

Draft Extended Abstract
FlatPack: An Affordable Range of Large
Multi-Dimensional Arrays

Malcolm Wallace
University of York, UK

October 22, 2009

Abstract

There are many application areas that need to deal with vast quantities of raw number data. This paper sets out a new approach to dealing with such large arrays. Computations over arrays can be split into two stages: first, a calculation of the order in which memory blocks should be traversed, then second, the actual traversal and computation over elements. A domain-specific language for array manipulations is constructed in such a way that a traversal order can be analysed and optimised before any numeric operations take place.

1 Introduction

There are many application areas that need to deal with vast quantities of raw number data. Financial markets may generate thousands of pricings every minute of the trading day. A raw DV-encoded video stream, produced by a consumer-level camcorder, contains about 1Gb of data per five minutes of content. An astrophysics simulation used as the basis of the 2008 IEEE Contest on Visualisation Design contains over 200Gb of binary-encoded data on gas temperature, density, and ion concentrations. Such numeric data is usually arranged in a rectangular grid of many dimensions. For instance, the above physical simulation encodes 10 scalar quantities and 3 vector components at every point in three spatial dimensions $600 \times 250 \times 250$, along a time axis with 200 steps.

Some of the common operations needed over such large quantities of data are: to take slices or subranges across some selection of dimensions; to down-sample within a dimension; to transpose, chop, join, and map over certain dimensions. These kinds of operation are *structural*, and largely independent of the actual data values. By contrast there are also numeric operations that combine data values in arbitrary ways: in general these numeric operations are applied repeatedly throughout some sub-range of the dimensions.

Clearly, it is essential for any application that processes such array-structured data to keep the data closely-packed, and to avoid wherever possible to move it into a different data structure with heavier pointer overheads, such as the classic boxed list. So the challenge, especially for a purely-functional approach, is twofold: can the data be processed without copying the structure around in memory? and, when dealing only with sections of the data, can we avoid creating intermediate structures of similar size?

This paper sets out a new approach to dealing with large arrays:

- Choose an array representation that separates the dimensional structure from the data contained within it. The data is flat-packed in memory; the dimensional structure is a conceptual (or virtual) layer on top.
- Distinguish structural operations over dimensions (e.g. slicing), from any numeric operations over data values.
- Structural operations calculate a traversal order over the data, whilst the numeric operations are held until they can be embedded into the traversal.

Array-processing code becomes a two-step abstraction. Previously, one might write a composition chain of functions that directly manipulate the array value. Now, it is understood that the same composed function merely calculates *how* to manipulate the array, but does not perform the action itself. One further step, to invoke the action, is required.

A second contribution of this new approach is to develop a small domain-specific language (DSL) for array manipulations. The language restricts the operations that can be performed to those that can be built from a given set of primitives and combinators. But the benefit is that the DSL is highly analysable: straightforward rule-based transformations can enable significant optimisation of array traversals, similar to deforestation or fusion transformations in a compiler.

Indeed, the DSL approach offers up a choice of implementation strategies for array-processing code:

- The DSL can simply be interpreted at runtime.
- The DSL can be optimised before interpretation at runtime. For larger arrays, the time spent in optimisation might be recovered by savings in the actual traversal.
- The DSL can be optimised and compiled (to Haskell, C, or even assembler) so that runtime is minimised for multiple invocations.

A further benefit of the DSL approach is that the DSL compiler can easily be extended with multiple backends: for instance, to use a simple and readable, but inefficient, data model for validation of the correctness of transformations, together with a complex and efficient data model for performance.

2 Flat-packing arrays

In order to deal efficiently with large quantities of data, it is essential to attempt (a) to keep the data closely packed, (b) to avoid moving or copying it, and (c) to traverse it in a cache-friendly order. In addition, it is useful to avoid, wherever possible, the rapid allocate/use/discard cycle of temporary storage typically implemented in functional languages to deal with constructing and pattern-matching intermediate data structures, that do not form a part of the final result.

We conceptualise multi-dimensional datasets as essentially flat blocks of packed values. The dimensions are represented implicitly by the order of the packing. Thus, a two-dimensional array (n, m) of numbers, with the x dimension varying fastest, has a sequence of n values at y -coordinate 0, followed by n values at y -coordinate 1, and so on, up to y -coordinate $m - 1$.

A tuple is very like a short array, and therefore we treat tuples and arrays interchangeably.

For instance, a pair of arrays is logically isomorphic to an array of pairs. There is no need to change the internal representation or packing of the data itself, depending on which type we want to operate over. It is sufficient simply to change the manner in which the data is accessed.

Array computations in general must traverse memory blocks in some order, performing smaller computations on the values held within the block. We observe that changing the order of traversal is equivalent to moving the elements of data around, but the former is usually much cheaper.

The central idea then, of this work, is that a computation over arrays can be split into two stages: first, a calculation of the order in which memory blocks should be traversed, then second, the actual traversal and computation over elements. The first stage can be computed statically, employing analysis techniques to fuse, prune, and optimise traversals. The result of the first stage thus makes the second stage more efficient. Even if both stages are calculated at runtime, one can hope that the performance improvement introduced by optimisation, will outweigh the initial cost of calculating the improvement. For larger datasets, this should become increasingly worthwhile, especially if the improvement affects computational complexity, and not merely the constant factors.

3 DSL for array computation

Figure ?? is the abstract syntax of a DSL for manipulating flat arrays of arbitrary dimension (source code in *Data.Array.FlatPack.DSL*). The language is expressed as a Haskell datatype, *ArrayComp a*, so that we can easily manipulate the *program* (for instance, analyse it, optimise it) before the program gets to manipulate the *data* it is supposed to operate on.

The type parameter a in *ArrayComp a* represents the numeric type of the primitive data stored in the arrays. We assume for now that this type is uniform,

```

type Var      = String
data ArithOp = Plus | Minus | Times | Divide
data ArrayComp a
  = GetData FilePath Shape      -- primitive injection of some packed numbers
  | Literal [a] Shape           -- another primitive injection
  | Scalar a                    -- a single value can expand to any shape
  | Enumerate a (a → a) Int     -- value generator loop (avoids using storage)
  | Split Int (ArrayComp a)    -- divide packed shape into n equal-sized shapes
  | Join (ArrayComp a)        -- opposite of split: fuse outer dimensions
  | Zip [ArrayComp a]         -- zip several equal-sized shapes together
  | Repack (ArrayComp a)      -- semantic identity, representational change
  | Let Var (ArrayComp a)     -- let-binding
      (ArrayComp a)
  | Var Var                   -- a bound variable
  | LetMap Var (ArrayComp a)  -- A let-binding and structural map combined
      (ArrayComp a)
  | Arith ArithOp (ArrayComp a) -- pointwise arithmetic over all shapes
      (ArrayComp a)
  | Fold ArithOp (ArrayComp a) -- fold only numeric operations
      (ArrayComp a)
  | Twist Structural (ArrayComp a) -- apply a purely structural transformation
  | HasShape (ArrayComp a) Shape -- shape annotation (user or inferred)

data Structural
  = Transpose                 -- (s 'Of' t 'Of' u) -> (t 'Of' s 'Of' u)
  | RotateL                   -- (s 'Of' t 'Of' u) -> (t 'Of' u 'Of' s)
  | RotateR                   -- (s 'Of' t 'Of' u) -> (u 'Of' s 'Of' t)
  | Slice      Index         -- (s 'Of' t 'Of' u) -> (t 'Of' u)
  | SubRange  Index Index   -- (s 'Of' t 'Of' u) -> ((j-i) 'Of' t 'Of' u)
  | DownSample Int          -- (s 'Of' t 'Of' u) -> (i 'Of' t 'Of' u)
  | Smap      Structural    -- f (s 'Of' t) -> (s 'Of' f t)
  | Compose  Structural Structural -- natural composition

```

Figure 1: A domain-specific language for array manipulation.

that is, there is no need to mix *Float*, *Double*, and *Int*.

In order to model the dimensionality of arrays, we do *not* use Haskell's type system. Every array computation has the same type. But nevertheless we can represent the intended shape of arrays in the *value* world. For instance:

```
data Shape = N Int | Shape 'Of' Shape
```

For any array computation, it is possible to infer its *Shape*, by recursive traversal of the *ArrayComp* structure.

We examine each of the constructors of the array computation language in turn, by looking at the type signature of the operation each constructor represents.

```
GetData :: FilePath → Shape → ArrayComp a
```

How can primitive data enter the computation? It must come from somewhere. The *GetData* constructor glosses over the details somewhat, but our view here is that a file is immutable, read-only, and therefore a pure array value. The *Shape* argument tells how many samples live in this array, and how the sequence is to be interpreted as a multi-dimensional structure.

```
Literal :: [a] → Shape → ArrayComp a
```

In some programs, particularly for testing, it may be convenient to write small arrays literally in the source code. The data here is held in a list, and the *Shape* argument is used as a check that the size of the list matches the shape specification.

```
Scalar :: a → ArrayComp a
```

Scalar values are frequently useful in array computations, e.g. to multiply every element of a vector by the same number. However, there is no need to treat scalars differently from actual array values. Conceptually, a *Scalar* is a vector of any size or dimension, filled with a single value. Inference thus determines the *Shape* of a *Scalar* by the context where it is used.

```
Enumerate :: a → (a → a) → Int → ArrayComp a
```

Sometimes the values in an array can be calculated entirely from some seed value and a transformation function. (*Enumerate* is like an *unfold* in conventional Haskell.) In the implementation, there may be no need to store the values in memory blocks, if they can be calculated on-the-fly as needed.

```
Split :: Int → ArrayComp a → ArrayComp a
```

A packed block of n values can be viewed equally as having a *Shape* that is one-dimensional $N\ n$, or two-dimensional $N\ i\ \text{'Of'}\ N\ j$ **where** $i * j \equiv n$. The *Split* i function adds one dimension to an array's shape, by cutting it into i pieces.

$Join :: ArrayComp\ a \rightarrow ArrayComp\ a$

Join is in some respects the opposite of *Split*. It fuses the two outer dimensions of the given array, so for instance a *Shape* consisting of $N\ p\ 'Of'\ N\ q\ 'Of'\ N\ r$, would be transformed into $N\ (p * q)\ 'Of'\ N\ r$.

$Zip :: [ArrayComp\ a] \rightarrow ArrayComp\ a$

Zip is like a *Join*, but where the (equal-shaped) arrays in its list argument may come from separate sources, not from simply the dimensional transformation of a single array. Since the arrays being zipped together are unlikely to live originally in the same physical block of memory, so this operation causes contiguous copies to be made. However, it is believed that it may be possible (with intelligent analysis and compilation, not yet developed) to avoid copying blocks in certain situations.

$Repack :: ArrayComp\ a \rightarrow ArrayComp\ a$

Occasionally, we envisage that it may be useful to have an explicit action in a program that allocates and packs a new array from an old array, such that elements that were index-wise close together, but physically far apart in memory, now become physically close. Semantically, *Repack* is an identity transformation, but practically, it may lead to better cache performance. The user is free to insert a *Repack* anywhere. Eventually, we hope a static DSL program analysis will be able to insert and remove *Repacks* automatically.

$Let :: Var \rightarrow ArrayComp\ a \rightarrow ArrayComp\ a \rightarrow ArrayComp\ a$

We permit let-binding: a way of giving a name to the first computation, so that its evaluation can be shared in the body of the second computation.

$Var :: Var \rightarrow ArrayComp\ a$

Hence, if the result of any computation can be named, we must be able to refer to its value later on, through its *Var* name.

$LetMap :: Var \rightarrow ArrayComp\ a \rightarrow ArrayComp\ a \rightarrow ArrayComp\ a$

The functorial map is the usual idea, taking a function and an array computation, to produce a new array computation with each element of the result array constructed by the application of the function to the elements of the argument array. The choice of representation of the function here is interesting however. It is essentially a let-binding: the *Var* is bound in turn to each of the one-dimensionally lower sub-arrays of the first array argument; the *Var* is in scope inside the second *ArrayComp* argument, which represents the body of the mapped function.

Consider for instance the following 3-D array computation to generate in each position, a triple representing its location in space.

```

positions :: (Num a) => Int -> Int -> Int -> ArrayComp a
positions x y z =
  LetMap "k" (Enumerate 0 (+1) z)
    (LetMap "j" (Enumerate 0 (+1) y)
      (LetMap "i" (Enumerate 0 (+1) x)
        (Join [Var "i", Var "j", Var "k"])))

```

There are other possible choices of function representation. One would be the more obvious:

```

AMap :: (ArrayComp a -> ArrayComp a) -> ArrayComp a -> ArrayComp a

```

where the function transforms an array computation. However, this is a poor choice because it suggests that the function might be able to return different results depending on the *construction* of its array computation argument, e.g. by pattern-matching over *Scalar* vs *Enumerate*. This would be highly undesirable, as it is not referentially transparent.

Another choice could be to treat the function more like a lambda-binding than a let-binding:

```

AMap :: (Var -> ArrayComp a) -> ArrayComp a -> ArrayComp a

```

Like the let-bound version, the *Var* is bound in turn to each one-dimensional lower structure of the array argument. The difference is that the programmer does not get to choose the name of the *Var*. Further, it is more difficult for the compiler to analyse and transform this style, because the function is a Haskell function, and therefore opaque.

```

Arith :: ArithOp -> ArrayComp a -> ArrayComp a -> ArrayComp a

```

Simple arithmetic operations can be interpreted to apply at any dimension, by point-wise application to the individual elements of the argument arrays.

```

Fold :: ArithOp -> ArrayComp a -> ArrayComp a -> ArrayComp a

```

For now, we permit only numeric operations as folds, rather than anything more general. The second argument is the accumulator array, of one dimension lower than the third argument, which is the array being folded over.

In principle, this signature could perhaps be generalised to accept a less constrained function, $Var \rightarrow Var \rightarrow ArrayComp a$, by analogy with the *LetMap* construction. Similar issues arise over whether to lambda-bind the variables or let-bind them.

```

Twist :: Structural -> ArrayComp a -> ArrayComp a

```

Many operations on arrays are purely structural transformations, not needing to look at the values in the array. The *Structural* type abstracts all of these operations. In the flat-packed representation, all these operations are extremely

cheap, because they only need to manipulate the traversal order of the array, without moving or copying the data itself. Multiple composed structural transformations can be fused at compile-time into a single re-arrangement of traversal order.

The available *Structural* transformations are as follows: *Transpose* swaps the two outer dimensions; *RotateL* rotates the dimensions such that the outermost is treated as the innermost; *RotateR* rotates the dimensions in the opposite direction, such that the innermost becomes the outermost; *Slice i* indexes the array's outermost dimension at *i*, producing a result one dimension smaller; *SubRange i j* takes a contiguous subsection of the outermost dimension of the array, that is, all of the slices between the two indices (inclusive); *DownSample i* reduces the size of the outermost dimension by keeping only every *i*th element; *Map s* applies a *Structural* to the elements of the outer dimension, reconstructing the outer layer from the results; *Compose s t* is the natural composition of two *Structural* operations.

$$\text{HasShape} :: \text{ArrayComp } a \rightarrow \text{Shape} \rightarrow \text{ArrayComp } a$$

Where there might be doubt in the programmer's mind about the dimension or size of an array, any expression can be given a *Shape* annotation. This will be checked by the inferencer. (Additionally, the inferencer can use this construction to annotate all nodes of the expression tree with shapes, since they will prove useful in compilation or interpretation of the DSL.)

4 Sugaring the language

The DSL can be made to resemble ordinary Haskell, through the definition of a library of Prelude-like functions to construct terms of the language. Here is a very incomplete sketch:

```

head :: ArrayComp a → ArrayComp a
tail :: ArrayComp a → ArrayComp a
head a = Twist (Slice 0) a
tail a = Twist (SubRange 1 end) a
  where end = size a

zipWith :: ArithOp → ArrayComp a → ArrayComp a → ArrayComp a
zipWith (*) a b = Arith (*) (Twist RotateL (Zip [a, b]))

concat :: [ArrayComp a] → ArrayComp a
concat = Join ∘ Zip

instance Num a ⇒ Num (ArrayComp a) where
  (+) = Arith Plus
  (*) = Arith Times

```

A higher-level library of standard vector and matrix functions can be defined as well. For instance:

```

dotProduct :: ArrayComp a → ArrayComp a → ArrayComp a
dotProduct xs ys =
  Twist Transpose
    (LetMap "y" (Twist Transpose ys)
      (Fold (+) (Scalar 0)
        (Twist Transpose
          (LetMap "x" xs
            (Arith (*) (Var "x") (Var "y"))))))))

```

5 A two-stage model for manipulating arrays

The DSL admits many possible implementations, through different representation choices for the real arrays being manipulated. We present a relatively simple implementation, with the expectation that different representations may be developed later to provide better performance at the cost of greater complexity. (Source code in *Data.Array.Flatpack.Interpret*.)

```

data Block a = Block{ block_data :: (UArr a)
                      , block_start :: Int
                      , block_steps :: Iterator }
type Iterator = [ForLoop]
data ForLoop = FL { jump :: Int
                   , count :: int }

```

Flat blocks of values are stored in memory as unboxed arrays (here using *UArr* from the *uvector* package for convenience). Associated with every block is a starting position, the origin in the rectangular multi-dimensional space, beyond which all values lie. To represent the dimensional structuring, an ordered series of *ForLoops* is also stored. All such *ForLoops* start at the origin *block_start*, and have parameters for *jump* and *count*. The order of the sequence corresponds to the nesting of loops for traversing the dimensions.

As an illustration, consider the simple case of a 2-dimensional array, of size 3×4 , where the first dimension varies slowest. Its *Iterator* representation would be $[FL\ 4\ 3, FL\ 1\ 4]$. Its meaning can be read as: to get from any item to its next adjacent item in the outer dimension, jump forward by 4 (unless you have exhausted all 3 of them). To get from any item to its next adjacent item in the inner dimension, jump forward by 1 (unless you have exhausted all 4 of them).

Now consider the transpose of the same array. Without changing, or even looking at, the order of the data stored in the array, we can simply swap the order of the *ForLoops* in the *Iterator* to $[FL\ 1\ 4, FL\ 4\ 3]$. The desired semantics still hold!

At the final stage of computation over some array data, the resulting block of data must eventually be traversed in the calculated order, using a function like *repack* to ensure that the result is contiguous for output to a file, for instance.

```

repack  :: (UA a) => Block a -> UArr a
repack b = let src = block_data b in
           mapU (src `indexU`) (walk (block_start b) (block_steps b))

```

However before the final result is needed, our evaluator *interpret* is free to convert an *ArrayComp* *a* to a *Block* by means of many intermediate *Blocks* which share the same data in memory, but have different Iterators. Many of the steps of interpretation involve no manipulation of the array data itself, only of the associated *ForLoops*, whilst some of the constructions of the DSL will require *repacking* and copying to intermediate blocks.

```

interpret :: (UA a, Num a) =>
            Environment a -> ArrayComp a -> Block a

```

Because our language allows bindings of variable names, the interpreter requires an *Environment* *a* to map variable names to their values.

6 Laws of array computation

We expect algebraic laws over array computations to become useful in this work. For instance,

```

Transpose (Transpose a) ≡ a
Index i (LetMap v a e) ≡ Let v (Index i a) e
Index i (Arith op a b) ≡ Arith op (Index i a) (Index i b)

```

In optimisation, the direction in which to apply these laws will be determined by the relative performance of each side. Thus, it is cheaper to do arithmetic only over small parts of an array, rather than take a small part of an array that is the result of much arithmetic.

6.1 Parallel: SIMD and multi-core

With a slightly more complicated model of the data dependencies between array computations, it should be possible to automatically split the traversals of memory blocks into separate parallel loops over different parts of the block. The individual pieces of the block could then be farmed out to separate processing threads on a multi-core machine for parallel evaluation.

Conversely, where an analysis can determine that successive values in arithmetic operations are physically close-packed in memory (and will be traversed in that order by the algorithm), then SIMD-parallelism can be exploited to execute multiple operations simultaneously in a single machine instruction.

7 Open issues

7.1 Dynamic sizing

The DSL currently deals only with computations over arrays of fixed and statically-known size. There is no way to express computations returning an array whose size is unknown, like a *filter* predicate.

$$\text{Filter} :: (\text{Var} \rightarrow \text{Bool}) \rightarrow \text{ArrayComp } a \rightarrow \text{ArrayComp } a$$

It is not clear what type should be used for the predicate argument. As here, a lambda-bound function over the iterator variable gives very little flexibility in what can actually be tested.

It is also not clear what shape can be inferred for the resultant array computation. Its total dimension is the same as the argument array, its inner dimensions are also of known size, but the size of its outer dimension is unknown. In any case it can be no larger than that of the argument. If the result is to be used in an enclosing array expression, whose constructor is shape-sensitive (such as arithmetic), then do we need to introduce dynamic size-checking?

7.2 Conditionals

Related to dynamic sizes, many algorithms require conditional computation based on the values found in the arrays. It is common for hardware technology to use zero-values to represent False, and this could be the interpretation we adopt too, in order to make conditionals possible. What conditional primitive should we add to the language? Is *If – Then – Else* adequate? Is *Case* discrimination desirable as an alternative?

7.3 Lazy Streaming

The sketch shown so far assumes that data will live in memory blocks. When the data becomes too large to fit, what then? Is it possible to come up with yet another implementation model for array computations, based on lazy streaming windows over the file-based dataset? The DSL array computation language is independent of the implementation model, but how much (and what) extra analysis would be needed to determine whether any given expression's traversal order is streaming-window friendly, or if not, can it be transformed by use of algebraic laws into one that is window-friendly?

Another possible data model could be a chain-of-chunks. Each individual chunk is a flat-packed array, but once its boundary is exceeded, the next chunk in the sequence can be loaded.

7.4 Shape inference over functions

Simple shape inference of values is implemented. Is it possible to infer a shape signature for a Haskell function that creates an array computation from constituent computations? Such a signature would likely not contain definite sizes

and dimensions, but rather constraints on minimum dimension and size, and perhaps whether a size is a multiple of some divisor.

7.5 Heterogeneous tuples

Currently, we treat homogeneous tuples like arrays, viz. the triple (a, a, a) is like a 3-place array of a . But can we extend the model to permit heterogeneous tuples: to what extent is $Array(a, b)$ isomorphic to $(Array\ a, Array\ b)$? What extra constraints does this place on shape inference, and does it make implementation of evaluation more complex?

8 Related Work

There are a variety of interesting approaches to high-performance computing with arrays. This is very incomplete and hasty survey of them: Single-Assignment C (SAC) has very sophisticated shape analysis, and achieves very good performance, comparable with Fortran for some computations. Data-Parallel Haskell is an attempt to gain speed by splitting an array computation over multiple parallel hardware resources. It uses a flattening transformation to deal with nested array structures, together with a powerful traversal-fusion framework implemented as transformation rules for the GHC compiler's extensible optimiser.

Historically, the ideas of nested data parallelism came from the language NESL.

9 Progress and Conclusion

This extended abstract has set out the basis of a preliminary investigation into dealing with large quantities of numeric data in a functional language such as Haskell. We have a draft definition of an array DSL, and an interpreter for it that uses underlying flat unboxed arrays, with a separate model of traversal patterns. This interpreter has been used to validate the semantics of the language. It remains to demonstrate empirically the performance of the FlatPack approach.

Future work might improve the internal representation of array computations to remove more intermediate copying operations, as well as to adapt the execution model for SIMD, multi-core, or GPGPU resources.