Parallel Prefix Sum – Scan

Objective

- **To master parallel Prefix Sum (Scan) algorithms**
  - Frequently used for parallel work assignment and resource allocation
  - A key primitive in many parallel algorithms to convert serial computation into parallel computation
  - Based on reduction tree and reverse reduction tree
- **Reading – Mark Harris, Parallel Prefix Sum with CUDA**
  - http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf

(Inclusive) 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

$$[x_0, (x_0 \oplus x_1), \ldots, (x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1})].$$

Example: If $\oplus$ is addition, then the all-prefix-sums operation on the array [3 1 7 0 4 1 6 3], would return [3 4 11 11 15 16 22 25].

Inclusive Scan Application Example

- Assume we have a 100-inch sandwich to feed 10
- We know how many inches each person wants
  - [3 5 2 7 28 4 3 0 8 1]
- How do we cut the sandwich quickly?
- How much will be left?

- Method 1: cut the sections sequentially: 3 inches first, 5 inches second, 2 inches third, etc.
- Method 2: calculate Prefix scan and cut in parallel
  - [3, 8, 10, 17, 45, 49, 52, 52, 60, 61] (39 inches left)

Typical Applications of Scan

- **Scan is a simple and useful parallel building block**
  - Convert recurrences from sequential :

```
for(j=1;j<n;j++)
    out[j] = out[j-1] + f(j);
```

  - into parallel:

```
forall(j) { temp[j] = f(j) };
scan(out, temp);
```

- **Useful for many parallel algorithms:**

•Radix sort
•Quicksort
•String comparison
•Lexical analysis
•Stream compaction
•Polynomial evaluation
•Solving recurrences
•Tree operations
•Histograms
•Etc.

Other Applications

- Assigning space in farmers market
- Allocating memory to parallel threads
- Allocating memory buffer for communication channels
- …

A Inclusive Sequential Prefix-Sum

Given a sequence $[x_0, x_1, x_2, \ldots]$

Calculate output $[y_0, y_1, y_2, \ldots]$

Such that $y_0 = x_0$

$$y_1 = x_0 + x_1$$

$$y_2 = x_0 + x_1 + x_2$$

…

Using a recursive definition

$$y_i = y_{i-1} + x_i$$

A Work Efficient C Implementation

```
y[0] = x[0];
for (i=1; i < Max_i; i++)
    y[i] = y[i-1] + x[i];
```

Computationally efficient:

N additions needed for N elements - O(N)

A Naïve Inclusive Parallel Scan

- Assign one thread to calculate each y element
- Have every thread add up all x elements needed for the y element

$$y_0 = x_0$$
$$y_1 = x_0 + x_1$$
$$y_2 = x_0 + x_1 + x_2$$

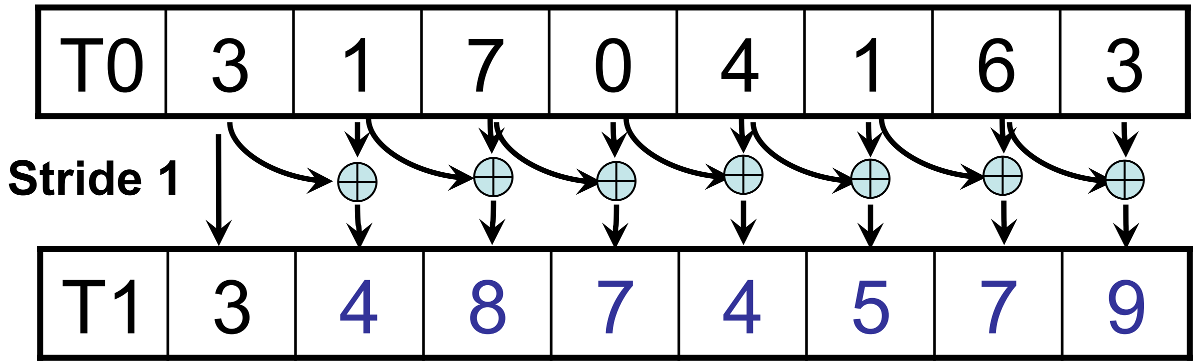
Parallel programming is easy as long as you don't care about performance.

A Slightly Better Parallel Inclusive Scan Algorithm

T0	3	1	7	0	4	1	6	3

Each thread reads one value from the input array in device memory into shared memory array T0. Thread 0 writes 0 into shared memory array.

1. Read input from device memory to shared memory



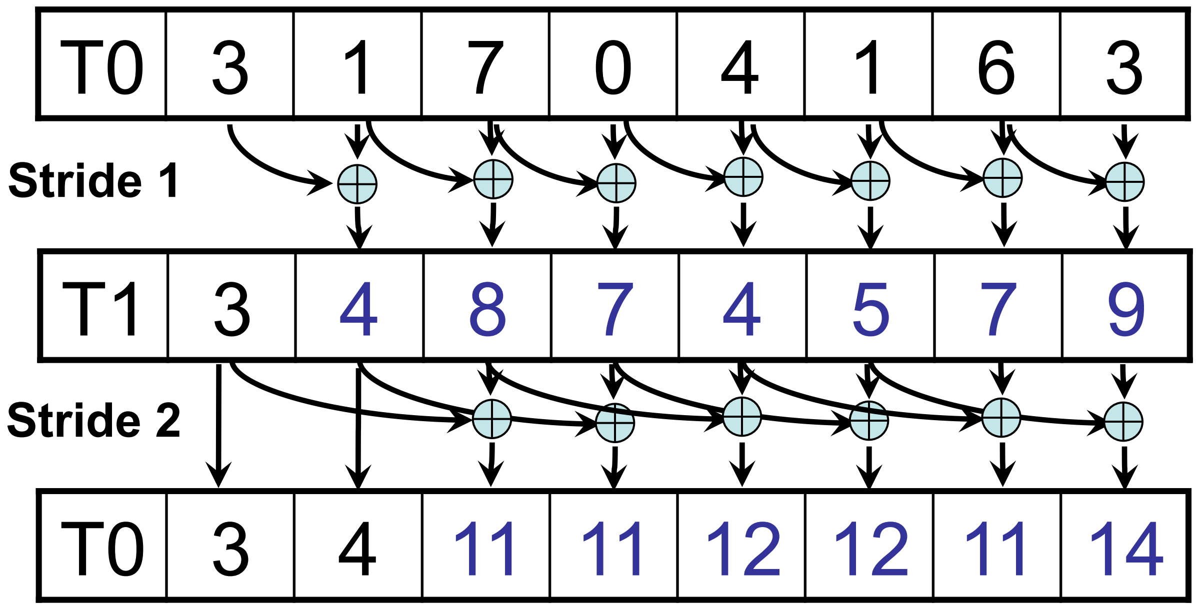
1. Read input from device memory to shared memory

2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (*note*: must double buffer shared mem arrays)

Iterate #1
Stride = 1

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T0 and writes result into shared memory buffer T1 (ping-pong)

Scan



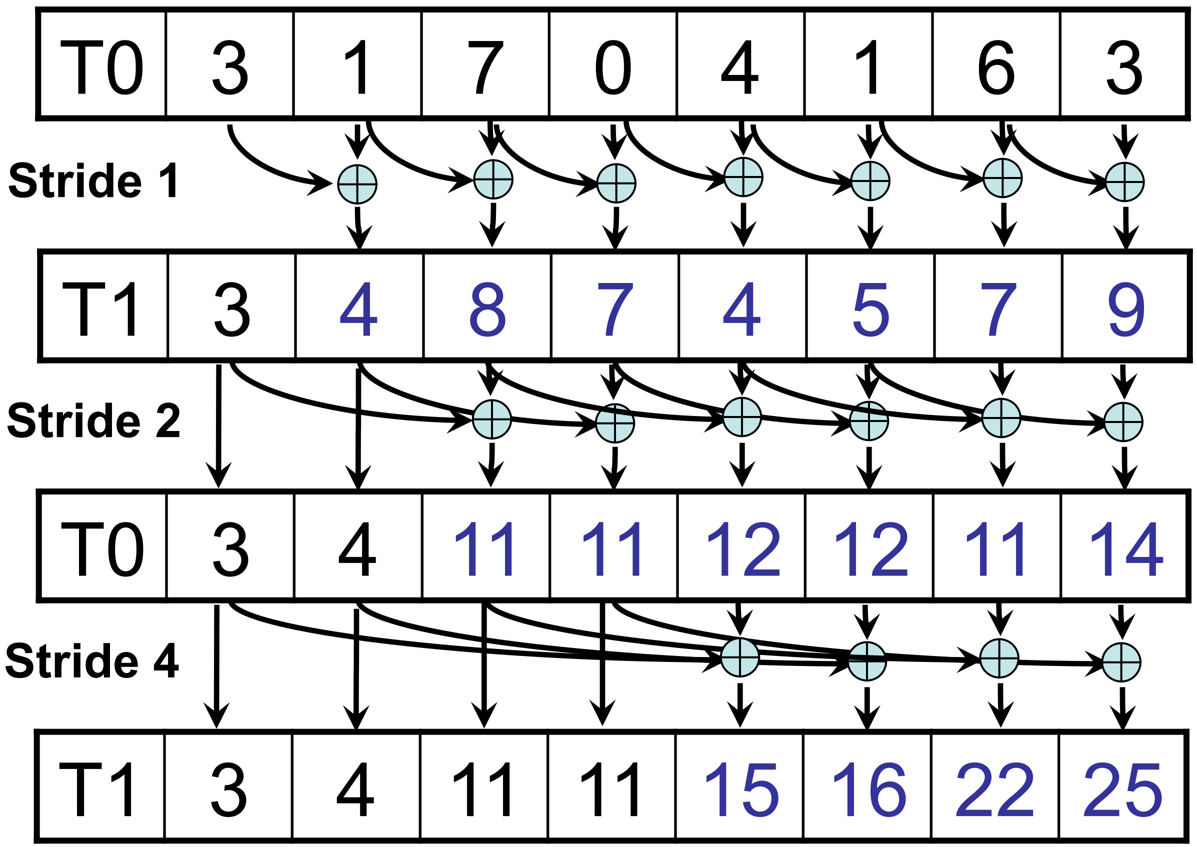
1. (Read input from device memory to shared memory

2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (*note*: must double buffer shared mem arrays)

Iterate #2
Stride = 2

- Active threads: *stride* to *n*-1 (*n-stride* threads)
- Thread *j* adds elements *j* and *j-stride* from T1 and writes result into shared memory buffer T0 (ping-pong)

Scan



1. (Read input from device memory to shared memory

2. Iterate log(n) times: Threads *stride* to *n:* Add pairs of elements *stride* elements apart. Double *stride* at each iteration. (*note*: must double buffer shared mem arrays)

3. Write output from shared memory to device memory

Iterate #3
Stride = 4

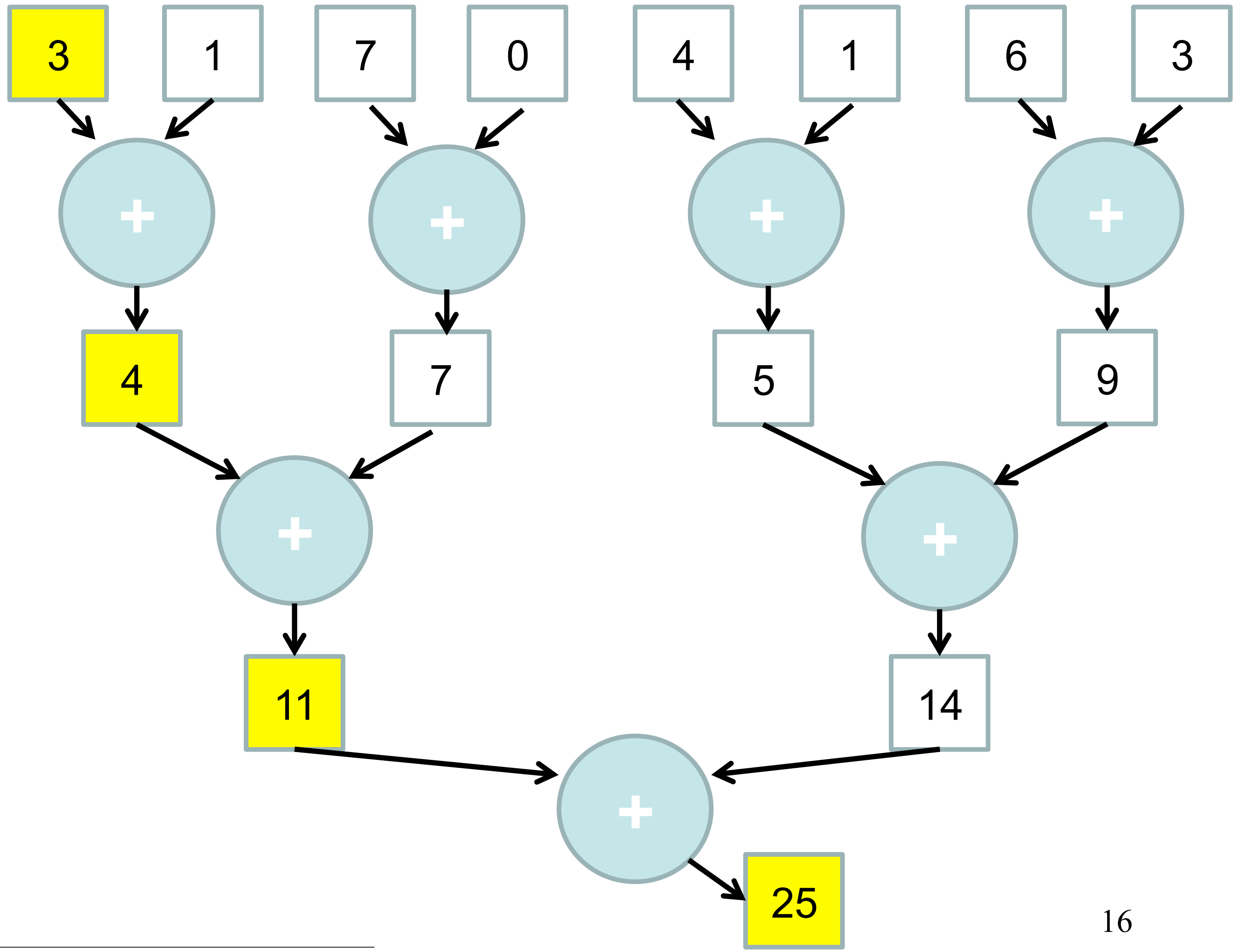
Work Efficiency Considerations

- **The first-attempt Scan executes log(n) parallel iterations**
  - The steps do (n-1), (n-2), (n-4),..(n - n/2) adds each
  - Total adds: n * log(n) - (n-1) → **O(n*log(n))** work
- **This scan algorithm is not very work efficient**
  - Sequential scan algorithm does *n* adds
  - A factor of log(n) hurts: 20x for 10^6 elements!
- **A parallel algorithm can be slow when execution resources are saturated due to low work efficiency**

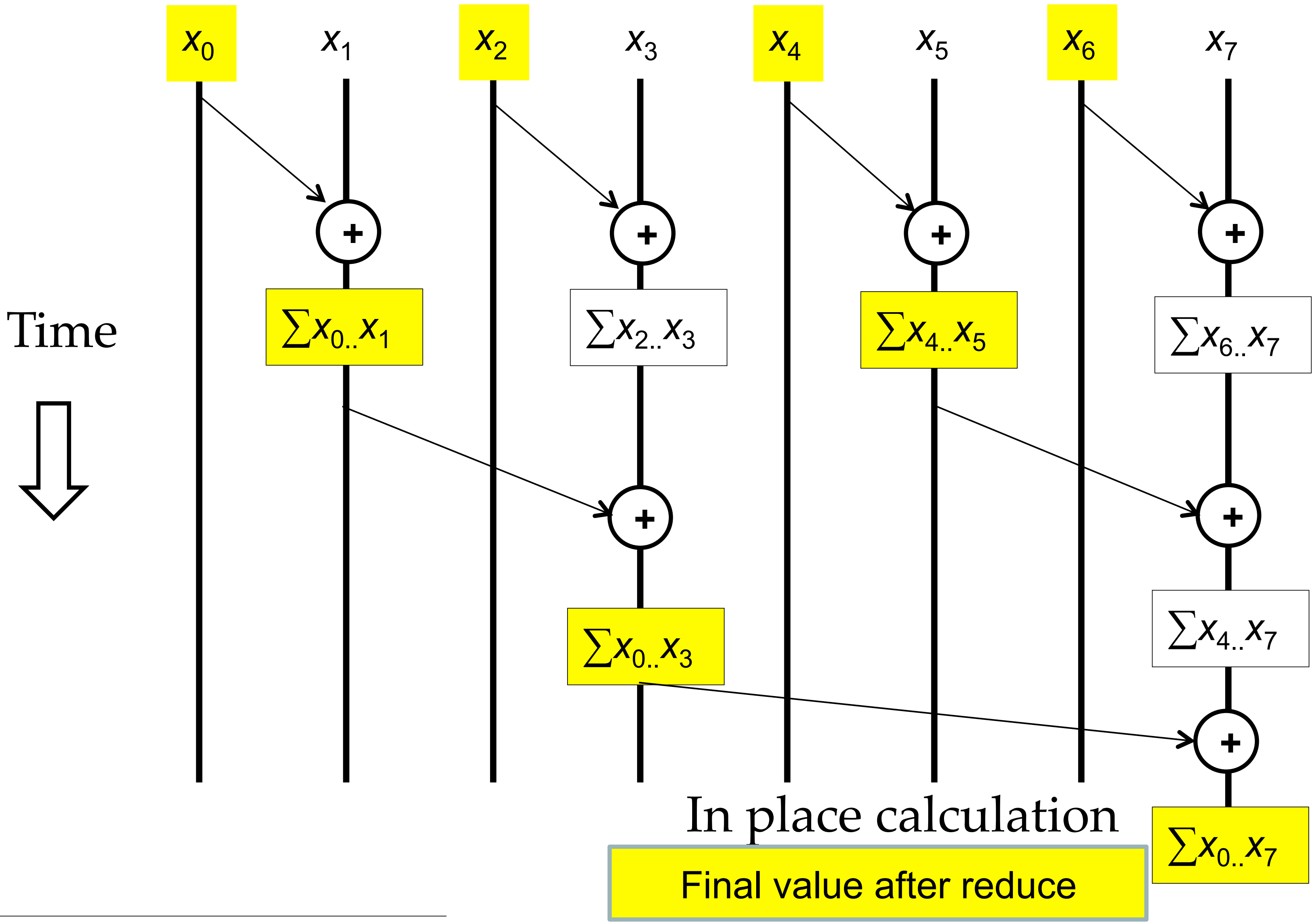
Improving Efficiency

- **A common parallel algorithm pattern:**
  Balanced Trees
  - Build a balanced binary tree on the input data and sweep it to and from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step
- **For scan:**
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves
  - Traverse back up the tree building the scan from the partial sums

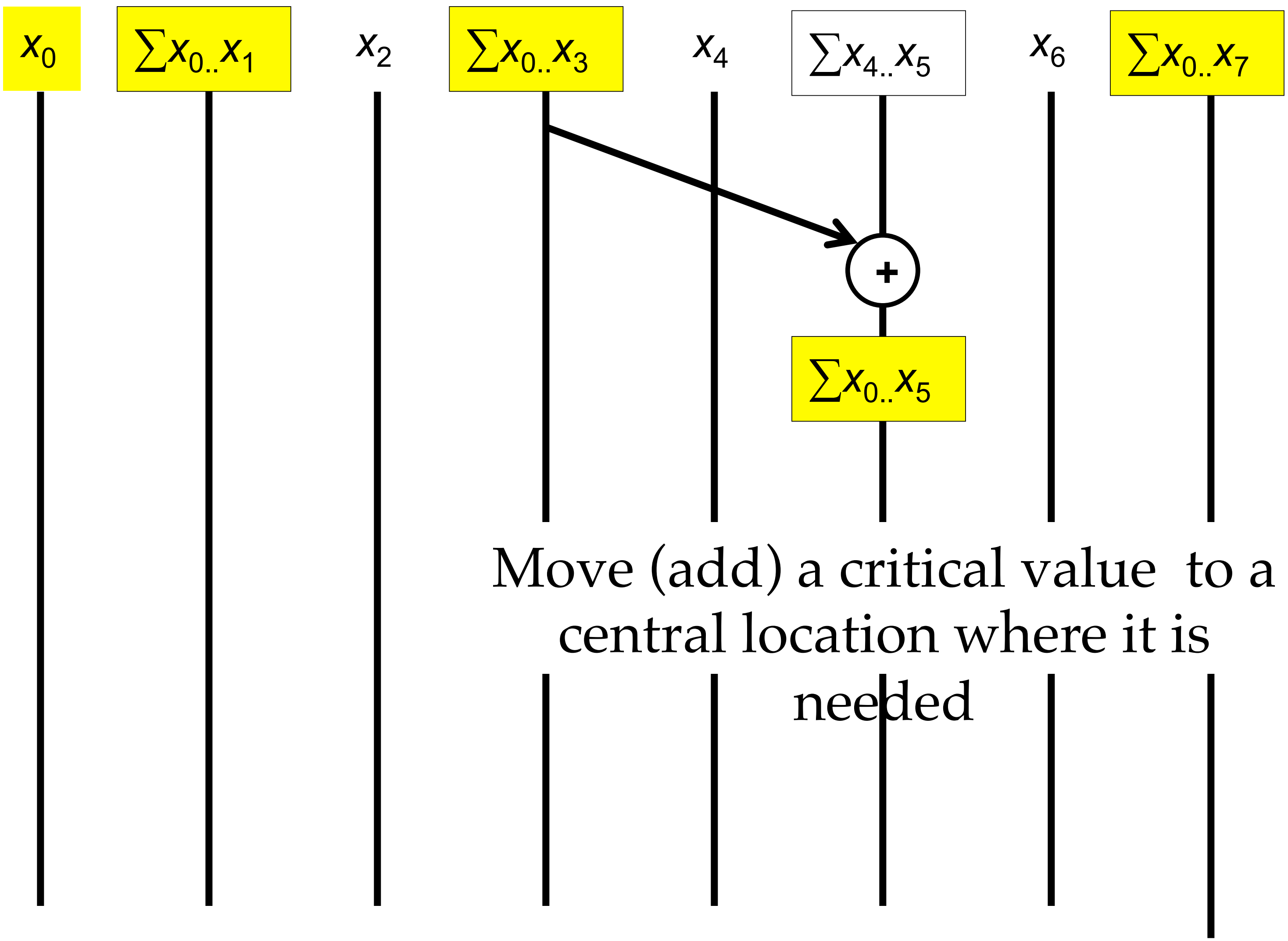
Let's Look at the Reduction Tree Again



Parallel Scan – Reduction Step

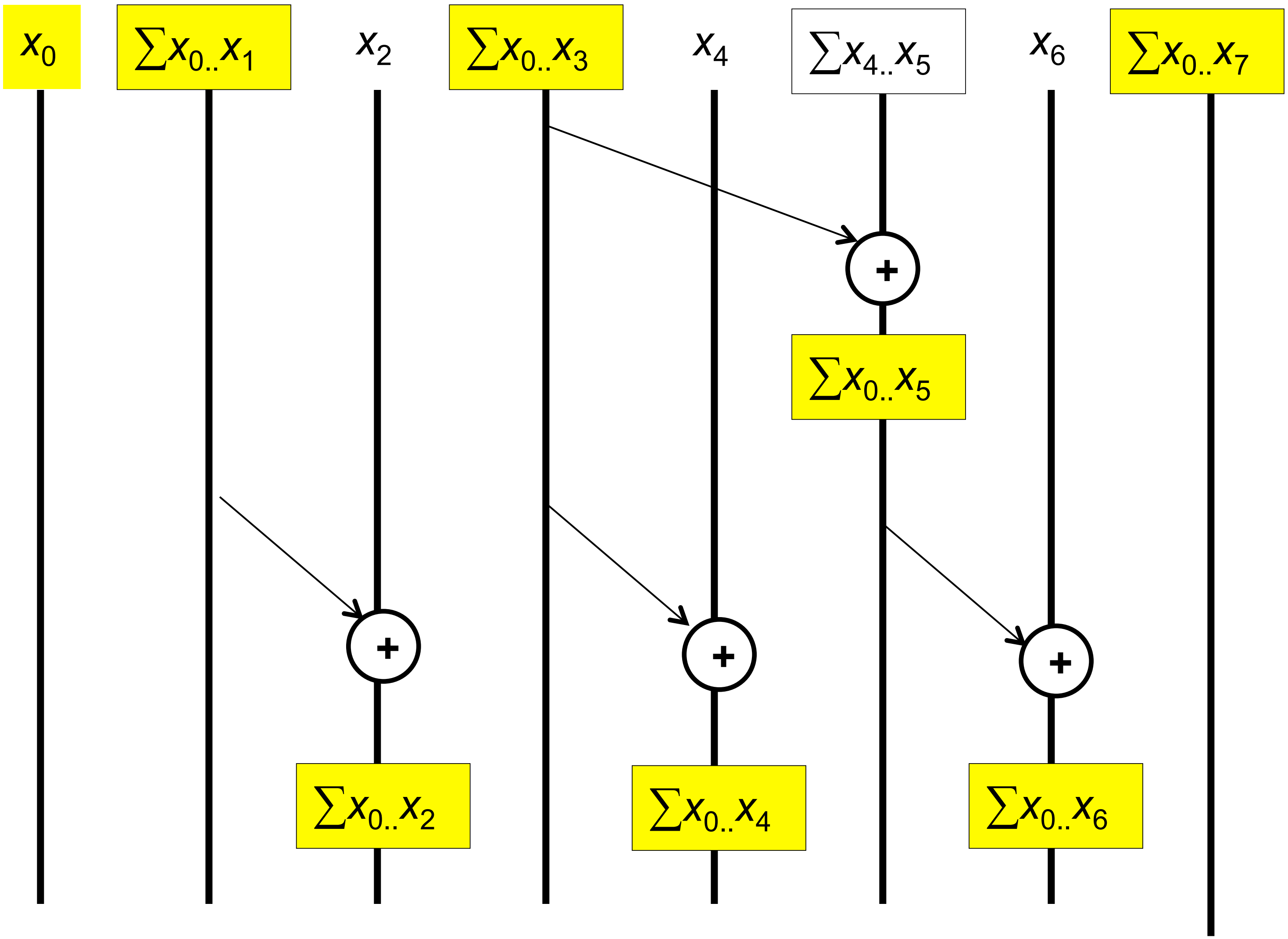


Inclusive Post Scan Step

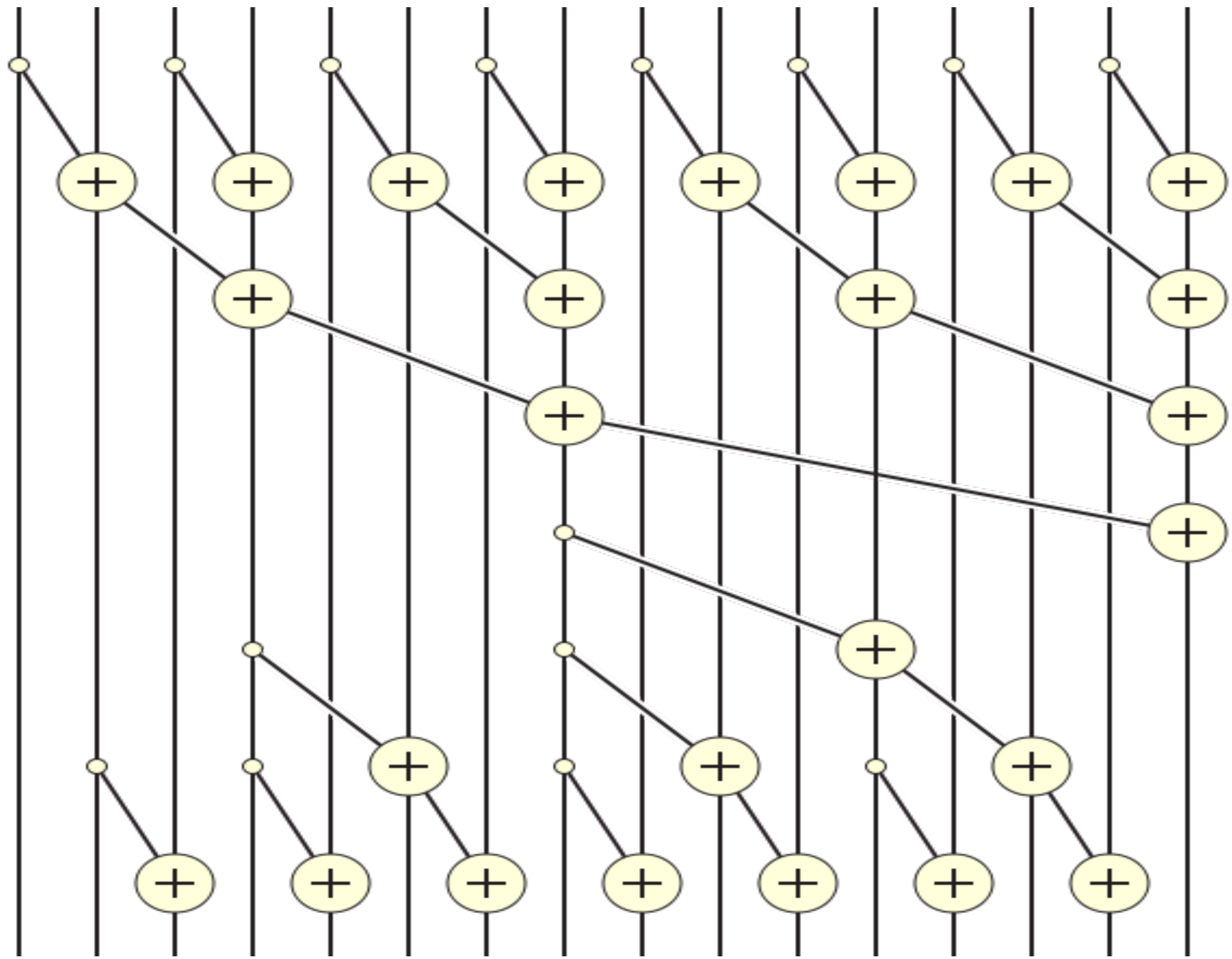


Move (add) a critical value to a central location where it is needed

Inclusive Post Scan Step



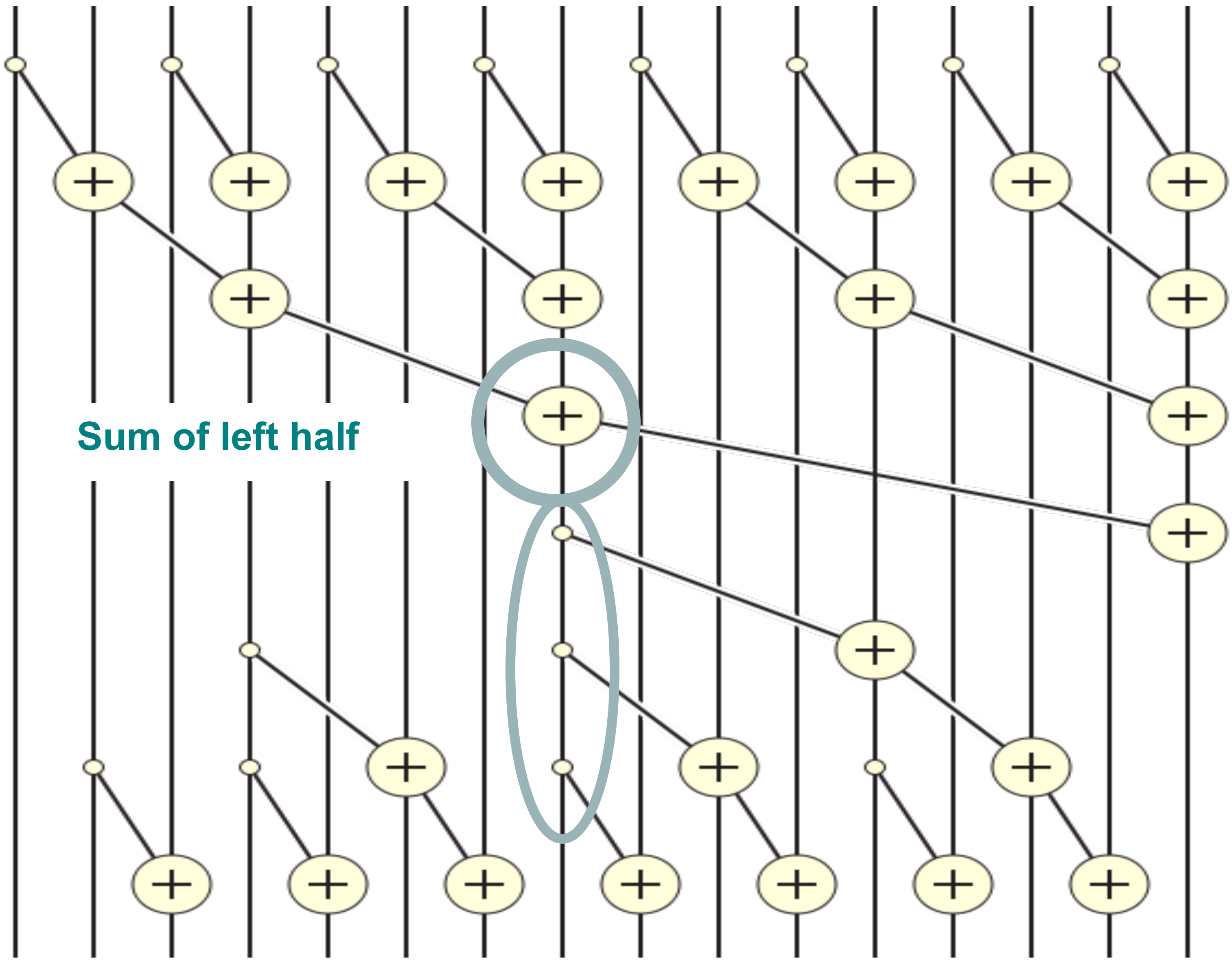
Putting it Together



Reduction Step Kernel Code

```
// scan_array[2*BLOCK_SIZE] is in shared memory

int stride = 1;
while(stride <= BLOCK_SIZE)
{
    int index = (threadIdx.x+1)*stride*2 - 1;
    if(index < 2*BLOCK_SIZE)
        scan_array[index] += scan_array[index-stride];
    stride = stride*2;

    __syncthreads();
}
```

threadIdx.x+1 = 1, 2, 3, 4….
stride = 1, index =


Sum of left half

Post Scan Step

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

(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].

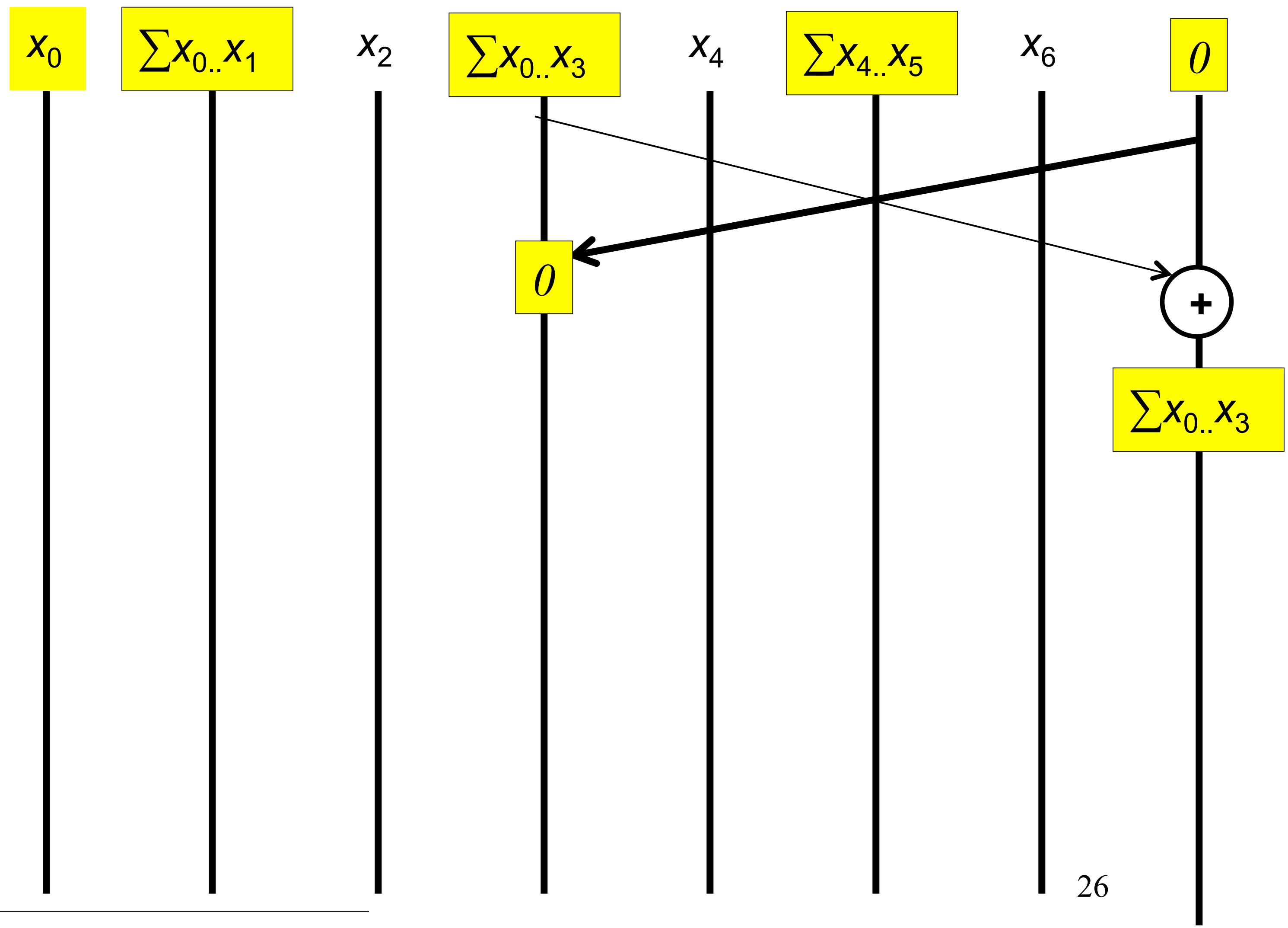
Why Exclusive Scan

- To find the beginning address of allocated buffers

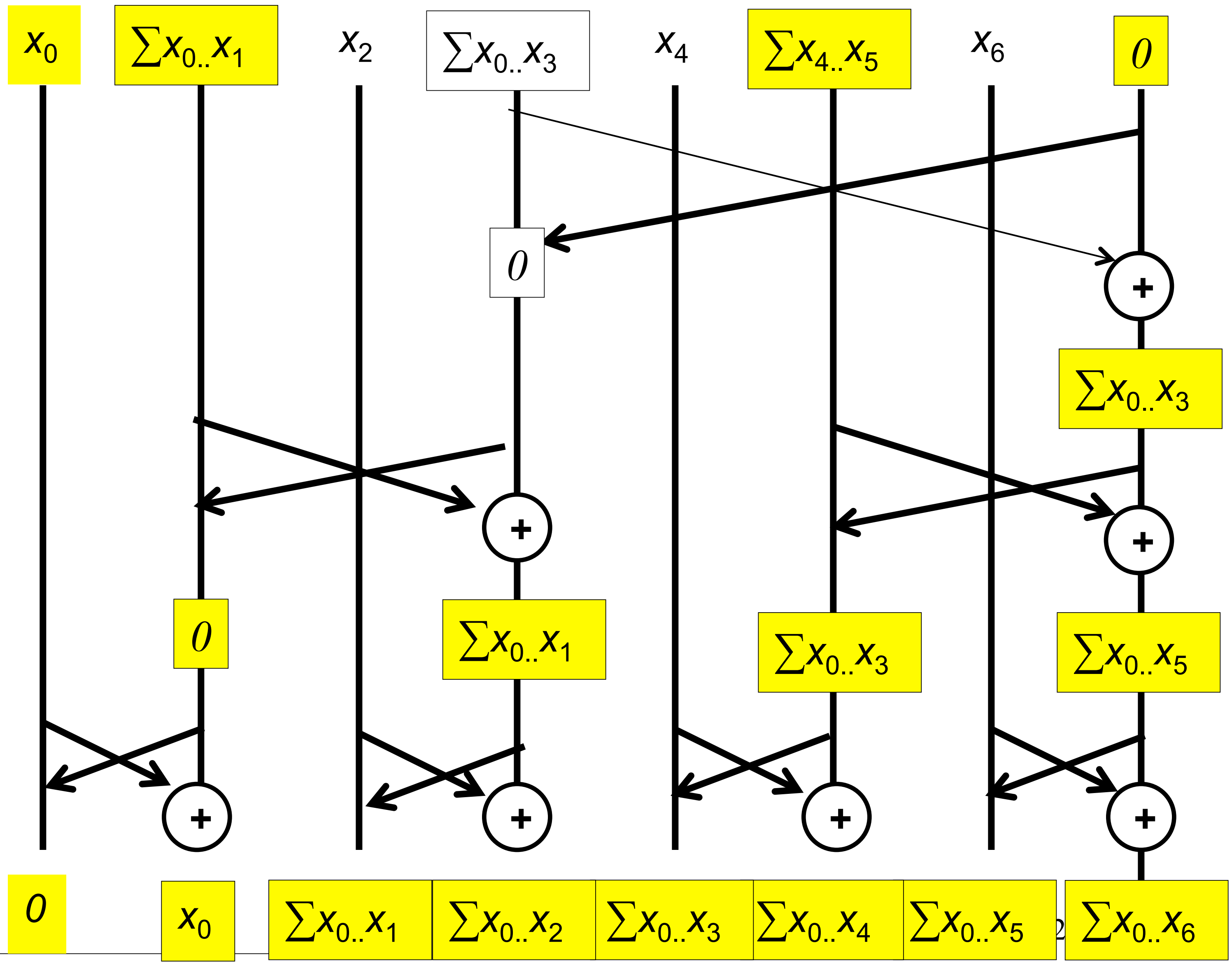
- Inclusive and Exclusive scans can be easily derived from each other; it is a matter of convenience

	[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]

Exclusive Post Scan Step (Add-move Operation)



Exclusive Post Scan Step



Exclusive Post Scan Step

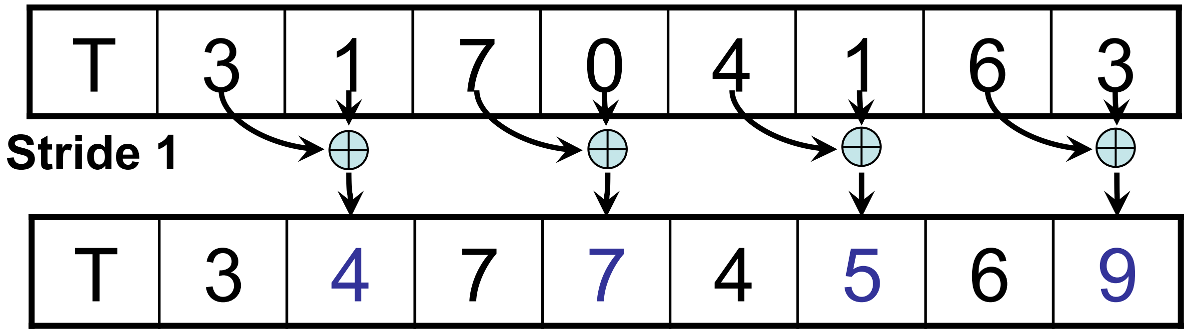
```
if (threadIdx.x==0) scan_array[2*blockDim.x-1] = 0;
int stride = BLOCK_SIZE;

while(stride > 0)
{
   int index = (threadIdx.x+1)*stride*2 - 1;
   if(index < 2* BLOCK_SIZE)
   {
      float temp = scan_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.)

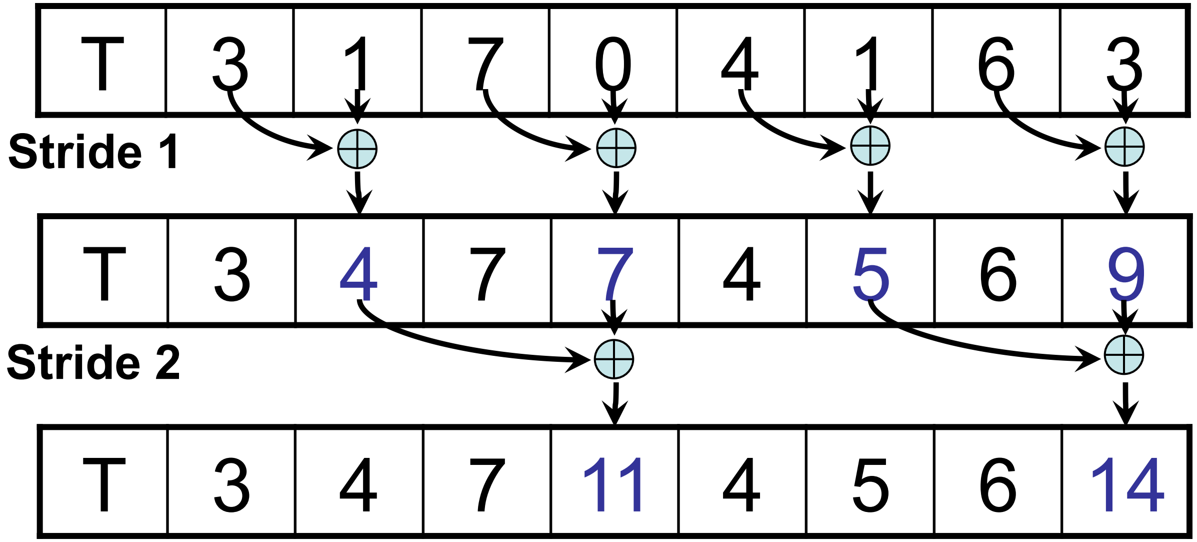


Iteration 1, *n*/2 threads

Each ⊕ corresponds to a single thread.

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

Reduction Step (cont.)

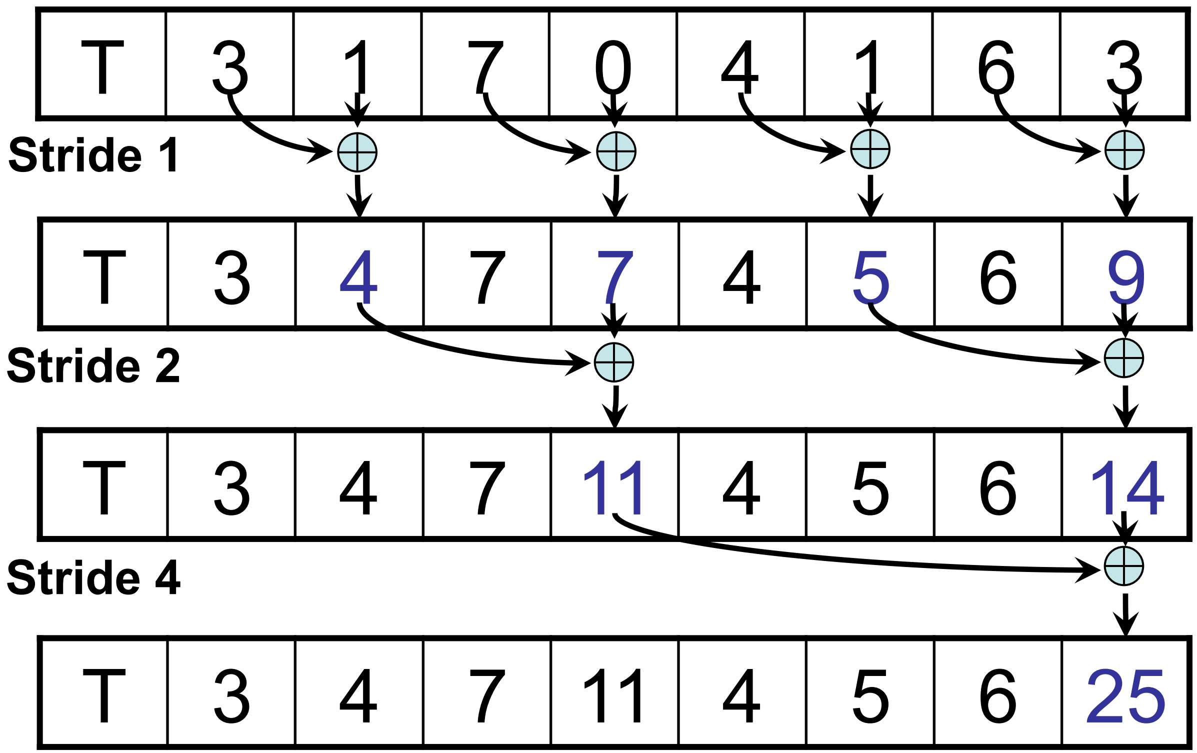


Iteration 2, *n*/4 threads

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

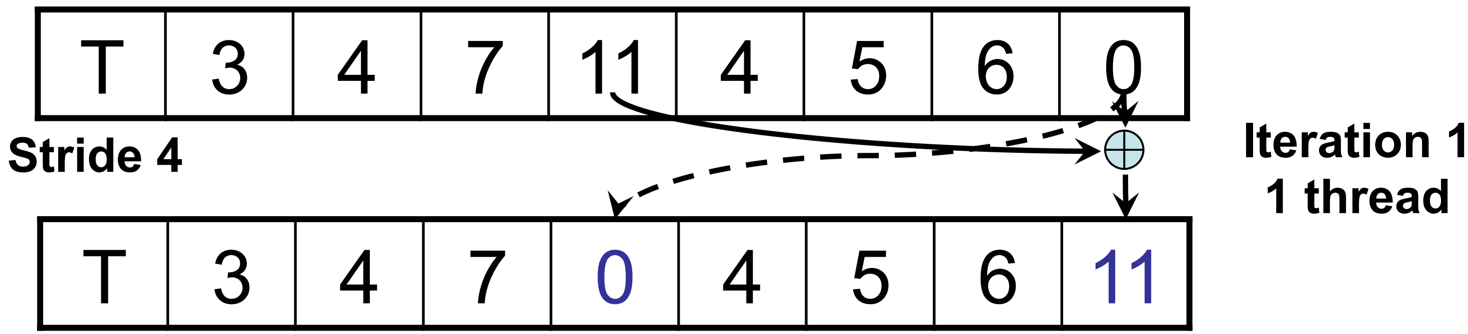
T	3	4	7	11	4	5	6	0

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

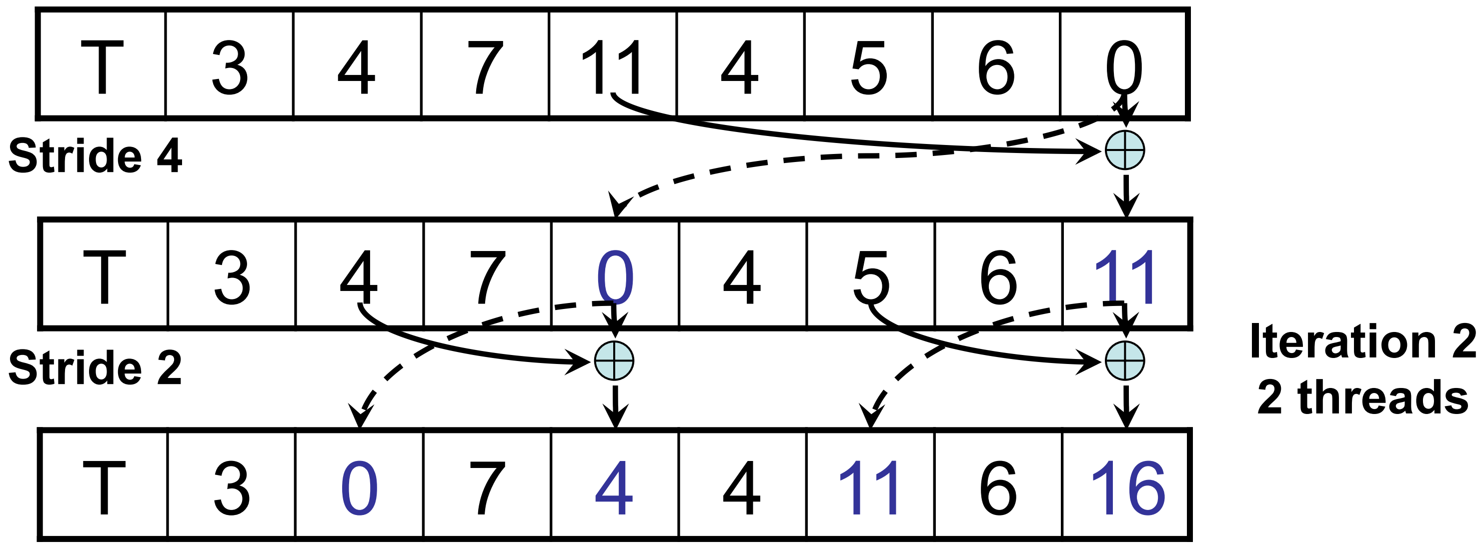
Post Scan Step from Partial Sums



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.

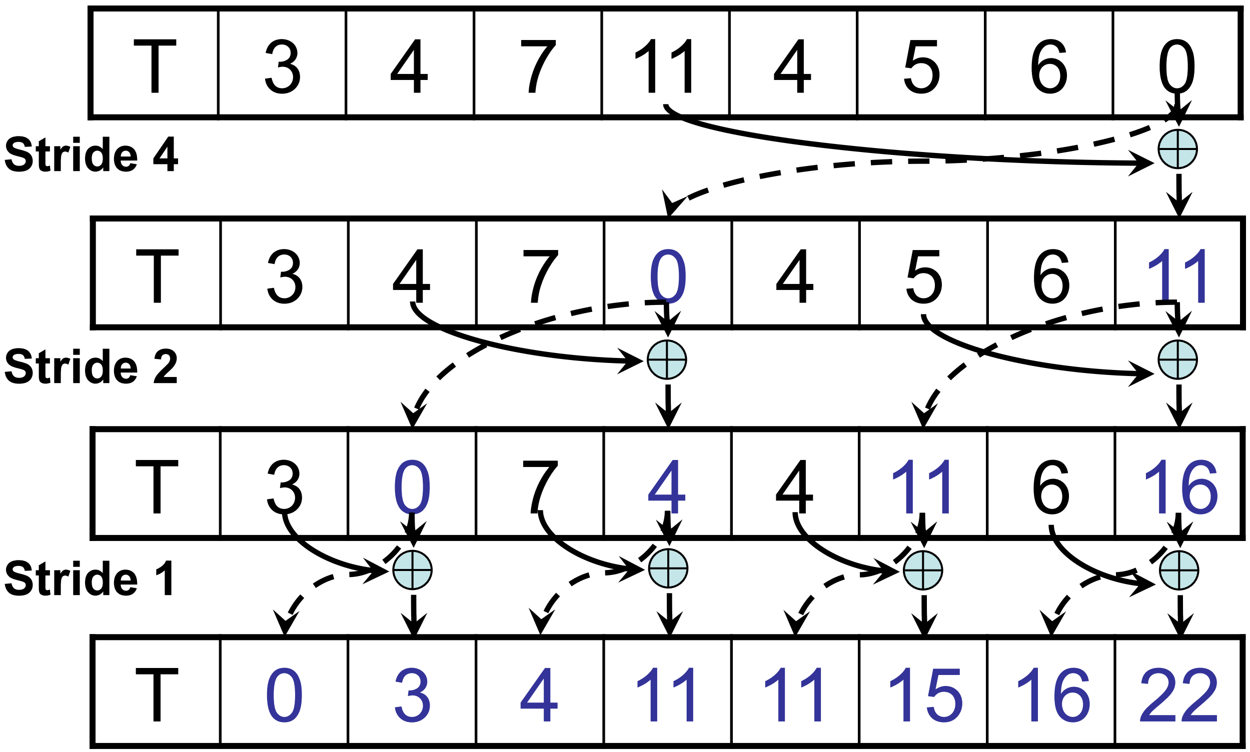
Post Scan Step from Partial Sums



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



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 that done in the efficient sequential algorithm**
  - The benefit of parallelism can easily overcome the 2X work when there is sufficient hardware

Working on Arbitrary Length Input

- **Build on the scan kernel that handles up to 2*blockDim.x elements**
- **Assign each section of 2*blockDim elements to a block**
- **Have each block write the sum of its section into a Sum array indexed by blockIdx.x**
- **Run parallel scan on the Sum array**
  - May need to break down Sum into multiple sections if it is too big for a block
- **Add the scanned Sum array values to the elements of corresponding sections**

Overall Flow of Complete Scan