Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix *

Paolo Bientinesi
Brian Gunter
Robert van de Geijn
The University of Texas at Austin
Austin, TX 78712
{pauldj, rvdg}@cs.utexas.edu, gunter@csr.utexas.edu

FLAME Working Note #19

April 8, 2006

Abstract

We present families of algorithms for operations related to the computation of the inverse of a Symmetric Positive Definite (SPD) matrix: Cholesky factorization, inversion of a triangular matrix, multiplication of a triangular matrix by its transpose, and one-sweep inversion of an SPD matrix. These algorithms are systematically derived and implemented via the Formal Linear Algebra Methodology Environment (FLAME), an approach for developing linear algebra algorithms. How different members of these families of algorithms are more or less suited for a given architecture is demonstrated via implementations for sequential, shared-memory, and distributed memory parallel architectures. Performance on various platforms is reported.

1 Introduction

We demonstrate how to create a set of provably correct dense linear algebra algorithms to attain high performance for a variety of settings and architectures. Our derivation methods [17, 3] have successfully been used to derive linear algebra operations such as the level-3 BLAS [11] and LU factorization [17], in addition to the more complex solution of the triangular Sylvester equation [21, 4]. This paper applies the methods to the inverse of a Symmetric Positive Definite (SPD) matrix. Our implementations were developed for the computation of the covariance matrix of a linear least-squares problem. This operation has a particular value to the Earth science and aerospace communities, where the solution of large overdetermined dense linear systems is still a common procedure, and the statistics of the solution are often desired [18, 27, 23].

The inverse of an SPD matrix A, is typically obtained by first computing the upper triangular Cholesky factor R of A, $A = R^T R$, after which $A^{-1} = (R^T R)^{-1} = R^{-1}R^{-T}$ can be computed by first inverting the matrix R ($U = R^{-1}$) and then multiplying the result by its transpose ($A^{-1} = UU^T$). We will show that each of these three operations can be orchestrated so that the result overwrites the input without requiring temporary space. It will also be shown that A can be overwritten by its inverse without the

*This work was supported in part by NSF grants ACR-0203685, ACI-0305163, and CCF-0342369. 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.

explicit computation of these intermediate results, requiring only a single sweep through the matrix, as was already briefly mentioned in [22].

Another aspect of this work lies with the fact that different algorithmic variants of the same operation will perform differently based on the system architecture being used. Differences in memory configurations, BLAS implementations, compilers and many other factors may favor one variant over another. The ability to derive a suite of algorithms, and to then choose the most appropriate variant, allows the user to obtain improved performance for their particular application.

This paper makes the following contributions:

- It provides what we believe to be the most complete treatment to date of loop-based algorithms for computing the studied operations.
- It shows the benefit of using a single sweep algorithm for computing an operation when the same can be computed via several intermediate steps [22].
- It demonstrates that on different architectures different algorithmic variants achieve superior performance. This motivates the necessity for libraries like LAPACK [2], ScaLAPACK [7], FLAME, and PLAPACK [28, 1] to include multiple algorithmic variants for every supported operation.
- It highlights the benefits of deriving and implementing high-performance linear algebra algorithms via the Formal Linear Algebra Methodology Environment (FLAME) [17, 3, 5] approach.

The organization of the paper is as follows: Section 2 introduces the systematic approach for deriving algorithms by applying it to the example of the Cholesky factorization. Having demonstrated the method, Sections 3 and 4 illustrate how the same technique can be applied to the problem of the triangular matrix inversion and the multiplication of a triangular matrix by its transpose. Section 5 shows how the three individual operations of the SPD matrix inversion can be combined into a single-sweep algorithm using the same formal derivation methodology. Section 6 provides some brief remarks regarding the numerical stability of the derived algorithms. Section 7 highlights the performance of the newly derived algorithms. Section 8 gives a final summary and comments.

2 Cholesky Factorization: $A := \text{CHOL}(A)$

We shall use the Cholesky factorization to demonstrate the formal derivation approach. The Cholesky factorization of an SPD matrix A is given by $A = R^T R$ where R is upper triangular. This is the first step towards the inversion of an SPD matrix.

2.1 Traditional derivation

Before demonstrating the FLAME approach, let us review how one particular algorithm for the Cholesky factorization is usually motivated. Consider $A = R^T R$ and partition[1]

$$A = \left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline \star & \alpha_{11} \end{array}\right) \quad \text{and} \quad R = \left(\begin{array}{c|c} R_{00} & r_{01} \\ \hline 0 & \rho_{11} \end{array}\right).$$

[1] We adopt the commonly used notation where Greek lower case letters refer to scalars, lower case letters refer to (column) vectors, and upper case letters refer to matrices. The $\star$ refers to the symmetric part of A that neither stored nor updated.

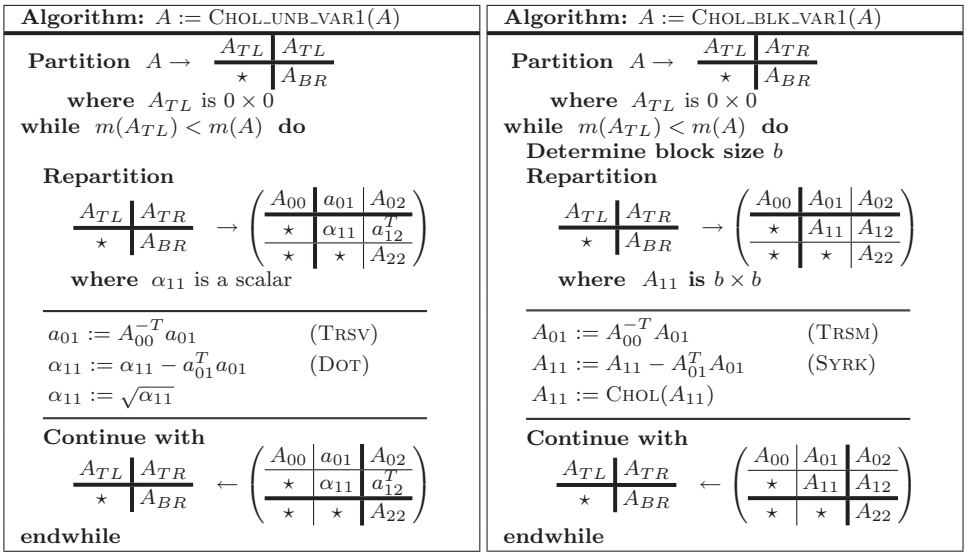

Figure 1: Unblocked and blocked algorithms for computing the Cholesky factorization (Variant 1).

By substituting the partitioned matrices A and R into $A = R^T R$ we find that

$$\left(\begin{array}{c|c} A_{00} & a_{01} \\ \hline \star & \alpha_{11} \end{array}\right) = \left(\begin{array}{c|c} R_{00} & r_{01} \\ \hline 0 & \rho_{11} \end{array}\right)^T \left(\begin{array}{c|c} R_{00} & r_{01} \\ \hline 0 & \rho_{11} \end{array}\right) = \left(\begin{array}{c|c} R_{00}^T R_{00} & R_{00}^T r_{01} \\ \hline \star & r_{01}^T r_{01} + \rho_{11}^2 \end{array}\right),$$

from which we conclude that

$$\left(\begin{array}{c|c} R_{00} = \text{CHOL}(A_{00}) & r_{01} = R_{00}^{-T} a_{01} \\ \hline \star & \rho_{11} = \sqrt{(\alpha_{11} - r_{01}^T r_{01})} \end{array}\right).$$

These three equalities motivate the algorithm in Fig. 1(left). To understand that algorithm, consider that before every iteration, quadrant A_{TL} has already been overwritten by R_{TL}. In the body of the loop, a next row and column are exposed, and updated. The row and the column are then included in A_{TL} so that at the end of the iteration, again A_{TL} contains the result R_{TL}. Eventually A_{TL} envelops the entire matrix, at which point A contains the desired R.

Remark 1. *In the figures in this paper we further clarify the operations that are being performed by indicating the BLAS-like operation that would be used to implement it. These are tabulated in Fig. 2.*

In order to attain high performance, the computation is typically cast in terms of matrix-matrix multiplications by so-called blocked algorithms [12]. For the Cholesky factorization, a blocked version of the algorithm can be derived by partitioning

$$A \rightarrow \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline \star & A_{11} \end{array}\right) \quad \text{and} \quad R \rightarrow \left(\begin{array}{c|c} R_{00} & R_{01} \\ \hline \star & R_{11} \end{array}\right),$$

where A_{11} and R_{11} are $b \times b$. By substituting into $A = R^T R$ we find that

$$\left(\begin{array}{c|c} R_{00} = \text{CHOL}(A_{00}) & R_{01} = R_{00}^{-T} A_{01} \\ \hline \star & R_{11} = \text{CHOL}(A_{11} - R_{01}^T R_{01}) \end{array}\right).$$

Name	Operation
DOT	Dot product
GEMV	General matrix-vector multiply
SYR	Symmetric rank-1 update
TRSV	Triangular solve
GEPP	General rank-k update
GEMP	Matrix times panel-of-columns multiply
GEPM	Panel-of-rows times Matrix multiply
SYMM	Symmetric matrix multiply
SYRK	Symmetric rank-k update
TRMM	Triangular matrix multiplication
TRSM	Triangular solve with multiple right-hand sides

Figure 2: Basic operations used to implement the different algorithms.

A blocked algorithm is then given in Fig. 1(right).

2.2 Systematic derivation

We now show how the same algorithmic variant for the Cholesky factorization, as well as other variants, can be created systematically using the FLAME methodology [3]. The idea is that the user fills out a "worksheet" in a prescribed sequence of steps, so that various conditions are met at specific stages of the computation. The manner in which the worksheet is populated guarantees the correctness of the algorithm. As an example, the worksheet for the derivation of the blocked algorithm from Section 2.1 is illustrated in Fig. 3. The order in which the worksheet was filled out is given in the column marked "Step". Assertions (predicates) indicating the *desired* state of variables at various points in the algorithm are given in the grey boxes. These desired states then dictate the updates to variables, in the clear boxes. Further details for each of the various steps is provided below.

Step 1: Precondition and postcondition. The precondition and postcondition for computing $A := \text{CHOL}(A)$ are given by $\left(A = \hat{A}\right)$ and $\left(A = R \wedge R = \text{CHOL}(\hat{A})\right)$, respectively. Here $\hat{A}$ denotes the original contents of the matrix A and the matrix R is only introduced to express the postcondition as a constraint. The predicates are entered in Steps 1(a) and 1(b) in Fig. 3.

Step 2: Loop Invariant P_{inv}. Next, we choose what the state (contents) of matrix A must be before and after every iteration. For the algorithms in Section 2.1, the state that is maintained is given by the loop-invariant:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} R_{TL} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right) \wedge R_{TL} = \text{CHOL}\left(\hat{A}_{TL}\right).$$

The predicate P_{inv} is entered in four places, marked Step 2 in Fig. 3.

Steps 3 and 4: We first determine the loop-guard G, which is the condition under which the computation remains in the loop. If we choose G to equal $m(A_{TL}) < m(A)$, then the predicate $\neg G$ means that the matrices A_{TL} and $\hat{A}_{TL}$ equal all of A and $\hat{A}$, respectively. As a consequence, the predicate

$$\left(\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} R_{TL} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right) \wedge R_{TL} = \text{CHOL}\left(\hat{A}_{TL}\right)\right) \wedge \neg G$$

implies the desired result $P_{post} : A = R_{TL} \wedge R_{TL} = \text{CHOL}(\hat{A})$. The loop-guard G is entered in Step 3.

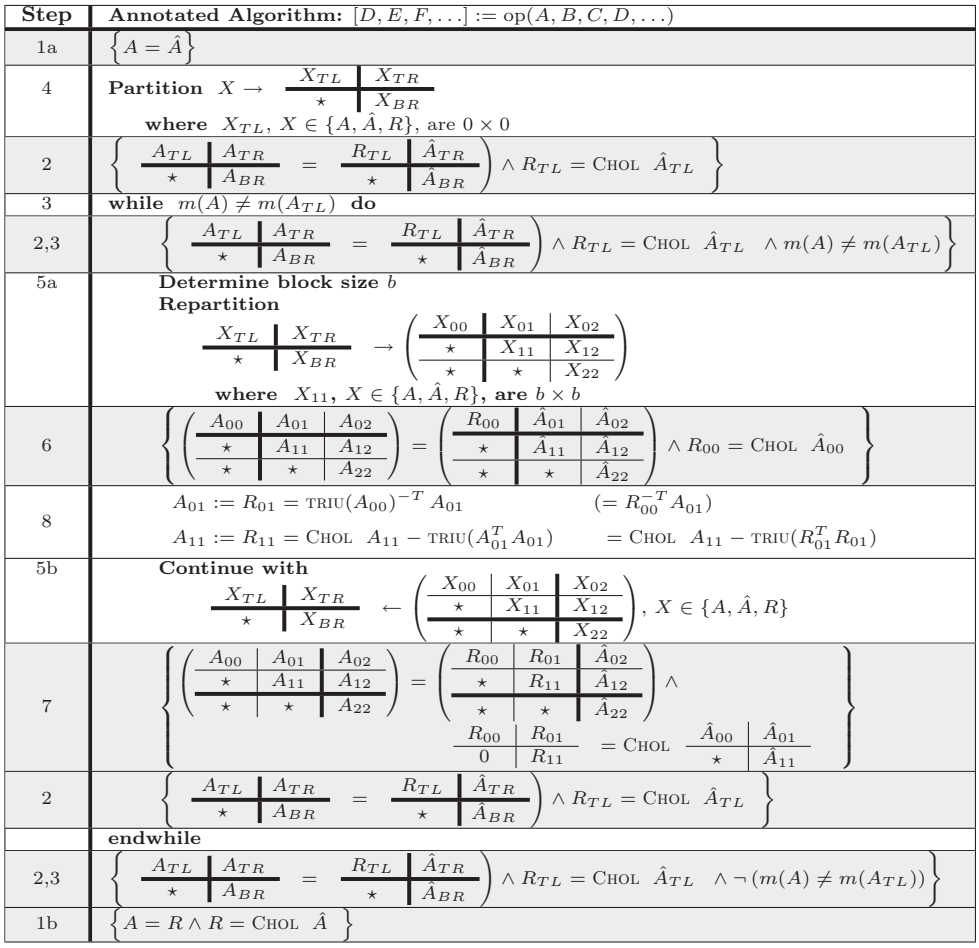

Figure 3: Worksheet for the computation of the Cholesky factorization of matrix A via Variant 1.

Remark 2. *The postcondition and the loop-invariant prescribe a loop-guard.*

Similarly, the initialization given in Step 4 in Figure 3 can be derived from the fact that before the initialization the equality $A = \hat{A}$ holds, while after the initialization the loop-invariant (Step 2) must hold. The initialization given in Step 4 performs no computation: it is merely an indexing operation.

Remark 3. *The precondition and the loop-invariant prescribe the initialization.*

Steps 5a and 5b: For all variants, A_{TL}, $\hat{A}_{TL}$, and R_{TL} start as 0×0 matrices and must end by encompassing all of A, $\hat{A}$, and R_{TL}, respectively. Thus, to expand these submatrices, we must identify submatrices of A, $\hat{A}$, and R to be added to A_{TL}, $\hat{A}_{TL}$, and R_{TL}, respectively. This is accomplished through the repartitioning and redefinition of the quadrants given in Steps 5a and 5b in Figure 3.

Remark 4. *The initialization and loop-guard prescribe in what direction the matrices are to be traversed.*

Step 6: Now we finally get to the point where the loop-invariant dictates the update of submatrices to

be performed in the loop. The thick and thin lines in Step 5a have semantic meaning:

$$\begin{array}{c|c} X_{TL} = X_{00} & X_{TR} = \left(\begin{array}{c|c} X_{01} & X_{02} \end{array}\right) \\ \hline \star & X_{BR} = \left(\begin{array}{c|c} X_{11} & X_{12} \\ \hline \star & X_{22} \end{array}\right) \end{array}, \quad X \in \{A, \hat{A}, R\}.$$

Substituting the matrices $A, \hat{A}$ and R into the loop-invariant we find that

$$\left(\begin{array}{c|c} A_{00} & \left(\begin{array}{c|c} A_{01} & A_{02} \end{array}\right) \\ \hline \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) & \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline \star & A_{22} \end{array}\right) \end{array}\right) = \left(\begin{array}{c|c} R_{00} & \left(\begin{array}{c|c} \hat{A}_{01} & \hat{A}_{02} \end{array}\right) \\ \hline \left(\begin{array}{c} \star \\ \hline \star \end{array}\right) & \left(\begin{array}{c|c} \hat{A}_{11} & \hat{A}_{12} \\ \hline \star & \hat{A}_{11} \end{array}\right) \end{array}\right) \wedge R_{00} = \textsc{Chol}\left(\hat{A}_{00}\right),$$

which simplifies to the expression in Step 6 in Fig. 3.

Step 7: Similarly, the thick and thin lines in Step 5b have semantic meaning:

$$\begin{array}{c|c} X_{TL} = \left(\begin{array}{c|c} X_{00} & X_{01} \\ \hline \star & X_{11} \end{array}\right) & X_{TR} = \left(\begin{array}{c} X_{02} \\ \hline X_{12} \end{array}\right) \\ \hline \star & X_{BR} = X_{22} \end{array}, \quad X \in \{A, \hat{A}, R\}.$$

Again, by substituting $A, \hat{A}$ and R into the loop-invariant shows that, after moving the thick lines, the required contents of A are described by 5b:

$$\left(\begin{array}{c|c} \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline \star & A_{11} \end{array}\right) & \left(\begin{array}{c} A_{02} \\ \hline A_{12} \end{array}\right) \\ \hline \left(\begin{array}{c|c} \star & \star \end{array}\right) & A_{22} \end{array}\right) = \left(\begin{array}{c|c} \left(\begin{array}{c|c} R_{00} & R_{01} \\ \hline \star & R_{11} \end{array}\right) & \left(\begin{array}{c} \hat{A}_{02} \\ \hline \hat{A}_{12} \end{array}\right) \\ \hline \left(\begin{array}{c|c} \star & \star \end{array}\right) & \hat{A}_{22} \end{array}\right) \wedge \left(\begin{array}{c|c} R_{00} & R_{01} \\ \hline \star & R_{11} \end{array}\right) = \textsc{Chol}\left(\left(\begin{array}{c|c} \hat{A}_{00} & \hat{A}_{01} \\ \hline \star & \hat{A}_{11} \end{array}\right)\right),$$

which simplifies to Step 7 in Fig. 3.

Remark 5. *The expressions in Steps 6 and 7 are, in general, obtained via substitution and algebraic manipulation.*

Step 8: Finally, the updates in Step 8 of Figure 3 are determined by comparing the states in Step 6 and 7 of that figure.

Final algorithm: The worksheet in Figure 3 includes the assertions required to systematically derive the algorithm. Variables $\hat{A}$ and R were only introduced to assist the derivation. The final algorithm in Fig. 1(right) is obtained by deleting the extra variables $\hat{A}$ and R and the assertions.

2.3 Deriving the loop based algorithms

Section 2.2 showed how, *given* the loop-invariant, the algorithm can be systematically derived. The fact that the assertions dictate the actual computational steps then proves the correctness of the algorithms as a byproduct. The question now becomes how to systematically *derive* loop-invariants for a given operation. We show next that multiple loop-invariants exist for the computation of the Cholesky factorization, each of which leads to a different algorithm.

The operation is described by the precondition $A = \hat{A}$ and postcondition $\left(A = R \wedge R = \textsc{Chol}\left(\hat{A}\right)\right)$. The partitioning of A, $\hat{A}$, and R into quadrants,

$$X \rightarrow \left(\begin{array}{c|c} X_{TL} & X_{TR} \\ \hline \star & X_{BR} \end{array}\right), \quad X \in \{A, \hat{A}, R\}$$

Invariant 1:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} R_{TL} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right) \wedge R_{TL} = \text{CHOL}\left(\hat{A}_{TL}\right)$$

Invariant 2:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right) \begin{array}{l} \wedge R_{TL} = \text{CHOL}(\hat{A}_{TL}) \\ \wedge R_{TR} = R_{TL}^{-T}\hat{A}_{TR} \end{array}$$

Invariant 3:

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline \star & \hat{A}_{BR} - R_{TR}^{T}R_{TR} \end{array}\right) \begin{array}{l} \wedge R_{TL} = \text{CHOL}(\hat{A}_{TL}) \\ \wedge R_{TR} = R_{TL}^{-T}\hat{A}_{TR} \end{array}$$

Figure 4: States maintained in matrix A corresponding to the algorithms given in Fig. 5 below.

Algorithm: $A := \text{CHOL\_UNB}(A)$

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

where A_{TL} is 0×0

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

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline \star & \alpha_{11} & a_{12}^T \\ \hline \star & \star & A_{22} \end{array}\right)$$

where α_{11} is 1×1

Variant 1:

$a_{01} := A_{00}^{-T} a_{01}$ (TRSV)
$\alpha_{11} := \alpha_{11} - a_{01}^T a_{01}$ (DOT)
$\alpha_{11} := \sqrt{\alpha_{11}}$

Variant 2:

$\alpha_{11} := \alpha_{11} - a_{01}^T a_{01}$ (SYR)
$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{12}^T := a_{12}^T - a_{01}^T A_{02}$ (GEMV)
$a_{12}^T := a_{12}^T / \alpha_{11}$

Variant 3:

$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{12}^T := a_{12}^T / \alpha_{11}$
$A_{22} := A_{22} - a_{12} a_{12}^T$ (SYR)

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline \star & \alpha_{11} & a_{12}^T \\ \hline \star & \star & A_{22} \end{array}\right)$$

endwhile

Algorithm: $A := \text{CHOL\_BLK}(A)$

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

where A_{TL} is 0×0

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

Determine block size b

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$

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

Variant 1:

$A_{01} := A_{00}^{-T} A_{01}$ (TRSM)
$A_{11} := A_{11} - A_{01}^T A_{01}$ (SYRK)
$A_{11} := \text{CHOL}(A_{11})$

Variant 2:

$A_{11} := A_{11} - A_{01}^T A_{01}$ (SYRK)
$A_{11} := \text{CHOL}(A_{11})$
$A_{12} := A_{12} - A_{01}^T A_{02}$ (GEPM)
$A_{12} := A_{11}^{-T} A_{12}$ (TRSM)

Variant 3:

$A_{11} := \text{CHOL}(A_{11})$
$A_{12} := A_{11}^{-T} A_{12}$ (TRSM)
$A_{22} := A_{22} - A_{12}^T A_{12}$ (SYRK)

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$$

endwhile

Figure 5: Unblocked and blocked algorithms for computing the Cholesky factorization.

tracks how the iteration moves through A and allows the current contents of A to be related to $\hat{A}$ and R.

Substituting the partitioned matrices $A, \hat{A}$ and R into the postcondition yields

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline \star & R_{BR} \end{array}\right) \wedge \tag{1}$$

$$\left(\begin{array}{c|c} R_{TL} & R_{TR} \\ \hline \star & R_{BR} \end{array}\right) = \textsc{Chol}\left(\left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right)\right). \tag{2}$$

We call (1)–(2) the *Partitioned Matrix Expression (PME)* for the Cholesky factorization. It defines the factorization as a recurrence relation. Equation (2) is equivalent to the equalities:

$$\begin{array}{rcl} R_{TL} & = & \textsc{Chol}\left(\hat{A}_{TL}\right) \\ R_{TR} & = & R_{TL}^{-T}\hat{A}_{TR} \\ R_{BR} & = & \textsc{Chol}\left(\hat{A}_{BR} - R_{TR}^{T}R_{TR}\right). \end{array}$$

Remark 6. *The PME represents* all *the computations that must be performed, as a function of the quadrants that are tracked as the iteration proceeds.*

A natural state (loop-invariant) for matrix A before and after any given iteration is that *some* of the computations have been performed. Given constraints on the order in which A can be overwritten, this leaves three loop-invariants that can be maintained at the top of the loop, shown in Fig. 4. The unblocked and blocked algorithms in Fig. 5 are obtained by applying the techniques in Section 2.2 with each of these three loop-invariants.

Remark 7. *The family of loop-invariants for computing a given operation is dictated by the way operands are partitioned, the postcondition, and inherent contraints on the order in which the result must be computed.*

Remark 8. *The question of whether these are indeed all possible loop-based algorithms is not one that lies within the scope of this paper.*

3 Inversion of an Upper Triangular Matrix: $R := R^{-1}$

In this section we discuss the "in-place" inversion of a triangular matrix, overwriting the original matrix with the result. By in-place it is meant that no work space is required. The derivation of this algorithms is very similar to those already described for the Cholesky decomposition, so the step-by-step details of the worksheet will be left for the reader to pursue.

We will concentrate on the inversion of an upper-triangular matrix, $R := \hat{R}^{-1}$, where $\hat{R}$ represents the initial contents of the matrix. The precondition and postcondition are given by the predicates $\left(R = \hat{R}\right)$ and $\left(R = \hat{R}^{-1}\right)$, respectively. The fact that both $\hat{R}$ and the result are upper triangular will be implicitly assumed.

Recall from Section 2 that the loop-invariants prescribe the algorithms. Thus, we will concentrate on the

Invariant 1:	Invariant 2:
$\left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & \hat{R}_{TR} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & -\hat{R}_{TL}^{-1}\hat{R}_{TR}\hat{R}_{BR}^{-1} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)$
Invariant 3:	**Invariant 4:**
$\left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & -\hat{R}_{TL}^{-1}\hat{R}_{TR} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & -\hat{R}_{TR}\hat{R}_{BR}^{-1} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)$

Figure 6: States maintained in matrix R corresponding to the algorithms given in Fig. 7 below.

Algorithm: $R := R^{-1}$

Partition $R \rightarrow \left(\begin{array}{c\|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right)$
where R_{TL} is 0×0

while $m(R_{TL}) \neq m(R)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c\|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} R_{00} & R_{01} & R_{02} \\ \hline 0 & R_{11} & R_{12} \\ \hline 0 & 0 & R_{22} \end{array}\right)$$

where R_{11} **is** $b \times b$

Variant 1		Variant 2	
$R_{01} := -R_{00}R_{01}$	(TRMM)	$R_{12} := -R_{12}R_{22}^{-1}$	(TRSM)
$R_{01} := R_{01}R_{11}^{-1}$	(TRSM)	$R_{12} := R_{11}^{-1}R_{12}$	(TRSM)
$R_{11} := R_{11}^{-1}$		$R_{11} := R_{11}^{-1}$	
Variant 3		**Variant 4**	
$R_{12} := -R_{11}^{-1}R_{12}$	(TRSM)	$R_{12} := -R_{12}R_{22}^{-1}$	(TRSM)
$R_{02} := R_{02} + R_{01}R_{12}$	(GEPP)	$R_{02} := R_{02} - R_{01}R_{12}$	(GEPP)
$R_{01} := R_{01}R_{11}^{-1}$	(TRSM)	$R_{01} := R_{00}R_{01}$	(TRMM)
$R_{11} := R_{11}^{-1}$		$R_{11} := R_{11}^{-1}$	

Continue with

$$\left(\begin{array}{c\|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c\|c\|c} R_{00} & R_{01} & R_{02} \\ \hline 0 & R_{11} & R_{12} \\ \hline 0 & 0 & R_{22} \end{array}\right)$$

endwhile

Figure 7: Four algorithmic variants for the inversion of an upper triangular matrix, R.

derivation of loop-invariants for this operation. Partition

$$R \rightarrow \left(\begin{array}{c\|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right) \quad \text{and} \quad \hat{R} \rightarrow \left(\begin{array}{c\|c} \hat{R}_{TL} & \hat{R}_{TR} \\ \hline 0 & \hat{R}_{BR} \end{array}\right),$$

where R_{TL} and $\hat{R}_{TL}$ are square to exploit the triangular structure of the matrix. Substituting these partitioned matrices into the postcondition yields the PME

$$\left(\begin{array}{c\|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array}\right) = \left(\begin{array}{c\|c} \hat{R}_{TL} & \hat{R}_{TR} \\ \hline 0 & \hat{R}_{BR} \end{array}\right)^{-1} = \left(\begin{array}{c\|c} \hat{R}_{TL}^{-1} & -\hat{R}_{TL}^{-1}\hat{R}_{TR}\hat{R}_{BR}^{-1} \\ \hline 0 & \hat{R}_{BR}^{-1} \end{array}\right). \tag{3}$$

As with the Cholesky, this PME relates the quadrants of the overwritten R (the output matrix, containing the inverse of $\hat{R}$) to the quadrants of $\hat{R}$ (the input matrix) in the following way

$$\left(\begin{array}{c|c} R_{TL} = \hat{R}_{TL}^{-1} & R_{TR} = -\hat{R}_{TL}^{-1}\hat{R}_{TR}\hat{R}_{BR}^{-1} \\ \hline & R_{BR} = \hat{R}_{BR}^{-1} \end{array}\right).$$

From the PME, the loop-invariants in Fig. 6 can be derived for four algorithms that traverse the matrix from the top-left to the bottom-right. An additional four loop-invariants exist that represent algorithms progressing from the bottom-right to the top-left; however, we will focus only on the first four.

With the loop-invariants in place, the algorithms for each variant can now be derived. The blocked algorithms are given in Fig. 7.

4 Triangular matrix multiplication by its transpose: $C = UU^T$

We shall now talk briefly about the in-place multiplication of an upper triangular matrix, U, times its own transpose, overwriting the original matrix with the result. As before, we shall focus the upper triangular case in which $C := UU^T$, pointing out that C is symmetric so that only its upper triangular portion is computed and it overwrites the upper triangular part of U.

The precondition and postcondition for this operation are given by $\left(U = \hat{U}\right)$ and $\left(U = \text{TRIU}\left(\hat{U}\hat{U}^T\right)\right)$, respectively. Because of the symmetric structure of the output matrix U and the triangular structure of $\hat{U}$, we partition

$$U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right), \quad \text{and} \quad \hat{U} = \left(\begin{array}{c|c} \hat{U}_{TL} & \hat{U}_{TR} \\ \hline 0 & \hat{U}_{BR} \end{array}\right),$$

where U_{TL} and $\hat{U}_{TL}$ are square. Substituting these partitioned matrices into the postcondition yields the PME:

$$\begin{array}{rcl} \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right) & = & \left(\begin{array}{c|c} \hat{U}_{TL} & \hat{U}_{TR} \\ \hline 0 & \hat{U}_{BR} \end{array}\right)\left(\begin{array}{c|c} \hat{U}_{TL} & \hat{U}_{TR} \\ \hline 0 & \hat{U}_{BR} \end{array}\right)^T \\ & = & \left(\begin{array}{c|c} \hat{U}_{TL}\hat{U}_{TL}^T + \hat{U}_{TR}\hat{U}_{TR}^T & \hat{U}_{TR}\hat{U}_{BR}^T \\ \hline \star & \hat{U}_{BR}\hat{U}_{BR}^T \end{array}\right). \end{array}$$

By taking into account dependencies that arise from the fact that U is being overwritten, the three loop-invariants given in Fig. 8 are identified. Blocked algorithms that are derived from these loop-invariants are presented in Figure 9.

5 Inversion of a Symmetric Positive Definite Matrix

Two algorithms are derived for computing the inverse of an SPD matrix. We show how one of these algorithms can also be obtained by merging carefully chosen algorithms from Sections 2–4 into a one-sweep algorithm.

Invariant 1:

$$\left(\begin{array}{c|c} \hat{U}_{TL}\hat{U}_{TL}^T & \hat{U}_{TR} \\ \hline \star & \hat{U}_{BR} \end{array}\right)$$

Invariant 2:

$$\left(\begin{array}{c|c} \hat{U}_{TL}\hat{U}_{TL}^T + \hat{U}_{TR}\hat{U}_{TR}^T & \hat{U}_{TR} \\ \hline \star & \hat{U}_{BR} \end{array}\right)$$

Invariant 3:

$$\left(\begin{array}{c|c} \hat{U}_{TL}\hat{U}_{TL}^T + \hat{U}_{TR}\hat{U}_{TR}^T & \hat{U}_{TR}\hat{U}_{BR}^T \\ \hline \star & \hat{U}_{BR} \end{array}\right)$$

Figure 8: States mantained in matrix U corresponding to the algorithms given in Fig. 9 below.

Algorithm: $U := UU^T$

Partition $U \rightarrow \left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right)$
where U_{TL} is 0×0

while $m(U_{TL}) \neq m(U)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline \star & U_{11} & U_{12} \\ \hline \star & \star & U_{22} \end{array}\right)$$

where U_{11} **is** $b \times b$

Variant 1:		Variant 2:	
$U_{00} := U_{00} + U_{01}U_{01}^T$	(Syrk)	$U_{01} := U_{01}U_{11}^T$	(Trmm)
$U_{01} := U_{01}U_{11}^T$	(Trmm)	$U_{01} := U_{01} + U_{02}U_{12}^T$	(Gemp)
$U_{11} := U_{11}U_{11}^T$		$U_{11} := U_{11}U_{11}^T$	
		$U_{11} := U_{11} + U_{12}U_{12}^T$	(Syrk)

Variant 3:

$U_{11} := U_{11}U_{11}^T$	
$U_{11} := U_{11} + U_{12}U_{12}^T$	(Syrk)
$U_{12} := U_{12}U_{22}^T$	(Trmm)

Continue with

$$\left(\begin{array}{c|c} U_{TL} & U_{TR} \\ \hline \star & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_{00} & U_{01} & U_{02} \\ \hline \star & U_{11} & U_{12} \\ \hline \star & \star & U_{22} \end{array}\right)$$

endwhile

Figure 9: Three algorithmic variants for multiplying an upper triangular matrix U with its own transpose.

5.1 Derivation of algorithms

Algorithms for computing the inverse can be derived directly from the postcondition, $\left(A = \hat{A}^{-1}\right)$. First, partition these matrices into quadrants,

$$A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \quad \text{and} \quad \hat{A} \rightarrow \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right)$$

Invariant 1:	Invariant 2:
$\left(\begin{array}{c\|c} \hat{A}_{TL}^{-1} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right)$	$\left(\begin{array}{c\|c} \hat{A}_{TL}^{-1} & -A_{TL}^{-1}\hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} - \hat{A}_{TR}^T\hat{A}_{TL}^{-1}\hat{A}_{TR} \end{array}\right)$

Figure 10: States maintained in matrix A corresponding to the algorithms given in Fig. 12 below.

Algorithm: $A := A^{-1}$ (Variant 1)

Partition $A \rightarrow \left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while G **do**
Determine block size b
Repartition
$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
where A_{11} **is** $b \times b$

$Aux := -A_{00}A_{01}$	(Gemp)
$A_{11} := A_{11} + A_{01}^T Aux$	(Gepp)
$A_{11} := \text{Chol}(A_{11})$	
$Aux := Aux\, A_{11}^{-1}$	(Trsm)
$A_{01} := Aux\, A_{11}^{-T}$	(Trsm)
$A_{00} := A_{00} + Aux\, Aux^T$	(Syrk)
$A_{11} := A_{11}^{-1}$	
$A_{11} := A_{11}A_{11}^T$	

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

Algorithm: $A := A^{-1}$ (Variant 2)

Partition $A \rightarrow \left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right)$
where A_{TL} is 0×0
while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition
$\left(\begin{array}{c\|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c\|c\|c} A_{00} & A_{01} & A_{02} \\ \hline \star & A_{11} & A_{12} \\ \hline \star & \star & A_{22} \end{array}\right)$
where A_{11} **is** $b \times b$

$A_{11} := \text{Chol}(A_{11})$	
$A_{01} := A_{01}A_{11}^{-1}$	(Trsm)
$A_{00} := A_{00} + A_{01}A_{01}^T$	(Syrk)
$A_{12} := A_{11}^{-T}A_{12}$	(Trsm)
$A_{02} := A_{02} - A_{01}A_{12}$	(Gepp)
$A_{22} := A_{22} - A_{12}^T A_{12}$	(Syrk)
$A_{01} := A_{01}A_{11}^T$	(Trsm)
$A_{12} := -A_{11}^{-1}A_{12}$	(Trsm)
$A_{11} := A_{11}^{-1}$	
$A_{11} := A_{11}A_{11}^T$	

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

Figure 11: One-sweep algorithms for inverting an SPD matrix.

where A_{TL} and $\hat{A}_{TL}$ are square and of equal size. Substitution of the partitioned matrices into the postcondition yields the PME:

$$\begin{aligned}\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline \star & A_{BR} \end{array}\right) &= \left(\begin{array}{c|c} \hat{A}_{TL} & \hat{A}_{TR} \\ \hline \star & \hat{A}_{BR} \end{array}\right)^{-1} \\ &= \left(\begin{array}{c|c} \hat{A}_{TL}^{-1} + \hat{A}_{TL}^{-1}\hat{A}_{TR}B_{BR}\hat{A}_{TR}^T\hat{A}_{TL}^{-1} & -\hat{A}_{TL}^{-1}\hat{A}_{TR}B_{BR} \\ \hline \star & B_{BR} \end{array}\right),\end{aligned}$$

where we introduce $B_{BR} = \left(\hat{A}_{BR} - \hat{A}_{TR}^T\hat{A}_{TL}^{-1}\hat{A}_{TR}\right)^{-1}$. A rather involved dependence analysis done by the authors identified two loop-invariants, given in Fig. 10. Applying the derivation techniques with these loop-invariants yields the algorithms in Fig. 11.

It is possible to identify more loop invariants other than the two shown in Fig. 10, but the corresponding

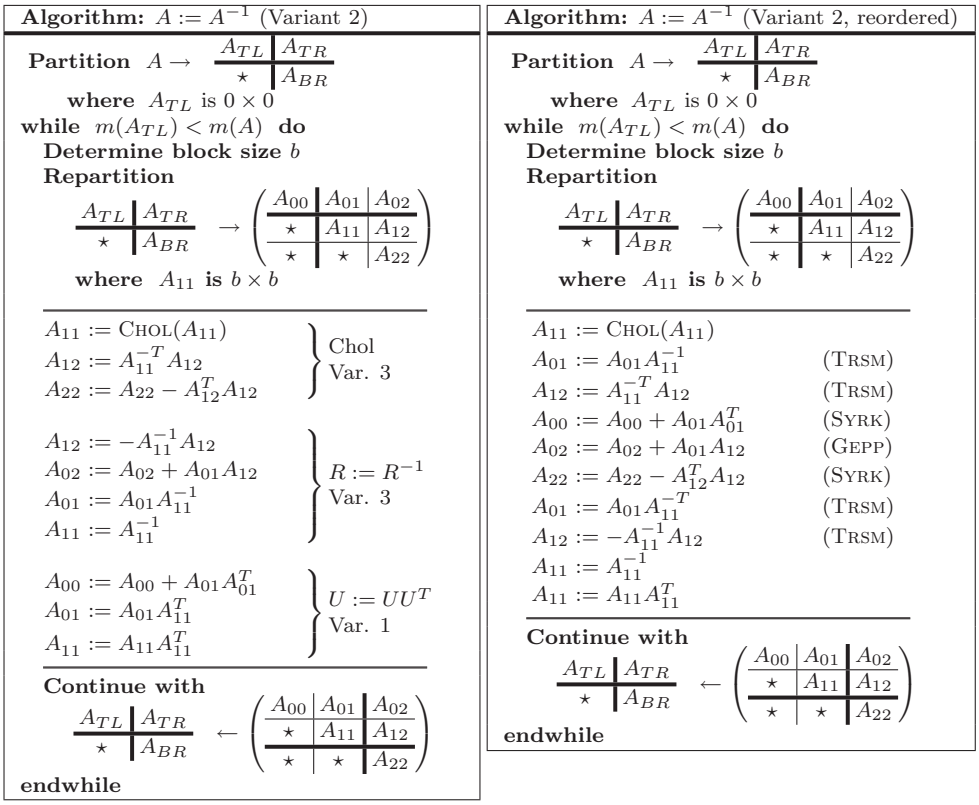

Figure 12: One-sweep algorithm for inverting an SPD matrix as a merging of three sweeps.

algorithms perform redundant computations and/or are numerically instable. More loop invariants yet can be devised by considering the alternative PME

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c} B_{TL} & -B_{TL}\hat{A}_{TL}\hat{A}_{BR}^{-1} \\ \hline \star & \hat{A}_{BR}^{-1} + \hat{A}_{BR}^{-1}\hat{A}_{TR}^T B_{TL}\hat{A}_{TR}\hat{A}_{BR}^{-1} \end{array}\right)$$

where $B_{TL} = \left(\hat{A}_{TL} - \hat{A}_{TR}\hat{A}_{BR}^{-1}\hat{A}_{TR}^T\right)^{-1}$. The corresponding algorithms compute the solution by sweeping the matrix from the bottow right corner as opposed to the two algorithms that we present that sweep the matrix from the top left corner.

5.2 Merging three sweeps into a one-sweep algorithm

As mentioned in the introduction, the inversion of an SPD matrix can be implemented by computing the Cholesky factor ($R := \text{CHOL}(A)$), inverting that factor ($U := R^{-1}$), and multiplying the inverted factor by its own transpose ($A^{-1} := UU^T$). We will call this a three-sweep algorithm, where for each sweep any of the algorithmic variants can be used. A one-sweep algorithm can be obtained by merging carefully chosen algorithmic variants for each of the three sweeps. The result, in Fig. 12, is identical to Fig. 11 (right), which was obtained by applying the FLAME approach. The conditions under which algorithms can be merged is a topic of current research and goes beyond the scope of this paper.

5.3 Discussion

The real benefit of the one-sweep algorithm in Fig. 12 (left) comes from the following observation: The order of the updates in that variant can be changed as in Fig. 12 (right), so that the most time consuming computations ($A_{22} - A_{12}^T A_{12}$, $A_{00} + A_{01}A_{01}^T$, and $A_{01} + A_{01}A_{12}$) can be scheduled to be computed simultaneously:

$$\left(\begin{array}{c|c|c} A_{00} + A_{01}A_{01}^T & & A_{02} + A_{01}A_{12} \\ \hline \star & & \\ \hline \star & \star & A_{22} - A_{12}^T A_{12} \end{array}\right).$$

On a distributed memory architecture, where the matrix is physically distributed among memories, there is the opportunity to: 1) consolidate the communication among processors by first performing the collective communications for the three updates followed by the actual computations, and 2) improve load-balance since during every iteration of the merged algorithm, on each element of the quadrants A_{00}, A_{02} and A_{22} the same amount of computation is performed.

6 Stability

Thus far, we have discussed how to compute the inverse of a symmetric positive definite matrix by means of a three-sweep approach and a single-sweep approach. As part of the three-sweep approach, we described three algorithms for computing the Cholesky factor of an SPD matrix, four algorithms to invert a triangular matrix and three algorithms to multiply a triangular matrix and its transpose. Alternatively, we described two variants to compute the inverse by a single sweep only. In the following we comment on the stability of each of these variants.

Cholesky: In Golub-Van Loan [13], the three Variants presented in Table 5 are derived (two unblocked and one blocked), and the stability is asserted based on an early work of Wilkinson [31]. In that paper, Wilkinson proves the norm-wise backward stability of the factorization, e.g. $\check{R}^T\check{R} = A + \Delta A$, for $\|\Delta A\| \leq k_1 n^{3/2}\|A\|$, where $\check{R}$ is the computed Cholesky factor. Higham [20], proves instead that variant 1 is element-wise backward stable, i.e., $|\Delta A| \leq k_2 n\, |\check{R}^T|\,|\check{R}|$. Pete Stewart in [25] points out that the difference between the Cholesky and the LU factorization (applied to an SPD matrix) is that the former has no element growth, and therefore the algorithm is "unconditionally stable".

Triangular Inverse: Similar to the Cholesky factorization, the first analysis for the inversion of a matrix has been performed by Wilkinson [30]. More recently, Higham [9, 20] presented the analysis for Variants 1 and 3 from Table 6, and claimed the equivalence of the rounding errors for Variant 1 to 2 and 3 under 'suitable implementations'. The unblocked Variant 3 is used in LINPACK [10], while Variant 1 (unblocked and blocked) is used in LAPACK [2]. Variant 4, (Table 6) is unstable because of cancellation: it is possible to show that the non-diagonal entries are computed by (unnecessarily) adding and subtracting many times similar quantities.

Computing UU^T: The operation is a matrix multiplication and therefore perfectly stable.

SPD Inverse: The stability analysis for the invertion of an SPD matrix by means of a three-sweep process is given in [20]. Our presentation shows that the computations performed by the one-sweep algorithm (Variant 2) is identical to that performed by the three-sweep approach, and therefore the stability results carry over

to the one-sweep algorithm.

Numerical experiments were conducted that provide evidence that all algorithmic variants, unblocked and blocked, for any of the operations that were discussed have, for all practical purposes, equivalent numerical properties, with the exception of Variant 4 for inverting a triangular matrix. Also, using Mathematica [32], it was possible to perform symbolic computations. The numeric computation was augmented so that the accumulation (again symbolically) of roundoff error could be observed. The results support the observation that the different variants are equally numerically stable, except for Variant 4 for inverting a triangular matrix. We do note that some care must be taken to use triangular solve with multiple right-hand sides whenever possible over the explicit inversion of the triangular matrix combined with matrix multiplication.

7 Performance Experiments

To evaluate the performance of the algorithms derived in the previous sections, both serial and parallel implementations were tested on a variety of problem sizes and on different architectures.

Remark 9. *Although the best algorithms for each operation attain very good performance, this study is primarily about the qualitative differences between the performance of different algorithms on different architectures.*

7.1 Implementations

Implementing all the algorithms discussed in this paper on sequential, SMP, and distributed memory architectures would represent a considerable coding effort. FLAME was already mentioned as the methodology used to derive the families of algorithms in this paper. In addition, FLAME encompasses a set of APIs for different programming environments, including MATLAB (FLAME@lab), C (FLAME/C), and C interfaced with MPI (PLAPACK) [5, 28, 8, 15, 24]. These APIs have the benefit that the code closely resembles the algorithms as they are presented in this paper. Most importantly, they hide the indexing that makes coding in a traditional style time-consuming. For blocked algorithms, the cost of raising the level of abstraction of the code is amortized over enough computation that it does not noticeably affect performance.

The FLAME/C and PLAPACK APIs were used for all the implementations, making the coding effort quite manageable. For examples see the paper [5].

7.2 Platforms

The two machines chosen for this study were designed to highlight performance variations when using substantially different architectures and/or programming models.

Shared Memory IBM Power 4 SMP System. This architecture consists of SMP nodes, each containing sixteen 1.3 GHz Power4 processors and 32 GBytes of shared memory. The processors operate at four FLOPS (floating point operations per second) per cycle for a peak theoretical performance of 5.2 GFLOPS/proc (billions of FLOPS, per processor), with a DGEMM (matrix-matrix multiply) benchmarked by the authors at 3.7 GFLOPS/proc. We only measured performance within a single SMP node.

On this architecture, we compared performance when parallelism was attained in two different ways: 1) Implementing the algorithms with PLAPACK, which uses message passing via calls to IBM's MPI library; and 2) invoking the sequential algorithms with calls to the multithreaded BLAS that are part of IBM's ESSL library.

Distributed Memory Cray-Dell Linux Cluster. This system consists of an array of Intel PowerEdge 1750 Xeon processors operating at 3.06 GHz. Each compute node contains two processors and has 2 GB of total shared memory (1 Gb/proc). The theoretical peak for each processor is 6.12 GFLOPS (2 FLOPS per clock cycle), with the DGEMM, as part of Intel's MKL 7.2.1 library, benchmarked by the authors at roughly 4.8 GFLOPS.

On this system we measured the performance of PLAPACK-based implementations, linked to the MPICH MPI implementation [16] and Intel's MKL library as well as to the GotoBLAS [14].

7.3 Reading the graphs

The performance attained by the different implementations is given in Figs. 13–16. The top line of most of the graphs represents the performance attained on the architecture by matrix-matrix multiplication (DGEMM). Since all the algorithms cast most computation in terms of this operation, its performance is the limiting factor. In the case where different BLAS implementations were employed, the theoretical peak of the machine was used as the top line of the graph. The following operation counts were used for each of the algorithms: $\frac{1}{3}n^3$ for each of CHOL(A), R^{-1}, and UU^T, and n^3 for the inversion of an SPD matrix. In the legends, the variant numbers correspond to the numbering of Figs. 4, 6, 8, and 10, while the operations within parentheses indicate the BLAS operation in which the bulk of the computation for that variant is cast (See Fig. 2 for details).

7.4 Sequential performance

In Fig. 13 we show performance on a single CPU of the IBM Power4 system. In these experiments, a block size of 96 was used for all algorithms. Variant 4 for computing $R := R^{-1}$ attains considerably worse performance since it performs more computations than necessary. From the graphs, it is obvious which algorithmic variant was incorporated in LAPACK.

7.5 Parallel performance

In Fig. 14 we report performance results from experiments on a single sixteen CPU SMP node of the IBM Power4 system and on 16 processors (eight nodes with two processors each) of the Cray-Dell cluster. Since the two systems attain different peak rates of computation, the fraction of DGEMM performance that is attained by the implementations is reported.

On the IBM system parallelism was attained in two different ways: by linking sequential FLAME implementations to the ESSL multithreaded BLAS library and by executing PLAPACK implementations. For the FLAME experiments an (algorithmic) block size of 96 was used. The PLAPACK experiments distributed the matrix using a block size of 32 and used an algorithmic block size of 96.

The experiments on the IBM systems show that linking to multithreaded BLAS yields better performance than the PLAPACK implementations since exploiting the SMP features of the system avoids much of the overhead of communication and load-balancing.

For the Cholesky factorization the PLAPACK Variant 1 performs substantially worse than the other variants. This is due to the fact that this variant is rich in triangular solves with a limited number of right-hand sides. This operation inherently does not parallelize well on distributed memory architectures due to dependencies. Interestingly, Variant 1 for the Cholesky factorization attains the best performance in the sequential experiment on the same machine.

The PLAPACK implementations of Variants 1 and 2 for computing R^{-1} do not perform well. Variant 1 is rich in triangular matrix times panel-of-columns multiply where the matrix being multiplied has a limited number of columns. It is not inherent that that operation does not parallelize well. Rather, it is the

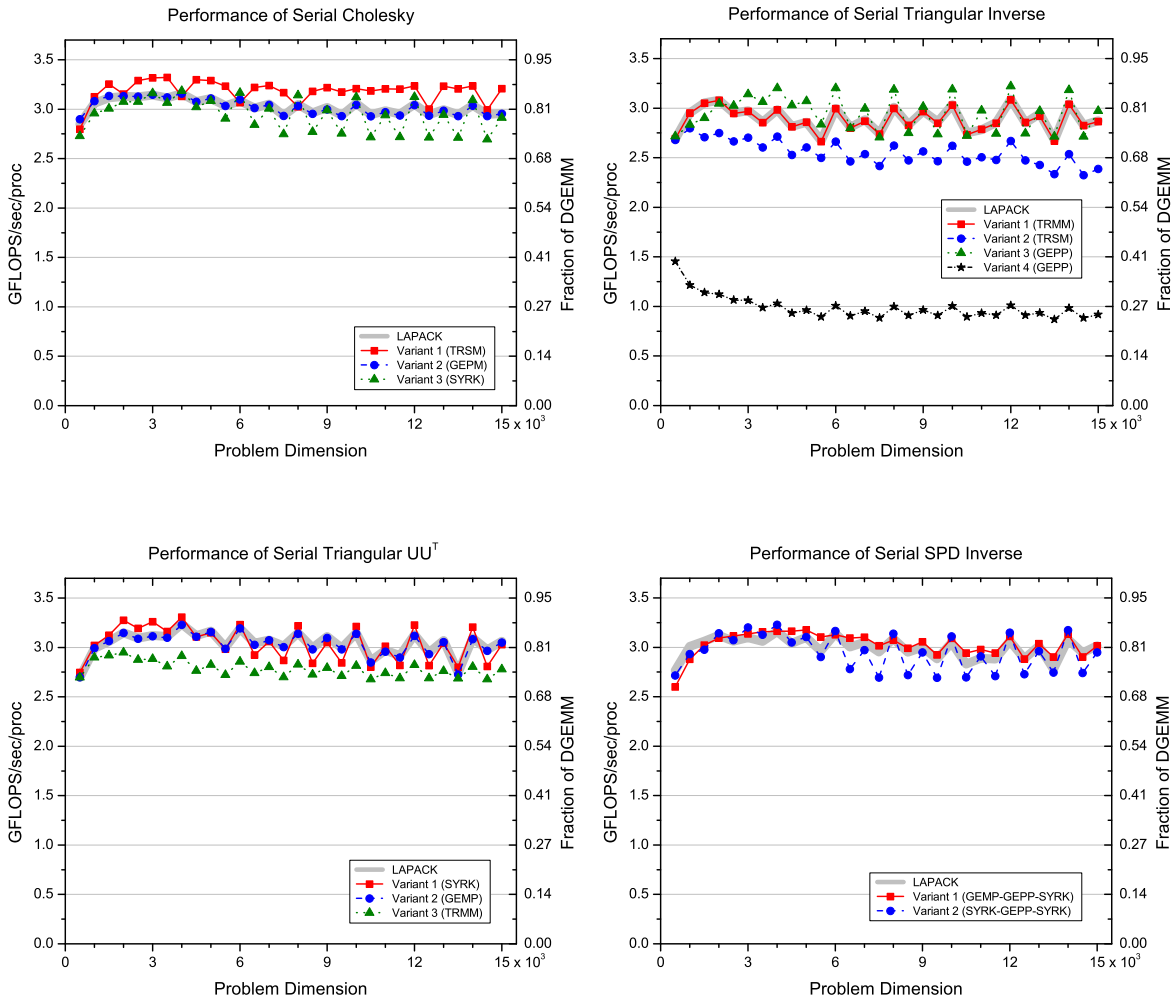

Figure 13: Sequential performance on the IBM Power4 system.

PLAPACK implementation for that BLAS operation that is not completely optimized. Similar comments apply to PLAPACK Variant 3 for computing UU^T and PLAPACK Variant 2 for computing the inversion of an SPD matrix. Note the cross-over between the curves for the SMP Variants 2 and 3 for the parallel triangular inverse operation. This shows that different algorithmic variants may be appropriate for different problem sizes.

It is again obvious from the graphs which algorithmic variant is used for each of the three sweeps as part of LAPACK. The LAPACK curve does not match either of the FLAME variants in the SPD inversion graph since LAPACK uses a three-sweep algorithm.

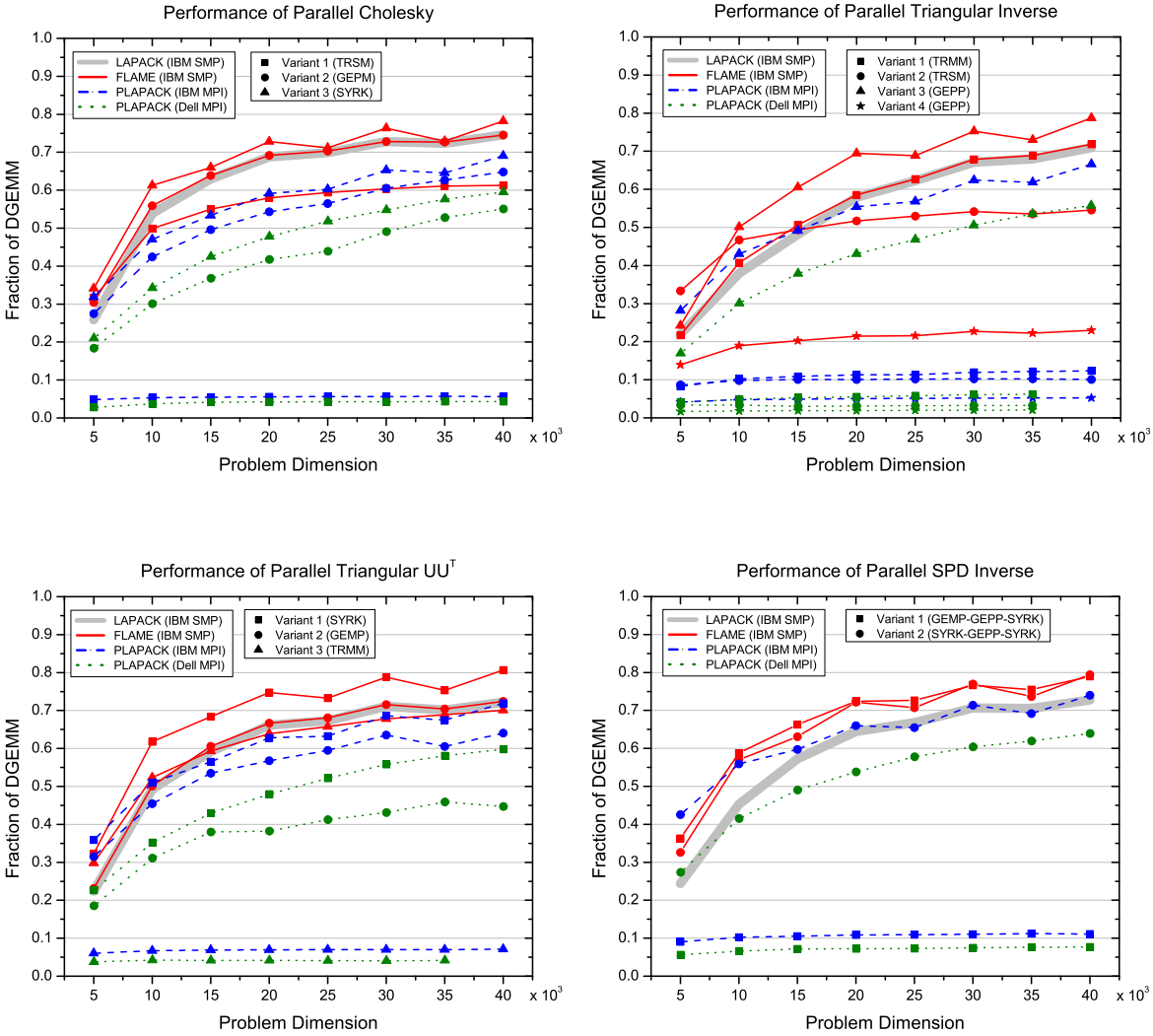

Figure 14: Parallel Performance.

7.6 Scalability

In Fig. 15 we report the scalability of the best algorithmic variants for each of the four operations. It is well-known that for these types of distributed memory algorithms it is necessary to scale the problem size with the square-root of the number of processors, so that memory-use per processor is kept constant [19, 26]. Notice that as the number of processors is increased, the rate of performance attained eventually decreases very slowly, indicating that the implementations are essentially scalable.

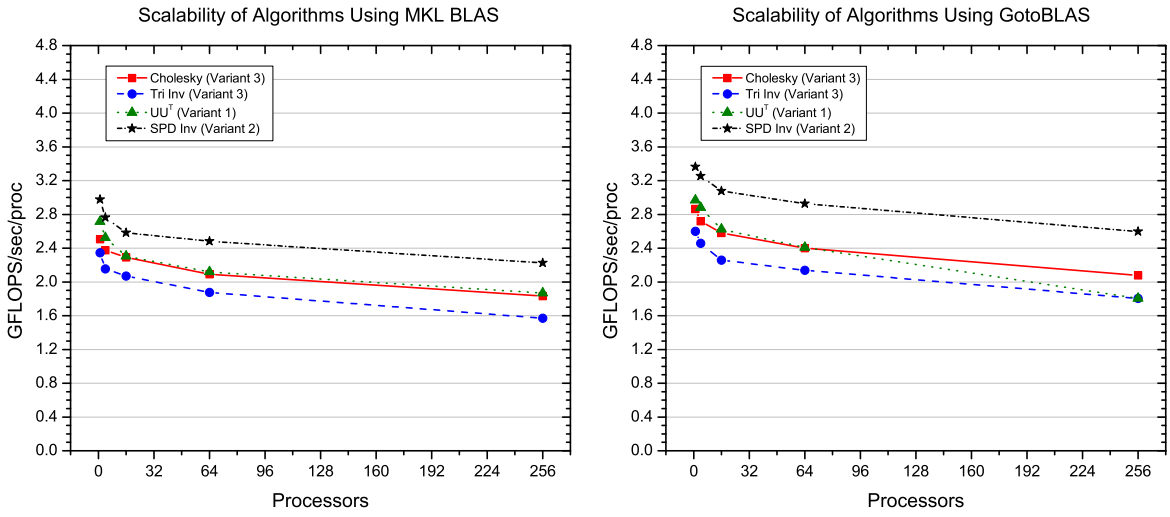

Figure 15: Scalability on the Dell Cluster. Here the matrix size is scaled to equal $5000 \times \sqrt{p}$ so that memory use per processor is held constant. Left: when linked to MKL 7.X. Right: when linked to GotoBLAS 0.97.

7.7 Comparison of the Three-Sweep and Single-Sweep Algorithms

Finally we examine the benefits of consolidating the collective communications and improving the load balancing in the single-sweep algorithm. In Fig. 16 (left) we show improvements in raw performance on the Cray-Dell system. The improvement over three-sweep algorithm is quite substantial, in the 15-30% range. Fig. 16(right) shows the time savings gained for the PLAPACK implementations of the SPD inverse algorithms.

On serial and SMP architectures, essentially no performance improvements were observed by using the single-sweep algorithms over the best three-sweep algorithm. This is to be expected, since for these architectures the communications and load balancing are not an issue.

8 Conclusion

In this paper, we have shown that it is beneficial to be able to find different algorithmic variants for dense linear algebra operations. The best algorithm can be then be chosen for a given situation. This choice is often a function of the architecture, the problem size, and the optimized libraries to which the implementations are linked. The FLAME approach to deriving algorithms enables a systematic generation of such families of algorithms.

Another contribution of the paper lies with the link it establishes between the three-sweep and one-sweep approach to computing the inverse of an SPD matrix. The observation that the traditional three-sweep algorithm could be fused together so that only a single pass through the matrix is required has a number of advantages. The single-sweep method provides for greater flexibility because the sequence of operations can be arranged differently than they would be if done as three separate sweeps. This allows the operations of the SPD inverse to be organized to optimize load balance and communication. The resulting single-sweep algorithm consistently outperforms the three-sweep method.

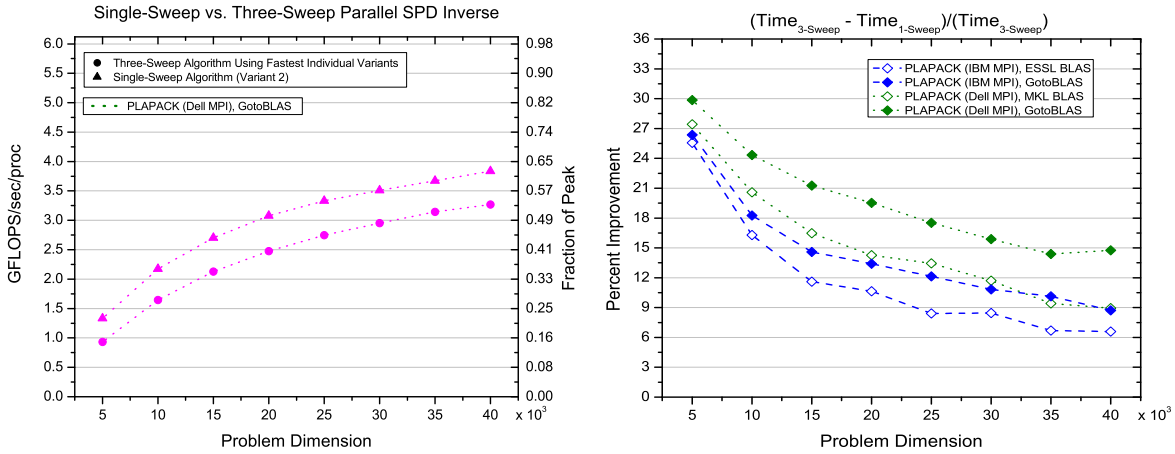

Figure 16: Comparison of the Three-Sweep and Single-Sweep SPD inverse algorithms. The left panel shows the performance difference for the case run on the Cray-Dell system linked to the GotoBLAS. The right panel shows the wall-clock savings for all PLAPACK cases.

The paper raises many new questions. In particular, the availability of many algorithms and implementations means that a decision must be made as to when to use what algorithm. One approach is to use empirical data from performance experiments to tune the decision process. This is an approach that has been applied in the simpler arena of matrix-matrix multiplication (DGEMM) by the PHiPAC and ATLAS projects [6, 29]. An alternative approach would be to carefully design every layer of a library so that its performance can be accurately modeled. We intend to pursue this second approach in the future.

9 Acknowledgements

The authors would like to acknowledge the Texas Advanced Computing Center (TACC) for providing access to the IBM Power4 and Cray-Dell PC Linux cluster machines, along with other computing resources, used in the development of this study.

More Information

For more information on FLAME and PLAPACK visit

`http://www.cs.utexas.edu/users/flame`
`http://www.cs.utexas.edu/users/plapack`

References

[1] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye Wu. PLAPACK: Parallel Linear Algebra Package. In *Proceedings of the SIAM Parallel Processing Conference*, 1997.

[2] E. Anderson, Z. Bai, C. H. Bischof, S. Blackford, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, and D. C. Sorensen. *LAPACK Users' Guide*. Third edition, 1999.

[3] 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 Trans. Math. Soft.*, 31(1):1–26, March 2005.

[4] Paolo Bientinesi, Sergey Kolos, and Robert van de Geijn. Automatic derivation of linear algebra algorithms with application to control theory. In *Proceedings of PARA'04 State-of-the-Art in Scientific Computing*, June 20-23 2004.

[5] 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), March 2005.

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

[7] J. Choi, J. Dongarra, R. Pozo, and D. 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.

[8] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R. van de Geijn. Parallel Implementation of BLAS: General Techniques for Level 3 BLAS. *Concurrency: Practice and Experience*, 9(9):837–857, Sept. 1997.

[9] Jeremy J. Du Croz and Nicholas J. Higham. Stability of methods for matrix inversion. *IMA Journal of Numerical Analysis*, 12:1–19, 1992.

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

[11] Jack 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.

[12] Jack Dongarra, Iain Duff, Danny Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, 1991.

[13] Gene Golub and Charles van Loan. *Matrix Computations, 3rd Ed.* The Johns Hopkins University Press, Baltimore, MD, 1996.

[14] Kazushige Goto and Robert van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Soft.* submitted.

[15] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[16] William Gropp and Ewing Lusk. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL-06/6, Argonne National Laboratory, 1994.

[17] 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.

[18] Brian Gunter. *Computational Methods and Processing Strategies for Estimating Earth's Gravity Field.* PhD thesis, Department of Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin, 2004.

[19] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM J. Sci. Stat. Comput.*, 15(5):1201–1226, 1994.

[20] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[21] Enrique Quintana and Robert van de Geijn. Formal derivation of algorithms for the triangular Sylvester equation. *ACM Trans. Math. Soft.*, 29(2):218–243, July 2003.

[22] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.

[23] R. Sanso and R. Rummel, editors. *Theory of Satellite Geodesy and Gravity Field Determination*, volume 25 of *Lecture Notes in Earth Sciences.* Springer-Verlag, Berlin, 1989.

[24] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference.* The MIT Press, 1996.

[25] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, Orlando, Florida, 1973.

[26] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[27] B. Tapley, B. Schutz, and G. Born. *Statistical Orbit Determination.* Elsevier Academic Press, 2004.

[28] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[29] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

[30] J. H. Wilkinson. Error analysis of direct methods of matrix inversion. *J. ACM*, 8(3):281–330, 1961.

[31] J. H. Wilkinson. A priori error analysis of algebraic processes. In I. G. Petrovsky, editor, *Proc. International Congress of Mathematicians, Moscow 1966*, pages 629–640. Mir Publishers, Moscow, 1968.

[32] Stephen Wolfram. *The Mathematica Book: 3rd Edition.* Cambridge University Press, 1996.