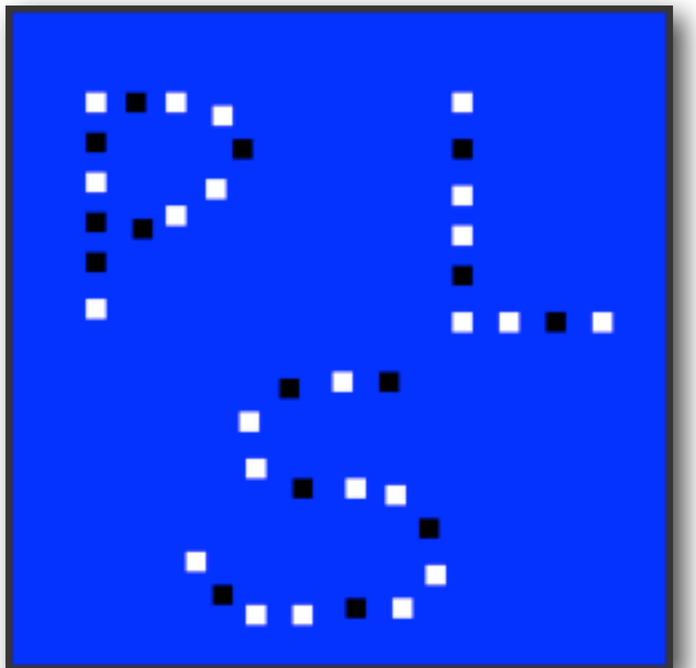
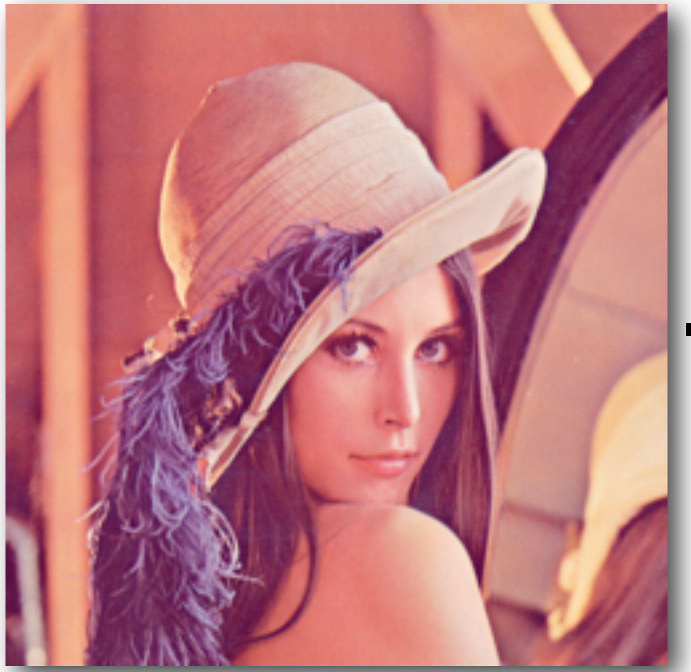
Practical Parallel Array Fusion with Repa

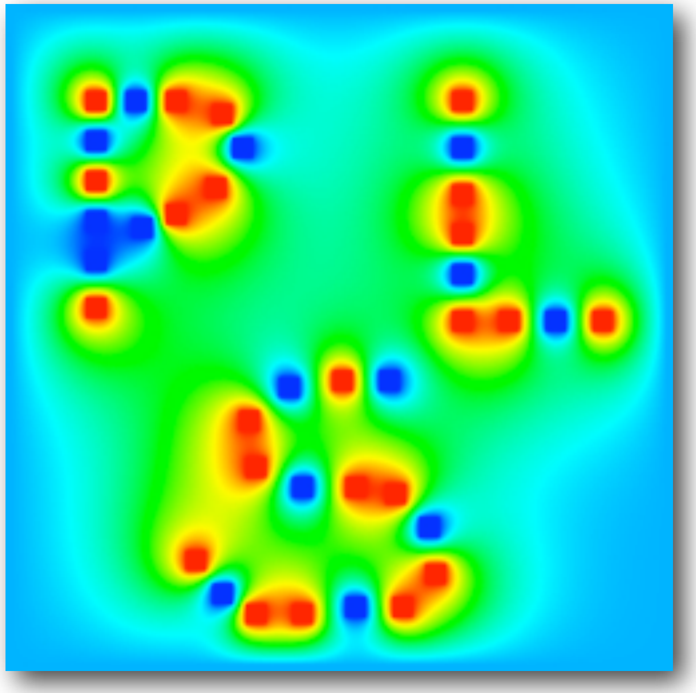
Ben Lippmeier
University of New South Wales
LambdaJam 2013

Flat Regular Data Parallelism



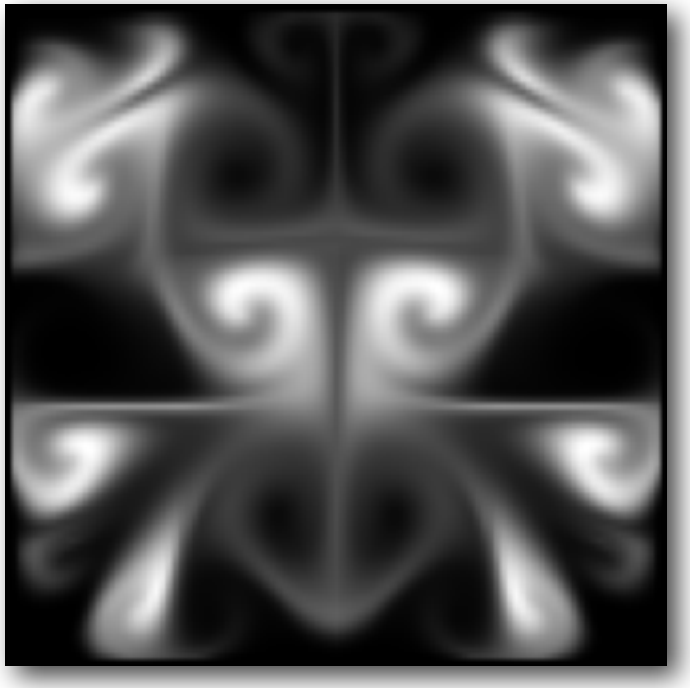


Edge Detection



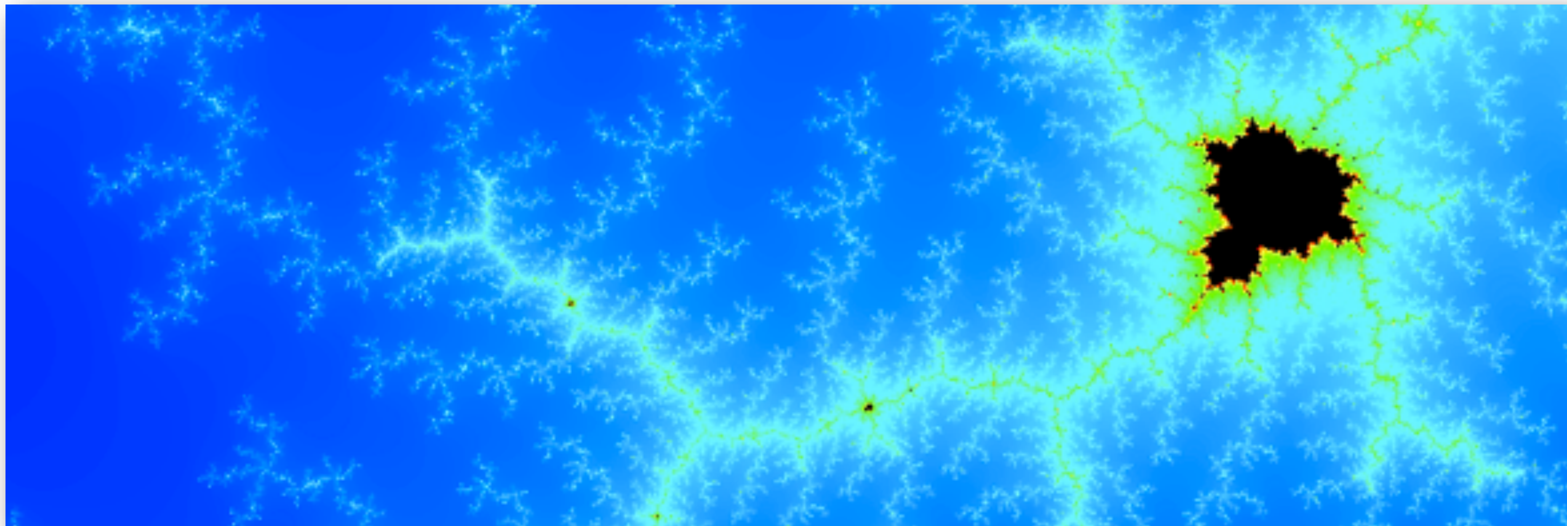
Matrix Relaxation

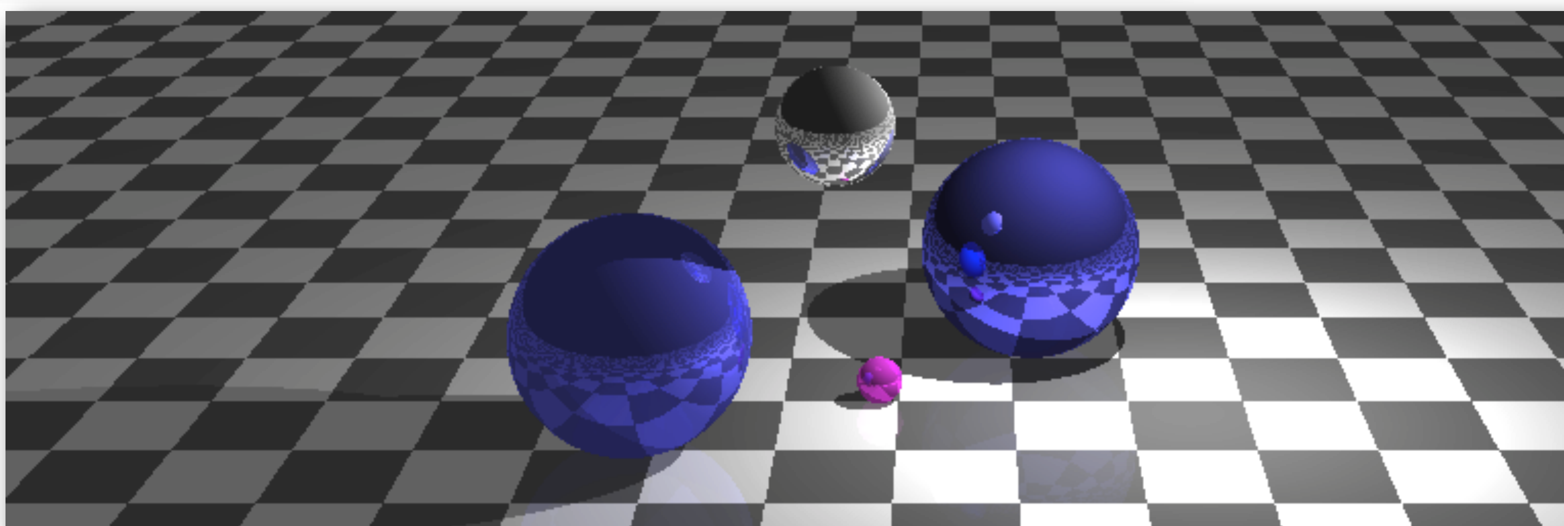
High Pass /w FFT

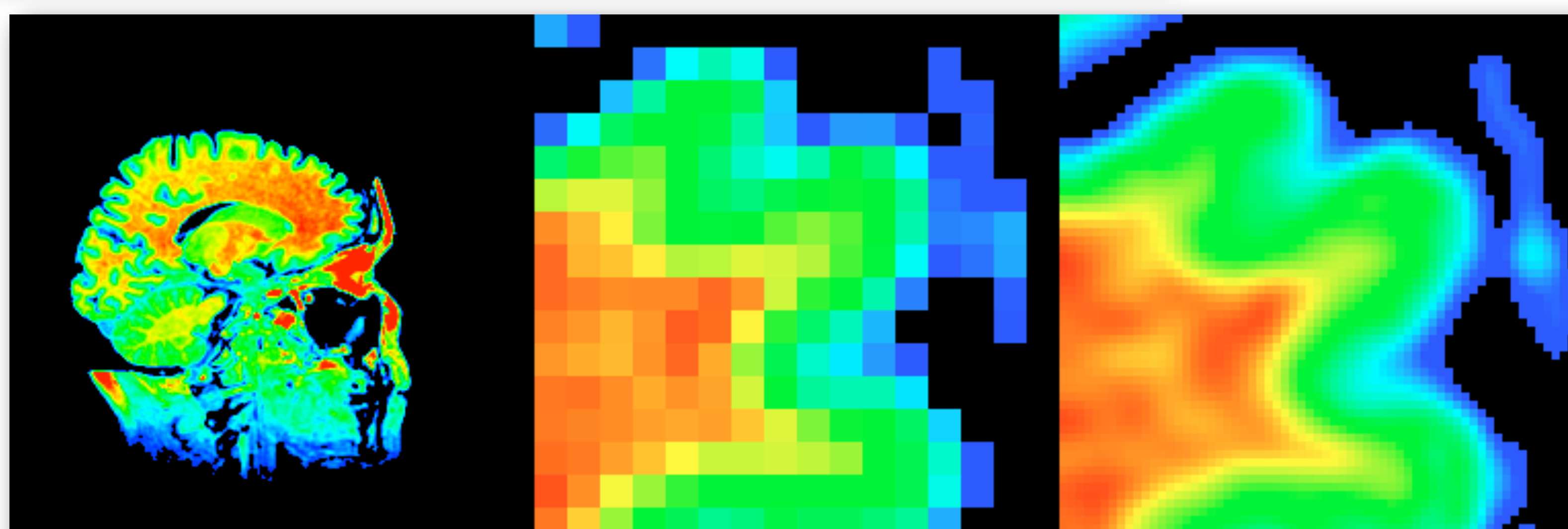


Fluid Simulation

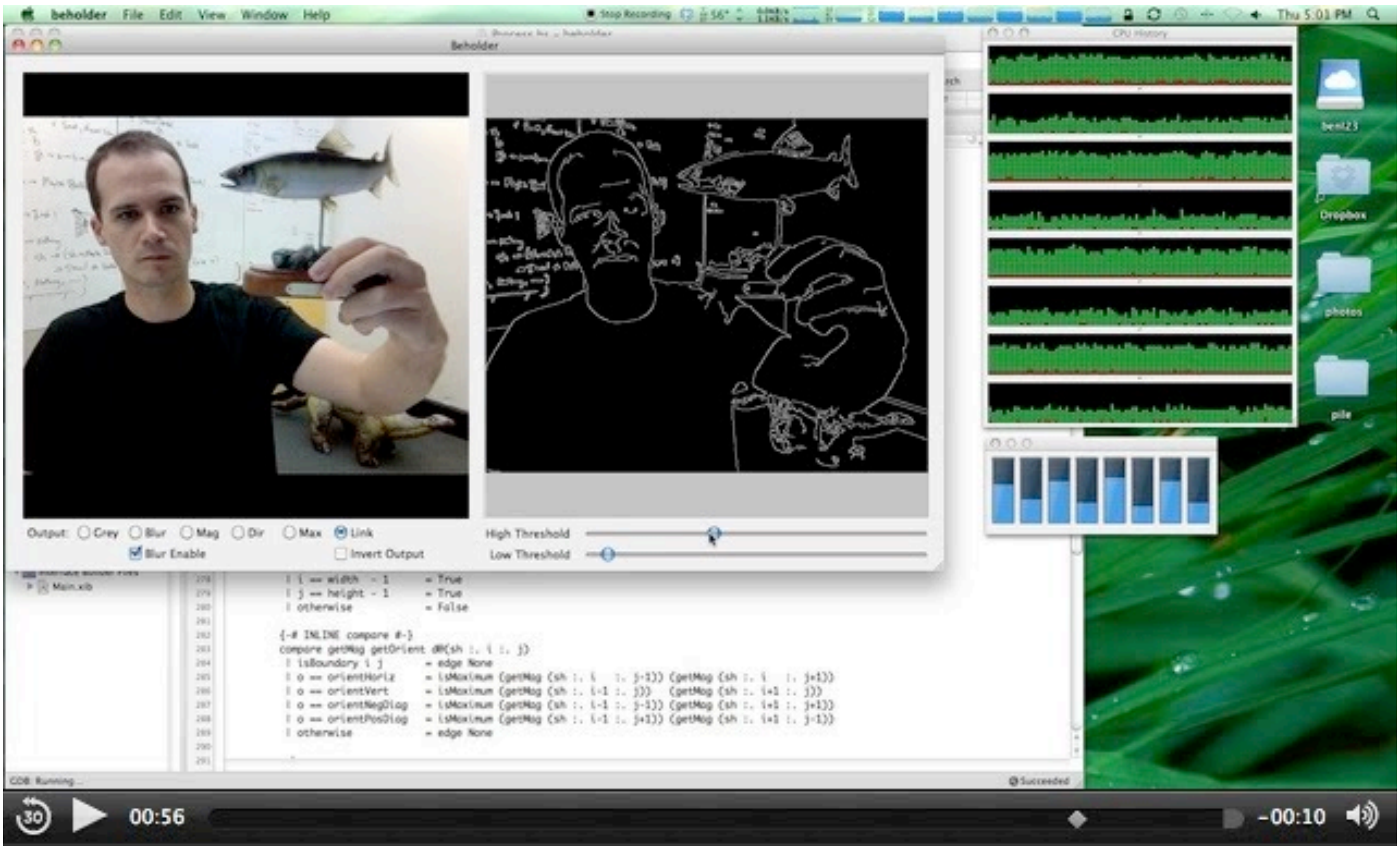
Unbalanced Flat Data Parallelism

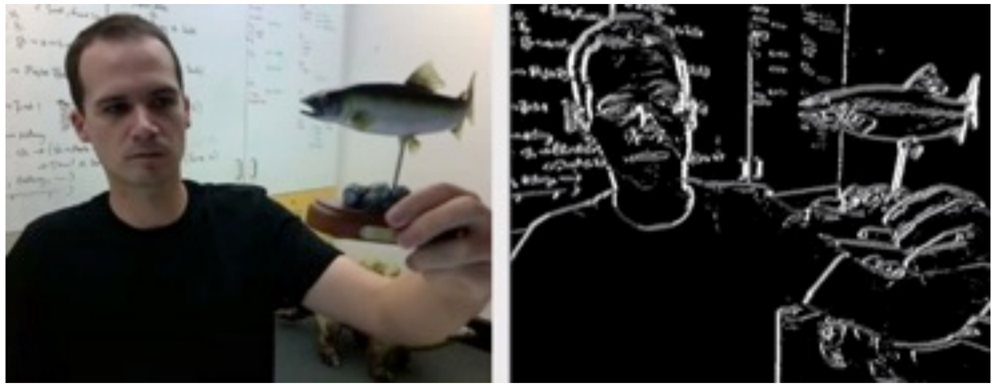




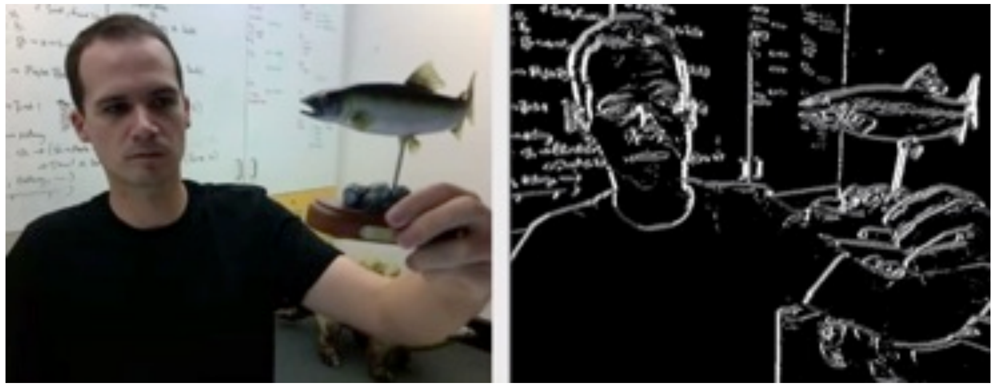


Edge Detection Demo





```haskell
import Data.Array.Repa  as R

type Image a = Array U DIM2 a

gradientMagOrient
    :: Float -> Image Float -> Image Float
    -> IO (Image (Float, Word8))

gradientMagOrient !threshLow dX dY
 = R.computeP $ R.zipWith magOrient dX dY
 where
    magOrient :: Float -> Float -> (Float, Word8)
    magOrient !x !y = (magnitude x y, orientation x y)
    {-# INLINE magOrient #-}

    magnitude :: Float -> Float -> Float
    magnitude !x !y = sqrt (x * x + y * y)
    {-# INLINE magnitude #-}

    orientation :: Float -> Float -> Word8
    orientation x y = ...
```



```
import Data.Array.Repa  as R

type Image a = Array U DIM2 a

gradientMagOrient
    :: Float -> Image Float -> Image Float
    -> IO (Image (Float, Word8))

gradientMagOrient !threshLow dX dY
 = R.computeP $ R.zipWith magOrient dX dY
 where
    magOrient :: Float -> Float -> (Float, Word8)
    magOrient !x !y = (magnitude x y, orientation x y)
    {-# INLINE magOrient #-}

    magnitude :: Float -> Float -> Float
    magnitude !x !y = sqrt (x * x + y * y)
    {-# INLINE magnitude #-}

    orientation :: Float -> Float -> Word8
    orientation x y = ...
```

How can you tell if someone is a Mathematician or a `Computer Scientist`?

Hermann Grassmann

Seymour Cray

How can you tell if someone is a Mathematician or a `Computer Scientist`?

Hermann Grassmann

Seymour Cray

Ask them what a vector is

```
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e
```

```haskell
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e


data D
```

```haskell
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e


data D

instance Source D e where
 data Array D sh e
    = ADelayed !sh (sh -> a)
```

```
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e


data D

instance Source D e where
 data Array D sh e
     = ADelayed !sh (sh -> a)
```

Function!

```haskell
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e


data D

instance Source D e where
 data Array D sh e
    = ADelayed !sh (sh -> a)

 extent (ADelayed sh _)       = sh
 index  (ADelayed sh get) ix = get ix
```

```
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e


data U
```

```haskell
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e


data U

instance Unbox e => Source U e where
 data Array U sh e
    = AUnboxed !sh (U.Vector e)
```

```haskell
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e


data U

instance Unbox e => Source U e where
 data Array U sh e
    = AUnboxed !sh (U.Vector e)

 extent (AUnboxed sh _)       = sh
 index  (AUnboxed sh uvec) ix
  = uvec `U.index` (Shape.toLinearIndex sh ix)
```

```
class Source r e where
 data Array r sh e
 extent :: Array r sh e -> sh
 index  :: Array r sh e -> sh -> e

class Shape sh where
 rank               :: sh -> Int
 toLinearIndex      :: sh -> sh  -> Int
 fromLinearIndex :: sh -> Int -> sh
 ...

instance Shape DIM1 where ...
instance Shape DIM2 where ...

type DIM1 = Z :. Int                          (Z :. 5)
type DIM2 = Z :. Int :. Int             (Z :. 10 :. 16)
```

Important!

Delayed arrays are functions!

```
data D
instance Source D e where
 data Array D sh e
  = ADelayed !sh (sh -> a)
```

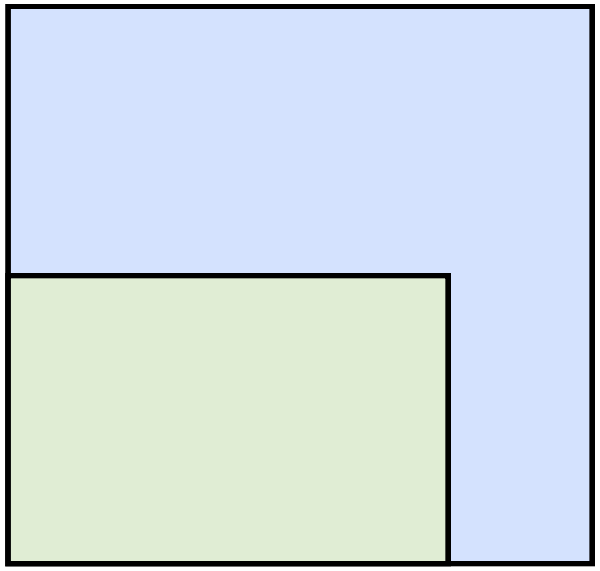
Unboxed arrays are real data!

```
data U
instance Unbox e => Source U e where
 data Array U sh e
  = AUnboxed !sh (U.Vector e)
```

```haskell
map  :: (Shape sh, Source r a)
     => (a -> b)
     -> Array r sh a -> Array D sh b

map f arr
 = ADelayed (extent arr)
            (\ix -> f (arr `index` ix))
```

```haskell
zipWith :: (Shape sh, Source r1 a, Source r2 a)
        => (a -> b -> c)
        -> Array r1 sh a -> Array r2 sh b
        -> Array D  sh c

zipWith f arr1 arr2
 = ADelayed
    (intersectDim (extent arr1) (extent arr2))
    (\ix -> f (arr1 `index` ix) (arr2 `index` ix))
```

```haskell
zipWith :: (Shape sh, Source r1 a, Source r2 a)
        => (a -> b -> c)
        -> Array r1 sh a -> Array r2 sh b
        -> Array D  sh c

zipWith f arr1 arr2
 = ADelayed
    (intersectDim (extent arr1) (extent arr2))
    (\ix -> f (arr1 `index` ix) (arr2 `index` ix))
```



```
example :: Array D DIM2 Int
example
 = map f (zipWith g arr1 arr2)
```

```
example :: Array D DIM2 Int
example
 = map f (zipWith g arr1 arr2)
```

```
zipWith f arr1 arr2
 = ADelayed
    (intersectDim (extent arr1) (extent arr2))
    (\ix -> f (arr1 `index` ix) (arr2 `index` ix))
```

```haskell
example :: Array D DIM2 Int
example
 = map f (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix -> g (arr1 !! ix) (arr2 !! ix)))
```

```haskell
zipWith f arr1 arr2
 = ADelayed
    (intersectDim (extent arr1) (extent arr2))
    (\ix -> f (arr1 `index` ix) (arr2 `index` ix))
```

```
example :: Array D DIM2 Int
example
 = map f (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix -> g (arr1 !! ix) (arr2 !! ix)))
```

```
example :: Array D DIM2 Int
example
 = let sh' =
       g'  =
   in  map f (ADelayed (intersectDim (extent arr1) (extent arr2))
                       (\ix -> g (arr1 !! ix) (arr2 !! ix)))
```

```haskell
example :: Array D DIM2 Int
example
 = let sh'  = intersectDim (extent arr1) (extent arr2)
       g'   = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  map f (ADelayed (                                          )
                       (                                     ))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  map f (ADelayed sh‘ g’)
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  map f (ADelayed sh' g')
```

```
map f arr
 = ADelayed (extent arr)
            (\ix -> f (arr `index` ix))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (extent (ADelayed sh' g'))
                (\ix2 -> f (ADelayed sh' g' !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (extent (ADelayed sh’ g’))
                (\ix2 -> f (ADelayed sh’ g’ !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (extent (ADelayed sh' g'))
                (\ix2 -> f (ADelayed sh' g' !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh'  = intersectDim (extent arr1) (extent arr2)
       g'   = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh'
                (\ix2 -> f (ADelayed sh' g' !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh'
                (\ix2 -> f (ADelayed sh' g' !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (ADelayed sh’ g’ !! ix2))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed sh’
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh' = intersectDim (extent arr1) (extent arr2)
       g'  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 = let sh’ = intersectDim (extent arr1) (extent arr2)
       g’  = \ix -> g (arr1 !! ix) (arr2 !! ix)
   in  ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example :: Array D DIM2 Int
example
 =     ADelayed (intersectDim (extent arr1) (extent arr2))
                (\ix2 -> f (g (arr1 !! ix2) (arr2 !! ix2)))
```

```
example' :: Array D DIM2 Int
example'
 = map (+ 1) (zipWith (*) arr1 arr2))

 = ADelayed (intersectDim (extent arr1) (extent arr2))
            (\ix2 -> (+ 1) ((*) (arr1 !! ix2) (arr2 !! ix2))
```

```
example' :: Array D DIM2 Int
example'
 = map (+ 1) (zipWith (*) arr1 arr2))

 = ADelayed (intersectDim (extent arr1) (extent arr2))
            (\ix2 -> (+ 1) ((*) (arr1 !! ix2) (arr2 !! ix2))
```

```
example' :: Array D DIM2 Int
example'
 = map (+ 1) (zipWith (*) arr1 arr2))

 = ADelayed (intersectDim (extent arr1) (extent arr2))
            (\ix2 -> (arr1 !! ix2) * (arr2 !! ix2) + 1)
```

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```haskell
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```haskell
computeP arr
 = ...
   ...



 where
    fill !lix !end
    | lix >= end       = return ()
    | otherwise
    = do write lix
           (arr `index` fromLinearIndex lix)
         fill (lix + 1) end
    ...
```

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```
computeP (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix2 -> (arr1 !! ix2) * (arr2 !! ix2) + 1))
 = ...
   ...


 where
    fill !lix !end
    | lix >= end       = return ()
    | otherwise
    = do write lix
          (arr `index` fromLinearIndex lix)
         fill (lix + 1) end
    ...
```

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```
computeP (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix2 -> (arr1 !! ix2) * (arr2 !! ix2) + 1))
 = ...
   ...


 where
    fill !lix !end
    | lix >= end          = return ()
    | otherwise
    = do write lix
          (arr `index` fromLinearIndex lix)
         fill (lix + 1) end
    ...
```

```
computeP :: Array D sh a -> Array U sh a
```

(not the whole story)

```
computeP (ADelayed (intersectDim (extent arr1) (extent arr2))
                   (\ix2 -> (arr1 !! ix2) * (arr2 !! ix2) + 1))
 = ...
   ...


 where
    fill !lix !end
    | lix >= end          = return ()
    | otherwise
    = do write lix
          (let ix' = fromLinearIndex lix
           in  (arr1 !! ix') * (arr2 !! ix') + 1)
         fill (lix + 1) end
    ...
```

Delayed arrays

```
data D
instance Source D e where
  data Array D sh e = ADelayed !sh (sh -> a)
```

Unboxed arrays

```
data U
instance Unbox e => Source U e where
  data Array U sh e = AUnboxed !sh !(U.Vector e)
```

Delayed arrays

```
data D
instance Source D e where
  data Array D sh e = ADelayed !sh (sh -> a)
```

Unboxed arrays

```
data U
instance Unbox e => Source U e where
  data Array U sh e = AUnboxed !sh !(U.Vector e)
```

Strict ByteStrings

```
data B
instance Source B Word8 where
  data Array B sh Word8 = AByteString !sh !ByteString
```

Delayed arrays

```
data D
instance Source D e where
 data Array D sh e = ADelayed !sh (sh -> a)
```

Unboxed arrays

```
data U
instance Unbox e => Source U e where
 data Array U sh e = AUnboxed !sh !(U.Vector e)
```

Strict ByteStrings

```
data B
instance Source B Word8 where
 data Array B sh Word8 = AByteString !sh !ByteString
```

Cursor Functions

```
data C
instance Source C e where
 data Array C sh e = ACursored ...
```

```
computeP :: (Load r1 sh e, Target r2 e, Monad m)
         => Array r1 sh e -> m (Array r2 sh e)
```

```haskell
computeP :: (Load r1 sh e, Target r2 e, Monad m)
         => Array r1 sh e -> m (Array r2 sh e)
```

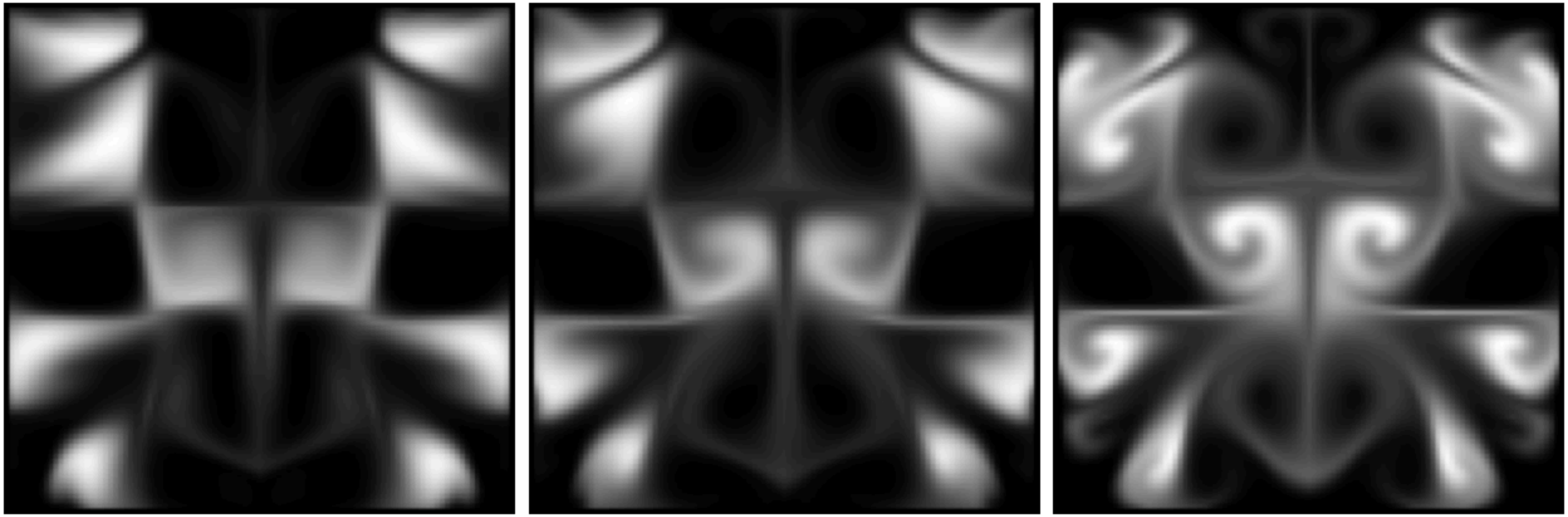
```haskell
class Target r e where
 data MVec r e
 newMVec    :: Int -> IO (MVec r e)
 writeMVec  :: MVec r e -> Int -> e -> IO ()
 freezeMVec :: sh -> MVec r e -> IO (Array r sh e)
```

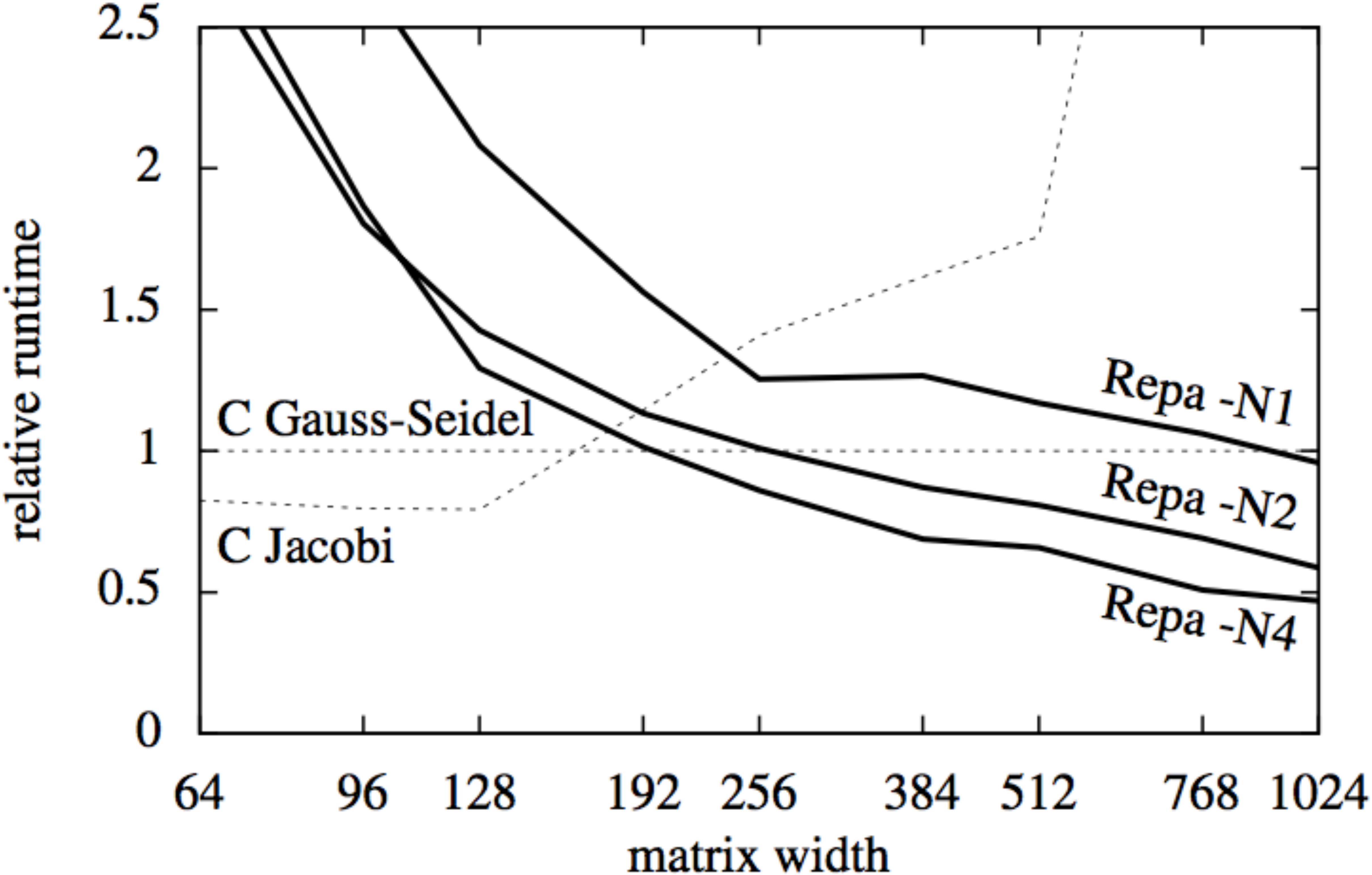
```haskell
class (Source r1 e, Shape sh)
    => Load r1 sh e where
 loadS :: Target r2 e => Array r1 sh e -> MVec r2 e -> IO ()
 loadP :: Target r2 e => Array r1 sh e -> MVec r2 e -> IO ()
```

Foreign Buffers

```
data F

instance Storable e => Source F e where
 data Array F e
  = AForeignPtr !sh !Int !(ForeignPtr e)

instance Target F e where
 data MVec F e
  = FPVec !Int !(ForeignPtr e)
 ...


computeIntoP
    :: (Load r1 sh e, Storable e)
    => ForeignPtr e -> Array r1 sh e -> IO ()

computeIntoP !fptr !arr
    = loadP arr (FPVec 0 fptr)
```

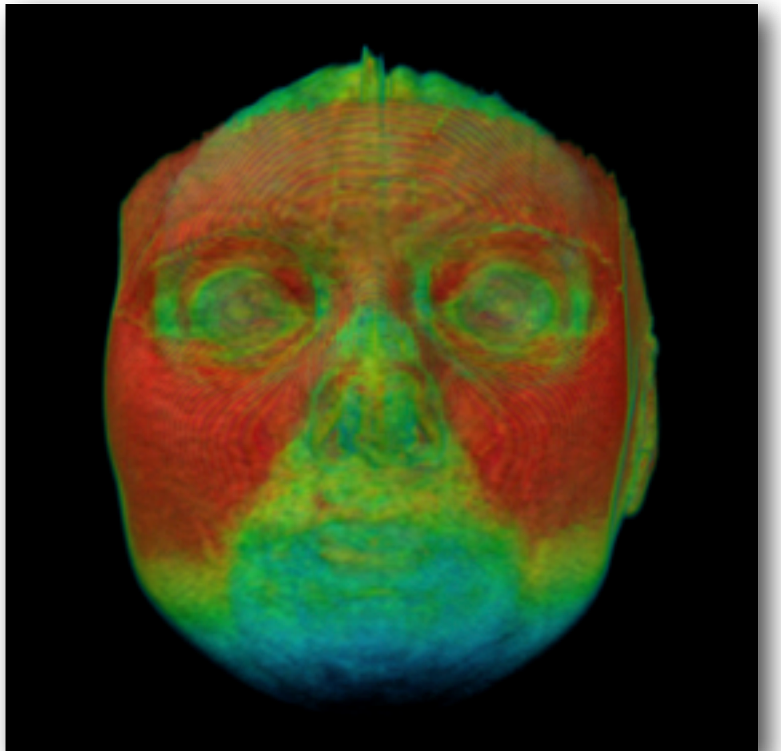
Fluid Flow Benchmark

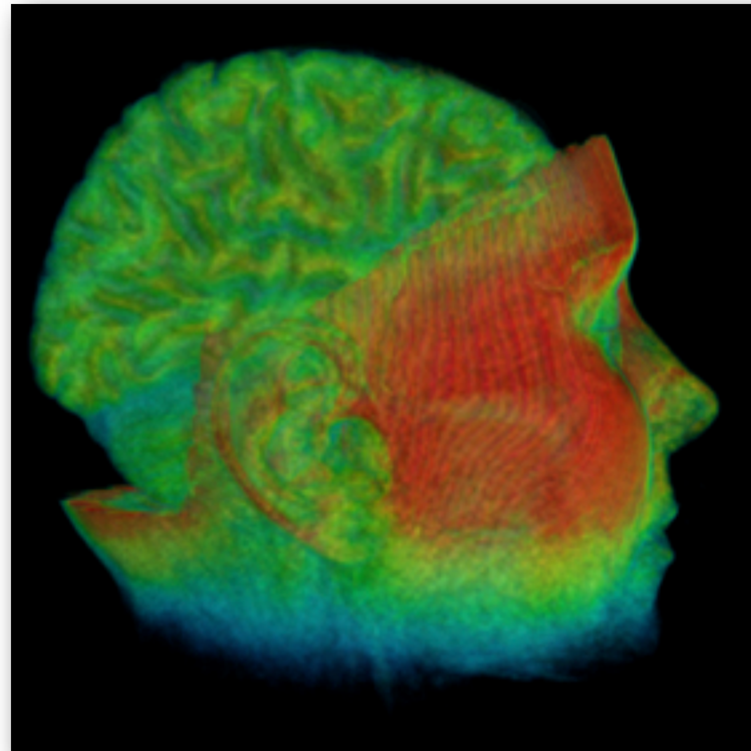


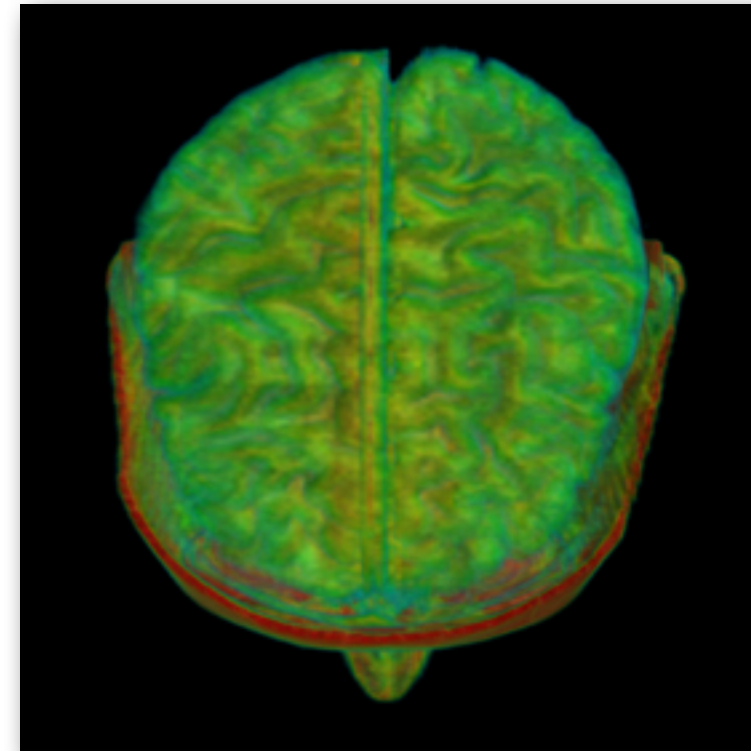


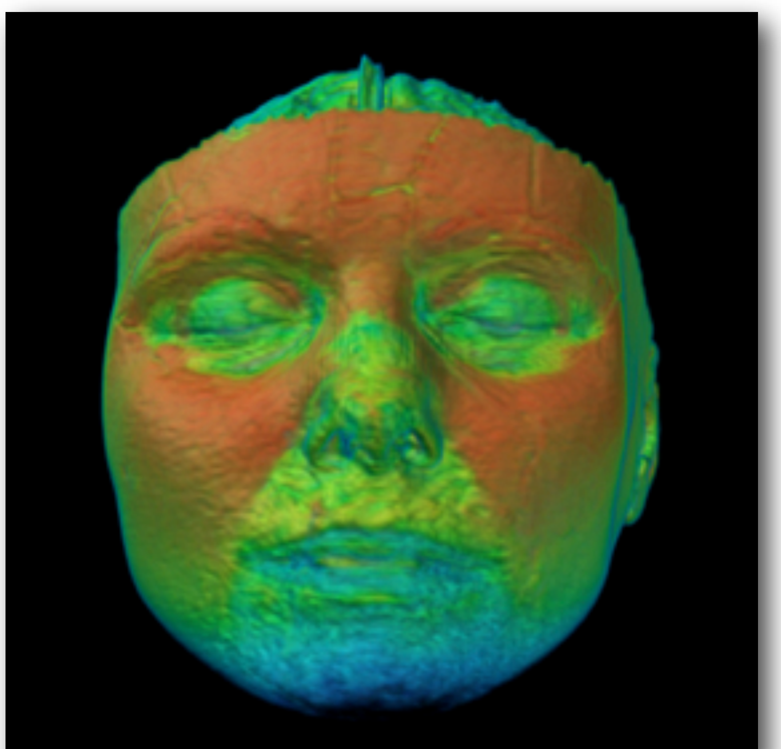
2.5
2
1.5
1
0.5
0
relative runtime
64
96
128
192
256
384
512
768
1024
matrix width
C Gauss-Seidel
C Jacobi
Repa -N1
Repa -N2
Repa -N4

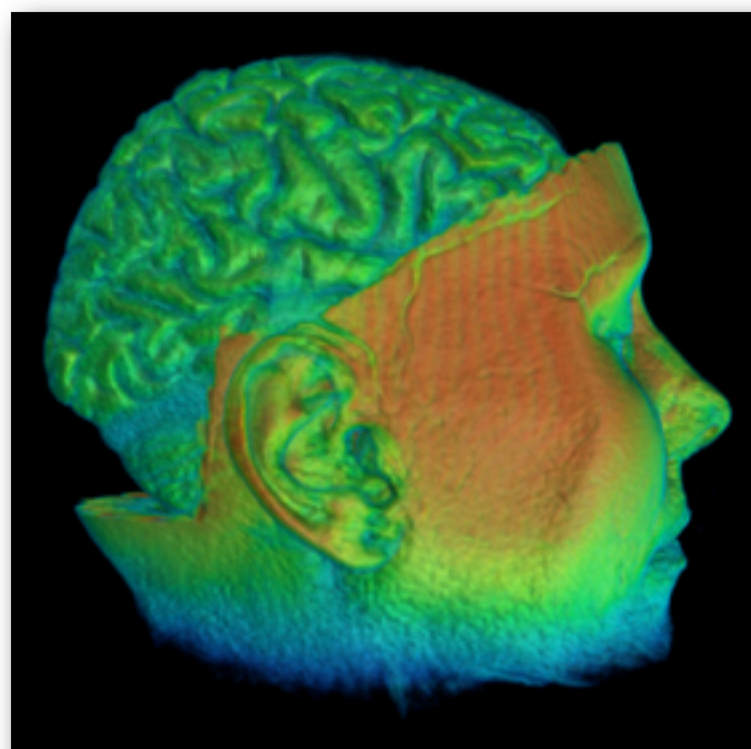
Volumetric Interpolation by Michael Orlitzky

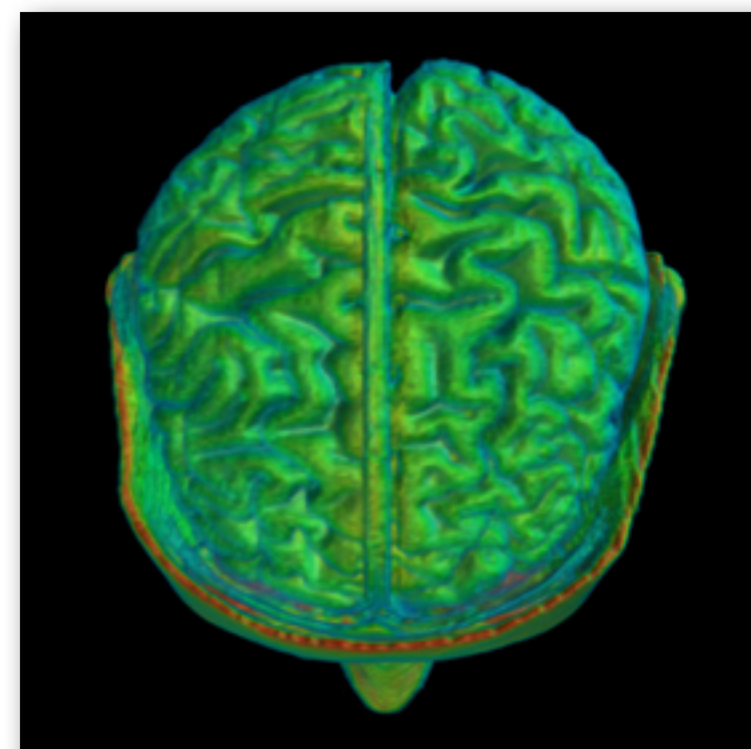

original


original

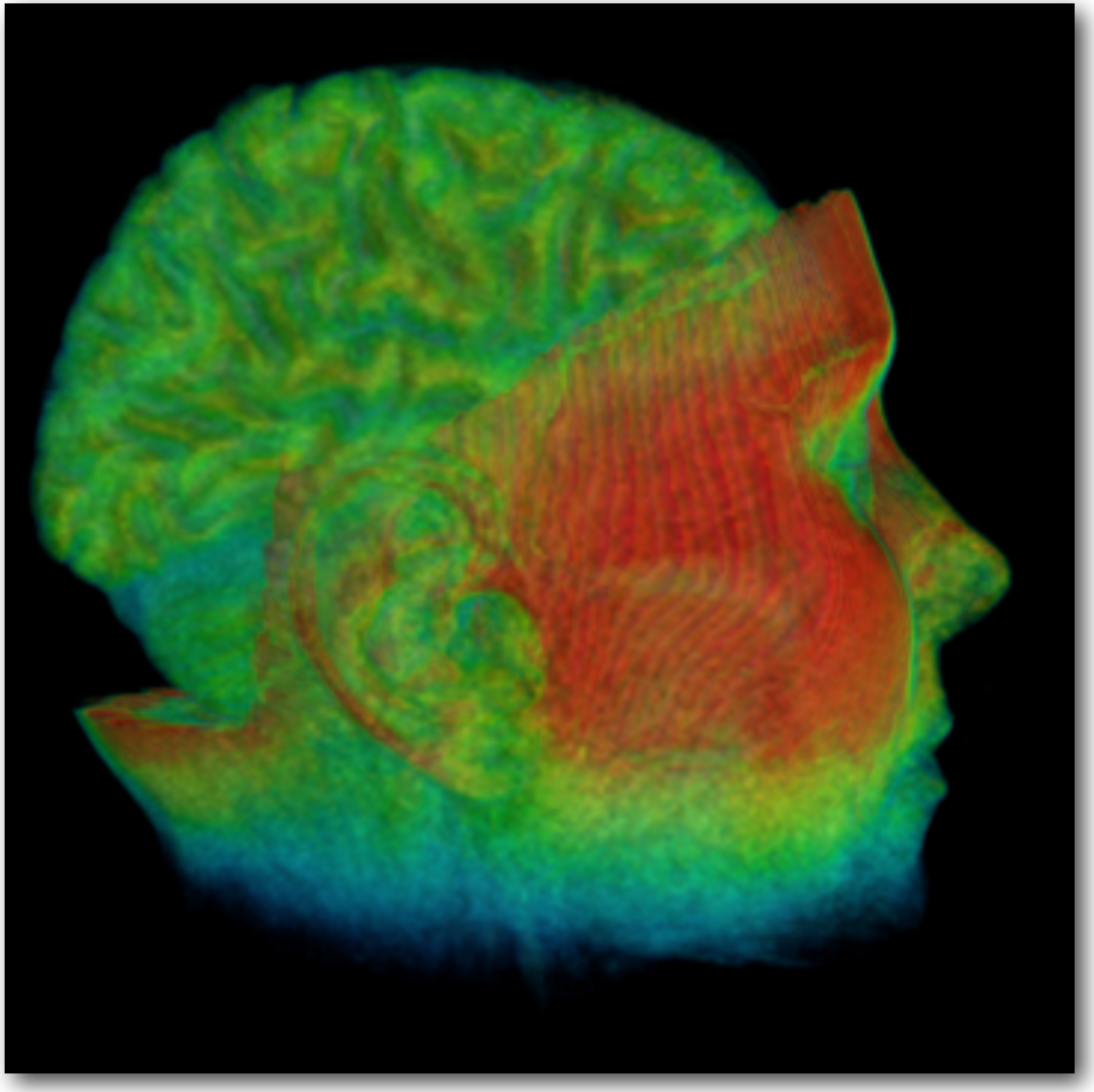

original

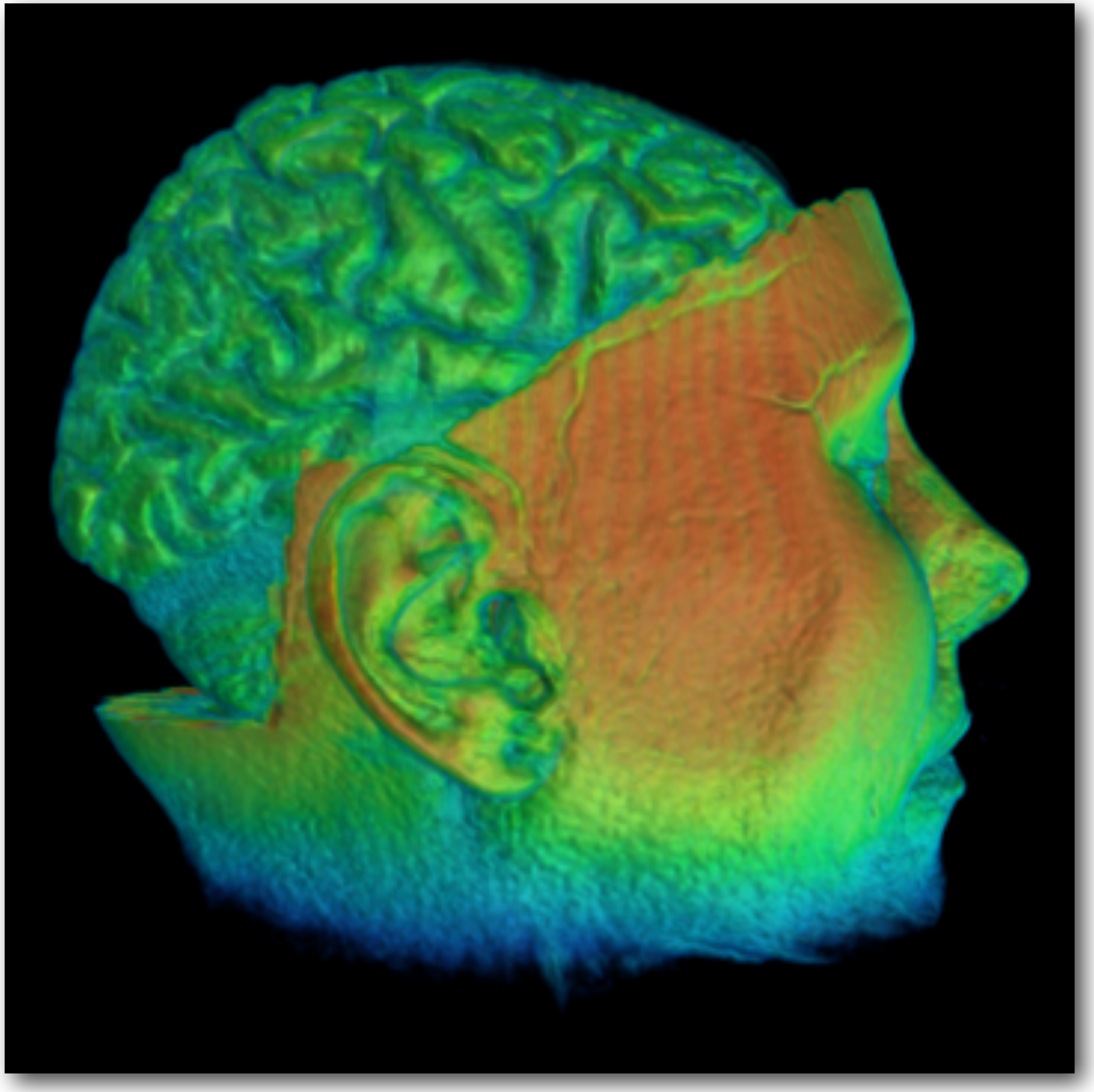

4x4 scale


4x4 scale


4x4 scale

Volumetric Interpolation by Michael Orlitzky





Indexing overhead

```
zipWith :: (Shape sh, Source r1 a, Source r2 a)
        => (a -> b -> c)
        -> Array r1 sh a -> Array r2 sh b
        -> Array D  sh c

zipWith f arr1 arr2
 = ADelayed
    (intersectDim (extent arr1) (extent arr2))
    (\ix -> f (arr1 `index` ix) (arr2 `index` ix))
```

Repeated conversion of possibly high-rank index to flat linear index.

```
offset = x + y * height
```

Questions?

Index Space Transforms and Matrix Transposition

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

Replicate and Matrix Multiplication

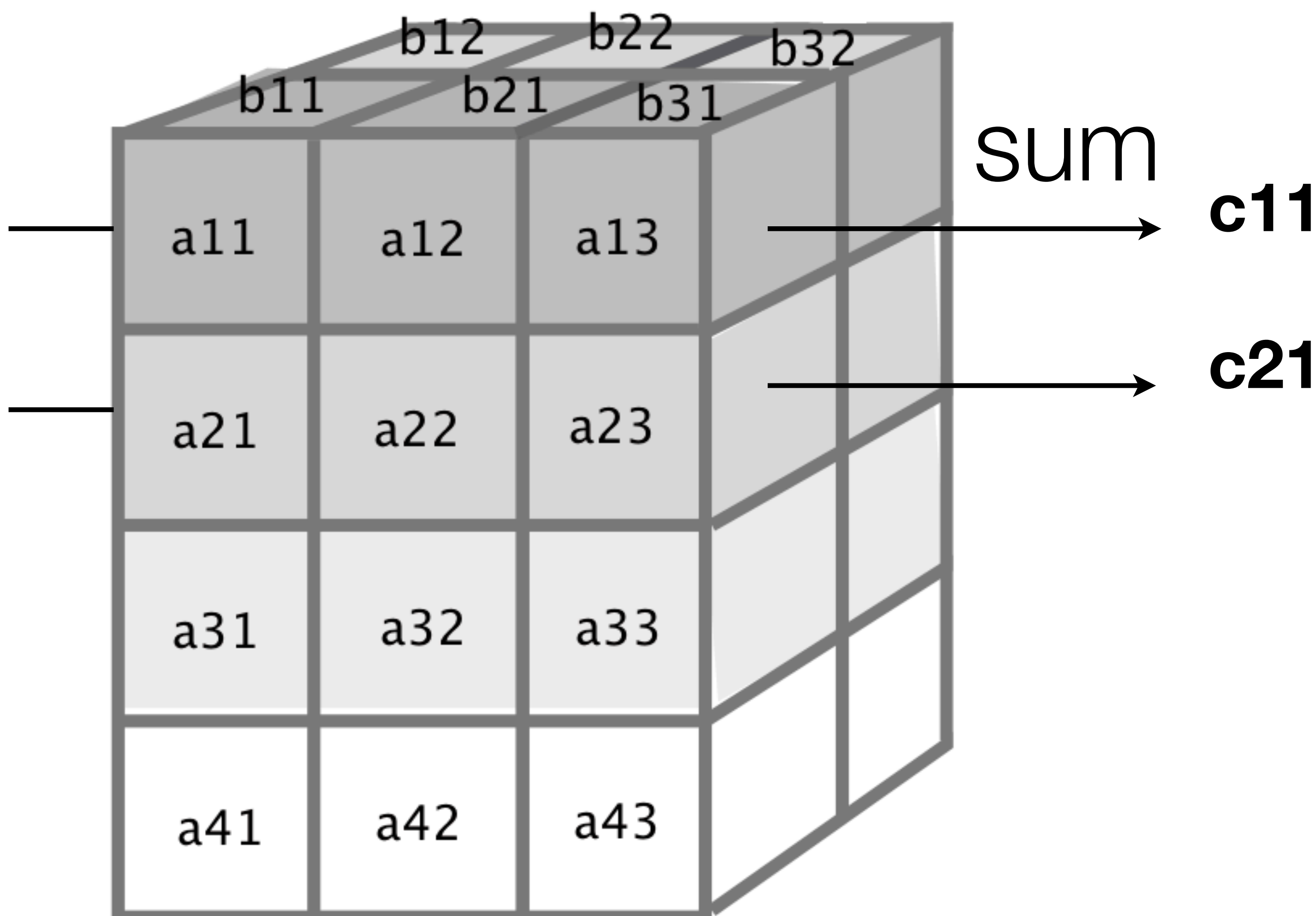
Matrix Multiplication

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

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

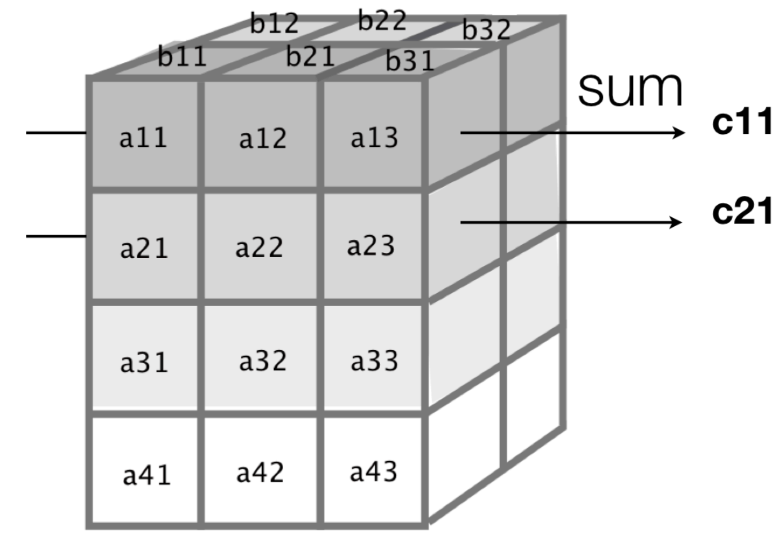
Matrix Multiplication

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



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

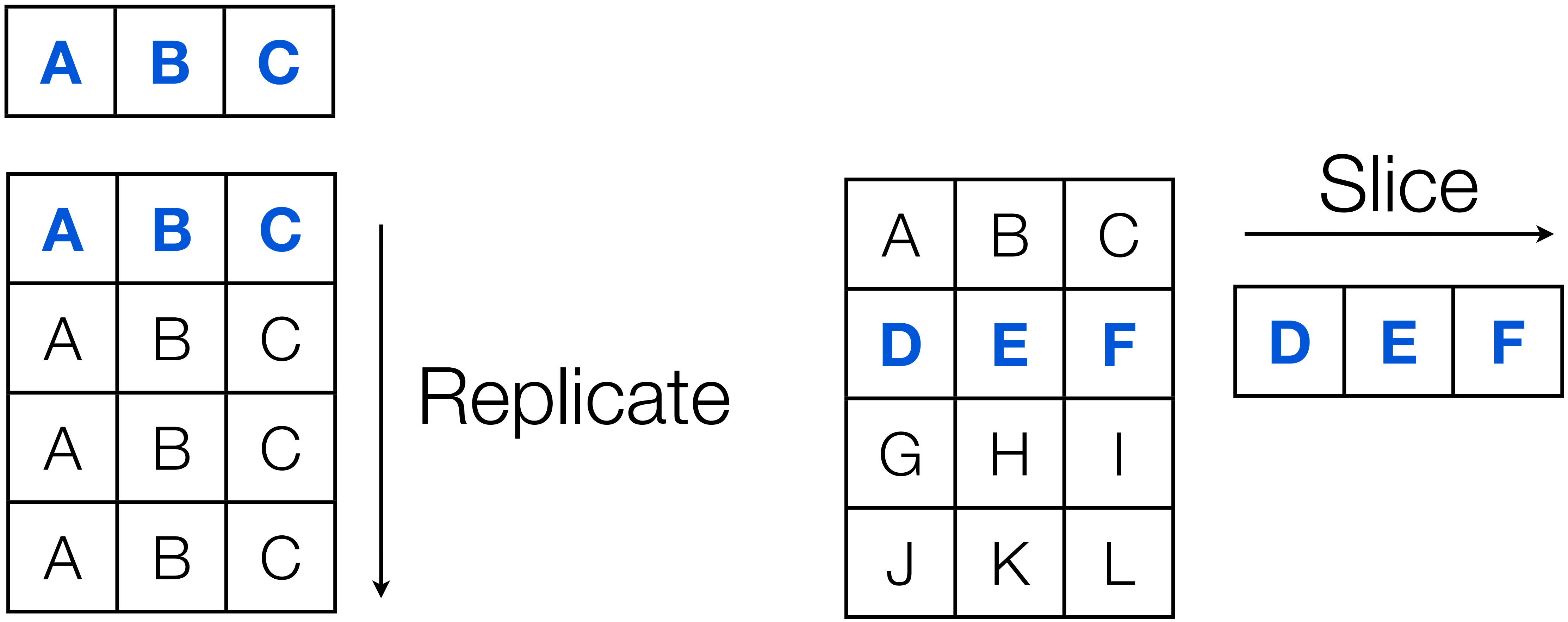
Matrix Multiplication




```
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
```

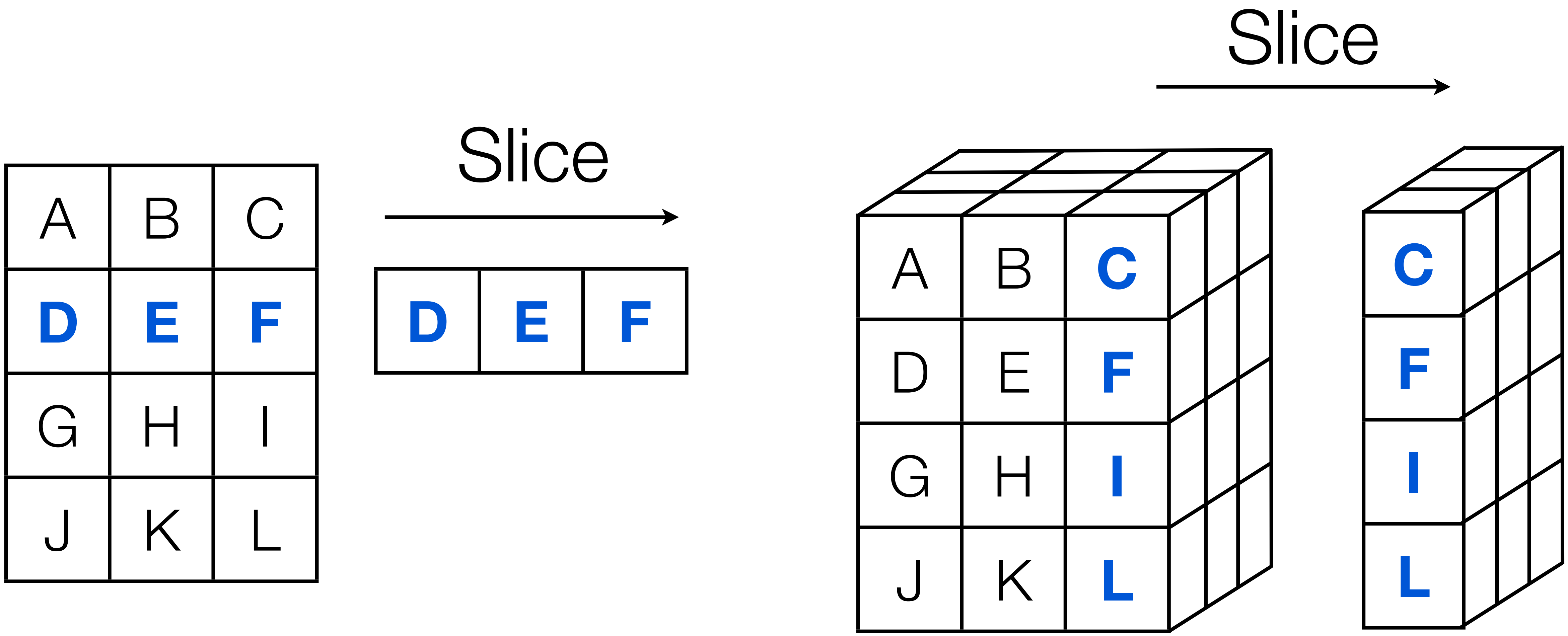
High Rank Replicate and Slice

Replicate and Slice are duals.



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

Type hackery




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

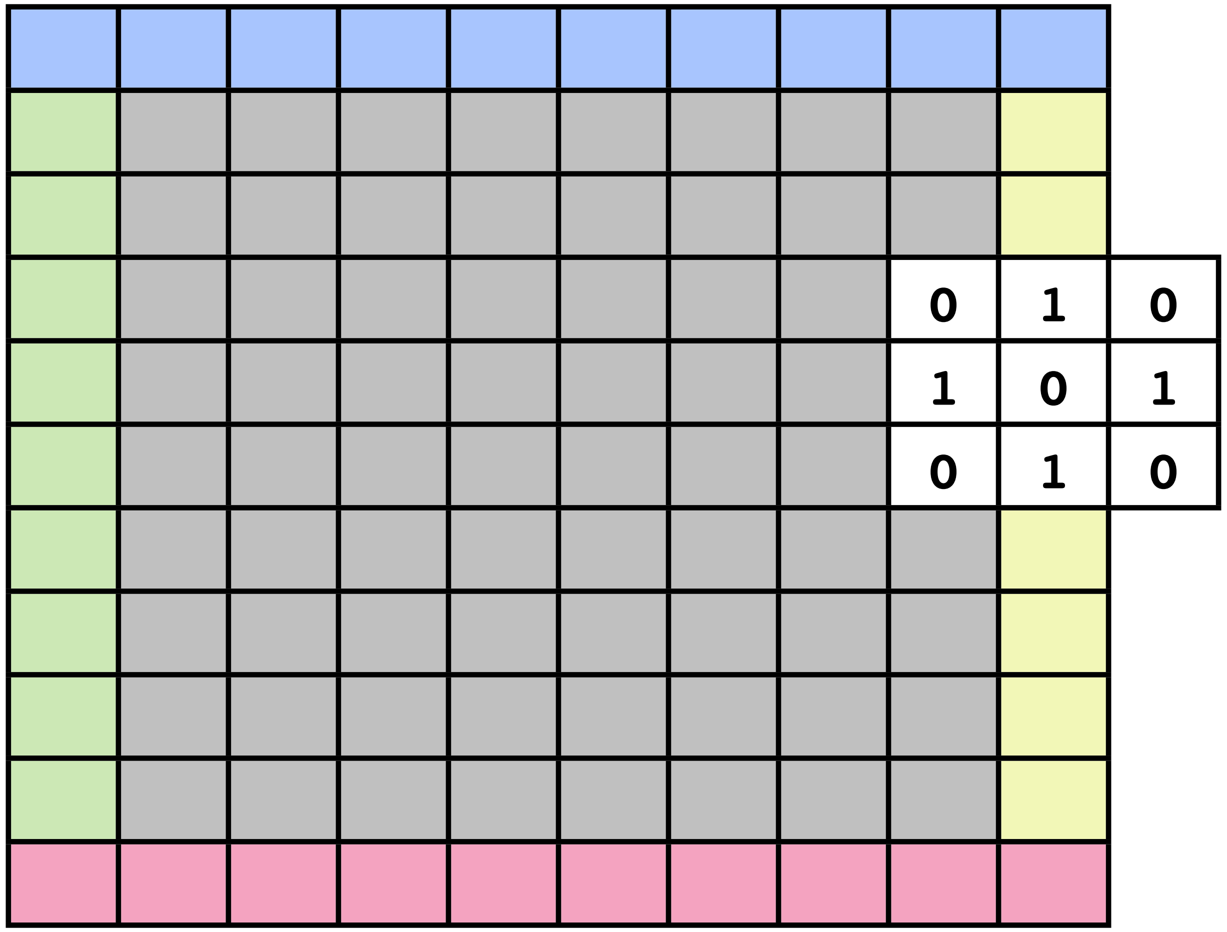
Partitioned Arrays and Smallness hints

cursored arrays (delayed)

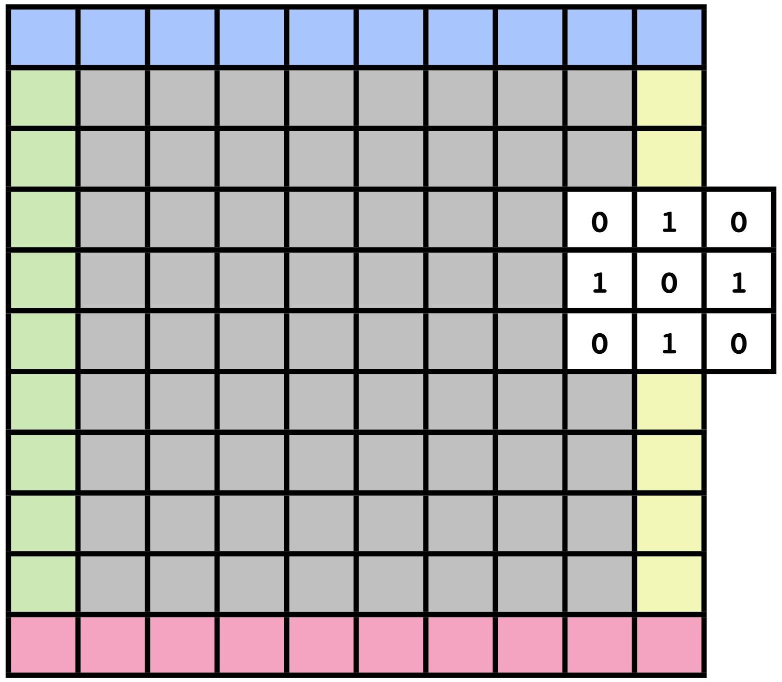
```
data C
data instance Array C sh e = ...
```

partitioned arrays (meta)

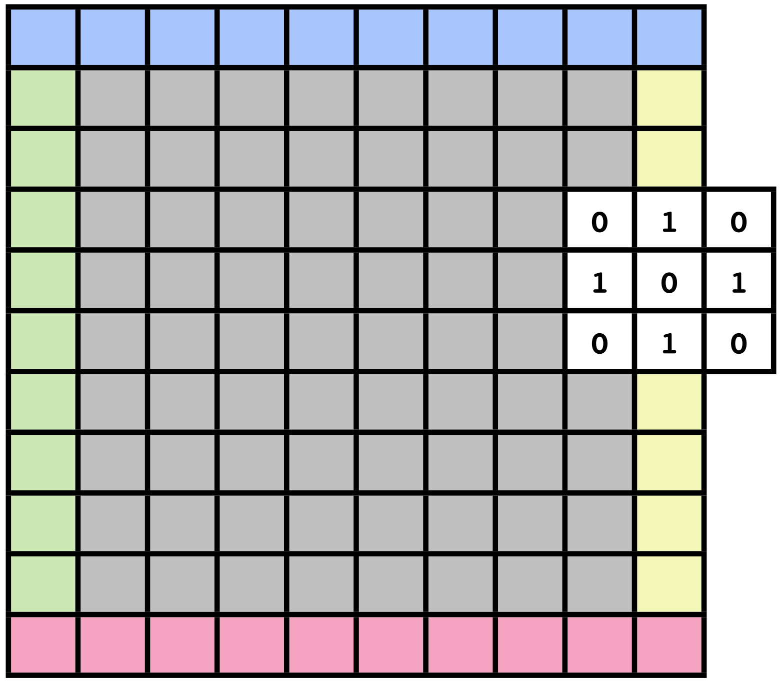
```
data P r1 r2
data instance Array (P r1 r2) sh e = ...
```



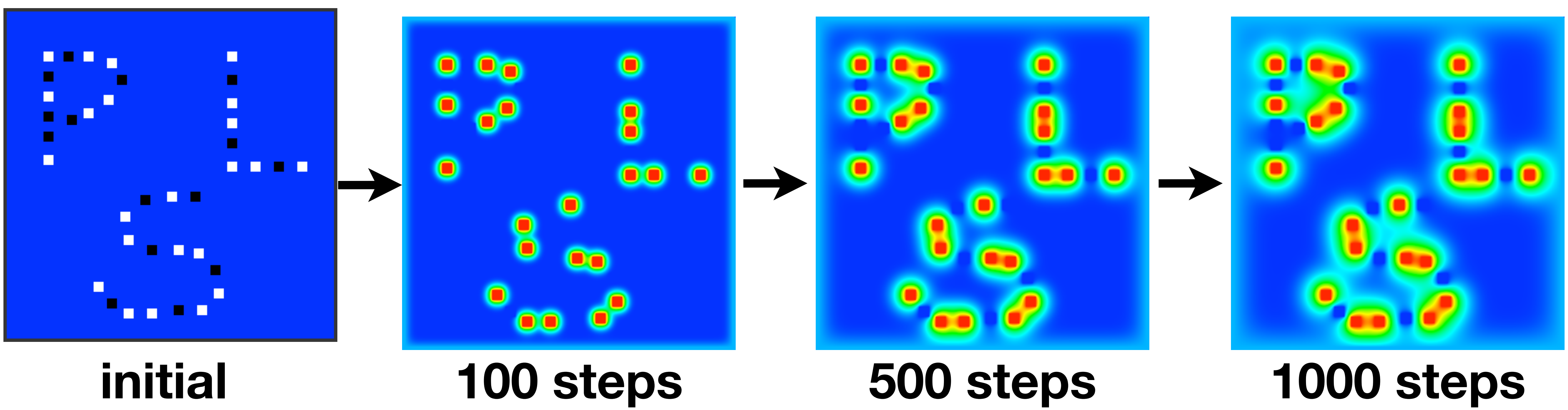

```
Array (P C (P D (P D (P D D)))) sh e
```

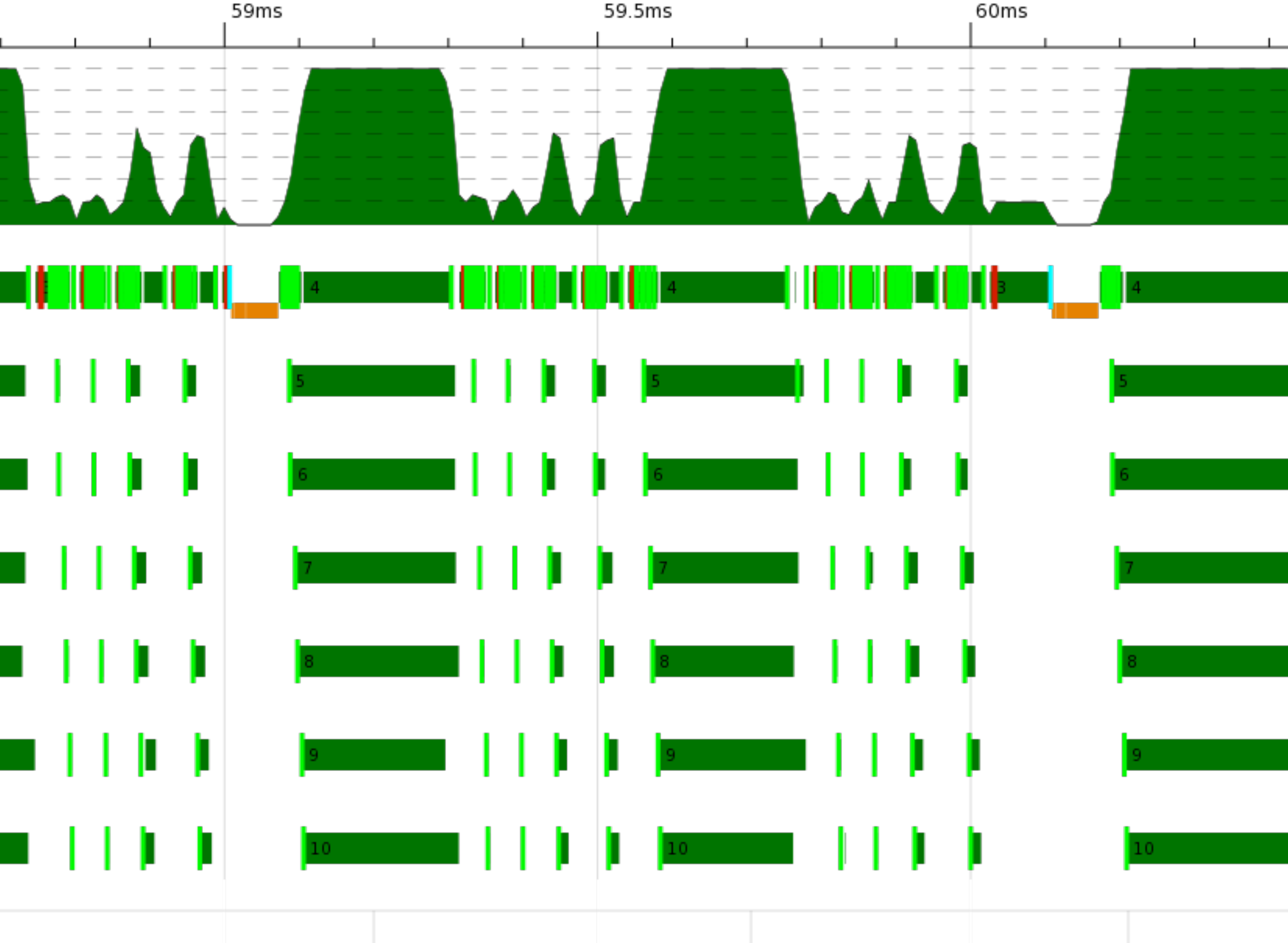


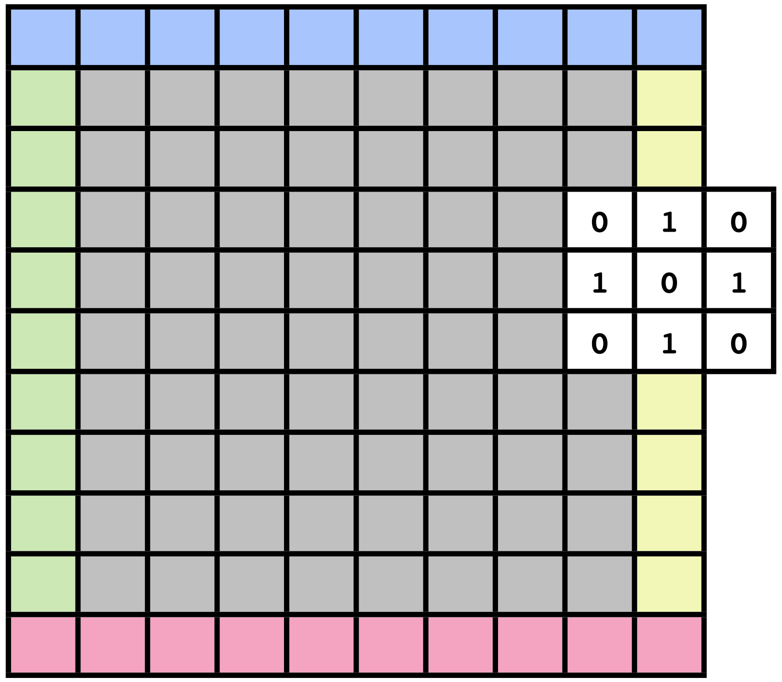
Array (P C (P D (P D (P D D)))) sh e




```
Array (P C (P D (P D (P D D)))) sh e
```




59ms
59.5ms
60ms

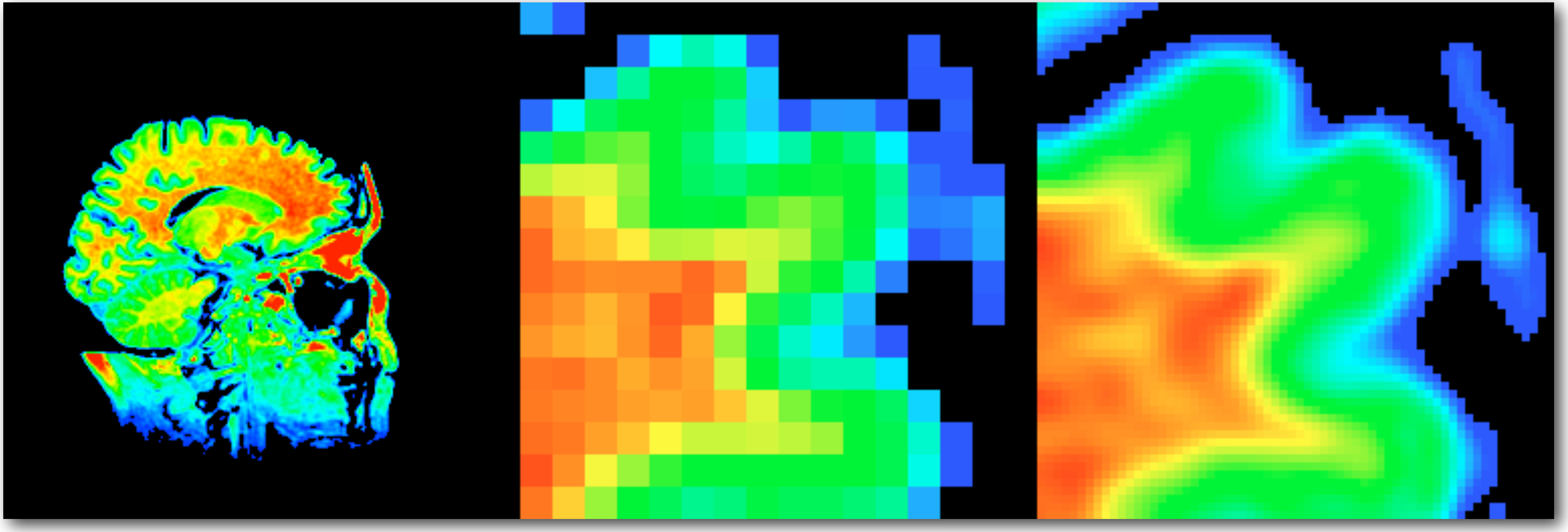



```
Array PSD4 sh e
type  PSD4 = P C (P (S D) (P (S D) (P (S D) (S D))))
```

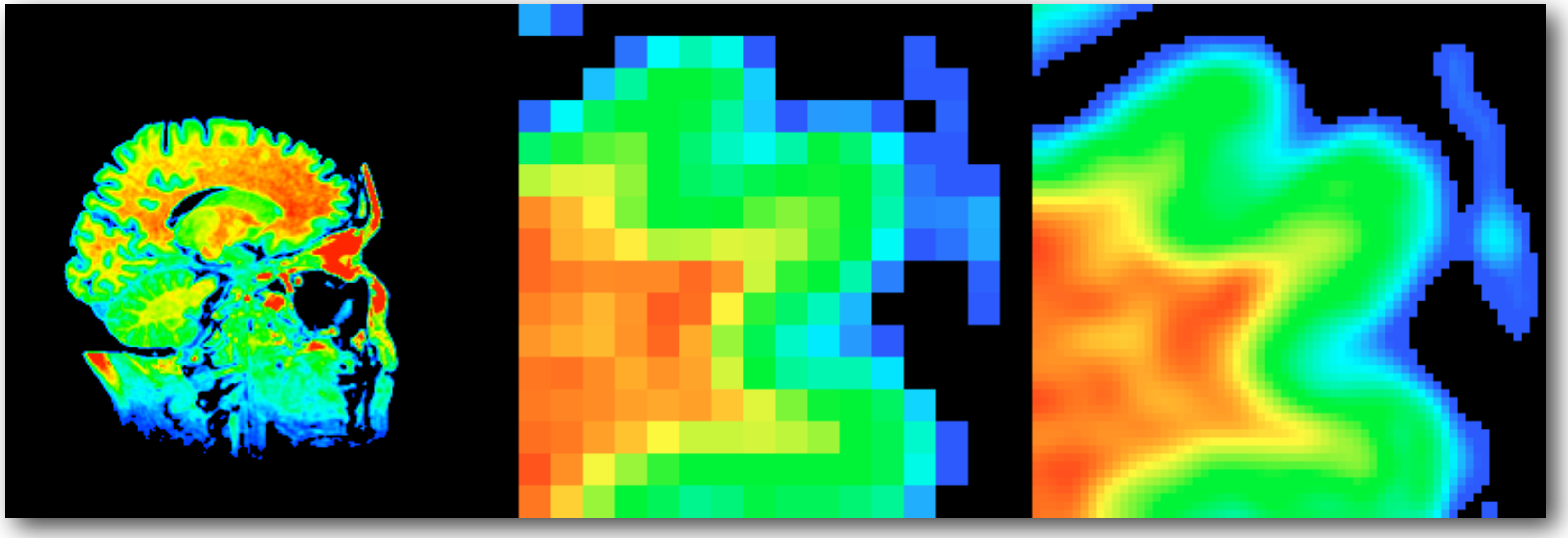
```
data S r
instance Source (S r) sh e where
 data Array (S r) sh e = HintSmall (Array r sh e)

instance (Shape sh, Load r sh e)
  => Load (S r) sh e where
 loadP (HintSmall arr) marr = loadS arr marr
 loadS (HintSmall arr) marr = loadS arr marr
```


4
4
4
4
5
5
5
5
6
6
6
6
7
7
7
7
8
8
8
8
9
9
9
9
10
10
10
10

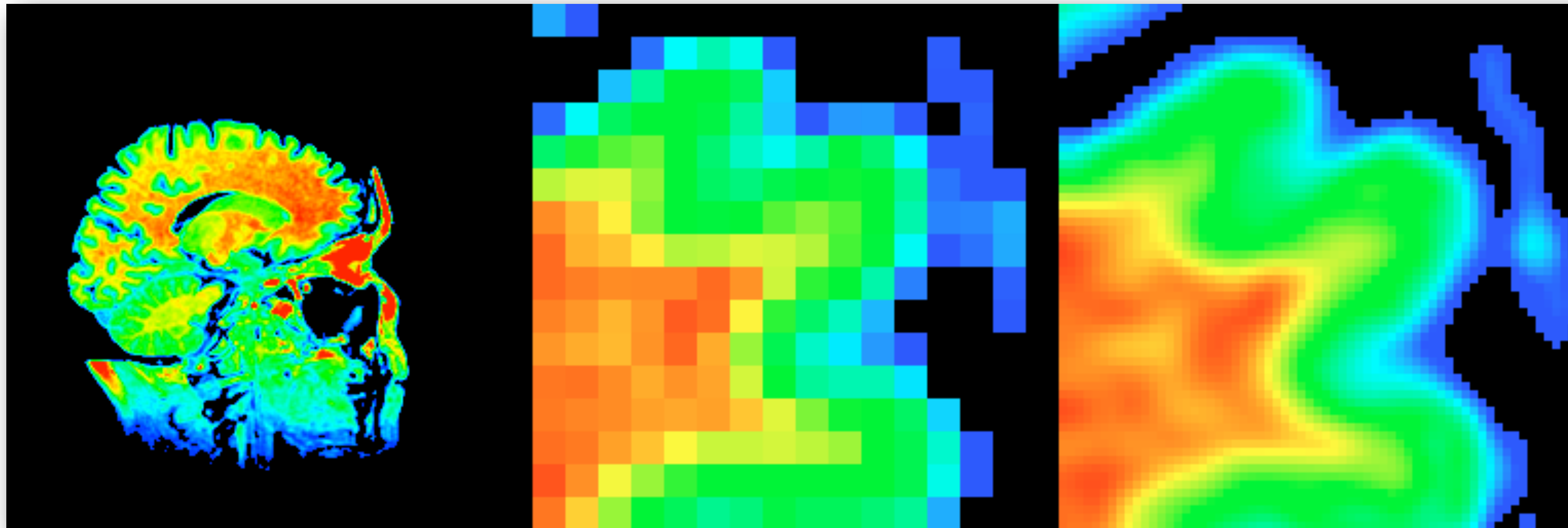
Interleave Hints



volumetric interpolation by Michael Orlitzky



volumetric interpolation by Michael Orlitzky



volumetric interpolation by Michael Orlitzky

evaluation orders

1	1	1	1	1
1	1	2	2	2
2	2	2	2	3
3	3	3	3	3

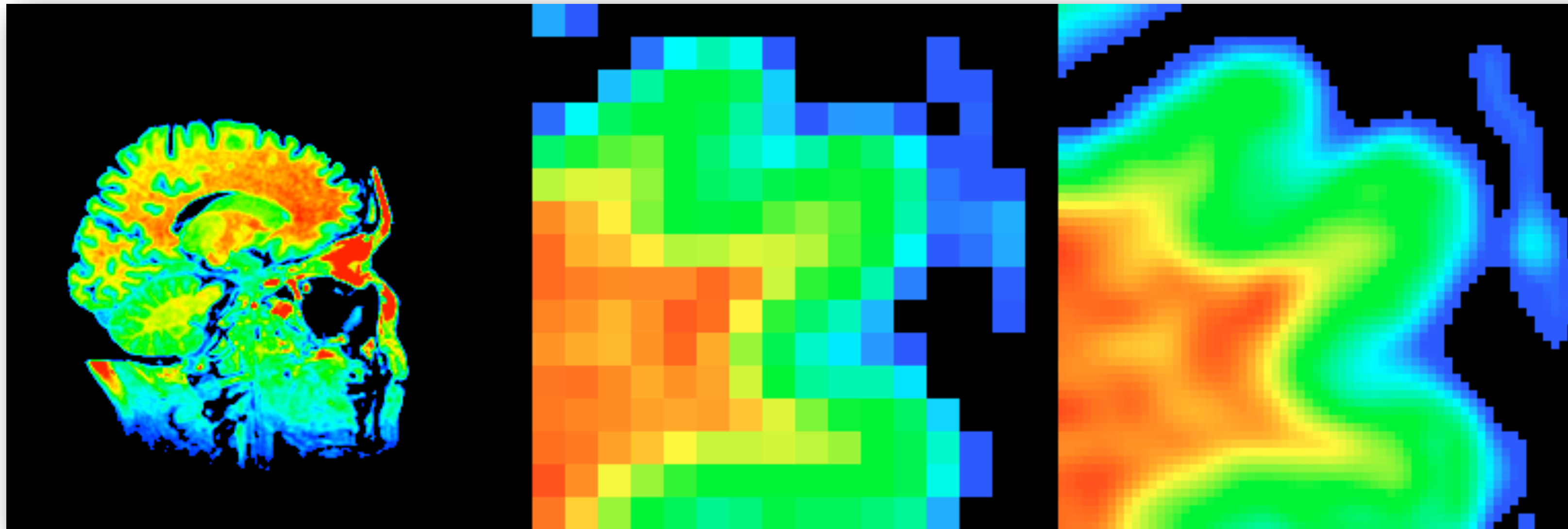
chunked

1	1	2	2	3
1	1	2	2	3
1	1	2	2	3
1	1	2	2	3

column-wise

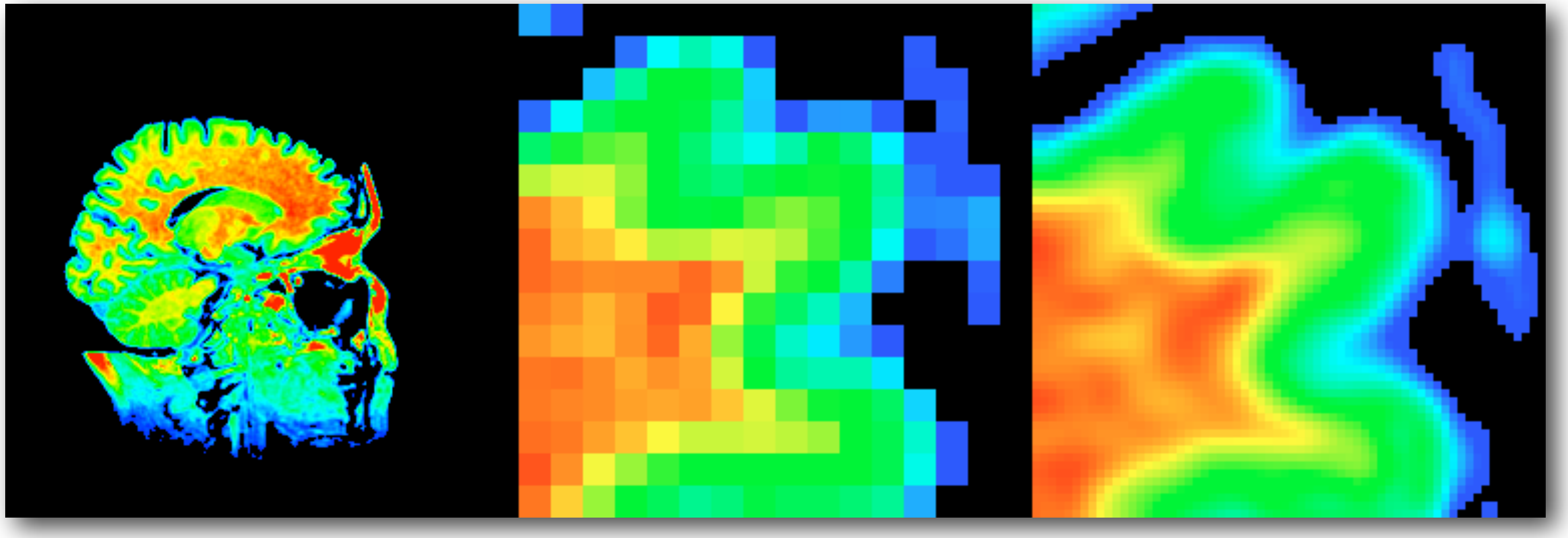
1	2	3	1	2
3	1	2	3	1
2	3	1	2	3
1	2	3	1	2

interleaved



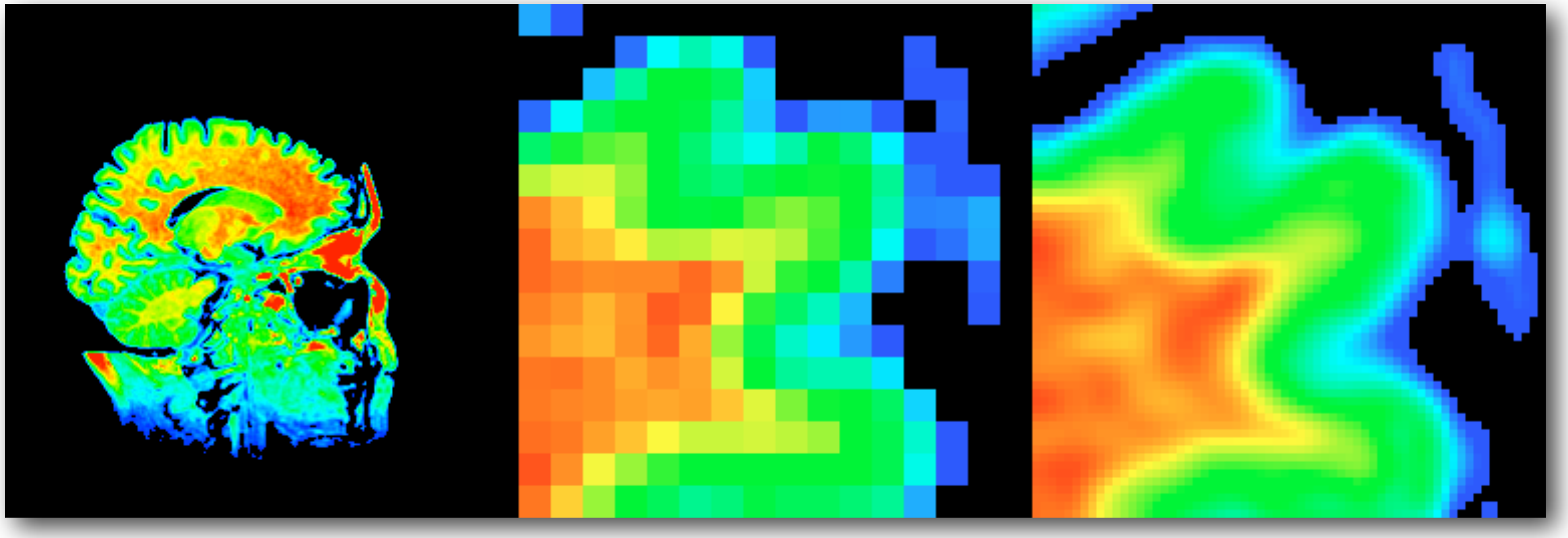
volumetric interpolation by Michael Orlitzky

```
data I r1
instance Source (I r1) sh e where
 data Array (I r1) sh e
        = HintInterleave (Array r1 sh e)

instance ( Shape sh, Load D sh e)
        => Load (I D) sh e where
 loadP (HintInterleave (ADelayed sh getElem)) marr
   = fillInterleavedP ...
```

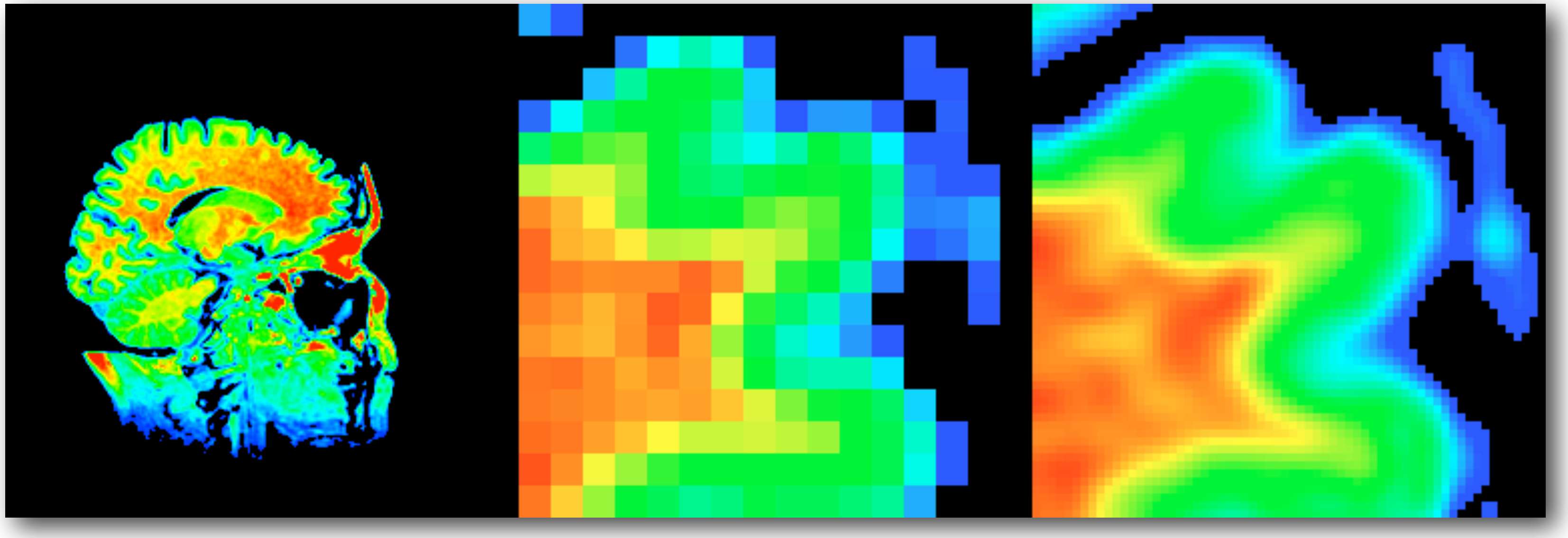


volumetric interpolation by Michael Orlitzky

```
interpolate :: Array U     DIM3 Double
            -> Array (I D) DIM3 Double
```



volumetric interpolation by Michael Orlitzky



volumetric interpolation by Michael Orlitzky