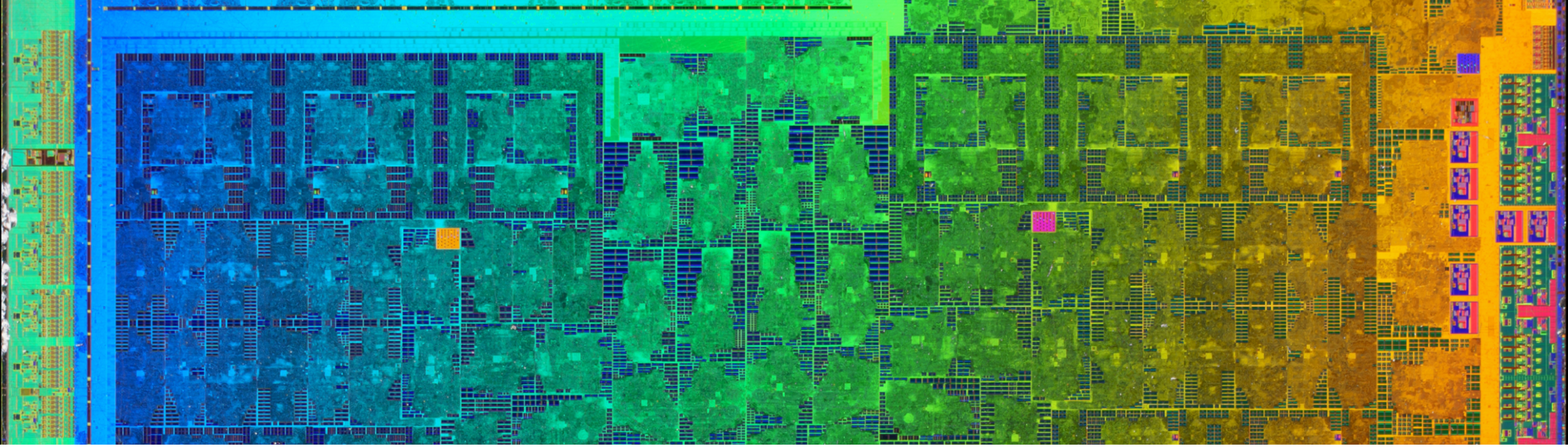
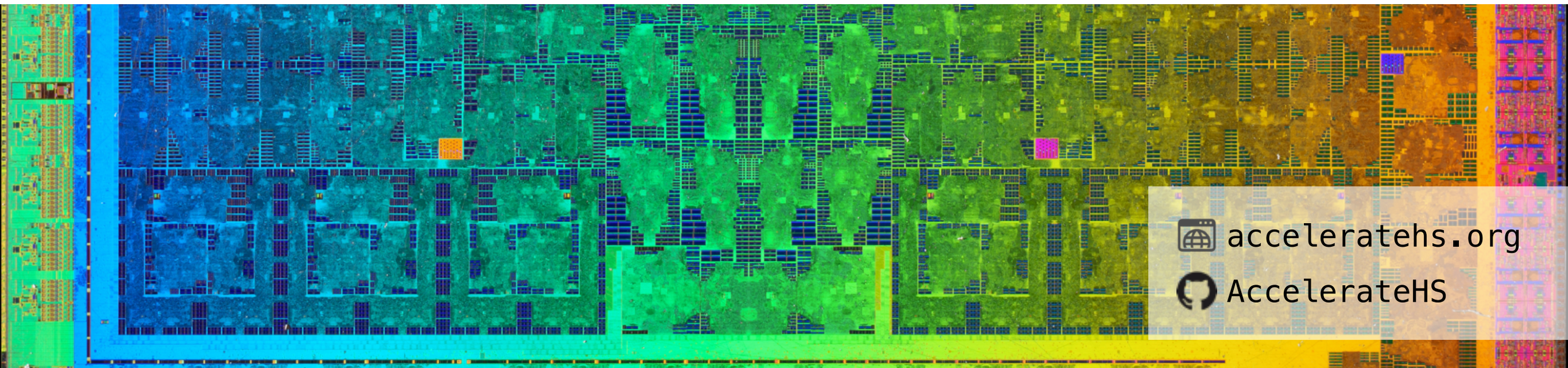


Data Parallel Programming III

Accelerate

high performance computing in Haskell

Gabriele Keller, Utrecht University



Overview

- Programming in Accelerate
- Implementation aspects of Accelerate
- Implementing an irregular parallel algorithm in Accelerate

Accelerate

- a spin-off from Data-Parallel Haskell
- similar to Futhark in many respects
  - restricts the computations to (mostly) regular nested computations
  - occupies a different place in the design space:
    - somewhat more abstract
    - is deeply embedded in Haskell
      - not a stand-alone language, but a Haskell library with the look and feel of one
    - our research is both about high-performance computing and the implementation of embedded languages

shallow embedding

fixed set of data types, operations

each operation can be highly efficient

no domain specific inter-function optimisations

host-language compiler in charge of optimisations

deep embedding

‘language in a language’

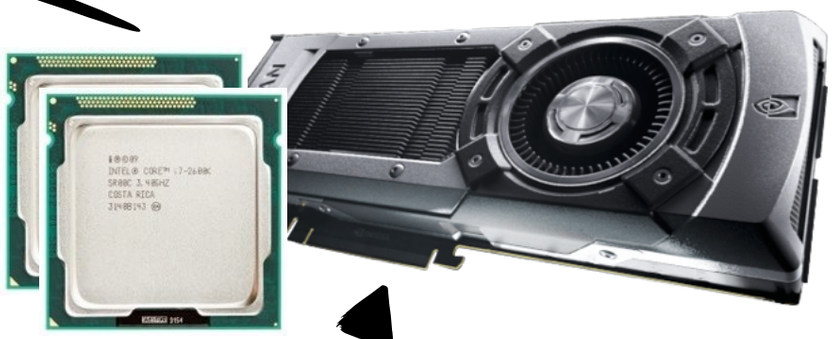
abstract syntax tree of the computation available at run time

EDSL implementation in charge of code generation, optimisation

Accelerate

An deeply embedded language for data-parallel arrays

Copy result back to Haskell



Haskell/Accelerate program

Reify and optimise Accelerate program

Target code

Compile and run on the CPU/GPU (LLVM)

currently just for Nvidia GPUs

Example: dot product

Futhark:

```
let dotp (x: []i32) (y: []i32): i32 =
  reduce (+) 0 (map2 (*) x y)
```

Example: dot product

```
import Prelude

dotp :: Num a
     => [a] -> [a] -> a
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

Example: dot product

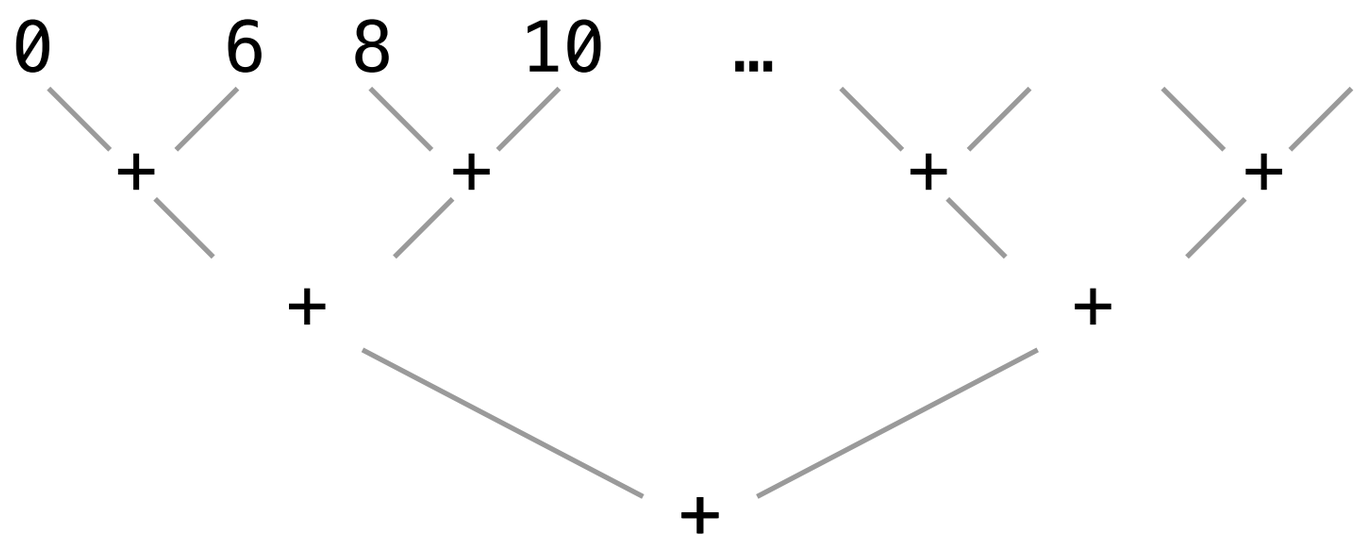
```
import Data.Vector.Unboxed

dotp :: (Unbox a, Num a)
     => Vector a
     -> Vector a
     -> a
dotp xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

Accelerate

Collective array operations compile to parallel code

```
import Data.Array.Accelerate
dotp :: (Elt a, Num a)
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
```

```
dotp xs ys = fold  (+) 0 (zipWith (*) xs ys)
```



Accelerate

Collective operations which compile to parallel code

```
import Data.Array.Accelerate
dotp :: (Elt a, Num a)
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
```

```
dotp xs ys = fold  (+) 0 (zipWith (*) xs ys)
```

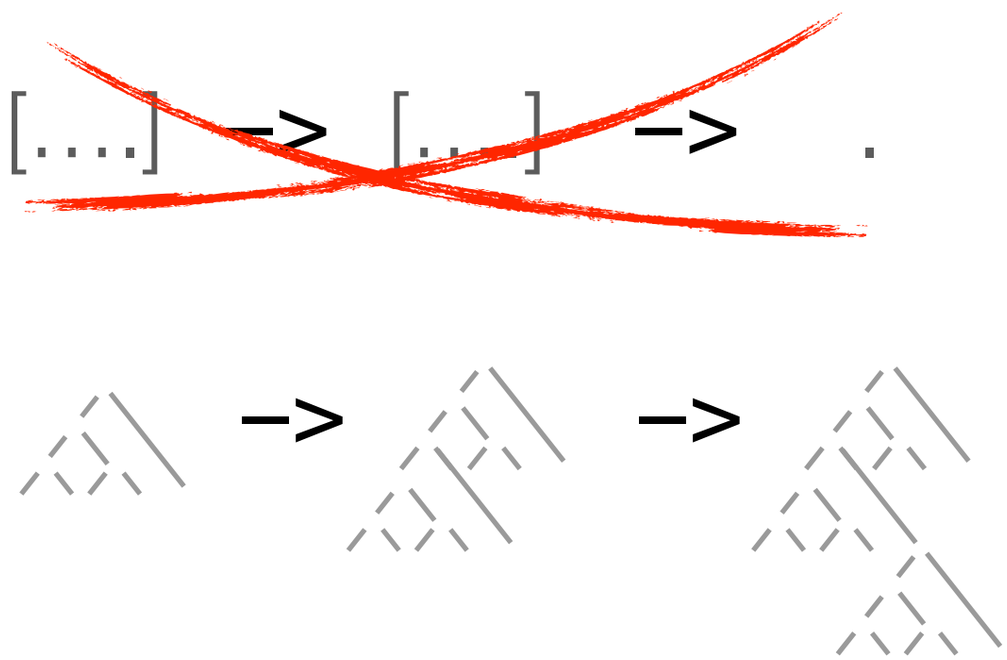
Accelerate

Collective operations which compile to parallel code

```
import Data.Array.Accelerate
dotp :: (Elt a, Num a)
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
```

language of collective, parallel operations

```
dotp xs ys = fold  (+) 0 (zipWith (*) xs ys)
```



Accelerate

Collective operations which compile to parallel code

```
import Data.Array.Accelerate
dotp :: (Elt a, Num a)
     => Acc (Vector a)
     -> Acc (Vector a)
     -> Acc (Scalar a)
dotp xs ys = fold  (+) 0 (zipWith (*) xs ys)
```

Accelerate

Collective operations which compile to parallel code

language of sequential, scalar expressions

```
fold (+) 0
```

```
fold :: (Shape sh, Elt e)
     => (Exp e -> Exp e -> Exp e)
     -> Exp e
     -> Acc (Array (sh :. Int) e)
     -> Acc (Array sh e)
```

language of collective, parallel operations

rank-polymorphic

To enforce hardware restrictions,
general irregular nested parallel computation can't be expressed

almost

```
import Data.Array.Accelerate
dotp :: (Elt a, Num a, Shape sh)
     => Acc (Array (sh :. Int) a)
     -> Acc (Array (sh :. Int) a)
     -> Acc (Array sh a)
```

```
dotp xs ys = fold  (+) 0 (zipWith (*) xs ys)
```

$$\simeq$$

```
dotp ∪ (map dotp) ∪ (map (map dotp)) ∪ …
```

Accelerate

Collective operations which compile to parallel code

```
fold :: (Shape sh, Elt e)
     => (Exp e -> Exp e -> Exp e)
     -> Exp e
     -> Acc (Array (sh :. Int) e)
     -> Acc (Array sh e)
```

shape `sh` of the form `Z :. Int :. Int :. …`

```
type DIM0      = Z
type Scalar a = Array DIM0 a


type DIM1      = DIM0 :. Int
type Vector a = Array DIM1 a
```

Accelerate

a =

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49

```
shape a  = (Z :. 5 :. 10)

arr ! (Z :. 2 :. 3) = 23

slice  arr (Z :. 2 :. All) = [20 21 22 23 24 25 26 27 28 29]
```

- Accelerate supports the usual collection oriented operations:
  - maps/zipWith, various flavours of scan and folds,
  - indexing, slicing, permutations
  - segmented operations
  - parallel ‘forall’:

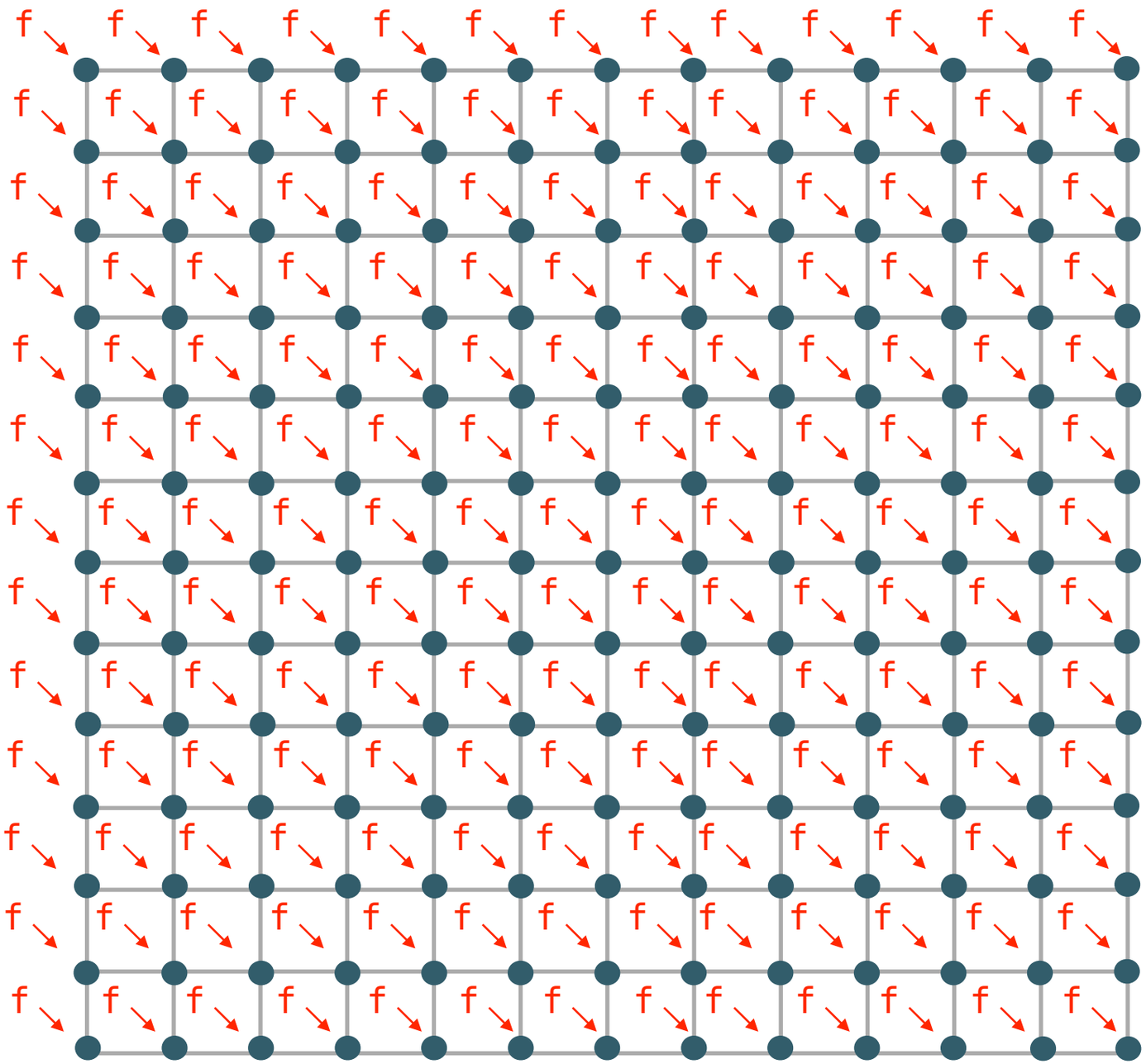
```
generate :: (Shape sh, Elt a)
         => Exp sh -> (Exp sh -> Exp a)
         -> Acc (Array sh a)
```

- while loops and iteration on element and array level

```
while:: Elt a
         =>(Exp a -> Exp Bool)
         ->(Exp a -> Exp a)
         -> Exp a
```

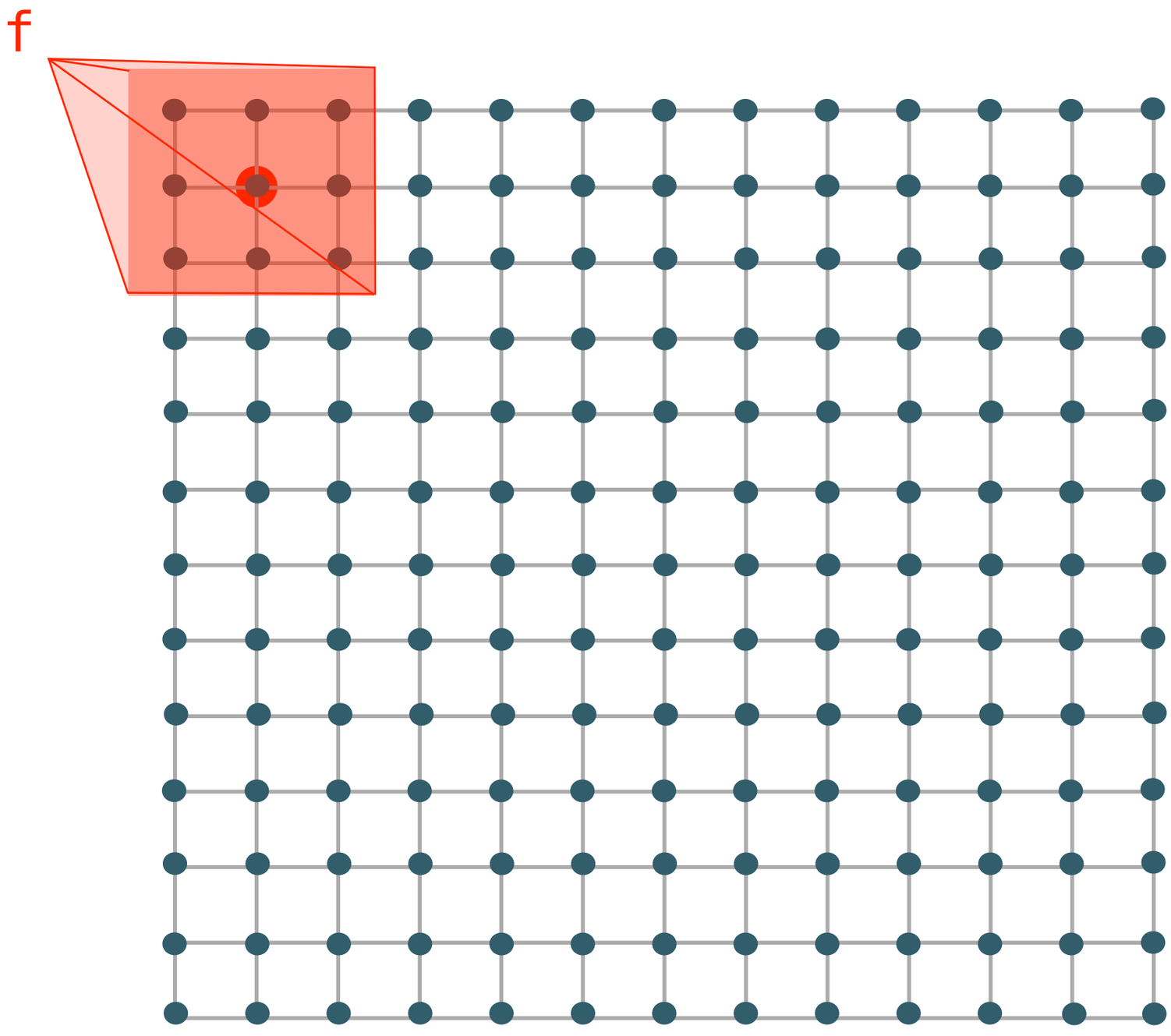
Stencil (convolution matrix) computations

```
map :: (Shape sh, Elt a, Elt b)
    => (Exp a -> Exp b)  -> Acc (Array sh a)
    -> Acc (Array sh a)
```

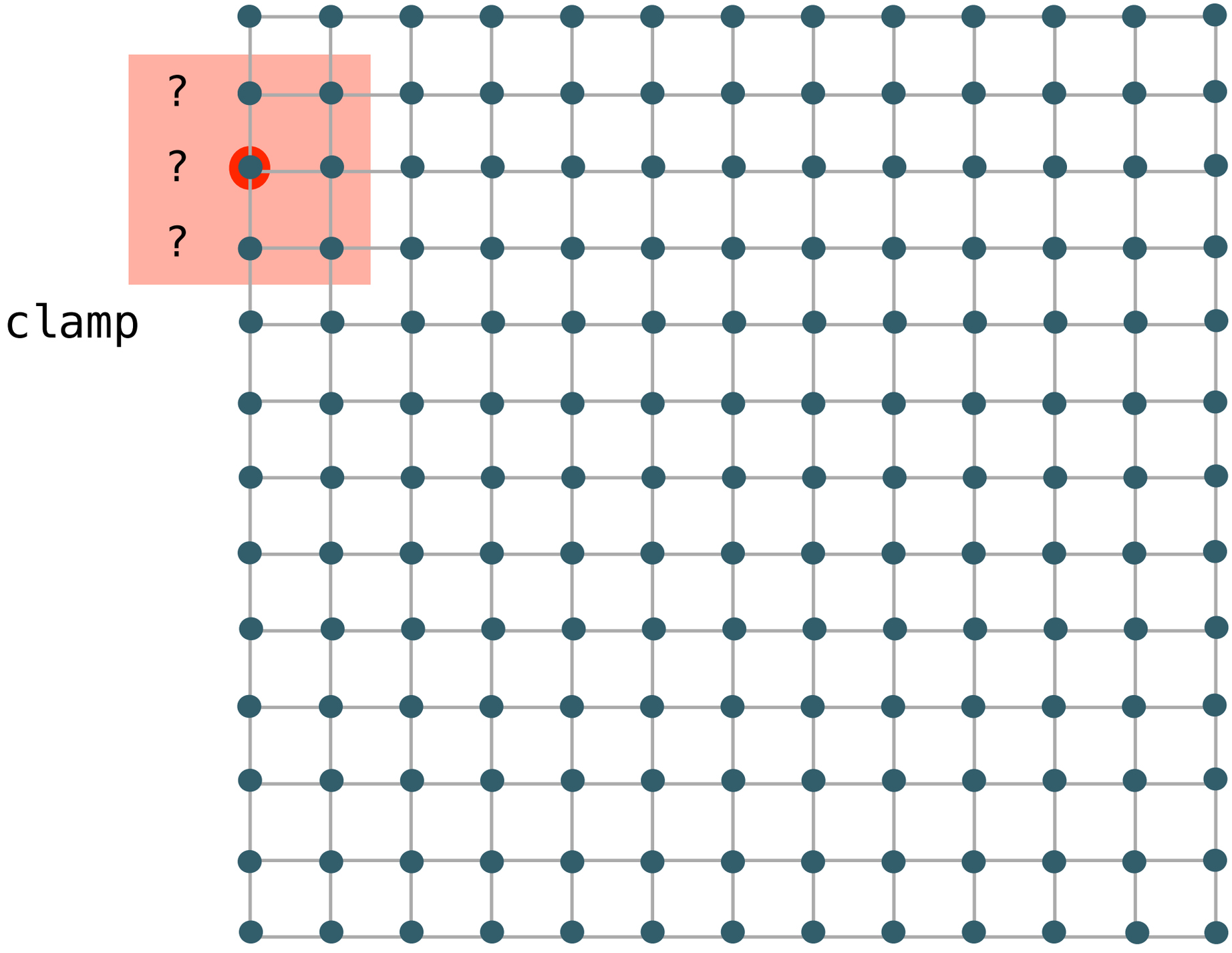


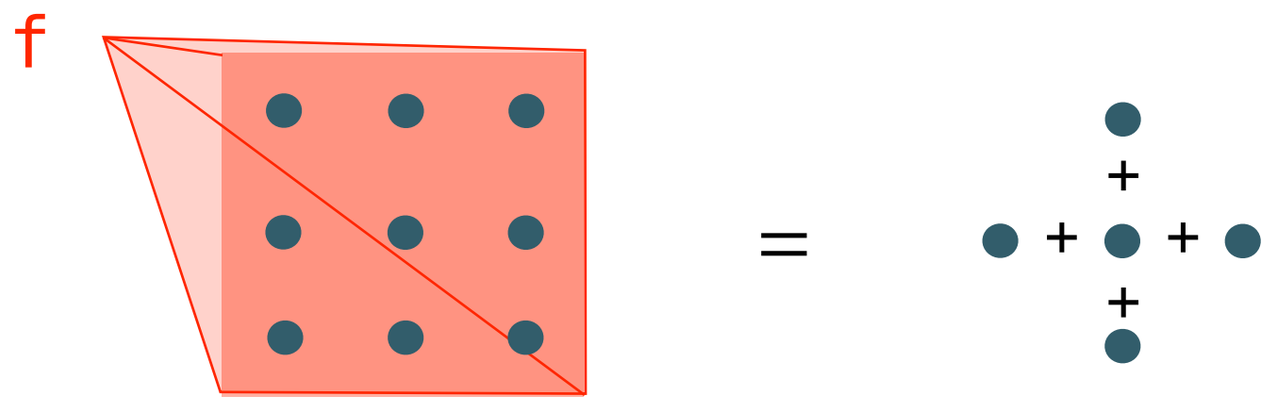
Stencil (convolution matrix) computations

```
stencil :: (Stencil sh a stencil, Elt b)
        => (stencil -> Exp b) -> Boundary (Array sh a) -> Acc (Array sh a)
        -> Acc (Array sh b)
```



Stencil computations - boundaries



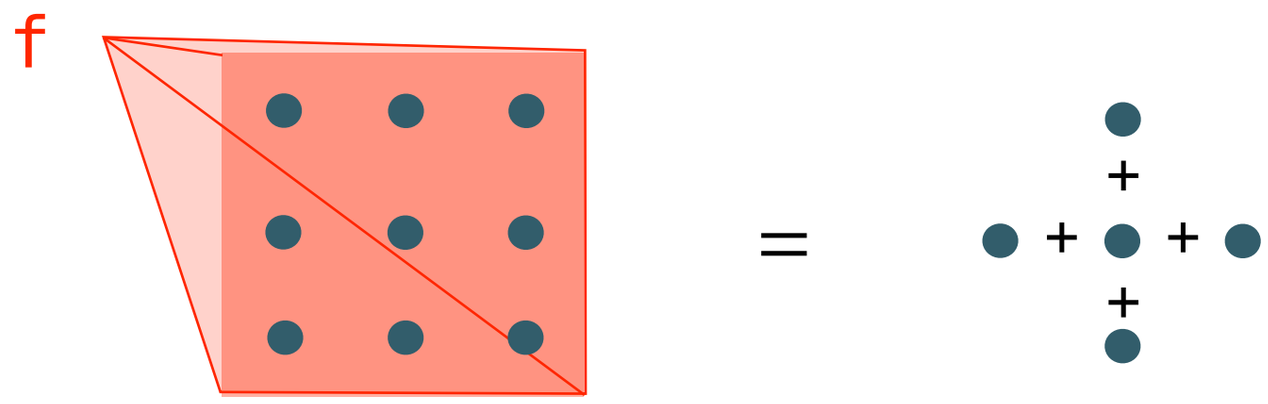


OpenCL

```
__kernel void simulate (__global float* arr
                        ,__global float* new_arr)
{
 const size_t cur = get_global_id(0);
 const size_t row = (size_t)cur/(size_t)Width;
 const size_t col = (size_t)cur%(size_t)Height;

 if (  row > 0 && row    < height-1
    && col > 0 && col < width-1) {
    new_arr[curr] =
      arr[cur] + arr[row * Width + col-1] … ;
 } else if (row == 0 && col < width-1) {
     …
 } else if …
```

Accelerate

```
simulate :: Stencil3x3 Float-> Exp Float
simulate ((_,       top,   _      ),
          (left,    curr, right   ),
          (_,       bot,   _      ))
  = top + left + curr + right + bot

new_matrix
  = stencil simulate clamp matrix
```



OpenCL

```
__kernel void simulate (__global float* arr
                        ,__global float* new_arr)
{
 const size_t cur = get_global_id(0);
 const size_t row = (size_t)cur/(size_t)Width;
 const size_t col = (size_t)cur%(size_t)Height;

 if (  row > 0 && row    < height-1
    && col > 0 && col < width-1) {
    new_arr[curr] =
      arr[cur] + arr[row * Width + col-1] … ;
 } else if (row == 0 && col < width-1) {
     …
 } else if …
```

Accelerate

```
simulate :: Stencil3x3 Float-> Exp Float
simulate ((_,       top,   _      ),
          (left,    curr, right   ),
          (_,       bot,   _      ))
  = top + left + curr + right + bot

new_matrix
  = stencil simulate clamp matrix
```

Executing an Accelerate Program

```
run :: Arrays a => Acc a -> a
```

```
import Data.Array.Accelerate
import Data.Array.Accelerate.LLVM.Native – CPU
```

```
vec1, vec2  :: Acc (Array DIM1 Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)

main =
  putStrLn $ show $ run (dotp vec1 vec2)

runN f :: (Arrays a, Arrays b, ... Arrays c)
       => a -> b -> ... -> c
```

The `Elt` class

- Members of the `Elt` class contain admissible surface types for array elements:
  - `()`
  - `Int, Int32, Int64, Word, Word32, Word64 …`
  - `Float, Double`
  - `Char`
  - `Bool`
  - Array indices formed from `Z` and `(:.)`
  - Tuples of all of these, e.g. `(Bool, Int, (Float, Float))`
  - To meet hardware restrictions, there are no nested arrays in Accelerate

Lifting values into the embedded language

```
0 :: (Num a, Elt a) => Exp a
```

```
fold (+) 0 (zipWith (*) xs ys)
```

- we declare `Num a => Exp a` to be an instance of `Num`
- `Num` values get automatically lifted into the language
- not possible for non-overloaded types and type constructors
  - `Bool`
  - `(0, 4) ::(Num a, Num b) => (a, b)`

```
(Exp a, Exp b)

Exp (a, b)
```

```
lift :: Lift e => e -> Expr (Plain e)
```

- where `Plain` is an associated type synonym which strips all the `Expr` constructors

```
Plain (Exp Float, Exp Float) ~ Plain (Exp (Float, Exp Float))
```

Lifting and unlifting values

```
swap :: (a, b) -> (b, a)
swap (x, y) = (y, x)
```

can’t use pattern matching directly!

```
swapExp :: Exp (a, b) -> Exp (b, a)
swapExp xy = lift (sndExp xy, fstExp xy)
```

can we have an `unlift` function?

```
… => Exp a -> a
```

in general, this would mean evaluating the AST (`run`)!

Lifting and unlifting values

But it does work if the type has only one constructor:

```
Exp ()          ()
Exp (a, b)      (Exp a, Exp b)
```

```
unlift :: Unlift e => Exp (Plain e) -> e

swapExp (unlift -> (x, y)) = lift (y, x)

swapExp (unlift -> (x :: Exp a, y :: Exp b)) = lift (y, x)
```

```
• Couldn't match type ‘Plain a’ with ‘Plain a0’
  Expected type: c (Plain b, Plain a) -> c1 (Plain a, Plain b)
    Actual type: c (Plain (b0, a0)) -> c1 (Plain (a0, b0))
  NB: ‘Plain’ is a non-injective type family
  The type variable ‘a0’ is ambiguous
• When checking that the inferred type
    swapExp :: forall (c1 :: * -> *) b a (c2 :: * -> *).
               (Unlift c1 (b, a), Lift c2 (a, b)) =>
               c1 (Plain (b, a)) -> c2 (Plain (a, b))
  is as general as its inferred signature
    swapExp :: forall (c1 :: * -> *) b a (c2 :: * -> *).
               (Unlift c1 (b, a), Lift c2 (a, b)) =>
               c1 (Plain b, Plain a) -> c2 (Plain a, Plain b)

• Could not deduce (Unlift c (b0, a0))
  from the context: (Unlift c (b, a), Lift c1 (a, b))
    bound by the inferred type for ‘swapExp’:
               forall (c :: * -> *) b a (c1 :: * -> *).
               (Unlift c (b, a), Lift c1 (a, b)) =>
               c (Plain b, Plain a) -> c1 (Plain a, Plain b)
    at …. Accelerate Examples/Main.hs:32:1-42
  The type variables ‘b0’, ‘a0’ are ambiguous
• When checking that the inferred type
    swapExp :: forall (c1 :: * -> *) b a (c2 :: * -> *).
               (Unlift c1 (b, a), Lift c2 (a, b)) =>
               c1 (Plain (b, a)) -> c2 (Plain (a, b))
```

Lifting and unlifting

Haskell's bidirectional pattern synonyms can do the job for us:

```
data Calc  = BinOpApp BinOp Calc Calc
           | …

data BinOp = Plus | Times | …
```

```
pattern Add e1 e2 = BinOpApp Plus e1 e2

swapArgs :: Calc -> Calc
swapArgs (Add e1 e2) = Add e2 e1
```

Accelerate provides pattern synonyms `T2`, `T3`, `T4`,… for pairs, triples,…

```
swapExp :: Exp (a, b) -> Exp (b, a)
swapExp (T2 x y) = T2 y x
```

User defined data types - the `Elt` class

- What about user defined data types?
  - tuples support built in

```
type Point   = (Float, Float)
type Complex = (Float, Float)
```

  - we want to support product and sum types, recursive types
- Two problems need to be solved:
  - how to map them efficiently to target hardware
  - how to represent them in the AST
  - both should happen without much overhead for the user

User defined data types - the `Elt` class

- We use Haskell Generics to map arbitrary algebraic data types to uniform representation
  - we use that both for user defined and built-in types
  - for representation in the AST and for mapping to hardware
  - for product types, mapping is clear
    - (.., .., .., ..) (((), ..), ..), ..)
  - sum types are much more complicated:
    - `Exp (Either a b)` `Either(Exp a) (Exp b)`

User defined data types - the `Elt` class

```
data Point = Point Float Float
  deriving (Show, Generic, Elt, IsProduct Elt)

pattern Point_ :: Exp Float → Exp Float → Exp Point
pattern Point_ x y = Pattern (x,y)
```

```
foo :: Exp Point → Exp Point
foo (Point_ x y) = (Point_ y x)
```

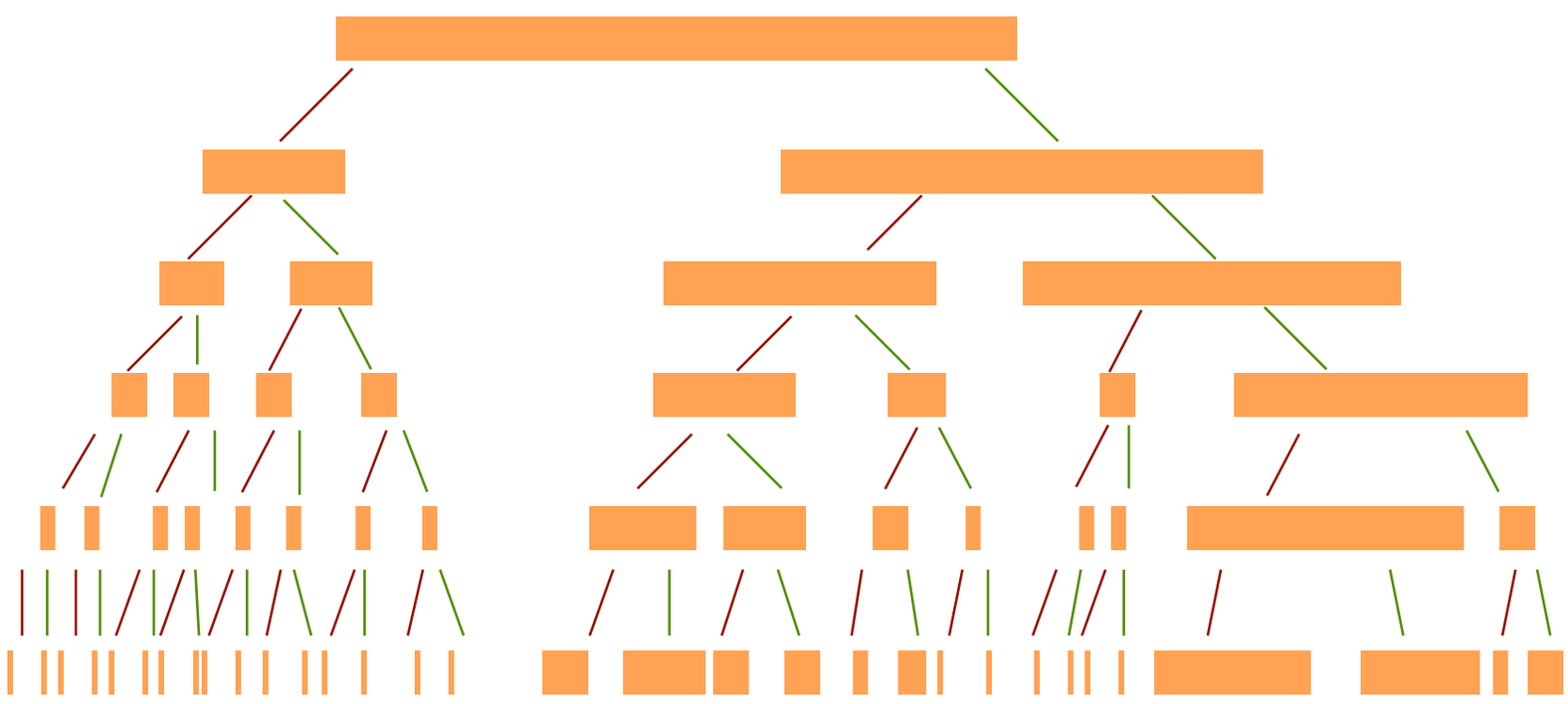
User defined data types - the `Elt` class

```
data Option a
  = None
  | Some a deriving (Generic, Elt, IsTuple)

mkPattern ''Option

simple :: Exp (Option Int) → Exp Int
simple None_     = 0
simple (Some_ x) = x
```

But now, let's have another look at how to implement an algorithm which is has nested irregular parallelism

Implementing Quicksort in Accelerate

```
qsort  []       = []
qsort (p : xs) = smaller’ ++ [p] ++ greater'
  where
    smaller = filter (< p) xs
    greater = filter (> p) xs
    [smaller’, greater’] = map qsort [smaller, greater]
```

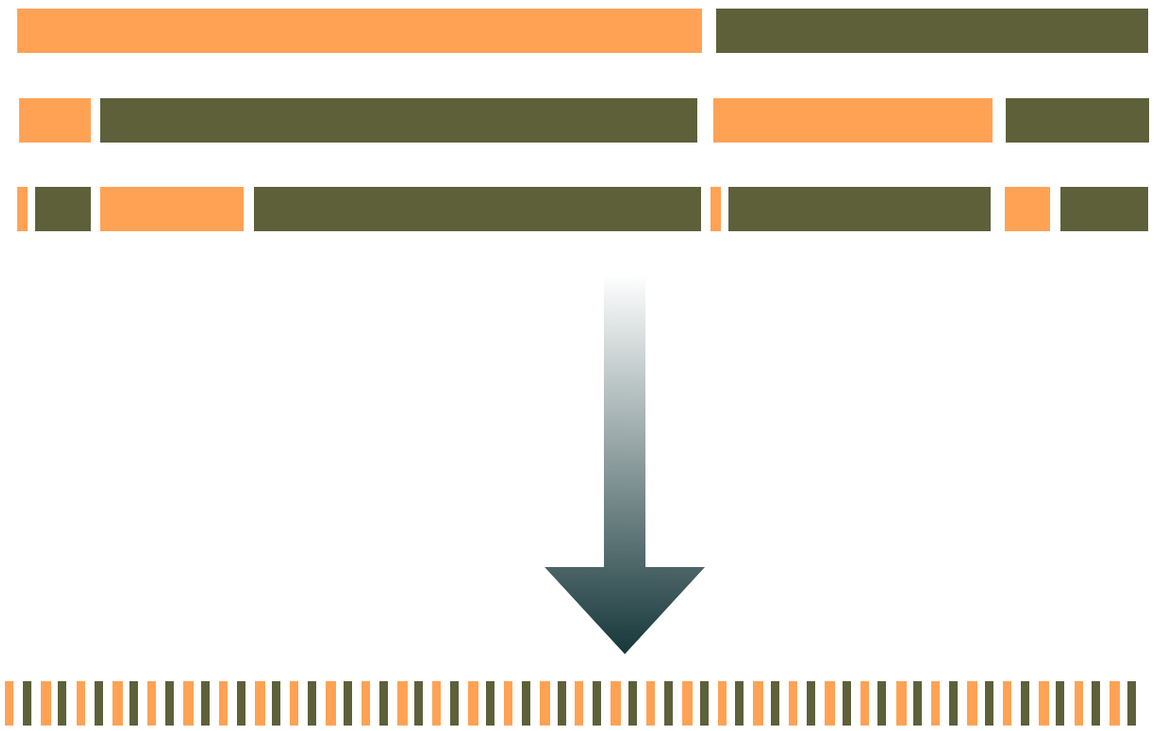


Implementing Quicksort in Accelerate

- NESL would generate
  - a vectorised version of `quicksort :: [[Float]] -> [[Float]]`
  - represent data and shape information separately

```
qsort  []     = []
qsort (p : xs) = smaller’ ++ [p] ++ greater'
  where
    smaller = filter (< p) xs
    greater = filter (> p) xs
    [smaller’, greater’] = map qsort [smaller, greater]

quicksort' :: ([Float], [Int]) -> ([Float], [Int])
quicksort’ xs seg = …
    where
      nonEmptySegs = filter (/= 0) segs
      segStarts    = preScan 0 (+) nonEmptySegs
      ps           = backpermute xs segStarts
      smallerFlags = zipWith (<) (repl’ ps seg) xs
      smallerSegs  = fold’ (+) …  smallerFlags
      …
      sortedSegs   =
         quicksort’ ((smaller, smallerSegs) ++’ (greater, greaterSegs)
```

Implementing Quicksort in Accelerate



- we have segmented ops in Accelerate , so we could implement the manually flattened quick sort
  - not trivial to get right
  - would suffer from the similar performance problems

Implementing Quicksort in Accelerate

- We can still take the inspiration from the NESL approach, but avoid inefficiencies
  - shape information can be represented using segment size:

```
[[1,2],[],[3,4],[5,6,7,8]]
```

```
[1,2,   3,4,5,6,7,8]
[2,   0,2,  4       ]
```

- ratio between segment descriptor size and data size can become arbitrarily big (many empty segments)
  - problem for quick sort

Implementing Quicksort in Accelerate

- flag vectors can be used to mark segment boundaries

```
[[1,2],[],[3,4],[5,6,7,8]]

[1,2,3,4,5,6,7,8]
[T,F,T,F,T,F,F,F]
```

- same size as data vector, can't represent empty segments
  - doesn't matter for quicksort - segment information will be discarded when sorted

Implementing Quicksort in Accelerate

- We can implement quicksort using `scans`, `map`/`zipWith`s and a permutation
  - idea: we count the number of elements greater and smaller than the pivot, in parallel for each segment
  - using this information, we calculate the index of each element in the partitioned array

Implementing Quicksort in Accelerate

```
postscanl op v [] = []
postscanl op v (x:xs)
 = v' : postscanl op v' xs
   where v' = v `op` x
```

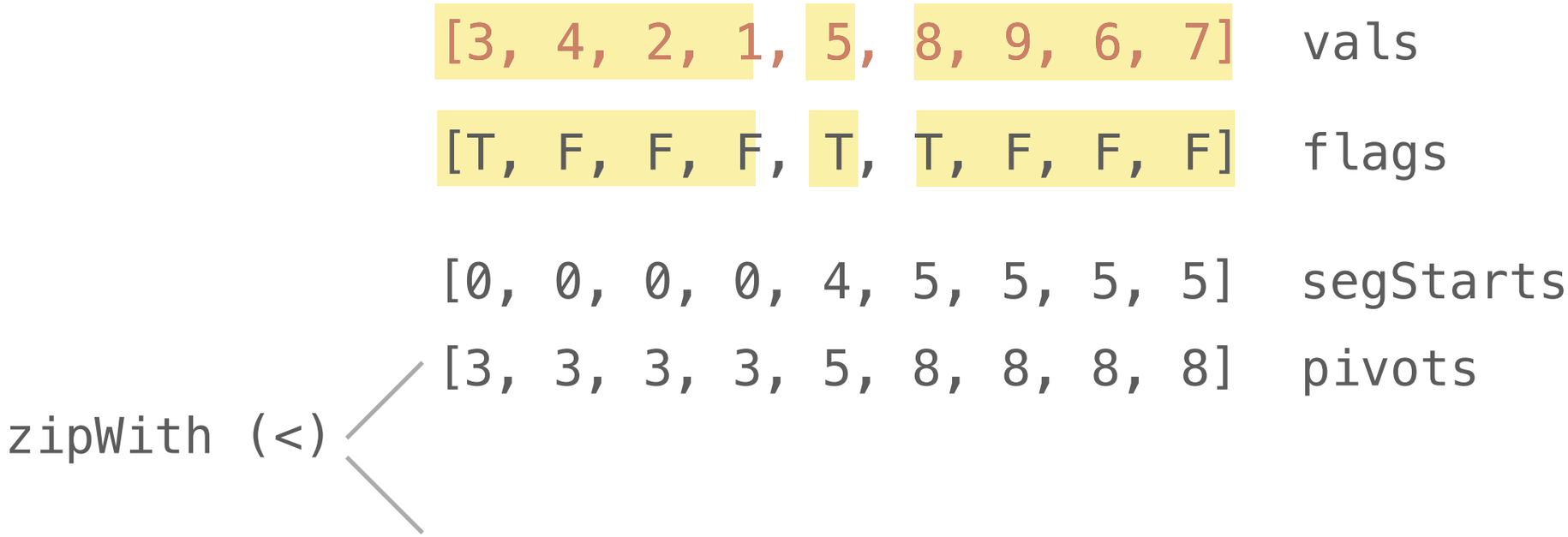
```
postscanl (+) 10 [1,2,3]
                =
      [11, 13, 16]
```

```
postScanSeg op vals flags
  = postscanl
       (\x -> \(y, flag) -> if flag
                            then y
                            else x `op` y) undefined
```

```
[3, 4, 2, 1, 5, 8, 9, 6, 7]   vals
[T, F, F, F, T, T, F, F, F]   flags
```

```
postScanSeg (+)   vals flags    = [3, 7, 9,10, 5, 8,17,23,30]
postScanSeg const vals flags    = [3, 3, 3, 3, 5, 8, 8, 8, 8]
postScanSeg const [0,1..] flags = [0, 0, 0, 0, 4, 5, 5, 5, 5]
```

```
[3, 4, 2, 1, 5, 8, 9, 6, 7]   vals
[T, F, F, F, T, T, F, F, F]   flags

[0, 0, 0, 0, 4, 5, 5, 5, 5]   segStarts
[3, 3, 3, 3, 5, 8, 8, 8, 8]   pivots
```


[3, 4, 2, 1, 5, 8, 9, 6, 7]
vals
[T, F, F, F, T, T, F, F, F]
flags
[0, 0, 0, 0, 4, 5, 5, 5, 5]
segStarts
[3, 3, 3, 3, 5, 8, 8, 8, 8]
pivots
zipWith (<)
[T, T, F, F, T, T, T, F, F]
lessOrEq

```
[3, 4, 2, 1, 5, 8, 9, 6, 7]   vals
[T, F, F, F, T, T, F, F, F]   flags

[0, 0, 0, 0, 4, 5, 5, 5, 5]   segStarts
[3, 3, 3, 3, 5, 8, 8, 8, 8]   pivots


[T, T, F, F, T, T, T, F, F]   lessOrEq
```

```
[3, 4, 2, 1, 5, 8, 9, 6, 7]   vals

[T, F, F, F, T, T, F, F, F]   flags

[0, 0, 0, 0, 4, 5, 5, 5, 5]   segStarts

[T, T, F, F, T, T, T, F, F]   lessOrEq

map (\f -> if f
             then 1
             else 0)
[1, 1, 0, 0, 1, 1, 1, 0, 0]

segmented (+)
 postscan,
 starting at -1
[0, 1, 1, 1, 0, 0, 1, 1, 1]   indexLargerEq
```

```
[3, 4, 2, 1, 5, 8, 9, 6, 7]   vals

[T, F, F, F, T, T, F, F, F]   flags

[0, 0, 0, 0, 4, 5, 5, 5, 5]   segStarts

[T, T, F, F, T, T, T, F, F]   lessOrEq
```

```
map (\f -> if f
              then 0
              else 1)
```

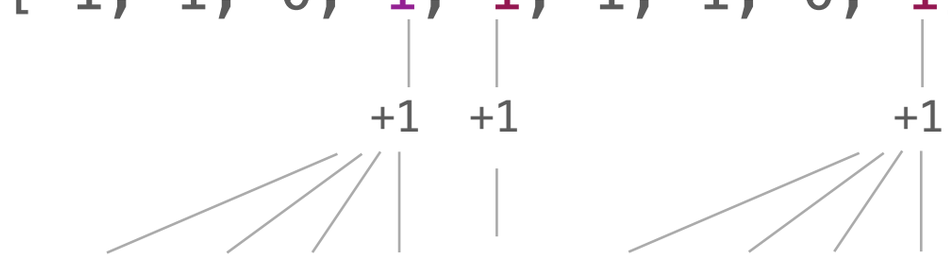
```
[1, 1, 0, 0, 1, 1, 1, 0, 0]
```

```
segmented (+)
 postscan,
 starting at -1
```

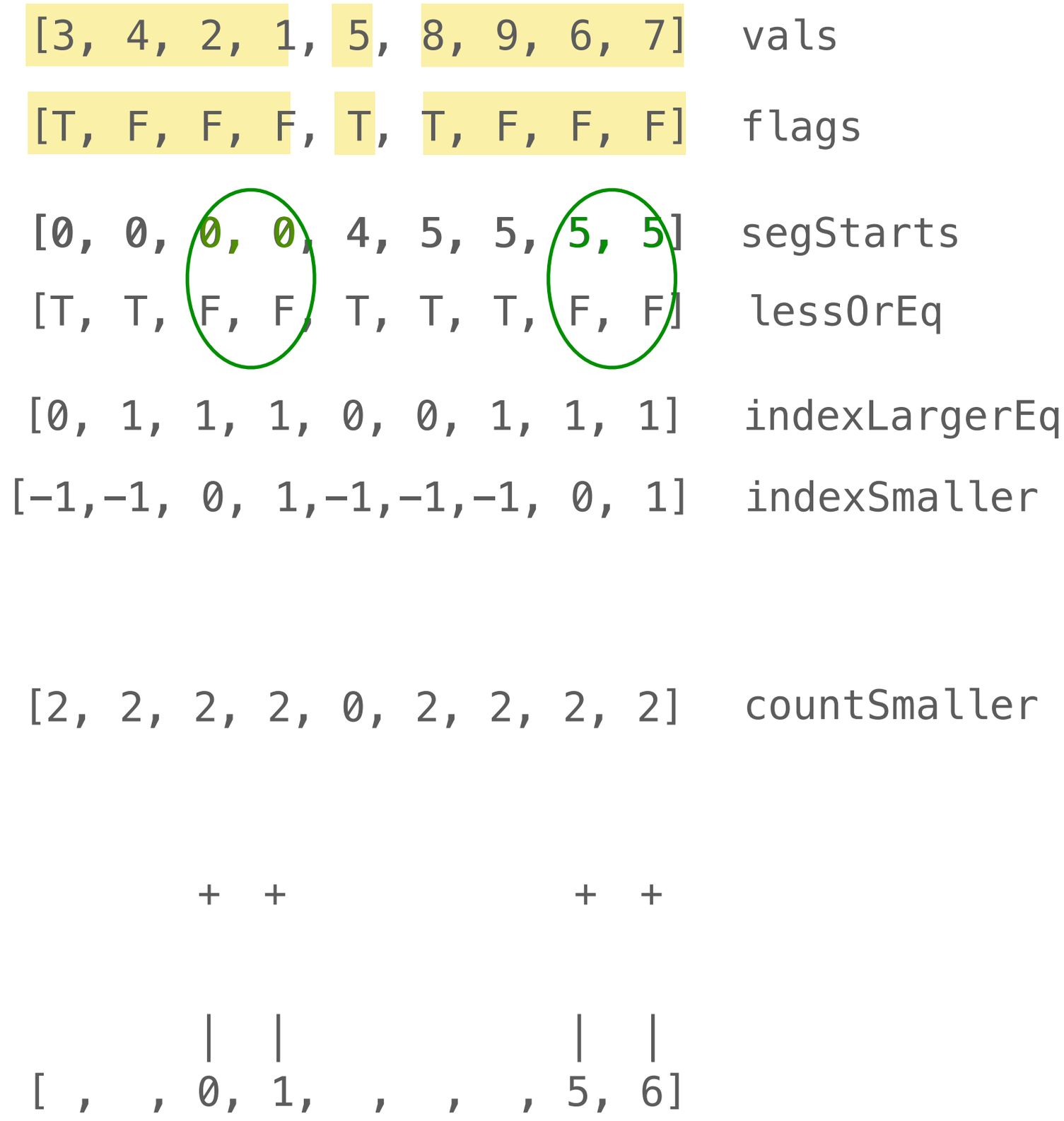
```
[-1,-1, 0, 1,-1,-1,-1, 0, 1]   indexSmaller
```

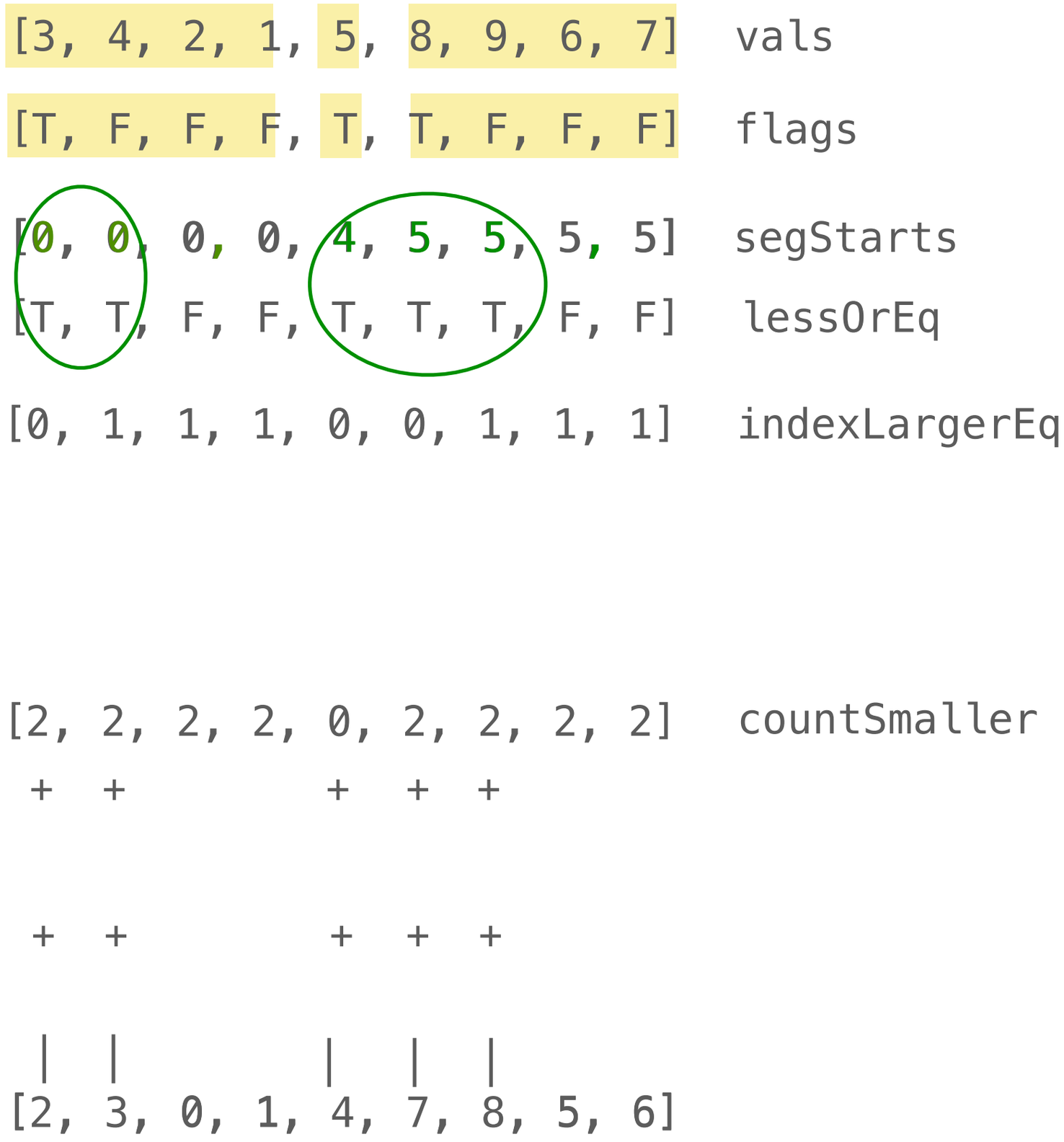
```
[3, 4, 2, 1, 5, 8, 9, 6, 7]   vals
[T, F, F, F, T, T, F, F, F]   flags

[0, 0, 0, 0, 4, 5, 5, 5, 5]   segStarts
[T, T, F, F, T, T, T, F, F]   lessOrEq

[0, 1, 1, 1, 0, 0, 1, 1, 1]   indexLargerEq
[-1,-1, 0, 1,-1,-1,-1, 0, 1]  indexSmaller
```

```
[3, 4, 2, 1, 5, 8, 9, 6, 7]   vals

[T, F, F, F, T, T, F, F, F]   flags

[0, 0, 0, 0, 4, 5, 5, 5, 5]   segStarts

[T, T, F, F, T, T, T, F, F]   lessOrEq

[0, 1, 1, 1, 0, 0, 1, 1, 1]   indexLargerEq

[-1,-1, 0, 1,-1,-1,-1, 0, 1]   indexSmaller
```




```
[2, 2, 2, 2, 0, 2, 2, 2, 2]   countSmaller
```


[3, 4, 2, 1, 5, 8, 9, 6, 7] vals
[T, F, F, F, T, T, F, F, F] flags
[0, 0, 0, 0, 4, 5, 5, 5, 5] segStarts
[T, T, F, F, T, T, T, F, F] lessOrEq
[0, 1, 1, 1, 0, 0, 1, 1, 1] indexLargerEq
[-1,-1, 0, 1,-1,-1,-1, 0, 1] indexSmaller
[2, 2, 2, 2, 0, 2, 2, 2, 2] countSmaller
[, , 0, 1, , , , 5, 6]


[3, 4, 2, 1, 5, 8, 9, 6, 7] vals
[T, F, F, F, T, T, F, F, F] flags
[0, 0, 0, 0, 4, 5, 5, 5, 5] segStarts
[T, T, F, F, T, T, T, F, F] lessOrEq
[0, 1, 1, 1, 0, 0, 1, 1, 1] indexLargerEq
[2, 2, 2, 2, 0, 2, 2, 2, 2] countSmaller
[2, 3, 0, 1, 4, 7, 8, 5, 6]

[3, 4, 2, 1, 5, 8, 9, 6, 7] vals

permute

[T, F, F, F, T, T, F, F, F] flags

[2, 1, 3, 4, 5, 6, 7, 8, 9] vals

[T, F, T, F, T, T, F, T, F] flags

[2, 3, 0, 1, 4, 7, 8, 5, 6]

Implementing Quicksort in Accelerate

- algorithm has depth (log n) on average
- implementation not Accelerate specific
  - would benefit from destructive updates in the permute (we're working on it!)
- changes and optimisations necessary to get from flattened NDP algorithm to *efficient* manually flattened are non-trivial
  - not likely to be derivable by compiler
  - more promising to identify and support common irregular patterns?

Current research directions

- Support for a limited form of irregular nested computations
  - aimed at detecting regular subcomputations
- Automatic derivation of uniqueness information
  - enables more destructive updates
- Support for more flexible schedules
- Improving fusion optimisation
  - ‘horizontal fusion/tupling’
- Other backends (AMD, Metal)
- Automatic differentiation in Accelerate