NASA Technical Memorandum 4028

An Efficient Sparse Matrix Multiplication Scheme for the CYBER 205 Computer

Jules J. Lambiotte, Jr.

MARCH 1988

NASA

NASA Technical Memorandum 4028

An Efficient Sparse Matrix Multiplication Scheme for the CYBER 205 Computer

Jules J. Lambiotte, Jr.
Langley Research Center
Hampton, Virginia

National Aeronautics
and Space Administration

**Scientific and Technical
Information Division**

1988

Introduction

Many of the important numerical techniques used today to solve linear equations require repeated computation of a symmetric matrix times a vector. Examples are the conjugate gradient method, with all its variants, for solving simultaneous linear equations (refs. 1 and 2) and the Lanczos algorithm for eigenvalue and eigenvector extraction (ref. 3). These methods are particularly attractive when the matrix is sparse since, unlike direct methods, they do not require storage of the entire matrix. The matrix is only used to multiply a vector, and thus one needs to know only the nonzero elements and their positions within the matrix.

The primary objective of this work has been to develop software for the Control Data Corporation CYBER 205 computer that provides an efficient means for computing $\mathbf{b} = \mathbf{Ax}$ when $\mathbf{A}$ is an $n \times n$, symmetric, sparse matrix and $\mathbf{x}$ is a vector.

Because use of vector hardware instructions on a vector processor has very definite implications about the storage, a user's goals of minimizing both the required central processing unit (CPU) time and the total storage needed to represent $\mathbf{A}$ often conflict. Thus, a more specific objective of the work has been to design the software so that it provides alternative storage and computational procedures for the matrix $\mathbf{A}$ and automatically selects the procedure which best reflects the user's relative concerns about minimizing the two resources.

These objectives have led to the development of a diagonal-based storage and computation scheme in which a preprocessing subroutine, Q4CMPCTD, chooses one of four storage methods for each diagonal using CPU time and storage estimates and user-provided resource weighting information. A matrix multiplication subroutine, Q4CMPYD, can then be called repeatedly to compute $\mathbf{Ax}$ using the compact form of matrix $\mathbf{A}$.

Subsequent sections of the paper describe the relevant CYBER 205 instructions used, the diagonal-based algorithm with the trade-offs between the methods, a description of the implementation used, and the results for a sample sparse matrix.

Symbols

$a_{j,k}$	the j,kth element in the matrix $\mathbf{A}$
$\mathbf{A}$	symmetric banded matrix
$\mathbf{A}(\ell)$	the ℓth subdiagonal (or superdiagonal) in $\mathbf{A}$
b_j	the jth element of $\mathbf{b}$
$\mathbf{b}$	right-hand-side vector in the matrix equation $\mathbf{Ax} = \mathbf{b}$
BT	bit vector
C	FORTRAN array containing compacted form of matrix $\mathbf{A}$
c	predicted CPU requirement
d	fraction of nonzeros in a diagonal, diagonal density
ℓ	diagonal designator ($\ell = 0$ is main diagonal)
m	arbitrary diagonal length
n	number of rows of matrix $\mathbf{A}$
p	number of 64-bit words needed for bit vector
r	weighted resource
s	predicted storage requirement
$\mathbf{x}$	vector to be multiplied by matrix $\mathbf{A}$
z	number of nonzeros in diagonal

Subscripts:

w	weight
min	minimum value
j,k	general indices

Abbreviations:

CPU	central processing unit
MFLOPS	millions of floating point operations per second
.NE.	FORTRAN "not equal to" relation

CYBER 205 Characteristics

The CYBER 205 at Langley Research Center, designated the VPS-32 there, is a vector processing computer manufactured by Control Data Corporation, which has a peak computing rate of 400 million 32-bit floating point operations per second (MFLOPS) on certain computations. For the more prevalent computations such as vector multiplication or addition, this two-pipe configuration can achieve up to 100 MFLOPS in 64-bit arithmetic mode. The VPS-32 is a bit addressable, virtual memory computer which has 32 million 64-bit words of central memory.

The high CPU rates are achieved by operations on long vectors whose components, by definition, are consecutively stored in memory. However, if vector lengths are short (say, 10 or less), the scalar CPU speed makes serial computation superior.

In addition to the usual floating point arithmetic operations (addition, subtraction, multiplication, and division), several nontypical hardware instructions exist which have proved useful in this work. These are the vector *compare, compress, expand, bit count, gather*, and *scatter*. Figure 1 demonstrates their use. Note that the *compress* and *expand* sequence and the *gather* and *scatter* sequence can be used to accomplish the same data movement. The relative efficiency depends on the sparsity of the data list being accessed. The *compress* and *expand* instructions each take 10 nsec per element in the bit vector (including off bits). The *gather* and *scatter* instructions each require 25 nsec per element moved, but since they move only nonzero elements, they may be faster for sparse lists.

Diagonal-Based Matrix Multiplication

It is possible to describe the multiplication process $\mathbf{b} = \mathbf{Ax}$ for a matrix $\mathbf{A}$ in terms of elements of each diagonal. Let $\mathbf{A}(\ell)$ denote the ℓth superdiagonal (also the ℓth subdiagonal since $\mathbf{A}$ is symmetric) and let $A_k(\ell)$ be the kth component; that is, $A_k(\ell) = a_{k,k+\ell} = a_{k+\ell,k}$. The procedure for computing $\mathbf{b} = \mathbf{Ax}$ for the $n \times n$ matrix $\mathbf{A}$ is

$$b_k \leftarrow A_k(0)x_k \qquad (k = 1, 2, \ldots, n)$$

For $\ell = 1, 2, \ldots, n-1$

$$b_k \leftarrow b_k + A_k(\ell)x_{k+\ell} \qquad (\text{for } k = 1, 2, \ldots, n-\ell) \quad (1)$$

$$b_{k+\ell} \leftarrow b_{k+\ell} + A_k(\ell)x_k \qquad (\text{for } k = 1, 2, \ldots, n-\ell) \quad (2)$$

End For

Note that if $\mathbf{A}$ is banded, ℓ need range only from 1 to the bandwidth, and that if any diagonals are identically zero, they can be easily identified and all computation for them in equations (1) and (2) can be omitted.

The diagonal-based scheme has been selected as the foundation for this work for several reasons:

1. *Nonzero structure of real problems*—Many matrices arising from finite difference or finite element formulations naturally lead to a sparsity pattern in which most of the nonzeros lie along a few of the diagonals. The five-diagonal matrix arising from central differencing of Poisson's equation is an extreme example. Of course, there the pattern is so predictable that special storage techniques are not needed; but for more complex equations with more complicated differencing or for finite element formulations using nonuniform elements, the sparsity is not so easily specified.
2. *Vectorization*—The $n - \ell$ multiplications and additions in equations (1) and (2) can be carried out by vector operations of length $n - \ell$.
3. *Symmetry of diagonals*—The ℓth subdiagonal is also the ℓth superdiagonal. Since equations (1) and (2) are identical in form, the storage and computation most appropriate for the subdiagonal are also most appropriate for the superdiagonal.

Storage Trade-Offs

The vector computations implied in equations (1) and (2) assume that $\mathbf{A}(\ell)$ is available as a vector of length $n - \ell$. However, if the diagonal is relatively sparse, one might not want to store the entire diagonal with all its zeros. In fact, if the diagonal is very sparse, neither vector storage nor vector computation is likely to be very efficient.

Described below are four types of diagonal storage and associated computation to execute equations (1) and (2). Note that types 3 and 4 differ only in the computational scheme employed.

Type 1. *Full vector*—The entire diagonal is stored including any zeros. Vectors of length $n-\ell$ are achieved using vector computation according to the algorithm described by equations (1) and (2). This mode is most efficient when $\mathbf{A}(\ell)$ is very dense.

Type 2. *Compressed vector plus bit pattern*—Only the nonzeros are stored along with a bit vector to give positional information within the diagonal. The computation is identical to that with type 1 diagonals after an *expand* (see fig. 1(c)) is performed to generate the full diagonal $\mathbf{A}(\ell)$. The extra *expand* makes type 2 CPU time always exceed that for type 1, but the storage can be considerably less.

Type 3. *Compressed vector plus row pointers with serial computation*—Only the nonzeros are stored along with an index vector to provide positional information. The assumption is that $\mathbf{A}(\ell)$ is so sparse and short that it would be inefficient to *expand* the compressed vector to use vector computation. Equations (1) and (2) are executed serially making use of the row indices stored in the index vector.

Type 4. *Compressed vector plus row pointers with vector computation*—The storage is identical to a type 3 diagonal. The difference lies in the manner in which the computation is carried out. The index vector is used to *gather* (see fig. 1(b)) the appropriate elements from $\mathbf{x}$ and $\mathbf{b}$, and then to *scatter* back out to $\mathbf{b}$ the result of the computation which has been carried out on vectors which have the length of

the number of nonzeros in the diagonal. If the diagonal is very sparse, so that type 1 or type 2 storage is inappropriate, yet long enough that the number of nonzeros leads to long vectors, this computational procedure is superior to the serial computation associated with type 3 diagonals.

Figures 2 and 3 show the CPU and storage requirements for a diagonal of length 1000 as a function of density d, where d is the fraction of nonzeros in the diagonal. A comparison of the two figures shows that, unfortunately, one cannot identify intervals of density where a particular diagonal type is most efficient with respect to both resources. For instance type 4 diagonals require the least CPU time for $d < 0.13$, but require greater storage than type 2 diagonals for $d > 0.02$. Even in those regions where one diagonal type is most efficient for both resources (type 1 for very dense and type 3 or 4 for very sparse), the boundaries of these regions vary with the length of the diagonal.

Since the minimization of both resources is frequently not possible and since different users may attach different importances to the two resources, it was decided to let the user influence the storage selection through resource weighting factors. To implement this decision, the initialization subroutine Q4CMPCTD does the following for each diagonal:

1. Estimates the CPU and storage requirements for each of the four candidate types
2. Applies a user-supplied weight to compute the weighted resource requirement for each method
3. Selects the storage type that minimizes the sum of the two weighted resource requirements

With the predicted storage and CPU requirements for the ith diagonal type denoted by s_i and c_i, the minimums over i denoted by $s_{\min}$ and $c_{\min}$, and the user-specified weightings denoted by s_w and c_w, then the normalized and weighted resource r_i for the ith diagonal type is computed as

$$r_i = \frac{s_i}{s_{\min}} s_w + \frac{c_i}{c_{\min}} c_w \qquad (i = 1, 2, 3, 4)$$

Subroutine Q4CMPCTD computes an r_i for each diagonal and selects the diagonal type which corresponds to the minimum value. For this approach, Q4CMPCTD must be able to estimate s_i and c_i for all diagonal lengths m and densities d. The storage estimates are easily made in terms of a diagonal of length m having z nonzeros:

$$s_1 = m$$
$$s_2 = z + p$$
$$s_3 = s_4 = 2z$$

where p is the least number of 64-bit words needed to hold m bits.

The CPU estimates were obtained by timing the computation for a range of diagonal length m and density d. For types 1, 3, and 4 diagonals, single formulas were obtained, but the complexity of the *expand* used in type 2 diagonal computation required a table of values (table I). The time in microseconds (on the CYBER 205 computer) to perform the computations implied in equations (1) and (2) for a single diagonal can be estimated by

$$c_1 = 8 + 0.040m$$
$$c_3 = 5 + 1.619z$$
$$c_4 = 18 + 0.2066z$$

and c_2 is obtained through linear interpolation for each variable within table I. Since the values of c_2 are used only in a selection process, their accuracy to a few percent is sufficient.

Table I. CPU Times for Type 2 Diagonals as a Function of Diagonal Length m and Density d

	CPU time, μsec			
d	$m = 25$	$m = 500$	$m = 1000$	$m = 10000$
0	11	36	61	511
.2	11	36	63	539
1.0	11	36	61	511

Implementation

The matrix is passed to subroutine Q4CMPCTD in its expanded form as an $n \times n_d$ array. Each of the n_d diagonals is treated individually as the compact representation, array C, is formed. Array C is linear with the pertinent data for the ℓth diagonal stored behind that for the $(\ell - 1)$st diagonal. As illustrated in figure 4, this can be, for type 1, 2, or 3, respectively, either the entire diagonal, the nonzero bit pattern for the diagonal followed by the nonzeros, or the nonzeros and index vector. Type 4 diagonals are stored in the same way as type 3. A vector *compare* with zero generates the bit pattern and provides the number of nonzeros and density (fig. 1(a)). If

the weighting procedure determines that the diagonal should be type 2, 3, or 4, a *compress* is performed to extract the nonzeros (fig. 1(b)). In addition, four integers for each diagonal are stored in a separate array. The first identifies the diagonal type; the second is the number of nonzeros in the diagonal; the third and fourth identify the positions of the first and last nonzeros within the diagonal, respectively. The latter two integers provide a relatively simple means for increasing efficiency. For the small price of storing these two extra values per diagonal, the leading and trailing zeros for each diagonal no longer have to be included in type 1 or type 2 diagonal storage or computation. The effect this can have is demonstrated in the next section.

The initialization subroutine returns to the user the CPU and storage estimates for the user-provided weights. In addition, the estimates for the combinations $s_w = 1$, $c_w = 0$ and $s_w = 0$, $c_w = 1$ are returned to aid the user in adjusting the weights in subsequent computations.

Once the compacted array C has been formed, subroutine Q4CMPYD can use it and the four integers describing the diagonal to carry out the **A** by **x** multiplication.

Results

Results from a test matrix are presented in table II to demonstrate the effect and control the user has on the matrix storage and computational requirements by giving the statistics for different combinations of s_w and c_w.

The test matrix is a sparse matrix resulting from a finite element formulation with triangular elements and three degrees of freedom at each node. The matrix has 1086 equations and a bandwidth of 81. Most of the diagonals are quite sparse. In fact, 57 of them are less than 5 percent dense. Approximately half of the nonzeros lie on the main diagonal and the three closest subdiagonals. The average density is 7.8 percent. The effective density of each diagonal is increased by considering only the portion of the diagonal beginning with the first nonzero and ending with the last one as discussed in the previous section. Considered in this way, the average density of the matrix is increased to 25.7 percent. There are now only 11 diagonals whose effective density is less than 5 percent. Forty-four of the diagonals have a density between 5 percent and 25 percent.

This example demonstrates the conflicting goals of minimizing both resources. It also shows that use of the weighting factors can give the user a rather wide range of resource distributions. For instance, a weighting of 1 for c_w and 0 for s_w leads to a CPU time that is minimum but a storage requirement which is 2.41 times that if one set $s_w = 1$ and $c_w = 0$. However, setting $s_w = 1$ yields a CPU time which is 1.40 times the minimum. A reasonable middle ground occurs when $s_w = c_w = 0.5$. In this case, the CPU time is 1.20 times the minimum, and the storage is 1.09 times the minimum.

Concluding Remarks

This paper has described a computational and storage algorithm for sparse matrix multiplication on a Control Data Corp. CYBER 205 computer. The multiplication is performed using diagonals of the matrix as the candidate vectors since this is where nonzero patterns predominate in many scientific applications. Four types of diagonal sparsity patterns are identified (dense, moderately dense, sparse and long, and sparse and short) and storage and computational procedures developed for each.

Since, for most densities, no single diagonal type minimizes both storage and CPU requirements, an initialization subroutine selects the most "efficient" type for the diagonal on the basis of estimated resource requirements and user-provided weights that indicate the relative importance the user attaches to each resource.

Table II. Storage and Computational Requirements for 81 × 1086 Finite Element Matrix
[Average diagonal density, 25.7 percent]

Weights		Resources		Diagonal selection			
c_w	s_w	CPU time, μsec	Storage, words	Type 1	Type 2	Type 3	Type 4
0	1.0	2.40	7254	2	79	0	0
0.3	.7	2.30	7452	4	71	0	6
.5	.5	2.09	7905	7	58	0	16
.7	.3	1.96	8406	11	44	0	26
1.0	0	1.74	17455	54	0	1	26

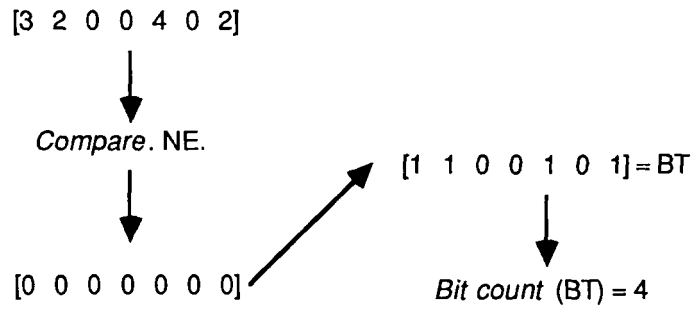
The example given demonstrated that, for a given matrix, the weights could be used to achieve minimal CPU time (at the expense of storage) or minimal storage (at the expense of CPU time) or some compromise between the two. For an example matrix (with average diagonal density of 25.7 percent), a choice of weights which minimized CPU time gave a CPU requirement that was only 70 percent of what would have been required if one had chosen to minimize the storage, but at the expense of nearly 2.5 times the storage. On the other hand, a selection of equal weights led to requirements which were within 20 percent of the respective minimums.

References

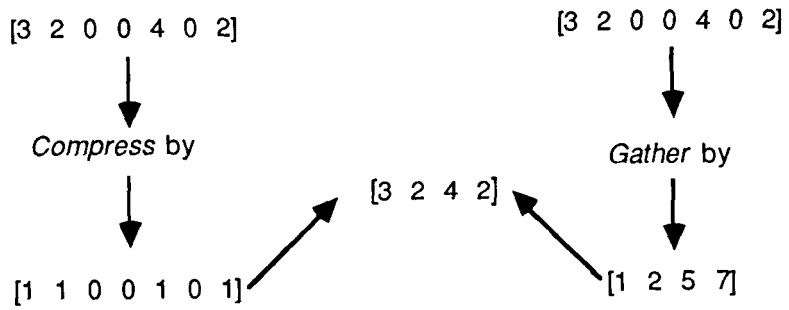
1. Hestenes, Magnus R.; and Stiefel, Eduard: Methods of Conjugate Gradients for Solving Linear Systems. Res. Paper 2379, *J. Res. Natl. Bur. Stand.*, vol. 49, no. 6, Dec. 1952, pp. 409–436.

2. Kershaw, David S.: The Incomplete Cholesky–Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations. *J. Comput. Phys.*, vol. 26, no. 1, Jan. 1978, pp. 43–65.

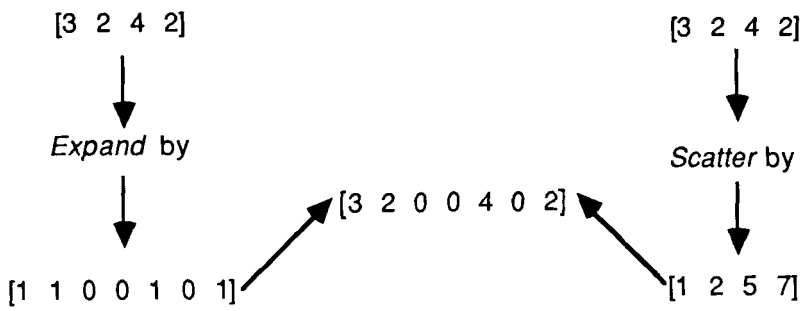
3. Wilkinson, J. H.: *The Algebraic Eigenvalue Problem.* Clarendon Press (Oxford), 1965, p. 388.



(a) Vector *compare* to **0** results in bit vector; *bit count* of BT gives number of nonzeros in original vector.



(b) Vector *compress* or *gather* results in compacted form.



(c) Vector *expand* or *scatter* returns vector to uncompacted form.

Figure 1. CYBER 205 nontypical vector instructions.

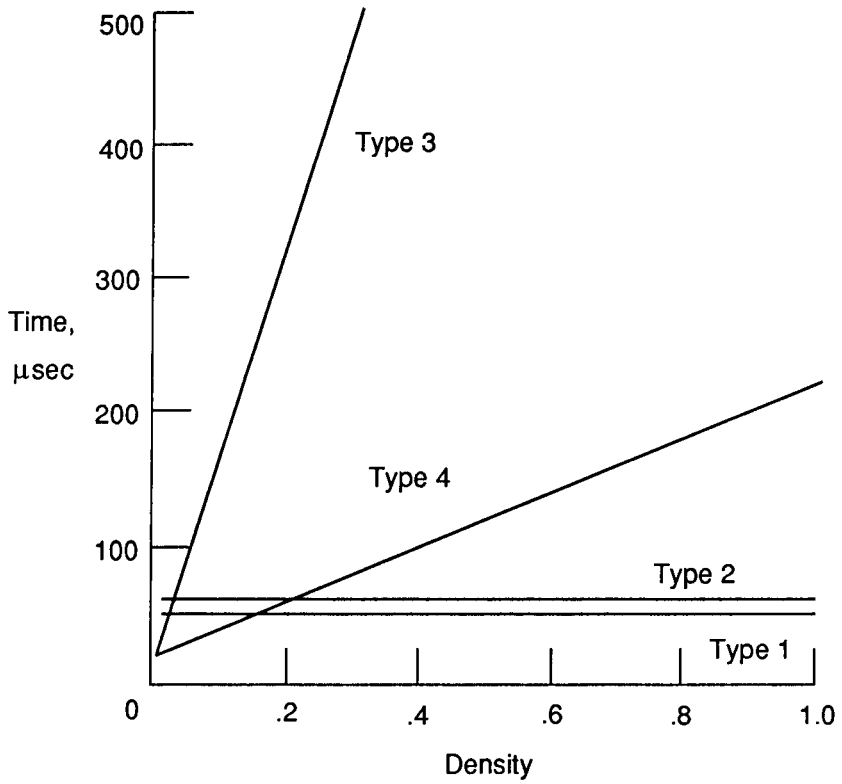


Figure 2. CPU time for diagonal with length 1000.

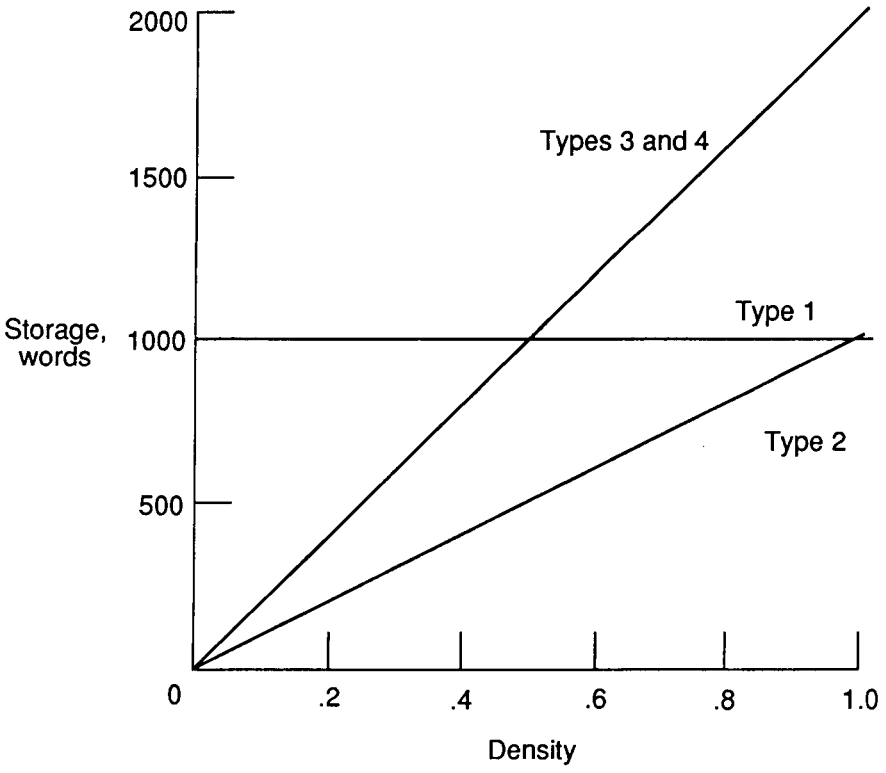
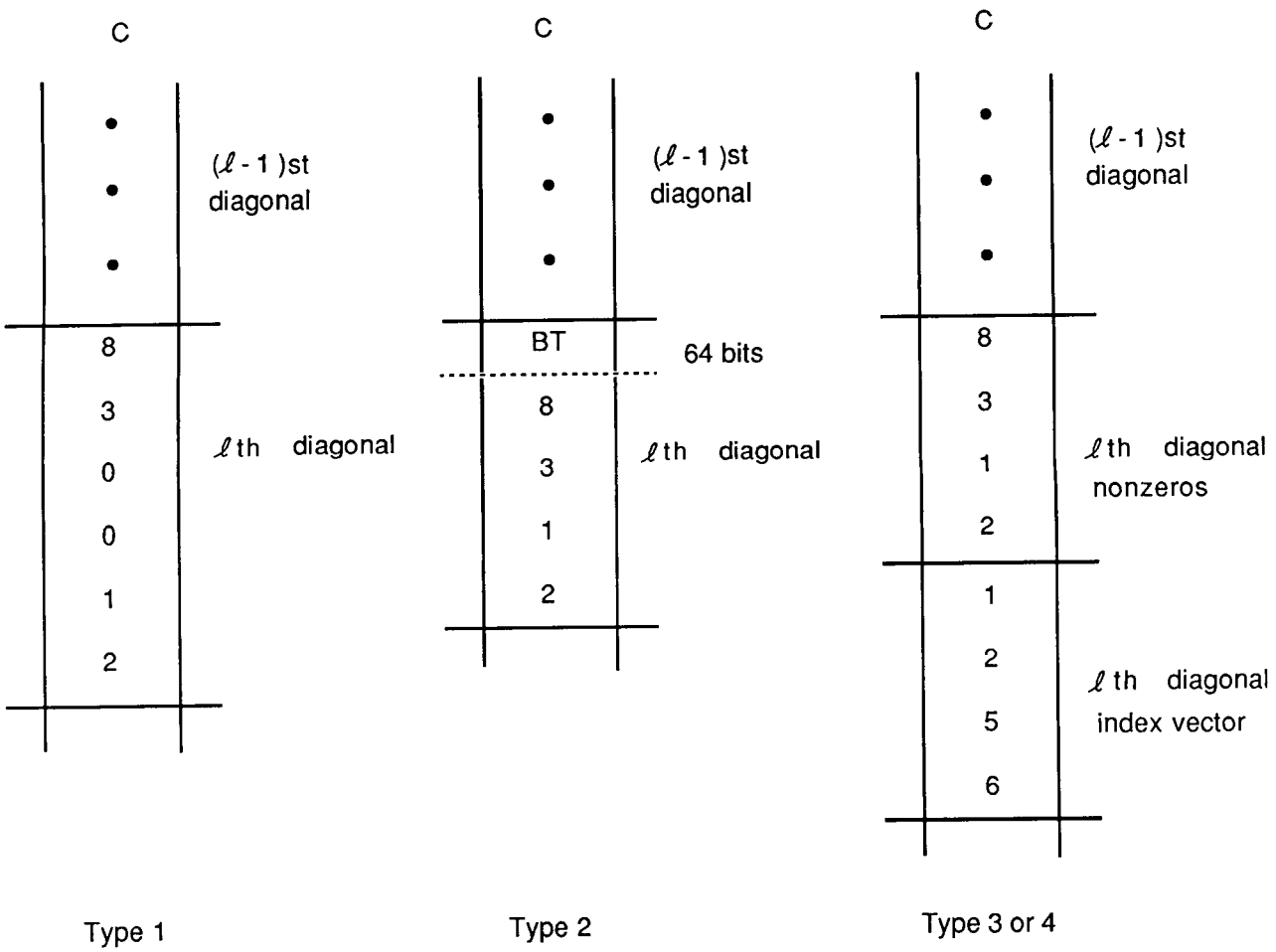


Figure 3. Storage requirements for diagonal with length 1000.



A(ℓ) = [8 3 0 0 1 2]
BT = [1 1 0 0 1 1 0 . . . 0]
64 bits

Figure 4. Storage for $\mathbf{A}(\ell)$ with $n - \ell = 6$.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-4028		

4. Title and Subtitle	5. Report Date
An Efficient Sparse Matrix Multiplication Scheme for the CYBER 205 Computer	March 1988
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Jules J. Lambiotte, Jr.	L-16403
9. Performing Organization Name and Address	10. Work Unit No.
NASA Langley Research Center Hampton, VA 23665-5225	505-90-21-02
	11. Contract or Grant No.
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, DC 20546-0001	Technical Memorandum
	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

This paper decribes the development of an efficient algorithm for computing the product of a matrix and vector on a CYBER 205 vector computer. The desire to provide software which allows the user to choose between the often conflicting goals of minimizing central processing unit (CPU) time or storage requirements has led to a diagonal-based algorithm in which one of four types of storage is selected for each diagonal. The candidate storage types employed were chosen to be efficient on the CYBER 205 for diagonals which have nonzero structure which is dense, moderately sparse, very sparse and short, or very sparse and long; however, for many densities, no diagonal type is most efficient with respect to both resource requirements, and a trade-off must be made. For each diagonal, an initialization subroutine estimates the CPU time and storage required for each storage type based on results from previously performed numerical experimentation. These requirements are adjusted by weights provided by the user which reflect the relative importance the user places on the two resources. The adjusted resource requirements are then compared to select the most efficient storage and computational scheme.

17. Key Words (Suggested by Authors(s))	18. Distribution Statement
Matrix-vector product Sparse matrices Diagonal storage Vector computer	Unclassified—Unlimited Subject Category 61

19. Security Classif.(of this report)	20. Security Classif.(of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	9	A02

For sale by the National Technical Information Service, Springfield, Virginia 22161-2171