Lecture 8:

Data Parallel Algorithms

Modern Parallel Computing

John Owens

EEC 289Q, UC Davis, Winter 2018

Basic Efficiency Rules

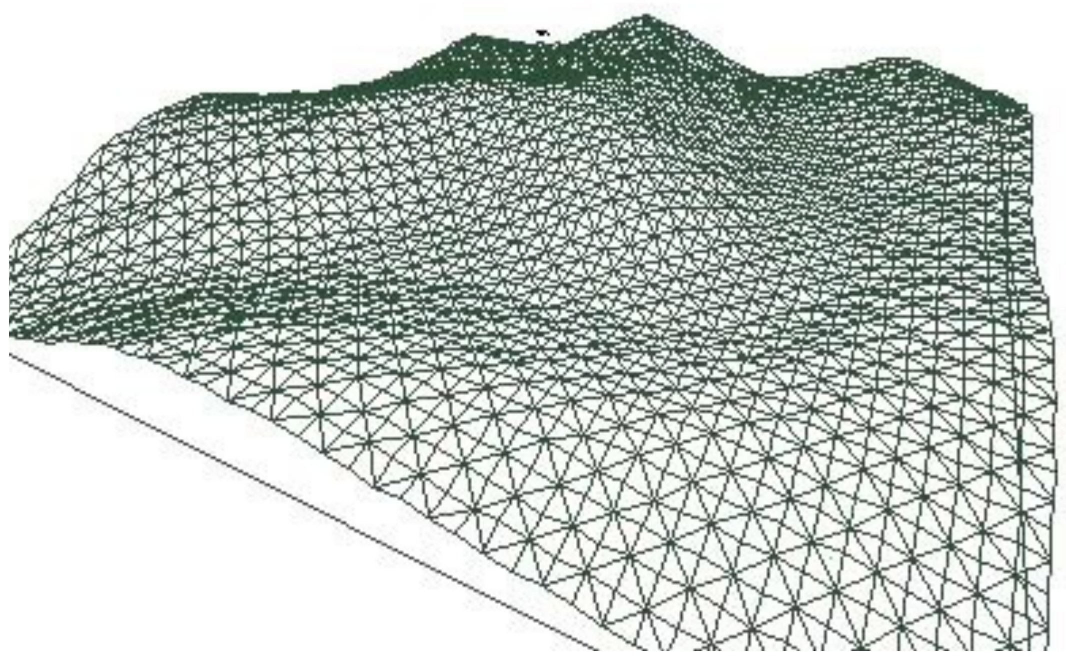
- Develop algorithms with a data parallel mindset
- Minimize divergence of execution within blocks
- Maximize locality of global memory accesses
  - "Coalescing"
- Exploit per-block shared memory as scratchpad
- Expose enough parallelism

Data-Parallel Algorithms

- Efficient algorithms require efficient building blocks
- Five data-parallel building blocks
  - Map
  - Gather & Scatter
  - Reduce
  - Scan
  - Sort
- Concentrate today on the algorithms
- Concentrate next lecture on the implementations

Sample Motivating Application

- How bumpy is a surface that we represent as a grid of samples?

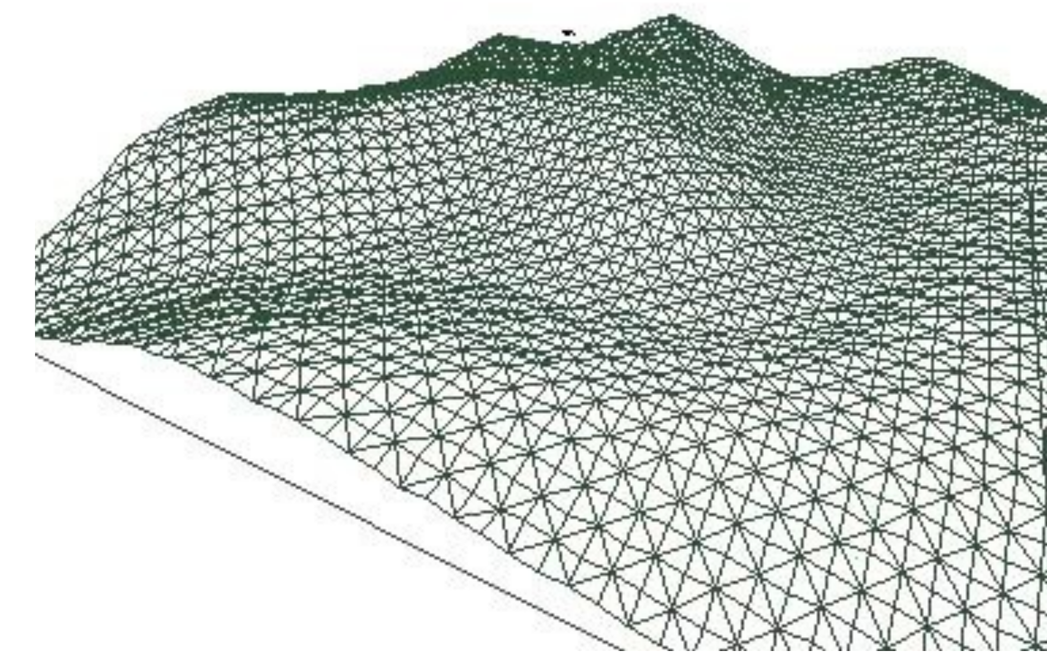


- Algorithm:
  - Loop over all elements
  - At each element, compare the value of that element to the average of its neighbors ("difference"). Square that difference.
  - Now sum up all those differences.
    - But we don't want to sum all the diffs that are 0.
    - So only sum up the non-zero differences.
  - This is a fake application—don't take it too seriously.

Picture courtesy http://www.artifice.com

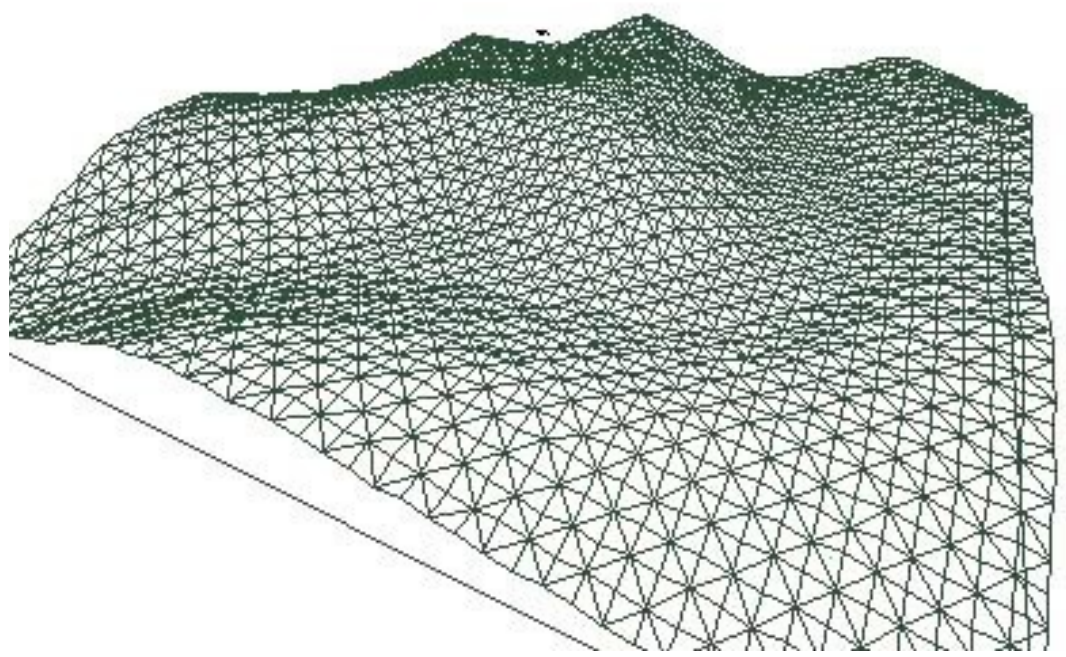
Sample Motivating Application

```
for all samples:

   neighbors[x,y] =
     0.25 * (  value[x-1,y] +
               value[x+1,y] +
               value[x,y+1] +
               value[x,y-1] ) )

   diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

   result += diff

return result
```



Sample Motivating Application

```
for all samples:

   neighbors[x,y] =
     0.25 * (  value[x-1,y] +
               value[x+1,y] +
               value[x,y+1] +
               value[x,y-1] ) )

   diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

   result += diff

return result
```

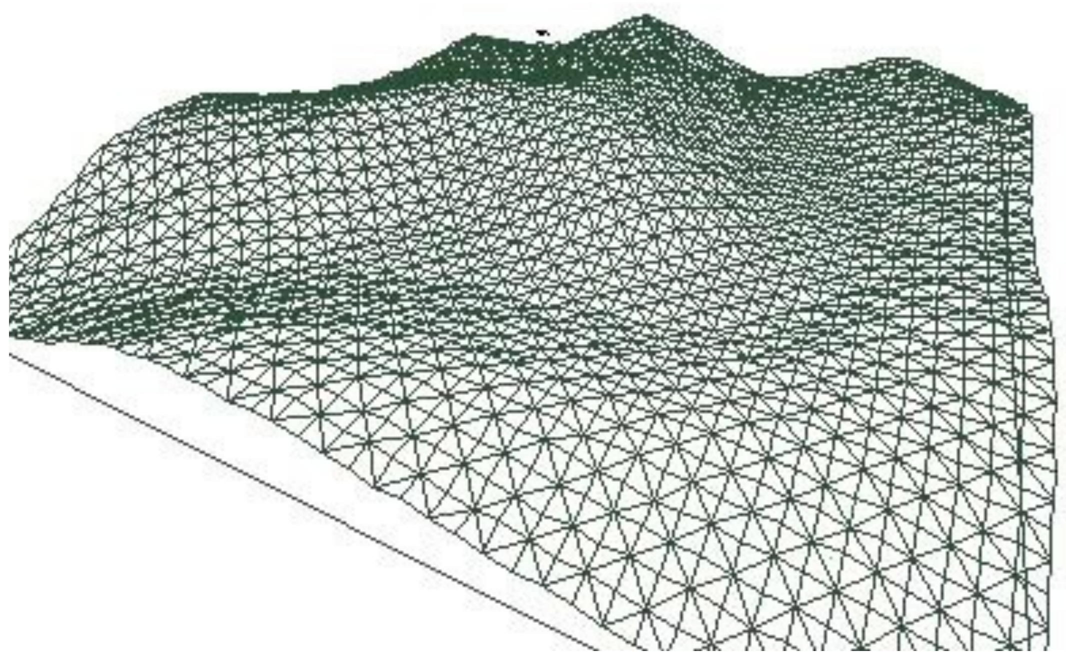


The Map Operation

- Given:
  - Array or stream of data elements A
  - Function f(x)
- map(A, f) = applies f(x) to all $a_i \in A$
- How does this map to a data-parallel processor?

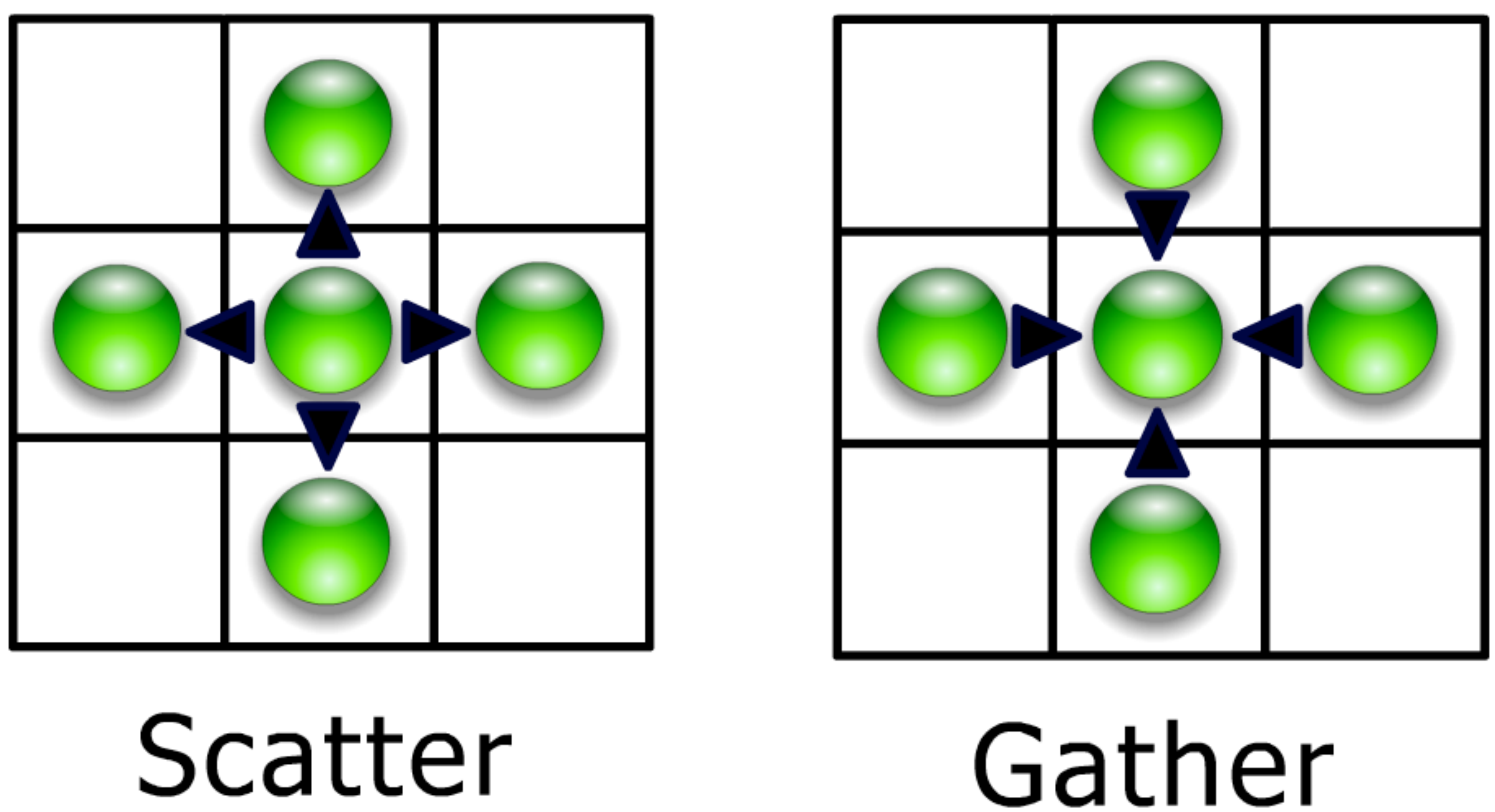
Sample Motivating Application

```
for all samples:

   neighbors[x,y] =
     0.25 * ( value[x-1,y] +
              value[x+1,y] +
              value[x,y+1] +
              value[x,y-1] ) )

   diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

   result += diff

return result
```



Scatter vs. Gather

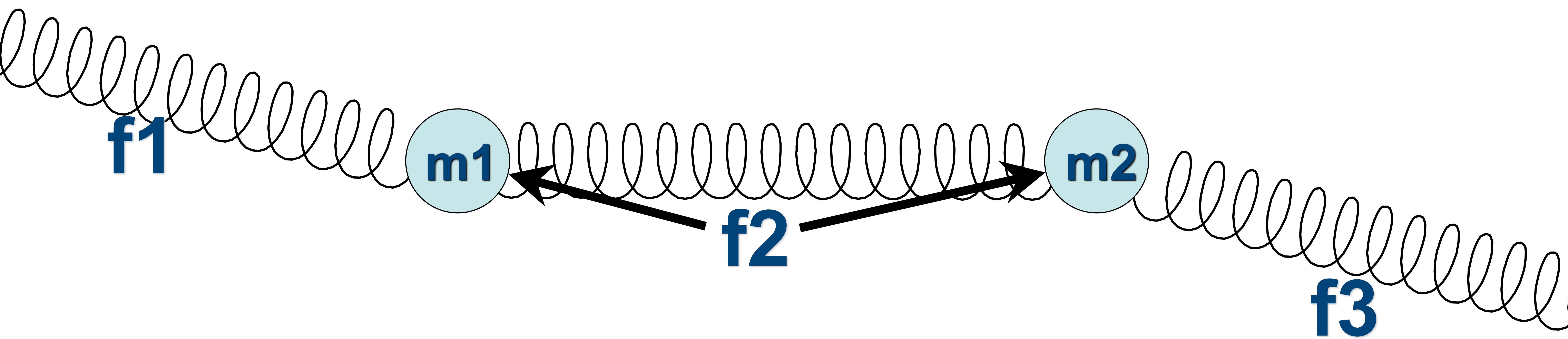
- **Gather:** `p = a[i]`
- **Scatter:** `a[i] = p`
- **How does this map to a data-parallel processor?**



Scatter Techniques

- Convert to Gather

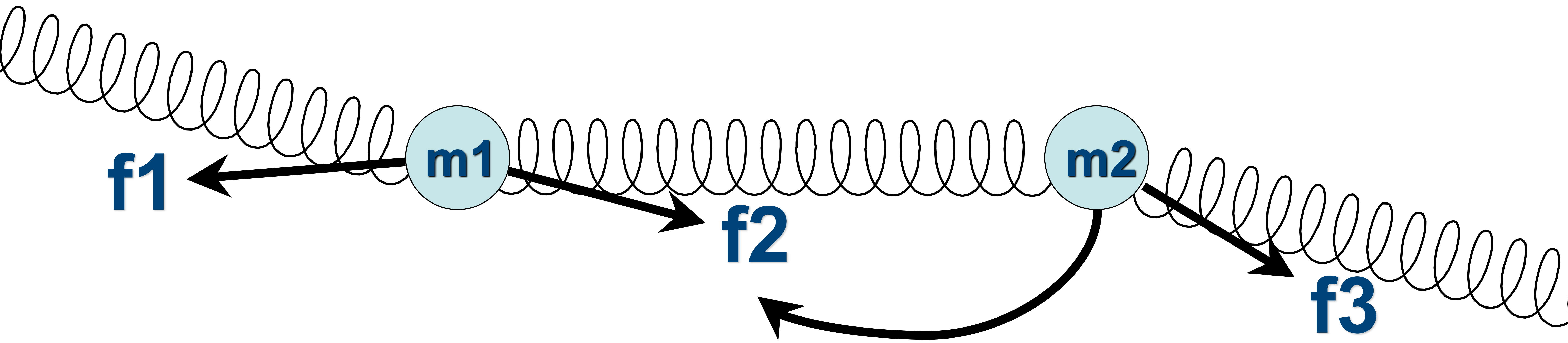
```
for each spring
    f = computed force
    mass_force[left]  += f;
    mass_force[right] -= f;
```



Scatter Techniques

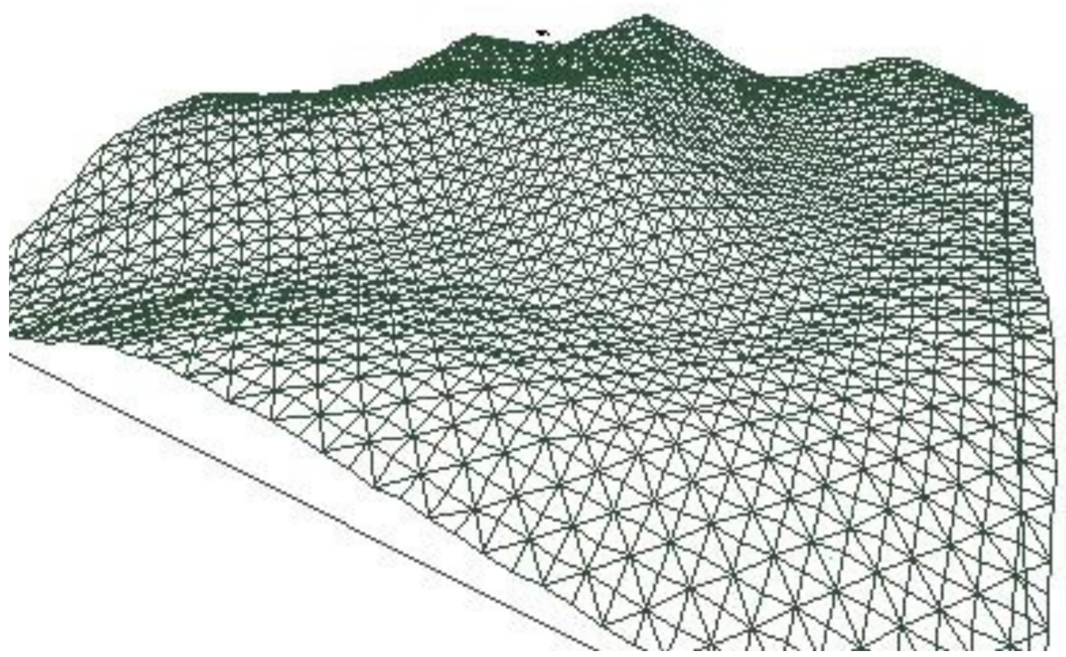
- Convert to Gather

```
for each spring
    f = computed force
for each mass
    mass_force = f[left] -
                 f[right];
```



Sample Motivating Application

```
for all samples:

   neighbors[x,y] =
     0.25 * (  value[x-1,y] +
               value[x+1,y] +
               value[x,y+1] +
               value[x,y-1] ) )

   diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

   result += diff

return result
```



Parallel Reductions

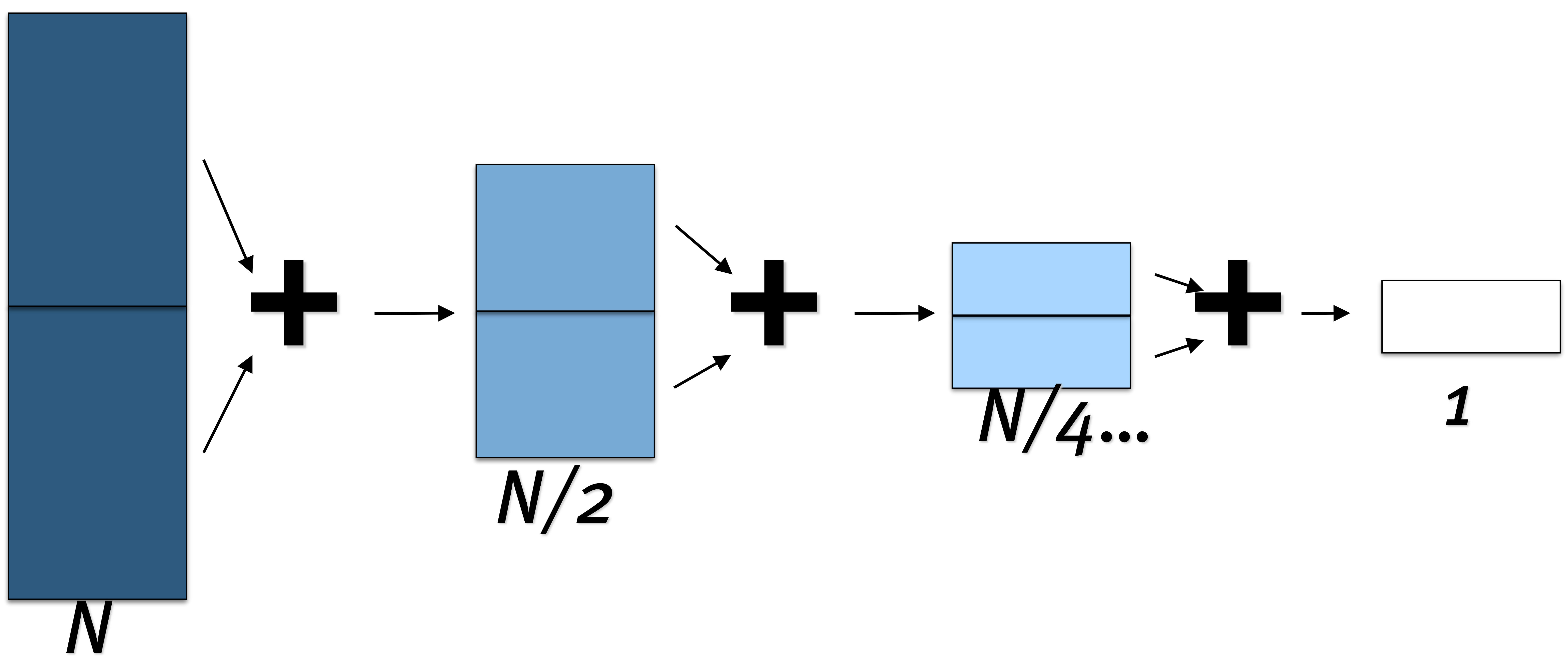
- **Given:**
  - Binary associative operator $\oplus$ with identity I
  - Ordered set $s = [a_0, a_1, \ldots, a_{n-1}]$ of n elements
- reduce($\oplus$, s) returns $a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$
- **Example:**
  reduce(+, [3 1 7 0 4 1 6 3]) = 25
- **Reductions common in parallel algorithms**
  - Common reduction operators are +, $\times$, min and max
  - Note floating point is only pseudo-associative

Efficiency

- **Work efficiency:**
  - **Total amount of work done over all processors**
- **Step efficiency:**
  - **Number of steps it takes to do that work**
- **With parallel processors, sometimes you're willing to do more work to reduce the number of steps**
- **Even better if you can reduce the amount of steps and still do the same amount of work**

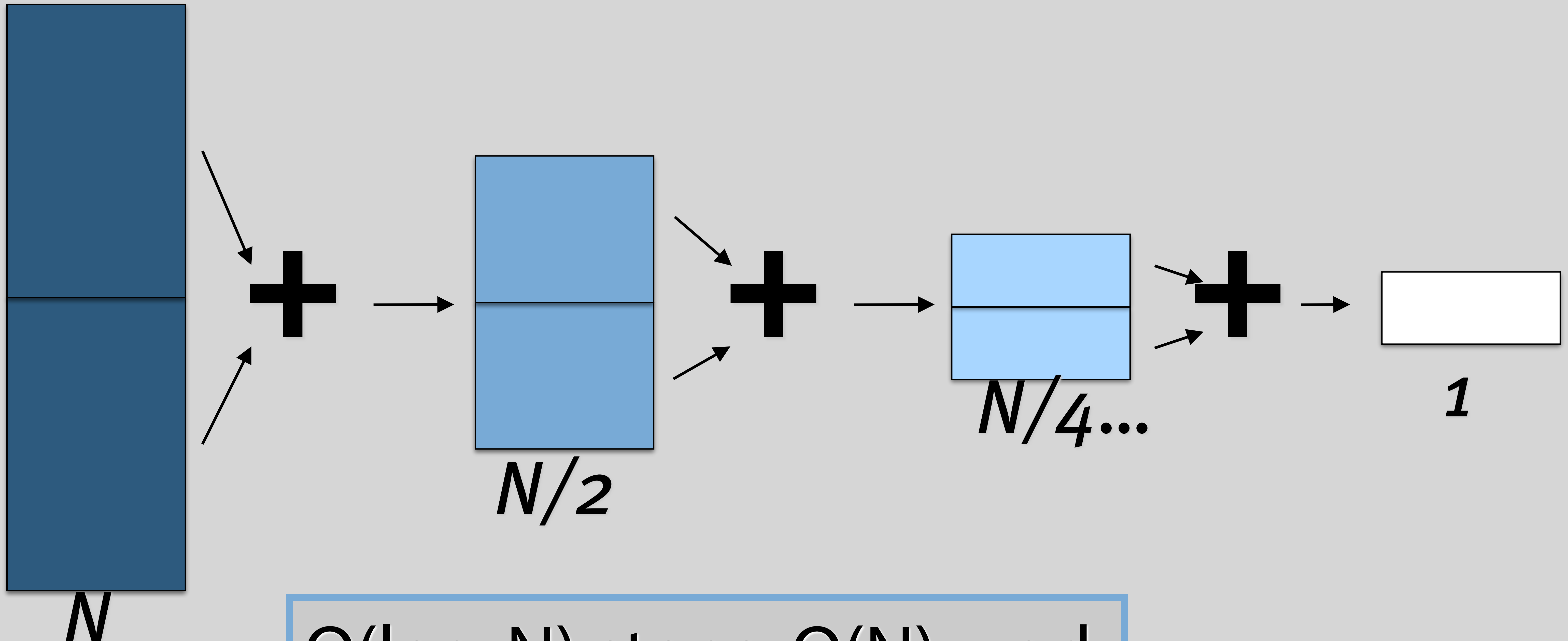
Parallel Reductions

- **1D parallel reduction:**
  - add two halves of domain together repeatedly...
  - … until we're left with a single row



Parallel Reductions

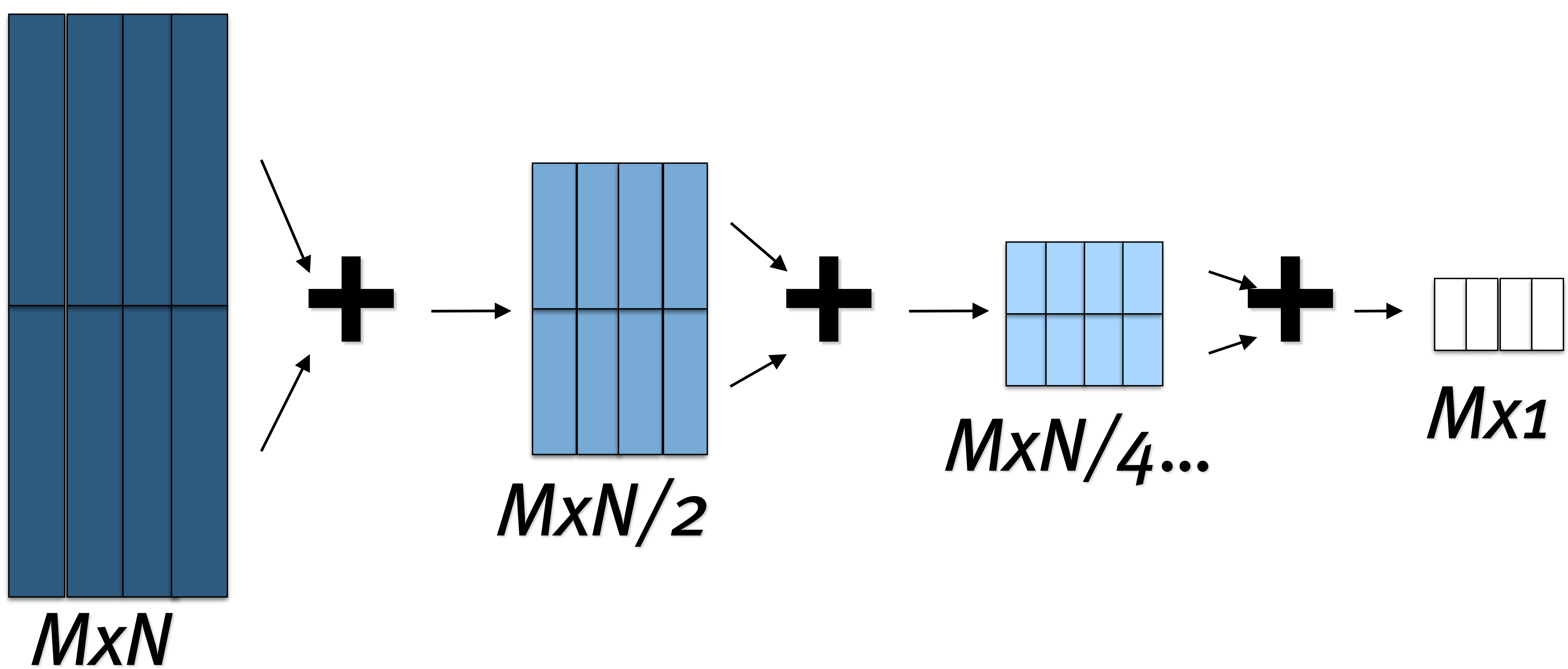
- 1D parallel reduction:
  - add two halves of domain together repeatedly...
  - . . . until we're left with a single row



O($\log_2$N) steps, O(N) work

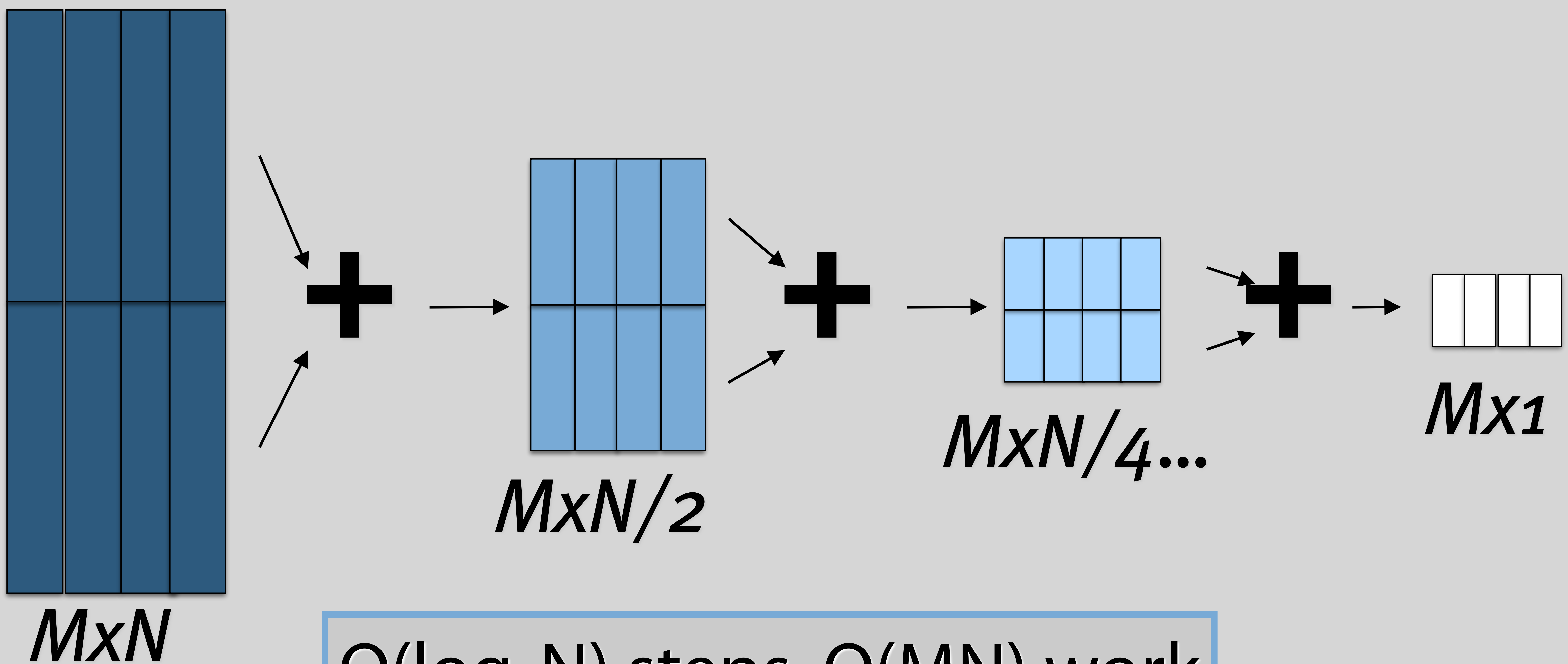
Multiple 1D Parallel Reductions

- Can run many reductions in parallel
- Use 2D grid and reduce one dimension



Multiple 1D Parallel Reductions

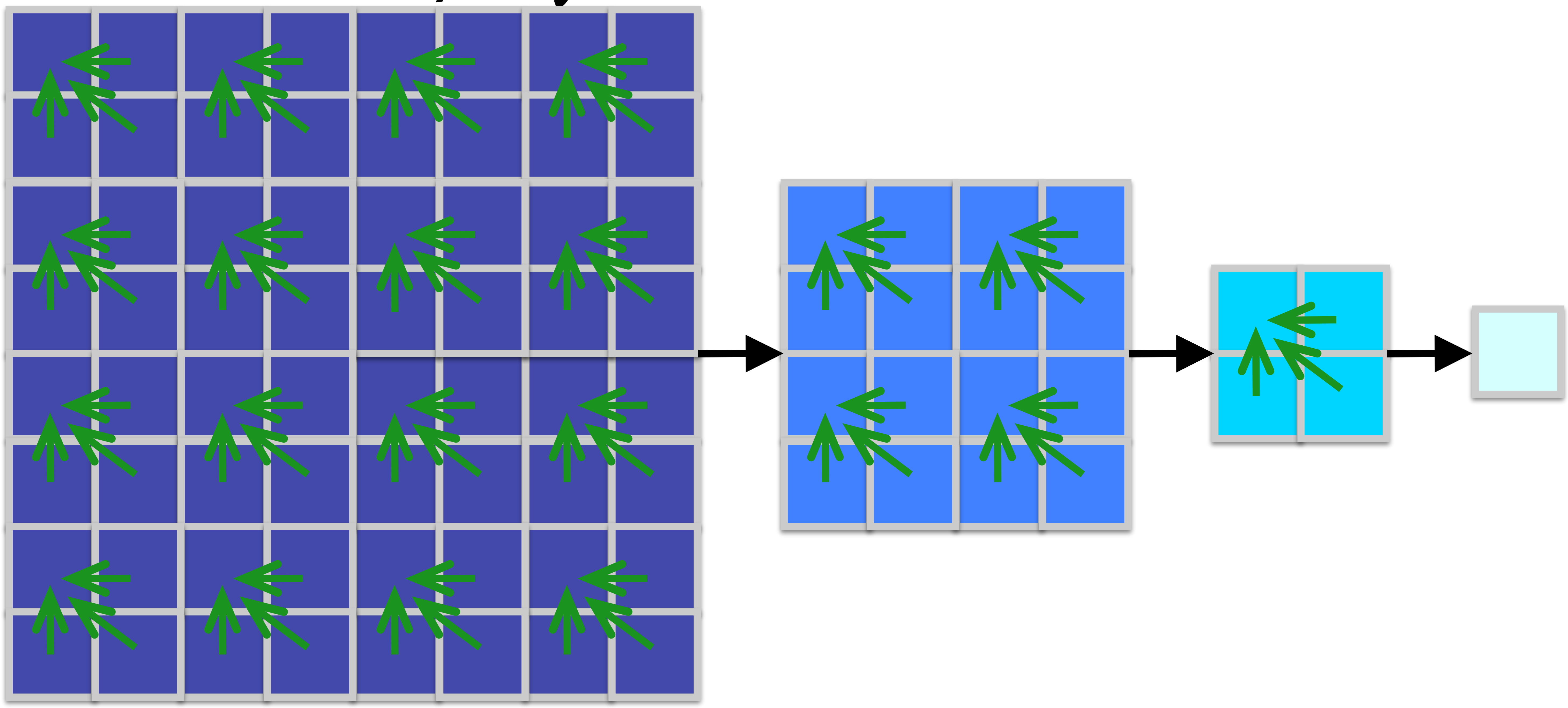
- Can run many reductions in parallel
- Use 2D grid and reduce one dimension



O($\log_2$N) steps, O(MN) work

2D reductions

- Like 1D reduction, only reduce in both directions simultaneously



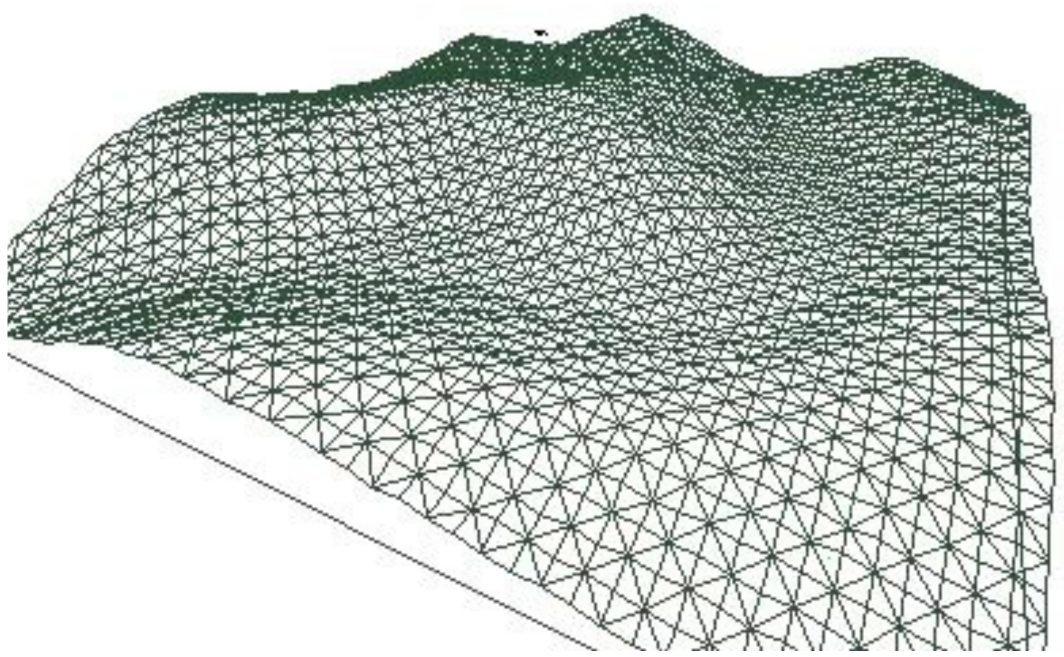
  - Note: can add more than 2x2 elements per step
    - Trade per-pixel work for step complexity
    - Best perf depends on specific hardware (cache, etc.)

Parallel Reduction Complexity

- log(n) parallel steps, each step S does n/2S independent ops
  - Step Complexity is O(log n)
- Performs $n/2 + n/4 + \ldots + 1 = n$-1 operations
  - Work Complexity is O(n)—it is work-efficient
    - i.e., does not perform more operations than a sequential algorithm
- With p threads physically in parallel (p processors), time complexity is O($n/p + \log n$)
  - This is "Brent's Theorem"
  - Compare to O(n) for sequential reduction

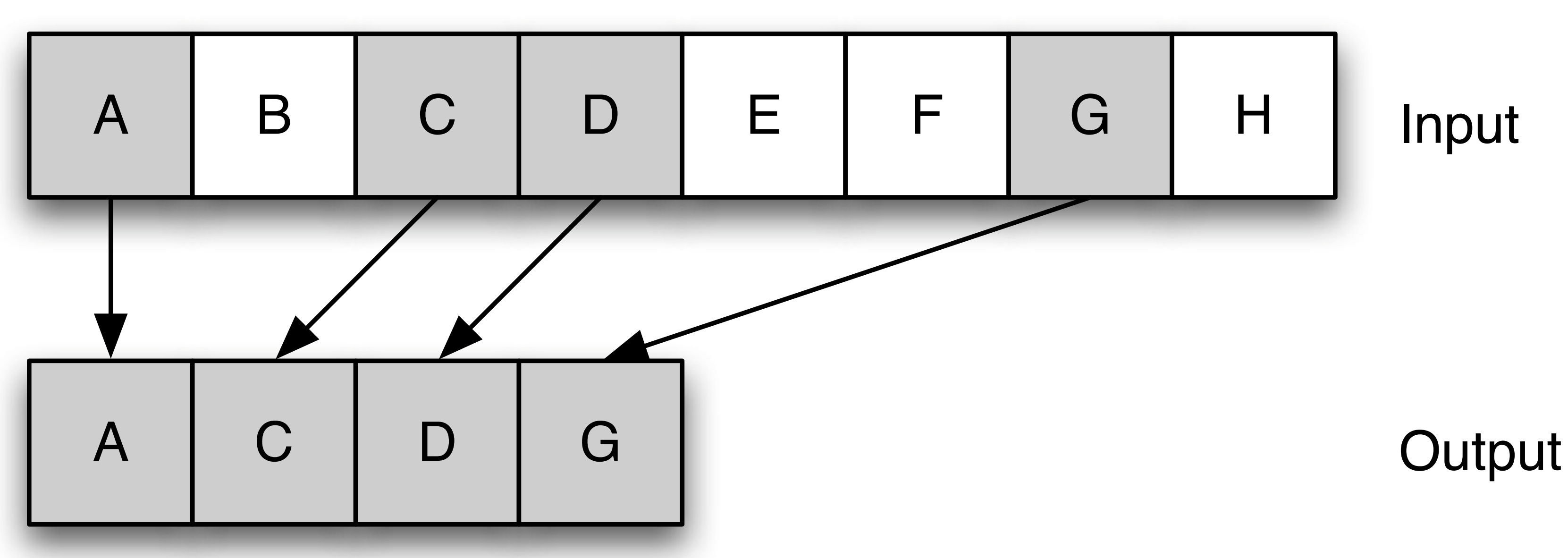
Sample Motivating Application

```
for all samples:

   neighbors[x,y] =
     0.25 * (  value[x-1,y] +
               value[x+1,y] +
               value[x,y+1] +
               value[x,y-1] ) )

   diff = (value[x,y] - neighbors[x,y])^2

result = 0

for all samples where diff != 0:

   result += diff

return result
```



Stream Compaction

- **Input: stream of 1s and 0s**
  [1 0 1 1 0 0 1 0]
- **Operation: "sum up all elements before you"**
- **Output: scatter addresses for "1" elements**
  [0 1 1 2 3 3 3 4]
- **Note scatter addresses for gray elements are packed!**

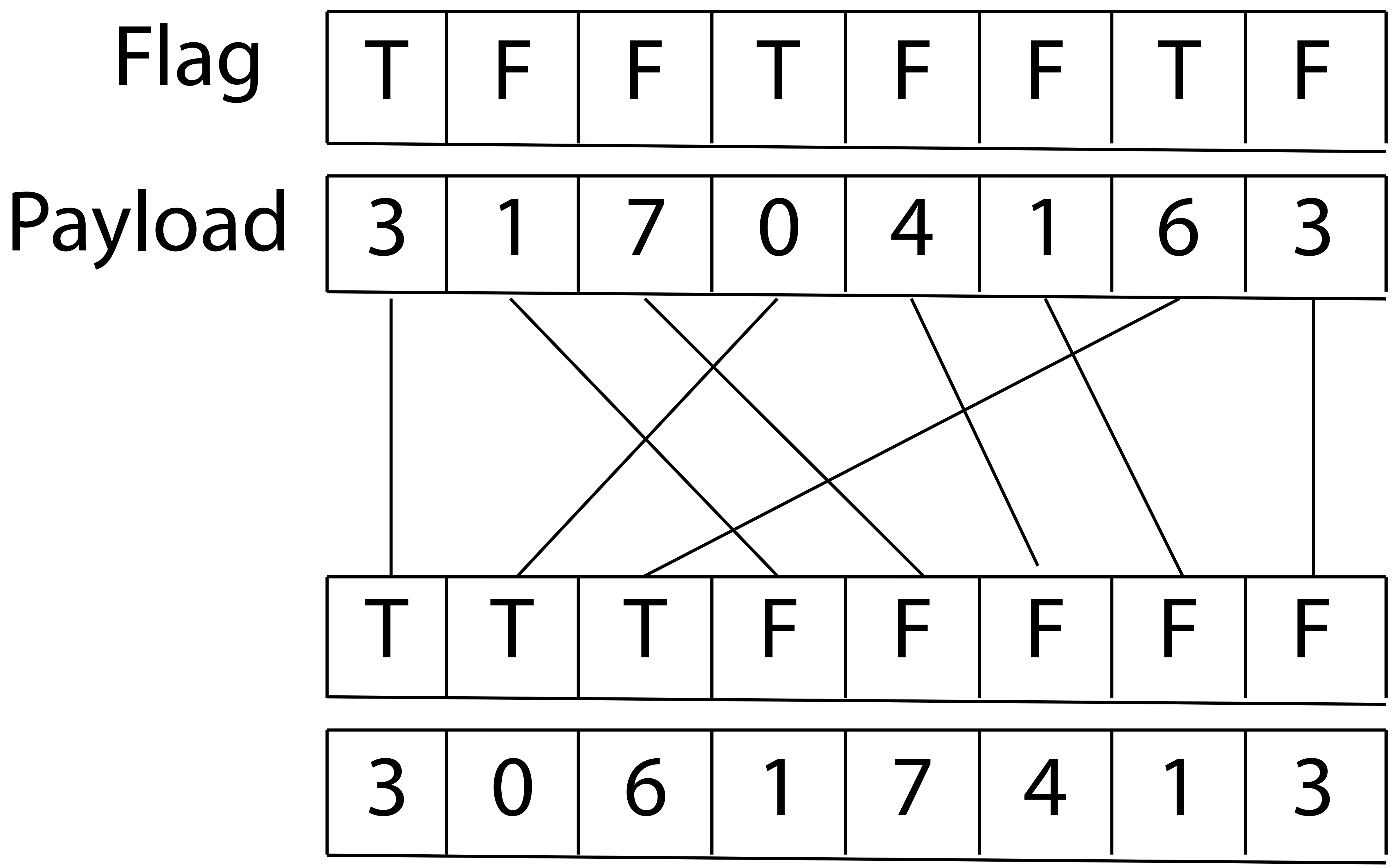


Common Situations in Parallel Computation

- Many parallel threads that need to partition data
  - Split
- Many parallel threads and variable output per thread
  - Compact / Expand / Allocate
- More complicated patterns than one-to-one or all-to-one
  - Instead all-to-all

Split Operation

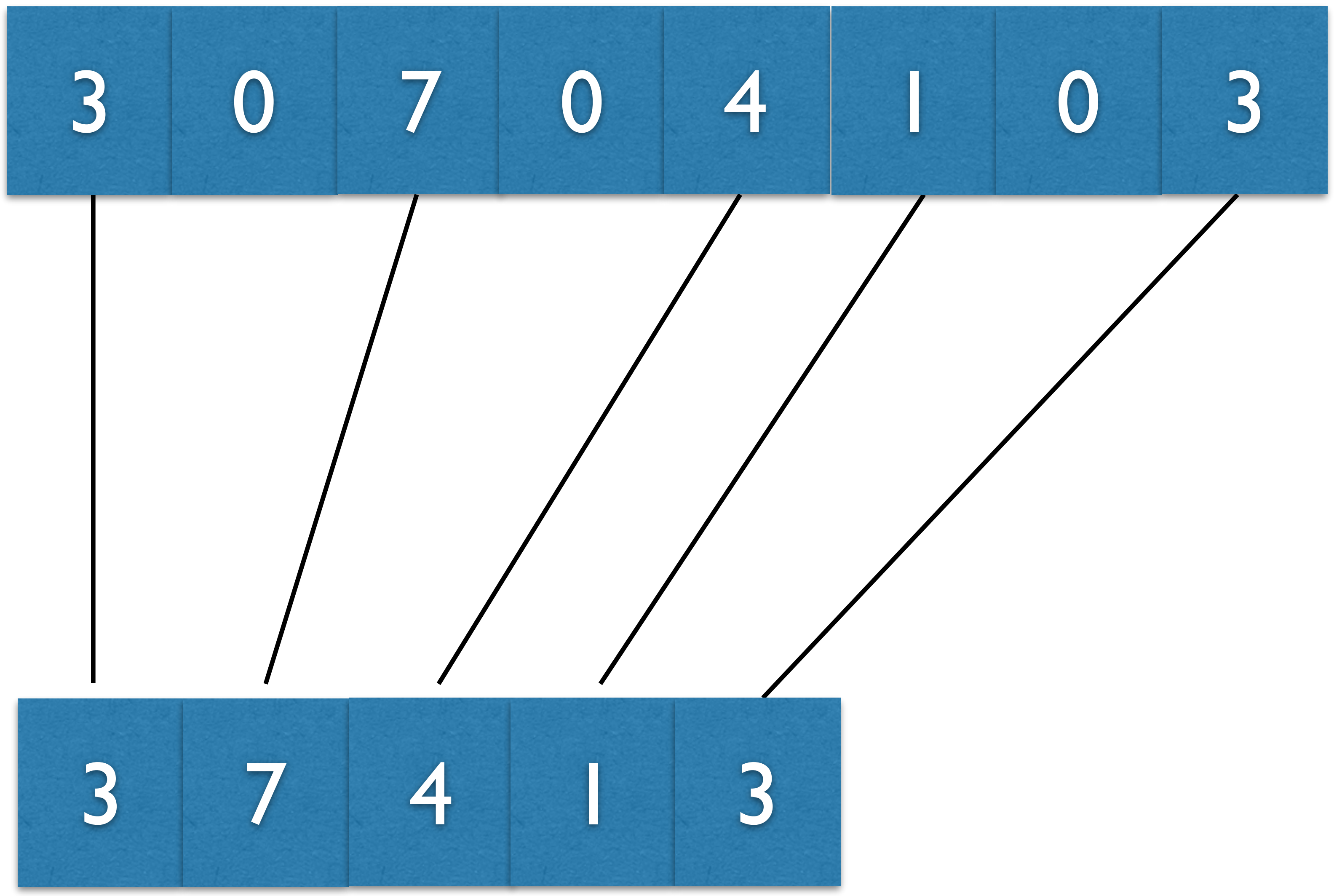
- Given an array of true and false elements (and payloads)



- Return an array with all true elements at the beginning
- Examples: sorting, building trees

Variable Output Per Thread: Compact

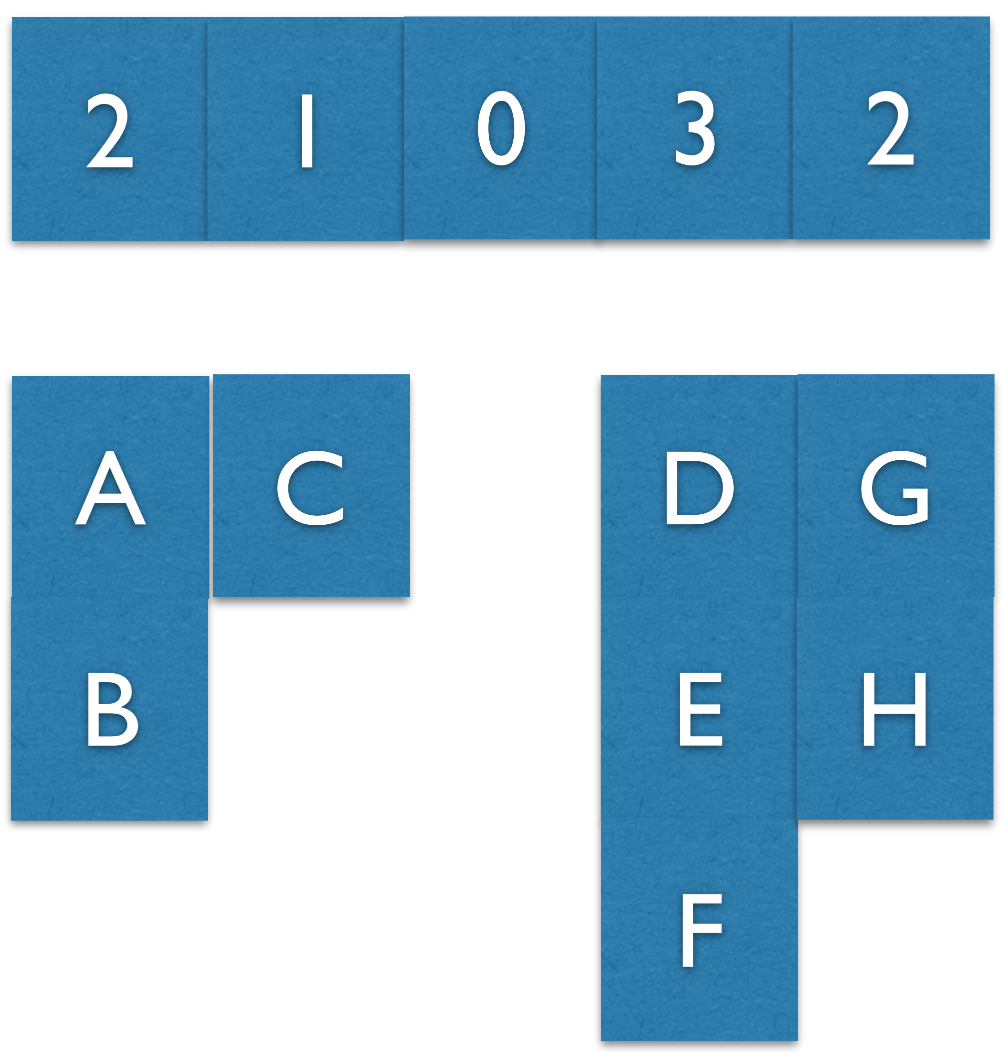
- Remove null elements



- Example: collision detection

Variable Output Per Thread

- Allocate Variable Storage Per Thread



- Examples: marching cubes, geometry generation

“Where do I write my output?”

- In all of these situations, each thread needs to answer that simple question
- The answer is:
- “That depends on how much the other threads need to write!”
  - In a serial processor, this is simple
- “Scan” is an efficient way to answer this question in parallel

Parallel Prefix Sum (Scan)

- Given an array $A = [a_0, a_1, \ldots, a_{n-1}]$
  and a binary associative operator $\oplus$ with identity I,

- $\text{scan}(A) = [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$

- Example: if $\oplus$ is addition, then scan on the set
  - [3 1 7 0 4 1 6 3]
- returns the set
  - [0 3 4 11 11 15 16 22]

Segmented Scan

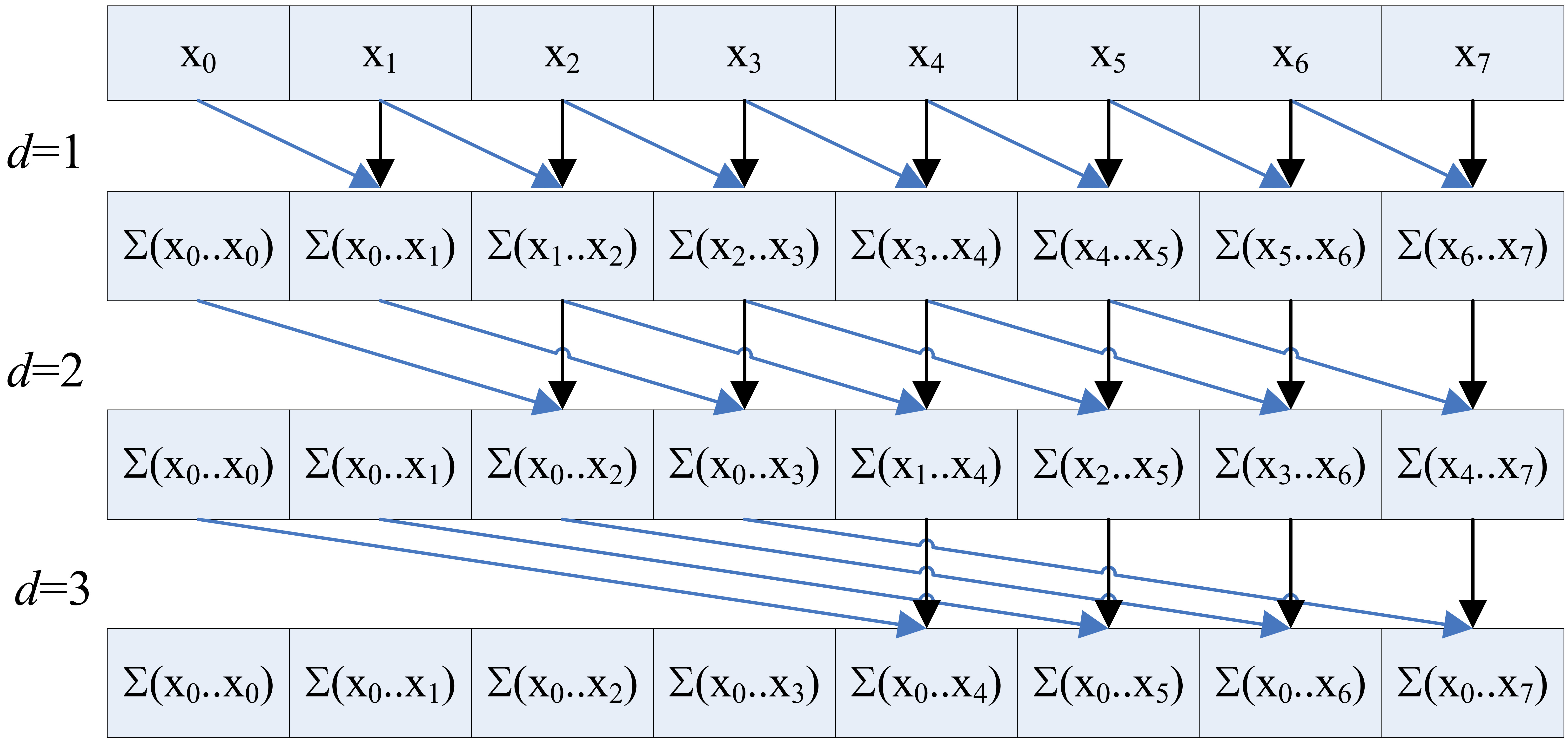
- Example: if $\oplus$ is addition, then scan on the set
  - [3 1 7 | 0 4 1 | 6 3]
- returns the set
  - [0 3 4 | 0 0 4 | 0 6]
- Same computational complexity as scan, but additionally have to keep track of segments (we use head flags to mark which elements are segment heads)
- Useful for *nested data parallelism* (quicksort)

Quicksort

```
[5 3 7 4 6]   # initial input
[5 5 5 5 5]   # distribute pivot across segment
[f f t f t]   # input > pivot?
[5 3 4][7 6]  # split-and-segment
[5 5 5][7 7]  # distribute pivot across segment
[t f f][t f]  # input >= pivot?
[3 4 5][6 7]  # split-and-segment, done!
```

$O(n \log n)$ Scan

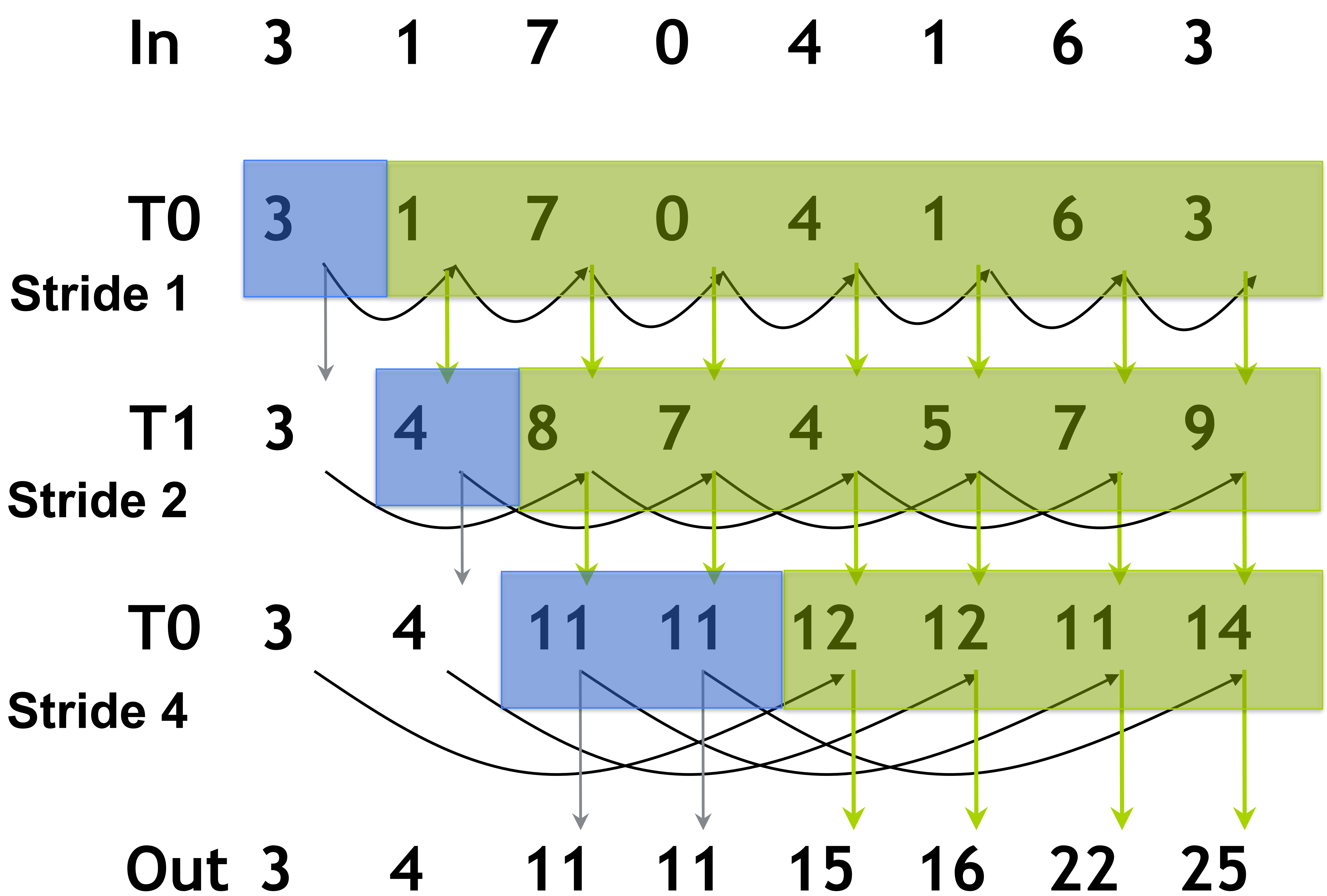
- **Step efficient (log *n* steps)**
- **Not work efficient (*n* log *n* work)**



O(n log n) Parallel Scan Algorithm

For i from 1 to log(n)-1:
Render a quad from 2^i to n. Fragment k computes

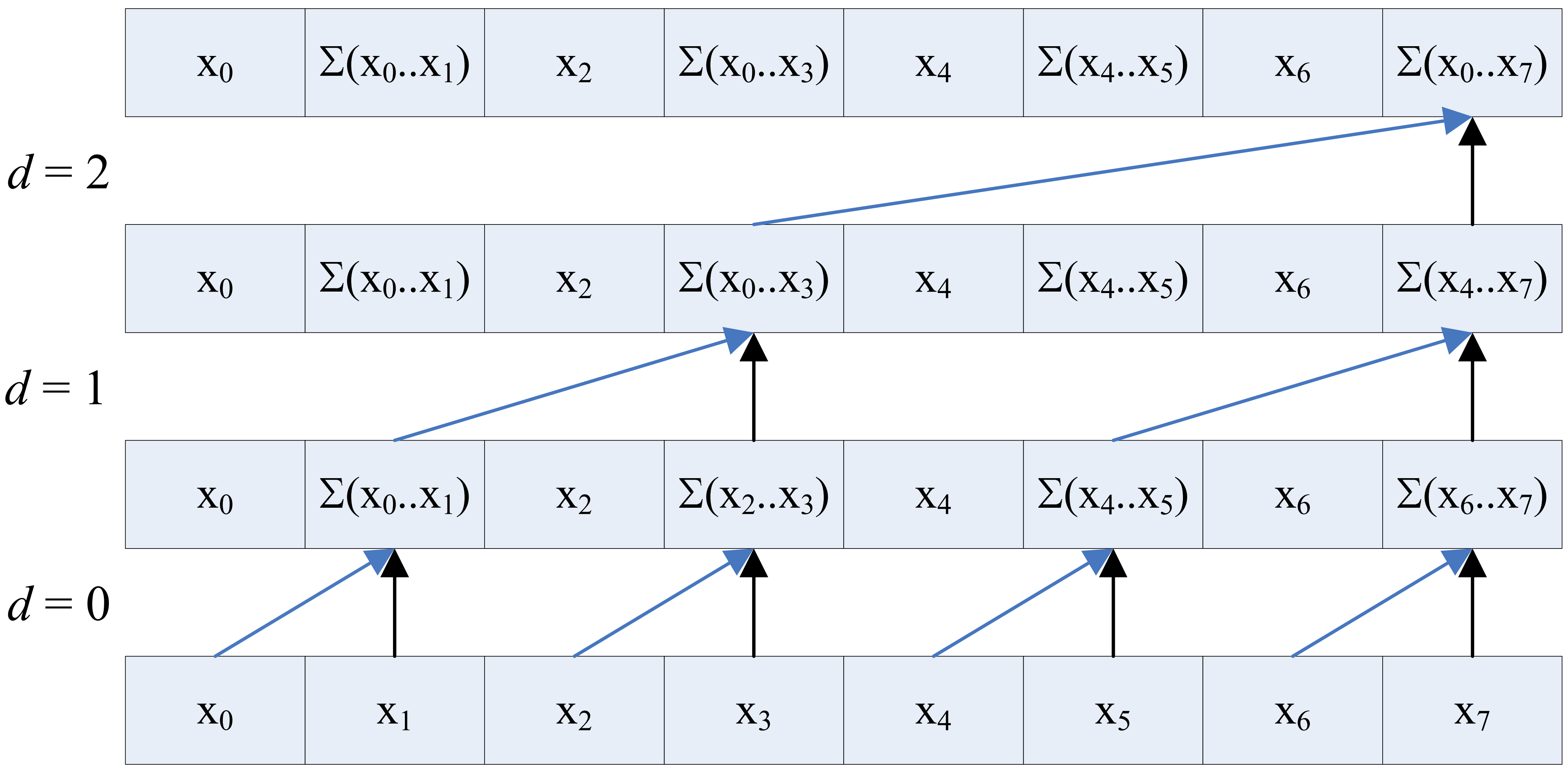
$$v_{out} = v[k] + v[k-2^i].$$



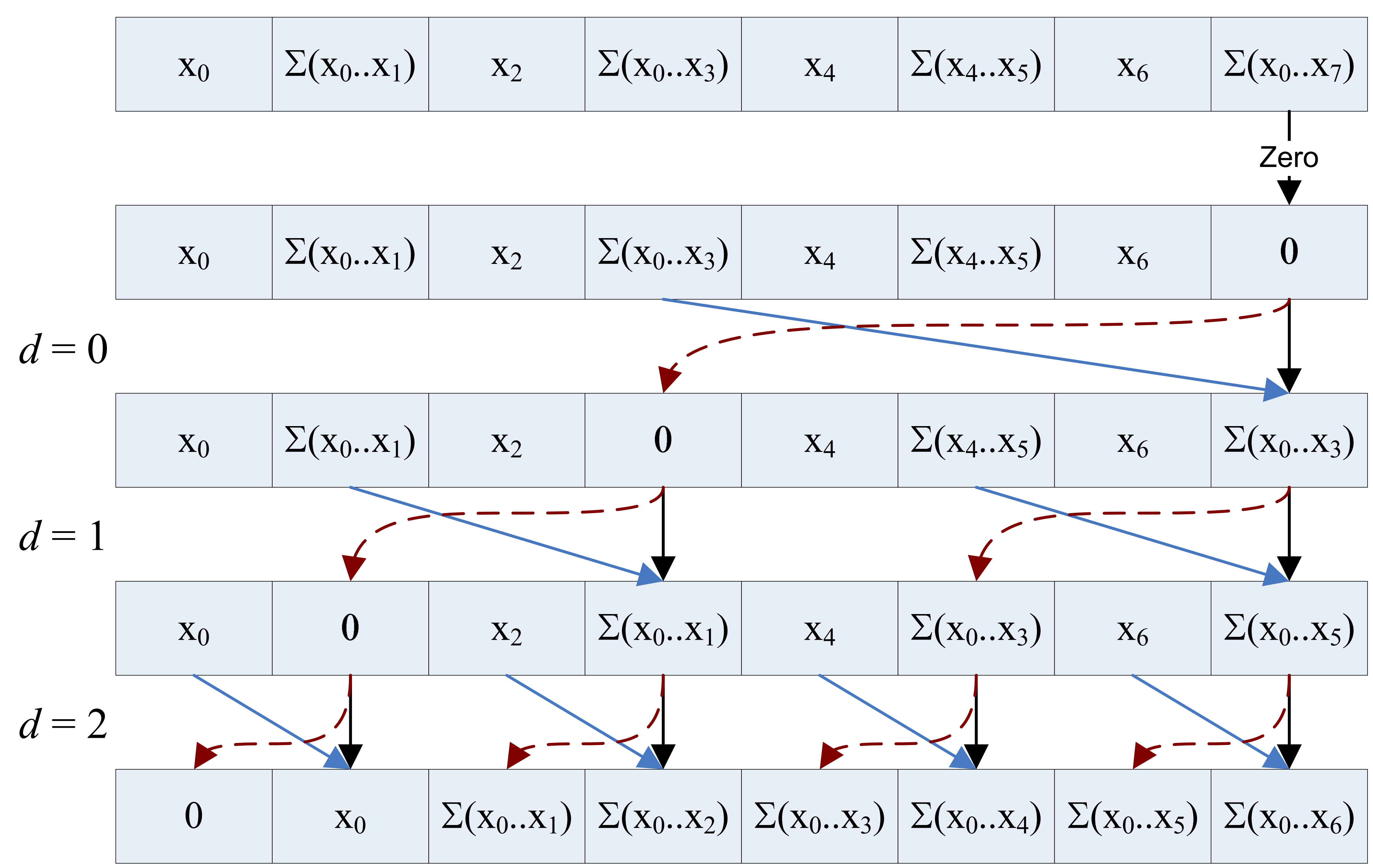
- Due to ping-pong, render a 2^{nd} quad from $2^{(i-1)}$ to 2^i with a simple pass-through shader

$$v_{out} = v_{in}.$$

O(*n*) Scan



- **Not step efficient (2 log *n* steps)**
- **Work efficient (*O*(*n*) work)**



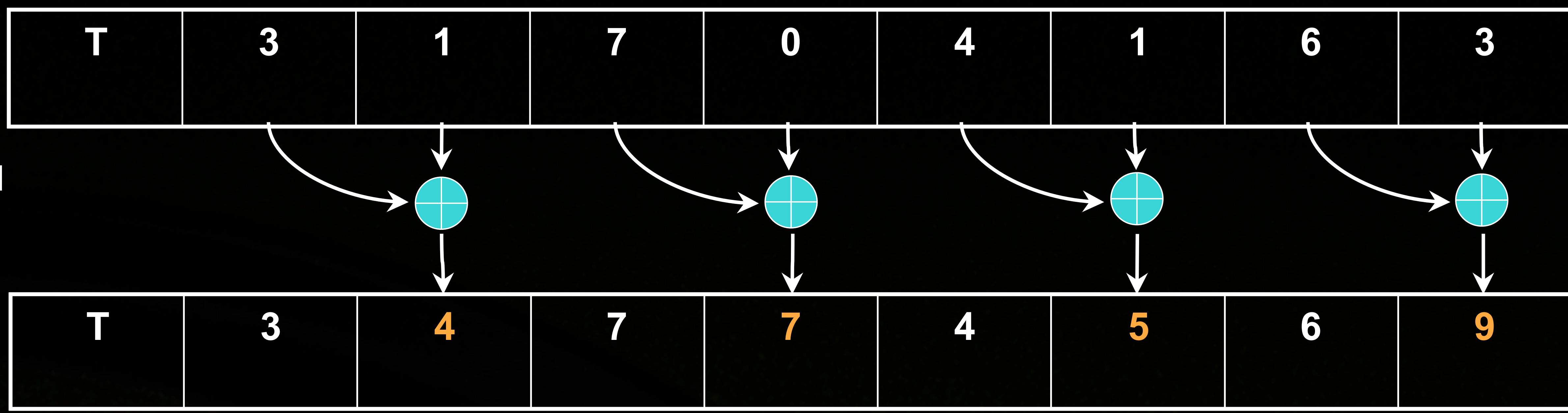
Build the Sum Tree

T	3	1	7	0	4	1	6	3

Assume array is already in shared memory

Build the Sum Tree

Stride 1

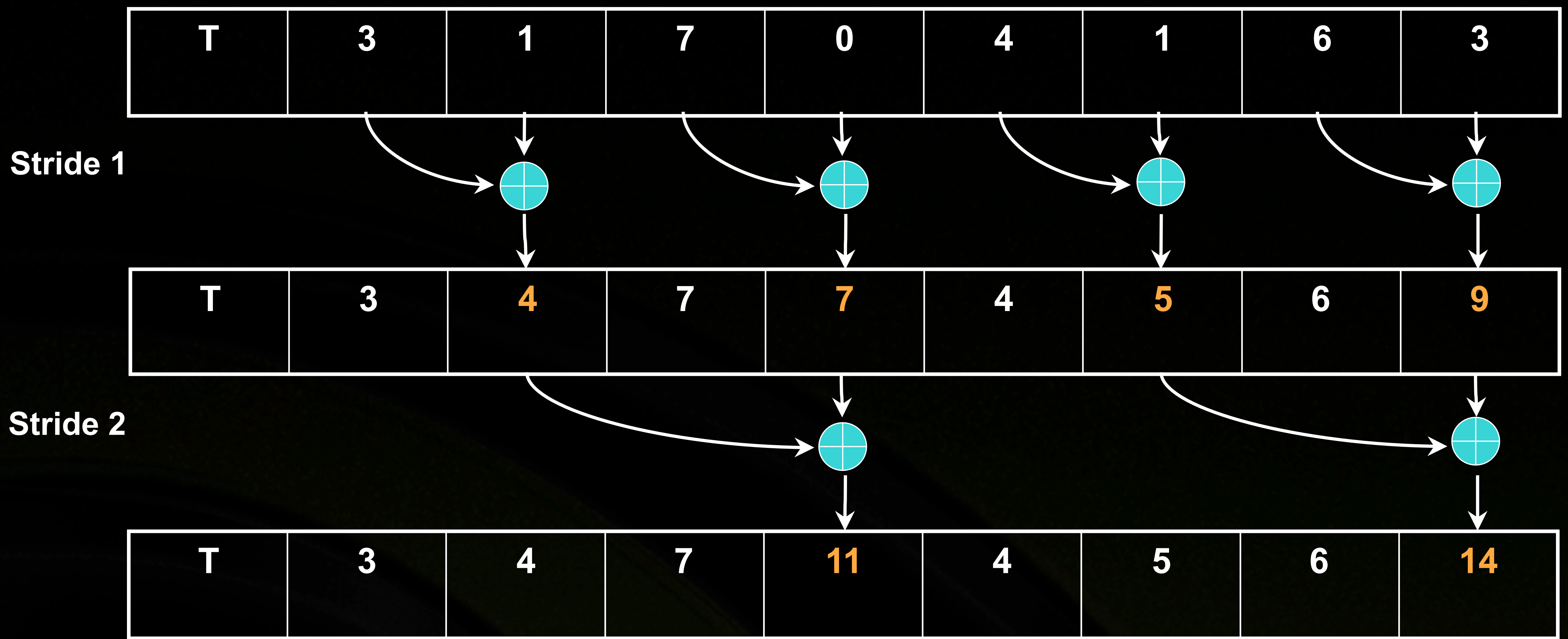


Iteration 1, *n*/2 threads

Each ⊕ corresponds to a single thread.

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

Build the Sum Tree

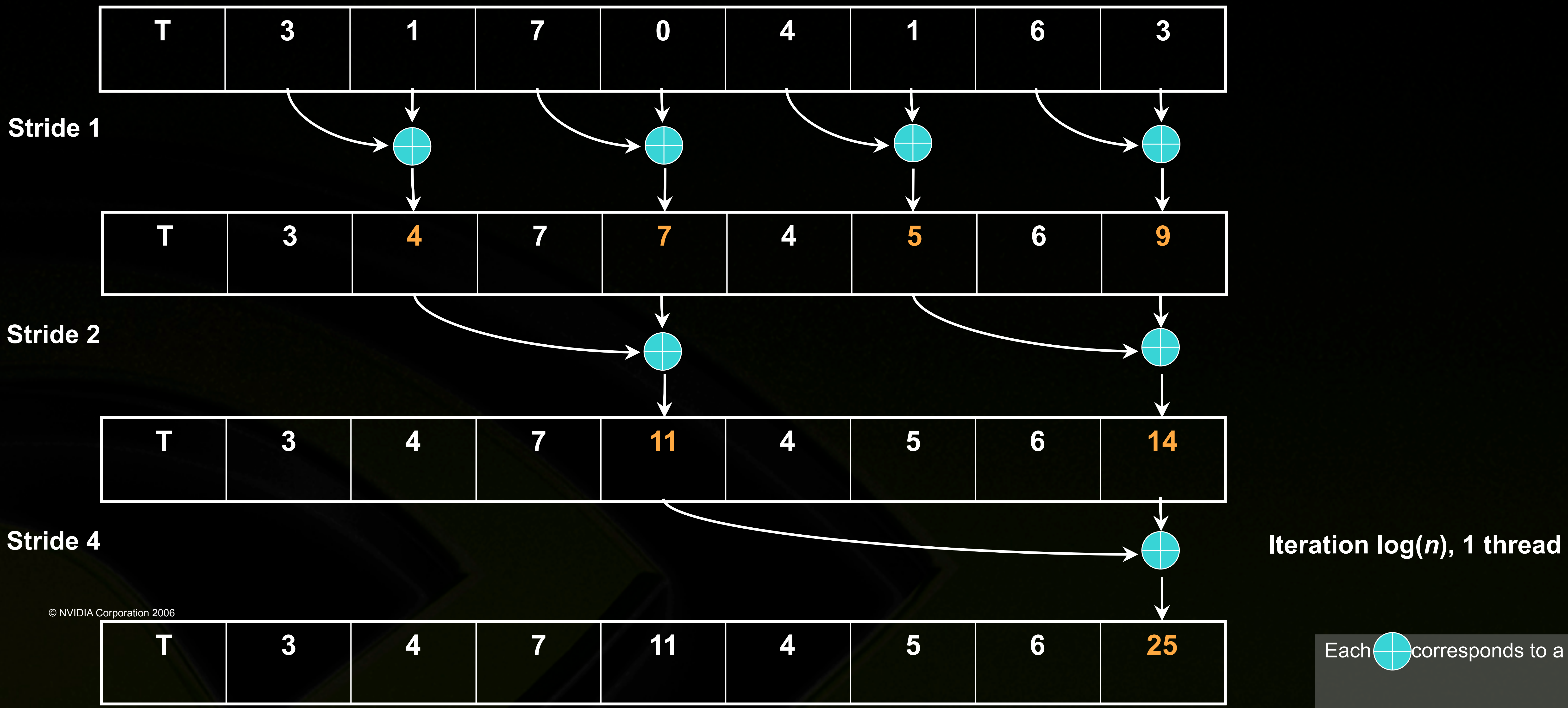


Iteration 2, *n*/4 threads

Each ⊕ corresponds to a single thread.

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

Build the Sum Tree



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

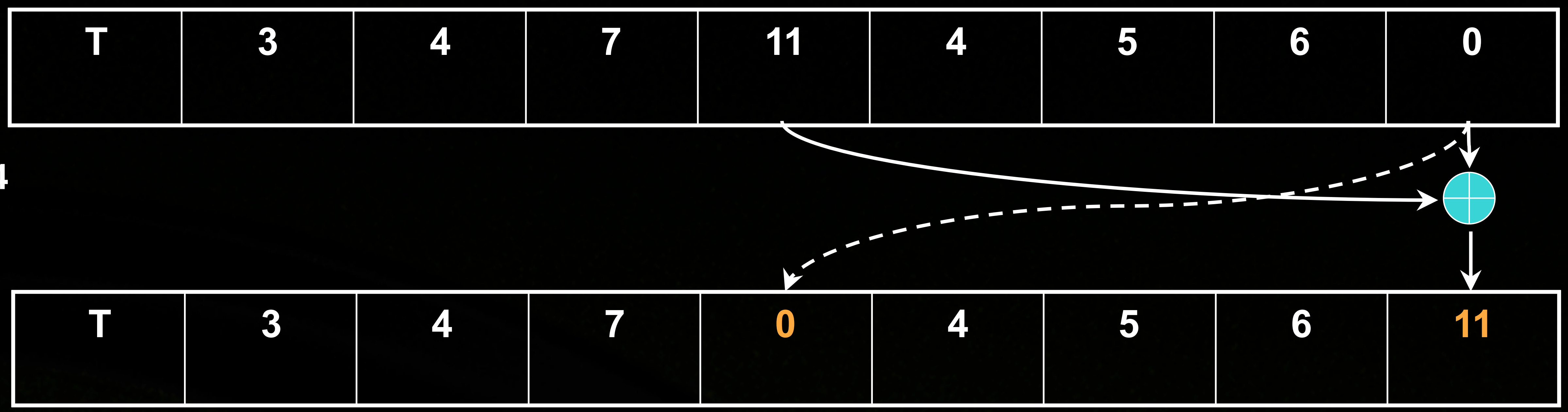
We now have an array of partial sums. Since this is an exclusive scan, set the last element to zero. It will propagate back to the first element.

Build Scan From Partial Sums

T	3	4	7	11	4	5	6	0

Build Scan From Partial Sums

Stride 4

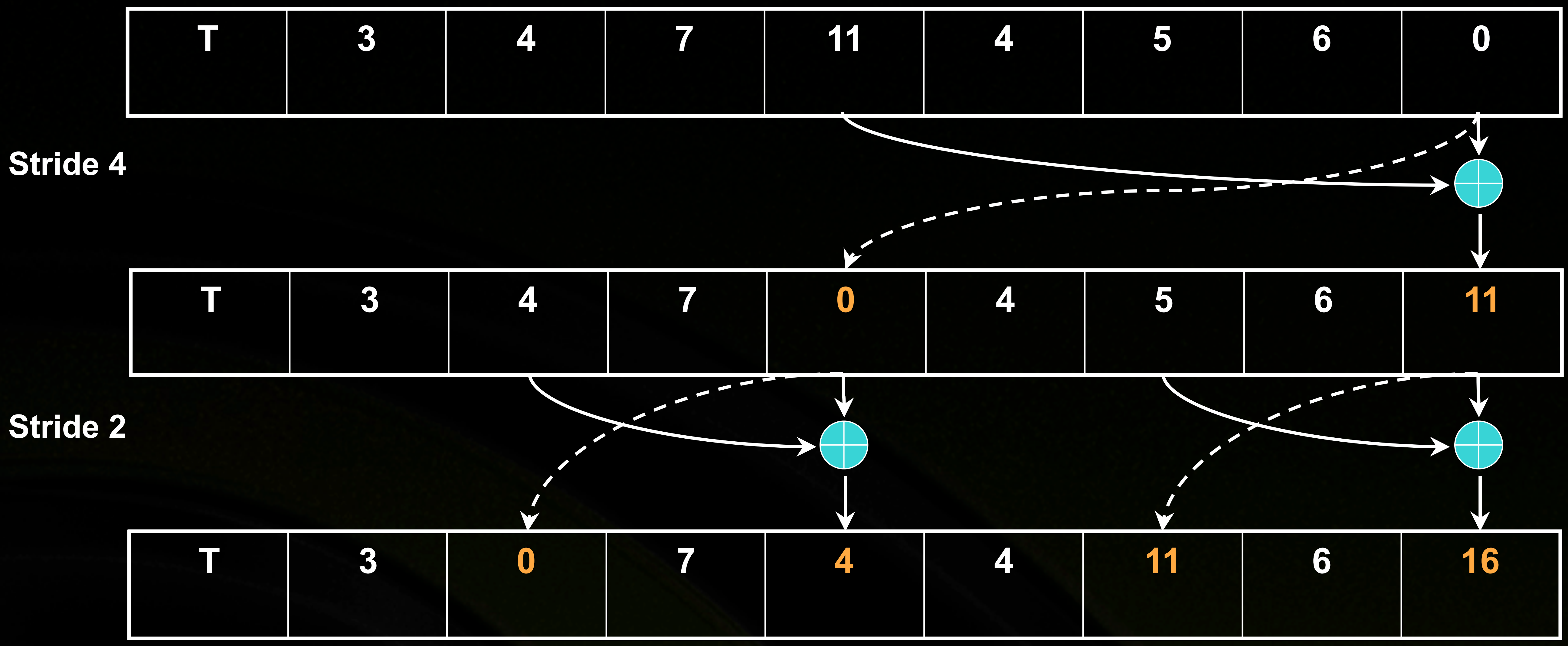


Iteration 1
1 thread

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.

Build Scan From Partial Sums

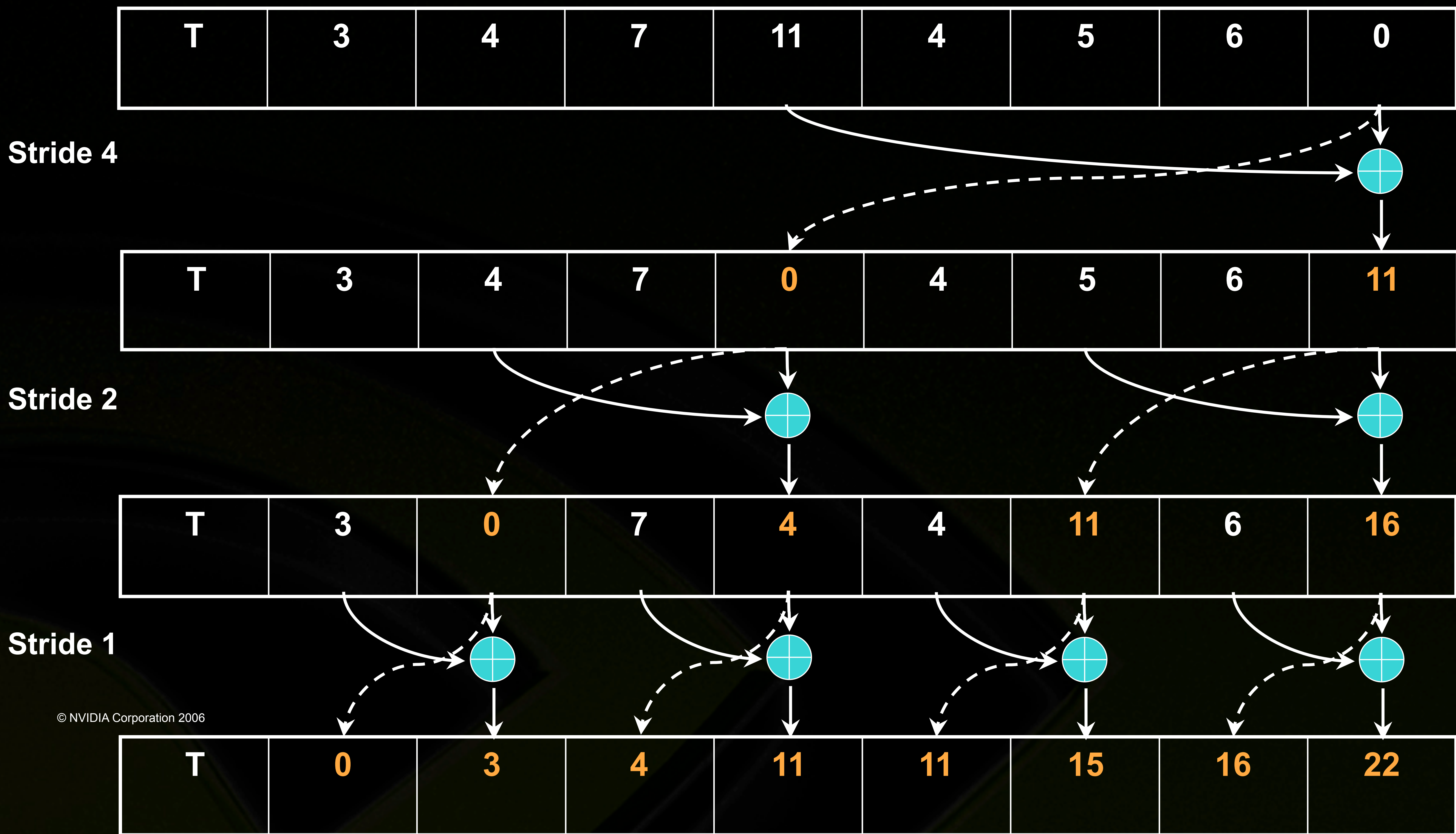


Iteration 2
2 threads

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.

Build Scan From Partial Sums



Iteration log(n)
n/2 threads

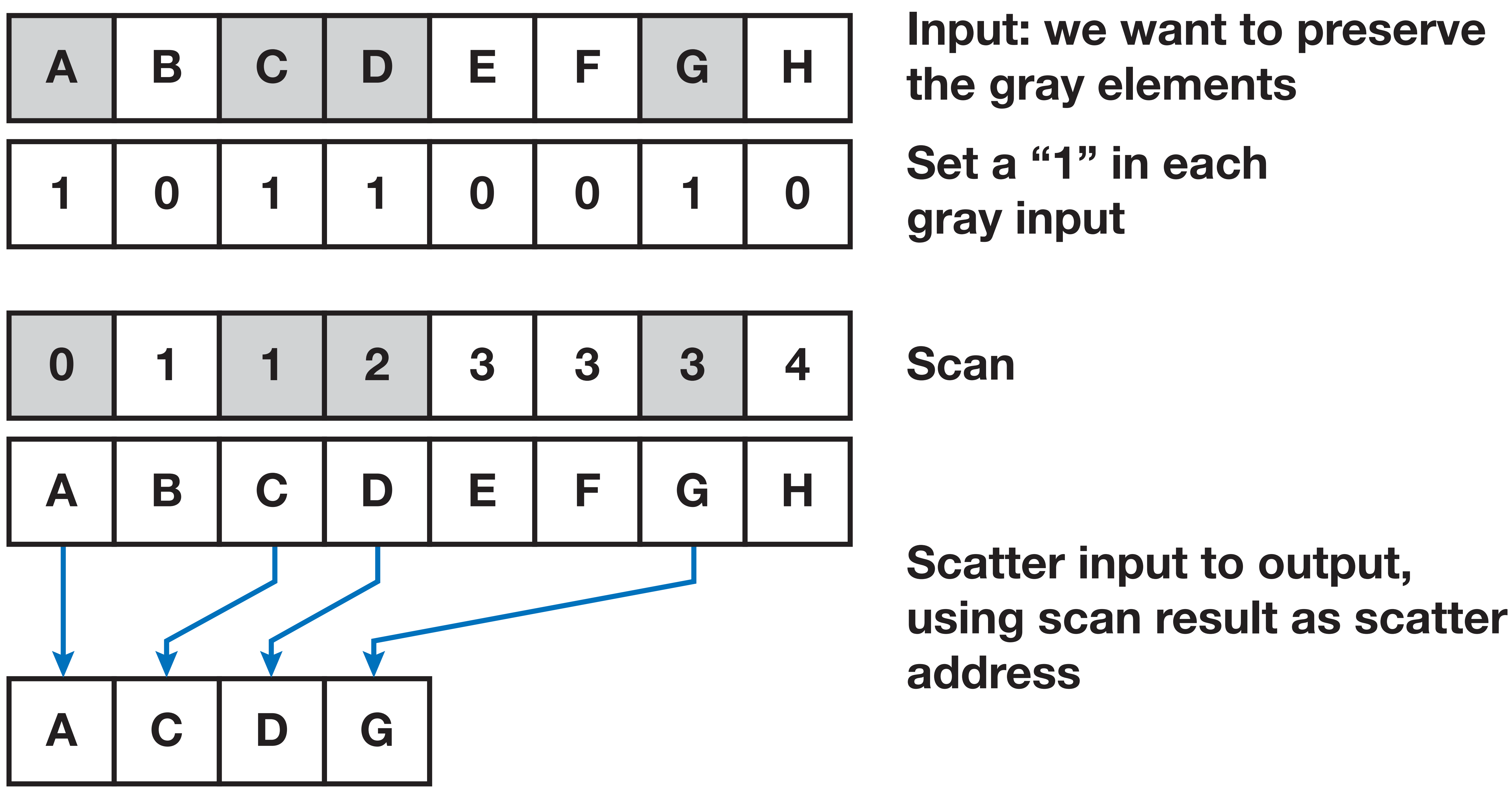
Each ⊕ corresponds to a single thread.

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!**

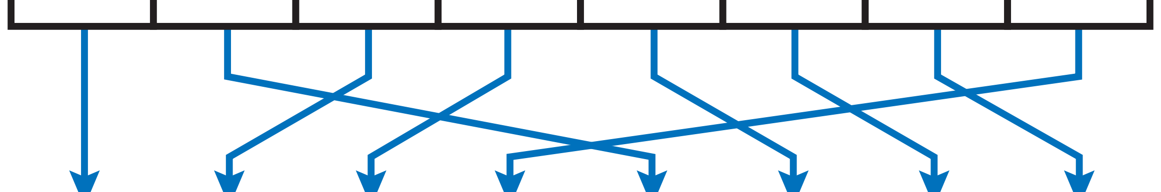
Application: Stream Compaction

- 1M elements: ~0.6-1.3 ms (on really old hardware)
- 16M elements: ~8-20 ms
- Perf depends on # elements retained



Application: Radix Sort

100	111	010	110	011	101	001	000	Input
0	1	0	0	1	1	1	0	Split based on least significant bit b
1	0	1	1	0	0	0	1	e = Set a “1” in each “0” input
0	1	1	2	3	3	3	3	f = Scan the 1s
								totalFalses = e[max] + f[max]
0-0+4 =4	1-1+4 =4	2-1+4 =5	3-2+4 =5	4-3+4 =5	5-3+4 =6	6-3+4 =7	7-3+4 =8	t = i – f + totalFalses
0	4	1	2	5	6	7	3	d = b ? t : f
100	111	010	110	011	101	001	000	



Scatter input using d as scatter address

100	010	110	000	111	011	101	001

- Sort 16M 32-bit key-value pairs: ~120 ms (on really old hardware)
- Perform split operation on each bit using scan
- Can also sort each block and merge
  - Efficient merge on GPU an active area of research

Application: Move To Front

- "banana", initial array "abcdefghijklmn…"
- For each symbol:
  - Find in array
  - Record index
  - Move symbol to front of array
- banana, 1, bacdefghijklmn…
  anana, 1, abcdefghijklmn…
  nana, 13, nabcdefghijklm…
  ana, 1, anbcdefghijklm…
  na, 1, nabcdefghijklm…
  a, 1, anbcdefghijklm…   encoding: {1 1 13 1 1 1}

Application: Move To Front

- 2 insights to parallelize:
  - Without knowing anything about predecessor or successor of sublist, can generate partial MTF list for that sublist
  - Easy to combine 2 consecutive sublists
- {dead} + {beef}, partial MTF lists are "dae" and "feb"
- Combine two lists [AppendUnique()]: Take symbols from first list that are absent from second list, append to end of second list.
  - Example: "feb" + "da" = "febda" = MTF list for "deadbeef"
- This is scan: datatype is partial MTF list, operator is AppendUnique(), identity is initial MTF list

GPU Design Principles

- **Data layouts that:**
  - **Minimize memory traffic**
  - **Maximize coalesced memory access**
- **Algorithms that:**
  - **Exhibit data parallelism**
  - **Keep the hardware busy**
  - **Minimize divergence**