Regular, shape-polymorphic, parallel arrays in Haskell

Gabriele Keller[†] Manuel M. T. Chakravarty[†] Roman Leshchinskiy[†]
Simon Peyton Jones[‡] Ben Lippmeier[†]

[†]Computer Science and Engineering
University of New South Wales, Australia
{keller,chak,rl,benl}@cse.unsw.edu.au

[‡]Microsoft Research Ltd
Cambridge, England
{simonpj}@microsoft.com

Abstract

We present a novel approach to regular, multi-dimensional arrays in Haskell. The main highlights of our approach are that it (1) is purely functional, (2) supports reuse through shape polymorphism, (3) avoids unnecessary intermediate structures rather than relying on subsequent loop fusion, and (4) supports transparent parallelisation.

We show how to embed two forms of shape polymorphism into Haskell's type system using type classes and type families. In particular, we discuss the generalisation of regular array transformations to arrays of higher rank, and introduce a type-safe specification of array slices.

We discuss the runtime performance of our approach for three standard array algorithms. We achieve absolute performance comparable to handwritten C code. At the same time, our implementation scales well up to 8 processor cores.

1. Introduction

In purely functional form, array algorithms are often more elegant and easier to comprehend than their imperative, explicitly loop-based counterparts. The question is, can they also be efficient?

Experience with Clean, OCaml, and Haskell has shown that we can write efficient code if we sacrifice purity and use an *imperative array interface* based on reading and writing *individual array elements*, possibly wrapped in uniqueness types or monads [9, 11, 13]. However, using impure features not only obscures clarity, but also forfeits the transparent exploitation of the data parallelism that is abundant in array algorithms.

In contrast, using a *purely-functional array interface* based on *collective operations* —such as maps, folds, and permutations— emphasises an algorithm's high-level structure and often has an obvious parallel implementation. This observation was the basis for previous work on algorithmic skeletons and the use of the *Bird-Meertens Formalism (BMF)* for parallel algorithm design [17]. Our own work on *Data Parallel Haskell (DPH)* is based on the same premise, but aims at irregular data parallelism which comes with its own set of challenges [16]. Other work on high-performance *byte arrays* [7] also aims at abstracting over loop-based low-level code using a purely-functional combinator library.

We aim higher by supporting multi-dimensional arrays, more functionality, and transparent parallelism. We present a Haskell library of regular parallel arrays, which we call *Repa*[1] (Regular Parallel Arrays). While Repa makes use of the Glasgow Haskell Compiler's many existing extensions, it is a pure library: it does not require any compiler support that is specific to its implementation. The resulting code is not only as fast as when using an imperative array interface, it approaches the performance of handwritten C code, and exhibits good parallel scalability on the configurations that we benchmarked.

In addition to good performance, we achieve a high degree of reuse by supporting *shape polymorphism*. For example, `map` works over arrays of arbitrary rank, while `sum` decreases the rank of an arbitrary array by one – we give more details in Section 4. The value of shape polymorphism has been demonstrated by the language Single Assigment C, or SAC [18]. Like us, SAC aims at purely functional high-performance arrays, but in contrast to our work, SAC is a specialised array language based on a purpose-built compiler. We show how to embed shape polymorphism into Haskell's type system.

The main contributions of the paper are the following:

- An API for purely-functional, collective operations over dense, rectangular, multi-dimensional arrays supporting shape polymorphism (Section 5).
- Support for various forms of constrained shape polymorphism in a Hindley-Milner type discipline with type classes and type families (Section 4).
- An aggressive loop fusion scheme based on a functional representation of delayed arrays (Section 6).
- A scheme to transparently parallelise array algorithms based on our API (Section 7)
- An evaluation of the sequential and parallel performance of our approach on the basis of widely used array algorithms (Section 8).

Before diving into the technical details of our contributions, the next section illustrates our approach to array programming by way of an example.

```
extent  :: Array sh e -> sh

sum     :: (Num e, Elt e)
        => Array (sh :. Int) e -> Array sh e

zipWith :: (Shape sh, Elt e1, Elt e2, Elt e3)
        => (e1 -> e2 -> e3)
        -> Array sh e1 -> Array sh e2
        -> Array sh e3

backpermute :: (Shape sh, Shape sh', Elt e)
            => sh' -> (sh' -> sh)
            -> Array sh e -> Array sh' e
```

Figure 1. Types of library functions

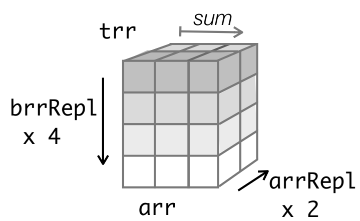

Figure 2. Matrix-matrix multiplication illustrated

2. Our approach to array programming

A simple operation on two-dimensional matrices is transposition. With our library we express transposition in terms of a permutation operation that swaps the row and column indices of a matrix:

```
transpose2D :: Elt e => Array DIM2 e -> Array DIM2 e
transpose2D arr
  = backpermute new_extent swap arr
  where
    swap (Z :.i :.j) = Z :.j :.i
    new_extent       = swap (extent arr)
```

Like Haskell 98 arrays, our array type is parameterised by the array's *index type*, here DIM2, and by its element type e. The index type gives the *rank* of the array, which we also call the array's *dimensionality*, or *shape*.

Consider the type of backpermute, given in Figure 1. The first argument is the bounds (or *extent*) of the result array, which we obtain by swapping the row and column extents of the input array. For example transposing a 3×12 matrix gives a 12×3 matrix.[2] The backpermute function constructs a new array in terms of an existing array solely through an *index transformation*, supplied as its second argument, swap: given an index into the result matrix, swap produces the corresponding index into the argument matrix.

A more interesting example is matrix-matrix multiplication:

```
mmMult :: (Num e, Elt e)
       => Array DIM2 e -> Array DIM2 e
       -> Array DIM2 e
mmMult arr brr
 = sum (zipWith (*) arrRepl brrRepl)
 where
  trr     = transpose2D brr
  arrRepl = replicate (Z :.All :.colsB :.All)   arr
  brrRepl = replicate (Z :.All :.All   :.rowsA) trr

  (Z:. colsA:. rowsA) = extent arr
  (Z:. colsB:. rowsB) = extent brr
```

The idea is to expand both rank-two argument arrays into rank-three arrays by replicating them across a new dimension, or axis, as illustrated in Figure 2. The front face of the cuboid represents the array arr, which we replicate as often as brr has columns (colsB), producing arrRepl. The top face represents trr (the transposed brr), which we replicate as often as arr has rows (rowsA), producing brrRepl. As indicated by the figure, the two replicated arrays have the same extent, which corresponds to the index space of matrix multiplication:

$$(AB)_{i,j} = \Sigma_{k=1}^{n} A_{i,k} B_{k,j}$$

where i and j correspond to rowsA and colsB in our code. The summation index k corresponds to the innermost axis of the replicated arrays and to the left-to-right axis in Figure 2. Along this axis we perform the summation after an elementwise multiplication of the replicated elements of arr and brr by zipWith (*).

A naive implementation of the operations used in mmMult would result in very bad space and time performance. In particular, it would be very inefficient to compute explicit representations of the replicated matrices arrRepl and brrRepl. Indeed, a key principle of our design is to avoid generating explicit representations of the intermediate arrays that can be represented as the original arrays combined with a *transformation of the index space* (Section 3.2). A rigorous application of this principle results in aggressive loop fusion (Section 6) producing code that is similar to imperative code. As a consequence, this Haskell code has about the same performance as handwritten C code for the same computation. Even more importantly, we measured very good absolute speedup, $\times 7.7$ for 8 cores, on multicore hardware — a property that the C code does not have without considerable additional effort!

3. Representing arrays

The representation of arrays is a central issue in any array library. Our library uses two devices to achieve good performance:

1. We represent array data as contiguously-allocated ranges of *unboxed* values.
2. We *delay* the construction of intermediate arrays to support constant-time index transformations and slices, and to combine these operations with traversals over successive arrays.

We describe these two techniques in the following sections.

3.1 Unboxed arrays

In Haskell 98 arrays are lazy, so that each element of an array is evaluated only when the array is indexed at that position. Although convenient, laziness is Very Bad Indeed for array-intensive programs:

- A lazy array of (say) Float is represented as an array of pointers to either heap-allocated thunks, or boxed Float objects, depending on whether they have been forced. This representation requires at least three times as much storage as a conventional, contiguous array of unboxed floats. Moreover, when iterating through the array, the lazy representation imposes higher memory traffic. This is due to the increased size of the individual elements, as well as their lower spacial locality.
- In a lazy array, evaluating one element does not mean that the other elements will be demanded. However, the overwhelm-

[1] Repa means "turnip" in Russian.

[2] For now, just read the notation (Z :.i :.j) as if it was the familiar pair (i,j). The details are in Section 4 where we discuss shape polymorphism.

ingly common case is that the programmer intends to demand the entire array, and wants it evaluated in parallel.

We can solve both of these problems simultaneously using a Haskell-folklore trick. We define a new data type of arrays, which we will call `UArr`, short for "unboxed array". These arrays are *one-dimensional*, indexed by `Int`, and are slightly stricter than Haskell 98 arrays: a `UArr` *as a whole* is evaluated lazily, but an attempt to evaluate any *element* of the array (e.g. by indexing) will cause evaluation of all the others, in parallel.

For the sake of definiteness we give a bare sketch of how `UArr` is implemented. However, this representation is not new; it is well established in the Haskell folklore, and we use it in Data Parallel Haskell (DPH) [5, 16], so we do not elaborate the details.

```
class Elt e where
  data UArr e
  (!) :: Array e -> Int -> e
  ...more methods...

instance Elt Float where
  data UArr Float = UAF Int ByteArray#
  (UAF max ba) ! i
     | i < max   = F# (indexByteArray ba i)
     | otherwise = error "Index error"
  ...more methods...

instance (Elt a, Elt b) => Elt (a :*: b) where
  data UArr (a :*: b) = UAP (UArr a) (UArr b)
  (UAP a b) ! i = (a!i :*: b!i)
  ...more methods...
```

Here we make use of Haskell's recently added *associated data types* [4] to represent an array of `Float` as a contiguous array of unboxed floats (the `ByteArray#`), and an array of pairs as a pair of arrays. Because the representation of the array depends on the element type, indexing must vary with the element type too, which explains why the indexing operation `(!)` is in a type class `Elt`.

In addition to an efficient underlying array representation, we also need the infrastructure to operate on these arrays in parallel, using multiple processor cores. To that end we reuse part of our own parallel array library of Data Parallel Haskell. This provides us with an optimised implementation of `UArr` and the `Elt` class, and with parallel collective operations over `UArr`. It also requires us represent pairs using the strict pair constructor `(:*:)`, instead of Haskell's conventional `(,)`.

3.2 Delayed arrays

When using Repa, index transformations such as `transpose2D` (discussed in Section 2) are ubiquitous. As we expect index transformations to be cheap, it would be wrong to (say) copy a 100Mbyte array just to transpose it. It is much better to push the index transformation into the consumer, which can then consume the original, unmodified array.

We could do this transformation statically, at compile time, but doing so would rely on the consumer being able to "see" the index transformation. This could make it hard for the programmer to predict whether or not the optimisation would take place. In Repa we instead perform this optimisation dynamically, and offer a guarantee that index transformations perform no data movement. The idea is simple and well known: just represent an array by its indexing function, together with the array bounds[3]:

```
data Array sh e = Array sh (sh -> e)
```

[3] NB: this is not our final array representation!

With this representation, functions like `backpermute` (discussed in Section 2, with type signature in Figure 1) are quite easy to implement:

```
backpermute sh' fn (Array sh ix1)
  = Array sh' (ix1 . fn)
```

We can also wrap a `UArr` as an `Array`:

```
wrap :: (Shape sh, Elt e)
     => sh -> UArr e -> DArray sh e
wrap sh uarr  = Array sh idx
  where idx i = uarr ! index sh i
```

When wrapping an `DArray` over a `UArr`, we also take the opportunity to generalise from one-dimensional to multi-dimensional arrays. The index of these multi-dimensional arrays is of type `sh`, where the `Shape` class (to be described in Section 4) includes the method `index :: Shape sh => sh -> sh -> Int`. This method maps the bounds and index of an `Array` to the corresponding linear `Int` index in the underlying `UArr`.

3.3 Combining the two

Unfortunately, there are at least two reasons why it is not always beneficial to delay an array operation. One is *sharing*, which we discuss later in Section 6. Another is *data layout*. In our `mmMult` example from Section 2, we want to delay the two applications of `replicate`, but not the application of `transpose2D`. Why? We store multi-dimensional arrays in row-major order (the same layout Haskell 98 uses for standard arrays). Hence, iterating over the second index of an array of rank 2 is more cache friendly than iterating over its first index.

It is well known that the order of the loop nest in an imperative implementation of matrix-matrix multiplication has a dramatic effect on performance due to these cache effects. By forcing `transpose2D` to produce its result as an unboxed array in memory —we call this a *manifest* array— instead of leaving it as a delayed array, the code will traverse both matrices by iterating over the second index in the inner loop. Overall, we have the following implementation

```
mmMult arr brr
 = sum (zipWith (*) arrRepl brrRepl)
 where
  trr     = force (transpose2D brr)  -- New! force!
  arrRepl = replicate (Z :.All :.colsB :.All)   arr
  brrRepl = replicate (Z :.All :.All   :.rowsA) trr

  (Z:. colsA:. rowsA) = extent arr
  (Z:. colsB:. rowsB) = extent brr
```

We could implement `force` by having it produce a value of type `UArr` and then apply `wrap` to turn it into a `DArray` again, providing the appropriate memory layout for a cache-friendly traversal. This would work, but we can do better. The function `wrap` uses array indexing to access the underlying `UArr`. In cases where this indexing is performed in a tight loop, GHC can optimise the code more thoroughly when it is able to inline the indexing operator, instead of calling an anonymous function encapsulated in the data type `DArray`. For recursive functions, this also relies on the *constructor specialisation* optimisation [15]. However, as explained in Coutts et al. [6], Section 7.2, to allow this we must make the special case of a wrapped `UArr` explicit in the datatype, so the optimiser can see whether or not it is dealing directly with a manifest array.

Hence, we define regular arrays as follows:

```
data Array sh e = Manifest sh (UArr e)
                | Delayed  sh (sh -> e)
```

```
infixl 3 :.
data Z            = Z
data tail :. head = tail :. head

type DIM0 = Z
type DIM1 = DIM0:.Int
type DIM2 = DIM1:.Int
type DIM3 = DIM2:.Int

class Shape sh where
  rank  :: sh -> Int
  size  :: sh -> Int         -- Number of elements
  index :: sh -> sh -> Int  -- Index into row-major
  fromIndex                 --   representation
        :: sh -> Int -> sh  -- Inverse of 'index'
  <..and so on..>

instance             Shape Z        where ...
instance Shape sh => Shape (sh:.Int) where ...
```

Figure 3. Definition of shapes

We can unpack an arbitrary `Array` into delayed form thus:

```
delay :: (Shape sh, Elt e)
      => Array sh e -> (sh, sh -> e)
delay (Delayed  sh f)    = (sh, f)
delay (Manifest sh uarr)
  = (sh, \i -> uarr ! index sh i)
```

This is the basis for a general `traverse` function that produces a delayed array after applying a transformation. The transformation may include index space transformations or other computations:

```
traverse :: (Shape sh, Shape sh', Elt e)
         => Array sh e
         -> (sh -> sh')
         -> ((sh -> e) -> sh' -> e')
         -> Array sh' e'
traverse arr sh_fn elem_fn
  = Delayed (sh_fn sh) (elem_fn f)
  where (sh, f) = delay arr
```

We use `traverse` to implement many of the other operations of our library — for example, `backpermute` is implemented as:

```
backpermute :: (Shape sh, Shape sh', Elt e)
            => sh' -> (sh' -> sh) -> Array sh e
            -> Array sh' e
backpermute sh pm = traverse (const sh) (. pm)
```

We discuss the use of `traverse` in more detail in Sections 5 & 7.

4. Shapes and shape polymorphism

In Figure 1 we gave this type for `sum`:

```
sum :: (Shape sh, Num e, Elt e)
    => Array (sh:.Int) e -> Array sh e
```

As the type suggests, `sum` is a *shape-polymorpic* function: it can sum the rightmost axis of an array of arbitrary rank. In this section we describe how shape polymorphism works in Repa. We will see that combination of parametric polymorphism, type classes, and type families enables us to track the rank of each array in its type, guaranteeing the absence of rank-related runtime errors. We can do this even in the presence of operations such as slicing and replication that change the rank of an array. However, bounds checks on indices are still performed at runtime — tracking them requires more sophisticated type system support [20, 24].

4.1 Shapes and indices

Haskell's tuple notation does not allow us the flexibility we need, so we introduce our own notation for indices and shapes. As defined in Figure 3, we use an inductive notation of tuples as heterogenous *snoc* lists. On both the type-level and the value-level, we use the infix operator `(:.)` to represent *snoc*. The constructor `Z` corresponds to a rank zero shape, and we use it to mark the end of the list. Thus, a three-dimensional index with components `x`, `y` and `z` is written `(Z:.x:.y:.z)` and has type `(Z:.Int:.Int:.Int)`. This type is the *shape* of the array. Figure 3 gives type synonyms for common shapes: a singleton array of shape `DIM0` represents a scalar value; an array of shape `DIM1` is a vector, and so on.

The motivation for using *snoc* lists, rather than the more conventional *cons* lists, is this. We store manifest arrays in row-major order, where the *rightmost* index is the most *rapidly-varying* when traversing linearly over the array in memory. For example, the value at index `(Z:.3:.8)` is stored adjacent to that at `(Z:.3:.9)`. This is the same convention adopted by Haskell 98 standard arrays.

We draw array indices from `Int` values only, so the shape of a rank-n array is:

$$\mathtt{Z}\ \underbrace{\mathtt{:.\ Int\ :.}\ \cdots\ \mathtt{:.\ Int}}_{n \text{ times}}$$

In principle, we could be more general and allow non-`Int` indices, like Haskell's index type class `Ix`. However, this would complicate the library and the presentation, and is orthogonal to the contributions of this paper; so we will not consider it here. Nevertheless, shape types, such as `DIM2` etc, explicitly mention the `Int` type. This is for two reasons: firstly, it simplifies the transition to using the `Ix` class if that is desired; and secondly, in Section 4.4 we discuss more elaborate shape constraints that require an explicit index type.

The *extent* of an array is a value of the shape type:

```
extent :: Array sh e -> sh
```

The corresponding Haskell 98 function, `bounds`, returns an upper and lower bound, whereas `extent` returns only the upper bound. Repa uses zero-indexed arrays only, so the lower bound is always zero. For example, the extent `(Z:.4:.5)` characterises a 4×5 array of rank two containing 20 elements. The extent along each axis must be at least one.

The shape type of an array also types its indices, which range between zero and one less than the extent along the same axis. In other words, given an array with shape $(\mathtt{Z:.}n_1\mathtt{:.}\cdots\mathtt{:.}n_m)$, its index range is from `(Z:.0:.···:.0)` to $(\mathtt{Z:.}n_1-1\mathtt{:.}\cdots\mathtt{:.}n_m-1)$. As indicated in Figure 3, the methods of the `Shape` type class determine properties of shapes and indices, very like Haskell's `Ix` class. These methods are used to allocate arrays, index into their row-major in-memory representations, to traverse index spaces, and are entirely as expected, so we omit the details.

4.2 Shape polymorphism

We call functions that operate on a variety of shapes *shape polymorphic*. Some such functions work on arrays of any shape at all. For example, here is the type of `map`:

```
map :: (Shape sh, Elt a, Elt b)
    => (a -> b)
    -> Array sh a -> Array sh b
```

The function `map` applies its functional argument to all elements of an array without any concern for the shape of the array. The type class constraint `Shape sh` merely asserts that the type variable `sh` ought to be a shape. It does not constrain the shape of that shape in any way.

4.3 At-least constraints and rank generalisation

With indices as *snoc* lists, we can impose a lower bound on the rank of an array by fixing a specific number of lower dimensions, but keeping the tail of the resulting *snoc* list variable. For example, here is the type of `sum`:

```
sum :: (Shape sh, Num e, Elt e)
    => Array (sh:.Int) e -> Array sh e
```

This says that `sum` takes an array of any rank $n > 1$ and returns an array of rank $n - 1$. For a rank-1 array (a vector), `sum` adds up the vector to return a scalar. But what about a rank-2 array? In this case, `sum` adds up all the rows of the matrix *in parallel*, returning a vector of the sums. Similarly, given a three-dimensional array `sum` adds up each row of the array in parallel, returning a two-dimensional array of sums.

Functions like `sum` impose a lower bound on the rank of an array. We call such constraints shape polymorphic *at-least constraints*. Every shape-polymorphic function with an at-least constraint is implicitly also a data-parallel map over the unspecified dimensions. This is a major source of parallelism in Repa. We call the process of generalising the code defined for the minimum rank to higher rank *rank generalisation.*

The function `sum` only applies to the rightmost index of an array. What if we want to reduce the array across a different dimension? In that case we simply perform an index permutation, which is guaranteed cheap, to bring the desired dimension to the rightmost position:

```
sum2 :: (Shape sh, Elt e, Num e)
     => Array (sh:.Int:.Int) e -> Array (sh:.Int) e
sum2 a = sum (backpermute new_extent swap a)
  where
    new_extent          = swap (extent a)
    swap (is :.i2 :.i1) = is :.i1 :.i2
```

In our examples so far, we have sometimes returned arrays of a different rank than the input, but their extent in any one dimension has always been unchanged. However, shape-polymorphic functions can also change the extent:

```
selEven :: (Shape sh, Elt e)
       => Array (sh:.Int) e -> Array (sh:.Int) e
selEven arr = backpermute new_extent expand arr
  where
   (ns :.n)        = extent arr
   new_extent      = ns :.(n `div` 2)
   expand (is :.i) = is :.(i * 2)
```

As we can see from the calculation of `new_extent`, the array returned by `selEven` is half as big as the input array, in the rightmost dimension. The index calculation goes in the opposite direction, selecting every alternate element from the input array.

Note carefully that the extent of the new array is calculated from on the extent of the old array, *but not from the data in the array*. That guarantees that we can do rank generalisation and still have a rectangular array. To see the difference, consider:

```
filter :: Elt e => (e -> Bool)
       -> Array DIM1 e -> Array DIM1 e
```

The `filter` function is not, and cannot be, shape-polymorphic. If you filter each row of a matrix, based on the element values, each new row might have a different length, so there would be no guarantee that the resulting matrix was rectangular. We have carefully chosen our shape-polymorphic primitives to guarantee that this cannot happen.

```
data All    = All
data Any sh = Any

class Slice ss where
  type FullShape  ss
  type SliceShape ss
  replicate :: Elt e
            => ss
            -> Array (SliceShape ss) e
            -> Array (FullShape  ss) e
  slice     :: Elt e
            => Array (FullShape  ss) e
            -> ss
            -> Array (SliceShape ss) e

instance Slice Z where
  type FullShape  Z = Z
  type SliceShape Z = Z
  <..definition of replicate and slice omitted..>

instance Slice (Any sh) where
  type FullShape  (Any sh) = sh
  type SliceShape (Any sh) = sh
  replicate Any a = a
  slice a Any = a

instance (Shape (FullShape sl), Shape (SliceShape sl),
          Slice sl) => Slice (sl:.Int) where
  type FullShape  (sl:.Int) = FullShape  sl :. Int
  type SliceShape (sl:.Int) = SliceShape sl

  replicate (sl:.i) arr
     = backpermute (ex:.i) drop arr2
     where
       ex   = extent arr2
       arr2 = replicate sl arr
       drop (is:._) = is

  slice arr (sl:.i) = slice arr2  sl
     where
       arr2    = backpermute ex add arr
       (ex:._) = extent arr
       add is  = is:.i

instance Slice ss => Slice (ss:.All) where
  type FullShape  (ss:.All) = FullShape  ss :. Int
  type SliceShape (ss:.All) = SliceShape ss :. Int
  <..definition of replicate and slice omitted..>
```

Figure 4. Definition of slices

4.4 Slices and slice constraints

Shape types characterise a *single* shape. However, some collective array operations require a relationship between *pairs* of shapes. One such operation is `replicate`, which we used in `mmMult`. The function `replicate` takes an array of any rank and replicates it along one or more additional dimensions. We cannot uniquely determine the behaviour of `replicate` from the shape of the original and resulting array alone. For example, suppose that we want to use replicate to expand a rank-2 array into a rank-3 array. There are three ways of doing so, depending on which dimension of the result array is the duplicated one. Indeed, the two calls to `replicate` in `mmMult` performed replication along two different dimensions, corresponding to different sides of the cuboid in Figure 2.

It should be clear that `replicate` needs an additional argument, a *slice specifier*, that expresses exactly how the shape of the result array depends on the shape of the argument array. A slice specifier has the same format as an array index, but some index positions may use the value `All` instead of a numeric index.

```
data All = All
```

In `mmMult`, we use `replicate (Z:.All:.colsB:.All) arr` to indicate that we replicate `arr` across the second innermost axis, `colsB` times. We use `replicate (Z:.All:.All:.rowsA) trr` to specify that we replicate `trr` across the innermost axis, `rowsA` times.

The type of the slice specifier `(Z :.All :.colsB :.All)` is `(Z :.All :.Int :.All)`. This type is sufficiently expressive to determine the shape of *both* the original array, before it gets replicated, *and* of the replicated array. More precisely, both of these types are a function of the slice specifier type. In fact, we derive these shapes using *associated type families*, a recent extension to the Haskell type system [3, 19], using the definition for the `Slice` type class shown in Figure 4.

Unsurprisingly, `replicate` is a method of the `Slice` class, as is a closely-related function `slice`, which extracts a slice along multiple axes of an array. Their full types appear in Figure 4. We chose their argument order to match that used for lists: `replicate` is a generalisation of `Data.List.replicate`, while slice is a generalisation of `Data.List.(!!)`. The implementations of `replicate` and `slice` in the various instances of `Slice` are straightforward uses of `backpermute`, so we only give them for one of the instances.

Finally, to enable rank generalisation for `replicate` and `slice`, we add a last slice specifier, namely `Any`, which is also defined in Figure 4. It is used in the tail position of a slice, just like `Z`, but gives a shape variable for rank generalisation. With its aid we can write `repN` which replicates an arbitrary array `n` times, with the replicating being on the rightmost dimension of the result array:

```
repN :: Int -> Array sh e -> Array sh e
repN n a = replicate (Any:.n) a
```

5. Rectangular arrays, purely functional

As mentioned in Section 3, the type class `Elt` determines the set of types that can be used as array elements; we adopt this class from the library of unboxed one-dimensional arrays in Data Parallel Haskell. With this library, array elements can be of the basic numeric types, `Bool`, and pairs formed from the strict pair constructor:

```
data a :*: b = !a :*: !b
```

We also extended this to support index types, formed from `Z` and `(:.)` as array elements. Although it would be straightforward to allow other product and enumeration types as well, support for general sum types appears impractical in a framework based on regular arrays. Adding this would require irregular arrays and nested data parallelism [16].

Table 1 summarises the central functions of our library Repa. They are grouped according to the structure of the implemented array operations. We discuss the groups and their members in the following sections.

5.1 Structure-preserving operations

The simplest group of array operations are those that apply a transformation on individual elements without changing the shape, array size, or order of the elements. We have the plain `map` function, `zip` for element-wise pairing, and a family of `zipWith` functions that apply workers of different arity over multiple arrays in lockstep. In the case of `zip` and `zipWith`, we determine the shape value of the result by intersecting the shapes of the arguments — that is, we take the minimum extent along every axis. This behaviour is the same as Haskell's `zip` functions when applied to lists.

The function `map` is implemented as follows:

```
map :: (a -> b) -> Array sh a -> Array sh b
map f = traverse id (f .)
```

The various zip functions are implemented in a similar manner, although they also use a method of the `Shape` type class to compute the intersection shape of the arguments.

5.2 Reductions

Our library, Repa, provides two kinds of reductions: (1) generic reductions, such as `foldl`, and (2) specialised reductions, such as `sum`. In a purely sequential implementation, the latter would be implemented in terms of the former. However, in the parallel case we must be careful.

Reductions of an n element array can be computed with parallel tree reduction, providing $\log n$ asymptotic step complexity, but only if the reduction operator is associative. Unfortunately, Haskell's type system does not provide a way to express this side condition on the first argument of `foldl`. Hence, the generic reduction functions need to retain their sequential semantics to remain deterministic. In contrast, for specialised reductions such as `sum`, when we know that the operators they use meet the associativity requirement, we can use parallel tree reduction.

As outlined in Section 4.3, all reduction functions are defined with a shape polymorphic at-least constraint and admit rank generalisation. Therefore, even generic reductions, with their sequential semantics, are highly parallel if used with rank generalisation.

Rank generalisation also affects specialised reductions, as they can be implemented in one of the following two ways. If we want to maximise parallelism, we can use a segmented tree reduction that conceptually performs multiple parallel tree reductions concurrently. Alternatively, we can simply use the same scheme as for general reductions, and perform all rank one reductions in parallel. We follow the latter approach and sacrifice some parallelism, as tree reductions come with some sequential overhead.

In summary, when applied to an array of rank one, generic reductions (`foldl` etc.) execute purely sequentially with an asymptotic step complexity of n, whereas specialised reductions (`sum` etc.) execute in parallel using a tree reduction with an asymptotic step complexity of $\log n$. In contrast, when applied to an array of rank strictly greater than one, both generic and specialised reductions use rank generalisation to execute many sequential reductions on one-dimensional subarrays concurrently.

5.3 Index space transformations

The structure-preserving operations and the reductions transform array elements, where index space transformation only alter the index at which an element is placed — i.e., they rearrange and possibly drop elements. A prime example of this group of operations is `reshape`. It imposes a new shape on the elements of an array. A precondition of `reshape` is that the size of the extent of the old and new array is the same — i.e., the number of elements stays the same:

```
reshape :: Shape sh
        => sh -> Array sh' e -> Array sh e
reshape sh' (Manifest sh ua)
  = assert (size sh == size sh') $
      Manifest sh' ua
reshape sh' (Delayed sh f)
  = assert (size sh == size sh') $
      Delayed sh' (f . fromIndex sh . index sh')
```

Structure-preserving operations	
`map :: (Shape sh, Elt a, Elt b)` `=> (a -> b) -> Array sh a -> Array sh b`	Apply function to every array element
`zip :: (Shape sh, Elt a, Elt b)` `=> Array sh a -> Array sh b -> Array sh (a :*: b)`	Elementwise pairing
`zipWith :: (Shape sh, Elt a, Elt b, Elt c)` `=> (a -> b -> c) -> Array sh a -> Array sh b -> Array sh c`	Apply a function elementwise to two arrays (result shape value by shape intersection)
⟨Other map-like operations: `zipWith3`, `zipWith4`, and so on⟩	
Reductions	
`foldl :: (Shape sh, Elt a, Elt b)` `=> (a -> b -> a) -> a -> Array (sh:.Int) b -> Array sh a`	Left fold
⟨Other reduction schemes: `foldr`, `foldl1`, `foldr1`, `scanl`, `scanr`, `scanl1` & `scanr1`⟩	
`sum :: (Shape sh, Elt e, Num e) => Array (sh:.Int) e -> Array sh a`	Sum an array along its innermost axis
⟨Other specific reductions: `product`, `maximum`, `minimum`, `and` & `or`⟩	
Index space transformations	
`reshape :: Shape sh => sh -> Array sh' e -> Array sh e`	Impose a new shape on the same elements
`replicate :: (Slice sl, Elt e)` `=> sl -> Array (SliceShape sl) e -> Array (FullShape sl) e`	Extend an array along new dimensions
`slice :: (Slice sl, Elt e)` `=> Array (FullShape sl) e -> sl -> Array (SliceShape sl) e`	Extract a subarray according to the slice specification `sl`
`(+:+) :: Shape sh => Array sh e -> Array sh e -> Array sh e`	Append the second array at the first
`backpermute :: (Shape sh, Shape sh')` `=> sh' -> (sh' -> sh) -> Array sh e -> Array sh' e`	Backwards permutation

`backpermuteDft :: (Shape sh, Shape sh')`
`=> Array sh' e -> (sh' -> Maybe sh) -> Array sh e`
`-> Array sh' e`
	Default backwards permutation

`unit :: e -> Array Z e`	Wrap a scalar into a singleton array
`(!:) :: (Shape sh, Elt e) => Array sh e -> sh -> e`	Extract an element at a specific index
General traversal	

`traverse :: (Shape sh, Shape sh', Elt e)`
`=> Array sh e -> (sh -> sh') -> ((sh -> e) -> sh' -> e')`
`-> Array sh' e'`
	Unstructured traversal

`force :: (Shape sh, Elt e) => Array sh e -> Array sh e`	Force a delayed array into manifest form
`extent :: Array sh e -> sh`	Obtain size in all dimensions of an array

Table 1. Summary of array operations

The functions `index` and `fromIndex` are methods of the class `Shape` from Figure 3.

The functions `replicate` and `slice` were already discussed in Section 4.4, and `unit` and `(!:)` are defined as follows:

```
unit :: e -> Array Z e
unit = Delayed Z . const

(!:) :: (Shape sh, Elt e) => Array sh e -> sh -> e
arr !: ix = snd (delay arr) ix
```

The implementation of these two functions clearly shows that they do not depend on any methods of the `Shape` and `Elt` classes.

A simple operator to rearrange elements is the function `(+:+)`; it appends its second argument to the first and can be implemented with `traverse` by adjusting shapes and indexing.

In contrast, general shuffles operations, such as backwards permutation, require the detailed mapping of target to source indices. We have seen this in the example `transpose2D` in Section 2. Another example is the following function that extract the diagonal of a square matrix:

```
diagonal :: Elt e => Array DIM2 e -> Array DIM1 e
diagonal arr = assert (n == m) $
  backpermute n (\x -> (x, x)) arr
  where
    (n, m) = extent arr
```

Code that uses `backpermute` appears more like element-based array processing. However, it is still a collective operation with a clear parallel interpretation.

Backwards permutation is defined in terms of the general `traverse` as follows:

```
backpermute :: sh' -> (sh' -> sh) -> Array sh e
            -> Array sh' e
backpermute sh perm = traverse (const sh) (. perm)
```

The variant `backpermuteDft`, known as default backwards permutation, operates in a similar manner, except that the target index is partial. When the target index maps to `Nothing`, the corresponding element from the default array is used. Overall, `backpermuteDft` can be interpreted as a means to bulk update the contents of an array. As we are operating on purely functional, immutable arrays, the original array is stil available and the repeated use of `backpermuteDft` is only efficient if large part of the array are updated on each use.

5.4 General traversal

The most general form of array traversal is `traverse`, which supports an arbitrary change of shape and array contents. Nevertheless, it is still represented as a delayed computation as detailed in Section 3.3. Although for efficiency reasons it is better to use specific functions such as `map` or `backpermute`, it is always possible to fall back on `traverse` if a custom computational structure is required.

For example, `traverse` can be used to implement stencil-based relaxation methods, such as the following update function to solve the Laplace equation in a two dimensional grid [14]:

$$u'(i,j) = (u(i-1,j) + u(i+1,j) + u(i,j-1) + u(i,j+1))/4$$

To implement this stencil, we use `traverse` as follows:

```
stencil :: Array DIM2 Double -> Array DIM2 Double
stencil arr
  = traverse id update arr
  where
    _ :. height :. width = extent arr

    update get d@(sh :. i :. j)
     = if isBoundary i j
        then  get d
        else (get (sh :. (i-1) :. j)
          +   get (sh :. i     :. (j-1))
          +   get (sh :. (i+1) :. j)
          +   get (sh :. i     :. (j+1))) / 4

    isBoundary i j
      =  (i == 0) || (i >= width  - 1)
      || (j == 0) || (j >= height - 1)
```

As the shape of the result array is the same as the input, the first argument to `traverse` is `id`. The second argument is the update function that implements the stencil, while taking the grid boundary into account. The function `get`, passed as the first argument to `update`, is the lookup function for the input array.

To solve the Laplace equation we would set boundary conditions along the edges of the grid and then iterate `stencil` until the inner elements converge to their final values. However, for benchmarking purposes we simply iterate it a fixed number of times:

```
laplace :: Int
        -> Array DIM2 Double -> Array DIM2 Double
laplace steps arr = go steps arr
 where
  go s arr
    | s == 0    = arr
    | otherwise = go (s-1) (force $ stencil arr)
```

The use of `force` after each recursion is important, as it ensures that all updates are applied and that we produce a manifest array. Without it, we would accumulate a long chain of delayed computations with a rather non-local memory access pattern. In Repa, the function `force` triggers all computation, and as we will discuss in Section 7, the size of forced array determines the amount of parallelism in an algorithm.

6. Delayed arrays and loop fusion

We motivated the use of delayed arrays in Section 3.2 by the desire to avoid superfluous copying of array elements during index space transformation, such as in the definition of `backpermute`. However, another major benefit of delayed arrays is that it gives *by-default automatic loop fusion*. Recall the implementation of `map`:

```
map :: (a -> b) -> Array sh a -> Array sh b
map f = traverse id (f.)
```

and imagine evaluating `(map f (map g a))`. If you consult the definition of `traverse` (Section 3.3) it should be clear that the two `map`s simply build a delayed array whose indexing function first indexes `a`, then applies `g`, and then applies `f`. No intermediate arrays are allocated and, in effect, the two loops have been fused. Moreover, this fusion does not require some sophisticated compiler transformation, nor does it even require the two calls of `map` to be statically juxtaposed; fusion is a property of the data representation.

Guaranteed, automatic fusion sounds too good to be true — and so it is. The trouble is that we cannot *always* use the delayed representation for arrays. One reason not to delay arrays is data layout, as we discussed in Section 3.3. Another is parallelism: `force` triggers data-parallel execution (Section 7). But the most immediately pressing problem with the delayed representation is *sharing*. Consider the following:

```
let b = map f a
in  mmMult b b
```

Every access to an element of `b` will apply the (arbitrarily-expensive) function `f` to the corresponding element of `a`. It follows that *these arbitrarily-expensive computations will be done at least twice*, once for each argument of `mmMult`, quite contrary to the programmer's intent. Indeed, if `mmMult` itself consumes elements of its arguments in a non-linear way, accessing them more than once, the computation of `f` will be performed each time. If instead we say

```
let b = force (map f a)
in  mmMult b b
```

then the now-manifest array `b` ensures that `f` is called only once for each element of `a`. In effect, a manifest array is simply a memo table for a delayed array. Here is how we see the situation:

- In most array libraries, every array is *manifest by default*, so that sharing is guaranteed. However, loop fusion is difficult, and must often be done manually, doing considerable violence to the structure of the program.
- In Repa every array is *delayed by default*, so that fusion is guaranteed. However, sharing may be lost; it can be restored manually by adding calls to `force`. These calls do not affect the structure of the program.

Using `force` Repa allows the programmer tight control over some crucial aspects of the program: sharing, data layout, and parallelism. The cost is, of course, that the programmer must exercise that control to get good performance. Ignoring the issue altogether can be disastrous, becuase it can lead to arbitrary loss of sharing. In further work, beyond the scope of this paper, we are developing a compromise approach that offers guaranteed sharing with aggressive (but not guaranteed) fusion.

7. Parallelism

As described in Section 3.1, all elements of a Repa array are demanded simultaneously. This is the source of all parallelism in the library. In particular, an application of the function `force` triggers the parallel evaluation of a delayed array, producing a manifest one. Assuming that the array has n elements and that we have P parallel *processing elements (PEs)* available to perform the work, each PE is responsible for computing n/P consecutive elements in the row-major layout of the manifest array. In other words, the structure of parallelism is always determined by the layout and partitioning of a forced array.

Let us re-consider the function `mmMult` from Section 3.3 and Figure 2 in this light. We assume that `arr` is a manifest array, and know that `trr` is manifest because of the explicit use of `force`. The rank-2 array produced by the rank-generalised application of `sum` corresponds to the right face of the cuboid from Figure 2. Hence, if we `force` the result of `mmMult`, the degree of available parallelism is proportional to the number of elements of the resulting array — 8 in the figure. As long as the hardware provides a sufficient number of PEs, each of these elements may be computed in parallel. Each involves the element-wise multiplication of a row from `arr` with a row from `trr` and the summation of these products. If the hardware provides fewer PEs, which is usually the case, the evaluation is evenly distributed over the available PEs.

Let's now turn to a more sophisticated parallel algorithm, the three-dimensional fast Fourier transform (FFT). Three-dimensional FFT works on one axis at a time: we apply the one dimensional FFT to all vectors along one axis, then the second and then the third. Instead of writing a separate transform for each dimension, we implement one-dimensional FFT as a shape polymorphic function that operates on the innermost axis. We combine it with a three-dimensional rotation, `rotate3D`, which allows us to cover all three axes one after another:

```
ftt3D :: Array DIM3 Complex   -- roots of unity
      -> Array DIM3 Complex   -- data to transform
      -> Array DIM3 Complex
fft3d rofu = fftTrans . fftTrans . fftTrans
  where
   fftTrans = rotate3D . fft1D rofu
```

The first argument, `rofu`, is an array of complex roots of unity, which are constants that we wish to avoid recomputing for each call. The second is the three-dimensional array to transform, and we require both arrays to have the same shape. We also require each dimension to have a size which is a power of 2.

If the result of `fft3D` is `force`d, evaluation by P PEs is again on P consecutive segments of length n^3/P of the row-major layout of the transformed cube, where n is the side length of the cube. However, the work that needs to be performed for each of the elements is harder to characterise than for `mmMult`, as the computations of the individual elements of the result are not independent and as `fft1D` uses `force` internally.

Three-dimensional rotation is easily defined based on the function `backpermute` which we discussed previously:

```
rotate3D :: Array DIM3 Complex
            -> Array DIM3 Complex
rotate3D arr = backpermute (Z:.m:.k:.l) f
 where
   (Z:.k:.l:.m) = extent arr
   f (Z:.m:.k:.l) = (Z:.k:.l:.m)
```

The one-dimensional fast Fourier transform is significantly more involved: it requires us to recursively split the input vector in half and to apply the transform to the split vectors. To facilitate the splitting, we first define a function `halve` that drops half the elements of a vector, where the elements to pick of the original are determined by a selector function `sel`.

```
halve :: (sh:.Int -> sh:.Int)
      -> Array (sh:.Int) Complex
      -> Array (sh:.Int) Complex
halve sel arr =
  backpermute (sh :. n `div` 2) sel arr
  where
    sh:.n = extent arr
```

By virtue of rank generalisation, this shape polymorphic function will split all rows of a three-dimensional cube at once and in the same manner.

The following two convenience functions use `halve` to extract all elements in even and odd positions, respectively.

```
evenHalf, oddHalf :: Array (sh:.Int) Complex
                  -> Array (sh:.Int) Complex
evenHalf = halve (\(ix:.i) -> ix :. 2*i)
oddHalf  = halve (\(ix:.i) -> ix :. 2*i+1)
```

Now, the definition of the one-dimensional transform is a direct encoding of the Cooley-Tukey algorithm:

```
fft1D :: Array (sh:.Int) Complex
      -> Array (sh:.Int) Complex
      -> Array (sh:.Int) Complex
fft1D rofu v
 | n > 2  = (left +^ right) :+: (left -^ right)
 | n == 2 = traverse id swivel v
 where
  swivel f (ix:.0) = f (ix:.0) + f (ix:.1)
  swivel f (ix:.1) = f (ix:.0) - f (ix:.1)

  rofu' = evenHalf rofu
  left  = force . (*^ rofu) . fft1D rofu' . evenHalf $ v
  right = force .             fft1D rofu' . oddHalf  $ v

  (+^) = zipWith (+)
  (-^) = zipWith (-)
  (*^) = zipWith (*)
```

All index space transformations implemented in terms of `backpermute` and also the elementwise arithmetic based on `zipWith` produce delayed arrays. It is only the use of `force` in the definition of `left` and `right` that triggers the parallel evaluation of subcomputations. In particular, as we `force` the recursive calls in the definition of `left` and `right` separately, the recursive calls are performed in sequence. The rank-generalised input vector `v` is halved with each recursive call, and hence, the amount of available parallelism decreases.

However, keep in mind that —by virtue of rank generalisation— we perform the one-dimensional transform in parallel on all vectors of a cuboid. That is, if we apply `fft3D` to a $64 \times 64 \times 64$ cube, then `fft1D` still operates on $64 * 64 * 2 = 8192$ complex numbers in one parallel step at the base case, where $\mathtt{n} = 2$.

8. Benchmarks

In this section, we discuss the performance of three programs presented in this paper: matrix-matrix multiplication from Section 3.3, the Laplace solver from Section 5.4 and the fast Fourier transform from Section 7. We ran the benchmarks on two different machines:

- a 2x Quad-Core 3GHz Xeon server and
- 1.4GHz UltraSPARC T2.

The first machine is a typical x86-based server with good single-core performance but frequent bandwidth problems in memory-intensive applications. The bus architecture directly affects the scalability of some of our benchmarks, e.g., the Laplace solver, which cannot utilise multiple cores well due to bandwidth limitations.

The SPARC-based machine is much more interesting, as the T2 processor has 8 cores and supports up to 8 hardware threads per core which allows it to effectively hide memory latency in massively multithreaded programs. Thus, despite a significantly worse single-core performance than the Xeon it exhibits much better scalability which is clearly visible in our benchmarks.

8.1 Absolute performance

Before discussing the parallel behaviour of our benchmarks, let us investigate how Repa programs compare to hand-written C code when executed with only one thread. Figure 5 shows the results for the Laplace solver and the matrix multiplication together with the fastest running times obtained through parallel execution (we do

		C	Repa 1 thread	Repa fastest parallel
Laplace	400×400	10.3s	15.8s	6.6s
Matrix mult	1024×1024	7.8s	8.2s	1.1s
FFT	128×128×128		9.3s	2.3s

Figure 5. Performance on the Xeon

		C	Repa 1 thread	Repa fastest parallel
Laplace	300×300	6.1s	24.9s	3.2s
Matrix mult	512×512	3.1s	10.2s	0.4s
FFT	64×64×64		9.8s	1.2s

Figure 6. Performance on the SPARC

not have a C version of FFT but provide the running times of the Repa program). On the Xeon, Repa is slower than C when executed sequentially but not by much.

The picture changes dramatically on the SPARC, however (Figure 6). Unfortunately, GHC generates very poor code for SPARC architectures, making the performance difference between Repa programs and the corresponding C versions significanlty worse than on the Xeon. This is a temporary problem, however, as we expect the new LLVM backend [10, 21] to produce much faster code. Unfortunately, it has not been ported to the SPARC yet and sometimes generates incorrect code even on x86. We have only been able to use it for the Laplace benchmark on the Xeon machine but fully expect this situation to improve in the near future. In any case, when run on multiple threads the benchmarks are still able to achieve better performance than the sequential C programs.

We also compared the performance of the Laplace solver to an alternative, purely sequential Haskell implementation based on unboxed, mutable arrays running in the IO Monad (`IOUArray`). This version was about two times slower than the Repa program, probably due to the overhead introduced by bounds checking, which is currently not supported by our library. Note, however, that bounds checking is unnecessary for many collective operations such as `map` and `sum`, so even after we introduce it in Repa we still expect to see better performance than a low-level, imperative implementation based on mutable arrays.

8.2 Parallel behaviour

The parallel performance of matrix multiplication is show in Figure 7. Here, we get excellent scalability on both machines. On the Xeon, where we achieve 7.7 with 8 threads the program is able to avoid bandwidth problems, probably by utilising the cache efficiently. On the SPARC, it scales with up to 31 threads with a peak speedup of 22.9.

Figure 8 shows the relative speedups for the Laplace solver. The program achieves good scalability on the SPARC, reaching a speedup of 7.8 with 15 threads but performs much worse on the Xeon, stagnating at a speedup of 2.6. As the benchmark is memory bound, we attribute this behaviour to the insufficient bandwidth of the Xeon machine.

Finally, the parallel behaviour of the FFT implementation is shown in Figure 9. This program scales well on both machines, achieving a relative speedup of 4 on with 8 threads on the Xeon and 8.3 on 16 threads on the SPARC. Compared to the Laplace solver, the scalability is much better on the former but practically unchanged on the latter. As this benchmark is less memory intensive, this supports our conclusion that the Laplace solver suffers

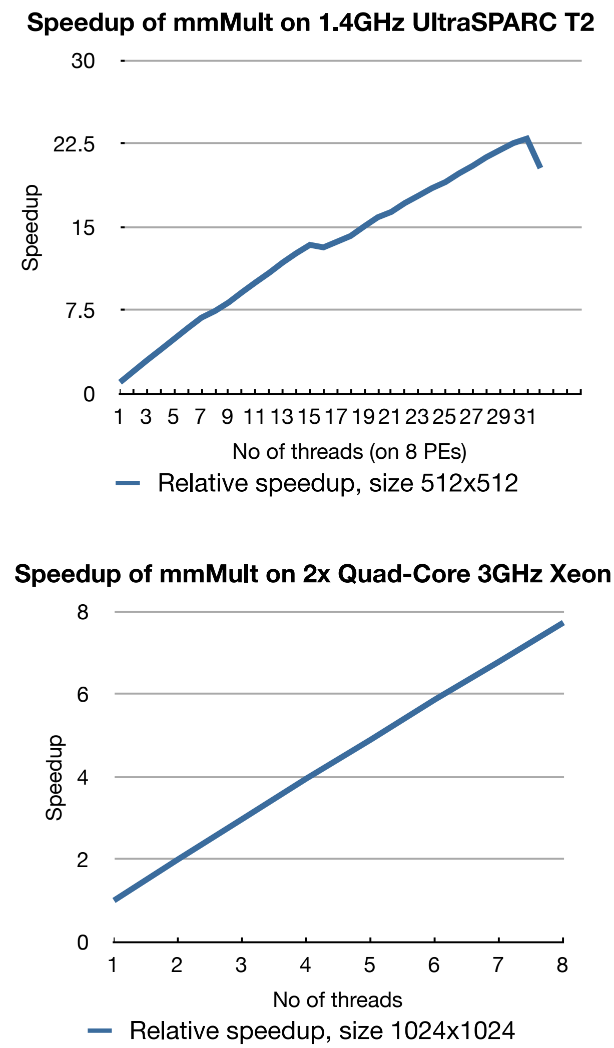

Figure 7. Matrix-matrix multiplication

from bandwidth problems on the Xeon machine while the SPARC is able to execute the two program equally well by utilising its very fast hardware threads.

9. Related Work

Array programming is a highly active research area so the amount of related work is quite significant. In this section, we have to restrict ourselves to discussing only a few most closely related approaches.

9.1 Haskell array libraries

Haskell 98 already defines an array type as part of its prelude which, in fact, even provides a certain degree of shape polymorphism. These arrays can be indexed by arbitrary types as long as they are instances of `Ix`, a type class which plays a similar role to our `Shape`. This allows for fully shape-polymorphic functions such as `map`. However, standard Haskell arrays do not support at-least constraints and rank generalisation which are crucial for implementing highly expressive operations such as `sum` from Section 4.3. This inflexibility precludes many advanced uses of shape polymor-

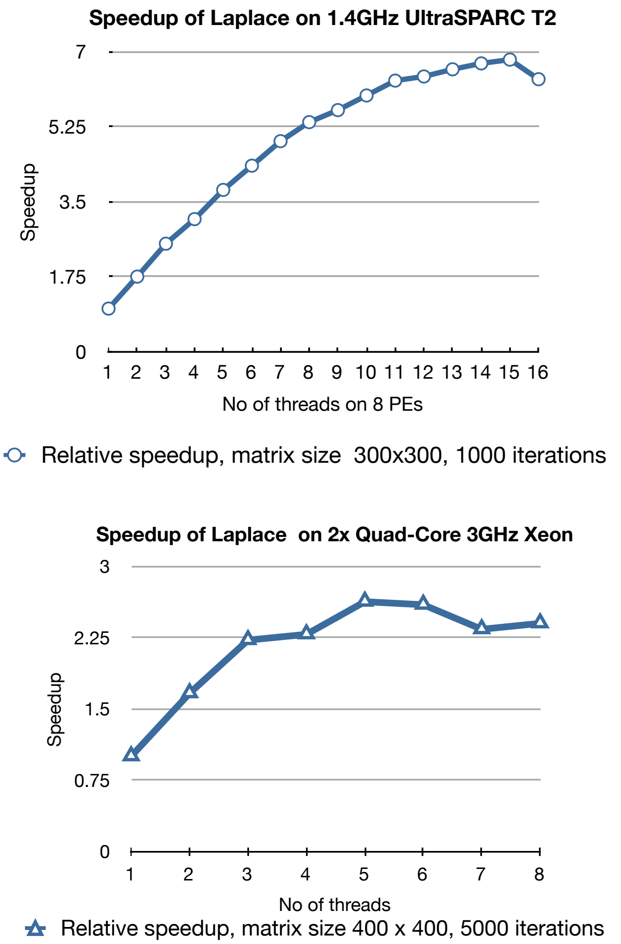

Figure 8. Laplace solver

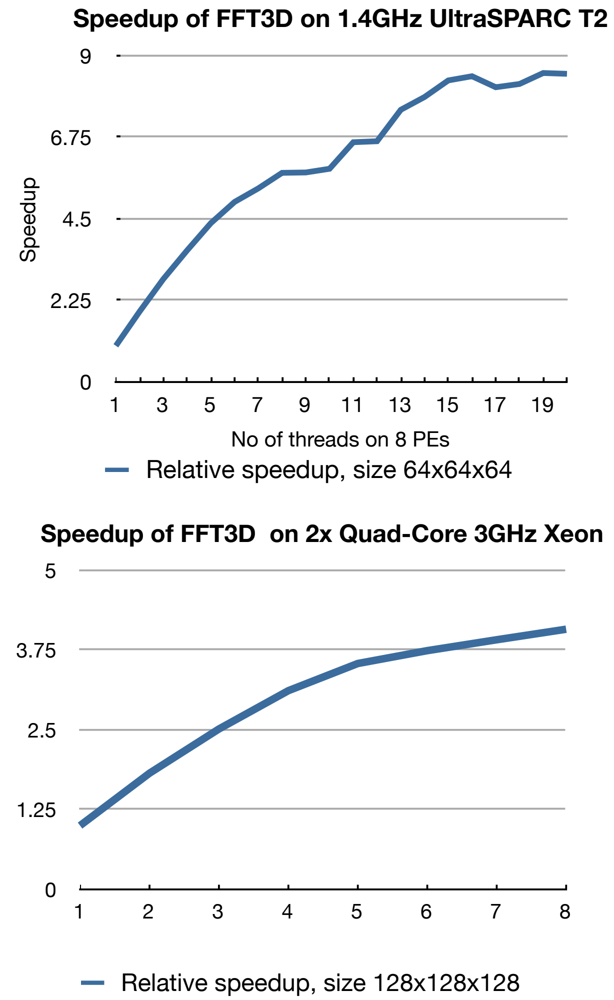

Figure 9. Fast Fourier transform

phism described in this paper and makes even unboxed arrays based on the same interface a bad choice for a parallel implementation.

Partly motivated by the shortcomings of standard arrays, numerous Haskell array libraries have been proposed in recent years. These range from highly specialised ones such as ByteString [7] to full-fledged DSLs for programming GPUs [12]. However, these libraries do not provide the same degree of flexibility and efficiency for manipulating regular arrays if they support them at all. Our own work on Data Parallel Haskell is of particular relevance in this context as the work presented in this paper shares many of its ideas and large parts of its implementation with that project. Indeed, Repa can be seen as complementary to DPH. Both provide a way of writing high-performance parallel programs but DPH supports irregular, arbitrarily nested parallelism which requires it to sacrifice performance when it comes to purely regular computations. One of the goals of this paper is to plug that hole. Eventually, we intend to integrate Repa into DPH, providing efficient support for both regular and irregular arrays in one powerful framework.

9.2 C++ Array Libraries

Due to its powerful type system and its wide-spread use in high-performance computing, C++ has a significant number of array libraries that are both fast and generic. In particular, Blitz++ [23] and `Boost.MultiArray` [1] feature multidimensional arrays with a restricted form of shape polymorphism. However, our library is much more flexible in this regard and also has the advantage of a natural parallel implementation which neither of the two C++ libraries provide. Moreover, these approaches are inherently imperative while we provide a purely functional interface which allows programs to be written at a much higher level of abstraction.

9.3 Array Languages

In addition to libraries, there exist a number of special-purpose array programming languages. Of these, Single Assignment C (SAC) [18] has exerted the most influence on our work and is the closest in spirit as it is purely functional and strongly typed. SAC provides many of the same benefits as Repa: high-performance arrays with shape polymorphism, expressive collective operations and extensive optimisation. In addition to a rich standard library, the basic building blocks of SAC programs are *with-loops*, a special-purpose language construct for constructing, traversing and reducing arrays. With-loops allow array programs to be written at a high level of abstraction and are also amenable to aggressive loop fusion which is crucial for achieving good performance.

While providing a similar level of expressiveness and performance, Repa also has the significant advantage of being integrated into a mainstream functional language and not requiring specific compiler support. This allows Repa programs to utilise the rich Haskell infrastructure and to drop down to a very low level of abstraction if required in specific cases. This, along with strong typing and purity, are also the advantages Repa has over other array languages such as APL, J and Matlab [2, 8, 22].

Acknowledgements. We are grateful to Arvind for explaining the importance of delaying index space transformations and thank Simon Winwood for comments on a draft. This research was funded in part by the Australian Research Council under grant number LP0989507.

References

[1] The Boost Multidimensional Array Library, April 2010. URL `http://www.boost.org/doc/libs/1_42_0/libs/multi_array/doc/user.html`.

[2] C. Burke. *J and APL*. Iverson Software Inc., 1996.

[3] M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ICFP '05: Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming*, pages 241–253, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-064-7. doi: http://doi.acm.org/10.1145/1086365.1086397.

[4] M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Sysposium on Principles of Programming Languages*, pages 1–13. ACM Press, 2005. ISBN 1-58113-830-X. doi: http://doi.acm.org/10.1145/1040305.1040306.

[5] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a status report. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM Press, 2007.

[6] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*. ACM Press, 2007.

[7] D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting haskell strings. In *Practical Aspects of Declarative Languages 8th International Symposium, PADL 2007*, pages 50–64. Springer-Verlag, Jan. 2007.

[8] A. Gilat. *MATLAB: An Introduction with Applications 2nd Edition*. John Wiley & Sons, 2004. ISBN 978-0-471-69420-5.

[9] J. H. v. Groningen. The implementation and efficiency of arrays in Clean 1.1. In W. Kluge, editor, *Proceedings of Implementation of Functional Languages, 8th International Workshop, IFL '96, Selected Papers*, number 1268 in LNCS, pages 105–124. Springer-Verlag, 1997.

[10] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[11] J. Launchbury and S. Peyton Jones. Lazy functional state threads. In *Proceedings of Programming Language Design and Implementation (PLDI 1994)*, pages 24–35, New York, NY, USA, 1994. ACM.

[12] S. Lee, M. M. T. Chakravarty, V. Grover, and G. Keller. GPU kernels as data-parallel array computations in haskell. In *EPAHM 2009: Workshop on Exploiting Parallelism using GPUs and other Hardware-Assisted Methods*, 2009.

[13] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.11, documentation and user's manual. Technical report, INRIA, 2008.

[14] J. Mathews and K. Fink. *Numerical Methods using MATLAB, 3rd edition*. Prentice Hall, 1999.

[15] S. Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proceedings of the International Conference on Functional Programming (ICFP 2007)*, pages 327 –337, 1997.

[16] S. Peyton Jones, R. Leshchinskiy, G. Keller, and M. M. T. Chakravarty. Harnessing the multicores: Nested data parallelism in Haskell. In R. Hariharan, M. Mukund, and V. Vinay, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2008)*, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. URL `http://drops.dagstuhl.de/opus/volltexte/2008/1769`.

[17] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2003.

[18] S.-B. Scholz. Single assignment C – efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[19] T. Schrijvers, S. Peyton-Jones, M. M. T. Chakravarty, and M. Sulzmann. Type checking with open type functions. In *Proceedings of ICFP 2008 : The 13th ACM SIGPLAN International Conference on Functional Programming*, pages 51–62. ACM Press, 2008.

[20] W. Swierstra and T. Altenkirch. Dependent types for distributed arrays. In *Trends in Functional Programming*, volume 9, 2008.

[21] D. A. Terei. Low Level Virtual Machine for the Glasgow Haskell Compiler, 2009.

[22] The International Standards Organisation. *Programming Language APL*. ISO standard 8485, 1989.

[23] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object Oriented Parallel Environments (ISCOPE'98)*. Springer-Verlag, 1998. ISBN 978-3-540-65387-5.

[24] H. Xi. Dependent ML: an approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007.