

# Flattening Irregular Nested Parallelism

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

Department of Computer Science (DIKU)  
University of Copenhagen

December 2024 DPP Lecture Slides

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

- What is "Flattening"? Recipe for Applying Flattening
- Several Re-Write Rules (inefficient for replicate & iota)
- Jagged (Irregular Multi-Dim) Array Representation
- Revisiting the Rewrites for Replicate & Iota Nested Inside Map
- Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

- Several Applications of Flattening
- More Flattening Rules
- Flattening by Function Lifting
- Flattening Quicksort
- Flattening Prime-Number (Sieve) Computation

# Zip, Unzip, iota, replicate

- $\text{zip} : [n]_{\alpha_1} \rightarrow [n]_{\alpha_2} \rightarrow [n](\alpha_1, \alpha_2)$
- $\text{zip } [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [(a_1, b_1), \dots, (a_n, b_n)],$
- $\text{unzip} : [n](\alpha_1, \alpha_2) \rightarrow ([n]_{\alpha_1}, [n]_{\alpha_2})$
- $\text{unzip } [(a_1, b_1), \dots, (a_n, b_n)] \equiv ([a_1, \dots, a_n], [b_1, \dots, b_n]),$
- 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.,  $e : \alpha, \quad x_i : \alpha, \forall i$ , 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.,  $e : \alpha, \quad x_i : \alpha, \forall i$ , 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.,  $e : \alpha, \quad x_i : \alpha, \forall i$ , and  $\odot : \alpha \rightarrow \alpha \rightarrow \alpha$ .

## Map2, Filter

- $\text{map2} : (\alpha_1 \rightarrow \alpha_2 \rightarrow \beta) \rightarrow [n]\alpha_1 \rightarrow [n]\alpha_2 \rightarrow [n]\beta$
- $\text{map2 } \odot [a_1, \dots, a_n] [b_1, \dots, b_n] \equiv [a_1 \odot b_1, \dots, a_n \odot b_n]$
- $\text{map3 } \dots$
- $\text{filter} : (\alpha \rightarrow \text{Bool}) \rightarrow [n]\alpha \rightarrow [m]\alpha \ (m \leq n)$
- $\text{filter } p [a_1, \dots, a_n] = [a_{k_1}, \dots, a_{k_m}]$  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]

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

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

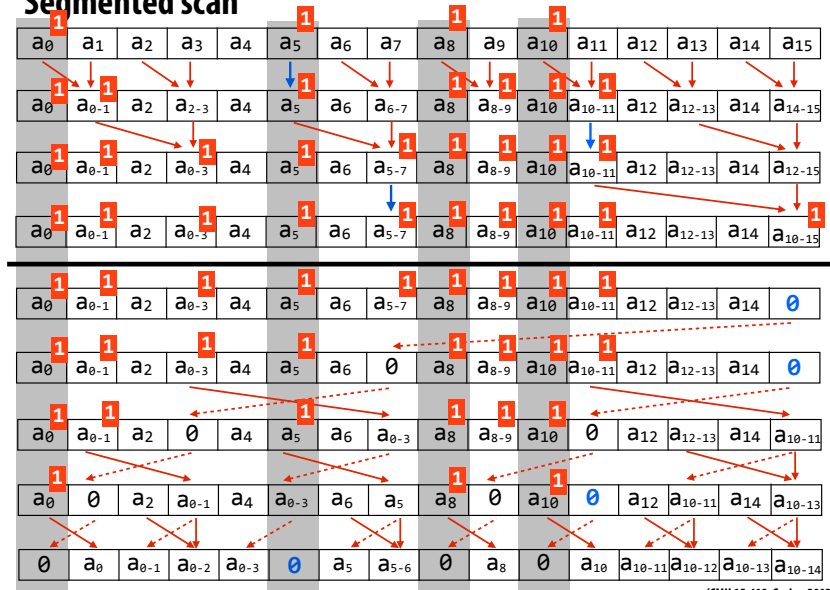
In result, the elements satisfying the predicate occur before the others. Can be implemented by means of `map`, `scan`, `scatter`.

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

Assume  $X = [5, 4, 2, 3, 7, 8]$ , and  
cond is  $T(\text{rue})$  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

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

**Correctness Argument:**

## Segmented Scan Is a Sort of Scan

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

### Correctness Argument:

verify sequential semantics + associative operator  $\Rightarrow$   
parallel semantics also holds

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

- What is "Flattening"? Recipe for Applying Flattening
- Several Re-Write Rules (inefficient for replicate & iota)
- Jagged (Irregular Multi-Dim) Array Representation
- Revisiting the Rewrites for Replicate & Iota Nested Inside Map
- Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

- Several Applications of Flattening
- More Flattening Rules
- Flattening by Function Lifting
- Flattening Quicksort
- Flattening Prime-Number (Sieve) Computation

# What is "Flattening"?

A code transformation, attributed to Blelloch in the context of the NESL languages, that takes as input a nested parallel program—possibly involving recursion and irregular/jagged arrays—and produces a semantically-equivalent, flat-parallel programs that runs optimally on a PRAM machine.

**Meaning: *it is guaranteed to preserve the work and depth of the original nested-parallel program.***<sup>\*\*</sup>

<sup>\*\*</sup> As long as scan has  $O(1)$  depth and concat has  $O(1)$  work and . . .

# Flattening Pros and Cons

## Pros:

- + clever code transformation
- + important as a programming technique as well (promotes parallel **thinking**)
- + perhaps the only way of mapping a set of challenging problems to capricious architectures such as GPUs (e.g., that do not supports dynamic scheduling of parallelism)

# Flattening Pros and Cons

## Pros:

- + clever code transformation
- + important as a programming technique as well (promotes parallel **thinking**)
- + perhaps the only way of mapping a set of challenging problems to capricious architectures such as GPUs (e.g., that do not supports dynamic scheduling of parallelism)

## Cons:

- does not consider communication/locality and hardware gets more and more heterogeneous
- worse, it tends to destroy the available locality and may explode memory footprint
- useful to cover datasets that fall outside the “common case”

**Demonstration at the end of the second Flattening lecture**



# Flattening: A Bird's Eye View

Incomplete recipe for flattening a nested-parallel program consisting of maps and scan/reduce/scatters at innermost level:

- I. **Normalize the program (think 3-address form).**  
The easy way is to replicate free variables appearing in the current map if they are variant in an outer, enclosing map.
- II. **Distribute the parallel context (perfect nest of maps) across the enclosed let-binding statements and handle recurrences by function lifting or map-loop interchange.** Systematic application results in a smallish number of code patterns.
- III. **Apply a set of rewrite rules to flatten each pattern**, e.g., treating the cases of a reduce, scan, replicate, iota, scatter, array index which is perfectly nested inside the context.

# Flattening: A Bird's Eye View

Incomplete recipe for flattening a nested-parallel program consisting of maps and scan/reduce/scatters at innermost level:

- I. **Normalize the program (think 3-address form).**  
The easy way is to replicate free variables appearing in the current map if they are variant in an outer, enclosing map.
- II. **Distribute the parallel context (perfect nest of maps) across the enclosed let-binding statements and handle recurrences by function lifting or map-loop interchange.** Systematic application results in a smallish number of code patterns.
- III. **Apply a set of rewrite rules to flatten each pattern**, e.g., treating the cases of a reduce, scan, replicate, iota, scatter, array index which is perfectly nested inside the context.

**Differences w.r.t. the PMPH material:**

- 1 **also optimize the number of accesses to global memory**  
flattening results in memory-bound performance behavior.
- 2 **cover more rewrite rules and more challenging problems**  
(e.g., divide-and-conquer recursion)

# PMPH Recap: A Simple Demonstration of How to Flatten

## Contrived Example:

```
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]]
```

# PMPH Recap: A Simple Demonstration of How to Flatten

## Contrived Example:

```
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]]
```

## I. 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
```

# PMPH Recap: A Simple Demonstration of How to Flatten

## Contrived Example:

```
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]]
```

## I. 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
```

## II. Distribute the map across every statement in the body

and adjust the inputs accordingly ( $\mathcal{F}$  denotes the transformation)

```
 $\mathcal{F}(\text{map } (\lambda i \rightarrow \text{map } (+(i+1)) \text{ (iota } i)) \text{ arr}) \equiv$ 
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=  $\mathcal{F}(\text{map2 } (\lambda i \text{ ip1} \rightarrow \text{replicate } i \text{ ip1})) \text{ arr ip1s}$ 
4. in  $\mathcal{F}(\text{map2 } (\lambda \text{ ip1r iot} \rightarrow \text{map2 } (+) \text{ ip1r iot}) \text{ ip1rs iots})$ 
```

**For simplicity we assume `arr` contains strictly-positive integers.**

# PMPH Recap: A Simple Demonstration of How to Flatten

## According to inefficient rule “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)  -- [1, 2, 0, 3, 0, 0, 4, 0, 0, 0]
      inds arr
tmp  = replicate size 1
iots = sgmScanexc (+) 0 flag tmp   -- [0, 0, 1, 0, 1, 2, 0, 1, 2, 3]
```

# PMPH Recap: A Simple Demonstration of How to Flatten

## According to inefficient rule “replicate nested inside a map”

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

```
3. let ip1rs=  $\mathcal{F}$ (map2 (\ i ip1 -> replicate i ip1) arr ip1s)
 $\equiv$ 
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 “map nested inside a map”

```
 $\mathcal{F}$ (map2 (\ ip1r iot -> map2 (+) ip1r iot) ip1rs iots)
 $\equiv$ 
4. result = map2 (+) 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
```

**At each step we also reason about the shape of the resulting array.**  
**The shape of** the 2D jagged arrays `iots`, `ip1rs`, `result` **is** `arr`.

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening

Several Re-Write Rules (inefficient for replicate & iota)

Jagged (Irregular Multi-Dim) Array Representation

Revisiting the Rewrites for Replicate & Iota Nested Inside Map

Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening

More Flattening Rules

Flattening by Function Lifting

Flattening Quicksort

Flattening Prime-Number (Sieve) Computation



# Nested vs Flattened Parallelism: Scan inside a Map

## (1) Scan nested inside a 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 nested inside a 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:

- $S_{arr}^1, F_{arr}, D_{arr}$  denote the shape, flag & flat data of input `arr`.
- The flat-data result is obtained by a segmented scan.
- The shape of the result array is the same as the input array.

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

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

$D_{res} = \text{sgmScan } (\odot) 0_{\odot} F_{arr} D_{arr}$

# 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.

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

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

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

# Nested vs Flattened Parallelism: Replicate inside 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 inside 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 inside 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.

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

$$1. S_{res}^1 = ns$$

2. inds = **scan**<sup>exc</sup> (+) 0 ns

```
3. |> map2 (\n i->if n>0 then i else -1) ns
```

```
4. size = (last inds) + (last ns)
```

```
5. vls = scatter (replicate size 0) inds ms
```

```
6. Fres = scatter (replicate size false) inds
               (replicate size true)
```

$$7. D_{res} = \text{sqmScan}^{inc} (+) 0 F_{res} \text{ vls}$$

```
-- ms = [7,3,8,9]
```

```
-- ns = [1,0,3,2]
```

```
-- [0,1,1,4]
```

```
-- inds = [0,-1,1,4]
```

$$-- 4 + 2 = 6$$

```
-- [7, 8, 0, 0, 9, 0]
```

$$-- [1, 1, 0, 0, 1, 0]$$

```
-- [7, 8, 8, 8, 9, 9]
```

## Nested vs Flattened Parallelism: Iota inside 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 inside 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  $\text{iota}$ ,  $ns$  contains the size of each subarray, hence the shape of the result is  $ns$ ;
- 2-3. the flag-array of the result,  $F_{res}$ , is constructed from  $ns$ ; (we will introduce function `mkFlagArray` a bit later).
4. the result is obtained by an exclusive segmented scan operation applied to an array of ones.

```
 $\mathcal{F}$ (res = map (\n -> iota n) ns)  $\Rightarrow$ 
```

1.  $S_{res}^1 = ns$  -- ns = [1, 3, 2]
2. `true`s = **replicate** (length ns) **true**
3.  $(-, F_{res}) = \text{mkFlagArray } ns \text{ false } \text{true}s$  --  $F_{res} = [1, 1, 0, 0, 1, 0]$
4.  $D_{res} = \text{sgmScan}^{exc} (+) 0 F_{res} (\text{replicate } \text{flen}_{res} 1)$  --  $[0, 0, 1, 2, 0, 1]$

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

**Note 2:** 1 and 0 denote `true` and `false`;  $\text{flen}_{res}$  is the sum of  $ns$ .

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening  
Several Re-Write Rules (inefficient for replicate & iota)

#### Jagged (Irregular Multi-Dim) Array Representation

Revisiting the Rewrites for Replicate & Iota Nested Inside Map  
Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening

More Flattening Rules

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] ]
```

⇒

```
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] ] ]
```

⇒

```
Sarr0 = [3]
```

```
Sarr1 = [0, 4, 3]
```

```
Sarr2 = [3, 1, 0, 2, 1, 0, 3]
```

```
flenarr = 10
```

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

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

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

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

# 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 true value (could also use  $\neq 0$  integrals), e.g., used for segmented scans:

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

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

- **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:  $ys$  and  $zs$  are much smaller than  $xss$ , hence reused from L1/2\$.**

# 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 exclusive scanning the corresponding shape and reindexing!

```
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]u32) (zero: t)           -- aoa_shp=[0,3,1,0,4,2,0]
  (aoa_val: [m]t) : ([m]u32, []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 0i64     -- aoa_len = 10
                else i64.u32 <|
                  shp_scn[m-1]+aoa_shp[m-1]
  let shp_ind = map2 (\shp ind ->      -- shp_ind=
                    if shp==0 then -1i64 -- [-1,0,3,-1,4,8,-1]
                    else i64.u32 ind    -- scatter
                    ) aoa_shp shp_scn   -- [0,0,0,0,0,0,0,0,0,0]
  let r = scatter (replicate aoa_len zero) -- [-1,0,3,-1,4,8,-1]
              shp_ind aoa_val           -- [ 1,1,1, 1,1,1, 1]
  in (shp_scn, r)                      -- r=[1,0,0,1,1,0,0,0,1,0]
```

**Versatile:** computes  $B^1$  and  $F^1$  of a 2D jagged array of shape `aoa_shp`, with the start-segment values taken from `aoa_val`.

**Unless you have a good reason,  $F$  should be a bool array (to reduce memory traffic).**

# 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):

```
(Barr1, Farr) = mkFlagArray Sarr1 0 (iota (length Sarr1))  
-- ([0, 3, 4, 4, 6, 7, 7], [0, 0, 0, 1, 3, 0, 4, 6, 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:

```
(-, Farr) = mkFlagArray Sarr1 0 (iota (length Sarr1))  
-- [0, 0, 0, 1, 3, 0, 4, 6, 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 these:

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

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

$$iota\ 10 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]$$

$II_{arr}^1$  and  $II_{arr}^2$  have the same length as flat `arr`, in our case 10.

How?

We can also construct it by binary searching  $B_{arr}$  or by means of a segmented scan.



# 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):

```
(Barr1, F'arr) = mkFlagArray Sarr1 0 (iota (length Sarr1))  
-- ([0, 3, 4, 4, 6, 7, 7], [0, 0, 0, 1, 3, 0, 4, 6, 0, 0])
```

```
Farr = map bool.u32 F'arr
```

```
IIarr1 = sgmScaninc (+) 0 Farr F'arr
```

```
IIarr2 = map2 (\ i sgm -> i - Barr1[sgm] ) (iota flenarr) IIarr1  
-- ^ this fuses better & performs less memory traffic than the below:
```

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

# $B^{inc}$ and $II^1$ Are the Important Ones

Because you can deduce the other arrays by means of simple maps, that fuse better and generate less traffic.

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

⇒

```
 $S_{arr}^0$  = [7]
```

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

```
flenarr = reduce (+) 0  $S_{arr}^1$  = 10
```

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

If we know the Segment Offsets ( $B_{arr}^{inc}$ ) and Indices ( $II_{arr}^1$ ):

$B_{arr}^{inc}$  = [3, 4, 4, 6, 7, 7, 10] -- *inclusive scan of  $S_{arr}^1$  and*

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

We can efficiently compute the  $S_{arr}^1$  and  $F_{arr}$  arrays (also  $II_{arr}^2$ ) by:

```
 $S_{arr}^1$  = iota (length  $B_{arr}^{inc}$ )
```

```
> map (\i -> if i == 0 then  $B_{arr}^{inc}[i]$  else  $B_{arr}^{inc}[i] - B_{arr}^{inc}[i-1]$ )
```

```
 $F_{arr}$  = iota flenarr -- flenarr is the length of  $II_{arr}^1$ 
```

```
> map (\i -> if i == 0 then true else  $II_{arr}^1[i] \neq II_{arr}^1[i-1]$ )
```

Note:  $B_{arr}^{inc}$  different than  $B_{arr}^1$ : inclusive vs exclusive scan of shape!

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening

Several Re-Write Rules (inefficient for replicate & iota)

Jagged (Irregular Multi-Dim) Array Representation

Revisiting the Rewrites for Replicate & Iota Nested Inside Map

Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening

More Flattening Rules

Flattening by Function Lifting

Flattening Quicksort

Flattening Prime-Number (Sieve) Computation

# Revisiting Replicate inside 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 very simple gather operation:**

1. the shape of the result array is ns
2. build  $II^1$  of a jagged array of shape ns

# Revisiting Replicate inside 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 very simple gather operation:

1. the shape of the result array is ns
2. build  $II^1$  of a jagged array of shape ns
3. gather the corresponding values from ms by indexing through  $II^1$

```
 $\mathcal{F}$ (res = map2 (\n m -> replicate n m) ns ms)  $\Rightarrow$     -- ms = [7,3,8,9]  
1.  $S_{res}^1 = ns$                                            -- ns = [1,0,3,2]  
2.  $II_{res}^1 = \dots$  -- construct  $II^1$  for a jagged array of shape ns  
                                           -- [0, 2,2,2, 3,3]  
3.  $D_{res} = \mathbf{map} (\ \text{sgm} \rightarrow ms[\text{sgm}] \ ) \ II_{res}^1$     -- [7, 8,8,8, 9,9]
```

# Nested vs Flattened Parallelism: Iota inside 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] ]
```

```
res = map (\n -> iota n) ns
```

**The result is exactly the  $\text{II}^2$  array of a jagged array of shape ns**

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

1.  $S_{\text{res}}^1 = \text{ns}$   $-- \text{ ns} = [1, 3, 2]$
2.  $\text{II}_{\text{res}}^2 = \dots$   $-- \text{construct } \text{II}^2 \text{ of a jagged array of shape ns}$
4.  $D_{\text{res}} = \text{II}_{\text{res}}^2$

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

- What is "Flattening"? Recipe for Applying Flattening
- Several Re-Write Rules (inefficient for replicate & iota)
- Jagged (Irregular Multi-Dim) Array Representation
- Revisiting the Rewrites for Replicate & Iota Nested Inside Map
- Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

- Several Applications of Flattening
- More Flattening Rules
- Flattening by Function Lifting
- Flattening Quicksort
- Flattening Prime-Number (Sieve) Computation

# Revisiting Our Demonstration of How to Flatten

## Contrived Example:

```
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]]
```

## I. 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
```

## II. Distribute the map across every statement in the body

and adjust the inputs accordingly ( $\mathcal{F}$  denotes the transformation)

```
 $\mathcal{F}(\text{map } (\lambda i \rightarrow \text{map } (+i+1)) (\text{iota } i)) \text{ arr}) \equiv$ 
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  $\mathcal{F}(\text{map2 } (\lambda \text{ ip1r } \text{iot} \rightarrow \text{map2 } (+) \text{ ip1r } \text{iot}) \text{ ip1rs } \text{iots})$ 
```

We do **not** assume that arr contains strictly-positive integers.



# Revisiting Our Example: Think Like a Compiler

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

## Applying the new rules results in:

```
1.  $S_{res}^1 = \text{arr}$  -- arr = [1, 2, 3, 4]  
2. ( $B_{res}^1, F'_{res}$ ) = mkFlagArray arr 0 (iota (length arr))  
--  $B_{res}^1 = [0, 1, 3, 6]$   
3.  $F_{res} = \text{map bool.u32 } F'_{res}$   
4.  $II_{res}^1 = \text{sgmScan}^{inc} (+) 0 F_{res} F'_{res}$  -- [0, 1,1, 2,2,2, 3,3,3,3]  
5. ip1s = map (\ i -> i+1 ) arr -- [2, 3, 4, 5]  
6. iots = map2 (\ ind sgm -> ind -  $B_{res}^1[\text{sgm}]$  ) (iota flenres)  $II_{res}^1$   
-- =  $II_{arr}^2 = [0, 0,1, 0,1,2, 0,1,2,3]$   
7. ip1rs= map (\ sgm -> ip1s[sgm] )  $II_{res}^1$  -- [2, 3,3, 4,4,4, 5,5,5,5]  
8. in map2 (+) ip1rs iots -- [2, 3,4, 4,5,6, 5,6,7,8]
```

Lines 6 – 8 are trivially fusable  $\Rightarrow$  the iots and ip1rs arrays are not manifested in memory.

# Revisiting Our Example: Think Like a Human

## I. 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
```

## Using the new intuition results in:

```
1.  $S_{res}^1 = arr$  -- arr = [1, 2, 3, 4]
2.  $(B_{res}^1, F'_{res}) = mkFlagArray\ arr\ 0\ (iota\ (length\ arr))$ 
3.  $F_{res} = map\ bool.u32\ F'_{res}$ 
4.  $II_{res}^1 = sgmScan^{inc}\ (+)\ 0\ F_{res}\ F'_{res}$ 
5. in map2 (\ sgm ind -> let ip1      = arr[sgm] + 1
6.                      let iot_el = ind -  $B_{res}^1[sgm]$ 
7.                      in ip1 + iot_el
8.      )  $II_{res}^1\ (iota\ (length\ II_{res}^1))$ 
```

Have done a tiny bit better job than the compiler, as array ip1s is not manifested either.

# Fusion in Futhark

Map fusion:

$$(\text{map } g) \circ (\text{map } f) \equiv \text{map } (g \circ f)$$

$$x = \text{map } f \begin{bmatrix} a_1, & a_2, & \dots, & a_n \end{bmatrix}$$

$$x \equiv \begin{bmatrix} \downarrow & \downarrow & & \downarrow \\ f \ a_1, & f \ a_2, & \dots, & f \ a_n \end{bmatrix}$$

$$\text{map } g \ x = \begin{bmatrix} \downarrow & \downarrow & & \downarrow \\ g(f \ a_1), & g(f \ a_2), & \dots, & g(f \ a_n) \end{bmatrix}$$

$$\equiv \begin{matrix} = & = & & = \end{matrix} \\ \text{map } (g \circ f) \ x = \begin{bmatrix} g(f \ a_1), & g(f \ a_2), & \dots, & g(f \ a_n) \end{bmatrix}$$

**All other SOACs (reduce, scan, reduce-by-index, scatter) fuse with a map producer, if the mapped array is not used elsewhere.**

**Direct indexing in the map-produced array prevents fusion.**

E.g., assuming array `as` of length `n` the following will **not fuse**:

```
let xs = map f as
let ys = map (\i -> if i == 0 || i == n-1 then 0
                  else xs[i-1] + xs[i] + xs[i+1]) (iota n)
```

# **PERFORMANCE DEMONSTRATION**

# Demo on Prime Numbers: Haskell Implementation

If we have all primes from 2 to  $\sqrt{n}$  we could generate all multiples of these primes (up to  $n$ ) at once:  $\{[2*p:n:p]: p \text{ in } \text{sqr\_primes}\}$  in NESL. Also call algorithm recursively on  $\sqrt{n}$   
 $\Rightarrow$  Depth:  $O(\lg \lg n)$  (solution of  $n^{(1/2)^{\text{depth}}} = 2$ ). Work:  $O(n \lg \lg n)$

```
primesOpt :: Int -> [Int]
primesOpt n =
  if n <= 2 then [2]
  else
    let sqrtN = floor (sqrt (fromIntegral n))
        sqrt_primes = primesOpt sqrtN
        nested = map (\p->let m = (n `div` p)
                        in map (\j-> j*p)
                           [2..m])
                  sqrt_primes
        not_primes = reduce (++) [] nested
        mm = length not_primes
        zeros = replicate mm False
        prime_flags = scatter (replicate (n+1) True)
                          not_primes zeros
        (primes, _) = unzip $ filter (\(i,f)->f)
                                $ (zip [0..n] prime_flags)
    in drop 2 primes
```

# Demo on Prime Numbers: Haskell Implementation

If we have all primes from 2 to  $\sqrt{n}$  we could generate all multiples of these primes (up to  $n$ ) at once:  $\{[2*p:n:p]: p \text{ in } \text{sqr\_primes}\}$  in NESL. Also call algorithm recursively on  $\sqrt{n}$   
 $\Rightarrow$  Depth:  $O(\lg \lg n)$  (solution of  $n^{(1/2)^{\text{depth}}} = 2$ ). Work:  $O(n \lg \lg n)$

```
primesOpt :: Int -> [Int]
primesOpt n =
  if n <= 2 then [2]
  else
    let sqrtN = floor (sqrt (fromIntegral n))
        sqrt_primes = primesOpt sqrtN
        nested = map (\p->let m = (n `div` p)
                        in map (\j->j*p) [2..m]) sqrt_primes
        not_primes = reduce (++) [] nested
        mm = length not_primes
        zeros = replicate mm False
        prime_flags = scatter(replicate (n+1) not_primes)
        (primes, _) = unzip $ filter (\(i,f)->f) prime_flags
    in drop 2 primes
```

Assume  $n = 9$ ,  $\text{sqrtN} = 3$

call `primesOpt 3`  
 $n = 3, \text{sqrtN} = 1, \text{sqrt\_primes} = []$   
 $\text{nested} = [[]]; \text{not\_primes} = []$   
 $\text{mm} = 0; \text{zeros} = []$   
 $\text{prime\_flags} = [\text{T}, \text{T}, \text{T}, \text{T}]$   
 $\text{primes} = [0, 1, 2, 3]; \text{returns } [2..m]$

in `primesOpt 9`, after  
 return from `primesOpt 3`,  
 $\text{sqrt\_primes} = [2, 3]$   
 $\text{nested} = [[4, 6, 8], [6, 9]]$   
 $\text{not\_primes} = [4, 6, 8, 6, 9]$   
 $\text{mm} = 5; \text{zeros} = [\text{F}, \text{F}, \text{F}, \text{F}, \text{F}]$   
 $\text{prime\_flags} = [\text{T}, \text{T}, \text{T}, \text{T}, \text{F}, \text{T}, \text{F}, \text{T}, \text{F}]$   
 $\text{primes} = [0, 1, 2, 3, 5, 7]$   
 returns  $[2, 3, 5, 7]$

# **PERFORMANCE DEMONSTRATION**

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening  
Several Re-Write Rules (inefficient for replicate & iota)  
Jagged (Irregular Multi-Dim) Array Representation  
Revisiting the Rewrites for Replicate & Iota Nested Inside Map  
Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening  
More Flattening Rules  
Flattening by Function Lifting  
Flattening Quicksort  
Flattening Prime-Number (Sieve) Computation



# Eratosthenes Alg. for Computing Prime Numbers up To $n$

See also "Scan as Primitive Parallel Operation" [Bleeloch].

Start with an array of size  $n$  filled initially with 1, i.e., all are primes, and iteratively zero out all multiples of numbers up to  $\sqrt{n}$ .

```
int res[n] = {0, 0, 1, 1, 1, ..., 1}
for(i = 2; i <= sqrt(n); i++) { //sequential
    if ( res[i] != 0 ) {
        forall m  $\in$  multiples of  $i \leq n$  do {
            res[m] = 0;
        }
    }
}
```

Work:  $O(n \lg \lg n)$  but Depth:  $O(\sqrt{n})$  (Not Good Enough!)

# Eratosthenes Algorithm Improved for Parallel Execution

If we have all primes from 2 to  $\sqrt{n}$  we could generate all multiples of these primes (up to  $n$ ) at once:  $\{[2*p:n:p]: p \text{ in } \text{sqr\_primes}\}$  in NESL. Also call algorithm recursively on  $\sqrt{n}$   
 $\Rightarrow$  Depth:  $O(\lg \lg n)$  (solution of  $n^{(1/2)^{\text{depth}}} = 2$ ). Work:  $O(n \lg \lg n)$

```
primesOpt :: Int -> [Int]
primesOpt n =
  if n <= 2 then [2]
  else
    let sqrtN = floor (sqrt (fromIntegral n))
        sqrt_primes = primesOpt sqrtN
        nested = map (\p->let m = (n `div` p)
                          in map (\j->j*p)
                               [2..m])
                  sqrt_primes
        not_primes = reduce (++) [] nested
        mm = length not_primes
        zeros = replicate mm False
        prime_flags=scatter(replicate (n+1) True)
                          not_primes zeros
        (primes,_) = unzip $ filter (\(i,f)->f)
                          $ (zip [0..n] prime_flags)
    in drop 2 primes
```

# Batch of Rank-Search K Problems

**Rank-Search  $k$ :** finds the  $k^{th}$  smallest element of a vector.

Typically used for median computation.

```
let rankSearch (k: i64) (A: [] f32) : f32 =  
  let p = random_element A  
  let A_lth_p = filter (< p) A  
  let A_eqt_p = filter (==p) A  
  let A_gth_p = filter (> p) A  
  
  if (k <= A_lth_p.length)  
  then rankSearch k A_lth_p  
  else if (k <= A_lth_p.length + A_eqt_p.length)  
    then p  
    else rankSearch (k - A_lth_p.length - A_eqt_p.length) A_gth_p  
  
let main [m] (ks: [m] i64) (As: [m] [] f32) : [m] f32 =  
  map2 rankSearch ks As
```

# Quicksort with Nested Parallelism

```
nestedQuicksort :: [a] -> [a]
nestedQuicksort arr =
  if (length arr) <= 1 then arr else
  let i = getRand (0, (length arr) - 1)
      a = arr !! i
      s< = filter (\ x -> x < a) arr
      s= = filter (\ x -> x == a) arr
      s> = filter (\ x -> x > a) arr
      rs = map nestedQuicksort [s<, s>]
  in (rs !! 0) ++ s= ++ (rs !! 1)
```

Using  $n$  for the input's length: Average Work is  $O(n \lg N)$ .

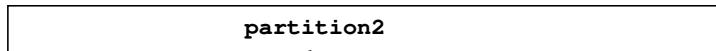
If filter would have depth 1, then Average Depth:  $O(\lg n)$ .

In practice we have depth:  $O(\lg^2 n)$ .

In principle, the implementation can be re-structured to use one `partition2` instead of three `filters`.

# Quicksort: Illustrating Flat-Parallel Execution

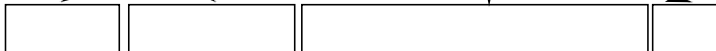
Macro  
Step 0



Macro  
Step 1



Macro  
Step 2



.....

# QuickHull with Nested Parallelism

---

## Algorithm 1 QuickHull

---

**Require:**  $S$ : a set of  $n \geq 2$  two-dimensional points

**Ensure:**  $CH$ : the convex-hull set of  $S$

$(A, B)$  = the leftmost  
and rightmost  
points of  $S$

$S_{1,2}$  = points of  $S$  above  
and below line  $AB$

$CH = \{A, B\} \cup$   
 $\text{findHull}(S_1, A, B) \cup$   
 $\text{findHull}(S_2, A, B)$

---

# QuickHull with Nested Parallelism

---

## Algorithm 2 Divide-And-Conquer Helper

---

```
1: procedure findHull( $S, P, Q$ )
2:    $Hull = \emptyset$ 
3:   if  $S \neq \emptyset$  then
4:      $C =$  furthest point of  $S$  from line  $PQ$ 
5:      $(S_l, S_r) =$  the points of  $S$  on the left-
6:                   and right-hand side of lines
7:                    $CP$  and  $CQ$ , respectively
8:                   (and not inside  $\triangle PCQ$ )
9:      $Hull = \{C\} \cup$ 
10:       $\text{findHull}(S_l, P, C) \cup$ 
11:       $\text{findHull}(S_r, C, Q)$ 
12:   return  $Hull$ 
```

---

## Parallel Basic Blocks Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening  
Several Re-Write Rules (inefficient for replicate & iota)  
Jagged (Irregular Multi-Dim) Array Representation  
Revisiting the Rewrites for Replicate & Iota Nested Inside Map  
Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening  
**More Flattening Rules**  
Flattening by Function Lifting  
Flattening Quicksort  
Flattening Prime-Number (Sieve) 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]
```

## 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.

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

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

$$1. S_{res}^0 = S_{arr}^0 \quad \rightarrow S_{res}^0 = [2]$$

```
2. indsp1 = scan (+) 0 S1arr -- indsp1 = [3, 5]
```

```
3. tmp = sgmScan ( $\odot$ )  $0_{\odot}$   $F_{arr}$   $D_{arr}$       -- tmp = [1, 4, 8, 6, 13]
```

```
4. Dres = map2(\s ip1 -> if s ≤ 0 then 0
                else tmp[ip1-1]) Sarr1 indsp1 -- Dres = [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  $\text{II}_{arr}^1$  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.  $D_{res} = \text{map2 } (\lambda y \text{ sgmind} \rightarrow f \text{ xs[sgmind] } y) \text{ } D_{yss} \text{ II}_{yss}^1$   
--  $\text{II}_{yss}^1 = [0,0,0,1,1]$ ,  $D_{res} = [5,6,7,12,10]$ 
```

# 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.  $S_{res}^0 = S_{i_s}^0 \text{ -- } = S_{i_s}^0 = [2]$

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]], S1xss=[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])
(S1xssthen, S1xsselse) = split spl (map (\ii -> S1xss[ii]) iinds) -- ([4,1],[3,2])
maskxss = map (\sgmind -> bs[sgmind]) ||1xss -- [F,F,F,T,T,T,T,F,F,T]
(brk, Dpxss) = partition2 maskxss Dxss
(Dxssthen, Dxsselse) = split brk Dpxss -- ([4,5,6,7,10],[1,2,3,8,9])
(S1resthen, Dresthen) = F(map f) (S1xssthen, Dxssthen) -- ([4,1], [5,6,7,8,11])
(S1reselse, Dreselse) = F(map g) (S1xsselse, Dxsselse) -- ([3,2], [2,4,6,16,18])
S1Pres = S1resthen ++ S1reselse -- [4,1,3,2]
S1res = scatter (replicate (length bs) 0) iinds S1Pres -- [3,4,2,1]
B1res = scanexc (+) 0 S1res -- [0,3,7,9]
FPres = mkFlagArray S1Pres 0 (map (+1) iinds) -- [2,0,0,0,4,1,0,0,3,0]
||1Pres = sgmscan (+) 0 FPres FPres |> map (\x -> x-1) -- [1,1,1,1,3,0,0,0,2,2]
||2Pres = ||2resthen ++ ||2reselse -- [0,1,2,3,0, 0,1,2,0,1]
sindsres = map2 (\sgm iin -> B1res[sgm] + iin) ||1Pres ||2Pres
-- [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 (Dresthen ++ Dreselse)
-- [2,4,6, 5,6,7,8, 16,18, 11]
(S1res, 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.  $n_{max} = \text{reduce } \max \text{ } 0 \text{ } i32 \text{ ns}$
2.  $g \text{ i m arr} = \text{if } i < m \text{ then } f \text{ arr } \text{else } \text{arr}$
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.  $-- (g \text{ i})^L \text{ ns } (S_{xss}^1, D_{xss})$

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

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 Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening  
Several Re-Write Rules (inefficient for replicate & iota)  
Jagged (Irregular Multi-Dim) Array Representation  
Revisiting the Rewrites for Replicate & Iota Nested Inside Map  
Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening  
More Flattening Rules  
**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  $f^L$  [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  $f^L$  (xss: [][]i32) : [][]i32 = ( $g^L$ )L -- ???
```

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

## 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 Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening  
Several Re-Write Rules (inefficient for replicate & iota)  
Jagged (Irregular Multi-Dim) Array Representation  
Revisiting the Rewrites for Replicate & Iota Nested Inside Map  
Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening  
More Flattening Rules  
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 Recap

### Part I: Flattening Nested and Irregular Parallelism

What is "Flattening"? Recipe for Applying Flattening  
Several Re-Write Rules (inefficient for replicate & iota)  
Jagged (Irregular Multi-Dim) Array Representation  
Revisiting the Rewrites for Replicate & Iota Nested Inside Map  
Revisiting the Solution to Our Example

### Part II: Flattening Nested and Irregular Parallelism

Several Applications of Flattening  
More Flattening Rules  
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!