AN INTERFACE FOR EMBEDDING THE BLAS IN HASKELL

by

BOSCO NDEMEYE

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

September 2020

THESIS APPROVAL PAGE

Student: Bosco Ndemeye

Title: An Interface for Embedding the BLAS in Haskell

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Boyana Norris	Chair

and

Kate Mondloch	Interim Vice Provost and Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded September 2020

© 2020 Bosco Ndemeye

THESIS ABSTRACT

Bosco Ndemeye

Master of Science

Department of Computer and Information Science

September 2020

Title: An Interface for Embedding the BLAS in Haskell

Scientific algorithms have been built on top of linear algebra subprograms, historically implemented in languages such as C/C++ or Fortran, to optimize their performance, sometimes at the cost of their conciseness. Recent work in these languages has sought to address the problem of optimizing the implementations of the subprograms through the use of domain-specific languages (DSLs). However, it has been shown that using various techniques such as fusion, concise yet optimal implementations of array computation DSLs in functional languages — such as Haskell— are possible. Consequently, we investigate an interface to a library that supports subprogram computations in Haskell. We apply the *delayed fusion* technique to separate data stored in memory and their delayed versions, providing the user with the option to force computation as they deem fit. We present implementations of several subprograms, abstracting over the choice of data sparsity layouts in memory.

CURRICULUM VITAE

NAME OF AUTHOR: Bosco Ndemeye

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
Hendrix College, Conway, AR

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2020, University of Oregon
Bachelor of Arts, Computer Science, 2018, Hendrix College

AREAS OF SPECIAL INTEREST:

Programming Language Foundations
Parallel Computing
High Performance Computing

PROFESSIONAL EXPERIENCE:

Graduate Research Aide, Argonne National Labs, Summer 2020

Graduate Research Fellow, University of Oregon, Summer 2019

Graduate Teaching Fellow, University of Oregon, 2018 – 2019

Undergraduate Research Assistant, Hendrix College, Summer 2017

TABLE OF CONTENTS

Chapter Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

A collection of small, pure[1] linear algebra functions and their compositions, form the backbone of many computational algorithms. Due to their wide applicability, specifications of such common subprograms as the Basic Linear Algebra Subprograms [7] have been put forth to facilitate application optimizations, and establish a standard interface. The main idea is that larger application programs are built up from one or more of these subprograms as well as their compositions; implying that optimizing an individual, or sequences of subprogram calls, should permeate a bump in performance throughout applications using them. This makes the subprograms well suited for a purely functional implementation. Works such as Built-To-Order BLAS (BTO) [18] view optimizing these subprogram subprograms as a compiler construction problem. By designing a Domain Specific Language (DSL) that reduces a given subprogram—specified in MATLAB-like syntax—to a form suitable for such different optimizations as fusion and data parallelism, the authors are able to generate C code that sometimes performs better than hand-optimized versions of the same subprograms or sequences of subprograms. This indicates that type-directed compiler construction techniques can and should still be used to tackle such problems.

The research question that still remains, however, is that of whether it is absolutely necessary to compile down such intrinsically functional formulations into mostly imperative languages such as C, C++, or Fortran. Because an embedding of the subprograms into a purely functional language seems to imply unreasonable space complexity— due to the fact that every function call in such a language

[1] A pure function is one which will return the same output given the same input every time, without any side effects.

generates an intermediary structure which subsequent functions in the composition chain can act upon—it is tempting to throw away the idea. However, advancements in purely-functional-language-compiler-optimizations seem to indicate that there are working solutions around the space complexity issue. Works such as [15], and [17] tackling multi-dimensional array computations in the purely functional and lazy language Haskell for example, seem to indicate that its most widely used compiler—the Glasgow Haskell Compiler (GHC)—has matured into a powerful compiler, ready to compete with imperative giants on the performance stage. Therefore, it is not clear whether an embedding of the subprograms into a purely functional language would not be just as beneficial. Moreover, the fact that GHC has a backend for LLVM IR—which C/C++ *clang* implementations also support—is yet another reason why an embedding of the BLAS in Haskell would be a worthwhile investment. It would provide a foundation for not only Haskell linear algebra applications, but also similar C/C++ projects.

BTO is relatively successful in providing data parallelism as well as subprogram fusion. Coincidentally, fusion has been a hot research topic in the GHC community, and [15] has demonstrated that Haskell is indeed capable of handling parallel computation. In addition to this, it is formulations that use Haskell's strong type system that popularized the Embedded Domain Specific Language technique [14]. Consequently, it seems reasonable to ask whether BTO's goals for the BLAS couldn't be achieved by an embedding in Haskell.

This thesis' goal is to establish a starting point by designing a generic interface for serial implementations of the BLAS. This means that we don't seek to provide explicit parallelism in the subprograms', nor do we claim performance improvements over existing implementations in other languages. Instead, we focus on designing an interface that would allow for fusing intro-subprogram instructions

as well as subprogram compositions. Moreover, the interface is designed to be agnostic in the memory layout of the matrices' used thereby providing a foundation for application specific space optimizations that takes advantage of matrices' different sparsity patterns. Haskell's type system allows us to expose said sparsity choice in the type, thereby leaving the choice of a matrix's storage format completely in the hands of the library's client.

The specific contributions of this work can be summarized as follows:

1. A generic interface for matrices and vectors that encodes the choice of matrix memory representation as well as the underlying elements[2].
2. A review of the *fusion* optimizations problem in a functional context.
3. An implementation of three matrix storage formats as examples of how the library supports different memory layouts, and can be extended.
4. An implementation of several subprograms using our interface.
5. A report of benchmark results on several real-world sparse matrices with a relatively small number of non-zeros.

The rest of this thesis is structured as follows. Section II provides the background necessary by reviewing Haskell's type system, sparsity formats as well as the fusion optimization. Section III dives into the details of the implementation, while Section IV provides benchmark results for three subprograms (*axpy*, *gemv*, and *atax*) on several matrices from the Suite Sparse Collection [6]. Finally, Section V provides a summary of this work and directions for future work.

[2]The techniques used (i.e using type indices to guide computation) is well known in Haskell. We have just applied to the specific context of the BLAS.

CHAPTER II

BACKGROUND

In the following sections, first we provide a brief overview of Haskell's type system summarizing features we use to achieve parametric polymorphism[1] for the subprograms. Second, we review sparse data compression formats we use as our running examples, and last, we explore different techniques used to achieve fusion in GHC. Section III puts all of this together as we discuss our implementation.

2.1 Haskell Type System Brief Overview

The complete features of Haskell's type system are beyond the scope of this thesis [2]. This section provides a brief over-view of only those features we use as part of our interface. These include *boxed vs unboxed types, concrete vs abstract types, algebraic data types or ADTs,* and *type classes* as well as their *associated data types* or *type families.*

2.1.0.1 Boxed VS Unboxed Types. Haskell distinguishes between *boxed* and *unboxed* types. Unboxed types are the primitive types corresponding to the normal C-like types such as *int*, *double*, *float*, etc. They are thus more suitable for High Performance Computing (HPC) and they cannot be defined in Haskell itself. The occupants of these types are represented by values, as opposed to boxed types which are represented by pointers to objects in the heap. Objects similar to C arrays are known as *boxed unlifted*[3] *types*, and they are represented by a pointer to the primitive array in the heap. This contrasts them from *boxed lifted types* whose pointers, in GHC, point to heap *thunks*[4] or *indirections.* Therefore correct

[1] The use of the same code for different types

[2] For features not discussed, interested readers are referred to [4].

[3] The difference between a *lifted* and an *unlifted* type is that undefined values ($\perp$) can inhabit the former but not the latter.

[4] In its simplest definition, a thunk is a representation of an unevaluated expression.

use of *unboxed* or other *unlifted* types (such as primitive arrays) can yield better performance than their *boxed* counterparts. However, their correct use is governed by a strict list of rules [2], which can make them difficult to program. Thus, high-level Haskell programming generally deals with boxed types.

2.1.0.2 Concrete vs Abstract Types. *Abstract* data types involve type variables while *concrete* ones do not. For example:

```
Integer, Maybe Bool, Tree Double
```

are all *concrete*, while

```
data Maybe a = Nil | Just a
```

is *abstract* because type variable *a* hasn't been specialized yet. Therefore, our implementation will mostly use *abstract* data types due to the fact that they are more suitable for generic interface design.

2.1.1 ADTs. Types in Haskell can be combined together to form a larger type through an *algebra* of *sum*s and *product*s. The resulting type is known as an *Algebraic Data Type (ADT)*. *Sums* can be thought of as C *union*s with tags and products as C *struct*s with tags[5]. For example, a type that represents all days in a week:

```
data Day = Monday | Tuesday | Wednesday | Thursday | Friday
```

is a *sum* type, whereas the type of points of a coordinate plane with integers on the x and y axes is a *product* type:

```
data Point = P Int Int
```

[5]Because Sums and Products can involve more complex types, this analogy is a loose one.

These "tags" are known as constructors, and they can be thought of as functions that accept values of the smaller types to produce values of the more complex types. For example, `P` can be thought of as having the following type:

```
P :: Int -> Int -> Point
```

This type is read as: `P` accepts two values of type `Int` and produces one value of type `Point`. Sums and Products can be combined to form even larger type. In addition, Haskell supports the declaration of recursive data types. For example, the type of *Peano natural numbers* can be defined as

```
data Nat = Zero | Succ Nat
```

Parametric polymorphism presents itself through the use of type variables as parameters of type constructors. For example, the type of generic "tagged unions" of two types can be declared as:

```
data Either a b = Right a | Left b
```

Thus, `Either` is a *type constructor* with parameters `a` and `b` whereas `Right` and `Left` are *data* or *value constructors*.

Specialized manifest representation of data stored according to different storage formats, use ADTs in our implementation of Section III.

2.1.2 Type Classes. Haskell supports *ad hoc polymorphism*, or *function overloading* through the use of *type classes*. A type class declaration is a parameterized interface that lays out the type signature of functions in the class, whereas an `instance` declaration of the class provides implementations of the interface. For example a type class `Show` that defines how to render a value to its string representation can be declared as:

```
class Show a where
  show :: a -> String


instance Show Nat where
  show Zero = "0"
  show (Succ n) = "Succ (" ++ show n ++ ")"


instance Show Point where
  show (P x y) = "(" ++ show x ++ "," ++ show y ++ ")"


show :: Show a => a -> String
```

One implication of this is that calling the `show` function on any value that is an element of a type with a `Show` instance is guaranteed to type check. GHC extends the overloading idea by not only supporting *functions*, but also *data overloading.* This is supported through the use of *associated data types* and *type families* [8].

For example, [16] defines the class of generic map keys along with its associated data type of generic finite maps as follows:

```
class Key k where
    data Map k :: * -> *
    empty :: Map k v
    lookup :: k -> Map k v -> Maybe v
```

The *star* ($*$), represents the *kind*[6] of all types that can possess run-time values. Thus, we can declare an instance `Either` of the `Key` class as follows:

[6]Just like we can talk about a type as containing values, we can talk about a kind as containing types.

```
instance (Key a, Key b) => Key (Either a b) where
  data Map (Either a b) elt = MS (Map a elt) (Map b elt)
  empty = MS empty empty
  lookup (Left k) (MS m _) = lookup k m
  lookup (Right k) (MS _ m) = lookup k m
```

The important concept to note in the above example being that the `k` parameter of the `Map` data type was abstract in the class declaration but has been specialized to `Either` in the instance declaration. This lets the generic operations of the class (`empty`, and `lookup`) be overloaded in the `k` parameter of the `Map` type, allowing clients to define the specific form their data is allowed to take. Our own `Sparse` type class definition of Section 3.1 defines a type class along with its associated `SparseData` data type; which is what allows us to abstract over the memory layouts of matrices.

2.2 Sparsity

The sparsity of a matrix can be thought to be a purely economic issue[12]. For computing purposes, a matrix is only considered sparse if taking advantage of its zeros reduces either the application's run-time or its storage space. Consequently, there exist multiple sparse matrix storage formats, each suited for a different type of application. In this section, we review three such formats: the coordinate (**COO**), the compressed sparse row (**CSR**), as well as the ellpack (**ELL**) format.

For example, consider the following matrix.

$$A = \begin{pmatrix} 13 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 44 & 0 & 0 & 0 & 0 \\ 0 & 0 & 54 & 53 & 72 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 83 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 92 \end{pmatrix}$$

1. **The Coordinate Format (COO):**

   Perhaps the most intuitive format, **COO** uses three arrays to store a sparse matrix. As the matrix is traversed in row major order its non-zeros are stored in A_{nz}, their corresponding row indices in A_r, while the corresponding column indices are stored in A_c. Therefore, A will be stored as:

   $$A_{nz} = 13, 2, 3, 44, 54, 53, 72, 83, 92$$

   $$A_r = 0, 0, 1, 1, 2, 2, 2, 3, 4$$

   $$A_c = 0, 1, 1, 2, 2, 3, 4, 5, 6$$

   If a matrix contains nz non-zeros, the **COO** format will use $3nz$ space to store it. The row major order traversal insures that both A_r and A_c are column-sorted, facilitating individual elements look up times.

   In addition to the COO format being good for easy construction of sparse matrices, conversion to and from the format to other schemas such as CSR is fast. This convenience makes COO the default format for popular scientific computing packages such as SciPy [5].

2. **The Compressed Sparse Row Format (CSR):**

   Similar to the **COO** format, the **CSR** format uses three arrays to store a sparse matrix, A_{nz} for non-zeros, A_r for rows, and A_c for its columns. In CSR

however, A_r is compressed: If k is an index in A_{nz}, and $A_{nz}(k) = A_{ij}$ where A_{ij} is the element at row i and column j in A, then $A_r(i) \leq k \leq A_r(i+1)$. By convention $A_r(0) = 0$. Consequently, matrix A will be stored as:

$$A_{nz} = 13, 2, 3, 44, 54, 53, 72, 83, 92$$

$$A_r = 0, 2, 4, 7, 8, 9$$

$$A_c = 0, 1, 1, 2, 2, 3, 4, 5, 6$$

Thus, if a sparse matrix contains nz non-zeros, and is of height h, the **CSR** format will use $2nz + h + 1$ space to store it. Compressing A_r allows for efficient row slicing, which in turn allows for fast matrix-vector multiplications as well as an efficient element to element operations such as addition. Our implementation expounds on this topic in section III of this thesis.

3. **The Ellpack Format (ELL):**

The **ELL** format compresses a sparse matrix into two arrays. The first stores the matrix's non-zeros, while the second stores their corresponding column indices. Zero values are padded at the end of rows whose non-zeros are less than the maximum number of non-zeros per row. For example, A will be

stored as:

$$A_{nz} = \begin{pmatrix} 13 & 2 & 0 \\ 3 & 44 & 0 \\ 54 & 53 & 72 \\ 83 & 0 & 0 \\ 92 & 0 & 0 \end{pmatrix}$$

$$A_c = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 2 & 0 \\ 2 & 3 & 4 \\ 5 & 0 & 0 \\ 6 & 0 & 0 \end{pmatrix}$$

Consequently, if M is the maximum number of non-zeros per row, and the sparse matrix has h rows, $2hM$ space will be used to store it. Assuming w columns for the matrix, this is still less than wh for most sparse matrices.

In the discussion of Section III, we present how each of these formats is encoded as a type index which a client can specialize according to the needs of their application.

2.3 Fusion

As alluded to in the introduction, [20] and [18] seek to tackle the problem of fusing subprogram call sequences in the context of imperative languages. However, fusing sequences of subprogram calls in functional programming is a different problem. Without any compiler optimizations, every operation in the paradigm generates an intermediary structure and thus leads to unreasonable space usage. However, optimizations to eliminate these intermediaries have been studied for years and this Section reviews several such systems in the context of GHC.

Figure 1. map/map rule pragma

```
{-# RULES
"map/map" forall f g xs. map f (map g xs) = map (f.g) xs
#-}
```

[19] presents a mechanism through which equivalence laws could be instructed to the compiler to transform programs, without modifying the compiler itself. The work was encapsulated in a new *RULES* pragma that lets library writers use domain-specific knowledge about their work to define such laws using standard Haskell notation. Among the first potential applications presented for the rules was list fusion, capturing well-known laws such as that of Listing 1.

2.3.1 build/foldr. [13] identifies a single such rule general enough for the purposes of a list library by "*standardising the way in which lists are consumed, and standardising the way in which they are produced.*"

As an algebraic data type, a list is constructed recursively by a *nil* ([]) constructor and a *cons* (:) constructor. For example, a list such as $[1, 2, 3, 4]$ is constructed as $(1 : 2 : 3 : 4 : [])$. Therefore, standardizing the consumption of lists boils down to specifying how consumption works for each constructor. This done by use of the standard function `foldr` (Listing 2).

The opposite to `foldr`, is a *build* (Listing 2) function which works by replacing all the occurrences of the *cons* and *nil* constructors in a *function-representation* of a list, such as `func1234` (Listing 2), with their data equivalents.

The universal quantifications in the type signature of `build`, ensure that its input is indeed a *function-representation* of a list and not perhaps some more

Figure 2. foldr, build definitions

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr g y []     = y
foldr g y (x:xs) = g x (foldr g y xs)

func1234 = \cons nil ->
              1 `cons`
              (2 `cons`
              (3 `cons`
              (4 `cons` nil)))

build :: forall a.
         (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```

Figure 3. Lambda-list to Data-list illustration

```
lst1234 = func1234 (:) []
        = (\cons nil ->
                     1 `cons` (2 `cons`
                              (3 `cons` (4 `cons` nil)
                              ))) (:) []
        = 1 : 2 : 3 : 4 : []
```

complex function that involves manipulating its sub-components (such as reversing the list, for example).

The **build/foldr** rule thus, works by omitting to construct the intermediary list generated by applying `build` to its *function-representation*, if that same list is immediately consumed by `foldr`. That is

```
foldr f z (build g) = g f z
```

Figure 4. zip as limitation to build/foldr

```
1 zip xs ys = foldr f (\_ -> []) xs ys
2         where f x g []     = []
3               f x g (y:ys) = (x, y) : g ys
```

Therefore, as long as a list is constructed with **build** and is immediately consumed with **foldr**, all compositions of these functions can be fused by the **foldr**/**buildr**. This encompasses functions such as `sum`, `and`, `map`, `++`, as well as expressions such as `[x..y]`. However, limitations for the system exist, and among them are functions like `zip`.

Although, as Listing 4 illustrates, `zip` can be specified in terms of `foldr`, by consuming `xs` and constructing a function which takes `ys` as an argument and produces the zipped list, even if both `xs` and `ys` are constructed with `build`, only `xs` can be eliminated by the system since `ys` is never directly used by `foldr`.

2.3.2 destroy/unfoldr. The **destroy/unfoldr** system [21] seeks to solve the shortcomings of the previous system, introduced by `zip`-like functions, by shifting the focus from how a list is consumed, to how it is constructed. From the list's "folding" to its "unfolding":

Much like `foldr` specifies how to consume consecutive elements of a list to build a new value, `unfoldr` (Listing 5), expresses how to construct consecutive list elements when a "seed" value is provided.

Thus, the fusion system's goal is to find a function that composes with `unfoldr` to eliminate its resulting list without modifying the underlying semantics of the composition. As [21] demonstrates, `destroy` (Listing 6) is that function.

Figure 5. `unfoldr` definition

```
unfoldr :: (b -> Maybe (a, b)) -> b -> [a]
unfoldr f b = case f b of
    Nothing       -> []
    Just (a, b1) -> a : unfoldr f b1
```

Figure 6. `destroy` definition

```
destroy :: (forall a.
            (a -> Maybe (b, a)) -> a -> c)
            -> [b] -> c
destroy g xs = g listpsi xs
   where listpsi :: [a] -> Maybe (a, [a])
         listpsi []     = Nothing
         listpsi (a:as) = Just (a, as)
```

Thus the same reasoning used in Section 2.3.1 results in the **destroy/unfoldr** equivalence rule written down as:

```
destroy g (unfoldr psi e) = g psi e.
```

This fusion[7] system is strong enough to optimize compositions of multiple standard functions including some of those that stumped the **build/foldr** systems, such as functions with accumulating parameters and zip-like functions. However, it is incapable of handling functions with nested lists such as `concatMap` and is inefficient in its handling of `filter`-like functions.

2.3.3 Stream Fusion. Similar to **destroy/unfoldr**, **stream fusion** achieves its goal by focusing on list co-structures or `streams`. However, it sets itself

[7]To complete the fusion system, [21] defines a second rule: *destroy/destroy*. Readers are referred to the original work for more details about the interplay between the two rules and their use to fuse most list operations

Figure 7. `Stream` data type

```
1 Stream :: forall s  a. (s -> Step a s) -> s -> Stream a
2 unfoldr :: forall s a. (s -> Maybe ~a, s)) -> s -> [a]

3 data Stream a  = exists s. Stream (s -> Step a s) s

4 data Step a s = Done | Yield a s | Skip s
```

apart by explicitly defining a `Stream` data type and providing functions to convert to and from lists to the new data type. List operations are thus re-implemented as `Stream` operations that result in `Stream` values, in the confidence that constructors for the values will be eliminated by GHC's general-purpose optimizations.

The constructor for the `Stream` data type (Listing 7) is similar in signature to `unfoldr`, relying on a provided initial state and a function to produce subsquent elements.

`Stream`'s algebraic data type is defined as in Listing 7. The two functions `stream` and `unstream` used to inter-convert between the list structure and the stream structure are defined as in Listing 8.

Although the **stream fusion** system is designed to allow general-purpose compiler optimizations on operations involving the `Stream` data type, it also provides one rewrite rule that helps eliminate intermediate list conversions. For example, in the composition of the two list `map`'s of Listing (10), the rule gets applied resulting in the removal of an intermediate list structure.

Since by design all `Stream` producers are non-recursive, by transforming a pipeline of list function compositions into their equivalent `Stream` versions, the compiler is allowed to use existing general-purpose optimizations that eliminate intermediate `Step` constructors, thereby achieving the proposed fusion. Our

Figure 8. `stream`, `unstream` definition

```
stream :: [a] -> Stream a
stream xso = Stream next xso
   where
     next []     = Done
     next (x:xs) = Yield x xs

unstream :: Stream a -> [a]
unstream (Stream nexto so) = unfold so
    where
      unfold x = case nexto s of
         Done     -> []
         Skip s' -> unfold s'
         Yield x s' -> x : unfold s'
```

Figure 9. `stream`/`unstream` rule pragma

```
{-# RULES
"stream/unstream" stream . unstream = id
#-}
```

Figure 10. `stream`/`unstream` illustration

```
(map f) . (map g) = unstream . map_s f
                             . stream   . unstream
                             . map_s g . stream
-- becomes
(map f) . (map g) = unstream . map_s f
                             . map_s g . stream
```

Figure 11. Delayed lists example implementation

```
1 data FList a = L Int (Int -> a)

2 map :: (a -> b) -> FList a -> FList b
3 map f (L len g) = L len (f . g)

4 zip :: (a -> b -> c) -> FList a -> FList b -> FList c
5 zip f (L len g) (L _ h) = L len (\i -> f (g i) (h i))

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

implementation relies on vectors from the vector library [9] which fuses its operations by stream fusion.

2.3.4 Delayed Fusion. As another option, a popular work-around to generating intermediate structures is to avoid generating them at all, by manipulating their index transformations instead. For example, rather than representing a list as an algebraic data type with an "nil" and a "cons" constructor, a pair comprised of a pure function whose domain equates the list index range and the length of the list could be used, as Listing (11) illustrates. Using this representation, standard functions such as `map` and `zip` operate by composing their appropriate argument functions with the list's indexing function, and as a consequence, fusion becomes nothing more than function composition.

[15] generalizes this pattern of programming to multi-dimensional arrays and discusses different shortcomings of this approach, among them `sharing` (Listing 12).

In the example of 12, computing x will cause f to be applied to the appropriate element of y. But, considering that f might be an expensive operation, an implementation such as the above that calls f more times than necessary should

Figure 12. Problems with sharing when delaying

```
1 let x = map f y in zip3 x x x

2 -- introduce force
3 let x = force $ map f y in zip3 x x x
```

be avoided. To get around this problem, a function that `force`'s computation of the delayed form to its raw data equivalent is usually provided. As a result, in the example of Listing (12), f is only called once per element of y; speeding up the program by a factor of three. As described in section III, our own implementation follows the design presented in [15] and delays computation to avoid unnecessary intermediaries. This means that we also provide a mechanism to `force` any sequence of subprograms, thereby gaining subprogram fusion by leaving the choice of when to perform computation in the hands of the application writer.

CHAPTER III

METHODOLOGY AND IMPLEMENTATION

Because the matrix is at the core of computational linear algebra, existing systems such as MATLAB center around this structure in order to provide direct adaptations of subroutines from high-level linear algebra specifications such as LINPACK[1] [10, 12]. This means that standard linear algebra operations like addition and constant multiplication are expressed as high-level collective operations in order to provide clear implementations that stay faithful to their specifications. However, it is also well known that these operations can be implemented in terms of the `zipWith`, and `map` functions at the core of functional programming. This section describes the design of an interface to such an implementation. The main contributions of the design is that, in addition to abstracting over the memory layout of the matrices as well as their elements, all operations are expressed as pure functions; meaning that all fusion is nothing more than function composition[2]. Using Haskell's type system, our interface represents matrices in terms of their indexing functions (in `delayed` form) so that the implementation of an operation such as $A\vec{x} + p\vec{y}$ is nothing but a chain of function compositions. However, since eventually these functions need to be computed into actual data, we also provide an overloaded operation (`force`) that evaluates a matrix's function representation into its memory (`manifest`) representation.

This separation between delayed representations versus manifest representations of data, means that optimizations such as *data parallelism* can be added to the system by only modifying the `force` function.

---

[1] LINPACK relies on the BLAS for clarity and efficiency

[2] Repa[15] does this in the context of dense array computations, but as far we know, we are the first to extend this design to tackle BLAS computations in Haskell

Figure 13. Sparse matrix, vector definition

```
type SVector a = (Int -> a, Int)
data RepIndex = U | D
class (Unbox a, Num a, Eq a)
      => Sparse r (ty :: RepIndex) a where
    data SparseData r (ty :: RepIndex) a :: *
    index :: SparseData r ty a -> (Int, Int) -> a
    dim   :: SparseData r ty a -> (Int, Int)
```

3.1 Data Representation

Similar to [15], the matrices and vectors in our implementation are `delayed` by default. This means that all computation is expressed as a chain of function compositions until the user decides they want to `force` the delayed representation into what [15] calls its *manifest* representation. For manifest data storage, we employ *unboxed vectors* provided by Haskell's vector library [9], to reap the various benefits it provides such as constant array indexing as well as stream fusion. We use *type classes* for operation overloading, and their *associated data types* feature [8] for data type overloading. The distinction between *manifest* and `delayed` arrays is achieved though the use of datatype kind promotion using GHC's `-XDataKinds` extension [3]. Listing 13 defines our vectors' type, as well as the type class capturing our generic matrices.

Delayed vectors (`SVector`) are represented as a pair of the manifest vector's indexing function and its length (similar to Section II's `FList`), and generic matrices are represented as an *associated data type* of a *Sparse* type class with three type indices: `r` for sparsity formats, `ty` to indicate whether the matrix is in its *delayed* (`D`) or *manifest* (`U`) form, and `a` for the underlying element type.

Consequently, a *delayed* matrix with `Double` elements, stored using the `COO` format will be specified as `SparseData COO D Double` whereas a *manifest* matrix with `Int` elements, stored using the `CSR` format will be specified as `SparseData CSR U Int`.

To provide an instance of the `Sparse` class means to specify a *manifest* representation for a given `r` index, a *delayed* representation for a `ty` index, a function to retrieve an element stored at a given row and column (`index`), and a function to retrieve the dimensions of the matrix [3] (`dim`).

3.2 Vector Operations

We discuss vector operations first, as they are relied upon by the rest of the implementation: Conversions between *unboxed vectors* (*Vector*) and their *delayed* counterparts (`SVector`) are provided by Listing (14). To convert a *delayed* vector to its *unboxed* form means to apply its indexing function for all indices in the range of the vector's length. This is achieved by use of the `generate` function provided by the vector library. Conversely, to delay an *unboxed* vector means to pair up its indexing function and its length. The retrieval of the indexing function—through *partial application* of the vector's indexing operation to the vector—and the length, both take place in constant time courtesy of the vector library.

The `map` and `zipWith` vector functions are similar to those provided in Section II. Scalar multiplication, element-wise addition, subtraction, as well as the dot product of vectors can thus all, also be defined as the listing illustrates.

3.3 Delayed Instance

Because there are practically no distinctions between *delayed* versions of matrices with different memory layouts (they are all functions that return the

[3] As [15] illustrates, it is possible to encode the dimensions of the matrix in the type thereby preventing unnecessary runtime checks, and maybe providing a performance boost. This is a feature we are considering for future work.

Figure 14. Delayed vector operations

```
1  to_vector (f, len) = generate len f

2  from_vector vec =  let len = length vec
3                     in ((!) vec, len)

4  vmap f (g, len) = (f . g, len)

5  vzipWith f (g, len1) (h, len2) = (\i -> f (g i) (h i)
6                                     , len1)

7  (!+!)   = vzipWith (+)
8  (!-!)   = vzipWith (-)
9  (!*!) x = vmap (* x)

10 -- this takes twice as long
11 -- as summing up unboxed vectors
12 vsum = sum . to_vector

13 (!.!) v1 = vsum . vzipWith (*) v1
```

Figure 15. Delayed matrix *Sparse* instance

```
1 import qualified Data.Vector as B
2 instance (Unbox a, Num a, Eq a) => Sparse r D a where
3     data SparseData r D a = SDelayed (Int, Int)
4                                      ((Int, Int) -> a)
5     index (SDelayed _ f) (r, c)     = f (r, c)
6     dim (SDelayed (h, w) _)         = (h, w)
```

element stored at a given row and column), the r index of the type class should be abstracted over to provide a generic *delayed* instance that works for all sparsity formats. As Listing 15 shows, we do this by defining: (1) The data representation as a product type with a constructor that accepts the dimensions of the matrix, and the indexing function that takes a row and column as a parameter and returns the corresponding non-zero element; (2) The indexing operation as directly applying the indexing function stored in the constructor; and (3) By directly returning the matrix's dimensions for the corresponding operation of the class.

3.4 Polymorphic Operations

This section provides implementations of a number of overloaded operators which are later used for generic implementations of larger linear algebra operations (see Section 3.7). All of these operations leave their results in delayed form to facilitate fusion through function composition, while also leaving actual computation responsibilities to one function (`force`). This is partly to make it convenient for extension with data-parallelism.

3.4.1 Conversion. An operation that converts manifest data representations to their delayed equivalents is needed in order for the matrices to interact with the outside world. Our definition for this operation shown in listing

Figure 16. Sparse matrix polymorphic operations

```
1  delay arr = SDelayed (dim arr) (index arr)


2  map f arr = case delay arr of
3      SDelayed (h, w) g -> SDelayed (h, w) (f . g)


4  zipWith f arr1 arr2 = SDelayed (h1, w1) get
5    where
6      SDelayed (h1, w1) f1 = delay arr1
7      SDelayed _ f2        = delay arr2
8      get val              = f (f1 val) (f2 val)

9  transpose mat = let
10                     (h, w)          = dim mat
11                     index' m (r, c) = index m (c, r)
12                 in SDelayed (h, w) (index' mat)

13 (#+)    = zipWith (+)
14 (#-)    = zipWith (-)
15 scale n = map (* n)
```

(16) as `delay`[4], assumes indexing for the representation argument has been defined and partially applies it to the matrix to extract the index transformation.

Conversion in the other direction, which we call `force`, is the topic of Section 3.6.

3.4.2 Collective operations. Listing (16) illustrates how collective operations such as `map`, `zipWith`, and `transpose` can also be defined polymorphically to work with the provided delayed representation by mainly reusing the same logic used for delayed vectors. Consequently, similar

[4]Type signature have been omitted in preference for readability

Figure 17. Generic Delayed Matrix-Delayed Vector Product

```
1 (#.) mat v@(_, len) = case mat of
2   (SDelayed (w, h) m_index_f) -> ((B.!) dot_ps, len)
3     where
4        row_funcs  = B.map (\ri ->
5                                ((curry m_index_f) ri
6                                , w))
7                                 $ B.enumFromN 0 h
8        dot_ps     = B.map (\r -> r !.! v) row_funcs
```

straightforward definitions of operations such as scalar multiplication, addition and subtraction can also be defined in equivalent terms [5].

3.4.3 Matrix-Vector Product. Recall that one of the goals of this design is to be able to express subprogram computations such $A\vec{x} + b$ in high-level terms, without computing any manifest intermediaries. So far, we have set up a generic representation of matrices in delayed form and defined the sum operation for both matrices and vectors. Therefore, the only remaining operation in the subprogram is the product between a delayed matrix and a delayed vector. This is the subject of Listing (17).

The matrix-vector product between matrix A and vector $\vec{x}$ can be defined as the vector whose entries are dot products between row vectors of A and the vector $\vec{x}$. Indexing functions for these rows can be extracted by enumerating all the rows, and partially applying the *curried*[6] version of the index transformation of A. This computation (extracting row index transformations) is bound to the `row_funcs`

[5] A quick comparison of lines 7,8,9 of Listing 14 and lines 13, 14, 15 of Listing 16 reveals that they are in fact the same. Merging these two (*rank polymorphism*) is possible in Haskell. For example, see [11]. Incorporating that into this work has been left as a subject for future work

[6] Currying refers to the transformation of a function taking a tuple of values as an argument, into a function that takes each of the components of the tuple, one at a time. Functions in Haskell are curried by default.

Figure 18. An implementation of $A\vec{x} + b$

```
1 axb a x b = a #. x !+! b
```

variable in Listing 17. The dot product between each of these delayed functions and $\vec{x}$ can thus be simulated by the (!.!) operation defined in Section 3.2—this is the computation bound to `dot_ps`. Therefore, the function corresponding to the resulting vector is the indexing function of `dot_ps`.

Note that although the final operation returns an indexing function to elements of a *boxed* vector, any attempt to actually compute this function will store the elements in an *unboxed* vector, de-referencing on pointer per element, as per the specification also of Section 3.2.

The computation $A\vec{x} + b$ can now be expressed as shown in Listing 18. For more subprogram implementations, see Section 3.7.

3.5 Manifest Instances

In this section we describe our encoding of the **COO**, **CSR**, and **CSR** sparse data compression formats as type indices in Haskell. These are the encodings we use to store our data, and forcing a delayed representation of data involves evaluating the data's indexing function into one of these formats. We provide a description of their respective `Sparse` instances that stays faithful to the discussion of Section II. Overall, this section should provide a guide to how a new format can be added to the system.

3.5.0.1 COO. As described in Section II, the **COO** compression format stores a sparse matrix using three arrays. An array for all the non-zeros of the matrix, an array for column indices of those elements, and one for the row indices. Haskell's vector library will store a vector of tuples of "unboxable" values

into consecutive memory slots the size of the vector. Consequently, our **COO** data representation consists of a vector of tuples `(a, Int, Int)` , along with the matrix's dimensions as captured by Listing 19.

To index into a matrix represented with the **COO** format thus boils down to finding the triple whose second and third element match the provided row and column, and returning its first element. We use vector's `find` function to traverse the array of triples and return the first one that matches the predicate. In the worse case, this operation will take $O(h + w)$ time.

3.5.0.2 CSR. As opposed to the **COO** format, arrays used to store a sparse matrix using **CSR** (Listing 19) are not necessarily of the same length; thus prohibiting the concise use of triples for representation. The compressed row offsets are stored in an array of their own, and so are columns and non-zero values.

Indexing into the matrix involves first slicing both the columns and non-zero arrays by a size equal to the difference between the offset stored at the given row and the element after it in the compressed array, then returning the element stored at the column index in the slice. In the worst case, this operation will take $O(|s|)$ time, where $|s|$ is the size of the slice[7], making this operation significantly faster than its **COO** correspondent for large matrices.

3.5.0.3 ELL. As described in Section II, two arrays of the same length, one for non-zeros and one for column indices, are used to store a matrix using the ellpack format (**ELL**) (Listing 20). The size of each of these arrays equals the maximum-elements-per-row multiplied by the number of rows. Rows with non-zero values less than the maximum are padded with zeros. Thus, to capture these

[7]This is under the assumption that zip and indexing take constant time, and that fusion works. The vector library makes all of this possible.

Figure 19. COO and CSR Format unboxed instances

```
-- COO instance
data COO
instance (Unbox e, Num e, Eq e) => Sparse COO U e where
    data SparseData COO U e
                   =   COO {
                         coo_vals :: Vector (e, Int, Int)
                       , width    :: Int
                       , height   :: Int }
    index (COO nnz w h) (r, c) = e
     where
       e = case find (\(a, x, y) ->
                           and [x == r, y == c]) nnz of
                 Nothing -> 0
                 Just (a1, _, _) -> a1
    dim (COO _ w h) = (h, w)

-- CSR instance
data CSR
instance (Unbox e, Num e, Eq e) => Sparse CSR U e where
    data instance SparseData CSR U e
        = CSR { row_offsets   :: Vector Int
                  , columns   :: Vector Int
                  , nnz       :: Vector e
                  , height    :: Int
                  , width     :: Int}
    index (CSR row_offs cols vals h w) (r, c) = el
        where
            to_slice = row_offs ! r
            to_start = case row_offs !? (r - 1) of
                                  Nothing -> 0
                                  Just n  -> n
            vec = slice to_start (to_slice - to_start)
                                   $ zip cols vals
            el  = case find (\(x, _) -> x == c) vec of
                          Nothing       -> 0
                          Just (_, a1)  -> a1
    dim (CSR _ _ _ h w) = (h, w)
```

Figure 20. ELL format unboxed instance

```
-- ELL instance
data ELL
instance (Unbox e, Num e, Eq e) => Sparse ELL U e where
    data instance SparseData ELL U e
       = ELL { max_e_row    :: Int
             , columns      :: Vector Int
             , nnz          :: Vector e
             , height       :: Int
             , width        :: Int}
    index (ELL max_e_r col_ind vals h w) (r, c) = el
       where
         to_start = r  * max_e_r
         vec      = slice to_start max_e_r
                          $ zip col_ind vals
         el = case  find (\(x,_) -> x == c) vec of
                          Nothing      -> 0
                          Just (_, a1) -> a1
    dim (ELL _ _ _ h w) = (h, w)
```

Figure 21. The *Force* type class

```
1 class (Sparse r D e, Sparse r U e) => Force r e where
2     force :: SparseData r D e -> SparseData r U e
```

properties, the maximum-elements-per-row value, a vector for the non-zeros, and one for the columns are encapsulated by a product type.

Indexing into a matrix stored with this format is thus similar to the **CSR** format in the sense that they both involve slicing a zipped structure comprised of column index, non-zero value pairs. The only difference being that in **ELL**, the size of the slice equals the number of maximum elements per row. Thus, to retrieve the correct element, first, the slice's starting position is found, the slice is applied, and finally the non-zero element returned from the slice is the second element of the pair whose first element equals the column argument.

Consequently, in the worst case, indexing takes $O(max)$ time, where max is the maximum number of elements per row in the matrix.

3.6 Forcing

To define an overloaded operator that encodes the action of *forcing* any delayed representation of a matrix stored with any sparsity format, we define the *Force* type class (Listing 21). This makes it easier to extend this work with parallelism because all that is needed is to embed the data parallelism implementation into the *force* function associated with this class. As a consequence, restrictions are put in place to use only collective functions provided by the vector library, so that parallelizing those operations should imply the parallelization of *force*.

Figure 22. COO *Force* instance

```
instance (Sparse COO D e
          , Sparse COO U e) => Force COO e where
    force (SDelayed (h, w) func) = COO vals w h
      where
        vals_r r = unfoldrN w (\c ->
                                  if func (r, c) /= 0
                                  then Just ((func (r,c)
                                             , c), c + 1)
                                  else Nothing) 0
        rows     = Prelude.map (\r -> U.map (\(x, c)
                                     -> (x, r, c))
                                     (vals_r r)) [0..h-1]
        vals     = concat rows
```

3.6.1 COO. To force an index transformation into a **COO** encoded matrix (Listing 22), first we calculate a function `val_r` to generate all the elements of a row by looping over the range of all columns applying the indexing transformation. Next, we compute a list comprised of a vector per row—with values of the row as well as their associated column and row indices—by looping over all the rows applying `val_r`. Last, these vectors are concatenated into one large vector.

3.6.2 CSR. To force an index transformation into a **CSR** encoded matrix (Listing 23), a similar first, second and third step is taken, by defining a function to generate all the elements of a given row, looping over all the rows applying the function to generate the rows as vectors, and concatenating these into one large vector containing all non-zero values. However, an extra step involving counting the number of elements per row is taken, and a left scan performed to obtain the compressed array of row offsets.

Figure 23. CSR *Force* instance

```
instance (Sparse CSR D e
         , Sparse CSR U e) => Force CSR e where
    force (SDelayed (h, w) func)
          = CSR r_offs cols vals h w
        where
            vals_r r = unfoldrN w (\c ->
                                     if func (r, c) /= 0
                                     then
                                       Just ((func (r,c)
                                              , c)
                                             , c + 1)
                                     else Nothing) 0
            rows         = Prelude.map (\r -> vals_r r)
                                       [0..h-1]
            all_vals_c   = concat rows
            r_counts     = Prelude.map length rows
            r_offs       = scanl (+) 0 r_counts
            (vals, cols) = unzip all_vals_c
```

Figure 24. ELL *Force* instance

```
instance (Sparse ELL D e
         , Sparse ELL U e) => Force ELL e where
force (SDelayed (h, w) func) = ELL r_max cols vals h w
        where
          vals_r r   = unfoldrN w (\c ->
                                     if func (r, c) /= 0
                                     then Just ((func (r,c)
                                          , c), c + 1)
                                     else Just ((0, c)
                                         , c + 1)) 0
          rows       = Prelude.map (\r -> vals_r r)
                                   [0..h-1]
          all_vals_c = concat rows
          r_max = let len_list = Prelude.map length rows
                  in if not $ Prelude.null len_list
                     then Prelude.maximum len_list
                     else 0
          (vals, cols) = unzip all_vals_c
```

3.6.3 ELL. Similar to the other two formats, computing into a **ELL** encoded matrix, involves evaluating the rows and concatenating them into one large vector of values. However, to support indexing, an extra step is done to find the row with the maximum number of elements and store that number along with the non-zeros, column indices, the height and the width.

3.6.4 Cross-Format Conversion. Note that, as Listing 25 illustrates, conversion between any two matrices, whether they be delayed or not is fairly straightforward. Conversion between delayed formats requires no extra work but to specify the correct type for the target format, while conversion between manifest versions first delays the structure to be converted, then converts it to the appropriate target index, to which the *force* function of Section 3.5 gets applied.

Figure 25. Cross-Format Conversions

```
convert :: (Sparse r1 D e, Sparse r2 D e)
        => SparseData r1 D e
        -> SparseData r2 D e
convert (SDelayed (w, h) func) = (SDelayed (w, h) func)


manifest_convert :: (Force r1 e, Force r2 e)
                 => SparseData r1 U e
                 -> SparseData r2 U e
manifest_convert  = force . convert . delay
```

subprogram	**Operation**
AXPY	$\vec{x} \leftarrow \alpha\vec{x} + p\vec{y}$
VADD	$\vec{x} \leftarrow \vec{w} + \vec{y} + \vec{z}$
WAXPY	$\vec{x} \leftarrow \alpha\vec{x} + \beta\vec{y}$
ATAX	$\vec{y} \leftarrow A^T A\vec{x}$
BICGK	$\vec{q} \leftarrow A\vec{p}$, $s \leftarrow A^T\vec{r}$
DGEMV	$\vec{z} \leftarrow \alpha A\vec{x} + \beta\vec{y}$

Table 1. subprogram specifications

3.7 subprogram Implementations

To illustrate that the interface does indeed achieve the proposed genericity goal of Section I, implementations of the subprograms of 1 are provided by Listing 26.

Figure 26. Example subprogram Implementations

```
1  axpydot w v u alpha = (z, r)
2     where
3       z        = w !-! (alpha !*! v)
4       r        = z !.! u
5  vadd v1 v2 v3 = v1 !+! v2 !+! v3

6  waxpby a x b y = a !*! x !+! (b !*! y)

7  twiceAxpyNoForce a x p y = let n = axpy a x p y
8                             in axpy a n p n

9  twiceAxpyForce a x p y = let n = to_vector
10                                  $ axpy a x p y
11                            in axpy a (from_vector n) p
12                                      (from_vector n)

13 atax a x = (transpose a) #. (a #. x)

14 bicgk a p r = (a #. p, transpose a #. r)

15 smvm_xpy mat vec1 vec2 alpha = ((alpha !*! mat)
16                                 #. vec1) !+! vec2

17 gemv alpha beta a x y = (alpha `scale` a #. x)
18                         !+! (beta !*! y)

19 gemvt alpha beta a y z = let x = force
20                                  $ (beta !*!
21                                  ((transpose a)
22                                    #. y)) !+! z
23                           in (delay x, alpha
24                                     !*! (a #. (delay x)))
25 gesummv alpha beta a b x = (alpha !*! (a #. x))
26                                  !+! (beta !*!
27                                      (b #. x))
```

CHAPTER IV

RESULTS AND ANALYSIS

First, let us note that, because Haskell is a lazy language, the usual manner of timing execution involving the time difference between recorded starting and ending points, such as shown in Listing 27, does not quite work. For example, the compiler is free to choose not to execute function `f`, if its result never gets used. Consequently, it is possible to get bogus benchmark results if one is not cautious about the laziness aspect of the language. To allow users to control the laziness/strictness of their programs, Haskell does provide functions to control how deeply expressions get computed. To compute an expression into their *weak head normal form*[1], the language provides the `seq` function, whereas computation into *normal form* is provided through `deepseq`. These functions are rarely used, but can be critical for performance.

For our performance analysis, we use the Criterion library[1] which requires the result of a benchmark to either be in *weak head normal form* or *normal form* before it can proceed. The library uses an *Ordinary Least-Squares (OLS) regression* model to estimate execution time for a single loop iteration of a given benchmark. It reports the mean execution time, as well as the standard deviation statistics and an R^2 *goodness-of-fit* value indicating how accurately the regression model fits

[1]Expressions in weak head normal form have been computed up until the outer most constructor whereas expressions in normal form have been fully computed. Therefore, all expressions in normal form are also in weak head normal form, but not vice versa.

subprogram	**Operation**
DAXPY	$x \leftarrow \alpha\vec{x} + p\vec{y}$
ATAX	$y \leftarrow A^T(Ax)$
DGEMV	$z \leftarrow \alpha Ax + \beta y$

Table 2. Example Operations

matrix	width	height	non-zeros	percentage (%)
rdb800l	800	800	4640	0.725
rdb450	450	450	2580	1.27
rdb200	200	200	1120	2.8
bcsstk03	112	112	640	5.1
bcsstk09	1083	1083	18437	1.57
bcsstk11	1473	1473	34241	1.57
bcsstk14	1806	1806	63454	1.94
tub100	100	100	396	3.96
bcsstm03	112	112	72	0.57
pores_1	30	30	180	20
LF10	18	18	82	25

Table 3. Example matrix dimensions and sparsity.

Figure 27. The Problem of Timing Execution in Haskell

```
start = time.time()
y = f()
end = time.time()
print (end - start)
```

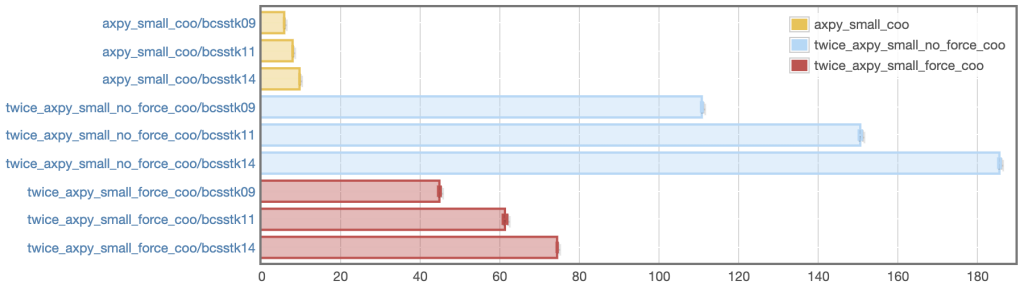

Figure 28. Force vs no-force DAXPY runtimes (μs)

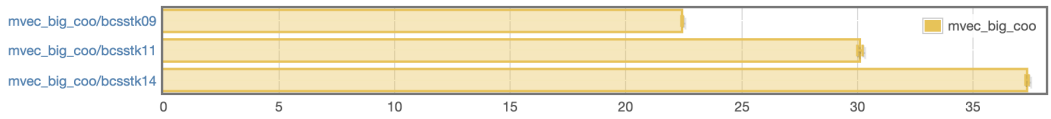

Figure 29. Matrix-vector multiplication runtimes COO (μs)

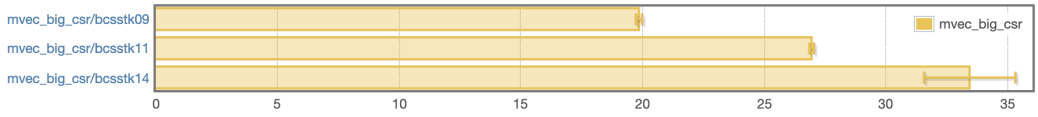

Figure 30. Matrix-vector multiplication runtimes CSR (μs)

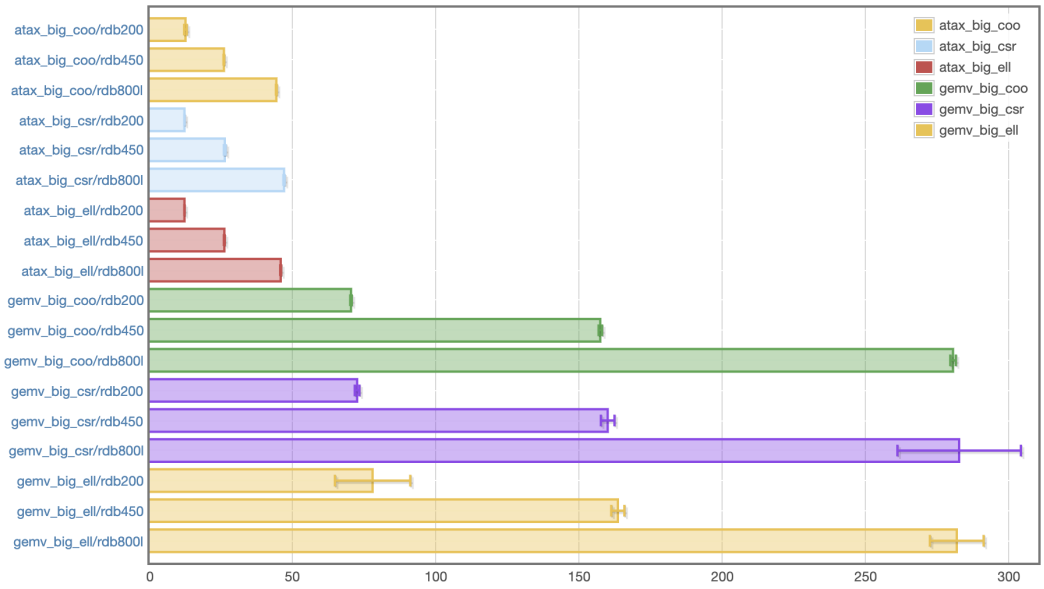

Figure 31. Small matrices atax/gemv runtime (μs)

the observed measurements. According to the authors, for non-bogus results, the R^2 number should lie between 0.99 and 1. Finally, Criterion reports and plots a *subprogram density estimate* of time measurements to indicate the probability of any given measurement occurring.

We report the results of running experiments involving the three linear algebra algorithms of Table 2 on an Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz processor with L1 data and instruction caches of 32K each, an L2 cache of 1024K and L3 cache of 28160K. *ATAX* can be used in numerical algorithms as part of solutions to least-squares equations [20], and involves two matrix-vector operations applied in sequence. *DAXPY* is part of the BLAS level-1 specification, while *DGEMV* is part of level 2. DAXPY is composed of scalar multiplication followed by vector addition, while DGEMV captures matrix-vector multiplication followed by vector addition.

To benchmark these operations, we use real-world matrices from the popular SuiteSparse matrix collection[6]; we have preserved their names in this report, and their statistics are displayed in Table 3. Figure 28 shows box plots of running DAXPY once, then twice when the forcing function has been called, and when it has not[2]. As the figure shows, *sharing* (see Listing 12) gets lost if computation does not get forced.

Figures 30 and 29 present the execution times for sparse matrix-vector multiplication. Even though the matrices we use for our benchmarks are relatively small (Table 3), **CSR** still introduces an observable small performance boost ($\approx 3\mu s$) as its indexing function generally runs faster.

Figure 31 shows the results of running both the ATAX and GEMV subprograms on several matrices of table 3. Because these matrices are not large

[2]The implementation of these functions is shown in Listing 26.

enough, the effects of changing their storage format is not directly visible. For future work, we plan on running experiments on matrices of larger dimensions and larger non-zero counts. However, the figure does show that the execution times of both the GEMV and the ATAX routines increase almost linearly with the number of rows/column but not the density of the matrix. Among other things, this can be attributed to the fact that the implementation of matrix-vector product we provided will generate a function for each row and try to get its delayed dot product with the vector, regardless of whether the dot product evaluates to zero (i.e., the row is made entirely of zeros). This means that, under the current implementation, any matrix with a larger number of rows will always run slower than one with a smaller number. This is only a hypothesis that has not been confirmed. If this is the issue, one starting point would be between sparse and dense vectors where zero values of a sparse vector could be omitted by storing the vector as a pair of indices and non-zero elements.

The use of Criterion for benchmarking is quick and simple but it has its drawbacks. For example, one of the reasons for evaluating smaller benchmark problems is because Criterion takes an unreasonably long time to perform its experiments before returning, and it is not clear how to control this excessive overhead.

Overall, our matrices are not large enough for any big noticeable effects. However, from the examples provided, we have shown that the original goal of fusing intro-subprogram instructions as well as different subprograms themselves can be achieved with the interface presented. For future work, we plan to perform experiments on larger matrices before we tackle the introduction of data parallelism into the system.

CHAPTER V

CONCLUSION AND FUTURE WORK

Keeping in accordance with the Basic Linear Algebra Specifications, we have shown that we can write element-type and memory-layout-agnostic implementations of linear algebra subprograms using Haskell. We have also shown that through the technique of separating delayed and manifest representations of matrices and vectors, we can enable users to control when a given subprogram or chains of subprograms get computed. We have used this approach to provide implementations of subprograms that use the *coordinate* format, the *compressed sparse row* format, as well as the *ellpack* format. Finally, we have presented performance figures highlighting the potential loss of sharing that could arise if computation is not forced, as well as the difference in run-times between different formats' indexing functions.

Although our implementation is entirely serial, the interface was designed to be easily extensible with data parallelism by modifying the function used to *force* a delayed representation of a matrix into its manifest equivalent. As such, several steps can be taken to extend the presented design for the future. These include running more tests to observe the behavior of the current implementation across cache lines, using Haskell's type system to check for dimension mismatches at compile rather than run-time, looking into providing data-parallel implementations of functions from Haskell's vector library we use as part our force function, and last, comparing the performance our implementation with already existing implementations.

If a high-performance embedding of linear algebra computations in Haskell was successful, its strong type system would not only ensure their purity but would also open up more compiler-specific optimizations. Moreover, as Haskell has

an LLVM backend, this would be a step towards a cross-paradigm, cross-lingual standard interface for the BLAS set of functions and other numerical computations. We have presented one step towards that goal.

REFERENCES CITED

[1] A criterion tutorial. `http://www.serpentine.com/criterion/tutorial.html`. (Accessed on 08/28/2020).

[2] Ghc users' guide section 7.2. unboxed types and primitive operations. `https://downloads.haskell.org/~ghc/7.0.3/docs/html/users_guide/primitives.html`. (Accessed on 08/25/2020).

[3] Ghc users' guide section 7.8. kind polymorphism and promotion. `https://downloads.haskell.org/~ghc/7.4.1/docs/html/users_guide/kind-polymorphism-and-promotion.html`. (Accessed on 08/28/2020).

[4] Haskell online report: https://www.haskell.org/onlinereport/haskell2010/. `https://www.haskell.org/onlinereport/haskell2010/`. (Accessed on 08/25/2020).

[5] Sparse matrices (scipy.sparse) — scipy v1.5.2 reference guide. `https://docs.scipy.org/doc/scipy/reference/sparse.html`. (Accessed on 08/02/2020).

[6] Suitesparse matrix collection. `https://sparse.tamu.edu/`. (Accessed on 08/28/2020).

[7] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[8] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 1–13, New York, NY, USA, 2005. Association for Computing Machinery.

[9] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

[10] Jack J Dongarra, Cleve Barry Moler, James R Bunch, and Gilbert W Stewart. *LINPACK users' guide*. SIAM, 1979.

[11] Jeremy Gibbons. Aplicative programming with naperian functors. In *European Symposium on Programming*, pages 556–583. Springer, 2017.

[12] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[13] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[14] P. Hudak. Modular domain specific languages and tools. In *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, pages 134–142, 1998.

[15] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *SIGPLAN Not.*, 45(9):261–272, September 2010.

[16] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. Fun with type functions. In *Reflections on the Work of CAR Hoare*, pages 301–331. Springer, 2010.

[17] Ben Lippmeier and Gabriele Keller. Efficient parallel stencil convolution in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, page 59–70, New York, NY, USA, 2011. Association for Computing Machinery.

[18] Thomas Nelson, Geoffrey Belter, Jeremy G Siek, Elizabeth Jessup, and Boyana Norris. Reliable generation of high-performance matrix algebra. *ACM Transactions on Mathematical Software (TOMS)*, 41(3):1–27, 2015.

[19] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *2001 Haskell Workshop*. ACM SIGPLAN, September 2001.

[20] Jeremy G Siek, Ian Karlin, and Elizabeth R Jessup. Build to order linear algebra kernels. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[21] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 124–132, New York, NY, USA, 2002. ACM.