Parallel Algorithms

Shehzan Mohammed

CIS 5650 - Fall 2024

Parallel Reduction

Agenda - Parallel Algorithms

- Parallel Reduction
- Scan (Naive and Work Efficient)
- Applications of Scan
  - Stream Compaction
  - Summed Area Tables (SAT)
  - Radix Sort

Parallel Reduction

- Given an array of numbers, design a parallel algorithm to find the sum.
- Consider:
  - **Arithmetic intensity**: compute to memory access ratio

Parallel Reduction

- Given an array of numbers, design a parallel algorithm to find:
  - The sum
  - The maximum value
  - The product of values
  - The average value
- How different are these algorithms?

Parallel Reduction

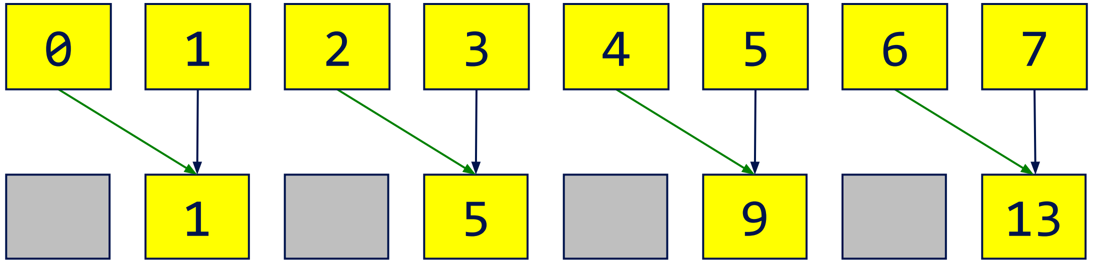
- ***Reduction***: An operation that computes a single result from a set of data
- ***Parallel Reduction***: Do it in parallel.

Parallel Reduction

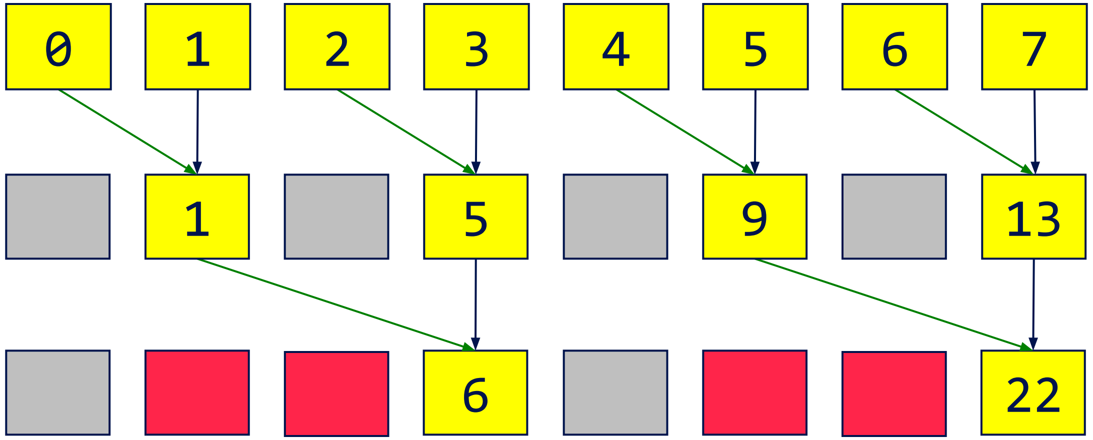
- Example: Find the sum:

0	1	2	3	4	5	6	7

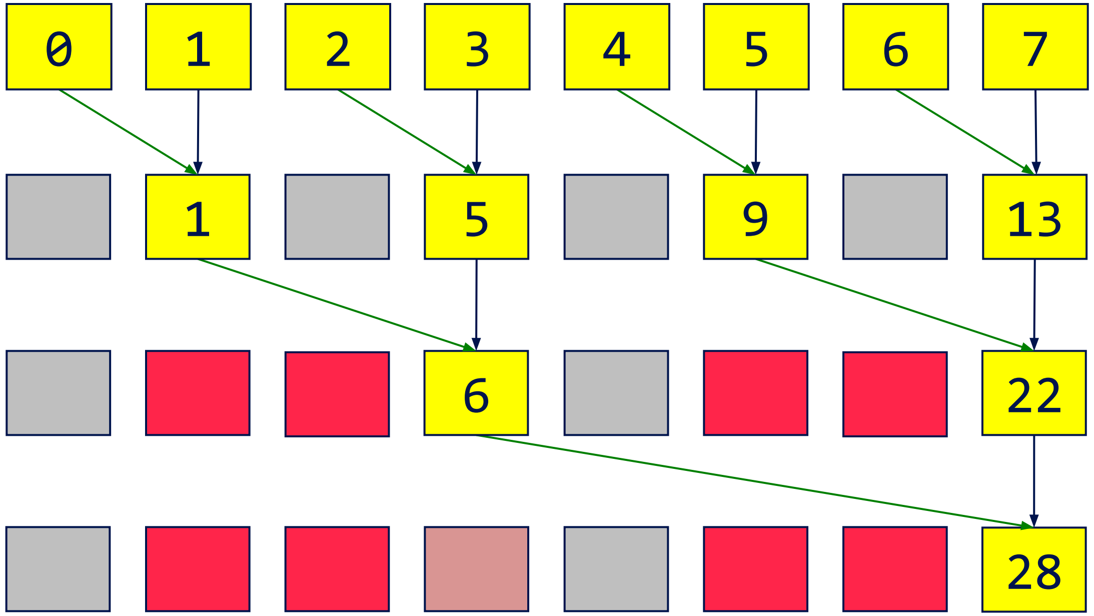
Parallel Reduction



Parallel Reduction



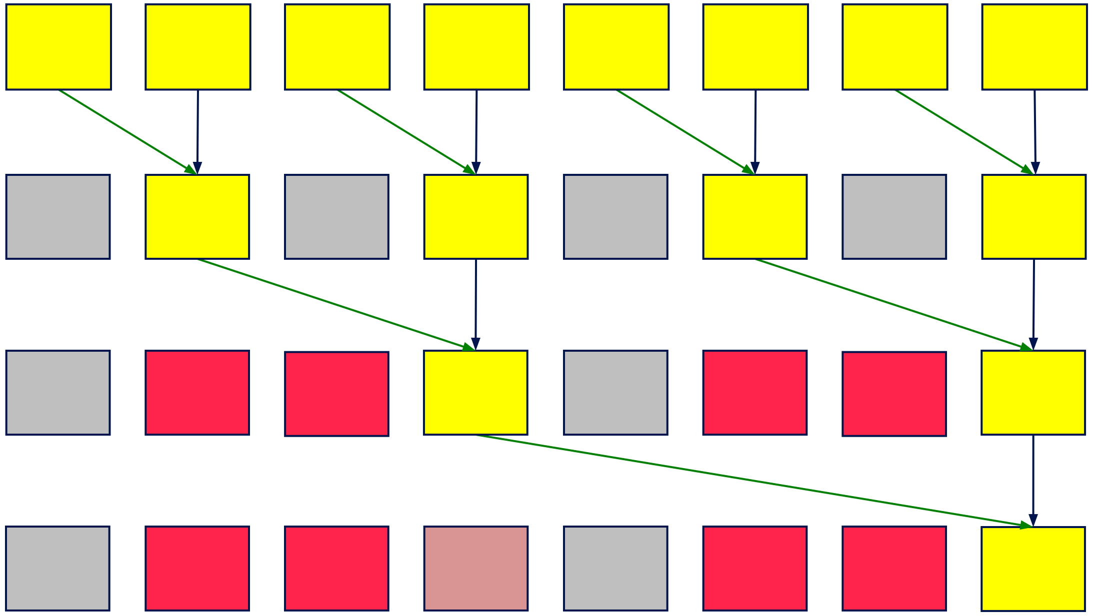
Parallel Reduction



Penn Engineering

Parallel Reduction

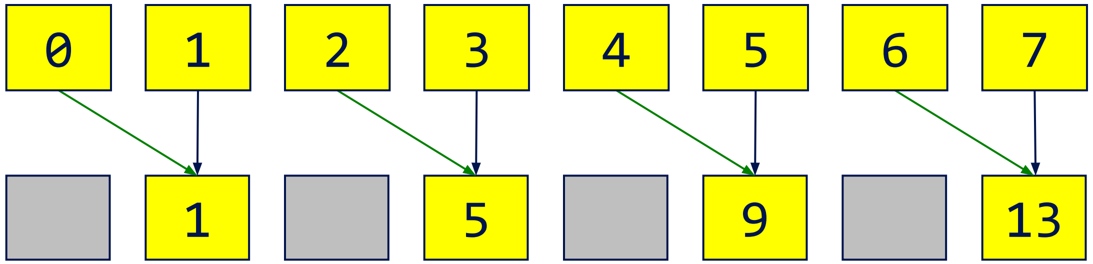
- Similar to brackets for a basketball tournament
- log(n) passes for n elements



Parallel Reduction

- $d = 0, 2^{d+1} = 2$
- $2^{d+1} - 1 = 1$
- $2^d - 1 = 0$

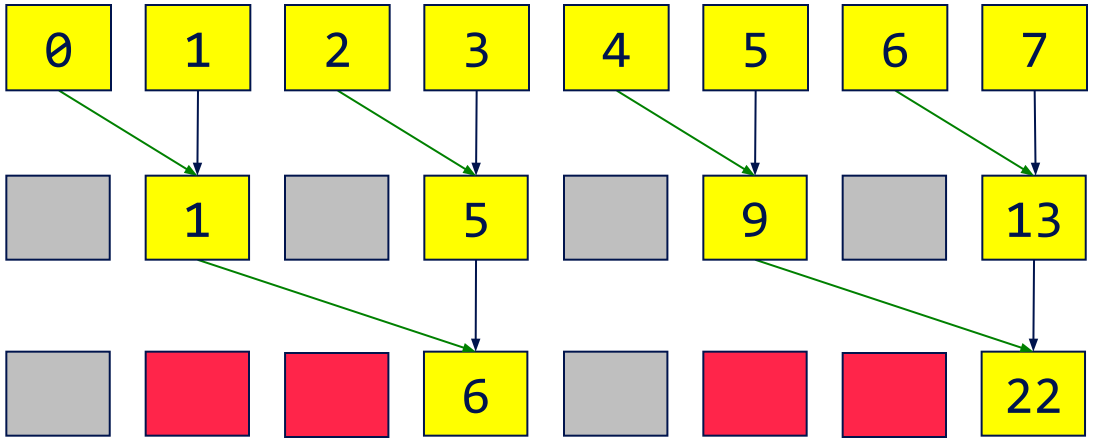
```
for d = 0 to log₂n - 1
  for all k = 0 to n – 1 by 2^(d+1) in parallel
    x[k + 2^(d+1) – 1] += x[k + 2^d – 1];

// In this pass, for k = (0, 2, 4, 6)
//   x[k + 1] += x[k];
```



Parallel Reduction

- $d = 1,\ 2^{d+1} = 4$
- $2^{d+1} - 1 = 3$
- $2^d - 1 = 1$

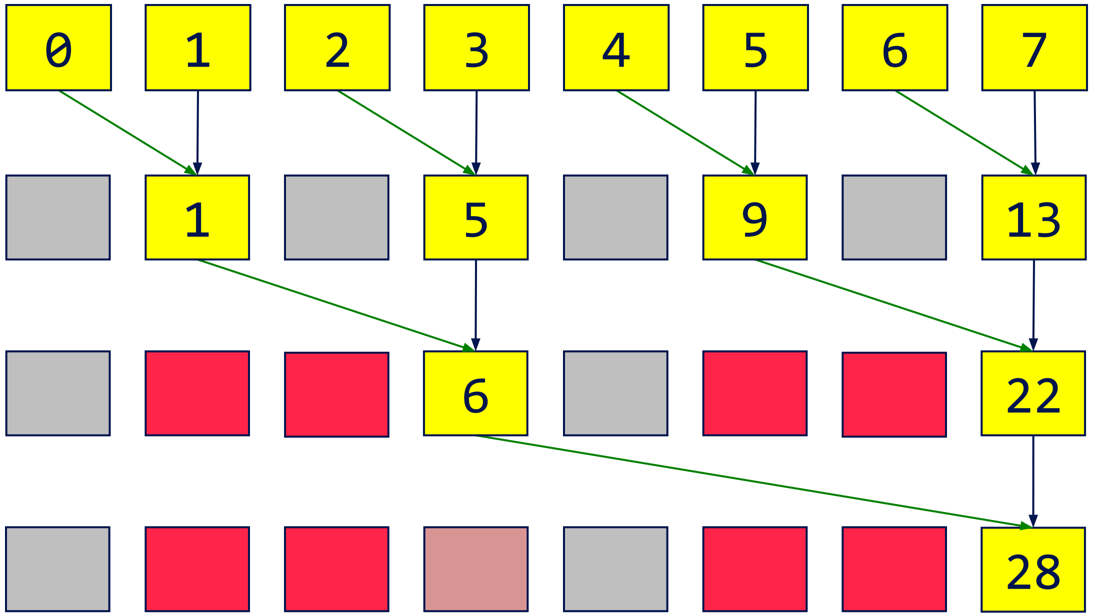
```
for d = 0 to log2n - 1
  for all k = 0 to n – 1 by 2^(d+1) in parallel
    x[k + 2^(d+1) – 1] += x[k + 2^d – 1];

// In this pass, for k = (0, 4)
//   x[k + 3] += x[k + 1];
```



Parallel Reduction

- $d = 2,\ 2^{d+1} = 8$
- $2^{d+1} - 1 = 7$
- $2^{d} - 1 = 3$

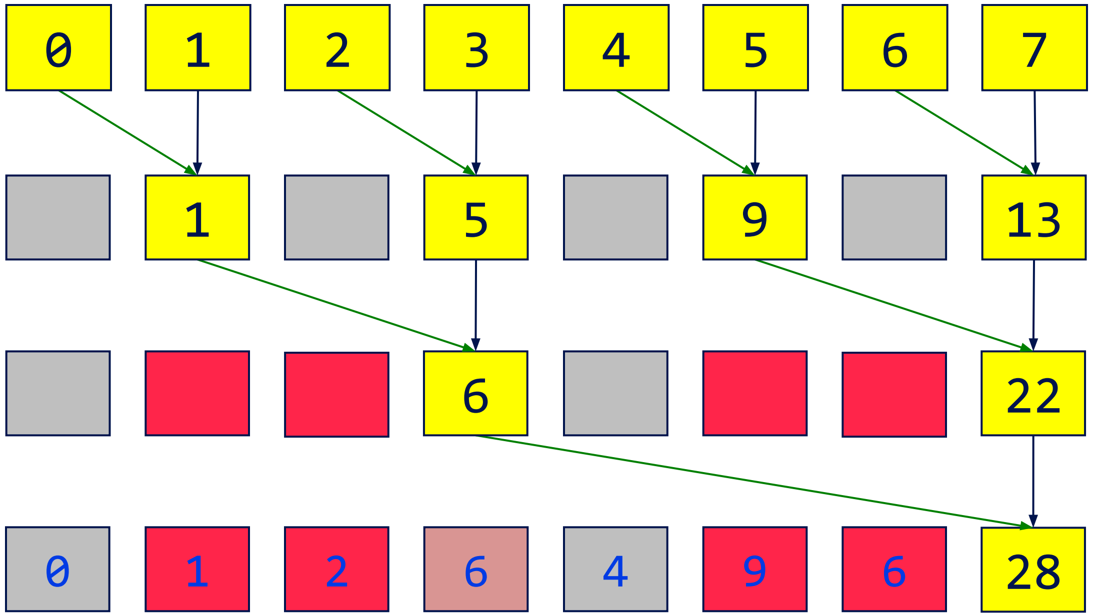
```
for d = 0 to log₂n - 1
  for all k = 0 to n – 1 by 2^(d+1) in parallel
    x[k + 2^(d+1) – 1] += x[k + 2^d – 1];

// In this pass, for k = (0)
//   x[k + 7] += x[k + 3];
```



Parallel Reduction

- Note the +=
- The array is modified in place

```
for d = 0 to log2n - 1
  for all k = 0 to n – 1 by 2^(d+1) in parallel
    x[k + 2^(d+1) – 1] += x[k + 2^d – 1]
```



Penn Engineering

Scan

All-Prefix-Sums

- All-Prefix-Sums
- Input
  - Array of n elements: `[a0, a1, a2,........, an-1]`
  - Binary associate operator: $\oplus$
  - Identity: `I`
- Outputs the array:
  $[I, a_0, (a_0 \oplus a_1), (a_0 \oplus a_1 \oplus a_2) \ldots\ldots, (a_0 \oplus a_1 \oplus a_2 \oplus a_{n-2})]$

http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

All-Prefix-Sums

- Example
  - If ⊕ is addition, the array
    - `[3 1 7 0 4 1 6 3]`
  - is transformed to
    - `[0 3 4 11 11 15 16 22]`
- Seems sequential, but there is an efficient parallel solution

Scan

- *Exclusive Scan*: Element *j* of the result does not include element *j* of the input:
  - `In:  [ 3  1  7  0  4  1  6  3]`
  - `Out: [ 0  3  4 11 11 15 16 22]`

- *Inclusive Scan* (*Prescan*): All elements including *j* are summed
  - `In:  [ 3  1  7  0  4  1  6  3]`
  - `Out: [ 3  4 11 11 15 16 22 25]`

Scan

- How do you generate an `exclusive scan` from an `inclusive scan`?
  - `Input:     [ 3  1  7  0  4  1  6  3]`
  - `Inclusive: [ 3  4 11 11 15 16 22 25]`
  - `Exclusive: [ 0  3  4 11 11 15 16 22]`
  - `// Shift right, insert identity`

- How do you go in the opposite direction?

Scan

- Design a parallel algorithm for `Inclusive Scan`
  - `In:  [ 3  1  7  0  4  1  6  3]`
  - `Out: [ 3  4 11 11 15 16 22 25]`

- Consider:
  - Total number of additions

Scan

- Single thread is straightforward

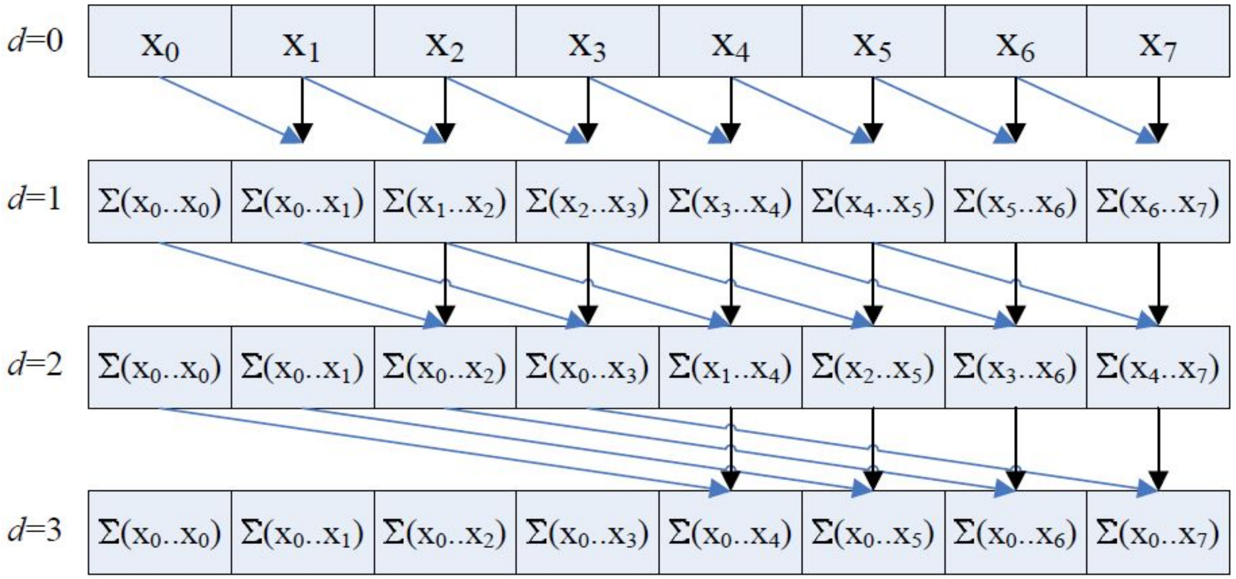
```
out[0] = in[0]; // assuming n > 0
for (int k = 1; k < n; ++k)
  out[k] = out[k – 1] + in[k];
```

- *n - 1* adds for an array of length *n*
  - (ignoring array indices)

- How many adds will our parallel version have?

Scan

- Naive Parallel Scan




```
for d = 1 to log_2 n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Is this exclusive or inclusive?
- Each thread
  - Writes one sum
  - Reads two values

Image from http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf

Scan

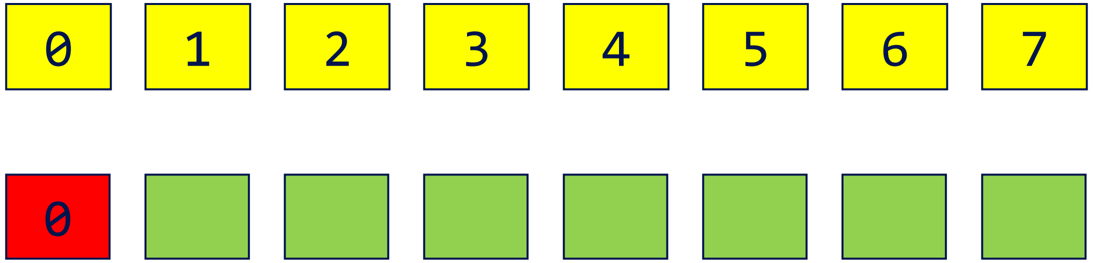
- Naive Parallel Scan: Input

0	1	2	3	4	5	6	7

Scan

- Naive Parallel Scan: `d = 1, 2^(d-1) = 1`

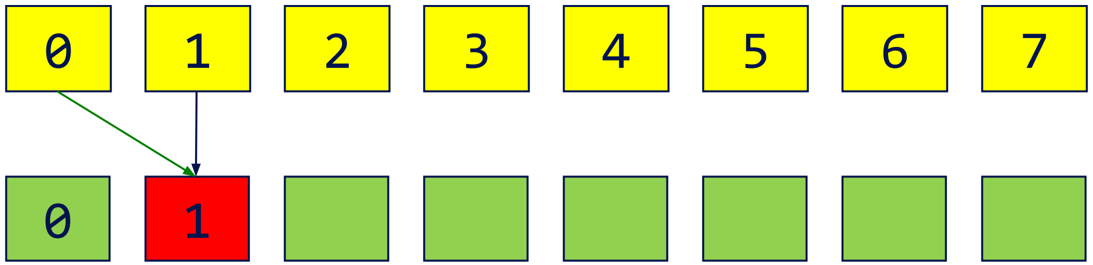
```
for d = 1 to log_2 n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

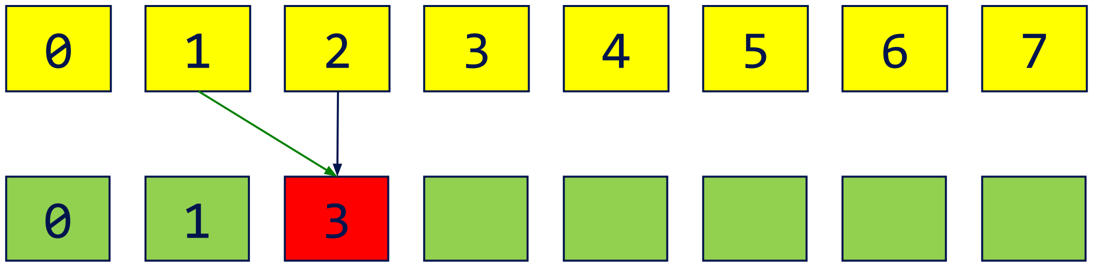
- Naive Parallel Scan: $d = 1,\ 2^{d-1} = 1$



Scan

- Naive Parallel Scan: $d = 1$, $2^{d-1} = 1$

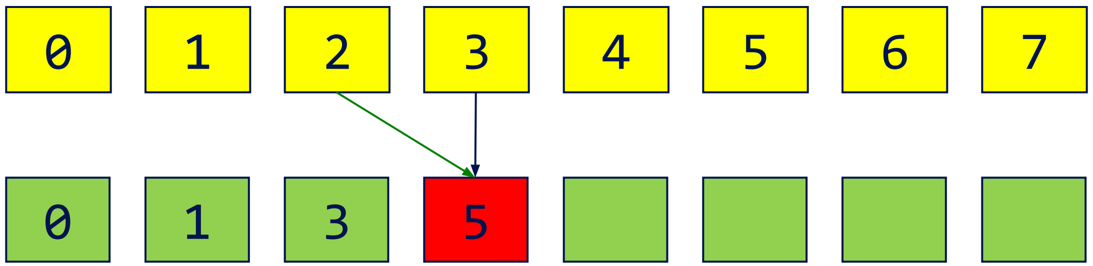
```
for d = 1 to log2 n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

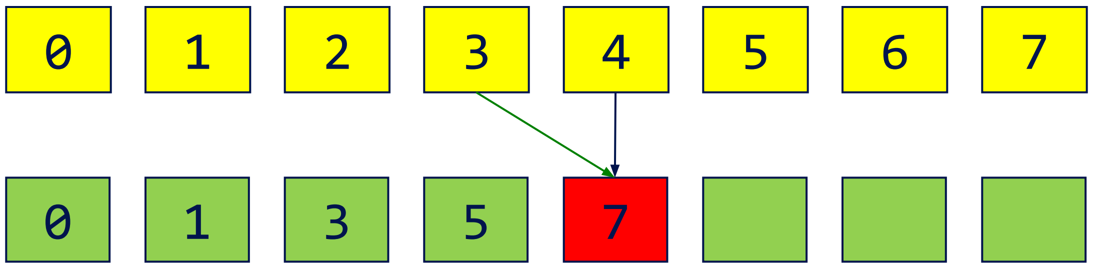
- Naive Parallel Scan: $d = 1,\ 2^{d-1} = 1$



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Naive Parallel Scan: $d = 1,\ 2^{d-1} = 1$



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

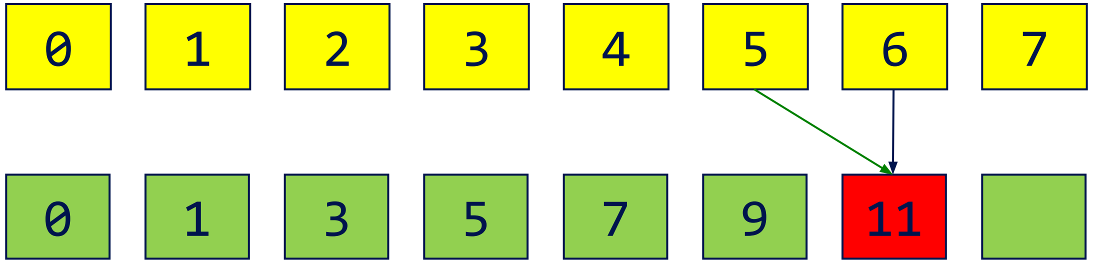
- Naive Parallel Scan: $d = 1,\ 2^{d-1} = 1$

0	1	2	3	4	5	6	7
0	1	3	5	7	9		

Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

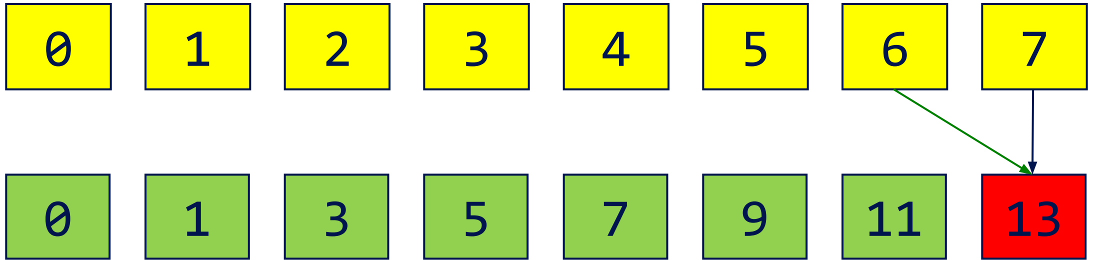
- Naive Parallel Scan: $d = 1,\ 2^{d-1} = 1$



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

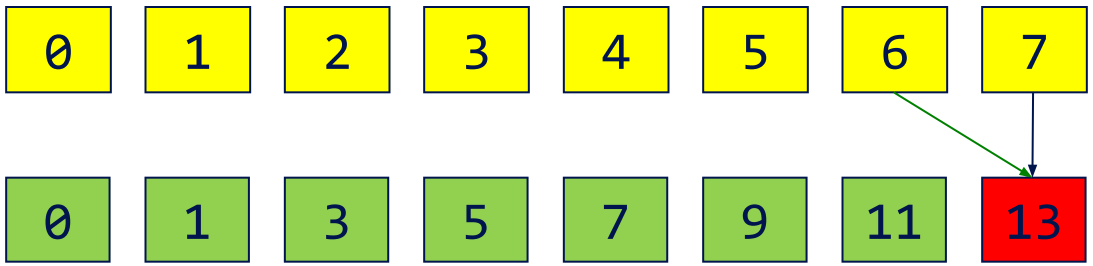
- Naive Parallel Scan: $d = 1,\ 2^{d-1} = 1$



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Naive Parallel Scan: d $= 1,\ 2^{d-1} = 1$

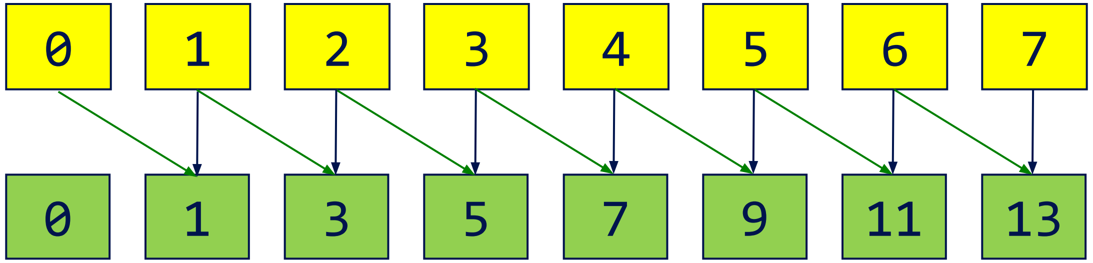


- But remember, it runs in parallel!

Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Naive Parallel Scan: $d = 1,\ 2^{d-1} = 1$

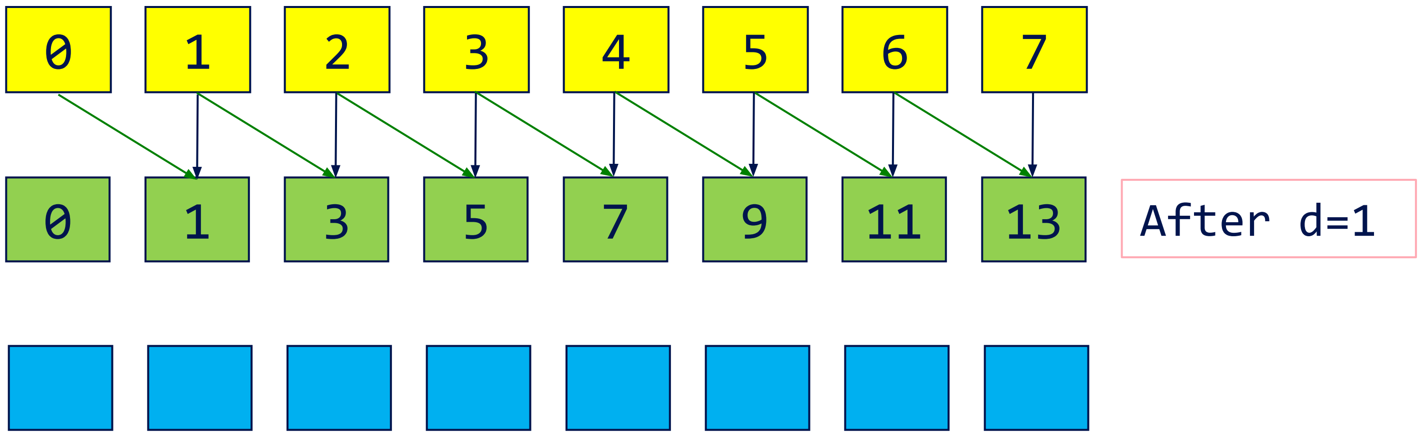


- But remember, it runs in parallel!

Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k - 2^(d-1)] + x[k];
```

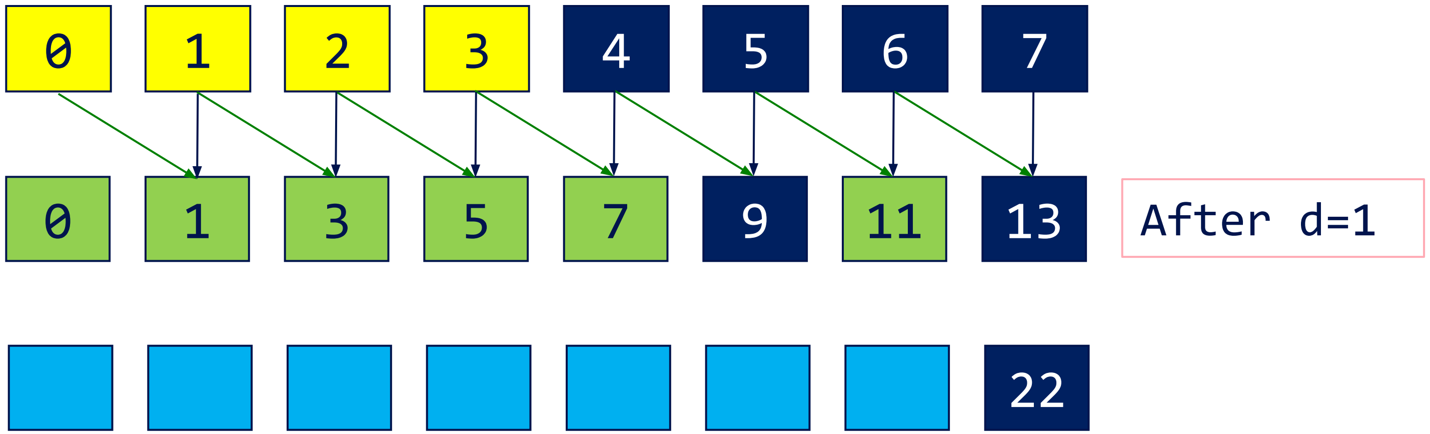
- Naive Parallel Scan: $d = 2,\ 2^{d-1} = 2$



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Naive Parallel Scan: $d = 2,\ 2^{d-1} = 2$



- Consider k=7

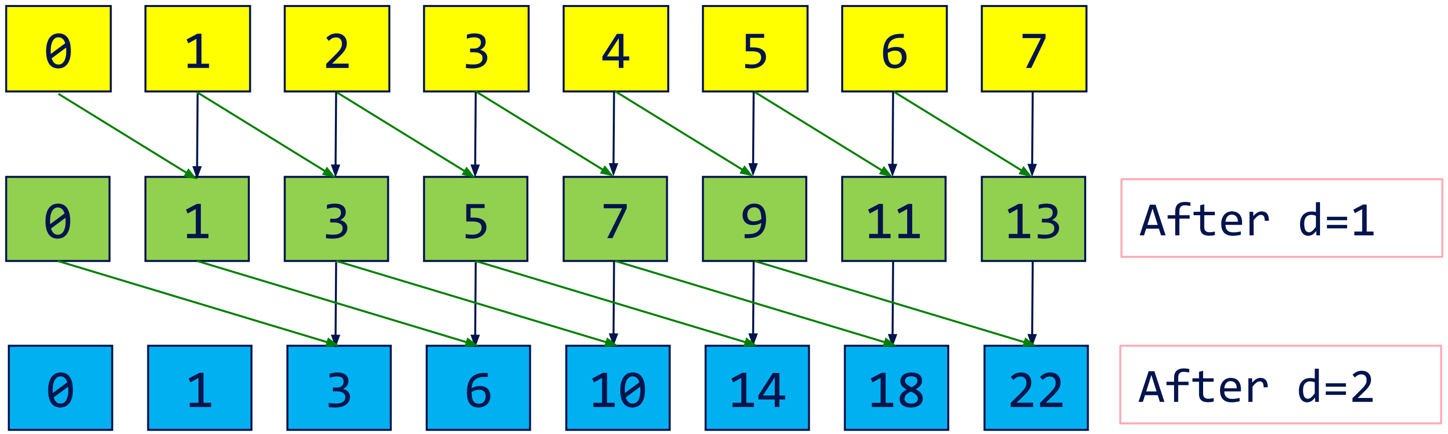
if (7 >= 2^2-1)

x[7] = x[7 - 2^2-1] + x[7]

Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Naive Parallel Scan: $d = 2,\ 2^{d-1} = 2$



- Consider k=7

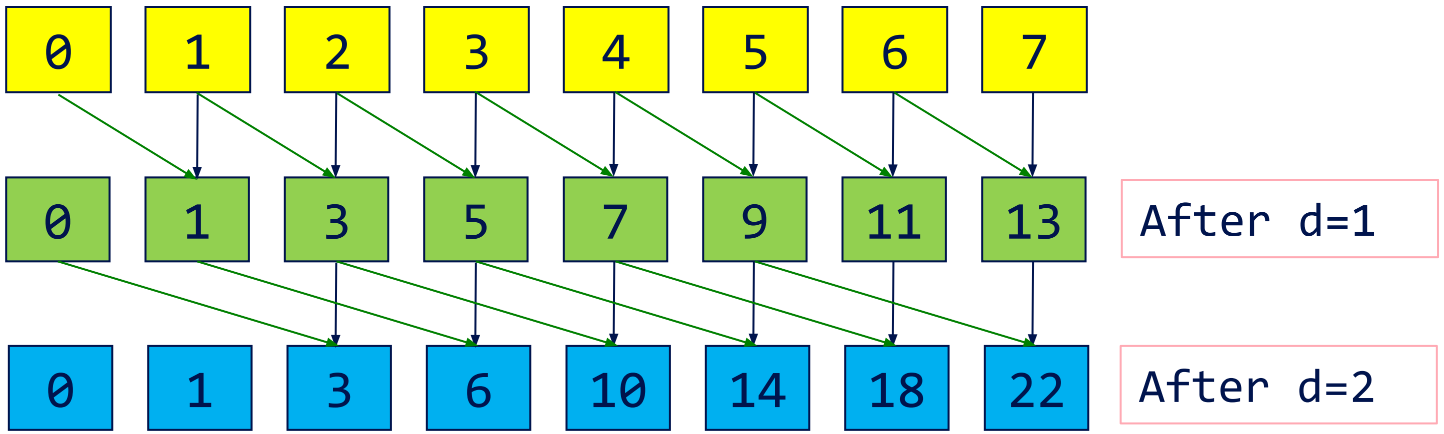
if (7 >= 2^(2-1))

x[7] = x[7 - 2^(2-1)] + x[7]

Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

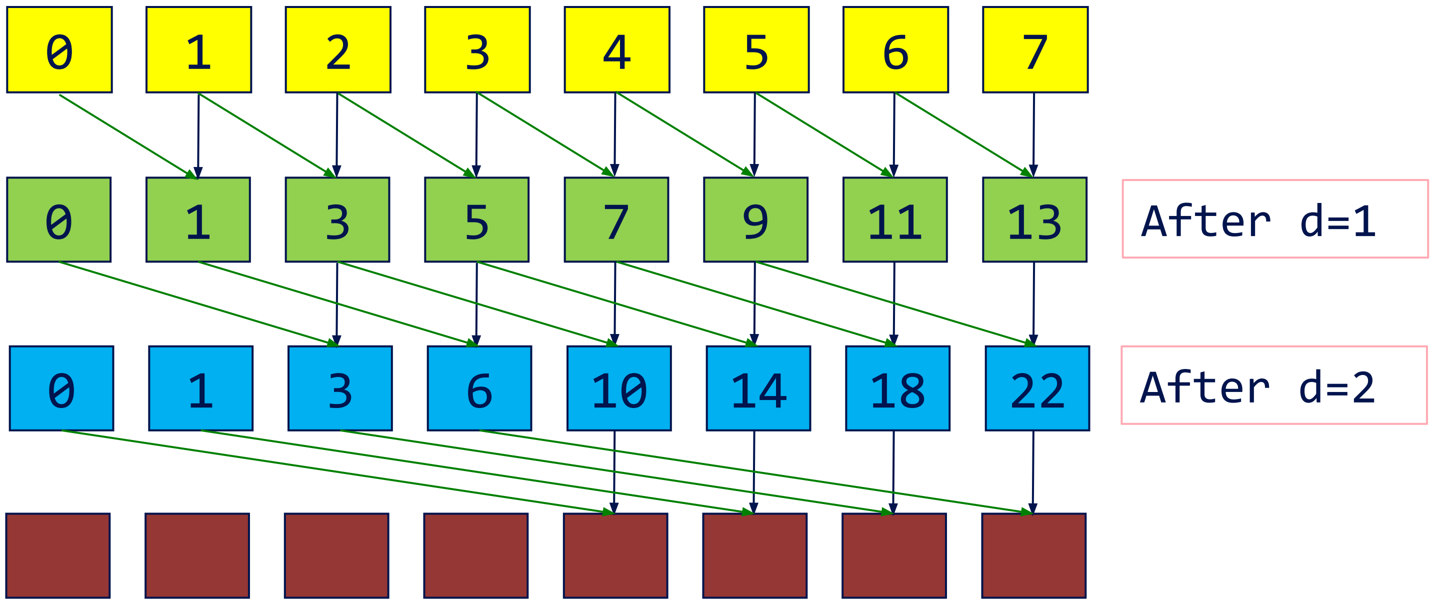
- Naive Parallel Scan: $d = 2,\ 2^{d-1} = 2$



Scan

```
for d = 1 to log₂n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

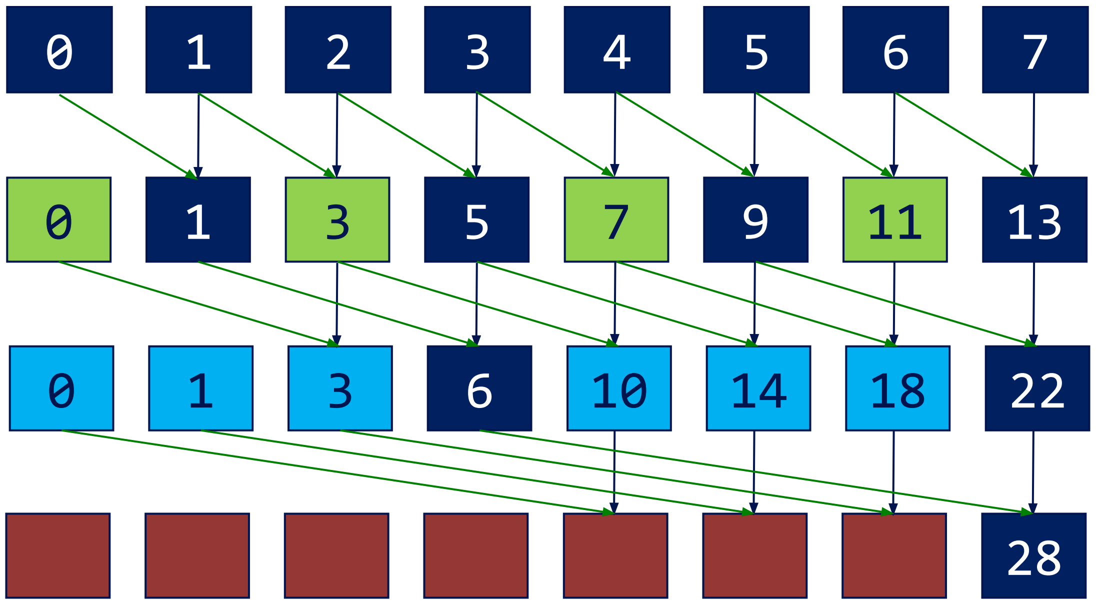
- Naive Parallel Scan: $d = 3,\ 2^{d-1} = 4$



Scan

```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```

- Naive Parallel Scan: $d = 3,\ 2^{d-1} = 4$



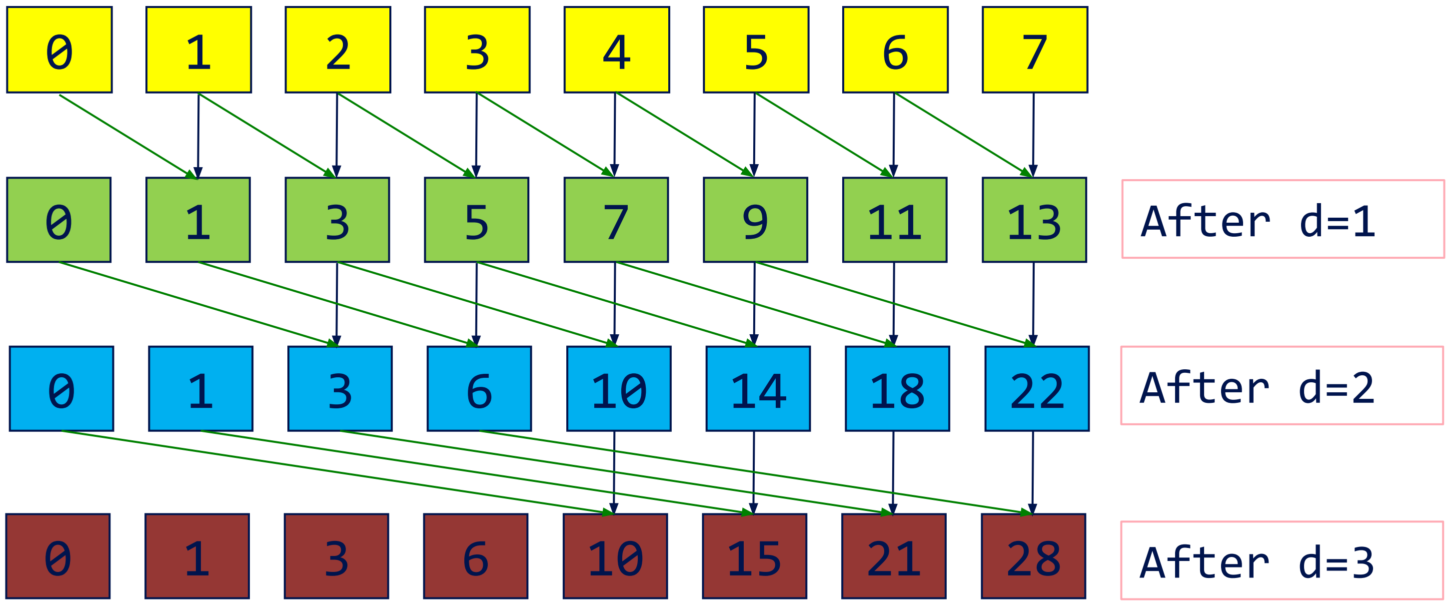
- Consider k=7

if (7 >= 2^(3-1))
  x[7] = x[7 - 2^(3-1)] + x[7]

Scan

- Naive Parallel Scan: Final

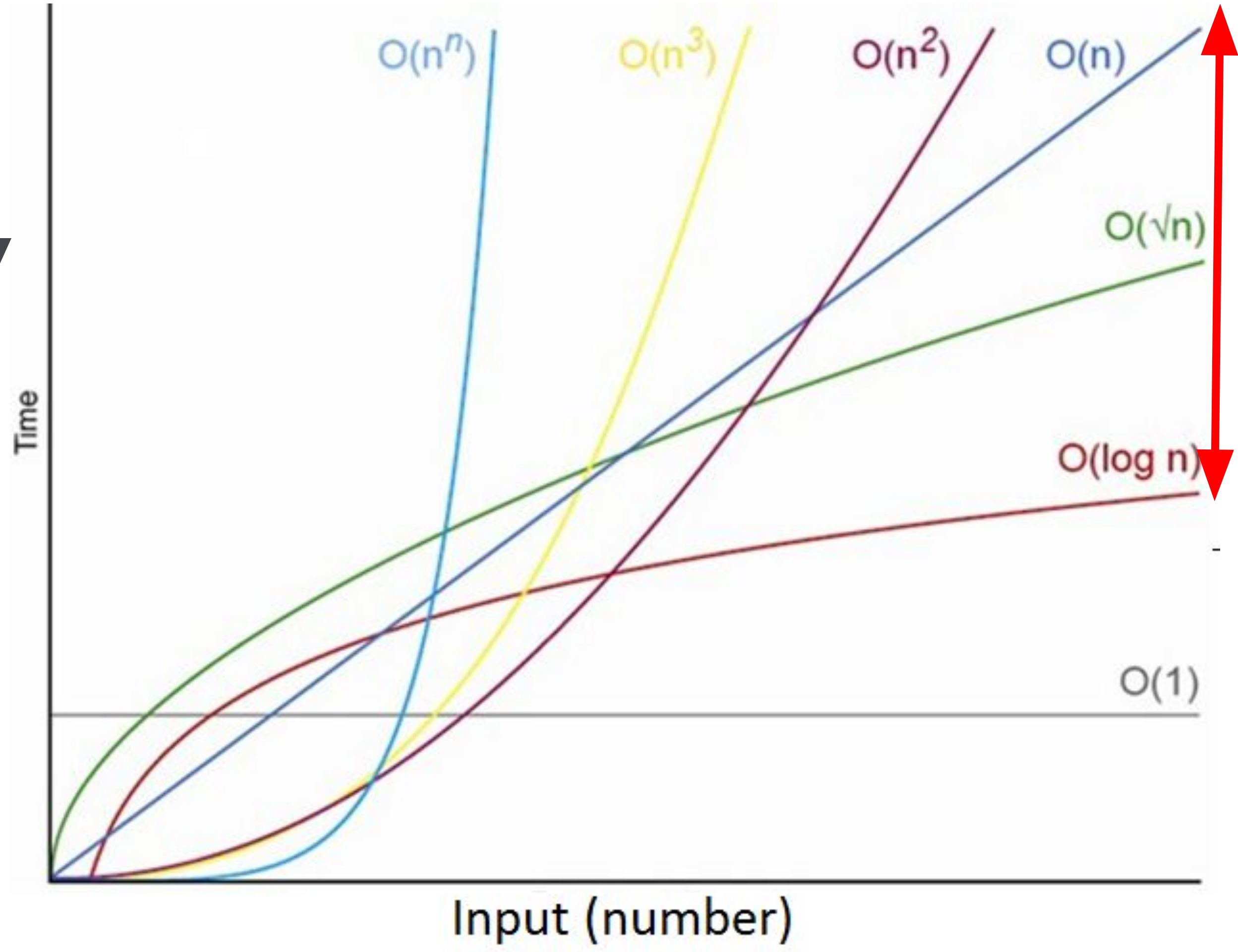
```
for d = 1 to log2n
  for all k in parallel
    if (k >= 2^(d-1))
      x[k] = x[k – 2^(d-1)] + x[k];
```



Penn Engineering

Naive Parallel Scan

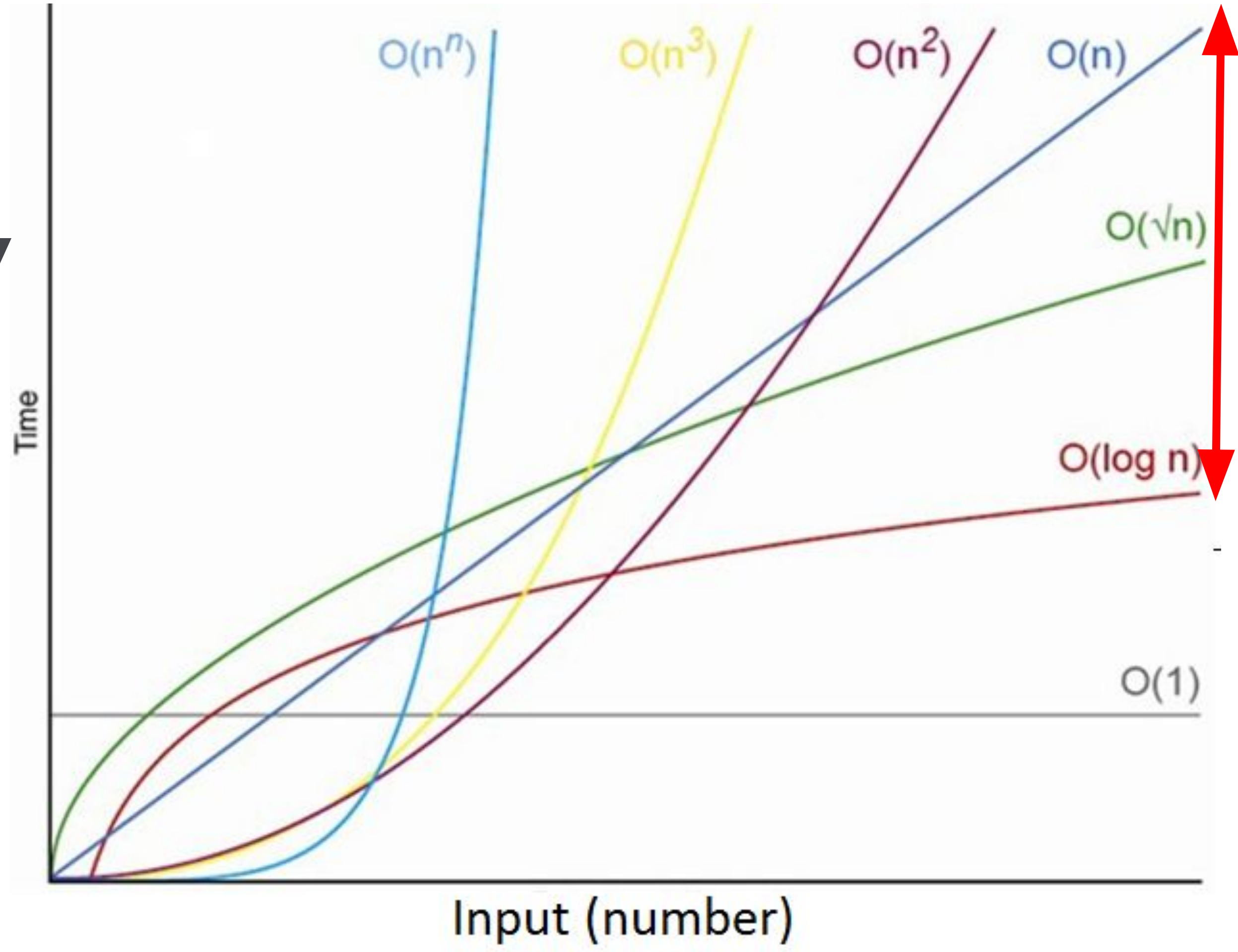
- **Number of adds**
  - Sequential Scan: O(n)
  - Naive Parallel Scan: O(nlog2(n))
- **Algorithmic Complexity**
  - Sequential Scan: O(n)
  - Naive Parallel Scan: O(log2(n))



Penn Engineering

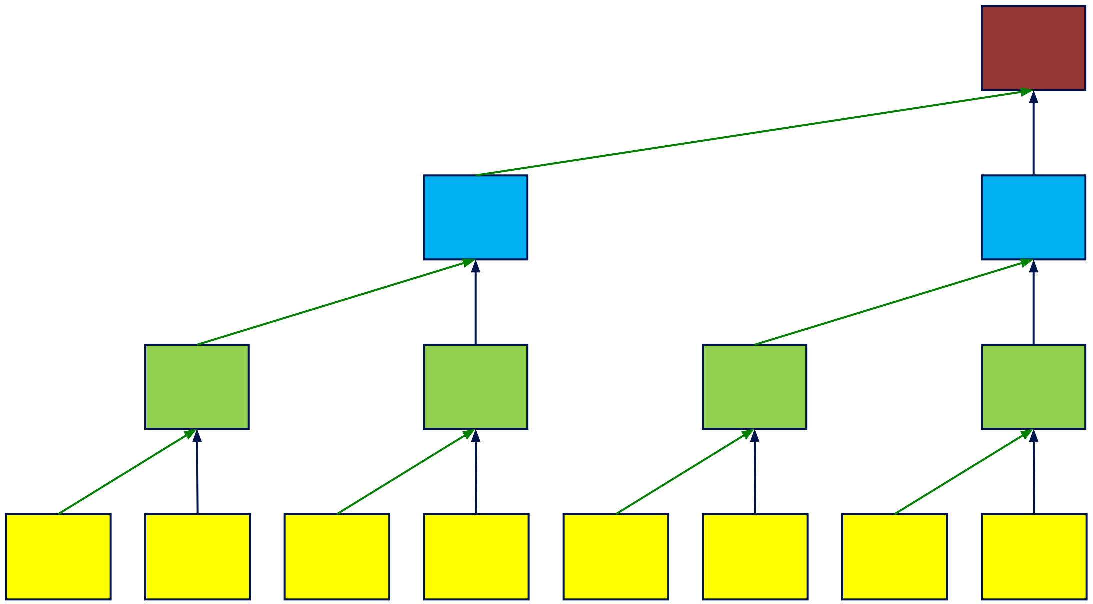
Naive Parallel Scan

- **Number of adds**
  - Sequential Scan: O(n)
  - Naive Parallel Scan: O(nlog2(n))
- **Algorithmic Complexity**
  - Sequential Scan: O(n)
  - Naive Parallel Scan: O(log2(n))
- Can we make it faster?



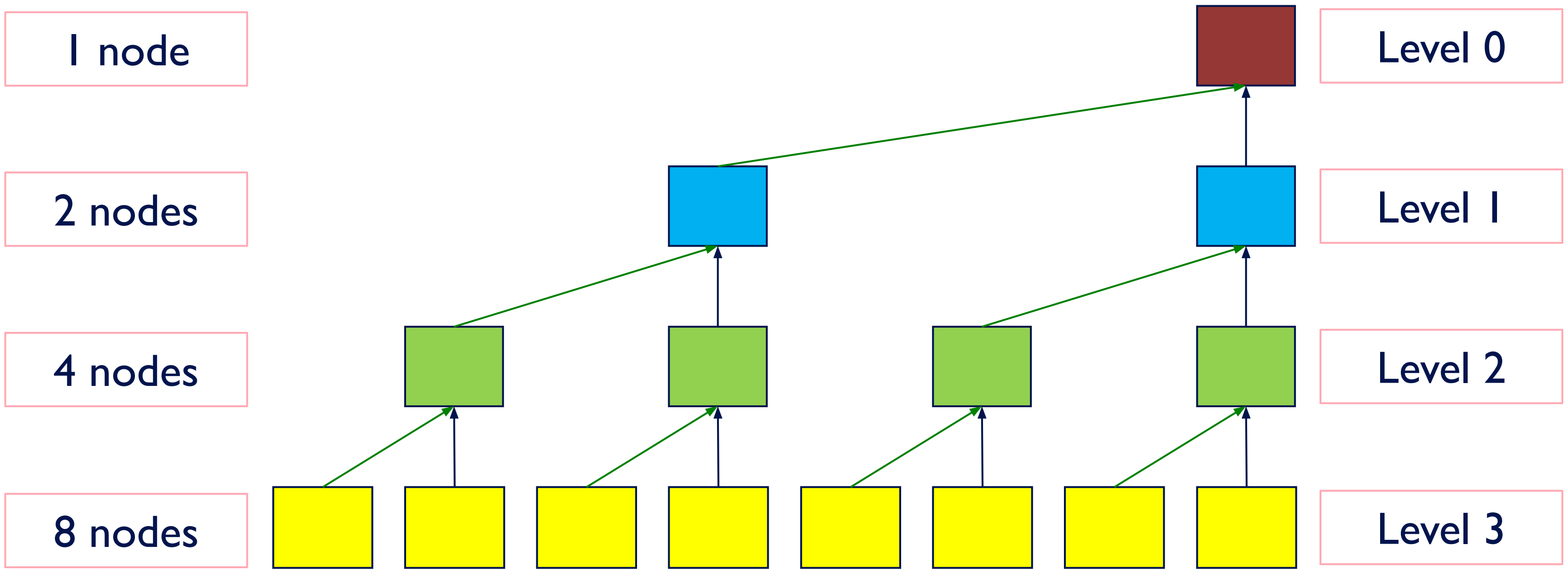
Work-Efficient Parallel Scan

- **Balanced binary tree**
  - $\mathbf{n}$ leafs = $\mathbf{log_2 n}$ levels
  - Each level, $\mathbf{d}$, has $\mathbf{2^d}$ nodes



Work-Efficient Parallel Scan

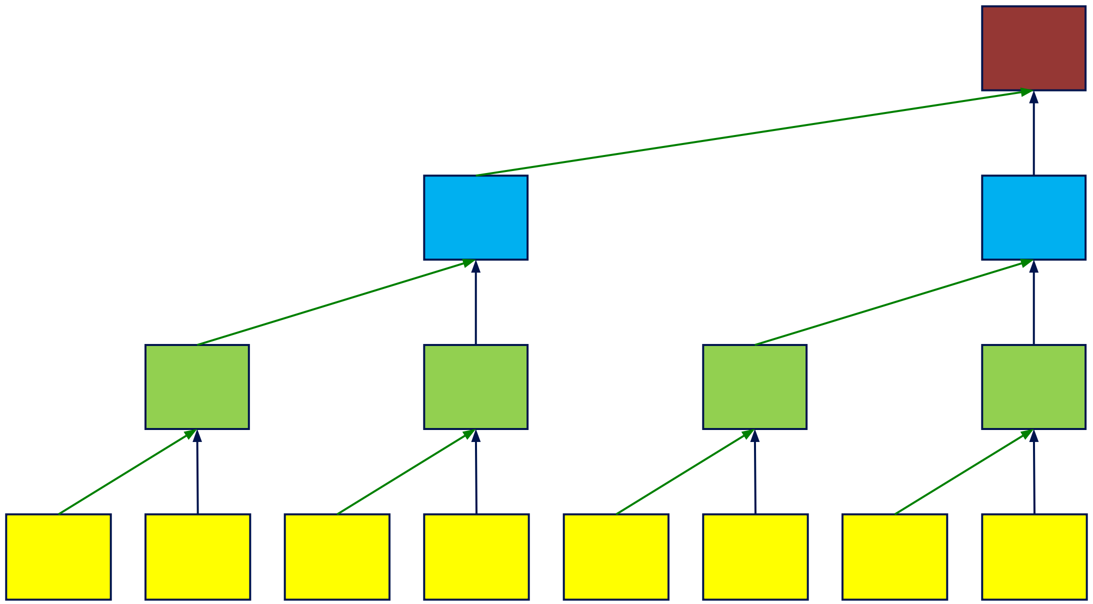
- **Balanced binary tree**
  - n leafs = $\log_2 n$ levels
  - Each level, d, has 2^d nodes



Penn Engineering

Work-Efficient Parallel Scan

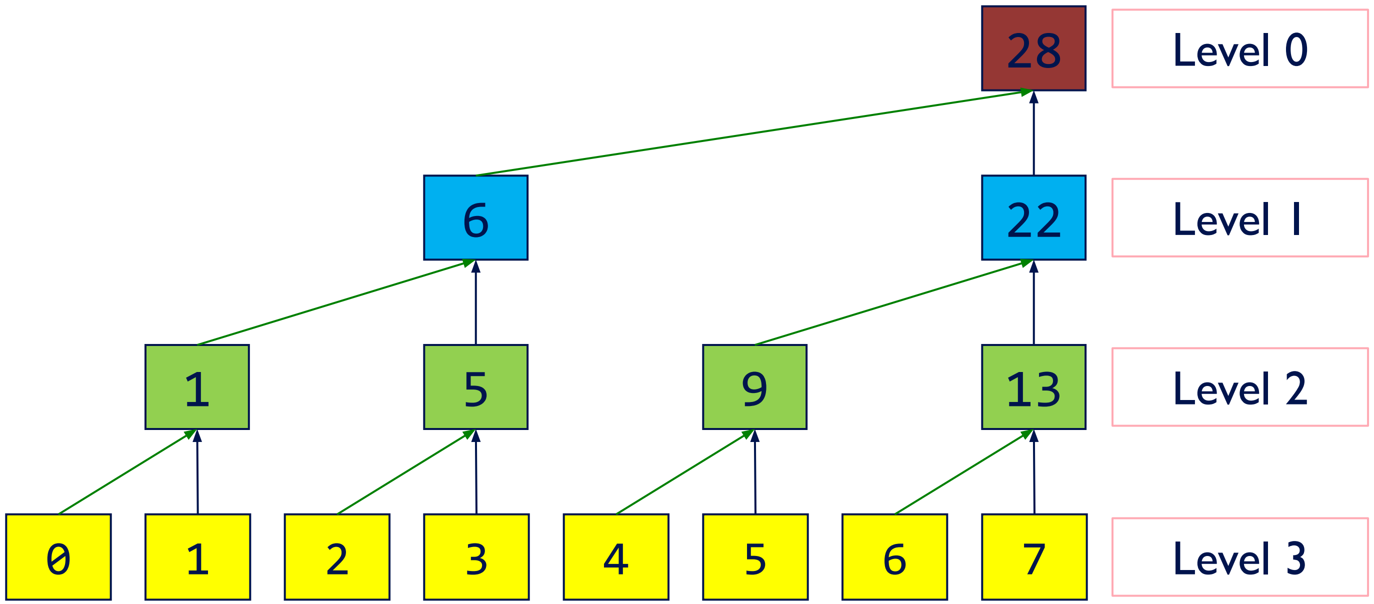
- Use a *balanced binary tree* (in concept) to perform Scan in two phases:
  - **Up-Sweep** (Parallel Reduction)
  - **Down-Sweep**



Work-Efficient Parallel Scan

- Up-Sweep

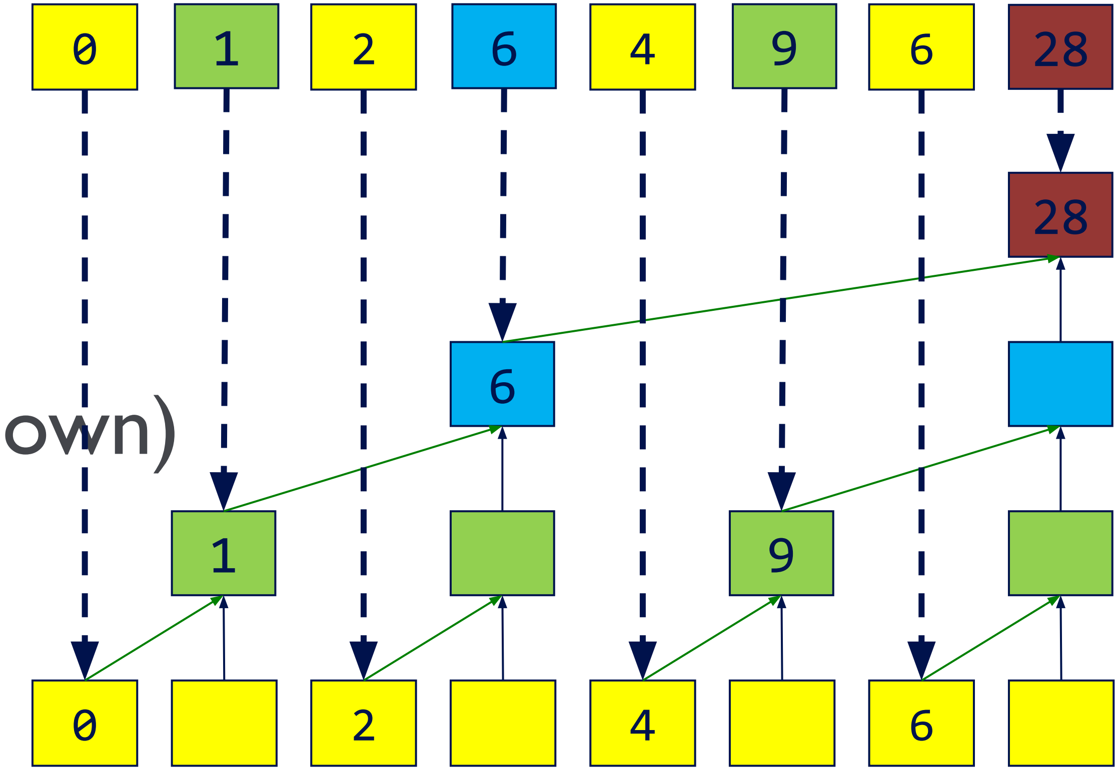
```
// Same code as our Parallel Reduction
for d = 0 to log2n - 1
  for all k = 0 to n – 1 by 2^(d+1) in parallel
    x[k + 2^(d+1) – 1] += x[k + 2^d – 1];
```



Penn Engineering

Work-Efficient Parallel Scan

- Imagine array as a tree
  - Array stores **only left child**
  - Right child is the element itself
- For node at index n
  - Left child index = n/2 (rounds down)
  - Right child index = n



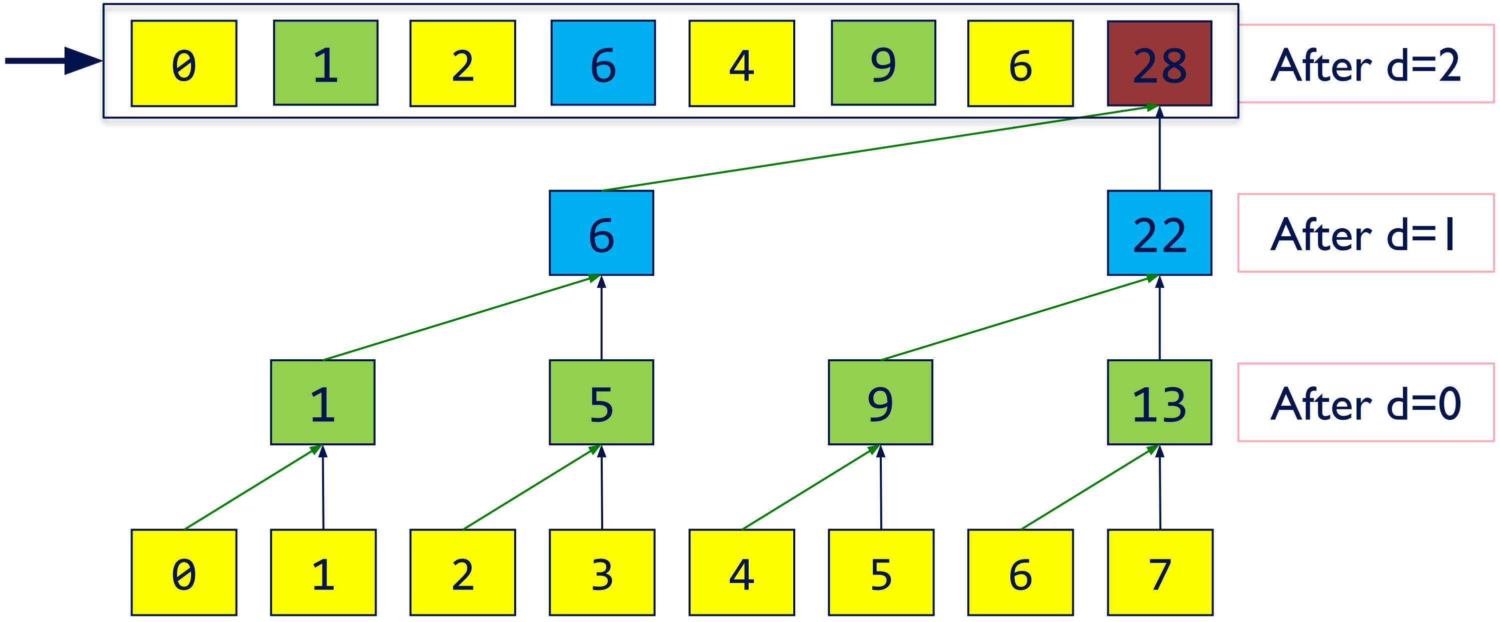
Penn Engineering

Work-Efficient Parallel Scan

- Up-Sweep

```
// Same code as our Parallel Reduction
for d = 0 to log2n - 1
  for all k = 0 to n – 1 by 2^(d+1) in parallel
    x[k + 2^(d+1) – 1] += x[k + 2^d – 1];
```

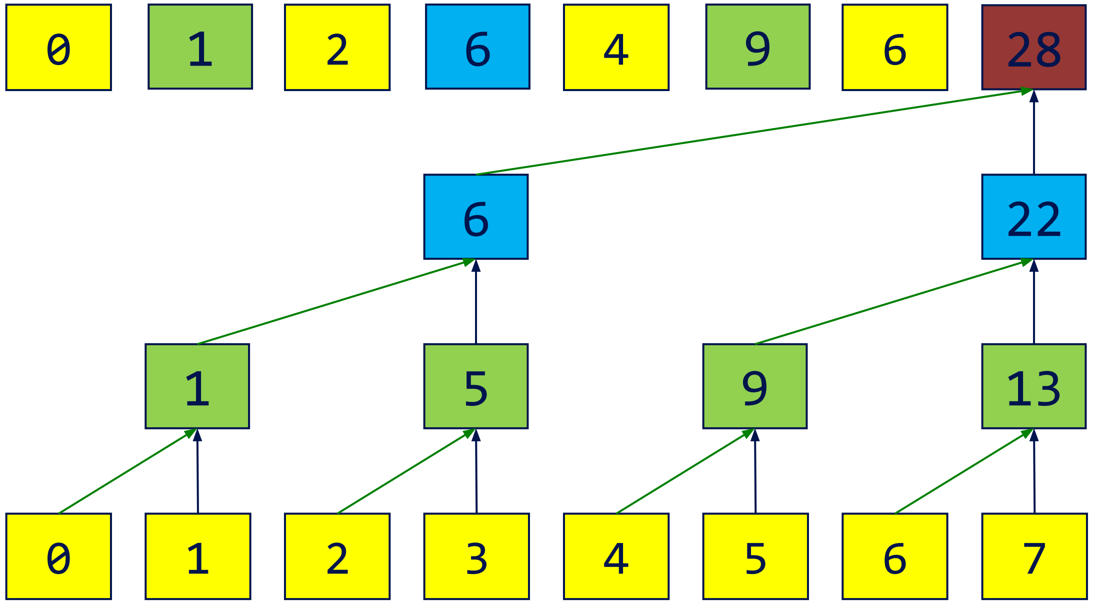
In-place reduction
On input array



Penn Engineering

Work-Efficient Parallel Scan

- Down-Sweep
  - "Traverse" back down tree using **partial sums** to build the scan in place.
    - Set root to zero
    - At each pass, a node passes its value to its left child, and sets the right child to the sum of the previous left child's value and its value



Work-Efficient Parallel Scan

- Down-Sweep
  - "Traverse" back down tree using partial sums to build the scan in place.
    - Set root to zero
    - At each pass, a node passes its value to its left child, and sets the right child to the sum of the previous left child's value and its value

```
x[n - 1] = 0
for d = log2n – 1 to 0
  for all k = 0 to n – 1 by 2^(d+1) in parallel
    t = x[k + 2^d – 1];                   // Save left child
    x[k + 2^d – 1] = x[k + 2^(d+1) – 1];  // Set left child to this node’s value
    x[k + 2^(d+1) – 1] += t;              // Set right child to old left value +
                                          // this node’s value
```

Work-Efficient Parallel Scan

- Down-Sweep

- Remember: This is a tree, but stored as linear array

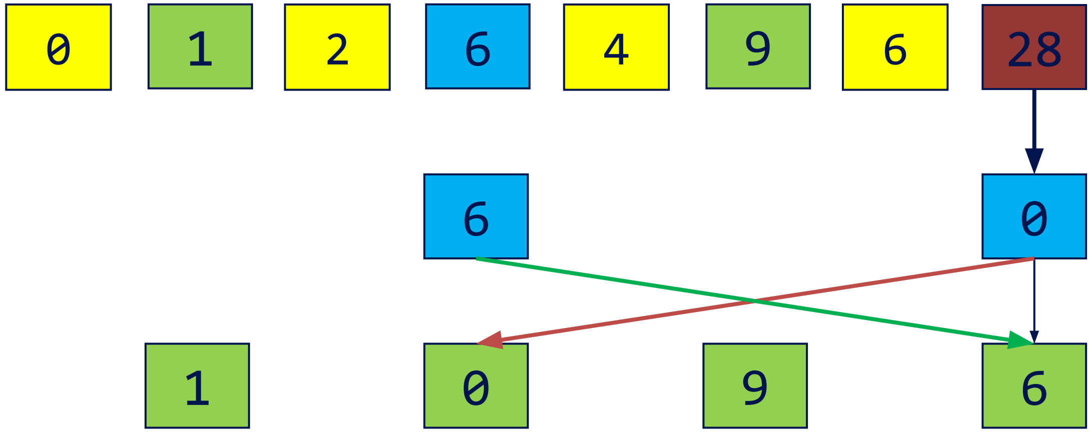
Work-Efficient Parallel Scan

At each level
- Left child: **Copy** the parent value
- Right child: **Add** the parent value and left child value copying root value.

Remember to think of this as a tree, not as array

Orange Arrow = Copy
Green Arrow + Black Arrow = Add

- Down-Sweep



Replace with 0

After d=0

Penn Engineering

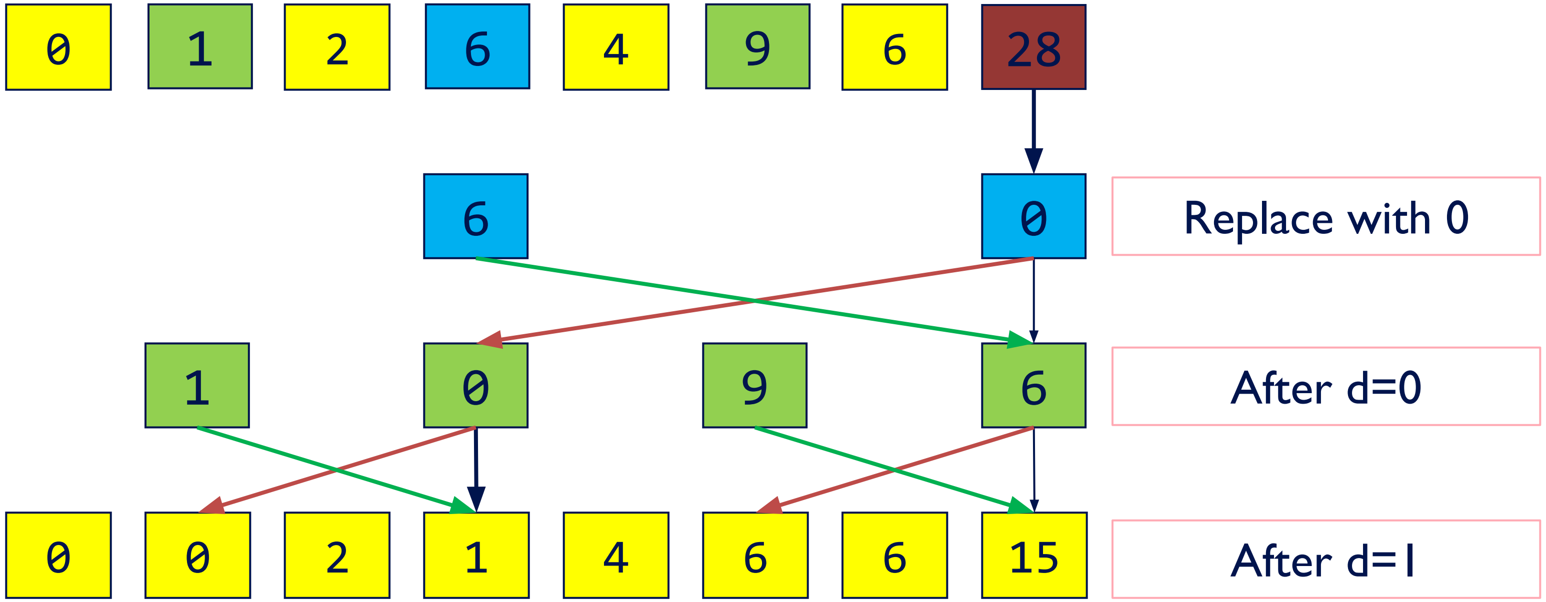
Work-Efficient Parallel Scan

- Down-Sweep

At each level
- Left child: **Copy** the parent value
- Right child: **Add** the parent value and left child value copying root value.

Remember to think of this as a tree, not as array

Orange Arrow = Copy
Green Arrow + Black Arrow = Add



Penn Engineering

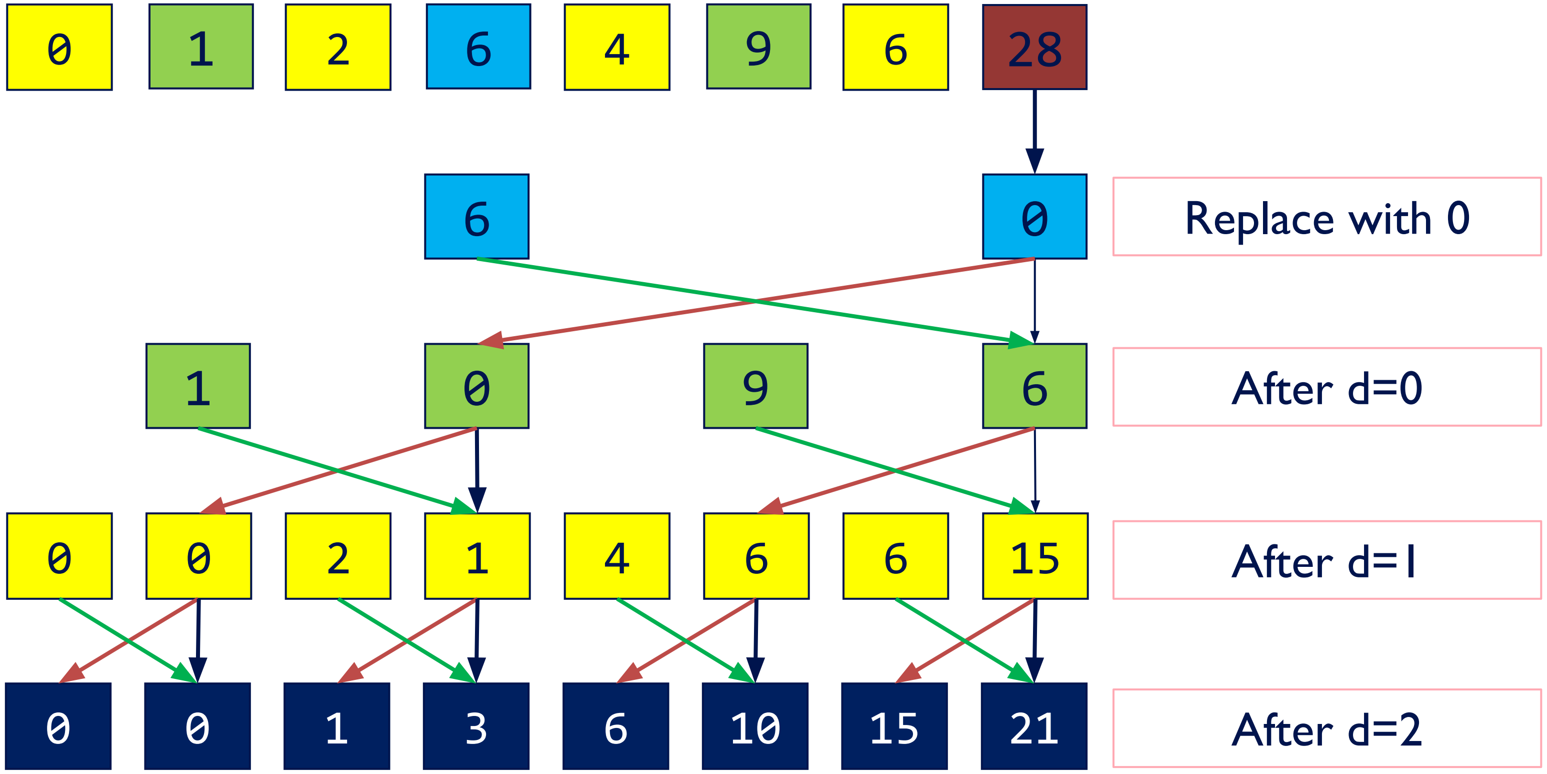
Work-Efficient Parallel Scan

- Down-Sweep

At each level
- Left child: **Copy** the parent value
- Right child: **Add** the parent value and left child value copying root value.

Remember to think of this as a tree, not as array

Orange Arrow = Copy
Green Arrow + Black Arrow = Add



Penn Engineering

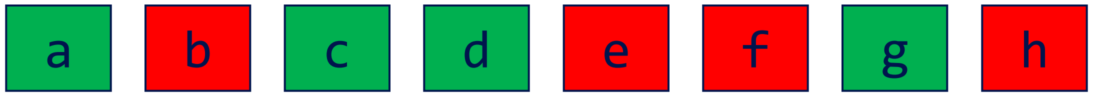
Work-Efficient Parallel Scan

- Up-Sweep
  - O(n) adds
- Down-Sweep
  - O(n) adds
  - O(n) swaps
- This ***Exclusive Scan*** can then be converted to an ***Inclusive Scan***

Stream Compaction

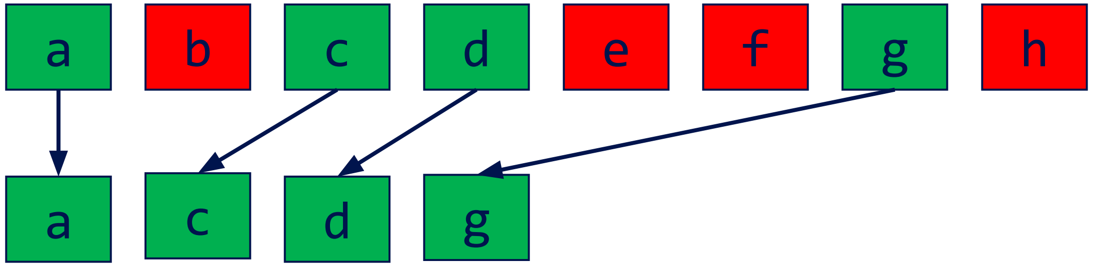
Stream Compaction

- Given an array of elements
  - Create a new array with elements that meet a certain criteria, e.g. non null
  - Preserve order



Stream Compaction

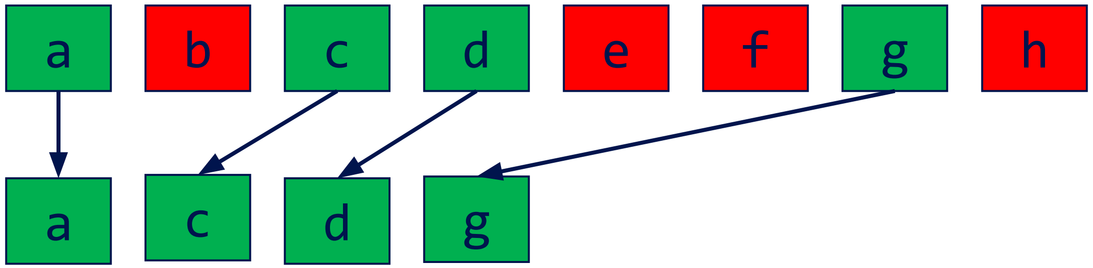
- Given an array of elements
  - Create a new array with elements that meet a certain criteria, e.g. non null
  - Preserve order



Penn Engineering

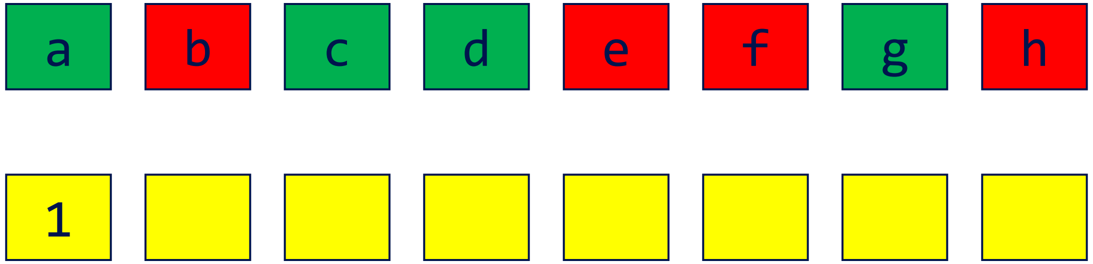
Stream Compaction

- Used in path tracing, collision detection, sparse matrix compression, etc.
- Can **reduce data** transferred from GPU to CPU



Stream Compaction

- *Step 1*: Compute temporary array containing
  - 1 if corresponding element meets criteria
  - 0 if element does not meet criteria



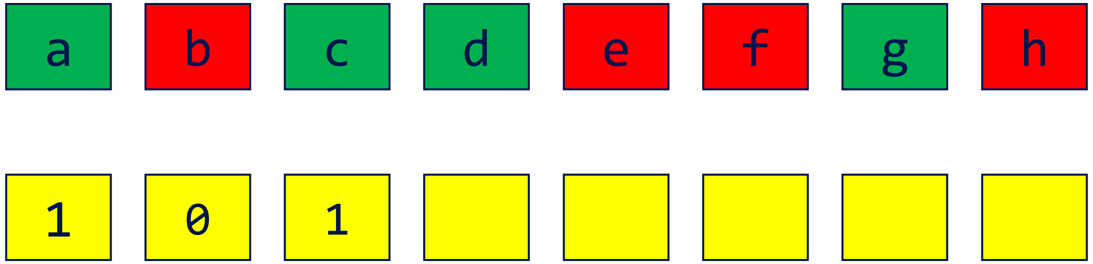
Stream Compaction

- *Step 1*: Compute temporary array containing
  - 1 if corresponding element meets criteria
  - 0 if element does not meet criteria

a	b	c	d	e	f	g	h
1	0						

Stream Compaction

- *Step 1*: Compute temporary array containing
  - 1 if corresponding element meets criteria
  - 0 if element does not meet criteria



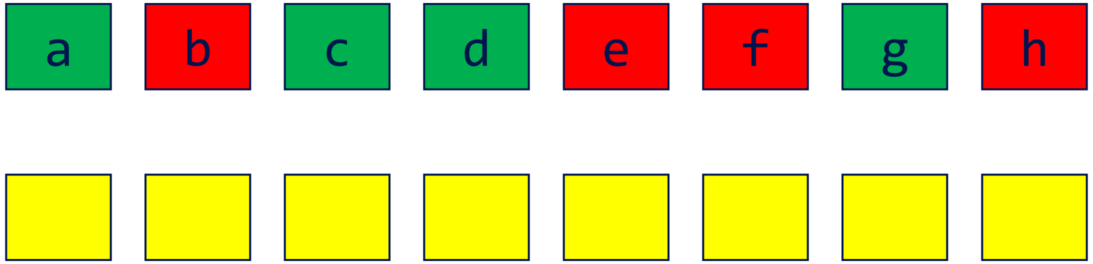
Stream Compaction

- *Step 1*: Compute temporary array containing
  - 1 if corresponding element meets criteria
  - 0 if element does not meet criteria

a	b	c	d	e	f	g	h
1	0	1	1	0	0	1	0

Stream Compaction

- *Step 1*: Compute temporary array containing
  - 1 if corresponding element meets criteria
  - 0 if element does not meet criteria
- **Runs in parallel !!**



Stream Compaction

- *Step 1*: Compute temporary array containing
  - 1 if corresponding element meets criteria
  - 0 if element does not meet criteria
- **Runs in parallel !!**

a	b	c	d	e	f	g	h
1	0	1	1	0	0	1	0

Stream Compaction

- Step 2: Run **exclusive scan** on temporary array

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:								

Stream Compaction

- Step 2: Run **exclusive scan** on temporary array
- Scan also runs in parallel
- What can we do with the result?

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Stream Compaction

- Step 3: Scatter
  - Result of scan is index into final array
  - Only write an element if temporary array has a 1

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Final Array:				
	0	1	2	3

Stream Compaction

- Step 3: Scatter
  - Result of scan is index into final array
  - Only write an element if temporary array has a 1

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Final Array:	a			
	0	1	2	3

Stream Compaction

- Step 3: Scatter
  - Result of scan is index into final array
  - Only write an element if temporary array has a 1

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Final Array:	a	c		
	0	1	2	3

Stream Compaction

- Step 3: Scatter
  - Result of scan is index into final array
  - Only write an element if temporary array has a 1

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Final Array:	a	c	d	
	0	1	2	3

Stream Compaction

- Step 3: Scatter
  - Result of scan is index into final array
  - Only write an element if temporary array has a 1

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Final Array:	a	c	d	g
	0	1	2	3

Stream Compaction

- Step 3: Scatter
  - Result of scan is index into final array
  - Only write an element if temporary array has a 1

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4
Final Array:	a	c	d	g				
	0	1	2	3				

Stream Compaction

- Step 3: Scatter – **Runs in parallel !!**

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Final Array:				
	0	1	2	3

Stream Compaction

- Step 3: Scatter – **Runs in parallel !!**

	a	b	c	d	e	f	g	h
	1	0	1	1	0	0	1	0
Scan Result:	0	1	1	2	3	3	3	4

Final Array:	a	c	d	g
	0	1	2	3

Summed Area Table (SAT)

Summed Area Table

- **Summed Area Table (SAT)**: 2D table where each element stores the sum of all elements in an input image between the lower left corner and the entry location.

Summed Area Table

- Example

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

$$(1 + 1 + 0) + (1 + 2 + 1) + (0 + 1 + 2) = 9$$

Summed Area Table

- Benefit
  - Used to perform different width filters at every pixel in the image in constant time per pixel
  - Just sample four pixels in SAT:

$$s_{filter} = \frac{s_{ur} - s_{ul} - s_{lr} + s_{ll}}{w \times h},$$

Image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Summed Area Table

- Uses
  - Approximate depth of field
  - Glossy environment reflections and refractions

Image from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1			

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2		

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2	2	

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

2			
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

2	5		
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

2	5	6	
1	2	2	4

Penn Engineering

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

2	5	6	8
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

Summed Area Table

How would implement
this on the GPU?

Summed Area Table

How would implement
this on the GPU?

Hint: Inclusive Scan

Summed Area Table

- Step 1 of 2: Row wise inclusive scan

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

Partial SAT

2	3	3	3
0	1	3	3
1	3	4	4
1	2	2	4

One inclusive scan for each row

Summed Area Table

- Step 2 of 2: Column wise inclusive scan

Input image

2	1	0	0
0	1	2	0
1	2	1	0
1	1	0	2

SAT

4	9	12	14
2	6	9	11
2	5	6	8
1	2	2	4

One inclusive scan for each column

Radix Sort

Radix Sort

- Efficient for small sort keys
  - k-bit keys require k passes

Radix Sort

- Each radix sort pass **partitions** its input based on one bit
- First pass starts with the *l*east *s*ignificant *b*it (*LSB*). Subsequent passes move towards the *m*ost *s*ignificant *b*it (*MSB*)

MSB `0110` *LSB*

Penn Engineering

Radix Sort

- Example input:

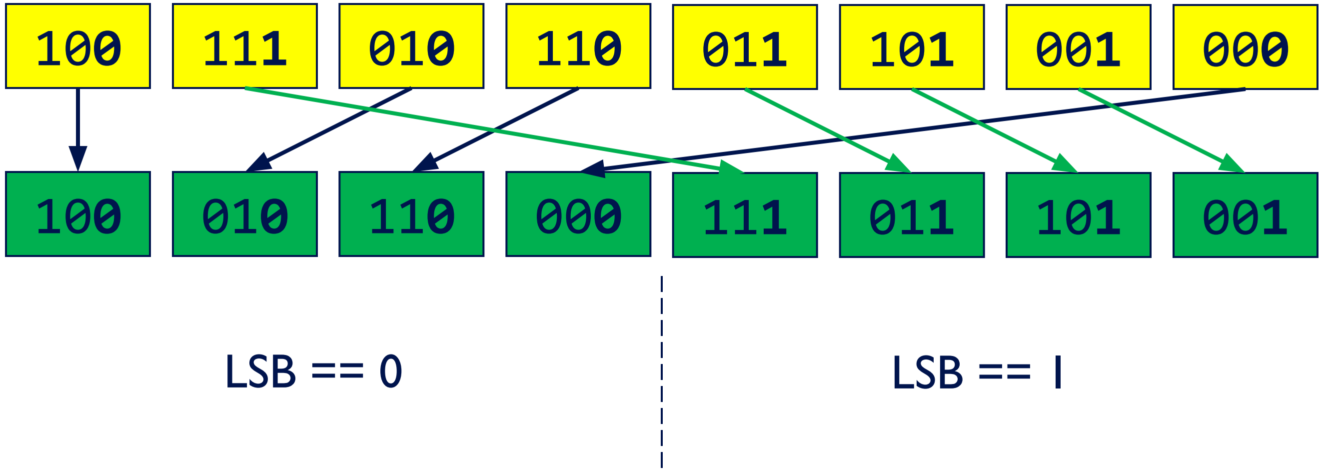
100	111	010	110	011	101	001	000

Penn Engineering

Example from http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html

Radix Sort

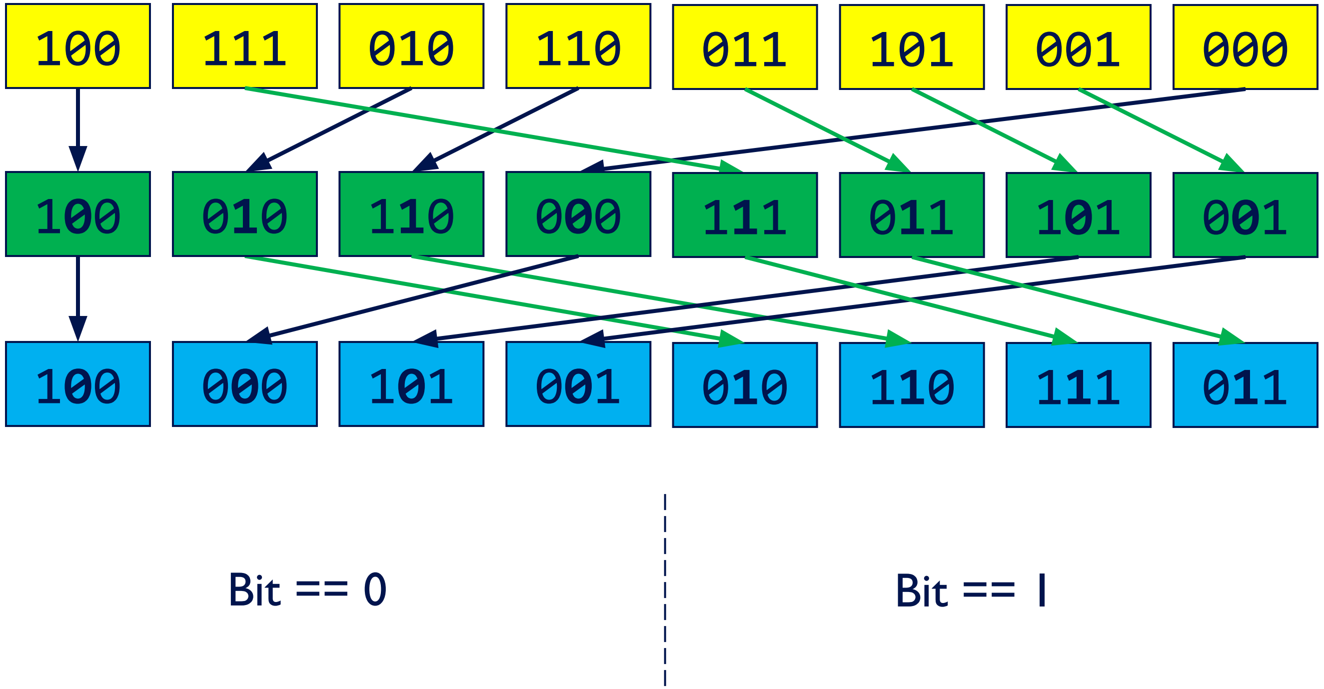
- First pass: partition based on LSB



Penn Engineering

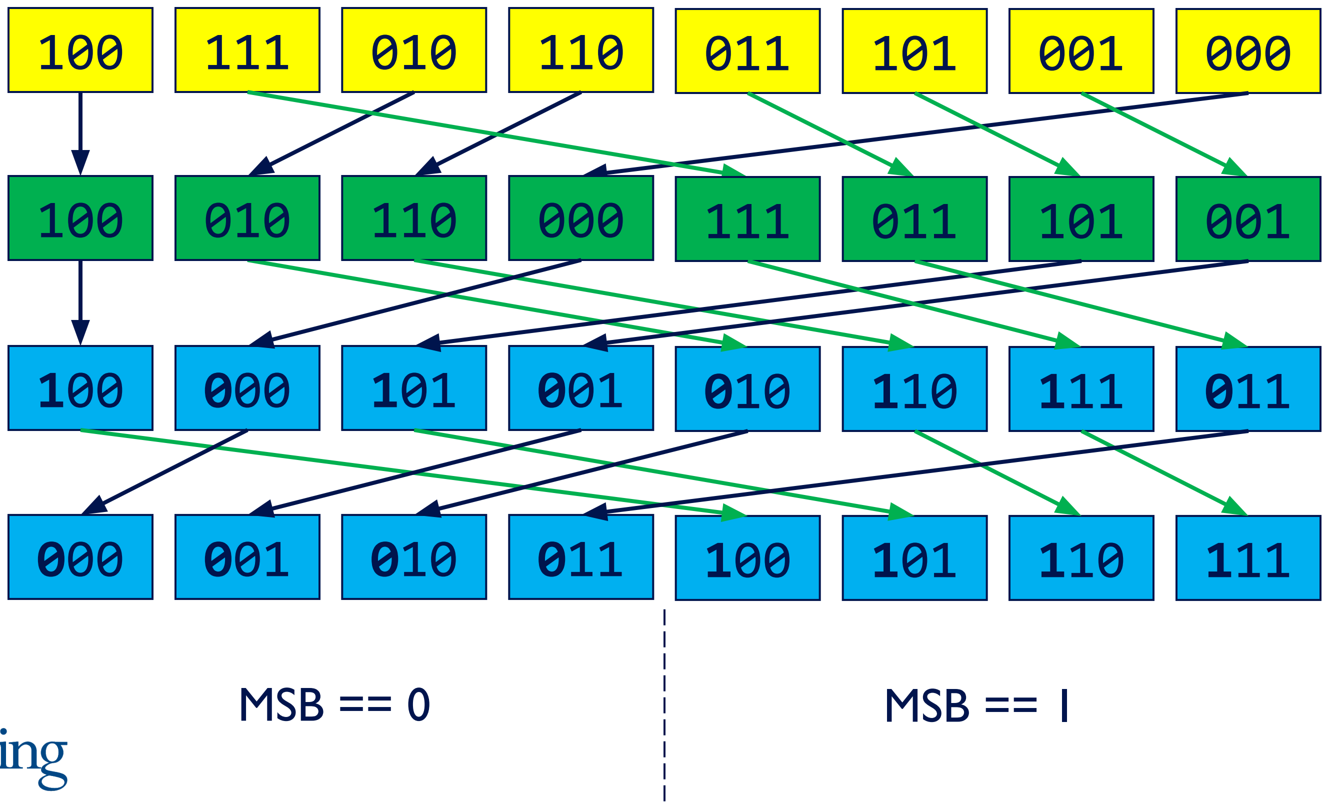
Radix Sort

- Second pass: partition based on middle bit



Radix Sort

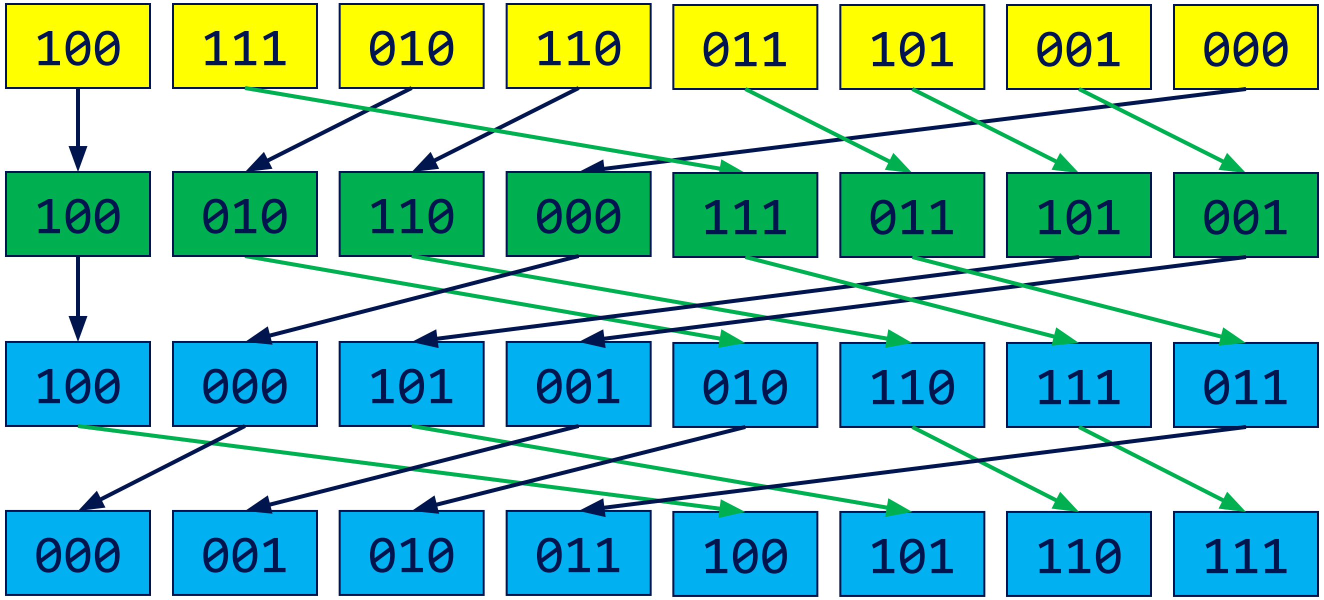
- Final pass: partition based on MSB



Penn Engineering

Radix Sort

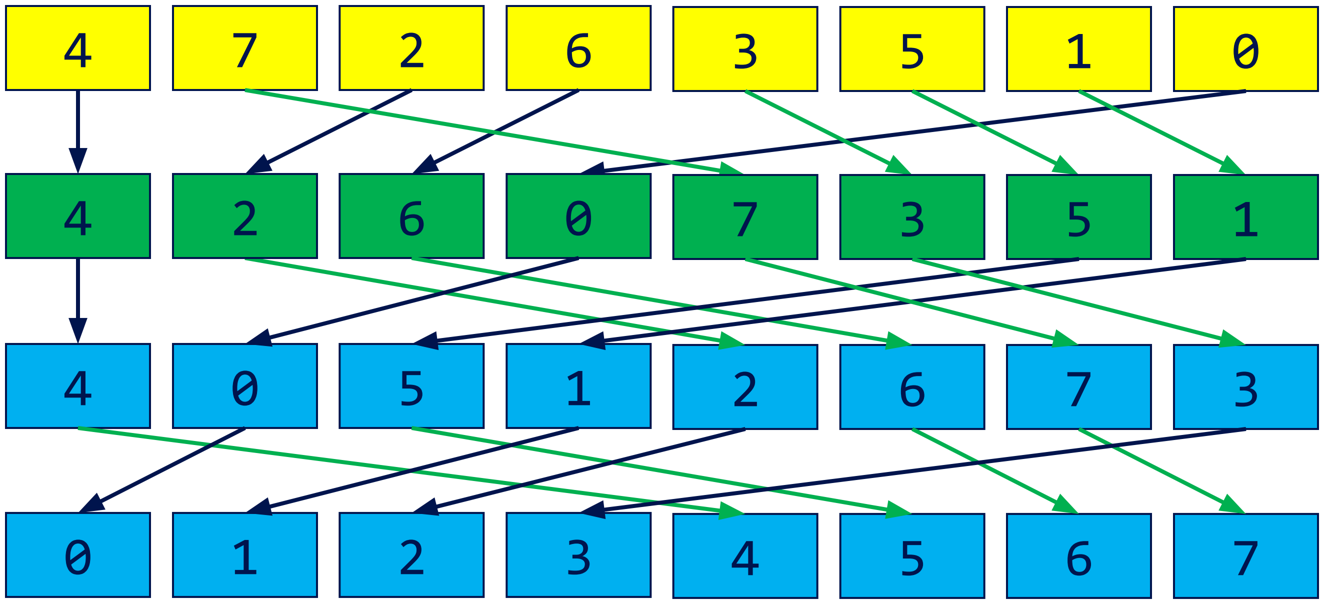
- Completed



Penn Engineering

Radix Sort

- Completed



Parallel Radix Sort

- Where is the parallelism?

Parallel Radix Sort

1. Break input arrays into tiles
   - Each tile fits into shared memory for an SM
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?

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

- Implement spilt. 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
1	0	1	1	0	0	0	1	**e** array (= !b)

Parallel Radix Sort

- Step 2: Exclusive scan e array and store it in f

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array (= !b)
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
1	0	1	1	0	0	0	1	**e** array (= !b)
0	1	1	2	3	3	3	3	**f** array

```
totalFalses = e[n – 1] + f[n – 1]
totalFalses = 1 + 3
totalFalses = 4
```

Parallel Radix Sort

- Step 4: Compute t array
  - t array is address for writing true keys

100	111	010	110	011	101	001	000	i array
0	1	0	0	1	1	1	0	b array
1	0	1	1	0	0	0	1	e array (= !b)
0	1	1	2	3	3	3	3	f array
								t array

```
totalFalses = 4
t[i] = i – f[i] + totalFalses
```

Penn Engineering

Parallel Radix Sort

- Step 4: Compute **t** array
  - **t** array is **address** for writing true keys

100	111	010	110	011	101	001	000	**i** array
0	1	0	0	1	1	1	0	**b** array
1	0	1	1	0	0	0	1	**e** array (= !b)
0	1	1	2	3	3	3	3	**f** array
4								**t** array

```
totalFalses = 4
t[i] = i – f[i] + totalFalses ==> t[0] = 0 – 0 + 4
```

Parallel Radix Sort

- Step 4: Compute **t** array
  - **t** array is **address** for writing true keys

100	111	010	110	011	101	001	000	**i** array
0	1	0	0	1	1	1	0	**b** array
1	0	1	1	0	0	0	1	**e** array (= !b)
0	1	1	2	3	3	3	3	**f** array
4	4							**t** array

```
totalFalses = 4
t[i] = i – f[i] + totalFalses ==> t[1] = 1 – 1 + 4
```

Parallel Radix Sort

- Step 4: Compute **t** array
  - **t** array is **address** for writing true keys

100	111	010	110	011	101	001	000	**i** array
0	1	0	0	1	1	1	0	**b** array
1	0	1	1	0	0	0	1	**e** array (= !b)
0	1	1	2	3	3	3	3	**f** array
4	4	$\boxed{5}$						**t** array

```
totalFalses = 4
t[i] = i – f[i] + totalFalses ==> t[2] = 2 – 1 + 4
```

Parallel Radix Sort

- Step 4: Compute **t** array
  - **t** array is **address** for writing true keys

100	111	010	110	011	101	001	000	**i** array
0	1	0	0	1	1	1	0	**b** array
1	0	1	1	0	0	0	1	**e** array (= !b)
0	1	1	2	3	3	3	3	**f** array
4	4	5	5	5	6	7	8	**t** array

```
totalFalses = 4
t[i] = i – f[i] + totalFalses
```

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
1	0	1	1	0	0	0	1	**e** array (= !b)
0	1	1	2	3	3	3	3	**f** array
4	4	5	5	5	6	7	8	**t** array
								d[i] = b[i] ? t[i] : f[i]

Penn Engineering

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
1	0	1	1	0	0	0	1	e array (= !b)
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
1	0	1	1	0	0	0	1	e array (= !b)
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
1	0	1	1	0	0	0	1	e array (= !b)
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]

Penn Engineering

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
1	0	1	1	0	0	0	1	e array (= !b)
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]

Penn Engineering

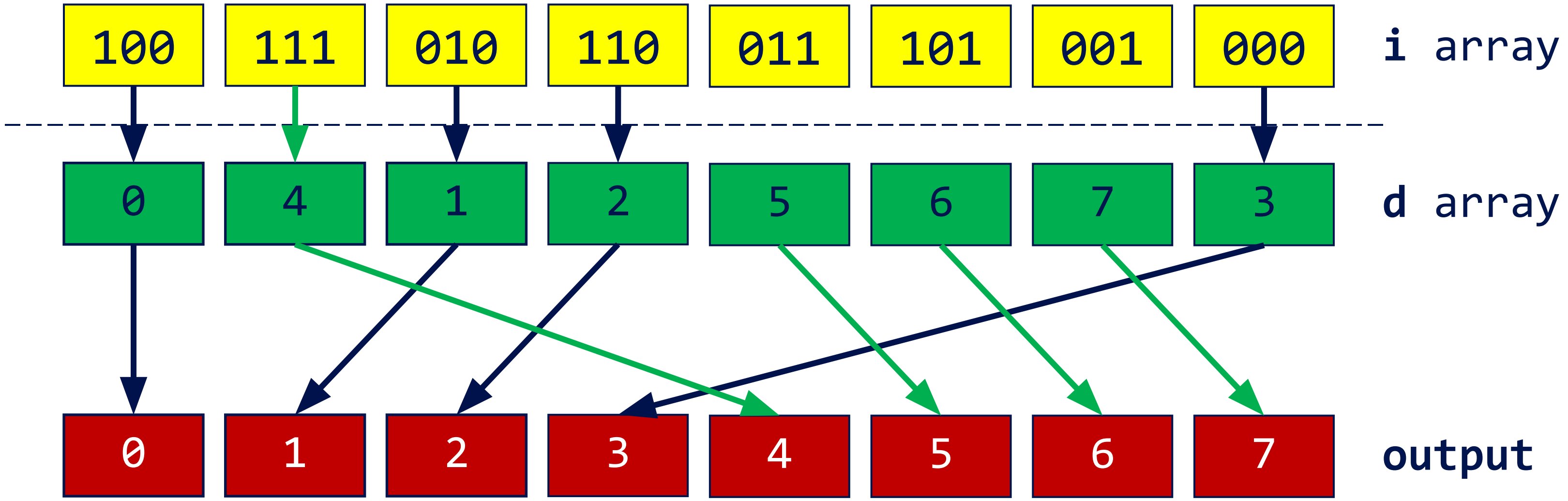
Parallel Radix Sort

- 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 array
								output

Parallel Radix Sort

- Step 5: Scatter based on address d



Parallel Radix Sort

- Given k-bit keys, how do we sort using our new *split* function?

- Once each tile is sorted, how do we merge tiles to provide the final sorted array?

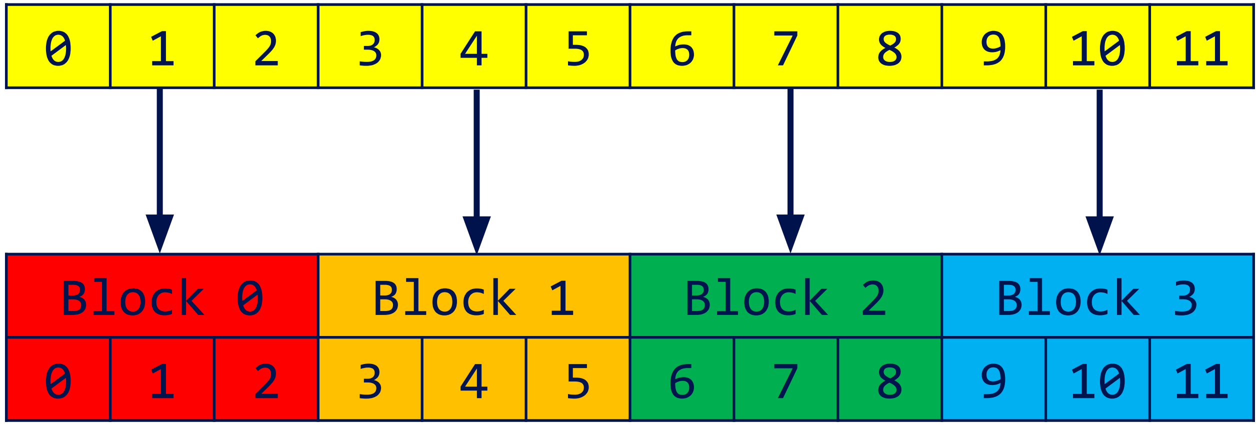
Scan Revisited

Scan Limitations

- Requires power-of-two length
- Executes in one block (unless only using global memory)
  - Length up to twice the number of threads in a block

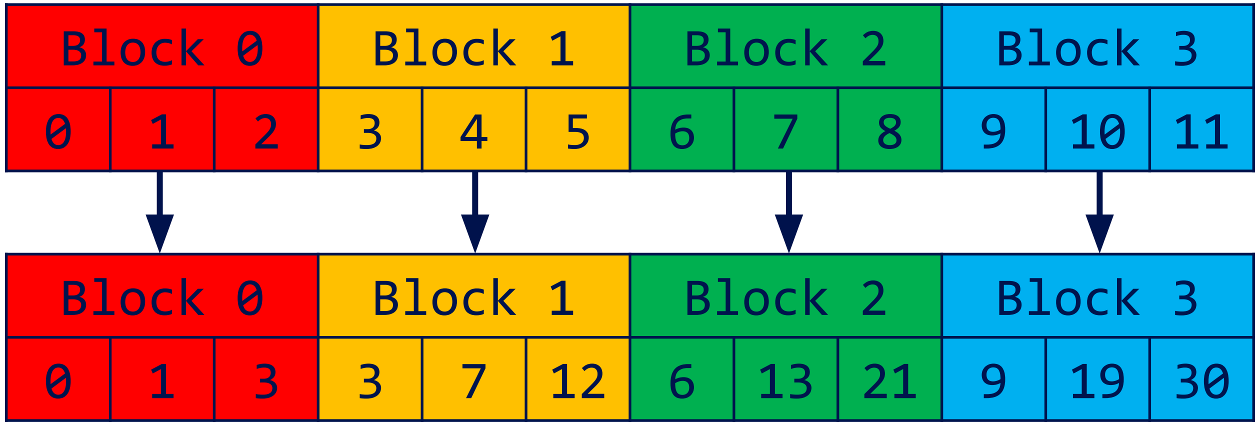
Scan on Arrays of Arbitrary Length

1. Divide the array into blocks



Scan on Arrays of Arbitrary Length

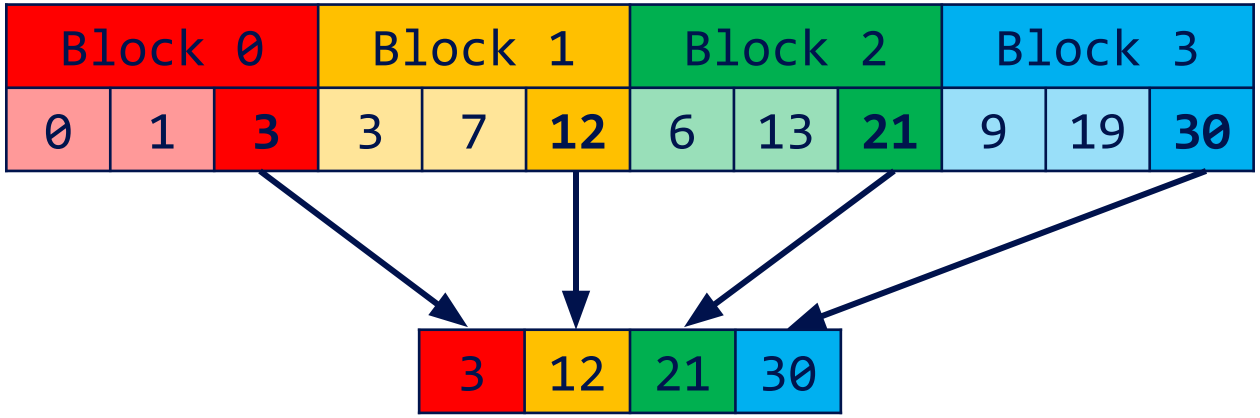
2. Run scan on each block



Run in parallel over many SMs

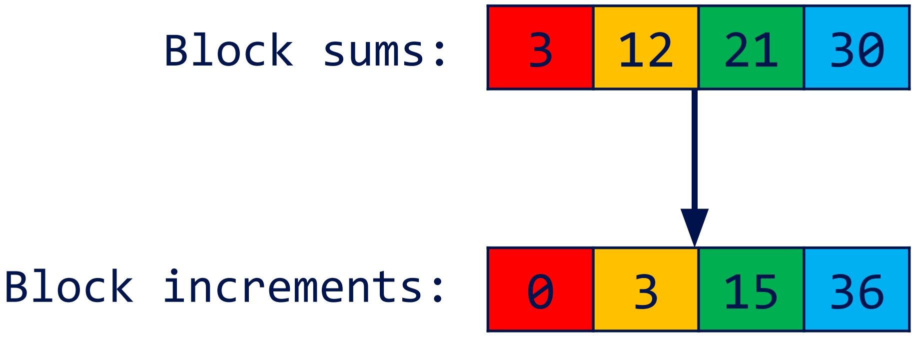
Scan on Arrays of Arbitrary Length

3. Write total sum of each block into a new array



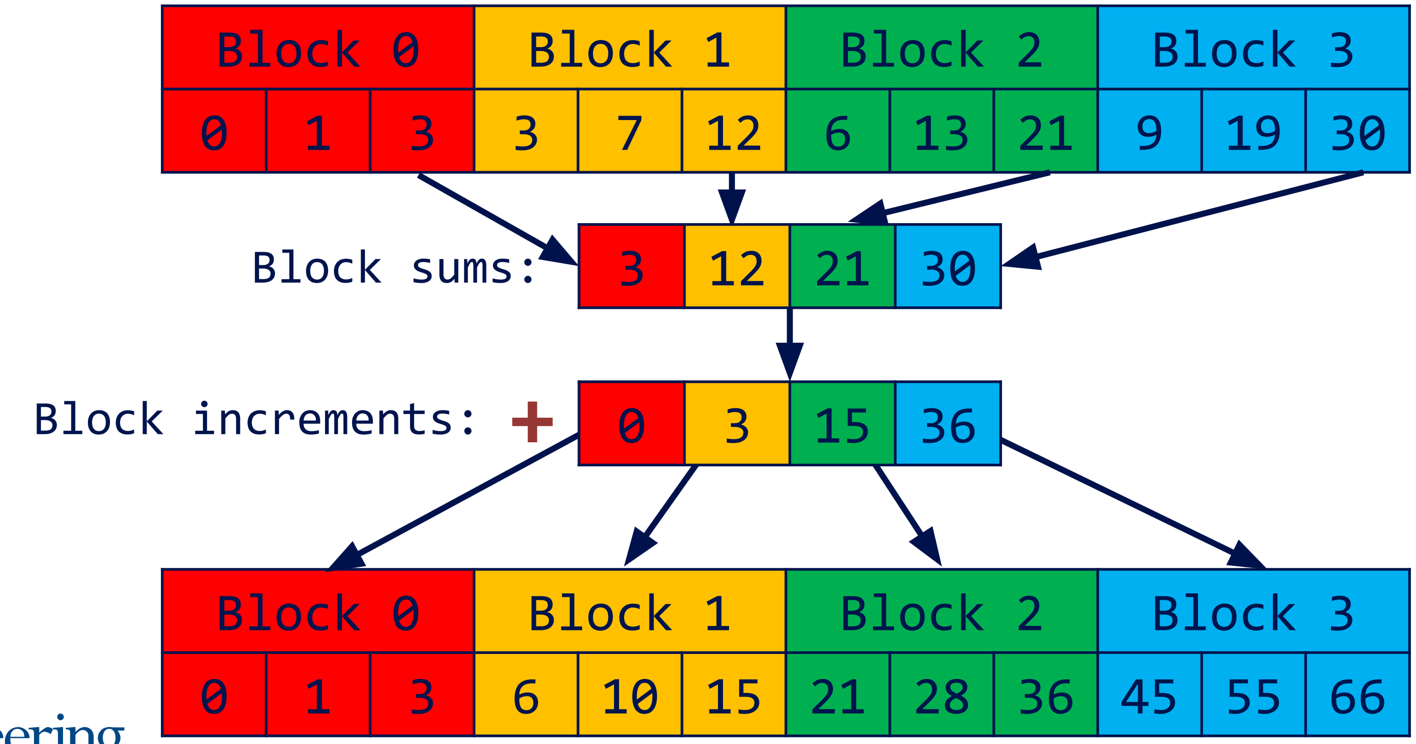
Scan on Arrays of Arbitrary Length

4. Exclusive scan block sums to compute block increments



Scan on Arrays of Arbitrary Length

5. Add block increments to each element in the corresponding block



Penn Engineering

Scan on Arrays of Arbitrary Length

- Non-power-of-two length:
- Pad last block with zeros up until the block size

Summary

- Parallel reduction, scan, and sort are building blocks for many algorithms
- An understanding of parallel programming **and** GPU architecture yields efficient GPU implementations