Parallel Programming in Haskell Almost for Free

an embedding of Intel's Array Building Blocks

Bo Joel Svensson
Chalmers University of Technology
joels@chalmers.se

Mary Sheeran
Chalmers University of Technology
ms@chalmers.se

Abstract

Nowadays, performance in processors is increased by adding more cores or wider vector units, or by combining accelerators like GPUs and traditional cores on a chip. Programming for these diverse architectures is a challenge. We would like to exploit all the resources at hand without putting too much burden on the programmer. Ideally, the programmer should be presented with a machine model abstracted from the specific number of cores, SIMD width or the existence of a GPU or not. Intel's Array Building Blocks (ArBB) is a system that takes on these challenges. ArBB is a language for data parallel and nested data parallel programming, embedded in C++. By offering a retargetable dynamic compilation framework, it provides vectorisation and threading to programmers without the need to write highly architecture specific code. We aim to bring the same benefits to the Haskell programmer by implementing a Haskell frontend (embedding) of the ArBB system. We call this embedding EmbArBB. We use standard Haskell embedded language procedures to provide an interface to the ArBB functionality in Haskell. EmbArBB is work in progress and does not currently support all of the ArBB functionality. Some small programming examples illustrate how the Haskell embedding is used to write programs. ArBB code is short and to the point in both C++ and Haskell. Matrix multiplication has been benchmarked in sequential C++, ArBB in C++, EmbArBB and the Repa library. The C++ and the Haskell embeddings have almost identical performance, showing that the Haskell embedding does not impose any large extra overheads. Two image processing algorithms have also been benchmarked against Repa. In these benchmarks at least, EmbArBB performance is much better than that of the Repa library, indicating that building on ArBB may be a cheap and easy approach to exploiting data parallelism in Haskell.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; Concurrent, distributed, and parallel languages; D.3.4 [*Programming Languages*]: Processors—Code generation

General Terms Languages, Performance

Keywords Data parallelism, array programming, dynamic compilation, embedded language

FHPC'12, September 15, 2012, Copenhagen, Denmark.

1. Introduction

Modern hardware architectures provide much parallelism in the form of multi- and many-core processors and heterogeneous combinations of the two. We would like to exploit this parallelism without over-burdening the programmer, and without sacrificing too much performance. Ideally, the programmer should be encouraged to write the code in a way that is not too closely linked to the precise combination of cores, SIMD vector processing power and GPU that is currently targeted. It should also be possible to easily retarget the code to new architectures. As processor architectures develop and are combined at an ever increasing pace, these requirements place considerable demands on programming languages and libraries for parallel programming.

Intel's Array Building Blocks (ArBB) is one approach to the problem of how to increase productivity for programmers who need to exploit hardware parallelism, but for whom very low level parallel programming (as typically found in High Performance Computing) is too difficult or time consuming [11]. ArBB is a retargettable dynamic compilation framework that is provided as an embedded language in C++. It aims to give the programmer access to data and thread parallelism, while protecting him from common pitfalls such as race conditions and deadlock.

Many of the ideas in ArBB feel familiar to functional programmers. For example, when referring to collection data types, reference [11] states that "Expressions over collections always act as if a completely new collection were created and assignment between collections always behaves as if the entire input collection were copied." This rather functional view of data is accompanied by extensive optimisation that (as the paper states) removes almost all copying. A key to the approach is the use of standard patterns of computation such as scan and reduce – familiar as higher order functions in a functional setting. The system tries to fuse away intermediate data-structures when these patterns are composed. The patterns are deterministic, so that problems with race conditions are avoided by design. Programs are stated to be short and close to the mathematical specification, and that, again, is a familiar mantra. The system allows C++ users to construct code objects called closures. The first time the *call* operation is applied to a particular function, ArBB captures the sequence of operations over ArBB types and creates an intermediate representation. This representation is then compiled to vector machine language, which is cached for later use. The machine code is also invoked (in parallel on multiple cores, if appropriate). The next time the function is called, ArBB reuses the cached machine code rather than regenerating it. Although dynamically typed closures are available, ArBB is generally statically typed.

ArBB is an embedded language, implemented as a library in the host language C++. The embedding makes heavy use of the C++ template machinery. There is also a lower level interface to ArBB, the ArBB Virtual Machine C API [7]. The low level C API

is not intended for use directly by applications programmers, but rather as a base for implementations of alternative ArBB frontends (embeddings) [11]. The ArBB VM implements the compilation and parallelisation. Reference [14] presents Haskell bindings to the ArBB C API as well as initial steps towards using ArBB as a backend to the embedded language Accelerate [3].

The design of ArBB seems particularly well suited to a functional setting. In this paper, we present work in progress on embedding ArBB in Haskell. We call the embedding EmbArBB. It is our hope that using Haskell as a host language will be just as user friendly and efficient as the C++ version. EmbArBB does not yet include all of the functionality of ArBB, and this is discussed further in section 8. However, our first benchmarks show that the performance of EmbArBB is very similar to that of ArBB in C++. This bodes well.

EmbArBB is available at GitHub as `github.com/svenssonjoel/EmbArBB`.

2. Related Work

2.1 Data.Array.Accelerate

Accelerate [3] is an embedded language for general purpose, flat data parallel computations on GPUs.

The Accelerate programmer expresses algorithms using collective operations over vectors. These collective operations are similar to the parallel patterns of ArBB, but are generally higher order. That is, where ArBB and EmbArBB have `addReduce`, `mulReduce` and so on, Accelerate has a single higher-order fold function. This means that the Accelerate library gets away with a much smaller set of operations, while maintaining a higher level of expressivity in the language.

Accelerate arrays have their dimensionality (shape) encoded in the type and this was the inspiration for the similar approach taken in EmbArBB. In Accelerate, a one dimensional array of integers has type `Array DIM1 Int`. However, Accelerate does not limit the dimensionality to one, two or three, as ArBB does, but rather supports arbitrary dimensionality.

2.2 Data Parallel Haskell

Data Parallel Haskell (DPH) is an extension to GHC for nested data parallelism. This is one thing that DPH and ArBB have in common.

In DPH, the programmer can use parallel arrays of type `[:e:]` that look similar to normal Haskell lists, but give access to data parallel execution. The similarity to Haskell list processing doesn't end there. The operations on these parallel arrays are also reminiscent of Haskell's normal list processing functions. For example, `mapP`, `unzipP` and `sumP` are parallel versions of these well known functions.

DPH relies on a technique called flattening to transform its nested data parallelism into flat data parallelism. A recent article about DPH is reference [9].

2.3 Feldspar

Feldspar is a language for Digital Signal Processing (DSP) programming developed at Chalmers, ELTE University and Ericsson (the telecom company) [1]. The functional `while` loop of EmbArBB is inspired by the same concept in Feldspar.

Feldspar is based on a deeply embedded core language and implements a vector library as a shallow embedding on top of that. This means that there are no vector specific constructs in the core abstract syntax.

2.4 Nikola

Nikola is an embedded language for GPU programming[10], also in Haskell. During the early phases of implementation of EmbArBB, the source code of Nikola was often studied for inspiration.

The embedding used in Nikola makes use of an untyped expression data structure wrapped up in phantom types, in the style of Pan [4]. The EmbArBB embedding works in the same way. One of the listed strengths of Nikola is the ability to generate GPU functions from Haskell functions, which enables function reuse and the ability to amortise the cost of code generation over several launches in an easy way. The ability the generate target language function from Haskell functions is a present in EmbArBB as well.

Nikola, like Accelerate, provides a set of higher-order functions with general `map`, `reduce` and `zipWith` functionality.

2.5 Repa

Repa is a library for regular, shape polymorphic parallel arrays [8]. Repa uses a concept of `delayed` arrays to obtain fusion of operations, such as the typical map fusion:

```
map f . map g  = map (f . g)
```

A delayed array has a representation that is quite direct to parallelise; it is represented as a function from an index-space to an element and the extents of that same index-space. Concretely:

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

Parallelising the computation of such an array is done by splitting up the index-space over the available parallel resources of the system. Repa does not use SIMD (vector) instructions; this is something that ArBB does and that EmbArBB gets for free.

Repa is compared to EmbArBB in the benchmarks in section 7.

3. Motivation

We have two main motivations for implementing EmbArBB.

Firstly, we are interested in exploring programming idioms for data parallel programming in a functional setting. Embedding ArBB gives a quick route to a platform for such exploration, without too much implementation effort. This is what the "almost for free" in the title is intended to refer to. We wish to explore ways to make use of the fact that we have an array programming library embedded in a sophisticated, strongly typed host language. In our earlier work on the embedded hardware description language Lava, we investigated various approaches to exploiting the host language during netlist synthesis [12], and we have also experimented with the use of search and dynamic programming in generating parallel prefix (or scan) networks [13]. We intend to apply similar methods to the development of data parallel programs once the EmbArBB implementation is more complete and stable.

A second motivation is the desire to teach NESL-style data parallel programming in a Masters course on parallel functional programming that has recently been introduced at Chalmers [6]. The first instance of the course (in Spring 2012) covered Blelloch's NESL language, with its associated cost model [2], but did not provide any satisfactory way for the students to experience real, nested data parallel programming. This was due not only to a lack of suitable hardware, but also to deficiencies in the available tools. We used the Repa library to get flat data parallelism, but then suffered from the lack of a built-in scan primitive (as many of Blelloch's NESL examples use scan). Perhaps as a result of needing scan, we did not get good performance. We have not done enough examples or experiments yet, but it seems to us that EmbArBB could be used to give students an experience of real parallel programming in a NESL-like functional language. It remains to be seen whether the limited degree of nesting allowed in ArBB can be offset, at least

to some extent, by clever use of the host programming language during generation of the desired abstract syntax tree.

4. Programming in ArBB

What the authors of ArBB call *latent parallelism* is expressed both by using operations like scan and reduce (or fold) over vectors and by mapping functions over collection data-types. ArBB provides both dense and nested vectors. Dense vectors can be one, two or three dimensional; they correspond to ordinary arrays.

Matrix-vector multiplication can be expressed in ArBB and C++ like this:

```
void arbb_matrix_vector(const dense<f32, 2>& a,
                        const dense<f32>& x,
                        dense<f32>& b) {
  b = add_reduce(a * repeat_row(x, a.num_rows()));
}
```

This function takes a dense matrix (a two dimensional array), `a`, and a dense vector, `x`, and returns a dense vector `b`. It uses `add_reduce` and `repeat_row`, which are built in ArBB operations. Note that we are not provided with a general reduction operator that takes a function as parameter, but rather with specific instances.

The matrix-vector multiplication code above is very concise, but some further steps are necessary before it can be run on real data. The code below shows how to set up a 4x4 scaling matrix and a vector to multiply by that matrix using ArBB. This entails allocating memory on the ArBB heap and copying data into it, reminiscent of the copying of data from host to device that is common in general purpose GPU (GPGPU) programming .

```
int main(void)
{
  float matrix[4][4] = {{2.0,0.0,0.0,0.0},
                        {0.0,2.0,0.0,0.0},
                        {0.0,0.0,2.0,0.0},
                        {0.0,0.0,0.0,2.0}};
  float vector[4] = {1.0,2.0,3.0,4.0};

  // set up the matrix in ArBB
  dense<f32, 2> a(4, 4);
  range<f32> write_a = a.write_only_range();
  float* a_ = &write_a[0];
  memcpy(a_,matrix,16*sizeof(float));

  // set up the vector in ArBB
  dense<f32> x(4);
  range<f32> write_x = x.write_only_range();
  float* x_ = &write_x[0];
  memcpy(x_, vector,4*sizeof(float));

  // Room for the result
  dense<f32> arbb_b(4);
  const_range<f32> read_arbb_b;

  call(arbb_matrix_vector)(a, x, arbb_b);
  read_arbb_b = arbb_b.read_only_range();

  for (int i = 0; i < 4; ++i) {
    printf("%f ", read_arbb_b[i]);
  }

  return 0;
}
```

After setting up and copying the data into ArBB, the function `arbb_matrix_vector` is *called* using the `call` command. At this point, a number of things happen. If the function is called for the first time, it is *captured*. This means that the function is run as a C++ function, which produces an intermediate representation (IR) of the computation (which is standard in such embeddings). For example, if the function being captured contains a normal C++ *for loop*, it will be unrolled, much in the same way as recursion in a Haskell embedded DSL results in unrolled code. After the function is captured, the IR is further optimised and finally executed. Compilation and capture only occur the first time a function is called. The compilation procedure is outlined in [11].

Nested vectors in ArBB allow the creation of an array of dense vectors of varying length – giving the ability to express less regular parallelism. There is a limit, though, in that only one level of such nesting is allowed. Section 5.7 shows a small example that uses this nestedness in EmbArBB to implement sparse matrix vector multiplication. The program in ArBB itself is very similar. We note, though, that one of the samples distributed with ArBB is also sparse matrix vector multiplication, but implemented using dense vectors and a function that can take sections of a vector. We will need to experiment further with nestedness and when it should be used.

5. Programming in EmbArBB

In EmbArBB, ArBB functions are expressed as Haskell functions on expressions (abstract syntax trees). This is standard Haskell embedded language procedure. A function that adds a constant to all elements of a vector is

```
addconst :: Num a
         => Exp a
         -> Exp (DVector Dim1 a)
         -> Exp (DVector Dim1 a)
addconst s v = v + ss
    where
      ss = constVector (length v) s
```

In this function `(+)` is used for elementwise addition of two vectors. The function `constVector` is part of the EmbArBB library and creates a vector of a given length containing the same value at each index.

EmbArBB supports one, two and three dimensional dense vectors. These vectors are represented by a datatype called `DVector` (for Dense Vector). `DVector` takes two arguments, the first specifying its dimensionality and the second its payload type. Hence, an array of 32 bit words has type `DVector Dim1 Word32`. The one dimensional `DVector` also has the alias `Vector`. The type of the `addconst` function above could also have been written:

```
addconst :: Num a
         => Exp a
         -> Exp (Vector a)
         -> Exp (Vector a)
```

The nested vectors of ArBB correspond to `NVectors` in EmbArBB.

The `addconst` function needs to be *captured* before it can be executed. The term *capture* is borrowed from ArBB nomenclature and corresponds to compiling the embedded language function into an ArBB function. In EmbArBB, a function is captured using the function `capture`; in the C++ embedding, a function is captured when it is called for the first time. The same could have been done in the Haskell embedding of course, but we chose to make capture explicit for simplicity. The `capture` function takes a Haskell function as input and produces an opaque typed function identifier as output. The identifier points out the function in an environment that is managed by a monad called `ArBB`. To make this concrete,

```
1  main =
2    withArBB $
3    do
4       f <- capture addconst
5       x <- copyIn $ mkDVector
6                       (V.fromList [1..10 :: Float])
7                       (Z:.10)
8       r1 <- new (Z:.10) 0
9       c <- mkScalar 1
10      execute f (c :- x)  r1
11      r <- copyOut r1
12      liftIO$ putStrLn$ show r
```

Figure 1. The code shows how to capture a function, upload data to ArBB and how to execute the captured function.

Figure 1 shows the complete procedure from capturing to execution.

The `withArBB` function (line two) is the "run"-function of the `ArBB` monad. It turns something of type `(ArBB a)` into `(IO a)`. This is how ArBB computations are interfaced with Haskell. A `withArBB` session also sets the scope for any captured functions or vectors created in the ArBB monad.

The result of `capture addconst` on line four in the code has type

```
Function (BEScalar Float :- BEDVector Dim1 Float)
         (BEDVector Dim1 Float)
```

representing a function that takes a float and a vector of floats as input and gives a vector of floats as result. The "BE" prefixes in those names (`BEScalar`,`BEDvector`) refers to this being vectors and scalar that reside in the ArBB heap (Backend vectors). BEVectors and Scalars are mutable and the vector used to store the result needs to be allocated by the programmer. This interface seems rather imperative but ArBB requires output data to be placed into an output vector of the correct size. Because of this, the options were either to try to infer result size (which can be dependent on the value of input data) or to have the programmer supply storage for the result. We chose the latter, and thus also avoid additional runtime overhead due to size inference.

Continuing with the example: the function, once captured, can be executed on data. On line five, a `DVector` is copied into ArBB using the `copyIn` function. A vector to hold the result is created using the `new` function on line eight. The `new` function takes a shape description (`(Z:.10)`, meaning a one dimensional vector of length 10) and an element to fill the vector with initially. A scalar `c` is created using `mkScalar` on line nine. Next, on line ten, the function is executed by issuing

```
execute f (c :- x) r1
```

The `(c :- x)` is a heterogeneous list of inputs. The `:-` operator is similar to normal cons (`:`) on Haskell lists. The output is stored in `r1`. Had there been more outputs, the output list would have also had a heterogeneous list type of the form `(r1 :- ... :- rn)`. If the result is a scalar, a storage location can be created using a function called `mkScalar` that takes its initial value as input.

All that remains is to copy the output vector out of ArBB and show the result; this is done on lines eleven and twelve in the figure.

In this example, we have seen all of the important parts of the Haskell to EmbArBB interface: how embedded language functions are compiled and executed, and how to transfer data from Haskell into the ArBB world. The following subsections show EmbArBB versions of some common data parallel computations.

```
-- Scatter values into a vector
-- resolves collisions by adding elements
addMerge :: Exp (DVector (t:.Int) USize) -- indices
         -> Exp USize                    -- res length
         -> Exp (DVector (t:.Int) a)     -- src
         -> Exp (DVector (t:.Int) a)

-- Reduce a vector using addition
addReduce :: Num a
          => Exp USize   -- rows, cols or pages
          -> Exp (DVector (t:.Int) a)
          -> Exp (DVector t a)

-- Segmented version of addReduce
addReduceSeg :: Num a
             => Exp (NVector a)  -- nested input
             -> Exp (DVector Dim1 a)

-- Create a nested vector from a dense
applyNesting :: Exp USize  -- lengths or offsets
             -> Exp (DVector Dim1 USize) -- nesting
             -> Exp (DVector Dim1 a)
             -> Exp (NVector a)


-- Create a vector with a constant value
constVector :: Exp USize  -- length
            -> Exp a      -- value
            -> Exp (DVector Dim1 a)

-- Extract a column from a 2D vector
extractCol :: Exp USize              -- col. index
           -> Exp (DVector Dim2 a)
           -> Exp (Vector a)

-- Fill a portion of a vector with a constant
fill :: Exp a                -- fill value
     -> Exp USize            -- start
     -> Exp USize            -- end
     -> Exp (DVector Dim1 a) -- dst
     -> Exp (DVector Dim1 a)

-- Gather elements from a vectors
gather1D :: Exp (DVector Dim1 USize)  -- indices
         -> Exp a                     -- default
         -> Exp (DVector Dim1 a)      -- values
         -> Exp (DVector Dim1 a)

-- Get the number of rows
getNRows :: Exp (DVector (t:.Int:.Int) a)
         -> Exp USize

-- Turn a zero dimensional vector into a scalar
index0 :: Exp (DVector Z a) -> Exp a

-- Create a 2D vector by repeating a 1D vector
repeatRow :: Exp USize            -- #repetitions
          -> Exp (DVector Dim1 a) -- row
          -> Exp (DVector Dim2 a)

-- Replace one column in a 2D vector
replaceCol :: Exp USize            -- col
           -> Exp (DVector Dim1 a) -- new values
           -> Exp (DVector Dim2 a)
           -> Exp (DVector Dim2 a)

-- Convert every element of a vector to USize
vecToUSize :: Exp (Vector a)
           -> Exp (Vector USize)
```

Figure 2. A list of EmbArBB functions that are used in the examples with short descriptions

- `ISize` is an integer type used to specify for example offsets.
- `USize` is an unsigned integer type used for lengths or indices.
- `Boolean` replaces the `Bool` type for EmbArBB programs. The reason for this is that ArBB internally represents booleans as 8bit words. while the `Storable` instance for `Bool` does not.
- `DVector` is the type of regular shaped vectors.
- `NVector` is the type of irregularly shaped vectors.

Figure 3. A list of EmbArBB types with short descriptions

In the examples one can assume that following imports have been made:

```
import Data.Vector as V
import Prelude as P
```

In this way, wherever there is potential for mixup between EmbArBB functions and Prelude or Data.Vector functions, these are prefixed by either "`V.`" or "`P.`".

5.1 Saxpy

Saxpy is a vector operation that gets its name from the description of what it does "*S*ingle-precision *A*lpha *X P*lus *Y*". There is also a double-precision version called *daxpy*. Here we can use a single source to obtain both versions:

```
saxpy :: Num a
      => Exp a
      -> Exp (DVector Dim1 a)
      -> Exp (DVector Dim1 a)
      -> Exp (DVector Dim1 a)
saxpy s x y = (ss*x) + y
    where
      ss = constVector (length x) s
```

Now, `capture` can be used to instantiate the function at either float or double types.

```
main =
  withArBB $
  do
     f <- capture saxpy

     let v1 = V.fromList [1,3..10::Float]
         v2 = V.fromList [2,4..10::Float]

     x <- copyIn $ mkDVector v1 (Z:.5)
     y <- copyIn $ mkDVector v2 (Z:.5)

     r1 <- new (Z:.5) 0

     c <- mkScalar 1

     execute f (c :- x :- y)  r1

     r <- copyOut r1

     liftIO$ putStrLn$ show r
```

Changing `Float` to `Double` in the definitions of `v1` and `v2` gives the double precision version of the function. A function needs to be captured at every type at which it is to be used. This is because the resulting ArBB function is not polymorphic over something corresponding to `Num` types (while the embedded language function is).

5.2 Matrix-vector multiplication

In section 4, matrix-vector multiplication was shown using the C++ ArBB interface. Here, EmbArBB is used to implement the same function. The function definition itself is very similar to the C++ one:

```
matVec :: Exp (DVector Dim2 Float)
       -> Exp (DVector Dim1 Float)
       -> Exp (DVector Dim1 Float)
matVec m v = addReduce rows
           $ m * (repeatRow (getNRows m) v)
```

The `addReduce` function takes a parameter that specifies whether to reduce rows or columns.

The following Haskell `main` function completes the comparison to the C++ version shown in the Introduction:

```
main =
  withArBB $
  do
     f <- capture matVec
     let m1 = V.fromList [2,0,0,0,
                          0,2,0,0,
                          0,0,2,0,
                          0,0,0,2]
         v1 = V.fromList [1,2,3,4]

     m <- copyIn $ mkDVector m1 (Z:.4:.4)
     v <- copyIn $ mkDVector v1 (Z:.4)

     r1 <- new (Z:.4) 0

     execute f (m :- v)  r1

     r <- copyOut r1

     liftIO$ putStrLn$ show r
```

The complete Haskell version of this program is slightly shorter than the corresponding C++ version. However, the C++ version could probably be made shorter as well; the Haskell and C++ versions must be considered similar in implementation complexity.

5.3 Matrix-matrix multiplication

The following EmbArBB implementation of matrix-matrix multiplication is used as a benchmark in section 7:

```
matmul :: Exp (DVector Dim2 Float)
       -> Exp (DVector Dim2 Float)
       -> Exp (DVector Dim2 Float)
matmul a b = fst $ while cond body (a,0)
  where
    m = getNRows a
    n = getNCols b
    cond (c,i) = i <* n
    body (c,i) =
      let mult = a * repeatRow m (extractCol i b)
          col  = addReduce rows mult
      in (replaceCol i col c, i+1)
```

This example uses a `while` loop. In the C++ embedding of ArBB, the programmer has the option of using a C++ `while` loop

or a special `while_` loop. The normal C++ while loop unrolls at capture time; the `while_` is kept by ArBB and corresponds to an ArBB loop. The two loops differ in the kinds of values on which they can depend. The C++ loop can only depend on C++ values, so the number of iterations in a normal `while` loop must be known at capture time. All of these concepts have Haskell counterparts. The `while` loop used in the example corresponds to the `while_` loop and will remain in the generated ArBB code; it can depend on ArBB values at runtime. Haskell recursion is used to get the kind of unrolling that one gets by using a C++ `while` loop.

The EmbArBB `while` loop takes three parameters, a condition, a body and an initial state. In every iteration of the loop, the body is applied to the state and the condition checked.

5.4 Histogram

The histogram of a grayscale image contains the frequency with which each shade occurs in the image. In this example, the image used represents the shade of each pixel with a single byte, so that 256 different shades are possible. An array of length 256 is created to hold at index `i` the number of occurrences of the shade `i` in the image. In essence, this is an array of buckets.

```
histogram :: Exp (DVector Dim2 Word8)
          -> Exp (Vector Word32)
histogram input = addMerge (vecToUSize flat) 256 cv
    where
      flat = flatten input
      cv = constVector (r * c) 1
      r  = getNRows input
      c  = getNCols input
```

The `addMerge` operation takes a vector of inputs, a vector of indices and finally a size (specifying the result size). Elements from the input vector (`cv` in the example) are placed into the result at the index specified at the same location in the indices vector. Elements placed at the same index are added together. Here, the input vector contains all ones, while the image is *cast* into a vector of indices. The result is that index `i` of the output vector contains the number of times shade `i` appears in the image. The histogram computation becomes very concise through the use of the `addMerge` operation.

The code below creates the image shown in figure 4. It takes as input a vector on frequencies and outputs a two dimensional vector, a grayscale image.

```
histImage :: Exp (Vector Word32)
          -> Exp (DVector Dim2 Word8)
histImage input = fst $ while cond body (cvn,0)
    where
      cond (img,i) = i <* n
      body (img,i) = (replaceCol i col' img,i+1)
          where
            val = input ! i
            col = extractCol i img
            col' = fill black 0 n col
            n = 255 - scale 255 m val

      n = length input
      cv = constVector (n*n) white
      cvn = setRegularNesting2D n n cv
      m   = index0 (maxReduce rows input)
      black = 0
      white = 255
```

Figure 4. The righ-hand image visualises the frequency with which different shades of gray occur in the left-hand one.

5.5 Sobel edge detection filter

Sobel edge detection is an example of a stencil computation over an array. At each element of the array, a computation is performed that depends on that element and on elements close by. Which of the nearby elements to use, and how much they influence the result is typically described using a matrix (in the two dimensional case) called the stencil. Below are two stencils used in the sobel edge detection filter:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

In ArBB, the *map* operation gives the programmer a way to implement stencil computation. It takes a function and a vector on which to apply the function at each element. Inside the function being mapped, the programmer may use a set of *getNeighbor* functions to access nearby elements.

The elements used by the stencils are kept (in the form of coordinates relative to the centre point of the matrix) in two Haskell lists. This is an example of how host language (Haskell) features are used as a kind of scripting aid.

```
s1, s2 :: [(Exp ISize,Exp ISize)]
s1 = [(1,1),(0,1),(-1,1),(1,-1),(0,-1),(-1,-1)]
s2 = [(1,1),(1,0),(1,-1),(-1,1),(-1,0),(-1,-1)]
```

The actual weights are placed in a separate list.

```
coeffs :: [Exp Float]
coeffs = [-1,-2,-1,1,2,1]
```

The following functions implement the stencil computation part of the program using the elements pointed out by `s1` and `s2` together with the weights in `coeffs`. Each of the six neighbours is multiplied by the appropriate weight. Haskell's `foldl` function is used to sum up the values, to give the result for the location in question. Using a Haskell fold means that the summation is unrolled.

```
gx :: Exp Word8 -> Exp Float
gx x = P.foldl (+) 0
     $ P.zipWith (*) [toFloat (getNeighbor2D x a b)
                      / 255
                     | (a,b) <- s1] coeffs


gy :: Exp Word8 -> Exp Float
gy x = P.foldl (+) 0
     $ P.zipWith (*) [toFloat (getNeighbor2D x a b)
                      / 255
                     | (a,b) <- s2] coeffs
```

Figure 5. The image on the right shows the result of applying the sobel edge detection filter to that on the left.

The helper functions `convertToWord8` and `clamp` take care of converting floats between 0 and 1 to bytes between 0 and 255, and of clamping floating point values into the 0 to 1 range.

```
convertToWord8 :: Exp Float -> Exp Word8
convertToWord8 x = toWord8 $ (clamp x)  * 255

clamp :: Exp Float -> Exp Float
clamp x = max 0 (min x 1)
```

The complete kernel, the program that is to be executed at every index of the image, is given below. The earlier parts are brought together into a function that transforms one `Word8` into another.

```
kernel :: Exp Word8 -> Exp Word8
kernel x = convertToWord8 $ body x
  where
    body x = sqrt (x' * x' + y' * y')
     where
       x' = gx x
       y' = gy x
```

In order to execute the kernel, an ArBB `map` is used.

```
sobel :: Exp (DVector Dim2 Word8)
         -> Exp (DVector Dim2 Word8)
sobel image = map kernel image
```

The implementation of `sobel` is pleasingly simple and concise in this setting. Despite the fact that this is a small example, the value of being able to use Haskell lists and operations is already becoming obvious.

5.6 Image filtering

The blur operation, which is a kind of spatial linear filter, shows another way to implement a stencil operation. The `mapStencil` function used is a composite function in the EmbArBB library; it is not provided as a primitive from the ArBB VM.

```
blur :: Exp (DVector Dim2 Word8)
        -> Exp (DVector Dim2 Word8)
blur image = vec2DToWord8 (res `div`  all16)
  where
    all16 = constVector2D (getNRows image)
                          (getNCols image)
                          16
    res = mapStencil (Stencil [1,2,1
                              ,2,4,2
                              ,1,2,1] (Z:.3:.3))
                     image'

    image' = vec2DToWord32 image
```

In Repa, similar `mapStencil` functionality is present but using Template Haskell to give a safer way to specify the stencil. In our version there is no protection agains mischievous stencil specification. This guarantees only that if the programmer says the stencil is two dimensional then it can only be applied to two dimensional vectors. Applying similar Template Haskell based extension here as well is probably quite easy now that Repa has shown how.

5.7 Sparse matrix multiplication

A sparse matrix can be represented by three vectors, one (cidx) containing column indices, one (offsets) indicating indices in the vector of values at which the first non-zero element of each row appears, and one containing the values themselves. This is known as the Compressed Sparse Row (CSR) format. Given such a matrix and a dense vector, the relevant elements of the dense vector can be gathered into a new vector using the column indices. The resuting vector is multiplied pointwise by the vector of non-zero values from the sparse matrix. All that remains, then, is to divide the result into rows and sum each one. The offsets vector points to where the division should happen. The code in EmbArBB is as follows:

```
smvm :: Exp (Vector USize)
     -> Exp (Vector USize)
     -> Exp (Vector Float)
     -> Exp (Vector Float)
smvm mval cidx os vec = addReduceSeg nps
    where
      ps = mval * gather1D cidx 0 vec
      nps = applyNesting offsets os ps
```

The multiplication acts pointwise on the elements of `vals` and the vector (of the same length) containing relevant elements of the dense vector (produced using `gather1D`). The function `applyNesting` produces a nested vector of rows. The function `addReduceSeg` sums each of the rows (or segments) in that array, producing the non-zero elements of the result vector. Note that there is irregular data parallelism here because the rows may have different lengths. We would hope to get parallelism both in the individual summations and between summations. This is a well known algorithm that can be traced back to Blelloch [2].

6. Implementation

EmbArBB is a deeply embedded language. There is a constructor for every operation that ArBB can perform. A deep embedding is useful when there is need to apply optimisations or transformations to the AST before executing the operations. However, EmbArBB currently relies entirely on ArBB to perform optimisations to the program. The only optimisation performed on the EmbArBB side is sharing detection. Detecting sharing reduces the number of calls into the ArBB C library. Should we add a GPU backend to EmbArBB, further GPU specific optimisations of the AST will likely become necessary.

6.1 Vectors

EmbArBB has support for one, two and three dimensional dense vectors, represented by a datatype called `DVector`.

```
import qualified Data.Vector.Storable as V

data DVector d a
  = Vector {vectorData  :: V.Vector a,
            vectorShape :: d}
```

The payload data in a `DVector` is stored in a vector from the `Data.Vector` library. There is also a `d` parameter that specifies the shape of the DVector.

The d parameter is used for a type level representation of the dimensionality of a vector, as in the Repa library [8]. The dimensionality is encoded using the following types together with the Int type.

```
data a :. b = a :. b
infixl :.

data Z = Z
```

The dimensions zero to three are represented as follows:

```
type Dim0 = Z
type Dim1 = Dim0 :. Int
type Dim2 = Dim1 :. Int
type Dim3 = Dim2 :. Int
```

For example in the reduction functions provided by EmbArBB the output vector is of a dimensionality one less than the input vector used. Below is the type of addReduce to illustrate this.

```
addReduce :: Num a
          => Exp USize
          -> Exp (DVector (t:.Int) a)
          -> Exp (DVector t a)
```

A somewhat unfortunate side effect of this is that currently EmbArBB has two kinds of scalars, Exp (DVector Dim0 a) and Exp a. The result of reducing a one-dimensional vector is a zero-dimensional vector. An operation called index0 converts from zero-dimensional vectors to scalars.

Irregular container, in ArBB called Nested vectors, are represented in EmbArBB by the type NVector. Currently there are no versions of the copyIn, copyOut or new functions for nested vectors. The programmer must transfer dense vectors into the ArBB heap and then apply nesting to them as part of the computation to perform thereupon. This means that having a concrete representation for a NVector is currently not useful. In the hope of being able to implement some of the transfer functions, even without direct support from the ArBB VM, we represent NVectors as a vector of dense data together with a vector containing segment lengths.

```
data NVector a =
       NVector { nVectorData     :: V.Vector a
               , nVectorNesting  :: V.Vector USize}
```

As part of the interface between Haskell and EmbArBB there are also mutable vectors, of a type called BEDVector. These are vectors that reside in the ArBB heap and are represented only by an integer identifier with which the actual data can be accessed from ArBB. These are used to store the inputs and outputs of calls to execute on a captured function.

```
data BEDVector d a =
       BEDVector { beDVectorID :: Integer
                 , beDVectorShape :: d}
```

New BEVectors are created using the function new. The function copyIn copies a DVector from Haskell to the ArBB heap and there is a function copyOut to retrieve data from ArBB.

6.2 The language

The EmbArBB language is implemented as a set of library functions, operating on an expression datatype:

```
data Expr = Lit Literal
          | Var Variable

          | Index0 Expr
          | ResIndex Expr Int

          | Call (R GenRecord) [Expr]
          | Map  (R GenRecord) [Expr]

          | While ([Expr] -> Expr)
                  ([Expr] -> [Expr])
                  [Expr]

          | If Expr Expr Expr
          | Op Op [Expr]
```

Most of the ArBB functionality is taken care of by the Op constructor. The datatype Op used to represent operations has over 120 constructors, so only a selection is shown in figure 6. Having all these 120+ operations taken care of by one constructor in the expression datatype simplifies the implementation of the backend, since all of these operations are handled in a very similar way. The few remaining special ArBB capabilities such as loops, and function mapping and calling are handled by their own cases, discussed below. There are also some implementation specific details that result in their own constructors in the Expr type, namely Index0 for turning a zero dimensional DVector into a scalar, and resIndex that helps with implementing operations that have more than one result. An example of an such an operation is SortRank, which returns both the sorted result of an input vector and a vector of indices that specifies a permutation that would have sorted the input vector.

The Expr type also has constructors for the *call* and *map* functionality. To call a function means to apply it to input data. The map operation specifies elementwise application of a function over vectors, like NESL's *apply-to-each* [2], so it corresponds to Haskell's map and zipWith. Both Call and Map take a (R GenRecord). This is inspired by the delayed expressions that enable the implementation of vapply in Nikola [10]. The R is the reification monad used to create DAGs (directed acyclic graphs) from embedded language functions. These DAGs are part of a GenRecord that contains all information that the ArBB code generator needs to generate the ArBB function.

The While loop is represented using higher order abstract syntax, that is using functions to represent the condition and body. The state is represented by a list of expressions; this needs to be generalised somewhat in order to support loops with general tuples in the state. Something more structured than a list is needed for this.

The expression data type used in EmbArBB is untyped (not using a GADT). A typed interface to the language is supplied using the same phantom types method as used in *pan* [4] and many other Haskell embedded languages since then. The choice to use an untyped Expr datatype was based on the wish to keep the backend (ArBB code generation) as simple as possible.

```
-- Phantom types
type Exp a = E Expr
```

As an example, the operation addReduce, which reduces a vector across a specified dimension, is implemented as follows in the EmbArBB library:

```
addReduce :: Num a
          => Exp USize
          -> Exp (DVector (t:.Int) a)
          -> Exp (DVector t a)
addReduce (E lev) (E vec) =
  E $ Op AddReduce [vec,lev]
```

```
data Op =
   -- elementwise and scalar
     Add | Sub | Mul | Div | Max | Min
   | Sin | Cos | Exp
   ...

   -- operations on vectors
   | Gather | Scatter | Shuffle | Unshuffle
   | RepeatRow | RepeatCol | RepeatPage
   | Rotate | Reverse | Length | Sort
   | AddReduce | AddScan | AddMerge
   ...
```

Figure 6. ArBB scalar, elementwise and vector operations, which are handled by the `Op` constructor in the `Expr` datatype. This is just a selection from the more than 120 different operations ArBB provides.

6.3 Interfacing with Haskell and Code generation

The interface between ArBB and Haskell consists of the `DVector`, `NVector`, `BEDVector` and `BEScalar` types, the `capture`, and `execute` functions, and the ArBB monad with its `withArBB` "run"-function. This section describes what happens when the programmer captures an embedded language function, and when `execute` is called on a captured function.

Capture and execution of functions takes place in the `ArBB` monad, which manages state of type `ArBBState`:

```
data ArBBState =
  ArBBState
    { arbbFunMap :: Map.Map Integer
                            ArBBFun
    , arbbVarMap :: Map.Map Integer
                            VM.Variable
    , arbbUnique :: Integer }

type ArBBFun = (VM.ConvFunction, [Type], [Type])
```

This state contains a map from function names to ArBB functions and their input and output types. The `VM.ConvFunction` is how ArBB functions are represented by the ArBB-VM bindings. There is a map from vector and scalar IDs to their corresponding ArBB variables. The last item in `ArBBState` is an Integer that is used to generate new function names and variable IDs as the programmer captures more functions or creates new arrays on the ArBB heap. Now, the `ArBB` monad is defined as:

```
type ArBB a = StateT ArBBState VM.EmitArbb a
```

`VM.EmitArbb` is also a concept from the virtual machine bindings. It manages low-level functions (the `VM.ConvFunction` functions) and implements an interface to the low level ArBB-VM API. `EmitArBB` is the ArBB code generating monad from the arbb-vm bindings. For more or less everything that can be done with the arbb-vm bindings, there is a function of the form

```
f :: arg1 -> ... argn -> EmitArBB out
```

For example, for generating an operation node (such as +) in the ArBB IR, there is a function of type

```
op_ :: Opcode
     -> [Variable]
     -> [Variable]
     -> EmitArbb ()
```

while the function for generating a while loop in the ArBB IR has type

```
while_ :: (EmitArbb Variable)
        -> EmitArbb a
        -> EmitArbb a
```

The details of the translation are omitted for brevity, as the approach is standard.

When a function `f` of type

```
Exp tin1 -> ... -> Exp tinN -> Exp tout
```

is captured, it is first applied to expressions that represent variable names. For each of the inputs, `(tin1,..,tinN)`, a variable is created. The result is an expression (or expressions) representing the function `f`. On this expression, sharing detection is performed, and a directed acyclic graph (DAG) is created. The method of sharing detection used is based on the StableNames method [5].

Then, the code generation is implemented using a very direct approach; no extra optimisations or transformations are applied. This is a reasonable choice, since the whole point of ArBB is that the built-in JIT compiler knows and performs architecture specific optimisations. Sharing detection on the Haskell side makes sense because code generation for each node in the DAG results in at least two calls into the ArBB-VM API, which means going through the FFI and incurring the associated cost. Detecting the sharing already in the host language should give a smaller workload for the ArBB JIT compiler, thus reducing the time spent on JITing. It remains to be seen how important JIT cost will be in practice, however, as we expect it to be amortised over a large number of executions of the JITed code. We will need to conduct experiments with a suite of larger examples in order to decide if sharing detection on the Haskell side is worthwhile.

Applying `capture` to a function gives an object of type

```
type FunctionID = Integer

data Function i o = Function FunctionID
```

The `i` and `o` parameters to `Function` represent the input and output types of the captured function. As an example, capturing

```
f :: Exp (DVector Dim1 Word32)
   -> Exp (DVector Dim1 Word32)
```

results in an object of type

```
Function (BEDVector Dim1 Word32)
         (BEDVector Dim1 Word32)
```

This is just phantom types placed over a function name that is just a `String`, but it does offer a typed interface for the `capture` and `execute` functions.

The `execute` function that launches a captured function takes a `Function i o` object, inputs of type `i` and outputs of type `o`. The function name is looked up in the `ArBB` environment (the monad). The inputs and outputs are also looked up in the environment and then the function is executed.

```
execute :: (VariableList a, VariableList b)
         => Function a b -> a -> b -> ArBB ()
execute (Function fid) a b =
  do
    (ArBBState mf mv _) <- S.get
    case Map.lookup fid  mf of
      Nothing -> error "execute: Invalid function"
      (Just (f,tins,touts)) ->
        do
          ins <- vlist a
          outs <- vlist b

          liftVM$ VM.execute_ f outs ins
```

The `vlist` function goes through the heterogeneous list of inputs or outputs and looks up each of the elements in the `arbbVarMap`; the result is a Haskell list of `VM.Variable`. The function `VM.execute_` is part of the virtual machine API bindings, and corresponds directly to a C function in that library.

7. Benchmarks

In this section, matrix multiply, Sobel edge detection and an image blur algorithm are used as benchmarks in comparing sequential C++ code, ArBB, EmbArBB, and a Haskell library called Repa. Repa provides regular shape polymorphic arrays; it permits parallel execution of the resulting code, making use of multiple cores, but not of SIMD parallelism [8]. The Repa versions of the benchmarks used in the comparison come from the `repa-examples-3.2.1.1` package on Hackage.

The processor used for all measurements is a four core Intel Core-I7 930 at 2.80Ghz.

7.1 matrix-matrix multiplication

The matrix multiplication benchmarks consist of square matrices of sizes 256x256, 384x384, 512x512, 640x640 and 768x768. Figure 7 shows the runtimes for the four implementations being compared, using the best settings in numbers of cores or threads found by previous experiments. The sequential C++ and ArBB versions come directly from the ArBB distribution.

7.2 Sobel edge detection

In the Repa Sobel code, only the applications of the stencils are timed. This part is defined as follows

```
gradientX :: Monad m => Image -> m Image
gradientX img
        = computeP
        $ forStencil2 (BoundConst 0) img
              [stencil2| -1  0  1
                         -2  0  2
                         -1  0  1 |]

gradientY :: Monad m => Image -> m Image
gradientY img
        = computeP
        $ forStencil2 (BoundConst 0) img
             [stencil2|  1  2  1
                         0  0  0
                        -1 -2 -1 |]
```

A corresponding EmbArBB code to use in this comparison was implemented.

```
gx :: Exp (DVector Dim2 Float)
    -> Exp (DVector Dim2 Float)
gx = mapStencil
         (Stencil [-1,0,1
                  ,-2,0,2
                  ,-1,0,1] (Z:.3:.3))

gy :: Exp (DVector Dim2 Float)
    -> Exp (DVector Dim2 Float)
gy = mapStencil
         (Stencil [ 1, 2, 1
                  , 0, 0, 0
                  ,-1,-2,-1] (Z:.3:.3))
```

We compare these two examples, in which only the applications of the stencils are timed; we also time the complete Sobel implementation in EmbArBB (shown in section 5.5). The Sobel benchmark is run om images of sizes 256x256, 512x512, 1024x1024, 2048x2048 and 4096x4096. Again, Figure 8 shows that EmbArBB performs well. As we had expected, the Haskell embedding, once a function is captured, seems to impose little or no overhead compared to the C++ implementation.

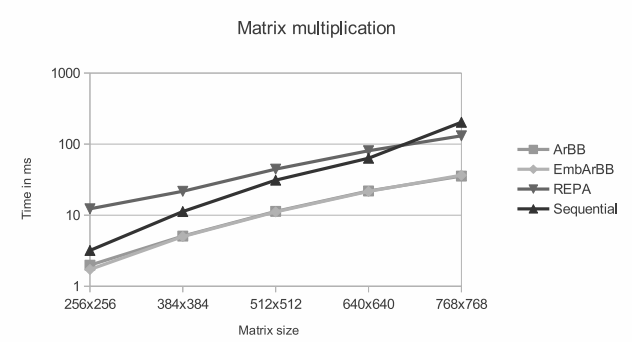

Figure 7. Shows in a log-scale the execution time of matrix-matrix multiplication comparing ArBB, EmbArBB, Repa and sequential C++. The ArBB and EmbArBB lines are indistinguishable.

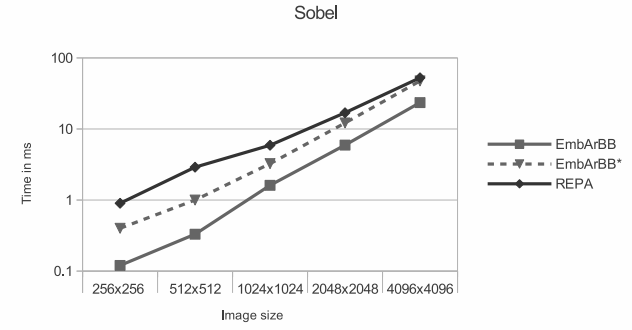

Figure 8. Shows in a log-scale the execution time of a key part of the sobel edge detection program. The chart compares EmbArBB to REPA and also displays for reference the execution times of the full sobel program as implemented in section 5.5, called *EmbArBB** in the chart.

7.3 Blur

The `blur` benchmark is performed using an algorithm similar to the example code in section 5.6 but with the following changes. The image used is in RGB color. This means that the stencil needs to be applied three times, once to each color plane. Also the image is converted into a form where each color intensity is represented by a `Double`.

Here as well as in the sobel case the Repa code from `repa-examples-3.2.1.1` times only the actual computational kernel (the application of the stencils) including conversion to Doubles. The corresponding EmbArBB part was broken out and timed separately as well. The chart 9 shows comparison of runtime for the key part of the computation but also adds the full execution time of the EmbArBB version (called *EmbArBB** in the chart). The full EmbArBB implementation of the blur filter including conversion into Doubles, image decomposition into R,G and B planes, application of the stencil and reconstructing a planar RGB image in the end.

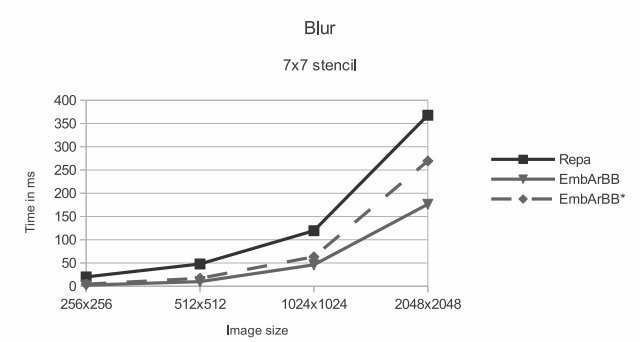

Figure 9. Shows the execution time of a 7x7 blur filter applied to images of various sizes.

	256x256	512x512	1024x1024	2048x2048
Repa 3x3	12	27	72	190
EmbArBB 3x3	1	2	20	77
EmbArBB* 3x3	2	9	33	153
Repa 5x5	13	32	87	254
EmbArBB 5x5	1	5	28	105
EmbArBB* 5x5	3	12	46	204
Repa 7x7	20	48	119	368
EmbArBB 7x7	2	10	46	176
EmbArBB* 7x7	4	17	63	270

Table 1. The table shows execution times (rounded to ms) for various image and stencil size combinations in both Repa and EmbArBB.

7.4 About the numbers

This section presents three benchmarks, two of which compare to Repa only and one that compares to Repa, ArBB in C++ and sequential C++ code. In all of these comparisons, JIT compilation time is excluded.

The comparison to ArBB in C++ shows that the Haskell embedding does not impose any extra overhead (at least not in this benchmark); this matched our expectations. More comparisons to the C++ version of ArBB are needed to confirm these first impressions about overhead in the Haskell embedding. If future directions of EmbArBB development develop techniques (such as size inference) that impose a runtime overhead, then the comparison in execution to the C++ version becomes more important.

The comparisons to Repa all show that the performance of EmbArBB compares favourably. This can be attributed to the way in which ArBB's developers at Intel have incorporated vectorisation and threading.

8. Future Work

The C++ embedding of ArBB allows for dense containers of structs in some cases. The operations on vectors supplied by the ArBB virtual machine are exclusively over vectors of scalar types. So the C++ embedding must be performing a AOS to SOA (Array of Struct to Struct of Array) transformation. The Haskell embedding does not implement any similar transformation. This is an important addition that would for example make implementing functions on complex numbers easier.

We stress that this paper presents first steps in the implementation of EmbArBB. Our benchmarks, while promising, are very limited. We must devote effort to developing a suite of interesting, larger data parallel programs for use in benchmarking EmbArBB. It is particularly important to explore the nested vectors, and the effects of the limitation to one level of nesting in ArBB. Those parts of ArBB that support nested vectors seem to be less well developed than those supporting dense vectors, as evidenced by the sample applications distributed with ArBB, none of which uses nesting. We expect ArBB to become more complete, and perhaps we will be able to contribute interesting examples both in the C++ and Haskell embeddings.

Having exercised EmbArBB more thoroughly, we will assess the results of the benchmarking and experiments with programming idioms, and decide on future research directions. We expect to focus on ways to provide users with an interface that is more functional in style than the current C++ oriented one.

9. Conclusion

We have shown that Intel's Array Building Blocks (ArBB) provides an interface that is well suited to functional programming. The programs are quite close to mathematical specifications, in the style of NESL [2]. We have only just completed the embedding of the part of ArBB that deals with nested vectors, and we need to tackle many more case studies. In our case studies using dense vectors, ArBB seems to do a good job of efficiently using parallel hardware resources – both cores and vector units. By embedding ArBB, we can, with little implementation effort, provide quite an attractive data parallel programming language in Haskell. Our benchmarks are preliminary and small, but they show very good performance. Once we have completed the ArBB embedding, we will have an interesting platform on which to experiment with and develop new programming idioms that exploit the fact that we have a data parallel programming language embedded in an expressive, strongly typed host language. We feel that work in this area (as distinct from implementation methods) is overdue. We would be happy to receive suggestions for interesting case studies or collaborations.

Acknowledgments

Svensson was first introduced to ArBB while on a three month internship at Intel during 2011. Thanks go to Ryan R. Newton for inspiring supervision during that internship.

This research has been funded by the Swedish Foundation for Strategic Research (which funds the Resource Aware Functional Programming (RAW FP) Project) and by the Swedish Research Council.

References

[1] E. Axelsson, K. Claessen, G. Dévai, Z. Horváth, K. Keijzer, B. Lyckegård, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: A domain specific language for digital signal processing algorithms. In *8th ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2010)*, pages 169–178. IEEE Computer Society, 2010.

[2] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3), 1996.

[3] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3. URL `http://doi.acm.org/10.1145/1926354.1926358`.

[4] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. URL `http://conal.net/papers/jfp-saig/`.

[5] A. Gill. Type-Safe Observable Sharing in Haskell. In *Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, 09/2009 2009. URL `http://www.ittc.ku.edu/csdl/fpg/sites/default/files/Gill-09-TypeSafeReification.pdf`.

[6] J. Hughes and M. Sheeran. Teaching parallel functional programming at Chalmers. In *presentation at Trends in Functional Programming in Education, Workshop associated with Conf. on Trends in Functional Programming, St. Andrews*, 2012.

[7] Intel. Intel(r) Array Building Blocks Virtual Machine Specification. `http://software.intel.com/sites/whatif/arbb/arbb_vm.pdf`.

[8] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. URL `http://doi.acm.org/10.1145/1863543.1863582`.

[9] B. Lippmeier, M. M. T. Chakravarty, G. Keller, R. Leshchinskiy, and S. P. Jones. Work Efficient Higher-Order Vectorisation. `http://www.cse.unsw.edu.au/~chak/papers/replicate-tr.pdf`, 2012. ICFP'12.

[10] G. Mainland and G. Morrisett. Nikola: Embedding Compiled GPU Functions in Haskell. In *Proceedings of the third ACM Haskell symposium*, pages 67–78. ACM, 2010. ISBN 978-1-4503-0252-4. URL `http://doi.acm.org/10.1145/1863523.1863533`.

[11] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's array building blocks: A retargetable, dynamic compiler and embedded language. In *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8.

[12] M. Sheeran. Generating fast multipliers using clever circuits. In *Int. Conf. on Formal Methods in Computer Aided Design (FMCAD)*, volume 3312 of *LNCS*, pages 6–20, 2004.

[13] M. Sheeran. Functional and dynamic programming in the design of parallel prefix networks. *J. Funct. Program.*, 21(1):59–114, 2011.

[14] B. J. Svensson and R. Newton. Programming Future Parallel Architectures with Haskell and ArBB. `http://faspp.ac.upc.edu/faspp11/pdf/faspp11-final12.pdf`, 2011. Presented at the workshop: Future Architectural Support for Parallel Programming (FASPP), in conjuction with ISCA '11.