Towards a functional run-time for dense NLA domain

Mauro Blanco Pablo Perdomo Pablo Ezzatti Alberto Pardo Marcos Viera

Instituto de Computación, Universidad de la República, Montevideo, Uruguay

{mblanco,pperdomo,pezzatti,pardo,mviera}@fing.edu.uy

Abstract

We investigate the use of functional programming to develop a numerical linear algebra run-time; i.e. a framework where the solvers can be adapted easily to different contexts and task parallelism can be attained (semi-) automatically. We follow a bottom up strategy, where the first step is the design and implementation of a framework layer, composed by a functional version of BLAS (Basic Linear Algebra Subprograms) routines. The framework allows the manipulation of arbitrary representations for matrices and vectors and it is also possible to write and combine multiple implementations of BLAS operations based on different algorithms and parallelism strategies. Using this framework, we implement a functional version of Cholesky factorization, which serves as a proof of concept to evaluate the flexibility and performance of our approach.

Categories and Subject Descriptors D.1.1 [*Programming techniques*]: Applicative (Functional) Programming; D.1.3 [*Programming techniques*]: Parallel Programming

Keywords Parallelism; Haskell; NLA; BLAS

1. Introduction

Numerical Linear Algebra (NLA) is a research area that in the last 40 years has been characterized by the use of libraries that are *de facto* standards. Examples are the use of BLAS in the case of basic operations, such as vector or matrix multiplication, and LAPACK (Linear Algebra Package) or ScaLAPACK (Scalable LAPACK [11]) in the case of NLA basic problems, like the solution of linear systems, linear-least squares problems, and eigenvalue computations.

These libraries are the kernel of several scientific programs, and in general, the most costly stage of them in execution time. This situation has motivated several efforts to improve the performance of these building blocks. In this line, in the 90s, the concern on studying automatic optimizations of BLAS routines started to grow. Examples of these efforts are PHiPAC[1] [6] and ATLAS[2] [24], which use empirical performance experiments to tune different routine parameters.

[1] Portable High Performance ANSI C

[2] Automatically Tuned Linear Algebra Software

More recently, the problem of tuning has become even more decisive. This is due to the increasing adoption of heterogeneous hardware like graphic (GPUs) or ARM processors for high performance computing platforms, and the associated propagation of multi-objective approaches to problem requirements. That is, in addition to the traditional goal of reducing execution time, nowadays other requirements, such as low energy consumption and good ratios of GFLOPs (Giga FLoating-point Operations Per Second) per dollar, are also considered important.

To address this problem the NLA community has studied several approaches. One of the most promising ones is the introduction of *run-time systems*. These are frameworks where solvers can be easily adapted to different contexts and task parallelism can be attained (semi-)automatically. Such systems typically decompose a problem in two stages: (i) revealing the dependency graph, and (ii) performing the best possible computation out of it. For the first stage two different approaches are used. One is to perform an analysis of the task dependencies to build the dependency graph, whereas the other is based on annotating the data dependencies on the source code through a simple set of clauses. In the second stage, a dynamic scheduling guides the computations. The scheduling is built around the constructed dependency graph and the hardware platform features. With this strategy a good level of task parallelism can be attained on each hardware platform. Some examples in the NLA field are the projects SuperMatrix–FLAME [10], MAGMA-PLASMA[3] [2], and SMPSs[4] [1].

Considering this trend, in this paper we investigate the use of functional programming to develop a NLA run-time. Our development follows a bottom up strategy, where the first step is the design of a layer composed by a functional framework for the parallel implementation of BLAS operations. This framework, developed in Haskell [22], provides us with the necessary infrastructure to write multiple versions of BLAS operations, each one based on a determinate representation for matrices and vectors, a particular algorithm and parallelism strategy. Aiming at the validation of this layer, we present the implementation of the *Cholesky factorization*, a well-known NLA operation that belongs to the LAPACK library, as a proof of concept. The goal is to evaluate the flexibility of our framework around this example. With such a flexible framework we are able to easily implement different variants of the algorithm in order to compare them in terms of performance. We show some benchmarks obtained by running the Cholesky factorization with different libraries for parallel programming in Haskell, different parallelism strategies, and several matrix and vector representations.

This work is the first step towards the development of a second layer (on top of the first one), a framework for LAPACK, with similar qualities as the framework for BLAS, i.e. flexibility in terms

[3] Parallel Linear Algebra for Scalable Multi-core Architectures

[4] SMP superscalar

of parallelism strategy and algorithm, and independence of the data representation.

The rest of the paper is structured as follows. In Section 2 we describe our approach and the running example employed in this work, the Cholesky factorization, to develop a proof of concept. In Section 3, we present a brief summary of the state of the art in parallel tools in Haskell. The first layer of our framework, i.e. the functional BLAS routines, is presented in Section 4. After that, in Section 5, we introduce the functional implementation of the Cholesky factorization built in terms of our framework. This is followed by experimental results in Section 6. Finally, in Section 7, concluding remarks and future work are exposed.

2. Our approach

Before delving into details, we show how our approach works. As we stated previously, the problem we have chosen as running example is the *Cholesky factorization* [14]. This operation takes a symmetric positive definite (SPD) matrix A and returns a triangular inferior matrix L, such that $L * L^T = A$ (or equivalently, a triangular superior matrix U such that $U^T * U = A$). This is a well-known operation in the NLA field, implemented in the LAPACK library [3] by the POTRF family of routines. The implementation of these routines requires the invocations of several BLAS operations:

- DOT: scalar product
- SCAL: vector scale
- GEMV: general matrix-vector multiplication
- GEMM: general matrix-matrix multiplication
- SYRK: rank k update of symmetric matrix
- TRSM: triangular system resolution

There exists a myriad of possibly implementations of these operations, varying on the used algorithm, structure representation, parallelization library, etc. Furthermore, BLAS (and LAPACK) commonly uses a block-based variant of each algorithm to attain a high performance strategy by blocks, adding the dimension of the blocks as another parameter. In Section 4 we develop a framework for BLAS which permits the implementation and use of such variants (as well as many others) of the BLAS operations in a flexible way. Based on this framework we implement two variants of Cholesky factorization, a blocked variant and an unblocked one, called $chol\_blk$ and $chol\_unb$, respectively.

To have an advance of how the framework works, let us analyze the type of the block-based version of the factorization:

$$
\begin{aligned}
chol\_blk\ ::\ & (Elt\ e \\
& , MatrixVector\ m\ v\ e \\
& , DOT\ \ dots\ \ v\ e \\
& , GEMV\ \ gemvs\ \ v\ m\ e \\
& , SCAL\ \ scals\ \ v\ e \\
& , SYRK\ \ syrks\ \ v\ m\ e \\
& , GEMM\ \ gemms\ v\ m\ e \\
& , TRSM\ \ trsms\ \ v\ m\ e) \\
& \Rightarrow Int \\
& \rightarrow (StratCxt\ dots, StratCxt\ gemvs, StratCxt\ scals \\
& \quad , StratCxt\ syrks, StratCxt\ gemms \\
& \quad , StratCxt\ trsms) \\
& \rightarrow TriangType\ (m\ e) \\
& \rightarrow ResM\ (dots, gemvs, scals, syrks, gemms, trsms) \\
& \qquad v\ m\ e
\end{aligned}
$$

The type constructors m and v correspond to the used $matrix$ and $vector$ representations, respectively. The type e is the type of the elements the input matrix stores. The constraint $MatrixVector\ m\ v\ e$ assures that some operations to deal with matrices and vectors are provided.

The constraints DOT, $GEMV$, $SCAL$, $SYRK$, $GEMM$ and $TRSM$ impose the existence of instances of classes representing the respective BLAS operations. Each of those instances implements an evaluation *strategy* for the respective operation. The first parameter of those instances is an empty data type that specifies the chosen strategy. For example, the type $DefSeq$ corresponds to sequential algorithms. Thus, an instance of the form $GEMM\ \ DefSeq\ \ m\ \ v\ \ e$ (i.e. with $gemms$ being $DefSeq$) implements a sequential version of GEMM.

Some strategies may use sub-strategies to solve certain parts of the algorithm. For instance, the most important properties of block-based algorithms is that they simply divide matrices into blocks and then use another algorithm to solve the reduced problem. That is the case of the block-based matrix product, which needs another product of matrices. The same happens with block-based triangular systems which require a matrix product and a resolution of triangular systems. This means that block-based GEMM depends on another version of GEMM, whereas block-based TRSM depends on versions of GEMM and TRSM. This dependency can be easily represented by using parametrized types. In the case of GEMM, for example, the strategy type $GemmParBlk\ \ gs$ is parametrized by the strategy gs of the inner GEMM. This way not only ease at strategies combination is achieved, but generality is also obtained, as sub-strategies are never explicitly mentioned and the responsibility of verifying the consistency of the combinations is delegated to Haskell's type system.

The first parameter of $chol\_blk$ is an integer determining the size of the blocks. The second parameter is a tuple containing the *context information* needed by each strategy of the used BLAS operations. The types of such contexts depend on each strategy. For example, the simple sequential algorithms usually do not need any extra information, while the block-based algorithms usually have to receive the size of the blocks and the context of the sub-strategies. We specify the context information required by each strategy by means of a type family [17] $StratCxt$, which has the strategy as type argument. The definition of this type family is given in Section 4. Then, for example, $StratCxt\ \ DefSeq$ is $NullCxt$, representing the empty context, and $StratCxt\ (GemmParBlk\ gs)$, for a strategy gs, is defined to be a pair of type $(Int, StratCxt\ \ gs)$ containing the dimension of the blocks and the context required by the sub-strategy gs. The third parameter of $chol\_blk$ is the type $(m\ e)$ of the matrix to factorize, wrapped by a constructor that indicates if the matrix is upper or lower. The matrix returned as result is wrapped in the type $ResM$ which is parametrized by the types of the used strategies and structures.

Then, for example, given some data types $MyMatrix$, $MyVector$, and their corresponding instance of $MatrixVector$, the following is a possible application of $chol\_blk$ that uses sequential versions of all its BLAS ingredients and block size of 256.

$$
\begin{aligned}
chol\_seq\ ::\ & TriangType\ (MyMatrix\ Double) \\
& \rightarrow ResM\ (DefSeq, DefSeq, DefSeq \\
& \qquad , DefSeq, DefSeq, DefSeq) \\
& \qquad MyVector\ MyMatrix\ Double \\
chol\_seq\ m = & chol\_blk\ 256\ (NullCxt, NullCxt, NullCxt \\
& \qquad , NullCxt, NullCxt, NullCxt)\ m
\end{aligned}
$$

By changing the types specifying the strategies and the contexts that are passed as argument, we obtain a parallel version:

$$
\begin{aligned}
chol\_par\ ::\ & TriangType\ (MyMatrix\ Double) \\
& \rightarrow ResM\ (DefSeq, DefSeq, DefSeq \\
& \qquad , SyrkParBlk\ DefSeq \\
& \qquad , GemmParBlk\ DefSeq
\end{aligned}
$$

$$
\begin{aligned}
&, TrsmParBlk\ DefSeq\ DefSeq) \\
&MyVector\ MyMatrix\ Double \\
chol\_par\ m = chol\_blk\ 256\ &(NullCxt, NullCxt, NullCxt \\
&, (128, NullCxt) \\
&, (128, NullCxt) \\
&, (128, NullCxt, NullCxt))\ m
\end{aligned}
$$

This version uses blocked-based parallel versions of SYRK and GEMM, both with block size of 128 and a sequential sub-strategy, and a blocked-based parallel version of TRSM with block size of 128 and two sub-strategies, one for internal matrix multiplication and another for TRSM resolving a minimal sub-problem.

3. Parallelism in Haskell

As technology advances, the intention to solve growing problems or to obtain more accurate results has led to a cycle in which the growth of computing need has an exponential behavior. While computer designers seek to create ever more powerful hardware, they are limited by physical and economic problems. One option to tackle these limits is working with multiple computing units to solve a problem in a coordinated manner, using techniques of High Performance Computing (HPC), which range from computer architectures up to program designs. However, an important drawback of HPC techniques is the increase in the complexity of algorithm design and development, because not only the requirements of serial algorithms must be met, but also new requirements are introduced, e.g. efficient synchronization of tasks. Due to that, in recent years several lines of work, such as declarative languages and automatic code analysis, were developed in order to simplify the use of HPC techniques. One promising line of research to simplify the programming on parallel hardware consists of using purely functional languages. Since a purely functional program has no side effects, each sub-expression has the potential to be evaluated in parallel, which seems to generate a very convenient environment for parallel programming.

There are several libraries that allow the introduction of parallelism in Haskell. Some of them are designed to parallelize deterministic computations, which means that the result should only depend on its parameters. This restriction makes it possible to deal with classical problems associated with parallel programming, such as deadlock or race conditions, in a transparent manner for the programmer. Libraries of this kind are Repa [16], DPH[5] [9, 21], Accelerate [8], Monad-Par [20] and Sparks [18, 19]. Following we highlight the main features of each of these libraries.

Sparks provides a model of pure, deterministic parallelism. The type of parallelism of this library is semi-explicit because the programmer must indicate possible points of parallelism and the sequence of operations, but can not directly control it. Its implementation is based on combinators which are used as annotations for the Haskell runtime system to inform possible places where to generate parallelism.

In the case of Monad-Par, the parallelism is expressed explicitly through a monadic interface. It provides a simple interface which prevents the programmer from combining parallelism with laziness, so sharing and granularity are completely under the control of the programmer. Its implementation is based on a scheduler that divides the work as evenly as possible between the available processors at runtime, by default it uses a simple work-stealing scheduler, but there are others available [12].

The libraries Repa and DPH provide data parallelism. The first one focuses on regular data parallelism while the latter focuses on irregular data parallelism. Repa's interface provides several representations of multidimensional arrays and shape-polymorphic operations to handle them; parallelism is automatically generated by handling the structure and alternating from representations with its provided operations, but with the caution (or inability) of nesting parallel computations. DPH, on the other hand, provides one-dimensional arrays and parallel operations that allows the programmer to nest parallel computation, it is more useful for irregular parallel computations.

Accelerate is a library that generates code to compute on GPUs through CUDA (Compute Unified Device Architecture). It provides data parallelism like Repa with a similar interface and the same programming constraint of nesting parallel computations.

[5] Data Parallel Haskell

$$
\begin{aligned}
&\textbf{class}\ Vector\ v\ e\ \textbf{where} \\
&\quad generate_v :: Int \to (Int \to e) \to v\ e \quad \text{-- Required} \\
&\quad fromList_v :: [e] \to v\ e \\
&\quad concat_v :: [v\ e] \to v\ e \\
&\quad (!_v) :: v\ e \to Int \to e \quad \text{-- Required} \\
&\quad length_v :: v\ e \to Int \quad \text{-- Required} \\
&\quad foldr_v :: (e \to a \to a) \to a \to v\ e \to a \\
&\quad map_v :: (e \to e) \to v\ e \to v\ e \\
&\quad zipWith_v :: (e \to e \to e) \to v\ e \to v\ e \to v\ e
\end{aligned}
$$

Figure 1. Class for vectors.

4. A framework for BLAS

In this section we present the building blocks of our functional framework for the BLAS library. We restrict ourselves to analyze the BLAS operations used in the implementation of one variant of the Cholesky factorization.

BLAS (Basic Linear Algebra Subprograms) [7] is a library that provides basic linear algebra operations such as matrix multiplication and triangular system resolution. It is divided into three levels: BLAS-1 which provides vector-vector operations, BLAS-2 with vector-matrix operations and BLAS-3 with matrix-matrix operations.

4.1 Vectors and Matrices

It is usual that BLAS libraries provide vector and matrix operations on a prefixed representation for these structures, allowing one simply to choose the type of the values to be stored in their cells. Our approach, in contrast, is to decouple the representation of vector and matrices from the implementations of the library operations. We introduce type classes $Vector$, $Matrix$ and $MatrixVector$ which define the necessary operations that concrete implementations of vectors and matrices should provide. Their declarations shown herein include only the operations necessary for implementing the BLAS operations presented in the paper. The actual declarations of these classes include more methods.

Default definitions are provided for all operations except for those with a "Required" comment. Therefore, to create an instance a few operations are in principle necessary to be defined. If desired, default definitions may be overridden with instance-specific definitions to benefit from the particularities of the concrete structure used. For space restrictions we do not show the default definitions but are in general straightly defined in terms of the required operations.

The class $Vector$ (see Figure 1) takes as parameters a vector constructor v and a type e of vector elements. It contains three required operations: $generate_v\ n\ f$ constructs a vector of dimension n with the values $f\ 1, \ldots, f\ n$; $v\ !_v\ i$ returns the i-th element of vector v; and $length_v\ v$ returns the dimension of v.

```
class Matrix m e where
  generate_m    :: Int → Int → (Int → Int → e)
                   → m e                          -- Required
  fromList_m    :: Int → Int → [e] → m e
  transpose_m   :: m e → m e
  (!_m)         :: m e → (Int, Int) → e           -- Required
  dim_m         :: m e → (Int, Int)               -- Required
  subMatrix_m   :: Int → Int → Int → Int → m e → m e
  toBlocks_m    :: Int → Int → m e → [[m e]]
  fromBlocks_m  :: [[m e]] → m e
  map_m         :: (e → e) → m e → m e
  zipWith_m     :: (e → e → e) → m e → m e → m e
```

Figure 2. Class for matrices.

Matrices are represented by a class $Matrix$ (see Figure 2) with parameters a matrix constructor m and a type e of matrix elements. Like $Vector$, this class contains three required operations: $generate_m\ m\ n\ f$ constructs a matrix of dimension $m \times n$ with value $f\ i\ j$ at each position (i, j); $m\ !_m\ (i, j)$ returns the value contained in m at position (i, j); and $dim_m\ m$ returns the dimension of m. Function $subMatrix_m\ i\ j\ rs\ cs$ returns the submatrix of dimension $rs \times cs$ starting at position (i, j) of a given matrix. Functions $toBlocks_m$ and $fromBlocks_m$ allow to split/construct a matrix in/from a sequence of blocks. An application $toBlocks_m\ r\ c\ m$ splits the matrix m into blocks of size $r \times c$ (the blocks at the borders of the matrix can have a lower dimension), which are returned in a list of lists of matrices such that the inner lists represent rows of blocks. The function $fromBlocks_m$ is the inverse of $toBlocks_m$. Functions map_m and $zipWith_m$ are conceptually similar to their relatives for lists.

There is a third class $MatrixVector$ (see Figure 3) that contains operations relating matrix and vector representations. This means that it is possible to define implementations for matrices and vectors independently and then connect them through this class.

```
class (Vector v e, Matrix m e) ⇒
  MatrixVector m v e where
    -- Columns
  col_mv        :: Int → m e → v e
  fromCols_mv   :: [v e] → m e
  toCols_mv     :: m e → [v e]
    -- Rows
  row_mv        :: Int → m e → v e
  fromRows_mv   :: [v e] → m e
  toRows_mv     :: m e → [v e]
```

Figure 3. Class that relates matrices and vectors

The three classes include the type of the elements e as one of their parameters, in order to allow the definition of instances that need to add constraints to the type e and because BLAS operations need to restrict the type of the elements contained in vectors and matrices. The occurrence of e as a parameter of these classes opens also the possibility to define instances for specific types of elements, for example, to obtain more efficient implementations of the class.

```
class (Eq e, Fractional e) ⇒ Elt e where
  getConjugate :: e → e
  getConjugate = id

instance Elt Double
instance Elt Float
instance RealFloat e ⇒ Elt (Complex e) where
  getConjugate = conjugate
```

Figure 4. BLAS elements.

4.2 Defining BLAS operations

Our intention behind the development of the functional BLAS routines is to capture the essence of this library. This means, for example, that operation typing is adapted in order to better fit in a functional language. Therefore, signatures of the BLAS operations implemented in the framework do not necessarily conform exactly to those of their original counterparts.

Since BLAS is a library for linear algebra, its operations only manipulate real or complex numbers. To deal with this restriction on elements, we introduce a class Elt (Figure 4) that needs to be included in the context of every BLAS operation declaration. The class Elt contains a single method $getConjugate$ that is the identity function except for complex numbers in which case it returns the conjugate of a number. This makes it possible to write instances of the BLAS operations that can be equally used on real and complex numbers.

Our framework admits the definition of multiple implementations for BLAS operations, based on different algorithms, parallel strategies, or structure representations. This is achieved by wrapping each BLAS operation in a class with a single method. Different implementations of an operation can then be defined by introducing alternative instances of the associated class.

The functional BLAS routines are described below.

Dot Product (DOT)

```
class (Elt e, Vector v e) ⇒ DOT s v e where
  dot :: StratCxt s → v e → v e → ResS s e
```

Figure 5. DOT declaration

The dot (or scalar) product, is a BLAS-1 operation, that takes two equal-length vectors and returns a scalar.

$$a * b = \sum_{i=1}^{n} a_i \cdot b_i \tag{1}$$

Associated to this operation we define a class DOT with a single method dot. The definition of this class is depicted in Figure 5. The different implementations of the DOT operation will appear as instances of this class.

A strategy is a possible combination of algorithms that implements an operation. In our framework, strategies are defined at the type level and are one of the type parameters of BLAS operations. Different implementations of an operation can then be obtained by specifying different strategies. A strategy (name) is specified by introducing an empty data type (a data type without constructors). Associated with each strategy we have to declare the amount of context information the strategy requires. The type of that information is declared by means of the type family [17] $StratCxt$:

$\mathbf{type}\ family\ StratCxt\ s :: *$

For instance, a possible strategy is $DefSeq$ which corresponds to sequential algorithms. This strategy is not particularly interesting since it does not require any context information.

$\mathbf{data}\ DefSeq$

$\mathbf{data}\ NullCxt = NullCxt$

$\mathbf{type\ instance}\ StratCxt\ DefSeq = NullCxt$

We enforce the appearance of the strategy s in the result type by declaring a parameterised data type $ResS$ that takes this type as parameter. This is required by the compiler to correctly determine the instance in use. Doing so we avoid passing strategies as parameters to BLAS operations. The strategy corresponding to an operation invocation is then deduced at compile time from type information, which is used to determine the specific instance corresponding to that invocation and the associated type of the strategy context.

$\mathbf{data}\ ResS\ s\ e = ResS\ \{unResS :: e\}$

Notice that s is not used on the right hand side of the definition of $ResS$. Such a type is called a *phantom type* [15].

A call to dot that fixes the $DefSeq$ strategy is given by:

$$dot\_seq :: DOT\ DefSeq\ v\ e \Rightarrow v\ e \to v\ e \to ResS\ DefSeq\ e$$
$$dot\_seq = dot\ NullCxt$$

Vector Scale (SCAL)

$\mathbf{class}\ (Elt\ e, Vector\ v\ e) \Rightarrow SCAL\ s\ v\ e\ \mathbf{where}$
$\quad scal :: StratCxt\ s \to v\ e \to e \to ResV\ s\ v\ e$

Figure 6. SCAL declaration

The BLAS-1 operation SCAL, scales a vector by a scalar. Figure 6 shows the definition of SCAL in our framework. In this case we use a type $ResV$ to wrap the result:

$\mathbf{data}\ ResV\ s\ v\ e = ResV\ \{unResV :: v\ e\}$

General Matrix Multiplication (GEMM)

Given matrices A, B and C, and scalar coefficients α and β, the GEMM operation computes the matrix that results from the expression:

$$\alpha\ \cdot\ A^{tA} * B^{tB} + \beta\ \cdot\ C, \tag{2}$$

where, for a matrix X, X^{tX} denotes one of the following matrices: simply X, the transpose X^T, or the hermitian (or conjugate transpose) $X^H = conjugate\ (X^T)$.

Figure 7 shows the definition of GEMM in our framework. $TransType$ is used as a wrapper type for matrices that contains the information about which matrix has to be considered by the operation: N (normal), T (transpose) or H (hermitian). So, a call to this method corresponds to $gemm\ (cxt, A^{tA}, B^{tB}, \alpha, \beta, C)$.

The following functions on a wrapped matrix compute their result on the matrix directly without applying any transposition.

$$dim_t :: Matrix\ m\ e \Rightarrow TransType\ (m\ e) \to (Int, Int)$$
$$dim_t\ (N\ m) = dim_m\ m$$
$$dim_t\ (T\ m) = swap\ \$\ dim_m\ m$$
$$dim_t\ (H\ m) = swap\ \$\ dim_m\ m$$

$$(!_t)\ ::\ (Elt\ e, Matrix\ m\ e)$$
$$\quad \Rightarrow TransType\ (m\ e) \to (Int, Int) \to e$$

$\mathbf{data}\ TransType\ a = N\ a \mid T\ a \mid H\ a$

$\mathbf{class}\ (Elt\ e, MatrixVector\ m\ v\ e) \Rightarrow$
$\quad GEMM\ s\ m\ v\ e\ \mathbf{where}$
$\quad gemm\ ::\ StratCxt\ s$
$\quad\quad \to TransType\ (m\ e) \to TransType\ (m\ e)$
$\quad\quad \to e \to e \to m\ e \to ResM\ s\ v\ m\ e$

Figure 7. GEMM declaration

$$(N\ m)\ !_t\ (i,j) = m\ !_m\ (i,j)$$
$$(T\ m)\ !_t\ (i,j) = m\ !_m\ (j,i)$$
$$(H\ m)\ !_t\ (i,j) = getConjugate\ \$\ m\ !_m\ (j,i)$$

The result type of a BLAS-3 function has to include not only the types of the strategy s, the matrix m and the elements e, but also the type of the vectors v we use to decompose (and compose) the matrix.

$\mathbf{data}\ ResM\ s\ v\ m\ e = ResM\ \{unResM :: m\ e\}$

We have analyzed three algorithms for computing GEMM: sequential, block-based and Strassen algorithm. Below we present the sequential one.

The sequential algorithm follows directly from GEMM expression (2). Its Haskell implementation is shown in Figure 8. It uses the $DefSeq$ strategy which, as we saw above, manipulates an empty context. The returned matrix is constructed with method $generate_m$ from the class Matrix which, in its function argument, specifies how to compute each element (i,j) using GEMM formula. Computation of element (i,j) of the product between matrices mA and mB is performed by a separate function $matMultIJ$, which first builds an intermediate vector with the pairwise products of the corresponding positions, and then computes the sum of the elements of that vector. Notice that the positions of the matrices mA and mB are accessed using function $(!_t)$, meaning that we do not really compute the transpose, but directly access the corresponding positions in the "virtual" transpose. The type annotation $v\ \ e$ is necessary because the type of the intermediate vector does not occur in the type of $matMultIJ$; for this to work the $ScopedTypeVariables$ flag of GHC (Glasgow Haskell Compiler) has to be enabled.

In our implementation of BLAS operations, we are assuming as pre-condition that the input matrices conform to the matrix operations involved, that is, their respective dimensions permit those matrix operations. Having this pre-condition we do not need to do any conformance testing inside the operation, as can be observed in the sequential implementation of $gemm$.

Figure 9 shows the code of a parallel instance of $GEMM$ which is based on its sequential definition. A data type $DefPar$ is created to represent the strategy. Like $DefSeq$, this new strategy does not require any context information and therefore the same context $NullCtx$ is associated to it. In this case Sparks was used to provide parallelism. Comparing the codes of the sequential and the parallel versions, it can be observed that very few changes were required to introduce parallelism. The only difference is that, instead of generating the matrix directly using $generate_m$, the parallel version calls function $generatePar$ to build each column of the matrix in parallel. Such parallel computation is performed using the $parMap$ funtion, which applies a function to each element of a list in parallel. The parameter $rdeepseq$, indicates that each application has to be fully evaluated.

```
instance (Elt e, MatrixVector m v e) ⇒ GEMM DefSeq m v e where
  gemm _ mA mB α β mC
    = ResM $ generate_m m n (λi j → α * matMultIJ i j + β * (mC !_m (i,j)))
    where
      (m, p) = dim_t mA
      (_, n) = dim_t mB
      matMultIJ i j = foldr_v (+) 0 (generate_v p (λk → (mA !_t (i,k)) * (mB !_t (k,j)) :: v e)
```

Figure 8. Sequential definition of GEMM.

```
instance (NFData (v e), Elt e, MatrixVector m v e) ⇒ GEMM GemmPar m v e where
  gemm _ mA mB α β mC
    = ResM $ generatePar m n (λi j → α * matMultIJ i j + β * (mC !_m (i,j)))
    where
      (m, p) = dim_t mA
      (_, n) = dim_t mB
      matMultIJ i j
        = foldr_v (+) 0 (generate_v p (λk → (mA !_t (i,k)) * (mB !_t (k,j)) :: v e)
      generatePar m n gen =
        fromCols_mv ∘ parMap rdeepseq (λj → generate_v m (λi → gen i j) :: v e) $ [0..(n − 1)]
```

Figure 9. Parallel definition of GEMM.

Matrix-Vector Multiplication (GEMV)

```
class GEMV s m v e where
  gemv :: StratCxt s
       → TransType (m e) → v e
       → e → e → v e → ResV s v e
```

Figure 10. GEMV declaration

The BLAS-2 operation GEMV can be seen as a special case of the BLAS-3 GEMM, where the second operand is a vector instead of a matrix.

$$\alpha \cdot A^{tA} * b + \beta \cdot c, \tag{3}$$

Figure 10 shows the definition of GEMV in our framework.

Rank-k Update of Symmetric Matrix (SYRK)

Another special case of GEMM, is the BLAS-3 operation SYRK, which instead of multiplying two different matrices, it multiplies a matrix with its transpose:

$$\alpha \cdot A^{tA} * A^{(\neg\, tA)} + \beta \cdot C \tag{4}$$

The matrix C is assumed to be symmetric. Figure 11 shows the definition of SYRK in our framework.

TriangType is a wrapper type that declares which triangle (*Lower* or *Upper* one) of the argument matrix C has to be considered in the computation.

Triangular System Resolution (TRSM)

A linear system is a set of linear equations involving the same set of variables. A linear system can be represented as a matrix equation $A * x = b$ where A is a $m \times m$-matrix of coefficients, x is a m-vector of variables, and b is a m-vector of independent terms. If the system is nonsingular, in which case A is invertible, then x can be expressed as $x = A^{-1} * b$. This can be easily extended to support general matrices of variables X and independent terms B, solving multiple systems $A * X = B$ simultaneously by solving a system for each column of X and B. Again, if A is invertible, then we can rewrite the matrix equation as $X = A^{-1} * B$.

```
data TriangType a = L a | U a

class (Elt e, MatrixVector m v e) ⇒
      SYRK s m v e where
  syrk :: StratCxt s
       → TransType (m e)
       → e → e → TriangType (m e) → Res s v m e
```

Figure 11. SYRK declaration

A triangular linear system is the case when A is a $m \times m$ lower or upper triangular matrix (that is, a square matrix in which the elements of the upper/lower triangular matrix are all zero). The BLAS-3 TRSM operation solves a generalized form of the extended equation for triangular systems:

$$X = \alpha \cdot (A^{-1})^{tA} * B \tag{5}$$

This operation is defined in our framework by declaring the class *TRSM* shown in Figure 12. *TriangType* declares which triangle of the argument matrix has to be considered. *TransType* means the same as in GEMM. *UnitType* specifies if, in the operation computation, the argument matrix has to be considered as being a *unit* triangular matrix (a matrix with ones on its diagonal). Then, for a triangular matrix m, *Unit* m means to consider m with its diagonal substituted by ones, whereas *NoUnit* m means simply m.

```
data UnitType a = Unit a | NoUnit a

class (Elt e, MatrixVector m v e) ⇒
      TRSM s m v e where
  trsm :: StratCxt s → e
       → TransType (TriangType (UnitType (m e)))
       → m e → ResM s v m e
```

Figure 12. TRSM declaration.

A possible algorithm to solve a triangular system is to compute the inverse of A and then multiply it by b. However, this is an expensive alternative. A better solution is to apply forward or backward substitution, iterative methods to solve lower or upper triangular systems, respectively. For example, the forward version implies the following equations:

$$x_1 = \frac{b_1}{\alpha_{1,1}} \quad (6)$$

$$x_i = \frac{(b_i - \sum_{j=1}^{i-1} \alpha_{i,j} \cdot x_j)}{\alpha_{i,i}} \quad (7)$$

Where the case x_i applies for $i = 2 \ldots n$. Another option to compute this recursive step is to substitute Equation 7 with the following one:

$$b_k = (b_k - \alpha_{k,i-1} \cdot x_{i-1}), k = i \ldots n$$

$$x_i = \frac{b_i}{\alpha_{i,i}} \quad (8)$$

As in the case of GEMM, there exists a blocked version of TRSM. The system is divided into blocks and solved following the same procedure as in the unblocked version, considering these blocks as computing elements (instead of scalars in the unblocked version). Thus, addition and multiplication are now matrix operations (GEMM) and the scalar division is replaced by a triangular system resolution (TRSM).

$$X_1 = Trsm(A_{1,1}, B_1) \quad (9)$$

$$X_i = Trsm(A_{i,i}, (B_i - [A_{i,1}, \ldots, A_{i,i-1}] * [X_1, \ldots, X_{i-1}]^t)) \quad (10)$$

$$X_i = Trsm(A_{i,i}, ([B_i, \ldots, B_n] - [A_{i,i-1}, \ldots, A_{n,i-1}] * X_{i-1})) \quad (11)$$

Equations 9, 10 and 11 are, respectively, the analog blocked versions to equations 6, 7 and 8.

In Figure 13 we show a fragment of the definition of the instance of TRSM by blocks. The data type $TrsmParBlk$, representing the strategy, has kind $* \to * \to *$, which means that two type parameters are needed. The first parameter is the strategy for GEMM and the second is a strategy for the inner TRSM. The context information for this strategy includes an integer, indicating the size of the blocks, and the contexts for the sub-strategies gs and ts. When declaring the instance for the block-based version of TRSM, it is required that the corresponding instances $GEMM\ gs\ m\ v\ e$ and $TRSM\ ts\ m\ v\ e$ exist.

5. Cholesky factorization

Figures 14 and 15 present the unblocked and blocked versions of Cholesky factorization, respectively, following the FLAME[6] notation [13, 23]. These versions are taken from [4, 5].

[6] Formal Linear Algebra Methodology Environment

```
data TrsmParBlk gs ts

type instance StratCxt (TrsmParBlk gs ts) =
    (Int, StratCxt gs, StratCxt ts)

instance (GEMM gs m v e, TRSM ts m v e)
         ⇒ TRSM (TrsmParBlk gs ts) m v e where
  trsm (ctx, gctx, tctx) trt tt dt α mA mB = ...
    where
      call_trsm trt tt dt α mA mB = unResM $
          trsm tctx trt tt dt α mA mB :: ResM ts v m e
      call_gemm tt1 tt2 α mA mB β mC = unResM $
          gemm gctx tt1 tt2 α mA mB β mC
            :: ResM gs v m e
```

Figure 13. Architecture of TRSM instance by blocks

Algorithm: $A := Chol\_unb(A)$

Partition $A \to \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0 and A_{BR} is $n \times n$

while $m(A_{TL}) < m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \to \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline \alpha_{10} & \alpha_{11} & \star \\ \hline A_{20} & \alpha_{21}^T & A_{22} \end{array}\right)$$

where α_{11} is a scalar

α_{11}	:=	$\alpha_{11} - \alpha_{10}\alpha_{10}^T$	(`dot`)
α_{11}	:=	$\sqrt{\alpha_{11}}$	
α_{21}	:=	$\alpha_{21} - A_{20}\alpha_{10}^T$	(`gemv`)
α_{21}	:=	α_{21}/α_{11}	(`scal`)

Continue with

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline \alpha_{10} & \alpha_{11} & \star \\ \hline A_{20} & \alpha_{21}^T & A_{22} \end{array}\right)$$

endwhile

Figure 14. Unblocked variant of Cholesky factorization.

It should be noted that in both algorithms only the lower part of the matrix is computed. Consequently, the elements above the main diagonal are neither computed nor referenced. The blocks in the upper triangular part of the matrix are then denoted by "$\star$".

In the unblocked variant of Cholesky factorization (Figure 14), at a given iteration, the element placed on the main diagonal (the pivot element α_{11}) is computed by a dot product of vectors and a scalar square root. Notice that, since the A matrix is symmetric, the column-vector α_{10}^T is the same as α_{01}. Afterwards, the rest of the column (α_{21}) is updated by a matrix vector multiplication (`gemv`) and a scalar vector multiplication (`scal`).

The blocked counterpart (Figure 15) proceeds in a similar way, but working with a $b \times b$ matrix (A_{11}) instead of a scalar value (α_{11}). To leverage the symmetry of the factorized matrix only the triangular inferior matrix of the A_{11} block is computed. Furthermore, in this version all the operations are matrix computations, which makes it more suitable for parallel computation.

Algorithm: $A := Chol\_blk(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$

where A_{TL} is 0×0 and A_{BR} is $n \times n$

while $m(A_{TL}) < m(A)$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$

$$\begin{array}{lcll} A_{11} & := & A_{11} - A_{10}A_{10}^T & \texttt{(syrk)} \\ A_{11} & := & \text{Chol\_unb}(A_{11}) & \\ A_{21} & := & A_{21} - A_{20}A_{10}^T & \texttt{(gemm)} \\ A_{21} & := & A_{21}A_{11}^{-1} & \texttt{(trsm)} \end{array}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

endwhile

Figure 15. Blocked variant of Cholesky factorization.

5.1 Implementation

Figures 16 and 17 show our implementation of the block-based version of Cholesky factorization. In this case, we assume the input matrix is lower triangular. In Figure 16 the function $chol\_blk$ simply delegates the job of calculating the factorization of a lower triangular matrix to $chol\_blk\_l$, which receives two extra parameters, a block counter (that indicates the pivot on which the algorithm is standing) and an accumulator containing the value of the result matrix computed so far.

The implementation of $chol\_blk\_l$ shown in Figure 17 follows the description of Figure 15. When the block counter reaches the number of blocks on the matrix, the function returns the value of the accumulator (mAL) as result. Otherwise, it computes a new approximation of the result matrix (mAL') and makes a recursive call. Values $mA10$, $mA11$, $mA20$ and $mA21$ are as depicted of Figure 15. It is important to note that $mA11$ and $mA21$ are sub-matrices of mA, while $mA10$ and $mA20$ are sub-matrices of mAL. Values $mA11'$, $mA11''$, $mA21'$, and $mA21''$ correspond to each step shown in Figure 15. To complete the generation of mAL' we append the columns of $mA21''$ (the final step of Figure 15) to mAL.

6. Experimental evaluation

In this section we evaluate the performance of several parallel instances of the Cholesky factorization implemented in terms of our framework. The experimental evaluation was performed in a computer with AMD FX 8120 3.1 GHz 8 cores processor, 8 GB RAM, EVGA GeForce GTX 570 HD 2.5GB graphics card, and OS Kubuntu 13.04 i686. All the experiments were performed using GHC-7.6.2.

6.1 GEMM

We first evaluate several options (data types, strategies) to perform the basic operations on Cholesky factorization. Table 1 summarizes the execution times (in seconds) for GEMM applied to 512×512 and 1024×1024 matrices. We defined instances of $Vector$, $Matrix$ and $MatrixVector$ for different structures: $HMatrix$ (based on $Data.Packed.Matrix$[7]), $VectorMatrix$ (vector of column vectors using $Data.Vector$[8]), $LLMatrixByRows$ (list of lists, inner lists represent rows), $LLMatrixByCols$ (list of lists, inner lists represent columns), $RepaMatrix$ (Repa bi-dimensional array in its delayed representation) and $Accelerate$.

```
chol_blk :: (Elt e, MatrixVector m v e
           , DOT   dots  v e
           , GEMV  gemvs v m e
           , SCAL  scals v e
           , SYRK  syrks v m e
           , GEMM  gemms v m e
           , TRSM  trsms v m e)
         ⇒ Int
         → (StratCxt dots, StratCxt gemvs, StratCxt scals
           , StratCxt syrks, StratCxt gemms
           , StratCxt trsms)
         → TriangType (m e)
         → ResM (dots, gemvs, scals, syrks, gemms, trsms)
                v m e
chol_blk block ctx (Lower mA)
  = ResM $ chol_blk_l block ctx 0 mA (generate_m 0 0 ⊥)
```

Figure 16. Cholesky.

	512	1024
HMatrix-BindC	0.39	2.47
VectorMatrix-BindC	0.29	2.08
HMatrix-GemmPar	6.40	50.72
VectorMatrix-GemmPar	5.11	45.49
LLMatrixByCols-GemmPar	295.17	4867.92
LLMatrixByRows-GemmPar	230.01	4578.70
Repa-DefSeq	4.93	60.78
Accelerate-AccSeq	0.12	0.92

Table 1. GEMM execution times (in seconds).

For each structure we show the strategy we use to evaluate it: *BindC* just binds to a sequential C implementation of BLAS, *GemmPar* is the parallel Sparks implementation of Figure 9 (evaluated with 8 cores), *DefSeq* is the sequential implementation of Figure 8 and *AccSeq* is a sequential instance created to be able to use the *Accelerate* library. Both *Repa* and *Accelerate* use sequential algorithms because they implement data parallelism.

The results for GEMM operations allow identifying three different groups of structures according to their performance. The first group includes the non pure functional variants (*HMatrix-BindC*, *VectorMatrix-BindC* and *Accelerate*). These versions reach the best level of performance. In a second group, we can place the efficient pure functional versions, i.e. *HMatrix-GemmPar*, *VectorMatrix-GemmPar* and *Repa-DefSeq*, with a middle performance. At the end, the non efficient variants, based on lists of lists data types.

6.2 Cholesky

Let us now focus on Cholesky factorization. Specifically, we computed several combinations of data types, strategies and block dimensions to execute the factorization over our framework. In this

[7] `http://hackage.haskell.org/package/hmatrix`

[8] `http://hackage.haskell.org/package/vector`

```
chol_blk_l :: (Elt e, MatrixVector m v e
              , DOT  dots  v e,   GEMV  gemvs  v m e, SCAL  scals  v e
              , SYRK syrks v m e, GEMM gemms v m e, TRSM trsms v m e)
           ⇒ Int
           → (StratCxt dots, StratCxt gemvs, StratCxt scals, StratCxt syrks, StratCxt gemms, StratCxt trsms)
           → Int → m e → m e → ResM (dots, gemvs, scals, syrks, gemms, trsms) v m e
chol_blk_l block ctx@(dot_ctx, gemv_ctx, scal_ctx, syrk_ctx, gemm_ctx, trsm_ctx) k mA mAL
  | k ≡ cantBlocks = mAL
  | otherwise      = chol_blk_l block ctx (k + 1) mA mAL'
  where
    mA10   = subMatrix_m (block ∗ k) 0              block (block ∗ k) mAL
    mA11   = subMatrix_m (block ∗ k) (block ∗ k)    block block       mA
    mA11'  = call_syrk (−1) (N mA10) 1 (Lower mA11)
    mA11'' = call_chol_unb (Lower mA11')
    mA20   = subMatrix_m ((k + 1) ∗ block) 0              (mA_dim − (k + 1) ∗ block) (k ∗ block) mAL
    mA21   = subMatrix_m ((k + 1) ∗ block) (k ∗ block) (mA_dim − (k + 1) ∗ block) block       mA
    mA21'  = call_gemm (N mA20) (T mA10) (−1) 1 mA21
    mA21'' = transpose_m $ call_trsm 1 (N ∘ Lower $ NoUnit mA11'') (transpose_m mA21')
    mAx1   = add_zeros k block ∘ iif (k ≡ cantBlocks − 1) mA11'' $ concatByCol_m mA11'' mA21''
    mAL'   = fromCols_vm $ (toCols_vm mAL :: [v e]) ++ (toCols_vm mAx1 :: [v e])

    call_syrk m1 α β m2       = unResM $ syrk  syrk_ctx m1 α β m2       :: Res syrks  v m e
    call_gemm m1 m2 α β m3    = unResM $ gemm gemm_ctx m1 m2 α β m3 :: Res gemms v m e
    call_trsm α m1 m2         = unResM $ trsm  trsm_ctx α m1 m2         :: Res trsms  v m e
    call_chol_unb m           = unResM $ chol_unb (dot_ctx, gemv_ctx, scal_ctx) m :: Res (dots, gemvs, scals) v m e

    block      = getSqrBlockDim block_ctx
    cantBlocks = mA_dim `div` block
    mA_dim     = cantCols_m mA

add_zeros :: Int → Int → m e → m e
add_zeros k block v = concatByCol_m (generate_m (k ∗ block) (block) (λ_ _ → 0) :: m e) v
concatByCol_m m1 m2 = let rows_m1 = cantRows_m m1
                          cols_m1 = cantCols_m m1
                          rows_m2 = cantRows_m m2
                          cols_m2 = cantCols_m m2
                      in generate_m (rows_m1 + rows_m2) cols_m1
                                    (λi j → iif (i ⩾ rows_m1) (!_m (i − rows_m1) j m2) (!_m i j m1))
```

Figure 17. Cholesky.

section we present a representative subset of these experimental results. Since the list of list options implied really large execution times, we decided to exclude them in our experimental results for Cholesky. We will also exclude the *BindC* and *Accelerate* options, because we want to rely on pure functional implementations.

Firstly, we evaluate the Repa implementations. Table 2 presents the execution times (in seconds) to factorize a SPD matrix with 512 columns for the sequential unblocked and blocked variants of Cholesky factorization. Since Repa employs data parallelism it is reasonable to reach better results for algorithms with low data dependencies (even on sequential contexts), such as blocked methods.

In Figure 18, we describe the performance evolution in a SPD matrix with 512 columns when different number of cores (1 to 8) are employed. The speedup, or algorithmic speedup, is the ratio of the runtime for the best sequential version against the runtime of the given parallel version. In this case we measure

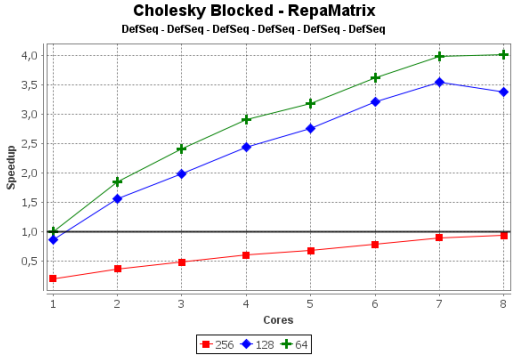

Figure 18. Speedup of Unblocked Cholesky using Repa

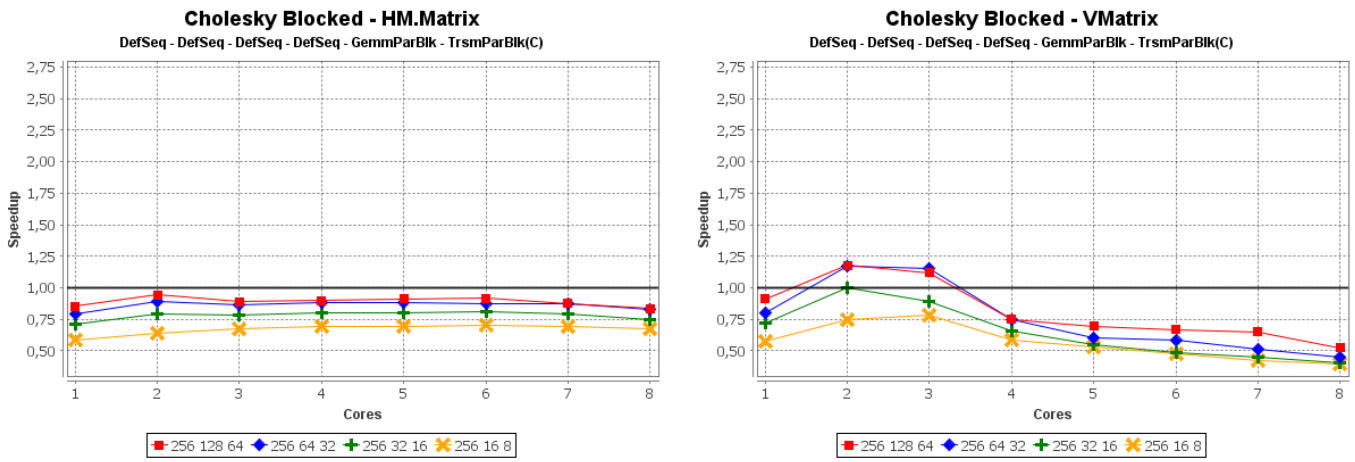

Figure 19. Speedup of Blocked Cholesky with sequential SYRK and blocked parallel GEMM and TRSM

	Chol_unb	Chol_blk		
		64	128	256
Repa	2953.56	72.68	84.75	378.45

Table 2. Execution time (in seconds) of unblocked and blocked variants of Cholesky factorization (matrix of 512 columns).

against a sequential blocked variant with blocks of size 64. Speedup values reached for this variant are good, close to linear speedups. However, it should be noted that in the best case its execution time with one core is approximately 73 seconds, which is slow considering the sequential unblocked version for *HMatrix* and *VectorMatrix* (see Table 3, for matrices of 1024 columns). The important data dependencies on Cholesky factorization is a possible explanation for this performance values. Particularly, blocked variants of Cholesky avoid part of these dependencies, although not completely.

	Chol_unb	Chol_blk		
		64	128	256
HMatrix	39.88	52.32	47.05	61.47
VectorMatrix	43.27	64.29	58.92	194.63

Table 3. Execution time (in seconds) of unblocked and blocked variants of Cholesky factorization (matrix of 1024 columns).

We will follow the study considering the *HMatrix* and *VectorMatrix* based versions. In this line, Table 3 presents the execution times (in seconds) to factorize a SPD matrix with 1024 columns for the sequential unblocked and blocked variants of Cholesky factorization. These experimental results enable us to deduce that in the case of sequential evaluation, using both *HMatrix* and *VectorMatrix*, the application of blocked approaches offers no benefit at all.

In the rest of the section we present algorithmic speedup results for different variants of the blocked Cholesky factorization, in a SPD matrix with 1024 columns, measured against the corresponding sequential unblocked variants shown on Table 3 (39.88 seconds for $HMatrix$ and 43.27 seconds for $VMatrix$). Specifically, with a block size fixed in 256 for the Cholesky factorization, we obtain the following results:

- In Figure 19 we show the speedup of different versions of the factorization with blocked parallel TRSM, sequential SYRK and blocked parallel GEMM. We vary the sizes of the GEMM blocks from 16 to 128 and the TRSM blocks from 8 to 64.
- In Figure 20 we show the speedup of different versions of the factorization with sequential TRSM, parallel SYRK and blocked parallel GEMM. We vary the sizes of the GEMM blocks from 16 to 128.
- In Figure 21 we show the speedup of different versions of the factorization with sequential GEMM, parallel SYRK and blocked parallel TRSM. We vary the sizes of the TRSM blocks from 16 to 128.
- In Figure 22 we show the speedup of different versions of the factorization with parallel SYRK and blocked parallel GEMM and TRSM. We vary the sizes of the GEMM blocks from 16 to 128 and the TRSM blocks from 8 to 64.

Considering Figure 19, it should be noted that all blocked variants (data types–block dimensions) are outperformed by their corresponding unblocked sequential versions; i.e. their speedups are below one. Taking into account that SYRK is one of the most costly operations in our blocked variant of Cholesky factorization, and related to the results summarized on Table 3, it was expected to achieve poor results in this experiment. On the other hand, the results for the parallel SYRK combinations (Figures 20, 21 and 22) show different levels of improvement.

One major observation from this analysis is that we can reach only modest values of speedup, up to 2.8 times for 8 cores. Additionally, the $VectorMatrix$ based versions attain, in general, better speedup values than their $HMatrix$ counterparts.

Another relevant remark is the importance that an adjusted selection for block dimensions plays on basic operations. Additionally, this decision is influenced by the operation and the data type employed. Combined with this observation we can therefore note that $HMatrix$ versions improve when the parallel versions of the basic kernels are included. However, $VectorMatrix$ shows a counterintuitive behavior, since the best version (the general best version also) uses a sequential GEMM.

As a summary of this experimental evaluation we can stress the importance of the flexibility of our framework, since the use of this tool allows us to leverage several features on different contexts such as concurrency processing, avoid data dependencies and improve data types manipulation.

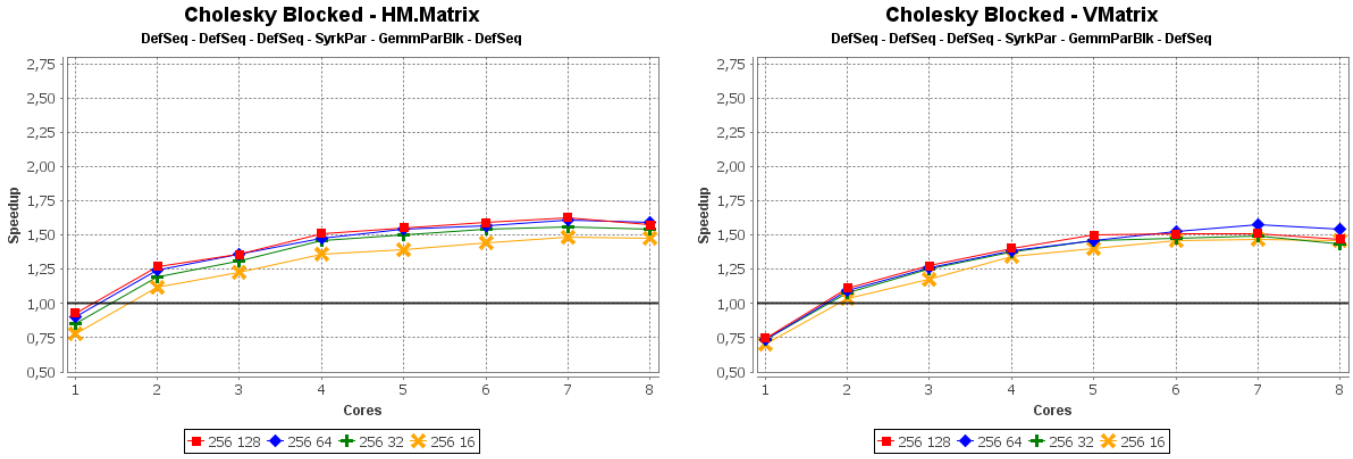

Figure 20. Speedup of Blocked Cholesky with sequential TRSM, parallel SYRK and blocked parallel GEMM

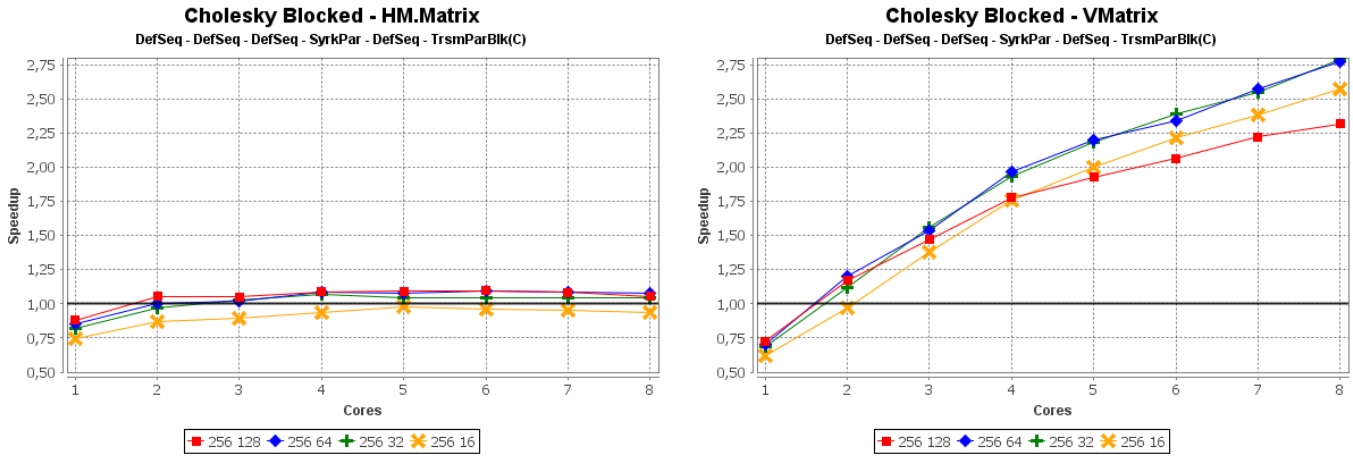

Figure 21. Speedup of Blocked Cholesky with sequential GEMM, parallel SYRK and blocked parallel TRSM

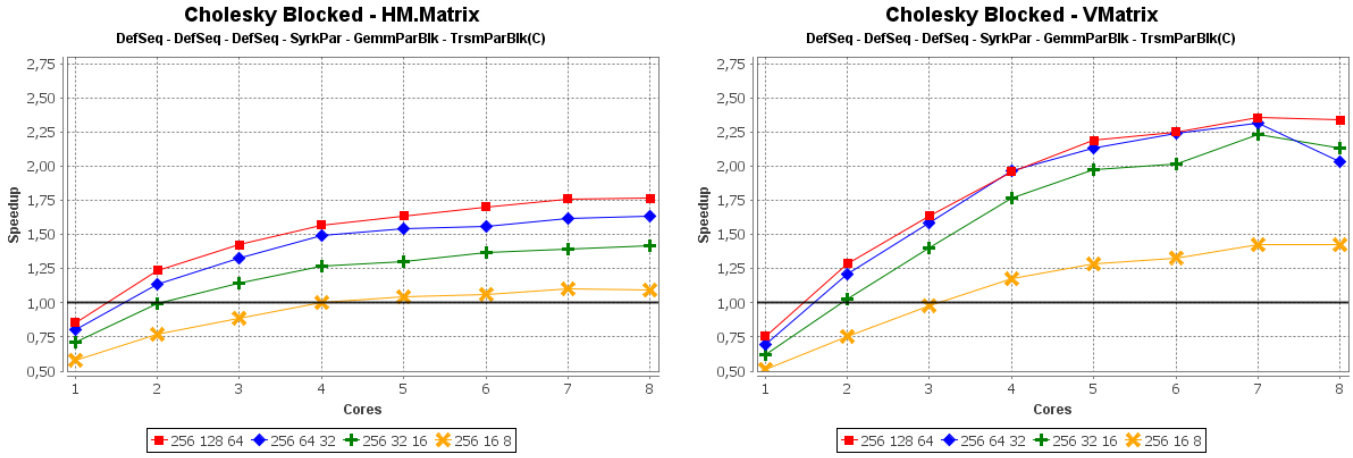

Figure 22. Speedup of Blocked Cholesky with parallel SYRK and blocked parallel GEMM and TRSM

7. Conclusions and Future Work

We presented the first layer of our framework to implement a functional run-time system for NLA field. Specifically, we described a flexible framework layer to define parallel implementations of BLAS operations in Haskell. Moreover, we showed the use of this tool to leverage the concurrence of the Cholesky factorization, thus testing its performance increase through multiple strategies in a painless manner. We use this really known operation of NLA as a proof of concept.

As main features, our framework permits:

- definition of multiple instances of BLAS operations, each one associated to a different sequential/parallel strategy;
- definition of multiple representations of vectors and matrices with a unified interface;
- easy-to-define combinations of strategies;
- arbitrary combination of strategies and structure representations in instance definitions;
- type-safe manipulation of context information associated to each strategy.

The framework also provides default definitions for most methods associated to vector and matrix representations, but it is open to define specific, possibly more efficient, implementations of them. It is also possible to define strategies specialized for a given representation.

Future research lines include experimentation with further combinations of strategies, and parallel implementations of our operations over other hardware platforms.

As there are inherent performance benefits of mutable data structures, the development of a monadic interface would not only allow us to handle other data structures but also to achieve a higher goal in performance, and even to handle in a different manner some structures we are already using.

Other line of work is the development of the second layer of our framework, composed by functional parallel LAPACK operations. In addition, the study of a dynamic optimization module in order to build a complete dense NLA run-time system.

References

[1] *SMP Superscalar (SMPSs) User's Manual, Version 2.4*. Barcelona Supercomputing Center, Barcelona, 2011.

[2] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series*, 180, 2009. .

[3] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide, Third Edition*. SIAM, Philadelphia, 1999.

[4] R. M. Badia, J. R. Herrero, J. Labarta, J. M. Pérez, E. S. Quintana-Ortí, and G. Quintana-Ortí. Parallelizing dense and banded linear algebra libraries using smpss. *Concurrency and Computation: Practice and Experience*, 21(18):2438–2456, 2009.

[5] P. Bientinesi, B. Gunter, and R. A. v. d. Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35(1):3:1–3:22, July 2008. ISSN 0098-3500. . URL `http://doi.acm.org/10.1145/1377603.1377606`.

[6] J. Bilmes, K. Asanović, C. whye Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[7] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28:135–151, 2001.

[8] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating haskell array codes with multicore gpus. In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, pages 3–14. ACM, 2011. ISBN 978-1-4503-0486-3.

[9] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton-Jones, G. Keller, and S. Marlow. Data parallel Haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18. ACM, 2007. ISBN 978-1-59593-690-5. .

[10] E. Chan, F. G. V. Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. A. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In S. Chatterjee and M. L. Scott, editors, *PPOPP*, pages 123–132. ACM, 2008. ISBN 978-1-59593-795-7.

[11] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[12] A. Foltzer, A. Kulkarni, R. Swords, S. Sasidharan, E. Jiang, and R. Newton. A meta-scheduler for the par-monad: composable scheduling for the heterogeneous cloud. *SIGPLAN Not.*, 47(9):235–246, Sept. 2012. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2398856.2364562`.

[13] A. V. Gerbessiotis. Algorithmic and Practical Considerations for Dense Matrix Computations on the BSP Model. PRG-TR 32, Oxford University Computing Laboratory, 1997. URL `http://web.njit.edu/~alexg/pubs/papers/PRG3297.ps`.

[14] G. Golub and C. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[15] R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones of Computing, pages 245–262. Palgrave Macmillan, 2003.

[16] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th Intl. Conf. on Funct. Progr.*, ICFP '10, pages 261–272. ACM, 2010. ISBN 978-1-60558-794-3.

[17] O. Kiselyov, S. P. Jones, and C. chieh Shan. Fun with type functions. In A. W. Roscoe, C. B. Jones, and K. Wood, editors, *Reflections on the work of C. A. R. Hoare*. Springer, 2010.

[18] S. Marlow, S. L. Peyton-Jones, and S. Singh. Runtime support for multicore Haskell. In *ICFP 2009, Intl. Conf. on Functional Programming*, pages 65–78. ACM, 2009. .

[19] S. Marlow, P. Maier, H.-W. Loidl, M. K. Aswad, and P. W. Trinder. Seq no more: Better strategies for parallel Haskell. In *Haskell Symposium 2010*, Baltimore, MD, USA, Sept. 2010. ACM Press.

[20] S. Marlow, R. Newton, and S. Peyton-Jones. A monad for deterministic parallelism. In *Haskell '11: Proceedings of the Fourth Symposium on Haskell*. ACM, 2011.

[21] S. Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89329-5.

[22] S. Peyton-Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003.

[23] E. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2000.

[24] C. Whaley, A. Petitet, and J. J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. In *PARALLEL COMPUTING*, volume 27, 2000.