Scan Primitives for GPU Computing

Agenda

- **What is scan**
- A naïve parallel scan algorithm
- A work-efficient parallel scan algorithm
- Parallel segmented scan
- Applications of scan
- Implementation on CUDA

Prefix sum

- Prefix sum
  - Input:
    - An ordered set: $[x_0, x_1, x_2, x_3, \ldots]$
      - Array or linked-list
    - A binary *associative* operator: +
      - Associative: $(a+b)+c = a+(b+c)$
      - Don't need to be commutative: it's okay that $a+b \neq b+a$ (String concatenation for example.)
      - Identity element exists (additive identity :"0") $0+a=a+0=a$
      - Examples: add, multiply, max, min, AND, OR, Minkowski sum, etc.
  - Output:
    - $[x_0, x_0+x_1, x_0+x_1+x_2, x_0+x_1+x_2+x_3, \ldots]$

Scan

- Scan is array-based prefix sum
  - Elements are located continuously in the memory
  - Easy for GPU implementation
  - Inclusive scan
    - $[x_0, x_0+x_1, x_0+x_1+x_2, x_0+x_1+x_2+x_3, \ldots]$
  - Exclusive scan (prescan)
    - $[0, x_0, x_0+x_1, x_0+x_1+x_2, \ldots]$

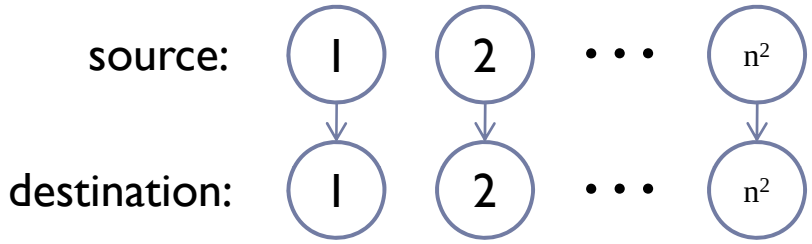
Sequential Implementation of Inclusive Scan

```
1: s ← x[0]
2: for i ← 1 to n − 1 do
3:     s ← s + x[i]
4:     x[i] ← s
```

- Step Complexity: $O(n)$
  - Total number of steps
- Work Complexity: $O(n)$
  - Total number of operations

Work Complexity

- Work complexity matters in parallel algorithm
- Look at a simple parallel problem: copy n^2 data



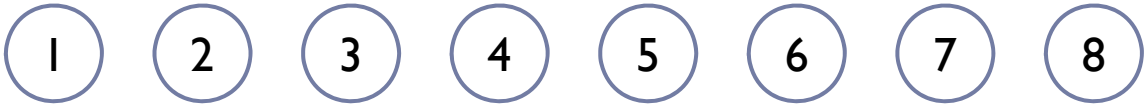
  - Step complexity = 1; work complexity = n^2
- If we have n^2 processors, we only need 1 parallel step.
- If we have n processors, each process needs to do n copies. So we need n parallel steps.
- (Brent 1974) step complexity S, work complexity W, p processors

$$\#\, parallel\ \ steps \leq \left\lfloor \frac{W}{p} \right\rfloor + S$$

Agenda

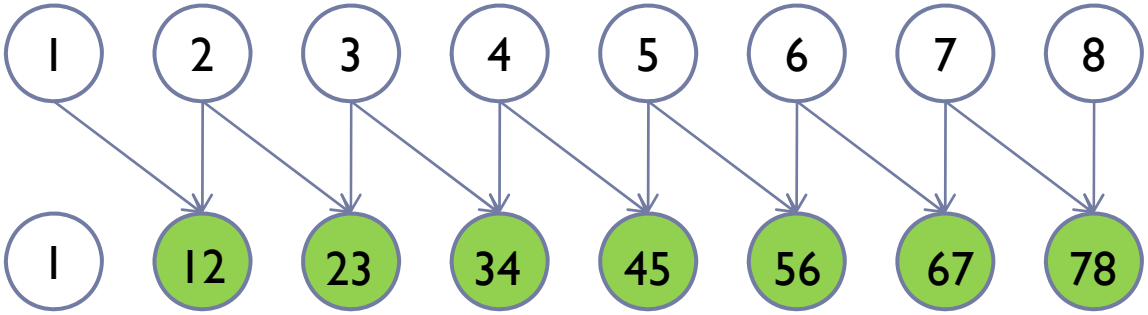
- What is scan
- **A naïve parallel scan algorithm**
- A work-efficient parallel scan algorithm
- Parallel segmented scan
- Applications of scan
- Implementation on CUDA

Horn's GPU Implementation

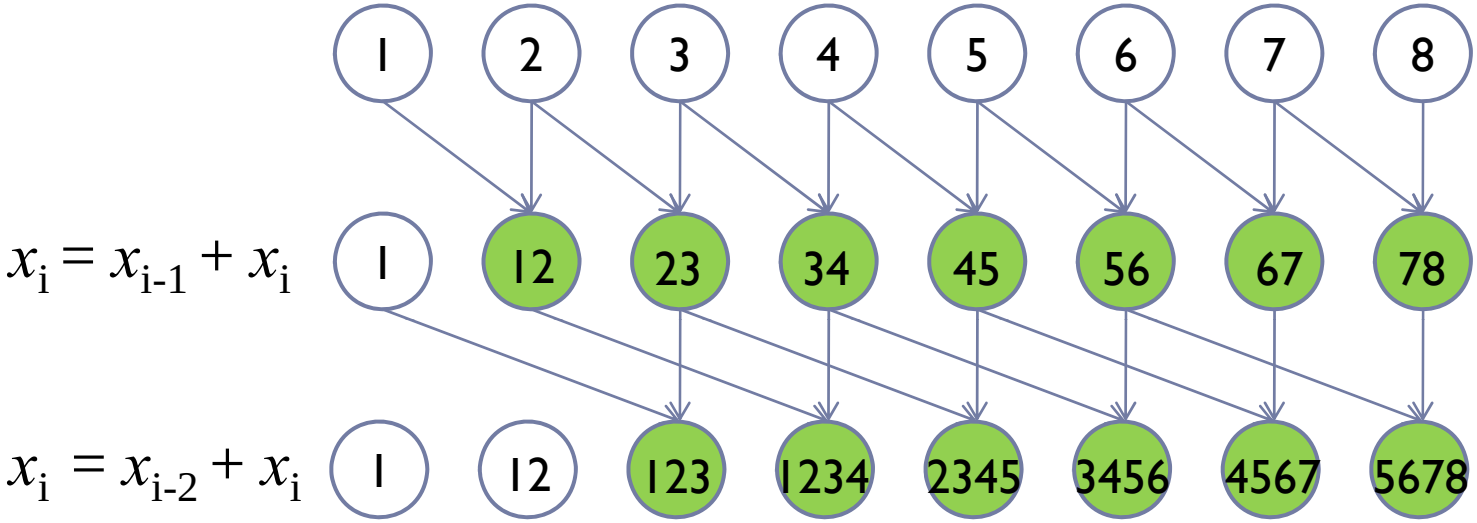


Horn's GPU Implementation

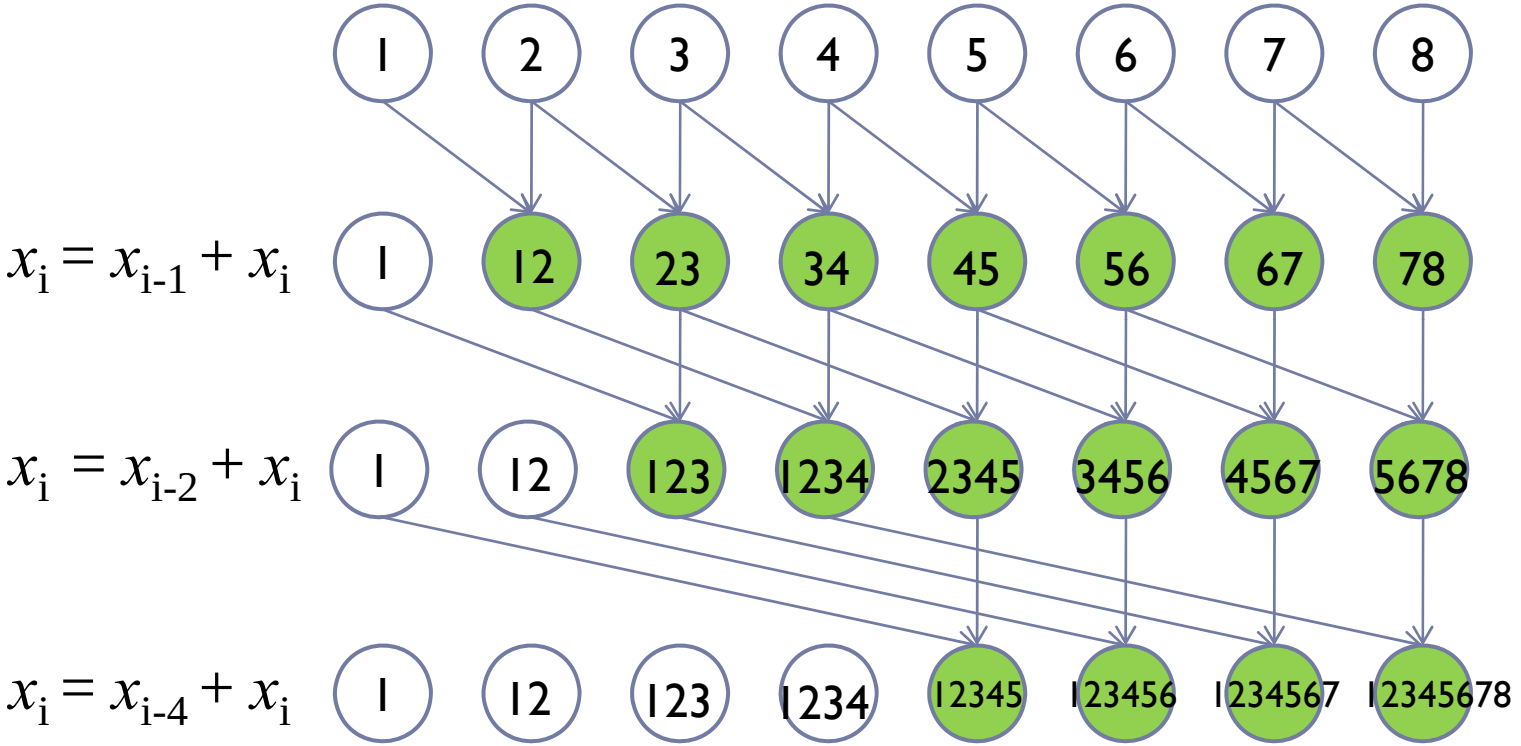
$x_i = x_{i-1} + x_i$



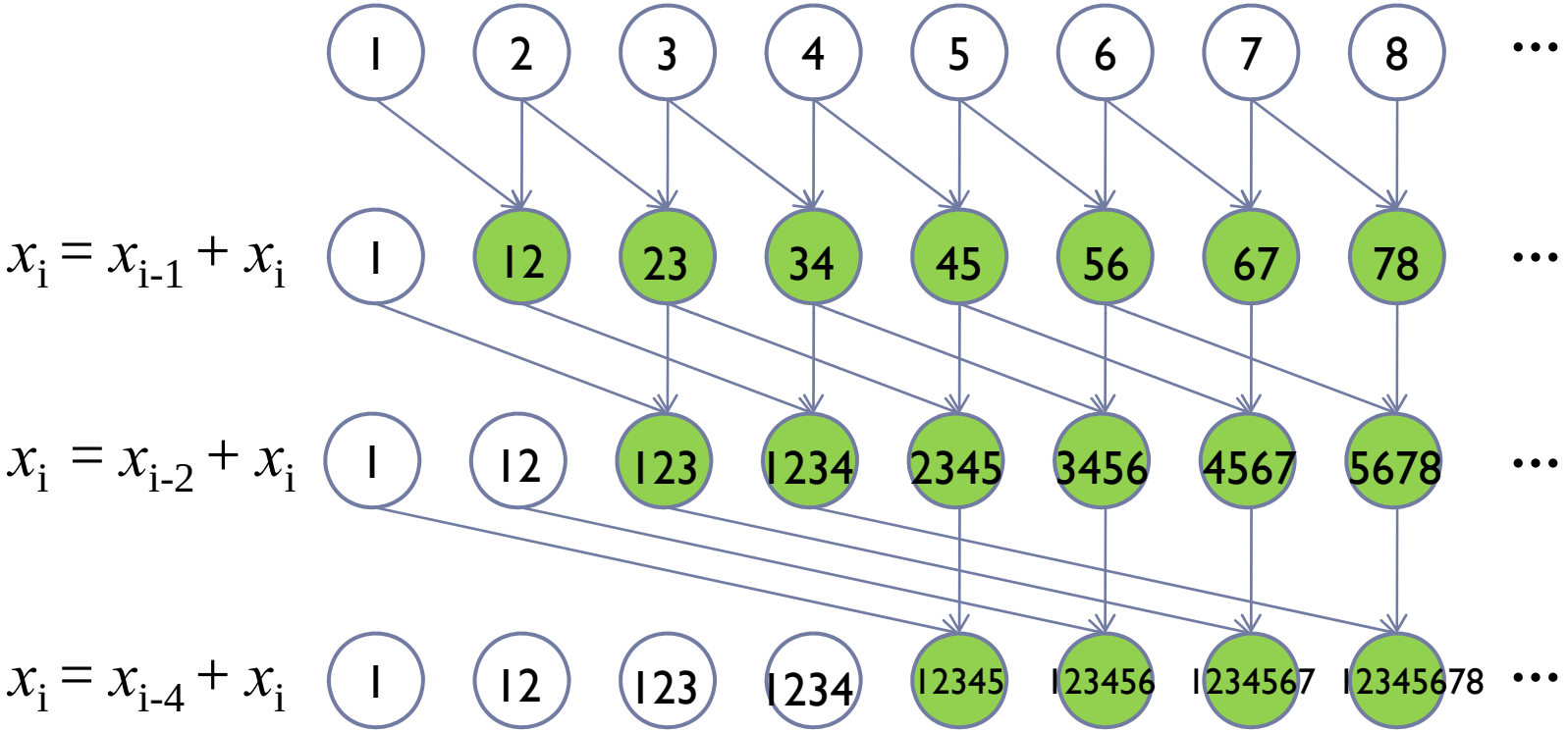
Horn's GPU Implementation



Horn's GPU Implementation



If we have 16 numbers



$\boldsymbol{x_i = x_{i-8} + x_i}$ ••••••

Horn's GPU Implementation

- Horn, Daniel. Stream reduction operations for GPGPU applications. In *GPU Germs 2*.
- Step Complexity: $O(\log n)$
  - Step complexity of sequential algorithm is $O(n)$
  - Efficient
- Work Complexity:
  - $= (n\text{-}1)+(n\text{-}2)+(n\text{-}4)+\ldots$
  - $= \sum_{d=0}^{\log n}(n-2^d)$
  - $= n\log n\text{-}n+1 = O(n\log n)$
  - Work complexity of sequential algorithm is $O(n)$
  - Not work-Efficient

Agenda

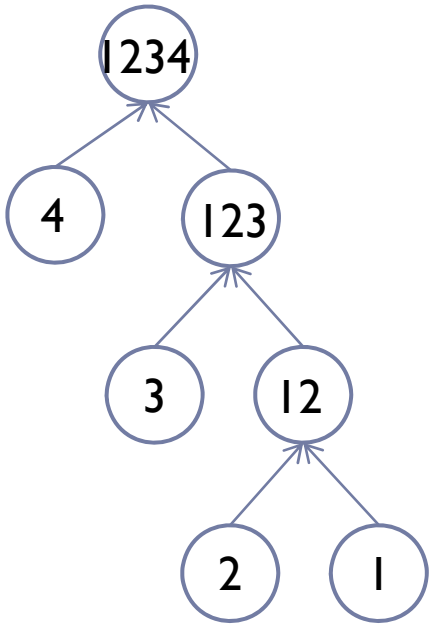
- What is scan
- A naïve parallel scan algorithm
- **A work-efficient parallel scan algorithm**
- Parallel segmented scan
- Applications of scan
- Implementation on CUDA

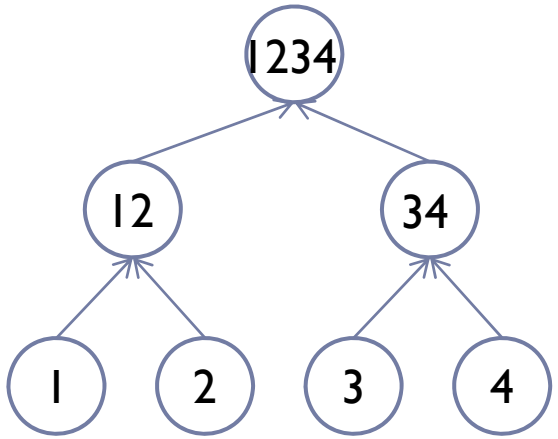
A Work-Efficient Scan Algorithm

- Exclusive Scan
- Step Complexity: $O(\log n)$
- Work Complexity: $O(n)$
- Use (Implicit) Balanced Binary Tree
  - Just a concept. Don't need to build a real tree.
  - Perform calculations on the array directly.
  - Easy for implementation on GPU.
- Two passes over the array (tree)
  - Reduce (up-sweep)
  - Down-sweep

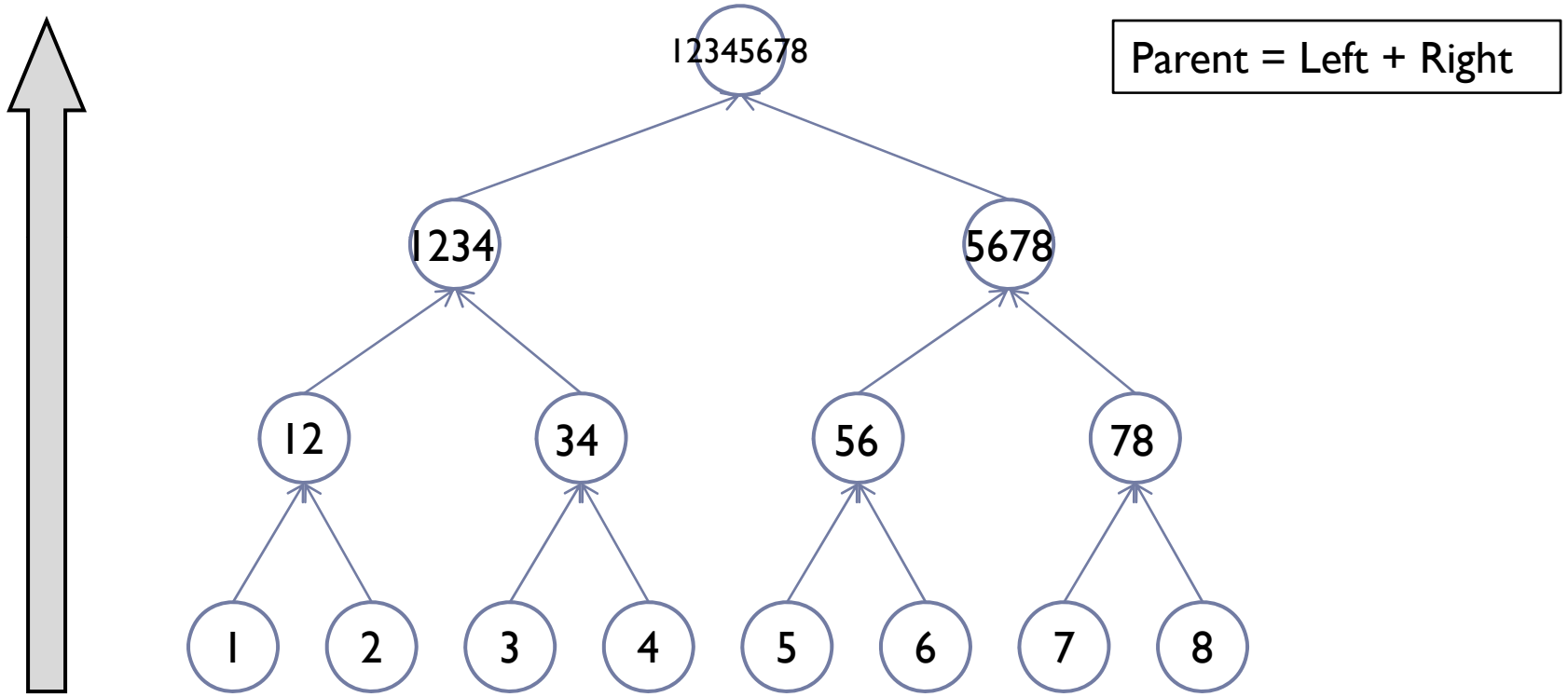
Balanced Binary Tree

- A binary tree where the depth of all the leaves differs by at most 1.
- The height of a balanced binary tree is minimized.
  - Take reduction for an example

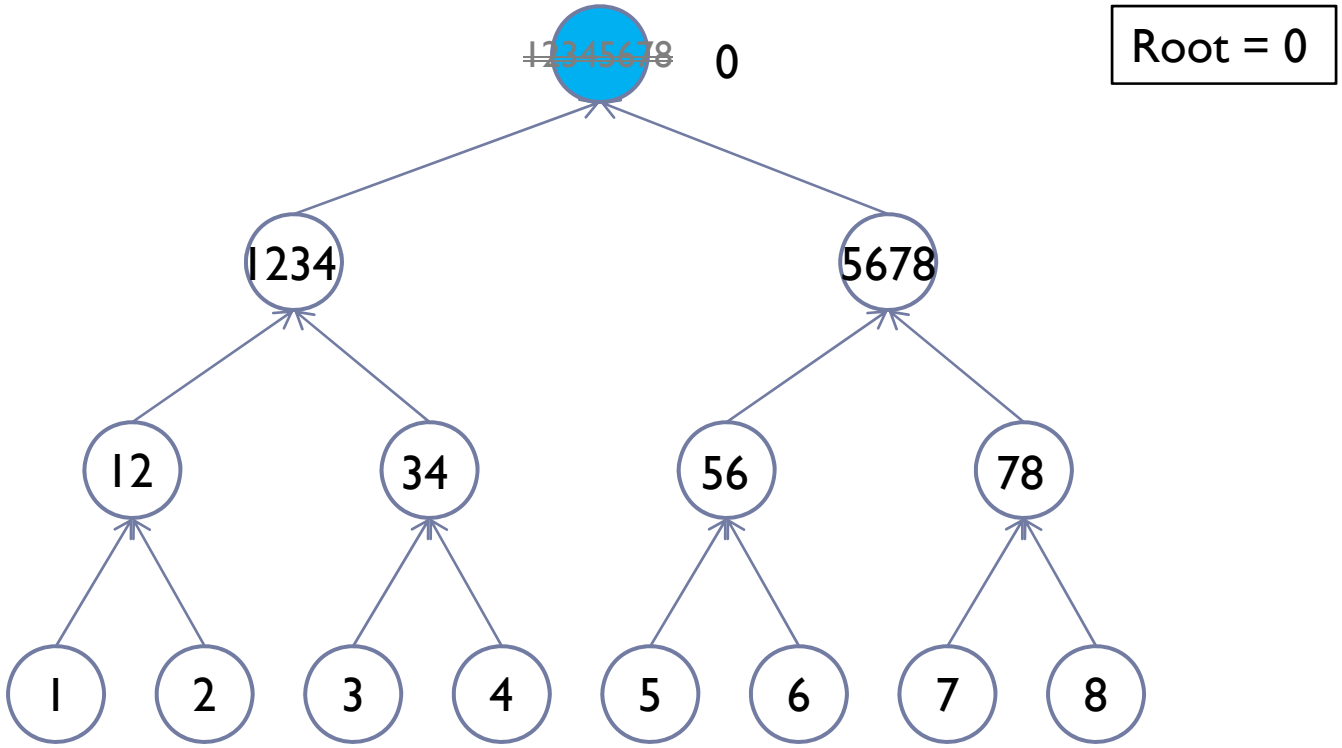




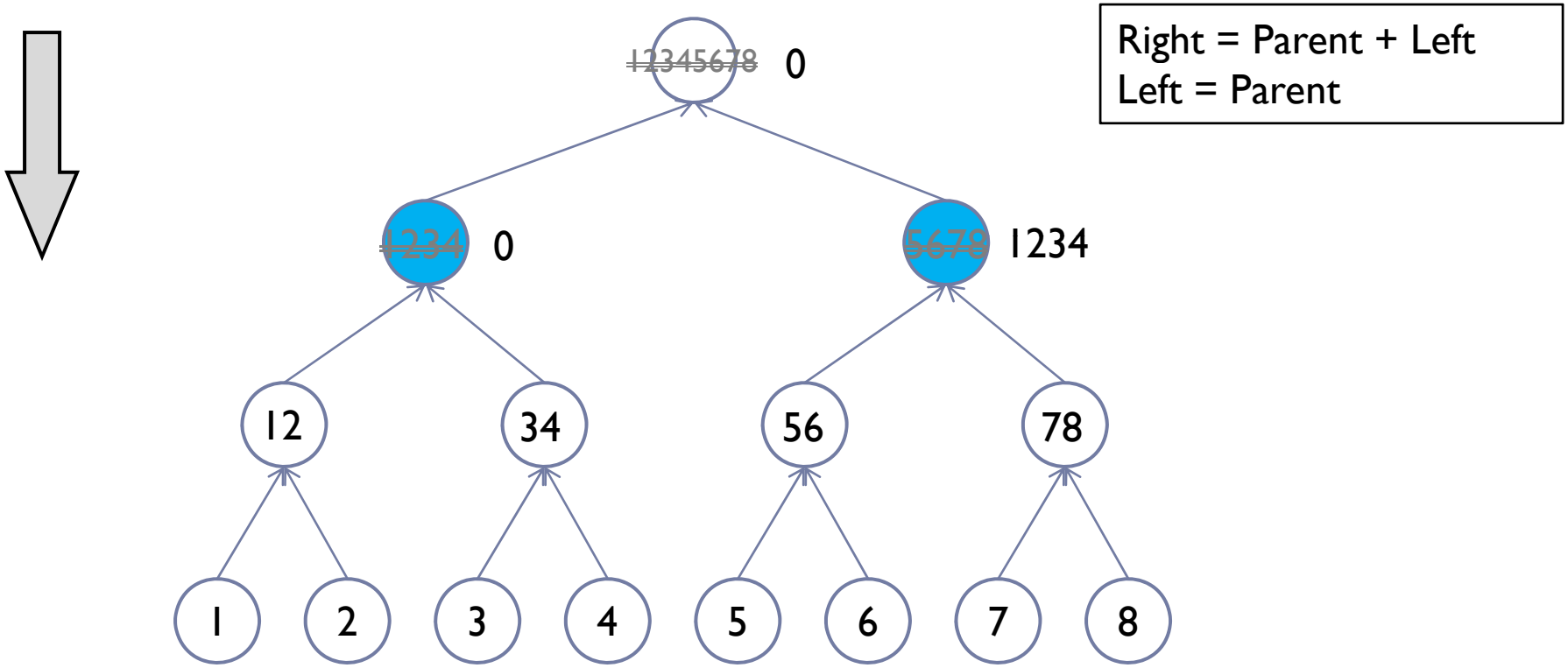
Up-sweep (Balanced Tree)



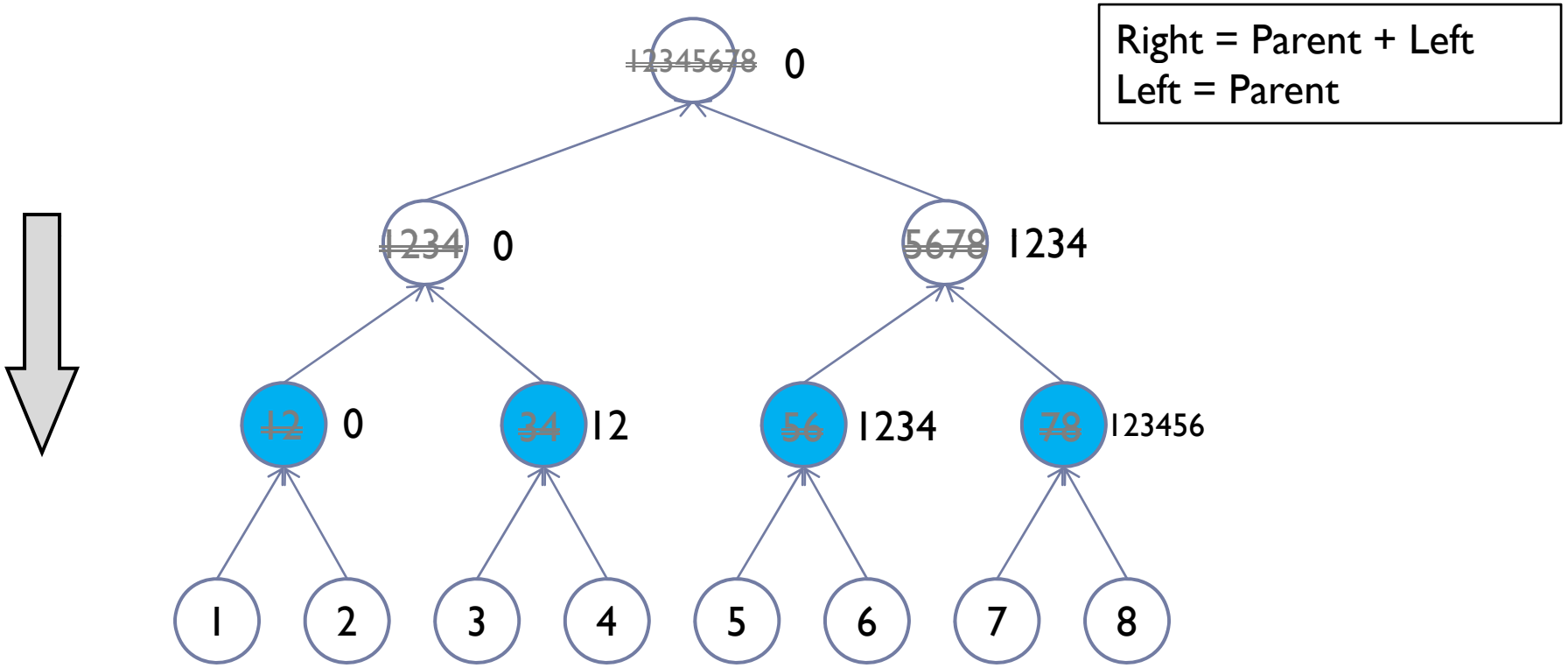
Down-sweep (Balanced Tree)



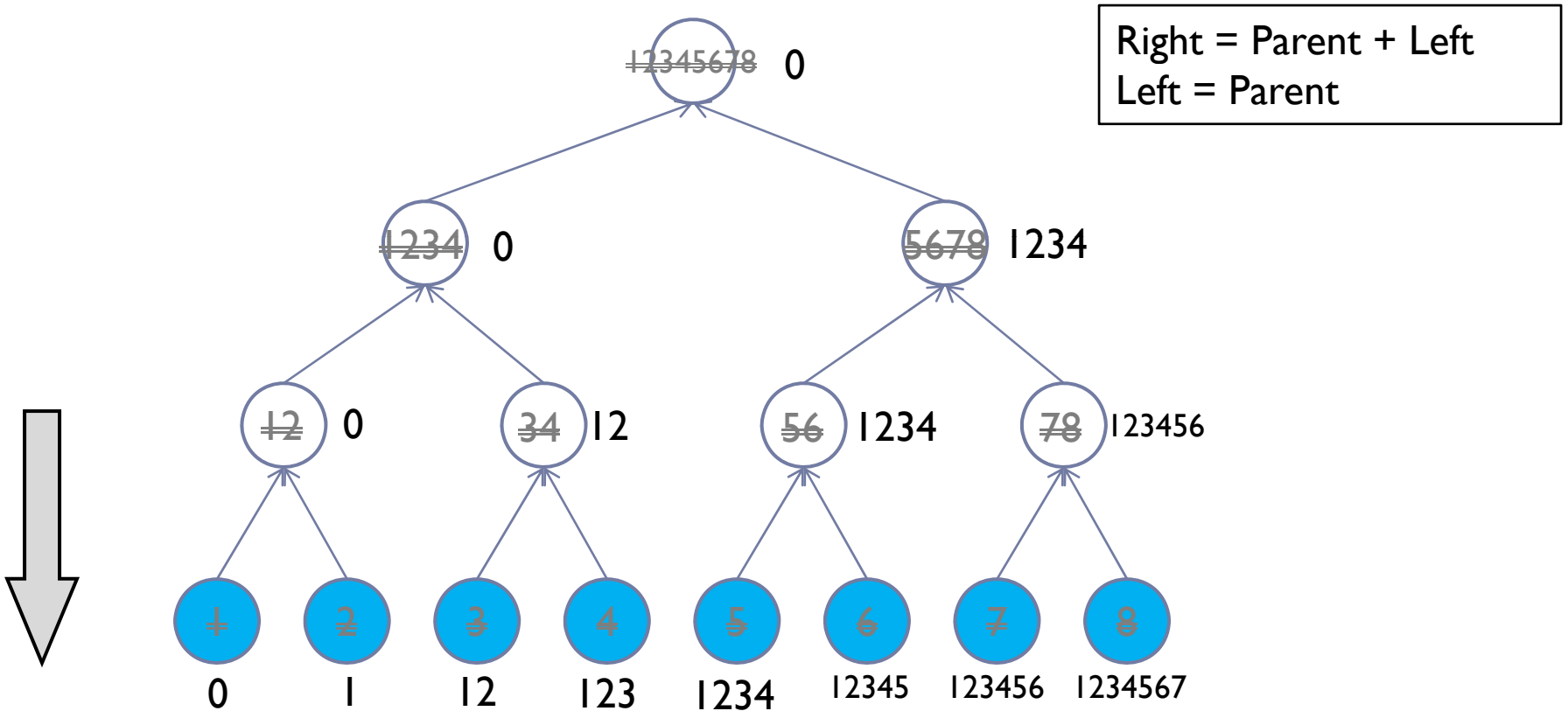
Down-sweep (Balanced Tree)



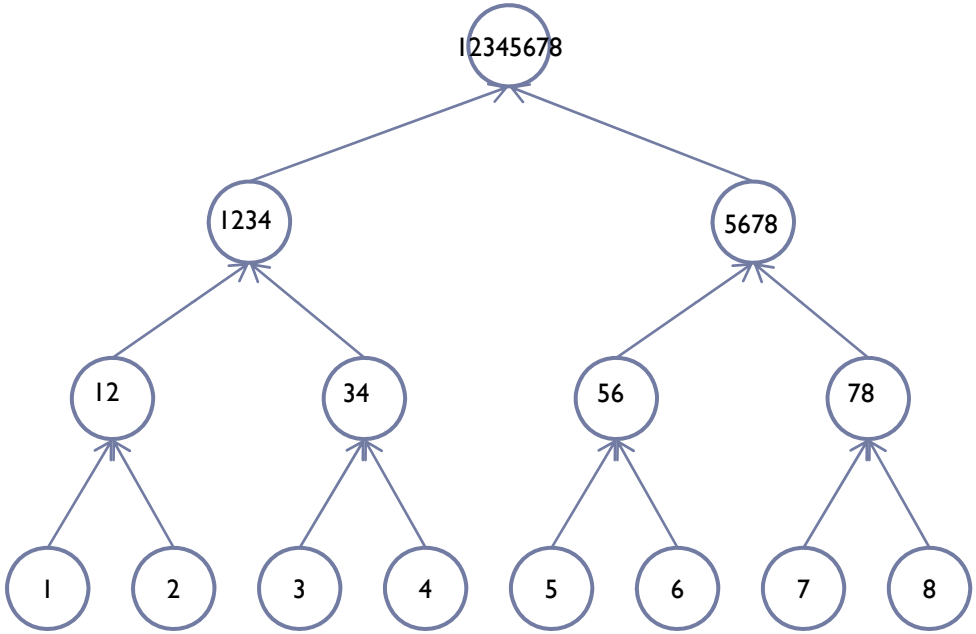
Down-sweep (Balanced Tree)



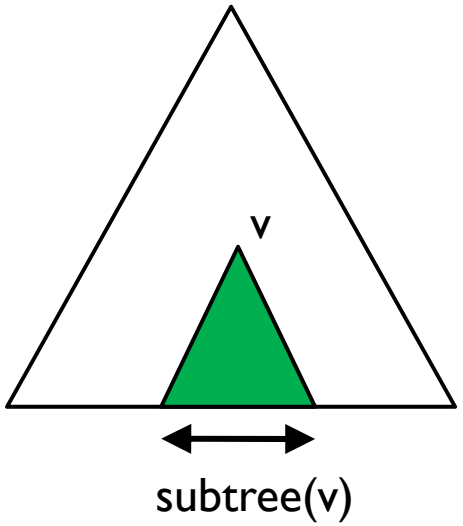
Down-sweep (Balanced Tree)



Why It Works? (up-sweep)

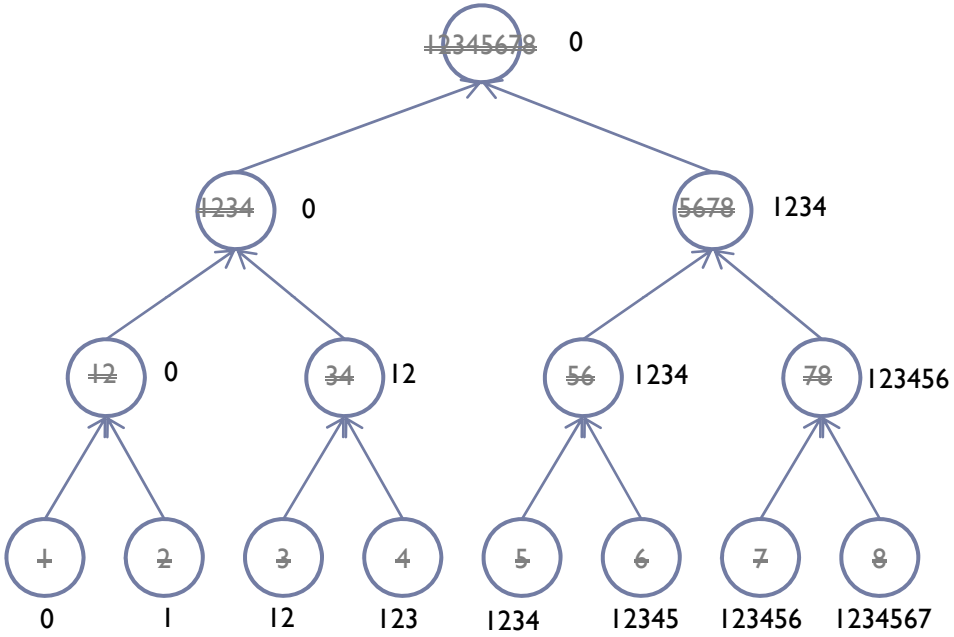


Parent = Left + Right

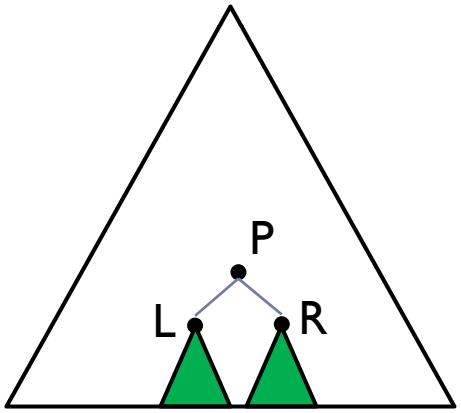


After reduction (up-sweep), each node contains the partial sum of its subtree

Why It Works? (down-sweep)



Right = Parent + Left
Left = Parent



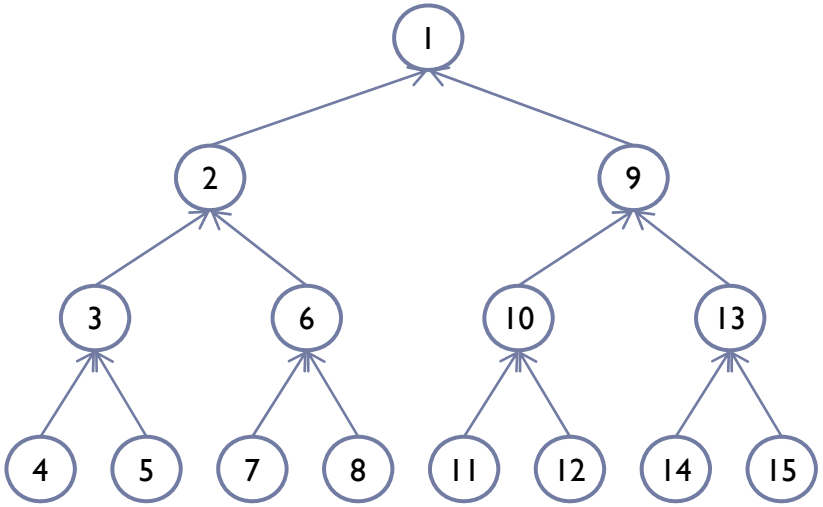
pre(P), pre(L) ⟷

pre(R) ⟷

After down-sweep, each node contains the sum of all the leaves that precede it in the preorder traversal.

Pre-order traversal

- Traverse the tree recursively in the following order:
  - Visit the root
  - Visit the left subtree
  - Visit the right subtree



Items preceding the green node (9) are marked in yellow.

What are items preceding the left (10) and right (13) child of the green node (9)?

Right = Parent + Left
Left = Parent

Why we reset the root to zero at the beginning of down-sweep?

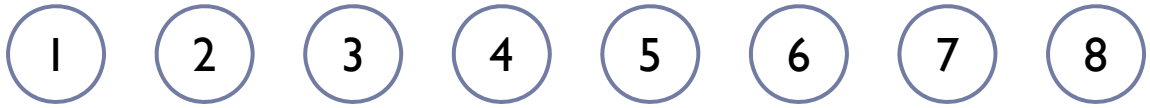
Why pre-order?

- Actually in-order and post-order also work!
  - Try to derive their formulas
- But pre-order gives the simplest formula.
  - Post-order gives a beautiful and concise formula for inclusive scan; however it requires to define a corresponding "minus" operator, which is not always trivial (like Minkowski sum)
  - In-order also requires "minus" operator. In addition, its formula involves three continuous levels (parent, child, grandchild)
- Pre-order doesn't need "minus" operator, because the children always come after the parent.

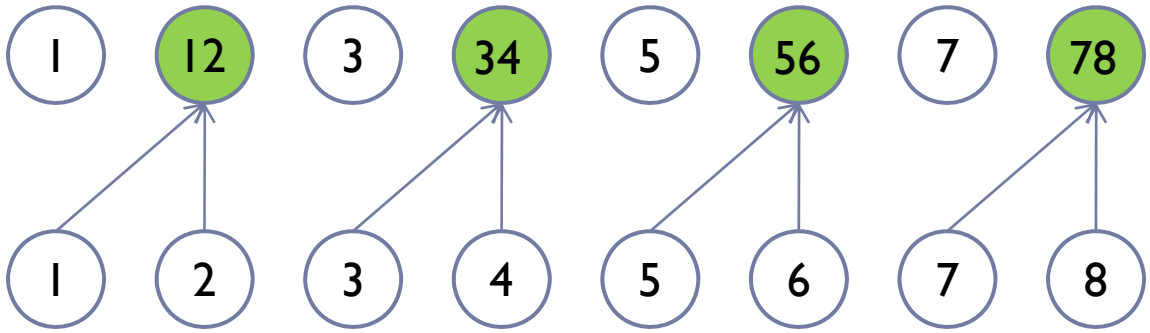
In-place computation without a tree

- Balanced Binary Tree is just a concept. We don't really build such a tree.
- All the computations are performed on the array directly.
  - Given an item in the array, we know the exact index of its parent or left/right child. So there is no need to record the parent-child relationship like a tree data structure does.

Up-sweep (in-place)

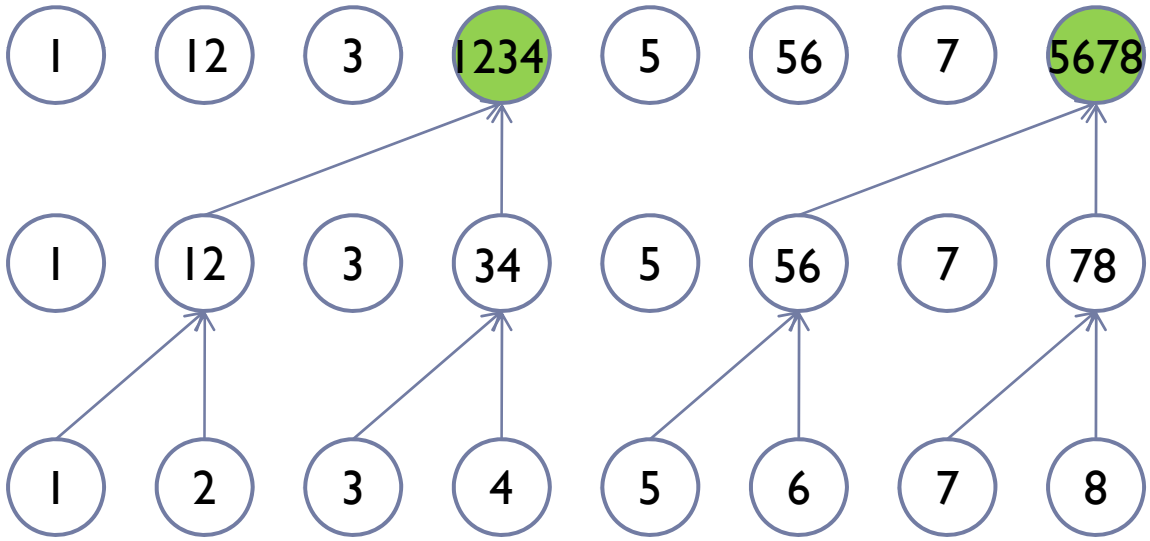


Up-sweep (in-place)



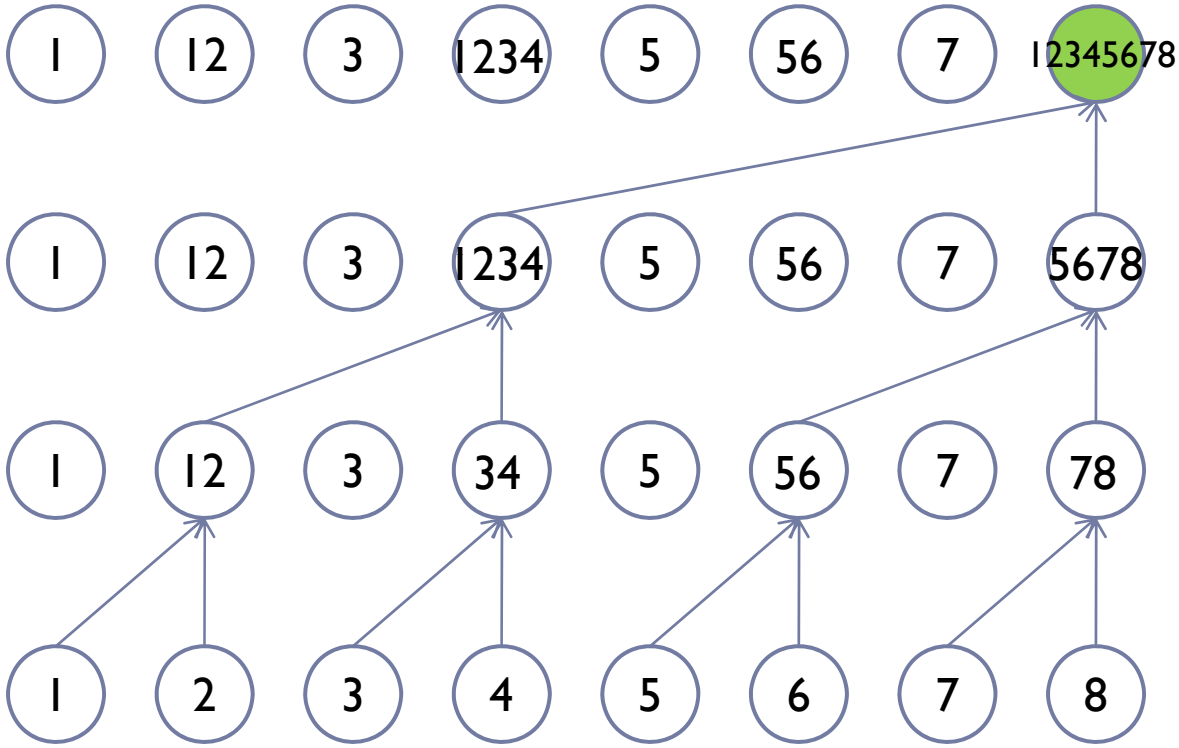
Parent = Left + Right

Up-sweep (in-place)



Parent = Left + Right

Up-sweep (in-place)



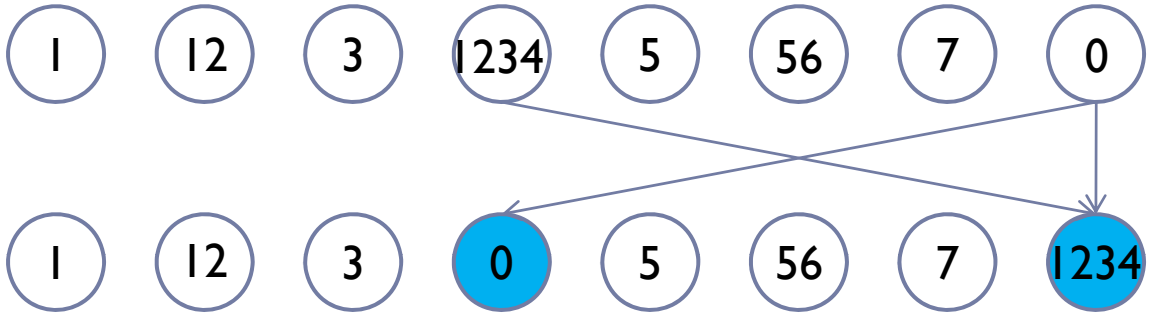
Parent = Left + Right

Down-sweep (in-place)

Down-sweep (in-place)

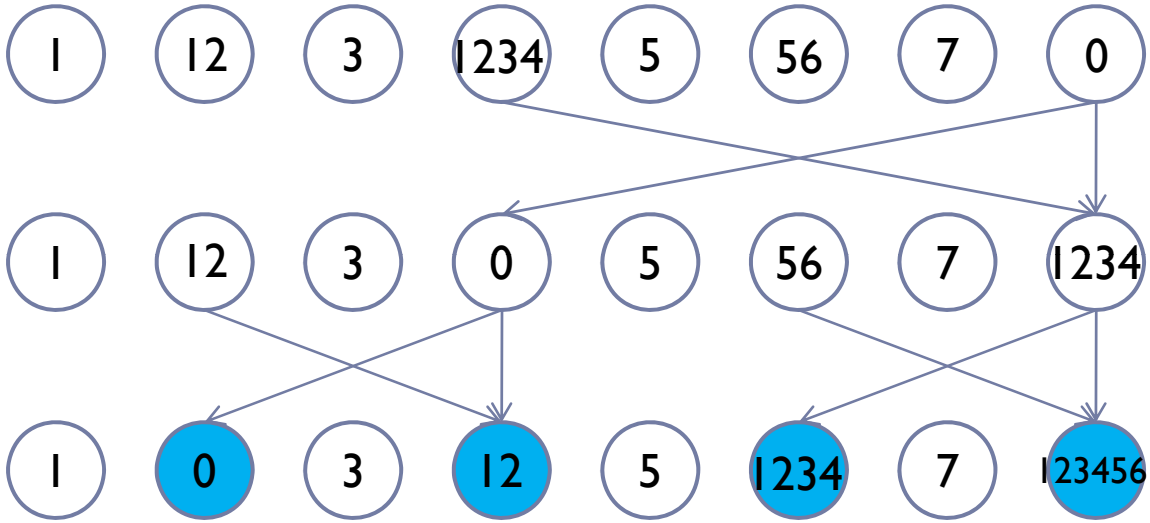
Root = 0

Down-sweep (in-place)



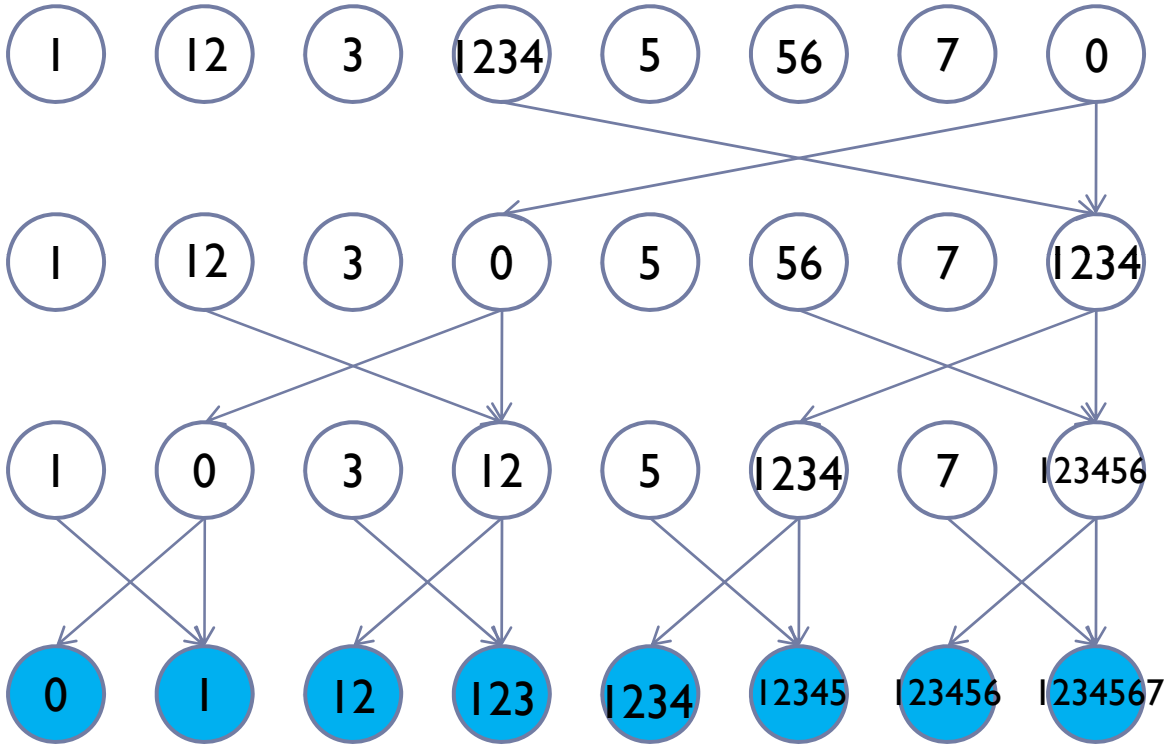
Right = Parent + Left
Left = Parent

Down-sweep (in-place)



Right = Parent + Left
Left = Parent

Down-sweep (in-place)



Right = Parent + Left
Left = Parent

Pseudocode

Up-sweep (reduce):

```
1: for d = 0 to log2 n − 1 do
2:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
3:         x[k + 2^(d+1) − 1] ← x[k + 2^d − 1] + x[k + 2^(d+1) − 1]
```

Down-sweep:

```
1: x[n − 1] ← 0
2: for d = log2 n − 1 down to 0 do
3:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
4:         t ← x[k + 2^d − 1]
5:         x[k + 2^d − 1] ← x[k + 2^(d+1) − 1]
6:         x[k + 2^(d+1) − 1] ← t + x[k + 2^(d+1) − 1]
```

Complexity Analysis

- Step complexity
  - Up-sweep: $\log(n)$
  - Down-sweep: $\log(n)$
- Work complexity
  - Up-sweep

$$\sum_{d=0}^{\log n-1} \frac{n}{2^{d+1}} = n-1$$

  - Down-sweep

$$1+3\sum_{d=0}^{\log n-1} \frac{n}{2^{d+1}} = 3n-2$$

Agenda

- What is scan
- A naïve parallel scan algorithm
- A work-efficient parallel scan algorithm
- **Parallel segmented scan**
- Applications of scan
- Implementation on CUDA

Segmented Scan

- Input array is broken into arbitrary segments;
- Scan is performed in each segment;
  - Additional input: flag array (1: start of a new segment)
  - Flag: [1 0 0 1 0 0 1 0]
  - Data: [4 2 1 3 0 2 1 5] =>
  - Result: [0 4 6 0 3 3 0 1]
- Still $O(n)$ work complexity and $O(\log n)$ step complexity;
- Only three times slower than scan;
- Algorithm is slightly different, but more tricky.

Segmented Scan

```
1: for d = 1 to log2 n − 1 do
2:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
3:         if f[k + 2^(d+1) − 1] is not set then
4:             x[k + 2^(d+1) − 1] ← x[k + 2^d − 1] + x[k + 2^(d+1) − 1]
5:         f[k + 2^(d+1) − 1] ← f[k + 2^d − 1] | f[k + 2^(d+1) − 1]
```

```
1: x[n − 1] ← 0
2: for d = log2 n − 1 down to 0 do
3:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
4:         t ← x[k + 2^d − 1]
5:         x[k + 2^d − 1] ← x[k + 2^(d+1) − 1]
6:         if f_l[k + 2^d] is set then
7:             x[k + 2^(d+1) − 1] ← 0
8:         else if f[k + 2^d − 1] is set then
9:             x[k + 2^(d+1) − 1] ← t
10:        else
11:            x[k + 2^(d+1) − 1] ← t + x[k + 2^(d+1) − 1]
12:        Unset flag f[k + 2^d − 1]
```

Scan

```
1: for d = 0 to log2 n − 1 do
2:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
3:         x[k + 2^(d+1) − 1] ← x[k + 2^d − 1] + x[k + 2^(d+1) − 1]
```

```
1: x[n − 1] ← 0
2: for d = log2 n − 1 down to 0 do
3:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
4:         t ← x[k + 2^d − 1]
5:         x[k + 2^d − 1] ← x[k + 2^(d+1) − 1]
6:         x[k + 2^(d+1) − 1] ← t + x[k + 2^(d+1) − 1]
```

Why segmented scan?

- Why not scan in each segment independently?
  - Suppose we have m arrays, each of which has n numbers. Scanning them independently takes $O(m \log n)$ step complexity.
  - If we combine them into a single array with mn numbers, and do a segmented scan, it takes only $O(\log mn)$ step complexity.
- Segmented scan has lot of applications, such as quicksort, sparse matrix-vector multiplication, processor allocation, etc.

Convert segmented scan into scan

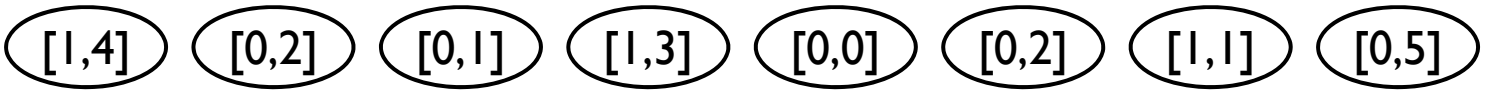
- Given flag array: f_i (0 or 1)
- Given data array: x_i
- Define the array of pairs: $c_i = [f_i, x_i]$
- Define a new binary operator $\odot$ on c_i

$$c_1 \odot c_2 = [f_1, x_1] \odot [f_2, x_2]$$

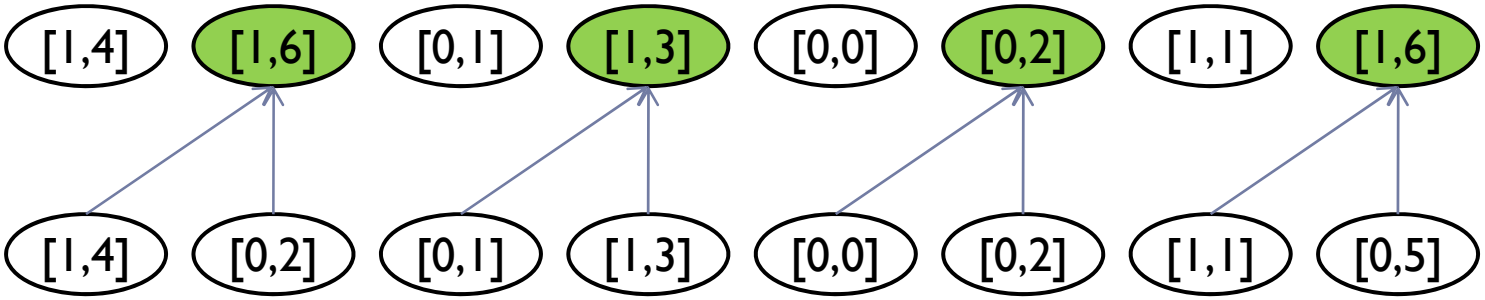
$$= \begin{cases} [f_1 \mid f_2, x_1 + x_2] & \text{if } f_2 = 0 \\ [f_1 \mid f_2, x_2] & \text{if } f_2 = 1 \end{cases}$$

  - Prove that $\odot$ is associative
  - The identity element for $\odot$ is (0,'0')
- Now scan over array c_i with operator $\odot$

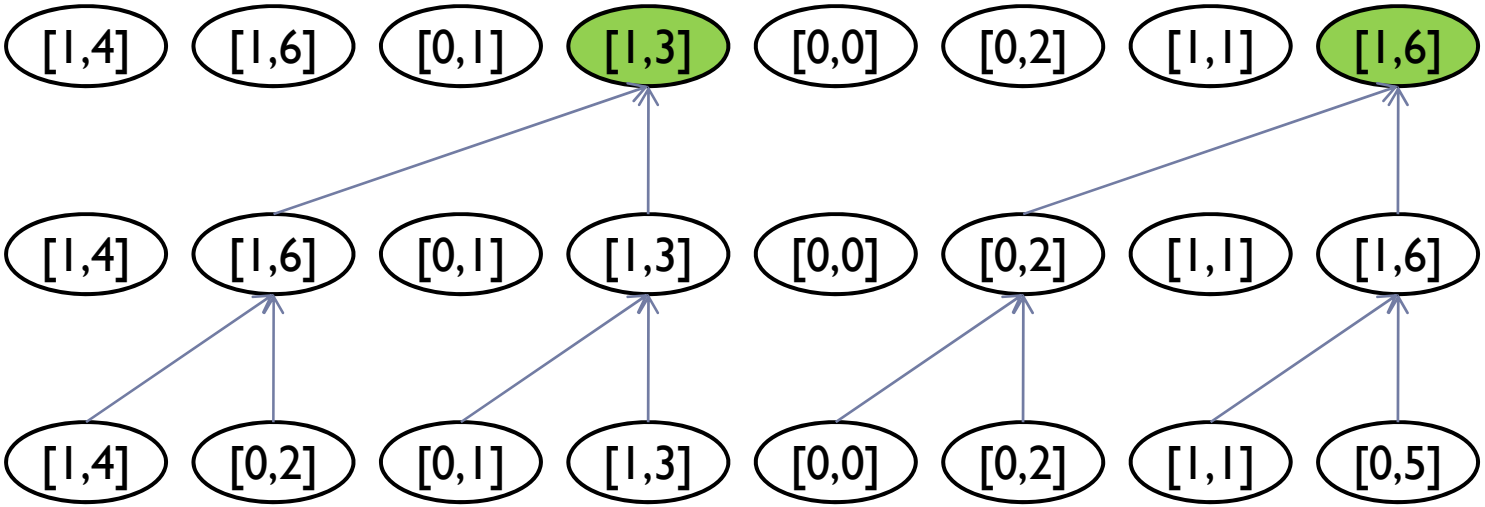
Reduce



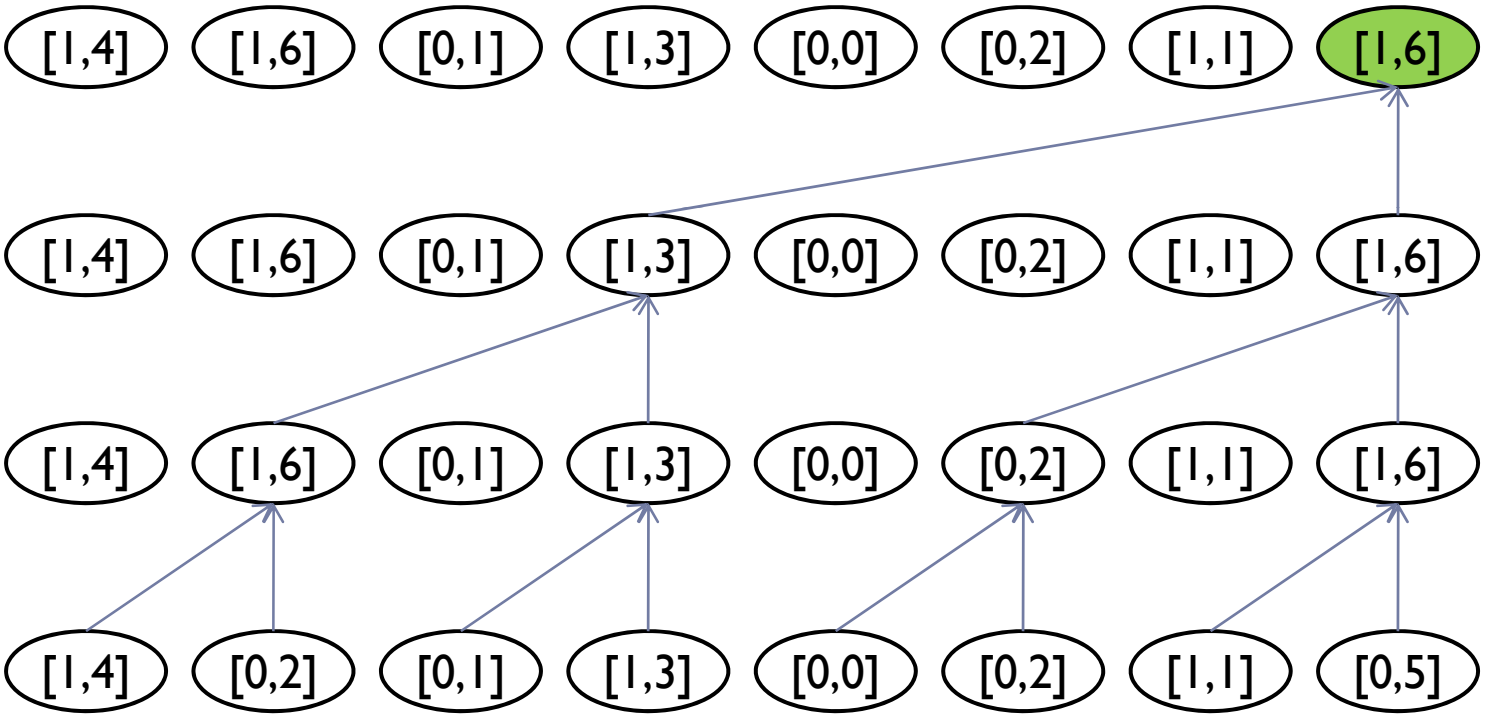
Reduce



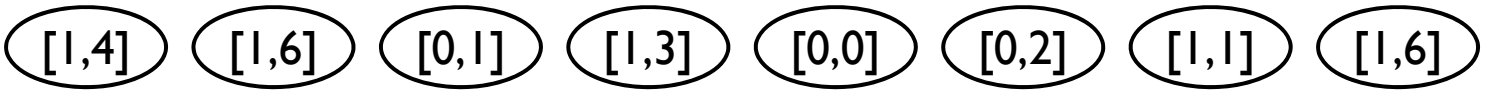
Reduce



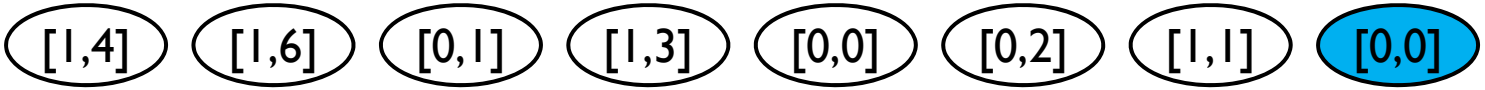
Reduce


[1,4]
[1,6]
[0,1]
[1,3]
[0,0]
[0,2]
[1,1]
[1,6]
[1,4]
[1,6]
[0,1]
[1,3]
[0,0]
[0,2]
[1,1]
[1,6]
[1,4]
[1,6]
[0,1]
[1,3]
[0,0]
[0,2]
[1,1]
[1,6]
[1,4]
[0,2]
[0,1]
[1,3]
[0,0]
[0,2]
[1,1]
[0,5]

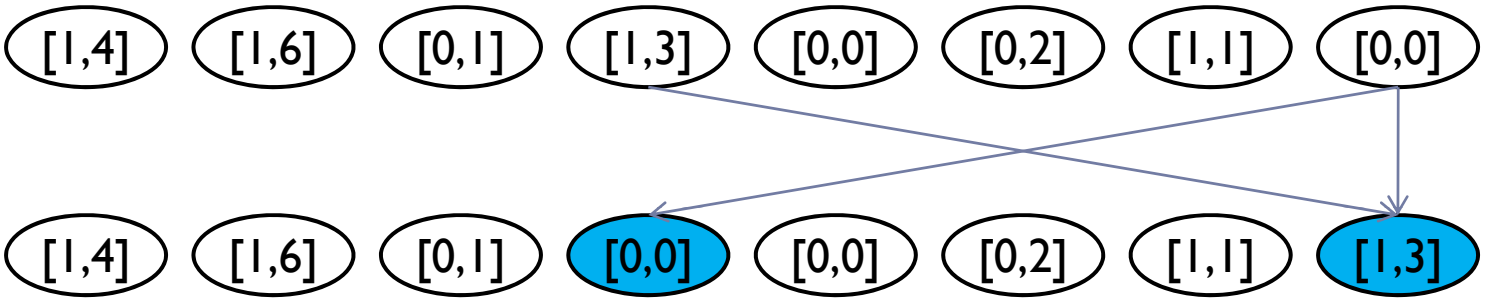
Down-sweep



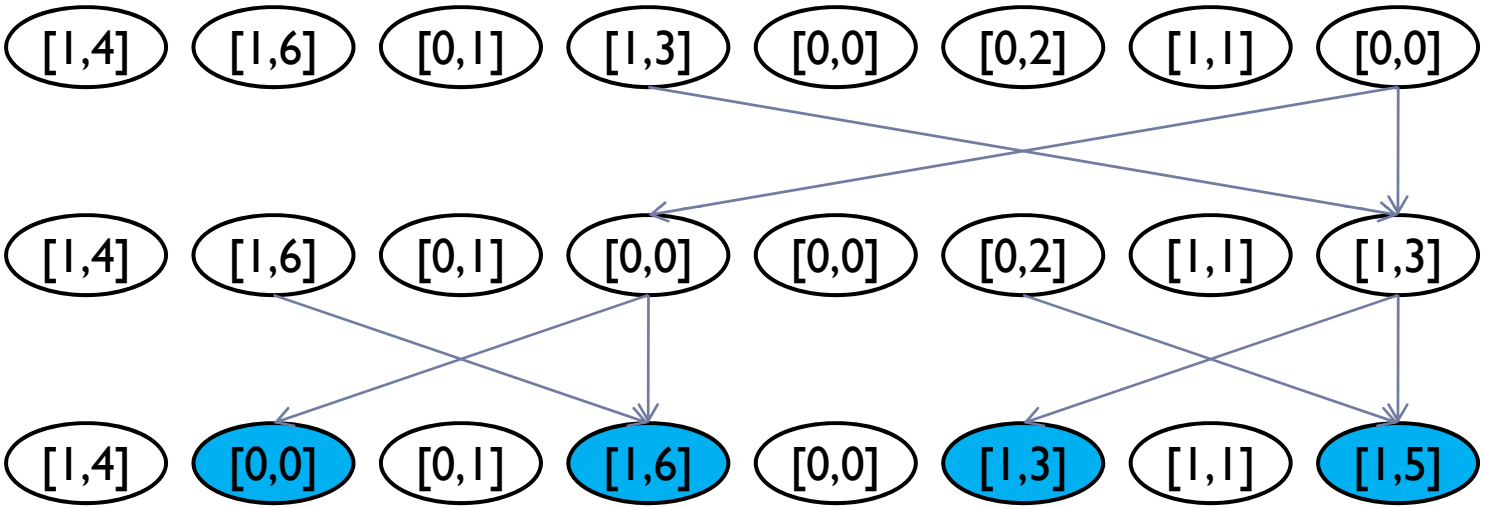
Down-sweep



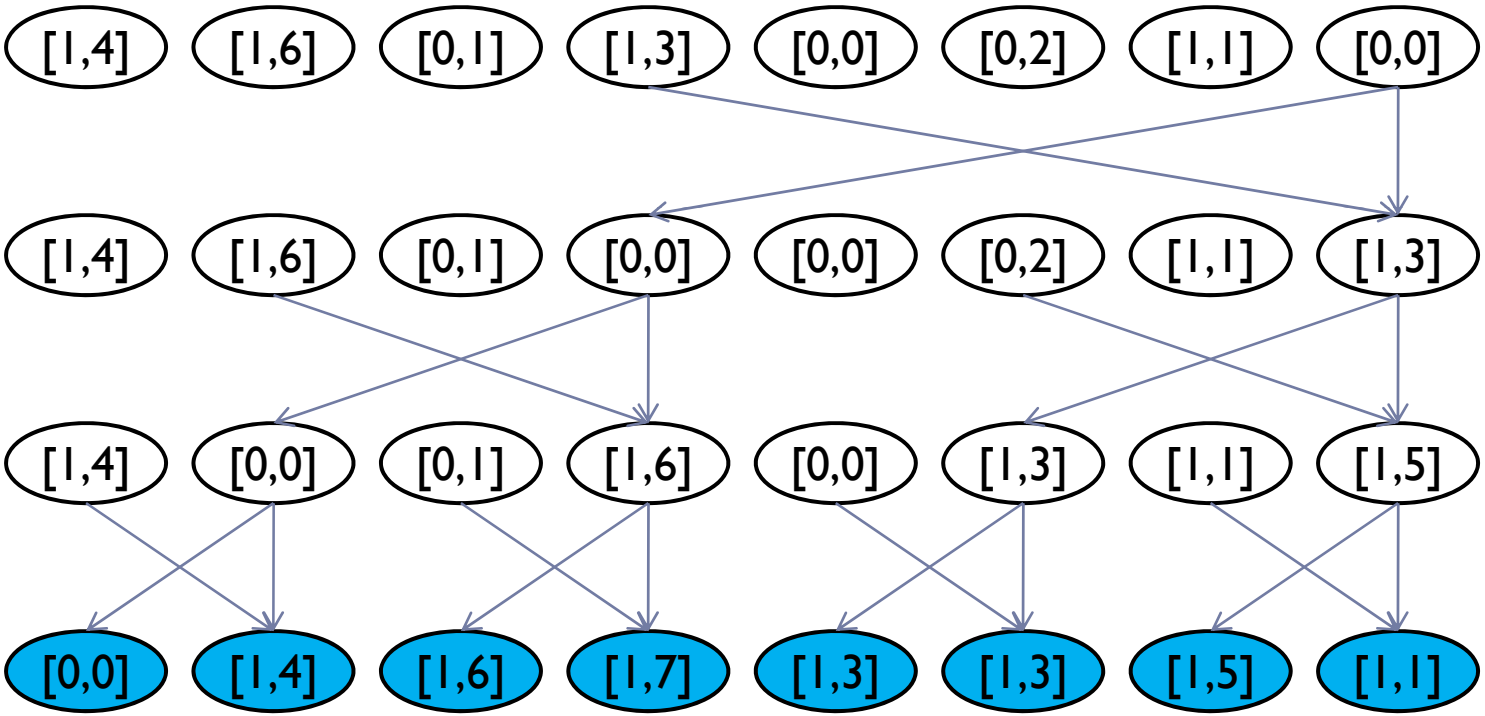
Down-sweep



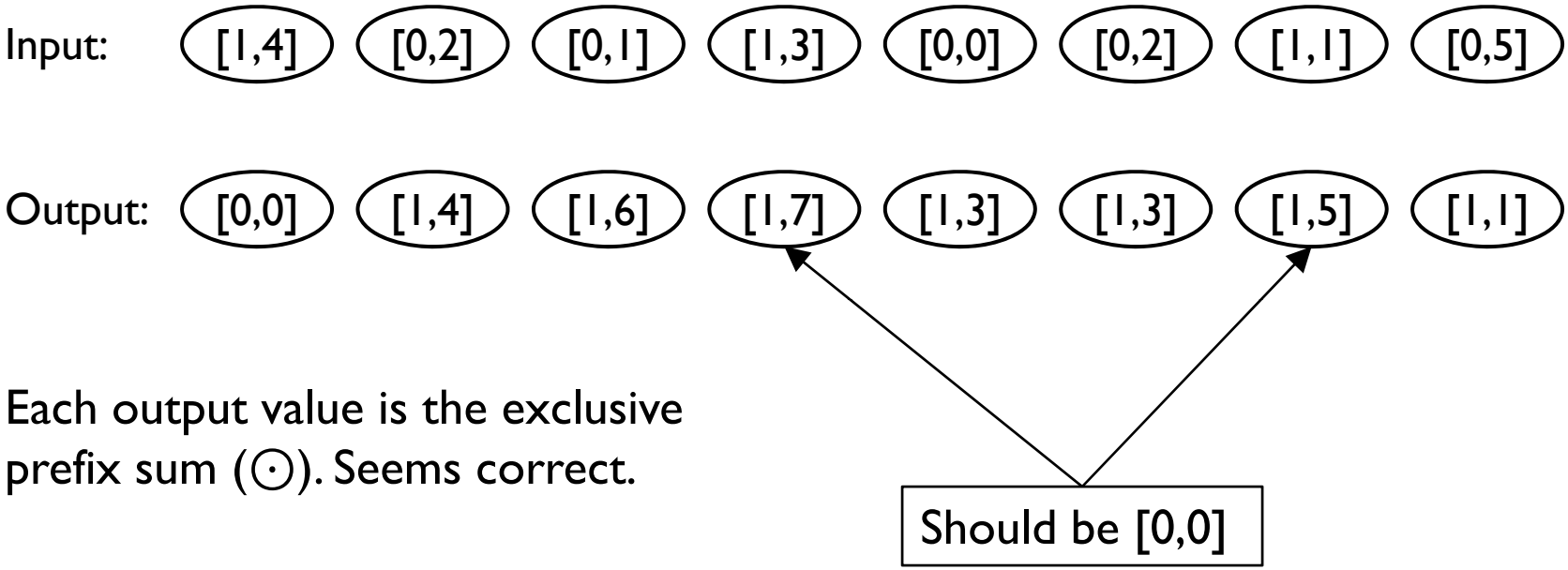
Down-sweep



Down-sweep


[1,4]
[1,6]
[0,1]
[1,3]
[0,0]
[0,2]
[1,1]
[0,0]
[1,4]
[1,6]
[0,1]
[0,0]
[0,0]
[0,2]
[1,1]
[1,3]
[1,4]
[0,0]
[0,1]
[1,6]
[0,0]
[1,3]
[1,1]
[1,5]
[0,0]
[1,4]
[1,6]
[1,7]
[1,3]
[1,3]
[1,5]
[1,1]

Now What We Get



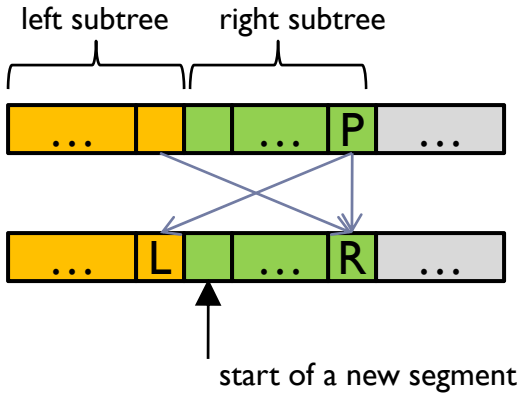
Each output value is the exclusive prefix sum ($\odot$). Seems correct.

But something is WRONG…

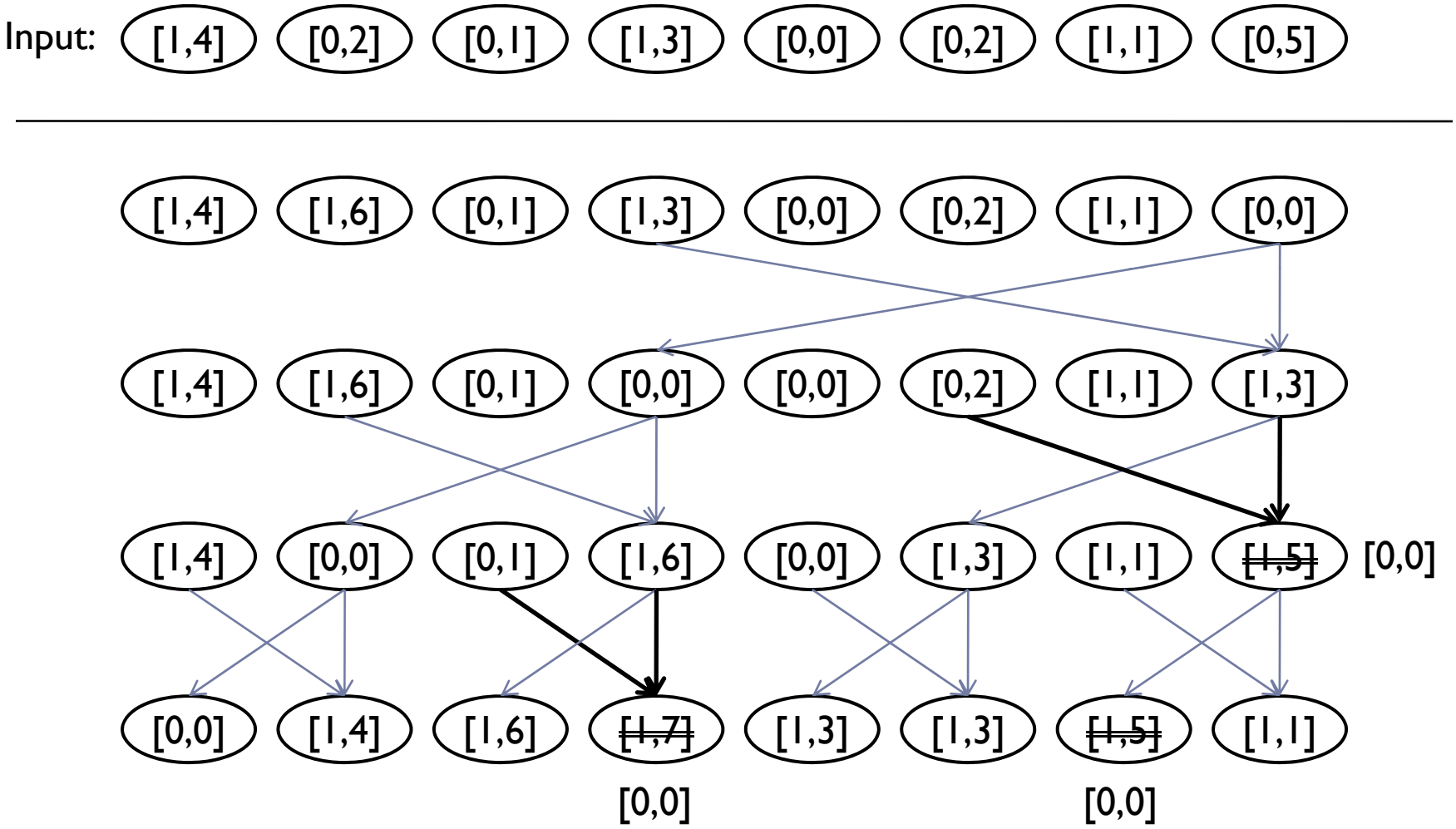
How to correct this?

- Need to "cut off" the sum across different segments.
- Note that the root is propagated along the left side the tree.
- Reset the top node of a subtree to zero when its leftmost leave is the start of a new segment, just like we reset the top root to zero at the beginning of the down-sweep.

Adjustment to down-sweep:
if the ORIGINAL flag at the position to the right of the left child is 1, the right child is set to 0.



Adjustment to Down-sweep



Segmented Scan Algorithm: Revisit

Reduce:

```
1: for d = 1 to log2 n − 1 do
2:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
3:         if f[k + 2^(d+1) − 1] is not set then
4:             x[k + 2^(d+1) − 1] ← x[k + 2^d − 1] + x[k + 2^(d+1) − 1]
5:         f[k + 2^(d+1) − 1] ← f[k + 2^d − 1] | f[k + 2^(d+1) − 1]
```

Down-sweep:

```
1: x[n − 1] ← 0
2: for d = log2 n − 1 down to 0 do
3:     for all k = 0 to n − 1 by 2^(d+1) in parallel do
4:         t ← x[k + 2^d − 1]
5:         x[k + 2^d − 1] ← x[k + 2^(d+1) − 1]
6:         if f[k + 2^d] is set then
7:             x[k + 2^(d+1) − 1] ← 0
8:         else if f[k + 2^d − 1] is set then
9:             x[k + 2^(d+1) − 1] ← t
10:        else
11:            x[k + 2^(d+1) − 1] ← t + x[k + 2^(d+1) − 1]
12:        Unset flag f[k + 2^d − 1]
```

Adjustment

Scan over (flag, data) pairs

Agenda

- What is scan
- A naïve parallel scan algorithm
- A work-efficient parallel scan algorithm
- Parallel segmented scan
- **Applications of scan**
- Implementation on CUDA

Application

- Other Primitives
  - Enumerate
  - Distribute
  - Split
- Radix Sort
- Quick Sort
- Recurrence Equations
- Sparse Matrix-Vector Multiply
- Tridiagonal Matrix Solver
- Multi-precision addition
- …

Split

- Data: [5 7 3 1 4 2 7 2]
- Flag: [1 1 0 1 0 0 1 0]
- Result: [5 7 1 7 3 4 2 2]
- Steps:

[1 1 0 1 0 0 1 0]	# e
[0 1 2 2 3 3 3 4]	# f = scan over e
4	# NF = add last elements of e and f
[0 1 2 3 4 5 6 7]	# id
[4 4 4 5 5 6 7 7]	# t = id – f + NF
[0 1 4 2 5 6 3 7]	# addr = e ? f : t
[5 7 1 7 3 4 2 2]	# out[addr] = in

Split (hands-on)

- Data: [1 5 6 2 3 7 8 4]
- Flag: [1 0 0 1 1 0 0 1]
- Result: [ ]
- Steps: [1 0 0 1 1 0 0 1] # e

[ ] # f = scan over e

NF = add last elements of e and f

[ ] # id

[ ] # t = id – f + NF

[ ] # addr = e ? f : t

[ ] # out[addr] = in

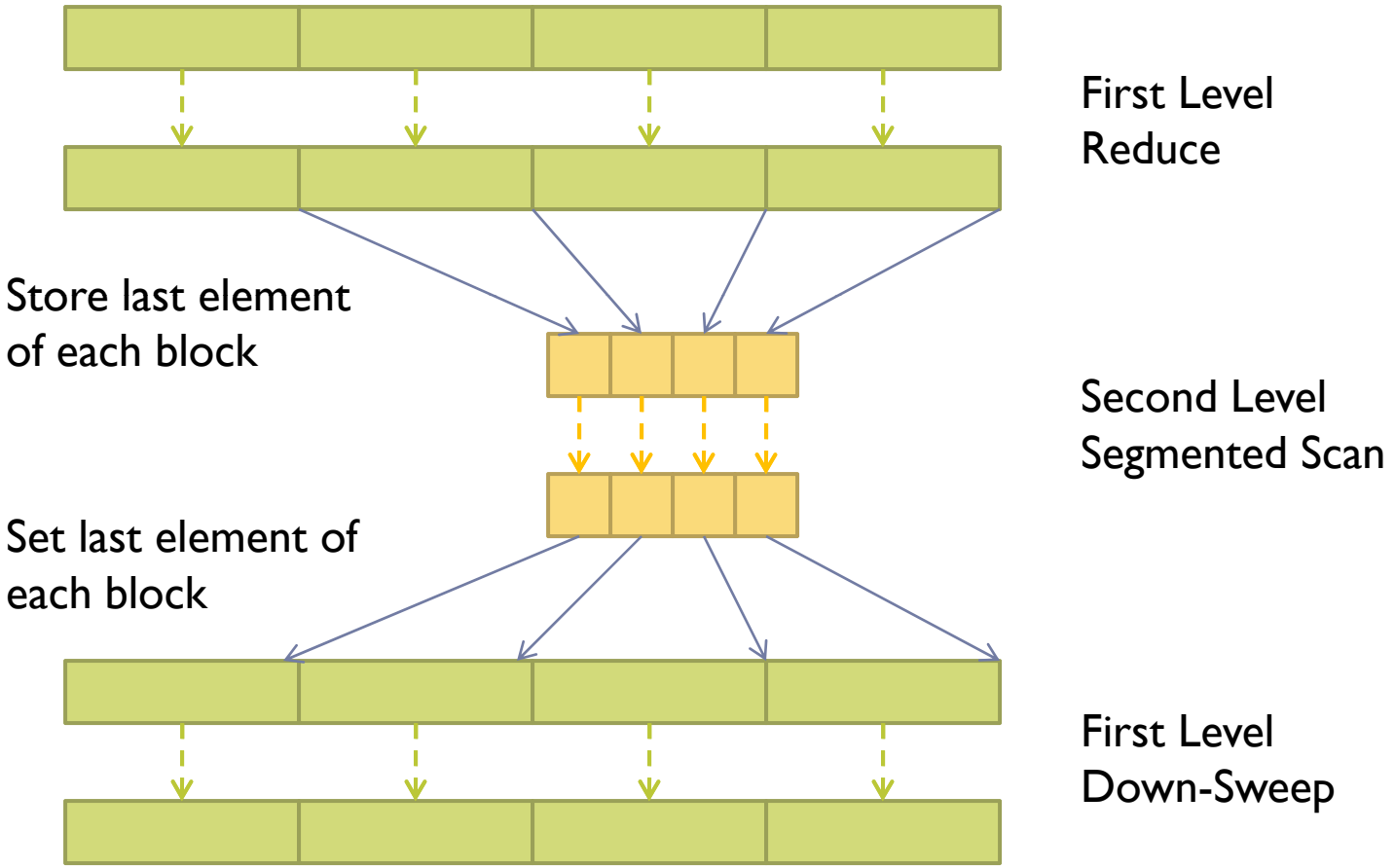
Agenda

- What is scan
- A naïve parallel scan algorithm
- A work-efficient parallel scan algorithm
- Parallel segmented scan
- Applications of scan
- **Implementation on CUDA**

Implementation on CUDA

- All the threads should be in the same thread block so that they can share memory and synchronize with each other.
- On G80 GPUs, this limits us to a maximum of 1024 elements.
- Extend to large arrays by dividing the array into blocks. Each block is scanned by a single thread block.

Multi-block Segmented Scan



CUDA Source codes for scan

```
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern  __shared__  float temp[];// allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

A   temp[2*thid]   = g_idata[2*thid]; // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d)
        {
B           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }
        offset *= 2;
    }

C   if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
D           int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t    = temp[ai];
            temp[ai]   = temp[bi];
            temp[bi] += t;
        }
    }

    __syncthreads();

E   g_odata[2*thid]   = temp[2*thid]; // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
```