An Alternative Compressed Storage Format for Sparse Matrices

Anand Ekambaram and Eurípides Montagne

School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816
{ekambara,eurip}@cs.ucf.edu

Abstract. The handling of the sparse matrix vector product(SMVP) is a common kernel in many scientific applications. This kernel is an irregular problem, which has led to the development of several compressed storage formats such as CRS, CCS, and JDS among others. We propose an alternative storage format, the Transpose Jagged Diagonal Storage(TJDS), which is inspired from the Jagged Diagonal Storage format and makes no assumptions about the sparsity pattern of the matrix. We present a selection of sparse matrices and compare the storage requirements needed using JDS and TJDS formats, and we show that the TDJS format needs less storage space than the JDS format because the permutation array is not required. Another advantage of the proposed format is that although TJDS also suffers the drawback of indirect addressing, it does not need the permutation step after the computation of the SMVP.

1 Introduction

The irregular nature of sparse matrix-vector multiplication, $Ax = y$, has led to the development of a variety of compressed storage formats, which are widely used because although these formats suffer the drawback of indirect addressing, they have the advantage that they do not store any unnecessary elements [1], [4], [5]. The main idea considered in compressed storage formats is to avoid the handling and storage of zero values. This is accomplished by means of the storage of the non-zero elements of the sparse matrix in a contiguous way using a linear array. However, some additional arrays are needed for knowing where the non-zero elements fit into the sparse matrix. The number of subsidiary arrays varies depending on the storage format used. Research into sparse matrix reorganization has dealt with developing various static storage reduction schemes such as Compressed Row Storage(CRS), Compressed Column Storage(CCS), and Jagged Diagonal Storage(JDS). One of these methods, the Jagged Diagonal Storage format(JDS) is, in addition, considered very convenient for the implementation of iterative methods on parallel and vector processors [3], [5]. In this work we present the Transpose Jagged Diagonal Storage format(TJDS), which is inspired from the Jagged Diagonal Storage format. The performance of both

© Springer-Verlag Berlin Heidelberg 2003

algorithms is similar in terms of the memory access pattern, four load operations and one store operation to compute each partial result. However, TJDS does not need a permutation array nor does it need a permutation step to compute the matrix vector product, $Ax = y$.

In Sect. 2 we describe how departing from the CRS and CCS formats we obtain the compressed matrices, A_{jds} and A_{tjds}, using the JDS format and the TJDS format respectively. In Sect. 3 we show a comparison of the memory space required for a set of matrices using each storage format. Also we give the execution times for the matrix-vector product using the JDS and TJDS algorithms. Finally, in Sect. 4 we present our conclusions.

2 Storage Requirements for the JDS and TJDS Formats

When a sparse matrix A is compressed using the Jagged Diagonal Storage format, all the non-zero elements(nze) in each row are shifted to the left and this way we obtain the matrix A_{crs}, a compressed row version of matrix A. Then we reorder the rows of the matrix A_{crs} in decreasing number of non-zero elements from top to bottom, in order to create a A_{jds} matrix, which is the jagged diagonal version of A_{crs}. As follows, we present the three steps required to transform a matrix A into is JDS version A_{jds}.

Step 1: The matrix vector product $Ax = y$ is expressed in matrix form as follows:

$$\begin{bmatrix} a_{11} & a_{12} & 0 & a_{14} & 0 & 0 \\ 0 & a_{22} & a_{23} & 0 & a_{25} & 0 \\ a_{31} & 0 & a_{33} & a_{34} & 0 & 0 \\ 0 & a_{42} & 0 & a_{44} & a_{45} & a_{46} \\ 0 & 0 & 0 & 0 & a_{55} & a_{56} \\ 0 & 0 & 0 & 0 & a_{65} & a_{66} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}$$

Step 2: By applying the CRS scheme to the sparse matrix A we obtain the matrix A_{crs}, while the vectors x and y remain the same.

$$A_{crs} = \begin{bmatrix} a_{11} & a_{12} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{23} & a_{25} & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{34} & 0 & 0 & 0 \\ a_{42} & a_{44} & a_{45} & a_{46} & 0 & 0 \\ a_{55} & a_{56} & 0 & 0 & 0 & 0 \\ a_{65} & a_{66} & 0 & 0 & 0 & 0 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}$$

Step 3: We obtain A_{jds} a Jagged Diagonal Storage version from A_{crs} by reordering the rows of A_{crs} in decreasing order from top to bottom according to the number of nonzero elements per row. It can be seen that the rows one and four of the matrix A_{crs} were permuted as shown below.

$$A_{jds} = \begin{bmatrix} a_{42} & a_{44} & a_{45} & a_{46} & 0 & 0 \\ a_{11} & a_{12} & a_{14} & 0 & 0 & 0 \\ a_{22} & a_{23} & a_{25} & 0 & 0 & 0 \\ a_{31} & a_{33} & a_{34} & 0 & 0 & 0 \\ a_{55} & a_{56} & 0 & 0 & 0 & 0 \\ a_{65} & a_{66} & 0 & 0 & 0 & 0 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}$$

We now store the A_{jds} as a single dimension array. The nonzero elements of the A_{jds} matrix are stored in a floating-point linear array *value_list*, one column after another. Each one of these columns is called a jagged diagonal and we use a semi-colon(;) in the array *value_list* to separate them. The length of the array *value_list* is equal to the number of non-zeros elements in A_{jds}. We also need another array *column_indexes* to store the column indices of the non-zero elements. Since the rows have been permuted to obtain a different matrix, A_{jds}, from the original matrix A, we need an array *perm_vector* to permute the resulting vector back to the original ordering. Obviously the size of this array is n for an n-dimensional matrix. A fourth array is also needed, *start_positions*, which stores the starting position of the jagged diagonals in the array *value_list*. The length of this array is the number of elements in the row of A having the maximum number of non-zero elements plus one. The last element of this array is used to control the inner *for* loop of the algorithm shown below. The data structures required to compute $Ax = y$ using JDS are shown below:

value_list	a_{42}	a_{11}	a_{22}	a_{31}	a_{55}	$a_{65};$	a_{44}	a_{12}	a_{23}	a_{33}	a_{56}	$a_{66};$	a_{45}	a_{14}	a_{25}	$a_{34};$	a_{46}

column_indexes	2	1	2	1	5	5;	4	2	3	3	6	6;	5	4	5	4;	6

start_positions	1	7	13	17	18

perm_vector	4	2	3	1	5	6

X	x_1	x_2	x_3	x_4	x_5	x_6

Y	y_1	y_2	y_3	y_4	y_5	y_6

The sequential algorithm to perform the matrix-vector product $Ax = y$ using the JDS format is shown below. In this algorithm, *num_jdiag* stands for the number of jagged diagonals:

```
$disp := 0$
for $i \leftarrow 1$ to $num\_jdiag$
    for $j \leftarrow 1$ to (start_position[i+1] - start_position[i] - 1)
        $Y[j] := Y[j] + value\_list[disp] \times X[column\_indexes[disp]]$
        $disp := disp + 1$
    endfor
endfor
```

The performance of this algorithm is clearly determined by the number of memory accesses which are four load operations and one store operation to compute each partial result. In addition, we need to perform the permutation of the resulting vector back to the original ordering using an algorithm similar to the one shown below:

```
for $i \leftarrow 1$ to N
    $Temp[perm\_vector[i]] := Y[i]$
for $i \leftarrow 1$ to N
    $Y[i] := Temp[i]$
```

Another popular format for storing sparse matrices is the Compressed Column Storage(CCS) format or Harwell-Boeing sparse matrix format. In this storage format a matrix A is compressed along the columns by shifting all the non-zero elements(nze) upwards. By applying the CCS to the sparse matrix A we obtain the A_{ccs} matrix, while the vectors x and y remain the same, as shown below:

$$A_{ccs} = \begin{bmatrix} a_{11} & a_{12} & a_{23} & a_{14} & a_{25} & a_{46} \\ a_{31} & a_{22} & a_{33} & a_{34} & a_{45} & a_{56} \\ 0 & a_{42} & 0 & a_{44} & a_{55} & a_{66} \\ 0 & 0 & 0 & 0 & a_{65} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}$$

The Transpose Jagged Diagonal Storage(TJDS) scheme is obtained from the CCS format by reordering the columns of A_{ccs}, from left to right, in decreasing order of the number of non-zero elements per column and reordering the elements of the vector x accordingly as if it were an additional row of A. This new arrangement, A_{tjds}, is presented as follows:

$$A_{tjds} = \begin{bmatrix} a_{25} & a_{12} & a_{14} & a_{46} & a_{11} & a_{23} \\ a_{45} & a_{22} & a_{34} & a_{56} & a_{31} & a_{33} \\ a_{55} & a_{42} & a_{44} & a_{66} & 0 & 0 \\ a_{65} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad x = \begin{bmatrix} x_5 \\ x_2 \\ x_4 \\ x_6 \\ x_1 \\ x_3 \end{bmatrix} \quad y = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}$$

To obtain the corresponding arrays, $value\_list, row\_indexes$, and $start\_position$ we proceed as follows. The nonzero elements of the compressed ordered matrix A_{tjds} are stored in a floating point linear array $value\_list$, one row after another. Each of these rows is called a transpose jagged diagonal(tjd). Another array of the same length of $value\_list$ is used to store the row indexes of the non-zero elements in the original matrix A. This new array that we denote, $row\_indexes$, is of type integer. Finally, a third array called $start\_position$ stores the starting position of each tjd stored in the array $value\_list$. These arrays are shown below:

value_list	a_{25}	a_{12}	a_{14}	a_{46}	a_{11}	$a_{23;}$	a_{45}	a_{22}	a_{34}	a_{56}	a_{31}	$a_{33;}$	a_{55}	a_{42}	a_{44}	$a_{66;}$	a_{65}

row_indexes	2	1	1	4	1	2;	4	2	3	5	3	3;	5	4	4	6;	6

start_position	1	7	13	17	18

X	x_5	x_2	x_4	x_6	x_1	x_3

Y	y_1	y_2	y_3	y_4	y_5	y_6

In the above data structures, each transpose jagged diagonal is separated by a semi-colon(;) in the array *value_list*. The sequential algorithm to perform the matrix-vector product $Ax = y$ using the TJDS format is shown below. In this algorithm, *num_tjdiag* stands for the number of transpose jagged diagonals:

```
for $i \leftarrow 1$ to $num\_tjdiag$
    $k := 1$
    for $j \leftarrow start\_position[i]$ to $start\_position[i+1]-1$
        $p := row\_indexes[k]$
        $Y[p] := Y[p] + value\_list[j] \times X[k]$
        $k := k + 1$
    endfor
endfor
```

3 Evaluation of Memory Requirements and Execution Times

We present a selection of sparse matrices from the Matrix-Market collection [2] to evaluate the storage requirements of TJDS compared to JDS. For each matrix we present its name, dimension, N_{nze}, the longest N_{nze} per column and the longest N_{nze} per row as shown in Table. 1 and Table. 2.

We see that, to store the non-zero elements that constitutes the actual data, we also need subsidiary data, which varies for each storage scheme. This subsidiary data includes storing of row/column indices(JDS, TJDS), the permutation vector(JDS), and the number of jagged diagonals(JDS, TJDS). As you can see in the case of JDS, we need an array of size N to store the original permutation. However, TJDS eliminates the need for this array of size N. The subsidiary storage requirements for the JDS, and TJDS formats are calculated using the following equations respectively:

$$STORAGE_{jds} = (N_{nze} \times 1) + N + N_{jd}. \tag{1}$$

$$STORAGE_{tjds} = (N_{nze} \times 1) + N_{tjd}. \tag{2}$$

Where,
N_{nze} denotes the number of non-zero elements of matrix A.

Table 1. Selection of small sparse matrices from Matrix Market.

	Matrix	Dimension	N_{nze}	Longest N_{nzec}	Longest N_{nzer}
1	CRY2500	2500 x 2500	12349	6	5
2	GEMAT11	4929 x 4929	33185	28	27
3	LNS3937	3937 x 3937	25407	13	11
4	SHERMAN5	3312 x 3312	20793	17	21
5	BCSPWR10	5300 x 5300	13571	14	14
6	DW8192	8192 x 8192	41746	8	8
7	DWT_2680	2680 x 2680	13853	19	19
8	LSHP3466	3466 x 3466	13681	7	7
9	BCSSTM25	15439 x 15439	15439	1	1

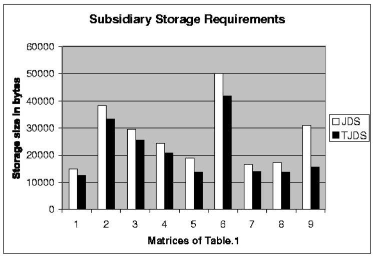
Table 2. Selection of large sparse matrices from Matrix Market.

	Matrix	Dimension	N_{nze}	Longest N_{nzec}	Longest N_{nzer}
1	CRY10000	10000 x 10000	49699	6	5
2	BCSSTK18	11948 x 11948	80519	49	49
3	BCSSTK25	15439 x 15439	133840	59	59
4	MEMPLUS	17758 x 17758	126150	353	353

$(N_{nze} \times 1)$ gives the storage required to store the array indices of the non-zero elements.
N denotes the number of rows and M denotes the number of columns. Since we are considering square matrices, $N = M$. In the case of JDS, N stands for the length of the *perm_vector*.
N_{jd} denotes the number of jagged diagonals.
N_{tjd} denotes the number of transpose jagged diagonals.

In Fig. 1 we show the subsidiary storage required for the selected matrices using the two storage formats. On the other hand, it is worthwhile mentioning that the number of jagged diagonals in any sparse matrix is equal to the maximum number of non-zero elements per row. Likewise, in the case of the TJDS format the number of transpose jagged diagonals is equal to the maximum number of non-zero elements per column. Hence, for symmetric matrices we have $N_{jd} = N_{tjd} = longest\_N_{nzpr} \leq N$.

When we compare TJDS with other widely used storage formats such as Compressed Row Storage(CRS) and Compressed Column Storage(CCS)[6], we still have a substantial subsidiary storage that is saved. These two storage formats need to store the starting positions of either the rows or the columns which is of size N or M. For square matrices, we will have that N equals M and N_{tjd} is generally much smaller than N or M. For example, in the case of the matrix MEMPLUS in Table. 2, $N = M = 17,758$ and $N_{tjd} = 353$.



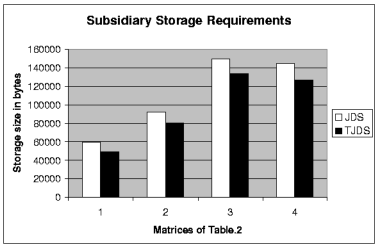
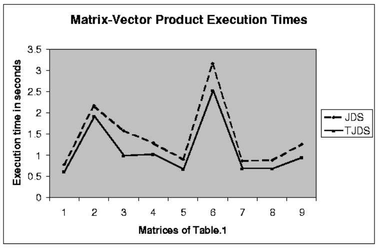


Fig. 1. Subsidiary storage requirements using the JDS and TJDS formats.

The matrix-vector product algorithm for JDS and TJDS are similar in the sense that they both have four load operations and one store operation to compute each partial result. But we see that TJDS outperforms JDS because the permutation step needed in the JDS algorithm is not required for the TJDS algorithm. Fig. 2 gives the execution time for the matrix-vector product using the JDS and TJDS algorithms. The JDS algorithm and TJDS algorithm have been executed sequentially on a AMD T-Bird 900MHz processor with 256MB RAM. The matrix-vector product was executed 1000 times to obtain the timing results.



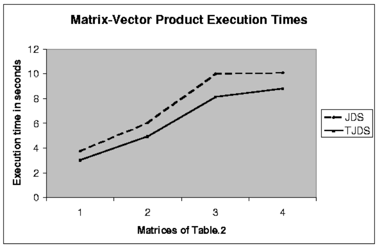


Fig. 2. Matrix-Vector product execution time using the JDS and TJDS algorithms.

4 Conclusions

We have presented a solution to the sparse matrix vector product problem using TJDS, an alternative storage format that requires less storage space than the JDS format. We have also shown that TJDS does not need the permutation vector required by JDS to permute the resulting vector back to the original ordering nor does it need the permutation step to compute the sparse matrix vector product. This new format is suitable for parallel and distributed processing because the data partition scheme inherent to the data structures keeps the locality of reference on the non-zero values of the matrix and the elements of the x array. Currently a parallel implementation of the SMVP using the TJDS format is under way.

Acknowledgements. We would like to thank the anonymous referees for their valuable comments and suggestions to improve the presentation of this paper.

References

1. J. Dongarra, *Sparse Matrix Storage Formats*, In Zhaojun Bai et al, Eds., Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM, Philadelphia, 2000. Electronic version available at: http://www.cs.utk.edu/˜dongarra/etemplates/node372.html.
2. Matrix Market, the electronic version is available at the following URL: http://math.nist.gov/MatrixMarket/.
3. Y. Saad,*Krylov Subspace methods on Supercomputers*, Siam J. Sci. Stat. Comp., vol 10(6), pp. 1200–1232, 1989.
4. Y. Saad, *SPARSKIT : A basic tool kit for sparse matrix computations Report RIACS-90-20*, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.
5. Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS publishing company, ITP, Boston, MA, 1996.
6. E. Montagne and A. Ekambaram, *An Optimal Storage Format for Sparse Matrices*, Dec. 2002(submitted for publication).