Lecture 8:

Data-Parallel Thinking

Parallel Computing
Stanford CS149, Fall 2020

Today's theme

- Many of you are now likely accustomed to thinking about parallel programming in terms of "what workers do"

- Today I would like you to think about describing algorithms in terms of operations on sequences of data
  - map
  - filter
  - fold / reduce
  - scan / segmented scan
  - sort
  - groupBy
  - join
  - partition / flatten

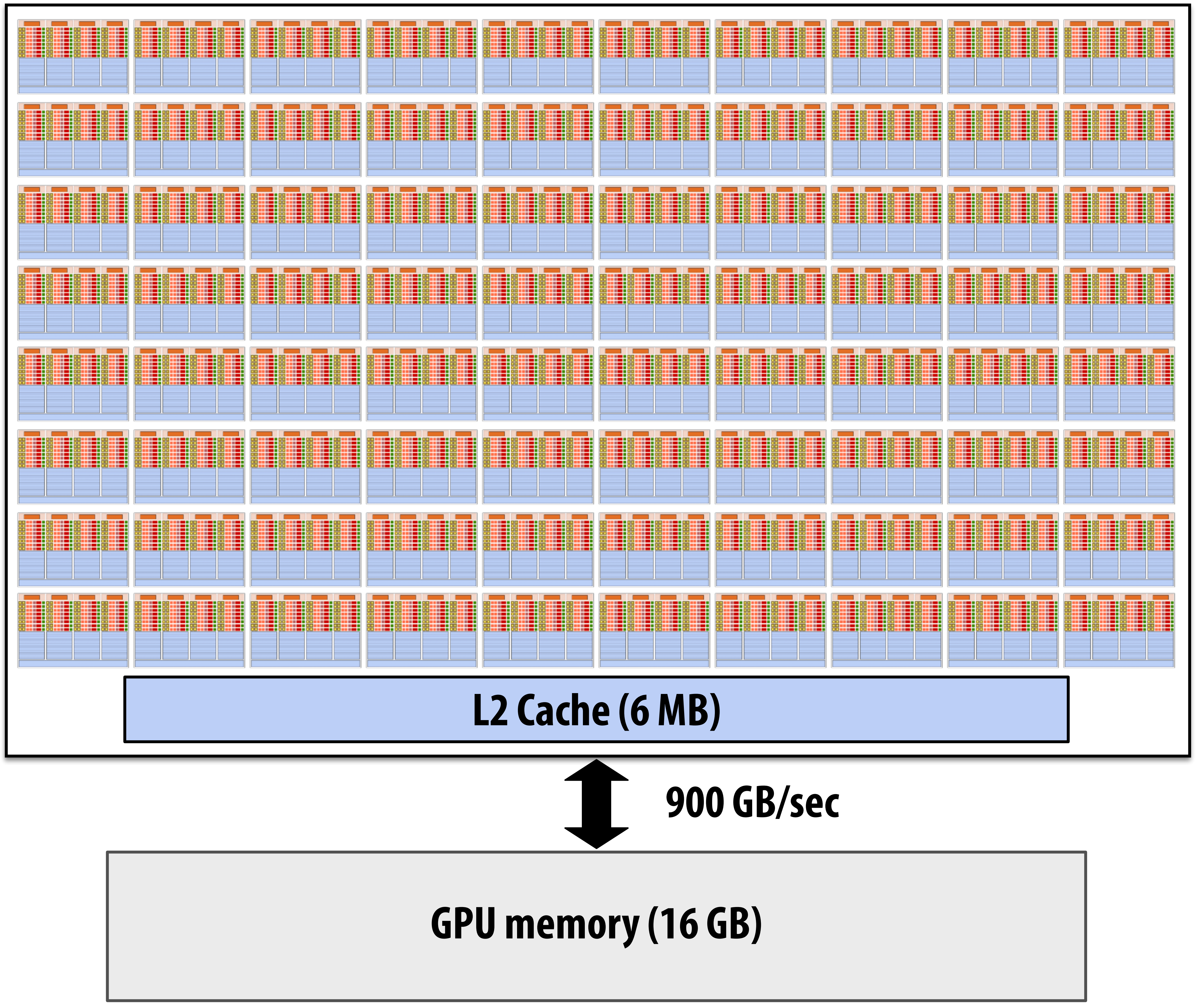
- Main idea: high-performance implementations of these operations exist. So programs written in terms of these primitives can often run efficiently on a parallel machine

Motivation

- Why must an application expose large amounts of parallelism?

- Utilize large numbers of cores
  - High core count machines
  - Many machines (e.g., cluster of machines in the cloud)
  - SIMD processing + multi-threaded cores require even more parallelism
  - GPU architectures require very large amounts of parallelism

Recall: geometry of the V100 GPU



1.245 GHz clock

80 SM cores per chip

80 x 4 x 16 = 5,120 fp32 mul-add ALUs
= 12.7 TFLOPs *

Up to 80 x 64 = 5120 interleaved warps per chip (163,840 CUDA threads/chip)

This chip can concurrently execute up to 163,860 CUDA threads!
(programs that do not expose significant amounts of parallelism, and don't have high arithmetic intensity, will not run efficiently on GPUs!)

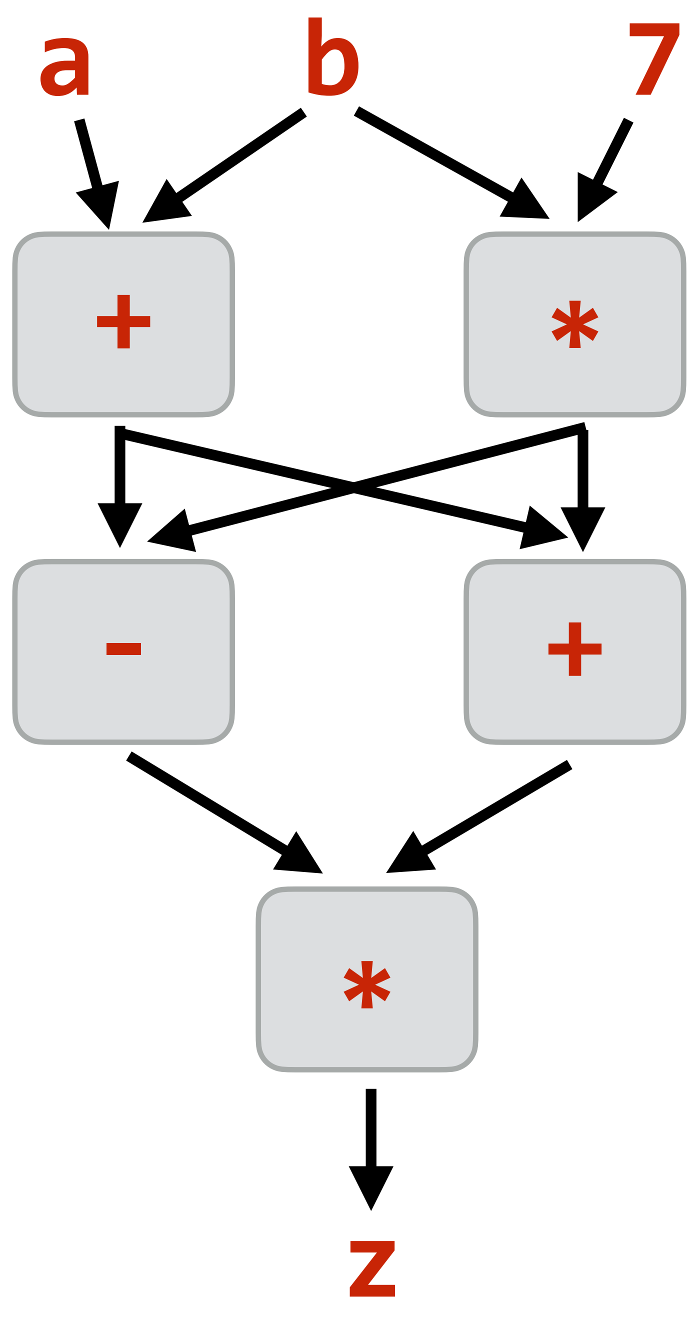
* mul-add counted as 2 flops:

Understanding dependencies is key

- **Key part of parallel programming is understanding when dependencies exist between operation**

- **Lack of dependencies implies potential for parallel execution**

```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



Data-parallel model

- **Organize computation as operations on sequences of elements**
  - **e.g., perform same function on all elements of a sequence**

- **Historically: same operation on each element of an array**
  - **Matched capabilities SIMD supercomputers of 80's**
  - **Connection Machine (CM-1, CM-2): thousands of processors, one instruction decode unit**
  - **Early Cray supercomputers were vector processors**
    - `add(A, B, n)` ← **this was one instruction on vectors A, B of length n**

- **A well-known modern example: NumPy: C = A + B (A, B, and C are vectors of same length)**

Key data type: sequences

- Ordered collection of elements
- For example, in a C++ like language: Sequence<T>
- e.g., Scala lists: List[T]
- In a functional language (like Haskell): seq T

- Important: unlike arrays, programs can only access elements of a sequence through specific operations

Map

- Higher order function (function that takes a function as an argument)
- Applies side-effect free unary function `f :: a -> b` to all elements of input sequence, to produce output sequence of the same length
- In a functional language (e.g., Haskell)
  - `map :: (a -> b) -> seq a -> seq b`
- In C++:

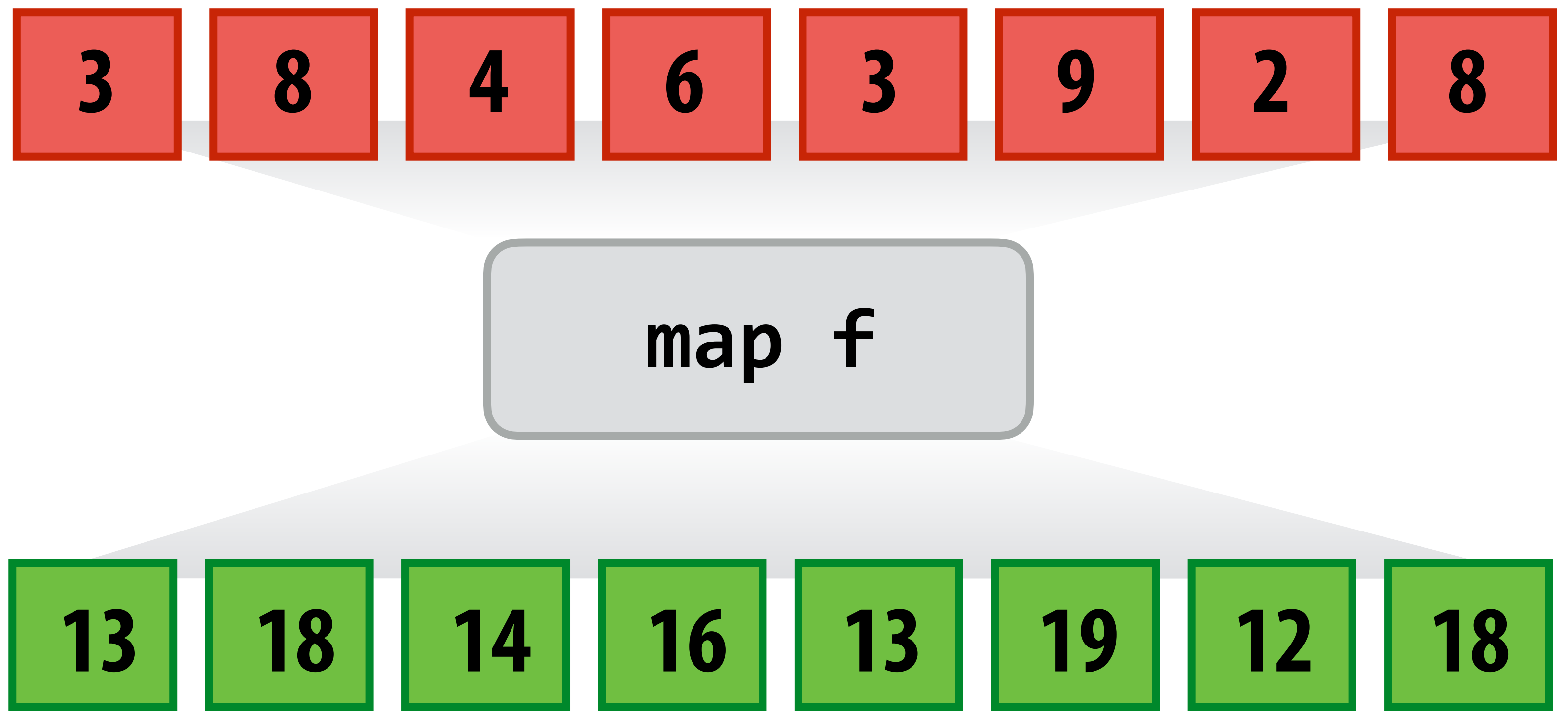
```
template<class InputIt, class OutputIt,
         class UnaryOperation>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first,
                   UnaryOperation unary_op);
```

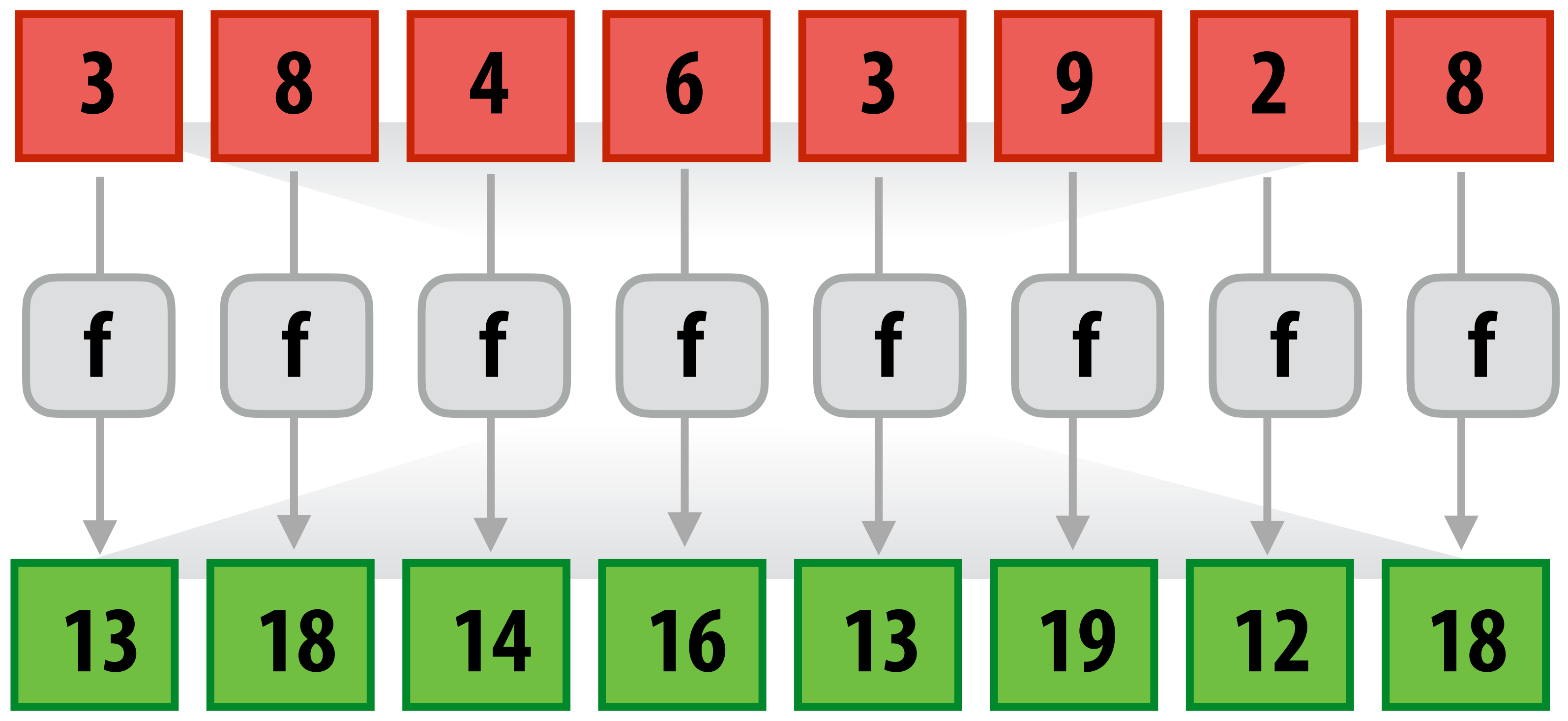
C++

```
int f(int x) { return x + 10; }

int a[] = {3, 8, 4, 6, 3, 9, 2, 8};
int b[8];
std::transform(a, a+8, b, f);
```

Haskell

```
a = [3, 8, 4, 6, 3, 9, 2, 8]
f x = x + 10
b = map f a
```





Parallelizing map

- Since `f :: a -> b` is a function (side-effect free), then applying `f` to all elements of the sequence can be performed in *any order* without changing the output of the program

- The implementation of map has flexibility to reorder/parallelize processing of elements of sequence however it sees fit

```
map f s =
    partition sequence s into P smaller sequences
    for each subsequence s_i (in parallel)
        out_i = map f s_i
    out = concatenate out_i’s
```

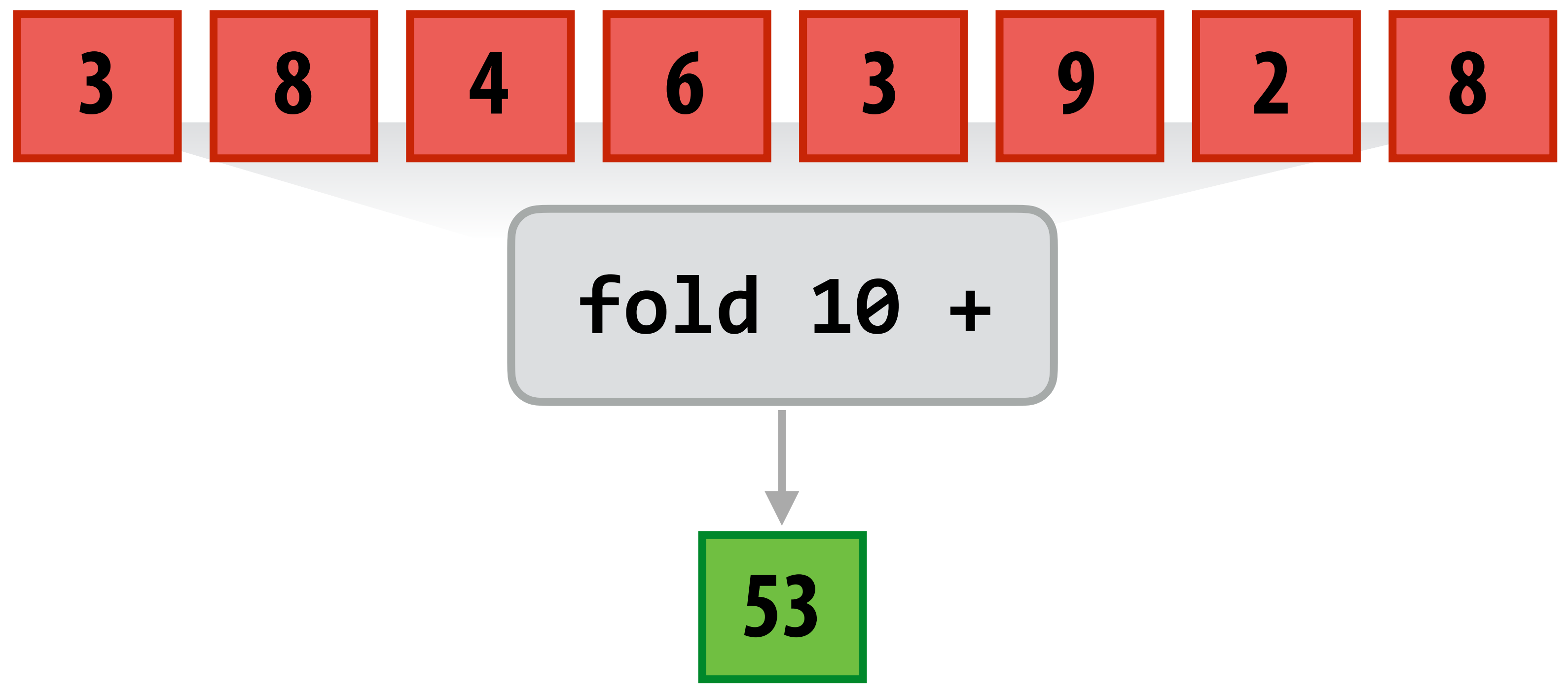
Fold (fold left)

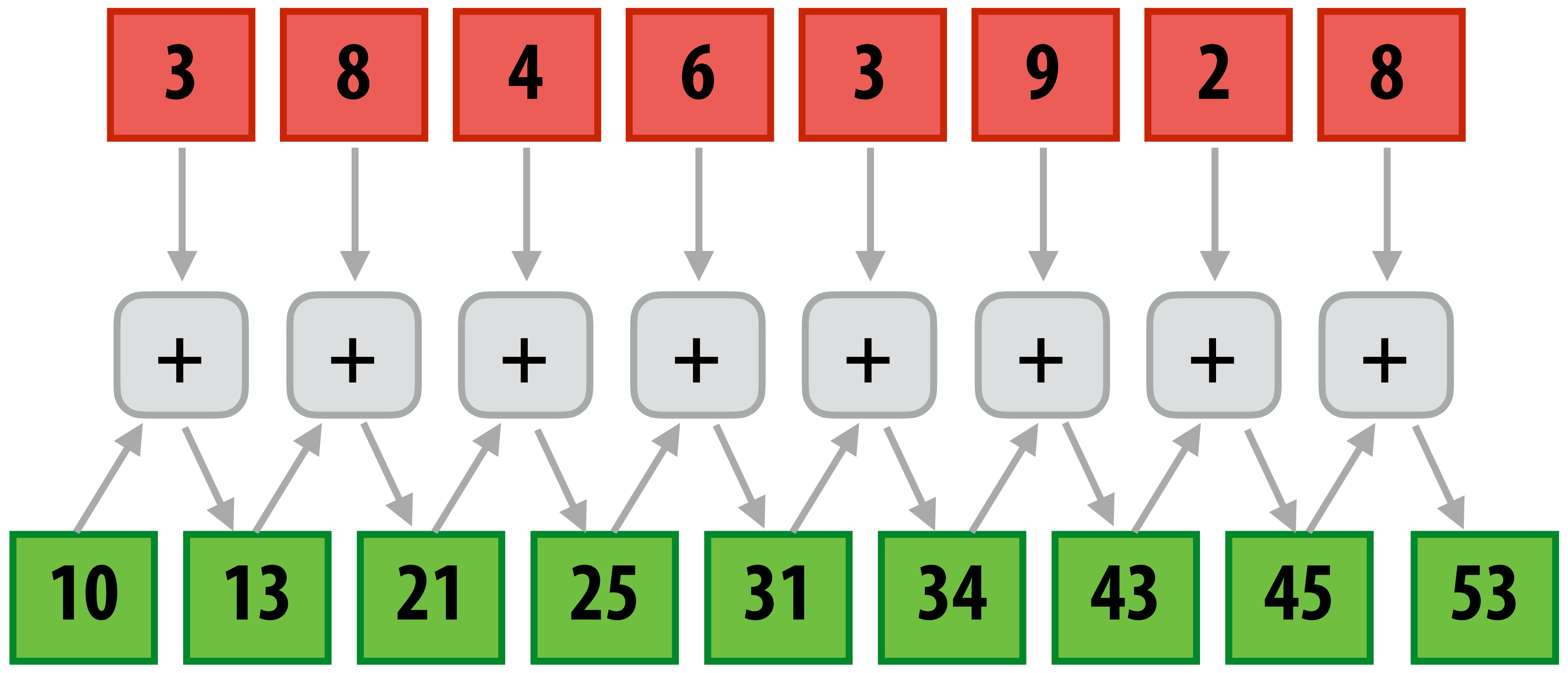
- Apply binary operation f to each element and an accumulated value
  - Seeded by initial value of type b

```
f :: (b,a) -> b
fold :: b -> ((b,a) -> b) -> seq a -> b
```

E.g., in Scala:

```
def foldLeft[A, B](init: B, f: (B, A) => B, l: List[A]): B
```

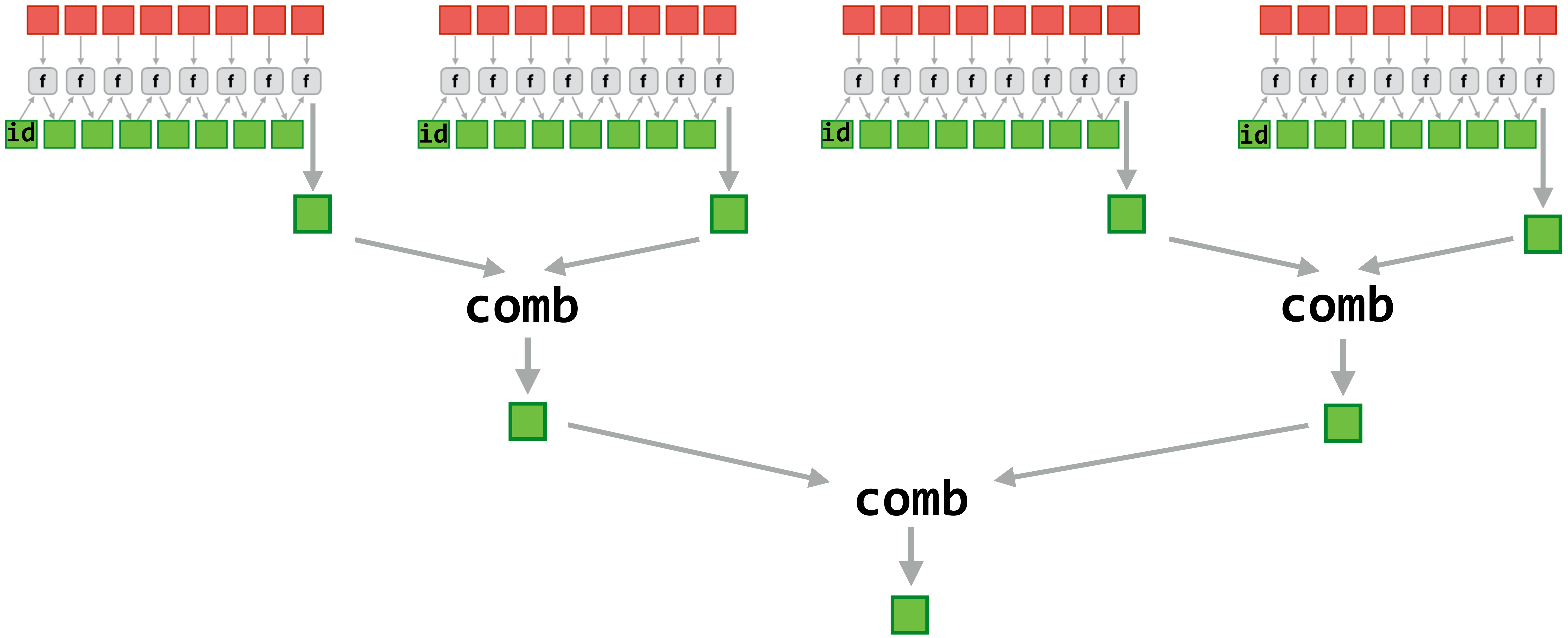




Parallel fold

- Apply f to each element and an accumulated value
  - In addition to binary f, needs binary "combiner" function *
  - Seeded by initial value of type b (must be identity for *f* and *comb*)

```
f :: (b,a) -> b
comb :: (b,b) -> b
fold_par :: b -> ((b,a) -> b) -> ((b,b)->b) ->seq a -> b
```

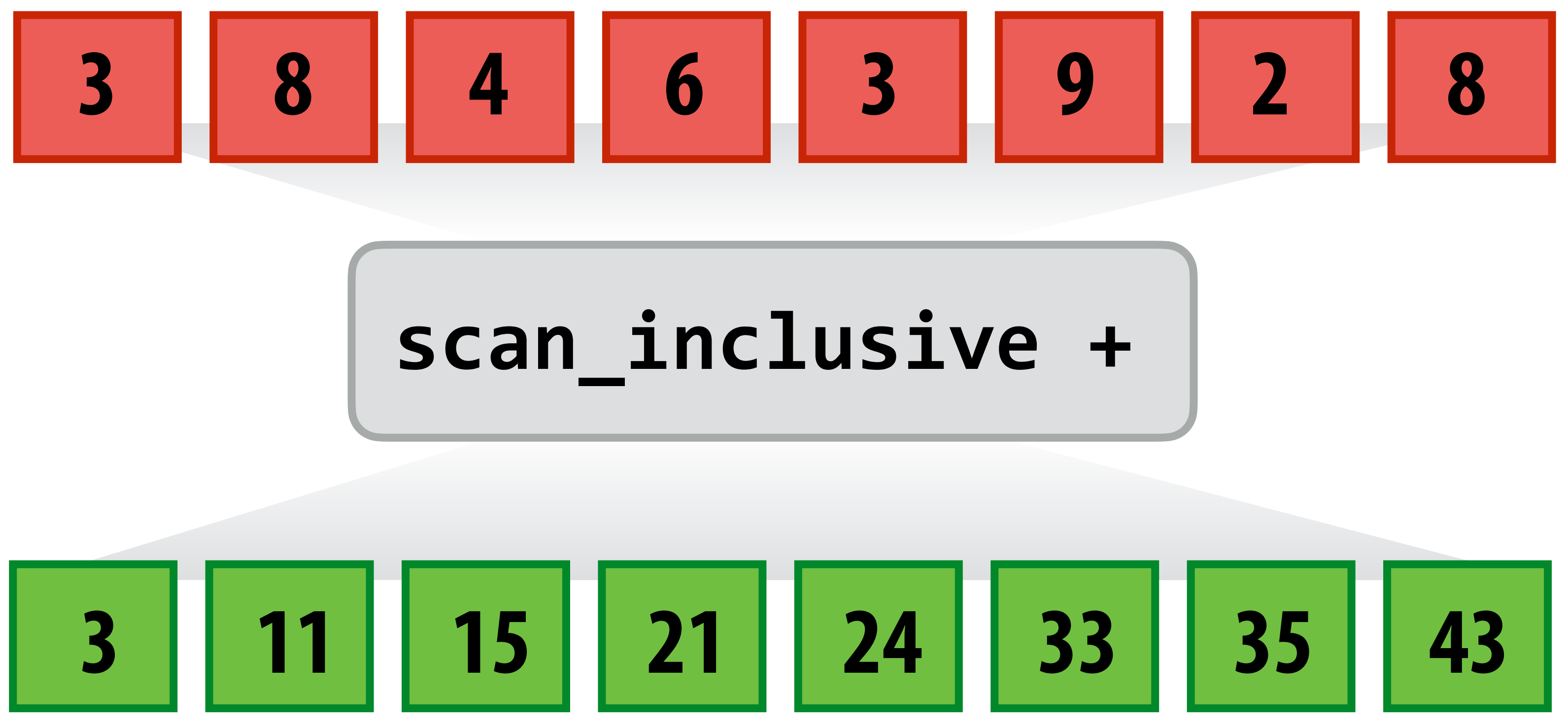


* No need for comb if `f::(b,b)->b` is an associative binary operator

Scan

`f :: (a,a) -> a` **(associative binary op)**

`scan :: a -> ((a,a) -> a) -> seq a -> seq a`



```
float op(float a, float b) { … }
scan_inclusive(float* in, float* out, int N) {
 out[0] = in[0];
 for (i=1; i<N; i++)
   out[i] = op(out[i-1], in[i]);
}
```

Alternative: "scan exclusive": the value of out[i] is the scan result for all elements up to, but excluding, in[i].

Parallel Scan

Data-parallel scan

let `A = [`a_0`,`a_1`,`a_2`,`a_3`,...,`a_{n-1}`]`

let $\oplus$ be an associative binary operator with identity element `I`

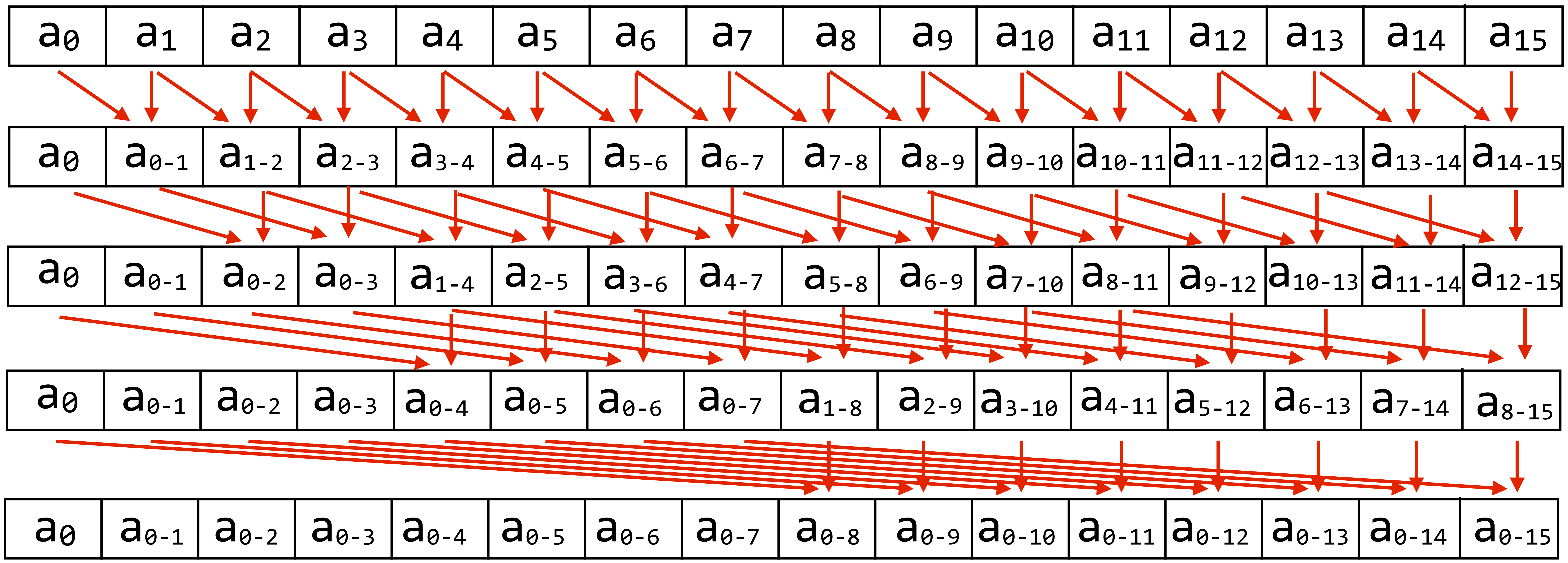
```
scan_inclusive(⊕, A) = [a₀, a₀⊕a₁, a₀⊕a₁⊕a₂, ...
scan_exclusive(⊕, A) = [I, a₀, a₀⊕a₁, ...
```

If operator is +, then `scan_inclusive(+,A)` **is called "a prefix sum"**

```
prefix_sum(A) = [a₀, a₀+a₁, a₀+a₁+a₂, ...
```

Data-parallel inclusive scan

(Subtract original vector to get exclusive scan result: not shown)



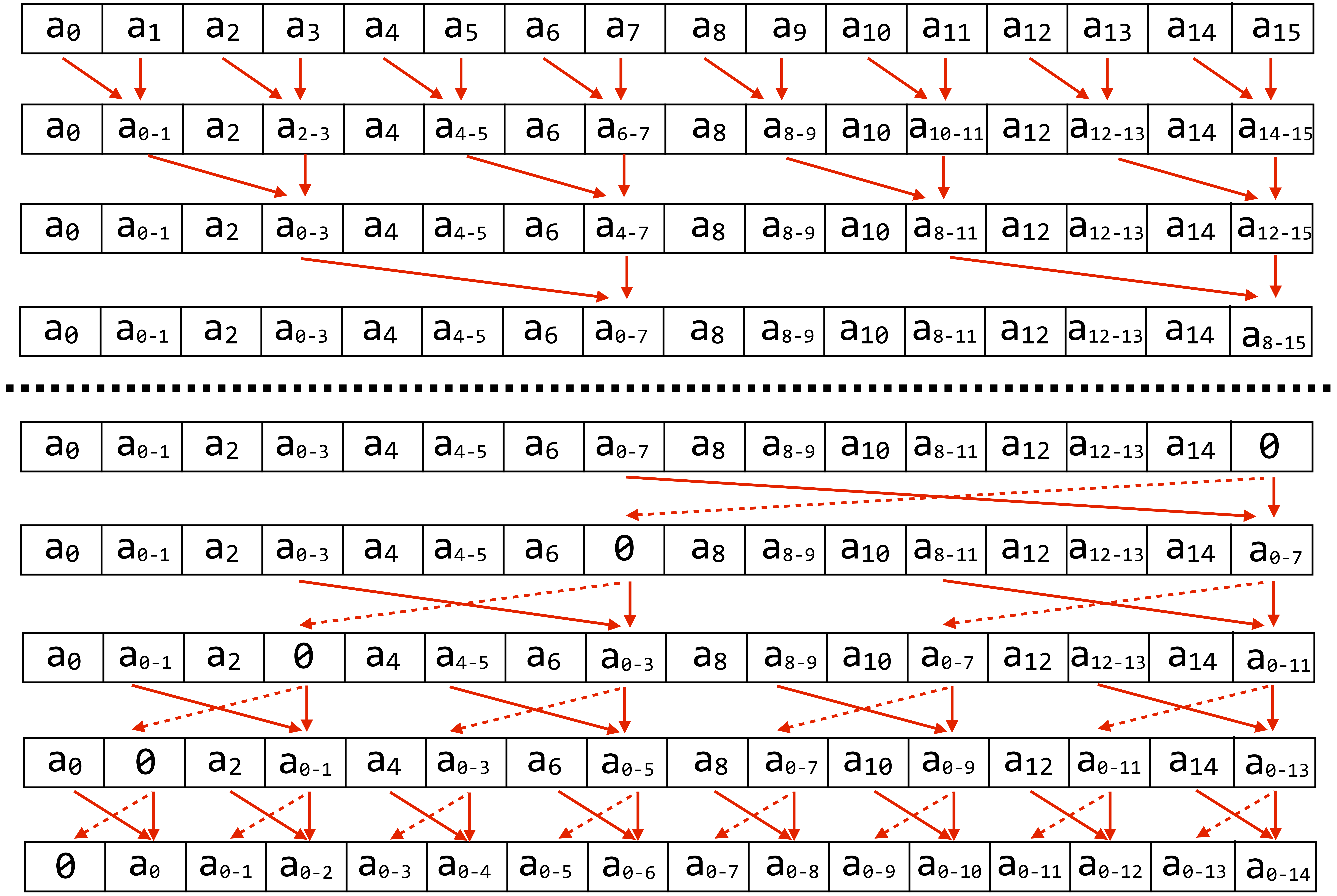
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8 a_9 a_{10} a_{11} a_{12} a_{13} a_{14} a_{15}
a_0 a_{0-1} a_{1-2} a_{2-3} a_{3-4} a_{4-5} a_{5-6} a_{6-7} a_{7-8} a_{8-9} a_{9-10} a_{10-11} a_{11-12} a_{12-13} a_{13-14} a_{14-15}
a_0 a_{0-1} a_{0-2} a_{0-3} a_{1-4} a_{2-5} a_{3-6} a_{4-7} a_{5-8} a_{6-9} a_{7-10} a_{8-11} a_{9-12} a_{10-13} a_{11-14} a_{12-15}
a_0 a_{0-1} a_{0-2} a_{0-3} a_{0-4} a_{0-5} a_{0-6} a_{0-7} a_{1-8} a_{2-9} a_{3-10} a_{4-11} a_{5-12} a_{6-13} a_{7-14} a_{8-15}
a_0 a_{0-1} a_{0-2} a_{0-3} a_{0-4} a_{0-5} a_{0-6} a_{0-7} a_{0-8} a_{0-9} a_{0-10} a_{0-11} a_{0-12} a_{0-13} a_{0-14} a_{0-15}

Work: O(N lg N)

Inefficient compared to sequential algorithm!

Span: O(lg N)

Work-efficient parallel exclusive scan (O(N) work)


a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15
a0 a0-1 a2 a2-3 a4 a4-5 a6 a6-7 a8 a8-9 a10 a10-11 a12 a12-13 a14 a14-15
a0 a0-1 a2 a0-3 a4 a4-5 a6 a4-7 a8 a8-9 a10 a8-11 a12 a12-13 a14 a12-15
a0 a0-1 a2 a0-3 a4 a4-5 a6 a0-7 a8 a8-9 a10 a8-11 a12 a12-13 a14 a8-15
a0 a0-1 a2 a0-3 a4 a4-5 a6 a0-7 a8 a8-9 a10 a8-11 a12 a12-13 a14 0
a0 a0-1 a2 a0-3 a4 a4-5 a6 0 a8 a8-9 a10 a8-11 a12 a12-13 a14 a0-7
a0 a0-1 a2 0 a4 a4-5 a6 a0-3 a8 a8-9 a10 a0-7 a12 a12-13 a14 a0-11
a0 0 a2 a0-1 a4 a0-3 a6 a0-5 a8 a0-7 a10 a0-9 a12 a0-11 a14 a0-13
0 a0 a0-1 a0-2 a0-3 a0-4 a0-5 a0-6 a0-7 a0-8 a0-9 a0-10 a0-11 a0-12 a0-13 a0-14

Work efficient exclusive scan algorithm

(with $\oplus$ = "+")

Up-sweep:

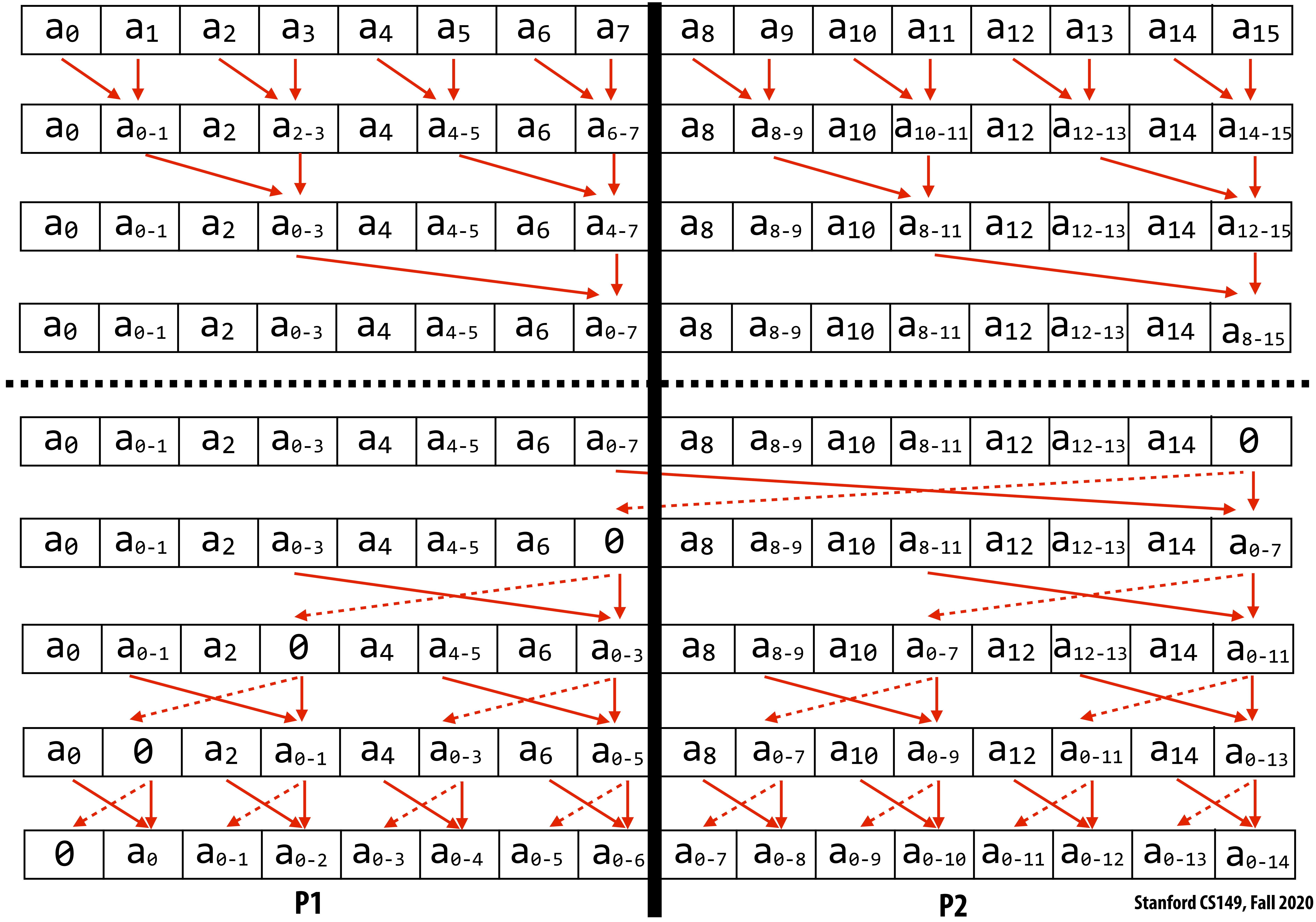
```
for d=0 to ($\log_2 n$ - 1) do
   forall k=0 to n-1 by $2^{d+1}$ do
     a[k + $2^{d+1}$ - 1] = a[k + $2^d$ - 1] + a[k + $2^{d+1}$ - 1]
```

Down-sweep:

```
x[n-1] = 0
for d=($\log_2 n$ - 1) down to 0 do
   forall k=0 to n-1 by $2^{d+1}$ do
     tmp = a[k + $2^d$ - 1]
     a[k + $2^d$ - 1] = a[k + $2^{d+1}$ - 1]
     a[k + $2^{d+1}$ - 1] = tmp + a[k + $2^{d+1}$ - 1]
```

Work: O(N)   (but what is the constant?)
Span: O(lg N)   (but what is the constant?)
Locality: ??

Now consider scan implementation on just two cores



a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15
a0 a0-1 a2 a2-3 a4 a4-5 a6 a6-7 a8 a8-9 a10 a10-11 a12 a12-13 a14 a14-15
a0 a0-1 a2 a0-3 a4 a4-5 a6 a4-7 a8 a8-9 a10 a8-11 a12 a12-13 a14 a12-15
a0 a0-1 a2 a0-3 a4 a4-5 a6 a0-7 a8 a8-9 a10 a8-11 a12 a12-13 a14 a8-15
a0 a0-1 a2 a0-3 a4 a4-5 a6 a0-7 a8 a8-9 a10 a8-11 a12 a12-13 a14 0
a0 a0-1 a2 a0-3 a4 a4-5 a6 0 a8 a8-9 a10 a8-11 a12 a12-13 a14 a0-7
a0 a0-1 a2 0 a4 a4-5 a6 a0-3 a8 a8-9 a10 a0-7 a12 a12-13 a14 a0-11
a0 0 a2 a0-1 a4 a0-3 a6 a0-5 a8 a0-7 a10 a0-9 a12 a0-11 a14 a0-13
0 a0 a0-1 a0-2 a0-3 a0-4 a0-5 a0-6 a0-7 a0-8 a0-9 a0-10 a0-11 a0-12 a0-13 a0-14
P1
P2

Scan: two processor (shared memory) implementation

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}	a_{15}

P1	P2
Sequential scan on elements [0-7]	Sequential scan on elements [8-15]
Let base = a_{0-7}	
Add base to elements a_8 thru a_{8-11}	Add base to elements a_{8-12} thru a_{8-15}

Work: O(N) (but constant is now only 1.5)

Data-access:

- **Very high spatial locality (contiguous memory access)**
- **P1's access to a_8 through a_{8-11} may be more costly on large core count "NUMA" system, but on small-scale multi-core system the access cost is likely the same as from P2**

Exclusive scan: SIMD implementation (in CUDA)

Example: perform exclusive scan on 32-element array: SPMD program, assume 32-wide SIMD execution

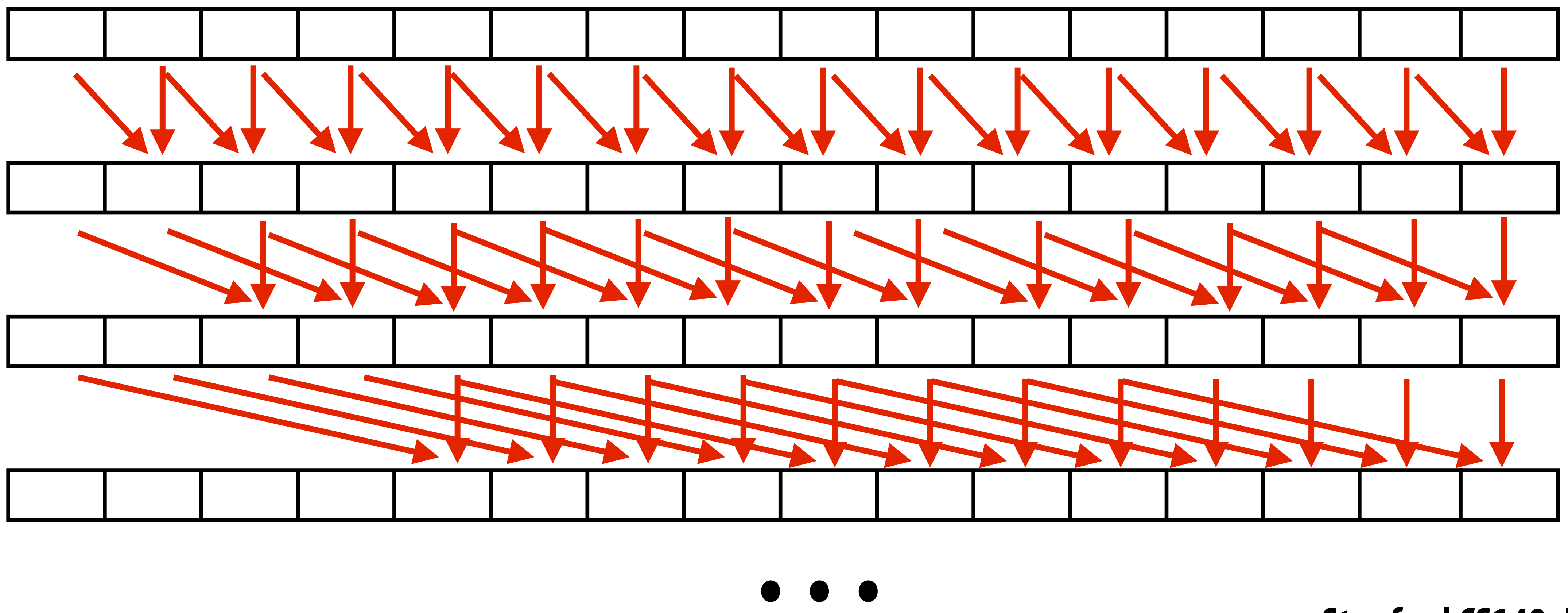
When `scan_warp` is run by a group of 32 CUDA threads, each thread returns the exclusive scan result for element `idx`

(also: upon completion `ptr[]` stores inclusive scan result)

CUDA thread index of caller

```
__device__ int scan_warp(volatile int *ptr, const unsigned int idx)
{
   const unsigned int lane = idx & 31; // index of thread in warp (0..31)

   if (lane >= 1)  ptr[idx] = ptr[idx - 1]  +  ptr[idx];
   if (lane >= 2)  ptr[idx] = ptr[idx - 2]  +  ptr[idx];
   if (lane >= 4)  ptr[idx] = ptr[idx - 4]  +  ptr[idx];
   if (lane >= 8)  ptr[idx] = ptr[idx - 8]  +  ptr[idx];
   if (lane >= 16) ptr[idx] = ptr[idx - 16] +  ptr[idx);

   return (lane > 0) ? ptr[idx-1] : 0;
}
```

Work: ??



Exclusive scan: SIMD implementation (in CUDA)

CUDA thread
index of caller

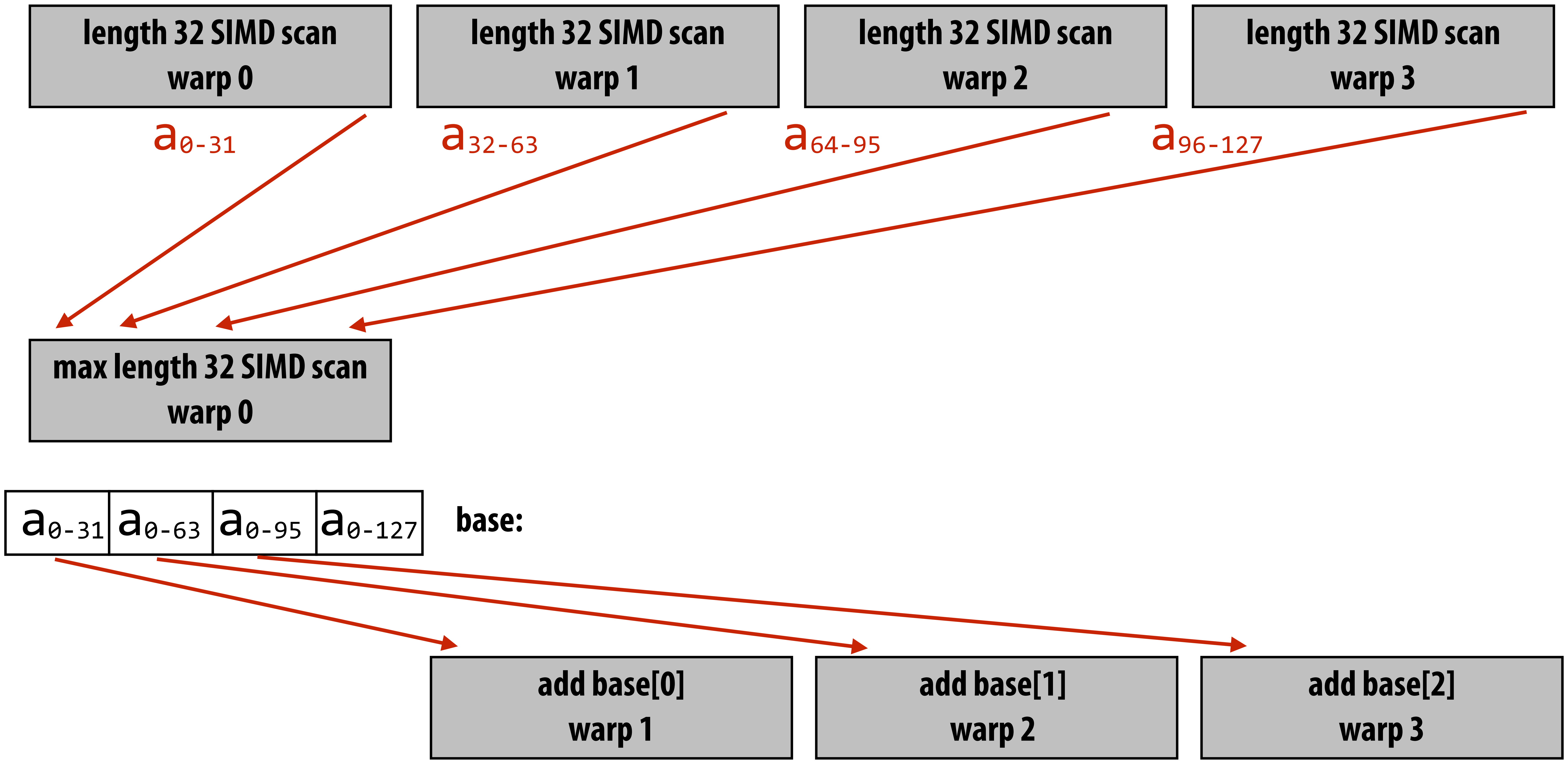
```
__device__ int scan_warp(volatile int *ptr, const unsigned int idx)
{
   const unsigned int lane = idx & 31; // index of thread in warp (0..31)

   if (lane >= 1)  ptr[idx] = ptr[idx - 1]  +  ptr[idx];
   if (lane >= 2)  ptr[idx] = ptr[idx - 2]  +  ptr[idx];
   if (lane >= 4)  ptr[idx] = ptr[idx - 4]  +  ptr[idx];
   if (lane >= 8)  ptr[idx] = ptr[idx - 8]  +  ptr[idx];
   if (lane >= 16) ptr[idx] = ptr[idx - 16] +  ptr[idx];

   return (lane > 0) ? ptr[idx-1] : 0;
}
```

Work: N lg(N)

Work-efficient formulation of scan is not beneficial in this context because it results in low SIMD utilization. Work efficient algorithm would require more than 2x the number of instructions as the implementation above!

Building scan on larger array

Example: 128-element scan using four-warp thread block



length 32 SIMD scan warp 0
length 32 SIMD scan warp 1
length 32 SIMD scan warp 2
length 32 SIMD scan warp 3
a_{0-31}
a_{32-63}
a_{64-95}
a_{96-127}
max length 32 SIMD scan warp 0
a_{0-31} a_{0-63} a_{0-95} a_{0-127}
base:
add base[0] warp 1
add base[1] warp 2
add base[2] warp 3

Multi-threaded, SIMD CUDA implementation

Example: cooperating threads in a CUDA thread block perform scan

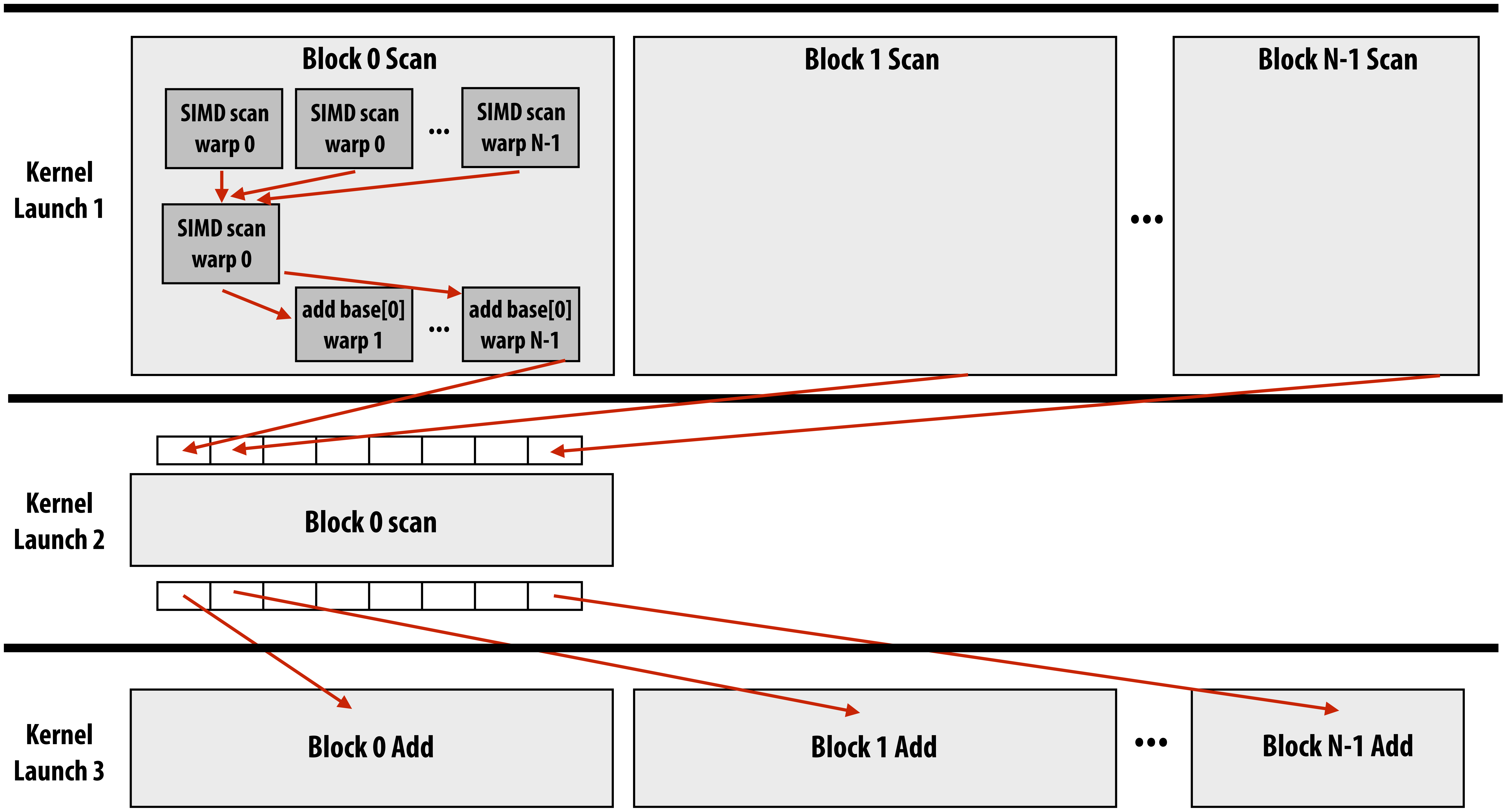
We provided similar code in assignment 3.
Code assumes length of array given by `ptr` is same as number of threads per block.

CUDA thread index of caller

```
__device__ void scan_block(volatile int *ptr, const unsigned int idx)
{
   const unsigned int lane = idx & 31;          // index of thread in warp (0..31)
   const unsigned int warp_id = idx >> 5;       // warp index in block

   int val = scan_warp(ptr, idx);               // Step 1. per-warp partial scan
                                                // (Performed by all threads in block,
                                                // with threads in same warp communicating
                                                // through shared memory buffer ‘ptr’)

   if (lane == 31)  ptr[warp_id] = ptr[idx];    // Step 2. thread 31 in each warp copies
   __syncthreads();                             // partial-scan result into per-block
                                                // shared mem

   if (warp_id == 0) scan_warp(ptr, idx);       // Step 3. scan to accumulate bases
   __syncthreads();                             // (only performed by warp 0)

   if (warp_id > 0)                             // Step 4. apply bases to all elements
       val = val + ptr[warp_id-1];              // (performed by all threads in block)
   __syncthreads();

   ptr[idx] = val;
}
```

Building a larger scan

Example: one million element scan (1024 elements per block)



Exceeding 1 million elements requires partitioning phase two into multiple blocks

Scan implementation

- **Parallelism**
  - Scan algorithm features O(N) parallel work
  - But efficient implementations only leverage as much parallelism as required to make good utilization of the machine
    - Goal is to reduce <u>work</u> and <u>reduce</u> communication/synchronization
- **Locality**
  - Multi-level implementation to match memory hierarchy (CUDA example: per-block implementation carried out in local memory)
- **Heterogeneity in algorithm: different strategy for performing scan at different levels of the machine**
  - CUDA example: different algorithm for intra-warp scan than inter-thread scan
  - Low-core count CPU example: based largely on sequential scan

Parallel Segmented Scan

Segmented scan

- Common problem: operating on a *sequence of sequences*
- Examples:
  - For each vertex v in a graph:
    - For each edge incoming to v:
  - For each particle p in a simulation
    - For each particle within cutoff radius of p
  - For each document d in a collection
    - For each word in d
- There are two levels of parallelism in the problem that a programmer might want to exploit
- But it is irregular: the size of edge lists, particle neighbor lists, words per document, etc, may be very different from vertex to vertex (or particle to particle)

Segmented scan

- **Generalization of scan**
- **Simultaneously perform scans on contiguous partitions of input sequence**

```
let A  = [[1,2],[6],[1,2,3,4]]
let ⊕  = +
segmented_scan_exclusive(⊕,A) = [[0,1], [0], [0,1,3,6]]
```

Assume a simple "start-flag" representation of nested sequences:

```
A = [[1,2,3],[4,5,6,7,8]]
flag: 1 0 0 1 0 0 0 0
data: 1 2 3 4 5 6 7 8
```

Work-efficient segmented scan (with $\oplus$ = "+")

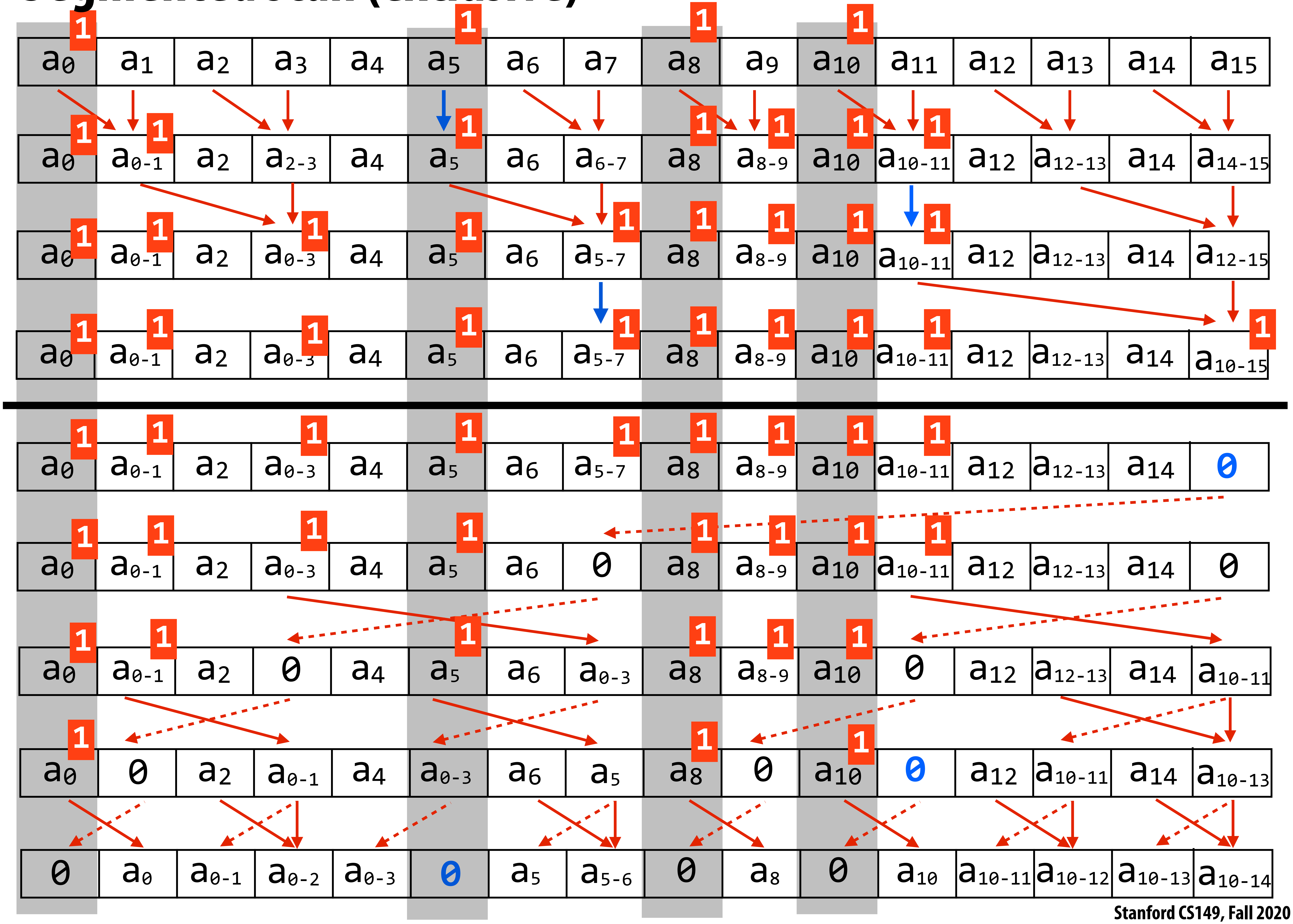
Up-sweep:

```
for d=0 to (log₂n - 1) do:
   forall k=0 to n-1 by 2^(d+1) do:
     if flag[k + 2^(d+1) - 1] == 0:
        data[k + 2^(d+1) - 1] = data[k + 2^d - 1] + data[k + 2^(d+1) - 1]
     flag[k + 2^(d+1) - 1] = flag[k + 2^d - 1] || flag[k + 2^(d+1) - 1]
```

Down-sweep:

```
data[n-1] = 0
for d=(log₂n - 1) down to 0 do:
   forall k=0 to n-1 by 2^(d+1) do:
     tmp = data[k + 2^d - 1]
     data[k + 2^d - 1] = data[k + 2^(d+1) - 1]
     if flag_original[k + 2^d] == 1:        # must maintain copy of original flags
        data[k + 2^(d+1) - 1] = 0            # start of segment
     else if flag[k + 2^d - 1] == 1:
        data[k + 2^(d+1) - 1] = tmp
     else:
        data[k + 2^(d+1) - 1] = tmp + data[k + 2^(d+1) - 1]
     flag[k + 2^d - 1] = 0
```

Segmented scan (exclusive)



1
a0 a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15
a0 a0-1 a2 a2-3 a4 a5 a6 a6-7 a8 a8-9 a10 a10-11 a12 a12-13 a14 a14-15
a0 a0-1 a2 a0-3 a4 a5 a6 a5-7 a8 a8-9 a10 a10-11 a12 a12-13 a14 a12-15
a0 a0-1 a2 a0-3 a4 a5 a6 a5-7 a8 a8-9 a10 a10-11 a12 a12-13 a14 a10-15
a0 a0-1 a2 a0-3 a4 a5 a6 a5-7 a8 a8-9 a10 a10-11 a12 a12-13 a14 0
a0 a0-1 a2 a0-3 a4 a5 a6 0 a8 a8-9 a10 a10-11 a12 a12-13 a14 0
a0 a0-1 a2 0 a4 a5 a6 a0-3 a8 a8-9 a10 0 a12 a12-13 a14 a10-11
a0 0 a2 a0-1 a4 a0-3 a6 a5 a8 0 a10 0 a12 a10-11 a14 a10-13
0 a0 a0-1 a0-2 a0-3 0 a5 a5-6 0 a8 0 a10 a10-11 a10-12 a10-13 a10-14

Sparse matrix multiplication example

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} 3 & 0 & 1 & \cdots & 0 \\ 0 & 2 & 0 & \cdots & 0 \\ 0 & 0 & 4 & \cdots & 0 \\ & \vdots & & & \\ 0 & 2 & 6 & \cdots & 8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix}$$

- **Most values in matrix are zero**
  - **Note: easy parallelization by parallelizing the different per-row dot products**
  - **But different amounts of work per row (complicates wide SIMD execution)**
- **Example sparse storage format: compressed sparse row**

values = [[3,1], [2], [4], ..., [2,6,8]]

cols = [[0,2], [1], [2], ...,]

row_starts = [0, 2, 3, 4, ...]

Sparse matrix multiplication with scan

x = [x0,x1,x2,x3]

values = [[3,1], [2], [4], [2,6,8]]

cols = [[0,2], [1], [2], [1,2,3]]

row_starts = [0, 2, 3, 4]

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 6 & 8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

1. Map over all non-zero values: products[i] = values[i] * x[cols[i]]

   products = [$3x_0$, x_2, $2x_1$, $4x_2$, $2x_1$, $6x_2$, $8x_3$]

2. Create flags vector from row_starts: flags = [1,0,1,1,0,0]

3. Perform inclusive segmented-scan on (products, flags) using addition operator

   [$3x_0$, $3x_0+x_2$, $2x_1$, $4x_2$, $2x_1$, $2x_1+6x_2$, $2x_1+6x_2+8x_2$]

4. Take last element in each segment:

   y = [$3x_0+x_2$, $2x_1$, $4x_2$, $2x_1+6x_2+8x_2$]

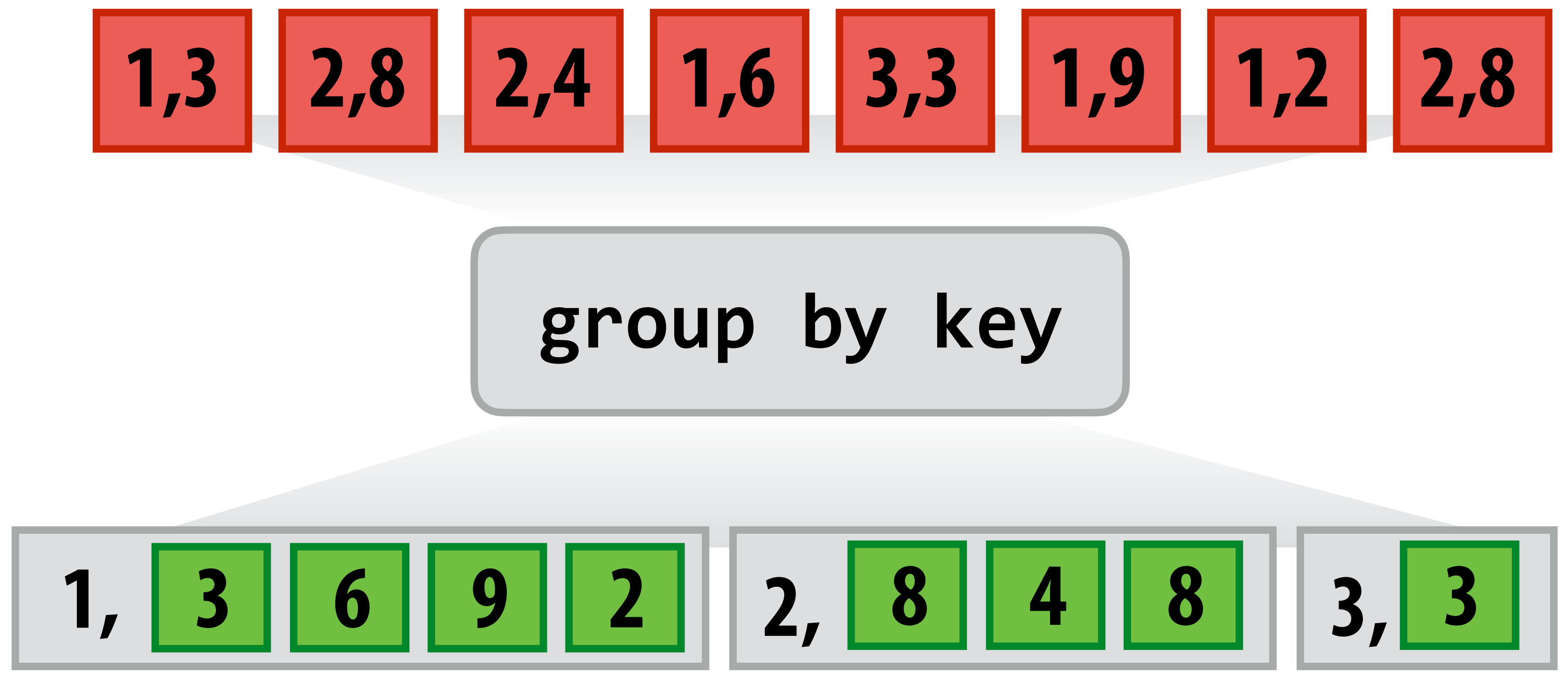
Scan/segmented scan summary

- **Scan**
  - Theory: parallelism in problem is linear in number of elements
  - Practice: exploit locality, use only as much parallelism as necessary to fill the machine's execution resources
    - Great example of applying different strategies at different levels of the machine

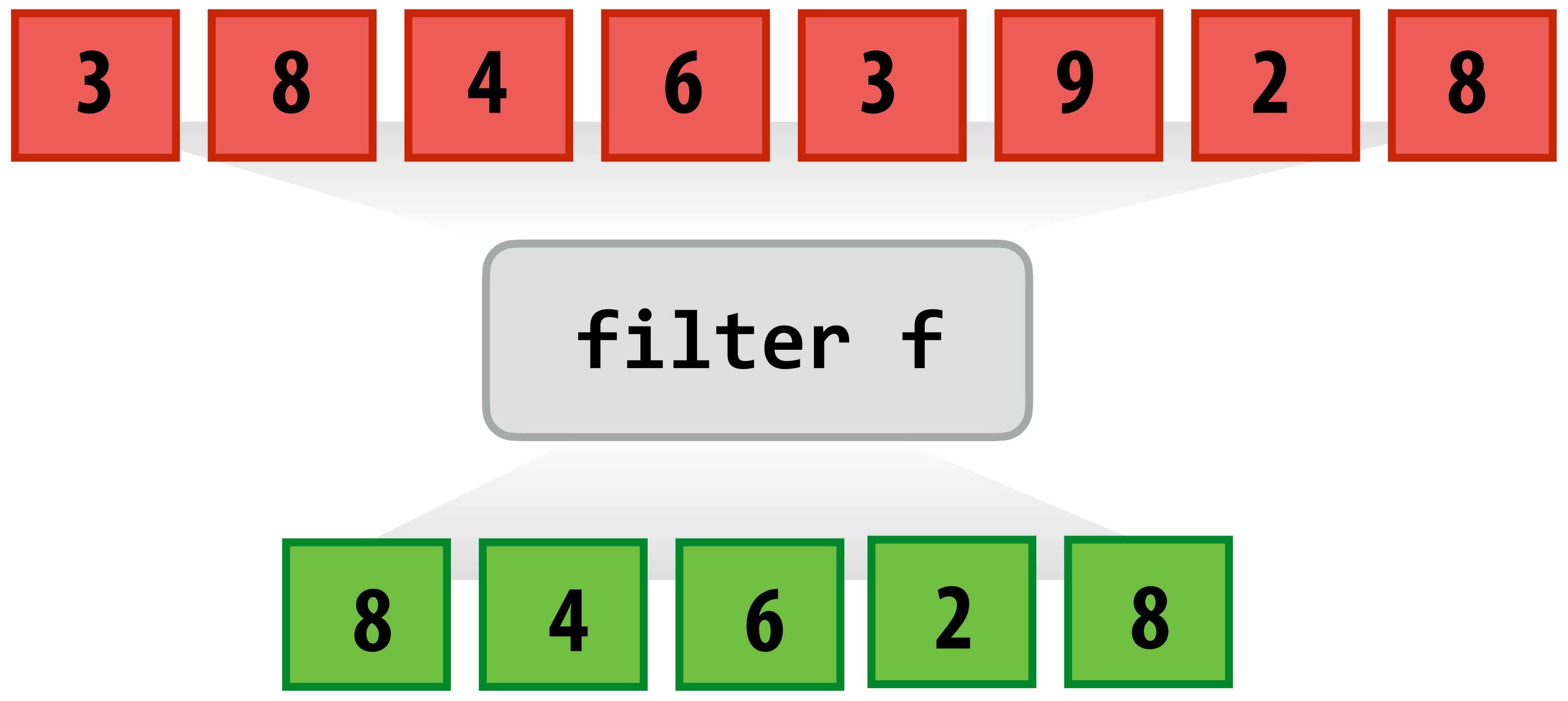
- **Segmented scan**
  - Express computation and operate on irregular data structures (e.g., list of lists) in a regular, data parallel way

More sequence operations

- Group by key
  - Seq (key, T) —> Seq (key, Seq T)
  - Creates a sequence of sequences containing elements with the same key



- Filter
  - Remove elements from sequence that do not match predicate

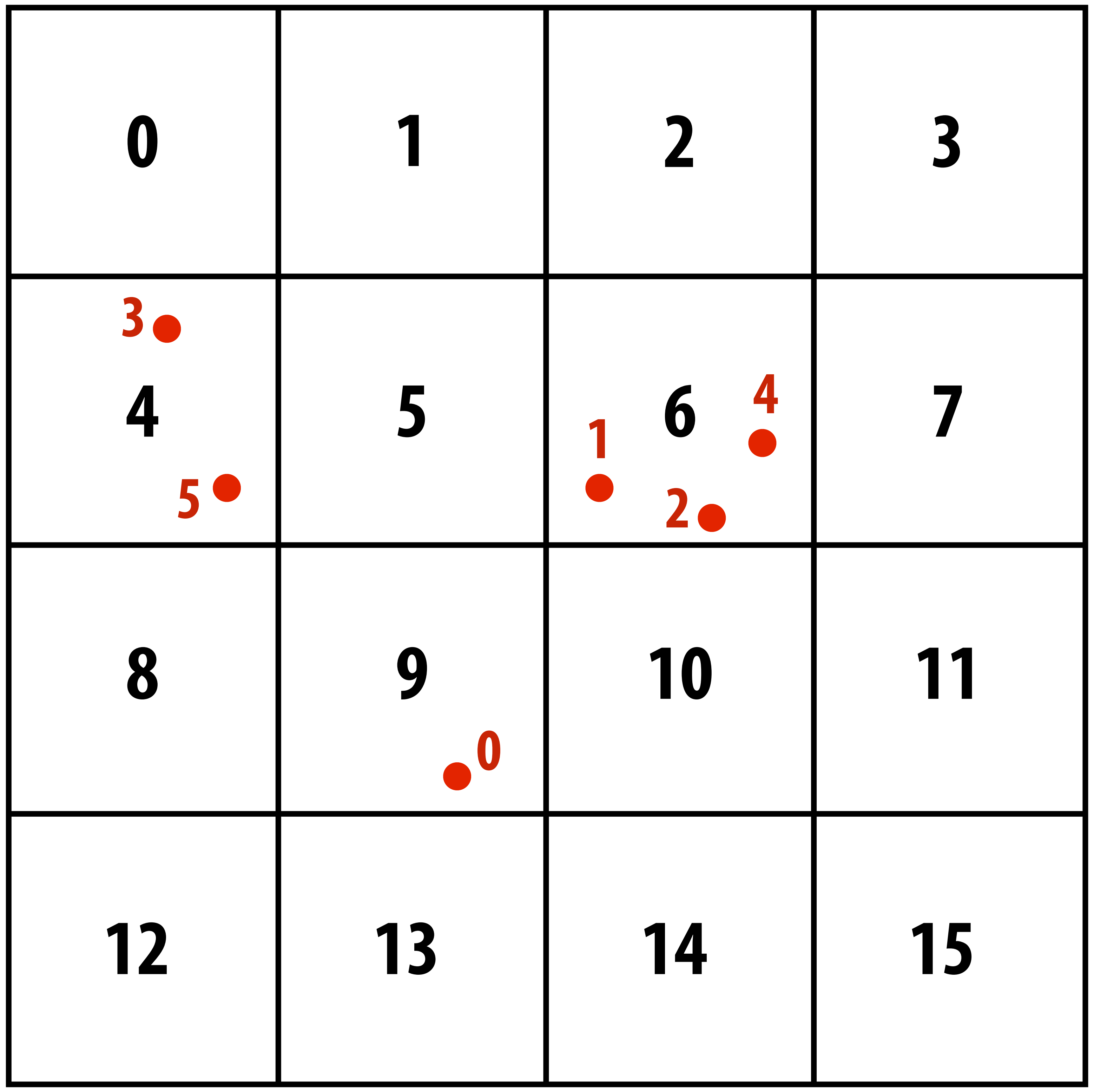


Assume f filters elements whose value is odd

- Sort

Example: create grid of particles data structure on large parallel machine (e.g., a GPU)

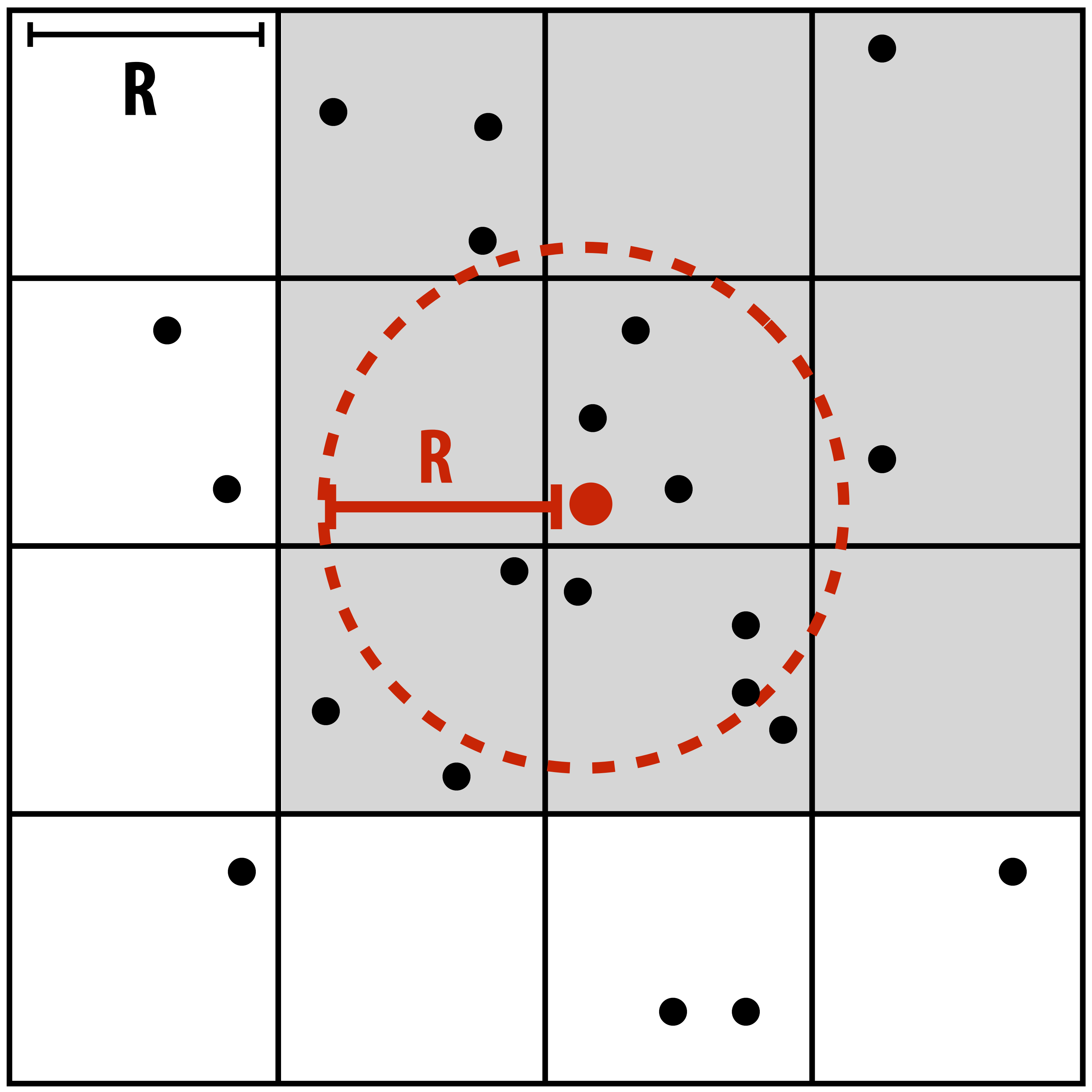
- Problem: place 1M point particles in a 16-cell uniform grid based on 2D position
  - Parallel data structure manipulation problem: build a 2D array of lists
- Recall: Up to 2048 CUDA threads per SM core on a V100 GPU (80 SM cores)



Cell id	Count	Particle id
0	0	
1	0	
2	0	
3	0	
4	2	3, 5
5	0	
6	3	1, 2, 4
7	0	
8	0	
9	1	0
10	0	
11	0	
12	0	
13	0	
14	0	
15	0	

Common use of this structure: N-body problems

- A common operation is to compute interactions with neighboring particles
- Example: given a particle, find all particles within radius R
  - Organize particles by placing them in grid with cells of size R
  - Only need to inspect particles in surrounding grid cells



Solution 1: parallelize over particles

- **One answer: assign one particle to each CUDA thread. Each thread computes cell containing particle, then atomically updates per cell list.**
    - **Massive contention: thousands of threads contending for access to update single shared data structure**

```
list cell_list[16];    // 2D array of lists
lock cell_list_lock;

for each particle p          // in parallel
   c = compute cell containing p
   lock(cell_list_lock)
   append p to cell_list[c]
   unlock(cell_list_lock)
```

Solution 2: use finer-granularity locks

- **Alleviate contention for single global lock by using per-cell locks**
  - Assuming uniform distribution of particles in 2D space... ~16x less contention than previous solution

```
list cell_list[16];    // 2D array of lists
lock cell_list_lock[16];

for each particle p           // in parallel
   c = compute cell containing p
   lock(cell_list_lock[c])
   append p to cell_list[c]
   unlock(cell_list_lock[c])
```

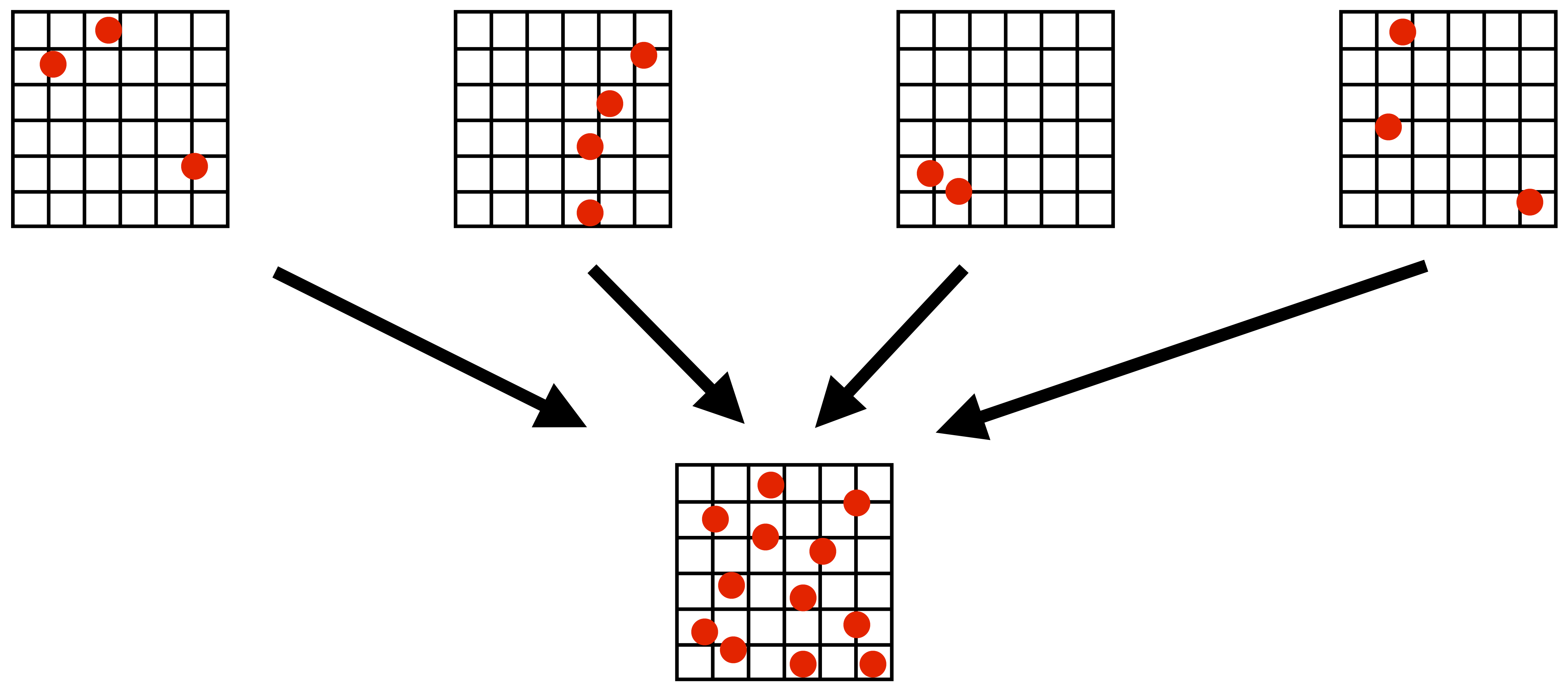
Solution 3: parallelize over cells

- **Decompose work by cells: for each cell, independently compute what particles are within it (eliminates contention because no synchronization is required)**
  - **Insufficient parallelism: only 16 parallel tasks, but need thousands of independent tasks to efficiently utilize GPU)**
  - **Work inefficient: performs 16 times more particle-in-cell computations than sequential algorithm**

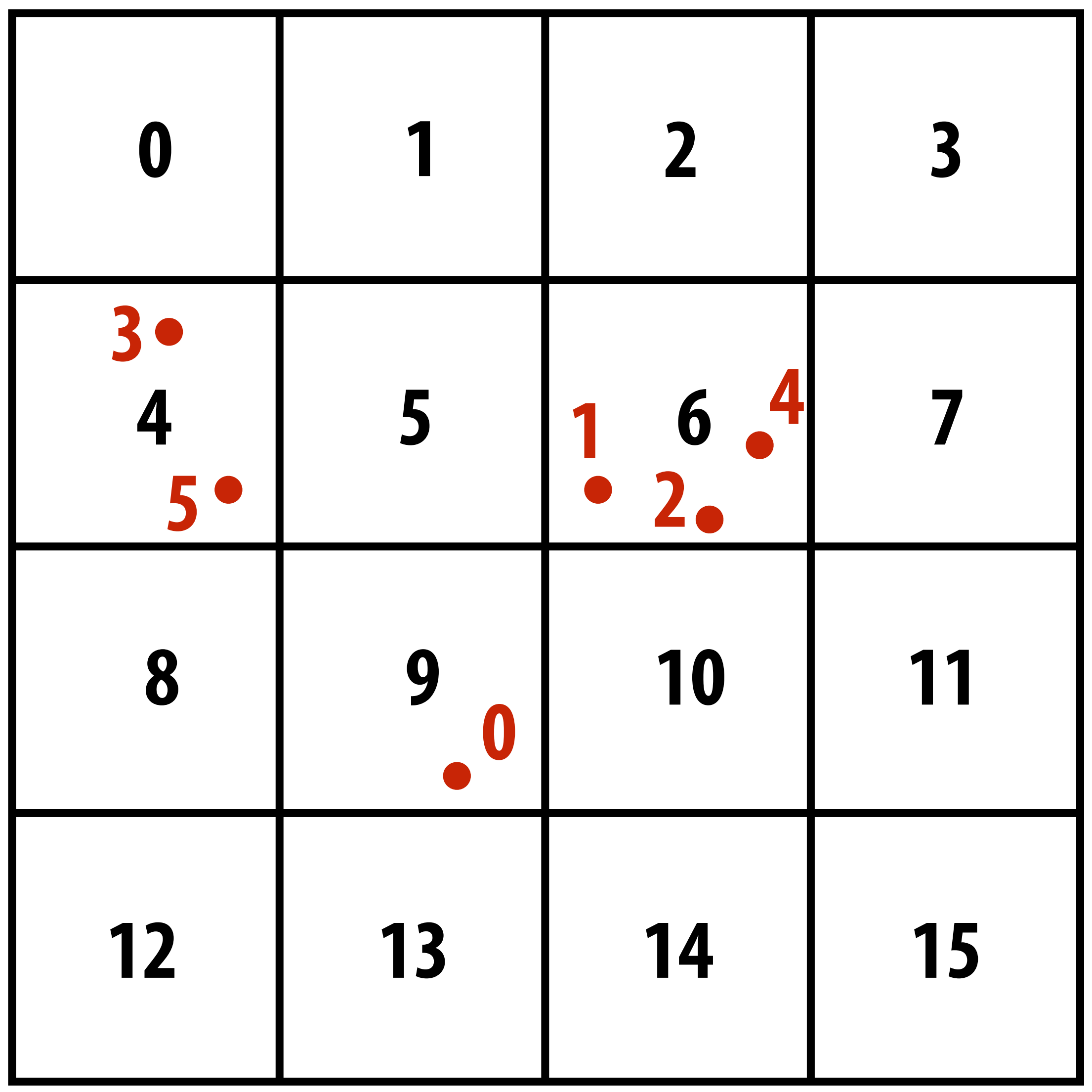
```
list cell_lists[16];      // 2D array of lists

for each cell c           // in parallel
   for each particle p    // sequentially
       if (p is within c)
          append p to cell_lists[c]
```

Solution 4: compute partial results + merge

- Yet another answer: generate N "partial" grids in parallel, then combine
  - Example: create N thread blocks (at least as many thread blocks as SM cores)
  - All threads in thread block update same grid
    - Enables faster synchronization: contention reduced by factor of N and cost of synchronization is lower because it is performed on block-local variables (in CUDA shared memory)
  - Requires extra work: merging the N grids at the end of the computation
  - Requires extra memory footprint: Store N grids of lists, rather than 1



Solution 5: data-parallel approach



Step 1: map

compute cell containing each particle (parallel over input particles)

particle_index:	0	1	2	3	4	5
grid_cell:	9	6	6	4	6	4

Step 2: sort results by cell (particle index array permuted based on sort)

particle_index:	3	5	1	2	4	0
grid_cell:	4	4	6	6	6	9

This solution maintains a large amount of parallelism and removes the need for fine-grained synchronization... at cost of a sort and extra passes over the data (extra BW)

Step 3: find start/end of each cell (parallel over particle_index elements)

```
particle_cell = grid_cell[index];
if (index == 0)
    cell_starts[particle_cell] = index;
else if (particle_cell != grid_cell[index-1]) {
    cell_starts[particle_cell] = index;
    cell_ends[grid_cell[index-1]] = index;
}
if (index == numParticles-1) // special case for last cell
    cell_ends[particle_cell] = index+1;
```

This code is run for each element of the particle_index array. (each invocation has a unique valid of 'index')

	0	1	2	3	4	5	6	7	8	9	10	
cell_starts					0		2			5		...
cell_ends (not inclusive)					2		5			6		...

Another example: parallel histogram

- **Consider computing a histogram for a sequence of values**

```
int f(float value);              // maps array values to bin id’s

float input[N];
int histogram_bins[NUM_BINS];  // assume bins are initialized to 0

for (int i=0; i<N; i++) {
   histogram_bins[f(input[i])]++;
}
```

- **Create a massively parallel implementation of histogram given only map() and sort() on sequences**

Data-parallel histogram construction

```
void compute_bin(float* input, int* bin_ids) {
   bin_ids[idx] = f(input[idx]);
}

void find_starts(int* bin_ids, int* bin_starts) {
   if (idx == 0 || bin_ids[idx] != bin_ids[idx-1])
      bin_starts[bin_ids[idx]] = idx;
}

float input[N];
int   bin_ids[N];         // bin_ids[i] = id of bin that element i goes in
int   sorted_bin_idx[N];
int   bin_starts[NUM_BINS];  // initialized to -1

// map f onto input sequence to get bin ids of all elements
launch<<<N>>>compute_bin(input, bin_ids);

// find starting point of each bin in sorted list
sort(N, bin_ids, sorted_bin_ids);
launch<<<N>>>find_starts(sorted_bin_ids, bin_starts);
```

Assume variable idx is the "thread idx" associated with the invocation of the kernel function

Data-parallel histogram construction

```
void bin_sizes(int* bin_starts, int* histogram_bins, int num_items, int num_bins) {

if (bin_starts[idx] == -1) {
    histogram_bins[idx] = 0;        // no items in this bin
} else {

    // find start of next bin in order to determined size of current bin

    // Tricky edge case: if the next bin is empty, then must search forward to find
    the next non-empty bin
    int next_idx = idx+1;
    while(next_idx < num_bins && bin_starts[next_idx] == -1)
       id++;

    if (next_idx < num_bins)
       histogram_bins[idx] = bin_starts[next_idx] - bin_starts[idx];
    else
       histogram_bins[idx] = num_items - bin_starts[idx];
   }
}



launch<<<NUM_BINS>>>bin_sizes(bin_starts, histogram_bins, N, NUM_BINS);
```

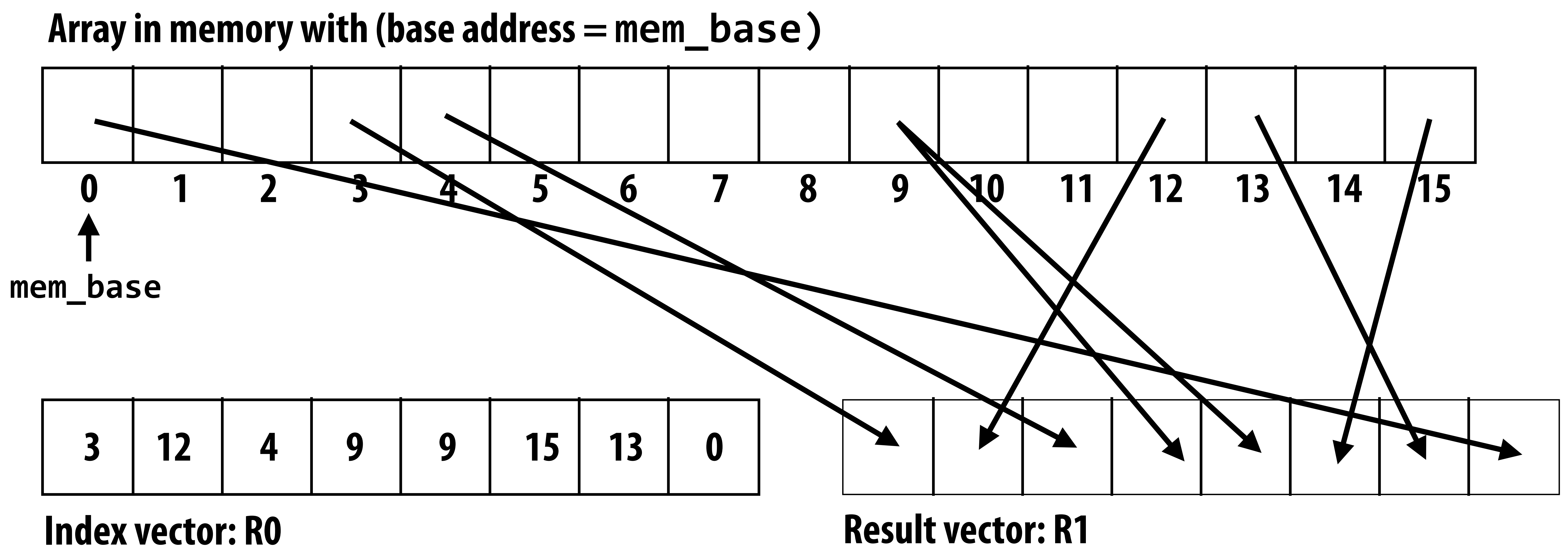
Scatter/gather operations on sequences

- **gather(index, input, output)**
  - `output[i] = input[index[i]]`

- **scatter(index, input, output)**
  - `output[index[i]] = input[i]`

Gather instruction

`gather(R1, R0, mem_base);` **"Gather from buffer `mem_base` into R1 according to indices specified by R0."**



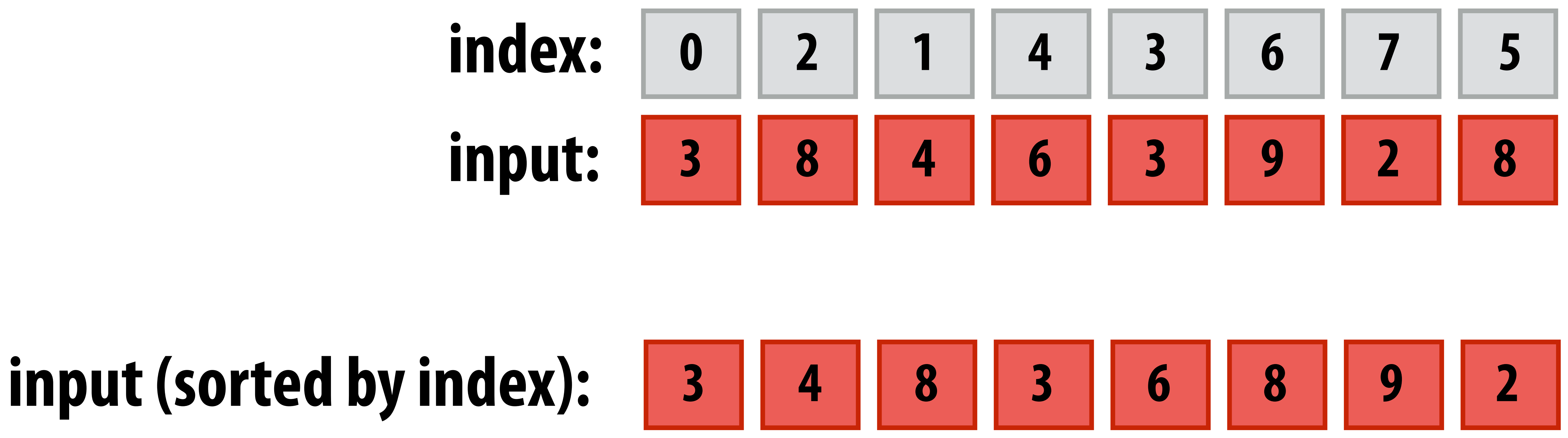
Gather supported with AVX2 in 2013
But AVX2 does not support SIMD scatter (must implement as scalar loop)
Scatter instruction exists in AVX512

Hardware supported gather/scatter does exist on GPUs.
(still an expensive operation compared to load/store of contiguous vector)

Turning a scatter into sort/gather

Special case: assume elements of index are unique and all elements referenced in index (scatter is a permutation)

```
scatter(index, input, output) {
  temp = sort input sequence by values in index sequence
  output[i] = temp[i]
}
```



Implementing scatter with atomic sort/map/scan

Assume elements in index are not unique, so synchronization is required for atomicity!

```
for all elements in sequence
    output[index[i]] = atomicOp(output[index[i]], input[i])
```

```
e.g,: index = [1, 1, 0, 2, 0, 0]
```

Step 1: Sort input sequence according to values in index sequence:

Sorted index:

```
[0, 0, 0, 1, 1, 2]
```

Input sorted by index:

```
[input[2], input[4], input[5], input[0], input[1], input[3]]
```

Step 2: Compute starts of each range of values with the same index number

```
starts: [1, 0, 0, 1, 0, 1]
```

Step 3: Segmented scan (using 'op') on each range

```
[op(op(input[2], input[4]), input[5]), op(input[0], input[1]), input[3])
```

Summary

- Data parallel thinking:
  - Implementing algorithms in terms of simple (often widely parallelizable, efficiently implemented) operations on large data collections

- Turn irregular parallelism into regular parallelism

- Turn fine-grained synchronization into coarse synchronization

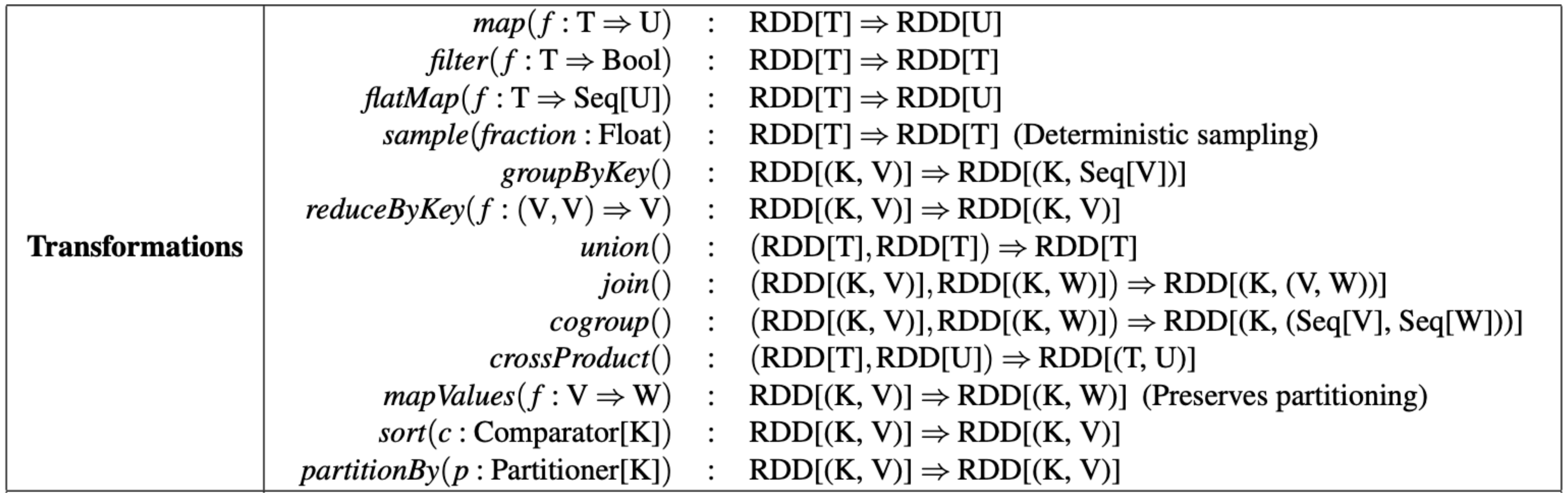
- But most solutions require multiple passes over data — bandwidth hungry!

Summary

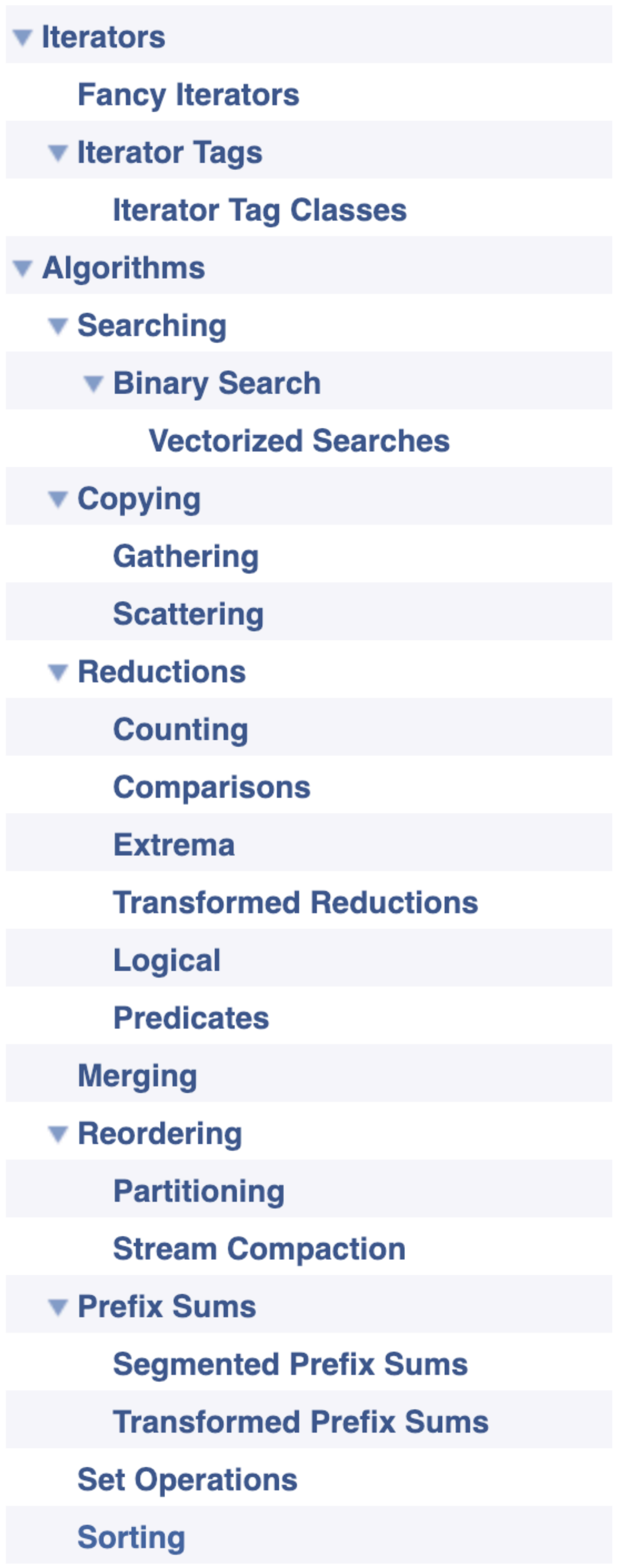
- Data parallel primitives are basis for many parallel/distributed systems today

- CUDA's Thrust Library

- Apache Spark / Hadoop



Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (Seq[V], Seq[W]))]
	crossProduct()	:	(RDD[T], RDD[U]) ⇒ RDD[(T, U)]
	mapValues(*f* : V ⇒ W)	:	RDD[(K, V)] ⇒ RDD[(K, W)] (Preserves partitioning)
	sort(*c* : Comparator[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	partitionBy(*p* : Partitioner[K])	:	RDD[(K, V)] ⇒ RDD[(K, V)]



- ▼ Iterators
  - Fancy Iterators
  - ▼ Iterator Tags
    - Iterator Tag Classes
- ▼ Algorithms
  - ▼ Searching
    - ▼ Binary Search
      - Vectorized Searches
  - ▼ Copying
    - Gathering
    - Scattering
  - ▼ Reductions
    - Counting
    - Comparisons
    - Extrema
    - Transformed Reductions
    - Logical
    - Predicates
  - Merging
  - ▼ Reordering
    - Partitioning
    - Stream Compaction
  - ▼ Prefix Sums
    - Segmented Prefix Sums
    - Transformed Prefix Sums
  - Set Operations
  - Sorting