LA-UR -77-234

TITLE: TO VECTORIZE OR TO "VECTORIZE": THAT IS THE QUESTION

AUTHOR(S): R. N. Remund and K. A. Taggart

SUBMITTED TO: National Science Foundation and University of Illinois at Urbana-Champaign Symposium April 1977

los alamos
scientific laboratory
of the University of California
LOS ALAMOS, NEW MEXICO 87544

Form No. 836
St. No. 2629
1/75

UNITED STATES
ENERGY RESEARCH AND
DEVELOPMENT ADMINISTRATION
CONTRACT W-7405-ENG. 36

TO VECTORIZE OR TO "VECTORIZE": THAT IS THE QUESTION

by

Remund, R. N. and Taggart, K. A.
Los Alamos Scientific Laboratory
Los Alamos, N.M. 87545
FTS Number: 843-6519 or 7322
Phone Number: (505) 667-6591 or 7322

ABSTRACT

Large computer codes can benefit from vectorization in two ways. The first is an increase in running speed and the second is more transparent and orderly code structure. However, not all operations in a large code are vectorizable in the usual sense. We have found that one can expand the scope of vectorization on both the CDC 7600 and CDC STAR-100. We present a general discussion of the considerations required when constructing a vectorized code. The characteristics and performance of a representative set of vector subroutines for the CDC 7600 are presented. These subroutines allow for a much broader definition of vector operations, and a single coding example serves as an indication that the concept of vectorization can be greatly expanded on the STAR-100.

INTRODUCTION

The vector approach to computation was conceived a number of years ago as a way to maximally exploit pipe-lined arithmetic units and highly interleaved and/or multi-word accessed memories. In this approach, single valued (scalar) operations are applied successively to the elements of one dimensional argument arrays (i.e., multi-dimensional vectors) to produce a result array (or a single result in the case of functions such as dot product). In programming terms such operations are represented by:

```
      DO 10 I=1,N
      R(I) = F(A1(I),A2(I),...,Ak(I))
   10 CONTINUE
```

where one or more of the A arrays may be R(I); and

```
      R=0
      DO 10 I=1,N
      R = F(A1(I),A2(K),...,Ak(I),R)
   10 CONTINUE
```

respectively.

In actual practice one or the other of two generalizations of such operations is required. In the first generalization, the assignment statement is performed for only a predetermined subset of the values of I. In the preceding DO-loops this is accomplished by making execution of the assignment statement conditional on the state of the I^{th} element of a control vector array, ICV. Thus the assignment is prefixed with either IF(ICV(I)) or IF(.NOT.ICV(I)) as required.

In the second generalization, the order of selection of elements in the result and/or argument arrays depends on a predetermined array of indices, IXV(I), such that $1 \leq IXV(I) \leq N$ with $IXV(I) = IXV(J)$ for one or more $J \neq I$ specifically allowed. In the preceding DO loops this is accomplished by replacing some or all occurrences of I in the assignment with IXV(I). Such

random accessing (or more appropriately, non-sequential accessing) of arguments and/or results is inherently foreign to the usual concept of vector operations.

It is interesting to note that the Univac 1103 provided hardware implementation for a limited set of vector operations in 1954. But without pipelined arithmetic units and interleaved main memory, this facility primarily made up for the machines total lack of index registers. Currently three computers provide hardware implementations (and the appropriate related architecture) for such operations. The Control Data STAR 100 and the Texas Instruments ASC both provide a very large set of operations while the Cray Research CRAY-1 provides a carefully selected set of basic operations.

It was not generally recognized that the Control Data 7600 was also well adapted to vectorized programming until McMahon, Sloan, and Long at Lawrence Livermore Laboratory developed a package of routines (STACKLIB) to emulate the vector functions of the STAR 100. In general codes organized to exploit these functions ran up to twice as fast as their "scalar" counterparts. For some codes such a performance gain could be obtained by specialized assembly-language programming of inner-loop computations. However, such coding is not only very poorly adapted to evolutionary development, but also is generally useless for other applications. Furthermore, many codes, especially large hydrodynamics codes, have no single inner-loop which accounts for a large fraction of computational time; but, rather, the time spent is evenly distributed throughout the code. Hence, the pseudo-vector approach on the 7600 is vastly more useful and cost effective. Vectorization, in general, also produces a more transparent, more flexible, and more compact set of code.

These results were known in 1973 when the authors began to develop a new large scale coupled hydro-heat flow code at the Los Alamos Scientific Laboratory. Also, machines with vector hardware were soon to be available

Consequently we decided to organize the code in a "vector" manner. We did not directly apply the STACKLIB vector subroutine package for three main reasons:

1) Certain specialized operations which were not efficient to implement in the STAR repertoire could be easily implemented as pseudo-vector operations on the 7600.
2) The control vector approach implemented on STAR was not the most effective approach for the 7600 and was not suitable for the specialized operations which were required.
3) The effort involved in modifying the calling sequences, mnemonic subroutine names, and control vector logic to conform to our conventions was too great.

Initially the code was written for the 7600 with vector functions implemented as FORTRAN DO-loops. These were then replaced by assembly-coded loops as the associated fraction of the computation time became large enough to justify or require it. Originally it was thought that this code would run on the STAR-100, so that some time was spent trying to see how some of the specialized functions could be efficiently implemented on this machine. When it became apparent that Los Alamos would not obtain a STAR-100 this effort was terminated. Consequently, the bulk of the considerations in this paper are aimed at the 7600.

CODE VECTORIZATION

The standard approach to vectorization is to decide which processes require sequential data access and which require nonsequential access. The sequential processes are then broken up into elementary vector operations and the nonsequential processes are either ignored or perhaps one finds sequential processes to replace them. In extreme cases one may feel required by machine

architecture to switch algorithms from one not amenable to vector processing to another less suitable algorithm which can be executed in a sequential manner. Our concept of vectorization on the 7600 is to break all processes up into a set of very basic operations which are then implemented in much the same manner, regardless of the fact that they may not be sequential. One can define exceptions which can be handled on an individual basis but there should not be too many of these and the reduction in running time obtained should be significant in each case. An example of such an exception will be given later.

IMPLEMENTATION APPROACH ON THE 7600

There are two major differences between our approach and that used in the existing vector computers. First, we choose to implement control-vectors and index vectors in precisely the same way, as arrays of indices with one index per word. While this would appear to be wasteful of storage, the actual amount involved turns out to be trivial and the savings in execution time is worthwhile. We use the convention that the effective end of the control vector is indicated by a zero index, and so a single test serves to control the loop.

Second, we emphasize the use of multi-operation functions (with and without control/index-vectors), for example:

```
      DO 10 I=1,N
      R(I) = A(I)*B(I)+C(I)*D(I)
   10 CONTINUE
```

which replaces three of the single-operation commands on STAR. The example given is, except for inclusion of a control/index-vector, the practical limit of complexity for the 7600 due to a shortage of registers. Also, this is about the limit one wishes to go to in proliferating routines. We did implement two more complex but highly specialized functions as pseudo-vector routines, but only because they were major consumers of central processor time. It is

worth noting that the CRAY-1 has a hardware feature which makes possible multi-operation vector computation. The motivation for this feature is precisely the same as ours: the reduction of redundant stores and re-fetches of intermediate results.

The following functions are typical of those implemented. It should be noted that the index I takes on all values from 1 to N.

$$R(I) = A(I)*B(I) + C(I) \quad (1)$$

$$R(I) = A(I)*B(I) + C(I)*D(I) \quad (2)$$

$$R(J(I)+K) = R(J(I)+K) + A(I)*W(I) \quad (3)$$

$$R(I) = R(I) + A(I)*B(J(I)+K) \quad (4)$$

The result rates achieved are respectively one result per 10.25, 12.25, 10.25, and 11.0 machine cycles on the 7600. The achievement of these rates depends on proper interleaving of operations so that an instruction is executed on every machine cycle. In particular, fetches of operands must be started on the previous pass through the loop so that they are ready in registers when needed. This means that there is a start up time associated with each subroutine. Furthermore, some time can be saved by doing two results each time through the loop (or better yet four) so as to better amortize the cost of incrementing the loop index and performing the conditional jump to the beginning. Of course the more doubling one does the longer the start up time. Also the loop must fit in the 7600 instruction stack for maximum execution efficiency and this limits the amount of doubling. In actual practice the lack of registers turned out to be the limiting factor as one started loosing cycles waiting for registers to become free. Since our application was oriented to short vectors and since the more complex functions ran into the register problem, we generally limited the routines to two results per loop. One other consideration is memory conflicts. If two vectors happen to start in the same memory bank, the second value cannot be fetched until the bank has recovered from the first fetch. These memory conflicts can be avoided if one is careful.

However, for random accesses on random length vectors we have found that a 13% to 18% degredation of running time from optimum performance is to be expected. Table I presents some typical results for 50 vectors selected randomly for operations 1) and 2) above. Results are for vector length of 100 and are based on 300 samples for each result. The start of each vector is 4*32 + m after the previous one where m is called the displacement. Results given represent maximum, minimum, and average number of cycles. Note that the results do not depend significantly on whether or not the result-vector is distinct from the input-vectors.

TABLE I. Timing results for randomly chosen input-vectors

	R=A*B+C			R=A*B+C*D		
Case 1: Result-Vector Distinct from Input-Vectors						
Displacement	Max	Min	Average	Max	Mtn	Average
0	3256	2767	2817	4516	3989	4006
5	3813	1022	1347	2630	1213	1806
10	2266	1022	1179	3035	1213	1517
15	1939	1022	1188	2961	1213	1429
20	2820	1022	1353	3990	1213	1799
25	1926	1022	1177	2346	1213	1424
30	1990	1022	1166	2393	1213	1508
Ratio (Average/Minimum) for Odd Displacements						
			1.17			1.18
Case 2: Result-Vector Not Necessarily Distinct from Input-Vectors						
0	3221	2756	2816	4661	3997	4003
5	1999	1022	1168	2346	1220	1411
10	2269	1022	1194	3089	1220	1551
15	1977	1022	1179	2341	1220	1398
20	2840	1022	1282	2099	1220	1720
25	2323	1022	1192	2770	1220	1455
30	1935	1022	1159	3099	1220	1550
Ratio (Average/Minimum) for Odd Displacements						
			1.16			1.17

It should be noted that operations 3) and 4) above are not true vector operations but this makes no difference on the 7600. In addition, one further step is possible, as exemplified by the following operation. This subroutine was written for purposes of improved performance, as it contributed a sizeable fraction of the overall running time of the code. The routine is

```
      DO 10 I=1,N
      J(I) = X(I)
      Z1 = X(I) - J(I)
      Z2 = Y(I) - AINT(Y(I))
      W1(I) = 0.25*Z1*Z2
      W2(I) = 0.25*Z2
      W4(I) = 0.25*Z3
   10 W5(I) = 0.25

      DO 20 I=1,N
      W3(I) = W2(I) - W1(I)
      W6(I) = 0.25 - W4(I)
      W7(I) = W4(I) - W1(I)
      W8(I) = 0.25 - W2(I)
   20 W9(I) = W6(I) - W3(I).
```

The routine is broken into two loops, each of which fits in the instruction stack. The "vectorized" routine runs 4.5 times faster than the FORTRAN version and requires 4.25 machine cycles per result produced.

It is hoped that the previous examples demonstrate that "vectorization" on the 7600 is not limited to the standard sequential data access definition.

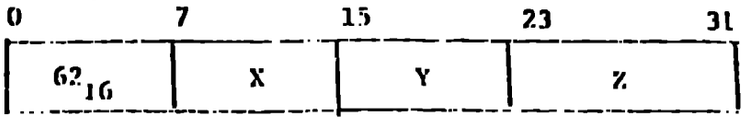
A STAR EXAMPLE

The following example shows that at least one nonsequential access function for which there is no STAR hardware implementation can be "vectorized" in the more general sense, even on STAR (a machine whose hardware is designed for sequential access operations).

What we have done is to compose a method for computing S(J(I))=S(J(I)) + W(I)*P(I) which uses some large subset of the 256 registers on the STAR-100 as a fast random access cache memory and creates the necessary executable

instruction string by means of vector operations alone. The instruction that can then be streamed through the central processor to perform the desired computation. This method is based on the recognition of two basic facts of STAR-100 architecture: 1) that scalar operations involving random access to central memory are very much slower than on the 7600: 2) it is possible to load scalar instructions four times faster than it is possible to execute them. So, by using the registers as a cache memory and by streaming the instructions to the CPU at near maximum rate one should be able to execute the scalar code at the maximum machine rate of 2 cycles/instruction (subject to register reservation conflicts). The rate of result production is 7 cycles/result which is comparable to the results obtained by hand coding the same operation on the 7600. This estimate assumes that one uses the code construction only once. In actual practice the same code would be used many times simply by changing the base address for the block transfers to the registers.

An outline of the procedure used is relatively simple. First the vector operation W(I)*P(I) = R(I) is done for all I. So our problem is trivially reduced to S(J(I)) = S(J(I)) + R(I). Next the instruction string is constructed. The method here is to see that, first some number of S values are block-fetched into the registers and then R values are brought into the registers a block at a time and added into the proper S register until the values of R are exhausted. In order for this to be done in a time competitive with the 7600 the stream of instructions must be constructed using only vector operations. The STAR-100 hardware instruction for scalar register addition is 62 ADD N; (X) + (Y) to (Z) with the format



The details of the method then become clearer. We need only make a stream of such instructions with the register needs as inserted in the proper places and intersperse 7D and 63 instructions as appropriate to load R values and to increment the starting address for the block of R values. All that is left then is to demonstrate that one can indeed construct the necessary instructions using only vector operations.

The actual building of the instruction stream requires three initial non-vector steps which need only be done once at the beginning of a program.

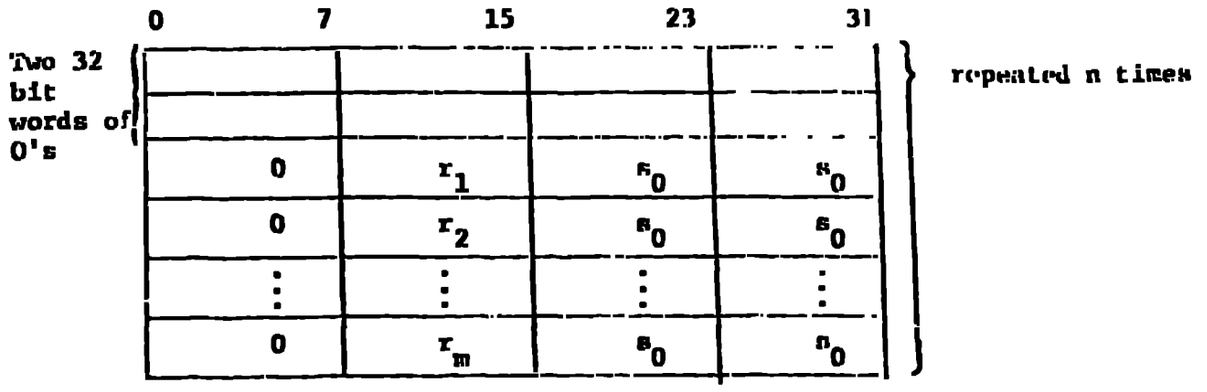
1. Construct a control vector (CV).

   $\underbrace{1\ 0\ .\ .\ .\ 0}_{m/2\ \text{bits}}\ 1\ 0\ .\ .\ .\quad 0\quad \text{etc.}$

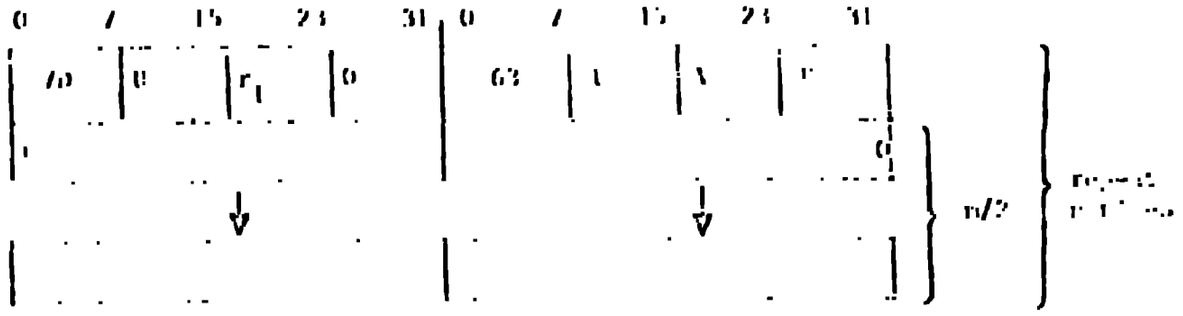
   replecated n times

   Note here that n x m = total number of R values.

2. Construct a register bias vector of 32 bit words (RBV).



Where $r_1 \rightarrow r_m$ are the register numbers for the m values of R swapped into the register file, and S_0 is the base register number minus 1 for the k values of S swapped into the register file.



Here U is the register containing the length of block to be transferred and the base address of B, and V is the register containing the increment to be added to [illegible] block transfer and the 63 instruction is the increment address instruction.

With these preliminaries out of the way and having previously constructed the index vector J(I) with the same structure as KEY we can now proceed to actually construct our code stream. The vector instructions used are

89 MPY L 32: J $\underbrace{* 101_{16}}_{\text{Broadcast}} \Rightarrow$ J1

80 ADD U 32; J1 + KEY $\Rightarrow$ J2

9B PACK 32; $\underbrace{62_{16}}_{\text{Broadcast}}$/J2 $\Rightarrow$ J3

BD MERGE 64 ; LTV,J3 $\Rightarrow$ CODE per CV

It should be noted that 32 or 64 denotes the number of bits in the operands. The result, CODE, is then ready to be streamed through the CPU.

CONCLUSION

The conclusion of this paper is, hopefully, obvious. There is vectorization and there is "vectorization". One should not be intimidated by vector hardware into believing that new algorithms are needed when all that may be needed is "vectorization".