Guiding Parallel Array Fusion with Indexed Types

Ben Lippmeier✿ Manuel Chakravarty✿ Gabriele Keller✿
Simon Peyton Jones★

✿University of New South Wales
★Microsoft Research Ltd

Haskell Symposium 2012

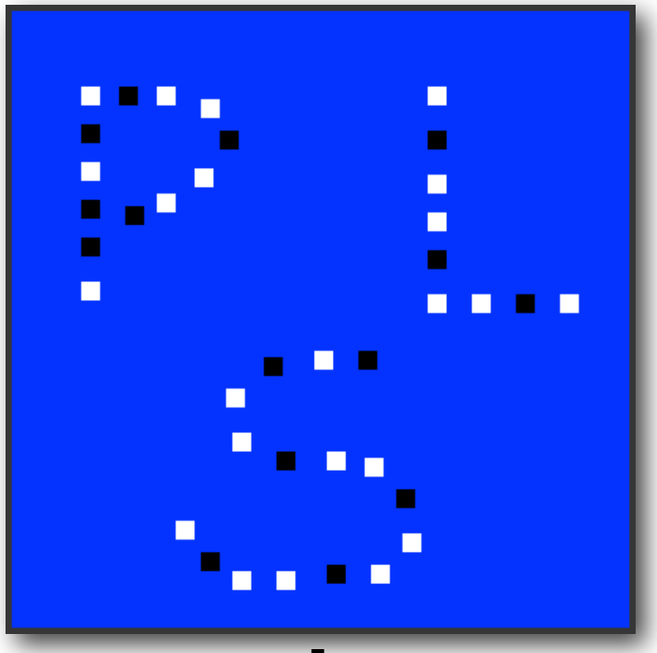
aka Repa 3

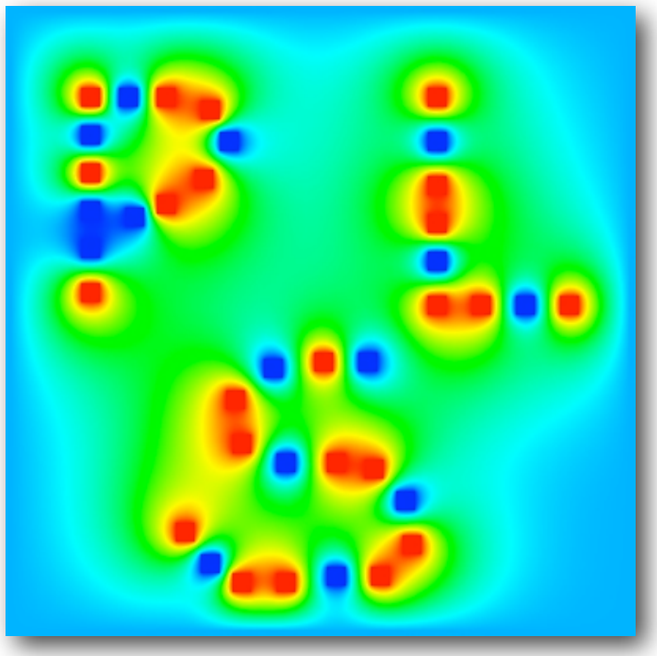
Ben Lippmeier✿ Manuel Chakravarty✿ Gabriele Keller✿
Simon Peyton Jones★

✿University of New South Wales
★Microsoft Research Ltd

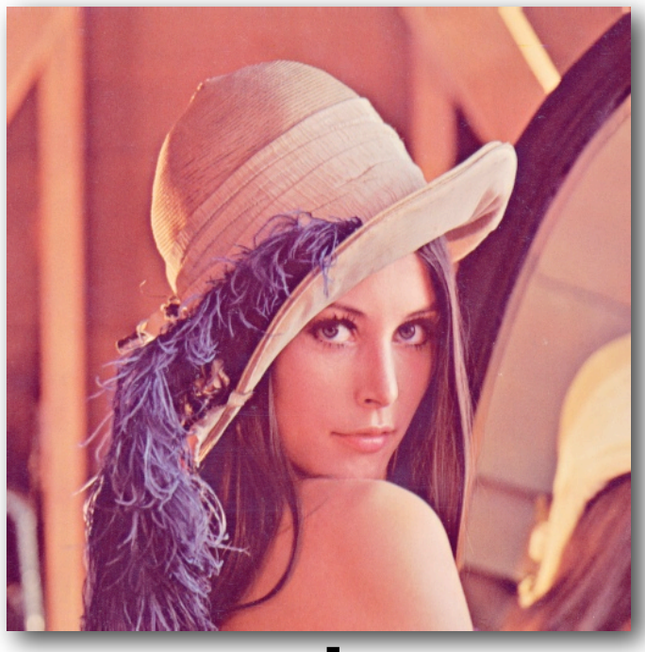
Haskell Symposium 2012

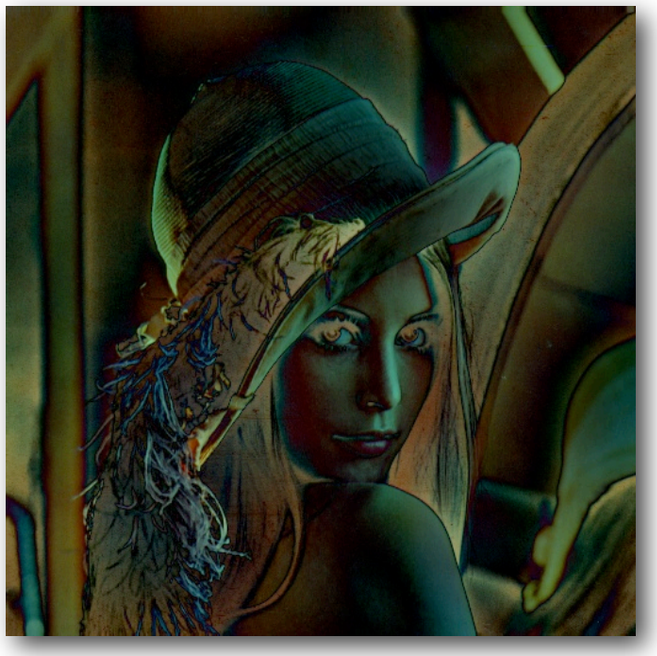
Flat Regular Data Parallelism





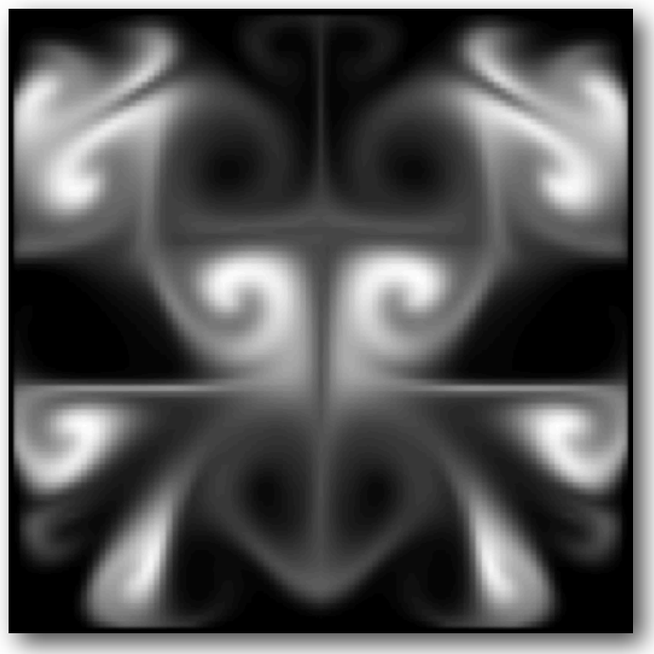
Matrix Relaxation





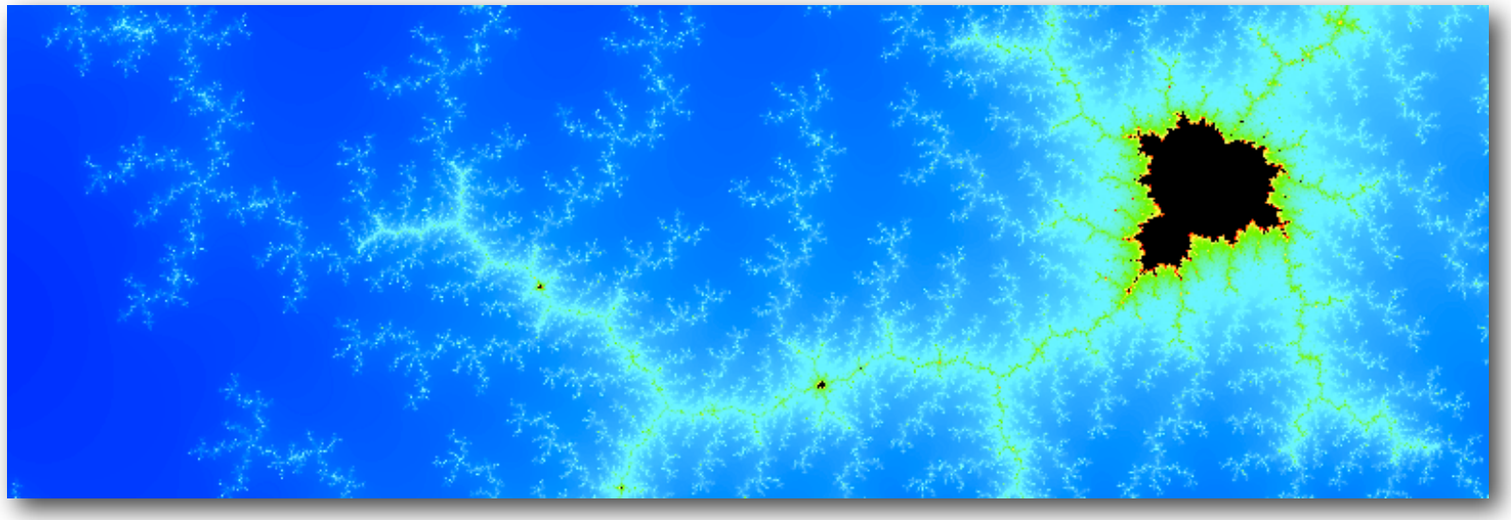
High Pass /w FFT

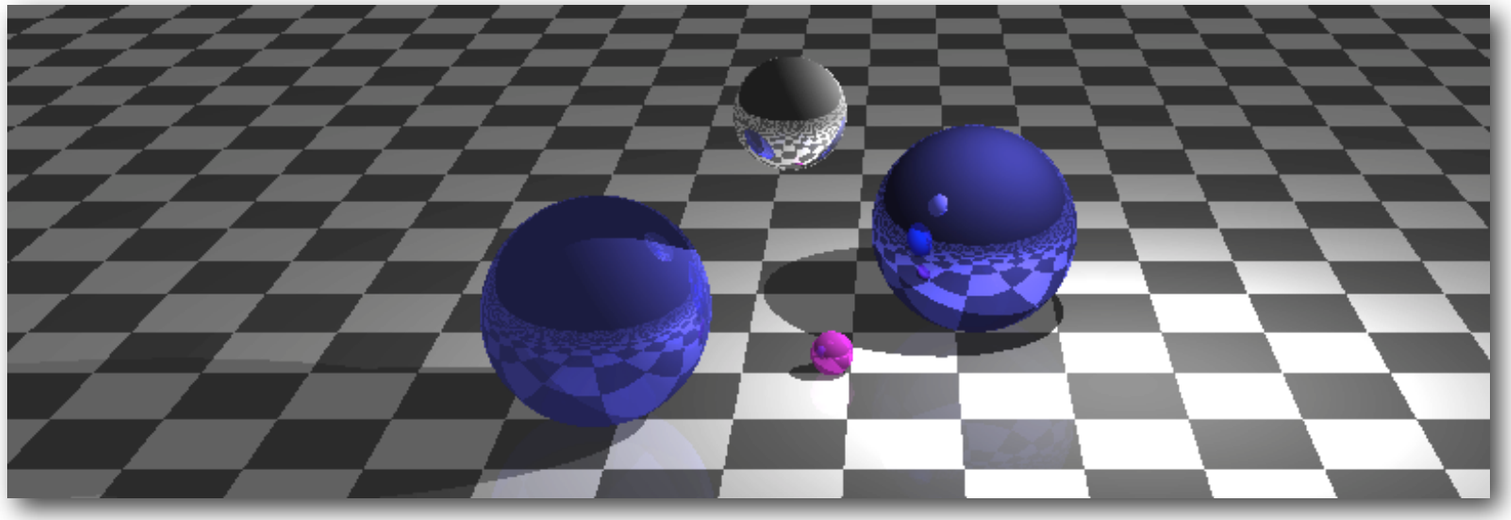
Edge Detection

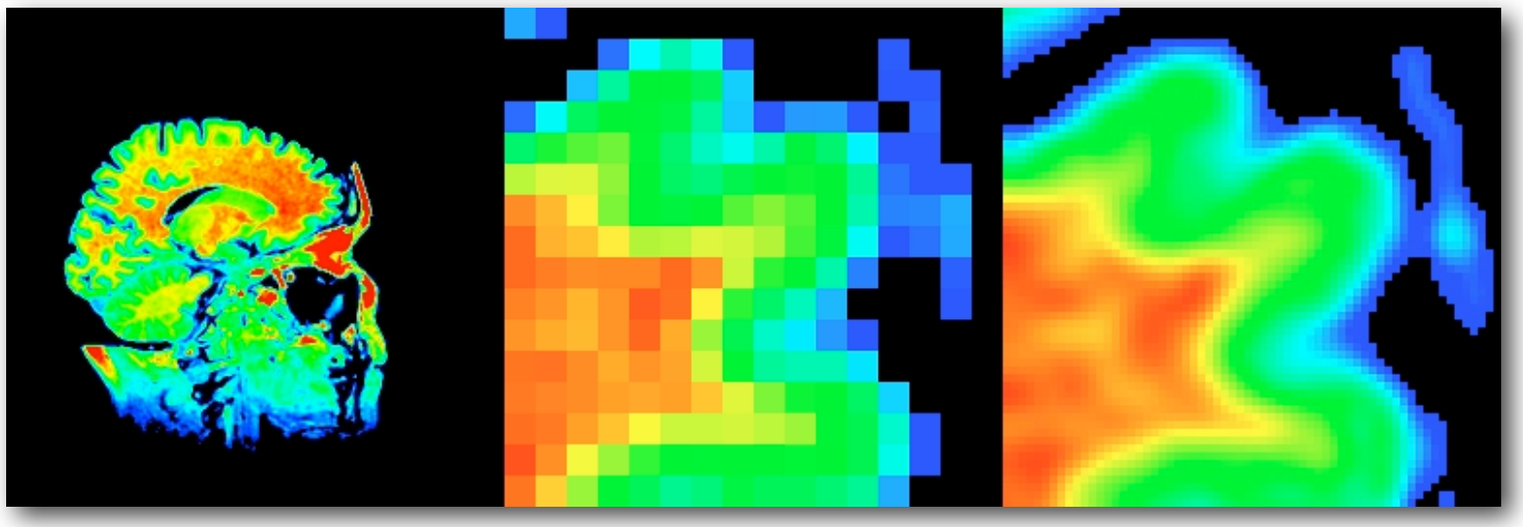


Fluid Simulation

Unbalanced Flat Data Parallelism







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

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

append    :: (Shape sh, Elt a)
          => Array (sh :. Int) a -> Array (sh :. Int) a
          -> Array (sh :. Int) a

fold      :: (Shape sh, Elt a)
          => (a -> a -> a) -> a
          -> Array (sh :. Int) a -> Array sh a

traverse :: (Shape sh1, Shape sh2, Elt a)
          => Array sh1 a
          -> (sh1 -> sh2)
          -> ((sh1 -> a) -> sh2 -> b)
          -> Array sh2 b
```

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = map (* 2) $ zipWith (+) arr1 arr2
```

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = map (* 2) $ zipWith (+) arr1 arr2
```

About 100x slower
than with `Data.Vector`

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = map (* 2) $ zipWith (+) arr1 arr2
```

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

- A `Manifest` array is a flat unboxed array in row-major order

- A `Delayed` array is a function that computes elements on the fly.

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = map (* 2) $ zipWith (+) arr1 arr2
```

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

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

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = map (* 2) $ zipWith (+) arr1 arr2
```

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

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

- **`force`** turns a delayed array into a Manifest one,
  using fast, parallel, tail-recursive loops.

- If **`force`** is not used the program gives the correct answer,
  but is slow and sequential.

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = force $ map (* 2) $ zipWith (+) arr1 arr2
```

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

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

- **`force`** turns a delayed array into a Manifest one,
  using fast, parallel, tail-recursive loops.

- If **`force`** is not used the program gives the correct answer,
  but is slow and sequential.

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = force $ map (* 2) $ zipWith (+) arr1 arr2
```

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

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

- The type of **`force`** is an instance of the type of **`id`**, which does not reveal how critical it is to performance.

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1 arr2
 = force $ map (* 2) $ zipWith (+) arr1 arr2
```

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1@Manifest{} arr2@Manifest{}
 = force $ map (* 2) $ zipWith (+) arr1 arr2
```

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1@(Manifest !_ !_)
          arr2@(Manifest !_ !_)
 = force $ map (* 2) $ zipWith (+) arr1 arr2
```

```
doubleZip :: Array DIM2 Int -> Array DIM2 Int
          -> Array DIM2 Int
doubleZip arr1@(Manifest !_ !_)
          arr2@(Manifest !_ !_)
 = force $ map (* 2) $ zipWith (+) arr1 arr2
```

Runs fine....

...but is ugly, and non-obvious.

data family Array rep sh e

```
data family Array rep sh e
```

Representation tag indexes the possible array representations.

```
data family Array rep sh e
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

```
data family Array rep sh e
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

manifest unboxed arrays

```
data U
data instance Array U sh e
      = AUnboxed sh (Vector e)
```

```
data family Array rep sh e
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

manifest unboxed arrays

```
data U
data instance Array U sh e
      = AUnboxed sh (Vector e)
```

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

```
data family Array rep sh e
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

manifest unboxed arrays

```
data U
data instance Array U sh e
      = AUnboxed sh (Vector e)
```

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

```
data family Array rep sh e
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

manifest unboxed arrays

```
data U
data instance Array U sh e
      = AUnboxed sh (Vector e)
```

```
computeS :: Array D sh e -> Array U sh e
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

manifest unboxed arrays

```
data U
data instance Array U sh e
      = AUnboxed sh (Vector e)
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

manifest unboxed arrays

```
data U
data instance Array U sh e
      = AUnboxed sh (Vector e)
```

manifest byte arrays

```
data B
data instance Array B sh Word8
      = AByteArray sh ByteArray
```

delayed arrays

```
data D
data instance Array D sh e
      = ADelayed sh (sh -> e)
```

manifest unboxed arrays

```
data U
data instance Array U sh e
      = AUnboxed sh (Vector e)
```

manifest byte arrays

```
data B
data instance Array B sh Word8
      = AByteArray sh ByteArray
```

manifest foreign memory buffers

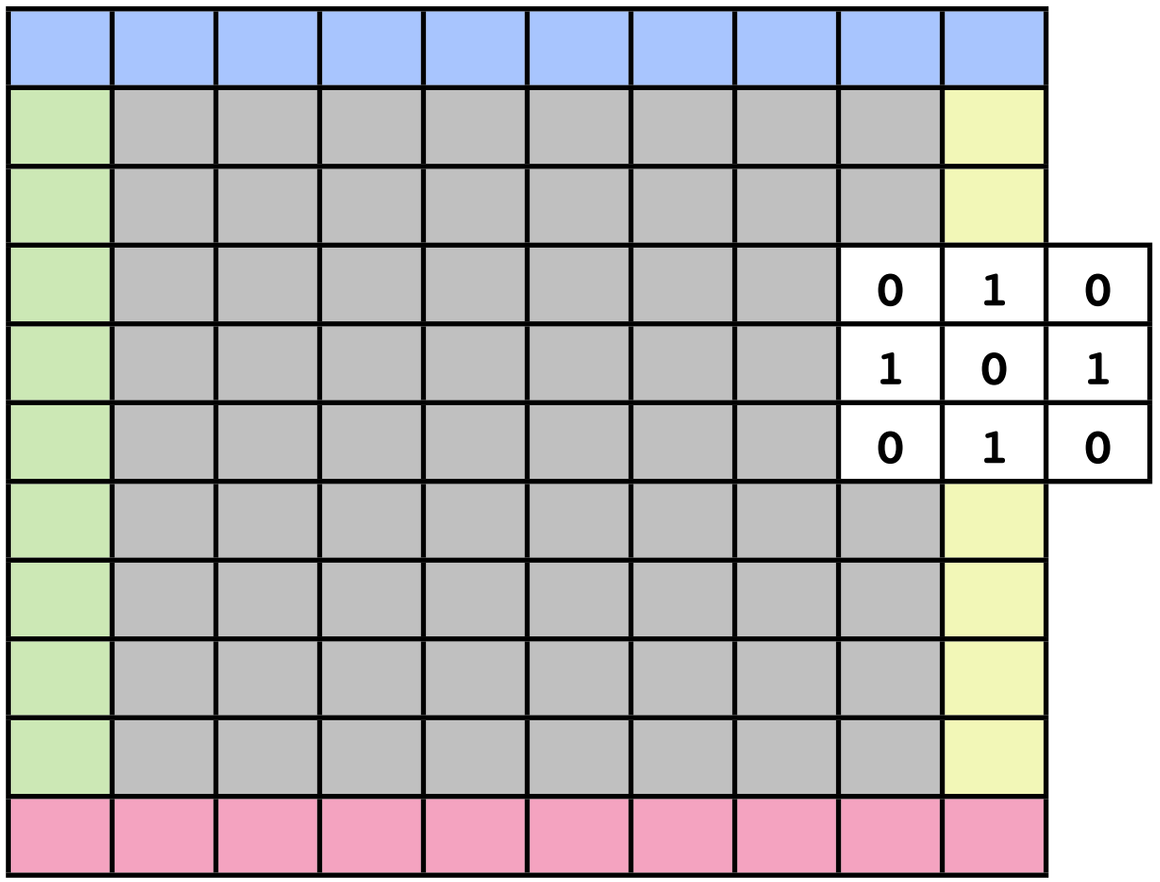
```
data F
data instance Array F sh e
      = AForeignPtr sh Int (ForeignPtr e)
```

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
```

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

```
computeP :: (Load rs sh e, Target rt e)
         => Array rs sh e -> Array rt sh e
```

```
computeP :: (Load rs sh e, Target rt e)
         => Array rs sh e -> Array rt sh e

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

```
computeP :: (Load rs sh e, Target rt e)
         => Array rs sh e -> Array rt sh e

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

manifest foreign memory buffers

```
data F
data instance Array F sh e
      = AForeignPtr sh Int (ForeignPtr e)
```

```
computeP :: (Load rs sh e, Target rt e)
         => Array rs sh e -> Array rt sh e

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

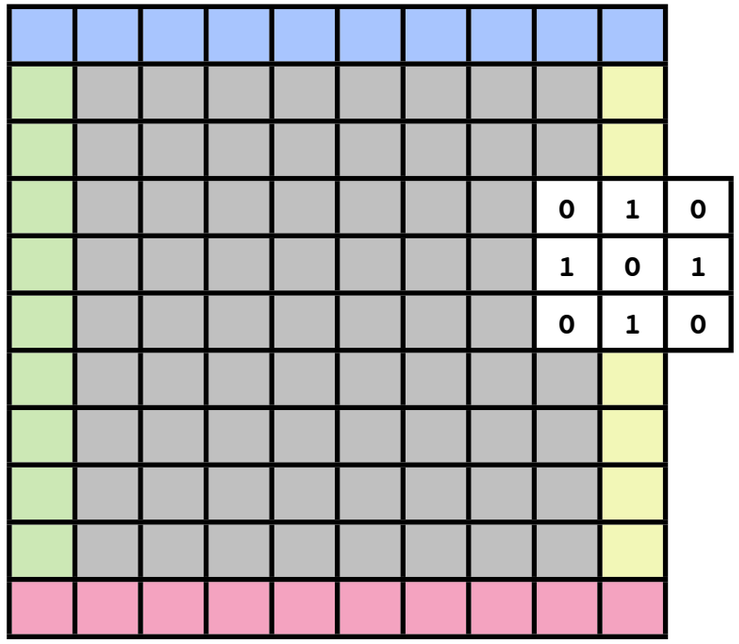
manifest foreign memory buffers

```
data F
instance Storable a => Source F a where
 data Array F sh a
        = AForeignPtr sh Int (ForeignPtr a)
 extent = ...
 index  = ...
```

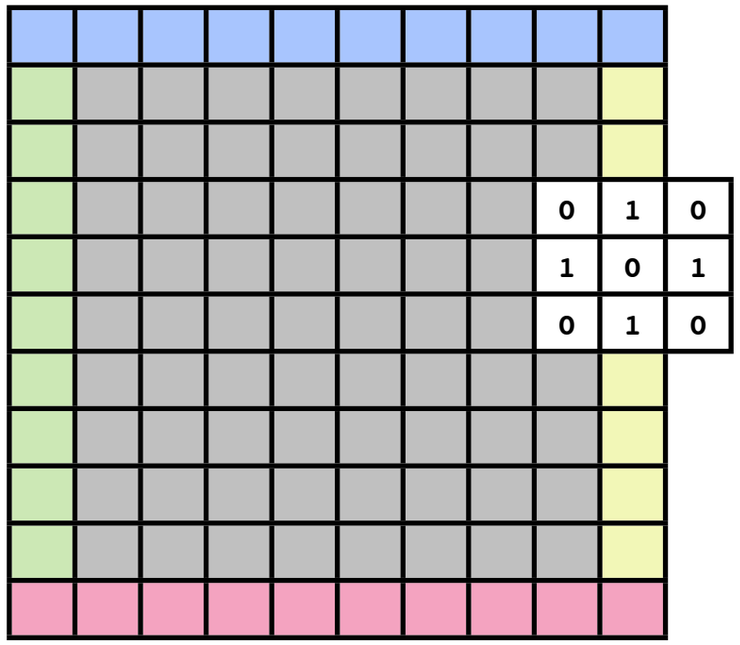
```
computeP :: (Load rs sh e, Target rt e)
         => Array rs sh e -> Array rt sh e

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

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)
```

```
    computeP :: (Load rs sh e, Target rt e)
             => Array rs sh e -> Array rt sh e

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

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)

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

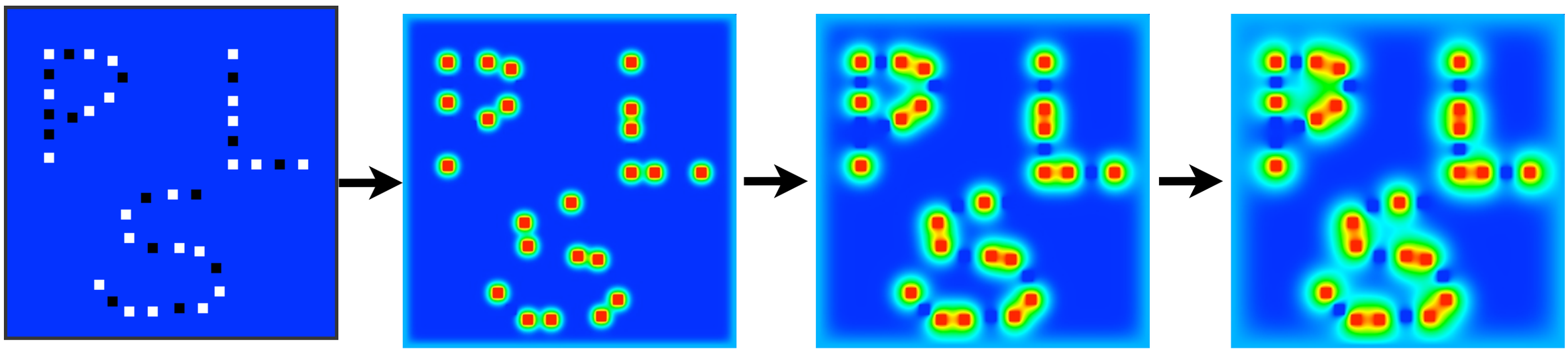
```
computeP :: (Load rs sh e, Target rt e)
         => Array rs sh e -> Array rt sh e
computeP arr1
 = unsafePerformIO
 $ do mvec2 <- newMVec (size $ extent arr1)
      loadP arr1 mvec2
      freezeMVec (extent arr1) mvec2
```

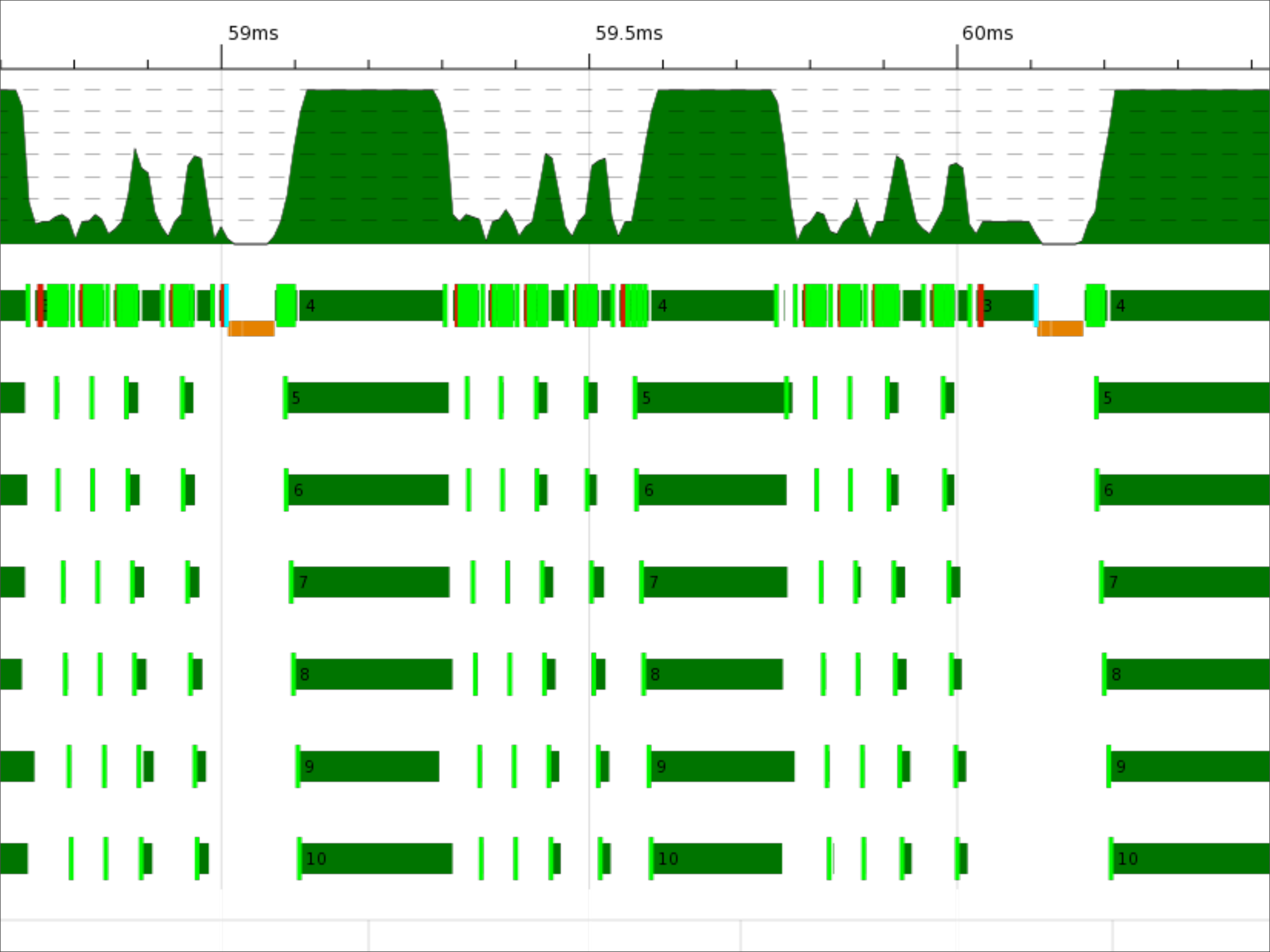


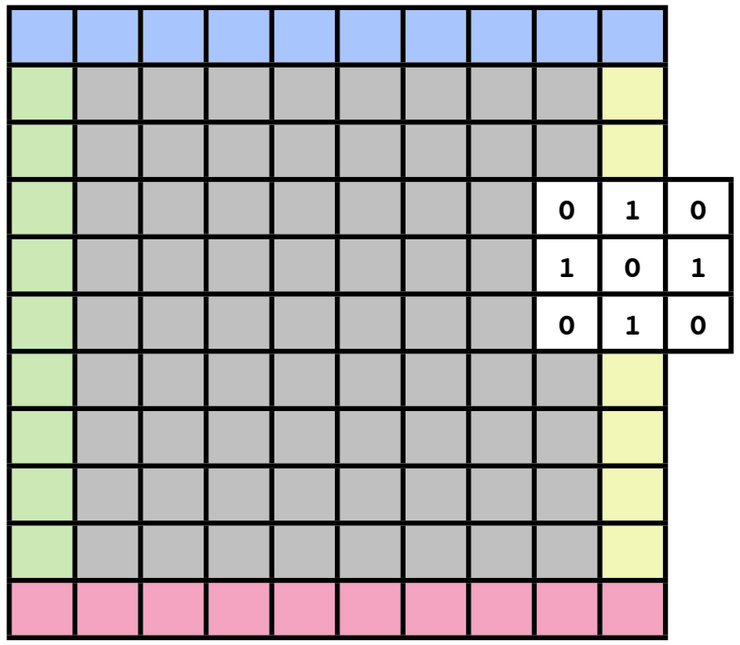

```
Array (P C (P D (P D (P D D)))) sh e
```




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

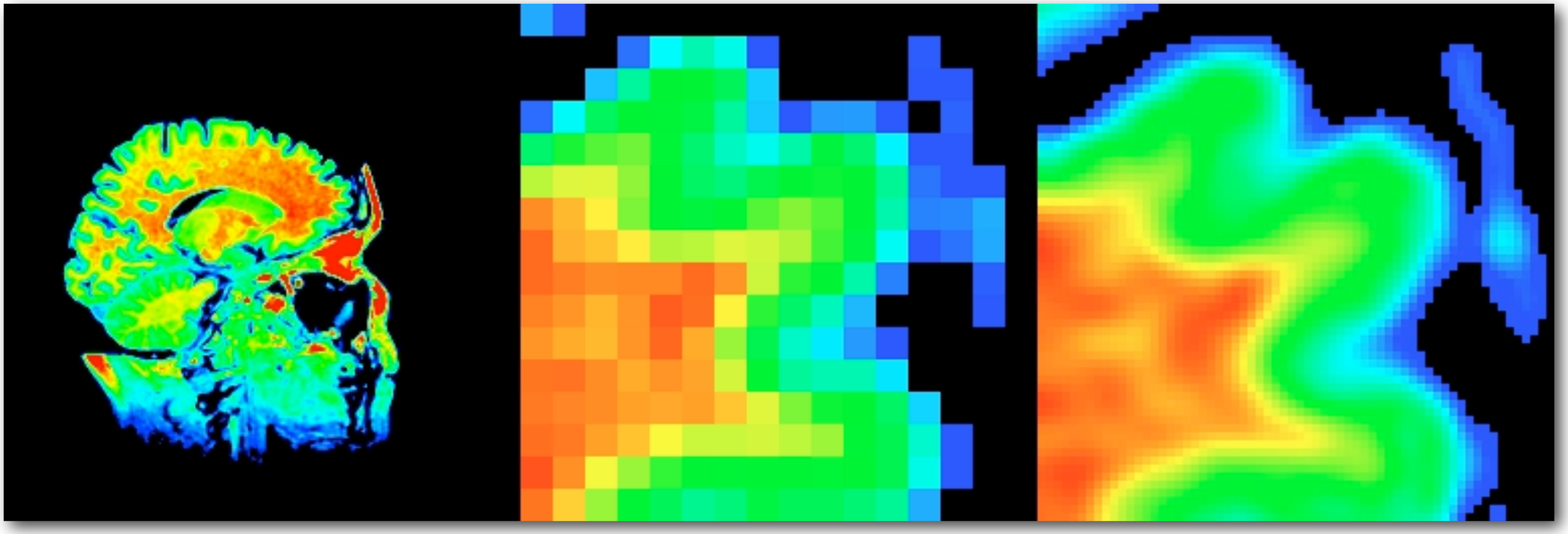



59ms
59.5ms
60ms
4
4
3
4
5
5
5
6
6
6
7
7
7
8
8
8
9
9
9
10
10
10

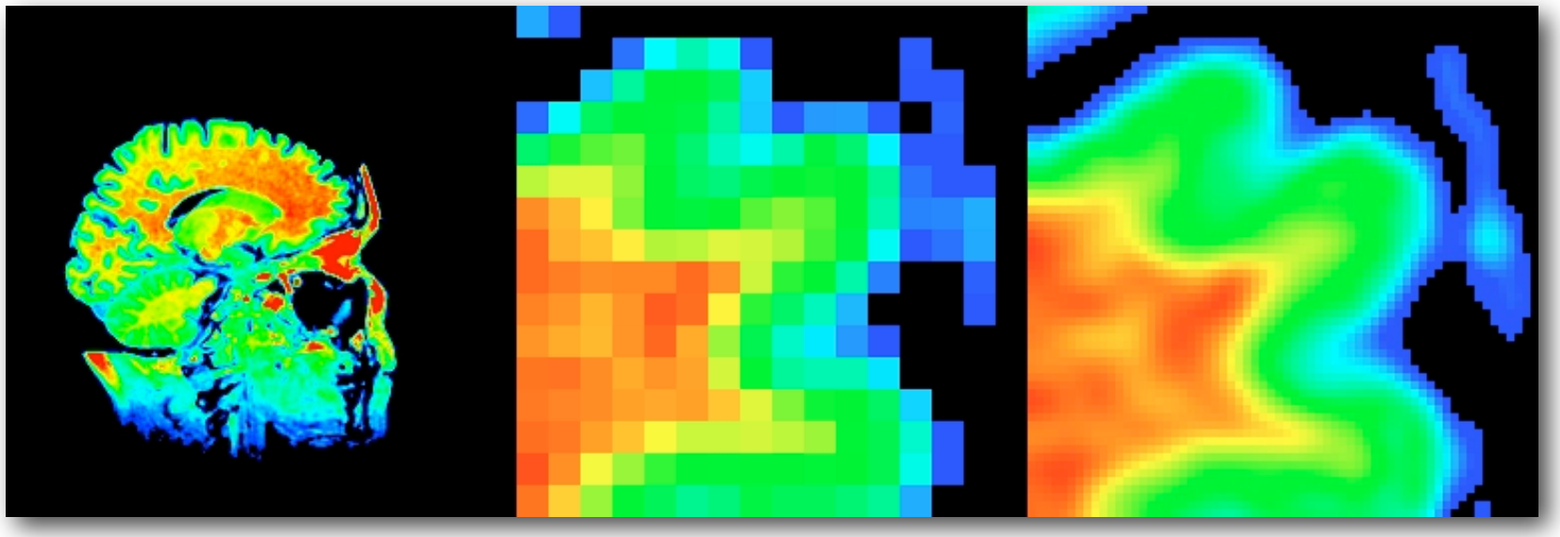



```
Array PSD4 sh e
type  PSD4 = P C (P (S D) (P (S D) (P (S D) (S D))))


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
```

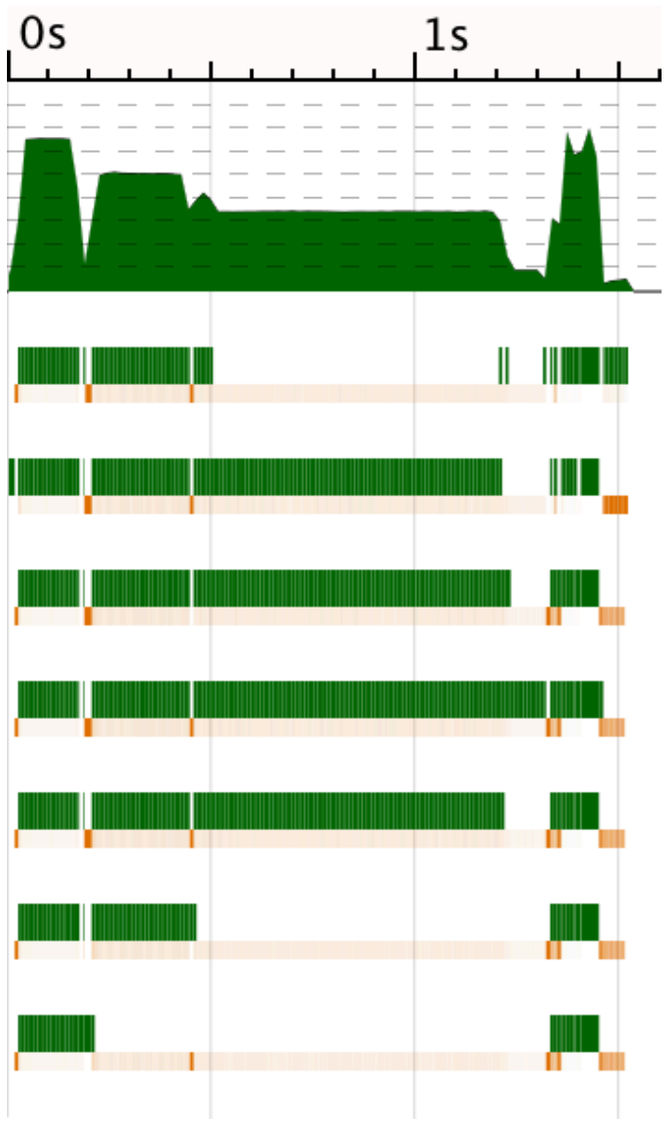

4
4
4
4
5
5
5
5
6
6
6
6
7
7
7
7
8
8
8
8
9
9
9
9
10
10
10
10

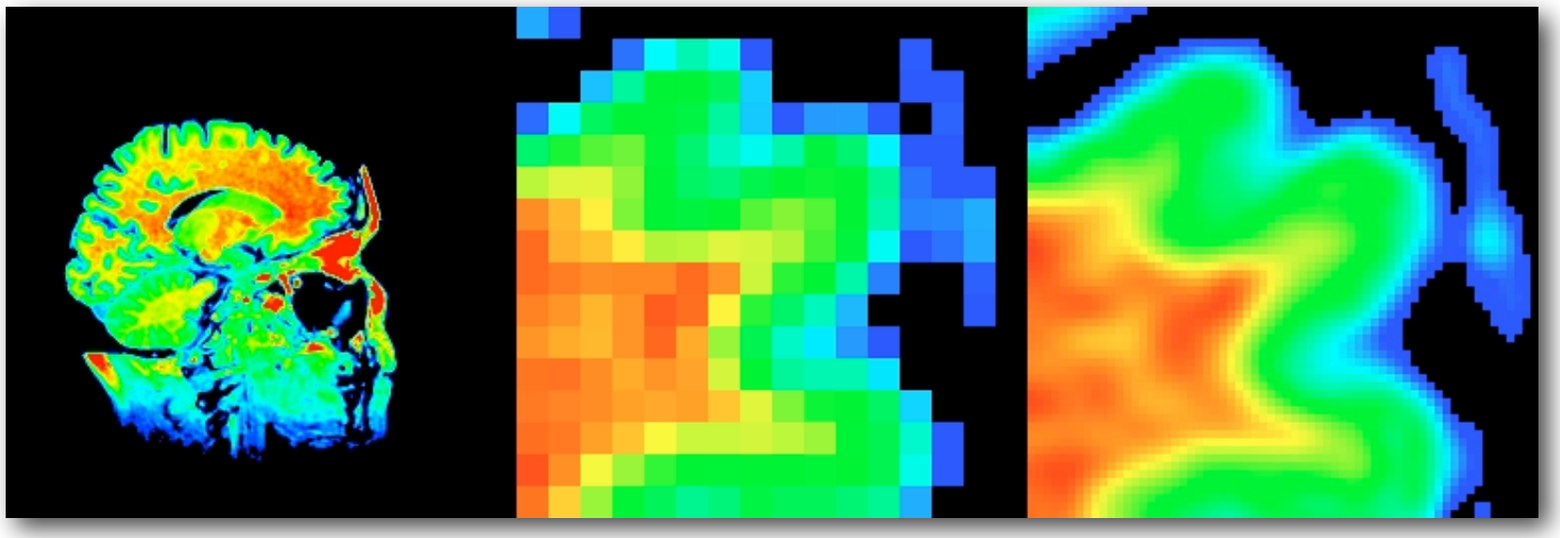


volumetric interpolation by Michael Orlitzky



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

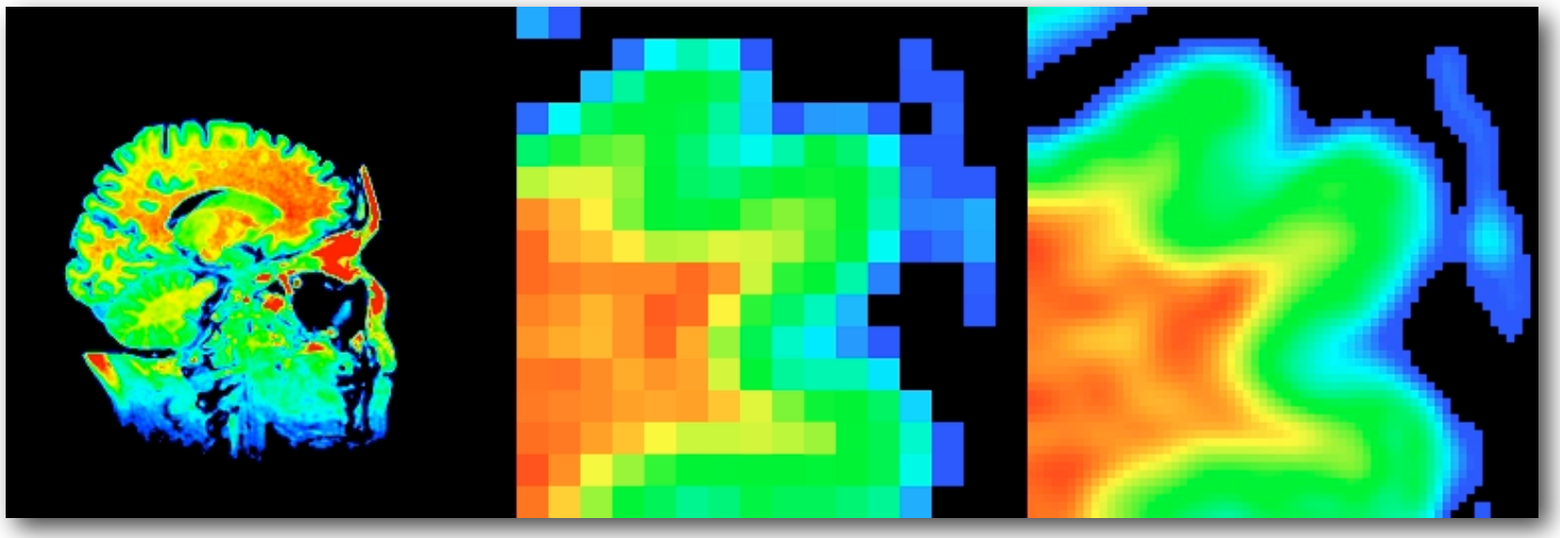
chunked

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

column-wise

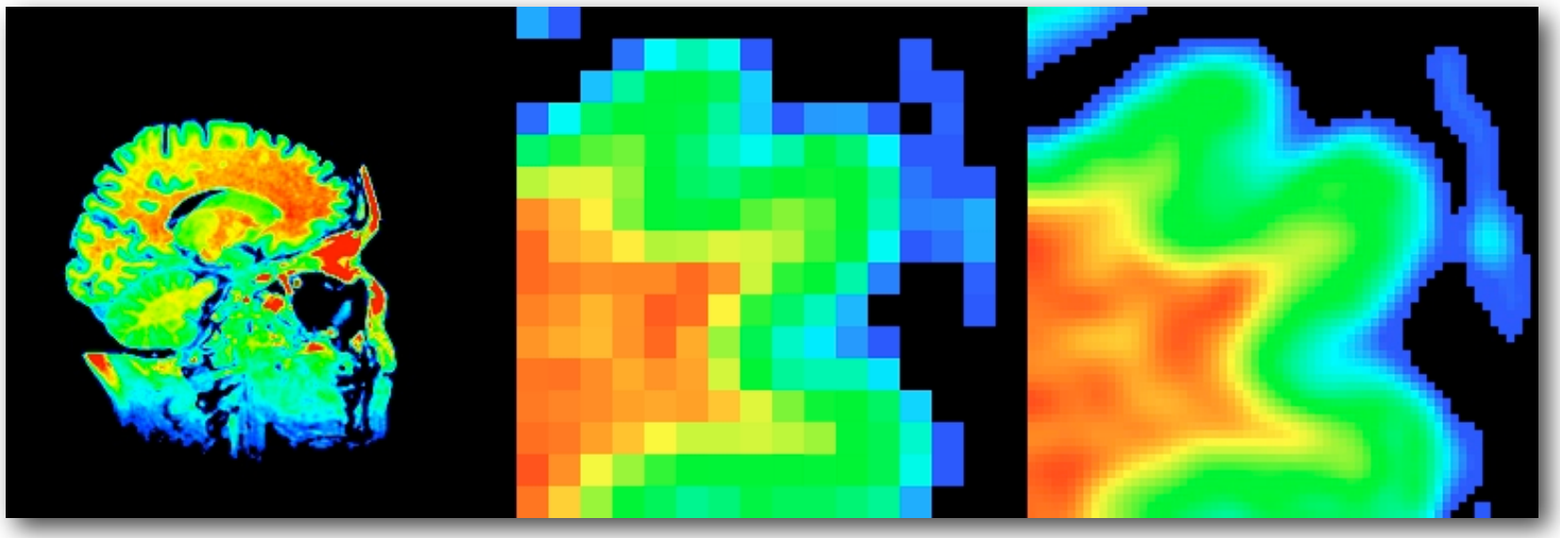
1	2	3	1	2
3	1	2	3	1
2	3	1	2	3
1	2	3	1	2

interleaved



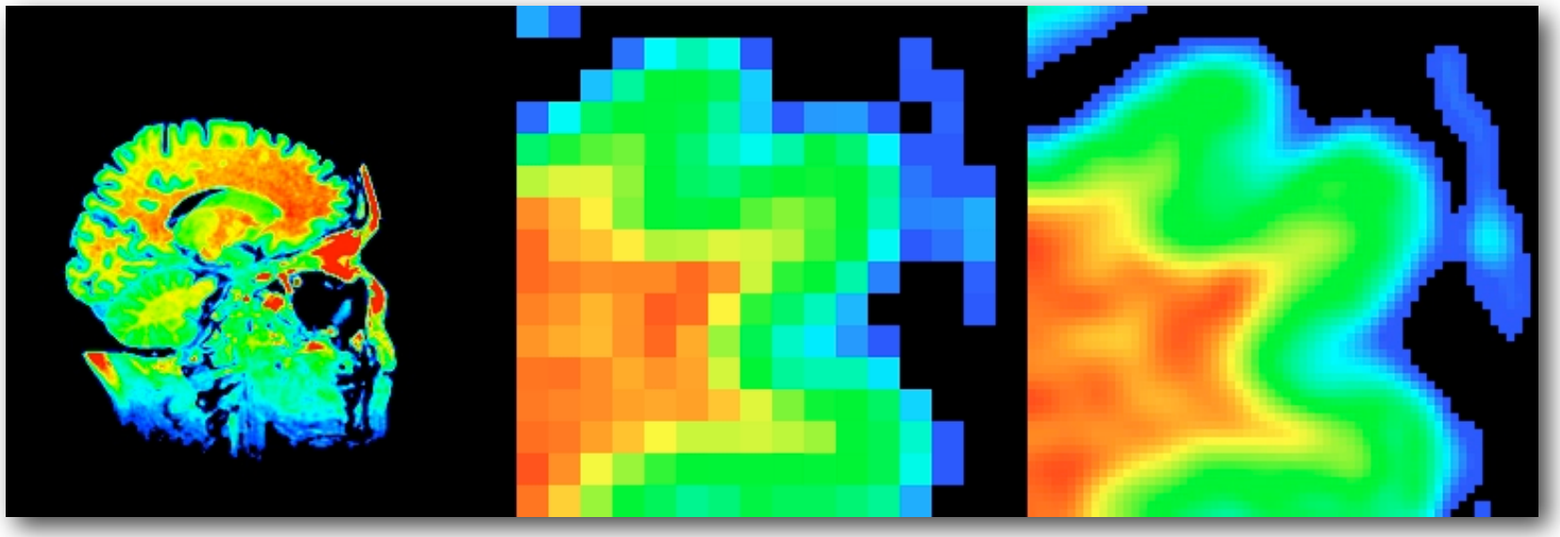
volumetric interpolation by Michael Orlitzky

```
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 ...
```

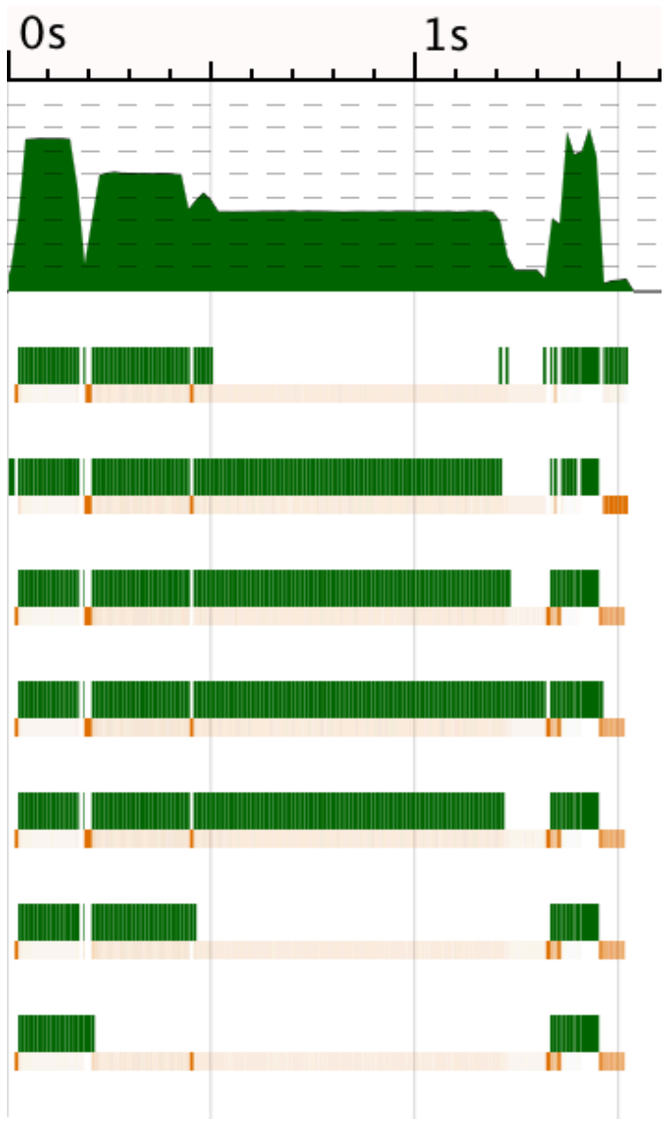


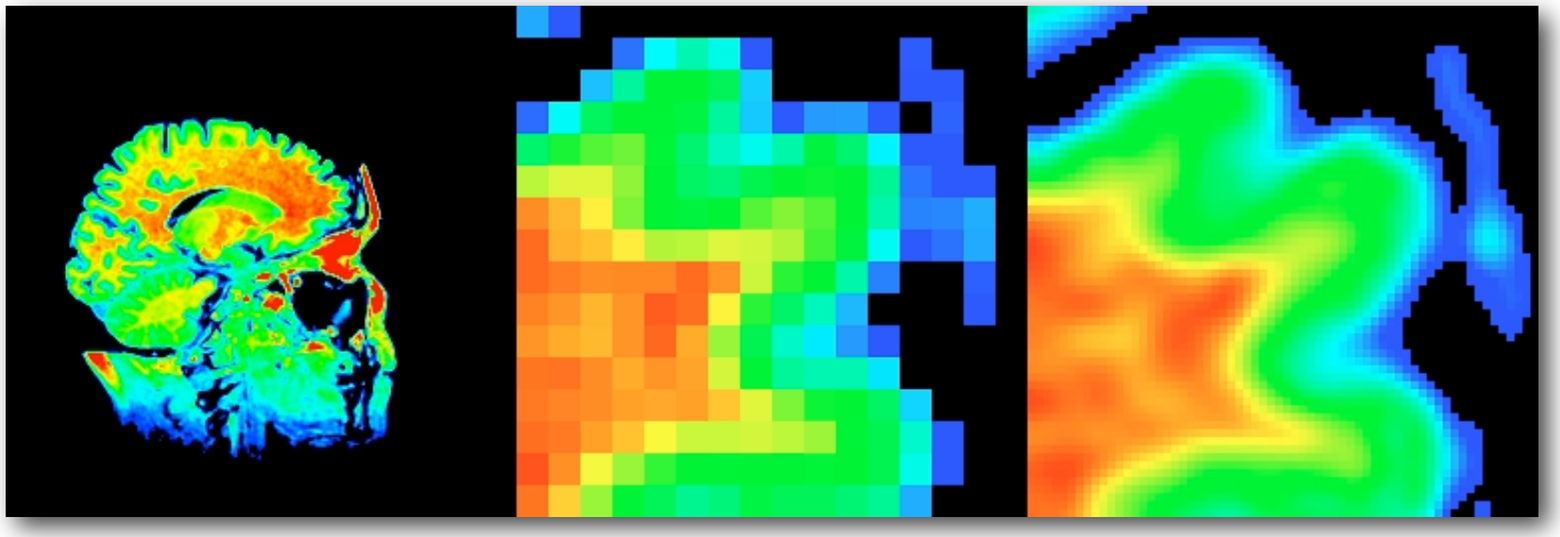
volumetric interpolation by Michael Orlitzky

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

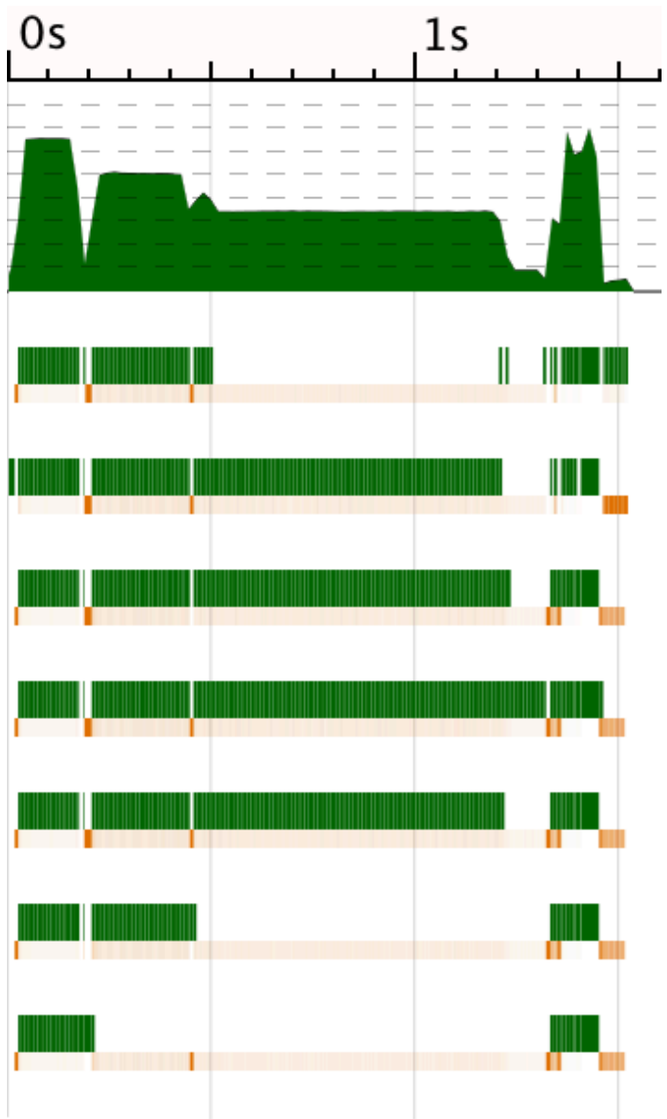


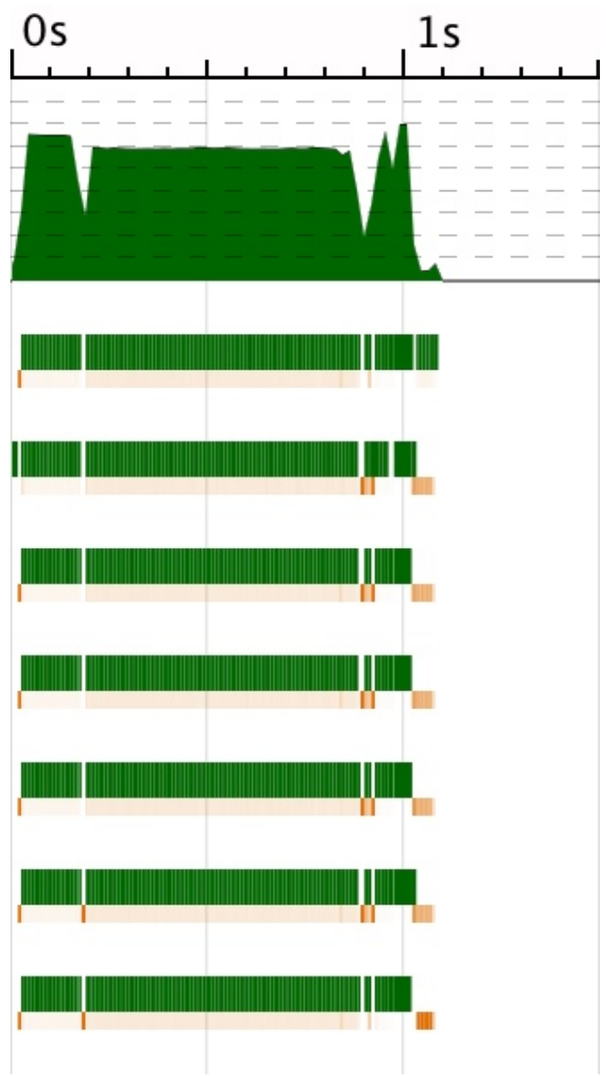
volumetric interpolation by Michael Orlitzky



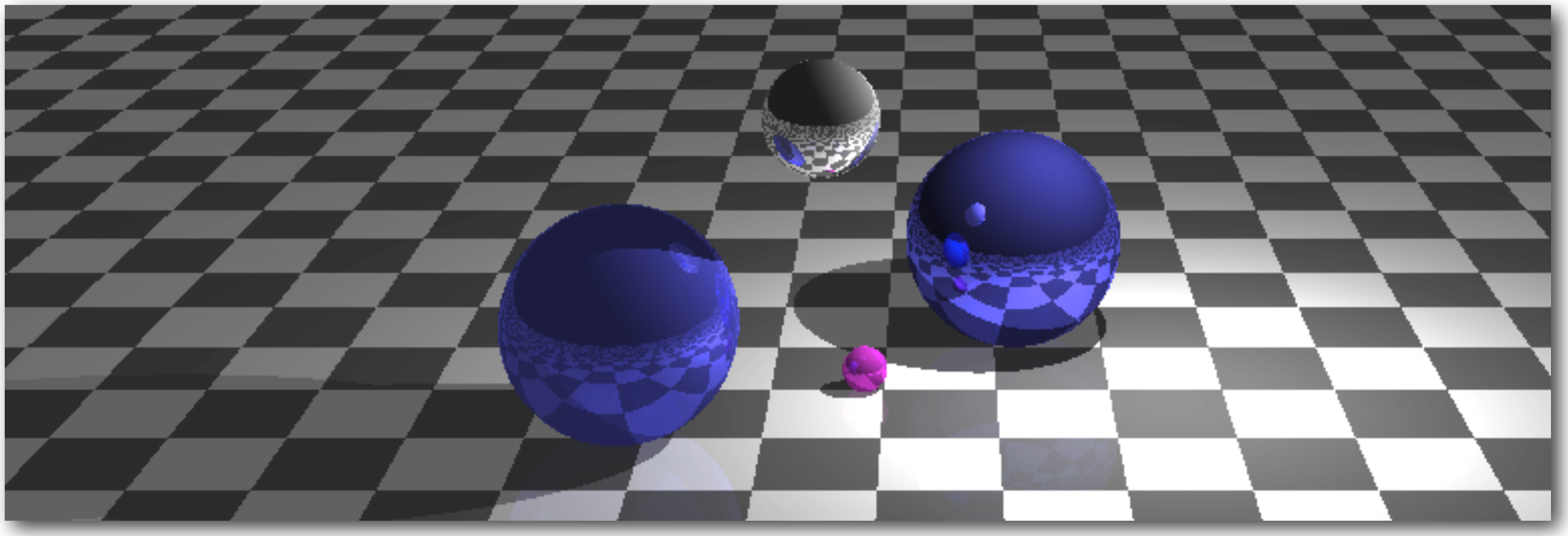


volumetric interpolation by Michael Orlitzky





real-time ray tracer demo



Spare Slides

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
  = APartitioned
      sh
      (Range sh) (Array r1 sh e)
      (Array r2 sh e)


data Range sh
   = Range
   { rangeLow  :: sh
   , rangeHigh :: sh
   , inRange   :: sh -> Bool }
```