GUIDELINES FOR DEVELOPING VECTORIZABLE COMPUTER PROGRAMS

E.W. Miner

Naval Research Laboratory
Washington, D.C. 20375

SUMMARY

This paper presents some fundamental principles for developing computer programs which are compatible with array-oriented computers. The emphasis is on basic techniques for structuring computer codes which are applicable in FORTRAN and do not require a special programming language or exact a significant penalty on a scalar computer. The intent is that researchers who are using numerical techniques to solve problems in engineering can apply these basic principles and thus develop transportable computer programs (in FORTRAN) which contain much vectorizable code. These principles are based primarily on the author's experience in running programs on the Texas Instruments Advanced Scientific Computer (TI-ASC), a vector processor, at the Naval Research Laboratory. The vector architecture of the ASC is discussed so that the requirements of array processing can be better appreciated. The "vectorization" of a finite-difference viscous shock-layer code is used as an example to illustrate the benefits and some of the difficulties involved. Increases in computing speed with vectorization are illustrated with results from the viscous shock-layer code and from a finite-element shock tube code. The applicability of these principles has been substantiated through running programs on other computers with array-associated computing characteristics, such as the Hewlett-Packard (H-P) 1000-F.

INTRODUCTION

The past decade has seen some considerable changes in the capabilities available to researchers involved in computational physics. Near the beginning of the last decade, scalar computers were the standard, but computers which would achieve higher computational speeds through parallelism or pipelining were already in the design stage and creating excitement among researchers. For example, the lead paper at the AIAA Computational Fluid Dynamics Conference in 1973 (ref. 1) was devoted to the future vector and parallel processors, their hardware, and their anticipated usefulness to the computational physics community. Since then, vector computers and other array oriented processors have become an actuality. The principal vector computers, for example, the CRAY-1, are still rather few in number and thus are available only to limited groups of researchers. However, other, more widely available computers have significant array-oriented features. In addition to the category of attached array processors, some mini-computers have array processing features. Specifically, the Hewlett-Packard (H-P) 1000-F series computers have what is called a Vector Instruction Set (VIS) implemented in firmware (microcode) which provides many of the benefits of vector programming and can increase computational speed by a factor of five or more. Even desktop computers (for example, the Tektronix 4050 series and the H-P System 35 and 45 desktops) have array oriented computational features in the BASIC language which they use. Thus, awareness of guidelines for developing computer codes which are compatible with array-oriented computing (i.e., vectorizable) should be advantageous to many researchers. Development of vectorizable codes should enhance code interchange and minimize reprogramming efforts when codes are exchanged between different computers.

Numerical techniques used by researchers to solve problems in computational physics typically are array oriented but common coding procedures sometimes reduce their compatibility with array processing. Such was the case with the particular program (a large boundary-layer type program) which is used as an illustration in this paper. By applying basic principles of "vectorization" as discussed in this paper, codes can be developed which are compatible with array processing and which require only a minimal, if any, increase in computing time on a scalar computer. In the case of the example program, the process of vectorization led to some code optimization and the scalar computing time was actually reduced.

Since FORTRAN is the most commonly used engineering programming language, attention is restricted in this paper to increasing the compatibility with array processing of programs coded in FORTRAN and is further restricted to FORTRAN code which is transportable. Although some of the examples used are specific to the Texas Instruments (TI) Advanced Scientific Computer (ASC), a vector processor, the examples illustrate considerations and techniques for developing codes which are compatible with array processing.

THE CONCEPT OF VECTORIZATION

In order to develop code which is vectorizable, it is necessary first to understand what vectorization is and how an array-oriented or vector computer differs from a scalar computer. The concept of vectorization is most easily introduced by example. Consider arrays A and B, each consisting of 100 numbers. Assume that one wishes to compute array C where $c_i = a_i + b_i$, $i = 1, 100$. The traditional scalar computer executes five assembly language instructions one hundred times. There are two memory fetches (a_i and b_i), one addition, one store to memory (for c_i), and an instruction that increments a counter, tests and branches back to load the next pair of input operands. Thus 500 scalar instructions are executed to add arrays A and B. A vector computer, or an array processor, has hardware which performs the 100 additions on the hundred pairs of input operands concurrently with the memory fetches and the stores to memory, greatly reducing the time required for computing array C. Such vector hardware may be available for virtually every arithmetic and logical operation.

The above example describes the singly subscripted FORTRAN DO loop:

```
      DO 100 I=1,100
      C(I) = A(I) + B(I)
100   CONTINUE
```

Doubly or triply subscripted arrays in loops nested 2 or 3 levels deep also may be collapsed on a vector machine into a single vector instruction.

Any FORTRAN DO loop representing one operation performed unconditionally on elements of one or two input arrays and producing elements of one output array is a candidate for a vector instruction and is thus vectorized. Vectorization may then be defined as getting as many operations as possible to vectorize. This requires designing, organizing, and writing programs so that the maximum possible number of arithmetic and logical operations can be executed as hardware vector instructions. Further, vectorization will be maximized when a programmer plans to operate on arrays of data instead of individual points of data. Such planning takes place at the program design level, at the subroutine level and at the line level within each subroutine.

ARRAY-ORIENTED PROGRAMMING

While vectorization is achieved by array-oriented programming, applications to specific computers may impose quite different constraints. For example, the attached support processors seem to be constrained by modest data transfer rates to and from the main computer's central memory. Thus the programmer would need to organize calculations such that many operations would be performed on the transferred data. In contrast, vector computers, such as the TI-ASC, can readily access large amounts of main memory and the vector architecture permits very rapid transfer of large arrays of data between the arithmetic units and main memory. In this case, the programmer needs to be less concerned with how the calculations are organized and can concentrate more fully on array-oriented programming.

For either hardware situation, array-oriented programming will require the following: choosing array-oriented algorithms which are vectorizable, planning program units which operate on arrays of data instead of points of data, minimizing the use of conditional operations, and planning array storage in memory for most rapid data transfer. These items are discussed more fully below.

Conditionality

A DO loop is the FORTRAN programmer's idiom for representing operations on arrays of data. The loop

```
        DO 100 I=1,50
        D(I)=A(I)*B(I)+C(I)
100     CONTINUE
```

represents two vector instructions; one which multiplies A and B, element-wise, and one which adds the elements of the product array to the elements of array C. The two instructions execute serially; the vector addition follows the vector multiplication. If this loop contains conditionality, e.g.

```
        DO 100 I=1,50
        IF(I.EQ.ITEST(I)) GO TO 100
        D(I)=A(I)*B(I)+C(I)
100     CONTINUE
```

it is no longer vectorizable. In this case the multiplication and addition may take place on some, but not all, of the array elements. An arithmetic or logical operation is vectorizable only if it is performed unconditionally on all elements of one or two input arrays to produce one resultant array.

Conditionality (the if-test) is often intrinsic to a computation, but significant vectorization may be achieved in the face of conditionality. A conditional operation can sometimes be transformed into one which is not conditional. Consider the loop

```
        DO 100 I=1,500
        IF(D(I).GT.DMAX) D(I)=DMAX
100     CONTINUE
```

which tests each element of array D and replaces only those values which pass the test. In the equivalent replacement loop

```
        DO 100 I=1,500
        D(I) = AMIN (D(I),DMAX)
100     CONTINUE
```

conditionality is eliminated and the operation is potentially vectorizable. In fact, vector machines may invoke a vector AMIN function which calculates the resultant vector D as a sequence of vector instructions on input vector D and scalar DMAX. Such vector functions are explained in a later section entitled "Vector Library Functions." Eliminating the "if-test" and consequently achieving array operations instead of scalar operations reduces the run time of the above example loop from 0.86×10^{-3} seconds to 0.4×10^{-4} seconds on the vector computer at NRL, a TI-ASC.

When using a vectorizing compiler an "if statement" within a loop will often inhibit vectorization of subsequent statements in the loop which are vector in character. Removing the "if-test" from the loop, or breaking the loop into several shorter loops, may result in significant vectorization. When the loop

```
      DO 100 I=1,500
      IF (X(I).GT.XMAX) X(I)=XMAX
      A(I)=C(I)*D(I)+X(I)
100   CONTINUE
```

is replaced by two loops

```
      DO 100 I=1,500
      IF (X(I).GT.XMAX) X(I)=XMAX
100   CONTINUE
      DO 110 I=1,500
      A(I)=C(I)*D(I)+X(I)
110   CONTINUE
```

its ASC execution time decreases from 0.17×10^{-2} seconds to 0.12×10^{-2} seconds. When conditionality is eliminated totally by using vector library function AMIN, the time drops to 0.86×10^{-4}. Minimizing the ill effects of conditionality is a central theme in the development of code which is compatible with vector computers and array processors.

Subroutine Organization for Array Operations

The fundamental principle for subroutine design is: plan, organize, and create subroutines which operate on arrays of data instead of points of data. For example, the program

```
      PROGRAM MAIN
      DIMENSON A(100),B(100),C(100)
      DO 20 I=1,100
      CALL SUB1 (A(I),B(I),C(I))
      CALL SUB2 (A(I),B(I),C(I))
      CALL SUB3 (A,(I),B(I),C(I))
20    CONTINUE
      END
```

locks the computation into scalar operations on points a_i,b_i,c_i and requires that the three subroutines be called 100 times each. The program above should be replaced by

```
      PROGRAM MAIN
      DIMENSION A(100),B(100),C(100)
      CALL SUB1 (A,B,C)
      CALL SUB2 (A,B,C)
      CALL SUB3 (A,B,C)
      END
```

where each subroutine operates on arrays A,B,C. This structure not only permits vectorized computation and but also minimizes costly subroutine linkage.

Vector Library Functions

Vectorizing compilers are built to apply the fundamental vectorization principle for subroutines. If a programmer codes

```
          DO 100 I=1,100
          B(I)=SIN(A(I))
    100   CONTINUE
```

a vectorizing compiler can be expected to collapse the loop into a single call to a vector sine function with input vector A and resultant vector B.

Scalar computers have one system FORTRAN library. When a trigonometric function, square-root, maximum/minimum function, etc. is invoked, a point-wise (scalar) function is called with a scalar answer. Vector computers have such scalar functions, and, in addition, have a library of vector functions which operate on arrays of points. Vector functions are themselves vectorized. Five hundred sine calculations on the ASC take 0.11×10^{-1} seconds when done in scalar mode and 0.16×10^{-2} seconds in vector mode.

Algorithms and Mathematical Methods

Vectorization principles governing the choice of algorithms and mathematical methods may be deduced from the line level and subroutine level principles previously discussed. Methods chosen should involve significant unconditional computation on large arrays of data. Algorithms which entail more arithmetic operations may be preferred over those involving fewer arithmetic operations which do not vectorize.

On the ASC, recursive computations are intrinsically unvectorizable. Consider the loop

```
          DO 100 I=2,100
          A(I)=A(I-1)*B(I)
    100   CONTINUE
```

where each element a_i of array A is computed from the element just previously computed, a_{i-1}. If this loop were performed in a vector mode, it would be equivalent to

```
          DO 100 I=1,100
          AA(I)=A(I)
    100   CONTINUE
          DO 110 I=2,100
          A(I)=AA(I-1)*B(I)
    110   CONTINUE
```

which yields different results from the original recursive code. The vectorizing compiler flags such loops as "vector hazards" and does not generate vector instructions for them.

When a recursive computation is required, it may be done in a loop by itself, isolated from other calculations. This prevents the vector hazard which it presents from inhibiting vectorization of subsequent calculations.

Memory Management

Vector instructions are most efficiently executed when the elements of operand arrays are stored, in central memory, contiguously with respect to the computation. The FORTRAN code

```
          DIMENSION A(10,50),B(50)
          DO 100 I=1,50
          A(K,I)=A(K,I)*B(I)
   100    CONTINUE
```

exhibits non-contiguity for input operand A. The FORTRAN dimension statement declares that A is a 2-dimensional array and is stored column-wise in central memory. The multiplication occurs, element-wise, on a row or A. Thus every 10th value of A as it resides in memory is input and output to this computation. This substantially reduces the speed of the vector computation. A preferable coding for this situation would be

```
          DIMENSION A(50,10),B(50)
          DO 100 K=1,10
          DO 100 I=1,50
          A(I,K)=A(I,K)*B(I)
   100    CONTINUE
```

With this arrangement (A is transposed) data streams from memory to the arithmetic unit quickly enough to ensure maximal execution speed.

Summary of Programming Principles for Vectorization

This list summarizes principles and guidelines already presented:

- Plan programs and subroutines which operate on arrays of data instead of points of data.
- Choose algorithms and mathematical methods which are array-oriented and vectorizable.
- Minimize and/or eliminate conditionality.
- Do not follow non-vectorizable calculations by vectorizable calculations in the same DO loop.
- Store vector operands contiguously in Central Memory.

THE TEXAS INSTRUMENTS ASC, A VECTOR COMPUTER

The rationale for developing computer codes compatible with array processing may be better appreciated by consideration of a specific system as an example. In some ways, the TI-ASC is a representative vector computer. The vectorizing FORTRAN compiler developed for the ASC recognizes array constructions in standard FORTRAN and generates vector instructions when appropriate. While it requires the programmer's attention to vectorization principles in the code, the compiler does not require special syntax or FORTRAN dialect to generate vector instructions. The ASC system thus illustrates the vectorization principles discussed above.

ASC Architecture

Three architectural features distinguish the Texas Instruments (TI) Advanced Scientific Computer (ASC). It is a pipeline computer; it has a full set of hardware vector instructions in addition to a full set of scalar instructions; and it is a multi-pipe computer.

An ASC arithmetic unit (AU) is logically and physically organized as a twelve-level pipe. Four levels are devoted to instruction decoding and processing, and eight to arithmetic or logical sub-operations. Thus when the AU is operating in scalar mode, up to twelve operations are concurrently at some stage of execution. At each CP clock cycle (80 nanoseconds) each arithmetic or logical operation in progress in the pipe drops to a lower level, and one answer may exit to the memory buffer. Pipe levels unnecessary to a particular instruction are bypassed. Memory buffers are considered part of the pipeline. Operands for calculations and answers are fetched and stored while the calculations are progressing through the pipe.

The most powerful computational capability of the ASC is its ability to run in vector mode. In this situation, a single operation is performed on many pairs of operands. For example, if A, B and C are vectors of length 100, only one vector instruction is needed for computing $c_i = a_i + b_i$; $i = 1, 100$. The A and B values stream continuously into the pipe, additions are performed in discrete steps within the pipe and answers flow back to central memory at the rate of one per clock cycle. The power of the vector instruction is that it guarantees optimum flow of calculations and data through the pipe.

An ASC may have one, two, three, or four pipes. The NRL computer has two pipes and, for fully optimal codes, can provide twice the computing power of a single pipe ASC.

The ASC Vectorizing Complier

A vectorizing/optimizing FORTRAN compiler, known as "NX", is available on the ASC. This compiler transforms ordinary FORTRAN code into vectorized object code which optimally exploits the ASC vector architecture. The NX compiler recognizes vectorizable FORTRAN constructions. When it fails to generate vector instructions, messages to the programmer may suggest how to rearrange or modify the code to achieve vectorization.

The NX compiler has three major levels of optimization. When invoking the compiler, a user specifies either I, J, or K level. An I level compile generates efficient, but unoptimized, scalar code. It is comparable to code generated by the IBM FORTRAN H compiler with OPT=0 or 1. At J level, the NX compiler generates optimized scalar code much like the IBM H compiler with OPT=2. Operating at level K, the NX compiler generates vectorized object code where possible, optimized scalar code elsewhere, and writes vectorization summaries and messages.

VECTORIZATION OF VISCOUS SHOCK-LAYER CODE AND COMPUTING TIME REDUCTIONS

To illustrate the process of vectorizing an existing code and to show the benefits which might be obtained, the vectorization of a moderately large FORTRAN program is discussed.

Description of Viscous Shock-Layer Program

The computer code which was vectorized is a laminar, hypersonic viscous shock-layer code previously developed by the author, (references 2-4). The code was written in FORTRAN and developed on an IBM 370/158. As discussed in references 2-4, the program uses an implicit finite-difference, marching integration procedure to solve the viscous shock-layer equations. Two flow field chemistries were available: dissociating oxygen and multi-component, ionizing air. As in the previous work of Davis (ref. 5), the governing equations are second-order accurate in the inverse Reynolds number parameter ϵ from the body to the shock.

By some criteria, the code might be a poor candidate for vectorization. The program has a significant amount of scalar code and, with 51 grid points used across the viscous layer, the arrays or vectors are much shorter than the vectors of length of 300 or more which have been often suggested for efficient pipeline use. In two ways this code is typical of large computer programs commonly used in solving engineering problems. First it was developed on a scalar processor, and second, efforts were made during its development to write code requiring minimum memory, not to write code that would vectorize. It was also coded in readily transportable FORTRAN. The size of the code, about 3000 FORTRAN statements, is perhaps typical of moderately large programs in use in solving engineering problems.

Computing Time Reductions

Since FORTRAN as implemented on the TI-ASC is very similar to IBM 370 FORTRAN, no changes were needed to run the code on the ASC. Runs were made to verify that the calculations of skin friction and surface heat transfer agreed with previously published results (ref. 2). After verifying the accuracy of the computed results, the program was compiled using the NX compiler at level I (no vectorization), and runs were timed. Other runs were made with the code compiled at J level to determine the gains in computing speed with scalar optimization and at K level to determine how much of the code would vectorize without further modification. Computing times for the viscous shock-layer code on the TI-ASC are given in table 1. The first three lines, for the "scalar" version of the code, give the times for the runs mentioned above. The optimiztion of the object code at J level reduced the computing time by 16%. At K level, enough code with in DO loops vectorized for an additional six percent reduction in the computing time.

As discussed earlier, it is often possible to get statements, which did not originally vectorize, to vectorize with only minor recoding. Recoding segments of the most repetitively used routines reduced computing time by a factor of 4. The computing time for the vectorized code is given in line 4 of table 1.

In vectorizing the code, it was necessary to add additional statements. The vectorized code contained 3497 FORTRAN statements compared with 3176 statements for the scalar version, though some of the additional statements were non-executable (e.g. DIMENSION and COMMENT) statements. The additional statements did not increase computation time when the vectorized version was run in a scalar mode. Line 5 of table 1 gives the computing time for the vectorized version of the code when compiled using the I (scalar) level of the NX compiler. Comparing the times in lines 1 and 5 shows a slight (4%) reduction in computing time for the vectorized version of the code when run in a scalar mode compared with the unvectorized version. The code in the vectorized version is just as transportable as the code in the scalar version and would be expected to run faster on a scalar processor than did the original code.

Table 1 shows the large reduction in time which was obtained by vectorizing the code. Table 2 lists the computing times for the code as various subroutines were modified. Most reductions in computing time were incremental except for subroutines VISCNA and WISUB which gave major reductions. These two subroutines calculate species and mixture properties at each point across the viscous layer. In the original code, the outer loops had the larger range (across the layer) and the inner loops had the smaller range (over the number of species, for example). While loops of length 6 will vectorize, the speed is comparable to scalar code speed. Most of the computing time reduction for these two routines was obtained by rearranging the loops so that the inner loops had the larger range and by eliminating conditionality from the inner loops. This gave typical vector lengths of 51 which run much faster than scalar speeds. It was also necessary to "code around" the exponentiation function (X**Y) which is not yet implemented as an ASC vector library function. Other runs were made with the viscous shock-layer code to determine how the number of grid points used in the program affected the computing time. These runs showed that increasing grid resolution is much less costly with a vectorized code running on a vector processor than with a scalar code running on a scalar processor.

We also considered how memory requirements are affected by vectorization and how scalar optimization affects computing time with the vectorized code. In many instances, the code had used scalar temporary variables within loops to conserve memory. In vectorizing the code, the scalar temporary variables caused problems. Either vectorization was inhibited or very inefficient vector code was generated by the NX compiler. By converting the scalar temporary variables to array temporary variables, the problems were overcome; but at the cost of some increase in memory requirements. However, the increase in memory can be minimized by storing the temporary arrays in a scratch common block which can be shared between routines.

A more complete discussion of the reductions in computing times for the viscous shock-layer code has been given by Miner and Brooks (ref. 6). Further information on the TI-ASC architecture and ASC programming considerations is given in references 6 and 7.

VECTORIZATION OF SHOCK TUBE CODE

During the past year, the author has had the opportunity to work with a shock tube code and make some vectorization tests with it. This particular code is a relatively small research code. It had been developed to investigate ways of solving the shock tube equations using a finite-element spatial discretization and various finite-difference techniques for the time integration. The program was developed (in mostly standard FORTRAN) on a minicomputer, a Hewlett-Packard (H-P) 1000-F. During development, the program was coded in standard FORTRAN and the firmware routines of the Vector Instruction Set (VIS) were not used.

The Vector Instruction Set is a group of firmware routines on the H-P 1000-F series computers and a group of equivalent software routines on the other H-P 1000 computers. The appropriate arithmetic operations have corresponding routines, and each routine is equivalent to a "DO" which performs that particular operation. The routines cannot be interrupted by the program logic and thus the programmer cannot include conditionality in these pseudo DO loops. The conditionality might still be coded, but it doesn't inhibit vectorization of neighboring code. A disadvantage of the VIS is that the programmer must vectorize the code explicitly by "commenting out" the old DO loop and inserting the VIS routine calls. This process can be somewhat cumbersome but not overly so. Since the readability of the code is reduced, it is convenient to retain the original code in comment lines. The principal factor motivating the use of the H-P VIS was not the cost of running the code on the H-P 1000 but the long execution time, thirty-five minutes. Fortunately, vectorizing the shock tube code was neither difficult nor time consuming. The vectorization was done in several stages and test runs were made to check results and computing time reductions. After the code had been mostly vectorized, an operations count indicated that about 95% of the candidate arithmetic operations had been replaced by vector routine calls. The computing time was reduced by a factor of six from 2100 seconds to 350 seconds.

It was also of interest to determine the computing times for this code on the TI-ASC. Runs were made with both the vectorizing NX compiler and the non-optimizing, non-vectorizing FX compiler. Since the FX compiler neither optimizes nor vectorizes the object code which it produces, it executes quite rapidly and is normally used for short test runs and code debugging. The resultant object code can be expected to execute slightly slower than the code from the NX compiler in level I. The nominal advantage of the FX compiler can be seen in the fact that it needed only 0.5 seconds to compile the shock tube code while the NX compiler in level K (vectorizing) required 8.0 seconds. The disadvantage of the FX compiler is really only in comparison to the vectorizing NX compiler. The shock tube code compiled using the FX compiler required 103.5 seconds to execute, while the NX-level K (vectorized) object code executed in only 6.25 seconds. For the shock tube code there was a much larger speed increase than for the viscous shock-layer code discussed above primarily because the shock tube code contained less scalar code and vectorized more completely.

Table 3 summarizes the computing times and relative computing speeds for the shock tube code. On the H-P 1000 vectorization increased the relative computing speed by 6, and on the ASC vectorization increased the computing speed by 16. The shock tube code provides an additional example of the potential benefit of designing a computer code so that vectorized object code can easily be generated.

SUMMARY

This paper presents some basic principles for writing FORTRAN code which is compatible with array processing. Many of these principles exact little, if any, penalty in computing time or memory requirements when used on scalar computers. All can be implemented in standard, transportable FORTRAN. When such guidelines are followed, substantial reductions in reprogramming effort will occur if the program is run on an array-oriented computer. Since vector computers, array processors and other array oriented computers are becoming more widely available, easy transportability between scalar and vector computers is a significant, desirable feature of FORTRAN programs.

These principles are illustrated by applying them to a viscous shock-layer code which was written for a scalar computer and then transported to the Texas Instruments Advanced Scientific Computer, a vector machine.

FORTRAN compilers and other software can be expected to recognize vectorizable FORTRAN constructions. The programmer must, however, be responsible for appropriate array-oriented program design, organization, and attention to the details of vectorization.

REFERENCES

1. Feustel, E. A.; Jensen, C. A.; and McMahon, F. H.: Future Trends in Computer Hardware, Proceedings of AIAA Computational Fluid Dynamics Conference, pp. 1-7, Palm Springs, CA., 1973.
2. Miner, E. W. and Lewis, C. H.: Hypersonic Ionizing Air Viscous Shock-Layer Flows over Sphere Cones, AIAA Journal, Vol. 13, January 1975, pp. 80-88.
3. Miner, E. W. and Lewis, C. H.: Computer User's Guide for a Chemically Reacting Viscous Shock-Layer Program, CR-2551, NASA, May 1975.
4. Miner, E. W. and Lewis, C. H.: Viscous Shock-Layer Flows for the Space Shuttle Windward Plane of Symmetry, AIAA Journal, Vol. 14, January 1976, pp. 64-69.
5. Davis, R. T.: Numerical Solution of the Hypersonic Viscous Shock-Layer Equations, AIAA Journal, Vol. 8, May 1970, pp. 843-851.
6. Miner, E. W. and Brooks, B. J.: Comparative Computer Times Between Vectorized and Scalar Versions of a Large Hypersonic Viscous Shock-Layer Code, AIAA Paper 78-1207, Seattle, WA., 1978.
7. Brooks, B.; Brock H.; and Miller, M : Guide to Vectorization on the Naval Research Laboratory's Texas Instruments Advanced Scientific Computer: Volume 1 Vectorization Primer, Memorandum Report 4102, Naval Research Laboratory, Washington, D.C., November, 1979.

TABLE 1.- COMPARISON OF COMPUTING TIMES ON THE TI-ASC FOR VISCOUS SHOCK-LAYER PROGRAM[a]

Code Version	Compiler Level[b]	Computing Time; sec
scalar	I	123.6
scalar	J	104.1
scalar	K	98.1
vectorized	K	25.5
vectorized	I	118.7

[a]Central Processor time, test case, 51 grid points across viscous layer.

[b]I level—scalar code only; J level—optimized scalar code; K level—vectorized code with scalar optimization.

TABLE 2.- COMPUTING TIMES ON THE TI-ASC AS SUBROUTINES VECTORIZED[a]

Subroutine Vectorized	Function of Subroutine	Computing Time; sec
- - -	Base Line Case	98.07
DERIV3	Array Differentiation	96.59
SOLVE	Tridiagonal Solver	96.16
ENERGY	Energy Eq. Coefficients	95.97
SMOMNT	S-Momentum Eq. Coefficients	96.04
NMOMNT	N-Momentum Eq. Coefficients	95.80
SPECIE	Species Eq. Coefficients	95.19
THERM	Thermodynamic Properties	93.84
WISUB	7 Species Production Terms	62.37
VISCNA	7 Species Viscosity and Conductivity	27.52
MASS	Continuity Eq. Integration	27.37
HCPA	Interpolation for H and C_p	25.50

[a]Central Processor time, 51 grid points across viscous layer.

TABLE 3.- COMPUTING TIMES AND SPEEDS FOR SHOCK TUBE COMPUTER CODE

Computer and Compiler	Computing Time; sec[a]	Relative Computing Speed
H-P, without VIS	2100	1/6
H-P, with VIS	350	1
ASC, FX[b]	103.5	3.4
ASC, NX-K	6.25	56

[a]Central Processor time

[b]CP time on the ASC with FX is estimated to be equivalent to CP time on an IBM 370/168.