Making Programming Synonymous with Programming for Linear Algebra Libraries

FLAME Working Note #31

Maribel Castillo* Ernie Chan† Francisco D. Igual* Rafael Mayo*
Enrique S. Quintana-Ortí* Gregorio Quintana-Ortí* Robert van de Geijn†
Field G. Van Zee†

Abstract

We have invested heavily in hardware development but software tools and methods to use the hardware continue to fall behind. The Sky is Falling. Panic is rampant. We respectfully disagree for the domain of linear algebra, which is a typical example used to motivate high performance. Over the last ten years, we have been developing software tools and methods targeting this domain specifically to stay ahead of architectural development. In this paper, we give an overview of these methods and software tools, developed as part of the FLAME project. We show how when applied to a new architecture (GPUs), they provide an out-of-the-box solution that attains high performance almost effortlessly.

1 Introduction

Everyone finally gets it: parallel computing is here to stay and a broad cross-section of programmers will have to embrace it. Will software need to evolve or be rewritten? As part of the FLAME project we have been studying this question in the context of dense and banded matrix computations. In this paper we address the programmability issue head-on and demonstrate that our solution, which departs from the traditional evolutionary path, supports portability to new architectures by relating our experience with an NVIDIA multi-GPU system.

1.1 The evolutionary path: LAPACK

The most widely used linear algebra library is LAPACK [1]. This library evolved from the 1970s era LINPACK [16] when cache-based architectures became the norm rather than the exception. While the LAPACK project yielded many algorithmic innovations, it inherited the software architecture of LINPACK: it was coded in Fortran-77 and cast computations in terms of kernels supported by the Basic Linear Algebra Subprograms (BLAS). With the spread of parallel distributed-memory architectures in the early 1990s, LAPACK evolved into ScaLAPACK [14] which was built upon many of the same design decisions. We quote [31]:

> One of the design goals of ScaLAPACK was to have the ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

*Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain. {castillo,figual,mayo,quintana,gquintan}@icc.uji.es.

†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712. {echan,rvdg,field}@cs.utexas.edu.

A recent paper by LAPACK participants [18] can be taken as an indication that some LAPACK developers believe that this evolutionary strategy was successful.

How well-adapted the LAPACK family of solutions is to the drastic climate change that came with the arrival of multicore and other architectures that promise low power and high performance can be questioned. Their latest effort to port LAPACK to such architectures, the PLASMA project [10, 9], is conceptually similar to our own. Yet we are left to wonder whether it solves the programmability problem in this domain since the PLASMA project source code (as of this writing) has not yet been released for peer review. Indeed, the team leaders of LAPACK themselves have started to acknowledge the need for dense and banded linear algebra libraries to be rewritten from scratch [17].

1.2 The revolutionary path: FLAME

Since the advent of the 21st century, we have pioneered a radically different approach to library development. This effort built upon our observation that ScaLAPACK did not solve the programmability problem for parallel distributed-memory architectures. In the mid 1990s, this led us to develop the PLAPACK application programming interface (API) for programming such computations for that class of computers. This in turn made us rethink how to develop and program linear algebra libraries for all architectures as part of the Formal Linear Algebra Methods Environment (FLAME) project (`http://www.cs.utexas.edu/users/flame`). There is often confusion about what FLAME really is: a methodology, a library, or a programming-style choice? Here we briefly clarify.

The FLAME notation. The fundamental innovation that enabled FLAME is the notation we use for expressing dense and banded linear algebra algorithms (glimpse Figure 1 (left)). First used to discuss the numerical stability of an algorithm based on Gauss-Jordan transformations for inverting a general matrix [26], it allows linear algebra algorithms to be stated in a way that mirrors the corresponding illustrations that are often drawn on a chalkboard. The key is that the lack of indices –the ones that do appear are identifiers rather than indices in the usual sense– does not hinder but actually facilitates expressing how the computation progresses through matrices and vectors.

The FLAME methodology for deriving algorithms. The FLAME notation enabled the FLAME methodology, a set of verifiable methods for deriving algorithms from the mathematical specification of the operation.

Given an operation, a series of steps are employed to derive families of algorithms (multiple algorithmic variants) for computing the result [21, 20, 6]. The significance of this for scientific computing is that often different algorithmic variants deliver higher performance on different platforms and/or problem sizes [7, 27]. The methodology yields all loop-based algorithms and has been shown to be sufficiently systematic that it can be used in undergraduates classes for students with limited or no background in high-performance computing. This derivation of algorithms has also been made mechanical [5].

The FLAME APIs for representing algorithms in code. The third innovation came from the recognition that properly-defined APIs would allow the resulting code to closely mirror the algorithms as expressed using the FLAME notation [8]. We discuss one of these APIs, FLAME/C for the C programming language, in Section 2.1.

The FLASH extension for implementing algorithms-by-blocks. It has long been recognized that storing matrices by blocks, possibly recursively, rather than by rows or columns improves data locality and therefore has the potential to improve performance [22]. Unfortunately, it was also noted that programming algorithms when matrices are stored contiguously by blocks (sometimes called tiled algorithms or algorithms-by-blocks) is difficult [19]. It has been assumed that even though the matrices are stored by blocks (possibly recursively), the algorithms should be coded by indexing matrices as if they were flat. The basic observation in [25] was that programming algorithms-by-blocks becomes easy if one allows the blocked nature of the object to be expressed within the data structure that stores the matrices[1]. This storage by blocks is supported with FLASH, an extension of the FLAME/C API, by allowing elements in a matrix to be matrices themselves. It is discussed in Section 2.2 where we will show that using the right abstractions, coding algorithms-by-blocks becomes easy.

The SuperMatrix run-time system. Conventional wisdom would dictate that solutions to difficult problems, such as algorithm-by-blocks implementations, are necessarily difficult to port to a variety of platforms. Fortunately, the FLASH API supports a separation of concerns that allows the code to execute on multithreaded architectures without modification.

The key is to realize that, for algorithms-by-blocks, a block of the matrix is a basic unit of data, and that operations with blocks are basic units of computation. Techniques pioneered for use on superscalar architectures, such as out-of-order scheduling, can then be employed on submatrix operations. The operations and associated dependencies are inspected and represented in a directed acyclic graph (DAG). Much as code written in a high-level language does not need to express individual operations to the superscalar architecture, code written using the FLASH API does not need to express the individual operations to a run-time system that uses the DAG to schedule operations. Thus, the separation of concerns cleanly divorces the library code written with the FLASH API from the run-time system that schedules the operations. Details are given in Section 3 as well as various other papers published since January 2007 [11, 13, 28, 29, 12, 30].

We note that the idea of viewing this problem as a DAG of operations and dependencies is not new and is currently also being pursued by a number of other projects [4, 10, 9, 23].

An alternative evolutionary path. Our experience over the last decade has shown this new approach to developing linear algebra libraries to be highly adaptive to new environments. In Section 4 we show how a new computing environment, characterized by multi-GPU architectures, is handily supported.

1.3 Contributions of this paper

The first contribution of this paper is that it openly poses the question of whether it is time for a drastic change in course for linear algebra library development. With the advent of multicore architectures, programmability has become **the** issue. This paper gives an overview of a project that provides an alternative that may be better suited for the change that we are trying to accommodate.

The second contribution comes from a much more detailed discussion of how our approach supports a separation of concerns between the library code programmed using FLASH and the scheduling performed by the run-time system.

The third contribution comes from a performance section that focuses on the programmability issue rather than just performance. In it we describe the effort required to retarget our libraries to a multi-GPU platform.

1.4 How to read this paper

The busy reader could skip Sections 2–3 and go directly to the sections that discuss the port to architectures with GPU-based accelerators and the performance that is attained. But that would be a mistake: this paper is about programmability. It is our experience that even after reading our other papers on this subject, experts in the field still do not appreciate how well the FLAME/FLASH/SuperMatrix approach solves the programmability problem. It is precisely in Sections 2–3 that we attempt to better expose that feature of our work.

[1] Such data structures were already proposed as early as the 1970s [32] and were revived in [15], yet were ignored by the mainstream community.

```
Algorithm: $A := \textsc{Chol\_blk\_var1}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
        where $A_{11}$ is $b \times b$

    $A_{11} := \{L \backslash A\}_{11} = \textsc{Chol\_unb}(A_{11})$
    $A_{21} := L_{21} = A_{21} L_{11}^{-\mathrm{T}}$
    $A_{22} := A_{22} - L_{21} L_{12}^{\mathrm{T}} = A_{22} - A_{21} A_{21}^{\mathrm{T}}$

    Continue with
        $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
endwhile
```

```
FLA_Error FLA_Chol_blk_var1( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    b = min( FLA_Obj_length(ABR), nb_alg );
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,  b, b, FLA_BR );
    /*----------------------------------------------*/
    FLA_Chol_unb_var1( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11, A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21, FLA_ONE,       A22 );
    /*----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,      FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Figure 1: Blocked algorithm for computing the Cholesky factorization (left) and the corresponding FLAME/C implementation (right).

2 Programming Algorithms-by-Blocks Made Easy

To describe how developing algorithms-by-blocks is a matter of minutes rather than weeks or months, we will consider the Cholesky factorization of an $n \times n$ symmetric positive definite (SPD) matrix A. In this operation the matrix is decomposed into the product $A = LL^T$, where L is the $n \times n$ lower triangular Cholesky factor. (Alternatively, A can be decomposed as $A = U^T U$, with U being upper triangular.) In traditional algorithms for this factorization, L overwrites the lower triangular part of A while the strictly upper triangular part remains unmodified. Here, we denote this as $A := \{L \backslash A\}$.

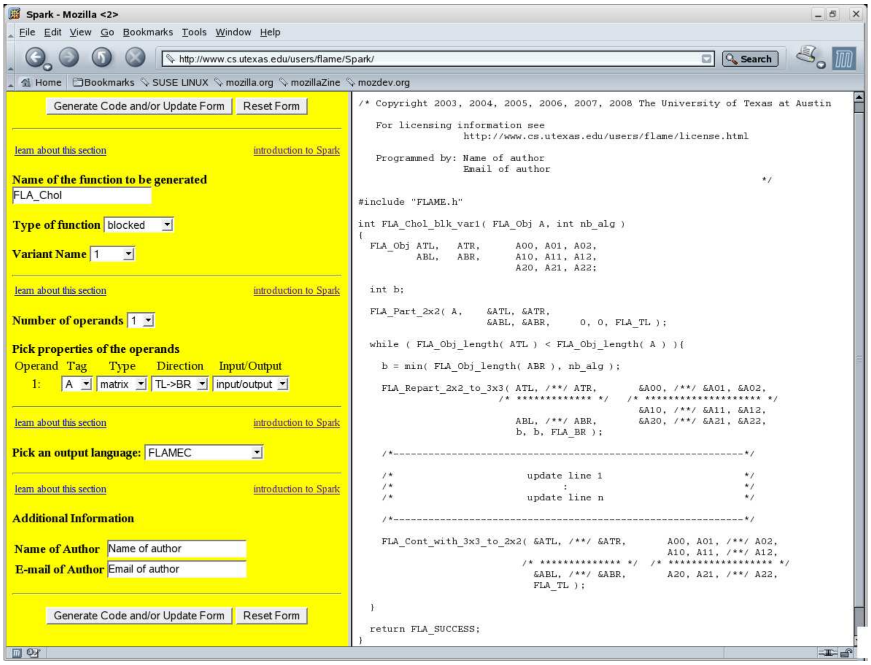
2.1 The FLAME API for expressing linear algebra operations

Our starting point will be the blocked algorithm in Figure 1 (left) for overwriting (the lower triangular part of) a SPD matrix A with the Cholesky factor L. The algorithm, expressed in the FLAME notation, builds upon an unblocked variant (Chol_unb) which computes the factorization of the diagonal block A_{11}. There, $m(B)$ stands for the number of rows of B. We believe the rest of notation to be intuitive.

While having a domain-specific notation to typeset linear algebra algorithms at at high level of abstraction is nice, no current compiler can process the algorithm in Figure 1 (left) and a transformation (manual or automatic) is necessary. The FLAME/C API for the C programming language captures the notation in which we express our algorithms. Using this API, the blocked algorithm can be transformed into the C code in Figure 1 (right). Note the close resemblance between algorithm and code.

Next, we recognize that indentation plays an importing role in making the FLAME/C code look like the algorithm. Making the code aestetically pleasing using the FLAME/C API by hand is an arduous task, and therefore we recommend instead the use of a high-level mechanical tool like the Spark webpage (`http://www.cs.utexas.edu/users/flame/Spark/`) which automatically yields a code skeleton. A screen capture of the tool is shown in Figure 2. The sequence of steps to generate the code using Spark is the following:

1. Choose the name of the function and indicate whether an unblocked or a blocked routine is to be generated.


Spark - Mozilla <2>
File Edit View Go Bookmarks Tools Window Help
http://www.cs.utexas.edu/users/flame/Spark/
Search
Home Bookmarks SUSE LINUX mozilla.org mozillaZine mozdev.org
Generate Code and/or Update Form
Reset Form
learn about this section
introduction to Spark

Name of the function to be generated: FLA_Chol

Type of function: blocked

Variant Name: 1

Number of operands: 1

Pick properties of the operands

Operand	Tag	Type	Direction	Input/Output
1:	A	matrix	TL->BR	input/output

Pick an output language: FLAMEC

Additional Information

Name of Author: Name of author

E-mail of Author: Email of author

```
/* Copyright 2003, 2004, 2005, 2006, 2007, 2008 The University of Texas at Austin

   For licensing information see
                  http://www.cs.utexas.edu/users/flame/license.html

   Programmed by: Name of author
                  Email of author
                                                                          */

#include "FLAME.h"

int FLA_Chol_blk_var1( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL,   ATR,      A00, A01, A02,
          ABL,   ABR,      A10, A11, A12,
                           A20, A21, A22;

  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,     0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ){

    b = min( FLA_Obj_length( ABR ), nb_alg );

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                        /* ************* */   /* ******************** */
                                                &A10, /**/ &A11, &A12,
                           ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
                           b, b, FLA_BR );

    /*------------------------------------------------------------*/

    /*                      update line 1                         */
    /*                            :                               */
    /*                      update line n                         */

    /*------------------------------------------------------------*/

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,       A00, A01, /**/ A02,
                                                     A10, A11, /**/ A12,
                            /* ************** */  /* ****************** */
                              &ABL, /**/ &ABR,       A20, A21, /**/ A22,
                              FLA_TL );

  }

  return FLA_SUCCESS;
}
```

Figure 2: The Spark FLAME code-skeleton generator.

2. Specify the number of operands, their types (matrix, vector or scalar), the direction in which they are traversed in the algorithm (e.g., in the Cholesky factorization, A is traversed from the top left corner to the bottom right one), and indicate whether the operand is an input (only read), an input/output (both read and written), or a temporary operand.

3. Select the output language: FLAME/C for C, FLAME@lab for M-script, FLaTeX for documents, etc.

4. Cut and paste the result into your favorite text editor, and in case the output FLAME/C is chosen, fill-in the updates (in between the dashed lines) using the high-level wrappers to the external BLAS in the FLAME BLAS library (`http://www.cs.utexas.edu/users/flame/Reference/index.htm`).

We encourage the reader to reproduce the code for the Cholesky factorization using the Spark FLAME code-skeleton generator. Further details of how to use the Spark tool can be found in [33].

The benefits of representing algorithms in code using APIs like FLAME/C and tools like Spark should be self-evident. Creating a blocked code for the Cholesky factorization like the one in Figure 1 (right) with a high confidence of correctness takes a few minutes.

2.2 The FLASH API for storage-by-blocks

Fred Gustavson (IBM) has long advocated for *algorithms-by-blocks* [19] which view matrices as collections of submatrices and express their computation in terms of these submatrix blocks. Algorithms are then written as before, except with scalar operations replaced by operations on the blocks. Although a number of solutions have been proposed to solve this problem, none of these have yielded a consistent methodology that allows the development of high-performance libraries with functionality that rivals those of LAPACK or FLAME. The problem is primarily one of *programmability*.

Our approach to the problem views the matrix as a matrix of smaller matrices using the FLASH API. This view thus yields a matrix hierarchy, potentially with multiple levels. Code for an algorithm-by-blocks for the Cholesky factorization using the FLASH API is given in Figure 3 (left). This code has evolved from the blocked implementation in Figure 1 following three very simple steps:

```
FLA_Error FLASH_Chol_by_blocks_var1( FLA_Obj A )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*---------------------------------------------*/
    FLA_Chol_unb_var1( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21,
                FLA_ONE,       A22 );
    /*---------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
void FLASH_Trsm_rltn( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                 FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                               )
   Assumption: L consists of one block and
               B consists of a column of blocks     */
{
  FLA_Obj BT,      B0,
          BB,      B1,
                   B2;

  FLA_Part_2x1( B,    &BT,
                      &BB,    0, FLA_TOP );

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    FLA_Repart_2x1_to_3x1( BT,       &B0,
                        /* ** */   /* ** */
                                     &B1,
                           BB,       &B2,    1, FLA_BOTTOM );
    /*---------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              alpha, FLASH_MATRIX_AT( L ),
                     FLASH_MATRIX_AT( B1 ) );
    /*---------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                                          B1,
                            /* ** */   /* ** */
                              &BB,        B2,     FLA_TOP );
  }
}
```

Figure 3: FLASH implementation of the Cholesky factorization and the corresponding triangular system solve.

1. Replace references to `b` in `FLA_Repart_2x2_to_3x3` by the scalar "1" and eliminate remaining references to `b` and `nb_alg`.

2. Add the macro `FLASH_MATRIX_AT` to access the operands in those routines that operate on a single block: `FLA_Chol_unb_var1`.

3. Substitute routines that operate on blocks (`FLA_Trsm`, `FLA_Syrk`) by the corresponding algorithms-by-blocks (`FLASH_Trsm`, `FLASH_Syrk`) to which now matrices of matrix blocks are passed.

It may seem that the complexity of the algorithm is merely hidden in the routines `FLASH_Trsm` and `FLASH_Syrk`. The abbreviated implementation of an algorithm-by-blocks for the former is given in Figure 3 (right) while the latter routine has a similar implementation, which itself can be coded in a matter of minutes using the Spark webpage. The reader can see here that many of the details of the FLASH implementation have been buried within the FLASH-aware FLAME object definition.

In summary, the FLASH API can be used to rapidly evolve blocked implementations of these operations into algorithms-by-blocks. As a result, transforming blocked algorithms into algorithms-by-blocks and/or developing algorithms-by-blocks from scratch using the FLASH API is straightforward and allows us to keep the details of hierarchically storing the matrices by blocks completely hidden from the user.

3 Parallelizing Algorithms-by-Blocks Made Easy

Our approach to parallelizing dense and band linear algebra operations is based on a separation of concerns. The code, which embeds a formal representation of the knowledge in the numerical method, should not change depending on the target platform. Although the traditional approach on multithreaded architectures (basically shared-memory parallel platforms as SMP systems, NUMA, or multicore processors) satisfies this requirement by hiding all complexity under the covers of a multithreaded implementation of BLAS, we have demonstrated that for many dense and banded linear algebra operations this comes at a cost in performance [11, 13, 28, 29, 12, 30]. While a performance penalty may be acceptable if it benefits programmability, in these papers we argue that neither programmability nor performance need be sacrificed. It show be noted, furthermore, that when multiple specialized accelerators are to be used, a parallel implementation of BLAS needs to be developed from scratch.

Our approach is different. We shift the burden of parallel execution onto a run-time system, SuperMatrix, to keep the coded algorithms simple and unaltered. SuperMatrix extracts the parallelism at a high level of abstraction, decomposing the operation into tasks, identifying the dependencies among these, scheduling them for execution when ready (all operands available/dependencies fulfilled), and mapping tasks to execution units (cores/accelerators) taking into account the target platform. All of this is done without exposing any of the details of the parallelization to the application programmer.

In particular, the SuperMatrix run-time computes the Cholesky factorization by executing the algorithm-by-blocks in Figure 3 (left) in two stages: the *analyzer* and *dispatcher* stages.

- **Analyzer.** During this first stage, the run-time system executes the algorithm code, but instead of executing operations immediately as they are encountered, SuperMatrix enqueues these operations in a list of pending tasks. This happens inside the calls to `FLA_Chol_unb_var1`, `FLA_Trsm`, `FLA_Syrk`, and `FLA_Gemm` encountered in the routines `FLASH_Chol_by_blocks_var1`, `FLASH_Trsm`, and `FLASH_Syrk`. This stage is performed sequentially and uses the order in which suboperations appear in the code and the operands they read and/or write (which are determined by the syntax of the BLAS) to determine the dependencies. The result is a DAG that contains the dependencies among suboperations of the overall problem.

- **Dispatcher.** In the second stage, tasks with all dependencies fulfilled are dequeued from this list and scheduled for computation to the execution units. Upon completion of a task, the list of pending tasks is revised to update the dependencies that are fulfilled.

To illustrate these stages, let us show what happens during the first steps of each stage when the code in Figure 3 is to operate on the 3 × 3 blocked matrix

$$A \rightarrow \left(\begin{array}{ccc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{array} \right). \tag{1}$$

Consider first the analyzer stage. During the first iteration of the `while` loop in the code, A_{TL} is an empty matrix so that $A = A_{BR}$ and

$$A \rightarrow \left(\begin{array}{|c|c} \hline \texttt{A11} & \texttt{A12} \\ \hline \texttt{A21} & \texttt{A22} \end{array} \right) = \left(\begin{array}{|c|cc} \hline \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{array} \right). \tag{2}$$

Thus, when the SuperMatrix run-time system (executed by a single thread) encounters the call

```
FLA_Chol_unb_var1( FLASH_MATRIX_AT( A11 ) );
```

it enqueues the computation of the Cholesky factorization of block `A11`$= \bar{A}_{00}$ as a pending task. Next, the call to

```
FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
            FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
            FLA_ONE, A11, A21 );
```

is reached and, correspondingly, the code in Figure 3 (right) is executed with `L` $=$ `A11` $= \bar{A}_{00}$ and `B` $=$ `A21`$= \left(\begin{array}{c} \bar{A}_{10} \\ \bar{A}_{20} \end{array} \right)$ as arguments. During execution of the latter routine, two calls are encountered to

```
FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
          FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
          FLA_ONE, FLASH_MATRIX_AT( L ), FLASH_MATRIX_AT( B1 ) );
```

	Original table			After 1st oper.			After 3rd oper.			After 6th oper.		
Operation/Result	In		In/out	In		In/out	In		In/out	In		In/out
1. $\textsc{Chol}(\bar{A}_{00})$			$\bar{A}_{00}\surd$									
2. $\bar{A}_{10}\,\textsc{tril}(\bar{A}_{00})^{-\mathrm{T}}$	$\bar{A}_{00}$		$\bar{A}_{10}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{10}\surd$						
3. $\bar{A}_{20}\,\textsc{tril}(\bar{A}_{00})^{-\mathrm{T}}$	$\bar{A}_{00}$		$\bar{A}_{20}\surd$	$\bar{A}_{00}\surd$		$\bar{A}_{20}\surd$						
4. $\bar{A}_{11}-\bar{A}_{10}\bar{A}_{10}^{\mathrm{T}}$	$\bar{A}_{10}$		$\bar{A}_{11}\surd$	$\bar{A}_{10}$		$\bar{A}_{11}\surd$	$\bar{A}_{10}\surd$		$\bar{A}_{11}\surd$			
5. $\bar{A}_{21}-\bar{A}_{20}\bar{A}_{10}^{\mathrm{T}}$	$\bar{A}_{20}$	$\bar{A}_{10}$	$\bar{A}_{21}\surd$	$\bar{A}_{20}$	$\bar{A}_{10}$	$\bar{A}_{21}\surd$	$\bar{A}_{20}\surd$	$\bar{A}_{10}\surd$	$\bar{A}_{21}\surd$			
6. $\bar{A}_{22}-\bar{A}_{20}\bar{A}_{20}^{\mathrm{T}}$	$\bar{A}_{20}$		$\bar{A}_{22}\surd$	$\bar{A}_{20}$		$\bar{A}_{22}\surd$	$\bar{A}_{20}\surd$		$\bar{A}_{22}\surd$			
7. $\textsc{Chol}(\bar{A}_{11})$			$\bar{A}_{11}$			$\bar{A}_{11}$			$\bar{A}_{11}$			$\bar{A}_{11}\surd$
8. $\bar{A}_{21}\,\textsc{tril}(\bar{A}_{11})^{-\mathrm{T}}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}$	$\bar{A}_{11}$		$\bar{A}_{21}\surd$
9. $\bar{A}_{22}-\bar{A}_{21}\bar{A}_{21}^{\mathrm{T}}$	$\bar{A}_{21}$		$\bar{A}_{22}$	$\bar{A}_{21}$		$\bar{A}_{22}$	$\bar{A}_{21}$		$\bar{A}_{22}$	$\bar{A}_{21}$	$\bar{A}_{12}$	$\bar{A}_{22}\surd$
10. $\textsc{Chol}(\bar{A}_{22})$			$\bar{A}_{22}$			$\bar{A}_{22}$			$\bar{A}_{22}$			$\bar{A}_{22}$

Figure 4: An illustration of the scheduling of operations for the Cholesky factorization of a 3×3 matrix of blocks in (1) using the algorithm-by-blocks `FLASH_Chol_by_blocks_var1`. The “$\surd$”-marks denote those operands that are available (i.e., those operands that are not dependent upon other operations).

one per block in `A21` ($\bar{A}_{10}$ and $\bar{A}_{20}$), upon which two new tasks are enqueued. Here a dependency is encountered: `A11` is an output (result) for `FLA_Chol_unb_var1` but an input operand to each one of these calls to `FLA_Trsm` (as parameter `L`). Therefore a read-after-write dependency must be recorded as part of the list that implicitly represents the DAG of dependencies. The execution of the analyzer stage proceeds in this manner until all operations on blocks have been recorded. Conceptually, the result is a list with the information contained in the two leftmost columns of Figure 4.

The dispatcher stage begins with the list of tasks and dependencies, the DAG, as input. As parallel execution proceeds, idle threads consult the list for tasks with all operands available (dependencies fulfilled) to discover that, initially, the only task to satisfy such requirement corresponds to the Cholesky factorization of $\bar{A}_{00}$. A single thread executes this task and, upon completion, updates the list marking the triangular systems solves with $\bar{A}_{10}$ and $\bar{A}_{20}$ as ready (see column labeled as “After 1st oper.” in Figure 4). Two threads can then dequeue these tasks and execute them in parallel, updating the corresponding entries in the list once they are completed (see column labeled as “After 3rd oper.”). Execution continues in this manner till all tasks have been completed. Note that the real scheduling of operations does not necessarily occur in the described order. As long as dependencies are respected, other schedulings are possible. Thus, e.g., the operation marked as 4. in Figure 4 could be computed before that marked as 3. These are possible optimizations. Thanks to the separation of concerns, this only affects the dispatcher stage of SuperMatrix. The algorithm implementations remain unchanged.

In summary, we combine two techniques from superscalar processors, dynamic scheduling and out-of-order execution, while hiding the management of data dependencies from both library developers and users. Our approach separates programming of linear algebra operations from the execution on the specific parallel architecture: multithreaded architecture or specialized accelerator. As a result no change is needed in the FLASH codes that implement the linear algebra operations to execute them using multiple cores or GPUs.

4 A Demonstration of Programmability: Porting libFLAME to a multi-GPU system

The arrival of an NVIDIA Tesla S870 computing system with 4 NVIDIA G80 GPUs provided us with a perfect opportunity to test the portability of the FLAME/FLASH/SuperMatrix solution. The system was assembled on April 8, only 6 days before the deadline for submissions to SC08.

4.1 The experiment

At the time of arrival, we had some experience with a single NVIDIA G80 GPU processor, which we had programmed using an LAPACK style of coding, employing NVIDIA CUBLAS as we implemented various LAPACK-level operations [3]. At the time of arrival, CUBLAS were not parallelized to employ multiple GPUs and therefore proceeding in a fashion where parallelism was derived from parallel BLAS was not an option.

What we did have available was libFLAME Version 2.0 [24], the library that has resulted from the FLAME project. This library includes implementation of the FLAME/C and FLASH interfaces and the SuperMatrix run-time system. It is written in C and distributed as free software under the GNU *Lesser General Public License* (LGPL). The library supports all BLAS operations as well as the major dense factorizations (Cholesky, LU with pivoting, and QR) for the solution of linear systems and linear least-squares problems. On a single processor, libFLAME provides performance comparable with that of the underlying BLAS implementation. On multithreaded platforms with several processing cores, libFLAME exploits the scheduling mechanism in the SuperMatrix run-time system to deliver much higher performance than the LAPACK approach based on multithreaded implementations of BLAS. This library had been successfully used by us to attain high performance on various multicore architectures [11, 13, 28, 29, 12, 30]. Now the goal was to get libFLAME up and running on the NVIDIA multi-GPU system.

Now, no parallel implementation of BLAS is needed to execute libFLAME on multithreaded architectures. Thus, porting libFLAME to a system with multiple GPUs only required us to write wrappers for the CUBLAS which internally dealt with the data transfer between RAM and video memory.

All experiments were performed using single-precision floating-point arithmetic.

4.2 The target platform

The target platform was an NVIDIA Tesla S870 computing system with 4 NVIDIA G80 GPUs and 6 GBytes of DDR3 memory (1.5 GBytes per GPU), which exhibits a global peak performance of almost 1.4 TeraFLOPS (346×10^9 floating-point single precision arithmetic operations, or flops, per second on each GPU). The Tesla system is connected to a server with one Intel Xeon QuadCore E5405 processor running at 2.0 GHz and with 9 GBytes of DDR2 RAM. The Intel 5400 chipset provides two PCIExpress Gen2 interfaces, for a peak bandwidth of 48 GBytes/second on each interface, to connect with the Tesla. NVIDIA CUBLAS (version 1.1) built on top of the CUDA API (version 1.1) together with NVIDIA driver (171.05) were used in our tests.

4.3 Results

When computing the rate of computation, we consider the cost of the Cholesky factorization to be the standard $n^3/3$ flops for a square matrix of order n so that the GFLOPS rate is computed as $n^3/(3t \times 10^{-9})$, where t equals elapsed time in seconds.

Figure 5 reports the programming effort that was required to port the algorithm-by-blocks for the Cholesky factorization to the multi-GPU platform, once the system was up and the software libraries installed. Transforming the existing implementation of SuperMatrix to execute operations with blocks on the Tesla S870 using the native CUBLAS required about 1 hour from a single programmer and resulted in the performance reported in the line labeled as A. From there on, we implemented a few optimizations to tailor the run-time system and increase the performance (lines labeled as B, C, and D). All times reported in the figure refer to hours of work by one of the coauthors. Optimization B reduced the volume of data transfered by employing a 2-D distribution of blocks to GPUs and an “owner-computes” rule to map computations to GPUs. Optimization C further reduced this overhead by implementing a simple software-base cache system of blocks in the GPU memory. Optimization D employed page-locked memory.

Overall 12 hours from a single programmer with no experience with a multi-GPU system were required to attain around 180 GFLOPS on the Cholesky factorization.

Although the number is low when compared with the peak performance of the system (1.4 TeraFLOPS), let us put this into perspective. In Figure 6 we compare the performance with that of optimized implemen-

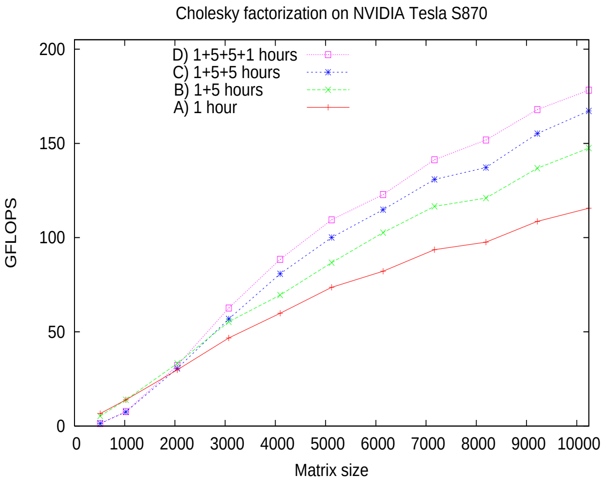


Figure 5: Programming effort required to port the algorithm-by-blocks for the Cholesky factorization to NVIDIA Tesla s870.

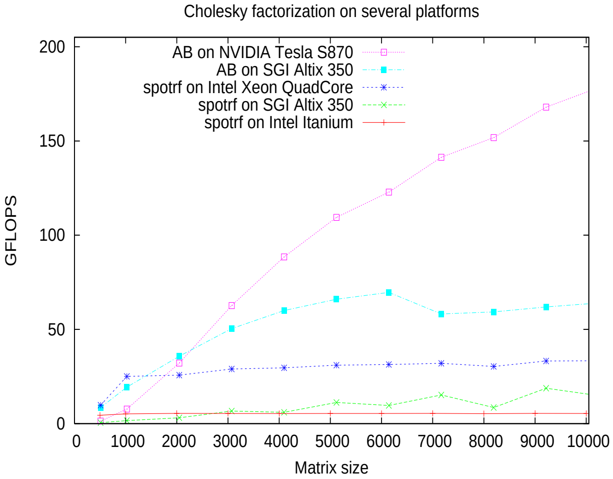


Figure 6: Performance of the Cholesky factorization on several platforms.

tations of the Cholesky factorization on current high-performance platforms. Single precision was employed in all cases:

- `spotrf` on Intel Itanium: Multithreaded MKL 8.1 implementation of LAPACK routine `spotrf` executed on a 1.5 GHz Intel Itanium2 processor.
- `spotrf` on SGI Altix 350: Multithreaded MKL 8.1 implementation of LAPACK routine `spotrf` executed on a ccNUMA platform with 16 Intel Itanium2 processors at 1.5 GHz which share 32 GBytes of RAM and connected via a SGI NUMAlink.
- `spotrf` on Intel Xeon QuadCore: Multithreaded MKL 10.0 implementation of LAPACK routine `spotrf` executed on a quad-core Xeon SMP platform.

- `AB` on SGI Altix 350: Our algorithm-by-blocks linked with sequential BLAS in MKL on the the SGI Altix 350.
- `AB` on NVIDIA Tesla S870: Our algorithm-by-blocks linked with CUBLAS 1.1 on the Tesla platform (4 GPUs).

The results show that, on the SGI Altix 350, our algorithm-by-blocks clearly outperforms the highly tuned implementations provided by MKL. The Tesla S870 combined with the algorithm-by-blocks offers a remarkable GFLOPS rate when compared with the multithreaded architectures at a much lower price. (The SGI Altix system, which is about 4 years old, cost 20-30 times more than the current retail price of the Tesla system.)

A few notes are due to conclude this section:

– Our implementation will immediately benefit from any improvement in the sequential performance of the CUBLAS [2] and/or hardware (e.g., higher data transference rate).

– The optimizations on the run-time system to address multi-GPU environments are representative of all other operations in BLAS and dense/banded factorization algorithms. There is no need to develop a parallel implementation of BLAS for these factorizations. One optimization thus targets multiple operations, reducing the development effort.

– We envision that a similar strategy is valid for other platforms with multiple hardware accelerators (e.g., systems with several ClearSpeed boards) or multicore-like environments (Cell B.E.).

5 Conclusion

While architectural advances promise to deliver a high level of parallelism in the form of many-core platforms, it is widely accepted that it is programmability that will determine the success of these architectures. In this paper we have illustrated how the notation, APIs, and tools that are part of the FLAME project provide modern abstractions that increase developer productivity and user-friendliness alike in the context of dense and banded linear algebra libraries. This makes this approach a modern alternative to libraries that have evolved since the 1970s.

In particular we show that, with the appropriate support from the FLAME and FLASH APIs, developing algorithms-by-blocks for a dense linear algebra operation like the Cholesky factorization is easy. (Here, we assume that the algorithm for the operation is already known. Otherwise, we need to add to that the time required to use the mechanical system in [5] to derive it.)

Using the algorithm-by-blocks as input, SuperMatrix transparently identifies suboperations on blocks of the matrices and schedules the corresponding tasks to a system with multiple cores/processors. Our experience with the Cholesky factorization shows that porting this mechanism to platforms with multiple hardware accelerators, like GPUs, is straight-forward.

Additional information

For additional information on FLAME visit `http://www.cs.utexas.edu/users/flame/`.

Acknowledgements

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. Additional support came from the *J. Tinsley Oden Faculty Fellowship Research Program* of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin. The researchers at the Universidad Jaime I were supported by projects CICYT TIN2005-09037-C02-02 and FEDER, and P1B-2007-19 and P1B-2007-32 of the *Fundación Caixa-Castellón/Bancaixa* and UJI.

We thank NVIDIA for the generous donation of equipment that was used in the experiments.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[2] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing – PDSEC'08*, 2008. To appear.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. Technical Report ICC 02-02-2008, Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores, February 2008.

[4] Pieter Bellens, Josep M. Pérez, Rosa M. Badía, and Jesús Labarta. CellSs: a programming model for the Cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM Press.

[5] Paolo Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms.* PhD thesis, 2006.

[6] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[7] Paolo Bientinesi, Brian Gunter, and Robert A. Van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 2009. to appear.

[8] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

[9] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 190 UT-CS-07-600, University of Tennessee, September 2007.

[10] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. LAPACK Working Note 190 UT-CS-07-598, University of Tennessee, July 2007.

[11] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007a. ACM.

[12] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN 2008 symposium on Principles and practices of parallel programming (PPoPP'08)*, 2008. to appear.

[13] Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Satisfying your dependencies with SuperMatrix. In *Proceedings of IEEE Cluster Computing 2007*, pages 91–99, September 2007b.

[14] 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.

[15] Timothy Collins and James C. Browne. Matrix++: An objectoriented environment for parallel high-perfomance matrix computations. In *Proc. of the Hawaii Intl. Conf. on Systems and Software*, 1995.

[16] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users' Guide*. SIAM, Philadelphia, 1979.

[17] Jack Dongarra. Future directions in high performance computing. Talk presented at Google, Mountain View, CA, January 2008.

[18] Jack Dongarra and Piotr Luszczek. How elegant code evolves with hardware: The case of gaussian elimination. In Andy Oram and Greg Wilson, editors, *Beautiful Code. Leading Programmers Explain How They Think*, pages 229–252. O'Reilly, 2007.

[19] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[20] John Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, UT-Austin, in preparation.

[21] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[22] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, Feb. 1992.

[23] C. Leiserson and A. Plaat. Programming parallel applications in cilk. *SINEWS: SIAM News*, 1998.

[24] libFLAME 2.0. `www.cs.utexas.edu/users/flame/ReleaseNotes/`, April 2008.

[25] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-2004-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[26] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert vande Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Stat. Comput.*, 22(5):1762–1771, 2000.

[27] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Trans. Math. Soft.*, 29(2):218–243, June 2003.

[28] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design of scalable dense linear algebra libraries for multithreaded architectures: the LU factorization. In *First Workshop on Programmability Issues for Multi-Core Computers*, 2008.

[29] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Field G. Van Zee, and Robert A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In F. Spies D. El Baz, J. Bourgeois, editor, *16th Euromicro International Conference on Parallel, Distributed and Network-based Processing – PDP 2008*, pages 301–310, 2008.

[30] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Alfredo Remón, and Robert van de Geijn. Supermatrix for the factorization of band matrices. FLAME Working Note #27 TR-07-51, The University of Texas at Austin, Department of Computer Sciences, September 2007b.

[31] `http://www.netlib.org/scalapack/scalapack_home.html`.

[32] G.M. Skagestein. *Rekursiv Unterteilte Matrizen Sowie Methoden Zur Erstellung Von Rechnerprogrammen Fur Ihre Verarbeitung*. PhD thesis, Universitat Stuttgart, 1972.

[33] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www.lulu.com`, 2008.