Department of Electrical and Computer Engineering
Faculty of Engineering and Architecture
American University of Beirut

EECE 696: Applied Parallel Programming

Spring 2018

Lab 8

List Scan

1. Inclusive Scan

a. Objective

The lab's objective is to implement a kernel to perform an inclusive parallel scan on a 1D list. The scan operator will be the addition (plus) operator. You should implement the work inefficient kernel in Lecture 10.2 as well as the work efficient kernel in lecture 10.3. Your kernel should be able to handle input lists of arbitrary length. To simplify the lab, you can assume that the input list will be at most of length 1024x1024 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.

b. Handling Arrays of Arbitrary Lengths

For a number of applications, a scan operation can process elements in the millions or even billions. The kernels presented in the lectures assume that the entire input can be loaded in the shared memory. Obviously, we cannot expect all input elements of these large scan applications to fit into the shared memory, which is why we say that these kernels process a section of the input. Fortunately, a hierarchical approach can extend the scan kernels that we have generated so far to handle inputs of arbitrary size. The approach is illustrated in Fig. 8.1.

For a large data set, we first partition the input into sections so that each of them can fit into the shared memory and be processed by a single block. Assume that we launch the scan kernel on a large input data set. At the end of the grid execution, the Y array will contain the scan results for individual sections, called *scan blocks*, as sown in Fig. 8.1. Each result value in a scan block only contains the accumulated values of all preceding elements within the same scan block. These scan blocks need to be combined into the final result; i.e., we need to write and launch another

kernel that adds the sum of all elements in preceding scan blocks to each element of a scan block.

To implement the hierarchical scan, we need to add an Auxiliary array S as input to the scan kernel, which has the dimension of InputSize/SECTION_SIZE. At the end of the kernel, we add a conditional statement. The last thread in the block writes the output value of the last XY element in the scan block to the blockIdx.x position of S.

Then we need to launch the scan kernel on the vector S to produce the Scanned Auxiliary Array. This kernel is simply any of the parallel scan kernels, which takes S as input and writes S as output.

Finally, we need to launch a kernel that takes the initial output Y and fixes it using the Scanned Auxiliary Array. This kernel takes the S and Y arrays as inputs and writes its output back into Y.

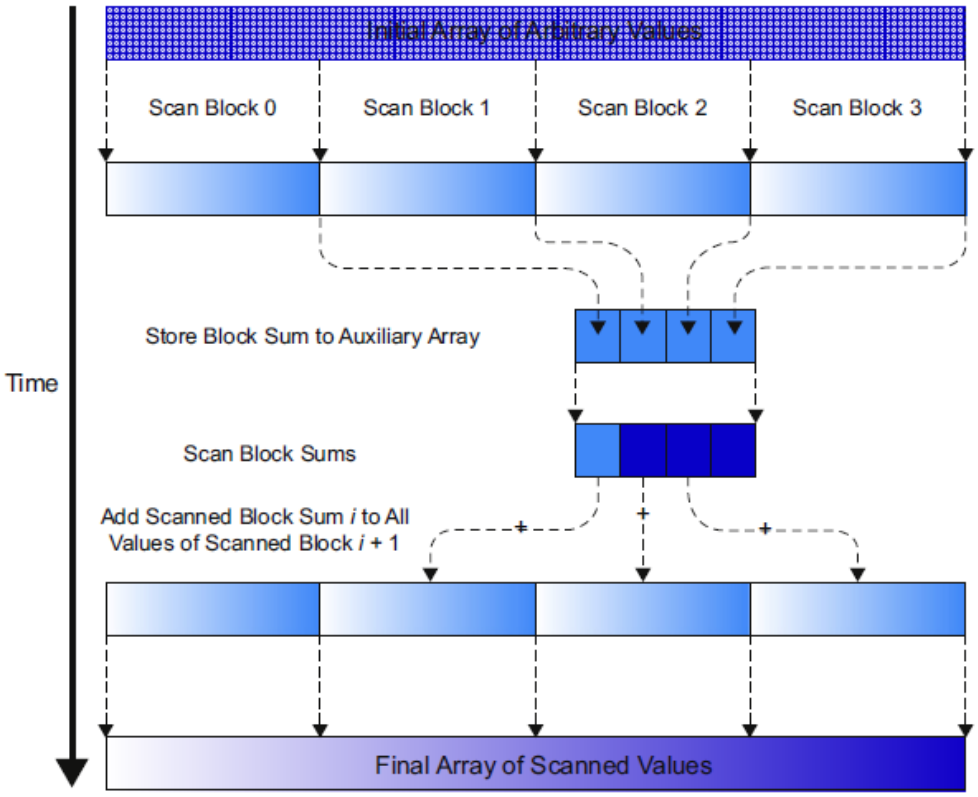

Fig. 8.1 A hierarchical scan for arbitrary length inputs.

c. Instructions

In this exercise, you have to write the CUDA program to implement an inclusive scan kernel that uses shared memory. A CUDA code template, **Scan_Template.cu**, is provided to you. Timers have been inserted in the code to record the time needed to perform the different steps in the

program, as well as to compare the performance of GPU parallel implementation of the algorithm with its serial execution on the CPU.

- Create a visual studio CUDA 8.0 project and add the **Scan_Template.cu** file downloaded from Moodle to your project.
- Add the file **GpuTimer.h** downloaded from Moodle to your project.
- Edit the **Scan_Template.cu** code template to perform the following:

- Allocate device memory
- Copy host memory to device
- Initialize thread block and kernel grid dimensions
- Invoke CUDA kernels
- Write the CUDA kernel that implements **the work inefficient kernel based on lecture 10.2**. Use shared memory to reduce the number of global accesses, handle the boundary conditions when loading input list elements into the shared memory.
- Write a CUDA kernel **fixup** that fixes the generated array from first call of the scan kernel. This kernel allows your program to handle arrays of arbitrary length that don't fit into the shared memory of a single block.
- Copy final output array from device to host
- Write the CPU serial function that performs the same inclusive scan operation
- Free device memory
- Instructions about where to place each part of the code is demarcated by the //@@ comment lines.

Save your code with the name **Scan_WorkInefficient.cu** and submit it to Moodle.

d. Questions

(1) Report the following times: **for input length of 1024*1024**

a. Time needed to perform Memory allocation on the GPU
b. Time needed to perform Memory copy from the host to the device.
c. Execution time of parallel scan on the GPU
d. Time needed to copy the result to the CPU
a. Execution time of the serial implementation of the algorithm on the CPU (i.e. execution time of the CPU function for inclusive scan).

Important Remark: Make sure that you run your code in Release Mode and not Debug Mode to get accurate Timing information.

(2) What is the speedup of the GPU implementation? Note that Speedup in this case is given by:

$$Speedup = \frac{Total\ Time\ needed\ to\ execute\ the\ serial\ implemenation\ on\ the\ CPU}{Total\ Time\ needed\ to\ perform\ parallel\ scan\ \ on\ the\ GPU}$$

(3) Name three applications of the scan operation
(4) For the scan kernel, how many floating point operations are being performed?
(5) For the scan kernel, how many global memory reads are being performed by your kernel? EXPLAIN.
(6) For the scan kernel, how many global memory writes are being performed by your kernel? EXPLAIN
(7) 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 value.
(8) How many times does a single thread block synchronize to reduce its portion of the array to a single value?
(9) Write the improved work-efficient kernel **based on lecture 10.3** that performs better mapping of threads to data elements to allow for less control divergence and better resource efficiency. Save your code with the name **ImprovedScan.cu** and submit it to Moodle.
(10) Repeat questions 1 and 2 for your improved kernel. What do you notice?