SIAM REVIEW
Vol. 26, No. 1, January, 1984

IMPLEMENTING LINEAR ALGEBRA ALGORITHMS FOR DENSE MATRICES ON A VECTOR PIPELINE MACHINE*

J. J. DONGARRA,† F. G. GUSTAVSON‡ AND A. KARP§

Abstract. This paper examines common implementations of linear algebra algorithms, such as matrix-vector multiplication, matrix-matrix multiplication and the solution of linear equations. The different versions are examined for efficiency on a computer architecture which uses vector processing and has pipelined instruction execution. By using the advanced architectural features of such machines, one can usually achieve maximum performance, and tremendous improvements in terms of execution speed can be seen over conventional computers.

1. Introduction. In this paper we describe why existing algorithms for linear algebra are not usually suited for computers that employ advanced concepts such as pipelining and vector constructs to achieve enhanced performance. We examine the process of refitting or reorganizing an underlying algorithm to conform to the computer architecture, thereby gaining tremendous improvements in execution speeds while sacrificing neither accuracy nor algorithm clarity. This reorganization, where it can be done, is usually conceptually simple at the algorithm level. This paper will not address the issues involved with parallel processing. For a survey of parallel algorithms in linear algebra see the review paper by Heller [8].

We will not concern ourselves here with an actual implementation on a specific architecture: To do so, one must understand all the subtlety and nuances of that architecture and risk obscuring the fundamental ideas. Rather, we use the features of a vector pipeline machine to understand how various aspects interrelate and how they can be put together to achieve very high execution rates.

We use the term architecture in reference to the organization of the computer as seen by the programmer or algorithm designer. Within the architecture we focus on the instruction set and memory references, and their interaction in terms of performance.

We will concentrate our examination on the behavior of linear algebra algorithms for dense problems that can be accommodated in the main memory of a computer. The solutions proposed here do not, in general, carry over to sparse matrices because of the short vector lengths and the indirect addressing schemes that are so prevalent in sparse matrix calculations. For a discussion of methods for handling the sparse matrix case, see [5], [7].

We will focus in particular on algorithms written in Fortran and assembly language. Fortran is an appropriate language, given the scientific nature of the application; occasional use of assembly language enables us to gain the maximum speed possible. The use of Fortran implies some storage organization for array elements. By definition Fortran must have matrix elements stored sequentially by column. As one accesses consecutive elements in a column of an array, the next element is found in the adjacent location. If references are made by rows of the array, accesses to the next element of a row must be offset by the number of elements in a column of the array. This organization will be important in the actual implementation.

*Received by the editors September 21, 1982, and in revised form June 28, 1983. This research was supported in part by IBM.

†Argonne National Laboratory, Argonne, Illinois 60439. The research of this author was supported in part by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under contract W-31-109-Eng-38.

‡IBM T. J. Watson Research Center, Yorktown Heights, New York 10598.

§IBM Palo Alto Scientific Center, Palo Alto, California 94304.

2. Vector pipeline concepts. As background we will describe some of the basic features found in supercomputers, concentrating on those features that are particularly relevant to implementing linear algebra algorithms. For a more thorough discussion of supercomputers, see [9], [11], [13]. We will concentrate our attention on an architecture that is "Cray-like" in structure, i.e., one that performs vector operations in a vector register configuration with concurrency provided by pipelining and independent instruction execution. We choose a configuration like the Cray-1 [16] for a number of reasons: It performs well on short vectors, it has a simple instruction set, and it has an extremely fast execution rate for dense, in-core linear algebra problems. Nevertheless, the underlying concepts can be applied to other machines in its class, e.g., the Cyber 205. (A few words of caution: Not everything can be carried over. For instance, in the Cyber 205 access must be made by column to allow for sequential referencing of matrix elements.)

A computer that is "Cray-like" derives its performance from several advanced concepts. One of the most obvious is the use of vector instructions. By means of a single instruction, all elementwise operations that make up the total vector operation are carried out. The instructions are performed in vector registers. The machine may have k such elements in a vector register in addition to having a conventional set of registers for scalar operations. A typical sequence of instructions would be as follows:

Load a scalar register from memory
Load a vector register from memory
Perform a scalar-vector multiplication
Load a vector register from memory
Perform a vector-vector addition
Store the results in memory.

These six instructions would correspond to perhaps $6k + 1$ instructions on a conventional computer, where k instructions are necessary for loop branching. Clearly, then, the time to interpret the instructions has been reduced by almost a factor of k, resulting in a significant savings in overhead.

Cray-like hardware typically provides for "simultaneous" execution of a number of elementwise operations through pipelining. Pipelining generally takes the approach of splitting the function to be performed into smaller pieces or stages and allocating separate hardware to each of these stages. Pipelining is analogous to an industrial assembly line where a product moves through a sequence of stations. Each station carries out one step in the manufacturing process, and each of the stations works simultaneously on different units in different phases of completion. With pipelining, the functional units (floating point adder, floating point multiplier, etc.) can be divided into several suboperations that must be carried out in sequence. A pipelined functional unit, then, is divided into stages, each of which does a portion of the work in, say, one clock period. At the end of each clock period, partial results are passed to the next stage and partial results are accepted from the previous stage.

The goal of pipelined functional units is clearly performance. After some initial startup time, which depends on the number of stages (called the length of the pipeline, or pipe length), the functional unit can turn out one result per clock period as long as a new pair of operands is supplied to the first stage every clock period. Thus, the rate is independent of the length of the pipeline and depends only on the rate at which operands are fed into the pipeline. Therefore, if two vectors of length k are to be added, and if the floating point adder requires 3 clock periods to complete, it would take $3 + k$ clock periods to add the two vectors together, as opposed to $3 * k$ clock periods in a conventional computer.

Another feature that is used to achieve high rates of execution is chaining. *Chaining* is a technique whereby the output register of one vector instruction is the same as one of the input registers for the next vector instruction. If the instructions use separate functional units, the hardware will start the second vector operation during the clock period when the first result from the first operation is just leaving its functional unit. A copy of the result is forwarded directly to the second functional unit and the first execution of the second vector is started. The net result is that the execution of both vector operations takes only the second functional unit startup time longer than the first vector operation. The effect is that of having a new instruction which performs the same operation as that of the two functional units that have been chained together. On the Cray in addition to the arithmetic operations, vector loads from memory to vector registers can be chained with other arithmetic operations.

For example, let us consider a case involving a scalar-vector multiplication, followed by a vector-vector addition, where the addition operation depends on the results of the multiplication. Without chaining, but with pipelined functional units, the operation would take $a + k + m + k$ clock periods, where a is the time to start the vector addition (length of the vector addition pipeline) and m is the time to start a vector multiplication (length of the vector multiplication pipeline). With chaining, as soon as a result is produced from the adder, it is fed directly into the multiplication unit, so the total time is $a + m + k$. We may represent this process graphically as in Fig. 1.

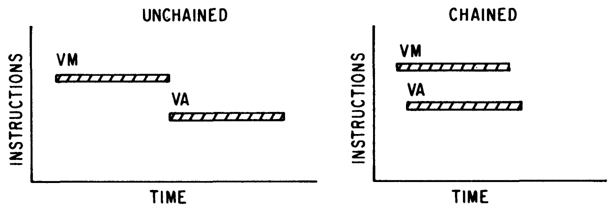


FIG. 1.

It is also possible to overlap operations if the two operations are independent. If a vector addition and an independent vector multiplication are to be processed, the resulting timing graph might look like Fig. 2.

To describe the time to complete a vector operation, we use the concept of a chime [6]. A *chime* (for *ch*aining t*ime*) is a measure of the time needed to complete a sequence of vector operations. To compute the number of chimes necessary for a sequence of operations, one divides the total time to complete the operations by the vector length. Overhead of startup and scalar work are usually ignored in counting chimes, and only the integer part is reported. For example, in the graph for unchained operations above there are two chimes, whereas in the graph for the chained operation there is one chime.

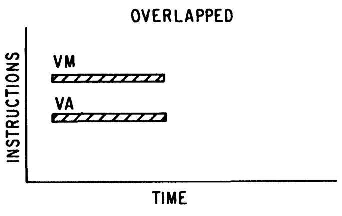


FIG. 2.

As Fong and Jordan [6] have pointed out, there are three performance levels for algorithms on the Cray. The two obvious ones are scalar and vector performance. Scalar performance is achieved when operations are carried out on scalar quantities, with no use of the vector functional units. Vector performance is achieved when vectors are loaded from memory into registers, operations such as multiplication or addition are performed, and the results are stored into memory. The third performance level is called supervector [6], [10]. This level is achieved when vectors are retained in registers, operations are performed using chaining, and the results are stored in registers.

Dramatic improvements in rates of execution are realized in going from scalar to vector and from vector to supervector speeds. We show in Fig. 3 a graph of the execution rate in MFLOPS (million floating point operations per second) for *LU* decomposition of a matrix of order *n* as performed on the Cray-1. When supervector rates are achieved, the hardware is being driven at close to its highest potential. Later in this paper we describe what leads to this supervector performance.

In summary, then, vector machines rely on a number of techniques to enhance their performance over conventional computers:

Fast cycle time,
Vector instructions to reduce the number of instructions interpreted,
Pipelining to utilize a functional unit fully and to deliver one result per cycle,
Chaining to overlap functional unit execution, and
Overlapping to execute more than one independent vector instruction concurrently.

Programs that use these features properly will fully utilize the potential of the vector machine.

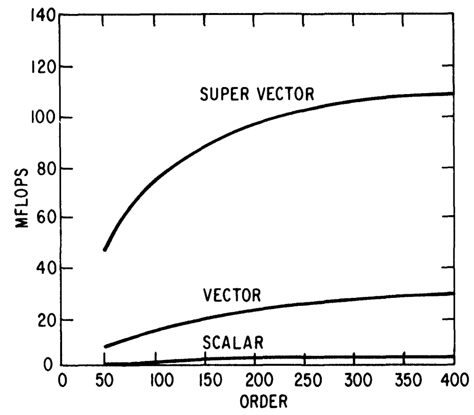


FIG. 3.

3. Matrix-vector example. We are now ready to examine a simple but important operation that pervades scientific computations: the matrix-vector product $y \leftarrow A * x$, where y is a vector with m elements, A is a matrix with m rows and n columns, and x is a vector with n elements. We will write the matrix-vector product as follows:

Generic matrix-vector multiplication algorithm.
```
for ____ = 1 to ____
  for ____ = 1 to ____
    $y_i = y_i + a_{ij} * x_j$
  end
end
```

We have intentionally left blank the loop indices and termination points of the loop indices. There are basically two ways to encode this operation:

```
for $i = 1$ to $m$
  $y_i = 0$
  for $j = 1$ to $n$
    $y_i = y_i + a_{ij} * x_j$
  end
end

$y \leftarrow A * x$
```

Form ij

```
for $i = 1$ to $m$
  $y_i = 0$
end
for $j = 1$ to $n$
  for $i = 1$ to $m$
    $y_i = y_i + a_{ij} * x_j$
  end
end
$y \leftarrow A * x$
```

Form ji

In form *ij*, references to the matrix A are made by accessing across the rows. Since Fortran stores matrix elements consecutively by column, accesses made to elements in a row need to have an increment, or stride, different from one in order to reference the next element of the row. For most machines a stride of one or any other constant value can be accommodated.

With a "Cray-like" computer in which the memory cycle time is longer than the processor cycle time, however, there exists the possibility of a memory bank conflict.

After an access, a memory bank requires a certain amount of time before another reference can be made. This delay time is referred to as the "memory bank cycle time." This memory cycle time on both the Cray-1 and the Cyber 205 is four processor cycles. The operation of loading elements of a vector from memory into a vector register can be pipelined, so that after some initial startup following vector load, vector elements arrive in the register at the rate of one element per cycle. To keep vector operands streaming at this rate, sequential elements are stored in different banks in an "interleaved" memory. If a memory bank is accessed before the memory has a chance to cycle, chaining operations will stop. Instead of delivering one result per cycle, the machine will deliver one result per functional unit time. In other words, it will operate at scalar speeds, seriously degrading performance. Note, too, that the Cyber 205 must gather data that is not contiguous in memory before it can begin computing.

When matrix elements are referenced by column, a bank conflict cannot occur; but if accesses are by row, there is a real potential of such a conflict. (Whether or not conflicts occur depends on the number of memory banks and on the size of the array used to contain the matrix.) The moral is that column accesses should be used whenever possible in a Fortran environment. (Note also that in computers which use virtual memory and/or cache memory, row accesses greatly increase the frequency of page faults.) As we shall see, in most situations by a simple reorganization the inner product can be replaced by a complete vector operation.

Form *ji* uses column operations exclusively. The basic operation here is taking a scalar multiple of one vector, adding it to another vector and storing the result. We refer to this operation as *SAXPY*, the name given it in the BLAS (Basic Linear Algebra Subprograms) [12]. If we examine Form *ji*, we see that the vector y can reside in a vector register and that it need not be stored into memory until the entire operation is completed. Columns of the matrix A are loaded into a register and scaled by the appropriate element of x; a vector accumulation is made with the intermediate result. In a sense what we have here is a *generalized* from of the *SAXPY*. We give the name *GAXPY* to the operation of loading a sequence of vectors from memory, multiplying the vectors by a sequence of scalars and accumulating the sum in a vector register. As we shall see, the *GAXPY* is a fundamental operation for many forms of linear algebra algorithms and utilizes a machine like the Cray to its full potential. We use the term *GAXPY* to conform to the style of the BLAS. *GAXPY* is simply a matrix vector product.

We recommend that in general that Form ji be used over Form ij. This suggestion holds when m and n are roughly the same size. Indeed, Form ji usually provides the best results. Nevertheless, the following caveat should be made: When m is much less than n and less than the number of elements in a vector register, Form ij should be given consideration. In such cases, Form ji will have short vector lengths, whereas Form ij will have long vectors.

In some situations it may be necessary to calculate a matrix vector product with additional precision, e.g., when a residual calculation is needed. One immediately thinks of using accumulation of inner product; but from our discussion above, we see that a *GAXPY* can be used to accomplish the same task in a purely vector fashion, provided that vector operations can use extended precision arithmetic.

4. Matrix multiplication. We will now look at another fundamental operation in linear algebra: the process of multiplying two matrices together. The process is conceptually simple, but the number of operations to carry out the procedure is large. In comparison to other processes such as solving systems of equations ($2/3\ n^3$ operations) or performing the QR factorization of a matrix ($4/3\ n^3$ operations [17]), matrix multiplication requires more operations ($2n^3$ operations). (Here we have used matrices of order n for the comparisons. The operation counts reflect floating point multiplication as well as floating point addition. Since addition and multiplication take roughly the same amount of time to execute and each unit can deliver one result per cycle, the standard practice is to count separately each floating point multiplication and floating point addition.)

We wish to find the product of A and B and store the result in C. From relationships in linear algebra we know that if A and B are of dimension $m \times n$ and $n \times p$, respectively, then the matrix C is of dimension $m \times p$. We will write the matrix multiplication algorithm as

Generic matrix multiplication algorithm.
```
for ____ = 1 to ____
  for ____ = 1 to ____
    for ____ = 1 to ____
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    end
  end
end
```

We have intentionally left blank the loop indices and termination points. The loop indices will have variable names i, j, and k, and the termination points for the indices will be m, p, and n, respectively.

Six permutations are possible for arranging the three loop indices. The generic algorithm will give rise to six forms of matrix multiplication. Each implementation will have quite different memory access patterns, which will have an important impact on the performance of the algorithm on a "Cray-like" processor. For an alternative derivation of these six variants, see [1].

We summarize the algorithms below.

```
for $i = 1$ to $m$
  for $j = 1$ to $p$
    $c_{ij} = 0$
    for $k = 1$ to $n$
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    end
  end
end
```

Form ijk

```
for $j = 1$ to $p$
  for $i = 1$ to $m$
    $c_{ij} = 0$
    for $k = 1$ to $n$
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    end
  end
end
```

Form jik

```
for $i = 1$ to $m$
  for $j = 1$ to $p$
    $c_{ij} = 0$
  end
end
for $k = 1$ to $n$
  for $i = 1$ to $m$
    for $j = 1$ to $p$
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    end
  end
end
```

Form kij

```
for $j = 1$ to $p$
  for $i = 1$ to $m$
    $c_{ij} = 0$
  end
end
for $k = 1$ to $n$
  for $j = 1$ to $p$
    for $i = 1$ to $m$
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    end
  end
end
```

Form kji

```
for $i = 1$ to $m$
  for $j = 1$ to $p$
    $c_{ij} = 0$
  end
  for $k = 1$ to $n$
    for $j = 1$ to $p$
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    end
  end
end
```

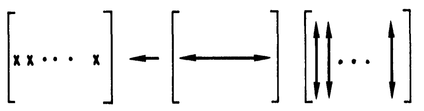
Form ikj

```
for $j = 1$ to $p$
  for $i = 1$ to $m$
    $c_{ij} = 0$
  end
  for $k = 1$ to $n$
    for $i = 1$ to $m$
      $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
    end
  end
end
```

Form jki

We have placed the initialization of the array in natural locations within each of the algorithms. All the algorithms displayed above perform the same arithmetic operations but in a different sequence; even the roundoff errors are the same. Their performance when implemented in Fortran can vary greatly because of the way information is accessed. What we have done is simply "interchanged the loops" [14]. This rearrangement dramatically affects performance on a "Cray-like" machine.

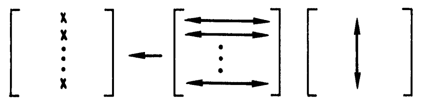
The algorithms of the form *ijk* and *jik* are related by the fact that the inner loop is performing an inner product calculation. We will describe the vector operation and data sequencing graphically by means of a diagram:



Form ijk

The diagram describes the sequence that the inner products of all columns of B with a row of A are computed to produce a row of C, one element at a time.

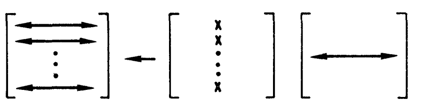
For the form *jik* the data are referenced slightly differently so that the diagram is of the form



Form jik

Both descriptions use an inner product as the basic operation. For reasons stated earlier about bank conflicts when accessing elements in a row of a matrix, we do not recommend the use of inner products such as in form *jik* on a "Cray-like" machine.

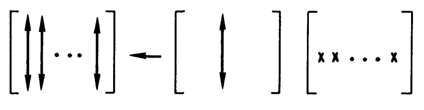
The algorithms of the form *kij* and *kji* are related in that they use the *SAXPY* as a basic operation, taking a multiple of a vector added to another vector. For the form *kij* we have



Form kij

In this case, a row of B is scaled by elements of a column of A, and the result is used to update rows of C.

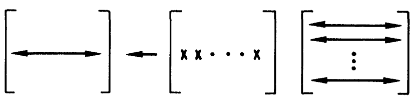
For the form *kji* the access pattern appears as



Form kji

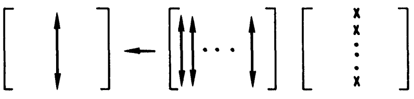
Since the access patterns for form *kji* are by column, we recommend it over *kij* in a Fortran environment. It is important to point out that in a PL/I or a Pascal environment, form *kij* is preferred because of the row orientation. For Algol 60 and Ada no specification in the language describes the array storage in memory, and Algol 68 allows for either.

In the final two forms, *ikj* and *jki*, we see that the access patterns look like



Form ikj

and



Form jki

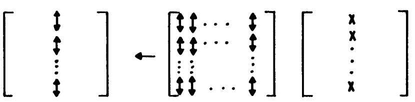
These forms use the *GAXPY* operation; that is, multiples of a set of vectors are accumulated in a single vector before the storing of that vector is required. On a machine like the Cray, algorithms that use the *GAXPY* perform at supervector speeds.

Operations such as load, multiplication, and addition can be chained together on the Cray so that, after an initial startup, one result can be delivered per cycle. The store operation, however, cannot be chained to any operation since one cannot then guarantee the integrity of the data being loaded. That is, in a sequence like vector load—vector multiplication—vector addition—vector store, the vector load will still be sending its result to a vector register when the vector store is started, and a possible bank conflict may arise (or worse yet, an element of a vector may be referenced first in a load and next in a store operation, but because the increment or stride value on the vector in the store operation may differ from that in the load, the store may overwrite an element before it is loaded). Since such a check is not made in the hardware of the Cray-1, the store operation is prohibited from chaining.

In terms of performance on the Cray-1, vector is 4 times faster than scalar, and supervector 4 times faster than vector, so from scalar to supervector one can expect performance to be 16 times faster. (These numbers are coarse and are meant to reflect realistic values that have been gathered from experience with various linear algebra programs on the Cray-1; they are not optimum values, [2].)

In terms of the six algorithms being discussed for matrix multiplication, only the forms that use a *GAXPY* (forms *ikj* and *jki*) can achieve a supervector speed; and of them only the form *jki* performs well in a Fortran environment because of its column orientation.

Up to this point we have not been concerned with the lengths of vectors. We will now assume that vector registers have some length, say *vl* (in the case of the Cray-1, the vector length is 64). When matrices have row and/or column dimensions greater than *vl*, an additional level of structure must be imposed to handle this situation. (A machine like the Cyber 205 does not have this problem since it is a memory to memory architecture; data is streamed from memory to the functional units and back to memory.) Specifically with algorithm *jki*, columns of the matrix can be segmented to look like the following:



where each segment is of length *vl* except for the last segment which will contain the remaining elements. The two ways to organize the algorithm are 1) to sequence column segments across the matrix *A*, developing a single complete segment of *C*, or 2) to sequence column segments down a column of *A*, developing a partial segment of *C*.

These algorithms have an additional looping structure to take care of the vector segmentation needed to process *n* elements in a vector register of length *vl*. The two matrix multiplication algorithms for dealing with vectors of length greater than *vl* are shown below:

```
for $j = 1$ to $p$
  for $i = 1$ to $m$
    $c_{ij} = 0$
  end
  for $k = 1$ to $n$
    for $l = 1$ to $m$ by $vl$
      for $i = l$ to $\min(l + vl - 1, m)$
        $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
      end
    end
  end
end
```

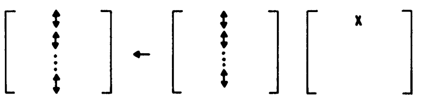
Form jki by block column

```
for $j = 1$ to $p$
  for $l = 1$ to $m$ by $vl$
    for $i = 1$ to $\min(l + vl - 1, m)$
      $c_{ij} = 0$
    end
    for $k = 1$ to $n$
      for $i = l$ to $\min(l + vl - 1, m)$
        $c_{ij} = c_{ij} + a_{ik} * b_{kj}$
      end
    end
  end
end
```

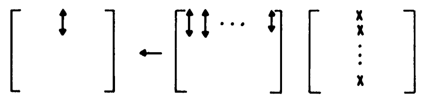
Form jki by block row

The aims are to keep information in vector registers and to store the results only after all operations are sequenced on that vector. The block row form accomplishes this goal: A vector register is used to accumulate the sum of vector segments over all columns of a matrix. In the block column form, however, after each multiple of a vector is added in a register, it is stored and the next segment is processed.

Graphically we have the following situation:



Block column



Block row

Clearly we want to maintain the vector segment of C in a register to minimize data traffic to and from memory, but mainly we want to avoid storing the information. The block column algorithm does not allow the repeated accumulation of a column of C, since a register cannot hold an entire column of the matrix C. It is interesting to note that while sequential access is essential in a paged environment, a "Cray-like" architecture does not suffer from paging problems. Therefore, the block row provides the best possible situation and will lead to a very high rate of execution.

5. Linear equations. We now turn to one of the procedures that probably uses the most computer time in a scientific environment: solving systems of equations. We will concentrate our analysis on solving systems of equations by Gaussian elimination or, more precisely, the reduction to upper triangular form by means of elementary elimination. To retain clarity, we initially will omit the partial pivoting step; nevertheless, pivoting is vital to ensure numerical stability and will be dealt with later.

Gaussian elimination usually transforms the original square matrix A into the product of two matrices L and U, so that

$$A = LU.$$

The matrices L and U have the same dimension as A; L is unit lower triangular (i.e., zeros above the diagonal and the value one on the diagonal), and U is upper triangular (i.e., zero below the diagonal). The algorithm that produces L and U from A, in general, overwrites the information in the space that A occupied, thereby saving storage. The algorithm, when given the matrix in an array A, will produce in the upper triangular portion of the array A the information describing U and in the lower triangular portion below the diagonal the information describing L.

Like the matrix multiplication algorithm, the algorithm for Gaussian elimination can be described as follows:

Generic Gaussian elimination algorithm.

```
for ________
   for ________
      for ________
         $a_{ij} = a_{ij} - (a_{ik} * a_{kj})/a_{kk}$.
      end
   end
end
```

As before, we have intentionally left blank the information describing the loops. The loop indices will have variable names i, j, and k, but their ranges will differ. Six permutations are possible for arranging these loop indices.

If we fill in the blanks appropriately, we will derive six algorithmic forms of Gaussian elimination. Each form produces exactly the same matrices L and U from A; even the roundoff errors are the same. (We have omitted some numerically critical features, like pivoting, in order to keep the structure simple.)

These six algorithms have radically different performance characteristics on a "Cray-like" machine. As in the case of matrix multiplication, we can best investigate these differences by analyzing the patterns of data access.

Forms *ijk* and *jik* are variants of the Crout algorithm of Gaussian elimination [17]. (To be more precise, these are variants of the Doolittle algorithm, but for simplicity we refer to them as Crout.) The Crout algorithm can be characterized by its use of inner products to accomplish the decomposition. At the ith step of the algorithm the matrix has the form shown in Fig. 4.

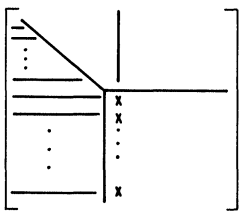


FIG. 4.

For form *ijk*, inner products are formed with the ith column of U_i and rows 1 through n of the formed L to create the new elements of L_i. A similar procedure is followed for form *jik*, but with the roles of U and L interchanged. Notice that since inner products are performed, if one accumulates the result in extended precision, a more accurate factorization will result.

Form *kij* is the form most often taught students in a first course in linear algebra. In brief, a multiple of a given row is subtracted from all successive rows to introduce zeros between the diagonal elements of the given row *k* to the last row (see Fig. 5). After zeroing out an element, the zeroing transformation is applied to the remainder of the matrix. This algorithm references elements by rows and, as a result, is not really suited for Fortran.

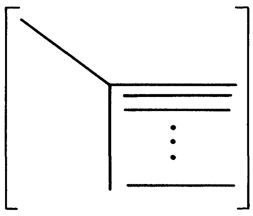


FIG. 5.

Form *kji* is the column variant of the *kij* row algorithm as described in [15] and is used in the LINPACK collection [3]. This form is organized so that sweeps are made down a column instead of across a row of the matrix. As with form *kji*, a zeroing transformation is performed, and then that transformation is applied to the remainder of the matrix. Since the operations occur within columns of the matrix, this algorithm is more attractive in Fortran than either of the previous approaches. The basic operation is a *SAXPY*. Since updates are made to all remaining columns of the matrix, there is no opportunity to accumulate the intermediate result in extended precision.

The final two algorithms, forms *ikj* and *jki*, differ from forms *kij* and *kji* primarily in how transformations are applied. Here, before zeros are introduced, all previous transformations are applied; and only afterwards is the zeroing transformation applied. Specifically, in form *jki*, columns 1 through $i - 1$ are applied to column i before zeros are introduced in column i. This is the algorithm described in Fong and Jordan [6]; see Fig. 6.

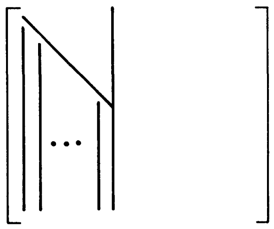


FIG. 6.

The basic operation of this algorithm is a *GAXPY*. As the previous transformations are applied, a column of the matrix can remain in a register and need not be stored in memory. Thus, the algorithm has the possibility of accumulating the intermediate vector results in vector extended precision.

Since column orientation is preferable in a Fortran environment, we will concentrate our attention on only three forms: *jik*, *kji*, *jki* which we will refer to as *SDOT*, *SAXPY*, and *GAXPY* respectively. If we were dealing with some other language where row orientation was desirable, then the other three forms would be appropriate for discussion.

Notice that in choosing these forms, only in the Crout variants do we have to consider a "stride" or worry about memory bank conflicts.

Appendix A lists the three forms implemented in Fortran 77.

We turn now to implementing the three forms in a pseudovector assembly language (PVAL). This language is idealized to the extent that we assume that vector segmentation and length of vector registers are not a problem. Segmentation is, nevertheless, an important detail on some vector computers, and we will discuss it later. We also do not label vector registers, but refer instead to their content.

In PVAL, then, we have the following instructions:

INCR	$i = n1$ **to** $n2$	*loop on* i *from* $n1$ *to* $n2$
LOOP	i	*scope of loop on* i
L	s	*scalar load*
ST	s	*scalar store*
VL	v	*vector load*
VST	v	*vector store*
VA	$v \leftarrow v + v$	*vector-vector addition*
VSM	$v \leftarrow v * s$	*vector-scalar multiplication*
VSD	$v \leftarrow v/s$	*vector-scalar division*
VIP	$s \leftarrow s + v \cdot v$	*initialized vector inner product.*

Appendix B lists the translations of the three algorithms into PVAL.

As one might expect, the three forms perform the same number of arithmetic operations; see Table 1.

TABLE 1

	number of divisions	number of multiplications
SAXPY version	$\sum_{k=1}^{n-1} \sum_{i=k+1}^{n} 1$	$\sum_{k=1}^{n-1} \sum_{j=k+1}^{n} \sum_{i=k+1}^{n} 1$
GAXPY version	$\sum_{j=1}^{n-1} \sum_{i=j+1}^{n} 1$	$\sum_{j=1}^{n} \sum_{k=1}^{j-1} \sum_{i=k+1}^{n} 1$
SDOT version	$\sum_{i=1}^{n} \sum_{j=2}^{i} 1$	$\sum_{i=1}^{n} \sum_{j=2}^{n} \left(\sum_{k=1}^{j-1} 1 + \sum_{k=1}^{i-1} 1 \right)$
Total for each	$\frac{1}{2}(n^2 - n)$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$

A more important quantity to measure with respect to vector machines is the memory traffic, i.e., the loads and stores. In most situations the arithmetic will actually be *free;* just the time spent in loading and storing results is actually observed, since the arithmetic is often hidden under the cost of the loads and stores. We will use the word "touch" to describe a load or a store operation. By examining the PVAL code found in Appendix B we can count the vector load and store instructions used in the various versions of *LU* decomposition; see Table 2.

In counting vector and scalar touches, some optimization has been performed. Vectors are retained as long as possible in registers before storing. In sequences of instructions where a scalar and vector are referred to and the scalar is located adjacent to the vector, a vector load is performed of length one greater than the required vector; this procedure saves the scalar load operation, thereby hiding the cost in the vector load. This

TABLE 2
Summary of loads and stores in LU decomposition

	SAXPY version	*GAXPY* version	*SDOT* version
Total Touches	$\frac{2n^3}{3}+\frac{n^2}{2}-\frac{n}{6}$	$\frac{n^3}{3}+\frac{3n^2}{2}-\frac{5n}{6}$	$\frac{n^3}{3}+\frac{3n^2}{2}-\frac{5n}{6}$

optimization assumes that we can extract scalars from the vector registers; some machines may require additional work to do the extraction.

We notice that the *SAXPY* approach has almost twice as many touches as the *GAXPY* or *SDOT* approach. Most of this additional activity comes from vector store operations, which will have a dramatic effect in machines where vector stores disrupt the flow of computations. Even on a machine like the Cray X-MP, which has hardware enhancements to avoid this bottleneck found on the Cray-1, the *GAXPY* version never hinders performance. Experience on the Cray X-MP has shown that performance is actually increased when a *GAXPY* is used; this results from fewer bank conflicts and poor code generation by the compiler for the *SAXPY* version.

An important distinction to remember when comparing *GAXPY* and *SDOT* is that the fundamental operation is different; *GAXPY* operates on vectors and produces a vector, while *SDOT* operates on vectors and produces a scalar. The total number of touches is roughly the same in the two algorithms.

We will next examine the execution times of the three algorithms. The sequence of instructions in the innermost loop of the vector code is different for each implementation. For the various versions the instructions are

L	L	L
VL	VL	VL
VSM	VSM	VIP
VA	VA	ST
VST		
SAXPY	*GAXPY*	*SDOT*

We are interested in how these sequences are affected in different environments. Specifically, we will look at four architectural settings:

No chaining or overlapping of operations.

Chaining of vector loads, vector addition, vector-scalar multiplication, but not vector store.

Chaining of vector loads, vector addition, vector-scalar multiplication and vector stores.

No chaining of operations, but overlapping of vector loads, vector addition, vector–scalar multiplication and vector stores.

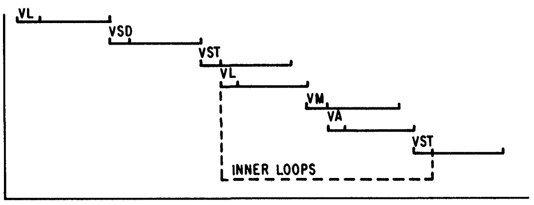
Throughout the different architectural settings we assume that each vector instruction is pipelined in the sense that, after an initial startup, results are delivered at one per cycle.

Let us see how the architecture and the algorithm combine to give a measure of performance. Consider first a machine that allows no chaining or overlapping of instructions. The time taken by each algorithm then is the sum of the arithmetic time and the number of touches. Since *GAXPY* and *SDOT* touch the data half as much as *SAXPY*, they take less time. If the inner product is implemented in a way that allows the multiplication and addition to proceed in tandem, then the *SDOT* approach is fastest. From the timing diagrams it is clear that *SAXPY* takes four chimes, *GAXPY* three chimes, and *SDOT* only two chimes. (We assume that one vector has been prefetched and remains in a register during the course of the operation.)

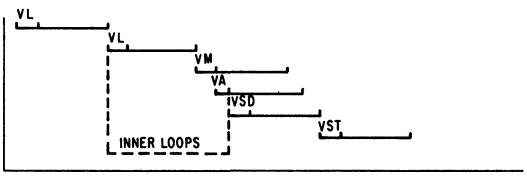
The situation is different on a machine that allows chaining. While the *GAXPY* algorithm achieves supervector speeds, *SAXPY* only gives vector speeds; i.e., it takes two chimes. The reason is simple: In *GAXPY* the vector store is outside the innermost loop, while in *SAXPY* it is inside the loop. It is very difficult to analyze *SDOT*. The timing diagram shows that *SDOT* and *GAXPY* both take one chime; the difference in time is due to the different amounts of overhead. On the Cray-1, *SDOT* has the potential to seriously degrade performance because of memory bank conflicts arising from row access patterns.

An interesting alternative to chaining is overlapping. Recall that on the Cray-1, chaining will not occur if there are memory bank conflicts. The problem can be avoided if memory access is overlapped with arithmetic. This approach allows data to be loaded into the vector registers at an irregular rate while guaranteeing that the arithmetic units get one input per machine cycle. The effect on timing can be seen by the first two passes through the inner loops of *SAXPY* and *GAXPY*; see Fig. 7.

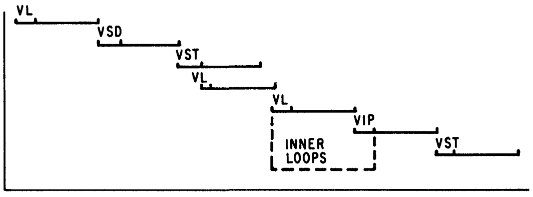
SAXPY takes two chimes per pass plus *vl* machine cycles to get started; *GAXPY* and *SDOT* each take only one chime plus *vl* cycles to start. Although there is some overhead that is not needed on a machine that chains, overlapping will reach vector or super-vector speeds in situations that will not chain. Notice that we have allowed loads and stores of



SAXPY timing diagram with overlapping.



GAXPY timing diagram with overlapping.



SDOT timing diagram with overlapping.

FIG. 7.

TABLE 3

Architecture	*SAXPY*	*GAXPY*	*SDOT*
Sequential	4	3	2
Chain loads	2	1	1
Chain loads/stores	1	1	1
Overlap operations	2	1	1

independent data to proceed simultaneously. If this feature were not allowed, then *SAXPY* would take three chimes.

These results are summarized in Table 3. The entries are the number of chimes needed to complete the inner loop. However, the table tells only part of the story. Even though *SDOT* looks as fast as *GAXPY* on a machine that chains, it can be slower on the Cray-1 because of inefficient computation of inner product.

6. Segmentation of loops. Vector segmentation occurs when the vectors are longer than the vector registers. As noted earlier, segmentation requires an extra loop. To study segmentation, we extend the definitions of the PVAL instructions INCR and LOOP to take an argument VSEG. Thus, the instruction

$$\text{INCR} \quad vseg = m \textbf{ to } n$$

breaks vector loads and stores from element m to element n into segments of length vl. The first vl elements will be loaded or stored on the first pass, the next vl elements on the next pass, and so on. The vector load then looks like

$$\text{VL} \quad a_{vseg,j},$$

which will load the next segment of column j into the register.

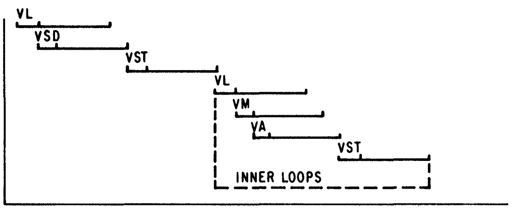
If the segmentation is in the inner loop, then none of the algorithms is faster than vector speed. The reason can be seen from the PVAL code for the inner loops of *GAXPY*, given in Appendix B. The vector store is inside the loop on k instead of outside. (On a paged machine architecture this would not be the optimum choice, see [4] for details.) The timing diagram for the inner loop is given in Fig. 8.

If the segmentation is done outside the loop on j instead, the vector store can be kept outside the inner loop. Supervector speed can then be achieved. Just as with the matrix multiplication, we want to hold the result in the register as long as possible. Figure 9 shows the optimal access pattern.

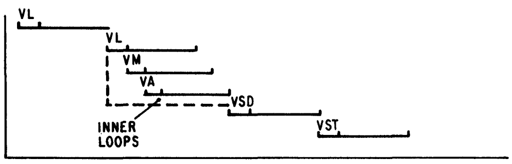
The PVAL code in Appendix C has been modified to allow for segmentation. In each case, the segmentation has been chosen to minimize the overhead. The notation $a_{vseg>j,k}$ means that the corresponding operation is to be performed only for rows $j+1$ through the end of the segment. In addition, operations on zero-length vectors are ignored.

In summary, we can make the following statements about performance. If we can be assured that memory bank conflicts will not occur, and if we use *GAXPY*, then chaining as implemented on the Cray-1 gives nearly optimal performance. If, on the other hand, memory bank conflicts are frequent or the vectors are sufficiently long, then overlapping is a viable alternative. Here either *SAXPY* or *GAXPY* will run nearly as fast as when chaining occurs. Some machines, such as array processors, have efficient inner-product hardware; on such machines, *SDOT* will be fastest.

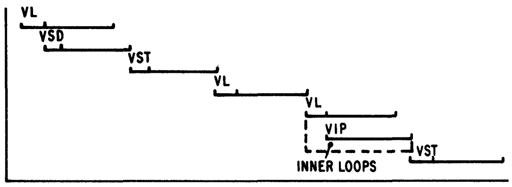
Notice that *GAXPY* is fast or almost as fast as *SDOT* and that *SAXPY* is always slower. In addition, *SAXPY* touches the data twice as much, which can cause memory



SAXPY timing diagram with chaining.



GAXPY timing diagram with chaining.



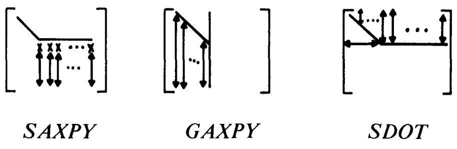
SDOT timing diagram with chaining.

FIG. 8.

bandwidth bottlenecks. Therefore, if there is any uncertainty about the characteristics of the machine, the coding should probably be done using *GAXPY*. To gain the most in terms of performance on the Cray, simple vector operations are not enough. The scope must be expanded to include the next level, matrix–vector operations, such as the matrix–vector multiplication of *GAXPY*.

7. Pivoting. Up to this point we have ignored pivoting in the *LU* factorization. Pivoting is necessary to stabilize the underlying algorithm. If the algorithm is applied without pivoting to a general matrix, the solution will almost certainly contain unacceptable errors. Pivoting is a relatively simple procedure; in partial pivoting an interchange is performed based on the largest element in absolute value of a column of the matrix. In algorithm *jki* the pivot procedure is performed before the scaling operation (see Appendix A).

In terms of performance, the pivoting procedure is just some additional overhead and does not significantly affect the performance. We must, however, qualify this statement



SAXPY *GAXPY* *SDOT*

FIG. 9.

somewhat. A pivoting operation involves the interchange of two complete rows of the matrix. Therefore, we must issue two vector loads and two vector stores for each vector segment in the rows. However, the segmentation is done in the outer loop, i.e., once per column, and takes only about $2n$ additional vector touches. Unfortunately, these are row accesses that may produce memory bank conflicts.

8. Conclusions. Our aim was not to rewrite an existing program but to restructure the algorithm. This is more than a matter of words, *program* or *algorithm*. In looking at a program, one has a tendency to focus on statements, detecting one vector operation here, another there. To produce a truly vectorized implementation, however one must go back to the algorithm and restructure it with vectors or matrix–vector constructions in mind. We hope in the future that compilers will routinely carry out the process of restructuring.

Looking back, we have been pleasantly relieved to see that algorithms restructured for a vector pipeline architecture perform as well or better than their predecessors on conventional machines.

Looking forward, we admit that we may again have to go through this exercise for multiprocessor machines such as the Cray X-MP, Cray-2, Cyber 2xx and the Denelcor HEP.

Appendix A. Column variants of the generic Gaussian elimination algorithm.

```
      SUBROUTINE KJI(A,LDA,N)
C
C    SAXPY
C     FORM KJI - SAXPY
C
      REAL A(LDA,N)
      DO 40 K = 1,N-1
         DO 10 I = K+1,N
            A(I,K) = -A(I,K)/A(K,K)
   10    CONTINUE
         DO 30 J = K+1,N
            DO 20 I = K+1,N
               A(I,J) = A(I,J) + A(I,K)*A(K,J)
   20       CONTINUE
   30    CONTINUE
   40 CONTINUE
      RETURN
      END
```

Form KJI

```
      SUBROUTINE JKI(A,LDA,N)
C
C     GAXPY
C     FORM JKI - GAXPY
C
      REAL A(LDA,N)
      DO 40 J = 1,N
         DO 20 K = 1,J-1
            DO 10 I = K+1,N
               A(I,J) = A(I,J) + A(I,K)*A(K,J)
   10       CONTINUE
   20    CONTINUE
         DO 30 I = J+1,N
            A(I,J) = -A(I,J)/A(J,J)
```

```
 30     CONTINUE
 40 CONTINUE
    RETURN
    END
```

Form JKI

```
      SUBROUTINE IJK(A,LDA,N)
C
C     SDOT
C     FORM IJK - DOT
C
      REAL A(LDA,N)
      DO 50 I = 1,N
         DO 20 J = 2,I
            A(I,J-1) = -A(I,J-1)/A(J-1,J-1)
            DO 10 K = 1,J-1
               A(I,J) = A(I,J) + A(I,K)*A(K,J)
 10         CONTINUE
 20      CONTINUE
         DO 40 J = I+1,N
            DO 30 K = 1,I-1
               A(I,J) = A(I,J) + A(I,K)*A(K,J)
 30         CONTINUE
 40      CONTINUE
 50   CONTINUE
      RETURN
      END
```

Form IJK

```
      SUBROUTINE JKIPVT(A,LDA,N)
C
C     GAXPY
C     FORM JKI - GAXPY
C     WITH PIVOTING
C
      REAL A(LDA,N),T
      DO 60 J = 1,N
         DO 20 K = 1,J-1
            DO 10 I = K+1,N
               A(I,J) = A(I,J) + A(I,K)*A(K,J)
 10         CONTINUE
 20      CONTINUE
C
C        PIVOT SEARCH
C
         T = ABS(A(J,J))
         L = J
         DO 30 I = J+1,N
            IF(ABS(A(I,J)) . GT . T) THEN
               T = ABS(A(I,J))
               L = I
            END IF
 30      CONTINUE
```

```
C
C         INTERCHANGE ROWS
C
          DO 40 I = 1,N
             T = A(J,I)
             A(J,I) = A(L,I)
             A(L,I) = T
 40       CONTINUE
C
          DO 50 I = J+1,N
             A(I,J) = -A(I,J)/A(J,J)
 50       CONTINUE
 60 CONTINUE
    RETURN
    END
```

Form JKI

Appendix B. Translations into PVAL.

```
INCR        $k = 1$ to $n-1$
  VL        $x_{k:n} \leftarrow a_{k:n,k}$
  VSD       $x_{k+1:n} \leftarrow -x_{k+1:n}/x_k$
  VST       $a_{k+1:n,k} \leftarrow x_{k+1:n}$
  INCR      $j = k+1$ to $n$
    VL      $y_{k:n} \leftarrow a_{k:n,j}$
    VSM     $z_{k+1:n} \leftarrow x_{k+1:n} * y_k$
    VA      $y_{k+1:n} \leftarrow y_{k+1:n} + z_{k+1:n}$
    VST     $a_{k+1:n,j} \leftarrow y_{k+1:n}$
  LOOP      $j$
LOOP        $k$
```

SAXPY

```
INCR        $j = 1$ to $n$
  VL        $x_{1:n} \leftarrow a_{1:n,j}$
  INCR      $k = 1$ to $j-1$
    VL      $y_{k+1:n} \leftarrow a_{k+1:n,k}$
    VSM     $y_{k+1:n} \leftarrow y_{k+1:n} * x_k$
    VA      $x_{k+1:n} \leftarrow x_{k+1:n} + y_{k+1:n}$
  LOOP      $k$
  VSD       $x_{j+1:n} \leftarrow -x_{j+1:n}/x_j$
  VST       $a_{1:n,j} \leftarrow x_{1:n}$
LOOP        $j$
```

GAXPY

```
INCR        $i = 1$ to $n$
  VL        $x_{1:n} \leftarrow a_{i,1:n}$
  INCR      $j = 2$ to $i$
    D       $x_{j-1} \leftarrow -x_{j-1}/y_{j-1}$
    VL      $y_{1:j-1} \leftarrow a_{1:j-1,j-1}$
    VIP     $x_j \leftarrow x_j + y_{1:j-1} x_{1:j-1}$
  LOOP   $j$
  INCR   $j = i+1$ to $n$
    VL   $y_{1:i-1} \leftarrow a_{1:i-1,\, j}$
    VIP  $x_j \leftarrow x_j + y_{1:i-1}\, x_{1:i-1}$
  LOOP   $j$
  VST    $a_{i,1:n} \leftarrow x_{1:n}$
LOOP     $i$
```

SDOT

Appendix C. PVAL code with segmentation.

```
INCR          $k = 1$ to $n-1$
  INCR        $vseg,\ k{:}n$
  VL          $x \leftarrow a_{vseg,k}$
  VSD         $x_{vseg>k} \leftarrow -x_{vseg>k}/x_k$
  VST         $a_{vseg,k} \leftarrow x$
  INCR        $j = k+1$ to $n$
    VL        $y \leftarrow a_{vseg,j}$
    VSM       $z \leftarrow x_{vseg>k} * y_k$
    VA        $y_{vseg>k} \leftarrow y_{vseg,k} + z$
    VST       $a_{vseg,j} \leftarrow y$
    LOOP      $j$
  LOOP        $vseg$
LOOP          $k$
```

SAXPY

```
INCR          $j = 1$ to $n$
  INCR        $vseg,\ 1{:}n$
    VL        $x \leftarrow a_{vseg,j}$
    INCR      $k = 1$ to $j-1$
      VL      $y \leftarrow a_{vseg>k,k}$
      VSM     $y \leftarrow y * x_k$
      VA      $x_{vseg>k} \leftarrow x_{vseg>k} + y$
    LOOP      $k$
    VSD       $x_{vseg>j} \leftarrow -x_{vseg>j}/x_j$
    VST       $a_{vseg,j} \leftarrow x$
  LOOP        $vseg$
LOOP          $j$
```

GAXPY

```
INCR          $i = 1$ to $n$
  INCR        $vseg,\ 1{:}n$
    VL        $x \leftarrow a_{i,vseg}$
    INCR      $j = 2$ to $i$
    D         $x_{j-1} \leftarrow -x_{j-1}/y_{j-1}$
      VL      $y \leftarrow a_{vseg<j-1,j}$
      VIP     $x_j \leftarrow x_j + y \cdot x_{vseg<j-1}$
```

```
        LOOP   $j$
        INCR   $j = i+1$ **to** $n$
          VL   $y \leftarrow a_{vseg<i-1,j}$
         VIP   $x_j \leftarrow x_j + y \cdot x_{vseg<i-1}$
        LOOP   $j$
         VST   $a_{i,vseg} \leftarrow x$
      LOOP     $vseg$
    LOOP       $i$
```

SDOT

REFERENCES

[1] J. M. Boyle, *Towards automatic synthesis of linear algebra programs,* Proc. Conference on Production and Assessment of Numerical Software, M. Delves and Hennel, ed., Academic Press, New York, 1980, pp. 223–245.

[2] J. J. Dongarra, *Some* LINPACK *timings on the* CRAY-1, Tutorial in Parallel Processing, R. H. Kuhn and D. A. Padua, eds., IEEE, 1981, pp. 363–380.

[3] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart, LINPACK *Users' Guide,* Society for Industrial and Applied Mathematics, Philadelphia, 1979.

[4] J. J. Du Croz, S. M. Nugent, J. K. Reid and D. B. Taylor, *Solving large full sets of linear equations in a paged virtual store,* ACM Trans. Math. Software, 7 (1981), pp. 527–536.

[5] I. S. Duff and J. K. Reid, *Experience of sparse matrix codes on the* Cray-1, Computer Science and System Division, AERE Harwell CSS 116, October 1981.

[6] K. Fong and T. L. Jordan, *Some linear algebra algorithms and their performance on* CRAY-1, Los Alamos Scientific Laboratory, UC-32, Los Alamos, NM, June 1977.

[7] F. G. Gustavson, *Some basic techniques for solving sparse systems of linear equations,* Sparse Matrices and Their Applications, D. J. Rose and R. A. Willoughby, eds. Plenum, New York, 1972, pp. 41–52.

[8] D. Heller, *A survey of parallel algorithms in numerical linear algebra,* this Review, 20 (1978), pp. 740–777.

[9] R. W. Hockney and C. R. Jesshope, *Parallel Computers,* J. W. Arrowsmith, Bristol, 1981.

[10] T. L. Jordan, Private communications, 1982.

[11] P. M. Kogge, *The Architecture of Pipelined Computers,* Academic Press, New York, 1981.

[12] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, *Basic linear algebra subprograms for Fortran usage,* ACM Trans. Math. Software, 5 (1979), pp. 308–371.

[13] R. Levine, *Supercomputers,* Scientific American, January 1982, pp. 118–135.

[14] C. B. Moler, Private communication, 1978.

[15] ———, *Matrix computations with Fortran and paging,* Comm. ACM, 15 (1972), pp. 268–270.

[16] R. M. Russell, *The* CRAY-1 *computer system,* Comm. ACM, 21 (1978), pp. 63–72.

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