

# Flattening Irregular Nested Parallelism

Cosmin E. Oancea  
`cosmin.oancea@diku.dk`

Department of Computer Science (DIKU)  
University of Copenhagen

December 2023 DPP Lecture Slides

## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

- Flattening Recipe

- Irregular Multi-Dimensional Array Representation

- Flattening Recipe (Redundant)

- Rules For Flattening

- Flattening by Function Lifting

- Flattening Quicksort

- Flattening Prime-Number (Sieve) Computation

## Zip, Unzip, iota, replicate

- $\text{zip} : [n]_{\alpha_1} \rightarrow [n]_{\alpha_2} \rightarrow [n](\alpha_1, \alpha_2)$
- $\text{zip } [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [(a_1, b_1), \dots, (a_n, b_n)],$

## Zip, Unzip, iota, replicate

- $\text{zip} : [n]_{\alpha_1} \rightarrow [n]_{\alpha_2} \rightarrow [n](\alpha_1, \alpha_2)$
- $\text{zip } [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [(a_1, b_1), \dots, (a_n, b_n)],$
- $\text{unzip} : [n](\alpha_1, \alpha_2) \rightarrow ([n]_{\alpha_1}, [n]_{\alpha_2})$
- $\text{unzip } [(a_1, b_1), \dots, (a_n, b_n)] \equiv ([a_1, \dots, a_n], [b_1, \dots, b_n]),$
- In some sense zip/unzip are syntactic sugar

## Zip, Unzip, iota, replicate

- $\text{zip} : [n]_{\alpha_1} \rightarrow [n]_{\alpha_2} \rightarrow [n](\alpha_1, \alpha_2)$
- $\text{zip } [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [(a_1, b_1), \dots, (a_n, b_n)],$
- $\text{unzip} : [n](\alpha_1, \alpha_2) \rightarrow ([n]_{\alpha_1}, [n]_{\alpha_2})$
- $\text{unzip } [(a_1, b_1), \dots, (a_n, b_n)] \equiv ([a_1, \dots, a_n], [b_1, \dots, b_n]),$
- In some sense zip/unzip are syntactic sugar
- $\text{replicate} : (n: \text{int}) \rightarrow \alpha \rightarrow [n]_{\alpha}$
- $\text{replicate } n \ a \equiv [a, a, \dots, a],$

# Zip, Unzip, iota, replicate

- $\text{zip} : [n]_{\alpha_1} \rightarrow [n]_{\alpha_2} \rightarrow [n](\alpha_1, \alpha_2)$
- $\text{zip } [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [(a_1, b_1), \dots, (a_n, b_n)],$
- $\text{unzip} : [n](\alpha_1, \alpha_2) \rightarrow ([n]_{\alpha_1}, [n]_{\alpha_2})$
- $\text{unzip } [(a_1, b_1), \dots, (a_n, b_n)] \equiv ([a_1, \dots, a_n], [b_1, \dots, b_n]),$
- In some sense zip/unzip are syntactic sugar
- $\text{replicate} : (n: \text{int}) \rightarrow \alpha \rightarrow [n]\alpha$
- $\text{replicate } n \ a \equiv [a, a, \dots, a],$
- $\text{iota} : (n: \text{int}) \rightarrow [n]\text{int}$
- $\text{iota } n \equiv [0, 1, \dots, n-1]$

Note: in Haskell zip does not expect same-length arrays;  
in Futhark it does!

# Map, Reduce, and Scan Types and Semantics

- $[n]\alpha$  denotes the type of an array of  $n$  elements of type  $\alpha$ .
- $\text{map} : (\alpha \rightarrow \beta) \rightarrow [n]\alpha \rightarrow [n]\beta$   
 $\text{map } f [x_1, \dots, x_n] = [f \ x_1, \dots, f \ x_n],$   
i.e.,  $x_i : \alpha, \forall i$ , and  $f : \alpha \rightarrow \beta$ .
- $\text{reduce} : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [n]\alpha \rightarrow \alpha$   
 $\text{reduce } \odot \ e [x_1, x_2, \dots, x_n] = e \odot x_1 \odot x_2 \odot \dots \odot x_n,$   
i.e.,  $e : \alpha$ ,  $x_i : \alpha, \forall i$ , and  $\odot : \alpha \rightarrow \alpha \rightarrow \alpha$ .
- $\text{scan}^{\text{exc}} : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [n]\alpha \rightarrow [n]\alpha$   
 $\text{scan}^{\text{exc}} \odot \ e [x_1, \dots, x_n] = [e, e \odot x_1, \dots, e \odot x_1 \odot \dots \odot x_{n-1}]$   
i.e.,  $e : \alpha$ ,  $x_i : \alpha, \forall i$ , and  $\odot : \alpha \rightarrow \alpha \rightarrow \alpha$ .
- $\text{scan}^{\text{inc}} : (\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow [n]\alpha \rightarrow [n]\alpha$   
 $\text{scan}^{\text{inc}} \odot \ e [x_1, \dots, x_n] = [e \odot x_1, \dots, e \odot x_1 \odot \dots \odot x_n]$   
i.e.,  $e : \alpha$ ,  $x_i : \alpha, \forall i$ , and  $\odot : \alpha \rightarrow \alpha \rightarrow \alpha$ .

## Map2, Filter

- $\text{map2} : (\alpha_1 \rightarrow \alpha_2 \rightarrow \beta) \rightarrow [n]\alpha_1 \rightarrow [n]\alpha_2 \rightarrow [n]\beta$
- $\text{map2 } \odot [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [a_1 \odot b_1, \dots, a_n \odot b_n]$
- $\text{map3 } \dots$



## Map2, Filter

- $\text{map2} : (\alpha_1 \rightarrow \alpha_2 \rightarrow \beta) \rightarrow [n]\alpha_1 \rightarrow [n]\alpha_2 \rightarrow [n]\beta$
- $\text{map2} \odot [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [a_1 \odot b_1, \dots, a_n \odot b_n]$
- $\text{map3} \dots$
- $\text{filter} : (\alpha \rightarrow \text{Bool}) \rightarrow [n]\alpha \rightarrow [m]\alpha \ (m \leq n)$
- $\text{filter } p [a_1, \dots, a_n] = [a_{k_1}, \dots, a_{k_m}]$  such that  $k_1 < k_2 < \dots < k_m$ , and denoting  $\bar{k} = k_1, \dots, k_m$ , we have  $(p \ a_j == \text{true}) \ \forall j \in \bar{k}$ , **and**  $(p \ a_j == \text{false}) \ \forall j \notin \bar{k}$ .

Note: in Haskell `map2`, `map3` do not expect same-length arrays; in Futhark they do!

## Scatter: A Parallel Write Operator

Scatter **updates in parallel** a base array with a set of values at specified indices:

$\text{scatter} : *[m]\alpha \rightarrow [n]\text{int} \rightarrow [n]\alpha \rightarrow *[m]\alpha$

A (data vector) = [b0, b1, b2, b3]

I (index vector) = [2, 4, 1, -1]

X (input array) = [a0, a1, a2, a3, a4, a5]

$\text{scatter } X \mid A = [a0, b2, b0, a3, b1, a5]$

# Scatter: A Parallel Write Operator

Scatter **updates in parallel** a base array with a set of values at specified indices:

**scatter** :  $*[m]_{\alpha} \rightarrow [n]_{\text{int}} \rightarrow [n]_{\alpha} \rightarrow *[m]_{\alpha}$

A (data vector) = [b0, b1, b2, b3]

I (index vector) = [2, 4, 1, -1]

X (input array) = [a0, a1, a2, a3, a4, a5]

**scatter X I A** = [a0, b2, b0, a3, b1, a5]

**scatter** has  $D(n) = \Theta(1)$  and  $W(n) = \Theta(n)$ ,  
i.e., requires  $n$  update operations ( $n$  is the size of  $I$  or  $A$ , not of  $X$ !).

- 1 Array  $X$  is consumed by **scatter**; following uses of  $X$  are illegal!
- 2 Similarly,  $X$  can alias neither  $I$  nor  $A$ !

In Futhark, **scatter** check and ignores the indices that are out of bounds (no update is performed on those). This is useful for padding the iteration space in order to obtain regular parallelism.

## Partition2/Filter Implementation

`partition2: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[n]\alpha \rightarrow (\text{i32}, [n]\alpha)$`

In result, the elements satisfying the predicate occur before the others. **Can be implemented by means of map, scan, scatter.**

```
let partition2 't [n] (dummy: t)
  (cond: t -> bool) (X: [n]t) :
  (i64, [n]t) =
```

Assume  $X = [5, 4, 2, 3, 7, 8]$ , and  
cond is T(rue) for even nums.

```
let cs = map cond X
let tfs = map (\ f -> if f then 1
                      else 0) cs
let isT = scan (+) 0 tfs
let i = isT[n-1]

let ffs = map (\ f -> if f then 0
                      else 1) cs
let isF = map (+ i) <| scan (+) 0 ffs
let inds = map (\ (c, iT, iF) ->
                  if c then iT-1
                  else iF-1
                ) (zip3 cs isT isF)
let tmp = replicate n dummy
in (i, scatter tmp inds X)
```

## Partition2/Filter Implementation

`partition2: ( $\alpha \rightarrow \text{Bool}$ )  $\rightarrow$   $[n]\alpha \rightarrow (\text{i32}, [n]\alpha)$`

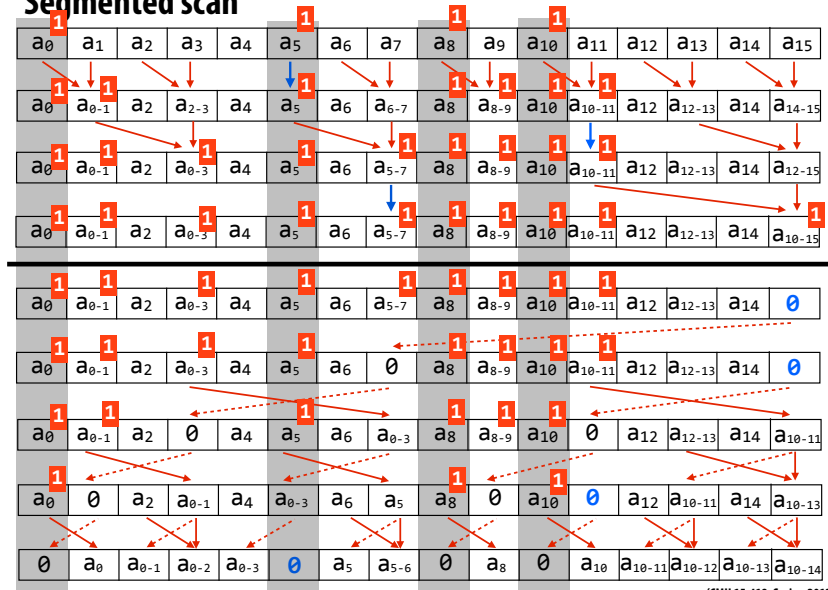
In result, the elements satisfying the predicate occur before the others. Can be implemented by means of `map`, `scan`, `scatter`.

```
let partition2 't [n] (dummy: t)
  (cond: t  $\rightarrow$  bool) (X: [n]t) :
  (i64, [n]t) =
  let cs = map cond X
  let tfs = map (\ f  $\rightarrow$  if f then 1
                        else 0) cs
  let isT = scan (+) 0 tfs
  let i = isT[n-1]

  let ffs = map (\ f  $\rightarrow$  if f then 0
                        else 1) cs
  let isF = map (+i) <| scan (+) 0 ffs
  let inds = map (\(c, iT, iF)  $\rightarrow$ 
                    if c then iT-1
                    else iF-1
                  ) (zip3 cs isT isF)
  let tmp = replicate n dummy
  in (i, scatter tmp inds X)
```

Assume  $X = [5, 4, 2, 3, 7, 8]$ , and  
cond is T(rue) for even nums.  
 $n = 6$   
 $cs = [F, T, T, F, F, T]$   
 $tfs = [0, 1, 1, 0, 0, 1]$   
 $isT = [0, 1, 2, 2, 2, 3]$   
 $i = 3$   
 $ffs = [1, 0, 0, 1, 1, 0]$   
 $isF = [4, 4, 4, 5, 6, 6]$   
 $inds = [3, 0, 1, 4, 5, 2]$   
 $flags = [3, 0, 0, 3, 0, 0]$   
Result = [4, 2, 8, 5, 3, 7]

## Segmented scan



# Segmented Scan Is a Sort of Scan

Futhark Implementation:

```
def sgmscan 't [n] (op: t->t->t) (ne: t)
    (flg : [n]i64) (arr : [n]t) : [n]t =
  let flgs_vals =
    scan ( \ (f1, x1) (f2,x2) ->
      let f = f1 | f2 in
      if f2 != 0 then (f, x2)
      else (f, op x1 x2) )
    (0,ne) (zip flg arr)
  let (_, vals) = unzip flgs_vals
  in vals
```

```
sgmscan (+) 0 [1,0,0,1,0, 0, 0]
               [1,2,3,4,5, 6, 7]
               = = = = = = =
               [1,3,6,4,9,15,22]
```

```
map ( \ row -> scan (+) 0 row)
    [[1,2,3], [4,5, 6, 7]]
    = = =     = = = = =
    [[1,3,6], [4,9,15,22]]
```

## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

- Flattening Recipe

- Irregular Multi-Dimensional Array Representation

- Flattening Recipe (Redundant)

- Rules For Flattening

- Flattening by Function Lifting

- Flattening Quicksort

- Flattening Prime-Number (Sieve) Computation



# How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in  
map (\i -> map (+(i+1)) (iota i)) arr  
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

Normalize the code:

# How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in
map (\i -> map (+(i+1)) (iota i)) arr
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

Normalize the code:

```
map (\i -> let ip1 = i+1 in
            let iot = (iota i) in
            let ip1r= (replicate i ip1) in
            map2 (+) ip1r iot                ) arr
```

Distribute the map over every instruction in the body

(bottom-up if  $\text{nest} > 2$ ), where  $\mathcal{F}$  denotes the flattening transf, and modify the inputs (results) accordingly.

# How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in
map (\i -> map (+(i+1)) (iota i)) arr
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

Normalize the code:

```
map (\i -> let ip1 = i+1 in
            let iot = (iota i) in
            let ip1r= (replicate i ip1) in
            map2 (+) ip1r iot                ) arr
```

Distribute the map over every instruction in the body

(bottom-up if  $\text{nest} > 2$ ), where  $\mathcal{F}$  denotes the flattening transf, and modify the inputs (results) accordingly.

```
 $\mathcal{F}(\text{map } (\lambda i \rightarrow \text{map } (+(i+1)) (\text{iota } i)) [0..n-1]) \equiv$   
1. let ip1s = map (\i -> i+1) arr in -- [2, 3, 4, 5]  
2. let iots =  $\mathcal{F}(\text{map } (\lambda i \rightarrow (\text{iota } i)) \text{arr})$  in  
3. let ip1rs=  $\mathcal{F}(\text{map2 } (\lambda i \text{ ip1} \rightarrow (\text{replicate } i \text{ ip1})) \text{arr } \text{ip1s})$   
4. in  $\mathcal{F}(\text{map2 } (\lambda \text{ ip1r } \text{iot} \rightarrow \text{map2 } (+) \text{ip1r } \text{iot}) \text{ip1rs } \text{iots})$ 
```

# How to Flatten? A Relatively Simple Case

**According to rule (4) iota nested inside a map**

(assuming `arr = [1,2,3,4]`):

```
2. let iots =  $\mathcal{F}$ (map (\i -> iota i) arr)
```

≡

```
inds = scanexc (+) 0 arr -- [0,1,3,6]
size = (last inds) + (last arr) -- 6 + 4 = 10
flag = scatter (replicate size 0)
      inds arr
--      [1, 2, 0, 3, 0, 0, 4, 0, 0, 0]
tmp  = replicate size 1
iots = sgmScanexc (+) 0 flag tmp -- [0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

# How to Flatten? A Relatively Simple Case

## According to rule (3) replicate nested inside a map

(assuming `arr = [1,2,3,4]`):

```
3. let ip1rs=  $\mathcal{F}$ (map2 (\ i ip1 -> replicate i ip1) arr ip1s)
   ≡
vals = scatter (replicate size 0) inds ip1s -- [2,3,0,4,0,0,5,0,0,0]
ip1rs= sgmScaninc (+) 0 flag vals           -- [2,3,3,4,4,4,5,5,5,5]
```

## According to rule (2) map nested inside a map

```
 $\mathcal{F}$ (map2 (\ ip1r iot -> map2 (+) ip1r iot) ip1rs iots)
≡
4. result = map (+) ip1rs iots
-- [2, 3, 3, 4, 4, 4, 5, 5, 5, 5]
-- [0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
-- + + + + + + + + +
-----
-- [2, 3, 4, 4, 5, 6, 5, 6, 7, 8] values
-- [1, 2, 0, 3, 0, 0, 4, 0, 0, 0] flags
```

## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

Flattening Recipe

**Irregular Multi-Dimensional Array Representation**

Flattening Recipe (Redundant)

Rules For Flattening

Flattening by Function Lifting

Flattening Quicksort

Flattening Prime-Number (Sieve) Computation

# Shape-Based Representation

- Two dimensional arrays:

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

⇒

$S_{arr}^0 = [4]$

$S_{arr}^1 = [3, 1, 0, 2]$

$D_{arr} = [1, 2, 3, 4, 5, 6]$

# Shape-Based Representation

- Two dimensional arrays:

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

⇒

```
Sarr0 = [4]
```

```
Sarr1 = [3, 1, 0, 2]
```

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

- Three dimensional arrays:

```
arr = [ [], [ [1,2,3], [4], [], [5,6] ], [ [7], [], [8,9,10] ] ]
```

⇒



# Shape-Based Representation

- Two dimensional arrays:

$arr = [ [1, 2, 3], [4], [], [5, 6] ]$

$\Rightarrow$

$S_{arr}^0 = [4]$

$S_{arr}^1 = [3, 1, 0, 2]$

$D_{arr} = [1, 2, 3, 4, 5, 6]$

- Three dimensional arrays:

$arr = [ [], [ [1, 2, 3], [4], [], [5, 6] ], [ [7], [], [8, 9, 10] ] ]$

$\Rightarrow$

$S_{arr}^0 = [3]$

$S_{arr}^1 = [0, 4, 3]$

$S_{arr}^2 = [3, 1, 0, 2, 1, 0, 3]$

$flen_{arr} = 10$

$D_{arr} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

Assume a n-dimensional array; The following invariant holds:

$length\ S_{arr}^i = reduce\ (+)\ 0\ S_{arr}^{i-1}, \forall 1 \leq i < n$

$length\ D_{arr} = reduce\ (+)\ 0\ S_{arr}^{n-1}$

# Flat Representation: Auxiliary Structures

```
arr = [ [], [ [1,2,3], [4], [], [5,6] ], [ [7], [], [8,9,10] ]
```

⇒

$S_{arr}^0 = [3]$

$S_{arr}^1 = [0, 4, 3]$

$S_{arr}^2 = [3, 1, 0, 2, 1, 0, 3]$

$D_{arr} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

- **Offset Indices (B):** segment-start offset in the flat data:

$B_{arr}^1 = [0, 0, 6]$

$B_{arr}^2 = [0, 3, 4, 4, 6, 7, 7]$

- **Flag Array (F):** start of a segment indicated by a value  $\neq 0$   
for example used for segmented scan operations:

$F_{arr}^1 = [1, 0, 0, 0, 0, 0, 1, 0, 0, 0]$

$F_{arr}^2 = [1, 0, 0, 1, 1, 0, 1, 1, 0, 0]$

- **Segment and Inner indices (II):**

$II_{arr}^1 = [1, 1, 1, 1, 1, 1, 2, 2, 2, 2]$

$II_{arr}^2 = [0, 0, 0, 1, 3, 3, 0, 2, 2, 2]$

$II_{arr}^3 = [0, 1, 2, 0, 0, 1, 0, 0, 1, 2]$

# Auxiliary Structures: Intuitive Motivation

Auxiliary structures are useful to optimize the replication of values.

## Nested-Execution Example:

```
let xss = [ [1,2,3], [], [5,7] ]  
let ys  = [ 4, 2, 1 ]  
let rss = map2 (\ xs y -> map (+y) xs ) xss ys
```

⇒

```
rss = [ map (+4) [1,2,3], map (+2) [], map (+1) [5,7] ]  
rss = [ [5,6,7], [], [6,8] ]
```

Traditional flattening would replicate the values of y:

```
let (Syss1, Dyss) =  $\mathcal{F}$ (map2 (\ xs y -> replicate (length xs) y) xss ys)  
let Drss = map2 (\ x y -> x + y) Dxss Dyss
```

⇒

```
Dxss = [ 1, 2, 3, 5, 7 ]  
      + + + + +  
Dyss = [ 4, 4, 4, 1, 1 ]  
      = = = = =  
Drss = [ 5, 6, 7, 6, 8 ]
```

# Auxiliary Structures: Intuitive Motivation

Auxiliary structures are useful to optimize the replication of values.

## Nested-Execution Example:

```
let xss = [ [1,2,3], [], [5,7] ]  
let ys  = [ 4, 2, 1 ]  
let rss = map2 (\ xs y -> map (+y) xs ) xss ys  
=>  
rss = [ map (+4) [1,2,3], map (+2) [], map (+1) [5,7] ]  
rss = [ [5,6,7], [], [6,8] ]
```

Using the auxiliary structures we indirectly access other arrays:

```
let Drss = map2 (\ x sgmind -> x + ys[sgmind]) Dxss II1rss  
=>  
S1rss = [3, 0, 2]  
II1rss = [0, 0, 0, 2, 2]  
Dxss = [1, 2, 3, 5, 7]  
Drss = [1+4, 2+4, 3+4, 5+1, 7+1] = [5, 6, 7, 6, 8]
```

But what have we gained? Creating  $II_{rss}^1$  is as expensive as  $xss$  (or better said the expanded  $yss$  from the other slide) ...

# Auxiliary Structures: Intuitive Motivation

Auxiliary structures are useful to optimize replication:

- they depend only on the shape of the result (created once)
- can indirectly access several lower-dimensional arrays, sharing parallel dimensions!

Nested-Execution Example:

```
let xss = [ [1,2,3], [], [5,7] ]
let ys  = [ 4, 2, 1 ]
let zs  = [ 1, 2, 3 ]
let rss = map3 (\ xs y z -> map (\ x -> x*y + z ) xs ) xss ys zs
⇒
rss = [ [5,9,13], [], [8,10] ]
```

Using the auxiliary structures we indirectly access other arrays:

```
let Drss = map2 (\ y sgmind -> x*ys[sgmind] + zs[sgmind]) Dxss II1rss
⇒
II1rss = [0, 0, 0, 2, 2]
Dxss = [1, 2, 3, 5, 7]
Drss = [1*4+1, 2*4+1, 3*4+1, 5*1+3, 7*1+3] = [5, 9, 13, 8, 10]
```

We build  $II_{rss}^1$  once and reuse it twice; also improves locality!

# Auxiliary Structures: Intuitive Motivation

## Nested-Execution Example:

```
let xss = [ [1,3], [2] ]
let yss = [ [2], [4,5] ]
let rss = map2 (\xs ys -> map (\x -> map (+x) ys ) xs ) xss yss
⇒
rss = [ [[3],[5]], [[6,7]] ]
```

Using the auxiliary structures we indirectly access other arrays:

```
let Drss = map3(\ s1 s2 s3 -> let ind_x = B1xss[s1] + s2
                                let ind_y = B1yss[s1] + s3
                                in X[ind_x] + Y[ind_y]
                                ) ||1rss ||2rss ||3rss
```

⇒

```
B1xss = [0, 2]
B1yss = [0, 1]
||1rss = [0, 0, 1, 1]
||2rss = [0, 1, 0, 0]
||3rss = [0, 0, 0, 1]
Drss = [ Dxss[0+0]+Dyss[0+0], Dxss[0+1]+Dyss[0+0]
          , Dxss[2+0]+Dyss[1+0], Dxss[2+0]+Dyss[1+1] ]
Drss = [ 1+2, 3+2, 2+4, 2+5 ] = [ 3, 5, 6, 7 ]
```

# Constructing the Offset Indices (B)

```
arr = [ [], [ [1,2,3], [4], [], [5,6] ], [ [7], [], [8,9,10] ]
```

⇒

```
S0arr = [3]
```

```
S1arr = [0, 4, 3]
```

```
S2arr = [3, 1, 0, 2, 1, 0, 3]
```

```
Darr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Offset Indices (B): segment-start offset in the flat data:

```
B1arr = [0, 0, 6]
```

```
B2arr = [0, 3, 4, 4, 6, 7, 7]
```

How to construct Offset Indices (B)?

# Constructing the Offset Indices (B)

```
arr = [ [], [ [1,2,3], [4], [], [5,6] ], [ [7], [], [8,9,10] ]
```

⇒

```
S0arr = [3]
```

```
S1arr = [0, 4, 3]
```

```
S2arr = [3, 1, 0, 2, 1, 0, 3]
```

```
Darr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Offset Indices (B): segment-start offset in the flat data:

```
B1arr = [0, 0, 6]
```

```
B2arr = [0, 3, 4, 4, 6, 7, 7]
```

How to construct Offset Indices (B)?

By using exclusive scan on the shape!

```
B2arr = scanexc (+) 0 S2arr — [0, 3, 4, 4, 6, 7, 7]
```

```
B1arr = scanexc (+) 0 S1arr — [0, 0, 4]  
      |> map (\i -> B2arr[i]) — [0, 0, 6]
```



# Constructing the Flag Array

From now on, we discuss only TWO-dimensional irregular arrays!

```
def mkFlagArray 't [m]
  (aoa_shp: [m]i64)(zero: t)      —aoa_shp=[0,3,1,0,4,2,0]
  (aoa_val: [m]t)      : []t =    —aoa_val=[1,1,1,1,1,1,1]
  let shp_rot = map (\i->if i==0 then 0 —shp_rot=[0,0,3,1,0,4,2]
                     else aoa_shp[i-1]
                     ) (iota m)
  let shp_scn = scan (+) 0 shp_rot —shp_scn=[0,0,3,4,4,8,10]
  let aoa_len = if m == 0 then 0
                else shp_scn[m-1]+aoa_shp[m-1] —aoa_len= 10
  let shp_ind = map2 (\shp ind -> —shp_ind=
                     if shp==0 then -1 — [-1,0,3,-1,4,8,-1]
                     else ind —scatter
                     ) aoa_shp shp_scn — [0,0,0,0,0,0,0,0,0,0]
  in scatter (replicate aoa_len zero) — [-1,0,3,-1,4,8,-1]
             shp_ind aoa_val — [1,1,1,1,1,1,1]
                               — F = [1,0,0,1,1,0,0,0,1,0]
```

Why do we need aoa\_val?

# Constructing the Flag Array

From now on, we discuss only TWO-dimensional irregular arrays!

```
def mkFlagArray 't [m]
  (aoa_shp: [m]i64)(zero: t)      —aoa_shp=[0,3,1,0,4,2,0]
  (aoa_val: [m]t)      : []t =    —aoa_val=[1,1,1,1,1,1,1]
  let shp_rot = map (\i->if i==0 then 0 —shp_rot=[0,0,3,1,0,4,2]
                     else aoa_shp[i-1]
                     ) (iota m)
  let shp_scn = scan (+) 0 shp_rot —shp_scn=[0,0,3,4,4,8,10]
  let aoa_len = if m == 0 then 0
                 else shp_scn[m-1]+aoa_shp[m-1] —aoa_len= 10
  let shp_ind = map2 (\shp ind -> —shp_ind=
                     if shp==0 then -1 — [-1,0,3,-1,4,8,-1]
                     else ind —scatter
                     ) aoa_shp shp_scn — [0,0,0,0,0,0,0,0,0,0]
  in scatter (replicate aoa_len zero) — [-1,0,3,-1,4,8,-1]
             shp_ind aoa_val — [1,1,1,1,1,1,1]
                          — F = [1,0,0,1,1,0,0,0,1,0]
```

Why do we need `aoa_val`?

Because there are many valid flag arrays, i.e., the start of the segment can be denoted by any value different than zero!

# Constructing the Segment and Inner Indices

From now on, we discuss only TWO-dimensional irregular arrays!

```
arr = [ [1,2,3], [4], [], [5,6], [7], [], [8,9,10] ]
```

⇒

```
Sarr0 = [7]
```

```
Sarr1 = [3, 1, 0, 2, 1, 0, 3]
```

```
flenarr = reduce (+) 0 Sarr1 = 10
```

```
Darr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Segment and Inner indices (II):

```
IIarr1 = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]
```

```
IIarr2 = [0, 1, 2, 0, 0, 1, 0, 0, 1, 2]
```

Constructing Segment and Inner indices (II):

```
Farr = mkFlagArray Sarr1 0 [1...length Sarr1]  
      — [1, 0, 0, 2, 4, 0, 5, 7, 0, 0]
```

```
IIarr1 = ???
```

```
IIarr2 = ???
```

# Constructing the Segment and Inner Indices

I need to get this:

$II_{arr}^1 = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]$

from this:

$F_{arr} = \text{mkFlagArray } S_{arr}^1 \ 0 \ [1 \dots \text{length } S_{arr}^1]$   
—  $[1, 0, 0, 2, 4, 0, 5, 7, 0, 0]$

How?

# Constructing the Segment and Inner Indices

I need to get this:

$$II_{arr}^2 = [0, 1, 2, 0, 0, 1, 0, 0, 1, 2]$$

from this:

$$F_{arr} = \text{mkFlagArray } S_{arr}^1 \ 0 \ [1 \dots \text{length } S_{arr}^1]$$

—  $[1, 0, 0, 2, 4, 0, 5, 7, 0, 0]$

How?

# Constructing the Segment and Inner Indices

From now on, we discuss only TWO-dimensional irregular arrays!

```
arr = [ [1,2,3], [4], [], [5,6], [7], [], [8,9,10] ]
```

⇒

```
Sarr0 = [7]
```

```
Sarr1 = [3, 1, 0, 2, 1, 0, 3]
```

```
flenarr = reduce (+) 0 Sarr1 = 10
```

```
Darr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Segment and Inner indices (II):

```
IIarr1 = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]
```

```
IIarr2 = [0, 1, 2, 0, 0, 1, 0, 0, 1, 2]
```

Constructing Segment and Inner indices (II):

```
Farr = mkFlagArray Sarr1 0 [1...length Sarr1]  
      — [1, 0, 0, 2, 4, 0, 5, 7, 0, 0]
```

```
IIarr1 = sgmscan (+) 0 Farr Farr |> map (-1)
```

```
IIarr2 = sgmscan (+) 0 Farr (replicate flen 1) |> map (-1)
```

## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

Flattening Recipe

Irregular Multi-Dimensional Array Representation

**Flattening Recipe (Redundant)**

Rules For Flattening

Flattening by Function Lifting

Flattening Quicksort

Flattening Prime-Number (Sieve) Computation

# How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in  
map (\i -> map (+(i+1)) (iota i)) arr  
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```



# How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in
map (\i -> map (+(i+1)) (iota i)) arr
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

Normalize the code:

```
map (\i -> let ip1 = i+1 in
            let iot = (iota i) in
            let ip1r= (replicate i ip1) in
            map2 (+) ip1r iot                ) arr
```

Distribute the map over every instruction in the body

(bottom-up if  $\text{nest} > 2$ ), where  $\mathcal{F}$  denotes the flattening transf, and modify the inputs (results) accordingly.

# How to Flatten? A Relatively Simple Case

```
let arr = [1, 2, 3, 4] in
map (\i -> map (+(i+1)) (iota i)) arr
-- Result: [[2],[3,4],[4,5,6],[5,6,7,8]]
```

Normalize the code:

```
map (\i -> let ip1 = i+1 in
            let iot = (iota i) in
            let ip1r= (replicate i ip1) in
            map2 (+) ip1r iot                ) arr
```

Distribute the map over every instruction in the body

(bottom-up if  $\text{nest} > 2$ ), where  $\mathcal{F}$  denotes the flattening transf, and modify the inputs (results) accordingly.

```
 $\mathcal{F}(\text{map } (\lambda i \rightarrow \text{map } (+(i+1)) (\text{iota } i)) [0..n-1]) \equiv$ 
1. let ip1s = map (\i -> i+1) arr in -- [2, 3, 4, 5]
2. let iots =  $\mathcal{F}(\text{map } (\lambda i \rightarrow (\text{iota } i)) \text{ arr})$  in
3. let ip1rs=  $\mathcal{F}(\text{map2 } (\lambda i \text{ ip1} \rightarrow (\text{replicate } i \text{ ip1})) \text{ arr } \text{ip1s})$ 
4. in  $\mathcal{F}(\text{map2 } (\lambda \text{ ip1r } \text{iot} \rightarrow \text{map2 } (+) \text{ ip1r } \text{iot}) \text{ ip1rs } \text{iots})$ 
```

# How to Flatten? A Relatively Simple Case

**According to rule (4) iota nested inside a map**

(assuming `arr = [1,2,3,4]`):

```
2. let iots =  $\mathcal{F}$ (map (\i -> iota i) arr)
```

≡

```
inds = scanexc (+) 0 arr -- [0,1,3,6]
size = (last inds) + (last arr) -- 6 + 4 = 10
flag = scatter (replicate size 0)
      inds arr
--      [1, 2, 0, 3, 0, 0, 4, 0, 0, 0]
tmp  = replicate size 1
iots = sgmScanexc (+) 0 flag tmp --[0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

# How to Flatten? A Relatively Simple Case

## According to rule (3) replicate nested inside a map

(assuming `arr = [1,2,3,4]`):

```
3. let ip1rs= F(map2 (\ i ip1 -> replicate i ip1) arr ip1s)
≡
vals = scatter (replicate size 0) inds ip1s -- [2,3,0,4,0,0,5,0,0,0]
ip1rs= sgmScaninc (+) 0 flag vals           -- [2,3,3,4,4,4,5,5,5,5]
```

## According to rule (2) map nested inside a map

```
F(map2 (\ ip1r iot -> map2 (+) ip1r iot) ip1rs iots)
≡
4. result = map (+) ip1rs iots
-- [2, 3, 3, 4, 4, 4, 5, 5, 5, 5]
-- [0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
-- + + + + + + + + +
-----
-- [2, 3, 4, 4, 5, 6, 5, 6, 7, 8] values
-- [1, 2, 0, 3, 0, 0, 4, 0, 0, 0] flags
```

## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

Flattening Recipe

Irregular Multi-Dimensional Array Representation

Flattening Recipe (Redundant)

#### Rules For Flattening

Flattening by Function Lifting

Flattening Quicksort

Flattening Prime-Number (Sieve) Computation

# Nested vs Flattened Parallelism: Scan inside a Map

## (1) Scan inside a nested map:

```
res = map (\row->scaninc (+) 0 row) [[1,3], [2,4,6]]
```

≡

```
res = [ scaninc (+) 0 [1,3],      scaninc (+) 0 [2,4,6] ]
```

≡

```
res = [ [ 1, 4],                  [2, 6, 12] ]
```

# Nested vs Flattened Parallelism: Scan inside a Map

## (1) Scan inside a nested map:

```
res = map (\row->scaninc (+) 0 row) [[1,3], [2,4,6]]  
≡  
res = [ scaninc (+) 0 [1,3],      scaninc (+) 0 [2,4,6] ]  
≡  
res = [ [ 1, 4],                  [2, 6, 12] ]
```

**becomes a segmented scan**, which requires a flag array as arg:

```
sgmScaninc (+) 0 [1, 0, 1, 0, 0] [1, 3, 2, 4, 6] ≡ [ 1, 4, 2, 6, 12 ]
```

Flattening a scan directly nested inside a map:

- the flat-data array is obtained by a segmented scan;
- the shape of the result array is the same as the input array.

$\mathcal{F}(\text{res} = \text{map } (\backslash \text{row} \rightarrow \text{scan } (\odot) 0_{\odot} \text{ row}) \text{ arr}) \Rightarrow$   
 $S_{\text{res}}^1 = S_{\text{arr}}^1$   
 $D_{\text{res}} = \text{sgmScan } (\odot) 0_{\odot} F_{\text{arr}} D_{\text{arr}}$

# Nested vs Flattened Parallelism: Map inside a Map

## (2) Map nested inside a map:

```
res = map (\row->map f row) [[1,3], [2,4,6]]
```

≡

```
res = [ map f [1, 3],      map f [2, 4, 6] ]
```

≡

```
res = [ [f(1),f(3)], [f(2),f(4),f(6)] ]
```



# Nested vs Flattened Parallelism: Map inside a Map

## (2) Map nested inside a map:

```
res = map (\row->map f row) [[1,3], [2,4,6]]  
≡  
res = [ map f [1, 3],      map f [2, 4, 6] ]  
≡  
res = [ [f(1),f(3)], [f(2),f(4),f(6)] ]
```

## Flattening a map directly nested inside a map:

- the flat-data array is obtained by a map on the flat input;
- the shape of the result array is the same as the input array.

$\mathcal{F}(\text{res} = \text{map } (\lambda \text{row} \rightarrow \text{map } f \text{ row}) \text{ arr}) \Rightarrow$

$$S_{res}^1 = S_{arr}^1$$

$$D_{res} = \text{map } f \ D_{arr}$$

# Nested vs Flattened Parallelism: Replicate in a Map

## (3) Replicate nested inside a map:

```
res = map2 (\ n m -> replicate n m) [1,0,3,2] [7,3,8,9] ≡  
res = [ replicate 1 7, replicate 0 3, replicate 3 8, replicate 2 9 ] ≡  
res = [ [7], [], [8,8,8], [9,9] ]
```

# Nested vs Flattened Parallelism: Replicate in a Map

## (3) Replicate nested inside a map:

```
res = map2 (\ n m -> replicate n m) [1,0,3,2] [7,3,8,9] ≡  
res = [ replicate 1 7, replicate 0 3, replicate 3 8, replicate 2 9 ] ≡  
res = [ [7], [], [8,8,8], [9,9] ]
```

```
res = map2(\n m-> replicate n m) ns ms
```

**becomes a scan-scatter composition:**

# Nested vs Flattened Parallelism: Replicate in a Map

## (3) Replicate nested inside a map:

```
res = map2 (\ n m -> replicate n m) [1,0,3,2] [7,3,8,9] ≡  
res = [ replicate 1 7, replicate 0 3, replicate 3 8, replicate 2 9 ] ≡  
res = [ [7], [], [8,8,8], [9,9] ]
```

```
res = map2(\n m-> replicate n m) ns ms
```

**becomes a scan-scatter composition:**

1. the shape of the result array is ns
- 2-3. builds the indices at which segment start (-1 for null shape)
4. get the size of the flat array (summing ns)
- 5-6. write the ms and ns values at the start of their segments
7. propagate the ms values throughout their segments.

■ **Implementation shortcomings:**

# Nested vs Flattened Parallelism: Replicate in a Map

## (3) Replicate nested inside a map:

```
res = map2 (\ n m -> replicate n m) [1,0,3,2] [7,3,8,9] ≡  
res = [ replicate 1 7, replicate 0 3, replicate 3 8, replicate 2 9 ] ≡  
res = [ [7], [], [8,8,8], [9,9] ]
```

res = map2(\n m-> replicate n m) ns ms

**becomes a scan-scatter composition:**

1. the shape of the result array is ns
  - 2-3. builds the indices at which segment start (-1 for null shape)
  4. get the size of the flat array (summing ns)
  - 5-6. write the ms and ns values at the start of their segments
  7. propagate the ms values throughout their segments.
- **Implementation shortcomings:**  $\text{sgmScan}^{inc} (+)$ ?

$\mathcal{F}(\text{res} = \text{map2} (\backslash n m \rightarrow \text{replicate } n m) \text{ ns ms}) \Rightarrow$

1.  $S_{res}^1 = \text{ns}$
2.  $\text{inds} = \text{scan}^{exc} (+) 0 \text{ ns}$
3.  $|> \text{map2} (\backslash n i \rightarrow \text{if } n > 0 \text{ then } i \text{ else } -1) \text{ ns}$
4.  $\text{size} = (\text{last inds}) + (\text{last ns})$
5.  $\text{vls} = \text{scatter} (\text{replicate size } 0) \text{ inds ms}$
6.  $F_{arr} = \text{scatter} (\text{replicate size } 0) \text{ inds ns}$
6.  $D_{res} = \text{sgmScan}^{inc} (+) 0 F_{res} \text{ vls}$

—  $ms = [7, 3, 8, 9]$   
—  $ns = [1, 0, 3, 2]$   
—  $[0, 1, 1, 4]$   
—  $\text{inds} = [0, -1, 1, 4]$   
—  $4 + 2 = 6$   
—  $[7, 8, 0, 0, 9, 0]$   
—  $[1, 3, 0, 0, 2, 0]$   
—  $[7, 8, 8, 8, 9, 9]$

## Nested vs Flattened Parallelism: Iota in a Map

**(4) Iota nested inside a map**  $((\text{iota } n) \equiv [0, \dots, n-1])$ :

```
res = map (\i -> iota i) [1,3,2]  $\equiv$ 
```

```
res = [iota 1, iota 3, iota 2]  $\equiv$  [ [0], [0,1,2], [0,1] ]
```

# Nested vs Flattened Parallelism: Iota in a Map

**(4) Iota nested inside a map** ( $(\text{iota } n) \equiv [0, \dots, n-1]$ ):

```
res = map (\i -> iota i) [1,3,2]  $\equiv$   
res = [iota 1, iota 3, iota 2]  $\equiv$  [ [0], [0,1,2], [0,1] ]
```

**boils down to a segmented scan applied to an array of ones:**

1. by definition of `iota`, `ns` contains the size of each subarray, hence the shape of the result is `ns`;
2. the flag-array of the result,  $F_{res}$ , is constructed from `ns`;
3. the result is obtained by an exclusive segmented scan operation applied to an array of ones.

$\mathcal{F}(\text{res} = \text{map } (\backslash n \rightarrow \text{iota } n) \text{ ns}) \Rightarrow$

1.  $S^1_{res} = ns$   $\text{--- } ns = [1, 3, 2]$
2.  $F_{res} = \text{mkFlagArray } ns \ 0 \ ns$   $\text{--- } F_{res} = [1, 1, 0, 0, 1, 0]$
3.  $D_{res} = \text{sgmScan}^{exc} (+) \ 0 \ F_{res} \ (\text{replicate } \text{flen}_{res} \ 1)$   $\text{--- } [0, 0, 1, 2, 0, 1]$

Note that  $\text{iota } n \equiv \text{scan}^{exc} (+) \ 0 \ (\text{replicate } n \ 1)$ .

# Nested vs Flattened Parallelism: Reduce Inside Map

## (5) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in  
let res = map (\x -> reduce (+) 0 x) arr  
-- should result in [8, 13]
```



# Nested vs Flattened Parallelism: Reduce Inside Map

## (5) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in
let res = map (\x -> reduce (+) 0 x) arr
-- should result in [8, 13]
```

translates to a **scan-pack** composition:

1. the length of `res` equals the number of subarrays of `arr`;
2. the shape of `arr` is scanned: the result records the position of the last element in a segment plus one;
3. segmented scan is applied on the input array: the last element in a segment holds the reduced value of the segment;
4. segment's last element is extracted by a map operation.

$\mathcal{F}(\text{res} = \text{map } (\backslash \text{row} \rightarrow \text{reduce } \odot 0 \odot \text{row}) \text{ arr}) \Rightarrow$

—  $S_{arr}^0 = [2]$ ,  $S_{arr}^1 = [3, 2]$ ,  $F_{arr} = [1, 0, 0, 1, 0]$ ,  $D_{arr} = [1, 3, 4, 6, 7]$

1.  $S_{res}^0 = S_{arr}^0$  —  $S_{res}^0 = [2]$
2.  $\text{indsp1} = \text{scan } (+) 0 S_{arr}^1$  —  $\text{indsp1} = [3, 5]$
3.  $\text{tmp} = \text{sgmScan } (\odot) 0 \odot F_{arr} D_{arr}$  —  $\text{tmp} = [1, 4, 8, 6, 13]$
4.  $D_{res} = \text{map2}(\backslash s \text{ ip1} \rightarrow \text{if } s \leq 0 \text{ then } 0 \odot \text{else tmp[ip1 - 1]]) S_{arr}^1 \text{ indsp1}$  —  $D_{res} = [8, 13]$

# Nested vs Flattened Parallelism: Reduce Inside Map

## (5) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in
let res = map (\x -> reduce (+) 0 x) arr
-- should result in [8, 13]
```

# Nested vs Flattened Parallelism: Reduce Inside Map

## (5) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in  
let res = map (\x -> reduce (+) 0 x) arr  
-- should result in [8, 13]
```

**We can also “cheat” and use a histogram-like computation**

# Nested vs Flattened Parallelism: Reduce Inside Map

## (5) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in
let res = map (\x -> reduce (+) 0 x) arr
-- should result in [8, 13]
```

We can also “cheat” and use a histogram-like computation

$\mathcal{F}(\text{res} = \text{map } (\backslash \text{row} \rightarrow \text{reduce } \odot 0 \odot \text{row}) \text{ arr}) \Rightarrow$

—  $S_{arr}^0 = [2]$ ,  $S_{arr}^1 = [3, 2]$ ,  $F_{arr} = [1, 0, 0, 1, 0]$ ,  $D_{arr} = [1, 3, 4, 6, 7]$

1.  $S_{res}^0 = S_{arr}^0$  —  $S_{res}^0 = [2]$

2.  $D_{res} = \text{hist } (\odot) 0 \odot (S_{arr}^0[0]) \parallel_{arr}^1 D_{arr}$

How else can I try to optimize this code by hand?

# Nested vs Flattened Parallelism: Reduce Inside Map

## (5) Reduce Inside a Map or Segmented Reduce:

```
let arr = [[1, 3, 4], [6, 7]] in
let res = map (\x -> reduce (+) 0 x) arr
-- should result in [8, 13]
```

We can also “cheat” and use a histogram-like computation

$\mathcal{F}(\text{res} = \text{map } (\backslash \text{row} \rightarrow \text{reduce } \odot 0 \odot \text{row}) \text{ arr}) \Rightarrow$   
—  $S_{arr}^0 = [2]$ ,  $S_{arr}^1 = [3, 2]$ ,  $F_{arr} = [1, 0, 0, 1, 0]$ ,  $D_{arr} = [1, 3, 4, 6, 7]$   
1.  $S_{res}^0 = S_{arr}^0$  —  $S_{res}^0 = [2]$   
2.  $D_{res} = \text{hist } (\odot) 0 \odot (S_{arr}^0[0]) \parallel_{arr}^1 D_{arr}$

How else can I try to optimize this code by hand?

- practical performance refers to how many global-memory accesses you perform
- accessing  $II_{arr}^1$  from memory has significant cost
- in some practical cases, it might be more efficient to not manifest  $II_{arr}^1$ , but instead to compute its elements by binary searching the  $B_{arr}^1$  array.

## Flattening Scatter and Histogram

How does one flattens a scatter perfectly nested inside a map?

How does one flattens a histogram perfectly nested inside a map?

## Flattening Scatter and Histogram

How does one flattens a scatter perfectly nested inside a map?

How does one flattens a histogram perfectly nested inside a map?

You will have to answer it yourselves as part of the third weekly assignment :)

# Treating a Scalar Variant to the Outer Map

## (6) The inner construct uses a scalar variant to the outer map:

```
let res = map2 (\x ys -> map (+x) ys) [1,3] [[4,5,6], [9,7]] ≡  
let res = [map (+1) [4,5,6], map (+3) [9,7]]  
let res = [ [5,6,7], [12,10] ]
```



# Treating a Scalar Variant to the Outer Map

## (6) The inner construct uses a scalar variant to the outer map:

```
let res = map2 (\x ys -> map (+x) ys) [1,3] [[4,5,6], [9,7]] ≡  
let res = [map (+1) [4,5,6], map (+3) [9,7]]  
let res = [ [5,6,7], [12,10] ]
```

Traditionally, this is handled by expanding (replicating) each x across the whole segment

```
let Dxss = [1, 1, 1, 3, 3]  
let res = map2 (+) [1, 1, 1, 3, 3 ]  
                  [4, 5, 6, 9, 7 ]  
                  = = = = =  
                  [5, 6, 7, 12, 10]
```

Instead, we use  $\Pi_{arr}^1$  to indirectly access in the xs array:

```
 $\mathcal{F}(\text{res} = \text{map2 } (\lambda x \text{ ys} \rightarrow \text{map } (f \ x) \text{ ys}) \text{ xs yss}) \Rightarrow$   
—  $xs = [1, 3], S_{yss}^1 = [3, 2], F_{yss} = [1, 0, 0, 1, 0], D_{yss} = [4, 5, 6, 9, 7]$   
1.  $S_{res}^1 = S_{yss}^1$   
2.  $D_{res} = \text{map2 } (\lambda y \text{ sgmind} \rightarrow f \text{ xs[sgmind] } y) \ D_{yss} \ \Pi_{yss}^1$   
—  $\Pi_{yss}^1 = [0, 0, 0, 1, 1], D_{res} = [5, 6, 7, 12, 10]$ 
```

# Treating Indexing Variant to the Outer Map

## (7) Indexing Operations Variant to the Outer Map:

```
let res = map2 (\i xs -> xs[i]) [2,0] [[4,5,6], [9,7]] ≡  
let res = [ 6, 9 ]
```

# Treating Indexing Variant to the Outer Map

## (7) Indexing Operations Variant to the Outer Map:

```
let res = map2 (\i xs -> xs[i]) [2,0] [[4,5,6], [9,7]] ≡  
let res = [ 6, 9 ]
```

To corresponding flat index in  $D_{y_{ss}}$  is obtained by summing up

- the start offset of every segment, which we get from  $B_{y_{ss}}^1$ , and
- the index inside the segment, which we get from  $i_s$

$\mathcal{F}(\text{res} = \text{map2 } (\lambda i \text{ xs} \rightarrow \text{xs}[i]) \text{ is } x_{ss}) \Rightarrow$

—  $i_s = [2, 0]$ ,  $S_{x_{ss}}^1 = [3, 2]$ ,  $B_{x_{ss}}^1 = [0, 3]$ ,  $D_{x_{ss}} = [4, 5, 6, 9, 7]$

1.  $S_{res}^0 = S_{i_s}^0$  — =  $S_{i_s}^0 = [2]$

2.  $D_{res} = \text{map2 } (\lambda \text{ off } i \rightarrow D_{x_{ss}}[\text{off} + i]) B_{x_{ss}}^1 \text{ is } — D_{res} = [6, 9]$

# Nested vs Flattened Parallelism: If Inside a Map 2D Case

## (8) If-Then-Else with inner parallelism nested inside a map:

```
bs  = [F,T,F,T]
xss = [[1,2,3],[4,5,6,7],[8,9],[10]]
res = map(\b xs -> if b then map (+1) xs else map (*2) xs) bs xss
res = [ map(*2)[1,2,3], map(+1)[4,5,6,7], map(*2)[8,9], map(+1)[10] ]
res = [ [2,4,6], [5,6,7,8], [16,18], [11] ]
```

# Nested vs Flattened Parallelism: If Inside a Map 2D Case

## (8) If-Then-Else with inner parallelism nested inside a map:

```
bs = [F,T,F,T]
xss = [[1,2,3],[4,5,6,7],[8,9],[10]]
res = map(\b xs -> if b then map (+1) xs else map (*2) xs) bs xss
res = [ map(*2)[1,2,3], map(+1)[4,5,6,7], map(*2)[8,9], map(+1)[10] ]
res = [ [2,4,6], [5,6,7,8], [16,18], [11] ]
```

translates to a **scatter-map-gather** composition. Intuition:

1. compute `iinds`, the permutation of segments w.r.t. `bs`;
  - 2-3. partition the `xss` array based on `bs`;
  - 4-5. **flatten outer map and/on top of the parallel code of the then and else branches**;
  6. inverse permute the resulted segments according to `iinds`.
- ```
1. iinds = partition2 (\i -> bs[i]) (iota (length b)) -- [1,3,0,2]
2. xssthen = gatherThen iinds xss -- ([4,1], [4,5,6,7, 10])
3. xsselse = gatherElse iinds xss -- ([3,2], [1,2,3, 8,9])
-- Recursively Flatten the Then and Else Branches!
4. resthen = F(map (map (+1)) xssthen) -- ([4,1], [5,6,7,8, 11])
5. reselse = F(map (map (*2)) xsselse) -- ([3,2], [2,4,6,16,18])
6. res = inversePermute iinds (resthen++reselse)
-- ([3,4,2,1], [2,4,6, 5,6,7,8, 16,18, 11])
```

# Nested vs Flattened Parallelism: If Inside a Map 2D Case

## (8) If-Then-Else with inner parallelism nested inside a map:

bs = [F,T,F,T], xss = [[1,2,3],[4,5,6,7],[8,9],[10]], S<sub>xss</sub><sup>1</sup>=[3,4,2,1], f=map (+1), g=map (\*2)

```
F(res = map2 (\b xs -> if b then f xs else g xs) bs xss) =>
(spl, iinds) = partition2 bs (iota (length bs)) — (2, [1,3,0,2])
(Sxssthen1, Sxsselse1) = split spl (map (\ii -> Sxss1[ii]) iinds) — ([4,1],[3,2])
maskxss = map (\sgmind -> bs[sgmind]) ||xss1 — [F,F,F,T,T,T,T,F,F,T]
(brk, Dxssp) = partition2 maskxss Dxss
(Dxssthenp, Dxsselsep) = split brk Dxssp — ([4,5,6,7,10],[1,2,3,8,9])
(Sresthen1, Dresthen) = F(map f) (Sxssthen1, Dxssthenp) — ([4,1], [5,6,7,8,11])
(Sreselse1, Dreselse) = F(map g) (Sxsselse1, Dxsselsep) — ([3,2], [2,4,6,16,18])
Sres1P = Sresthen1 ++ Sreselse1 — [4,1,3,2]
Sres1 = scatter (replicate (length bs) 0) iinds Sres1P — [3,4,2,1]
Bres1 = scanexc (+) 0 Sres1 — [0,3,7,9]
Fresp = mkFlagArray Sres1P 0 (map (+1) iinds) — [2,0,0,0,4,1,0,0,3,0]
||res1P = sgmscan (+) 0 Fresp Fresp |> map (\x -> x-1) — [1,1,1,1,3,0,0,0,2,2]
||res2P = ||resthen2 ++ ||reselse2 — [0,1,2,3,0, 0,1,2,0,1]
sindsres = map2 (\sgm iin -> Bres1[sgm] + iin) ||res1P ||res2P
— [3+0,3+1,3+2,3+3, 9+0, 0+0,0+1,0+2, 7+0,7+1]=[3,4,5,6,9,0,1,2,7,8]
Dres = scatter (replicate flenres 0) sindsres (Dresthenp ++ Dreselsep)
— [2,4,6, 5,6,7,8, 16,18, 11]
(Sres1, Dres)
```

# Nested vs Flattened Parallelism: Do Loop Inside a Map

## (9) Flattening a Do Loop Nested Inside a Map:

- compute the maximal loop count  $n_{max}$
- interchange the loop and the map:
  - ▶ loop count becomes  $n_{max}$
  - ▶ the loop body is wrapped inside a `if i < n` condition, and
  - ▶ the new loop body is flattened!

$\mathcal{F}(\text{res} = \text{map2} (\backslash n \text{ xs} \rightarrow \text{loop} (\text{xs}) \text{ for } i < n \text{ do } f \text{ xs}) \text{ ns } \text{xss}) \Rightarrow$

1.  $n_{max} = \text{reduce max } 0 \text{ i } 32 \text{ ns}$
2.  $g \text{ i } m \text{ arr} = \text{if } i < m \text{ then } f \text{ arr} \text{ else arr}$
3.  $\text{loop}(S_{xss}^1, D_{xss}) \text{ for } i < n_{max} \text{ do}$
4.  $\mathcal{F}(\text{map2 } (g \text{ i})) \text{ ns } (S_{xss}^1, D_{xss})$
5.  $\text{--- } (g \text{ i})^L \text{ ns } (S_{xss}^1, D_{xss})$

But this treatment does not necessarily preserve the work asymptotic ... what to do?

# Nested vs Flattened Parallelism: Do Loop Inside a Map

## (9) Flattening a Do Loop Nested Inside a Map:

- compute the maximal loop count  $n_{max}$
- interchange the loop and the map:
  - ▶ loop count becomes  $n_{max}$
  - ▶ the loop body is wrapped inside a `if i < n` condition, and
  - ▶ the new loop body is flattened!

$\mathcal{F}(\text{res} = \text{map2} (\backslash n \text{ xs} \rightarrow \text{loop}(\text{xs}) \text{ for } i < n \text{ do } f \text{ xs}) \text{ ns } \text{xss}) \Rightarrow$

1.  $n_{max} = \text{reduce max } 0i32 \text{ ns}$
2.  $g \ i \ m \ arr = \text{if } i < m \text{ then } f \ arr \text{ else } arr$
3. **loop**( $S_{xss}^1$ ,  $D_{xss}$ ) **for**  $i < n_{max}$  **do**
4.      $\mathcal{F}(\text{map2} (g \ i)) \text{ ns } (S_{xss}^1, D_{xss})$
5.     —  $(g \ i)^L \text{ ns } (S_{xss}^1, D_{xss})$

But this treatment does not necessarily preserve the work asymptotic ... what to do?

If the size of the result can be deduced/inferred:

- allocate the flat-result array before the loop
- filter out the empty segments, and make the loop iterate until the shape is empty
- each time a segment finish execution (1) it is scattered into the result, and (2) it is filtered out from the running set of segments.



## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

Flattening Recipe

Irregular Multi-Dimensional Array Representation

Flattening Recipe (Redundant)

Rules For Flattening

**Flattening by Function Lifting**

Flattening Quicksort

Flattening Prime-Number (Sieve) Computation

## Flattening by Function Lifting: Basic Idea

Assume a simple function  $f$ :

```
let f (x: i32) : i32 = x + 1
```

$f$  lifted, denoted  $f^L$  semantically corresponds to `map f`, where the arguments have been expanded to an extra array dimension, and the inner operators/functions have also been lifted:

```
let +L [n] (as: [n]i32) (bs: [n]i32) : [n]i32 =  
  map2 (+) as bs
```

```
let fL [n] (xs: [n]i32) : [n]i32 =  
  xs +L (replicate n 1)
```

# Flattening by Function Lifting: Basic Idea

Assume a simple function  $f$ :

```
let f (x: i32) : i32 = x + 1
```

$f$  lifted, denoted  $f^L$  semantically corresponds to `map f`, where the arguments have been expanded to an extra array dimension, and the inner operators/functions have also been lifted:

```
let +L [n] (as: [n]i32) (bs: [n]i32) : [n]i32 =  
  map2 (+) as bs
```

```
let fL [n] (xs: [n]i32) : [n]i32 =  
  xs +L (replicate n 1)
```

- Locals such as  $x \Rightarrow$  left alone
- Global such as  $+$   $\Rightarrow$  lifted ( $+^L$ )
- Constants such as  $k \Rightarrow \text{replicate (length xs) } k$ 
  - ▶ good for vectorization, bad for locality, asymptotics
  - ▶ for GPU better to indirectly index into a smaller array, rather than `replicate`.

## Flattening by Function Lifting: Key Insight!

```
let f (xs: []i32) : []i32 = map g xs  -- =  $g^L$  xs  
let fL (xss: [][]i32) : [][]i32 = (gL)L  -- ???
```

How do we stop lifting?  $g$  and  $g^L$  are enough: no need for  $(g^L)^L$ !

## Flattening by Function Lifting: Key Insight!

```
let f (xs: [i32]) : [i32] = map g xs -- =  $g^L$  xs  
let  $f^L$  (xss: [[i32]]) : [[i32]] = ( $g^L$ )L -- ???
```

How do we stop lifting?  $g$  and  $g^L$  are enough: no need for  $(g^L)^L$ !

```
let f (xs: [i32]) : [i32] = map g xs -- =  $g^L$  xs  
-- in nested parallel form
```

```
let  $f^L$  (xss: [[i32]]) : [[i32]] =  
    segment xss ( $g^L$  (concat xss))  
-- in flatten form
```

```
let  $f^L$  ( $S_{xss}^1$ : [i32],  $D_{xss}$ : [i32]) : ([i32], [i32]) =  
    ( $S_{xss}^1$ ,  $g^L$   $D_{xss}$ )
```

In Haskell Notation:

```
concat  :: [[a]] -> [a]  
segment :: [[a]] -> [b] -> [[b]]  
          shape    flat data    nested data
```

## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

Flattening Recipe

Irregular Multi-Dimensional Array Representation

Flattening Recipe (Redundant)

Rules For Flattening

Flattening by Function Lifting

**Flattening Quicksort**

Flattening Prime-Number (Sieve) Computation

# Recounting Quicksort

Recount the classic nested-parallel definition:

```
let quicksort [n] (arr : [n]f32) : [n]f32 =  
  if n < 2 then arr else  
    let i = getRand (0, (length arr) - 1)  
    let a = arr[i]  
    let s1 = filter (< a) arr  
    let s2 = filter (== a) arr  
    let s3 = filter (> a) arr  
    in (quicksort s1) ++ s2 ++ (quicksort s3)  
— can be re-written as:  
— rs = map nestedQuicksort [s1, s3]  
— in (rs[0]) ++ s2 ++ (rs[1])
```

Note: Futhark does not support recursive calls, hence not valid code!

# Nested-Parallel Quicksort Simplified

For simplicity we will rewrite it in terms of `partition2`:

```
let isSorted [n] (as: [n]f32) : bool =  
  map (\i -> if i==0 then true else as[i-1] < as[i]) (iota n)  
  |> reduce (&&) true  
  
let quicksort [n] (arr: [n]f32) : [n]f32 =  
  if isSorted arr then arr else  
  let i = getRand (0, (length arr) - 1)  
  let a = arr[i]  
  let bs = map (< a) arr  
  let (q, arr') = partition2 bs 0.0f32 arr  
  let (arr<, arr≥) = split q arr'  
  in concat <| map quicksort [arr<, arr≥]
```

Note: Futhark does not support recursive calls, irregular map operation, or concat!



## Partition2

Reorders the elements of an array such that those that correspond to a true mask come before those corresponding to false.

```
let partition2 [n] 't (conds: [n]bool) (dummy: t) (arr: [n]t)
  : (i32, [n]t) =
  let tflgs = map (\ c => if c then 1 else 0) conds
  let fflgs = map (\ b => 1 - b) tflgs

  let indsT = scan (+) 0 tflgs
  let tmp    = scan (+) 0 fflgs
  let lst    = if n > 0 then indsT[n-1] else -1
  let indsF = map (+lst) tmp

  let inds   = map3 (\ c indT indF => if c then indT-1 else indF-1)
                  conds indsT indsF

  let fltarr = scatter (replicate n dummy) inds arr
  in (lst, fltarr)
```

For example:

```
conds = [F,T,F,T,F,F,T]
xss   = [1,2,3,4,5,6,7]
partition2 conds 0 xss => (3, [2,4,7,1,3,5,6])
```

# Lifting Quicksort

**Key Idea: write a function with the semantics of**

`map nestedQuicksort`, i.e., it operates on array of arrays.

```
let isSorted [n] (as: [n]f32) : bool =  
  map (\i -> if i==0 then true else as[i-1] < as[i]) (iota n)  
  |> reduce (&&) true  
  
let quicksortL (xss: [][]f32) : [][]f32 =  
  map (\xs ->  
    if isSorted xs then xs else — oh, boy, not if-then-else!  
    let i = getRand (0, (length xs) - 1)  
    let a = xs[i]  
    let bs = map (< a) xs  
    let (q, xsp) = partition2 bs 0.0f32 xs  
    let (xs<, xs≥) = split q xsp  
    in concat <| map quicksort [xs<, xs≥]  
  ) xss
```

**Important observations:**

- `map quicksort`  $\equiv$  `quicksortL`
- `map(map quicksort)`  $\equiv$  `quicksortL`  $\equiv$  `segment o quicksortL o concat`
- the flat data of `[xs<, xs≥]`  $\equiv$  `xsp`, the result of `partition2`

# Lifting Quicksort

Let us treat the last three lines from the previous implem.:

```
let quicksortL (Sxss1:[ ] i32 , Dxss:[ ] f32): ([ ] i32 , [ ] f32) = — (xss: [ ] [ ] f32)
  if isSorted Dxss then (Sxss1:[ ] i32 , Dxss:[ ] f32) else — big cheat!
  let (Sbss1 , Dbss) =  $\mathcal{F}$  (
    map (\ xs →
      let i = getRand (0 , (length xs) - 1)
      let a = xs[i]
      let bs = map (< a) xs
      in bs
    ) xss
  )
  let (ps , (Sxssp1 , Dxssp)) = partition2L Dbss 0.0f32 (Sxss1 , Dxss)
  — Invariant: Sxssp1 == Sbss1 == Sxss1
  let S[xss<,xss≥]1 = filter (≠0) <| flatten <|
    map2 (λ p s → if s==0 then [0,0] else [p,s-p]) ps Sxss1
  in quicksortL (S[xss<,xss≥]1 , Dxssp)
```

- S<sub>[xss<a,xss≥a]</sub><sup>1</sup> is the shape of [xs< , xs≥]
- (concat <| quicksort<sup>L</sup>)<sup>L</sup> xsss ≡ concat <| segment xsss <| quicksort<sup>L</sup> (concat xsss) ≡ quicksort<sup>L</sup> (concat xsss)
- The function looks tail recursive now: let's replace it with a loop!

# Lifting Quicksort: Final Implementation

```
let quicksortL [m][n] (SXSS1:[m]i32, DXSS:[n]f32): [n]f32 =  
  let (stop, count) = (isSorted DXSS, 0i32)  
  let (_, res, _, _) =  
    loop (SXSS1, DXSS, stop, count) while (!stop) do  
      — compute helper-representation structures  
      let BXSS1 = scanexc (+) 0 SXSS1  
      let FXSS1 = mkFlagArray SXSS1 0i32 <| map (+1) <| iota m  
      let IIXSS1 = sgmscan (+) 0 FXSS1 <|  
        map (\f → if f==0 then 0 else f-1) FXSS1  
      — flattening quicksort:  
      let rL = map (\u → randomInd (0,u-1) count) SXSS1  
      let aL = map3 (\r l i → if l <= 0 then 0.0 else DXSS[BXSS1[i]+r]  
        ) rL SXSS1 (iota m)  
      let Dbss = map2 (\x sgmind → aL[sgmind] > x ) DXSS IIXSS1  
      let (ps, (SXSSp1, DXSSper)) = partition2L Dbss 0.0f32 (SXSS1, DXSS)  
      let S[XSS<,XSS≥]1 = filter (!=0) <| flatten <|  
        map2 (\ p s → if s==0 then [0,0] else [p,s-p]) ps SXSS1  
      in (S[XSS<,XSS≥]1, DXSSper, isSorted DXSSper, count+1)  
  in res
```

PFP Weekly 2 Exercise: Implement partition2<sup>L</sup>

## Parallel Basic Blocks

### Flattening Nested and Irregular Parallelism

Flattening Recipe

Irregular Multi-Dimensional Array Representation

Flattening Recipe (Redundant)

Rules For Flattening

Flattening by Function Lifting

Flattening Quicksort

Flattening Prime-Number (Sieve) Computation

# How Does One Flatten Prime Numbers?

## The important bit with nested parallelism:

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p -> let m = (n `div` p)
                  in map (\j -> j*p) [2..m]
                ) sqrt_primes
not_primes = reduce (++) [] nested
```

# How Does One Flatten Prime Numbers?

## The important bit with nested parallelism:

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p -> let m = (n `div` p)
                    in map (\j -> j*p) [2..m]
                    ) sqrt_primes
not_primes = reduce (++) [] nested
```

## Normalize the nested map:

```
sqrt_primes = primesOpt (sqrt (fromIntegral n))
nested = map (\p ->
    let m    = n `div` p      in      -- distribute map
    let mm1  = m - 1         in      -- distribute map
    let iot  = iota mm1      in      --  $\mathcal{F}$  rule 4
    let twom = map (+2) iot   in      --  $\mathcal{F}$  rule 2
    let rp   = replicate mm1 p in      --  $\mathcal{F}$  rule 3
    in map (\(j,p) -> j*p) (zip twom rp) --  $\mathcal{F}$  rule 2
    ) sqrt_primes
not_primes = reduce (++) [] nested      -- ignore, already flat
```

Flattening PrimeOpt was part of PMPH's Weekly Assignment 2!