Compiler Blockability of Numerical Algorithms

Steve Carr
Ken Kennedy

CRPC-TR92208
April, 1992

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Revised August, 1992.

Compiler Blockability of Numerical Algorithms*

Steve Carr
Ken Kennedy

Department of Computer Science
Rice University
Houston TX 77251-1892

Abstract

Over the past decade, microprocessor design strategies have focused on increasing the computational power on a single chip. Unfortunately, memory speeds have not kept pace. The result is an imbalance between computation speed and memory speed. This imbalance is leading machine designers to use more complicated memory hierarchies. In turn, programmers are explicitly restructuring codes to perform well on particular memory systems, leading to machine-specific programs. This paper describes our investigation into compiler technology designed to obviate the need for machine-specific programming. Our results reveal that through the use of compiler optimizations many numerical algorithms can be expressed in a natural form while retaining good memory performance.

1 Introduction

The trend in high-performance microprocessor design is toward increasing computational power on chip. Unfortunately, memory speed is not increasing at the same rate. The result has been an increase in the number of cycles for a memory access—a latency of 10 to 20 machine cycles is now quite common.

Although cache helps to ameliorate these problems, it performs poorly on scientific calculations with working sets larger than the cache size. This situation has led many programmers to restructure their codes by hand to improve performance in the memory hierarchy. We believe that this is a step in the wrong direction. The user should not be creating programs that are specific to a particular machine. Instead, the task of specializing a program to a target architecture should fall to the compiler.

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 sublanguage 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 this approach, we embarked on an experiment to determine if a compiler could automatically generate the block algorithms in LAPACK from the corresponding point algorithms expressed in Fortran 77 [DDSvdV91]. To assist this research, Dongarra and Sorensen have contributed point and block versions of several algorithms used in LAPACK. This paper extends some preliminary results of our efforts to define the compiler algorithms that would be needed to generate the block algorithms from the point algorithms automatically [CK89]. In performing this study, we address the question "What information does a compiler need in order to derive block versions of real-world codes that are competitive with the best hand-blocked versions?"

In the course of this study, we have discovered an algorithmic approach that can be used to analyze and block programs that exhibit complex dependence patterns. In addition, we have found transformation algorithms that can be successfully used on triangular loops, which are quite common in linear algebra, and trapezoidal loops. These methods have been successfully applied to block LU decomposition without pivoting. The key to many of these results is a transformation known as *index-set splitting*. Our results with this transformation show that a wide class of numerical algorithms can be automatically optimized for a

*Research supported by Darpa through ONR Grant N00014-91-J-1989 and by NSF Grant CCR-9120008

particular machine's memory hierarchy when they are expressed in a natural form. In addition, we have discovered that specialized knowledge about which operations commute with one another can enable compilers to block codes that were previously thought to be unblockable by automatic methods.

We begin our presentation with a review of background material related to memory optimization. Next, we present the transformations that are necessary to block LAPACK automatically. Then, we describe a study of the application of these transformations to derive the block algorithms in LAPACK from their corresponding point algorithms. For those algorithms that cannot be blocked by a compiler, we propose a set of language extensions to allow the expression of block algorithms in a machine-independent form. Finally, we review related work and present a summary.

2 Background

2.1 Dependence

The fundamental tool available to the compiler is the same tool used in vectorization and parallelization—namely *dependence*. 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 [Kuc78].

- 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 of the dependence are on different iterations of the loop and the dependence is not carried by an outer loop [AK87].

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 [CK87, HK91]. Sections describe common substructures of arrays such as elements, rows, columns and diagonals.

2.2 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.

2.3 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 [Wol87, Por89, WL91]. 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)
```

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 [LRW91].

A transformation analogous to strip-mine-and-interchange is unroll-and-jam [CCK90]. 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 .

3 Index-set splitting

Iteration-space blocking cannot always be directly applied as shown in the previous section. Sometimes safety constraints only permit a partial application of blocking. In these cases, a transformation called *index-set splitting* can be applied. 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.

3.1 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 [Wol86, Wol87]. Therefore, blocking triangular 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 α and β are literal integer constants (β 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 1 gives a graphical description of the iteration space of this loop. To interchange the II and J loops, the intersection of the line J=αII+β with the iteration space at the point (I,αI+β) must be handled. Therefore, interchanging the loops requires the

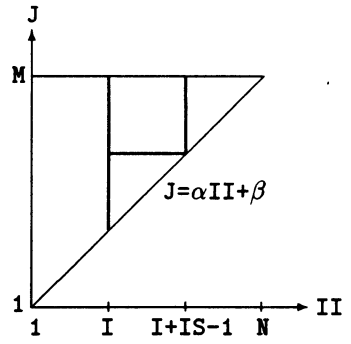

Figure 1: Upper Left Triangular Iteration Space

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 [Car92].

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 α(I+IS-1)+β creates one loop that iterates over the triangular region below the line J=α(I+IS-1)+β 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 α and β, 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 [Car92].

3.2 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 $I = \frac{N-\beta}{\alpha}$ 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. As an example, consider the following loop that computes the adjoint convolution of two time series.

```
      DO 10 I = 0,N3
        DO 10 K = I,MIN(I+N2,N1)
10        F3(I) = F3(I)+DT*F1(K)*F2(I-K)
```

The lower bound is a linear function of the outer-loop induction variable, resulting in rhomboidal and triangular regions. To handle this loop, blocking can be extended to rhomboidal regions using index-set splitting similar to the case for triangular regions [Car92]. As another more complex example, consider the following loop which computes the convolution of two time series.

```
      DO 10 I = 0,N3
        DO 10 K = MAX(0,I-N2),MIN(I,N1)
10        F3(I) = F3(I)+DT*F1(K)*F2(I-K)
```

The MAX function in the lower bound can be index-set split similar to a MIN function [Car92]. In this example, complete splitting to remove the functions from the loop bounds would result in four separate loops that can each be blocked.

The previous two loops come from an oil exploration program and constitute 20% of the program's execution time. After performing index-set splitting, unroll-and-jam and a transformation called scalar replacement on both loops, we ran them on arrays of double-precision REALS [CCK90]. Below is a table of the results on an IBM RS/6000 540. For timing measurements, we iterated over each kernel 1000 times with 75% of the execution in the triangular regions.

Loop	Size	Original	Xformed	Speedup
Aconv	300	4.59s	2.55s	1.80
	500	12.46s	6.65s	1.87
Conv	300	4.61s	2.53s	1.82
	500	12.56s	6.63s	1.91

3.3 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
        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. Unfortunately, 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) [Wol82]. 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 2. 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

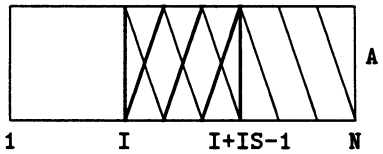

Figure 2: Data Space for A

Procedure *IndexSetSplit*

For each transformation-preventing dependence
repeat the following steps until failure
or a region is created that may be blocked.

1. Calculate the sections of the source and sink of the preventing dependence.
2. Intersect and union the sections using symbolic information.
3. If the intersection and union are equal then stop.
4. Set the subscript expression of the larger section equal to the boundary between the disjoint and common sections and solve for the inner-loop induction variable.
5. Split the index set of the inner loop at this point.
6. Repeat steps 4 and 5 if there are multiple boundaries.

Figure 3: Procedure IndexSetSplit

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
        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 positions (e.g., A(II) and A(K) in the previous example). Figure 3 presents the method *IndexSetSplit* used after strip mining to handle partial blocking.

The effectiveness of *IndexSetSplit* 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 [HK91]. In Section 5, we show that this representation allows *IndexSetSplit* to greatly enhance the performance of solving systems of linear equations.

4 IF-inspection

In addition to iteration-space shapes and dependence patterns, the effects of control flow on blocking must also be considered. It may be the case that an inner loop is guarded by an IF-statement to prevent unnecessary computation. Consider the following matrix multiply code that is take from the BLAS routine SGEMM [DDSvdV91].

```
      DO 20 J = 1,N
        DO 20 K = 1,N
          IF (B(K,J) .EQ. 0.0) GOTO 20
            DO 10 I = 1,N
10            C(I,J) = C(I,J) + A(I,K) * B(K,J)
20      CONTINUE
```

If the IF-statement were ignored and unroll-and-jam were performed on the K-loop, references to B would be introduced that would never be checked by the guard. Therefore, statements that were not executed in the original code may be unsafely executed in the unrolled code.

One possible method to preserve correctness is to move the guard into the innermost loop and replicate it for each unrolled iteration. However, this would result in a performance degradation due to a decrease in loop-level parallelism and an increase in instructions executed. Instead, a combination of IF-conversion and sparse-matrix techniques, called IF-inspection, can be used to keep the guard out of the innermost loop and still allow unroll-and-jam [AK87]. The idea is to inspect at run-time the values of an outer-loop induction variable for which the guard is true and the inner loop is executed. Then, the inner-loop nest is executed only for those values.

To effect IF-inspection, code is inserted within the IF-statement to record loop bounds information for the loop to be transformed. On the true branch of the guard to be inspected the following code is inserted, where KC is initialized to 1, FLAG is initialized to false, K is the induction variable of the loop to be inspected and KLB is the lower bound of an executed range.

```
IF (.NOT. FLAG) THEN
  KC = KC + 1
  KLB(KC) = K
  FLAG = .TRUE.
ENDIF
```

On the false branch of the inspected guard, the following code is inserted to store the upper bound of each executed range.

```
IF (FLAG) THEN
  KUB(KC) = K-1
  FLAG = .FALSE.
ENDIF
```

Note that the value of the guard could be true on the last iteration of the loop, requiring a test of FLAG to store the upper bound of the last range after the IF-inspection loop body.

After inserting the inspection code, the loop to be transformed is distributed around the inspection code

```
      FLAG = .FALSE.
      DO 10 J = 1,N
       KC = 0
       DO 20 K = 1,N
         IF (B(K,J) .NE. 0.0) THEN
           IF (.NOT. FLAG) THEN
             KC = KC + 1
             KLB(KC) = K
             FLAG = .TRUE.
           ENDIF
         ELSE
           IF (FLAG) THEN
             KUB(KC) = K-1
             FLAG = .FALSE.
           ENDIF
         ENDIF
20     CONTINUE
       IF (FLAG) THEN
         KUB(KC) = N
         FLAG = .FALSE.
       ENDIF
       DO 10 KN = 1,KC
         DO 10 K = KLB(KN),KUB(KN)
           DO 10 I = 1,N
10           C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

Figure 4: Matrix Multiply After IF-Inspection

and a new loop nest that executes over the iteration space where the innermost loop was executed is created. The result of IF-inspection on matrix multiply is shown in Figure 4. The KN-loop executes over the number of ranges where the guarded loop is executed and the new K-loop executes within those ranges.

If the ranges over which the inner loop is executed in the original loop are large, the slight increase in run-time cost caused by IF-inspection can be more than counteracted after transforming the new loop nest for better data locality. To show this, we performed unroll-and-jam on our IF-inspected matrix multiply example and ran it on an IBM RS/6000 model 540 on 300x300 arrays of REALS. In the table below, *Frequency* shows how often B(K,J) = 1, UJ is the result of performing unroll-and-jam after moving the guard into the innermost loop and UJ+IF is the result of performing unroll-and-jam after IF-inspection.

Frequency	Original	UJ	UJ+IF	Speedup
2.5%	3.33s	3.84s	2.25s	1.48
10%	3.08s	3.71s	2.13s	1.45

5 Systems of linear equations

The goal of LAPACK is to replace the algorithms in LINPACK and EISPACK with block algorithms that have better cache performance. Unfortunately, the LAPACK designers have achieved additional performance at the expense of machine independence. To adapt its kernels from one machine to another, a programmer must perform machine-specific hand optimization on each LAPACK subroutine to obtain high performance. In contrast, we believe that programmers should express each kernel 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 LAPACK in a machine-independent form, this section examines the blockability of three of LAPACK's algorithms for solving systems of linear equations using the techniques developed in Section 3. An algorithm is "blockable" if a compiler can automatically derive the *best* known block algorithm, the one found in LAPACK, from its corresponding machine-independent point algorithm. Our study shows that LU decomposition without pivoting is a blockable algorithm using the method *IndexSetSplit*, LU decomposition with partial pivoting is blockable using *IndexSetSplit* and information about commutative operations, and QR decomposition with Householder transformations is not blockable. The study also shows how to improve the memory performance of a fourth non-LAPACK algorithm, QR decomposition with Givens rotations.

5.1 LU decomposition without pivoting

Gaussian elimination is a form of LU decomposition where the matrix A is decomposed into two matrices, L and U, such that

$$A = LU,$$

L is a unit lower triangular matrix and U is an upper triangular matrix. This decomposition can be obtained by multiplying the matrix A by a series of elementary lower triangular matrices, where $L = M_k^{-1} \ldots M_1^{-1}$, as follows [Ste73].

$$U = M_k \ldots M_1 A$$

Using this equation, an algorithm for LU decomposition without pivoting using Gaussian elimination is derived. The point algorithm, where statement 20 computes M_k and statement 10 applies M_k to A, is shown below after strip mining.

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

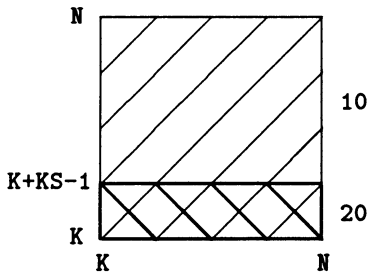

Figure 5: Sections of A in LU Decomposition

Unfortunately, this algorithm exhibits poor cache performance on large matrices. To improve its cache performance, 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 [DDSvdV91]. To attain the best block version, strip-mine-and-interchange is completed for the K-loop on only a portion of the inner loop nest. We show how to attain the block algorithm using *IndexSetSplit*.

To complete the blocking of strip-mined LU decomposition, 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. 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 done.

Figure 5 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)
```

Now, triangular interchange can be used to put the KK-loop in the innermost position (see Figure 6). At this point, the best block algorithm has been obtained. Therefore, LU decomposition is blockable. Not only does this block algorithm exhibit better data

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

Figure 6: Block LU Decomposition

locality, it also has increased parallelism as the J-loop that surrounds statement 10 can be made parallel.

We applied *IndexSetSplit* by hand to LU decomposition and compared its performance with the point algorithm and a hand-coded version by Sorensen. In the table below, "1" refers to the Sorensen version and "2" refers to the algorithm in Figure 6. In addition, we performed trapezoidal unroll-and-jam and scalar replacement to our blocked code, producing the version referred to as "2+". The experiment was run on an IBM RS/6000 540 using double-precision REALS. Note that the final transformations could have been applied to the Sorensen version as well, with similar improvements.

Size	Block	Point	1	2	2+	Speedup
300	32	1.47s	1.37s	1.35s	0.49s	3.00
300	64	1.47s	1.42s	1.38s	0.58s	2.53
500	32	6.76s	6.58s	6.44s	2.13s	3.17
500	64	6.76s	6.59s	6.38s	2.27s	2.98

5.2 LU decomposition with partial pivoting

Although the compiler can discover the potential for blocking in LU decomposition without pivoting using the algorithm *IndexSetSplit*, the same cannot be said when partial pivoting is added (see Figure 7 for LU decomposition with partial pivoting). In the partial pivoting algorithm a new recurrence exists that does not fit the form handled by *IndexSetSplit*. Consider the following sections of code after applying *IndexSetSplit* to the algorithm in Figure 7.

```
      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)

```
      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 7: LU Decomposition with Partial Pivoting

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 (see Figure 8) [DDSvdV91]. This block algorithm also exhibits the increased loop-level parallelism found in the algorithm in Figure 6.

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.

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

Figure 8: Block LU with Partial Pivoting

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. The question is, however, "Will the increase in knowledge be profitable?" To see the potential profitability of making the compiler more sophisticated, consider the table below, where "1" refers to the algorithm given in Figure 8 and "1+" refers to that algorithm after unroll-and-jam and scalar replacement. This experiment was run on an IBM RS/6000 540 using double-precision REALS.

Size	Block	Point	1	1+	Speedup
300	32	1.52s	1.42s	0.58s	2.62
300	64	1.52s	1.48s	0.67s	2.27
500	32	7.01s	6.85s	2.58s	2.72
500	64	7.01s	6.83s	2.73s	2.57

5.3 Householder QR

The key to Gaussian elimination is the multiplication of the matrix A by a series of elementary lower triangular matrices that introduce zeros below each diagonal element. Any class of matrices that have this property can be used to solve a system of linear equations. One such class, having orthonormal columns, is used in QR decomposition [Ste73].

If A has linearly independent columns, then A can be written uniquely in the form

$$A = QR,$$

where Q has orthonormal columns, $QQ^T = I$ and R is upper triangular with positive diagonal elements. One class of matrices that fits the properties of Q is elementary reflectors or *Householder transformations* of the form $I - 2vv^T$.

The point algorithm for Householder QR consists of iteratively applying the elementary reflector $V_k = I - 2v_k v_k^T$ to A_k to obtain A_{k+1} for $k = 1, \ldots, n-1$. Each V_k eliminates the values below the diagonal in the kth column. For a more detailed discussion of the QR algorithm and the computation of V_k, see Stewart [Ste73].

Although pivoting is not necessary for QR decomposition, the best block algorithm is not an aggregation of the original algorithm. The block application

of a number of elementary reflectors involves both computation and storage that does not exist in the original algorithm [DDSvdV91]. Given

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

the first step is to factor

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{pmatrix} \begin{pmatrix} R_{11} \\ 0 \end{pmatrix},$$

and then solve

$$\begin{pmatrix} \hat{A}_{12} \\ \hat{A}_{22} \end{pmatrix} = \hat{Q} \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix},$$

where

$$\begin{aligned} \hat{Q} = & (I - 2v_1v_1^T)(I - 2v_2v_2^T)\cdots(I - 2v_bv_b^T) \\ = & I - 2VTV^T. \end{aligned}$$

The difficulty for the compiler comes in the computation of $I - 2VTV^T$ because it involves space and computation that did not exist in the original point algorithm. To illustrate, consider the case where the block size is 2.

$$\begin{aligned} \hat{Q} = & (I - 2v_1v_1^T)(I - 2v_2v_2^T) \\ = & I - 2(v_1v_2)\begin{pmatrix} 1 & (v_1^Tv_2) \\ 0 & 1 \end{pmatrix}\begin{pmatrix} v_1^T \\ v_2^T \end{pmatrix} \end{aligned}$$

Here, the computation of the matrix

$$T = \begin{pmatrix} 1 & (v_1^Tv_2) \\ 0 & 1 \end{pmatrix}$$

is not part of the original algorithm, making it impossible to determine the computation of $\hat{Q}$ from the data dependence information.

The expression of block Householder QR requires the choice of a machine-dependent blocking factor. We know of no way to express this algorithm in a current programming language in a manner that would allow a compiler to automatically chose that factor. However, the expressibility of a language can be enhanced to allow block algorithms to be stated in a machine-independent form. In Section 6, we address this issue.

5.4 Givens QR

Another form of orthogonal matrix that can be used in QR decomposition is the Givens rotation matrix [Sew90]. We currently know of no best block algorithm to derive from the point algorithm, so instead we show that *IndexSetSplit* and IF-inspection have wider applicability.

Consider the Fortran code for Givens QR shown in Figure 9 [Sew90]. The references to A in the inner K-loop have a long stride between successive accesses, resulting in poor cache performance. Interchanging the J-loop to the innermost position would

```
      DO 10 L = 1,M
        DO 10 J = L+1,M
          IF (A(J,L) .EQ. 0.0) GOTO 10
            DEN = DSQRT(A(L,L)*A(L,L) + A(J,L)*A(J,L))
            C = A(L,L)/DEN
            S = A(J,L)/DEN
            DO 10 K = L,M
              A1 = A(L,K)
              A2 = A(J,K)
              A(L,K) = C*A1 + S*A2
              A(J,K) = -S*A1 + C*A2
   10     CONTINUE
```

Figure 9: QR Decomposition with Givens Rotations

give stride-one access to the references to A(J,K) and make the references to A(L,K) invariant with respect to the innermost loop. In this case, loop interchange would necessitate distribution of the J-loop around the IF-block and the K-loop. However, a recurrence consisting of a true and antidependence between the definition of A(L,K) and the use of A(L,L) prevents distribution. Examining the sections for these references reveals that the recurrence only exists for the element A(L,L), allowing index-set splitting of the K-loop at L, IF-inspection of the J-loop, distribution (with scalar expansion) and interchange (see Figure 10) [KKP+81]. Below is a table of the results of the performance of Givens QR.

Array Size	Point	Optimized	Speedup
300x300	6.86s	3.37s	2.04
500x500	84.0s	15.3s	5.49

6 Language extensions

The examination of QR decomposition with Householder transformations shows that some block algorithms cannot be derived by a compiler from their corresponding point algorithms. In order to maintain the goal of machine-independent coding styles, the expression of these types of block algorithms in a machine-independent form must be made possible. Specifically, the compiler needs to be directed to pick the machine-dependent blocking factor for an algorithm automatically.

To this end, we present a preliminary proposal for two looping constructs to guide the compiler's choice of blocking factor. These constructs are BLOCK DO and IN DO. BLOCK DO specifies a DO-loop whose blocking factor is chosen by the compiler. IN DO specifies a DO-loop that executes over the region defined by a corresponding BLOCK DO and guides the compiler to the regions that it should analyze to determine the blocking factor. The bounds of an IN DO statement are optional. If they are not expressed, the bounds are assumed to start at the first value in the specified

```
      DO 10 L = 1,N
        DO 20 J = L+1,M
          IF (A(J,L) .EQ. 0.0) GOTO 20
            DEN = DSQRT(A(L,L)*A(L,L)+A(J,L)*A(J,L))
            C(J) = A(L,L)/DEN
            S(J) = A(J,L)/DEN
            A1 = A(L,L)
            A2 = A(J,L)
            A(L,L) = C(J)*A1 + S(J)*A2
            A(J,L) = -S(J)*A1 + C(J)*A2

C
C     IF-Inspection Code including 20
C
        DO 10 K = L+1,N
          DO 10 JN = 1,JC
            DO 10 J = JLB(JN),JUB(JN)
              A1 = A(L,K)
              A2 = A(J,K)
              A(L,K) = C(J)*A1 + S(J)*A2
 10           A(J,K) = -S(J)*A1 + C(J)*A2
```

Figure 10: Optimized Givens QR

```
BLOCK DO K = 1,N-1
  IN K DO KK
    DO I = KK+1,N
      A(I,KK) = A(I,KK)/A(KK,KK)
    ENDDO
    DO J = KK+1,LAST(K)
      DO I = KK+1,N
        A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
      ENDDO
    ENDDO
  ENDDO
  DO J = LAST(K)+1,N
    DO I = K+1,N
      IN K DO KK = K,MIN(LAST(K),I-1)
        A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Figure 11: Block LU in Extended Fortran

block and end at the last value with a step of 1. To allow indexing within a block region, **LAST** returns the last index value. For example, if LU decomposition were not a blockable algorithm, it could be coded as in Figure 11 to achieve machine independence.

The principal advantage of the extensions is that the programmer can express a non-blockable algorithm in a natural block form, while leaving the machine-dependent details, namely the choice of blocking factor, to the compiler. In the case of LAPACK, the language extensions could be used, when necessary, to code the algorithms for a machine-independent source-level library. Then, compiler technology could be used to port the library from machine to machine and still retain good performance. By doing so, the machine-dependency problem of LAPACK would be removed, making LAPACK readily accessible to new architectures.

7 Previous work

Wolfe has done a significant amount of work on cache blocking [Wol86, Wol87, Wol89]. In particular, he discusses strip-mine-and-interchange for triangular-shaped iteration spaces, but he does not present general compiler algorithms nor extend the work to unroll-and-jam. He also shows by example how to use index-set splitting to handle a trapezoidal region that arises from triangular iteration-space blocking. We take this a step further by developing a general technique that handles more cases.

Irigoin and Triolet describe a general technique for blocking iteration spaces for memory that uses a dependence abstraction called a *dependence cone* [IT88]. This technique does not work on non-perfectly nested loops, which are common in linear algebra codes, nor does it handle partially blockable loops.

Wolf and Lam present a framework for applying blocking transformations and ordering a loop nest for memory performance and parallelism [WL91]. However, their framework does not include the application of index-set splitting nor is it applicable to non-perfectly nested loops.

8 Summary

We have set out to determine whether a compiler can automatically restructure computations well enough to avoid the need for hand blocking. To that end, we have examined a collection of programs in LAPACK for which we were able to examine both the block version and the corresponding point algorithm. 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 can block triangular and trapezoidal loops and we have found that many of the problems introduced by complex dependence patterns can be overcome by the use of the transformation known as "index-set splitting". In many cases, index-set splitting yields codes that exhibit performance at least as good as the best block algorithms produced by LAPACK developers. In addition, we have shown that knowledge about which operations commute can enable a compiler to succeed in blocking codes that could not be blocked by any compiler based strictly on dependence analysis. Unfortunately, our success has not been universal. For methods like QR decomposition, the block algorithm

has no corresponding point algorithm because block sizes larger than one require additional computation to compensate for the blocking. If automatic blocking is to succeed, machine-independent expression of block algorithms, such as that proposed in Section 6, must be developed.

Our goal has been to find compiler techniques that make it possible for the user to express numerical algorithms naturally with the expectation of good memory hierarchy performance. We have demonstrated that there exist readily implementable methods that can automatically block many, but not all, linear algebra codes. Currently, we have implemented triangular, trapezoidal and rhomboidal blocking in an experimental system. In the future, we plan to add *IndexSetSplit* and commutativity knowledge. Then, we will apply the resulting compiler to a collection of scientific programs in order to better understand the breadth of coverage supplied by these methods. In addition, we will continue to investigate language extensions that would make it possible to express block algorithms in a machine-independent style.

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. In this paper we have established that, with a few additional methods, the compiler can do a good job on many key algorithms from linear algebra. If these results extend to more general computations, it will represent a significant step toward fully automatic memory-hierarchy management.

Acknowledgments

Preston Briggs, Rebecca Carr, Uli Kremer and Kathryn McKinley made many helpful suggestions during the preparation of this document. Danny Sorensen provided us with block and point versions of the LAPACK algorithms and gave us guidance in understanding these algorithms. To all of these people go our heartfelt thanks.

References

[AK87] 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.

[Car92] S. Carr. *Memory-Hierarchy Management.* PhD thesis, Rice University, Department of Computer Science, 1992.

[CCK90] 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.

[CK87] 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.

[CK89] 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.

[DDSvdV91] 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.

[HK91] 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.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 319–328, January 1988.

[KKP+81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eight ACM Symposium on the Principles of Programming Languages*, 1981.

[Kuc78] D. Kuck. *The Structure of Computers and Computations Volume 1.* John Wiley and Sons, New York, 1978.

[LRW91] 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.

[Por89] A.K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989.

[Sew90] G Sewell. *Computational Methods of Linear Algebra.* Ellis Horwood, England, 1990.

[Ste73] G.W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[WL91] 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.

[Wol82] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois, October 1982.

[Wol86] M. Wolfe. Advanced loop interchange. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[Wol87] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[Wol89] M. Wolfe. More iteration space tiling. In *Proceedings of the Supercomputing '89 Conference*, 1989.