Programming style on the IBM 3090 Vector Facility considering both performance and flexibility

by H. Samukawa

To obtain high performance from the IBM 3090 Vector Facility, we must investigate vector instruction constructs in terms of the loop context of the application algorithm. We exemplify the method by linear algebra subroutines for basic matrix operations and a linear equation solver. In these examples, we clarify the mathematical meaning that each loop is computed by analyzing the loops in terms of a generic algorithm. This analysis helps us to achieve optimal loop selection. We then obtain additional performance gain by considering cache capacity. These procedures suggest that there are three levels of performance classification. They also show that program structure yields great benefits in terms of performance and generality of the program.

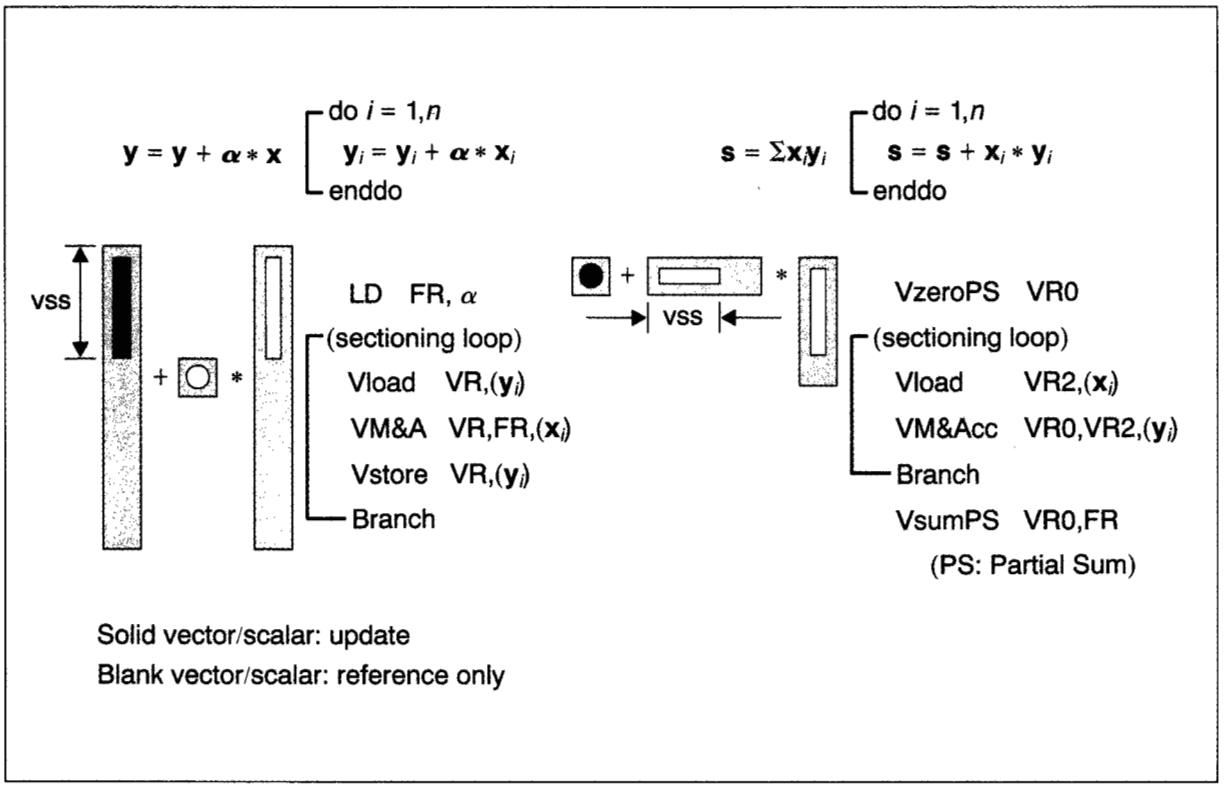
High-performance computing is now a common requirement in many engineering and scientific environments. Vector processors have become very popular among engineering and scientific users, because programmers can use vector processors with little or no knowledge of those machines. This is due to the built-in capability of vectorizing compilers. However, there are many applications in which careful tuning can yield great performance enhancements. Numerical methods such as linear algebra are among these applications. In such applications, tuning methods differ according to the types of vector machines, especially according to their architecture and hardware implementations.

In this paper, we use examples of typical and fundamental matrix operation routines to show the properties to consider in tuning the IBM 3090 with Vector Facility (VF).[1] Discussed first is the importance of examining the possibilities for changing the calculation sequence. After discussing tuning methods, we present ways of separating system-dependent properties from the application programming. If we had to develop and maintain programs involving such properties, much effort would be required for further program modifications or for future hardware changes.

Prior to discussing numerical algorithms, we introduce the following vector architecture notation, and the instruction notation used in this paper, which we hope will be helpful in providing an intuitive understanding of how VF works.

The following vector architecture concepts are used in this paper:

Figure 1 Scalar multiple of a vector and dot product



- The virtual storage (VS) system discussed here has 31-bit addressing capability.
- The vector register in the vector architecture presented here is one of 16 vector registers, the length of which is described by the vector section size $(VSS)^2$ parameter.
- The instructions for vector operations can have up to three operands, one of which can point to a vector in memory as a source operand. An instruction called the *compound-operation instruction* provides two operations, such as multiplication and addition/subtraction or multiplication and accumulation.

In this paper, pseudo-vector instructions are used to show how VF works and to provide a rough estimate of performance.

The following are pseudo-instructions used in this paper:

- Vload VR, $(\mathbf{x}_i)$ — Load $(\mathbf{x}_i)$ to VR (vector register), where $(\mathbf{x}_i)$ is a section of vector $\mathbf{x}_i$ in memory.
- Vstore VR, $(\mathbf{x}_i)$ — Store the current value of VR.
- VM&A VR, FR, $(\mathbf{x}_i)$ — Multiply a section of $\mathbf{x}_i$ by FR (floating-point register) and add to VR; the result remains in VR, i.e., VR ⇐ VR + FR * $(\mathbf{x}_i)$.

(Note that the strict mnemonic codes such as VLD, VSTD, and VMADS require some associated scalar instructions.[2])

A FORTRAN program to compute a scalar times a vector plus a vector—known as DAXPY in the basic linear algebra subprogram (BLAS) interface—can be expressed using these pseudo-instructions, as shown in Figure 1. Because $(\mathbf{x}_i)$ and $(\mathbf{y}_i)$ are only sections of each vector, these instructions must be executed repeatedly to complete logical vector length n, where n is greater than VSS. The sectioning loop in Figure 1 is the place where control is transferred from the Branch; this loop is generated by a VS FORTRAN compiler. Details of the sectioning loop are discussed in Reference 3.

Figure 1 also illustrates another example of the dot product:

• VM&ACC	VR0,VR2, (x_i)	Multiply VR2 by (x_i) and accumulate the successive products, which are called partial sums, in VR0.
• VSUMPS	VR0, FR	Sum the partial sums; the result is in FR.
• VZEROPS	VR	Clear the partial sums from VR0.

Though the vector architecture defines 171 vector instructions, these six pseudo-vector codes are sufficient for the scope of this paper.

When these programs are executed, vector instructions work at a peak rate of one element per cycle after start-up time has elapsed. Two important elements should be added in a discussion of performance: (1) The execution of vector instructions does not overlap; (2) all memory references are made through the cache (high-speed buffer) in the central processor. Also, the size of cache and length of vector register are 64K bytes (8K double words) for each processor and 128 bytes in the case of the 3090 Model E.

In the following section, examples of linear algebraic problems are discussed and computation performance is compared for various orderings of the pseudo-code. Finally, measured performance comparisons are made. The tuning approach mentioned in this paper might be used in applications other than linear algebra if they have similarities to vector tuning, such as loop nesting and triangular matrix.

Program loops and performance

The most important characteristic of the 3090 VF as a vector machine is that it is a virtual storage machine (VS) with vector registers. In this section, we discuss the relationship between program loops and performance that takes advantage of these characteristics.

Double-loop procedure. Programs with a single loop work as described in the Introduction. When loop nesting occurs, the 3090 VF enhances the million-floating-point-operations-per-second (MFLOPS) performance by eliminating vector load/store instructions within the innermost loop. In other words, reusing the vector register enhances performance. For example, consider the matrix–vector product $\mathbf{y} = \mathbf{y} + \mathbf{A} * \mathbf{x}$, where $\mathbf{y}$ is a vector with m elements, $\mathbf{A}$ is a matrix with m rows and n columns, and $\mathbf{x}$ is

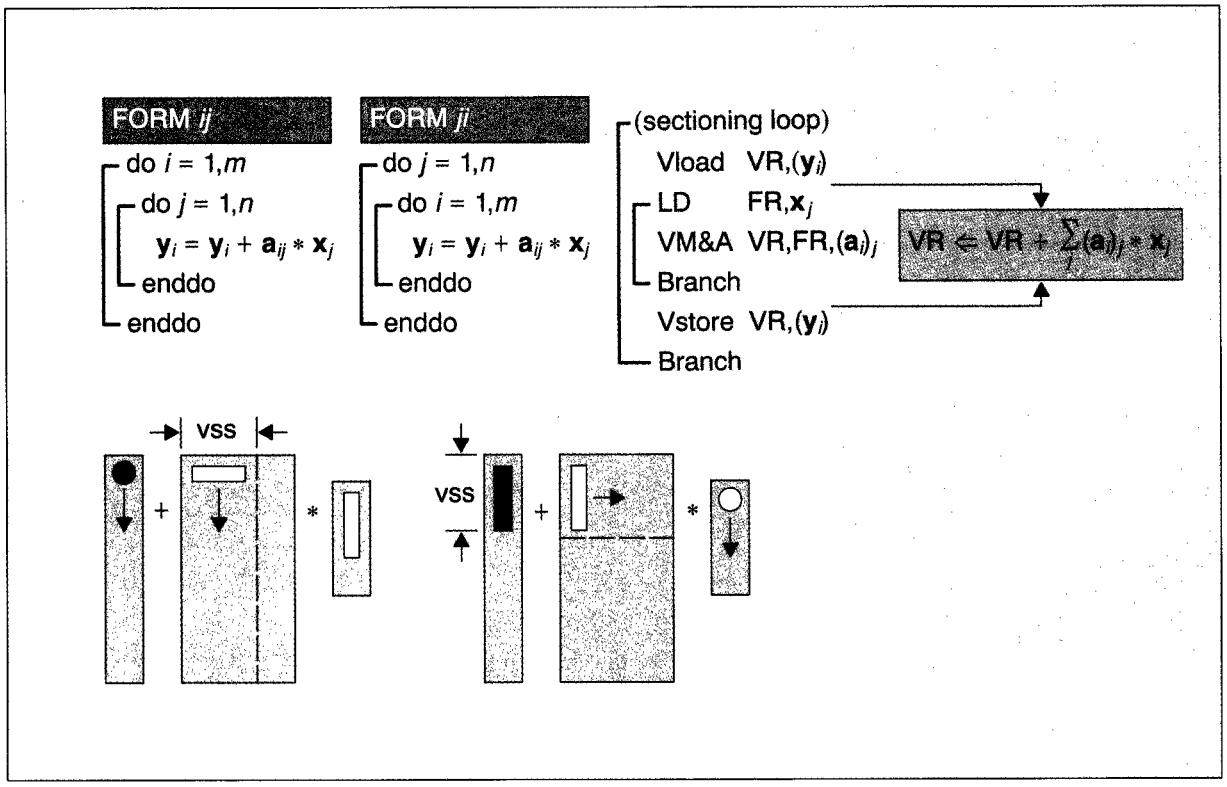
When loop nesting occurs, the 3090 VF enhances performance by eliminating vector load/store instructions within the innermost loop.

a vector with n elements. We write the matrix–vector product as a generic algorithm, as discussed in Reference 4:

```
do _ = _ , _
 do _ = _ , _
    y_i = y_i + a_ij * x_j                (1)
  enddo
enddo
```

We can substitute the loop indices i, j and termination points of the loop indices m, n in the underlined blanks in Equation (1). Two choices are possible: form ij or form ji, as illustrated in Figure 2. In this figure, sectioning loops that result from the limited length of the vector registers are considered. (Note that the steps introduced by the algorithm are not the same as the steps of the FORTRAN program itself. Since VS FORTRAN V2 can vectorize the outer loop, form ij is changed to form ji automatically.) In this paper, we refer to an index that appears on the right side and does not appear on the left (such as index j in the example) as a *dummy index*. The innermost loop computes the dot product when a dummy index is used as an index of the innermost loop. On the other hand, when a dummy index is used as an index of the outer loop, it computes a vector–scalar product of the type known as "two vector operations and one vector-memory reference." In Figure 2, this corresponds to the two floating-point operations of VM&A and one vector-memory reference ($\mathbf{a}_{ij}$). Both form ij and form ji have only one pseudo-vector

Figure 2 Matrix–vector multiplication



instruction in the innermost loop, but form *ij* suffers performance degradation in the case of large matrices. Because the FORTRAN convention of storing matrices in column order requires the access pattern of $\mathbf{a}_{ij}$ skipping m elements, the cache cannot be used efficiently. This is usually called "stride m." On the other hand, form *ji* becomes stride 1 (sequential memory reference), and is faster than form *ij*.

In the scalar computation environment, selecting the dummy index as the innermost loop index eliminates store instruction from the innermost loop by accumulating results on the floating-point register. While in the vector computation environment, because the innermost loop is vectorized, selecting a dummy index as the outer loop index eliminates vector load and vector store instructions by accumulating results in a vector register. This can be described as VR $\Leftarrow$ VR + $\sum_j (\mathbf{a}_i)_j * \mathbf{x}_j$ in Figure 2; i.e., the double-loop configuration of form *ji* is translated into the codes with loop of LD, VM&A, and Branch, and carries out double-loop calculations over a section. The first difference between a vector and a scalar appears here, when we consider hierarchical programming organization. That is, in the scalar computation, a large performance difference may not appear if we call a single-loop subroutine from a double-loop procedure as follows:

```
do j = 1,n
   call DAXPY (m, X(j), A(1, j) 1, Y, 1)
enddo
```

However, in the vector computation, we cannot take advantage of the "two vector operations and one vector-memory reference" from this programming. Thus, we must prepare a double-loop subroutine as a stand-alone, lowest-level subroutine.

Triple-loop procedure. We now look at an operation in linear algebra, the process of matrix–matrix multiplication and addition:

$$\mathbf{c}_{ij} = \mathbf{c}_{ij} + \sum_{k=1}^{m} \mathbf{a}_{ik} \cdot \mathbf{b}_{kj}, \qquad (2)$$

where $i = 1,l,\ j = 1,n$.

We write Equation (2) as a generic algorithm as follows:

```
do _ = _, _
 do _ = _, _
    do _ = _, _                    (3)
      c_ij = c_ij + a_ik * b_kj
    enddo
  enddo
enddo
```

Figure 3 shows that six (=3!) permutations are possible for arranging the three-loop indices, as discussed in Reference 4. In Figure 3, operations carried out by the inner double loop are illustrated, but the sectioning loop is ignored. Among these six forms of matrix–matrix multiplication, form *jki* is the fastest because both "two vector operations and one vector-memory reference" and "sequential memory reference" are achieved. That is, form *jki* in Figure 3 shows that instruction construction of the inner double loop becomes the same as form *ji* of the matrix–vector product. (We can ensure this by writing $\mathbf{c}_i = \mathbf{c}_i + \sum_k \mathbf{a}_{ik} * \mathbf{b}_k$ and dropping the outermost index *j*.)

In the 3090, vector instruction execution speed is faster when the operand vector exists within cache than when it is absent and must be transferred from memory.

In the 3090, vector instruction execution speed is faster when the operand vector ($\mathbf{a}_{ik}$) exists within cache than when it is absent and must be transferred from memory.[5] By considering this property, we can create a triple-loop routine that is faster than a double-loop routine in terms of MFLOPS. To be concrete, the progress of loop index *k* of form *jki* is interrupted when the cache is filled with the referenced data $\mathbf{a}_{ik}$, so that the outer-loop index *j* goes forward, as illustrated in Figure 4.

Memory hierarchy and hierarchical programming. We can enhance this performance of the vector machine with hierarchical memory. Figure 5 sketches differences among three memory hierarchy and loop configurations. In the sketch, the innermost loops compute the same scalar multiple of a vector. A single loop requires three vector instructions, Vload, VM&A, and Vstore, but a double loop reduces them to one. A comparison of vector-machine operations is sketched in the right-hand portion of the figure. Solid arrows represent an instruction that requires one execution cycle per vector element. In the triple-loop procedure, the operand of the instruction can remain in the cache. However, it is an advantage for the application programmer to be able to compose his program without having to consider such machine-dependent factors.

Now let us introduce an example of matrix–matrix multiplication by concealing the properties affected by the system. (See Figure 6.) We store system-dependent parameters in a common SYSTEM, such as a vector register length and cache size available for vectors in memory. Subroutine DMULAD computes the same algorithm as the parent subroutine, i.e., $\mathbf{C}' = \mathbf{C}' + \mathbf{A}' * \mathbf{B}'$, but this subroutine has no sectioning loop at all. However, the size of matrix $\mathbf{A}'$ is controlled by NXM to be NZL, where NZL is smaller than the vector register length. The kernel of this subroutine may be written as follows:

```
loopj  Vload   VR, (c_i)_j
loopk  LD      FR, b_kj
       VM&A    VR,FR, (a_i)_k where (a_i)_k mostly
               remain in cache
       Branch  loopk (k = 1,NXM)
       Vstore  VR, (c_i)_j
       Branch  loopj (j = 1,N)
```

The performance gain obtained by this cache consideration compared with the form *jki* without considering the cache is about 20 percent in the case of a square matrix of order 1000 using a 3090 Model E.

Though the computation order—considering both loop and cache capacity—seems to be rather complicated, as shown in Figure 4, we can write the program as a simple algorithm of submatrix–submatrix multiplication by constructing a hierarchical

Figure 3 Six permutations for multiplication

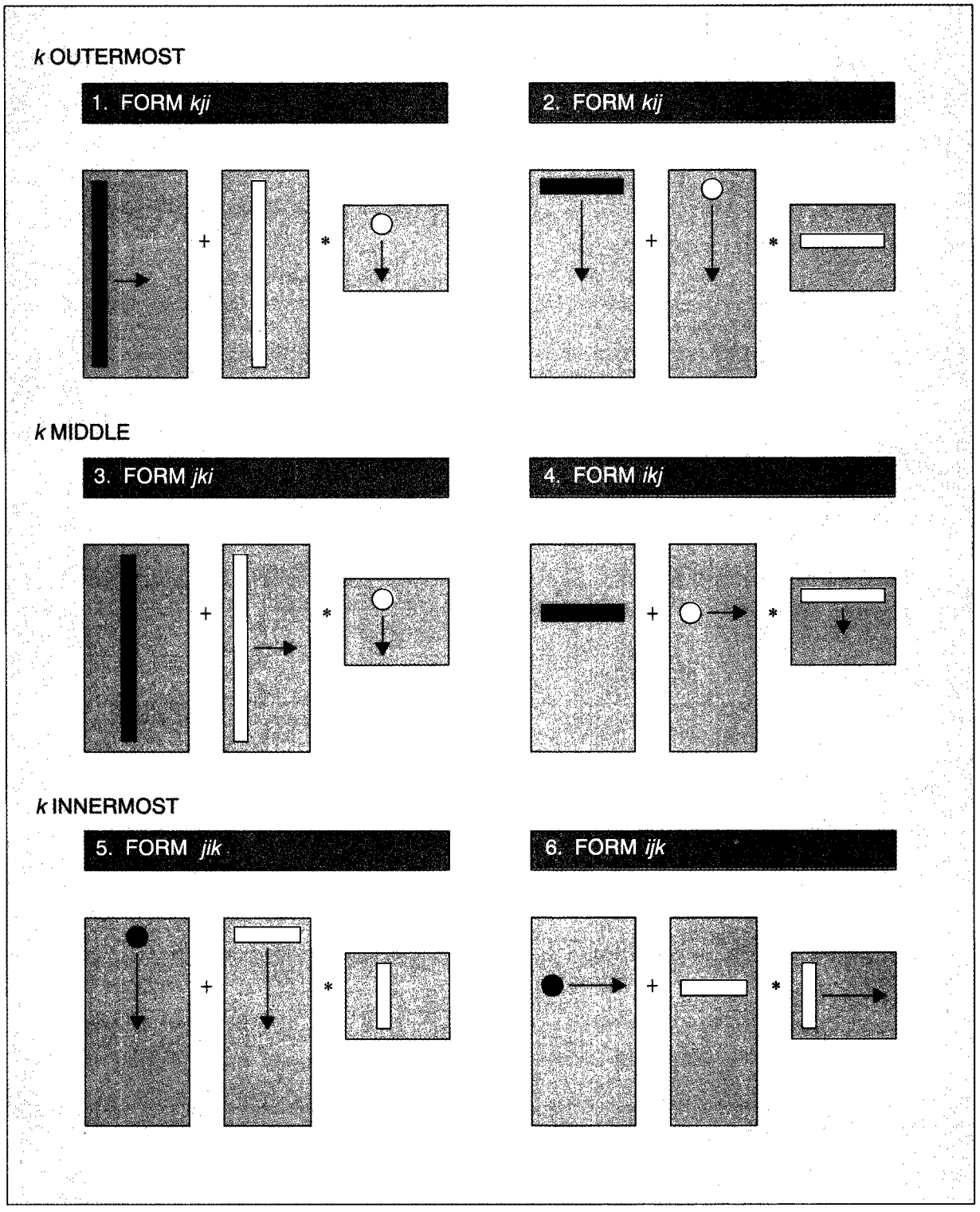
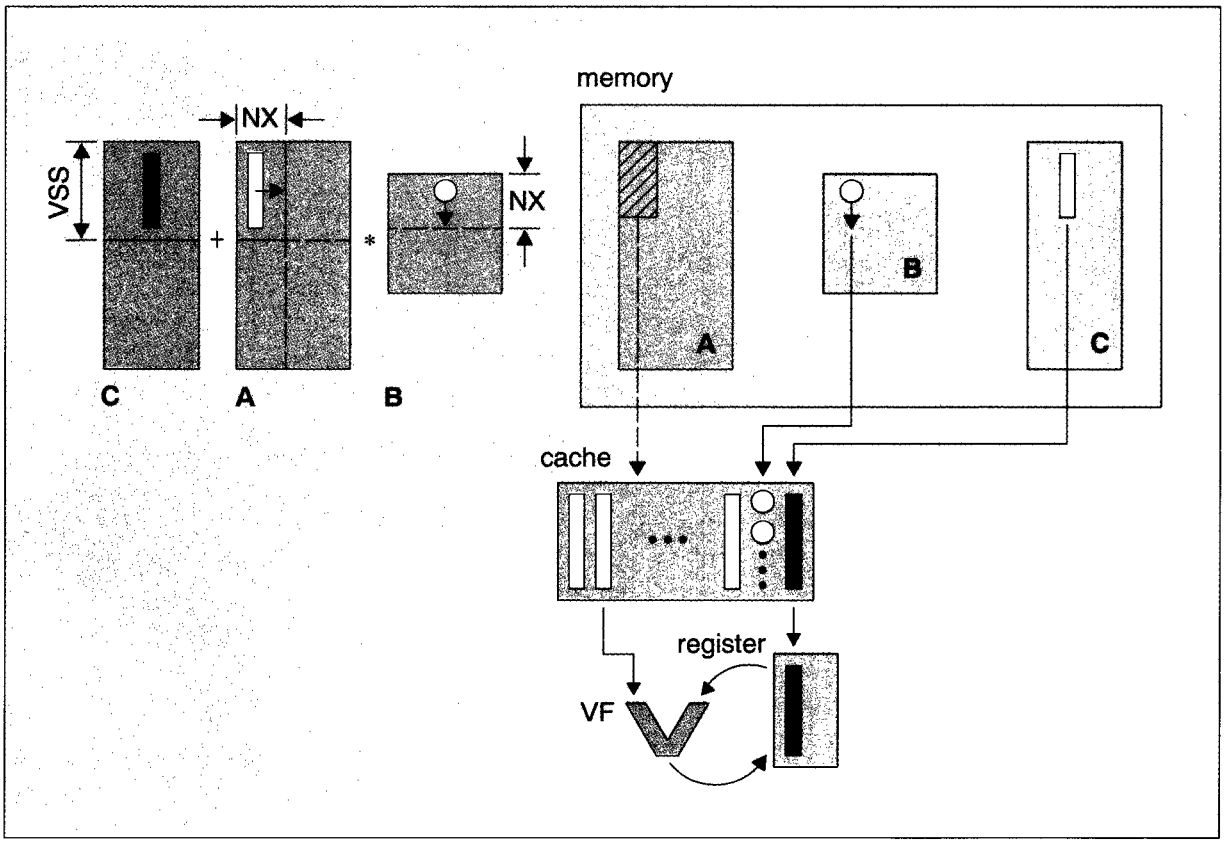


Figure 4 Method with cache consideration



program structure. In addition to this, we can cause this routine to manage changes in vector register length or cache capacity by initializing the common SYSTEM. We can also cause it to manage the scalar environment by replacing the low-level subroutine DMULAD with a scalar tuned routine. Note that the consideration of the cache capacity gives us increased performance, even in the scalar environment.

Let us summarize here the way to work with vector machines. In the scalar computation environment, we may compose our program directly to translate a given formula into FORTRAN language statements. While in the vector mode, we may achieve *super vector performance*[4] if we can find optimal codes by investigating possible reorderings of the computation sequence. This investigation step is important in the vector mode and, if possible, it would be useful in separating the properties affected by the system from application algorithms.

It may be most practical to solve the problem of the coexistence of program performance and generality by separating the program into routines of more primitive functions that resolve the differences caused by architecture or hardware. In the routines of more complicated functions, matrix–matrix multiplication routines are lower-level subroutines.

Linear equations

We now turn to the problem of solving systems of linear equations, using an algorithm usually called *triangular decomposition*, which is based on Gaussian elimination.

As in the previous discussion of matrix–matrix multiplication, we discuss loop selection first and then consider cache capacity. Gaussian elimination usually transforms the original square matrix **A** into the product of two matrices **L** and **U**:

Figure 5 Memory hierarchy and loop context

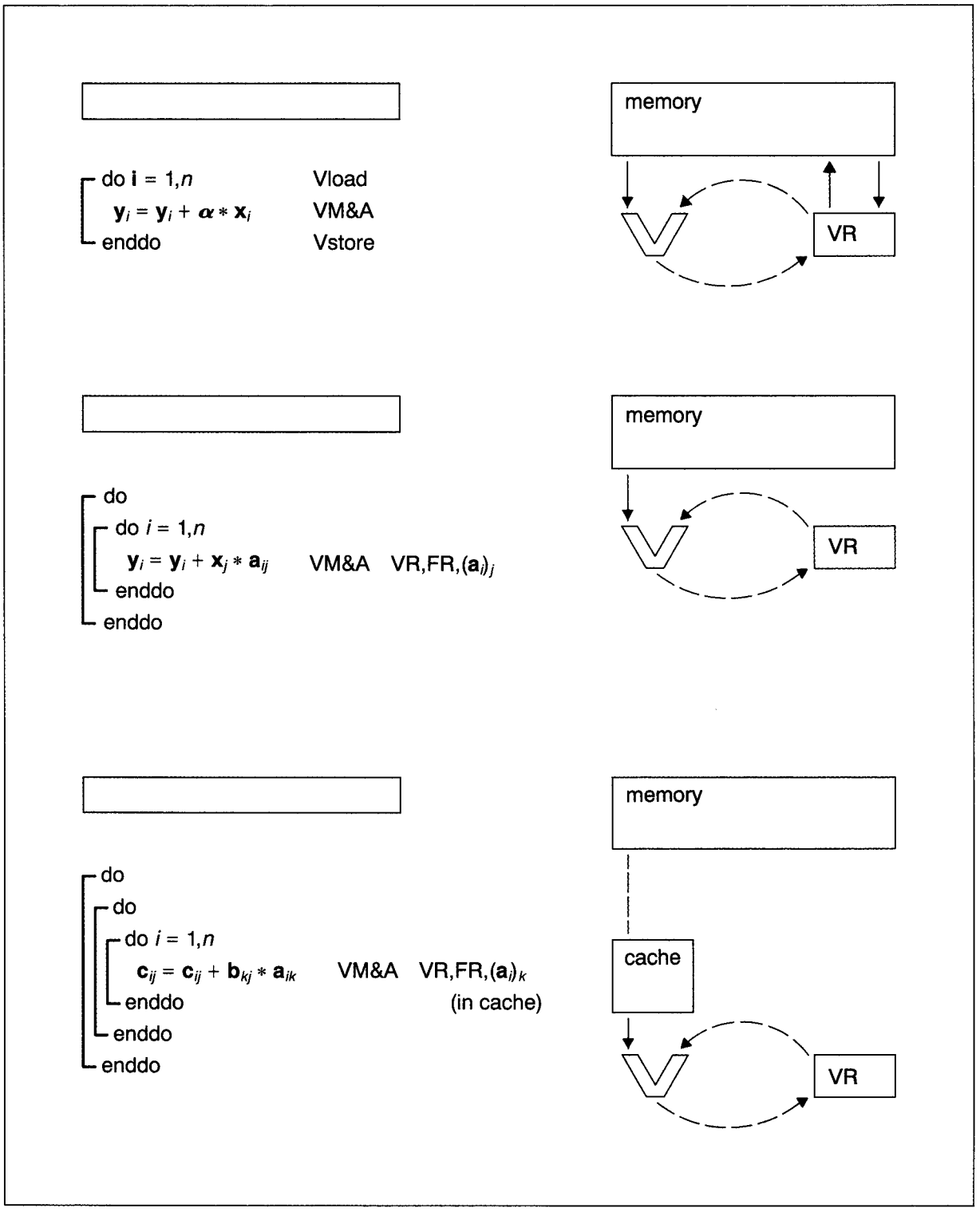
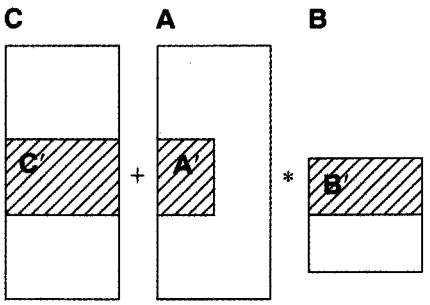


Figure 6 Subroutine DMPYAD example

```
Subroutine DMPYAD (A,lda,B,ldb,C,ldc,l,m,n)
                  (DMPYAD computes C = C + A * B)
double precision A(lda,m),B(ldb,n),C(ldc,n)
common /SYSTEM/ lvs,ncache
nz = min(lvs,l)
nx = ncache/nz
nii = (l - 1)/nz + 1
njj = (m - 1)/nx + 1
do ii = 1,nii
  i = (ii - 1) * nz + 1
  NZL = min(nz,l - i + 1)
  do jj = 1,njj
    j = (jj - 1) * nx + 1
    NXM = min(nx,m - j + 1)
    call DMULAD (A(i,j),lda,B(j,1),ldb,C(i,1),ldc,NZL,NXM,n)
  enddo          (DMULAD computes C' = C' + A' * B')
enddo
return
```



$$\mathbf{A} \Rightarrow \mathbf{LU}. \tag{4}$$

The matrices **L** and **U** have the same dimension as **A**; the matrix **L** is a unit lower triangular matrix, and **U** is an upper triangular matrix. The algorithm that produces **L** and **U** from **A**, in general, overwrites the information in the space that **A** occupied, thereby saving memory. The algorithm, when given the matrix in an array **A**, produces 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**.

Loop selection. As in matrix multiplication, the algorithm for Gaussian elimination can be written in generic form[4] as follows:

```
do _ = _, _
 do _ = _, _
    do _ = _, _
      a_ij = a_ij - a_ik * a_kj/a_kk          (5)
    enddo
  enddo
enddo
```

In this algorithm, we have omitted the partial pivoting step to retain clarity. The procedure seems to

resemble matrix–matrix multiplication, but is complicated by the fact that the ranges where loop indices (i, j, k) move are described by other indices. We first investigate the meaning of the dummy index k, then consider the possibility of loop selection. For this purpose, we digress to describe the elimination algorithm, using an elementary similarity transformation matrix.

Elementary similarity transformation matrix. Gaussian elimination can be achieved by multiplying the elementary similarity transformation matrix $\mathbf{P}_k$ repeatedly by matrix $\mathbf{A}$. Matrix $\mathbf{P}_k$ is identical to the identity matrix except for the kth column, which is

$$(0, 0, \cdots, 1, -p_{k+1,k}, -p_{k+2,k}, \cdots, -p_{n,k}).$$

Given the n by n matrix $\mathbf{A}$, choosing $p_{i1} = a_{i1}/a_{11}$ enables us to eliminate the first column as follows:

$$\mathbf{P}_1\mathbf{A} \Rightarrow \mathbf{A}^{(1)},$$

where the first column of $\mathbf{A}^{(1)}$ consists of zeros, except for the first row, which is $(X, 0, 0, \cdots, 0)$. Successive manipulations of the form

$$\mathbf{P}_{n-1}\mathbf{P}_{n-2} \cdots \mathbf{P}_2\mathbf{P}_1\mathbf{A} \Rightarrow \mathbf{A}^{(n-1)} \tag{6}$$

give an upper triangular factor $[\mathbf{A}^{(n-1)} = \mathbf{U}]$.

$\mathbf{P}_k$ has the following important properties:

- The inverse of $\mathbf{P}_k$ can be obtained simply by changing signs in the off-diagonal elements.
- No products of the elements p_{ij} occur in the product $\mathbf{P}_1^{-1}\mathbf{P}_2^{-1} \cdots \mathbf{P}_{n-1}^{-1}$, which is equal to $\mathbf{L}$:

$$\mathbf{P}_1^{-1}\mathbf{P}_2^{-1} \cdots \mathbf{P}_{n-1}^{-1} = \begin{bmatrix} 1 & & & & & \\ p_{21} & 1 & & & & \\ p_{31} & p_{32} & 1 & & & \\ \cdot & \cdot & \cdot & 1 & & \\ \cdot & \cdot & \cdot & \cdot & & \\ p_{n1} & p_{n2} & \cdot & \cdot & 1 & \end{bmatrix} = \mathbf{L}. \tag{7}$$

Thus, we can obtain triangular factors by multiplying $\mathbf{L}$ on both sides of Equation 6:

$$\mathbf{L} \cdot \mathbf{P}_{n-1}\mathbf{P}_{n-2} \cdots \mathbf{P}_2\mathbf{P}_1\mathbf{A} \Rightarrow \mathbf{L} \cdot \mathbf{U};$$

that is,

$$\mathbf{A} \Rightarrow \mathbf{LU}.$$

Form kji. The most popular algorithm, called Gaussian elimination, can be expressed by substituting *kji* or *kij* into Equation 5 as shown in Figure 7. Index k is considered to be the subscript of the elementary transformation matrix $\mathbf{P}_k$. Since k is outermost, the procedure of form *kji* or *kij* appears as follows:

- Outermost loop picks $\mathbf{P}_k$.
- Inner double-loop procedure operates to update the square matrix of order $n - k$ in the bottom right corner of $\mathbf{A}$,

$$\mathbf{P}_k \cdot \mathbf{A}^{(k-1)} \Rightarrow \mathbf{A}^{(k)}.$$

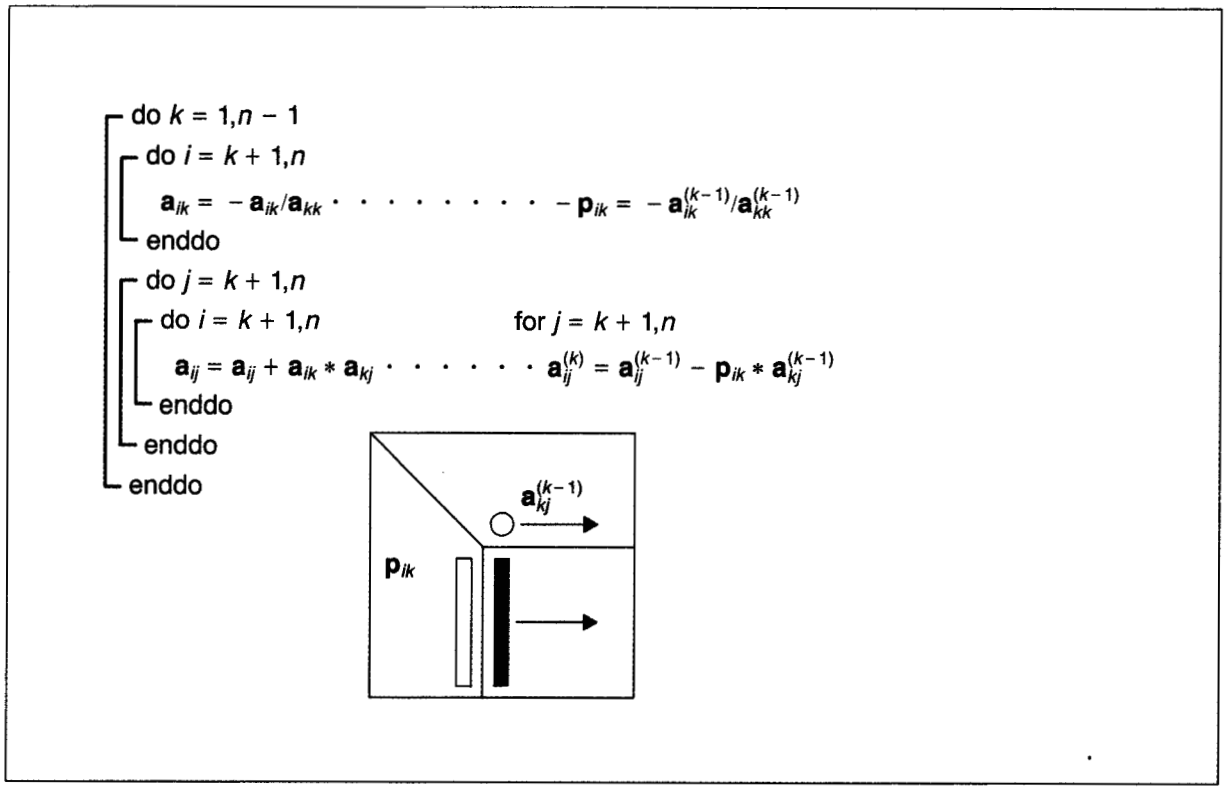
As in matrix–matrix multiplication, the three indices i, j, k represent row, column, and dummy indices, respectively. Since the dummy index is outermost, the innermost (vectorized) scalar multiple of a vector, $(\mathbf{y} = \mathbf{y} + \alpha \cdot \mathbf{x})$, requires the extra step of a vector load/store operation.

Form jki. Figure 8 shows the procedure of form *jki*, which satisfies two criteria: location of the dummy index in the middle loop, and sequential memory reference. Since the column index (j) is outermost, it becomes a vector from a matrix that each outer loop updates. The inner double-loop procedure updates the jth column of $\mathbf{A}^{(0)}$ to that of $\mathbf{A}^{(j-1)}$ using a trapezoidal matrix that consists of nonzero columns of $\mathbf{P}_1$ to $\mathbf{P}_{j-1}$, i.e., $\mathbf{a}^{(j-1)} \Leftarrow \mathbf{a}^{(0)} - \sum_k (\mathbf{p}_i)_k * \mathbf{a}_{kj}$. Though the dummy index is in the middle loop, an efficient loop construct (two vector operations and one vector-memory reference) is not generated, because the updated vector changes its beginning and length with respect to the second loop index progression. This is shown by the expression $i = k + 1, n$; i.e., the innermost loop index move is described by the next outer loop index (k). This is why triangular decomposition is more complicated than matrix–matrix multiplication. (If VF had vector instructions both to compute from the middle of the vector register and to extract an element from the vector register to store in the floating-point register, the k loop could be carried out, leaving the jth column of $\mathbf{A}$ resident in the vector register. The latter is provided but the former is not.)

Loop unrolling. There exists a technique called *loop unrolling*, which rectifies a triangular matrix to accumulate results on the vector register. Figure 9 shows three-way unrolling. This operates the three columns $(\mathbf{P}_k, \mathbf{P}_{k+1}, \mathbf{P}_{k+2})$ together with the jth column of $\mathbf{A}$ (which resides on the vector register). The number of k loops is reduced by two thirds. The deeper the loop unrolling, the faster the processing rate. Figure 10 illustrates computation density, which can be defined as the following ratio:[6]

$$\text{computation density} = \frac{\text{total operation performed}}{\text{total data items required in and out of memory}}.$$

Figure 7 Form *kji*



In the case of n-way unrolling, computational density is expressed as follows:

$$\text{computational density} = \frac{(n\text{-way}) * 2}{(n\text{-way}) + 2}.$$

Even if the n of n-way becomes large, the loop-unrolling technique has several disadvantages:

- As shown in Figure 10, computation density is still less than 2.0, which is the maximum of which the 3090 Vector Facility is capable.
- A portion of the scalar calculation that is required to rectify a triangular matrix becomes large.
- Programming becomes extensive and complicated.

Form *jki* can be considered an attractive selection if loop unrolling is carried out automatically by the compiler.

Form ijk. Because i is outermost, the outermost loop updates the row vector (ith row of $\mathbf{A}^{(0)}$), and because the dummy index is innermost, the innermost loop computes the dot product. As shown in Figure 11, form *ijk* appears as follows:

- For the elements of the lower triangular factor,

$$\mathbf{a}_{ij}^{(j-1)} \Leftarrow \mathbf{a}_{ij}^{(0)} - \sum_{k=1}^{j-1} \mathbf{p}_{ik}\, \mathbf{a}_{kj}^{(k-1)}.$$

- For the elements of the upper triangular factor,

$$\mathbf{a}_{ij}^{(j-1)} \Leftarrow \mathbf{a}_{ij}^{(0)} - \sum_{k=1}^{i-1} \mathbf{p}_{ik}\, \mathbf{a}_{kj}^{(k-1)}.$$

The advantage of form *ijk* is that, for the upper triangular elements, because the $(p_{i1}, p_{i2}, \cdots, p_{i,i-1})$ vector remains unchanged within the double-loop context, this vector can be kept in the vector register. However, for the lower triangular factor, computation must be performed by touching the triangular portion of a matrix, since the range where the innermost loop index (k) moves is described by the next outer loop index (j). Summation must be carried

Figure 8 Form *jki*

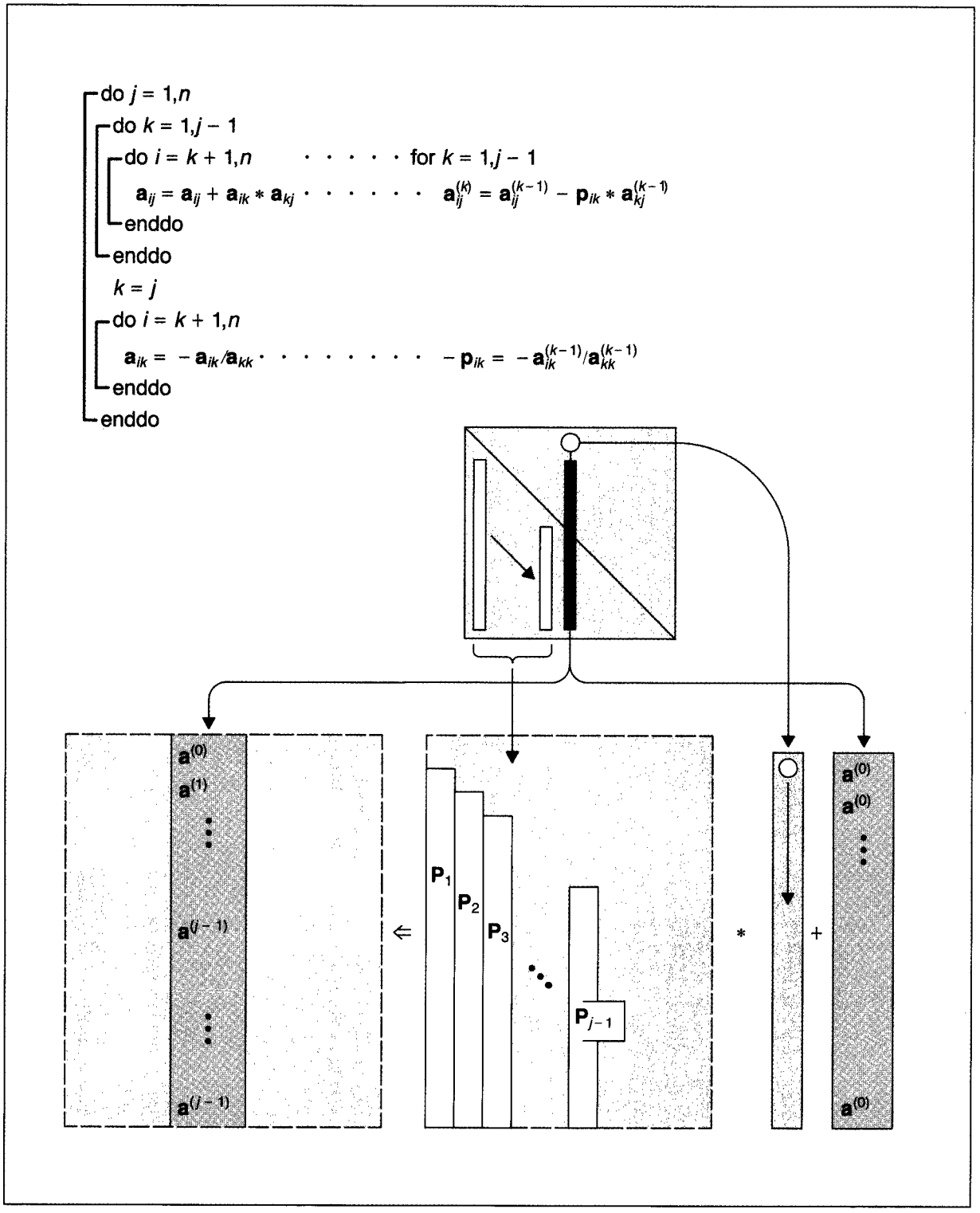


Figure 9 Loop unrolling applied to form *jki*

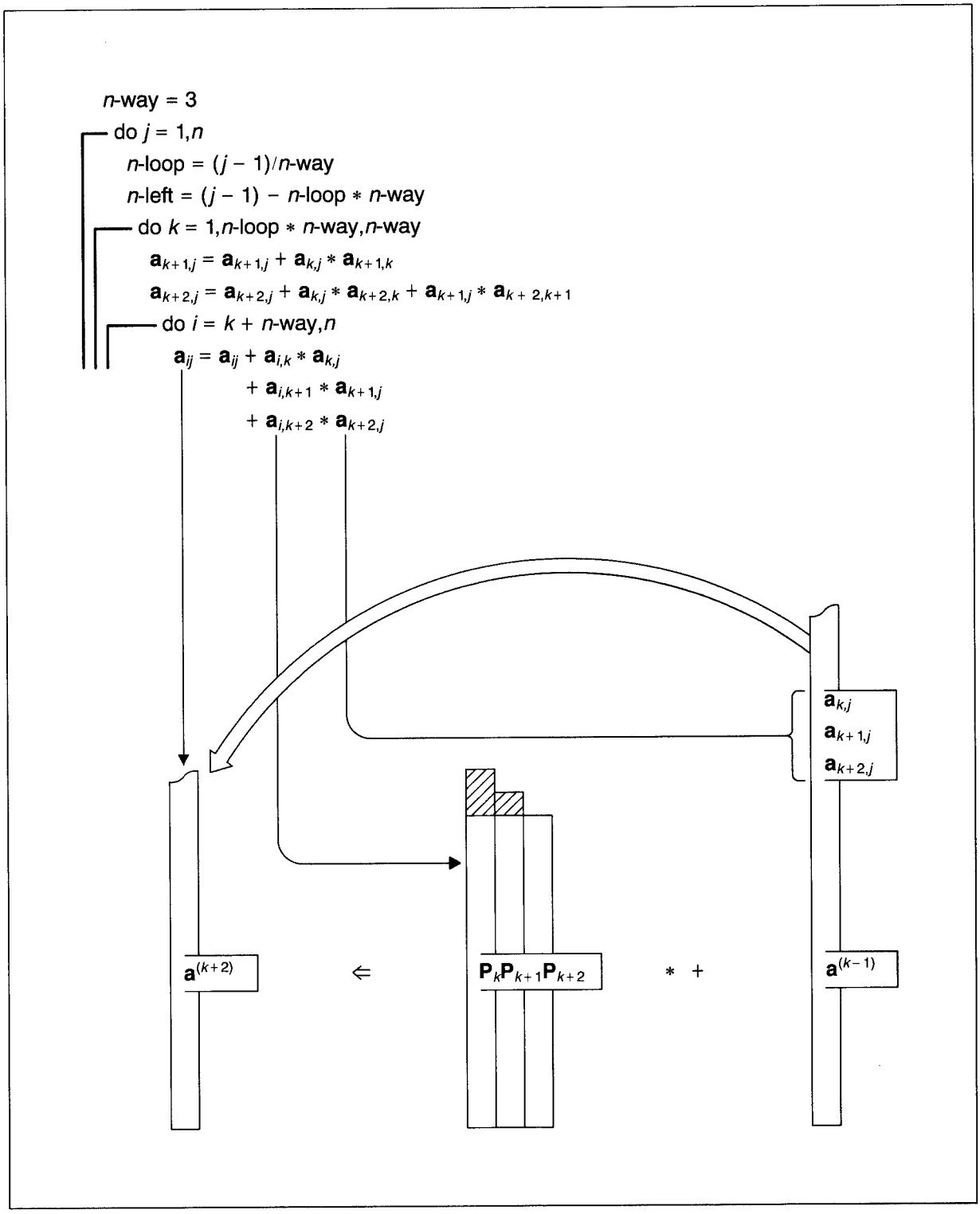
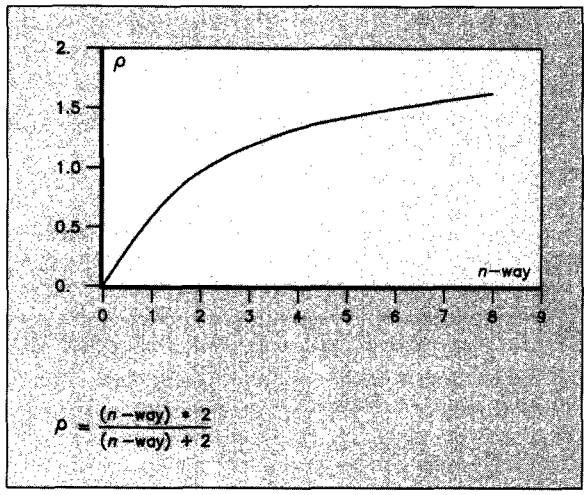


Figure 10 Computation density



out from 1 to $j - 1$, which is the index of the middle loop.

Comparison of six permutations and other selections. Figure 12 shows six permutations of the triple-loop procedure for *i, j, k*. Among these six permutations, form *jki* with loop unrolling is the fastest. Among these six, the latter part of form *jki* (where the elements of the lower triangular factor are created) and the latter part of form *ijk* (where elements of the upper triangular factor are created) satisfy the following two criteria: (1) two vector operations and one vector-memory reference, and (2) sequential memory reference. Choosing loop selections independently for the upper and lower factors to combine these two attractive portions, we can obtain a new form, form *ijk* for the upper triangular factor and form *jki* for the lower triangular factor. Figure 13 illustrates this method, in which elements of $\mathbf{u}_{ij}$ are created row-wise, then elements of $\mathbf{l}_{ij}$ are created column-wise alternatively. We call this method *form ijk/jki*; it is also known as "Crout decomposition."

Since the expressions of the ranges where innermost loop indices move are described by the next-next outer-loop indices (leaping one loop), the inner double-loop context touches the rectangular matrix.

Either the subroutine for computing consecutive dot products or the subroutine for computing the transposed matrix product with a vector can be adapted to compute $\mathbf{u}_{ij}$.

The subroutine for computing the matrix product with a vector can be adapted for computing $\mathbf{l}_{ij}$. That is, a building-block approach for program construction can be applied, which may yield the benefit that this computation-intensive part of the process can be passed to subroutines of more primitive functions.

Two popular methods from the conventional scalar computation environment, Gaussian elimination and Crout decomposition, have the property that they can be broken down into hierarchical blocks, adding algorithmic elegance and program portability to the process.

Up to this point, we have investigated the possibility of selecting *i, j, k* permutations of the triple-loop procedure for triangular decomposition. The aim of this discussion is to find the selection that gives an efficient loop construction in which the key vector could remain in the vector register—i.e., two vector operations and one vector-memory reference. In the discussion of matrix–matrix multiplication, the six permutations produced from the generic algorithm in Equation 3 are enough to search for efficient loop constructs. However, permutations in addition to those six (i.e., permutations produced from the generic algorithm in Equation 5) may also be used to realize efficient loop constructs with triangular decomposition. This is a noteworthy consequence when we have to deal with the triangular/trapezoidal portion of a matrix.

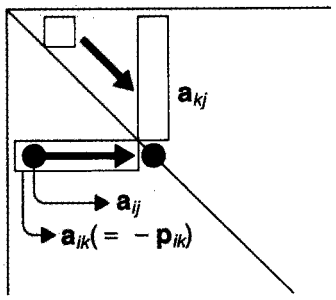
We now go to the next step to consider the capacity of the cache, so that one vector-memory reference remains in cache.

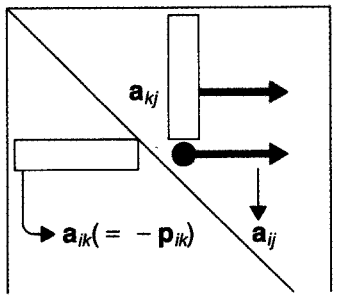
Cache consideration. In the discussion of matrix–matrix multiplication, we obtained performance improvement by dividing the original matrix into several submatrices. This is the technique by which the progress of three indices, *i, j, k*, which move from 1 to *l, n, m*, respectively, are interrupted and interlaced so as to keep the vectors from the inner double-loop procedure in the cache. Since the order of submatrices can be selected arbitrarily, we select them in a way that is suitable for the Vector Facility system, i.e., VSS or NX, as shown by Figure 4. In this section, we discuss the possibility of adapting the same approach to decomposition. Prior to this discussion, we detour to analyze the structure of triangular decomposition.

Structure of triangular decomposition matrices. We show the relations between various submatrices of

Figure 11 Form *ijk*

```
do i = 2,n
  do j = 2,i
    k = j − 1
    a_ik = −a_ik / a_kk
    do k = 1,j − 1          (for lower triangular elements)
      a_ij = a_ij + a_ik * a_kj · · · − p_ij^(j−1) = (a_ij^(0) − p_i1 a_1j^(0) − p_i2 a_2j^(1) · · · − p_i,i−1 a_i−1,j^(i−2)) / a_jj^(j−1)
    enddo
  enddo
  do j = i + 1,n
    do k = 1,i − 1          (for upper triangular elements)
      a_ij = a_ij + a_ik * a_kj · · · a_ij^(i−1) = a_ij^(0) − p_i1 a_1j^(0) − p_i2 a_2j^(1) · · · − p_i,i−1 a_i−1,j^(i−2)
    enddo
  enddo
enddo
```





(for lower triangular elements)

$$a_{ij}^{(j-1)} \Leftarrow a_{ij}^{(0)} - p_{i1}p_{i2} \cdots p_{i,j-1} * \begin{bmatrix} a_{1j}^{(0)} \\ a_{2j}^{(1)} \\ \vdots \\ a_{j-1,j}^{(j-2)} \end{bmatrix}$$

(for upper triangular elements)

$$a_{ij}^{(i-1)} \Leftarrow a_{ij}^{(0)} - p_{i1}p_{i2} \cdots p_{i,i-1} * \begin{bmatrix} a_{1j}^{(0)} \\ a_{2j}^{(1)} \\ \vdots \\ a_{i-1,j}^{(i-2)} \end{bmatrix}$$

Figure 12 Six permutations for decomposition

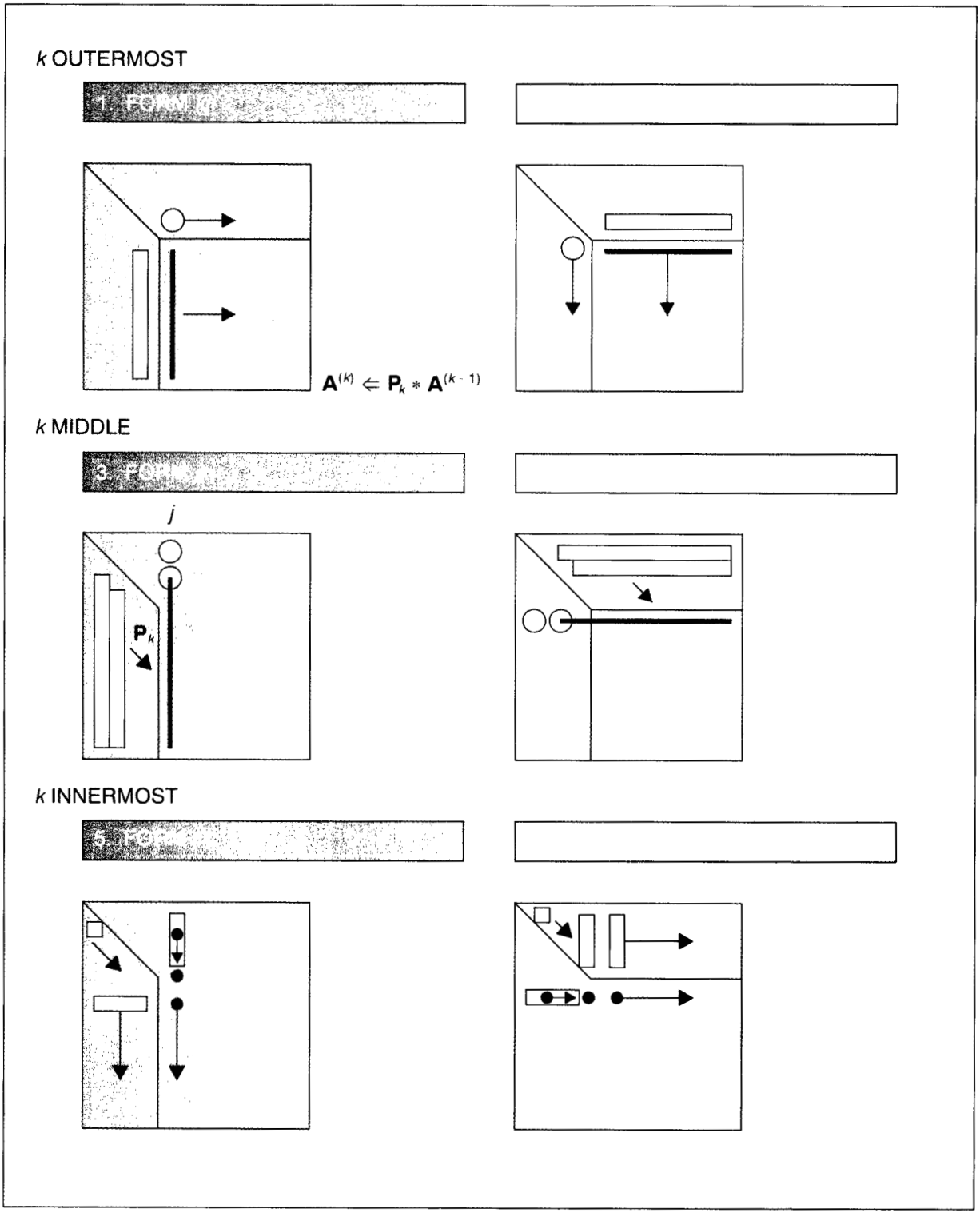
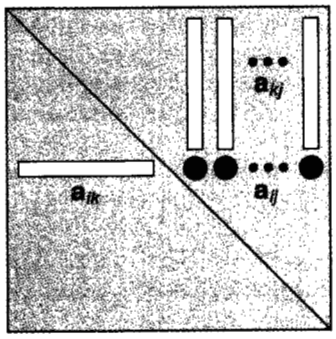
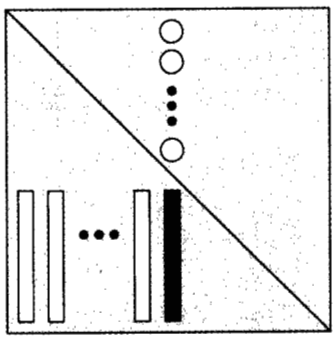


Figure 13 Form *ijk/jki*

```
do ij = 1,n
  i = ij
  do j = i,n
    do k = 1,j - 1
      a_ij = a_ij + a_ik * a_kj  . . .  u_ij = a_ij - Σ(k=1 to i-1) l_ik * u_kj
    enddo
  enddo
  j = ij
  do i = j + 1,n
    a_ij = - a_ij/a_jj
  enddo
  do k = 1,j - 1
    do i = j + 1,n
      a_ij = a_ij + a_ik * a_kj  . . .  l_ij = (a_ij - Σ(k=1 to j-1) l_ik * u_kj)/u_jj
    enddo
  enddo
enddo
```





$\mathbf{A}^{(0)}$, $\mathbf{P}$, $\mathbf{A}^{(n-1)}$ and intermediate $\mathbf{A}^{(r)}$. Considering the completion of decomposition, $\mathbf{A}^{(0)} = \mathbf{LU}$ is described by partitioned form[7] as follows:

$$\left[\begin{array}{c|c} \mathbf{A}^{(0)}_{rr} & \mathbf{A}^{(0)}_{r,n-r} \\ \hline \mathbf{A}^{(0)}_{n-r,r} & \mathbf{A}^{(0)}_{n-r,n-r} \end{array}\right] = \left[\begin{array}{c|c} \mathbf{L}_{rr} & 0 \\ \hline \mathbf{L}_{n-r,r} & \mathbf{L}_{n-r,n-r} \end{array}\right] * \left[\begin{array}{c|c} \mathbf{U}_{rr} & \mathbf{U}_{r,n-r} \\ \hline 0 & \mathbf{U}_{n-r,n-r} \end{array}\right].$$

Equating the last partitions, we have

$$\mathbf{A}^{(0)}_{n-r,n-r} = \mathbf{L}_{n-r,r}\mathbf{U}_{r,n-r} + \mathbf{L}_{n-r,n-r}\mathbf{U}_{n-r,n-r}. \quad (8)$$

From the first r major steps of Gaussian elimination (Equation 6) we obtain the equation

$$\mathbf{P}_r\mathbf{P}_{r-1} \cdots \mathbf{P}_2\mathbf{P}_1\mathbf{A}^{(0)} = \mathbf{A}^{(r)},$$

giving

$$\mathbf{A}^{(0)} = \mathbf{P}_1^{-1}\mathbf{P}_2^{-1} \cdots \mathbf{P}_{r-1}^{-1}\mathbf{P}_r^{-1}\mathbf{A}^{(r)}$$

$$= \left[\begin{array}{c|c} \mathbf{L}_{rr} & 0 \\ \hline \mathbf{L}_{n-r,r} & \mathbf{I} \end{array}\right] * \left[\begin{array}{c|c} \mathbf{U}_{rr} & \mathbf{U}_{r,n-r} \\ \hline 0 & \mathbf{W}_{n-r,n-r} \end{array}\right],$$

where $\mathbf{W}_{n-r,n-r}$ is the square matrix of order $(n - r)$ in the bottom right corner of $\mathbf{A}^{(r)}$.

From the last partitions we obtain the equation

$$\mathbf{A}^{(0)}_{n-r,n-r} = \mathbf{L}_{n-r,r}\mathbf{U}_{r,n-r} + \mathbf{W}_{n-r,n-r}. \quad (9)$$

Equations 8 and 9 give the following two important pieces of information:

- $\mathbf{W}_{n-r,n-r}$ can be obtained by subtracting from $\mathbf{A}^{(0)}_{n-r,n-r}$ a product of submatrices created after the first major r steps: $\mathbf{W}_{n-r,n-r} = \mathbf{A}^{(0)}_{n-r,n-r} - \mathbf{L}_{n-r,r}\mathbf{U}_{r,n-r}$.
- $\mathbf{L}_{n-r,n-r}\mathbf{U}_{n-r,n-r}$ is again the triangular decomposition of $\mathbf{W}_{n-r,n-r}$: $\mathbf{W}_{n-r,n-r} = \mathbf{L}_{n-r,n-r}$.

Recursive block elimination. We can exploit recursive block elimination by using these two equations. Each block step of this method comprises two phases:

- Crout elimination with termination after r major steps
- Matrix multiplication and subtraction

After the completion of the Crout elimination phase, the square matrix of order $(n - r)$ in the bottom right corner remains $\mathbf{A}^{(0)}$, but other partitions have been already completed, as follows:

$$\left[\begin{array}{c|c} \mathbf{U}_{rr} & \mathbf{U}_{r,n-r} \\ \hline \mathbf{L}_{n-r,r} & \mathbf{A}^{(0)}_{n-r,n-r} \end{array}\right].$$

Here we compute in the latter phase

$$\mathbf{W}_{n-r,n-r} = \mathbf{A}^{(0)}_{n-r,n-r} - \mathbf{L}_{n-r,r}\mathbf{U}_{r,n-r};$$

the problem shrinks from order n to $n - r$. We then move forward to the next block step to solve

$$\mathbf{W}_{n-r,n-r} \Rightarrow \mathbf{L}_{n-r,n-r}\mathbf{U}_{n-r,n-r}.$$

Computing these two phases, using Crout decomposition with termination and multiplication/subtraction recursively, we can obtain the final triangular factors. Fortunately, these two components are almost at hand; i.e., for the Crout decomposition part, form *ijk/jki* with termination can easily be created by adding arguments that specify range r; for the multiplication/subtraction part, the multiplication subroutine considering cache capacity mentioned in the section on memory hierarchy and hierarchical programming can be used. Both components, especially the latter, can run fast.

Figure 14 shows a sample of this method, which is mathematically similar to the method used in the conventional scalar computation called block elimination.

The aim of this classical method is to overcome the limited size of memory available by allocating matrix data to secondary storage (file). As a result, restrictions appeared caused by the matrix format in the file. Now, however, we can hold a large-scale matrix in memory using vs capability. Block elimination based on large memory is free from this restriction, so we can select an appropriate size for matrix blocking. In fact, we can select the matrix size independently, on the one hand, of decomposition with termination and, on the other hand, of multiplication/subtraction, in the example of coding.

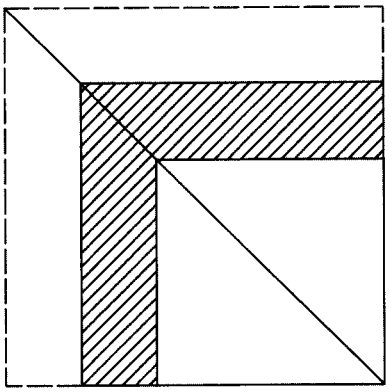
Another benefit is that the major portion, which affects performance (and is affected by hardware/system), can be computed by more primitive subroutines.

One of the most practical ways to resolve the problem of the coexistence of program performance and generality is to write a program using the building-block approach. Figure 15 shows this hierarchy. Each subroutine works with submatrices, illustrated by the solid line, and updates the shaded areas.

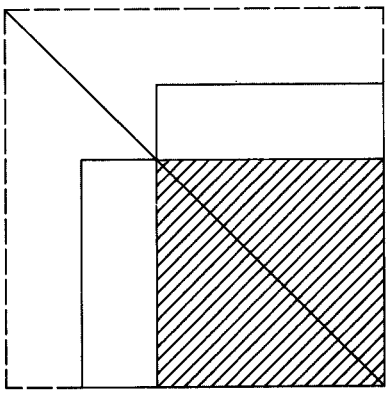
Performance comparison. For the performance estimation, we count the number of operations done by

Figure 14 Form *ijk/jki* with cache consideration

```
Subroutine DIJKIC (A,lda,n,ipvt)
double precision A(lda,n)
integer ipvt(n)
common /system/ · · ·
      .
      .
      .
 mr = r
 do m = 1,m-step
  j1 = (m - 1) * mr + 1
  j2 = min(n, j1 + mr - 1)
   call DIJKIT (A,lda,n,ipvt,j1,j2)
       (DIJKIT decomposes rows from j1 to j2)
```

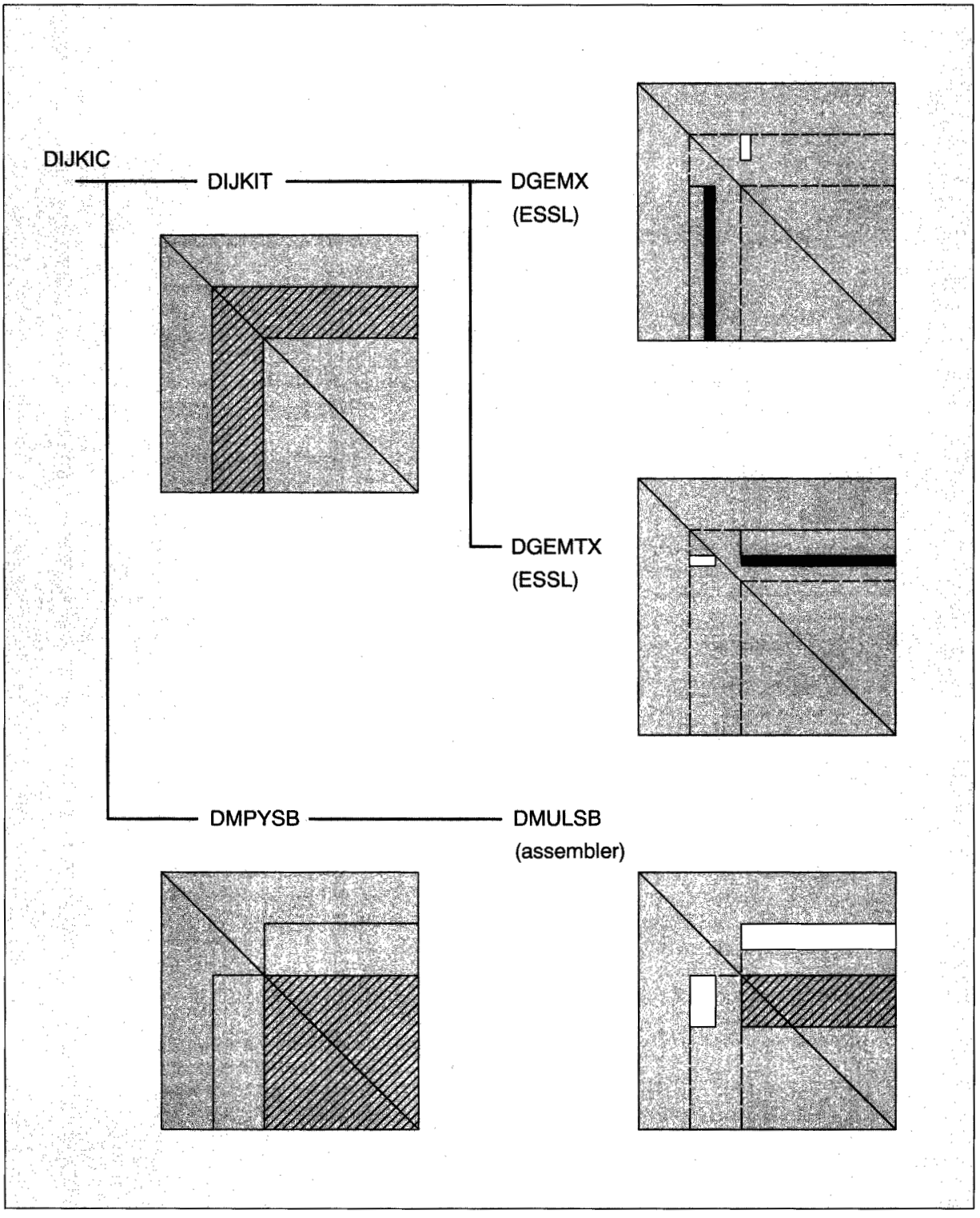


```
   call DMPYSB (A(j2 + 1, j1), lda, A(j1, j2 + 1), lda, A(j2 + 1, j2 + 1), lda,
                                                  n - j2, j2 - j1 + 1, n - j2)
       (DMPYSB computes A ⇐ A - L * U)
```



```
 enddo
 return
```

Figure 15 Building block construction for form *ijk/jki* with cache consideration



the multiplication/subtraction routine. Assuming n=1000 and choosing a ninefold reduction with r = 100, we find that 85 percent of the operations can be carried out by the matrix–matrix multiplication routine. This percentage is expressed by the ratio

$$2 \cdot \sum_{k=1}^{9} (n - k \cdot r)^2 \cdot r/(2/3n^3).$$

The higher this percentage becomes, the faster the processing rate that can be expected.

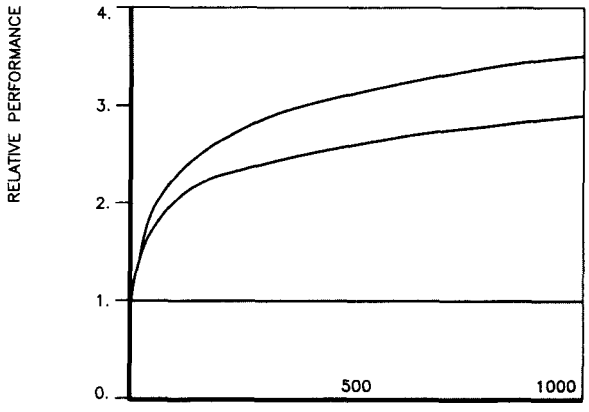
Figure 16 shows the relative performance comparison of the three methods—form *jki*, form *ijk/jki*, and the method using cache considerations. The order of matrix is less than 1000. The difference between form *jki* and form *ijk/jki* is obtained from the loop procedure ("two vector operations and three vector-memory references" and "two vector operations and one vector-memory reference"). An additional performance improvement of about 20 percent obtained from cache consideration can be evaluated between form *ijk/jki* and the method with cache consideration. The highest performance reaches about 70 MFLOPS on the 3090 Model E, when $n = lda = 1000$ and $r = 40$.

Concluding remarks

In order to obtain high performance from the IBM 3090 Vector Facility, we must modify the way in which we construct our numerical method programs. We have investigated vector instruction constructs in terms of the loop performance of a matrix–matrix multiplication algorithm in which relations between loop method and performance are classified in three levels. This classification corresponds to the arguments discussed in References 8–10. We have given examples of triangular decomposition, in which we applied the tuning technique at the start with loop analysis and searched the method to obtain additional performance improvement. Fortunately, we have obtained this improvement with minimal modifications to the program in this example. If we had had to extract high performance from both vector and parallel processing, we would have required different kinds of formulations, such as those given in Reference 10.

Since the methods of augmenting engineering and scientific computing power are trending toward the use of more complicated mechanisms, we think it better to establish some interface between machine-dependent routines and independent routines. A building-block approach based on this interface will give us not only high performance but also flexibility to deal with possible changes in hardware/systems. The interface suggested by BLAS would be a candidate for this kind of interface.

Figure 16 Relative performance comparison



Since this study was initiated during the benchmarking phase of a project to create a fast matrix inversion routine, we built our program based on Release 1 of that program, the Engineering and Scientific Subroutine Library (ESSL) interface. We now have Release 2 of ESSL, which provides a wider menu of functions. The experience we gained through this study tells us that there exists something sensitive to leading dimension A (LDA) of the given matrix A. It suggests that, even for the routines of primitive functions, elaborate tuning is necessary. For this reason, we recommend the ESSL interface for rebuilding our numerical method routines.

Cited references

1. D. H. Gibson, D. W. Rain, and H. F. Walsh, "Engineering and scientific processing on the IBM 3090," *IBM Systems Journal* **25,** No. 1, 36–50 (1986).
2. *IBM System/370 Vector Operations*, SA22-7125, IBM Corporation (1986); available through IBM branch offices.
3. W. Buchholz, "The IBM System/370 vector architecture," *IBM Systems Journal* **25,** No. 1, 51–62 (1986).
4. J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine," *SIAM Review* **26,** No. 1, 91–112 (1984).
5. R. C. Agarwal and J. W. Cooley, "Fourier transform and convolution subroutines for the IBM 3090 Vector Facility," *IBM Journal of Research and Development* **30,** No. 2, 145–162 (1986).
6. H. H. Wang, *Introduction to Vectorizing Techniques on the IBM 3090 Vector Facility*, G370-3489, IBM Corporation; available through IBM branch offices.

7. J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford (1965).
8. C. L. Lawson, R. L. Hanson, D. R. Kincaid, and F. T. Krough, "Basic linear algebra subprograms for FORTRAN usage," *ACM Transactions on Mathematical Software* **5**, No. 3, 308–323 (1979).
9. J. J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, "A proposal for an extended set of FORTRAN basic linear algebra subprograms," Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 41 (1984).
10. J. J. Dongarra, J. D. Croz, I. Duff, and S. Hammarling, "A proposal for a set of level 3 basic linear algebra subprograms," Argonne National Laboratory Mathematics and Computer Science Division, Technical Memorandum No. 88 (1987).

Hikaru Samukawa *IBM Japan Ltd., Shinjuku Sumitomo Building, 6-1, Nishi-Shinjuku 2-chome, Shinjuku-ku, Tokyo 163, Japan.*
Mr. Samukawa is an advisory industry specialist in the NIC Marketing Center, where he is working on marketing activities for the IBM 3090 Vector Facility. He joined IBM Japan in 1984 and has been working on the Vector Facility since 1985. Mr. Samukawa worked as a programmer/designer of numerical analysis software, especially structural analysis such as NASTRAN, at UNIVAC Japan from 1972 to 1984. He received his B.S. in mechanical engineering from Waseda University in 1972. Mr. Samukawa is a member of the Japanese Information Processing Society.

Reprint Order No. G321-5338.