# Direct and transposed sparse matrix-vector multiplication

**Article**

CITATIONS
3

READS
744

**2 authors**, including:

Sorin Cotofana
Delft University of Technology
**338** PUBLICATIONS **2,263** CITATIONS

SEE PROFILE

# Direct and Transposed Sparse Matrix-Vector Multiplication

Sorin Cotofana
*Computer Engineering Laboratory*
*Delft University of Technology*
sorin@dutepp0.et.tudelft.nl

Pyrrhos Stathis
*Computer Engineering Laboratory*
*Delft University of Technology*
pyrrhos@dutepp0.et.tudelft.nl

Stamatis Vassiliadis
*Computer Engineering Laboratory*
*Delft University of Technology*
stamatis@dutepp0.et.tudelft.nl

***Abstract***

*In this paper we investigate the execution of $Ab$ and $A^Tb$, where $A$ is a sparse matrix and $b$ a dense vector, using the Blocked Based Compression Storage (BBCS) scheme and an Augmented Vector Architecture (AVA). In particular, we demonstrate that by using the BBCS format, we can represent both the direct and the transposed matrix for the purposes of matrix-vector multiplication with no additional costs in storage, access time and computation performance. To achieve this, we propose a new instruction and a hardware modification for the AVA. Subsequently we evaluate the performance of the transposed Sparse Matrix Vector Multiplication (SMVM) and demonstrate that like for the direct SMVM, the BBCS scheme outperforms other general schemes like the Jagged Diagonal (JD) and the Compressed Row Storage (CRS) by 1.7 to 4.1 times. Furthermore we show that the BBCS scheme outperforms CRS and JD when the aforementioned SMVM is used in the Conjugate Gradient and Bi-Conjugate Gradient iterative solve algorithms for which speedups of 1.78 to 4.13 depending were achieved in simulations.*

## 1. Introduction

A significant part of many scientific computing applications consists of sparse matrix operations. Research efforts have focused on improving the efficiency of such operations since irregular sparsity patterns can be detrimental for the performance. This holds especially for data parallel machines that benefit from large regular structures. Both software [5, 6] and hardware [1, 8, 10] approaches have have been presented in literature to address this inefficiency problem. In this line of work, recently, a scheme involving a Blocked Base Compression Storage (BBCS) format on an Augmented Vector Architecture (AVA) was proposed in [9] to alleviate the performance degradation in sparse matrix vector multiplication, one of the most important kernels in sparse matrix computations, and a speedup of up to 3 times has been achieved when compared to the Jagged Diagonal scheme. However, many applications also involve the multiplication of a vector with the transposed of a matrix ($A^Tb$). For instance, algorithms such as some Iterative Solvers involve multiplications with both the direct and the transposed matrix. These algorithms are usually avoided in favor of others that do not involve multiplications with the transposed matrix. The reason is that most general matrix formats are non-symmetrical and therefore favor either the direct or the transposed multiplication but not both.[1] With this as an incentive, in this paper we investigate the possibility of providing both operations in an efficient manner for the aforementioned AVA using the BBCS scheme. The contributions of this paper can be summarized as follows:

- We demonstrate that by using the BBCS format we can represent both the direct and transposed sparse matrix for the purposes of Sparse Matrix Vector Multiplication (SMVM) with no costs in storage, access time and computation performance.
- In order to achieve the aforementioned result we propose and describe a new instruction and a functional unit modification to the augmented vector architecture.
- We evaluate the performance of the transposed SMVM and show that like with the direct SMVM the BBCS

[1] The transposed matrix vector multiplication ($A^Tb$) should be regarded separately from the transposition of the matrix itself. When performing $A^Tb$ the matrix $A$ does not need to be stored as transposed. By changing the access patterns to the elements the same effect can be achieved.

scheme outperforms other general schemes like the Jagged Diagonal (JD) and the Compressed Row Storage (CRS). In particular, speedups of 1.7 to 4.1 times were achieved for both direct and transposed SMVM versus the JD and CRS schemes.

- Finally, we evaluate the performance of the BBCS scheme versus the JD and CRS on the execution of the Conjugate gradient and Bi-Conjugate Gradient Iterative Solve Methods and show speedups varying from 1.78 to 4.13 versus the JD and CRS schemes.

The remainder of this paper is organized as follows: In Section 2 a number of sparse matrix storage formats including the BBCS format are discussed. In Section 3 we describe the transposed SMVM using the BBCS scheme on the AVA. In Section 4 we present the results of the evaluation of the BBCS scheme for direct and transposed SMVM. Finally, in Section 5 we draw some conclusions.

## 2. Sparse Matrix Storage Formats: Background

In this Section we provide with some background information on the Jagged Diagonal (JD), Compressed Row Storage (CRS) and Block Based Compression Storage (BBCS) general sparse matrix storage formats.

A matrix is called sparse when it contains a small amount of non-zero elements. In order to avoid operating on, storing and transferring the large amount of zeros, compressed formats are routinely used to manipulate the sparse matrices. A multitude of different storage formats exist [5, 6], many of which take advantage of specific properties of certain types of matrices. In this paper however we will focus on general sparse matrix storage formats, that is, formats that are not tuned on specific types of sparse matrices. Such are the Compressed Row Storage (CRS), the Jagged Diagonal (JD) and the Blocked Based Compression Storage (BBCS). In CRS (depicted in Figure 1 left) the non-zero matrix elements are stored in a row-wise fashion in an array. The corresponding column positions of those elements are stored in a second array. Finally, the positions in the aforementioned arrays of each of the first non-zero elements in each row are stored in a third array. To store a matrix in the JD format we proceed as follows (see Figure 1 right): First all the non-zero matrix elements are shifted to the left, forming a set of column vectors. The corresponding original column positions are stored in a second set of column vectors. The column vectors are now permuted to form dense vectors of decreasing length. Finally the permutation vector and the column vector lengths are stored. Of the two presented general storage formats the CRS storage is the most commonly used storage format since it is simple and straightforward to

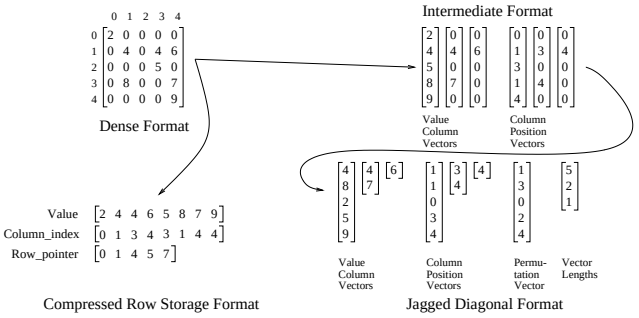



**Figure 1. Compressed Row Storage (CRS) and Jagged Diagonal (JD) Formats**

use and offers fairly good performance. The main disadvantage though of the CRS format for vector processing is that when a matrix contains only few non-zero elements per row the vector instructions need to operate on vectors with short vector lengths. This is detrimental for the performance of a vector processor due to vector startup overhead costs. The JD format does not suffer from this problem and is therefore preferred for operations like matrix vector multiplication on vector processors especially for matrices with only few elements per row. However, for matrices that contain few rows with many non-zeros the JD too suffers from short vector forming and therefore startup overhead on a vector processor. Furthermore, both CRS and JD formats need a storage space which is more than two times the storage space needed to store the values of the non-zero elements alone. This is due to the need to store the corresponding original column position of the non-zero elements.

The BBCS format which is described in more detail in [9] addresses the aforementioned problems of JD and CRS. The BBCS format is designed to be supported by an architectural extension to a conventional vector architecture which we will call an Augmented Vector Architecture (AVA). This architectural extension will be described in Section 3 and consists of two new vector instructions and a specialized functional unit that enable the AVA to read and operate on the BBCS format. To obtain the BBCS format from a $n \times n$ matrix $A$ we first partition the matrix into $\lceil \frac{n}{s} \rceil$ vertical blocks (VBs) $A_k, k = 0, 1, \ldots, \lceil \frac{n}{s} \rceil - 1$, of at most $s$ columns wide, where $s$ is the section size[2] (or maximum vector length) of the vector architecture (see Figure 2). Subsequently the values of the non-zero elements within each VB are stored in a row-wise fashion. Along with the non-zero values the column position of each element within the VB is also stored. Furthermore for each non-zero element a bit is stored to indicate whether the ele-

[2]The section size of a vector architecture is the maximum number of elements that can be processed at one time by a vector instruction.

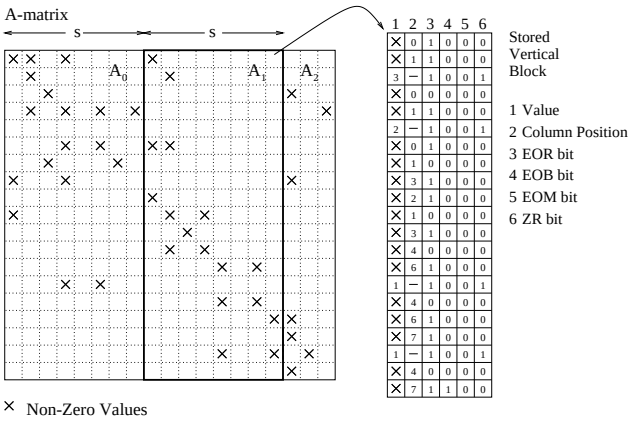



**Figure 2. Blocked Based Compression Storage (BBCS) Format**

ment is the last in its row (End Of Row (EOR) bit). In the same fashion a bit is stored to indicate the end of the VB (EOB bit) and a bit to indicate end the matrix (EOM bit). Finally, a Zero Row (ZR) bit indicates that the entry (the value, column position and the EOR, EOB, EOM and ZR bits) represents the existence of one or more empty rows within the VB rather than a non-zero element. In the case that the ZR bit is set the value will denote the number of empty rows. Figure 2 graphically depicts the storing of the second VB, $A_1$, of a matrix in BBCS format. Using this method of storing the sparse matrix we can achieve lower storage needs compared to CRS or JD because the column positions of the non-zero elements can be stored using only $\log_2 s$ per non-zero per entry since the column position is within the VB ($s$ elements wide) rather than the entire matrix. Furthermore, $s$ has been selected as the width of the VB to optimize the performance of operation on this format when using a vector processor with the same section size.The BBCS format is supported by an architectural extension which is described in the next Section. The extension enables the architecture to read the BBCS format and to operate on several row per vector instruction in this way eliminating the vector startup overhead costs. It has been shown in [7] that for the sparse matrix vector multiplication the BBCS method outperforms the JD sparse matrix vector multiplication up to three times depending on the sparsity pattern. This is due to the reduced bandwidth needed for the BBCS scheme and the reduction of overhead costs induced by short vectors. However, besides direct SMVM, often the transposed SMVM of a matrix is needed in applications. Such an application is the Bi-Conjugate Gradient Iterative Solving Algorithm [4] that requires one normal and one transposed matrix vector multiplication at each iteration. This is presented in literature [4] as a disadvantage of the algorithm. This relates to the fact that storage formats such as CRS and JD either perform worse for the Transpose SMVM or it is not possible at all to perform the transposed SMVM (e.g. JD for vector code). In the latter case we need to use a second copy of the matrix which is stored as the transposed of the matrix. Obviously, this requires twice the space in memory and twice the bandwidth since the matrix elements need to be loaded twice during execution, once for the SMVM and once for the transposed SMVM. In the next section we will demonstrate that we can use the BBCS format for both the transposed and direct SMVM.

## 3. BBCS transpose Sparse Matrix Vector Multiplication

In this section we present the transposed SMVM using the BBCS format on an AVA. To proceed we will first give a brief presentation of the direct SMVM scheme. For further details on the multiplication using BBCS see [9].

As was mentioned in the previous section, the SMVM for the BBCS scheme is supported by an architectural extension to a traditional vector architecture, which we refer to as an Augmented Vector Architecture (AVA). The extension includes two new instructions a vector Functional Unit and a special vector register. The first instruction, LoaD Section (`LDS`), loads the matrix elements stored in BBCS format into the processor and the second instruction, Multiple Inner Product and Accumulate (`MIPA`), performs the multiplication using the `MIPA` functional unit. More specifically, suppose we want to multiply a sparse matrix $A$ by a dense vector $b$. The matrix $A$ is stored in the BBCS format as a sequence of Vertical Blocks of width $s$ where $s$ is the section size (or maximum vector register size). `LDS` loads a portion (up to $s$ elements) of the VB's non-zero elements into a vector register VR1. At the same time the corresponding column positions of each element are loaded in the special vector register called the CPR (the Column Position Register). A Bit Vector (BV) is also created containing ones at the positions where the corresponding non-zero elements are the last in their rows. Finally, using the positional information implicitly contained in the BBCS format (EOB bits and ZR entries), `LDS` creates an index vector that contains the non-empty row positions in the Vertical block and therefore the positions of the vector elements that will result from the multiplication. After the execution of the `LDS` instruction, the `MIPA` instruction can be executed to calculate the inner products of the rows of the VB with their corresponding values of vector $b$ using the `MIPA` functional unit. The data flow of the `MIPA` functional unit is depicted in Figure 3 (left). The vector code to perform the multiplication of one VB with the corresponding section of the $b$ vector is as follows (some code details have been omitted for simplicity):

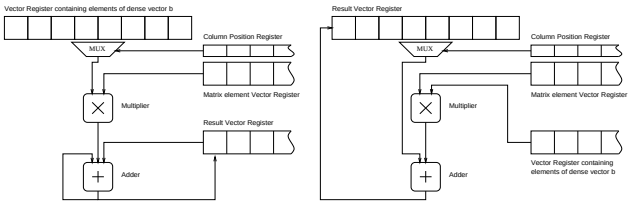



**Figure 3. Functional difference of MIPA and MIPAT**

```
Standard Matrix vector Multiplication Code
for BBCS
  LV  VR2, b         ; load section of b in
                     ; vector register VR2
l LDS A, VR1, VR4    ; non-zero elements
                     ; are loaded in VR1 and
                     ; index vector  in VR4
  LVI VR3, c(VR4)    ; c elements in VR3 from
                     ; address c, VR4 is index
  MIPA VR3, VR1, VR2; multiple inner product
  SVI VR3, c(VR4)    ; store c elements VR2 to
                     ; address c, VR4 is index
Repeat l: for entire VB
```

Note that the LDS instruction sets the column position in the special vector register CPR. During execution of the MIPA instruction the CPR index values are used to select the correct values from the $b$ vector residing in vector register VR2 as indicated in Figure 3 (left).

To implement the transpose SMVM we have to consider the fact the transpose $A^T$ of a matrix $A$ is essentially the same matrix with the rows and columns exchanged. Therefore, for the transpose SMVM we can use the elements of $A$ stored in the same order as in the direct SMVM and change all the row related accesses to column related accesses and vice versa. This technique needs to be performed both at the algorithm level and at the level of the working of the MIPA functional unit. The algorithm to perform the transpose SMVM becomes as follows:

```
Transpose Matrix Vector Multiplication Code
for BBCS
  SUB VR3, VR3, VR3  ; initialize result
                     ; VR3 = 0,0,0,...
l LDS A, VR1, VR4    ; non-zero elements
                     ; are loaded in VR1 and
                     ; index vector in VR4
  LVI VR2, b(VR4)    ; b elements in VR2 from
                     ; address b, VR4 is index
  MIPAT VR3, VR1, VR2; multiple inner product
                     ; transposed
Repeat l: for entire VB
  SV  VR3, c         ; store result VR3
                     ; starting from adress c
```

We observe that the access patterns involving vector $b$ and $c$ (the result) have been almost exchanged. As indicated above the working of the MIPA instruction has to be altered and therefore we have introduced a new instruction, MIPAT (MIPA Transposed), similar to MIPA to support the transposed SMVM. Due to the exchange of rows and columns, MIPAT has a different behavior than MIPA as depicted in Figure 3. However, despite the functionality difference, the MIPA and MIPAT can be executed using the same functional unit (FU) if the data flow of the FU is slightly reconfigured depending on the instruction which is executed. In Figure 3(right) the differing data flows required by the MIPA and MIPAT is depicted. Thus, we can implement support for the transposed SMVM on the AVA using the same storage format by only reconfiguring the data flow of the Functional unit of the AVA.

## 4. Evaluation

In this section we will present an evaluation of the direct and transposed SMVM that was described in the previous Section using the BBCS scheme. First we will evaluate the direct and transposed SMVM as stand-alone operations and subsequently within the Conjugate Gradient (CG) and Bi-Conjugate Gradient (BiCG) iterative linear system solving algorithms.

All evaluations that will be presented have been conducted on a augmented vector architecture simulator. This simulator has been based on the Simplescalar simulator [3], a scalar processor simulator. In order to evaluate the BBCS scheme, the Simplescalar simulator was extended to support traditional vector instructions as well as the LDS, MIPA and MIPAT vector instructions which were described in the previous section. The vector processor model uses 2 vector functional units (including functionality for the MIPA and MIPAT instructions) with a startup overhead of 8 cycles and a functional unit parallelism of 4. The section size (or maximum vector register size) used was $s = 64$. For the memory interface we have used the cache memory interface provided by the original Simplescalar simulator. All the benchmark programs were hand coded in vector assembly. The BBCS scheme was evaluated versus the JD and CRS schemes since they are, to the best of our knowledge, the most commonly used general sparse matrix formats. For the evaluation we have used a number of benchmark matrices that were chosen from the Matrix Market online sparse matrix collection [2].

The matrices, depicted in table 1 were chosen based on their pattern and dimension diversity rather than their suitability for the CG and BiCG linear system solvers in order to cover a wide variety of different matrix types. We have divided the matrices into two main categories: large (cavity16, memplus, mbeause) and small (gre_185, fs_183_3,

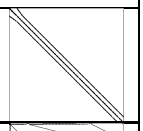
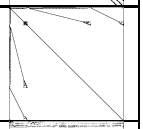
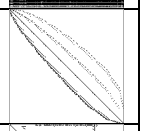
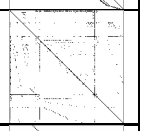
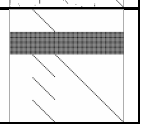
| Matrix Name | Dimensions of Matrix | Number of non-zeros | Average NZPR | NZPR Vari-ance | longest NZPR | Non-zero Structure |
|---|---|---|---|---|---|---|
| cavity16 | 4562 x 4562 | 138187 | 30 | 15 | 62 |  |
| memplus | 17758 x 17758 | 126150 | 5.6 | 13 | 353 |  |
| mbeause | 496 x 496 | 41063 | 83 | 130 | 489 |  |
| gre__185 | 185 x 185 | 1005 | 5.3 | 0.85 | 6 |  |
| fs_183_3 | 183 x 183 | 1069 | 5.8 | 9.1 | 72 |  |
| tols90 | 90 x 90 | 1746 | 19 | 35 | 90 |  |

**Table 1. Sparse Matrix Test Suite (NZPR = Non-Zeros Per Row)**

tols90). As can be observed in Figure 1 we have focused on the variety of the statistical properties of the number of Non-Zeros Per Row (NZPR). As indicated in Section 2 these properties influence the efficiency of the SMVM.

### 4.1. Evaluation of Direct and Transposed SMVM

The evaluation of the performance of the various schemes for the direct and transposed SMVM is depicted in Figure 4 (the prefix `T-` denotes the transposed SMVM rather than the direct). The transposed SMVM for the JD scheme is omitted since the format does not permit a vector algorithm for the Transposed SMVM. We observe that for all matrices the BBCS scheme outperforms the JD and CRS schemes for both SMVM and transposed SMVM. In particular, the performance speedup varies from 1.42 to 4.1. Furthermore, we notice that for the BBCS scheme the execution of the transposed SMVM is faster than the SMVM. The reason for this can be best illustrated by examining the algorithms used for SMVM and transposed SMVM for one vertical block that were presented in Section 3. When we compare the codes we see that the indexed vector store (`SVI` instruction) which is within the loop in the direct SMVM case, becomes a regular (access stride one) vector store outside the loop in the transposed case. This results, depending on the sparsity pattern, in a lower number of stores and therefore in a performance benefit. The aforementioned difference in performance between the direct and transposed SMVM for the BBCS method provides us with the option to influence the performance of an application in the cases that we know a priori the percentage of direct and transposed SMVM operations involved. This is based on the matrix property $(A^T)^T = A$. Therefore if we create the BBCS format in such a way that the stored matrix represents $A^T$, the transposed SMVM algorithm will effectively perform the direct SMVM and vice versa. Using this technique we can optimize the performance if have information about the ratio of direct and transposed SMVM operations. Obviously when the the occurrences of direct and transposed SMVM operations are equal, as is the case in the case of the Bi-Conjugate Gradient which is discussed in the following subsection, no preference needs to be made. However, for the Conjugate Gradient that only contains one type of SMVM we can chose the one that executes faster which in the CG case will be to store the matrix as $A^T$ and perform the transposed SMVM.

### 4.2. Evaluation of CG and BiCG

In the previous subsection we demonstrated the superior performance of the BBCS scheme for both direct and transposed SMVM when compared to the CRS and JD

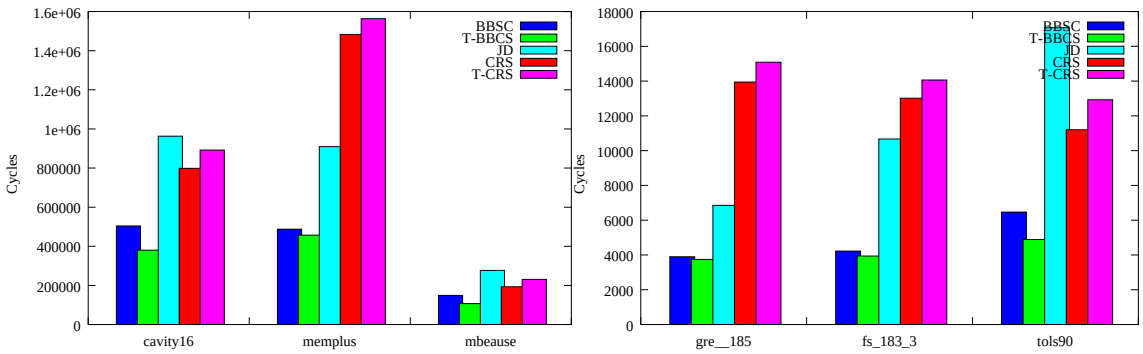



**Figure 4. Performance of various schemes executing direct and transposed SMVM**

Compute $r_0 = b - Ax_0$ using initial guess $x_0$
$\tilde{r}_0 = r_0$
**for** $i = 1, 2, 3, \ldots$
  **if** $i = 1$
    $p_i = z_{i-1}$
    $\tilde{p}_i = \tilde{z}_{i-1}$
  **else**
$\Rightarrow$ $p_i = z_{i-1} + \beta p_{i-1}$
$\Rightarrow$ $\tilde{p}_i = \tilde{z}_{i-1} + \beta_{i-1}\tilde{p}_{i-1}$
  **endif**
$\Rightarrow$ $q_i = Ap_i$
$\Rightarrow$ $\tilde{q}_i = A^T\tilde{p}_i$
$\Rightarrow$ $\alpha_i = \rho_{i-1}/\tilde{p}_i^T q_i$
$\Rightarrow$ $x_i = x_{i-1} + \alpha_i p_i$
$\Rightarrow$ $r_i = r_{i-1} + \alpha_i q_i$
$\Rightarrow$ $\tilde{r}_i = \tilde{r}_{i-1} + \alpha_i \tilde{q}_i$
  check convergence; continue if necessary
**end for**

**Figure 5. The Non-preconditioned (Bi)Conjugate Gradient Iterative Algorithm**

schemes. However, to assess the overall impact of the BBCS scheme's advantage it is necessary to evaluate the performance within the context of full applications. We have chosen to use the Conjugate Gradient (CG) and Bi-Conjugate Gradient non-stationary iterative methods as described in [4]. CG and BiCG are iterative algorithms used to solve linear systems with sparse matrices on vector processors. The non-preconditioned version of the BiCG algorithm is depicted in Figure 4.2 where $x$, $p$, $z$, $q$, $\tilde{p}$, $\tilde{z}$ and $\tilde{q}$ denote dense vectors and Greek letters denote scalars. The CG algorithm is essentially the same as BiCG but with lines containing $\tilde{p}$, $\tilde{z}$, $\tilde{q}$ removed. These are all the operations related to the transposed SMVM, namely $\tilde{q}_i = A^T\tilde{p}_i$.

We will focus our attention on the asymptotic performance of the iteration loop and therefore only the lines indicated with an arrow $\Rightarrow$ in Figure 4.2 will concern us. Consequently, the main time consuming operations in the iteration loops for CG and BiCG are the following:

| Conjugate Gradient |
|---|
| 3 operations of the form $x = x + \alpha y$ (SAXPY) |
| 1 inner product |
| 1 direct matrix vector multiplication |

| Bi-Conjugate Gradient |
|---|
| 5 SAXPY operations |
| 1 inner product |
| 1 direct matrix vector multiplication |
| 1 transposed matrix vector multiplication |

The SAXPY and inner product operations will be implemented identically for the BBCS on the AVA and for JD and CRS on a traditional vector architecture since dense operations ($x$, $p$, $z$, $q$, $\tilde{p}$, $\tilde{z}$ and $\tilde{q}$ are dense vectors) have no additional support provided by the AVA. Therefore the performance benefit of the BBCS scheme for direct and transposed SMVM will also benefit CG and BiCG if the bulk of the computations is spend in the SMVMs.

We implemented each algorithm on the vector processor simulator for each of the BBCS, JD and CRS schemes. The execution of the CG and BiCG algorithms yielded the results which are depicted in Figure 6 and Figure 7 respectively. Note that, as we mentioned in Section 2 in the case of the JD scheme we cannot perform the transposed SMVM using vector code and therefore a second copy of the matrix was used which is stored as the $A^T$. Consequently, due to increased bandwidth required to load the matrix twice, the performance of the JD method for BiCG is degraded. For the CRS and BBCS schemes however the matrix $A$ can be loaded once and serve for both $A$ and $A^T$. From Figure 7 we observe that the impact of the superior performance of the BBCS scheme for direct and transpose SMVM is clearly visible when these operations are incorporated in the above algorithms. In particular for CG a speedup of 1.61 to 3.06 over the JD method and 1.78 to 2.75 over CRS is achieved. For BiCG a speedup of 2.18 to 4.13 over the JD method

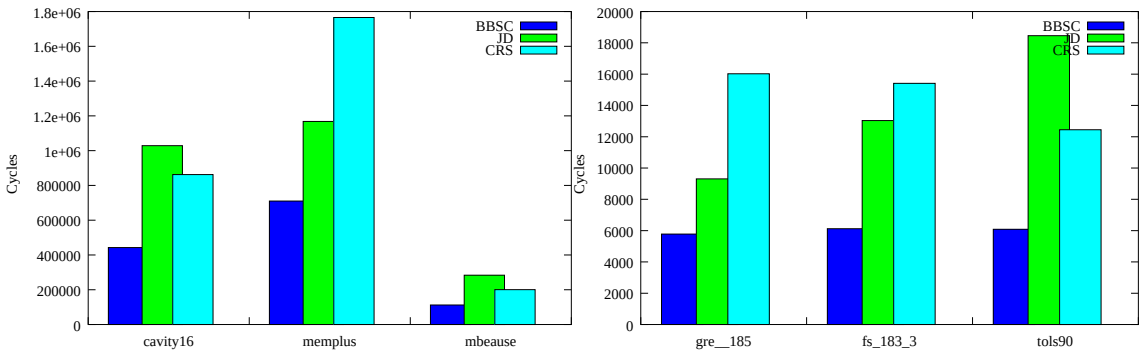



**Figure 6. Conjugate Gradient performance for the BBCS, JD and CRS schemes**

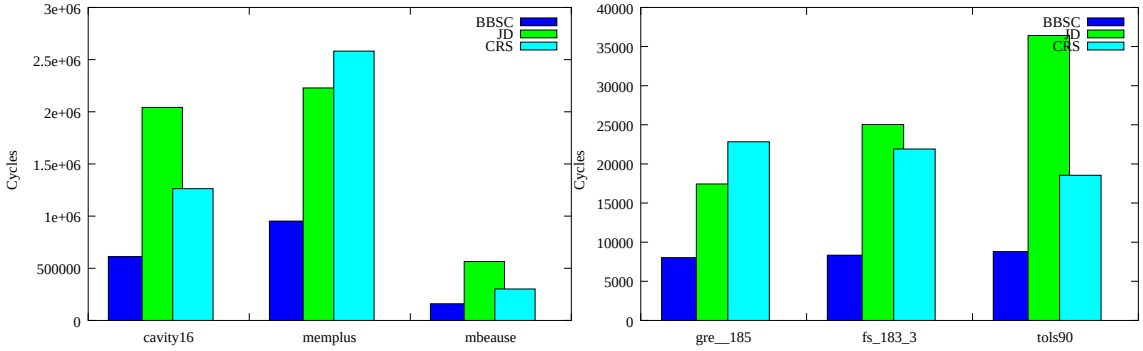



**Figure 7. Bi-Conjugate Gradient performance for the BBCS, JD and CRS schemes**

and 1.89 to 2.85 over CRS is achieved. Notice that due to the aforementioned shortcoming of the JD scheme for the transposed SMVM that the speedups obtained versus the JD scheme are larger than the speedups mentioned earlier for the SMVM.

## 5. Conclusions

In this paper we investigated the performance of the transposed Sparse Matrix Vector Multiplication (SMVM) using the Block Based Compression Storage (BBCS) scheme. In particular, we demonstrated that by using the BBCS format we can represent both the direct and transposed sparse matrix for the purposes of sparse matrix vector multiplication with no costs in storage, access time and computation performance. In order to achieve the aforementioned result we proposed and described a new instruction and a functional unit modification to the augmented vector architecture. Consequently, we evaluated the performance of the transpose SMVM and showed that like with the direct SMVM the BBCS scheme outperforms other general schemes like the Jagged Diagonal (JD) and the Compressed Row Storage (CRS). In particular, speedups of 1.7 to 4.1 times were achieved for both direct and transposed SMVM versus the JD and CRS schemes. Finally, we evaluated the performance of the BBCS scheme versus the JD and CRS on the execution of the Conjugate gradient and Bi-Conjugate Gradient Iterative Solve Methods and demonstrated speedups varying from 1.78 to 4.13 versus the JD and CRS schemes.

## References

[1] H. Amano, T. Boku, T. Kudoh, and H. Aiso. $(SM)^2$-II: A new version of the sparse matrix solving machine. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 100–107, Boston, Massachusetts, June 17–19, 1985. IEEE Computer Society TCA and ACM SIGARCH.

[2] R. F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J. J. Dongarra. The Matrix Market: A web resource for test matrix collections. In R. F. Boisvert, editor, *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137, London, 1997. Chapman & Hall.

[3] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[4] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. Van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM Publications, Philadelphia, PA, 1990.

[5] V. Eijkhout. LAPACK working note 50: Distributed sparse data structures for linear algebra operations. Technical Re-

port UT-CS-92-169, Department of Computer Science, University of Tennessee, Sept. 1992. Mon, 26 Apr 99 20:19:27 GMT.

[6] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical report, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, June 1994. Version 2.

[7] P. Stathis, S. Vassiliadis, and S. Cotofana. Sparse matrix vector multiplication evaluation using the BBCS scheme. In *8th PCI Proceedings*, Nov 2001.

[8] V. E. Taylor, A. Ranade, and D. G. Messerschitt. SPAR: A New Architecture for Large Finite Element Computations. *IEEE Transactions on Computers*, 44(4):531–545, April 1995.

[9] S. Vassiliadis, S. Cotofana, and P. Stathis. Vector ISA extension sprase matrix multiplication. In *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685, pages 708–715. Springer-Verlag, 1999.

[10] A. Wolfe, M. Breternitz, Jr., C. Stephens, A. L. Ting, D. B. Kirk, R. P. Bianchini, Jr., and J. P. Shen. The white dwarf: A high-performance application-specific processor. In H. J. Siegel, editor, *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 212–222, Honolulu, Hawaii, May–June 1988. IEEE Computer Society Press.