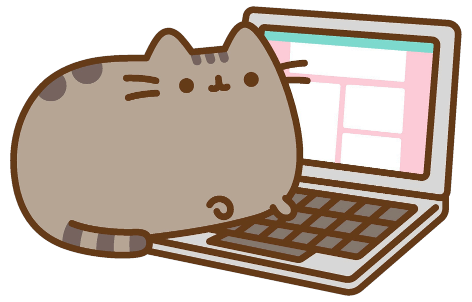
Announcements

HW3 (vectorization) out today - due Feb 15 (~1.5 weeks)

CSE 6230:
HPC Tools and Applications

+



Lecture 9: Data-Parallel Algorithms

Helen Xu

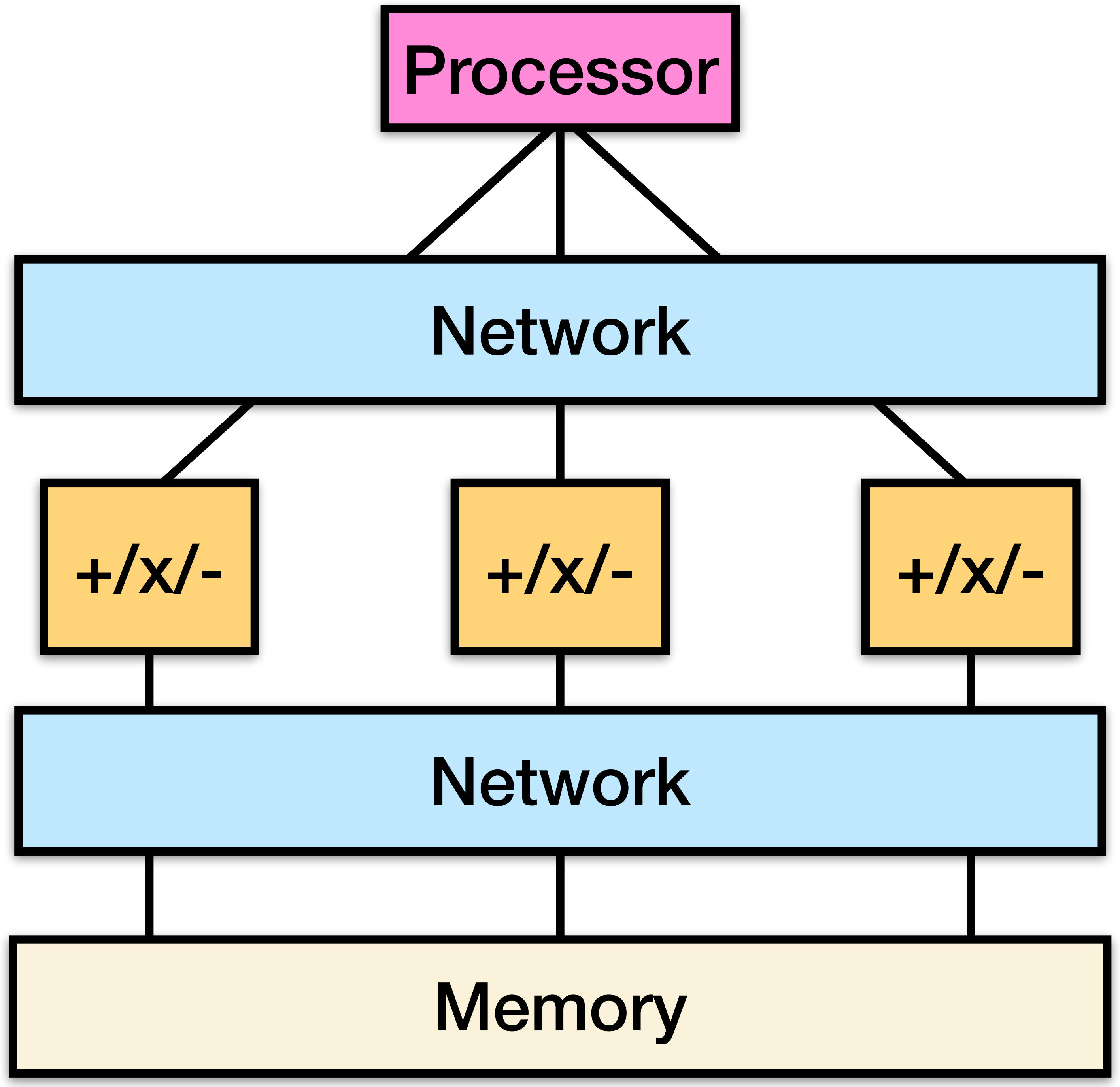
hxu615@gatech.edu

Georgia Tech College of Computing
School of Computational Science and Engineering

Single Instruction Multiple Data (SIMD)

SIMD machines run **one instruction stream** (all compute units run the same instruction).

The processing units communicate **through memory**.



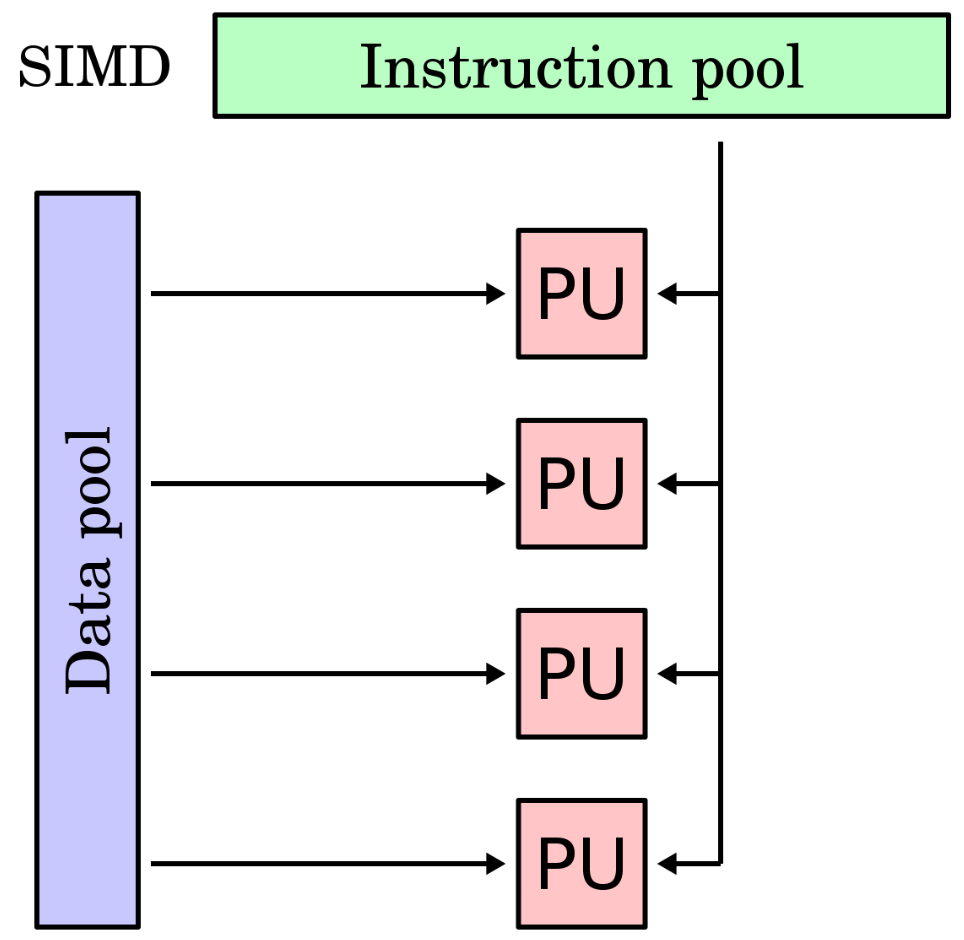
The Power of Data Parallelism

Data parallelism: perform **the same operation on multiple values** (often array elements)

- Also includes reductions, broadcast, scan…

Many **parallel programming models** use some data parallelism

- SIMD units (and previously SIMD supercomputers)
- CUDA / GPUs
- MapReduce
- MPI collectives



Data-Parallel Programming: Unary Operators

Unary operations applied to **all elements** of an array.

Example: squaring (or any unary function, i.e., with one argument)

A	3	1	1	2	3	3	4	2	2	2
B = f(A)	9	1	1	4	9	9	16	4	4	4

Data-Parallel Programming: Binary Operators

Binary operations applied to all **pairs** of elements.

Example: minus (or any binary operator)

A	3	1	0	2	3	0	4	2	0	2

minus applied to each pair

B	0	1	1	4	1	0	2	1	4	3
C	3	0	-1	-2	2	0	2	1	-4	-1

Data-Parallel Programming: Broadcast

Broadcast fill a value into all elements of an array

a = scalar
B = a

a = 3

B	3	3	3	3	3	3	3	3	3	3

Z = a*X + Y

a = 2

X	3	1	1	2	3	3	4	2	2	2
Y	1	2	0	2	1	3	1	2	0	2

↓ axpy

Z	7	4	2	6	7	9	9	6	4	6

{s, d}axpy is for single or double precision

From UC Berkeley CS267

Memory Operations: Assignment

Array assignment works if the arrays are the **same shape**:

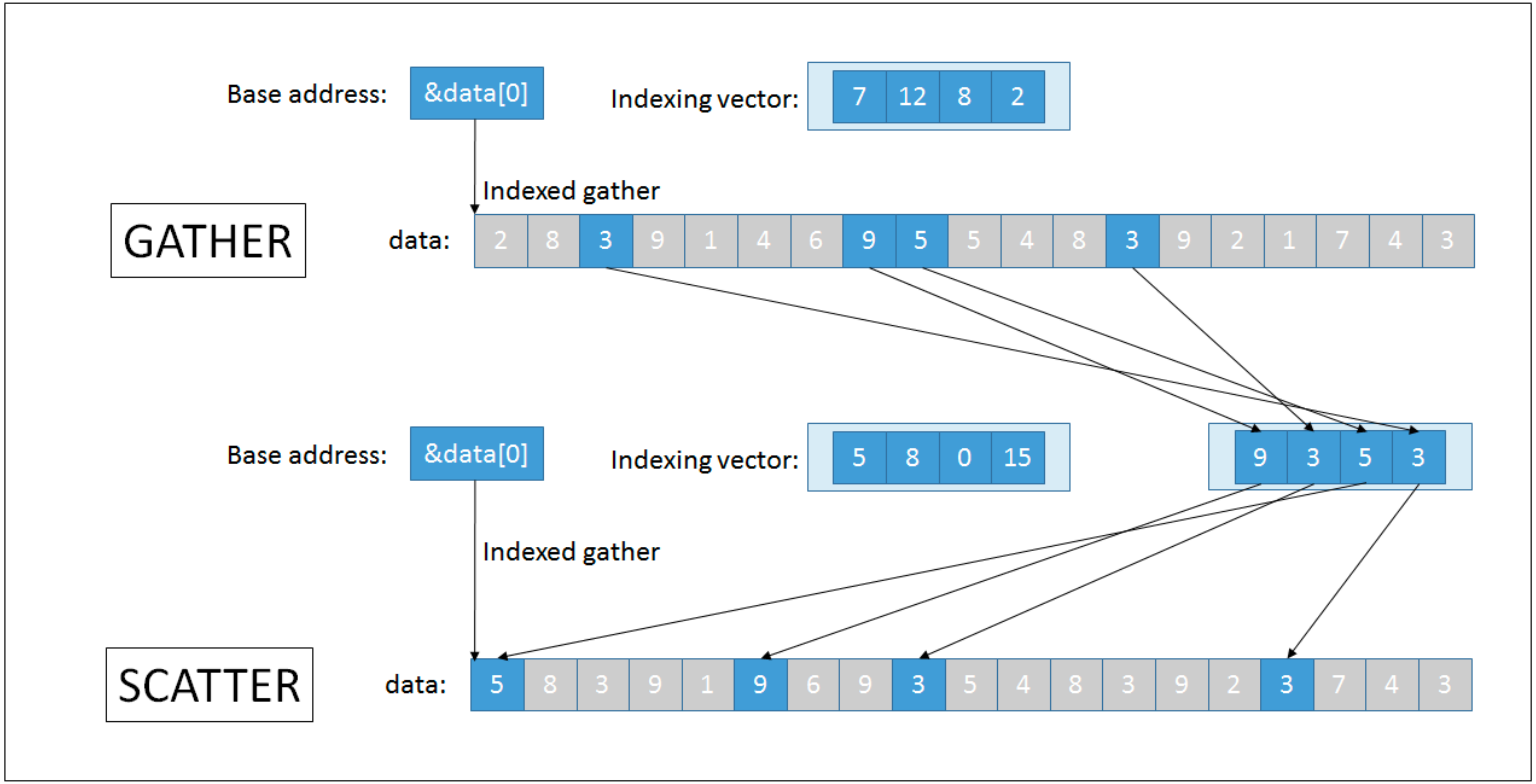
A = double[0:4] // 5 elements
B = double[0:4] = [0.0, 1.1, 2.2, 3.3, 4.4]
A = B

May have a **stride**: i.e., might not be contiguous in memory

A = B[0:4:2] // copy with stride 2
(every other element)

C = double[0:4, 0:4] // 5x5 mtx
A = C[*, 3] // copy column of C

Memory Operations: Scatter/Gather



Scatter/gather are often used in sparse linear algebra, sorting algorithms, FFT, etc.

https://gainperformance.wordpress.com/2017/06/15/umesimd-tutorial-9-gatherscatter-operations/

Data-Parallel Programming: Masks

Can apply operations under a bitmask

A	3	2	8	2	3	3	4	2	2	2
B	0	1	1	4	1	0	2	1	4	3
M	1	0	0	1	1	0	0	0	1	1

Array of 0/1 (True/false)

A = A+B under M	3	2	8	6	4	3	4	2	6	5

Data-Parallel Programming: Reduce

Reduce an array to a value with + or any **associative** op

A = array
b = scalar
b = sum(A)

A										
	3	1	1	2	3	3	4	2	2	2

b = 23

Useful for **dot products** (ddot, sdot, etc.)

b = dot(X, Y)
= sum(X*Y)

X:	1	1	1	3	3	2	1
Y:	1	2	0	2	1	3	1

Intermediate products

1	2	0	6	3	6	1

b = 19

Data-Parallel Programming: Scan

Input: an array $x = [x_0, x_1, \ldots, x_{n-1}]$ of n elements

Output: an array $y = [y_0, y_1, \ldots, y_{n-1}]$ of running sums, where

$$y_k = \begin{cases} x_0 & \text{if } k = 0 \\ x_k + y_{k-1} & \text{if } k \geq 1. \end{cases}$$

Data-Parallel Programming: Scan Examples

Fill array with **partial reductions** from any associative operation

A = array
B = array
B = scan(A, +)

A	3	1	1	2	3	3	4	2	2	2
B	3	4	5	7	10	13	17	19	21	23

A = array
B = array
B = scan(A, max)

A	3	1	1	2	3	3	4	2	2	2
B	3	3	3	3	3	3	4	4	4	4

Inclusive and Exclusive Scans

Inclusive scan: includes x_k when computing y_k - as in our previous examples.

Another variant: **exclusive scan** does not include x_k when computing y_k.

A = array
B = array
B = inclusive_scan(A, +)

C = array
C = exclusive_scan(A, +)

A	3	1	1	2	3	3	4	2	2	2
B	3	4	5	7	10	13	17	19	21	23
C	0	3	4	5	7	10	13	17	19	21

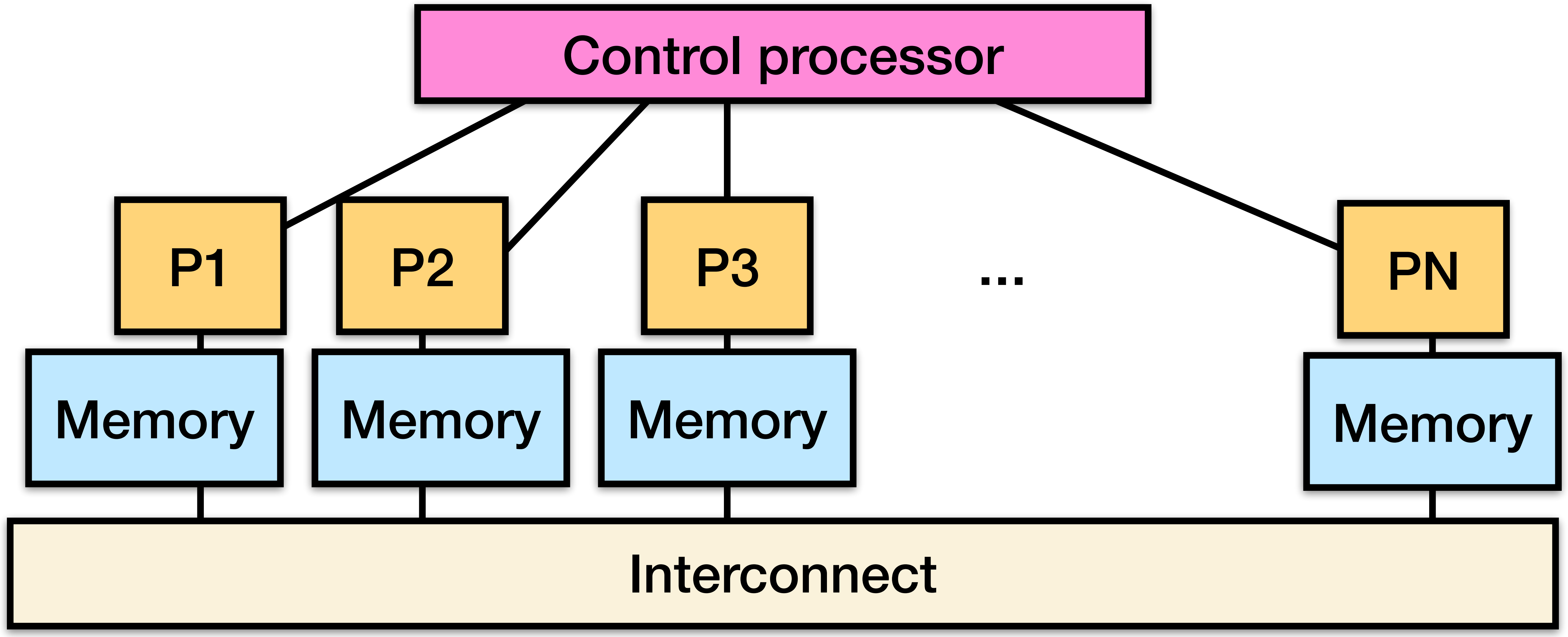
Can easily get the inclusive version from the exclusive by adding the input element-wise.

For the other way, you need an inverse for the operator.

Idealized Hardware and Performance Model

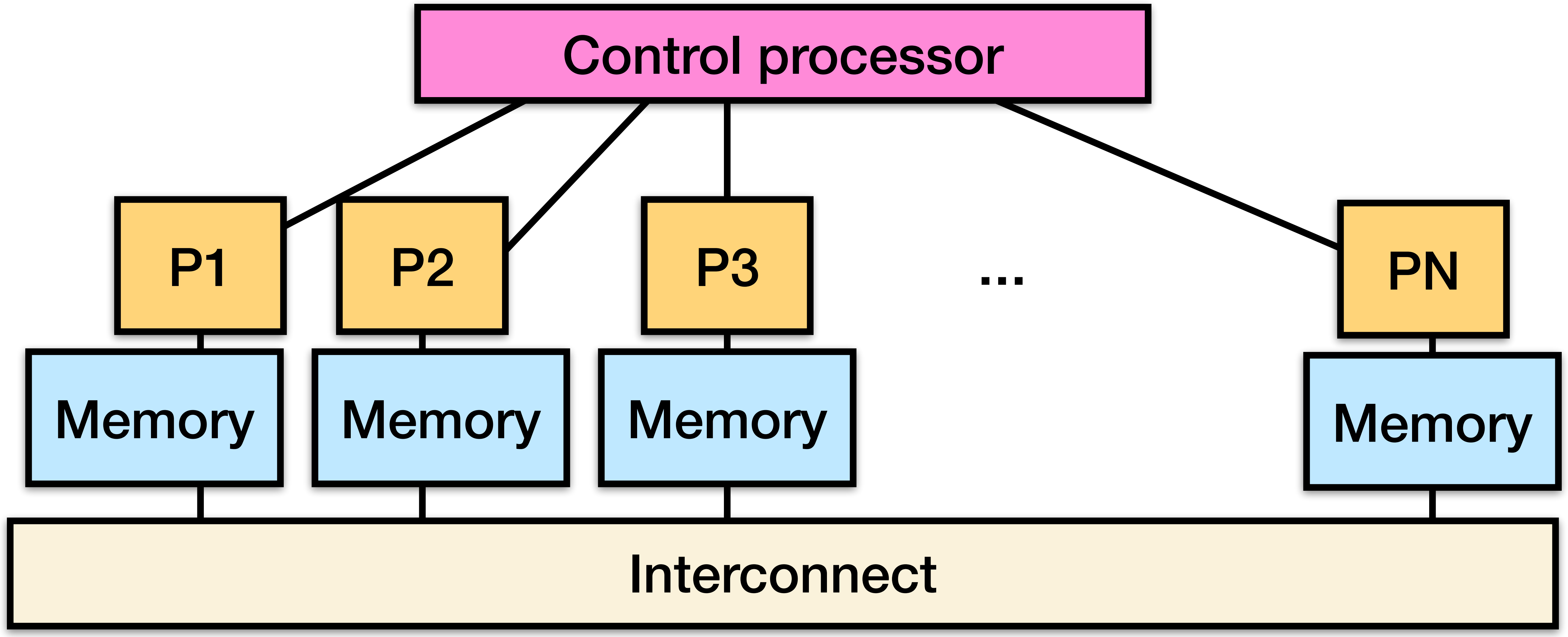
SIMD Systems Implemented Data Parallelism

- A SIMD machine has a large number of (usually) tiny processors
  - A single “control processor” issues each instruction
  - Each processor executes the same instruction
  - Some processors may be turned off on some instructions



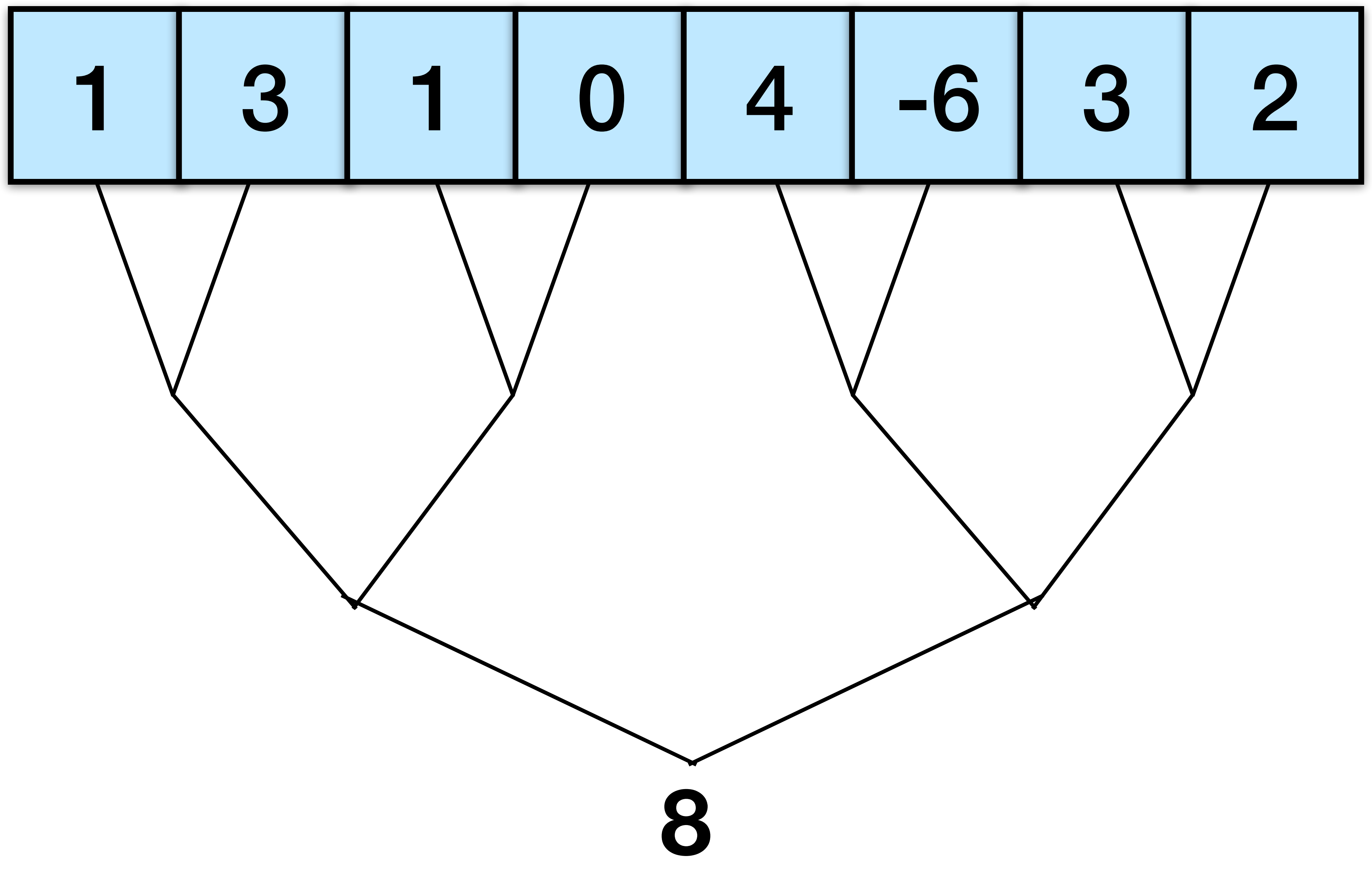
Ideal Cost Model for Data Parallelism

- Machine:
  - An unbounded number of processors (p)
  - Control overhead is free
  - Communication is free
- Cost (complexity) on this abstract machine is the algorithm's **span** T_∞



Reduction on Processor Tree

- Reduction of n values to 1 with $\log(n)$ span.
- Takes advantage of associativity in +, *, min, max, etc.

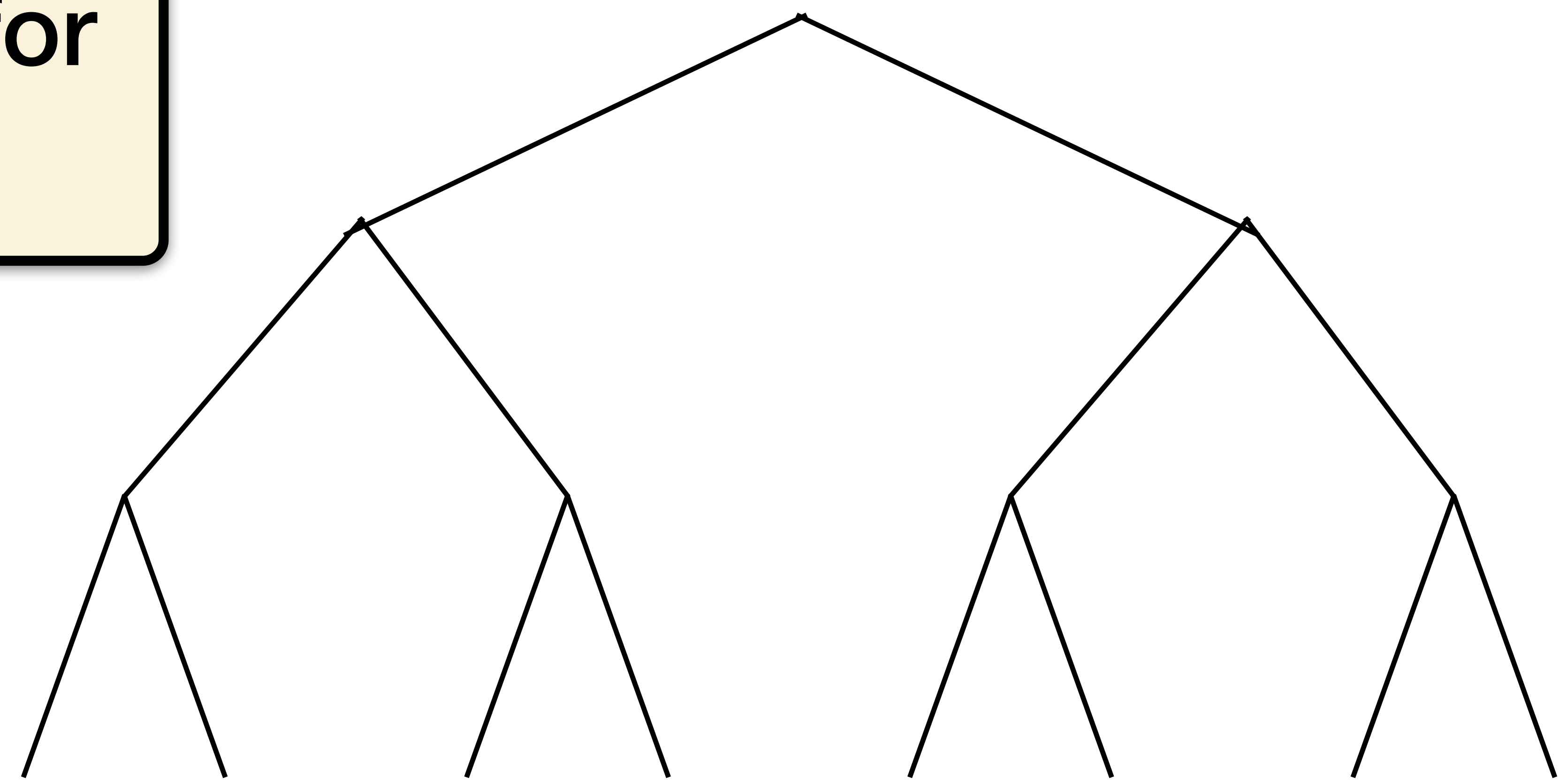


Reduction Lower Bound

Given a function $f(x_1, \ldots, x_n)$ of n input variables and 1 input variable, how fast can we evaluate it in parallel?

- Assume we only have binary operations, one per time step
- After 1 time step, an output can depend on two inputs.
- Therefore, by induction, after k time units, an output can depend on at most 2^k inputs.

Binary tree performs such a computation for $k = \log(n)$



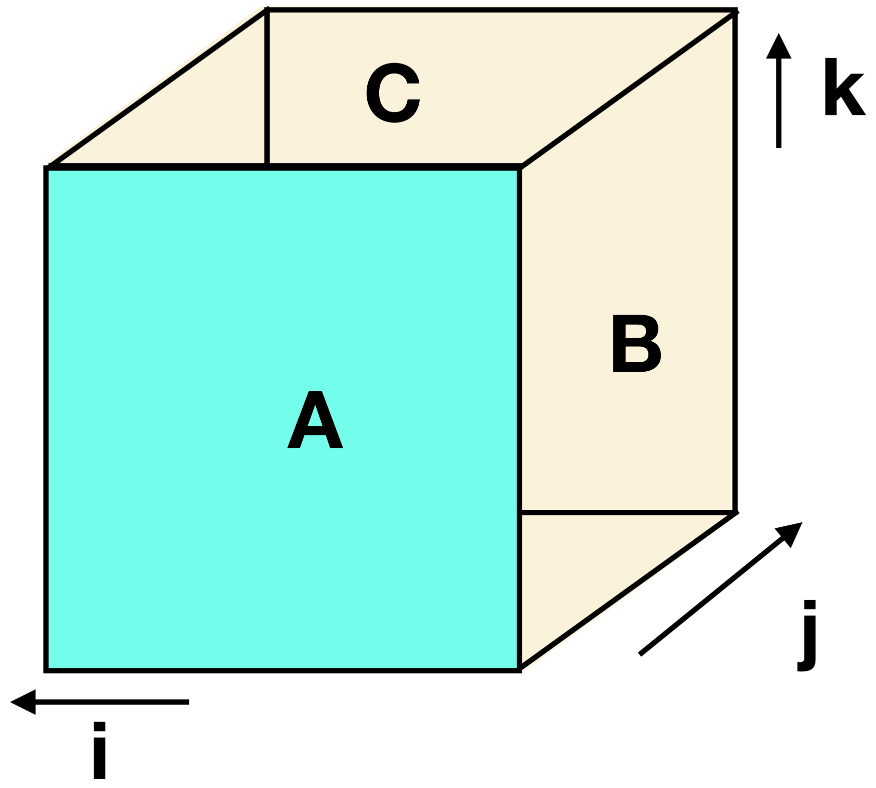
Multiplying n-by-n matrices on O(log n) span

Step 1: For all $1 \leq i, j, k \leq n$, do $P(i,j,k) = A(i,k) \times B(k,j)$

Adds 1 to span, using n^3 processors

Step 2: For all $1 \leq i, j \leq n$, do $C(i,j) = \sum_{k=1}^{n} P(i,j,k)$

Adds $\log(n)$ to span, using n^2 trees, each with n/2 processors



From UC Berkeley CS267

Parallel Prefix (Scan)

Can we parallelize a scan?

Serial scan takes n-1 operations.

The i-th iteration iteration of the loop **depends completely** on the (i-1)-th iteration.

```
y[0] = 0;
for (size_t i = 1; i < n; i++) {
  y[i] = y[i-1] + x[i];
}
```

First try: Parallel But Inefficient

Apply tree-like reduction at every element: put 1 processor at element 1, 2 at element 2, etc.

Span: $\lg(n)$

Work: $O(n^2)$

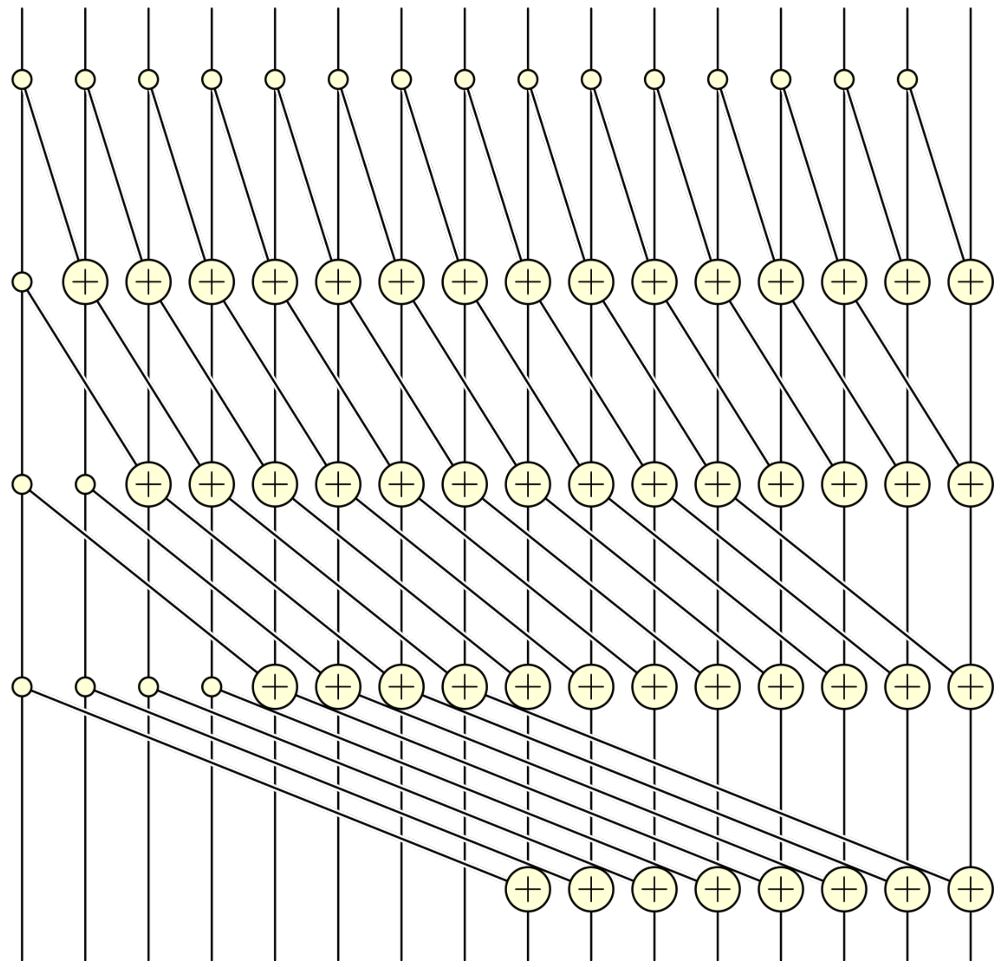
Work-efficient parallel algorithms perform **no more than a constant factor of work** over the best serial algorithm for the problem

A	1	2	3	4	5	6	7	8
B	1	3	6	10	15	21	28	36

Hillis-Steele Prefix Sum

for i = 0 up to $\log(n)$:
  for j = 0 up to n-1:
    if j < 2^i:
      $x_j^{i+1} \leftarrow x_j^i$
    else:
      $x_j^{i+1} \leftarrow x_j^i + x_{j-2^i}^i$

What is the work and span?



https://en.wikipedia.org/wiki/Prefix_sum

"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Hillis-Steele Prefix Sum

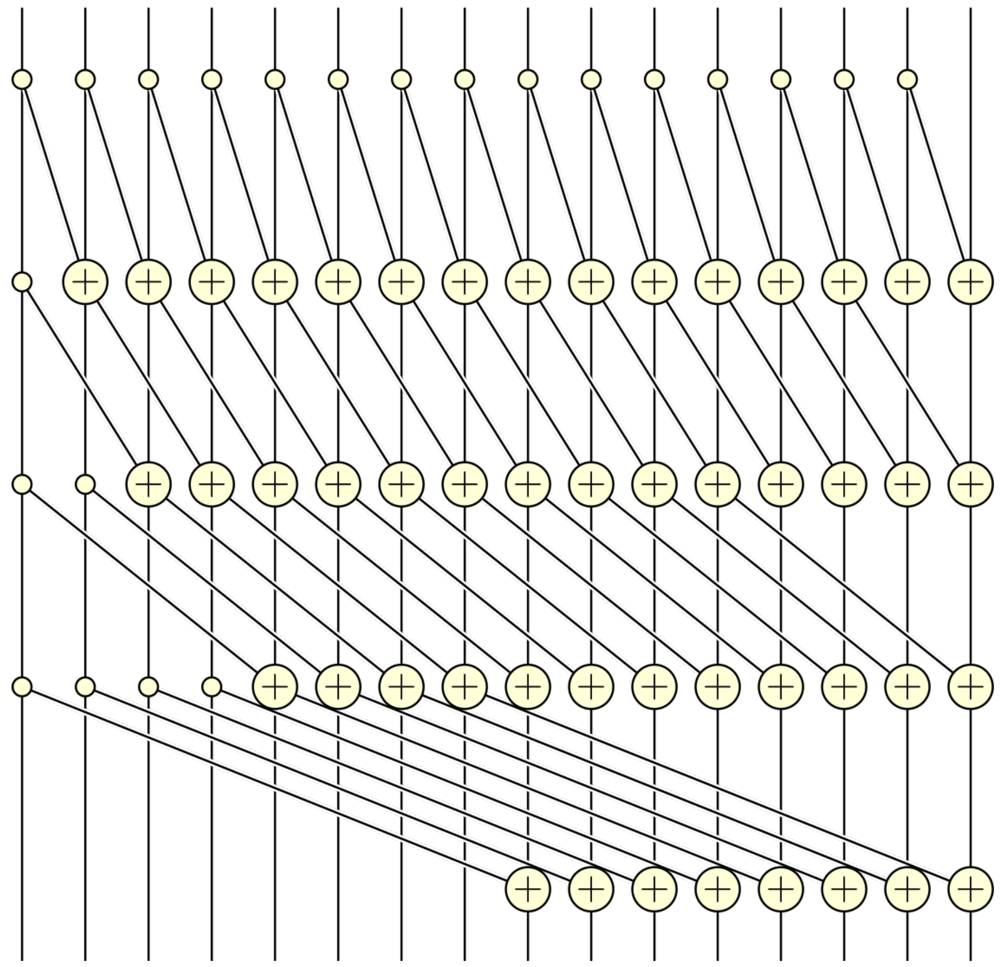
for i = 0 up to $\log(n)$:
  for j = 0 up to n-1:
    if j < 2^i:
      $x_j^{i+1} \leftarrow x_j^i$
    else:
      $x_j^{i+1} \leftarrow x_j^i + x_{j-2^i}^i$

What is the work and span?

Work = $O(n \lg n)$

Span = $O(\lg n)$

Better, but still not work efficient



https://en.wikipedia.org/wiki/Prefix_sum

"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Work-Efficient Parallel Prefix

Idea: Save the partial sums computed via parallel reduction (**upsweep**) and use those values in a **downsweep** pass to compute the total prefix.

The downsweep works by performing sums **down the prefix-sum tree**: at each step, each vertex at a given level passes its own value to its left child, and its right child gets the sum of the left child and the parent.

Exclusive scan:

	Step	Vector in Memory							
	0	[$\boxed{3}$	$\boxed{1}$	$\boxed{7}$	$\boxed{0}$	$\boxed{4}$	$\boxed{1}$	$\boxed{6}$	$\boxed{3}$]
up	1	[3	$\boxed{4}$	7	$\boxed{7}$	4	$\boxed{5}$	6	$\boxed{9}$]
	2	[3	4	7	$\boxed{11}$	4	5	6	$\boxed{14}$]
	3	[3	4	7	11	4	5	6	$\boxed{25}$]
clear	4	[3	4	7	11	4	5	6	$\boxed{0}$]
down	5	[3	4	7	$\boxed{0}$	4	5	6	$\boxed{11}$]
	6	[3	$\boxed{0}$	7	$\boxed{4}$	4	$\boxed{11}$	6	$\boxed{16}$]
	7	[$\boxed{0}$	$\boxed{3}$	$\boxed{4}$	$\boxed{11}$	$\boxed{11}$	$\boxed{15}$	$\boxed{16}$	$\boxed{22}$]

sum[v] = sum[L[v]] + sum[R[v]]

scan[L[v]] = scan[v]
scan[R[v]] = sum[L[v]] + scan[v]

https://en.wikipedia.org/wiki/Prefix_sum

"Prefix Sums and Their Applications," Blelloch, 1990.

Upsweep Example

```
[13,   9,  -4,  19,  -6,   2,   6,   3]

[13,  22,  -4,  15,  -6,  -4,   6,   9]

[13,  22,  -4,  37,  -6,  -4,   6,   5]

[13,  22,  -4,  37,  -6,  -4,   6,  42]
```

Downsweep Example

For an inclusive scan, only the downsweep needs to change:

```
[13,   22,   -4,   37,   -6,   -4,    6,   42]

[13,   22,   -4,   37,   -6,   33,    6,   42]

[13,   22,   18,   37,   31,   33,   39,   42]
```

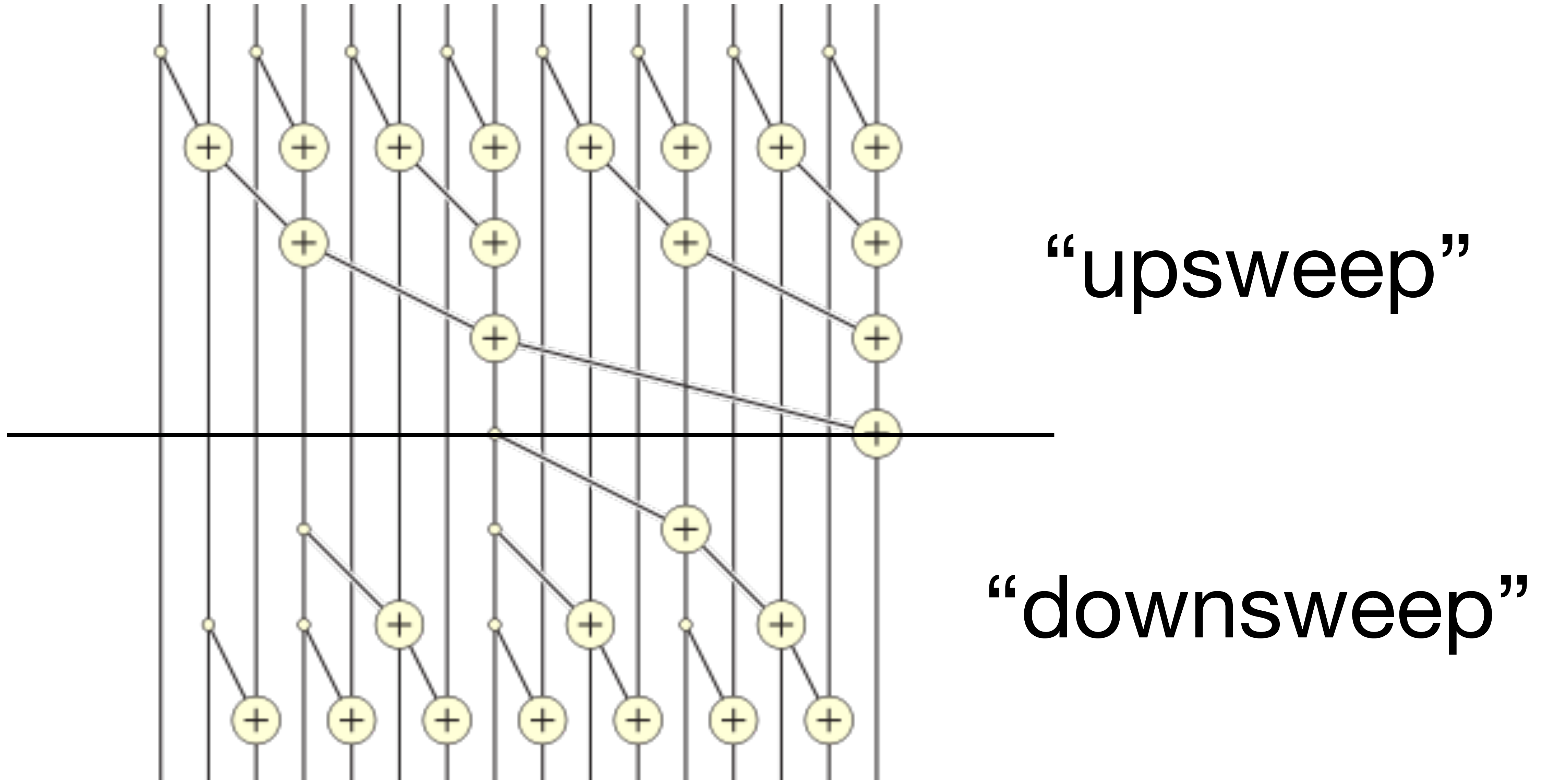
- Recall, we started with:

```
[13,    9,   -4,   19,   -6,    2,    6,    3]
```

Work-Efficient Parallel Prefix Pseudocode

Upsweep:
for d from 0 to $\lg(n) - 1$:
  parallel_for i from 0 to $n - 1$, i += 2^{d+1}:
    $A[i + 2^{d+1} - 1] \leftarrow A[i + 2^d - 1] + A[i + 2^{d+1} - 1]$

Downsweep:
for d from $\lg(n) - 1$ to 0:
  parallel_for i from $2^d - 1$ to $n - 1 - 2^d$, i += 2^{d+1}:
    if $i - 2^d \geq 0$:
      $A[i] = A[i] + A[i - 2^d]$

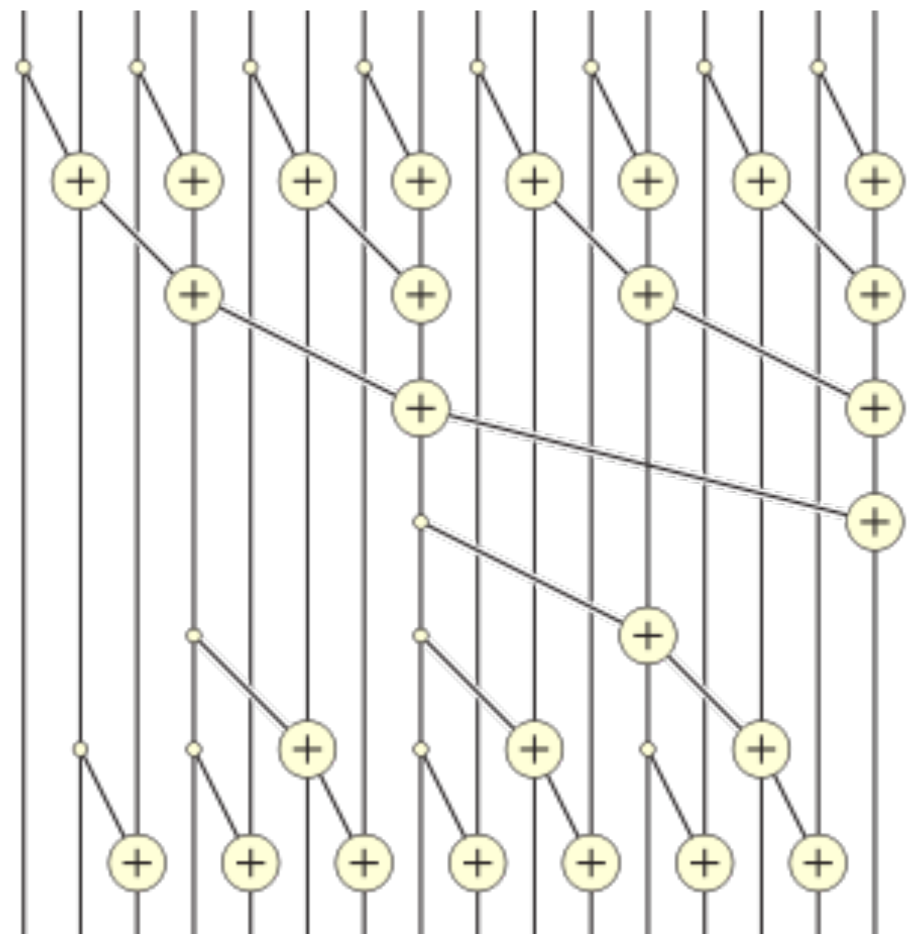


https://en.wikipedia.org/wiki/Prefix_sum

“Prefix Sums and Their Applications,” Blelloch, 1990.

Analysis of Parallel Prefix

What is the work and span?



Analysis of Parallel Prefix

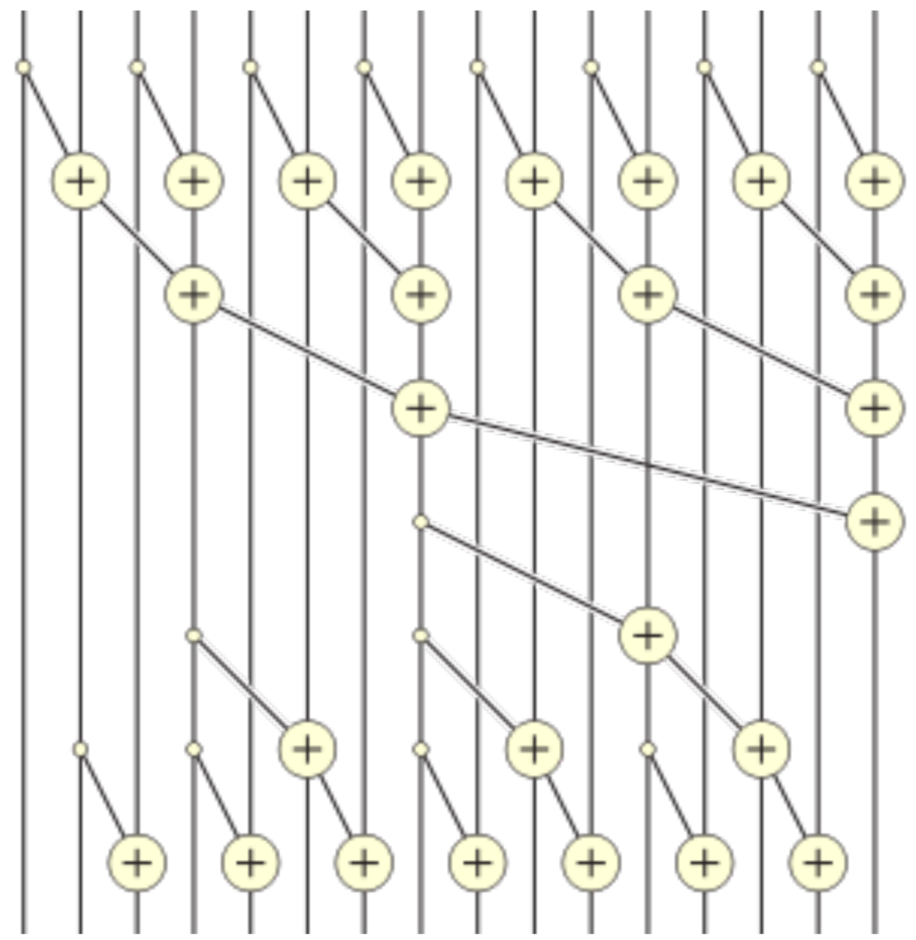
What is the work and span?

Work efficient

Work: $W(n/2) + O(n) = O(n)$

Span: $S(n) = S(n/2) + O(1) = O(\lg n)$

Gave up a constant factor in span compared to Hillis-Steele algorithm



Applications of Data-Parallelism
(using scans)

A Partial List of Applications for Parallel Prefix

- Adding two n-bit integers in $O(\log n)$ time
- Evaluating polynomials
- Solving recurrences
- Radix sort
- “2D parallel prefix” for image segmentation
- Traversing linked lists
- and many others!

Application: Stream Compression (aka Filter)

- Definition: Given a sequence $A=[x_0, x_1, \ldots, x_{n-1}]$ and a Boolean array of flags $B[b_0, b_1, \ldots, b_{n-1}]$, output an array A' containing just the elements A[i] where B[i] = true (maintaining relative order)
- Example:

A =

2	4	3	1	3

B =

T	F	T	T	F

A' =

2	3	1

- Can you implement filter using prefix sum?

Filter Implementation

Can use to filter on some condition

A =

2	4	3	1	3

B =

T	F	T	T	F

1	0	1	1	0

Prefix sum

B' =

0	1	1	2	3

Total sum = 3

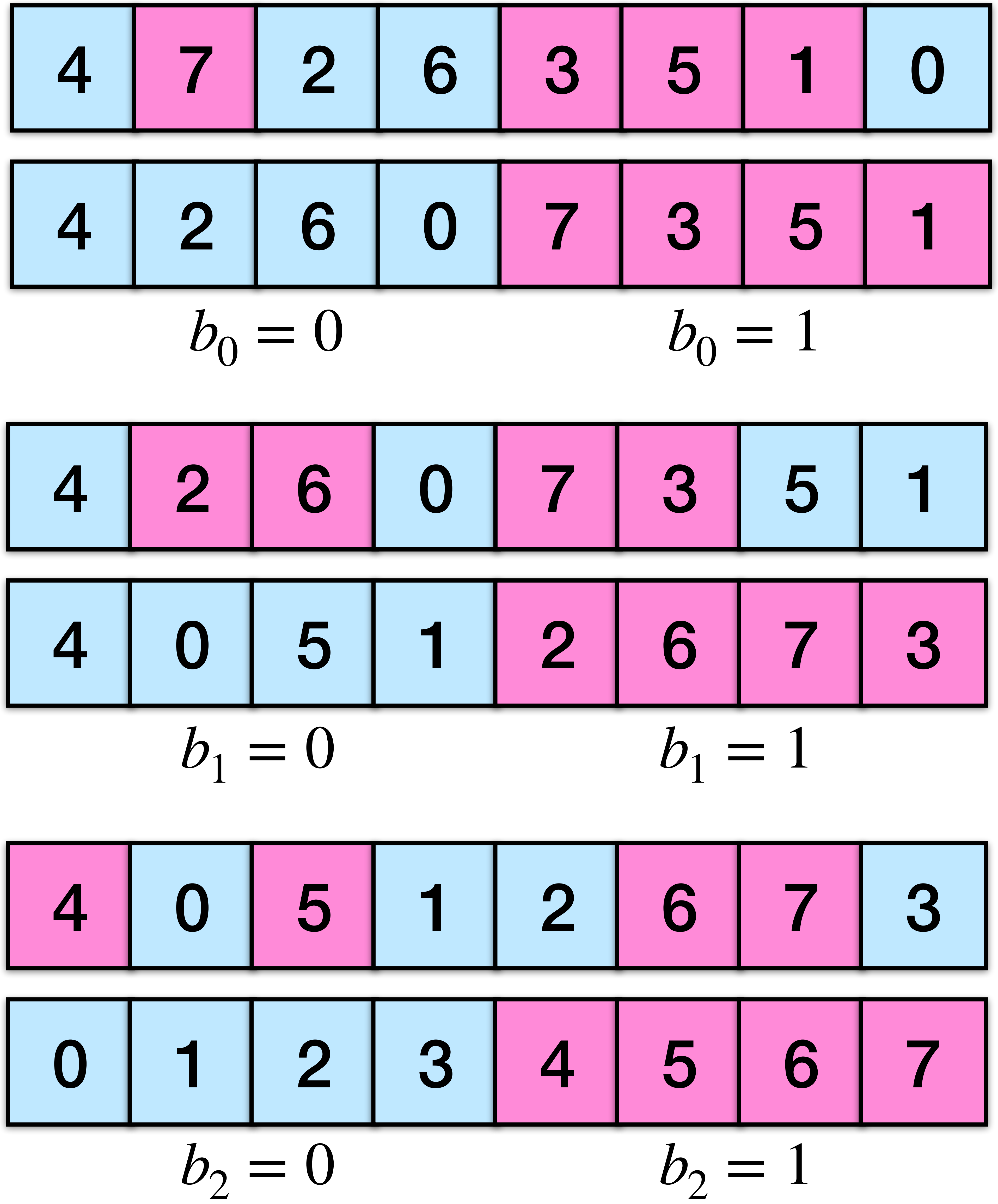
```
//Assume B'[n] = total sum
parallel-for i=0 to n-1:
    if(B'[i] != B'[i+1]):
        A'[B'[i]] = A[i];
```

Allocate array of size 3

A' =

2	3	1

Application: Radix Sort (Serial)



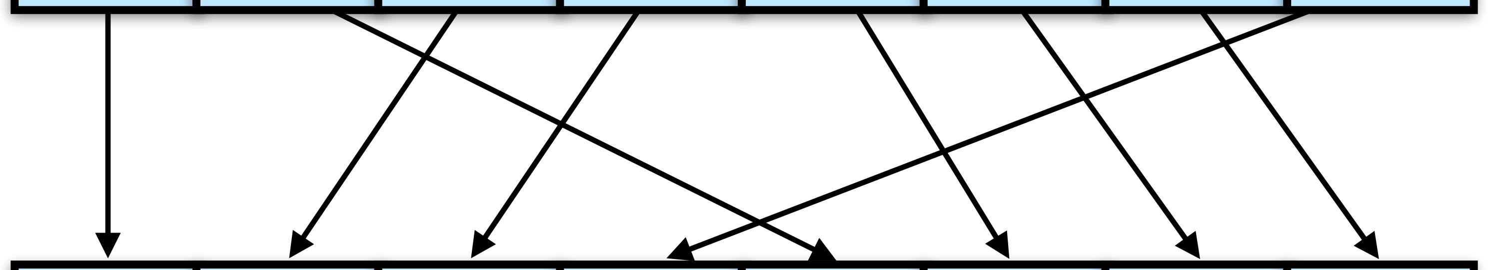
Sort on least significant bit (b_0 in $b_2 b_1 b_0$)
XX0 < XX1 (evens before odds)

Stably sort entire array on next bit
X0X < X1X

Stably sort entire array on next bit
0XX < 1XX

Application: Data-Parallel Radix Sort

4	7	2	6	3	5	1	0		Input
0	1	0	0	1	1	1	0		Odds = last bit of each element
1	0	1	1	0	0	0	1		Evens = complement of odds
0	1	1	2	3	3	3	3	(4)	Even_positions = exclusive scan of evens
4	4	4	4	4	4	4	4		totalEvens = broadcast last element
0	1	2	3	4	5	6	7		index = constant array of 0 .. n
4	4	5	5	5	6	7	9		odd_positions = #evens + indx - even_pos
0	4	1	2	5	6	7	3		pos = get positions using masked assignment
4	7	2	6	3	5	1	0		Scatter input according to pos



4	2	6	0	7	3	5	1	(repeat with next bit until you are out of bits)

Analyzing Data-Parallel Radix Sort

ALGORITHM: $\text{RADIX\_SORT}(A, b)$

for i **from** 0 **to** $b-1$
  $\text{FLAGS} := \{(a >> i)\ \ mod\ \ 2 : a \in A\}$
  $\text{NOTFLAGS} := \{1 - b : b \in \text{FLAGS}\}$
  $R_0 := \text{SCAN}(\text{NOTFLAGS})$
  $s_0 := \text{SUM}(\text{NOTFLAGS})$
  $R_1 := \text{SCAN}(\text{FLAGS})$
  $R := \{\textbf{if}\ \ \text{FLAGS}[j] = 0\ \ \textbf{then}\ \ R_0[j]\ \ \textbf{else}\ \ R_1[j] + s_0 : j \in [0..|A|)\}$
  $A := A \leftarrow \{(R[j], A[j]) : j \in [0..|A|)\}$
return A

What is the work and span?

Analyzing Data-Parallel Radix Sort

ALGORITHM: RADIX_SORT(A, b)

for i **from** 0 **to** $b - 1$
  FLAGS $:= \{(a >> i) \;\; mod \;\; 2 : a \in A\}$
  NOTFLAGS $:= \{1 - b : b \in$ FLAGS$\}$
  $R_0 :=$ SCAN(NOTFLAGS)
  $s_0 :=$ SUM(NOTFLAGS)
  $R_1 :=$ SCAN(FLAGS)
  $R := \{$**if** FLAGS$[j] = 0$ **then** $R_0[j]$ **else** $R_1[j] + s_0 : j \in [0..|A|)\}$
  $A := A \leftarrow \{(R[j], A[j]) : j \in [0..|A|)\}$
return A

$O(\log n)$ span

What is the work and span?

Work: There are b iterations of the for loop, and iteration takes $O(n)$ work, so the total work is $O(bn)$.

Span: There are b iterations of the for loop, and iteration has $O(\log n)$ span, so the total span is $O(b \log n)$.

Application: Adding n-bit Integers

Problem: Computing sum of two n-bit binary numbers, a and b.

$a = a_{n-1}a_{n-2}\ldots a_0$ and $b = b_{n-1}b_{n-2}\ldots b_0$

$s = a + b = s_n s_{n-1}\ldots s_0$ (using carry bit array $c = c_{n-1}, \ldots, c_0, c_{-1}$)

```
c[-1] = 0 // rightmost carry bit
for i = 0 to n - 1: //compute right to left
  s[i] = (a[i] xor b[i]) xor c[i-1] // one or three 1s
  c[i] = ((a[i] xor b[i]) and c[i-1]) or (a[i] and b[i]) // next carry bit
```

Example:
a = 22
b = 29

a	1 0 1 1 0
b	1 1 1 0 1
c	1 1 1 0 0 0 0
s	1 1 0 0 1 1

s[0] depends on these

Goal: Compute all c_i in $O(\log n)$ span via parallel prefix

Application: Adding n-bit Integers

```
c[-1] = 0 // rightmost carry bit
for i = 0 to n - 1: //compute right to left
  c[i] = ((a[i] xor b[i]) and c[i-1]) or (a[i] and b[i]) // next carry bit
```

Idea: Split carry bit into two cases that indicate information about the carry-out (c_i) regardless of carry-in (c_{i-1}):

- Generate (g_i): This column will generate a carry-out **whether or not** the carry-in is 1.

  $$g_i = a_i \,\&\&\, b_i$$

- Propagate (p_i): This column will propagate a carry-in **if there is one** to the carry-out.

  $$p_i = a_i \,||\, b_i$$

can be computed in parallel

$$\boxed{c_i = g_i + p_i c_{i-1}}$$

Carry Lookahead Logic

Idea: Define each carry-in in terms of p_i, g_i and the initial carry in c_{i-1} and not in terms of carry chain (i.e., unwind the recursion):

- $c_0 = g_0 + p_0 c_{-1}$
- $c_1 = g_1 + p_1 c_0 = g_1 + p_1 g_0 + p_1 p_0 c_{-1}$
- …

Can be expressed with 2-by-2 boolean matrix multiplication:

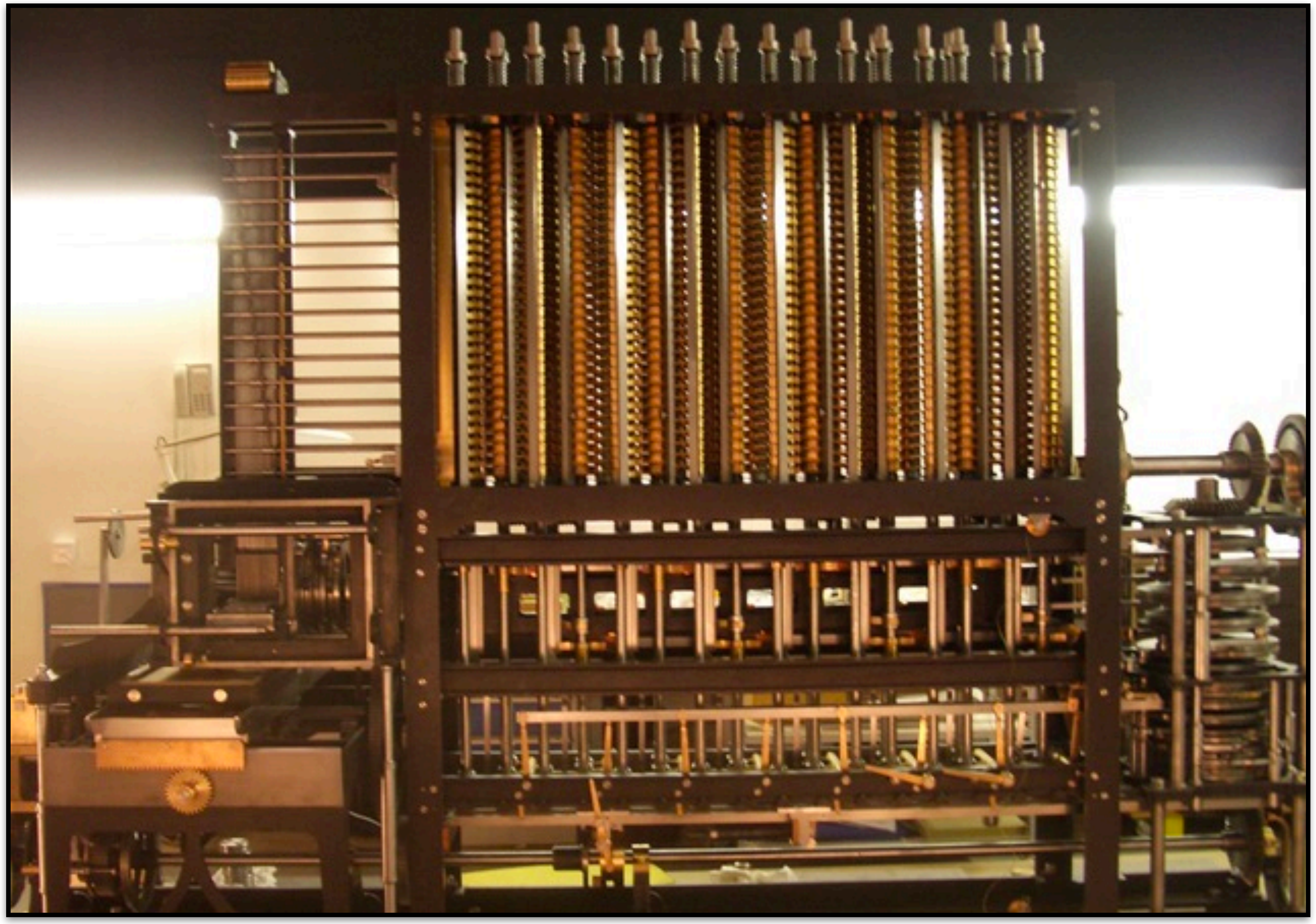
$$M[i] = \begin{pmatrix} p[i] & g[i] \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} c[i] \\ 1 \end{pmatrix} = M[i] \times M[i-1] \times \ldots \times M[0] \times \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Carry-lookahead addition is used in all computers

This idea is used in all hardware

The design goes back to Babbage in the 1800s:



https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/carry2

Application: Lexical Analysis

Lexical analysis **divides a long string of characters into tokens** - often the first thing a compiler does when processing a program.

Suppose we have a regular language - we can represent it with a **finite-state automaton** that begins in a certain state and makes transitions between states based on the characters read.

Seems to depend on the previous state, which depends on characters read up to some point

if	x	<=	n	then	print	(	"x = "	,	x	)	;

TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read													
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

Goal: Perform lexical analysis in parallel

“Data-Parallel Algorithms,” Hillis and Steele, CACM 1986.

Application: Lexical Analysis

Idea: replace every character in the string with the array representation of its **state-to-state function** (column).

Then perform a parallel-prefix operation with $\oplus$ as the **array composition**. Each character becomes an array representing the **state-to-state function** for that prefix.

Use the **initial state** to index into the arrays.

Example of state-to-state function for the space character

TABLE I. A Finite-State Automaton for Recognizing Tokens

Old State	Character Read														
•	A	B	...	Y	Z	+	−	*	<	>	=	"	Space	New line
N	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
A	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
Z	Z	Z	...	Z	Z	*	*	*	<	<	*	Q	N	N
*	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
<	A	A	...	A	A	*	*	*	<	<	=	Q	N	N
=	A	A	...	A	A	*	*	*	<	<	*	Q	N	N
Q	S	S	...	S	S	S	S	S	S	S	S	E	S	S
S	S	S	...	S	S	S	S	S	S	S	S	E	S	S
E	E	E	...	E	E	*	*	*	<	<	*	S	N	N

"Data-Parallel Algorithms," Hillis and Steele, CACM 1986.

Application: Segmented Scans

Inputs: value array, flag array, associative operator $\oplus$

A:	1	2	3	4	5	6	7	8
flags:	0	0	1	0	0	1	0	1
Result:	1	3	3	7	12	6	13	8

Can be used to parallelize sparse-matrix vector multiply (SpMV)

Application: Image Processing

- A **summed-area table** is an algorithm/data structure for quickly generating the sum of values in some rectangular subset of a grid.
- Often used in image processing [Crow, 84].
- Computes **prefix sums in both dimensions**, and then **inclusion-exclusion on the corners** to compute the sum within any rectangular area.

$$I(x,y) = \sum_{x' \le x, y' \le y} i(x', y')$$

$$A = (x_0, y_0), B = (x_1, y_0), C = (x_0, y_1), D = (x_1, y_1)$$

$$\sum_{x_0 < x \le x_1, y_0 < y \le y_1} i(x,y) = I(D) + I(A) - I(B) - I(C)$$

1.

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

2.

31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

15 + 16 + 14 + 28 + 27 + 11 =
101 + 450 − 254 − 186 = 111

https://en.wikipedia.org/wiki/Summed-area_table

Application: Image Processing

- A **summed-area table** is an algorithm/data structure for quickly generating the sum of values in some rectangular subset of a grid.
- Often used in image processing [Crow, 84].
- Computes **prefix sums in both dimensions**, and then **inclusion-exclusion on the corners** to compute the sum within any rectangular area.

$$I(x, y) = \sum_{x' \le x, y' \le y} i(x', y')$$

$$A = (x_0, y_0), B = (x_1, y_0), C = (x_0, y_1), D = (x_1, y_1)$$

$$\sum_{x_0 < x \le x_1, y_0 < y \le y_1} i(x, y) = I(D) + I(A) - I(B) - I(C)$$

Requires inverse

1.

31	2	4	33	5	36
12	26	9	10	29	25
13	17	21	22	20	18
24	23	15	16	14	19
30	8	28	27	11	7
1	35	34	3	32	6

2.

31	33	37	70	75	111
43	71	84	127	161	222
56	101	135	200	254	333
80	148	197	278	346	444
110	186	263	371	450	555
111	222	333	444	555	666

15 + 16 + 14 + 28 + 27 + 11 =
101 + 450 − 254 − 186 = 111

https://en.wikipedia.org/wiki/Summed-area_table

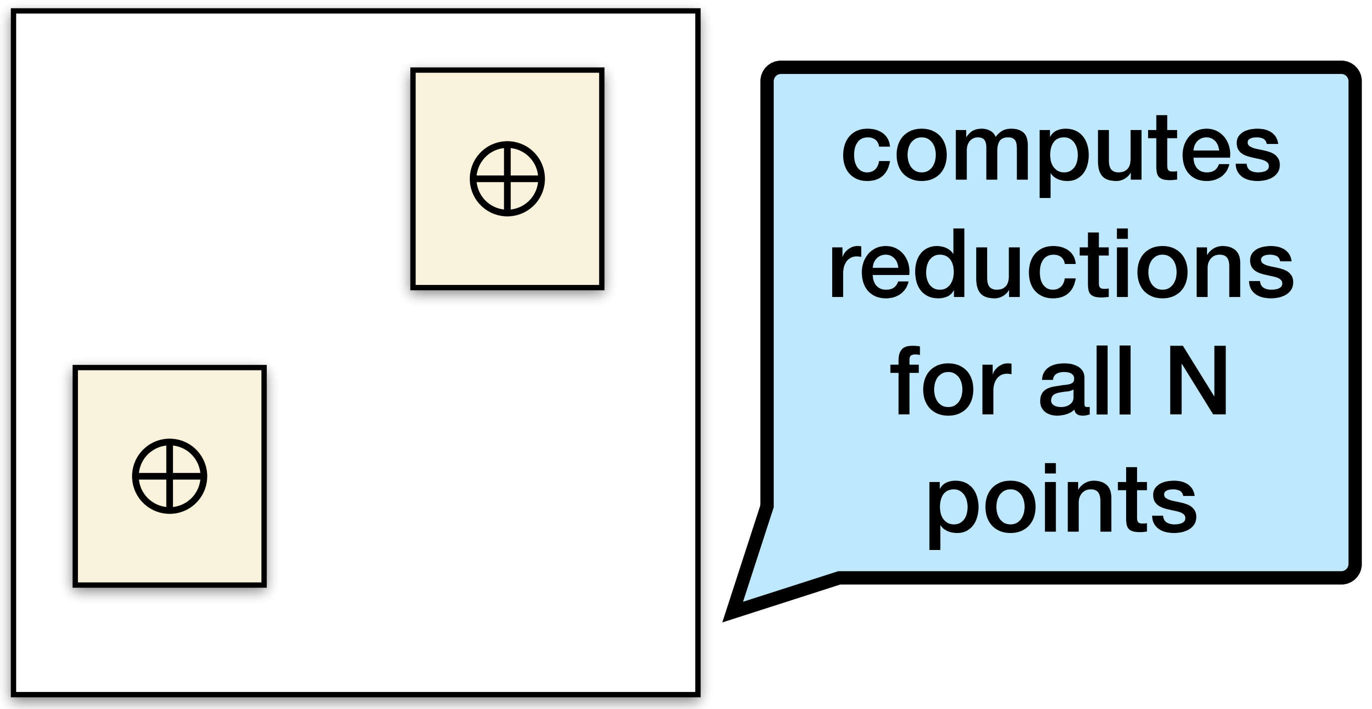
Application: Tensor Region Sums

Several scientific computing applications involve reducing **many (potentially overlapping) regions** of a tensor to a single value for each region, using a binary associative operator $\oplus$.

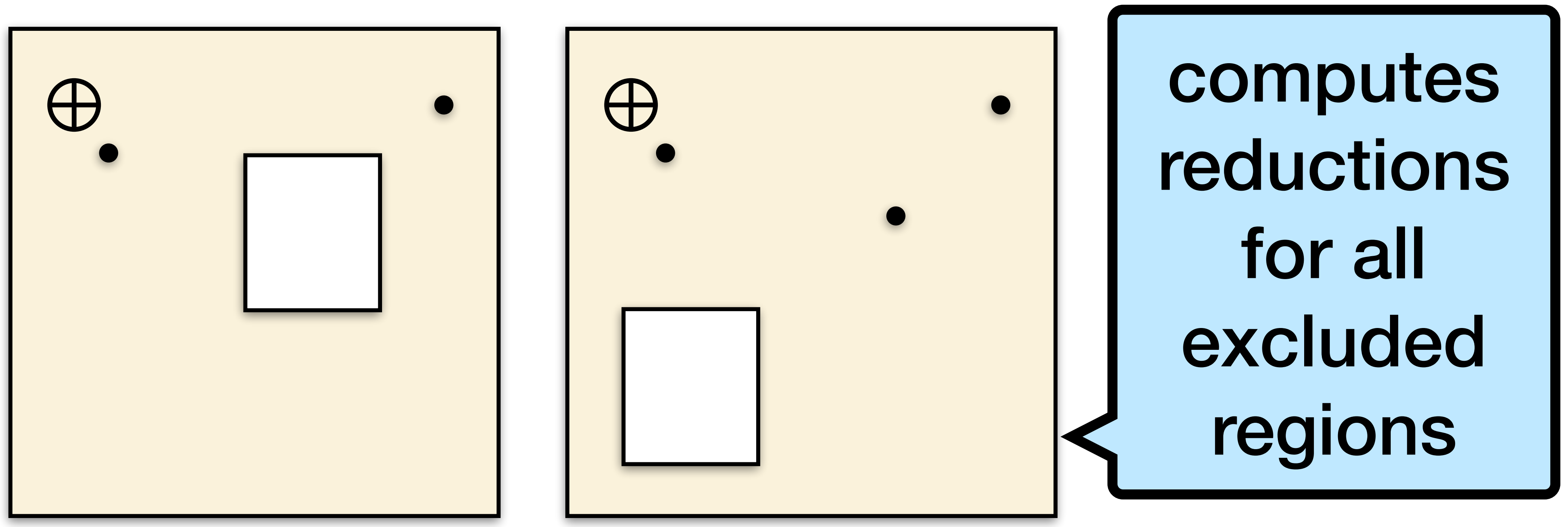
Inclusion: The **summed-area table** (SAT) method preprocesses an image to answer queries about the sum in rectangular subregions of a tensor [C84].

Exclusion: The essence of the **fast multipole method** (FMM) is a reduction of a subregion's elements, excluding elements too close [BG97, CRW93, D00].

Summed-area Table



Fast Multipole Method



“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

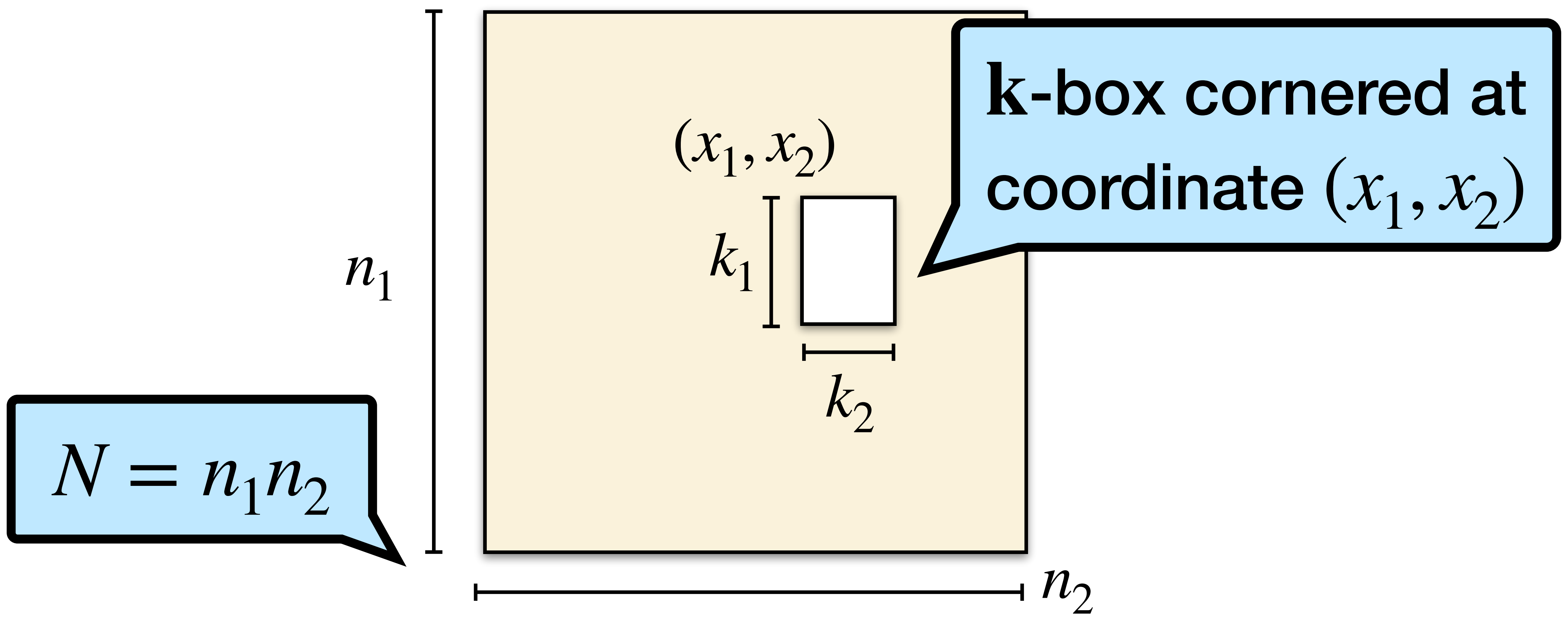
Binary associative operator

Excluded-sums problem

The excluded-sums problem [DDELP05] underlies applications that require **reducing regions of a tensor** to a single value using $\oplus$.

In 2D, it takes as input an $n_1 \times n_2$ matrix A and "box size" $\mathbf{k} = (k_1, k_2)$ where $k_1 \leq n_1, k_2 \leq n_2$.

The problem involves reducing the excluded region outside of **every** $\mathbf{k}$-box in the matrix.



"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

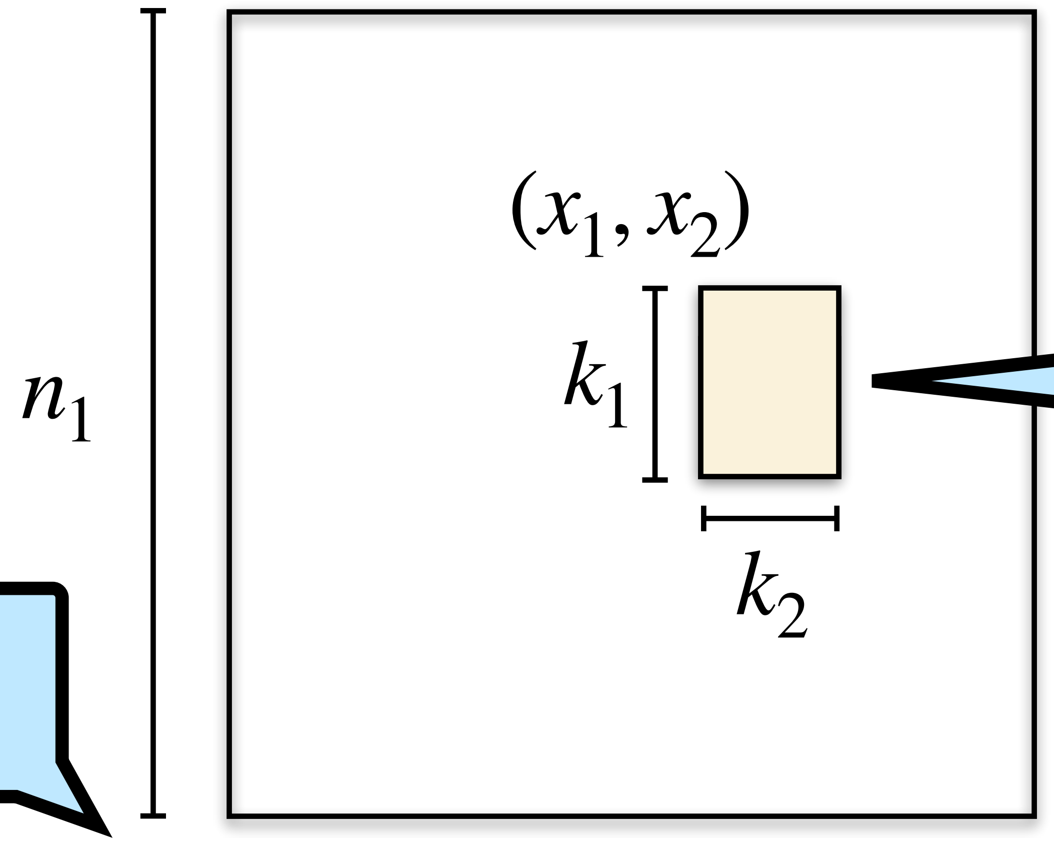
Included-sums Problem

Binary associative operator

The included-sums problem takes the same input as the excluded-sums problem.

In 2D, the included sum at coordinate (x_1, x_2) involves reducing (accumulating with $\oplus$) all elements **in the k-box** cornered at (x_1, x_2).

Can be computed straightforwardly with four nested loops in $\Theta(n_1 n_2 k_1 k_2)$ time.



$N = n_1 n_2$

$$\bigoplus_{y_1=x_1}^{x_1+k_1-1} \bigoplus_{y_2=x_2}^{x_2+k_2-1} A[y_1, y_2]$$

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Inclusion and Exclusion Example

Input

$$\begin{pmatrix} 1 & 3 & 6 & 2 & 5 \\ 3 & 9 & 1 & 1 & 2 \\ 5 & 1 & 5 & 3 & 2 \\ 4 & 3 & 2 & 0 & 9 \\ 6 & 2 & 1 & 7 & 8 \end{pmatrix}$$

Inclusion

$$\begin{pmatrix} 16 & 19 & 10 & 10 & 7 \\ 18 & 16 & 10 & 8 & 4 \\ 13 & 11 & \boxed{10} & 14 & 11 \\ 15 & 8 & 10 & 24 & 17 \\ 8 & 3 & 8 & 15 & 8 \end{pmatrix}$$

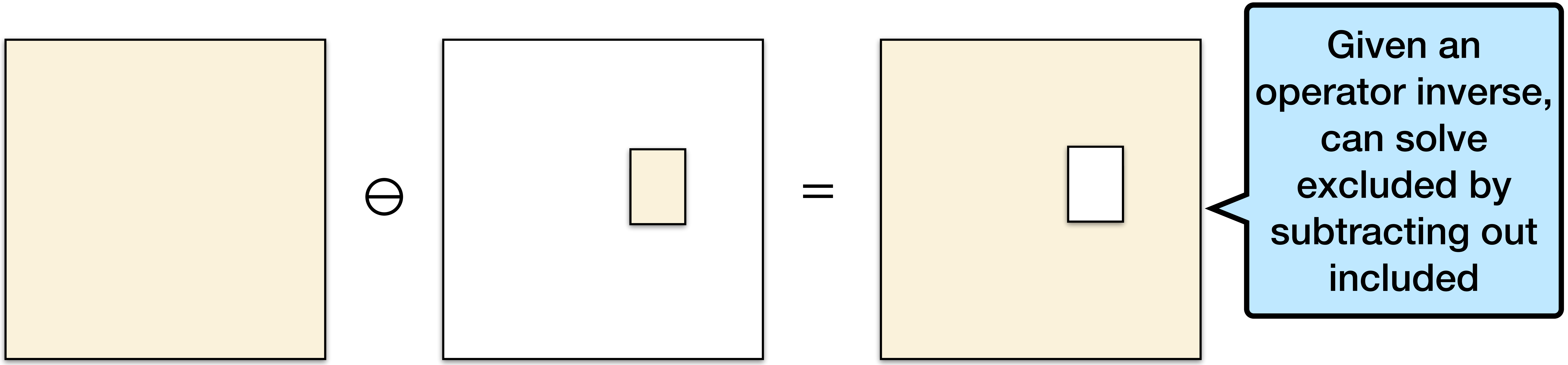
Exclusion

$$\begin{pmatrix} 75 & 72 & 81 & 81 & 84 \\ 73 & 75 & 81 & 83 & 87 \\ 78 & 80 & \boxed{81} & 77 & 80 \\ 76 & 83 & 81 & 67 & 74 \\ 83 & 78 & 83 & 76 & 83 \end{pmatrix}$$

We present an example with addition for ease of understanding, but in general an algorithm for these problems should work with general operators.

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Included and Excluded Sums With and Without Operator Inverse



This approach fails for **operators without inverse** such as max, or the FMM's functions, which may exhibit singularities [DDELP05].

We refine the included- and excluded-sums problems into **weak** and **strong** versions. The weak version requires an operator inverse, while the strong version does not.

"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

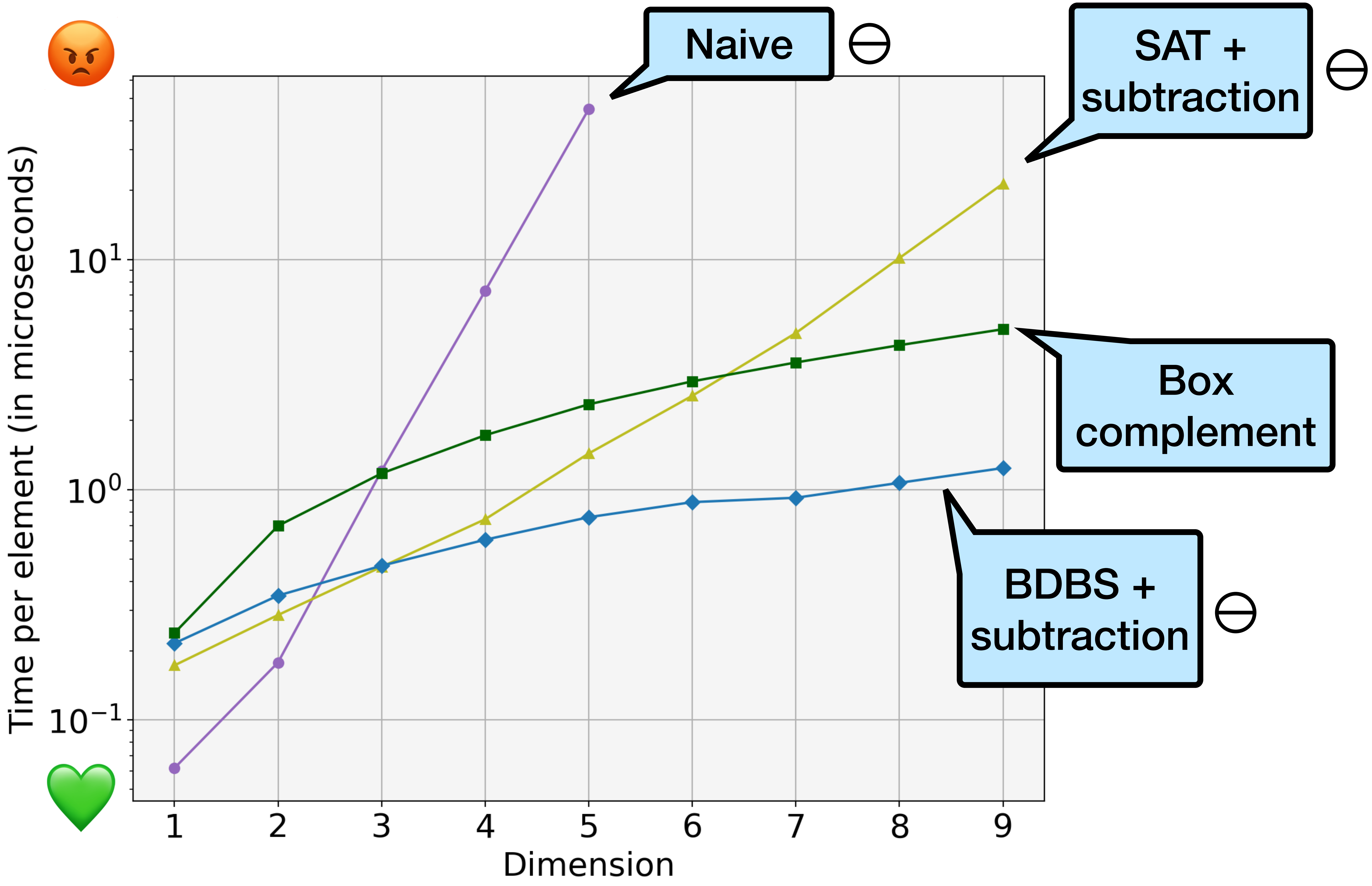
Algorithmic Bounds

Given a d-dimensional tensor with N elements:

Algorithm	Problem	Weak/Strong	Time	Space
Summed-area table	Included	Weak 😐	$\Theta(2^d N)$ 😐	$\Theta(N)$ 💚
Corners(c)	Excluded	Strong 💚	$\Omega(2^d N)$ 😐	$\Theta(cN)$ 😡
Bidirectional box-sum (BDBS)	Included	Strong 💚	$\Theta(dN)$ 💚	$\Theta(N)$ 💚
Box complement	Excluded	Strong 💚	$\Theta(dN)$ 💚	$\Theta(N)$ 💚

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Weak and Strong Excluded Sums in Higher Dimensions



“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Bidirectional Box-sum Algorithm for Strong Included Sums

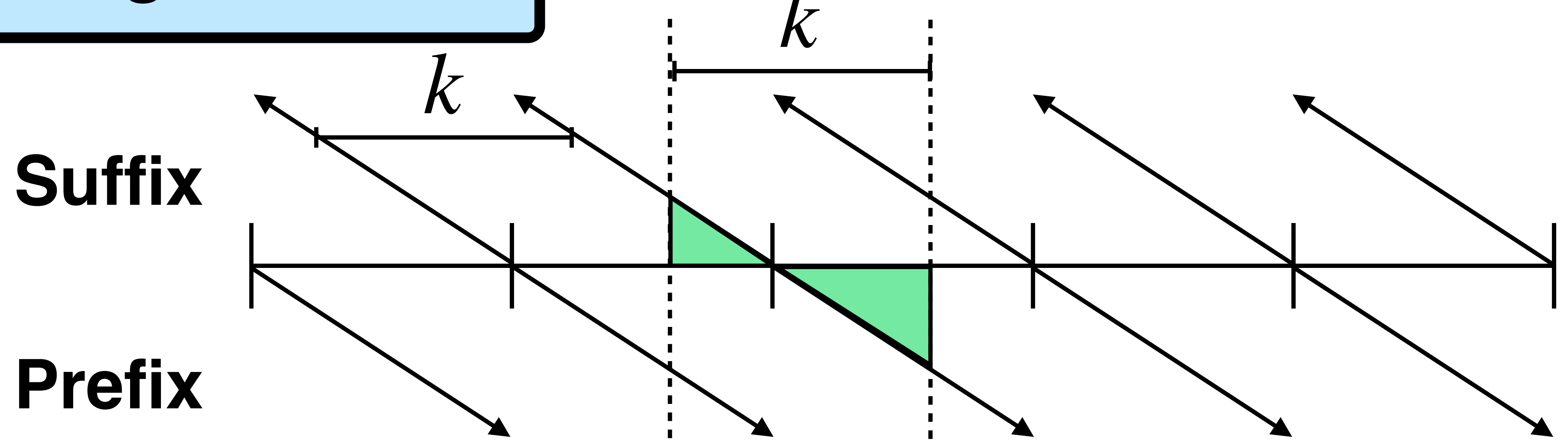
We will start with the bidirectional box-sum algorithm (BDBS) in one dimension then show how to extend the technique to higher dimensions.

Given a list A of length N and a (scalar) box size k, output a list A' of included sums.

$k = 4$

	Position	1	2	3	4	5	6	7	8
Input	A	2	5	3	1	6	3	9	0
Prefix	A_p	2	7	10	11	6	9	18	18
Suffix	A_s	11	9	4	1	18	12	9	0
Target	A'	11	15	13	19	18	12	9	0

Compute intermediate prefix and suffix arrays with N/k prefixes and suffixes of length k each.



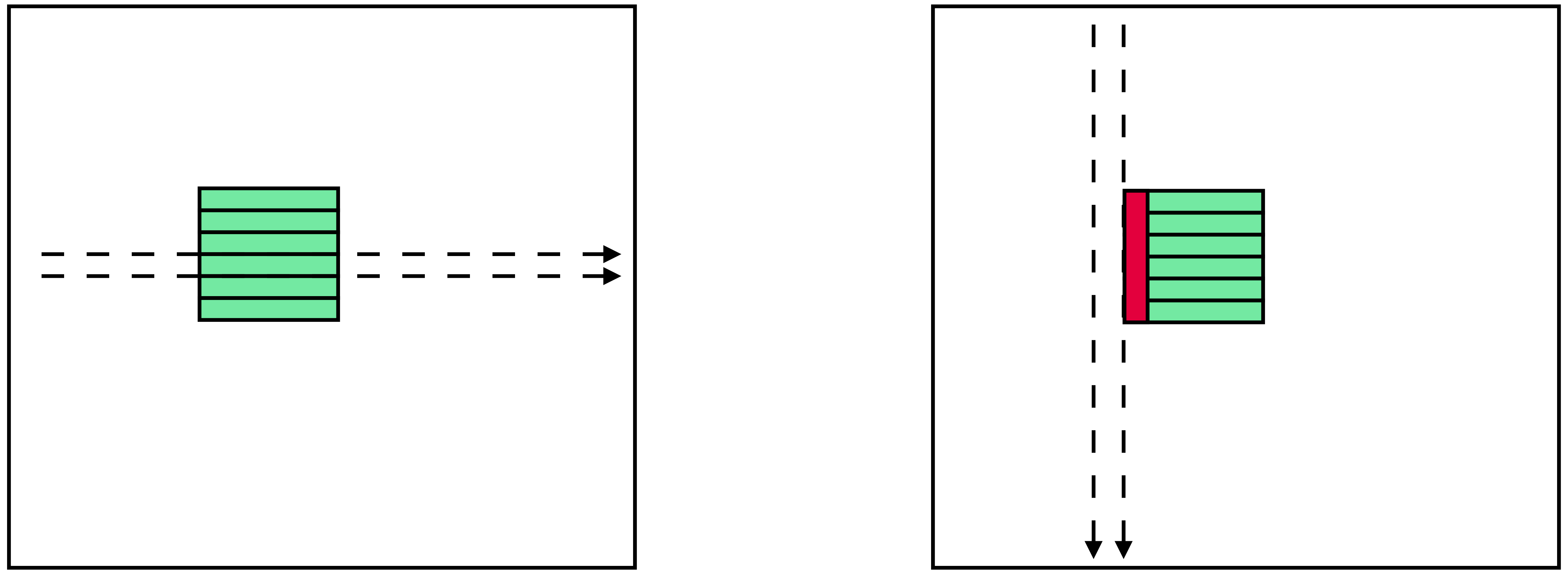
$\Theta(N)$ time, space in 1D

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Multidimensional Bidirectional Box Sum

The BDBS technique extends into arbitrary dimensions by performing the prefixes and suffixes along each dimension in turn.

Bidirectional box sum along first dimension | **Bidirectional box sum along second dimension on result**



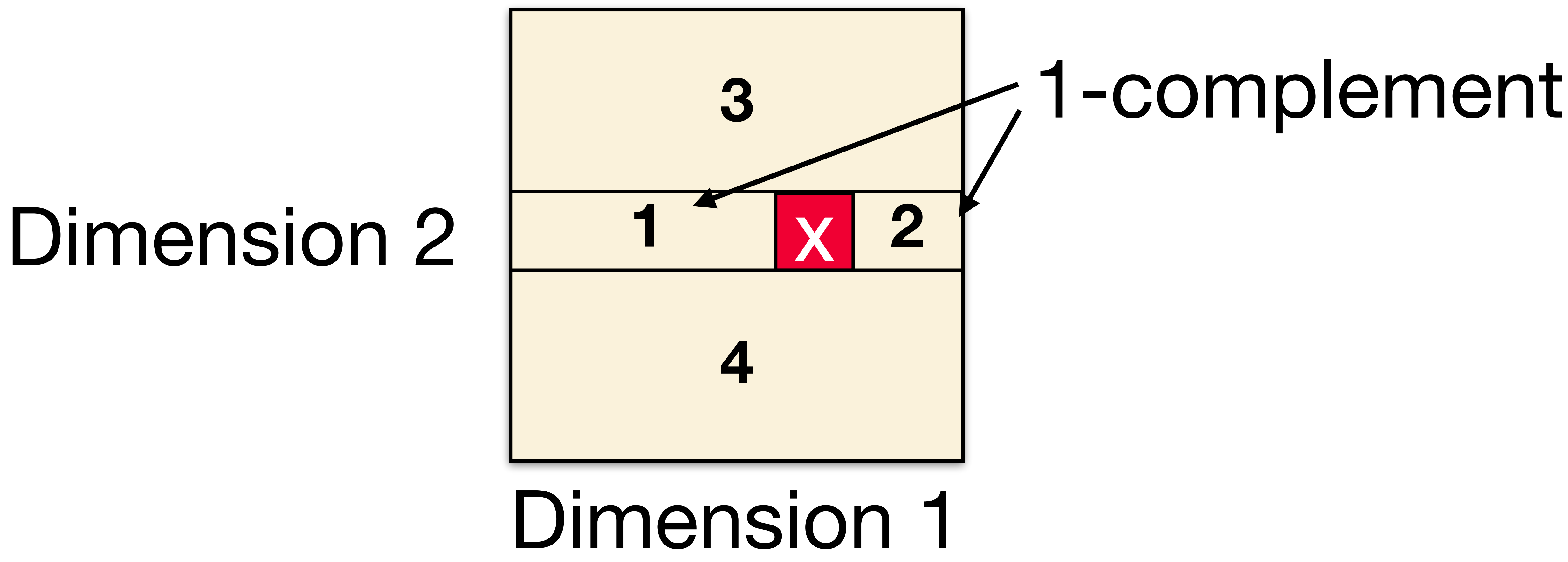
Given a d-dimensional tensor with N elements, BDBS solves the strong included-sums problem in $\Theta(dN)$ time and $\Theta(N)$ space.

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Formulating the Excluded Sum as the Box Complement

Given a d-dimensional tensor and a "box size", we will first sketch how to decompose the excluded region for each point into $2d$ disjoint regions.

At a high level, the "i-complement" of a box such that there is some coordinate in dimension $j \in [1,i]$ that is "out of range" in dimension j, and the coordinates are "in range" for all dimensions $m \in [i+1,d]$.

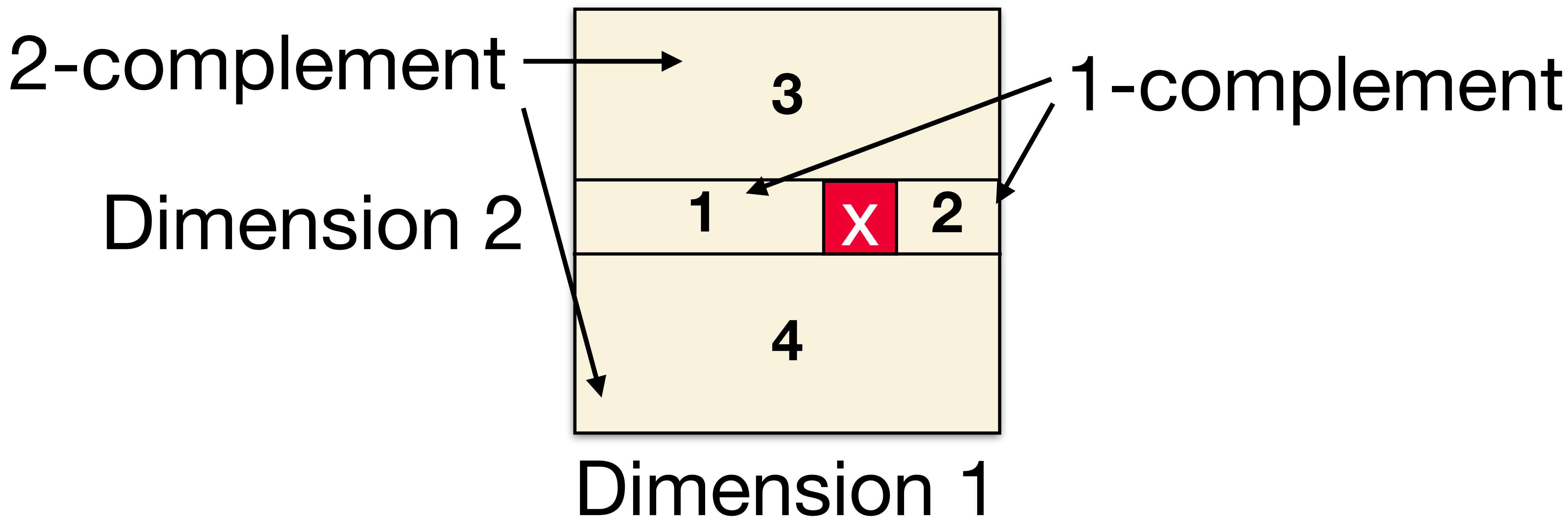


"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

Formulating the Excluded Sum as the Box Complement

Given a d-dimensional tensor and a “box size”, we will first sketch how to decompose the excluded region for each point into $2d$ disjoint regions.

At a high level, the “i-complement” of a box such that there is some coordinate in dimension $j \in [1,i]$ that is “out of range” in dimension j, and the coordinates are “in range” for all dimensions $m \in [i+1,d]$.

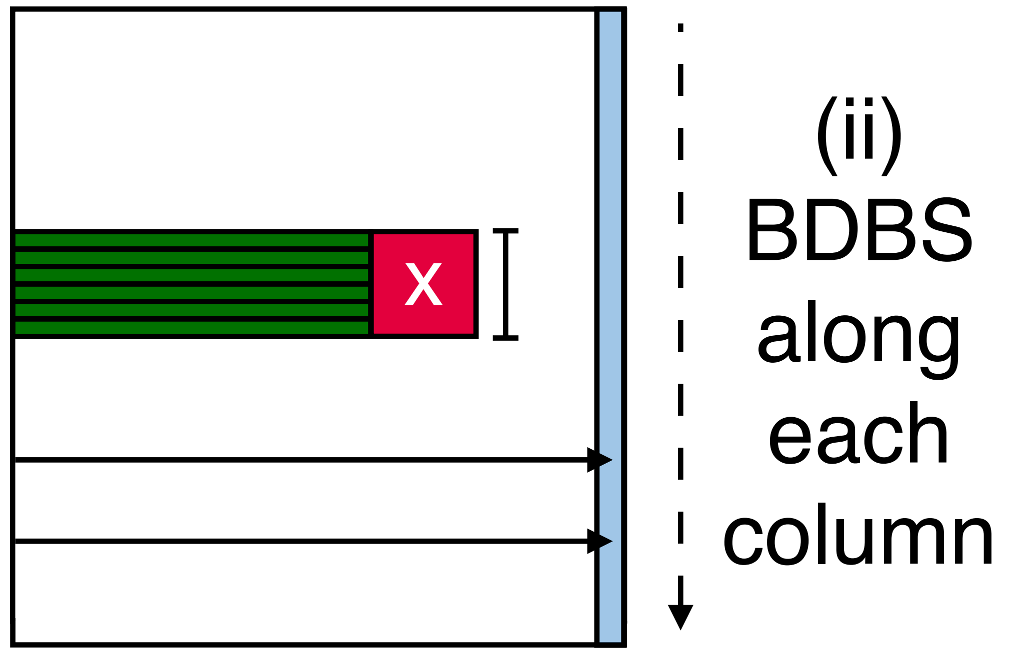


“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

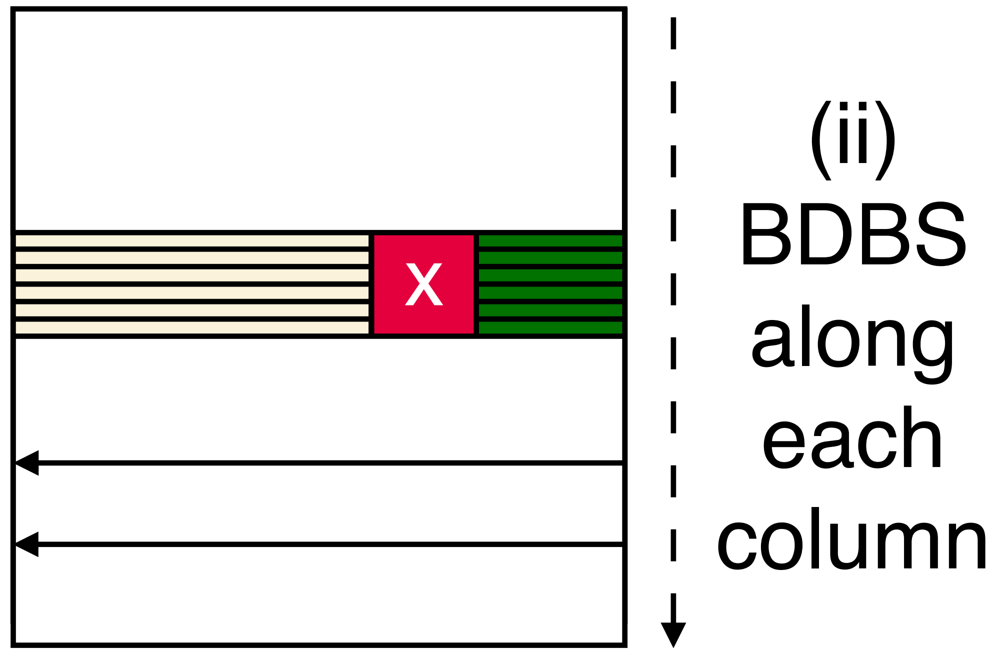
Box-Complement Algorithm for Strong Excluded Sums

The box-complement algorithm uses **dimension reduction** to compute the "i-complement" for all $i = 1, \ldots, d$.

The **BDBS algorithm** for included sums is a major subroutine in the box-complement algorithm to sum up elements "in the range."



(i) Prefix along each row



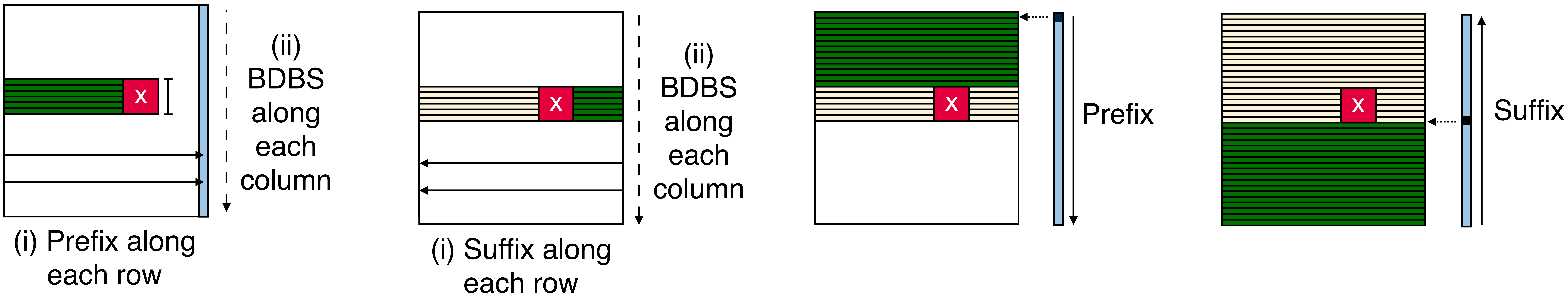
(i) Suffix along each row

"Multidimensional Included and Excluded Sums," Xu, Fraser, and Leiserson. ACDA 2021.

Box-Complement Algorithm for Strong Excluded Sums

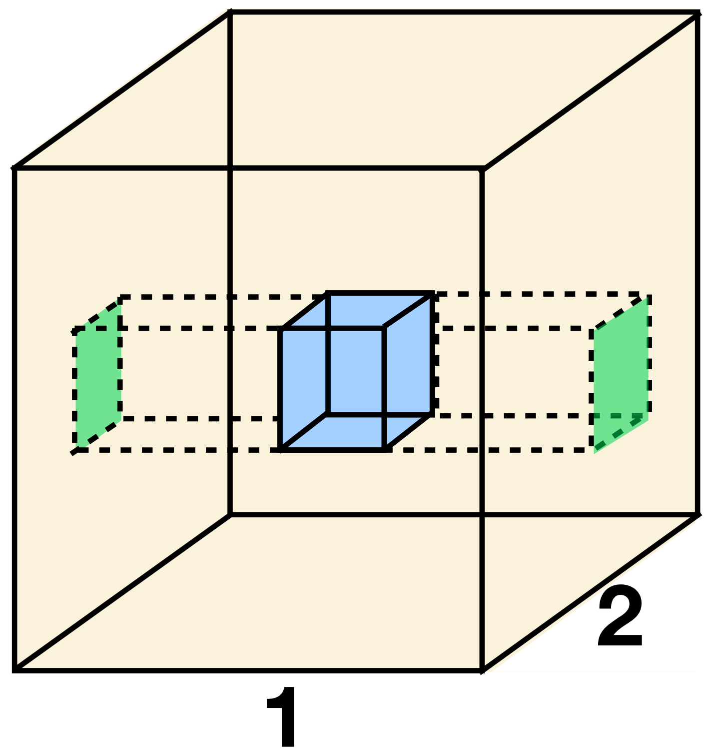
The box-complement algorithm uses **dimension reduction** to compute the "i-complement" for all $i = 1, \ldots, d$.

The **BDBS algorithm** for included sums is a major subroutine in the box-complement algorithm to sum up elements "in the range."



61

Extending the Box-complement Algorithm to Higher Dimensions

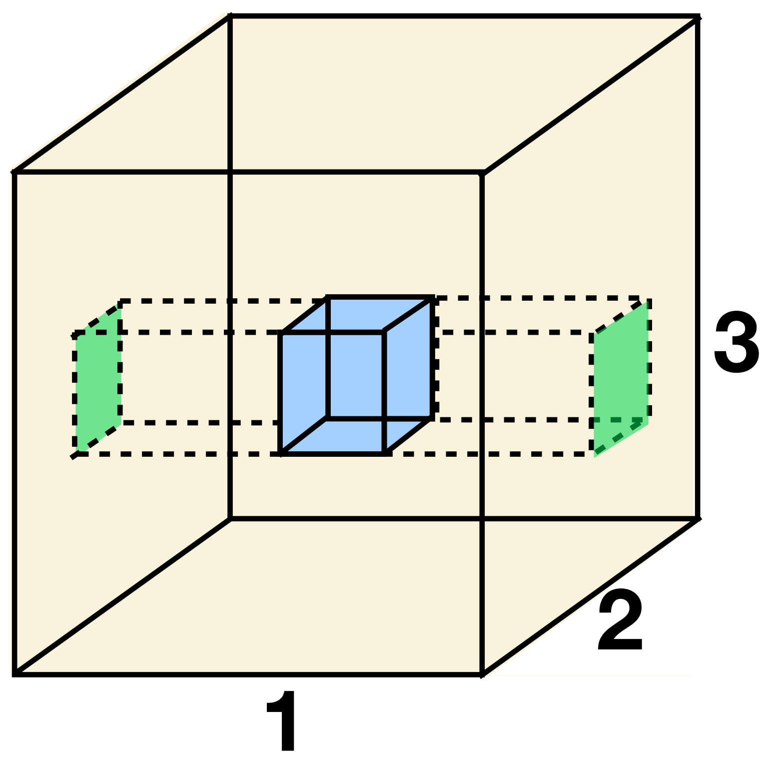


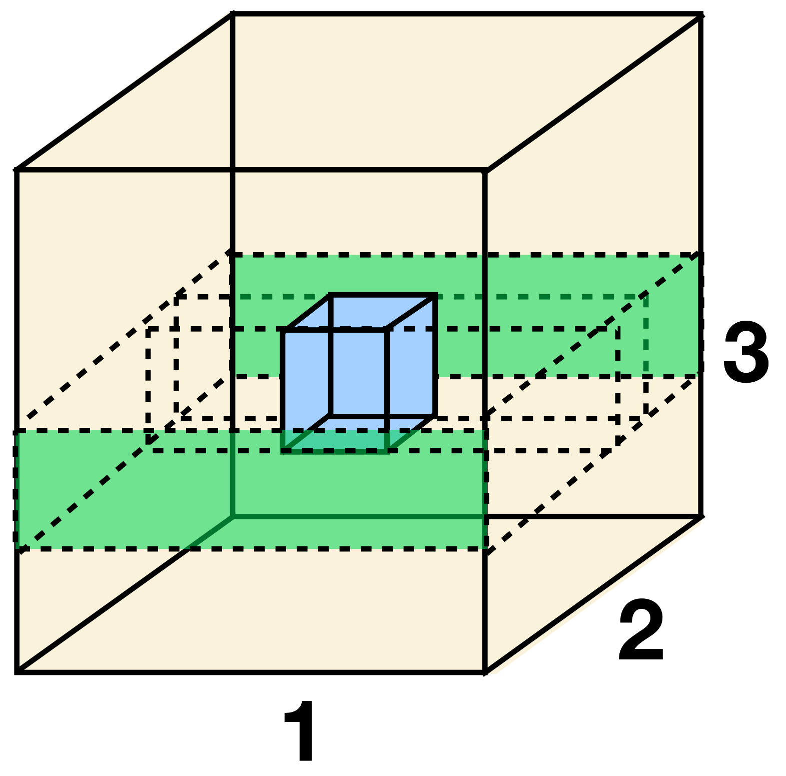
Full Prefix / Suffix Dimensions: 1

Included Sum Dimensions: 2, 3

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Extending the Box-complement Algorithm to Higher Dimensions

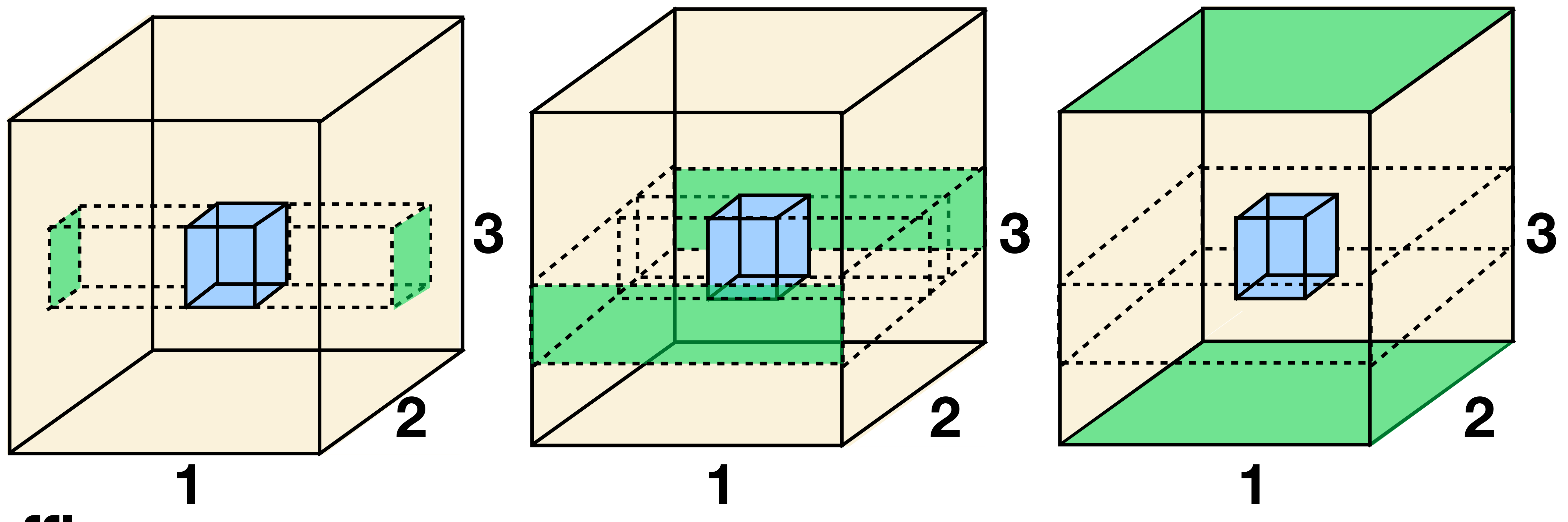




Full Prefix / Suffix Dimensions:	**1**	**1, 2**
Included Sum Dimensions:	**2, 3**	**3**

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Extending the Box-complement Algorithm to Higher Dimensions



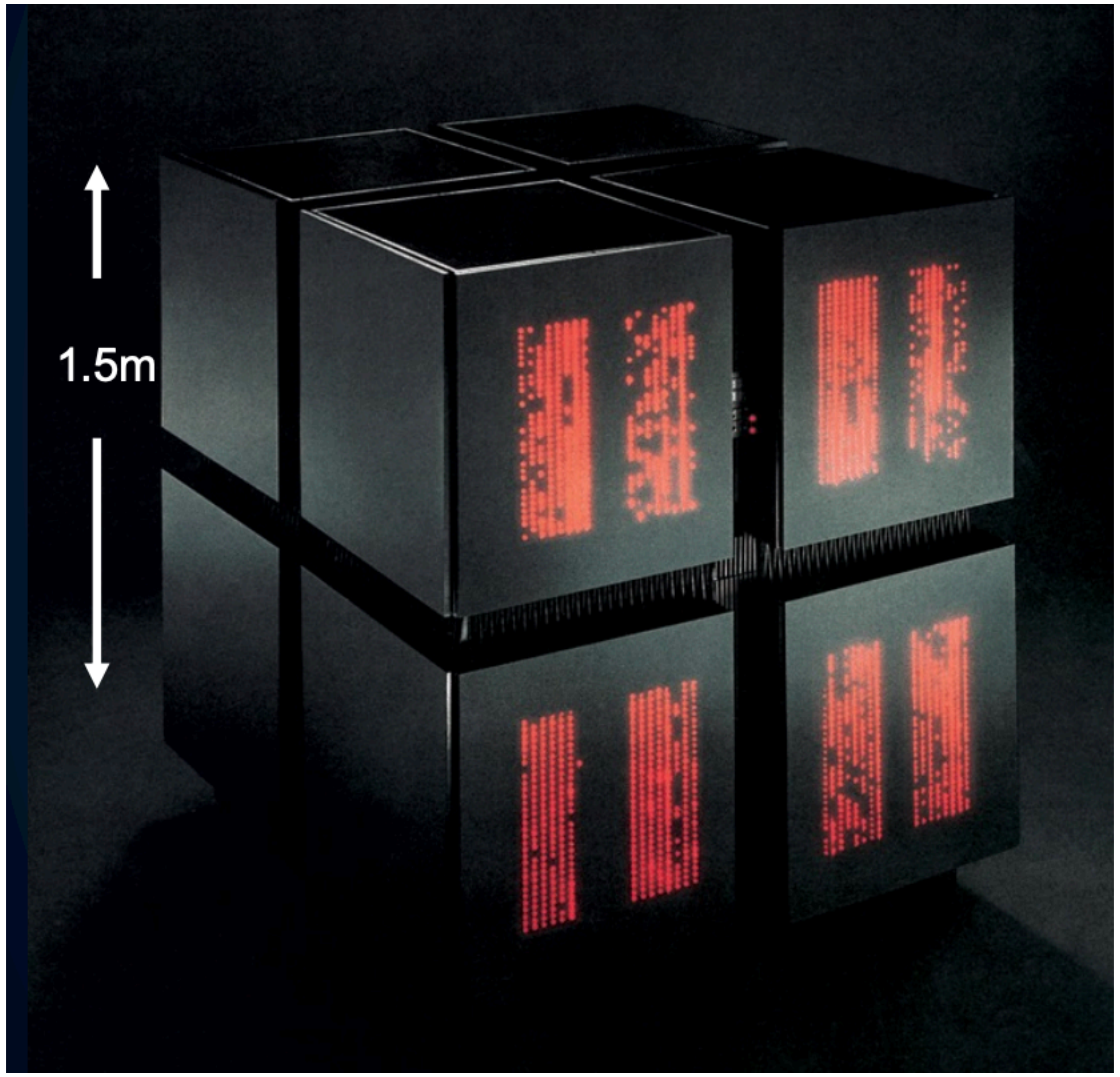
Full Prefix / Suffix Dimensions:	**1**	**1, 2**	**1, 2, 3**
Included Sum Dimensions:	**2, 3**	**3**	**None**

Each dimension-reduction step takes $\Theta(N)$ time and reuses the same temporaries, for a total of $\Theta(dN)$ time and $\Theta(N)$ space.

“Multidimensional Included and Excluded Sums,” Xu, Fraser, and Leiserson. ACDA 2021.

Mapping Data Parallelism to Real Hardware

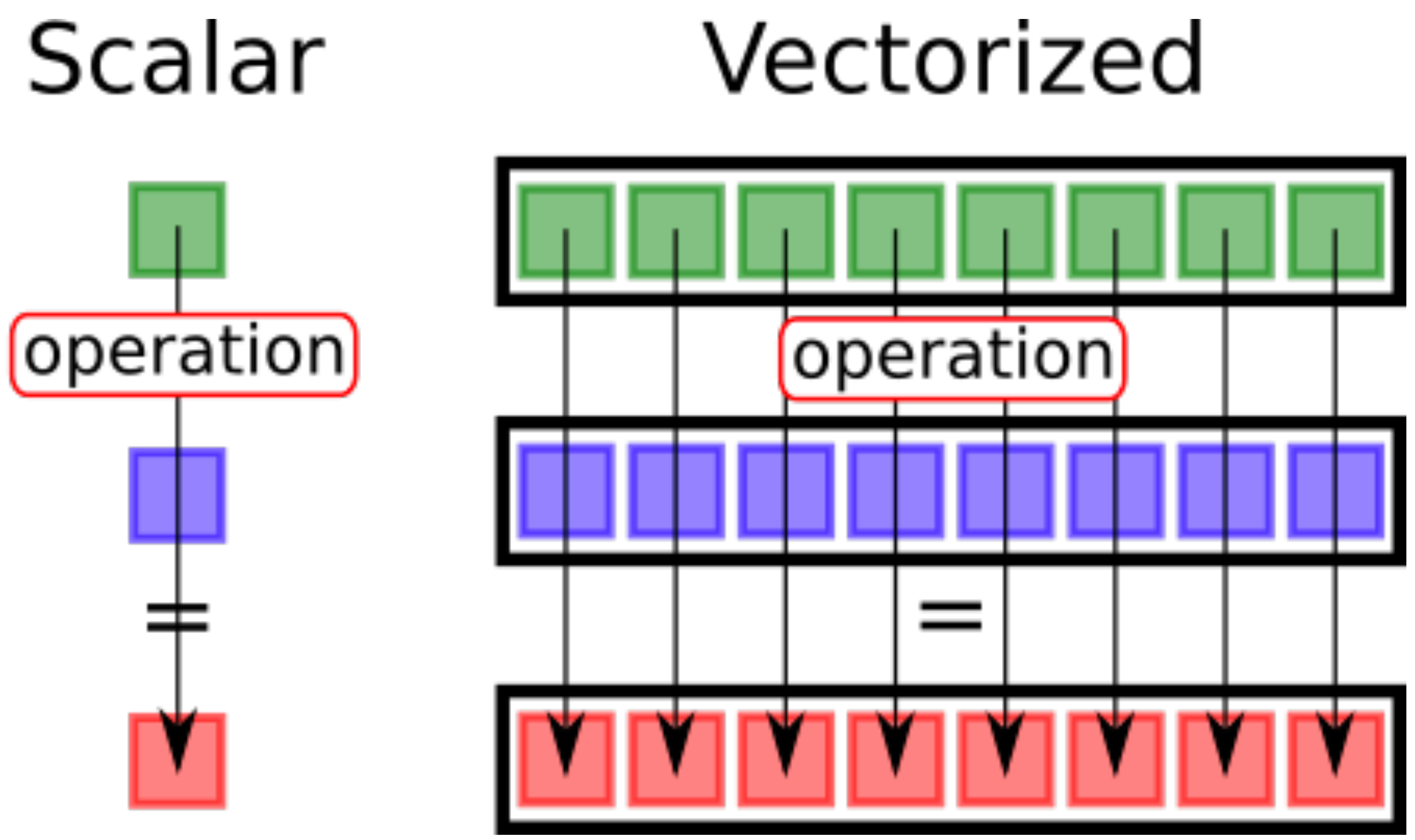
Connection Machine (CM-1,2)



- Designed for AI by Thinking Machines Corporation (Hillis and Handler)
- CM-1 and CM-2 SIMD Design
  - – 65,536 1-bit processors with 4 KB of memory each
  - – 12-D boolean n-cube network (Feynman)
  - – CM-2 add 1 floating point processor per 32 1-bit
- Programmed with data parallel languages (Lisp, C)
- CM-5 was RISC+Vectors

SIMD/Vector Processors Use Data Parallelism

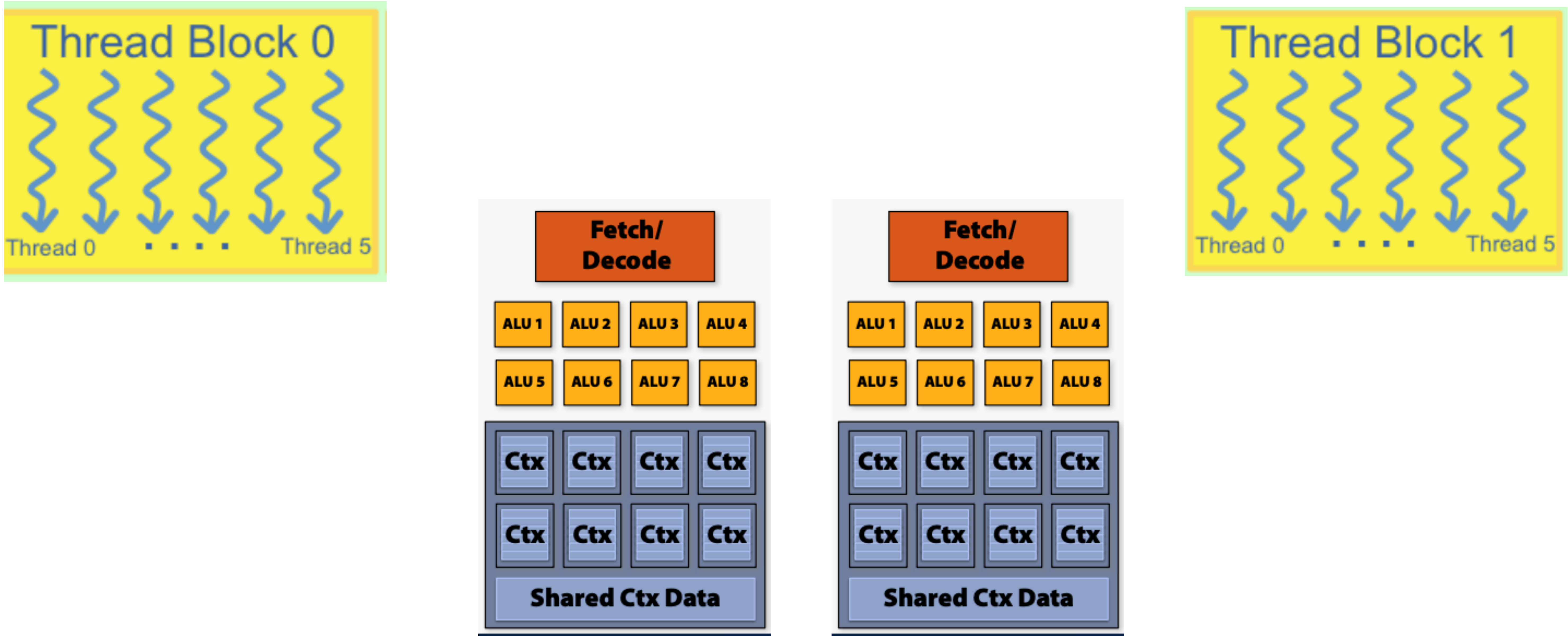
SIMD instructions are specified as operations on **vector registers**.



Data-Level Parallelism
(e.g. SIMD [Flynn72]**)**

Mapping to GPUs

- For n-way parallelism, a GPU may use n threads divided into blocks
- Mapping threads to ALUs and blocks to streaming multiprocessors (SMs) is a compiler / hardware problem.



Summary

- Data-parallel algorithms - applying the same operation to multiple data simultaneously (single-instruction multiple-data).
- Prefix sums and their applications - sometimes can find surprisingly parallel solutions to problems that look serial.
- SIMD implemented via vectors in CPUs, main programming model for GPUs.

Backup past here

Application: Fibonacci via Matrix Multiply Prefix

$$F_{n+1} = F_n + F_{n-1}$$

$$\begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}$$

Can compute all F_n by `matmul_prefix` on

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \cdots$$

$$\downarrow$$

$$\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 3 & 2 \\ 2 & 1 \end{pmatrix}, \begin{pmatrix} 5 & 3 \\ 3 & 2 \end{pmatrix}, \begin{pmatrix} 8 & 5 \\ 5 & 3 \end{pmatrix}, \begin{pmatrix} 13 & 8 \\ 8 & 5 \end{pmatrix} \cdots$$

Select the upper left entry

The same idea works for any linear recurrence.

Proof in "Prefix Sums and Their Applications," Blelloch, 1990.

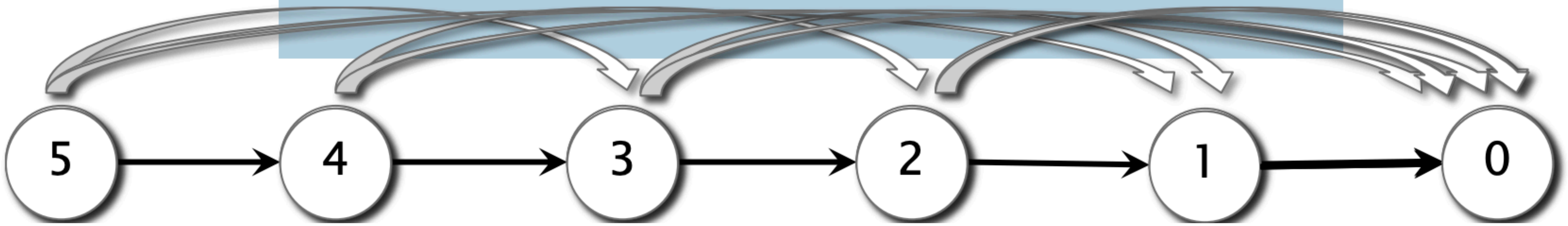
Application: List Ranking

- Have every node in linked list determine its distance to the end

```
parallel-for i=0 to n-1:
    if P[i] == i then rank[i] = 0
    else rank[i] = 1

for j=0 to ceil(log n)-1:
    temp, temp2;
    parallel-for i=0 to n-1:
        temp[i] = rank[P[i]];
        temp2[i] = P[P[i]];
    parallel-for i=0 to n-1:
        rank[i] = rank[i] + temp[i];
        P[i] = temp2[i];
```

What is the work and span?



Work-Span Analysis

```
parallel-for i=0 to n-1:
    if P[i] == i then rank[i] = 0
    else rank[i] = 1

for j=0 to ceil(log n)-1:
    temp, temp2;
    parallel-for i=0 to n-1:
        temp[i] = rank[P[i]];
        temp2[i] = P[P[i]];
    parallel-for i=0 to n-1:
        rank[i] = rank[i] + temp[i];
        P[i] = temp2[i];
```

Not work-efficient: sequential algorithm only requires O(n) work

Work = $O(n \log n)$

Span = $O(\log n)$