# Compiler Blockability of Dense Matrix Factorizations

*Steve Carr*
*Richard Lehoucq*

**CRPC-TR95557-S**
**August 1995**

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Also available as *TR95-08* from the Department of Computer Science, Michigan Technological University.

# Compiler Blockability of Dense Matrix Factorizations*

Steve Carr† R. B. Lehoucq‡

August 9, 1995

**Abstract**

Recent architectural advances have made memory accesses a significant bottleneck for computational problems. Even though cache memory helps in some cases, it fails to alleviate the bottleneck for problems with large working sets. As a result, scientists are forced to restructure their codes by hand to reduce the working-set size to fit a particular machine. Unfortunately, these hand optimizations create machine-specific code that is not portable across multiple architectures without a significant loss in performance or a significant effort to re-optimize the code.

It is the thesis of this paper that most of the hand optimizations performed on matrix factorization codes are unnecessary because they can and should be performed by the compiler. 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.

In this paper, we show that Cholesky and LU factorizations may be optimized automatically by the 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 matrix sizes that are typically run on nodes of massively parallel systems. Our approach allows us to conclude that matrix factorizations can be expressed in a machine-independent form with the expectation of good memory performance across a variety of architectures.

## 1 Introduction

Over the past decade, the computer industry has realized dramatic improvements in the power of processors. These gains have been achieved both by cycle-time improvements and by architectural innovations like multiple processors, multiple-instruction issue and pipelined functional units. As a result of these architecture improvements, today's machines can perform many more operations per cycle than their predecessors. Unfortunately, memory speeds have not kept pace. Therefore, with these gains in computational power has come an increase in the number of cycles for a memory access—a latency of 10 to 20 machine cycles is now quite common [20, 30].

Because the latency and bandwidth of memory systems have not kept pace with processor speed, computations are often delayed waiting for data from memory. As a result, processors see idle computational cycles more frequently. In fact, memory delays have become a principal

*Research supported by NSF Grant CCR-9120008 and by NSF grant CCR-9409341

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

‡Department of Computational and Applied Mathematics, Rice University, Houston TX 77251-1892, *lehoucq@rice.edu.*

performance bottleneck for high-performance computers. Although cache helps to ameliorate these problems, it performs poorly on scientific calculations with working sets larger than the cache size.

To overcome these performance problems, programmers have learned to employ a coding style that achieves a good balance between memory references and floating-point operations and improves program locality. This is usually done by unrolling outer loops and merging the resulting copies of inner loops and by restructuring the code so that it operates on blocks of data that can fit in cache. Unfortunately, this coding method almost always leads to ugly code that is dependent upon a single machine architecture. More importantly, this method is error-prone, expensive, tedious and time consuming.

There is a long history of the use of sophisticated compiler optimizations to achieve machine independence. The Fortran I compiler included enough optimizations to make it possible for scientists to abandon machine-language programming. More recently, advanced vectorization technology has made it possible to write machine-independent vector programs in a sub-language of Fortran 77. We contend that it will be possible to achieve the same success for memory-hierarchy management on scalar processors. More precisely, enhanced compiler technology will enable programmers to express an algorithm in a natural, machine-independent form and achieve memory-hierarchy performance good enough to obviate the need for hand optimization.

To investigate the viability of memory-hierarchy management by the compiler, experiments were undertaken to determine if a compiler could automatically generate the block algorithms in LAPACK from the corresponding point algorithms expressed in Fortran 77 [7, 8, 14, 25]. This study addressed the following question: *what information does a compiler need in order to derive block versions of matrix factorization codes that are competitive with the best hand-blocked versions*?

This study has yielded two major results. The first, which is detailed in another paper [9], reveals that the hand loop unrolling performed when optimizing the level 3 BLAS [12] subroutines is rarely necessary. While the BLAS are useful, the hand optimization that is required to obtain good performance on a particular architecture should be left to the compiler. Experiments show that the compiler can automatically unroll loops as effectively as hand optimization in most cases. The second result, which we discuss in this paper, reveals an algorithmic approach that can be used to analyze and block matrix factorization algorithms automatically in the compiler. Our results with this algorithmic approach 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 feel this is important since the architectural trend is toward the amalgamation of many smaller processors.

We begin our presentation with a review of background material related to memory optimization. Next, we present the transformations that are necessary to block matrix factorization codes automatically. Then, we describe a study of the application of these transformations to derive the block algorithms in LAPACK from their corresponding point algorithms. Finally, we present an experiment comparing the performance of hand-optimized LAPACK algorithms with the compiler-derived algorithms attained using our techniques.

# 2 Background

## 2.1 Memory Hierarchy

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 can be 10's to 100's 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 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.

Because of the memory hierarchy and shared-memory multiprocessors, the LAPACK and the level 2 and 3 BLAS and BLAS projects became necessary. The projects restructured basic linear algebra algorithms in order to provide portable software that runs efficiently in the presence of a memory hierarchy [14, 2, 13].

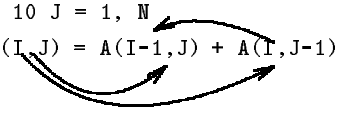
## 2.2 Dependence

The fundamental tool available to the compiler is that of *dependence*—the same tool used in vectorization and parallelization. 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 [21].

- 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 [5, 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.

### 2.3 Cache Reuse

When applied to memory-hierarchy management, a dependence can be thought of as an opportunity for reuse. There are two types of reuse: *temporal* and *spatial*. Temporal reuse occurs when a reference in a loop accesses data that has previously been accessed in the current or a previous iteration of a loop. Spatial reuse occurs when a reference accesses data that is in the same cache line as some previous access. In the following loop,

```
    DO 10 I = 1,N
10    A(I) = A(I-5) + B(I)
```

the reference to `A(I-5)` has temporal reuse of the value defined by `A(I)` 5 iterations earlier. The reference to `B(I)` has spatial reuse since consecutive elements of `B` will likely be in the same cache line.

### 2.4 Iteration-Space Blocking

To improve the memory behavior of loops that access more data than fit in cache, the iteration space of a loop can be grouped into blocks whose working sets are small enough for cache to capture the available temporal reuse. Strip-mine-and-interchange is a transformation that achieves this result [34, 28, 31]. It shortens the distance between the source and sink of a dependence so that it is more likely for the datum to reside in cache when the reuse occurs. Consider the following loop nest.

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

Assume the value of `M` is much greater than the size of the cache. Temporal reuse exists for `B`, but not for `A`. To exploit `A`'s temporal reuse, strip-mine-and-interchange is applied to the `J`-loop as shown below.

```
      DO 10 J = 1,N,JS
        DO 10 I = 1,M
          DO 10 JJ = J, MIN(J+JS-1,N)
10          A(I) = A(I) + B(JJ)
```

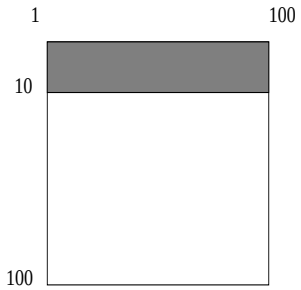



FIGURE 1: Section of A

Temporal reuse of A now occurs. In addition, the temporal reuse of JS values of B out of cache occurs for every iteration of the J-loop if JS is less than the size of the cache and there is no interference [23].

A transformation analogous to strip-mine-and-interchange is unroll-and-jam [3]. Unroll-and-jam is used for register blocking instead of cache blocking and can be seen as an application of strip mining, loop interchange and loop unrolling. Essentially, the inner loop is completely unrolled after strip-mine-and-interchange to effect unroll-and-jam. When JS doesn't divide N, a pre-loop is used to handle the extra iterations instead of a MIN function. In the previous example, if JS were equal to 3 and N were divisible by 3, then the following code would result from unroll-and-jam.

```
      DO 10 J = 1,N,3
        DO 10 I = 1,M
          A(I) = A(I) + B(J)
          A(I) = A(I) + B(J+1)
10        A(I) = A(I) + B(J+2)
```

# 3 Transformation of Complex Loop Forms

For matrix factorization problems, iteration-space blocking cannot be directly applied as shown in the previous section. As will be shown in Section 4, the shape of the loop body or safety constraints may permit only a partial application of blocking. In these cases, a transformation called *index-set splitting* can be applied. In the rest of this section, we present index-set splitting and its use in optimizing complex loop nests. Each use of index-set splitting discussed below is applicable to the transformation of matrix factorization code.

## 3.1 Index-Set Splitting

Index-set splitting creates multiple loops from one original loop with each new loop iterating over nonintersecting portions of the original iteration space. Execution order is unchanged and the original iteration space is still completely executed. As an example of index-set splitting, consider the following loop.

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

The index set of I can be split at iteration 100 to obtain

```
      DO 10 I = 1,MIN(N,100)
10      A(I) = A(I) + B(I)
      DO 20 I = MAX(1,MIN(N,100)+1),N
20      A(I) = A(I) + B(I)
```

Although this transformation does nothing by itself, its application can enable the blocking of complex loop forms. This section uses index-set splitting to enable the blocking of triangular and trapezoidal iteration spaces and loops with complex dependence patterns.

## 3.2 Triangular Iteration Spaces

If the iteration space of a loop is not rectangular, iteration-space blocking cannot be directly applied. Interchanging loops that iterate over a triangular regions requires the modification of the loop bounds to preserve the semantics of the loop [33, 34]. Therefore, blocking triangular

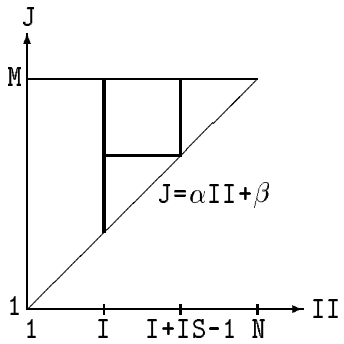



FIGURE 2: Upper Left Triangular Iteration Space

regions also requires loop bound modification. Below, we derive the formula for determining loop bounds when blocking is performed on triangular iteration spaces. We begin with the derivation for strip-mine-and-interchange and then extend it to unroll-and-jam.

The general form of one type of strip-mined triangular loop is given below, where $\alpha$ and $\beta$ are literal integer constants ($\beta$ may be symbolic) and $\alpha > 0$.

```
    DO 10 I = 1,N,IS
      DO 10 II = I,I+IS-1
        DO 10 J = αII+β,M
10        loop body
```

Figure 2 gives a graphical description of the iteration space of this loop. To interchange the `II` and `J` loops, the intersection of the line `J=`$\alpha$`II+`$\beta$ with the iteration space at the point `(I,`$\alpha$`I+`$\beta$`)` must be handled. Therefore, interchanging the loops requires the `II`-loop to iterate over a trapezoidal region with an upper bound of $\frac{J-\beta}{\alpha}$ until $\frac{(J-\beta)}{\alpha} >$ `I+IS-1`. This gives the following loop nest.

```
    DO 10 I = 1,N,IS
      DO 10 J = αI+β,M
        DO 10 II = I,MIN((J-β)/α,I+IS-1)
10        loop body
```

This formula can be trivially extended to handle the cases where $\alpha < 0$ and where a linear function of `I` appears in the upper bound instead of the lower bound [6].

Triangular strip-mine-and-interchange can be extended to triangular unroll-and-jam as follows. Since the iteration space defined by the two inner loops is a trapezoidal region, the number of iterations of the innermost loop vary with `J`, making unrolling more difficult. Index-set splitting of `J` at $\alpha$`(I+IS-1)+`$\beta$ creates one loop that iterates over the triangular region below the line `J=`$\alpha$`(I+IS-1)+`$\beta$ and one loop that iterates over the rectangular region above the line. Since the length of the rectangular region is known, it can be unrolled to give the following loop nest.

```
    DO 10 I = 1,N,IS
      DO 20 II = I,I+IS-2
        DO 20 J = αII+β,MIN(α(I+IS-2)+β,M)
20        loop body
      DO 10 J = α(I+IS-1)+β,M
10      unrolled loop body
```

Depending upon the values of $\alpha$ and $\beta$, it may also be possible to determine the size of the triangular region; therefore, it may be possible to completely unroll the first loop nest to eliminate the overhead. Additionally, triangular unroll-and-jam can be extended to handle other common triangles [6].

### 3.3 Trapezoidal Iteration Spaces

While the previous method applies to many of the common non-rectangular-shaped iteration spaces, there are still some important loops that it will not handle. In linear algebra, seismic and partial differential equation codes, loops with trapezoidal-shaped iteration spaces occur. Consider the following example, where L is assumed to be a constant, and $\alpha > 0$.

```
   DO 10 I = 1,N
     DO 10 J = L,MIN(αI+β,N)
10     loop body
```

The MIN function defines one rectangular region and one triangular region separated at the point where $\alpha$I+$\beta$ = N. Because rectangular and triangular regions can be handled already, the index set of I can be split into two separate regions at the point $\mathtt{I} = \frac{\mathtt{N}-\beta}{\alpha}$ with blocking applied to each new loop separately. Splitting gives the following loop nests that can be blocked.

```
   DO 10 I = 1,MIN(N,(N-β)/α)
     DO 10 J = L,αI+β
10     loop body
   DO 20 I = MAX(1,MIN(N,(N-β)/α)+1),N
     DO 20 J = L,N
20     loop body
```

The lower bound, L, of the inner loop in a trapezoidal nest need not be a constant value. It may be any function that, after index-set splitting, produces an iteration space that can be blocked.

### 3.4 Complex Dependence Patterns

In some cases, it is not only the shape of the iteration space that presents difficulties for the compiler but also the dependence patterns within the loop. Consider the strip mined example below.

```
    DO 10 I = 1,N,IS
      DO 10 II = I, I+IS-1
        T(II) = A(II)
        DO 10 K = II,N
10        A(K) = A(K) + T(II)
```

To complete iteration-space blocking, the II-loop must be interchanged to the innermost position. However, there is a recurrence between the definition of A(K) and the use of A(II) carried by the II-loop, preventing interchange with distribution. Standard dependence abstractions, such as distance or direction vectors, report that the recurrence exists for every value defined by A(K) [32]. This means blocking is prevented. However, analyzing the sections of the arrays that are accessed at the source and sink of the backward true dependence reveals that there is potential to apply blocking. Consider Figure 3. The section of the array A read by the reference to A(II) goes from I to I+IS-1 and the section written by A(K) goes from I to N. Therefore, the recurrence does not exist for the section from I+IS to N.

To allow partial blocking of the loop, the index set of K can be split so that one loop iterates over the iteration space where A(K) and A(II) access common memory locations and one loop

iterates over the iteration space where they access disjoint locations. To determine the split point that creates these loops, the subscript expression that defines the larger section is set equal to the boundary between the sections accessed by the source and sink of the dependence and the equation is solved for the inner induction variable. In the above example, let `K = I+IS-1` and solve for `K`. Splitting at this point yields

```
      DO 10 I = 1,N,IS
        DO 10 II = I,I+IS-1
          T(II) = A(II)
          DO 20 K = I,I+IS-1
20          A(K) = A(K) + T(II)
          DO 10 K = I+IS,N
10          A(K) = A(K) + T(II)
```

The `II`-loop can now be distributed around statements `10` and `20` and blocking can be completed on the loop nest surrounding statement `10`.

The method just described may be applicable when the references involved in the preventing dependences have different induction variables in corresponding subscript positions (e.g., `A(II)` and `A(K)` in the previous example). The effectiveness of index-set splitting depends upon the representation of sections. The precision must be enough to relate the locations in the array to index variable values. The representation that we have chosen is equivalent to Fortran 90 array notation [19].

# 4 Automatic Blocking of Dense Matrix Factorizations

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 supercomputers, the efficiency of the factorizations were seen to depend dramatically upon the algorithmic form chosen for the implementation. Dongarra, Gustavson and Karp [15] gave a detailed study the algorithmic issues involved in constructing an efficient LU factorization on the early CRAY supercomputers. The work of Ortega [27], and Gallivan, Plemmons and Sameh [17] considered both algorithmic and computational issues involved in the efficient implementation of matrix factorizations on vector and parallel computers. The single most important factor governing the efficiency of a software implementation in computing a factorization is : Managing the memory hierarchy.

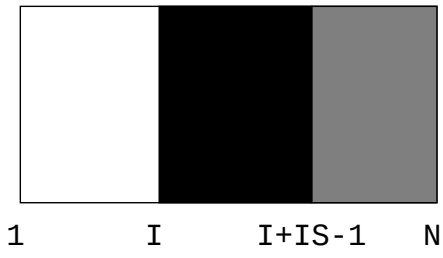



FIGURE 3: Data Space for `A`

Part of the motivation of the LAPACK [2] project was to recast the matrix factorization algorithms in LINPACK [11] with *block* ones. A block form of a factorization restructures the algorithm in terms of matrix operations that minimize the amount of data moved within the memory hierarchy while keeping the arithmetic units of the machine occupied. LAPACK blocks matrix factorizations by restructuring the algorithms to use the level 2 and 3 BLAS [13, 12]. The motivation for the BLAS [24](Basic Linear Algebra Subprograms) was to provide a set of commonly used vector operations such as vector addition and dot product so that the programmer could invoke the subprograms instead of writing the code directly. The responsibility for optimizing these "level 1" operations was left up to the machine vendor or some other interested party. The level 2 and 3 BLAS followed with matrix-vector and matrix-matrix operations 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. The reader is referred to the work of Dongarra, Duff, Sorensen and Van der Vorst [14] for further information regarding the BLAS and their use in deriving block matrix factorizations. As with the level 1 BLAS, responsibility for optimizing the higher level BLAS was left to others. Unfortunately, the cost of developing optimized versions in programming effort is significant. In contrast, we believe that programmers should express algorithms in a machine-independent form with the compiler handling the machine-specific optimization details.

To investigate whether compiler technology can make it possible to express matrix factorization in a machine-independent form, this section examines the block-ability of the three factorization algorithms using the techniques developed in Section 3. An algorithm is determined to be "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. Our study shows algorithms that perform a triangularization of a matrix are amenable to the technique described in Section 3. In particular, 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. The block algorithm variant performs well on matrix sizes typically run on workstations and nodes of massively parallel systems.

## 4.1 LU Factorization

The LU decomposition factors a non-singular matrix $A$ into the product of two matrices, $L$ and $U$, such that $A = LU$ [29]. $L$ 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 $L^{-1} = M_{n-1}P_{n-1} \cdots M_1 P_1$ and $U = L^{-1}A$. The pivot matrices are used to make the LU factorization a stable process.

In this section, we 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.

### 4.1.1 No Pivoting

Since the point algorithm for LU factorization exhibits poor cache performance on large matrices, scientists have developed a block algorithm that essentially groups a number of updates to the matrix $A$ together and applies them all at once to a block portion of the array [14]. Consider the strip-mined version of the point algorithm 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)
```

To complete the blocking of strip-mined LU factorization, 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. However, there is a recurrence between A(I,KK) in statement 20 and A(I,J) in statement 10 carried by the KK-loop that prevents distribution unless index-set splitting is performed.

Figure 4 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, the index-set of J can be split at the point J = K+KS-1 to create a new loop that executes over the iteration space where the memory locations accessed by A(I,J) are disjoint from those accessed by A(I,KK) in statement 20. This loop is shown below.

```
    DO 10 KK = K,K+KS-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)
```

At this point, the best block algorithm has been obtained. Therefore, LU factorization is blockable. The loop nest above is matrix-matrix multiply that can be further optimized depending upon the architecture. For superscalar architectures whose performance is bound by cache, loop interchange can be used to put the KK-loop in the innermost position and unroll-and-jam can be applied to the J- and I-loops to further improve performance [26, 9]. The techniques described in Section 3 for handling triangular and trapezoidal loops are necessary to optimize the above matrix-matrix multiply For vector architectures, a different loop optimization strategy may be more beneficial [1].

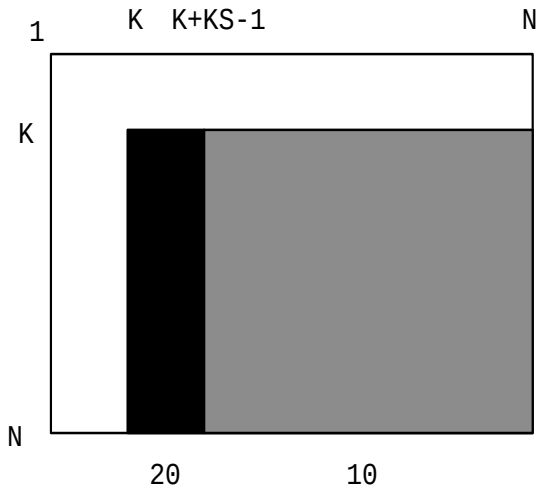



FIGURE 4: Sections of A in LU Factorization

### 4.1.2 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 5 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 5.

```
    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 [14].

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

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

```
        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 5: LU Decomposition with Partial Pivoting

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 [14].

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, so it is reasonable to believe the situation in LU decomposition can be recognized.

## 4.2 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 4), 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.

## 4.3 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.

### 4.3.1 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 $m \geq n$. 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 [14]. 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). However, the optimized BLAS codes are often not portable and the application programmer must rely upon the machine vendor to make the software investment to optimize the kernels for the architecture. It would be better to express an algorithm in a machine-independent form with the compiler handling the machine-dependent optimizations.

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 [14]. 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.

### 4.3.2 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 [8]. For reference, segments of the code for the point algorithm after strip mining of the outer loop are shown in Figure 6. To complete the transformation of the code in Figure 6 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 7 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.

# 5 Experiment

We measured the performance of each block factorization algorithm on on three different architectures: the IBM RS/6000 model 530, the HP PA-RISC model 712/80 and the SGI Indigo2 with a MIPS R4400. The RS/6000 has a 32KB 4-way set-associative cache with 128 byte lines. The PA-RISC has a 256K direct-mapped cache with 32-byte lines. Finally the MIPS has a 16K direct-mapped first-level cache with 32-byte lines. These architectures were chosen because they are representative of the typical high-performance workstation and because the chips are used as building blocks for massively parallel architectures.

On the RS/6000, we used the hand optimized level 3 BLAS subroutine `dgemm` [16] obtained from Netlib. For the HP and SGI we used the optimized BLAS distributed with the machine. 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 [4, 6]. All software was compiled with full optimization on all the machines. On the RS/6000, version 2.2 of the *xlf* compiler was used. On the HP, version 9.16 of the f77 compiler was used, and on the SGI, version 5.3 of the f77 compiler was used.

In each figure below, performance is reported in *normalized execution time* where the hand-optimized LAPACK algorithm is the base performance and the compiler-derived version is reported as a percentage of the LAPACK algorithm. All operations are performed on double-precision

```
                  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 6: Strip-Mined Point QR Decomposition

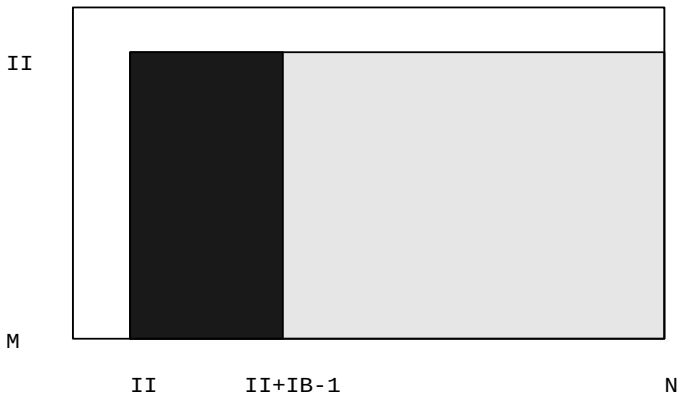



FIGURE 7: Regions of A Accessed by QR Decomposition

floating-point numbers. Each factorization routine was run with block sizes of 1, 2, 4, 8, 16, 24, 32, 48, and 64. The performance using the best block factor of this set for each matrix size is displayed in the figures. Although the compiler can effectively choose blocking factors automatically, we did not use the available algorithms [22, 10].

### 5.1 LU Factorization

Figures 8 and 9 show the performance of the compiler-derived version of LU factorization versus the LAPACK version. The results for the RS/6000 show that our compiler-derived version outperforms the hand-optimized LAPACK code for matrices of size less than 150x150. Once the cache becomes too small for the working set, both algorithms perform approximately the same. For the HP, the two versions performed equivalently for small matrices, but for larger matrices the compiler-derived version performed significantly better. This result reinforces the notion that hand-optimization is tedious and may not always yield optimal results. It is better for the compiler to perform the optimization.

On the MIPS R4400, we see the performance of our algorithm consistently 10% worse than the LAPACK version (see Figure 10). The main reason for the performance difference is that the R4400 has a 16K cache. This small cache made it significantly more difficult to obtain good performance.

### 5.2 Cholesky Factorization

Figures 11, 12 and 13 show the performance of the compiler-derived version of Cholesky factorization versus the LAPACK version. The relative performance on both the HP and IBM mimics that of LU factorization. This is because the codes are similar and because the LAPACK algorithm can be derived by the compiler in both cases. The performance on the R4400 in Figure 13 was good for small matrices, but degraded fast on larger matrices. This was due to the small 16K cache on the chip.

### 5.3 QR Factorization

Figures 14, 15 and 16 show 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 became critical. On the RS/6000, this point occurred at the 150x150 matrix size. However, we were still within a reasonable percentage of the LAPACK algorithm until size 300x300. Given that the compiler works with no algorithm-specific knowledge, unlike the hand-coded LAPACK algorithm, the fact that a compiler can equal or even beat hand optimization is significant.

The performance that we attained on the PA-RISC was quite unexpected. Here, our compiler-derived algorithm increasingly outperforms the LAPACK version as matrix sizes increase. This is in spite of the inferior cache behavior of the compiler-derived algorithm. This phenomena can only be explained by poorly optimized BLAS routines on the HP.

For the MIPS R4400, we could not get our code to run using the "-mips2" flag. So, the numbers represent code generated for a MIPS R3000. As can be seen in Figure 16, the small cache on the MIPS limits the effectiveness of our algorithm to matrices of size less than 100x100.

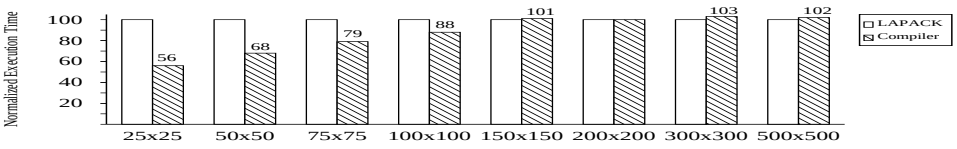



FIGURE 8: LU Factorization on the IBM RS/6000

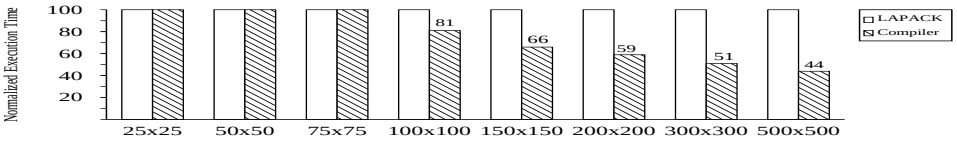



FIGURE 9: LU Factorization on the HP PA-RISC

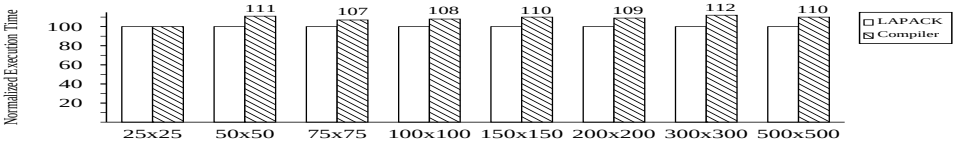



FIGURE 10: LU Factorization on MIPS R4400

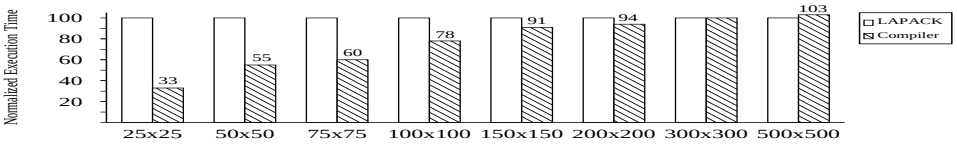



FIGURE 11: Cholesky Factorization on the IBM RS/6000

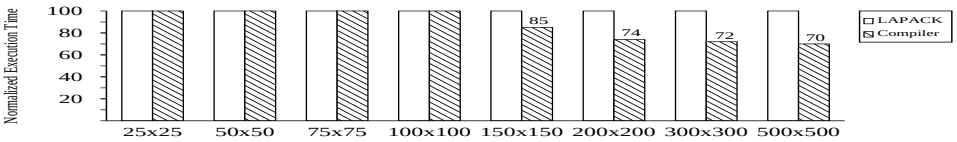



FIGURE 12: Cholesky Factorization on the HP PA-RISC

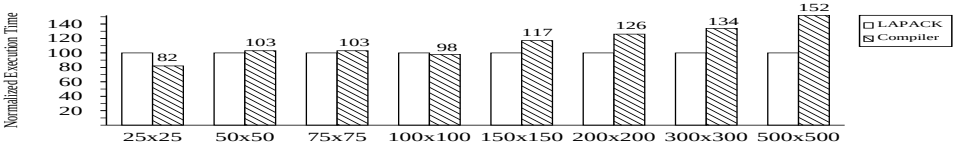



FIGURE 13: Cholesky Factorization on MIPS R4400

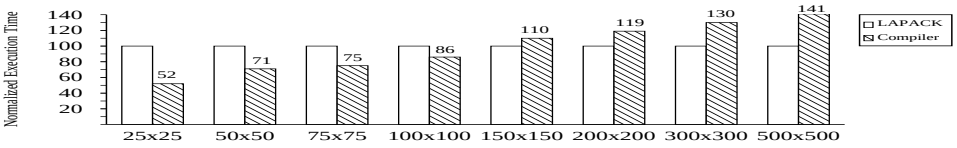



FIGURE 14: QR Factorization on the IBM RS/6000

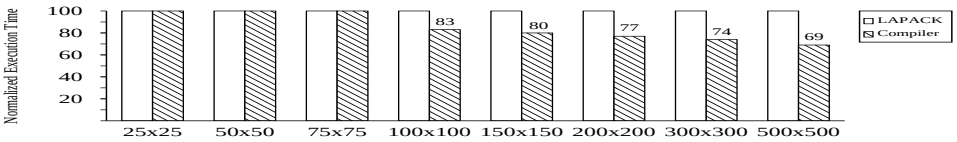



FIGURE 15: QR Factorization on the HP PA-RISC

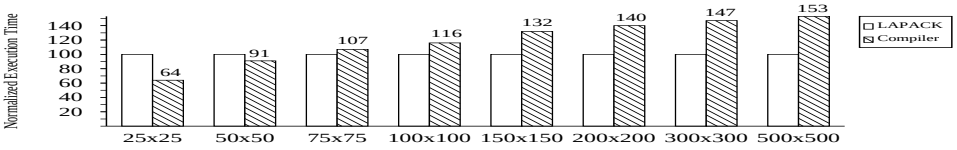



FIGURE 16: QR Factorization on MIPS R4400

### 5.4 Performance Summary

The results of this study show that compiler technology obviates the need for hand-optimized BLAS for smaller problems that are typically run on high-performance workstations. This is even more significant when we consider the fact that the microprocessors used in this study are often building blocks for massively parallel architectures. With smaller matrix sizes being processed on each individual node of a massively parallel system, the compiler technology presented in this paper can have a significant effect on performance.

## 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 readily 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 and nodes of massively parallel systems. We remark that this strategy does not necessarily translate to good performance on vector processors. Although compiler technology exists to generate efficient code on vector processors [1], we did not use this technology.

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 performance attainable with completely automatic techniques, we believe that it is possible for the user to express machine-independent point matrix factorization algorithms naturally with the expectation of good memory hierarchy performance across a variety of architectures.

## 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. Robert Reynolds and Phil Sweany gave helpful suggestions during the preparation of this document.

## 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. Dongara, J. Du Croz, A. Geenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK User's Guide.* SIAM, Philadelphia, Pennsylvania, 1992.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[4] 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.

[5] 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.

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

[7] S. Carr and K. Kennedy. Blocking linear algebra codes for memory hierarchies. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

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

[9] 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.

[10] 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.*

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

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

[13] J.J. Dongarra, J. DuCroz, S. Hammerling, and R. Hanson. An extendend set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[14] 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.

[15] 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.

[16] J.J. Dongarra, P. Mayes, and G. Radicati. The IBM RISC system/6000 and linear algebra operations. Technical Report CS-90-122, Department of Computer Science, University of Tennessee, 1990. LAPACK Working Note 28.

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

[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] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kauffman, San Mateo, California, 1990.

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

[22] 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.

[23] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architecural Support for Programming Languages and Operating Systems*, April 1991.

[24] 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.

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

[26] Kathryn McKinley, Steve Carr, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, California, 1994.

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

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

[29] G.W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[30] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1990.

[31] 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.

[32] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois, October 1982.

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

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