Performance Evaluation of Storage Formats for Sparse Matrices in Fortran

Anila Usman[1], Mikel Luján[2], Len Freeman[2], and John R. Gurd[2]

[1] Department of Computer & Information Sciences,
PIEAS (Pakistan Institute of Engineering & Applied Science),
Islamabad, Pakistan
anila@pieas.edu.pk
[2] Centre for Novel Computing, The University of Manchester,
Oxford Road, Manchester M13 9PL, United Kingdom
{mlujan, len.freeman, john.r.gurd}@manchester.ac.uk

Abstract. Many storage formats have been proposed to represent sparse matrices. This paper extends to Fortran 95 the performance evaluation of sparse storage formats in Java presented at ICCS 2005, [7]. These experiments have the same set up (almost 200 sparse matrices and matrix-vector multiplication), but now consider the Fortran 95 Sparse BLAS reference implementation.

Keywords: Sparse matrix, storage format, Sparse BLAS, performance evaluation, JSA (Java Sparse Array).

1 Introduction

Sparse matrices (matrices with a substantial minority of nonzero elements, normally less than 10% nonzero elements) are pervasive in many mathematical and scientific applications. These matrices provide an opportunity to minimise storage and computational requirements by storing, and performing arithmetic with, only the nonzero elements. The many existing storage formats for sparse matrices are derived from different means of taking advantage of sparsity patterns in frequently occurring matrices.

The Basic Linear Algebra Subroutines (BLAS) standard includes for the first time a set of subroutines dedicated to operating with sparse matrices [3]. This part of the standard, hereafter referred to as the *Sparse BLAS*, primarily provides functionality for *iterative methods*. The Sparse BLAS does not state which storage formats must be supported, but ensures that the storage format used is completely transparent to the users. The Sparse BLAS leaves each specific hardware vendor the freedom to select the storage format (or formats) that performs best for its specific platforms.

In ICCS 2005 [7], the authors present a performance evaluation of different storage formats for sparse matrices in Java using the main kernel of iterative methods for linear systems: matrix-vector multiplication. The results show that a recently proposed storage format for Java, namely the Java Sparse Array (JSA)

M. Gerndt and D. Kranzlmüller (Eds.): HPCC 2006, LNCS 4208, pp. 160–169, 2006.
© Springer-Verlag Berlin Heidelberg 2006

[6] performs similarly to, or better than, other long established storage formats. This paper concentrates on the performance of the same matrix operation and same storage formats with the same test matrices, but now their implementations are in Fortran 95 and belong to the reference implementation of the Sparse BLAS [4]. The JSA implementation has been translated to Fortran 95 and it is considered in the experiments.

The outline of the paper is as follows. Section 2 provides a brief description of some commonly used storage formats for sparse matrices and the JSA storage format. The Sparse BLAS reference implementations of the matrix-vector multiplication and other implementations used in the performance evaluation are described in Section 3. The performance evaluation (see Section 4) compares this operation using eight different storage formats. Around two hundred different symmetric/non-symmetric sparse matrices are considered in the performance study. Preliminary conclusions are given in Section 5.

2 Storage Formats for Sparse Matrices

There are many documented versions of different storage formats for sparse matrices. One of the most complete sources is the book by Duff *et al.* [2] (for a historical source see [8]). Some examples of these storage formats follow.

2.1 Compressed Sparse Row/Column Storage Formats (CSR/CSC)

CSR and CSC storage formats are not based on any particular matrix property and hence can be used to store any sparse matrix. In CSR, the nonzero values of every row in the matrix are stored together with their column number, consecutively in two parallel arrays, *Value* and *Col*. There is no particular order with respect to the column number, *Col*. The *Size* and *Pointer* for each row define the number of nonzero elements (nnze) in the row and point to the relative position of the first nonzero element of the row. Fig. 1 presents a sparse matrix stored in CSR.

The column based version, CSC, instead stores *Value* and *Row*, in two parallel arrays. *Size* and *Pointer* of each column allow each member of *Value* to be associated with a column as well as the row given in *Row*.

2.2 Block Entry Storage Formats

Block entry storage formats divide a matrix into blocks or submatrices (squares or rectangles) and define schemes to describe the memory position of a single block. If the block size remains fixed, for example, Block Sparse Row/Column (BSR/BSC) storage format can be obtained from CSR/CSC, respectively. Similarly, when the block size can vary the Variable Block Compressed Sparse Row/Column formats (VBR/VBC) are obtained.

$$A = \begin{pmatrix} & a_{12} & & & \\ & & a_{23} & & a_{25} \\ & & a_{33} & & a_{35} \\ a_{41} & a_{42} & & a_{44} & \\ & & & a_{54} & \end{pmatrix}$$

i	1	2	3	4	5
Size	1	2	2	3	1
Pointer	1	2	4	6	9

	1	2	3	4	5	6	7	8	9
Col	2	3	5	5	3	1	4	2	3
Value	a_{12}	a_{23}	a_{25}	a_{35}	a_{33}	a_{41}	a_{44}	a_{42}	a_{54}

Fig. 1. An example sparse matrix A stored using CSR

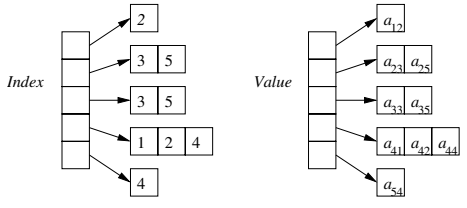

Fig. 2. The sparse matrix A stored using JSA

2.3 Java Sparse Array (JSA)

A recent storage format, designed particularly to suit Java, is JSA (Java Sparse Array), see Gundensen and Steihaug [6] for more details. JSA relies on being able to declare an array with individual elements being arrays – arrays of arrays. JSA is a row oriented storage format, similar to CSR. In Fortran 95, it is implemented as two arrays, each element of which is a pointer to an array. One of these arrays, *Value*, stores pointers to arrays which contain the matrix elements – each row in the matrix has its elements in a separate array. All the separate arrays can be reached through the pointers in the *Value* array; that is an array of pointers to arrays. The second array *Index* stores pointers to arrays which contain the column indices of the matrix, again one array per row. Fig. 2 shows the matrix A stored using JSA.

3 Fortran Implementations of Matrix-Vector Multiply

3.1 Reference Implementation of the Sparse BLAS

The Fortran 95 reference implementation of the Sparse BLAS was developed by CERFACS [4]. Although the standard does not indicate which storage formats must be supported, the Sparse BLAS reference implementation (SBF95) includes

nine different storage formats (Coordinate or COO, CSR, CSC, diagonal or DIA, Block Coordinate or BCO, BSR, BSC, Block Diagonal or BDI and VBR[1] — see [2] for descriptions).

To use sparse matrices with the Sparse BLAS consist of three steps: (1) create a sparse matrix handle, (2) use this handle as a parameter in the Sparse BLAS routines, and (3) free any resources associated with the handle when it is no longer required.

For the first step and third step, the SBF95 uses a linked list to keep track of the different created/freed sparse matrices or handles. The handle is an integer used to access to the linked list which points to internal[2] data types implementing each of the mentioned storage formats. These hold specific information about the created matrix (such as the number of rows and columns, whether it is symmetric, etc.) and pointers to the arrays necessary for each of the different storage formats.

For the second step, the SBF95 transforms the handle into an internal pointer to the sparse matrix representation. Then several checks are performed for the correctness of the parameters (if these checks fail no part of the multiplication is performed and the execution stops immediately). For matrix-vector multiplication, 5 different checks are carried out, of which 4 involve subroutine calls to access data in the internal data types. After these checks, the implementations of matrix-vector multiplication present an if-then-else code structure separating special cases, where for example, the matrix is symmetric and only the elements in the upper triangular region are stored, from the general case. The actual implementation for each of the cases is part of the same subroutine; i.e. after the 5 checks no further subroutine call is made.

By default the SBF95 creates matrices in COO. Using internal utilities subroutines, users can transform from COO to the other storage formats.

3.2 Fortran Implementation of Java Sparse Array

The implementation of JSA is a separate and self-contained Fortran 95 module which defines the storage format as a data type. The data type holds the same information as the internal data types in the SBF95. The module has subroutines to allocate and assign the elements of a matrix, perform the matrix-vector multiplication and deallocate the memory used by a matrix.

Comparing the implementation of matrix-vector multiplication in JSA with those in the SBF95, the same checks and the same if-then-else structure are present. However the checks do not involve subroutine calls, and there is no handle to be translated into an internal representation.

4 Performance Evaluation

Matrix test data consists of 182 real, sparse symmetric and non-symmetric matrices available to download from the Matrix Market Collection [1] covering both

[1] The storage formats BDI and VBR are not used in the performance evaluation.
[2] Hereafter the word internal also means not part of the Sparse BLAS standard.

Table 1. Legend for the storage formats axis in Fig. 3

1	COO		7	BSC block size 2		13	BSC block size 8
2	CSR		8	BCO block size 4		14	BCO block size 16
3	CSC		9	BSR block size 4		15	BSR block size 16
4	DIA		10	BSC block size 4		16	BSC block size 16
5	BCO block size 2		11	BCO block size 8			
6	BSR block size 2		12	BSR block size 8			

systems of linear equations and eigenvalue problems, and representing many fields of Computational Science & Engineering.

The test program reads a matrix from file, multiplies a random vector by that matrix and records both the result vector and the time taken to calculate the result vector. The test program progresses in this way through the different implementations of the matrix operation. The test program checks the output results and we note that all the experiments produce the correct results.

The test machines are the same as those used in [7] to allow direct comparison. However, due to space limitations only results for one of the machines are reported; an Ultra Sparc 10 running Solaris and Sun's Fortran 95 compiler with optimisation flag `-fast`. In order to obtain meaningful times, the experiments report the execution times for each matrix-vector multiplication repeated 50 times. This not only gives timing results that are large enough compared with the accuracy of the timers (milliseconds), but is also a realistic simulation of the sequence of matrix-vector multiplications that dominates iterative methods in numerical linear algebra. The complete test program is run 10 times and the average of these times are reported.

4.1 SBF95 Performance Results

Fig. 3 gives the results of 8 different storage formats included in SBF95 for all matrices. Square block sizes between 2 and 16 are considered. No distinction is made between eigenvalue problems and systems of linear equations, nor between symmetric and non-symmetric matrices. The matrices in the Matrix Number axis are ordered by increasing nnze (the total number of non-zero elements).

Comparing these results with the results for the other machines not reported here, all the storage formats follow the same general pattern. For all of the storage formats there is a correlation between nnze and the observed execution times.

For the smaller problem sizes the block entry storage formats do not perform significantly differently for different block sizes. However, for the largest problems, the block entry storage formats, with block sizes of 4 and 8, perform better than the block formats with other block sizes. This suggests that there is a compromise between the gain resulting from more efficient use of the cache and the loss due to increases in the number of zero elements that are stored as the block sizes increase. For most of the test matrices, the point entry storage formats COO, CSR, CSC and DIA perform better than the block storage formats.

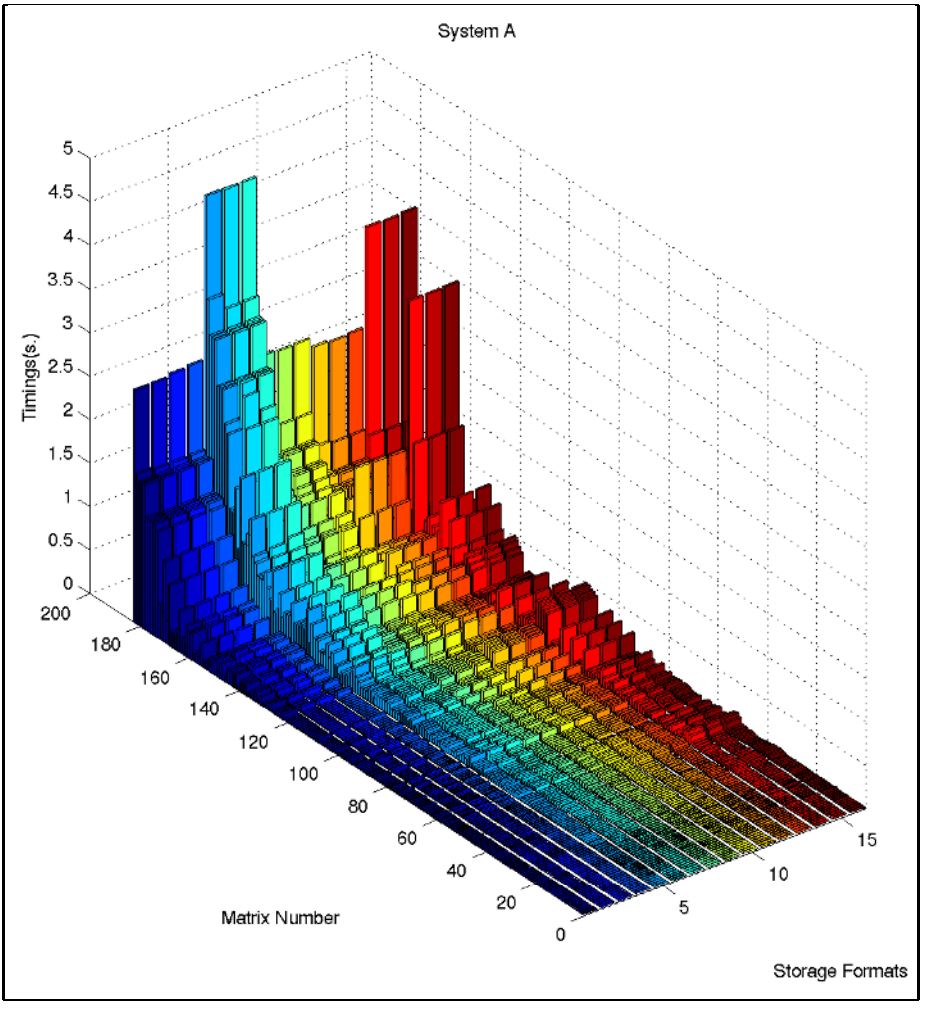

Fig. 3. Time results (seconds) for all matrices and storage formats. Table 1 presents the legend for the storage formats axis.

Fig. 4 takes a closer look at the results of CSR (as a typical representative of COO, CSC and DIA) and BSR with block sizes 4 and 8 (the fastest results among the block entry storage formats). The results are divided into two graphs (the upper graph covering matrices 1 to 140 and the lower graph covering matrices 141 to 182). For a few of the matrices with large numbers of nonzero elements (from matrix 173), the performance for BSR with block sizes of 4 and 8 is better than the performance of the point entry storage formats COO, CSR, CSC and DIA.

Performance Results for JSA

Fig. 5 presents the results of JSA and of the SBF95 implementation of CSR, referred to as CSR(SBF95) on the figure.

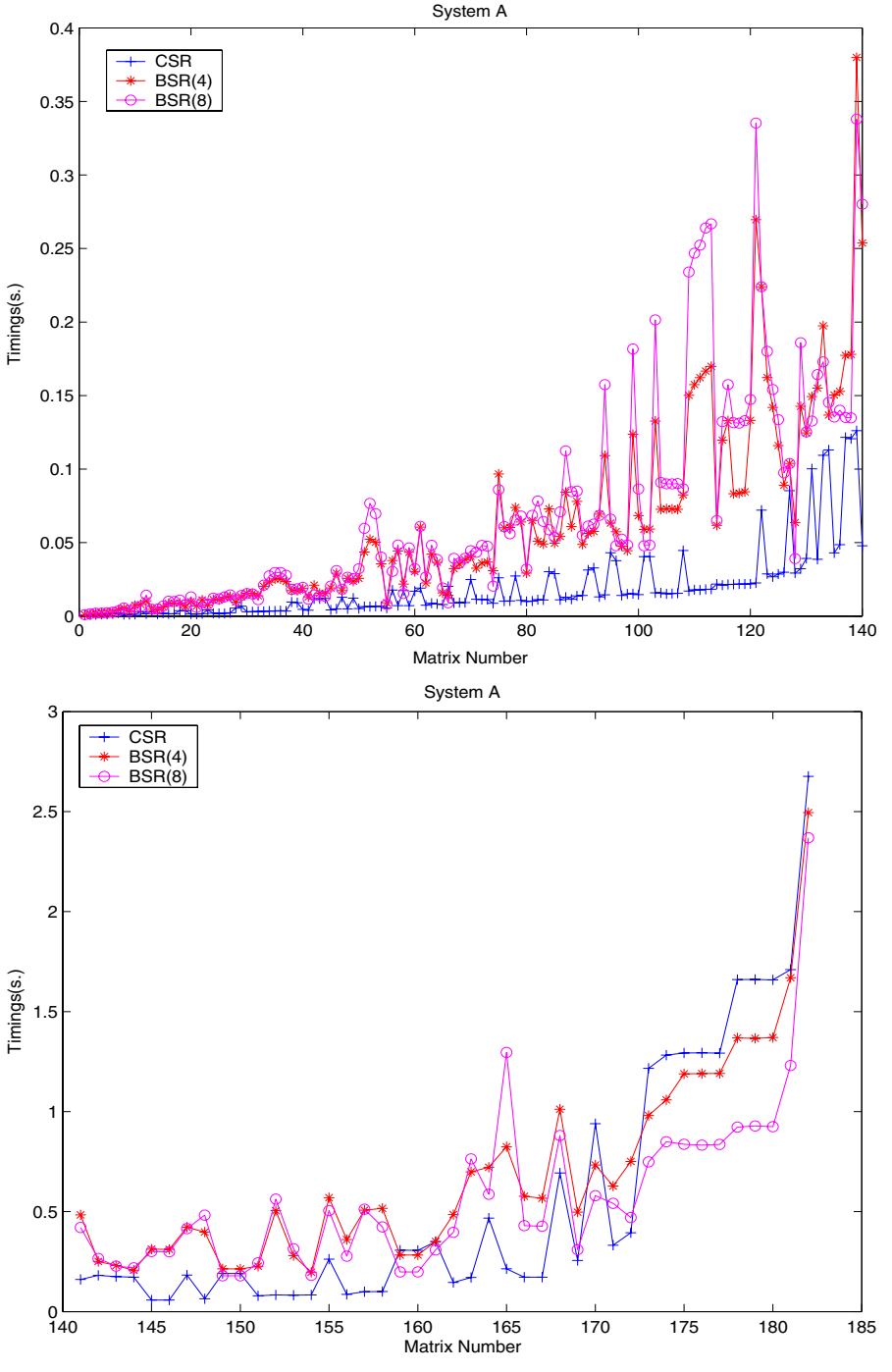

Fig. 4. Time results (seconds) for some selected storage formats

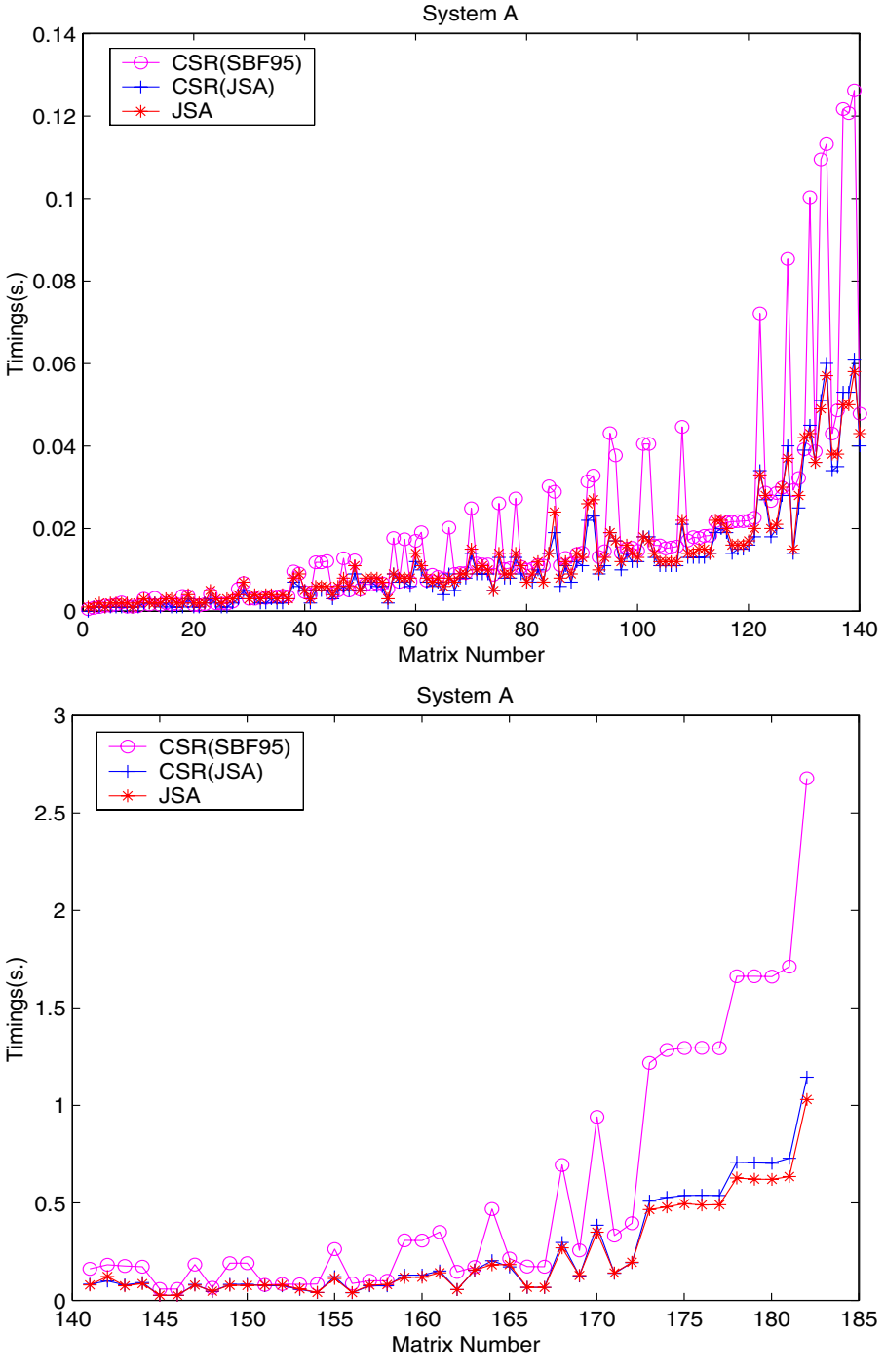

Fig. 5. Time results (seconds) for JSA

For all matrices JSA is comparable to, or significantly more efficient than, CSR(SBF95). Note that CSR(SBF95) is the fastest storage format among the SBF95 implementations for matrix 1 to matrix 173 (see Fig. 4). For the remaining matrices (matrix 174 to matrix 192), the fastest storage format among the SBF95 implementations is BSR with a block size of 8. Nonetheless, for these matrices JSA is approximately twice as efficient as BSR with a block size of 8.

The results of Fig. 5 give a clear indication of the advantage of JSA over the other storage formats. However, although the implementation of the matrix-vector multiplication for JSA and the SBF95 are similar, there are significant differences (see Section 3): (1) there is no need for the handle and the linked list translation or indirection in JSA; and (2) there are four checks, without subroutine calls in JSA, but with subroutine calls in the SBF95. Given that the code is executed 50 times, these differences could be the source of the performance advantage of JSA. To test this hypothesis, a CSR implementation following exactly the same style as the JSA implementation (no handle, no subroutine calls in the four checks) is developed and Fig. 5 presents these results as CSR(JSA). Comparing JSA and CSR(JSA), the results are indistinguishable up to matrix 177. For the remaining matrices the difference in performance is relatively small, although it favours the JSA implementation.

Taking this addition experiment into consideration, the conclusion remains that JSA is as good as any of the point entry storage formats, with a small advantage for JSA for those matrices with larger numbers of nonzero elements (matrices 177 to 182).

5 Conclusions

It would be presumptuous to say that all the storage formats for sparse matrices are covered by this work. Especially since there are many minor variations which can create entirely new storage formats. Nonetheless, this paper has presented a comprehensive performance comparison of storage formats for sparse matrices. The results have shown that point entry storage formats perform better than block entry storage formats for most matrices. For a set of around 30 of the matrices with the larger numbers of nonzero elements, BSR with block size 8 has performed better than the other block entry and point entry storage formats. The performance evaluation has shown that JSA delivers performance similar to, or better than, existing storage formats. The performance evaluation has also identified a performance problem with the SBF95 (Sparse BLAS reference implementation). This performance problem can be addressed by favouring common execution paths — check whether the last call to a subroutine was made with the same handle as the current parameter and accordingly eliminate checks and translation to internal data structures. The most relevant related work is the Sparsity project [5]. For a given sparse matrix the Sparsity project has developed compile time techniques to optimise automatically several sparse matrix kernels using a specific block entry storage format and selecting an optimal block size.

References

1. The matrix market. http://math.nist.gov/MatrixMarket/.
2. I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.
3. I. S. Duff, M. A. Heroux, and R. Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software*, 28(2):239–267, 2002.
4. I. S. Duff and C. Vömel. Algorithm 818: A reference model implementation of the Sparse BLAS in Fortran 95. *ACM Transactions on Mathematical Software*, 28(2):268–283, 2002.
5. Eun-Jin, K. A. Yelick, and R. Vuduc. SPARSITY: An optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.
6. G. Gundersen and T. Steihaug. Data structures in Java for matrix computations. *Concurrency and Computation: Practice and Experience*, 16(8):799–815, 2004.
7. M. Luján, A. Usman, P. Hardie, T. L. Freeman, and J. R. Gurd. Storage formats for sparse matrices in Java. In *Proceedings of the 5th International Conference on Computational Science – ICCS 2005, Part I*, volume 3514 of *Lecture Notes in Computer Science*, pages 364–371. Springer-Verlag, 2005.
8. U. W. Pooch and A. Nieder. A survey of indexing techniques for sparse matrices. *ACM Computing Surveys*, 5(2):109–133, 1973.