Aachen Institute for Advanced Study in Computational Engineering Science

Preprint: AICES-2008-2

17/October/2008

Automation in Dense Linear Algebra

P. Bientinesi and R. van de Geijn

RWTH AACHEN UNIVERSITY

AICES

Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged.

List of AICES technical reports: http://www.aices.rwth-aachen.de/preprints

Automation in Dense Linear Algebra

Paolo Bientinesi[1] and Robert van de Geijn[2]

[1] AICES, RWTH Aachen University, Aachen, Germany,
pauldj@aices.rwth-aachen.de
[2] Department of Computer Sciences,
The University of Texas at Austin, Austin, TX.

Abstract. In this article we look at the generation of libraries for dense linear algebra operations from a different perspective: instead of focusing on the optimization (possibly automatically) of a routine, we address the question "what would it take for a computer to mechanically (automatically) generate high-performance algorithms, much like a human expert?". We will show that for a large class of operations, the mathematical description of the input and output variables represents the necessary and sufficient information for a symbolic system to generate loop-based algorithms as well as high-performance routines. Surprisingly, the generation process is entirely prescribed by a proof of correctness: taking Dijkstra's advice, rather than starting from an algorithm and trying to prove its correctness, we systematically build one so that its correctness is guaranteed. This methodology is thus fundamentally different with respect to a standard auto-tuning approach: while auto-tuning often performs a parameter optimization over a large search space, our methodology and system rely on symbolic and algebraic transformations, and in principle, can be used to produce cost and error analyses hand in hand with the generated algorithms.

1 Introduction

Dense linear algebra libraries provide the building blocks for a wide range of scientific applications. Even applications that yield sparse linear systems often require the manipulation of smaller dense subproblems. Consequently, these libraries are needed for each and every computing architecture, and undergo a constant process of revision and upgrade. It is more than ten years that automation has been proposed as a means to both improve the quality, and decrease the development effort, of such libraries [6, 11]. In practice, the term automatic is often used as synonym for optimization: given a routine and a target architecture, implementation parameters—such as block size and unrolling factor—are tuned over a search space by generating, executing and timing a massive number of different implementations.

In a contrasting definition, automation is used to indicate that the decision process leading to the solution of a target problem is so systematic that can be reproduced by a mechanical system without human intervention. With respect to libraries for dense linear algebra, we asked ourselves whether it is possible to

have a computer generate them, and what would be the necessary and sufficient knowledge for this to happen. In this paper we demonstrate that for a class of dense linear algebra operations, it is possible to transform the mathematical description of the input and output variables a target operation into a family of loop-based algorithms and into high-performance routines. Since these transformation steps can be performed by a computer algebra system like Mathematica, then automation is achieved.

We show that the process for generating loop-based algorithms and routines is entirely dictated by a proof of correctness. By adopting a formal derivation approach, we transform the description of the target operation into constraints that must be satisfied at different stages of the computation. The statements constituting an algorithm are then constructed so that they satisfy such constraints. The key discovery was that if we limit the structure of the algorithm to contain an initialization block followed by a loop, then for many operations it is possible to identify –*systematically* and *a priori*– one or more loop-invariants for the loop [1]. Once a loop-invariant is available, the corresponding loop can be obtained through symbolic and algebraic manipulations, i.e., automatically.

The focus of this paper is on the formal derivation techniques that lead to the automatic derivation of loop-based algorithms. Loop-based algorithms are preferred to purely recursive ones as they allow the users to carefully select the size of the submatrices involved in each iteration of the loop. This size, in turn, directly affects the amount of data movement and the performance of the calls to the BLAS library. Although not directly addressed in the derivation process, performance comes out as a byproduct of the fact that our methodology returns not one, but a family of loop-based algorithms for each target operation; such a richness is necessary as it is often observed that the best algorithm, in terms of performance, varies depending on factors like target architecture, parallelism, problem size and even underlying BLAS [4].

A separate note is in place for multi-core processors. The simplest way to exploit the computational power of these architectures is by linking the routines to multi-threaded BLAS. This solution leads to high-performance for problems from medium to very large size as it decomposes the computation into large calls to highly optimized library routines. Recently it has been shown that for small and medium sized problems a different approach yields higher performance than multi-threaded routines: the execution of out of order algorithms-by-blocks [5]. Because of space limitation we only state here that out methodology encompasses the automatic generation of this type of routines.

The rest of the paper is organized as follows. Section 2 introduces the concept of formal correctness and describes how a proof of correctness can be exploited to guide the generation of algorithms. In Section 3 we uncover the actual derivation of algorithms and we present results from a prototype symbolic system. In Section 4 we we draw conclusions.

2 Automation from Formal Correctness

In this section we show how a symbolic system can transform the mathematical definition of a target linear algebra operation into a family of loop-based algorithms and/or routines. Key to this result are concepts from classical computer science, such as formal correctness and goal-oriented programming. In order simplify the exposition, we will carry out the derivation of algorithms for the Cholesky factorization as an example.

Input. The initial information required for the generation of algorithms is the mathematical definition of a target operation $\mathcal{O}p$. This is specified by means of two predicates, the *Precondition* (P_{pre}) and the *Postcondition* (P_{post}). P_{pre} describes the domain, the dimensions and the properties of the input and output operands before the execution of $\mathcal{O}p$, while P_{post} defines the relations involving the input and output operands that hold *true* upon completion of the operation.

For the Cholesky factorization $L := \Gamma(A)$, P_{pre} and P_{post} are defined as

$$P_{\text{pre}} : \{\ m(A) = n(A) = m(L) = n(L)\ \wedge\ \text{SPD}(A)\ \wedge \\ \text{LowerTriangular}(L)\ \wedge\ \text{Unknown}(L)\ \},\ \text{and} \tag{1}$$

$$P_{\text{post}} : \{\ LL^T = A\ \}, \tag{2}$$

and they represent a description of the input matrix A and the output matrix L. In these definitions, the functions $m(Z)$ and $n(Z)$ return the number of rows and columns of matrix Z, the predicate SPD(Z) is *true* iff Z is a symmetric positive definite matrix, LowerTriangular(Z) returns *true* iff Z is a square lower triangular matrix, and predicate Unknown(Z) indicates that the matrix Z is an output variable.

Partitioned Matrix Expression (PME). The PME is a predicate that establishes how different submatrices of the output matrices for $\mathcal{O}p$ can be expressed in terms of submatrices of the input operands. This predicate, in conjunction with P_{pre} and P_{post} contains the sufficient and necessary information to generate algorithms. If an operation does not admit PME, then the following methodology is not applicable.

For a given operation, the PME is not unique. For many operations the PMEs can be directly derived from the precondition and postcondition, solely by means of partitioning and pattern matching, i.e., mechanically. For other operations, like the LU factorization with pivoting, the PMEs are still derived from P_{pre} and P_{post}, but not exclusively through symbolic transformations. In these cases, it is more challenging to achieve automation, as ad-hoc logic rules are to be deployed. Depending on the target operation, one or more PMEs may be provided by the user.

In general, the derivation of loop-based algorithms for an operation $\mathcal{O}p$ requires knowledge of more PMEs than just those for $\mathcal{O}p$. Consequently, the PMEs are not arguments to be passed as input to the symbolic system. Instead, they

constitute a library of definitions, in the form of rewrite rules, that specify how the computation of an operation can be decomposed into simpler operations. A symbolic system will deduce these pieces of information either from the input (P_{pre} and P_{post}), or by accessing a database of pre-computed PMEs.

The PME for the Cholesky factorization is easily obtained by partitioning the input matrix A in the postcondition, expanding, and through pattern matching. The result is

$$\text{PME}: \left\{ \left(\begin{array}{c|c} L_{TL} & 0 \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c|c} \Gamma(A_{TL}) & 0 \\ \hline A_{BL}L_{TL}^{-T} & \Gamma(A_{BR} - L_{BL}L_{BL}^T) \end{array}\right) \ \wedge \tag{3}$$

$$Size(L_{TL}) = Size(A_{TL}) = k \times k,\ k \in [0, m(L)] \Big\},$$

where the subscript letters T, B, L and R indicate a matrix partitioning, and signify Top, Bottom, Left and Right, respectively. This predicate indicates that independently of the partitioning size, the Top Left quadrant of the solution matrix L contains the Cholesky factor of the corresponding quadrant of matrix A: $L_{TL} = \Gamma(A_{TL})$. Similarly, the Bottom Left and Bottom Right quadrants of L contain respectively $A_{BL}L_{TL}^{-T}$ and the Cholesky factor of matrix $A_{BR} - L_{BL}L_{BL}^T$. These three relations have to be satisfied at the same time for a matrix L to be the Cholesky factor of A.

Not only does the PME carry information on how to decompose an operation into suboperations, but it also contains information on dependencies among them. Such dependencies can be easily detected through pattern-matching. Looking again at the PME (3), quantity L_{TL} is function of A_{TL} only, therefore can be directly computed. Quadrant L_{BL} depends on A_{BL} as well as L_{TL}, so it should be computed only once L_{TL} is available; likewise, L_{BR} is function of A_{BR} and L_{BL}. These considerations lead to the dependency chain $L_{TL} \rightarrow L_{BL} \rightarrow L_{BR}$, where one quantity in the chain should be computed only once all the preceding quantities have been computed.

Formal Correctness of Algorithms. Once the predicates P_{pre}, P_{post} and PME are available, we can set up a proof of correctness that the loop-based algorithms for $\mathcal{O}p$ will have to satisfy. In E.W.Dijkstra's words: "[...] the programmer should let correctness proof and program grow hand in hand. [...] If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance." [7].

Hoare triples are the central tool of the Floyd-Hoare logic [8, 10], a formal system for reasoning about the correctness of computer programs. A triple describes how the execution of a section of code changes the state of the computation. A Hoare triple is of the form $\{P\}\ C\ \{Q\}$ where P and Q are predicates—precondition and postcondition, respectively—and C is a command. Such a triple is read as: whenever the command C is executed in a state in which P holds, it will terminate and Q holds upon completion. In our settings, knowing the

predicates P_{pre} and P_{post}, the goal is to construct an algorithm $\mathcal{A}$ such that the Hoare's triple

$$\{P_{\text{pre}}\}\ \mathcal{A}\ \{P_{\text{post}}\} \tag{4}$$

is satisfied. If such an algorithm $\mathcal{A}$ can be found, then it will correctly compute the operation $\mathcal{O}p$, by construction.

The Skeleton of a Proof of Correctness. In seeking algorithms for $\mathcal{O}p$, we restrict our search to algorithms that consist of a simple initialization followed by a loop. Expression (4) can be rewritten to take into account that $\mathcal{A}$ has the structure $\mathcal{I}nit; \mathcal{A}2$, where $\mathcal{I}nit$ represents an initialization block and $\mathcal{A}2$ is a loop:

$$\{P_{\text{pre}}\}\ \mathcal{I}nit;\ \mathcal{A}2\ \{P_{\text{post}}\}.$$

The Floyd-Hoare logic tells us that this triple is satisfied if the following two triples are in turn satisfied for a suitable state Q:

$$\{P_{\text{pre}}\}\ \mathcal{I}nit\ \{Q\}, \qquad \{Q\}\ \mathcal{A}2\ \{P_{\text{post}}\}.$$

Substituting $\mathcal{A}2$ with the explicit loop structure **While** G **do** $\mathcal{B}ody$ **end**, where G is the loop-guard and $\mathcal{B}ody$ denotes the computation executed at each iteration of the loop, we obtain 5the triples

$$\{P_{\text{pre}}\}\ \mathcal{I}nit\ \{Q\}, \qquad \{Q\}\ \textbf{While}\ G\ \textbf{do}\ \mathcal{B}ody\ \textbf{end}\ \{P_{\text{post}}\}. \tag{5}$$

In order to fix the predicate Q, we exploit the "Fundamental Invariance Theorem for Loops". This theorem refers to an assertion P_{inv} that holds before and after each iteration of loop-based algorithms. Such a predicate is called a *loop-invariant*. Here we paraphrase the theorem from Gries and Schneider' book *A Logical Approach to Discrete Math* [9]:

> (12.43) **Fundamental Invariance Theorem.** Suppose
> 1. $\{P_{\text{inv}} \wedge G\}\ S\ \{P_{\text{inv}}\}$ holds – i.e., execution of S begun in a state in which P_{inv} and G are *true* terminates with P_{inv} *true* – and
> 2. $\{P_{\text{inv}}\}$ **while** G **do** S **end** $\{true\}$ – i.e., execution of the loop begun in a state in which P_{inv} is *true* terminates.
>
> Then $\{P_{\text{inv}}\}$ **while** G **do** S **end** $\{P_{\text{inv}} \wedge \neg G\}$ holds. In other words, if the loop is entered in a state where P_{inv} is *true*, it will complete in a state where P_{inv} is *true* and guard G is *false*.

The theorem tells us that if we could 1) discover a loop-invariant P_{inv} for our loop $\mathcal{A}2$, and 2) prove the termination of such a loop, then the following Hoare's triple would hold *true*:

$$\{P_{\text{inv}}\}\ \textbf{While}\ G\ \textbf{do}\ \mathcal{B}ody\ \textbf{end}\ \{P_{\text{inv}} \wedge \neg G\}. \tag{6}$$

A comparison between Expressions (5), our goal, and (6), the theorem's thesis, leads us to the following consideration: if the predicate $\{P_{\text{inv}} \wedge \neg G\}$ implies

the postcondition P_{post}, then the natural choice for the state Q is P_{inv} (note that a loop-invariant is *true* before the loop is ever entered). The triples (5) can now be rewritten as

$$\{P_{\text{pre}}\}\ \mathit{Init}\ \{P_{\text{inv}}\}, \qquad \{P_{\text{inv}}\}\ \textbf{While}\ G\ \textbf{do}\ \mathit{Body}\ \textbf{end}\ \{\neg G \wedge P_{\text{inv}}\} \tag{7}$$

under the following assumptions:

1. The predicate P_{inv} is a loop-invariant for loop $\mathcal{A}2$;
2. The loop $\mathcal{A}2$, when executed in a state in which P_{inv} is *true*, terminates;
3. The predicate $\{\neg G \wedge P_{\text{inv}}\}$ implies P_{post}.

We have thus reduced the problem of building correct loop-based algorithms $\mathcal{A}$ that compute $\mathcal{O}p$ to the the problem of finding a predicate P_{inv} and statements Init, G and Body that satisfy the above assumptions and the triples (7).

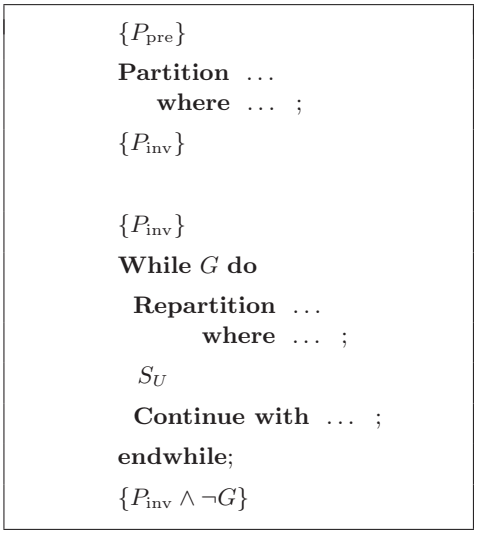
3 Derivation of Algorithms

In this section we outline the steps necessary to derive systematically predicate P_{inv} and statements Init, G and Body from the predicates P_{pre}, P_{post} and PME.

We begin by enforcing the termination of the loop $\mathcal{A}2$. To this end we require 1) the loop-guard G to measure the advancement in the traversal of the operands, and 2) the loop-body to include statements to make progress towards traversing the operands. Specifically, we fix the structure of Body: we impose that each iteration consist of a first stage in which one or more **Repartition** statements expose new submatrices and subvectors of the operands, a second stage of numerical computations (S_U) and a third and last stage in which **Continue with** statements re-assemble submatrices and subvectors to guarantee progress. In addition, we also require the Init to only consist of simple assignments and partitioning statements. Figure 1 (left) shows the structure of the two Hoare triples (7) in light of these new constraints.

A Worksheet for Deriving Algorithms. Figure 1 (right) contains a generic "worksheet" for deriving linear algebra algorithms of the form specified in Fig. 1 (left). The gray-shaded boxes contain statements that would appear in actual code (Steps 3, 4, 5, 8), while the white boxes contain predicates expressing the status of input/output variables at different stages in the algorithm (Steps 1a, 2, 2,4, 6, 7, 1b). The numbers in the "Step" column refer to the order in which the worksheet is filled out to generate algorithms.

Step 1 requires no work, as the predicates P_{pre} and P_{post} represent the input to our methodology. Step 2 concerns with the identification of viable loop-invariants. This piece of information will allow us to select the initialization block, Step 3, and the loop guard, Step 4. The loop guard, in turn, allows us to determine how the input and output operands are traversed in the algorithm, corresponding to Steps 5a and 5b. These statements, which only consist of repartitioning (indexing) operations, provide us with the tools to express the



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

Fig. 1. Left: template for a formal proof of correctness for algorithms consisting of an initialization step followed by a loop. Right: worksheet for developing algorithms.

loop-invariant before the execution of any computation, Step 6, and right after the execution of the numerical computation, Step 7. Finally, by comparing these two predicates we will identify the computational statements in Step 8.

Figure 2 shows a complete worksheet, filled out to generate one algorithm for the Cholesky factorization.

Loop-Invariants (Step 2). The loop-invariant is a predicate that expresses the state of the input and output variables during the execution of a loop. It has the property of being *true* at the top and bottom of each loop iteration. If we are able to find a loop-invariant for an algorithm that computes $\mathcal{O}p$, then the first hypothesis of the Fundamental Invariance Theorem [9] is satisfied:
$\{P_{\text{inv}} \wedge G\}\ S\ \{P_{\text{inv}}\}$.

Different loop-invariants are derived from the PME by considering individual operations that contribute to the final result. Each such operation may or may not have been performed at an intermediate stage of the computation: a loop-invariant is a subset of the operations appearing in the PME. On the other hand, not every subset of the operations from the PME yields a feasible loop-invariant. We have already seen that the PME carries information about dependencies among quadrants: one condition for a subset to be feasible is that it must satisfy the dependency chains. A mechanical system can build the dependency chain and keep track of the dimensions for each PME subexpression, thus discarding infeasible choices of loop-invariants.[3]

[3] A loop-invariant may result to be infeasible even if it satisfies the dependency chain.

The following table contains three loop-invariants for the Cholesky factorization, corresponding to three choices of subsets of the operations appearing in the PME. For the rest of this section we focus on the derivation of the algorithm corresponding to loop-invariant #1.

#	Loop-invariants for Cholesky Factorization
1	$\left(\begin{array}{c\|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = 0 & L_{BR} = 0 \end{array}\right)$
2	$\left(\begin{array}{c\|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL} L_{TL}^{-T} & L_{BR} = 0 \end{array}\right)$
3	$\left(\begin{array}{c\|c} L_{TL} = \Gamma(A_{TL}) & * \\ \hline L_{BL} = A_{BL} L_{TL}^{-T} & L_{BR} = A_{BR} - L_{BL} L_{BL}^{T} \end{array}\right)$

Initialization (Step 3). Since the loop-invariant holds *true* before the loop commences, the initialization $\mathcal{I}nit$, Step 3 in Fig. 1, must have the property that when executed in the state P_{pre}, sets the variables to a state in which P_{inv} holds: $\{P_{\text{pre}}\}\ \mathcal{I}nit\ \{P_{\text{inv}}\}$. We require the initialization to consist of a (possibly empty) list of statements to partition input and output operands and/or to set them to specific values.

Since the derivation methodology requires no computation to be performed at this stage, the initialization reduces to partitioning some or all the variables (from P_{pre}) in such a way that, for each variable that is partitioned, at least one of the submatrices (vectors are a special case of matrices) is null.[4] A mechanical system can exhaustively try out all the possible partitionings (conformally to the matrix properties describes in P_{pre}), selecting the ones that render *true* the implication $P_{\text{pre}} \implies P_{\text{inv}}$. If no such partitioning is found, the loop-invariant is labelled as infeasible and no further steps are executed.

In our example, the initialization that makes the loop invariant *true* is

$$L = \left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right), A = \left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right), \text{ with } L_{TL} \text{ and } A_{TL} \text{ empty matrices.}$$

Loop Guard (Step 4). The loop-guard G is the condition under which the program enters the loop; conversely, when the loop completes, $\neg G$ is *true*. If the loop terminates, then the second hypothesis of the Fundamental Invariance Theorem is satisfied too, hence the thesis of the theorem can be asserted: $(P_{\text{inv}} \wedge \neg G)$ holds *true* after the loop. Note that if $(P_{\text{inv}} \wedge \neg G)$ implies P_{post} then we can conclude that, upon termination, the loop correctly computes P_{post}. An appropriate loop-guard can be derived by comparing the loop-invariant P_{inv} with the postcondition P_{post}.

[4] A matrix is null if one or both its dimensions are zero.

The variables that have been partitioned as part of the initialization contain at least one null submatrix. At each iteration of the loop, one of the empty submatrices is expanded, until it encompasses the entire matrix. The loop-guard is given by a predicate of the form "the null submatrix Z_{xy} is not equal in size to the entire matrix Z;" the submatrix xy can be found by a system, testing exhaustively all the alternatives and selecting one for which the implication $P_{\text{inv}} \wedge \neg G \implies P_{\text{post}}$ is *true*. If no such partitioning is found, the loop-invariant is labelled as infeasible and no further steps are executed.

The loop guard for the Cholesky factorization is $\neg\text{SameSize}(L, L_{TL})$.

Traversing the Operands (Steps 5a & 5b). Step 5a and 5b are responsible to guarantee termination of the loop by traversing the operands. Step 5a exposes new regions of the operands and Step 5b re-assembles the regions so that progress is achieved. In terms of a Hoare triple, these two statements are chosen so that $\{P_{\text{inv}} \wedge G\}$ **while G do** Step 5a; Step 5b; **end** $\{true\}$ is satisfied. No actual computation happens at these stages, they merely represent indexing operations.

How to traverse through the variables follows directly from the initialization and the loop-guard. The **Repartition** and **Continue** statements are responsible to make progress towards making G *false* by increasing the size of the initially null matrix in G. Every other variable is then re-partitioned conformally. Furthermore, operands with a particular structure (triangular, symmetric, diagonal) can only be partitioned and traversed in a way that preserves the structure. This analysis can be made mechanically.

The **Repartition** statements for the Cholesky example are

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

where L_{11} and A_{11} are $b \times b$

and the **Continue** statements are

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

From Loop-Invariant to Algorithm (Steps 6–8). With respect to the skeleton in Fig. 1 (left), we are left to determine the computational statements S_U. We know that for each iteration it holds

$$\{P_{\text{inv}}\}\ \textbf{Repartition}\ldots;\ S_U;\ \textbf{Continue}\ldots\ \{P_{\text{inv}}\},$$

where the **Repartition** and **Continue** statements have already been determined. Thus it is possible to compute states P_{before} and P_{after} to satisfy the chain of triples

$$\{P_{\text{inv}}\}\ \textbf{Repartition}\ldots\ \{P_{\text{before}}\}\ S_U\ \{P_{\text{after}}\}\ \textbf{Continue}\ldots\ \{P_{\text{inv}}\},$$

and finally S_U will be determined by comparing P_{before} and P_{after}.

P_{before}*:* This predicate corresponds to P_{inv} expressed in terms of newly exposed parts of the operands, right after the **Repartition** statements. The expression for P_{before} is computed by 1) applying the textual substitution rules dictated by **Repartition** to the P_{inv}, 2) expanding by means of the rewrite rules defined by the PMEs (and by common linear algebra identities) and, 3) simplifying. Performing textual substitution is straightforward, while the expansion and simplification of the expressions requires powerful symbolic computation tools.

In our example P_{before} becomes

$$\left(\begin{array}{c|c|c} L_{00} & & \\ \hline L_{10} & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \Gamma(A_{00}) & & \\ \hline 0 & 0 & \\ \hline 0 & 0 & 0 \end{array}\right).$$

P_{after}*:* Similarly to the previous step, this predicate only expresses P_{inv} in terms of partitioned operands. In particular, P_{after} equals P_{inv} before the execution of the **Continue** statements. The computation of this predicate is analogous to the computation of P_{before} except that the textual substitution rules are dictated by the **Continue** statements.

Predicate P_{after} for the Cholesky factorization is

$$\left(\begin{array}{c|c|c} L_{00} & & \\ \hline L_{10} & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c|c|c} \Gamma(A_{00}) & & \\ \hline A_{10}L_{00}^{-T} & \Gamma(A_{11} - L_{10}L_{10}^{T}) & \\ \hline 0 & 0 & 0 \end{array}\right).$$

S_U*:* The computational statements are chosen so that the triple $\{P_{\text{before}}\}\ \mathbf{S}_U\ \{P_{\text{after}}\}$ holds *true*. In other words, we have to identify a set of updates that transform the state P_{before} into P_{after}. The goal is achieved by a direct comparison of the two states. This step can be executed mechanically by a system with strong pattern matching capabilities.

From a comparison of the predicates P_{before} and P_{after} that we derived for the Cholesky factorization, we realize the following facts. The contents of submatrix L_{11} are the same before and after S_U, so no computation is required. This is not the case for the quadrants L_{10} and L_{11}, which need to be updated for the loop-invariant to be satisfied at the end of the iteration. L_{10} requires a triangular solve, and because of the dependency chain, this operation has to be performed before any update on L_{11} can happen. In fact, in order to compute L_{11}, one has to first update matrix A_{11} by means of the just computed L_{10}, and then solve a small Cholesky subproblem. The updates are listed here:

$$\begin{aligned} L_{10} &:= A_{10}\ L_{00}^{-T} && \text{(TRSM)} \\ L_{11} &:= A_{11} - L_{10}L_{10}^{T} && \text{(GEMM)} \\ L_{11} &:= \Gamma(L_{11}) && \text{(Cholesky)} \end{aligned}$$

8-step Procedure. We have demonstrated that there exists an 8-step procedure to fill out the worksheet in Fig. 1. The procedure transforms predicates P_{pre} and P_{post} into an algorithm—appearing in Steps 3, 4, 5 and 8—together with its proof of correctness—in Steps 1, 2, 2,4, 6, and 7. This procedure can be repeated as many times as the number of feasible loop-invariants, returning a family of distinct algorithms. A complete worksheet for loop-invariant #1 of the Cholesky factorization is shown in Fig. 2. The worksheet for a different loop-invariant would differ only in Steps 6–8.

Thanks to a set of APIs developed as part of the FLAME project [2], the same procedure is used to generate routines in a variety of languages, including Matlab, C and Fortran, and targeting both sequential and parallel architectures. Fig. 4 shows the Matlab implementation for the algorithm we just derived.

A Prototype System. Not every step in the 8-step procedure requires complicated calculations. Steps 2 through 5 are relatively simple, and often times coincide across many algorithms in the same family. The bulk of the complexity lies with Steps 6, 7 and 8, as they require decomposition (through PMEs), manipulation, simplification, and matching of symbolic expressions. Furthermore, these steps differ for each loop-invariant.

In order to facilitate the computation of P_{before}, P_{after} and S_U, we prototyped a mechanical system. The system provides an environment to perform symbolic operations with blocked matrices, and allows us to manipulate matrix functions by means of substitution rules, simplifications and pattern matching [3]. When fed with a loop-invariant, the system returns the corresponding loop-based algorithm, in the form of a filled worksheet or a Matlab or C routine.

Figures 3 and 4 illustrate the worksheet and a Matlab routine, respectively, as generated by the mechanical system for loop-invariant #1 of the Cholesky factorization. The graphical style of Fig. 3, in particular, is designed to resemble the worksheet in Fig. 2. The boxes labelled "loop invariant" correspond to Steps 2, 6 and 7 in the worksheet. The expression for P_{after} (Step 7) contains annotations to simplify the successive task of identifying computational updates (Step 8). Expressions with a gray background are quantities readily available in one of the operands and require no computation. Expressions in red, instead, indicate a dependency: computation is required, but it can be reused. The computational statements in Step 8 are ordered so that the dependencies are satisfied. In Fig. 3, for example, the update for submatrix L_{11} contains the submatrix L_{10}, highlighted in red. This means that the update for L_{10} should be computed first, and then the result used for computing L_{11}. The computational statements satisfy such an ordering.

Our prototype system has been applied to a number of problems. In some cases it acted as a sanity check against which to test results; examples are the derivation of algorithms for the Sylvester equation $AX + XB = C$, and the inversion of symmetric positive definite matrices. In many other cases it was used to tackle problems that otherwise could not have been solved by hand. For example, as part of the development of a full Level-3 BLAS library, the system

Step	$L := \Gamma(A)$
1a	$\{$ (see Predicate (1))$\}$
3	**Partition** $L = \left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right)$ and $A = \left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right)$ **where** L_{TL} and A_{TL} are 0×0
2	$\left\{ \left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \Gamma(A_{TL}) & \\ \hline 0 & 0 \end{array}\right) \right\}$
4	**While** $\neg$SameSize(L, L_{TL}) **do**
2,4	$\left\{ \left(\left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \Gamma(A_{TL}) & \\ \hline 0 & 0 \end{array}\right) \right) \wedge (\neg\text{SameSize}(L, L_{TL})) \right\}$
5a	**Repartition** $\left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} L_{00} & & \\ \hline L_{10} & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$ **where** L_{11} and A_{11} are $b \times b$
6	$\left\{ \left(\begin{array}{c\|c\|c} L_{00} & & \\ \hline L_{10} & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \Gamma(A_{00}) & & \\ \hline 0 & 0 & \\ \hline 0 & 0 & 0 \end{array}\right) \right\}$
8	$L_{10} := A_{10}\ L_{00}^{-T}$ (TRSM) $L_{11} := A_{11} - L_{10}L_{10}^T$ (GEMM) $L_{11} := \Gamma(L_{11})$ (Cholesky)
7	$\left\{ \left(\begin{array}{c\|c\|c} L_{00} & & \\ \hline L_{10} & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right) = \left(\begin{array}{c\|c\|c} \Gamma(A_{00}) & & \\ \hline A_{10}L_{00}^{-T} & \Gamma(A_{11} - L_{10}L_{10}^T) & \\ \hline 0 & 0 & 0 \end{array}\right) \right\}$
5b	**Continue with** $\left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} L_{00} & & \\ \hline L_{10} & L_{11} & \\ \hline L_{20} & L_{21} & L_{22} \end{array}\right), \left(\begin{array}{c\|c} A_{TL} & \star \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} A_{00} & \star & \star \\ \hline A_{10} & A_{11} & \star \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
2	$\left\{ \left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \Gamma(A_{TL}) & \\ \hline 0 & 0 \end{array}\right) \right\}$
	endwhile
2,4	$\left\{ \left(\left(\begin{array}{c\|c} L_{TL} & \\ \hline L_{BL} & L_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \Gamma(A_{TL}) & \\ \hline 0 & 0 \end{array}\right) \right) \wedge \neg\left(\neg\text{SameSize}(L, L_{TL})\right) \right\}$
1b	$\{L = \Gamma(A), \text{i.e., } LL^T = A\}$

Fig. 2. Worksheet completed for the computation of the Cholesky factor.

generated algorithms for each operation and for each parameter combination (transpose/no-transpose, upper/lower triangular, etc.), resulting in more than 300 mechanically derived algorithms. More examples come from control theory equations, on which our system has been extensively tested: the system generated dozens of algorithms for many such equations including the particularly challenging triangular coupled Sylvester equation; for this operation we found more than 50 algorithms, out of which only three were previously known.

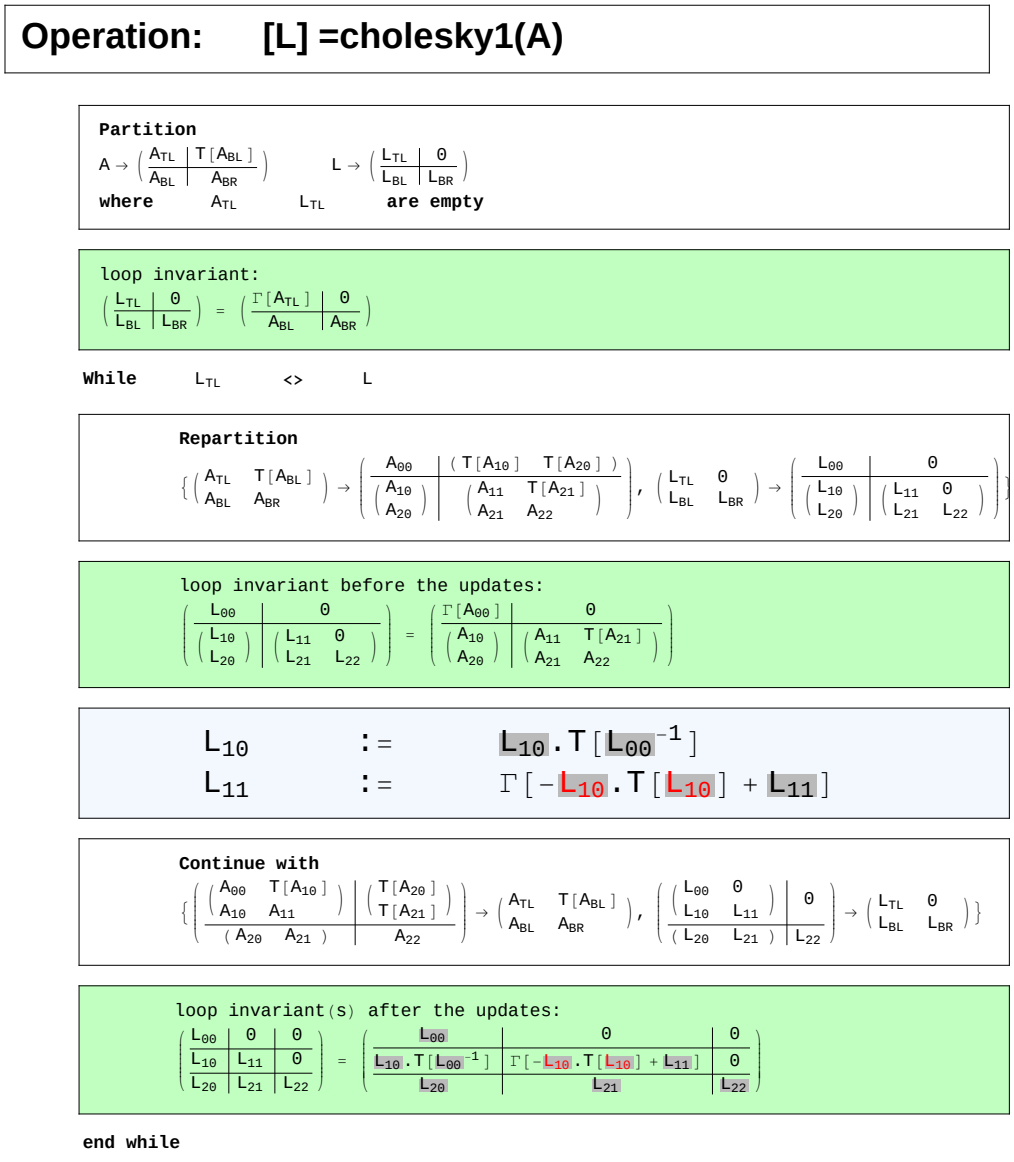


Fig. 3. Mechanically derived blocked algorithm for computing the Cholesky factor.

4 Conclusions

We have shown that for a class of linear algebra operations, the mathematical description of the operation—encoded in the predicates P_{pre}, P_{post} and PME—contains the necessary and sufficient information to automatically generate algo-

```
function [A] = choleskyL1( A , nb )

[ ATL, ATR, ...
  ABL, ABR ] = FLA_Part_2x2( A,0,0,'FLA_TL');

%% Loop Invariant
%% ATL=choleskyL[ATL]
%% ABL'=0
%% ABL=ABL
%% ABR=ABR

while( size(ATL,1) ~= size(A,1) | size(ATL,2) ~= size(A,2) )
   b = min( nb, min( size(ABR,1), size(ABR,2) ));

   [ A00, A01, A02, ...
     A10, A11, A12, ...
     A20, A21, A22 ] = FLA_Repart_2x2_to_3x3(ATL, ATR,...
                                             ABL, ABR,...
                                             b, b, 'FLA_BR');

   %* ************************************************** *%
   A10 = A10 . inv(A00)';
   A11 = choleskyL(A11 - A10 . A10');
   %* ************************************************** *%
   [ ATL, ATR, ...
     ABL, ABR ] = FLA_Cont_with_3x3_to_2x2(A00, A01, A02, ...
                                           A10, A11, A12, ...
                                           A20, A21, A22, ...
                                           'FLA_TL');
end;
A = ATL;
return;
```

Fig. 4. Mechanically generated Matlab routine for computing the Cholesky factor.

rithms. From this information it is in fact possible to identify, *a priori*, a family of loop-invariants for algorithms that compute a target operation. Using formal derivation techniques, we set up a proof of correctness that algorithms have to satisfy. The constraints set by the proof, together with the knowledge of a loop-invariant, fully dictate the steps of a derivation procedure that transforms P_{pre}, P_{post} and PME into algorithms and routines.

We also presented a prototype of a mechanical system that implements the most challenging steps of the aforementioned derivation procedure. The system takes a loop-invariant as input and returns a formally correct algorithm description or routine as output. Since we reasoned that loop-invariants can also be

mechanically derived from the operation specifications, we conclude that formally correct algorithms can be generated mechanically.

Performance is a direct consequence of the fact that our derivation procedure yields not one, but a family of algorithms. A variety of algorithms, together with a set of APIs for distributed and shared memory architectures, leads to high-performance in many different scenarios. In the case of multicore processors it has been shown that the algorithms-by-block parallel paradigm leads to high-performance. Our methodology naturally extends to this paradigm too.

Finally one comment on numerical stability. We have established that formally correct algorithms can be mechanically generated. Unfortunately, since we deal with floating point computations, formal correctness does not translate into accuracy. A stability analysis of every generated algorithm is needed. To this end we have extended the derivation worksheet and the derivation procedure to investigate numerical properties [3]. The analysis is made systematic and modular, and in principle can be automated.

References

1. Bientinesi, P. and Gunnels, J. and Myers, M. and Quintana-Ortí, E. and van de Geijn, R.: The Science of Deriving Dense Linear Algebra Algorithms. ACM TOMS, 31(1) (2005)
2. Bientinesi, P. and Quintana-Ortí, E. and van de Geijn, R.: Representing Linear Algebra Algorithms in Code: The FLAME Application Programming Interfaces. ACM TOMS, 31(1) (2005)
3. Bientinesi, P.: Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms. Ph.D. dissertation, The University of Texas, Department of Computer Sciences Technical Report TR-06-46 (2006)
4. Bientinesi, P. and Gunter, B. and van de Geijn, R.: Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix. ACM TOMS, 35(1) (2008)
5. Bientinesi, P. and Chan, E. and Quintana-Ortí, E. and Quintana-Ortí, G. and van de Geijn, R. and Van Zee, F.: SuperMatrix: a Multithreaded Runtime Scheduling System for Algorithms-by-Blocks. ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'08) (2008)
6. Bilmes, J. and Asanović, K. and Chin, C. and Demmel, J.: Optimizing Matrix Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology. Proc. International Conference on Supercomputing (1997)
7. Dijkstra, E. W.: The humble programmer (EWD340). Communications of the ACM 15, 10, 859–866 (1972)
8. Floyd, R. W.: Assigning meanings to programs. Symposium on Applied Mathematics 19, 19–32 (1967)
9. Gries, D. and Schneider, F. B.: A Logical Approach to Discrete Math. Springer Verlag (1992)
10. Hoare, C. A. R.: An axiomatic basis for computer programming. Communications of the ACM, Oct, 576–580 (1969)
11. Whaley, R. C. and Dongarra, J.: Automatically Tuned Linear Algebra Software. Supercomputing 1998: High Performance Networking and Computing, CD-ROM Proceedings (1998)