

# Cost models and advanced Futhark programming

Troels Henriksen (athas@sigkill.dk)  
Some material by Martin Elsman

DIKU  
University of Copenhagen

23rd of November, 2022

## Parallel cost models

Prefix sums (scans)

Using scans

Auxiliary

## The need for cost models

Which is better?

```
import numpy as np

def inc_scalar(x):
    for i in range(len(x)):
        x[i] = x[i] + 1

def inc_par(x):
    return x + np.ones(x.shape)
```

# The need for cost models

Which is better?

```
import numpy as np

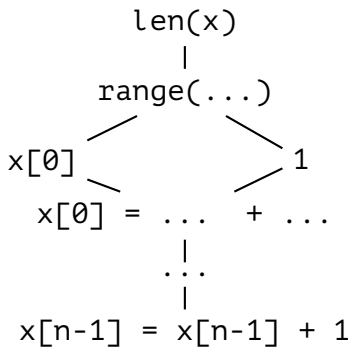
def inc_scalar(x):
    for i in range(len(x)):
        x[i] = x[i] + 1

def inc_par(x):
    return x + np.ones(x.shape)
```

Intuitively, `inc_par` is better because it is “more parallel”.

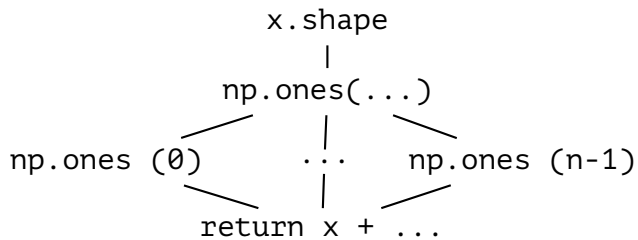
**Parallel cost models make this notion precise.**

## Dependency DAG for `inc_scalar`



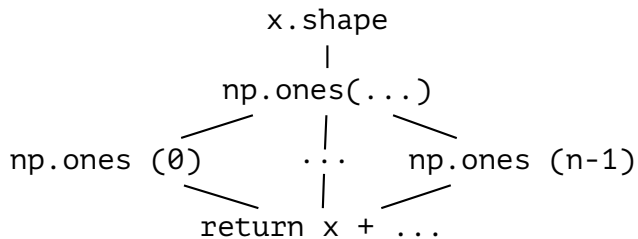
- Total count of nodes is the *work*,  $W(p)$ .
- Length of longest path from a leaf to the root is the *span*.
- **With an infinite number of processors, if a program  $p$  has span  $k$ , written  $S(p) = k$ , the program can execute in  $O(k)$  time.**
- Here,  $W(p) = O(n)$ ,  $S(p) = O(n)$ .

## Dependency DAG for `inc_par`



**What is the work and span complexity?**

## Dependency DAG for `inc_par`



**What is the work and span complexity?**

- $W(p) = O(n)$
- $S(p) = O(1)$

## Parallel cost model based on work and span

Instead of giving just a simple cost-model based on the total notion of work carried out by a program, we give instead a *refined* cost model, which aims at providing both:

- a notion of how much total work ( $W$ ) the program does;
- a notion of the *span*<sup>1</sup> ( $S$ ) of the program, specifying the maximum depth required by the computation.

### Notice:

- The span is the length of the longest sequence of operations that must be performed sequentially due to data dependencies.
- With an infinite number of processors, if a program  $p$  has span  $k$ , written  $S(p) = k$ , the program can execute in  $O(k)$  time.

---

<sup>1</sup>Sometimes also called *depth*.



## Brent's Theorem (1974)

(or Lemma, or Law...)

Writing  $T_i$  for the time taken to execute an algorithm on  $i$  processors, Brent's Theorem states that

$$\frac{T_1}{p} \leq T_p \leq T_\infty + \frac{T_1}{p}$$

**Proof sketch:** At level  $j$  of the DAG there are  $M_j$  independent operations, which can clearly be executed by  $p$  processors in time

$$\left\lceil \frac{M_j}{p} \right\rceil$$

Sum these for each level of the DAG.



### Ramification

We can simulate an “infinitely parallel” machine on a real machine at an overhead proportional to the amount of “missing” hardware parallelism.

# Language-based cost models

- Tallying up levels in an infinite DAG is impractical for real programs. Instead we prefer a *language-based cost model*
- E.g.  $W(x + y)$  is defined as  $W(x) + W(y)$ .
- The following slides define work and span cost for a small subset of Futhark.
- Write  $\llbracket e \rrbracket$  for the result of evaluating expression  $e$  (we are being intuitive about scopes and such).

# Language-based cost models

- Tallying up levels in an infinite DAG is impractical for real programs. Instead we prefer a *language-based cost model*
- E.g.  $W(x + y)$  is defined as  $W(x) + W(y)$ .
- The following slides define work and span cost for a small subset of Futhark.
- Write  $\llbracket e \rrbracket$  for the result of evaluating expression  $e$  (we are being intuitive about scopes and such).

## Cost model must be implementable

*A provable time and space efficient implementation of NESL*—Guy Blelloch and John Greiner, 1996

## Simple cases

$$W(v) =$$

$$S(v) =$$

$$W(e_1 \oplus e_2) =$$

$$S(e_1 \oplus e_2) =$$

$$W(\backslash x \rightarrow e) =$$

$$S(\backslash x \rightarrow e) =$$

## Simple cases

$$W(v) = 1$$

$$S(v) = 1$$

$$W(e_1 \oplus e_2) =$$

$$S(e_1 \oplus e_2) =$$

$$W(\backslash x \rightarrow e) =$$

$$S(\backslash x \rightarrow e) =$$

## Simple cases

$$W(v) = 1$$

$$S(v) = 1$$

$$W(e_1 \oplus e_2) = W(e_1) + W(e_2) + 1$$

$$S(e_1 \oplus e_2) = S(e_1) + S(e_2) + 1$$

$$W(\backslash x \rightarrow e) =$$

$$S(\backslash x \rightarrow e) =$$

## Simple cases

$$W(v) = 1$$

$$S(v) = 1$$

$$W(e_1 \oplus e_2) = W(e_1) + W(e_2) + 1$$

$$S(e_1 \oplus e_2) = S(e_1) + S(e_2) + 1$$

$$W(\backslash x \rightarrow e) = 1$$

$$S(\backslash x \rightarrow e) = 1$$

## Simple cases

$$W(v) = 1$$

$$S(v) = 1$$

$$W(e_1 \oplus e_2) = W(e_1) + W(e_2) + 1$$

$$S(e_1 \oplus e_2) = S(e_1) + S(e_2) + 1$$

$$W(\backslash x \rightarrow e) = 1$$

$$S(\backslash x \rightarrow e) = 1$$

$$W([e_1, \dots, e_n]) =$$

$$S([e_1, \dots, e_n]) =$$

$$W((e_1, \dots, e_n)) =$$

$$S((e_1, \dots, e_n)) =$$



## Simple cases

$$W(v) = 1$$

$$S(v) = 1$$

$$W(e_1 \oplus e_2) = W(e_1) + W(e_2) + 1$$

$$S(e_1 \oplus e_2) = S(e_1) + S(e_2) + 1$$

$$W(\backslash x \rightarrow e) = 1$$

$$S(\backslash x \rightarrow e) = 1$$

$$W([e_1, \dots, e_n]) = W(e_1) + \dots + W(e_n) + 1$$

$$S([e_1, \dots, e_n]) = S(e_1) + \dots + S(e_n) + 1$$

$$W((e_1, \dots, e_n)) = W(e_1) + \dots + W(e_n) + 1$$

$$S((e_1, \dots, e_n)) = S(e_1) + \dots + S(e_n) + 1$$

## Interesting cases

$W(\text{iota } e) =$

$S(\text{iota } e) =$

## Interesting cases

$$W(\text{iota } e) = W(e) + \llbracket e \rrbracket$$

$$S(\text{iota } e) = S(e) + 1$$

## Interesting cases

$$W(\text{iota } e) = W(e) + \llbracket e \rrbracket$$

$$S(\text{iota } e) = S(e) + 1$$

$$W(\textbf{let } x = e \textbf{ in } e') =$$

$$S(\textbf{let } x = e \textbf{ in } e') =$$

## Interesting cases

$$W(\text{iota } e) = W(e) + \llbracket e \rrbracket$$

$$S(\text{iota } e) = S(e) + 1$$

$$W(\text{let } x = e \text{ in } e') = W(e) + W(e'[x \mapsto \llbracket e \rrbracket]) + 1$$

$$S(\text{let } x = e \text{ in } e') = S(e) + S(e'[x \mapsto \llbracket e \rrbracket]) + 1$$

## Interesting cases

$$W(\text{iota } e) = W(e) + \llbracket e \rrbracket$$

$$S(\text{iota } e) = S(e) + 1$$

$$W(\text{let } x = e \text{ in } e') = W(e) + W(e'[x \mapsto \llbracket e \rrbracket]) + 1$$

$$S(\text{let } x = e \text{ in } e') = S(e) + S(e'[x \mapsto \llbracket e \rrbracket]) + 1$$

$$W(e_1 \ e_2) =$$

$$S(e_1 \ e_2) =$$

## Interesting cases

$$W(\text{iota } e) = W(e) + \llbracket e \rrbracket$$

$$S(\text{iota } e) = S(e) + 1$$

$$W(\text{let } x = e \text{ in } e') = W(e) + W(e'[x \mapsto \llbracket e \rrbracket]) + 1$$

$$S(\text{let } x = e \text{ in } e') = S(e) + S(e'[x \mapsto \llbracket e \rrbracket]) + 1$$

$$W(e_1 \ e_2) = W(e_1) + W(e_2) + W(e'[x \mapsto \llbracket e_2 \rrbracket]) + 1$$

$$\text{where } \llbracket e_1 \rrbracket = \lambda x. x \rightarrow e'$$

$$S(e_1 \ e_2) = S(e_1) + S(e_2) + S(e'[x \mapsto \llbracket e_2 \rrbracket]) + 1$$

$$\text{where } \llbracket e_1 \rrbracket = \lambda x. x \rightarrow e'$$

## Work and span of map

$$W(\text{map } e_1 \ e_2) =$$

$$S(\text{map } e_1 \ e_2) =$$



## Work and span of map

$$W(\text{map } e_1 \ e_2) =$$

$$W(e_1) + W(e_2) + W(e'[x \mapsto v_1]) + \dots + W(e'[x \mapsto v_n])$$

$$\text{where } \llbracket e_1 \rrbracket = \lambda x. e'$$

$$\text{where } \llbracket e_2 \rrbracket = [v_1, \dots, v_n]$$

$$S(\text{map } e_1 \ e_2) =$$

$$S(e_1) + S(e_2) + \max(S(e'[x \mapsto v_1]), \dots, S(e'[x \mapsto v_n])) + 1$$

$$\text{where } \llbracket e_1 \rrbracket = \lambda x. e'$$

$$\text{where } \llbracket e_2 \rrbracket = [v_1, \dots, v_n]$$

## Reduction by contraction

```
def npow2 (n:i64) : i64 =  
  loop a = 2 while a < n do 2*a  
  
-- Pad a vector to make its size a power of two  
def padpow2 [n] (ne: i32) (v:[n]i32) : []i32 =  
  concat v (replicate (npow2 n - n) ne)  
  
-- Reduce by contraction  
def red (xs : []i32) : i32 =  
  let xs =  
    loop xs = padpow2 0 xs  
    while length xs > 1 do  
      let n = length xs / 2  
      in map2 (+) xs[0:n] xs[n:2*n]  
  in xs[0]
```

## Work and span of `loop`

$$W(\text{loop } x = e_1 \text{ while } e_2 \text{ do } e_3) =$$

$$S(\text{loop } x = e_1 \text{ while } e_2 \text{ do } e_3) =$$

## Work and span of **loop**

$W(\mathbf{loop} \ x = e_1 \ \mathbf{while} \ e_2 \ \mathbf{do} \ e_3) = W(e_1) + W(e_2[x \mapsto \llbracket e_1 \rrbracket]) +$   
if  $\llbracket e_2[x \mapsto \llbracket e_1 \rrbracket] \rrbracket = \mathbf{false}$   
then 0  
else  $W(e_3[x \mapsto \llbracket e_1 \rrbracket]) +$   
 $W(\mathbf{loop} \ x = \llbracket e_3[x \mapsto \llbracket e_1 \rrbracket] \rrbracket \ \mathbf{while} \ e_2 \ \mathbf{do} \ e_3)$

$S(\mathbf{loop} \ x = e_1 \ \mathbf{while} \ e_2 \ \mathbf{do} \ e_3) = S(e_1) + S(e_2[x \mapsto \llbracket e_1 \rrbracket]) +$   
if  $\llbracket e_2 \rrbracket[x \mapsto \llbracket e_1 \rrbracket] = \mathbf{false}$   
then 0  
else  $S(e_3[x \mapsto \llbracket e_1 \rrbracket]) +$   
 $S(\mathbf{loop} \ x = \llbracket e_3[x \mapsto \llbracket e_1 \rrbracket] \rrbracket \ \mathbf{while} \ e_2 \ \mathbf{do} \ e_3)$

**Work and Span for  $n^{\text{pow}2}$   $n$**

## Work and Span for $\text{npow2 } n$

By inspection, we have

$$W(\text{npow2 } n) = S(\text{npow2 } n) = O(\log n)$$

## Work and Span for $\text{padpow2 } ne \text{ } v$

## Work and Span for npow2 n

By inspection, we have

$$W(\text{npow2 } n) = S(\text{npow2 } n) = O(\log n)$$

## Work and Span for padpow2 ne v

Because  $\text{npow2 } n \leq 2n$ , we have (where  $n = \text{length } v$ )

$$\begin{aligned} W(\text{padpow2 } ne \ v) &= W(\text{concat } v \ (\text{replicate } (\text{npow2 } n - n) \ ne)) \\ &= O(n) \end{aligned}$$

$$S(\text{padpow2 } ne \ v) = O(\log n)$$

## Work and Span for red

## Work and Span for npow2 n

By inspection, we have

$$W(\text{npow2 } n) = S(\text{npow2 } n) = O(\log n)$$

## Work and Span for padpow2 ne v

Because  $\text{npow2 } n \leq 2n$ , we have (where  $n = \text{length } v$ )

$$\begin{aligned} W(\text{padpow2 } ne \ v) &= W(\text{concat } v \ (\text{replicate } (\text{npow2 } n - n) \ ne)) \\ &= O(n) \end{aligned}$$

$$S(\text{padpow2 } ne \ v) = O(\log n)$$

## Work and Span for red

Each loop iteration in `red` has span  $O(1)$ . Because the loop is iterated at-most  $\log(2n)$  times, we have (where  $n = \text{length } v$ )

$$W(\text{red } v) = O(n) + O(n/2) + O(n/4) + \dots + O(1) =$$



## Work and Span for npow2 n

By inspection, we have

$$W(\text{npow2 } n) = S(\text{npow2 } n) = O(\log n)$$

## Work and Span for padpow2 ne v

Because  $\text{npow2 } n \leq 2n$ , we have (where  $n = \text{length } v$ )

$$\begin{aligned} W(\text{padpow2 ne } v) &= W(\text{concat } v (\text{replicate } (\text{npow2 } n - n) \text{ ne})) \\ &= O(n) \end{aligned}$$

$$S(\text{padpow2 ne } v) = O(\log n)$$

## Work and Span for red

Each loop iteration in `red` has span  $O(1)$ . Because the loop is iterated at-most  $\log(2n)$  times, we have (where  $n = \text{length } v$ )

$$\begin{aligned} W(\text{red } v) &= O(n) + O(n/2) + O(n/4) + \dots + O(1) = O(n) \\ S(\text{red } v) &= \end{aligned}$$

## Work and Span for npow2 n

By inspection, we have

$$W(\text{npow2 } n) = S(\text{npow2 } n) = O(\log n)$$

## Work and Span for padpow2 ne v

Because  $\text{npow2 } n \leq 2n$ , we have (where  $n = \text{length } v$ )

$$\begin{aligned} W(\text{padpow2 } ne \ v) &= W(\text{concat } v \ (\text{replicate } (\text{npow2 } n - n) \ ne)) \\ &= O(n) \end{aligned}$$

$$S(\text{padpow2 } ne \ v) = O(\log n)$$

## Work and Span for red

Each loop iteration in `red` has span  $O(1)$ . Because the loop is iterated at-most  $\log(2n)$  times, we have (where  $n = \text{length } v$ )

$$\begin{aligned} W(\text{red } v) &= O(n) + O(n/2) + O(n/4) + \dots + O(1) = O(n) \\ S(\text{red } v) &= O(\log n) \end{aligned}$$

## Work efficiency

A parallel algorithm is said to be *work efficient* if it has at most the same work as the best sequential algorithm.

Is red work efficient?

## Work efficiency

A parallel algorithm is said to be *work efficient* if it has at most the same work as the best sequential algorithm.

Is red work efficient?

**Yes**, because it does  $O(n)$  work, which is as good as a sequential summation.

Is it also *efficient*?

## Performance Compared to the Built-in Reduction SOAC

```
-- ==  
-- entry: test_red test_reduce  
-- random input { [10000000]i32 }  
entry test_red = red  
entry test_reduce = reduce (+) 0
```

## Performance Compared to the Built-in Reduction SOAC

```
-- ==  
-- entry: test_red test_reduce  
-- random input { [10000000]i32 }  
entry test_red = red  
entry test_reduce = reduce (+) 0
```

```
$ futhark bench --backend=opencl reduce.fut  
Compiling reduce.fut...  
Results for reduce.fut:test_red:  
dataset [10000000]i32:      4675.40 $\mu$ s  
Results for reduce.fut:test_reduce:  
dataset [10000000]i32:      273.80 $\mu$ s
```

## Performance Compared to the Built-in Reduction SOAC

```
-- ==  
-- entry: test_red test_reduce  
-- random input { [10000000]i32 }  
entry test_red = red  
entry test_reduce = reduce (+) 0
```

```
$ futhark bench --backend=opencl reduce.fut
```

```
Compiling reduce.fut...
```

```
Results for reduce.fut:test_red:
```

```
dataset [10000000]i32:      4675.40 $\mu$ s
```

```
Results for reduce.fut:test_reduce:
```

```
dataset [10000000]i32:      273.80 $\mu$ s
```

**If you are not using `futhark bench`, then you are probably doing it wrong.**

Parallel cost models

Prefix sums (scans)

Using scans

Auxiliary



# Inclusive and exclusive prefix sum

## Exclusive prefix sum ("prescan")

Given

[1, 2, 3, 4]

produce

[0, 1, 3, 6]

## Inclusive prefix sum

Given

[1, 2, 3, 4]

produce

[1, 3, 6, 10]

## Prefix sums are scans

Generalising the addition and zero used by a prefix sum to an arbitrary associative operator  $\oplus$  and neutral element  $0_{\oplus}$ , we get *scan*.

*-- The scan in Futhark is inclusive.*

```
> scan (+) 0 [1,2,3,4]  
[1, 3, 6, 10]
```

# Prefix sums are scans

Generalising the addition and zero used by a prefix sum to an arbitrary associative operator  $\oplus$  and neutral element  $0_{\oplus}$ , we get *scan*.

*-- The scan in Futhark is inclusive.*

```
> scan (+) 0 [1,2,3,4]  
[1, 3, 6, 10]
```

- Scans are a fundamental tool for parallelising seemingly-sequential algorithms.
- Let us see how scans can be computed in parallel.

## Sequential prefix sum

```
acc = 0
for i < n:
    acc = acc + input[i]
    scanned[i] = acc
```

## Sequential prefix sum

```
acc = 0
for i < n:
    acc = acc + input[i]
    scanned[i] = acc
```

Work:  $O(n)$

Span:  $O(n)$

## Brute force

To calculate the prefix sum of  $[x_0, \dots, x_{n-1}]$ , compute

$$\begin{aligned} &[sum([x_0]) \\ &sum([x_0, x_1]) \\ &\vdots \\ &sum([x_0, x_1, \dots, x_{n-1}])] \end{aligned}$$

Assume  $S(sum([x_0, \dots, x_{n-1}])) = \log_2(n)$ .

# Brute force

To calculate the prefix sum of  $[x_0, \dots, x_{n-1}]$ , compute

$$\begin{aligned} &[sum([x_0]) \\ &sum([x_0, x_1]) \\ &\vdots \\ &sum([x_0, x_1, \dots, x_{n-1}])) \end{aligned}$$

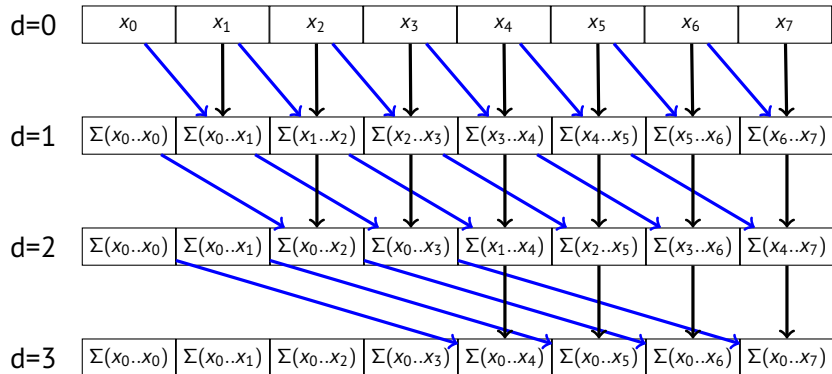
Assume  $S(sum([x_0, \dots, x_{n-1}])) = \log_2(n)$ .

**Work:**  $O(\sum_{i < n} i) = O(n^2)$

**Span:**  $O(\max(S(sum([x_0])), \dots, S(sum([x_0, \dots, x_{n-1}]))) = O(\log_2(n))$

**Terrible.** The sequential implementation is faster for large  $n$ !

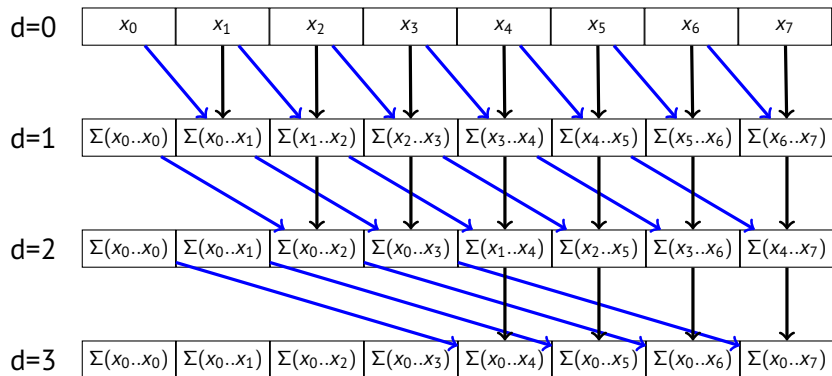
# Hillis–Steele scan (1986)



For each  $d$ , element  $x_i^d$  is updated by  $x_{i-2^d}^{d-1} + x_i^{d-1}$ .



# Hillis–Steele scan (1986)



For each  $d$ , element  $x_i^d$  is updated by  $x_{i-2^d}^{d-1} + x_i^{d-1}$ .

**Work:** For  $n = 2^m$ ,  $O(\sum_{i < m} 2^m - 2^i) = O(n \log(n))$

**Span:**  $\log(n)$

# Work-efficient scan

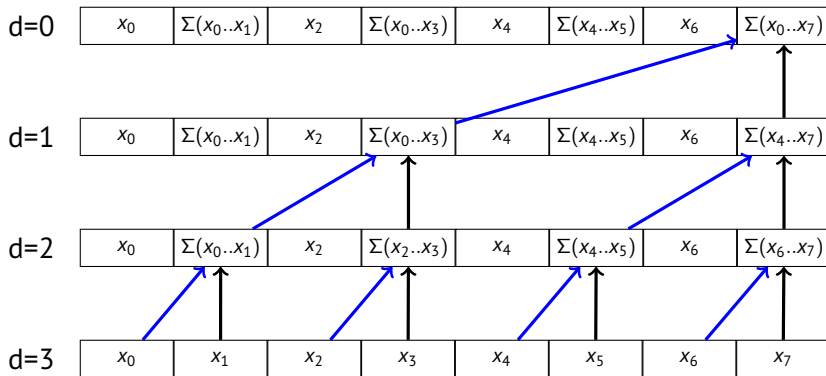
## Two passes

**Upsweep** Build a balanced binary tree of partial sums stored in every other cell.

**Downsweep** Use the partial sums to fill out the missing parts.

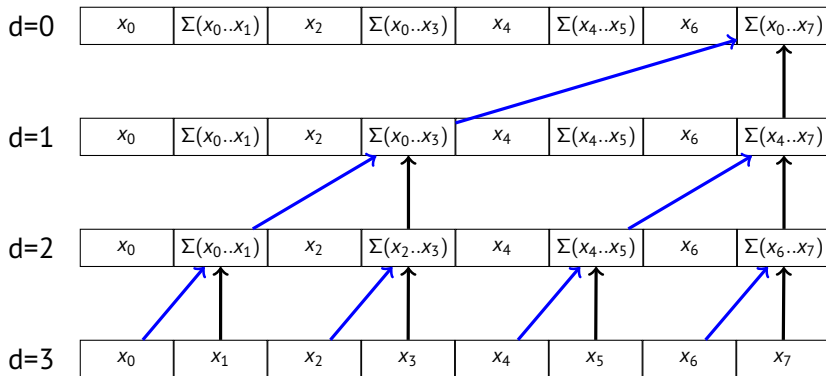
**The binary tree does not actually exist as a recursive pointer structure, but is just a communications concept.**

# Upsweep (“reduction phase”)



$$x_i^d = x_{i-2^{m-d-1}}^{d+1} + x_i^{d+1}$$

# Upsweep (“reduction phase”)

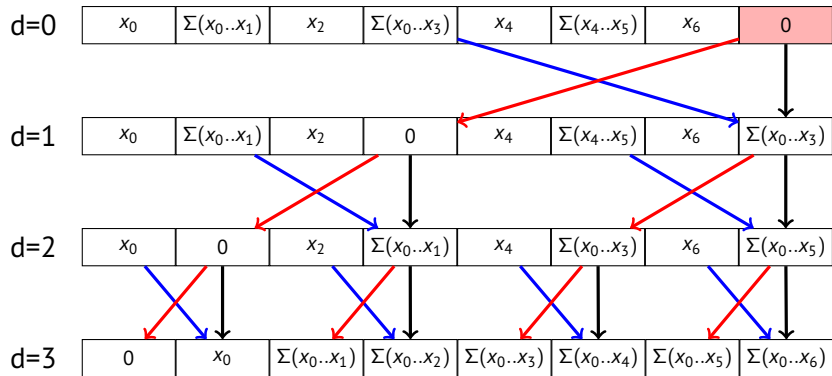


$$x_i^d = x_{i-2^{m-d-1}}^{d+1} + x_i^{d+1}$$

**Work:** For  $n = 2^m$ ,  $O(\sum_{i < m} 2^i) = O(n)$

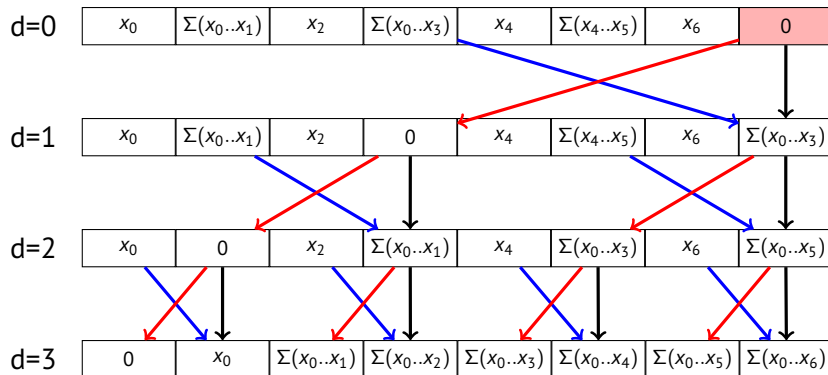
**Span:**  $\log(n)$

# Downsweep



Inverse indexing of the upsweep phase.

# Downsweep



Inverse indexing of the upsweep phase.

**Work:** For  $n = 2^m$ ,  $O(\sum_{i \leq m} 2^i) = O(n)$

**Span:**  $\log(n)$

# Work efficient scan

## Complexity of *scan* on size- $n$ input

Work:  $O(n)$

Span:  $\log(n)$

- Optimal, as *reduce* is the same.
- Can now depend on scan as a relatively cheap building block.

**Real-world scan implementations are often very different for technical reasons, but we can depend on these asymptotics when analysing and designing parallel algorithms.**

Parallel cost models

Prefix sums (scans)

Using scans

Auxiliary



## Filtering

Suppose we wish to remove negative elements from the list

```
let as = [-1, 2, -3, 4, 5, -6]
```

## Filtering

Suppose we wish to remove negative elements from the list

```
let as = [-1, 2, -3, 4, 5, -6]
```

For each element, see if we want to keep it:

```
let keep = map (\a -> if a >= 0 then 1 else 0) as  
-- [ 0, 1, 0, 1, 1, 0]
```

## Filtering

Suppose we wish to remove negative elements from the list

```
let as = [-1, 2, -3, 4, 5, -6]
```

For each element, see if we want to keep it:

```
let keep = map (\a -> if a >= 0 then 1 else 0) as  
-- [ 0, 1, 0, 1, 1, 0]
```

```
let offsets1 = scan (+) 0 keep  
-- [ 0, 1, 1, 2, 3, 3]
```

## Filtering

Suppose we wish to remove negative elements from the list

```
let as = [-1, 2, -3, 4, 5, -6]
```

For each element, see if we want to keep it:

```
let keep = map (\a -> if a >= 0 then 1 else 0) as  
-- [ 0, 1, 0, 1, 1, 0]
```

```
let offsets1 = scan (+) 0 keep  
-- [ 0, 1, 1, 2, 3, 3]
```

```
let offsets = map (\x -> x - 1) offsets1  
-- [-1, 0, 0, 1, 2, 2]
```

# Filtering

Suppose we wish to remove negative elements from the list

```
let as = [-1, 2, -3, 4, 5, -6]
```

For each element, see if we want to keep it:

```
let keep = map (\a -> if a >= 0 then 1 else 0) as  
-- [ 0, 1, 0, 1, 1, 0]
```

```
let offsets1 = scan (+) 0 keep  
-- [ 0, 1, 1, 2, 3, 3]
```

```
let offsets = map (\x -> x - 1) offsets1  
-- [-1, 0, 0, 1, 2, 2]
```

offsets[i] now indicates position in filtered list iff

```
keep[1] == 1
```

# scatter

scatter xs is vs computes equivalent of the imperative pseudocode

```
for j < n:
```

```
    xs[is[j]] = vs[j]
```

- Out-of-bound writes are ignored
- Writing different values to same index is *undefined*<sup>2</sup>
- Work  $O(n)$ , span  $O(1)$

**Just what we need for filtering!**

---

<sup>2</sup>reduce\_by\_index handles conflicts with provided operator.

# scatter

scatter xs is vs computes equivalent of the imperative pseudocode

```
for j < n:
```

```
  xs[is[j]] = vs[j]
```

- Out-of-bound writes are ignored
- Writing different values to same index is *undefined*<sup>2</sup>
- Work  $O(n)$ , span  $O(1)$

**Just what we need for filtering!**

```
scatter (replicate (last offsets1) 0)
      (map2 (\i k -> if k == 1 then i else -1)
            offsets keep)
      as
```

---

<sup>2</sup>reduce\_by\_index handles conflicts with provided operator.

## Implementing filter

```
def filter [n] 'a (p: a -> bool) (as: [n]a): []a =  
  let keep = map (\a -> if p a then 1 else 0) as  
  let offsets1 = scan (+) 0 keep  
  let num_to_keep = n == 0 || last offsets1 == 0  
  in if num_to_keep  
    then []  
    else scatter (replicate num_to_keep as[0])  
      (map2 (\i k -> if k == 1  
                    then i-1  
                    else -1)  
             offsets1 keep)  
    as
```



# Radix sort

- Many classical sorting algorithms are a poor fit for data parallelism, but *radix sort* works well.
- Radix-2 sort works by repeatedly partitioning elements according to one bit at a time, while preserving the ordering of the previous steps.

## Example with radix-10

3 2 6  
4 5 3  
6 0 8  
8 3 5  
7 5 1  
4 3 5  
7 0 4  
6 9 0

⇒

6	9	0
7	5	1
4	5	3
7	0	4
8	3	5
4	3	5
3	2	6
6	0	8

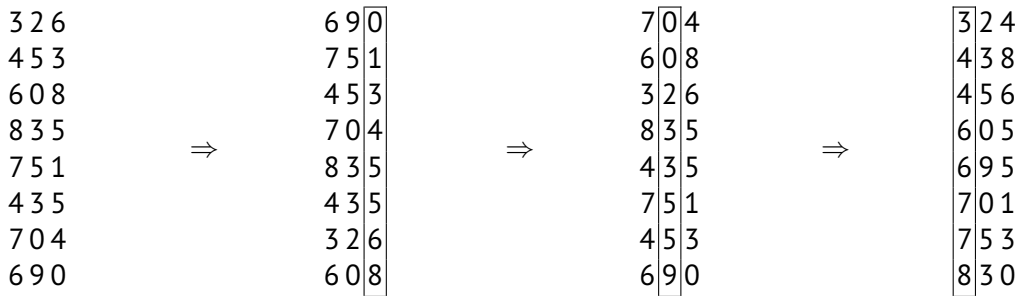
⇒

7	0	4
6	0	8
3	2	6
8	3	5
4	3	5
7	5	1
4	5	3
6	9	0

⇒

3	2	4
4	3	8
4	5	6
6	0	5
6	9	5
7	0	1
7	5	3
8	3	0

## Example with radix-10



- **Radix sort is not as general as a comparison-based sort.**
- Assumes sorting key can be decomposed into “digits”.

## Sorting `xs : [n]u32` by bit `b`

```
let bits = map (u32.get_bit b) xs
let bits_neg = map (1-) bits
let offs = reduce (+) 0 bits_neg
```

## Sorting xs : [n]u32 by bit b

```
let bits = map (u32.get_bit b) xs
let bits_neg = map (1-) bits
let offs = reduce (+) 0 bits_neg
```

### Example

```
b          = 0
xs         = [0, 1, 2, 3, 4]
bits       = [0, 1, 0, 1, 0]
bits_neg   = [1, 0, 1, 0, 1]
offs       = 3
```

```
let idxs0 = map2 (*)  
              bits_neg  
              (scan (+) 0 bits_neg)  
let idxs1 = map2 (*)  
              bits  
              (map (+offs) (scan (+) 0 bits))
```

```
let idxs0 = map2 (*)  
              bits_neg  
              (scan (+) 0 bits_neg)  
let idxs1 = map2 (*)  
              bits  
              (map (+offs) (scan (+) 0 bits))
```

## Example

```
bits           = [0, 1, 0, 1, 0]  
bits_neg       = [1, 0, 1, 0, 1]  
offs           = 3  
idxs0          = [1, 0, 2, 0, 3]  
idxs1          = [0, 4, 0, 5, 0]  
map2 (+) idxs0 idxs1 = [1, 4, 2, 5, 3]
```

**Then scatter as when filtering.**

## The whole step

```
def radix_sort_step [n] (xs: [n]u32) (b: i32): [n]u32 =  
  let bits = map (u32.get_bit b) xs  
  let bits_neg = map (1-) bits  
  let offs = reduce (+) 0 bits_neg  
  let idxs0 = map2 (*) bits_neg  
    (scan (+) 0 bits_neg)  
  let idxs1 = map2 (*) bits  
    (map (+offs) (scan (+) 0 bits))  
  let idxs2 = map2 (+) idxs0 idxs1  
  let idxs  = map (\x->x-1) idxs2  
  let xs' = scatter (copy xs) idxs xs  
  in xs'
```



## Radix sort in Futhark

```
def radix_sort [n] (xs: [n]u32): [n]u32 =  
    loop xs for i < 32 do radix_sort_step xs i
```

See worked example at

<https://futhark-lang.org/examples/radix-sort.html>

## Segmented scan

```
val segmented_scan [n] 't
  : (op: t -> t -> t) -> (ne: t)
  -> (flags: [n]bool) -> (as: [n]t)
  -> [n]t
```

true starts a segment and false continues a segment.

### Example

```
segmented_scan (+) 0
  [true, false, true, false, false, true]
  [0, 1, 2, 3, 4, 5]
== scan (+) 0 [0,1] ++
   scan (+) 0 [2,3,4] ++
   scan (+) 0 [5]
== [0, 1, 2, 5, 9, 5]
```

## Segmented reduction

```
val segmented_reduce [n] 't
  : (op: t -> t -> t) -> (ne: t)
  -> (flags: [n]bool) -> (as: [n]t)
  -> []t
```

### Example

```
segmented_reduce (+) 0
  [true, false, true, false, false, true]
  [0, 1, 2, 3, 4, 5]
== [reduce (+) 0 [0,1],
    reduce (+) 0 [2,3,4],
    reduce (+) 0 [5]]
== [1, 9, 5]
```

# Generalised histograms

Like scatter, but uses a provided reduce-like operator to handle multiple writes to same index.

## Type

```
val reduce_by_index [k] [n] 'a :  
    (dest: *[k]a)  
    -> (f: a -> a -> a) -> (ne: a)  
    -> (is: [n]i64) -> (vs: [n]a) -> *[k]a
```

## Semantics

```
for index in 0..k-1:  
    i = is[index]  
    v = vs[index]  
    dest[i] = f(dest[i], v)
```

Futhark uses parallel implementation  
with GPU *atomics*.

## Proving associativity and neutral elements

```
let op (x, i) (y, j) : (i32, i32) =  
  if x < y then (y, j) else (x, i)
```

```
let argmax [n] (xs: [n]i32) =  
  reduce op  
    (i32.smallest, -1)  
    (zip xs (iota n))
```

- Is op associative?
- Is (i32.smallest, -1) a neutral element?

## argmax: associativity

First, inline definitions:

```
(a 'op' b) 'op' c
== ((ax, ai) 'op' (bx, bi)) 'op' (cx, ci)
== let (x, i) = if ax < bx then (bx, bi)
                        else (ax, ai)
   in if x < cx then (cx, ci)
                        else (x, i)
```

---

```
a 'op' (b 'op' c)
== (ax, ai) 'op' ((bx, bi) 'op' (cx, ci))
== let (x, i) = if bx < cx then (cx, ci)
                        else (bx, bi)
   in if ax < x then (x, i)
                        else (ax, ai)
```

Then enumerate all possible comparisons between ax, bx, and cx and show that these two expressions are equivalent.

E.g. for `!(ax < bx) && bx < cx && cx < ax`

```
let (x, i) = if ax < bx then (bx, bi)
              else (ax, ai)
in if x < cx then (cx, ci)
    else (x, i)
== if ax < cx then (cx, ci)
    else (ax, ai)
== (ax, ai)
```

---

```
let (x, i) = if bx < cx then (cx, ci)
              else (bx, bi)
in if ax < x then (x, i)
    else (ax, ai)
== if ax < cx then (cx, ci)
    else (ax, ai)
== (ax, ai)
```



## argmax: neutral element

Similarly, by equational reasoning.

```
(a 'op' (i32.smallest, -1))  
== ((x, i) 'op' (i32.smallest, -1))  
== if x < i32.smallest then (i32.smallest, -1)  
    else (x, i)  
== (x, i)
```

---

```
((i32.smallest, -1) 'op' a)  
== ((i32.smallest, -1) 'op' (x, i))  
== if i32.smallest < x then (x, i)  
    else (i32.smallest, -1)  
== (x, i)
```





## A more calculational approach

<https://byorgey.wordpress.com/2020/02/23/what-would-dijkstra-do-proving-the-associativity-of-min/>

- Worth a read!
- More elegant and concise, but requires more creative thinking to characterise a useful property of the operator.

## Commutativity?

**Exercise for home:** The `argmax` operator is not commutative. Try to come up with a counterexample, and see if you can change its definition such that it becomes commutative.

# Commutativity?

**Exercise for home:** The `argmax` operator is not commutative. Try to come up with a counterexample, and see if you can change its definition such that it becomes commutative.

## Commutative reductions

Futhark has a `reduce_comm` function that can be used for commutative operators. This runs faster than normal `reduce`. Not necessary for built-in operators.

# Summary

- *Work* measures the total number of operations, *span* measures the longest chain of dependencies.
- Language-based cost models let us reason about program performance in a hardware-agnostic and composable way.
- Scans are a useful building block in advanced data parallel algorithms, but an efficient implementation is not straightforward.