Guest Lecture on Data Parallel Programming

Troels Henriksen

Sunday 13th September, 2026 at 18:12

Who am I

- Researcher at the University of Copenhagen.
- I work on
  - Data parallel programming
  - Data parallel algorithms
  - Programming language design and implementation
  - High performance computing, particularly on GPUs
  - Compiler optimisations
  - Various other programming language topics
- My main *practical* research artifact is the *Futhark programming language*.

Two kinds of parallelism

Task parallelism
Simultaneously doing different things to different data

- E.g., multithreading, message passing.
- Usually nondeterministic.

Data parallelism
Simultaneously doing *the same thing* to different data.

- Bulk operations on entire collections of data.
- E.g., `map`!
- Usually deterministic.

You already know data parallel programming!

You already know data parallel programming!

If `x` and `y` are vectors, then `x + y` is a data parallel operation.

You already know data parallel programming!

If `x` and `y` are vectors, then `x + y` is a data parallel operation.

- NumPy, R, Matlab, Julia, SQL are all data parallel models.
- Completely sequential and deterministic semantics.
- Parallel *execution* straightforward.
- **Distinguish between parallel execution and parallel potential.**
  - Implementations can always be parallelised as long as the code is written in a parallel style.

We will make this notion precise in a bit.

Data parallel vocabulary

Data parallel programming involves expressing operations on *entire data collections* at a time—in our case, arrays.

`map`

$$\texttt{map}\ f\ [x_0, \ldots, x_{n-1}] = [f\ x_0, \ldots, f\ x_{n-1}]$$

`sum`

$$\texttt{sum}\ [x_0, \ldots, x_{n-1}] = x_0 + \ldots + x_{n-1}$$

Summation, sequentially

```
def sum (xs: []i32) =
  loop y = 0 for i < length xs do
    y + xs[i]
```

Is this good?

Summation, in parallel

```
def sum (xs: []i32) =
  loop ys = xs while length ys > 1 do
    let ys_first = take (length ys / 2) ys
    let ys_last  = drop (length ys / 2) ys
    in map2 (+) ys_first ys_last
```

Summation, in parallel

```
def sum (xs: []i32) =
  loop ys = xs while length ys > 1 do
    let ys_first = take (length ys / 2) ys
    let ys_last  = drop (length ys / 2) ys
    in map2 (+) ys_first ys_last
```

```
ys       =  [1, 2, 3, 4, 5, 6, 7, 8]
ys_first =           [1, 2,  3,  4]
ys_last  =           [5, 6,  7,  8]
---------------------------------------
ys       =           [6, 8, 10, 12]
ys_first =                [ 6,  8]
ys_last  =                [10, 12]
---------------------------------------
ys       =                [16, 20]
ys_first =                    [16]
ys_last  =                    [20]
---------------------------------------
ys       =                    [36]
```

Comparison

```
def sum (xs: []i32) =
  loop y = 0 for i < length xs do
    y + xs[i]
```

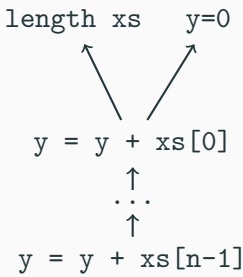
$O(n)$ work, but sequential.

```
def sum (xs: []i32) =
  loop ys = xs while length ys > 1 do
    let ys_first = take (length ys / 2) ys
    let ys_last  = drop (length ys / 2) ys
    in map2 (+) ys_first ys_last
```

$O(n)$ work, but parallel.

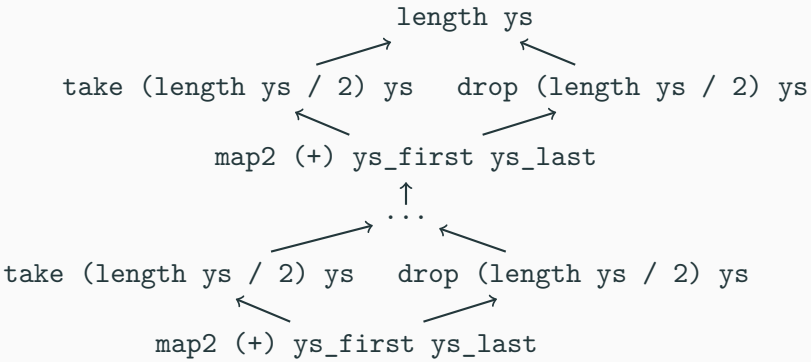
We need a framework in which we can be precise about why the parallel version is better.

Computation DAG for sequential `sum`



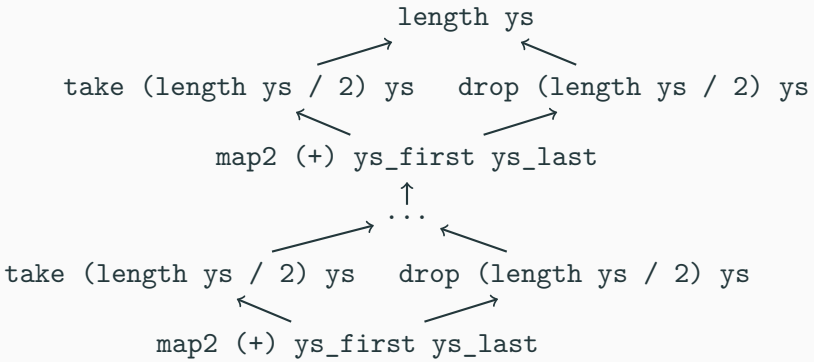
- Total count of nodes is the *work*, $W(p)$.
- Length of longest path from a leaf to the root is the *span*.
- **With an infinite number of processors, if a program p has span k, written $S(p) = k$, the program can execute in $O(k)$ time.**
- Here, $W(p) = O(n)$, $S(p) = O(n)$.

Computation DAG for parallel `sum`



What is the work and span complexity?

Computation DAG for parallel `sum`



What is the work and span complexity?

$W(p) = O(n)$ $\quad$ $S(p) = O(\log_2 n)$

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* (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.

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.

Wait, is that summation correct?

In our summation, we changed

$$(((((((x_0 + x_1) + x_2) + x_3) + x_4) + x_5) + x_6) + x_7)$$

to

$$((x_0 + x_4) + (x_2 + x_6)) + ((x_1 + x_5) + (x_3 + x_7))$$

Is that valid?

Wait, is that summation correct?

In our summation, we changed

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7)$$

to

$$((x_0 \oplus x_4) \oplus (x_2 \oplus x_6)) \oplus ((x_1 \oplus x_5) \oplus (x_3 \oplus x_7))$$

Is that valid?

What about now?

Wait, is that summation correct?

In our summation, we changed

$$(((((((x_0 \oplus x_1) \oplus x_2) \oplus x_3) \oplus x_4) \oplus x_5) \oplus x_6) \oplus x_7)$$

to

$$((x_0 \oplus x_4) \oplus (x_2 \oplus x_6)) \oplus ((x_1 \oplus x_5) \oplus (x_3 \oplus x_7))$$

Is that valid?

What about now?

In general, what must hold for an operator $\oplus$ for such a rewrite to be valid?

Commutativity and associativity

A binary operator $\oplus : \alpha \to \alpha \to \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

Commutativity and associativity

A binary operator $\oplus : \alpha \to \alpha \to \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

It is *associative* if

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

If we are a little more clever, we can do a parallel "sum" with any associative operator.

Commutativity and associativity

A binary operator $\oplus : \alpha \to \alpha \to \alpha$ is said to be *commutative* if

$$x \oplus y = y \oplus x$$

It is *associative* if

$$x \oplus (y \oplus z) = (x \oplus y) \oplus z$$

If we are a little more clever, we can do a parallel "sum" with any associative operator.

For convenience, we also tend to require a *neutral element* $0_\oplus$:

$$x \oplus 0_\oplus = 0_\oplus \oplus x = x$$

Monoidal summation

An associative binary operator $\oplus : \alpha \to \alpha \to \alpha$ with a neutral element $0_\oplus$ is called a *monoid* and written $(\oplus, 0_\oplus)$[1]. Examples:

- $(+, 0)$
- $(*, 1)$
- $(++, [])$

[1] A monoid without a neutral element is called a *semigroup*

Monoidal summation

An associative binary operator $\oplus : \alpha \to \alpha \to \alpha$ with a neutral element $0_\oplus$ is called a *monoid* and written $(\oplus, 0_\oplus)$[1]. Examples:

- $(+, 0)$
- $(*, 1)$
- $(++, [])$

For any monoid $(\oplus, 0_\oplus)$ we can construct a function `sum` with *semantics*

$$\texttt{sum}([x_0, \ldots, x_{n-1}]) = x_0 \oplus \ldots \oplus x_{n-1}$$

but which can *operationally* be executed in parallel.

[1] A monoid without a neutral element is called a *semigroup*

Monoidal reduction

We can abstract the notion of summation into a higher-order function typically called `reduce`:

$$\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to \alpha$$

Then

$$\texttt{sum} = \texttt{reduce}\ (+)\ 0$$

Monoidal reduction

We can abstract the notion of summation into a higher-order function typically called `reduce`:

$$\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to []\alpha \to \alpha$$

Then

$$\texttt{sum} = \texttt{reduce}\ (+)\ 0$$

This is one of the main components of the data parallel vocabulary.

The Futhark programming language

- A *functional array language*.
- Parallel programming with data-parallel combinators.
- Compiles to efficient parallel CPU or GPU code.
- Feels like a restricted subset of standard functional languages.

```
def inc2 [n] (xs: [n]i32) : [n]i32 = map (+2) xs

def sum [n] (xs: [n]i32) : i32 = reduce (+) 0 xs

def sumrows [n][m] (xss: [n][m]i32) : [n]i32 = map sum xss
```

Core restrictions

No irregular arrays: `[[1], [2,3]]` is a type error.

Core restrictions

No irregular arrays: `[[1], [2,3]]` is a type error.

No recursive data types: while recursive *functions* are allowed, they are discouraged.

Core restrictions

No irregular arrays: `[[1], [2,3]]` is a type error.

No recursive data types: while recursive *functions* are allowed, they are discouraged.

No side effects: completely pure.

Core restrictions

No irregular arrays: `[[1], [2,3]]` is a type error.

No recursive data types: while recursive *functions* are allowed, they are discouraged.

No side effects: completely pure.

Some limitations on first class functions:

- Cannot return from `if`
- Cannot be variant in `loop`
- Cannot recurse in polymorphic way.

Restrictions rooted in desire for good performance, and all checked at compile-time.

As a language

Not a language for writing applications—a Futhark program is compiled to a *library* and then invoked from other languages.

Some material:

- `https://futhark-lang.org`
- `https://futhark-lang.org/docs/prelude/`
- `https://futhark.readthedocs.io/en/latest/versus-other-languages.html`
- `https://futhark.readthedocs.io/en/latest/c-api.html`

But my focus today is general **data parallel thinking**, so we will see only a small part of the language.

map and reduce

```
-- Work: O(n * W(f))
-- Span: O(1 * S(f))
val map [n] 'a 'b : (f: a -> b) -> [n]a -> [n]b

-- Work: O(n * W(op))
-- Span: O(log(n) * S(op))
val reduce [n] 'a : (op: a -> a -> a) -> a -> [n]a -> a
```

`map` and `reduce`

```
-- Work: O(n * W(f))
-- Span: O(1 * S(f))
val map [n] 'a 'b : (f: a -> b) -> [n]a -> [n]b

-- Work: O(n * W(op))
-- Span: O(log(n) * S(op))
val reduce [n] 'a : (op: a -> a -> a) -> a -> [n]a -> a
```

```
-- Work: O(n)
-- Span: O(1)
val zip [n] 'a 'b : [n]a -> [n]b -> [n](a,b)
```

map and reduce

```
-- Work: O(n * W(f))
-- Span: O(1 * S(f))
val map [n] 'a 'b : (f: a -> b) -> [n]a -> [n]b

-- Work: O(n * W(op))
-- Span: O(log(n) * S(op))
val reduce [n] 'a : (op: a -> a -> a) -> a -> [n]a -> a

-- Work: O(n)
-- Span: O(1)
val zip [n] 'a 'b : [n]a -> [n]b -> [n](a,b)

-- Work: O(n * W(f))
-- Span: O(1 * S(f))
val map2 [n] 'a 'b 'c : (f: a -> b -> c) -> [n]a -> [n]b -> [n]c
```

Matrix multiplication

```
-- Work: O(n*m)
-- Span: O(1)
val transpose [n] [m] 'a : [n][m]a -> [m][n]a
```

Matrix multiplication

```
-- Work: O(n*m)
-- Span: O(1)
val transpose [n] [m] 'a : [n][m]a -> [m][n]a
```

```
-- Work: O(n)
-- Span: O(log(n))
def dotprod [n] (x: [n]f64) (y: [n]f64) : f64 =
  reduce (+) 0 (map2 (*) x y)
```

Matrix multiplication

```
-- Work: O(n*m)
-- Span: O(1)
val transpose [n] [m] 'a : [n][m]a -> [m][n]a
```

```
-- Work: O(n)
-- Span: O(log(n))
def dotprod [n] (x: [n]f64) (y: [n]f64) : f64 =
  reduce (+) 0 (map2 (*) x y)
```

```
def matmul [n] [m] [p] (A: [n][m]f64) (B: [m][p]f64) : [n][p]f64 =
  map (\A_row ->
         map (\B_col ->
                dotprod A_row B_col)
             (transpose B))
      A
```

What is the work and span?

Matrix multiplication

```
-- Work: O(n*m)
-- Span: O(1)
val transpose [n] [m] 'a : [n][m]a -> [m][n]a
```

```
-- Work: O(n)
-- Span: O(log(n))
def dotprod [n] (x: [n]f64) (y: [n]f64) : f64 =
  reduce (+) 0 (map2 (*) x y)
```

```
def matmul [n] [m] [p] (A: [n][m]f64) (B: [m][p]f64) : [n][p]f64 =
  map (\A_row ->
         map (\B_col ->
                dotprod A_row B_col)
             (transpose B))
      A
```

What is the work and span? $W = O(n * p * m), S = O(log(m))$.

Scan

```
-- Work: O(n * W(op))
-- Span: O(log(n) * S(op))
val scan [n] 'a : (op: a -> a -> a) -> a -> [n]a -> [n]a
```

Semantically, a scan is the reduce of all prefixes of an array.

```
scan op ne xs = [reduce op ne xs[:1],
                 reduce op ne xs[:2],
                 ...,
                 reduce op ne xs[:n]]
```

In practice it is implemented more efficiently.

Example of `scan`

```
> scan (+) 0 [1,2,3,4,5]
[1, 3, 6, 10, 15]
```

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 if `keep[i] == 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*.[2]
- Work $O(n)$, span $O(1)$.

Just what we need for filtering!

[2] `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*.[2]
- 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
```

[2] `reduce_by_index` handles conflicts with provided operator.

Implementing a polymorphic `filter`

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

The `copy` is because `scatter` *consumes* its first argument.

Working with irregular arrays

Forbidden in Futhark: `[[1], [2,3], [4,5,6]]`

We can still work with such values by *encoding* them in ways that are allowed.

Working with irregular arrays

Forbidden in Futhark: `[[1], [2,3], [4,5,6]]`

We can still work with such values by *encoding* them in ways that are allowed.

We represent irregular arrays as a pair of two vectors of the same size.

Value vector: one-dimensional vector of values, e.g.

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

Boolean vector: denotes when a new subarray begins, e.g.

```
[true, true, false, true, false, false]
```

We can then define "segmented operations" on this encoding—a big topic, but we will look at one example.

Segmented scans

```
val segscan [n] 't
   :  (op: t -> t -> t) -> (ne: t)
   -> (fs: [n]bool) -> (vs: [n]t)
   -> [n]t
```

`true` starts a segment and `false` continues a segment.

Example

```
segscan (+) 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]
```

Implementation of `segscan`

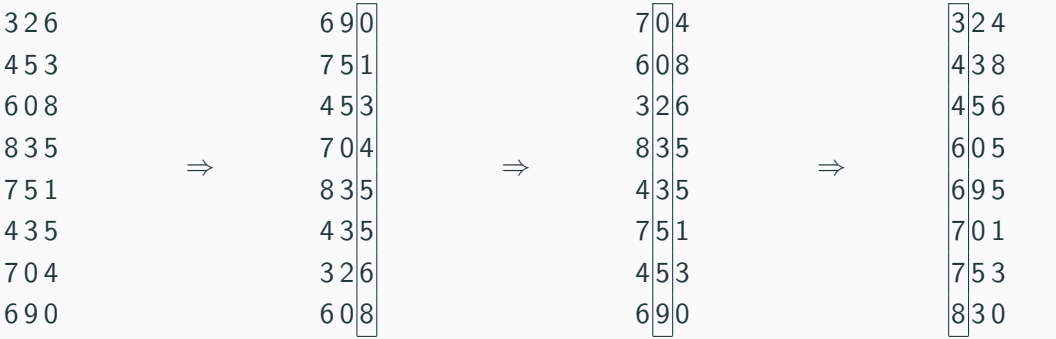
```
def segscan 't [n]
            (op: t -> t -> t) (ne: t)
            (fs: [n]bool) (vs: [n]t) : [n]t =
  let pairs =
    scan (\(v1, f1) (v2, f2) ->
            let f = f1 || f2
            let v = if f2 then v2 else op v1 v2
            in (v, f))
         (ne, false)
         (zip vs fs)
  let (res, _) = unzip pairs
  in res
```

- Having an intuitive understanding for why this works is not required.
- We can just treat this as another primitive.

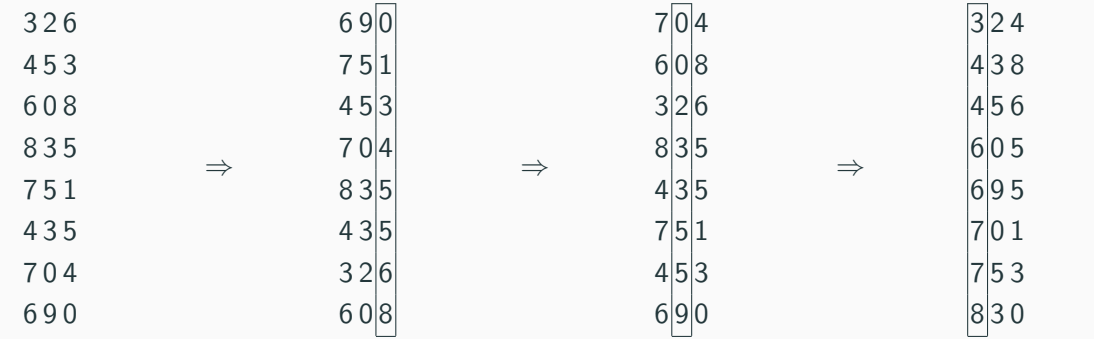
Radix sort

- 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.
  - \+ *Stable* partitioning.

Example with radix-10



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

Conclusions

- Data parallel programming is about bulk operations on entire collections of data.
- Compelling because it fits hardware well, but is also semantically easy to understand for humans because of its determinism.
- Requires us to think about algorithms in a different way.

Conclusions

- Data parallel programming is about bulk operations on entire collections of data.
- Compelling because it fits hardware well, but is also semantically easy to understand for humans because of its determinism.
- Requires us to think about algorithms in a different way.
- Futhark is not the only language for data parallel programming, but it is the best one.
  - Others include: APL (and its variants), Accelerate (for Haskell), Jax, Single-Assignment C, Julia, Dex.
- You can also do data parallel programming in normal languages, but it may not run so fast.