Progress Report 2

Nested Data Parallelism for
Image Processing Algorithms

Chandrakant Swaneet Kumar Sahoo

25.05.2015

Summary This is the second progress report on my thesis. After giving a summary of my work and my fuutre plans, I show my remaining tasks and give a impression of the algorithms and the code transformation.

Contents

1 Leading question

Given a slightly irregular algorithm like Histogram Balancing.

How can we implement this high-level irregularly-parallel algorithm such that it compiles to efficient mashine level code?

Subquestions

- ~~How much faster~~ *How easy to code and efficient in runtime* is a parallel variant against the sequential one?
- ~~How much faster~~ *How easy to code and efficient in runtime* is a compiled variant to human-written low-level parallel code?

Programs The following name for the source codes are given. These programs are already written/transformed.

P_{seq} A sequential implementation using arrays of pointers and maps written in $<$ 30min.

P_{man} An advanced implementation which tries to focus on efficiency. it uses simple top-level parallization and bytearrays to achive this. also written in $<$ 30min.

P_{nest} An advanced implementation which can parallelize on every level. Form the view of the programer, it seems to use nested pointer-based arrays called `[:a:]`. They are very inefficient if implemented directly - however, $P_{nest}$is automatically transformed to $P_{vect}$and optimised by the compiler before its executed. also written in $<$ 30min.

P_{vect} The highly optimized program generated by the compiler. It uses an type-dependent pointer-avoiding flat representation of the user array. I needed $>$ 30 hours to make this transformation. And there is still more space to optimise.

2 Previous work

I have implemented all three forms of the algorithm and have almost finished transformations and optimizations done by the compiler. The optimizations by the compiler is an almost never-ending task, as the compiler can produce so much code, such that it were a few meters high if printed on paper.

Initially, I was considering to show the benefits in a scalar and lifted call of the histogram balancing function. The function `hbalanceBulk` in the P_{nest}-implementation - which looks exactly like the other implementations - is actually more expressive than it looks like. I had planned to show the transformed version of this code to show which optimizations it can achieve. However, the program transformation is quite tiresome (see. Section "Code Transformation") and I abandoned this idea after a week(`(it would double-my already stuffed work...)`). However, I think that this is an important aspect.

I have also started analysing the algorithms and did some early (inaccurate) analysis. They can be found in the accompanied document `(cost-centre.pdf)`

Please give me feedback on the algorithms, my current progress and especially on further analysis of `hbalanceBulk`!! I can still give some basic analysis of the bulk-application case, even though I won't manually finish its transformation.

3 Future work

As my next important step. I will make an more accurate program analysis- for that I will also finish the last optimizations I can do on the transformed program.

And after that: I will write down all my work until the current point, maybe including the general introduction. It will allow me to retrace any mistakes I have done yet. This task will also significantly benefit the quality of my thesis text, since I can now also focus on collecting all of any tiny bits. I have already written a bit in this progress report and the accompanying cost centre document.

4 Progress

- 80% Read more papers on Nested Data Parallel Haskell
- 60% Read more papers on Analysis of Parallel Progrmas
- 100% Decide on an algorithm (Histogram Balancing)
- 90% Program Transformation:
  - 100% Desugar
  - 100% Vectorization
  - 70% Inlining & Fusioning (means Optimization)
- 100% Implement sequential variant
- 100% Implement manually-parallelized variant
- 100% Fix wrong implementation and redo steps of Desugaring and Vectorization
- 70% Analyse costs of direct interpretation of P_{nest} (needs more accuracy)
- 0% Analyse and discuss runtime costs of vectorized interpretation of P_{nest}
- 0% Analyse and discuss runtime costs of sequential implementation
- 0% Analyse and discuss runtime costs of vector-parallel implementation
- 0% Analyse and discuss human workload for the implementations
- 0% Show differences between NDP and vector-parallel
- 0% Answer first subquestion
- 0% Answer second subquestion
- ?% *stuff*
- 0% Writte thesis
- 0% Colloquium

5 Time Table

Table 1: Time table

Current Week	CW	monday	thesis work
	17	20.04	reading remaining papers, reading parallel complexity theory
	18	27.04	deciding on an algorithm
	19	4.05	implementing P_{nest}, vectorizing and optimizing P_{nest}
	20	11.05	implementing $P_{seq}$and P_{man}, vectorizing and optimizing P_{nest}
	21	18.05	vectorizing and optimizing, rereading details of implementation, analysis
now	22	25.05	finish transformation, analysis & comparision
	23	1.06	~~*puffer*~~ analysis & comparision
	24	8.06	*puffer*
	25	15.06	Begin to write down, prepare for exams
	26	22.06	Writing..., prepare for exams
	27	29.06	Writing..., prepare for exams
	28	6.07	Prepare Colloquium, Writing..., exams week 1
	29	13.07	Prepare Colloquium, Finalize writing, exams week 2
	30	20.07	Colloquium and Release
	31	27.07	Last week for Colloquium and Release
	32	3.08	Fin `:D`

6 Recommended Literature

I have come over the following papers. The papers are sorted by their relevance. The percentage shows how much of each paper I have throughly read. You are recommended to read the essential papers. Important ones are included in Papers.zip.

Essential

100% Data Parallel Haskell: A Status Report [Chakravarty et al., 2007]

100% Harnessing the Multicores: Nested Data Parallelism in Haskell [Jones, 2008]

100% Data Parallel Haskell - 2010 Video - S.P. Jones [Jones, 2010]

100% Programming Parallel Algorithms [Blelloch, 1996]

100% On the distributed implementation of aggregate data structures by program transformation [Keller and Chakravarty, 1999]

60% Costing Nested Arrays Codes [Leshchinskiy et al., 2002]

Compulsatory

20% Higher Order Flattening [Leshchinskiy et al., 2006]

100% Stream Fusion: From Lists to Streams to Nothing at All [Coutts et al., 2007]

20% Exploiting Vector Instructions with Generalized Stream Fusion [Mainland et al., 2013]

234324% JSKDFLSDF

20% Reevaluating Amdahl's Law [Gustafson, 1988]

20% Work Efficient Higher-order Vectorisation [Lippmeier et al., 2012]

20% Playing by the Rules: Rewriting as a practical optimization technique in GHC [Peyton Jones et al., 2001]

100% An Approach to Fast Arrays in Haskell [Chakravarty and Keller, 2003]

40% Functional Array Fusion [Chakravarty and Keller, 2001]

Details

40% Systematic Efficient Parallelization of Scan and Other List Homomorphisms [Gorlatch, 1996]

30% Simon Peyton Jones - Data Parallel Haskell papers [Jones, 2013]

0% Implementation of a Portable Nested Data-parallel Language [Blelloch et al., 1993]

60% More Types for Nested Data Parallel Programming (PA implementations replPA,indexPA...) [Chakravarty and Keller, 2000]

20% Paritial Vectorization of Haskell programs [Chakravarty et al., 2008]

30% Associated Type with Class [Chakravarty et al., 2005]

Alternate Approaches

10% Composable Memory Transactions [Harris et al., 2005]

30% Algorithm + Strategy = Parallelism [Trinder et al., 1998]

20% Regular, Shape-polymorphic, Parallel Arrays in Haskell [Keller et al., 2010]

20% Parallel and Concurrent Programming in Haskell [Marlow, 2012]

20% A monad for deterministic parallelism [Marlow et al., 2011]

10% Optimising Purely Functional GPU Programs [McDonell et al., 2013]

7 Algorithms

The following sections will explain the conceptiual algorithm and then show the the code for the P_{seq}, $P_{man}$and P_{nest}implementations and a pseudo-interface all of them implement.

7.1 Histogram Balancing

Suppose we have an $w \times h$-8-bit-gray-tone image with low contrast.

Figure 1: An image with low constrast

Our goal is to make details more visible to the viewer. For that, lets take a look at the histogram of the image.

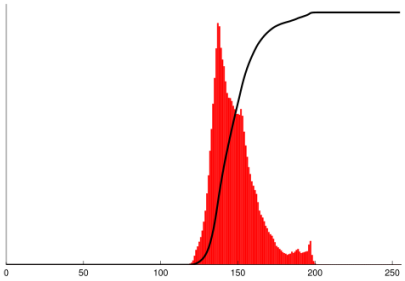

Figure 2: The histogram of the original image with absolute(red) and accumulative(black) count.

A histogram shows the gray tone distribution of the image in interest. The x-axis denotes the gray tone and the y-axis denotes the number of pixels with a gray tone x (red). The black curve denotes the total number of pixels with a gray tone $g \leq x$.

We can now see, why details are difficult to recognize in our original image. The gray tones are tightly packed together and the entire image only uses values in the range $[120, ..., 205]$. Histogram Balancing solves this problem by defining a mapping $f : Graytone \rightarrow Graytone$, such that the accumulating count of the resulting image increases as uniformly as possible from 0 to 255. The histogram 3 shows our goal.

Notice the new curve for the accumulating count(black). We can define this mapping by spreading out the gray tones such that more important gray tones get a larger range to occupy. We interpret a high histogram-value for a gray tone as a high importance. Given this interpretation, we can - visually - build the histogram(1), lay down the bars consecutively - remembering which

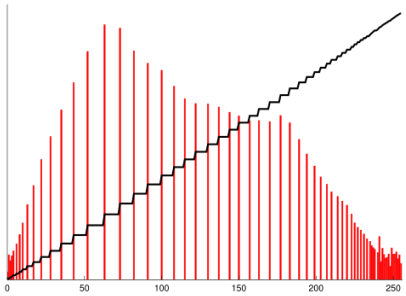

Figure 3: The histogram of the balanced image

bar belongs to which gray tone(2), normalize(3) and scale(4) the bars to $[0, ..., 255]$ - and finally - assign each bar (and therefore its gray tone) a new gray tone using the bars location in the range $[0, ..., 255]$(5). (I should include visual diagrams for this in the thesis.)

To define the transformation $hbalance : Image \to Image$ we need the following definitions:

- $Image := (Width, Height, Width \times Height \to Graytone)$: An image
- $Histogram_a : Graytone \to a$: A histogram assigns a count to each gray tone(for $a = Int$).
- gmax : $Graytone$: The maximum gray tone for the data type in use. (e.g. 255 for 8-bit gray tones)
- $hist : Image \to Histogram_{Int}$ calculates the histogram of an image. (1)
- $accu : Histogram_{Int} \to Histogram_{Int}$ calculates the accumulating histogram from the original histogram. (2)
- $normalize : Int \times Int \times Histogram_{Int} \to Histogram_{Double}$ normalizes the bars (in the range defined by the first two arguments) to a range from 0 to 1. (3) The second argument (called $agmax$) denotes the number of total pixels in the image. It can be calculated with a(gmax) or $a(g)$ where g is highest gray tone in the image. Both are equal, so the implementations take the liberty to use any of these formulas.
- $scale : Graytone \times Histogram_{Double} \to Histogram_{Int}$ scales the normalized values to the maximum gray tone(255 in our case) and rounds down to the nearest integer. (4)
- $apply : Histogram_{Int} \times Image \to Image$ maps each gray tone to its new value as dictated by the first argument. (5)

Given these functions we can define $hbalance : Image \to Image$ as:

$$\begin{aligned} hbalance(img) &:= apply(s, img) \\ s &:= scale(\text{gmax}, n) \\ n &:= normalize(a(0), a(\text{gmax}), a) \\ a &:= accu(h) \\ h &:= hist(img) \end{aligned} \tag{1}$$

Concrete implementations are given in the next sections. Applying the algorithm to our initial image gives 4. Details are more distinguishable in our new image.

Figure 4: The equalized image

7.2 Pseudo-Interface

All implementations have to give concrete definitions for this pseudo-interface. It defines names for the functions and data types, so that each implementation can take the libery to define a different representation of images, histograms etc... The implementations are flexible in that the gray tone type (32-bit Int, 8-bit Int etc...) can be changed at any time. This is why Image takes a type parameter and why we have to let the program know the maximum gray tone gmax. I am calling this a pseudo-interface, because it resembles interfaces from Java and associated type synonyms in Haskell.

```
type Image a -- the type for Images of gray-tone type a
type Many a -- thy type to represent bulks of images

img :: Image Int -- this shall be an example image
images :: Many (Image Int) -- this shall be an example collection of images

hbalance :: Image Int -> Image Int -- single application of histogram balancing
hbalanceBulk :: Many (Image Int) -> Many (Image Int) -- bulk application of histogram balancing
```

The maximum gray tone gmax in the implementations will be set to 255 although a much larger value is possible.

7.3 Implementation of P_{seq}

This section gives the implementation of $P_{seq}$and notes on it. $P_{seq}$is a very direct implementation without strong focus on efficiency and was written within 30 minutes.

```
module ListHistogramBalance (hbalance,hbalanceBulk) where

import qualified Data.Map.Strict as M
import Data.Map.Strict (Map)
import qualified Data.Vector as V

type Image a = V.Vector (V.Vector a)
type Many a  = V.Vector a

type Hist a = Map Int a

-- apply histogram balancing on many images by applying it on each image
hbalanceBulk :: Many (Image Int) -> Many (Image Int)
hbalanceBulk = V.map hbalance

-- histogram balancing
hbalance :: Image Int -> Image Int
hbalance img =
  let h = hist img
      a = accu h
      a0 = snd $ head $ M.toAscList a
      agmax = snd $ head $ M.toDescList a
      n = normalize a0 agmax a
      s = scale gmax n
      img' = apply s img
  in img'

gmax :: Int
gmax = 255

hist :: Image Int -> Hist Int
hist = V.foldr (\i -> M.insertWith (+) i 1) M.empty . concat

accu :: Hist Int -> Hist Int
accu = scanl (+) 0

normalize :: Int -> Int -> Hist Int -> Hist Double
normalize a0' agmax' as =
    let a0 = fromIntegral a0'
        agmax = fromIntegral agmax'
        divisor = agmax - a0
    in M.map (\freq' -> (fromIntegral freq' - a0) / divisor) as

scale :: Int -> Hist Double -> Hist Int
scale gmax = M.map (\d -> floor (d * fromIntegral gmax))

apply :: Hist Int -> Image Int -> Image Int
apply as img = V.map (V.map (as M.!)) img
```

The implementation uses Haskell Vectors [1]. Vectors are immutable pointwise-lazy arrays and exhibit different behaviour than conventional C arrays. Immutable means, that (like every value in Haskell) their value cannot be changed after creation. Pointwise-lazyness basically means, that every value inside the array is not caluclated until its value is actually demanded. We don't need to know more than that. However, this means that vectors are roughly like immutable C-arrays of pointers where each pointer refers to the actual data on the heap. The implementation uses two-dimensional vectors to represent an image.

Vectors can hold data of arbitrary complexity. This includes an image itself. Therefore we can also use a vector to represent a collection of images.

The main algorithm (hbalance) is implemented pretty much like the definition. The representation of an histogram uses a Map data structure [2]. A histogram is defined as a map from finitely many integer values to some fixed type a. The expressions for $a(0)$ and $a(gmax)$ extract the value for the smallest and largest gray tone from the map respectively.

The other functions are now briefly explained:

- hist: The histogram calculation flattens the image to a one-dimensional vector and iterates through all elements creating a map of graytones and their number of occurrences.
- accu: calculates the prefixsum of the gray tones counts in the map. The map is traversed from smaller gray tones to larger ones.
- normalize: fromIntegral is the explicit conversion from Double to Int.
- scale: Scales each value from 0 to 1 to 0 to gmax and rounds down the floating point results.
- apply: maps each pixel in each row by looking up (this is what M.! does) that pixels gray tone in the final equalized histogram.

[1] See vector package on Hackage: `http://hackage.haskell.org/package/vector-0.10.0.1/docs/Data-Vector.html`

[2] See containers package on Hackage: `http://hackage.haskell.org/package/containers-0.5.6.1/docs/Data-Map-Strict.html`

7.4 Implementation of P_{man}

The manually-parallelized program focuses a bit more on parallelizing the task. Given the time constraint of 30 minutes, it can only implement simple optimizations for parallel behaviour. Let's take a look at its code.

```
module MthreadedHistogramBalance (hbalance,hbalanceBulk) where

import qualified Data.Vector.Unboxed as V -- dense arrays
import qualified Data.Vector as VB        -- array of pointers
import Data.Vector.Strategies

type Image a = V.Vector a     -- flat pointerless image representation
type Many a  = VB.Vector a    -- pointer-based representation of the collection

type Hist a    = V.Vector a

-- processor parallel application of histogram balancing over a collection of images
hbalanceBulk :: Int -> Many (Image Int) -> Many (Image Int)
hbalanceBulk cores imgs = (VB.map hbalance imgs) `using` (parVector cores)

-- single sequential histogram balancing
hbalance :: Image Int -> Image Int
hbalance img =
  let h = hist img
      a = accu h
      a0 = V.head a
      agmax = V.last a
      n = normalize a0 agmax a
      s = scale gmax n
      img' = apply s img
  in img'

gmax :: Int
gmax = 255

hist :: Image Int -> Hist Int
hist =
  let init = V.replicate (gmax + 1) 0
      step g v = update v (g,1 + v V.! g)
  in foldr step init . V.toList

accu :: Hist Int -> Hist Int
accu = V.scanl1 (+)

normalize :: Int -> Int -> Hist Int -> Hist Double
normalize a0' agmax' as =
    let a0 = fromIntegral a0'
        agmax = fromIntegral agmax'
        divisor = agmax - a0
    in V.map (\freq' -> (fromIntegral freq' - a0) / divisor) as

scale :: Int -> Hist Double -> Hist Int
scale gmax = V.map (\d -> floor (d * fromIntegral gmax))

apply :: Hist Int -> Image Int -> Image Int
```

```
apply as = V.map (as V.!)
```

In constrast to P_{seq}, the two-dimensional vectors have been flatten into a single dimension. Images are now represented by a single flat unboxed vector. Such unboxed vectors use no pointers, are bulk-strict in their elements and are immutable as well.

To retrieve a specific pixel one needs to calculate the offset using the images width. Fortunately, for histogram balancing, we don't need to retrieve pixels by their indices anyways. (However, other algorithms operating on these images would have to cope with that complication.) The unboxed vectors are stored as a contiguousblock of memory with the actual raw data types as values. They are the closest equvalent of classic C-arrays.

Avoiding pointers and an extra level of nesting on the images makes P_{man}more efficient than the P_{seq}- in cost of a more complicated implementation. Also note that, unlike normal (boxed) vectors, unboxed vectors cannot contain every data type. (Because, if you want to avoid pointers, you need to know the size of the elements in advance to allocate the space for it. This is not possible for complex types like lists, trees or even vectors.) This is the reason, why a colleciton of images is represented by a (boxed) vector (just like in P_{seq}).

However, the efficient representation of images comes at a (possibly high) cost. Since unboxed vectors are bulk-strict and immutable, its not possible to parallelize the construction/calculation of it's elements. That would either require mutability and explicit spawning and synchronization of threads, or a fallback to boxed vectors. The first option greatly increases the complexity of the program and the time to implement it. Though possible in Haskell, I choose against it, to keep my implementation understandable, simple and bug-free. The second option is viable, since boxed vectors allow for parallel evaluation very easily. However, as mentioned, they use pointers and introduce great constant factor slowdown during execution. Therefore hbalance itself is essentially implemented sequentially. (An Prof. Bußmeier: ^Bitte unbedingt diskutieren, inwiefern ich stattdessen eine parallelisierbare aber pointer-basierte implementation nehmen soll. Der Aufwand stiege von 30min auf ca 1h. ^)

Since the collection of images itself is stored in a unboxed vector, it's evaluation can be parallelized. The definition of hbalanceBulk uses [Trinder et al., 1998] to parallelize the work. Operationally, the expression spawns cores-many threads, assigns each thread a chunk of the entire collection of images, makes each thread process it's chuck sequentially, and finally joins the threads. (More presicely, it evaluates the elements to (Weak Head) Normal Form in parallel instead of evaluating them sequentially as usual.)

The remaining functions are implemented very much like in P_{seq}. The only main difference is the representation of the histogram as an unboxed vector instead of a Map. An unboxed vector of length $\text{gmax} + 1$ is used where the element at index i denotes the number of pixels with gray tone i, e.g. a vector $[0, 3, 4]$ represents the histogram $\{0 \mapsto 0, 1 \mapsto 3, 2 \mapsto 4\}$.

7.5 Implementation of P_{nest}

We can now finally discuss the main implementation of my thesis. This implementation uses Nested Data Parallelism [Jones, 2008] and is the most promising one from all three implemetations. It was written in < 30 minutes and was easier to write than P_{man}. in code, [: a :] refers to a parallel array of type a. The various common functions over lists (map,group,sort,etc.) have been replicated for arrays and have a P suffix appended (e.g. mapP).

```
module HistogramBalance (hbalance, hbalanceBulk) where

import qualified Prelude as P
import Data.Array.Parallel -- work of Chakravarty, Leshchinskiy, Jones, Keller and Marlow

type Many a = [: a :]
type Image a = [:[: a :]:]

type Hist a = [: a :]

hbalanceBulk :: Many (Image Int) -> Many (Image Int)
hbalanceBulk = mapP hbalance

hbalance :: Image Int -> Image Int
hbalance img =
    let h = hist img
        a = accu h
        a0 = headP a
        agmax = lastP a
        n = normalize a0 agmax a
        s = scale gmax n
        img' = apply s img
    in img'

gmax :: Int
gmax = 255

hist :: Image Int -> Hist Int
hist =
    sparseToDenseP (gmax+1) 0
    . mapP (\g -> (headP g,lengthP g))
    . groupP
    . sortP
    . concatP

accu :: Hist Int -> Hist Int
accu = scanlP (+) 0

normalize :: Int -> Int -> Hist Int -> Hist Double
normalize a0' agmax' as =
    let a0 = P.fromIntegral a0'
        agmax = P.fromIntegral agmax'
        divisor = agmax D.- a0
    in [: (P.fromIntegral freq' D.- a0) D./ divisor | freq' <- as :]

scale :: Int -> Hist Double -> Hist Int
scale gmax as = [: P.floor (a D.* P.fromIntegral gmax) | a <- as :]
```

```
apply :: Hist Int -> Image Int -> Image Int
apply as img = mapP (mapP (as !:)) img
```

The implementation uses polymorphic type-indexed bulk-strict arrays [:a:] to represent an image and a collection of images. Contrary to it's appearance, parallel arrays of this type are more flexible than vectors and on average comparably efficient. I won't repeat it's features and semantics now and refer to [Jones, 2008].

Except for hist, the entire implementation is very similar to the original definition and to P_{man}. However, in this case the entire implementation of hbalance is also parallel - and even distributed!

The definition of hist uses an alternative way to build the histogram. This is where previous implementation used an iteration over all values to compute the histogram. This implementation instead uses bulk-operations to group together the gray tones and count them in parallel. This implementation, to me personally, looks more elegant than the iterative implementation in $P_{seq}$and P_{man}. To our benefit, it is also more parallelizable than the others.

8 Code Transformation

This section briefly shows the code resulting from applying the transformations described in [Jones, 2008] on P_{nest}. The transformations are not commented yet and not entirely finished. (Stream and communication fusion at the end is still missing.) I needed much time to make these transformations - as they are ususally the compilers work. You can see the full, step-by-step, transformations on my github repository [1].

After the transformations of the individual functions, the entire desugared and vectorized function and my current progress in on optimization is shown. I don't expect the code to be understood yet - therefore I don't give explanations now.

I have also ommitted the lifted variants of the code.

(I some sense, I want to show that I have not been slaking around doing nothing in 3/4 weeks.)

8.1 Individual functions in P_{nest}

8.1.1 hbalance

```
-- vectorized gmax
V[gmax] :: Int
V[gmax] = 225

V[hbalance] :: PA (PA Int) :-> PA (PA Int)
  = Clo {
       env = ()
     , scalar  = V[hbalanceBody]
     , lifted  = (... omitted here ...)
    }

V[hbalanceBody] :: () -> PA (PA Int) -> PA (PA Int)
  = \() img ->
      (\h ->
        (\a ->
          V[apply]
              $: (V[scale]
                    $: gmax
                    $: V[normalize]
                          $: (headPV $: a)
                          $: (lastPV $: a)
                          $: a
                 )
              $: img
         ) (V[accu] $: h)
      ) (V[hist] $: img)
```

8.1.2 hist

```
hist1 :: PA (PA Int)) :-> PA Int
V[hist] $: img
  = sparseToDensePV
      $: (plusIntV $: gmax $: 1)
      $: 0
      $: mapPV
```

[1] In the top-level files at `https://github.com/GollyTicker/Nested-Data-Parallel-Haskell/tree/64abb73dcc50ee39816f8d02a18fbb00b48d7b57`

```
        $: Clo () _ lambdaGL
        $: groupPV
          $: sortPV
            $: concatPV
              $: img
```

8.1.3 accu

```
V[accu] :: PA Int :–> PA Int
V[accu] $: h
  = (\xs –> scanlPV $: plusIntV $: 0 $: xs) $: h
  = scanlPV $: plusIntV $: 0 $: h
```

8.1.4 normalize

```
V[normalize] :: Int :–> Int :–> PA Int :–> PA Double
V[normalize] $: someA0 $: someAgmax $: someAccu
  = (\a0 –>
      (\ divisor –>
        mapPV
          $: Clo {
                env = (a0, division )
              , scalar = (... ignored inside mapP...)
              , lifted =
                \(ATup2 n a0 division ) a –>
                  replPA n divV
                    $:L replPA n minusV
                          $:L (replPA n fromIntegralV $:L a)
                          $:L a0
                    $:L divisor
            }
          $: someAccu
      ) (minusDoubleV $: (fromIntegralV $: someAgmax) $: a0)
    ) (fromIntegralV $: someA0)
```

8.1.5 scale

```
V[scale] :: Int :–> PA Double :–> PA Int
V[scale] $: someInt $: someNormHist
  = mapPV
      $: Clo {
            env = (someInt)
          , lifted = \(ATup1 n gmax) a –> replPA n floorV $:L (replPA n multDoubleV $:L (replPA fromIntegralV n $:L gm
          , scalar = (... ignored inside mapP...)
          }
      $: someNormHist
```

8.2 Vectorized P_{nest}

```
"Result of manual vectorization"

"$  and .  are application/composition of usual functions"
"$: and $:L are scalar and lifted application of vectorized functions"

V[hbalance] $: img :: PA (PA Int)
  = let a = scanlPS plusIntV 0                    -- accu
```

```
          . sparseToDensePS (plusIntS gmax 1) 0 -- hist
          . mapPS
            $ Clo { env = ()
                  , lifted  = \(ATup0 n) g -> (,)L (replPS n headPV $:L g) (replPS n lengthPV $:L g)
                  }
          . groupPS
          . sortPS
          . concatPS
          $ img
in  mapPS                                       -- apply.
    $ Clo {
        lifted  =
          \(ATup1 n as) xs ->
            replPS n mapPV                      -- core of nested data parallelism here!
              $:L AClo { aenv = ATup1 n as      -- apply on every pixel
                       , lifted  = \(ATup1 n as) g -> replPS n indexPV $:L as $:L g
                       }
              $:L xs
       ,env =  mapPS                            -- scale
                 Clo { env = (gmax)
                     , lifted  =
                       \(ATup1 n gmax) a ->
                         replPS n floorV        -- scale each grayvalue
                           $:L (replPS n multDoubleV $:L (replPS n fromIntegralV n $:L gmax) $:L a)
                     }
                 . mapPS                        -- normalize
                     Clo {
                         env = (int2Double (headPS a), minusDoubleS (int2Double (lastPS a)) a0)
                       , lifted  =
                         \(ATup2 n a0  divisior ) a ->
                           replPS n divV
                             $:L replPS n minusV
                                   $:L (replPS n fromIntegralV $:L a)
                                   $:L a0
                             $:L  divisor
                     }
                 $ a
      }
    $ img
```

8.3 Partially optimised P_{nest}

"First step of optimization. Optimizing on parallel arrays"
"After this step comes distributed types and stream/communication fusioning."

"$ and . are application/composition of usual functions"
"$: and $:L are scalar and lifted application of vectorized functions"

```
V[hbalance] $: img :: PA (PA Int)
  = let a = scanlPS plusIntV 0     -- accu
          . sparseToDensePS (plusIntS gmax 1) 0   -- hist end
          . (\g -> (,)L (headPL g) (lengthPL g))  -- ignored argument
          . groupPS
          . sortPS
```

```
        . concatPS                                  -- hist begin
        $ img
    n = length a
    as = replPS (lengthPS img)              -- replicate width
        . floorL                                    -- normalize and scale
          (multDoubleL (int2DoubleL (replPS n gmax)))
        . divL
            (minusL (int2DoubleL a) (   replPS n (int2Double (headPS a))  ))
        . replPS n
        $ minusDoubleS (int2Double (lastPS a)) a0
in (\xs -> -- apply on every pixel -- core of nested data parallelism here!
     unconcatPS xs . indexPL (concatPS . replPL (lengths (getSegd as)) as) . concatPS $ xs
   ) img
```

9 References

[Blelloch, 1996] Blelloch, G. E. (1996). Programming parallel algorithms. *Commun. ACM*, 39(3):85–97.

[Blelloch et al., 1993] Blelloch, G. E., Hardwick, J. C., Chatterjee, S., Sipelstein, J., and Zagha, M. (1993). Implementation of a portable nested data-parallel language. *SIGPLAN Not.*, 28(7):102–111.

[Chakravarty and Keller, 2003] Chakravarty, M. and Keller, G. (2003). An approach to fast arrays in haskell. In Jeuring, J. and Jones, S., editors, *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 27–58. Springer Berlin Heidelberg.

[Chakravarty and Keller, 2000] Chakravarty, M. M. T. and Keller, G. (2000). More types for nested data parallel programming. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 94–105, New York, NY, USA. ACM.

[Chakravarty and Keller, 2001] Chakravarty, M. M. T. and Keller, G. (2001). Functional array fusion. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, volume 36 of *ICFP '01*, pages 205–216, New York, NY, USA. ACM.

[Chakravarty et al., 2005] Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. (2005). Associated types with class. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 1–13, New York, NY, USA. ACM.

[Chakravarty et al., 2008] Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., and Keller, G. (2008). Partial vectorization of haskell programs. In *Standalone Paper*.

[Chakravarty et al., 2007] Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., Keller, G., and Marlow, S. (2007). Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA. ACM.

[Coutts et al., 2007] Coutts, D., Leshchinskiy, R., and Stewart, D. (2007). Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, volume 42 of *ICFP '07*, pages 315–326, New York, NY, USA. ACM.

[Gorlatch, 1996] Gorlatch, S. (1996). Systematic efficient parallelization of scan and other list homomorphisms. In *Proceedings of the Second International Euro-Par Conference on Parallel Processing-Volume II*, Euro-Par '96, pages 401–408, London, UK, UK. Springer-Verlag.

[Gustafson, 1988] Gustafson, J. L. (1988). Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533.

[Harris et al., 2005] Harris, T., Marlow, S., Jones, S. P., and Herlihy, M. (2005). Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 48–60, New York, NY, USA. ACM.

[Jones, 2008] Jones, S. P. (2008). Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, page 138, Berlin, Heidelberg. Springer-Verlag.

[Jones, 2010] Jones, S. P. (2010). Data parallel haskell - 2010 video - S.P. jones. `https://www.youtube.com/watch?v=NWSZ4c9yqW8`.

[Jones, 2013] Jones, S. P. (2013). Simon peyton jones - data parallel haskell papers. `http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/`.

[Keller and Chakravarty, 1999] Keller, G. and Chakravarty, M. (1999). On the distributed implementation of aggregate data structures by program transformation. In Rolim, J., Mueller, F., Zomaya, A., Ercal, F., Olariu, S., Ravindran, B., Gustafsson, J., Takada, H., Olsson, R., Kale, L., Beckman, P., Haines, M., ElGindy, H., Caromel, D., Chaumette, S., Fox, G., Pan, Y., Li, K., Yang, T., Chiola, G., Conte, G., Mancini, L. V., Méry, D., Sanders, B., Bhatt, D., and Prasanna, V., editors, *Parallel and Distributed Processing*, volume 1586 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg.

[Keller et al., 2010] Keller, G., Chakravarty, M. M. T., Leshchinskiy, R., Jones, S. P., and Lippmeier, B. (2010). Regular, shape-polymorphic, parallel arrays in haskell. *SIGPLAN Not.*, 45(9):261–272.

[Leshchinskiy et al., 2002] Leshchinskiy, R., Chakravarty, M. M. T., and Keller, G. (2002). Costing nested array codes. *Parallel Process. Lett.*, 12(02):249–266.

[Leshchinskiy et al., 2006] Leshchinskiy, R., Chakravarty, M. M. T., and Keller, G. (2006). Higher order flattening. In *Proceedings of the 6th International Conference on Computational Science - Volume Part II*, ICCS'06, pages 920–928, Berlin, Heidelberg. Springer-Verlag.

[Lippmeier et al., 2012] Lippmeier, B., Chakravarty, M. M. T., Keller, G., Leshchinskiy, R., and Jones, S. P. (2012). Work efficient higher-order vectorisation. *SIGPLAN Not.*, 47(9):259–270.

[Mainland et al., 2013] Mainland, G., Leshchinskiy, R., and Jones, S. P. (2013). Exploiting vector instructions with generalized stream fusio. *SIGPLAN Not.*, 48(9):37–48.

[Marlow, 2012] Marlow, S. (2012). Parallel and concurrent programming in haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School*, CEFP'11, pages 339–401, Berlin, Heidelberg. Springer-Verlag.

[Marlow et al., 2011] Marlow, S., Newton, R., and Jones, S. P. (2011). A monad for deterministic parallelism. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA. ACM.

[McDonell et al., 2013] McDonell, T. L., Chakravarty, M. M. T., Keller, G., and Lippmeier, B. (2013). Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA. ACM.

[Peyton Jones et al., 2001] Peyton Jones, S., Tolmach, A., and Hoare, T. (2001). Playing by the rules: Rewriting as a practical optimization technique in GHC. In *Proceedings of the 2001 Haskell Workshop*, pages 203–233.

[Trinder et al., 1998] Trinder, P. W., Hammond, K., Loidl, H. W., and Jones, S. L. P. (1998). Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60.