EE 382C/361C: Multicore Computing **Fall 2016**

Lecture 15: October 18

Lecturer: Vijay Garg *Scribe: Avin Tung*

15.1 Introduction

This lecture will complete the Cuda examples and begin covering algorithms for parallel programming using GPU:

1. reduce.cu
2. Parallelism Vocabulary
3. Blelloch Scan
4. Brick Sort

15.2 reduce.cu

Check out the Cuda code for reduce.cu [1].

It uses multiple blocks, each with multiple threads, of the GPU to compute the sum of the array by reduction. It splits the array into multiple section for each block to sum using its multiple threads. When each block is complete with its portion, a single block will sum the final sums and return.

Each block runs the following code. The code splits the array in half, then uses n/2 threads to sum entries n+i and n+i+(n/2) for i= 0 to n/2 into a new array of size n/2. Before the next iteration, *syncthreads* function is called to make sure all adds are done before moving on. This is repeated until there is only one entry left which is the sum of the total array. The new value is then written back to global with a single thread.

The key things to note is the program runs a global memory kernel in data and a shared memory kernel in sdata. This example is to show the difference in time between threads running using global memory and threads running on their own shared memory.

15.3 Parallelism Vocabulary

There are specific terms that are often used when talking about writing programs for GPUs.

15.3.1 Map

Map maps every entry to another entry, one to one, and returns it. So for every i, we return it's corresponding f(i). In terms of parallelism, it can be done in O(1) time depending on the complexity of the mapping function.

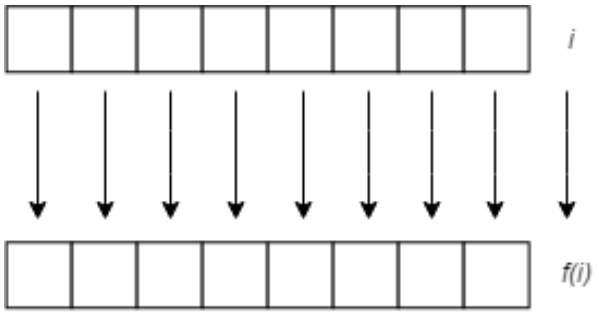

Figure 15.1: Mapping

15.3.2 Reduce

Reduce is taking an associative operator applied to multiple entries and reducing it to one- many to one. The important thing to note is to understand the identity of the operation. For example, the values for sum should be initialized to 0, the values for max should be initialized to -, the values for min should be initialized to , and so on.

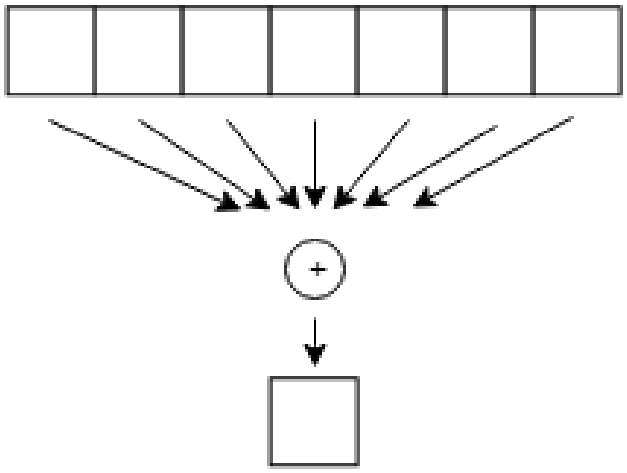

Figure 15.2: Reduce with Add Operation

15.3.3 Scan

Scan takes an array and creates a new array such that the new array's element j is the sum of all the elements before j and including/excluding j. If we include j, it is classified as an inclusive scan, and if j is excluded, then it is an exclusive scan. This is also known as the prefix sum.

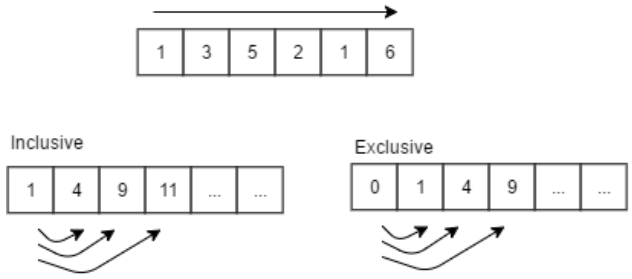

Figure 15.3: Scan

15.3.4 Scatter

Scatter takes an array scatters the entries into a new array. An example of scatter can be merging two sorted arrays into one sorted array using binary search to calculate the new address into the new array.

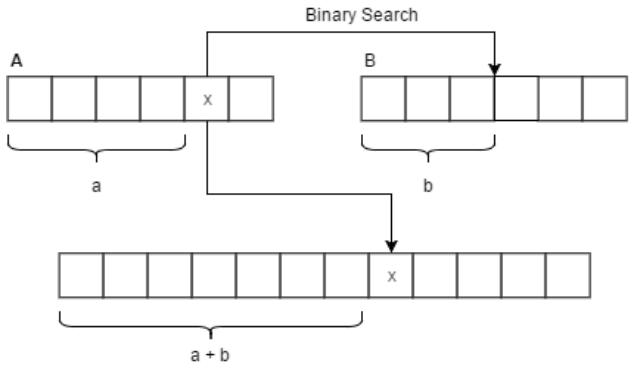

Figure 15.4: Scatter Example: Merging Two Sorted Arrays

15.3.5 Gather

Gather takes multiple entries and does a sequence of operations to them to reduce it to one entry- many to one. The key thing about gather is that some synchronization and atomicity may be used depending on the sequence of operations. For example, we can take 3 entries, and average them into one entry in a new array. This requires an atomic add for each of the 3 entries as well as a barrier before dividing it by 3.

15.3.6 Transpose

Transpose is moving memory from one piece position in memory to another, or liken to transposing a matrix. This is useful in parallel programming since it moving entries into memory that is faster to access and operate on, such as into local cache, can significantly improve the performance time.

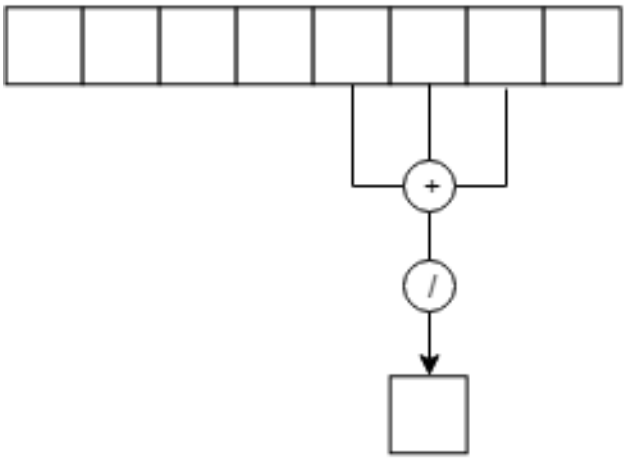

Figure 15.5: Gather Example: Average Example

15.3.7 Stencil

Stencil is like the 3D version of gather. If on a matrix of entries, stencil can take multiple entries surrounding the desired entry, operate on them, and store the value into a new grid of entries.

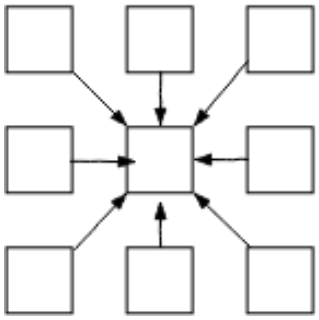

Figure 15.6: Stencil

15.3.8 Compact

Compact is looking at an array, operate on it, and return a compact version of the original array. For example, an array can have multiple threads reading each entry and test whether it is prime, and return only the prime values into a new, smaller array.

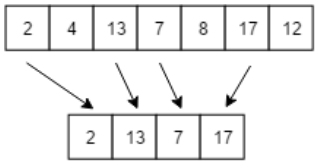

Figure 15.7: Compact Example: Prime

15.4 Blelloch Scan

Blelloc Scan is used to find the total sum of all entries while being both work optimal as well as time efficient. The speed and work is summarized and compared to other total scan sum algorithms we have discussed previously.

Scan Algorithm Complexities		
Scan Algorithm	Time Complexity	Work Complexity
Sequential Scan	O(n)	O(n)
Parallel Prefix Scan (Hillis-Steele)	O(logn)	O(nlogn)
Blelloch Scan	O(logn)	O(n)

For Blelloch Scan, there are two phases to compute the exclusive scan- an upsweep and a downsweep. In the upsweep phase, we do a sum operation on each neighboring pair, keeping track of each of the summed intermediate entries for reference for the downsweep phase. Once the upsweep phase is complete, we set the last node we touch to 0. We then follow it with the downsweep phase, such that at each level we take the previous left entry and add it to the current entry and set that to its new right entry, and carry the current entry to the new left entry.

Upsweep can be computed as:

$$sum[v] = sum[L[v]] + sum[R[v]] \tag{15.1}$$

Downsweep can be computed as:

$$\begin{aligned} scan[L[v]] &= scan[v] \\ scan[R[v]] &= sum[L[v]] + scan[v] \end{aligned} \tag{15.2}$$

Where *sum[L[v]]* is the most recent, nearest, unused summed left value during the upsweep phase.

The parallelism exist since each pair addition is done by an independent thread.

See bottom of document for Blelloch Scan Example

15.5 Brick Sort (Odd-Even Sort)

Brick sort sorts an array of entries like bubble sort with parallelism. It breaks the array into pairs, then compares (and swaps) all pairs into order. Then, on the new array, the entries pair off with the neighbor it did not previously pair with, and repeat. Each pair comparison is handled by an independent thread. This process will take n pairings to complete, therefore O(n).

Brick sort is easy to implement, but is an oblivious sorting algorithm. This means it does not take the values into account and will always take the same amount of time based on the number of entries in the array. Furthermore, brick sort can be viewed as a sorting network, where the entries are the input to the network, and the output will be the sorted entries. In the following figure, the vertical lines are comparators and the horizontal lines are the wire the entries can travel across. The a's are the initial entries and the b's are the sorted entries.

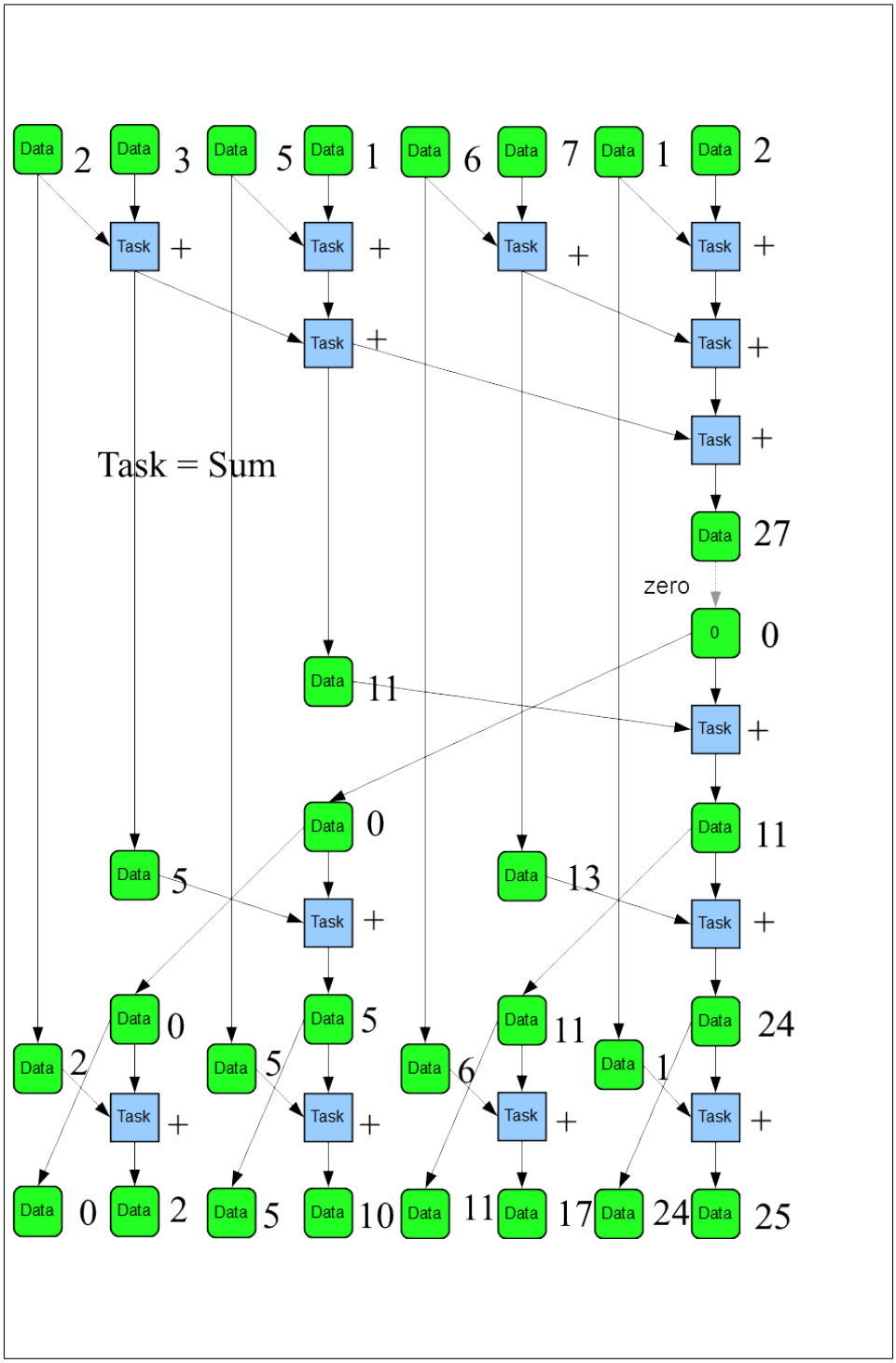

Figure 15.8: Blelloch Scan [4]

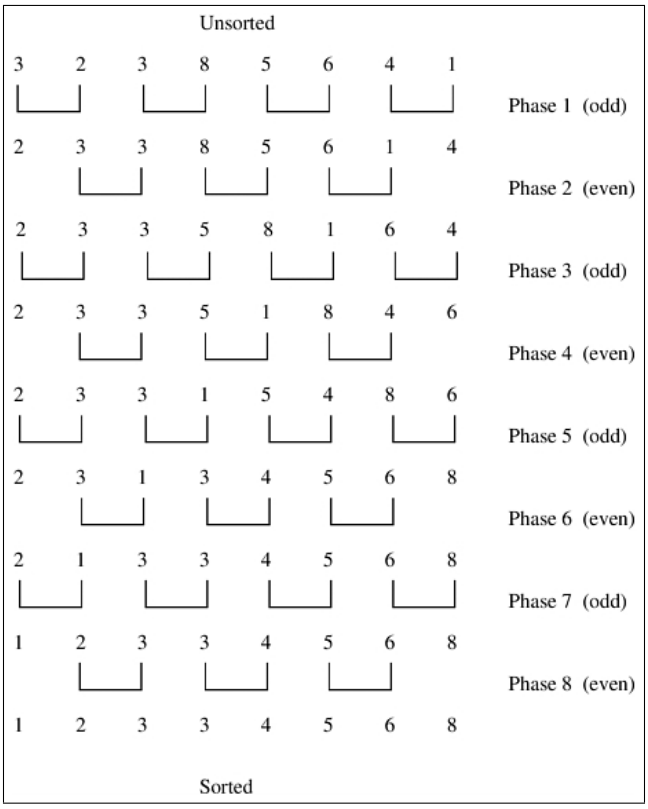

Figure 15.9: Brick Sort [2]

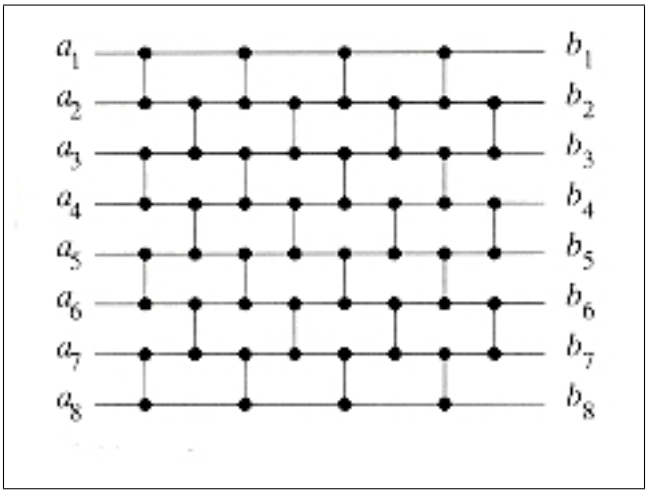

Figure 15.10: Brick Sort Network View [3]

References

[1] https://github.com/vijaygarg1/UT-Garg-EE382C-EE361C-Multicore/blob/master/chapter3-synchronization_primitives/BoundedBufferMonitor.java

[2] http://d1gjlxt8vb0knt.cloudfront.net//wp-content/uploads/Even-Odd-Sort.gif

[3] http://staff.ustc.edu.cn/ csli/graduate/algorithms/book6/651_a.gif

[4] https://scs.senecac.on.ca/ gpu621/pages/images/prefix-scan-balanced.png