Efficient Parallel Stencil Convolution in Haskell

Ben Lippmeier  Gabriele Keller
School of Computer Science and Engineering
University of New South Wales, Australia
{benl, keller}@cse.unsw.edu.au

Abstract

Stencil convolution is a fundamental building block of many scientific and image processing algorithms. We present a declarative approach to writing such convolutions in Haskell that is both efficient at runtime and implicitly parallel. To achieve this we extend our prior work on the Repa array library with two new features: partitioned and cursored arrays. Combined with careful management of the interaction between GHC and its back-end code generator LLVM, we achieve performance comparable to the standard OpenCV library.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Concurrent programming structures; Polymorphism; Abstract data types

General Terms Languages, Performance

Keywords Arrays, Data parallelism, Haskell

1. Introduction

This paper addresses an implicit challenge put to us by Rami Mukhtar of NICTA (the Australian equivalent of INRIA). At the time, Rami was starting a project on writing image processing algorithms in declarative languages. Having read our previous work on the Repa library for parallel arrays [15], he took it to heart, and promptly implemented the Canny edge detection algorithm [6] as a benchmark. Unfortunately, he then informed us that the Repa version was more than 10x slower than the equivalent implementation in OpenCV [5], an industry standard library of computer vision algorithms. Due to this, he instead based his project around the Accelerate EDSL [8] for writing parallel array codes on GPGPUs, produced by a competing (but friendly) faction in our research group. Clearly, we could not let this stand.

Simply put, our aim is to implement parallel image processing algorithms that run as fast (faster!) than the highly optimised ones for imperative languages. We also want to write this code directly in Haskell and use the GHC runtime system, instead of, say, implementing an EDSL that produces LLVM or CUDA code. Using Haskell directly gives us access to GHC’s powerful inliner and simplifier, which we use to convert declarative code into the tight loops we rely on for performance. The GHC runtime provides the primitives we use to implement parallelism in a portable way.

At the core of many image processing algorithms is the 2D convolution operator *, whose definition is as follows:

\[(A * K)(x, y) = \sum_i \sum_j A(x+i, y+j) K(i, j)\]

Here, A is the image being processed, and K is the convolution kernel or stencil. The stencil is a small matrix, with typical dimensions 3x3 or 1x5, that defines a transformation on the image. Typical transformations include the Gaussian blur, and the Sobel differentiation operator, both of which are used in the Canny edge detection algorithm. For this paper we focus on the efficient parallel implementation of stencil convolution, though we will return to the larger Canny algorithm near the end. As we are primarily interested in image processing we also focus on arrays of rank 2, though our techniques are equally applicable to arrays of higher rank.

Our contributions are as follows:

- An array fusion approach to writing stencil functions in Haskell that yields performance comparable to the industry standard OpenCV library.
- To achieve this we extend our previous approach [15] with two new features: partitioned and cursored arrays. These features allow us to optimise array programs that use different functions to construct the various regions of the array, and to share subcomputations of adjacent elements.
- A declarative API that allows us to write cache-friendly programs that access data in a block-wise manner, while cleanly separating the evaluation code from the specification of the array elements.
- As array fusion is sometimes perceived as “brittle” due to its dependency on poorly understood code transformations, we seek to mitigate this problem by summarising the main details that must be accounted for to repeatedly generate efficient object programs. This includes the staging of inliner phases, and the interaction between GHC and its back-end code generator, LLVM.
- Finally, with the ultimate aim of writing declarative code that runs as fast as competing libraries, we discuss the current challenges to array fusion and suggest directions for future research.

The Ypnos [22] and PASTHA [18] libraries also address stencil convolution in Haskell, though [22] presents no performance figures and [18] lacks absolute numbers. On the other hand, Ypnos deals elegantly with arrays of higher rank, and PASTHA also has support for managing convergence conditions for iterative convolution, which we don’t address here. Our philosophy of array programming is also shared by the Chapel language [3], in that the value of an array should be defined declaratively, using bulk operations. This specification is then mapped onto physical processors in separate, orthogonal code.

[Copyright notice will appear here once ‘preprint’ option is removed.]
2. The Laplace Equation, Reloaded

Although we have found the general principle of Repa’s array representation to work well, when applied to the problem of stencil convolution we now have enough experience with it to point out several infelicities. We will reuse our example from [15] of the numerical solution of the Laplace equation. The overall structure of this example is similar to the code in the original Canny implementation which we are trying to improve.

The \texttt{solveLaplace} function in Figure 1 solves the Laplace equation \( \nabla^2 u = 0 \) in a 2D grid, with constant value boundary conditions. Numerically, the equation can be solved by applying the following update function to every point in a grid until we reach a fixed point:

\[
  u'(i, j) = \frac{u(i-1, j) + u(i+1, j) + u(i, j-1) + u(i, j+1)}{4}
\]

This process has the same effect as convolving the input image with the Laplace stencil shown in Figure 3, and then dividing every element in the result by four. Although in practice we would iterate the above function to a fixed point, for benchmarking we simply iterate it a fixed number of times, hence the \texttt{steps} parameter to \texttt{solveLaplace}. The boundary conditions are specified by two arrays, \texttt{arrBoundValue} and \texttt{arrBoundMask}. The first gives the value to use at a particular point, while the second contains 0 where the boundary applies and 1 otherwise. If we are too close to the border of the array to apply the update function, then we return the original value. The \texttt{traverse} function used in \texttt{relaxLaplace} produces a new array by calling \texttt{elemFn} for every index in the result. The \texttt{elemFn} worker is passed a lookup function \texttt{get}, which it can use to get values from the source array. The type of \texttt{traverse} is given in the same figure. The expression \((Z :. i :. j)\) is an array index to row \(i\) and column \(j\). See [15] for further details.

Although \texttt{solveLaplace} gives the correct answer, it has several runtime performance problems:

1. We test for the border at every iteration (the call to \texttt{isBorder} in \texttt{elemFn}), even though in the vast majority of iterations we are far from it. We will discuss border handling further in §4.1.
2. Every lookup of the source array must be bounds checked by the library implementation. Concretely, the user-defined \texttt{elemFn} might apply \texttt{get} to an out-of-bounds index (if, say, \texttt{isBorder} was not implemented correctly), so get must conservatively check bounds before indexing the array.
3. As potentially arbitrary array indices could be passed to \texttt{get}, the library performs computations of the form \(x + y * \text{width}\) to gain the flat indices into the underlying buffer. However, in Figure 1 the flat indices needed by \texttt{get} could be computed by simple addition once the flat index of the center point is known.

We will return to these problems in later sections, but for now note that the bounds checking overhead is the easiest to mitigate, as we can simply disable it. Replacing the use of \((!)\) in the definition of \texttt{traverse} with an “unsafe” indexing operator removes the overhead, but this is clearly unsatisfying. Far better would be to write the code so that it is correct by construction. Nevertheless, in Figure 2 we present part of GHC’s Core Intermediate Representation (IR) for the inner loop of an unsafe version of our \texttt{solveLaplace} function. This is the code resulting from array fusion, after GHC has unfolded all of the library functions, inlined the user defined functions into them, and performed a large number of code transformations. The presented code loads the surrounding elements from the source array, applies the stencil kernel and boundary conditions, and updates the destination. The actual loop construct is defined in the library, as a part of the \texttt{force} function used in \texttt{solveLaplace}.

In the Core IR, infix operators like \(+\#\) and \(*\#\) (one hash) work on integers, while operators like \(+##\) and \(*##\) (two hashes) work on floats.\footnote{In GHC proper, \(*##\) and \(*###\) actually work on doubles, but we’re using them for floats for clarity.} Hashes imply that these operators work on native, unboxed values. There is no overhead due to boxing, unboxing, or laziness, and each unboxed operator essentially corresponds to a single machine operation. The fact that our (unsafe) inner loop is already so “clean” gives us heart that we may reach the proverbial “C-like performance”. Of course, it would be better if the code was fast and safe, instead of just fast.

3. Delayed Arrays in Repa

In this section we give a quick summary of Repa’s original array representation, which we will improve over in the next. The main features of Repa are:

- \textit{shape polymorphism:} functions can be written that operate on arrays of arbitrary rank.
- \textit{implicit data parallelism:} functions written with Repa can be run in parallel without any extra work by the programmer.
- \textit{array fusion:} we write array functions in a compositional style, using “virtual” intermediate arrays, but the need to actually create the intermediate arrays is eliminated during compilation.
force

A delayed array is represented by an unboxed array provided by the
manifest array is one represented by real data, that is,
type used for the indices, and
map
which we will extend in
shape polymorphism is not of particular help so we will not con-
consider it further. What of interest is parallelism and fusion. Repa
achieves this by using the following representation for arrays,
which we will extend in §4.1.
data Array sh a
  = Manifest (Vector a)
  | Delayed (sh -> a)

Our array type is polymorphic over sh (shape), which is the
type used for the indices, and a, which is the type of the elements
contained. A manifest array is one represented by real data, that is,
held in flat unboxed array provided by the Data.Vector library.
A delayed array is represented by an element function that takes
an array index and produces the corresponding element. Delayed
arrays are the key to Repa’s approach to array fusion. For example,
the map function for arrays is defined as follows:
{-# INLINE map #-} map :: (Shape sh, Elt a, Elt b)
  => (a -> b) -> Array sh a -> Array sh b
map f arr
  = case arr of
    Manifest vec -> Delayed (f . (vec !))
    Delayed g -> Delayed (f . g)

Here, Shape is the class of types that can be used as indices,
and Elt the class of types that can be used as array elements. Both
cases of map produce a Delayed array, and the second corresponds
to the following familiar identity:

    map f (map g xs) = map (f . g) xs

Similar traversal functions such as zipWith are defined in the
same way. We also support reductions such as sum and foldl, but
do not support general filtering operations as the resulting array
is not necessarily rectangular. Fusion is achieved via the judicious
use of INLINE pragmas, and the magic of the GHC simplifier.
During compilation, the outer structure of functions such as map is
eliminated, leaving code that applies the worker function directly
to each element of the array. Parallelism is introduced by using the
force function:

  force :: (Shape sh, Elt a)
  => Array sh a -> Array sh a

For Manifest arrays, force is the identity. For Delayed
delayed array, and the second corresponds
to the following familiar identity:

    map f (map g xs) = map (f . g) xs

Similar traversal functions such as zipWith are defined in the
same way. We also support reductions such as sum and foldl, but
do not support general filtering operations as the resulting array
is not necessarily rectangular. Fusion is achieved via the judicious
use of INLINE pragmas, and the magic of the GHC simplifier.
During compilation, the outer structure of functions such as map is
eliminated, leaving code that applies the worker function directly
to each element of the array. Parallelism is introduced by using the
force function:

  force :: (Shape sh, Elt a)
  => Array sh a -> Array sh a

For Manifest arrays, force is the identity. For Delayed
delayed array, and the second corresponds
to the following familiar identity:

    map f (map g xs) = map (f . g) xs

Several common stencils are used in Figure 3. For stencil names
written with subscripts, the subscript indicates that it is just one
member of a closely related family of stencils. For example, SobelX
differentiates along the X axis only, but rotating it 90 degrees
eysields SobelY which differentiates along the Y axis. By “rotate”
we mean to permute the coefficients of the matrix, so that +1 is in
the top-left in this example. The SobelXY stencils are used in
Canny edge detection, while RobertsX and KirschY also perform
differentiation. The PeakPoint stencil is used for noise
detection, HighPass is a high-pass filter, and BinomialXY is a low-
pass filter. The Laplace stencil is used to compute the average ofour surrounding pixels, which we discussed in §2. How these
stencils are derived is not important to the discussion, but see
[21] for a nice introduction to stencil convolution and other image
processing algorithms. For the example stencils, we note several
features of computational interest, along with exceptions:

1. All coefficients are statically known.
2. Most coefficients are small integers.
3. Many coefficients are zero.
4. All stencils are symmetric.

Figure 3. Common convolution stencils

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SobelX</td>
<td>RobertsX</td>
<td>KirschY</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>+1</td>
</tr>
<tr>
<td>-2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. Old core IR for solveLaplace using unsafe indexing

In this paper, as we are dealing with stencils of a fixed rank,
shape polymorphism is not of particular help so we will not con-
sider it further. What of interest is parallelism and fusion. Repa
achieves this by using the following representation for arrays,
which we will extend in §4.1.

In this paper, as we are dealing with stencils of a fixed rank,
shape polymorphism is not of particular help so we will not con-
sider it further. What of interest is parallelism and fusion. Repa
achieves this by using the following representation for arrays,
which we will extend in §4.1.

In this paper, as we are dealing with stencils of a fixed rank,
shape polymorphism is not of particular help so we will not con-
sider it further. What of interest is parallelism and fusion. Repa
achieves this by using the following representation for arrays,
which we will extend in §4.1.
Points 1 and 2 suggest that we can specialise our stencil functions based on the values of the coefficients. For example, multiplication by two can be achieved by addition, and multiplication by one is no-op. This is opposed to, say, writing a general purpose function that reads coefficients from an array, and performs all multiplications explicitly. Points 3, 4 and 5 suggest that there are savings to be had by common sub-expression and dead-code elimination. Point 6 suggests that being able to handle stencils smaller than a certain fixed size would allow us to support most of the common cases. Points 7 and 8 have implications for border handling, which we discuss in the next section.

4.1 Partitioned Arrays

When implementing convolution, an immediate concern is what to do when the stencil “falls off” the edge of the array. For example, Figure 4 shows the application of a 3x3 stencil in this circumstance. The white squares indicate the internal region, where the stencil is entirely within the array. The grey squares indicate the border, where part of the stencil falls outside. There are several ways of handling the border case, with two popular options being to return a constant value (like zero) for out-of-bounds elements, or to return the same value as the nearest in-bounds element.

With the array sizes commonly encountered during image processing, only a tiny fraction of the elements are in the border region. This fact implies that for optimal performance, we should avoid testing for the border each time we compute an element. To achieve this, we represent the partitioning of the array into various regions directly. Partitioning allows us to define the result array using element functions specialised to each region, and guarantee that the one producing the internal elements is not applied in the border region. In effect, partitioning the array allows us to lift the if-expression that tests for the border out of the main loop of our program, and have the library code construct the border and internal regions separately. With partitioned arrays, it does not matter if the element function for the border takes a little longer to evaluate than the one for the internal region, as the former is only applied a small number of times. Provided the simpler, internal case is well optimised, we will still get good overall performance.

Our new data types are shown in Figure 5. An Array is defined as an extent, and a list of distinct Regions. In the rank-2 (two-dimensional) case the extent will represent the width and height of the array. Each region has a Range that defines the set of indices belonging to the region. A Range can either be RangeAll, which indicates the entire array, or a RangeRects which gives a list of rectangles (of arbitrary rank). Given a RangeRects, we can determine whether a particular index is inside the range either by checking whether it falls in any of the Rects, or using the predicate rangeMatch. This predicate gives the same result as the former, but can use a more efficient implementation than checking each Rect individually. In general, for “local” array operations such as indexing a single element, we use the predicate to quickly determine which region the provided index is in. In contrast, rangeRects is used when forcing the entire array, and allows us to create a loop specialised to each region.

Each Region also has a Generator that encodes how the array elements in that region should be computed. As before, generators of Manifest arrays are just flat vectors of unboxed values that hold the elements in row-major order. Delayed arrays are now represented in cursorred form. The cursorred representation allows us to share indexing computations when forcing adjacent array elements, which is discussed further in §5. The regions of a partitioned array must provide full coverage, meaning that every array element must be within some region. Regions are permitted to overlap, with the first one in the list taking precedence. Using overlapping allows us to define a default value for array elements with a RangeAll, while carving out specific areas with a RangeRects earlier in the list.

In general, partitioning an array allows us to generate loops specialised to each region. Specialisation can occur on both a per-element and per-region basis. An example of the first is the optimisation of border handling, which we discussed earlier. An example of the second is to use different loop code to evaluate regions of different sizes. For example, when evaluating a region that is short and wide it is best to operate in a row-wise manner, computing an entire row before moving to the next. This helps to recover sharing between horizontally adjacent elements. In contrast, to evaluate a region that is tall but thin it is best to operate column-wise, to exploit sharing between vertically adjacent elements. As the Region type provides a direct description of the size of each region, we can specialise the library code based on this information. The user invokes the appropriate specialisation automatically with each application of force. We discuss specialisation further in §5.3.

4.2 Bounds Checking and Co-Stencils

Firstly, with respect to bounds checking, we sheepishly admit that the old version of Repa didn’t actually do any. This issue was mentioned in [15]. As such, it was possible for buggy client programs to crash at runtime. The trouble is that bounds checking each array access adds a substantial overhead, and the comparison and branching constructs involved interfere with fusion. We tried adding it, by
having the Data.Vector library check each of the indexing operations on the underlying manifest array, but this resulted in a 35% slowdown for our Laplace example applied to a 300x300 array.

Ultimately, the problem is that client code written by the user of a library is “untrusted”, meaning that the library must assume it will index out-of-bounds elements. With respect to the code in Figure 1, without a more “heavy weight” technology like dependent types, or some clever analysis, the compiler cannot prove that when the predicate inBorder succeeds, the indexing operations in the else branch of elemFn are guaranteed to be valid. This problem is compounded by the fact that to support shape polymorphism we must check indices of arbitrary rank against the array bounds. Failing that we could check the linear indexing of the underlying manifest vector, but we would still need to manage the mapping between these indices and the original indices of arbitrary rank.

Our solution to this problem is to invert the relationship between the stencil definition (elemFn) and the source array. Instead of having the (untrusted) elemFn fetch elements from the source array itself, we instead write the client code to combine source elements led to it by the (trusted) library. This distinction is similar to the one between recursive and co-recursive functions in stream fusion [10], where the latter is the “co-stencil” case. Related work on Ypnos [22] mentions the co-monadic structure of grid computations, but does not discuss the relationship with bounds checking.

Figure 6 gives the data type that represents stencils, while Figure 7 contains our new implementation of solveLaplace. Figure 6 also gives the definition of makeStencil which is a utility function defined by our library. The type Stencil sh a specifies a stencil function of shape sh that operates on arrays of element type a. It consists of a size such as Z :. 3 :. 3 for the 3x3 case, as well as a zero value and accumulator function which define a fold operation over array elements. Figure 7 shows how to define the Laplace stencil. The iterateBlockwise function repeatedly applies its parameter function to an array, forcing it after each iteration. In this and latter code we have elided INLINE pragmas, as well as the Shape and Elt type class constraints to save space. We have also elided explicit matches against the input arrays arrBoundMask, arrBoundValue and arrInit that require them to be manifest. These matches are needed in our concrete implementation for performance reasons, but we hope to improve the compiler so they are not required in future. This is discussed further in §7.4.

The lambda abstraction in the definition of laplace defines the coefficient function for the Laplace stencil. The coefficient function gives the coefficients for each position in the stencil, and has type (sh -> Maybe a). It gives the coefficient at a particular offset from the focus of the stencil, or if that coefficient is zero it returns Nothing instead. Handling of zeros is discussed further in the next section. As a syntactic convenience, our library also provides some Template Haskell code to make listing the coefficients easier. An example of this syntax is in the niceLaplace function of Figure 7.

The operation of computing the sum-of-products of array elements and stencil coefficients is defined by the Just case of makeStencil. We could have embedded the coefficient function directly in the definition of Stencil, but instead define stencils in terms of a more general fold operation. Using a fold leaves the door open for other stencil-like operations that are not expressed as a sum-of-products, such as the creation of a histogram of the neighbourhood of each pixel.

Returning to the issue of bounds checking, with our new definitions, client code does not need direct access to the source array at all. All of the library functions used in Figure 7 operate on the whole array at a time, and their safety depends on the correctness of the library, instead of the correctness of the client.

Finally, we note that in virtually all related work using imperative languages it is simply assumed that bounds checking is not performed. The focus of recent papers such as [11] and [16] is usually on optimising cache usage, and they presume the existence of correct, heavily optimised straight line code for computing the individual array elements. In contrast, we are trying to produce a (safe!) general purpose functional array library, which also has support for efficient stencil convolution.

4.3 Zeros in Stencil Definitions

Although the stencils we use often contain zero-valued coefficients, we want to avoid wasting cycles performing the corresponding multiplications, as they do not contribute to the final sum of products. The simple, neat and wrong solution is to allow terms of the form 0 * x in the intermediate code, but then add a GHC rewrite rule [23] to implement the obvious identities 0 * x ≡ 0 and x + 0 ≡ x. Unfortunately, the first one of these is invalid for standard IEEE-704 floating point numbers because the operation 0 * ∞ is supposed to produce NaN (Not a Number). Although this hardly matters for image processing, we still don’t want to add the rule as it would apply globally and we risk breaking other code. Instead, we define the coefficient function to return Nothing where the stencil does not apply, and use this to skip over the associated term in makeStencil. Nevertheless, in the literature, stencils are usually specified using zeros. Due to this we allow zeros in our Template Haskell version, but eliminate them while desugaring to the coefficient function.

5. Sharing and Cursored Arrays

Suppose we apply a 3x3 stencil to a single internal point in an image, and that every coefficient is non-zero. At the least, this
so we are not home free yet. However, GHC can now compile via
fortunately, no current Haskell compiler implements this transform,
(GVN) [2, 25] transformation performed by some compilers. Un-
piler optimisation and is the target of the Global Value Numbering
potential savings for index computations are even greater.
compute four elements of the result array in a single operation, the
the others by adding +1/-1 or +width/-width. In the case where we
pute the index of the focus (center) of the stencil, and then get to
the coordinates of the element of interest. However, as the spacial
with the familiar expression
\[ x + y \times \text{width} \]
production of
tion of
18 loads, as well as the four stores to the result. There is also
the potential to share the evaluation of array indices, and well as
multiplications, depending on the form of the stencil.
The potential for sharing indexing computations can be seen in
Figure 2 which shows the Core IR for part of the inner loop of
our Laplace example. Although this code only computes a single
point in the result, note that the second argument to each application
of indexFloatArray\# produces the offset into the array for
each point in the stencil. Computation of these offsets is performed
with the familiar expression \( x + y \times \text{width} \), where \( x \) and \( y \)
are the coordinates of the element of interest. However, as the spacial
relationship between the elements is fixed, we could instead com-
pute the index of the focus (center) of the stencil, and then get to
the others by adding \(+1/-1\) or \(+\text{width}/-\text{width}\). In the case where we
compute four elements of the result array in a single operation, the
potential savings for index computations are even greater.

Recovering this sort of sharing is a well known problem in com-
piler optimisation and is the target of the Global Value Numbering
(GVN) [2, 25] transformation performed by some compilers. Un-
Fortunately, no current Haskell compiler implements this transform,
so we are not home free yet. However, GHC can now compile via
LLVM [27], and LLVM does implement a GVN pass. Provided we
expose enough of the internal indexing computations, the LLVM
compiler will do the rest for us. This brings us to cursored arrays.

5.1 Cursored Arrays

Recall the new Repa array representation from Figure 5. The defi-
nition of element generators is repeated below for reference.

data Generator sh a
  = GenManifest { genVector :: Vector a }
  | forall cursor.
  GenCursored { genMake :: sh -> cursor
    , genShift :: sh -> cursor -> cursor
    , genLoad :: cursor -> a }

A cursor is an abstract representation of an index into the array.
The specific form of the cursor is defined by the producer of
the array, while the consumer must use the provided cursor functions
to access elements. As hinted in the previous section, for stencil
functions we represent the cursor by a linear index into the array.
Given the coordinates of an element, genMake computes the linear
index of that element, the genShift function shifts a cursor by an
offset, and genLoad produces the array element for a given cursor.
Using cursors allows us to avoid repeated indexing computations
like \( x + y \times \text{width} \), as we can now just compute the linear index
of the centre of the stencil, then shift it around to get the other
neighbouring elements.

As well as enabling sharing between index computations, cur-
sored arrays strictly subsume our old delayed array representation.
To see this, suppose we added an alternative to our Generator type
that implemented delayed arrays as given in §3

data Generator sh a
  = ...
  | GenDelayed { genGetElem :: sh -> a }

It turns out this alternative is unnecessary, because we can write
functions to convert between the delayed and cursored representa-
tions. Given a cursored array, we construct the element function for
the delayed version making a cursor then immediately loading from
it. Given a delayed array, we construct the cursored one by using
the index itself as the cursor. This is possible due to the existential
quantification of the cursor type.

delayedOfCursored :: Generator sh a -> Generator sh a
delayedOfCursored (GenCursor make _ load) = GenDelayed (load . make)
cursoredOfDelayed :: Generator sh a -> Generator sh a
cursoredOfDelayed (GenDelayed getElem) = GenCursored id addIndex getElem
addIndex :: Shape sh => sh -> sh -> sh -> sh
addIndex = ...

To see that cursored arrays also support the delayed array ap-
proach to fusion, note that we can implement map by composing
its parameter with the load function of the cursored array. The fol-
lowing code gives the definition of mapGen which operates on the
generator. The version for arrays is easily defined in terms of it.

mapGen :: (a -> b) -> Generator sh a -> Generator sh b
mapGen f gen
  = case arr of
    GenManifest _
      -> GenCursored id addDim (\ix -> f (vec ! ix))
    GenCursored make shift load
      -> GenCursored make shift (f . load)

Finally, note that although we use cursored arrays internally to
the library, there is usually no need for client programs to construct
them explicitly. In the clients we have written, arrays are usually
constructed with higher level utility functions, and combinators
such as map and fold produce the same result independent of the
representation of their arguments.

5.2 Applying the Stencil

Now that we have the definition for cursored arrays, we can see
about creating one. Figure 9 gives the definition of mapStencil2
which takes the definition of a rank 2 stencil, a source array, and
produces a cursored result array. The definitions of the rectangles
for the border and internal regions have been elided to save space,
as have the inInternal and inBorder predicates, though they are
straightforward.

We have also elided the INLINE pragmas for the make, shift
and load* functions. When compiling with GHC we must de-
fine these functions as separate bindings and give them INLINE

| INLINE | INLINE |

\[
\begin{array}{cccc}
  y0 & y1 & y2 & y3 \\
\end{array}
\]

Figure 8. Overlapping support of four adjacent 3x3 stencils

4 \times 9 = 36 loads of the source array, and four stores to the result.
However, evaluating all four points in one operation requires only
18 loads, as well as the four stores to the result. There is also
the potential to share the evaluation of array indices, and well as
multiplications, depending on the form of the stencil.

To see this, suppose we added an alternative to our Generator type
that implemented delayed arrays as given in §3

data Generator sh a
  = ...
  | GenDelayed { genGetElem :: sh -> a }

It turns out this alternative is unnecessary, because we can write
functions to convert between the delayed and cursored representa-
tions. Given a cursored array, we construct the element function for
the delayed version making a cursor then immediately loading from
it. Given a delayed array, we construct the cursored one by using
the index itself as the cursor. This is possible due to the existential
quantification of the cursor type.

delayedOfCursored :: Generator sh a -> Generator sh a
delayedOfCursored (GenCursor make _ load) = GenDelayed (load . make)
cursoredOfDelayed :: Generator sh a -> Generator sh a
cursoredOfDelayed (GenDelayed getElem) = GenCursored id addIndex getElem
addIndex :: Shape sh => sh -> sh -> sh -> sh
addIndex = ...

To see that cursored arrays also support the delayed array ap-
proach to fusion, note that we can implement map by composing
its parameter with the load function of the cursored array. The fol-
lowing code gives the definition of mapGen which operates on the
generator. The version for arrays is easily defined in terms of it.

mapGen :: (a -> b) -> Generator sh a -> Generator sh b
mapGen f gen
  = case arr of
    GenManifest _
      -> GenCursored id addDim (\ix -> f (vec ! ix))
    GenCursored make shift load
      -> GenCursored make shift (f . load)

Finally, note that although we use cursored arrays internally to
the library, there is usually no need for client programs to construct
them explicitly. In the clients we have written, arrays are usually
constructed with higher level utility functions, and combinators
such as map and fold produce the same result independent of the
representation of their arguments.

5.2 Applying the Stencil

Now that we have the definition for cursored arrays, we can see
about creating one. Figure 9 gives the definition of mapStencil2
which takes the definition of a rank 2 stencil, a source array, and
produces a cursored result array. The definitions of the rectangles
for the border and internal regions have been elided to save space,
as have the inInternal and inBorder predicates, though they are
straightforward.

We have also elided the INLINE pragmas for the make, shift
and load* functions. When compiling with GHC we must de-
fine these functions as separate bindings and give them INLINE

| INLINE | INLINE |
data Boundary a
  = BoundConst a
  | BoundWrap
  | ...

mapStencil2 boundary stencil@(Stencil sExtent _ _) arr
  = let (Z :. aHeight :. aWidth) = extent arr
      (Z :. sHeight :. sWidth) = sExtent
        rectsInternal   = ...
        rectsBorder    = ...
        inInternal ix   = ...
        inBorder ix     = ...
        
        make (Z :. y :. x) = Cursor (x + y*aWidth)
        shift (Z :. y :. x) (Cursor offset) = Cursor (offset + x + y*aWidth)
        loadBorder ix = case boundary of ... 
        loadInner cursor
          = unsafeAppStencil2 stencil arr shift cursor 
          in Array (extent arr)
            [ Region (RangeRects inBorder rectsBorder) (GenCursored id addIndex loadBorder) ]
            , Region (RangeRects inInternal rectsInternal) (GenCursored make shift loadInner) ]

unsafeAppStencil2
  :: Stencil DIM2 a -> Array DIM2 a
  -> (DIM2 -> Cursor -> Cursor) -- shift cursor
  -> Cursor -> a

unsafeAppStencil2
  stencil@(Stencil sExtent sZero sAcc)
  arr@(Array aExtent [Region RangeAll (GenManifest vec)])
  cursor
  | _ :. sHeight :. sWidth <- sExtent
   , sHeight <= 3, sWidth <= 3
    = template3x3 loadFromOffset sZero
  | otherwise = error "stencil too big for this method"
where
  getData (Cursor index)
    = vec 'unsafeIndex' index
  loadFromOffset oy ox
    = let offset = Z :. oy :. ox
        cur' = shift offset cursor 
        in sAcc offset (getData cur')

template3x3 :: (Int -> Int -> a -> a) -> a -> a
template3x3 f sZero
  = f (-1) (-1) $ f (-1) 0 $ f (-1) 1
  $ f 0 (-1) $ f 0 0 $ f 0 1
  $ f 1 (-1) $ f 1 0 $ f 1 1
  $ sZero

---

Figure 9. Applying the stencil to an array

The values of the border elements depend on the boundary parameter, and two options are shown at the top of the figure. The inner elements are defined via unsafeAppStencil2, which produces a function from the cursor value to the corresponding array element. Note that this function requires the provided source array to be manifest, so that elements can be extracted directly from the underlying vector using unsafeIndex. We use unsafeIndex to access the vector because this function performs no bounds checks, so we do not suffer the associated overhead. The safety of these accesses depends on the correctness of our library code, namely the rectsInternal list from mapStencil2, so that loadInner is not applied too close to the border.

Computation of the inner array elements is performed by the loadFromOffset and template3x3 functions. The latter spells out every possible offset from the centre of a 3x3 stencil. We must spell out these offsets “long hand” instead of writing a recursive function to compute the result because we need each application of f to be specialised for the provided offset. During compilation, f will be bound to a coefficient function like the one defined in laplace of Figure 7. In effect, we are using template3x3 to select every possible coefficient that the coefficient function could have defined. By virtue of makeStencil of Figure 6, if the coefficient function returns a valid coefficient for a particular offset then we end up with a term that multiplies that coefficient with data from the source array. If not, then the Nothing branch of makeStencil comes in to play and the result is unperturbed. Note that this mechanism permits us to use any stencil that fits inside the 3x3 template. For example, stencils of size 3x1 and 2x2 also work.

Sadly, the fact that we must spell out every possible offset means that our unsafeAppStencil2 function is limited to handling stencils of a particular maximum size. In this case we have set the maximum to 3x3, so that it fits on the page. However, the limit is easy to increase and our concrete implementation currently uses 7x7. Limiting the size of the stencil in this way does not affect what coefficients or zero elements can be used; it just requires the entire stencil to fit inside the template. If we had instead written a recursive version of the template function, then GHC would not inline it, killing performance. In general, repeatedly inlining a recursive function may not terminate, leading to divergence at compile time. We can think of several ways of addressing this issue, but all require modification to the compiler, and we defer further discussion to §7.2. If the stencil does not fit inside the template then we fall back to the standard approach of loading the coefficients into a manifest array and iterating directly over that. This gets the job done, but obviously misses out on the benefits of the cursored approach. A follow on effect of spelling out every offset is that it also limits mapStencil2 to arrays of rank 2. It is straightforward to write versions for other ranks, as the general structure is the same as the rank-2 case, but we don’t have a way of doing this polymorphically.

Finally, note that unsafeAppStencil2 defines a function between a cursor and a single array element. The task of actually filling the result array while exposing sharing between adjacent elements is performed by fillCursoredBlock2 which we discuss in the next section.
case quotInt# ixLinear width of { ix ->
case remInt# ixLinear width of { iy ->
case *# iy (*# 1Y width) of { ixCenter ->
writeFloatArray# world arrDest ixLinear

("# (indexFloatArray# arrBV
  (* (arrBV start (* (arrBV width 1Y iy) 1X)))
  (* (indexFloatArray# arrBM
    (* (arrBM mask (* (arrBM width iy) 1X)))
    (* (indexFloatArray# arrSc
      (* (arrSc start (* ixCenter width))))
    (indexFloatArray# arrSc
      (* (arrSc start (* ixCenter 1)))
    (indexFloatArray# arrSc
      (* (arrSc start (* ixCenter (-1))))
    (indexFloatArray# arrSc
      (* (arrSc start (* ixCenter (* (-1) width))))
    (4.0))))))

Figure 10. New core IR for Laplace with index sharing

fillCursoredBlock2 :: IOVector a -- vector to write into
  -> (DIM2 -> cursor) -- makeCursor
  -> (DIM2 -> cursor -> cursor) -- shiftCursor
  -> (cursor -> a) -> Int -- loadElem, width
  -> Int -> Int -> Int -- coords of block
  -> IO ()
fillCursoredBlock2 !vec !make !shift !load
| !width !x0 !y0 !x1 !y1 = ... fillRow4 ...
where
  fillRow4 !y !x -- fill a single row in the block
  | x + 4 > x1 = ... less than 4 elems remaining
  | otherwise
    | do let srcCur0 = make (Z:.y:.x)
    | let srcCur1 = shift (Z:.0:.1) srcCur0
    | let srcCur2 = shift (Z:.0:.1) srcCur1
    | let srcCur3 = shift (Z:.0:.1) srcCur2
    | let val0 = load srcCur0
    | let val1 = load srcCur1
    | let val2 = load srcCur2
    | let val3 = load srcCur3
    | touch val0; touch val1; touch val2; touch val3
    | let !dstCur0 = x + y * width
    | unsafeWrite vec (dstCur0) val0
    | unsafeWrite vec (dstCur0 + 1) val1
    | unsafeWrite vec (dstCur0 + 2) val2
    | unsafeWrite vec (dstCur0 + 3) val3
    | fillRow4 y (x + 4)

Figure 11. Block evaluation function for cursored DIM2 arrays

5.3 Filling the Array, and Interaction with LLVM

Using our original force function (not shown), but with cursored arrays, produces a loop whose inner fragment consists of the Core IR shown in Figure 10. The loop as a whole iterates through the linear indices of the result vector. In the body, each linear index (ixLinear) is converted to a rank-2 index, then back to a cursor value ixCenter. As the source and destination arrays have the same dimensions, ixLinear and ixCenter will have the same value. The intermediate conversion is successfully eliminated by the LLVM optimiser, so doesn’t appear in the object code.

Note how each of the elements of the source array are indexed relative to the cursor ixCenter. To recover sharing between adjacent elements we must evaluate several in the same iteration, which requires a new version of force. The inner loop of this new version is defined by fillCursoredBlock2 in Figure 11, which is also part of the library. This function takes a mutable IOVector, along with the functions that form a cursored array, and uses them to fill a rectangular block in the vector. Parallelism is introduced by having force fork off several threads, with each filling a different block of array elements. Performing block-wise evaluation also improves cache usage, as the evaluation of each successive row in a block usually requires source elements that were loaded into cache during the evaluation of previous rows.

In the definition of fillCursoredBlock2 we have manually applied the unroll-and-jam transformation [7] to evaluate groups of four consecutive elements per iteration. We operate row-wise, which is good for regions that are at least four elements wide. To evaluate narrow regions such as the one pixel wide left-hand border from Figure 4 it is better to operate column-wise, using a separate filling function derived from the presented code.

The touch function in the inner loop is used to place a dependency on the computed array values, and prevent GHC from floating the srcCur* and val* bindings into the applications of unsafeWrite. The touch function has the following type, and is defined in terms of the GHC primitive operation touch#.

    touch :: Elt a => a -> IO ()

We need all four element values to be computed before any of them are written to the result array. This is to avoid a hairy interaction with the LLVM optimiser. Specifically, LLVM does not know that the low-level representation of the source and result arrays do not alias, nor does it know that the result array and GHC stack do not alias. Any write to the result array or stack is assumed to also modify the source array, which invalidates data held in registers at that point. This in turn breaks the GVN (Global Value Numbering) optimisation which we depend on to recover sharing.

The disassembled x86_64 object code for the inner part of our loop is given in Figure 12. This is for the Sobel5 stencil shown in Figure 3. Floating point loads are marked with round bullets, while floating point stores are marked with diamonds. There are 18 loads and 4 stores, and examining Figure 8 shows that this is the optimal number for such a 3x3 stencil. However, we still have a slight inefficiency due to aliasing issues. Note the repeated instruction mov 0x6(rbx),rcx after each floating point store. The rbx register contains a pointer to the stack, and each floating point store invalidates the previously loaded value in rcx. Aliasing becomes more of a problem when compiling to architectures with insufficient floating point registers. For example 32bit x86 code can only address 8 of the 16 XMM registers available in 64bit mode. If the LLVM compiler runs out of registers then it spills values to the stack, which also invalidates previously loaded values. Fixing this will require more work on GHC’s LLVM backend, and/or a type system or analysis that recovers the non-aliasing of heap objects.

Finally, note that the optimal number of elements to compute per iteration depends on the form of the stencil, namely how many coefficients overlap when several stencils are placed side-by-side. Computing too few elements per iteration limits how much sharing can be recovered, while computing too many increases register pressure and can cause intermediate values to be spilled to the stack. Currently, we always compute four at once, which works well for most 3x3 stencils. In future work we intend to add a size-hint to our Array type, which would be set by mapStencil2. The fillCursoredBlock2 function would use this hint to choose between several loops, all with the same form as fillRow4, but computing a different number of elements per iteration.

6. Benchmarks

In this section we discuss the performance of our stencil benchmarks. All measurements were taken on a 2GHz 2xQuadCore Intel
Figure 13. Laplace on 2xQuad Core 2.0GHz Intel Harpertown

Figure 12. x86_64 assembly for Sobel applied to four consecutive pixels. FP loads and stores are marked with * and o.

Figure 14. Variation in runtime of Unrolled Stencil Laplace

Figure 15 shows the runtimes of the Sobel stencil applied to three

dependently scheduled whenever the OS “feels like it”. which ensures that all threads run in lockstep, instead of being in-

duced to our earlier work in [15]. In the end, it appears as though the

in Figure 11, and there isn’t enough sharing inherent in the

calculations corresponding to the application of

side. Still, it’s not an unreasonable result for a Haskell program,

or fusion on a given benchmark to cause in excess of a 10x linear

size of 300x300, matching to our earlier work in [15]. In the end, it appears as though the

speedup of this benchmark is limited by scheduling issues. Figure 14 shows the huge variation in runtime for 100 consecutive runs using 4 threads. Increasing the efficiency of our inner loop has also

reduced the grain size of the computation. When the grain size is

small, there is a high chance that some threads will have started (or completed) their work before the others are scheduled by the OS. To fix this problem we expect that we need gang scheduling [12], which ensures that all threads run in lockstep, instead of being independently scheduled whenever the OS “feels like it”.

6.1 Laplace Again

Figure 13 shows the performance of our Laplace benchmark. The Safe Get version uses the code from Figure 1 with bounds checked indexing, while Unsafe Get is the same code with unchecked indexing. Unsafe Unrolled Get is the bounds checked version, but using the unrolled filling function in Figure 11. Safe Unrolled Stencil uses the unrolled filling function, as well as our cursored arrays. The single threaded Handwritten C version is about 45% faster than our best Haskell result, which is achieved with 3 threads. The C version beats the Haskell one because it doesn’t need the initial

x + y + width calculations corresponding to the application of

make in Figure 11, and there isn’t enough sharing inherent in the

Laplace stencil for the Haskell version to exploit. For this, note the small amount of overlap in four Laplace stencils placed side-by-

side. Still, it’s not an unreasonable result for a Haskell program, considering that the C version produces an inner loop that appears close to optimal. We tried eliminating the application of make, but this turned out not to be an improvement due to the extra register required to maintain the centre index between loop iterations.

For this benchmark we used an image size of 300x300, matching to our earlier work in [15]. In the end, it appears as though the

speedup of this benchmark is limited by scheduling issues. Figure 14 shows the huge variation in runtime for 100 consecutive runs using 4 threads. Increasing the efficiency of our inner loop has also

reduced the grain size of the computation. When the grain size is

small, there is a high chance that some threads will have started (or completed) their work before the others are scheduled by the OS. To fix this problem we expect that we need gang scheduling [12], which ensures that all threads run in lockstep, instead of being independently scheduled whenever the OS “feels like it”.

6.2 Sobel Operator

Figure 15 shows the runtimes of the Sobel stencil applied to three

image sizes. Also shown is a single threaded version using the

cv : :Sobel function of OpenCV 2.2.0. This is using 32bit floats for the array values. To mitigate variance in runtime due to scheduling issues, we took the best result of 10 runs for each point. In this case, single threaded OpenCV is faster than our single threaded Haskell code primarily because it is using SSE SIMD intrinsics that we do not have access to from Haskell. The LLVM compiler also does not
Figure 15. Sobel and Canny runtimes, 100 iterations

yet support auto-vectorisation to collect separate operations into fused SIMD instructions itself. With SSE, the OpenCV version is able to perform loads, stores, additions and multiplications on four packed 32bit floats at a time. However, in all cases we are able to match OpenCV, with the larger image sizes only needing two threads to break even.

6.3 Edge Detection

Figure 16 shows the result of applying the Canny algorithm to an example image, with our implementation using two thresholds for edge linking hysteresis. Our implementation is broken into several stages: 1) convert the input RGB image to greyscale; 2) perform a Gaussian blur to suppress high frequency noise; 3) differentiate the image with $\text{Sobel}_x$, $\text{Sobel}_y$; 4) compute magnitude and orientation of the vector gradient; 5) classify local maxima of the gradient into strong and weak edges using the thresholds; 6) select points marked as strong edges; 7) link weak edges that are attached to strong edges. The output consists of all points marked as strong edges, as well as any weak edges that are attached to strong edges. A breakdown of runtimes for each of these stages applied to a 1024x1024 image is shown in Figure 17, while other sizes are also in Figure 15.

When all is said and done our single threaded implementation is about 4 times slower than OpenCV. With 8 threads it’s about 50% slower with a 512x512 image, 10% slower for 768x768, and on par for 1024x1024. We feel this is a good result considering that the blur and differentiation stages for the OpenCV version use SIMD operations that we cannot access from Haskell. The OpenCV implementation also uses different data formats for the various stages, converting between 8-bit unsigned and 16-bit signed integers during the application of $\text{Sobel}_x, y$. The other stages are performed in a mixture of 8 and 16 bit integer formats. In our own code we also perform the greyscale conversion and edge linking with 8 bit integers. However, using integer operations for the other stages does not help us due to the lack of registers and the aliasing issues mentioned in §5.3.

The OpenCV implementation also hand-fuses the “local maxima” and “select strong” stages, recording an array of indices for strong edges pixels while computing the local maxima. To duplicate this behaviour we would need to provide a joint `mapFilter` operation, with a corresponding version of `fillCursoredBlock2`. The delayed array approach cannot recover this form of fusion automatically as it cannot be expressed by simple function composition.

On the positive side, the performance of our Haskell code is more than adequate for real-time edge detection of a video stream. We have an OSX demo available from the Repa homepage [24].

7. Challenges of Array Fusion

In this section we summarise the main challenges we have encountered with this work, and suggest avenues for future research.

7.1 Lack of Support for SIMD Operations

At face value, using 4-way SIMD instructions such as available in the SSE or MMX set has the potential to improve the performance
of certain algorithms 4-fold. This assumes that our application isn’t memory bound, though note that even a 1024x1024 image of 32bit floats sits comfortably in the 12MB cache of our test machine. Of course, the fact we are using a super-scalar architecture implies that we won’t necessarily get a 4-fold speedup on a linear instruction stream, though we note that using SIMD also effectively increases the size of the register set. Expanding the register set would help to avoid the aliasing issues discussed in §5. Whether it’s better to introduce SIMD instructions in Haskell code, or have the LLVM compiler reconstruct them is an open question.

7.2 Manual Unwinding of Recursive Functions
As mentioned in §5.2 we must manually unfold loops over regions and rectangles as GHC avoids inlining the definitions of recursive functions. The nice way to fix this would be some form of super-compilation [4, 19]. Support for supercompilation in GHC is currently being developed, though still in an early stage. Failing that, we could perhaps add a new form of the INLINE pragma that unfolded recursive functions indiscriminately, or a fixed number of times. The downside of the first is potential divergence at compile time, the downside of the second is lack of generality.

7.3 Unboxing Outside of Loops
As mentioned in §4.2 we currently need to add some boilerplate code to functions such as solveLaplace to ensure that arrays are unboxed outside of loops, instead of once per iteration. This code has the following form, and is applied to each input array:

```
f arr@((Array _ [RangeAll (GenManifest _)])
  = arr 'deepSeqArray' ...
```

The `deepSeqArray` function places a demand on every boxed object in `arr` before returning its second argument. The pattern match is added to functions that we know will only ever be passed forced arrays, and ensures that indexing operations in the body of the function are specialised for this case. The root problem is that unboxing operations are represented as case matches, but while let-bindings can be floated out of loops, case matches cannot. We hope to fix this particular inelicity in the near future.

7.4 INLINES and Whole Program Compilation
As almost every function definition in the Repa library has an INLINE pragma, we are essentially doing whole program compilation, at least for the array part. In a syntactic sense, the INLINES do clutter up the code, and we have spent hours hunting performance problems that were due to a lack of an INLINE. In a deeper sense, we feel uneasy about the fact that performance depends so heavily on the syntactic structure of the code, but we don’t have a general solution for this. In §5.2 we mentioned the need to write the `make`, `shift` and `load` functions as separate function bindings, and attach INLINE pragmas. The need to write separate bindings is simply driven by the need to add INLINES, as in GHC this information is attached to the name of the function binding to be inlined.

Although we could imagine adding a desugaring pass that converted the source code to the desired form, in practice we also want to manually attach inlining stage numbers to many of the bindings. Stage numbers are used to ensure that some bindings are inlined and specialised before others. This can avoid the need for the compiler to optimise large swathes of code only to discard it due to case specialisation later in the compilation.

7.5 Promises of Purity
Figure 18 shows the code for the `force` function that produces a manifest array from a delayed one. This function also has the distinction of being the interface between the IO code that fills the array using concurrent threads, and our pure Repa API. Constructing the pure interface consists of two aspects, which are embodied by the following functions:

```
unsafePerformIO :: IO a -> a
unsafeFreeze :: MVector IO a -> IO (Vector a)
```

The `unsafePerformIO` function breaks the monadic encapsulation of an IO action. Remembering that we’re using a lazy language, this is effectively a promise by the programmer that the result can be evaluated at any time without affecting its final value. Similarly, `unsafeFreeze` coerces a mutable vector (MVector) to an immutable one (Vector), and serves as a promise that after that point in time, the underlying data will not be mutated further. Importanty, failing to respect the two promises results in undefined behaviour at runtime, and neither of the promises can be statically checked by the compiler. Due to this, we would prefer if such promises were enforced by someone else. Actually, the Data.Vector library almost provides what we want:

```
create :: (forall s. ST s (MVector (ST s) a)) -> Vector a
```

This function takes an ST action that produces a mutable vector, evaluates it, then coerces the result to an immutable one. The soundness of the coercion is guaranteed by the ST state variable (s), which ensures that no references to the mutable vector can escape from the scope of the action that produced it [17]. Unfortunately, there is no equivalent version of `create` for the IO monad, and we need IO because the primitive functions we use to implement parallelism produce IO actions.

More specifically, the `readMVar` and `putMVar` functions operate on mutex variables, and the result of the first can depend on the order in which concurrent threads are scheduled. Note that it is concurrency, not destructive update that is the essential problem here, as destructive update by itself can be safely encapsulated in ST. In related work, the type system of Deterministic Parallel Java (DPJ) [1] can express that concurrent writes to non-overlapping partitions of an array do not interfere. However, the published version of DPJ does not support parametric polymorphism of value or effect expressions. This lack of polymorphism makes it impractical to work with the higher order functions we use in Repa. Until a more fine-grained control over effects makes it into a higher-order system, it seems that we are forced to use primitives like `unsafePerformIO` and subvert the guarantees of our purely functional language.

On a happy note, although we can’t statically check the soundness of our purifying coercions, at least they are confined to a single place in the code – the `force` function. This function is also the logical place for such coercions, as it converts the “abstract” delayed array into a concrete, manifest one.

---

**Figure 18.** The interface between pure and monadic code
8. Related Work

Stencil computations are central to a wide range of algorithms in scientific computing. Recent implementations based on the imperative or object oriented paradigm include the Fortran CAPTools toolkit [14], which is designed to automate the process of parallelising sequential Fortran 77 code as much as possible; Co-array Fortran [20], an extension to Fortran 95 which includes explicit notation for data decomposition; and the parallel Java dialect Titanium [13].

From the declarative language community, the work on Single Assignment C (SAC) [26] has exerted the most influence on our work on Repa. However, SAC does not have specific support for stencil computations, as far as we know. ZPL [9], a predecessor of Chapel [3] is a parallel array language with Modula-2 like syntax. Both languages define array values in a declarative fashion, similarly to our own approach.

There are two other Haskell libraries targeting parallel stencil computations: Ypnos [22], which in contrast to Repa, provides a special stencil pattern matching syntax for one and two dimensional stencils. Ypnos also supports historic stencils, meaning that the stencil can reference arrays computed in previous iterations. The paper does not provide any performance figures, so we cannot compare this aspect to Repa. It would be interesting to investigate whether Repa would be a suitable backend for Ypnos.

The PASTHA [18], whose implementation is based around IDArray, supports historic stencils and includes the specification of the convergence condition as part of the stencil problem definition. In the paper, only relative speedup numbers are provided, so we were not able to compare PASTHA’s performance to Repa.

Accelerate [8] is a high-level embedded language for multidimensional array computations. In contrast to Repa, it has at its core an online code generator which targets NVIDIA’s CUDA GPGPU programming framework. Accelerate recently gained support for stencil generation as well, but follows a rather different approach, due to its embedded nature.

Acknowledgements Thanks to Rami Mukhtar and Ben Lever for writing the original Canny Edge Detection code, Roman Leshchinsky for suggesting the use of cursored arrays, and Simon Peyton Jones for describing the behaviour of the GHC simplifier. This work was supported in part by the Australian Research Council under grant number LP0989507.

References


