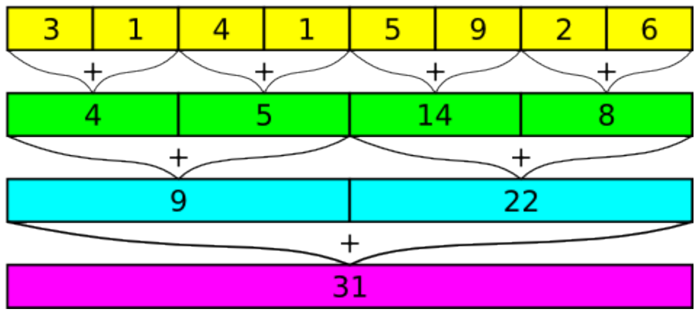
Massively Parallel Algorithms

Parallel Prefix Sum

And Its Applications



G. Zachmann

University of Bremen, Germany

cgvr.cs.uni-bremen.de

- Remember the *reduction* operation
  - Extremely important/frequent operation → Google's *MapReduce*
- Definition prefix sum:

  Given an input sequence

  $$A = (a_0, a_1, a_2, \ldots, a_{n-1}) ,$$

  the (*inclusive*) *prefix sum* of this sequence is the output sequence

  $$\hat{A} = (a_0, a_1 \oplus a_0, a_2 \oplus a_1 \oplus a_0, \ldots, a_{n-1} \oplus \cdots \oplus a_0)$$

  where $\oplus$ is an arbitrary binary associative operator.

  The *exclusive prefix sum* is

  $$\hat{A}' = (\iota, a_0, a_1 \oplus a_0, \ldots, a_{n-2} \oplus \cdots \oplus a_0)$$

  where ι is the identity/zero element, e.g., 0 for the + operator.
- The prefix sum operation is sometimes also called a scan (operation)

- Example:
  - Input: $A = (3\ 1\ 7\ 0\ 4\ 1\ 6\ 3)$
  - Inclusive prefix sum: $\hat{A} = (3\ 4\ 11\ 11\ 15\ 16\ 22\ 25)$
  - Exclusive prefix sum: $\hat{A}' = (0\ 3\ 4\ 11\ 11\ 15\ 16\ 22)$
- Further variant: backward scan
- Applications: many!
  - For example: polynomial evaluation (Horner's scheme)
  - In general: "What came before/after me?"
  - "Where do I start writing my data?"

- The prefix sum problem appears to be "inherently sequential"

- Actually, *prefix-sum* (a.k.a. *scan*) was considered such an important operation, that it was implemented as a primitive in the *CM-2 Connection Machine* (of Thinking Machines Corp.)

Variation: Segmented Scan

- Input: *segments* of numbers in one large vector

3	1	7	0	4	1	6	3	← Payload data
1	0	1	0	0	1	0	0	← Segmentation array (*head-tail flags*)

- Task: scan (prefix-sum) *within* each segment
- Output: prefix-sums for *each* segment, in one vector

0	3	0	7	7	0	1	7

- Forms the basis for a wide variety of algorithms:
  - E.g., Quicksort, Sparse Matrix-Vector Multiply, Convex Hull
- Won't go into details here

Application from "Everyday" Life

- Given:
  - A 100-inch sandwich
  - 10 persons
  - We know how many inches each person wants: [3 5 2 7 28 4 3 0 8 1]
- Task: cut the sandwich quickly
- Sequential method: one cut after another (3 inches first, 5 inches next, ...)
- Parallel method:
  - Compute prefix sum
  - Cut in parallel
  - How quickly can we compute the prefix sum??

Importance of the Scan Operation

- Assume the scan operation is a primitive that has *unit* time costs, then the following algorithms have the following complexities:

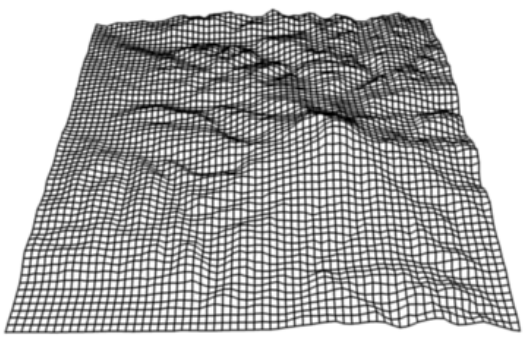
	Model		
Algorithm	EREW	CRCW	Scan
Graph Algorithms (n vertices, m edges, m processors)			
Minimum Spanning Tree	$\lg^2 n$	$\lg n$	$\lg n$
Connected Components	$\lg^2 n$	$\lg n$	$\lg n$
Maximum Flow	$n^2 \lg n$	$n^2 \lg n$	n^2
Maximal Independent Set	$\lg^2 n$	$\lg^2 n$	$\lg n$
Biconnected Components	$\lg^2 n$	$\lg n$	$\lg n$
Sorting and Merging (n keys, n processors)			
Sorting	$\lg n$	$\lg n$	$\lg n$
Merging	$\lg n$	$\lg \lg n$	$\lg \lg n$
Computational Geometry (n points, n processors)			
Convex Hull	$\lg^2 n$	$\lg n$	$\lg n$
Building a K-D Tree	$\lg^2 n$	$\lg^2 n$	$\lg n$
Closest Pair in the Plane	$\lg^2 n$	$\lg n \lg \lg n$	$\lg n$
Line of Sight	$\lg n$	$\lg n$	1
Matrix Manipulation ($n \times n$ matrix, n^2 processors)			
Matrix $\times$ Matrix	n	n	n
Vector $\times$ Matrix	$\lg n$	$\lg n$	1
Matrix Inversion	$n \lg n$	$n \lg n$	n

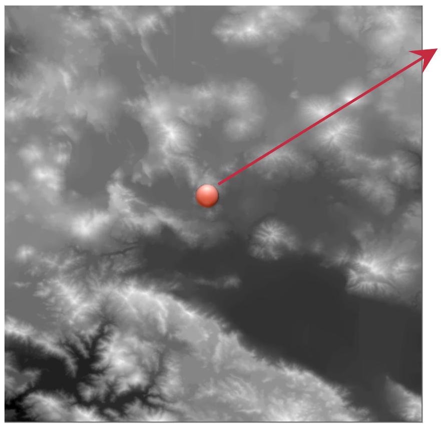
EREW = exclusive-read, exclusive-write PRAM
CRCW = concurrent-read, concurrent-write PRAM
Scan = EREW with scan as unit-cost primitive

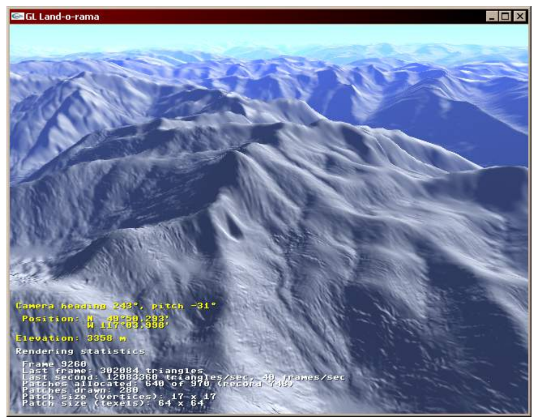
Guy E. Blelloch: *Vector Models for Data-Parallel Computing*

Example: Line-of-Sight

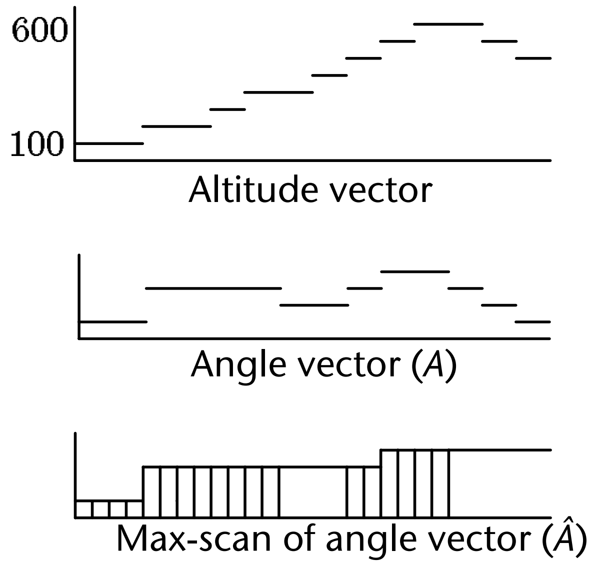
- Given:
  - Terrain as grid of height values (*height map*)
  - Point X in the grid (our "viewpoint", has a height, too)
  - Horizontal viewing direction (we can look up and down, but not to the left or right)
- Problem: find all *visible* points in the grid along the view direction
- Assumption: we have already extracted a vector of heights from the grid containing all cells' heights that are in our horizontal viewing direction





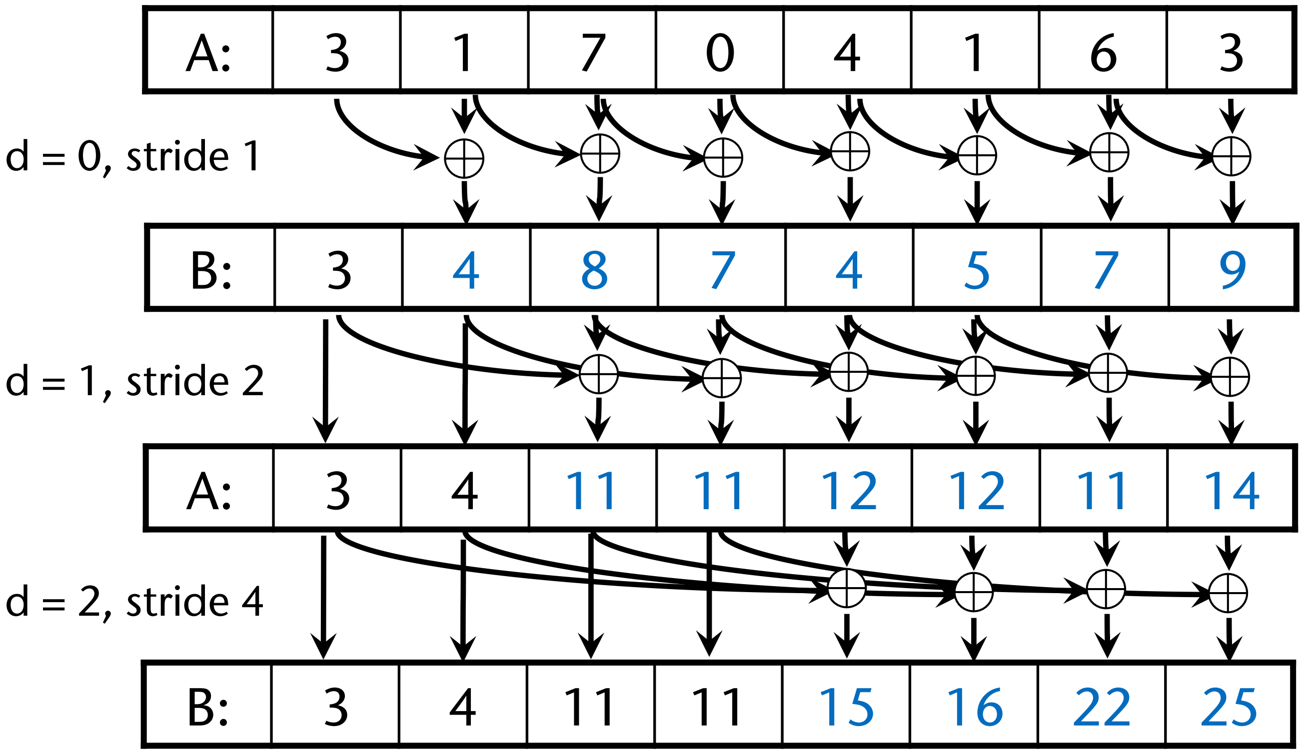


- The algorithm:
  1. Convert height vector to vertical angles (as seen from X) → A
     - One thread per vector element
  2. Perform *max-scan* on angle vector (i.e., prefix sum with the max operator) → $\hat{A}$
  3. Test $\hat{a}_i < a_i$, if true then grid point is visible form X



The Hillis-Steele Algorithm

- Iterate log(n) times:



- Notes:
  - Blue = active threads
  - Each thread reads from "another" thread, too → must use double buffering and barrier synchronization

- The algorithm as pseudo-code:

```
forall i in parallel do        // n threads
   for d = 0...log(n)-1:
      if i >= 2^d :
         x[i] = x[ i - 2^d ] + x[i]
```

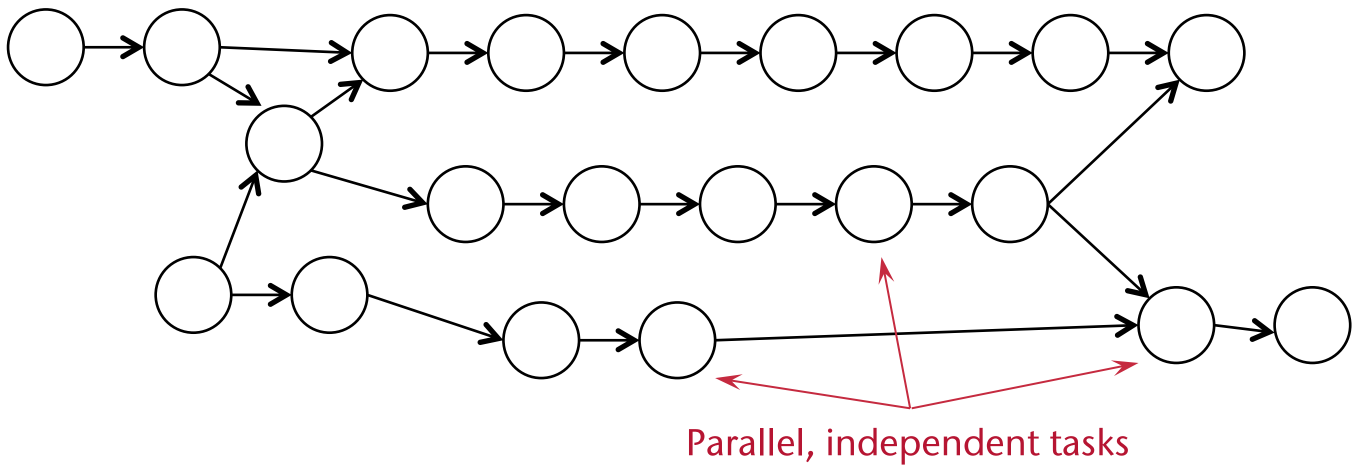
   - Note: we omitted the double-buffering and the barrier synchronization

Terminology

- Algorithmic technique: recursive/iterative doubling technique = "Accesses or actions are governed by increasing powers of 2"
  - Remember the algo for maintaining dynamic arrays? ($2^{nd}/1^{st}$ semester)

- Definitions:
  - Depth $D(n)$ = "#iterations" = parallel running time $T_p(n)$
    - (Think of the loops unrolled and "baked" into a hardware pipeline)
    - Sometimes also called step complexity
  - Work $W(n)$ = total number of operations performed by all threads together
    - With *sequential* algorithms, *work complexity* = *time complexity*
  - Work-efficient:
    A parallel algorithm is called *work-efficient*, if it performs no more work than the sequential one

- Visual definition of depth/work complexity:
  - Express computation as a dependence graph of parallel *tasks*:



  - Work complexity = total amount of work performed by all tasks
  - Depth complexity = length of the "critical path" in the graph

- Parallel algorithms should be always both work and depth efficient!

- Complexity of the Hillis-Steele algorithm:
  - Depth $d = T_p(n)$ = # iterations = $\log(n)$ $\rightarrow$ good
  - In iteration d: $n - 2^{d-1}$ adds
  - Total number of adds = work complexity

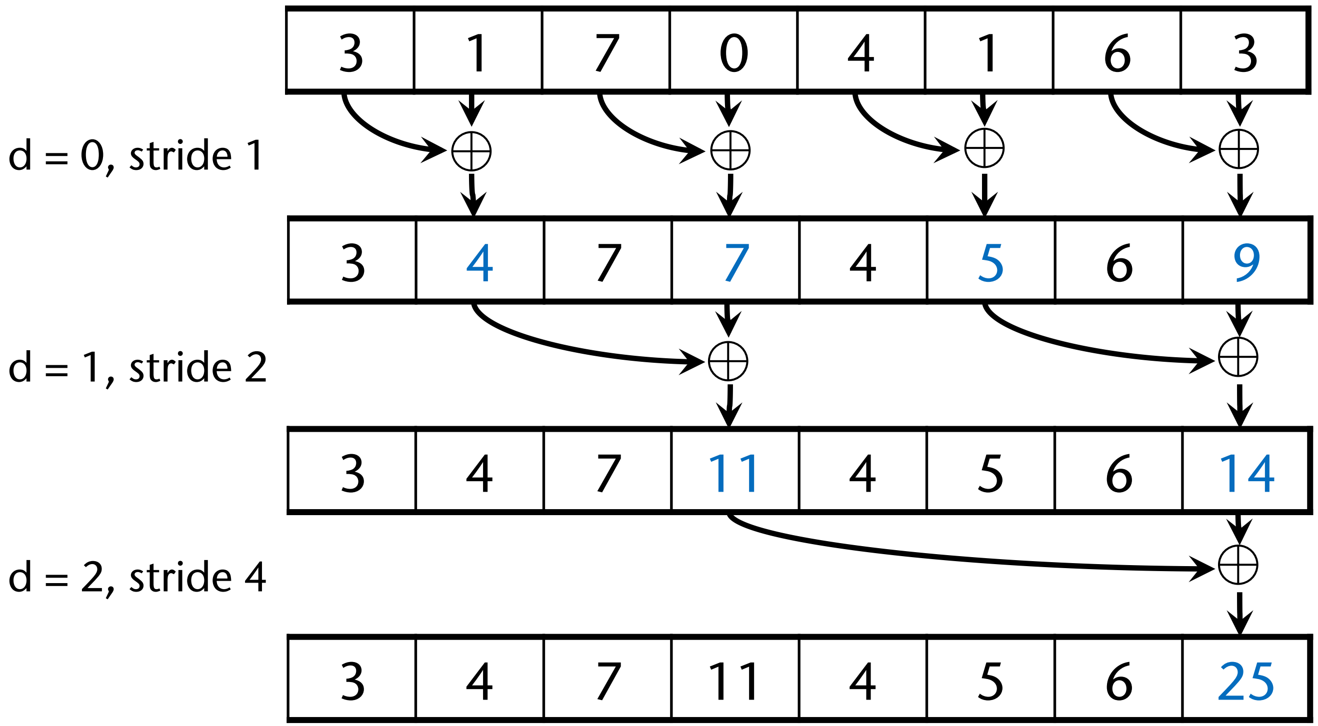
$$W(n) = \sum_{d=1}^{\log_2 n} (n - 2^{d-1}) = \sum_{d=1}^{\log_2 n} n - \sum_{d=1}^{\log_2 n} 2^{d-1} = n \cdot \log n - n \in O\big(n \log n\big)$$

- Conclusion: not work-efficient
  - A factor of $\log(n)$ can hurt: 20x for 10^6 elements

The Blelloch Algorithm (for Exclusive Scan)

- Consists of two phases: *up-sweep* (= reduction) and *down-sweep*

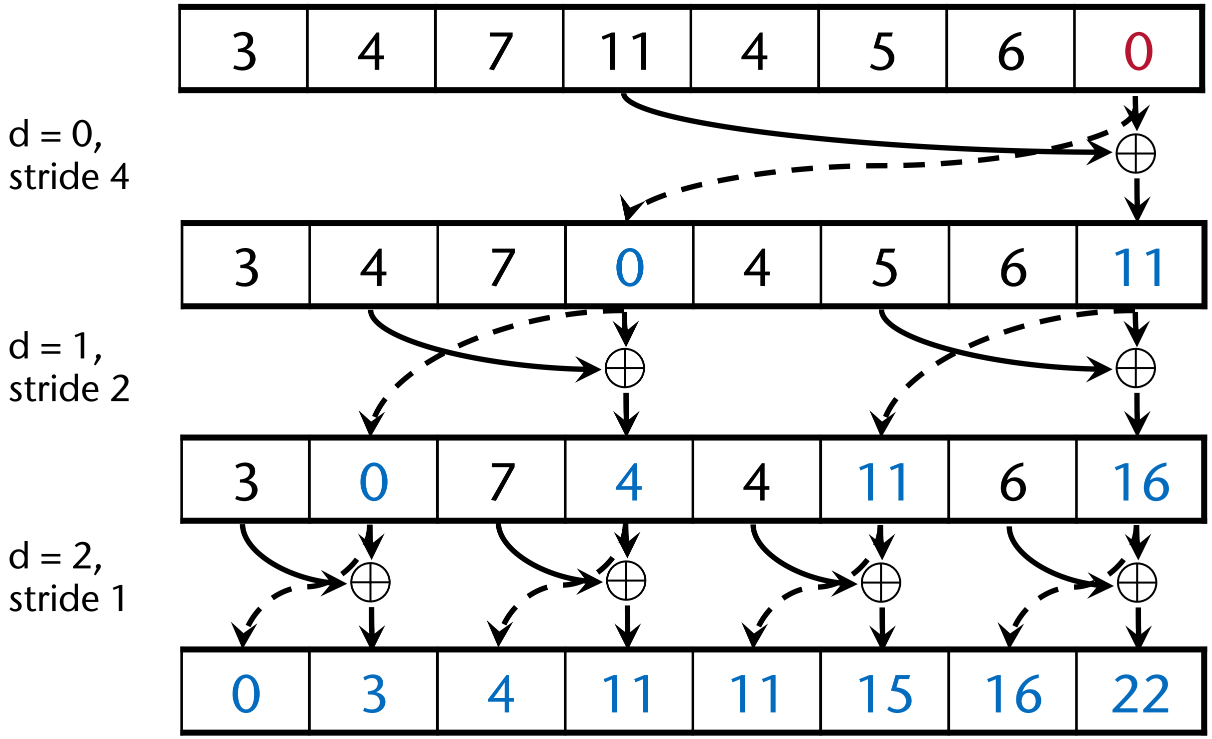
1. Up-sweep:



- Note: no double-buffering needed! (sync is still needed, of course)

2. Down-sweep:

- First: zero last element (might seem strange at first thought)



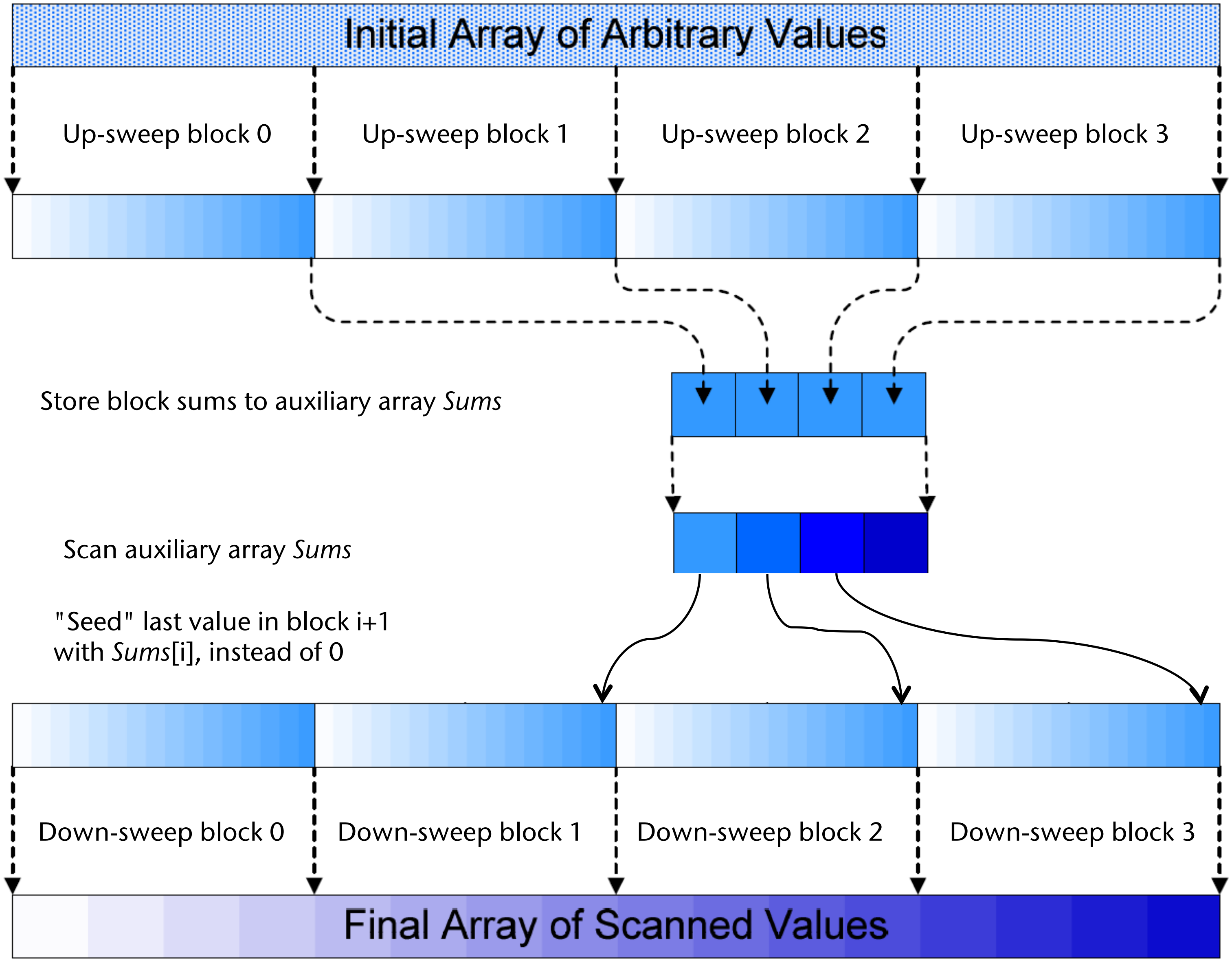
- Dashed line means "store into" (overwriting previous content)

- Depth complexity:
  - Performs $2 \cdot \log(n)$ iterations
  - $D(n) \in O(\log n)$
- Work-efficiency:
  - Number of adds: $n/2 + n/4 + .. + 1 + 1 + \ldots + n/4 + n/2$
  - Work complexity $W(n) = 2 \cdot n = O(n)$
  - The Blelloch algorithm is *work efficient*
- This *up-sweep followed by down-sweep* is a very common *pattern* in massively parallel algorithms!
- Limitations so far:
  - Only one block of threads (what if the array is larger?)
  - Only arrays with power-of-2 size

Working on Arbitrary Length Input

- One kernel launch handles up to 2*blockDim.x elements
- Partition array into blocks
  - Choose fairly small block size = 2^k, so we can easily pad array to $b \cdot 2^k$

1. Run up-sweep on each block
2. Each block writes the sum of its section (= last element after up-sweep) into a *Sums* array at blockIdx.x
3. Run prefix sum on the *Sums* array
4. Perform down-sweep on each block
5. Add *Sums*[blockIdx.x] to each element in "next" array section blockIdx.x+1


Initial Array of Arbitrary Values
Up-sweep block 0
Up-sweep block 1
Up-sweep block 2
Up-sweep block 3
Store block sums to auxiliary array *Sums*
Scan auxiliary array *Sums*
"Seed" last value in block i+1
with *Sums*[i], instead of 0
Down-sweep block 0
Down-sweep block 1
Down-sweep block 2
Down-sweep block 3
Final Array of Scanned Values

Further Optimizations

- A *real* implementation needs to do all the nitty-gritty optimizations
  - E.g., worry about *bank conflicts* (very technical, pretty complex)
- A simple & effective technique:
  - Each thread **`i`** loads 4 floats from global memory → **`float4 x`**
  - Store $\Sigma_{j=1..4}$ **`x[i][j]`** in shared memory **`a[i]`**
  - Compute the prefix-sum on **`a`** ⟶ **`â`**
  - Store 4 values back in global memory:
    - **`â[i] + x[0]`**
    - **`â[i] + x[0] + x[1]`**
    - **`â[i] + x[0] + x[1] + x[2]`**
    - **`â[i] + x[0] + x[1] + x[2] + x[3]`**
  - Experience shows: 2x faster
  - Why does this improve performance? → Brent's theorem

Brent's Theorem

- Assumption when formulating parallel algorithms: we have arbitrarily many processors
  - E.g., $O(n)$ many processors for input of size n
  - Kernel launch even reflects that!
    - Often, we run as many threads as there are input elements
    - I.e., CUDA/GPU provide us with this (nice) abstraction
- Real hardware: only has fixed number p of processors
  - E.g., on current GPUs: $p \approx 200$–2000 (depending on viewpoint)
- Question: how fast can an implementation of a massively parallel algorithm really be?

- Assumptions for Brent's theorem: PRAM model
  - No explicit synchronization needed
  - Memory access = free

- Brent's Theorem:
  Given a massively parallel algorithm A; let $D(n)$ = its depth (i.e., parallel time complexity), and $W(n)$ = its work complexity.
  Then, A can be run on a p-processor PRAM in time

$$T(n,p) \leq \left\lfloor \frac{W(n)}{p} \right\rfloor + D(n)$$

(Note the "≤")

- Proof:
  - For each iteration step i, $1 \le i \le D(n)$, let $W_i(n)$ = number of operations in that step
  - Distribute those operations on p processors:
    - Groups of $\left\lceil \frac{W_i(n)}{p} \right\rceil$ operations in parallel on the p processors
    - Takes $\left\lceil \frac{W_i(n)}{p} \right\rceil$ time steps on the PRAM
  - Overall :

$$T(n,p) = \sum_{i=1}^{D(n)} \left\lceil \frac{W_i(n)}{p} \right\rceil \le \sum_{i=1}^{D(n)} \left(\left\lfloor \frac{W_i(n)}{p} \right\rfloor + 1 \right) \le \left\lfloor \frac{W(n)}{p} \right\rfloor + D(n)$$