A Fully Portable High Performance Minimal Storage Hybrid Format Cholesky Algorithm

Bjarne S. Andersen
UNI•C Danish IT Center for Education and Research
and
John A. Gunnels and Fred G. Gustavson
IBM T.J. Watson Research Center
and
John K. Reid
Atlas Centre, Rutherford Appleton Laboratory
and
Jerzy Waśniewski
Technical University of Denmark

We consider the efficient implementation of the Cholesky solution of symmetric positive-definite dense linear systems of equations using packed storage. We take the same starting point as that of LINPACK and LAPACK, with the upper (or lower) triangular part of the matrix being stored by columns. Following LINPACK and LAPACK, we overwrite the given matrix by its Cholesky factor. We consider the use of a hybrid format in which blocks of the matrices are held contiguously and compare this to the present LAPACK code. Code based on this format has the storage advantages of the present code, but substantially outperforms it. Furthermore, it compares favourably to using conventional full format (LAPACK) and using the recursive format of Andersen, Gustavson, and Waśniewski.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra – Linear Systems (symmetric and Hermitian); G.4 [**Mathematics of Computing**]: Mathematical Software

General Terms: Algorithms, BLAS, Performance.

Additional Key Words and Phrases: real symmetric matrices, complex Hermitian matrices, positive definite matrices, Cholesky factorization and solution, recursive algorithms, novel packed matrix data structures, linear systems of equatons.

1. INTRODUCTION

It was apparent by the late 1980s, when the Level-3 BLAS [Dongarra et al. 1990] were designed, that blocking would be needed to obtain high performance on com-

Authors' addresses: B. A. Andersen, UNI•C, Building 304, DTU Lyngby, Denmark DK-2800; J.A. Gunnels and F.G. Gustavson, IBM T.J. Watson Research Center, Yorktown Heights NY, 10598, USA; J. K. Reid, Atlas Centre, Rutherford Appleton Laboratory, Didcot, Oxon OX11 OQX, UK; J. Waśniewski, Department of Informatics and Mathematical Modeling, Building 305, DTU Lyngby, Denmark DK-2800.

puters with multi-level memory systems (see, for example, Calahan [1986], IBM [1986], and Gallivan, Jalby, Meier, and Sameh [1987]). This approach is now firmly established. More recently, it has become recognized that the most significant Level-3 BLAS is `_GEMM`, which performs the computation

$$C = \alpha op(A)op(B) + \beta C \tag{1}$$

where A, B, and C are matrices held in conventional full storage, $op(A)$ is A or A^T, $op(B)$ is B or B^T, and α and β are scalars. Agarwal, Gustavson, and Zubair [1994] and, later, Whaley, Petitet, and Dongarra [2000], point out that this can be performed particularly efficiently when $op(A) = A^T$ and $op(B) = B$ on any cache-based machine if A is small enough to remain in the level-1 cache while the columns of B and C are read into the cache and the changed columns of C are written out. The cache does not need to be big enough to hold all three arrays; rather, it needs to be big enough to hold A and a few columns of B and C. This mode of working is called 'streaming'. Note that it requires B and C to be held by columns. Multiplying by A^T rather than A facilitates the calculation of inner products.

If A is too big for the cache or B or C is not held by columns, the benefits of streaming may be obtained by making copies in contiguous memory of blocks of a suitable size. We will refer to this as 'data copy'. For very large matrices, this will be an insignificant overhead, but for medium-sized blocks arising within a bigger calculation such as Cholesky factorization, the data-copy overhead may be significant.

On a computer with a single level of cache, it is easy enough to choose a suitable block size. With more than one level of cache, nested blocking is desirable and Gustavson [1997], Chatterjee, Jain, Lebeck, Mundhra, and Thottethodi [1999], Valsalam and Skjellum [2002], and Frens and Wise [1997] achieve this by recursive nesting, which has the added advantage that there is no need to choose a block size. In [Waśniewski et al. 1998; Andersen et al. 2001; Andersen et al. 2002], the recursive blocking is applied to triangular matrices in full and packed storage format.

In designing the Level-3 BLAS, Dongarra, Du Croz, Duff, and Hammarling [1990] chose not to address packed storage schemes for symmetric, Hermitian or triangular matrices because 'such storage schemes do not seem to lend themselves to partitioning into blocks ... Also packed storage is required much less with large memory machines available today'. In this paper, our aim is to demonstrate that packing is possible without any loss of performance. While memories continue to get larger, problems that people solve get larger too and there will always be an advantage in saving storage.

We achieve this by using a blocked hybrid format in which each block is held contiguously in memory. It avoids the data copies that are inevitable when Level-3 BLAS are applied to matrices held conventionally in rectangular arrays. Note, too, that many data copies may be needed for the same submatrix in the course of a Cholesky factorization [Gustavson 1997].

The rest of the paper is organized as follows. Our proposed blocked hybrid format for a lower-triangular matrix is explained in Section 2. How Cholesky factorization and solution of equations can be implemented using this format is described in Sections 3 and 4, respectively. In Section 5, we consider the lower-triangular case

1a. Lower Packed Format

0									
1	10								
2	11	19							
3	12	20	27						
4	13	21	28	34					
5	14	22	29	35	40				
6	15	23	30	36	41	45			
7	16	24	31	37	42	46	49		
8	17	25	32	38	43	47	50	52	
9	18	26	33	39	44	48	51	53	54

1b. Lower Blocked Hybrid Format

0									
1	2								
3	4	5							
6	7	8	27						
9	10	11	28	29					
12	13	14	30	31	32				
15	16	17	33	34	35	45			
18	19	20	36	37	38	46	47		
21	22	23	39	40	41	48	49	50	
24	25	26	42	43	44	51	52	53	54

Fig. 1. Lower Packed and Blocked Hybrid Formats.

similarly. A kernel code for triangular factorization of the blocks on the diagonal of our blocked form is needed and we explain this in Section 6. The results of performance testing, with comparisions against conventional full format (LAPACK) and the recursive format of Andersen, Gustavson, and Waśniewski, are given in Section 7. We also show the (much inferior) performance of the LAPACK code for packed format that relies on Level-2 BLAS. Our conclusions are drawn in Section 8.

2. LOWER PACKED FORMATS

The form of packed storage used by LINPACK [Dongarra et al. 1979] is that the upper-triangular part is held by columns. The Level-2 BLAS [Dongarra et al. 1988] and LAPACK [Anderson et al. 1999] permit the lower-triangular or upper-triangular part to be held by columns. We refer to these as the lower and upper packed formats, respectively. We show an example of the lower packed format in Figure 1a, with blocks of size 3 superimposed. In this and all the other figures that illustrate formats, we show where each matrix element is stored within the array that holds it.

It is apparent that the blocks are not suitable for passing to the BLAS since the stride between elements of a row is not uniform. We therefore propose to rearrange each trapezoidal block column so that it is stored by blocks with each block in row-major order, as illustrated in Figure 1b. Unless the order is an integer multiple of the block size, the final block will be shorter than the rest. Later in this section, we will explore alternative formats. We assume that the block size is chosen so that a block fits comfortably in level-1 cache. We defer discussion of the upper packed format to Section 5.

If the matrix order is n and the block size is nb, this rearrangement may be performed efficiently in place with the aid of a buffer of size $n \times nb$. For each block column, we copy the data to the buffer, reordering it so that it is held by rows in packed format, then copy the data back, overwriting the original block column. To reduce cache misses, the blocks are copied one by one, each copy being completed before the next is commenced. Within each block, we access the columns one by one and copy each to its new position. Little cache memory will be needed for the original columns and the whole block in the buffer should remain in the cache until its last column has been formed. On some machines, there is an efficiency gain

2a. Each block by columns

0									
1	3								
2	4	5							
6	9	12	27						
7	10	13	28	30					
8	11	14	29	31	32				
15	18	21	33	36	39	45			
16	19	22	34	37	40	46	48		
17	20	23	35	38	41	47	49	50	
24	25	26	42	43	44	51	52	53	54

2b. Rect. part of each block col. by cols

0									
1	3								
2	4	5							
6	13	20	27						
7	14	21	28	30					
8	15	22	29	31	32				
9	16	23	33	37	41	45			
10	17	24	34	38	42	46	48		
11	18	25	35	39	43	47	49	50	
12	19	26	36	40	44	51	52	53	54

Fig. 2. Alternative Lower Blocked Formats.

from the use of the Level-1 BLAS _COPY [Lawson et al. 1979], which copies vectors with uniform strides and uses loop unrolling. The subroutine _COPY may be used directly for copying a column of a rectangular block to the buffer (with stride nb in the buffer) and for copying the whole rearranged block column back from the buffer. We may use _COPY for the triangular block, too, by treating it separately; we expand the triangle to full form in the buffer and copy it back row by row; a simple copy may still be used for the rest of the block column.

It would have been easier to perform this rearrangement had we held each block by columns, as illustrated in Figure 2a, or the whole rectangular part of each block column by columns, as illustrated in Figure 2b and used by Duff and Reid [1996]. We explain in the next section why we expect that holding the blocks by columns will be less efficient.

We use the same block form for holding the Cholesky factor, which of course means that new codes are needed for forward and back substitution. We could have rearranged the matrix back to the packed format used by LINPACK and LAPACK, but the blocking is also helpful during forward and back substitution, particularly for multiple right-hand sides where the blocking allows the use of Level-3 BLAS.

3. BLOCK CHOLESKY FACTORIZATION

The Cholesky factorization of a symmetric and positive-definite matrix A usually takes the form

$$A = LL^T \tag{2}$$

where L is a lower-triangular matrix. If the same partitioning is applied to the rows and columns of A and the rows and columns of L, the resulting blocks A_{ij} and L_{ij}, with i and j in the range 1 to l, satisfy the relation

$$A_{ij} = \sum_{k=1}^{j} (L_{ik} L_{jk}^T) \ , \quad i \geq j \tag{3}$$

which allows the diagonal blocks L_{jj} of L to be found from the equation

$$L_{jj} L_{jj}^T = A_{jj} - \sum_{k=1}^{j-1} (L_{jk} L_{jk}^T) \tag{4}$$

```
do j = 1, l                                  ! l = ⌈n/nb⌉
    do k = 1, j − 1
        A_jj = A_jj − L_jk L_jk^T            ! Call of Level-3 BLAS _SYRK
        do i = j + 1, l
            A_ij = A_ij − L_ik L_jk^T        ! Call of Level-3 BLAS _GEMM
        end do
    end do
    L_jj L_jj^T = A_jj                       ! Call of LAPACK subroutine _POTRF
    do i = j + 1, l
        L_ij L_jj^T = A_ij                   ! Call of Level-3 BLAS _TRSM
    end do
end do
```

Fig. 3. LL^T Implementation for Lower Blocked Hybrid Format. The BLAS calls take the forms `_SYRK('U','T',...)`, `_GEMM('T','N',...)`, `_POTRF('U',...)`, and `_TRSM('L','U','T',...)`.

and the off-diagonal blocks L_{ij} $(i > j)$ to be found from the equation

$$L_{ij} L_{jj}^T = A_{ij} - \sum_{k=1}^{j-1} (L_{ik} L_{jk}^T). \tag{5}$$

We will assume that the first $l-1$ blocks are of size nb and that the final block is of size at most nb, so that $l = \lceil n/nb \rceil$.[1]

We have some choice over the order in which these operations are performed and how they are grouped. For the lower blocked hybrid form, it is natural to calculate the blocks column by column as shown in Figure 3. Each of the computation lines in the figure can be implemented by a single call of a Level-3 BLAS or LAPACK subroutine and we show which as a comment. However, it may be better to make a direct call to an equivalent 'kernel' routine that is fast because it has been specially written for matrices that are held in contiguous memory and are of a form and size that permits efficient use of the level-1 cache. We propose a recursive full storage format Cholesky subroutine in Section 6 for situations where a fast kernel is not available from the vendor or elsewhere.

For large problems, the most significant computation is that of `_GEMM`, which is performed on matrices held in contigous memory and is actually called for the transpose of the equation in Figure 3, that is,

$$A_{ij}^T = A_{ij}^T - L_{jk} L_{ik}^T.$$

Note that L_{jk} is held by rows and A_{ij}^T and L_{ik}^T are held by columns. The operation can therefore be applied efficiently with streaming, as explained in Section 1, without any data rearrangement. This is why we have not chosen either of the alternative formats of Figure 2. Similarly, the format of Figure 1b is favourable for the call of `_SYRK`; in fact, L_{jk} should remain in cache for the call of `_SYRK` and all the calls of `_GEMM` in the loop that follows. Similarly, the array that holds first A_{jj} and then L_{jj} will remain resident in the cache during the factorization of A_{jj} by `_POTRF` and throughout the `_TRSM` call, during which the rows are streamed into the cache as A_{ij} and out of the cache as L_{ij}.

[1]The notation $\lceil x \rceil$ refers to the least integer $i \geq x$, that is, the ceiling function.

```
do j = 1, l                                   ! l = ⌈n/nb⌉
    do k = 1, j − 1
        A_jj = A_jj − L_jk L_jk^T             ! Call of Level-3 BLAS _SYRK
        A_ij = A_ij − L_ik L_jk^T, ∀i > j     ! Single call of Level-3 BLAS _GEMM
    end do
    L_jj L_jj^T = A_jj                        ! Call of LAPACK subroutine _POTRF
    L_ij L_jj^T = A_ij, ∀i > j                ! Single call of Level-3 BLAS _TRSM
end do
```

Fig. 4. LL^T Implementation for Lower Blocked Hybrid Format with BLAS Called Once for Each Block Column. The BLAS calls take the forms _SYRK('U','T',...), _GEMM('T','N',...), _POTRF('U',...), and _TRSM('L','U','T',...).

The fact that our blocks are held contiguously is significant. Without this (see, for example, Figure 2b) either the cache is used less efficiently with unneeded data being brought in, or a preliminary rearrangement is needed.

For the operations with A_{jj}, use of the subroutines _SYRK and _POTRF requires that a temporary full-format copy of A_{jj} be made at the beginning of the main loop. _POTRF overwrites this by L_{jj}, which is used by _TRSM and then packed to overwrite A_{jj}. A buffer of size $nb \times nb$ is needed for the full-format copy.

A further merit of the lower blocked hybrid format (Figure 1b) is that all the operations

$$A_{ij} = A_{ij} - L_{ik} L_{jk}^T, \forall i > j$$

may be performed in a single call of _GEMM that involves multiplying a matrix of order $n - j \times nb$ by nb by a matrix of order nb by nb. This is possible since the whole trapezoidal block column is held by rows, which means that all the rectangular blocks of the block column can be passed as a single matrix to _GEMM. Similarly, the equation

$$L_{ij} L_{jj}^T = A_{ij}, \forall i > j$$

may be solved with a single call of _TRSM for a matrix of order $n - j \times nb$ by nb. We summarize the resulting code in Figure 4. The data format allows both _GEMM and _TRSM to perform their operations with streaming and without data copying, but we have no guarantee that library versions that are written for more general situations will do this.

The LAPACK subroutine _POTRF performs block Cholesky factorization of a symmetric positive-definite matrix held in full format (see LAPACK Users' Guide [Anderson et al. 1999, pages 29 and 295]). We show in Figure 5 how _POTRF is organized. The subroutine _POTF2 is an unblocked version of _POTRF. Note that Figures 4 and 5 are very similar. The difference is that the inner do loop of Figure 4 has been replaced by single calls of _SYRK and _GEMM, possible with the full format since any off-diagonal submatrix can be passed directly to a Level-3 BLAS as a rectangular matrix. For a large problem, most of the work is done by _GEMM and passing it a bigger problem gives it more scope for optimization. However, there is the disadvantage that the matrix that is sent to it is not in contiguous memory. It is our belief that most implementations of Level-3 BLAS begin with a data copy for each block of each operand to put it in contiguous memory and for each block of the

```
do j = 1, l                                          ! l = ⌈n/nb⌉
   A_jj = A_jj − Σ_{k=1}^{j−1} (L_jk L_jk^T)         ! Single call of Level-3 BLAS _SYRK
   L_jj L_jj^T = A_jj                                ! Call of LAPACK subroutine _POTF2
   A_ij = A_ij − Σ_{k=1}^{j−1} (L_ik L_jk^T), ∀i > j ! Single call of Level-3 BLAS _GEMM
   L_ij L_jj^T = A_ij, ∀i > j                        ! Single call of Level-3 BLAS _TRSM
end do
```

Fig. 5. LAPACK Cholesky Implementation for Lower Full Format (_POTRF). The BLAS calls take the forms `_SYRK('L','N',...)`, `_POTF2('L',...)`, `_GEMM('N','T',...)`, and `_TRSM('R','L','T',...)`.

result to return it to its original format. For this reason, we expect that the code for the blocked hybrid format will usually be faster (as well as needing about half the memory).

4. SOLVING EQUATIONS USING A BLOCK CHOLESKY FACTORIZATION

Given the Cholesky factorization (2), we may solve the equations

$$AX = B \tag{6}$$

by forward substitution

$$LY = B \tag{7}$$

followed by back-substitution

$$L^T X = Y. \tag{8}$$

If the rows of B, Y, and X are partitioned in the same way as A and L, equations (7) and (8) take the form

$$L_{ii} Y_i = B_i - \sum_{j=1}^{i-1} (L_{ij} Y_j),\ i = 1, 2, \ldots, l \tag{9}$$

and

$$L_{jj}^T X_j = Y_j - \sum_{i=j+1}^{l} (L_{ij}^T X_i),\ j = l, l-1, \ldots, 1, \tag{10}$$

where $l = \lceil n/nb \rceil$.

We begin by discussing the important special case where B has only one column. This, of course, means that B_i, Y_i, and X_i are all blocks that have a single column. If L is held in lower blocked hybrid format, the forward substitution may be performed by subtracting $L_{ij} Y_j$ from B_i as soon as Y_j is available using a single call of `_GEMV`, as shown in the first part of Figure 6. The back-substitution can be performed with a single call of `_GEMV` for each summation, as shown in the second part of Figure 6. The whole of L must be accessed once during forward substitution and once during back-substitution and in both cases the whole of the rectangular part of each block column is accessed in a single call of `_GEMV`.

Similar code could be applied in the case where B has $m > 1$ columns, with `_TRSM` replacing `_TPSV` (this will require making a copy of L_{jj} in full storage) and `_GEMM`

```
do j = 1, l
    L_jj Y_j = B_j                  ! Call of Level-2 BLAS _TPSV('U','T','N',...)
    B_i = B_i − L_ij Y_j, ∀i > j    ! Single call of Level-2 BLAS _GEMV('T',...)
end do
do j = l, 1, −1
    Y_j = Y_j − Σ_{i>j}(L^T_ij X_i) ! Single call of Level-2 BLAS _GEMV('N',...)
    L^T_jj X_j = Y_j                ! Call of Level-2 BLAS _TPSV('U','N','N',...)
end do
```

Fig. 6. Forward substitution and back-substitution for a single right-hand side using Lower Blocked Hybrid Factorization LL^T.

0	3	6	9
1	4	7	10
2	5	8	11
12	15	18	21
13	16	19	22
14	17	20	23
24	27	30	33
25	28	31	34
26	29	32	35
36	37	38	39

Fig. 7. The blocked format for B.

replacing _GEMV. However, the blocks B_i, Y_i, and X_i would not occupy contiguous memory, so there is scope for improving the performance.

If the number of columns m is modest, we therefore make a copy of B as a block matrix, with each block B_i held contiguously by columns and the blocks held contiguously, as illustrated in Figure 7.

Code for forward substitution and back-substitution is shown in Figure 8. Each of the operations $L_{ij}Y_j$ and $L_{ij}^T X_i$ is performed by a separate call of _GEMM, but each submatrix involved is held contiguously and streaming is available. Finally, X is copied back to overwrite the original B.

The overhead of copying all the diagonal blocks L_{jj} to full storage may be halved

```
do j = 1, l
    L_jj Y_j = B_j            ! Call of _TRSM('L','U','T',...)
    do i = j+1, l
        B_i = B_i − L_ij Y_j  ! Call of _GEMM('T','N',...)
    end do
end do
do i = l, 1, −1
    L^T_ii X_i = Y_i          ! Call of _TRSM('L','U','N',...)
    do j = 1, i − 1
        Y_j = Y_j − L^T_ij X_i ! Call of _GEMM('N','N',...)
    end do
end do
```

Fig. 8. Forward substitution and back-substitution for many right-hand sides using Lower Blocked Hybrid Factorization LL^T.

if we have a buffer that is big enough to hold them all, so that they are already available for the back-substitution. The size needs to be $n \times nb$, which is the same as we used in Section 2. Of course, this means that we need two buffers, one of size $n \times nb$ for the diagonal blocks and one of size $n \times m$ for the right-hand sides.

For very small numbers of columns, say 2 or 3, the copying overhead probably means that it is better to handle each column separately, as in Figure 6.

For very large numbers of columns, we simply apply the same process of forward substitution and back-substitution (Figure 8) to successive blocks of columns. In our code, we allow for a different block size mb. Buffers of total size $n \times (nb{+}mb)$ are needed. Note that with two buffers, the overhead of copying the diagonal blocks is incurred only once for all the forward substitutions and back-substitutions on all the block columns.

For very big problems, the block size mb is probably best chosen to be similar to nb for two reasons. The first is simply that making it much bigger would substantially increase the total buffer size. The second is that for a very big problem the factorized matrix has to be moved from memory or level-3 cache for each block of mb columns, a total data movement of about $n^2\frac{m}{mb}$ reals. Meanwhile, if $n{\times}mb$ is too big for level-2 cache, the data movement for the right-hand side averages at about $\frac{n}{2}{\times}mb$ reals for each block step of the forward or back substitution to give a total of about $(n{\times}mb){\times}\frac{n}{nb}{\times}\frac{m}{mb} = n^2\frac{m}{nb}$ reals. It follows that the data movement is balanced with $mb = nb$ and will be increased if either is mb or nb is increased at the expense of the other.

If the problem is big but not such that the level-2 cache cannot hold $n{\times}nb$ reals, there are likely to be performance advantages in chosing the block size mb to be larger than nb (or even equal to m). This is because each of the blocks of the factorized matrix is accessed once for each block column of B, so increasing mb reduces level-2 cache movement provided there is still room in level-2 cache for $n{\times}mb$ reals.

9a. Upper Packed Format

0	1	3	6	10	15	21	28	36	45
	2	4	7	11	16	22	29	37	46
		5	8	12	17	23	30	38	47
			9	13	18	24	31	39	48
				14	19	25	32	40	49
					20	26	33	41	50
						27	34	42	51
							35	43	52
								44	53
									54

9b. Upper Blocked Hybrid Format

0	1	3	6	9	12	21	24	27	45
	2	4	7	10	13	22	25	28	46
		5	8	11	14	23	26	29	47
			15	16	18	30	33	36	48
				17	19	31	34	37	49
					20	32	35	38	50
						39	40	42	51
							41	43	52
								44	53
									54

Fig. 9. Upper Packed and Blocked Hybrid Formats.

5. UPPER PACKED FORMATS

We will now consider the Cholesky factorization of a matrix in upper packed format (see Figure 9a). Our chosen blocked hybrid format is illustrated in Figure 9b. It is ordered by block columns, which allows us to get the same desirable properties

```
do i = 1, l                               ! l = ⌈n/nb⌉
    do k = 1, i − 1
        A_ii = A_ii − U_ki^T U_ki         ! Call of Level-3 BLAS _SYRK
        do j = i + 1, l
            A_ij = A_ij − U_ki^T U_kj     ! Call of Level-3 BLAS _GEMM
        end do
    end do
    U_ii^T U_ii = A_ii                    ! Call of LAPACK subroutine _POTRF
    do j = i + 1, l
        U_ii^T U_ij = A_ij                ! Call of Level-3 BLAS _TRSM
    end do
end do
```

Fig. 10. U^TU Cholesky Implementation for Upper Blocked Hybrid Format. The BLAS calls take the forms `_SYRK('U','T',...)`, `_GEMM('T','N',...)`, `_POTRF('U',...)`, and `_TRSM('L','U','T',...)`.

```
do i = 1, l                                          ! l = ⌈n/nb⌉
    A_ii = A_ii − Σ_{k=1}^{i−1} (U_ki^T U_ki)        ! Call of Level-3 BLAS _SYRK
    U_ii^T U_ii = A_ii                               ! Call of LAPACK subroutine _POTF2
    A_ij = A_ij − Σ_{k=1}^{i−1} (U_ki^T U_kj), ∀j > i ! Single call of Level-3 BLAS _GEMM
    U_ii^T U_ij = A_ij, ∀j > i                       ! Single call of Level-3 BLAS _TRSM
end do
```

Fig. 11. LAPACK Cholesky Implementation for Upper Full Format. The BLAS calls take the forms `_SYRK('U','T',...)`, `_POTF2('U',...)`, `_GEMM('T','N',...)`, and `_TRSM('L','U','T',...)`.

for the rearrangement that we had for the lower packed format (Section 2). The individual blocks are ordered by columns to permit efficient calls of Level-3 BLAS. A consequence is that the rearrangement code runs a little faster than for the lower packed hybrid format since the entries of each column of each block are contiguous before and after rearrangement.

LAPACK returns the matrix U of the U^TU factorization, so we first consider how this may be done with a blocked hybrid format. We then consider the alternative of using a backwards pivot sequence, that is, performing a UU^T factorization. From the error analysis point of view, this is equally satisfactory; both are unconditionally stable for a symmetric positive-definite matrix. The UU^T factorization has performance advantages, but is unsuitable if compatibility with LAPACK is needed.

The algorithm we have chosen may be obtained by transposing every matrix in Figure 3 and interchanging i with j to obtain Figure 10. The outer loop is over block rows and within it there is a block outer product. Note also the similarity with the way the LAPACK subroutine `_POTRF` performs a block U^TU factorization of a matrix held in upper full format, see Figure 11.

If we use the upper blocked hybrid format of Figure 9b, we now get all the same desirable properties for level-1 cache usage, but we cannot combine the `_GEMM` calls in the inner loop into one call since the blocks are not held contiguously. Note, however, that the same block U_{ki}^T is used for each call in the inner loop, so this will remain resident in level-1 cache for all the calls in the inner loop. Similarly, the _TRSM calls cannot be merged, but the block U_{ii}^T will remain resident in level-1 cache.

We anticipate that the effect of the blocks not being contiguous in memory will be quite minor and this is borne out by the results in Section 7, where we see that the code for the upper packed hybrid format is only sightly slower than that for the lower packed hybrid format. There are more procedure calls, but each does a significant amount of work, so the call itself is a small overhead. There will also be some unnecessary data movement to and from the cache at the edges of the blocks, but this will be small compared with that needed for the blocks themselves. Finally, if the level-2 cache is not big enough for the whole matrix but is big enough (with a reasonable margin) for a block row, the pivot block row will be resident in level-2 cache for all the operations associated with the block pivot even though the blocks are not contiguous.

To get contiguous blocks, we considered rearranging the blocks so that they are held by block rows, see Figure 12a. Actually, this rearranged form is also the lower blocked hybrid format representation of the matrix (see Figure 1b), since any ordering of the upper triangular entries a_{ij}, $i \le j$, of a symmetric matrix A is also an ordering of its lower triangular entries a_{ji}, $i \le j$. Hence, if we did this rearrangement, we could apply a code implementing the algorithm of Figure 3 and thereby get all the desirable properties of that algorithm. However, we rejected this approach because of the difficulty of constructing an efficient in-place code for rearrangement. The best rearrangement algorithm that we found requires an additional integer array of length the total number of blocks and we estimate that it would run about twice as slowly as our present rearrangement code.

12a. The Blocks Ordered by Rows

0	1	3	6	9	12	15	18	21	24
	2	4	7	10	13	16	19	22	25
		5	8	11	14	17	20	23	26
			27	28	30	33	36	39	42
				29	31	34	37	40	43
					32	35	38	41	44
						45	46	48	51
							47	49	52
								50	53
									54

12b. Each Block Ordered by Rows

0	1	2	6	7	8	21	22	23	45
	3	4	9	10	11	24	25	26	46
		5	12	13	14	27	28	29	47
			15	16	17	30	31	32	48
				18	19	33	34	35	49
					20	36	37	38	50
						39	40	41	51
							42	43	52
								44	53
									54

Fig. 12. Alternative Upper Blocked Hybrid Formats.

For solving sets of equations, similar considerations apply for the upper blocked hybrid format to those discussed in Section 4. The code for many right-hand sides is illustrated in Figure 13. The code for a single right-hand side is very similar, again with _GEMM replaced by _GEMV and _TRSM replaced by _TPSV. Note that we are not able to combine calls of _GEMV with this data format.

If it is acceptable to reverse the pivot order, that is perform a UU^T factorization

$$A = UU^T \tag{11}$$

where U is upper triangular, we can get the desirable properties without an additional rearrangement. Now we have the relation

```
do j = 1, l
      do i = 1, j − 1
            B_j = B_j − U_ij^T Y_i      ! Call of _GEMM('T','N',...)
      end do
      U_jj^T Y_j = B_j                  ! Call of _TRSM('L','U','T',...)
end do
do j = l, 1, −1
      U_jj X_j = Y_j                    ! Call of _TRSM('L','U','N',...)
      do i = 1, j − 1
            Y_i = Y_i − U_ij X_j        ! Call of _GEMM('N','N',...)
      end do
end do
```

Fig. 13. Forward substitution and back-substitution for many right-hand sides using Upper Blocked Hybrid Factorization $U^T U$.

$$A_{ij} = \sum_{k=j}^{l} (U_{ik} U_{jk}^T) \; , \quad i \leq j \tag{12}$$

which allows the diagonal blocks U_{jj} to be found from the equation

$$U_{jj} U_{jj}^T = A_{jj} - \sum_{k=j+1}^{l} (U_{jk} U_{jk}^T) \tag{13}$$

and the off-diagonal blocks U_{ij} $(i < j)$ to be found from the equation

$$U_{ij} U_{jj}^T = A_{ij} - \sum_{k=j+1}^{l} (U_{ik} U_{jk}^T). \tag{14}$$

We now need to loop through the block columns in reverse order, as illustrated in Figure 14. Again, each of the computation lines in the figure can be implemented by a single call of a Level-3 BLAS or LAPACK subroutine or an equivalent kernel routine and we show which as a comment. If each block is held by rows, see Figure 12b, we get all the desirable properties of the Figure 3 algorithm for level-1 cache usage and we can merge the BLAS calls in the inner loops into single calls. There is the disadvange that we will need to permute the diagonal block A_{jj} before passing it to _POTRF and permute the factor when packing it back.

For the use of a UU^T factorization to solve a set of equations, similar considerations apply for the upper blocked hybrid format to those discussed in Section 4. Here, we need to perform a back-substitution followed by a forward substitution, see Figure 15.

The disadvantage of having to permute the diagonal block may be avoided by reversing the order within each block and of the blocks within each block column. We do not reverse the order of the block columns themselves since this does not impact the efficiency of the algorithm and allows the rearrangement to blocked hybrid format to be performed independently for each block column. The format is illustrated in Figure 16a.

$$
\begin{array}{ll}
\text{do } j = l, 1, -1 & !\ l = \lceil n/nb \rceil \\
\quad \text{do } k = j+1, l & \\
\quad\quad A_{jj} = A_{jj} - U_{jk} U_{jk}^T & !\ \text{Call of Level-3 BLAS \_SYRK} \\
\quad\quad \text{do } i = 1, j-1 & \\
\quad\quad\quad A_{ij} = A_{ij} - U_{ik} U_{jk}^T & !\ \text{Call of Level-3 BLAS \_GEMM} \\
\quad\quad \text{end do} & \\
\quad \text{end do} & \\
\quad U_{jj} U_{jj}^T = A_{jj} & !\ \text{Call of LAPACK subroutine \_POTRF} \\
\quad \text{do } i = 1, j-1 & \\
\quad\quad U_{ij} U_{jj}^T = A_{ij} & !\ \text{Call of Level-3 BLAS \_TRSM} \\
\quad \text{end do} & \\
\text{end do} &
\end{array}
$$

Fig. 14. UU^T Implementation for Upper Blocked Hybrid Format. With the format of Figure 12b, the BLAS calls take the forms `_SYRK('L','T',...)`, `_GEMM('T','N',...)`, `_POTRF('L',...)`, and `_TRSM('L','L','T',...)`.

$$
\begin{array}{ll}
\text{do } j = l,\ 1,\ -1 & \\
\quad U_{jj} Y_j = B_j & !\ \text{Call of \_TRSM('L','L','T',...)} \\
\quad B_i = B_i - U_{ij} Y_j, \forall i < j & !\ \text{Call of \_GEMM('T','N',...)} \\
\text{end do} & \\
\text{do } j = 1,\ l & \\
\quad Y_j = Y_j - \sum_{i<j} (U_{ij}^T X_i) & !\ \text{Call of \_GEMM('N','N',...)} \\
\quad U_{jj}^T X_j = Y_j & !\ \text{Call of \_TRSM('L','L','N',...)} \\
\text{end do} &
\end{array}
$$

Fig. 15. Back-substitution and forward substitution for many right-hand sides using Upper Blocked Hybrid Factorization UU^T with the format of Figure 12b.

16a. The Ordering

0	9	8	7	27	26	25	54	53	52
	6	5	4	24	23	22	51	50	49
		3	2	21	20	19	48	47	46
			1	18	17	16	45	44	43
				15	14	13	42	41	40
					12	11	39	38	37
						10	36	35	34
							33	32	31
								30	29
									28

16b. When Rotated by 180 degrees

28									
29	30								
31	32	33							
34	35	36	10						
37	38	39	11	12					
40	41	42	13	14	15				
43	44	45	16	17	18	1			
46	47	48	19	20	21	2	3		
49	50	51	22	23	24	4	5	6	
52	53	54	25	26	27	7	8	9	0

Fig. 16. Another Upper Blocked Hybrid Format, obtained by reversing the order within each block and of the blocks within each block column.

Figure 16b depicts Figure 16a rotated by 180 degrees and shows how it represents the matrix PAP^T for P the permutation that reverses the order. Figure 16b is closely related to Figure 1b. The only difference is that the order of the trapezoidal block columns is reversed. If the offsets -28, 17, 44, 54 are applied to all locations in block columns 1, 2, 3, and 4, respectively, of Figure 16b, we obtain Figure 1b. This means that we can apply the algorithm depicted in Figure 3 to PAP^T by adding the appropriate offset to all the addresses in each block column. And for forward and back substitution, the code in Figure 8 will work by adding the appropriate offset to all addresses in each block column.

6. LEVEL-3 CHOLESKY KERNEL SUBROUTINES

For each of our block factorizations (see Figures 3, 4, 10, and 14), we have employed the LAPACK subroutine `_POTRF` to factorize the diagonal blocks, held in upper full format. This is itself a block algorithm and is summarized in Figure 11. We expect its block size to be about the same as ours. This means that a significant proportion of its computation, or perhaps all of it, is performed by `_POTF2`. This is unsatisfactory since it uses Level-2 BLAS. The purpose of this section is to consider alternatives that use Level-3 BLAS.

In this section, we still use the notation A for the matrix, n for its order, and LL^T and U^TU for its Cholesky factorizations, but these refer to a diagonal block of the overall computation. This should be no confusion since the discussion is confined to factorizing a block. If should be borne in mind that n will be modest, small enough for the computation to reside in level-1 cache.

The subroutine `_POTRF` uses full storage mode, that is, it requires n^2 memory locations even though it only access $n(n+1)/2$ matrix elements. The rest of the locations are not used. We have chosen to follow this for our alternatives because all our algorithms rely on `_SYRK` and `_TRSM` and because it allows Level-3 BLAS to be used on submatrices without any further data movement.

Recently, four papers focusing on recursive Cholesky algorithms were published, [Gustavson 1997], [Waśniewski et al. 1998], [Gustavson and Jonsson 2000], and [Andersen et al. 2001]. These papers demonstrate that full storage data format recursive Cholesky algorithms can be developed using Level-3 BLAS for almost all the computation. The appendices of the first and fourth papers contain Fortran 77 and Fortran 90 implementations of the algorithms.

For the upper packed format, these recursive algorithms rely on partitioning the matrices A and U into submatrices:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{12}^T & A_{22} \end{pmatrix} \quad \text{and} \quad U = \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix},$$

where A_{11} and U_{11} are $p \times p$ and $p = \lfloor n/2 \rfloor$. Of course, only the upper-triangular parts of A_{11} and A_{22} are stored. The matrices U_{11} and U_{22} are upper triangular and the matrices A_{12} and U_{12} are square or nearly square (of size $p \times n{-}p$).

We arrive at a recursive algorithm by simple algebraic manipulations on the partitioning indicated above, see Figure 17. Note that the only floating-point operations that are performed outside calls to `_TRSM` and `_SYRK` are the calculation of n square roots.

In Figure 17, the recursion continues all the way down to the single diagonal

if $n > 1$ then	
$U_{11}^T U_{11} = A_{11}$	Cholesky factor A_{11} recursively
$U_{12}^T U_{11} = A_{12}$	_TRSM (triangular multiple solve)
$\hat{A}_{22} = A_{22} - U_{12}^T U_{12}$	_SYRK (symmetric rank-k update)
$U_{22}^T U_{22} = \hat{A}_{22}$	Cholesky factor $\hat{A}_{22}$ recursively
else	
$U = \sqrt{A}$	A and U are 1 by 1

Fig. 17. Recursive Cholesky Kernel.

do $i = 1, l$	! $l = \lceil n/kb \rceil$
$A_{ii} = A_{ii} - \sum_{k=1}^{i-1}(U_{ki}^T U_{ki})$	! Like Level-3 BLAS _SYRK
$U_{ii}^T U_{ii} = A_{ii}$	! Cholesky factorization of block
do $j = i + 1, n$	
$A_{ij} = A_{ij} - \sum_{k=1}^{i-1}(U_{ki}^T U_{kj})$	! Like Level-3 BLAS _GEMM
$U_{ii}^T U_{ij} = A_{ij}$	! Like Level-3 BLAS _TRSM
end do	
end do	

Fig. 18. Cholesky Kernel Implementation for Upper Full Format.

elements. During the factorization of very small matrices close to the end of the recursion, there are very few floating-point operations in comparison to the number of subroutine calls, administrative, and fixed-point calculations in the recursively called subroutine. This makes the factorization relatively inefficient. A way to alleviate this inefficiency is to inline the code for these small matrices; for instance, for matrices of size up to 4×4. The first line of Figure 17 must be replaced by

$$\text{if } n > 4 \text{ then}$$

and the last line by the Cholesky factorization of A, using special unrolled code.

By choosing the recursive division of the coefficient matrix differently, it is possible to make sure that the final factorization at the leaves of the recursion is a 4×4 matrix in all cases except the final leaf. This is done by choosing p always to be a multiple of 4. It means that a special inlined code can be used for the 4×4 case. Of course, when n is not a multiple of 4, there must be some code that handles a single block that is smaller than 4×4.

Another possibility is to use a block algorithm with a very small block size kb, designed to fit in registers. To avoid procedure call overheads for a very small computations, we replace all calls to BLAS by in-line code. This means that it is not advantageous to perform a whole block row of _GEMM updates at once and a whole block row of _TRSM updates at once (see last two lines of the loop in Figure 11) . This leads to the algorithm summarized in Figure 18.

We have found the block size $kp = 2$ to be suitable. The key loop is the one that corresponds to _GEMM. For this, the code of Figure 19 is suitable. The block $A_{i,j}$ is held in the four variables, `t11`, `t12`, `t21`, and `t22`. We reference the underlying array directly, with $A_{i,j}$ held from `a(ii,jj)`. It may be seen that a total of 8 local variables are involved, which hopefully the compiler will arrange to be held in registers. The loop involves 4 memory accesses and 8 floating-point operations.

On some processors, faster execution is possible by having an inner _GEMM loop

```
DO k = 1, ii - 1
   aki = a(k,ii)
   akj = a(k,jj)
   t11 = t11 - aki*akj
   aki1 = a(k,ii+1)
   t21 = t21 - aki1*akj
   akj1 = a(k,jj+1)
   t12 = t12 - aki*akj1
   t22 = t22 - aki1*akj1
END DO
```

Fig. 19. Code corresponding to _GEMM.

Table 1. Computers Used. The IBM level-2 and level-3 caches are shared between two processors.

Processor	MHz	Peak Mflops	Cache sizes level-1	level-2	level-3	TLB entries
IBM Power4	1700	6800	64K	1.5M*	32M*	1024
SUN UltraSparc IIICu	900	1800	64K	8M	None	512
SGI MIPS R12000	300	600	32K	8M	None	64
HP Alpha EV6	500	1000	64K	4M	None	128
HP Itanium 2	1000	4000	32K	256K	1.5M	128
INTEL Pentium III	500	500	16K	512K	None	32

*Shared with another processor.

that updates $A_{i,j}$ and $A_{i,j+1}$. The variables `aki` and `aki1` need only be loaded once, so we now have 6 memory accesses and 16 floating-point operations and need 14 local variables, hopefully in registers. We found that this algorithm gave very good performance (see next section).

We will make our implementation of this kernel available in a companion paper, but alternatives should be considered. A version of a recursive Cholesky algorithm was programmed and added to the ATLAS library by [Whaley et al. 2000]. The LAPACK library can be updated to use this new recursive subroutine by using a simple script from the ATLAS subdirectory (`ATLAS/doc/LAPACK.txt`; the ATLAS library can be found on `http://www.netlib.org/atlas/`). Further, every computer hardware vendor is interested in having good and well-tuned software libraries.

We recommend that all the alternatives of the previous paragragh be compared. Our kernel routine is available if the user is not able to perform such a comparison procedure or has no time to do it. Finally, note that LAPACK, ATLAS and the development of computer vendor software are ongoing activities. The implementation that is the slowest today might be the fastest tomorrow.

7. PERFORMANCE

For our performance testing, we have used the computers listed in Table 1 with the compilers listed in Table 2 and the libraries listed in Table 3.

In normal running, the shared level-2 cache on the IBM resulted in speeds that varied by up to about 30% according to the activity of the sharing processor. This made it impossible to conduct detailed comparisons between the algorithms. We therefore ran on a quiet machine, which means that the speeds we report are

Table 2. Compilers Used.

Processor	Compiler	Option
IBM Power4	F95, version 8.1	`-O5`
SUN UltraSparc IIICu	F95, Forte 7.1	`-fast -xarch=v8plusb -xchip=ultra3 -free`
SGI MIPS R12000	F95, version, 7.3	`-O3 -64 -freeform`
HP Alpha EV6	F95, version 5.3-915	`-free -O`
HP Itanium 2	F95, version v2.7	`-O3`
Intel Pentium III	NAG F95, 4.1(340)	`-free -O4`

Table 3. Computer Libraries Used.

Processor	Library
IBM Power4	ESSL, version 3.3; LAPACK routines for `_PPTRF` and `_PPTRS`
SUN UltraSparc IIICu	Sun Performance Library 4.0
SGI MIPS R12000	SGI Scientific Library, 7.3.1.2
HP Alpha EV6	CXML Extended Math Library V3.6
HP Itanium 2	HP MLIB BLAS Library
Intel Pentium III	ATLAS, version 3.0

optimistic for normal runs.

7.1 Kernels

We begin by considering the alternatives for the Cholesky kernel subroutine (Section 6). We consider orders 40, 72, and 100 since these will typically allow the computation to fit comfortably in level-1 cache. For each computer, we have compared

—the optimized LAPACK implementation provided by the vendor, if available, or otherwise the ATLAS optimized code;

—the published LAPACK source code compiled with the highest level of optimization;

—the recursive code of Section 6;

—the mini-blocked code of the end of Section 6 with all blocks of size 2×2; and

—the mini-blocked code of the end of Section 6 with blocks of sizes 2×2 and 2×4.

It may be seen from the results in Table 4 that the mini-blocked code with blocks of sizes 2×2 and 2×4 is remarkably successful. In all six cases, it significantly outperforms the compiled LAPACK code and the recursive algorithm. It outperforms the vendor's optimized codes except on the IBM platform at order 100. If compared with the mini-blocked code with all blocks of size 2×2, the performance is significantly better on the SUN (about 20% times better except at order 72), slightly better on the Alpha (about 15% times better), and much the same on the other four.

For our remaining performance testing, we use the vendor's optimized kernel on the IBM and the mini-blocked code with blocks of sizes 2×2 and 2×4 on the others.

Table 4. Performance in Mflops of the Kernel Cholesky Algorithm. Comparison between different computers and different versions of subroutines.

Order	LAPACK Vendor	LAPACK Comp.	Recursive	Mini-block 2×2	Mini-block 2×2 & 2×4
IBM Power4, 1700 MHz, ESSL Library:					
40	1658	1503	707	1999	1999
72	2653	2303	1447	2751	2753
100	3037	2481	1930	2957	2945
SUN Ultra III, 900 MHz, Sunperf BLAS Library:					
40	392	427	251	803	941
72	598	664	417	1191	1012
100	619	830	589	1143	1506
SGI Origin 2000, R12000, 300 MHz, Math Library:					
40	117	121	94	369	372
72	197	212	166	455	466
100	238	289	217	485	496
Alpha EV6, 500 MHz, DXML Library:					
40	311	313	165	457	533
72	340	343	250	523	625
100	393	400	323	551	647
HP Itanium 2, 1000 MHz, HP MLIB BLAS Library:					
40	449	448	153	1125	1133
72	597	595	266	1711	1722
100	567	559	364	2103	2103
Intel Pentium III, 500 MHz, ATLAS BLAS Library:					
40	83	97	86	177	169
72	138	148	132	193	184
100	157	155	147	182	179

7.2 Factorization

The LAPACK subroutine _PPTRF performs Cholesky factorization of a symmetric positive-definite matrix held in unblocked packed format and the LAPACK subroutine _PPTRS solves corresponding sets of equations using the factorization (see LAPACK Users' Guide [Anderson et al. 1999, pages 29 and 301]). Equations (2) to (5) remain applicable if we take all the blocks to have size 1. For the upper packed format, the source-code implementation of _PPTRF uses the Level-2 BLAS _TPSV for each column of U to solve the triangular set of equations that determine its off-diagonal entries (see equation (5)). For the lower packed format, _PPTRF uses the Level-2 BLAS _SPR [Dongarra et al. 1988] to perform a rank-1 update of the remaining matrix. In both cases, the code is inefficient because the Level-2 BLAS involve an amount of data movement through the cache that is proportional to the number of arithmetic operations performed.

We have written Fortran 90 subroutines _HPPTF and _HPPTS with the same objectives for the blocked hybrid format as _PPTRF and _PPTRS have for the unblocked packed format and their argument lists are very similar. In addition, the subroutines _PPHPP and _HPPPP perform in-place rearrangements between the formats, using the ideas discussed in section 2. These subroutines are available in the companion algorithm [Andersen et al. 2004].

We have used the computers listed in Table 1 with the libraries listed in Table 3

Table 5. Mflops, Cholesky factorizations including rearrangement, different nb values, IBM Power4.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Lower Packed Hybrid											
nb=40	956	1444	2055	2679	3156	3626	3861	3933	3968	3960	4040
nb=72	957	1520	1972	2650	3100	3626	3971	4086	4162	4166	4292
nb=100	943	1494	2103	2741	3193	3715	3971	4119	4162	4117	4258
nb=200	947	1515	2116	2417	2895	3440	3786	4059	4213	4322	4500
Upper Packed Hybrid											
nb=40	1117	1425	1916	2523	2911	3404	3598	3655	3710	3720	3716
nb=72	1111	1743	2052	2590	3006	3489	3761	3947	4015	4084	4102
nb=100	1102	1732	2376	2831	3159	3678	3832	4033	4112	4150	4191
nb=200	1105	1732	2375	2584	2997	3447	3761	4033	4266	4340	4481

Table 6. Size of the packed matrix in megabytes.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Mbytes	0.006	0.016	0.039	0.098	0.239	0.612	1.6	3.8	9.8	23.9	61.1

to compare our blocked hybrid codes with the full-format codes in LAPACK and the recursive packed codes of Andersen, Gustavson, and Waśniewski [2001]. Note that the full-format code does most of its work in calls of `_GEMM` for subarrays that are not held contiguously in memory and the recursive packed code does most of its work in calls of `_GEMM` for arrays that are held contiguously but may be too large for level-1 cache. In both cases, the performance is therefore marred by `_GEMM` doing data copying before starting its actual computation and, possibly, upon completion of that computation.

We compiled the LAPACK source codes with the highest level of optimization available and called the vendor-supplied BLAS. We believe that this is what most users will do, but we also ran the vendor-supplied LAPACK codes or the ATLAS LAPACK codes on the Intel Pentium III where vendor codes were unavailable.

The speeds were obtained by running the code repeatedly until at least a second and a half had elapsed, which gives reasonably reliable figures, though some of the final digits shown in the tables varied when runs were repeated.

For the block size nb in the blocked hybrid codes, we have experimented with the values 40, 72, 100, and 200. The first three were used in Section 7.1. We added nb=200 to see what would happen with a larger value.

The speeds for factorization, including rearrangement to the packed hybrid format, on the IBM Power4 are shown in Table 5. Since a square matrix of order 72 occupies 41.5K bytes of memory, this is the biggest of our nb values to permit full advantage of streaming to be taken in calls of `_GEMM`. However, this value rarely gives the best speed. For small n, the kernel performance is of overriding importance. For $n \geq 640$, the packed matrix does not fit into level-2 cache, see Table 6, so a larger value of nb will reduce the movement from level-3 cache. Our conclusion is that nb=100 is suitable for this machine, but note that higher performance is available for large n with nb=200.

Similar performance figures for the SUN Ultra III are shown in Table 7. This has a larger level-1 cache, which permits full advantage of streaming to be taken

Table 7. Mflops, Cholesky factorizations including rearrangement, different nb values, SUN Ultra III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Lower Packed Hybrid											
nb=40	394	548	644	773	832	963	1046	1106	1110	949	842
nb=72	391	558	660	760	857	993	1107	1182	1215	1120	1045
nb=100	389	557	708	738	824	959	1095	1115	1137	1144	1083
nb=200	390	557	708	782	867	965	1080	1180	1201	1206	1254
Upper Packed Hybrid											
nb=40	526	608	680	804	830	943	1006	1040	1052	915	791
nb=72	520	769	767	838	916	1032	1104	1182	1168	1123	1004
nb=100	516	770	957	841	901	1026	1116	1177	1213	1145	1065
nb=200	518	768	954	988	983	1032	1144	1181	1240	1280	1243

Table 8. Mflops, Cholesky factorizations including rearrangement, different nb values, SGI Origin 2000.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Lower Packed Hybrid											
nb=40	197	199	242	292	339	384	420	445	454	451	447
nb=72	201	269	274	327	366	411	448	430	474	464	450
nb=100	193	269	330	339	382	427	461	484	476	461	439
nb=200	192	264	323	342	359	369	377	397	409	523	529
Upper Packed Hybrid											
nb=40	236	204	245	290	335	380	416	441	448	442	436
nb=72	237	312	294	331	367	411	447	442	456	459	449
nb=100	235	320	389	352	386	429	462	484	470	458	453
nb=200	237	316	384	381	380	386	383	404	406	488	528

in calls of _GEMM when nb=72. But, again, the kernel performance is of overriding importance for small n and level-2 cache is important for large n (the packed matrix does not fit in level-2 cache for $n \geq 1600$). Here, our conclusion is that nb=200 is suitable.

Tables 8 to 11 show the speed variation with nb on our other platforms and they show similar patterns. We conclude that suitable values for nb are 100 for the SGI Origin 2000, 200 for the Alpha EV6, 200 for the Itanium rx2600s and 40 for the Intel Pentium III.

Having chosen nb values, we now make comparisons with other algorithms and show the rearrangement overheads. In Table 12, we show factorization speeds on the IBM Power4 computer. The first two rows show the performance of the LAPACK code _PPTRF for the lower and upper packed formats when compiled with full optimization and calling the vendor-supplied BLAS. Their performance deteriorates markedly as n increases beyond 640. We believe that this is because they are using Level-2 BLAS. None of the other codes have this defect. It is clear that using Level-2 BLAS is not a good strategy. The ESSL library contains an equivalent code for the lower packed format and we show its speed in the third line of the table. Part of this code is based on the work of Gustavson and Jonsson [2000].

The next four lines show the speeds of comparable codes for the full format and

Table 9. Mflops, Cholesky factorizations including rearrangement, different nb values, Alpha EV6.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Lower Packed Hybrid											
nb=40	370	326	380	456	525	576	622	638	645	649	657
nb=72	368	473	396	450	511	582	642	677	694	713	726
nb=100	345	457	497	476	533	600	655	689	712	738	749
nb=200	343	455	491	537	525	591	648	695	731	764	789
Upper Packed Hybrid											
nb=40	432	347	388	465	518	588	626	645	660	652	655
nb=72	412	531	426	460	521	585	642	659	688	703	711
nb=100	431	525	563	494	538	600	655	689	718	733	742
nb=200	432	521	554	595	566	591	662	701	744	767	790

Table 10. Mflops, Cholesky factorizations including rearrangement, different nb values, Itanium rx2600s.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Lower Packed Hybrid											
nb=40	696	388	510	608	805	1094	1418	1694	1978	2225	2326
nb=72	700	1034	600	683	827	1109	1385	1687	2027	2244	2449
nb=100	668	1011	1701	763	869	1108	1419	1709	2017	2284	2483
nb=200	657	1017	1689	1731	1279	1384	1552	1829	2089	2216	2402
Upper Packed Hybrid											
nb=40	805	437	508	674	854	1144	1437	1687	1941	2188	2238
nb=72	800	1142	680	742	899	1150	1419	1709	2017	2274	2443
nb=100	762	1107	1630	831	945	1180	1446	1709	1998	2274	2446
nb=200	768	1110	1837	1805	1385	1469	1609	1840	2167	2468	2690

Table 11. Mflops, Cholesky factorizations including rearrangement, different nb values, Intel Pentium III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Lower Packed Hybrid											
nb=40	116	113	139	182	215	241	261	273	282	284	286
nb=72	118	116	136	169	194	226	257	277	296	308	317
nb=100	111	113	131	169	191	224	244	274	294	307	317
nb=200	108	113	136	137	148	180	219	257	292	310	324
Upper Packed Hybrid											
nb=40	131	136	157	199	233	259	278	294	300	302	303
nb=72	131	136	150	179	202	226	249	267	280	290	296
nb=100	127	133	148	182	202	227	241	267	282	289	297
nb=200	122	131	152	152	166	199	236	265	288	301	311

Table 12. Mflops, Cholesky factorizations, nb = 100, IBM Power4.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	747	951	1043	1024	1059	1101	1037	709	638	621	635
Packed LAPACK U	530	864	1201	1530	1772	2038	2055	1616	1460	1400	1426
Vendor Packed LAPACK L	1750	2359	2658	2346	3107	3560	3773	3870	3969	3815	3836
Full LAPACK L	440	722	1390	2119	2562	3242	3495	3797	3901	3787	4010
Full LAPACK U	436	646	1271	2063	2573	3390	3810	4008	4039	3975	4102
Vendor Full LAPACK L	1492	2165	2486	3194	3454	3677	3832	3921	4162	4037	4327
Vendor Full LAPACK U	1651	2448	3020	3269	3553	3878	3924	4000	4137	4053	4301
Packed Recursive+ L	170	379	593	1024	1586	2077	2621	3030	3434	3555	3943
Packed Recursive L	181	406	618	1060	1652	2133	2700	3111	3523	3604	3980
Packed Recursive+ U	194	418	629	1122	1724	2189	2931	3289	3690	3801	4094
Packed Recursive U	210	444	660	1185	1783	2249	3031	3401	3750	3858	4142
Packed Hybrid+ L	878	1488	2085	2721	3211	3754	3974	4112	4188	4200	4275
Packed Hybrid L	1006	1717	2334	2977	3441	3938	4149	4279	4266	4269	4309
Packed Hybrid+ U	1090	1702	2339	2792	3211	3690	3832	4034	4137	4150	4207
Packed Hybrid U	1095	1702	2339	2990	3416	3868	4045	4194	4214	4200	4249

provide our benchmark. We aim to get similar performance while saving storage with a packed format. We note that the vendor codes are much faster for small n, which is probably because the LAPACK code uses Level-2 BLAS (`_POTF2`) to factorize the blocks on the diagonal, but only slightly faster for $n \geq 1000$ where the speed of `_GEMM` is of prime importance.

There are two rows for each of the recursive and hybrid formats, according to whether the overheads of rearrangement to this format are included. Whether rearrangment will be needed in practice will vary from case to case. Where the data is generated by computer code, it may be equally efficient to generate it in the chosen format. We do not include rearrangement of the factor back to ordinary packed format since the recursive or hybid format is more suitable for forward and back substitution.

The two lines in Table 12 labelled 'Packed Hybrid+ L' and 'Packed Hybrid+ U' do not show exactly the same times as the rows labelled 'nb=100' in Table 5 because they come from separate runs, but the differences are very small.

The recursive algorithms achieve performance that approaches that of the LAPACK full codes when the order is large. This is because both are then doing most of their work in significant calls of the Level-3 BLAS `_GEMM`. However, for smaller n, their performance is poor, probably because of the larger ratio of procedure calls to actual computation.

The hybrid algorithm is much faster than the recursive algorithm for small n, significantly faster for medium n, and slightly faster for large n.

If we compare the lower-packed hybrid code with the compiled lower-full LAPACK code, we see that it is always faster, and significantly so for small n. We see this as very encouraging. Furthermore, it is slightly faster than the vendor packed code except for small n. It is sometimes faster than the vendor full code and would have been faster for all $n \geq 1600$ if we had switched to nb=200 at n=1600 (see Table 5). The hybrid code for the upper-packed format does not permit us to combine calls of `_GEMM`, so we expect it not to perform quite so well as the lower-packed

Table 13. Percentage overheads for rearrangment, IBM Power4.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed Recursive L	6.1	6.7	4.0	3.4	4.0	2.6	2.9	2.6	2.5	1.4	0.9
Packed Recursive U	7.6	5.9	4.7	5.3	3.3	2.7	3.3	3.3	1.6	1.5	1.2
Packed Hybrid L	12.7	13.3	10.7	8.6	6.7	4.7	4.2	3.9	1.8	1.6	0.8
Packed Hybrid U	0.5	0.0	0.0	6.6	6.0	4.6	5.3	3.8	1.8	1.2	1.0

Table 14. Mflops, Cholesky Factorizations, $nb = 200$, SUN UltraSPARC III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	183	247	296	312	321	325	328	331	315	200	169
Packed LAPACK U	203	256	302	336	365	393	412	425	409	256	218
Full LAPACK L	299	436	637	842	933	1086	864	1173	1203	1169	1236
Full LAPACK U	302	403	627	810	942	1087	1137	1211	1259	1209	1050
Packed Recursive+ L	95	202	275	426	550	727	913	1010	1093	1118	1215
Packed Recursive L	102	219	290	454	581	760	945	1043	1146	1162	1249
Packed Recursive+ U	90	187	251	381	554	687	864	1044	1119	1173	1238
Packed Recursive U	97	201	264	403	586	717	893	1076	1176	1227	1278
Packed Hybrid+ L	390	557	708	782	867	965	1080	1180	1201	1206	1254
Packed Hybrid L	529	778	959	973	1003	1077	1164	1267	1277	1257	1304
Packed Hybrid+ U	518	768	954	988	983	1032	1144	1181	1240	1280	1243
Packed Hybrid U	520	770	955	988	1008	1068	1178	1209	1268	1280	1275

Notes: Vendor Packed Lapack results very similar to Packed Lapack results. Vendor Full Lapack results similar to Full Lapack results.

format, but the difference is slight and it is faster for small n (see next paragraph).

The rearrangement overheads are shown as percentages in Table 13. We see that it is tiny for large n, but can be quite significant for small n. The higher percentage overhead for the hybrid code is a consequence of the greater speed of the hybrid code and not an inherent inefficiency of the rearrangement. The very small overhead for the upper-packed hybrid case for small n deserve an explanation. This because the upper packed and upper blocked hybrid formats are identical if there is only one block, that is, if $n \leq nb$. Thus the overheads arise only from the procedure call and some simple tests. On the other hand, for the lower blocked hybrid format (Figure 1), the block has to be rearranged from being held by columns to being held by rows.

We show in Table 14, factorization speeds on the SUN UltraSPARC III in the same format as that of Table 12. Here, we do not show the vendor LAPACK speeds since they were very similar to the corresponding speeds for the compiled codes. Once again, the packed LAPACK codes perform poorly, the recursive codes are competitive with the full LAPACK codes for large n but not for small n and the packed hybrid codes significantly outperform the packed recursive codes for small n. The hybrid performance is slightly better than that of the recursive code for large n, except in one case where the difference is within the timing uncertainty. When we compare the hybrid code with the full LAPACK code, we see that it faster in more than half the cases, with some particularly marked improvements for small n.

The rearrangement overheads for our principal packed hybrid code (hybrid L) are less than on the IBM, see Table 15. For the other packed hybrid U code, the

Table 15. Percentage overheads for rearrangment, SUN UltraSPARC III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed Recursive L	6.9	7.8	5.2	6.2	5.3	4.3	3.4	3.2	4.6	3.8	2.7
Packed Recursive U	7.2	7.0	4.9	5.5	5.5	4.2	3.2	3.0	4.8	4.4	3.1
Packed Hybrid L	26.3	28.4	26.2	19.6	13.6	10.4	7.2	6.9	6.0	4.1	3.8
Packed Hybrid U	0.4	0.3	0.1	0.0	2.5	3.4	2.9	2.3	2.2	0.0	2.5

overheads are similar to those of the packed hybrid L code except when n is small. The rearrangement overheads for the recursive algorithms are broadly comparable with those on the IBM.

We show in Tables 16, 17, 18 and 19, corresponding factorization speeds on our other computers. We do not show the vendor LAPACK speeds since they were always very similar to the corresponding speeds for the compiled codes. The packed LAPACK codes always perform poorly, the recursive codes are always competitive with the full LAPACK codes for large n but not for small n and the packed hybrid codes outperform the packed recursive codes for small n. There is a slight fall in the hybrid performance for large n on the Sgi Origin, but not on the other platforms. When we compare the hybrid code with the full LAPACK code, we see that it faster on each machine in more than half the cases, with some particularly marked improvements for small n. The performance of the upper hybrid code is on the whole better than that of the lower hybrid code, which is contrary to our expectation.

We do not show separate tables for the rearrangement overheads because they are similar to those on the IBM (Table 13).

Table 16. Mflops, Cholesky Factorizations, $nb = 100$, Sgi Origin 2000, R12000.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	111	144	157	143	144	145	145	143	80	55	35
Packed LAPACK U	77	123	167	203	230	251	265	260	121	52	21
Full LAPACK L	126	207	294	373	446	478	446	444	414	404	331
Full LAPACK U	116	175	238	306	375	435	474	491	496	446	471
Packed Recursive+ L	38	77	135	225	324	412	462	491	466	453	425
Packed Recursive L	41	83	145	241	344	431	478	506	484	468	435
Packed Recursive+ U	34	70	119	198	287	370	422	479	465	466	476
Packed Recursive U	37	74	128	210	302	385	435	493	482	482	489
Packed Hybrid+ L	193	269	330	339	382	427	461	484	476	461	439
Packed Hybrid L	236	324	390	375	411	449	480	499	496	474	447
Packed Hybrid+ U	235	320	389	352	386	429	462	484	470	458	453
Packed Hybrid U	238	322	389	375	411	449	480	500	490	472	460

7.3 Solution with many right-hand sides

We next consider the solution for many right-hand sides of a system whose matrix is already factorized. In Table 20, we show the speed on the IBM Power4 for $\max(100, n/10)$ right-hand sides. There is no rearrangement of the matrix here, since we assume that its factorization has been retained in its recursive or hybrid packed form. For all our values of n, the LAPACK codes using the packed formats

Table 17. Mflops, Cholesky Factorizations, $nb = 200$, HP Alpha EV6.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	198	238	265	276	268	275	273	219	141	134	123
Packed LAPACK U	163	238	292	352	374	400	415	325	203	181	184
Full LAPACK L	318	442	398	469	526	607	607	625	655	679	706
Full LAPACK U	300	402	411	481	552	619	661	701	724	738	752
Packed Recursive+ L	109	195	267	379	480	535	622	659	666	683	723
Packed Recursive L	122	216	286	405	508	555	642	674	677	691	729
Packed Recursive+ U	100	186	246	351	458	514	571	618	650	680	733
Packed Recursive U	111	202	264	376	484	538	588	645	666	691	740
Packed Hybrid+ L	343	455	491	537	525	591	648	695	731	764	789
Packed Hybrid L	426	498	566	596	562	618	669	720	751	779	799
Packed Hybrid+ U	432	521	554	595	566	591	662	701	744	767	790
Packed Hybrid U	437	521	554	595	572	616	669	714	758	781	799

Table 18. Mflops, Cholesky Factorizations, $nb = 200$, HP-UX Itanium rx2600s.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	218	328	463	644	825	952	1039	660	609	595	590
Packed LAPACK U	167	224	272	320	359	387	406	416	425	430	433
Full LAPACK L	376	581	838	1307	1728	2189	2546	2777	2904	3028	3141
Full LAPACK U	433	734	559	633	889	1146	1385	1821	2068	2540	2676
Packed Recursive+ L	110	232	378	698	1079	1608	2170	2531	2758	2861	3013
Packed Recursive L	121	256	411	742	1147	1675	2305	2666	2874	2942	3056
Packed Recursive+ U	82	176	313	580	951	1479	1967	2424	2576	2726	2867
Packed Recursive U	89	187	333	616	1004	1551	2063	2531	2659	2785	2910
Packed Hybrid+ L	657	1017	1689	1731	1279	1384	1552	1829	2089	2216	2402
Packed Hybrid L	763	1139	1849	1815	1390	1504	1661	1910	2167	2274	2438
Packed Hybrid+ U	768	1110	1837	1805	1385	1469	1609	1840	2167	2468	2690
Packed Hybrid U	768	1124	1849	1805	1394	1489	1664	1886	2214	2480	2724

Table 19. Mflops, Cholesky Factorizations, $nb = 40$, Intel Pentium III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	111	130	104	99	95	57	41	37	36	35	35
Packed LAPACK U	96	127	130	137	146	102	82	77	75	72	71
Full LAPACK L	106	113	161	193	226	247	266	289	300	298	297
Full LAPACK U	81	98	144	180	210	237	213	278	244	303	325
Packed Recursive+ L	32	60	93	148	161	199	225	259	288	306	325
Packed Recursive L	34	64	101	160	170	212	236	271	296	313	330
Packed Recursive+ U	34	66	101	161	175	210	233	263	288	307	325
Packed Recursive U	36	72	111	175	187	223	242	273	296	313	330
Packed Hybrid+ L	117	113	139	183	215	241	261	276	284	285	287
Packed Hybrid L	134	131	156	202	236	258	274	288	292	290	290
Packed Hybrid+ U	132	135	156	201	232	263	281	294	303	304	305
Packed Hybrid U	134	151	174	222	253	283	295	307	311	309	308

perform less well than the other codes. The hybrid codes give performance that is always at least as good as the recursive codes, and is much better for small n.

We show comparable figures for the other systems in Tables 21, 22, 23, 24, and 25. We see that the performance of the packed LAPACK codes is poor in

Table 20. Mflops, Solution, many right-hand sides, Notes: Results for Vendor Packed Lapack L and Vendor Full Lapack very similar to corresponding Lapack results. $nb = 100$, $mb = 100$, IBM Power4.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	919	1298	1577	1822	2027	2182	2088	1733	1526	1460	1456
Packed LAPACK U	912	1296	1572	1836	2044	2197	2088	1710	1517	1467	1469
Full LAPACK L	3195	3368	3750	4021	4210	4266	4259	4257	4441	4320	4368
Full LAPACK U	3205	3392	3740	4045	4175	4238	4259	4222	4495	4320	4353
Packed Recursive L	1240	1510	1956	2658	3075	3306	3524	3734	4147	4166	4571
Packed Recursive U	1225	1493	1935	2655	3091	3306	3510	3750	4147	4166	4555
Packed Hybrid L	2359	2941	3411	3778	3991	4084	4231	4285	4362	4422	4620
Packed Hybrid U	2374	2927	3389	3736	3965	4132	4231	4293	4415	4380	4555

Notes: Results for Vendor Packed Lapack L and Vendor Full Lapack very similar to corresponding Lapack results.

every case and is always inferior to the other codes. The hybrid codes always give performance that is reasonally close to that of the full code. It is slightly inferior for small n but slightly superior for large n. On the SUN, Sgi Origin, and HP Alpha, the recursive codes are inferior to the hybrid codes for small n, but superior for large n; this is particularly marked on the SUN. For the Itanium, the recursive code is remarkably successful, consistently out-performing the full code and the hybrid code. On the Pentium III, the full, recursive, and hybrid codes have broadly comparable performance.

We are very pleased to see that the performance of the hybrid code continues to improve as n gets very large, which demonstrates the success of working with blocks of mb columns and rearranging each to the form illustrated in Figure 7. We believe that this explains why the packed hybrid codes outperform the Full LAPACK codes for large n on the SUN, Origin, Alpha, and Itanium. For $n \leq nb$, the rearrangement probably does not speed up the solution since the columns of B are already contiguous in memory, but we have not written special code that avoids the rearrangement in this case.

Table 21. Mflops, Solution, many right-hand sides, $nb = 200$, $mb = 100$, SUN UltraSPARC III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	289	341	373	265	220	209	203	199	119	63	54
Packed LAPACK U	276	326	357	294	271	277	281	270	179	99	87
Full LAPACK L	901	933	1045	1089	1103	1279	1238	1300	1268	1312	1299
Full LAPACK U	894	974	1042	1095	1103	1276	1244	1312	1261	1390	1290
Packed Recursive L	280	467	489	678	849	981	1081	1241	1296	1362	1436
Packed Recursive U	267	466	477	659	843	968	1071	1239	1291	1360	1434
Packed Hybrid L	766	852	959	1023	1083	1236	1317	1334	1373	1420	1436
Packed Hybrid U	767	858	959	1031	1074	1234	1316	1336	1401	1453	1435

Notes: Vendor Packed Lapack results very similar to Packed Lapack results (slightly inferior for small n). Vendor Full Lapack results inferior to Full Lapack results.

Table 22. Mflops, Solution, many right-hand sides, $nb = 100$, $mb = 100$, Sgi Origin 2000, R12000.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	125	162	185	201	212	218	221	170	57	56	39
Packed LAPACK U	123	161	184	200	211	218	220	171	51	53	38
Full LAPACK L	264	330	394	442	483	510	510	507	492	481	447
Full LAPACK U	260	316	373	420	461	496	491	512	477	476	446
Packed Recursive L	206	209	319	370	434	475	483	516	497	480	484
Packed Recursive U	206	209	319	370	434	475	487	516	498	494	487
Packed Hybrid L	229	290	345	403	444	482	504	509	513	517	510
Packed Hybrid U	228	290	345	402	445	481	504	509	503	518	506

Notes: Results for Vendor Packed Lapack and Vendor Full Lapack very similar to corresponding Lapack results.

Table 23. Mflops, Solution, many right-hand sides, $nb = 200$, $mb = 100$, HP Alpha EV6.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	224	293	360	380	403	421	427	318	254	203	203
Packed LAPACK U	226	296	358	384	403	428	422	330	219	203	197
Full LAPACK L	443	478	556	624	669	718	732	712	750	759	762
Full LAPACK U	444	488	553	625	668	709	732	727	750	756	761
Packed Recursive L	328	395	425	576	649	702	687	712	687	732	766
Packed Recursive U	324	389	432	579	635	702	672	712	687	732	766
Packed Hybrid L	407	460	526	597	656	722	739	774	792	804	827
Packed Hybrid U	413	448	522	597	645	717	747	774	799	811	823

Notes: Results for Vendor Packed Lapack same as Packed Lapack and for Vendor Full Lapack same as Full Lapack, apart from timing uncertainties.

Table 24. Mflops, Solution, many right-hand sides, $nb = 200$, $mb = 100$, HP-UX Itanium rx2600s.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	223	305	385	471	538	576	608	628	640	649	655
Packed LAPACK U	225	308	385	465	532	578	614	632	642	648	655
Full LAPACK L	272	411	593	844	1150	1525	1878	2222	2452	2637	2813
Full LAPACK U	271	406	593	835	1157	1538	1881	2222	2452	2626	2813
Packed Recursive L	712	958	1325	1641	2071	2440	2678	2765	3015	3094	3062
Packed Recursive U	695	958	1289	1617	2077	2392	2636	2784	3015	3094	3062
Packed Hybrid L	251	396	581	817	1103	1473	1853	2196	2560	2765	2976
Packed Hybrid U	251	396	577	808	1103	1463	1853	2193	2544	2777	2969

Notes: Results for Vendor Packed Lapack same as Packed Lapack and for Vendor Full Lapack same as Full Lapack, apart from timing uncertainties. No significant difference with mb=200 or mb=48.

7.4 Solution with one right-hand side

Finally, we have measured the speeds while solving for single right-hand side, for a system whose matrix is already factorized. The results are shown in Tables 26 to 31. It is inherently more difficult to obtain a high speed when n is large since the matrix will need to be read into cache once for forward substitution and once for back-substitution. It should therefore be possible for the packed codes to execute faster than the full codes. For the LAPACK codes, this is quite often the case; it is particularly so for large n on the Sgi Origin and the Itanium. However, on the SUN, the packed LAPACK codes do not perform well.

Table 25. Mflops, Solution, many right-hand sides, $nb = 40$, $mb = 100$, Intel Pentium III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	110	128	123	120	123	94	67	60	55	54	53
Packed LAPACK U	114	132	127	123	124	95	67	59	55	53	53
Full LAPACK L	160	131	223	259	277	300	262	317	339	346	327
Full LAPACK U	161	192	231	259	274	294	262	319	341	346	325
Packed Recursive L	164	160	230	228	274	290	267	314	330	342	352
Packed Recursive U	163	157	225	226	271	285	268	310	328	341	352
Packed Hybrid L	144	187	221	256	277	300	313	314	311	315	317
Packed Hybrid U	141	188	222	257	277	302	315	314	305	310	314

Notes: Results for Vendor Packed Lapack within timing uncertainties of Packed Lapack Results for Vendor Full Lapack same as Full Lapack, apart from timing uncertainties.

Table 26. Mflops, Solution, one right-hand side, $nb = 40$, IBM Power4.

n	40	64	100	160	250	400	640	1000	1600	2500
Packed LAPACK L	906	1205	1496	1798	2021	2253	1907	1645	1479	1432
Packed LAPACK U	899	1187	1488	1814	2022	2275	1868	1612	1508	1473
Vendor Packed Lapack L	894	1196	1489	1789	2022	2257	1892	1592	1488	1479
Full LAPACK L	896	1175	1505	1852	2073	2206	1669	1331	1358	1432
Full LAPACK U	885	1178	1537	1804	2085	2217	1578	1294	1313	1390
Vendor Full LAPACK L	896	1161	1512	1824	2069	2204	1683	1309	1356	1430
Vendor Full LAPACK U	860	1152	1520	1793	2066	2231	1579	1297	1306	1444
Packed Recursive L	283	417	555	810	887	1335	1299	1095	1239	1290
Packed Recursive U	280	419	558	808	907	1344	1356	1107	1242	1274
Packed Hybrid L	880	1177	1498	1647	1851	2098	1480	1150	1027	1009
Packed Hybrid U	874	1167	1474	1622	1826	1993	1452	1121	1017	993

Table 27. Mflops, Solution, one right-hand side, $nb = 200$, SUN UltraSPARC III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	285	339	383	268	221	208	204	199	84	56	51
Packed LAPACK U	270	325	371	290	274	280	281	280	154	96	85
Full LAPACK L	363	479	575	657	665	798	798	495	370	285	286
Full LAPACK U	373	472	601	698	667	786	791	540	340	306	288
Packed Recursive L	71	113	155	229	299	404	504	560	318	258	252
Packed Recursive U	72	114	155	229	306	405	505	566	314	263	277
Packed Hybrid L	247	310	354	294	338	411	519	554	348	265	229
Packed Hybrid U	240	309	352	292	337	411	520	584	325	231	224

Notes: Vendor Packed Lapack results inferior to Full Lapack results.

For small n, the hybrid codes give consistently better performance than the recursive codes, and they are broadly comparable with the packed LAPACK codes.

For large n, the recursive code is usually better than the hybrid code, but there is a notable exception on the Pentium III (Table 31) and there is little difference on the Sgi Origin (Table 28). When compared to the best LAPACK code, the recursive code is usually slightly slower but is much faster on the Sgi Origin and Itanium. Where the packed LAPACK code performs well, consideration should be given to rearranging the factorized matrix back to the packed format.

For the IBM processor, we have omitted the results for n=4000 since those that

Table 28. Mflops, Solution, one right-hand side, $nb = 100$, Sgi Origin 2000, R12000.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	117	157	199	208	211	217	215	184	78	59	55
Packed LAPACK U	116	156	193	204	209	215	213	182	77	56	56
Full LAPACK L	166	236	254	258	260	263	161	120	33	19	12
Full LAPACK U	175	230	252	243	283	270	160	160	34	19	12
Packed Recursive L	30	45	66	98	133	171	210	214	82	88	74
Packed Recursive U	30	45	67	100	132	174	215	216	82	87	86
Packed Hybrid L	115	151	190	233	270	295	323	327	91	58	87
Packed Hybrid U	112	148	187	231	266	291	313	316	83	60	53

Notes: Results for Vendor Packed Lapack and Vendor Full Lapack very similar to corresponding Lapack results.

Table 29. Mflops, Solution, one right-hand side, $nb = 200$, HP Alpha EV6.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	217	290	351	400	409	425	424	322	228	203	192
Packed LAPACK U	220	289	358	399	406	423	410	306	238	214	192
Full LAPACK L	190	249	298	312	324	338	283	257	217	199	192
Full LAPACK U	191	243	286	304	320	334	289	263	214	197	174
Packed Recursive L	105	152	200	245	286	312	354	261	194	172	174
Packed Recursive U	102	152	200	242	280	317	344	257	194	172	174
Packed Hybrid L	215	293	345	379	377	403	384	253	201	183	174
Packed Hybrid U	218	284	343	382	381	403	379	258	201	183	160

Notes: Results for Vendor Packed Lapack within timing uncertainties of Packed Lapack Results for Vendor Full Lapack same as Full Lapack, apart from timing uncertainties.

Table 30. Mflops, Solution, one right-hand side, $nb = 200$, HP-UX Itanium rx2600s.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	221	301	382	471	518	587	655	622	640	649	639
Packed LAPACK U	225	301	385	462	542	584	738	663	657	653	640
Full LAPACK L	181	263	341	452	522	625	317	307	314	285	266
Full LAPACK U	183	261	344	438	554	603	344	312	317	287	266
Packed Recursive L	76	118	173	268	370	529	615	662	779	886	1065
Packed Recursive U	72	118	169	258	382	527	591	658	771	888	799
Packed Hybrid L	194	279	371	449	618	847	743	649	624	620	640
Packed Hybrid U	193	280	370	450	620	855	776	670	640	633	639

Notes: Results for Vendor Packed Lapack within timing uncertainties of Packed Lapack except that Vendor Packed Lapack achieved 799 Mflops for n=4000. Results for Vendor Full Lapack same as Full Lapack, apart from timing uncertainties.

we obtained look unreliable. We are investigating this case further.

8. SUMMARY AND CONCLUSIONS

Our primary goal when we commenced this work was to investigate how much better the recursive algorithm performed than a simple blocking algorithm. We expected that it to be better since it can take advantage of all levels of cache, but did not know by how much. What we have found is that it tends to be inferior for systems of small order where a single well-chosen block size can take good advantage of level-1 cache, often by a factor of about 3-4 for our implementations.

Table 31. Mflops, Solution, one right-hand side, $nb = 40$, Intel Pentium III.

n	40	64	100	160	250	400	640	1000	1600	2500	4000
Packed LAPACK L	112	121	120	128	100	67	59	57	55	54	54
Packed LAPACK U	115	140	131	133	119	85	65	59	54	53	54
Full LAPACK L	38	48	68	90	75	70	61	56	57	52	45
Full LAPACK U	39	51	73	95	83	69	60	57	56	52	44
Packed Recursive L	12	19	28	42	44	51	52	52	52	49	45
Packed Recursive U	12	19	28	42	44	50	51	51	52	48	46
Packed Hybrid L	100	108	121	139	146	120	110	109	109	109	106
Packed Hybrid U	91	108	120	137	129	111	101	98	98	97	99

Notes: Results for Vendor Packed Lapack within timing uncertainties of Packed Lapack Results for Vendor Full Lapack same as Full Lapack, apart from timing uncertainties.

For large systems, performance of the recursive algorithm is similar on the SUN and Sgi, worse on the IBM, Alpha, and better only on the Itanium and Pentium. An advantage of the recursive method, of course, is that no choice of block size is needed.

On balance, we therefore see that the hybrid algorithm performs better than the recursive algorithm. Also, we see that it often outperforms the full-storage LAPACK code.

In the course of this investigation, we have considered carefully exactly how the blocks should be chosen to permit good advantage to be taken of level-1 cache during factorization and solution to allow rapid rearrangement to the block form without undue use of temporary memory. We noted that where the data is generated by computer code, it may be equally efficient to generate it in the chosen format so that no rearrangement is needed.

We have also developed some new kernel codes for the Cholesky factorization of the diagonal blocks because the LAPACK kernel _POTF2 is unsatisfactory as it uses Level-2 BLAS. We obtained remarkably fast performance for our kernel by using mini-blocks of order 2 in standard Fortran code.

9. ACKNOWLEDGEMENTS

The work described here is partly the outcome of collaborations with HPCN at the University of Umea, Sweden and UNI•C in Lyngby, Denmark. At Umea we thank Isak Jonsson.

We would like to thank Niels Carl W Hansen for consulting on the IBM and SGI systems; Bernd Dammann for consulting on the SUN system; Susanne Balle and Martin Antony Walker for making the HP Itanium 2 system available to us; Tim Regan for consulting on the HP Itanium 2 system; and Minka and Alexander Karaivanov for several discussions.

REFERENCES

Agarwal, R., Gustavson, F., and Zubair, M. 1994. Exploiting functional parallelism on power2 to design high-performance numerical algorithms. *IBM Journal of Research and Development 38*, 5 (September), 563–576.

Andersen, B., Gunnels, J., Gustavson, F., Reid, J., and Waśniewski, J. 2004. Fortran 90 Subroutines for the Cholesky Algorithm in Blocked Hybrid Format. For submission to the ACM Transactions on Mathematical Software.

Andersen, B., Gunnels, J., Gustavson, F., and Waśniewski, J. 2002. A Recursive Formulation of the Inversion of symmetric positive definite Matrices in Packed Storage Data Format. In J. Fagerholm, J. Haataja, J. Järvinen, M. Lyly, and P. R. V. Savolainen Eds., *Proceedings of the* 6^{th} *International Conference, PARA 2002, Applied Parallel Computing*, Number 2367 in Lecture Notes in Computer Science (Espoo, Finland, June 2002), pp. 287–296. Springer.

Andersen, B., Gustavson, F., and Waśniewski, J. 2001. A Recursive Formulation of Cholesky Facorization of a Matrix in Packed Storage. *ACM Transactions on Mathematical Software 27*, 2 (Jun), 214–244.

Anderson, E., Bai, Z., Bischof, C., Blackford, L. S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. 1999. *LAPACK Users' Guide* (Third ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

Calahan, D. 1986. Block-Oriented local memory-based linear equation solution on the Cray-2; Uniprocessor algorithms. In *Proceedings International Conference on Parallel Processing*, IEEE Computer Society Press (Yew York, USA, August 1986).

Chatterjee, S., Jain, V. V., Lebeck, A. R., Mundhra, S., and Thottethodi, M. 1999. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing* (1999), pp. 444–453.

Dongarra, J., Bunch, J., Moler, C., and Stewart, G. 1979. *Linpack Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

Dongarra, J., Du Croz, J., Duff, I. S., and Hammarling, S. 1990. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft. 16*, 1 (March), 1–17.

Dongarra, J., Du Croz, J., Hammarling, S., and Hanson, R. J. 1988. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft. 14*, 1 (March), 1–17.

Duff, I. S. and Reid, J. K. 1996. Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software 22*, 2, 227–257.

Frens, J. D. and Wise, D. S. 1997. Auto-blocking matrix-multiplication or tracking blas3 performance from source code. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles and practice of parallel programming* (1997), pp. 206–216. ACM Press.

Gallivan, K., Jalby, W., Meier, U., and Sameh, A. 1987. The impact of hierarchical memory systems on linear algebra algorithm design. CSRD Report 625 (SEP), CSRD.

Gustavson, F. 1997. Recursion Leads to Automatic Variable Blocking for Dense Linear-Algebra Algorithms. *IBM Journal of Research and Development 41*, 6 (November), 737–755.

Gustavson, F. and Jonsson, I. 2000. Minimal storage high performance cholesky via blocking and recursion. *IBM Journal of Research and Development 44*, 6 (Nov), 823–849.

IBM. 1986. *Engineering and Scientific Subroutine Library, Guide and Reference*. First Edition (Program Number 5668-863).

Lawson, C. L., Hanson, R. J., Kincaid, D., and Krogh, F. T. 1979. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Soft. 5*, 308–323.

Valsalam, V. and Skjellum, A. 2002. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *14*, 10 (Aug.), 805–839.

Waśniewski, J., Andersen, B., and Gustavson, F. 1998. Recursive Formulation of Cholesky Algorithm in Fortran 90. In B. Kågström, J. Dongarra, E. Elmroth, and J. Waśniewski Eds., *Proceedings of the* 4^{th} *International Workshop, Applied Parallel Computing, Large Scale Scientific and Industrial Problems, PARA'98*, Number 1541 in Lecture Notes in Computer Science Number (Umeå, Sweden, June 1998), pp. 574–578. Springer.

Whaley, R., Petitet, A., and Dongarra, J. 2000. ATLAS: Automatically Tuned Linear Algebra Software. http://www.netlib.org/atlas/. University of Tennessee at Knoxville, Tennessee, USA.