A Hierarchical Sparse Matrix Storage Format for Vector Processors

Pyrrhos Stathis, Stamatis Vassiliadis, and Sorin Cotofana
Computer Engineering Laboratory, Electrical Engineering Department,
Delft University of Technology,
2600 GA Delft, The Netherlands
{pyrrhos,stamatis,sorin}@dutepp0.et.tudelft.nl

Abstract

We describe and evaluate a Hierarchical Sparse Matrix (HiSM) storage format designed to be a unified format for sparse matrix applications on vector processors. The advantages that the format offers are low storage requirements, a flexible structure for element manipulations and allowing for efficient operations. To take full advantage of the format we also propose a vector architecture extension that supports the HiSM format. We show that utilizing the HiSM format we can achieve 40% reduction of storage space when comparing to the Compressed Row Storage (CRS) and Jagged Diagonal (JD) storage methods. Utilizing the HiSM storage on a vector processor we can significantly increase the vector performance for Sparse Matrix Vector Multiplication (SMVM) by 5.3 times compared to CRS and 4.07 times compared to JD. Finally, to illustrate the flexibility of the format we compared the cost of an element insertion against the JD and CRS formats. We show that for an element insertion operation HiSM outperforms JD for average and large matrices although it has a slight disadvantage for small matrices and always outperforms CRS between 2 *and* 400 *times.*

1 Introduction

Sparse matrix operations constitute a significant part of many scientific computations and the irregularity of the operations induced by the sparsity patterns often inhibit the efficient use of the computational resources of conventional scalar and vector architectures. Software approaches seem to indicate that the support offered for sparse matrices by current architectures is insufficient. Augmenting traditional architectures for sparse matrix support mainly focuses on vector architectures [12, 11, 7, 10, 1] since they are best suited for large scale repetitive operations and will be our focus in the remainder of this paper.

Operations on sparse matrices introduces various problems and the advantages offered by vector processors cannot be fully utilized. First, the need to operate only on the non-zero elements often introduces indexed vector loads and stores. Such vector memory accesses are detrimental for the efficient utilization of the available bandwidth. Secondly, despite the large dimension sizes of most sparse matrices, the average number of non-zeros in a row or column is usually low. The result is that we have to operate on small vectors, a fact that does not allow for the amortization of the startup overhead and therefore the performance is again affected negatively. Furthermore, the existing sparse matrix storage formats for general types of matrices, (that is, not tied to a particular type of matrix) use a large quantity of storage space to store the positional information associated with each stored non-zero element.

Finally, the existing formats for sparse matrix storage and operation are either specific for the type of matrix used or specific for the operation. Therefore when we need to do different types of operations on a matrix, we have to either transform from one format to the other which introduces overhead or to use one type of operation inefficiently. Hence, we need a *unified format* that is efficient on all operations. To this end we have studied the various operations on sparse matrices by consulting various tool-sets for sparse matrices [9]. For our purposes we have divided them into 2 main categories:

1. *Value Related Operations* (VROs). These operations include arithmetic operations such as multiplication, addition, inner product etc. In these operations the values of the elements as well as the positional information are important. These operations will perform efficiently if we can provide the following: (1) access to long arrays which results in long vectors, (2) minimal positional data for low memory bandwidth and (3) no need for indexed loads.

2. *Position Related Operations* (PROs). These include operations such as element searching, element insertion, matrix transposition etc. Generally operations

where the actual values of the elements are not important for the outcome. These operations are facilitated when we use (4) a flexible structure where small parts can be accessed, moved or altered.

We have addressed some of the aforementioned difficulties raised by sparse matrix operations on vector processors in [12] where average speedups of 2.5 have been obtained using the proposed Block based Compression Storage (BBCS) method versus the JD method for SMVM. The BBCS scheme successfully tackles the problems of short vectors and large storage overhead by the use of a special storage format and an architectural support mechanism. However, it does not fully alleviate the need to use indexed accesses and does not offer the flexibility to allow for position related operations. Therefore we propose the Hierarchical Storage Format (HiSM) to alleviate the aforementioned problems. The contributions of this paper can be summarized as follows:

- We present a Hierarchical Sparse Matrix (HiSM) storage format that offers low storage overhead, access to long vectors, elimination of the need for indexed loads when operating and a flexible structure for manipulating the matrix. Furthermore, we outline a vector architecture extension that supports the HiSM format for taking full advantage of the proposed format.
- We show that utilizing the HiSM format we can achieve 40% reduction of storage space when comparing to the Compressed Row Storage (CRS) and Jagged Diagonal (JD) storage methods.
- To demonstrate the performance benefits of HiSM for value related operations we evaluate the performance of SMVM on a vector processor simulator. The simulations indicate that performing the SMVM utilizing HiSM executes 5.3 times faster than utilizing the CRS format and 4.07 faster than utilizing the JD format, averaged over a wide variety of matrices.
- Regarding positional operations, represented by element insertion, the introduced hierarchical sparse matrix storage format outperforms CRS in all cases in that it provides a speedup varying from 2 to 400 times. The proposed scheme is less efficient when compared to JD when considering small matrix dimensions but it substantially outperforms (up to 43 times) JD for medium and large matrix dimensions.

The remainder of the paper is organized as follows: In Section 2 we present a more detailed motivation and derive requirements for the proposed format, followed by a description of the Hierarchical Sparse Matrix Storage format. In Section 3 we evaluate the performance HiSM by means of simulation and a sparse matrix benchmark suit. Finally, in Section 4 we draw some conclusions.

2 The Hierarchical Sparse Matrix Storage Format

With the introduction of of the HiSM format we aim to tackle all the difficulties mentioned in the previous Section regarding an efficient sparse matrix format. The structure of the format is in essence a combination of an adapted version of the BBCS format, described in [11], and a hierarchical way of storing a sparse matrix, discussed in [5].

To obtain the HiSM an $M \times N$ sparse matrix A is partitioned in $\lceil \frac{M}{s} \rceil \times \lceil \frac{N}{s} \rceil$ square $s \times s$ sub-matrices where s is the section size[1] of the targeted vector architecture. Each of these $s \times s$ sub-matrices, which we will call s^2-blocks, is then stored separately in memory in the following way: All the non-zero values as well as the positional information combined are stored in a row-wise fashion in an array (s^2-blockarray) in memory. In Figure 1 (bottom left) we can observe how such a blockarray is formed containing both the position and value data from the top left s^2-block of an 64×64 sparse matrix. For demonstration purposes we have chosen here a small section size of $s = 8$. Note that the positional data consists of only the column and row position of the non-zero elements within the sub-matrix. This means that when $s < 256$, which is typical for vector architectures, we only need to store 8 bits for each row and column position. This is significantly less than other sparse matrix storage format schemes where at least a 32-bit entry has to be stored for each non-zero element. For instance, in the Compressed Row Storage format we need to store a 32-bit column position for each element and an extra vector with length equal to the number of rows of the matrix.

The s^2-blockarrays can contain up to s^2 non-zero elements. These s^2-blockarrays that describe the non-empty s^2-blocks form the lowest (*zero*) level of the hierarchical structure of our format. As can be observed in Figure 1, the non-empty s^2-blocks form a similar sparsity pattern as the non-zero values within an s^2-block, Therefore, the next level of the hierarchy, level-1, is formed in exactly the same way as level *zero* with the difference that the values of non-zero elements are now pointers to the s^2-blockarrays in memory that describe non-empty s^2-blocks. This new array which contains the pointers to the lower level is stored in exactly the same fashion in memory (see Figure 1 (bottom right). Notice that at level-1 the pointers are stored in a column-wise fashion. This can be chosen freely and is not restricted by the format. Furthermore, in Figure 1 we see that for level one an extra vector is stored in memory

[1]The *Section Size*, also called maximum vector size, is the maximum number of elements that can be processed by a vector architecture's vector instruction [8].

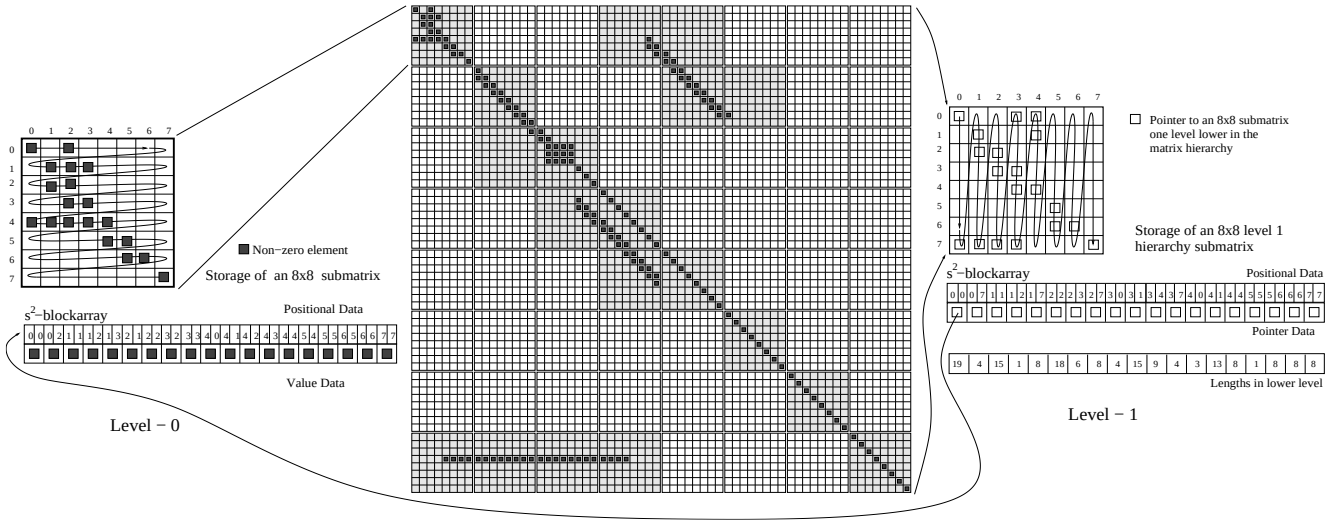

Figure 1. Example of the Hierarchical Sparse Matrix Storage Format

which is of the same length as the s^2-blockarray and contains the lengths of the level 0 s^2-blockarrays. This is necessary in order to access the correct number of elements at the level below since the number of elements can vary. The next level, level-2, if there is one (in the example of Figure 1 there is none), is formed in the same way as level-1 with the pointers pointing at the s^2-blockarrays of level-1. Further, as in any hierarchical structure the higher levels are formed in the same way and we proceed until we have covered the entire matrix in $max(\lceil \log_s M \rceil, \lceil \log_s N \rceil)$ levels. The sparse matrix is completely stored when all the levels have been stored and the matrix can be referred to in terms of the memory position of the start of the top level s^2-blockarray and its length. We can summarize the description of the Hierarchical sparse matrix storage format as follows:

- The entire matrix is divided hierarchically into blocks of size $s \times s$ (called s^2-blocks) with the lowest level containing the actual value of the non-zero elements and the higher levels containing pointers to the non-empty s^2-blocks of one level lower.
- The s^2-blocks at all levels are represented as an array (called a s^2-blockarray whose entries are *non-zero values* (for level-0) or *pointers to non-empty lower level s^2-blockarrays* (for all higher levels) along with their corresponding positional information within the block.

As can be observed in Table 1 the HiSM format offers a storage reduction of about 40% versus the JD and CRS formats and is equivalent to the reduction offered by the BBCS format. Furthermore, from the locality measure which we describe in the next Section we can conclude that each s^2-blockarray contains on average $2.18 \times s$ non-zero elements

Matrix Name	Dimension	Matrix Size	Storage Space vs JD	Storage Space vs CRS	Storage Space vs BBCS
cavity16	4562	138187	0.62	0.61	1.003
gre__185	185	1005	0.57	0.53	1.006
memplus	17758	126150	0.58	0.55	1.016
fs_183_3	183	1069	0.58	0.53	1.008
mbeause	496	41063	0.62	0.62	1.001
tols90	90	1746	0.61	0.60	1.002

Table 1. Hierarchical Sparse Matrix Storage Format compared to JD and CRS for Section size 64

thus providing access to long arrays when operating on this format. Additionally, as is illustrated in the next section, the format offers the opportunity to avoid indexed loads. Finally, the hierarchical structure offers an flexible way to access and alter parts and elements of the matrix without the need to scan the whole or a large part of the matrix.

The proposed format can be handled by conventional vector processors. However, to take full advantage of the features of the HiSM format we have incorporated a number of adaptations to the traditional vector architecture. These concerns mainly the accessing of the format from the memory and the operation on multiple rows that are contained in an s^2-blockarray. To access a s^2-blockarray we have extended the architecture by the LoaD Blockarray (LDB) instruction. LDB reads the value (or pointer) data and the positional data by one stride-1 vector access from memory and stores the resulting data in two vector registers as described in the next Section. Besides adding the LDB instruction to vector instruction set we also need to enhance the function-

ality of the vector load/store unit of the vector processor. To enable the operation on an s^2-blockarray after the LDB has read the data into the processor we provide the BMIPA instruction and an associated Pipelined Vector Functional Unit. BMIPA enables the vector processor to perform a multiplication of a section of the s^2-blockarray residing and a dense vector of size s. Both the s^2-blockarray segment and the dense vector reside in regular vector registers. We have shown in [12] how such a functional unit can be pipelined. Specifically, the MIPA functional unit described in [12] supports the non hierarchical BBCS sparse matrix storage format on which the HiSM format is based. Consequently, some implementation details differ but the principles of operation are identical and therefore the BMIPA functional unit will not be described in this paper. Some further details of the LDB and BMIPA instructions will be discussed in the following Section.

3 Performance Evaluation

In this section we evaluate the benefits of utilizing the HiSM storage method on vector processors augmented to support HiSM. The evaluation will be split into two main parts. In the first subsection, as a representative of the value related operations, we evaluate the performance of the SMVM by means of simulation on a vector processor simulator. In the second part, as a representative of the position related operations, we evaluate the cost of an element insertion in a sparse matrix. All evaluations are done by comparing with the most used methods for storing general sparse matrices, the CRS and JD. CRS is mostly used as a generic storage method for different operations and JD is specifically used for SMVM since it generally yields better results than CRS for SMVM on vector processors. Furthermore, we will also compare our results with the BBCS method, described in [12].

In CRS all the non-zeros of the sparse matrix are scanned row-wise and are stored in a long array. Additionally, the corresponding column positions of each element is stored in an equally long array. Finally, a third array of length M, where M is the number of rows of the matrix, is stored that contains the positions of the the starts of each rows in the previous two arrays.

When storing in the JD storage format the following method is used: k arrays of length M (the number of rows) are created, where k is the number of non-zeros of the row with the highest number of non-zeros. Subsequently each nth array, where $0 < n < k$, is filled with the nth non-zero of each row. If a row does not have enough non-zeros then the position is left empty. Simultaneously, a second set of k arrays is used to store the corresponding column positions of each non-zero element. Finally, all the arrays are permuted in such a way that the empty entries are moved to the end of each of the arrays and are then truncated, yielding 2 sets of k arrays with varying lengths. The 2 sets of k arrays, the permutation vector and an arrays containing the lengths of the k arrays form the full JD storage format.

Finally, in the BBCS scheme the matrix is divided into vertical block-columns of width equal to the section size s of the vector processor. For each block-column the non-zeros are scanned in a row-wise fashion and are stored in an array. An extra set of arrays store the column positions of each corresponding element plus a flag for each non-zero that denotes the end of a row (within the vertical block-column). Since the column position c is of within the range $0 < c < s$ we only need to use $\log_2 s + 1(flag)$ bits for each non-zero to store the positional information. Therefore, similarly to the HiSM the BBCS can thus achieve lower storage needs and less data transfers to access the matrix.

3.1 SMVM Evaluation

We have chosen to evaluate the SMVM for evaluating the performance of the HiSM scheme for value related operations because it is the most prevalent operation in applications involving sparse matrices such as iterative solvers and exposes the aforementioned problems of short vectors, high bandwidth, and indexed memory accesses. the SMVM involves the multiplications of a sparse matrix stored in a compressed format (CRS, JD, HiSM, BBCS) with a dense vector producing a dense vector as a result.The storage methods of the matrix defines how the SMVM algorithm will work. In this paper we will refer to the algorithms by the name of the storage method used.

3.1.1 The Simulation Environment

Before proceeding with the evaluation of the SMVM we will first describe the simulation environment.

The Simulator

All methods for the SMVM have been executed on a cycle accurate vector simulator (VS) that supports traditional vector instructions [6] as well as vector instructions and functional units that support the HiSM and BBCS formats. The VS is based on the SimpleScalar 2.0 tool-set [4], a superscalar processor simulator. SimpleScalar has been modified to support vector instructions, vector functional units and vector registers. The memory model has also been modified to reflect more correct a vector memory access model. Vector accesses are not realized through the cache but instead it is directly connected to the memory. This is due to the low temporal locality of typical vector applications. However, the spatial locality is typically high and therefore mostly a high bandwidth bank memory organization is used. We have chosen to use a memory organization with a behavior similar to the Torrent-0 vector architecture [2]. The

maximum bandwidth that the memory can provide is 4 32-bit words per cycle after a startup latency of 20 cycles. An extra cycle penalty is paid when the starting address is not aligned. The maximum bandwidth is not achievable when utilizing indexed accesses. In this case only one 32-bit word can be accessed per cycle. Therefore, for example, a contiguous vector of 64 words can be loaded in $20 + \frac{64}{4} = 36$ cycles whereas for an indexed load the access of a 64 word vector will cost $20 + 64 = 84$ cycles.

The Sparse Matrix Suite

The benchmark matrices that were used to evaluate the scheme were chosen from a wide variety of matrices that are available from the Matrix Market Collection [3]. The collection offers 551 matrices collected from various applications and includes several other collections of sparse matrices and is therefore the most complete we could get access to. Of these matrices we have selected 132 matrices taking care not to select similar matrices in terms of application, size and sparsity patterns in order to reduce the amount while keeping the variety intact. The 132 matrices matrices have been sorted using 3 different criteria that relate to various matrix properties:

- *Matrix Size*. The metric is the number of non-zeros within the matrix. The range is from 48 non-zeros for matrix *bcsstm01* to 3753461 non-zeros for matrix *s3dkt3m2* with an average of 115081.

- *Locality*. The locality is calculated as follows: First, each matrix is divided into blocks of 32×32. For each non-empty block the number of non-zeros is divided by 32 to express the number in terms of the dimension of the block. The average over all non-empty blocks is the *locality* metric. The range is from 0.07 for matrix *bcspwr10*, a matrix with a very uniform distribution of the non-zeros over the matrix to 12.85 for matrix *qc324* a matrix that contains large dense blocks. The average value is 2.18. This metric gives an indication for the vector filling efficiency when loading s^2-blockarrays utilizing the HiSM storage format.

- The *Average non-zeros per row* varies from 1 for matrix *bcsstm20*, a matrix with only a diagonal and 172 for matrix *psmigr_1* and an average of 15.9. This metric is a good indication of the efficiency of CRS versus JD. Low values result into very short vector for CRS and therefore are detrimental for the performance when using vector processors.

From each of the previous sortings of the matrices we have chosen 10 matrices to represent the set due to space limitation. The steps are constant in a logarithmic scale since we observed from the data that the distributions after sorting for each criterion was logarithmic rather than linear. Therefore for instance for the matrix size criterion each matrix is approximately 3.5 times larger than the previous one. For each metric we will refer to the 3 first matrices as *small*, the following 4 matrices as *middle* and the last as *large*.

The SMVM Methods

For the evaluation of the HiSM scheme for SMVM 5 methods are being compared to each other. For all methods programs have been coded by hand in vector assembly language and executed on the vector simulator. We provide here a brief description of each method:

CRS SMVM. This method suffers from high relative startup times because of short vectors especially when the matrix has a low average number of non-zeros per row. furthermore, the multiplicand vector is accessed by indexed loads which deteriorates performance thus to the memory behavior.

JD SMVM. This algorithm is more complicated and arduous but is more efficient on vector processors since the loaded value vectors are typically long, usually in the order of the number of rows of the matrix and thus does not suffer from high startup overhead. However, the performance will deteriorate when there are a few rows in the matrix that have a large number of non-zeros since that can result in a relatively high number of short vectors. Furthermore, similarly to CRS, JD also suffers for the need to perform indexed loads.

HiSM SMVM. As mentioned, to execute the SMVM utilizing the HiSM we need the support of a vector hardware extension and a set of new vector instructions. these have been incorporated into the Vector simulator. Because of the hierarchical structure of the HiSM format we use a recursive algorithm for performing the SMVM:

```
HiSMSMVM(s^2-Blockarray-pointer,length of s^2-blockarray,col_pos,row_pos)
1 if not lowest level (s^2-Blockarray contains pointers to (s^2-Blockarrays)
2   Load s^2-Blockarray pointers and positions → PV, PIV (LDB instruction
3   Load lengths vector → LV
4   for i in all (PV, CPV, RPV, LV) do
5     HiSMSMVM(PV[i], LV[i], col_pos * s +
                  + textPVI_CP[i], row_pos * s + textPVI_RP[i]*s )
6 if lowest level (s^2-Blockarray contains nonzero values)
7   Load s^2-Blockarray pointers and positions → VV, PIV (LDB instruction)
8   if (col_pos <> previous_col_pos) then
9     Store previous intermediate result IR_previous_col_pos
10    Load intermediate result → IR_col_pos
11    IR_previous_col_pos = IR_col_pos (update intermediate result)
12  if (row_pos <> previous_row_pos) then Load multiplicand vector → MV
13  Exec BMIPA(VV,IR_col_pos,MV,textPVI_CP,textPVI_RP) → IR_col_pos
```

In this algorithm PV stands for Pointer Vector and stores the elements of the s^2-blockarray that are pointers to s^2-blockarrays one level lower. VV stands for Value Vector and holds the non-zero values of the blockarray if we are at level-0. PIV stands for Position Information Vector and holds the column and row position of each corresponding pointer or non-zero value in the PV or VV respectively. Each element of PIV holds the column position in the least significant 16-bits of the PIV entry and row the row position

at the 16 high significant bits and are denoted by PVI_{CP} and PVI_{RP} respectively. The LoaD Block (LDB) instruction is a special HiSM support vector instruction and loads an s^2-blockarray from memory into the PV (or VV) and the PIV. In line (4) the algorithm is called recursively for each s^2-blockarray pointer and associated length. Note that for the position parameters provided to the HiSMSMVM() function the position of the current s^2-blockarray is added to the the position of the parent s^2-blockarray multiplied by the section size s in order to keep track of the position in the matrix when we arrive at the lowest level where the actual position of the s^2-blockarray in the matrix must be known in order to load the correct segments of the multiplicand vector and partial result. When these are loaded the BMIPA instruction executes a matrix vector multiplication of the non-zero elements in the current level-0 s^2-blockarray with the loaded segment of the multiplicand vector using the positional information in PVI and adds the result to the intermediate result vector IR_{col_pos} The multiplication completed when all the level-0 s^2-blockarrays have been visited by the recursive algorithm and operated upon. The whole process is initiated by calling HiSMSMVM(top level s^2-Blockarray-pointer, top level length , 0 , 0).

The main advantage of HiSM SMVM over the other methods, albeit a bit more complicated in structure, is that we do not use any indexed loads. The reason for this is that the multiplicand vector and result segments that are loaded for the BMIPA operation are treated as dense vectors.

CHiSM SMVM. CHiSM stands for Cached HiSM. The algorithm for this method is similar to the HiSM algorithm previously described with one difference. The processor is provided with a vector Cache memory which can store $s \times s$ elements and is directly accessible by the MIPA unit. This way we can load the intermediate result vector in the level-1 of the hierarchy rather than level-0 as in the HiSM algorithm. This reduces the number of accesses to the memory and increase performance.

BBCS SMVM. The BBCS algorithm is described in detail in [12] and we not be described here. However we mention that the algorithm also makes use of an Vector architecture extension support and tackles the problem of short vector access and mainly draws its advantage by having a lower memory bandwidth overhead compared with CRS and JD. However, it still suffers from the need to use indexed accesses.

3.1.2 SMVM Simulation Results

In this section we present the performance results from executing the various previously described algorithms on the vector processor simulator. All simulation have been run with a section size of 64, a typical vector processor section size. The Figures 2, 3 and 4 depict the performance results for each set of matrices obtained by the sorting criteria that were described in Subsection 3.1.1. For each matrix the 5 bars correspond to each of the 5 compared SMVM methods and represent execution times. Due to wide variety of sizes of the matrices, and thus cycle counts for performing the SMVM, all the results have been normalized to the method that has the best overall performance: CHiSM. This is depicted in column 2 of Table 2 where the non-weighted average execution times for all the matrices are compared. The CHiSM algorithm achieves a speedup of 5.3 over the CRS method, 4.07 over JD and 1.5 when compared to BBCS. In Figure 2 the matrices have been sorted according to the number of non-zero elements, increasing in size from left to right. We observe that in spite of the fact that CHiSM has an overall superior performance than HiSM, for small matrices the performance is lower. This is attributed to the fact that the CHiSM algorithm loads and stores the intermediate result vector on level-1 of the matrix hierarchy. For a section size of $s = 64$ this translates into a vector load of $64 \cdot 64 = 4096$ elements. For a very small matrix with only 1000 or less non-zero elements this results in a large number of unnecessary memory accesses. However, for medium and large sized matrices the results are on average superior. This is depicted in columns 2,3 and 4 of Table 2. The numbers depicted shown here are derived from Figure 2 grouped by size.

Comparison	All Matrices	Small Matrices	Medium Matrices	Large Matrices
HiSM vs CRS	4.36	7.16	6.00	4.14
HiSM vs JD	3.34	3.23	2.92	5.52
HiSM vs BBCS	1.23	1.39	1.43	2.30
CHiSM vs CRS	5.30	4.05	10.2	5.33
CHiSM vs JD	4.07	1.92	5.01	7.00
CHiSM vs BBCS	1.50	0.83	2.43	2.94

Table 2. SMVM performance comparison averages for all matrices in the matrix suite and separately for small, medium and large sized matrices

In Figure 3 the matrices are ordered by the average number of non-zero elements per row. We observe that for larger average number of non-zero elements per row the CRS performs better than JD. This is due to the fact that longer vectors can be formed when loading each row. Inversely, JD performs better for low average number of non-zero elements per row. There is one exception, matrix *shl_0*, where JD exhibits similar bad performance as CRS. The reason for this is that this particular matrix has one row with a large number of non-zeros (422) while having a low average (2). As we described in the JD SMVM algorithm, this too results in short vectors and is detrimental for the per-

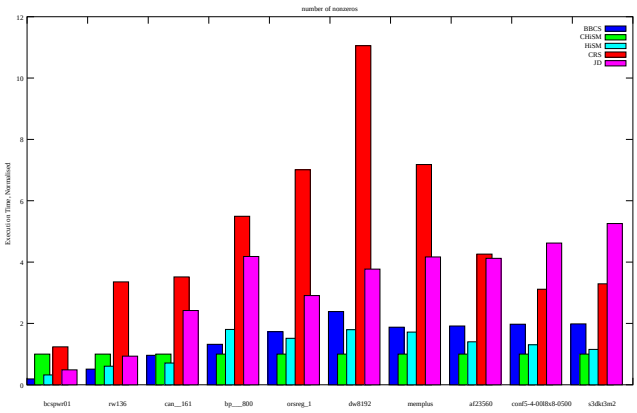

Figure 2. SMVM Performance results for increasing matrix size

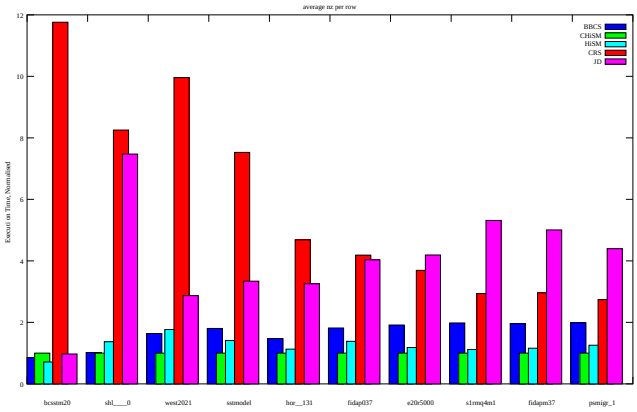

Figure 3. SMVM Performance results for increasing average non-zero elements per row

formance. We can also observe that the behavior of BBCS, HiSM and CHiSM methods are not affected by the average non-zero elements per row and always outperform CRS and JD. The only exception is matrix *bcsstm* where CHiSM is outperformed by JD. However, *bcsstm* is a very small matrix (485×485, 485 non-zero elements). The reasons for this behavior are described in the previous paragraph.

Finally, in Figure 4 the performance of SMVM is depicted for matrices with increasing locality, as discussed in the previous subsection. Here too we can observe that HiSM and CHiSM outperform CRS and JD. The exception is matrix *bcspwr10*, the matrix with the lowest locality in our matrix suite, where HiSM exhibits the highest execution times. The performance suffers from the fact that the non-empty s^2-blockarrays are poorly filled. Therefore, a large number of unnecessary memory accesses are performed. This performance degradation can be amortized by

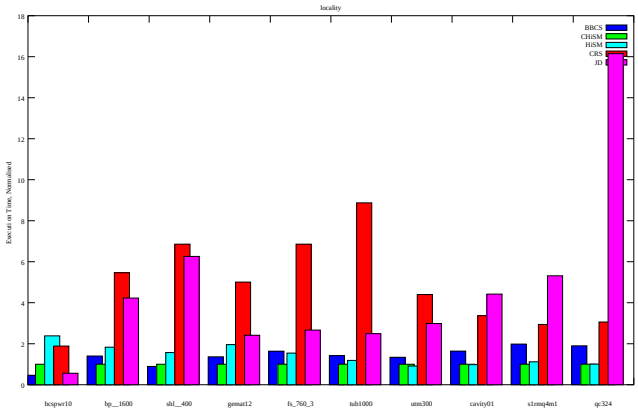

Figure 4. SMVM Performance results for increasing locality

the use of a vector cache and we see that CHiSM has a better performance although it is still outperformed by JD and BBCS. However, matrices with such a low locality a rare exceptions and therefore we can accept this disadvantage. Another point of interest in Figure 4 is the high execution time for matrix *qc324* using the JD SMVM method. In this case, the structure of the matrix is such that the JD algorithm results in memory accesses that repeatedly accesses the same memory bank. This can happen only with a indexed vector access or a stride which is a multiple of the number of banks. HiSM and CHiSM do not suffer from this drawback since all the accesses are done with stride-1.

3.2 Element Insertion Evaluation

In this subsection we illustrate the flexibility of the format to facilitate position related operations. We illustrate this by calculating the number of steps are needed to insert an element into the sparse matrix stored in a given format. This operation includes the searching of the correct position and the displacing of a number of elements in order to insert the new element. The complexity (or the number of comparisons) of the search operation is very dependent on the sparsity pattern and therefore can not be calculated analytically. For the actual values we will have to execute the algorithms on real matrices. However we can calculate a maximum, or worst case for an $N \times N$ sparse matrix A and section size s: The number of levels is $\lceil \log_s N \rceil$. Each blockarray can have up to s^2 elements and thus in the worst case we would have to make $\frac{\lceil \log_s N \rceil \times s^2}{2}$ comparisons, having therefore a complexity of $O(s^2)$. For a 10000×10000 matrix and $s = 64$, both typical values, the worst case is on average equal to 6144 comparisons. However, the worst case will only hold when the sparse matrix is dense or almost dense. Sparse matrices contain only a small

Matrix Name	Dimension	Matrix Size	Search & Insert vs BBCS	Search & Insert vs JD	Search & Insert vs CRS
cavity16	4562	138187	346	4.4	200
gre_185	185	1005	145	0.42	3.46
memplus	17758	126150	136	43	464
fs_183_3	183	1069	122	0.5	4.38
mbeause	496	41063	672	0.25	30.5
tols90	90	1746	438	0.06	2

Table 3. Hierarchical Sparse Matrix Storage Format compared to JD and CRS for Section size 64

percentage of non-zeros and therefore the amount of comparisons is considerably smaller than the worst case. In fact, as we calculated with the *locality* metric in Section 3.1.1, s^2-blockarrays contain only a 2.18 $\times$ (block dimension), which makes the search complexity $2.18 \times s$ on average, or $O(s)$, rather than $O(s^2)$. In Table 3, we give the number of steps (comparisons + displacements) needed to insert an element in the matrix for a number matrices. The results are compared to the BBCS, CRS and JD storage methods. Using the BBCS method the insertion of an element has a cost of $O(n)$ where n is the number of non-zero elements in the matrix and we see clearly that for all matrices and specially for the large matrices the speedup is considerable. Comparing with the JD format we observe that for matrices with small dimensions JD can outperform the HiSM storage format. This is attributed to the fact that all the elements of a small matrix can fit into a few s^2-blockarrays and therefore we cannot benefit from the hierarchical structure. However, for average and large matrices the HiSM format yields better results. Finally we observe that the HiSM outperforms the CRS for all formats mainly because it has linear structure. However, we should note that methods exist to reduce the overhead of inserting new elements when using the CRS method by allowing for extra elbow room and taking advantage of the fact that usually multiple elements are inserted at one time rather one at a time. Still however, CRS remains less flexible than the Hierarchical HiSM structure.

4 Conclusions

In this paper we proposed the HiSM storage format for the manipulation and operation of sparse matrices on vector processors. We showed that by utilizing the HiSM format we can achieve 40% reduction of storage space when comparing to the Compressed Row Storage (CRS) and Jagged Diagonal (JD) storage methods. We demonstrated the performance benefits of HiSM for value related operations by evaluating the performance of SMVM on a vector processor simulator. The simulations indicated that performing the SMVM utilizing HiSM executes 5.3 times faster than utilizing the CRS format and 4.07 faster than utilizing the JD format, averaged over a wide variety of matrices. Finally, regarding positional operations, represented by element insertion, the introduced hierarchical sparse matrix storage format outperformed CRS in all cases and provides a speedup varying from 2 to 400 times.

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] K. Asanovic and D. Johnson. Torrent architecture manual. Technical Report CSD-97-930, University of California, Berkeley, Jan. 24, 1997.

[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] 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.

[5] S. E. Computer implementation of the finite-element procedure. In *ONR Symposium on numerical and computer methods in structural mechanics*, 1971.

[6] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantative Approach*. Morgan Kaufman, San Mateo, California, 1990.

[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] A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz. The IBM System/370 vector architecture: Design considerations. *IEEE Transactions on Computers*, 37:509–520, 1988.

[9] 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.

[10] 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.

[11] 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.

[12] S. Vassiliadis, S. Cotofana, and P. Stathis. Block based compression storage expected performance. In *Proceedings of HPCS2000, Victoria*, pages 389–406, 2000.