Minimal-storage high-performance Cholesky factorization via blocking and recursion

by F. G. Gustavson
I. Jonsson

We present a novel practical algorithm for Cholesky factorization when the matrix is stored in packed format by combining blocking and recursion. The algorithm simultaneously obtains Level 3 performance, conserves about half the storage, and avoids the production of Level 3 BLAS for packed format. We use recursive packed format, which was first described by Andersen et al. [1]. Our algorithm uses only `DGEMM` and Level 3 kernel routines; it first transforms standard packed format to packed recursive lower row format. Our new algorithm outperforms the Level 3 LAPACK routine `DPOTRF` even when we include the cost of data transformation. (This is true for three IBM platforms—the POWER3, the POWER2, and the PowerPC 604e.) For large matrices, blocking is not required for acceptable Level 3 performance. However, for small matrices the overhead of pure recursion and/or data transformation is too high. We analyze these costs analytically and provide detailed cost estimates. We show that blocking combined with recursion reduces all overheads to a tiny, acceptable level. However, a new problem of nonlinear addressing arises. We use two-dimensional mappings (tables) or data copying to overcome the high costs of directly computing addresses that are nonlinear functions of *i* and *j*.

1. Introduction

We present a novel practical algorithm for Cholesky factorization by combining recursion and blocking. Our aim is to conserve storage and to simultaneously obtain Level 3 performance. Also, we want to avoid producing Level 3 BLAS for packed format.

In [1] a new data format called recursive packed format was used to produce a new algorithm for Cholesky factorization. We use that format to produce a Level 3 performing code using only `DGEMM` and Level 3 kernel routines. By so doing we achieve the algorithm alluded to

0018-8646/00/$5.00 © 2000 IBM

in the first paragraph. Our algorithm requires that the user's matrix AP, input in lower packed format [2, 3], first be transformed to packed recursive lower row format, PRLF. Our new algorithm is then applied to PRLF, which has overwritten the input matrix AP. Standard algorithms for Cholesky factorization are given in Version 3 of LAPACK. They are called DPPTRF and DPOTRF. The PP algorithm accepts input in packed format AP, while the PO algorithm requires input to be in full format, which requires about twice the storage of packed format. However, on many platforms, including RISC workstations, the LAPACK PO codes perform about three times faster. It is our view that most users prefer higher performance, so they use twice the storage. However, some applications can only run using half the storage, and these applications must use the packed routines.

Our new algorithm outperforms the LAPACK Level 3 DPOTRF even when we include the cost of the data transformation from packed lower format to packed recursive lower format. A slight drawback is that the data transformation requires a temporary buffer of size $\frac{1}{8}n^2$, which is allocated and then deallocated.

The new algorithm was implemented and tuned for the ESSL library, in which only lower packed format is supported. However, ESSL provides DPPF, which produces both an LL^T factorization (Cholesky) and an LDL^T factorization (Gauss). Additionally, like LAPACK, the ESSL library provides routine DPPICD, which computes the inverse matrix AP^{-1}. We mention that we produced new codes for LDL^T and matrix inverse for data in packed recursive lower row format. However, we describe only the Cholesky factorization LL^T in this paper.

For large matrices, blocking is not needed, because the pure recursive algorithm for Cholesky gives acceptable performance; see [1] for details. However, for small matrices the overhead of recursion slows down the performance to an unacceptable level for a high-performance library such as ESSL. This paper examines in detail the overheads related to pure recursion. We introduce blocking and combine it with recursion to overcome the losses due to the overhead by essentially reducing the overhead to a tiny acceptable amount. However, because the packed recursive formats do not allow for any data expansion, we are faced with a new problem: nonlinear addressing associated with packed recursive storage format. We solve this problem by one of two methods. The first is to use a two-dimensional mapping (table) whose (i, j) entry is the location in PRLF where $a(i, j)$ is stored. The other method is to data-copy a small submatrix A in PRLF to a buffer which stores A in full format so that standard linear addressing can be used. Additionally, our new algorithms require only that two mappings be generated, which results in only a tiny amount of additional storage. Also, since these mappings are used many times by the algorithm, the cost of their generation is amortized by their multiple reuse.

Our paper is organized into five sections, including the Introduction as Section 1. Section 2, called *Algorithmic considerations*, contains six subsections. In the first, we discuss the existing routines for doing Cholesky and LDL^T factorization on matrices stored in packed format. We give our recursive formulation of the Cholesky factorization, called CHOL, in the second subsection, together with sketches of the conventional LAPACK algorithms. In the third, a new data format is presented which enables routines to make better use of high-performance DGEMM routines. The fourth subsection shows how the DTRSM and DSYRK routines are affected by the recursive packed format. We produce recursive algorithms TRSM and SYRK and show proofs of correctness. In the fifth subsection, we explain why one of the formats gives a better data access pattern. Finally, in the sixth subsection, we provide an outline of how to transform data in standard packed format to recursive packed format. Section 3 is titled *Algorithmic components*. In the first subsection, we describe the problem of a large recursion tree; the second describes our blocked version and how we overcome the problem. We present block counterparts of CHOL, TRSM, and SYRK, which we call BC, BT, and BS, and we briefly develop a major theme of computer science—that an algorithm is tied to its data structure when we apply this theme to BC, BT, and BS. The third subsection describes three types of kernel routines, two of which are the mapping approach and the data entry approach described briefly above. The third type, called the compiled code approach, was not used.

We see in Section 4 that we obtain excellent Level 3 performance using recursive packed row format for Cholesky factorization, especially for large n. The new algorithm uses half the storage (an allocation of $\frac{1}{8}n^2$ elements is returned after the data transformation is complete) and performs better than most Level 3 implementations. Also, the majority of the processing is done in calls to DGEMM on submatrix blocks of variable size. In the SMP environment, ESSL's DGEMM automatically uses the existing threads to obtain excellent SMP performance. These two factors automatically make our new recursive Cholesky implementation an SMP parallel implementation.

In general, a negative feature of the recursive approach, and for this data format in particular, is that the addressing of the individual (i, j) elements in any triangle becomes nonlinear. This drawback is inevitable: It is ironic, however, that good data locality comes at the cost of making its reference patterns nonlinear.

In Section 5, we give our conclusions, summarizing the successful features of our new algorithm. We briefly indicate why packed format should become more

important, and the role recursion plays in its return to prominence. Next, we discuss some other results of our paper: why recursion is particularly suited to the Cholesky algorithm, results not covered, and issues related to Level 2 and Level 3 packed BLAS. Finally, we explain why our current algorithm is not the fastest-performing Cholesky routine.

2. Algorithmic considerations

In this section, we show why routines using the packed format have not displayed high performance. We show how to overcome this by using a recursive data format.

• *Preliminary remarks and rationale*

Existing codes for Cholesky and LDL^T factorization use either full storage or packed storage data format. In earlier times, when there was a uniform memory hierarchy, packed format was usually the method of choice because it conserved memory and performed about as well as a full-storage implementation. See the LINPACK User's Guide [4] for more details.

ESSL [3] and LAPACK [2] support both data formats, so a user can choose either for his application. Also, the ESSL packed storage implementation has always been Level 3, sometimes at great programming cost. On the other hand, LAPACK and some other libraries do not produce Level 3 implementations for packed data formats.

For portability and other reasons, mainly performance and ease of programming, users today generally use the full data format to represent their symmetric matrices. Nonetheless, saving half the storage is an important consideration, especially for those applications which will run with packed storage and fail to run with full storage.

Another important user consideration is migration. Many existing codes use one or the other or both of these formats. Producing an algorithm such as Cholesky using a new data format has little appeal, since existing massive programs cannot use the new algorithm. Thus, the approach we took in ESSL was to redo its packed Cholesky and LDL^T factorization codes but then instead use the new recursive packed data structure.

The idea was simple: Given AP holding symmetric A in lower packed storage mode, overwrite AP with A in the recursive packed row format. To do so requires a temporary array, which we allocate and then deallocate, of size $\frac{1}{8}n^2$. Next, we execute the new recursive Level 3 Cholesky algorithm using only the $\frac{1}{2}n^2$ original storage. Even when one includes the cost of converting the data from lower packed to recursive packed lower row format, the performance, as we will see, is better than that of the LAPACK Level 3 `DPOTRF`.

In summary, the users can now obtain full Level 3 performance using packed format and thereby save about half of their storage.

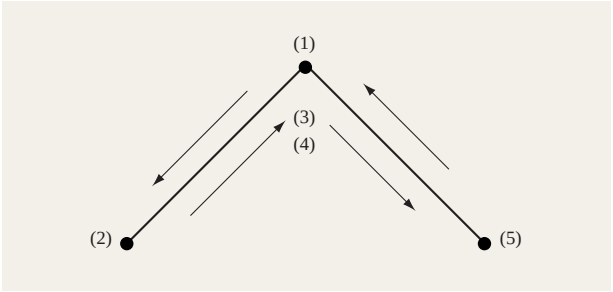

Figure 1

Generic node of the Cholesky factorization.

• *Recursive Cholesky factorization*

Our recursive algorithm of the Cholesky factorization of A uses a divide and conquer procedure, which gives rise to a binary tree. At every non-leaf node, the identical algorithm executes on one of the submatrices of A. In **Figure 1**, a parent node and its two children depict this situation.

Let a submatrix A of size n be associated with the parent node. Since A is symmetric, we are dealing with an isosceles right triangle of size n. Divide A into two congruent triangles of size $n/2$ and a square between them. The computation proceeds in the direction of the arrows, and Equation (1), which constitutes Equations (2) to (5) (see below), describes the computation that is done at the parent node. Now we formally describe our algorithm.

In our recursive algorithm, the Cholesky factorization of a positive definite symmetric $n \times n$ matrix A,

$$A = \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix} = LL^T = \begin{pmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ 0 & T_{22} \end{pmatrix}, \qquad (1)$$

is initiated by a recursive Cholesky factorization of the upper left square matrix A_{11} of order $n_1 = \lfloor n/2 \rfloor$; i.e.,

$$A_{11} = L_{11}L_{11}^T. \qquad (2)$$

The lower left matrix A_{21} is then transformed into L_{21} by the multiple solving of $n_2 = \lceil n/2 \rceil$ triangular systems of equations (each of size $n_1 = \lfloor n/2 \rfloor$),

$$L_{21}L_{11}^T = A_{21}. \qquad (3)$$

Then, the lower right square matrix of order n_2 is symmetrically rank-k updated by L_{21}; i.e.,

$$\tilde{A}_{22} = A_{22} - L_{21}L_{21}^T. \qquad (4)$$

Finally, the matrix $\tilde{A}_{22}$ is recursively factored,

$$\tilde{A}_{22} = L_{22}L_{22}^T. \qquad (5)$$

```
A(1:n, 1:n) = CHOL[A(1:n, 1:n)]
if(n = 1) then
  a(1, 1) = SQRT[a(1, 1)]
else ! n > 1
  n1 = n/2; j1 = n1 + 1; n2 = n - n1
  A(1:n1, 1:n1) = CHOL[A(1:n1, 1:n1)] !                          (2)
  Triangular solve for X = A(j1:n, 1:n1) in X A11^T = A(j1:n, 1:n1) !   (3)
  Update A(j1:n, j1:n) = A(j1:n, j1:n) - A(j1:n, 1:n1) · A(j1:n, 1:n1)^T !   (4)
  A(j1:n, j1:n) = CHOL[A(j1:n, j1:n)] !                          (5)
endif
```

Figure 2

Algorithm for recursive Cholesky factorization; A is lower triangular.

```
do j = 1, n, nb ! nb is the block size
   jb = min(n - j + 1, nb)
   call DSYRK['L', 'N', jb, j - 1, -one, A(j, 1), lda, one, A(j, j), lda]
   call DPOTF2['L', jb, A(j, j), lda, info]
   if (j + jb ≤ n) then
     call DGEMM['N', 'T', n - j - jb + 1, jb, j - 1, -one, A(j + jb, 1), lda, ...
                A(j, 1), lda, one, A(j + jb, j), lda]
     call DTRSM['R', 'L', 'T', 'N', n - j - jb + 1, jb, one, A(j, j), lda, A(j + jb, j), lda]
   endif
enddo
```

Figure 3

Algorithm for the DPOTRF subroutine.

```
jj = 1
do j = 1, n
   AP(jj) = DSQRT[AP(jj)]
   if (j < n) then
      call DSCAL[n - j, one/AP(jj), AP(jj + 1), 1]
      call DSPR['L', n - j, -one, AP(jj + 1), 1, AP(jj + n - j + 1)]
      jj = jj + n - j + 1
   endif
enddo
```

Figure 4

Algorithm for the DPPTRF subroutine.

The recursion in Equations (2) and (5) stops when the matrices to be factored A_{11}, A_{22} have small order or, if full recursion is used, have order 1.

Proof of correctness Our method of proof is mathematical induction. The recursion takes place on the order n of A. We want to prove correctness for $n = 1, 2, \cdots$. However, recursion breaks the problem into two nearly equal parts, $n_1 = \lfloor n/2 \rfloor$ and $n_2 = \lceil n/2 \rceil = n - n_1$. On the basis of this observation, we use mathematical induction on $k = \log_2 n$, $k = 0, 1, \cdots$.

Suppose that the result is true for $0 < n \leq 2^k$. Then we establish the result for all j, $2^k < j \leq 2^{k+1}$. Initially we need to establish the result for $n = 1$.

Now we give the proof: For $n = 1$, we compute $l_{11} = \sqrt{a_{11}}$. Since A is positive definite, we know all principal minors are positive, so l_{11} exists. Now suppose that the result is true for $0 < j \leq 2^k$. Let $2^k < j \leq 2^{k+1}$, $j_1 = \lfloor j/2 \rfloor$ and $j_2 = j - j_1$. Since $j > 1$, we do the computations indicated by Equations (2), (3), (4), and (5) in that order.

Equation (2) is satisfied by the induction hypothesis. Equation (3) is a calculation. It can be performed since the triangular matrix L_{11} has nonzero diagonal elements (in fact, they are positive). Equation (4) is also a calculation. Finally, Equation (5) is satisfied by the induction hypothesis. Using Equations (2)–(5), we want to show that Equation (1) is true. However, this is trivially true because all that is needed is to follow the rules of 2×2 block multiplication where the partitioning of A is $j = j_1 + j_2$. The conditions that the block elements of L_{11}, L_{21}, L_{22} must satisfy are exactly those of Equations (2)–(5), which we have shown to be true. □

Figure 2 gives the details of the recursive algorithm CHOL(n). We assume that A is an $n \times n$ positive definite symmetric matrix. The algorithm will detect that A is not positive definite symmetric in the if clause when and only when it finds $a_{ii} \leq 0$ for some i. In the else clause there are two recursive calls, one on matrix $A(1{:}n_1, 1{:}n_1)$, the other on matrix $A(j_1{:}n, j_1{:}n)$; the two computations solve $XA(1{:}n_1, 1{:}n_1)^T = A(j_1{:}n, 1{:}n_1)$ and update $A(j_1{:}n, j_1{:}n) = A(j_1{:}n, j_1{:}n) - A(j_1{:}n, 1{:}n_1) \cdot A(j_1{:}n, 1{:}n_1)^T$. Here we use colon notation; see page 19 of [5] for a definition. The two computations consist mostly of calls to DGEMM [actually, the first computation is DTRSM (multiple triangular solve) and the second is DSYRK (rank-k update of a symmetric matrix); however, both DTRSM and DSYRK consist mostly of calls to DGEMM]. The correctness of the algorithm CHOL follows immediately from our induction proof.

Figures 3 and **4** give annotated descriptions of the algorithms DPOTRF and DPPTRF. These two routines are the LAPACK Level 3 and Level 2 versions of Cholesky factorization, with *uplo* = 'L'. The former uses full storage and runs about three times faster than the latter, which uses packed storage.

The algorithm in Figure 3 is a block nb left-looking algorithm. The algorithm in Figure 4 is a Level 2 right-looking algorithm. In earlier versions of LAPACK, the algorithm DPOTRF was right-looking; i.e., it was the Level 3 block analog of Figure 4.

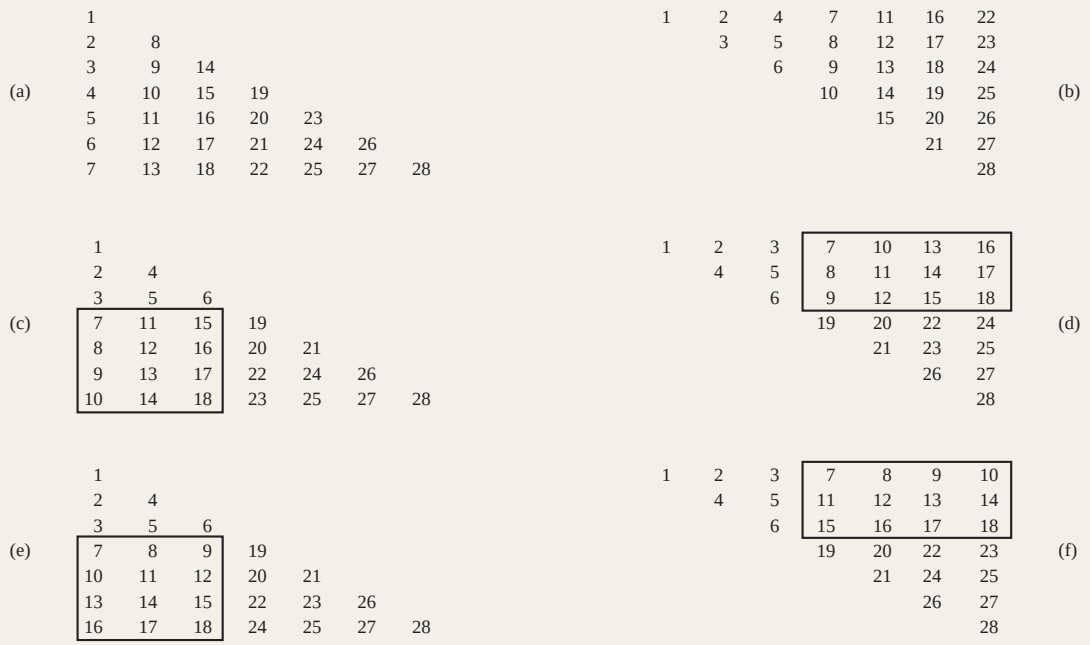

Figure 5

Six ways of storing the elements of an isosceles triangle of size 7: (a) Packed lower; (b) packed upper; (c) packed recursive column lower; (d) packed recursive column upper; (e) packed recursive row lower; (f) packed recursive row upper.

In Section 4 we compare the performance of DPOTRF and DPPTRF versus our new algorithm BC, which is the blocked form of the algorithm CHOL.

• *Packed recursive format*

In the previous subsection we gave the recursive Cholesky algorithm and proved its correctness for data assumed to be stored in full lower format. In this section we describe the new recursive packed formats. These data formats are modeled in the same way as the recursive blocked row and column formats of a triangle (see [6] for details). This modeling was first described by Andersen, Gustavson, and Waśniewski in [1]. It turns out that the recursive Cholesky algorithm will work on all four instances of this new data structure, referred to as *column lower*, *column upper*, *row lower*, and *row upper* (**Figure 5**). In each case, the algorithm and its proof are similar to the proof of the preceding subsection; hence, they are not repeated.

These packed recursive data formats are hybrid triangular formats consisting of $(n - 1)$ full-format rectangles of varying sizes and n triangles of size 1×1 on the diagonal. They use the same amount of data storage as the ordinary packed triangular format, i.e., $= n(n + 1)/2$. Because the rectangles (square submatrices) are in full format, it is possible to use high-performance Level 3 BLAS on the square submatrices. The difference between the formats is shown in Figure 5 for the special case $n = 7$.

Notice (as in Figure 6, shown later) that each of the original triangles is split into two triangles of sizes $n_1 = n/2$ and $n_2 = n - n_1$ and a rectangle of size $n_2 \times n_1$ for lower format and $n_1 \times n_2$ for upper format. The elements in the upper left triangle are stored first, the elements in the rectangle follow, and the elements in the lower right triangle are stored last. The order of the elements in each triangle is again determined by the recursive scheme of dividing the sides n_1 and n_2 by 2 and ordering these sets of points in the order *triangle*, *square*, *triangle*. The elements in the rectangle are stored in full format, either by row or by column.

Notice that we can store the elements of a rectangle in two different ways. The first is by column (standard Fortran order), and the second is by row (standard C order). Assume that A is in lower recursive packed format; then the rectangle is size $n_2 \times n_1$, $n_1 \leq n_2$. Suppose we store A by row; then $lda = n_1$, and the local address of the element $a(i, j)$ with respect to the beginning of the matrix A is

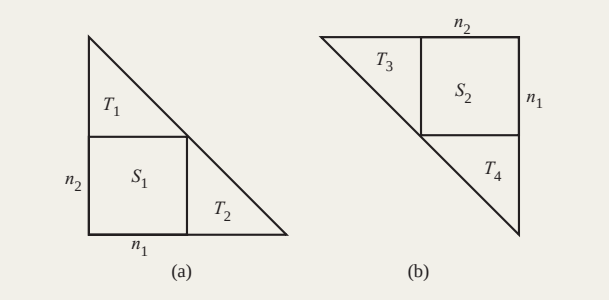

Figure 6

Correlation of the (a) L and (b) U formats of a symmetric matrix A.

$$loc[a(i, j)] = j + n_1 i. \tag{6}$$

Suppose we store A by column. Then $lda = n_2$, and

$$loc[a(i, j)] = i + n_2 j. \tag{7}$$

Next assume that A is in upper recursive packed format. Then the rectangle is size $n_1 \times n_2$, $n_1 \leq n_2$. Suppose we store A by row. Then $lda = n_2$, and

$$loc[a(i, j)] = j + n_2 i. \tag{8}$$

Suppose we store A by column. Then $lda = n_1$, and

$$loc[a(i, j)] = i + n_1 j. \tag{9}$$

Now, the storage layout for the matrix in L format is identical to the storage layout of the transpose matrix in U format. This can be seen by noting that (8) and (9) (for A^T) become

$$loc[at(i, j)] = i + n_2 j \tag{10}$$

and

$$loc[at(i, j)] = j + n_1 i. \tag{11}$$

We can now state a relationship among the four storage layouts of the lower and upper recursive data formats.

Theorem 1 The storage layout of the recursive lower packed row (column) data format is identical to the storage layout of the recursive upper packed column (row) data format.

Proof The algorithms for U and for L have the diagonal in common and hence define the diagonal elements in the same way. Alternatively, we can prove this by induction. Thus, the storage layout is identical for the two formats. Now consider the off-diagonal elements. We use induction. The induction hypothesis is for $k = \log_2 n$, $k = 0, 1, \cdots$. Assume that the result is true for $0 < j \leq 2^k$. Let $2^k < j \leq 2^{k+1}$. The induction hypothesis states that the layouts of T_1 and T_3 and of T_2 and T_4 are identical [**Figures 6(a)** and **6(b)**]. Now S_1 and S_2 start at the same location. By using Equations (6) and (11) we see that the row layout of S_1 is identical to the column layout of S_2, and by using Equations (7) and (10) we see that the column layout of S_1 is identical to the row layout of S_2. □

We have just seen that recursive row storage of L is identical to recursive column storage of U and that recursive column storage of L is identical to recursive row storage of U. We next examine an interesting consequence of this result.

Redundancy of symmetric-type algorithms leads to faster algorithms and less coding effort

Theorem 1 states that we have two essentially different formats for storing a symmetric matrix. Let us arbitrarily choose recursive packed lower column storage and recursive packed upper column storage as the two different formats. Next, note that most mathematical libraries, such as LAPACK, LINPACK, and ESSL, provide symmetric algorithms supporting both data formats. Because of symmetry, both algorithms perform exactly the same computation; the only difference between the algorithms is the way in which they access storage. Before continuing, consider running a symmetric algorithm in C instead of Fortran. Since C stores matrices by row instead of by column, the L algorithm of C becomes the U algorithm of Fortran, and vice versa. (We mention without proof that a theorem similar to Theorem 1 holds for full-format storage and hence in the Fortran and C programming languages.) Now we pose an interesting question: Of the two data formats, which one gives the faster execution for the symmetric algorithm? Several answers are possible, and the answer is dependent on the symmetric algorithm. However, the most natural answer is that the algorithms perform about the same for both data formats. Now, for definiteness, assume that the algorithm is the Cholesky factorization algorithm. In that case, we argue that one should always choose recursive row storage of L even if the input data for L is given in recursive lower column storage. The reason follows from the fact that computations done stride 1 are almost always faster than computations done stride n. Also, Cholesky factorization is a Level 3 computation, while matrix transposition is a Level 2 computation. Finally, if one expresses L in recursive lower row storage, the majority of Cholesky factorization will be done with stride 1 instead of stride n. The gain is multiplied by the Level 3/Level 2 ratio.

• *DTRSM and DSYRK using recursive packed format*

We now discuss adapting DTRSM and DSYRK to the new data structure. We examine a triangle T of size n spanning nt consecutive storage locations, where

$$nt = \frac{n(n+1)}{2}.$$

Without loss of generality, we can let T be lower triangular and occupy memory starting at 0 and ending at $nt - 1$. In global A, T starts at ist and ends at $ist + nt - 1$. Let $n = n_1 + n_2$, where $n_1 = \lfloor n/2 \rfloor$. By definition, T consists of two triangles T_1 and T_2 and a "square" S_1 of size $n_2 \times n_1$ [Figure 6(a)]. Hence, let T_1, T_2, and S_1 have "local" addresses in the space 0 to $nt - 1$. This means that T_1 has addresses 0 to $nt_1 - 1$, S_1 has addresses 0 to $n_2 n_1 - 1$, and T_2 has addresses 0 to $nt_2 - 1$, where

$$nt_1 = \frac{n_1(n_1+1)}{2}$$

and

$$nt_2 = \frac{n_2(n_2+1)}{2}.$$

In terms of T coordinates, T_1 starts at 0, S_1 starts at nt_1, and T_2 starts at

$$nt_1 + n_2 n_1 = \frac{n_1(n+n_2+1)}{2}.$$

It is clear from Figure 6(a) that T_1 and T_2 are stored recursively, whereas S_1 is stored in full format.

In order to use Level 3 BLAS DTRSM and DSYRK, we must have both the triangle (either T_1 or T_2) and the rectangle (S_1) in full format. In our case the triangle is not in full format. However, T_1 and T_2 consist respectively of $(n_1 - 1)$ and $(n_2 - 1)$ squares, and their total areas are respectively $nt_1 - n_1$ and $nt_2 - n_2$.

We propose that DTRSM and DSYRK be expressed recursively according to the recursive data layout of T_1 and T_2. Take DTRSM first. It requires T_1 and S as its operands. Now divide T_1 into T_{11}, S_1, and T_{12} according to its recursive definition. Let $A = T_1$ and $B = S$. The relation between Figure 6(a) and **Figure 7** is that $n_2 = m$ and $n_1 = k$. Now the computation $B = B \cdot A^{-T}$ becomes

$$XA^T = B, \tag{12}$$

or

$$m\{(\overbrace{X_1}^{k_1}\ \overbrace{X_2}^{k_2})\begin{pmatrix} A_{11}^T & A_{21}^T \\ 0 & A_{22}^T \end{pmatrix} = m\{(\overbrace{B_1}^{k_1}\ \overbrace{B_2}^{k_2}) = B.$$

If we break (12) into its component pieces, we obtain

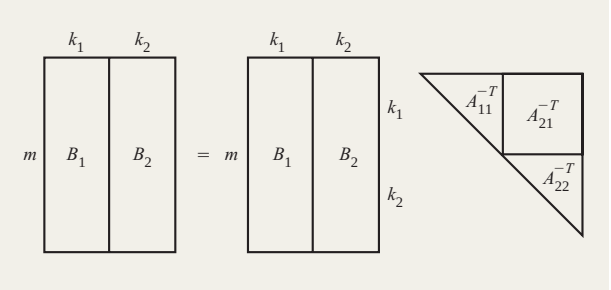

Figure 7

Block partitioning of the TRSM computation.

$$X_1 A_{11}^T = B_1, \tag{13}$$

$$\tilde{B}_2 = B_2 - X_1 A_{21}^T, \tag{14}$$

$$X_2 A_{22}^T = \tilde{B}_2. \tag{15}$$

Note that (14) is a DGEMM computation on matrices (A_{21}, B_2, B_1), which are stored in full format. Computations (13) and (15) are smaller instances of the original problem, i.e., a DTRSM computation where the triangle is stored recursively and the rectangle is stored in full format. It now follows that we can define DTRSM in our recursive data structure, and it comprises only calls to DGEMM and the Level 1 routine DSCAL. We now state the proof of the recursive DTRSM and give the full algorithm.

The recursive algorithm for the triangular solution resembles the Cholesky factorization algorithm. Assume that A is a nonsingular triangular matrix of size k and B is a rectangular matrix of size $m \times k$. Then the recursion starts by solving for X_1 using the upper left triangle A_{11} of order $k_1 = \lfloor k/2 \rfloor$; see (13). The lower matrix B_2 of order $k_2 = \lceil k/2 \rceil$ is transformed into $\tilde{B}_2$ by subtracting the product of the solved X_1 and lower left matrix A_{21}; see (14). Finally, we can recursively solve for X_2; see (15).

The recursion in (13) and (15) stops when the matrices to be solved, X_1 and X_2, have a small number of rows or, if full recursion is used, one row.

Proof of correctness Once again mathematical induction is used. The recursion takes place on the order k of A. We want to prove correctness for $k = 1, 2, \cdots$. Since recursion breaks the problem into two nearly equal parts, $k_1 = \lfloor k/2 \rfloor$ and $k_2 = \lceil k/2 \rceil = k - k_1$, we use mathematical induction on $i = \log_2 k$, $i = 0, 1, \cdots$.

Suppose that the result is true for $0 < k \le 2^i$. Then we establish the results for all j, $2^i < j \le 2^{i+1}$. As a base for our induction, we need to establish the result for $k = 1$.

For $k = 1$, we scale the row $X = B \cdot a_{11}^{-1}$. Since A is nonsingular, $a_{11} \neq 0$, so X exists. Assume that the result is

```
B(1:m, 1:k) = TRSM[A(1:k, 1:k), B(1:m, 1:k)]
if(k = 1) then
  B(1:m, 1) = B(1:m, 1)/a(1, 1) ! DSCAL operation
else ! k > 1
  k1 = k/2 and j1 = k1 + 1
  B(1:m, 1:k1) = TRSM[A(1:k1, 1:k1), B(1:m, 1:k1)] !                    (13)
  Update B(1:m, j1:k) = B(1:m, j1:k) − B(1:m, 1:k1) · A(j1:k, 1:k1)^T ! (14)
  B(1:m, j1:k) = TRSM[A(j1:k, j1:k), B(1:m, j1:k)] !                    (15)
endif
```

Figure 8

Algorithm for recursive triangular solve, $B \leftarrow B \cdot A^{-T}$. A is lower triangular, stored in packed recursive row format. B is stored in row major format.

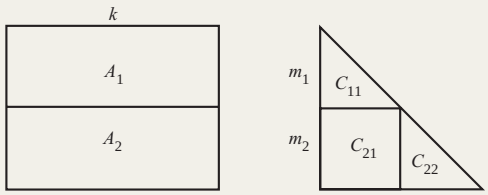

Figure 9

Block partitioning of the SYRK computation.

```
C(1:m, 1:m) = SYRK[A(1:m, 1:k), C(1:m, 1:m)]
if(m = 1) then
  C(1, 1) = C(1, 1) − A · A^T ! DDOT operation
else ! m > 1
  m1 = m/2 and j1 = m1 + 1
  C(1:m1, 1:m1) = SYRK[A(1:m1, 1:k), C(1:m1, 1:m1)]
  Update C(j1:m, 1:m1) = C(j1:m, 1:m) − A(j1:m, 1:k) · A(1:m1, 1:k)^T
  C(j1:m, j1:m) = SYRK[A(j1:m, 1:k), C(j1:m, j1:m)]
endif
```

Figure 10

Algorithm for recursive symmetric rank k update, $C \leftarrow C - A \cdot A^T$. C is lower triangular, stored in packed recursive row format. A is stored in row major format.

true for $0 < k \leq 2^i$. Let $2^i < j \leq 2^{i+1}$, $j_1 = \lfloor j/2 \rfloor$, and $j_2 = j - j_1$. Since $j > 1$, the computations described in (13), (14), and (15) are performed in that order.

Equation (13) is satisfied by the induction hypothesis. Equation (14) is a calculation (matrix multiply), and finally, Equation (15) is also satisfied by the induction hypothesis. Now we want to show that Equation (12) is true by using Equations (13)–(15). This is trivially true as we follow the rules of 2×2 block multiplication, where the partitioning of A is $k = k_1 + k_2$. The conditions that the block elements of X_1 and X_2 must satisfy are exactly those of Equations (13)–(15), which we have shown to be true. In **Figure 8** we give the details of recursive algorithms TRSM(k).

Now we express DSYRK recursively. Since the procedure is very similar to that presented for DTRSM, we just state the algorithm and prove its correctness. First we define some notation. DSYRK requires S [S_1 in Figure 6(a)] and T_2 as operands. Divide T_2 into T_{21}, S_2, and T_{22} according to its recursive definition. Now, let $C = T_2$ and $A = S$; see **Figure 9**. The relation between Figure 6(a) and Figure 9 is again $n_2 = m$ and $n_1 = k$. The algorithm is shown in **Figure 10**.

The correctness of SYRK follows by mathematical induction; its proof is omitted, since it is similar to the DTRSM proof.

• *Cholesky algorithm applied to recursive lower row storage produces stride 1 storage access throughout*

Having established the content of the preceding subsection, we may now turn to demonstrating why transposing recursive lower column storage leads to stride 1 performance throughout. This is true for Cholesky factorization and many other symmetric algorithms. Assume that we use full recursion. Then the factor part of the code becomes n square root calculations. The rest of the code consists of calls to RTRSM and RSYRK. However, both RTRSM and RSYRK are recursive, and when used with full recursion they always consist of calls to DGEMM and Level 1 calls to DSCAL and DDOT. Now take a DGEMM call from RTRSM. Note that $C = C - A \cdot B^T$ is computed, where C is $m \times n$, A is $m \times k$, and B is $n \times k$. Similarly, a DGEMM call from RSYRK has the same form, $C = C - A \cdot B^T$. Now transpose this generic DGEMM computation to obtain $C^T = C^T - B \cdot A^T$ and assume that L is stored in recursive lower row-wise storage. Since storing a full matrix row-wise is identical to storing its transpose column-wise, we see that $C^T = C^T - B \cdot A^T$ becomes $D = D - E^T \cdot F$, where $D = C^T$, $E = B^T$, and $A = F^T$. Note that each computation problem for C and D consists of doing mn dot products each of size k. The form $C = C - A \cdot B^T$ computes dot products stride lda, ldb, while the form $D = D - E^T \cdot F$ computes dot products stride 1.

Before continuing, we use the above results to state a possible result about symmetric full storage. Assume that the answer to the question as to which data format gives faster execution for the symmetric algorithm is that the U format algorithm is faster by a sufficient amount. Then two things occur: 1) better performance using the single

code, and 2) instead of having to write two codes, the *uplo* = 'L' code disappears. In its place one need only invoke an existing in-place square transpose code twice, at the beginning and at the end. Moving twice the data twice is necessary for migration purposes.

• *Data transformation*

In order for the recursive algorithm to use the recursive packed data format, the user's packed data must be transformed to recursive packed format. In the Cholesky case, the user's symmetric matrix is in lower packed format, which is overwritten with the matrix in recursive packed upper format, as described in the two preceding subsections. The recursive packed upper format is chosen so that the `DGEMM` operation will be A^TB; i.e., the A and B matrices are accessed with stride 1.

The in-place data transformation of a lower packed triangle of size n to recursive packed upper format is done in eight steps; see **Figure 11** and **Table 1**. These steps are listed below. Now we overview the algorithm. First we do an in-place data transformation of the first $\lfloor n/2 \rfloor$ columns of the packed array. This figure is a trapezoid and a rectangle of size $\lceil n/2 \rceil$ by $\lfloor n/2 \rfloor$. The lower triangle is moved out of place to reside in the auxiliary buffer in recursive upper packed format. Next the rectangle is moved in place to full (column major order) format to reside in its final position of the trapezoid. Finally, the contents of the auxiliary buffer are copied to the triangle part of the trapezoid. These three steps are described in steps 1 to 6 below. The second major step is described in steps 7 and 8 below.

1. The first left triangle of size $\lfloor n/2 \rfloor$ of the packed lower triangle is stored in packed recursive row lower format [see Figure 6(a)] in the auxiliary buffer at the first position (0) to position
$$\frac{\lfloor n/2 \rfloor \cdot (\lfloor n/2 \rfloor + 1)}{2} - 1.$$
This is done using an out-of-place data transformation subroutine. Notice that the upper left triangle is not stored contiguously in the packed format, so this copy to buffer is a gather operation. That is, all of the vectors in the original matrix, stored in packed format, are now stored recursively in the contiguous memory of the buffer. See Table 1, Step 1, where the vectors containing elements 11–31, 22–32, and 33 of the packed triangular array of size 7 are stored as one vector of length 6 in the auxiliary buffer.
2. It is easy to do an in-place transpose of a square, but almost impossible to do an in-place transpose of a rectangle. A transpose of a nonsquare rectangle is very hard to do in place, so if n is odd, the first row of the rectangle, which is of size $\lceil n/2 \rceil \times \lfloor n/2 \rfloor$, is also stored contiguously in the transfer buffer at position
$$\frac{\lfloor n/2 \rfloor \cdot (\lfloor n/2 \rfloor + 1)}{2}$$
to
$$\frac{\lfloor n/2 \rfloor \cdot (\lfloor n/2 \rfloor + 3)}{2} - 1.$$
In Table 1, Step 2, the elements 41, 42, and 43 (that is, the row directly above the lower left 3×3 square in the 7×7 triangle) are stored contiguously as a vector of length 3 in the auxiliary buffer, following directly after the triangle that was created in Step 1.
3. The gap between the columns in the remaining lower left square is removed by moving column i, $0 \le i < \lfloor n/2 \rfloor$, from positions
$$i\left(\frac{2n - i - 1}{2}\right) + \lceil n/2 \rceil$$
to
$$i\left(\frac{2n - i - 1}{2}\right) + n - 1$$
to positions
$$\lfloor n/2 \rfloor \left(\frac{2i + 2\lceil n/2 \rceil - \lfloor n/2 \rfloor + 1}{2}\right)$$

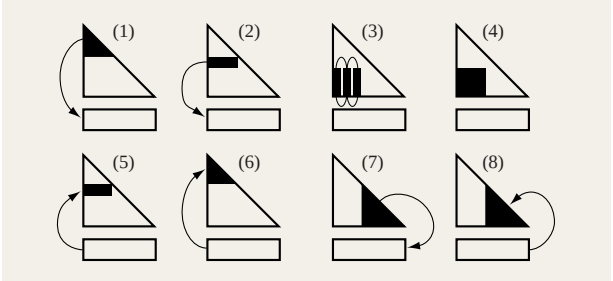

Figure 11

The eight steps of data transformation: (1) The upper left triangle of the packed lower triangle is stored in packed recursive row lower format in the auxiliary buffer; (2) if n is odd, the first row of the rectangle is also stored contiguously in the auxiliary buffer; (3) the gap between the columns in the remaining lower left square is removed by moving the columns; this is done in place, moving the columns one by one, right to left; (4) the lower left square is now contiguous in the packed array; this square matrix is now transposed in place; (5) if n is odd, the row stored in step 2 is now copied back to the packed array; (6) the triangle stored in step 1 is copied back to the packed array; (7) the lower right triangle is stored in packed recursive row lower format in the auxiliary buffer; (8) the triangle stored from step 7 is copied back to the packed array.

Table 1 Data transformation for $n = 7$. Compare with Figures 5 and 11.

Operation		*Storage*						*Auxiliary buffer*				
		1	*2*	*3*	*4*	*5*		*1*	*2*	*3*	*4*	*5*
Initial values (packed lower)	0 5 10 15 20 25	11 61 52 53 64 66	21 71 62 63 74 76	31 22 72 73 55 77	41 32 33 44 65	51 42 43 54 75	0 5	— —	— —	— —	— —	— —
(1) Copy/ transform triangle to buffer (gather)	0 5 10 15 20 25	(11)	(21)	(31) (22)	(32) (33)		0 5	[11] [33]	[21]	[31]	[22]	[32]
(2) Copy rectangle row (gather)	0 5 10 15 20 25				(41)	(42) (43)	0 5		[41]	[42]	[43]	
(3) Move square columns	0 5 10 15 20 25	$4^{(61)}$ $[3^{(52)61}]_4$ {53}	$4^{(71)}$ $[2^{(62)71}]_4$ {63}	$[1^{(72)52}]_3$ {73}	$[62]_2$	$4^{(51)}$ $[51]_4$ $[72]_1$	0 5					
(4) Transpose square in-place	0 5 10 15 20 25	52 71	53 72	61 73	62	51 63	0 5					
(5) Copy saved row back	0 5 10 15 20 25		[41]	[42]	[43]		0 5		(41)	(42)	(43)	
(6) Copy triangle back	0 5 10 15 20 25	[11] [33]	[21]	[31]	[22]	[32]	0 5	(11) (33)	(21)	(31)	(22)	(32)
(7) Copy/ transform lower right triangle (gather)	0 5 10 15 20 25	(64) (66)	(74) (76)	(55) (77)	(44) (65)	(54) (75)	0 5	[44] [74]	[54] [75]	[55] [66]	[64] [76]	[65] [77]

(continued)

Table 1 *Continued*

Operation		*Storage*						*Auxiliary buffer*				
		1	*2*	*3*	*4*	*5*		*1*	*2*	*3*	*4*	*5*
(8)	0						0	(44)	(54)	(55)	(64)	(65)
Copy	5						5	(74)	(75)	(66)	(76)	(77)
triangle	10											
back	15				[44]	[54]						
triangle	20	[55]	[64]	[65]	[74]	[75]						
(gather)	25	[66]	[76]	[77]								
Final	0	11	21	31	22	32	0					
result	5	33	41	42	43	51	5					
(recursive	10	52	53	61	62	63						
lower	15	71	72	73	44	54						
row	20	55	64	65	74	75						
format)	25	66	76	77								

Legend:
$_t(ij)$ Element $a(i, j)$ at location is accessed at time t (read operation). [Order of accesses is only important in operation (3).]
$[ij]_u$ Element $a(i, j)$ at location is modified at time u (write operation). [Order of accesses is only important in operation (3).]
{ } Data is part of the operation but neither read nor written [rightmost column in operation (3)].

to

$$\lfloor n/2 \rfloor \left(\frac{2i + 2\lceil n/2 \rceil - \lfloor n/2 \rfloor + 3}{2} \right) - 1.$$

This is done in place, starting with the rightmost column, by $\lfloor n/2 \rfloor - 1$ calls to `DCOPY`. Notice that when $i = \lfloor n/2 \rfloor - 1$, the rightmost column is already in its final place.

4. The square lower part of the rectangle is transposed, since we want the resulting matrix to be in recursive packed column upper format (which is equivalent to recursive packed row lower format). The lower left square of size $\lfloor n/2 \rfloor \times \lfloor n/2 \rfloor$ is contiguous in memory at positions

$$\lfloor n/2 \rfloor \left(\frac{2\lceil n/2 \rceil - \lfloor n/2 \rfloor + 1}{2} \right)$$

to

$$\lfloor n/2 \rfloor \left(\frac{2n - \lfloor n/2 \rfloor + 1}{2} \right) - 1.$$

This square is transposed in place by a call to the ESSL routine `DGETMI`.

5. If n is odd, the row stored in step 2 is now copied to positions

$$\frac{\lfloor n/2 \rfloor (\lfloor n/2 \rfloor + 1)}{2}$$

to

$$\frac{\lfloor n/2 \rfloor (\lfloor n/2 \rfloor + 3)}{2} - 1.$$

Compare with Table 1, Step 5. Since this is a contiguous vector, `DCOPY` is used.

6. The triangle stored in Step 1 is copied back. At this time, the trapezoid consisting of the entire left part of the triangle is in packed recursive row lower transposed format, with the upper left triangle in the recursive packed lower format and the rectangle in row-major order.
7. Now, the lower right triangle of size $\lceil n/2 \rceil$ is stored in packed recursive row lower format in the auxiliary buffer.
8. The triangle stored from Step 7 is copied back to the packed triangle.

In Table 1, explicit details for the illustrative example where $n = 7$ is given. This example contains the salient feature of the in-place data transformation algorithm for a general n. In Table 1, the notation (ij) in position p means that $a(i, j)$ is read from memory location p. The notation $[ij]$ in position p means that $a(i, j)$ is written into memory location p. The notation $[_t(ij)kl]_u$ in position p means that the element $a(i, j)$ is read from memory location p at time t and that the same memory location p is overwritten with element $a(k, l)$ at time u. Steps 1 and 2 of Table 1 are self-explanatory. Step (3) is more complex. The general idea is to move data in place so that data written at time t does not overwrite any data that will be read at time $u > t$. In step 3, column 2 [i.e., elements $a(5{:}7, 2)$ of AP] is moved from locations 11, 12, 13 to locations 13, 14, 15. Since these storage locations overlap, we move this data in a backward manner; i.e., at time step 1, $a(7, 2)$ is read from location 13 and written into location 15; at time

```
A(1:n, 1:n) = BC[A(1:n, 1:n)]
if (n < 2nb) then
  A(1:n, 1:n) = BCK[A(1:n, 1:n)] ! Block Cholesky factor A with Level 3 kernel routine
else ! n ≥ 2nb
  n1 = n/2 and j1 = n1 + 1
  A(1:n1, 1:n1) = BC[A(1:n1, 1:n1)]
  A(j1:n, 1:n1) = BT[A(1:n1, 1:n1), A(j1:n, 1:n1)]
  A(j1:n, j1:n) = BS[A(j1:n, 1:n1), A(j1:n, j1:n)]
  A(j1:n, j1:n) = BC[A(j1:n, j1:n)]
endif
```

Figure 12

Algorithm for block recursive Cholesky factorization. Matrix A is stored in packed recursive lower format PRLF.

step 2, $a(6, 2)$ is read from location 12 and written into location 14; at time step 3, $a(5, 2)$ is read from location 11 and written into location 13. Next, $a(5:7, 1)$ of AP is moved from locations 5:7 to locations 10:12. Since these storage locations do not overlap, this move can be done by calling DCOPY at composite time step 4. Note that column 3, $a(5:7, 3)$, is neither read nor written, as this column is already in place. After step 3, A is stored in full format as a square matrix. Step 4 transposes this matrix in place by a standard in-place square transpose algorithm. Thus, in step 4, we just exhibit the elements of the matrix after transposition has occurred. Steps 5, 6, 7, and 8 of Table 1 are self-explanatory.

3. Algorithmic components

- *The problem with a large recursion tree*

Blocking reduces a large recursion tree to a recursion tree of small size. Recall that a Cholesky program of size N has a binary tree with N leaves and $N - 1$ interior nodes. The overhead at any node increases with n, where n is the size of the Cholesky subproblem, since each interior node performs the same Cholesky computation. To see this, note that each call is recursive, so a call at size $2n$ includes two calls at size n plus calls to recursive routines RTRSM and RSYRK. What is important for high performance is the ratio r of the cost of the recursive call overhead to the computation cost (number of FLOPs). It turns out that when n is large, this ratio is tiny and can be safely neglected. We quantify this ratio in Section 3. Here we discuss the pruning of the recursion tree so that every node of the reduced tree has negligible r.

At a given tree level i, all nodes (there are 2^i nodes, each of size $m = n_i$ or $m = n_i + 1$, where $n_i = N/2^i$) again execute the Cholesky algorithm on a submatrix of size m. The FLOP count of Cholesky is $\frac{1}{3}m^3$. Thus, the overhead to FLOP-count ratio at tree level i is about eight times smaller than at level $i + 1$, and the number of nodes doubles. At some level, say k, this ratio times 2^k becomes too high. This requires that we prune the recursion tree at level k and below. Suppose nb is some blocking parameter where $2^k \leq nb < 2^{k+1}$. If the size of the Cholesky subproblem is $n \geq 2^{k+1}$, another recursion step is attempted. Otherwise, $n < 2^{k+1}$, and the given Cholesky problem of size n is factored using a direct method. Using this strategy guarantees that the smallest direct-method problem size will be $n \geq 2^k$ if the original problem size is $N \geq 2^k$. If $N < 2^k$, we used a simple packed format code that used register blocking because the entire packed matrix fit easily into cache. In our implementation k was chosen to be 4. An interior node has $n \geq 2^{k+1}$. Here we execute a recursive TRSM problem of size n_1 and a recursive SYRK problem of size n_2, where $n_1 + n_2 = n$, in addition to recursive Cholesky calls of sizes n_1 and n_2. The FLOP counts of TRSM and SYRK at a node of size m, n are m^2n and $(m^2 + m)n$, respectively. If $n_1 < 2^{k+1}$, we solve the TRSM BLAS problem by a direct method and solve the SYRK BLAS problem similarly. Otherwise, we can safely do a recursion step, since the ratio r for the problem will be tiny.

- *Block version of recursive packed format Cholesky*

In [1], Andersen, Gustavson, and Waśniewski show an implementation of recursive Cholesky. They let the recursion go down to a single element. Because of the overhead of the recursive calls, the implementation has a significant performance loss for small problems. However, for large problems most of the computation takes place in the DGEMM, so that performance loss, while significant for smaller problems, is minor. The difference between that implementation and the one described here is that we combine recursion with blocking.

In our implementation, however, we combine blocking and recursion to produce a blocked version of their recursive algorithm. We do recursion only down to a fixed block size $2nb$. To solve problems for sizes smaller than $2nb$, we use a variety of algorithmic techniques, some new and some old, to produce optimized unrolled kernel routines. These techniques are used for both the Cholesky factorization routine and the recursive TRSM and recursive SYRK routines. The main technique we use to produce these fast kernels is register and Level 1 cache blocking. The specific kernels are described in the subsection on three types of kernel routines.

Next we give details about our block version and demonstrate analytically that blocking the recursion tree via pruning completely eliminates performance problems for small matrix sizes.

Algorithms for block Cholesky, block TRSM, and block SYRK

In this short section we modify the algorithms CHOL (Figure 2), TRSM (Figure 9), and SYRK (Figure 10) to

$B(1{:}m, 1{:}n) = \texttt{BT}[A(1{:}n, 1{:}n), B(1{:}m, 1{:}n)]$
if $(n < 2nb)$ **then**
  $B(1{:}m, 1{:}n) = \texttt{BTK}[A(1{:}n, 1{:}n), B(1{:}m, 1{:}n)]$! Solve $XA^T = B$ using Level 3 RUNN kernel
else ! $n \geq 2nb$
  $n_1 = n/2$ and $j_1 = n_1 + 1$
  $B(1{:}m, 1{:}n_1) = \texttt{BT}[A(1{:}n_1, 1{:}n_1), B(1{:}m, 1{:}n_1)]$
  Update $B(1{:}m, j_1{:}n) = B(1{:}m, j_1{:}n) - B(1{:}m, 1{:}n_1) \cdot A(j_1{:}n, 1{:}n_1)^T$
  $B(1{:}m, j_1{:}n) = \texttt{BT}[A(j_1{:}n, j_1{:}n), B(1{:}m, j_1{:}n)]$
endif

Figure 13

Algorithm for block recursive triangular solution. A is in PRLF and B is in row major order with leading dimension n.

$C(1{:}m, 1{:}m) = \texttt{BS}[A(1{:}m, 1{:}n), C(1{:}m, 1{:}m)]$
if $(m < 2nb)$ **then**
  $C(1{:}m, 1{:}m) = \texttt{BSK}[A(1{:}m, 1{:}n), C(1{:}m, 1{:}m)]$! Perform rank n update using Level 3 LT kernel
else ! $m \geq 2nb$
  $m_1 = m/2$ and $j_1 = m_1 + 1$
  $C(1{:}m_1, 1{:}m_1) = \texttt{BS}[A(1{:}m_1, 1{:}n), C(1{:}m_1, 1{:}m_1)]$
  Update $C(j_1{:}m, 1{:}m_1) = C(j_1{:}m, 1{:}m_1) - A(j_1{:}m, 1{:}n) \cdot A(1{:}m_1, 1{:}n)^T$
  $C(j_1{:}m, j_1{:}m) = \texttt{BS}[A(j_1{:}m, 1{:}n), C(j_1{:}m, j_1{:}m)]$
endif

Figure 14

Algorithm for block recursive triangular rank n update. A is in row major order with leading dimension n and C is in PRLF.

produce their block counterparts, algorithms BC, BT, and BS in **Figures 12**, **13**, and **14**, respectively. In each case, we have a blocking factor nb, and the only change to these three algorithms is to make their if-then-else clauses dependent on nb. Also, the code of each if clause becomes a call to a kernel routine instead of a square root operation or a call to a Level 1 BLAS.

We omit all proofs of correctness because they are similar to the proofs given in Section 2. Note, however, that when $nb = 1$ we get algorithms CHOL, TRSM, and SYRK, because then the three Level 3 kernel routines BCK, BTK, and BSK reduce respectively to SQRT, DSCAL, and DDOT computations.

Now we discuss how the packed recursive lower format, PRLF, and algorithms BC, BT, and BS relate to one another. Take BC first. In the recursive part of the code

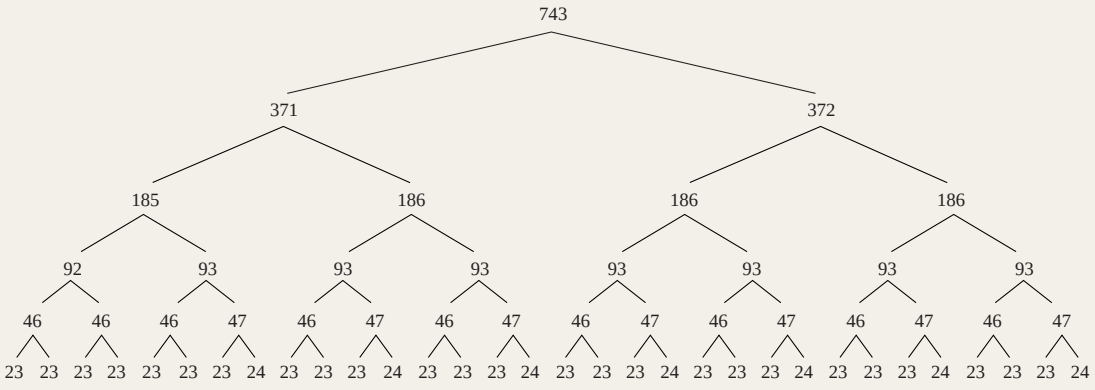

Figure 15

Recursion tree for $N = 743$.

there are four calls, two to BC and one each to BT and BS. The definition of PRLF guarantees that the two BC calls will receive lower triangular submatrices than are in PRLF. To see this, simply note that the if-then-else control logic of BC, where n is replaced by $n_1 = \lfloor n/2 \rfloor$ and $n_2 = n - n_1$, is precisely the same as the control logic used to define PRLF (see the subsection on data transformation). Now look at the calls to BT and BS. Each of these recursive routines has two operands: a triangle in PRLF and a conventional rectangular matrix stored in row major order with a leading dimension of n_1. This fact follows immediately from the definition of PRLF and the fact that BC control logic is identical to the control logic defining PRLF. Turning now to recursive algorithms BT and BS, we see the same if-then-else control logic that was used in BC. Hence, the two recursive calls in both BT and BS define triangle and rectangle operands that are in PRLF and row major format, respectively. Note, however, that the leading dimension of the rectangle remains constant for all recursive calls of BT and BS.

Finally, note that because the algorithm and the data structure are so closely tied together, no data movement of the operands occurs during any of the recursive calls. Also, the floating-point operations occur only in calls to BT, BS, at various internal nodes of the Cholesky tree, and in calls to the kernel routine BCK at the leaves of the Cholesky tree. Now consider any internal node of the Cholesky tree where there is one call each to BT and BS. In both the BT and BS calls there is an associated recursion tree. For BT, the tree is the left child of the node; for BS, the tree is the right child of the node. Now consider either of these trees. At each interior node there is one call to DGEMM, and at the leaves there are calls to kernel routines BTK and BSK. We discuss the implementation of the three kernel routines later in Section 3. The discussion of the DGEMM implementation is beyond the scope of this paper; see [7] for some details. We merely note that DGEMM is a Level 3 BLAS, and by its very nature should be high-performance and fully tuned for a given architecture/platform. The data movements of the operands occur in the calls to DGEMM, BCK, BTK, and BSK. In the Conclusion we give a further discussion of data operand movement in the calls to DGEMM.

Recursion tree has 2^i nodes at every level

Theorem 2 Given $N \geq nb$, let $2^q nb \leq N < 2^{q+1} nb$ determine q. Let $n_i = \lfloor N/2^i \rfloor$. Now either 1) $2^q nb \leq N \leq 2^{q+1} nb - 2^q$, or 2) $2^{q+1} nb - 2^q < N < 2^{q+1} nb$. When Case 1 holds, the binary recursion tree for the blocked Cholesky algorithm BC(N) has 2^q leaves and $2^q - 1$ interior nodes; i.e., it has $q + 1$ levels, where Level k has 2^k nodes, $0 \leq k \leq q$. When Case 2 holds, $N = 2^{q+1} nb - 2^q + i$, where $1 \leq i < 2^q$. The binary recursion tree for BC(N) has $2^q + i$ leaves and $2^q + i - 1$ interior nodes. It has $q + 2$ levels, where Level k has 2^k nodes, $0 \leq k \leq q$, and Level $q + 1$ has $2 \cdot i$ leaves. Consider the binary recursion tree associated with BC(N). In both cases, for $0 \leq k \leq q$ we have the following: At a given level k, α_k of these nodes denotes a block Cholesky algorithm BC(n_k), and the remaining $\beta_k = 2^k - \alpha_k$ denotes a block Cholesky algorithm BC($n_k + 1$). Also, $\alpha_k n_k + \beta_k (n_k + 1) = N$ holds. Additionally, in Case 2 we have $n_q = 2nb - 1$, $\alpha_q = 2^q - i$, and $\beta_q = i$, and at Level $q + 1$ there are $2i$ leaves with $n_{q+1} = nb$.

Proof We use mathematical induction. For $i = 0$, $n_0 = N$, $\alpha_0 = 1$, $\beta_0 = 0$ as we have the original problem BC(n_0); see Figure 12. Assume that the result is true for $i = j$. The induction hypothesis is $\alpha_j n_j + \beta_j(n_j + 1) = N$ and $\alpha_j + \beta_j = 2^j$. Now $n_{j+1} = \lfloor n_j/2 \rfloor$. There are two cases, depending on whether n_j is even or odd. When n_j is even, $\alpha_{j+1} = 2\alpha_j + \beta_j$ and $\beta_{j+1} = \beta_j$. When n_j is odd, $\alpha_{j+1} = \alpha_j$ and $\beta_{j+1} = \alpha_j + 2\beta_j$. The equations for α_{j+1} and β_{j+1} follow directly from the BC algorithm in Figure 12. In both cases, a direct calculation gives $\alpha_{j+1} n_{j+1} + \beta_{j+1}(n_{j+1} + 1) = \alpha_j n_j + \beta_j(n_j + 1)$, which equals N via the induction hypothesis. Also, $\alpha_{j+1} + \beta_{j+1} = 2(\alpha_j + \beta_j)$, which equals $2 \cdot 2^j$ via the induction hypothesis. Similar proofs hold for Algorithms BT and BS of Figures 13 and 14, respectively. □

Some notes: In going from Level i to Level $i + 1$, the number of Cholesky factorizations doubles, but their size is halved. This means that the total number of FLOPs decreases by a factor of approximately 4 in going down one tree level. Precise details about this are given on page 740 of [8].

In **Figure 15** we give an example with $N = 743$ and $nb = 16$. The recursion tree has 63 nodes. At level 0 there is one computation consisting of the single (BT, BS) pair of size (371, 372). The two trees are the two children of node 743, namely nodes 371 and 372. Hence, for BT there are 15 calls to DGEMM and 16 calls to kernel BTK. For BS there are also 15 calls to DGEMM and 16 calls to kernel BSK. For both BT and BS, the sizes (m, n, k) of the DGEMM calls are given by the node labels, which for kernels BT and BS are the same. At Level 1 there are two (BT, BS) pairs of sizes (185, 186) and (186, 186), respectively. The four subtrees are the four trees for BT and BS, two each. For each tree there are seven DGEMM and eight kernel calls. At Level 2 there are one (92, 93) pair and three (93, 93) pairs. The eight subtrees are the eight trees for BT and BS, four each. For each tree there are three DGEMM and four kernel calls. At Level 3 there are one (46, 46) pair and seven (46, 47) pairs. The sixteen subtrees are the sixteen trees for BT and BS, eight each. For each tree there are a single DGEMM and two kernel calls. At Level 4 there are nine (23, 23) pairs and seven (23, 24) pairs. There are 32 leaves, and these leaves correspond to sixteen BT and BS calls. Each of the 32 calls is a kernel routine call. At Level 5 there are 32 BCK calls, 25 of size 23 and seven of size 24. This is summarized in **Table 2**.

The example for $N = 743$ is the typical case. Let $512 \le N < 1024$. Dividing by $2nb$, we have $16 \le (N/2nb) < 32$. This partitions N into nb intervals [512, 544), [544, 576), $\cdots$, [992, 1024) labeled by the numbers nb to $2nb - 1$. Note that 743 belongs to the partition [736, 768) labeled by the number 23. Partitions 16 to 30 have the same binary tree as in Figure 15, namely one of 63 nodes. Partition 31, Case 2 of Theorem 2,

Table 2 Values for n_i, α_i, and β_i for different levels at the recursion tree for $N = 743$. Notice that $\alpha_i n_i + \beta_i(n_i + 1) = 743$ holds for all values of i, $0 \le i \le 5$.

Level i	n_i	α_i	β_i	$n_i + 1$
0	743	1	0	744
1	371	1	1	372
2	185	1	3	186
3	92	1	7	93
4	46	9	7	47
5	23	25	7	24

constitutes a transition between a binary tree of 31 interior nodes and 32 leaves and one of 63 interior nodes and 64 leaves, the next power of 2. There are $2nb$ numbers $992 + i$, $0 \le i < 2nb$ in partition 31. For $N = 992 + i$, the binary tree has $31 + i$ interior nodes and $32 + i$ leaves. The leaf nodes consist of $2i$ nodes of size nb and $32 - i$ nodes of size $2nb - 1$. Later in Section 3, we refer to this partition as the special case. Finally, note that the interval $[2^9, 2^{10})$ is generic, as Theorem 2 shows.

Overhead of recursion is negligible

To calculate the cost of the recursive calls, the recursive algorithm is written as a recursive cost function. Let t_{call} be the cost of a function call overhead, including cycles spent to initialize variables (e.g., $n_1 = \lfloor n/2 \rfloor$). Let t_{flop} be the cost of a floating-point operation.

Start with Algorithm BC(N) of Figure 12. When $n \ge 2nb$, the else clause of the if statement is executed, and two calls to BC, one call to BT, and one call to BS are performed. All of these calls operate on matrix sizes equal to about half of the original. After investigating the algorithms for BT in Figure 13 and BS in Figure 14, we can, when $n \ge 2nb$, write these costs as

$$C(n) = 1 \cdot t_{\text{call}C} + C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor, \lceil n/2 \rceil) + S(\lceil n/2 \rceil, \lfloor n/2 \rfloor); \quad (16)$$

$$T(n, m) = 1 \cdot t_{\text{call}T} + T(\lfloor n/2 \rfloor, m) + T(\lceil n/2 \rceil, m) + G(\lceil n/2 \rceil, m, \lfloor n/2 \rfloor); \quad (17)$$

$$S(n, m) = 1 \cdot t_{\text{call}S} + S(\lfloor n/2 \rfloor, m) + S(\lceil n/2 \rceil, m) + G(\lfloor n/2 \rfloor, \lceil n/2 \rceil, m), \quad (18)$$

where $C(n)$ is the cost of the Cholesky factorization, $T(n, m)$ is the cost of a triangular solution with m right sides, $S(n, m)$ is the cost of a symmetric rank-k update with $k = m$, and $G(m, n, k)$ is the cost of a matrix–matrix multiply. Notice that each function accounts for the call to itself. For $n < 2nb$, no recursive calls are performed, so the cost is approximated by

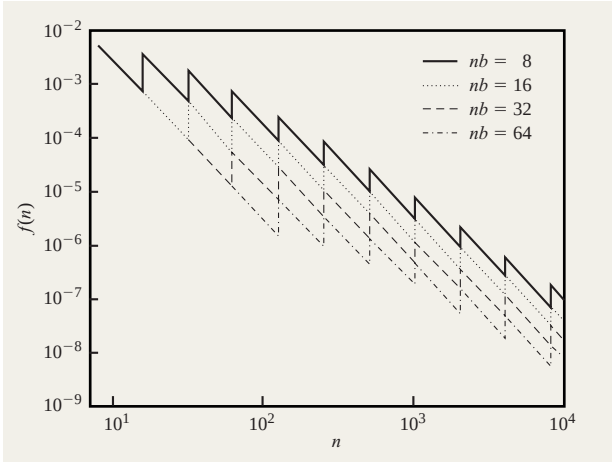

Figure 16

$f(n) = 6\frac{3k2^k - 2^{k+1} + 3}{n(n+1)(2n+1)}$, $k = \lceil \log_2 \lceil \frac{n}{2nb} \rceil \rceil$.

$$C(n) = 1 \cdot t_{\text{call}C} + \frac{2n^3 + 3n^2 + n}{6} \cdot t_{\text{flop}}; \tag{19}$$

$$T(n, m) = 1 \cdot t_{\text{call}T} + n^2 m \cdot t_{\text{flop}}; \tag{20}$$

$$S(n, m) = 1 \cdot t_{\text{call}S} + (n^2 + n)m \cdot t_{\text{flop}}. \tag{21}$$

Now, only the cost formulation of the matrix–matrix multiply is needed to solve the cost equations:

$$G(m, n, k) = 1 \cdot t_{\text{call}G} + 2mnk \cdot t_{\text{flop}}. \tag{22}$$

If Case 1 of Theorem 2 holds (i.e., $2^q nb \le n \le 2^q nb - 2^q$), the solution to Equation (16) can be written as

$$\begin{aligned} C(n) = {} & \frac{2n^3 + 3n^2 + n}{6} \cdot t_{\text{flop}} + (2 \cdot 2^q - 1) \cdot t_{\text{call}C} \\ & + [2^q \cdot (q-1) + 1] \cdot t_{\text{call}T} + [2^q \cdot (q-1) + 1] \cdot t_{\text{call}S} \\ & + [2^q \cdot (q-2) + 2] \cdot t_{\text{call}G}. \end{aligned} \tag{23}$$

That is, for Case 1, in order to Cholesky-factorize a symmetric matrix of size n, there will be $2 \cdot 2^q - 1$ calls to the Cholesky routine, $2^q \cdot (q-1) + 1$ calls to each of the TRSM and SYRK routines, and $2^q \cdot (q-2) + 2$ calls to the GEMM routine.

Bounds for (23) for all values of n are found in the Appendix. For now, we discuss the properties of Equation (23) for Case 1 values of n with $nb = 16$.

The important relation here is the number of t_{call}s compared to the number of t_{flop}s. Notice that the number of calls ($t_{\text{call}C} + t_{\text{call}T} + t_{\text{call}S} + t_{\text{call}G}$) for Cholesky is $3q2^q - 2^{q+1} + 3$, while the number of floating-point operations is well known, $\frac{1}{6}n(n+1)(2n+1)$. This clearly shows that the overhead due to recursion is very small, as the quotient between the leading terms of these two expressions is

$$\frac{\textit{no. of calls}}{\textit{no. of FLOPs}} = 9\,\frac{\log_2 \frac{n}{nb}}{n^2 nb}.$$

The relative overhead is illustrated in **Figure 16**, where the ratio between the number of calls to subroutines and arithmetic work is plotted. Notice that the recursive code handles problems where $n \ge nb$, since we provide special kernels for small problems where $1 \le n < nb$. The plot is exact only for Case 1 values of n, since the formula underestimates the number of function calls for Case 2 values of n. As an example, let $n = 128$, $nb = 16$ (kernels operate on problem sizes 16 to 31), and assume that the overhead of a function call, including the small setup part of the recursive functions, takes 200 times the amount of time it takes to perform a floating-point operation. Then the overhead due to the recursion is about 1.7%! This overhead is also dropping fast; for $n = 256$, it is about 0.6%. For our example with $n = 743$, the number of FLOPs is $n \cdot (n+1) \cdot (2n+1)/6 = 137000284$, while the number of calls is 419, with $nb = 16$; i.e.,

$$\frac{\textit{no. of calls}}{\textit{no. of FLOPs}} \cdot 200 = \frac{419}{137000284} \cdot 200 \approx 6.12 \cdot 10^{-4} \approx 0.06\%.$$

Block size is large enough to give a Level-3-like performance
In our implementation, $nb = 16$. This means that the smallest possible block on which the kernels operate is of size 16×16. The kernels use 4×4 loop unrolling and Level 3 algorithmic preloading [9]. This technique makes efficient use of the many floating-point registers available. It separates the load/store operations from the dependent floating-point operations. This nullifies the number of stalls in the superscalar architecture; i.e., the floating-point units are not delayed because their operand has not been completed. This means that dependent floating-point operations are scheduled at sufficient distance from one another. (Example: The operation $c_{11} = c_{11} + a_{11} \cdot b_{11}$ must be sufficiently separated from the operation $c_{11} = c_{11} + a_{12} \cdot b_{21}$, or the second operation will be delayed until the c_{11} value of the first operation is calculated.)

Not many extra cycles are consumed in the kernels anyway. Equations (19)–(22) can be refined further, by differentiating between the t_{flop}s in different kernels. After substituting $t_{\text{flop}C}$, $t_{\text{flop}T}$, $t_{\text{flop}S}$, and $t_{\text{flop}G}$ for t_{flop} in (19)–(22), the new cost expression for Cholesky factorization becomes

$$C(n) = \mathcal{O}(n \log n) \cdot t_{\text{call}} + \tfrac{1}{6} n(2nb + 1)(nb + 1) \cdot t_{\text{flop}C} + \tfrac{1}{2} n(n - nb)nb \cdot t_{\text{flop}T} + \tfrac{1}{2} n(n - nb)(nb + 1) \cdot t_{\text{flop}S} + \tfrac{1}{3} n(n - nb)(n - 2nb) \cdot t_{\text{flop}G}. \quad (24)$$

This means that the n^3 part of the calculation cost is spent entirely in the `DGEMM` routine! This can be seen by looking at how the number of FLOPs spent in routines other than `DGEMM` is related to the problem size, since the leading term of the quotient between the coefficients of the $t_{\text{flop}C}$, $T_{\text{flop}T}$, and $t_{\text{flop}S}$ terms and the total number of FLOPs in $C(n)$ in (24) is

$$\frac{\textit{no. of FLOPs not in } \texttt{DGEMM}}{\textit{no. of FLOPs}} = \frac{3nb}{n}.$$

Only two node sizes, n_i and $n_i + 1$

Assume that the topmost node is of size n_0. We now show that the sizes of all of the nodes on the same, arbitrary, level i in the tree will be one of two consecutive values; call these sizes n_i and $n_i + 1$. This result is also part of Theorem 2. We use induction on the level of the tree $i = 0, 1, \cdots$. Now, for $i = 0$, there is only one node, and the result is trivially true.

Suppose that the result is true when $i = k$; that is, for Level k in the tree, every node is either of size n_k or $n_k + 1$. Now, suppose that n_k is even; then $n_k + 1$ is odd. Then the children of the node of size n_k are $\lfloor n_k/2 \rfloor = \lceil n_k/2 \rceil = n_k/2$, and the children of the node of size $n_k + 1$ are $\lfloor (n_k + 1)/2 \rfloor = n_k/2$ and $\lceil (n_k + 1)/2 \rceil = n_k/2 + 1$. Thus, the children nodes are of two sizes, $n_k/2$ and $n_k/2 + 1$.

For the other case, if n_k is odd, then $n_k + 1$ is even. Then the children of the node of size n_k are $\lfloor n_k/2 \rfloor = (n_k - 1)/2$ and $\lceil n_k/2 \rceil = (n_k + 1)/2$, and the children of the node of size $n_k + 1$ are $\lfloor (n_k + 1)/2 \rfloor = \lceil (n_k + 1)/2 \rceil = (n_k + 1)/2$. Thus, the children nodes are of two sizes, $(n_k - 1)/2$ and $(n_k + 1)/2$.

This fact is used in our implementation, since it allows us to make at most two tables for the mapping of the data in the kernel routines—one map for n_q and another for $n_q + 1$. However, there is a special case in which $n_q = 2nb - 1$. In this case, the recursion goes one depth further. Thus, all of the children of the nodes of size $n_q + 1$ have the same size, $n_{q+1} = nb$, so we still have only two node sizes, nb and $2nb - 1$.

- *Three types of kernel routines*

In our implementation, three types of kernel routines have been considered. What distinguishes these kernels is how they deal with the nonlinear addressing of the packed recursive triangles. The first one is called a mapping kernel, and the technique is used for the Cholesky factorization kernel. Recall that this kernel operates solely on a triangular matrix, which is stored in packed recursive lower format. The ratio of the number of triangular matrix accesses to the number of operations is small, so the performance loss of copying the triangle to a full matrix in order to simplify addressing is not feasible. Also, the triangle is updated, so two copy operations would be necessary, one before the factorization and one after, to store the triangle back in packed recursive format. Instead, an address map of the structure of the elements in memory is preconstructed. As was previously shown, only two maps need be generated, M_1 for problem sizes $n_i + 1$ and M_2 for problem sizes n_i. These maps are initialized before the recursion starts. The map M_2 is used α_q times and the map M_1 is used β_q times during the execution of `BC`(N). (See Table 1 for the example $N = 743$, where $q = 5$, $\alpha_5 = 25$, and $\beta_5 = 7$.) In the kernel, the local address of the element $A(i, j)$ is computed as $loc[a(i, j)] = M_1[i + j \cdot (n_i + 1)]$ or $loc[a(i, j)] = M_2[j + i \cdot (n_i + 1)]$, depending on the problem size of the kernel. **Figure 17** illustrates the two maps generated for $n_i = 23$, perhaps for solving a problem of size $N = 743$ (see Table 2). To find the memory location of an element $A(i, j)$ in any triangle submatrix of size 24, a lookup at address $i + j \cdot 24$ of the map M_1 is done. To find the memory location of an element in any triangle submatrix of size 23, the kernel performs a lookup at address $j + i \cdot 24$ of the map M_2. Notice the order of i and j. In Section 2 we explained why packed recursive row lower format gave better performance due to better element-access stride. This format is also shown in Figure 17. For $n = 24$, the lower triangle, the locations for the elements in the rectangles are in row major order. However, for $n = 23$, the matrix map is transposed, so the elements in the map rectangles are in column major mode. Nonetheless, in both cases, the data access pattern for both kernels, when matrix elements are in the rectangle, is stride 1. Also observe that M_1 starts at the first memory position (0) of the memory occupied by the two tables, while M_2 starts at memory position 24. The reason for having one map in row order and the other map in column order was to conserve memory. We are trying to address two triangular arrays of sizes n and $n + 1$. Both of these tile into a single square array of size $(n + 1) \times (n + 1)$, as Figure 17 shows for $n = 23$.

The map design leads to one extra memory lookup and two extra index operations per reference. However, the

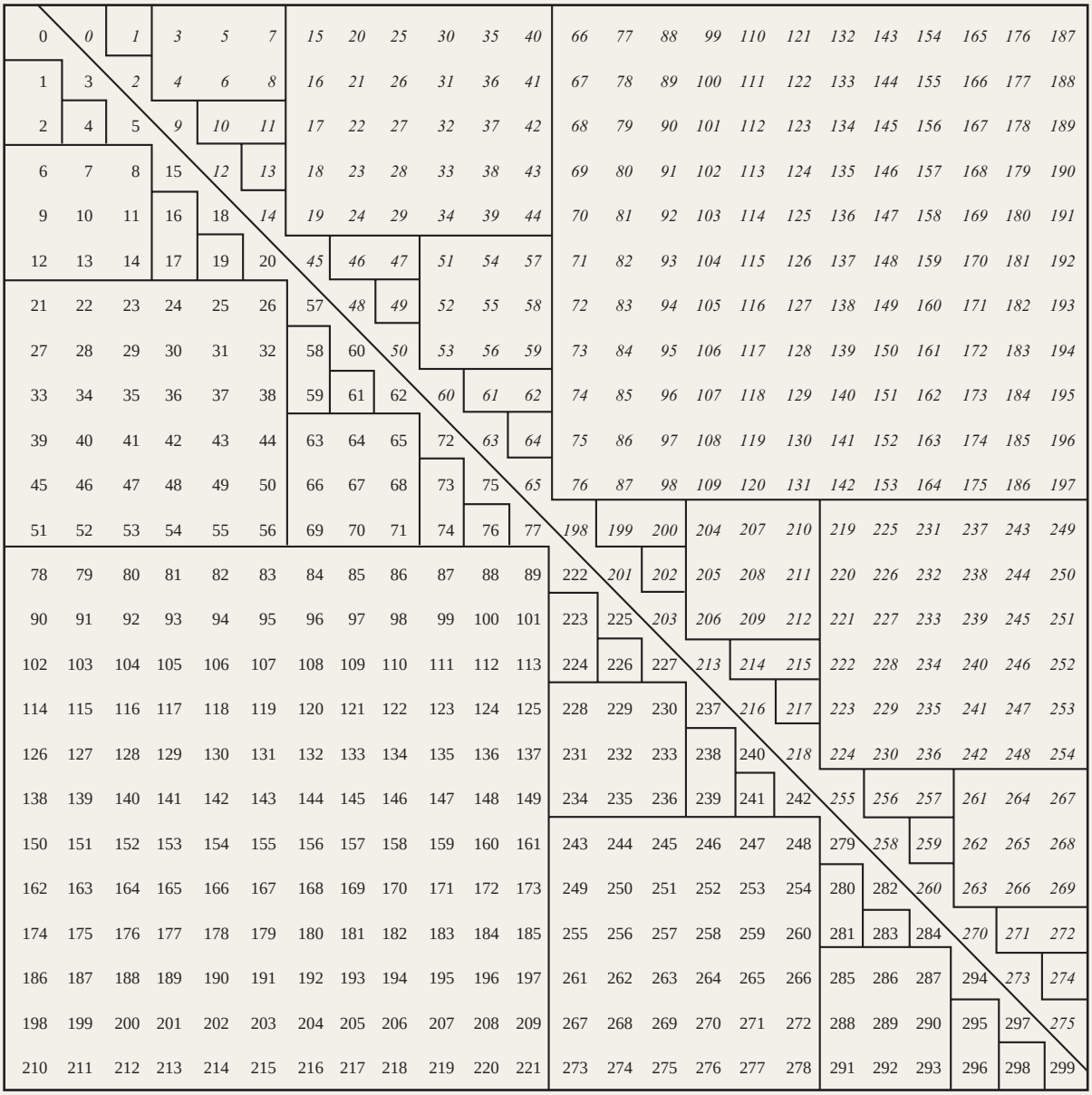

Figure 17

Maps for kernel problem sizes $n = 23$ (in italics) and $n = 24$. Viewed as a matrix of size 24×24 in column major (FORTRAN) order.

performance impact should be negligible. The extra operations are performed by the integer unit[1], which has many spare cycles in these floating-point-intensive subroutines. (This statement, however, is not true for processors which have a combined integer unit and load/store unit.) Therefore, these operations can be done in parallel with the floating-point operations. Second, many integer registers are available for use by the kernels, so several pieces of mapping data can be stored in registers for later reuse.

For the TRSM and SYRK kernels, more floating-point operations are performed, which reduces the ratio of the

[1] Sometimes referred to as the fixed-point unit, FXU.

Table 3 Methods to access nonlinearly stored triangle in kernels. Note that $nb/2 \leq n < nb$.

Operation	*No. of operations*	*No. of table accesses*	*Method used*
Cholesky	$\mathbb{O}(n^3)$	$\mathbb{O}(n^2)$	Maps
TRSM and SYRK	$\mathbb{O}(mn^2)$	$\mathbb{O}(n^2)$	Copy buffer if $m >$ *threshold*, otherwise maps
Level 2 solve	$\mathbb{O}(n^2)$	$\mathbb{O}(n^2)$	Maps, compiled code attempted

number of accesses to the number of operations. This suggests that it would be beneficial to copy the triangle to a buffer and store the triangle in full array format. This is especially true for the BT kernel, BTK, since there the triangle is only read, and thus copying is done only once. For the BS kernel, BSK, the triangle is both read and written, so data copying must be done twice. In fact, since the number of operations depends on the size of the rectangular operand as well, we use this size as a threshold. If the rectangle is large enough, the triangle is copied to a buffer.

A third method is the compiled code approach; see [10] for a reference. Here all loops have been unrolled; this means that for every line in the code, the indices are constant. Thus, the mapping indices are known, and the mapping reference can be replaced by a constant. This technique was tested for the Level 2 solve routine kernels, where the number of operations is equal to the number of triangle accesses. However, this attempt was not very fruitful, and gave inconsistent results. There are two major disadvantages to kernels produced by this approach. First, separate pieces of code are needed for every possible problem size on which the kernel operates. The code size tends to be huge. Second, this requires a high-bandwidth path to the instruction cache, and an instruction cache large enough to hold these unrolled operations. Since some processors have combined instruction and data caches, there can be a tradeoff here, between large code and large data. The compiled code approach was not used in the implementation.

The methods are summarized in **Table 3**.

- *Storage layout*

The algorithm PLPR transforms the input matrix from packed lower format to packed recursive row lower format. It is pictorially described in the subsection on data transformation, and is given in **Figure 18**. It makes use of the recursive algorithm PLPRK (**Figure 19**), which performs the data transformation out of place. In fact, if memory conservation were not an issue, PLPRK could be used alone. However, by using the fact that it is possible to do an in-place transpose of a square efficiently, PLPR reduces the needed temporary buffer space from $n^2/2$ to $n^2/8$. Notice the different meaning of the "K" in PLPRK, compared to the algorithms for block Cholesky, block TRSM, and block SYRK—e.g., BCK. PLPRK is a helper routine to PLPR in the same way as BCK, but here PLPRK is recursive, while BCK is not. On the other hand, PLPR is not recursive in itself, but BC is.

Performance is important in both PLPR and PLPRK, even though their complexity is only $\mathbb{O}(n^2)$. This is because both of these routines perform no arithmetic work, and thus only serve to reduce the MFLOP rate. To justify the new format, the speed gain given by the recursion and blocking must be large enough to cover the overhead by these routines. We accomplish this in PLPR by using optimized standard routines such as DCOPY and DGETMI, and in PLPRK we use unrolled code and a form of the compiled code approach (for a description and a reference, see the subsection on three types of kernel routines). Also, PLPRK combines blocking and recursion to reduce the recursion tree so that the calling overhead becomes negligible. In Figure 19 the code for the case $n = 4$ is written out, with the copying of the ten elements being done directly. This PLPRK algorithm does not handle cases where $n < 4$. This is not a problem because problems smaller than $nb = 16$ are solved using legacy, nonrecursive code.

4. Performance

In this section, we describe the different machines on which we have run our tests. The results for these runs are presented, both for uniprocessors and for multiprocessors using shared memory (SMP).

- *Machine characteristics*

The routine was run on three different machines. The first machine is the IBM RS/6000* SP Thin Node, with an IBM POWER2 CPU running at 120 MHz. This processor has a peak rate of 480 MFLOPs/s, an L1 data cache of 128 KB, and, due to the large memory bus, a very good peak memory bandwidth of 15.4 Gb/s.

The second machine is the IBM RS/6000 SP SMP Thin Node, with four IBM 604e CPUs running at 332 MHz. Each processor has a peak rate of 664 MFLOPs/s, an L1 data cache of 32 KB, an L2 cache of 256 KB, and a memory bandwidth of 1.3 Gb/s.

The third machine is the IBM RS/6000 SP POWER3

```
A[0:n · (n + 1)/2 − 1) = PLPR[A(0:n · (n + 1)/2 − 1), n, buf]
n1 = n/2; n2 = n − n1; ns = n1 · (n + n2 + 1)/2; ie = n1 · (n1 + 1)/2
! ie has size of T1 (see Figure 6); ns has size of T1 + S1; buf has size of T2
call PLPRK[A(0:ie − 1), n1, n, buf]                  ! Step 1, see section on data transformation
if (n2 ≠ n1) then                                    ! Save row (n1, 0:n1 − 1)
  tbuf = n2 · (n2 + 1)/2 − n1                        ! Pointer into buf
  js = n1                                            ! A(n1) = element (n1, 0)
  ji = n − 1                                         ! Initial leading dimension of row (n1, 0:n1 − 1)
  do is = 0, n1 − 1                                  ! Step 2
    buf(tbuf + is) = A(js)                           ! Copy row to end of buffer
    js = js + ji                                     ! Points to (n1, is + 1)
    ji = ji − 1                                      ! Reduce leading dimension
  enddo
  is = js − n + 1                                    ! New position of rightmost column of S1
  ie = ie + 1                                        ! New position of leftmost column of S1
else
  is = js − n                                        ! New position of rightmost column of S1
endif
js = ns − n1                                         ! js points to old position of columns, starting
ji = n2                                              ! with the rightmost; ji is leading dimension
do k = is, ie, −n1                                   ! Move fragmented S1 to compact S1
  ji = ji + 1                                        ! The leading dimension increases as we go from
  js = js − ji                                       ! left to right
  if (k ≥ js + n1) then                              ! Do source and target overlap?
    call DCOPY[n1, A(js), 1, A(k), 1]                ! Step 3, no overlap, use DCOPY
  else
    do j = js + n1 − 1, js, −1]                      ! Old and new place for column overlap
      A(k − js + j) = A(j)                           ! Step 3, overlap, use explicit do-loop
    enddo
  endif
enddo
if (n2 ≠ n1) then
  ie = ie − 1
  call DGETMI[A(ie + n1), n1, n1]                    ! Step 4, transpose S1
  call DCOPY[n1, buf(tbuf), 1, A(ie), 1]             ! Step 5, restore row (n1, 0:n1 − 1)
else
  call DGETMI[A(ie), n1, n1]                         ! Step 4, transpose S1
endif
call DCOPY ie, buf, 1, A, 1)                         ! Step 6, copy T1 back into A(0:ie − 1)
js = n2 · (n2 + 1)/2
call PLPRK[A(ns:ns + js − 1), n2, n2, buf]           ! Step 7
call DCOPY[js, buf, 1, A(ns), 1]                     ! Step 8
```

Figure 18

Algorithm for in-place transformation between packed lower format and packed recursive row lower format. (See the subsection on data transformation, Figure 11, and Table 1 for correlating material.)

```
PLPRK[A(0:n · (n + 1)/2 − 1), n, lda, B(0:n · (n + 1)/2 − 1)]
! Input: A is in packed lower format. Output: B is in packed recursive lower format
! lda is the leading dimension of the first column of A
if (n < 8) then
  if (n = 4) then
    B(0) = A(0 + 0 · lda); B(5) = A(3 + 0 · lda)
    B(1) = A(1 + 0 · lda); B(6) = A(2 − 1 · lda)
    B(2) = A(0 + 1 · lda); B(7) = A(−1 + 2 · lda)
    B(3) = A(2 + 0 · lda); B(8) = A(0 + 2 · lda)
    B(4) = A(1 + 1 · lda); B(9) = A(−3 + 3 · lda)
  else if (n = 5) then
    . . .
  else if (n = 6) then
    . . .
  else if (n = 7) then
    . . .
  endif
else
  n1 = n/2; n2 = n − n1
  ! Upper left triangle, T1 in Figure 6:
  call PLPRK{A[0:n1 · (n1 + 1)/2 − 1], n, lda, B[0:n1 · (n1 + 1)/2 − 1]}
  ns = n1 · (n + n2 + 1)/2; ms = n1 · (2m − n1 + 1)/2; is = n1 · (n1 + 1)/2
  ie = is + n1 · (n2 − 1); js = n1
  ! Rectangular block, S1 in Figure 6:
  do k = is, ie − 15n1, 16n1 ! Copy, transpose, compact rectangle, 16 columns at a time
    ji = m − 1; j1 = js
    do j = k, k + n1 − 1
      t1 = A(j1 + 0)
      ⋮
      t16 = A(j1 + 15)
      j1 = j1 + ji; ji = ji − 1
      B(j + n1 · 0) = t1
      ⋮
      B(j + n1 · 15) = t16
    enddo
    js = js + 16
  enddo

  ⋮ ! Fix up code for the cases when n1 is not a multiple of 16
  ! Lower right triangle, T2 in Figure 6:
  call PLPRK{A[ns:n · (n + 1)/2 − 1], n2, lda − n1, B[ns:n · (n + 1)/2 − 1]}
endif
```

Figure 19

Algorithm for out-of-place transformation between packed lower format and packed recursive row lower format. (See the subsection on data transformation, Figure 11, and Table 1 for correlating material.)

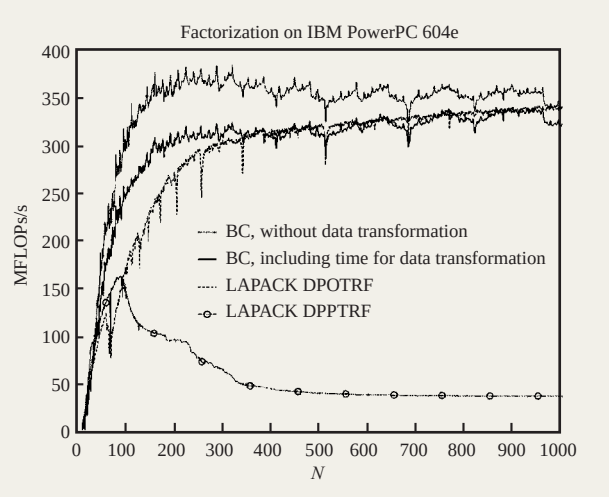

Figure 20

Performance for Cholesky factorization on the IBM 604e. Peak rate is 664 MFLOPs/s.

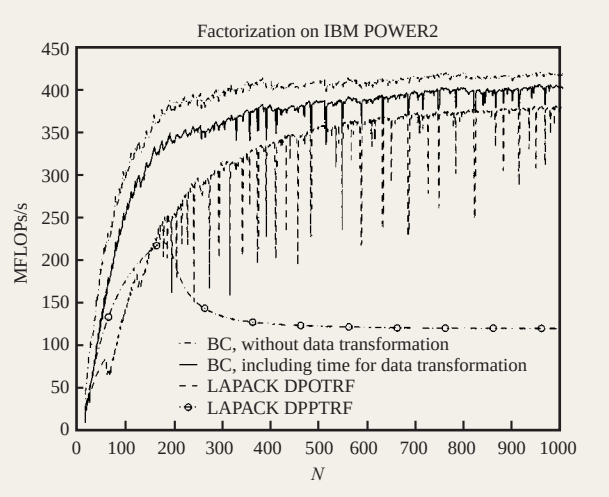

Figure 21

Performance for Cholesky factorization on the IBM POWER2. Peak rate is 480 MFLOPs/s.

SMP Thin Node, with two IBM POWER3 CPUs running at 200 MHz. Each processor has a peak rate of 800 MFLOPs/s, an L1 data cache of 64 KB, and an L2 cache of 4 MB. The memory bandwidth is 12.8 Gb/s; see page 88 of [11] for further information.

Please note the different use of the term *level*. A Level 1 cache (L1 cache) is the cache closest to the processor, on the processor chip itself. A Level 2 cache (L2 cache), if it exists, is situated between the Level 1 cache and the memory, or other caches (Level 3 cache).

- *Uniprocessor results*

In the preceding subsection, we discussed levels of memory; i.e., L1 cache, L2 cache, etc. Here we also discuss Level 2 and Level 3 routines. We briefly define these terms so as to better clarify the discussions to follow. A Level 1 BLAS routine is a vector–vector operation, a Level 2 BLAS routine is a matrix–vector operation, and a Level 3 BLAS routine is a matrix–matrix operation.

The IBM 604e chip has better floating-point arithmetic performance than memory bandwidth. It is therefore difficult to reach peak performance, since the machine design does not allow acceptable data reuse because of its tiny line size and tiny Level 1 cache. The LAPACK *uplo* = 'L' `DPPTRF` routine (Figure 4) is a Level 2 BLAS algorithm, and thus has poor data reuse. As a Level 2 BLAS algorithm, it does not block for any cache level, so performance drops when the cache is exhausted. This is seen in **Figure 20** at two points, one just below $N = 100$, when the Level 1 cache is full, and one at $N = 240$, just before the Level 2 cache is full. However, the recursive algorithm runs just as well as the LAPACK *uplo* = 'L' `DPOTRF` routine (Figure 3), a blocked Level 3 BLAS algorithm. Notice that the performance of the unblocked `DPPTRF` routine is bounded by the memory bandwidth. At the beginning of Section 4 we stated that the memory bandwidth was 1.3 Gb/s, or 20 MDW/s[2]. Each loaded element is used in approximately two floating-point operations before it is flushed out of cache; this, together with overhead for the table lookaside buffer (TLB), yields a large-problem performance of about 38 MFLOPs/s. In the `DPOTRF` graph, "dips" can be found corresponding to bad leading-dimension sizes, which cause poor usage of the four-way-associative cache. A leading dimension is bad when a small multiple of it is close to some multiple of 2^k. ("Close" means that the multiple lies in a window of the cache's line size.) The bad-leading-dimension problem is created by the way in which data from memory is mapped into the Level 1 cache. This mapping is based on cache congruence classes (page 568 of [7]). One example is 682, which multiplied by 3 is 2046, which lies within the small window of (2044, 2052).

The IBM POWER2 chip has a very good bandwidth due to its large memory bus of 256 bits (four doublewords). This is shown in **Figure 21**, where the `DPPTRF` routine does better compared to the peak rate on this machine than for the 604e. Still, it levels out at about a fourth of the peak rate. On this machine, the recursive algorithm

[2] One doubleword (DW) = 64 bits.

beats the LAPACK DPOTRF routine by a clear margin. Also, it uses about half the memory. The lack of a Level 2 cache on the POWER2 amplifies the effect of the congruence classes of the Level 1 cache, with very deep dips for DPOTRF. This is not the case for the recursive BC subroutine, because it uses many different leading dimensions in its recursive data structure.

The IBM POWER3 chip also has a very good bandwidth and built-in prefetches, which automatically detect vector accesses and prefetch them. These detectors recognize stride 1 data access patterns, and prefetch data which are predicted to be needed before they are actually called, thereby reducing the memory-latency overhead. Together, these reasons explain the good relative performance for the LAPACK DPPTRF routine. On this machine, the recursive routine also beats the LAPACK DPOTRF routine which uses full storage; see **Figure 22**. The Level 1 cache on the POWER3 chip is 128-way-associative, which almost eliminates the cache congruence-class effect. The only remaining dips are some small ones at $N = 256, 512, 768$, with $N = 512$ having the largest performance decrease.

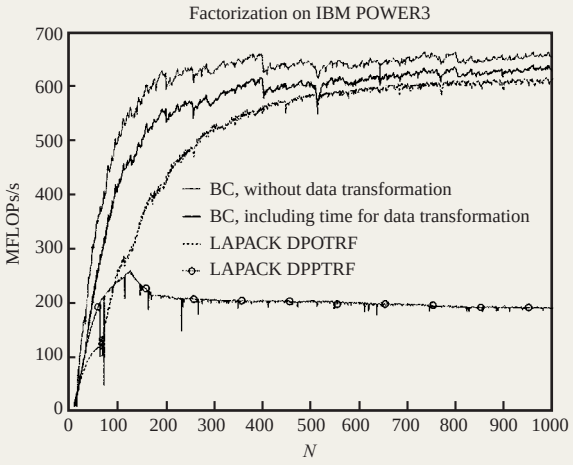

Figure 22

Performance for Cholesky factorization on the IBM POWER3. Peak rate is 800 MFLOPs/s.

- *Multiprocessor results*

We have parallelized the Cholesky routine by simply linking to an SMP version of DGEMM. The results for one SMP machine, a two-way POWER3 machine, are shown in **Figure 23**. The speedup is not impressive for small problems. At $N = 1000$, it is about 1.4 for POWER3. For larger problems, the speedup reaches more than 1.6. Also, this speedup is reached only by using an SMP version of the DGEMM routine. All other code is compiled without using any parallelization flags. This is not possible to do for the LAPACK DPPTRF routine, since it cannot make use of DGEMM because of the packed data structure. Also, DPOTRF is not as good with SMP DGEMM because it has fixed block size; take a look at SMP DGEMM plots for DPOTRF for large problem sizes. Figure 23 shows that algorithm BC, with fixed $NB = 16$, outperforms DPOTRF except for cases in which the blocking size for DPOTRF equals 64 and $400 \leq N \leq 1200$, where the performance is about equal.

We briefly explain why. The FLOPs of Algorithm BC follow a geometric progression. At Level 0, three quarters of the FLOPs are consumed in two calls to Algorithms BT and BS. At Level 1, three quarters of the remaining quarter of the FLOPs are consumed, etc. Now, both BT and BS call DGEMM with operands whose size may vary according to the recursion tree level. In essence, Algorithm BC spends most of its computing time in two calls to DGEMM at Level 0 of the recursion tree. On the other hand, the floating-point operations of Algorithm DPOTRF follow an arithmetic progression (for a figure, see [8]). Specifically, the N dimension of DGEMM is always fixed at or below NB, the block size. Hence, the C matrix of DGEMM cannot be easily broken into large pieces for SMP DGEMM to work on.

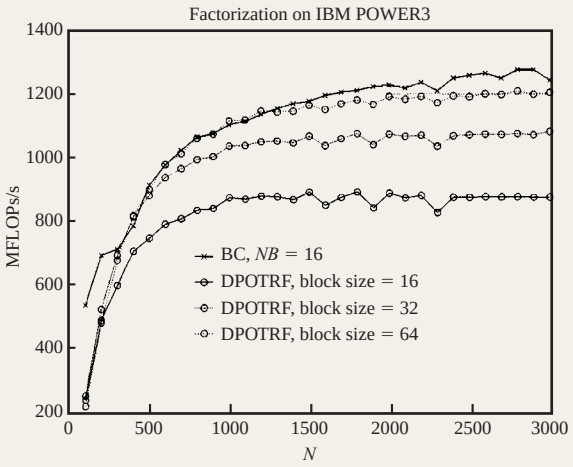

Figure 23

Performance for Cholesky factorization on the IBM POWER3; peak rate is 2×800 MFLOPs/s.

This means that parallel DPOTRF works well for parallel machines with only a few processors. On the other hand, BC should scale with more processors because of the geometric nature of the recursive algorithm. In Figure 23,

this is shown as DPOTRF performs worst with small NB, whereas BC performs better for small NB, as more FLOPs are being performed in SMP DGEMM with the C matrix large.

5. Conclusions

We have presented a novel algorithm for Cholesky factorization. It has three attractive features: It uses minimal storage, it attains Level 3 performance, and it uses standard packed format for input. The last feature, perhaps the most important, means that existing codes can use the new algorithm; i.e., it is portable and migratable. The new algorithm outperforms the standard algorithms DPPFA, DPOFA of Linpack and DPPTRF, DPOTRF of LAPACK. Compared to the standard packed routines, the performance ratio is greater than 4, sometimes as large as 7, when N is large. Compared to DPOTRF, the performance was greater by 31%, 40%, and 11% at $N = 200$ and 5%, 14%, and −5% at $N = 1000$ for the three IBM RISC machines POWER3, POWER2, and PowerPC 604e. These figures include the cost of the data transformation from lower packed to recursive lower packed row format. The memory bandwidth characteristics of these three machines decrease downward from POWER2 to POWER3 to PowerPC. Our performance results demonstrate this characteristic; i.e., the new algorithm is better able to exploit machines with better memory bandwidth. The performance is also more uniform, especially on POWER2, than for LAPACK DPOTRF. For example, on POWER2, for certain values of N, LAPACK DPOTRF performance drops by more than a factor of 2, whereas for the new algorithm it is closer to 10%.

In comparing the new algorithm to full-format algorithms, we note that the storage requirement is nearly a factor of 2 less. However, temporary storage of $\frac{1}{8}N^2$ elements is required for the data transformation. Finally, as N becomes very large, say $N > 1000$, and approaching $N = 4000$, the performance of the new algorithm gradually increases, whereas LAPACK DPOTRF levels off at its $N = 1000$ value. These five factors, namely portability and migratability, better performance, less storage, more uniformity, and slow improvement for large values of N, lead to the following conclusion: The use of packed format for Cholesky factorization will become a competitive method of choice over full format.

Several less important results were also demonstrated. A blocked form of the pure recursive algorithm was introduced and completely analyzed. The results showed that the overhead due to recursion became negligible. Some novelty was introduced to make this possible: The use of two-dimensional mappings and data copying was introduced. We showed that only two mappings were necessary, so the extra storage requirements become negligible. We also demonstrated that the percentage of FLOPs spent in DGEMM approach $1 - (3nb/2N)$ as N becomes large. This means that a parallel SMP version of the algorithm becomes automatic by merely producing a parallel SMP DGEMM.

We mention that recursion is ideally suited to Cholesky factorization. For other dense linear algorithms, the use of recursion may not be as relevant. For Cholesky, the reason can be seen in Figure 1 by noting that the recursive part truly divides the problem into a smaller instance of itself. To see this, note that the two children of the parent node Cholesky-factor a problem of half the size. On the other hand, other dense linear algebra problems such as dense LU with partial pivoting, QR factorization, reduction to tridiagonal form, and symmetric indefinite factorization algorithms all have some global feature that remains during their factorization phase. This global feature of the factorization phase appears to be inherent. It means that a sizable part of the matrix is needed for each of the "rank one pivot" steps. This implies that data movement from memory, Level 2 cache, and any cache levels between these cannot be avoided, and so the FLOPs consumed during the "rank one pivot" step cannot be Level-3-like; i.e, they must be either Level-2- or Level-1-like. For Cholesky, this is not true. The factor phase can be made Level-3-like because one merely factors an order n positive definite matrix which definitely fits in the Level 1 cache. In summary, for Cholesky, it definitely appears that recursion- and blocked-based formats will become the method of choice.

To summarize: We have demonstrated in this paper every reason to make a change. There were no negative results other than "status quo." Even there, we adapted our algorithm to accept the older data format of the status quo, so even that objection tends to disappear.

There are some issues we have not covered. In ESSL, there are also new algorithms for LDL^T factorization and matrix inverse where A is positive definite symmetric. We produced new algorithms similar to the algorithms presented here, and the results were essentially the same, although we do not discuss the results here owing to lack of time and space. On a more positive note, we had wondered in February 1999 if newly proposed BLAS libraries for packed formats would pick up these ideas [12]. As of December 1999, we learned that the new Level 3 BLAS routines for packed format were dropped, partly because of results such as those appearing in this paper. Finally, we note that Level 2 routines using the new recursive formats must use these formats as input; i.e., they cannot transform from packed format to recursive packed format as is done in the Level 3 algorithms. The reason is fairly simple. We cannot do a packed-to-recursive-packed data transform, followed by a recursive packed Level 2 BLAS, followed by a recursive-

packed-to-packed data transform. This is not practical because each of the two data transformations is as costly to perform as doing the Level 2 BLAS in the original packed data format.

Finally, a natural question arises: Does this new method produce the fastest Cholesky factorization routine for today's RISC-type processors? The answer is no. To see why this is so, recall that most of the FLOPs of our algorithms are consumed in DGEMM. Surprisingly, "DGEMM" performance on Cholesky factorization can be improved, but only by changing the input data format of AP. In [6, 8, 9] we indicate why. Briefly, for dense Level 3 factorization routines (Cholesky is a prime example), DGEMM is called multiple times on submatrices of the matrix A that is to be factored. A generic DGEMM does not know about this relationship and hence cannot exploit it. To generally exploit these relationships one must change the input data format; see again [6, 8, 9]. When this is done, the new DGEMM will work on the new data format, which is especially tailored to the factorization routine. The new DGEMM becomes simpler and faster because any data copying of the old DGEMM is avoided.

In this paper, we have partially applied the principle: We changed the input format AP to PRLF. However, we kept the resulting $n - 1$ rectangular (nearly square) arrays in standard row major order, with the leading dimension being n_i, $n_i + 1$, $i = 1, \cdots, q$. As is argued in [6, 8, 9], standard row and column major format is *not* the best input format for DGEMM when it is applied to dense linear algebra factorization routines. And since the standard format is used by the CHOL and BC algorithms here, performance will be sub-optimal.

The "better" data formats require some expansion of the input array space AP, whereas the current algorithms CHOL and BC do not. Hence, CHOL and BC are more portable and migratable than the other "better" formats. Also, tuned DGEMM routines are widely available on many platforms.

Appendix: Solution for Equation (16) and bounds for the solution

Equations (16) to (18), (19) to (22), and (23) define the cost $C(n)$ of Cholesky factorization in terms of t_{call} and t_{flop}. Before proceeding, note that for a given n, the recursive algorithm CHOL of Figure 2 makes multiple calls to the recursive algorithm TRSM of Figure 9 and the recursive algorithm SYRK of Figure 10. In fact, for each interior node of the recursion tree of Figure 1 there is a single call to both TRSM and SYRK. We define integer functions $st(n)$ and $ss(n)$ which sum together all of these recursive calls. Also, $c(n)$ and $g(n)$ represent respectively the number of calls to the Cholesky factor routine and to the GEMM routine. Here we forget about t_{flop} and compute only the part of $C(n)$ that pertains to t_{call}. We may write

Table 4 Number of calls to the different subroutines for some values of n.

n	$c(n)$	$st(n)$	$ss(n)$	$g(n)$
1	1	0	0	0
2	3	1	1	0
3	5	2	4	1
4	7	5	5	2
5	9	6	10	4
6	11	9	13	6
7	13	12	16	8
8	15	17	17	10
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.

$$C(n) = c(n)t_C + st(n)t_T + ss(n)t_S + g(n)t_G, \tag{A1}$$

where t_C, t_T, t_S, and t_G denote the $t_{\text{call}C}$, $t_{\text{call}T}$, $t_{\text{call}S}$, and $t_{\text{call}G}$ of Equation (23). We set $nb = 1$ here; i.e., we consider the full recursive call where there is no blocking factor. Using (16)–(23) with $nb = 1$, we obtain **Table 4**.

Our aim is to find general expressions for c, st, ss, and g. To do so we first write some recursive equations. From Equations (17) and (18) we find

$$T(n, m) = (2n - 1)t_T + (n - 1)t_G \tag{A2}$$

and

$$S(n, m) = (2n - 1)t_S + (n - 1)t_G. \tag{A3}$$

Using Equations (16), (A1), (A2), and (A3) and comparing coefficients of t_C, t_T, t_S, and t_G, we find that

$$c(n) = c(n_1) + c(n_2) + 1, \tag{A4}$$

$$st(n) = st(n_1) + st(n_2) + 2n_1 - 1, \tag{A5}$$

$$ss(n) = ss(n_1) + ss(n_2) + 2n_2 - 1, \tag{A6}$$

$$g(n) = g(n_1) + g(n_2) + n - 2, \tag{A7}$$

where $c(1) = 1$ and $st(1) = ss(1) = g(1) = 0$. Also, n_1 denotes $\lfloor n/2 \rfloor$, and n_2 denotes $\lceil n/2 \rceil$. Now Equation (A4), like (17) and (18), is easily solved to give

$$c(n) = 2n - 1. \tag{A8}$$

Similarly, it is fairly easy to see, via induction, that

$$g(n) = 2 + jn - 2^{j+1}, \tag{A9}$$

where $j = \lfloor \log_2 n \rfloor$. Now we turn to solving (A5) and (A6). Let

$$sa(n) = \tfrac{1}{2}[ss(n) + st(n)] \tag{A10}$$

and

$$d(n) = \tfrac{1}{2}[ss(n) - st(n)]. \tag{A11}$$

Table 5 Deviation in number of calls between SYRK and TRSM for some values of n.

n	$d(n)$	n	$d(n)$	n	$d(n)$	n	$d(n)$
1	1	9	3	17	4	25	10
2	0	10	4	18	6	26	10
3	1	11	5	19	8	27	10
4	0	12	4	20	8	28	8
5	2	13	5	21	10	29	8
6	2	14	4	22	10	30	6
7	2	15	3	23	10	31	4
8	0	16	0	24	8	32	0

Table 6 First maximum values of $d(n)$ in the interval $[2^j, 2^{j+1}]$ for some j.

j	2^j	$n(j)$	$d[n(j)]$
1	2	3	1
2	4	5	2
3	8	11	5
4	16	21	10
5	32	43	21
6	64	85	42
7	128	171	85
8	256	341	170

Here *sa* stands for the average of *st* and *ss*, and *d* stands for their deviation. Then, using (A5), (A6), (A10), and (A11), we obtain

$$sa(n) = sa(n_1) + sa(n_2) + n - 1 \tag{A12}$$

and

$$d(n) = d(n_1) + d(n_2) + n_2 - n_1, \tag{A13}$$

with $sa(1) = d(1) = 0$. Again, it is fairly easy to see, via induction, that a solution to (A12) is

$$sa(n) = 1 + (j+1)n - 2^{j+1}. \tag{A14}$$

To find a closed-form expression for $d(n)$, we need the following.

Lemma $\alpha_i d(n_i) + \beta_i d(n_i + 1) - \gamma_i = \alpha_{i+1} d(n_{i+1}) + \beta_{i+1} d(n_{i+1} + 1)$, where $\gamma_i = \alpha_i$ if n_i is odd and $\gamma_i = \beta_i$ if n_i is even. The terms α_i, β_i, and n_i are defined in Theorem 2.

Proof Let n_i be odd. Then $\alpha_{i+1} = \alpha_i$, $\beta_{i+1} = \alpha_i + 2\beta_i$, $d(n_i) = d(n_{i+1}) + d(n_{i+1} + 1) + 1$, and $d(n_i + 1) = 2d(n_{i+1} + 1)$. Thus, $\alpha_i d(n_i) + \beta_i d(n_i + 1) - \gamma_i = \alpha_i[d(n_{i+1}) + d(n_{i+1} + 1) + 1] + 2\beta_i d(n_{i+1} + 1) - \alpha_i = \alpha_i d(n_{i+1}) + (\alpha_i + 2\beta_i) d(n_{i+1} + 1) = \alpha_{i+1} d(n_{i+1}) + \beta_{i+1} d(n_{i+1} + 1)$. Let n_i be even. Then $\alpha_{i+1} = 2\alpha_i + \beta_i$, $\beta_{i+1} = \beta_i$, $d(n_i) = 2d(n_{i+1})$, and $d(n_i + 1) = d(n_{i+1}) + d(n_{i+1} + 1) + 1$. Thus, $\alpha_i d(n_i) + \beta_i d(n_i + 1) - \gamma_i = 2\alpha_i d(n_{i+1}) + \beta_i d(n_{i+1}) + \beta_i d(n_{i+1} + 1) + \beta_i - \beta_i = (2\alpha_i + \beta_i) d(n_{i+1}) + \beta_i d(n_{i+1} + 1) = \alpha_{i+1} d(n_{i+1}) + \beta_{i+1} d(n_{i+1})$. Since n_i must be odd or even, we are done. □

Corollary Let $N = n_0$. Then $d(N) = \alpha_i d(n_i) + \beta_i d(n_i + 1) + \sum_{k=0}^{i-1} \gamma_k$.

Proof We use mathematical induction. For $i = 0$, $d(N) = d(n_0)$. Assume that the result is true for $i = j$. The induction hypothesis is $d(N) = \alpha_j d(n_j) + \beta_j d(n_j + 1) + \sum_{k=0}^{j-1} \gamma_k$. Let $x = d(N) - \sum_{k=0}^{j-1} \gamma_k$. By the lemma and the induction hypothesis, $x = \alpha_j d(n_{j+1}) + \beta_{j+1} d(n_{j+1} + 1) + \gamma_j$, and the result follows.

Corollary $d(N) = \sum_{k=0}^{q-1} \gamma_k$, where q satisfies $2^q \le N < 2^{q+1} - 1$.

Proof Set $i = q$. Then $n_q = 1$, and so $d(n_q) = d(n_q + 1) = 0$. □

In **Table 5** we list some values of $d(n)$.

We now give some additional results with respect to $d(n)$. It is clear that $d(2^k) = 0$; also, $d(n)$ is a symmetric function between two successive powers of 2. Hence, we need to compute $d(n)$ from 2^j to $3 \cdot 2^{j-1}$. Now we claim that in the interval $[2^j, 2^{j+1}]$, d takes its first maximum value at $n(j) = 1 + \frac{2}{3}(2^{k+1} - 1)(2^{k+1} + 1)$ for $j\ (= 2k + 1)$ odd and $n(j) = \frac{1}{3}(2^{k+1} - 1)(2^{k+1} + 1)$ for $j\ (= 2k)$ even. For j odd, the maximum value is $d[n(j)] = \frac{1}{3}(2^{k+1} - 1)(2^{k+1} + 1)$; for j even, the maximum value is $d[n(j)] = \frac{2}{3}(2^{k+1} - 1)(2^{k+1} + 1)$. These claims can be established using mathematical induction. In **Table 6** we give some of the first maximum values of d.

Without proof we state that the following piecewise linear function bounds d from above. Consider the interval $[2^j, n(j)]$. We partition $[2^j, n(j)]$ into $[x_0, x_1], [x_1, x_2], \cdots, [x_{j-2}, x_{j-1}]$, where $x_i = 2^j + [0, 1, 3, \cdots, n(j-2)]$. Note that $n(0) = 1$. With each of these $j - 1$ intervals, associate ordinate intervals $[y_0, y_1], [y_1, y_2], \cdots, [y_{j-2}, y_{j-1}]$, where $y_0 = 0$. The slopes of lines in each of these intervals will be $m = j, j-2, j-3, \cdots, 1$. Now the piecewise linear majoring function is obtained as follows: Start at interval $[x_0, x_1]$, $[y_0, y_1]$. Use the first slope $m = j$ to compute y_1. Use the second slope $m = j - 2$ to compute y_2. Follow this procedure for the rest of the intervals. This defines y_2, etc.

From (A10) and (A11) we can solve for

$$st(n) = sa(n) - d(n) \tag{A15}$$

and

$$ss(n) = sa(n) + d(n). \tag{A16}$$

Let $t_A = (t_T + t_S)/2$ and $t_D = (t_S - t_T)/2$ be the average and deviation t of t_T and t_S. Then, using (A15) and (A16), we obtain

$$C(n) = c(n)t_C + 2sa(n)t_A + g(n)t_G + 2d(n)t_D . \qquad \text{(A17)}$$

Note that if $t_S = t_T$, $d(n)$ does not even enter into the computation.

Acknowledgments

We are grateful to Bo Kågström and Carl Tengwall, who were instrumental in setting up our fruitful collaboration. We especially thank Joan McComb and John Lemek for their help in getting our code ready for inclusion in ESSL. We thank the anonymous referee for carefully reading the manuscript. Finally, we thank the Free Lunch Foundation for improving the accuracy of our manuscript.

References

1. B. S. Andersen, F. G. Gustavson, and J. Waśniewski, "A Recursive Formulation of the Cholesky Factorization Operating on a Matrix in Packed Storage Form," *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, *PPSC99*, B. Hendrickson, K. A. Yelick, C. H. Bischof, I. S. Duff, A. S. Edelman, G. A. Geist, M. T. Heath, M. A. Heroux, C. Koelbel, R. S. Schreiber, R. F. Sincovec, and M. F. Wheeler, Eds., San Antonio, March 24–27, 1999; Scientific Computing (CD-ROM), Society for Industrial and Applied Mathematics, Philadelphia.
2. E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK User's Guide*, third edition, Society for Industrial and Applied Mathematics, Philadelphia, 1999.
3. IBM Corporation, *Engineering and Scientific Subroutine Library for AIX, Guide and Reference*, third edition, October 1999; Order No. SA 22-7272-02, available through IBM branch offices.
4. Jack J. Dongarra, James R. Bunch, Cleve B. Moler, and G. W. Stewart, *LINPACK User's Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1979.
5. Gene H. Golub and Charles Van Loan, *Matrix Computations*, third edition, Johns Hopkins University Press, Baltimore, 1996.
6. F. G. Gustavson, A. Henriksson, I. Jonsson, B. Kågström, and P. Ling, "Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms," *Applied Parallel Computing*, *PARA'98*, B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski, Eds., Vol. 1541 of *Lecture Notes in Computer Science*, pp. 195–206; Springer-Verlag, New York, 1998.
7. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Improving Performance of Linear Algebra Algorithms for Dense Matrices, Using Algorithmic Prefetch," *IBM J. Res. Develop.* **38,** No. 3, 265–275 (1994).
8. F. G. Gustavson, "Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms," *IBM J. Res. Develop.* **41,** No. 6, 737–755 (1997).
9. A. Henriksson and I. Jonsson, "High-Performance Matrix Multiplication on the IBM SP High Node," *Master's Thesis UMNAD 98.235*, Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden, June 1998.
10. F. G. Gustavson, W. Liniger, and R. Willoughby, "Symbolic Generation of an Optimal Crout Algorithm for Sparse Systems of Linear Equations," *J. ACM* **17,** No. 1, 87–109 (1970).
11. S. Andersson, R. Bell, J. Hague, H. Holthoff, P. Mayes, J. Nakano, D. Shieh, and J. Tuccillo, *RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide*, first edition, IBM Corporation, October 1998; Order No. SG24-5155-00.
12. *Document for the Basic Linear Algebra Subprograms (BLAS) Standard*, draft edition, Basic Linear Algebra Subprograms (BLAST) Forum, October 1999. Available at *http://www.netlib.org/cgi-bin/checkout/blast/blast.pl*.

Received June 7, 2000; accepted for publication July 28, 2000

Fred G. Gustavson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598 (GUSTAV at YKTVMV, gustav@watson.ibm.com).*
Dr. Gustavson manages the Algorithms and Architectures group in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center. He received his B.S. degree in physics, and his M.S. and Ph.D. degrees in applied mathematics, all from Rensselaer Polytechnic Institute. He joined IBM Research in 1963. One of his primary interests has been in developing theory and programming techniques for exploiting the sparseness inherent in large systems of linear equations. Dr. Gustavson has worked in the areas of nonlinear differential equations, linear algebra, symbolic computation, computer-aided design of networks, design and analysis of algorithms, and programming applications. He and his group are currently engaged in activities that are aimed at exploiting the novel features of the IBM family of RISC processors. These include hardware design for divide and square root, new algorithms for POWER2 for the Engineering and Scientific Subroutine Library (ESSL) and for other math kernels, and parallel algorithms for distributed and shared memory processors. Dr. Gustavson has received an IBM Outstanding Contribution Award, an IBM Outstanding Innovation Award, an IBM Invention Achievement Award, two IBM Corporate Technical Recognition Awards, and a Research Division Technical Group Award. He is a Fellow of the IEEE.

Isak Jonsson *Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden (isak@cs.umu.se).*
Mr. Jonsson is a graduate student and lecturer in parallel computing in the Department of Computing Science, Umeå University, Sweden. He received his M.S. degree in computing science from Umeå University in 1998. His research interests include dense linear algebra kernels for high-performance computing (HPC) platforms. He is currently focusing on recursion and superscalar techniques in order to produce high-performance algorithms, as well as compiler techniques to automate this work. Mr. Jonsson has also participated in the ACM International Collegiate Programming Contest, becoming European Champion in 1997 and achieving fourth place in the World Finals in 1998; he is currently a local organizer.