CS 4370/6370

Parallel Programming for Many-Core GPUs

This lecture slide is mainly based on the power point slides by the textbook author, Wen-mei Hwu (UIUC) and David Kirk (NVIDIA).

(Exclusive) Prefix-Sum (Scan) Definition

Definition: *The* all-prefix-sums *operation takes a binary associative operator* $\oplus$, *and an array of n elements*

$$[x_0, x_1, \ldots, x_{n-1}]$$

and returns the array

$$[0, x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-2})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array

[3 1 7 0 4 1 6 3],

would return

[0 3 4 11 11 15 16 22].

Why Exclusive Scan

To find the beginning address of allocated buffers

Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

input	[3 1 7 0 4 1 6 3]
Exclusive	[0 3 4 11 11 15 16 22]
Inclusive	[3 4 11 11 15 16 22 25]

Adapt an Inclusive, work-inefficient Prefix-Sum (Scan)

Block 0:

Thread 0 loads 0 into XY[0]

Other threads load X[threadIdx.x-1] into XY[threadIdx.x]

All other blocks:

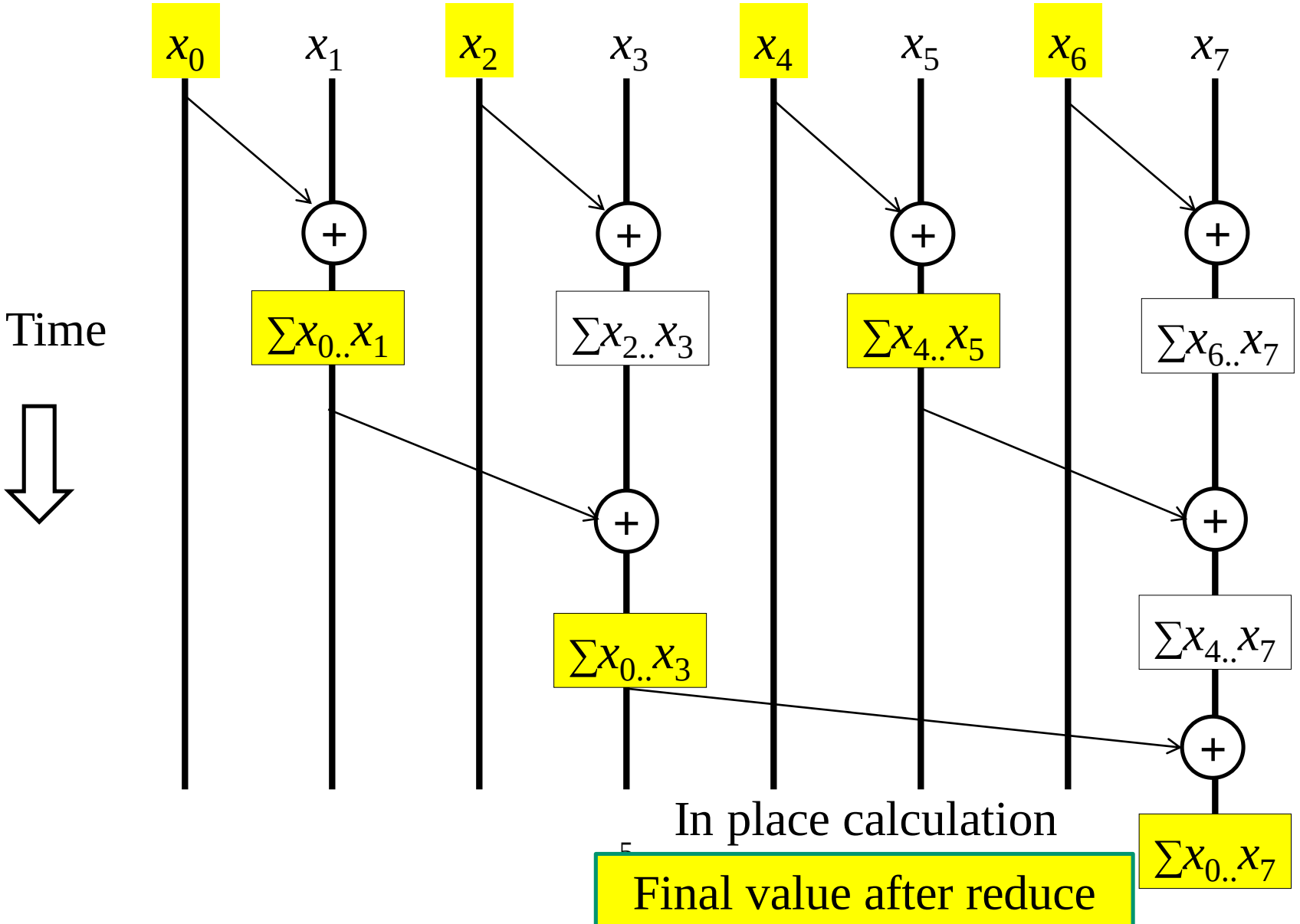
All thread load X[blockIdx.x*blockDim.x+threadIdx.x-1] into XY[threadIdex.x]

Similar adaption for work efficient scan kernel but pay attention that each thread loads two elements

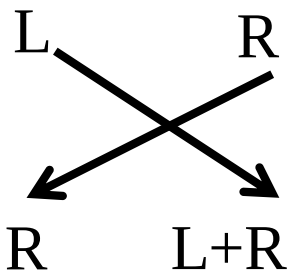
Only one zero should be loaded

All elements should be shifted by only one position

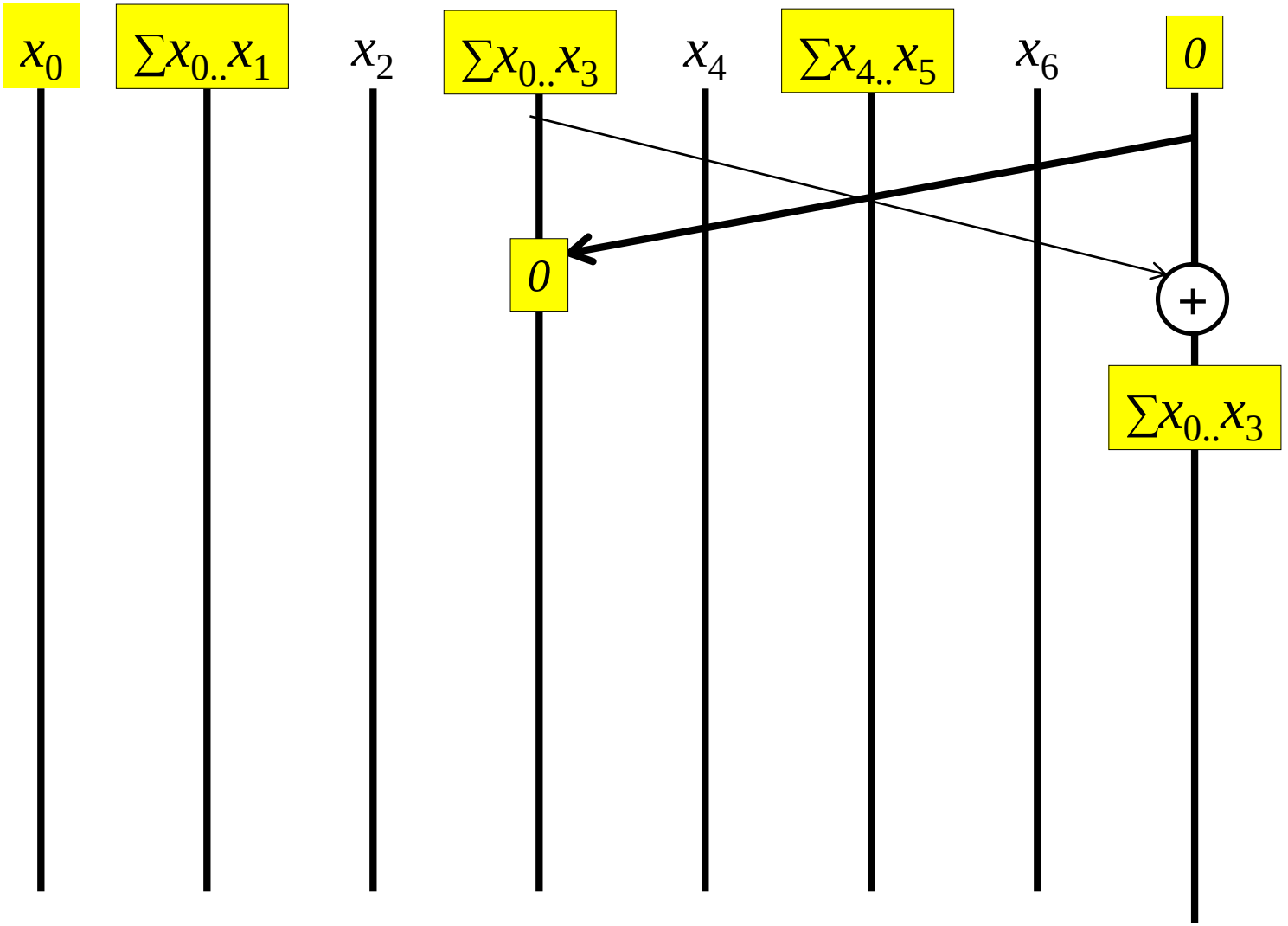
Parallel Scan - Reduction Step



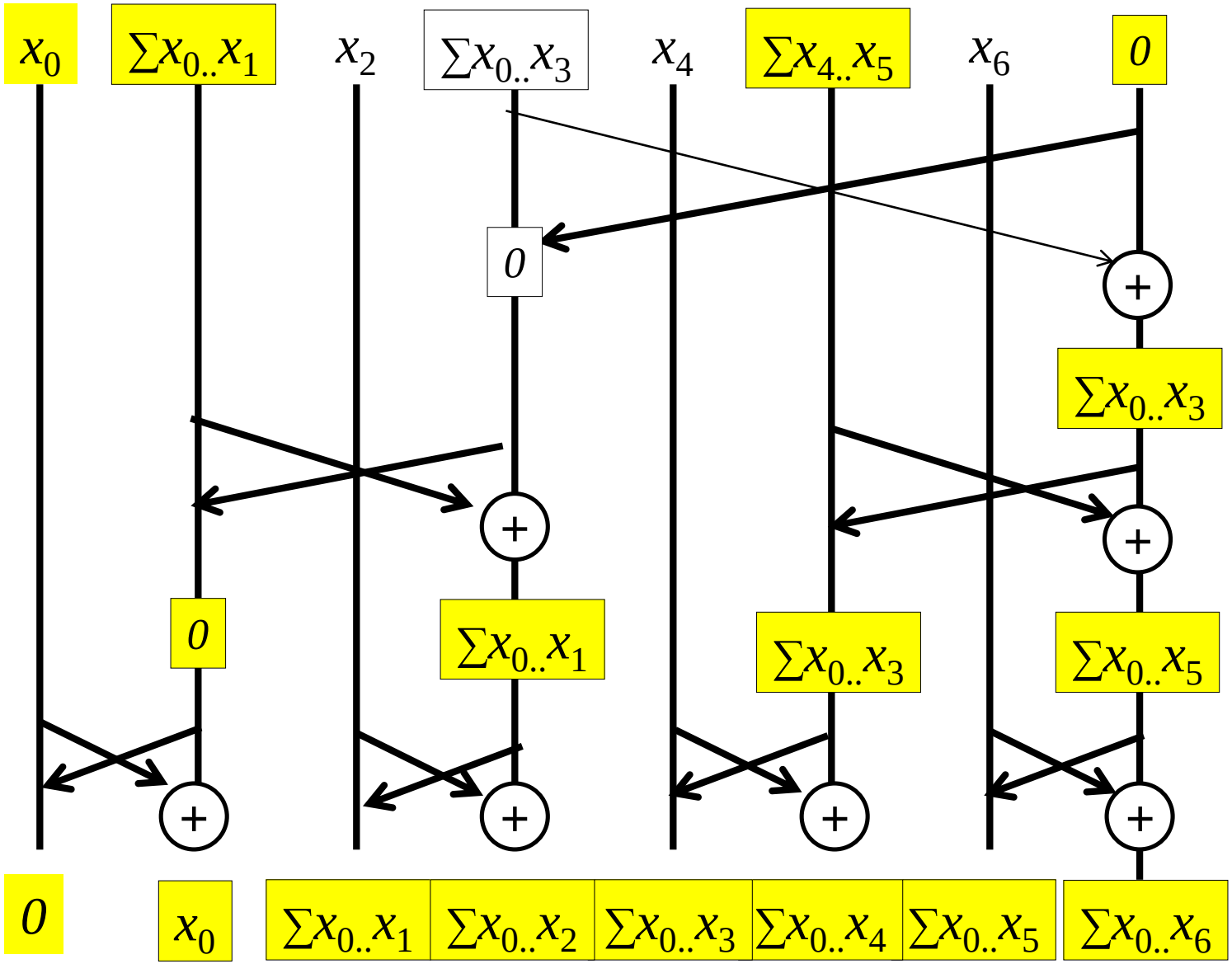
An Exclusive Post Scan Step (Add-move Operation)



Add-move



Exclusive Post Scan Step



Exclusive Post Scan Step

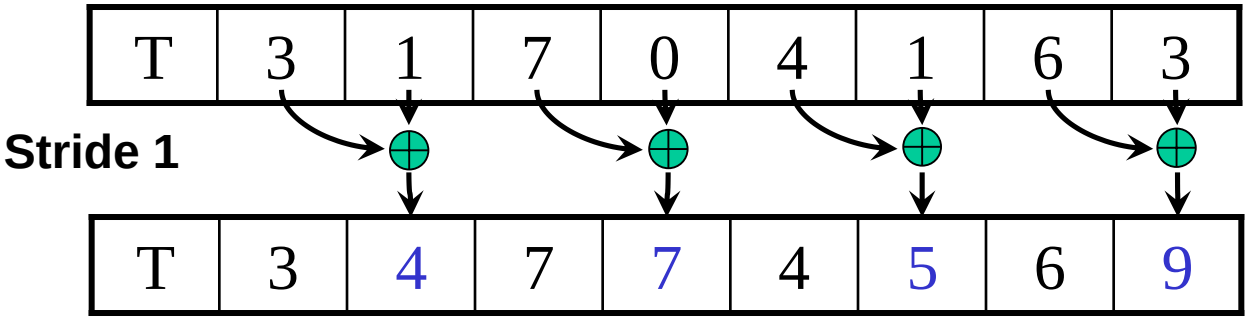
```
if (threadIdx.x == 0) scan_array[2*blockDim.x-1] = 0;
 int stride = BLOCK_SIZE;
 while(stride > 0)
 {
     int index = (threadIdx.x+1)*stride*2 - 1;
     if(index  < 2* BLOCK_SIZE)
     {
             float temp = array[index];
             scan_array[index] += scan_array[index-stride];
             scan_array[index-stride] = temp;
     }
     stride = stride / 2;
     __syncthreads();
 }
```

Exclusive Scan Example – Reduction Step

T	3	1	7	0	4	1	6	3

Assume array is already in shared memory

Reduction Step (cont.)

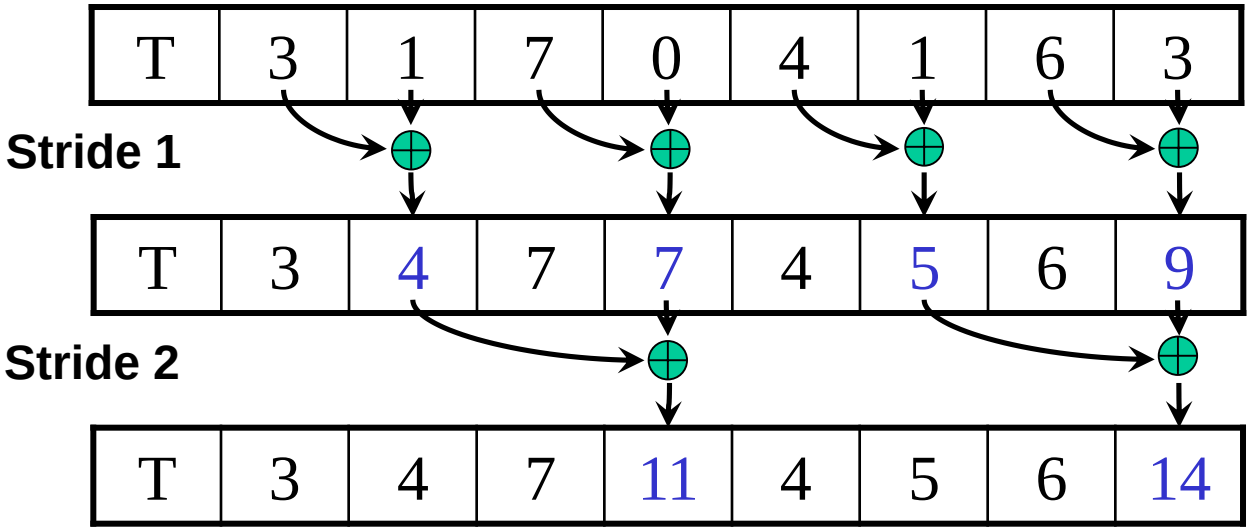


Iteration 1, *n*/2 threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value

Reduction Step (cont.)

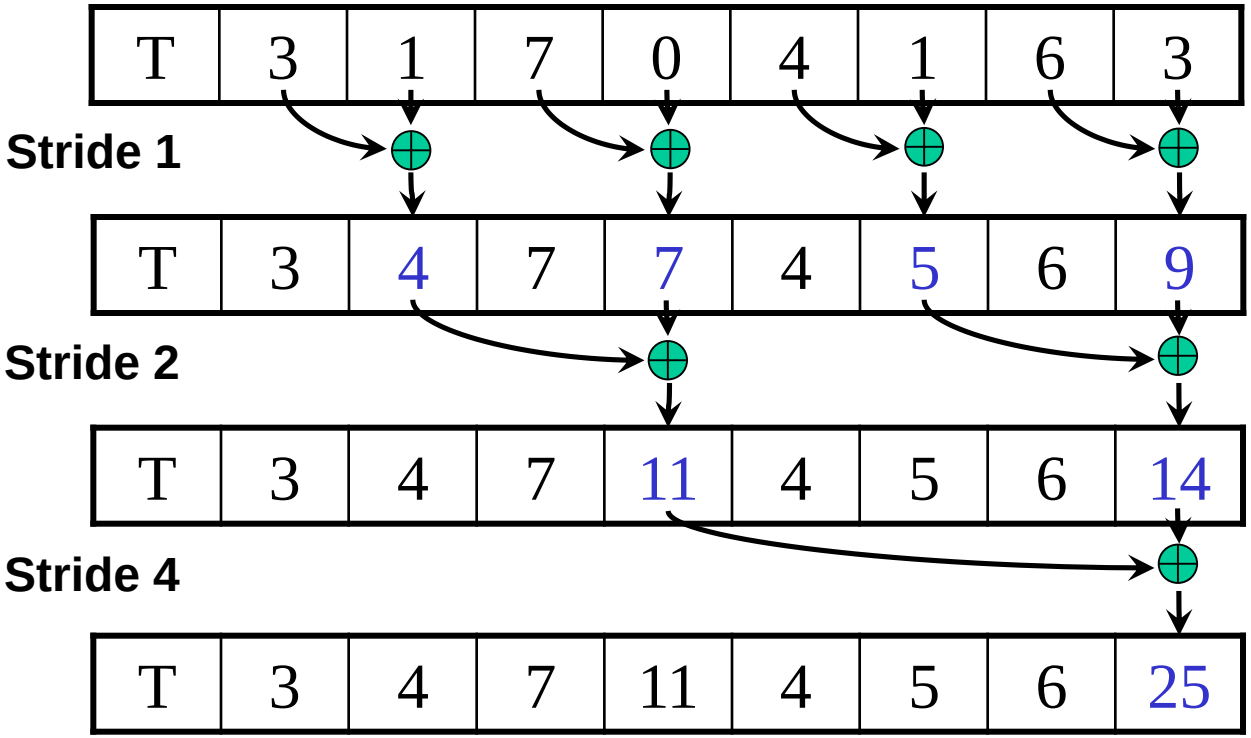


Iteration 2, *n*/4 threads

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value

Reduction Step (cont.)



Iteration log(*n*), 1 thread

Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value.

Note that this algorithm operates in-place: no need for double buffering

Zero the Last Element

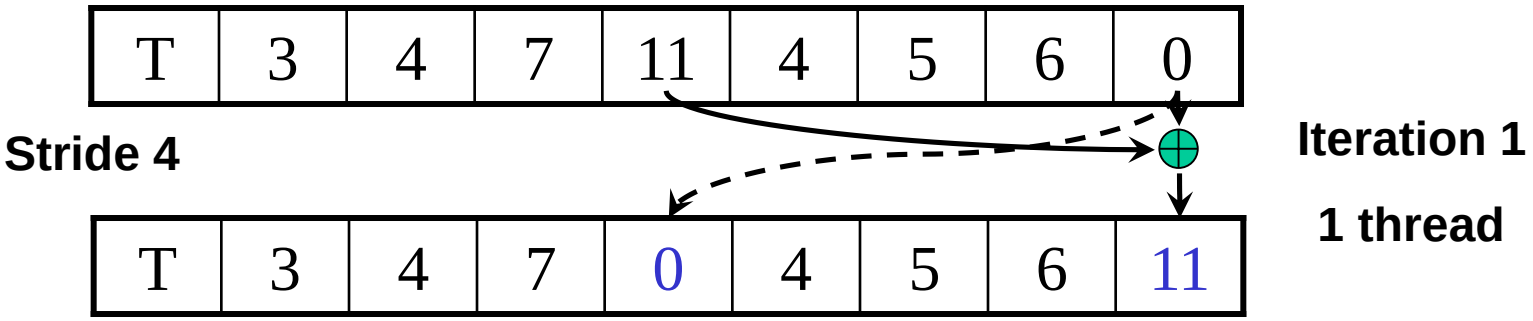
T	3	4	7	11	4	5	6	0

We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Post Scan Step from Partial Sums

T	3	4	7	11	4	5	6	0

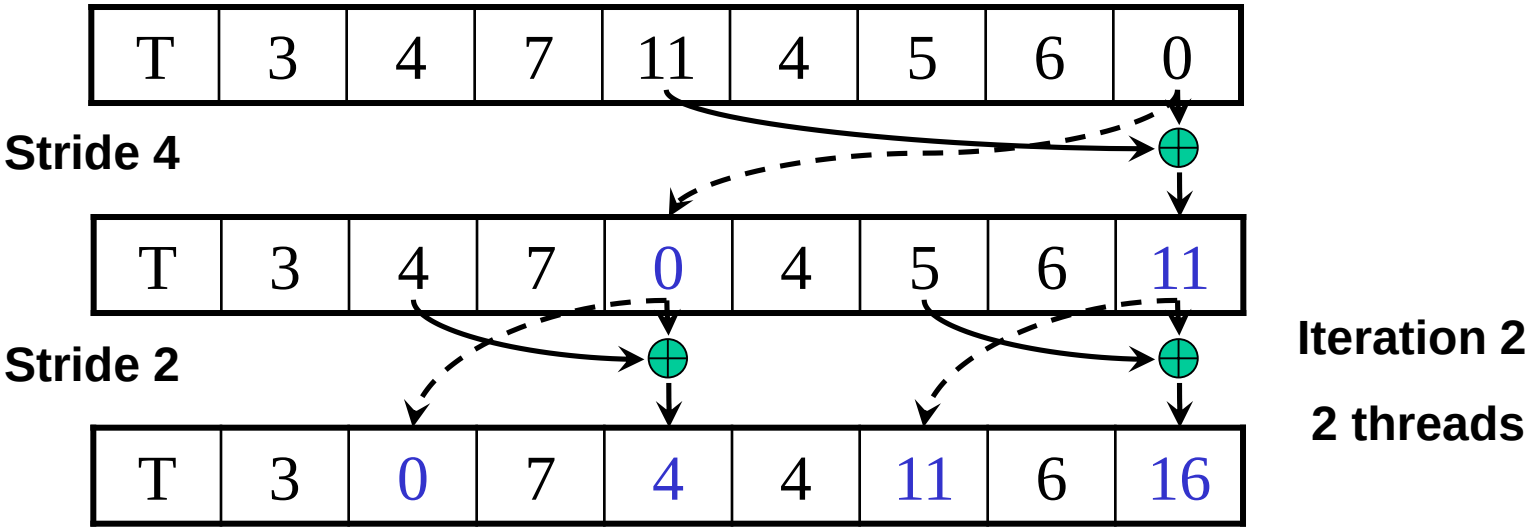
Post Scan Step from Partial Sums (cont.)



Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

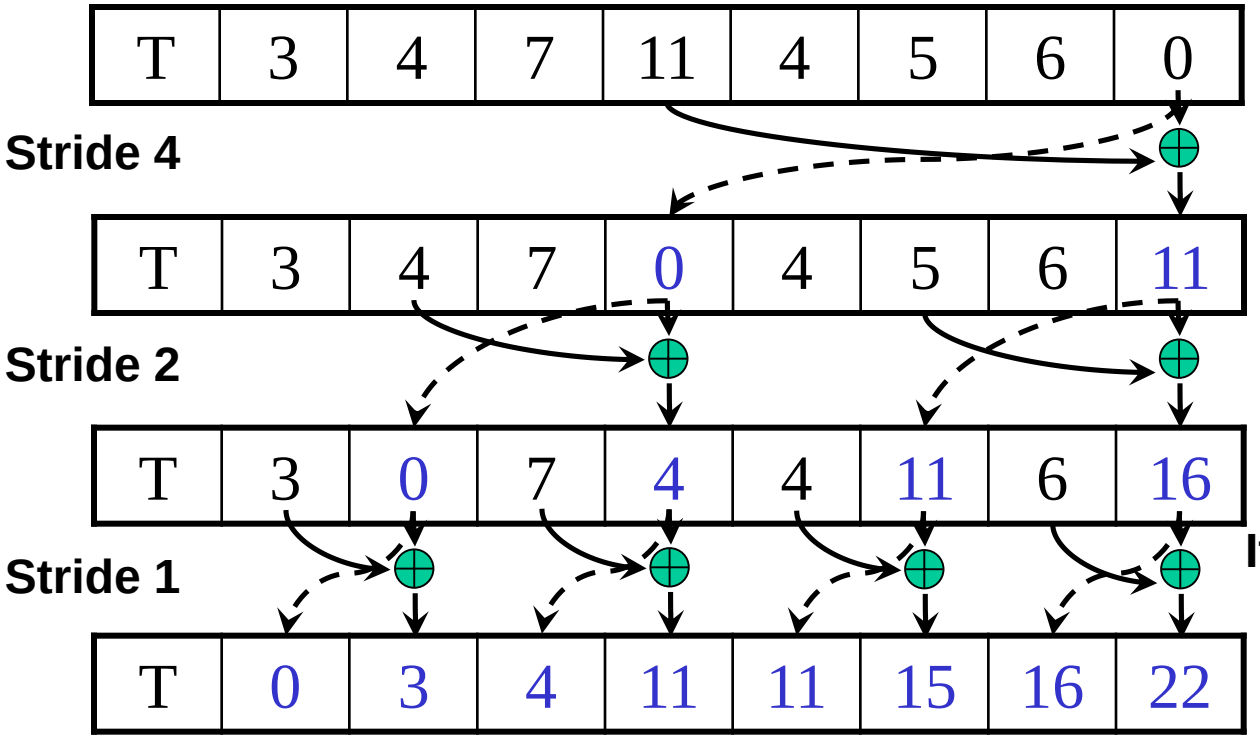
Post Scan From Partial Sums (cont.)



Each ⊕ corresponds to a single thread.

Iterate log(n) times. Each thread adds value *stride* elements away to its own value, and sets the value *stride* elements away to its own *previous* value.

Post Scan Step From Partial Sums (cont.)



Iteration log(*n*)
***n*/2 threads**

Each ⊕ corresponds to a single thread.

Done! We now have a completed scan that we can write out to device memory.

Total steps: 2 * log(*n*).
Total work: 2 * (*n*-1) adds = *O*(*n*) **Work Efficient!**

Work Analysis

The parallel Inclusive Scan executes 2* log(n) parallel iterations

- log(n) in reduction and log(n) in post scan
- The iterations do n/2, n/4,..1, 1, …., n/4. n/2 adds
- Total adds: 2* (n-1) → O(n) work

The total number of adds is no more than twice of that done in the efficient sequential algorithm

- The benefit of parallelism can easily overcome the 2X work when there is sufficient hardware