M. Louter-Nool

BLAS on the Cyber 205

Department of Numerical Mathematics Report NM-R8524 December

BLAS on the Cyber 205

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

The subject of this paper is to examine the efficiency of a set of linear algebra subprograms, the so-called BLAS, as implemented on a 1-pipe Cyber 205. Beside this implementation, we have developed an alternative version of the subprograms. The performance of both implementations is compared. Special attention is paid to operations on nonsequentially stored vector elements, leading to so-called stride problems. Several routines, appropriate to deal with those stride problems, are treated. In this paper, we shall only consider the single precision REAL and COMPLEX BLAS subprograms, with positive strides.

1980 Mathematics subject classification: Primary:65V05. Secondary:65FXX
Key Words & Phrases: vectorization, basic linear algebra subprograms, stride problems, operations on sequentially and nonsequentially stored REAL and COMPLEX vectors.

Note: This paper is submitted for publication

1. INTRODUCTION

The BLAS (Basic Linear Algebra Subprograms) is a set of basic routines for problems in linear algebra. The complete set as described in Lawson et al.[7] includes the operations of dot products, vector plus a scalar times a vector, copy, swap, Euclidian norm, sum of magnitudes, multiplying a scalar times a vector, locating an element of largest magnitude, the Givens transformation and the modified Givens transformation. The collection of routines is divided into four separate parts, COMPLEX, REAL, DOUBLE PRECISION and COMPLEX*16. The BLAS has been very successful and is used in a wide range of software, including LINPACK[4] , EISPACK[9] and is inserted in the IMSL[1] . The BLAS has been implemented on a number of scalar machines, and since all subprograms are performing operations on vectors, excluding the Givens transformations, the BLAS could be implemented on vector computers. On the Cyber 205, as located at SARA, Amsterdam, The Netherlands, the single precision REAL and COMPLEX BLAS routines are installed by CDC. In this paper, we shall refer to this implementation as the CDC BLAS.

The goal of this paper is to examine the efficiency of the CDC BLAS. For this purpose, we have written our own alternative version in FORTRAN 200, the CWI BLAS. CWI is an abbreviation of "Centrum voor Wiskunde en Informatica", the Dutch name for "Centre for Mathematics and Computer Science". We have time measured both implementations for large values of n, i.e. the number of elements in an array on which actually an operation is performed. Another approach of testing the efficiency of the CDC BLAS could be to determine for which values of n scalar optimization is more profitable than vectorization, but as yet no investigations have been done with respect to this approach.

The Cyber 205 is actually designed to operate on sequential memory locations. However, the BLAS permits a constant space, or stride, between the vector elements, described by an increment parameter. We have paid much attention to the performance of the subprograms, when a stride problem is involved. Though the BLAS also permits a negative stride, the CDC BLAS only allows positive values. For this reason, we have restricted ourselves to positive increment values. If the reader is

Report NM-R8524
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

interested in our implementation, please contact the author of this paper. All source texts are available and provided by (as yet, only in Dutch) comments on many pieces of code.

In Dongarra et al.[5] an extended set of BLAS at the level of matrix-vector operations is proposed. They state that, on vector machines, one needs to optimize at this level in order to approach the potential efficiency. At the CWI, many of the proposed routines of the Extended BLAS have already been developed. The routines are inserted in the NUMVEC library [8] .

In section 2 we shall specify the BLAS subprograms, their headings and parameter lists. The timing conventions are described in section 3. A distinction has been made by us between the REAL BLAS and the COMPLEX BLAS. For the REAL BLAS, the execution times and suggestions for more efficient implementations can be found in section 4. Especially, when a stride problem is involved, the CDC implementation of the COMPLEX BLAS can be substantially improved, as is shown in section 5.

Finally, we remark, that in this paper, one will meet two kinds of letter types for variables, the SMALL CAPITALS, used in FORTRAN context, and the *italics*, used in *text* environment. E.g., there is no difference in meaning between N and *n*, IX and *ix*, IY and *iy* . Furthermore, the vectors SX(REAL) and CX(COMPLEX) are often shortly denoted by *x*, and similarly, SY and CY by *y*. Moreover, all FORTRAN 'reserved words' and all subprogram names are written in SMALL CAPITALS too.

2. Specification of the BLAS subprograms

In section 2.1 some remarks on the increment parameters are made. They play an important role in our time measurements. Only positive increment values are considered in this paper. Next, we give information about the type and dimensions of the variables (section 2.2) and the type of the functions (section 2.3). Section 2.4 lists all of the 20 subprogram names and their parameter lists, and defines the operations performed by each subprogram.

2.1. Increment parameter or stride

All subprograms permits a constant spacing between the vector elements. This spacing, or **stride** is specified by an increment parameter, e.g. *ix*. This means, only those elements with storage location

$$1 + (i-1) * ix \qquad \text{for } i = 1, \ldots, n \text{ and } ix>0 \qquad (2.1.1)$$

are being used in the subprogram. All elements with *another* storage location are left *unused.*

2.2. Type and dimension information for variables

The variables of BLAS have the following type :

INTEGER	NX, NY, N, IX, IY, IMAX
REAL	SX(NX), SY(NY), SW, SA, SC, SS
COMPLEX	CX(NX), CY(NY), CW, CA

The variables *ix* and *iy* are the increment values of the arrays SX, CX and SY, CY respectively . The variable *n* denotes the number of elements of the array, on which actually an operation is performed. Note that the dimension sizes *nx* and *ny* must satisfy

$$\begin{aligned} nx &\geqslant (n-1)*ix + 1 \\ ny &\geqslant (n-1)*iy + 1 . \end{aligned} \qquad (2.2.1)$$

2.3. Type declarations for function names

The function names are of the following type :

INTEGER	ISAMAX, ICAMAX
REAL	SASUM, SCASUM, SNRM2, SCNRM2, SDOT
COMPLEX	CDOTC, CDOTU

2.4. The subprograms of BLAS and their specification

This section contains the subprogram names and their parameter lists and specifies what the routines actually do. For simplicity, we assume here, that the increment parameters *ix* and *iy* are unity.

1 find the largest component of a vector

(a) IMAX = ISAMAX (N, SX, IX)

(b) IMAX = ICAMAX (N, CX, IX)

The function ISAMAX determines the smallest integer i such that

$$|x_i| = \max \left\{ |x_j| : j = 1,...,n \right\}$$

and ICAMAX determines the smallest integer i such that

$$|x_i| = \max \left\{ |\mathrm{Re}\, x_j| + |\mathrm{Im}\, x_j| : j = 1,...,n \right\}$$

For argumentation of this definition of ICAMAX see[7] .

2 Euclidian length or l_2 norm of a vector

(a) SW = SNRM2 (N, SX, IX)

(b) SW = SCNRM2 (N, CX, IX)

Both functions calculate

$$w := \left[\sum_{i=1}^{n} |x_i|^2 \right]^{½} .$$

3 sum of the magnitudes of vector components

(a) SW = SASUM (N, SX, IX)

(b) SW = SCASUM (N, CX, IX)

The function SASUM computes

$$w := \sum_{i=1}^{n} |x_i|$$

and the function SCASUM computes

$$w := \sum_{i=1}^{n} \left\{ |\mathrm{Re}\, x_i| + |\mathrm{Im}\, x_i| \right\}$$

For argumentation of this specific definition of SCASUM, we refer to[7] .

4 dot products

(a) SW = SDOT (N, SX, IX, SY, IY)

(b) CW = CDOTU (N, CX, IX, CY, IY)

(c) CW = CDOTC (N, CX, IX, CY, IY)

SDOT and CDOTU calculate the following dot product

$$w := \sum_{i=1}^{n} x_i \cdot y_i$$

The suffix C on CDOTC indicates, that the vector components x_i are used conjugated, so CDOTC calculates

$$w := \sum_{i=1}^{n} \bar{x}_i \cdot y_i$$

5 elementary vector operation (a = scalar) : $y := a \,.\, x + y$

(a) CALL SAXPY (N, SA, SX, IX, SY, IY)

(b) CALL CAXPY (N, CA, CX, IX, CY, IY)

6 vector scaling (a = scalar) : $x := a \,.\, x$

(a) CALL SSCAL (N, SA, SX, IX)

(b) CALL CSCAL (N, CA, CX, IX)

(c) CALL CSSCAL (N, SA, CX, IX)

7 copy a vector x to y : $y := x$

(a) CALL SCOPY (N, SX, IX, SY, IY)

(b) CALL CCOPY (N, CX, IX, CY, IY)

8 interchange vectors x and y : $x :=: y$

(a) CALL SSWAP (N, SX, IX, SY, IY)

(b) CALL CSWAP (N, CX, IX, CY, IY)

9 apply a plane rotation

(a) CALL SROT (N, SX, IX, SY, IY, SC, SS)

(b) CALL CSROT (N, CX, IX, CY, IY, SC, SS)

These subroutines compute

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} := \begin{bmatrix} sc & ss \\ -ss & sc \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad \text{for } i = 1, \ldots, n.$$

3. Timing conventions

Timings of the BLAS are given here[1] for increment values 1 (no stride problem) and 2. All vectors in our testing program contain 20,000 REAL components, which results in

$$\begin{aligned} nx = ny &= 20{,}000 \qquad \text{for REAL vectors} \\ nx = ny &= 10{,}000 \qquad \text{for COMPLEX vectors .} \end{aligned} \tag{3.1}$$

1. In Appendix I, results for all combinations of strides 1 and 10 are given with an adjustable value of n

The timings of cases with a stride problem in one or both vectors are compared with the corresponding no stride case $ix(=iy)=1$ and the *same* value of n, i.e. the number of elements of the vectors on which actually the operation is performed. The value of n is chosen as large as possible such that the vectors still fit into the maximum allowed lengths in case of a stride problem. This implies

$$n = \frac{nx}{2} \qquad \text{for } ix,\ iy \in \left\{ 1,\ 2 \right\} \tag{3.2}$$

Furthermore, we define

$$\alpha := \text{number of calls of a subprogram.} \tag{3.3}$$

For all our time measuring, we took $\alpha = 10{,}000$, in order to obtain accurate results.

As an example, we now determine the theoretical time, needed for one single vector operation, i.e. a vector operation which is performed in one clock cycle per REAL vector element. The LINKED TRIAD, an operation involving 2 vectors, 1 scalar and 2 operators (one floating point multiply and a floating point add or subtract operator with a vector result) is considered as a single vector operation. In case of optimal vectorization, the theoretical time Ψ_m, needed for one REAL vector operation with m elements (stored in adjacent storage locations) and ignoring the startup times, would be at least :

$$\Psi_m = m \,.\, \alpha \,.\, c \text{ CPU seconds} \tag{3.4}$$

where c is the cycle time (20 nsec. for the Cyber 205).

Obviously, $m = n$ for the REAL case, and $m = 2n$ for the COMPLEX case. In particular, we shall work with one value of m, viz, $m = 10{,}000$ (see formulas (3.1) and (3.2)), for which

$$\Psi_m = 2. \text{ CPU seconds} \tag{3.5}$$

The BLAS is said to be a fast code. In our version, no restrictions with respect to memory locations have been made. When CPU-time could be reduced, temporary vectors are used, created by means of the ASSIGN-statement at the dynamic stack.

We have timed both the CDC BLAS implementation as well as our alternative version, the so-called CWI BLAS implementation. Our version is not a set of ready-made routines, but a piece of FORTRAN 200 code performing the operations as described in section 2.4. In the same main program both the CDC BLAS subprograms and the CWI FORTRAN 200 code are called α times. Outside these DO-loops, we measured the CPU times by SECOND. Although the CPU times of our code was quite good reproducible, this was not true for the CDC implementation, the execution times were depending on the length of the main program. In Van de Vorst[10] this phenomenon was already signalled. Here, we always list the minimally measured CPU time. Since we are only interested in the number of vector operations of the CDC BLAS, and not in small modifications, we have not further investigated this problem.

In several tables in this paper, percentages are listed; they have been calculated during execution and indicate the timing results of a CWI BLAS version with regard to the CDC BLAS subprogram.

All testing, during the first part of 1985, is executed on the 1-pipe Cyber 205, located at SARA, Amsterdam, the Netherlands. This vector computer has an 8 bank memory with a VSOS RSYS (cycles 607 and 631) operating system. All programs were compiled with FTN200 (Fortran compiler FORTRAN 200, cycles 607 and 631), and OPT=V (i.e., automatic vectorization, wherever possible) and the vector elements were stored on large pages of the virtual memory. We asked for 5 LP, but, unfortunately it is not possible for the user to find out what one really obtains ! However, in practice, we never had more than 3 LP FAULTS.

4. The real BLAS

The implementation and testing of these low-level operations needs some explanation, in particular when the elements are stored nonsequentially. Section 4.1 contains the timing of the REAL CDC BLAS routines without a stride problem. Besides, we shall describe how the execution times of some subprograms can be improved. In section 4.2 we shall treat some tools for vector operations with non-sequentially stored elements. The timing results of both the CDC BLAS and the CWI BLAS with and without stride problems in one (or both) vector(s) can be found in section 4.3. This chapter ends with some conclusions concerning the REAL BLAS.

4.1. Timing results of the real BLAS with increments equal to 1

To start with, we shall give the execution times of the CDC BLAS for $n = 10,000$.

real BLAS, $n = 10,000$ $ix(=iy)=1$	CDC
ISAMAX	2.0
SASUM	4.1
SNRM2	2.1
SDOT	2.1
SAXPY	2.0
SSCAL	2.0
SSWAP	6.1
SCOPY	2.0
SROT	12.1

TABLE 4.1.1 REAL BLAS, stride = 1

As mentioned before, one vector operation will take at least $\Psi_m = 2.0$ CPU seconds (cf. formula (3.5)). Clearly, all subprograms perform just one vector operation, except for SASUM, SSWAP and SROT. SSWAP needs three copy operations.

The execution times of the CDC SASUM and SROT can be decreased as follows :

SASUM

Let us assume n is even. An addition with *sign control* (i.e. take the absolute value of each input element) of the first $\frac{n}{2}$ and second $\frac{n}{2}$ elements of the vector x results in

$$xx_i = | x_i | + | x_{i+\frac{n}{2}} | \tag{4.1.1}$$

for $i = 1, \ldots, \frac{n}{2}$. This vector operation can be performed by means of the SPECIAL CALL routine Q8ADDNV (with X'05' being the value of the G-descriptor, to obtain the sign control addition). For more details on this routine, see the Hardware Manual[2] . Finally, a call of the intrinsic vector function Q8SSUM (see the FORTRAN 200 ref. manual[3]) on the vector xx, calculating the sum of all elements, delivers the required result.

In case of n being odd, the absolute value of the n-th element must be added. We remark that each operation is performed on $\frac{n}{2}$ elements, so the theoretical execution time of this implementation is $2 * \frac{1}{2} \Psi_m = \Psi_m$ CPU sec..

SROT

We notice that CDC SROT needs 6 vector operations. However, the same *numerical* result may be obtained in only 4 operations. This solution requires 2 temporary vectors, say v and w, containing

$$\begin{aligned} v_i &= sc * x_i \\ w_i &= ss * x_i \end{aligned} \tag{4.1.2'}$$

for $i=1, \ldots, n$. The computation can be accomplished by performing 2 LINKED TRIADS

$$\begin{aligned} x_i &= v_i + ss * y_i \\ y_i &= sc * y_i - w_i \end{aligned} \tag{4.1.2''}$$

for $i=1, \ldots, n$.

There is even a possibility to perform the SROT operation in 3 vector operations, with the introduction of only 1 temporary vector, say z, where z becomes

$$z_i = ss * (x_i + y_i) \tag{4.1.3'}$$

for $i=1, \ldots, n$. The following 2 LINKED TRIADS

$$\begin{aligned} x_i &= (sc - ss) x_i + z_i \\ y_i &= (sc + ss) y_i - z_i \end{aligned} \tag{4.1.3''}$$

for $i=1, \ldots, n$, deliver the desired result. We shall refer to this solution as SROT3 and to the 4 vector operation solution as SROT4. Though the numerical results may slightly differ from the 4 vector operation solution, in case of $sc \ll ss$ or $sc \gg ss$, there is a region in which this solution delivers more reliable numerical values.

The gain for these subprograms roughly amounts for SASUM 50% and for SROT4 67%, SROT3 50%, respectively.

4.2. Some tools for vector operations with increments unequal to one

As mentioned already in section 2.1, the Cyber 205 is actually designed to operate on sequential memory locations. There are two machine instructions, the so-called GATHER and SCATTER, available for the purpose of moving data elements from nonsequential to sequential locations (GATHER) and the reverse (SCATTER). Another possibility to operate on nonsequentially stored vector elements is to use control or BIT vectors, e.g. the usage of the WHERE statement.

4.2.1. The GATHER and SCATTER operations. Since the BLAS subprograms permits a *constant* stride, one can use the *periodic* GATHER and SCATTER. We have used the vector intrinsic functions Q8VGATHP (GATHER) and Q8VSCATP (SCATTER). The theoretical timings, from the Cyber 205 user's guide[6] , of the GATHER and SCATTER operations are as follows

$$\begin{aligned} &\text{GATHER} &&: \quad 39 + \tfrac{5}{4}\, n \ \text{ clock cycles} \\ &\text{SCATTER} &&: \quad 71 + \tfrac{5}{4}\, n \ \text{ clock cycles,} \end{aligned} \tag{4.2.1.1}$$

where n denotes the number of data elements to be moved. The first number in these timing formulas denotes the startup time. In Table 4.2.1.1 we give the execution times under the timing conventions of section 3. In all cases the n elements are scattered over 20,000 adjacent storage locations.

These machine instructions turned out to be very useful, because their timings are independent of the length of the vector from which the elements are gathered or to which they are scattered (as opposed to the control vector construction described below).

increment	2	4	5	8	10
n	10000	5000	4000	2500	2000
GATHER	2.9	1.5	1.2	.9	.7
SCATTER	2.9	1.7	1.4	.9	.6
theoretical	2.5	1.25	1.0	.625	.5

TABLE 4.2.1.1

4.2.2. The control vector solution. The WHERE statement controls operations by means of a BIT vector or control vector. Actually, an operation is performed on the whole vector and a result is stored, only if the corresponding bit is set. This implies, that the execution time of an operation guided by a control vector depends on the time needed to perform the operation on all elements. As an example, consider Table 4.2.1.1, a BIT vector of length 20,000 is needed for all increment values, which will result in an execution time of at least 4.0 CPU seconds for one single vector operation on n elements, where n takes the values as described in Table 4.2.1.1. As a consequence, the control vector solution can be useful, only if the increment parameter is small. Moreover, the WHERE construction can only be used for subprograms with one increment parameter, and, in cases where both parameters are equal.

4.3. Timing results of the real BLAS

We have timed the REAL CDC BLAS for $n = 10,000$ with the four possible combinations of increment values 1 and 2 for ix and iy. Table 4.3.1 gives the relevant execution times. The open places follow trivially. As an example, for SDOT the execution time for $ix = 2$ and $iy = 1$ equals that for $ix = 1$ and $iy = 2$.

real BLAS, $n = 10,000$				
	$ix=1$ $(iy=1)$	$ix=2$ $(iy=2)$	$ix=1$ $iy=2$	$ix=2$ $iy=1$
ISAMAX	2.0	5.1	-	-
SASUM	4.1	7.1	-	-
SNRM2	2.1	5.1	-	-
SDOT	2.1	8.1	5.1	-
SAXPY	2.0	11.0	7.9	5.1
SSCAL	2.0	8.0	-	-
SSWAP	6.1	11.9	7.9	-
SCOPY	2.0	5.9	2.9	3.0
SROT	12.1	20.1	16.0	-

TABLE 4.3.1 CDC version

The results of the first column are those of Table 4.1.1. The execution times of the other columns clearly display the substantial effort necessary to eliminate the stride problems. We note, that n is left unchanged for different increment values. The difference in execution times of column 1 as compared with the others are exclusively due to the nonsequential storage of the vector elements.

From the timing results of Table 4.2.1.1, it appears, that the execution time of one GATHER operation and also of one SCATTER operation, for $n = 10,000$ is about 3. CPU seconds. Obviously, subtracting the results of column 1 from the others, a multiple of approximately 3. CPU seconds is obtained. This suggests, that for increments unequal to one, the implementors of CDC BLAS utilise the GATHER and SCATTER instructions and not e.g., the WHERE construction.

Although the machine instructions GATHER/SCATTER are very fast, they do not always yield the most efficient code for small increment values. We shall illustrate this by means of an example with subprogram SAXPY.

EXAMPLE:

Consider SAXPY with $ix = iy = k > 1$.

GATHER/SCATTER :

Both x and y are stored nonsequentially; this implies that both vectors are to be gathered into temporary vectors with adjacent memory locations. Next, the operation can be performed on the temporary vectors and finally, the result vector is scattered into y. This whole operation on n elements takes

$$\left.\begin{array}{ll} \text{GATHER } x & \frac{5}{4}n \\ \text{GATHER } y & \frac{5}{4}n \\ \text{LINKED TRIAD} & n \\ \text{SCATTER } y & \frac{5}{4}n \end{array}\right\} 4\frac{3}{4}\,n \text{ clock cycles,}$$

ignoring startup times for n large.

WHERE :

Since ix equals iy, one can use the WHERE construction with a control vector to perform the LINKED TRIAD. The elements are scattered here over $ix * n$ memory locations, so the LINKED TRIAD is performed in

LINKED TRIAD $ix * n$ clock cycles,

ignoring the time needed to create a control vector. Obviously, creating such control vectors takes time too, except when this is done during compilation time, by means of a data statement. Therefore we have created some bit vectors of size 65535, the maximum length of a vector on a large page.

Performing SAXPY with a control vector is more efficient than with GATHER/SCATTER up to an increment of 4 (in practice, this value would be 5, cf the differences in the theoretical times and measured times for the GATHER and SCATTER operations as listed in Table 4.2.1.1). The total amount of work, ignoring the time needed to create a control vector, is reduced to 37 %, compared with the CDC BLAS implementation.

Analogously, one can determine for all REAL BLAS routines the maximum value k for which it is still profitable to use control vectors instead of the GATHER/SCATTER operations. The routines with the corresponding values of k are listed in Table 4.3.2.

REMARK : On a 2-pipe or 4-pipe Cyber 205, the gain of using control vectors for small increment values would have been much greater, since the execution times of operations guided by control vectors will decrease with a factor 2 or 4, respectively, and those of the GATHER / SCATTER operations will remain the same.

SDOT	$k \leqslant 4$
SAXPY	$k \leqslant 5$
SSCAL	$k \leqslant 4$
SCOPY	$k \leqslant 3$
SROT	$k \leqslant 2$

TABLE 4.3.2

The next table shows the execution times for various REAL BLAS subprograms with $n = 10{,}000$ and $ix(=iy)=2$, with both the GATHER/SCATTER (CDC BLAS) solution and the control vector solution (CWI BLAS). The last column gives an indication of the reduction in execution time using control vectors instead of the GATHER/SCATTER implementation. The percentages are calculated during execution.

real BLAS $n = 10{,}000$, $ix(=iy)=2$			
	CDC	CWI	%
SDOT	8.1	4.1	50
SAXPY	11.0	4.1	37
SSCAL	8.0	4.0	50
SSWAP	11.9	12.1	101
SCOPY	5.9	4.1	68
SROT	20.3	16.7	83

TABLE 4.3.3 REAL BLAS, stride = 2

4.4. Conclusions

The REAL BLAS subprograms without a stride problem, as implemented on the one-pipe Cyber 205, are all optimal for n large, (i.e. $n = 2{,}000$ or $n = 10{,}000$), except for the subprograms SASUM and SROT. The measured CPU times for our implementation of SASUM, SROT3 and SROT4 (3 respectively 4 vector operations) are

real BLAS, $n = 10{,}000$			
	$ix=1$ $(iy=1)$	$ix=2$ $(iy=2)$	$ix=1$ $iy=2$
SASUM	2.1	5.1	-
SROT4	8.2	16.7	14.4
SROT3	6.1	12.4	12.1

TABLE 4.4.1 CWI implementation

where the results of SROT4 and SROT3 with $ix = iy = 2$ are obtained by means of the WHERE construction. For all other REAL BLAS subprograms the execution times of our implementation are similar to those of Table 4.3.1.

To emphasize that the GATHER/SCATTER solution only depends on the value of n, and not on the size of the increment values, we also have timed for all combinations of increment values 1 and 10, and $n = 2{,}000$. The results can be found in Appendix I, Tables A1.1 and A1.2.

When the increments are not unity, the execution times can be decreased for small increment values. It is worthwhile to use control vectors, created at compilation time, in those subprograms, which need more time to gather and to scatter the elements, than performing the required operation.

5. THE COMPLEX BLAS

The implementation of the COMPLEX BLAS is more complicated, as for of the REAL BLAS. First, let us consider the storage of a COMPLEX vector. A COMPLEX vector is stored in the following way

$$R_1 \, I_1 \, R_2 \, I_2 \cdots \tag{5.1}$$

where R_i and I_i represent the real and imaginary part respectively of the i-th COMPLEX element. This way of storage implies, that a contiguous operation on the real parts results in a stride problem with stride 2. In section 5.1 we shall describe some subroutines to perform such a separation of COMPLEX vectors, together with their execution times.

For a definition of the subprograms of the COMPLEX BLAS we refer to section 2.4. The execution times of the COMPLEX CDC BLAS with increment value 1 and $n = 5{,}000$ are listed in section 5.2. In section 5.3 a description is given of our implementation of the COMPLEX BLAS operations.

Much attention is paid to the COMPLEX BLAS with increments greater than one. In section 5.4, we shall first propose some tools for handling stride problems in COMPLEX vectors. And in section 5.5, we shall demonstrate how we have implemented these subprograms, assuming that, in case of two increments, both are greater than 1. In section 5.6, we shall consider subprograms with two increment parameter one of which is equal to unity. Finally, in section 5.7 some conclusions are made.

5.1. Splitting of a complex vector into a real and an imaginary part

Working with COMPLEX vectors often makes it necessary to separate the real from the imaginary components. For vectors, just like for scalars, there exist intrinsic functions for this purpose, the so-called V-functions. To obtain the real and imaginary components, the functions VREAL and VAIMAG are available. Table 5.1.1 gives their execution times for $n = 5{,}000$ and $\alpha = 10{,}000$ (see section 3 for our timing conventions).

Since in a COMPLEX vector the real and imaginary components alternate, an EQUIVALENCE of a COMPLEX vector with a REAL vector delivers a vector, containing the real and imaginary components each with a stride of 2. As a consequence, one can use the periodic GATHER to gather the real/imaginary components into temporary vectors as follows :

```
PARAMETERS (N=5000, N2=2*N)
REAL      RCX (N2)
COMPLEX CX (N )
EQUIVALENCE (CX, RCX)
DESCRIPTOR D1, D2

ASSIGN D1, .DYN.N
ASSIGN D2, .DYN.N
D1 = Q8VGATHP (RCX(1;N2-1), 2, N; D1)
D2 = Q8VGATHP (RCX(2;N2-1), 2, N; D2)
```

Note, that in the second call of Q8VGATHP we start with the second element of RCX, or the first imaginary component. In the FORTRAN 200 manual[3] one can find a description of the vector intrinsic function Q8VGATHP. After compilation and execution of this part of FORTRAN 200 code, the DESCRIPTORS D1 and D2 will refer to vectors containing the real and imaginary components, respectively.

As can be seen in Table 5.1.1, the execution time of the periodic GATHER is about 1.5 CPU seconds, considerably faster than the VREAL and VAIMAG timing results. Apart from this, we were very surprised to find different timings of VREAL and VAIMAG on the Cyber 205. Moreover, both intrinsic functions appear to be too *slow* compared with the periodic GATHER.

$n = 5{,}000,\ \alpha = 10{,}000$	
VREAL	4.
VAIMAG	2.
GATHER (per.)	1.5

TABLE 5.1.1

In case of an increment value of 1, it is not necessary to separate the real and imaginary components. This will be illustrated in section 5.3.

5.2. Timing results of the complex BLAS without additional stride problems

In Table 5.2.1, we give the execution times of the COMPLEX CDC BLAS for $n = 5{,}000$, and $\alpha = 10{,}000$, i.e. the number of times a subprogram is called. Note, that there are 5,000 COMPLEX elements, so that each operation is performed on $m = 10{,}000$ memory locations.

complex BLAS, $n = 5{,}000$ $ix(=iy)=1$	
	CDC
ICAMAX	4.1
SCASUM	4.1
SCNRM2	2.1
CDOTU	8.2
CDOTC	6.1
CAXPY	6.3
CSCAL	6.1
CSSCAL	2.0
CSWAP	6.1
CCOPY	2.0
CSROT	12.1

TABLE 5.2.1 CDC version, no stride problem

We notice, that each subprogram roughly takes a multiple of the theoretical time of $\Psi_m = 2.$ CPU seconds (cf. formula (3.5)), from which the number of vector operations performed for each subprogram can be easily derived. As for our implementation, we shall show in the next section, how we have achieved or improved these times.

5.3. Our implementation of the complex BLAS without additional stride problems

In this section we shall demonstrate how we have vectorized the various COMPLEX BLAS operations. As mentioned in section 5.1, it is often quite convenient to use a vector of type REAL equivalenced with a COMPLEX vector. Such vectors enable us to operate on REAL vectors. We shall use the vectors *rx* and *ry*, where

$$\text{REAL } rx(1;2n) \approx \text{COMPLEX } x(1;n) \qquad (5.3.1')$$

and

$$\text{REAL } ry(1;2n) \approx \text{COMPLEX } y(1;n) \tag{5.3.1”}$$

and ≈ represents an EQUIVALENCE statement.

It is not our intention to give extended specifications on our implementation. The aim of this description is to show, which operations on the REAL vectors *rx* and *ry* actually are to be performed. The theoretical times, according to the timing conventions listed in section 3, are added.

ICAMAX

First of all we have to compute a temporary vector z of size n as follows:

$$z_i = |\text{Re}\, x_i| + |\text{Im}\, x_i| \quad \text{for } i = 1,...,n. \tag{5.3.2}$$

There are several methods to compute such a vector all having the same speed. To show the nice possibilities of so-called sparse vector instructions, we shall discuss one of these instructions. According to the hardware manual[2] , a sparse vector consists of a vector pair, one of which is a bit string, identified as an order vector, and the other is a floating-point array identified as the data vector. Sparse order vectors determine the positional significance of the segments of the corresponding sparse data vector.

To compute the vector z we used the sparse vector addition Q8ADDNV (see[2]) with the following vectors

$$\begin{array}{ccccccccccc}
.\,. & 1 & 1 & 0 & 1 & 1 & 0 & 1 & .\,. & (1) \\
.\,. & R_{i-1} & I_{i-1} & & R_i & I_i & & R_{i+1} & .\,. & (2) \\
.\,. & I_{i-1} & & R_i & I_i & & R_{i+1} & I_{i+1} & .\,. & (3) \\
.\,. & 1 & 0 & 1 & 1 & 0 & 1 & 1 & .\,. & (4) \\
\hline
 & & & & & & & & & + \\
.\,. & (|R_{i-1}|+|I_{i-1}|) & & & (|R_i|+|I_i|) & & & (|R_{i+1}|+|I_{i+1}|) & .\,. & (5) \\
.\,. & 1 & 0 & 0 & 1 & 0 & 0 & 1 & .\,. & (6)
\end{array}$$

We add by means of the logical .AND. operation on the BIT vectors (1) and (4), which results in the sparse order vector (6). The BIT vectors (1) and (4) are the same vectors, only with a different start address, and can be created by a data statement. The REAL vector (2) contains $rx(1;2n-1)$ and REAL vector (3) contains $rx(2;2n-1)$. As an important side effect the sparse vector instructions have sign control (i.e. the absolute value of all input data elements is taken). The resulting sparse data vector (5) contains just the desired values. GATHER operations or operations with control vectors are superfluous for further computations.

The length of the BIT or sparse order vectors determines the timing, so this instruction takes $1\frac{1}{2}$ * the time of a non-sparse addition, i.e. $1\frac{1}{2} * \Psi_m = 3.$ CPU sec., but, with profit of the desired storage of the result vector.

Finally, a call of Q8SMAXI (see[3]) will be sufficient to obtain the smallest index, such that

$$z_i = \max \{ z_j : j = 1, \ldots, n \}.$$

Notice that in this case all elements of z are positive. n being equal to $\frac{1}{2}\, m$, we observe that this operation is performed in $\frac{1}{2}\, \Psi_m$ CPU seconds. Summarizing, the theoretical time of this implementation of ICAMAX is 4. sec.

SCASUM

This subprogram illustrates why splitting of real and imaginary components may be superfluous. In analogy to the REAL BLAS routine SASUM, we start with a vector addition (with sign control) of the first n elements of the REAL vector rx and its last n elements (rx is of length $2n$, cf formula (5.3.1')). Next, we calculate the sum of the elements of the temporary vector ($2^*\frac{1}{2}\Psi_m = 2.$ sec.).

SCNRM2

The implementation of this routine is quite simple. The dot product of $rx \,.\, rx$ (2. sec.) followed by the square root on the result will satisfy.

CDOTU/CDOTC

The routines CDOTU/CDOTC compute the unconjugated and conjugated dot products of two vectors. Obviously the timings of the routines are different. CDOTU is performed in 4 vector operations on $2n$ REAL elements and CDOTC is performed in 3 vector operations. This is explained as follows :
For the real part of the result we have

$$\text{Re CDOT}_{\text{U}}^{\text{C}} = \sum_i \text{Re}\, x_i \,.\, \text{Re}\, y_i \overset{+}{\underset{-}{}} \sum_i \text{Im}\, x_i \,.\, \text{Im}\, y_i$$

and for the imaginary part we have

$$\text{Im CDOT}_{\text{U}}^{\text{C}} = \sum_i \text{Re}\, x_i \,.\, \text{Im}\, y_i \overset{-}{\underset{+}{}} \sum_i \text{Im}\, x_i \,.\, \text{Re}\, y_i$$

Consider the result part Re CDOTC, two dot products have to be calculated, one of the real components and one of the imaginaries. However, the real and imaginary elements of the COMPLEX vector x are stored one after the other in the REAL vector rx, as is the case for the elements of y in ry. So, only one dot product of the vectors rx and ry will suffice to obtain Re CDOTC ($\Psi_m = 2.$ sec.) and no gathering of the real (imaginary) components is needed.
In case of Re CDOTU we are dealing with a minus sign, and consequently one dot product will be insufficient. Here, two dot products on rx and ry each with control vector $1\ 0\ 1\ 0\ \cdots$ are needed, followed by a subtraction of the scalar result values ($\Psi_m = 4.$ sec.).
In Im $\text{CDOT}_{\text{U}}^{\text{C}}$ mixed products of real and imaginary components appear. The first sum in Im $\text{CDOT}_{\text{U}}^{\text{C}}$ is obtained by calculating the dot product (with control vector $1\ 0\ 1\ 0\ \cdots$) of rx and ry, where ry is shifted one element. In a similar way, the second sum is found.
Summarizing, the theoretical time of CDOTC amounts $3\Psi_m = 6.$ sec., and of CDOTU $4\Psi_m = 8.$ sec..

CAXPY

We have to compute

$$\text{Re}\, y_i := \text{Re}\, y_i + \text{Re}\, ca \,.\, \text{Re}\, x_i - \text{Im}\, ca \,.\, \text{Im}\, x_i$$

$$\text{Im}\, y_i := \text{Im}\, y_i + \text{Re}\, ca \,.\, \text{Im}\, x_i + \text{Im}\, ca \,.\, \text{Re}\, x_i$$

for $i = 1, \ldots, n$. These formulas can be rewritten into

$$ry(1;2n) = ry(1;2n) + \text{Re}\, ca \,.\, rx(1;2n)$$

followed by

$$\text{Re}\, y_i = \text{Re}\, y_i - \text{Im}\, ca \,.\, \text{Im}\, x_i$$

$$\text{Im}\, y_i = \text{Im}\, y_i + \text{Im}\, ca \,.\, \text{Re}\, x_i$$

for $i = 1, \ldots, n$. This solution requires three LINKED TRIADs, the first one without a control vector, the latter two with control vectors. We observe that for the computation of the second and third LINKED TRIAD real and imaginary components of different vectors are needed at the same cycle. Here too (cf. the implementation of Im $\text{CDOT}_{\text{U}}^{\text{C}}$) a shift on one element of the vector rx or ry

enables us to start with the first imaginary component. The expected execution time amounts to $3\Psi_m = 6.$ sec..

CSCAL

Obviously, the implementation of this scaling operation is almost equivalent to that of CAXPY. We have to compute

$$\text{Re } x_i := \text{Re } ca \,.\, \text{Re } x_i - \text{Im } ca \,.\, \text{Im } x_i$$

$$\text{Im } x_i := \text{Re } ca \,.\, \text{Im } x_i + \text{Im } ca \,.\, \text{Re } x_i$$

For the real as well as the imaginary part, this would take at least two vector operations with control vectors. However, similarly to CAXPY, also these formulas can be rewritten into

$$rz(1;2n) = \text{Im } ca \,.\, rx(1;2n)$$

$$\text{Re } x_i = \text{Re } ca \,.\, \text{Re } x_i - \text{Im } z_i$$

$$\text{Im } x_i = \text{Re } ca \,.\, \text{Im } x_i + \text{Re } z_i$$

for $i = 1, \ldots, n$. The REAL vector rz (COMPLEX vector z) is a temporary vector with $2n$ elements (cq n COMPLEX elements). Again we have 3 vector operations, the first without and the latter two with a control vector (6. sec.).

CSSCAL/CSWAP/CCOPY

The operations are performed on the REAL vectors rx and ry. CSSCAL and CCOPY each take one vector operation on $m = 2n$ elements (2. sec.). The computation of CSWAP needs a temporary vector rz. This requires three copy operations (6. sec.).

CSROT

The subprogram CSROT performs the following operation on the COMPLEX vectors x and y :

$$x_i := \quad sc \,.\, x_i + ss \,.\, y_i$$

$$y_i := -ss \,.\, x_i + sc \,.\, y_i$$

for $i = 1, \ldots, n$. Since the values ss and sc are of type REAL, we are dealing with simple scalar times a COMPLEX vector multiplications, or, using rx and ry with scalar times a REAL vector multiplications. In section 4.1, we have demonstrated how the SROT operation can be performed in 4 vector operations and even in 3 operations. Similarly, the performance of CSROT can be done in 4 or 3 vector operations on the REAL vectors rx and ry. (CSROT4 : 8. sec.; CSROT3 : 6. sec.)

The description of our implementation is now complete for increment parameters being unity. The execution times of our implementations agree with the times measured for the CDC BLAS implementation in Table 5.2.1, except for SCASUM (measured 2.1 CPU sec.), CSROT4 (8.2 CPU sec.) and CSROT3 (6.1 CPU sec.)

5.4. Some tools for operating with strides in complex vectors

A storage spacing between COMPLEX elements of size ix will deliver the next pattern

$$R_1\, I_1 \,\ldots\, R_{ix+1}\, I_{ix+1} \,\ldots\, R_{2ix+1}\, I_{2ix+1} \,\ldots$$

For the REAL BLAS subprograms the use of the periodic GATHER appeared to provide a fast solution of the stride problem. Here, however, two consecutive elements are required to keep the specific storage of a COMPLEX vector. Hence, the periodic GATHER will fail, because only one element of a period can be stored into a temporary vector. A compression of the original vector seems to be a possibility to keep the COMPLEX vector storage. Merging and masking operations can be used to scatter the elements back after computation.

Unfortunately, all these operations need control vectors. The length of such control vectors dominates the execution time of the operations, so that, if the increment is large, these operations may be very expensive. Moreover, the creation of the control vectors enhances the execution time.

A better solution is again the periodic GATHER, but now operating on the real and the imaginary components, separately. A merge operation on the resulting two vectors restores in the original COMPLEX storage. But, why should we do this ? We would like to emphasize on this important question. A possible answer could be, that the subprograms could be performed in the same way as in case of increment values equal to 1. But, precisely the troublesome storage of COMPLEX vectors involves considerable effort, like operating with control vectors to obtain only the real components or just the other ones. Why should we not take full advantage of working with two *separated* vectors ?

In the next section we shall discuss this new situation and give the execution times of our implementation compared with the times of the CDC BLAS.

5.5. Our implementation of the complex BLAS with stride problems

The periodic GATHER is particularly suitable to handle strides. The major motivation to use this instruction is its substantial speed, its execution time only depending on the number of elements to be gathered. So, for large increment values this method will always be preferable. Even for small increment values this method is to be preferred.

For COMPLEX vectors, we have to gather twice, once to create a vector with the real components and once to create a vector with the imaginary components. Notice that the original storage of a COMPLEX vector (with stride 1) is

$$R_1 \; I_1 \; R_2 \; I_2 \cdots \tag{5.5.1}$$

We shall refer to this storage as **Storage I.** Two periodic gatherings will lead to

$$\begin{matrix} R_1 \; R_2 \cdots \\ I_1 \; I_2 \cdots \end{matrix} \tag{5.5.2}$$

referred to as **Storage II.** To diminish the number of startup times, the real and imaginary vectors are stored in the *same* array, after each other, wherever possible.

Let us now discuss our implementation of the COMPLEX BLAS with an additional stride problem based on Storage II. Only those subprograms are treated below, which can take full advantage of this storage. Here we assume, that if two vectors are involved, both vectors are stored nonsequentially . In section 5.6 we shall discuss the case of dealing with two vectors, one of which with increment value 1.

ICAMAX

As pointed out in section 5.3, the main part of the computation in this subprogram is to construct the vector z (cf formula (5.3.2)). Here, we are dealing with two separated vectors, and only one simple addition of these vectors (with sign control) is all we need, followed by a call of Q8SMAXI, of course.

CDOTU/CDOTC

As we may recall from our description of CDOTU/CDOTC for increments equal to unity, that inner products with control vectors are invoked. In the case of separated vectors no more time is wasted to compute unnecessary results.

Both for CDOTU and CDOTC it is recommended to put the vector with the real components and that with the imaginaries after each other (or the reverse) into one array in order to reduce the startup times.

CAXPY/CSCAL

It will be clear, that all LINKED TRIADs in the implementation of both subprograms can now be performed without control vectors. Obviously, in both cases, the first LINKED TRIAD is performed on the disjunct vectors of real and imaginary components. This gives a substantial gain in speed.

CCOPY

The execution time of this routine depends exclusively on the CPU time of the GATHER and SCATTER operations. Obviously, first the relevant real components of x are gathered, and next they are scattered into the desired locations of the vector y. An analogue process on the imaginary components is performed afterwards.

CSWAP

The implementation of CSWAP with increments equal to one, needs three copies. In this case, it is only a matter of gathering all relevant elements into temporary vectors, followed by scatter operations to restore them into the right locations.

At this point we are left to deliver the timings of our implementation and to compare them with the CDC BLAS-timings. The timings were accomplished for the following three combinations of increment values, with associated n values :

(*i*) $ix(=iy)=2$; $n = 5{,}000$
(*ii*) $ix(=iy)=10$; $n = 1{,}000$
(*iii*) $ix=2$ (, $iy=10$) ; $n = 1{,}000$

A combination of ($ix=10$ (, $iy=2$)) should yield the same timing results as combination (iii). Moreover, from the special choice of the size of n (cf formula (3.2)), it turns out that combination (ii) and (iii) result in the same execution times, despite the fact that the relevant elements of the vector x are differently located. The measured CPU times of these combinations can be found in Appendix I, Table A1.3.

Table 5.5.1 gives the timing results of combination (i), all subprograms are called 10,000 times.

complex BLAS, $n = 5{,}000$, $ix(=iy)=2$			
	CDC	CWI	%
ICAMAX	13.9	5.1	37
SCASUM	7.0	5.0	72
SCNRM2	5.1	5.1	101
CDOTU	22.2	10.3	46
CDOTC	20.2	10.3	51
CAXPY	23.7	13.9	58
CSCAL	21.5	10.6	50
CSSCAL	8.5	8.5	101
CSWAP	21.8	13.1	60
CCOPY	14.4	6.5	45
CSROT4	29.8	21.8	72
CSROT3	29.8	19.1	64

TABLE 5.5.1 COMPLEX BLAS, stride = 2

COMMENTS :

- The difference in execution speed between SCASUM (CDC) and SCASUM (CWI) versions is caused by the alternative approach of the BLAS operation after elimination of the stride problem (see section 5.3).

- The improvements of 'our' CSROT implementations are not based upon the advantage of Storage II. In particular, both storage modes will result in the same theoretical times, no control vectors were needed at an increment of 1. The gain arises from the use of 3 or 4 LINKED TRIADs instead of 6 as in the CDC BLAS implementation !
- The timing results of the CDC BLAS implementation indicate, that only GATHER and SCATTER operations are involved to handle the stride problems.
- All execution times of the CDC BLAS subprograms can be improved for all increment values greater than one, except for the subprograms SCNRM2 and CSSCAL, whose performance do not suffer on the inherent troublesome storage of a COMPLEX vector.
- Comparing the gain percentages of Table 5.5.1 with those of Table A1.3 (the results of combinations (ii) and (iii), see Appendix I), the conclusion must be, that all improvements based on Storage II are independent of the size of the increment value (> 1).
- The execution times are totally dominated by the size of the value n. As an example, a multiplication of the CPU times of both the CDC and CWI implementation of Table A1.3 with a factor 5 (i.e. the quotient of the n values used) roughly delivers the execution times of Table 5.5.1.

5.6. Complex BLAS with only one increment unequal to one

First of all, we state that the periodic SCATTER is not always the most efficient way to put elements into the right locations. We mentioned before that merging of two vectors is a very expensive operation. However, if the control vector contains as much 1's as 0's, this is not true. More explicitly, a MERGE operation on the vectors with real and imaginary components is more efficient, than two SCATTER operations. Table 5.6.1 will illustrate this.

increment	2	4	5	8	10
n	5000	2500	2000	1250	1000
2*SCATTER	3.5	1.8	1.2	.8	.6
MERGE	2.0	1.0	.8	.5	.4

TABLE 5.6.1

We notice that a MERGE operation instead of two SCATTER operations decreases the execution time with a factor of about $1\frac{1}{2}$.

In practice, one should often deal with only one increment unequal to unity. In this case, a choice must be made between an implementation based on Storage I and one based on Storage II (cf. pattern (5.5.1) and (5.5.2)). To illustrate which of them is to prefer, we shall give two examples, both concerning the subprogram CAXPY ($y := y + a \, . \, x$). We shall distinguish between two cases, ($ix = k > 1$, $iy = 1$) and ($ix = 1$, $iy = k > 1$). For the theoretical timings of the GATHER/SCATTER we refer to formula (4.2.1.1). All startup times will be ignored.

EXAMPLE I : CAXPY with $ix = k > 1$ and $iy = 1$

Storage I :

Two GATHER operations are needed to obtain the relevant real and imaginary elements of the vector x, followed by a MERGE operation to restore to Storage I. At this point we can continue with the implementation as described in section 5.3 (LINKED TRIADs on $2n$ elements).

Storage II :
First we have to gather all relevant real/imaginary elements of the vectors x and y to achieve storage mode II. The operation can now be performed in 4 LINKED TRIADS on n REAL elements. Finally, a MERGE operation is needed to restore the result vector in COMPLEX storage.

The theoretical results for both methods are listed below.

	op.	#	I	#	II
2*GATHER	$2\frac{1}{2}\ n$	1	$2\frac{1}{2}\ n$	2	$5\ n$
MERGE	$2\ n$	1	$2\ n$	1	$2\ n$
LINKED TRIADS on $2\ n$	$2\ n$	3	$6\ n$	-	
LINKED TRIADS on $1\ n$	n	-		4	$4\ n$
			+		+
			$10\frac{1}{2}\ n$		$11\ n$

EXAMPLE II : CAXPY with $ix = 1$ and $iy = k > 1$

Storage I :
The first part is analogous to $ix = k > 1$, $iy = 1$; however, in this case the relevant elements of the vector y are gathered and merged. Here the elements of the temporary result vector must be copied to the desired locations with a stride of k. The most efficient way for large increment values is the GATHER/SCATTER solution developed in the next table.

Storage II :
This method is almost identical to the previous example of Storage II. Here, the MERGE operation is replaced by two SCATTER operations, since the result vector y has an increment value of size k.

The theoretical times for both methods are listed below.

	op.	#	I	#	II
2*GATHER	$2\frac{1}{2}\ n$	2	$5\ n$	2	$5\ n$
2*SCATTER	$2\frac{1}{2}\ n$	1	$2\frac{1}{2}\ n$	1	$2\frac{1}{2}\ n$
MERGE	$2\ n$	1	$2\ n$	-	
LINKED TRIADS on $2\ n$	$2\ n$	3	$6\ n$	-	
LINKED TRIADS on $1\ n$	n	-		4	$4\ n$
			+		+
			$15\frac{1}{2}\ n$		$11\frac{1}{2}\ n$

The conclusion from both examples must be that there is no unique answer with respect to which method is to prefer in case of only one increment equals one. It will be clear from the previous examples, that in case of ($ix = k > 1$, $iy = 1$) the computation based on Storage I is to be considered, and, if ($ix = 1$, $iy = k > 1$) the Storage II approach is to prefer.

We would like to emphasize, that the theoretical times for the GATHER/SCATTER operation are very optimistic (see Table A2.1, Appendix 2). In practice, the difference in execution time between the Storage I and Storage II approach in example I will be more divergent (Storage II needs more GATHER operations).

A similar analysis can be made for CDOTU/CDOTC. It appears to be more efficient to follow the Storage II approach for these functions, avoiding dot products with control vectors. The subprograms CSWAP and CCOPY are implemented accordingly to Storage II, as described in the previous section. A MERGE operation instead of two SCATTER operations is involved in case of the result vector with increment value of size 1.

In Table 5.6.2 the execution times of both the CDC BLAS as well as our implementation are listed, where either $ix = 1$ or $iy = 1$; the value of the other increment parameter is $k = 2$. The suffices 12 and 21 symbolize the nonsymmetrical cases ($ix = 1$, $iy = 2$) and ($ix = 2$, $iy = 1$).

complex BLAS, $n = 5{,}000$ ($ix=1,iy=2$) $\vee$ ($ix=2,iy=1$)			
	CDC	CWI	%
CDOTU	15.2	10.1	66
CDOTC	13.2	10.1	77
CAXPY 12	16.2	13.9	86
CAXPY 21	13.3	11.0	83
CSWAP	12.7	11.5	91
CCOPY 12	7.2	6.6	91
CCOPY 21	7.4	4.9	66
CSROT4	21.1	19.6	93
CSROT3	21.1	17.8	84

TABLE 5.6.2 COMPLEX BLAS, mixed increment values

Similarly, to the case where both increments are greater than 1, the timings for the CDC BLAS implementation have been improved (by our implementations). However, the gain is less spectacular, as can be seen in the Tables 5.5.1 and A1.3 (in Appendix I).

5.7. Conclusions

Splitting the real and imaginary components for the COMPLEX BLAS without an additional stride problem is not the most efficient manner. But, if there is a stride problem, and consequently disjunct vectors of real and of imaginary components are created, it is recommended to take full advantage of this obtained data structure.

All execution times of the CDC BLAS subprograms can be improved for all increment values greater than one, except for the subprograms SCNRM2 and CSSCAL. The performance of both SCNRM2 and CSCAL do not suffer on the inherent troublesome storage of a COMPLEX vector. Like the REAL SASUM and SROT, also the execution times of the COMPLEX routines SCASUM and CSROT can be speed up.

In case of an additional stride problem the performance with two disjunct vectors delivers such a considerable gain, that even when only one vector has an increment value greater than one, it is to prefer to split the real and imaginary components of the other sequentially stored vector.

In contradiction with the REAL BLAS, all suggestions are independent of the size of the increment value.

ACKNOWLEDGEMENTS

The author would like to thank Drs. J. Schlichting for his valuable suggestions on the implementation of S(C)ASUM and (C)SROT3. She wishes to acknowledge Prof.dr. H.A. van der Vorst and Dr.ir. H.J.J. te Riele for their constructive remarks and careful reading of the manuscript.

REFERENCES

1. R.L. ANDERSON ET AL. (1984). *IMSL user's manual,* Problem-solving software system for mathematical and statistical FORTRAN programming
2. CDC. Cyber 200 ASSEMBLER reference manual, version 2, publ. number 60485010
3. CDC. Cyber 200 FORTRAN reference manual, version 1, publ. number 60480200B
4. J.J. DONGARRA, J.R. BUNCH, C.B. MOLER, and G.W. STEWART (1979). LINPACK user's guide, *SIAM publications.*
5. J.J. DONGARRA, J. DU CROZ, S. HAMMARLING, and R.J. HANSON (1985). A proposal for an extended set of Fortran basic linear algebra subprograms, *SIGNUM*, 20.
6. A. EMMEN (1984). *Cyber 205 user's guide*, SARA, Amsterdam, part 3, optimization of FORTRAN programs.
7. 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.
8. NUMVEC. *A library of NUMerical software for VECtor and parallel computers in FORTRAN*, Centre for Mathematics and Computer Science, Amsterdam.
9. B.T. SMITH, J.M. BOYLE, J.J. DONGARRA, B.S. GARBOW, Y. IKEBE, V.C. KLEMA, and C.B. MOLER (1976). Matrix Eigensystem routines - EISPACK guide, *Lecture Notes in Computer Science*, 6, Springer-Verlag, Berlin.
10. H.A. VAN DER VORST and J.M. VAN KATS (1984). The performance of some linear algebra algorithms in FORTRAN on CRAY-1 and CYBER 205 supercomputers, *ACCU*, 5-6.

Appendix 1

In order to show, that not only the CDC BLAS, but also the CWI BLAS timing results are independent of the increment value, or stride, we have experimented with increment value 10. As a consequence, the relevant vector elements are more scattered through the vectors. Keeping the dimension sizes constant and equal to the values given in (3.1) (i.e. 20,000 for REAL vectors and 10,000 for COMPLEX vectors) and by doing this, avoiding needless expensive LP FAULTS, the value of n must be adapted, such that the elements still fit in the vectors. With formula (2.2.1) the maximum allowed value

$$n = \frac{nx}{10}$$

is obtained, this yields $m = 2{,}000$. Again, all subprograms are called $\alpha = 10{,}000$ times. The theoretical time, needed for one single vector operation becomes

$$\Psi'_m = .4 \text{ CPU seconds}$$

Comparing this value of Ψ'_m with the value of Ψ_m given in (3.5), we notice that the theoretical time decreases here with a factor 5, and consequently, all measured CPU times roughly decrease with this factor. Though the startup times are still neglectable with respect to the vector length, the overhead here is relatively larger than in case of a stride 2.

The tables list the following results :

A1.1 corresponds with Table 4.3.1, except here, $n = 2{,}000$ and ix, $iy \in \{1, 10\}$.

A1.2 corresponds with Table 4.4.1, and lists the timings of the improved subprograms SASUM and SROT. The other execution times of the CWI BLAS correspond with the values of Table A1.1.

A1.3 contains the results of combinations (ii) and (iii) as mentioned in section 5.5.

A1.4 corresponds with Table 5.6.2, with mixed increment values. The suffices k1 and 1k symbolize the nonsymmetrical cases, ($ix=k=10$, $iy=1$) and ($ix=1$, $iy=k=10$).

real BLAS, $n = 2{,}000$				
	$ix=1$ ($iy=1$)	$ix=10$ ($iy=10$)	$ix=1$ $iy=10$	$ix=10$ $iy=1$
ISAMAX	.5	1.2	-	-
SASUM	.9	1.5	-	-
SNRM2	.5	1.2	-	-
SDOT	.5	1.8	1.1	-
SAXPY	.5	2.4	1.7	1.2
SSCAL	.5	1.7	-	-
SSWAP	1.3	2.6	1.7	-
SCOPY	.5	1.3	.7	.7
SROT	2.5	4.2	3.4	-

TABLE A1.1 CDC version

real BLAS, $n = 2{,}000$			
	$ix=1$ $(iy=1)$	$ix=10$ $(iy=10)$	$ix=1$ $iy=10$
SASUM	.5	1.2	-
SROT4	1.7	4.3	3.0
SROT3	1.3	3.9	2.6

TABLE A1.2 CWI implementation

complex BLAS, $n = 1{,}000$ $ix(=iy)=10 \vee ix=2\ (iy=10)$			
	CDC	CWI	%
ICAMAX	2.9	1.1	38
SCASUM	1.4	1.1	61
SCNRM2	1.0	1.1	105
CDOTU	4.6	2.2	47
CDOTC	4.2	2.2	52
CAXPY	5.0	2.8	56
CSCAL	4.5	2.1	47
CSSCAL	1.6	1.6	104
CSWAP	4.4	2.4	53
CCOPY	3.1	1.3	39
CSROT4	6.0	4.1	68
CSROT3	6.0	3.7	60

TABLE A1.3 COMPLEX BLAS, combinations (ii) and (iii)

complex BLAS, $n = 1{,}000$ $(ix=1, iy=10) \vee (ix=10, iy=1)$			
	CDC	CWI	%
CDOTU	3.2	2.2	69
CDOTC	2.8	2.2	79
CAXPY $1k$	3.5	2.8	81
CAXPY $k1$	2.8	2.4	85
CSWAP	2.7	2.3	85
CCOPY $1k$	1.6	1.2	78
CCOPY $k1$	1.6	1.1	69
CSROT4	4.3	4.1	95
CSROT3	4.3	3.6	84

TABLE A1.4 COMPLEX BLAS, mixed increment values

Appendix 2

In formula 4.2.1.1, the theoretical timing of the GATHER and SCATTER operations are given. In practice, the factor $\frac{5}{4}$ appeared to be to optimistic. In Table A2.1, we list the really obtained factor f for a GATHER/SCATTER operation on $n = 1{,}000$ elements. Again $\alpha = 10{,}000$ and c denotes the cycle time on the Cyber 205. This yields

$$f = \frac{\text{measured CPU time}}{\alpha * n * c}$$

Both positive and negative increment values are involved.

increment	GATHER	SCATTER	increment	GATHER	SCATTER
-64	4.20	4.21	0	4.20	4.21
-63	2.58	2.74	1	2.83	2.76
-62	1.50	1.49	2	1.57	1.49
-61	1.54	1.65	3	1.92	1.66
-60	1.58	1.74	4	1.44	1.76
-59	1.92	1.82	5	1.68	1.82
-58	1.69	1.81	6	1.69	1.81
-57	1.88	1.85	7	1.62	1.85
-56	1.74	1.87	8	1.74	1.88
-48	1.52	1.51	16	1.51	1.51
-40	1.68	1.63	24	1.62	1.64
-32	2.27	2.27	32	2.27	2.27
-24	1.63	1.46	40	1.64	1.46
-16	1.63	1.39	48	1.63	1.40
-8	1.54	1.23	56	1.54	1.24
-7	1.73	1.42	57	1.38	1.42
-6	1.63	1.55	58	1.54	1.56
-5	1.56	1.46	59	1.56	1.47
-4	1.45	1.35	60	1.44	1.35
-3	1.65	1.36	61	1.56	1.35
-2	1.45	1.27	62	1.44	1.27
-1	2.83	2.72	63	2.58	2.74
0	4.20	4.21	64	4.20	4.21

TABLE A2.1 GATHER/SCATTER factors