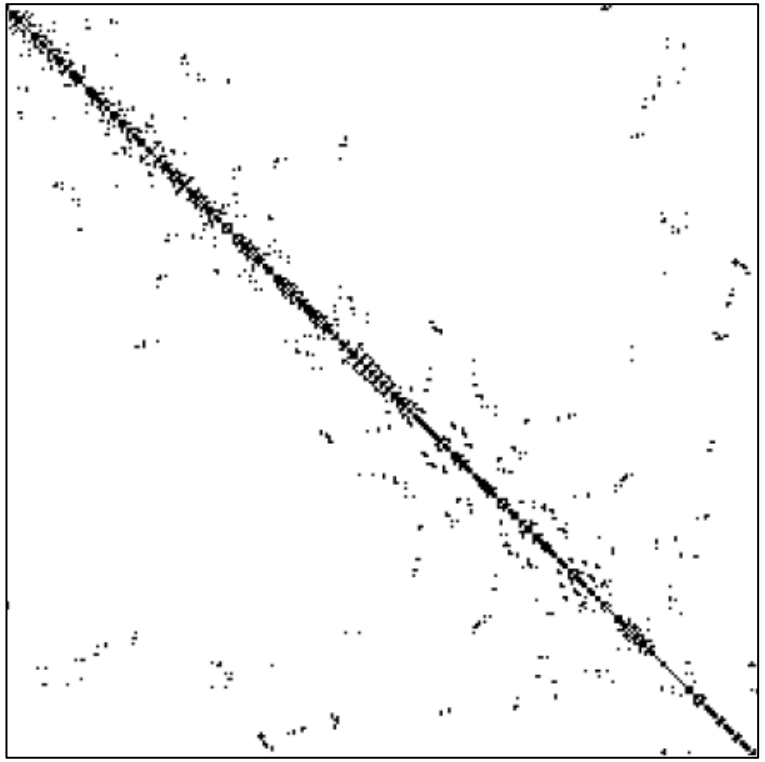
October 16, 2025

6.S894 Accelerated Computing

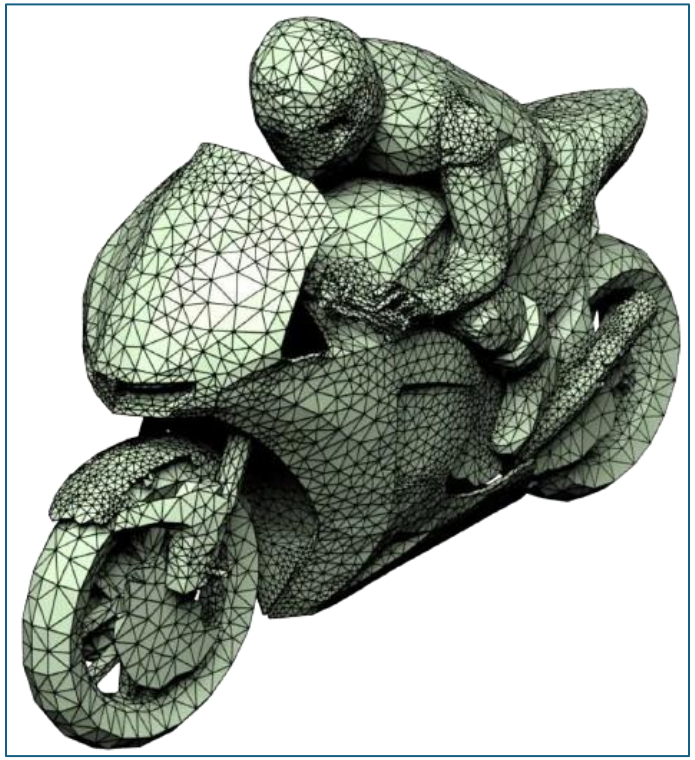
Lecture 7: Data-Parallel Primitives

Ahmed Mahmoud

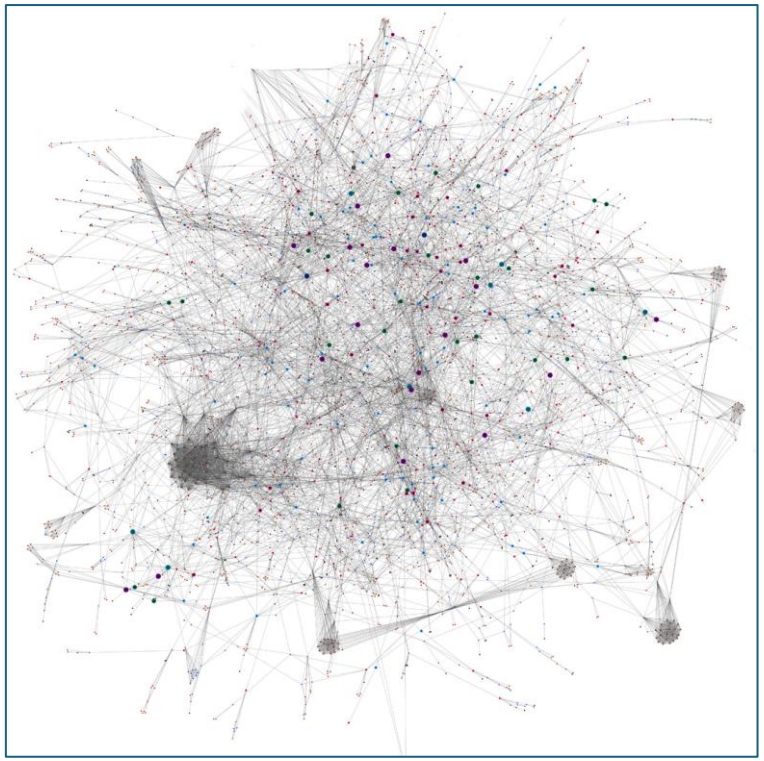
My work: *irregular* computation



Sparse Matrices



Meshes



Graphs

Outline:

- Thinking in “data-parallel patterns”

- Overview of data-parallel primitives
  - Map
  - Stencil
  - Reduce
  - Scan
  - Compaction
  - Merge

Main idea: high-performance parallel implementations of data-parallel primitives exist, allowing programs written with these primitives to run efficiently on the GPU.

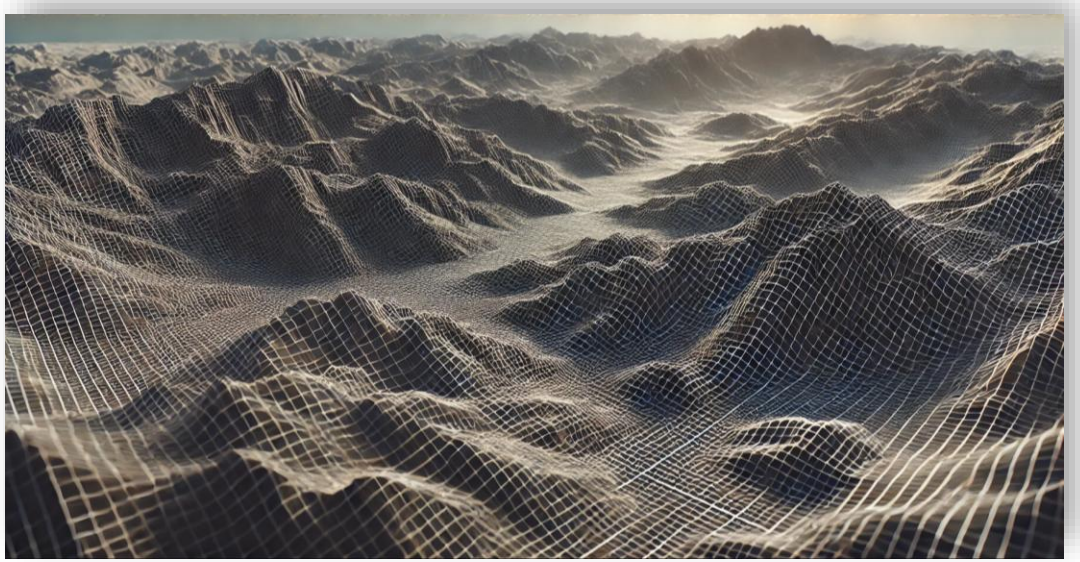
How bumpy is a surface?

Steps:

1. For each node, compute the difference between the node's height and the average height of its neighbors, then square that difference

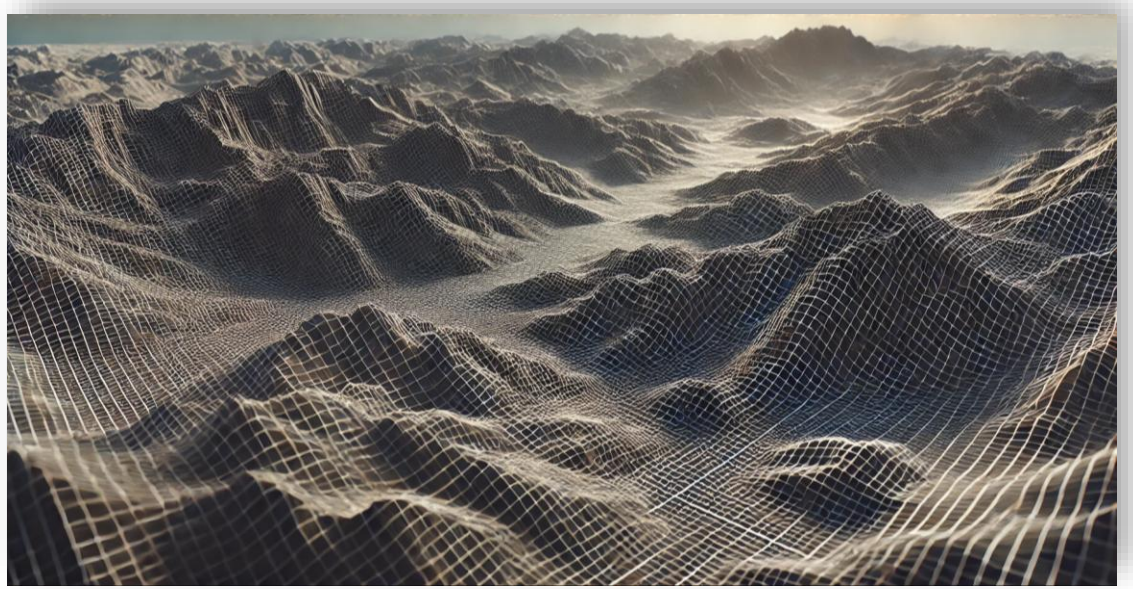
2. Sum up all those differences
   - Don’t sum all the differences that are zero

(Note: This is a fake application!)



Algorithm

```
result = 0

for all nodes:
    average =
        0.25 * (height[x - 1, y    ] +
                height[x + 1, y    ] +
                height[x    , y - 1] +
                height[x    , y + 1] )
    diff[x, y] = (height[x, y] - average)^2
```



Algorithm

```
result = 0

for all nodes:
    average =
        0.25 * (height[x - 1, y    ] +
                height[x + 1, y    ] +
                height[x    , y - 1] +
                height[x    , y + 1] )
    diff[x, y] = (height[x, y] - average)^2

 for all nodes where diff !=0:
    result += diff

return result
```

Algorithm

```
result = 0

for all nodes:
    average =
        0.25 * (height[x - 1, y    ] +
                height[x + 1, y    ] +
                height[x    , y - 1] +
                height[x    , y + 1] )
    diff[x, y] = (height[x, y] - average)^2

for all nodes where diff !=0:
    result += diff

return result
```

Pattern #1: Map

Given:

- Sequence of data elements A
- Unary function $f(x)$

map(A,f) = applies $f(x)$ to all $a_i \in A$

Algorithm

```
result = 0

for all nodes:
    average =
        0.25 * (height[x - 1, y    ] +
                height[x + 1, y    ] +
                height[x    , y - 1] +
                height[x    , y + 1] )
    diff[x, y] = (height[x, y] - average)^2

 for all nodes where diff !=0:
    result += diff

return result
```

Pattern #2: Stencil

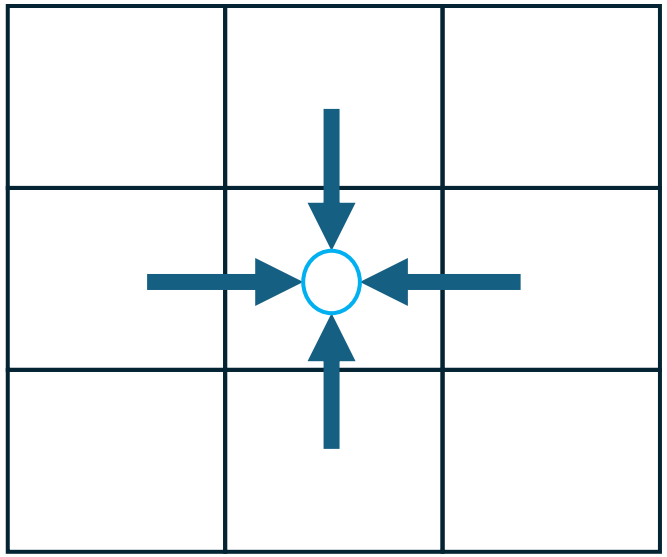
Given an input sequence a and a range I, *stencil* produces an output sequence p such that

$$p = a[I]$$

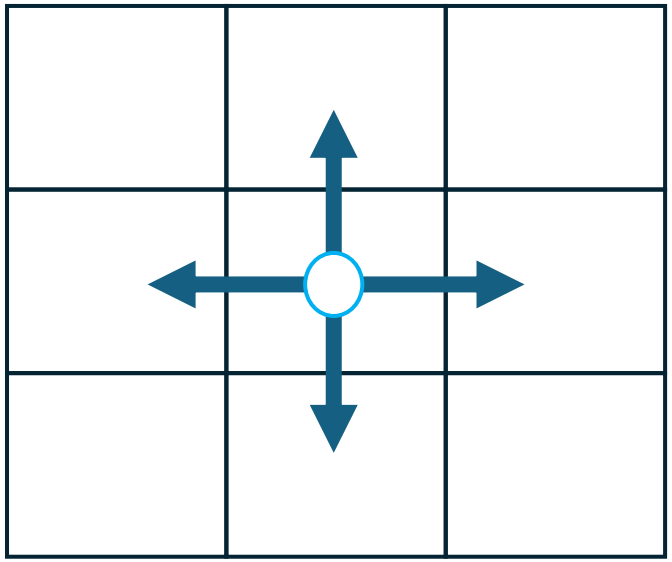
Pattern #2: Stencil

Given an input sequence a and a range I, *stencil* produces an output sequence p such that

$$p = a[I]$$



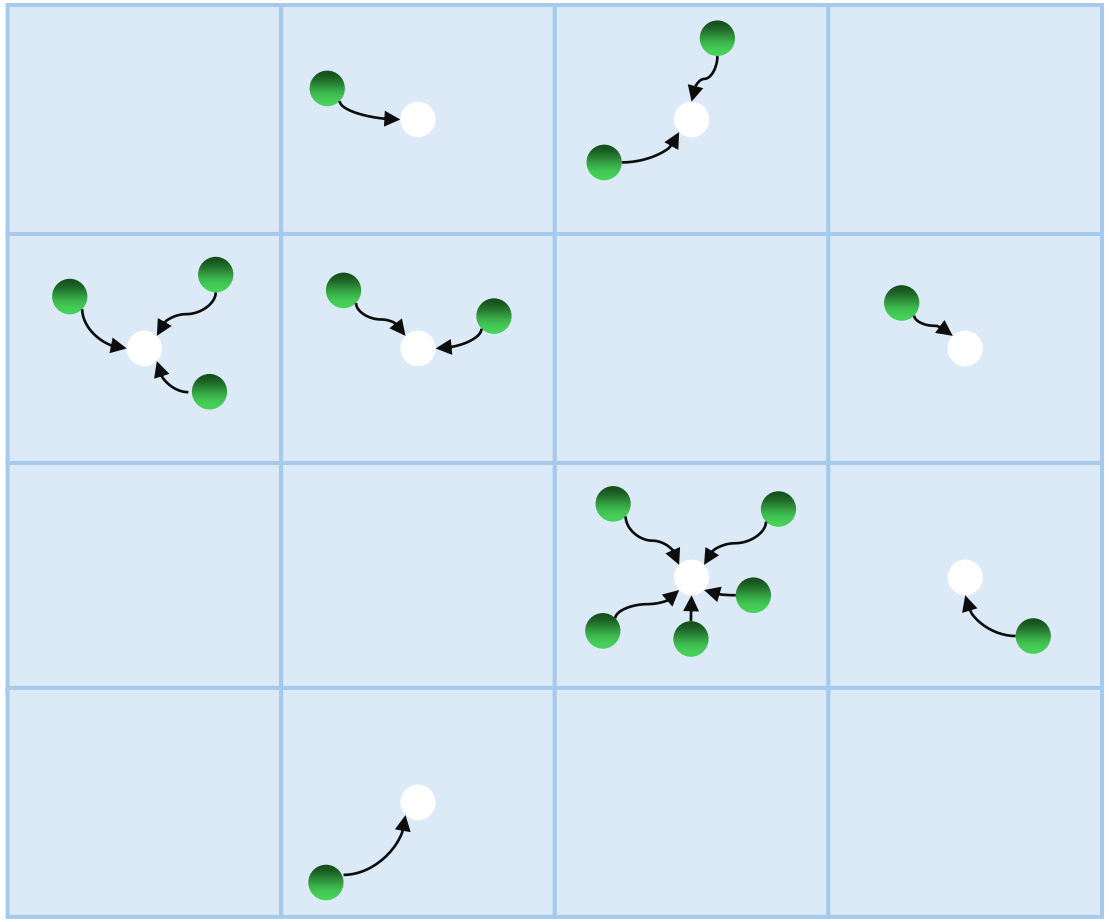
Gather



Scatter

Pattern #2: Stencil

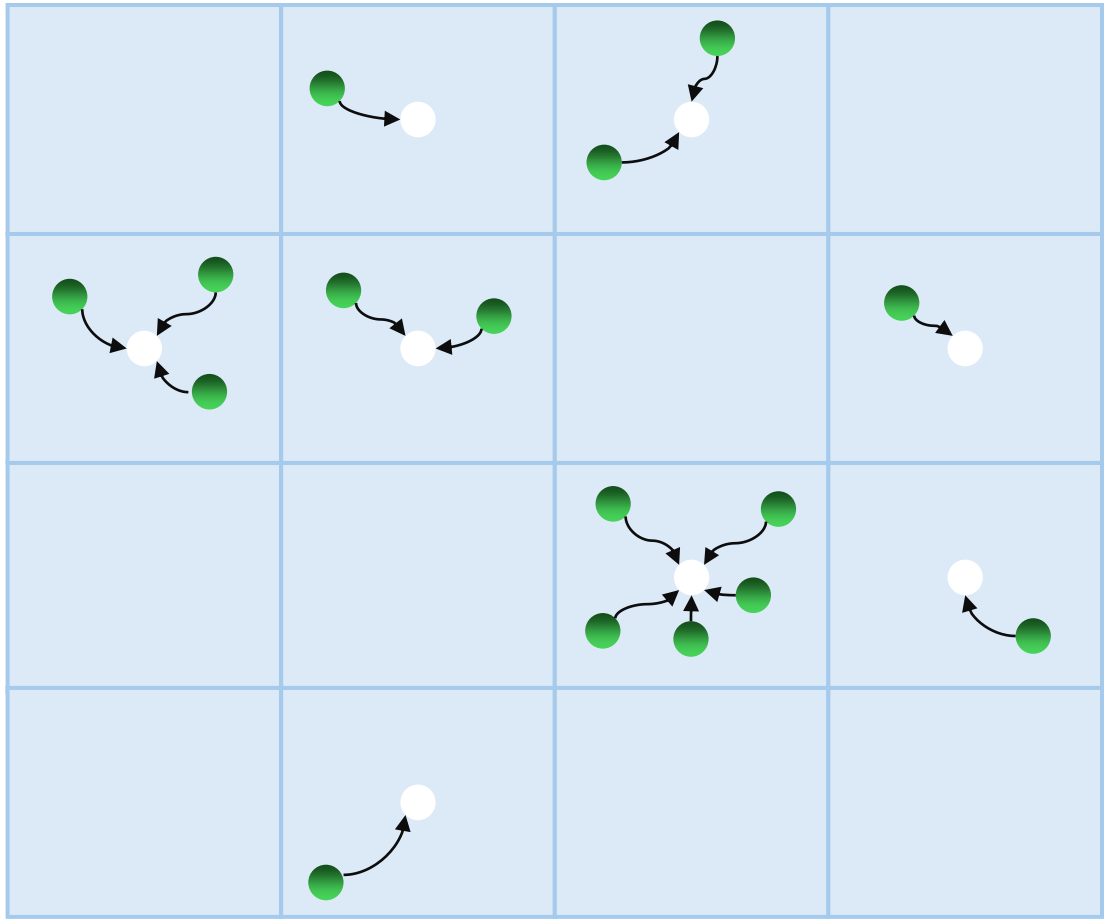
Particles-to-grid (P2G)



Pattern #2: Stencil

Particles-to-grid (P2G)

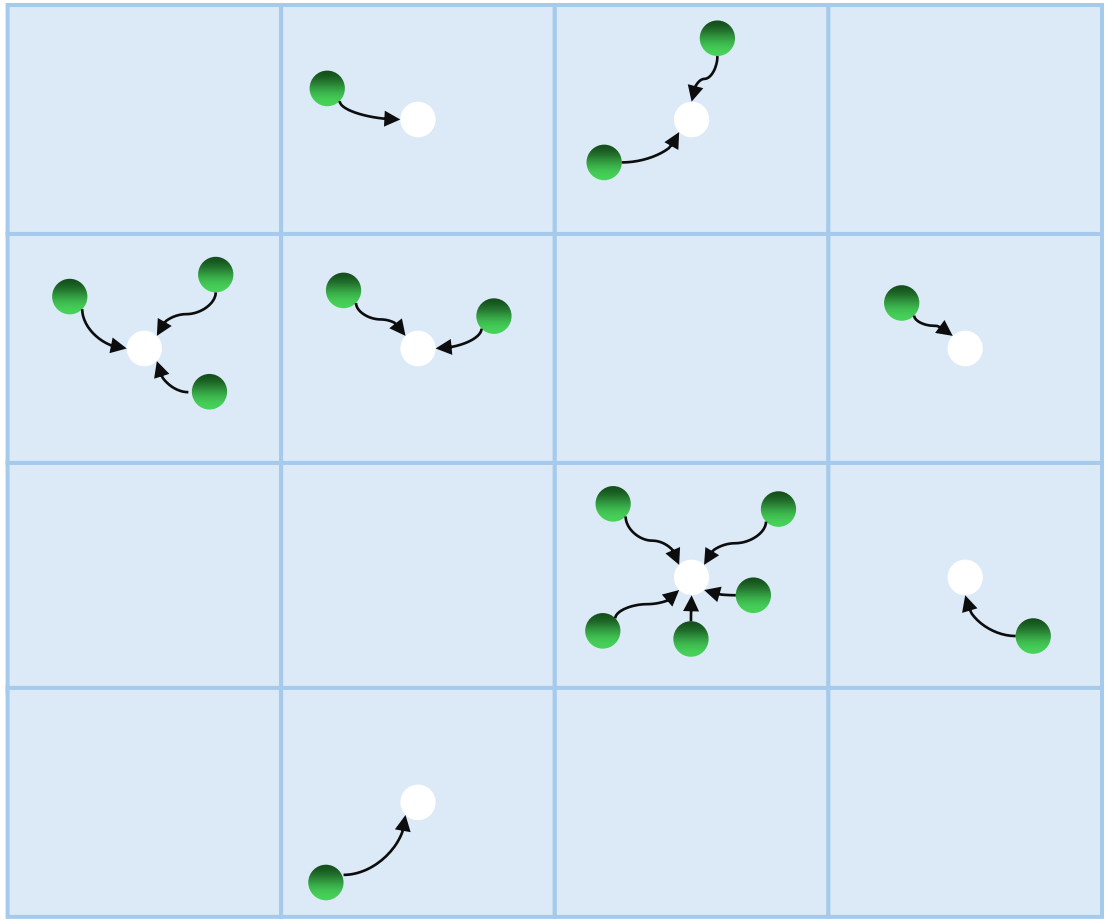
1. **Gather:** Each cell collects information from the particles within it



Pattern #2: Stencil

Particles-to-grid (P2G)

1. **Gather:** Each cell collects information from the particles within it
   - Needs to collect particles in each cell
   - Different number of particles per cell
   - Many empty cells
   - **Workload imbalance**

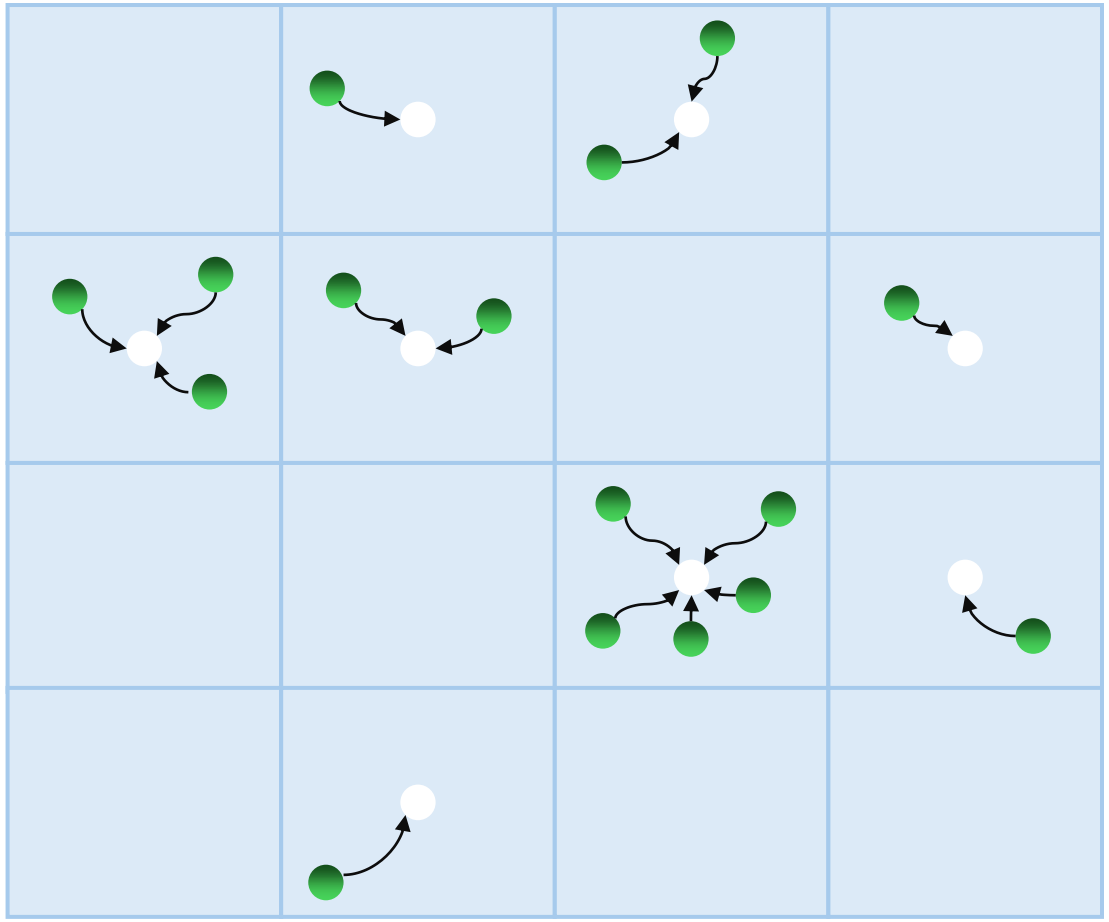


Pattern #2: Stencil

Particles-to-grid (P2G)

1. **Gather:** Each cell collects information from the particles within it
   - Needs to collect particles in each cell
   - Different number of particles per cell
   - Many empty cells
   - **Workload imbalance**

2. **Scatter:** Each particle distributes information to the cell it belongs to
   - Needs synchronization (e.g., atomics)



Gao, Wang, Wu, Pradhana, Sifakis, Yuksel, and Jiang. 2018.
GPU optimization of material point methods.
ACM Transactions on Graphics.

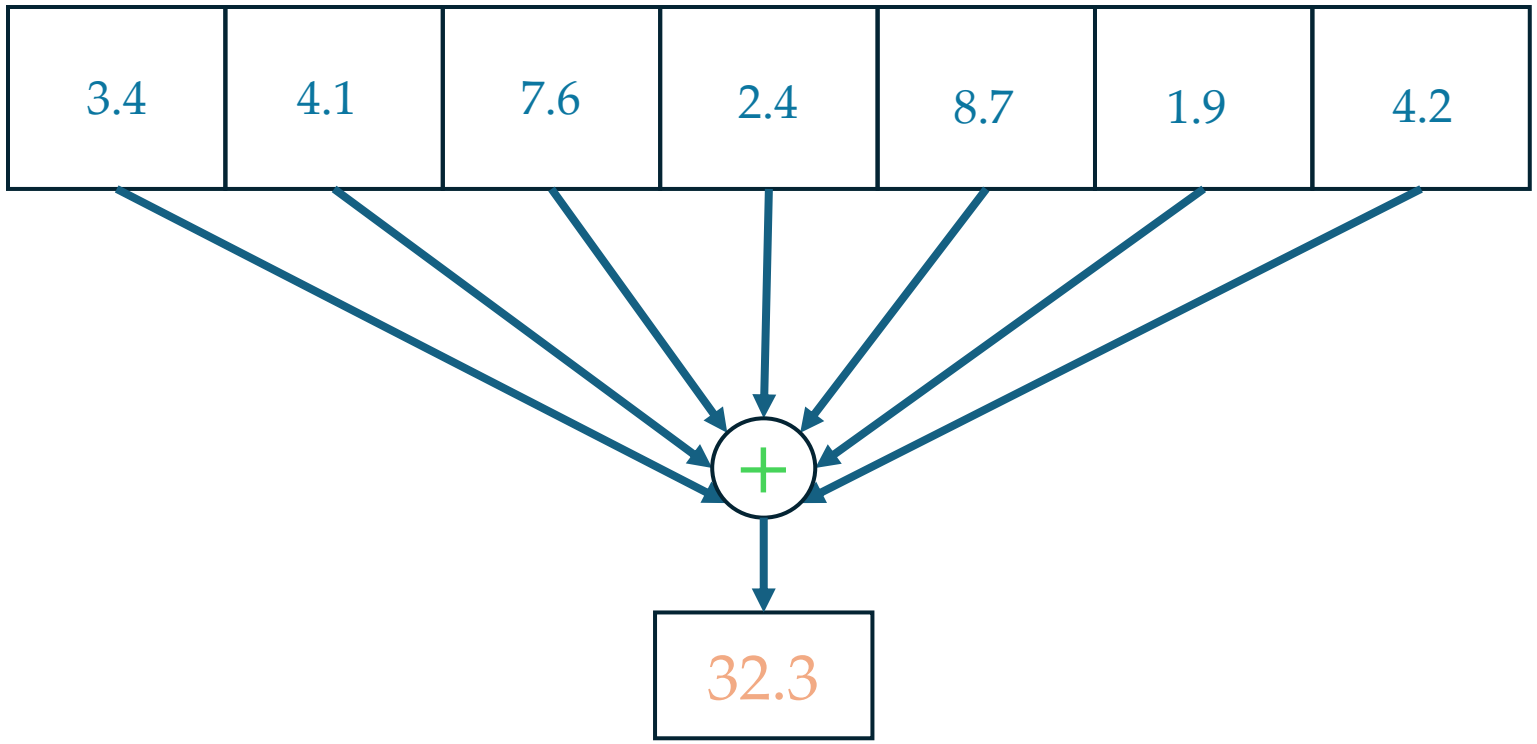
Algorithm

```
result = 0

for all nodes:
    average =
        0.25 * (height[x - 1, y    ] +
                height[x + 1, y    ] +
                height[x    , y - 1] +
                height[x    , y + 1] )
    diff[x, y] = (height[x, y] - average)^2

for all nodes where diff !=0:
    result += diff

return result
```

Pattern #3: Reduce

Given a sequence of input $S = [a_0, a_1, \ldots, a_{n-1}]$ and a binary associative and commutative operator $\oplus$ (e.g., +, *, min, max)

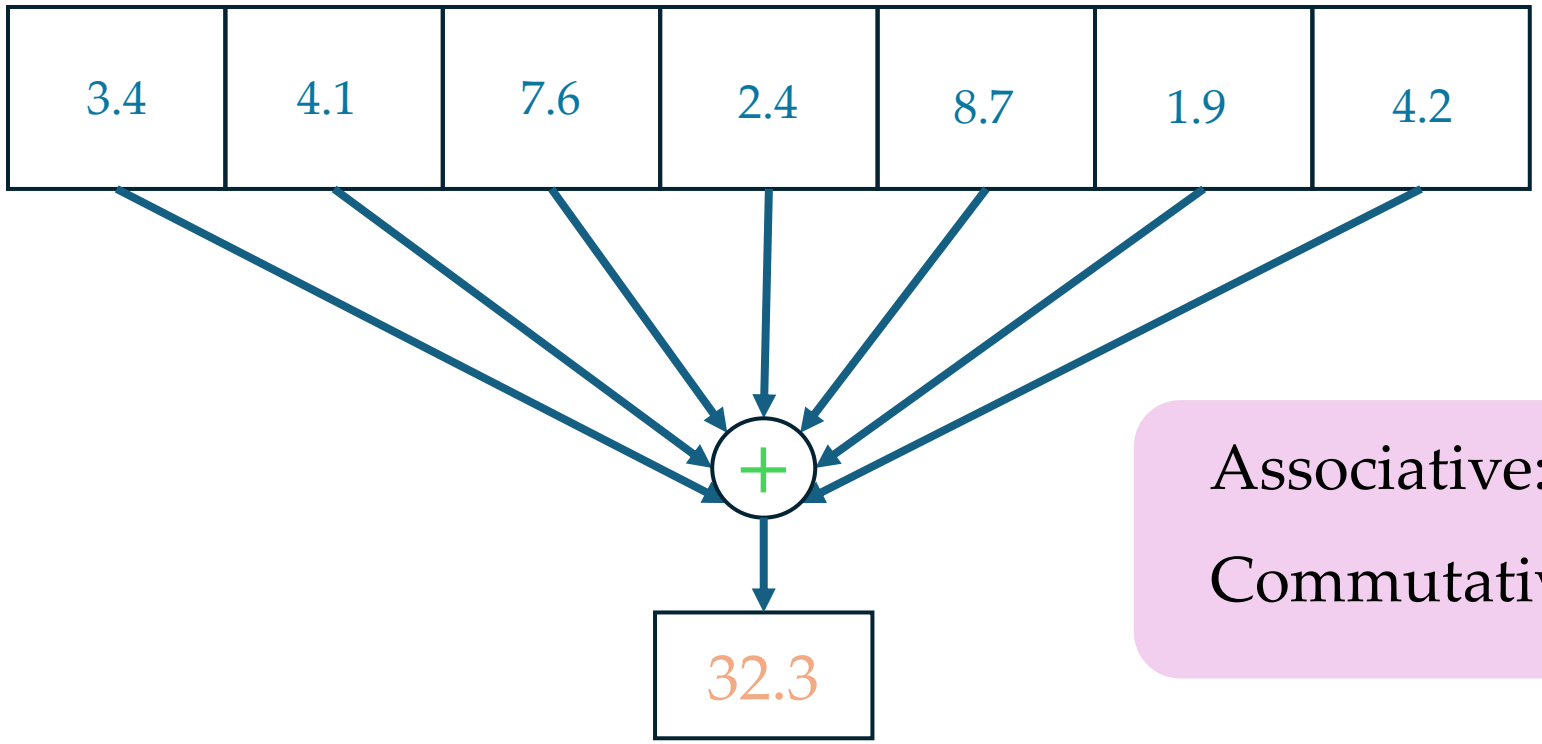
$$\text{Reduce}(\oplus, S) = a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$$



Pattern #3: Reduce

Given a sequence of input $S = [a_0, a_1, \ldots, a_{n-1}]$ and a binary associative and commutative operator $\oplus$ (e.g., +, *, min, max)

$$\text{Reduce}(\oplus, S) = a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1}$$



Associative: $(a_0 \oplus a_1) \oplus a_2 = a_0 \oplus (a_1 \oplus a_2)$

Commutative: $a_0 \oplus a_1 = a_1 \oplus a_0$

Pattern #3: Reduce

Work efficiency:

The total amount of work done over by all processors

Pattern #3: Reduce

Work efficiency:

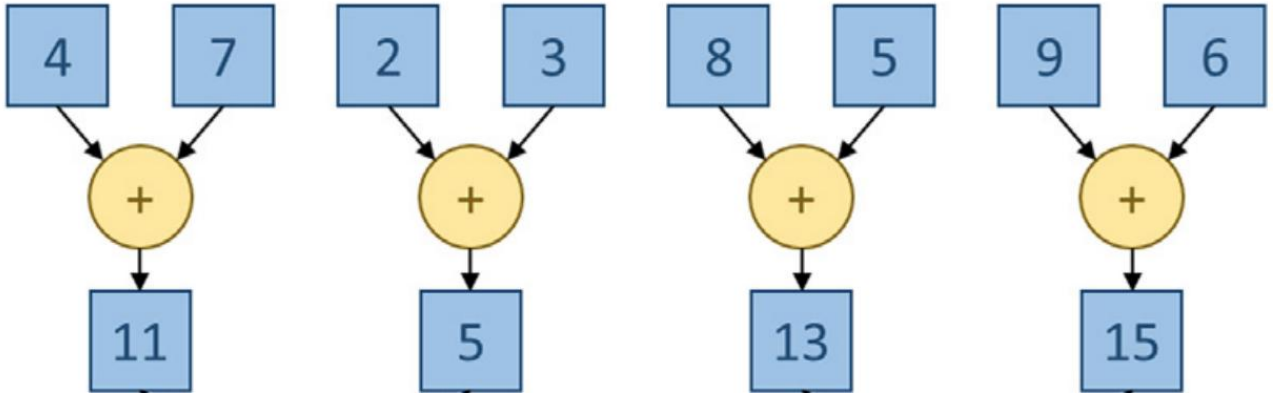
The total amount of work done over by all processors

Step efficiency:

The number of steps it takes to do that work

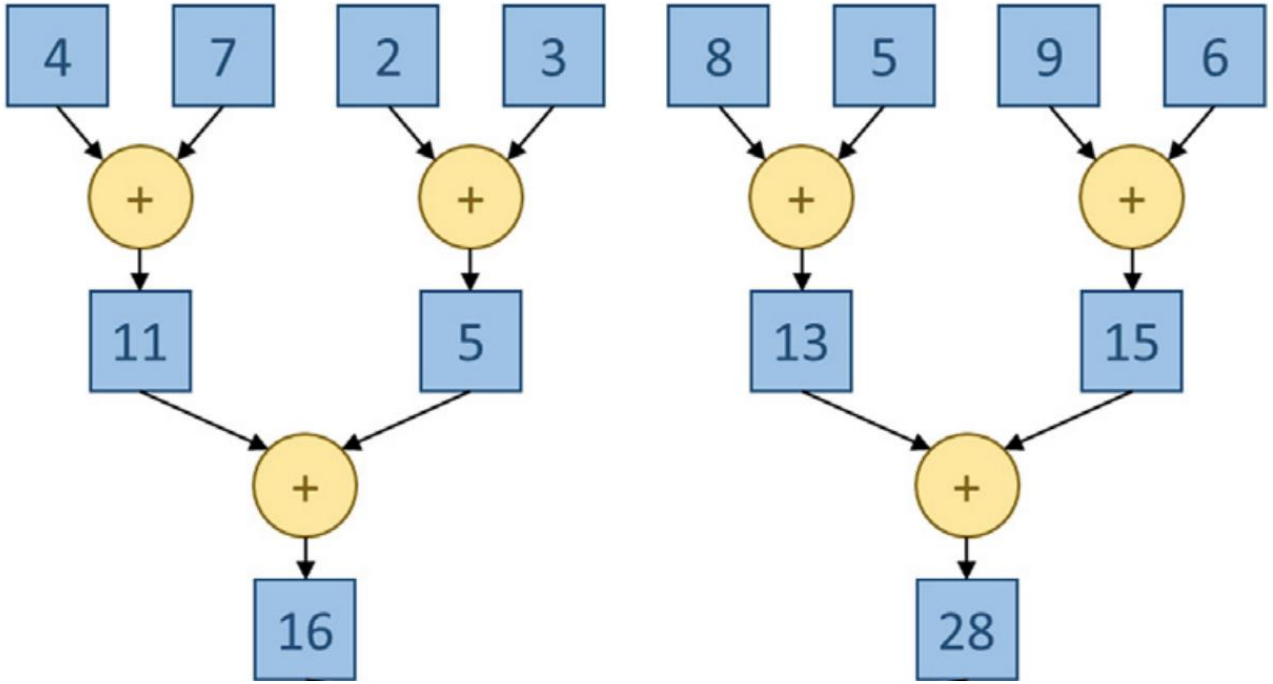
Pattern #3: Reduce

- Parallel Reduction
  - Add two halves of domain together repeatedly



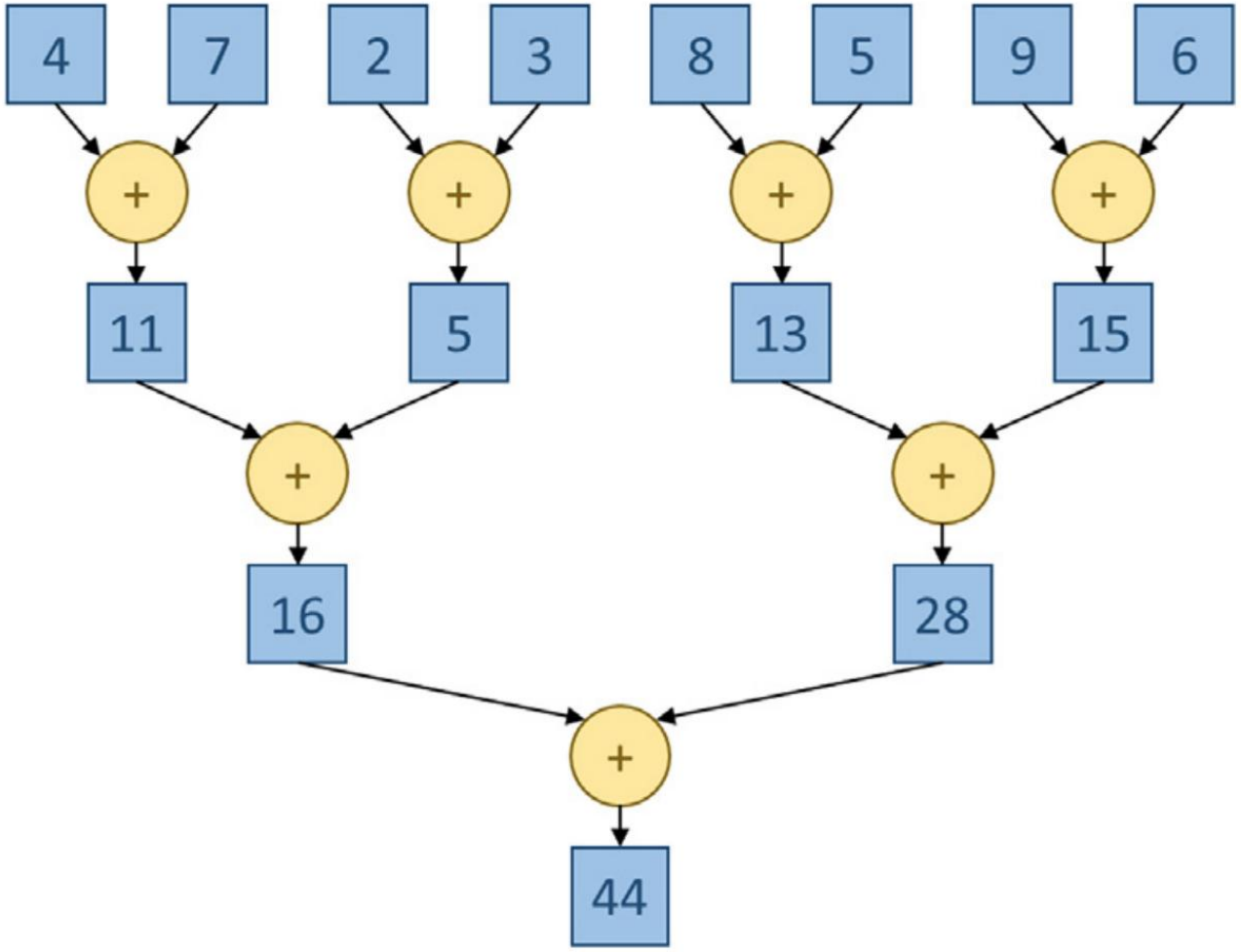
Pattern #3: Reduce

- Parallel Reduction
  - Add two halves of domain together repeatedly



Pattern #3: Reduce

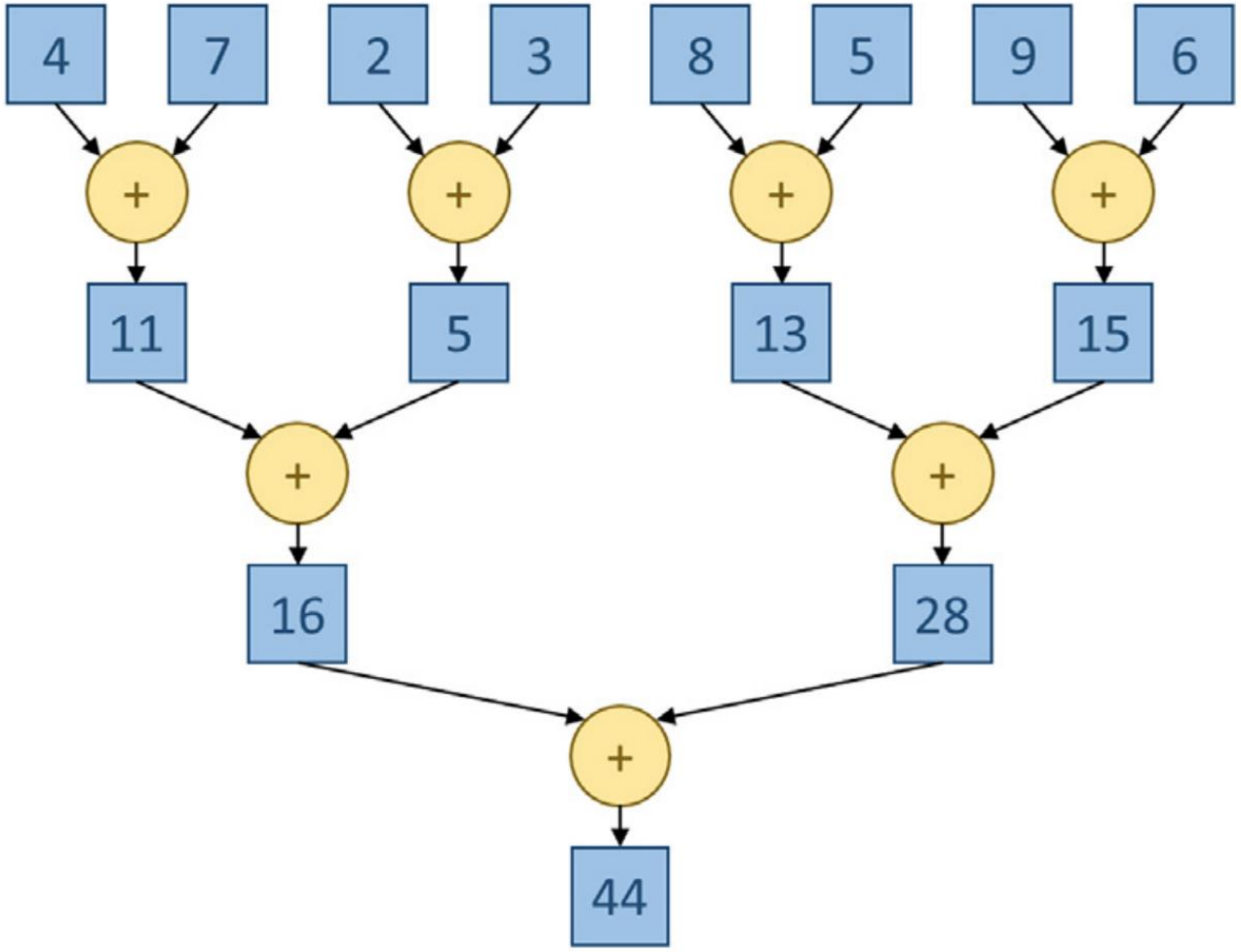
- Parallel Reduction
  - Add two halves of domain together repeatedly



Pattern #3: Reduce

- Parallel Reduction
  - Add two halves of domain together repeatedly

$O(\log_2 N)$ Steps
$O(N)$ Work

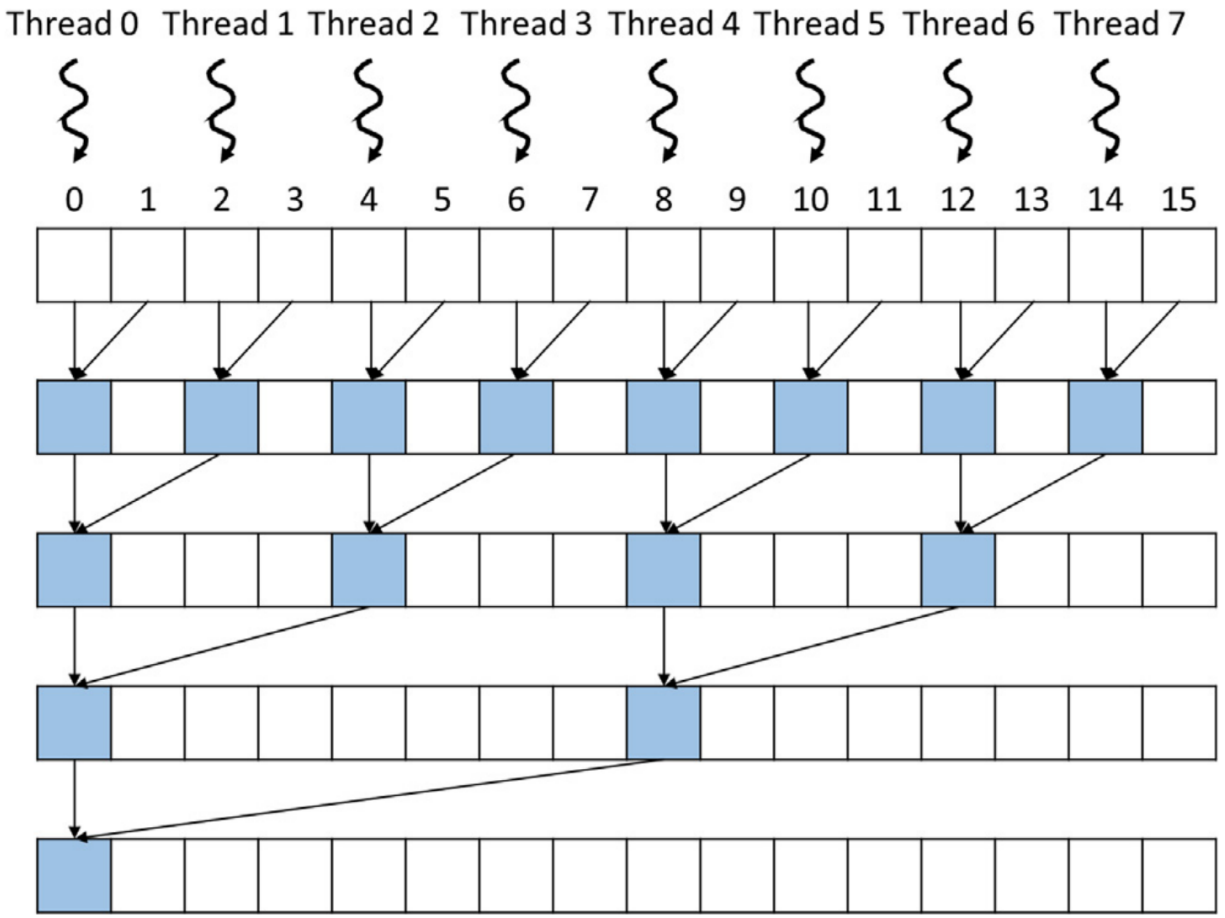


Pattern #3: Reduce

- Parallel Reduction
  - log(N) parallel steps, each step S does N/2S independent ops
    - Step complexity is $O(\log_2 N)$
  - Performs N/2+ N /4+…+1= N -1 operations
    - Work complexity is O(N)
    - It is work-efficient, i.e., does not perform more operations than a sequential algorithm

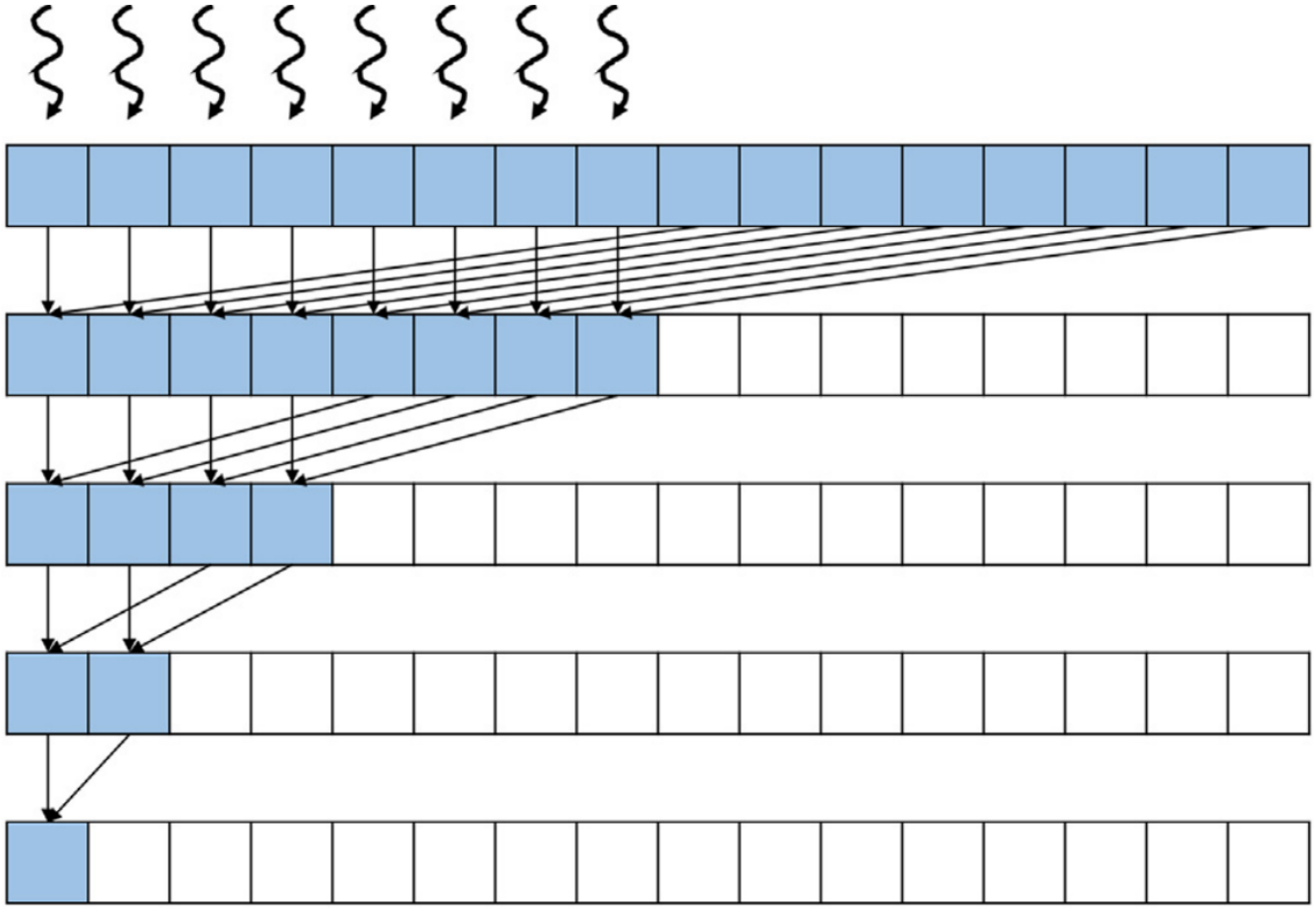
Pattern #3: Reduce

- Parallel Reduction Kernel



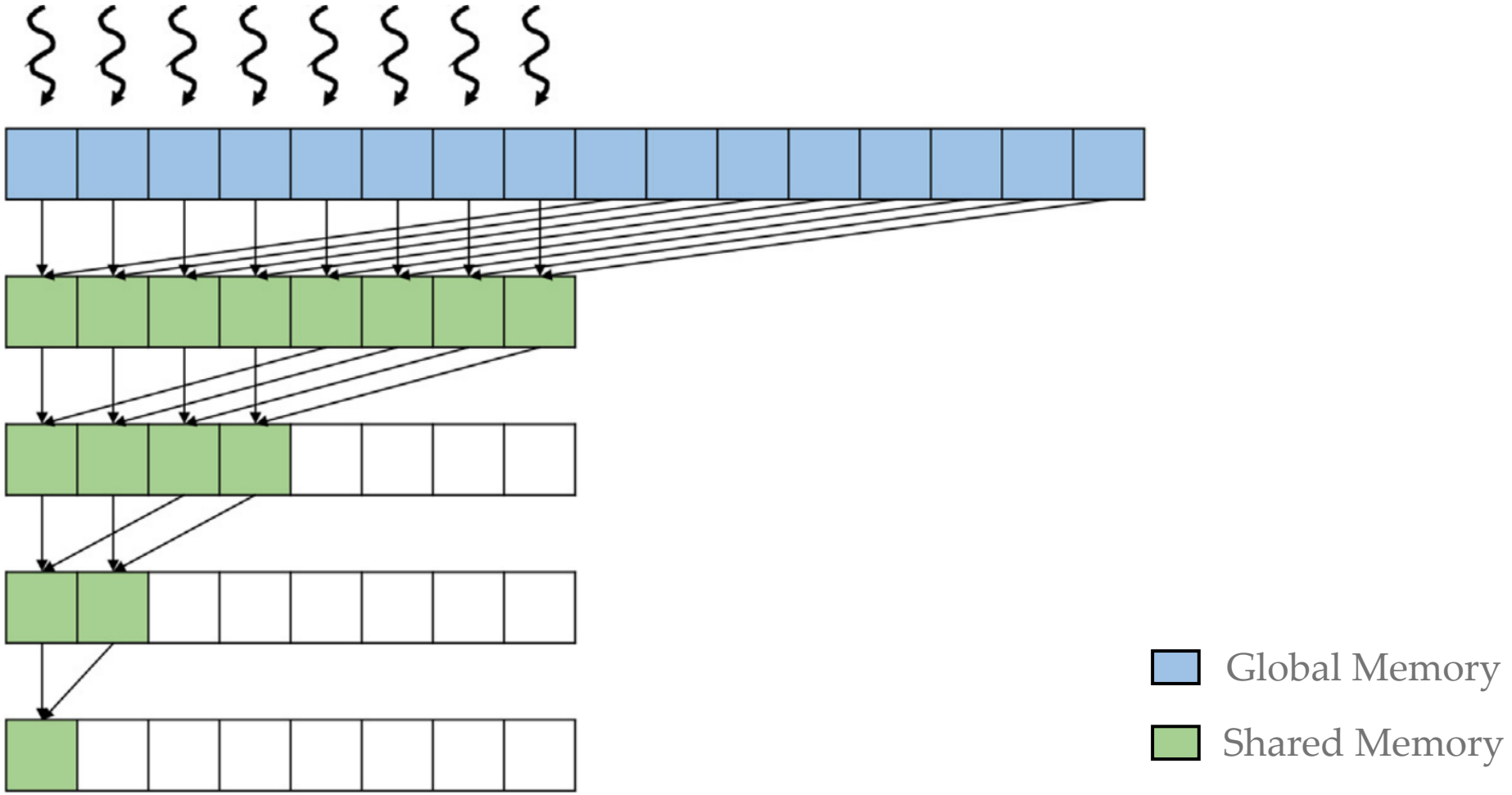
Pattern #3: Reduce

- Improve memory accesses



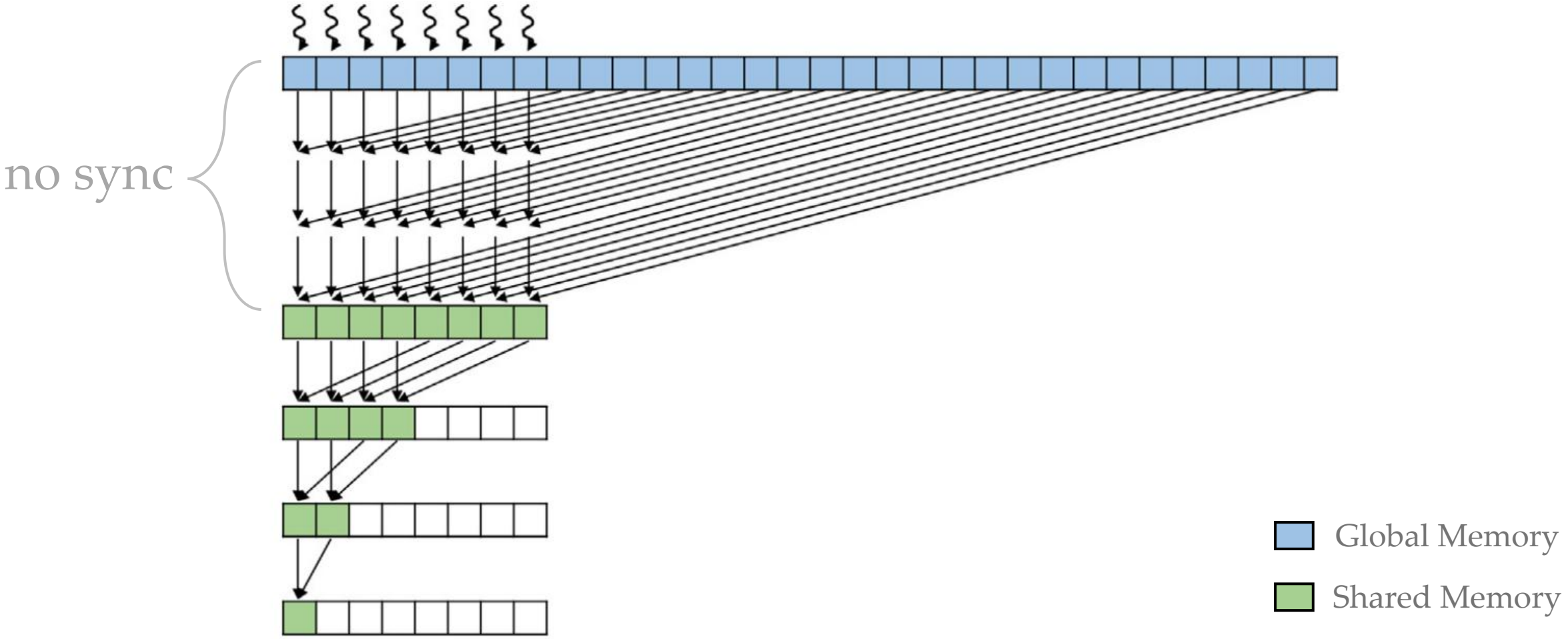
Pattern #3: Reduce

- Minimize global memory accesses



Pattern #3: Reduce

- Thread coarsening to reduce overhead

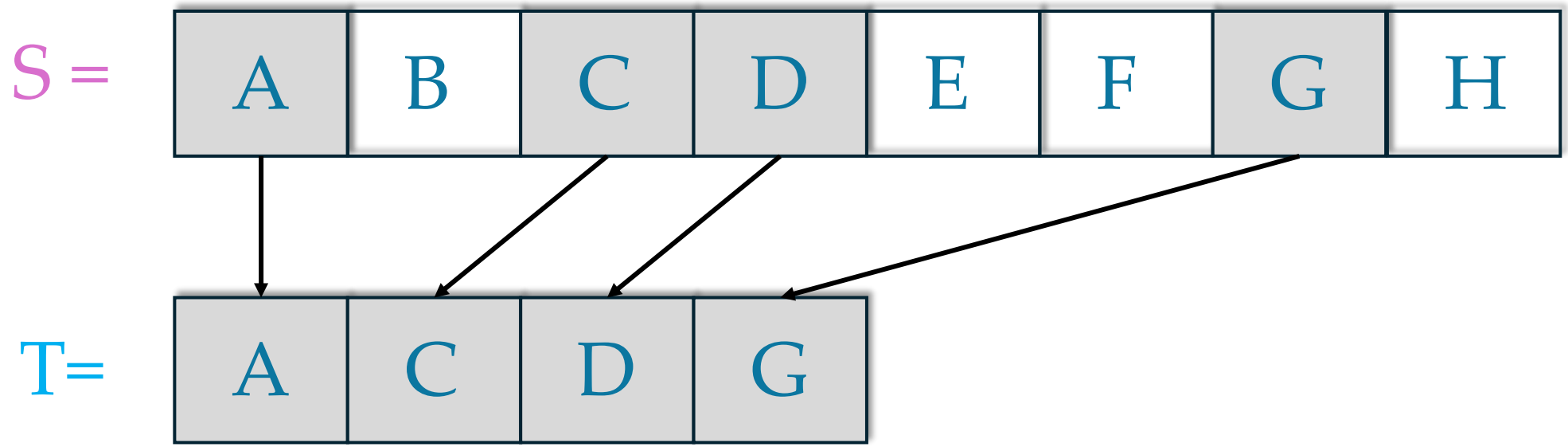


Algorithm

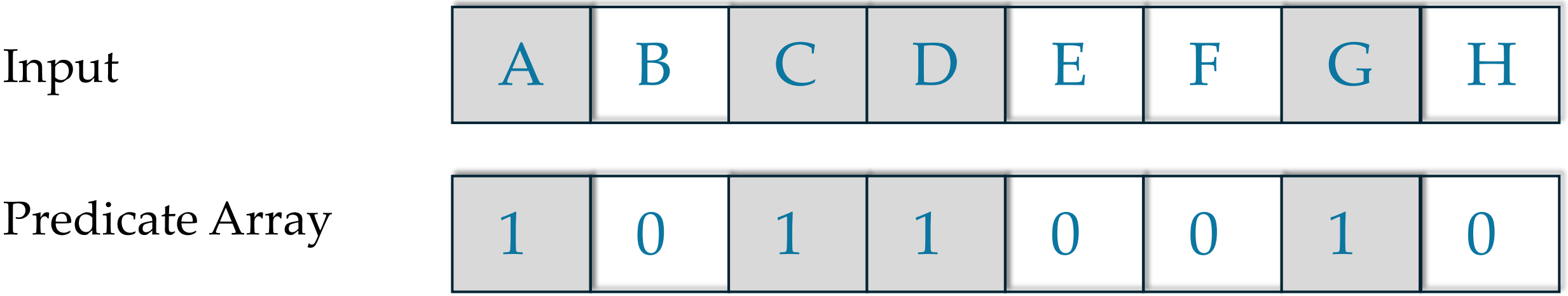
```
result = 0

for all quads:
    average =
        0.25 * (height[x - 1, y    ] +
                height[x + 1, y    ] +
                height[x    , y - 1] +
                height[x    , y + 1] )
    diff[x, y] = (height[x, y] - average)^2

for all nodes where diff !=0:
    result += diff

return result
```

Pattern #4: Stream Compaction

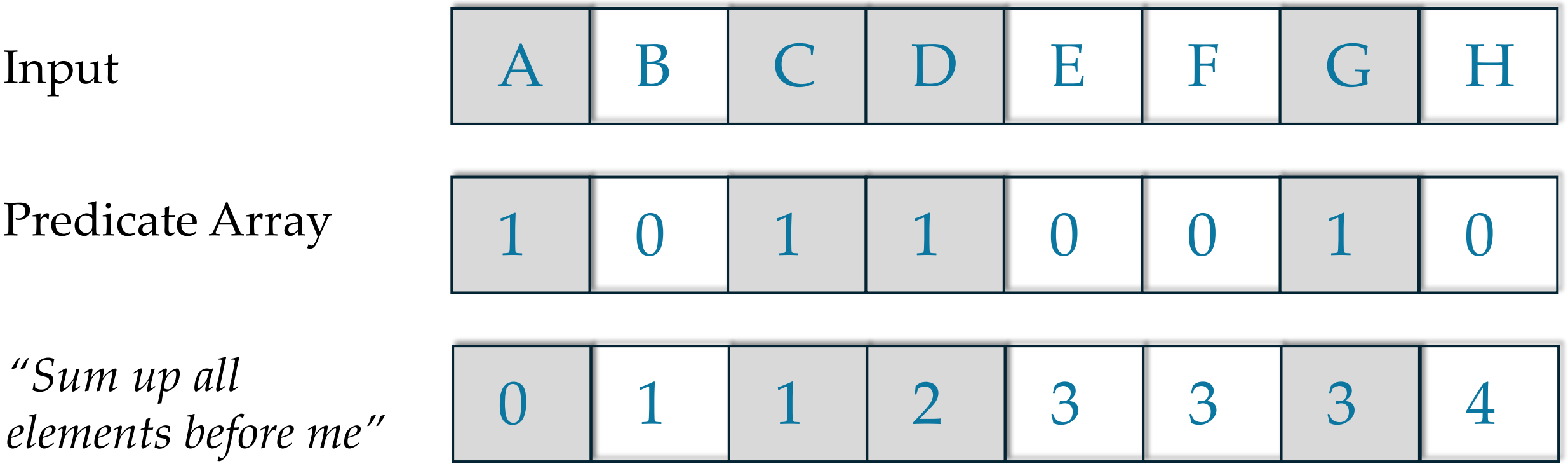
Given an input sequence S and a **predicate** P, output another sequence T that only *contains elements that satisfy the predicate*



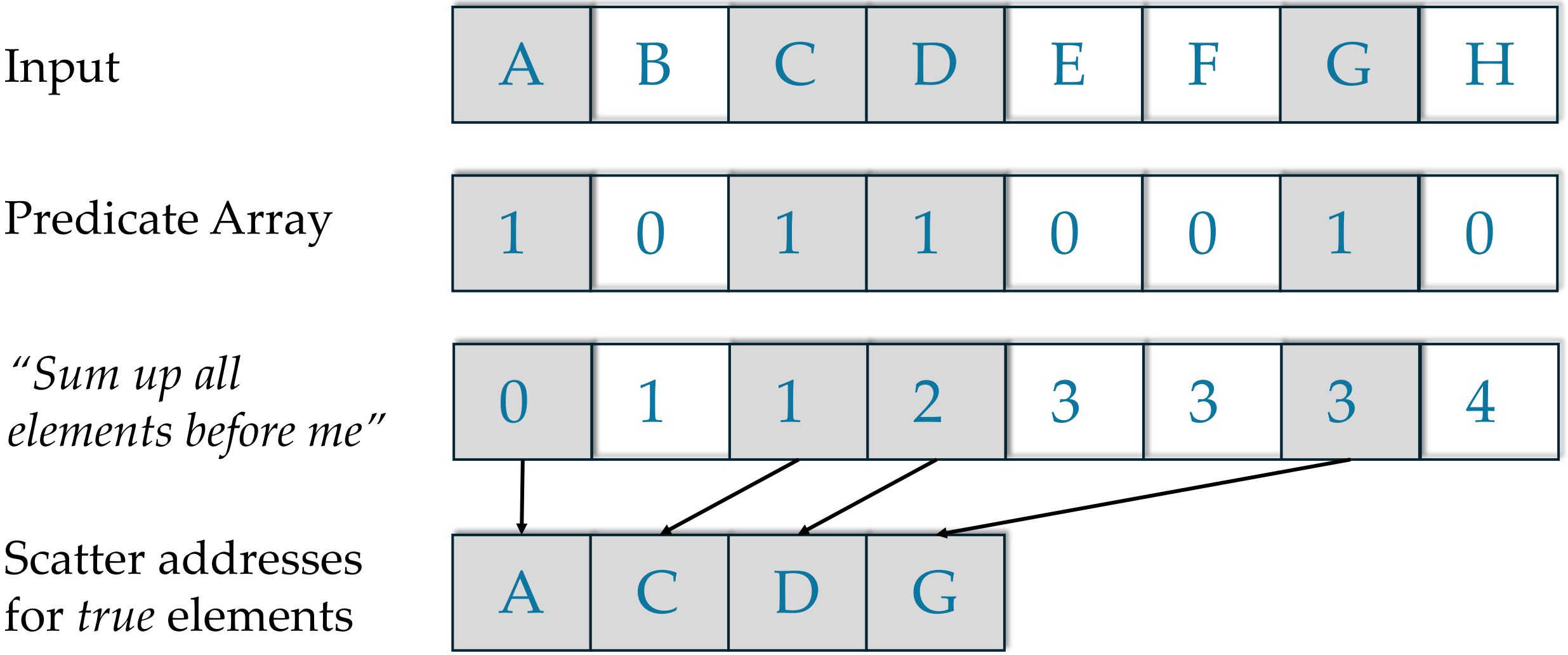
Pattern #4: Stream Compaction



Pattern #4: Stream Compaction



Pattern #4: Stream Compaction



“Where do I write my output?”

“Where do I write my output?”

- The answer is:

  “That depends on how much the other threads need to write!”

"Where do I write my output?"

- The answer is:

  "That depends on how much the other threads need to write!"

- **Scan** is an efficient way to answer this question in parallel

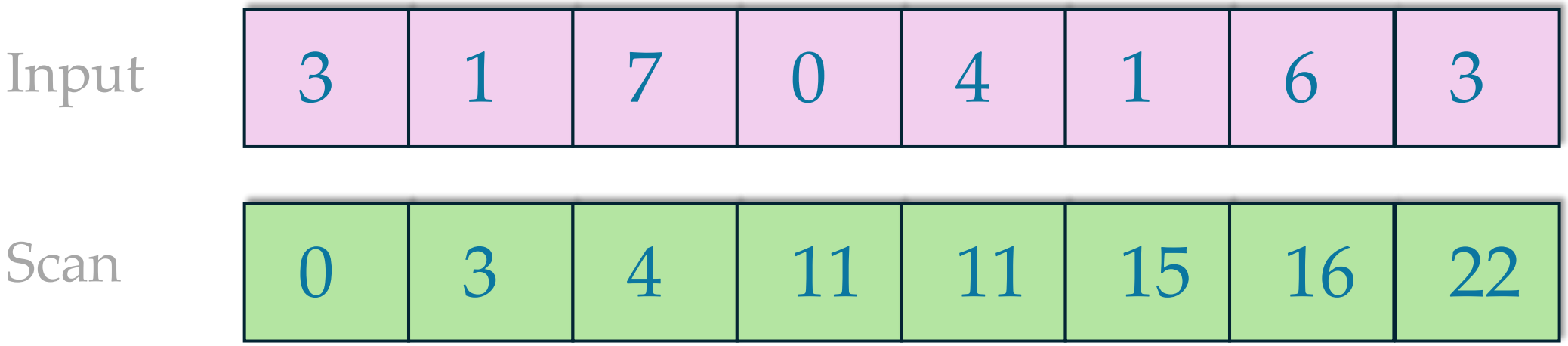
- Examples: marching cubes, collision detection, sorting, building trees

Pattern #5: Scan

Given a sequence of input $S = [a_0, a_1, \ldots, a_{n-1}]$ and a binary associative operator $\oplus$ with identity I

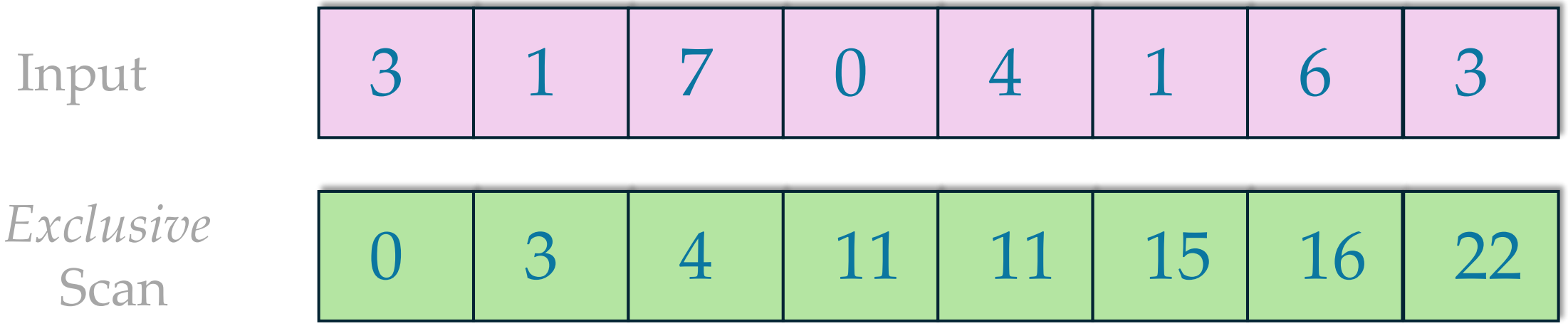
$$\text{Scan}(S, \oplus, I) = [I, a_0, a_0 \oplus a_1, \ldots, (a_0 \oplus a_1 \ldots \oplus a_{n-2})]$$

When $\oplus$ is addition, this corresponds to the **prefix sum**



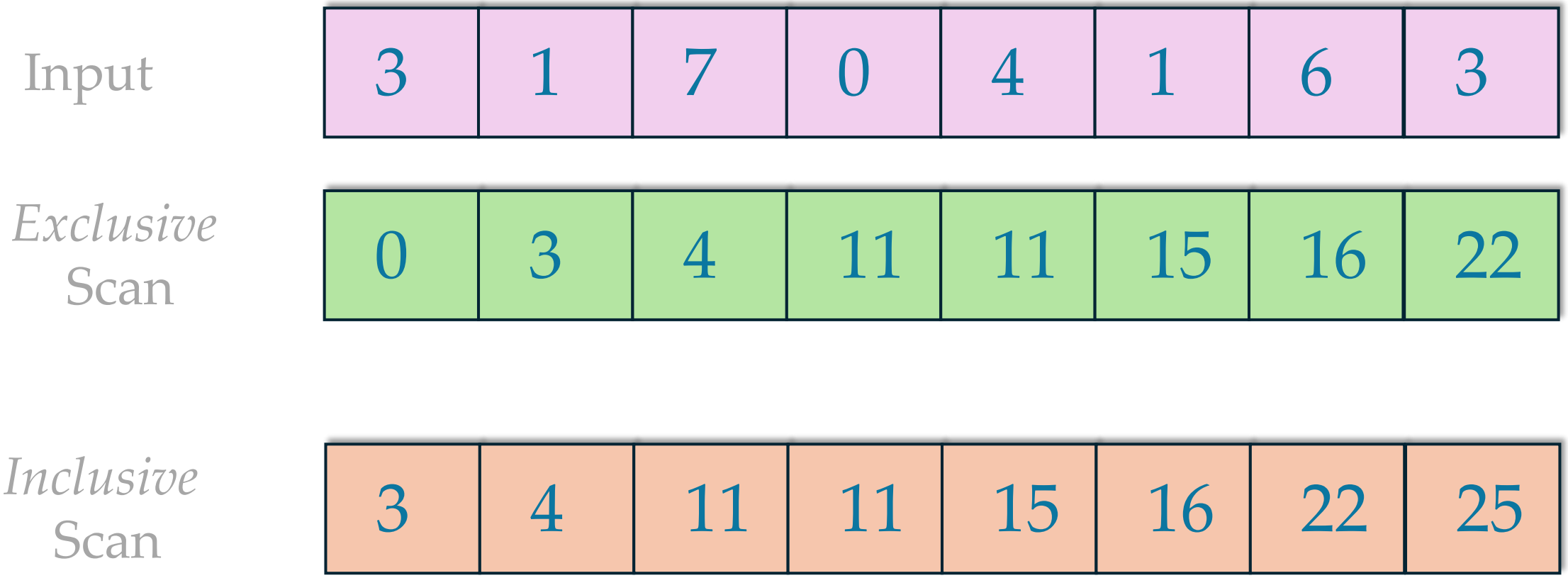
Pattern #5: Scan

Example for when $\oplus$ is addition, i.e., prefix sum



Pattern #5: Scan

Example for when $\oplus$ is addition, i.e., prefix sum



Pattern #5: Scan

Sequential implementation

- N additions needed for N elements—$O(N)$

Input $[x_0, x_1, x_2, \ldots]$

Output $[y_0, y_1, y_2, \ldots]$

Such that

$y_0 = x_0$

$y_1 = x_0 + x_1$

$y_2 = x_0 + x_1 + x_2$

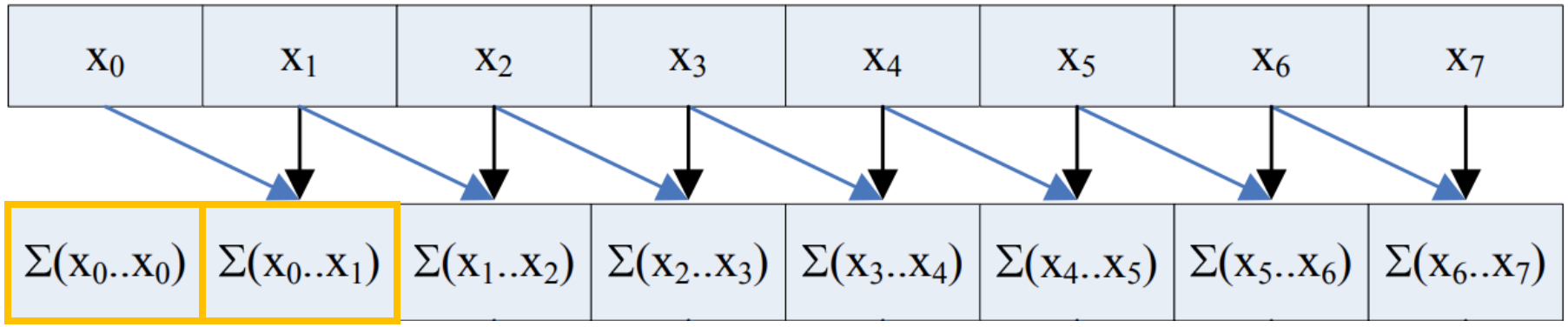
…

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

Pattern #5: Scan

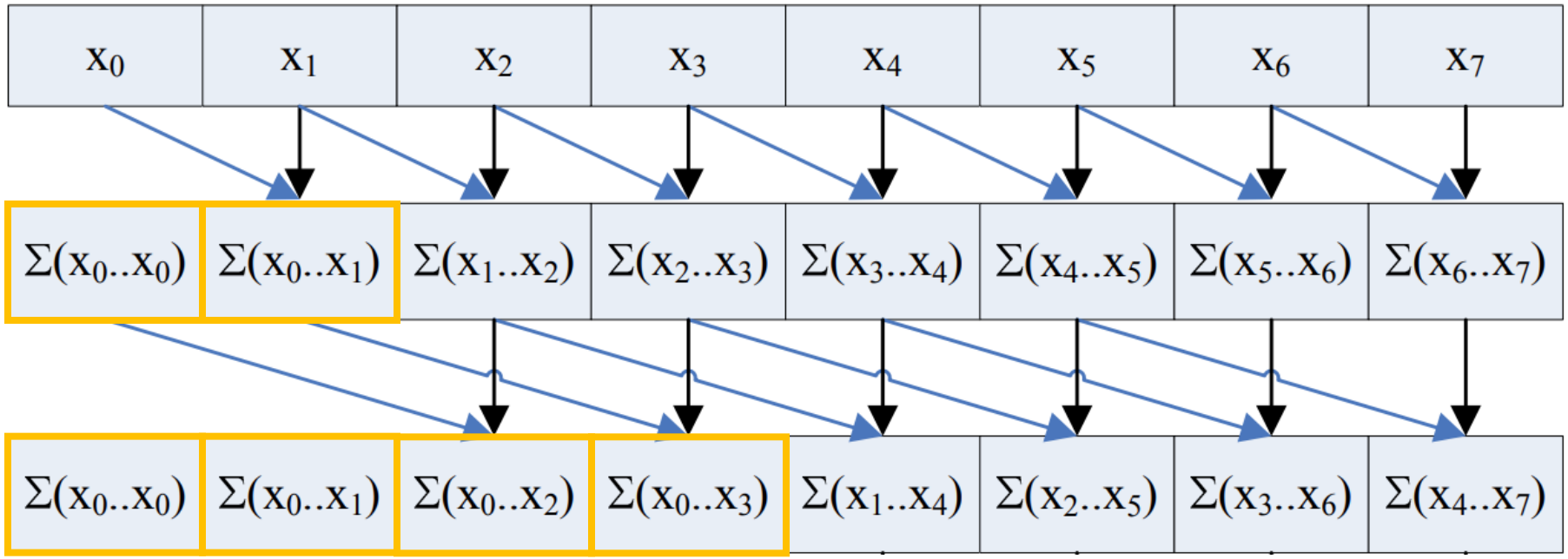
Pattern #5: Scan

iter 0, stride 2^0

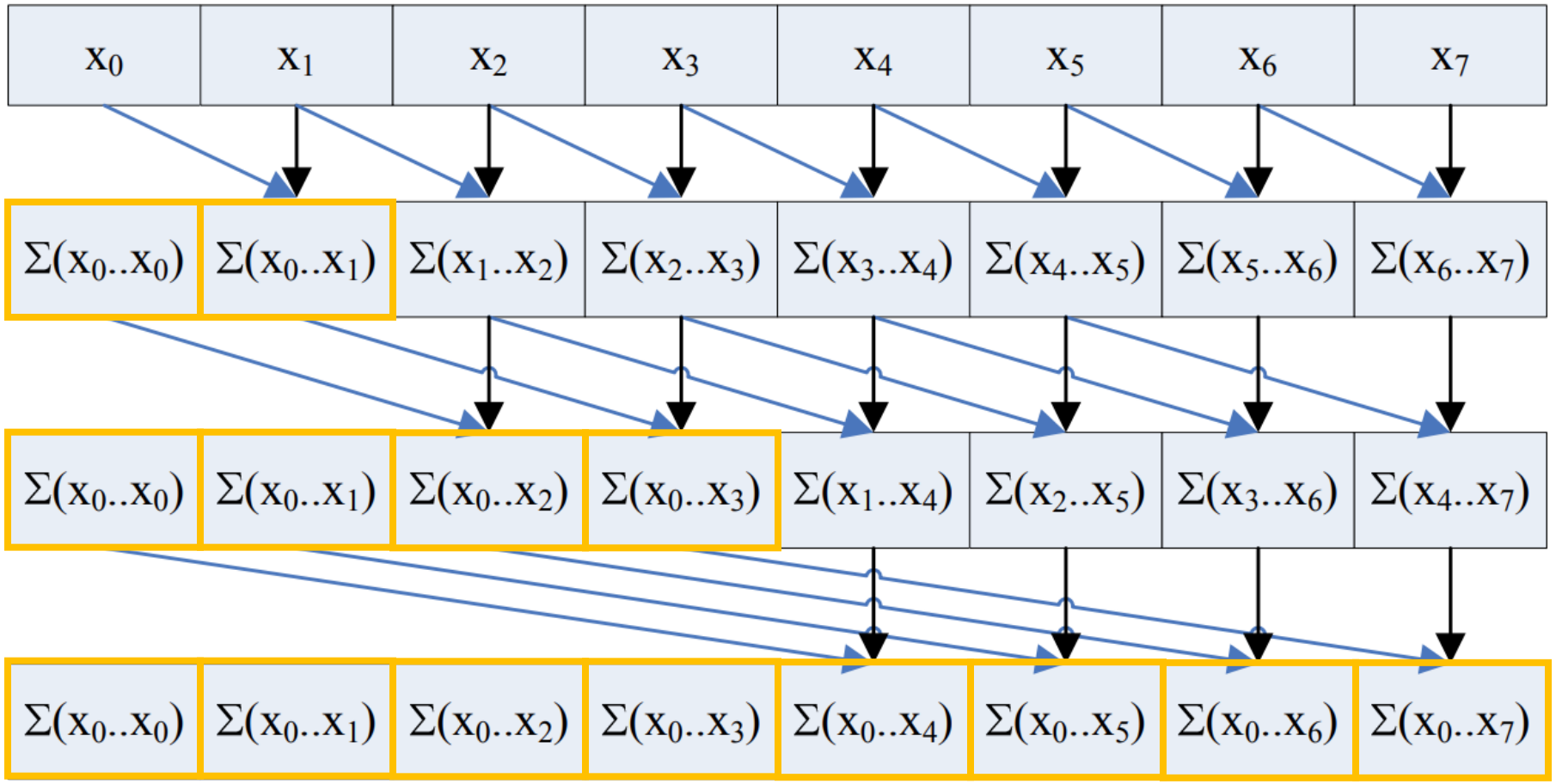


Pattern #5: Scan

iter 1, stride 2^1



Pattern #5: Scan

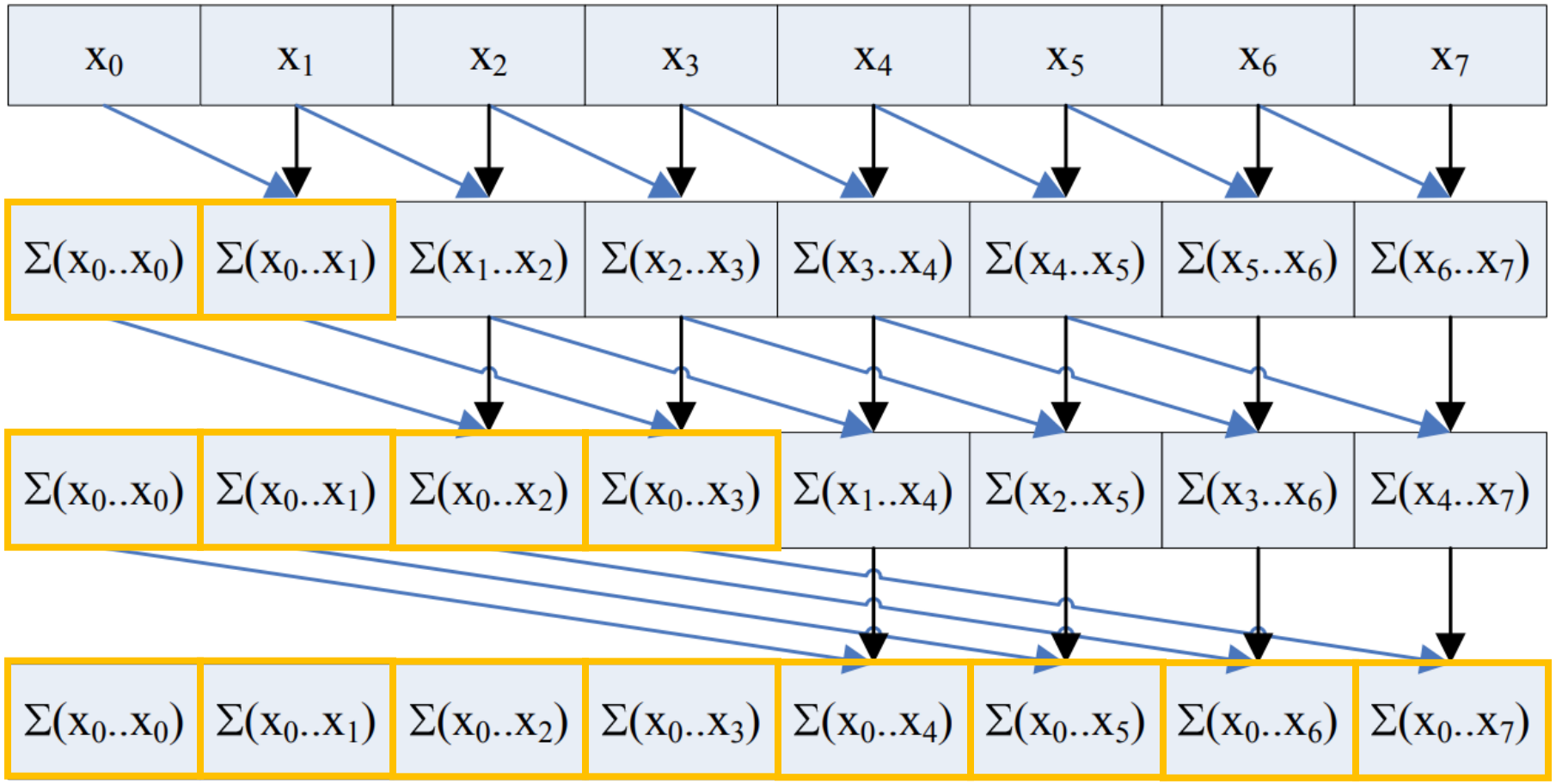


iter 2, stride 2^2

Pattern #5: Scan

- Kogge-Stone algorithm

$O(\log_2 N)$ Steps
$O(N \log_2 N)$ Work

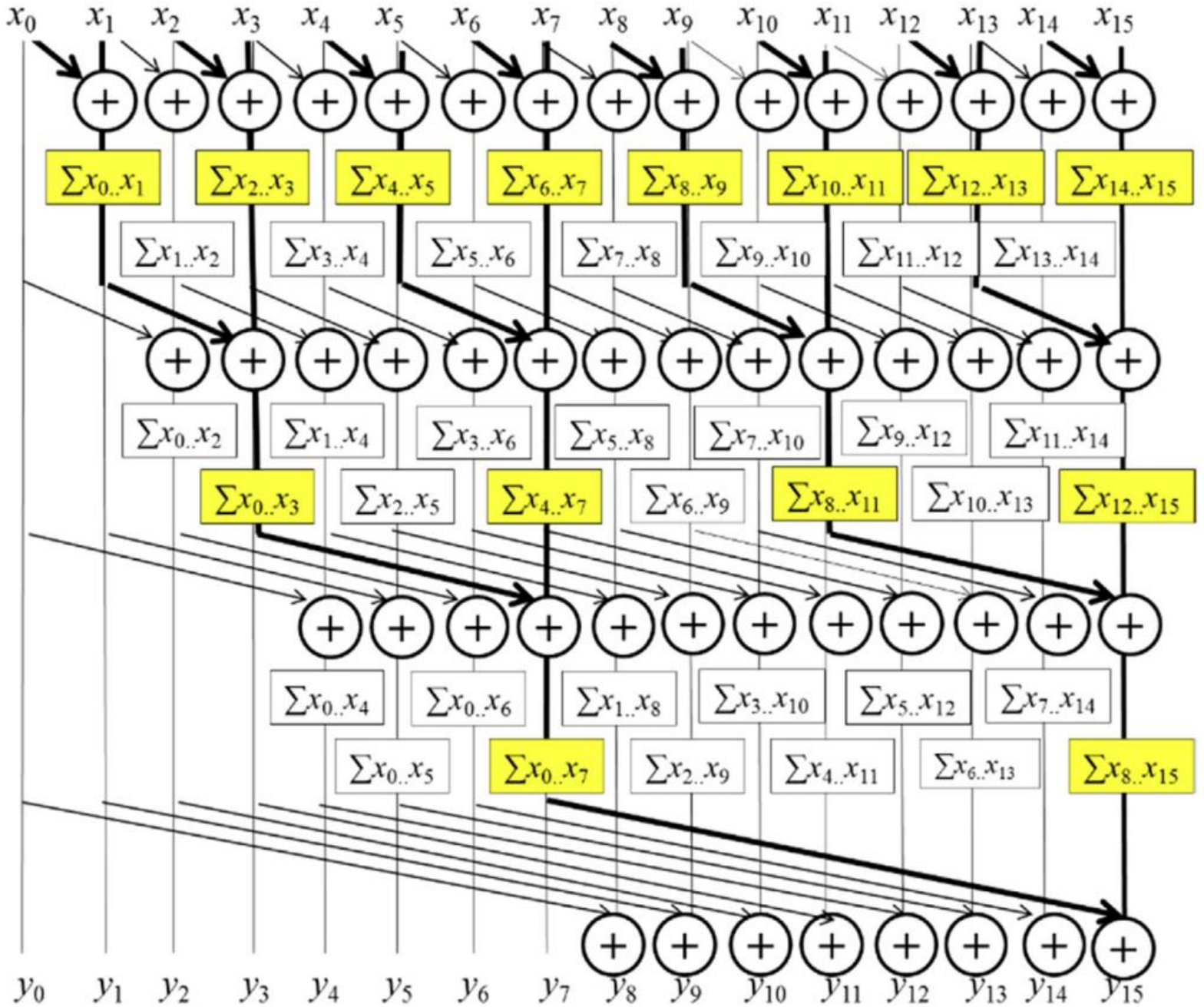


Step efficient

Not work efficient

Pattern #5: Scan

- Kogge-Stone algorithm



Pattern #5: Scan

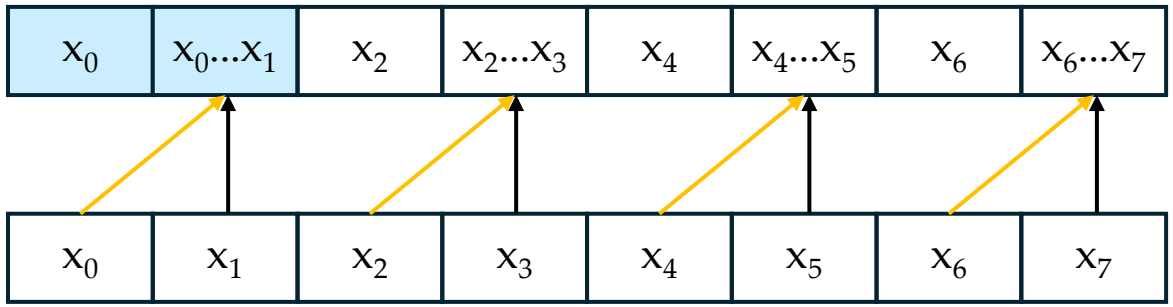
iter = 0

(tid+1)%2^1 == 0

stride = 2^0

Up-sweep Phase

Pattern #5: Scan



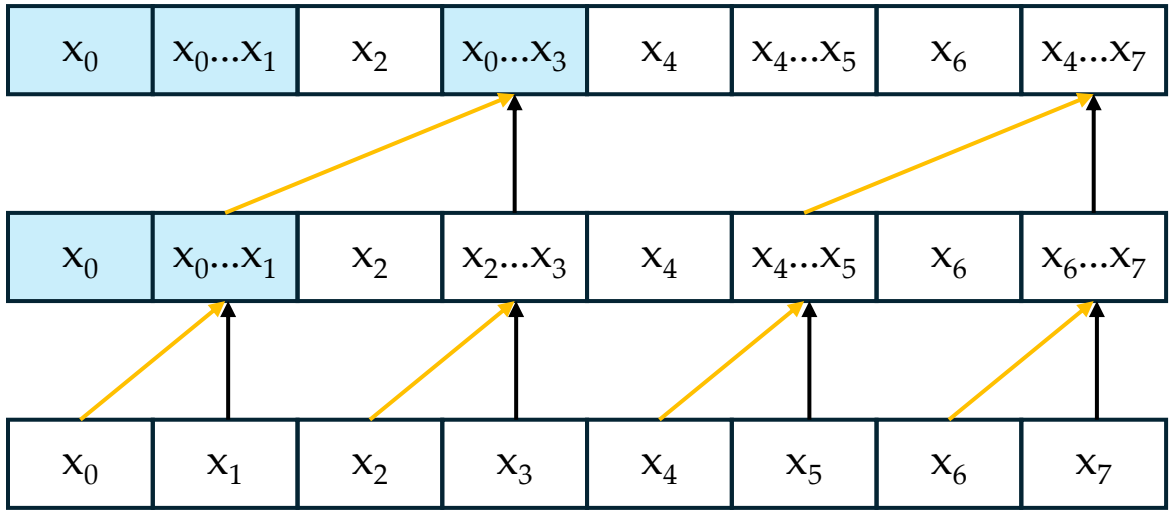
Up-sweep Phase

iter = 0

$(tid+1)\%2^1 == 0$

stride = 2^0

Pattern #5: Scan



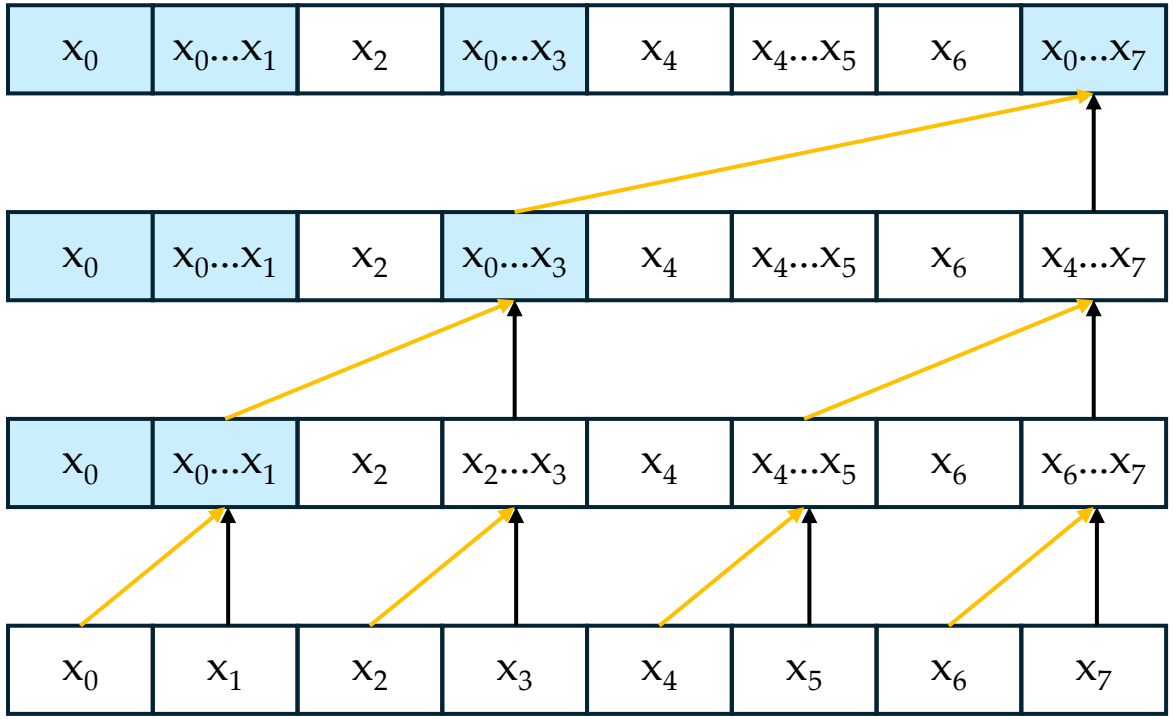
Up-sweep Phase

iter = 1

$(tid+1)\%2^2 == 0$

$stride = 2^1$

Pattern #5: Scan



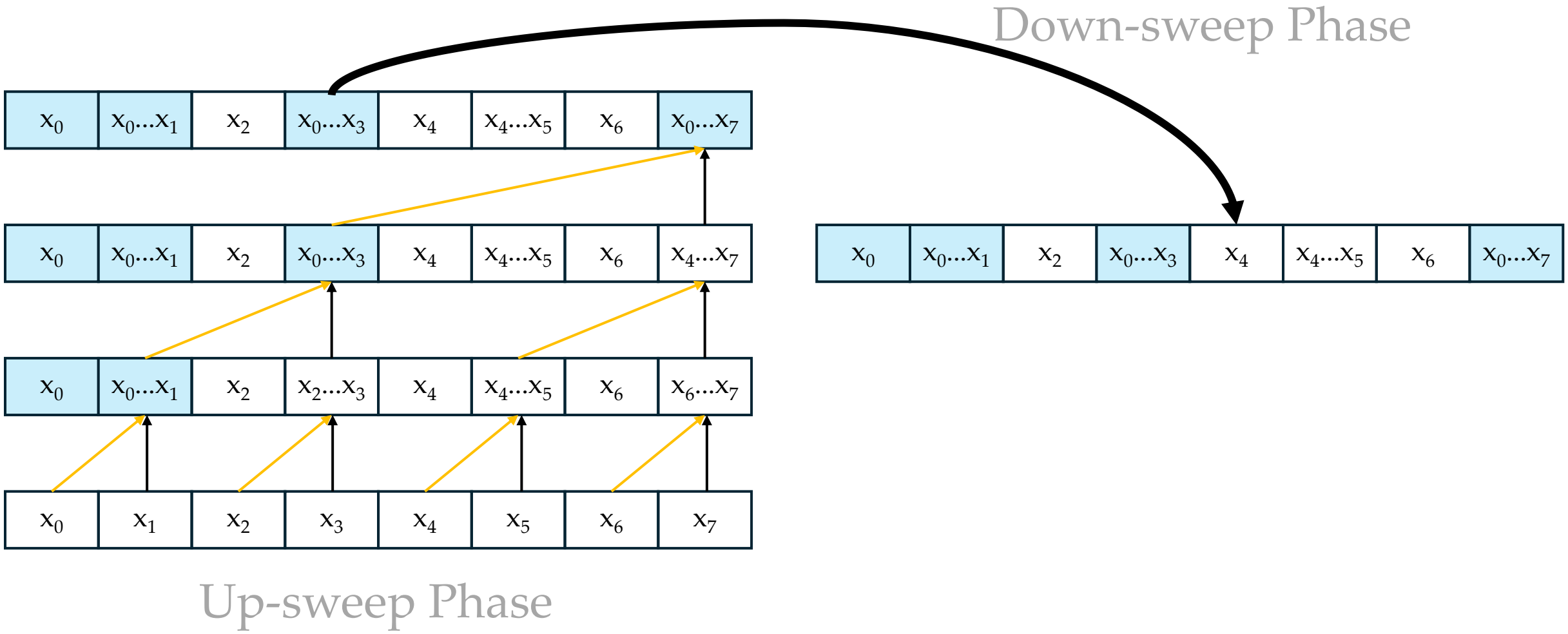
Up-sweep Phase

iter = 2

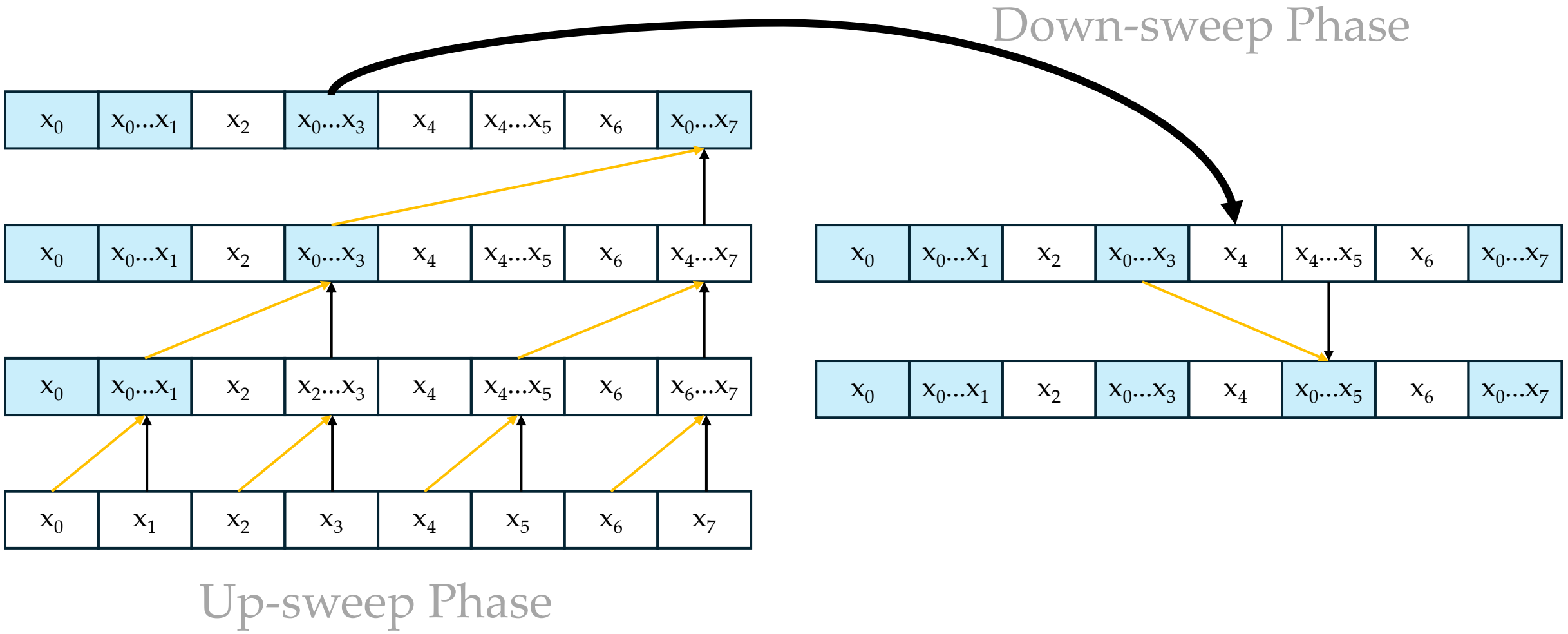
$(tid+1)\%2^3 == 0$

stride = 2^2

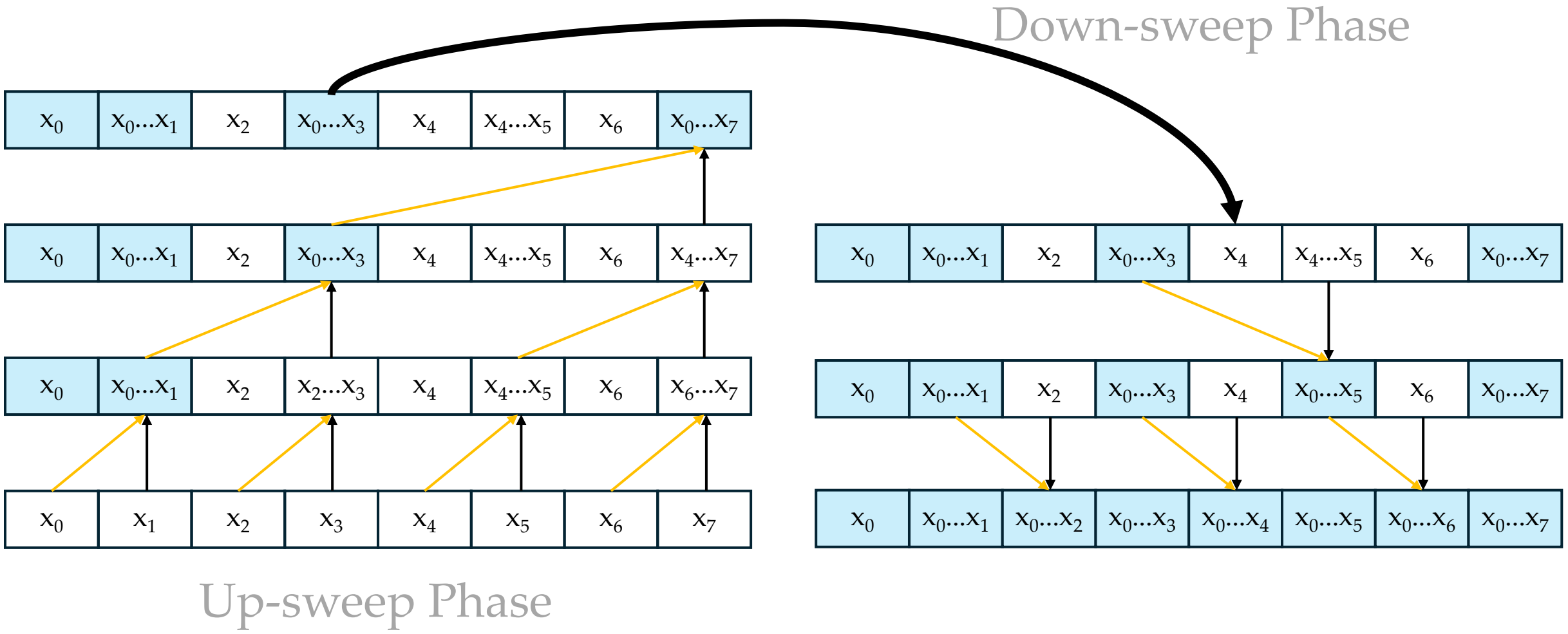
Pattern #5: Scan



Pattern #5: Scan

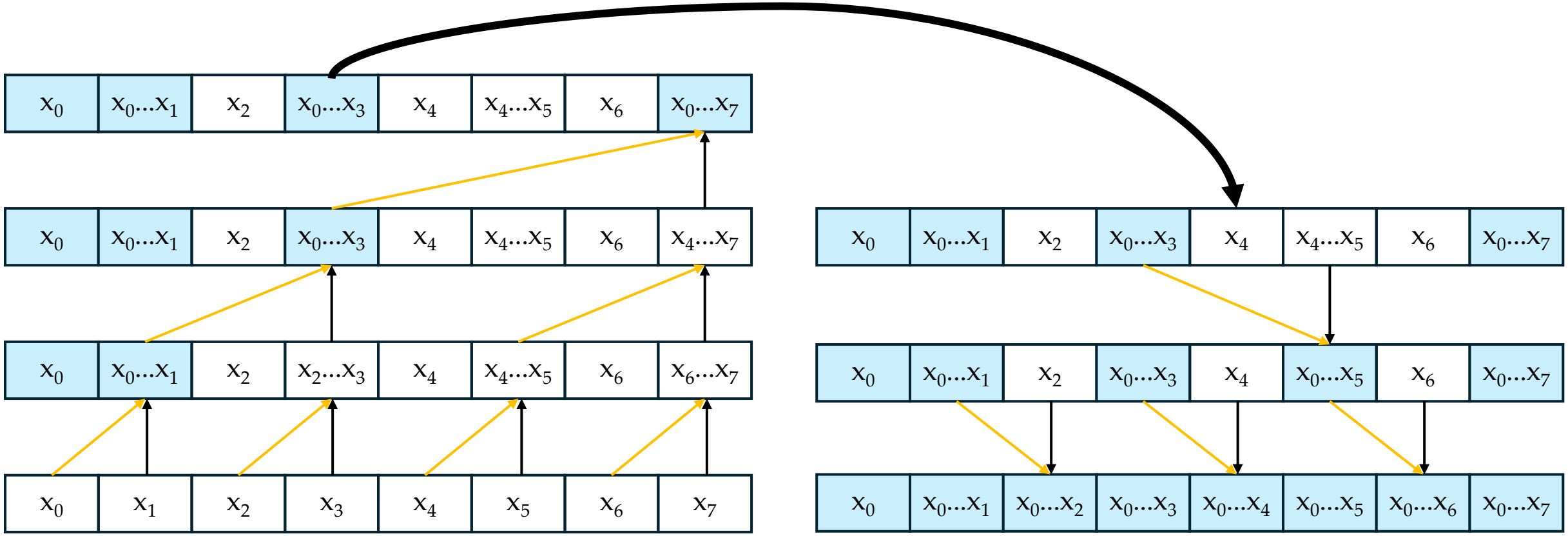


Pattern #5: Scan



Pattern #5: Scan

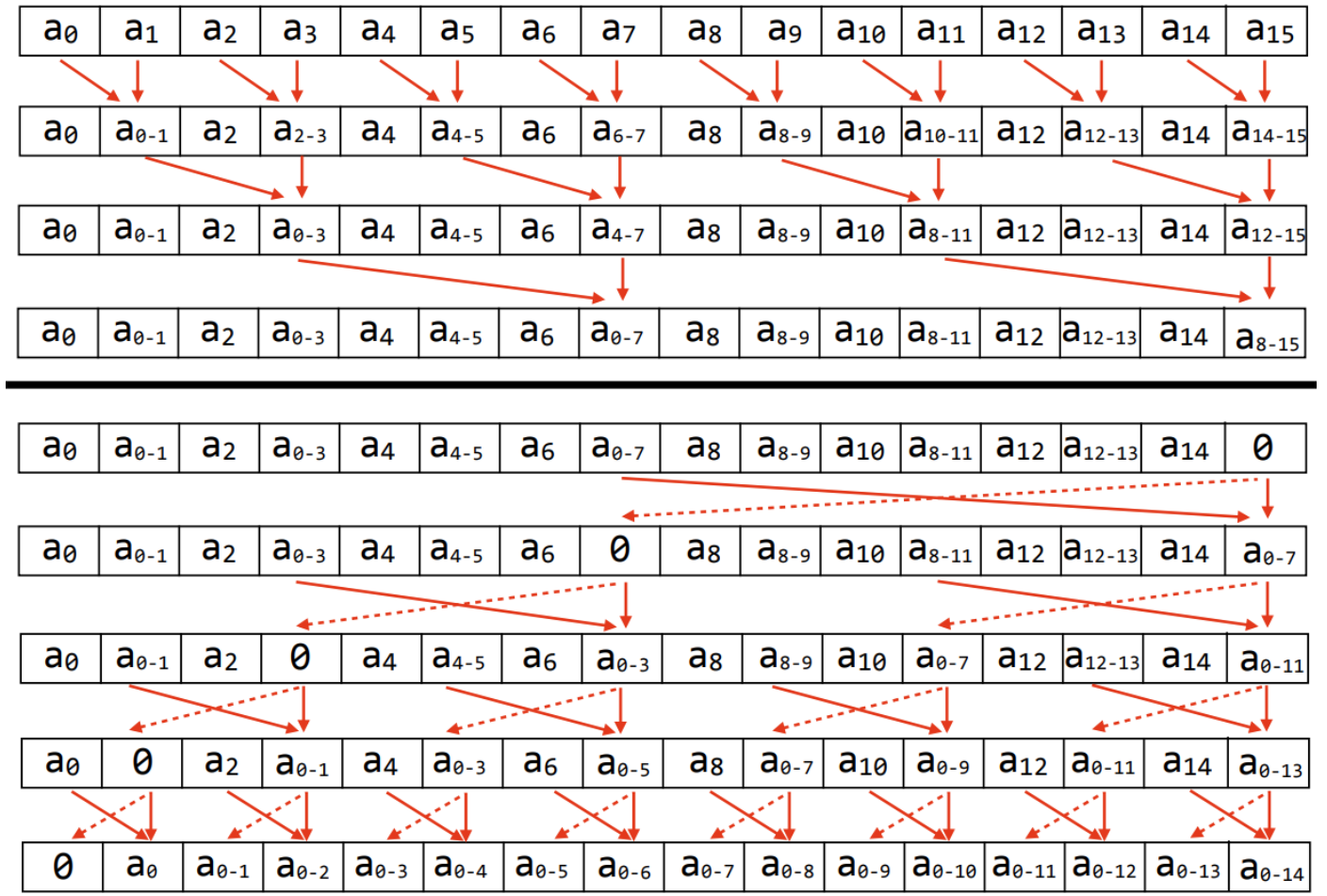
- Brent-Kung algorithm



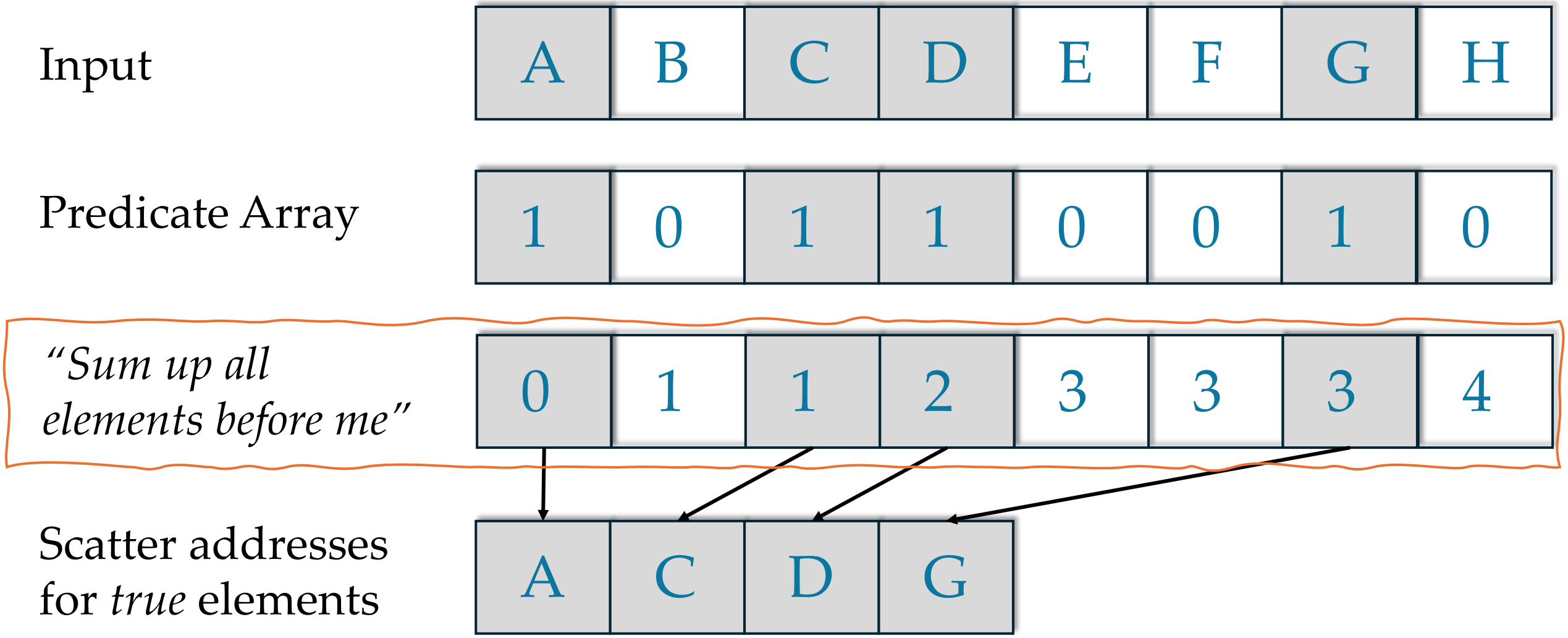
Pattern #5: Scan

- Improve work efficiency by reusing of computation results

$O(\log_2 N)$ Steps
$O(N)$ Work



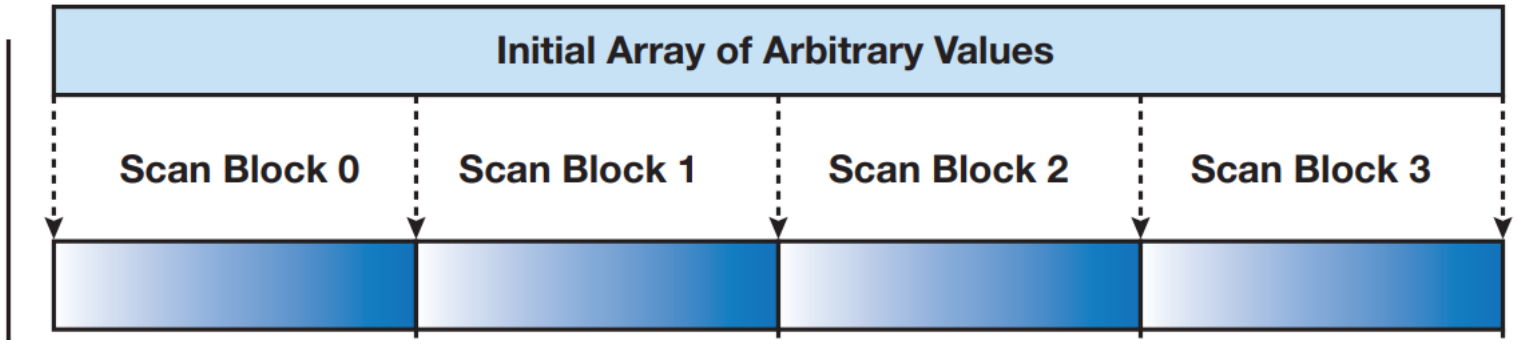
Back to *Stream Compaction*



Pattern #5: Scan

Scan across blocks

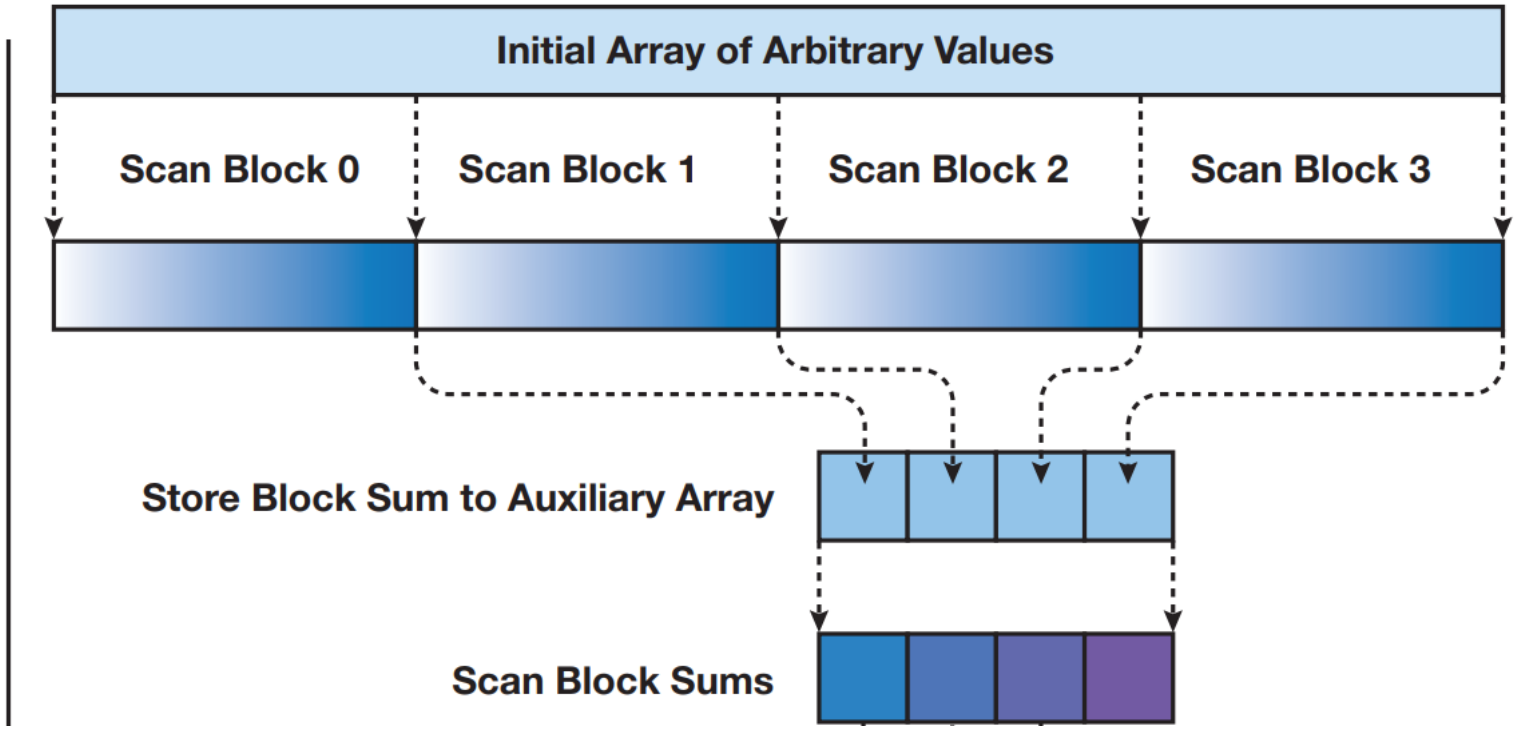
- Scan-and-add strategy



Pattern #5: Scan

Scan across blocks

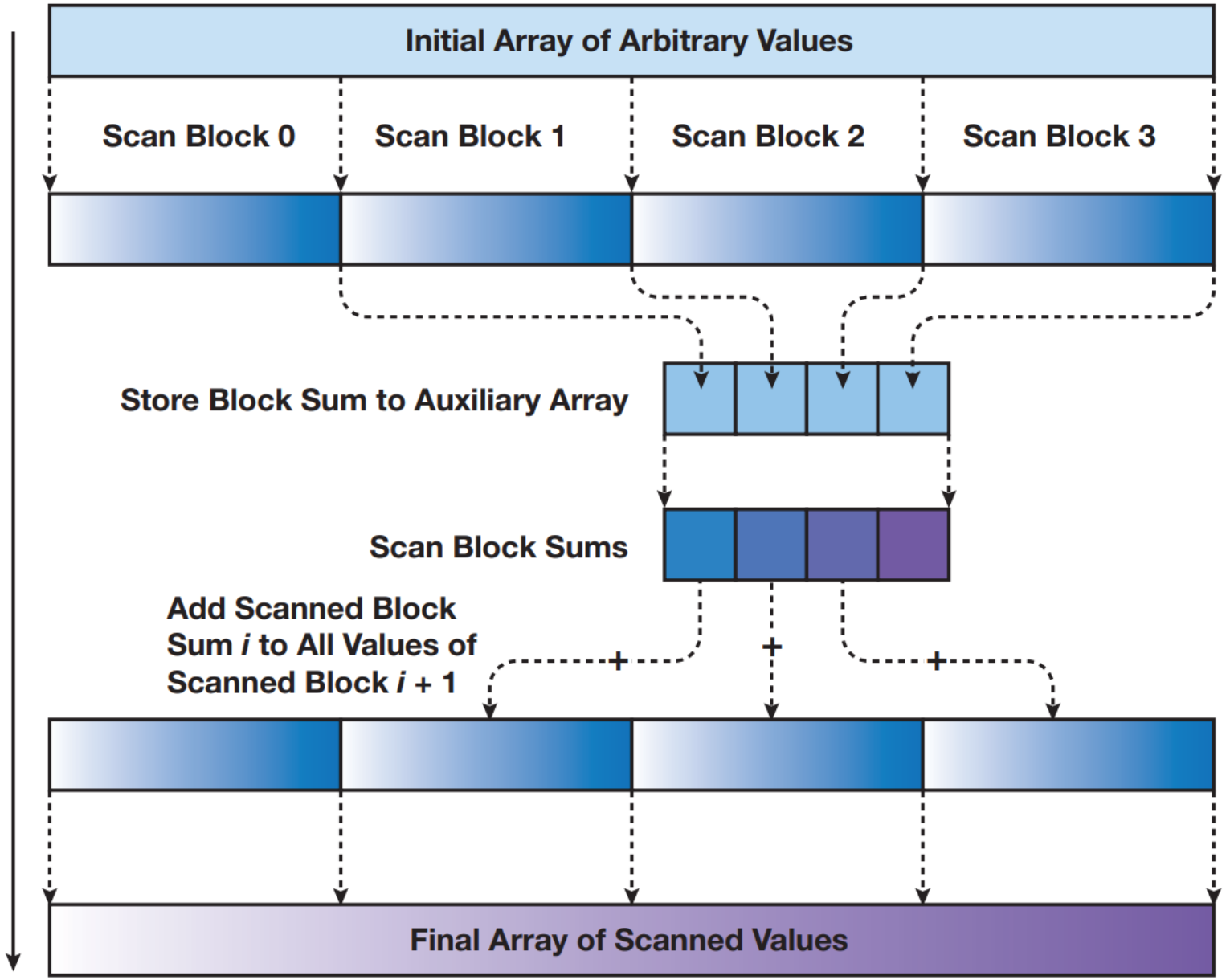
- Scan-and-add strategy



Pattern #5: Scan

Scan across blocks

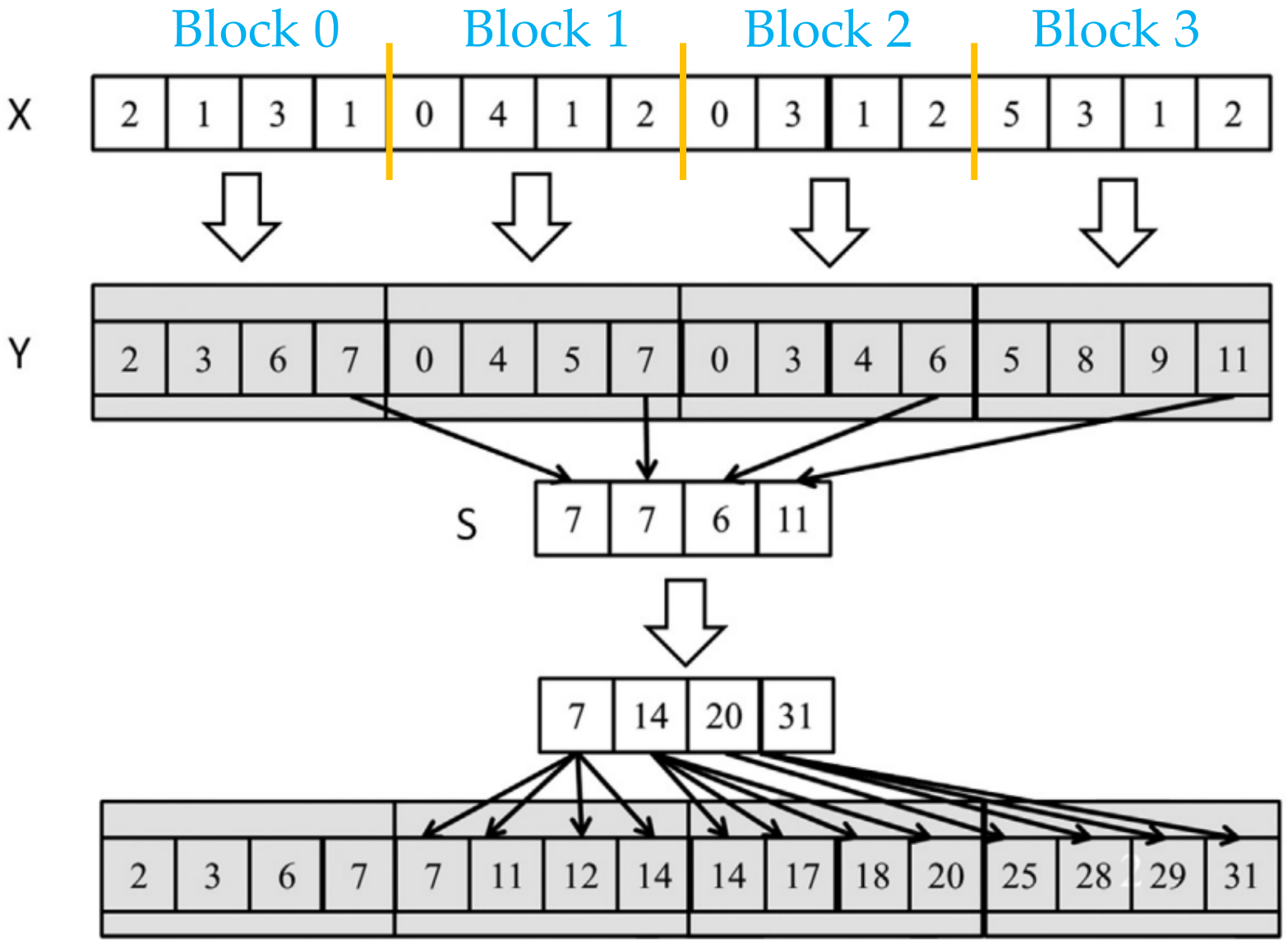
- Scan-and-add strategy



Pattern #5: Scan

Scan across blocks

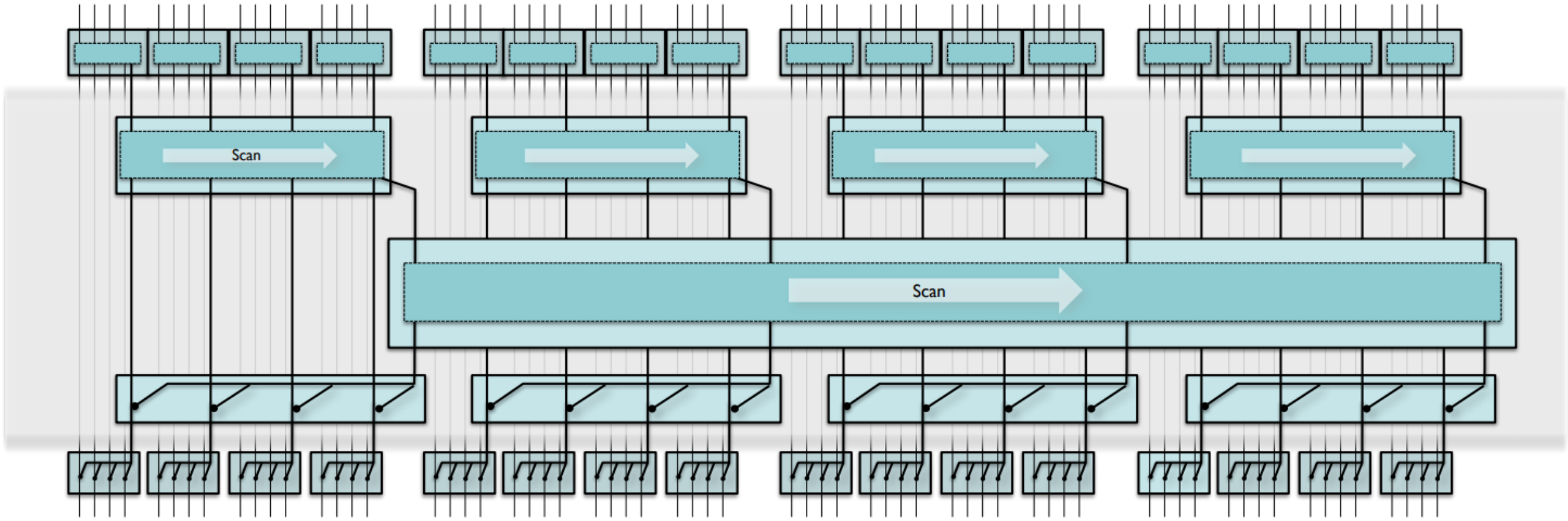
- Scan-and-add strategy



Pattern #5: Scan

Scan across blocks

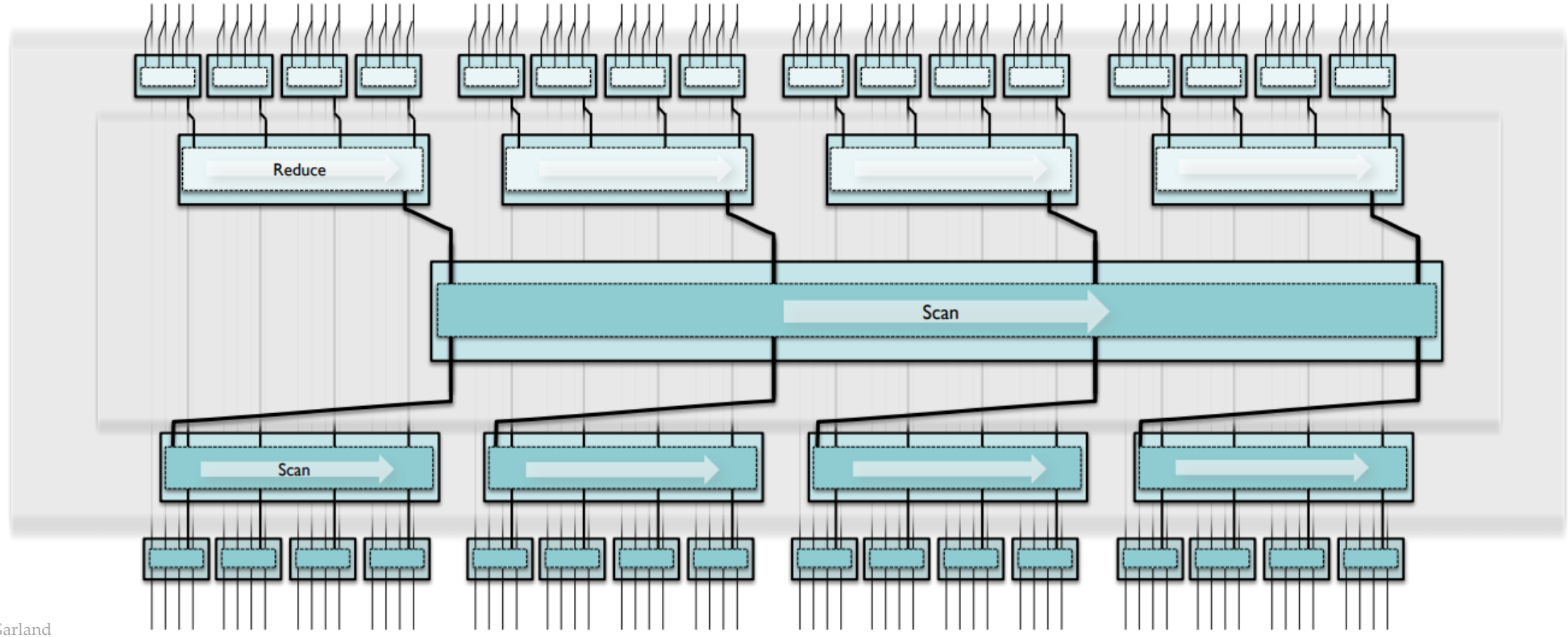
- Scan-and-add strategy *(4n read/write)*



Pattern #5: Scan

Scan across blocks

- Reduce-then-scan strategy *(3n read/write)*



Merrill, Garland
Single-pass Parallel Prefix Scan with Decoupled Look-back.
2013

Questions?!

Credits:

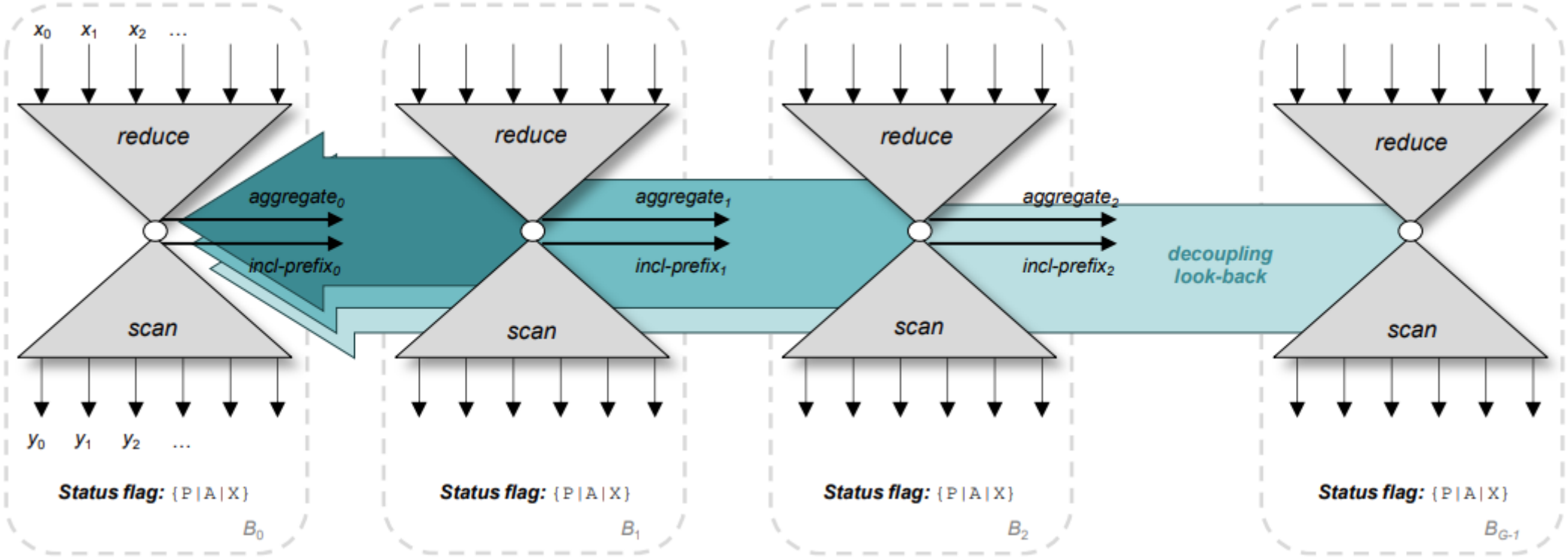
This lecture is primarily derived from:

- John Owens's course on Modern Parallel Computing (EEC 289Q, UC Davis, Winter 2018)
- Kayvon Fatahalian's course on Parallel Computing (CS149, Stanford, Fall 2023)
- Programming Massively Parallel Processors - A Hands-on Approach book, 4th edition by Wen-mei W. Hwu, David B. Kirk, and Izzat El Hajj, 2023

Pattern #5: Scan

Scan across blocks

- Decoupled look-back



Merrill, Garland
Single-pass Parallel Prefix Scan with Decoupled Look-back.
2013