Abstract. The Cholesky factorization operation is used to demonstrate consistent representations for dense linear algebra algorithms, the theory that motivates these algorithms, and the practical implementation of these algorithms in code. The notation mimics the visual aids, in the form of pictures, that are often used in a classroom setting to clarify algorithms. It supports the comparing and contrasting of different algorithms for computing the same operation as well as similar algorithms for different operations. Since the representation of the algorithm in code closely mirrors the algorithm itself, it enables high productivity by novices and experts alike. It is shown that the techniques support high performance implementations.

Key words. linear algebra, notation, algorithms, dense matrices, FLAME

AMS subject classifications. 65F05, 65Y10.

1. Introduction. A beginning course on numerical linear algebra should include an exposure to the issues related to the efficient implementation of the algorithms. This paper provides a stand-alone tutorial for students with a minimal exposure to linear algebra and programming. It uses the Cholesky factorization operation, which is simple enough to allow us to target a broad audience yet complex enough to illustrate the issues, as the central example. Although the paper is accessible to the novice, even experts may benefit from the techniques that we use.

In the past, hands-on activities related to the high-performance implementation of dense linear algebra algorithms in upper division undergraduate and beginning graduate courses on numerical linear algebra have required substantial guidance to ensure that most if not all students are successful. The use of class projects that utilize standard tools of the trade, like the Basic Linear Algebra Subprograms (BLAS) [7, 5, 4], invariably lead to long lines of students seeking help with, for example, indexing mistakes. This led us to design an Application Programming Interface (API) [2] that greatly reduces the opportunity for making such mistakes. While this has made office hours more lonely, it now allows us to assign much more challenging and meaningful projects in a beginning course.

A telling story that our API may be a powerful pedagogical tool relates the first time it was used in an undergraduate class. Given to the students was an implementation and a driver (program that tests the implementation) of an algorithm for computing a simple linear algebra operation. The exercise was to implement an alternative algorithm and test it. A student who had never seen the API before excitedly came to office hours upon completing this assignment. His implementation had given the right answer the first time he compiled and ran it, which was unlike any previous programming experience he had had. Upon examining his handywork, I noticed that he had forgotten to change the driver, which still called the implementation of the original algorithm. (In other words, he had forgotten to link the driver to his implementation of the alternate algorithm.) The initial disappointment of the student
after I pointed out his mistake quickly turned back to excitement when the linking problem was fixed, and without any further corrections it indeed did compile, link, and run correctly. This scenario continues to frequently reoccur, leading us to believe the methodology to be a powerful weapon of math instruction.

The primary pedagogical contribution of this paper lies with exercises that we make available that can be incorporated into a typical first year course on numerical linear algebra at the upper division undergraduate or beginning graduate level. These exercises allow a novice to quickly translate linear algebra algorithms to Matlab’s M-script language and the C programming language even if this person has very limited or no background in these languages. These appear as Exercises ??, ??, ... with details available at


These exercises teach the user the benefits of coding at a level of abstraction that hides the intricate indices that are the root of many programming errors.

The paper also shows that there is a consistent notation for presenting both the theory and the algorithms related to matrix operations that closely matches this alternative representation of the code that implements the algorithms. Together this creates a stand-alone unit that leads students through most issues related to this topic.

2. Preliminaries. In this section we review the notation, matrices with special properties, and linear algebra operations that will be encountered in the paper.

2.1. Notation. A (column) vector, $x$, of length $n$ will be denoted as $x \in \mathbb{R}^n$ where

$$x = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}.$$ 

An $m \times n$ matrix, $A$, is denoted as $A \in \mathbb{R}^{m \times n}$ and its elements are given by

$$A = \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \cdots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \cdots & \alpha_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \cdots & \alpha_{m-1,n-1} \end{pmatrix}.$$ 

**Remark 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 a part of $A$ that is neither stored nor updated. The exceptions are integers, for which we use lower case roman letter from the set $\{i, j, k, m, n\}$.

**Definition 2.1.** The transpose of $A \in \mathbb{R}^{m \times n}$ is defined by the matrix $B \in \mathbb{R}^{n \times m}$ with elements $\beta_{i,j} = \alpha_{j,i}$. The transpose of a matrix $A$ is denoted by $A^T$.

2.2. Partitioning matrices and vectors. In this paper we will heavily exploit matrices and vectors that have been partitioned: It may become convenient to partition $A \in \mathbb{R}^{m \times n}$ into left and right submatrices, top and bottom submatrices, or into quadrants:

$$A \to \left( \begin{array}{c|c} A_L & A_R \end{array} \right), \quad A \to \left( \begin{array}{c} A_T \\ A_B \end{array} \right), \quad \text{or} \quad A \to \left( \begin{array}{c|c} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{array} \right).$$
Here, "T", "B", "L", and "R" stand for "Top", "Bottom", "Left", and "Right", respectively. Similarly, a vector $x \in \mathbb{R}^n$ may be partitioned into a top and bottom subvector: $x \rightarrow \begin{pmatrix} x_T \\ x_B \end{pmatrix}$.

**Exercise 2.1.** Partition $A \in \mathbb{R}^{m \times n}$ as $A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$. Show that $A^T = \begin{pmatrix} A_{TL}^T \\ A_{TR}^T \end{pmatrix}$.

When we express our algorithms, we will often take a partitioning of a matrix and repartition it to expose submatrices:

$$
\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & a_{01} & A_{02} \\ a_{10} & a_{11} & a_{12} \\ \end{pmatrix} \text{ or } \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \end{pmatrix}.
$$

Here the thick lines have semantic meaning: The repartitioning represents the renamings/partitionings

$$
A_{TL} \rightarrow A_{00} \quad \quad A_{TR} \rightarrow (a_{01} \quad A_{02}) \\
A_{BL} \rightarrow (a_{10} \quad a_{11} \quad A_{12}) \quad \quad A_{BR} \rightarrow (a_{20} \quad a_{21} \quad A_{22})
$$

and

$$
A_{TL} \rightarrow A_{00} \quad \quad A_{TR} \rightarrow (A_{01} \quad A_{02}) \\
A_{BL} \rightarrow (A_{10} \quad A_{11}) \quad \quad A_{BR} \rightarrow (A_{12} \quad A_{20} \quad A_{21} \quad A_{22})
$$

respectively. Progress through a matrix will be indicated by the related operation that consolidates submatrices:

$$
\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & a_{01} & A_{02} \\ a_{10} & a_{11} & a_{12} \\ \end{pmatrix} \quad \text{ or } \quad \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \end{pmatrix}.
$$

Again the thick lines have semantic meaning: The notation represents the consolidation

$$
A_{TL} \leftarrow (A_{00} \quad a_{01} \quad a_{10} \quad a_{11}) \quad \quad A_{TR} \leftarrow (a_{02} \quad a_{12}) \\
A_{BL} \leftarrow (A_{20} \quad a_{21} \quad a_{21}) \quad \quad A_{BR} \leftarrow (a_{22})
$$

and

$$
A_{TL} \leftarrow (A_{00} \quad A_{01} \quad A_{10} \quad A_{11}) \quad \quad A_{TR} \leftarrow (A_{02} \quad A_{12}) \\
A_{BL} \leftarrow (A_{20} \quad A_{21} \quad A_{21}) \quad \quad A_{BR} \leftarrow A_{22},
$$

respectively.

Our notation attempts to give maximal clues regarding the submatrices. Consider

$$
\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} & a_{01} & A_{02} \\ a_{10} & a_{11} & a_{12} \\ \end{pmatrix}.
$$
The subscripts indicate the location of the submatrices as does the placement in the $3 \times 3$ partitioning. The choice of letters used, $\alpha$, $a$, and $A$, all indicate that this is a partitioning of an original matrix $A$. The center identifier $\alpha_{11}$ indicates that this submatrix is a scalar. This then implies that $a_{01}$ and $a_{21}$ are vectors, since they must have a unit column dimension. Hence the use of the lowercase Roman letter "a". The identifiers $a_{10}^T$ and $a_{12}^T$ are used to indicate that these are row vectors. By contrast, $$
abla \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}.$$ indicates that the center submatrix should be viewed as a matrix, which means that submatrices $A_{01}$, $A_{10}$, $A_{12}$, and $A_{21}$ are all matrices. Thus, no $^T$ is part of the identifiers $A_{10}$ and $A_{12}$. In our opinion, this combination of clues makes it easier to understand the subsequent explanations of theory and algorithms related to linear algebra operations.

**Exercise 2.2.** We provide a number of macros for typesetting our notation with $\LaTeX$. Familiarize yourself with these tools by visiting http://www.cs.utexas.edu/users/flame/Tutorial/Cholesky/.

**2.3. Matrices with special properties.** We now review matrices with special structure and properties that will be encountered in the remainder of this paper.

**Definition 2.2.** $A \in \mathbb{R}^{n \times n}$ is said to be lower (upper) triangular if all the elements above (below) the diagonal equal zero.

**Exercise 2.3.** Partition a lower triangular matrix $A$ as $A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$ where $A_{TL}$ is square. Show that $A_{TL}$ and $A_{BR}$ are both lower triangular and $A_{TR} = 0$.

**State and prove a similar property for an upper triangular matrix.**

**Definition 2.3.** $A \in \mathbb{R}^{n \times n}$ is said to be symmetric if $A = A^T$.

**Exercise 2.4.** Partition symmetric matrix $A$ as $A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$ where $A_{TL}$ is square. Show that $A_{TL}$ and $A_{BR}$ are both symmetric and $A_{TR} = A_{BR}^T$.

**Definition 2.4.** A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is said to be symmetric positive definite (SPD) if for all nonzero $x \in \mathbb{R}^n$ it holds that $x^T Ax > 0$.

**Exercise 2.5.** Let $A \in \mathbb{R}^{n \times n}$ be a SPD matrix. Prove that $A$ is nonsingular.

**2.4. Linear algebra operations and the basics of multi-level memory.**

In Fig. 2.1 the memory hierarchy of a typical current sequential cache-based computer architecture is depicted. At the top are the registers, of which there are few.
Computation can only be performed by the Central Processing Unit (CPU) once data resides in registers. Data that originally reside in main memory must be brought into the registers, which requires memory operations (memops) that represent overhead especially because memory is very slow relative to the speed of the CPU. A limited amount of fast (cache) memory is available to overcome this inherent memory bottleneck. Since the early 1990s multiple levels of cache have been added which balances the amount of very fast memory with the cost of providing that very fast memory.

All basic linear algebra operations that will be used as part of our discussions are summarized in Fig. 2.2. In that figure we also note the approximate number of memops and floating point operations (flops) that are performed when computing the operation on a computer. Let us examine them in detail:

**Dot product** (DOT) \( \alpha := x^T y \) where \( x, y \in \mathbb{R}^n \). Here \( \alpha \) can be accumulated in a register before writing out the result. Letting \( \chi_i \) and \( \psi_i \) represent the \( i \)th element of vectors \( x \) and \( y \), respectively, \( x^T y = \sum_{i=0}^{n-1} \chi_i \psi_i \). This requires \( n \) multiplies and \( n - 1 \) adds, for a total of approximately \( 2n \) flops. Each pair of elements, \( \chi_i \) and \( \psi_i \), must be read from memory, incurring \( 2n \) memops. The cost of writing \( \alpha \) back to memory is inconsequential. For every memop only one flop is performed which means that if memory is very slow, the DOT operation should be avoided: Algorithms should be arranged so that a smaller fraction of operations are cast in terms of the DOT operation.

**Scaling of a vector** (SCAL) \( x := \alpha x \) \( x \in \mathbb{R}^n \). Here \( \alpha \) is assumed to be symmetric and \( \alpha \) can be accumulated in a register before writing out the result. Letting \( \chi_i \) and \( \psi_i \) represent the \( i \)th element of \( x \). Here there is the opportunity to move \( x \) and/or \( y \) into cache memory. Since each element of \( A \) is used only once, the primary cost comes from reading these \( mn \) elements from memory.

**Symmetric rank-1 update** (SYR) \( A := \alpha \text{tril}(xx^T) + A \). Here \( \text{tril}(Z) \) equals the lower triangular part of square matrix \( Z \). Matrix \( A \) is assumed to be symmetric and only the lower triangular part is stored. The operation performs \( A := \alpha xx^T + A \) which is again symmetric so that only the lower triangular part of \( \alpha xx^T \) needs to be
added to the contents of the lower triangular part of $A$. Note that

$$
\text{tril}(\alpha xx^T) = \text{tril}(x(\alpha x)^T) = \begin{pmatrix}
\chi_0(\alpha \chi_0) & 0 & \cdots & 0 \\
\chi_1(\alpha \chi_0) & \chi_1(\alpha \chi_1) & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
\chi_{n-1}(\alpha \chi_0) & \chi_{n-1}(\alpha \chi_1) & \cdots & \chi_{n-1}(\alpha \chi_{n-1})
\end{pmatrix}.
$$

The scaled vector $\alpha x$ can be formed at a cost of $O(n)$ during the update of the lower triangular elements. Each lower triangular element must be read from memory followed by an add and a multiplication before it is written back to memory, for a total of approximately $n^2$ flops and $n^2$ memops.

**Triangular solve** ($\text{Trsv}$) $x := L^{-1}x$. This operation derives its name from the fact that it is typically computed as the solution of $Ly = x$ where the result vector $y$ overwrites the input vector $x$. As for matrix-vector multiplication, vectors $x$ and/or $y$ can be reused once they are read from memory. As for GEMV, it is the cost of reading the elements of $L$ that contributes most of the memops.

**General matrix-matrix multiplication** (GEMM) $C := \alpha AB + C$. This operation can be viewed as multiple matrix-vector multiplications with the same matrix $A$: Partition by columns as

$$
C \rightarrow ( c_0 \ | \ c_1 \ | \ \cdots \ | \ c_{n-1} ) \quad \text{and} \quad B \rightarrow ( b_0 \ | \ b_1 \ | \ \cdots \ | \ b_{n-1} ).
$$

Then column $c_j$ must be updated as $c_j := \alpha A b_j + c_j$. Now, if small enough, $A$ can be moved into one of the caches (typically the L2 cache) and reused many times. Regardless, this operation requires each of the elements of $C$, $A$, and $B$ to be read from memory and the result matrix $C$ to be written back to memory, for a total of at least $2mn + mk + kn$ memops. These are amortized over the required $2mnk$ flops. If matrix $A$ does not fit in cache memory, the matrices can be partitioned into blocks and the GEMM can be stages as multiple matrix-matrix multiplications with these smaller submatrices.

**Symmetric rank-k update** (SYRK) $C := \alpha \text{tril}(AA^T) + C$. The issues related to this operation are like matrix-matrix multiplication, except that only the lower triangular part of $A$ is updated, which means that approximately half the number of flops are performed.

**Triangular solve with multiple right-hand sides** ($\text{Trsm}$) $B := \alpha BL^{-T}$. This operation can be viewed as computing the solution, $X$, to $XL^T = B$, overwriting $B$ with $X$. Manipulating this further, we find that $(XL^T)^T = B^T$ so that $LX^T = B^T$. Partitioning by rows as

$$
X = \begin{pmatrix}
x_0^T \\
x_1^T \\
\vdots \\
x_{m-1}^T
\end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix}
b_0^T \\
b_1^T \\
\vdots \\
b_{m-1}^T
\end{pmatrix}
$$

we find that $L \begin{pmatrix} x_0 \ | \ x_1 \ | \ \cdots \ | \ x_{m-1} \end{pmatrix} = \begin{pmatrix} b_0 \ | \ b_1 \ | \ \cdots \ | \ b_{m-1} \end{pmatrix}$ so that the $i$th row of $B$, $b_i^T$, must be overwritten by $x_i^T$ where $Lx_i = b_i$, the solution of a triangular system. This shows that, indeed, this operation is equivalent to a triangular solve with many right-hand sides. The issues related to reuse of data in the caches is similar to those for GEMM: If small enough, $L$ can be kept in one of the caches (typically the
L2 cache), so that for each triangular solve the cost of bringing elements of $L$ into the registers is greatly reduced.

**In summary**, there is an opportunity to overcome the cost of a memory operation by casting computation in terms of matrix-matrix operations like GEMM, SYRK, and TRSM since they allow this overhead to be amortized over a large amount of useful computation (flops). This motivates the so-called blocked algorithms described later.

**Exercise 2.6.** Let $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times k}$, and $B \in \mathbb{R}^{k \times n}$. Justify the names “rank-1 update” and “rank-k update” by showing that matrices of the form $xy^T$ and $AB^T$ have ranks of at most one and $k$, respectively.

### 3. Cholesky Factorization: Basics

In this section, we introduce the Cholesky factorization. We present a very quick review of what this operation entails, the theory that supports it, the classical derivation of one algorithm for computing it, how to attain high performance by casting the computation in terms of matrix-matrix operations, and finally our notation for expressing the algorithms.

#### 3.1. Cholesky Factorization Theorem

We will prove the following theorem in Section 3.4:

**Theorem 3.1 (Cholesky Factorization Theorem).** Given a SPD matrix $A$ there exists a lower triangular matrix $L$ such that $A = LL^T$.

The lower triangular matrix $L$ is known as the Cholesky factor and $LL^T$ is known as the Cholesky factorization of $A$. It is unique if the diagonal elements of $L$ are restricted to be positive.

Naturally, an upper triangular matrix can be computed instead:

**Corollary 3.2.** Given SPD matrix $A$ there exists a upper triangular matrix $U$ such that $A = U^T U$.

The operation that overwrites the lower triangular part of matrix $A$ with its Cholesky factor will be denoted by $A := \Gamma(A)$, which should be read as “$A$ becomes its Cholesky factor.” Typically, only the lower (or upper) triangular part of $A$ is stored, and it is that part that is then overwritten with the result. In this discussion, we will assume that the lower triangular part of $A$ is stored and overwritten.

#### 3.2. Application

The Cholesky factorization is used to solve the linear system $Ax = y$ when $A$ is SPD:

$$Ax = (L^T x) = Lz = y.$$  

Thus, $z$ can be computed by solving the triangular system of equations $Lz = y$, after which the desired solution $x$ can be computed by solving the triangular linear system $L^T x = z$.

#### 3.3. Unblocked algorithm

The most common (right-looking) algorithm for computing $A := \Gamma(A)$ can be derived as follows: Consider $A = LL^T$. Partition

$$A = \begin{pmatrix} a_{11} & * \\ a_{21} & A_{22} \end{pmatrix} \quad \text{and} \quad L = \begin{pmatrix} l_{11} & 0 \\ l_{21} & L_{22} \end{pmatrix}.$$  

By substituting these partitioned matrices into $A = LL^T$ we find that

$$\begin{pmatrix} a_{11} & * \\ a_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} \lambda_{11} & 0 \\ l_{21} & L_{22} \end{pmatrix} \begin{pmatrix} \lambda_{11} & 0 \\ l_{21} & L_{22} \end{pmatrix}^T = \begin{pmatrix} \lambda_{11}^2 & \lambda_{11}^2 l_{21}^2 + L_{22}^2 \\ \lambda_{11} l_{21} & l_{21} l_{21}^T + L_{22} L_{22}^T \end{pmatrix}.$$
for $j = 1 : n$
\[ \alpha_{j,j} := \sqrt{\alpha_{j,j}} \]
endfor

for $i = j + 1 : n$
\[ \alpha_{i,j} := \alpha_{i,j}/\alpha_{j,j} \]
endfor

for $k = j + 1 : n$
for $i = k : n$
\[ \alpha_{i,k} := \alpha_{i,k} - \alpha_{i,j}\alpha_{k,j} \]
endfor
endfor

\[ \text{Fig. 3.1. Formulations of the Cholesky factorization that expose indices.} \]

from which we conclude that
\[ \lambda_{11} = \sqrt{\alpha_{11}} \quad l_{21} = a_{21}/\lambda_{11} \quad L_{22} = \Gamma(A_{22} - l_{21}l_{21}^T) \cdot \]

These equalities motivate the algorithm
\begin{enumerate}
\item Partition $A \rightarrow \begin{pmatrix} \alpha_{11} & * \\ a_{21} & A_{22} \end{pmatrix}.$
\item Overwrite $\alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11}}.$
\item Overwrite $a_{21} := l_{21} = a_{21}/\lambda_{11}.$
\item Overwrite $A_{22} := A_{22} - l_{21}l_{21}^T$ (updating only the lower triangular part of $A_{22}$).
\item Continue with $A = A_{22}.$ (Back to Step 1.)
\end{enumerate}

The algorithm is typically presented in a text using Matlab-like notation as illustrated in Fig. 3.1. In those algorithms we start indexing of elements in the matrix at “1” to be consistent with Matlab.

3.4. Proof of the Cholesky Factorization Theorem. We now show that if $A \in \mathbb{R}^{n \times n}$ is SPD, then the algorithm presented in Section 3.3 is guaranteed to executed to completion and therefore computes the Cholesky factor, which then in turn proves Theorem 3.1.

**Lemma 3.3.** Partition SPD matrix $A \rightarrow \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix}.$ Then $\alpha_{11} > 0.$

**Proof:** Choose $x = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ where 0 represents a vector of zeroes “of appropriate size”. Then
\[ 0 < x^T A x = \begin{pmatrix} 1 \\ 0 \end{pmatrix}^T \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \alpha_{11}. \]

**Exercise 3.1.** Show that all diagonal elements of a SPD matrix $A$ are positive.

Next, we state and prove a lemma that shows that matrix $A_{22}$ is again SPD after being updated with the symmetric rank-1 update in the algorithm in Section 3.3.

**Lemma 3.4.** Partition SPD matrix $A \in \mathbb{R}^{n \times n}$ as
\[ A \rightarrow \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix} \]
and let \( l_{21} = a_{21}/\sqrt{\alpha_{11}} \). Then \( A_{22} - l_{21}L_{21}^T \) is SPD.

**Proof:** Note that \( l_{21} \) is well-defined since (by Lemma 3.3) \( \alpha_{11} > 0 \). Since \( A \) is symmetric so are \( A_{22} \) and \( A_{22} - l_{21}L_{21}^T \). Let nonzero \( x_2 \in \mathbb{R}^{n-1} \) be an arbitrary nonzero vector and define \( x = \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix} \) where \( \chi_1 = -x_2^T a_{21}/\alpha_{11} \). Then \( x \neq 0 \) and thus

\[
0 < x^T A x = \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix}^T \begin{pmatrix} \alpha_{11} & a_{21}^T \\ a_{21} & A_{22} \end{pmatrix} \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix} = x_2^T (A_{22} - \frac{a_{21}a_{21}^T}{\alpha_{11}}) x_2
\]

We conclude that \( A_{22} - l_{21}L_{21}^T \) is SPD. \( \square \)

These two lemma show that the algorithm given at the end of Section 3.3 will execute to completion, since the updates 1.–4. are well-defined, and the resulting updated \( A_{22} \) is itself again SPD. We formalize these observations in the proof of the Cholesky Factorization Theorem.

**Proof:** [Cholesky Factorization Theorem] Proof by induction.

**Base case:** \( n = 1 \) Clearly the result is true for a \( 1 \times 1 \) matrix, \( A = \alpha_{11} \): In this case, the fact that \( A \) is SPD means that \( \alpha_{11} > 0 \) and its Cholesky factor is then given by \( \lambda_{11} = \sqrt{\alpha_{11}} \).

**Inductive step:** Induction hypothesis – Assume the result is true for all \( k \times k \) SPD matrices. We will show that it holds for \( A \in \mathbb{R}^{(k+1) \times (k+1)} \). Let \( A \in \mathbb{R}^{(k+1) \times (k+1)} \) be SPD. Partition \( A \) and \( L \) as in (3.1) and let \( \lambda_{11} = \sqrt{\alpha_{11}} \) (which is well-defined by Lemma 3.3), \( l_{21} = a_{21}/\alpha_{11} \), and \( L_{22} = \Gamma(A_{22} - l_{21}L_{21}^T) \) (which exists thanks to Lemma 3.4 and the induction hypothesis). Then \( L \) is the desired Cholesky factor of \( A \).

**By the principle of mathematical induction,** the theorem holds. \( \square \)

**Exercise 3.2.** Let \( A = LL^T \) represent the Cholesky factorization of SPD matrix \( A \). Then \( L \) is nonsingular.

**3.5. Blocked algorithm.** A blocked version of the algorithm, which casts most computation in terms of matrix-matrix operations, can be derived by partitioning

\[
A \rightarrow \begin{pmatrix} A_{11} & \ast \\ A_{21} & A_{22} \end{pmatrix} \quad \text{and} \quad L \rightarrow \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix},
\]

where \( A_{11} \) and \( L_{11} \) are \( b \times b \). By substituting these partitioned matrices into \( A = LL^T \) we find that

\[
\begin{pmatrix} A_{11} & \ast \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix}^T = \begin{pmatrix} L_{11}L_{11}^T & \ast \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{pmatrix}.
\]

From this we conclude that

\[
\begin{pmatrix} L_{11} = \Gamma(A_{11}) \\ L_{21} = A_{21}L_{11}^T \\ L_{22} = \Gamma(A_{22} - L_{21}L_{21}^T) \end{pmatrix}.
\]

An algorithm is then described by the steps

1. Partition \( A \rightarrow \begin{pmatrix} A_{11} & \ast \\ A_{21} & A_{22} \end{pmatrix} \), where \( A_{11} \) is \( b \times b \).
2. Overwrite \( A_{11} := L_{11} = \Gamma(A_{11}) \).
for $j = 1 : n$ in steps of $n_b$
  \[ b := \min(n - j + 1, n_b) \]
  \[ \alpha_{j+j+b-1,j+j+b-1} := \Gamma(\alpha_{j+j+b-1,j+j+b-1}) \quad \text{(CHOL)} \]
  \[ \alpha_{j+b,n,j+b-1} := \alpha_{j+b,n,j+b-1} - \alpha_{j,j+b-1}^{T} \quad \text{(TRSM)} \]
  \[ \alpha_{j+b,n,j+b} := \alpha_{j+b,n,j+b} - \text{TRIL}(\alpha_{j+b,n,j+b-1}^{T}) \quad \text{(SYRK)} \]
endfor

**Fig. 3.2.** Blocked algorithm for computing the Cholesky factorization. Here $n_b$ is the block size used by the algorithm.

3. Overwrite $A_{21} := L_{21} = A_{21} L_{11}^{T}$.
4. Overwrite $A_{22} := A_{22} - L_{21} L_{21}^{T}$ (updating only the lower triangular part).
5. Continue with $A = A_{22}$. (Back to Step 1.)

An algorithm that explicitly indexes into the array that stores $A$ is given in Fig. 3.2.

**Remark 2.** The Cholesky factorization $A_{11} := L_{11} = \Gamma(A_{11})$ can be computed with the unblocked algorithm or by calling the blocked Cholesky factorization algorithm recursively.

**Exercise 3.3.** Prove that the blocked algorithm will complete if $A$ is a SPD matrix. (Hint: develop lemmas similar to Lemma 3.3 and 3.4.)

4. **Alternative Representation of Algorithms.** When explaining the algorithms discussed in Section 3. in a classroom setting, invariably they are accompanied by a picture sequence like the one in Fig. 4.1(left) and the (verbal) explanation:

- **Beginning of iteration:** At some stage of the algorithm (Top of the loop), the computation has moved through the matrix to the point indicated by the thick lines. Notice that we have finished with the parts of the matrix that are in the top-left, top-right (which is not to be touched), and bottom-left quadrants. The bottom-right quadrant has been updated to the point where we only need to perform a Cholesky factorization of it.
- **Repartition:** We now repartition the bottom-right submatrix to expose $a_{11}$, $a_{21}$, and $A_{22}$.
- **Update:** $a_{11}$, $a_{21}$, and $A_{22}$ are updated as discussed before.
- **End of iteration:** The thick lines are moved, since we now have completed more of the computation, and only a factorization of $A_{22}$ (which becomes the new bottom-right quadrant) remains to be performed.
- **Continue:** The above steps are repeated until the submatrix $A_{BR}$ is empty.

To motivate our notation, we annotate this progression of pictures as in Fig. 4.1(right). In those pictures, “T”, “B”, “L”, and “R” stand for “Top”, “Bottom”, “Left”, and “Right”, respectively. This then motivates the format of the algorithm in Fig. 4.2(left). A similar explanation can be given for the blocked algorithm, which is given in Fig. 4.2(right). In the algorithms, $m(A)$ indicates the number of rows of matrix $A$.

**Remark 3.** The indices in our more stylized presentation of the algorithms are subscripts rather than indices in the conventional sense.

Fig. 4.2 does not present the algorithm as concisely as the algorithms given in Figs. 3.1 and 3.2. However, it does capture to a large degree the verbal description of the algorithm mentioned above and therefore, in our opinion, reduces both the effort required to interpret the algorithm and the need for additional explanations. The notation also mirrors that used for the proof in Section 3.4.
Exercise 4.1. We provide a number of macros for typesetting algorithms with BTPeX. Familiarize yourself with these tools by visiting http://www.cs.utexas.edu/users/flame/Tutorial/Cholesky/.

5. More Algorithms. In this section we present two more algorithms for Cholesky factorization.
Algorithm: $A := \text{CHOL}_\text{UNB}(A)$

Partition $A \rightarrow \begin{pmatrix} \hat{A}_{TL} & \star \\ \hat{A}_{BL} & A_{BR} \end{pmatrix}$

where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do

Repartition $\begin{pmatrix} \hat{A}_{TL} & \star \\ \hat{A}_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} a_{00} & \star & \star \\ a_{10} & \alpha_{11} & \star \\ a_{20} & a_{21} & A_{22} \end{pmatrix}$

where $a_{11}$ is $1 \times 1$

$\alpha_{11} := \sqrt{\alpha_{11}}$
$a_{21} := a_{21}/\alpha_{11}$
$A_{22} := A_{22} - \text{TRIL}(a_{21}a_{21}^T)$

Continue with $\begin{pmatrix} \hat{A}_{TL} & \star \\ \hat{A}_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} a_{00} & \star & \star \\ a_{10} & 0 & 0 \\ a_{20} & 0 & 0 \end{pmatrix}$

endwhile

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

Partition $A \rightarrow \begin{pmatrix} \hat{A}_{TL} & \star \\ \hat{A}_{BL} & A_{BR} \end{pmatrix}$

where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do

Determine block size $b$

Repartition $\begin{pmatrix} \hat{A}_{TL} & \star \\ \hat{A}_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} a_{00} & \star & \star \\ a_{10} & A_{11} & \star \\ a_{20} & a_{21} & A_{22} \end{pmatrix}$

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

$A_{11} := \Gamma(A_{11})$
$A_{21} := A_{21} \text{TRIL}(A_{11})^T$
$A_{22} := A_{22} - \text{TRIL}(A_{21}A_{21}^T)$

Continue with $\begin{pmatrix} \hat{A}_{TL} & \star \\ \hat{A}_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} a_{00} & \star & \star \\ a_{10} & A_{11} & \star \\ a_{20} & A_{21} & A_{22} \end{pmatrix}$

endwhile

Fig. 4.2. Unblocked and blocked algorithms for computing the Cholesky factorization.

5.1. “Bordered algorithm”. An algorithm that is sometimes referred to as the “bordered variant” can be derived as follows: Partition

$A \rightarrow \begin{pmatrix} A_{00} & \star \\ a_{10} & \alpha_{11} \end{pmatrix}$ and $L \rightarrow \begin{pmatrix} L_{00} & 0 \\ l_{10} & \lambda_{11} \end{pmatrix}$.

Then

$\begin{pmatrix} A_{00} & \star \\ a_{10} & \alpha_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ l_{10} & \lambda_{11} \end{pmatrix}^T \begin{pmatrix} L_{00} & 0 \\ l_{10} & \lambda_{11} \end{pmatrix} = \begin{pmatrix} L_{00}L_{00}^T & 0 \\ l_{10}l_{10}^T & l_{10}l_{10}^T + \lambda_{11}^2 \end{pmatrix}$

from which we conclude that

$L_{00} = \Gamma(A_{00})$
$l_{10} = L_{00}^{-1}a_{10}$
$\lambda_{11} = \sqrt{\alpha_{11} - l_{10}l_{10}^T}$

This in turn motivates the algorithm in Fig. 5.1(left) where only the updates related to “Variant 1: (Bordered)” are executed.

Exercise 5.1. Prove that when the unblocked Variant 1 is executed on a SPD matrix, the algorithm will complete. (Hint: Develop lemmas similar to Lemmas 3.3 and 3.4.)

Exercise 5.2. Derive the blocked Variant 1 in Fig. 5.1(right).

Exercise 5.3. Prove that when the blocked Variant 1 is executed on a SPD matrix, the algorithm will complete. (Hint: Develop lemmas similar to Lemmas 3.3 and 3.4.)

The aggregate number of floating point operations (flops) that are performed by each of the major suboperations will become important when performance is discussed in Section 7. For the unblocked algorithm, the iteration when $A_{TL} \in \mathbb{R}^{j \times j}$ spends approximately $j^2$ flops in Trsv and $2j$ flops in Dot. Thus, the unblocked algorithm,
when factoring $A \in \mathbb{R}^{n \times n}$, spends approximately $\sum_{j=0}^{n-1} j^2 \approx n^3/3$ in TRSV and $\sum_{j=0}^{n-1} j \approx n^2/2$ in DOT. For the blocked algorithm, the iteration when $A_{TL} \in \mathbb{R}^{j \times j}$, with $j = Jb$ for some integer $J$, spends approximately $j^3b$ flops in TRSM, $jb^2$ flops in SYRK, and $b^3/3$ flops in $\Gamma(A_{11})$. Thus, the blocked algorithm, if $n = Nb$ where $N$ is an integer, are spends approximately $\sum_{j=0}^{N-1} (Jb)^3b \approx b^3N^3/3 = n^3/3$ in TRSM, $\sum_{j=0}^{N-1} (Jb)b^2 \approx b^3N^2/2 = bn^2/2$ in SYRK, and $Nb^3/3 = nb^2/3$ in $\Gamma(A_{11})$. We thus note that the performance of TRSV and TRSM are critical for the unblocked and blocked algorithms, respectively.

5.2. “Left-looking algorithm”. An algorithm that is sometimes referred to as the “left-looking variant” can be derived as follows: Consider the partitionings

\[
\begin{align*}
\text{Algorithm: } & A := \text{Chol}_\text{UNB}(A) \\
\text{Partition } & A \rightarrow \begin{pmatrix} A_{TL} & * \\ A_{BL} & A_{BR} \end{pmatrix} \\
\text{where } & A_{TL} \text{ is } 0 \times 0 \\
\text{while } & m(A_{TL}) < m(A) \text{ do} \\
\text{Repartition } & \begin{pmatrix} A_{TL} & * \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & * & * \\ a_{10} & a_{11} & * \\ A_{20} & a_{21} & A_{22} \end{pmatrix} \\
\text{where } & a_{11} \text{ is } 1 \times 1 \\
\text{Variant 1: (Bordered)} & a_{10} := a_{10} \text{TRIL}(A_{00})^{-T} \\
& a_{11} := \sqrt{a_{11} - a_{10}a_{10}} \\
\text{Variant 2: (Left-looking)} & a_{21} := (a_{21} - A_{20}a_{10})/a_{11} \\
\text{Variant 3: (Right-looking)} & a_{22} := A_{22} - \text{TRIL}(a_{21}a_{21})T \\
\text{Continue with } & \begin{pmatrix} A_{TL} & * \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & * & * \\ a_{10} & a_{11} & * \\ A_{20} & a_{21} & A_{22} \end{pmatrix} \\
\text{endwhile} \\
\end{align*}
\]

\[
\begin{align*}
\text{Algorithm: } & A := \text{Chol}_\text{BLK}(A) \\
\text{Partition } & A \rightarrow \begin{pmatrix} A_{TL} & * \\ A_{BL} & A_{BR} \end{pmatrix} \\
\text{where } & A_{TL} \text{ is } 0 \times 0 \\
\text{while } & m(A_{TL}) < m(A) \text{ do} \\
\text{Determine block size } & b \\
\text{Repartition } & \begin{pmatrix} A_{TL} & * \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & * & * \\ A_{10} & A_{11} & * \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \\
\text{where } & A_{11} \text{ is } b \times b \\
\text{Variant 1: (Bordered)} & A_{10} := A_{10} \text{TRIL}(A_{00})^{-T} \\
& A_{11} := \Gamma(A_{11} - \text{TRIL}(A_{10}A_{10}T)) \\
\text{Variant 2: (Left-looking)} & A_{21} := (A_{21} - A_{20}A_{10}) \text{TRIL}(A_{11})^{-T} \\
\text{Variant 3: (Right-looking)} & A_{22} := A_{22} - \text{TRIL}(A_{21}A_{21})T \\
\text{Continue with } & \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & * & * \\ A_{10} & A_{11} & * \\ A_{20} & A_{21} & A_{22} \end{pmatrix} \\
\text{endwhile} \\
\end{align*}
\]

\[
\text{Fig. 5.1. Unblocked and blocked algorithms for computing the Cholesky factorization.}
\]
Now, consider Variant 2 in Fig. 5.1(left). It assumes that at the top of the loop $A_{TL}$ and $A_{BL}$ have already been overwritten with $L_{TL}$ and $L_{BL}$ and that $A_{BR}$ has not been changed at all. The update then overwrites
\[
\alpha_{11} := \lambda_{11} = \sqrt{\alpha_{11} - t_{10}^T l_{10}}
\]
\[
a_{21} := l_{21} = (a_{21} - L_{20} l_{10})/\lambda_{11},
\]
which can be easily deduced from (5.1) and (5.2). At the bottom of the loop it is again the case that the new $A_{TL}$ and $A_{BL}$ contain $L_{TL}$ and $L_{BL}$ while $A_{BR}$ still has not been changed at all.

Exercise 5.4. Prove that when the unblocked Variant 2 is executed on a SPD matrix, the algorithm will complete.

Exercise 5.5. Similarly justify the blocked Variant 2 in Fig. 5.1(right).

Exercise 5.6. Prove that when the blocked Variant 2 is executed on a SPD matrix, the algorithm will complete.

For the unblocked algorithm, the iteration when $A_{TL} \in \mathbb{R}^{J \times J}$ spends approximately $2j$ flops in DOT, $2(n - j)j$ flops in GEMV, and $j$ flops in SCAL. Thus, the unblocked algorithm, when factoring $A \in \mathbb{R}^{n \times n}$, are spends almost all flops in GEMV: $\sum_{j=0}^{n-1} (n-j)j \approx n^3/3$ flops. For the blocked algorithm, the iteration when $A_{TL} \in \mathbb{R}^{J \times J}$, with $j = Jb$ for some integer $J$, spends approximately $jb^2$ flops in SYRK, $(n-j-b)j b$ flops in GEMM, $b^3/3$ flops in $\Gamma(A_{11})$, and $(n-j-b)b^2$ flops in TRSM. Thus, the blocked algorithm, if $n = Nb \gg b$ where $N$ is an integer, are spends almost all flops in GEMM: $\sum_{j=0}^{N-1} (n-Jb-b)(Jb)b \approx n^3/3$ flops. We thus note that the performance of GEMV and GEMM are critical for the unblocked and blocked algorithms, respectively.

5.3. “Right-looking” algorithm. The algorithms discussed in Sections 3.3 and 3.5 are sometimes referred to as “right-looking” algorithms [6]. In Fig. 5.1 these algorithms appear as Variant 3.

For the unblocked algorithm, the iteration when $A_{TL} \in \mathbb{R}^{J \times J}$ spends approximately $j$ flops in SCAL and $(n-j)^2$ flops in SYR. Thus, the unblocked algorithm, when factoring $A \in \mathbb{R}^{n \times n}$, are spends almost all flops in SYR: $\sum_{j=0}^{n-1} (n-j)^2 \approx n^3/3$ flops. For the blocked algorithm, the iteration when $A_{TL} \in \mathbb{R}^{J \times J}$, with $j = Jb$ for some integer $J$, spends approximately $b^3/3$ flops in $\Gamma(A_{11})$, $(n-j-b)b$ flops in TRSM, and $(n-j-b)b^2$ flops in SYRK. Thus, the blocked algorithm, if $n = Nb \gg b$ where $N$ is an integer, spends almost all flops in SYRK: $\sum_{j=0}^{N-1} (n-Jb-b)^2b \approx n^3/3$ flops. We thus note that the performance of SYR and SYRK are critical for the unblocked and blocked algorithms, respectively.

5.4. Compare and contrast. The three different algorithms sweep through matrix $A$ in the same way, which is obvious from the fact that each algorithm only differs in the update that is performed in the body of the loop. We believe that the notation we use to present algorithms facilitates the comparing and contrasting of the different algorithms for this operation. In [3] we show the notation similarly supports, for a broad range of linear algorithms, the comparing and contrasting of different algorithms for the same operations as well as similar algorithms for different operations.
One way to qualify the different algorithms is to compare the state (contents) of \( A \) before and after each iteration of the loop. These are given in Fig. 5.2.

**Remark 4.** In [1] we show how algorithms for the Cholesky factorization can be systematically derived from the contents that are to be maintained in matrix by the algorithm.

**Exercise 5.7.** In Sections 5.3–5.1 the cost of the different variants was analyzed by approximating the results of summations. Perform an exact analysis of the costs of the different algorithms to show that all perform exactly the same number of flops.

**Remark 5.** It can be shown that all algorithmic variants for the Cholesky factorization that we have discussed perform exactly the same operations, but in a different order.

---

### 6. Representing Algorithms in Code

We now discuss how to translate the algorithms discussed so far into code.

#### 6.1. Basic Linear Algebra Subprograms (BLAS)

Very early on, the linear algebra community recognized the benefit of defining interfaces to subroutines that perform frequently encountered linear algebra operations like those listed in Fig. 2.2. The first interface set targeted vector-vector operations and are now known as the level-1 Basic Linear Algebra Subprograms (BLAS) [7]. Later it was recognized that better performance can be attained if algorithms are cast in terms of matrix-vector operations or, better yet, matrix-matrix operations. This led to the level-2 and level-3...
6.2. Traditional implementations in Fortran. As we mentioned, algorithms are often presented as in Figs. 3.1 and 3.2 since these representations are closer to how they are traditionally coded. In Fig. 6.1 we show Fortran code for the blocked Variant 3 in Fig. 3.2, written in a style similar to that used to code a popular linear algebra library, the Linear Algebra Package (LAPACK). Notice that the operations Sqrt, Scal, and Syrk are performed by the subroutines dsqrt, dscal, and dsyrk. While the reader may recognize the algorithm, especially given the comments we placed in the code, the translation is clearly complicated by the indexing into the array that stores $A$.

Exercise 6.1. Visit http://www.cs.utexas.edu/users/flame/Tutorial/Cholesky/ for a number of exercises related to the translation of the Cholesky factorization algorithms to LAPACK-like code. This exercise will give the reader a taste of the complexity that is avoided by coding with the FLAME APIs. (Note: it is best to do Exercise 6.2 and/or 6.3 first.)

6.3. FLAME implementations for M-script and C. The LAPACK style of coding these kinds of algorithms contrasts sharply with the FLAME style of representing algorithms in code. In Fig. 6.2 we show how the FLAME API for Matlab’s M-script is used to translate the algorithm in Fig. 4.2(right) into code. Similarly, in Fig. 6.2 FLAME API for the C programming language is illustrated for the same algorithm. We believe the code to closely resemble the algorithms, requiring little further explanation.

Exercise 6.2. Visit http://www.cs.utexas.edu/users/flame/Tutorial/Cholesky/ for a number of exercises related to the translation of the Cholesky factorization algorithms with the FLAME API for M-script.

Exercise 6.3. Visit http://www.cs.utexas.edu/users/flame/Tutorial/Cholesky/ for a number of exercises related to the translation of the Cholesky factorization algorithms with the FLAME API for C.

7. Attaining High Performance. NOTE: THE PAPER HAS NOT BEEN EDITED CAREFULLY BEYOND THIS. INDEED, IT IS QUITE INCOMPLETE.

In this section we discuss how performance is affected by the interaction between algorithms and architecture.

Something about the architecture.

To illustrate the relative performance benefits of the different algorithms we coded unblocked and blocked versions of all three variants in two styles: LAPACK-style: Using explicit indexing into arrays as in Fig. ??, FLAME-style: Using the FLAME/C API as in Fig. ?? In each case the blocked variant called the same unblocked variant for the smaller subproblem $A_{11} := \Gamma(A_{11})$ (e.g., blocked Variant 1 calls unblocked Variant 1 for that subproblem). In addition, we report performance for the case where Variant 3 is used for the blocked algorithm, which then calls unblocked Variant 2, which we will see is the best option for smaller matrices.

Fig. ?? reports typical performance for unblocked algorithms. The “hump” in the performance graph occurs for problem sizes where all or most of the matrix $A$ fits in the L2 cache. There, even matrix-vector operations achieve reasonable performance
function [ A_out ] = Chol_blk_var3( A, nb_alg )
[ ATL, ATR, ... 
  ABL, ABR ] = FLA_Part_2x2( A, 0, 0, 'FLA_TL' );

while ( size( ATL, 1 ) < size( A, 1 ) )
  b = min( size( ABR, 1 ), nb_alg );
  [ A00, A01, A02, ...
    A10, A11, A12, ...
    A20, A21, A22 ] = FLA_Repart_2x2_to_3x3( ATL, ATR, ... 
                                                ABL, ABR, ... 
                                                b, b, 'FLA_BR' );
  A11 = Chol_unb( A11 );
  A21 = A21 * inv( tril( A11 )' );
  A22 = A22 - tril( A21 * A21' );
%
[ ATL, ATR, ... 
  ABL, ABR ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02, ... 
                                         A10, A11, A12, ... 
                                         A20, A21, A22, ...
                                         'FLA_TL' );
end

A_out = [ ATL, ATR 
          ABL, ABR ];
return

Fig. 6.2. Blocked Variant 3 represented with the FLAME@lab API for Matlab’s M-script.

#include "FLAME.h"

int Chol_blk_var3( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR, 
  A00, A01, A02, 
  ABL, ABR, 
  A10, A11, A12, 
  A20, A21, A22;

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

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ){
    b = min( FLA_Obj_length( ABR ), nb_alg );
    FLA_Repart_2x2_to_3x3( ATL, &ATR, 
                          &A00, &A01, &A02, 
                          &A10, &A11, &A12, 
                          &A20, &A21, &A22, 
                          b, b, FLA_BR );
    FLA_Chol_unb_var3( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR, 
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG, ONE, A11, A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, 
              FLA_NONUNIT_DIAG, "MINUS_ONE", A21, A22 );
  }
return FLA_SUCCESS;
}

Fig. 6.3. Blocked Variant 3 represented with the FLAME/C API for C.
since the cost of moving data from that cache to the registers takes only moderately more time than the cost of subsequently computing with the data so that much of the cost of the data movement can be masked with computation with previously fetched data. Unblocked Variant 2, which casts most of its computation in terms of GEMV, attains about twice the performance of unblocked Variants 1 and 3. This is due to the more favorable ratio of memops to flops for that operation, as was summarized in Fig. 2.2. Qualitatively the two graphs look almost identical, although
the “LAPACK-style” graph seems to attain better performance for smaller matrices. This is confirmed in Fig. 7.3(left) where it is reported how much slower the FLAME-style implementation is relative to the LAPACK-style implementation. For small problem sizes, the indexing overhead from the more stylized API comes at a cost.

Fig. ?? similarly reports performance for the blocked algorithms. Again, qualitatively the two graphs are similar. As is reported in Fig. 7.3(right) the FLAME-style implementation is slower than the LAPACK-style implementation, especially for smaller problems. In Fig. 7.4 we show that if the FLAME-style blocked Variant 3 calls the LAPACK-style unblocked Variant 2 is called, the performance is almost identical to the corresponding LAPACK-style blocked algorithm.

Remark 6. Until compilers are developed that can optimize FLAME-style code, those who are truly concerned about performance may want to code blocked algorithms using the FLAME/C API and interface those to unblocked algorithms implemented with the more traditional LAPACK-style of coding.

8. Conclusion.

REFERENCES


Appendix A. Proofs of Lemmas.

Proof: (Lemma ??) By induction.
Base case: $n = 1$: Follows trivially from Lemma ??.

Inductive step: Assume that the result is true for $A \in \mathbb{R}^{n \times n}$. We will show that the result is true for $A \in \mathbb{R}^{(n+1) \times (n+1)}$. Let $A \in \mathbb{R}^{(n+1) \times (n+1)}$. Partition $A \rightarrow \begin{pmatrix} \alpha_{11} & a_{12} \\ a_{21} & A_{22} \end{pmatrix}$. By Lemma ??, $\alpha_{11} > 0$. By Lemma ??, $A_{22}$ is SPD and therefore, by the induction hypothesis, all diagonal elements of $A_{22}$ are positive. Hence all diagonal elements of $A$ are positive.

By the principle of mathematical induction, the theorem holds. $\square$