# Image filtering in Repa 3

Adam Sandberg Eriksson  Andreas Svanström

May 4, 2015

## 1 Introduction to Repa

### 1.1 What is Repa

Repa is a library for parellelising computations using regular arrays, the name REPA is an acronym for REgular PArallel arrays.

The cool thing with Repa is that as long as you run a program, written using the repa combinators, with the flags +RTS -Nx (where x is the number of threads), the program will automatically run the combination computations in parallel.

The Repa arrays can be multidimensional and this is defined by different shapes, they also have the two options of being stored as Delayed or Unboxed manifest. Delayed means that lazy evaluation will be used (i.e. the array is built up of functions that are computed when the respective values are needed), Unboxed manifest on the other hand means that the actual values are computed and stored in the array. This means that an array where a majority of the values will be accessed multiple times probably should be stored as Unboxed manifest to improve performance.

### 1.2 How to use Repa

A repa array is defined as

```
Array r sh a
```

where r can be either U or D, defining if the array is Unboxed manifest or Delayed. The sh parameter defines the shape of the array, i.e. the number of dimensions it has, and possibly their sizes. Lastly the a parameter defines which element type the array will contain, so if one would want to create a three-dimensional 2x3x3 delayed array with Integer elements, one could define it as

```
Array D (Z :. 2 :. 3 :. 3) Integer
```

If one wouldn't know yet how big each dimension should be, and would want to create an unboxed three-dimensional array of doubles, it could be defined as

```
Array U DIM3 Double
```

To get the element from a certain index in an array, one uses the `!` operator together with a shape, let's say that we have a two-dimensional array called nums, and that we wanted the element at `(5, 0)`, then we would get it by

```
nums ! (Z :. 5 :. 0)
```

similar to

```
nums !! 5 !! 0
```

would it be a normal Haskell list.

There are functions for easy creation of Repa arrays from Haskell lists or functions. For example if we dould like to create a one-dimensional unboxed array with the numbers 0-10, we would use the `fromListUnboxed` function, which takes a shape and a list as arguments, like this

```
zeroToTen = fromListUnboxed (Z :. 11) [0..10]
```

If we instead would like this array as a delayed one, we would use the `fromFunction` function, which takes a shape and a function, like this

```
zeroToTen = fromFunction (Z :. 11) (\n -> (size n))
```

The Repa library contains its own versions of some common functions, like `map` and `zipWith`, which enables normal "list operations" on Repa arrays. Since these often, as in the two cases mentioned, have the same names as the list functions in the Prelude, it is necessary to either import the Repa modules qualified or hide the functions from the Prelude.

### 1.3 Image manipulation with Repa

The `repa-io` library has built-in support for reading and writing bitmap images with the functions `readImageFromBMP` and `writeImageToBMP`. The Repa representation of bitmaps is a two-dimensional unboxed array of three-tuples of 8-bit words, meaning one element for each pixel, consisting of the RGB-value tuple. To manipulate an image, one simply changes the values of these Word8 colour values.

Because of the excellent parallelism scaling that Repa has when working on the individual elements in a pixel, Repa is very good for writing image manipulation functions for filters etc. that can operate on each pixel independently.

## 2 Image filters with Repa

We start with some imports. First we need to hide some prelude imports:

```
import Prelude hiding (map, zip3, unzip3)
import Data.Word (Word8)
```

And then we need some imports from the `repa` and `repa-io` packages

```
  -- from repa
import Data.Array.Repa
import Data.Array.Repa.Stencil
import Data.Array.Repa.Stencil.Dim2
import Data.Array.Repa.Repr.Unboxed
  -- from repa-io
import Data.Array.Repa.IO.BMP
```

`repa-io` will read `BMP` files into a triple of `Word8`'s but mostly we wish to work with numbers in the range $[0, 1]$, so we create some convenience functions `bmpToDouble` and `doubleToBmp` to help us.

```
type Image = Array U DIM2 (Double, Double, Double)
type Filter = forall m.Monad m ⇒ Image → m Image
  -- convert a rgb in [0, 255] to [0, 1]
bmpToDouble :: Monad m
  ⇒ Array U DIM2 (Word8, Word8, Word8)
  → m Image
```

```
bmpToDouble image =
  computeP (map (λ(r, g, b) →
    (toDouble r, toDouble g, toDouble b))
    image)
  where toDouble c =
    fromRational (toRational c / 255)
  -- convert from [0,1] to [0,255]
doubleToBmp :: Monad m
   ⇒ Image
   → m (Array U DIM2 (Word8, Word8, Word8))
doubleToBmp image =
  computeP (map (λ(r, g, b) →
    (toWord r, toWord g, toWord b))
    image)
  where toWord c = round (c ∗ 255)
  -- check if index is inside radius
inside :: Int → DIM2 → Bool
inside r (Z :. x :. y) = abs x ⩽ r ∧ abs y ⩽ y
 {-# INLINE inside #-}
  -- calulcate the extent from a radius
rToExtent :: Int → DIM2
rToExtent r = ix2 rr rr
  where rr = 2 ∗ r + 1
 {-# INLINE rToExtent #-}
```

We define `applyFilter` to read a `BMP` file, apply a filter and write to a new file:

```
applyFilter :: Filter → FilePath → FilePath → IO ()
applyFilter filter inFile outFile =
  do im ←
    fmap (either (error.show) id)
      (readImageFromBMP inFile)
    im′ ← bmpToDouble im
    filtered ← filter im′
    outIm ← doubleToBmp filtered
    writeImageToBMP outFile outIm
```

We begin with a simple filter: converting an image to grey scale. A nice way to translate colour into grey is to use a linear combination of the red, green and blue but deemphasising the red and the blue slightly, we copy the resulting grey scale to all three channels. In Repa we write:

```
greyscale :: Filter
greyscale image =
  do im ←
    computeP (map (λ(r, g, b) → 0.2126 ∗ r + 0.7152 ∗ g + 7.22e−2 ∗ b) image)
    return (zip3 im im im)
```

The filter has the effect we are looking for:

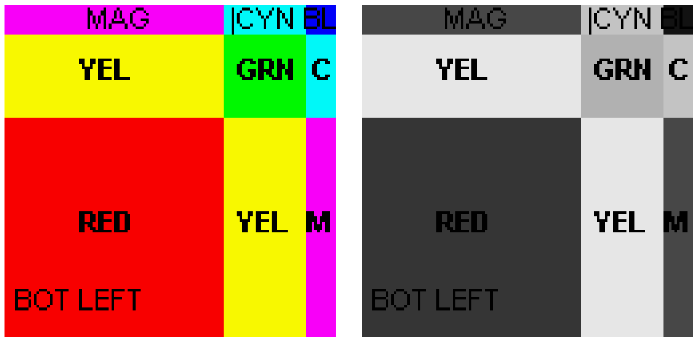

Figure 1: Image before and after greyscale conversion

To make more interesting filters we will use a Repa feature called `Stencils`, which are used to apply a matrix of coefficients to a pixel and its neighbours.

Such filters that are used to decrease the noise in images are called blur filters, first we implement a uniform blur filter that sums a pixel and its neighbours with equal weight:

```
simpleBlurStencil :: Int → Stencil DIM2 Double
simpleBlurStencil r =
  makeStencil
    (rToExtent r)
    (λix →
      if inside r ix
        then Just coeff
        else Nothing)
  where coeff = recip (fromIntegral (2 ∗ r + 1) ∗∗ 2)
```

To simplify the application of the stencil to the three channels of the image we define

```
applyStencil :: Monad m
   ⇒ Stencil DIM2 Double → Image → m Image
applyStencil stencil image =
  do let (r, g, b) = unzip3 image
    [r′, g′, b′] ←
      mapM (computeP.appSt)
        [r, g, b]
    return (zip3 r′ g′ b′)
  where appSt = mapStencil2 BoundClamp stencil
```

Finally we apply the uniform blur filter to an image

As we can see the image is very blurry and not very nice. A better blur filter is a Gaussian blur filter, it uses a 2 dimensional Gaussian to compute the stencil, this way pixels that are further away from the origin pixel get less weight and we blur the image in a nicer way.

We start with the Gaussian

```
gaussian2d :: Double → Int → Int → Double
gaussian2d sigma x y =
  (recip (pi ∗ s2) ∗
    exp (−(fromIntegral x ∗∗ 2 + fromIntegral y ∗∗ 2) / s2))
  where s2 = 2 ∗ sigma ∗∗ 2
 {-# INLINE gaussian2d #-}
```

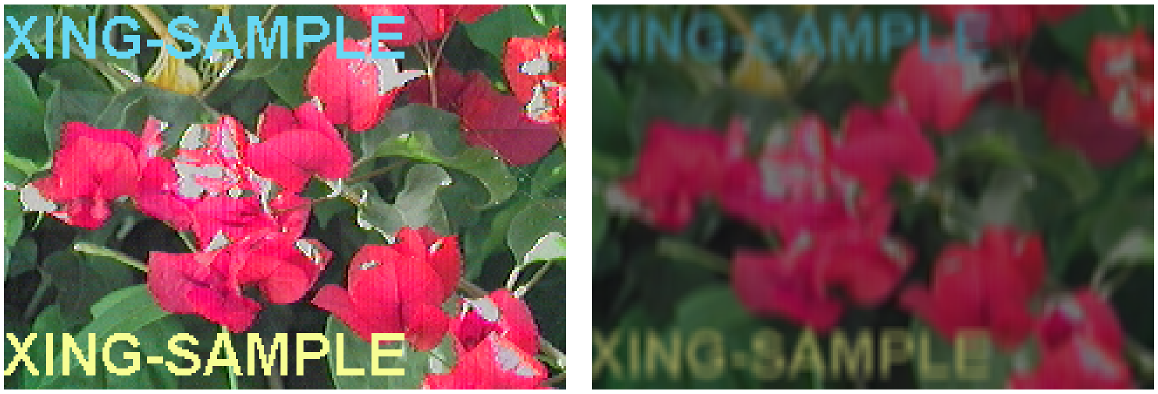

Figure 2: Image before and after an uniform blur

and for the stencil we compute the the Gaussian at each coordinate

```
gaussianStencil :: Double → Int → Stencil DIM2 Double
gaussianStencil sigma r =
  makeStencil
    (rToExtent r)
    (λix@(Z :. x :. y) →
      if inside r ix
        then Just (gaussian2d sigma x y)
        else Nothing)
```

Using this blur stencil we get a much nicer effect:

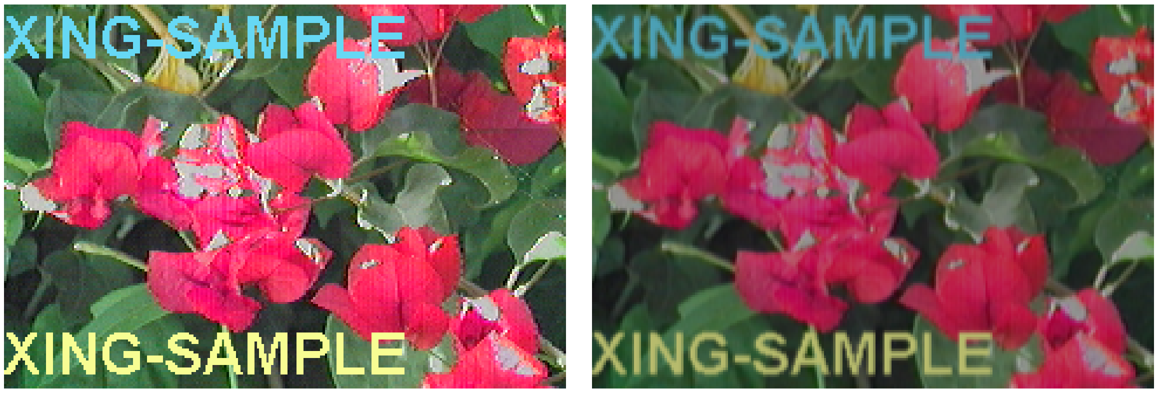

Figure 3: Image before and after a Gaussian blur