Monadic Warsaw #14

June 5th 2018

Alexey Kuleshevich

massiv - Haskell arrays that are easy and fast

About me:

- Live in **Minsk, Belarus**.
- Work at **FP Complete**
- BS and MS from **University of New Mexico**
- Zacząłem studia na **Uniwersytecie Warszawskim**

20 czerwca 2004 r.

Lotnisko Chopina - Warszawa	

Why do we need another array library?

array, vector, repa, accelerate, yarr, carr ...

Can we do better?

What's an Array?

- It is a data structure.

```
data family Array r ix e :: *
```

- A mapping from an index to a value:

```
index' :: Manifest r ix e => Array r ix e -> ix -> e
```

- Array has a size (aka. extent):

```
size :: Size r ix e => Array r ix e -> ix
```

Types of arrays:

- Vector (1-Dim), Matrix (2-Dim), Rectangular (N-Dim)
- Ragged or Jagged
- Sparse
- Dope, Judy, compact, etc.

```
Array r ix e
      \  \  \
       \  \  \ - Type for an element of the Array
        \  \
         \  \ - Index type of this Array
          \
           \ - Array representation type variable
```

What's Index?

```haskell
class Index ix where
  type Rank ix :: Nat
  rank :: ix -> Dim
  toLinearIndex :: ix -- ^ Size
                -> ix -- ^ Index
                -> Int -- ^ Linear index
  fromLinearIndex :: ix -> Int -> ix
  consDim :: Int -> Lower ix -> ix
  ...

newtype Dim = Dim Int
type family Lower ix :: *
```

- `Int` - for flat vectors.
- Tuples:
  - `(Int, Int) :: Ix2T`,
  - `(Int, Int, Int) :: Ix3T`, ... `Ix5T`

Proper way to index: `Ix`

```
data Ix0 = Ix0

type Ix1 = Int

data Ix2 = (:.) !Int !Int

data IxN (n :: Nat) = (:>) !Int !(Ix (n - 1))

type family Ix (n :: Nat) = r | r -> n where
  Ix 0 = Ix0
  Ix 1 = Ix1
  Ix 2 = Ix2
  Ix n = IxN n

type instance Lower Ix1 = Ix0
type instance Lower Ix2 = Ix1
type instance Lower (IxN n) = Ix (n - 1)
```

What's the point of `Ix n`?

- Infinite dimensionality with unambiguous types:

```
λ> :t (3 :. 2)
(3 :. 2) :: Ix2
λ> :t (4 :> 3 :. 2)
(4 :> 3 :. 2) :: IxN 3
λ> :t (9 :> 8 :> 7 :> 6 :> 5 :> 4 :> 3 :. 2)
(9 :> 8 :> 7 :> 6 :> 5 :> 4 :> 3 :. 2) :: IxN 8
```

- `Ix0` is not an instance if `Index`, so no scalar arrays.
- `Int` by itself is an index, unlike in Repa `(Z :. Int)`
- All `Ix n` are instances of `Num`, `Ord`, `Unbox`, ...

```
λ> (2 :> 3 :. 4) * (7 :> 6 :. 5) < (15 :> 19 :. 21)
True
```

Convenient index constructors.

```
λ> :t 5
5 :: Num t => t
λ> :t (Ix1 5)
(Ix1 5) :: Ix1
```

```
λ> Ix2 1 2
1 :. 2
λ> Ix4 1 2 3 4
1 :> 2 :> 3 :. 4
```

```
λ> Ix3 i j k = 1 :> 2 :. 3
```

```
λ> toIx4 (1, 2, 3, 4)
1 :> 2 :> 3 :. 4
λ> (2, 3, 4, 5) == fromIx4 (2 :> 3 :> 4 :. 5)
True
```

Let's do something constructive.

```
λ> import Data.Massiv.Array as A
λ> :t makeArray
makeArray :: Construct r ix e =>
  Comp -> ix -> (ix -> e) -> Array r ix e
```

```
λ> makeArray Seq 5 (* 10) :: Array D Ix1 Int
(Array D Seq (5)
  [ 0,10,20,30,40 ])
```

```
λ> :{
λ| let a :: Array D Ix2 Int
λ|     a = makeArray Seq (3 :. 5) (\ (i :. j) -> i * j)
λ| :}
λ> a
(Array D Seq (3 :. 5)
  [ [ 0,0,0,0,0 ]
  , [ 0,1,2,3,4 ]
  , [ 0,2,4,6,8 ]
  ])
```

Delayed Representation

Such an array is described by a *size* and a *function*.

Here is a simplistic implementation of such an array:

```
data Array ix e = Array ix (ix -> e)

makeArray :: ix -> (ix -> e) -> Array ix e
makeArray sz f = Array sz f

size :: Array ix e -> ix
size (Array sz _) = sz

index :: (Num ix, Ord ix) =>
         Array ix e -> ix -> Maybe e
index (Array sz f) ix
  | isSafe sz ix = Just (f ix)
  | otherwise    = Nothing

isSafe :: (Num ix, Ord ix) => ix -> ix -> Bool
isSafe sz ix = 0 <= ix && ix < sz
```

Cool things, bad things

```
instance Functor (Array ix) where
  fmap f (Array sz g) = Array sz (f . g)

imap :: (ix -> a -> b) -> Array ix a -> Array ix b
imap f (Array sz g) = Array sz (\ix -> f ix (g ix))
```

- Fuses computation
- Duplicates evaluation

```
λ> let arr = makeArray 5 succ :: Array Int Int
λ> let arr' = imap (,) $ fmap (* 2) arr
λ> index arr' 2
Just (2,6)
λ> index arr' 2
Just (2,6)
```

`Manifest` constraint.

For that reason, `index` is restricted to `Manifest r ix e`

```
λ> let a = makeArrayR D Seq (Ix1 10) (^ (10 :: Int))
λ> let aFused = imap (,) $ fmap (* 2) a
λ> aFused ! 1
<interactive>:663:1: error:
    • No instance for (Manifest D Ix1 (Ix1, Ix1))
        arising from a use of ‘!’
    • In the expression: aFused ! 1
      In an equation for ‘it’: it = aFused ! 1
```

```
λ> evaluateAt aFused (2 :. 3)
(2 :. 3,12)
```

```
λ> let b = A.zipWith (+) a a
```

Getting to `Manifest`

1. One way is to construct it. (*side note*: usage of `Ix1`)

```
λ> makeArrayR P Seq (Ix1 5) (* 10)
(Array P Seq (5)
  [ 0,10,20,30,40 ])
```

2. Another way is to compute a delayed array:

```
λ> let aComputed = computeAs B aFused
λ> index aComputed (2 :. 3)
Just (2 :. 3,12)
```

`Mutable` and memory representation

```
class Manifest r ix e => Mutable r ix e where
  data MArray s r ix e :: *

  unsafeThaw :: PrimMonad m =>
                Array r ix e
             -> m (MArray (PrimState m) r ix e)

  unsafeFreeze :: PrimMonad m =>
                  Comp
               -> MArray (PrimState m) r ix e
               -> m (Array r ix e)
  ...
```

- `B`, `N`, `P`, `U`, `S`

`P` - Primitive arrays

Arrays that can hold primitive elements: `Char`, `Int`, `Word`, etc.

- Backed by `ByteArray#`
- Direct access, so fast read/write
- Cache friendly
- Garbage collected (GC)
- Not safe to pass with Foreign Function Interface (FFI)
- Only elements that are instances of `Prim` from `primitive`

`B` - Boxed arrays

Array of pointers to values

```
λ> let f i = if even i then Nothing else Just i
λ> makeArrayR B Seq (Ix1 5) f
(Array B Seq (5)
  [ Nothing,Just 1,Nothing,Just 3,Nothing ])
```

- Backed by `Array#`
- Can hold any haskell data type (including other arrays)
- Elements are strict (WHNF - weak head normal form)
- Slow read/write

```
λ> let a = makeArrayR B Seq (Ix1 3) (5 `div`)
λ> a ! 1
*** Exception: divide by zero
```

`N` - Strict boxed arrays

- Elements are strict (N - normal form)
- Require `NFData` instance

```
λ> let f i = if even i then Nothing else Just undefined
λ> let bArr = makeArrayR B Seq (Ix1 5) f
λ> bArr ! 2
Nothing
λ> bArr ! 1
Just *** Exception: Prelude.undefined
```

```
λ> let bArr = makeArrayR N Seq (Ix1 5) f
λ> bArr ! 2
*** Exception: Prelude.undefined
```

`S` - Storable arrays

- Backed by a pinned `ByteArray#`, i.e. GC will not move it.
- Will hold any element that is instance of `Storable` class
- Fast read/write
- Safe with FFI
- Subject to fragmentation.

`U` - Unboxed arrays

It's just a cute way of using primitive arrays to store more complex data types, for example tuples of primitive values:

```
[(1,2),(3,4),(5,6)] :: Array U Ix1 (Int, Int)
```

```
([1,3,4],[2,4,6]) :: (Array P Ix1 Int, Array P Ix1 Int)
```

- Same properties as with `P`, including performance.
- Elements that have `Unbox` constraint implemented.

Computation strategies

Note: must be compiled with `-threaded -with-rtsopts=-N`

```
λ> :t Seq
Seq :: Comp
```

```
λ> :t Par
Par :: Comp
```

```
λ> :t ParOn
ParOn :: [Int] -> Comp
```

```
class (Typeable r, Index ix) => Construct r ix e where
  getComp :: Array r ix e -> Comp
  setComp :: Comp -> Array r ix e -> Array r ix e
  unsafeMakeArray ::
     Comp -> ix -> (ix -> e) -> Array r ix e
```

Loading

```haskell
class Construct r ix e => Size r ix e where
  size :: Array r ix e -> ix
  ... -- also resize and extract
```

```haskell
class Size r ix e => Load r ix e where
  loadS :: Monad m =>
           Array r ix e -- ^ Array to load
        -> (Int -> m e) -- ^ Read from the result array
        -> (Int -> e -> m ()) -- ^ Write into the array
        -> m ()
  loadP :: [Int]
        -> Array r ix e
        -> (Int -> IO e)
        -> (Int -> e -> IO ())
        -> IO ()
```

Constructible and computable

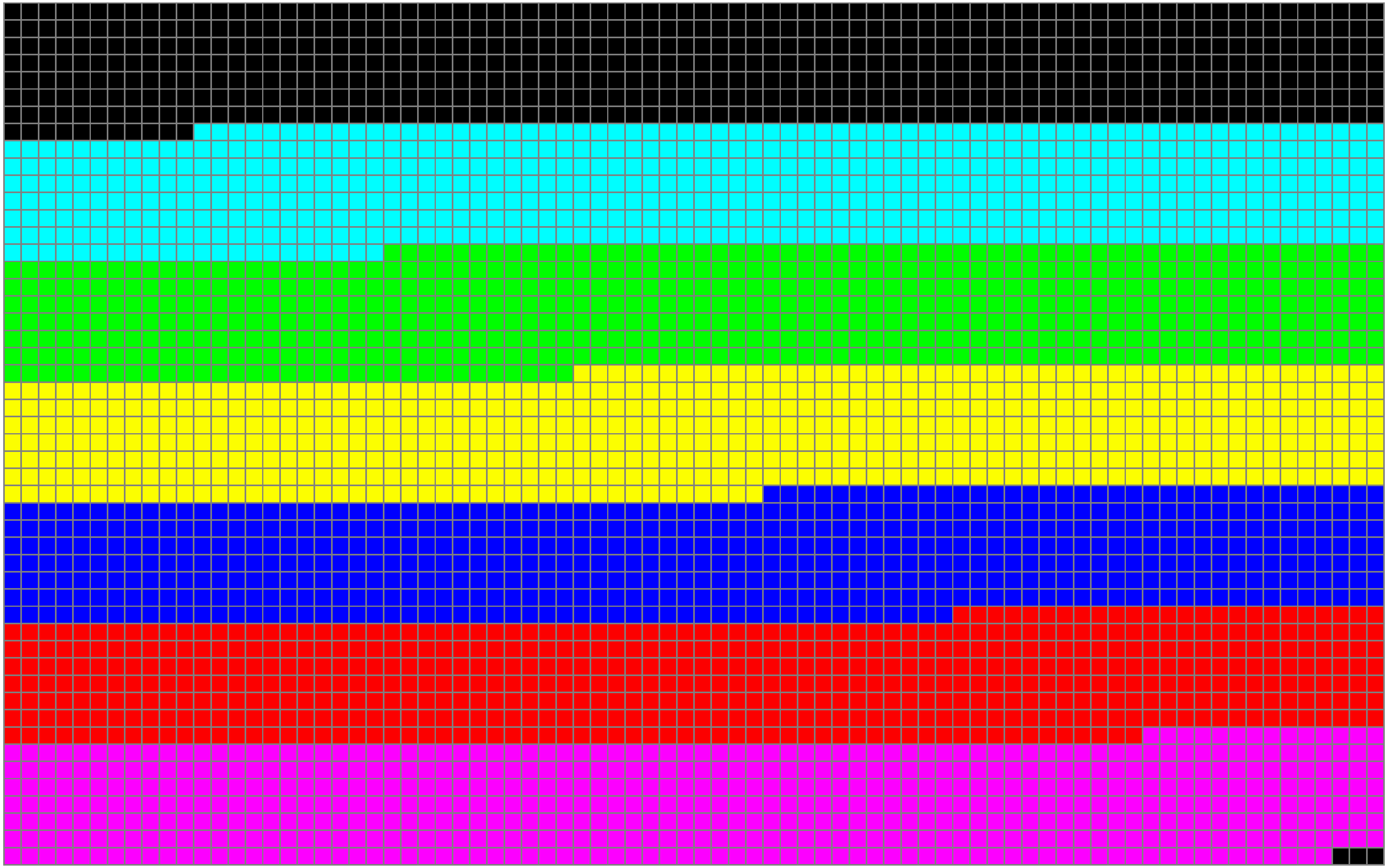
```haskell
makeArrayR :: Construct r ix e =>
  r -> Comp -> ix -> (ix -> e) -> Array r ix e
```

```haskell
computeAs :: (Load r' ix e, Mutable r ix e) =>
  r -> Array r' ix e -> Array r ix e
```

Example of loading:

```haskell
λ> let d = makeArrayR D (ParOn [0..6]) (50 :. 80) id
λ> :t computeAs U d
computeAs U d :: Array U Ix2 Ix2
```

D - Loading



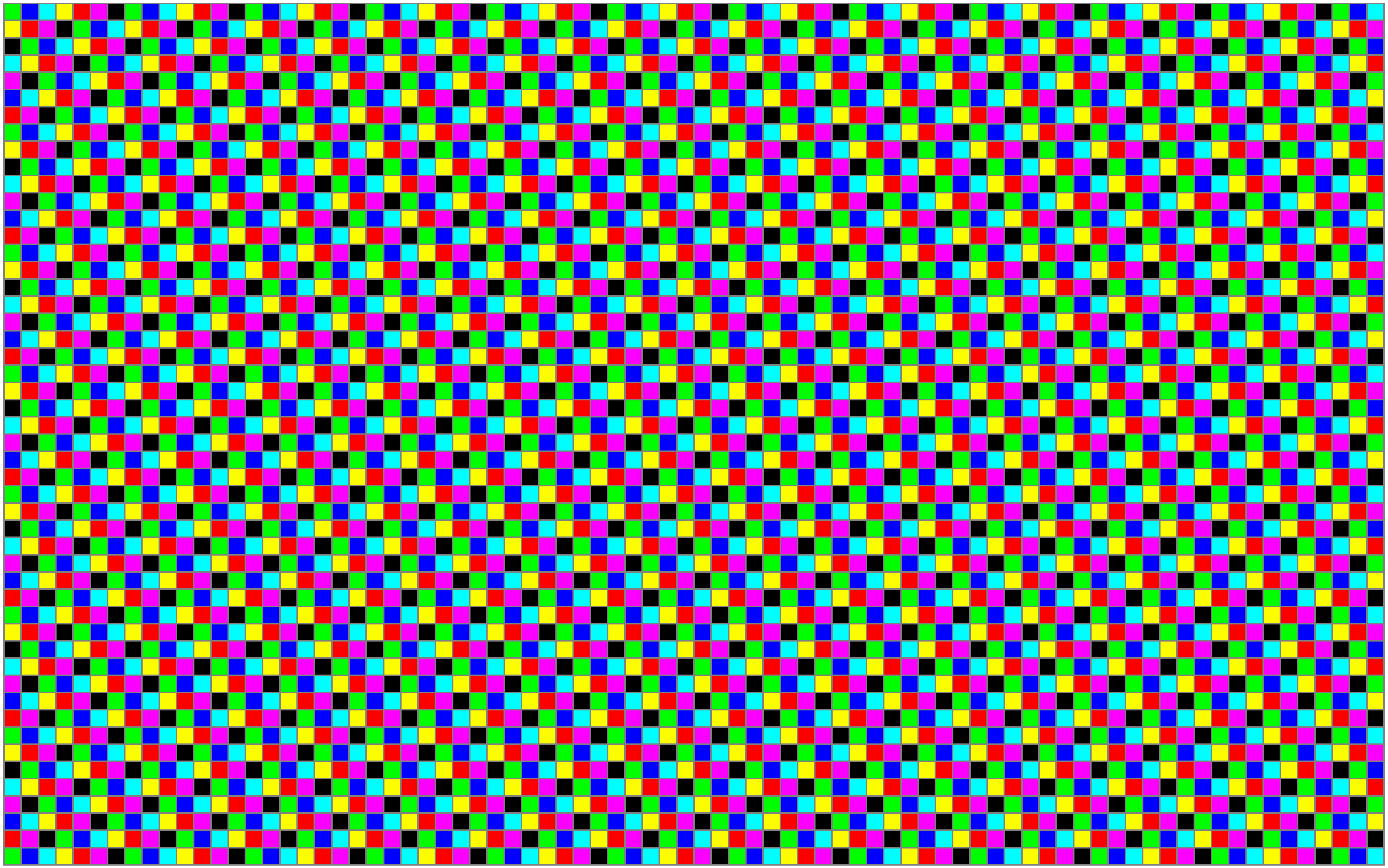
`DI` - Delayed Interleaved

For cases when computation is unbalanced (eg. Mandelbrot set):

```
computeAs U (toInterleaved d) :: Array U Ix2 Ix2
```

No pretty pictures, but hopefully ZuriHac will result some.

`DI` - Loading



Slicing

> Array slicing is an operation that extracts a subset of elements from an array and packages them as another array, possibly in a different dimension from the original.

Wikipedia

In massiv:

- slicing - always lowers dimension by one.
- extracting - stays in the same dimension.
- All are O(1) operation until computed.

Slicing from the outside

```
λ> let arr = makeArrayR U Seq (3 :> 2 :. 5) fromIx3
λ> arr
(Array U Seq (3 :> 2 :. 5)
  [ [ [ (0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4) ]
    , [ (0,1,0),(0,1,1),(0,1,2),(0,1,3),(0,1,4) ]
    ]
  , [ [ (1,0,0),(1,0,1),(1,0,2),(1,0,3),(1,0,4) ]
    , [ (1,1,0),(1,1,1),(1,1,2),(1,1,3),(1,1,4) ]
    ]
  , [ [ (2,0,0),(2,0,1),(2,0,2),(2,0,3),(2,0,4) ]
    , [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ]
    ]
  ])
λ> arr !> 2
(Array M Seq (2 :. 5)
  [ [ (2,0,0),(2,0,1),(2,0,2),(2,0,3),(2,0,4) ]
  , [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ]
  ])
λ> arr !> 2 !> 1
(Array M Seq (5)
  [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ])
```

Slicing from the inside

```
λ> arr
(Array U Seq (3 :> 2 :. 5)
  [ [ [ (0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4) ]
    , [ (0,1,0),(0,1,1),(0,1,2),(0,1,3),(0,1,4) ]
    ]
  , [ [ (1,0,0),(1,0,1),(1,0,2),(1,0,3),(1,0,4) ]
    , [ (1,1,0),(1,1,1),(1,1,2),(1,1,3),(1,1,4) ]
    ]
  , [ [ (2,0,0),(2,0,1),(2,0,2),(2,0,3),(2,0,4) ]
    , [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ]
    ]
  ])
λ> arr <! 2
(Array M Seq (3 :. 2)
  [ [ (0,0,2),(0,1,2) ]
  , [ (1,0,2),(1,1,2) ]
  , [ (2,0,2),(2,1,2) ]
  ])
λ> arr <! 2 <! 1
(Array M Seq (3)
  [ (0,1,2),(1,1,2),(2,1,2) ])
```

Slicing from within

```
(Array U Seq (3 :> 2 :. 5)
  [ [ [ (0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4) ]
    , [ (0,1,0),(0,1,1),(0,1,2),(0,1,3),(0,1,4) ]
    ]
  , [ [ (1,0,0),(1,0,1),(1,0,2),(1,0,3),(1,0,4) ]
    , [ (1,1,0),(1,1,1),(1,1,2),(1,1,3),(1,1,4) ]
    ]
  , [ [ (2,0,0),(2,0,1),(2,0,2),(2,0,3),(2,0,4) ]
    , [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ]
    ]
  ])
```

```
prop> arr !> i == arr <!> (rank (size arr), i)
prop> arr <! i == arr <!> (1,i)
```

```
λ> arr <!> (2, 0) -- dimensions start at 1
(Array M Seq (3 :. 5)
  [ [ (0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4) ]
  , [ (1,0,0),(1,0,1),(1,0,2),(1,0,3),(1,0,4) ]
  , [ (2,0,0),(2,0,1),(2,0,2),(2,0,3),(2,0,4) ]
  ])
```

Safe slicing and indexing

```
λ> arr !> 20 !> 1
(Array M *** Exception:
  (!>): Index out of bounds: 20 for Array of size: 3
```

```
λ> arr !?> 20 ??> 1
Nothing
```

```
λ> arr !?> 2 ??> 1
Just (Array M Seq (5)
  [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ])
```

```
λ> arr !?> 2 ??> 1 ??> 0
Just (2,1,0)
```

```
λ> arr !? (2 :> 1 :. 0)
Just (2,1,0)
```

`M` - Manifest arrays

- Constant time slicing and indexing

```
toManifest ::
  Manifest r ix e => Array r ix e -> Array M ix e
```

```
λ> let sl = arr !> 2 !> 1
λ> :t sl
sl :: Array M Ix1 (Int, Int, Int)
λ> sl ! 1
(2,1,1)
λ> sl !> 1
(2,1,1)
```

```
λ> delay arr !> 2 !> 1
(Array D Seq (5)
  [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ])
λ> delay arr !> 2 !> 1 !> 1 -- compile time error
λ> delay arr !> 2 !> 1 ! 1 -- compile time error
```

Extracting

```
λ> arr
(Array U Seq (3 :> 2 :. 5)
  [ [ [ (0,0,0),(0,0,1),(0,0,2),(0,0,3),(0,0,4) ]
    , [ (0,1,0),(0,1,1),(0,1,2),(0,1,3),(0,1,4) ]
    ]
  , [ [ (1,0,0),(1,0,1),(1,0,2),(1,0,3),(1,0,4) ]
    , [ (1,1,0),(1,1,1),(1,1,2),(1,1,3),(1,1,4) ]
    ]
  , [ [ (2,0,0),(2,0,1),(2,0,2),(2,0,3),(2,0,4) ]
    , [ (2,1,0),(2,1,1),(2,1,2),(2,1,3),(2,1,4) ]
    ]
  ])
λ> extract (1 :> 1 :. 1) (2 :> 1 :. 3) arr
Just (Array M Seq (2 :> 1 :. 3)
  [ [ [ (1,1,1),(1,1,2),(1,1,3) ]
    ]
  , [ [ (2,1,1),(2,1,2),(2,1,3) ]
    ]
  ])
```

resize ∘ extract ∘ slice

```
λ> arr!?>2 >>= extract (0:.1) (2:.3) >>= resize (3:.2)
Just (Array M Seq (3 :. 2)
  [ [ (2,0,1),(2,0,2) ]
  , [ (2,0,3),(2,1,1) ]
  , [ (2,1,2),(2,1,3) ]
  ])
```

- Index manipulations only, no new memory allocations.
- Easily composable.
- Safe, due to index checking and `Maybe` monad.

Pitfalls:

- Element lookup in the same location, means duplicate work.
- Use `computeSource` or `convertAs` to free up memory.

Folding

```
λ> A.sum $ range Par 0 10
45
```

```
λ> foldMono Product $ range Par 1 10
Product {getProduct = 362880}
```

```
fold :: Source r ix e =>
  (e -> e -> e) -> e -> Array r ix e -> e

foldlS :: Source r ix e =>
  (a -> e -> a) -> a -> Array r ix e -> a
```

- Automatic parallelization
- No memory allocation for the array
- Unless fold is monoidal in nature, transparent collection of parallel computation results isn't possible

Parallel directional folds

```
foldrP :: Source r ix e =>
   (e -> a -> a) -> a ->
   (a -> b -> b) -> b -> Array r ix e -> IO b
```

```
λ> foldrOnP [1..3] (:) [] (:) [] $ range Seq 0 10
[[0,1,2],[3,4,5],[6,7,8],[9]]
```

```
λ> foldrP (:) [] (++) [] $ range Seq 0 10
[0,1,2,3,4,5,6,7,8,9]
λ> toList $ range Seq 0 10
[0,1,2,3,4,5,6,7,8,9]
```

Side note: Nested parallel computation

```
λ> A.sum $ makeArrayR D Par 100 (A.sum . range Par 0)
161700
```

Stencil

> Stencil is a declarative way to describe how to compute an element of an array, by looking at an element in the same location of the source array as well as it's neighbors.

me

```
// TODO: handle elements near the border
for(i = 1; i < rows - 1; i++){
 for(j = 1; j < cols - 1; j++){
  res[i][j] =
   ( src[i-1][j-1] + src[i-1][j] + src[i-1][j+1] +
     src[i  ][j-1] + src[i  ][j] + src[i  ][j+1] +
     src[i+1][j-1] + src[i+1][j] + src[i+1][j+1] ) / 9;
 }
}
```

`Stencil` in massiv

```haskell
average3x3Filter ::
  (Default a, Fractional a) => Stencil Ix2 a a
average3x3Filter =
  makeStencil Edge (3 :. 3) (1 :. 1) $ \ get ->
  (get (-1 :. -1) + get (-1 :. 0) + get (-1 :. 1) +
   get ( 0 :. -1) + get ( 0 :. 0) + get ( 0 :. 1) +
   get ( 1 :. -1) + get ( 1 :. 0) + get ( 1 :. 1) ) / 9
{-# INLINE average3x3Filter #-}
```

```haskell
makeStencil
  :: (Index ix, Default e)
  => Border e -- ^ Border resolution technique
  -> ix -- ^ Size of the stencil
  -> ix -- ^ Center of the stencil
  -> ((ix -> Value e) -> Value a)
  -- ^ Stencil function that receives a "get" func...
  -> Stencil ix e a
```

Conway's Game of Life

```haskell
lifeRules :: Word8 -> Word8 -> Word8
lifeRules 0 3 = 1
lifeRules 1 2 = 1
lifeRules 1 3 = 1
lifeRules _ _ = 0

lifeStencil :: Stencil Ix2 Word8 Word8
lifeStencil =
  makeStencil Wrap (3 :. 3) (1 :. 1) $ \ get ->
    lifeRules <$> get (0 :. 0) <*>
      (get (-1 :. -1) + get (-1 :. 0) + get (-1 :. 1) +
       get ( 0 :. -1)         +         get ( 0 :. 1) +
       get ( 1 :. -1) + get ( 1 :. 0) + get ( 1 :. 1))

life :: Array S Ix2 Word8 -> Array S Ix2 Word8
life = compute . mapStencil lifeStencil
```

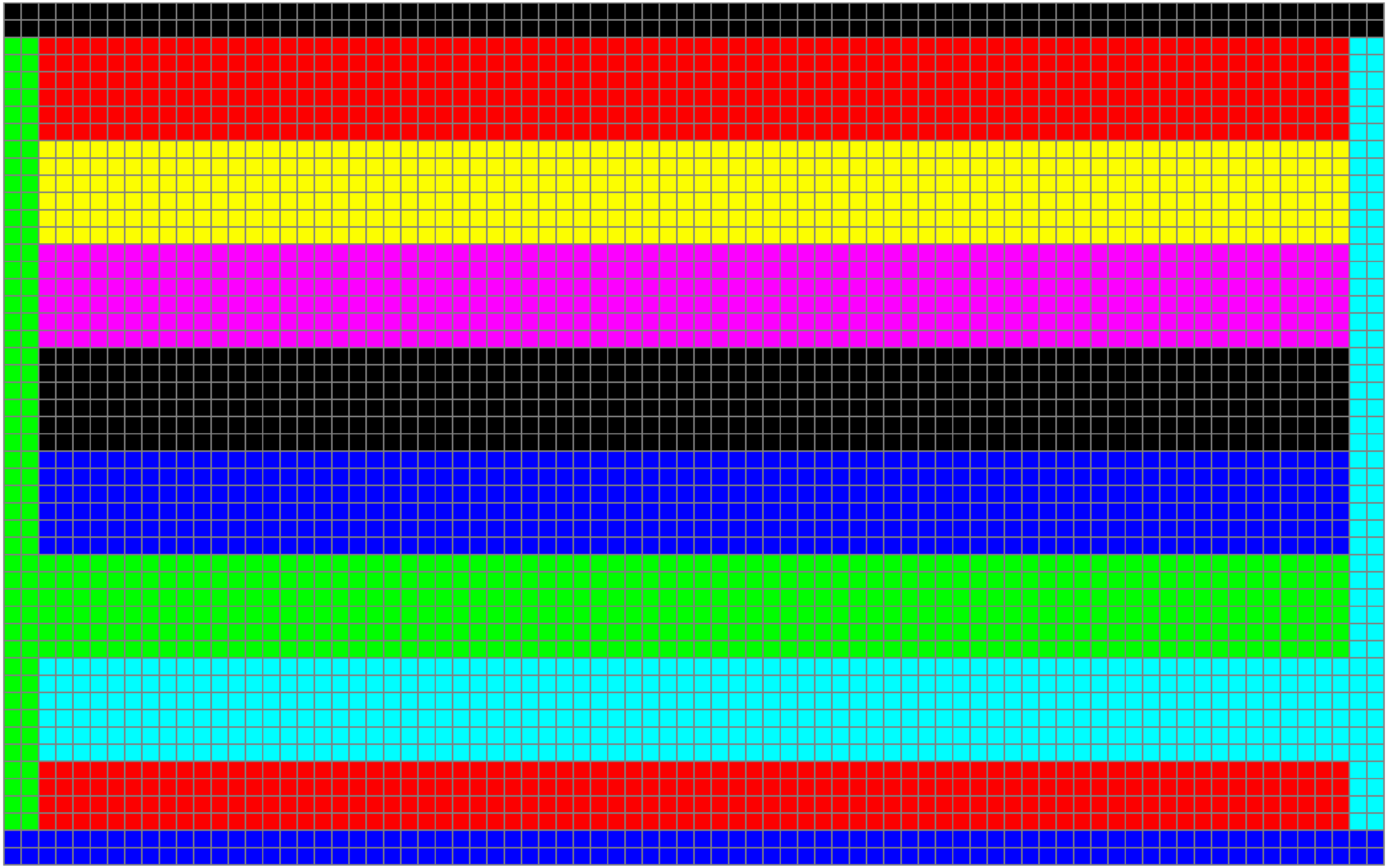
Performance gain from Stencil

The naïve approach would be to `imap` an index aware function.

```
mapStencil :: Manifest r ix e =>
     Stencil ix e a -> Array r ix e -> Array DW ix a
```

- Bounds are checked only near the border
- Cache friendly iteration
- Handy `Border` resolution techniques (`Fill e`, `Mirror`, etc.)

`DW` - Delayed Windowed arrays



The purpose of `Default` and `Value`.

```
badStencil :: Stencil Ix1 Int Int
badStencil =
  makeStencil Edge 3 1 $ \ get -> get 0 + get 2
```

```
λ> mapStencil badStencil $ computeAs U $ range Seq 0 10
(Array DW *** Exception:
 Index is out of bounds: 3 for stencil size: 3
```

```
dangerousStencil :: Stencil Ix1 Int Int
dangerousStencil =
  makeStencil Edge 3 1 $ \ get -> get (get 0)
```

Benchmarks

Environment for benchmarks is:

- 3rd gen quad core i7 and 32Gb DDR3 RAM
- Ubuntu 16.04 LTS
- GHC 8.4.2 (Stackage nightly-2018-05-27)

In order to run the same benchmarks look at the `monadic-warsaw` branch on github and run:

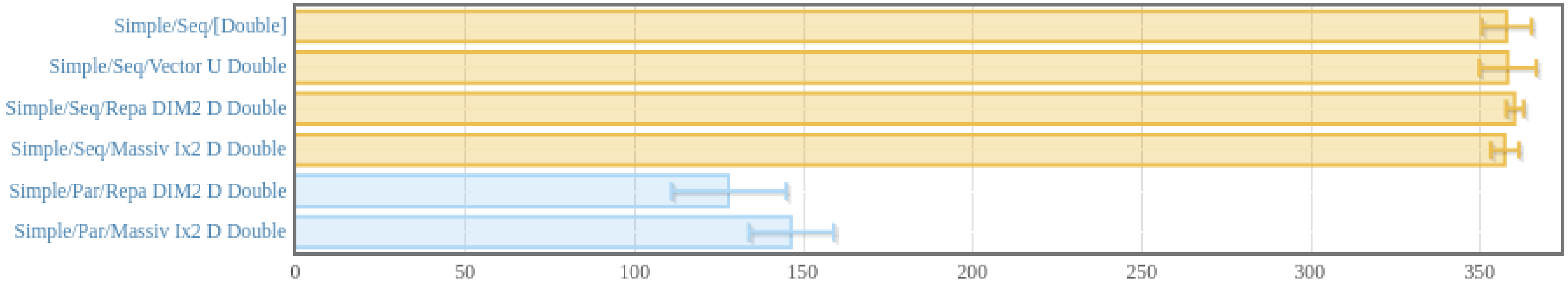
```
$ cd massiv-bench
$ stack --stack-yaml stack-ghc-8.4.yaml \
    bench massiv-bench:mw
```

Previous versions of GHC:

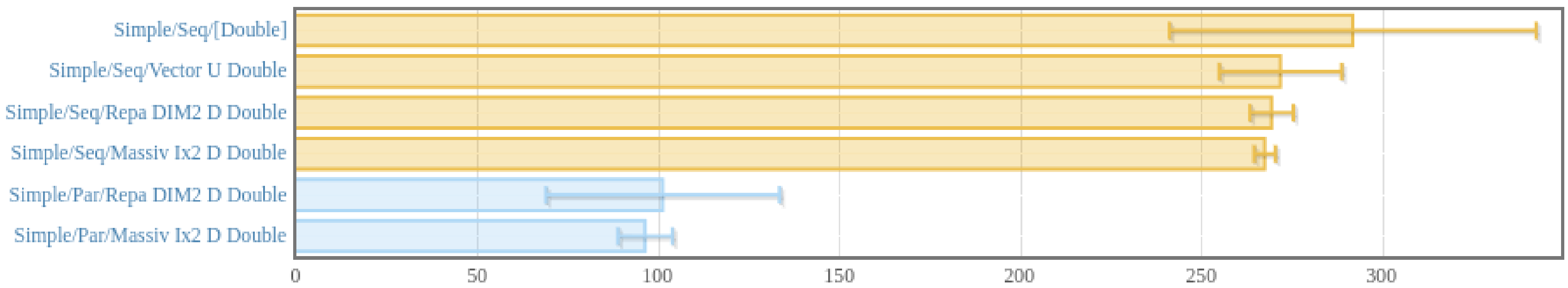
- GHC 8.2.2:
  - `vector` library performance was shot
  - Massiv and Repa performance wasn't affected
- GHC 8.0.2:
  - Massiv is much slower on some operations. I didn't bother investigated too much.
- GHC 7.10:
  - Benchmarks are not even run.
  - Massiv test suite does pass, so it should theoretically be usable.

Simple

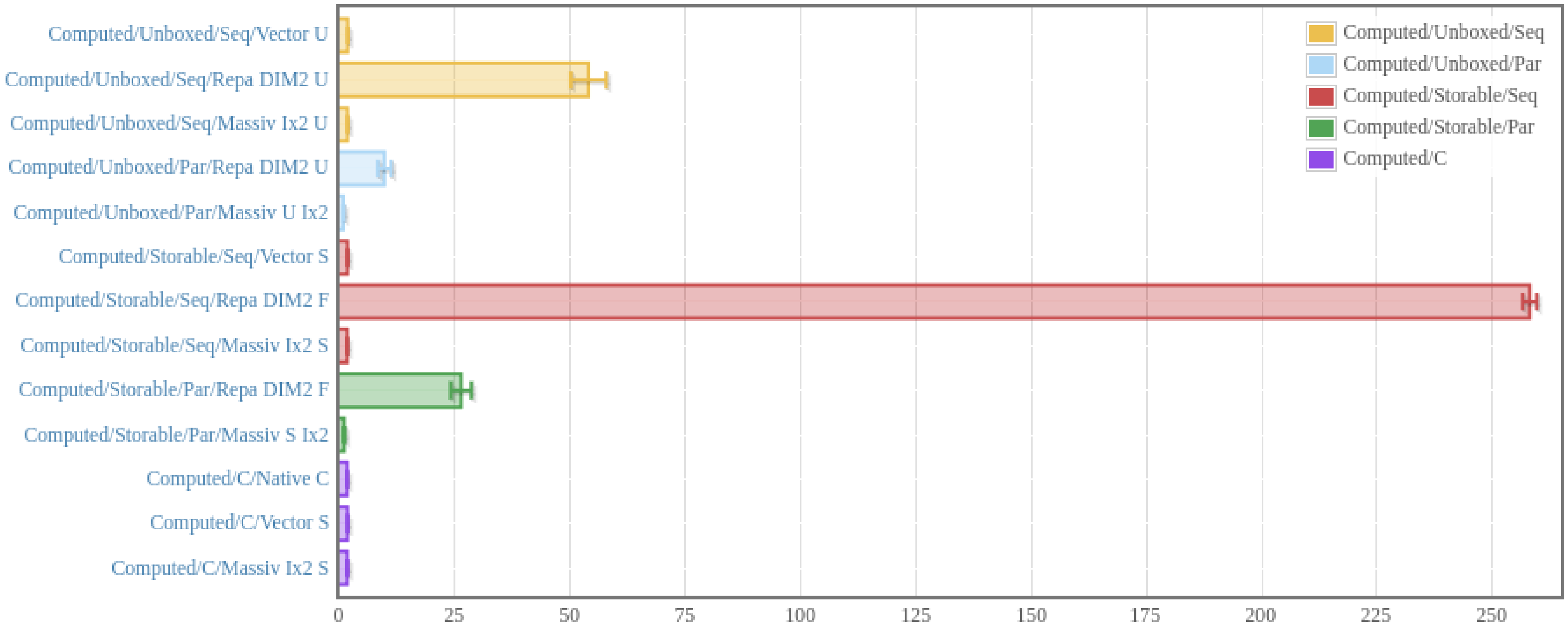
Sum first `320000` numbers as `Double` with fusion.



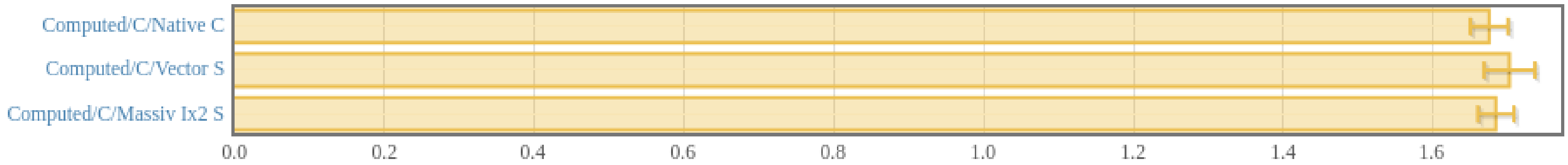
Using LLVM-5.0 backend (`-fllvm -optlo-O3`)



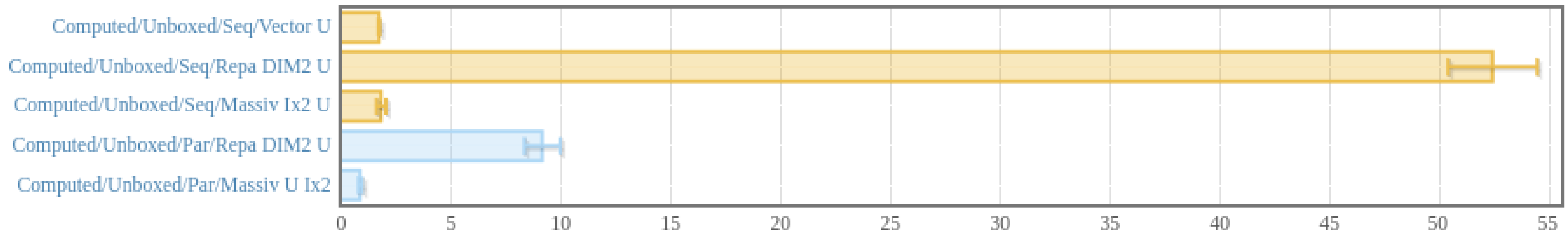
Computed



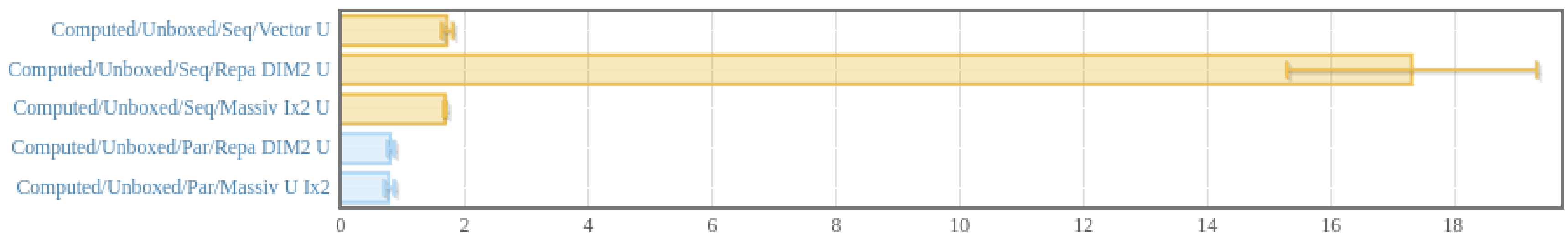
Sanity check, comparison to C:



Computed (Unboxed)

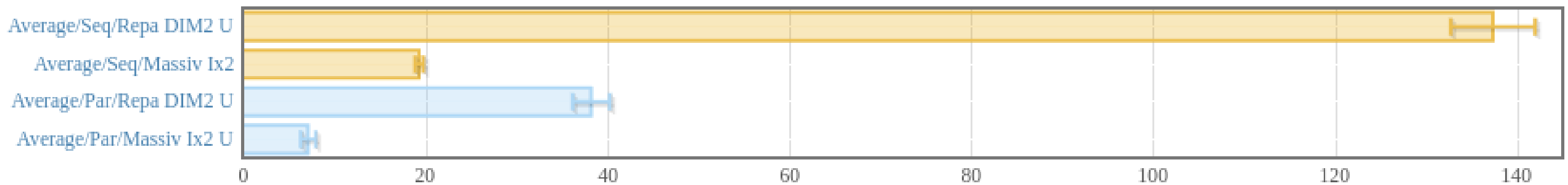


Compiled with LLVM

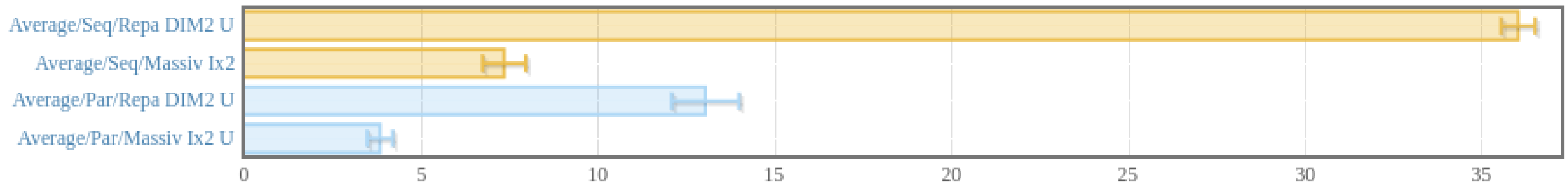


Turns out Repa does rely heavily on LLVM.

Average Stencil

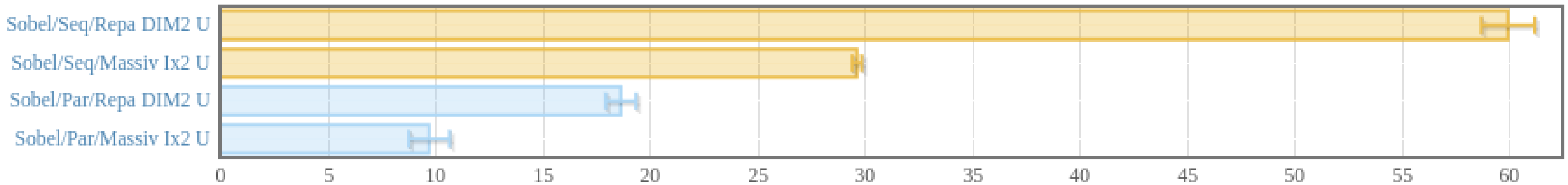


Compiled with LLVM

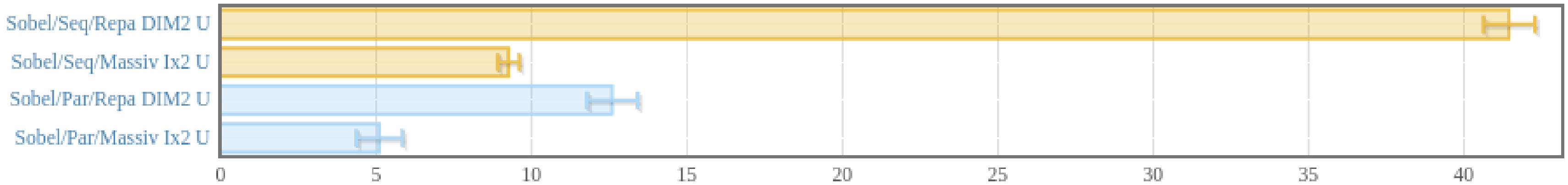


Sobel operator

Benchmarking stencil convolution (Repa has no generic stencils)

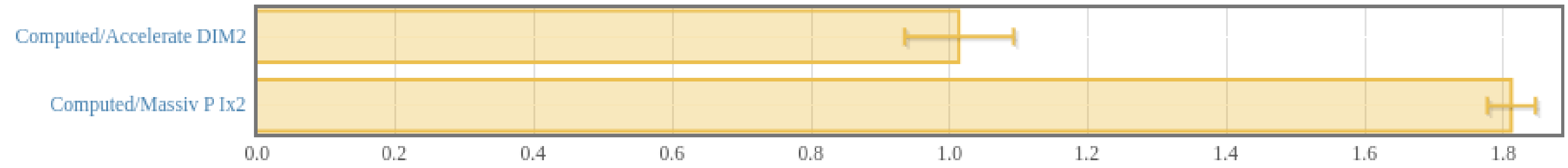


Compiled with LLVM

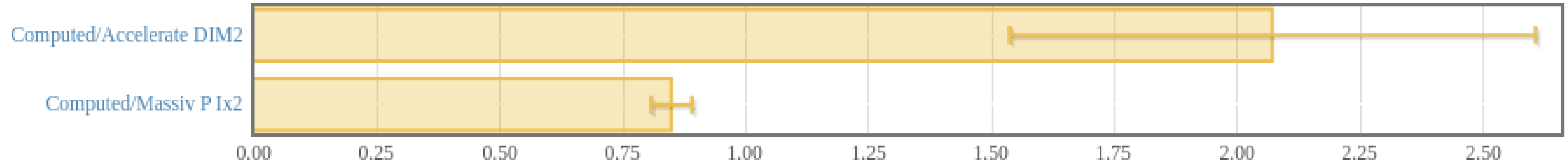


Accelerate (Sum)

- Sequential

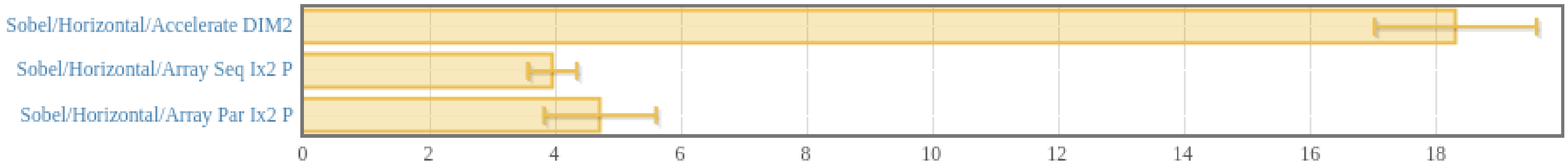


- Parallel

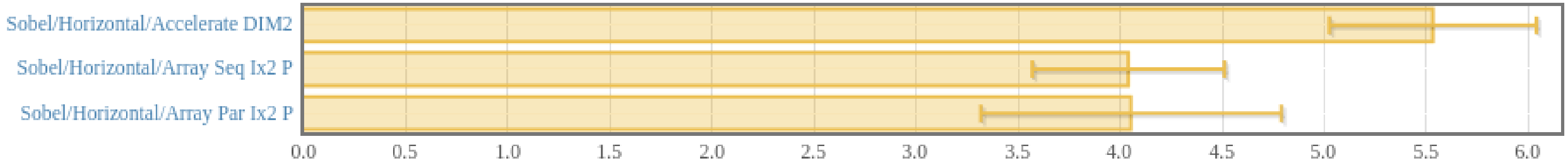


Accelerate (Sobel Horizontal Stencil)

- Sequential



- Parallel



The outcome

- A pretty fast array manipulation library
- Automatic parallelization
- Manual fusion
- Efficient slicing
- Directional parallel folding
- Fully featured Stencil support
- Very easy to use (biased opinion)

Thank you

Questions?

Extra

Automatic fusion

```
data Massiv ix e = Massiv (Array S ix e)

computeM :: (Load r ix e, Storable e) =>
  Array r ix e -> Massiv ix e
computeM = Massiv . compute
{-# INLINE [1] computeM #-}

delayM :: (Storable e, Index ix) =>
  Massiv ix e -> Array D ix e
delayM (Massiv arr) = delay arr
{-# INLINE [1] delayM #-}

{-# RULES
"delayM/computeM" forall arr . delayM (computeM arr) = ar
 #-}

imapMassiv :: (Index ix, Storable e', Storable e) =>
  (ix -> e' -> e) -> Massiv ix e' -> Massiv ix e
imapMassiv f = computeM . imap f . delayM
{-# INLINE [~1] imapMassiv #-}
```