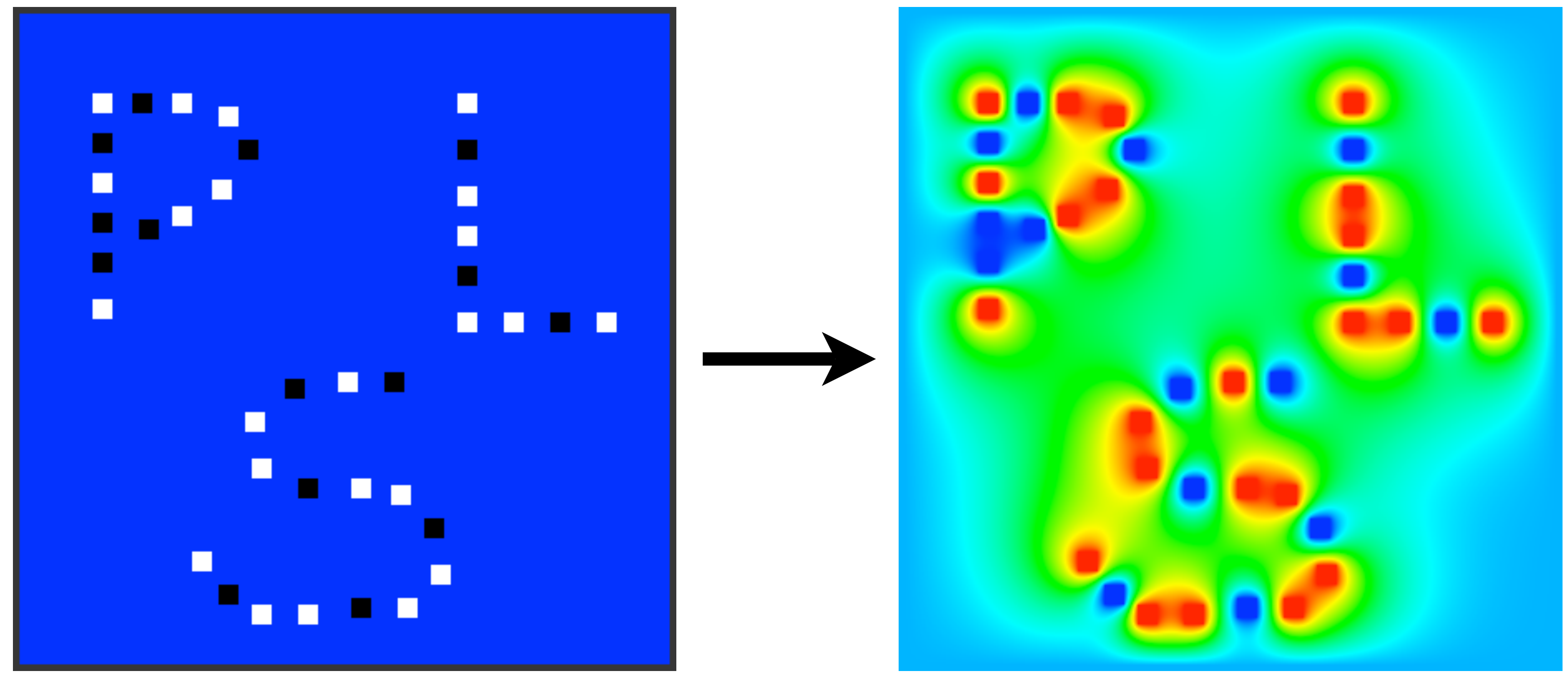
Repa
Regular, Shape-polymorphic, Parallel Arrays

Gabriele Keller[1] / Manuel Chakravarty[1]
Roman Leshchinskiy[1] / Simon Peyton Jones[2] / Ben Lippmeier[1]

[1]University of New South Wales
[2]Microsoft Research Ltd, Cambridge England

Example Applications

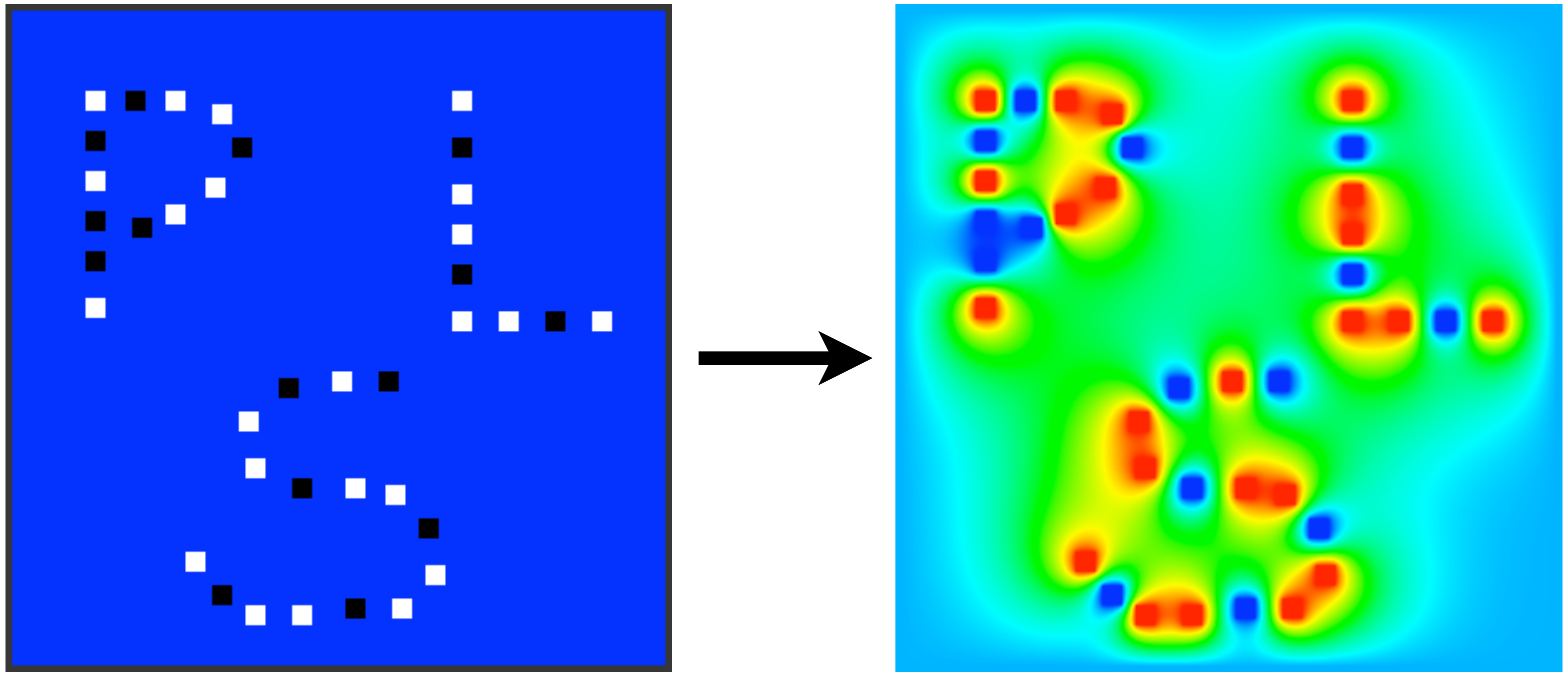
Solving the
Laplace Equation

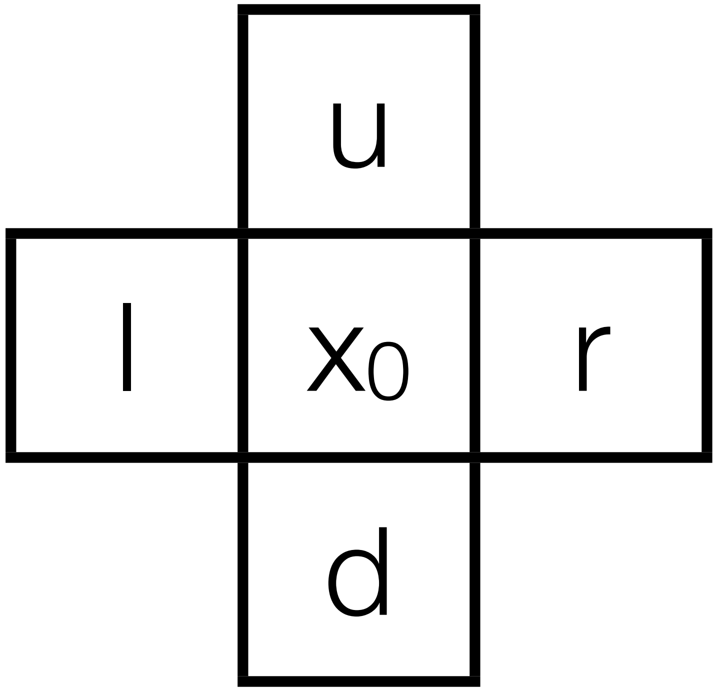


Fast Fourier
Transform
(highpass filter)

Example Applications

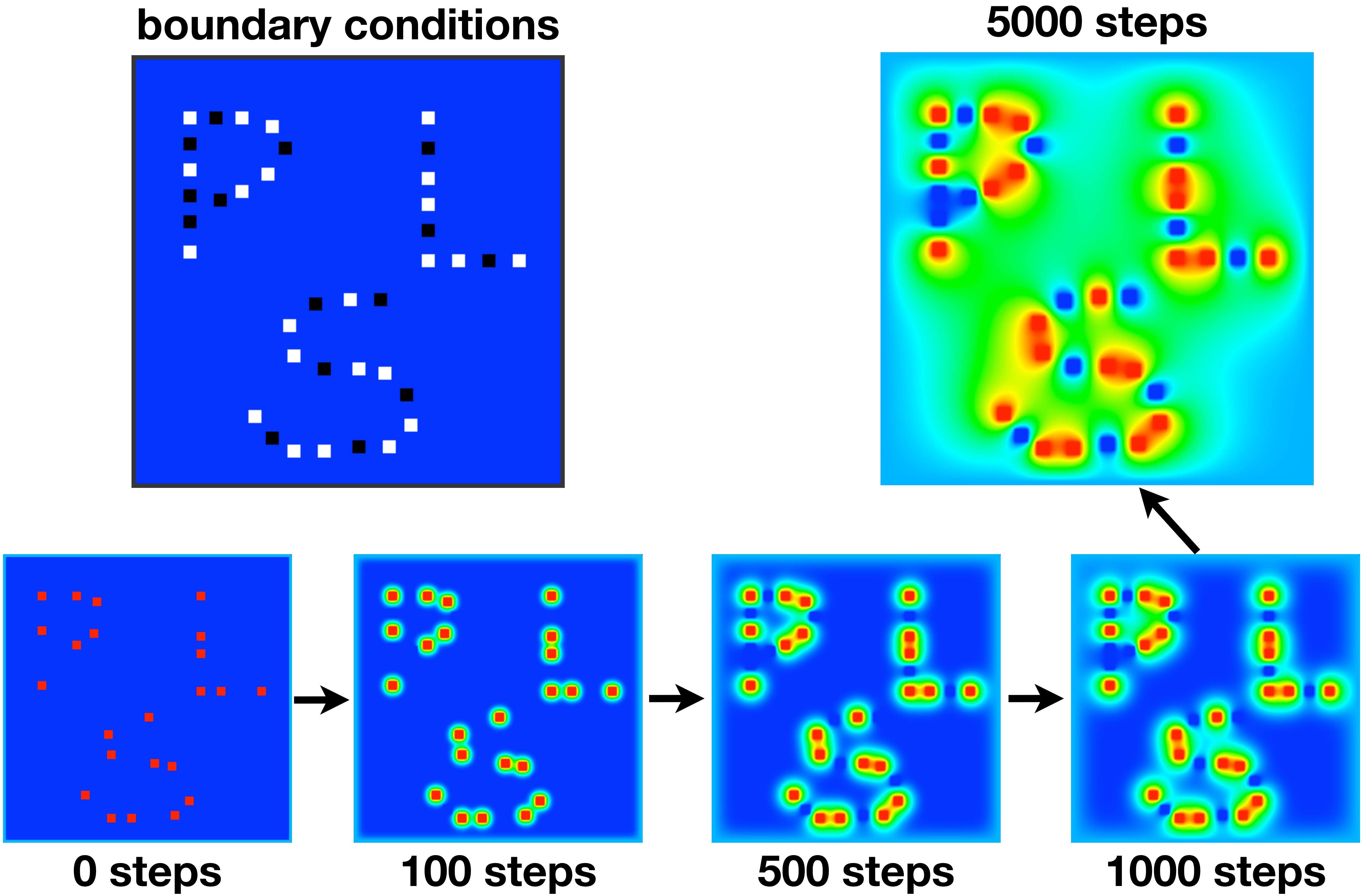
Solving the
Laplace Equation



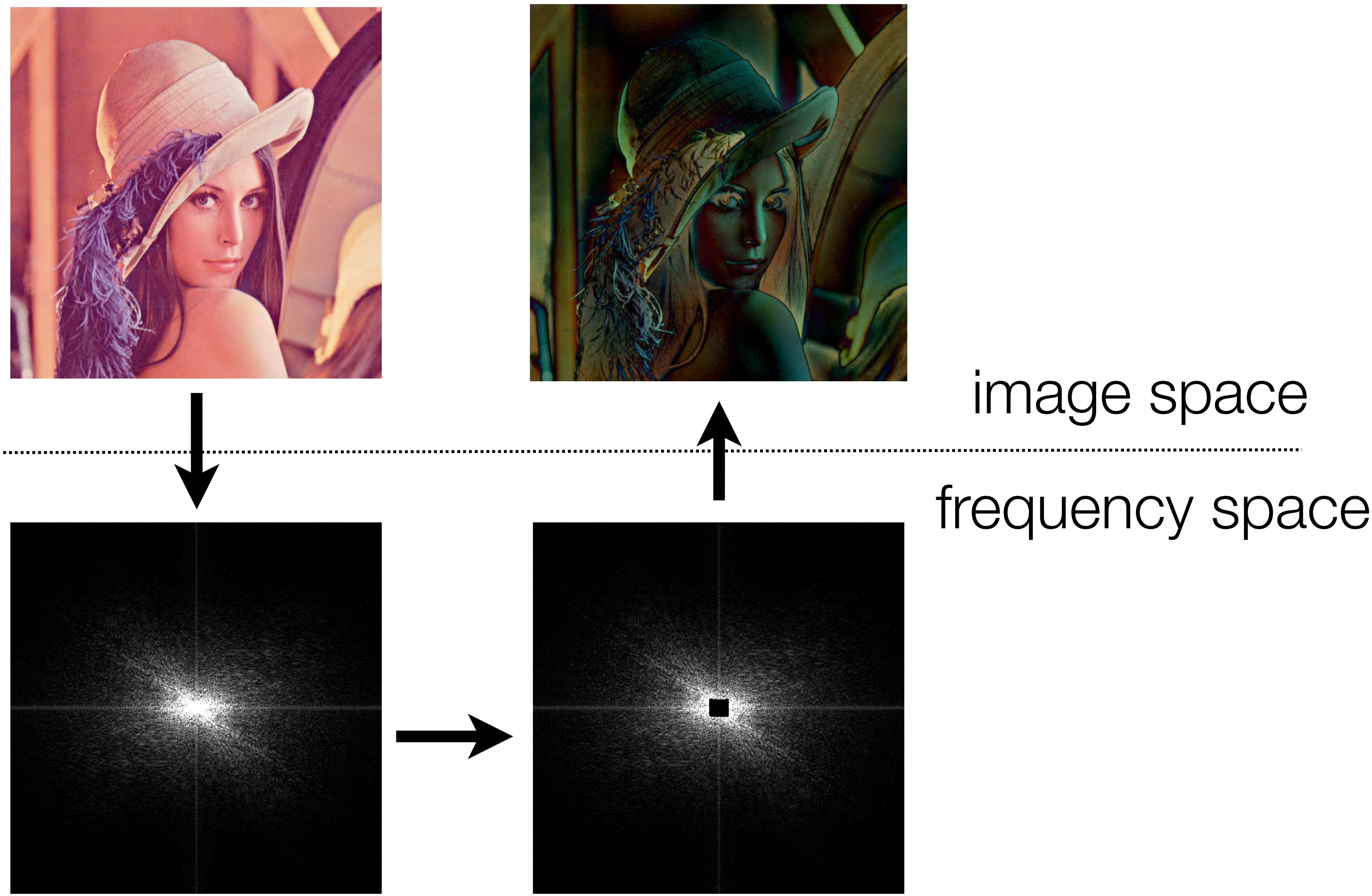


$$x_0' = (l + r + u + d) / 4$$

Laplace Equation



2D Fast Fourier Transform (FFT)



Regular, Shape-polymorphic, Parallel Arrays

- **Regular Arrays**
  Arrays are dense and rectangular.
  Most elements are non-zero.

Regular, Shape-polymorphic, Parallel Arrays

- **Regular Arrays**
  Arrays are dense and rectangular.
  Most elements are non-zero.

- **Shape Polymorphic**
  Functions work over arrays of arbitrary rank (dimension).

Regular, Shape-polymorphic, Parallel Arrays

- **Regular Arrays**
  Arrays are dense and rectangular.
  Most elements are non-zero.

- **Shape Polymorphic**
  Functions work over arrays of arbitrary rank (dimension).

- **Flat Data Parallelism**
  Individual computations don’t need to communicate.
  Parallel computations don’t spark further parallel computations

Matrix Transposition

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

transpose2D arr
 = backpermute newExtent swap arr
 where swap (Z :.i :.j) = Z :.j :.i
       newExtent        = swap (extent arr)
```

10	20	30
44	55	66

10	44
20	55
30	66

Matrix Transposition

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

transpose2D arr
 = backpermute newExtent swap arr
 where swap (Z :.i :.j) = Z :.j :.i
       newExtent        = swap (extent arr)
```

- An Index Space Transform
- The ordering of the elements changes, but the values do not.

10	20	30
44	55	66

10	44
20	55
30	66

Matrix Transposition

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

transpose2D arr
 = backpermute newExtent swap arr
 where swap (Z :.i :.j) = Z :.j :.i
       newExtent        = swap (extent arr)
```

- An Index Space Transform
- The ordering of the elements changes, but the values do not.
- We usually want to push such transforms into the consumer.

10	20	30
44	55	66

10	44
20	55
30	66

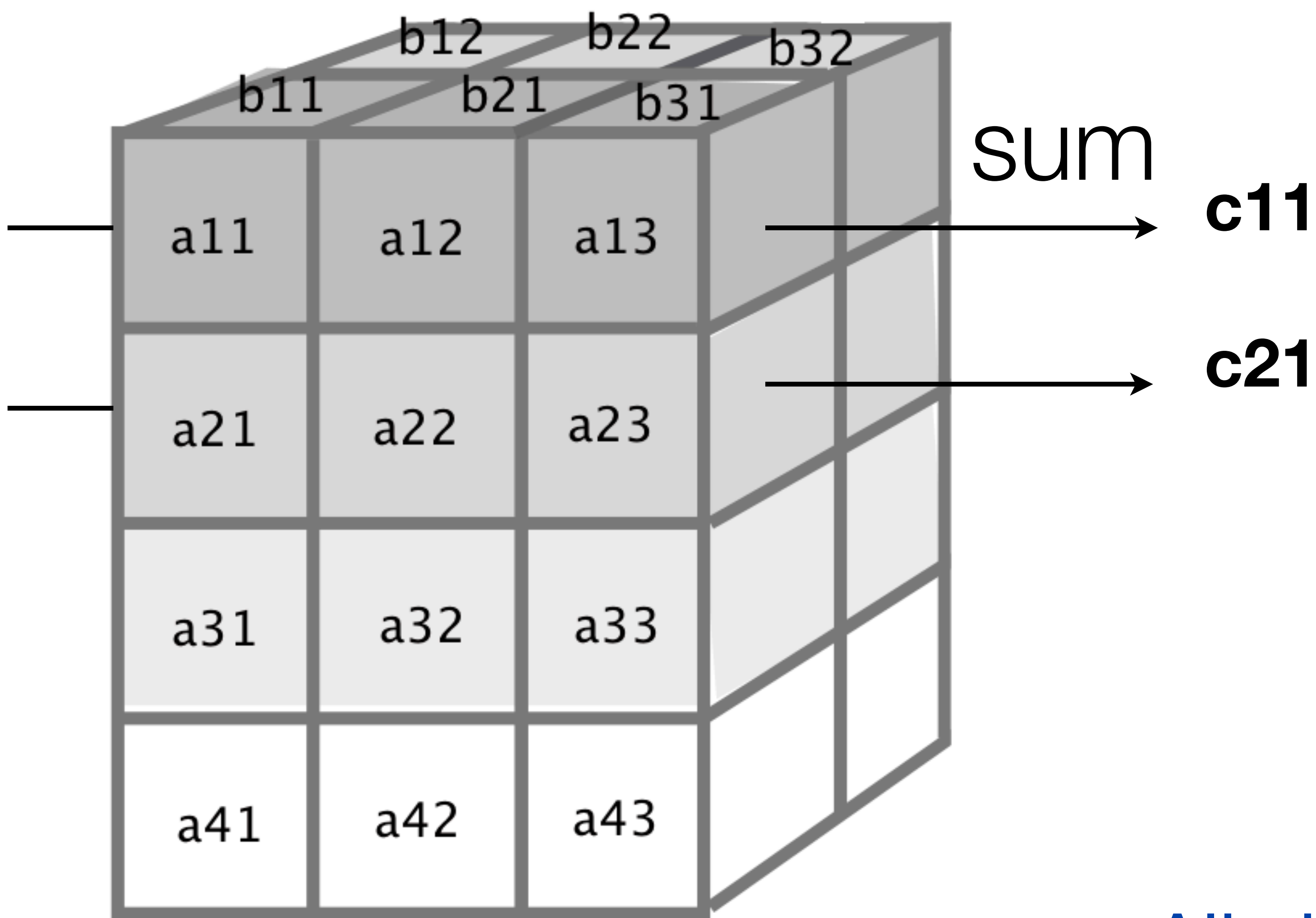
Matrix Multiplication

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

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

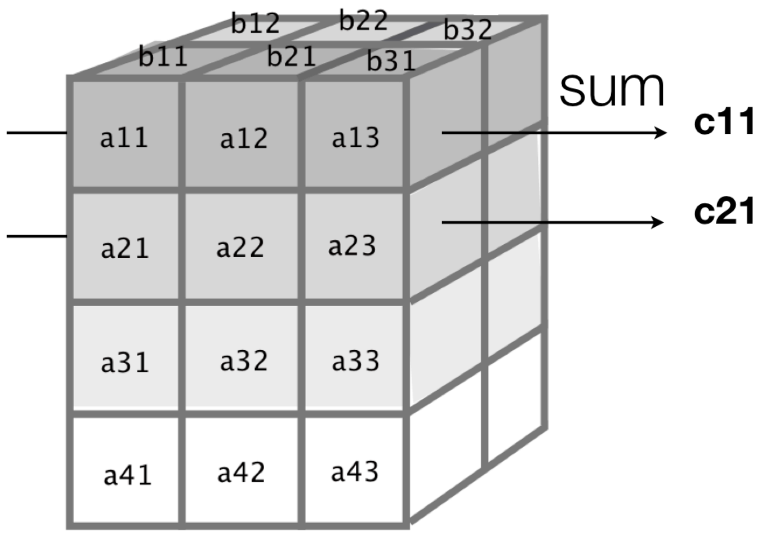
Matrix Multiplication

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



- All elements of the result can be computed in parallel!

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
```

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.

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).

Sharing and `force`

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

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`

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

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

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

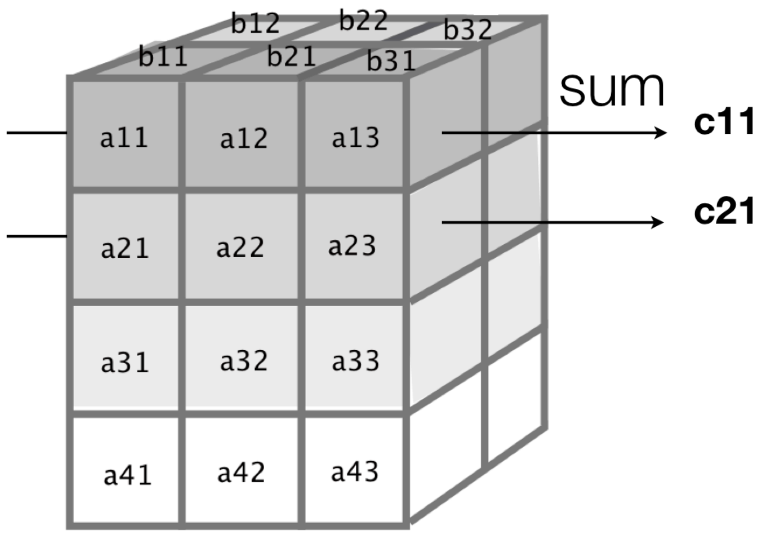
- For **`Manifest`** arrays, force is the identity.
- For **`Delayed`** arrays, it evaluates all the elements in parallel, producing a manifest array.
- The programmer must add force manually.

Using the force ...

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

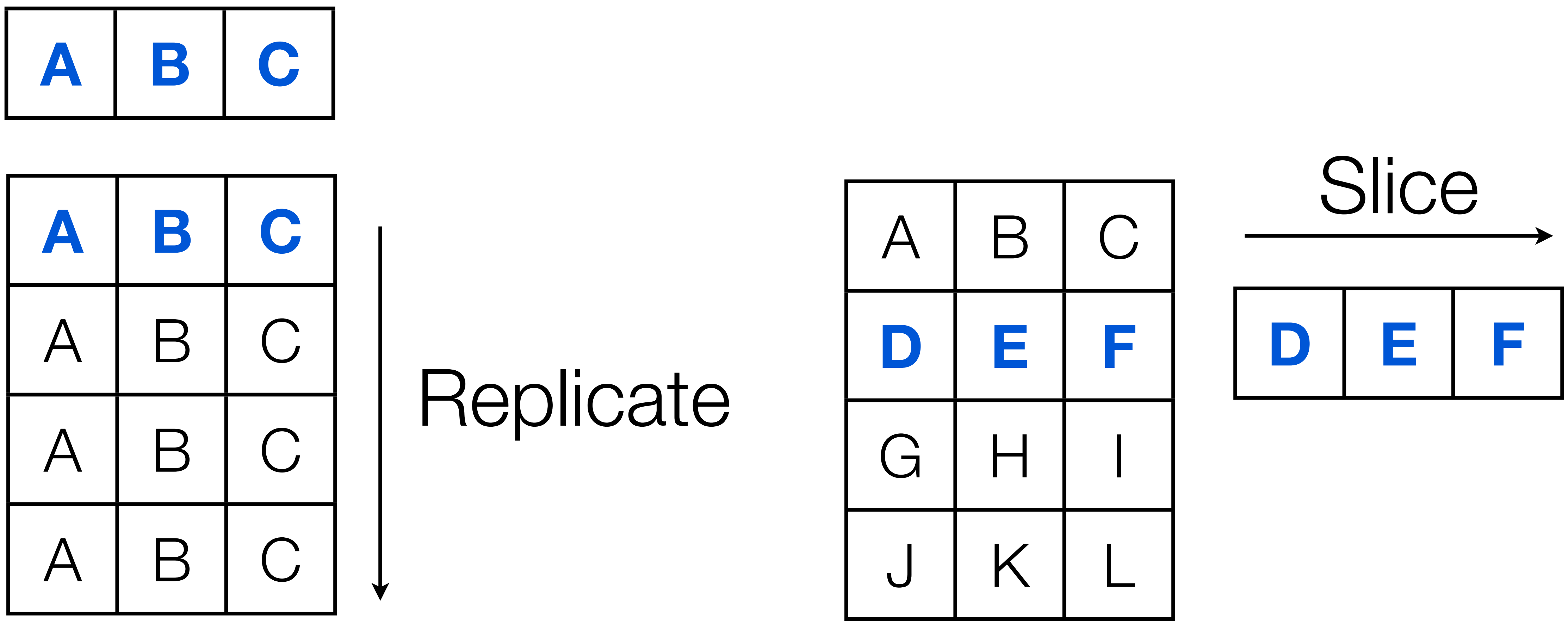
- We get better cache performance when accessing the **`b`** elements left-to-right rather than top-to-bottom

Using the force ...



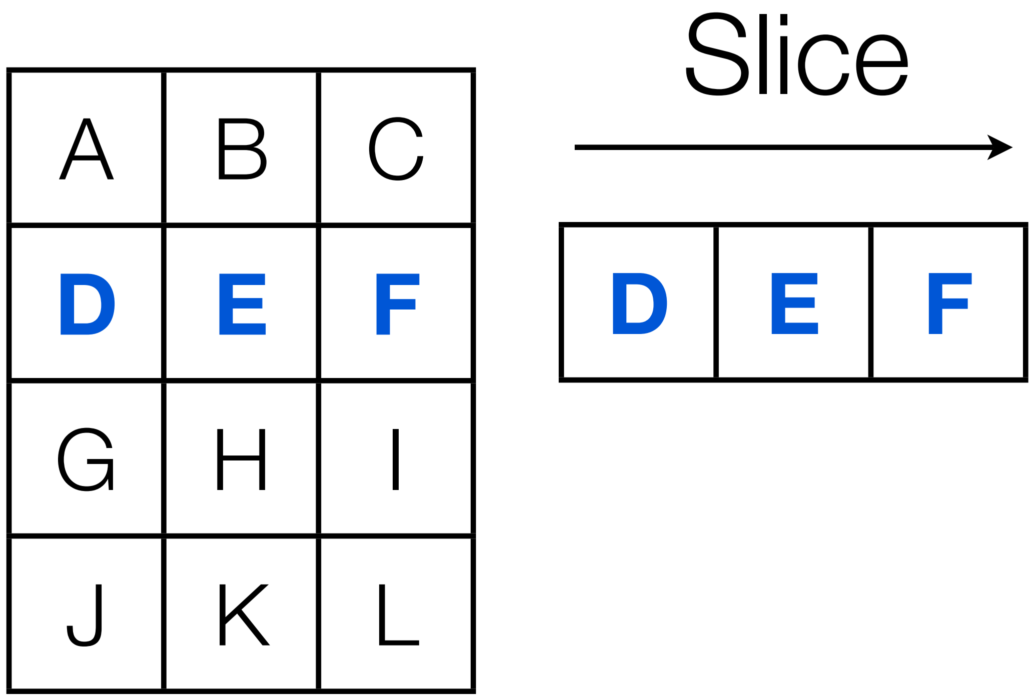

```
mmMult
   :: (Num e, Elt e)
   => Array DIM2 e
   -> Array DIM2 e -> Array DIM2 e

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
```

Replicate and Slice are duals.

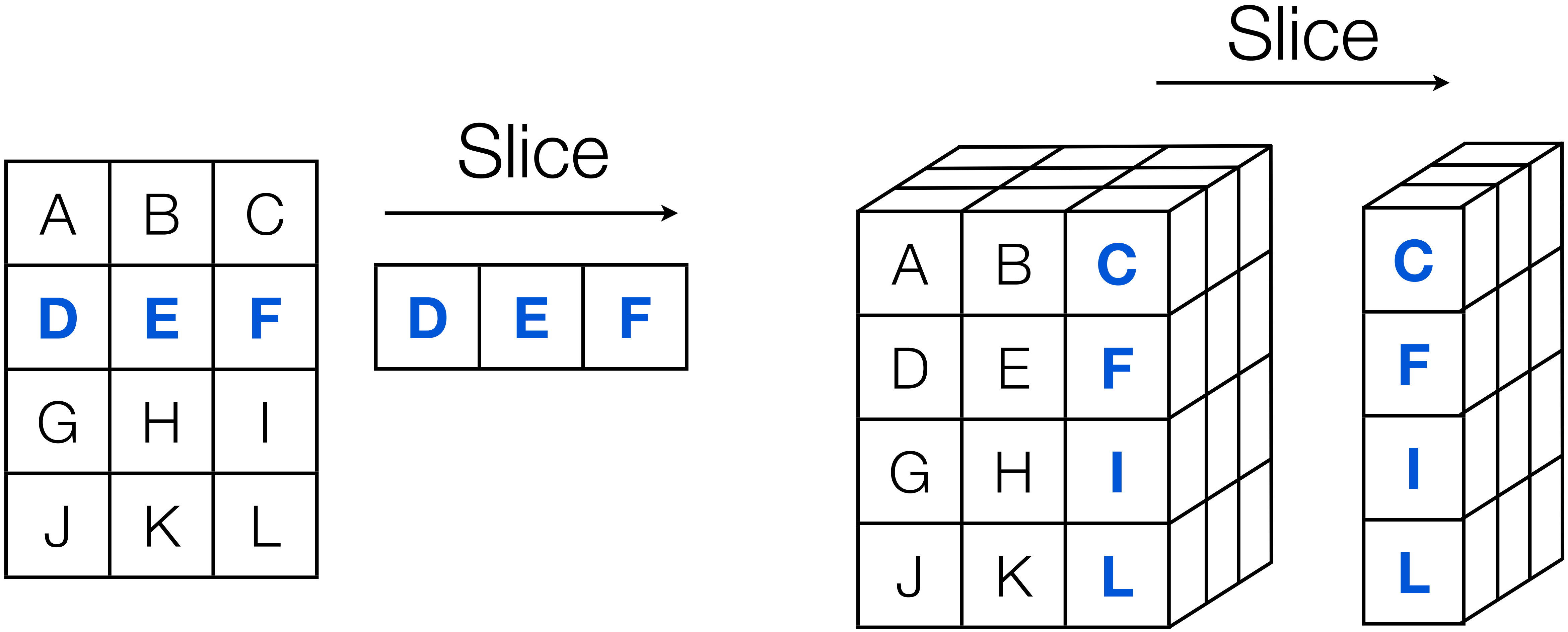


- Replicate and Slice are index transforms.
- The values of the array elements do not change.

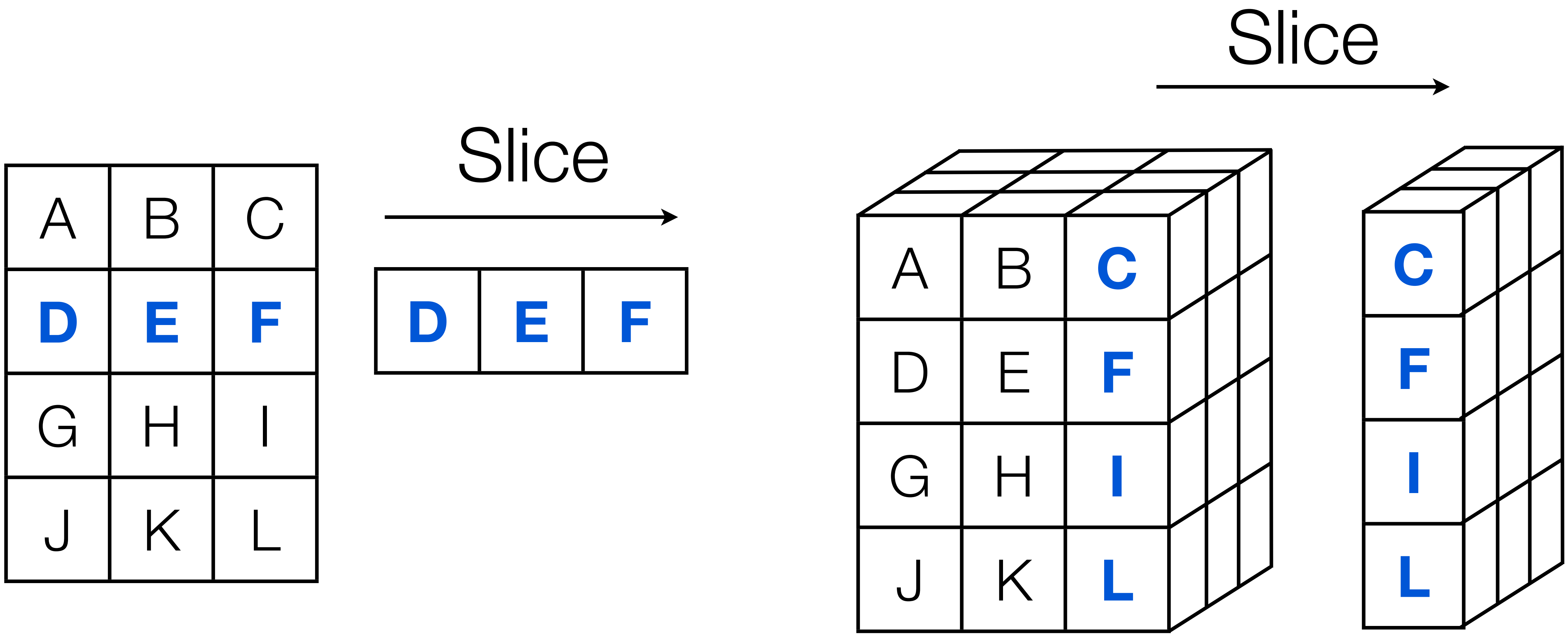
Type hackery



Type hackery



Type hackery

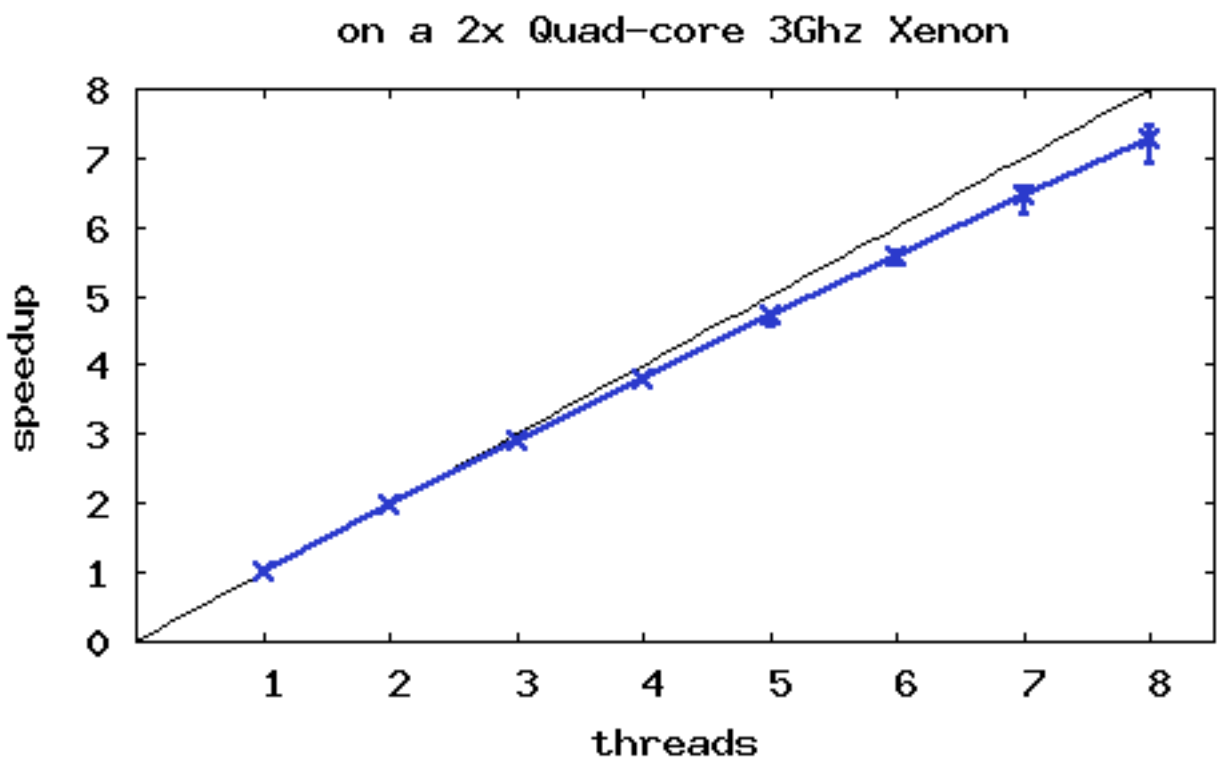



```
slice :: ( Slice sl, Elt e
         , Shape (FullShape  sl))
         , Shape (SliceShape sl))
      => Array (FullShape sl)  e
      -> sl -> Array (SliceShape sl) e
```

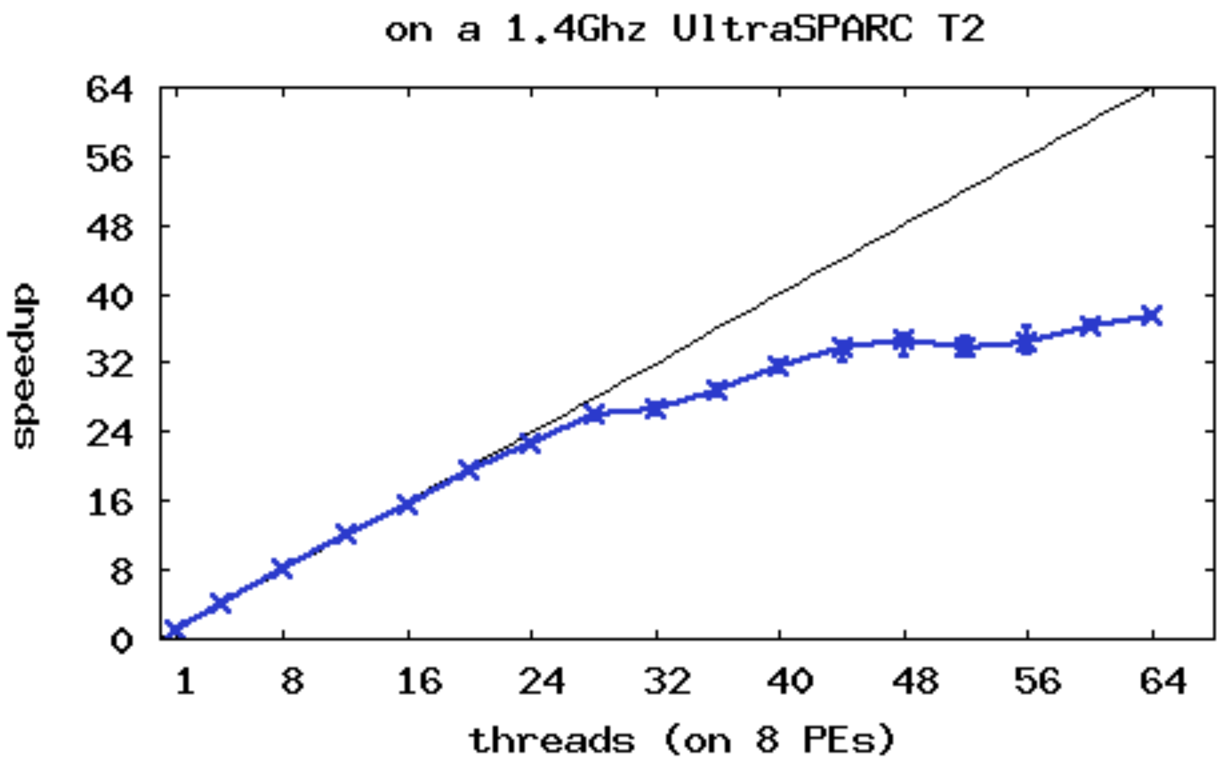
Other operations

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

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

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

reshape :: (Shape sh, Shape sh’, Elt e)
        => sh -> Array sh’ e -> Array sh e
```

Matrix Multiplication 1024x1024

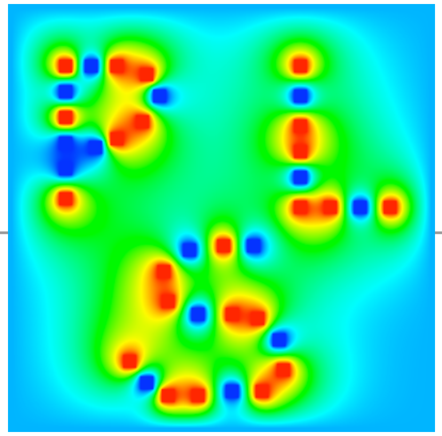


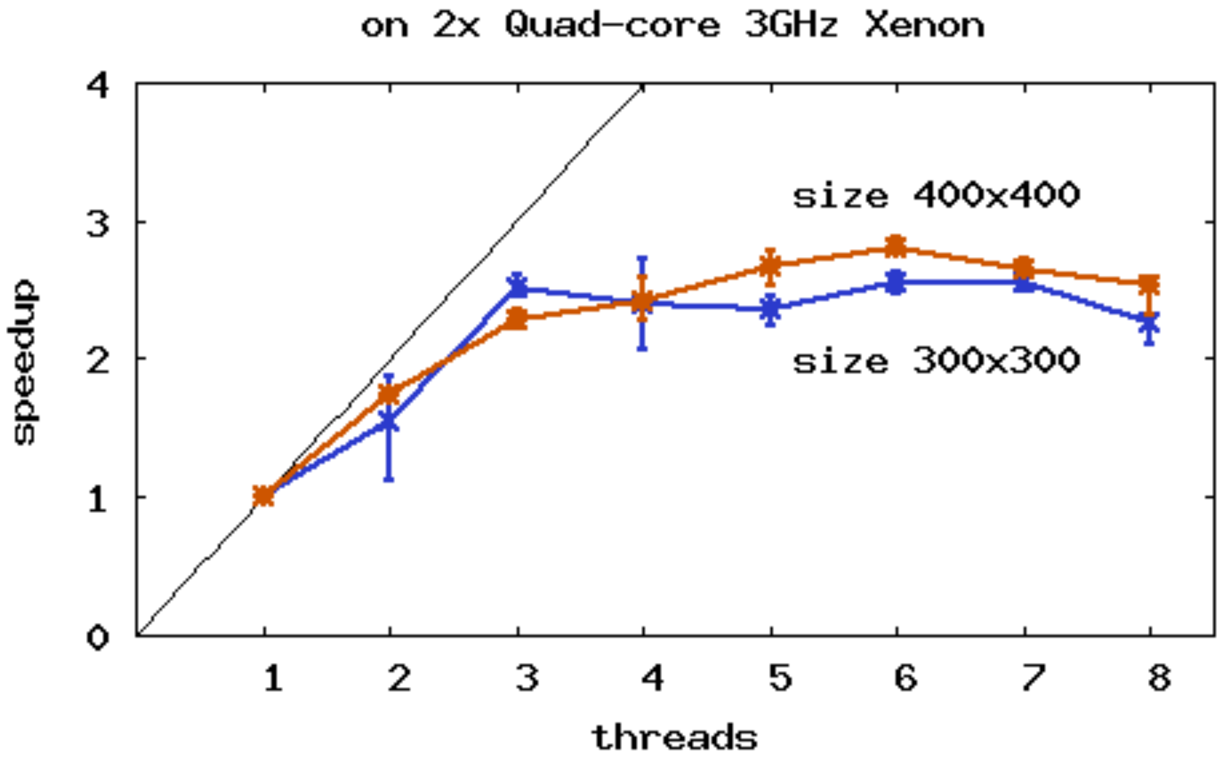
	GCC	single thread	fastest parallel
Xenon	3.8s	4.6s	0.64s
T2	52s	92s	2.4s



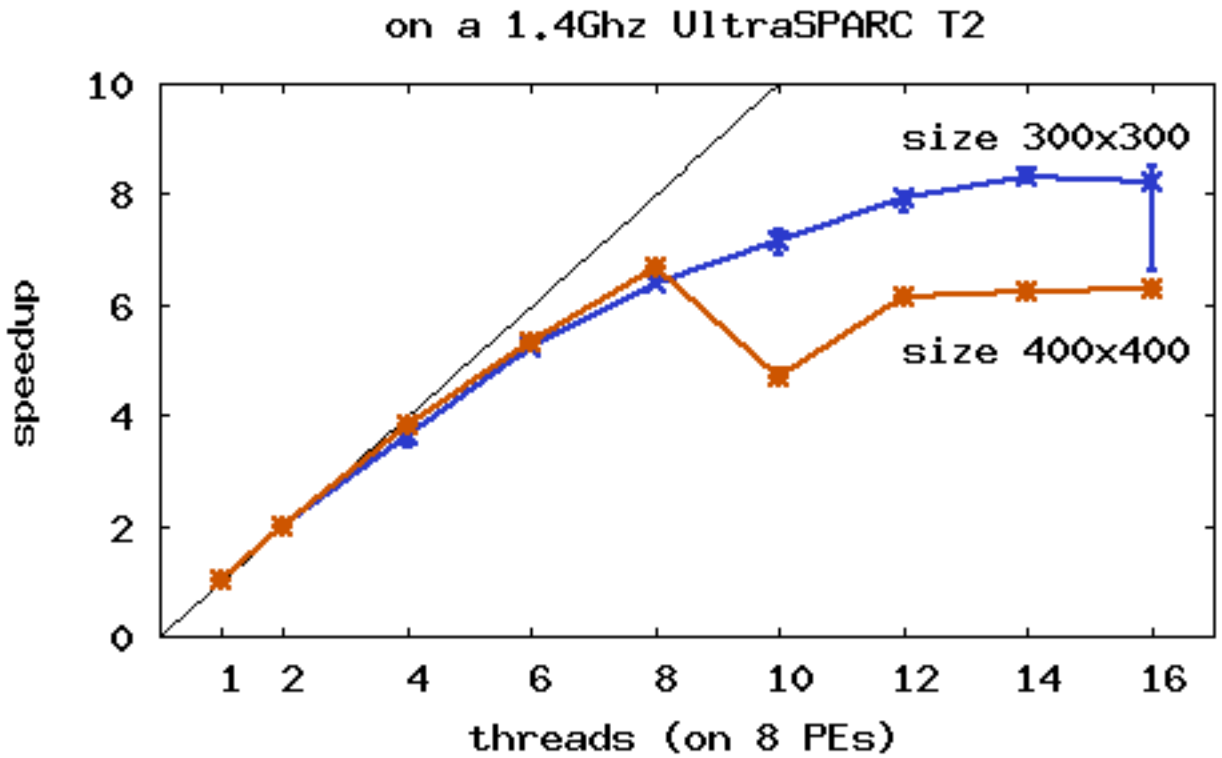
- C reference version uses double nested loops.
- Exposing sufficient parallelism on the T2 is a must.

Laplace Equation



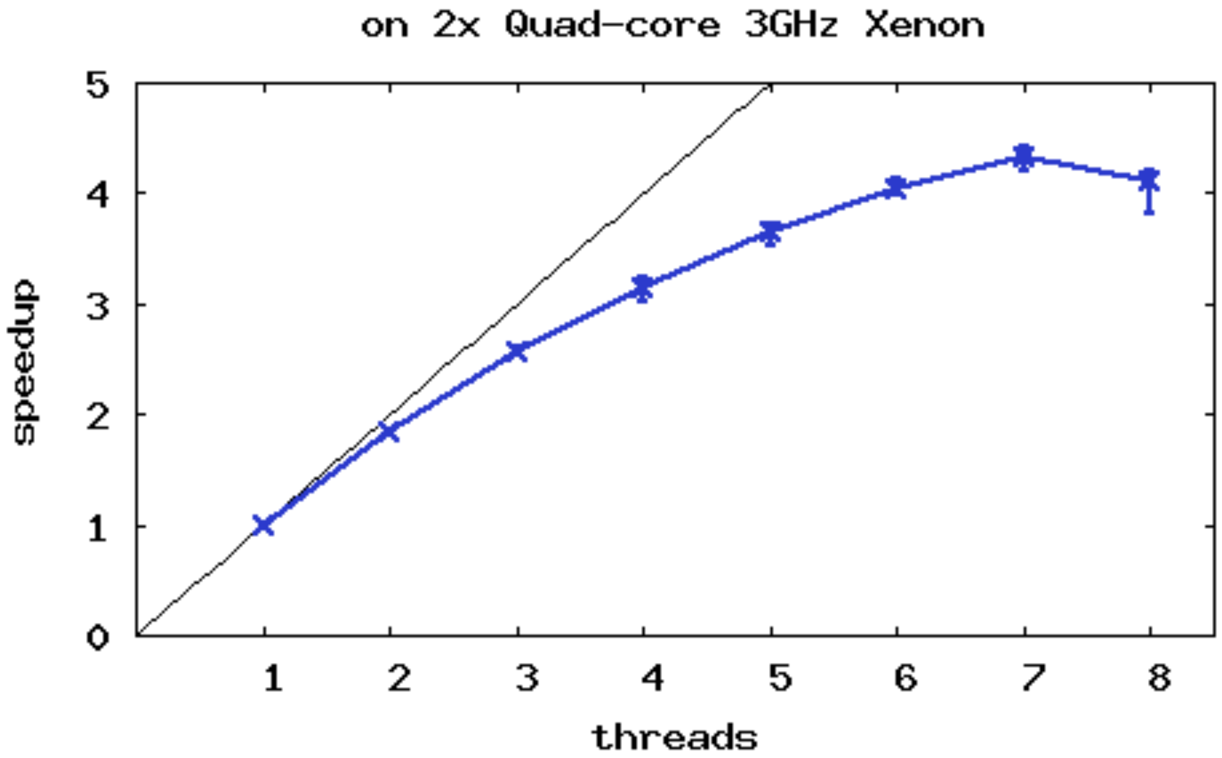


	GCC	single thread	fastest parallel
Xenon	0.70	1.7s	0.68s
T2	6.5s	32s	3.8s

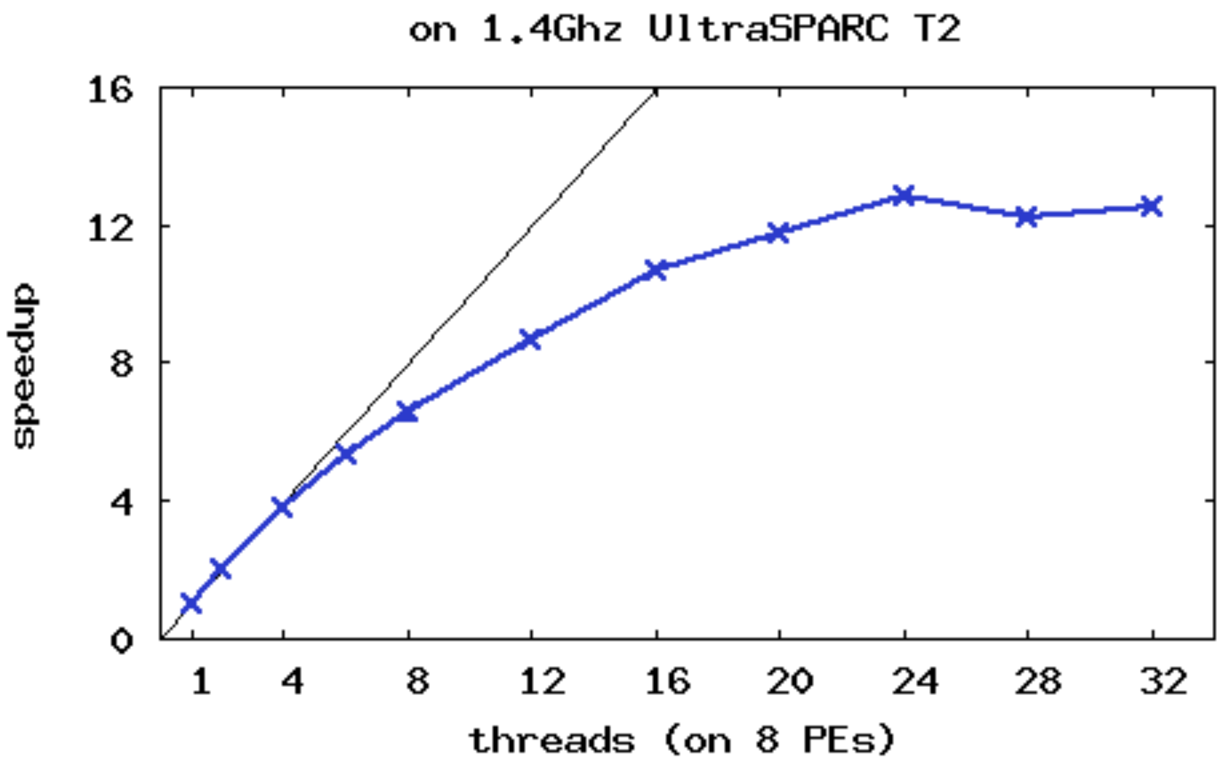


- GHC native code generator does no instruction reordering on SPARC. No LLVM ‘port.
- Single threaded on T2 is slow

2D Fast Fourier Transform (FFT)



	GCC	single thread	fastest parallel
Xenon	0.24	8.8s	2.0s
T2	2.4s	98s	7.7s



- C version is FFTW which uses in-place deep magic.
- Parallelism is no substitute for a better algorithm.

2D Fast Fourier Transform (FFT)

```
fft1D :: Array (sh:.Int) Double
      -> Array (sh:.Int) Double
      -> Array (sh:.Int) Double

fft1D rofu
 | n > 2  = (left +^ right) :+: (left -^ right)
 | n == 2 = traverse v id swivel
 where
  (_ :. n) = extent v
  swivel f (ix:.0) = f (ix:.0) + f (ix:.1)
  swivel f (ix:.1) = f (ix:.0) - f (ix:.1)

  rofu' = evenHalf rofu
  left  = force .            .fft1D rofu' .evenHalf $ v
  right = force .(*^ rofu).fft1D rofu' .oddHalf  $ v
```

Future work

- The examples we've presented are easy to write, but are cache naive.

Future work

- The examples we've presented are easy to write, but are cache naive.

- Using, block based matrix multiplication imposes a restriction on the order of evaluation...
  ... and makes it less obvious how to parallelise.

Future work

- The examples we've presented are easy to write,
  but are cache naive.

- Using, block based matrix multiplication imposes a restriction on the order of evaluation...
  ... and makes it less obvious how to parallelise.

- Repeated index computations can be expensive.
  GHC does not perform strength reduction on its loops.

Get it

http://trac.haskell.org/repa

- We depend on the the current version of the GHC head for decent performance.
- There will be a new release in a few weeks with GHC 7.0
- Send me your programs and I'll add them to our performance regression testsuite!

Questions?

Spare Slides

Shapes and Indices

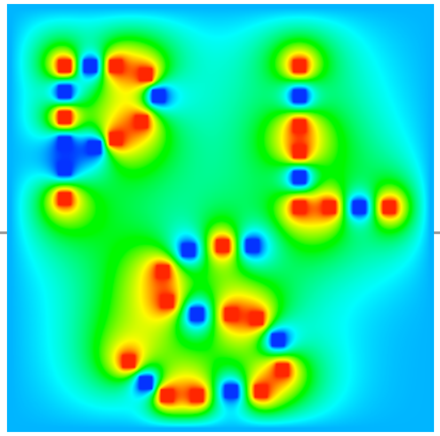
```
data Z              = Z
data tail :. head = tail :. head

type DIM0    = Z
type DIM1    = DIM0 :. Int
type DIM2    = DIM1 :. Int
...

class Shape sh where
  rank      :: sh -> Int
  size      :: sh -> Int
  toIndex   :: sh -> sh  -> Int
  fromIndex :: sh -> Int -> sh

instance             Shape Z          where ...
instance Shape sh => Shape (sh :. Int) where ...
```

Generic Traversal and the 3rd order function

```
traverse
  :: (Shape sh, Shape sh', Elt e)
  => Array sh e
  -> (sh -> sh')                   -- shape transform
  -> ((sh -> e) -> sh' -> e')      -- elem transform
  -> Array sh' e'
```

- A sane use for a third order function!
- Traverse takes a function to calculate the elements of the array.
- That function is passed a function to get elements of the source array.

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
```