

- Remember the reduction operation
 - Extremely important/frequent operation → Google's *MapReduce*

- Definition **prefix sum**:

Given an input sequence $A = (a_0, a_1, a_2, \dots, a_{n-1})$,
the (inclusive) prefix sum of this sequence is the output sequence

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

where \oplus is an arbitrary binary associative operator.

- The **exclusive prefix sum** is

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

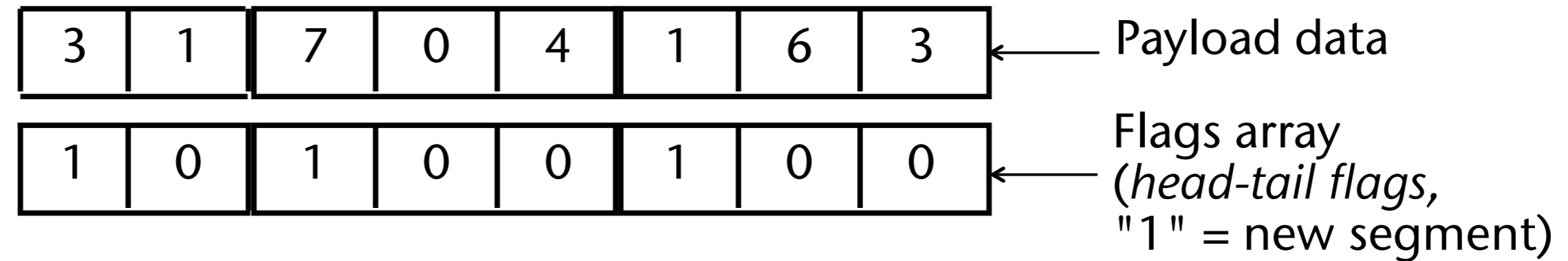
where ι is the identity/zero element, e.g., 0 for the + operator.

- 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"

Variation: Segmented Scan

- Input: segments of numbers in one large vector



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



- Forms the basis for a wide variety of algorithms:
 - E.g., Quicksort, Sparse Matrix-Vector Multiply, Convex Hull
- Note: take care to store the flags array **space- and bandwidth-efficient!** (one integer per flag is very in-efficient)

Application from "Everyday" Life

- Given:
 - A 100-inch sandwich
 - 10 persons
 - We know how many inches each person wants: [3 5 2 7 10 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
 - Make cuts in parallel with 10 knives
 - How quickly can we compute the prefix sum?



Illustration of the 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:

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

Algorithm	Model		
	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

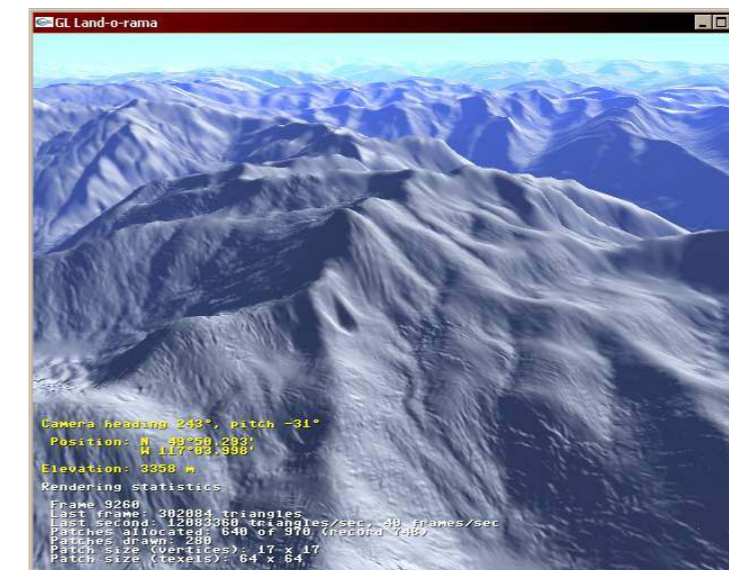
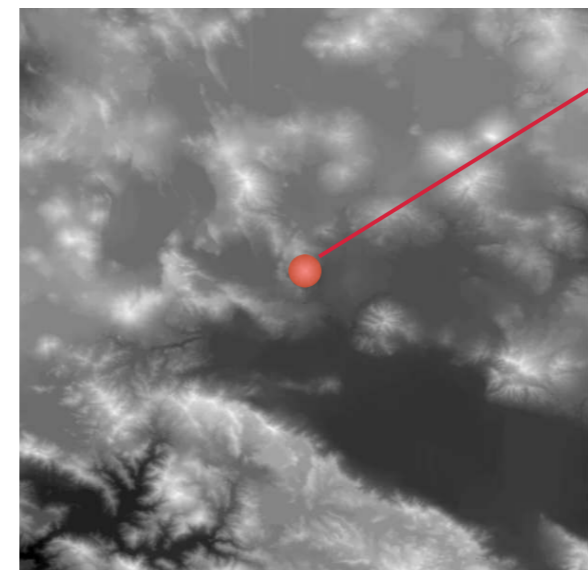
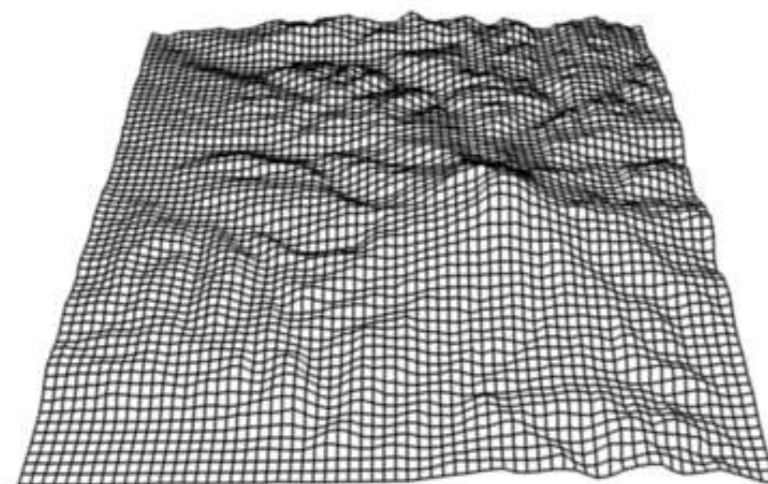
Guy E. Blelloch:
Vector Models for Data-Parallel Computing

- 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.)

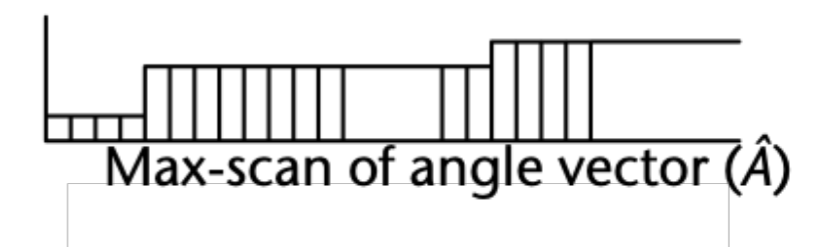
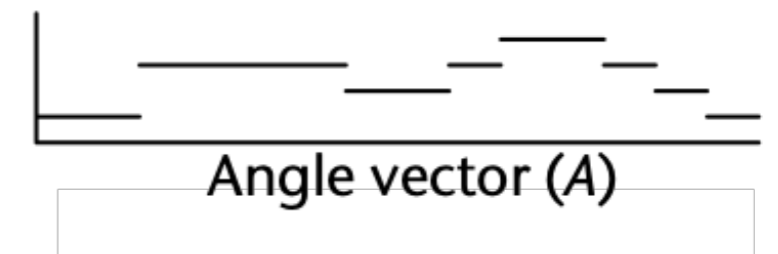
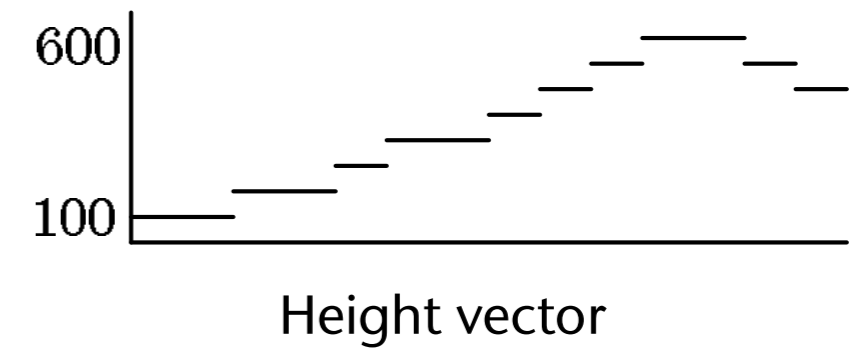


Example: Line-of-Sight

- Given:
 - Terrain as grid of height values (*height map*)
 - Point X in the grid (our "viewpoint", has a height, too)
 - 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 along our viewing direction

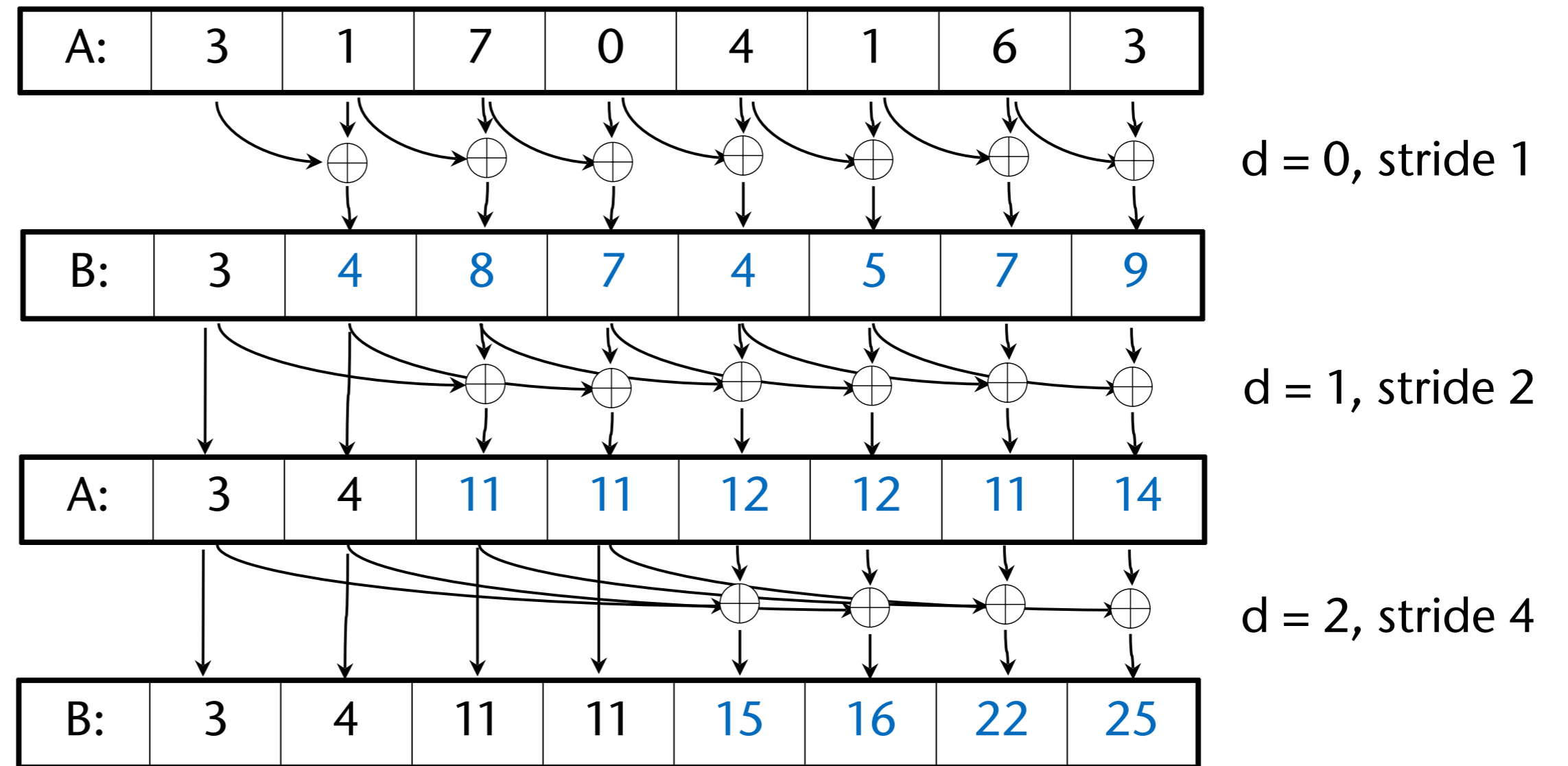


- The algorithm:
 - Convert height vector to vertical angles (as seen from X) $\rightarrow A$
 - One thread per vector element
 - Perform max-scan on angle vector (i.e., prefix sum with the max operator) $\rightarrow \hat{A}$
 - Test $\hat{a}_i < a_i$, if true then grid point is visible form X



The Hillis-Steele Algorithm (MassPar Pattern)

- Iterate $\log(n)$ times:



- Notes:

- Blue = active threads
- Each thread reads from another lane, too \rightarrow must use barrier sync
- Could save one barrier by double buffering

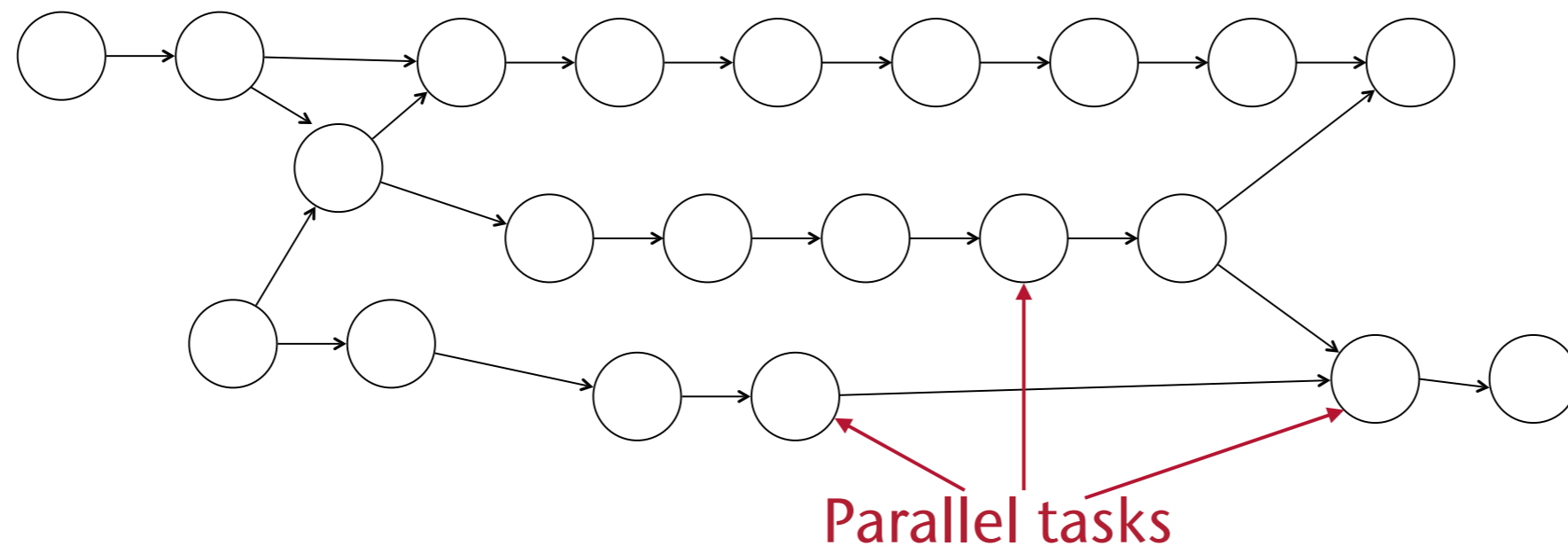
- The algorithm as pseudo-code:

```
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: barrier synchronization omitted for clarity
- Remark: precision is usually better than the naïve sequential algo
 - Because, in the parallel version, summands (in each iteration) tend to be of the same order
- Algorithmic technique: **recursive/iterative doubling technique** =
"Accesses or actions are governed by increasing powers of 2"
 - Remember the algo for maintaining dynamic arrays? (2nd/1st semester)

- **Depth complexity** $D(n)$ = "#iterations" = parallel running time $T_p(n)$
 - (Think of the loops unrolled and "baked" into a hardware pipeline)
 - Sometimes also called **step complexity**
- **Work complexity** $W(n)$ = total number of operations performed by all threads
 - 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 (in Big-O notation)

- 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

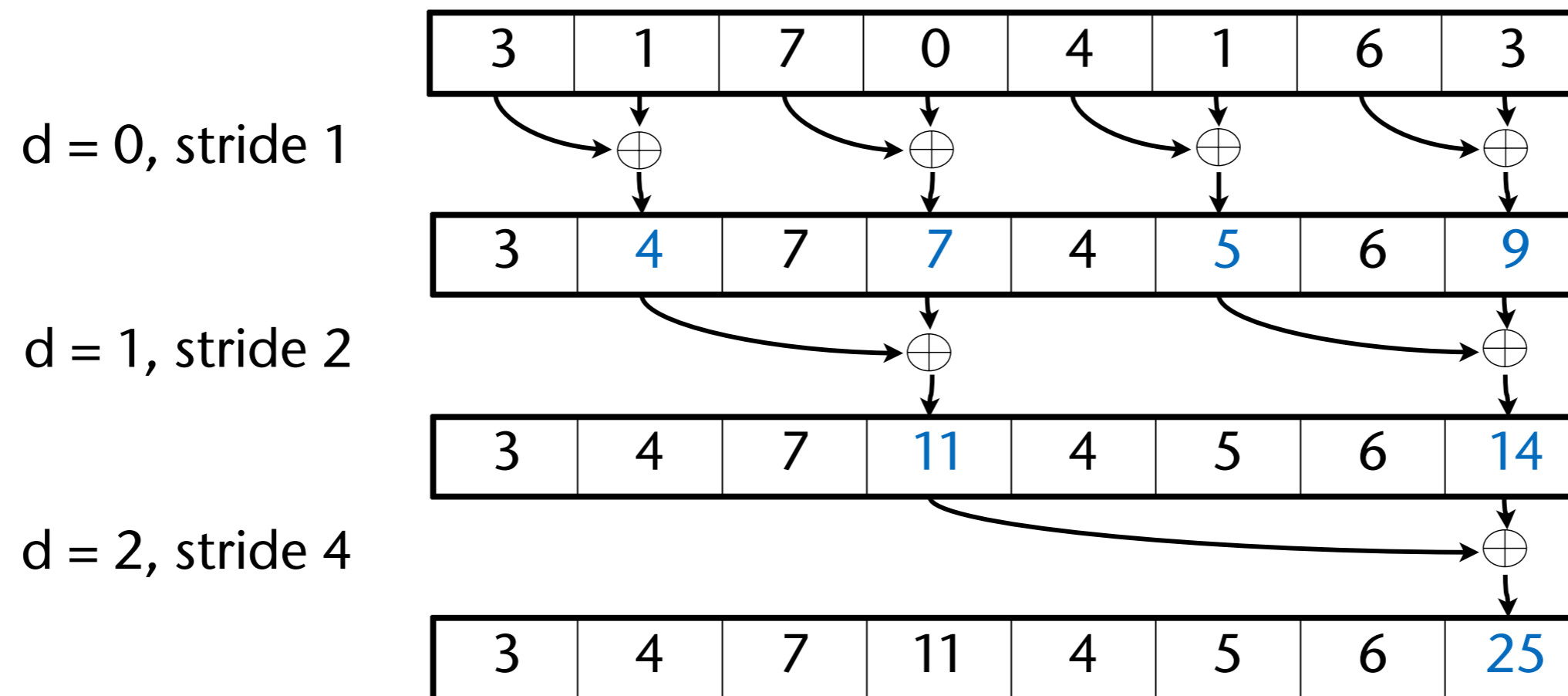
- Depth $D(n) = T_p(n) = \# \text{ iterations} = \log(n) \rightarrow \text{good}$
- In iteration d : $\# \text{ additions} = n - 2^{d-1}$
- Total number of adds = work complexity

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

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

The Blelloch Algorithm (here for Exclusive Scan)

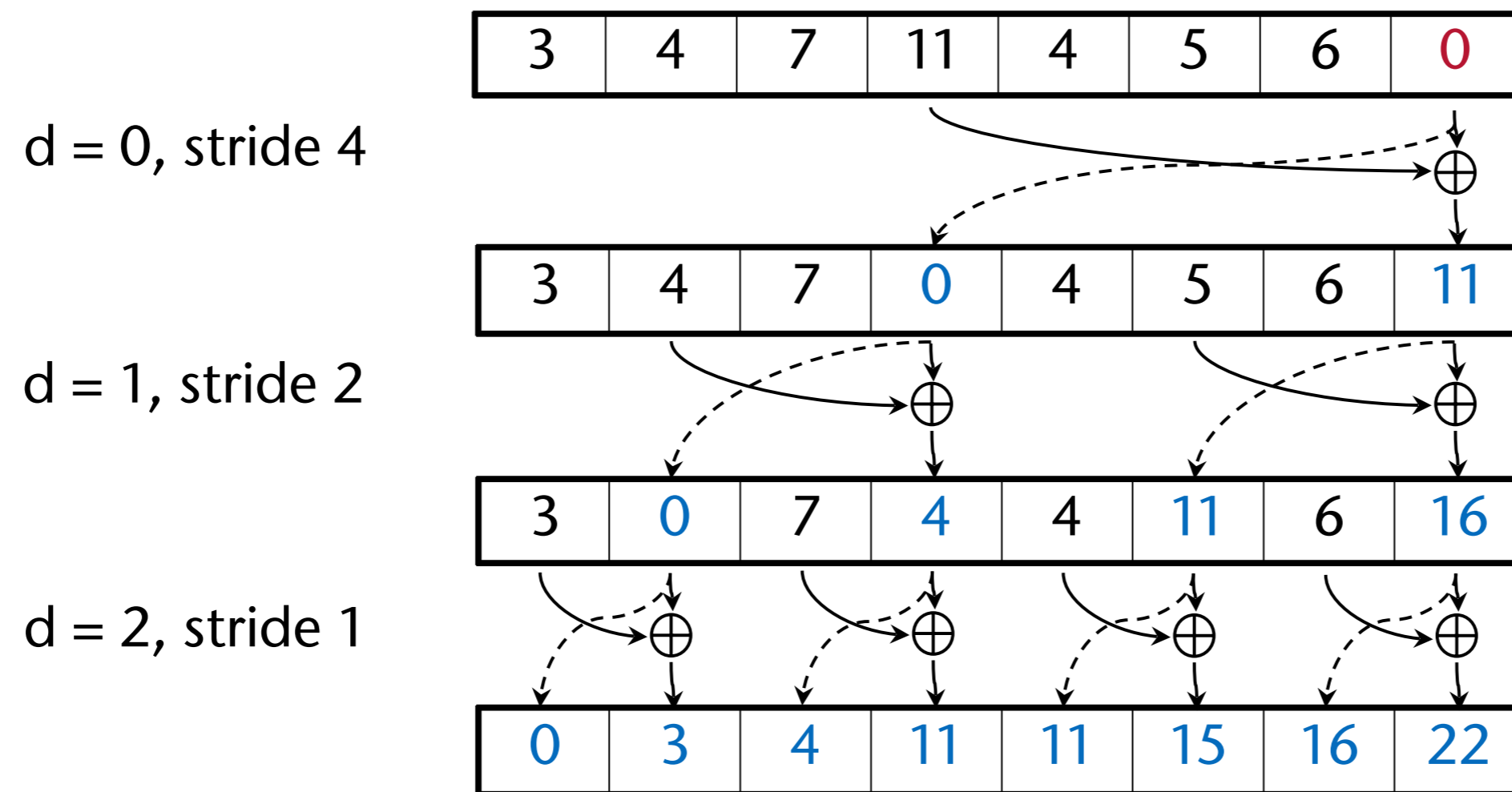
- Consists of two phases: *up-sweep* (= reduction) and *down-sweep*
1. Up-sweep:



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

2. Down-sweep:

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

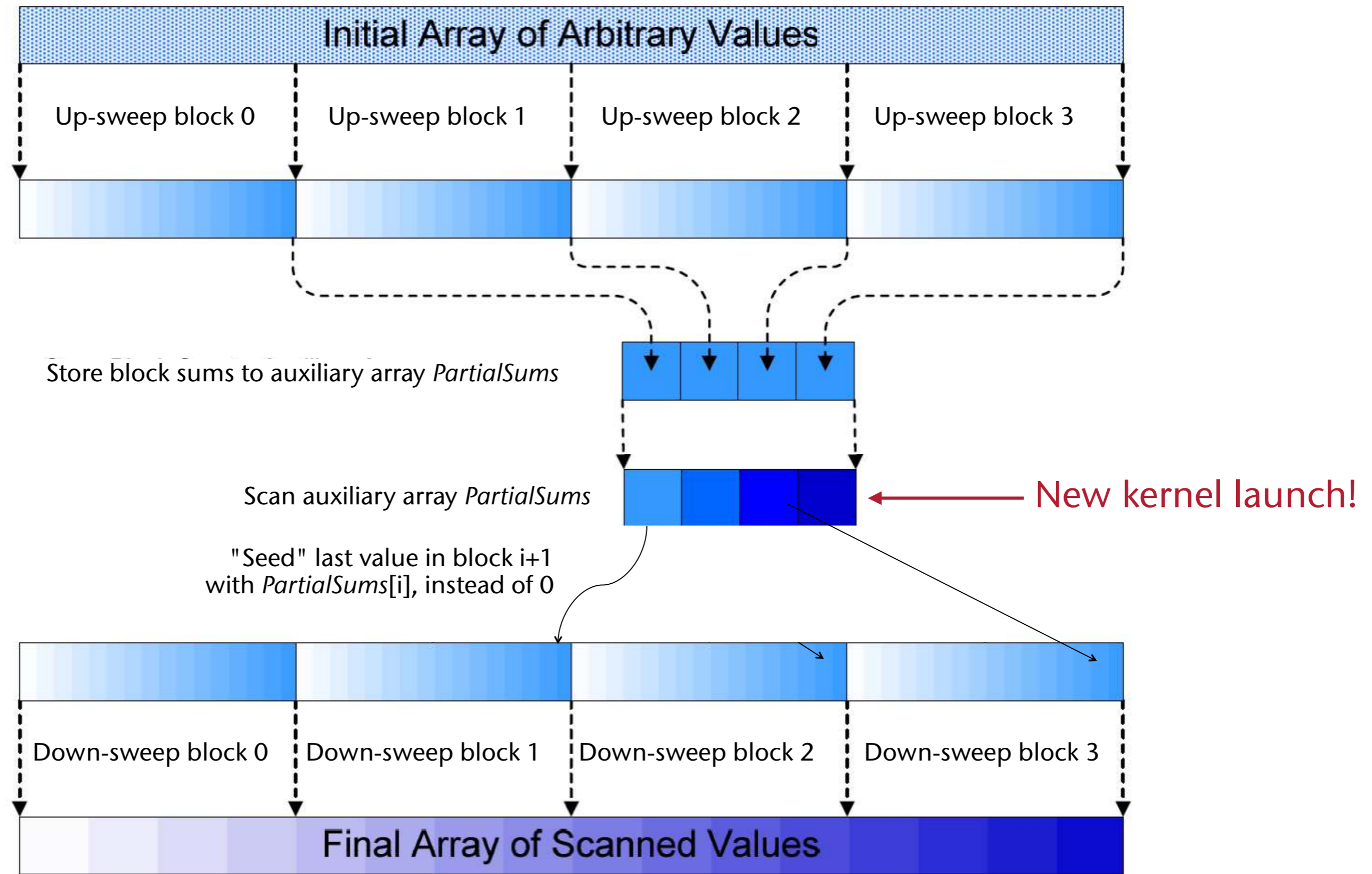


- Dashed line means "copy over" (overwriting previous content)

- Depth complexity:
 - Performs $2 \cdot \log(n)$ iterations
 - $D(n) \in O(\log n)$
- Work efficiency:
 - Number of adds: $n/2 + n/4 + \dots + 1 + 1 + \dots + n/4 + n/2$
 - 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

- Challenge: **syncthreads ()** works **only for all threads within a block**, but NOT across block borders!
- Partition array into b blocks
 - Choose fairly small block size = 2^k , so we can easily pad array to $b \cdot 2^k$
- Run up-sweep on each block
- Each block writes the *sum of its partition* (= last element after up-sweep) into a **PartialSums** array at **blockIdx.x**
- Run prefix sum on the **PartialSums** array
- Perform down-sweep on each block
- Add **PartialSums [blockIdx.x]** to *each* element in "next" array section **blockIdx.x+1**



Further Optimizations

- A simple & effective technique:
 - Each thread i loads 4 floats from global memory \rightarrow `float4 x`
 - Store $\sum_{j=0..3} x[i][j]$ in shared memory \rightarrow `a[i]`
 - Compute the exclusive prefix-sum on `a` \rightarrow `â`
 - Store 4 values back in global memory:
 - `A[4*i] = â[i] + x[0]`
 - `A[4*i+1] = â[i] + x[0] + x[1]`
 - `A[4*i+2] = â[i] + x[0] + x[1] + x[2]`
 - `A[4*i+3] = â[i] + x[0] + x[1] + x[2] + x[3]`
 - Experience shows: 2x faster
 - Why does this improve performance? \rightarrow Brent's theorem

Brent's Theorem

- Frequent 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: $p \approx 200\text{--}2000$ (depending on viewpoint and architecture)
- Question: how fast can an implementation of a parallel algorithm really be?

- Assumptions for Brent's theorem: PRAM model
 - No explicit synchronization needed
 - Memory access = free (no cost)
- 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 at most

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

(Note the " \leq ")

- Alternative statement of Brent's theorem:

$$T_p(n) = \frac{T_1(n)}{p} + T_\infty(n)$$

where $T_p(n)$ = time complexity using p processors, T_1 = sequential complexity, T_∞ = parallel complexity with unlimited number of processors.

Proof

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

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

Application of Brent's Theorem to our Optimization of Prefix-Sum

- Assume that the optimized version loads f floats into local registers
- Work complexity:
 - Without optimization: $W_1(n) = 2n$
 - With optimization: $W_2(n) = 2\frac{n}{f} + \frac{n}{f} \cdot f = n\left(1 + \frac{2}{f}\right)$
- Depth complexity:
 - Without optimization: $D_1(n) = 2 \log(n)$
 - With optimization: $D_2(n) = 2 \log\left(\frac{n}{f}\right) + 2f = 2 \log n - 2 \log f + 2f$
- If $f = 2$, then $W_2 = W_1$ and $D_2 = D_1$, i.e., we gain nothing
- If $f > 2$, **speedup** of version 2 (optimized) over version 1 (original):

$$\text{Speedup}(n) = \frac{T_1(n)}{T_2(n)} = \frac{\frac{W_1(n)}{p} + D_1(n)}{\frac{W_2(n)}{p} + D_2(n)} \approx \frac{2\frac{n}{p}}{\frac{n}{p}\left(1 + \frac{2}{f}\right)} = \frac{2f}{f + 2}$$

Other Consequences of Brent's Theorem

- Obviously, $\text{Speedup}(n) \leq p$
- In the sequential world, time = work: $T_S(n) = W_S(n)$
- In the parallel world: $T_P(n) = \frac{W_P(n)}{p} + D(n)$
- Our speedup is $\text{Speedup}(n) = \frac{T_S(n)}{T_P(n)} = \frac{W_S(n)}{\frac{W_P(n)}{p} + D(n)}$
- Assume, $W_P(n) \in \Omega(W_S(n))$
i.e., our parallel algorithm would do asymptotically more work
- Then, $\text{Speedup}(n) = \frac{W_S(n)}{\Omega(W_S(n)) + D(n)} \rightarrow 0$ as $n \rightarrow \infty$
because, on real hardware, p is bounded
- This is the reason why we want **work-efficient** parallel algorithms!

- Now, look at work-efficient parallel algorithms, i.e. $W_P(n) \in \Theta(W_S(n))$
- Then,

$$\text{Speedup}(n) = \frac{W(n)}{\frac{W(n)}{p} + D(n)}$$

- In this situation, we will achieve the optimal speedup of $O(p)$, so long as

$$p \in O\left(\frac{W(n)}{D(n)}\right)$$

- Consequence: given two work-efficient parallel algorithms, the one with the smaller depth complexity is better, because we can run it on hardware with more processors (cores) and still obtain a speedup of p over the sequential algorithm (in theory).

We say this algorithm **scales better**.

Limitations of Brent's Theorem

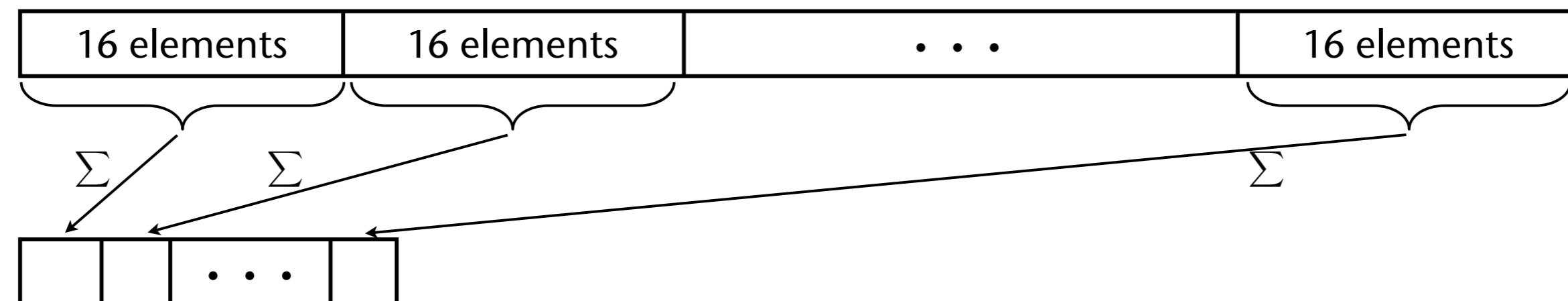
- Brent's theorem is based on the PRAM model
- That model makes a number of unrealistic assumptions:
 - Memory access has zero latency
 - Memory bandwidth is infinite
 - No synchronization among processors (threads) is necessary
 - Arithmetic operations cost unit time
- With current hardware, rather the opposite is realistic

Using Tensor Cores for Scan/Prefix Sum and Reduction

- Reduction ($\hat{a} = \sum_i a_i$) could be formulated as matrix multiplication:

$$\hat{a} = \underbrace{\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & & & & \\ 0 & 0 & 0 & \dots & 0 \end{pmatrix}}_P \begin{pmatrix} a_1 & 0 & 0 & \dots & 0 \\ a_2 & 0 & 0 & \dots & 0 \\ a_3 & \vdots & & & \\ a_n & 0 & 0 & \dots & 0 \end{pmatrix}$$

- Regular segmented reduction of length 16:

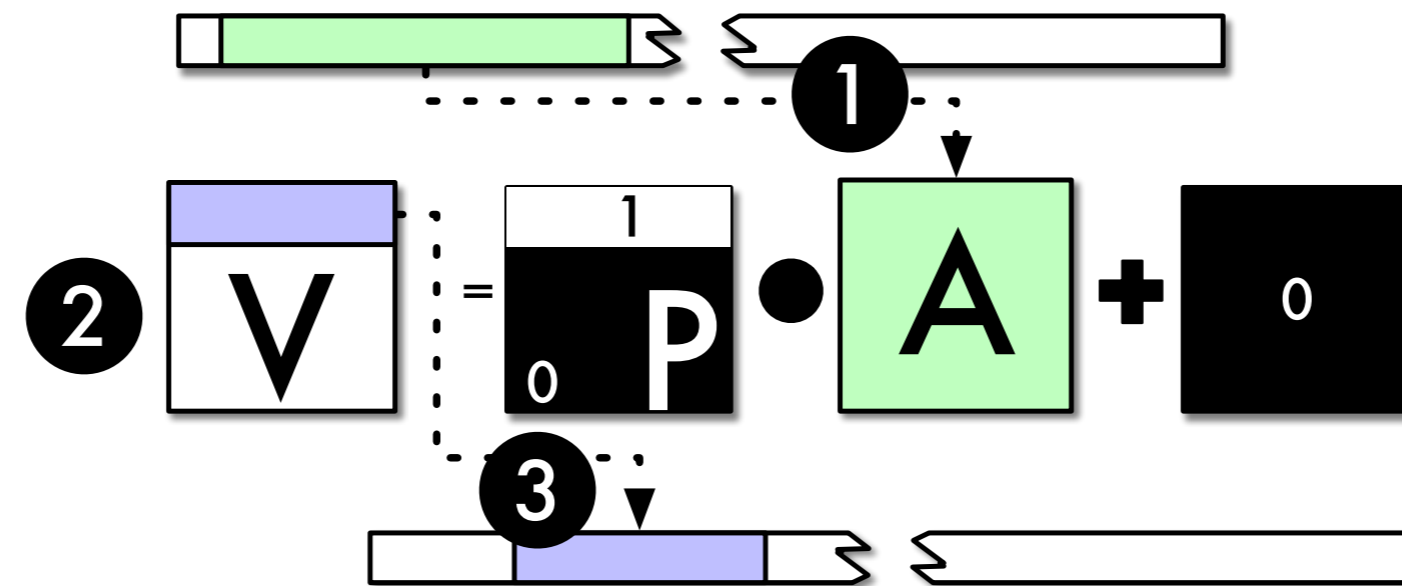


Algorithm for regular 16-segmented reduction

- Each warp loads parts of input array of size 256 = 16 segments of size 16, then performs a warp-level MMA (i.e., uses the tensor cores)

```

Reduction16( in array A, out array R ):
fragment_a ← init matrix P
idx = global offset into A for each warp
fragment_b ← load tile A[idx..idx+255] in column major
M = P·A + 0 // = mma_sync() in CUDA
if lane index < 16:
    R[ idx/16 + lane-index ] = M[lane-index]
    
```



Extension to Scan Over 256 Elements

- Input: $V = V[0], \dots, V[255]$
- Load V in to 16×16 matrix A in row-major order: $A = \begin{pmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,16} \\ a_{2,1} & a_{2,2} & \dots & a_{2,16} \\ \vdots & & & \vdots \\ a_{16,1} & a_{16,2} & \dots & a_{16,16} \end{pmatrix}$
- Define upper right 1-matrix: $U = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 0 & 1 & \dots & 1 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$
- Multiplication yields row-wise inclusive scan, i.e., regular segmented **inclusive** prefix sum:

$$A \cdot U = \begin{pmatrix} a_{1,1} & \dots & \sum_{j=1}^{16} a_{1,j} \\ a_{2,1} & \dots & \sum_{j=1}^{16} a_{2,j} \\ \vdots & & \vdots \\ a_{16,1} & \dots & \sum_{j=1}^{16} a_{16,j} \end{pmatrix}$$

- Multiplication of A with lower-left 1-matrix (0's on the diagonal here!) yields a column-wise, **exclusive** prefix sum:

$$L = \begin{pmatrix} 0 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \\ \vdots & & \ddots & \vdots \\ 1 & 1 & \dots & 0 \end{pmatrix} \quad L \cdot A = \begin{pmatrix} 0 & 0 & \dots & 0 \\ a_{1,1} & a_{1,2} & \dots & a_{1,16} \\ a_{1,1} + a_{2,1} & a_{1,2} + a_{2,2} & \dots & a_{1,16} + a_{2,16} \\ \sum_{j=1}^{15} a_{1,j} & \sum_{j=1}^{15} a_{2,j} & \dots & \sum_{j=1}^{15} a_{16,j} \end{pmatrix}$$

- Multiplication at right-hand side with an all-1-matrix yields row-wise **reduction**; all elements in the same row in the output matrix will be equal

$$J = \begin{pmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{pmatrix}$$

- Multiplication of $L \cdot A$ with J yields reduction of all elements in A *before* that row:

$$L \cdot A \cdot J = \begin{pmatrix} 0 & 0 & \dots & 0 \\ \sum_{i=j}^{16} a_{1,j} & \sum_{i=j}^{16} a_{1,j} & \dots & \sum_{i=j}^{16} a_{1,j} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=1}^{16} a_{16,j} & \sum_{j=1}^{16} a_{16,j} & \dots & \sum_{j=1}^{16} a_{16,j} \end{pmatrix}$$

- Add the segmented scan $A \cdot U$, resulting in the inclusive prefix sum over 256 elements:

$$L \cdot A \cdot J + A \cdot U = \begin{pmatrix} a_{1,1} & a_{1,1} + a_{1,2} & \dots & \sum_{j=1}^{16} a_{1,j} \\ \sum_{j=1}^{16} a_{1,j} + a_{2,1} & \sum_{j=1}^{16} a_{1,j} + a_{2,1} + a_{2,2} & \dots & \sum_{i=1}^2 \sum_{j=1}^{16} a_{i,j} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{i=1}^{15} \sum_{j=1}^{16} a_{i,j} + a_{16,1} & \sum_{i=1}^{15} \sum_{j=1}^{16} a_{i,j} + a_{16,1} + a_{16,2} & \dots & \sum_{i=1}^{16} \sum_{j=1}^{16} a_{i,j} \end{pmatrix}$$

Extension to Scan Over Whole Block and Grid

- Per block: precondition is $N = b \cdot 256$, $b \leq 256$ (for sake of simplicity)

```
PrefixSumN( in array A, out array S ):
```

```
perform warp-level prefix-sum's over segments of A, 256 elements each  
gather last element of each segment in array R
```

```
sync all threads within block
```

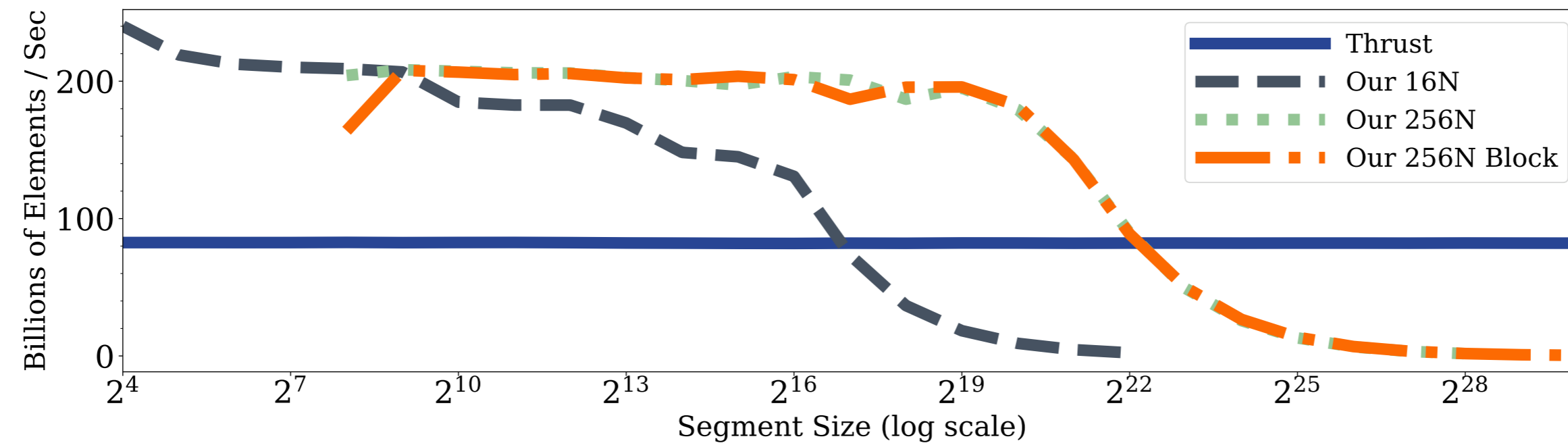
```
warp 0 performs exclusive prefix sum over R
```

```
sync all threads within block
```

```
all threads add R[warpIdx-1] to "their" element and output it to S
```

- Per grid:
 - Launch 3 kernels for 3 phases, similar to above procedure
 - First, block-wise (i.e., segmented) scan, gather last values of each segment (= reduced blocks) in intermediate array; second, prefix-sum over those values; third, distribute and accumulate values from intermediate scan to blocks

Performance for Segmented Prefix Sum



$N = 2^{31}$ elements

89% - 97% of theoretical peak throughput

Digression: Radix-Sort

- Modeled after sorting machines of post routing centers (but with a twist!)
- Disadvantages:
 - Not generic like Quicksort, which require only a *compare* operator on pairs of elements
 - Works only on elements with a known, pre-defined, fixed-length numeric representation (e.g., 32 bits)
 - Different representations require different versions of radix sort
- Advantage: very efficient!



- Observation: integers can be represented with any base r
- Naive (intuitive) idea:
 - Sort all elements according to the most significant digit into bins (one bin per digit)
 - Sort bin 0 using radix sort recursively
 - Sort bin 1 recursively, etc. ...
- This is called **MSD radix sort** (MSD = *most significant digit*)
- For the algorithm on the next slide:
 - Choose radix r and fix it
 - Define $z(t, a) = t$ -th digit of number a when represented over base r , where $t=0$ denotes the least significant digit (usually the right-most digit)

```
A = array of numbers
i = current digit used for sorting ( 0 <= i <= d-1 )
d = total number of digits (same for all keys)
def msd_radix_sort( A, i, d ):

    # init array of r empty lists = [ [], [], [], ... ]
    bin = r * [[]]

    # distribute all A's in bins according to z(i,..)
    for j in range(0, len(A) ):
        bin[ z(i, A[j]) ].append( A[j] )

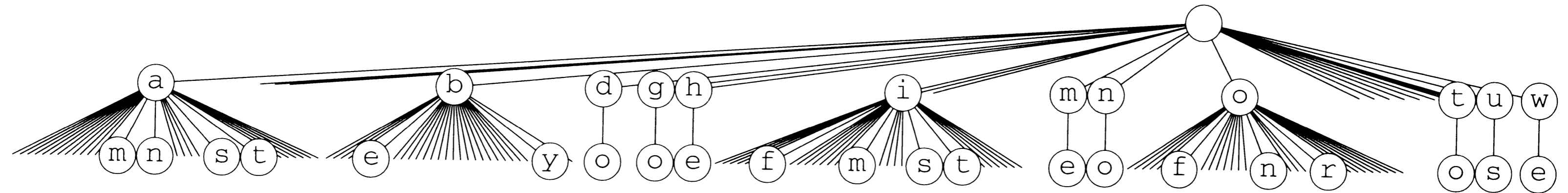
    # sort bins
    if i >= 0:
        for j in range(0, r):
            msd_radix_sort( bin[j], i-1, d )

    # gather bins
    A = []
    for j in range(0, r):
        A.extend( bin[j] )
        bin[j] = []
```

Example

- Keys = integers with 64 bits
- Size of input = 2^{24} (ca. 16m)
- We choose $r = 2^8 = 256$ as base
 - E.g. "digits" = characters in fixed-length strings
- On the first recursion level, the algo checks the left-most byte of the keys and distributes each key into one of 256 bins
- Average (expected) size of the bins (assuming uniform distribution of the keys) = $2^{24} / 2^8 = 2^{16} = 65536$

- Recursion tree:



- Problem: in each recursion, we need to save $r-1$ many bins (the remaining bin is passed down to the recursively called function)
 - Lots of house keeping necessary
 - Solutions: either use marker arrays like with Counting Sort
 - Or, use arrays of lists (lots of allocations / deallocations)

Solution: LSD Radix-Sort (aka. Backward Radix-Sort)

- First, sort according to least-significant digit, then according to least but second digit, etc.; do all of this *in place*, no auxiliary arrays needed!
- Let d = number of digits, digit 0 = least-significant one
- The algorithm:

```
lsd_radix_sort( A ):  
  for i = 0, ..., d-1:  
    do a stable sort on A with the  
      i-th digit of the elements as the key
```

- Use, e.g., Counting Sort inside the loop (check your Data Structures & Algorithms course)

Example

- Sort 12 letters according to the post code (zip code)
- In the first iteration, consider only the last digit

Brief 1	nach	3 5 0 3 7	Marburg	Brief 11	nach	8 2 3 4 0	Feldafing
Brief 2	nach	7 1 6 7 2	Marbach	Brief 2	nach	7 1 6 7 2	Marbach
Brief 3	nach	3 5 2 8 8	Wohratal	Brief 4	nach	3 5 2 8 2	Rauschenberg
Brief 4	nach	3 5 2 8 2	Rauschenberg	Brief 5	nach	8 8 6 6 2	Überlingen
Brief 5	nach	8 8 6 6 2	Überlingen	Brief 1	nach	3 5 0 3 7	Marburg
Brief 6	nach	7 9 6 9 9	Zell	Brief 8	nach	8 0 6 3 7	München
Brief 7	nach	8 0 6 3 8	München	Brief 12	nach	8 2 3 2 7	Tutzing
Brief 8	nach	8 0 6 3 7	München	Brief 3	nach	3 5 2 8 8	Wohratal
Brief 9	nach	5 5 1 2 8	Mainz	Brief 7	nach	8 0 6 3 8	München
Brief 10	nach	5 5 4 6 9	Simmern	Brief 9	nach	5 5 1 2 8	Mainz
Brief 11	nach	8 2 3 4 0	Feldafing	Brief 6	nach	7 9 6 9 9	Zell
Brief 12	nach	8 2 3 2 7	Tutzing	Brief 10	nach	5 5 4 6 9	Simmern

Letters before the first iteration

Letters after the first iteration

- Notice: letters with the same digit did **not** change their position relative to each other!

- Sort by last but second digit

Brief 11	nach	8 2 3 4 0	Feldafing	Brief 12	nach	8 2 3 2 7	Tutzing
Brief 2	nach	7 1 6 7 2	Marbach	Brief 9	nach	5 5 1 2 8	Mainz
Brief 4	nach	3 5 2 8 2	Rauschenberg	Brief 1	nach	3 5 0 3 7	Marburg
Brief 5	nach	8 8 6 6 2	Überlingen	Brief 8	nach	8 0 6 3 7	München
Brief 1	nach	3 5 0 3 7	Marburg	Brief 7	nach	8 0 6 3 8	München
Brief 8	nach	8 0 6 3 7	München	Brief 11	nach	8 2 3 4 0	Feldafing
Brief 12	nach	8 2 3 2 7	Tutzing	Brief 5	nach	8 8 6 6 2	Überlingen
Brief 3	nach	3 5 2 8 8	Wohratal	Brief 10	nach	5 5 4 6 9	Simmern
Brief 7	nach	8 0 6 3 8	München	Brief 2	nach	7 1 6 7 2	Marbach
Brief 9	nach	5 5 1 2 8	Mainz	Brief 4	nach	3 5 2 8 2	Rauschenberg
Brief 6	nach	7 9 6 9 9	Zell	Brief 3	nach	3 5 2 8 8	Wohratal
Brief 10	nach	5 5 4 6 9	Simmern	Brief 6	nach	7 9 6 9 9	Zell

Letters before the second iteration

Letters after the second iteration

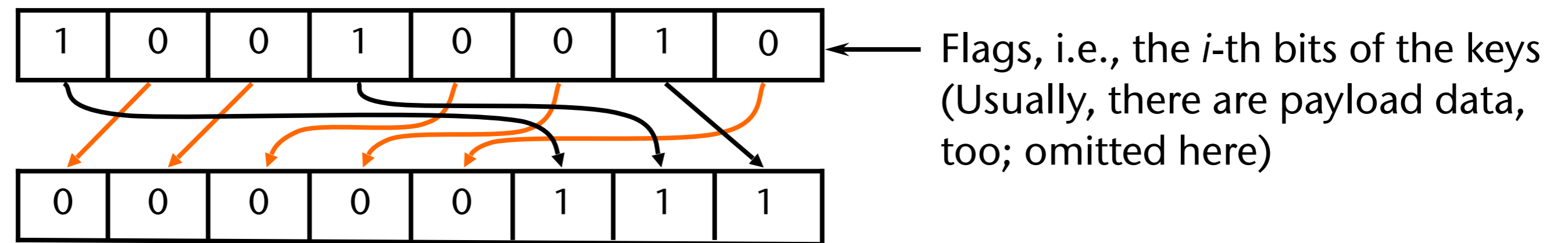
Brief 12	nach	8 2 3 2 7	Tutzing	Brief 1	nach	3 5 0 3 7	Marburg
Brief 9	nach	5 5 1 2 8	Mainz	Brief 9	nach	5 5 1 2 8	Mainz
Brief 1	nach	3 5 0 3 7	Marburg	Brief 4	nach	3 5 2 8 2	Rauschenberg
Brief 8	nach	8 0 6 3 7	München	Brief 3	nach	3 5 2 8 8	Wohratal
Brief 7	nach	8 0 6 3 8	München	Brief 12	nach	8 2 3 2 7	Tutzing
Brief 11	nach	8 2 3 4 0	Feldafing	Brief 11	nach	8 2 3 4 0	Feldafing
Brief 5	nach	8 8 6 6 2	Überlingen	Brief 10	nach	5 5 4 6 9	Simmern
Brief 10	nach	5 5 4 6 9	Simmern	Brief 8	nach	8 0 6 3 7	München
Brief 2	nach	7 1 6 7 2	Marbach	Brief 7	nach	8 0 6 3 8	München
Brief 4	nach	3 5 2 8 2	Rauschenberg	Brief 5	nach	8 8 6 6 2	Überlingen
Brief 3	nach	3 5 2 8 8	Wohratal	Brief 2	nach	7 1 6 7 2	Marbach
Brief 6	nach	7 9 6 9 9	Zell	Brief 6	nach	7 9 6 9 9	Zell

Brief 8	nach	8 0 6 3 7	München	Brief 1	nach	3 5 0 3 7	Marburg
Brief 7	nach	8 0 6 3 8	München	Brief 4	nach	3 5 2 8 2	Rauschenberg
Brief 2	nach	7 1 6 7 2	Marbach	Brief 3	nach	3 5 2 8 8	Wohratal
Brief 12	nach	8 2 3 2 7	Tutzing	Brief 9	nach	5 5 1 2 8	Mainz
Brief 11	nach	8 2 3 4 0	Feldafing	Brief 10	nach	5 5 4 6 9	Simmern
Brief 1	nach	3 5 0 3 7	Marburg	Brief 2	nach	7 1 6 7 2	Marbach
Brief 9	nach	5 5 1 2 8	Mainz	Brief 6	nach	7 9 6 9 9	Zell
Brief 4	nach	3 5 2 8 2	Rauschenberg	Brief 8	nach	8 0 6 3 7	München
Brief 3	nach	3 5 2 8 8	Wohratal	Brief 7	nach	8 0 6 3 8	München
Brief 10	nach	5 5 4 6 9	Simmern	Brief 12	nach	8 2 3 2 7	Tutzing
Brief 5	nach	8 8 6 6 2	Überlingen	Brief 11	nach	8 2 3 4 0	Feldafing
Brief 6	nach	7 9 6 9 9	Zell	Brief 5	nach	8 8 6 6 2	Überlingen

Letters after the fifth iteration

Parallel Radix Sort, Based on the Split Operation

- We can use base=2 (radix=2); nice consequence: we only need to maintain 2 bins, and we can re-use the input array to hold both bins
- The **split operation**: rearrange elements according to a flag



- Note: split maintains order within each group! (i.e., it is *stable*)
- Use double buffering to prevent expensive synchronization among threads

- Radix sort (massively parallel):

```
radix_sort( array a, int len ):  
  for i = 0...numbits-1: // important: go from low to high bit!  
    split(i, a)         // split a, based on bit i of the keys
```

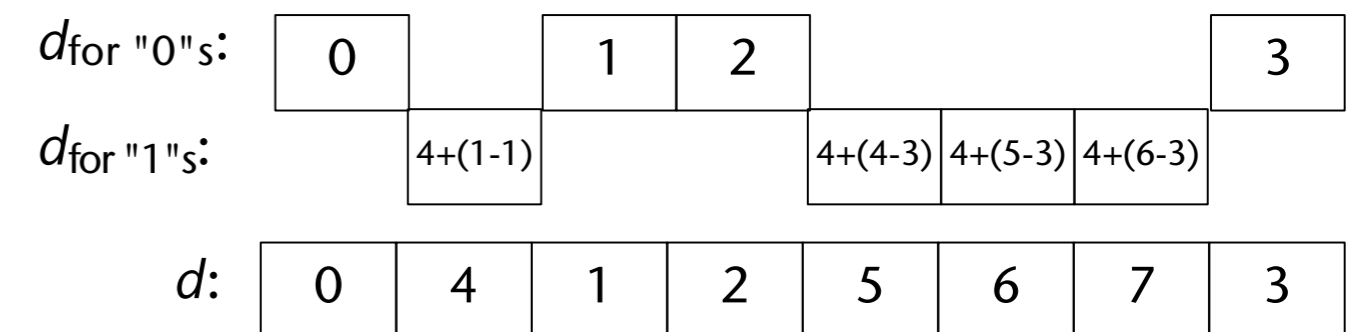
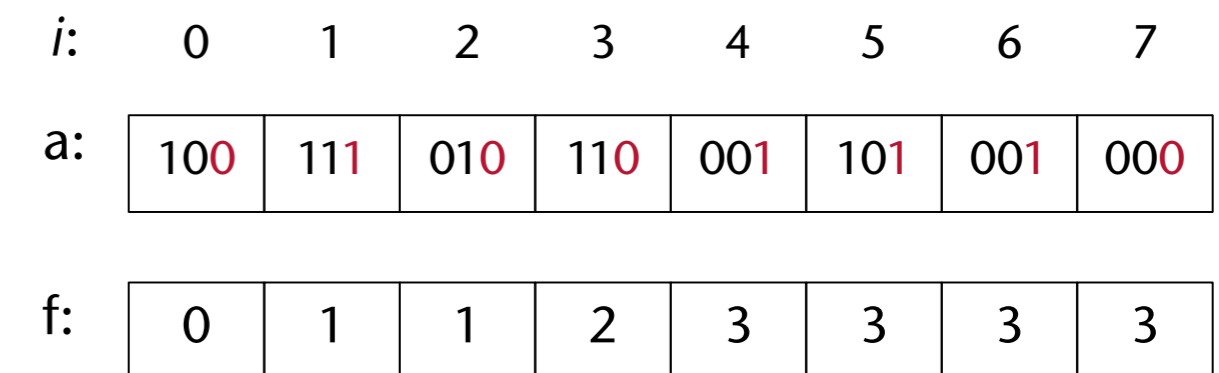
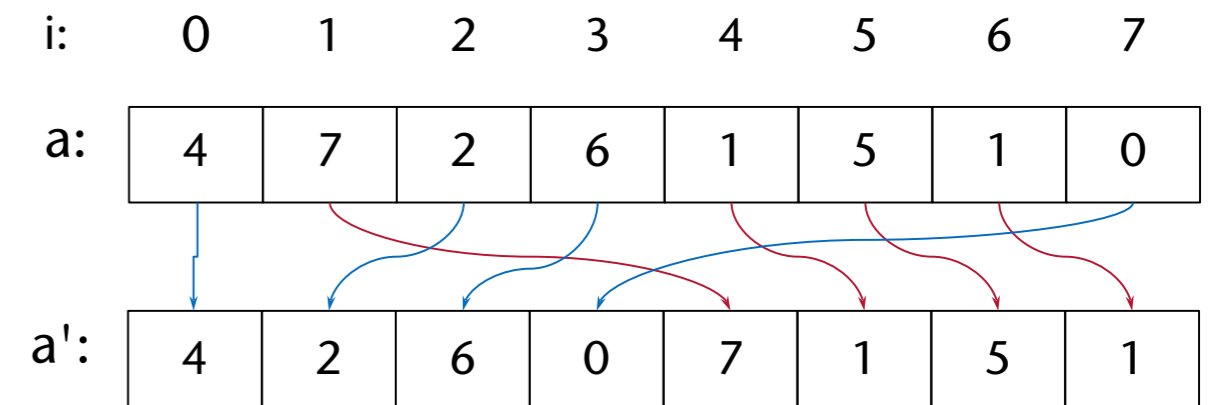
where `split(i, a)` rearranges `a` by moving all keys that have bit `i = 0` to the front, and all keys that have bit `i = 1` to the back (bit 0 = LSB)

- Reminder: stability of `split` is **essential!**
- Note: main job of the `split` operation is to compute "which key goes where"
- Hint: the prefix-sum is probably up to the job :-)

Algorithm for the Massively-Parallel Split Operation

- Split's job:
 - Determine new index for each element
 - Then perform the permutation (stable!)
- Algorithm (by way of the example):
 - Consider lowest bit of the keys
 - 1. Compute exclusive "0"-scan: $f_i = \# \text{ 0's in } (a_0, \dots, a_{i-1})$
 - 2. Set $F = \text{total number of 0's} = \begin{cases} f_{n-1} + 1 & , a_{n-1} = 0 \\ f_{n-1} & , a_{n-1} = 1 \end{cases}$
 - 3. Construct $d = \text{new positions of the } a_i\text{'s}$
 - If $a_i\text{'s bit} = 0 \rightarrow \text{new position } d_i = f_i$
 - If $a_i\text{'s bit} = 1 \rightarrow \text{new position } d_i = F + (i - f_i)$, because $i - f_i = \# \text{ 1's to the left of } a_i$

Example: split based on bit 0



- A conceptual algorithm for the "0"-scan:
 - Extract the relevant bit (conceptually only)
 - Invert the bit
 - Compute regular prefix sum with "+" operation
- In a real implementation, you would, of course, implement this as a native "0"-scan routine with a special "+" operation in the first iteration!
- Depth complexity:

$O(b \cdot \log(n))$, where $b = \# \text{bits per integer}$, and $n = \# \text{ elements}$

 - Amounts to $O(b^2)$, or $O(\log^2(n))$

a:

100	111	010	110	001	101	001	000
-----	-----	-----	-----	-----	-----	-----	-----

a':

1	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

f:

0	1	1	2	3	3	3	3
---	---	---	---	---	---	---	---

Stream Compaction

- Given: input stream A , and a *flag/predicate* for each a_i
- Goal: output stream A' that contains **only** a_i 's, for which $\text{flag} = \text{true}$
- Example:
 - Given: array of upper and lower case letters
 - Goal: delete lower case letters and compact the others to the front of the array
- Solution:
 - Just like with the split operation, except we don't compute indices for the "to-be-deleted" elements
- Frequent task, sometimes A/flags are not given explicitly (e.g., collision detection)
- Sometimes also called **list packing**, or **stream packing**

a:

A	x	C	P	h	w	b	Z
---	---	---	---	---	---	---	---

a':

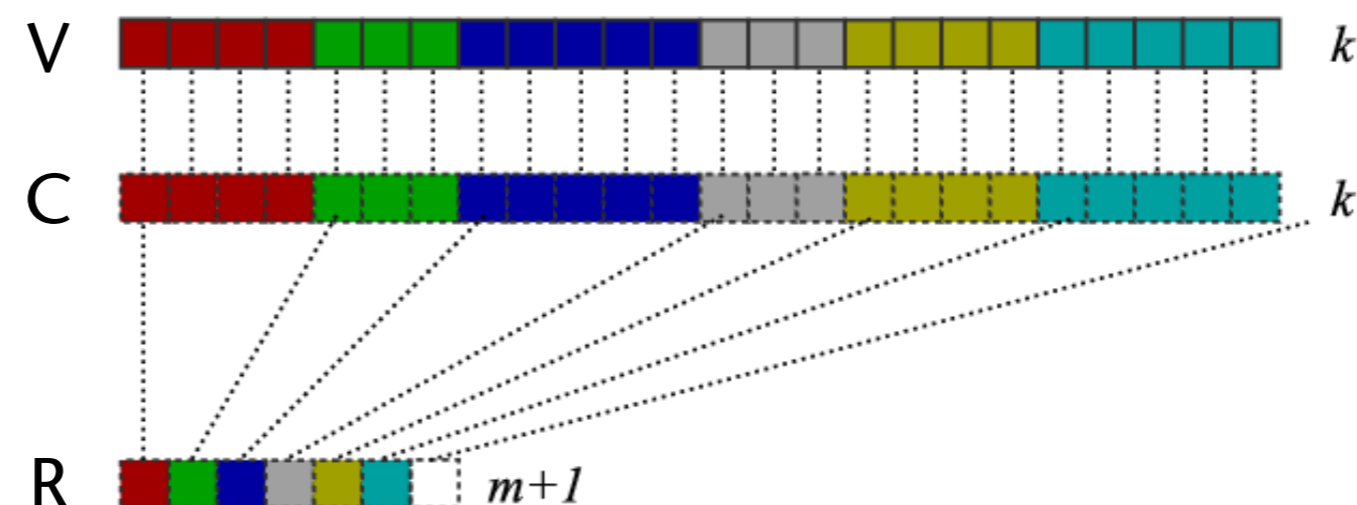
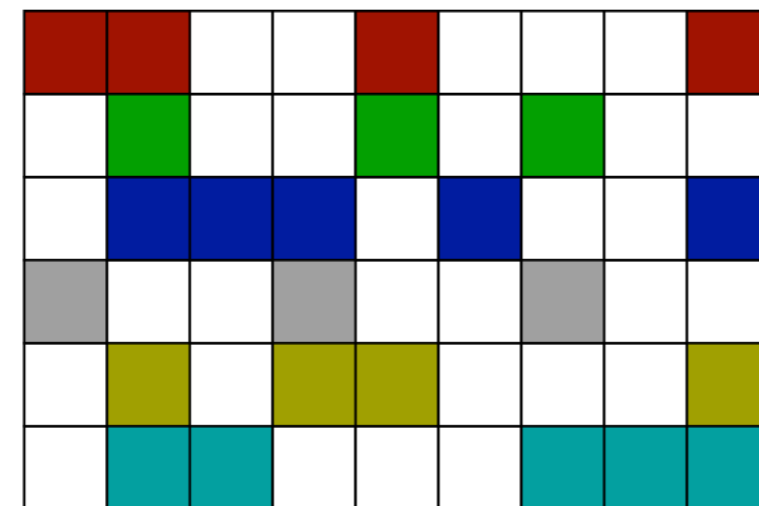
1	0	1	1	0	0	0	1
---	---	---	---	---	---	---	---

b:

A	C	P	Z				
---	---	---	---	--	--	--	--

Sparse Matrices

- "Unstructured" sparse matrices:
 - Most common storage format is **Compressed Sparse Row (CSR)**
 - Matrix M , size $m \times n$, k non-zero elements (a.k.a. "nnz")
 - Stored in three arrays V , C , R
 - Row i of matrix M is stored in $V_{R_i}, \dots, V_{R_{i+1}-1}$
 - C contains column indices: element V_j in M 's i -th row represents element M_{i,C_j}



Example

$$M = \begin{pmatrix} a_0 & 0 & 0 & a_1 & 0 \\ 0 & a_2 & 0 & 0 & 0 \\ 0 & 0 & a_3 & 0 & 0 \\ 0 & a_4 & 0 & a_5 & a_6 \\ 0 & 0 & 0 & 0 & a_7 \end{pmatrix}$$

$$V = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$$

$$C = (0, 3, 1, 2, 1, 3, 4, 4)$$

$$R = (0, 2, 3, 4, 7, 8)$$

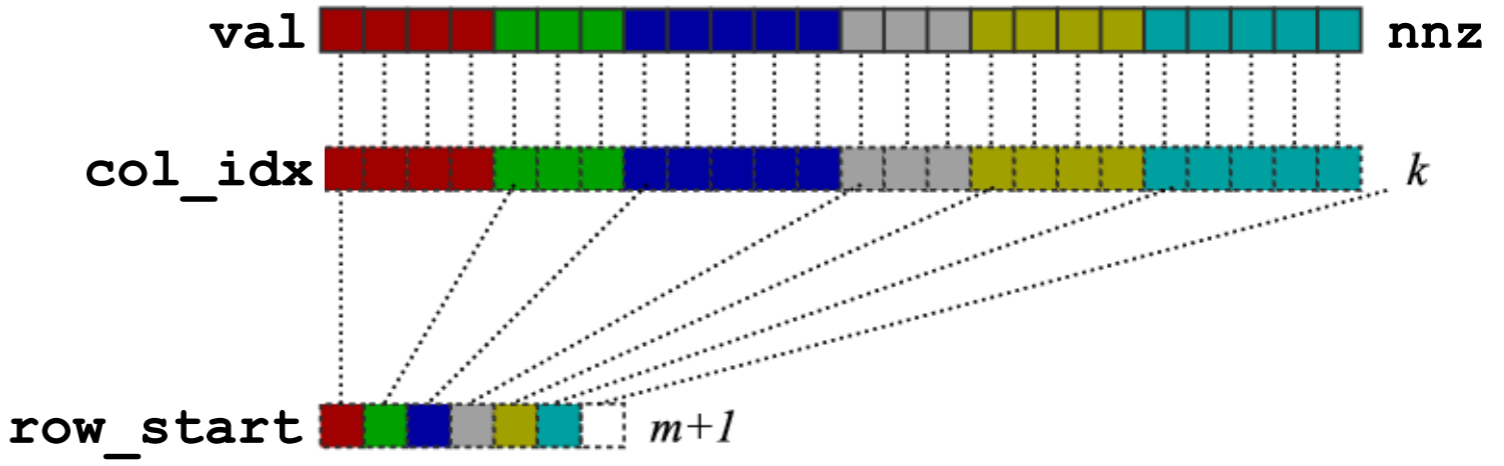
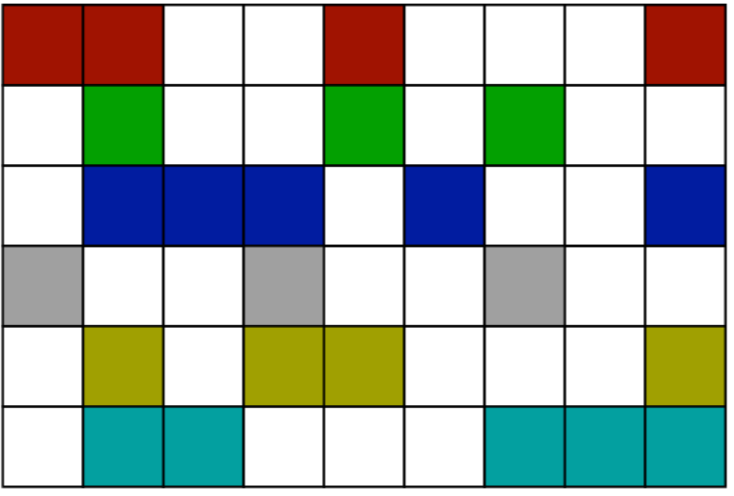
- Implementation in C:

```

struct {
    int n_rows;           // number of rows
    int nnz;              // = k = total number of non-zero elements
    int row_start[n_rows+1];
    int col_idx[nnz];
    double val[nnz];
}
    
```

where

$n_rows = m$,
 $nnz = k$,
 $val = V$,
 $col_idx = C$,
 $row_start = R$



Sparse Matrix-Vector Multiplication (SPMV)

- Task: $y = Mx$, where M is given as CSR

$$V = (a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7)$$

$$C = (0, 3, 1, 2, 1, 3, 4, 4)$$

$$R = (0, 2, 3, 4, 7, 8,)$$

- Multiply each element in V with its corresponding element in x :

$$V'_i = V_i \cdot x_{C_i}$$

$$V' = (a_0x_0, a_1x_3, a_2x_1, a_3x_2, \\ a_4x_1, a_5x_3, a_6x_4, a_7x_4)$$

- Compute flags array, signifying row starts:

$$F_i = 1 \Leftrightarrow i \in R$$

$$F = (1, 0, 1, 1, 1, 0, 0, 1)$$

- Inclusive segmented scan (one segment per row): $V' \rightarrow V''$

$$V'' = (a_0x_0, a_0x_0 + a_1x_3, a_2x_1, a_3x_2, \\ a_4x_1, a_4x_1 + a_5x_3, \\ a_4x_1 + a_5x_3 + a_6x_4, a_7x_4)$$

- Retrieve elements for y : $y_i = V''_{R_{i+1}-1}$

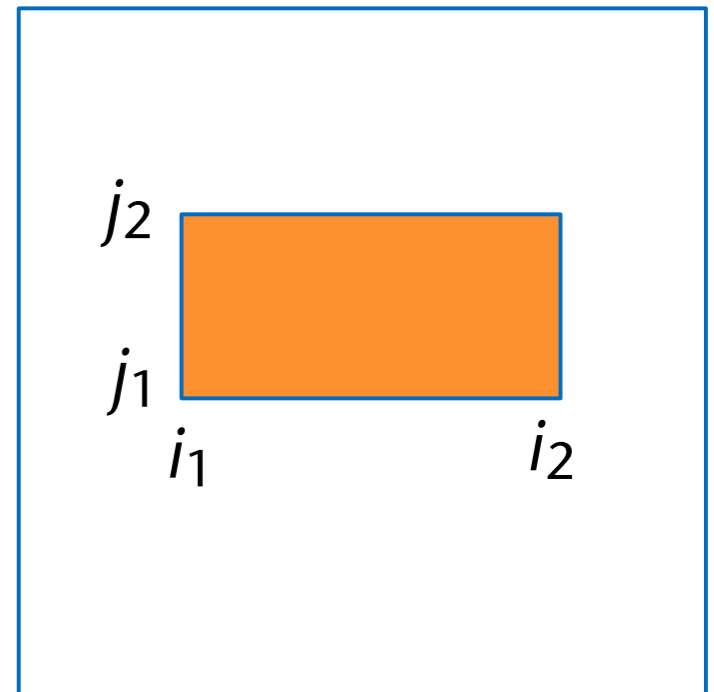
$$y_3 = V''_{R_4-1} = a_4x_1 + a_5x_3 + a_6x_4$$

Summed-Area Tables / Integral Images

- Given: 2D array T of size $w \times h$
- Wanted: a data structure that allows to compute

$$\sum_{k=i_1}^{i_2} \sum_{l=j_1}^{j_2} T(k, l)$$

for any i_1, i_2, j_1, j_2 in $O(1)$ time



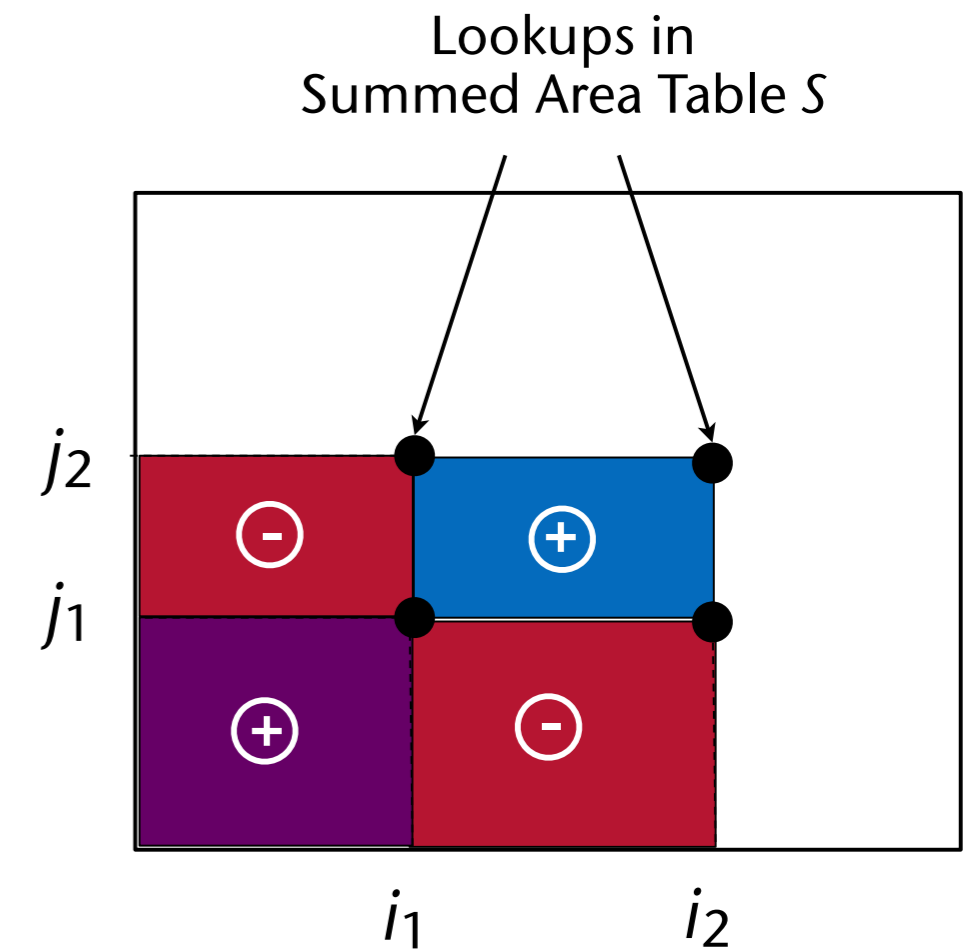
- The trick:

$$\sum_{k=i_1}^{i_2} \sum_{l=j_1}^{j_2} T(k, l) = \sum_{k=1}^{i_2} \sum_{l=1}^{j_2} T(k, l) - \sum_{k=1}^{i_1} \sum_{l=1}^{j_2} T(k, l) - \sum_{k=1}^{i_2} \sum_{l=1}^{j_1} T(k, l) + \sum_{k=1}^{i_1} \sum_{l=1}^{j_1} T(k, l)$$

- Define $S(i, j) = \sum_{k=1}^i \sum_{l=1}^j T(k, l)$

- With that, we can rewrite the sum:

$$\sum_{k=i_1}^{i_2} \sum_{l=j_1}^{j_2} T(k, l) = S(i_2, j_2) - S(i_1, j_2) - S(i_2, j_1) + S(i_1, j_1)$$



- Definition:
Given a 2D array of numbers, T , the **summed area table** S stores for each index (i,j) the sum of all elements in the rectangle $(0,0)$ and (i,j) (inclusively):

$$S(i,j) = \sum_{k=1}^i \sum_{l=1}^j T(k,l)$$

- Like the prefix-sum, but for higher dimensions
 - Summed area tables can also be defined for higher dimensions
- In computer vision, it is often called **integral image**
- Example:

Input

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

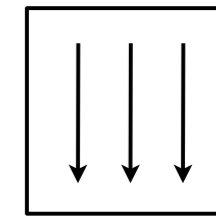
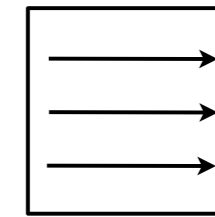
Summed Area Table

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

The Algorithm

- 2 phases (for 2D)

1. Do h prefix-sums horizontally (one per kernel launch)



2. Do w prefix-sums vertically (ditto)

- In order to maintain *coalesced memory access*): horizontal scan, transpose img., horiz. scan
- Or use texture memory (?)

- Depth complexity for d dimensions, $w = h$, and ignore transposition $d \cdot w^{d-1} \log w$

- Caveat: beware of precision loss in integer/floating-point arithmetic

- Assumption: each T_{ij} needs b bits

- Consequence: number of bits needed for $S_{wh} = \log w + \log h + b$

- Example: 1024x1024 grey scale image, each pixel = 8 bits $\rightarrow \geq 28$ bits needed in S

Increasing the Precision

- The following techniques actually apply to prefix-sums, too!

1. "Signed offset" representation:

- Set $T'(i, j) = T(i, j) - \bar{t}$
where $\bar{t} = \text{average of } T = \frac{1}{wh} \sum_{i=1}^w \sum_{j=1}^h T(i, j)$
- Effectively "removes the DC component from the signal"

- Consequence:

$$S'(i, j) = \sum_{k=1}^i \sum_{l=1}^j T'(k, l) = S(i, j) - i \cdot j \cdot \bar{t}$$

i.e., the values of S' are now in the same order as the values of T (less bits have to be thrown away during the summation)

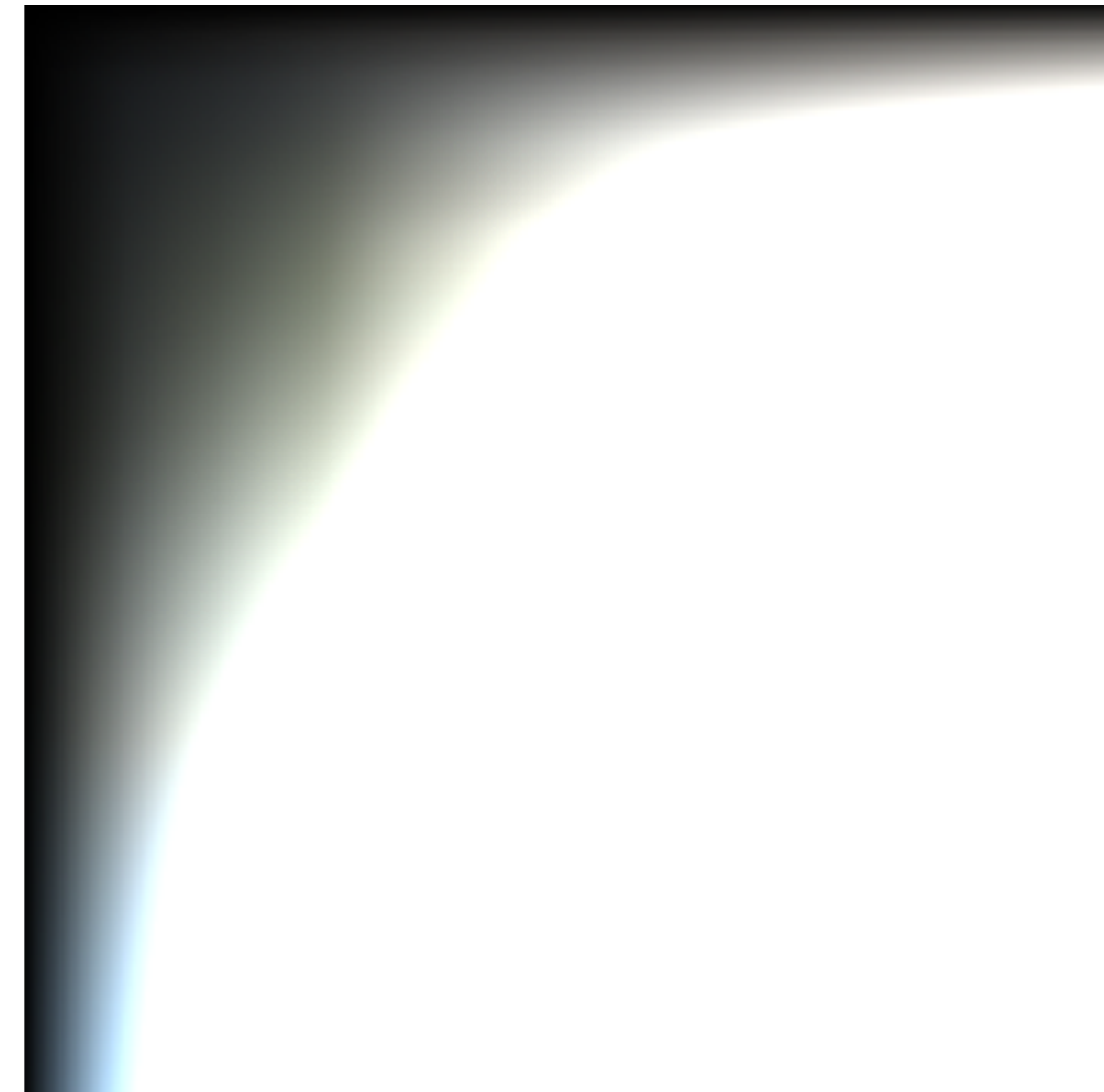
- Note 1: we need to set aside 1 bit (sign bit)
- Note 2: $S'(w, h) = 0$ (modulo rounding errors)

Example

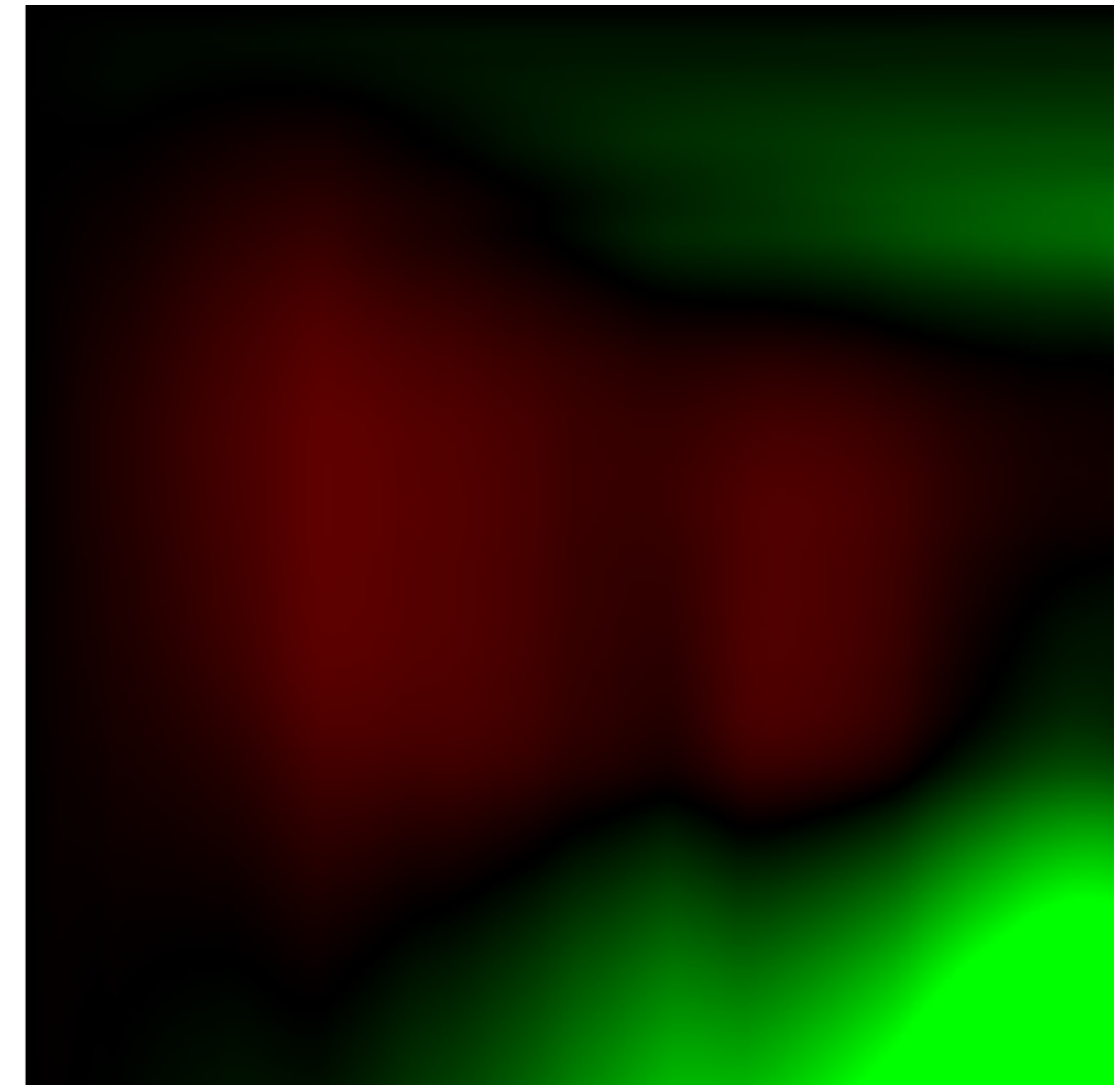
Input image



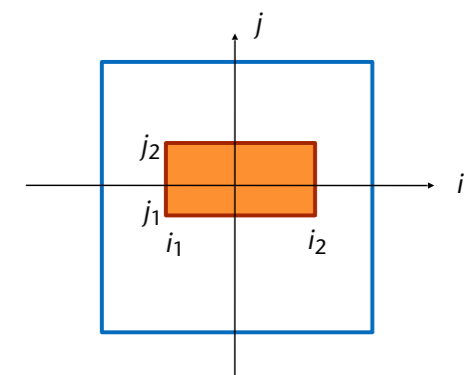
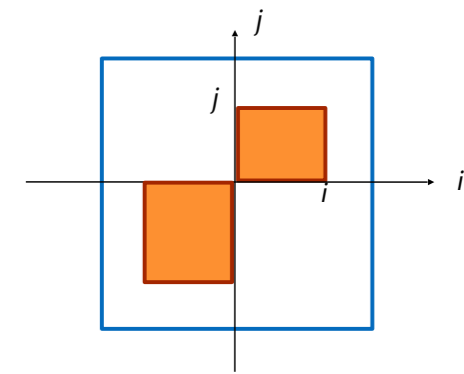
Original summed area table



With improved precision using "offset" representation



- Move the "origin" of the i, j "coordinate frame":
 - Compute 4 different S -tables, one for each quadrant
 - Result: each S -table comprises only $\frac{1}{4}$ of the pixels/values of T
- For computation of $\sum_{k=i_1}^{i_2} \sum_{l=j_1}^{j_2} T(k, l)$
do a simple case switch



Results

- Compute integral image
- From that, compute

$$\begin{aligned} & S(i, j) \\ & - S(i - 1, j) \\ & - S(i, j - 1) \\ & + S(i - 1, j - 1) \end{aligned}$$

- Should yield the original image (theoretically)

With methods 1 & 2

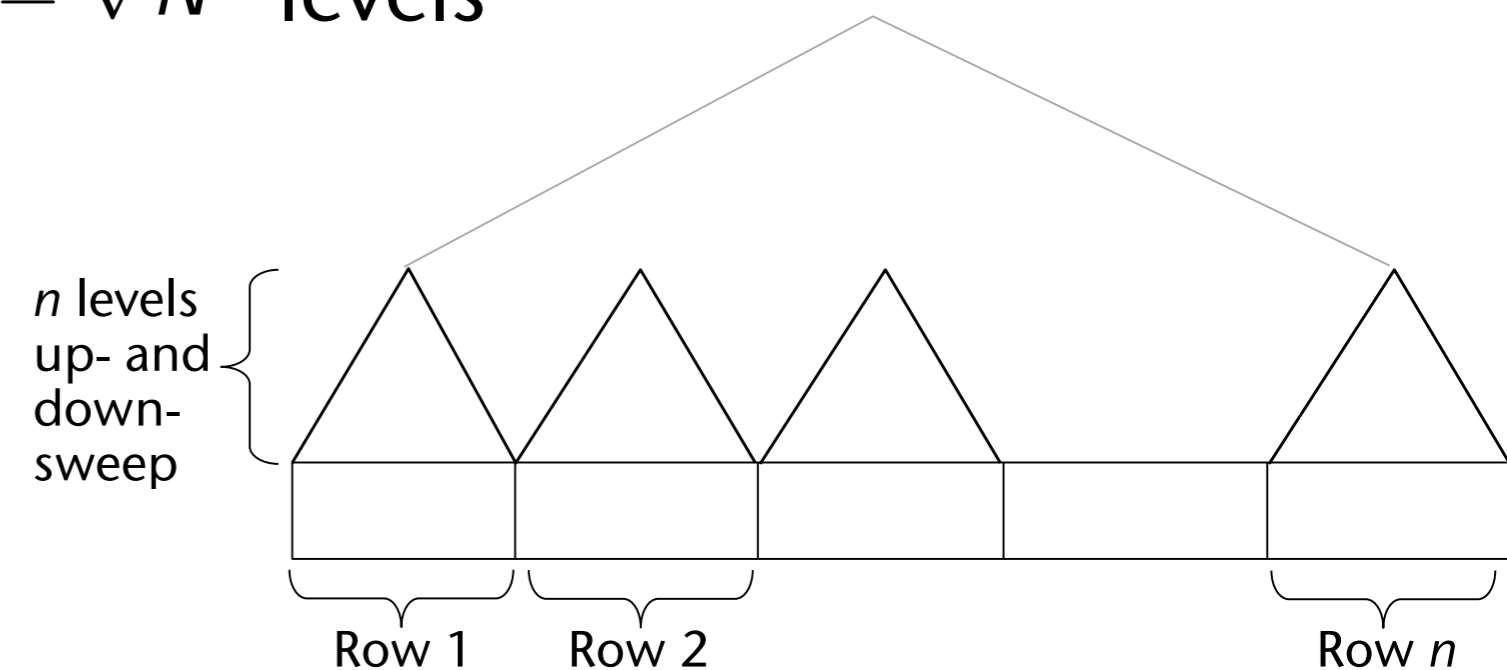


Simple method



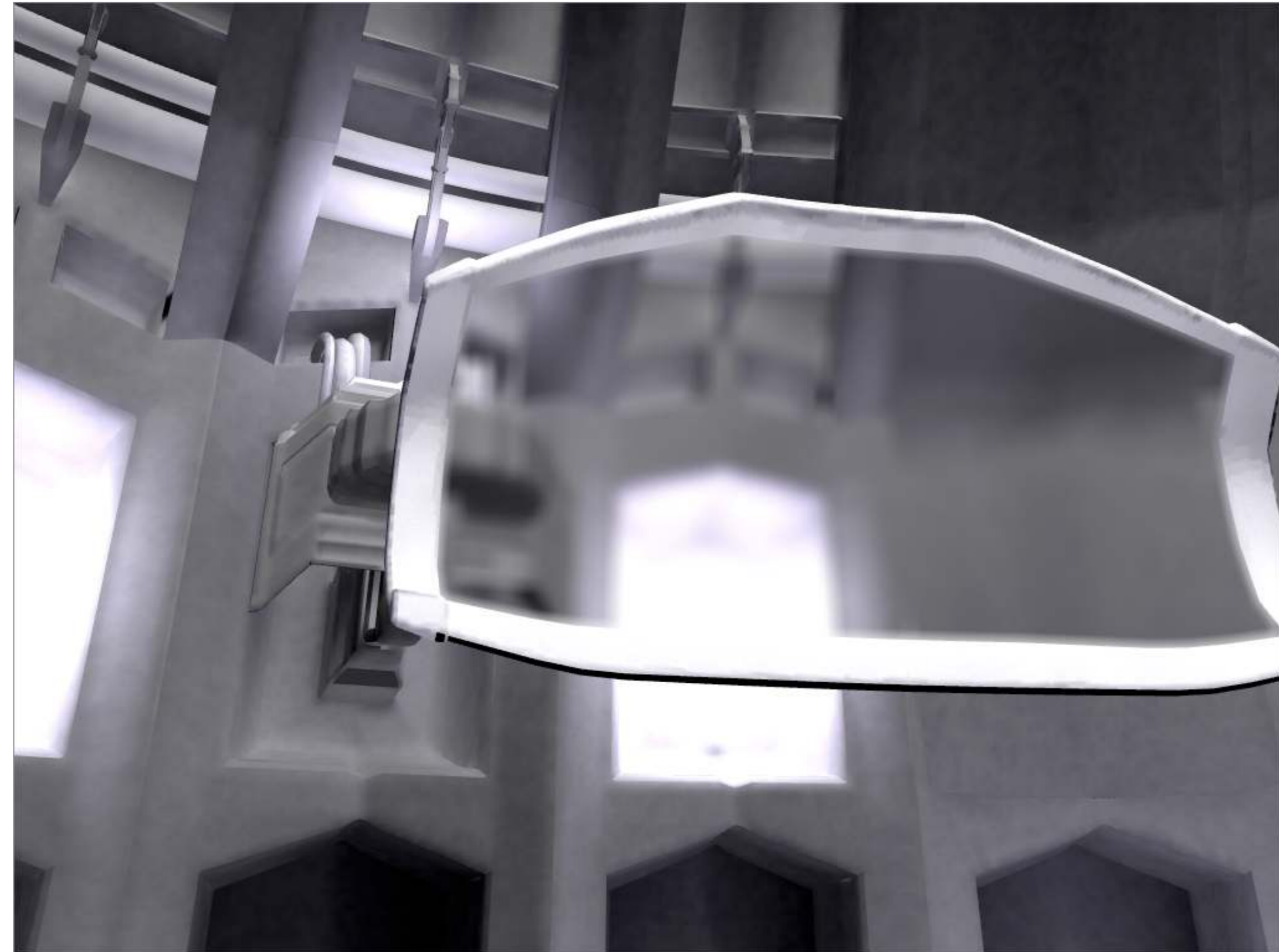
Efficient Computation of the Integral Image

- Assumption: image = N pixels
- Naïve approach: do a 1D prefix-sum per row (no transposition step)
 - Depth complexity: $O(\sqrt{N} \log N)$
 - Work complexity: $O(\sqrt{N} \cdot \sqrt{N}) = O(N)$
- Better solution:
 - Pack all rows into one linear array of size N
 - Do a 1D prefix-sum, but stop after the first $n = \sqrt{N}$ levels
 - Depth complexity = $O(\log N)$
 - Work complexity = $O(N)$
- Is a special case of **segmented prefix sum**



Applications of the Summed Area Table

- For filtering in general
- Simple example: **box filter** (blurring)
 - Slide box across image (convolution)
 - Compute average inside a box (= rectangle)
- Application: **translucent objects**, i.e., transparent & matte
 - E.g., "simulate" milky glass object in a game
 - 1. Render virtual scene without translucent objects
 - 2. Compute summed area table from frame buffer
 - 3. Render translucent object (using a fragment shader): replace pixel behind translucent object by average over original image within a (small) box



Rendering with **Depth-of-Field** (Tiefenunschärfe)

1. Render scene, save color buffer and z-buffer (e.g., in texture)

2. Compute summed area table over color buffer

3. For each pixel do *in parallel*:

1. Read depth of pixel from saved z-buffer

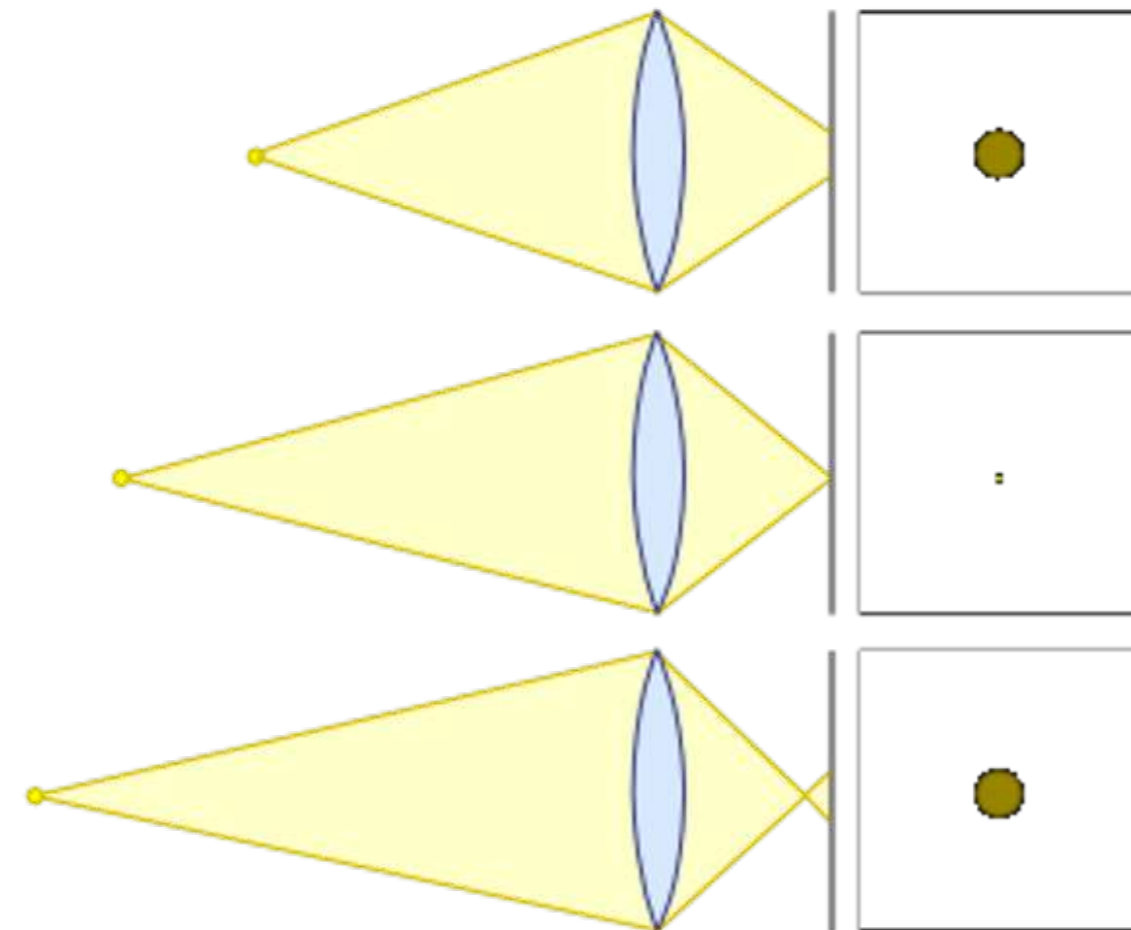
2. Compute **circle of confusion** (CoC)
(for details see "Advanced CG")

3. Determine size of box filter

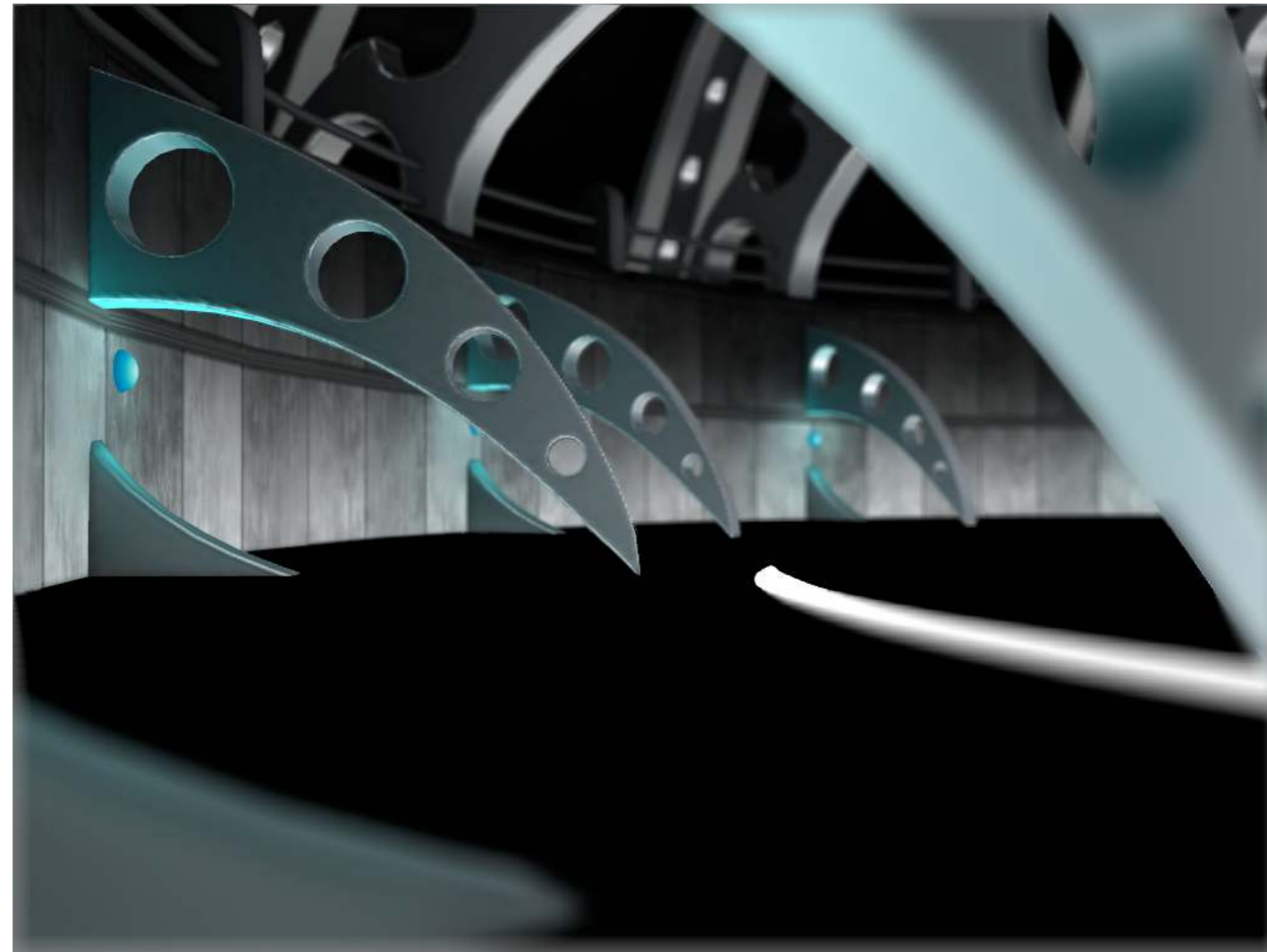
4. Compute average within box
over the orig. color buffer

5. Write in (new) color buffer

- Note: "For each pixel in parallel"
could be implemented in OpenGL
by rendering a screen-filling quad using special fragment shader



- Result:



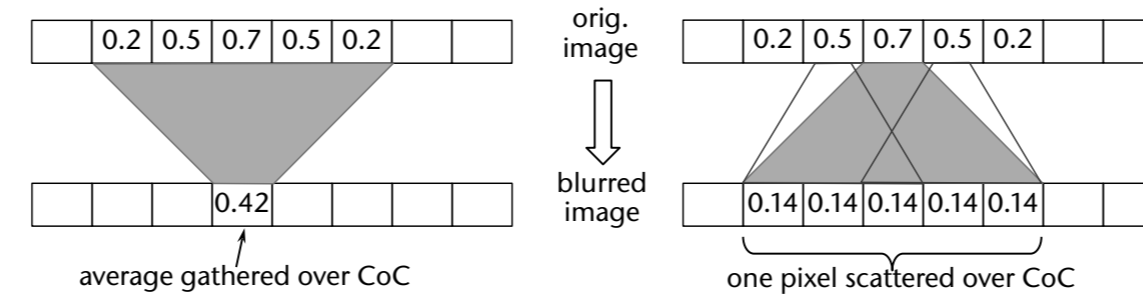
Artifacts of this Technique

- False sharp silhouettes: blurry objects (out of focus) have sharp silhouette, i.e., won't blur over sharp object (in focus)
- Color bleeding (a.k.a. pixel bleeding): areas in focus can incorrectly bleed into nearby areas out of focus
- Reason: the (indiscriminate) **gather operation**

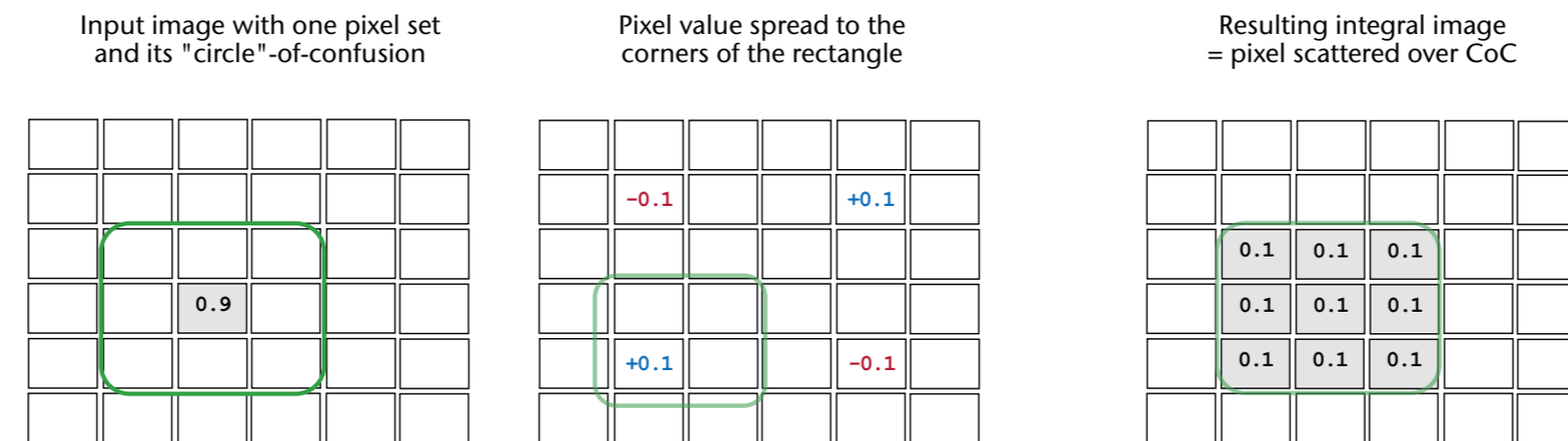


Depth-of-Field with Scattering

- Goal: turn gather operation into scatter operation



- Example: scatter one pixel using the 2D prefix-sum (integral image)



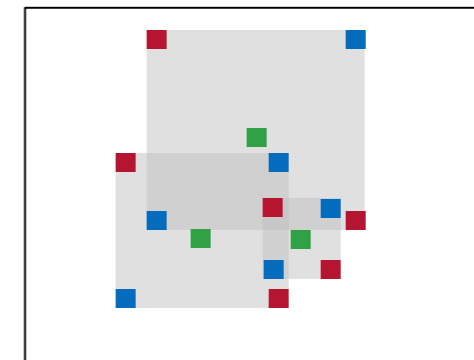
Algorithm

- Phase: for each pixel in original image do in parallel

- Spread $\frac{\text{pixel value}}{\text{area}(\text{CoC})}$ to CoC corners

- Use **atomic** accumulation operation !
- Do this for R, G, and B channels

- Phase: compute 2D prefix-sum over this "scatter image", result = final image with depth-of-field



- Research question: can you turn phase 1 into a *gather phase*?



Integral image, then gathering



Scattering , then integral image

[Kosloff, Tao, Barsky, 2009]

Application: Face Detection

- Goal: detect faces in images (not recognition)



digital camera



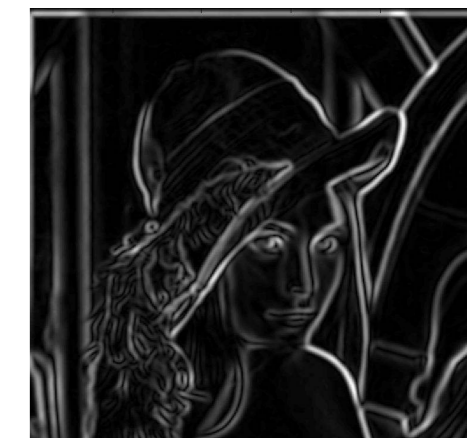
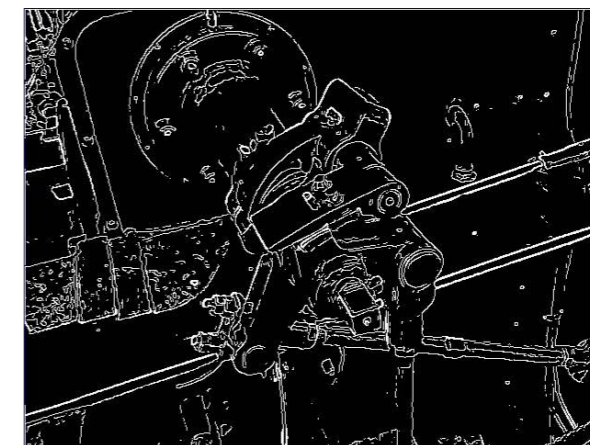
iPhoto



Includes a "false positive"
(or does it?)

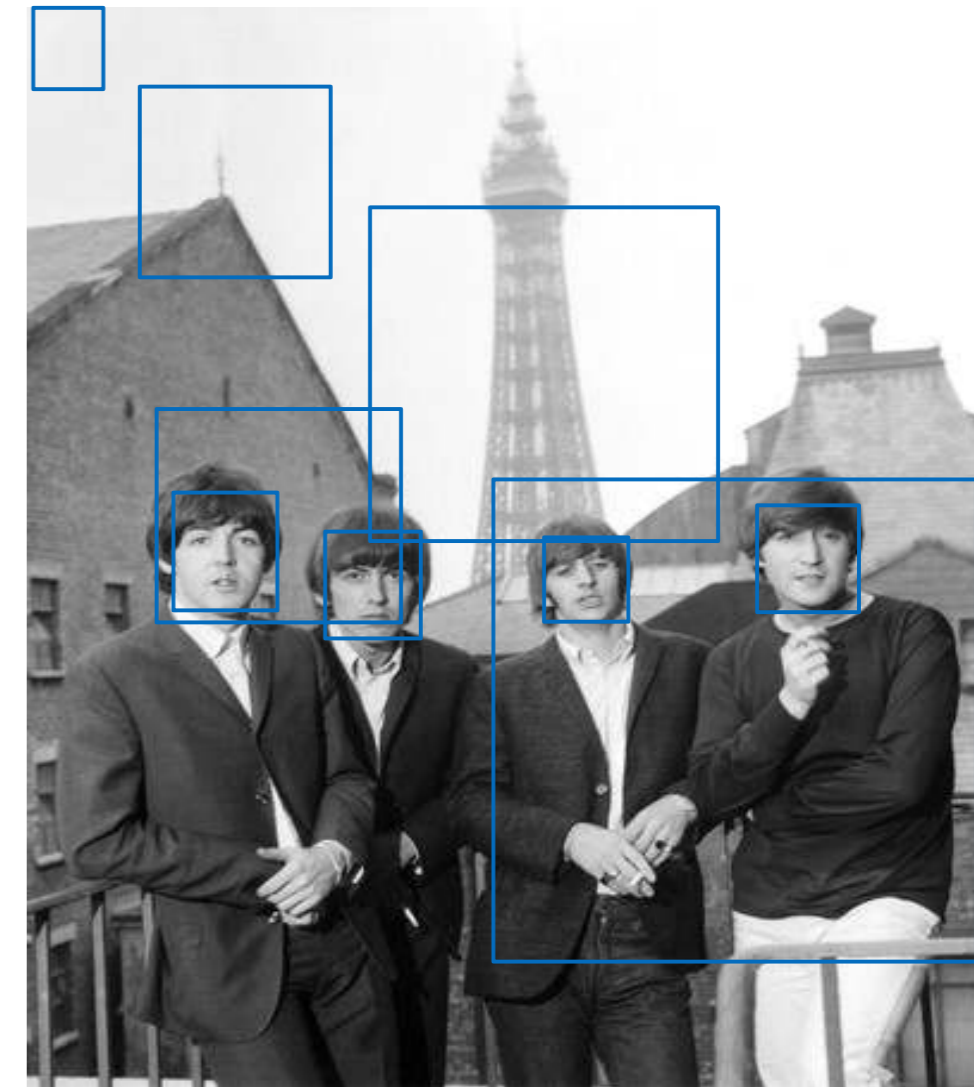
- Requirements (wishes):
 - Real-time or close (> 2 frames/sec)
 - Robust (high true-positive rate, low false-positive rate)
- Non-goal: face recognition
- In the following: no details, just overview!

- The term **feature** in computer vision:
 - Can be literally *any* piece of information/structure present in an image
 - Each kind of feature has a type, each feature has a value
- *Binary features* → present / not present
 - Examples:
 - Edges
 - Color of pixels is within specific range (e.g., skin)
- *Non-binary features* → probability of occurrence
 - Examples:
 - Gradient image
 - Sum of pixel values within a shape, e.g., rectangle



The Viola-Jones Face Detector

- The (simple) idea:
 - Move a sliding window across the image (all possible locations, all possible sizes)
 - Check, whether a face is in the window
 - We are interested only in windows that are filled by a face
- Observation:
 - Image contains 10's of faces
 - But $\sim 10^6$ candidate windows
- Consequence: to avoid having a false positive in every image, our false positive rate has to be $< 10^{-6}$



- Feature types used in the Viola-Jones face detector:

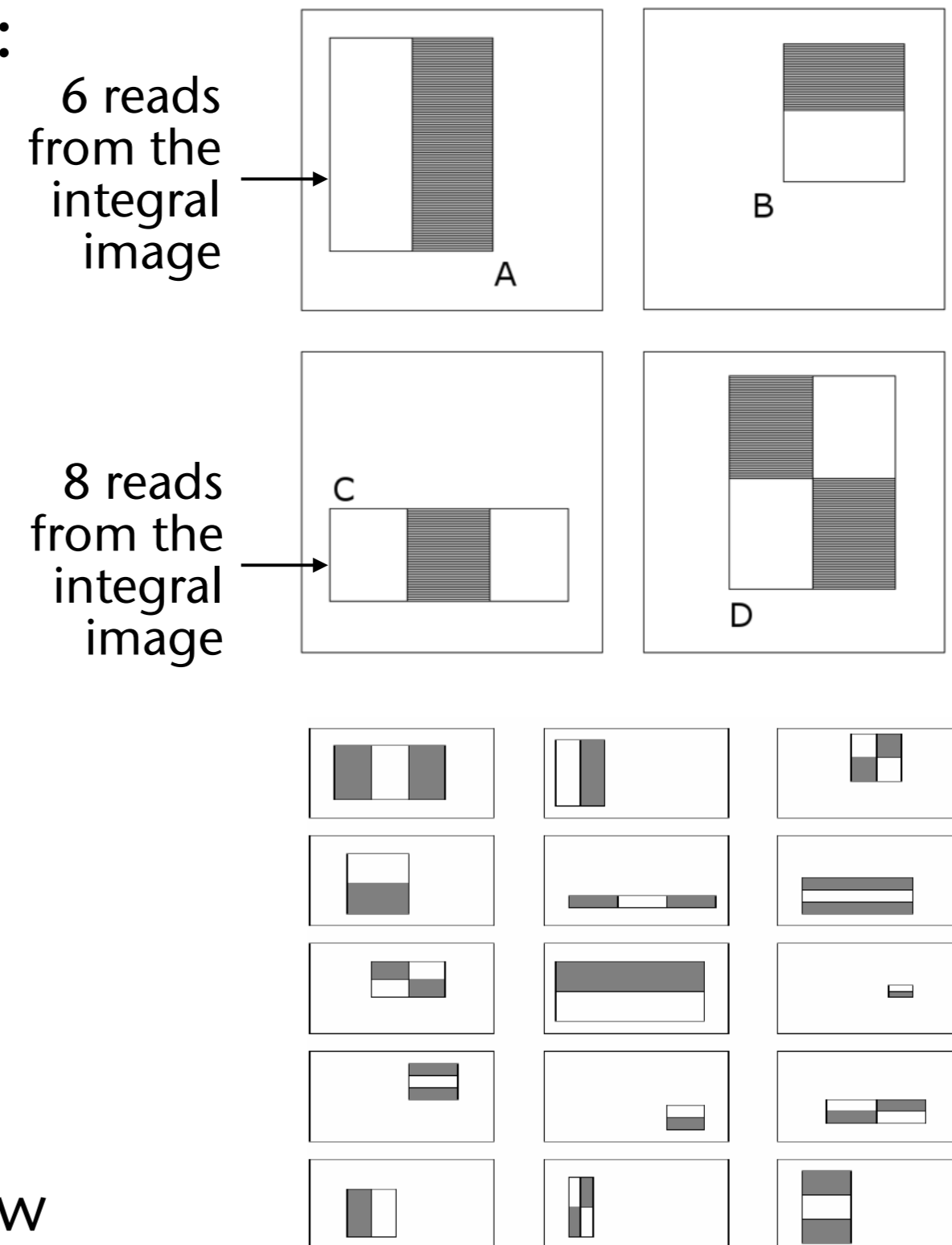
- 2, 3, or 4 rectangles placed next to each other
- Called **Haar features**

- Feature value $:= g_i =$
 $\text{pixel-sum}(\text{white rectangle(s)}) -$
 $\text{pixel-sum}(\text{black rectangle(s)})$

- Constant time per feature extraction

- In a 24x24 window (e.g., one of the sliding windows), there are
 $\sim 160,000$ possible features

- All variations of type, size, location within the window

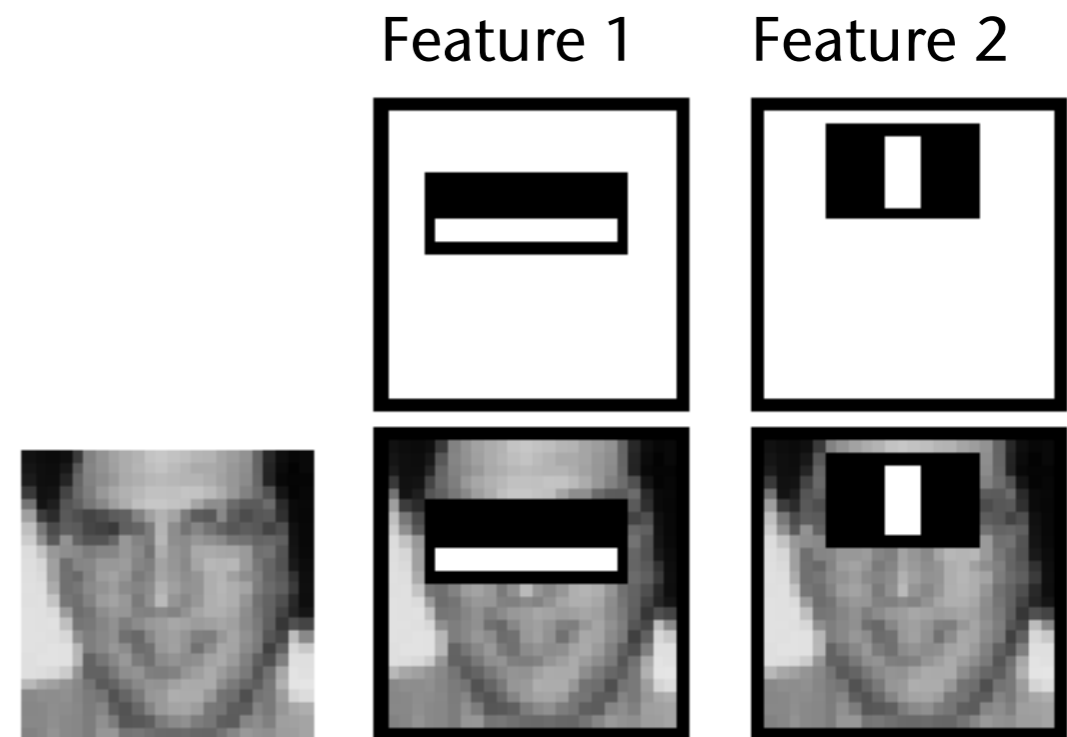


- Define a **weak classifier** for each feature:

$$f_i = \begin{cases} +1 & , g_i > \theta_i \\ -1 & , \text{else} \end{cases}$$

- For the two-rectangles feature, for instance, choose $\theta \approx \frac{1}{2} + \varepsilon$
- Called "weak", because such a classifier is only slightly better than a random "classifier"
- Idea: combine lots of weak classifiers to form one **strong classifier**

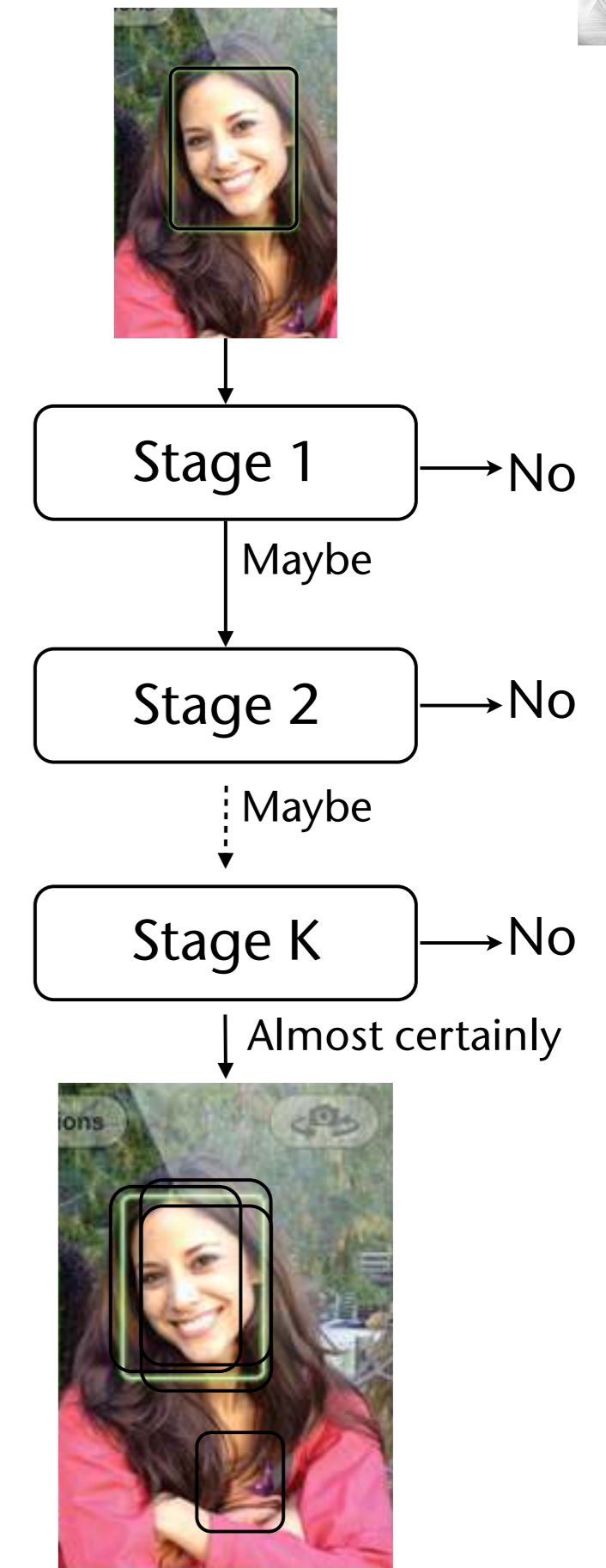
$$F(\text{window}) = \alpha_1 f_1 + \alpha_2 f_2 + \dots$$



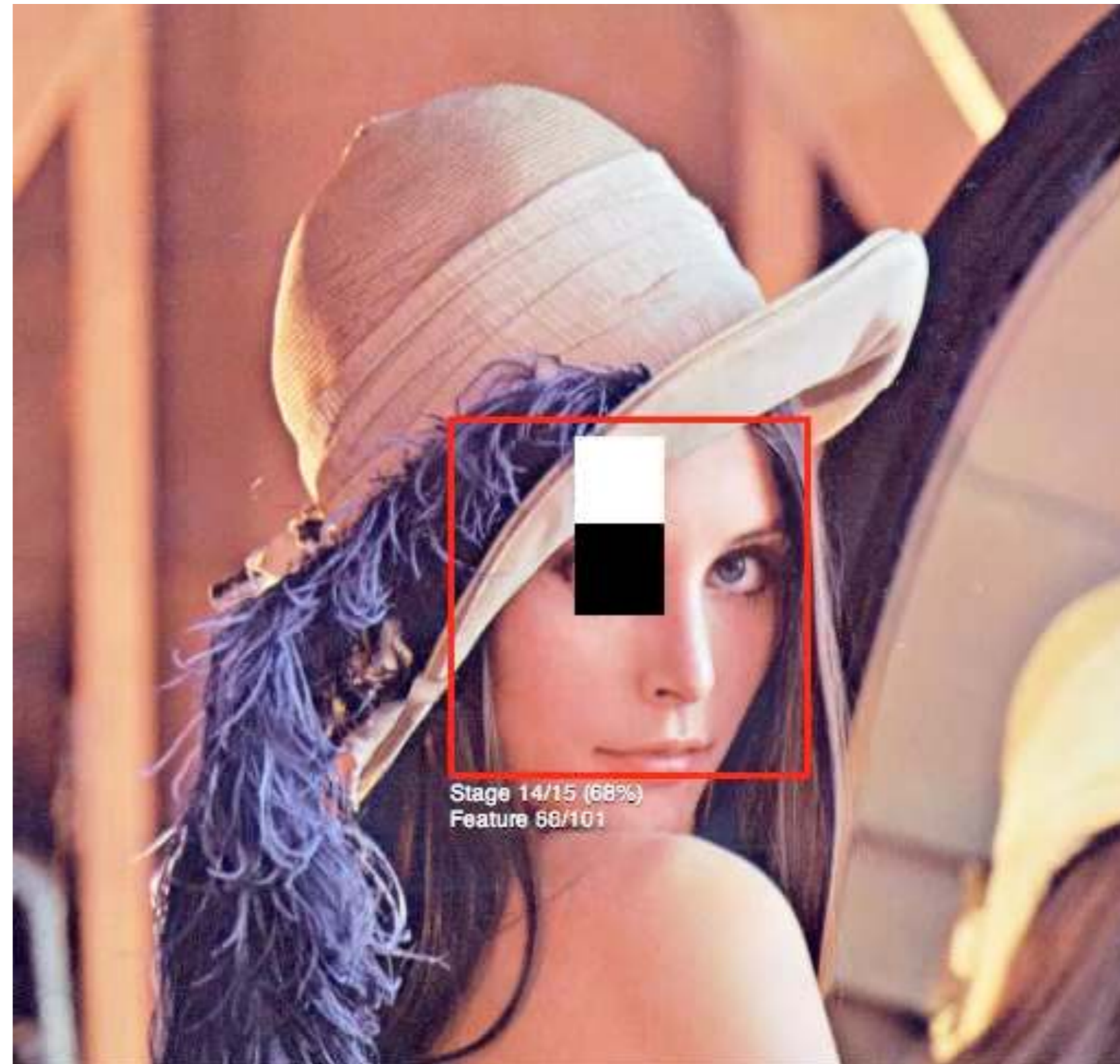
- Use learning algorithms to automatically find a set of **weak classifiers** and their optimal weights and thresholds, which together form a *strong classifier* (e.g., Random Forest)
 - More on that in AI & machine learning courses
- Training data:
 - 1000's of hand labeled faces containing many variations (illumination, pose, skin color, ...)
 - 10000 non-faces
- Weak classifiers with largest weights are meaningful and have high discriminative power (use first k of them):
 - Eyes region is darker than the upper-cheeks
 - Nose bridge region is brighter than the eyes



- Arrange in a **filter cascade**:
 - K classifiers with highest weights come first
 - If window fails one stage in the cascade → discard window
 - Advantage: "early exit" if "clearly" non-face
 - Typical detector has 38 stages in the cascade, ~6000 features/weak classifiers
- Final stage: only report face, if cascade finds several nearby face windows
 - Discard "lonesome" windows



Visualization of the Algorithm



Final remarks on Viola-Jones

- Pros:
 - Extremely fast feature computation
 - Scale and location invariant detector
 - Instead of scaling the image itself (e.g. pyramid-filters), we scale the features
 - Works also for some other types of objects
- Cons:
 - Doesn't work very well for 45° views on faces
 - Not **rotation invariant**