Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M. Louter-Nool

Translation of algorithm 539:
Basic Linear Algebra Subprograms for Fortran usage
in FORTRAN 200 for the Cyber 205

Department of Numerical Mathematics Report NM-R8702 January

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Translation of Algorithm 539:

Basic Linear Algebra Subprograms for Fortran Usage

in FORTRAN 200 for the Cyber 205

Margreet Louter-Nool
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

This paper describes the vectorization of the BLAS, a set of linear algebra subprograms for FORTRAN usage. In [5], the efficiency of the BLAS, as standard available on the CDC Cyber 205, was examined and suggestions for improvements were given. This examination has led to the vectorized BLAS as presented here. Moreover, this version admits negative increment values; i.e. vectors can also be treated in reverse order. The number of data movements has been kept to a minimum. This BLAS version has been written in CDC FORTRAN 200. It has been optimized for a 1-pipe Cyber 205, but it is also appropriate for the 2- and 4-pipe versions.

1980 Mathematics subject classification: Primary:65V05. Secondary:65FXX
Key Words & Phrases: vectorization, basic linear algebra subprograms, FORTRAN 200, stride problems, operations on contiguously and non-contiguously stored REAL and COMPLEX vectors, operations on vectors stored in reverse order.
Note: This paper will be submitted for publication elsewhere.

1. INTRODUCTION

The BLAS[4], or, Basic Linear Algebra Subprograms, is a set of subprograms, performing operations on vectors. The BLAS has been implemented on many scalar machines in highly-efficient assembly language. By the increased use of vector computers and the large speed improvement that can be obtained by vectorizing the algorithms, it is desirable to implement the BLAS efficiently on the available vector computers. On the CDC Cyber 205, an implementation of the BLAS is already available (referred to as the CDC BLAS).

The Cyber 205 has been designed to operate on vectors stored in contiguous memory locations. If the elements of a vector are stored in non-contiguous memory locations, as is permitted in the BLAS, generally more time is spent to rearrange the elements than to accomplish the BLAS operation itself. In [5], we examined the efficiency of the CDC BLAS. It appeared that in case of non-contiguously stored vectors, a substantial speed up could be obtained.

Furthermore, two BLAS operations, the _ASUM operation, calculating the sum of the absolute values of a vector, and the _ROT operation, applying a plane operation, could be speeded up by us, the first one with a gain of 50 %, the second one with 70 % (_ROT4), or, even 50 % (_ROT3), independently of how the elements of the vector are stored in memory.

This motivated us to present here another vectorized version of the BLAS, referred to as the CWI BLAS. The CWI BLAS has been written in FORTRAN 200. Some features of this programming language are briefly discussed in Section 2, together with characteristics of the CDC Cyber 205. In

Report NM-R8702
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Section 3, we review some vector functions appropriate for handling non-contiguously stored vectors.

In the BLAS, the constant spacing between the vector elements is specified by an increment parameter. Positive as well as negative values are allowed. Like the CDC BLAS, we restricted ourselves to the REAL and COMPLEX versions of the BLAS; unlike the CDC BLAS, the CWI BLAS admits **negative** increment values. In Sections 4 and 5, we sketch how we treated vectors stored in reverse order (see [4] for a description of the implementation in case of positive increment values). In Section 6, some remarks are made concerning the DOUBLE PRECISION BLAS.

2. THE CYBER 205 AND THE PROGRAMMING LANGUAGE FORTRAN 200

The Cyber 205 can be programmed in the CONTROL DATA FORTRAN 200 programming language[2]. FORTRAN 200 is a superset of the American National Standards Institute FORTRAN language, which is described in ANSI document X3.9 - 1978.

The FORTRAN 200 compiler provides a set of features intended to use the vector processing hardware. The features are extensions to the standard FORTRAN language. Use of the vector programming features can decrease the amount of time needed to execute a program. The FORTRAN 200 language also includes a number of SPECIAL CALL statements that directly generate machine language instructions. We tried to avoid the application of the SPECIAL CALLs, but in some cases it was necessary to use them in order to obtain optimal code.

The Cyber 205 central processor consists of two parts; a scalar instruction processor and a vector instruction processor. For each arithmetic operation three steps must be carried out; the values are loaded from memory, the operation on the values is performed, and finally the result value is stored into memory. On a scalar machine these steps cannot be overlapped; a vector machine can perform the three steps simultaneously, if the operation is performed on a stream of values. This stream is called a vector. The values in a vector must be located in contiguous memory locations.

For the vectorization of the BLAS, this is a very important restriction, since the BLAS permits a non-unit constant spacing between the elements. This implies, that the vector elements must be rearranged to adjacent storage locations before a vector instruction can be executed. The constant spacing is commonly referred to as the **stride**. We are talking about a **stride problem**, if the increment value (cf. 3.1) unequals 1.

On scalar machines, this spacing hardly influences the execution time. On a vector machine, however, the execution time can be increased by a factor of 3 or even more, in case of a stride problem. For that reason, we put much effort into the implementation of the vector operations with nonunity increment values. Nevertheless, we would like to emphasize the following:

REMARK 2.1: If a non-contiguous vector is frequently used in a part of a FORTRAN program, it is wise to copy such a vector to a contiguously stored vector, e.g., by means of a BLAS _COPY routine, otherwise the execution time would be determined mainly by the number of data movements.

The vectorized BLAS presented here, was developed and tested on a 1-pipe CDC Cyber 205, located at the Amsterdam Academic Computer Centre (SARA). Cyber 205 versions with two or four pipes are also available from CDC. In case of more pipes, the data of one vector operation are evenly distributed, and the execution time is accordingly reduced. We emphasize, that a multiple pipe configuration can not be used to process two or more vector instructions in parallel.

On the Cyber 205, any correct ANSI FORTRAN 77 program can be compiled and executed successfully. This implies, that also programmers who have no special experience with vector programming, can execute their FORTRAN codes on a Cyber 205 vector machine. When using these vectorized BLAS routines a great part of the execution can be performed at vector speed. If the number of "interesting" elements (the value of the N parameter of the BLAS routine) is sufficiently large (more than about 50), the execution time will decrease extremely as compared with scalar execution.

We remark, that at the level of matrix-vector operations the application of a vectorized BLAS will not always result in the most optimal implementation. In [3], an extended set of the BLAS is

proposed to optimize at this level, in order to allow for the potential efficiency. In the NUMVEC library[6], developed at the Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, many of the proposed Extended BLAS routines have already been included. These routines have also been coded in CDC FORTRAN 200.

3. VECTOR FUNCTIONS FOR NON-CONTIGUOUSLY STORED VECTORS

Each BLAS subprogram contains parameters like N, X and INCX. Here X is a one-dimensional REAL or COMPLEX array, in which the elements $x(i)$ of the vector x are stored according to the indexing pattern

$$x(i) = \begin{cases} \text{X}(1 + (i - 1) * \text{INCX}) & \text{if INCX} > 0 \\ \text{X}(1 - (\text{N} - i) * \text{INCX}) & \text{if INCX} < 0 \end{cases} \tag{3.1}$$

for $i = 1, \cdots, \text{N}$. The parameter N presents the number of elements, on which the vector operation has to be carried out. The length NX of X must satisfy

$$\text{NX} \geqslant 1 + |\text{INCX}| * (\text{N} - 1)$$

Moreover, for REAL vectors the value of N should not exceed the number of data elements on a LARGE PAGE (65535). For COMPLEX and DOUBLE PRECISION vectors, the length is restricted to 32767.

The original BLAS permits an INCX value of 0, but we did not implement this option. As described in [4], negative increment values are only allowed, if two vector arguments are involved.

Before we review vector functions to handle non-contiguously stored vectors, we consider the execution time for a vector operation. The expected execution time Ψ, in cycles of 20 nanoseconds, for a vector instruction can be expressed by

$$\Psi = S + R * m \ / \ p \tag{3.2}$$

for the Cyber 205. S denotes the startup time for an instruction. In general, this value is in the range 50-70 cycles. R is a constant of proportionality, depending on the kind of instruction. The variable m is the length of the input or output vector, and p denotes the number of pipes of the Cyber 205. In this paper, we assume that

$$m = \text{NX} = 1 + |\text{INCX}| * (\text{N} - 1) \tag{3.3}$$

If N is large, then

$$m \approx |\text{INCX}| * \text{N} \tag{3.4}$$

and the startup value S can be ignored.

There are a few important cases, namely the GATHER and SCATTER instructions (discussed in this section), the DOT instruction (calculating the inner product of two vectors) and the SUM and PRODUCT instructions (calculating the sum and product of the vector elements, respectively), in which the execution time does **not** depend on p. The GATHER and SCATTER instructions are the only instructions we used for the implementation of the vectorized BLAS, with a value of R unequal to 1.

Summarizing, the expected execution time Ψ can be expressed by

$$\Psi = \Psi_{mp} \approx m \ / \ p \tag{3.5a}$$

for vector instructions, depending on the number of pipes, or, by

$$\Psi = \Psi_{m} \approx m \tag{3.5b}$$

for pipe-independent vector instructions.

The tools to handle strides are

1 GATHER/SCATTER. These machine instructions are available for moving data elements from

nonsequential to sequential memory locations (GATHER) and the reverse (SCATTER). Since we are only dealing with a constant spacing between the elements, we can use the **periodic** GATHER (Q8VGATHP) and the **periodic** SCATTER (Q8VSCATP). The value of R was determined to be 1.5. The execution time Ψ depends on the number of data elements to be moved (and not on the value of INCX), and is independent of the number of pipes. The execution time for both instructions is given by

$$\Psi_{GS} \approx 1.5 \text{ N}$$

We remark, that both positive and negative periods can be treated by these instructions.

2 REVERSE. The Q8VREV routine performs the operation

$$y(i) = x(m-i) \quad \text{for } i = 1, \ldots, m.$$

The execution time is given by

$$\Psi_{REV} \approx \begin{cases} m \ / \ p & \text{for } p=1,2 \\ m \ / \ 2 & \text{for } p=4 \end{cases}$$

If $m \neq$ N, then the subscript REV is preceded by the value of |INCX|.

3 COMPRESS/EXPAND. The vector function Q8VCMPRS creates a vector consisting of selected elements of the input vector. A BIT-vector specifies which elements are to be copied. The contrasting vector function Q8VXPND to expand a vector is not useful here, because this function creates a new vector with zeros at the bit 0 position. The suitable function to restore the compressed result vector to the original vector is the vector function Q8VMASK. The execution time for both functions depends on the length of the BIT vector, and, on the number of pipes. Hence, the execution time can be expressed by

$$\Psi = \Psi_{mp} \approx m \ / \ p$$

4 CONTROL VECTOR INSTRUCTIONS. As opposed to the other vector instructions to handle strides, each creating a new vector of selected elements, the control vector instructions are performed on the non-contiguously stored vector(s). A result value is only stored if the corresponding bit is set. Consequently, the execution time depends on the length of the whole vector and can be expressed by

$$\Psi = \Psi_{mp} \approx m \ / \ p$$

for pipe-dependent vector instructions, otherwise by

$$\Psi = \Psi_{m} \approx m$$

Control vector instructions are very convenient in case of a small positive stride. For that reason, they were frequently used in the implementation of the COMPLEX BLAS too.

As the execution time for the GATHER and SCATTER instructions does not depend on |INCX|, these instructions appear to be very efficient in the case of a large increment value. In cases of small increment values, the REVERSE, the COMPRESS/EXPAND, and the CONTROL VECTOR instructions are often faster. Moreover, also the number of pipes must be taken into account. We illustrate the application of some of these vector instructions in the following

EXAMPLE 3.1: Consider the copy operation SCOPY (N, X, 1, Y, −2). Its purpose is: copy the contiguously stored vector X (INCX=1) in reverse order to the vector Y (INCY=−2), such that only the odd elements of y are replaced. We mention two possibilities to perform this operation. Firstly, we could use the SCATTER instruction with a negative period of -2. Secondly, we could reverse and expand the vector X. The execution time for both solutions on 1-, 2- and 4-pipe machines are listed below.

method	Ψ	1-pipe	2-pipe	4-pipe
SCATTER	Ψ_{GS}	$1\frac{1}{2}$ N	$1\frac{1}{2}$ N	$1\frac{1}{2}$ N
REVERSE + MASK	$\Psi_{REV} + \Psi_{2Np}$	3 N	$1\frac{1}{2}$ N	N

Hence, on a 1-pipe the SCATTER is favourite, on the 4-pipe the second approach, and on a 2-pipe machine both methods are comparable.

We have optimized our BLAS version for a 1-pipe Cyber. Consequently, we prefer GATHER / SCATTER instructions over pipe-dependent instructions. In the Appendix, we give an overview of the special cases, that we distinguished when implementing the BLAS. A distinction is made between the execution time needed to perform the operation, and the time required for data movements. Obviously, on a 2- and certainly on a 4-pipe machine the number of special cases will further increase.

4. NEGATIVE INCREMENT VALUES

Negative increment values are only permitted for those subprograms that have two vector arguments. As pointed out in (3.1), the first element of array X with negative increment value INCX to be considered is

$$x(1) = \text{X}(1-(\text{N}-1) * \text{INCX}) \tag{4.1a}$$

and the last element is

$$x(\text{N}) = \text{X}(1) \tag{4.1b}$$

In the previous Section, we discussed two methods to reverse the order of storage. The vector intrinsic function Q8VREV can only be applied efficiently, if the increment value is small in magnitude. The execution time depends on the length of the input vector, which is roughly |INCX|*N. Since, in general, only N elements are needed for further computation, this operation should be preceded by a compress of the vector. The second method for reversing a vector is the periodic GATHER instruction with a negative period. For that purpose, the operation starts with the last relevant element, i.e. X(1−(N−1)*INCX) (cf. (4.1a)). The desired elements are copied and stored in reverse order, by one vector operation. The execution time for a GATHER operation with negative period is comparable to the positive period case (cf. [5], Appendix II). So, if the elements are widely scattered over the vector, this method is to be preferred.

4.1. Two negative increments

Both methods to reverse a vector were frequently used in the implementation of the BLAS, but their usage was kept to a minimum. It is recommended in [4], that when both increment values are negative, the process should be carried out in reverse storage order. It will be clear, that on a vector computer this implies a lot of probably superfluous, reverse operations. We shall illustrate this in the next

EXAMPLE 4.1.1: Again we consider a copy operation, SCOPY (N, X, −1, Y, −1). Following [4], we should create the reverse of X(1;N), and copy this reverse into a temporary vector, and finally, we should reverse this temporary vector into Y(1;N). This whole operation will cost $3\Psi_{Np}$. However, a direct copy of X into Y

```
Y(1;N) = X(1;N)
```

at the cost of Ψ_{Np}, will deliver the same output vector decreasing the execution time by a factor of 3.

We assume that it does not really matter in which direction the process is carried out (though, e.g., for the _DOT operation the numerical result may differ slightly, if the inner product is calculated from 1 to N instead of from N to 1). Therefore we have chosen to save computing time, and, if both vectors would have a negative increment value, we act as if both were positive. If one really wants to compute the inner product of two vectors stored in reverse order (INCX $<$ 0, INCY $<$ 0), we recommend the use of a _COPY operation, like

```
ASSIGN XD, .DYN.N
ASSIGN YD, .DYN.N
CALL SCOPY (N, X, INCX, XD, 1)
CALL SCOPY (N, Y, INCY, YD, 1)
DOT = SDOT (N, XD, 1, YD, 1)
```

Finally, we remark that the _DOT operation is performed in DOUBLE PRECISION (by means of the special call CALL Q8DOTV (see [1])).

4.2. One positive and one negative increment value

Also in cases of one positive and one negative increment value, the number of reverse instructions can be minimized. We shall illustrate this by means of two examples.

EXAMPLE 4.2.1: Consider the SAXPY operation ($y := a.x+y$) with INCX$=1$ and INCY$=-1$. A direct approach involves a reverse of the vector y, followed by the computation of the LINKED TRIAD, and again a reverse of the result vector into y.
An alternative solution is to copy the vector x, instead of the vector y, in reverse order into a temporary vector. This saves one reverse operation.

EXAMPLE 4.2.2: Consider the same operation as in the above example, but now with INCX$=-1$ and INCY>1. Assume that the GATHER operation is the most efficient way to rearrange the elements of y. If the elements of y are gathered and scattered in reverse order, the reverse operation of x can be saved.

From both examples it will be clear, that each combination of increment values requires its own approach. Again, we refer to the Appendix for an overview of the special cases and their implementation.

5. THE COMPLEX BLAS

The best way to handle COMPLEX vectors is to use two REAL arrays for the representation of a COMPLEX vector. Unfortunately, the BLAS requires arrays of type COMPLEX. Let us consider the storage of a COMPLEX vector; a COMPLEX vector x is stored in the following way

$$\mathrm{Re}\, x_1 \quad \mathrm{Im}\, x_1 \quad \mathrm{Re}\, x_2 \quad \mathrm{Im}\, x_2 \quad \cdots$$

In regarding a COMPLEX vector as a two dimensional REAL array, like

```
REAL X (2, N)
```

we observe, that operating on the real part of the vector, i.e., on the first row of X, results in a stride problem of size 2. There are two possibilities to deal with such stride problems.

1 We can use control vectors to operate on only the real, or only the imaginary part. As a consequence, each operation takes twice as much computing time, since the period (or stride) is 2.
2 We can create temporary vectors with contiguously stored real parts and with contiguously stored the imaginary parts. Obviously, creating such vectors and finally restoring them to the original COMPLEX storage mode will cost extra execution time too.

In [5], section 5.3, it is pointed out, that in case of increment values equal to 1, the use of control vectors is to be preferred over the creation of separated vectors.

5.1. Negative increment values

In the implementation of the REAL BLAS we frequently use the Q8VREV routine for reversing a vector. When we use this routine to reverse the order of a COMPLEX vector, we obtain

$$\mathrm{Im}\, x_N \quad \mathrm{Re}\, x_N \quad \mathrm{Im}\, x_{N-1} \quad \mathrm{Re}\, x_{N-1} \quad \cdots$$

Hence, for a simple copy operation like CALL CCOPY (N, CX, −1, CY, 1), a call of Q8VREV will not suffice, because the order of the real and imaginary part of a COMPLEX number is changed too. The special call CALL Q8VREVV, as opposed to the vector intrinsic function Q8VREV, enables us to operate on a vector under control of a BIT vector. The elements of the output vector, whose BITS are set, are replaced by the corresponding elements of the reversed input vector.

For negative increment values unequal to -1, this method is too expensive for COMPLEX vectors. Its execution time depends on the length of the BIT vector and is about 2*|INCX|*N. Again the periodic GATHER seems to be the most suitable alternative. However, the periodic GATHER collects just equidistant vector elements, and not, e.g., two adjacent elements per period as is required for COMPLEX vector elements. As a consequence, the periodic GATHER - and analogously the periodic SCATTER - has to be applied twice.

In the following FORTRAN 200 codes the DESCRIPTOR REP10D is associated with the BIT vector REP10 :

```
BIT    REP10 (65536) /32768 * B'10'/
```

and, X and Y are two-dimensional REAL arrays, corresponding to the COMPLEX vectors CX and CY, respectively.

METHOD I: This method was used for cases in which INCX * INCY = −1, like CCOPY (N, CX, 1, CY, −1), or, CCOPY (N, CX, −1, CY, 1).

```
NT2M1 = 2*N-1
ASSIGN REP10D, REP10(1;NT2M1)
CALL Q8VREVV (X'00',,X(1,1;NT2M1),,,REP10D,Y(1,1;NT2M1))
CALL Q8VREVV (X'00',,X(2,1;NT2M1),,,REP10D,Y(2,1;NT2M1))
```

Here, the real and imaginary part of X are copied in reverse order into vector Y. Note that CALL Q8VREVV can not deliver a contiguously stored vector of only the real, or, of only the imaginary part, because the BIT vector controls the output vector.

METHOD II: This method deals with the periodic GATHER operation. It was applied for cases in which a non-contiguously stored complex vector had to be copied in reverse order into a contiguously stored vector, e.g., CCOPY (N, CX, −K, CY, 1), and, CCOPY (N, CX, K, CY, −1) for positive

values of K. Two temporary vectors REXD and IMXD, both of length N, are created.

```
IPER = 2 * IABS (INCX)
ASSIGN REXD, .DYN. N
ASSIGN IMXD, .DYN. N
REXD = Q8VGATHP (X(1,N;1), N, IPER; REXD)
IMXD = Q8VGATHP (X(2,N;1), N, IPER; IMXD)
ASSIGN REP10D, REP10(1;2*N)
Y(1,1;1) = Q8VMERG (REXD, IMXD, REP10D; Y(1,1;1))
```

The real part is gathered in reverse order into REXD, and the imaginary part into IMXD. A MERGE operation under control of the BIT descriptor REP10D completes the operation.

The execution time for both methods (cf. Section 3) is given by:

increments	reverse	Ψ	1-pipe	2-pipe	4-pipe
INCX * INCY = -1	METHOD I	$2\Psi_{2REV}$	4 N	2 N	2 N
INCX * INCY < -1 $\wedge$|INCY| = 1	METHOD II	$2\Psi_{GS}+\Psi_{2Np}$	5 N	4 N	$3\frac{1}{2}$ N

REMARK 5.1.1: Up to now, we did not mention the indexed GATHER. This instruction moves elements described in an index array of type INTEGER to adjacent memory locations. Although it seems to be a suitable instruction to reverse a COMPLEX vector - with real and imaginary parts at once in the right order -, it turns out to be less efficient than the periodic GATHER applied twice. We observed more than 5 cycles per COMPLEX element for negative increment values. Moreover, the execution time required to create such INTEGER index arrays increases the total CPU time (about 2N operations on a 1-pipe machine).

If in the process of changing the storage order of a COMPLEX vector two separated vectors are created, these separated vectors are used in the succeeding operations. The final COMPLEX output vector is constructed by means of a MERGE operation (if |INCX| = 1) or a SCATTER operation (if |INCX| $\neq$ 1).

In [5], tables 5.5.1, 5.6.2, A1.3 and A1.4, execution times for the COMPLEX CWI BLAS and the COMPLEX CDC BLAS are compared. The results show the substantial gain obtained by performing the operations on separated vectors, as we did, instead of on vectors in original COMPLEX vector storage mode, as was done in the CDC BLAS.

Evidently, the number of reverse operations was reduced to a minimum as was pointed out in section 4.1 and 4.2. For the implementation, for the case that at least one of the increment values is unequal to ± 1, the reader is referred to section 5.5 and 5.6 in [5], and to the Appendix of this paper.

REMARK 5.1.2: Only for the implementation of CDOTU with INCX * INCY = -1, the Q8VREV routine could be applied successfully. Just because the order of the imaginary and real part is changed, the number of vector operations can be decreased, since in this case, two inner products can be calculated simultaneously. In the implementation of CDOTC with INCX * INCY = 1 (cf. [5], section 5.5), an analogous situation occurs. CDOTU computes the inner product, and CDOTC the conjugated inner product of two COMPLEX vectors.

6. THE DOUBLE PRECISION BLAS

The hardware of the Cyber 205 handles only INTEGERs, HALF PRECISION and FULL PRECISION floating-point numbers. Hence, the DOUBLE PRECISION floating-point arithmetic must be handled by the software. The storage of a DOUBLE PRECISION vector resembles the storage of a COMPLEX vector; the upper and lower parts alternate. Similar to COMPLEX arithmetic, execution time has to be spent for data movements in order to obtain two contiguously stored arrays containing the upper parts and the lower parts, respectively. Therefore, it is better to start with two REAL arrays representing a DOUBLE PRECISION array. Unfortunately, the BLAS does not permit such a structure.

One reason for not implementing the DOUBLE PRECISION BLAS on a Cyber 205 could be the fact, that the hardware employs 64-bit full words and 32-bit half words. We believe that an implementation of a HALF PRECISION BLAS makes more sense than a DOUBLE PRECISION BLAS on such a machine. Computation in HALF PRECISION decreases the execution time for a vector arithmetic operation with a factor of about 2. On the other hand, e.g., the _AXPY operation on DOUBLE PRECISION (128-bits) vectors (contiguously stored) will take 20 cycles, and, on SINGLE PRECISION (64-bits) vectors 1 cycle per element on a 1-pipe Cyber 205.

For further information on the DOUBLE PRECISION BLAS, we refer to [7], in which a review is given of the operations that are needed, and their expected execution times on a 1-, 2- and 4-pipe machine.

REFERENCES

1. CDC. *Cyber 200 ASSEMBLER reference manual*, version 2, publ. number 60485010.
2. CDC. *Cyber 200 FORTRAN reference manual*, version 1, publ. number 60480200B.
3. J.J. DONGARRA, J. DU CROZ, S. HAMMERLING and R.J. HANSON (November 1985). A proposal for an extended set of Fortran basic linear algebra subprograms, *Technical Memorandum 41 (revision 1), Argonne National Laboratory.*
4. C.L. LAWSON, R.J. HANSON, D.R. KINCAID and F.T. KROGH (1979). Basic Linear Algebra Subprograms for FORTRAN usage, *ACM Transactions on Mathematical Software*, 5, 308-323.
5. M. LOUTER-NOOL. Basic Linear Algebra Subprograms (BLAS) on the CDC Cyber 205, *Parallel Computing (to appear).*
6. NUMVEC. *A library of NUMerical software for VECtor and parallel computers in FORTRAN*, Centre for Mathematics and Computer Science, Amsterdam.
7. J. SCHLICHTING. Double precision BLAS, to appear in: *Proceedings of the (CWI, UvA, THD)-Colloquium on Numerical Aspects of Vector and Parallel Processors, CWI, Amsterdam.*

Appendix

This Appendix gives a complete summary of the execution times for all BLAS subprograms as a function of N. This value N is an input parameter of all subprograms and represents the number of vector elements on which the operation must be carried out. The execution time Ψ is expressed by:

Ψ_{iNp}	$\approx$	iN/p	- for one vector operation on iN elements, $i=1,2,\cdots$; Ψ_{iNp} depends on p, the number of pipes.
Ψ_{iN}	$\approx$	iN	- for one vector operation on iN elements; Ψ_{iN} does not depend on p.
Ψ_{GS}	$\approx$	1.5N	- for one GATHER or one SCATTER operation on N elements. Ψ_{GS} does not depend on p.
Ψ_{iREV}	$\approx$	iN/p	- for one reverse operation on iN elements. On a 4-pipe $\Psi_{iREV} = iN/2$.

A distinction has been made between execution times for data movements (Column 5), like COPY, REVERSE, GATHER, SCATTER, MERGE, and MASK operations, and for arithmetic vector operations (Column 4), like LINKED TRIADs, DOT products and so on.

In the Columns 6, 7, and, 8, the execution times are given as a function of N, for a 1-, 2- and 4-pipe Cyber 205, respectively. Multiplying these values by the cycle time of 20 nanoseconds will give a quite good impression of the required CPU time in practice. In Column 9, the really obtained execution times in seconds on a 1-pipe Cyber 205 are listed for N = 5000. The listed values are average values over 100 calls.

For the REAL as well as the COMPLEX _ROT operation, two implementations are available; i.e. _ROT4 and _ROT3. Both perform the same BLAS operation, however, with different execution times. The numerical results may differ slightly. For further information we refer to [5], Section 4.

Finally, the listed increment values (or pair of increment values) in Columns 2 and 3 are representing special cases in the CWI BLAS. Again we stress, that if both increment values would be negative, we act as if both were positive.

Some remarks on the implementation of the CWI BLAS:

- Control or BIT vectors needed for CONTROL VECTOR instructions are initialized by means of a DATA statement, so that the use of these vectors will not charge the user's execution time.
- The execution times for the _COPY operations for the important cases (|INCX| = 1, |INCY| = K), and, (|INCX| = K, |INCY| = 1), for small values of K, could be improved on a 2- and 4-pipe machine by using the COMPRESS/MASK vector instructions.
- For the ISAMAX and ICAMAX functions, it turns out that the execution time required to execute CALL Q8MAX, which is called by these functions, depends on the relative magnitude of the elements of the vector. Each new maximum requires 16 additional cycles. The execution times as given in Columns 6, 7, and, 8 are obtained for monotonically decreasing vectors.
- For the REAL BLAS special cases with CONTROL VECTORs were implemented, if both increments are equal and small, or, in case of one - also small - increment. For the COMPLEX BLAS such cases were not considered. On a 2- or 4-pipe machine, CONTROL VECTOR operations are more lucrative. Consider for example the execution time for SAXPY as a function of N. For the 1-pipe, the maximum increment value for which a CONTROL VECTOR operation is still profitable is 5. On a 2-pipe this value is 10, and, on a 4-pipe, a LINKED TRIAD with a CONTROL VECTOR up to a period of 19 is still more efficient than GATHERing and SCATTERing.
- If separated vectors for the real and imaginary parts are used in a COMPLEX BLAS routine, then these vectors are stored after each other in the same array. Obviously, if for the real and imaginary part the same operation is required, this operation can be carried out only once, saving one startup time. Hence, the execution time is then given by Ψ_{2Np}, or, Ψ_{2N}, and not by $2\Psi_{Np}$, or, $2\Psi_{N}$.

subprogram	INCX	INCY	execution times for the vector instructions	execution times for the data movements	total execution times 1-pipe	total execution times 2-pipe	total execution times 4-pipe	measured times on 1-pipe * 100
ISAMAX	1		Ψ_N		N	N	N	.011
	2		Ψ_{2N}		2 N	2 N	2 N	.021
	else		Ψ_N	Ψ_{GS}	$2\frac{1}{2}$ N	$2\frac{1}{2}$ N	$2\frac{1}{2}$ N	.027
SASUM	1		$\Psi_{\frac{1}{2}Np} + \Psi_{\frac{1}{2}N}$		N	$\frac{3}{4}$ N	$\frac{5}{8}$ N	.012
	2		$\Psi_{Np} + \Psi_N$		2 N	$1\frac{1}{2}$ N	$1\frac{1}{4}$ N	.022
	else		$\Psi_{\frac{1}{2}Np} + \Psi_{\frac{1}{2}N}$	Ψ_{GS}	$2\frac{1}{2}$ N	$2\frac{1}{4}$ N	$2\frac{1}{8}$ N	.028
SNRM2	1		Ψ_N		N	N	N	.011
	2		Ψ_{2N}		2 N	2 N	2 N	.021
	else		Ψ_N	Ψ_{GS}	$2\frac{1}{2}$ N	$2\frac{1}{2}$ N	$2\frac{1}{2}$ N	.027
SDOT	1	1	Ψ_N		N	N	N	.011
	-1	-1						
	1	-1	Ψ_N	Ψ_{REV}	2 N	$1\frac{1}{2}$ N	$1\frac{1}{2}$ N	.022
	-1	1						
	±1	else	Ψ_N	Ψ_{GS}	$2\frac{1}{2}$ N	$2\frac{1}{2}$ N	$2\frac{1}{2}$ N	.026
	else	±1						
	2	2	Ψ_{2N}		2 N	2 N	2 N	.022
	-2	-2						
	3	3	Ψ_{3N}		3 N	3 N	3 N	.032
	-3	-3						
	4	4	Ψ_{4N}		4 N	4 N	4 N	.042
	-4	-4						
	else	else	Ψ_N	$2\Psi_{GS}$	4 N	4 N	4 N	.042
SAXPY	1	1	Ψ_{Np}		N	$\frac{1}{2}$ N	$\frac{1}{4}$ N	.011
	-1	-1						
	1	-1	Ψ_{Np}	Ψ_{REV}	2 N	N	$\frac{3}{4}$ N	.022
	-1	1						
	else	±1	Ψ_{Np}	Ψ_{GS}	$2\frac{1}{2}$ N	2 N	$1\frac{3}{4}$ N	.026
	±1	else	Ψ_{Np}	$2\Psi_{GS}$	4 N	$3\frac{1}{2}$ N	$3\frac{1}{4}$ N	.039
	2	2	Ψ_{2Np}		2 N	N	$\frac{1}{2}$ N	.022
	-2	-2						
	3	3	Ψ_{3Np}		3 N	$1\frac{1}{2}$ N	$\frac{3}{4}$ N	.032
	-3	-3						
	4	4	Ψ_{4Np}		4 N	2 N	N	.042
	-4	-4						
	5	5	Ψ_{5Np}		5 N	$2\frac{1}{2}$ N	$1\frac{1}{4}$ N	.052
	-5	-5						
	else	else	Ψ_{Np}	$3\Psi_{GS}$	$5\frac{1}{2}$ N	5 N	$4\frac{3}{4}$ N	.056
SSCAL	1		Ψ_{Np}		N	$\frac{1}{2}$ N	$\frac{1}{4}$ N	.011
	2		Ψ_{2Np}		2 N	N	$\frac{1}{2}$ N	.021

TABLE A.1 Execution times for REAL BLAS routines, Part I.

subprogram	INCX	INCY	execution times for the vector instructions	execution times for the data movements	total execution times 1-pipe	total execution times 2-pipe	total execution times 4-pipe	measured times on 1-pipe * 100
SSCAL	3		Ψ_{3Np}		3 N	$1\frac{1}{2}$ N	$\frac{3}{4}$ N	.031
	4		Ψ_{4Np}		4 N	2 N	N	.041
	else		Ψ_{Np}	$2\Psi_{GS}$	4 N	$3\frac{1}{2}$ N	$3\frac{1}{4}$ N	.045
SSWAP	1	1		$3\Psi_{Np}$	3 N	$1\frac{1}{2}$ N	$\frac{3}{4}$ N	.032
	-1	-1						
	1	-1		$\Psi_{Np}+2\Psi_{REV}$	3 N	$1\frac{1}{2}$ N	$1\frac{1}{4}$ N	.032
	-1	1						
	±1	else		$\Psi_{Np}+2\Psi_{GS}$	4 N	$3\frac{1}{2}$ N	$3\frac{1}{4}$ N	.039
	else	±1						
	else	else		$4\Psi_{GS}$	6 N	6 N	6 N	.061
SCOPY	1	1		Ψ_{Np}	N	$\frac{1}{2}$ N	$\frac{1}{4}$ N	.011
	-1	-1						
	1	-1		Ψ_{REV}	N	$\frac{1}{2}$ N	$\frac{1}{2}$ N	.011
	-1	1						
	±1	else		Ψ_{GS}	$1\frac{1}{2}$ N	$1\frac{1}{2}$ N	$1\frac{1}{2}$ N	.015
	else	±1						
	2	2		Ψ_{2Np}	2 N	N	$\frac{1}{2}$ N	.022
	-2	-2						
	3	3		Ψ_{3Np}	3 N	$1\frac{1}{2}$ N	$\frac{3}{4}$ N	.032
	-3	-3						
	else	else		$2\Psi_{GS}$	3 N	3 N	3 N	.032
SROT4	1	1	$4\Psi_{Np}$		4 N	2 N	N	.042
	-1	-1						
	1	-1	$4\Psi_{Np}$	$2\Psi_{REV}$	6 N	3 N	2 N	.062
	-1	1						
	±1	else	$4\Psi_{Np}$	$2\Psi_{GS}$	7 N	5 N	4 N	.070
	else	±1						
	2	2	$4\Psi_{2Np}$		8 N	4 N	2 N	.082
	-2	-2						
	else	else	$4\Psi_{Np}$	$4\Psi_{GS}$	10 N	8 N	7 N	.103
SROT3	1	1	$3\Psi_{Np}$		3 N	$1\frac{1}{2}$ N	$\frac{3}{4}$ N	.032
	-1	-1						
	1	-1	$3\Psi_{Np}$	$2\Psi_{REV}$	5 N	$2\frac{1}{2}$ N	$1\frac{3}{4}$ N	.052
	-1	1						
	±1	else	$3\Psi_{Np}$	$2\Psi_{GS}$	6 N	$4\frac{1}{2}$ N	$3\frac{3}{4}$ N	.060
	else	±1						
	2	2	$3\Psi_{2Np}$		6 N	3 N	$1\frac{1}{2}$ N	.062
	-2	-2						
	3	3	$3\Psi_{3Np}$		9 N	$4\frac{1}{2}$ N	$2\frac{1}{4}$ N	.092
	-3	-3						
	else	else	$3\Psi_{Np}$	$4\Psi_{GS}$	9 N	$7\frac{1}{2}$ N	$6\frac{3}{4}$ N	.093

TABLE A.1 Execution times for REAL BLAS routines, Part II.

subprogram	INCX	INCY	execution times for the vector instructions	execution times for the data movements	total execution times 1-pipe	total execution times 2-pipe	total execution times 4-pipe	measured times on 1-pipe * 100
ICAMAX	1		$\Psi_{3Np}+\Psi_N$		4 N	$2\frac{1}{2}$ N	$1\frac{3}{4}$ N	.042
	else		$\Psi_{Np}+\Psi_N$	$2\Psi_{GS}$	5 N	$4\frac{1}{2}$ N	$4\frac{1}{4}$ N	.052
SCASUM	1		$\Psi_{Np}+\Psi_N$		2 N	$1\frac{1}{2}$ N	$1\frac{1}{4}$ N	.021
	else		$\Psi_{Np}+\Psi_N$	$2\Psi_{GS}$	5 N	$4\frac{1}{2}$ N	$4\frac{1}{4}$ N	.052
SCNRM2	1		Ψ_{2N}		2 N	2 N	2 N	.021
	else		Ψ_{2N}	$2\Psi_{GS}$	5 N	5 N	5 N	.052
CDOTU	1	1	$4\Psi_{2N}$		8 N	8 N	8 N	.082
	-1	-1						
	1	-1	$3\Psi_{2N}$	Ψ_{2REV}	8 N	7 N	7 N	.082
	-1	1						
	else	else	$\Psi_{2N}+2\Psi_N$	$4\Psi_{GS}$	10 N	10 N	10 N	.099
CDOTC	1	1	$3\Psi_{2N}$		6 N	6 N	6 N	.062
	-1	-1						
	1	-1	$3\Psi_{2N}$	$2\Psi_{2REV}$	10 N	8 N	8 N	.102
	-1	1						
	else	else	$\Psi_{2N}+2\Psi_N$	$4\Psi_{GS}$	10 N	10 N	10 N	.099
CAXPY	1	1	$3\Psi_{2Np}$		6 N	3 N	$1\frac{1}{2}$ N	.062
	-1	-1						
	1	-1	$3\Psi_{2Np}$	$2\Psi_{2REV}$	10 N	5 N	$3\frac{1}{2}$ N	.102
	-1	1						
	else	±1	$3\Psi_{2Np}$	$2\Psi_{GS}+\Psi_{2Np}$	11 N	7 N	5 N	.111
	else	else	$\Psi_{2Np}+2\Psi_{Np}$	$6\Psi_{GS}$	13 N	11 N	10 N	.129
CSCAL	1		$3\Psi_{2Np}$		6 N	3 N	$1\frac{1}{2}$ N	.062
	else		$\Psi_{2Np}+2\Psi_{Np}$	$4\Psi_{GS}$	10 N	8 N	7 N	.106
CSSCAL	1		Ψ_{2Np}		2 N	N	$\frac{1}{2}$ N	.021
	else		Ψ_{2Np}	$4\Psi_{GS}$	8 N	7 N	$6\frac{1}{2}$ N	.086
CSWAP	1	1		$3\Psi_{2Np}$	6 N	3 N	$1\frac{1}{2}$ N	.062
	-1	-1						
	1	-1		$\Psi_{2Np}+4\Psi_{2REV}$	10 N	5 N	$4\frac{1}{2}$ N	.102
	-1	1						
	±1	else		$6\Psi_{GS}+\Psi_{2Np}$	11 N	10 N	$9\frac{1}{2}$ N	.113
	else	±1						
	else	else		$8\Psi_{GS}$	12 N	12 N	12 N	.127

TABLE A.2 Execution times for COMPLEX BLAS routines, Part I.

subprogram	INCX	INCY	execution times for the vector instructions	execution times for the data movements	total execution times 1-pipe	total execution times 2-pipe	total execution times 4-pipe	measured times on 1-pipe * 100
CCOPY	1	1		Ψ_{2Np}	2 N	N	$\frac{1}{2}$ N	.021
	-1	-1						
	1	-1		$2\Psi_{2REV}$	4 N	2 N	2 N	.041
	-1	1						
	else	± 1		$2\Psi_{GS}+\Psi_{2Np}$	5 N	4 N	$3\frac{1}{2}$ N	.050
	else	else		$4\Psi_{GS}$	6 N	6 N	6 N	.065
CSROT4	1	1	$4\Psi_{2Np}$		8 N	4 N	2 N	.082
	-1	-1						
	1	-1	$4\Psi_{2Np}$	$4\Psi_{2REV}$	16 N	8 N	6 N	.165
	-1	1						
	± 1	else	$4\Psi_{2Np}$	$6\Psi_{GS}+\Psi_{2Np}$	19 N	14 N	$11\frac{1}{2}$ N	.196
	else	± 1						
	else	else	$4\Psi_{2Np}$	$8\Psi_{GS}$	20 N	16 N	14 N	.210
CSROT3	1	1	$3\Psi_{2Np}$		6 N	3 N	$1\frac{1}{2}$ N	.062
	-1	-1						
	1	-1	$3\Psi_{2Np}$	$4\Psi_{2REV}$	14 N	7 N	$5\frac{1}{2}$N	.142
	-1	1						
	± 1	else	$3\Psi_{2Np}$	$6\Psi_{GS}+\Psi_{2Np}$	17 N	13 N	11 N	.172
	else	± 1						
	else	else	$3\Psi_{2Np}$	$8\Psi_{GS}$	18 N	15 N	$13\frac{1}{2}$ N	.185

TABLE A.2 Execution times for COMPLEX BLAS routines, Part II.