Flattening Irregular Nested Parallelism

Cosmin E. Oancea
cosmin.oancea@diku.dk

Department of Computer Science (DIKU)
University of Copenhagen

December 2025 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
- "To Flatten or Not To Flatten, that is the question"

Zip, Unzip, iota, replicate

- `zip` : $[n]\alpha_1 \rightarrow [n]\alpha_2 \rightarrow [n](\alpha_1, \alpha_2)$
- `zip` $[a_1, \ldots, a_n]$ $[b_1, \ldots, b_n] \equiv$ $[(a_1, b_1), \ldots, (a_n, b_n)]$,

- `unzip` : $[n](\alpha_1, \alpha_2) \rightarrow ([n]\alpha_1, [n]\alpha_2)$
- `unzip` $[(a_1, b_1), \ldots, (a_n, b_n)] \equiv ([a_1, \ldots, a_n], [b_1, \ldots, b_n])$,
- In some sense `zip`/`unzip` are syntactic sugar

- `replicate` : (n: int) $\rightarrow \alpha \rightarrow [n]\alpha$
- `replicate n a` $\equiv$ [a, a, ..., a],

- `iota` : (n: int) $\rightarrow$ [n]int
- `iota n` $\equiv$ [0, 1, ..., 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 α.
- $\texttt{map} : (\alpha \to \beta) \to [n]\alpha \to [n]\beta$
  $\texttt{map f } [x_1, \ldots, x_n] = [\texttt{f } x_1, \ldots, \texttt{f } x_n]$,
  i.e., $x_i : \alpha, \forall i$, and $\texttt{f} : \alpha \to \beta$.
- $\texttt{reduce} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to \alpha$
  $\texttt{reduce} \odot \texttt{e } [x_1, x_2, \ldots, x_n] = e \odot x_1 \odot x_2 \odot \ldots \odot x_n$,
  i.e., $e : \alpha, \quad x_i : \alpha, \forall i$, and $\odot : \alpha \to \alpha \to \alpha$.
- $\texttt{scan}^{exc} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$
  $\texttt{scan}^{exc} \odot \texttt{e } [x_1, \ldots, x_n] = [e, e \odot x_1, .., e \odot x_1 \odot .. x_{n-1}]$
  i.e., $e : \alpha, \; x_i : \alpha, \forall i$, and $\odot : \alpha \to \alpha \to \alpha$.
- $\texttt{scan}^{inc} : (\alpha \to \alpha \to \alpha) \to \alpha \to [n]\alpha \to [n]\alpha$
  $\texttt{scan}^{inc} \odot \texttt{e } [x_1, \ldots, x_n] = [e \odot x_1, \ldots, e \odot x_1 \odot \ldots x_n]$
  i.e., $e : \alpha, \; x_i : \alpha, \forall i$, and $\odot : \alpha \to \alpha \to \alpha$.

Map2, Filter

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

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:

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: 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: $(\alpha \to \text{Bool}) \to [n]\alpha \to (\text{i32},[n]\alpha)$

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(rue) for even nums.

Partition2/Filter Implementation

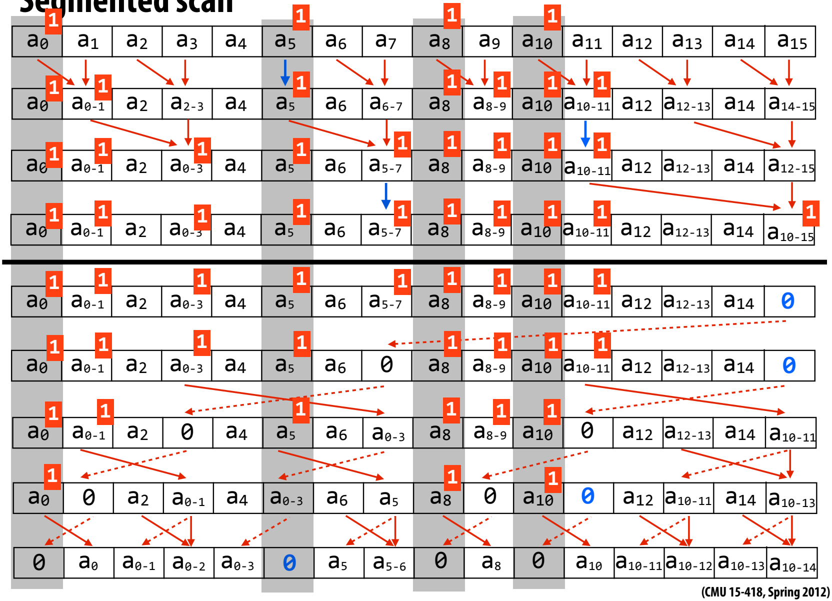
$\texttt{partition2:}\ (\alpha \to \texttt{Bool}) \to [\texttt{n}]\alpha \to (\texttt{i32},[\texttt{n}]\alpha)$

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

Correctness Argument:

verify sequential semantics + associative operator ⇒ parallel semantics also holds

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.***[**]

[**] 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)

```
𝓕(map (\i -> map (+(i+1)) (iota i)) arr) ≡
1. let ip1s = map (\i -> i+1) arr in -- [2, 3, 4, 5]
2. let iots = 𝓕(map (\i -> (iota i)) arr) in
3. let ip1rs= 𝓕(map2 (\ i ip1 -> (replicate i ip1)) arr ip1s)
4. in  𝓕(map2 (\ ip1r iot -> map2 (+) ip1r iot) 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 = 𝓕(map (\i -> iota i) arr)
```

$\equiv$

```
inds = scan^exc (+) 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 = sgmScan^exc (+) 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= 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= sgmScan^inc (+) 0 flag vals          -- [2,3,3,4,4,4,5,5,5,5]
```

According to rule “map nested inside a map”

```
𝓕(map2 (\ ip1r iot -> map2 (+) ip1r iot) ip1rs iots)
≡
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
- “To Flatten or Not To Flatten, that is the question”

Nested *vs* Flattened Parallelism: Scan inside a Map

(1) Scan nested inside a map:

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

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

```
sgmScan^inc (+) 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^1_{arr}, 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.

$\mathcal{F}$(`res = map (\row -> scan (`$\odot$`) 0`$_\odot$` row) arr)` $\Rightarrow$

S^1_{res} = S^1_{arr}

D_{res} = `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.

$\mathcal{F}$(res = map (\row -> map f row) arr) $\Rightarrow$

S^1_{res} = S^1_{arr}

D_{res} = 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.

$\mathcal{F}$(res = map2 (\n m -> replicate n m) ns ms) $\Rightarrow$ -- ms = [7,3,8,9]

1. S^1_{res} = ns -- ns = [1,0,3,2]
2. inds = **scan**^{exc} (+) 0 ns -- [0,1,1,4]
3. |> **map2** (\n i->**if** n>0 **then** i **else** -1) ns -- inds = [0,-1,1,4]
4. size = (last inds) + (last ns) -- 4 + 2 = 6
5. vls = **scatter** (**replicate** size 0) inds ms -- [7, 8, 0, 0, 9, 0]
6. F_{res} = **scatter** (**replicate** size **false**) inds (**replicate** size **true**) -- [1, 1, 0, 0, 1, 0]
7. D_{res} = sgmScan^{inc} (+) 0 F_{res} vls -- [7, 8, 8, 8, 9, 9]

Nested *vs* Flattened Parallelism: Iota inside Map

(4) Iota nested inside a map (`(iota n)`$\equiv$`[0,...,n-1]`):

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

Nested *vs* Flattened Parallelism: Iota inside Map

(4) Iota nested inside a map (`(iota n)≡[0,...,n-1]`):

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

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

Note 1: `iota n` ≡ scan^{exc} `(+) 0 (replicate n 1)`.

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

Shape-Based Representation

- Two dimensional arrays:

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

$\Rightarrow$

S^0_{arr} = [4]

S^1_{arr} = [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] ]
```

$\Rightarrow$

S^0_{arr} = [4]
S^1_{arr} = [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] ] ]
```

$\Rightarrow$

Shape-Based Representation

- Two dimensional arrays:

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

S^0_{arr} = [4]
S^1_{arr} = [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^0_{arr} = [3]
S^1_{arr} = [0, 4, 3]
S^2_{arr} = [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:

$\text{length } S^i_{arr} = \textbf{reduce } (+)\ 0\ S^{i-1}_{arr},\ \forall 1 \leq i < n$
$\text{length } D_{arr} = \textbf{reduce } (+)\ 0\ S^{n-1}_{arr}$

Flat Representation: Auxiliary Structures

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

$\Rightarrow$

S^0_{arr} = [3]
S^1_{arr} = [0, 4, 3]
S^2_{arr} = [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^1_{arr} = [0, 0, 6]
  B^2_{arr} = [0, 3, 4, 4, 6, 7, 7]

- Flag Array (F): start of a segment indicated by a `true` value (could also use `!=0` integrals), e.g., used for segmented scans:

  F^1_{arr} = [1, 0, 0, 0, 0, 0, 1, 0, 0, 0]
  F^2_{arr} = [1, 0, 0, 1, 1, 0, 1, 1, 0, 0]

- Segment and Inner indices (`II`):

  II^1_{arr} = [1, 1, 1, 1, 1, 1, 2, 2, 2, 2]
  II^2_{arr} = [0, 0, 0, 1, 3, 3, 0, 2, 2, 2]
  II^3_{arr} = [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:

$$\texttt{let}\ (S^1_{yss}, D_{yss}) = \mathcal{F}(\textbf{map2}\ (\backslash xs\ y \to \textbf{replicate}\ (\text{length}\ xs)\ y)\ xss\ ys)$$
$$\texttt{let}\ D_{rss} = \textbf{map2}\ (\backslash\ x\ y \to x + y)\ D_{xss}\ D_{yss}$$

⇒

$$\begin{array}{lcl} D_{xss} & = & [1,\ 2,\ 3,\ 5,\ 7] \\ & & \ \ +\ \ +\ \ +\ \ +\ \ + \\ D_{yss} & = & [4,\ 4,\ 4,\ 1,\ 1] \\ & & \ \ =\ \ =\ \ =\ \ =\ \ = \\ D_{rss} & = & [5,\ 6,\ 7,\ 6,\ 8] \end{array}$$

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 D_{rss} = **map2** (\ x sgmind -> x + ys[sgmind]) D_{xss} II^1_{rss}

⇒

S^1_{rss} = [3, 0, 2]

II^1_{rss} = [0, 0, 0, 2, 2]

D_{xss} = [1, 2, 3, 5, 7]

D_{rss} = [1+4, 2+4, 3+4, 5+1, 7+1] = [5, 6, 7, 6, 8]

But what have we gained? Creating II^1_{rss} 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 D_{rss} = **map2** (\ y sgmind -> x*ys[sgmind] + zs[sgmind]) D_{xss} II^1_{rss}
⇒
II^1_{rss} = [0, 0, 0, 2, 2]
D_{xss} = [1, 2, 3, 5, 7]
D_{rss} = [1*4+1, 2*4+1, 3*4+1, 5*1+3, 7*1+3] = [5, 9, 13, 8, 10]

We build II^1_{rss} 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
```

$\Rightarrow$

```
rss = [ [[3],[5]], [[6,7]] ]
```

Using the auxiliary structures we indirectly access other arrays:

$$
\begin{array}{l}
\textbf{let}\ D_{rss} = \mathtt{map3}(\backslash\ s1\ s2\ s3 \rightarrow \textbf{let}\ ind\_x = B^1_{xss}[s1] + s2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{let}\ ind\_y = B^1_{yss}[s1] + s3 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ X[ind\_x] + Y[ind\_y] \\
\qquad\qquad)\ II^1_{rss}\ II^2_{rss}\ II^3_{rss}
\end{array}
$$

$\Rightarrow$

$B^1_{xss} = [0,\ 2]$

$B^1_{yss} = [0,\ 1]$

$II^1_{rss} = [0,\ 0,\ 1,\ 1]$

$II^2_{rss} = [0,\ 1,\ 0,\ 0]$

$II^3_{rss} = [0,\ 0,\ 0,\ 1]$

$D_{rss} = [\ D_{xss}[0+0]+D_{yss}[0+0],\ D_{xss}[0+1]+D_{yss}[0+0]$
$\quad ,\ D_{xss}[2+0]+D_{yss}[1+0],\ D_{xss}[2+0]+D_{yss}[1+1]\ ]$

$D_{rss} = [\ 1+2,\ 3+2,\ 2+4,\ 2+5] = [\ 3,\ 5,\ 6,\ 7]$

Constructing the Offset Indices (B)

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

$\Rightarrow$

S^0_{arr} = [3]
S^1_{arr} = [0, 4, 3]
S^2_{arr} = [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^1_{arr} = [0, 0, 6]
B^2_{arr} = [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] ] ]
```

$\Rightarrow$

S^0_{arr} = [3]
S^1_{arr} = [0, 4, 3]
S^2_{arr} = [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^1_{arr} = [0, 0, 6]
B^2_{arr} = [0, 3, 4, 4, 6, 7, 7]

How to construct Offset Indices (B)?

By exclusive scanning the corresponding shape and reindexing!

B^2_{arr} = **scan**^{exc} (+) 0 S^2_{arr} -- *[0, 3, 4, 4, 6, 7, 7]*

B^1_{arr} = **scan**^{exc} (+) 0 S^1_{arr} -- *[0, 0, 4]*
  |> **map** (\ i -> B^2_{arr}[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]]
$\Rightarrow$
S^0_{arr} = [7]
S^1_{arr} = [3, 1, 0, 2, 1, 0, 3]
$flen_{arr}$ = **reduce** (+) 0 S^1_{arr} = 10
D_{arr} = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Segment and Inner indices (II):

II^1_{arr} = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]
II^2_{arr} = [0, 1, 2, 0, 0, 1, 0, 0, 1, 2]

Constructing Segment and Inner indices (II):

(B^1_{arr}, F_{arr}) = mkFlagArray S^1_{arr} 0 (**iota** (length S^1_{arr}))
-- ([0, 3, 4, 4, 6, 7, 7], [0, 0, 0, 1, 3, 0, 4, 6, 0, 0])

II^1_{arr} = ???
II^2_{arr} = ???

Constructing the Segment and Inner Indices

I need to get this:

II^1_{arr} = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]

from this:

```
(_, F_arr) = mkFlagArray S1_arr 0 (iota (length S1_arr))
        --  [0, 0, 0, 1, 3, 0, 4, 6, 0, 0]
```

How ?

Constructing the Segment and Inner Indices

I need to get this:

```
II²arr     = [0, 1, 2, 0, 0, 1, 0, 0, 1, 2]
```

from these:

```
Barr        = [0, 3, 4, 4, 6, 7, 7]
II¹arr     = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]
iota 10 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

$\mathtt{II}^1_{arr}$ and $\mathtt{II}^2_{arr}$ 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] ]
⇒
S⁰_arr = [7]
S¹_arr = [3, 1, 0, 2, 1, 0, 3]
flen_arr = reduce (+) 0 S¹_arr = 10
D_arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Segment and Inner indices (II):

```
II¹_arr = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]
II²_arr = [0, 1, 2, 0, 0, 1, 0, 0, 1, 2]
```

Constructing Segment and Inner indices (II):

```
(B¹_arr, F'_arr) = mkFlagArray S¹_arr 0 (iota (length S¹_arr))
        -- ([0, 3, 4, 4, 6, 7, 7], [0, 0, 0, 1, 3, 0, 4, 6, 0, 0])
F_arr = map bool.u32 F'_arr

II¹_arr = sgmScan^inc (+) 0 F_arr F'_arr

II²_arr = map2 (\ i sgm -> i - B¹_arr[sgm] ) (iota flen_arr) II¹_arr
-- ^ this fuses better & performs less memory traffic than the below:
II²_arr = sgmScan^inc (+) 0 F_arr (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] ]
⇒
S⁰_arr = [7]
S¹_arr = [3, 1, 0, 2, 1, 0, 3]
flen_arr = reduce (+) 0 S¹_arr = 10
D_arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

If we know the Segment Offsets (B^{inc}_{arr}) and Indices (II^1_{arr}):

```
B^inc_arr = [3, 4, 4, 6, 7, 7, 10] -- inclusive scan of S¹_arr and
II¹_arr   = [0, 0, 0, 1, 3, 3, 4, 6, 6, 6]
```

We can efficiently compute the S^1_{arr} and F_{arr} arrays (also II^2_{arr}) by:

```
S¹_arr = iota (length B^inc_arr)
    |> map (\i -> if i == 0 then B^inc_arr[i] else B^inc_arr[i] - B^inc_arr[i-1])

F_arr = iota flen_arr -- flen_arr is the length of II¹_arr
    |> map (\i -> if i == 0 then true else II¹_arr[i] != II¹_arr[i-1])
```

Note: B^{inc}_{arr} different than B^1_{arr}: 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
- “To Flatten or Not To Flatten, that is the question”

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 $\mathtt{II}^1$ of a jagged array of shape `ns`
3. gather the corresponding values from `ms` by indexing through $\mathtt{II}^1$

$\mathcal{F}$(res = map2 (\n m -> replicate n m) ns ms) $\Rightarrow$ -- *ms = [7,3,8,9]*

1. S^1_{res} = ns -- *ns = [1,0,3,2]*
2. II^1_{res} = ... -- *construct II^1 for a jagged array of shape ns*
   -- *[0, 2,2,2, 3,3]*
3. D_{res} = **map** (\ sgm -> ms[sgm]) II^1_{res} -- *[7, 8,8,8, 9,9]*

Nested *vs* Flattened Parallelism: Iota inside Map

(4) Iota nested inside a map (`(iota n)`≡`[0,...,n-1]`):

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

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

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

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

1. S^1_{res} = ns -- *ns = [1, 3, 2]*
2. II^2_{res} = ... -- *construct* II^2 *of a jagged array of shape ns*
4. D_{res} = II^2_{res}

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)

```
𝓕(map (\i -> map (+(i+1)) (iota i)) arr) ≡
1. let ip1s = map (\i -> i+1) arr in -- [2, 3, 4, 5]
2. let iots = 𝓕(map (\i -> (iota i)) arr) in
3. let ip1rs= 𝓕(map2 (\ i ip1 -> (replicate i ip1)) arr ip1s)
4. in  𝓕(map2 (\ ip1r iot -> map2 (+) ip1r iot) ip1rs iots)
```

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

Revisiting Our Example: Think Like a Compiler

```
𝓕(map (\i -> map (+(i+1)) (iota i)) arr) ≡
1. let ip1s = map (\i -> i+1) arr in -- [2, 3, 4, 5]
2. let iots = 𝓕(map (\i -> (iota i)) arr) in
3. let ip1rs= 𝓕(map2 (\ i ip1 -> (replicate i ip1)) arr ip1s)
4. in  𝓕(map2 (\ ip1r iot -> map2 (+) ip1r iot) ip1rs iots)
```

Applying the new rules results in:

1. S^1_{res} = arr -- *arr = [1, 2, 3, 4]*
2. (B^1_{res}, F'_{res}) = mkFlagArray arr 0 (**iota** (length arr))
   -- *B^1_{res} = [0, 1, 3, 6]*
3. F_{res} = **map** **bool**.u32 F'_{res}
4. II^1_{res} = $sgmScan^{inc}$ (+) 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^1_{res}[sgm]) (**iota** $flen_{res}$) II^1_{res}
   -- *= II^2_{arr} = [0, 0,1, 0,1,2, 0,1,2,3]*
7. ip1rs= **map** (\ sgm -> ip1s[sgm]) II^1_{res} -- *[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 ⇒ 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^1_{res} `= arr` `-- arr = [1, 2, 3, 4]`
2. `(`B^1_{res}`, F'`$_{res}$`) = mkFlagArray arr 0 (`**`iota`**` (length arr))`
3. F_{res} `=` **`map bool`**`.u32 F'`$_{res}$
4. II^1_{res} `= sgmScan`^{inc} `(+) 0` F_{res} `F'`$_{res}$
5. **`in map2`** `(\ sgm ind ->` **`let`** `ip1    = arr[sgm] + 1`
6. **`let`** `iot_el = ind -` B^1_{res}`[sgm]`
7. **`in`** `ip1 + iot_el`
8. `)` II^1_{res} `(`**`iota`** `(length` II^1_{res}`))`

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

Fusion in Futhark

Map fusion:

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

$$
\begin{array}{rlccccc}
x = & \text{map } f & [\ a_1, & a_2, & .., & a_n &] \\
 & & \downarrow & \downarrow & & \downarrow & \\
x \equiv & & [\ f\ a_1, & f\ a_2, & .., & f\ a_n &] \\
 & & \downarrow & \downarrow & & \downarrow & \\
\text{map } g\ x = & & [\ g(f\ a_1), & g(f\ a_2), & .., & g(f\ a_n) &] \\
\equiv & & = & = & & = & \\
\text{map } (g \circ f)\ x = & & [\ g(f\ a_1), & g(f\ a_2), & .., & g(f\ a_n) &]
\end{array}
$$

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)
```

Demonstrating Performance of New *vs* Old Rules

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 in sqr_primes}` in NESL. Also call algorithm recursively on $\sqrt{n}$ $\Rightarrow$ Depth: $O(lg\ lg\ n)$ (solution of $n^{(1/2)^{depth}} = 2$). 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 in sqr_primes}` in NESL. Also call algorithm recursively on $\sqrt{n}$ $\Rightarrow$ Depth: $O(lg\ lg\ n)$ (solution of $n^{(1/2)^{depth}} = 2$). 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
```

```
Assume n = 9, sqrtN = 3

call primesOpt 3
n = 3,sqrtN = 1,sqrt_primes=[2]
nested = [[]]; not_primes = []
mm = 0; zeros = []
prime_flags = [T,T,T,T]
primes = [0,1,2,3]; returns[2,3]

in primesOpt 9, afer
return from primesOpt3,
sqrt_primes = [2,3]
nested = [[4,6,8],[6,9]]
not_primes = [4,6,8,6,9]
mm=5; zeros=[F,F,F,F,F]
prime_flags=[T,T,T,T,F,T,F,T,F,F]
primes = [0,1,2,3,5,7]
returns [2,3,5,7]
```

Demonstrating Performance of New *vs* Old Rules

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
- “To Flatten or Not To Flatten, that is the question”

Eratosthenes Alg. for Computing Prime Numbers up To n

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

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 ∈ multiples of i ≤ 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 in sqr_primes}` in NESL. Also call algorithm recursively on $\sqrt{n}$
$\Rightarrow$ Depth: $O(lg\ lg\ n)$ (solution of $n^{(1/2)^{depth}} = 2$). 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)
```

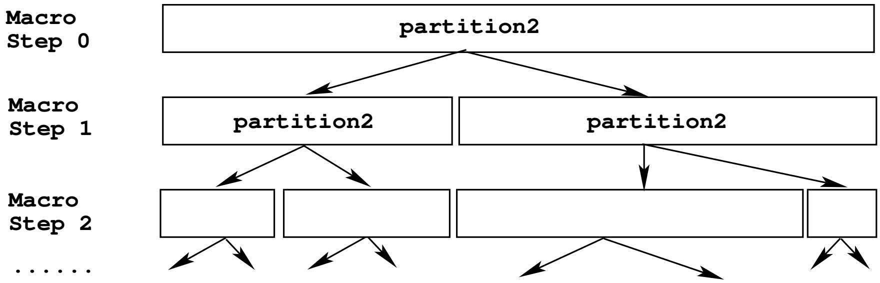
Using *n* for the input's length: Average Work is $O(n\ lg\ N)$.

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



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

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

QuickHull with Nested Parallelism

Algorithm 2 Divide-And-Conquer Helper

procedure *findHull*(S, P, Q)
    $Hull = \emptyset$
    if $S \neq \emptyset$ **then**
        C = furthest point of S from line PQ
        (S_l, S_r) = the points of S on the left-
            and right-hand side of lines
            CP and CQ, respectively
            (and not inside ΔPCQ)
        $Hull = \{C\} \cup$
            findHull$(S_l, P, C) \cup$
            findHull(S_r, C, Q)
    return $Hull$

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 elem in a segment holds the reduced value of the segment;
4. segment's last element is extracted by a map operation.

$\mathcal{F}$(res = map (\row -> reduce $\odot$ $0_\odot$ row) arr) $\Rightarrow$
-- S^0_{arr} = [2], S^1_{arr} = [3,2], F_{arr} = [1,0,0,1,0], D_{arr} = [1,3,4,6,7]
1. S^0_{res} = S^0_{arr} -- S^0_{res} = [2]
2. indsp1 = **scan** (+) 0 S^1_{arr} -- indsp1 = [3, 5]
3. tmp = sgmScan ($\odot$) $0_\odot$ F_{arr} D_{arr} -- tmp = [1, 4, 8, 6, 13]
4. D_{res} = **map2**(\s ip1 -> **if** s$\leq$0 **then** $0_\odot$
else tmp[ip1-1]) S^1_{arr} 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

$\mathcal{F}$(res = map (\row -> reduce $\odot$ $0_\odot$ row) arr) $\Rightarrow$

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

1. S^0_{res} = S^0_{arr} -- S^0_{res} = [2]
2. D_{res} = hist ($\odot$) $0_\odot$ (S^0_{arr}[0]) II^1_{arr} D_{arr}

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

- practical performance refers to how many global-memory accesses you perform
- accessing II^1_{arr} from memory has significant cost
- in some practical cases, it might be more efficient to not manifest II^1_{arr}, but instead to compute its elements by binary searching the B^1_{arr} 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 $\mathtt{II}^1_{arr}$ to indirectly access in the `xs` array:

$\mathcal{F}$(res = map2 (\x ys -> map (f x) ys) xs yss) $\Rightarrow$

-- xs = [1,3], S^1_{yss} = [3,2], F_{yss} = [1,0,0,1,0], D_{yss} = [4,5,6,9,7]

1. S^1_{res} = S^1_{yss}
2. D_{res} = **map2** (\y sgmind -> f xs[sgmind] y) D_{yss} II^1_{yss}

-- II^1_{yss} = [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_{yss} is obtained by summing up

- the start offset of every segment, which we get from B^1_{yss}, and
- the index inside the segment, which we get from `is`

$\mathcal{F}$(res = map2 (\i xs -> xs[i]) is xss) $\Rightarrow$

-- *is = [2,0], S^1_{xss} = [3,2], B^1_{xss} = [0,3], D_{xss} = [4,5,6,9,7]*

1. S^0_{res} = S^0_{is} -- *= S^0_{is} = [2]*
2. D_{res} = **map2** (\ off i -> D_{xss}[off + i]) B^1_{xss} 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 (λi -> bs[i]) (iota (length b)) -- [1,3,0,2]
2. xss_then = gatherThen iinds xss -- ([4,1], [4,5,6,7, 10])
3. xss_else = gatherElse iinds xss -- ([3,2], [1,2,3,  8,9])
-- Recursively Flatten the Then and Else Branches!
4. res_then = F(map (map (+1)) xss_then) -- ([4,1], [5,6,7,8, 11])
5. res_else = F(map (map (*2)) xss_else) -- ([3,2], [2,4,6,16,18])
6. res = inversePermute iinds (res_then++res_else)
-- ([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^1_{xss}=[3,4,2,1], f=map (+1), g=map (*2)

$\mathcal{F}$(res = map2 (\b xs -> if b **then** f xs **else** g xs) bs xss) $\Rightarrow$
(spl, iinds) = partition2 bs (**iota** (length bs)) *-- (2, [1,3,0,2])*
($S^1_{xss_{then}}$, $S^1_{xss_{else}}$) = split spl (**map** (\ii -> S^1_{xss}[ii]) iinds)*--([4,1],[3,2])*
$mask_{xss}$ = **map** (\sgmind -> bs[sgmind]) II^1_{xss} *-- [F,F,F,T,T,T,T,F,F,T]*
(brk, D^p_{xss}) = partition2 $mask_{xss}$ D_{xss}
($D_{xss_{then}}$, $D_{xss_{else}}$) = split brk D^p_{xss} *-- ([4,5,6,7,10],[1,2,3,8,9])*
($S^1_{res_{then}}$, $D_{res_{then}}$) = $\mathcal{F}$(**map** f) ($S^1_{xss_{then}}$, $D_{xss_{then}}$) *-- ([4,1], [5,6,7,8,11])*
($S^1_{res_{else}}$, $D_{res_{else}}$) = $\mathcal{F}$(**map** g) ($S^1_{xss_{else}}$, $D_{xss_{else}}$) *-- ([3,2], [2,4,6,16,18])*
S^{1P}_{res} = $S^1_{res_{then}}$++$S^1_{res_{else}}$ *-- [4,1,3,2]*
S^1_{res} = **scatter** (**replicate** (length bs) 0) iinds S^{1P}_{res} *-- [3,4,2,1]*
B^1_{res} = $\mathbf{scan}^{exc}$ (+) 0 S^1_{res} *-- [0,3,7,9]*
F^P_{res} = mkFlagArray S^{1P}_{res} 0 (**map** (+1) iinds) *-- [2,0,0,0,4,1,0,0,3,0]*
II^{1P}_{res} = sgmscan (+) 0 F^P_{res} F^P_{res}|> **map**(\x -> x-1) *-- [1,1,1,1,3,0,0,0,2,2]*
II^{2P}_{res} = $II^2_{res_{then}}$++$II^2_{res_{else}}$ *-- [0,1,2,3,0, 0,1,2,0,1]*
$sinds_{res}$ = **map2** (\sgm iin -> B^1_{res}[sgm] + iin) II^{1P}_{res} II^{2P}_{res}
--[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]
D_{res} = **scatter** (**replicate** $flen_{res}$ 0) $sinds_{res}$ ($D_{res_{then}}$++$D_{res_{else}}$)
-- [2,4,6, 5,6,7,8, 16,18, 11]
(S^1_{res}, D_{res})

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!

```
F(res = map2 (\n xs -> loop (xs) for i < n do f xs) ns xss) ⇒
1. n_max = reduce max 0i32 ns
2. g i m arr = if i < m then f arr else arr
3. loop(S^1_xss, D_xss) for i < n_max do
4.     F(map2 (g i)) ns (S^1_xss, D_xss)
5.     -- (g i)^L ns (S^1_xss, 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!

```
𝓕(res = map2 (\n xs -> loop (xs) for i < n do f xs) ns xss) ⇒
1. n_max = reduce max 0i32 ns
2. g i m arr = if i < m then f arr else arr
3. loop(S¹_xss, D_xss) for i < n_max do
4.     𝓕(map2 (g i)) ns (S¹_xss, D_xss)
5.     -- (g i)^L ns (S¹_xss, 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 (i) it is scattered into the result, and (2) it is filtered out from the running set of segments.

Nested *vs* Flattened Parallelism: Do Loop Inside a Map

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

The general case can be solved by the technique of
"reducing it to a more challenging/general problem" :))

A loop such as

```
loop (xs) = (xs0) while goOn xs do f xs
```

is equivalent with a call to a tail recursive function

```
fTailRec xs0
  where
    fTailRec xs = if goOn xs
                  then f xs
                  else xs     -- base case
```

We already know how to flatten an if-then-else; if we figure out how to flatten a function called directly inside a map, we are done

```
𝓕(res = map (\xs0 -> loop (xs) = (xs0) while goOn xs do f xs) xss0)
≡
𝓕(res = map (\xs0 -> fTailRec xs0) xss0)
```

Flattening by Function Lifting: Basic Idea

Assume a simple function `f`:

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

`f` lifted, denoted $\texttt{f}^L$ 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 f^L [n] (xs: [n]i32) : [n]i32 =
    xs +^L (replicate n 1)
```

- Locals such as `x` ⇒ left alone
- Global such as `+` ⇒ lifted ($+^L$)
- Constants such as `k` ⇒ `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: []f32) : [][]f32 = map g xs -- = gᴸ xs
let fᴸ (xss: [][]f32) : [][][]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: []f32) : [][]f32 = map g xs -- = gᴸ xs
let fᴸ (xss: [][]f32) : [][][]i32 = (gᴸ)ᴸ -- ???
```

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

```
let f (xs: []f32) : [][]f32 = map g xs -- = gᴸ xs
-- in nested parallel form
let fᴸ (xss: [][]f32) : [][][]f32 =
    segment xss (gᴸ (concat xss))
```

In Haskell Notation:

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

Flattening by Function Lifting: General Case!

let f (xs: []f32) : []…[]f32 = **map** g xs *-- = g^L xs*

let f^L (xss: [][]f32) : [][]…[]f32 = $(g^L)^L$ *-- ???*

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

A 3D array rsss: [][][]f32 has the representation (S^0_{rsss}, S^1_{rsss}, S^2_{rsss}, D_{rsss})

let f (xs: []f32) : []…[]f32 = **map** g xs *-- = g^L xs*

-- in nested parallel form

let f^L (xss: [][]f32) : [][]…[]f32 =
    segment xss (g^L (concat xss))

-- in flatten form

let f^L (S^0_{xss}: i64, S^1_{xss}: []i64, D_{xss}: []f32)
            : (i64, []i64, …, []i64, []f32) =
    let (Sf^0_{rsss}, …, Sf^q_{rsss}, D_{rss}) = g^L (S^0_{xss}, S^1_{xss}, D_{xss})
    let (S^0_{rsss}, S^1_{rsss}, …, S^q_{rsss}) = (S^0_{xss}, S^1_{xss}, …, Sf^q_{rsss})
    in (S^0_{rsss}, S^1_{rsss}, …, S^q_{rsss}, D_{rss})

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
- "To Flatten or Not To Flatten, that is the question"

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 quicksort^L (xss: [][]f32) : [][]f32 =
  map (\xs ->
        if isSorted xs then xs else  -- oh, nooooo, an if-then-else!
        let i = getRand (0, (length xs) - 1)
        let a = xs[i]
        let bs = map (< a) xs
        let (q, xs^p) = partition2 bs 0.0f32 xs
        let (xs_<, xs_>=) = split q xs^p
        in  concat <| map quicksort [xs_<, xs_>=]
  ) xss
```

Important observations:

- `map quicksort` $\equiv$ `quicksort`^{L}
- the flat data of $[xs_<, xs_\geq] \equiv xs^p$, the result of `partition2`
- `map(map quicksort)`$\equiv$`quicksort`$^{L^L}$$\equiv$`segment o quicksort`^{L}`o concat`

Lifting Quicksort

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

```
let quicksort^L (S^1_xss :[]i32, D_xss :[]f32): ([]i32,[]f32) = --(xss: [][]f32)
  if isSorted D_xss then (S^1_xss :[]i32, D_xss :[]f32) else -- big cheat!
  let (S^1_bss, D_bss) = 𝓕 (
      map (\xs ->
            let i = getRand (0, (length xs) - 1)
            let a = xs[i]
            let bs = map (< a) xs
            in  bs
          ) xss
    )
  let (ps, (S^1_xss^p ,D_xss^p )) = partition2^L D_bss 0.0f32 (S^1_xss, D_xss)
  -- Invariant: S^1_xss^p == S^1_bss == S^1_xss
  let S^1_[xss_<,xss_≥] = filter (!=0) <| flatten <|
        map2 (λ p s -> if s==0 then [0,0] else [p,s-p]) ps S^1_xss
  in  quicksort^L (S^1_[xss_<,xss_≥], D_xss^p)
```

- $S^1_{[xss_{<a}, xss_{\geq a}]}$ is the shape of `[xs<, xs≥]`
- `(concat <| quicksort^L)^L xsss` $\equiv$ `concat <| segment xsss <| quicksort^L (concat xsss)` $\equiv$ `quicksort^L (concat xsss)`
- The function looks tail recursive now: let's replace it with a loop!

Lifting Quicksort: Final Implementation

```
let quicksort^L [m][n] (S^1_xss:[m]i32, D_xss:[n]f32): [n]f32 =
  let (stop, count) = (isSorted D_xss, 0i32)
  let (_,res,_,_) =
    loop(S^1_xss, D_xss, stop, count) while (!stop) do
      -- compute helper-representation structures
      let B^1_xss = scan^exc (+) 0 S^1_xss
      let F^1_xss = mkFlagArray S^1_xss 0i32 <| map (+1) <| iota m
      let II^1_xss = sgmscan (+) 0 F^1_xss <|
                  map (\f -> if f==0 then 0 else f-1) F^1_xss
      -- flattening quicksort:
      let rL = map (\u -> randomInd (0,u-1) count) S^1_xss
      let aL = map3(\r l i-> if l <= 0 then 0.0 else D_xss[B^1_xss[i]+r]
                   ) rL S^1_xss (iota m)
      let D_bss= map2 (\x sgmind -> aL[sgmind] > x ) D_xss II^1_xss
      let (ps, (S^1_xssp ,D^per_xss)) = partition2^L D_bss 0.0f32 (S^1_xss, D_xss)
      let S^1_[xss<,xss>=] = filter (!=0) <| flatten <|
                map2 (\ p s -> if s==0 then [0,0] else [p,s-p]) ps S^1_xss
      in (S^1_[xss<,xss>=], D^per_xss, isSorted D^per_xss, count+1)
  in res
```

PFP Weekly 2 Exercise: Implement `partition2`^{L}

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
“To Flatten or Not To Flatten, that is the question”

How Does One Flattens 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 Flattens 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          -- 𝓕 rule 4
                 let twom= map (+2) iot    in          -- 𝓕 rule 2
                 let rp  = replicate mm1 p in          -- 𝓕 rule 3
                 in  map (\(j,p) -> j*p) (zip twom rp) -- 𝓕 rule 2
             ) sqrt_primes
not_primes  = reduce (++) [] nested        -- ignore, already flat
```

Flattening PrimeOpt was part of PMPH's Weekly Assignment 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
- “To Flatten or Not To Flatten, that is the question”

The Good: QuickHull and the Like

Flattening is perhaps the only way to go for achieving decent GPU performance for a set of challenging problems such as Quickhull:

	Circle			Rectangle			Quadratic		
	CPU		GPU	CPU		GPU	CPU		GPU
	1C	32C		1C	32C		1C	32C	
Baseline	4.42	0.20	–	3.36	0.11	–	35.1	2.92	–
Accelerate	7.39	1.57	0.160	3.60	1.175	0.114	48.4	12.5	4.28
APL	22.2	–	1.22	14.9	–	0.690	113	–	7.57
DaCe	–	–	–	–	–	–	–	–	–
Futhark	5.56	1.28	0.064	3.81	1.151	0.047	37.6	4.03	0.68
SaC	13.3			13.2			18.3		

The languages that do not support scan or parallel write as primitives (DaCe and SAC) could not express it.

Sparse Matrix Vector Multiplication: Flattened Kernel

Dense-Matrix ($A \in \mathbb{R}^{m\times q}$) - Vector ($V \in \mathbb{R}^{q}$) Multiplication:

$$X_i = \sum_{j=0\ldots q-1} A_{i,j} \times V_j$$

Sparse-matrix uses a CSR representation:

- *B* vector records the start of each row
- *A* flat array that tuples each non-zero element with its corresponding column index
- For simplicity we assume that all rows are non empty

$$X_i = \sum_{j=B[i]\ldots B[i+1]} A_j.value \times V_{A_j.colidx}$$

Using the flattening rule for reduce nested inside of map results in:

```
def spMatVecMulFlatS [m][flen][q] (B: [m]u32, spmat: [flen](u32,f32))
                                  (vec: [q]f32) : [m]f32 = #[unsafe]
  let flags = scatter (replicate flen false) (map i64.u32 B)
                      (replicate m true)
  let scn_mat = map (\ (ind,elm) -> elm * vec[i64.u32 ind]) spmat
             |> sgmScan (+) 0f32 flags prods
  in  tabulate m
        (\i -> if i == m-1 then last scn_mat
               else scn_mat[i64.u32 (B[i+1]-1)] )
```

Sparse Matrix Vector Multiplication: Other Heuristics

Sparse-matrix Vector Multiplication: $X_i = \sum_{j=B[i]\ldots B[i+1]} A_j.value \times V_{A_j.colidx}$

A kernel that exploit only the outer parallelism, i.e., each thread process a matrix row:

```
def spMatVecMulOuter [m][flen][q] (B: [m]u32, spmat: [flen](u32,f32))
                                   (vec: [q]f32) : [m]f32 = #[unsafe]
  let f i = let beg = i64.u32 B[i]
            let end = if i == m-1 then flen else i64.u32 B[i+1]
            in  loop sum = 0 for i < end - beg do
                    let (j,v) = spmat[i + beg]
                    in  sum + v*vec[i64.u32 j]
  in map f (iota m)
```

Sparse Matrix Vector Multiplication: Other Heuristics

Sparse-matrix Vector Multiplication: $X_i = \sum_{j=B[i]\ldots B[i+1]} A_j.value \times V_{A_j.colidx}$

A kernel that exploit only the outer parallelism, i.e., each thread process a matrix row:

```
def spMatVecMulOuter [m][flen][q] (B: [m]u32, spmat: [flen](u32,f32))
                                  (vec: [q]f32) : [m]f32 = #[unsafe]
  let f i = let beg = i64.u32 B[i]
            let end = if i == m-1 then flen else i64.u32 B[i+1]
            in  loop sum = 0 for i < end - beg do
                  let (j,v) = spmat[i + beg]
                  in  sum + v*vec[i64.u32 j]
  in map f (iota m)
```

A kernel that utilizes $m \cdot 64$ parallelism: each CUDA block of 64 threads processes a row:

```
def spMatVecMulMidpt [m][flen][q] (B: [m]u32, spmat: [flen](u32,f32))
                                  (vec: [q]f32) : [m]f32 = #[unsafe]
  let block = 64i64 -- should use a better heursitic based on B
  let f i = let beg = i64.u32 B[i]
            let end = if i == m-1 then flen else i64.u32 B[i+1]
            let g tid = loop sum=0 for i < (end-beg+block-1)/block do
                           let ind = beg + i*block + tid in
                           if ind >= end then sum
                           else sum + spmat[ind].1 *
                                      vec[i64.u32 spmat[ind].0]
            in iota block |> map g |> reduce (+) 0f32 sums
  in #[incremental_flattening(only_intra)] map f (iota m)
```

PERFORMANCE DEMONSTRATION

PERFORMANCE

DEMONSTRATION

The midpoint regular kernel commonly offers best performance, i.e., the one processing a row in a CUDA block of 64 threads!

Sparse-Matrix Dense-Matrix Multiplication: Kernels

Dense Matrix ($A \in \mathbb{R}^{m\times q}$) - Matrix ($B \in \mathbb{R}^{q\times n}$) & Sparse-Dense Matrix Multiplication:

$$X_{i,j} = \sum_{k=0\ldots q-1} A_{i,k} \times B_{k,j} \qquad\qquad X_{i,j} = \sum_{k=B[i]\ldots B[i+1]} A_k.value \times B_{A_k.colidx,j}$$

Dense-Dense Matrix Multiplication follows the classical implementation:

```
def denseMMM [m][n][q] (ass: [m][q]f32) (bss: [q][n]f32) : [m][n]f32=
  let dotprod xs ys = map2 (*) xs ys |> reduce (+) 0
  in  map (\as -> map (dotprod as) (transpose bss)) ass
```

Sparse-Dense utilizing the kernel obtained by flattening:

```
def spMMFlatS [m][flen][q][n] (B: [m]u32) (spmat: [flen](u32,f32)
                                (dense: [q][n]f32) : [m][n]f32 =
  transpose dense |> map (spMatVecMulFlatS (B,spmat)) |> transpose
```

Sparse-Dense utilizing the kernel exploiting the outer parallelism:

```
def spMMOuter [m][flen][q][n] (B: [m]u32) (spmat: [flen](u32,f32)
                                (dense: [q][n]f32) : [m][n]f32 =
  transpose dense |> map (spMatVecMulOuter (B,spmat)) |> transpose
```

Sparse-Dense utilizing the kernel exploiting midpoint parallelism:

```
def spMMOuter [m][flen][q][n] (B: [m]u32) (spmat: [flen](u32,f32)
                                (dense: [q][n]f32) : [m][n]f32 =
  transpose dense |> map (spMatVecMulMidpt (B,spmat)) |> transpose
```

PERFORMANCE DEMONSTRATION

PERFORMANCE DEMONSTRATION

- Flattened version performs the worst but it is useful to protect against degenerate cases, e.g., few rows (tiny m & n) and a huge common dimension q.
- Futhark runs dense-dense at 13 out of 19 Tflops of the A100
- At sparsity factor $64\times$ dense becomes better
- At sparsity factor $128\times$ dense is still better than flattened
- Cublas + tensor cores will increase the threshold to (tens of) thousands, i.e., use sparse when one in thousands of elements is non zero