Data Parallel Programming with Repa

Stephen A. Edwards

Columbia University

Fall 2022

Iowa

Arrays

- The Array Type: Shapes
- Element Indices
- Delayed Arrays and computeS
- fromFunction

Example: Shortest Paths on a Dense Graph

- The Floyd-Warshall Algorithm

Data-parallel, arithmetic-heavy algorithms over large arrays

REPA = REgular Parallel Arrays `http://repa.ouroborus.net/`

Good array performance demands near-zero overhead per element

Strategies need lazy data structures; Par uses IVars.

Repa automatically parallelizes array computations

```yaml
# stack.yaml
extra-deps:
- repa-3.4.1.4
```

```yaml
# package.yaml
executables:
  repa-demo-exe:

    dependencies:
    - repa
```

Import the Repa module

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

Some overlaps with the Prelude, so use, e.g., `Repa.map`, or specify exactly what you want to import, e.g.,

```
import Data.Array.Repa ( Array )
```

The Array Type: Includes the number of dimensions

```
data Array representation shape elements
```

Different ways to represent arrays in memory, e.g., `U` indicates unboxed

Haskell Types can't (yet) include numbers; the shape (number of dimensions) encoded as a *list of type constructors*

```
{-# LANGUAGE TypeOperators #-}

data Z           = Z           -- Zero-dimensional: a scalar
data tail :. head = tail :. head -- head: type of the dimension (Int)

type DIM0 = Z            -- Scalar
type DIM1 = DIM0 :. Int  -- Vector
type DIM2 = DIM1 :. Int  -- 2D Matrix
type DIM3 = DIM2 :. Int  -- 3D Array
```

First Example

```
*Main> :set -XTypeOperators
*Main> :t fromListUnboxed
fromListUnboxed
  :: (Shape sh, Unbox a) => sh -> [a] -> Array U sh a
*Main> fromListUnboxed (Z :. 10) [1..10] :: Array U (Z :. Int) Int
AUnboxed (Z :. 10) [1,2,3,4,5,6,7,8,9,10]
```

`Z` and `:.` are both *type constructors* and *data constructors*

`Z :. Int :. Int` is the type for the shape of a 2-dimensional array

`Z :. 5 :. 3` is the index for the element at row 5, column 3.

Matrices (2D Arrays)

Arrays are 1D vectors internally

```
*Main> fromListUnboxed (Z :. 3 :. 5) [1..15] :: Array U DIM2 Int
AUnboxed ((Z :. 3) :. 5) [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$$

Accessing a single element

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 \\ 11 & 12 & 13 & 14 & 15 \end{bmatrix}$$

Column/row indices start from 0

```
*Main> :t (!)
(!) :: (Shape sh, Source r e) => Array r sh e -> sh -> e
*Main> let a = fromListUnboxed (Z:.3:.5) [1..15] ::Array U DIM2 Int
*Main> a ! (Z :. 1 :. 3)
9
*Main> :t toIndex
toIndex :: Shape sh => sh -> sh -> Int
*Main> toIndex (Z :. 3 :. 5 :: DIM2) (Z :. 1 :. 3 :: DIM2)
8
```

Querying the Shape of an Array

```
*Main> let a = fromListUnboxed (Z:.3:.5) [1..15] ::Array U DIM2 Int
*Main> :t extent
extent :: (Source r e, Shape sh) => Array r sh e -> sh
*Main> :t extent ar
extent ar :: DIM2
*Main> extent a                   -- The shape
(Z :. 3) :. 5

*Main> :t rank                    -- Number of dimensions as an Int
rank :: Shape sh => sh -> Int
*Main> rank (extent a)
2

*Main> :t size                    -- Number of elements
size :: Shape sh => sh -> Int
*Main> size (extent a)
15
```

Operations on Arrays

Repa is designed to perfom operations *across* arrays, not on single elements.
It uses compile-time array operation fusion

A `D` (Delayed) array hasn't yet been computed.

`computeS` can turn a Delayed into an Unboxed:

```
*Main> let a = fromListUnboxed (Z:.10) [1..10] :: Array U DIM1 Int

*Main> :t Repa.map        -- Produces an Array D ..
Repa.map :: (Shape sh, Source r a) =>
            (a -> b) -> Array r sh a -> Array D sh b

*Main> :t Repa.map (+1) a
Repa.map (+1) a :: Array D DIM1 Int

*Main> computeS (Repa.map (+1) a) :: Array U DIM1 Int
AUnboxed (Z :. 10) [2,3,4,5,6,7,8,9,10,11]
```

fromFunction: Creating Delayed Arrays

```
*Main> :t fromFunction
fromFunction :: sh -> (sh -> a) -> Array D sh a

*Main> let a = fromFunction (Z:.10) (\(Z:.i) -> i * 10 :: Int)
*Main> :t a
a :: Array D (Z :. Int) Int

*Main> a ! (Z:.5)
50

*Main> computeS a :: Array U DIM1 Int
AUnboxed (Z :. 10) [0,10,20,30,40,50,60,70,80,90]
```

Roll-your-own Map over Arrays

```
*Main> let mymap f a = fromFunction (extent a) (\i -> f (a ! i))
*Main> :t mymap
mymap :: (Source r t, Shape sh) =>
         (t -> a) -> Array r sh t -> Array D sh a

*Main> let a = fromFunction (Z:.10) (\(Z:.i) -> i * 10 :: Int)
*Main> :t a
a :: Array D (Z :. Int) Int
*Main> :t mymap (+1) a
mymap (+1) a :: Array D (Z :. Int) Int
*Main> computeS (mymap (+1) a) :: Array U DIM1 Int
AUnboxed (Z :. 10) [1,11,21,31,41,51,61,71,81,91]
```

Floyd-Warshall Shortest Paths on a Dense Graph

In pseudocode:

```
shortestPath :: Graph -> Vertex -> Vertex -> Vertex -> Weight
shortestPath g i j 0 = weight g i j
shortestPath g i j k = min (shortestPath g i j (k-1))
                           (shortestPath g i k (k-1) +
                            shortestPath g k j (k-1))
```

An adjacency matrix is good for *dense* graphs

Sequential Implementation

```
type Weight = Int
type Graph r = Array r DIM2 Weight

shortestPaths :: Graph U -> Graph U
shortestPaths g0 = go g0 0
  where
    Z :. _ :. n = extent g0     -- Get # of vertices

    go !g !k | k == n    = g    -- Reached the end
             | otherwise = let  -- Compute new minimums
                   g' = computeS (fromFunction (Z:.n:.n) sp)
                 in go g' (k+1) -- Increase k and repeat
     where sp (Z:.i:.j) =
             min (g ! (Z:.i:.j))                    -- i → j
                 (g ! (Z:.i:.k) + g ! (Z:.k:.j)) -- i → k → j
```

Making it Parallel

`computeP` runs in a monad to enforce sequential operation

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

Making it Parallel

```
import Data.Functor.Identity

shortestPaths :: Graph U -> Graph U
shortestPaths g0 = runIdentity $ go g0 0
  where
    Z :. _ :. n = extent g0

    go !g !k | k == n    = return g -- Return in the monad
             | otherwise = do       -- We're in a monad
                g' <- computeP (fromFunction (Z:.n:.n) sp)
                go g' (k+1)
     where sp (Z:.i:.j) =
             min (g ! (Z:.i:.j))
                 (g ! (Z:.i:.k) + g ! (Z:.k:.j))
```

Compiling it efficiently

```
executables:
  fwdense1:
    main: fwdense1.hs
    source-dirs: app
    ghc-options:
    - -threaded         # Turn on parallelism
    - -rtsopts          # Enable +RTS
    - -O2               # Enable optimization
    - -fllvm            # Enable LLVM backend
    dependencies:
    - repa
```

Statistics

Without `-fllvm`

Command-line	Time (s)
fwdense 500	2.808
fwdense1 500 +RTS -N1	3.110
fwdense1 500 +RTS -N2	1.690
fwdense1 500 +RTS -N4	1.160
fwdense1 500 +RTS -N5	1.180
fwdense1 500 +RTS -N6	1.041
fwdense1 500 +RTS -N7	0.911
fwdense1 500 +RTS -N8	1.321

With `-fllvm`

Command-line	Time (s)
fwdense 500	1.692
fwdense1 500 +RTS -N1	2.100
fwdense1 500 +RTS -N2	1.130
fwdense1 500 +RTS -N4	0.820
fwdense1 500 +RTS -N5	0.781
fwdense1 500 +RTS -N6	0.681
fwdense1 500 +RTS -N7	0.731
fwdense1 500 +RTS -N8	0.961