Parallel Functional Programming
Lecture 8
Data Parallelism II

Mary Sheeran

(with thanks to Ben Lippmeier for borrowed slides)

http://www.cse.chalmers.se/edu/course/pfp

Data parallelism

Perform *same* computation on a collection of *differing* data values

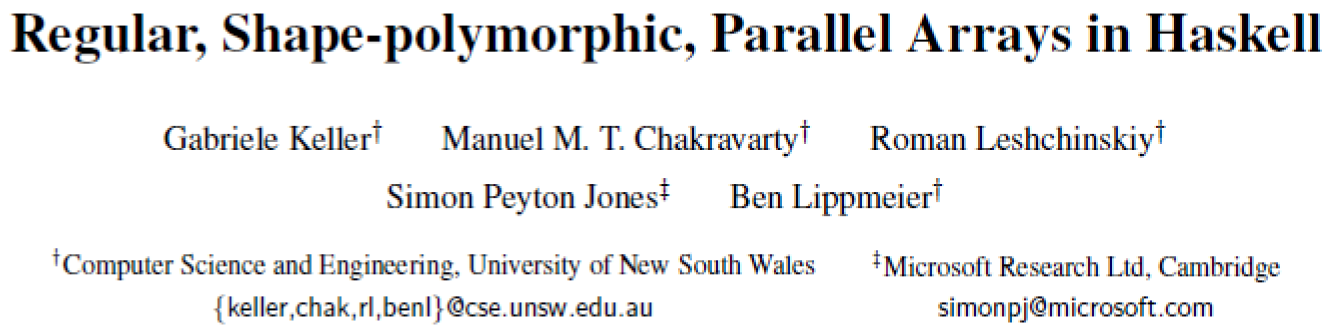
examples: HPF (High Performance Fortran)
CUDA

Both support only flat data parallelism

Flat : each of the individual computations on (array) elements is sequential

those computations don’t need to communicate

parallel computations don’t spark further parallel computations



Regular, Shape-polymorphic, Parallel Arrays in Haskell

Gabriele Keller[†] Manuel M. T. Chakravarty[†] Roman Leshchinskiy[†]
Simon Peyton Jones[‡] Ben Lippmeier[†]

[†]Computer Science and Engineering, University of New South Wales
{keller,chak,rl,benl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd, Cambridge
simonpj@microsoft.com

API for purely functional, collective operations over dense, rectangular, multi-dimensional arrays supporting shape polymorphism

ICFP 2010

Ideas

Purely functional array interface using collective (whole array) operations like map, fold and permutations can

- combine efficiency and clarity
- focus attention on structure of algorithm, away from low level details

Influenced by work on algorithmic skeletons based on Bird Meertens formalism

Provides shape polymorphism not in a standalone specialist compiler like SAC, but using the Haskell type system

terminology

Regular arrays

dense, rectangular, most elements non-zero

shape polymorphic

functions work over arrays of arbitrary dimension

terminology

Regular arrays

dense, rectan

shape polym

functions wo

n

note: the arrays are purely functional and immutable

All elements of an array are demanded at once -> parallelism

P processing elements, n array elements => n/P consecutive elements on each proc. element

version

I use Repa 2.1.1.5 (which works with the GHC that you get with the current Haskell platform)

If you have GHC 7.4 installed, you can use a later Repa, which has more array types (and doubtless better performance)

example

```
import Data.Array.Repa as A
```

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
```

example

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
```

element type
(basic numeric types, Bool, strict pairs)

example

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
```

index type
SHAPE
EXTENT

example

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
```

DIM0 = Z (scalar)
DIM1 = DIM0 :. Int
DIM2 = DIM1 :. Int

snoc lists

Haskell lists are cons lists
1:2:3:[]   is the same as [1,2,3]

Repa uses snoc lists at type level for shape types
and at value level for shapes

DIM2 = Z :. Int :. Int   is a shape type

Z :. i :. j   read as (i,j)   is an index into a two dim. array

examples

```
*Main>   let y = fromList ((Z :. 2 :. 3 :. 3) :: DIM3) [1..18]
```

shape

examples

```
*Main>   let y = fromList ((Z :. 2 :. 3 :. 3) :: DIM3) [1..18]
```

the type of the shape
needs to be there, otherwise
get very annoying error messages

examples

```
*Main>   let y = fromList ((Z :. 2 :. 3 :. 3) :: DIM3) [1..18]



*Main> y
Array (Z :. 2 :. 3 :. 3) [1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.
0,14.0,15.0,16.0,17.0,18.0]

*Main> extent y
((Z :. 2) :. 3) :. 3
```

examples

```
*Main>   let y = fromList ((Z :. 2 :. 3 :. 3) :: DIM3) [1..18]

*Main> y ! (Z :. 0 :. 0 :. 0)
1.0

*Main> y ! (Z :. 1 :. 1 :. 1)
14.0
```

examples

```
*Main>   let y = fromList ((Z :. 2 :. 3 :. 3) :: DIM3) [1..18]


*Main> y ! (Z :. 0 :. 0 :. 20)
*** Exception: .\Data\Vector\Generic.hs:237 ((!)): index out of bounds (20,18)
```

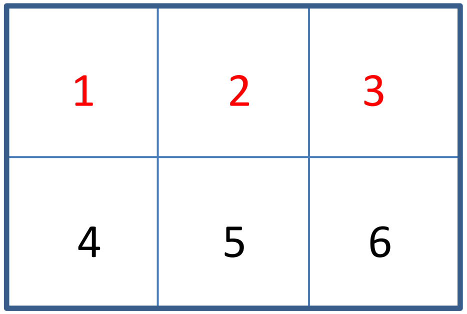
bounds checking is done at RUN TME

examples

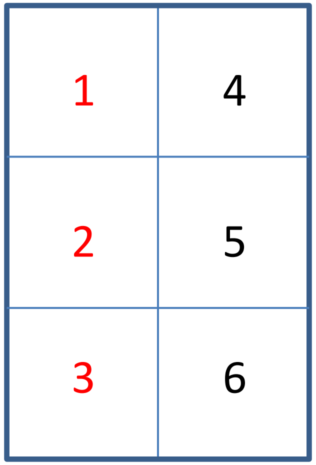
```
*Main>   let y = fromList ((Z :. 2 :. 3 :. 3) :: DIM3)
[1..18]

 *Main> y ! (Z :. 0 :. 0 :. 9)
 10.0

 *Main> y ! (Z :. 0 :. 0 :. 17)
 18.0

 *Main> y ! (Z :. 0 :. 17 :. 0)
 *** Exception:
 .\Data\Vector\Generic.hs:237 ((!)): index
 out of bounds (51,18)
```

This was unexpected!

```
*Main> let z = fromList (Z :. 2 :. 3 :: DIM2) [1..6]
*Main> transpose2D z
Array (Z :. 3 :. 2) [1.0,4.0,2.0,5.0,3.0,6.0]
```



1	2	3
4	5	6



1	4
2	5
3	6

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
transpose2D a = backpermute (swap s) swap a
   where
        s = extent a
        swap (Z :. i :. j) = Z :. j :. i
```

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
transpose2D a = backpermute (swap s) swap a
  where
     s = extent a
     swap (Z :. i :. j) = Z :. j :. i
```

s   is the shape (or extent) of the array a

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
transpose2D a = backpermute (swap s) swap a
  where
      s = extent a
      swap (Z :. i :. j) = Z :. j :. i
```

swap i and j

= swap rows and columns

an index space transformation

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
transpose2D a = backpermute (swap s) swap a
   where
        s = extent a
        swap (Z :. i :. j) = Z :. j :. i
```

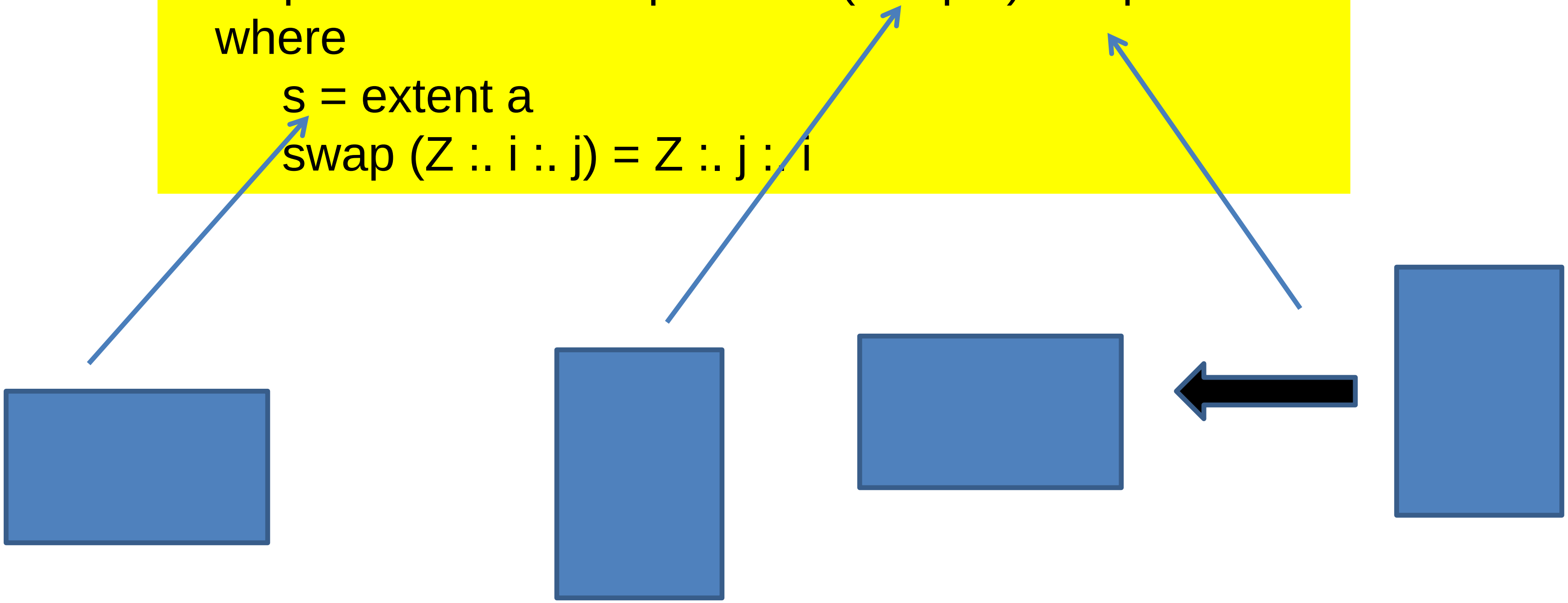
swap i and j
= swap rows and columns

an index space transformation

1	2	3
4	5	6

1	4
2	5
3	6

```
backpermute  :: (Shape shin, Shape shout, Elt a) =>
                 shout -> (shout -> shin) -> Array shin a -> Array shout a
```



```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
transpose2D a = backpermute (swap s) swap a
   where
        s = extent a
        swap (Z :. i :. j) = Z :. j :. i
```

more general transpose
(on inner two dimensions)

```
transpose :: (Shape sh, Elt e) => Array ((sh :. Int) :. Int)  e -> Array ((sh :. Int) :. Int)  e
```

more general transpose (on inner two dimensions) is provided

```
transpose :: (Shape sh, Elt e) => Array ((sh :. Int) :. Int)  e -> Array ((sh :. Int) :. Int)  e
```

This type says an array with at least 2 dimensions.
The function is shape polymorphic

more general transpose
(on inner two dimensions)
is provided

```
transpose :: (Shape sh, Elt e) => Array ((sh :. Int) :. Int)  e -> Array ((sh :. Int) :. Int)  e
```

Functions with at-least constraints become a parallel map over the unspecified dimensions (called rank generalisation)

Important way to express parallel patterns

```
*Main> let w = fromList (Z :. 2 :. 3 :. 3 :: DIM3) [1..(18 ::Int)]

*Main> A.transpose w
Array (Z :. 2 :. 3 :. 3) [1,4,7,2,5,8,3,6,9,  10,13,16,11,14,17,12,15,18]
```

```haskell
A.sum :: (Shape sh, Elt a, Num a) => Array (sh :. Int) a -> Array sh a
```

reduces shape by one dimension

```
*Main> w
Array (Z :. 2 :. 3 :. 3) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]
*Main> A.sum w
Array (Z :. 2 :. 3)         [6,   15,   24,     33,       42,      51]
```

```
*Main> w
Array (Z :. 2 :. 3 :. 3) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]
*Main> A.sum w
Array (Z :. 2 :. 3)           [6,     15,    24,          33,           42,         51]
```

note that for 1D arrays, sum is implemented as a parallel tree reduction as (+) is known to be associative.
Generic folds etc. are sequential (in this version of Repa)
For > 1D arrays, both are sequential, but many of them happen at once because of rank generalisation

backpermute can change the shape

```
selEven :: (Shape sh, Elt e) => Array (sh:.Int) e -> Array (sh:.Int) e
{-# INLINE selEven #-}
selEven !arr = force $ backpermute new_shape expand arr
  where
    (ns :.n) = extent arr
    new_shape = ns :.((n+1) `div` 2)
    expand (is :.i) = is :.(i * 2)
```

backpermute can change the shape

```
selEven :: (Shape sh, Elt e) => Array (sh:.Int) e -> Array (sh:.Int) e
{-# INLINE selEven #-}
selEven !arr = force $ backpermute new_shape expand arr
  where
    (ns :.n) = extent arr
    new_shape = ns :.((n+1) `div` 2)
    expand (is :.i) = is :.(i * 2)
```

Note how the new shape depends only on the old shape and not on the data in the array
(My def. differs slightly from that in the paper.)

backpermute can change the shape

```
selEven :: (Shape sh, Elt e) => Array (sh:.Int) e -> Array (sh:.Int) e
{-# INLINE selEven #-}
selEven !arr = force $ backpermute new_shape expand arr
  where
    (ns :.n) = extent arr
    new_shape = ns :.((n+1) `div` 2)
    expand (is :.i) = is :.(i * 2)

selOdd :: (Shape sh, Elt e) => Array (sh:.Int) e -> Array (sh:.Int) e
{-# INLINE selOdd #-}
selOdd !arr = force $ backpermute new_extent expand arr
  where
    (ns :.n) = extent arr
    new_extent = ns :.(n `div` 2)
    expand (is :.i) = is :.(i * 2  + 1)
```

```
*Main> let w = fromList (Z :. 2 :. 3 :. 3 :: DIM3) [1..(18 ::Int)]
*Main> selEven w
Array (Z :. 2 :. 3 :. 2) [1,3,4,6,7,9,10,12,13,15,16,18]
*Main> selOdd w
Array (Z :. 2 :. 3 :. 1) [2,5,8,11,14,17]
```

filter?

```
filter :: (Elt e ) => (E -> Bool) -> Array DIM1 e -> Array DIM1 e
```

can’t be shape polymorphic

the shape of the output depends on the value of the input

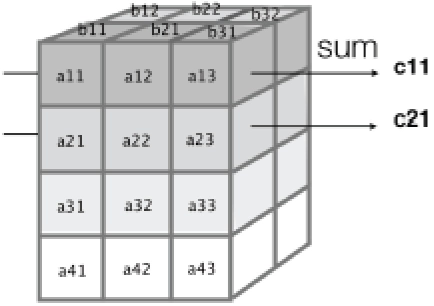
filtering rows in a matrix might give different lengths (but we only deal with rectangular arrays)

Matrix Multiplication $(A.B)_{i,j} = \Sigma_{k=1}^{n} A_{i,k} \, . \, B_{k,j}$

$$\begin{array}{|c|c|c|} \hline a_{11} & a_{12} & a_{13} \\ \hline a_{21} & a_{22} & a_{23} \\ \hline a_{31} & a_{32} & a_{33} \\ \hline a_{41} & a_{42} & a_{43} \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline b_{11} & b_{12} \\ \hline b_{21} & b_{22} \\ \hline b_{31} & b_{32} \\ \hline \end{array} = \begin{array}{|c|c|} \hline c_{11} & c_{12} \\ \hline c_{21} & c_{22} \\ \hline c_{31} & c_{32} \\ \hline c_{41} & c_{42} \\ \hline \end{array}$$

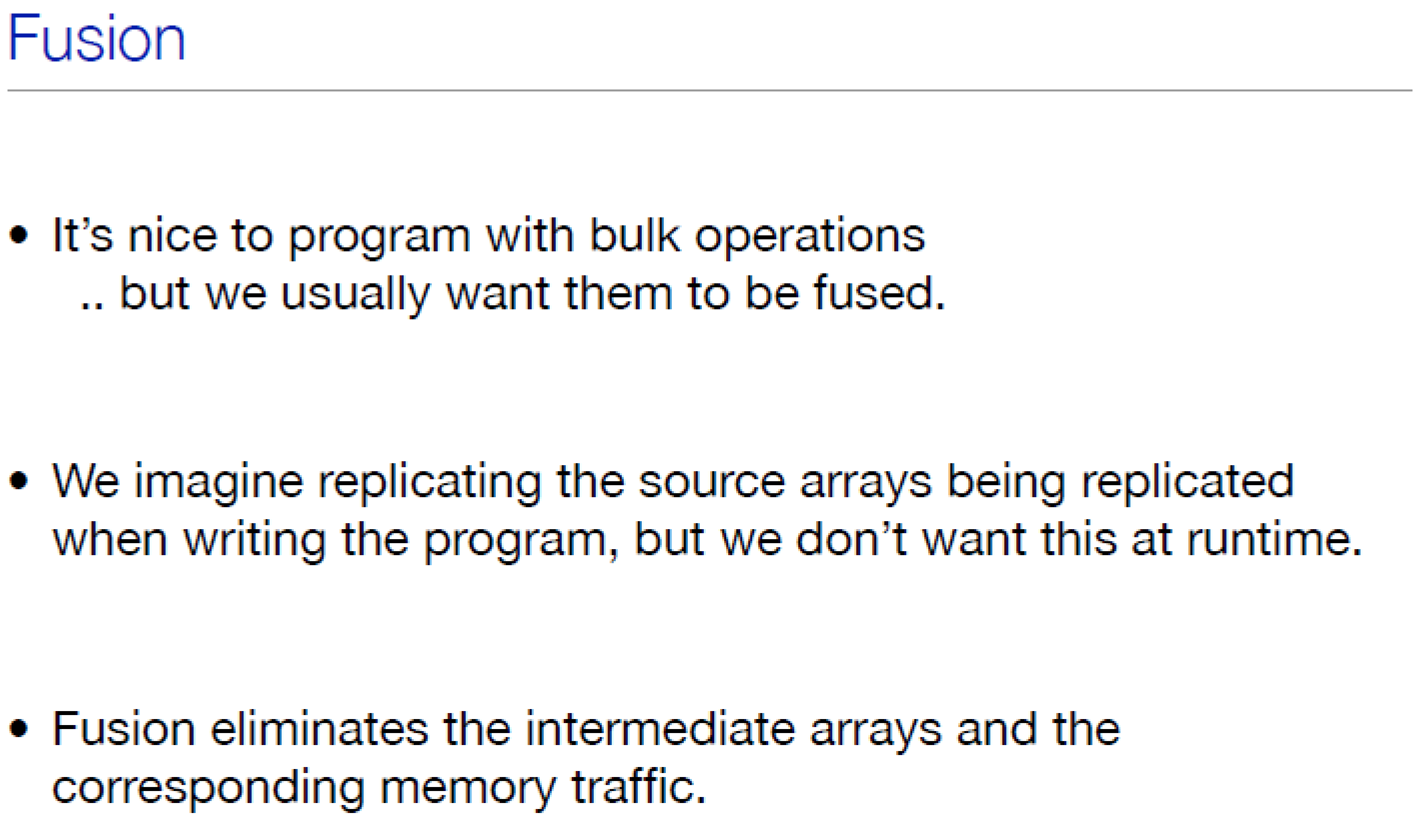
slide from Lippmeier’s ICFP 2010 talk on Repa

Matrix Multiplication



```
mmMult
  ::  (Num e, Elt e)
  =>  Array DIM2 e
  ->  Array DIM2 e -> Array DIM2 e

mmMult arr brr
 = sum (zipWith (*) arrRepl brrRepl)
 where
  trr  = transpose2D brr
  arrR = replicate (Z :.All   :.colsB :.All) arr
  brrR = replicate (Z :.rowsA :.All   :.All) trr
  (Z :. colsA :. rowsA) = extent arr
  (Z :. colsB :. rowsB) = extent brr
```

slide from Lippmeier’s ICFP 2010 talk on Repa



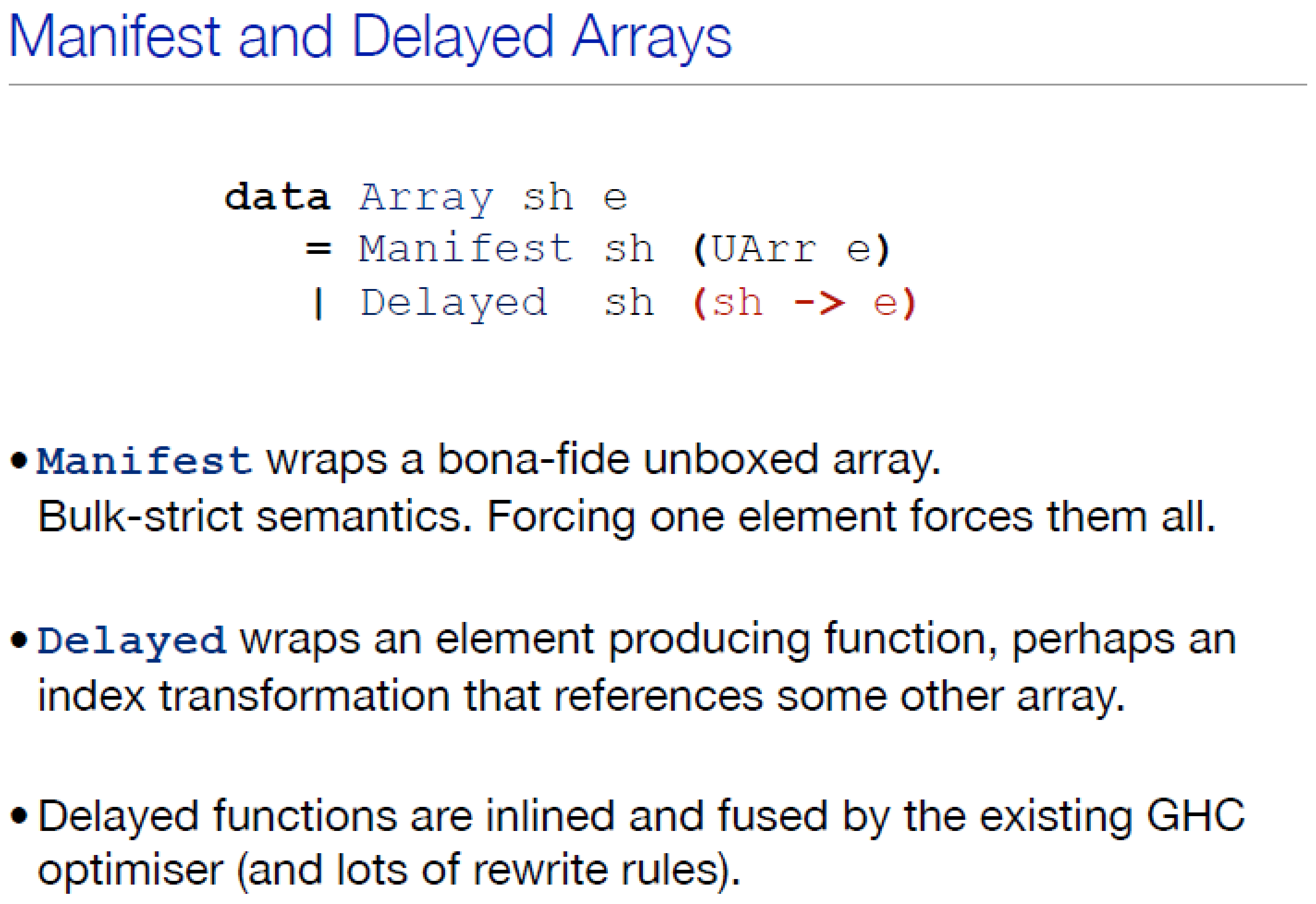
Fusion

- It’s nice to program with bulk operations
  .. but we usually want them to be fused.

- We imagine replicating the source arrays being replicated when writing the program, but we don’t want this at runtime.

- Fusion eliminates the intermediate arrays and the corresponding memory traffic.

slide from Lippmeier’s ICFP 2010 talk on Repa

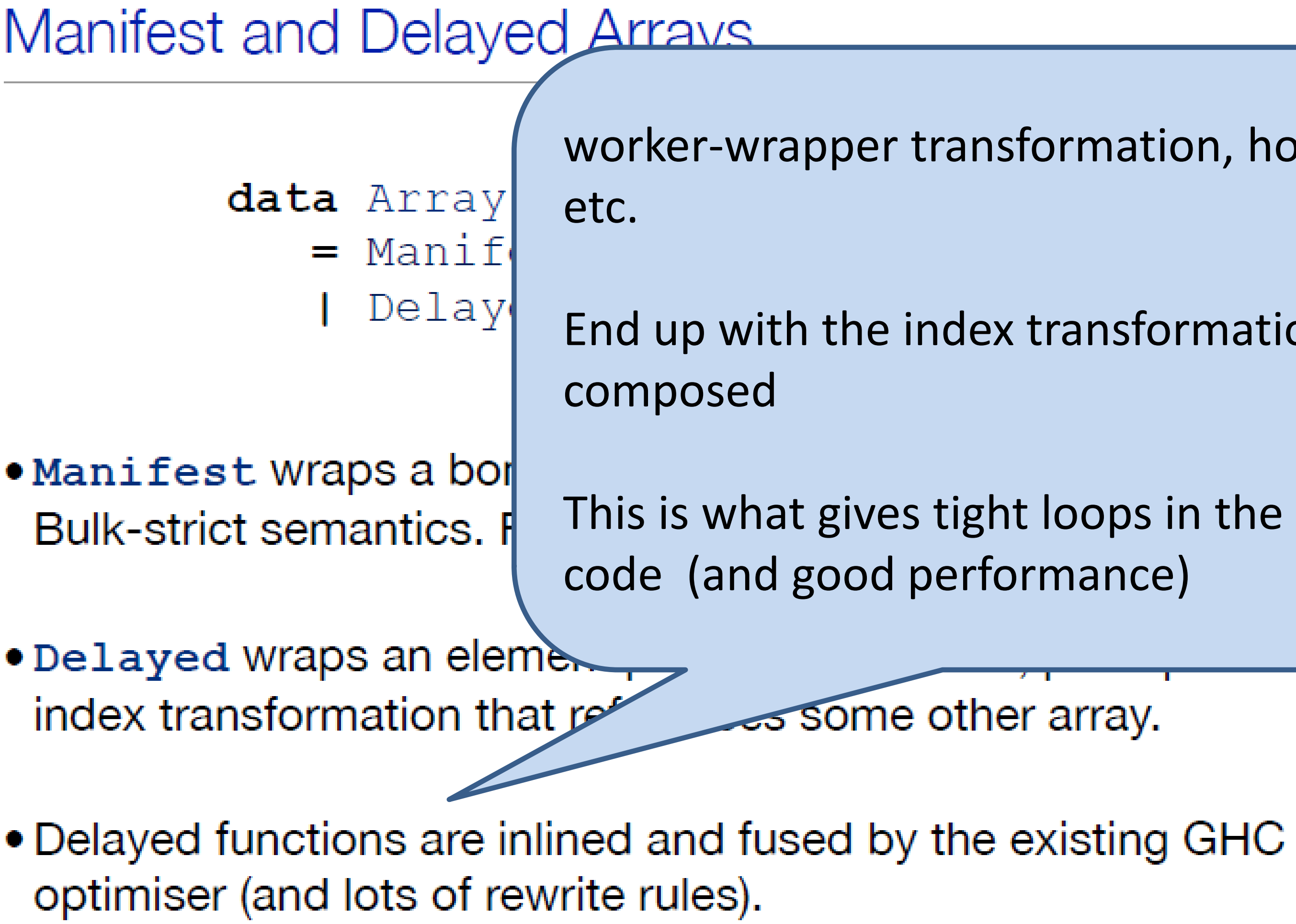


Manifest and Delayed Arrays

```
data Array sh e
   = Manifest sh (UArr e)
   | Delayed  sh (sh -> e)
```

- **`Manifest`** wraps a bona-fide unboxed array. Bulk-strict semantics. Forcing one element forces them all.
- **`Delayed`** wraps an element producing function, perhaps an index transformation that references some other array.
- Delayed functions are inlined and fused by the existing GHC optimiser (and lots of rewrite rules).

slide from Lippmeier's ICFP 2010 talk on Repa



Manifest and Delayed Arrays

```
data Array
   = Manif
   | Delay
```

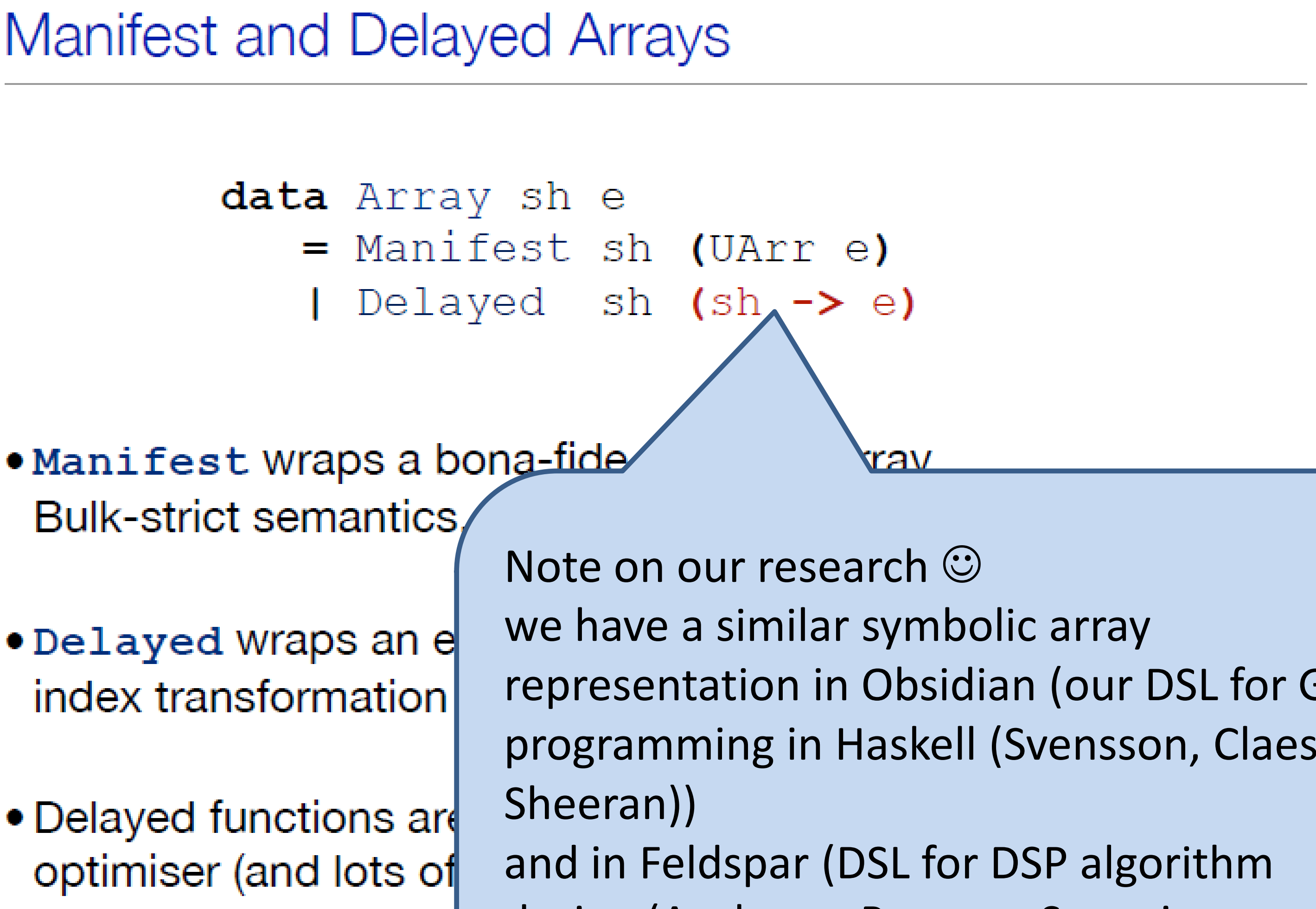
- **`Manifest`** wraps a bor
  Bulk-strict semantics. F
- **`Delayed`** wraps an eleme
  index transformation that re   some other array.
- Delayed functions are inlined and fused by the existing GHC optimiser (and lots of rewrite rules).

worker-wrapper transformation, hoisting etc.

End up with the index transformations nicely composed

This is what gives tight loops in the resulting code (and good performance)

slide from Lippmeier's ICFP 2010 talk on Repa



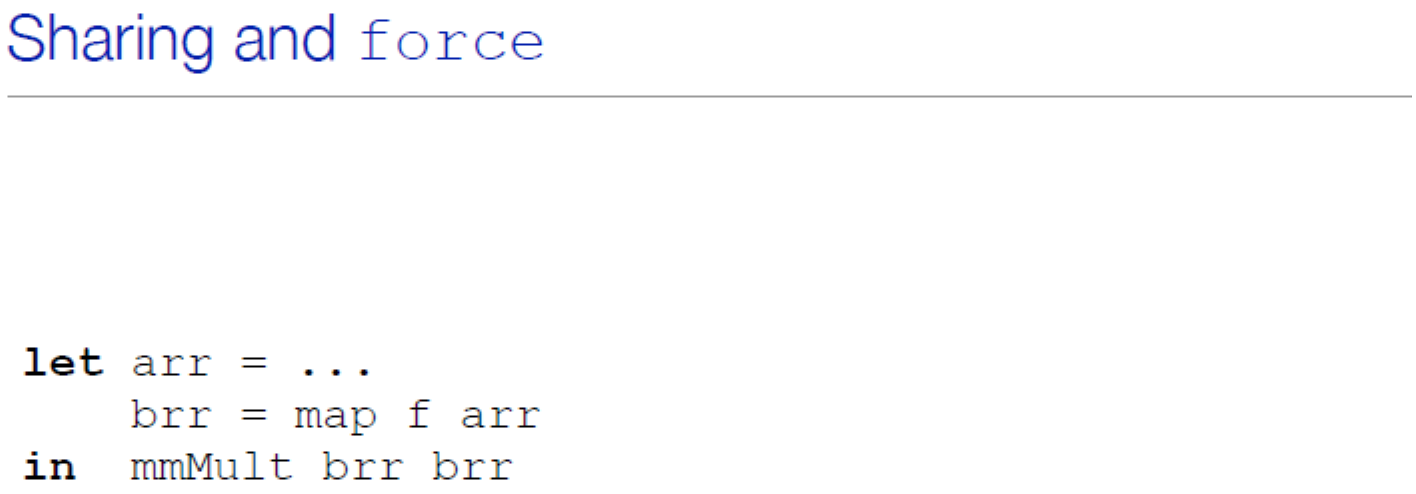
Manifest and Delayed Arrays

```
data Array sh e
    = Manifest sh (UArr e)
    | Delayed  sh (sh -> e)
```

- **`Manifest`** wraps a bona-fide rray
  Bulk-strict semantics.
- **`Delayed`** wraps an e
  index transformation
- Delayed functions are
  optimiser (and lots of

Note on our research ☺
we have a similar symbolic array representation in Obsidian (our DSL for GPU programming in Haskell (Svensson, Claessen, Sheeran))
and in Feldspar (DSL for DSP algorithm design (Axelsson, Persson, Svenningsson, Sheeran))

slide from Lippmeier's ICFP 2010 talk on Repa



Sharing and `force`

```
let arr = ...
    brr = map f arr
in  mmMult brr brr
```

slide from Lippmeier’s ICFP 2010 talk on Repa

Sharing and `force`

```
data Array sh e
 = Manifest sh (UArr e)
 | Delayed  sh (sh -> e)
```

```
let arr = ...
    brr = map f arr
in  mmMult brr brr
```

Sharing and `force`

```
force :: Array sh e
      -> Array sh e
```

```
data Array sh e
 = Manifest sh (UArr e)
 | Delayed  sh (sh -> e)
```

```
let arr = ...
    brr = force (map f arr)
in  mmMult brr brr
```

- For **`Manifest`** arrays, force is the identity.
- For **`Delayed`** arrays, it evaluates all the elements in parallel, producing a manifest array.
- The programmer must add force manually.

slide from Lippmeier's ICFP 2010 talk on Repa

`force :: (Shape sh, Elt a) => Array sh a -> Array sh a`

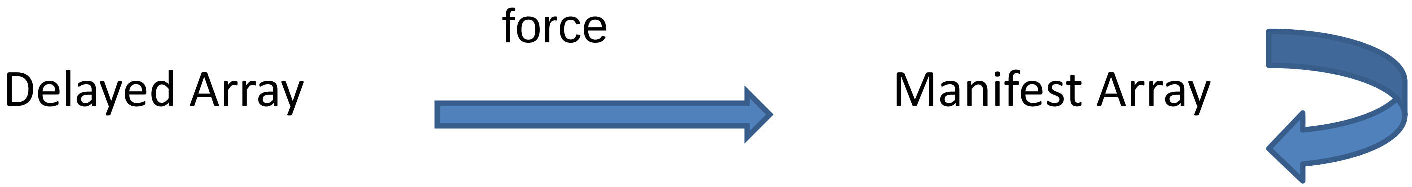
force

Delayed Array

Manifest Array

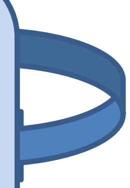
evaluate all elements in parallel

```
force :: (Shape sh, Elt a) => Array sh a -> Array sh a
```



evaluate all elements in parallel

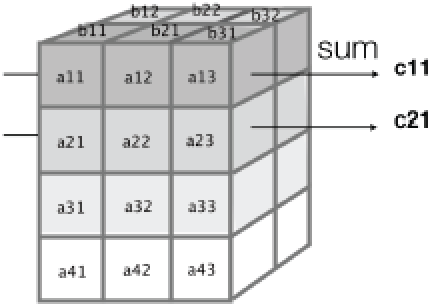
```
force :: (Shape sh, Elt a) => Array sh a -> Array sh a
```



Delayed Array

if you index into a delayed array without forcing it first, then each indexing operation costs a function call. It also *recomputes* the value of the array element at that index.

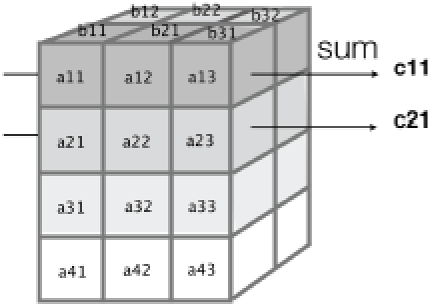
Using the force ...



```
mmMult
  :: (Num e, Elt e)
  => Array DIM2 e
  -> Array DIM2 e -> Array DIM2 e

mmMult arr brr
 = sum (zipWith (*) arrRepl brrRepl)
 where
  trr  = force (transpose2D brr)
  arrR = replicate (Z :.All   :.colsB :.All) arr
  brrR = replicate (Z :.rowsA :.All   :.All) trr
  (Z :. colsA :. rowsA) = extent arr
  (Z :. colsB :. rowsB) = extent brr
```

slide from Lippmeier’s ICFP 2010 talk on Repa

Using the force ...

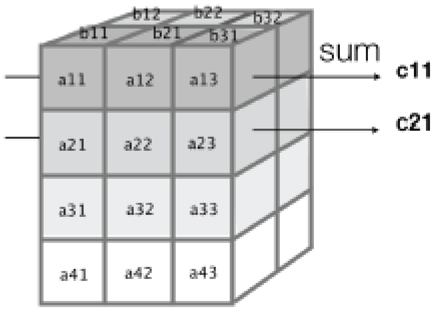


```
mmMult
   :: (Num e, Elt e)
   => Array DIM2 e
   -> Array DIM2 e -> Array DIM2 e

mmMult arr brr
 = sum (zipWith (*) arrRepl brrRepl)
 where
  trr  = force (transpose2D brr)
  arrR = re  licate (Z :.All   :.colsB :.All) arr
  brrR = re     cate (Z :.rowsA :.All   :.All) trr
  (Z :. col        rowsA) = extent arr
  (Z :. col        owsB) = extent brr
```

better cache performance when accessing the elements of b in row major order, so force the transposed version

de from Lippmeier's ICFP 2010 talk on Repa

Using the force ...



```
mmMult
  :: (Num e, Elt e,
  => Array DIM2 e
  -> Array DIM2 e -> A      DIM2 e

mmMult arr brr
 = sum (zipWith (*) arrRepl        l)
 where
  trr  = force (transp
  arrR = re licate (Z
  brrR = re    cate (Z
  (Z :. col       rowsA
  (Z :. col       owsB)    extent brr
```

2 uses of force
to get parallelism
to improve locality

better cache performance when accessing the elements of b in row major order, so force the transposed version

de from Lippmeier's ICFP 2010 talk on Repa

prescan in Repa (clear version)

```
import Data.Array.Repa   as   A

prescan0 :: (Elt a) => (a -> a -> a) -> a  -> (Array (Z :. Int) a) -> (Array (Z :. Int) a)

prescan0 f _ as | size (extent as) == 1 = fromList (Z :. (1 :: Int)) [i]

prescan0 f i as | otherwise = let es = selEven as
                                  os = selOdd  as
                                  ss = prescan0 f i (A.zipWith f es os)
                              in interleave2 ss (A.zipWith f ss es)
```

same prescan in Repa (my fastest so far)

```
-- assumes input of length a power of 2
prescan :: (Elt a) => (a -> a -> a) -> a  -> (Array (Z :. Int) a) -> (Array (Z :. Int) a)
{-# INLINE prescan #-}
prescan f !i !as = sc as
   where
       sc as | size (extent as) == 1 = force $ fromList (Z :. (1 :: Int)) [i]
       sc as | otherwise =
           let es = force $ selEven as
                os = force $ selOdd  as
                ss = force $ sc (A.zipWith f es os)
           in as `deepSeqArray` interleave2M ss (A.zipWith f ss es)
```

5 or 6 times faster   for sumAll . prescan (+) (0::Int)   on 2^20 inputs
still 3-4 times slower than scanl1 ☹   but good speedup on 2 cores -N4
and hopefully on more

more operations

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

Doesn’t care about shape of array. Just applies the function to each element.

plain Haskell

```
foldl :: (a -> b -> a) -> a -> [b] -> a
```

Repa

```
foldl :: (Shape sh, Elt a, Elt b) => (a -> b -> a) -> a -> Array (sh :. Int) b -> Array sh a
```

reduce shape by one dimension

```
*Main> let y = fromList ((Z :. 2 :. 3 :. 3) :: DIM3) [1..18]
*Main> A.fold (+) 0 y
Array (Z :. 2 :. 3) [6.0,15.0,24.0,33.0,42.0,51.0]

*Main> A.transpose y
Array (Z :. 2 :. 3 :. 3)
[1.0,4.0,7.0,2.0,5.0,8.0,3.0,6.0,9.0,10.0,13.0,16.0,11.0,14.0,17.0,12.0,15.0,18.0]



*Main> A.fold (+) 0 (A.transpose y)
Array (Z :. 2 :. 3) [12.0,15.0,18.0,39.0,42.0,45.0]
```

each fold is sequential, but they are all done at once

more operations

map :: (Shape sh, Elt a, Elt b) => (a -> b) ay sh a -> Array sh b

Doesn’t care abo n to each element.

Note: in this version, there is no map over (say) the inner dimension (e.g. each row of a DIM2 array)
That might cause NESTEDNESS
Note, though, that later Repa versions have chunked arrays and also a notion of regions in an array

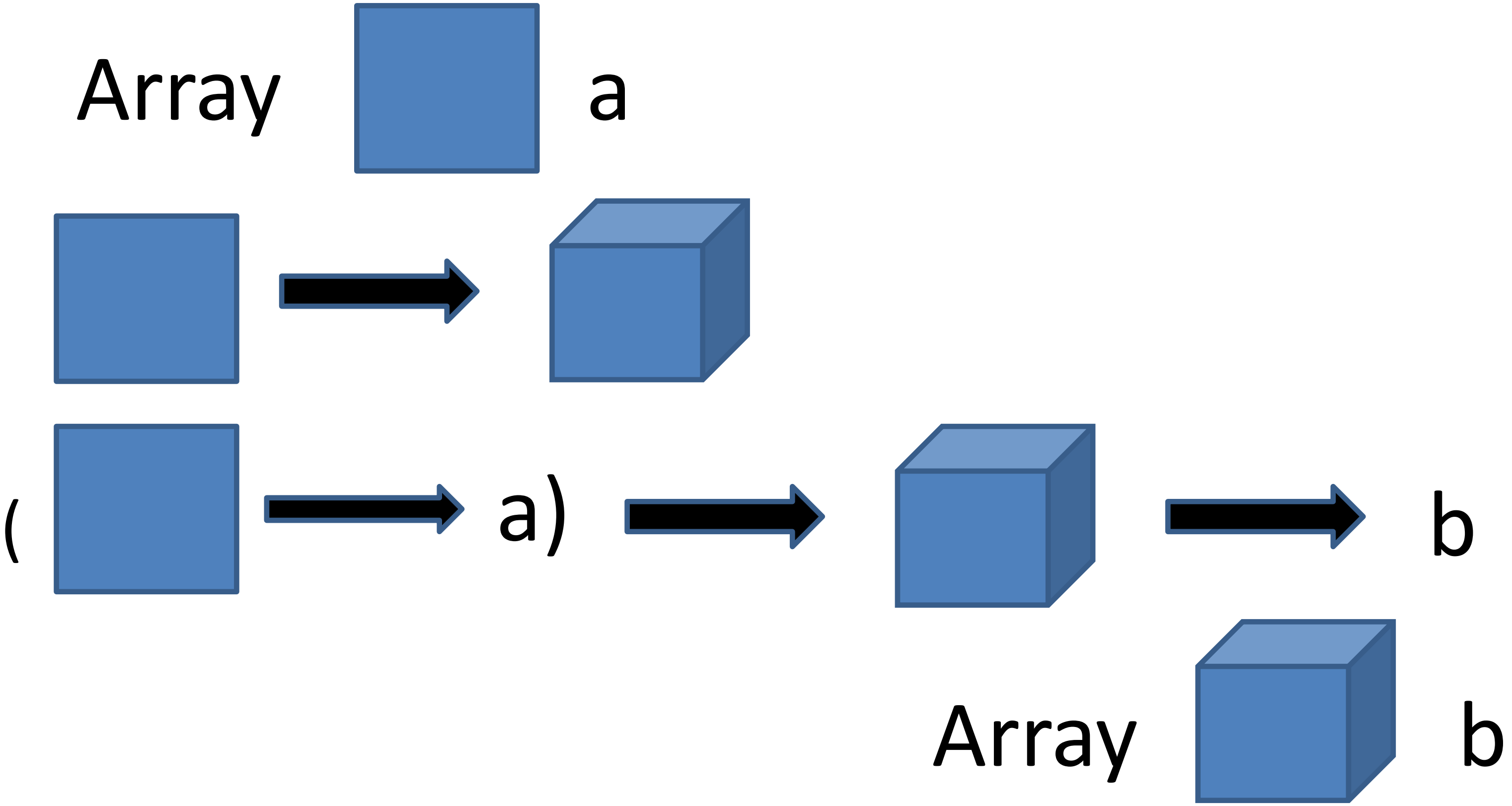
plain Haskell

Repa foldl :: (Shape :. Int) b -> Array sh a

reduce shape by one dimension

key function: traverse

```
traverse
  :: (Shape sh', Shape sh, Elt a) =>
     Array shin a -> (shin -> shout) -> ((shin -> a) -> shout -> b) -> Array shout b
```



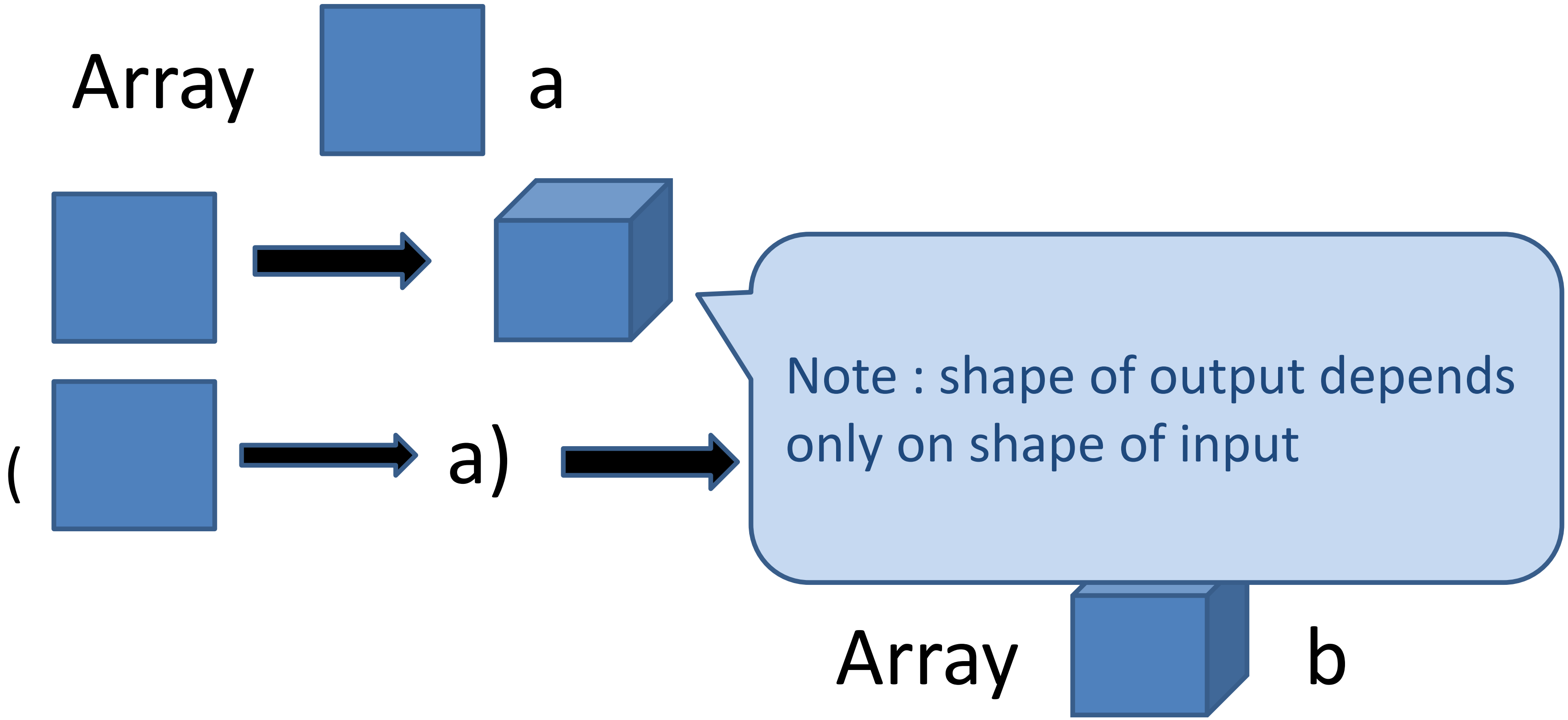
Array a

(a) b

Array b

key function: traverse

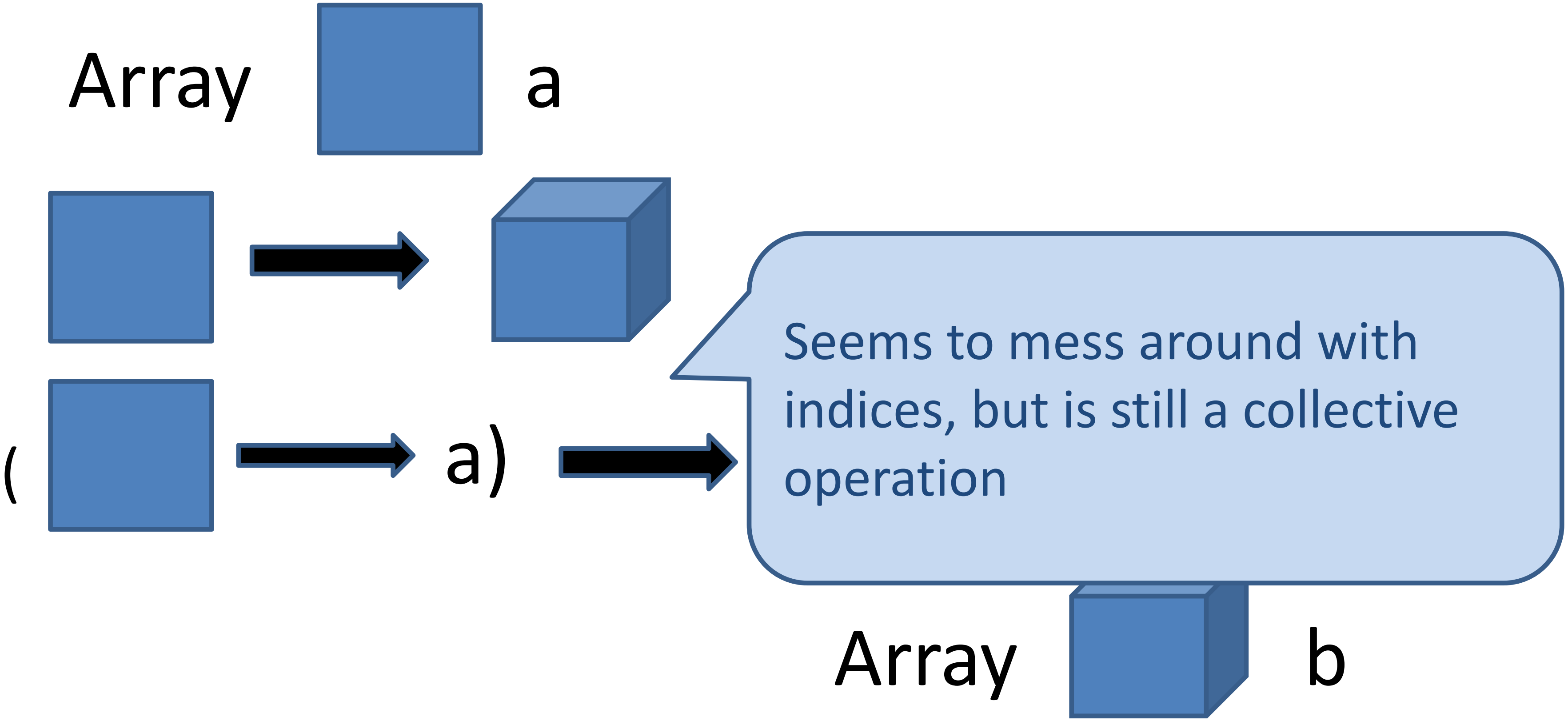
```
traverse
  :: (Shape sh', Shape sh, Elt a) =>
     Array shin a -> (shin -> shout) -> ((shin -> a) -> shout -> b) -> Array shout b
```



Note : shape of output depends only on shape of input

key function: traverse

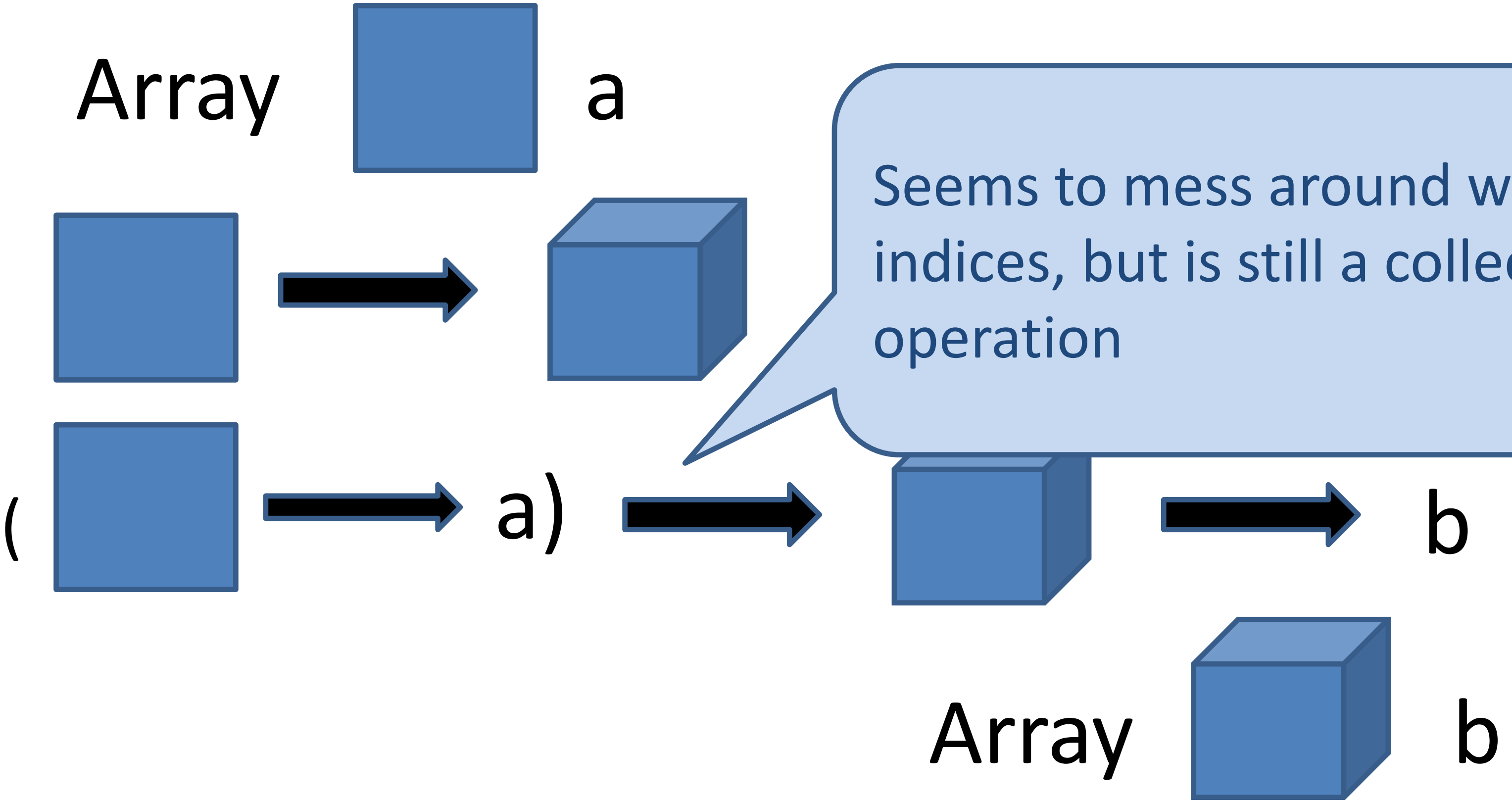
```
traverse
  :: (Shape sh', Shape sh, Elt a) =>
     Array shin a -> (shin -> shout) -> ((shin -> a) -> shout -> b) -> Array shout b
```



Seems to mess around with indices, but is still a collective operation

key function: traverse

```
traverse
  :: (Shape sh', Shape sh, Elt a) =>
     Array shin a -> (shin -> shout) -> ((shin -> a) -> shout -> b) -> Array shout b
```



Array a

(a) b

Array b

Seems to mess around with indices, but is still a collective operation

there is also a version called unsafeTraverse that skips bounds checking and so is faster

use of traverse

```
backpermute :: (Shape sh’, Shape s, Elt e) =>
                      shout -> (shout -> shin) -> Array shin e -> Array shout e
backpermute shout perm arr  =  traverse arr (const shout) (. perm)
```

output shape
shout

input shape -> output shape
shin   ->   shout

(ignore input shape)

perm :: (shout -> shin)
(. perm) :: (shin -> a) -> shout -> a

use of traverse

```
A.map :: (Shape sh, Elt b, Elt a) => (a -> b) -> Array sh a -> Array sh b

map f arr = traverse arr id (f .)
```

unsafeTraverse

```
{-# INLINE bfly #-}
bfly !k !as
  = unsafeTraverse as id (\f (s :. i) -> let a = f (s :. i)
                                             b = f (s :. (flipBit i k))
                                         in if (testBit i k) then (b-a) else (a+b))
```

unsafeTraverse

```
{-# INLINE bfly #-}
bfly !k !as
  = unsafeTraverse as id (\f (s :. i) -> let a = f (s :. i)
                                             b = f (s :. (flipBit i k))
                                         in if (testBit i k) then (b-a) else (a+b))

{-# INLINE twids #-}
twids !k !as
   = let k2 = 2^k
         k2' = 2*k2 in
     unsafeTraverse as id (\f (s :. i) -> let a = f (s :. i)
                                              t = tw (i `mod` k2) k2'
                                          in if (testBit i k) then t*a else a)
```

```
{-# INLINE interleave2M #-}
interleave2M arr1 arr2
 = arr1 `deepSeqArray` arr2 `deepSeqArray`
   unsafeTraverse2 arr1 arr2 shapeFn elemFn
 where
        shapeFn dim1 dim2
         | sh :. len1   <- dim1
         ,  sh :. len2   <- dim2
         = sh :. (len1 + len2)

        elemFn get1 get2 (sh :. ix)
         = case ix `mod` 2 of
                0          -> get1 (sh :. ix `div` 2)
                1          -> get2 (sh :. ix `div` 2)
```

```
*Main> let w = fromList (Z :. 2 :. 3 :. 3 :: DIM3) [1..(18 ::Int)]
*Main> selEven w
Array (Z :. 2 :. 3 :. 2) [1,3,4,6,7,9,10,12,13,15,16,18]
*Main> selOdd w
Array (Z :. 2 :. 3 :. 1) [2,5,8,11,14,17]


*Main> interleave2 (selEven w) (selOdd w)
Array (*** Exception: Data.Array.Repa.interleave2: arrays must
have same extent
*Main> interleave2M (selEven w) (selOdd w)
Array (Z :. 2 :. 3 :. 3)
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18]
```

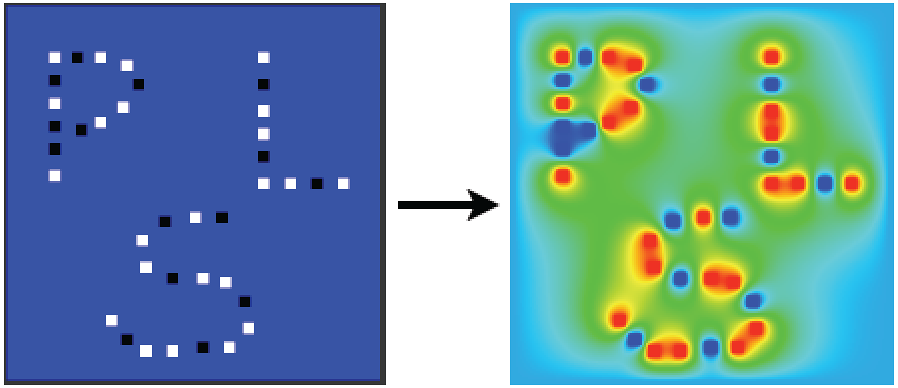
unsafeTraverse

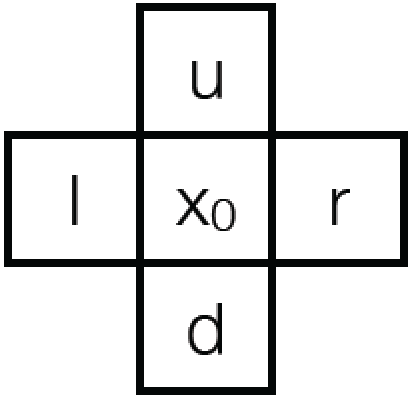
```
{-# INLINE bfly #-}
bfly !k !as
  = unsafeTraverse as id (\f (s :. i) -> let a = f (s :. i)
                                             b = f (s :. (flipBit i k))
                                         in if (testBit i k) then (b-a) else (a+b))

{-# INLINE twids #-}
twids !k !as
   = let k2 = 2^k
         k2' = 2*k2 in
     unsafeTraverse as id (\f (s :. i) -> let a = f (s :. i)
                                              t = tw (i `mod` k2) k2'
                                          in if (testBit i k) then t*a else a)
```

```
{-# INLINE fft4 #-}
fft4 !n  !as  = foldr1 (.) [ force . twids k . bfly k  | k <- [0..(n-1)]] as
```

Example Applications

Solving the Laplace Equation

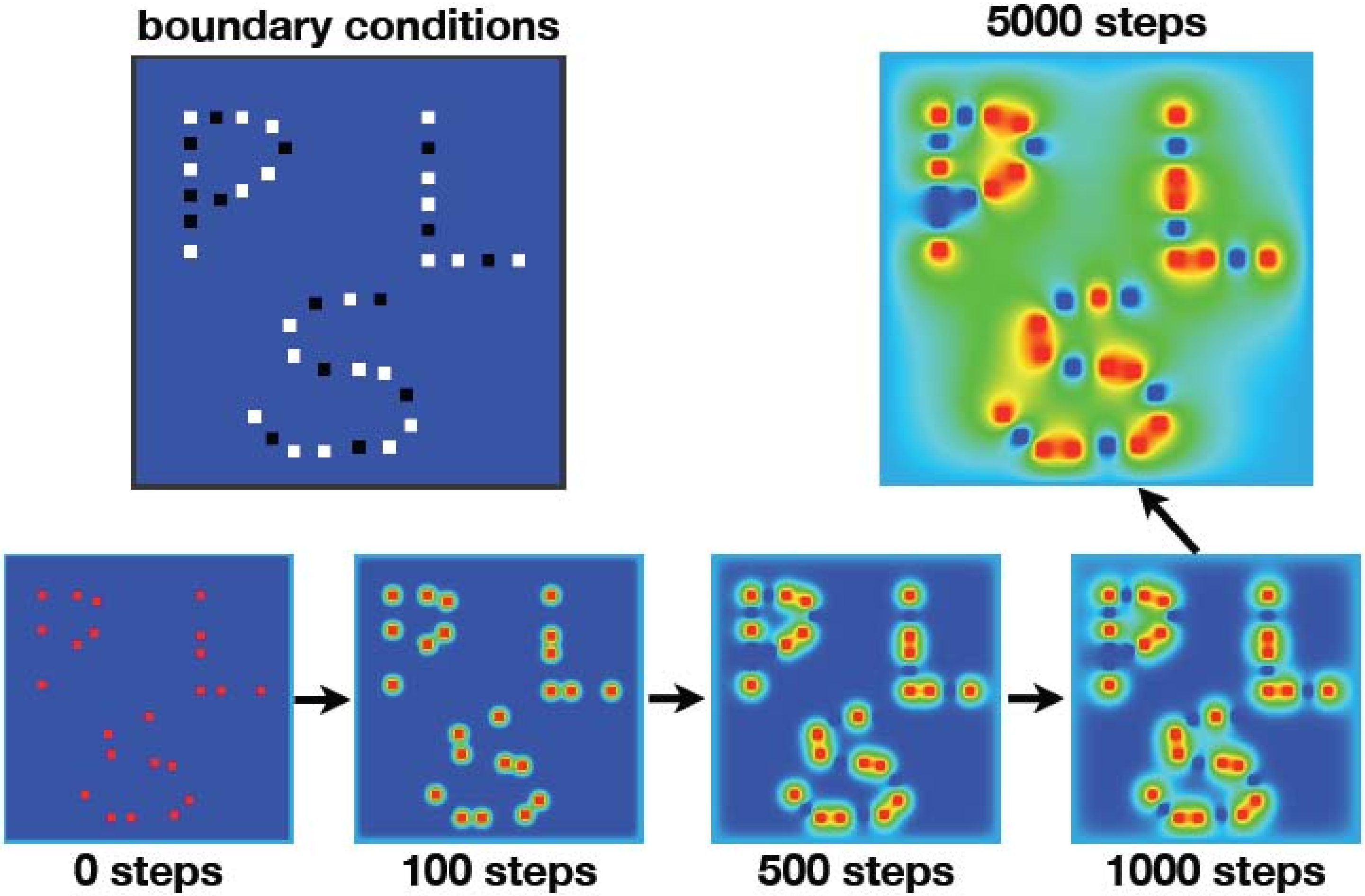




$$x_0' = (l + r + u + d) / 4$$

slide from Lippmeier's ICFP 2010 talk on Repa

Laplace Equation



boundary conditions

5000 steps

0 steps

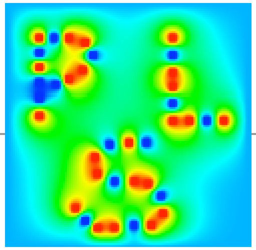
100 steps

500 steps

1000 steps

slide from Lippmeier's ICFP 2010 talk on Repa

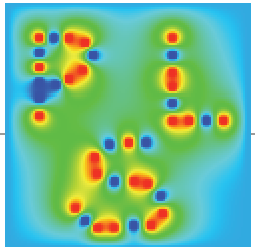
Laplace Equation

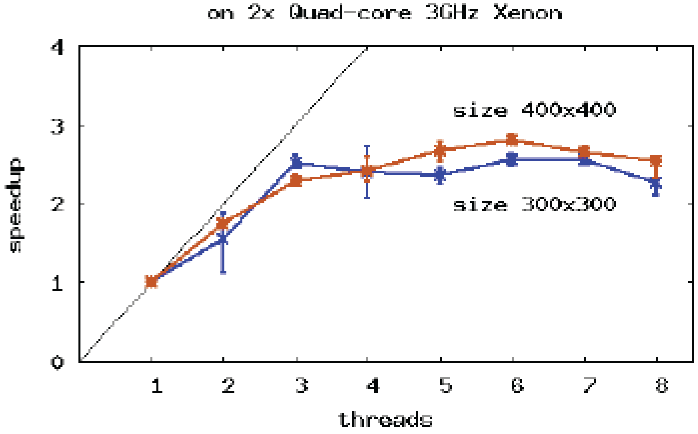


```
stencil :: Array DIM2 Double
        -> Array DIM2 Double

stencil arr
 = traverse arr id update
 where
   _ :. height :. width = extent arr

   update get d@(sh :. i :. j)
    = if isBoundary i j
         then get d
         else (get (sh :. (i-1)) :. j)
            +  get (sh :.  i     :. (j-1))
            +  get (sh :. (i+1)) :. j)
            +  get (sh :. i      :. (j+1))) / 4
```

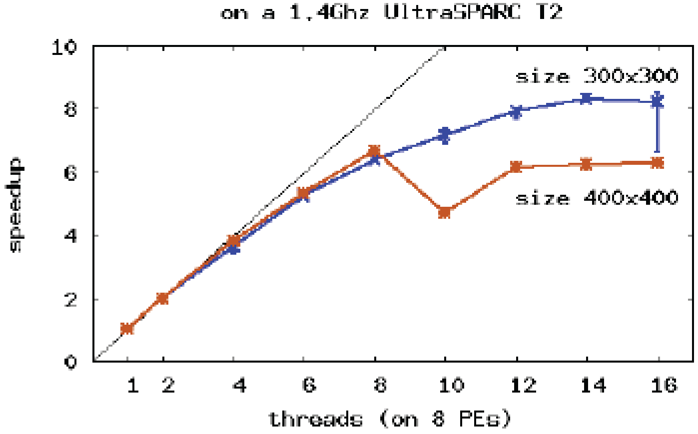
slide from Lippmeier’s ICFP 2010 talk on Repa

Laplace Equation





	GCC	single thread	fastest parallel
Xenon	0.70	1.7s	0.68s
T2	6.5s	32s	3.8s



- GHC native code generator does no instruction reordering on SPARC. No LLVM 'port.
- Single threaded on T2 is slow

slide from Lippmeier's ICFP 2010 talk on Repa

Conclusions

Based on DPH technology

Good speedups!

Neat programs

Good control of Parallelism

BUT CACHE AWARENESS needs to be tackled

Next lecture (Monday)

I would like to have a couple of student talks next wednesday (having talked to a couple of you earlier). Please contact to me again to confirm!

Student talks on topics related to the course would be most welcome!