TOWARDS OPTIMIZED VECTOR INSTRUCTIONS FOR HIGH-PERFORMANCE FUNCTIONAL PROGRAMMING

by

MAMTAJ AKTER

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

June 2020

THESIS APPROVAL PAGE

Student: Mamtaj Akter

Title: Towards Optimized Vector Instructions for High-Performance Functional Programming

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 June 2020

THESIS ABSTRACT

Mamtaj Akter

Master of Science

Department of Computer and Information Science

June 2020

Title: Towards Optimized Vector Instructions for High-Performance Functional Programming

The Basic Linear Algebra Subprograms or BLAS provide the foundation for much of the software used in scientific computing. To date, BLAS has been implemented in C, Fortran, and directly in assembly. These languages allow the implementations to be well optimized by hand ensuring when a BLAS routine is called that it is as fast a possible.

Functional programming languages, and in particular Haskell, do not allow the fine-grained control over memory, and their high-level features make it hard to optimize a single function to the level of C or assembly. However, Haskell has an advantage when optimizing combinations of container-based operations. Because of this we explore both implementing BLAS in Haskell and comparing the Glasgow Haskell Compiler's ability to optimize scientific programs to that of a C compiler.

CURRICULUM VITAE

NAME OF AUTHOR: Mamtaj Akter

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, Oregon
Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
Shahjalal University of Science and Technology, Sylhet, Bangladesh

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2020, University of Oregon, Eugene, Oregon
Master of Science, Computer Science and Engineering, 2015, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh
Bachelor of Science, Computer Science and Engineering, 2008, Shahjalal University of Science and Technology, Sylhet, Bangladesh

AREAS OF SPECIAL INTEREST:

Human-Computer Interaction
User Interfaces
Text-Entry Techniques
Adolescent Online Safety

PROFESSIONAL EXPERIENCE:

Graduate Teaching Assistant, University of Oregon, Sep 2017 – Jun 2018, Jan 2019 – Jun 2019, Sep 2019 – Dec 2019, Mar 2020 – Jun 2020

Graduate Research Assistant, University of Oregon, Sep 2018 – Dec, 2018, Dec 2019 – Mar 2020

Primary Instructor, University of Oregon, Jun 2019 – Aug 2019

Adjunct Lecturer, Ranada Prashad Saha University, Narayanganj, Bangladesh, Dec 2015 – Apr 2017

Programmer, IICT, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, Dec 2015 – Apr 2017

PUBLICATIONS:

Akter, M. & Islam, A. & Rahman, A. Fault Tolerant Optimized Broadcast for Wireless Ad hoc Networks. *In Proceedings of 2nd International Conference on Networking, Systems and Security 2016, January 7-9, 2016, IEEE.*

ACKNOWLEDGEMENTS

Most of all, I would like to thank my advisor, Dr. Boyana Norris, for her infinite guidance and unbounded support. Boyana taught me how to conduct HPC research and write readable technical papers. I would also like to thank my family for their never ending inspiration and support. I am also grateful to my labmate, Bosco Ndemeye for his constant help in learning functional programming better.

To my loving husband, Zakir.

TABLE OF CONTENTS

LIST OF FIGURES

Figure Page

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

Linear Algebra is a study of mathematics that involves mathematical objects like scalars, vectors, and matrices. The application of linear algebra is widely prevalent to solve the technical problems in the area of physics, mathematics, engineering, and scientific software. Many general-purpose or specialized linear algebra libraries are used by computer scientists and the major motivation behind these libraries is to hide the low-level calculations of the linear equations from the large scientific software implementations. Basic Linear Algebra Subprograms (BLAS) [6] is one of the most popular cross-platform libraries that provides a set of low-level routines to perform many common linear algebra operations like vector scaling, vector sum, the dot product, vector-matrix multiplication, triangular solve and so on.

The BLAS libraries are implemented in high-level imperative languages such C or Fortran (a low-level language like Assembly is also used in optimized versions) and therefore, these can be applied in any imperative language applications. In general, imperative programming is a style of programming that allows programmers to write a sequence of statements that helps the system to reach a certain goal. As each of these statements gets executed, the state of the program gets changed. To acquire the correct output, the program needs to follow the exact procedure and so, any modification of the given inputs often leads to confusing problems and errors in the code. Thus, the imperative language also has fallen short in the aspect of code clarity and understandability.

Functional Programming addresses the above-mentioned issues since it is a different paradigm of programming that binds any problem to a pure mathematical functional style. It mainly focuses on expressions that produce values depending on what goal or output a program needs to achieve. Functional language like Haskell brings three major benefits over imperative ones. First, with the help of some specific higher-order functions [14] such as maps, zips, folds, functional programs provide a more intuitive structure ensuring both modularity and clarity. Such shorter and clearer code leads to improved development productivity, higher quality and fewer bugs. Second, although Haskell lacks loops, programmers use tail-recursive functions [2], and the GHC (Glasgow Haskell Compiler) does not allocate memory for every recursive application; ratherm, it performs a tail call optimization [19] to avoid running in linear space, similar to an imperative language where a loop gets executed in constant space. Third, GHC offers stream fusion [5]—a straightforward idea of transforming the recursive structures into non-recursive co-structures to eliminate intermediate structures by combining adjacent transformations and applying equational laws [17].

Nevertheless, with a functional language like Haskell, programmers struggle in terms of program optimization due to features such as lazy evaluation, purely functional data structures, and lack of memory control. On the other hand, C and Fortran languages allow programmers to interact with memory in a low-level way. This leads us to two research questions: (1) Can a functional language implementation of the BLAS achieve performance comparable with imperative languages? (2) If not, what is limiting performance? To find the answers, we implement a linear algebra library titled HBLAS in Haskell and perform a comparative study with an imperative BLAS application.

In this thesis, our contributions include: 1) A functional implementation of the BLAS level-1 and level-2 interfaces; 2) Implementation of two linear iterative solvers, in both C and Haskell; 3) Optimization of the HBLAS library and compare the performance of the applications with the imperative applications.

The rest of the paper is organized as follows. Chapter 2 discusses the related work on optimizing linear algebraic operations. In Chapter 3, background and terminologies that we need to read the rest of the paper, are thoroughly described. The methodology is presented in Chapter 4. Finally, Chapter 5 illustrates the experimental results and analysis and Chapter 6 concludes the thesis and outlines future work directions.

CHAPTER II

RELATED WORK

Implementing the Basic Linear Algebra Subprograms (BLAS) in a functional language is a relatively new idea; by contrast, imperative approaches have been around for several decades. As functional language compilers begin to implement an increasing number of performance-improving strategies, the performance of functional programs may approach that of traditional HPC-language implementations. To evaluate the current state and identify current shortfalls, in this thesis, we implement the BLAS using Haskell (HBLAS). Moreocer, we create several implemenmtations using different data structures and we also apply some GHC language features to make the library as optimized as possible. In this chapter, we discuss the related works that we found so far on functional linear algebra libraries.

Mainland et al. in [8] proposed a novel efficient generalized stream fusion framework. To express the operations on vectors, the abstraction mechanism they used was a bundle of streams. The streams are chosen so that for any given high-level vector operation there is a stream in the bundle whose representation leads to an efficient implementation. Because the GHC eliminates intermediate bundle structures, this bundle abstraction has no run-time cost. They experimented on a BLAS level-1 function dot-product and showed that using their proposed vector library with generalized stream fusion, Dot-product achieved a better performance than the dot product of CBLAS.

Accelerate [4] is a library that GHC has launched to efficiently deal with the array-operations that needs superior performance. In this Data.Array.Accelerate module Array or matrix computations are expressed as parameterized collective operations, such as maps and reductions.

GHC also includes a simple way to access the BLAS library from Haskell and to call the BLAS routines from Haskell, the Haskell programmers need a BLAS and LAPACK binding that provides the full BLAS and LAPACK APIs. This interface is named HBlas [18] by GHC.

None of the above work focused on a complete solution for linear algebra computations with a functional language. Therefore we attempt to implement a linear algebra library useful for programmers who can exploit the benefits of functional language but also can achieve good performance with large data sets.

CHAPTER III

BACKGROUND

In this thesis we implement a library for basic linear algebra operations in Haskell and utilize different data structures to find an optimized solution. This chapter introduces the concepts used in our research.

Since our library is broadly inspired by BLAS [6], we need to discuss its functionality. BLAS is categorized into three levels- evel 1: scalar-vector and vector-vector operations, level 2: vector-matrix operations, and level 3: matrix-matrix operations.

BLAS level-1 consists only of vector-scalar and vector-vector operations. Table-1 lists the scalar and vector operations that are included in level-1. This table is organized with the scalar and vector reduction operations (Dot Product, Vector norms, Sum, Min value and location, Max value and location, Max abs value and location), the vector rotation operations (Generate plane rotation- _ROT , Generate Givens Plane rotation- _ROTG, Modified Givens Transformation- _ROTM , _ROTMG) and the vector operations (Scaled vector addition, scaled vector accumulation).

BLAS level-2 and level-3 include some important matrix-vector operations and matrix-matrix operations respectively. Table-2 and Table-3 depict the subroutines in detail. Both of these levels consist of the four most important vector-matrix operations: general matrix-vector or matrix-matrix multiplication, symmetric matrix-vector or matrix-matrix multiplication, triangular matrix-vector or matrix-matrix multiplication and triangular solve vector or triangular solve matrix.

Table 1. BLAS level-1 functions

Name	Arguments	Description	Equation
axpy	α,x,y	update vector	y = y+ αx
scal	α,x	scale vector	y = αy
copy	x,y	copy vector	y=x
swap	x,y	swap vectors	$y \leftrightarrow x$
dot	x,y	dot product	$= \sum x * y$
nrm2	x	Euclidean norm	$= \|x\|_2$
asum	x	1-norm	$= \sum \mid x_i \mid$
iamax	x	index of $max(\mid x_i \mid)$	
rotg	a,b,c,s	generate given rotation	
rot	x,y,c,s	apply plane rotation	
rotmg	d1,d2,a,b,param	generate modified plane rotation	
rotm	x, y, param	apply modified plane rotation	

In this thesis we test the performance of our HBLAS library in some real-world linear algorithms. There are several iterative methods to solve the linear equation though: x= Aˆ1b. And so we utilize our proposed HBLAS in two such iterative linear methods like Conjugate Gradient (CG) [3] and Transpose-Free Quasi Minimal Residual (TFQMR) [7] Methods. CG Method is one of the most prominent iterative algorithms for solving the system of linear equations A*x=b for x. It takes the matrix A of size N X N and the vector b as input where A must be a symmetric matrix and the vector b must have length N. It solves the linear equation in n iterations and each iteration requires a few vector operations like axpy, scale, dot product, euclidean norm, and matrix-vector multiplication. The solution x = $A^{-1}b$ is obtained in the conjugate gradient through Gaussian Elimination, so there is no need for matrix inversion which is a computationally expensive matrix operation. Therefore, this algorithm requires less memory space and so, it is particularly suitable for large scale systems. This method provides a

Table 2. BLAS level-2 functions

Name	Arguments	Description	Equation
gemv	α,A,x,β, y	general matrix-vector multiplication	$y = \alpha Ax + \beta y$
symv	α,A,x,β, y	symmetric matrix-vector multiplication	$y = \alpha Ax + \beta y$
trmv	A,x	triangular matrix-vector multiplication	$y = Ax$
trsv	A,x,y	triangular solve vector	$x = A^{-1}y$
syr	α,A,x	symmetric rank-1 update	$B = A + \alpha xx$
syr2	α,A,x,y	symmetric rank-2 update	$B = A + \alpha xy + \alpha yx$

monotonically improving approximation x_k to the exact solution x, which may reach the required tolerance after a relatively small (compared to the problem size) number of iterations. And the number of iteration required is being controlled by the euclidean norm of the residual r = b-A*x. On the other hand, the TFQMR algorithm is used to obtain fast solutions for linear systems with very large and very sparse coefficient matrices. It solves systems of the form A x = b where the operator A is a square non-symmetric matrix of size N X N and b is a vector of size N given as inputs and x is the output.

Table 3. BLAS level-3 functions

Name	Arguments	Description	Equation
gemm	α,A,B,β, C	general matrix-matrix multiplication	$C = \alpha AB + \beta C$
symm	α,A,B,β, C	symmetric matrix-matrix multiplication	$C = \alpha AB + \beta C$
trmm	α,A,B	triangular matrix-matrix multiplication	$B = \alpha AB$
trsm	α,A,B	triangular solve matrix	$B = \alpha A^{-1}B$
syrk	α,A,β,C	symmetric rank-1 update	$C = \alpha AA + \beta C$
syr2k	α,A,B,β,C	symmetric rank-2 update	$C = \alpha AB + \alpha BA + \beta C$

We implement BLAS in Haskell using the higher-order functions [12] that GHC provides. These functions are considered as the powerful abstraction mechanism that makes Haskell programs more declarative and often much shorter than their imperative counterparts. Let us consider the mathematical definition of the matrix-vector multiplication to discuss the merits of functional programming. In the equation below, A is a matrix of size m rows and n columns and x is a vector of size n. The matrix-vector product A*x is defined to be the vector y=A*x of size m. We also consider the Haskell and C code for the vector-matrix multiplication below where the contrast in programming style is apparent- the definition of the multiplication in Haskell has an intuitive structure whereas the C code has a complex structure that lacks code clarity and modularity.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{pmatrix}; A = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ & & . & \\ & & . & \\ & & . & \\ a_{m1} & a_{m2} & ... & a_{mn} \end{pmatrix}$$

$$y = A * x = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n \\ . \\ . \\ . \\ a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n \end{bmatrix}$$

```
mult  a  x  =  [[ sum $ zipWith (*) ai x ] | ai <- a ]
```

```
for (i=0;i<rows;i++)
        for (j=0;j<cols;j++)
            y[i]+=a[i][j]*x[j];
```

Haskell offers three different data structures - List, Boxed Vector and Unboxed Vector.For each of these data structures GHC provides three corresponding libraries: Data.List [15], Data.Vector and Data.Vector.Unboxed. Data.List module consists some very useful higher-order functions to do the list operations easily. Take, Drop, Map, Filter, (++), head, last, tail, foldl, foldr, reverse, repeat, zipWith are some examples of the functions. Data.Vector [26] module is used for boxed vectors and it supports a rich interface of list-like functions. Boxed vector data-structure provides the advantage of safe indexing and thus the complexity of the update operations are O(1). It also provides all the functionalities that Data.List library belongs. Data.Vector.Unboxed [13] library is a re-implementation of the vector library and hence, it includes the same set of functions. This module is used to take advantage of stream fusion, and GHC optimizations. In the unboxed-vector data representation, the elements are stored directly in the allocated array and therefore they bring a big advantage over boxed vectors - the elements are all stored contiguously.

Nevertheless, Haskell also has an important characteristic that makes it not so suitable in time-critical scientific computing. It follows a lazy evaluation strategy.

In strict evaluation, the function arguments are completely evaluated before passing them to a function, but in lazy evaluation, the evaluation of function arguments is delayed as long as possible. In the example code below, even though the expression [1..] produces an infinite list and the singleton list [5] is supposed to be appended at the end of the infinite list, the Glasgow Haskell Compiler (GHC) [24] chooses not to expand [1..] at once, rather it takes one element at a time. This very special attribute of functional programming causes inefficiency since it accumulates all the operands before starting evaluating each value and it also makes the code vulnerable to a stack overflow issue since evaluating an expression like (((5+2)+2)+7) requires pushing 2 and 7 in the stack before evaluating 5+2 and then popping 2 from the stack, adding along the way. However, the laziness feature of GHC makes variables to be evaluated only when they are needed and that creates a series of binds stacked up which leads to space leak [11], [20], [16]. This space leaks issue is a major reason why functional programs take larger execution time and its mostly due to the data immutability feature which forces the compiler to produce a lot of temporary data. To mitigate these issues, GHC packages provide some strict version of higher-order functions like foldr' and foldl' [22] that tells the system that the inner redex must be evaluated and reduced first before the outer ones.

```
print (take 4 ([1..] ++ [5]))

take 4 ([1..] ++ [5])
take 4 (1 : ([2..] ++ [5]))
```

```
1 : take 3 ([2..] ++ [5])
1 : take 3 (2 : ([3..] ++ [5]))
1 : 2 : take 2 ([3..] ++ [5])
1 : 2 : take 2 (3 : ([4..] ++ [5]))
1 : 2 : 3 : take 1 ([4..] ++ [5])
1 : 2 : 3 : take 1 (4 : [5..] ++ [5])
1 : 2 : 3 : 4 : take 0 ([5..] ++ [5])
1 : 2 : 3 : 4 : 5 : []
```

However, GHC offers some solutions that help to improve time-critical functions like our HBLAS subprograms. To force GHC to be more strict in its evaluation, GHC supports an extension of pattern matching called bang pattern [23], written as !x when x is a parameter of a function. To activate the bang pattern feature, we add $\{-\#LANGUAGEBangPatterns\#-\}$ to the top of our program file. Let's take the following simple add method with a bang pattern where function b gets evaluated without cluttering all the intermediate outputs together until the end.

```
{-# INLINE add #-}
add:: Int->Int->Int
add !m !n = m+n

a= add 2 (1+4)
b= add a 5
```

```
b= add (add !2 (1+4)) 5
b= add (add 2 5) 5
b= add 7 5
b= 12
```

To write performant functional programs, GHC has an optimization technique like inliner [25]. GHC can decide whether to inline a particular function or not, it looks at its size and assigns some sort of weight to that function. It considers several internal factors before inlining a function, for example- How much code duplication inlining would cause or how much work duplication would inlining cause. Therefore, we use INLINE pragma to exploit GHC's optimization.

CHAPTER IV

METHODOLOGY

In the previous chapter, we briefly discussed the supporting features and tools that we needed to implement and optimize our HBLAS. This chapter provides an in-depth discussion of the methodologies used in the implementation of HBLAS and we also discuss the optimization techniques that we used to get the best runtime performance. The HBLAS implementation can be accessed from our GitHub repository at [1].

We organize this chapter in the following sections: 1) Implementation of BLAS levels 2) GHC libraries, 3) Implementation of two iterative linear solvers, and 4) GHC optimization techniques.

4.1 Implementation of BLAS Levels

This section summarizes the vector and list operations of HBLAS. We implement levels 1 and 2 of HBLAS since our example applications extensively used scalar-vector, vector-vector, and vector-matrix operations.

4.1.1 HBLAS Level-1.

HBLAS level-1 addresses the scalar-vector and vector-vector operations. It includes the following functions:

- scal: Vector-scalar product, x=α x
- axpy: Scaled vector accumulation, y = α x + y
- copy: Copy vector, y = x

- swap: Vector-vector swap, y <-> x
- dot: Dot product, $= \sum x * y$
- nrm2: Vector 2-norm (Euclidean norm), $= \|x\|_2$
- asum: Sum of vector magnitudes, $= \sum |x|$
- iamax: Index of the maximum absolute element of a vector
- iamin: Index of the minimum absolute element of a vector
- rot: Plane rotation of points
- rotg: Generate Givens rotation of points
- rotm: Modified Givens plane rotation of points

Now we discuss the Haskell implementation of the ;evel-1 subprograms with Vector data structure. Scal function takes a scalar value alpha and a vector x and using the higher-order function fmap, the function (* alpha) is applied to each element of the vector and returns a scaled vector. On the other hand, axpy function does both scaling and addition - it takes two vectors xs, ys, and a scalar value alpha and zipWith applies the function (y + alpha * x) pairwise on each member of both vectors. axpy returns element-wise addition of vector ys and scaled vector of xs (by alpha). dot function provides a scalar value that is the dot product of two input vectors using the zipWith method. The nrm2 function returns the euclidean norm of an input vector using the function composition of three functions: fmap (i*i), sum, and sqrt.

```
scal :: (Num n) => n -> Vector n -> Vector n
scal alpha x= fmap (alpha*) x

axpy :: (Num n) => n -> Vector n -> Vector n
        -> Vector n
axpy alpha xs ys =zipWith (\x y -> y+alpha*x) xs ys

dot :: (Num n) => Vector n -> Vector n -> n
dot a b = sum (zipWith (*) a b)

nrm2 :: (Num n, Floating n) => Vector n -> n
nrm2 = sqrt . sum . fmap (\i -> i * i)

asum :: ( Num n) => V.Vector n -> n
asum y= foldr' (\x a -> abs x + a) 0 y

iamax :: (Num n, Ord n)=> Vector n -> Maybe Int
iamax a = elemIndex
        (maximum (fmap abs a)) (fmap abs a)

iamin :: (Num n, Ord n)=> V.Vector n -> Maybe Int
iamin a = elemIndex
        (minimum (fmap abs a)) (fmap abs a)

swap :: (a, b) -> (b, a)
```

```
swap (x, y) = (y, x)


copy :: ( Eq n, Num n) => Vector n -> Vector n
         -> Vector n
copy xs _ = xs
```

4.1.2 HBLAS Level-2.

Level-2 addresses the matrix-vector operations like matrix-vector multiplications and ranked updates. The functions that HBLAS level-2 includes are listed below:

- gemv: General Matrix-Vector Multiplication
- symv: Symmetric Matrix-Vector Multiplication
- trmv: Triangular Matrx-Vector Multiplication
- trsv: Triangular Solve
- ger: General Rank-1 Update
- syr: Symmetric Rank-1 Update
- syr2: Symmetric Rank-2 Update

We now discuss the gemv (general matrix-vector multiplication), symv (symmetric matrix-vector multiplication), trmv (triangular matrix-vector multiplication) and trsv (triangular solve) functions of HBLAS level-2.

```
gemv :: (Num n) => Vector (Vector n) -> Vector n
                -> Vector n -> n -> n -> Vector n
gemv A x y a b=
    let x1 = fmap (dot x) A
        y1= scal b y
    in  axpy a x1 y1
```

The equation of the general matrix-vector multiplication is $y = \alpha Ax + \beta y$ where A is an n X n matrix, x is an n sized vector and α and β are two scalars. In the above function, (dot x) is applied to the matrix A by the higher-order function fmap and the resultant vector is called x1. The vector y gets scaled by scalar b with the use of level-1 function scal. Lastly, the axpy operation produces the final output vector using the intermediate vectors x1 and y1 and the scalar input a.

```
trmvLower :: ( Num n, Fractional n) =>
        Vector (Vector n) -> Vector n ->
        Vector n -> Int->Int-> Vector n
trmvLower a x b i n
        | i>=n      =  x
        | otherwise = let ai= take (i+1) (a  ! i)
                          bi= take (i+1) b
                          xi = trmvHelper ai bi
                          nx  = update x (pure(i, xi))
                      in trmvLower a nx b (i+1) n
      where trmvHelper veca vecx =
```

```
                sum ( zipWith (\x y -> x*y) veca vecx)

trmvUpper :: ( Num n, Fractional n) =>
        Vector (Vector n) -> Vector n ->
        Vector n -> Int->Int-> Vector n
trmvUpper a x b i n
        | i>=n      =  x
        | otherwise = let ai= drop i (a  V.! i)
                          bi= drop i b
                          xi = trmvHelper ai bi
                          nx  = update x (pure (i, xi))
                      in trmvUpper a nx b (i+1) n
        where trmvHelper veca vecx =
                sum ( zipWith (\x y -> x*y) veca vecx)
```

The equation of the triangular matrix-vector multiplication is x = A * x. The main challenge here is that matrix A here comes as either an upper triangular or lower triangular. An example of the lower triangular matrix is shown below.

$$A = \begin{pmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{21} & a_{22} & a_{23} & \\ & & . & \\ & & . & \\ & & . & \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$$

```
symvLower :: (Num n) => Vector (Vector n) -> Vector n
    -> Vector n -> n -> n -> Vector n
symvLower matA vecX vecY alpha beta =
    let n= length vecX
        mat= getLowerTotalMatrix matA 0 n
        x1 = fmap (dot vecX) mat
        y1= scal beta vecY
    in  axpy alpha x1 y1
getLowerTotalMatrix a i n
    | i>= n= empty <> empty
    | otherwise = pure (getLowerTotalRow a x i j n)
                <> getLowerTotalMatrix a (i+1) n
        where x= a ! i
              j=i
getLowerTotalRow a x i j n
```

```
    | i>= n = x
    | otherwise= let aij= ((a  V.! i)  V.! j)
                     x1= take i x
                     new_x=  snoc x1 aij
                  in getLowerTotalRow a new_x (i+1) j n
```

The symmetric matrix-vector multiplication function covers both the implementation challenge of gemv and trmv as the equation of symv function is $y = \alpha Ax + \beta y$ where A is a lower or upper symmetric matrix.

```
trsvLower :: ( Num n, Fractional n) =>
          Vector (Vector n) -> Vector n
          -> Vector n -> Int->Int->
          (Vector n, Vector n)
trsvLower a x b i n
    | i>=n      =  (x,b)
    | otherwise =
        let nbi = trsvHelper (a ! i) x (b ! i)
            nb= V.update b (pure (i, nbi))
            aii = ((a ! i) ! i)
            nx  = update x (pure (i, (nbi/aii)))
        in trsvLower a nx nb (i+1) n
    where
    trsvHelper veca vecx valueb=
    valueb -  sum ( zipWith (\x y -> x*y) veca vecx)
```

```
trsvUpper :: ( Num n, Fractional n) =>
          Vector (Vector n)-> Vector n ->
          Vector n -> Int->Int->
          (Vector n, Vector n)
trsvUpper a x b i n
    |  i<=n = (x,b)
    |  otherwise =
        let nbi = trsvHelper (a  ! i) x (b  ! i)
            nb= update b (pure (i, nbi))
            aii = ((a ! i) ! i)
            nx  = update x (pure (i, (nbi/aii)))
        in trsvUpper a nx nb (i-1) n
    where
    trsvHelper veca vecx valueb=
    valueb -  sum ( zipWith (\x y -> x*y) veca vecx)
```

To solve the equation of triangular-solve function ($(x = A^{-1} * b)$), we utilize a robust technique like Gaussian Elimination that solve this equation efficiently and we follow the pseudocode of row-oriented forward substitution for Lower and Upper Triangular Linear Systems [9].

4.2 GHC Libraries

Since we utilize both list and vector as data-structures, we have three different implementations of HBLAS using the following three Haskell packages:

– Data.List

– Data.Vector

– Data.Vector.Unboxed

4.2.1 Data.List.

Both of the HBLAS level-1 and level-2 implementations use the higher-order functions of Data.List extensively. Some functions of List-HBLAS level-1 and level-2 are given below.

```
scal :: (Num n) => n -> [n] -> [n]
scal alpha x= fmap (alpha*) x

axpy :: (Num n) => n -> [n] -> [n]
        -> [n]
axpy alpha xs ys =zipWith (\x y -> y+alpha*x) xs ys

dot :: (Num n) => [n] -> [n] -> n
dot a b = sum (zipWith (*) a b)

nrm2 :: (Num n, Floating n) => [n] -> n
nrm2 = sqrt . sum . fmap (\i -> i * i)

gemv :: (Num n) =>
        [[n]] -> [n] -> [n] -> n -> n -> [n]
```

```
gemv matA vecX vecY alpha beta=
    let x1 = fmap (dot vecX) matA
        y1= scal beta vecY
    in  axpy alpha x1 y1
```

4.2.2 Data.Vector.

The function definitions in the Boxed Vector version of the HBLAS have already been discussed in section 4.2.1 and 4.2.2. There are no visible differences in List-HBLAS and Vector-HBLAS except the type sinatures of the BLAS functions. The important thing to notice here is that the matrices are represented as a vector of vectors and therefore the type signature of a matrix is Vector (Vector n).

4.2.3 Data.Vector.Unboxed.

We got a major setback when we attempt to implement the HBLAS level-2 with this package. To apply the higher-order functions of the Vector.Unboxed module on matrices, GHC requires Unbox (Vector n) which is not possible since (Vector n) is not a primitive data type. An example level-2 function gemv given below to show the difference with the function definition of gemv using Data.Vector that was given in section 4.1.2.

```
gemv :: (Num n,  Unbox n, Fractional n) =>
    Vector n -> Vector n -> Vector n
gemv matA xs = h1 matA (replicate n 0) xs 0 n
    where n= length xs
    h2 veca vecx =
```

```
        sum (zipWith (\x y -> x*y) veca vecx)
h1 a x b i n
    | i>=n      =  x
    | otherwise = let
                ai= take n a
                xi = h2 ai b
                nx  = update x (singleton (i, xi))
                na = drop n a
            in h1 na nx b (i+1) n
```

4.3 Iterative Linear Solvers

We implement two real-world linear algorithms - Conjugate Gradient Method (CGM) and Transpose-Free Quasi Minimal Residual (TFQMR) Algorithm since both have largely called the level-1 and level-2 subprograms. In this section, we describe these algorithms and discuss why it is worth to implement these to test the performance of our HBLAS.

4.3.1 Conjugate Gradient Method.

We implement the CG Method importing our HBLAS level-1 and level-2. In this cgm function, we see there are several HBLAS level-1 subroutines are used - axpy, scal, gemv, dot, and the only function called from level-2 is gemv. There are 2 axpy operations, 3 dot products, 1 scaling, and 1 general matrix-vector multiplication operation in each iteration.

```
import HBLAS.Level1
```

```
import HBLAS.Level2

cgm :: (Num n, Fractional n, Ord n, Floating n) =>
       Vector(Vector n) -> Vector n ->
       Vector n -> Vector n
cgm a b vec0= cg n ( r, y, z, s, t, x)
  where tol = 1e-10
        n = length b
        minus1 = negate 1
        ab = gemv a b
        r = axpy minus1 ab b vec0
        y = scal minus1 r
        z = gemv a y
        s = dot y z
        t = (dot y r) / s
        x = axpy t y b vec0
        cg 0 ( _ , _ , _ , _ , _ , x1) =  x1
        cg m  ( r1, y1, z1, s1, t1, x1) =
          case (nrm2 r1 < tol) of
            True  -> b
            False ->
              let minust =  minus1 * t1
                  rr = axpy minust z1 r1 vec0
              in case (nrm2 rr < tol) of
                   True  ->  x1
```

```
                False ->
                  let bb = (dot rr z1) / s1
                      by = scal bb y1
                      yy = axpy minus1 rr by vec0
                      zz = gemv a yy
                      ss = dot yy zz
                      tt = (dot rr yy) / ss
                      xx = axpy tt yy x1 vec0
                  in cg (m-1) (rr,yy,zz,ss,tt,xx)
```

4.3.2 Transpose-Free Quasi Minimal Residual Solver.

TFQMR algorithm requires 2 matrix-vector products, 2 dot products, 2 vector norms, and 10 axpy operations per iteration. And therefore, like the CGM, it needed to utilize the subprograms from HBLAS level-1 and level-2.

```
import HBLAS.Level1
import HBLAS.Level2

tfqmr :: (Num n, Fractional n, Ord n, Floating n) =>
         Vector (Vector n) -> Vector n -> n ->
         Vector n -> Vector n
tfqmr a b tol vec0= qmr k (m, r, w, y1, d, v, u1,
 theta, eta, tau, rho, x)
  where x = vec0
        r = b
```

```
w = r
y1 = r
k = 1
d = vec0
v = gemv' a y1
u1 = v
theta = 0
eta = 0
tau = nrm2 r
rho = tau * tau
m = 0
qmr 100 (_,_,_,_,_,_,_,_,_,_,_,x') = x'
qmr k' (m',r',w',y1',d',v',u1',theta',eta',
tau',rho',x') =
  let sigma =  dot r' v'
      alpha = rho' / sigma
      j = 1
      mm' = 2 * k' - 2 + j
      ww' = axpy ((negate 1) * alpha)  u1' w'
      dd'= axpy (theta' * theta' * eta'
                / alpha) d' y1'
      ttheta' = (nrm2 ww') / tau'
      c = 1 / sqrt (1 + ttheta' * ttheta')
      ttau' = tau' * ttheta' * c
      eeta' = c * c * alpha
```

```
      xx' = axpy eeta'  dd' x'
  in case ((ttau' * sqrt (mm' + 1)) <  tol) of
      True  -> xx'
      False ->
        let jj = 2
            yy2 = axpy ((negate 1) *
                  alpha)  v' y1'
            uu2 = gemv' a yy2
            mm = 2 * k' - 2 + jj
            ww = axpy ((negate 1) *
                 alpha) uu2 ww'
            dd = axpy (ttheta' * ttheta' *
                 eeta' / alpha) dd' yy2
            ttheta = nrm2 ww / ttau'
            cc = 1/sqrt (1+ttheta*ttheta)
            ttau = ttau' * ttheta * cc
            eeta = cc * cc * alpha
            xx = axpy eeta  dd xx'
    in case ((ttau * sqrt (mm + 1)) < tol) of
         True  ->  xx
         False ->
           let rhon = dot r' ww
               beta = rhon / rho'
               rrho = rhon
               yy1 = axpy beta yy2 ww
```

```
        uu1 = gemv' a yy1
        vv = axpyaxpy' beta beta
                        v' uu2 uu1
    in qmr (k'+1) (mm,r',ww,yy1,
    dd,vv,uu1,ttheta,eeta,ttau,
    rrho,xx)
```

4.4 GHC Optimization Techniques

This section discusses the optimization techniques that we applied to get the best runtime performance from HBLAS. Several optimization techniques have been followed, for example- Bang Pattern and Function Inlining.

Instead of using a two-dimensional matrix (list of lists or vector of vectors) in the Unboxed version, we implemented HBLAS level-2 with plain/flattened one-dimensional vectors. Next, we applied several GHC language features to work around the issue Haskell language possesses.

The example use of bang patterns and function inlining in level-1 subprograms are shown below:

```
{-# LANGUAGE BangPatterns #-}
module HBLAS.Level1 where

import Data.Vector
```

```
{-# INLINE scal #-}
scal :: ( Num n) => n -> Vector n -> Vector n
scal !a !x= fmap (a*) x

{-# INLINE axpy #-}
axpy :: ( Num n) => n -> Vector n -> Vector n
        -> V.Vector n
axpy !a !xs !ys =zipWith (\x y -> y+a*x) xs ys

{-# INLINE dot #-}
dot :: ( Num n) => Vector n -> Vector n -> n
dot !a !b = sum (V.zipWith (*) a b)
```

CHAPTER V

RESULTS AND ANALYSIS

In this chapter, we evaluate the performance of the proposed functional linear algebra library, followed by a discussion of the results. We organize this chapter in six sections: 1) Experiment setup, 2) Performance of the HBLAS level-1 subprograms, 3) Performance of the HBLAS level-2 subprograms, 4) Performance of CGM and TFQMR using HBLAS, 5) Performance comparison of CGM and TFQMR using HBLAS and CBLAS, and 6) Discussion.

5.1 Experiment Setup

We consider execution time as the performance metric to analyze the performance of HBLAS functions and its applications. To measure the time taken by the functions, we import GHC's Data.Time [21] package. We then preserve the current time using getCurrentTime IO action and then call a HBLAS function to execute. We pass the HBLAS function as a parameter in the deepseq function [10] to force it to get executed first and store the time afterward. To get the difference between two time values in seconds we call diffUTCTime function. we repeat the HBLAS function call r=3 times with the help of nTimes function and then measure the average time taken through dividing the time difference by r. We also set a timeout value at 600 to halt executing any function that takes more than 10 minutes. The nTimes function and the code snippet of the execution time measurement are shown below:

```
nTimes 0 _ = return ()
```

```
nTimes !n !action =
 do
  action
  nTimes (n-1) action

t <- getCurrentTime
t' <- deepseq (nTimes r (tfqmr a b x)) getCurrentTime
putStrLn $ (show  (diffUTCTime t' t) )
```

We measure the performance of the HBLAS functions using Float vectors and matrices of a variety of sizes ranging from $1e6$ to $1e10$. We generate 20 values that are uniformly spaced between $1e6$ and $1e10$ to set the vector sizes. For matrices, we get 20 square matrices sized between $1e3 \times 1e3$ and $1e5 \times 1e5$ through the same technique of uniform distribution.

5.2 Performance of the HBLAS Level-1 subprograms

In this section, we measure the execution time of the HBLAS level-1 functions with three different data structures: list, boxed vector, and unboxed vector. Since we get very insignificant amount of execution time for the functions axpy, scal and rot, we measure only the rest of the four functions - Dot, asum, nrm2, and idamax.

In Table 4, with the increase of vector size, we see a linear increase of execution time for the dot product - a sum of the products of two vector elements. As expected, we find the most efficient version of HBLAS is of the Unboxed vector. The dot product takes only 19.4 seconds for unboxed-vectors of size 1e10 whereas

Table 4. Execution Time of DOT

Vector Size	Execution Time (s) for List-HBLAS	Execution Time (s) for Boxed HBLAS	Execution Time (s) for Unboxed HBLAS
1.00E+06	0.0136	0.0154	0.0014
5.27E+08	8.0943	9.1221	0.8772
1.05E+09	16.3426	18.4440	1.9399
1.58E+09	23.6761	27.1380	3.0490
2.11E+09	35.3366	36.4823	3.6983
2.63E+09	40.1223	45.8592	4.8141
3.16E+09	53.3720	54.9276	6.1904
3.69E+09	67.6957	63.9520	7.2231
4.21E+09	63.4449	75.7353	10.3639
4.74E+09	92.8163	83.4776	9.1747
5.26E+09	93.3068	93.0513	10.2221
5.79E+09	114.1341	102.0565	11.2512
6.32E+09	125.2438	112.4046	12.2793
6.84E+09	137.0837	121.9609	13.2954
7.37E+09	146.5888	133.6974	14.3776
7.90E+09	141.8912	139.3538	15.2830
8.42E+09	GHC Error	148.2027	16.3740
8.95E+09	GHC Error	163.5955	17.3894
9.47E+09	GHC Error	167.5684	18.3276
1.00E+10	GHC Error	183.1827	19.4313

the boxed vector version took 183.2s. On the other hand, we get a GHC bug for lists sized more than 7.9e9.

IDAMAX is the level-1 function that returns the index of the maximum absolute element of a vector. In Table 5, we see The unboxed version of HBLAS take the least amount of time among the three versions and List-HBLAS takes the longest.

The euclidean norm or 2-norm of vectors follows the same pattern as we experienced in the previous two experiments. We get the Unboxed-HBLAS

Table 5. Execution Time of IDAMAX

Vector Size	Execution Time (s) for List-HBLAS	Execution Time (s) for Boxed HBLAS	Execution Time (s) for Unboxed HBLAS
1.00E+06	0.0053	0.0026	0.0005
5.27E+08	2.9471	1.4872	0.2449
1.05E+09	6.8902	3.3882	0.7758
1.58E+09	9.1315	4.7014	1.1588
2.11E+09	12.0890	6.6664	1.5491
2.63E+09	15.5308	7.8993	2.1619
3.16E+09	17.4125	9.6859	2.5942
3.69E+09	21.1626	11.0257	3.0444
4.21E+09	22.7281	12.4104	3.4682
4.74E+09	34.1058	13.8142	3.4988
5.26E+09	31.4762	15.5977	3.8647
5.79E+09	34.6712	16.9950	4.2605
6.32E+09	46.0119	18.5506	4.7478
6.84E+09	40.8549	20.0590	5.1636
7.37E+09	54.7667	22.0947	5.4112
7.90E+09	57.8944	23.0230	5.8253
8.42E+09	GHC Error	24.9393	6.3091
8.95E+09	GHC Error	26.3779	6.5727
9.47E+09	GHC Error	28.0681	6.9984
1.00E+10	GHC Error	30.0006	7.3248

performing significantly better than any of the other two versions, List-HBLAS and Boxed-HBLAS.

With the increase of vector sizes, the execution time of the function ASUM in Table 7 soar dramatically in both the list and boxed vector HBLAS whereas it rises very steadily in unboxed-HBLAS and peaks at only 21.8s for the largest vector of size 1e10.

Table 6. Execution Time of NRM2

Vector Size	Execution Time (s) for List-HBLAS	Execution Time (s) for Boxed HBLAS	Execution Time (s) for Unboxed HBLAS
1.00E+06	0.0032	0.0025	0.0014
5.27E+08	2.2649	1.4551	0.8759
1.05E+09	4.2411	3.2340	1.7486
1.58E+09	7.1917	5.0766	3.0399
2.11E+09	8.3546	6.4659	3.9387
2.63E+09	10.3598	8.1863	5.1044
3.16E+09	12.4184	9.7532	6.1418
3.69E+09	15.2242	11.3082	7.2762
4.21E+09	17.7859	12.9274	8.1493
4.74E+09	18.7108	14.5031	9.2102
5.26E+09	20.7409	16.8757	10.1641
5.79E+09	37.8415	18.1681	11.3601
6.32E+09	50.9295	20.2054	12.2363
6.84E+09	27.5107	22.4048	13.2626
7.37E+09	59.4244	26.7350	14.2167
7.90E+09	34.6751	25.7963	15.3669
8.42E+09	GHC Error	30.9421	16.2560
8.95E+09	GHC Error	32.4590	17.3098
9.47E+09	GHC Error	34.4061	18.2856
1.00E+10	GHC Error	38.5329	19.4023

5.3 Performance of the HBLAS Level-2 subprograms

In this section, we measure the execution times of HBLAS level-2 functions with different data structures. Here we measure the matrix-vector multiplications - GEMV (General Matrix), TRMV (Triangular Matrix), and the Triangular Solve TRSV.

Table 8 illustrates the execution time taken by the GEMV function of Boxed Vector and Unboxed Vector. Here the execution time of List version of HBLAS shows an sharp rise and went up to 90.8 seconds with the largest matrix. We do

Table 7. Execution Time of ASUM

Vector Size	Execution Time (s) for List-HBLAS	Execution Time (s) for Boxed HBLAS	Execution Time (s) for Unboxed HBLAS
1.00E+06	0.0199	0.0139	0.0017
5.27E+08	6.8488	8.1325	1.0445
1.05E+09	13.8728	16.3640	2.2635
1.58E+09	21.4626	25.6532	3.2902
2.11E+09	26.1160	33.6896	4.4473
2.63E+09	35.1102	40.5713	5.6156
3.16E+09	46.4950	50.9036	6.9157
3.69E+09	60.4777	59.4162	8.1242
4.21E+09	62.5723	68.2004	9.3574
4.74E+09	59.5461	79.9798	11.1118
5.26E+09	69.6402	89.1633	12.2816
5.79E+09	71.7132	95.2222	13.1444
6.32E+09	83.5791	103.9076	14.9425
6.84E+09	101.6129	112.8902	16.0592
7.37E+09	112.0052	125.0961	17.3368
7.90E+09	118.7160	134.1248	18.5501
8.42E+09	GHC Error	144.3305	18.6361
8.95E+09	GHC Error	153.3626	20.1479
9.47E+09	GHC Error	163.5726	20.7134
1.00E+10	GHC Error	172.0216	21.8049

not include the performance results of the Vector version since it went beyond the timeout limit for the fourth matrix (2.7e8). Conversely, the unboxed HBLAS showed a very gradual increase with the execution time over the matrix sizes.

We see a moderate increase in the execution time of Triangular Matrix-Vector Multiplication with Boxed Vector and it peaked at only 25 seconds for the largest matrix. On the other hand, we find a sharp rise in the List version of HBLAS and hit a high of 527.9s for 1e5X1e5 matrix. And, because of using the recursive functions in the implementation of Unboxed version, we get a significantly worse

Table 8. Execution Time of GEMV

Matrix Size	Execution Time (s) for List	Execution Time (s) for Unboxed Vector
1.00E+06	0.0072	0.0024
3.86E+07	0.2671	0.0783
1.30E+08	0.9710	0.2718
2.77E+08	2.1528	0.6485
4.77E+08	3.8342	1.1246
7.32E+08	6.0211	1.9356
1.04E+09	8.7557	3.0208
1.40E+09	11.8806	4.0661
1.82E+09	15.5568	5.1779
2.29E+09	19.8963	6.5477
2.82E+09	24.4420	8.2411
3.40E+09	29.9740	9.1727
4.04E+09	35.5228	11.4703
4.73E+09	41.4560	13.6859
5.47E+09	49.1067	16.0442
6.27E+09	56.3178	18.2436
7.12E+09	63.4974	21.0443
8.02E+09	72.2399	23.3883
8.99E+09	81.1139	25.1283
1.00E+10	90.8859	27.7812

performance compared to the List-HBLAS and Boxed-HBLAS and therefore, we fail to measure time for the TRMV of Unboxed-HBLAS.

For the triangular solve function that solves the linear equation $x = A^{-1}b$ we get a moderate increase in the execution time with the Boxed vector version of HBLAS. However, we see an exponential rise in List-HBLAS and the execution time went above the timeout limit with only 5.31e4X5.31e4 = 2.8e9 sized matrix.

Table 9. Execution Time of TRMV

Matrix Size	Execution Time (s) for List	Execution Time (s) for Boxed Vector
1.00E+06	0.0338	0.0008
3.86E+07	1.6797	0.0365
1.30E+08	6.2925	0.1332
2.77E+08	14.2730	0.3112
4.77E+08	24.1936	0.5930
7.32E+08	33.5384	0.9810
1.04E+09	50.7913	1.3745
1.40E+09	65.5770	2.1034
1.82E+09	77.7311	2.6959
2.29E+09	111.2142	3.9815
2.82E+09	127.8541	5.0256
3.40E+09	143.2380	6.0537
4.04E+09	201.2461	7.2296
4.73E+09	219.8811	11.3527
5.47E+09	254.4297	12.8577
6.27E+09	269.1939	15.0296
7.12E+09	413.7953	17.0665
8.02E+09	499.2131	19.2780
8.99E+09	521.3274	21.2068
1.00E+10	527.9201	25.3213

5.4 Performance of Iterative Solvers using HBLAS

In this section, we experiment on the performance of our implemented HBLAS in two real-life applications. We implement two linear iterative solver algorithms that called the HBLAS levels 1 and 2 functions extensively. The two algorithms are CGM (Conjugate Gradient Method) and TFQMR (Transpose-Free Quasi Minimal Residual) algorithm.

We measure the execution time taken by the CGM using the three HBLAS versions. For Boxed Vectors, we get the execution time more than the timeout

Table 10. Execution Time of TRSV

Matrix Size	Execution Time (s) for List	Execution Time (s) for Boxed Vector
1.00E+06	0.0538	0.0035
3.86E+07	4.2761	0.1458
1.30E+08	17.6554	0.4836
2.77E+08	45.4645	1.4950
4.77E+08	83.8043	2.0436
7.32E+08	129.3425	3.1405
1.04E+09	187.3335	4.0442
1.40E+09	250.6146	6.6485
1.82E+09	359.5572	7.2996
2.29E+09	462.6991	9.5906
2.82E+09	Timeout	11.8592
3.40E+09	Timeout	13.5047
4.04E+09	Timeout	18.3504
4.73E+09	Timeout	21.8102
5.47E+09	Timeout	23.3611
6.27E+09	Timeout	25.8556
7.12E+09	Timeout	36.4585
8.02E+09	Timeout	40.5659
8.99E+09	Timeout	43.4525
1.00E+10	Timeout	44.7483

limit for a small matrix of size 2.7e8, and therefore we do not include the boxed vector version here. The Unboxed CGM showed a gradual increase with the increase of matrix sizes and peaked at 163.8s for 1e5X1e5 matrix. However, we get a very high execution time values for the List version of CGM.

In the performance measurement table (Table-12) of the iterative solver TFQMR, as expected, we find the Unboxed version of TFQMR performing much better than the List version. For the largest matrix, Unboxed-TFQMR takes only 83 seconds whereas we get the List-TFQMR quite slower and crossed the timeout threshold with only 3.40e9 matrix.

Table 11. Execution Time of CGM using HBLAS

Matrix Size	Execution Time (s) for List	Execution Time (s) for Unboxed Vector
1.00E+06	0.08	0.01
3.86E+07	4.54	0.37
1.30E+08	8.57	1.32
2.77E+08	47.23	2.87
4.77E+08	71.57	6.00
7.32E+08	57.91	9.36
1.04E+09	220.88	13.43
1.40E+09	266.49	19.45
1.82E+09	314.89	23.57
2.29E+09	219.23	31.56
2.82E+09	585.59	36.56
3.40E+09	308.41	52.75
4.04E+09	Timeout	51.66
4.73E+09	Timeout	64.19
5.47E+09	Timeout	81.77
6.27E+09	Timeout	80.92
7.12E+09	Timeout	109.50
8.02E+09	Timeout	124.29
8.99E+09	Timeout	143.01
1.00E+10	Timeout	163.82

5.5 Performance comparison between HBLAS and CBLAS

In this section we compare the performance of both the iterative solver algorithms CGM and TFQMR with respect to iterative implementation and functional implementation. The former is a hand-written C program that calls the LAPACK subprograms using the cblas interface. The latter one is the same Haskell programs that we considered in the previous section. In the implementation of CGM and TFQMR with C, we measure the execution time with the same set of matrices.

Table 12. Execution Time of TFQMR using HBLAS

Matrix Size	Execution Time (s) for List	Execution Time (s) for Unboxed Vector
1.00E+06	0.04	0.00
3.86E+07	3.41	0.22
1.30E+08	6.76	1.05
2.77E+08	42.46	2.38
4.77E+08	47.87	4.25
7.32E+08	32.27	6.89
1.04E+09	166.11	9.95
1.40E+09	190.34	12.69
1.82E+09	119.21	21.37
2.29E+09	443.55	21.17
2.82E+09	434.68	25.26
3.40E+09	Timeout	30.84
4.04E+09	Timeout	47.28
4.73E+09	Timeout	43.23
5.47E+09	Timeout	59.09
6.27E+09	Timeout	69.32
7.12E+09	Timeout	80.79
8.02E+09	Timeout	72.59
8.99E+09	Timeout	83.00
1.00E+10	Timeout	93.84

In Figure 1, we depict the performance comparison between the CGM implementation using CBLAS and using Unboxed-HBLAS. With the increase of matrix sizes, the HBLAS-CGM shows a consistent sharp rise in the execution time. In the case of CBLAS, we see a fluctuation in the execution time with a very minimal increase. CBLAS is so optimized that the execution time never went above 50 seconds whereas with Unboxed HBLAS, the execution time increases exponentially and hit the level of around 165s.

Figure 2 illustrates the performance comparison between the two version of TFQMR implementation—one with the C language calling CBLAS subprograms

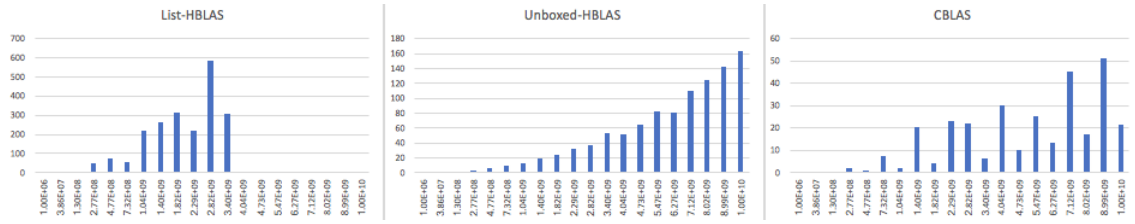

Figure 1. Performance Comparison between CBLAS CGM and HBLAS CGM.

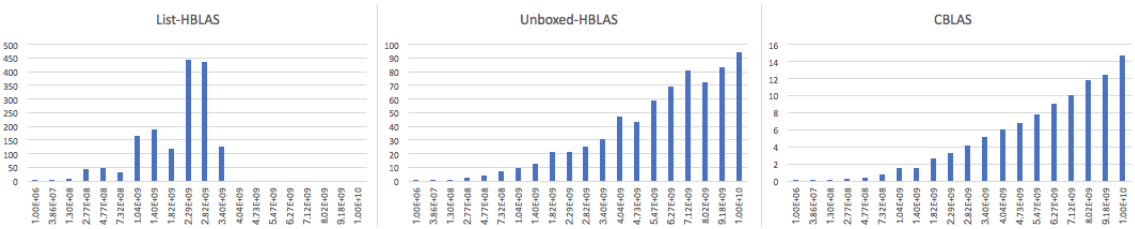

Figure 2. Performance Comparison between CBLAS TFQMR and HBLAS TFQMR.

and the other with Haskell through calling HBLAS functions. In Unboxed HBLAS, we see the similar upward curve as we saw in Figure 1. However, we find the execution time of CBLAS rises very minimally and stays below 25s even for the largest matrix. One interesting finding we get here, for the matrix of size 8.99E+09, we get an execution time that is larger than the timeout limit and therefore we discard that value.

Table-3 and Table-4 presents the normalized values of List-HBLAS and Unboxed Vector - HBLAS. Here, we consider the cBLAS execution times as base value and normalize the execution times of List-HBLAs and Unboxed-HBLAS to analyze the ratio that how slow the HBLAS-CGM and HBLAS-TFQMR performed compared to the CBLAS ones.

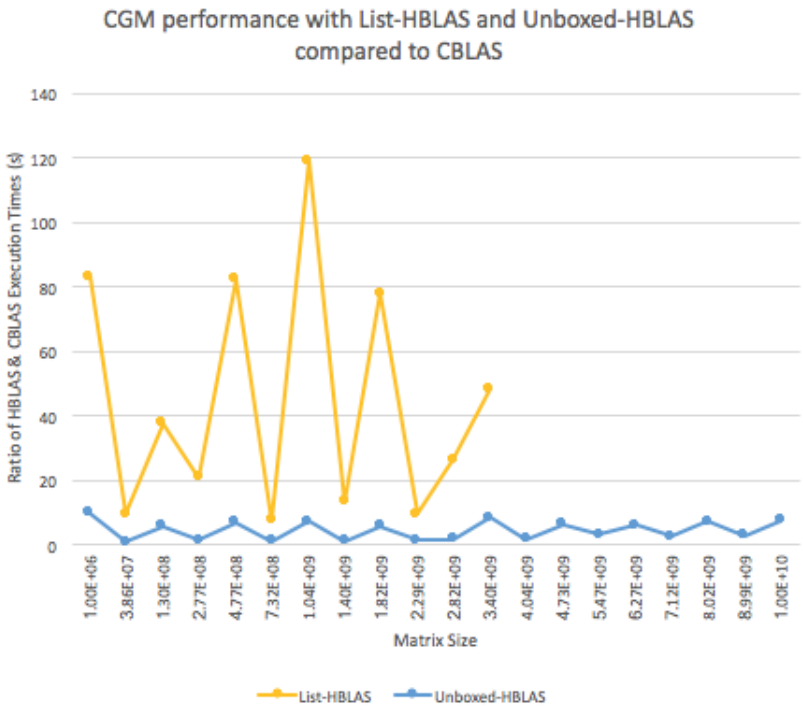

Figure 3. Performance of CGM in List-HBLAS and Unboxed-HBLAS normalized by CBLAS.

5.6 Discussion

In this section, we discuss the findings we get from the performance experiments that we showed in the previous sections. In Figure-3 and 4, we see both the List-HBLAS and the Unboxed-HBLAS are substantially slower than the CBLAS. The Unboxed-HBLAS is on average 4.4 times slower than the cBLAS one in Conjugate Gradient and 7.7 times slower in TFQMR. The List-HBLAS showed a fluctuated performance in both CGM and TFQMR with the matrix increases. However, the execution time with CBLAS has also shown an irregular oscillation with a little increase.

The CBLAS we used in the implementation of the iterative solvers in C language is an API and the underlying BLAS library we used is provided through

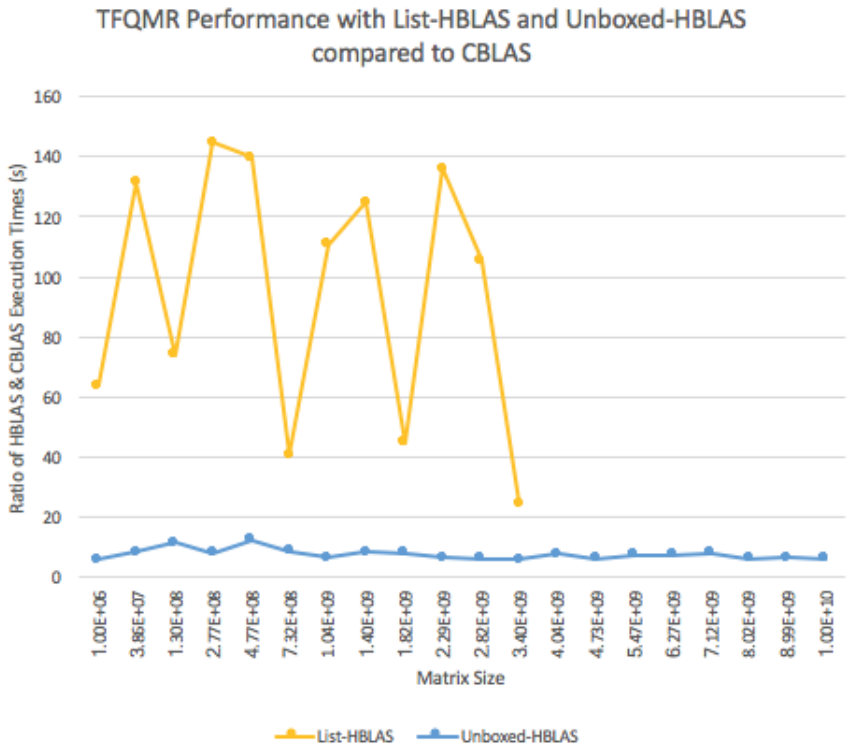

Figure 4. Performance of TFQMR in List-HBLAS and Unboxed-HBLAS normalized by CBLAS.

LAPACK [28]. All the BLAS libraries are made optimized and numerically stable by exploiting AVX and multi-threading and therefore they significantly boost performance on large datasets. For example, we got a very minimal rise in the execution times over the increase of the matrix sizes and it peaked at only 14 seconds with the largest matrix. We assume, CBLAS gets more optimized with the matrix sizes increase.

The list-HBLAS showed an upward trend with abrupt fluctuation (figure-3 and Figure-4) and was much slower than the unboxed version. In both the CGM and TFQMR, we found the execution time dropped with 3.4e9 matrix whereas it was at the highest for the immediate smaller matrix (size 2.82e9). Haskell lists are

single-linked lists and thus the insertion and update to the position i of a list of length n have complexity $O(i)$. And therefore, as the list size gets bigger, the functions that process the large lists get slower. However, the explanation of why the run-time performance got better for the matrix size of 3.4e9, is still not fully understood. The server where we ran our HBLAS performance experiment has L1 cache size 32KB. And the memory usages of the matrix size of 2.82e9 and 3.4e9 are around 11.28GB and 13.61GB, respectively. For both matrices, we observe similar L1 cache miss rates, 352584 and 425152.8, respectively. Although there were more cache misses for the second matrix, the runtime was shorter. We ran profiling to investigate why the larger matrix takes less execution time. Since the large matrix profiling takes too much time, we took another pair of matrices (4.77e8 and 7.32e8) that showed same trend. The time profiling reports are given in Figure 5 and 6. Here we can see the only bottlenecked BLAS function is gemv (matrix-vector multiplication). Even though the matrix is larger in Figure-6 we see that the gemv functions took equal amount of time for either of the matrices. We believe, the reason behind this unexpected behavior can be the list fusion [5].

In Figure-3 and 4, the Unboxed Vector version of HBLAS is shown to be better version than the List. This data structure provides strict evaluation and so, an unboxed value can never be unevaluated even though it is a functional language with laziness feature. Unboxed Vectors provide $O(1)$ insertion and update since the values are stored more efficiently- consecutive memory slots without pointers.

However, the Unboxed Vector version of HBLAS showed slower execution times than the CBLAS, although the performance difference is not that significant. We already know that functional language suffers from the lazy evaluation problem

```
        total time  =     1138.22 secs   (1138217 ticks @ 1000 us, 1 processor)
        total alloc = 1,137,492,040,248 bytes  (excludes profiling overheads)

COST CENTRE MODULE    SRC                  %time %alloc

gemv        Main      tfqmr05.hs:46:47-57   48.1   44.3
gemv        Main      tfqmr05.hs:62:54-64   25.1   22.1
gemv        Main      tfqmr05.hs:20:33-42   20.8   22.1
main.a      Main      tfqmr05.hs:79:15-37    6.0   11.4


                                                                                  individual     inherited
COST CENTRE  MODULE                                  SRC                     no.     entries  %time %alloc  %time %alloc

MAIN         MAIN                                    <built-in>              139           0    0.0    0.0   100.0  100.0
 CAF         Main                                    <entire-module>         274           0    0.0    0.0     6.0   11.4
  main       Main                                    tfqmr05.hs:(68,1)-(92,98) 278         1    0.0    0.0     6.0   11.4
   main.a    Main                                    tfqmr05.hs:79:15-37     282           1    6.0   11.4     6.0   11.4
   main.r    Main                                    tfqmr05.hs:74:15-19     306           1    0.0    0.0     0.0    0.0
   main.tol  Main                                    tfqmr05.hs:72:15-25     283           1    0.0    0.0     0.0    0.0
   main.vecX Main                                    tfqmr05.hs:77:15-38     280           1    0.0    0.0     0.0    0.0
   main.vecY Main                                    tfqmr05.hs:78:15-37     281           1    0.0    0.0     0.0    0.0
 CAF         Data.Time.Clock.Internal.NominalDiffTime <entire-module>         272           0    0.0    0.0     0.0    0.0
 CAF         Data.Time.Clock.System                  <entire-module>         268           0    0.0    0.0     0.0    0.0
 CAF         Data.Fixed                              <entire-module>         260           0    0.0    0.0     0.0    0.0
 CAF         GHC.Conc.Signal                         <entire-module>         241           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Encoding                         <entire-module>         227           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Encoding.Iconv                   <entire-module>         225           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Handle.FD                        <entire-module>         217           0    0.0    0.0     0.0    0.0
 CAF         GHC.Show                                <entire-module>         202           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Handle.Text                      <entire-module>         164           0    0.0    0.0     0.0    0.0
 main        Main                                    tfqmr05.hs:(68,1)-(92,98) 279         0    0.0    0.0    94.0   88.6
  gemv       Main                                    tfqmr05.hs:20:33-42     284           3   20.8   22.1    20.8   22.1
  nrm2       Main                                    tfqmr05.hs:24:35-40     285           3    0.0    0.0     0.0    0.0
  qmr        Main                                    tfqmr05.hs:13:43-96     286           3    0.0    0.0    73.2   66.5
   axpy      Main                                    tfqmr05.hs:54:46-62     300           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr05.hs:48:46-78     296           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr05.hs:45:47-79     294           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr05.hs:40:41-58     293           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr05.hs:34:41-73     289           6    0.0    0.0     0.0    0.0
   axpy_div  Main                                    tfqmr05.hs:49:50-96     299           6    0.0    0.0     0.0    0.0
   axpy_div  Main                                    tfqmr05.hs:35:45-87     292           6    0.0    0.0     0.0    0.0
   div       Main                                    tfqmr05.hs:31:42-53     288           6    0.0    0.0     0.0    0.0
   div_sqrt  Main                                    tfqmr05.hs:51:50-79     298           6    0.0    0.0     0.0    0.0
   div_sqrt  Main                                    tfqmr05.hs:37:43-74     291           6    0.0    0.0     0.0    0.0
   dot       Main                                    tfqmr05.hs:30:43-51     287           6    0.0    0.0     0.0    0.0
   gemv      Main                                    tfqmr05.hs:46:47-57     295           6   48.1   44.3    48.1   44.3
   nrm2_div  Main                                    tfqmr05.hs:50:54-68     297           6    0.0    0.0     0.0    0.0
   nrm2_div  Main                                    tfqmr05.hs:36:49-65     290           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr05.hs:61:54-69     303           3    0.0    0.0     0.0    0.0
   axpy_axpy Main                                    tfqmr05.hs:63:58-88     305           3    0.0    0.0     0.0    0.0
   div       Main                                    tfqmr05.hs:59:54-64     302           3    0.0    0.0     0.0    0.0
   dot       Main                                    tfqmr05.hs:58:54-62     301           3    0.0    0.0     0.0    0.0
   gemv      Main                                    tfqmr05.hs:62:54-64     304           3   25.1   22.1    25.1   22.1
```

Figure 5. Time Profiling Report of TFQMR with List-HBLAS of matrix size 4.77e8.

which causes memory consumption and thus execution time latency. And to avoid this limitation, we used techniques like deepseq, bang patterns, strict higher-order functions. We suspect a few issues of Haskell can make our Unboxed-HBLAS slower - data immutability, higher-order functions, and lack of memory management. On the other hand, CBLAS not only uses vectorizations, but also optimally exploits available vector extensions (SSE, AVX), multiple cores, and cache reuse. We failed to do the profiling for Unboxed version of HBLAS applications. Otherwise, we could get to know about the HBLAS function for which the iterative solvers take the majority portion of the execution times.

```
        total time  =     1654.12 secs   (1654121 ticks @ 1000 us, 1 processor)
        total alloc = 1,744,822,513,808 bytes  (excludes profiling overheads)

COST CENTRE MODULE    SRC                %time %alloc

gemv        Main      tfqmr06.hs:46:47-57   48.0   44.3
gemv        Main      tfqmr06.hs:62:54-64   24.5   22.1
gemv        Main      tfqmr06.hs:20:33-42   21.0   22.1
main.a      Main      tfqmr06.hs:79:15-37    6.4   11.4


                                                                                 individual      inherited
COST CENTRE  MODULE                                  SRC                      no.     entries  %time %alloc   %time %alloc

MAIN         MAIN                                    <built-in>               139           0    0.0    0.0   100.0  100.0
 CAF         Main                                    <entire-module>          274           0    0.0    0.0     6.4   11.4
  main       Main                                    tfqmr06.hs:(68,1)-(92,98) 278          1    0.0    0.0     6.4   11.4
   main.a    Main                                    tfqmr06.hs:79:15-37      282           1    6.4   11.4     6.4   11.4
   main.r    Main                                    tfqmr06.hs:74:15-19      306           1    0.0    0.0     0.0    0.0
   main.tol  Main                                    tfqmr06.hs:72:15-25      283           1    0.0    0.0     0.0    0.0
   main.vecX Main                                    tfqmr06.hs:77:15-38      280           1    0.0    0.0     0.0    0.0
   main.vecY Main                                    tfqmr06.hs:78:15-37      281           1    0.0    0.0     0.0    0.0
 CAF         Data.Time.Clock.Internal.NominalDiffTime <entire-module>          272           0    0.0    0.0     0.0    0.0
 CAF         Data.Time.Clock.System                  <entire-module>          268           0    0.0    0.0     0.0    0.0
 CAF         Data.Fixed                              <entire-module>          260           0    0.0    0.0     0.0    0.0
 CAF         GHC.Conc.Signal                         <entire-module>          241           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Encoding                         <entire-module>          227           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Encoding.Iconv                   <entire-module>          225           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Handle.FD                        <entire-module>          217           0    0.0    0.0     0.0    0.0
 CAF         GHC.Show                                <entire-module>          202           0    0.0    0.0     0.0    0.0
 CAF         GHC.IO.Handle.Text                      <entire-module>          164           0    0.0    0.0     0.0    0.0
 main        Main                                    tfqmr06.hs:(68,1)-(92,98) 279          0    0.0    0.0    93.6   88.6
  gemv       Main                                    tfqmr06.hs:20:33-42      284           3   21.0   22.1    21.0   22.1
  nrm2       Main                                    tfqmr06.hs:24:35-40      285           3    0.0    0.0     0.0    0.0
  qmr        Main                                    tfqmr06.hs:13:43-96      286           3    0.0    0.0    72.6   66.4
   axpy      Main                                    tfqmr06.hs:54:46-62      300           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr06.hs:48:46-78      296           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr06.hs:45:47-79      294           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr06.hs:40:41-58      293           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr06.hs:34:41-73      289           6    0.0    0.0     0.0    0.0
   axpy_div  Main                                    tfqmr06.hs:49:50-96      299           6    0.0    0.0     0.0    0.0
   axpy_div  Main                                    tfqmr06.hs:35:45-87      292           6    0.0    0.0     0.0    0.0
   div       Main                                    tfqmr06.hs:31:42-53      288           6    0.0    0.0     0.0    0.0
   div_sqrt  Main                                    tfqmr06.hs:51:50-79      298           6    0.0    0.0     0.0    0.0
   div_sqrt  Main                                    tfqmr06.hs:37:43-74      291           6    0.0    0.0     0.0    0.0
   dot       Main                                    tfqmr06.hs:30:43-51      287           6    0.0    0.0     0.0    0.0
   gemv      Main                                    tfqmr06.hs:46:47-57      295           6   48.0   44.3    48.0   44.3
   nrm2_div  Main                                    tfqmr06.hs:50:54-68      297           6    0.0    0.0     0.0    0.0
   nrm2_div  Main                                    tfqmr06.hs:36:49-65      290           6    0.0    0.0     0.0    0.0
   axpy      Main                                    tfqmr06.hs:61:54-69      303           3    0.0    0.0     0.0    0.0
   axpy_axpy Main                                    tfqmr06.hs:63:58-88      305           3    0.0    0.0     0.0    0.0
   div       Main                                    tfqmr06.hs:59:54-64      302           3    0.0    0.0     0.0    0.0
   dot       Main                                    tfqmr06.hs:58:54-62      301           3    0.0    0.0     0.0    0.0
   gemv      Main                                    tfqmr06.hs:62:54-64      304           3   24.5   22.1    24.5   22.1
```

Figure 6. Time Profiling Report of TFQMR with List-HBLAS of matrix size 7.32e8.

One important thing is that with the increase of matrix size, the ratio of the execution time of Unboxed-HBLAS and CBLAs hovered around the same range in the Figure-3 and 4. In CGM, we found that Unboxed-HBLAS is faster or almost equal to the CBLAS for some matrices sized 3.86e7, 2.77e8, 7.32e8, 1.4e9, 2.29e9, 2.82e9, and 4.04e9. In summary, the Unboxed-HBLAS performance is significantly better in CGM than in TFQMR. One possible reason of this is that the number of called HBLAS subprograms from TFQMR iterations is more than in CGM. Apart from this, there is a good chance that in TFQMR we did not get that much of stream fusion done although we have utilized the most efficient data structure.

CHAPTER VI

CONCLUSION AND FUTURE WORK

In this thesis, we attempted to recreate CBLAS with a Functional Language Haskell and optimize it with the help of efficient data structures and GHC optimization techniques. We named the basic linear algebra library HBLAS and we also implemented two linear iterative solver algorithms, CGM and TFQMR, that call the HBLAS subprograms. We applied different GHC language features like bang patterns and function inlining to optimize the library subroutines. For benchmarking, we considered the hand-written C programs of CGM and TFQMR which calls CBLAS subprograms. We then compared the performance of the functional iterative solvers with the performance of the imperative ones. We developed the HBLAS with three different data structures: List, Boxed Vector, and Unboxed Vector, to study and compare the performance of different data structures. In the experimental results, we found the unboxed vector version of HBLAS is faster than the list and boxed vector ones. With the known benefits of the unboxed vector, we assume the use of one-dimensional matrices in the level-2 implementation helped us to get better performance than the other two HBLAS versions. In the performance experiment, we found the execution times of HBLAS-CGM and HBLAS-TFQMR are on average 4.4 and 7.7 times slower than that of the CBLAS-CGM and CBLAS-TFQMR respectively. Though our purpose was not to beat CBLAS, yet we intended to make HBLAS as much optimized as possible.

There could be many new research directions we find from this thesis. One big limitation that we failed to profile the unboxed-HBLAS applications leaves us with

a concrete question, which HBLAs function took the majority portion of run-time and memory, and thus we could target those functions to optimize more. We also plan to use the parallel arrays of repa package [27] where all numeric data is stored as unboxed in the implementation of HBLAS. We hypothesize that with the use of repa arrays, we can make our HBLAS's performance far better than we have got in this thesis work. we also intend to implement the BLAS level-3 and combine the complete HBLAS package with the official GHC compiler. Our main goal is to make HBLAS useful for the scientific programmers who require not only a functional language's code clarity but also need better performance.

REFERENCES CITED

[1] M. Akter. Hblas: Basic linear algebra subprograms in haskell. `https://github.com/mamtajakter/HBLAS`, 2020.

[2] D. Stewart B. O'Sullivan and J. Goerzen. *Real World Haskell.* O'Reilly Media, 2008.

[3] S. Boyd. Convex optimization, 2018.

[4] Glasgow Haskell Compiler. Accelerate: An embedded language for accelerated array processing.

[5] R. Leshchinsky D. Coutts and D. Stewart. Stream fusion: From lists to streams to nothing at all. *In Proceedings of International Conference of Functional Programming*, 2007.

[6] L. S. Blackford et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2002.

[7] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems. *SIAM journal on scientific computing*, 14(2):470–482, 1993.

[8] R. Leshchinsky G. MainLand and S. Peyton Jones. Exploiting vector instructions with generalized stream fusion. *Communications of the ACM*, 60:83–91, 2017.

[9] E. Haber. Forward substitution for lower triangular linear systems.

[10] HaskellWiki. Control.deepseq module: Deep evaluation of data structures.

[11] J. Hughes. The design and implementation of programming languages. *A Ph.D. Thesis submitted to the University of Oxford*, 1983.

[12] J. Hughes. Why functional programming matters. *The Computer Journal*, 32:98–107, 1989.

[13] R. Leshchinskiy. Data.vector.unboxed: Adaptive unboxed vectors.

[14] Miran Lipovaca. *Learn You a Haskell for Geat Good.* April 2011.

[15] The University of Glasgow. Data.list.

[16] D. Lester S. P. Jones. Implementing functional languages: A tutorial. 2000/04/23.

[17] T. Hoare S. P. Jones, A. Tolmach. Playing by the rules: Rewriting as a practical optimisation technique in ghc. *In Proceedings of the ACM SIGPLAN International Conference on Functional programming*, 2001.

[18] C. T. Schonwald. hblas: Human friendly blas and lapack bindings for haskell., 2018.

[19] L. Sheehan. Tail call optimization.

[20] W. Stoye. The implementation of functional languages using custom hardware. *Technical Report at the University of Cambridge*, 1985.

[21] Glasgow Haskell Compiler Team. Data.time.

[22] Glasgow Haskell Compiler Team. Foldr foldl foldl'.

[23] Glasgow Haskell Compiler Team. Ghc language features: The bang patterns.

[24] Glasgow Haskell Compiler Team. The glassgow haskell compiler.

[25] Glasgow Haskell Compiler Team. Inlining and specialisation.

[26] Glasgow Haskell Compiler Team. The vector package.

[27] The DPH Team. repa: High performance, regular, shape polymorphic parallel arrays.

[28] The University of California Berkeley The University of Tennessee and The University of Colorado Denver. Lapack: Linear algebra package.