclass.

If you don't see this folder or are missing the executable, then one of two things happened. First your build could have failed and you should check for any errors in your build log in Visual Studios. Also you might have had the "Debug" configuration selected (see above). Make sure the "Submission" configuration is selected.

6. Mark Breakdown

It is important that you **do not** add or remove any print statements or wb functions. We use them for part of your grading so if you make changes the marking script could fail.

If you get all the test correct, you get 80%. If your kernel is optimized well (i.e. its speed is at most 2.5 times slower than the reference file on **test 11**), you get the other 20%

Part	Breakdown (%)
Exclusive Scan	80 %
Optimization	20 %