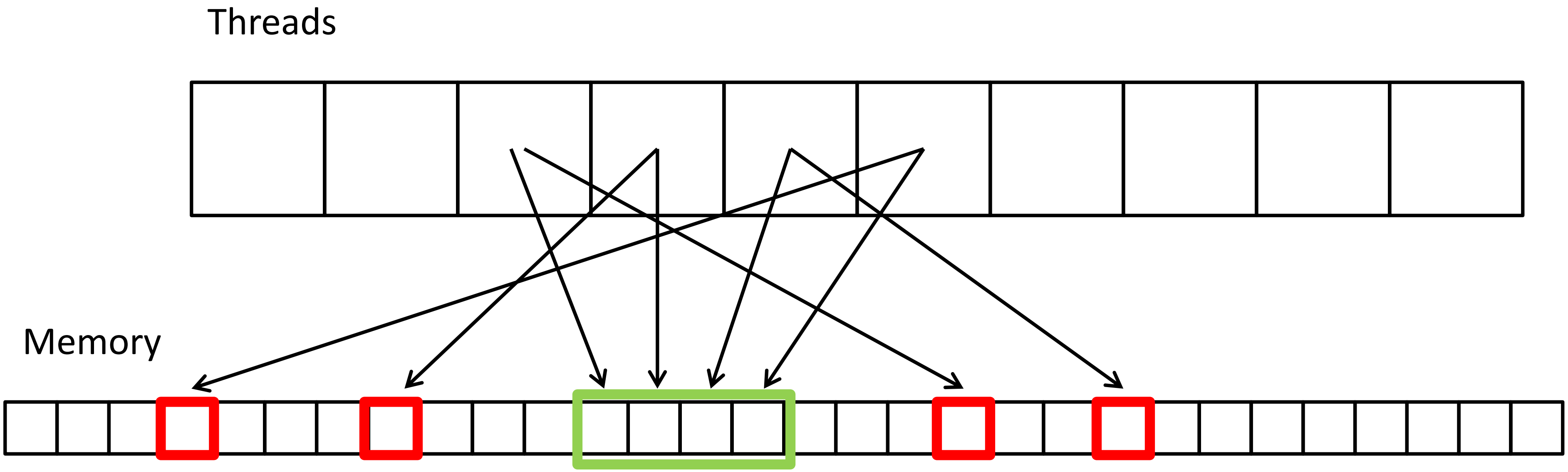
Introduction to data-parallelism
and OpenCL

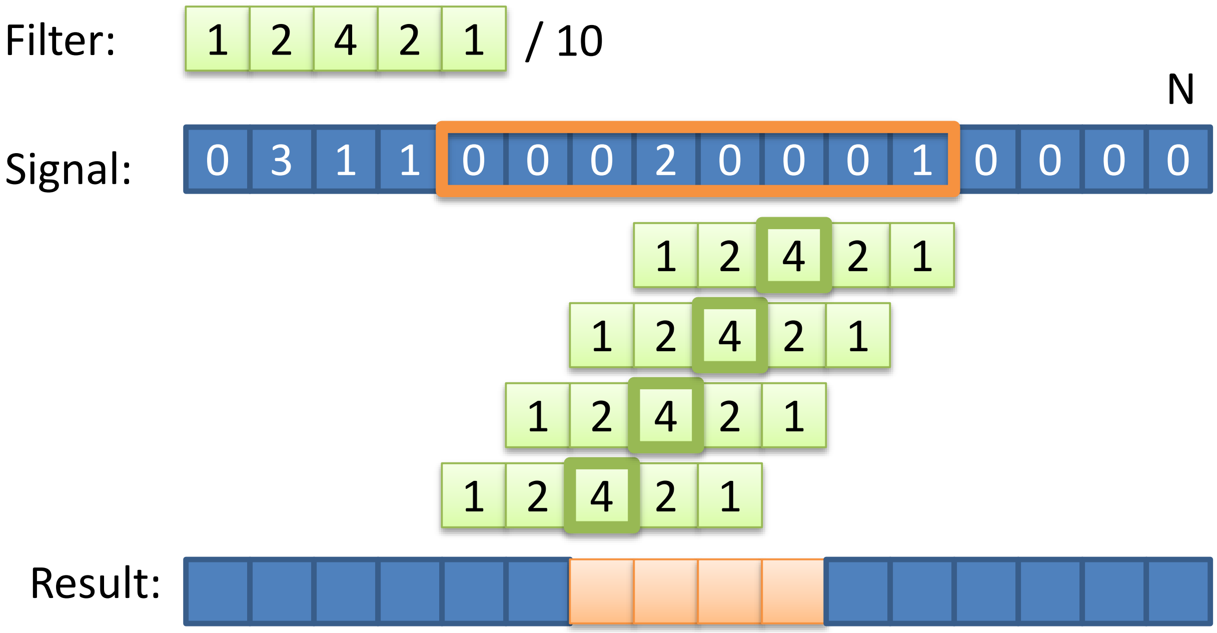
Sylvain Lefebvre - INRIA

Quick reminders

Coalesced accesses



Pre-fetch



Filter: 1 2 4 2 1 / 10

N

Signal: 0 3 1 1 0 0 0 2 0 0 0 1 0 0 0 0

1 2 4 2 1

1 2 4 2 1

1 2 4 2 1

1 2 4 2 1

Result:

Synchronization

- barrier
  - CLK_**LOCAL**_MEM_FANCE
  - CLK_**GLOBAL**_MEM_FANCE

- Atomics
  - inc/dec, add/sub, min/max, xchg/cmpxchg

Today

- Parallel Reduction

- Parallel Scan

- Applications

Parallel reduction

- Ex: Sum all entries of an array

- Sum-of-all is a special case of a *reduction*

- 'Add' could be replaced by
  - min / max / and / or / etc.

- Very important compute primitive
  - People have been working hard to optimize it

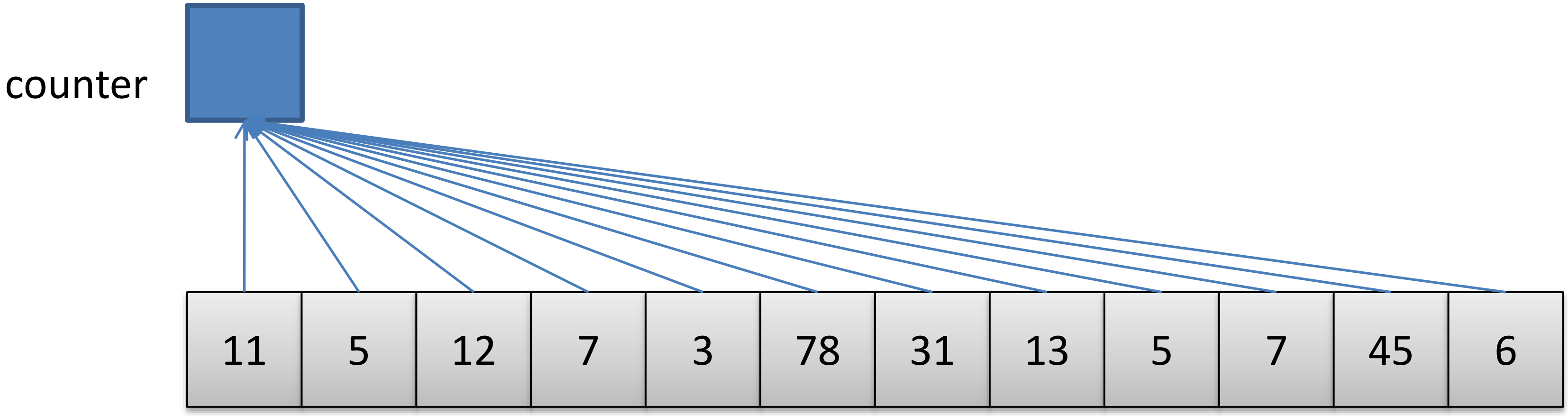
Reduction

- Assume ‘sum’ as operator

- First approach:
  - Atomic add

- Second approach:
  - Hierarchical sum

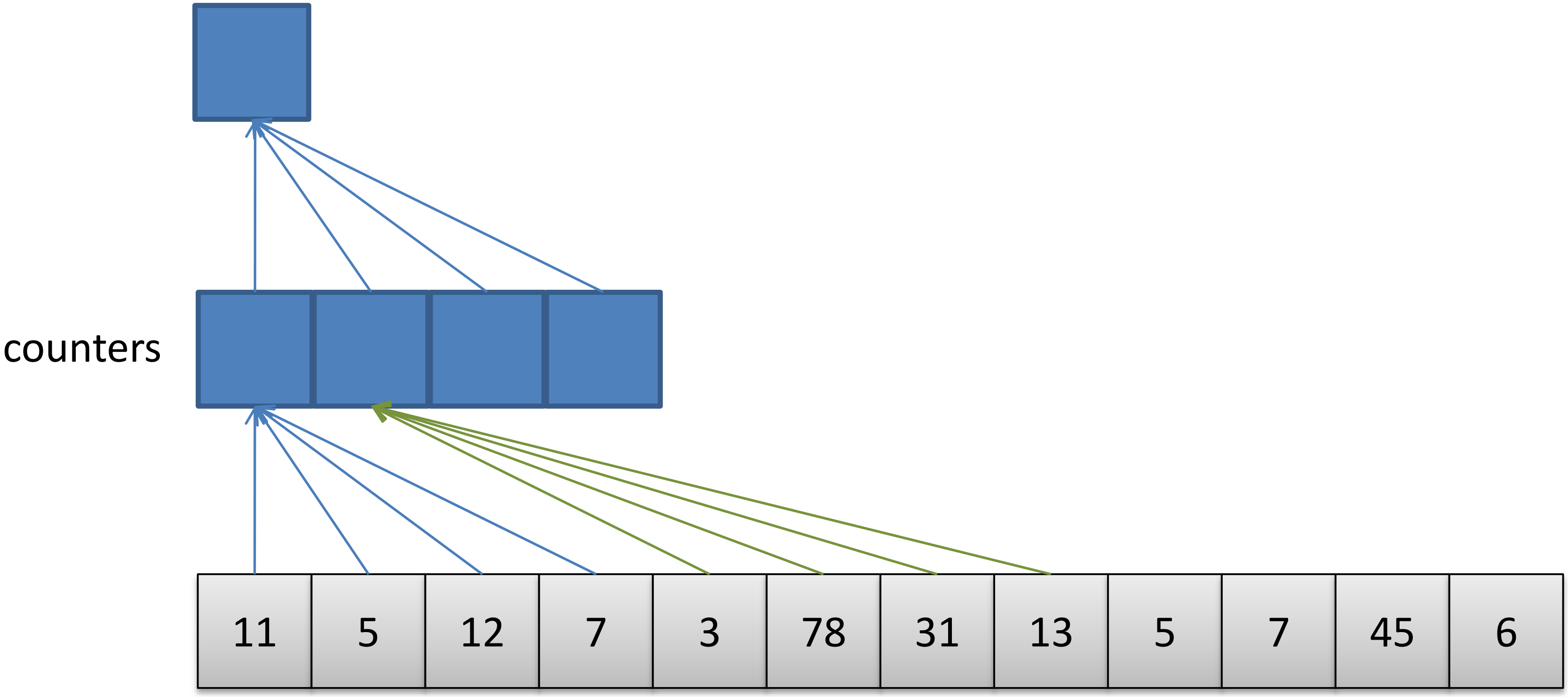
atomicAdd



counter

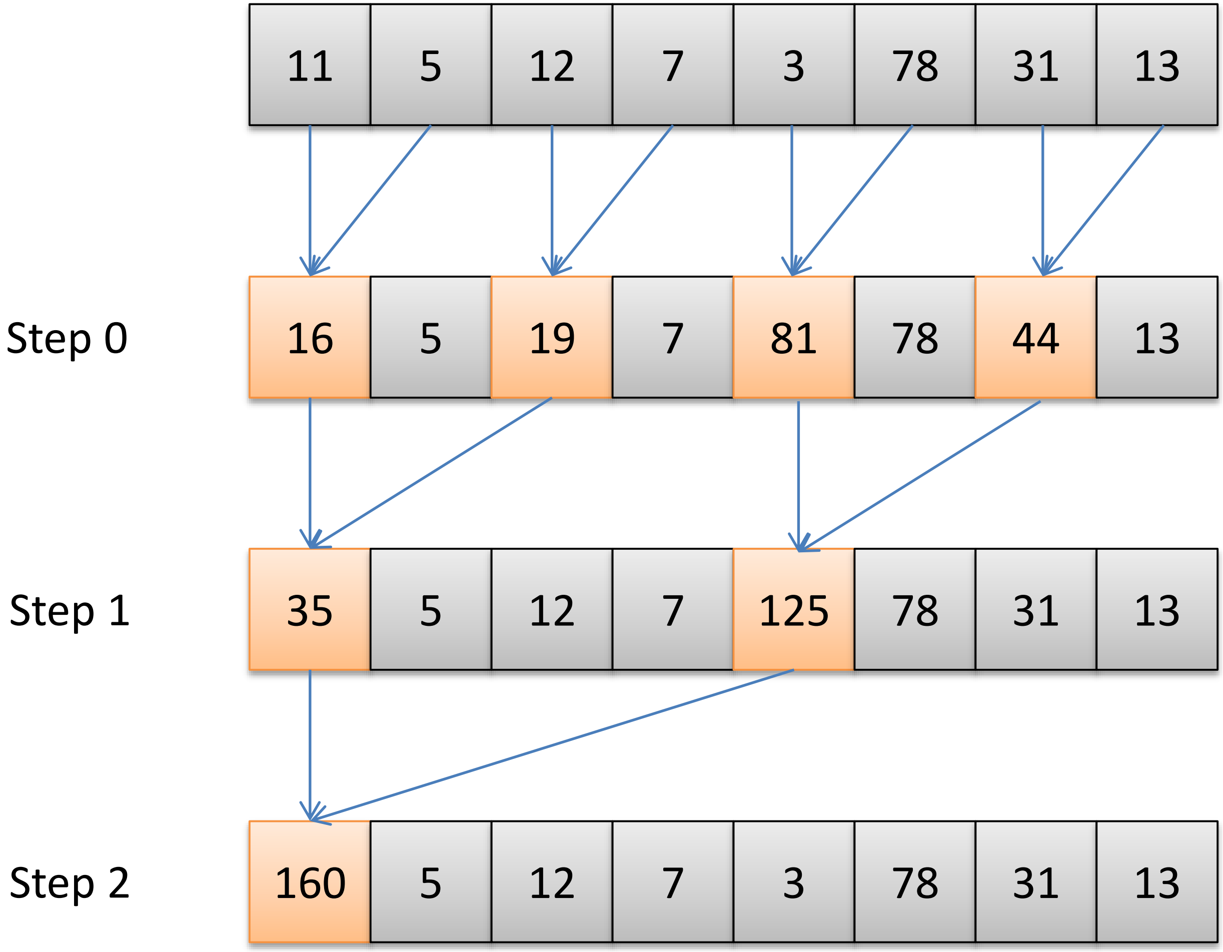
11	5	12	7	3	78	31	13	5	7	45	6

atomicAdd (2)



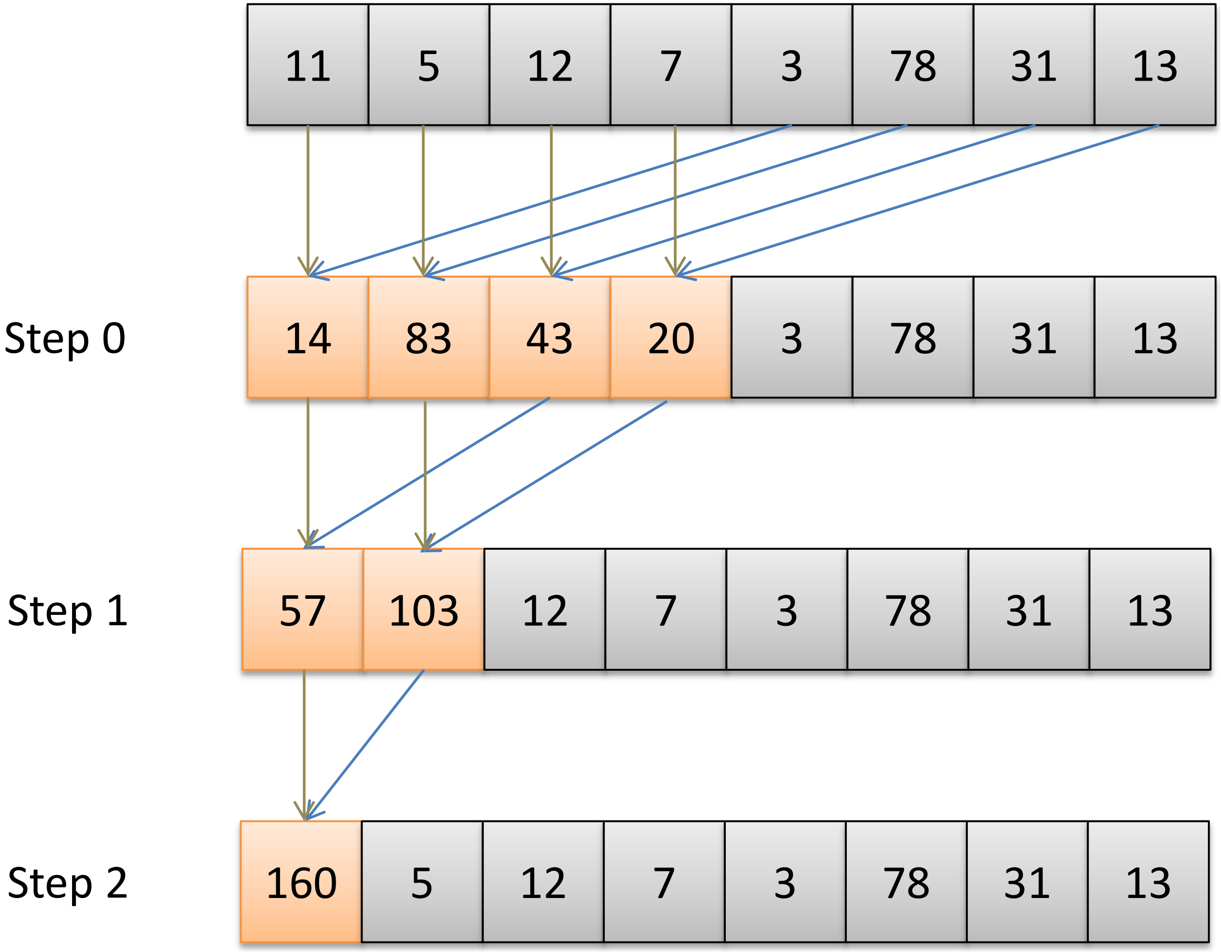
11	5	12	7	3	78	31	13	5	7	45	6

Hierarchy



	11	5	12	7	3	78	31	13
Step 0	16	5	19	7	81	78	44	13
Step 1	35	5	12	7	125	78	31	13
Step 2	160	5	12	7	3	78	31	13

Better



	11	5	12	7	3	78	31	13
Step 0	14	83	43	20	3	78	31	13
Step 1	57	103	12	7	3	78	31	13
Step 2	160	5	12	7	3	78	31	13

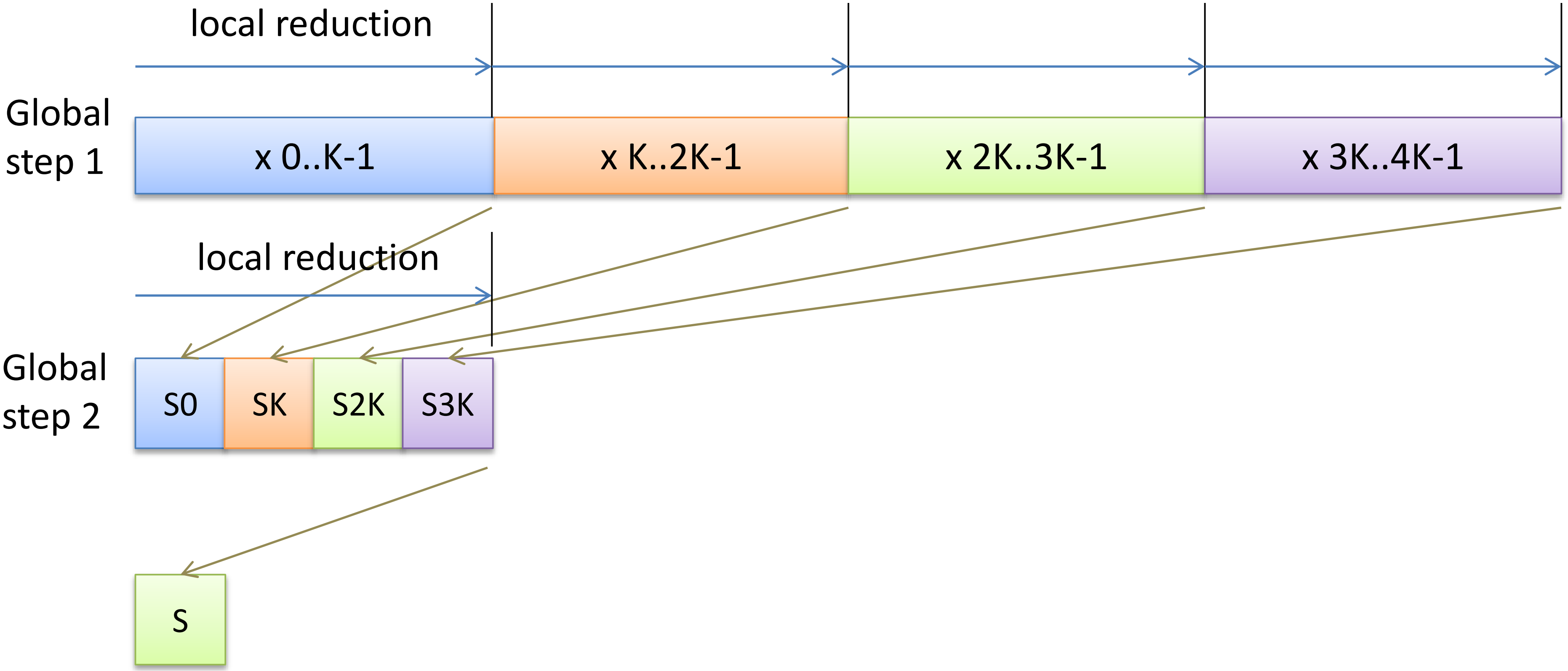
Local vs. global

- Local is faster

- Limited due to synchronization

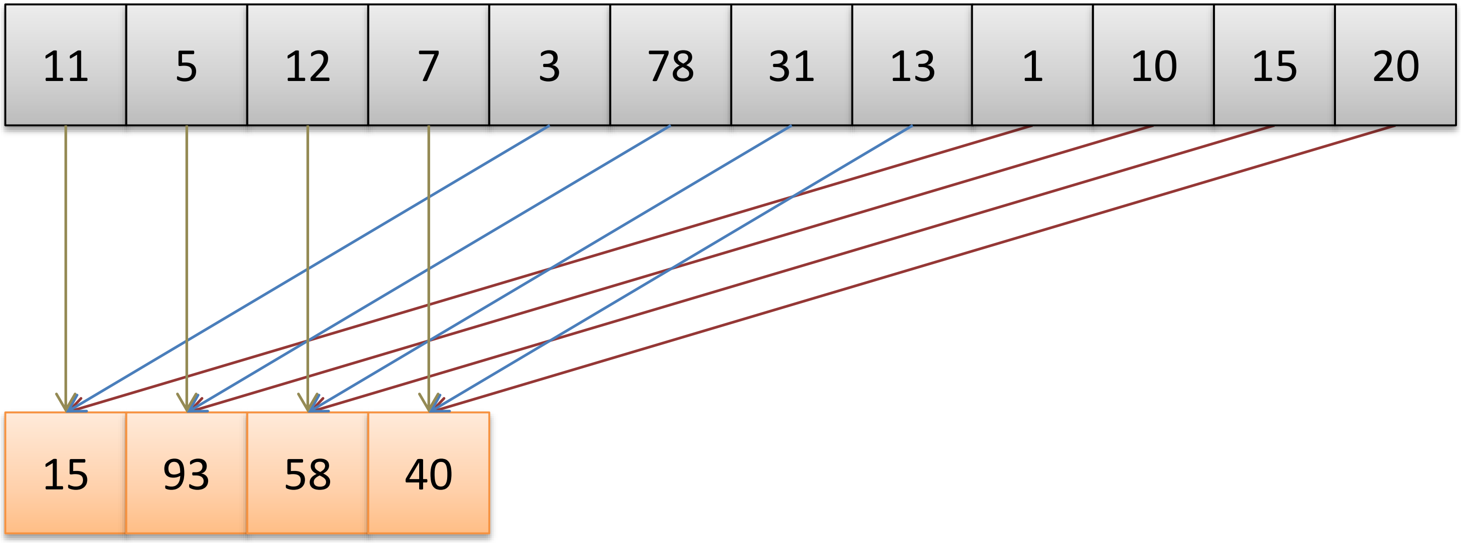
➔ Do local reduction on sub-arrays

Use local to minimize global



Increase per-thread work

- Small sequential sum in each thread
  - Further reduces global calls
  - More opportunities for latency hiding
- Start with a small loop
  - Careful with coalescence



11	5	12	7	3	78	31	13	1	10	15	20

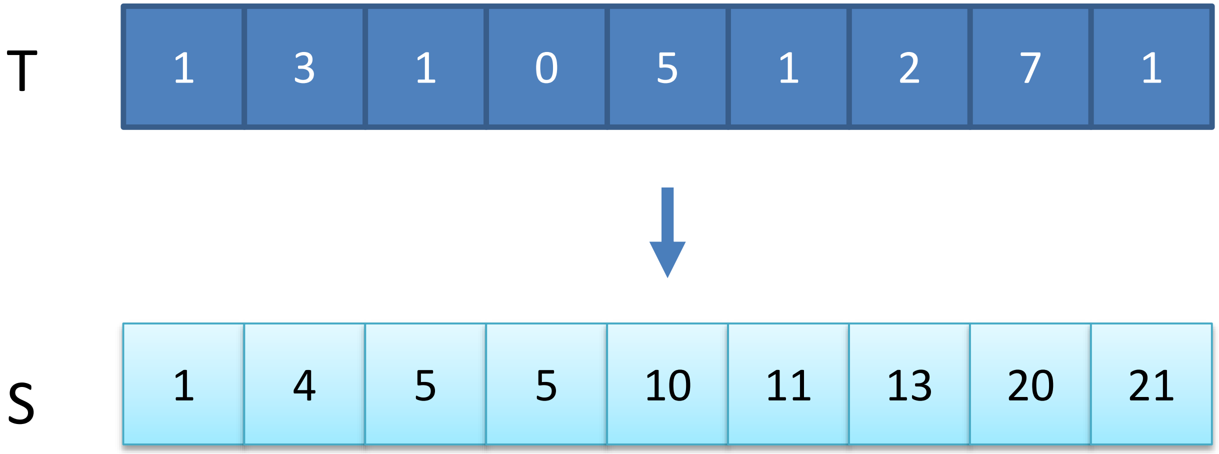
15	93	58	40

Parallel scan

Parallel scan

- *Scan* is also a very common operation

- Also known as prefix sum:



T	1	3	1	0	5	1	2	7	1
S	1	4	5	5	10	11	13	20	21

$$S[i] = \sum_{j<=i} T[j]$$

- Usually performed in-place

Sequential scan

- Very simple and efficient on a CPU:

```
for i : 1 .. N
        T[i] = T[i-1] + T[i]
```

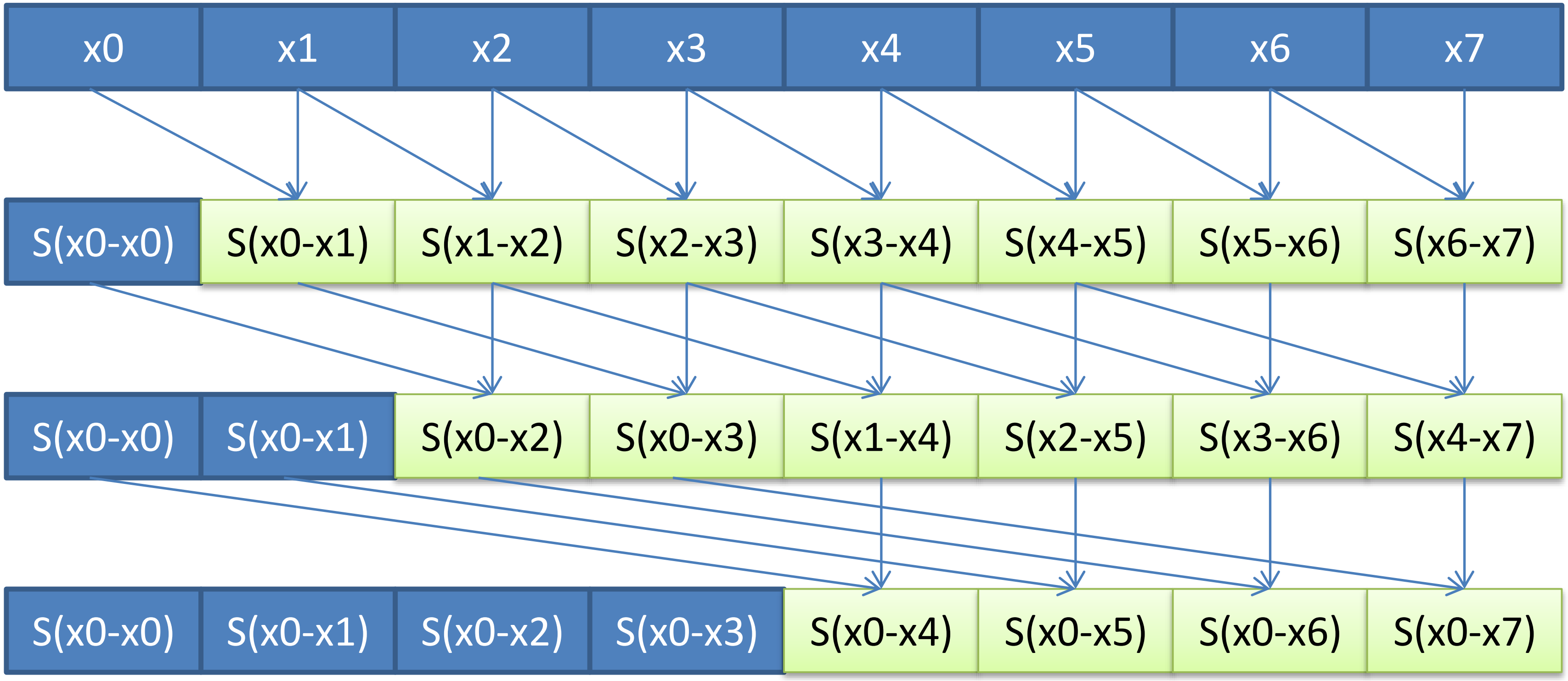
- That’s all!
- O(N) additions

Parallel scan

- Need to remove sequential dependency…

Parallel scan

- A first approach



x0	x1	x2	x3	x4	x5	x6	x7
S(x0-x0)	S(x0-x1)	S(x1-x2)	S(x2-x3)	S(x3-x4)	S(x4-x5)	S(x5-x6)	S(x6-x7)
S(x0-x0)	S(x0-x1)	S(x0-x2)	S(x0-x3)	S(x1-x4)	S(x2-x5)	S(x3-x6)	S(x4-x7)
S(x0-x0)	S(x0-x1)	S(x0-x2)	S(x0-x3)	S(x0-x4)	S(x0-x5)	S(x0-x6)	S(x0-x7)

- Number of additions?

O(N log N)

Parallel scan

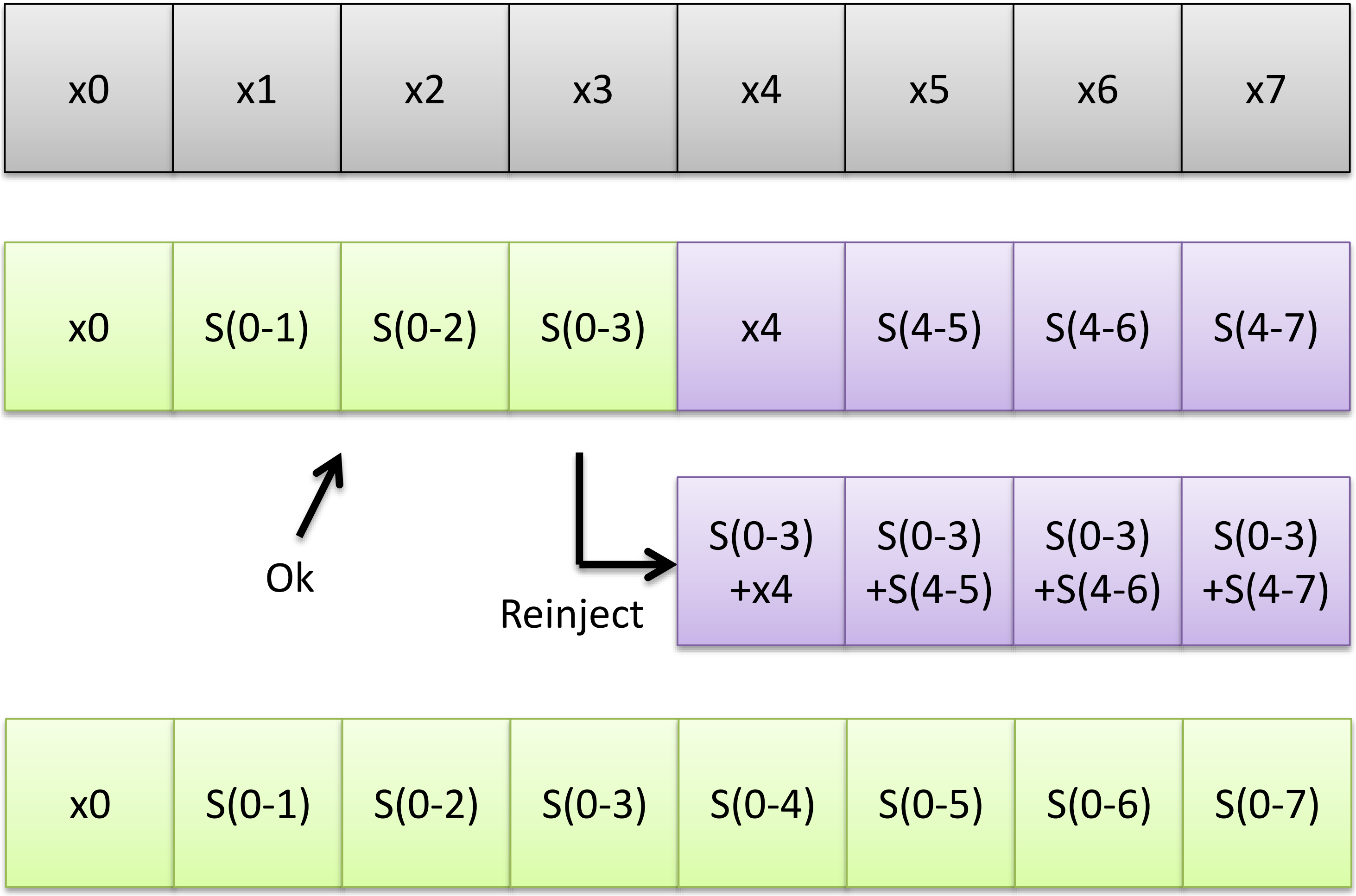
- First version:

O(N log N) vs. O(N) in sequential

- Not *work-efficient*

Parallel scan (2)

- Main idea:

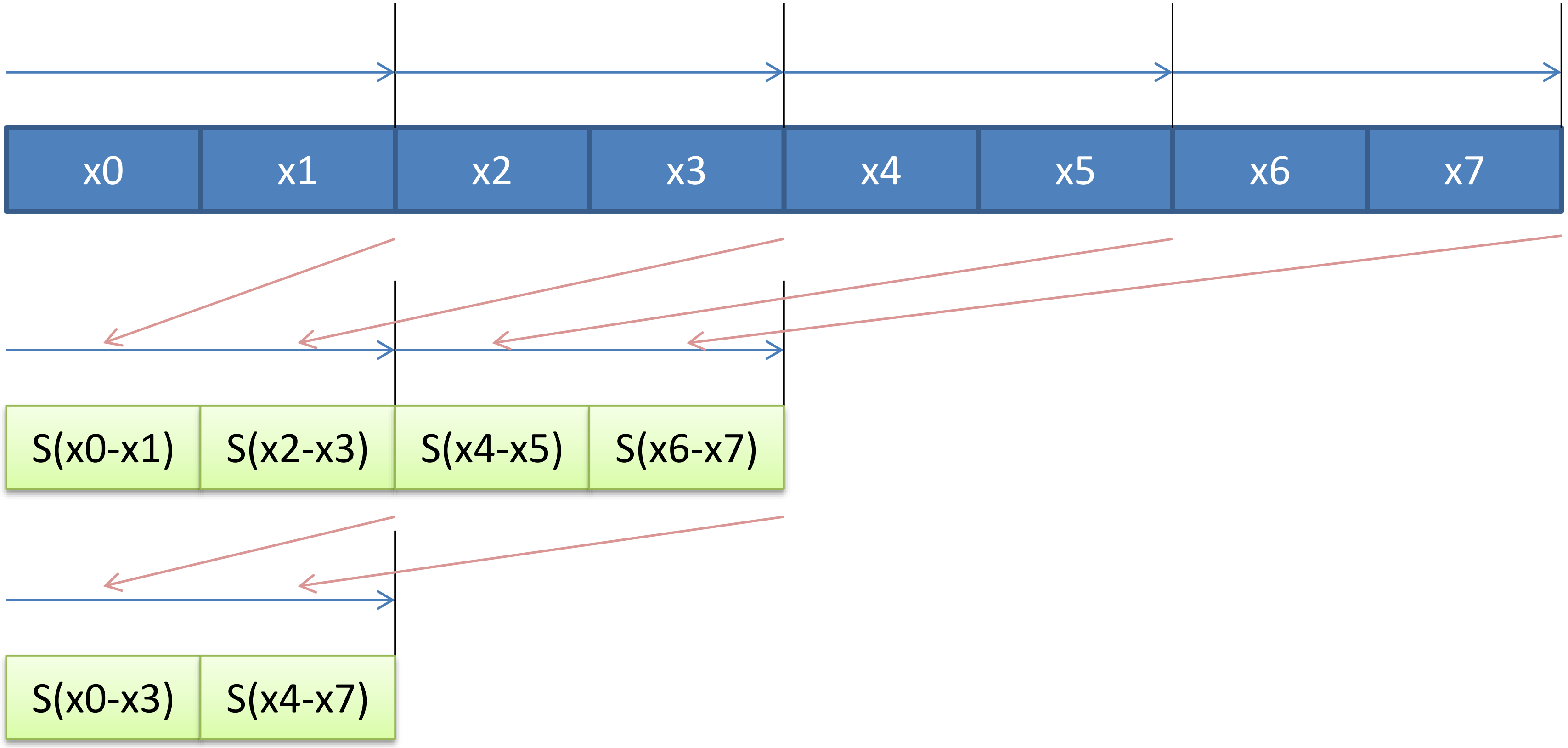


x0	x1	x2	x3	x4	x5	x6	x7
x0	S(0-1)	S(0-2)	S(0-3)	x4	S(4-5)	S(4-6)	S(4-7)

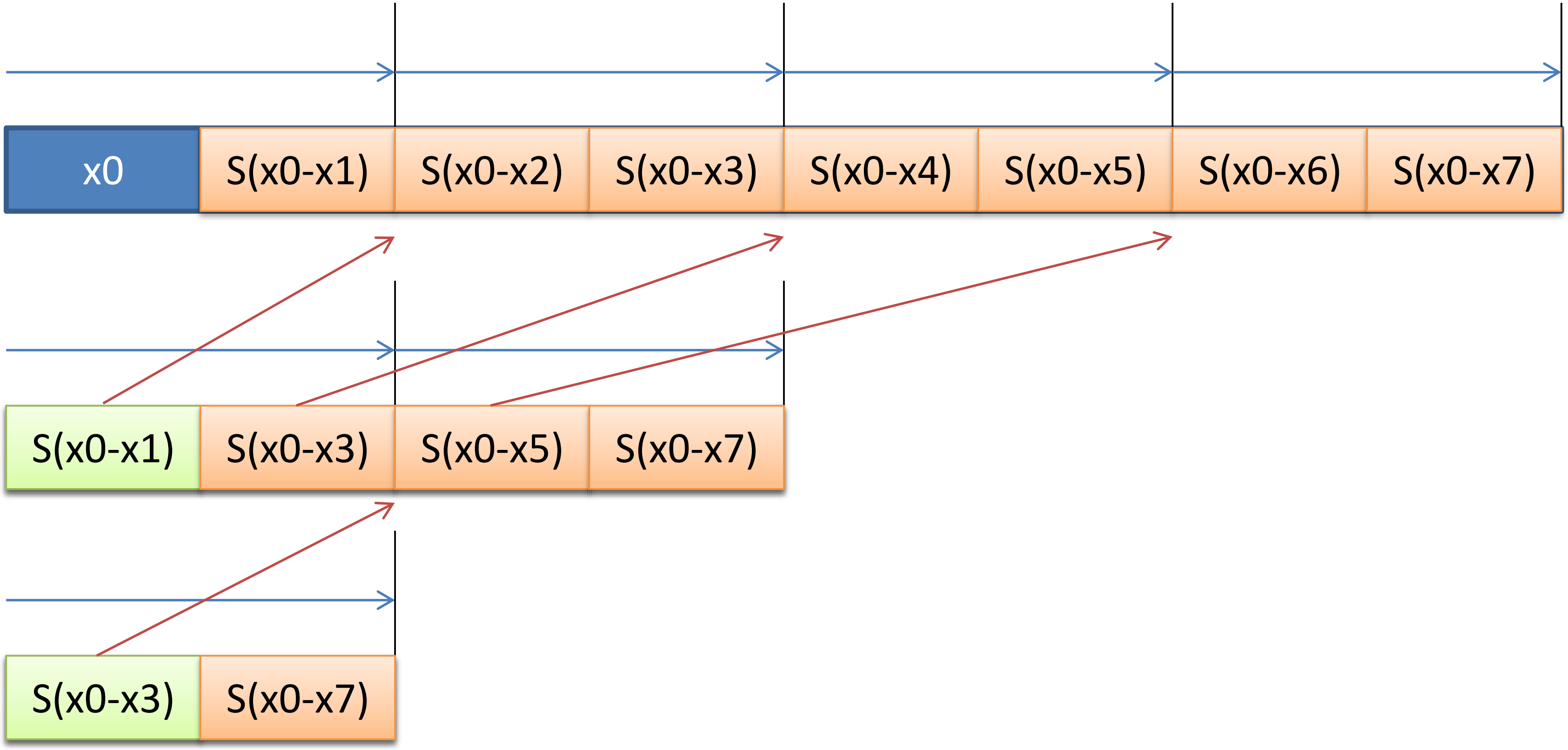
				S(0-3)
				+x4
					S(0-3)
					+S(4-5)
						S(0-3)
						+S(4-6)
							S(0-3)
							+S(4-7)

x0	S(0-1)	S(0-2)	S(0-3)	S(0-4)	S(0-5)	S(0-6)	S(0-7)

Parallel scan (2)



Parallel scan (2)



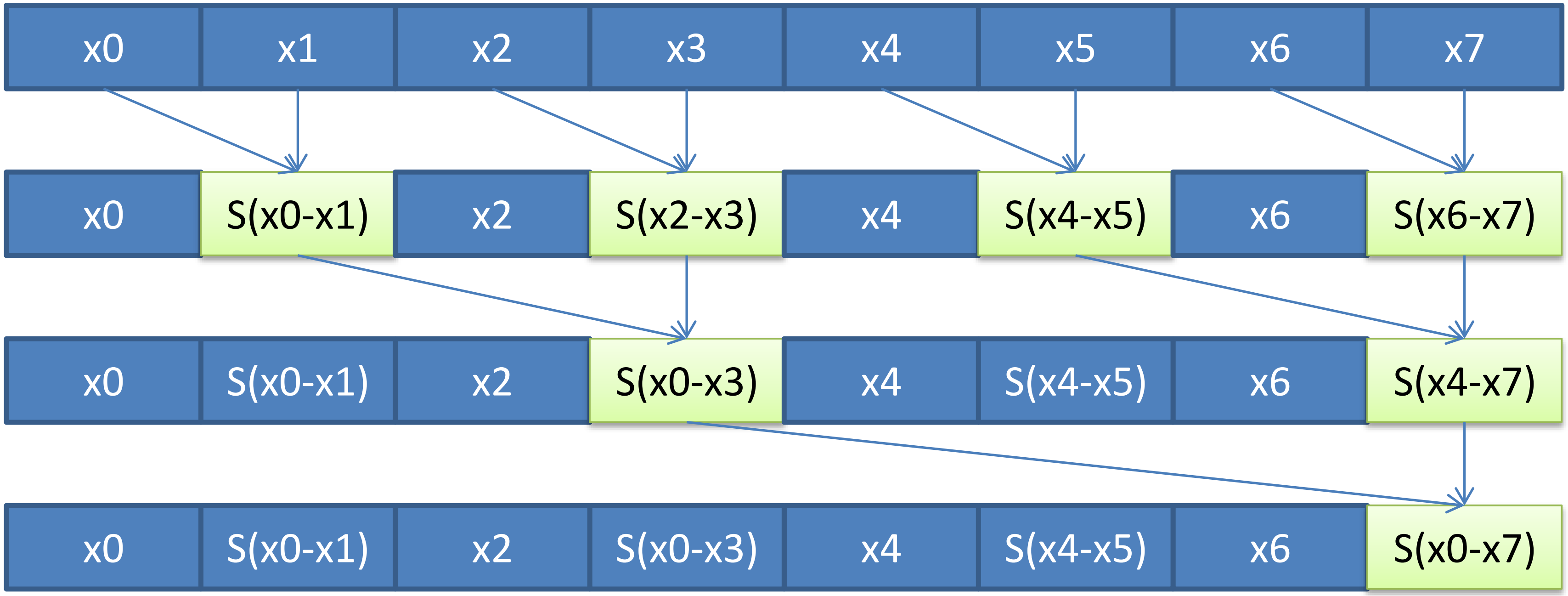
x0 S(x0-x1) S(x0-x2) S(x0-x3) S(x0-x4) S(x0-x5) S(x0-x6) S(x0-x7)

S(x0-x1) S(x0-x3) S(x0-x5) S(x0-x7)

S(x0-x3) S(x0-x7)

Parallel scan (2)

- Two phases:
  1. Perform a reduction (again!)

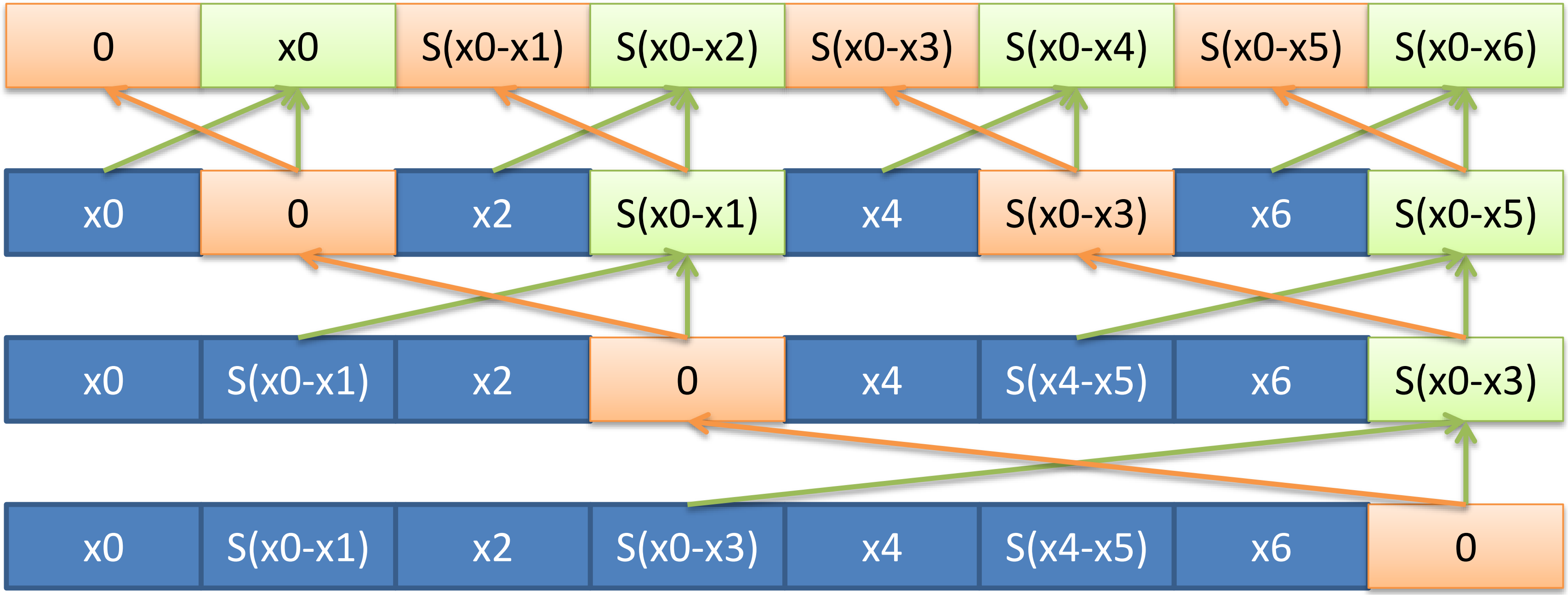


x0	x1	x2	x3	x4	x5	x6	x7
x0	S(x0-x1)	x2	S(x2-x3)	x4	S(x4-x5)	x6	S(x6-x7)
x0	S(x0-x1)	x2	S(x0-x3)	x4	S(x4-x5)	x6	S(x4-x7)
x0	S(x0-x1)	x2	S(x0-x3)	x4	S(x4-x5)	x6	S(x0-x7)

Parallel scan (2)

- Two phases:
  1. Perform a reduction (again!)
  2. Down-sweep

(slightly different result…)



0	x0	S(x0-x1)	S(x0-x2)	S(x0-x3)	S(x0-x4)	S(x0-x5)	S(x0-x6)
x0	0	x2	S(x0-x1)	x4	S(x0-x3)	x6	S(x0-x5)
x0	S(x0-x1)	x2	0	x4	S(x4-x5)	x6	S(x0-x3)
x0	S(x0-x1)	x2	S(x0-x3)	x4	S(x4-x5)	x6	0

Parallel scan (2)

- Number of additions?

O(N)

- Memory access pattern?

Many conflicts!!

- Efficient implementation should consider this.

Home Assignment

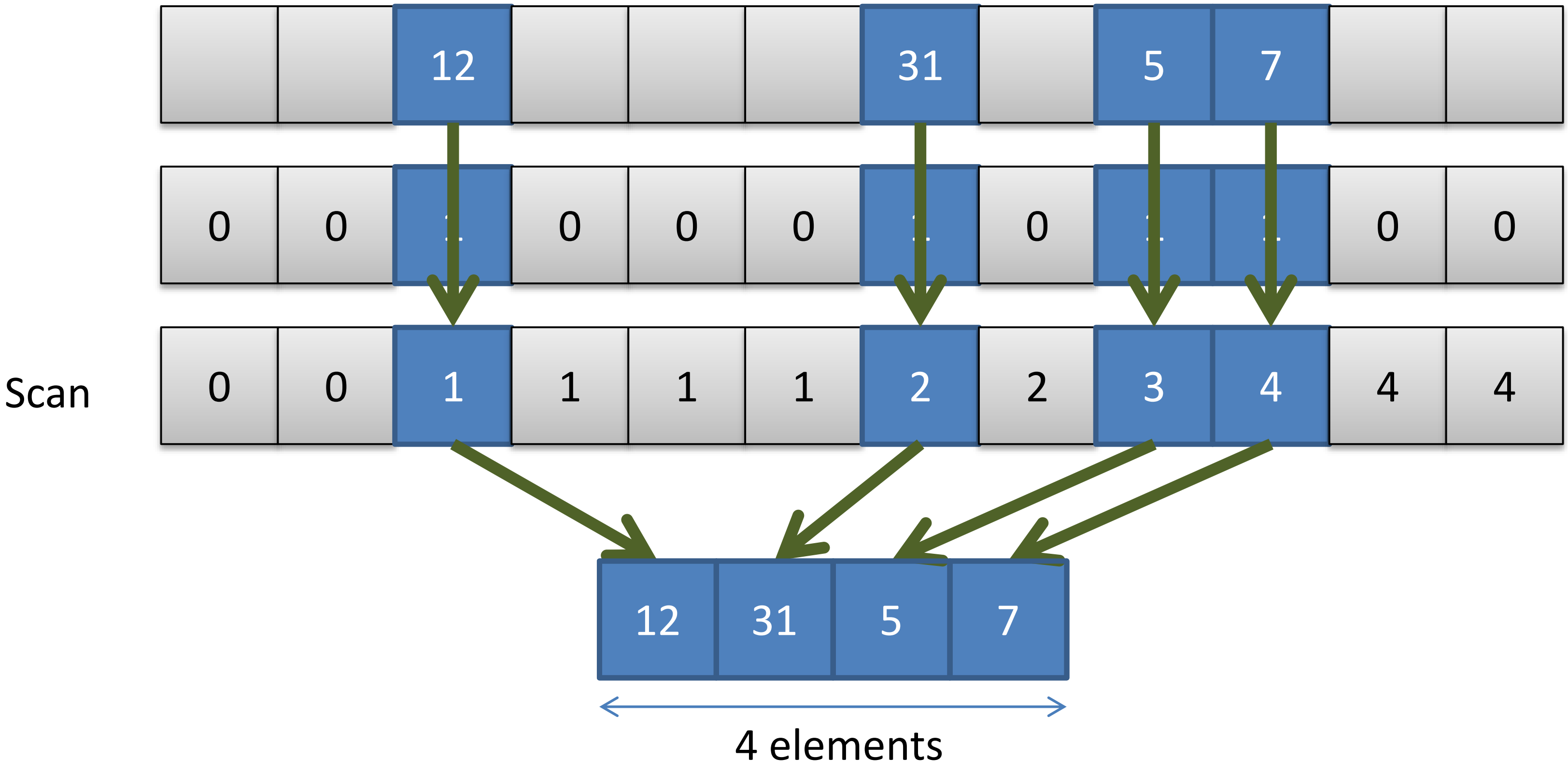
- Parallel scan
  - Global memory, all array sizes
  - First version (slide 20)
  - Second version (slide 26)
  - Compare performance
  - Implement application 1 + versus atomicInc (bonus)
- Due date: 2013-04-19
- Return to: sylvain.lefebvre@inria.fr

  Subject: [OpenCL DM]

- Warning: This is a test

Application #1

- Sparse array compaction



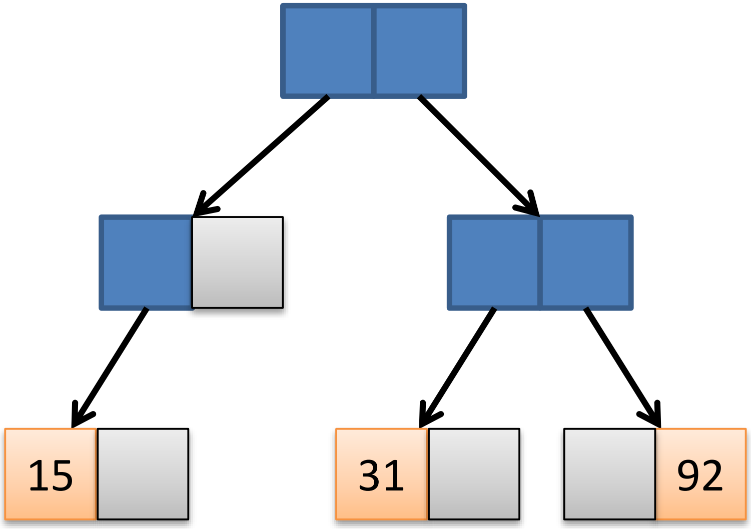
			12				31		5	7		
	0	0	1	0	0	0	1	0	1	1	0	0
Scan	0	0	1	1	1	1	2	2	3	4	4	4

12	31	5	7

4 elements

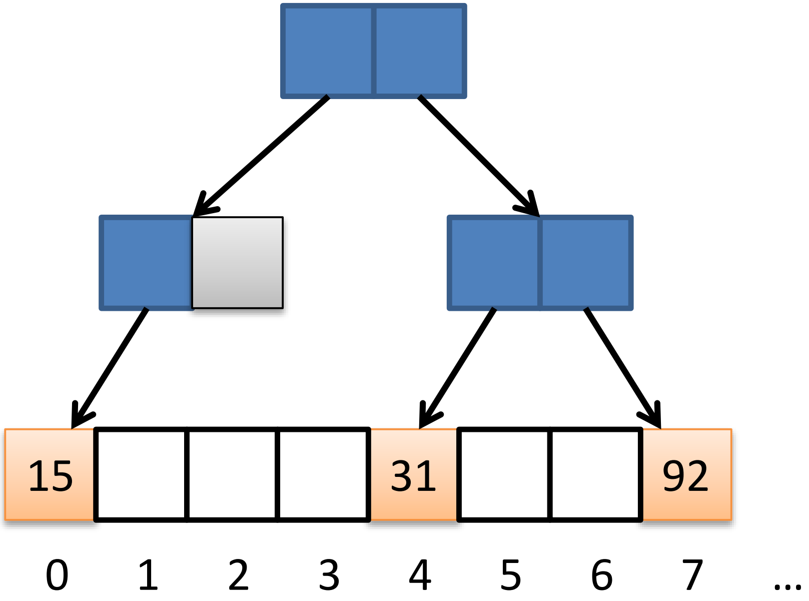
Application #2

- Special type of binary tree
  - Leaves at same level



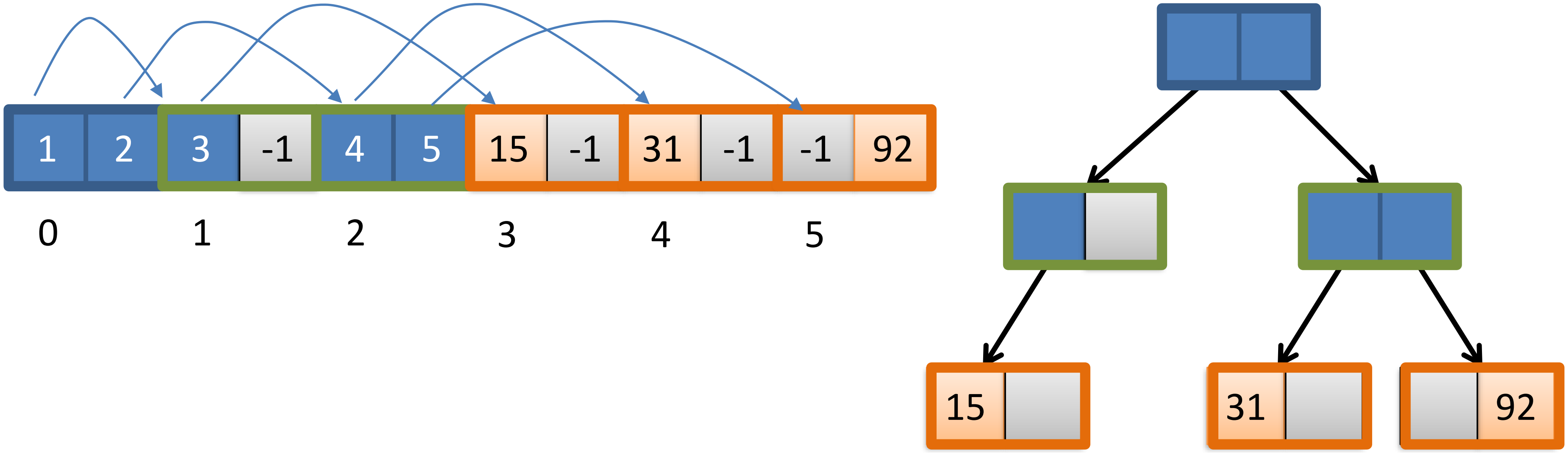
Binary-tree

- Captures a sparse array
  - Very common: Sparse linear algebra, integer sets, etc.
  - Fast retrieval O(log N)



Encoding in memory

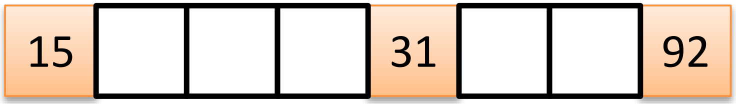
- All node records in a table
  - – Internal nodes: Two integer indices (left / right child)
  - – Leaves: Two integer data
- Called ‘autumnal tree’



Building a binary-tree

- Input:

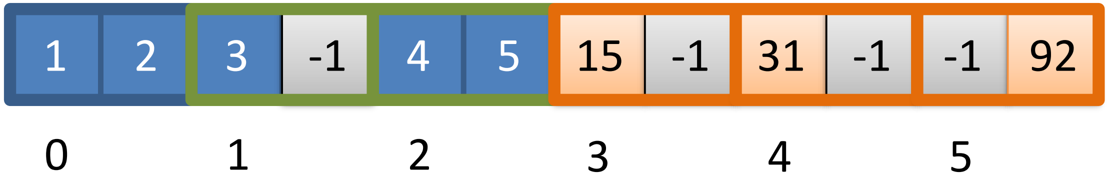
  Sparse table of integers



15				31			92

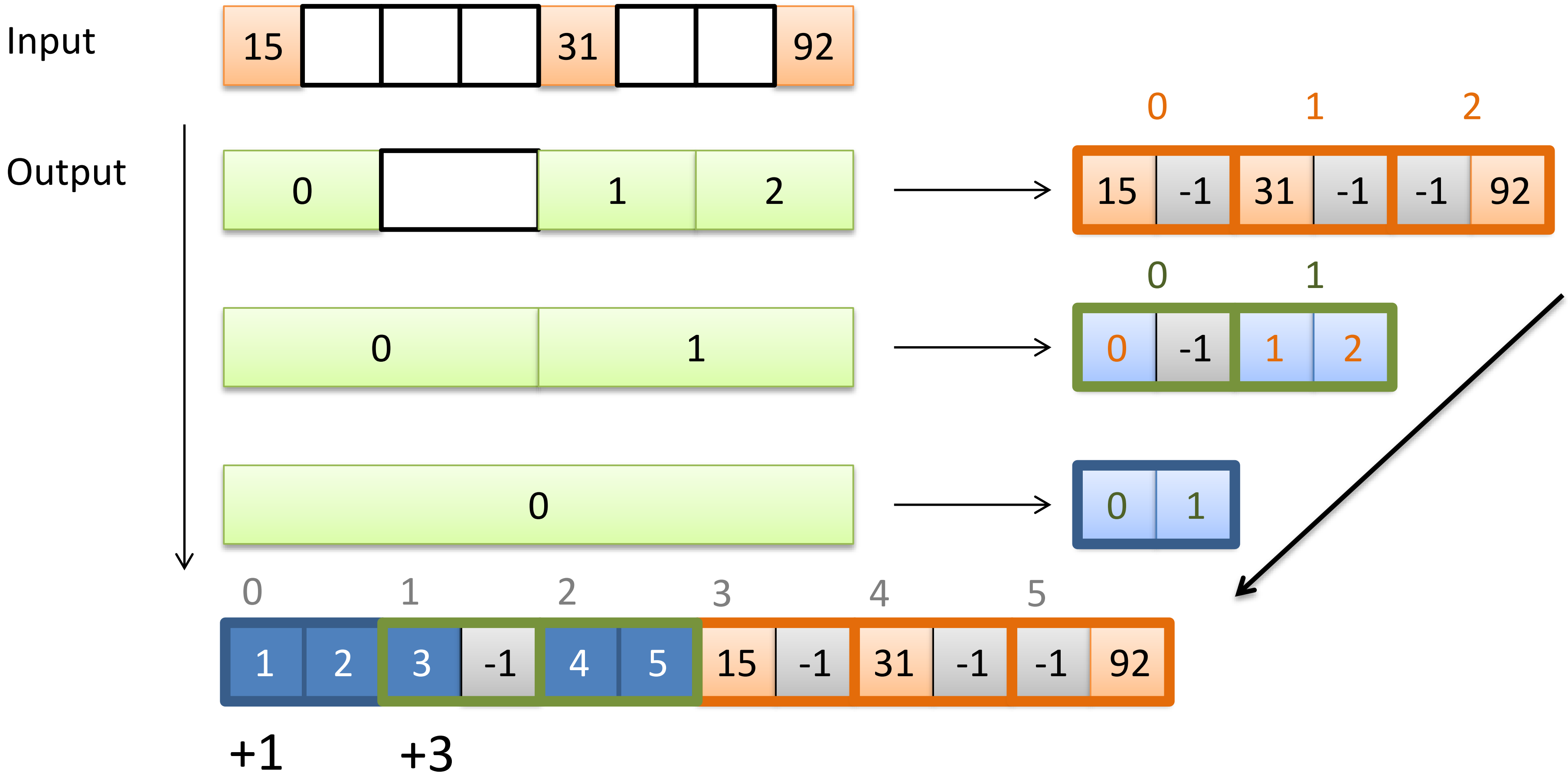
- Output:

  Node table



1	2	3	-1	4	5	15	-1	31	-1	-1	92
0		1		2		3		4		5	

Principle


Input
15
31
92
Output
0
1
2
0
1
0
0 1 2
15 -1 31 -1 -1 92
0 1
0 -1 1 2
0 1
0 1 2 3 4 5
1 2 3 -1 4 5 15 -1 31 -1 -1 92
+1
+3

Let’s practice!

- Reduction
  - atomicAdd (TD)
  - Hierarchy, global
  - Hierarchy, global, better coalescence
  - Local – global approach

- Further reading

  M. Harris
  - Talk on Reduction
  - GPU Gems 3 *Parallel Prefix Sum (Scan) with CUDA*