Rapid Development of High-Performance Linear Algebra Libraries

Paolo Bientinesi[1], John Gunnels[4], Fred Gustavson[4], Greg Henry[3] Margaret Myers[1], Enrique Quintana-Orti[2], and Robert van de Geijn[1]

[1] Department of Computer Sciences
The University of Texas at Austin
{pauldj, myers, rvdg}@cs.utexas.edu
[2] Universidad Jaume I, Spain
quintana@inf.uji.es
[3] Intel Corp.
greg.henry@intel.com
[4] IBM's T.J. Watson Research Center
{gunnels, fg2}@us.ibm.com

Abstract. We present a systematic methodology for deriving and implementing linear algebra libraries. It is quite common that an application requires a library of routines for the computation of linear algebra operations that are not (exactly) supported by commonly used libraries like LAPACK. In this situation, the application developer has the option of casting the operation into one supported by an existing library, often at the expense of performance, or implementing a custom library, often requiring considerable effort. Our recent discovery of a methodology based on formal derivation of algorithm allows such a user to quickly derive proven correct algorithms. Furthermore it provides an API that allows the so-derived algorithms to be quickly translated into high-performance implementations.

1 Introduction

We have recently written a series of journal papers where we illustrate to the HPC community the benefits of the formal derivation of algorithms [7, 2, 11, 3]. In those papers, we show that the methodology greatly simplifies the derivation and implementation of algorithms for a broad spectrum of dense linear algebra operations. Specifically, it has been successfully applied to all Basic Linear Algebra Subprograms (BLAS) [9, 5, 4], most operations supported by the Linear Algebra Package (LAPACK) [1], and many operations encountered in control theory [8]. We illustrate the methodology and its benefits by applying it to the inversion of a triangular matrix, $L := L^{-1}$, and operation supported by the LAPACK routine DTRTRI.

2 A worksheet for deriving linear algebra algorithms

In Fig. 1, we give a generic "worksheet" for deriving a large class of linear algebra algorithms. Expressions in curly-brackets denote predicates that describe the state of

Step	Annotated Algorithm: $[D, E, F, \ldots] = \mathrm{op}(A, B, C, D, \ldots)$
1a	$\{P_{\mathrm{pre}}\}$
4	**Partition** **where**
2	$\{P_{\mathrm{inv}}\}$
3	**while** G **do**
2,3	$\{(P_{\mathrm{inv}}) \wedge (G)\}$
5a	**Repartition** **where**
6	$\{P_{\mathrm{before}}\}$
8	S_U
7	$\{P_{\mathrm{after}}\}$
5b	**Continue with**
2	$\{P_{\mathrm{inv}}\}$
	enddo
2,3	$\{(P_{\mathrm{inv}}) \wedge \neg (G)\}$
1b	$\{P_{\mathrm{post}}\}$

Fig. 1. Worksheet for developing linear algebra algorithms.

the various variables at the given point of the algorithm. For this paper, it suffices to realize that the statements between the assertions in the curly-brackets must be such that, at the indicated points in the algorithm, those assertions hold.

3 Example: Triangular matrix inversion

Let us consider the example $L := L^{-1}$ where L is an $m \times m$ lower triangular matrix. This is similar to the operation provided by the LAPACK routine DTRTRI [4]. In the discussion below the "Steps" refer to the step numbers in the left column of Figs. 1 and 2.

Step 1: Determine P_{pre} and P_{post}. The conditions before the operation commences (the precondition) can be described by the predicate indicated in Step 1a in Fig. 2. Here $\hat{L}$ indicates the original contents of matrix L. The predicate in Step 1b in Fig. 2 indicates the desired state upon completion (the postcondition).

Step 2: Determine P_{inv}. In order to determine possible intermediate contents of the matrix L, one starts by partitioning the input and output operands, in this case L and $\hat{L}$. The partitioning corresponds to an assumption that algorithms progress through data in a systematic fashion. Since L is lower triangular, it becomes important to partition it into four quadrants,

$$L \rightarrow \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right),$$

where L_{TL} and L_{BR} are square so that these diagonal blocks are lower triangular. Here the indices T, B, L, and R stand for Top, Bottom, Left, and Right, respectively.

<table>
<tr><th>Step</th><th>Annotated Algorithm: $L := L^{-1}$</th></tr>
<tr><td>1a</td><td>$\left\{ L = \hat{L} \wedge \text{LowerTri}(L) \right\}$</td></tr>
<tr><td>4</td><td>Partition $L = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$ and $\hat{L} = \left(\begin{array}{c||c} \hat{L}_{TL} & 0 \\ \hline\hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right)$
where L_{TL} and $\hat{L}_{TL}$ are 0×0</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c} L_{TL}^{-1} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \right\}$</td></tr>
<tr><td>3</td><td>while $\neg\text{SameSize}(L, L_{TL})$ do</td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c} L_{TL}^{-1} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \right) \wedge (\neg\text{SameSize}(L, L_{TL})) \right\}$</td></tr>
<tr><td>5a</td><td>Determine block size b
Repartition
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right)$ and $\left(\begin{array}{c||c} \hat{L}_{TL} & 0 \\ \hline\hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} \hat{L}_{00} & 0 & 0 \\ \hline\hline \hat{L}_{10} & \hat{L}_{11} & 0 \\ \hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right)$
where L_{11} and $\hat{L}_{11}$ are $b \times b$</td></tr>
<tr><td>6</td><td>$\left\{ \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c||c|c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline\hline \hat{L}_{10} & \hat{L}_{11} & 0 \\ \hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right) \right\}$</td></tr>
<tr><td>8</td><td>$L_{10} := -L_{11}^{-1} L_{10} L_{00}$
$L_{11} := L_{11}^{-1}$</td></tr>
<tr><td>7</td><td>$\left\{ \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c|c||c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline -\hat{L}_{11}^{-1} \hat{L}_{10} \hat{L}_{00}^{-1} & \hat{L}_{11}^{-1} & 0 \\ \hline\hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right) \right\}$</td></tr>
<tr><td>5b</td><td>Continue with
$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array}\right)$ and $\left(\begin{array}{c||c} \hat{L}_{TL} & 0 \\ \hline\hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} \hat{L}_{00} & 0 & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} & 0 \\ \hline\hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right)$</td></tr>
<tr><td>2</td><td>$\left\{ \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c} L_{TL}^{-1} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \right\}$</td></tr>
<tr><td></td><td>enddo</td></tr>
<tr><td>2,3</td><td>$\left\{ \left(\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c} L_{TL}^{-1} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \right) \wedge \neg\,(\neg\text{SameSize}(L, L_{TL})) \right\}$</td></tr>
<tr><td>1b</td><td>$\left\{ L = \hat{L}^{-1} \right\}$</td></tr>
</table>

Fig. 2. Worksheet for developing an algorithm for symmetric matrix multiplication.

Now, this partitioned matrix is substituted into the postcondition after which algebraic manipulation expresses the desired final contents of the quadrants in terms of operation on the original contents of those quadrants:

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c} \hat{L}_{TL} & 0 \\ \hline\hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right)^{-1} = \left(\begin{array}{c||c} \hat{L}_{TL}^{-1} & 0 \\ \hline\hline -\hat{L}_{BR}^{-1} \hat{L}_{BL} \hat{L}_{TL}^{-1} & \hat{L}_{BR}^{-1} \end{array}\right)$$

At an intermediate stage (at the top of the loop-body) only some of the operations will have been performed. For example, the intermediate state

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c} \hat{L}_{TL}^{-1} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right)$$

comes from assuming that only L_{TL} has been updated with the final result while the other parts of the matrix have not yet been touched. Let us use this example for the remainder of this discussion: it becomes P_{inv} in the worksheet in Fig. 1 as illustrated in Fig. 2.

Step 3: Determine loop-guard G. We are assuming that after the loop completes, $P_{\text{inv}} \wedge \neg G$ holds. Thus, by choosing a loop-guard G such that $(P_{\text{inv}} \wedge \neg G) \Rightarrow P_{\text{post}}$, it is guaranteed that the loop completes in a state that implies that the desired result has been computed. Notice that when L_{TL} equals all of L,

$$\left(\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c||c} \hat{L}_{TL}^{-1} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right) \wedge \text{SameSize}(L, L_{TL})\right) \Rightarrow (L = \hat{L}^{-1})$$

Here the predicate $\text{SameSize}(L, L_{TL})$ is *true* iff the dimensions of L and L_{TL} are equal. Thus, the iteration should continue as long as $\neg\text{SameSize}(L, L_{TL})$, the loop-guard G in the worksheet.

Step 4: Determine the initialization. The loop-invariant must hold before entering the loop. Ideally, only the partitioning of operands is required to attain this state. Notice that the initial partitionings given in Step 4 of Fig. 2 result in an L that contains the desired contents, without requiring any update to the contents of L.

Step 5: Determine how to move boundaries. Realize that as part of the initialization L_{TL} is 0×0, while upon completion of the loop this part of the matrix should correspond to the complete matrix. Thus, the boundaries, denoted by the double lines, must be moved forward as part of the body of the loop, adding rows and columns to L_{TL}. The approach is to identify parts of the matrix that must be moved between regions at the top of the loop body, and adds them to the appropriate regions at the bottom of the loop body, as illustrated in Steps 5a and 5b in Fig. 2.

Step 6: Determine P_{before}. Notice that the loop-invariant is *true* at the top of the loop body, and is thus *true* after the repartitioning that identifies parts of the matrices to be moved between regions. In Step 6 in Fig. 2 the state, in terms of the repartitioned matrices, is given.

Step 7: Determine P_{after}. Notice that after the regions have been redefined, (as in Step 5b in Fig. 2), the loop-invariant must again be *true*. Given the redefinition of the regions in Step 5b, the loop-invariant, with the appropriate substitution of what the regions will become, must be *true* after the movement of the double lines. Thus,

$$\left(\begin{array}{c||c} \hat{L}_{TL}^{-1} & 0 \\ \hline \hat{L}_{BL} & \hat{L}_{BR} \end{array}\right) = \left(\begin{array}{c||c} \left(\begin{array}{c|c} \hat{L}_{00} & 0 \\ \hline \hat{L}_{10} & \hat{L}_{11} \end{array}\right)^{-1} & 0 \\ \hline\hline \left(\begin{array}{c|c} \hat{L}_{20} & \hat{L}_{21} \end{array}\right) & \hat{L}_{22} \end{array}\right) = \left(\begin{array}{c|c||c} \hat{L}_{00}^{-1} & 0 & 0 \\ \hline -\hat{L}_{11}^{-1}\hat{L}_{10}\hat{L}_{00}^{-1} & \hat{L}_{11}^{-1} & 0 \\ \hline\hline \hat{L}_{20} & \hat{L}_{21} & \hat{L}_{22} \end{array}\right)$$

must be *true* after the movement of the double lines.

Partition $L = \left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right)$

where L_{TL} is 0×0

while $\neg\text{SameSize}(L, L_{TL})$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & 0 & 0 \\ \hline\hline L_{10} & L_{11} & 0 \\ \hline L_{20} & L_{21} & L_{22} \end{array} \right)$$

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

$L_{10} := -L_{11}^{-1} L_{10} L_{00}$

$L_{11} := L_{11}^{-1}$

Continue with

$$\left(\begin{array}{c||c} L_{TL} & 0 \\ \hline\hline L_{BL} & L_{BR} \end{array} \right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & 0 & 0 \\ \hline L_{10} & L_{11} & 0 \\ \hline\hline L_{20} & L_{21} & L_{22} \end{array} \right)$$

enddo

Fig. 3. Final algorithm.

Step 8: Determine the update S_U. By comparing the state in Step 6 with the desired state in Step 7, the required update, given in Step 8, can be easily determined.

Final algorithm. Finally, by noting that $\hat{L}$ was introduced only to denote the original contents of L and is never referenced in the update, the algorithm for computing $L := L^{-1}$ can be stated as in Fig. 3.

Note that the second operation in update S_U requires itself an inversion of a triangular matrix. When $b = 1$, this becomes an inversion of the scalar L_{11}. Thus, the so-called "blocked" version of the algorithm, where $b > 1$, could be implemented by calling an "unblocked" version, where $b = 1$ and $L_{11} := L_{11}^{-1}$ is implemented by an inversion of a scalar.

Alternative algorithms. The steps we just described allow one to derive alternative algorithms: by applying Steps 3-8 with different loop-invariants one can obtain different variants for the same operation.

4 Implementation and Performance

In order to translate the proven correct algorithm into code, we have introduced APIs for the Mscript [10], C, and Fortran programming languages. The C code is illustrated in Fig. 4. The APIs were designed to mirror the algorithms as obtained from the worksheet. This allows for a rapid and direct translation to code, reducing chances of coding errors.

Performance attained on an SMP system based on the Intel Itanium2 (R) processor is given in Fig. 5. While the LAPACK implementation uses the algorithm given in Fig. 3, the FLAME uses a different algorithmic variant that is rich in rank-k updates,

```
FLA_Part_2x2( L,    &LTL, &LTR,
                    &LBL, &LBR,     0, 0, FLA_TL );

while ( FLA_Obj_length( LTL ) != FLA_Obj_length( L ) ){
  b = min( FLA_Obj_length( LBR ), nb_alg );
  FLA_Repart_2x2_to_3x3(
        LTL, /**/ LTR,       &L00, /**/ &L01, &L02,
     /* ************* */   /* ******************* */
                             &L10, /**/ &L11, &L12,
        LBL, /**/ LBR,       &L20, /**/ &L21, &L22,
        b, b, FLA_BR );
  /*------------------------------------------------*/

  FLA_Trsm(FLA_LEFT, FLA_LOWER_TRIANGULAR,
           FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
           MINUS_ONE, L11, L10);

  FLA_Trmm(FLA_RIGHT, FLA_LOWER_TRIANGULAR,
           FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
           ONE, L00, L10);

  FLA_TriInv( L11 );

  /*------------------------------------------------*/
  FLA_Cont_with_3x3_to_2x2(
        &LTL, /**/ &LTR,       L00, L01, /**/ L02,
                               L10, L11, /**/ L12,
     /* ************** */  /* ***************** */
        &LBL, /**/ &LBR,       L20, L21, /**/ L22,
        FLA_TL );
}
```

Fig. 4. C implementation.

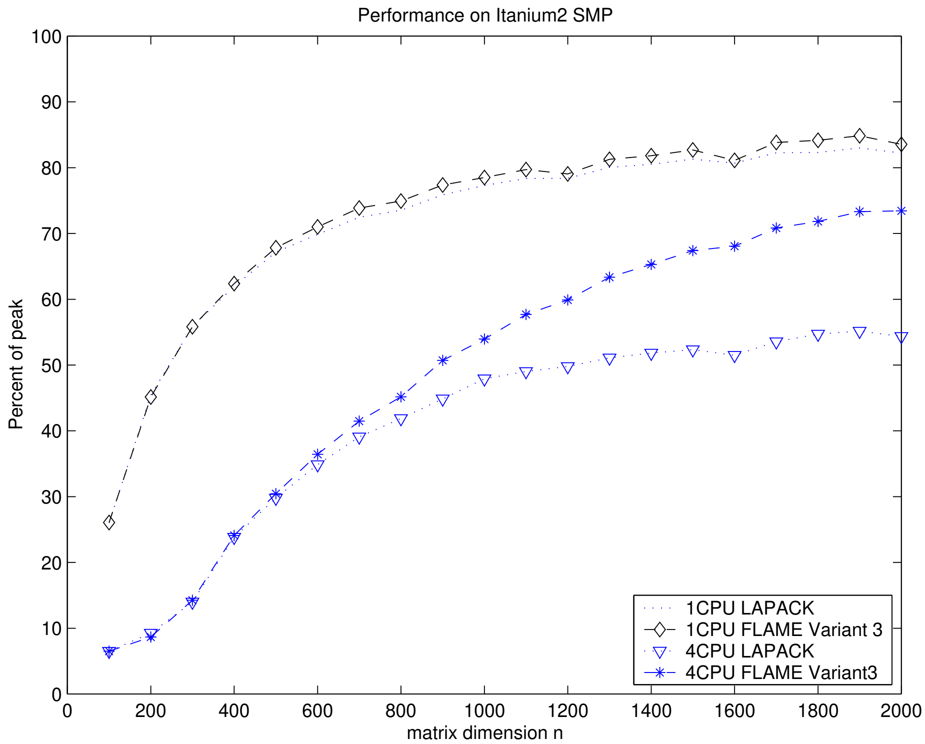

Fig. 5. Performance of LAPACK vs. FLAME on a single CPU and four CPUs of an Itanium2-based SMP. Here Variant3 indicates a different loop invariant which results in an algorithm rich in rank-k updates. LAPACK's variant is instead rich in triangular solves with multiple right hand sides (TRSM), where one of the operands is a long and skinny panel. It is known that the rank-k update is an operation that parallelizes better than the TRSM. This explain the better peformance of Variant3 with respoect to LAPACK.

which parallelize better with OpenMP. For this experiment, both libraries were linked to BLAS implemented by Kazushige Goto [6].

5 Conclusion

We presented a methodolody for rapidly deriving and implementing algorithms for linear algebra operations. The techniques in this paper apply to operations for which there are algorithms that consist of a simple initialization followed by a loop. While this may appear to be extremely restrictive, the linear algebra libraries community has made tremendous strides towards modularity. As a consequence, almost any operation can be decomposed into operations (linear algebra building blocks) that, on the one hand, are themselves meaningful linear algebra operations and, on the other hand, whose algorithms have the structure given by the algorithm in Fig. 1.

The derivation of algorithms is dictated by 8 steps, while the implementation is a direct translation of the algorithm thanks to a number of API's that we developed. Using PLAPACK [12], a C library based on MPI, even a parallel implementation for a distributed memory architecture closely mirrors the algorithm as represented in Fig. 3 and is no different from the C code shown in Fig. 4.

One final comment about the 8 steps necessary to fill the worksheet in: the process is so systematic that we were able to develop a semi-automated system capable of generating one algorithm starting from a loop invariant. In the paper *Automatic Derivation of Linear Algebra Algorithms with Application to Control Theory*, also presented at this conference, we show how to use the system to solve the Triangular Sylvester Equation.

Additional information

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

Acknowledgments

This research is partially supported by NSF grants ACI-0305163 and CCF-0342369. We thank NEC Solutions (America), Inc. for access to the Intel Itanium2 processor SMP used for the performance experiments.

References

1. 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*. SIAM, Philadelphia, 1992.
2. 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*. to appear.
3. Paolo Bientinesi, Enrique S. Quintana-Orti, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME APIs. *ACM Transactions on Mathematical Software*. To appear.
4. Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.
5. Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.
6. Kazushige Goto and Robert A. van de Geijn. On reducing tlb misses in matrix multiplication. Technical Report CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin, 2002.
7. John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

8. Isak Jonsson. *Recursive Blocked Algorithms, Data Structures, and High-Performance Software for Solving Linear Systems and Matrix Equations*. PhD thesis, Dept. Computing Science, Umeå University, SE-901 87, Sweden., 2003.
9. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.
10. C. Moler, J. Little, and S. Bangert. *Pro-Matlab, User's Guide*. The Mathworks, Inc., 1987.
11. Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Transactions on Mathematical Software*, 29(2):218–243, June 2003.
12. Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.