# Compiler Blockability of Dense Matrix Factorizations[*]

Steve Carr[†] R. B. Lehoucq[‡]

August 29, 1996

**Abstract**

The goal of the LAPACK project is to provide efficient and portable software for dense numerical linear algebra computations. By recasting many of the fundamental dense matrix computations in terms of calls to an efficient implementation of the BLAS (Basic Linear Algebra Subprograms), the LAPACK project has, in large part, achieved its goal. Unfortunately, the efficient implementation of the BLAS often results in machine-specific code that is not portable across multiple architectures without a significant loss in performance or a significant effort to re-optimize them.

This paper examines whether most of the hand optimizations performed on matrix factorization codes are unnecessary because they can (and should) be performed by the compiler. We believe that it is better for the programmer to express algorithms in a machine-independent form and allow the compiler to handle the machine-dependent details. This gives the algorithms portability across architectures and removes the error-prone, expensive and tedious process of hand optimization. Although there currently exist no production compiler that can perform all the loop transformations discussed in this paper, a description of current research in compiler technology is provided that will prove beneficial to the numerical linear algebra community.

We show that the Cholesky and LU factorizations may be optimized automatically by a compiler to be as efficient as the same hand-optimized version found in LAPACK. We also show that the QR factorization may be optimized by the compiler to perform comparably with the hand-optimized LAPACK version on modest matrix sizes. Our approach allows us to conclude that with the advent of the compiler optimizations discussed in this paper, matrix factorizations may be efficiently implemented in a machine-independent form.

## 1 Introduction

The processing power of microprocessors and supercomputers has increased dramatically and continues to do so. At the same time, the demand on the memory system of a computer is to increase dramatically in size. Due to financial costs, typical workstations and massively parallel machines cannot use memory chips that have the latency and bandwidth required by today's processors. Instead, main memory is constructed of cheaper and slower technology and the resulting delays may be up to hundreds of cycles for a single memory access.

To alleviate the memory speed problem, machine architects construct a hierarchy of memory where the highest level (registers) is the smallest and fastest and each lower level is larger but

[*]Research supported by NSF Grant CCR-9120008 and by NSF grant CCR-9409341. The second author was also supported by the U.S. Department of Energy Contracts DE-FG0f-91ER25103 and W-31-109-Eng-38.

[†]Department of Computer Science, Michigan Technological University, Houghton MI 49931, `carr@cs.mtu.edu`.

[‡]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439 `lehoucq@mcs.anl.gov`.

slower. The bottom of the hierarchy for our purposes is main memory. Typically, one or two levels of cache memory fall between registers and main memory. The cache memory is faster than main memory, but is often a fraction of the size. The cache memory serves as a buffer for the most recently accessed data of a program (the working set). The cache becomes ineffective when the working set of a program is larger than its size.

The three factorizations considered in this paper, the LU, Cholesky, and QR, are among the most frequently used by numerical linear algebra and its applications. The first two are used for solving linear systems of equations while the last is typically used in linear least squares problems. For square matrices of order $n$, all three factorizations involve on the order of $n^3$ floating point operations for data that needs $n^2$ memory locations. With the advent of vector and parallel supercomputers, the efficiency of the factorizations were seen to depend dramatically upon the algorithmic form chosen for the implementation [14, 16, 29]. These studies concluded that managing the memory hierarchy is the single most important factor governing the efficiency of the software implementation computing the factorization.

The motivation of the LAPACK [2] project was to recast the algorithms in the EISPACK [32] and LINPACK [15] software libraries with *block* ones. A block form of an algorithm restructures the algorithm in terms of matrix operations that attempt to minimize the amount of data moved within the memory hierarchy while keeping the arithmetic units of the machine occupied. LAPACK blocks many dense matrix algorithms by restructuring them to use the level 2 and 3 BLAS [11, 12]. The motivation for the Basic Linear Algebra Subprograms, BLAS [26], was to provide a set of commonly used vector operations so that the programmer could invoke the subprograms instead of writing the code directly. The level 2 and 3 BLAS followed with matrix-vector and matrix-matrix operations, respectivly, that are often necessary for high efficiency across a broad range of high performance computers. The higher level BLAS better utilize the underlying memory hierarchy. As with the level 1 BLAS, responsibility for optimizing the higher level BLAS was left to the machine vendor or another interested party.

This study investigates whether a compiler has the ability to block matrix factorizations. Although the compiler transformation techniques may be applied directly to the BLAS, it is interesting to draw a comparison with applying them directly to the factorizations. The benefit is the possibility of a BLAS-less linear algebra package that is nearly as efficient as LAPACK. For example, in [27], it was demonstrated that on some computers, the best LU factorization was an inlined approach even when a highly optimized set of BLAS were available.

We deem an algorithm *blockable* if a compiler can automatically derive the most efficient block algorithm (for our study, the one found in LAPACK) from its corresponding machine-independent point algorithm. In particular, we show that LU and Cholesky factorizations are blockable algorithms. Unfortunately, QR factorization with Householder transformations is not blockable. However, we show an alternative block algorithm for QR that can be derived using the same compiler methods as those used for LU and Cholesky factorizations.

This study has yielded two major results. The first, which is detailed in another paper [8], reveals that the hand loop unrolling performed when optimizing the level 2 and 3 BLAS [11, 12] is often unnecessary. While the BLAS are useful, the hand optimization that is required to obtain good performance on a particular architecture may be left to the compiler. Experiments show that, in most cases, the compiler can automatically unroll loops as effectively as hand optimization. The second result, which we discuss in this paper, reveals that it is possible to block matrix factorizations automatically. Our results show that the block algorithms derived by the compiler are competitive with those of LAPACK [2]. For modest sized matrices (on the order of 200 or less), the compiler-derived variants are often superior.

We begin our presentation with a review of background material related to compiler optimization. Then, we describe a study of the application of these transformations to derive the three block algorithms in LAPACK considered above from their corresponding point algorithms. We present an experiment comparing the performance of hand-optimized LAPACK algorithms with the compiler-derived algorithms attained using our techniques. We also breifly discuss a recent approach of Kågström, Ling and Van Loan [20] that aims to reduce the software costs associated with optimizing the level 3 BLAS. Finally, we summarize our results and provide and draw some general conclusions.

## 2 Background

The transformations that we use to create the block versions of matrix factorizations from their corresponding point versions are well known in the mathematical software community [13]. This section introduces the fundamental tools that the compiler needs to perform the same transformations automatically. The compiler optimizes point versions of matrix factorizations through analysis of array access patterns rather than through linear algebra.

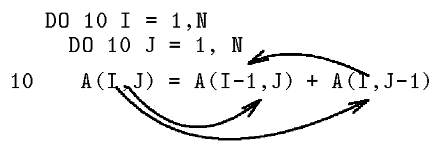
### 2.1 Dependence

The most important tool available to the compiler is that of *dependence*—the same tool used in vectorization and parallelization. Dependence is necessary for determining the legality of compiler transformations to create blocked versions of matrix factorizations.

A dependence exists between two statements if there exists a control flow path from the first statement to the second, and both statements reference the same memory location [23].

- If the first statement writes to the location and the second reads from it, there is a *true dependence*, also called a *flow dependence*.
- If the first statement reads from the location and the second writes to it, there is an *antidependence.*
- If both statements write to the location, there is an *output dependence.*
- If both statements read from the location, there is an *input dependence.*

A dependence is *carried* by a loop if the references at the source and sink (beginning and end) of the dependence are on different iterations of the loop and the dependence is not carried by an outer loop [1]. In the loop below, there is a true dependence from `A(I,J)` to `A(I-1,J)` carried by the I-loop, a true dependence from `A(I,J)` to `A(I,J-1)` carried by the J-loop and an input dependence from `A(I,J-1)` to `A(I-1,J)` carried by the `I-loop`.



```
      DO 10 I = 1,N
        DO 10 J = 1, N
10       A(I,J) = A(I-1,J) + A(I,J-1)
```

To enhance the dependence information, *section* analysis can be used to describe the portion of an array that is accessed by a particular reference or set of references [4, 19]. Sections describe common substructures of arrays such as elements, rows, columns and diagonals. As an example of section analysis consider the following loop.

```
      DO 10 I = 1,N
        DO 10 J = 1, 10
   10     A(J,I) = ...
```

If `A` were declared to be $100 \times 100$, the section of `A` accessed in the loop would be that shown in the shaded portion of Figure 1.

Matrix factorization codes require us to enhance basic dependence information because only a portion of the matrix is involved in the block update. The compiler uses section analysis to reveal that portion of the matrix that can be block updated. Section 3.1.1 discusses this in detail.

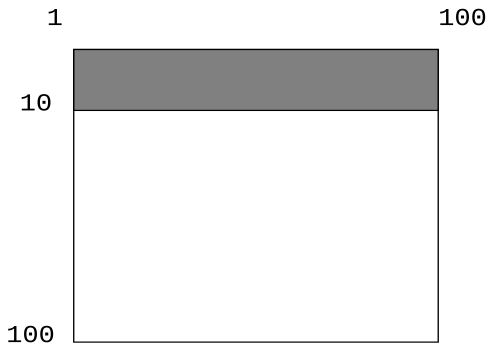


**Figure 1** Section of `A`

## 3 Automatic Blocking of Dense Matrix Factorizations

In this section, we show how to derive the block algorithms for the LU and the Cholesky factorizations using current compiler technology and section analysis to enhance dependence information. We also show that the QR factorization with Householder transformations is not blockable. However, we present a performance-competitive version of the QR factorization that is derivable by the compiler.

### 3.1 LU Factorization

The LU decomposition factors a non-singular matrix $A$ into the product of two matrices, $L$ and $U$, such that $A = LU$ [17]. $L$ is a unit lower triangular matrix and $U$ is an upper triangular matrix. This factorization can be obtained by multiplying the matrix $A$ by a series of elementary lower triangular matrices, $M_{n-1} \cdots M_1$ and pivot matrices $P_{n-1} \cdots P_1$, where $L^{-1} = M_{n-1}P_{n-1} \cdots M_1P_1$ and $U = L^{-1}A$. The pivot matrices are used to make the LU factorization a numerically stable process.

We first examine the blockablity of LU factorization. Since pivoting creates its own difficulties, we first show how to block LU factorization without pivoting. We then show how to handle pivoting.

#### 3.1.1 No Pivoting

Consider the following algorithm for LU factorization.

```
      DO 10 K = 1,N-1
        DO 20 I = K+1,N
20        A(I,K) = A(I,K) / A(K,K)
        DO 10 J = K+1,N
          DO 10 I = K+1,N
10          A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

This point algorithm is refered to as an unblocked right–looking [10] algorithm. It exhibits poor cache performance on large matrices. To transform the point algorithm to the block algorithm, the compiler must perfom *strip-mine-and-interchange* on the K-loop [35, 30, 33]. This transformation is used to create the block update of A. To apply this transformation, we first *strip* the K-loop into fixed size sections (this size is dependent upon the target architectures cache characteristics and is beyond the scope of this paper [25, 9]) as shown below.

```
      DO 10 K = 1,N-1,KS
        DO 10 KK = K,MIN(K+KS-1,N-1)
          DO 20 I = KK+1,N
20          A(I,KK) = A(I,KK) / A(KK,KK)
          DO 10 J = KK+1,N
            DO 10 I = KK+1,N
10            A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

Here KS is the machine-dependent strip size that is related to the cache size. To complete the transformation, the KK-loop must be distributed around the loop that surrounds statement 20 and around the loop nest that surrounds statement 10 before being interchanged to the innermost position of the loop surrounding statement 10 [34]. This distribution yields:

```
      DO 10 K = 1,N-1,KS
        DO 20 KK = K,MIN(K+KS-1,N-1)
          DO 20 I = KK+1,N
20          A(I,KK) = A(I,KK) / A(KK,KK)
        DO 10 KK = K,MIN(K+KS-1,N-1)
          DO 10 J = KK+1,N
            DO 10 I = KK+1,N
10            A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

Unfortunately, the loop is no longer correct. This loop scales a number of values before it updates them. Dependence analysis allows the compiler to detect and avoid this change in semantics by recoginizing the dependence cycle between A(I,KK) in statement 20 and A(I,J) in statement 10 carried by the KK-loop.

Using basic dependence analysis only, it appears that the compiler would be prevented from blocking LU factorization due to the cycle. However, enhancing dependence analysis with section information reveals that the cycle only exists for a portion of the data accessed in both statements. Figure 2 shows the sections of the array A accessed for the entire execution of the KK-loop. The section accessed by A(I,KK) in statement 20 is a subset of the section accessed by A(I,J) in statement 10.

Since the recurrence exists for only a portion of the iteration space of the loop surrounding statement 10, we can split the J-loop into two loops – one loop iterating over the portion of A where the dependence cycle exists, and one loop iterating over the portion of A where the cycle does not exist – using a transformation called *index-set splitting* [35]. J can be split at the point J = K+KS-1 to create the two loops as shown below.

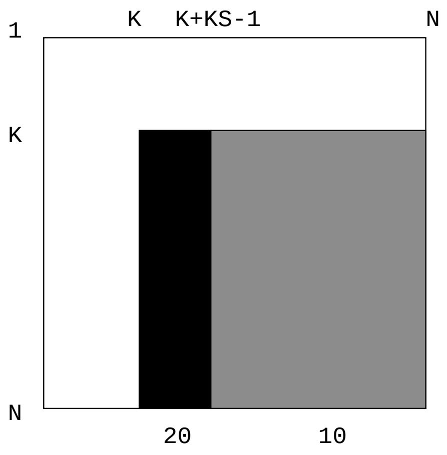


**Figure 2** Sections of `A` in LU Factorization

```
      DO 10 K = 1,N-1,KS
        DO 10 KK = K,MIN(K+KS-1,N-1)
          DO 20 I = KK+1,N
 20         A(I,KK) = A(I,KK) / A(KK,KK)
          DO 30 J = KK+1,MIN(K+KS-1,N)
            DO 30 I = KK+1,N
 30           A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
          DO 10 J = K+KS,N
            DO 10 I = KK+1,N
 10           A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

Now the dependence cycle exists between statements `20` and `30`, and statement `10` is no longer in the cycle. Strip-mine-and-interchange can be continued by distributing the `KK`-loop around the two new loops as shown below.

```
      DO 10 K = 1,N-1,KS
        DO 30 KK = K,MIN(K+KS-1,N-1)
          DO 20 I = KK+1,N
 20         A(I,KK) = A(I,KK) / A(KK,KK)
          DO 30 J = KK+1,MIN(K+KS-1,N)
            DO 30 I = KK+1,N
 30           A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
        DO 10 KK = K,MIN(K+KS-1,N-1)
          DO 10 J = K+KS,N
            DO 10 I = KK+1,N
 10           A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

To finish strip-mine-and-interchange, we need to move the `KK`-loop to the innermost position in the nest surrounding statement `10`. However, the lower bound of the `I`-loop contains a reference to `KK`. This creates a *triangular* iteration space as shown in Figure 3. To interchange the `KK` and `I` loops, the intersection of the line `I=KK+1` with the iteration space at the point `(K,K+1)` must be handled. Therefore, interchanging the loops requires the `KK`-loop to iterate over a trapezoidal region with an upper bound of `I-1` until `I-1` $>$ `K+KS-1` (see Wolfe, and Carr and Kennedy for more details on transforming non-rectangular loop nests [35, 7]). This gives the following loop nest.

```
      DO 10 K = 1,N-1,KS
        DO 30 KK = K,MIN(K+KS-1,N-1)
          DO 20 I = KK+1,N
20          A(I,KK) = A(I,KK) / A(KK,KK)
          DO 30 J = KK+1,MIN(K+KS-1,N)
            DO 30 I = KK+1,N
30            A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
        DO 10 J = K+KS,N
          DO 10 I = K+1,N
            DO 10 KK = K,MIN(I-1,MIN(K+KS-1,N-1))
10            A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

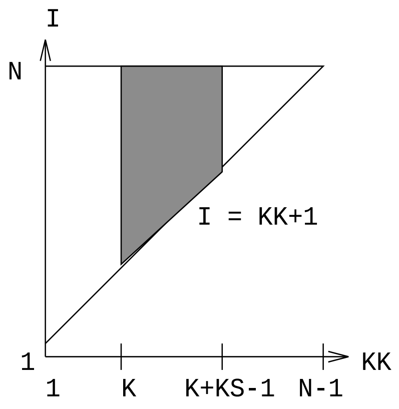


**Figure 3** Iterations Space of LU Factorization

At this point, a right–looking [10] block algorithm has been obtained. Therefore, LU factorization is blockable. The loop nest surrounding statement `10` is a matrix-matrix multiply that can be further optimized depending upon the architecture. For superscalar architectures whose performance is bound by cache, outer loop unrolling on non-rectangular loops can be applied to the `J`- and `I`-loops to further improve performance [7, 8]. For vector architectures, a different loop optimization strategy may be more beneficial [1].

Many of the transformations that we have used to obtain the block version of LU factorization are well known in the compiler community and exist in many commercial compilers (*e.g.,* HP, DEC and SGI). One of the contributions of this study to compiler research is to show how the use of section analysis is necessary for compilers to block matrix factorizations. Note that none of the aformentioned compilers uses section analysis for this purpose.

### 3.1.2 Adding Partial Pivoting

Although the compiler can discover the potential for blocking in LU decomposition without pivoting using index-set splitting, the same cannot be said when partial pivoting is added (see Figure 4 for LU decomposition with partial pivoting). In the partial pivoting algorithm, a new recurrence exists that does not fit the form handled by index-set splitting. Consider the following sections of code after applying index-set splitting to the algorithm in Figure 4.

```
    DO 10 KK = K,K+KS-1
      DO 30 J = 1,N
```

```
        TAU = A(KK,J)
 25     A(KK,J) = A(IMAX,J)
 30     A(IMAX,J) = TAU
      DO 10 J = KK+KS,N
        DO 10 I = KK+1,N
 10       A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

The reference to `A(IMAX,J)` in statement `25` and the reference to `A(I,J)` in statement `10` access the same sections. Distributing the `KK`-loop around both `J`-loops would convert the true dependence from `A(I,J)` to `A(IMAX,J)` into an antidependence in the reverse direction. The rules for the preservation of data dependence prohibit the reversing of a dependence direction. This would seem to preclude the existence of a block analogue similar to the non-pivoting case. However, a block algorithm that ignores the preventing recurrence and distributes the `KK`-loop can still be mathematically derived [13].

Consider the following. If

$$M_1 = \begin{pmatrix} 1 & 0 \\ -m_1 & I \end{pmatrix}, \quad P_2 = \begin{pmatrix} 1 & 0 \\ 0 & \hat{P}_2 \end{pmatrix}$$

then

$$P_2 M_1 = \begin{pmatrix} 1 & 0 \\ -\hat{P}_2 m_1 & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \hat{P}_2 \end{pmatrix} \equiv \hat{M}_1 P_2. \qquad (1)$$

This result shows that we can postpone the application of the eliminator $M_1$ until after the application of the permutation matrix $P_2$ if we also permute the rows of the eliminator. Extending Equation 1 to the entire formulation we have

$$U \quad = \quad M_{n-1}\hat{M}_{n-2}\hat{M}_{n-3}\cdots\hat{M}_1 P_{n-1}P_{n-2}P_{n-3}\cdots P_1 A \quad = \quad MPA.$$

In the implementation of the block algorithm, $P_i$ cannot be computed until step $i$ of the point algorithm. $P_i$ only depends upon the first $i$ columns of $A$, allowing the computation of $k$ $P_i$'s and $\hat{M}_i$'s, where $k$ is the blocking factor, and then the block application of the $\hat{M}_i$'s [13].

```
      DO 10 K = 1,N-1
C
C ... pick pivot --- IMAX
C
        DO 30 J = 1,N
          TAU = A(K,J)
 25       A(K,J) = A(IMAX,J)
 30       A(IMAX,J) = TAU
        DO 20 I = K+1,N
 20       A(I,K) = A(I,K) / A(K,K)
        DO 10 J = K+1,N
          DO 10 I = K+1,N
 10         A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

**Figure 4** LU Decomposition with Partial Pivoting

To install the above result into the compiler, we examine its implications from a data dependence viewpoint. In the point version, each row interchange is followed by a whole-column update in which each row element is updated independently. In the block version, multiple row interchanges may occur before a particular column is updated. The same computations (column updates) are performed in both the point and block versions, but these computations may occur in different locations (rows) of the array. The key concept for the compiler to understand is that row interchanges and whole-column updates are commutative operations. Data dependence alone is not sufficient to understand this. A data dependence relation maps values to memory locations. It reveals the sequence of values that pass through a particular location. In the block version of LU decomposition, the sequence of values that pass through a location is different from the point version, although the final values are identical. Without an understanding of commutative operations, LU decomposition with partial pivoting is not blockable.

Fortunately, a compiler can be equipped to understand that operations on whole columns are commutable with row permutations. To upgrade the compiler, one would have to install pattern matching to recognize both the row permutations and whole-column updates to prove that the recurrence involving statements 10 and 25 of the index-set split code could be ignored. Forms of pattern matching are already done in commercially available compilers. Vectorizing compilers pattern match for specialized computations such as searching vectors for particular conditions [28]. Other preprocessors pattern match to recognize matrix multiplication and, in turn, output a predetermined solution that is optimal for a particular machine. So, it is reasonable to believe that pivoting can be recognized and implemented in commercial compilers if its importance is emphasized.

## 3.2 Cholesky Factorization

When the matrix $A$ is symmetric and positive definite, the LU factorization may be written as

$$A = LU = LD(D^{-1}U) = LD^{1/2}D^{1/2}L^T \equiv \hat{L}\hat{L}^T,$$

where $\hat{L} = LD^{1/2}$ and $D$ is the diagonal of $U$. The decomposition of $A$ into the product of a triangular matrix and its transpose is called the Cholesky factorization. Thus we need only work with the lower triangular half of $A$ and essentially the same dependence analysis that applies to the LU factorization without pivoting may be used.

The strip mined version of the Cholesky factorization is shown below.

```
      DO 10 K = 1,N-1,KS
        DO 10 KK = K,MIN(K+KS-1,N-1)
          A(KK,KK) = SQRT( A(KK,KK) )
          DO 20 I = KK+1,N
20          A(I,KK) = A(I,KK) / A(KK,KK)
          DO 10 J = KK+1,N
            DO 10 I = J,N
10            A(I,J) = A(I,J) - A(I,KK) * A(J,KK)
```

As is the case with LU factorization, there is a recurrence between `A(I,J)` in statement 10 and `A(I,KK)` in statement 20 carried by the KK-loop. The data access patterns in Cholesky factorization are identical to LU factorization (see Figure 2), index-set splitting can be applied to the J-loop at `K+KS-1` to allow the KK-loop to be distributed, achieving the LAPACK block algorithm.

## 3.3 QR Factorization

In this section, we examine the blockability QR factorization. First, we show that the block algorithm from LAPACK is not blockable. Then, we give an alternate algorithm that is blockable.

### 3.3.1 LAPACK Version

The LAPACK point algorithm for computing the QR factorization consists of forming the sequence $A_{k+1} = V_k A_k$ for $k = 1, \ldots, n-1$. The initial matrix $A_1 = A$ has $m$ rows and $n$ columns, where for this study we assume $m = n$. The elementary reflectors $V_k = I - \tau_k v_k v_k^T$ update $A_k$ in order that the first $k$ columns of $A_{k+1}$ form an upper triangular matrix. The update is accomplished by performing the matrix vector multiplication $w_k = A^T v_k$ followed by the rank one update $A_{k+1} = A_k - \tau_k v_k w_k^T$. Efficiency of the implementation of the level 2 BLAS subroutines determines the rate at which the factorization is computed. For a more detailed discussion of the QR factorization see Golub and Van Loan [18].

The LAPACK block QR factorization is an attempt to recast the algorithm in terms of calls to level 3 BLAS [13]. If the level 3 BLAS are hand-tuned for a particular architecture, the block QR algorithm may perform significantly better than the point version on large matrix sizes (those that cause the working set to be much larger than the cache size).

Unfortunately, the block QR algorithm in LAPACK is not automatically derivable by a compiler. The block application of a number of elementary reflectors involves both computation and storage that does not exist in the original point algorithm [13]. To block a number of eliminators together, the following is computed

$$\begin{aligned} Q = &\ (I - \tau_1 v_1 v_1^T)(I - \tau_2 v_2 v_2^T) \cdots (I - \tau_{n-1} v_{n-1} v_{n-1}^T) \\ = &\ I - VTV^T. \end{aligned}$$

The compiler cannot derive $I - VTV^T$ from the original point algorithm using dependence information. To illustrate, consider a block of two elementary reflectors

$$\begin{aligned} Q = &\ (I - \tau_1 v_1 v_1^T)(I - \tau_2 v_2 v_2^T), \\ = &\ I - (v_1 v_2) \begin{pmatrix} \tau_1 & \tau_1 \tau_2 (v_1^T v_2) \\ 0 & \tau_2 \end{pmatrix} \begin{pmatrix} v_1^T \\ v_2^T \end{pmatrix}. \end{aligned}$$

The computation of the matrix

$$T = \begin{pmatrix} \tau_1 & \tau_1 \tau_2 (v_1^T v_2) \\ 0 & \tau_2 \end{pmatrix}$$

is not part of the original algorithm. Hence, the LAPACK version of block QR factorization is a different algorithm from the point version, rather than just a reshaping of the point algorithm for better performance. The compiler can reshape algorithms, but, it cannot derive new algorithms with data dependence information. In this case, the compiler would need to understand linear algebra to derive the block algorithm.

In the next section, a compiler-derivable block algorithm for QR factorization is presented. This algorithm gives comparable performance to the LAPACK version on small matrices while retaining machine independence.

### 3.3.2 Compiler-Derivable QR Factorization

Consider the application of $j$ matrices $V_k$ to $A_k$,

$$A_{k+j} = (I - \tau_{k+j-1} v_{k+j-1} v_{k+j-1}^T) \cdots (I - \tau_{k+1} v_{k+1} v_{k+1}^T)(I - \tau_k v_k v_k^T) A_k.$$

The compiler derivable algorithm, henceforth called *cd*-QR, only forms columns $k$ through $k+j-1$ of $A_{k+j}$ and then updates the remainder of matrix with the $j$ elementary reflectors. The final update of the trailing $n-k-j$ columns is "rich" in floating point operations that the compiler organizes to best suit the underlying hardware. Code optimization techniques such as strip-mine-and-interchange and unroll-and-jam are left to the compiler. The derived algorithm depends upon the *compiler* for efficiency in contrast to the LAPACK algorithm that depends on hand optimization of the BLAS.

*Cd*-QR can be obtained from the point algorithm for QR decomposition using array section analysis [7]. For reference, segments of the code for the point algorithm after strip mining of the outer loop are shown in Figure 5. To complete the transformation of the code in Figure 5 to obtain *cd*-QR, the I-loop must be distributed around the loop that surrounds the computation of $V_i$ and around the update before being interchanged with the J-loop. However, there is a recurrence between the definition and use of A(K,J) within the update section and the definition and use of A(J,I) in computation of $V_i$. The recurrence is carried by the I-loop and appears to prevent distribution.

Figure 6 shows the sections of the array A(:,:) accessed for the entire execution of the I-loop. If the sections accessed by A(J,I) and A(K,J) are examined, a legal partial distribution of the I-loop is revealed (note the similarity to LU and Cholesky factorization. The section accessed by A(J,I) (the black region) is a subset of the section accessed by A(K,J) (both the black and gray regions) and the index-set of J can be split at the point J = I+IB-1 to create a new loop that executes over the iteration space where the memory locations accessed by A(K,J) are disjoint from those accessed by A(J,I). The new loop that iterates over the disjoint region can be further optimized by the compiler depending upon the target architecture.

### 3.3.3 A Comparison of the Two QR Factorizations

The algorithm *cd*-QR does not exhibit as much cache reuse as the LAPACK version on large matrices. The reason is that the LAPACK algorithm is able to take advantage of the level 3 BLAS routine DGEMM, which can be highly optimized. *Cd*-QR uses operations that are closer to level 2 BLAS and that have worse cache reuse characteristics. Therefore, we would expect the LAPACK algorithm to perform better on larger matrices as it could possibly take advantage of a highly tuned matrix-matrix multiply kernel.

### 3.4 Summary of Transformations

In summary, Table 1 lists the analyses and transformations that must be used by a compiler to block matrix factorizations. Items 1 and 2 were discussed in Section 2. Items 3 through 7 were discussed in Section 3.1. Item 8 was discussed in the compiler literature [25, 9]. Item 9 is discussed in Section 3.1.2. Finally, it should be noted that items 2 and 9 are not likely to be found in today's commercial compilers.

## 4 Experiment

We measured the performance of each block factorization algorithm on four different architectures: the IBM POWER2 model 590, the HP model 712/80, the DEC Alpha 21164 and the SGI model Indigo2

```
                    DO II = 1, N, IB

                       DO I = II, MIN0(II+IB-1,N)
*
*                         Generate elementary reflector V_i.
*
                          DO J = I+1, M
                             A(J,I) = A(J,I)/(A(I,I)-BETA)
                          ENDDO
*
*                         Update A(i:m,i+1:n) with V_i.
*
                          DO J = I+1, N

                             T1 = ZERO
                             DO K = I, M
                                T1 = T1 + A(K,J)*A(K,I)
                             ENDDO

                             DO K = I, M
                                A(K,J) = A(K,J) - TAU(I)*T1*A(K,I)
                             ENDDO

                         ENDDO
                      ENDDO
                   ENDDO
```

**Figure 5** Strip-Mined Point QR Decomposition

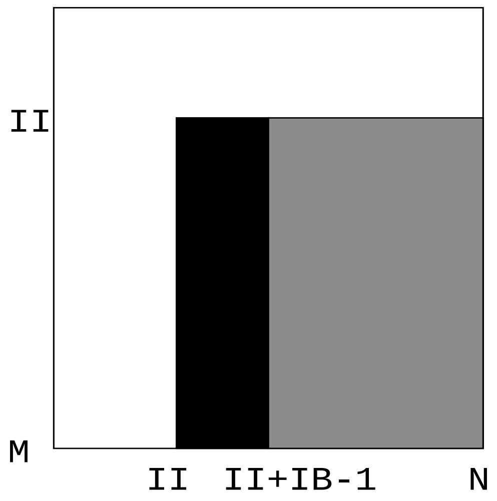


**Figure 6** Regions of `A` Accessed by QR Decomposition

**Table 1** Summary of the compiler transformations necessary to block matrix factorizations.

| | |
|---|---|
| 1 | Dependence Analysis |
| 2 | Array Section Analysis |
| 3 | Strip-Mine-and-Interchange |
| 4 | Unroll-and-Jam |
| 5 | Index-Set Splitting |
| 6 | Loop Distribution |
| 7 | Handling of Non-rectangular Iteration Spaces |
| 8 | Automatic Block-Size Selection |
| 9 | Pattern Matching for Pivoting |

with a MIPS R4400. Table 2 summarizes the characteristics of each machine. These architectures were chosen because they are representative of the typical high-performance workstation.

| Machine | Clock Speed | Peak Mflops | Cache Size | Associativity | Line Size | Compiler |
|---|---|---|---|---|---|---|
| IBM POWER2 | 66.5MHz | 264 | 256KB | 4 | 256 bytes | xlf AIX v1.3.0.24 |
| HP 712 | 80MHz | 80 | 256KB | 1 | 32 bytes | f77 v9.16 |
| DEC Alpha | 250MHz | 500 | 8KB | 1 | 32 bytes | f77 v3.8 |
| SGI Indigo2 | 200MHz | ?? | 16KB | 1 | 32 bytes | f77 v5.3 |

**Table 2** Machine Characteristics

On all the machines, we used the vendor's optimized BLAS. For example, on the IBM POWER2 and SGI Indigo2, we linked with the libraries `-lessl` and `-lblas`, respectively. Our compiler-optimized versions were obtained by hand using the algorithms in the literature. The reason that this process could not be fully automated is because of a current deficiency in the dependence analyzer of our tool [3, 5].

In each table below, performance is reported in double precision megaflops Each factorization routine is run with block sizes of 1, 2, 4, 8, 16, 24, 32, 48, and 64.[1] In each table, the columns should be interpreted as follows:

**LABlk:** The best blocking factor for the LAPACK algorithm.

**LAMf:** The best megaflop rate for the LAPACK algorithm (corresponding to LABlk).

**CBlk:** The best blocking factor for the compiler-derived algorithm.

**CMf:** The best megaflop rate for the compiler-derived algorithm (corresponding to CBlk).

[1]Although the compiler can effectively choose blocking factors automatically, we do not have an implementation of the available algorithms [25, 9].

**Table 3** LU Performance on IBM, HP, DEC and SGI

| | *IBM POWER2* | | | | | *HP 712* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** | **LABk** | **LAMf** | **CBlk** | **CMf** | **Speedup** |
| 25x25 | 1,16,32,64 | 21 | 8,16 | 44 | | 1 | 21 | 8 | 21 | 1.00 |
| 50x50 | 32 | 48 | 8,16 | 74 | | 1 | 33 | 8 | 28 | 0.83 |
| 75x75 | 16 | 81 | 16 | 95 | | 1 | 26 | 8 | 31 | 1.17 |
| 100x100 | 16 | 106 | 16 | 112 | | 1 | 25 | 8 | 31 | 1.23 |
| 150x150 | 16 | 132 | 16 | 132 | | 64 | 21 | 16 | 31 | 1.49 |
| 200x200 | 32 | 143 | 16 | 138 | | 64 | 20 | 16 | 33 | 1.63 |
| 300x300 | 32 | 157 | 32 | 147 | | 32 | 18 | 32 | 36 | 2.03 |
| 500x500 | 64 | 166 | 32 | 161 | | 32 | 17 | 32 | 40 | 2.28 |
| | *DEC Alpha* | | | | | *SGI Indigo2* | | | | |
| **Size** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** | **LABk** | **LAMf** | **CBlk** | **CMf** | **Speedup** |
| 25x25 | 1 | 43 | 8 | 53 | 1.25 | 8 | 20 | 8 | 21 | |
| 50x50 | 8 | 74 | 8 | 78 | 1.05 | 8 | 34 | 8 | 28 | |
| 75x75 | 16 | 96 | 8 | 96 | 1.00 | 8 | 34 | 8 | 29 | |
| 100x100 | 16 | 116 | 8 | 110 | 0.95 | 8 | 37 | 8 | 29 | |
| 150x150 | 32 | 138 | 8 | 113 | 0.82 | 8 | 39 | 8 | 28 | |
| 200x200 | 32 | 156 | 8 | 124 | 0.79 | 8 | 40 | 16 | 29 | |
| 300x300 | 32 | 181 | 16 | 132 | 0.73 | 8 | 41 | 16 | 30 | |
| 500x500 | 32 | 212 | 8 | 148 | 0.70 | 32 | 38 | 32 | 29 | |

## 4.1 LU Factorization

Table 3 show the performance of the compiler-derived version of LU factorization versus the LAPACK version.

For the HP 712, Table 3 indicates an unexpected trend. The compiler-derived version performs better on all matrix sizes except 50x50, with dramatic improvements as the matrix size increases. This indicates that the hand-optimized `dgemm` is not optimized well for cache performance. We have optimized for cache performance in our algorithm and when cache becomes a problem, we do much better than the hand-optimized version.

The significant performance degradation for the 50x50 case is interesting. For a matrix this small, cache performance is not a factor. We believe the performance difference comes from the way code is generated. For superscalar architectures like the HP, a code generation scheme called software pipelining is used to generate highly parallel code [24, 31]. However, software pipelining requires a lot of registers to be successful. In our code, we performed unroll-and-jam to improve cache performance. However, unroll-and-jam can significantly increase register pressure and cause software pipelining to fail [6]. On our version of LU decomposition, the HP compiler diagnostics reveal that software pipelining failed on the main computational loop due to high register pressure. Given that the hand-optimized version is highly software pipelined, the result would be a highly parallel hand-optimized loop and a not-as-parallel compiler-derived loop. At matrix size 25x25, there are not enough loop iterations to expose the difference. At matrix size 50x50, the difference is prevelant. At matrix sizes 75x75 and greater, cache performance becomes a factor. At this time, there are no known compiler algorithms that deal with the trade-offs between unroll-and-jam and software pipelining. This is an important area of future research.

For the DEC Alpha, Table 3 shows that our algorithm performs as well as or better than the LAPACK version on matrices of order 100 or less. After size 100x100, the second-level cache on the

Alpha, which is 96K, begins to overflow. Our compiler-derived version is not blocked for multiple levels of cache, while the hand-optimized version is blocked for 2 levels of cache [22]. Thus, the compiler-derived algorithm suffers many more cache misses in the level-2 cache than the LAPACK version. It is possible for the compiler to perform the extra blocking for multiple levels of cache, but we know of no compiler that currently does this. Additionally, the hand-optimized algorithm utilized the following architectural features that we do not [22]:

- The use of temporary arrays to eliminate conflicts in the level-1 direct-mapped cache and the translation lookaside buffer [25, 9].
- The use of the memory-prefetch feature on the Alpha to hide latency between cache and memory.

Although each of these optimizations could be done in the DEC product compiler, they are not. Each optimization would give additional performance to our algorithm.

## 4.2 Cholesky Factorization

Table 4 shows the performance of the compiler-derived version of Cholesky factorization versus the LAPACK version.

On the HP, we observe the same pattern on the as we did for LU factorization. When cache performance is critical, we outperform the hand-optimized version. When cache performance is not critical, the hand-optimized version give better results, except when the matrix is small. Our algorithm performed much better at 25x25 size most likely due to the high overhead associated with sofware pipelining on short loops. Since Cholesky factorization has fewer operations than LU factorization in the update portion of the code, we would expect a high overhead associated with

**Table 4** Cholesky Performance on IBM, HP, DEC and SGI

| | *IBM POWER2* | | | | | *HP 712* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** |
| 25x25 | 32,64 | 30 | 8 | 52 | | 1 | 10 | 8 | 21 | 2.00 |
| 50x50 | 64 | 60 | 8 | 84 | | 1 | 42 | 8 | 28 | 0.67 |
| 75x75 | 1 | 81 | 4 | 97 | | 1 | 37 | 8 | 31 | 0.83 |
| 100x100 | 8 | 101 | 4 | 108 | | 1 | 33 | 8 | 33 | 1.00 |
| 150x150 | 8 | 127 | 2 | 116 | | 1 | 32 | 16 | 34 | 1.05 |
| 200x200 | 8 | 144 | 8,16 | 118 | | 1 | 33 | 16 | 36 | 1.11 |
| 300x300 | 16 | 164 | 16 | 121 | | 1 | 23 | 16 | 39 | 1.72 |
| 500x500 | 16,32 | 183 | 32 | 123 | | 32 | 17 | 16 | 43 | 2.56 |
| | *DEC Alpha* | | | | | *SGI Indigo2* | | | | |
| **Size** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** |
| 25x25 | 1 | 36 | 4 | 53 | 1.50 | 1 | 19 | 4 | 23 | |
| 50x50 | 1 | 71 | 4 | 107 | 1.50 | 1 | 31 | 4 | 32 | |
| 75x75 | 1 | 94 | 4 | 117 | 1.25 | 1 | 33 | 4 | 34 | |
| 100x100 | 1 | 104 | 4 | 131 | 1.27 | 8 | 33 | 4 | 34 | |
| 150x150 | 1 | 113 | 4 | 141 | 1.24 | 16 | 36 | 4 | 34 | |
| 200x200 | 1 | 116 | 4 | 145 | 1.25 | 16 | 38 | 4 | 34 | |
| 300x300 | 64 | 134 | 4 | 146 | 1.09 | 16 | 40 | 4 | 34 | |
| 500x500 | 64 | 162 | 4 | 149 | 0.92 | 16 | 40 | 4 | 32 | |

small matrices. Also, the effects of cache are not seen until larger matrix sizes (compared to LU factorization). This is again due to the smaller update portion of the factorization.

On the DEC, we outperform the hand-optimized version up until the 500x500 matrix. This is the same pattern as seen in LU factorization except that it takes longer to appear. This is due to the smaller size of the update portion of the factorization.

## 4.3 QR Factorization

Table 5 shows the performance of the compiler-derived version of QR factorization versus the LAPACK version. Since the compiler-derived algorithm for block QR factorization has worse cache performance than the LAPACK algorithm, but $O(n^2)$ less computation, we would expect worse performance when the cache performance becomes critical.

On the HP, we see the same pattern as before. However, since the cache performance of our algorithm is not as good as the the LAPACK version, we see a much smaller improvement when our algorithm has superior performance. Again, we also see that when the cache performance of the hand-optimized algorithm is good, it outperforms our algorithm.

On the DEC, we see the same pattern as on the previous factorizations except that our degradations are much larger for large matrices. This is due to the inferior cache performance of *cd*-QR.

## 4.4 Performance Summary

NEED TO WRITE THIS WHEN ALL DATA IS PRESENT.

**Table 5** QR Performance on IBM and HP

| | *IBM POWER2* | | | | | *HP 712* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Size** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** |
| 25x25 | 32,64 | 30 | 8 | 52 | | 1 | 21 | 1 | 21 | 1.00 |
| 50x50 | 32 | 53 | 16 | 81 | | 1 | 37 | 2 | 28 | 0.75 |
| 75x75 | 32 | 81 | 8 | 114 | | 1 | 38 | 4 | 29 | 0.76 |
| 100x100 | 32 | 105 | 8 | 132 | | 1 | 38 | 4 | 30 | 0.80 |
| 150x150 | 32 | 136 | 16 | 151 | | 1 | 28 | 8 | 29 | 1.07 |
| 200x200 | 32 | 154 | 8 | 158 | | 64 | 25 | 16 | 31 | 1.23 |
| 300x300 | 32 | 185 | 32 | 170 | | 32 | 25 | 32 | 31 | 1.25 |
| 500x500 | 64 | 205 | 32 | 191 | | 32 | 23 | 32 | 31 | 1.38 |
| | *DEC Alpha* | | | | | *SGI Indigo2* | | | | |
| **Size** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** | **LABlk** | **LAMf** | **CBlk** | **CMf** | **Speedup** |
| 25x25 | 1 | 50 | 4 | 66 | 1.31 | 1 | 15 | 8 | 23 | |
| 50x50 | 1 | 85 | 2 | 98 | 1.15 | 4 | 26 | 8 | 30 | |
| 75x75 | 1 | 100 | 2 | 107 | 1.07 | 8 | 29 | 8 | 29 | |
| 100x100 | 16 | 114 | 4 | 111 | 0.98 | 8 | 34 | 8 | 29 | |
| 150x150 | 16 | 138 | 8 | 110 | 0.79 | 8 | 38 | 8 | 28 | |
| 200x200 | 16 | 158 | 16 | 115 | 0.72 | 8 | 39 | 8 | 27 | |
| 300x300 | 16 | 180 | 16 | 114 | 0.64 | 8 | 40 | 8 | 25 | |
| 500x500 | 32 | 213 | 16 | 115 | 0.54 | 8 | 39 | 8 | 25 | |

## 5 Blocking with a GEMM based Approach

Since LAPACK depends upon a set of highly tuned set of BLAS for efficiency, there remains the practical question of how they should be optimized. As discussed in the introduction, an efficient set of BLAS requires a non-trivial effort in software engineering. See [20] for a discussion on software efforts to provide optimal implementations of the level 3 BLAS.

An approach that is both efficient and practical is the GEMM-based one proposed by Kågström, Ling and Van Loan [20] in a recent study. Their approach advocates optimizing the general matrix-matrix multiply and add kernel `_GEMM` and then rewriting the remainder of the level 3 BLAS in terms of calls to this kernel. Their thorough analysis highlights the many issues that must be considered when attempting to construct a set of highly tuned BLAS. Most importantly, they provide high quality implementations of the BLAS for general use as well as a performance evaluation benchmark [21].

We emphasize that our study examines only whether the necessary optimizations may be left to the compiler, and, also whether they should be applied directly to the matrix factorizations. What is beyond the ability of the compiler is that of recasting the level 3 BLAS in terms of calls to `_GEMM`.

## 6 Summary

We have set out to determine whether a compiler can automatically restructure matrix factorizations well enough to avoid the need for hand optimization. To that end, we have examined a collection of implementations from LAPACK. For each of these programs, we determined whether a plausible compiler technology could succeed in obtaining the block version from the point algorithm.

The results of this study are encouraging: we have demonstrated that there exist implementable compiler methods that can automatically block matrix factorization codes to achieve algorithms that are competitive with those of LAPACK. Our results show that for modest-sized matrices on advanced microprocessors with a memory hierarchy, the compiler-derived variants are often superior. These matrix sizes are typical on workstations. We remark that a different set of optimizations is required to optimize the codes for vector machines [1]. However, our use of section analysis is still required to enable these transformations.

Given that future machine designs are certain to have increasingly complex memory hierarchies, compilers will need to adopt increasingly sophisticated memory-management strategies so that programmers can remain free to concentrate on program logic. Given the potential for performance attainable with automatic techniques, we believe that it is possible for the user to express machine-independent point matrix factorization algorithms without the BLAS and still get good performance if compilers adopt our enhancement to already existing methods.

## Acknowledgments

Ken Kennedy and Richard Hanson provided the original motivation for this work. Ken Kennedy, Keith Cooper and Danny Sorensen provided financial support for this research when it was begun at Rice University. We also wish to thank Tomas Lofgren and John Pieper of DEC for their help with obtaining the DXML libraries and diagnosing the compiler's performance, respectively.

## References

[1] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide.* SIAM, Philadelphia, PA, second edition, 1995.

[3] D. Callahan, K. Cooper, R. Hood, K. Kennedy, and L. Torczon. ParaScope: A parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*, Athens, Greece, June 1987.

[4] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing.* Springer-Verlag, Athens, Greece, 1987.

[5] S. Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, Department of Computer Science, September 1992.

[6] S. Carr, C. Ding, and P. Sweany. Improving software pipelining with unroll-and-jam. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, Maui, HI, January 1996.

[7] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing '92*, pages 114–124, Minneapolis, MN, November 1992.

[8] Steve Carr and Ken Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[9] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization. *SIGPLAN Notices*, 30(6):279–280, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*

[10] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. Van der Vorst. *Solving Linear systems on Vector and shared memory computers.* SIAM, Philadelphia, PA., 1991.

[11] J.J. Dongarra, J. DuCroz, I. S. Duff, and S. Hammarling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.

[12] J.J. Dongarra, J. DuCroz, S. Hammarling, and R. J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. on Math. Software*, 14(1):1–17, 1988.

[13] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared-Memory Computers.* SIAM, Philadelphia, 1991.

[14] J.J. Dongarra, F.G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

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

[16] K.A. Gallivan, R.J. Plemmons, and A.H. Sameh. Parllel algorithms for dense linear algebra computations. *SIAM Review*, 32:54–135, 1990.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins, Baltimore, second edition, 1989.

[18] G.H. Golub and C.F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, 1989.

[19] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[20] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. Technical Report UMINF-95.18, Department of Computing Science, University of Umeå, S-901 87 Umeå, Sweden, October 1995. Submitted to ACM Transactions on Mathematical Software. Also available as LAPACK Working Note 107.

[21] B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: Installation, tuning, and use of the model implementations and the performance evaluation benchmark. Technical Report UMINF-95.19, Department of Computing Science, University of Umeå, S-901 87 Umeå, Sweden, October 1995. Submitted to ACM Transactions on Mathematical Software. Also available as LAPACK Working Note 108.

[22] C. Kamath, R. Ho, and D.P. Manley. DXML: A high-performance scientific subroutine library. *Digital Technical Journal*, 6(3):44–56, 1994.

[23] D. Kuck. *The Structure of Computers and Computations Volume 1.* John Wiley and Sons, New York, 1978.

[24] Monica Lam. Software pipelining: An effective scheduling technique for VLIW machines. *SIGPLAN Notices*, 23(7):318–328, July 1988. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation.*

[25] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 63–74, Santa Clara, California, 1991.

[26] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5:308–329, 1979.

[27] Richard Lehoucq. Implementing efficient and portable dense matrix factorizations. In *Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing*, 1992.

[28] David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17:1223–1244, 1991.

[29] James M. Ortega. *Introduction to Parallel and Vector Solutions of Linear Systems.* Plenum Press, New York, New York, 1988.

[30] A.K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989.

[31] B. R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelined loops. *SIGPLAN Notices*, 27(7):283–299, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[32] B. T. Smith, J. M. Boyle, J. J. Dongarra B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *EISPACK Guide*. Springer–Verlag, Berlin, second edition, 1976. Volume 6 of Lecture Notes in Computer Science.

[33] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[34] M. Wolfe. Advanced loop interchange. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[35] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.