BBCS Based Sparse Matrix-Vector Multiplication: Initial Evaluation

Stamatis Vassiliadis* Sorin Cotofana† Pyrrhos Stathis‡

Abstract

This paper presents an evaluation of the BBCS scheme meant to alleviate the performance degradation experienced by Vector Processors (VPs) when manipulating sparse matrices. In particular we address the execution of Sparse Matrix Vector Multiplication (SMVM) algorithms on VPs. First we introduce a Block Based Compressed Storage (BBCS) sparse matrix representation format variants, and a BBCS based SMVM algorithm. Subsequently, we consider a set of benchmark matrices, report some preliminary performance evaluations, and compare our scheme with the Jagged Diagonal (JD) scheme. Our experiments suggest that our scheme achieves an average vector register filling larger then the one achieved by JD and that is not sensitive to the assumed value of the VP section size. Due to reduction of the startup penalty when executing vector instructions, higher vector register filling will translate into higher performance.

1 Introduction

Generally speaking, due to their intrinsic support for data parallelism, vector architectures [8] are potentially good candidates to efficiently execute Sparse Matrix Vector Multiplications (SMVMs) and other types of sparse matrix manipulations. In practice however they are not as efficient[1] on sparse matrices as they are on dense. This performance degradation mainly relates to the code and data irregularity induced by the fact that SMVM algorithms make use of sparse matrix representation formats [9] in order to avoid trivial operations on zero value elements and to reduce memory bandwidth requirements. Such sparse formats are meant to be a sparse matrix representation in memory in such a way that the large amount of zeros a sparse matrix contains are not stored. Consequently, only the non-zero elements accompanied by some positional information are stored in memory and this obviously implies poor data regularity and make otherwise powerful vector instructions, e.g., vector multiply, quite inefficient. One way to alleviate this performance degradation is to augment the vector processor with some architectural support for sparse matrix manipulation as suggested for example in [7]. In this line of reasoning there was recently proposed by the authors in [12] a vector processor ISA architectural extension and an associated sparse matrix storage format. This paper constitutes a preliminary evaluation of the performance of the scheme in [12] and its main contributions can be summarized as follows:

- We introduce two Block Based Compressed Storage (BBCS) format variants, the SBBCS and S^{+}BBCS to eliminate the need for indexed vector load and store instructions.
- We consider a set of benchmark matrices and show that the average number of elements present in vector registers during the execution of the BBCS based SMVM algorithm, i.e., the vector registers filling, is above 80% for the majority of the cases and is unaffected by the VP section size increase, for all considered cases, whereas the JD vector filling drops significantly when the VP section size increases, especially for small matrices. This high vector register filling translates into high VP performance.

*Delft University of Technology, stamatis@Plato.ET.TUDelft.NL
†Delft University of Technology, sorin@Plato.ET.TUDelft.NL
‡Delft University of Technology, pyrrhos@Plato.ET.TUDelft.NL

[1]As an example a CRAY Y-MP operates, as suggested in [11], at less than 33% of its peak floating-point unit throughput when executing FEM computations.

16th IMACS World Congress (© 2000 IMACS)

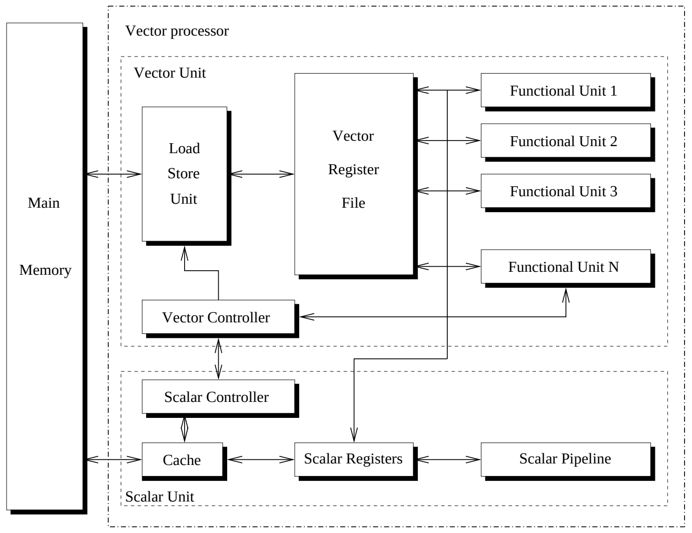

Figure 1: Vector Processor Organization

The presentation is organized as follows: In Section 2 we give a short description and preliminaries about the BBCS scheme. In Section 3 we introduce the BBCS scheme and the storage variants. In Section 4 we present some preliminary simulation results and compare the BBCS based SMVM scheme with the JD scheme. Finally, in Section 5 we draw some conclusions.

2 The BBCS Scheme

In this section we give some background information about the BBCS sparse matrix representation format and its associated architectural extension, in short the *BBCS scheme*[2] as described in [12]. The BBCS scheme is an extention to a vector processor as described in [8] (Fig 1). A vector processor differs from a scalar processor essentially in that it can execute instructions that act on series of data (*vectors*). To hold these vectors in the processor, the VP uses *vector registers*. The maximum number of elements that can fit in each the VP's vector registers is called the section size s. To support the BBCS matrix representation format an architectural extension to the assumed vector processor is described in [12] consisting of the *Multiple Inner Product and Accumulate* (`MIPA`) and *LoaD Section* (`LDS`) vector instructions. `LDS` Loads a part of the matrix that is stored in BBCS format into a vector register. The `MIPA` instruction operates on these vectors in order to perform SMVM. Further details considering the working of the architectural extension will be omited here since this does not directly affect the goal of our paper.

In short, in the associated BBCS sparse representation format an $n \times n$ matrix is partitioned in $\lceil \frac{n}{s} \rceil$ Vertical Blocks (VBs) of at most s columns, where s is the section size. Then, the non-zero values of each of these blocks are stored sequentially. Additionally, for each of the nonzero values their corresponding column positions within the block are stored as well as a number of flags that need not be discussed here. For a graphic depiction of the partitioning, see Figure 2, top left, where a 19×19 is considered and a section size of 8. The x-marks correspond to the non-zero values of the matrix.

3 BBCS based SMVM Algorithm and BBCS variants

Assuming the the architectural extensions and the basic BBCS format described in [12], the pseudo-assembler code describing the BBCS based SMVM can be written as follows:

```
for all VBs do:
       LDV    @b,VR2       ;load a vector b section in VR2
       ST     #0,RPR       ;reset RPR
loopA: LDS    @A,VR1,VR4   ;load  VB-section
```

[2]By scheme we mean here and from now on both the storage format and the algorithm used to execute SMVM.

```
LDVI   VR4,VR3      ;load vector c with index vector VR4
MIPA   VR1,VR2,VR3  ;multiply VB-section with b and add to c
STVI   VR3,@c,VR4   ;store vector c with index VR4
compute new @A
if not EOB or EOM jump to loopA
```

where `@c` is assumed to be the memory address of the first element of the vector $\vec{c}$, the result vector, and `@A` is assumed to be a pointer to the first data entry of the BBCS stored matrix.

As the indexed load/store vector instructions may loose any advantage coming for the interleaved organization [10] of the main memory, they might be much more inefficient than the standard load/store vector instructions. Within the previously described algorithm such indexed operations are used for the manipulation of the $\vec{c}$ vector. The need for using indexed loads and stores rises from the fact that when the VB entries are processed they may originate from any row position in the matrix. Since the row position of the entry corresponds to the position in the result $\vec{c}$ vector to contribute at, the $\vec{c}$ vector needs to be loaded with an index vector. To avoid using an indexed load/store based scheme we need to restrict the range of the matrix entry row positions that are loaded. Thus, in addition to the basic BBCS scheme, in the following subsections we propose two techniques that do not make any use of indexed load/store vector instructions.

3.1 Segmented BBCS (SBBCS)

Within these scheme the $\vec{A}$ matrix has to be partitioned into vertical blocks as before but now each VB is partitioned at its turn in $s \times s$ sub-blocks called *segments*. This extra partitioning is realized by adding a new field, the *EOS* (End Of Segment) flag, to the BBCS entry format. All the other field of the SBBCS format follow the BBCS structure. The *EOS* flag signals the end of a segment and thus it is set for entries representing the last non-zero element of a segment. In this new scenario the load unit executing an `LDS` instruction ceases loading, additionally to te previously specified cases, when an entry with a set *EOS* flag is encountered. This guarantees the fact that after the subsequent execution of a `MIPA` instruction the result lies within a known contiguous corresponding section of $\vec{c}$ which we can be loaded and stored without the aid of an index vector.

3.2 Extended SBBCS (S$^+$BBCS)

This scheme uses the SBBCS data format to represent the matrix. The difference is that any `LDS` instruction ceases loading after k segments and not just after one segment. This means that the result values subsequently produced by a `MIPA` instruction lies within a known contiguous $k \times s$ size section of the $\vec{c}$ vector. Consequently, the S$^+$BBCS method requires a special memory that we will call the *c-memory* that can hold $k \times s$ matrix elements and is directly accessible by the `MIPA` functional unit. There is no need to load and store the partial results in $\vec{c}$ like in the BBCS method as this is now done in the *c-memory*. We note that for this scheme to be realized there are extra architectural modifications needed, e.g., the *c-memory* and instructions to access it. Such instructions would be for instance initialize/load/store the *c-memory* and of course the `MIPA` functional unit has to be able to implicitly accesses the *c-memory*. Another difference between SBBCS and S$^+$BBCS is in the way the $\vec{A}$ matrix is accessed. Whereas SBBCS accesses the matrix in the same order as BBCS, S$^+$BBCS accesses first the first k segments of the first VB and then continues with the first k segments of the second VB and so on. After $(k \times s) \times \lceil \frac{n}{s} \rceil$ processed segments the *c-memory* has to be stored to the main memory and the next k segments of each VB can be processed and so on until the entire matrix is processed.

4 Experimental Results and Comparisons

In this section we present some initial performance evaluations for the BBCS schemes. As the BBCS schemes are generally applicable, that is, they make no assumption about the matrix properties as for instance a specific non-zero structure or a property that might arise from the type of problem that the matrix is used to solve, we do not consider in our comparisons methods that are specifically tuned for certain classes of matrices, e.g., block diagonal, etc. Given that, up to out best knowledge, from the general applicable methods reported in the literature the JD scheme [2] is considered to deliver the best performance we decided to compare our schemes with it.

The Jagged Diagonal (JD) [2] scheme provides the best SMVM performance, when no consideration is made about possible specific properties of the non-zero structure of the matrix.. The JD scheme reorders the non-zero matrix elements in columns in an attempt to provide an adequate vector register filling and to reduce the startup penalty.

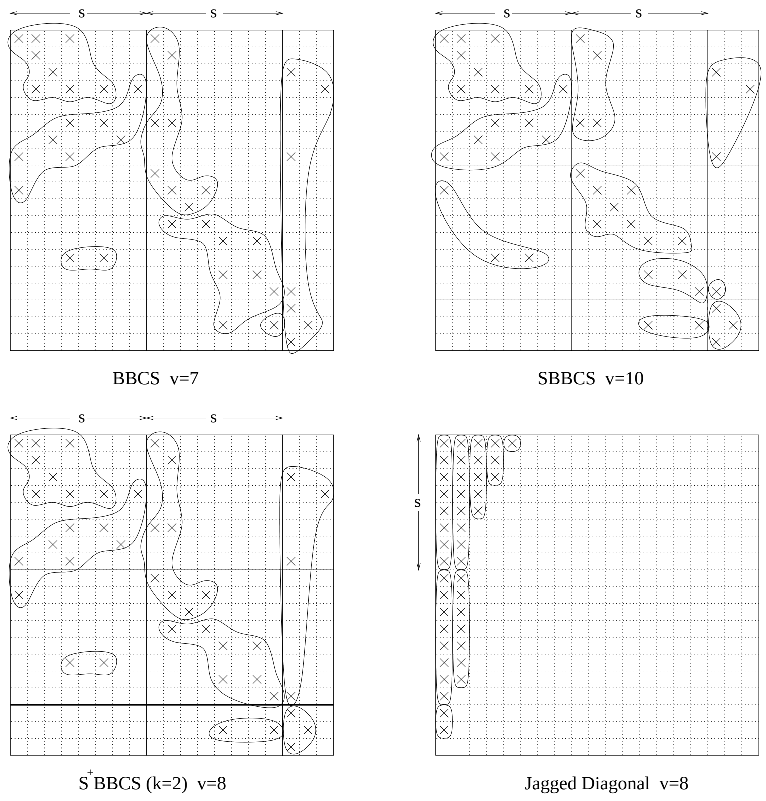

Figure 2: Vector Register Load Scenario

The JD format consists of shifting all the non-zero values of each row in the matrix to the right and form the non sparse column vectors (VV). An illustration can be seen in Figure 2.

To evaluate each of the schemes presented in the previous section we use a set of benchmark matrices and some simple simulation models that covers the loading of the matrices to be processed in vector registers and the execution of the SMVM core instructions by a functional unit. For all schemes the same assumptions have been made in order to make a fair comparison.

Within our experiments we targeted the evaluation of the average vector register filling.

The average *Vector Filling* is the average number of elements present in the vector registers during the execution of a SMVM algorithm. It is expressed as fraction of the VP section size s and calculated as follows: $VectorFilling = \frac{n}{sv}$ where n is the number of non-zero elements in the $\vec{A}$ matrix and v is the total number of vectors that are loaded by the load/store unit during the execution of a SMVM algorithm. As one may expect the value of v very much depends on the scheme. To better clarify this issue we depicted in Figure 2 the data load behavior corresponding to each of the SMVM methods we considered for the same example matrix and assuming a VP section size of 8. In the case of the BBCS scheme the load/store unit loads $s = 8$ elements to form each subsequent vector except at the end of a VB where the vector length will generally be shorter. For SBBCS the same effect occurs while loading when the end of a segment is reached and for S^+BBCS when the end of a $k \times s$ (here 2×8) segment is reached. For the JD scheme[3] the loaded vector length is smaller than s when the end of a VV column is reached.

We note here that for all the experiments where the scheme S^+BBCS is involved, we have chosen to present only the results where $k = 8$. This is because during our experiments we observed no significant improvement of the S^+BBCS performance for higher values of k on the benchmark matrices that we considered.

We selected a sparse matrix test suite from the Matrix Market on-line collection [3]. We carried a thorough examination of the statistical properties of the available matrices by focusing our attention on the statistical properties of the number of non-zeros elements per row (NZPR) because these are the parameters that mainly influence the performance of the SMVM schemes. The parameters we considered are: the average NZPR, the NZPR variance, and the existence of any rows with a large NZPR. Consequently, we divided the matrices into three categories as follows:

[3]In Figure 2 the JD format is displayed after being shifted and permuted.

Category	Matrix Name	Matrix Dimension	# of non-zeros	Average NZPR	NZPR Variance	longest NZPR
Regular Large	cavity16	4562 x 4562	138187	30	15	62
Regular Small	gre__185	185 x 185	1005	5.3	0.85	6
Irregular-1 Large	memplus	17758 x 17758	126150	5.6	13	353
Irregular-1 Small	fs_183_3	183 x 183	1069	5.8	9.1	72
Irregular-2 Large	mbeause	496 x 496	41063	83	130	489
Irregular-2 Small	tols90	90 x 90	1746	19	35	90

Table 1: Sparse Matrix Test Suite

- **Regular**: a sparse matrix that has a relatively (to others matrices of the same size) small NZPR variance and the row with the largest NZPR has an NZPR which is not significantly larger than the average NZPR plus the NZPR variance.
- **Irregular-1**: a sparse matrix that has approximately the same NZPR average and variance as a *regular* matrix but contains rows with a large NZPR.
- **Irregular-2**: a sparse matrix that has a large average NZPR, a large NZPR variance, and rows with a large NZPR.

The selected matrices are presented in detail in Table 1. For each category we have chosen two matrices that we consider representative for that category: one small and one large[4].

Figure 3 depicts the results of our first set of experiments that were targeted on the evaluation of the average vector register filling for the six matrices in the test suite. The vector filling is displayed as a function of the section size s on the x-axis. The y-axis represents the average vector length for a particular scheme divided by the section size s. By analyzing the curves in Figure 3 we can observe the following:

- The vector fillings for the BBCS schemes are rather unaffected by the section size increase and are above 80% for the majority of the cases. SBBCS and S$^+$BBCS even increase their filling percentages for larger section sizes. In contrast the JD scheme is decreasing the vector filling when the section size is increasing in all cases.
- For small matrices (left in Figure 3), the JD scheme has worse vector filling values than for large matrices (on the right). This can be explained by the fact that when the section size is larger than the dimension of the matrix a VV (of the JD scheme) can never be equal or larger than the section size and as a result the loaded vectors are shorter. In contrast, the BBCS schemes perform well on the same circumstances as they can handle the entire matrix in one or two vectors.

5 Conclusions

This paper constitutes a continuation of the research in [12]. First we introduced the Block Based Compressed Storage (BBCS) sparse matrix representation format variants, SBBCS and S$^+$BBCS, that aleviate the need for the use of the indexed loads and stores. Subsequently, we considered a set of benchmark matrices and indicated that the vector registers filling, i.e., the average number of elements in the vector registers during execution of the BBCS based SMVM algorithm, is above 80% for the majority of the cases and is unaffected by the increase of the VP section, whereas the JD vector filling drops significantly when the VP section increases, especially for small matrices.

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.

[4]Within this context a matrix is considered to be small when the number of non-zero elements it contains is in the order of the VP section size s, which is in the range of $32 < s < 1024$ in our experiments, and considered to be large otherwise.

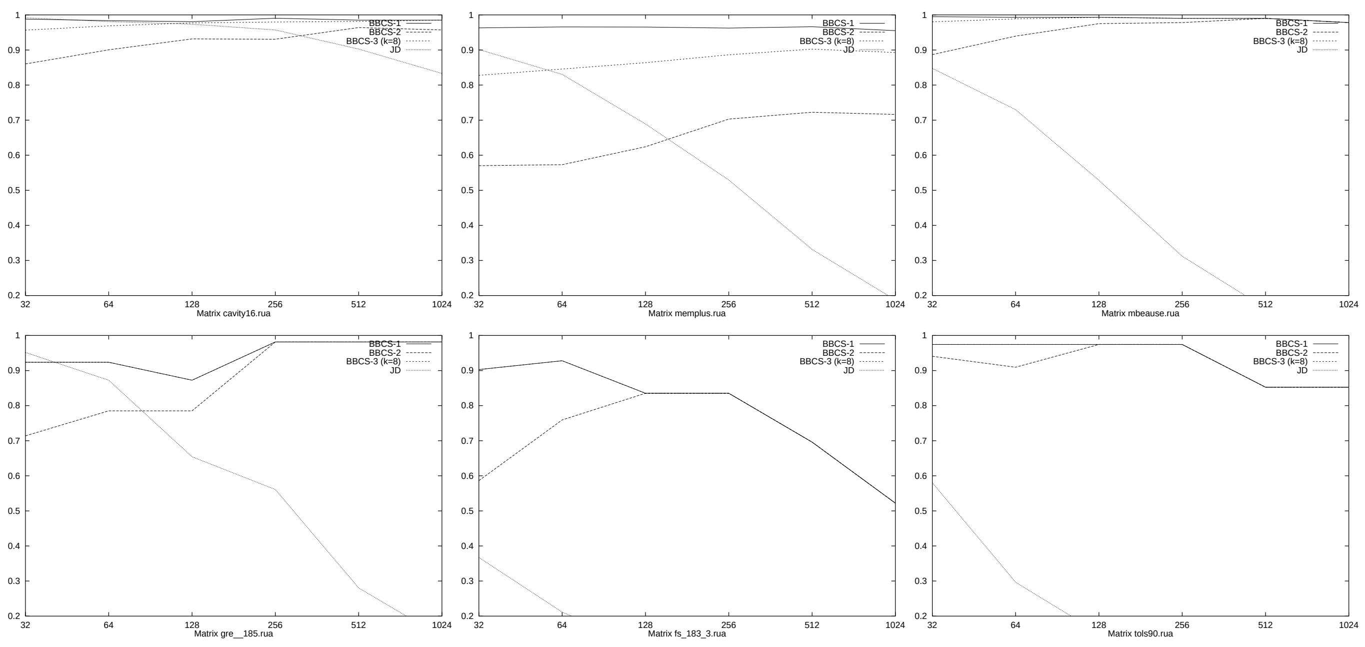

Figure 3: Average Vector Register Filling (fraction of s that is filled versus s)

[2] E. C. Anderson and Y. Saad. Solving sparse triangular systems on parallel computers. *International Journal of High Speed Computing*, 1:73–96, 1989.

[3] 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.

[4] V. Eijkhout. LAPACK working note 50: Distributed sparse data structures for linear algebra operations. Technical Report UT-CS-92-169, Department of Computer Science, University of Tennessee, Sept. 1992. Mon, 26 Apr 99 20:19:27 GMT.

[5] A. W. et al. The white dwarf: A high-performance application-specific processor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 212–222, Honolulu, Hawaii, May–June 1988. IEEE Computer Society Press.

[6] T. J. R. Hughes. *The Finite Element Method.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[7] R. N. Ibbett, T. M. Hopkins, and K. I. M. McKinnon. Architectural mechanisms to support sparse vector processing. In *Proceedings of the 16th ASCI*, pages 64–71, Jerusalem, Israel, June 1989. IEEE Computer Society Press.

[8] P. M. Kogge. *The Architecture of Pipelined Computers.* McGraw-Hill, New York, 1981.

[9] Y. Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, CA, 1990.

[10] G. S. Sohi. High-bandwidth interleaved memories for vector processors - a simulation study. *IEEE Transactions on Computers*, 38(4):484–492, Apr. 1989.

[11] 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.

[12] S. Vassiliadis, S. Cotofana, and P. Stathis. Vector ISA extension for sparse matrix dense vector multiplication. In *International Europar Conference (EURO-PAR'99)*, page In Press. Springer-Verlag, 1999.