

Lecture 11:

Data Parallel Thinking

Parallel Computing

Stanford CS149, Winter 2019

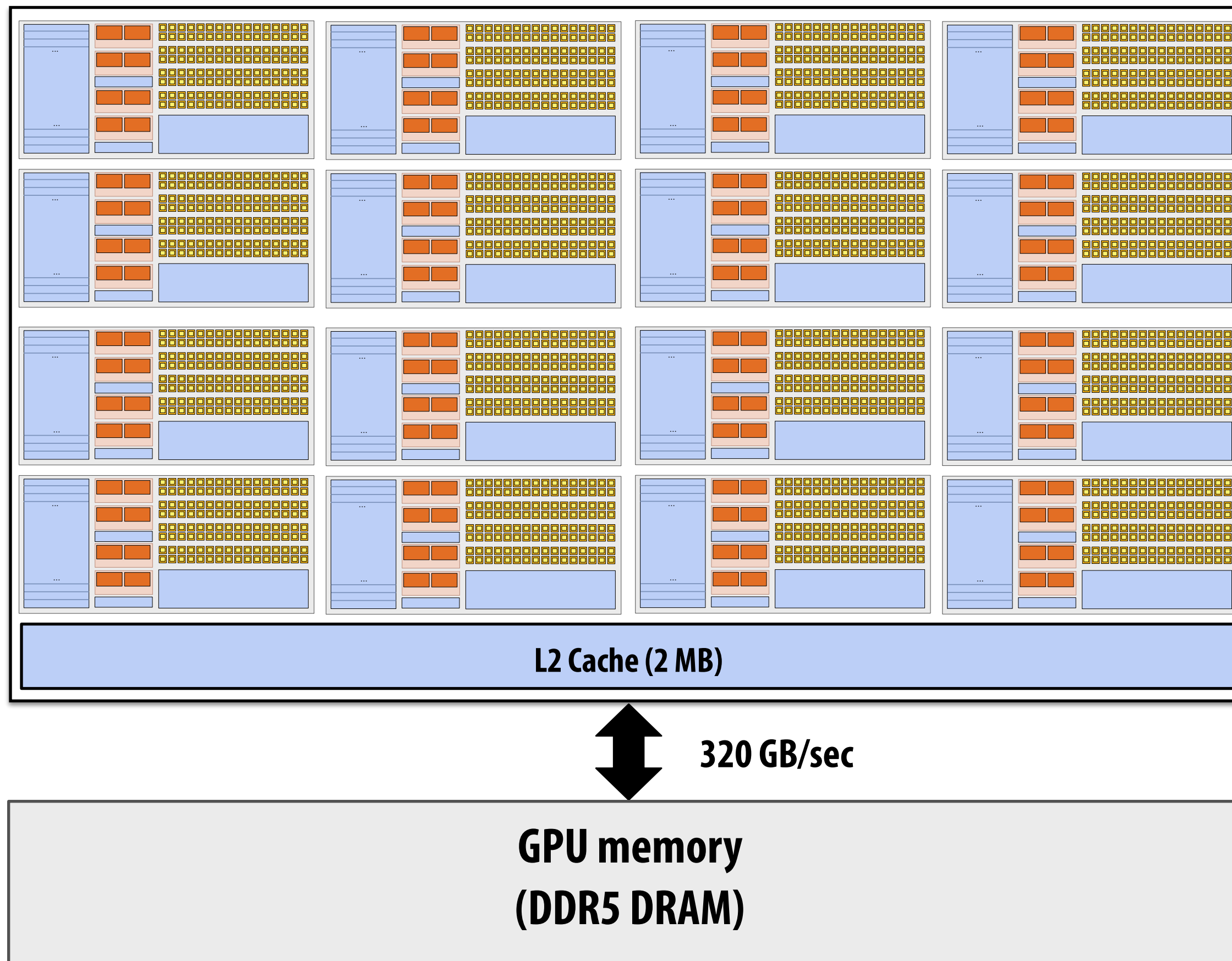
Today's theme

- **Many of you are likely accustomed to thinking about parallel programming in terms of “what workers do”**
- **Today I would like you to think about describing programs in terms of operations on sequences of data**
 - **map**
 - **filter**
 - **fold / reduce**
 - **scan / segmented scan**
 - **sort**
 - **groupBy**
 - **join**
- **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 GTX 1080 GPU



1.6 GHz clock

20 SM cores per chip

$20 \times 128 = 2,560$ SIMD mul-add ALUs
= 8.1 TFLOPs

Up to $20 \times 64 = 1280$ interleaved
warps per chip (40,960 CUDA
threads/chip)

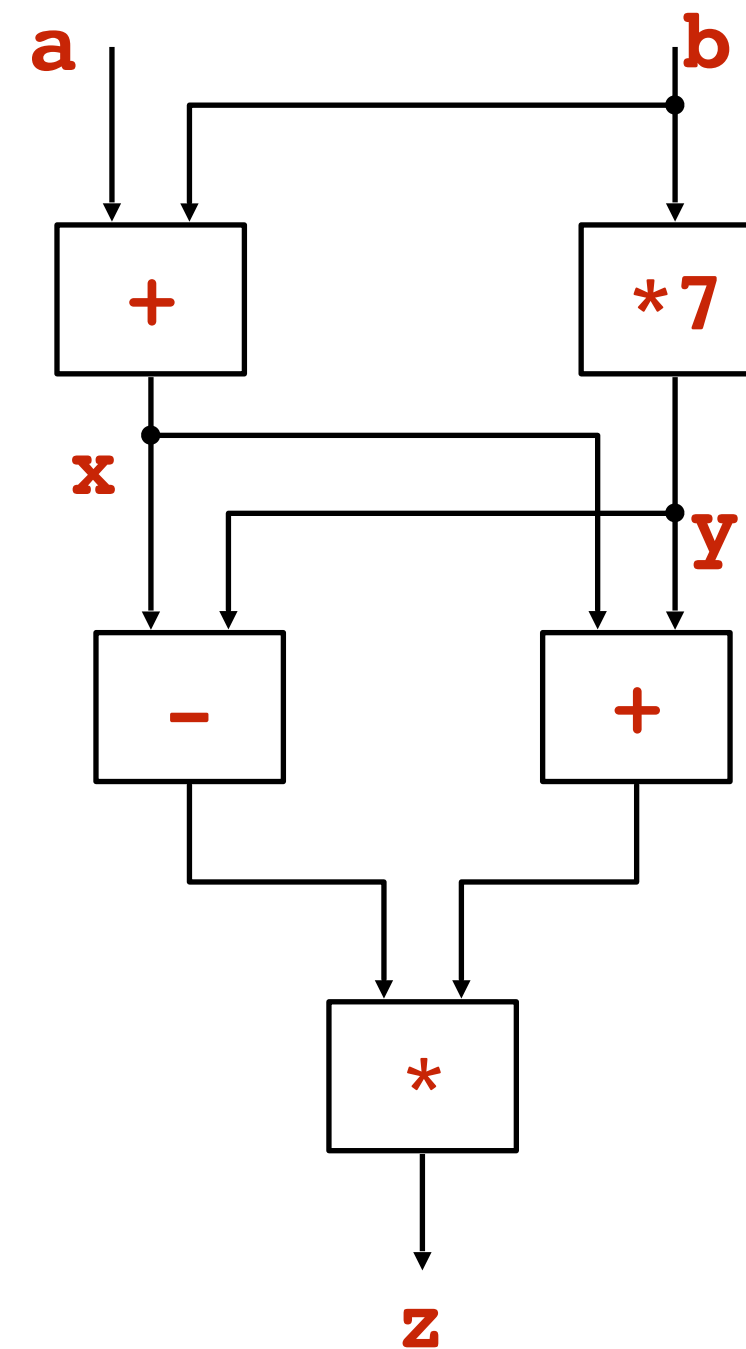
TDP: 180 watts

**Chip can concurrently execute up to 40,960 CUDA threads!
(programs that do not expose significant amounts of
parallelism will not run efficiently on GPUs!)**

Recall

- 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);
```

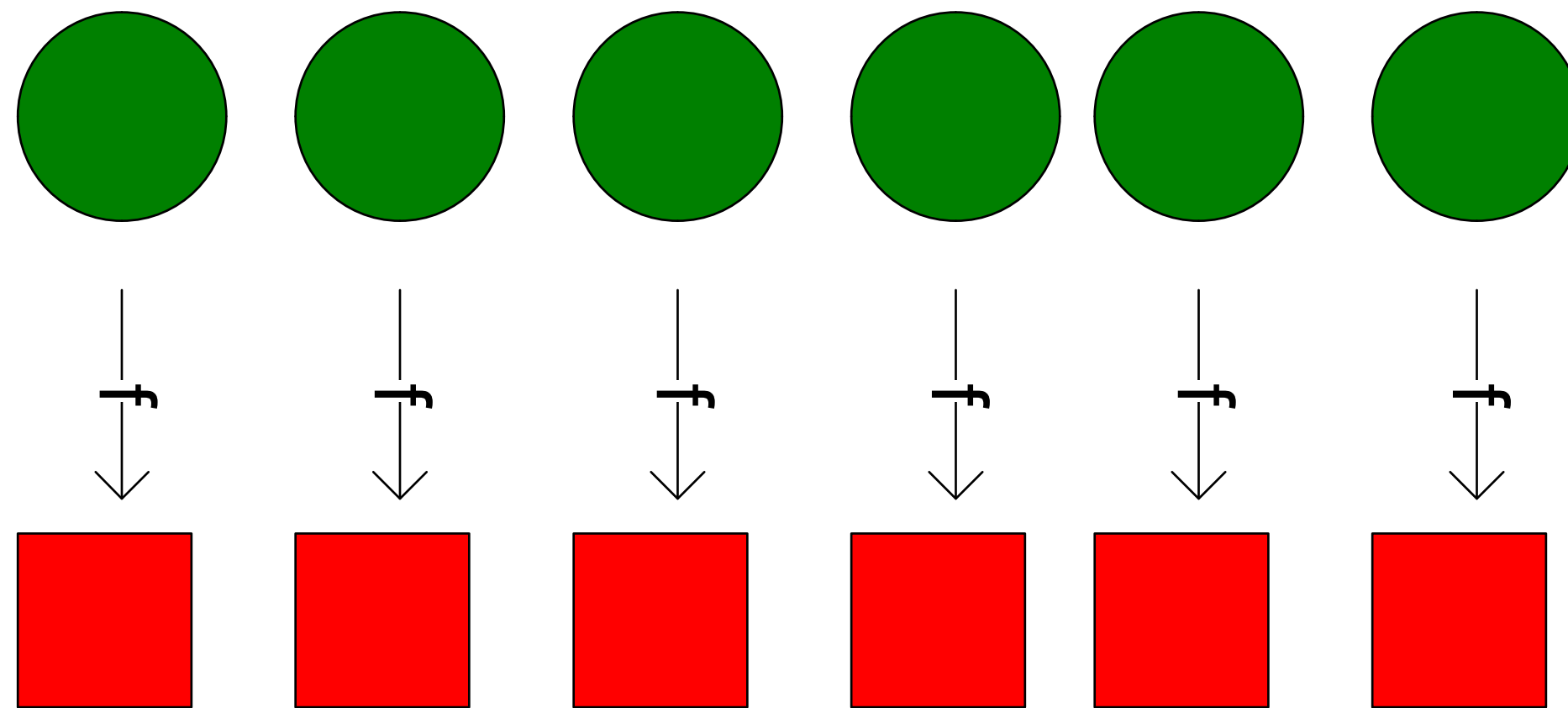


Sequences

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

Map

- Higher order function
- Apply side-effect free function $f :: a \rightarrow b$ to all elements of input sequence, to produce output sequence of the same length
- $\text{map} :: (a \rightarrow b) \rightarrow \text{seq } a \rightarrow \text{seq } b$



- E.g., in Scala: `def map[A, B](f: A => B, l: List[A]): List[B]`

Parallelizing map

- Since $f :: a \rightarrow b$ is a function (it is side effect free), then application of f to all elements of the sequence can be done in any order without changing the output of the program
- The implementation has flexibility to reorder/parallelize however it sees fit

```
map f s =  
  partition s into P smaller sequences  
  for each subsequence s_i, in parallel  
    out_i = execute map f s_i  
  out = concatenate out_i's
```


Fold (fold left)

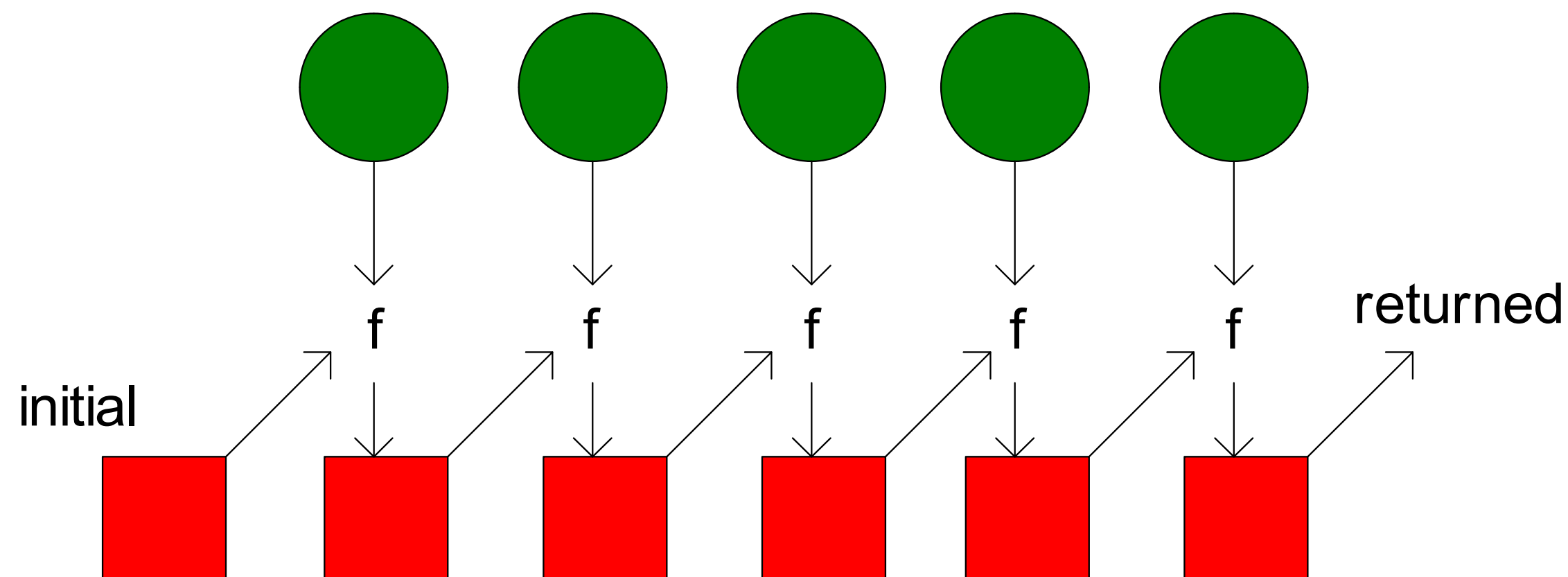
- Apply f to each element and an accumulated value
 - Seeded by initial value of type b

■ $f :: (b, a) \rightarrow b$

■ $\text{fold} :: b \rightarrow ((b, a) \rightarrow b) \rightarrow \text{seq } a \rightarrow b$

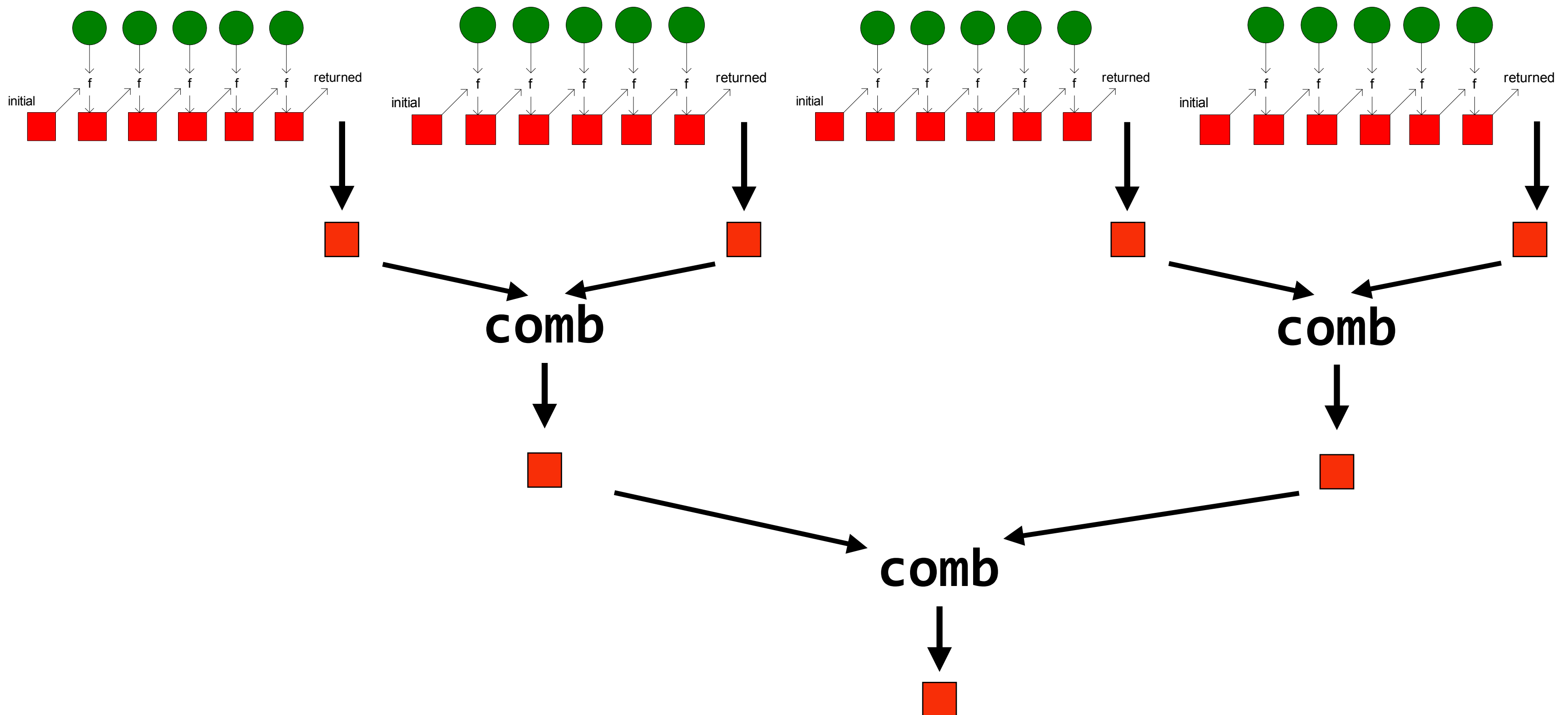
E.g., in Scala:

```
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
 - Seeded by initial value of type b (identity for f and comb)
- $f :: (b, a) \rightarrow b$
- $\text{comb} :: (b, b) \rightarrow b$
- $\text{fold_par} :: b \rightarrow ((b, a) \rightarrow b) \rightarrow ((b, b) \rightarrow b) \rightarrow \text{seq } a \rightarrow b$



Scan

- $f :: (a, a) \rightarrow a$ (associative binary op)
- $\text{scan} :: a \rightarrow ((a, a) \rightarrow a) \rightarrow \text{seq } a \rightarrow \text{seq } a$

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

Parallel Scan

Data-parallel scan

let $A = [a_0, a_1, a_2, a_3, \dots, a_{n-1}]$

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

scan_inclusive $(\oplus, A) = [a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \dots$

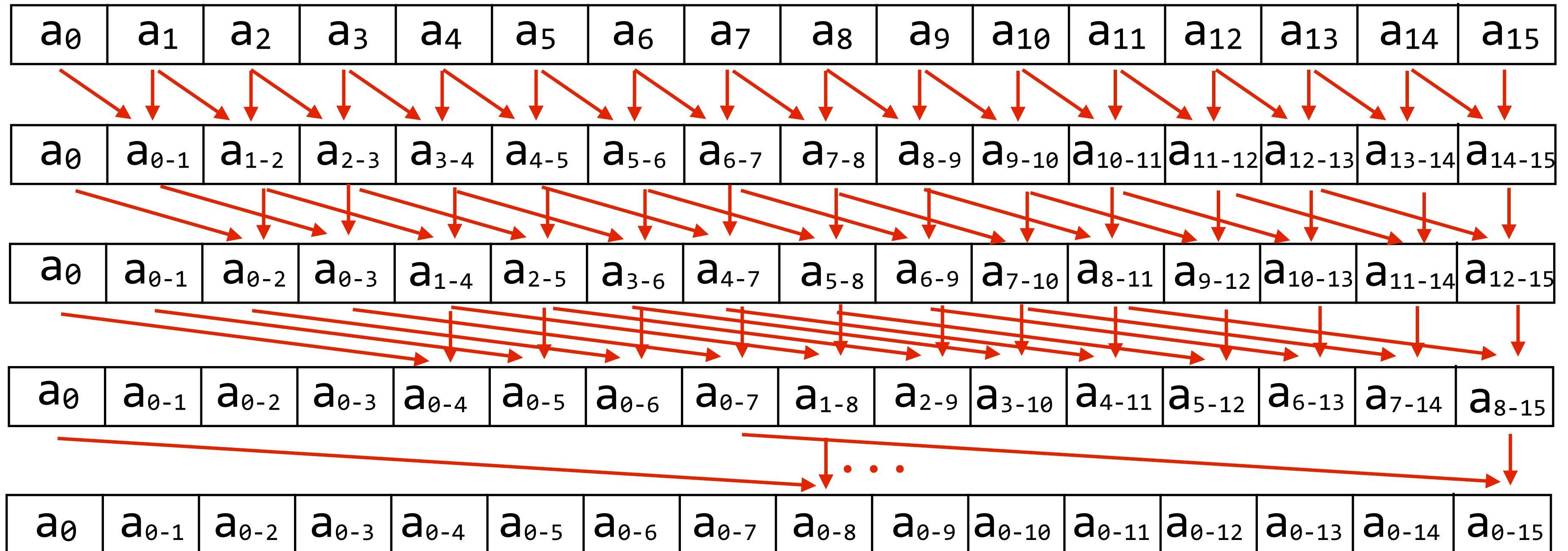
scan_exclusive $(\oplus, A) = [I, a_0, a_0 \oplus a_1, \dots$

If operator is $+$, **then** **scan_inclusive** $(+, A)$ **is a prefix sum**

prefix_sum $(A) = [a_0, a_0 + a_1, a_0 + a_1 + a_2, \dots$

Data-parallel inclusive scan

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



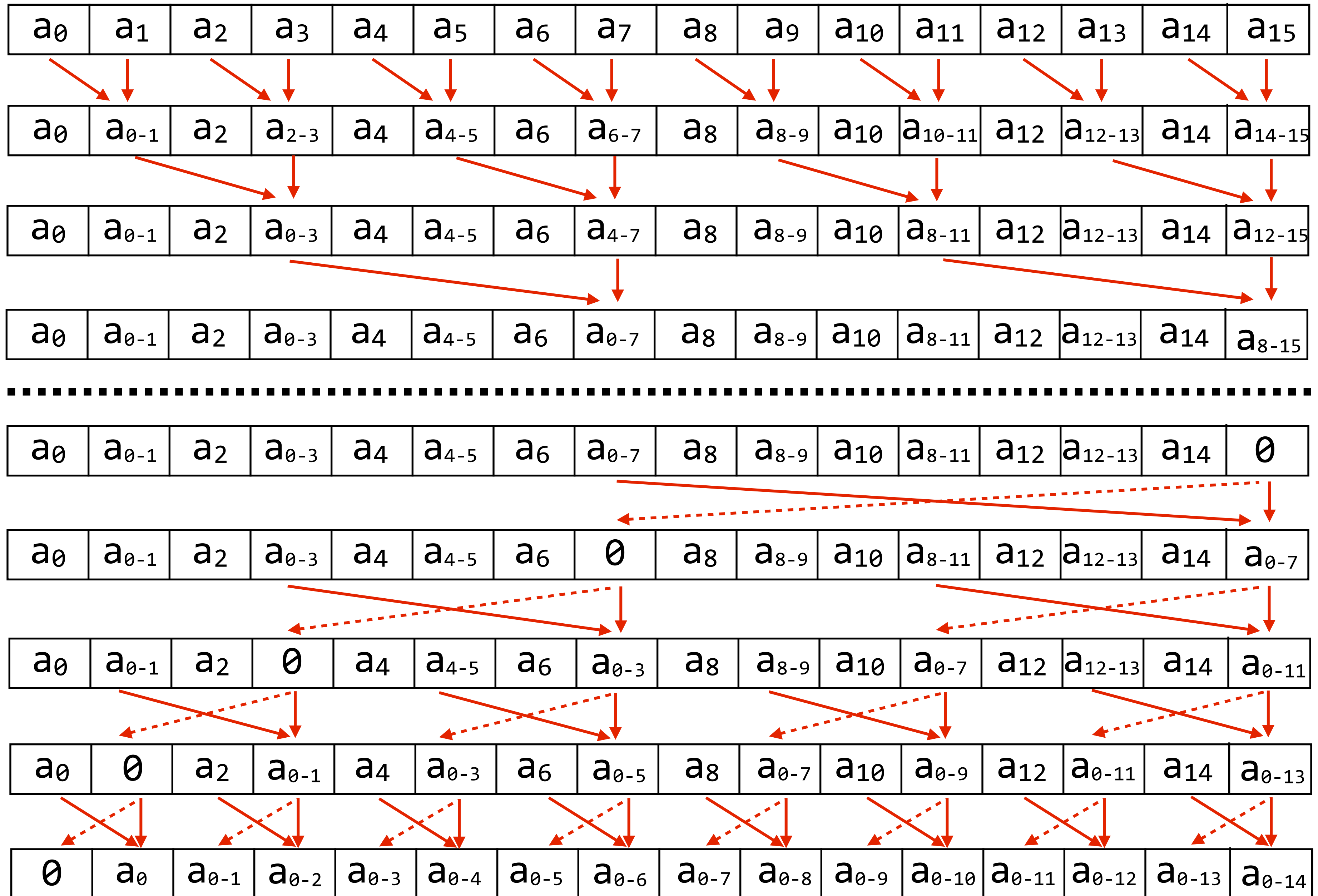
* not showing all dependencies in last step

Work: $O(N \lg N)$

Inefficient compared to sequential algorithm!

Span: $O(\lg N)$

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



Work efficient exclusive scan algorithm

(with $\oplus = "+"$)

Up-sweep:

```
for d=0 to (log2n - 1) do
  forall k=0 to n-1 by 2d+1 do
    a[k + 2d+1 - 1] = a[k + 2d - 1] + a[k + 2d+1 - 1]
```

Down-sweep:

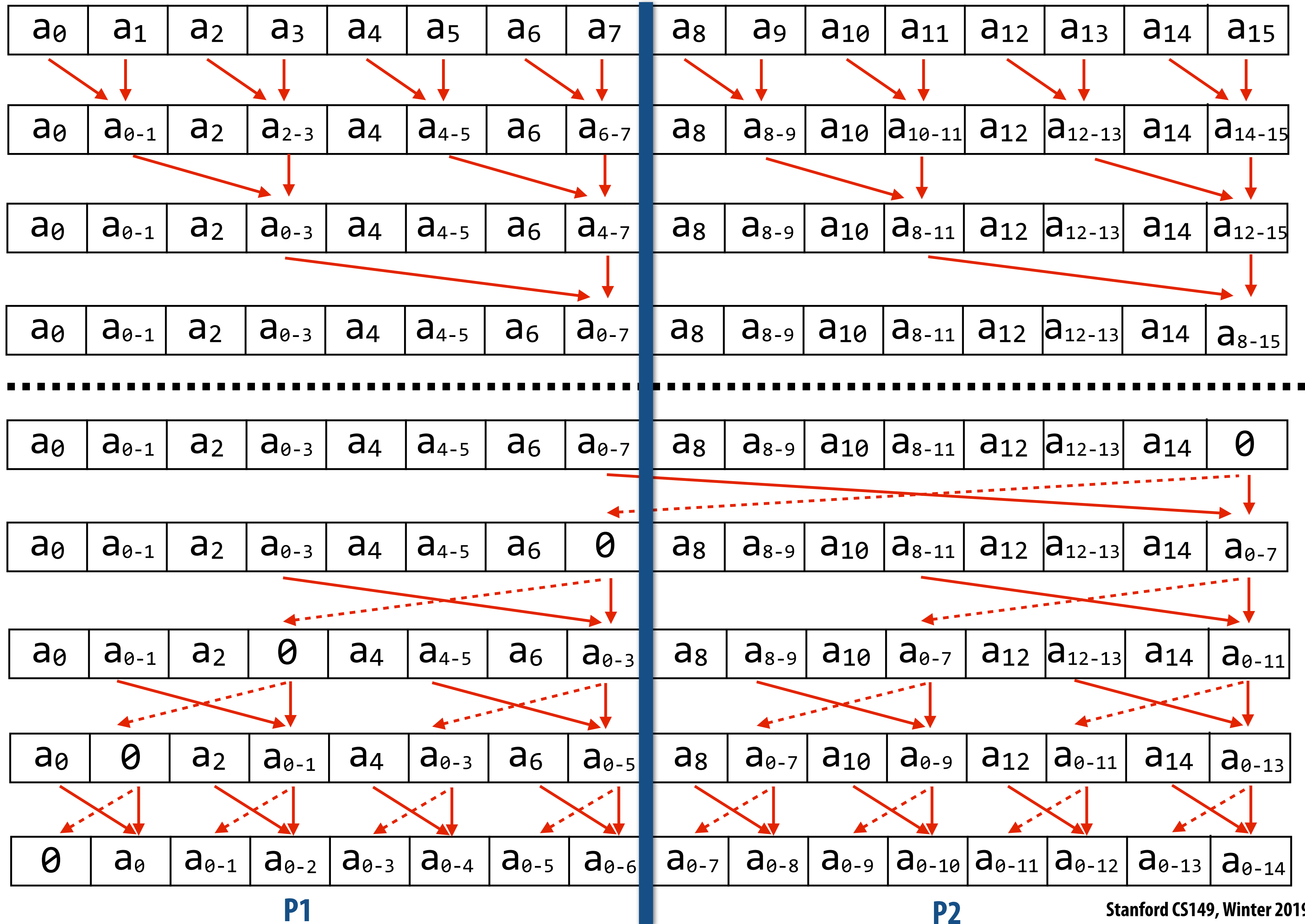
```
x[n-1] = 0
for d=(log2n - 1) down to 0 do
  forall k=0 to n-1 by 2d+1 do
    tmp = a[k + 2d - 1]
    a[k + 2d - 1] = a[k + 2d+1 - 1]
    a[k + 2d+1 - 1] = tmp + a[k + 2d+1 - 1]
```

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



Exclusive scan: two processor 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

Sequential scan on elements [0-7]

Let base = a_{0-7}

Add base to elements a_8 thru a_{8-11}

P2

Sequential scan on elements [8-15]

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 NUMA system, but on small-scale system access likely same cost 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)

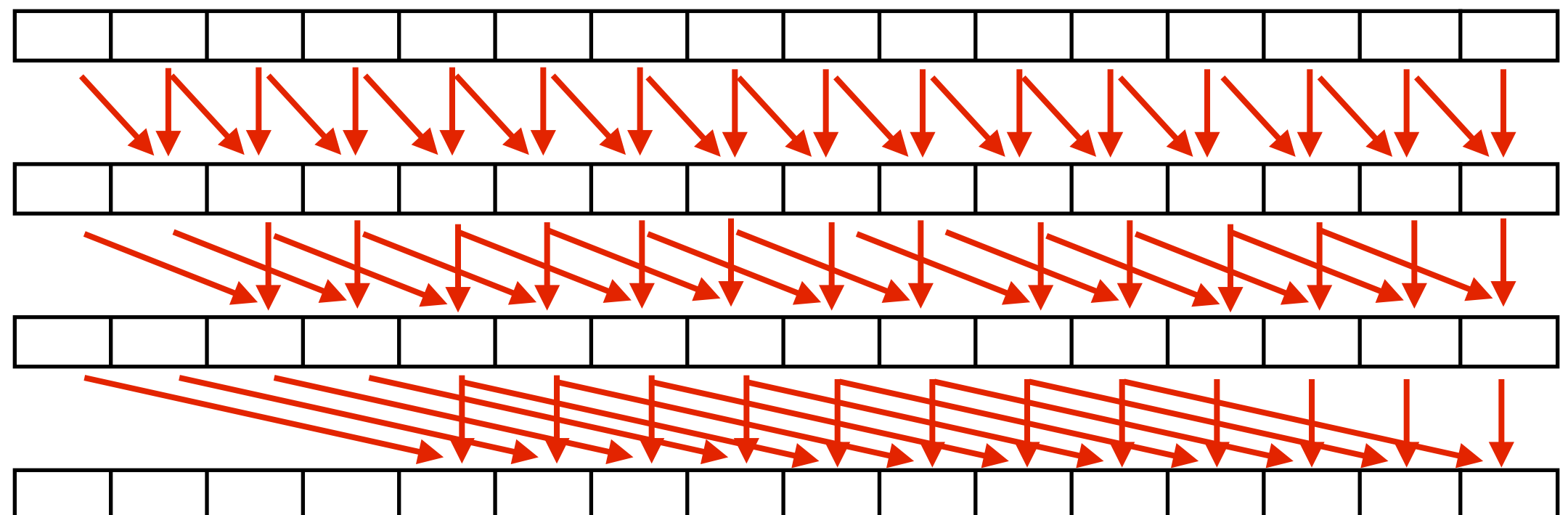
CUDA thread
index of caller

```
__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];

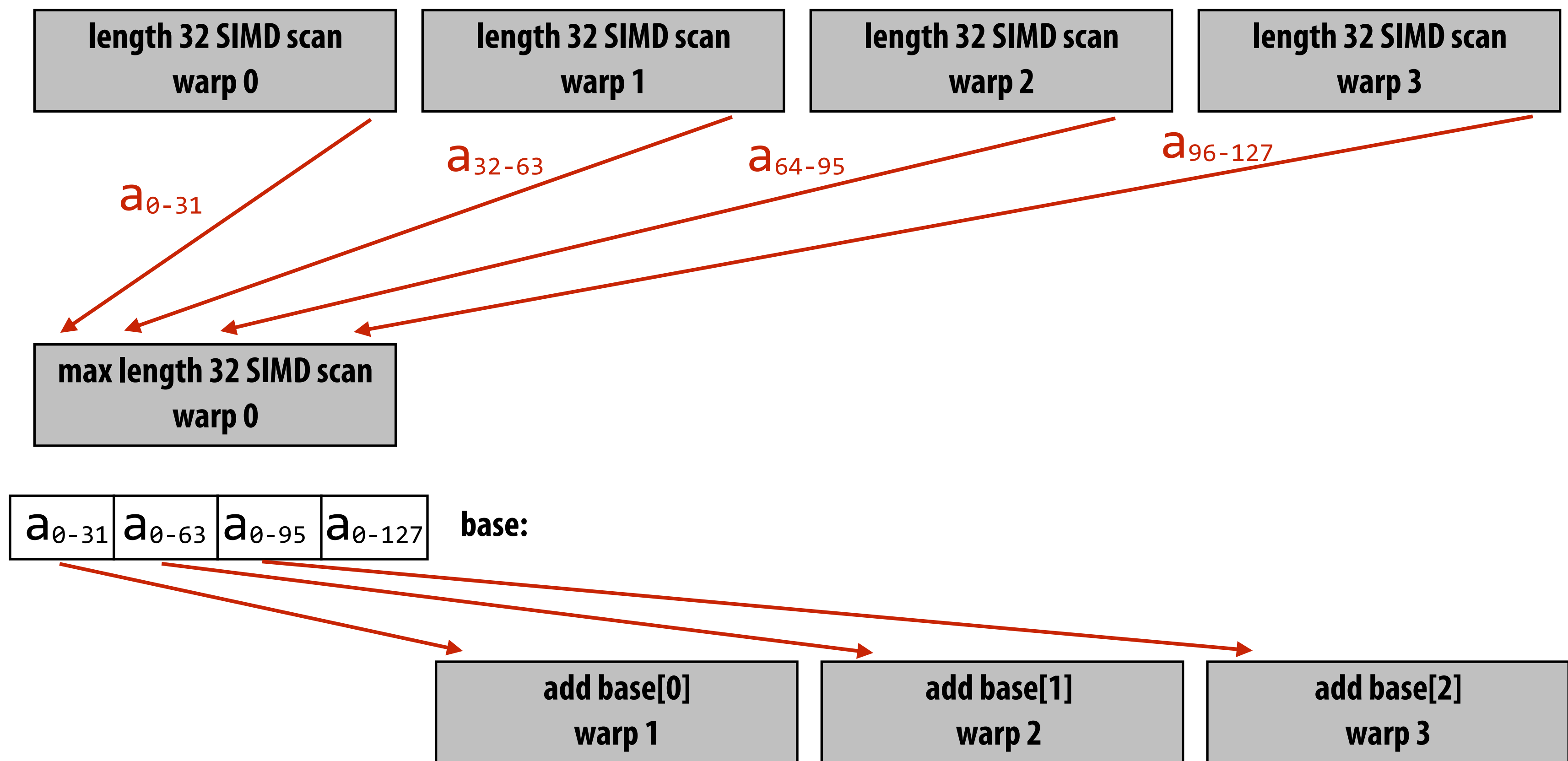
    return (lane > 0) ? ptr[idx-1] : 0;
}
```

Work: $N \lg(N)$

Work-efficient formulation of scan is not beneficial in this context because it results in low SIMD utilization. It 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



Multi-threaded, SIMD implementation

Example: cooperating threads in a CUDA thread block perform scan

We provided similar code in assignment 2.

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 bases in per-block
                                           // shared mem

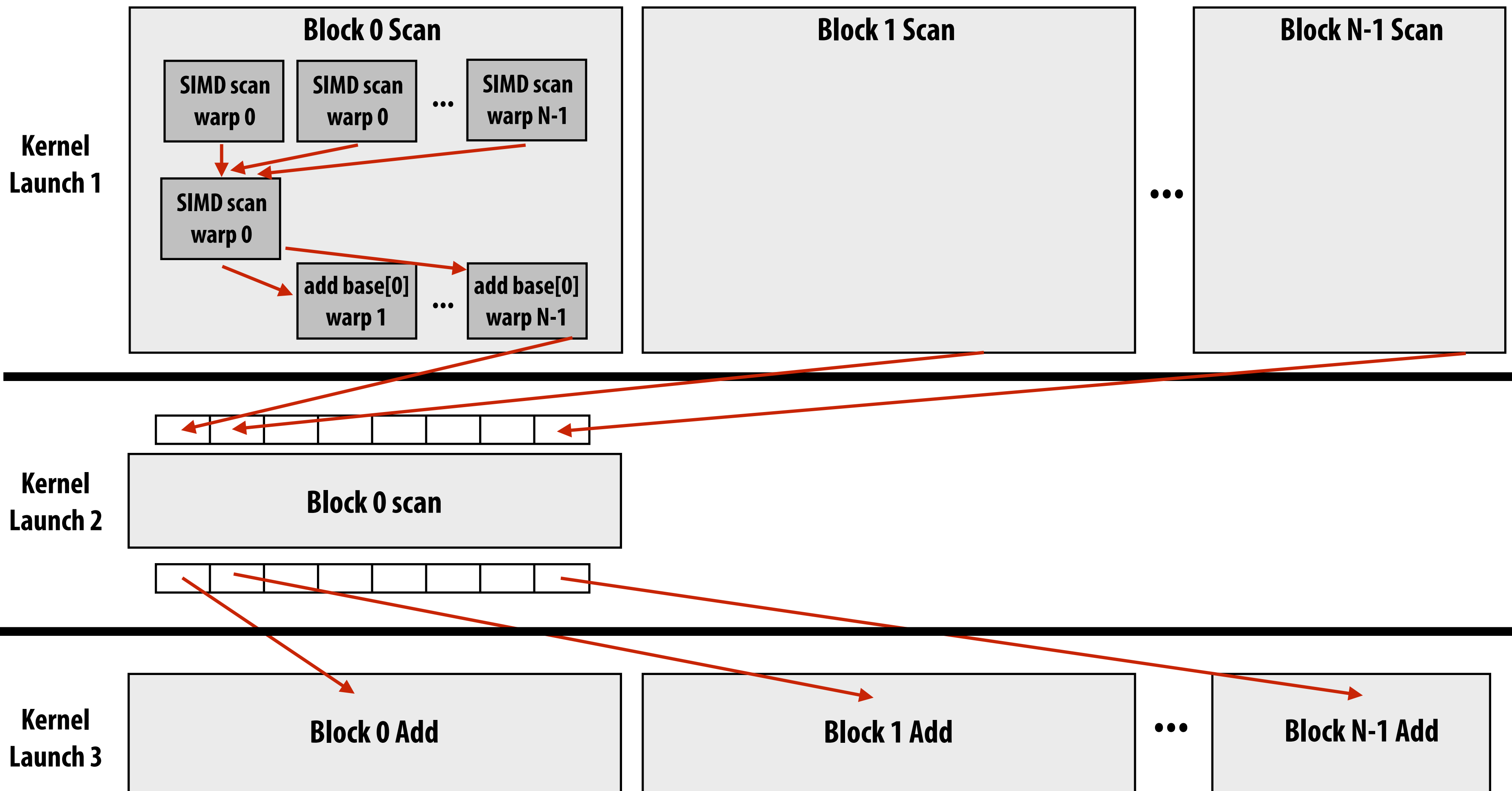
    if (warp_id == 0) scan_warp(ptr, idx);   // Step 3. scan to accumulate bases
    __syncthreads();                        // (only performed by warp 0)

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

    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 work and reduce communication/synchronization

■ Locality

- Multi-level implementation to match memory hierarchy
(CUDA example: per-block implementation carried out in local memory)

■ Heterogeneity: different strategy at different machine levels

- 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 sequence of sequences**
- **Examples:**
 - **For each vertex in a graph:**
 - **For each edge incoming to vertex:**
 - **For each particle in simulation**
 - **For each particle within cutoff radius**
 - **For each document in a collection**
 - **For each word in document**
- **There are two levels of parallelism in the problem that a programmer might want to exploit**
- **But its 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 arbitrary contiguous partitions of input collection

```
let A = [[1,2],[6],[1,2,3,4]]
```

```
let ⊕ = +
```

```
segmented_scan_exclusive(⊕,A) = [[0,1], [0], [0,1,3,6]]
```

We'll assume a simple "head-flag" representation:

```
A = [[1,2,3],[4,5,6,7,8]]
```

```
flag: 0 0 0 1 0 0 0 0
```

```
data: 1 2 3 4 5 6 7 8
```

Work-efficient segmented scan

(with $\oplus = "+"$)

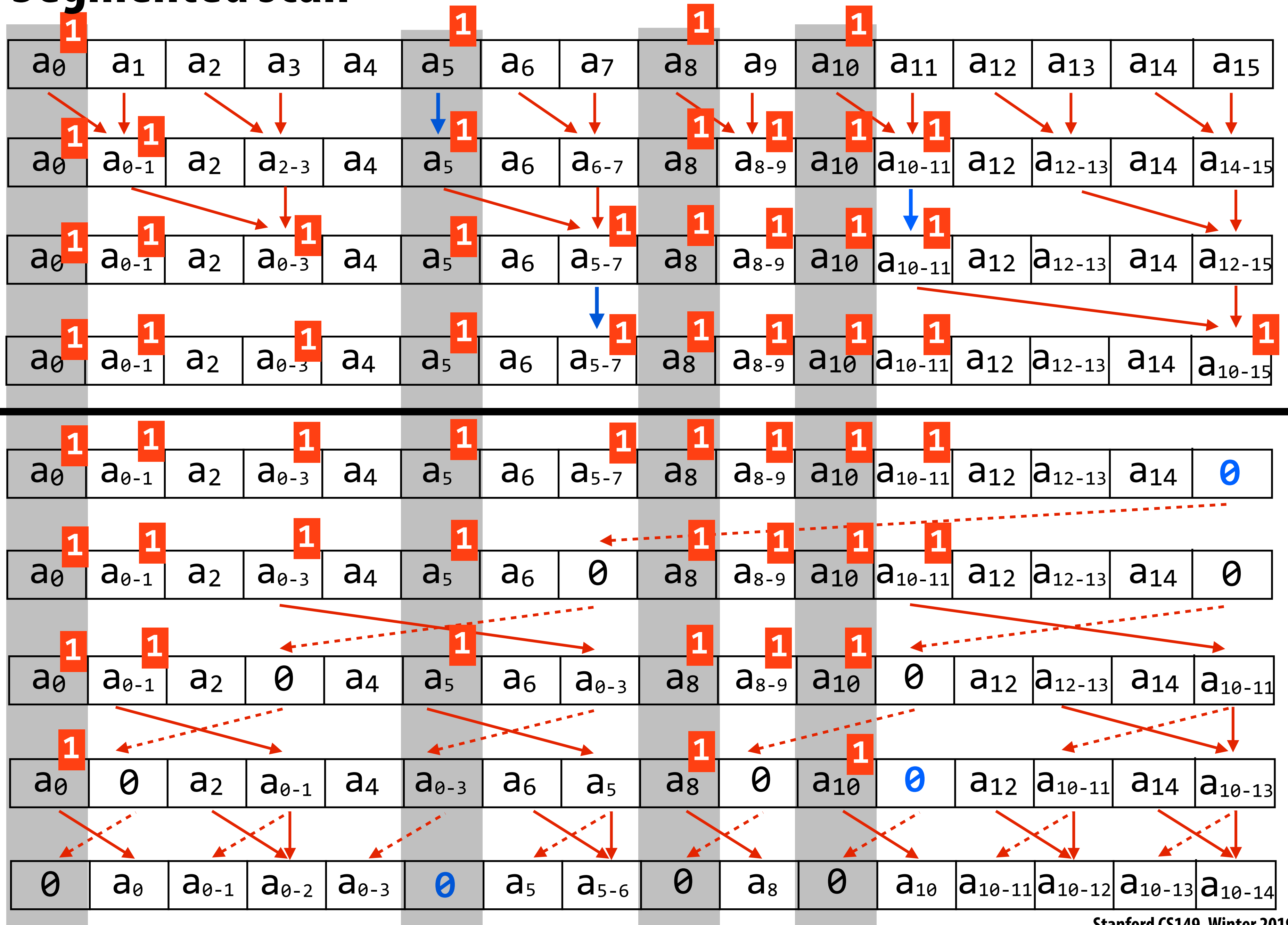
Up-sweep:

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

Down-sweep:

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

Segmented scan



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: logical parallelization is across 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

$$\begin{aligned} \text{values} &= [[3,1], [2], [4], [2,6,8]] \\ \text{cols} &= [[0,2], [1], [2], [1,2,3]] \\ \text{row_starts} &= [0, 2, 3, 4] \end{aligned} \quad \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 2 & 6 & 8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

1. Map over all non-zero values: $\text{products}[i] = \text{values}[i] * x[\text{cols}[i]]$
 $\text{products} = [3x_0, x_2, 2x_1, 4x_2, 2x_1, 6x_2, 8x_3]$
2. Create flags vector from row_starts: $\text{flags} = [1,0,1,1,0,0]$
3. Inclusive segmented-scan on (multiples, 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

- **Parallel implementation of (intuitively sequential application)**
- **Theory: parallelism in problem is linear in number of elements**
- **Practice: exploit locality, use only as much parallelism as necessary to fill the machine**
 - **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 operations

■ Group by key

- **Seq (key, T) \longrightarrow Seq (key, Seq T)**
- **Similar to: sort sequence by key, the performing segmented scan operations on segments with same key**

■ Filter

- **remove elements from sequence that do not match predicate**

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

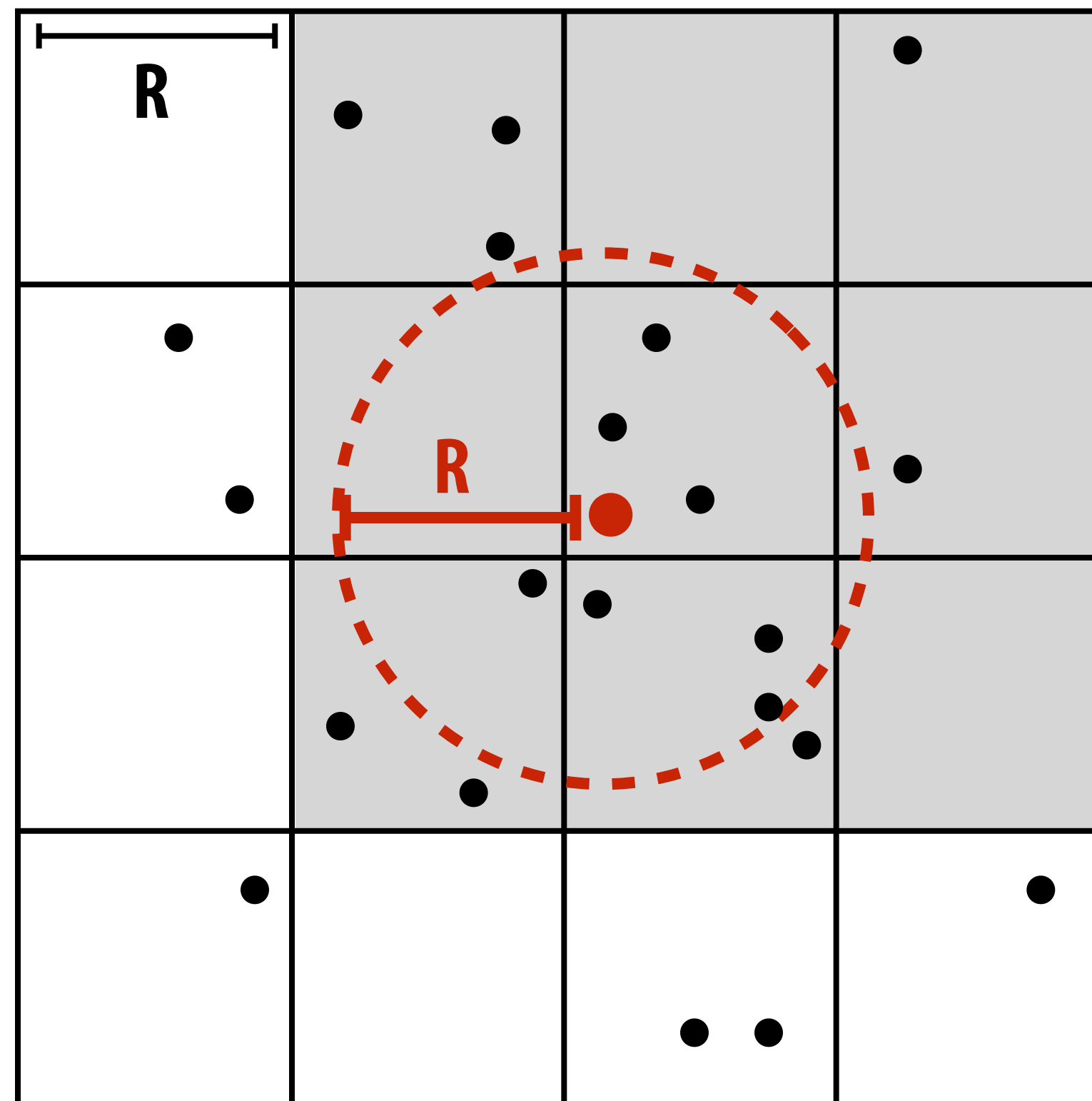
- **Problem: place 1M point particles in a 16-cell uniform grid based on 2D position**
 - **Parallel data structure manipulation problem: build a 2D array of lists**
- **Recall: Up to 2048 CUDA threads per SM core on a GTX 1080 GPU (20 SM cores)**

0	1	2	3
3 ● 4	5	1 ● 6 4 ● 2 ●	7
5 ●			
8	9	10	11
	0 ●		
12	13	14	15

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 particle, find all particles within radius R
 - Create grid with cells of size R
 - Only need to inspect particles in surrounding grid cells



Solution 1: parallelize over cells

- **One possible answer is to decompose work by cells: for each cell, independently compute particles 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 2: parallelize over particles

- **Another answer: assign one particle to each CUDA thread. Thread computes cell containing particle, then atomically updates 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 3: use finer-granularity locks

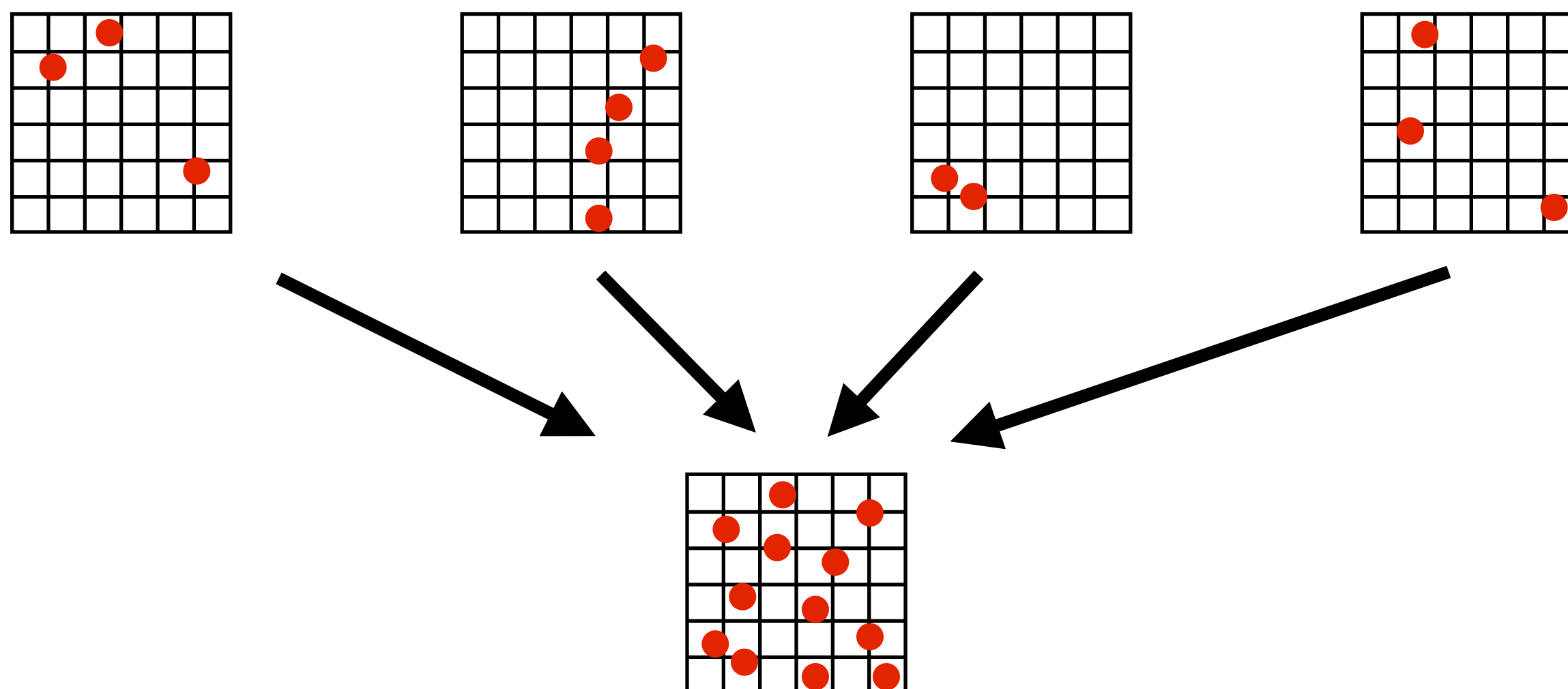
- **Alleviate contention for single global lock by using per-cell locks**
 - **Assuming uniform distribution of particles in 2D space... ~16x less contention than solution 2**

```
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 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_index:	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_index:	4	4	6	6	6	9

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

```

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

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)

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

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

0	1	2	3
3• 4 5•	5	1• 6 2•	4• 7
8	9 0•	10	11
12	13	14	15

Another example: parallel histogram

- Consider compute a histogram for a large sequence of values

```
int bin_func(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[bin_func(input[i])]++;
}
```

- Create a massively parallel implementation given map() and sort()

Another example: parallel histogram (part 1)

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

void find_starts(int* bin_ids, int* starts) {
    if (idx == 0 || bin_ids[idx] != bin_ids[idx-1])
        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 bin_func onto input 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_idx);
launch<<<N>>>find_starts(sorted_bin_idx, bin_starts);
```

Another example: parallel histogram

```
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);
```

Scatter/gather

- Earlier in class we introduced the sequence operations **gather** and **scatter**
- **gather(index, input, output)**
 - $\text{output}[i] = \text{input}[\text{index}[i]]$
- **scatter(index, input, output)**
 - $\text{output}[\text{index}[i]] = \text{input}[i]$

Turning scatter into sort/gather

- Assume elements of index are unique
- `scatter(index, input, output)`
 - `output[index[i]] = input[i]`

```
temp = sort input sequence according to values in index sequence
output[i] = temp[i]
```

Turning scatter with atomic sort/map/scan

for all elements in sequence

`atomicOp(output[index[i]], input[i])`

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

Sort input sequence according to values in index sequence

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

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

Compute starts of each range of the same index number

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

Segmented scan (using 'op') 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 (barrier) 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	$map(f : T \Rightarrow U) : RDD[T] \Rightarrow RDD[U]$
	$filter(f : T \Rightarrow Bool) : RDD[T] \Rightarrow RDD[T]$
	$flatMap(f : T \Rightarrow Seq[U]) : RDD[T] \Rightarrow RDD[U]$
	$sample(fraction : Float) : RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling)
	$groupByKey() : RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$
	$reduceByKey(f : (V, V) \Rightarrow V) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	$union() : (RDD[T], RDD[T]) \Rightarrow RDD[T]$
	$join() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$
	$cogroup() : (RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$
	$crossProduct() : (RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$
	$mapValues(f : V \Rightarrow W) : RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning)
	$sort(c : Comparator[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	$partitionBy(p : Partitioner[K]) : RDD[(K, V)] \Rightarrow RDD[(K, V)]$

- ▼ 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