Engineering and Scientific Subroutine Library for the IBM 3090 Vector Facility

by J. McComb
S. Schmidt

The Engineering and Scientific Subroutine Library (ESSL) provides FORTRAN, Assembler, and APL2 application programmers with a high-performance set of mathematical subroutines which take advantage of the performance gains offered by the IBM 3090 Vector Facility. This paper describes the contents of ESSL and presents some of the techniques that were used to develop high-performance vector subroutines. Other key design considerations such as accuracy, ease of use, and error handling are also discussed. This information should be useful to anyone developing programs for the IBM 3090 Vector Facility.

The Engineering and Scientific Subroutine Library (ESSL)[1,2] is an IBM Program Product that can be used with VS FORTRAN, Assembler, and APL2, running under the MVS/XA, VM/SP HPO, VM/XA SF, or VM/XA SP operating systems.

ESSL comprises both a vector library and a scalar library. The vector library subroutines have been highly tuned to take advantage of the performance gains offered by the IBM 3090 Vector Facility. The scalar library is provided for development and testing on scalar machines of application programs containing ESSL calls.

The ESSL subroutines can be divided into ten areas:

- Linear algebra subprograms
- Matrix operations
- Linear algebraic equations
- Eigensystems analysis
- Signal processing
- Sorting and searching
- Interpolation
- Numerical quadrature
- Random number generation
- Utilities

The vector and scalar libraries each contain 233 user-callable subroutines (Table 1) which are useful for many different types of scientific and engineering applications in such industries as aerospace, automotive, electronics, finance, petroleum, research, and utilities. Several versions of most subroutines are provided. These may include a short- and long-precision real version, a short- and long-precision complex version, and an integer version.

This paper first describes the IBM 3090 Vector Facility and the contents of ESSL. Next, some of the techniques that were used to optimize performance are discussed; these are illustrated by examining the performance of a few subroutines. Finally, accuracy, ease of use, and error handling are discussed.

Table 1 ESSL computational areas

Computational Areas	I*4	R*4	R*8	C*8	C*16
Linear algebra subprograms					
Vector-Scalar	0	18	18	17	17
Sparse Vector-Scalar	0	5	5	0	0
Matrix-Vector	0	7	7	3	3
Sparse Matrix-Vector	0	0	3	0	0
Matrix operations	0	5	5	5	5
Linear algebraic equations					
Dense	0	7	7	0	0
Banded	0	10	10	0	0
Sparse	0	0	2	0	0
Linear Least Squares	0	3	3	0	0
Eigensystem analysis	0	2	2	2	2
Signal processing					
Fourier Transforms	0	7	1	0	0
Convolution/Correlation	0	10	0	0	0
General	0	5	5	0	0
Sorting and searching	4	4	4	0	0
Interpolation	0	3	3	0	0
Numerical quadrature	0	5	5	0	0
Random number generators	0	1	1	0	0
Utilities	1	0	1	0	0
Total	5	92	82	27	27

I*4 = Integer subroutines
R*4 = Short-precision real subroutines
R*8 = Long-precision real subroutines
C*8 = Short-precision complex subroutines
C*16 = Long-precision complex subroutines

IBM 3090 Vector Facility overview

In February 1986, the IBM 3090 Vector Facility was made available. The essential computational advantages of the Vector Facility are realized by processing data in groups called vectors. IBM 3090 processing as well as the features of the IBM 3090 Vector Facility are described in a previous set of *IBM Systems Journal* articles.[3–7]

The IBM 3090 Vector Facility has 16 vector registers, each containing 128 (32-bit-wide) short-precision elements. Eight vector-register pairs can also be coupled into a 64-bit format to handle long-precision data. The length of a vector register is referred to as the vector section size (VSS). When vectors are longer than the VSS, the architecture provides instructions to handle vector sectioning.[3] There are a total of 171 vector instructions to process, move, and interrogate data. Neglecting overhead, most vector instructions (e.g., add, subtract, multiply) generate one floating-point result every cycle. Since it is possible to configure the multiply and add pipelines as one long pipeline, neglecting overhead, the compound vector instructions (multiply–add, multiply–subtract, and multiply–accumulate) generate two floating-point results every cycle. The architecture of the IBM System/370 Vector Facility[8] is compatible with the System/370 architecture.

In summary, therefore, working in conjunction with a FORTRAN vector compiler[9,10] and high-performance vector software such as ESSL, the IBM 3090 Vector Facility offers a significant potential for increased computational performance.

ESSL contents

Linear algebra subprograms. A collection of 103 linear algebra subprograms cover four computational areas: vector–scalar, sparse vector–scalar, matrix–vector, and sparse matrix–vector.

The vector–scalar linear algebra subprograms contain a subset of the Basic Linear Algebra Subprograms (BLAS)[11] and other commonly used vector computations, such as dot product, vector maximum element, vector copy, and vector update. The BLAS[11]

constitute an attempt to establish a standard set of computational subroutines that allow application software to be "portable" (i.e., usable in more than one machine environment) and at the same time efficient.

The sparse vector–scalar linear algebra subprograms operate on sparse vectors; that is, only the nonzero elements of the vector are stored. They provide functions similar to the vector–scalar linear algebra subprograms, and represent a subset of the proposed Sparse BLAS.[12]

The matrix–vector linear algebra subprograms contain a subset of the Level 2 BLAS[13] and include matrix–vector products, rank-one updates, and rank-two updates for real general, complex general, and real symmetric matrices.

The sparse matrix–vector linear algebra subprograms operate on sparse matrices; i.e., only the nonzero elements of the matrix are stored. Matrix–vector products for sparse matrices or their transposes are provided.

Matrix operations

There are 20 matrix operations subroutines, which perform matrix multiplication for real and complex matrices, their transposes, or their conjugate transposes, and matrix addition and subtraction for real and complex matrices or their transposes. In addition to the standard matrix multiplication, there is a matrix multiplication that uses Winograd's variation of Strassen's algorithm which provides improved performance for large matrices. A combination of matrix multiplication and addition, a proposed Level 3 BLAS,[14] is also included.

Linear algebraic equations. The 42 linear algebraic equations subroutines cover four areas: dense, banded, sparse, and linear least-squares.

The dense linear algebraic equations subroutines provide solutions to linear systems of equations for real general matrices or their transposes, and real positive definite symmetric matrices. The functions of factorization and solve with a condition number and determinant are provided. Matrix inversion with a condition number and determinant is also provided for real general matrices.

The banded linear algebraic equations subroutines provide solutions to linear systems of equations for real general band matrices, real positive definite symmetric band matrices, real general tridiagonal matrices, and real positive definite symmetric tridiagonal matrices. The functions of factorization and solve are provided for all matrix types.

The sparse linear algebraic equations subroutines provide an iterative solution to linear systems of equations for sparse positive definite and negative definite symmetric matrices using the conjugate gradient method, with or without preconditioning.

Three levels of vector performance can be achieved on the IBM 3090 Vector Facility.

The linear least-squares subroutines provide least-squares solutions to linear systems of equations for real general matrices. Two methods are provided: singular value decomposition, and a QR decomposition[2] with column pivoting.

Eigensystem analysis. Eight eigensystem analysis subroutines are provided to compute the eigenvalues and optionally the eigenvectors of real symmetric, complex Hermitian, real general, and complex general matrices.

Signal processing. The 28 signal processing subroutines cover three areas: Fourier transforms, convolution and correlation, and general signal processing.

The Fourier transform subroutines perform transforms in both one and two dimensions. The short-precision subroutines contain a high-performance mixed-radix capability. The convolution and correlation subroutines provide the choice of using Fourier, direct, or combined Fourier and direct methods. The autocorrelation subroutines provide the choice of using Fourier or combined Fourier and direct methods.

The general signal processing subroutines provide the same function as a subset of key IBM 3838 array

processor algorithms: polynomial evaluation, Ith zero crossing, time-varying recursive filter, quadratic interpolation, and Weiner–Levinson filter coefficients.

Sorting and searching. Twelve sorting and searching subroutines are provided for sorting in place with or without index designations, binary searching, and sequential searching of real and integer data.

Interpolation. Six interpolation subroutines provide the capability for polynomial interpolation, local polynomial interpolation, and cubic spline interpolation.

Numerical quadrature. Ten numerical quadrature subroutines provide methods for integrating a tabulated function and a user-supplied function over a finite, semi-infinite, or infinite region of integration by Gaussian quadrature methods.

Random number generator. There are two uniform-distribution pseudorandom number generators.

Utility subroutines. Two utility subroutines are provided for error handling and data format conversions.

Performance

The ESSL vector subroutines have been designed to provide high performance on the IBM 3090 Vector Facility. To achieve this performance, the vector subroutines use algorithms tailored to specific operational characteristics of the IBM 3090 Vector Facility, such as cache size (the cache is a high-speed buffer that is used to hold portions of main memory that have been most recently referenced), vector section size (VSS), number of vector registers, and page size. The key computational modules have been written in assembler language, the remaining modules in FORTRAN.

The following techniques were used to optimize performance:

- Access data that are stored contiguously; that is, use stride-1 computations.[15]
- Reuse data in vector registers, minimizing vector loads and stores.
- Manage the cache efficiently to maximize data reuse; i.e., algorithms are structured to operate on subblocks that are sized to remain in the cache until all computations involving the subblock are complete. For example, the number of rows in the subblock might be equal to the VSS, while the number of columns is chosen so that the subblock will fit in the cache.
- Use the most efficient machine instructions—for example, the multiply–add, multiply–subtract, and multiply–accumulate instructions (the compound vector instructions). Neglecting overhead, these instructions generate two floating-point results every cycle. Other vector instructions, such as multiply, add, and subtract, generate one floating-point result per cycle.
- Perform fewer loads and stores for short-precision data by using long-precision instructions.
- Use algorithms that minimize paging; for example, alternate forward and backward sweeps through the columns of a matrix.

Table 2 Three levels of vector performance (300 vector elements)

Subroutine	T(S)	T(E)	T(S)/T(E)
DAXPY	0.000089	0.000035	2.5
DGEMX	0.019496	0.003849	5.1
DGEMUL	5.935	0.768	7.7

T(S) = IBM 3090 CPU time in seconds for public domain scalar subroutine
T(E) = IBM 3090-VF CPU time in seconds for vector ESSL subroutine

While developing ESSL, it was observed that three levels of vector performance can be achieved on the IBM 3090 Vector Facility. These levels can be roughly characterized by the following Scalar CPU Time/Vector CPU Time speedup ratios: 3.0 or less, 3.0–6.0, and greater than 6.0.

These ratios generally correspond to the programmer being able to take advantage of vector instructions, reuse of data in vector registers, and reuse of data in cache.

A discussion of the performance of selected ESSL subroutines (see Table 2) illustrates these performance levels.

DAXPY. The DAXPY subroutine computes a vector update,

$$y \leftarrow y + \alpha x,$$

where x and y are long-precision vectors and α is a long-precision scalar.

DAXPY is representative of the subroutines that achieve Level 1 performance. Little can be done to

Table 3 Illustration of cache effect on performance

Number of Vector Elements	Empty Cache [TIME(E)]	Primed Cache [TIME(E)]	Empty/ Primed Ratio
DAXPY			
50	9.60	7.00	1.37
500	52.20	38.20	1.37
1000	105.40	73.80	1.43
DZAXPY			
50	11.86	7.06	1.68
500	67.20	38.40	1.75
1000	130.76	74.22	1.76

TIME(E) = IBM 3090-VF CPU time in microseconds for ESSL subroutine

optimize these subroutines; there is little data reuse, vector loads and stores are executed inside the innermost loop, and little benefit is derived from the vector compound instructions.

It is interesting to examine how the performance of DAXPY can vary. Table 3 illustrates the performance of DAXPY for a "primed cache" (all data required for the computation were in the cache before the subroutine was executed), and an "empty cache" (none of the required data were in the cache before the subroutine was executed). Even for this simple computation, we see that the performance penalty ranges from 37–43 percent when the data are not in the cache, and must be fetched from memory. (Unless stated otherwise, all other performance data are for an empty cache.)

Since the vector, y, is used for both input and output, DAXPY involves some data reuse. Consider the ESSL subroutine, DZAXPY, which is a minor modification of DAXPY that allows distinct vectors for the input and output, i.e.,

$z \leftarrow y + \alpha x.$

Table 3 gives the performance of DZAXPY for a primed cache and an empty cache. For DZAXPY, the performance penalty ranges from 68–76 percent when the data are not in the cache and must be fetched from memory.

DAXPY and DZAXPY perform the same number of floating-point operations but differ in their memory accesses. When all data are in the cache prior to the computation, they both require the same amount of time to perform their computation. However, this is not the case when the data must be fetched from main memory.

DGEMX. The DGEMX subroutine computes a matrix–vector product,

$y \leftarrow y + \mathbf{A}x,$

where x and y are long-precision vectors and $\mathbf{A}$ is a long-precision matrix.

DGEMX is representative of the subroutines that achieve Level 2 performance. In addition to the performance improvement gained from using vector instructions, the matrix–vector product is implemented so that vector loads and stores are removed from the inner loop by reusing data in vector registers.

Vector register reuse. To illustrate the reuse of data in vector registers, consider the task of computing $y \leftarrow \mathbf{A}x$; for simplicity, assume that the number of rows in the matrix $\mathbf{A}$ is less than the vector section size (VSS):

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ \cdot & \cdots & \cdot \\ a_{m1} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ \cdot \\ \cdot \\ \cdot \\ x_n \end{bmatrix}.$$

This computation can be done as a series of dot products,

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \qquad i = 1, \quad m.$$

However, using the dot-product approach involves accessing matrix $\mathbf{A}$ by rows. Since FORTRAN stores arrays in column major order, this would be a non-unit stride access. Fortunately, it is easy to reformulate this as a stride-1 computation:

$$\begin{bmatrix} y_1 \\ \cdot \\ \cdot \\ \cdot \\ y_m \end{bmatrix} = x_1 \begin{bmatrix} a_{11} \\ \cdot \\ \cdot \\ \cdot \\ a_{m1} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ \cdot \\ \cdot \\ \cdot \\ a_{mn} \end{bmatrix}.$$

This corresponds to the following FORTRAN loop:

```
   DO 10 I = 1,M
     Y(I) = 0.0
10 CONTINUE

   DO 30 J = 1,N
     DO 20 I = 1,M
       Y(I) = Y(I) + X(J)*A(I,J)
20   CONTINUE
30 CONTINUE
```

This can be vectorized using either DAXPY or DGEMX:

Using DAXPY

```
      DO 10 I = 1,M
        Y(I) = 0.0
   10 CONTINUE

      DO 30 J = 1,N
*     DAXPY segment
        LD      F0,X(J)
        VLD     V0, Y
        VMADS V0,F0,A(1,J)
        VSTD    V0,Y
   30 CONTINUE
```

Using DGEMX

```
      DO 10 I = 1,M
        Y(I) = 0.0
   10 CONTINUE

*     DGEMX segment
        LD      F0,X(1)
        VMDS    V0,F0,A(1,1)
        DO 30   J = 2,N
          LD    F0,X(J)
          VMADS V0,F0,A(1,J)
   30 CONTINUE
        VAD     V0,V0,Y
        VSTD    V0,Y
```

DAXPY must be called N times from within the DO 30 loop, each time adding the contribution from the jth column of **A** to the vector, y, stored in memory. DGEMX is called only once and computes the matrix–vector product by accumulating the contribution from each column of **A** in a vector register. Contrasting the two implementations, we notice that in the inner loop for DGEMX we have been able to eliminate the vector load and vector store instructions by reusing the data in vector register 0.

It is easy to extend the matrix–vector product to handle matrices of arbitrary size by breaking the matrix into submatrices.

Paging considerations. It is easy to extend the matrix–vector product to handle matrices of arbitrary size by breaking the matrix into submatrices. However, for large matrices, the impact of paging must be considered. It is well known (see the paper by Dubrulle[16]) that paging for matrices stored in column major order will be minimized if the computation is structured so that all elements in one column are used before proceeding to the next column. However, this conflicts with our desire to reuse data in vector registers. A compromise is to block the matrix (the size of the blocks is k*vss by n, where k depends on the number of available vector registers), and then to alternate forward and backward sweeps through the columns of **A**. This increases the likelihood that pages will be in memory when accessed, since a least-recently-used paging replacement algorithm is used. The following is an example of matrix blocking:

$$y = \begin{bmatrix} \vec{A}_1 \\ \leftarrow \\ A_2 \\ \rightarrow \\ A_3 \end{bmatrix} \begin{bmatrix} x_1 \\ . \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} A_1x \\ — \\ A_2x \\ — \\ A_3x \end{bmatrix};$$

Compute A_1x $\quad j = 1, n,$
Compute A_2x $\quad j = n, 1, -1,$
Compute A_3x $\quad j = 1; n.$

DGEMUL. The DGEMUL subroutine performs matrix multiplication, **C** ← **AB**, where **A**, **B**, and **C** are long-precision general matrices, and either the normal or transposed form of **A** and **B** can be selected.

DGEMUL is representative of the subroutines that achieve Level 3 performance. In addition to the performance improvement gained from using vector instructions, and reusing data in vector registers, DGEMUL is also able to reuse data in cache. DGEMUL is structured to operate on subblocks that are sized to remain in the cache until all computations involving the subblock are complete.

For example, consider multiplying two square matrices, **A** and **B**, of order n. Depending on the size of n, DGEMUL might block the **A** matrix as shown in Figure 1.

The value of k is determined such that the vss by k subblock of **A** will remain in the cache while it is used to compute n matrix–vector products with the k-element vectors of **B**.

Cache considerations. As discussed by Tucker,[7] the cache is a high-speed buffer that is used to hold portions of main memory that have been most recently referenced. For the IBM 3090 Vector Facility, the cache is a 64K-byte (8K-double-word) buffer divided into four sets of 2K double words, with data transfers carried out a line (16 double words) at a time. A double word can reside in only one particular location of any one of the four sets; the location is uniquely determined by the low-order bits of the virtual address of the double word. Accessing the elements of a vector with a particular stride can result in the selected double words frequently mapping to the same cache location; this has the effect of reducing the effective size of the cache. For ex-

Figure 1 Blocking of A matrix by the DGEMUL subroutine

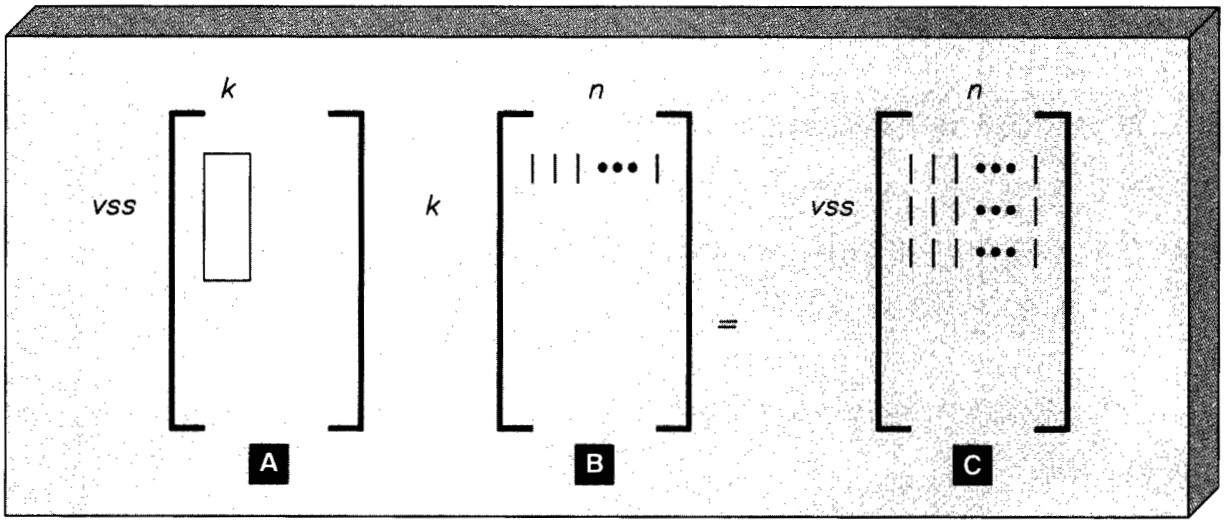

Table 4 IBM 3090 cache

Stride	Effective Cache Size (double words)
1	8192
16	512
32	256
64	128
128	64
256	32
512	16
1024	8
2048	4

Assumptions:
1. Long-precision array loaded on a page boundary.
2. Sequential pages in virtual memory are assigned sequential pages in real memory. (This may not always be the case since page assignments are under the control of the operating system.)

ample, suppose vector elements with a stride of 2048 are accessed (subject to the assumptions given in Table 4). All elements will map to the same cache location, so the first four elements will occupy that location in the four sets. The fifth element fetched replaces one of the first four elements, so for this computation the effective size of the cache has been reduced to four double words. The application programmer can help to alleviate this situation by not specifying the leading dimension (lda) of an array equal to or near a multiple of 128 for long-precision arrays, or 256 for short-precision arrays (see Table 4).

One important characteristic of the IBM 3090 Vector Facility is that if data are in the cache, there is no performance penalty associated with stride-*N* computations. This fact was used by a number of ESSL subroutines, among them the Fourier Transform subroutines,[17,18] the matrix multiplication subroutines, and the linear algebraic equation subroutines, to achieve high performance.

Stride-*N* considerations. It is well known that computations with non-unit stride should be avoided. However, this is not always possible. Consider computing the sum of two transposed matrices $\mathbf{C} \leftarrow \mathbf{A}^T + \mathbf{B}^T$, where **A**, **B**, and **C** are stored in normal form in two-dimensional FORTRAN arrays.

Intuitively for the computation $\mathbf{C} = \mathbf{A}^T + \mathbf{B}^T$, one might guess that the best strategy would be the following:

```
      DO 20 J = 1,N
        DO 10 I = 1,M
          C(J,I) = A(I,J) + B(I,J).
10    CONTINUE
20 CONTINUE
```

That is, fetch columns of matrices **A** and **B** (stride 1), and store rows of the matrix **C** (stride *N*).

However, this is not the case for the IBM 3090 Vector Facility. Due to the penalty for storing with large stride, better performance is obtained by fetching rows of matrices **A** and **B** (stride *N*), and storing columns of matrix **C** (stride 1):

```
      DO 20 J = 1,N
        DO 10 I = 1,M
          C(I,J) = A(J,I) + B(J,I).
10    CONTINUE
20 CONTINUE
```

In cases where stride-N computations cannot be avoided, better performance will be achieved if the program can be constructed to use loads with strides instead of stores with strides. In fact, in some cases, large stride-N computations are more effectively done with scalar code.

Other programming techniques

Next, some other programming techniques that were used in ESSL are discussed.

Although stride-N computations should be avoided if possible, there are some cases where it may be worthwhile to consider using such a computation (see the paper by Dongarra et al.[19]). As discussed earlier, the matrix–vector product can be calculated by accessing the m by n matrix **A** by columns (stride 1) or by rows (stride N). Generally, one would opt for accessing the matrix by columns, but if m is small and n is somewhat larger than m, the matrix–vector product is more efficiently computed using the stride N implementation. For this case, m dot products of length n are computed, i.e., fewer operations with longer vector lengths than if the stride-1 approach were used.

In some cases, loop overhead can be decreased by treating matrices as though they were vectors. For example, suppose one wishes to compute $\mathbf{C} = \mathbf{A} + \mathbf{B}$, where **A**, **B**, and **C** are $n \times n$ matrices stored compactly [i.e., the arrays have been dimensioned **A**(N, N), **B**(N, N), **C**(N, N)]. In this case, it is possible to cut down on loop overhead by doing one vector addition of length n^2 instead of n vector additions of length n.

Next, a technique to save load-and-store machine cycles is illustrated. Consider the problem of computing the element-by-element product of two complex short-precision vectors (an important computational element required by the convolution subroutines). Long-precision vector load-and-store instructions can be used in place of short-precision vector load-and-store instructions (provided arrays are aligned on double-word boundaries), resulting in the following loop:

```
      LA      R2,2              SET THE STRIDE REGISTER TO 2
LOOP VLVCU N                    SET THE VECTOR COUNT TO N
      VLD     V2,X              LOAD X WITH STRIDE 1
*                               V2 CONTAINS XR (REAL PART OF X)
*                               V3 CONTAINS XI (IMAGINARY PART OF X)
      VME     V0,V2, YR(R2) XR*YR
      VMSE    V0,V3,YI(R2)  ZR = XR*YR - XI*YI
      VME     V4,V2,YI(R2)  XR*YI
      VMAE    V4,V3,YR(R2)  ZI = XR*YI + XI*YR
      VLER    V1,V4         COPY ZI TO V1
      VSTD    V0,Z          STORE Z WITH STRIDE 1
      BP      LOOP

      COMPUTE Z(I) = X(I) * Y(I)
      WHERE   X, Y, AND Z ARE SHORT-PRECISION COMPLEX VECTORS
              X = (XR,XI)
              Y = (YR,YI)
              Z = (ZR,ZI)
```

The above loop contains eight vector instructions; a comparable loop that did not use the long-precision vector load-and-store instructions would contain nine vector instructions. In addition to saving one vector instruction, some of the stride-2 accesses normally required for complex data have been changed into stride-1 accesses.

Representative performance information for selected ESSL subroutines for one particular problem size is given in Table 5. ESSL performance varies depending on the application-program-dependent matrix and vector sizes. Additional performance information for ESSL subroutines is available.[18,20–22]

Accuracy

Accuracy is strongly dependent on the algorithm used (which may vary within a subroutine), the matrix and vector sizes, and the IBM 3090 Vector Facility model-dependent parameters [Vector Section Size (VSS) and Partial Sum Number (PSN)]. For the IBM 3090 Vector Facility, the VSS (number of elements in a vector register) is 128, and the PSN (length of the pipeline) is 4. The IBM 3090 Vector Facility uses a partial-sum technique (described in Reference 3) for the multiply and accumulate, accumulate, zero partial sums, and sum partial sums vector instructions. Since floating-point addition is not associative, this technique produces a result which is model-dependent and may differ from the result of sequential addition.

ESSL provides short- and long-precision versions of most subroutines. In most cases, short-precision subroutines use long-precision accumulations, with the final result truncated to short-precision to obtain increased accuracy.

Bitwise-identical results frequently occur, but are not guaranteed between the ESSL vector and scalar subroutines due to architectural differences between the

Table 5 Sample ESSL performance results

	T(S)	T(E)	T(S)/T(E)
$N = 300$			
General matrix factor	2.4083	0.3462	7.0
General matrix solve	0.0251	0.0056	4.5
Positive definite symmetric tridiagonal matrix factor	0.00034	0.00026	1.3
Positive definite symmetric tridiagonal matrix solve	0.00037	0.00015	2.5
Symmetric matrix eigenvalues and eigenvectors	35.92365	7.96425	4.5
$N = 4884$			
Sparse positive definite symmetric matrix iterative solver	20.454	9.912	2.1
$N = 2048$			
Complex Fourier transform	0.01557	0.00313	5.0
Complex to real Fourier transform	0.01001	0.00197	5.1
$N = 10000$			
Sort	0.07079	0.02756	2.6
Random number generator	0.00487	0.00213	2.3

N = Order of the matrix, size of the transform, or number of vector elements
T(S) = IBM 3090 CPU time in seconds for public domain scalar subroutine
T(E) = IBM 3090-VF CPU time in seconds for vector ESSL subroutine

vector and scalar hardware. For example, vector divide and multiply instructions do not permit unnormalized operands; scalar instructions do.

ESSL does not round numbers or mask underflow. However, for performance reasons we recommend that underflow be masked using the VS FORTRAN Library utility, XUFLOW.[9,10]

Ease of use

ESSL is callable from FORTRAN, Assembler, and APL2 programs. All ESSL subroutines follow standard FORTRAN calling conventions, and must run in the FORTRAN environment. When ESSL subroutines are called from a non-FORTRAN program, the FORTRAN conventions such as array ordering must be used.

ESSL functions are invoked from FORTRAN programs using a function reference, and ESSL subroutines are invoked from FORTRAN programs using a CALL statement. ESSL routines can be called from Assembler programs by coding an appropriate macro instruction, such as CALL, or by coding Assembler language branch instructions.

Most ESSL routines can be called from APL2 functions via Processor 11. APL2 and Processor 11 manage the necessary housekeeping and argument conversion based upon descriptive information contained in a NAMES file provided with ESSL. One important difference must be kept in mind when accessing ESSL routines from APL. In APL, arrays are stored in row major order, while in FORTRAN they are stored in column major order. Processor 11 does not modify array ordering. One solution is to transpose all input matrices before calling ESSL, and transpose all output matrices after the call. Of course, in some cases the cost of the transpositions may be more than the performance improvement obtained by calling ESSL. Another alternative is to build and process the matrix in its logically transposed form. Finally, it is sometimes possible to handle this problem by reversing arguments in the calling sequences of those ESSL routines that provide the capability of using either the normal or transposed form of the matrix. For additional information, see the APL2 Programming Guide.[23]

Because calling sequences are identical for the ESSL vector and scalar subroutines, it is not necessary to modify source code or recompile to switch between the scalar and vector libraries; the desired library is selected at link-edit or load time.

An attempt was made to be consistent across the library when the calling sequences were designed. Within each mathematical area, the calling sequence syntax and naming conventions are similar. An attempt was also made to achieve a balance between keeping the calling sequence as simple as possible and still allowing flexibility for the sophisticated user.

Inserting calls to ESSL subroutines in application programs is often very easy. For example, consider the following:

```
      DT = 0.0
      DO 10 J = 1, N
        DT = DT + X(I)*Y(I)
10 CONTINUE
```

The above loop can be replaced with the following function reference:[24]

```
DT = DDOT(N,X,1,Y,1)
```

(According to FORTRAN conventions, it would also be necessary to declare DT and DDOT as double-precision.)

Similarly, the loop

```
      DO 20 J = 1, N
      DO 10 I = 1,L
        S = 0.0
        DO 11 K = 1,M
          S = S + A(I,K)*B(K,J)
11      CONTINUE
        C(I,J) = S
10    CONTINUE
20 CONTINUE
```

can be replaced[24] with

```
CALL DGEMUL(A,LDA,'N',B,LDB,'N',C,LDC,L,M,N)
```

For other more involved computations (e.g., solving a system of linear equations), it may be possible to simply replace a call to a user-written or library subroutine with a call to a comparable ESSL subroutine.

As we have seen, it is sometimes very easy to insert calls to ESSL subroutines. In other cases, the obvious replacement of the inner loop by the comparable ESSL CALL may not be the most optimal modification. For example, consider the following code fragment from a FORTRAN program that typically occurs in an eigensystem analysis subroutine:

```
      DO 130 J = 1, L
        G = 0.0
        DO 110 K = 1, L
          G = G + Z(K,I)*Z(K,J)
110     CONTINUE
        G = G*H
        DO 120 K = 1, L
          Z(K,J) = Z(K,J) + G*Z(K,I)
120     CONTINUE
130   CONTINUE
```

The above code can be replaced with the following calls to ESSL subroutines:[24]

```
CALL DGEMV ('T',L,L,1.0D0,Z,LDZ,Z(1,I),1,0.0D0,AUX,1)

CALL DGER1(L,L,H,Z(1,I),1,AUX,1,Z,LDZ)
```

As discussed by Dongarra et al.,[25] the 110/130 loop is really a matrix–vector product for the transpose of a general matrix, and the 120/130 loop is really a rank-one update for a general matrix.

Error handling

Three different types of errors can occur when using ESSL subroutines: program exceptions, input argument errors, and computational errors. ESSL does extensive parameter checking, reporting multiple errors in one pass. Positive error messages specifying corrective action are issued, rather than describing what is incorrect.

Since ESSL uses FORTRAN error handling, all the features available for handling errors in FORTRAN are also available for ESSL errors. For example, it is possible to control the number of times an error is allowed to occur before the program terminates, the number of messages printed, whether an error should be considered terminal, and whether a traceback map will be printed. ESSL is shipped with recommended error-handling defaults that are tailored to the unsophisticated user. These defaults can be changed at installation, and can also be changed dynamically using the VS FORTRAN ERRSET facility. See the ESSL Guide Reference[2] for details.

Conclusions

In this paper, the ESSL subroutines have been discussed, and some of the techniques used to achieve high performance on the IBM 3090 Vector Facility have been illustrated. These techniques should be applicable to any application program developed for the IBM 3090 Vector Facility. Using ESSL in conjunction with the vectorizing VS FORTRAN Version 2 Compiler[9,10] should help application programmers to realize the full performance potential of the IBM 3090 Vector Facility rapidly and easily.

Acknowledgments

ESSL was a combined effort of the Engineering and Scientific Program Development Department, Kingston, NY; the T. J. Watson Research Center, Mathematical Sciences Department, Yorktown Heights, NY; and the European Center for Scientific and Engineering Computing, Parallel Processing Department, Rome, Italy, under the direction of the ESSL Product Manager, S. Schmidt. We would like to acknowledge the contributions of the developers (R. Agarwal, J. Cooley, F. Gustavson, L. Khislavsky,

J. McComb, D. Oakley, M. Pandian, G. Radicati Di Brozolo, J. Shearer, G. Slishman, J. Su, B. Tuckerman, M. Vitaletti, D. Wells, and L. Yao); the qualifiers (L. Ahern, M. Ivsin, D. Kapple, J. Martine, S. Matzou, A. Mistry, L. Wacker, and S. Wells); the writers (J. Druckerman, L. Mason, L. Michels, J. Mumper, and S. Tannenbaum); the planners (M. Ksoll, F. May, D. Parise, and J. Tether); the consultants (J. Dongarra and A. Dubrulle); and the Build and Test Group (D. Boyd, M. Ehmry, P. Edwards, P. Flowers, J. Jacobson, G. Lape, M. McMahon, E. Mahoney, S. Selzo, and C. Stoutenberg). We thank the people who were responsible for making ESSL accessible from APL2 programs (P. Furois, S. Suzzo, M. Van Der Muelen, and M. Wheatley), and the college cooperative students who contributed to the testing process (H. Akhter, D. Alberga, K. Barham, K. Chau, M. Ken, R. Keyes, A. Krammin, H. Shah, D. Shao, M. Silber, D. Smith, H. Soetjahja, S. Soubra, M. Taylor, E. Tsai, and K. Tsang). We acknowledge the support of our management (J. Cullum, A. Lim, B. Rudin, Y. Singh, T. Wilson, and S. Winograd), which was crucial to the success of ESSL. We also thank the many others who supported the successful delivery of the ESSL Program Product, and the referees for their helpful comments and suggestions.

Cited references and notes

1. *Engineering and Scientific Subroutine Library General Information*, GC23-0183, IBM Corporation; available through IBM branch offices.
2. *Engineering and Scientific Subroutine Library Guide and Reference*, SC23-0184, IBM Corporation; available through IBM branch offices.
3. W. Buchholz, "The IBM System/370 vector architecture," *IBM Systems Journal* **25,** No. 1, 51–62 (1986).
4. R. S. Clark and T. L. Wilson, "Vector system performance of the IBM 3090," *IBM Systems Journal* **25,** No. 1, 63–82 (1986).
5. 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).
6. Y. Singh, G. M. King, and J. W. Anderson, "IBM 3090 performance: A balanced system approach," *IBM Systems Journal* **25,** No. 1, 20–35 (1986).
7. S. G. Tucker, "The IBM 3090 system: An overview," *IBM Systems Journal* **25,** No. 1, 4–19 (1986).
8. *IBM System/370 Vector Operations*, SA22-7125, IBM Corporation; available through IBM branch offices.
9. *VS FORTRAN Version 2 Language and Library Reference*, SC26-4221, IBM Corporation; available through IBM branch offices.
10. *VS FORTRAN Version 2 Programming Guide*, SC26-4222, IBM Corporation; available through IBM branch offices.
11. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krough, "Basic Linear Algebra Subprograms for Fortran usage," *ACM Transactions on Mathematical Software* **5,** No. 3, 303–323 (September 1979).
12. D. S. Dodson and J. G. Lewis, "Proposed sparse extensions to the Basic Linear Algebra Subprograms," *ACM SIGNUM Newsletter* **20,** No. 1, 22–25 (January 1985).
13. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of Fortran Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software* **14,** No. 1, 1–17 (March 1988).
14. J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "Preliminary Proposal for a Set of Level 3 BLAS," presented at the Workshop on the Level 3 BLAS, Argonne National Laboratory, January 1987.
15. The stride of a vector is the increment used to step through an array to select the elements of a vector.
16. A. A. Dubrulle, "The Design of Matrix Algorithms for Fortran and Virtual Storage," Technical Report G320-3396, IBM Scientific Center, Palo Alto, CA (November 1979).
17. R. C. Agarwal, "An efficient formulation of the mixed-radix FFT algorithm," presented at the International Conference on Computers, Systems, and Signal Processing, December 10–12, 1984, Bangalore, India.
18. 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 (March 1986).
19. J. J. Dongarra, F. G. Gustavson, and A. Karp, "Implementing linear algebra algorithms for dense matrices on a vector pipeline machine," *SIAM Review* **1,** 91–112 (1984).
20. J. McComb, D. Kapple, and S. Schmidt, *Scientific and Technical Computing: Engineering and Scientific Subroutine Library Release 2 Performance on the IBM 3090 Vector Facility*, GG66-0276, IBM Corporation, available through IBM branch offices.
21. G. Radicati Di Brozolo and M. Vitaletti, "Sparse Matrix–Vector Product and Storage Representations on the IBM 3090," Technical Report G153-4098, IBM European Center for Scientific and Engineering Computing, Rome, Italy (1986).
22. G. Radicati Di Brozolo and M. Vitaletti, "Sparse Linear Systems Iterative Solvers on the IBM 3090 with Vector Facility," Technical Report ICE-0010, IBM European Center for Scientific and Engineering Computing, Rome, Italy (May 1987).
23. *APL2 Programming: Using the Supplied Routines*, SH20-9233, IBM Corporation, available through IBM branch offices.
24. Descriptions of the calling sequences and parameters for ESSL subroutines can be found in the ESSL Guide and Reference.[2]
25. J. J. Dongarra, L. Kaufman, and S. Hammarling, "Squeezing the Most Out of Eigenvalue Solvers on High-Performance Computers," Technical Memorandum 46, Argonne National Laboratory, Argonne, IL (January 1985).

General references

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

B. S. Garbow, J. M. Boyle, J. J. Dongarra, and C. B. Moler, *EISPACK Guide Extension: Lecture Notes in Computer Science* **51,** Springer-Verlag, New York (1977).

R. C. Grimes, D. R. Kincaid, and D. M. Young, *ITPACK 2.0 User's Guide*, CNA-150, Center for Numerical Analysis, University of Texas at Austin (1979).

D. R. Kincaid, T. C. Oppe, J. R. Respess, and D. M. Young, *ITPACKV 2C User's Guide*, CNA-191, Center for Numerical Analysis, University of Texas at Austin (1984).

C. L. Lawson and R. J. Hanson, *Solving Least Squares Problems*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1974).

P. A. W. Lewis, A. S. Goodman, and J. M. Miller, "A pseudo-random number generator for the System/360," *IBM Systems Journal* **8**, No. 2, 136–146 (1969).

B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler, *EISPACK Guide: Lecture Notes in Computer Science* **6**, Springer-Verlag, New York (1976).

Joan McComb *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Ms. McComb is an advisory programmer in the Engineering and Scientific Program Development Group in Kingston. She received a B.A. and M.S. in mathematics from New York University in 1976 and 1978, respectively, and an M.S. in computer engineering from Syracuse University in 1987. Ms. McComb joined IBM Owego in 1978, working on a real-time simulator for the LAMPS (Light Airborne Multipurpose System) project, and on the CPEXEC microcode for the IBM 3838 Array Processor. She transferred to Poughkeepsie in 1981 and became involved in design verification of large systems. Since 1984, she has worked on ESSL development in Kingston.

Stanley Schmidt *IBM Data Systems Division, Neighborhood Road, Kingston, New York 12401.* Mr. Schmidt is currently a senior engineering manager in the Engineering and Scientific Program Development Department in Kingston where he is responsible for the ESSL Program Product. He has B.S. and M.S. degrees in mathematics from the University of Wisconsin. He joined IBM in Poughkeepsie, New York, in 1963 in a Scientific Computations Department as a mathematical analyst. His primary involvement has been with numerical solutions to problems in circuit, device, and packaging analysis as well as algorithms for machine design. He has also participated in the development of APL mathematical functions. In 1972–73, Mr. Schmidt was a visiting Associate Professor at Hampton Institute, teaching numerical analysis and programming. In 1981 he joined Scientific and Engineering Processor Development where he worked on architecture issues. He subsequently initiated the ESSL development effort in which he continues to be presently involved. Mr. Schmidt has received an IBM Data Systems Division award for his ESSL contributions.

Reprint Order No. G321-5335.