

Parallel Programming

Data-Parallel Primitives:
Split and Sort

Split and Sort

Primitive: Split

Input: $R_{in}[1, \dots, n]$, $func(R_{in}[i]) \in [1, \dots, F]$, $i=1, \dots, n$.

Output: $R_{out}[1, \dots, n]$.

Function: $\{R_{out}[i], i=1, \dots, n\} = \{R_{in}[i], i=1, \dots, n\}$
and $func(R_{out}[i]) \leq func(R_{out}[j]), \forall i, j \in [1, \dots, n], i \leq j$.

Primitive: Sort

Input: $R_{in}[1, \dots, n]$.

Output: $R_{out}[1, \dots, n]$.

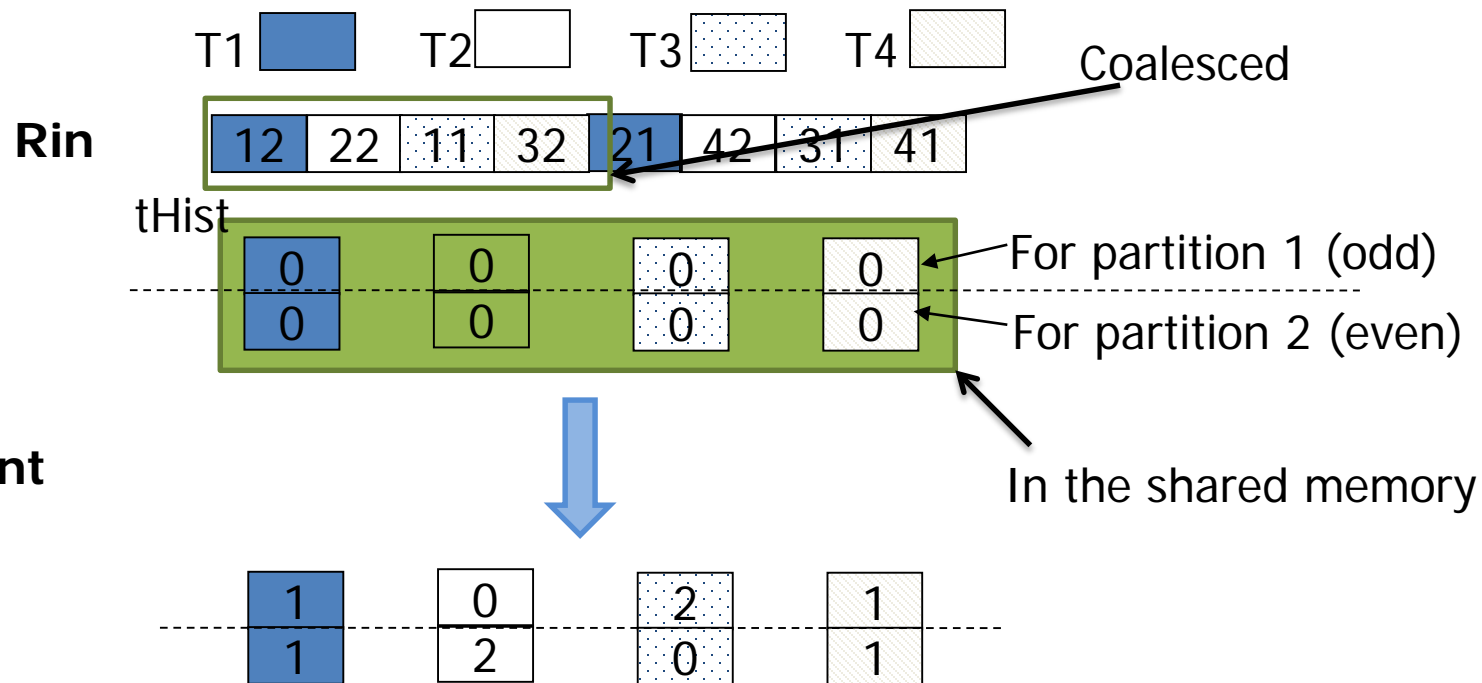
Function: $\{R_{out}[i], i=1, \dots, n\} = \{R_{in}[i], i=1, \dots, n\}$ and
 $R_{out}[i] \leq R_{out}[j], \forall i, j \in [1, \dots, n]$ and $i \leq j$.

Algorithm for Split

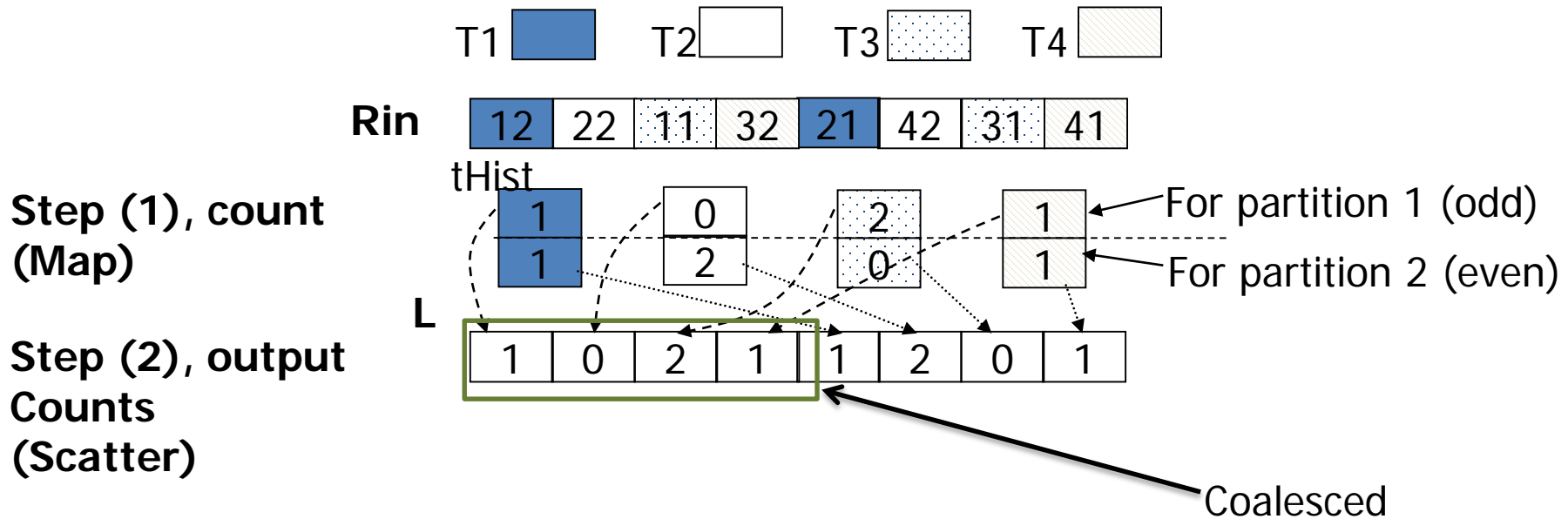
- A lock-free algorithm
 - Each thread is responsible for a portion of the input relation.
 - Each thread computes its local histogram (number of tuples in each output partition).
 - Given the local histograms, each thread computes its write locations.
 - Each thread writes the tuples to the output relation in parallel.

An Example of Split

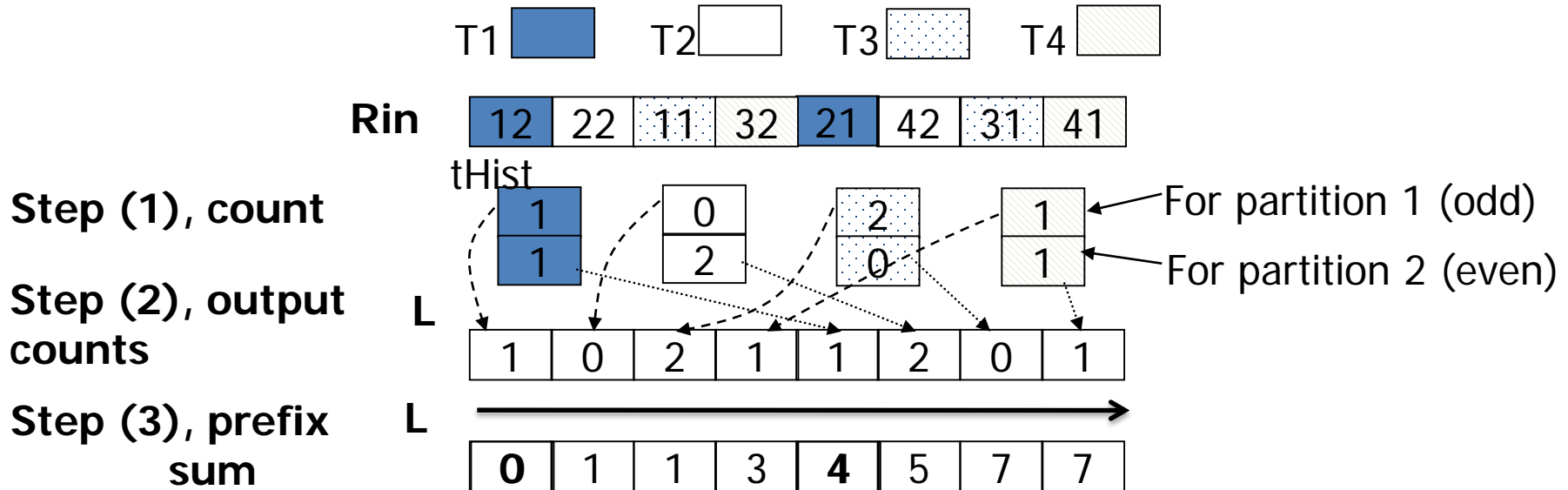
Split ($R_{in}[1, \dots, 8]$, fcn , $R_{out}[1, \dots, 8]$), $fcn(x) = x \bmod 2$



An Example of Split



An Example of Split



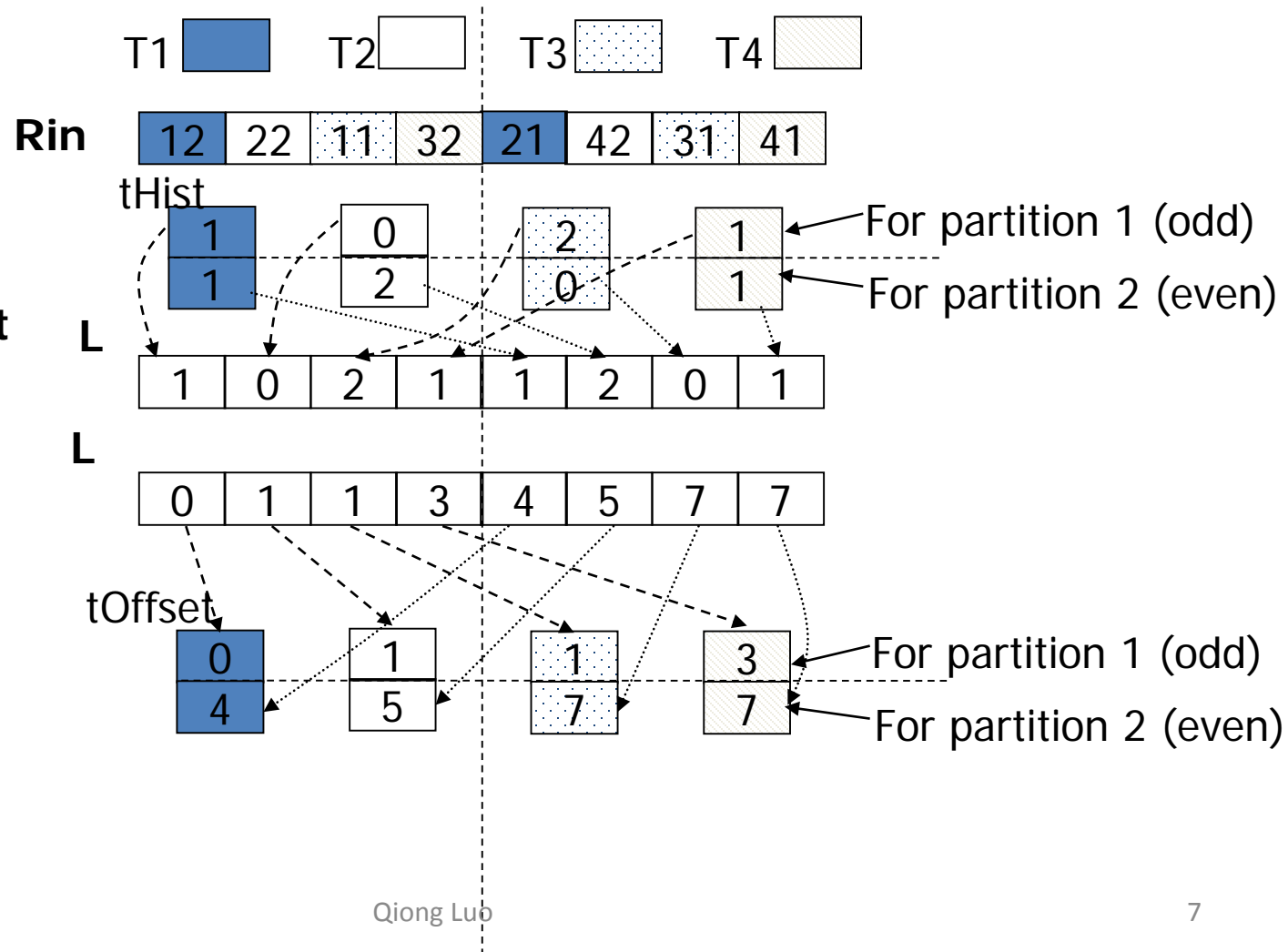
An Example of Split

Step (1), count

Step (2), output counts

Step (3), prefix sum

Step (4), load Counts (Gather)



An Example of Split

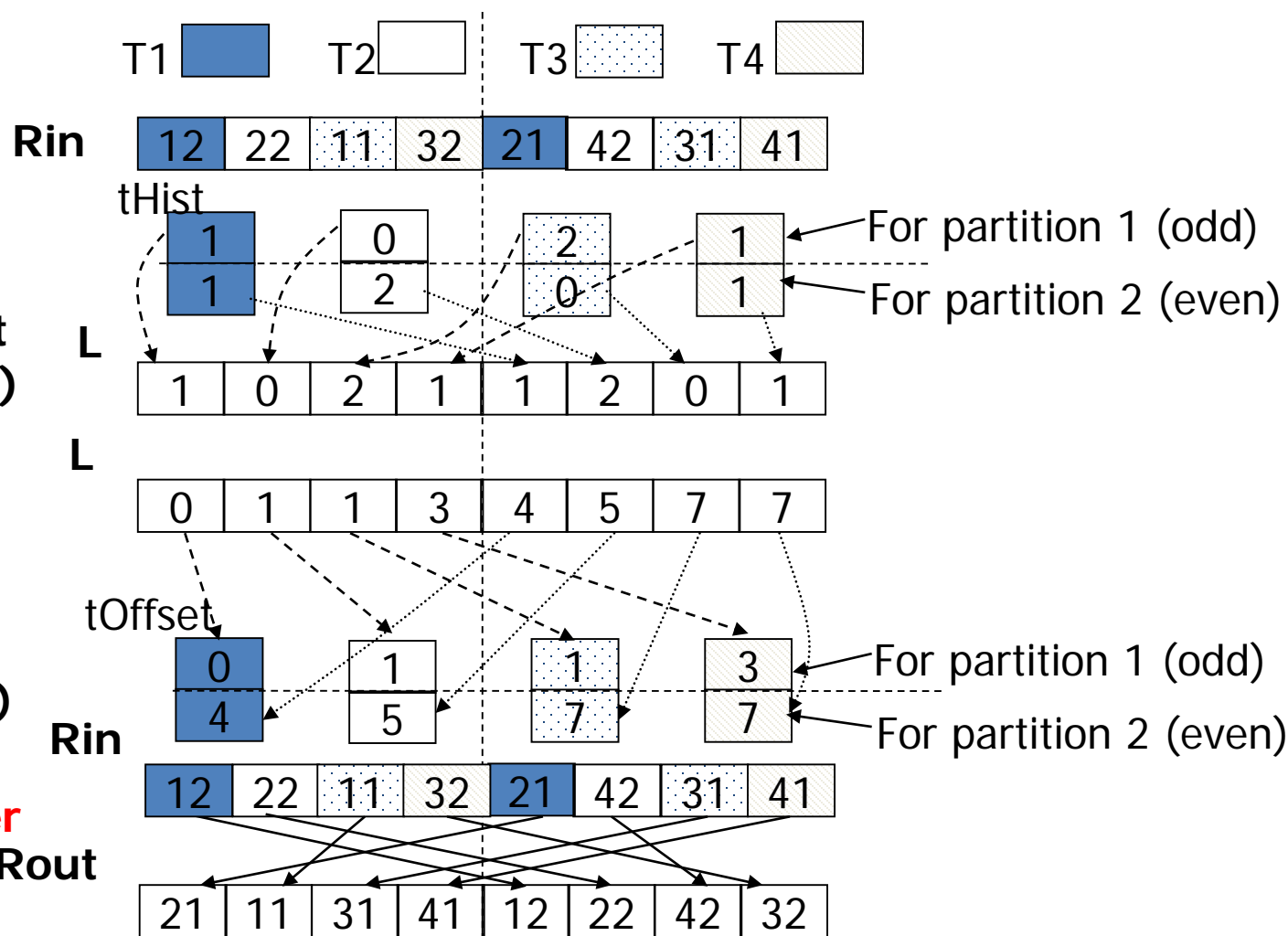
Step (1), count
(**Map**)

Step (2), output
Counts (**Scatter**)

Step (3), **prefix**
sum

Step (4), load
Counts (**Gather**)

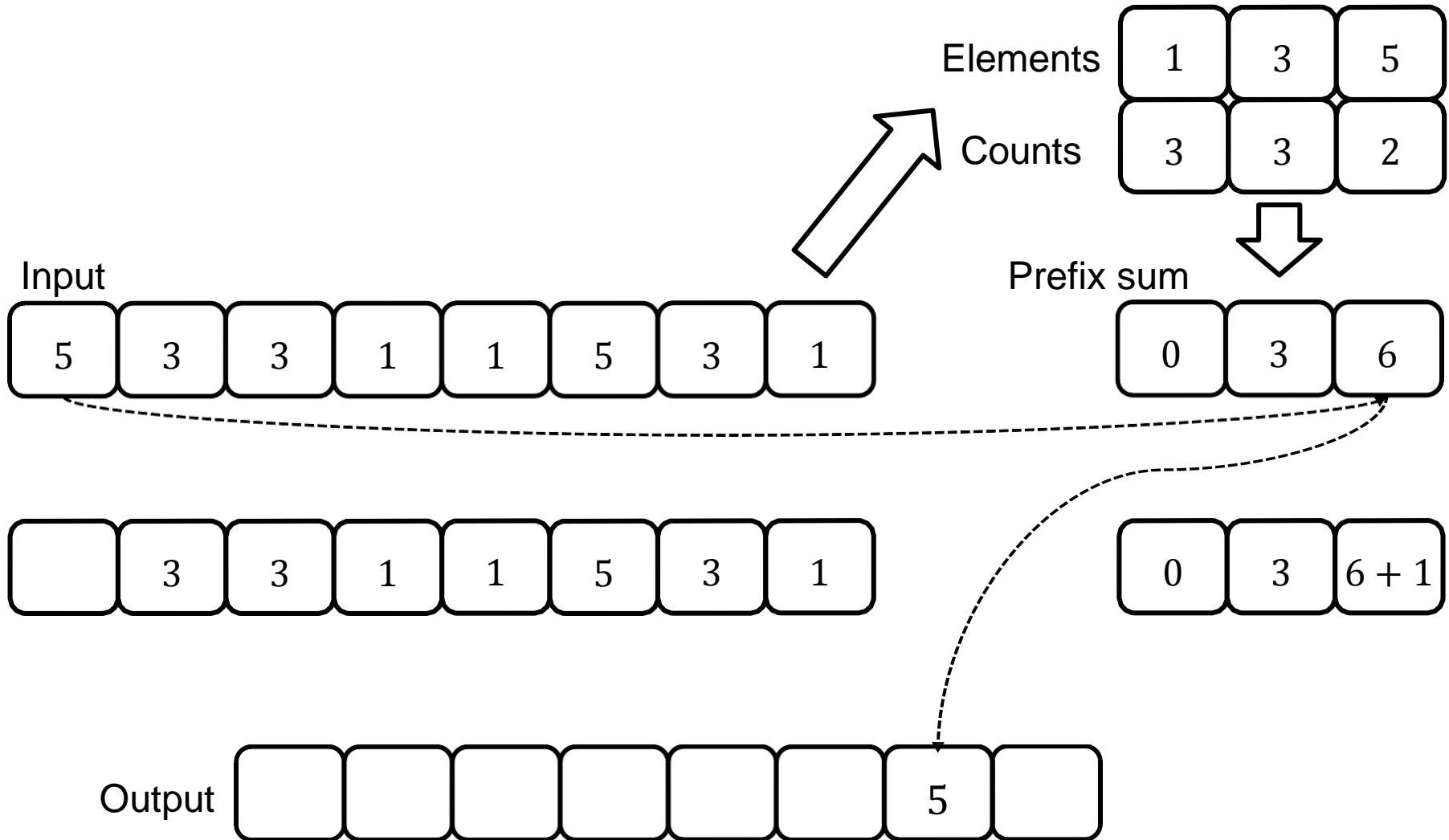
Step (5), **scatter**



Counting Sort Algorithm

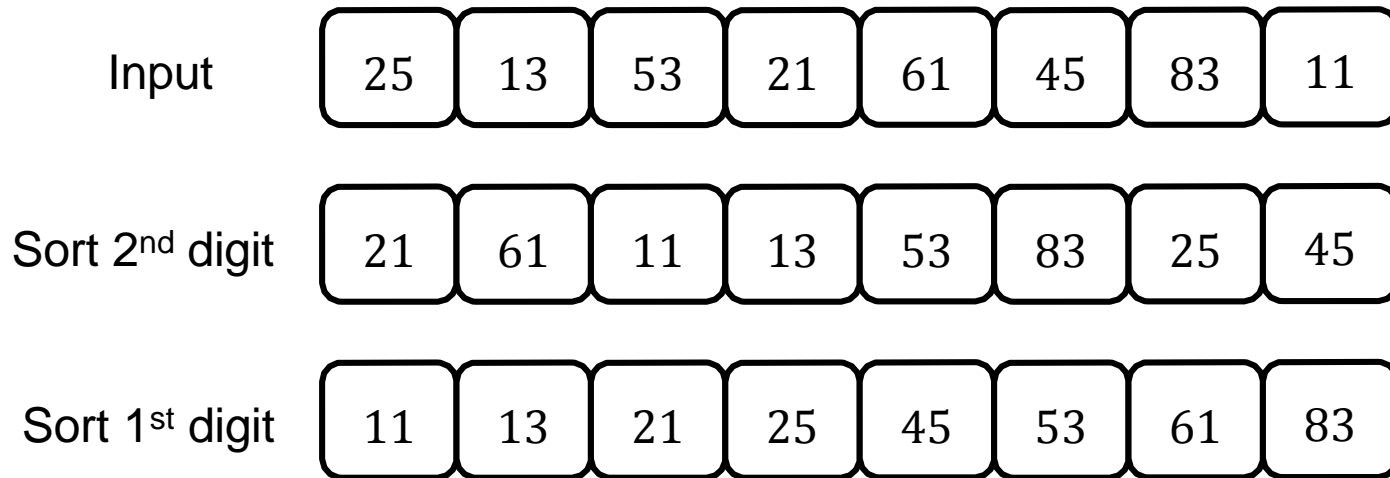
- Sorting for equality-comparison elements, e.g., integers
- Assume
 - Constant number of possible values
 - Possible values known in advance
- Algorithm outline
 - Compute the histogram of each element
 - Prefix sum over the histogram
 - Move each element to its location
- Complexity (sequential) - $O(n)$

Counting Sort Example



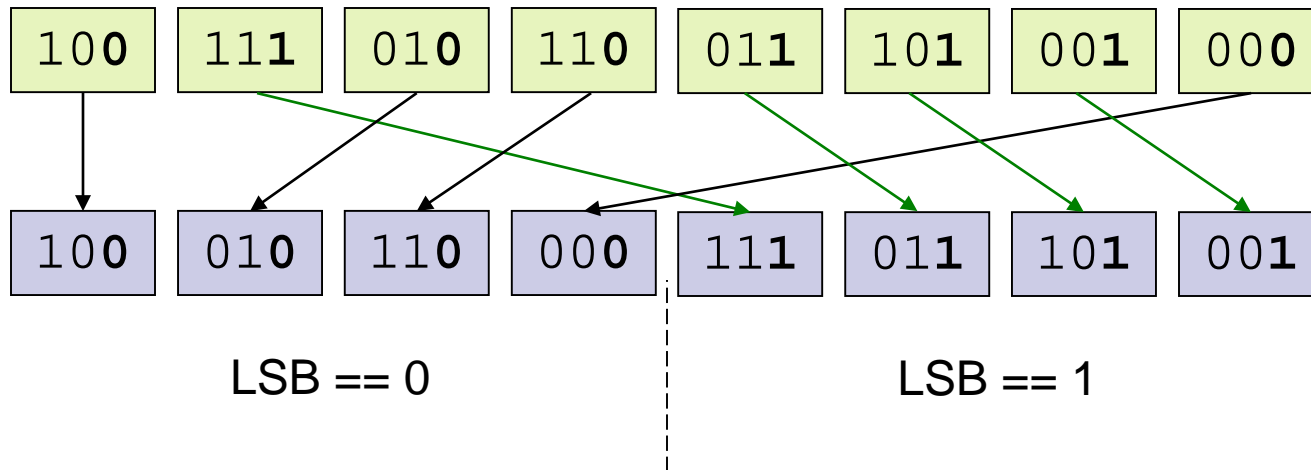
Radix Sort

- Iterated counting sort on individual digits (radix)
- Sort from the least significant bit (LSB) to the most significant bit (MSB)



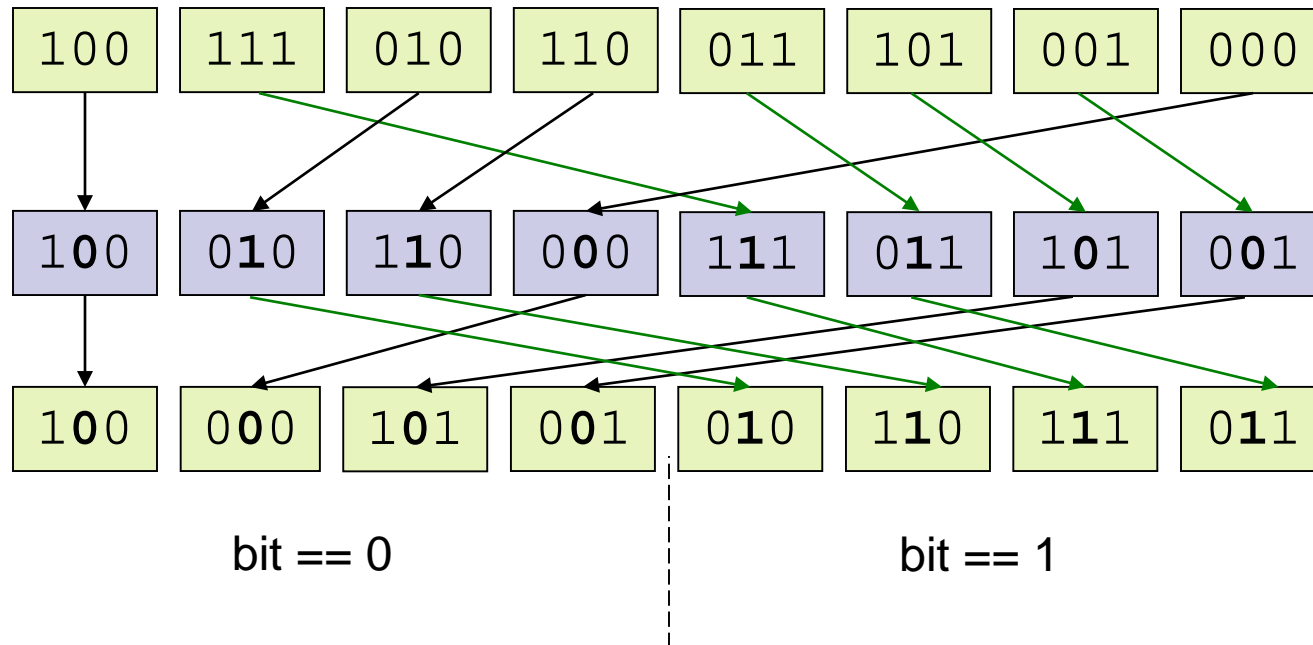
Radix Sort Example

- First pass: partition based on LSB



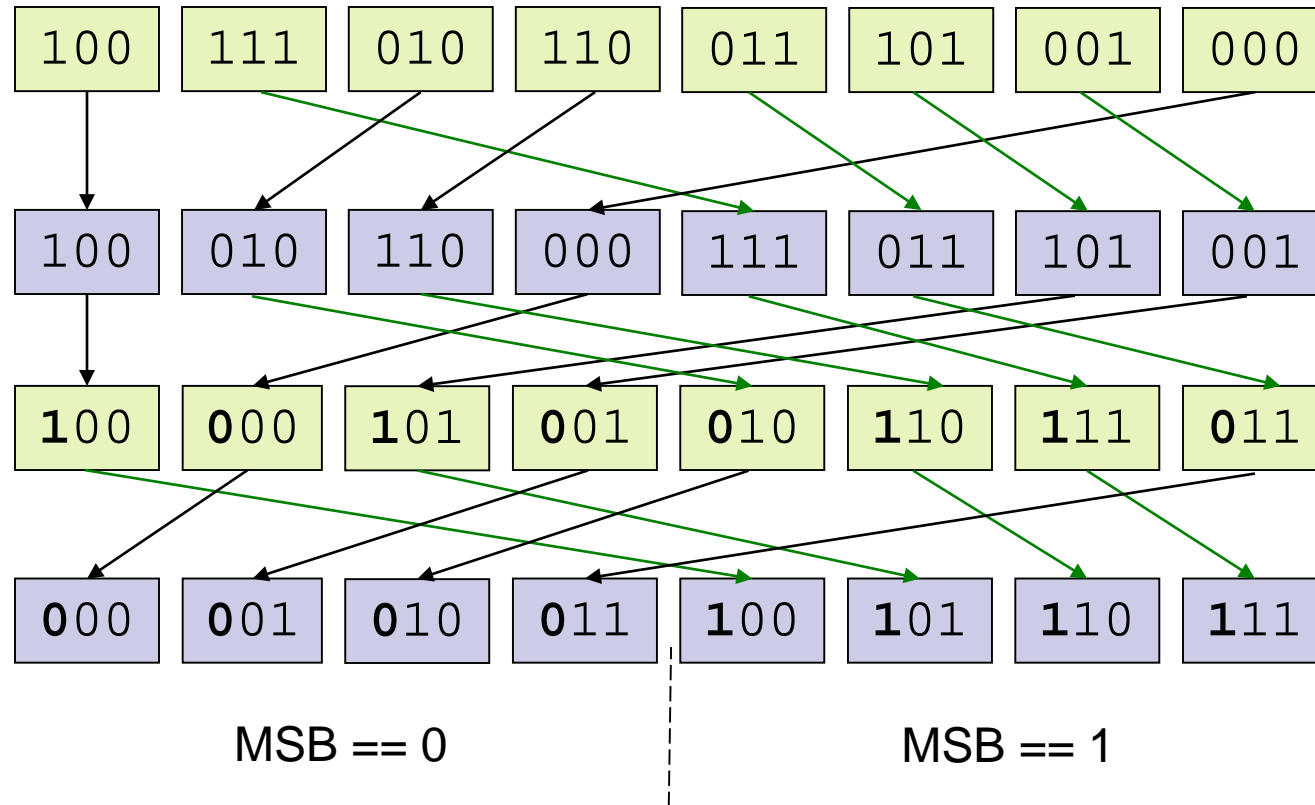
Radix Sort Example

- Second pass: partition based on second LSB



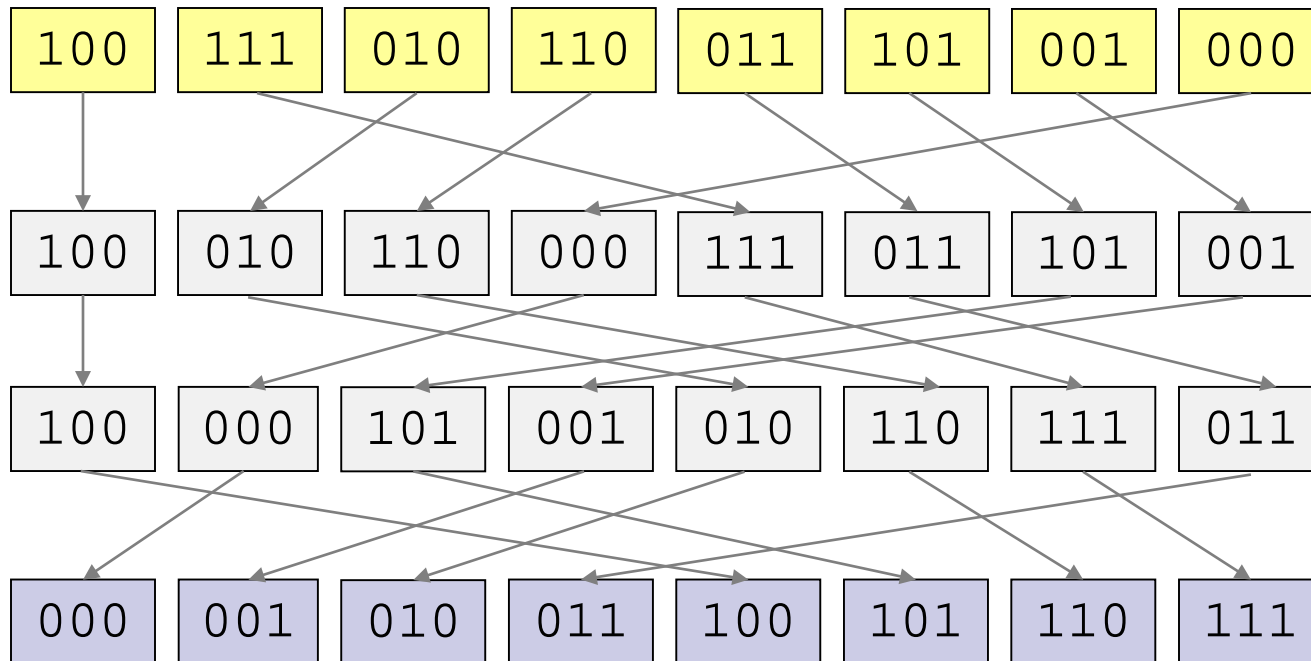
Radix Sort Example

- Final pass: partition based on MSB



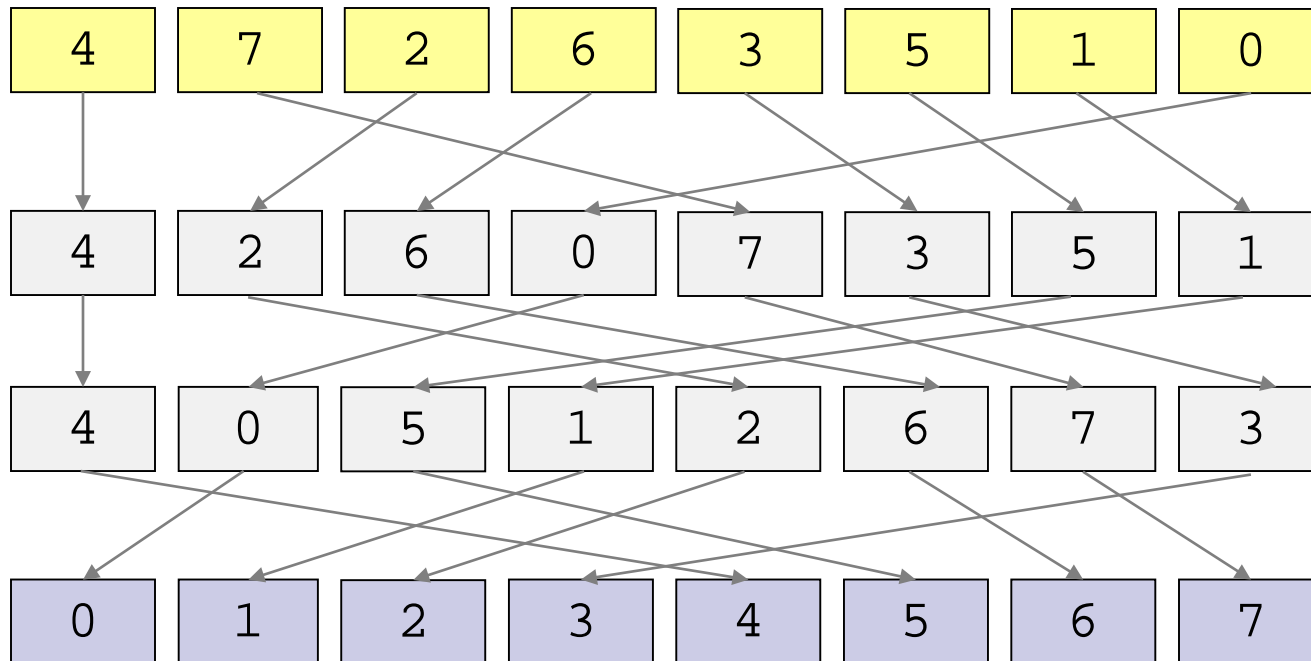
Radix Sort Example

- Completed:



Radix Sort Example

- Completed:



Parallel Radix Sort

1. Break input array into tiles
 - Each tile fits into shared memory for a thread block
2. Sort tiles in *parallel* with *radix sort*
3. Merge pairs of tiles using a *parallel bitonic merge* until all tiles are merged.

Our focus is on Step 2

Parallel Radix Sort

- Where is the parallelism?
 - Each tile is sorted in parallel
 - Where is the parallelism within a tile?
 - Each pass is done in sequence after the previous pass.
No parallelism
 - Can we parallelize an individual pass? How?
 - Merge also has parallelism

Parallel Radix Sort in Each Pass

- Implement Split on the current bit under comparison
 - Histogram-based computation to count the number of 0/1s
 - Prefix scan to determine the output position of each element.
 - Efficient scatter of elements to target locations
 - Shared memory optimization
 - Histograms are stored in the shared memory.

Parallel Radix Sort

- Implement *split*. Given:

- Array, *i*, at pass *n*:

100	111	010	110	011	101	001	000
-----	-----	-----	-----	-----	-----	-----	-----

- Array, *b*, which is true/false for bit *n*:

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

- Output array with false keys before true keys:

100	010	110	000	111	011	101	001
-----	-----	-----	-----	-----	-----	-----	-----

Parallel Radix Sort

- Step 1: Compute *e* array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array

Parallel Radix Sort

- Step 2: Exclusive Scan *e*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array

Parallel Radix Sort

- Step 3: Compute *totalFalses*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array

$$\begin{aligned}\text{totalFalses} &= e[n-1] + f[n-1] \\ \text{totalFalses} &= 1 + 3 \\ \text{totalFalses} &= 4\end{aligned}$$

Parallel Radix Sort

- Step 4: Compute t array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
								t array

$$t[i] = i - f[i] + \text{totalFalses}$$

totalFalses = 4

Parallel Radix Sort

- Step 4: Compute t array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4								t array

$$t[0] = 0 - f[0] + \text{totalFalses}$$

$$t[0] = 0 - 0 + 4$$

$$t[0] = 4$$

$$\text{totalFalses} = 4$$

Parallel Radix Sort

- Step 4: Compute t array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4							t array

$$t[1] = 1 - f[1] + \text{totalFalses}$$

$$t[1] = 1 - 1 + 4$$

$$t[1] = 4$$

$$\text{totalFalses} = 4$$

Parallel Radix Sort

- Step 4: Compute t array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5						t array

$$t[2] = 2 - f[2] + \text{totalFalses}$$

$$t[2] = 2 - 1 + 4$$

$$t[2] = 5$$

$$\text{totalFalses} = 4$$

Parallel Radix Sort

- Step 4: Compute t array

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array

$$t[i] = i - f[i] + \text{totalFalses}$$

totalFalses = 4

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0								$d[i] = b[i] ? t[i] : f[i]$

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0	4							$d[i] = b[i] ? t[i] : f[i]$

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0	4	1						$d[i] = b[i] ? t[i] : f[i]$

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
<hr/>								
1	0	1	1	0	0	0	1	e array
0	1	1	2	3	3	3	3	f array
4	4	5	5	5	6	7	8	t array
0	4	1	2	5	6	7	3	$d[i] = b[i] ? t[i] : f[i]$

Parallel Radix Sort

- Step 5: Scatter based on address *d*

100	111	010	110	011	101	001	000
-----	-----	-----	-----	-----	-----	-----	-----

i array

0	4	1	2	5	6	7	3
---	---	---	---	---	---	---	---

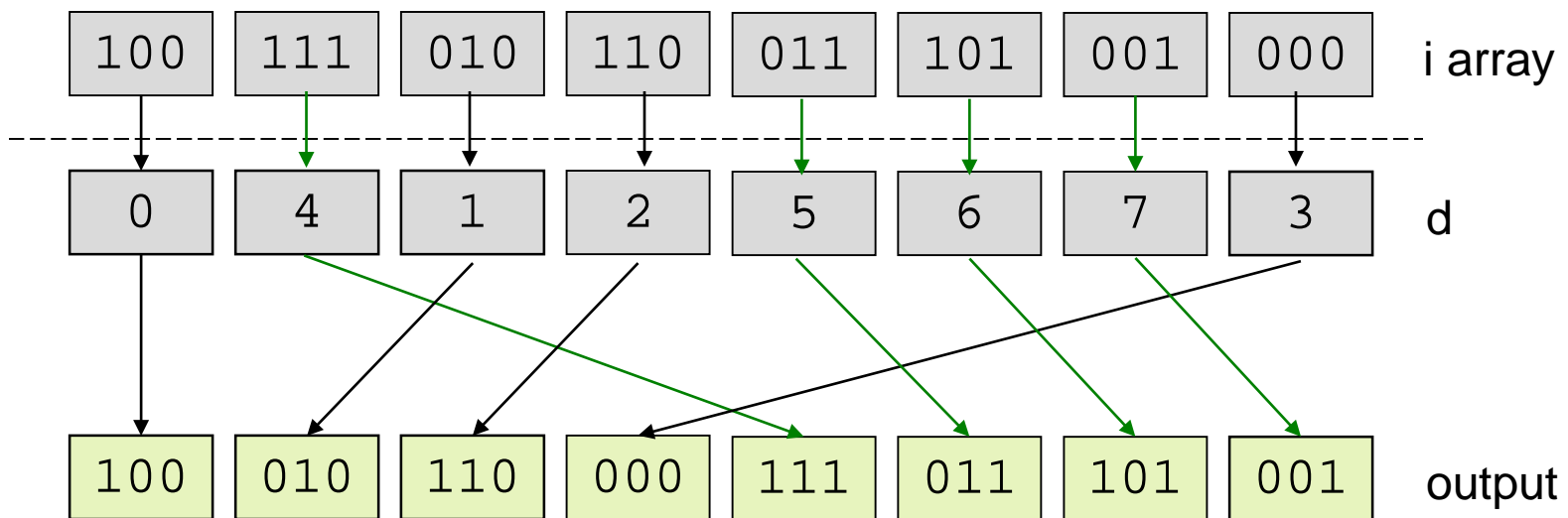
d

--	--	--	--	--	--	--	--

output

Parallel Radix Sort

- Step 5: Scatter based on address *d*



Radix Sort Analysis

- Integer sort
- Complexity $O(kn)$
 - n – number of elements
 - k – number of digits (constant)
- Parallel implementation (naïve)
 - For each radix (digit)
 - Compute radix histogram
 - Scan the histogram to compute offset for each element
 - Write the sorted elements
 - Work $O(kn)$
 - Time $O(k \log n)$

Summary

- Split and Sort are two common data-parallel primitives.
- They can be composed using simpler primitives.
- They are used widely in higher-level applications.
- Radix sort is currently the fastest sorting algorithm on the GPU.