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

Lecture 15: October 18

Lecturer: Vijay Garg *Scribe: Ben Fu*

15.1 Introduction

The lecture introduces the concept of *scan*, or *prefix-sum*. Scan is first defined and later implemented using three different algorithms.

15.2 Scanning Algorithms

A *scan* algorithm, also known as a *prefix-sum* algorithm, takes a binary associative operator $\oplus$ and an array of n elements:

$$[a_0, a_1, a_2, ..., a_{n-1}]$$

and returns the array with the operator applied cumulatively such as follows:

$$[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, ..., a_0 \oplus a_1 \oplus a_2 \oplus ... \oplus a_{n-1}]$$

For example, if the operator $\oplus$ is addition and we have an array such as follows:

$$[3, 1, 7, 0, 4, 1, 6, 3]$$

the result of the scan, or prefix-sum, would be the array:

$$[3, 4, 11, 11, 15, 16, 22, 25]$$

For simplicity, we will use addition as the operator $\oplus$ when discussing the following algorithms.

15.2.1 Sequential Scan

Naturally, the sequential algorithm is to iterate through the array, applying the operator to each new element. The pseudocode for the sequential algorithm is as follows [1]:

```
out[0] = in[0]
for all i from 1 to n
    out[i] = out[i-1] + in[i]
```

Code 1: Sequential Scan

The time and work complexity for this algorithm are both O(n).

15.2.2 Hillis-Steele Scan

The second algorithm was introduced by Hillis and Steele, which is a naive parallel implementation of the sequential scan algorithm. The naive parallel algorithm is as follows[1]:

```
out[0] = in[0]
for all i from 1 to log(n)
    for all k in parallel
        if k ≥ 2^d then x[k − 2^(d−1)] + x[k]
```

Code 2: Hillis-Steele Scan

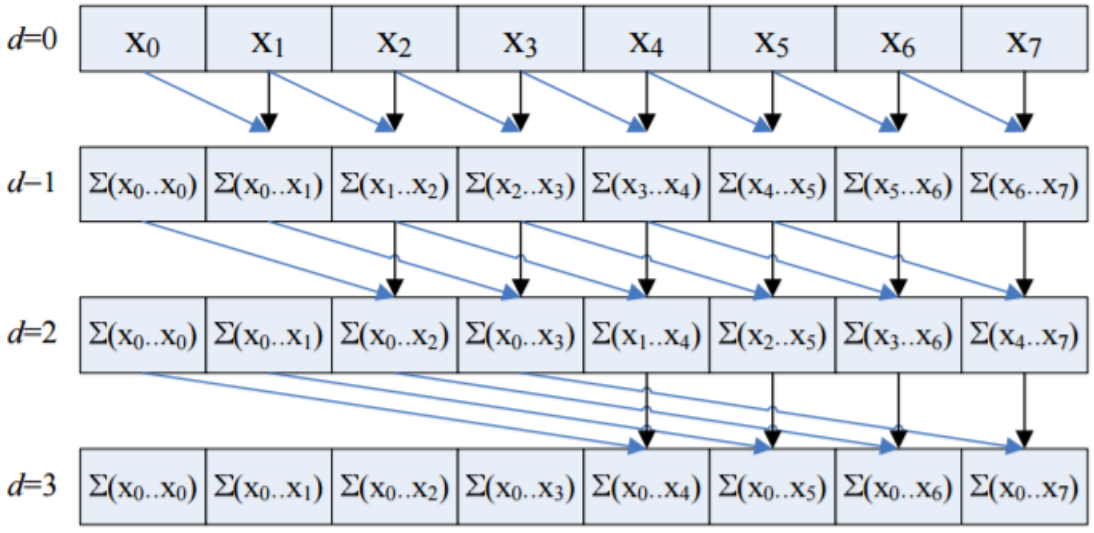

Figure 15.1: Hillis-Steele naive parallel scan

The algorithm is correct but not work optimal. The total number of operations the algorithm performs is $O(nlogn)$, which is sub-optimal.

15.2.3 Blelloch Scan

To build a work-optimal solution to scan, we use a balanced binary tree. Although an actual data structure is not allocated, the solution builds partial sums that can be represented by a balanced binary tree. In this solution, which was formed by Blelloch [2], there are two phases: an *upsweep* and a *downsweep* phase.

In the upsweep phase, each thread computes the partial sums of the two lower-level elements. This can be visualized by the following figure:

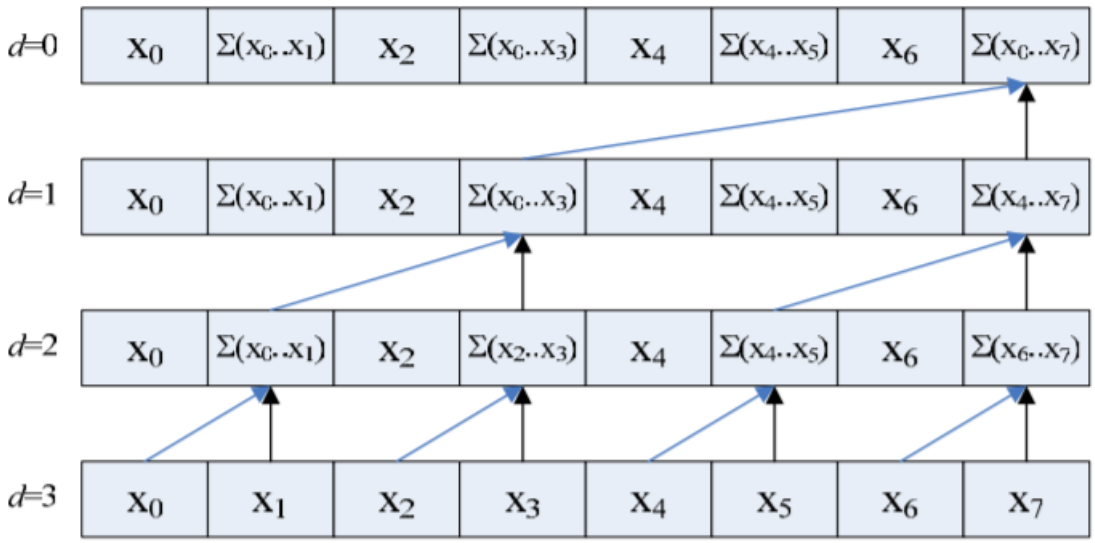

Figure 15.2: Upsweep phase in Blelloch Scan

The algorithm is as follows [1]:

```
for all d from 0 to log(n−1)
    for all k from 0 to n−1 by 2^{d+1} in parallel
        x[k + 2^{d+1} − 1] = x[k + 2^d − 1] + x[k + 2^{d+1} − 1]
```

Code 3: Upsweep Phase

In the downsweep phase, we build the scan using the partial sums computed during the upsweep phase, replacing the elements with the scan output in place. Note that the example provided is an exclusive scan; that is, a zero is inserted into the first step of the downsweep and the final array does not include the total sum.

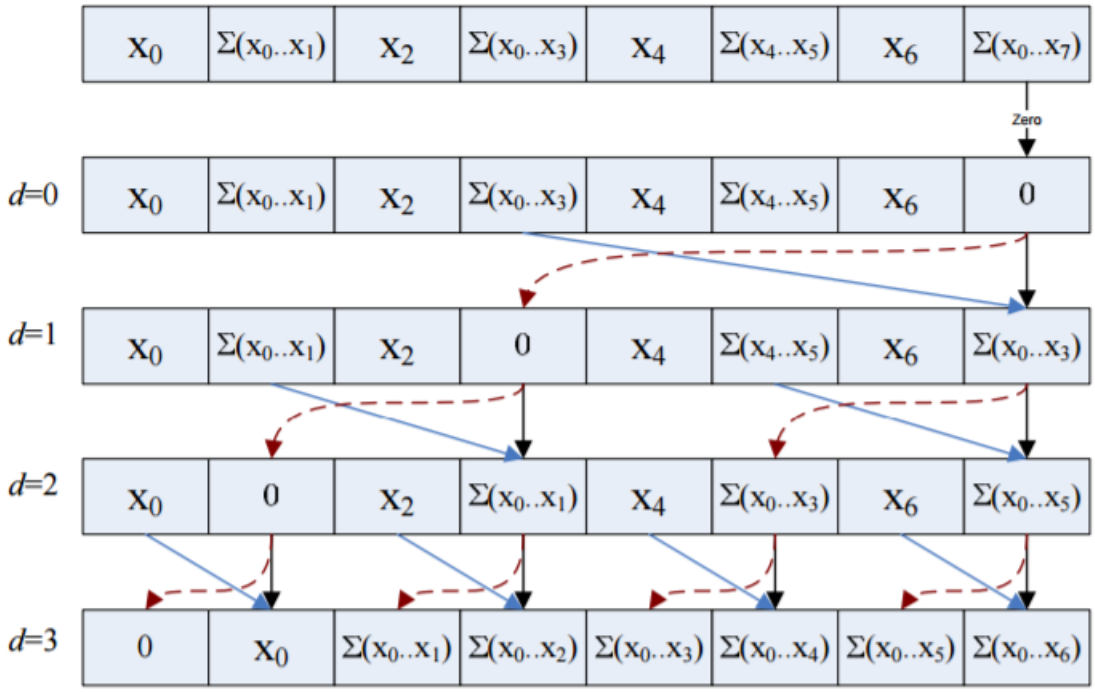

Figure 15.3: Downsweep phase in Blelloch Scan

The algorithm is as follows:

```
x[n - 1] = 0
for all d from log(n) down to 0
    for all k from 0 to n-1 by 2^(d+1) in parallel
        temp = x[k + 2^d - 1]
        x[k + 2^d - 1] = x[k + 2^(d+1) - 1]
        x[k + 2^(d+1) - 1] = temp + x[k + 2^(d+1) - 1]
```

Code 4: Downsweep Phase

A sample implementation of the Blelloch algorithm in CUDA C code is shown on the next page.

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[];// allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

A   temp[2*thid]   = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d)
        {
B           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }
        offset *= 2;
    }

C   if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
D           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t    = temp[ai];
            temp[ai]   = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();

E   g_odata[2*thid]   = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```

Figure 15.4: CUDA implementation for Blelloch Scan

15.2.4 Complexity Table

The complexity table for the time and work complexities of the previous three algorithms are shown in the following table:

Scan Algorithm	Time Complexity	Work Complexity
Sequential	$O(n)$	$O(n)$
Hillis-Steele	$O(log(n))$	$O(nlog(n))$
Blelloch (Work Optimal)	$O(log(n))$	$O(n)$

Table 15.1: Analysis of Scan Algorithms

References

[1] M. Harris, Parallel Prefix Sum (Scan) with CUDA, *Nvidia Corporation* (2007), pp. 3–10.

[2] G. E. Blelloch, Prefix Sums and Their Applications, *Synthesis of Parallel Algorithms* (1990).