Slide 1

An Approach to Fast Arrays in Haskell

Advanced Functional Programming (04s1)

Manuel M. T. Chakravarty
University of New South Wales

① A method for implementing arrays
② Code vectorisation
③ Generic programming of arrays

Slide 2

Why Not Haskell 98 Arrays?

The Dot Product:

$$\overline{v} \cdot \overline{w} = \sum_{i=0}^{n-1} v_i w_i$$

In Haskell 98:

$$\textbf{type } Vector = Array\ Int\ Float$$

$$\begin{aligned} (\cdot) \quad &:: Vector \to Vector \to Float \\ v \cdot w &= sum\ [v!i * w!i \mid i \leftarrow indices\ v] \end{aligned}$$

In C:

```
float dotp (float *v1, float *v2,
            int n) {
  int   i;
  float sum = 0;
  for (i = 0; i < n; i++)
    sum += v1[i] * v2[i];
  return sum; }
```

C is approximately 150 times faster!

(1e5 elements; GHC 5.04 with -O2; 1.2GHz PIIIM; x86/Linux)

Slide 3

What is most suspect about $sum\ [v!i * w!i \mid i \leftarrow indices\ v]$?

- ➜ Most operations are list operations
- ➜ It creates a superfluous intermediate structure

How can we improve this? Explicit fused loop:

$$\begin{aligned} &v \cdot w = loop\ 0\ 0 \\ &\quad \textbf{where} \\ &\qquad loop\ i\ acc \mid i > n \quad = acc \\ &\qquad\qquad\qquad \mid otherwise = loop\ (i+1)\ (v!i * w!i + acc) \\ &\qquad n \qquad\qquad\qquad\quad = snd\ (bounds\ v) \end{aligned}$$

This is less nice, but also only 3 times slower than the C code.

- ➜ The compiler should convert comprehension-based into loop-based code.
- ➜ List operations are not the only problem.

Slide 4

Matrix Vector Multiplication $\overline{w} = A\overline{v}$:

$$w_i = \sum_{j=0}^{m-1} A_{ij} v_j, 0 \leq i \leq n-1$$

Using the convenient, but slow list-based interface:

$$\textbf{type } Matrix = Array\ (Int, Int)\ Float$$

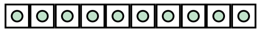
$$\begin{aligned} &mvm \quad :: Matrix \to Vector \to Vector \\ &mvm\ a\ v = listArray\ (0,\ n')\ [sum\ [a!(i,j) * v!j \mid j \leftarrow [0..m']] \mid i \leftarrow [0..n']] \\ &\quad \textbf{where} \\ &\qquad (n',\ m') = snd\ (bounds\ a) \end{aligned}$$

- ➜ This is a factor of 100 slower than the C version
- ➜ Using an explicit loop for the inner comprehension buys us only a factor of two

Slide 5

There are two problems:

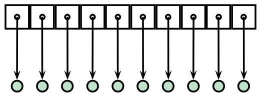
- ➜ We need to produce a new array: $listArray$ slows things down
- ➜ Parametric arrays: lazy, boxed arrays are slow



Slide 6

There are two problems:

- ➜ We need to produce a new array: $listArray$ slows things down
- ➜ Parametric arrays: lazy, boxed arrays are slow



How can we do better?

- ➜ GHC provides unboxed arrays (but the elements are restricted to be of basic type)
- ➜ We can even use mutable unboxed arrays

Slide 7

The Beauty...

$$\begin{aligned}
&mvm\ a\ v = listArray\ (0,\ n')\ [sum\ [a!(i,j) * v!j \mid j \leftarrow [0..m']] \mid i \leftarrow [0..n']] \\
&\quad \mathbf{where}\ (n',\ m') = snd\ (bounds\ a)
\end{aligned}$$

and the Beast:

$$\begin{aligned}
&mvm\ a\ v = runST\ (do \\
&\qquad\qquad ma \leftarrow newArray_\ (0,\ n) \\
&\qquad\qquad outerLoop\ ma\ 0 \\
&\qquad\qquad unsafeFreeze\ ma) \\
&\quad \mathbf{where} \\
&\qquad outerLoop :: STUArray\ s\ Int\ Float \rightarrow Int \rightarrow ST\ s\ () \\
&\qquad outerLoop\ ma\ i \mid i > n \quad = return\ () \\
&\qquad\qquad\qquad\quad \mid otherwise = \mathbf{do}\ writeArray\ ma\ i\ (loop\ i\ 0\ 0) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad outerLoop\ ma\ (i+1) \\
&\qquad loop\ i\ j\ acc \quad \mid j > m \quad = acc \\
&\qquad\qquad\qquad\quad \mid otherwise = loop\ i\ (j+1)\ (acc + a!(i,j) * v!j)
\end{aligned}$$

Such code should only be in the runtime system.

Slide 8

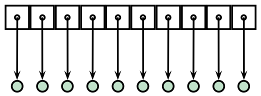
Sparse matrices are worse:

- ➜ Compressed sparse row format

$$\begin{aligned}
&\mathbf{type}\ SparseRow \quad\ = Array\ Int\ (Int,\ Float) \\
&\mathbf{type}\ SparseMatrix = Array\ Int\ SparseRow
\end{aligned}$$

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \begin{aligned}
listArray(0,4)[&listArray(0,0) \quad [(0,\ 5)], \\
&listArray(0,0) \quad [(3,\ 7)], \\
&listArray(0,1) \quad [(0,\ 3),(1,\ 4)], \\
&listArray(0,-1)\ []]
\end{aligned}$$

- ➜ Unboxed arrays? Array elements are not basic!
- ➜ Not too difficult to fix for pairs, but how about the subarrays?



Slide 9

Sparse matrices are worse:

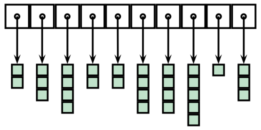
➜ Compressed sparse row format

$$\begin{aligned}&\mathbf{type}\ \mathit{SparseRow} &&= \mathit{Array}\ \mathit{Int}\ (\mathit{Int},\ \mathit{Float})\\ &\mathbf{type}\ \mathit{SparseMatrix} &&= \mathit{Array}\ \mathit{Int}\ \mathit{SparseRow}\end{aligned}$$

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \Longrightarrow \begin{array}{ll} \mathit{listArray}(0,4)[\mathit{listArray}(0,0) & [(0,\ \boxed{5}\ )], \\ \qquad\qquad\qquad\ \mathit{listArray}(0,0) & [(3,\ \boxed{7}\ )], \\ \qquad\qquad\qquad\ \mathit{listArray}(0,1) & [(0,\ \boxed{3}\ ), (1,\ \boxed{4}\ )], \\ \qquad\qquad\qquad\ \mathit{listArray}(0,-1) & []] \end{array}$$

➜ Unboxed arrays? Array elements are not basic!

➜ Not too difficult to fix for pairs, but how about the subarrays?



Slide 10

Parallel Arrays

Requirements in summary:

➜ Array interface avoiding lists

➜ Unboxed arrays, even when the elements are structured

➜ Array construction based on mutable arrays

➜ Present a purely functional, combinator-based interface

Approach based on three program transformations:

① Code vectorisation: functions lifted into vector space
(Currently, done manually; we discuss the basic idea)

② Structure flattening: all data put into unboxed arrays
(Library support; we discuss the implementation)

③ Array fusion: amalgamate consecutive loops
(Library support; we discuss the basic idea)

Implementation:

➜ All three transformations can be automated

Slide 11

Code Vectorisation

Introduction by way of an example; details in the notes.
(Blelloch & Sabot 1990)

The essential idea:

➜ Replace array comprehensions by vectorised code:

$$\begin{aligned}&[\!:\mathit{bexp} \mid x \leftarrow \mathit{gexp}:\!] \Rightarrow \mathbf{let}\ x = \mathit{gexp}\ \mathbf{in}\ \mathcal{L}[\![\mathit{bexp}]\!]^{[x]}\\ &\quad\mathbf{where}\\ &\qquad \mathcal{L} :: (\mathit{Exp} :: \tau) \rightarrow [\mathit{Var}] \rightarrow (\mathit{Exp} :: [\!:\tau:\!])\end{aligned}$$

(Generally, we have to eliminate occurrences of mapP.)

➜ For functions used in an array comprehension:

$$\begin{array}{lll} f & :: \alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow \beta & \text{-- original definition} \\ f\ x_1\ \ldots\ x_n & = \mathit{exp} & \\ f^{\uparrow} & = [\!:\alpha_1:\!] \rightarrow \cdots \rightarrow [\!:\alpha_n:\!] \rightarrow [\!:\beta:\!] & \text{-- vectorised version} \\ f^{\uparrow}\ x_1\ \ldots\ x_n & = \mathcal{L}[\![\mathit{exp}]\!]^{[x_1,\ldots,x_n]} & \end{array}$$

Slide 12

Vectorising sparse-matrix vector multiplication:

$$\begin{aligned}&\mathbf{type}\ \mathit{SparseRow} &&= [\!:(\mathit{Int},\ \mathit{Float}):\!]\\ &\mathbf{type}\ \mathit{SparseMatrix} &&= [\!:\mathit{SparseRow}:\!]\end{aligned}$$

$$\begin{aligned}&\mathit{smvm} :: \mathit{SparseMatrix} \rightarrow [\!:\mathit{Float}:\!] \rightarrow [\!:\mathit{Float}:\!]\\ &\mathit{smvm}\ \mathit{sm}\ \mathit{vec} =\\ &\quad [\!:\mathit{sumP}\ [\!:(\mathit{snd}\ \mathit{cx}) * (\mathit{vec}\ !\!:\ (\mathit{fst}\ \mathit{cx})) \mid \mathit{cx} \leftarrow \mathit{row}:\!] \mid \mathit{row} \leftarrow \mathit{sm}:\!]\end{aligned}$$

Slide 13

Vectorising the inner array comprehension:

$$[: (snd\ cx) * (vec\ !: (fst\ cx)) \mid cx \leftarrow row :]$$

$=$ {Replacing array comprehension & substituting row for cx}

$$\mathcal{L}[\![((snd\ row) * (vec\ !: (fst\ row)))]\!]^{[row]}$$

$=$ {Lift function application}

$$(\mathcal{L}[\![snd\ row]\!]^{[row]}) *^{\uparrow} (\mathcal{L}[\![vec\ !: (fst\ row)]\!]^{[row]})$$

$=$ {Lift function application and variables}

$$(snd^{\uparrow}\ row) *^{\uparrow} (\mathcal{L}[\![vec]\!]^{[row]})\ !:^{\uparrow}\ (\mathcal{L}[\![fst\ row]\!]^{[row]})$$

$=$ {Lift function application, variables, and free variables}

$$(snd^{\uparrow}\ row) *^{\uparrow} ((replicateP\ (lengthP\ row)\ vec)\ !:^{\uparrow}\ (fst^{\uparrow}\ row))$$

where

$$(*^{\uparrow}) = zipWithP\ (*)$$

Optimising lifted indexing:

➜ We have

$$(replicateP\ (lengthP\ inds)\ xs)\ !:^{\uparrow}\ inds = bpermuteP\ xs\ inds$$

(Please read the details on segmented arrays in the notes.)

Slide 14

Where Are We?

Approach based on three program transformations:

① Code vectorisation: functions lifted into vector space

② Structure flattening: all data put into unboxed arrays

③ Array fusion: amalgamate consecutive loops

Slide 15

Structure Flattening

How can we avoid boxing of structured array elements?

➜ We separate structure from data

The sparse matrix representation revisited:

$$\begin{aligned} &\textbf{type}\ SparseRow &&= [:(Int,\ Float):] \\ &\textbf{type}\ SparseMatrix &&= [:SparseRow:] \end{aligned}$$

➜ Tuple hoisting:
(array of pairs ⇒ pair of arrays)

$$\textbf{type}\ FlatSparseRow = ([:Int:],\ [:Float:])$$

➜ Segment descriptors:
(array of arrays ⇒ segment descriptor & flat array)

$$\begin{aligned} &\textbf{type}\ FlatSparseMatrix &&= (Segd,\ FlatSparseRow) \\ &\textbf{type}\ Segd &&= [:Int:] \end{aligned}$$

Slide 16

An example:

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\begin{aligned} [:\ &[:(0,\ \boxed{5.0}\ )], \\ &[(3,\ \boxed{7.0}\ )], \\ &[(0,\ \boxed{3.0}\ ), (1,\ \boxed{4.0}\ )], \\ &[]] \end{aligned}$$

is represented by the flattened structure

$$\begin{array}{ll} ([:1, 1, 2, 0:], & \text{-- segment descriptor} \\ \ ([:0\ \ \ ,3\ \ \ ,0\ \ \ ,1\ \ \ :], & \text{-- value position in each subarray} \\ \ \ [:\ \boxed{5.0}\ ,\ \boxed{7.0}\ ,\ \boxed{3.0}\ ,\ \boxed{4.0}\ :])) & \text{-- values of non-zero elements} \end{array}$$

Slide 17

Representation Transformation

A conceptual problem:

- ➜ With structure flattening, the type $[:\alpha:]$ is no longer parametric
- ➜ For example, considering $\mathit{replicateP} :: \mathit{Int} \rightarrow \alpha \rightarrow [:\alpha:]$,

$$\mathit{replicateP} :: \mathit{Int} \rightarrow (\mathit{Int}, \mathit{Double}) \rightarrow ([:\mathit{Int}:], [:\mathit{Double}:])$$
$$\mathit{replicateP} :: \mathit{Int} \rightarrow [:\mathit{Int}:] \rightarrow (\mathit{Segd}, [:\mathit{Int}:])$$

We are used to type constructors as free functions over types

Defined functions over types via intensional type analysis:

$$\begin{array}{l} \textbf{type } [:\alpha:] \ = \ \textbf{Typecase } \alpha \textbf{ of} \\ \qquad (\beta, \gamma) \rightarrow ([:\beta:], [:\gamma:]) \\ \qquad [:\beta:] \quad \rightarrow (\mathit{Segd}, [:\beta:]) \end{array}$$

(Harper & Morrisett 1995)

Slide 18

Generic programming (a la (Hinze 2000)):

- ➜ Type definitions including a **Typecase** are type-indexed or polytypic
- ➜ The same holds for function definitions including a **typecase**
- ➜ Intensional type analysis versus generic programming:
  - Intensional type analysis: types have a runtime representation, which is scrutinised during program execution
  - Generic programming: compile-time transformation eliminates polytypic functions and types

  Two extreme points in a continuum

Polytypic definitions:

- ➜ Instead of an explicit **typecase**, more concise equation-based notation
- ➜ Cases for unit, products, sums, basic types, and arrays

Slide 19

Polytypic definition of parallel arrays:

$$\begin{array}{lll} \textbf{type } [:\mathit{Unit}:] & = & \mathit{Int} \\ \quad [:\rho:] \ \mid \ \rho \in \{\mathit{Int}, \mathit{Float}, \ldots\} & = & \mathit{UArr}\ \rho \\ \quad [:\tau_1 :*: \tau_2:] & = & [:\tau_1:] :*: [:\tau_2:] \\ \quad [:\tau_1 :+: \tau_2:] & = & \mathit{Sel} :*: [:\tau_1:] :*: [:\tau_2:] \\ \quad [:[:\tau:]:] & = & \mathit{Segd} :*: [:\tau:] \end{array}$$

$$\begin{array}{lll} \textbf{type } \mathit{Sel} & = [:\mathit{Bool}:] & \text{-- selector} \\ \textbf{type } \mathit{Segd} & = [:\mathit{Int}:] & \text{-- segment desc} \end{array}$$

(Unit, $(:*:)$, $(:+:)$ are from GHC's $\mathit{Data.Generics}$)

Slide 20

Implementing the Representation Transformation

Possible approaches:

- ➜ Runtime type representation as in the TILT compiler: new compiler or rewrite GHC
- ➜ Compile-time elimination as in Generic Haskell: pre-processor or extend GHC
- ➜ Existential types & type classes a la (Cheney & Hinze 2002): disables some of GHC's optimisations
- ➜ Derivable type classes as implemented in GHC: types must have kind $\star$ and no multi-parameter type classes
- ➜ Multi-parameter type classes a la (Hinze, Jeuring & Löh): ~~limited genericity requires explicit marshalling~~ (Chakravarty & Keller)
- ➜ Associated Types extension of Haskell type classes to allow nested data types

Slide 21

The basic idea:

- To implement a polytypic type TI of kind κ indexed by τ

  $TI\langle\tau :: \star\rangle \; :: \; \kappa$

  introduce a type class with associated type TI

  class $TIElem\ \tau$ **where**
  data $TI\ \tau$ -- $:: \kappa$

- The associated type represents types $\tau :: \star$ as types $\rho :: \kappa$ (type-dependent representation transformation)
- The mapping is populated by instance declarations
- For each polytypic type equation

  $TI\langle t\rangle \; = \; s$

  define a class instance

  instance $TIElem\ t$ **where**
  data $TI\ t\ =\ s$
  -- Type-indexed functions over TI as methods of $TIElem$

Slide 22

Parallel arrays as associated types (first approximation):

- We implement $[:e:]$ as

  class $PAElem\ e$ **where**
  data $PArray\ e$

  where $PArray\ e$ is the representation type for elements e
- Arrays of $Unit$ elements, $[:Unit:]\ =\ Int$:

  instance $PAElem\ Unit$ **where**
  data $PArray\ Unit\ =\ PAUnit\ Int$ -- length of unit array
- Arrays of pairs, $[:\tau_1 :*: \tau_2:]\ =\ [:\tau_1:] :*: [:\tau_2:]$:

  instance $(PAElem\ a,\ PAElem\ b)\ \Rightarrow$
  $PAElem\ (a :*: b)$ **where**
  data $PArray\ (a :*: b)\ =\ PAProd\ (PArray\ a)\ (PArray\ b)$
- Arrays of basic types; e.g., $[:Int:]\ =\ UArr\ Int$:

  instance $PAElem\ Int$ **where**
  data $PArray\ Int\ =\ PAInt\ (UArr\ Int)$

Unfortunately, this is not the whole story.

Slide 23

IMMUTABLE VERSUS MUTABLE ARRAYS

- Array constructions needs to be based on mutable arrays
- The library interface should be purely functional

➨ Internally use mutable arrays to construct immutable arrays (Peyton Jones & Launchbury 1995)

Typical layout:

① Allocate a mutable array of sufficient size.
② Populate the array by way of a loop in the ST monad.
③ Coerce the fully defined mutable array into an immutable array.

Slide 24

An example:

$replicateU\ \ :: UAE\ e\ \Rightarrow\ Int\ \rightarrow\ e\ \rightarrow\ UArr\ e$
$replicateU\ n\ e\ =$
  $runST\ (do$
    $ma\ \leftarrow\ newMU\ n$
    $fill0\ ma$
    $unsafeFreezeMU\ ma\ n)$
  where
    $fill0\ ma\ =\ fill\ 0$
    where
      $fill\ off\ |\ off\ ==\ n\ =\ return\ ()$
      $\qquad\quad |\ otherwise\ =\ do$
      $\qquad\qquad writeMU\ ma\ off\ e$
      $\qquad\qquad fill\ (off\ +\ 1)$

Interface to unboxed arrays is in the lecture notes.

Slide 25

Where Are We?

Approach based on three program transformations:

① Code vectorisation: functions lifted into vector space

② Structure flattening: all data is stored in unboxed arrays

- ➜ Basic implementation of the representation transformation
- ➜ Generalised representation transformation

③ Array fusion: amalgamate consecutive loops

Slide 26

A Generalised Type Mapping

We have immutable and mutable unboxed arrays:

$$\begin{array}{ll} \textbf{data}\ UAE\ e \Rightarrow UArr\ \ e & \text{-- immutable} \\ \textbf{data}\ UAE\ e \Rightarrow MUArr\ s\ e & \text{-- mutable in } ST\ s\ a \end{array}$$

- ➜ Do we need $PArray$ and $MPArray$?
- ➜ If so, how would they interact?

Better, abstract over the type of base arrays.

Slide 27

$$\begin{array}{l} \textbf{class}\ PAElem\ e\ \textbf{where} \\ \quad \textbf{data}\ PArray\ e\ barr \end{array}$$

Representation constructor:

- ➜ The representation type $PArray\ e$ turns into a representation constructor
- ➜ It gets an additional argument: the type constructor of base arrays
- ➜ Hence, we have

$$PArray :: \star \to (\star \to \star) \to \star$$

For example:

$$\begin{array}{ll} PArray\ Int\ UArr & \text{-- immutable array of integers} \\ PArray\ (Int,\ Float)\ (MUArr\ s) & \text{-- mutable array of pairs} \end{array}$$

- ➜ $PArray$ encodes the structure of an array representation

Slide 28

We are only interested in two types of base arrays:

$$\begin{array}{l} \textbf{type}\ PArr\ \ \ e = PArray\ e\ UArr \\ \textbf{type}\ MPArr\ s\ e = PArray\ e\ (MUArr\ s) \end{array}$$

Polytypic functions on arrays:

$$\begin{array}{llllll} \multicolumn{6}{l}{\textbf{class}\ PAElem\ e\ \textbf{where}} \\ \multicolumn{6}{l}{\quad \textbf{data}\ PArray\ e\ arr} \\ \quad lengthP & :: PArr\ e & & & \to & Int \\ \quad indexP & :: PArr\ e & \to Int & & \to & e \\ \quad sliceP & :: PArr\ e & \to Int & \to Int & \to & PArr\ e \\ \quad newMP & :: Int & & & \to & ST\ s\ (MPArr\ s\ e) \\ \quad writeMP & :: MPArr\ s\ e & \to Int & \to e & \to & ST\ s\ () \\ \quad unsafeFreezeMP & :: MPArr\ s\ e & \to Int & & \to & ST\ s\ (PArr\ e) \end{array}$$

Slide 29

Instances for the representation constructor:

-- Unit arrays: $[:\mathit{Unit}:] = \mathit{Int}$

$\mathbf{instance}\ \mathit{PAElem}\ \mathit{Unit}\ \mathbf{where}$

$\quad \mathbf{data}\ \mathit{PAArray}\ \mathit{Unit}\ (\mathit{barr} :: \star \to \star) = \mathit{PAUnit}\ \mathit{Int}$

-- Arrays of pairs: $[:\tau_1 :*: \tau_2:] = [:\tau_1:] :*: [:\tau_2:]$

$\mathbf{instance}\ (\mathit{PAElem}\ a,\ \mathit{PAElem}\ b) \Rightarrow$

$\quad \mathit{PAElem}\ (a :*: b)\ \mathbf{where}$

$\quad \mathbf{data}\ \mathit{PArray}\ (a :*: b)\ (\mathit{ua} :: \star \to \star) = \mathit{PAProd}\ (\mathit{PArray}\ a\ \mathit{ua})\ (\mathit{PArray}\ b\ \mathit{ua})$

-- Basic arrays, e.g.: $[:\mathit{Int}:] = \mathit{UArr}\ \mathit{Int}$

$\mathbf{instance}\ \mathit{PAElem}\ \mathit{Int}\ \mathbf{where}$

$\quad \mathbf{data}\ \mathit{PArray}\ \mathit{Int}\ (\mathit{ua} :: \star \to \star) = \mathit{PAPrim}\ (\mathit{ua}\ \mathit{Int})$

-- Nested arrays: $[:[:\tau:]:] = \mathit{Segd} :*: [:\tau:]$

$\mathbf{instance}\ \mathit{PAElem}\ e \Rightarrow \mathit{PAElem}\ (\mathit{PArr}\ e)\ \mathbf{where}$

$\quad \mathbf{data}\ \mathit{PArray}\ (\mathit{PArr}\ e)\ (\mathit{ua} :: \star \to \star) = \mathit{PAPArr}\ (\mathit{Segd}\ \mathit{ua})\ (\mathit{PArray}\ e\ \mathit{ua})$

Slide 30

ARRAY FUSION

Implications of collective array operations:

- ➜ Sequences of combinators imply superfluous intermediate arrays, e.g.,

  $\mathit{sumP}\ (\mathit{mapP}\ f\ \mathit{arr})$

- ➜ Vectorisation aggravates the situation
- ➜ Particularly bad for arrays that are larger than the hardware cache

➨ we would like to fuse consecutive array traversals

Slide 31

Fusion in the case of lists:

- ➜ The most successful technique is $\mathit{foldr}/\mathit{build}$ shortcut deforestation
- ➜ All fusible list operations are expressed in terms of foldr and build
- ➜ Small number of fusion rules

What's the foldr and build for arrays?

- ➜ Construction:

  $\mathit{replicateP} :: \mathit{PAElem}\ e \Rightarrow \mathit{Int} \to e \to \mathit{PArr}\ e$

- ➜ Processing:

$$
\begin{array}{lll}
\mathit{loopP} :: (\mathit{PAElem}\ e,\ \mathit{PAElem}\ e') & & \\
\quad \Rightarrow (\mathit{acc} \to e \to (\mathit{acc},\ \mathit{Maybe}\ e')) & & \text{-- mapping \& folding elements} \\
\quad \to \mathit{acc} & & \text{-- initial accumulator value} \\
\quad \to \mathit{PArr}\ e & & \text{-- consumed array} \\
\quad \to (\mathit{PArr}\ e',\ \mathit{acc}) & &
\end{array}
$$

- ➜ Similar operations for segmented arrays

Slide 32

Examples of using loopP:

$$
\begin{array}{ll}
\mathit{mapP} & :: (\mathit{PAElem}\ e\ \mathit{arr},\ \mathit{PAElem}\ e') \\
 & \Rightarrow (e \to e') \to \mathit{PArr}\ e \to \mathit{PArr}\ e' \\
\mathit{mapP}\ f & = \mathit{fst} \circ \mathit{loopP}\ (\lambda\_\ e \to (\mathit{Unit},\ \mathit{Just}\ \$\ f\ e))\ \mathit{Unit} \\
 & \\
\mathit{foldlP} & :: \mathit{PAElem}\ e \\
 & \Rightarrow (b \to a \to b) \to b \to \mathit{PArr}\ a \to b \\
\mathit{foldlP}\ f\ z & = \mathit{snd} \circ \mathit{loopP}\ (\lambda a\ e \to (f\ e\ a,\ \mathit{Nothing}))\ z \\
 & \\
\mathit{scanlP} & :: \mathit{PAElem}\ e \\
 & \Rightarrow (b \to a \to b) \to b \to \mathit{PArr}\ e \to \mathit{PArr}\ e \\
\mathit{scanlP}\ f\ z & = \mathit{fst} \circ \mathit{loopP}\ (\lambda a\ e \to (f\ e\ a,\ \mathit{Just}\ a))\ z \\
 & \\
\mathit{enumFromToP} & :: (\mathit{Enum}\ e,\ \mathit{PAElem}\ e) \\
 & \Rightarrow e \to e \to \mathit{PArr}\ e \\
\mathit{enumFromToP}\ \mathit{from}\ \mathit{to} = & \\
\multicolumn{2}{l}{\quad \mathit{fst} \circ \mathit{loopP}\ (\lambda a\ \_ \to (\mathit{succ}\ a,\ \mathit{Just}\ a))\ \mathit{from} \circ \mathit{replicateP}\ (\mathit{to} - \mathit{from} + 1)\ \$\ \mathit{Unit}}
\end{array}
$$

Slide 33

Fusion by way of GHC's rewrite rules:

$$
\begin{array}{l}
\langle\ \text{loopP/replicateP fusion}\rangle\ \forall\ mf\ start\ n\ e\ .\\
\quad loopP\ mf\ start\ (replicateP\ n\ e)\ \mapsto\\
\qquad loopP\ (\lambda acc\ \_ \rightarrow\ mf\ acc\ e)\ start\ (replicateP\ n\ Unit)
\end{array}
$$

$$
\begin{array}{l}
\langle\ \text{loopP/loopP fusion}\rangle\ \forall\ mf_1\ start_1\ mf_2\ start_2\ arr.\\
\quad loopP\ mf_2\ start_2\ (loopArr\ (loopP\ mf_1\ start_1\ arr))\mapsto\\
\quad \mathbf{let}\\
\qquad mf\ (acc_1,\ acc_2)\ e\ =\\
\qquad \mathbf{case}\ mf_1\ e\ acc_1\ \mathbf{of}\\
\qquad\quad (acc_1', Nothing) \rightarrow ((acc_1',\ acc_2),\ Nothing)\\
\qquad\quad (acc_1', Just\ e') \ \ \rightarrow \mathbf{case}\ mf_2\ e'\ acc_2\ \mathbf{of}\\
\qquad\qquad\qquad\qquad\qquad\qquad (acc_2',\ res) \rightarrow ((acc_1',\ acc_2'),\ res)\\
\quad \mathbf{in}\\
\quad loopSndAcc\ (loopP\ mf\ (start_1,\ start_2)\ arr)
\end{array}
$$

Slide 34

Conclusions

Approach:

- ➜ Code vectorisation
- ➜ Structure flattening
- ➜ Array fusion

Preliminary results (using MPTC with FDs):

- ➜ Code is much faster than with Haskell standard arrays
- ➜ For simple examples, within factor of 3–4 of C