Sparse Matrix Transpose Unit

Pyrrhos Stathis Dmitry Cheresiz Stamatis Vassiliadis Ben Juurlink
Electrical Engineering Department,
Delft University of Technology,
Delft, The Netherlands
Email: {pyrrhos,cheresiz,stamatis,ben}@duteppO.et.tudelft.nl

Abstract— **A large number of scientific apllications involve the operation on, and manipulation of sparse matrices. Irregular structure of these matrices, however, causes hardware that otherwise behaves efficient on regular data to severely suffer in performance when handling sparse matrices. In order to tackle this problem, a scheme consisting of a novel Hierarchical Sparse Matrix (HiSM) storage format and an associated architectural concept have been presented. In this paper we propose, describe, and evaluate a hardware mechanism to facilitate transposition of a sparse matrix stored in the HiSM format. The proposed hardware is meant to be embedded in a vector processor as a functional unit. The main part of the unit consists of an $s \times s$ word in-processor memory, where s is the vector processor's section size. We determine the best parametrs for the proposed mechanism and show that it provides for the HiSM-based transposition a speedup in the range from 1.2 to 32.0 times (average 17.6) with respect to the transposition based on the most widely used Compressed Row Storage format.**

Keywords— **Vector processor, matrix transpose, sparse matrix, functional unit**

I. Introduction

In many scientific computing areas manipulation of sparse matrices constitutes the core computation. However, the irregularity of the matrix sparsity pattern, i.e. the distribution of the non-zeros within the matrix, make many operations on sparse matrices execute inefficiently on traditional scalar and vector architectures. This problem has been tackled by both software and hardware approaches. For the cost reasons, most of the techniques are software-based [1], [2], However, research focused on hardware approaches [3], [4], [5], [6] indicates that much greater improvements can be obtained. In [5] the authors report for multiplication of a sparse matrix with a vector a speedup of up to 5 times (depending on the sparsity pattern) using the novel *Hierarchical Sparse Matrix (HiSM)* storage format and an associated vector architecture extension, with respect to the Jagged Diagonal (JD) and Compressed Row Storage (CRS) methods on a conventional vector processor. It has also been suggested in [5] that the use of HiSM is likely to provide high speedups not only for the sparse matrix-vector multiplication but also for other operations.

In this paper we address the problem of transposition for sparse matrices, which is often inefficient on traditional vector processors due to the irregular structure of the data. This holds epsecially for transposition algorithms which work on general types of sparse matrices, i.e., sparse matrices which do not have an a priori known structure. The contributions of this paper can be summarized as follows:

- We propose the novel *Sparse matrix Transposition Mechanism (STM)*, which facilitates transposition of general sparse matrices stored in the HiSM format and can be implemented as a functional unit for a vector processor.
- We calculate the optimal parametrs for the mechanism and and study the performance improvements it provides with respect to the sparse matrix transposition for the most widely used CRS storage format implemented on a traditional vector architecture. We show that HiSM-based transposition exhibit a speedup of 17.6, averaged over a variety of different types of sparse matrices.

The remainder of the paper is organized as follows: In the next Section we provide with some background information on transposition, vector processors and the hierarchical sparse matrix storage format. In Section III we describethe proposed mechanism. Subsequently, in Section IV we evaluate the performance of the porposed mechnism and finally, in Section V we draw some conclusions.

II. Background

The transposition of an $M \times N$ matrix A produces the $N \times M$ matrix A^T, such that $a^T_{ij} = aji$. The operation consists of the exchange of the rows and columns of the matrix. Thus it is an operation that alters not the values of the elements but only their positions. For a dense matrix, the problem is trivial and can be solved by addressing a row-wise stored matrix with a stride equal to the number of rows of the matrix or vice versa. Sparse matrices however are usually stored in a more complex way that involves the storage of the non-zero values and their positional information [1], [2]. This results in the need of using costly sorting algorithms in order to perform the transposition. With the proposed mechanism we attempt to streamline this operation and make it suitable for a vector processor.

Vector processors, such as the one depicted in Figure 1 are based on architectures that support the execution of *vector instructions*. On most current vector architectures [7], the vectors are copied from the main memory into *vector registers* within the processor before they are operated upon. Vector registers are arrays of scalar registers that hold (parts of) the vectors to be processed. Due to the fact that the vector register length can not be arbitrarily large, when operating on large vectors they have to be divided into smaller parts, a technique that is usually called *strip mining*, each of which cannot be larger than the maximum amount of elements a vector register can hold, i.e., the architecturally defined *section size* of the VP. In a VP the operations are carried out by (usually) pipelined *Functional Units* (FU) that are able to fetch one or more new element per cycle from each of the source vector register(s) involved, operate on it/them, and return the result(s) to the result (vector) register. Typical functional units employed in a vector processor are the ALUs and multipliers. The proposed transposition mechanism for HiSM is to be incorporated as a functional unit of a vector processor as well.

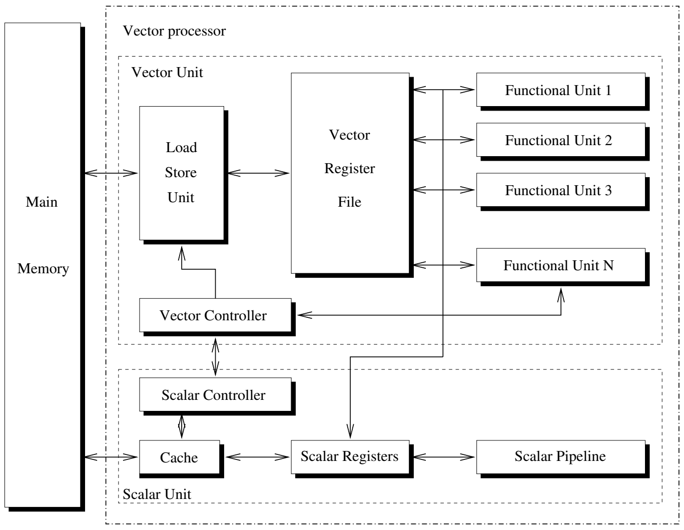


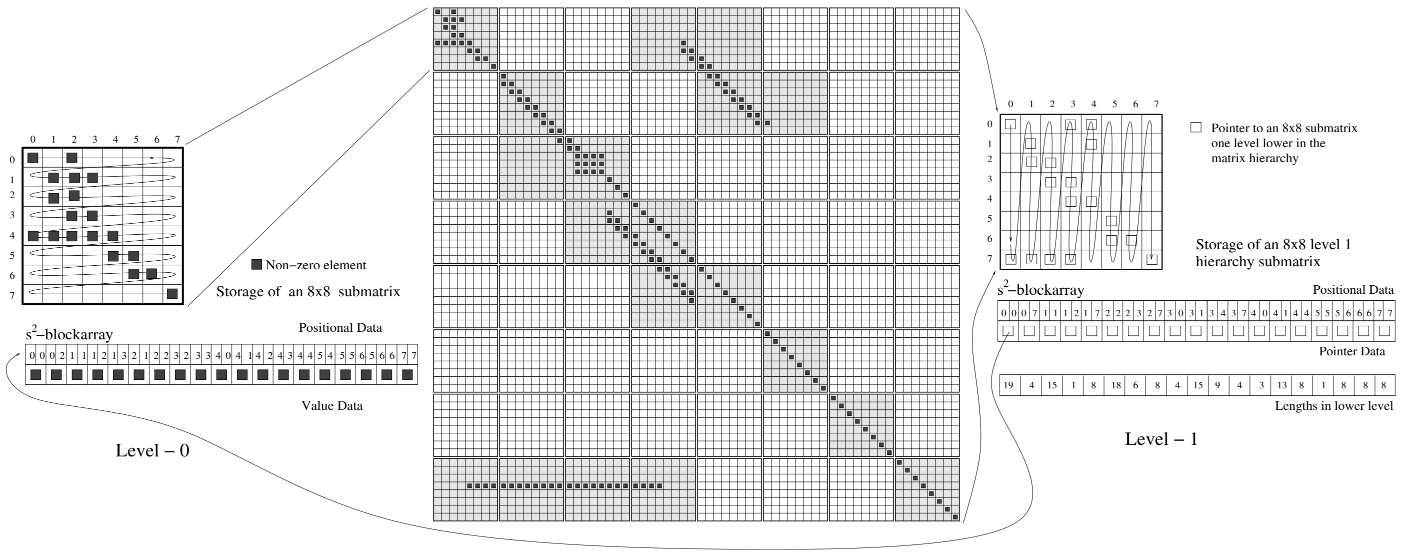
Fig. 1. Vector Architecture

Before proceeding with the description transposition mechanism we will first give a brief description of the *Hierarchical Sparse Matrix storage format* (HiSM), the format that we will assume for the remainder of our paper described 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 2 (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 2, 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 containing the pointers to the lower level is stored in exactly the same fashion in memory (see Figure 2 (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, as depicted in Figure 2, for level-one an extra vector is stored in memory. It has the same length as the s^2-blockarray and contains the lengths of the level-0 s^2-blockarrays. For each non-zero pointer p at level-1, the corresponding vector element holds the length (i.e., the number of nonzeroes) of the level-0 blockarray pointed by p. This vector is necessary to access the correct number of elements at the level-0, since this number can vary. The next level, level-2, if there is one (in the example of Figure 2 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

[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].



Non−zero element
Storage of an 8x8 submatrix
s^2−blockarray
Positional Data
Value Data
Level − 0
Pointer to an 8x8 submatrix one level lower in the matrix hierarchy
Storage of an 8x8 level 1 hierarchy submatrix
Pointer Data
Lengths in lower level
Level − 1

Fig. 2. Example of the Hierarchical Sparse Matrix Storage Format

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 remark that the provided HiSM description assumes that the dimension of the matrix is $s^q \times s^q$. If this is not the case, the matrix is padded by zeroes to form an $s^q \times s^q$ matrix.

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 an 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. For levels higher than level-0, next to each pointer to a non-empty lower level blockarray the length of this array is stored.

III. The Transposition Mechanism

As mentioned previously, the proposed Sparse matrix Transposition Mechanism (STM) can be implemented as a functional unit of a vector processor. An instance of the mechanism for a section size $s = 8$ is depicted in Figure 3. The main part of the unit consists of the $s \times s$-memory. The $s \times s$-memory is used to store an s^2-block of a hierarchically stored matrix. The mechanism can transpose one s^2-block at a time. First, the s^2-block is stored in the $s \times s$-memory one section at a time. When the complete s^2-block is stored, the s^2-block is then read from the $s \times s$-memory in the transpose fashion than storing. For example, if data was stored in a row-wise fashion (entering the $s \times s$ memory in Figure 3 from the top), it is read in a column-wise fashion, exiting the memory from from the left. We now illustrate the procedure in more detail, showing how the level-0 s^2-block depicted in Figure 2 is transposed. We assume that a part of an s^2 block is stored in a vector register R. The contents of a register R are stored in the $s \times s$-memory via the column-wise I/O-buffer located at the top of the unit. The depth of this buffer defines how many elements can maximally be stored per clock cy-

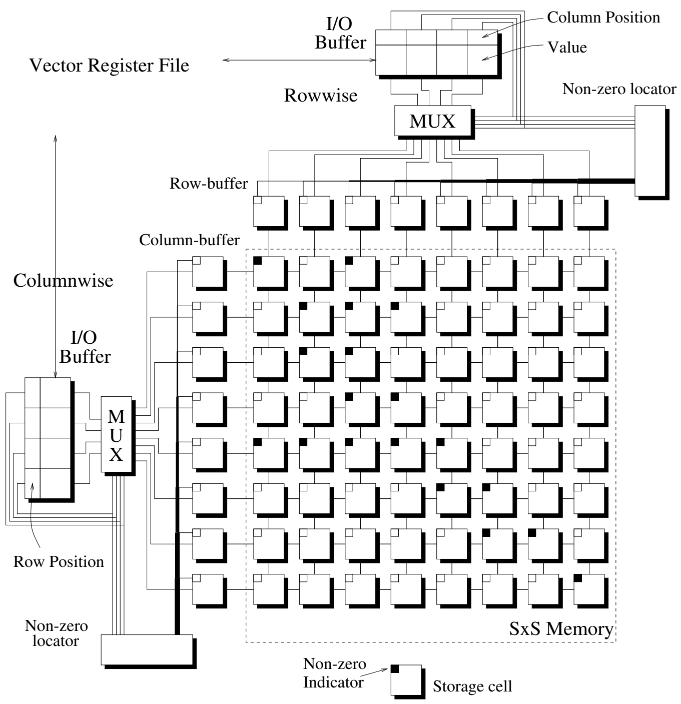


Fig. 3. The Sparse matrix Transposition Mechanism (STM)

cle and is referred to as the $s \times s$*-memory buffer bandwidth* B, which in the case of Figure 3 is 4. At each cycle the I/O-buffer is filled with non-zero elements of the same row along with their corresponding column positions. In the next cycle, the column positions are used by the Non-zero Locator unit to scatter the non-zero values to correct positions in the row-buffer. The non-zero indicator at the corresponding cells of the buffer are then set accordingly to indicate the a non-zero or a zero value. This process is repeated until all the non-zeroes in the current row are stored in the row buffer. Thereafter, the entire row buffer is copied into the $s \times s$-memory using the row position information (not shown in Figure 3). Then, the elements of all the following rows is stored in the $s \times s$ memory in the same way. Now, the transposition of the s^2-block can be obtained by reversing the order used for storing, at the column-wise section of the STM. Column by column, the s^2-block is moved into column-buffer. There, using the Non-zero-Locator, the non-zero values and their row positions are copied into I/O-buffer (maximally B at a time) and then stored into a register in the register file.

The implementation of the non-zero locator is not trivial and we, therefore, describe it below in further detail. Non-zero locator is graphically depicted in Figure 4. The function of this circuit is to extract from a string of input bits (the non-zero indicators) the position of the first B 1's. When there are more than B non-zero elements the located non-zeros are set to zero (not depicted in Figure 4) and the process is repeated in order to locate the following B non-zero elements. When there are less than B non-zero elements one or more of the “0”-counters will produce an overflow. This overflow indicates to the control logic that a new row or column needs to be fetched from the $s \times s$-memory.

As we have mentioned, the Transpose Mechanism can only transpose an s^2-block. However, because of the similar structure of the HiSM at all hierarchy levels we can apply the same transposition mechanism on all levels in order to achieve the transposition of the entire matrix, Figure 5 graphically illustrates this principle. In the following we show the validity of this approach more formally. Consider an element a_{ij} at position (i, j) of the matrix A which has to be transposed. We can write the position coordinates as a combination of the coordinates at each hierarchy level: $i = i_0 + i_1 s + i_2 s^2 + \ldots + i_q s^q$ where i_k is the row position of a_{ij} at level k, s–the section size (dimension of the s^2-block), and $q = max(\lceil \log_s M \rceil, \lceil \log_s N \rceil)$ is the number of hierarchy levels. For example, for the element $a_{10,10}$ of the matrix depicted in the left part of Figure 5, the i-coordiantes are as follows: $i = 10$, $i_0 = 2$, and $i_1 = 1$. Similarly, for the column coordinates we have

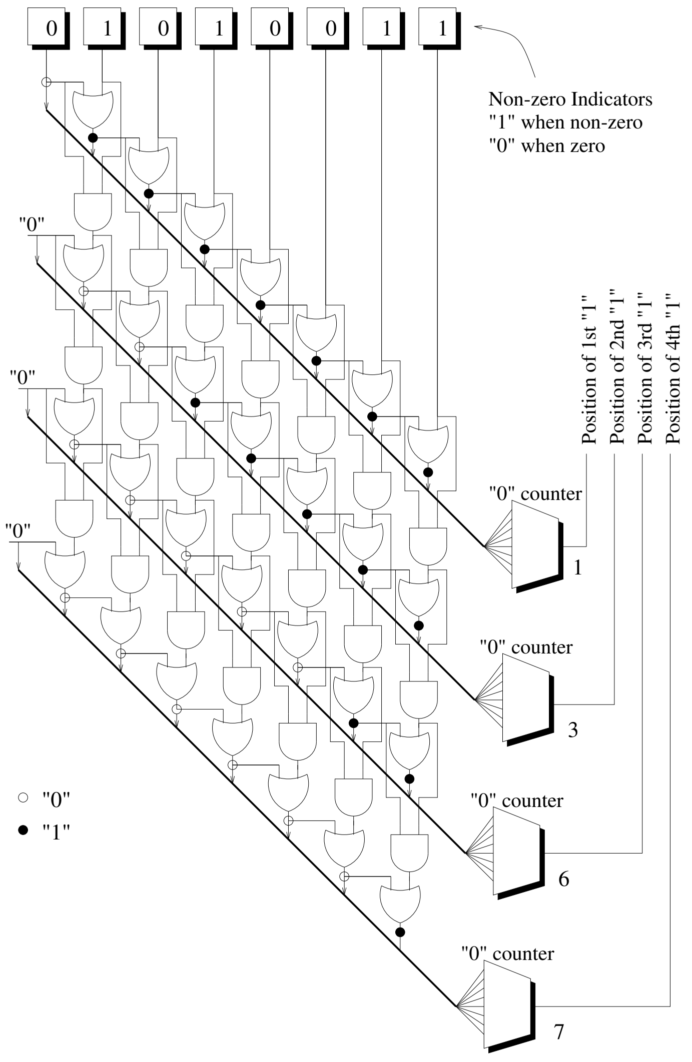


Fig. 4. The Non-zero Locator

$j = j_0 + j_1 s + j_2 s^2 + \ldots + i_q s^q$. We remark that i_k, j_k are exactly the position coordinates stored in the s^2 blocks in HiSM format. Let $\tilde{i}$ and $\tilde{j}$ denote the the new coordinates of the element a_{ij} after transposition, and $\tilde{i_k}$ and $\tilde{j_k}$ be the coordinates at hierarchy level k after transposition. Since we transform the element positions within the s^2-blocks at each level, the new coordinates at each level will become $\tilde{i_0} = j_0, \tilde{i_1} = j_1, \ldots, \tilde{i_q} = j_q$ and $\tilde{j_0} = i_0, \tilde{j_1} = i_1, \ldots, \tilde{j_q} = i_q$. Therefore,

$$\begin{aligned} i &= i_0 + i_1 s + i_2 s^2 + \ldots + i_q s^q = \\ &= \tilde{j_0} + \tilde{j_1} s + \tilde{j_2} s^2 + \ldots + \tilde{j_q} s^q = \tilde{j}, \\ j &= j_0 + j_1 s + j_2 s^2 + \ldots + j_q s^q = \\ &= \tilde{i_0} + \tilde{i_1} s + \tilde{i_2} s^2 + \ldots + \tilde{i_q} s^q = \tilde{i} \end{aligned}$$

This shows exactly that transposing the blocks at all level results in the transposition of the whole HiSM-stored matrix.

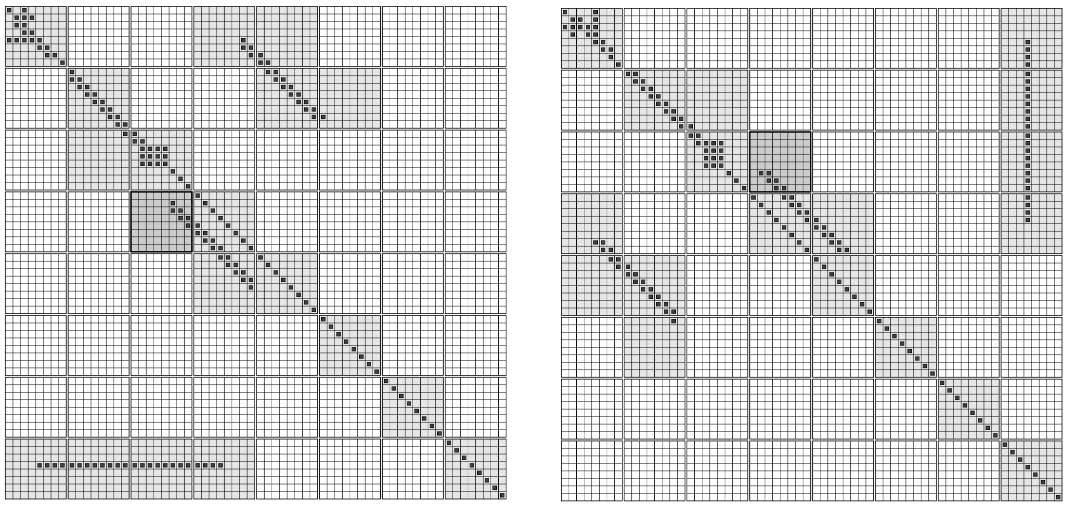


Fig. 5. Hierarchical Matrix Transposition: Transposing each block at all hierarchies is equivalent to the trasposition of the entire matrix

IV. Performance Evaluation

In this section we study the performance of the vector processor extended with the proposed matrix transposition mechanism for HiSM and compare it with the performance of a standard vector processor on the transposition algorithm based on the most widely used storage format for general sparse matrices, the Compressed Row Storage (CRS). Before proceeding to the simulation results we describe the simulation environment, the way the HiSM and CRS transposition algorithms has been vectorized, and benchmark matrices.

A. Experiment Methodology and Tools

The performance comparison of HiSM and CRS transpose is carried out on a vector processor simulator that we have developed. It supports both standard as well as HiSM funtionality and is based on the SimpleScalar [9] simulator. SimpleScalar has been extended to support vector instructions, vector functional units and a vector memory unit. The memory unit (Vector Load/Store) model is a high bandwidth memory that can support the access of 4 32-bit words a cycle after a startup latency of 20 cycles. For indexed accesses the memory can only provide one 32-bit word per cycle. Therefore, for example, a contiguous vector of 64 words can be loaded in $20 + \frac{64}{4} = 36$ cycles, whereas $20 + 64 = 84$ cycles are needed to perform an indexed load of a 64-element vector.

Furthermore, we have set the funtional unit paralellism p of the vector processor to be 4. This parameter denotes the maximum number of elements of a vector that can be processed in a single cycle. In the case of the proposed transpose mechanism p is equal to the buffer bandwidth parameter B. The section size (vector register length or maximum vector length) was set to 64. Finally, we configured the simulator to support the vector chaining (forwarding of the results of one functional unit to the next), which allows to overlap the execution of dependent vector instructions.

The vector code for both HiSM and CRS have been hand-coded in assembly. The HiSM implementation employes recursion since it has to deal with a hierarchical data structure. The pseudo code for it, with a numbfer of the actual implementation details been ommited, is presented in Figure 6. The transpose_block() procedure performs the transposotion of an $s \times s$-block and is called recursively at line 22. The BSA denotes the starting address of the s^2-blockarray The first two *for* loops (lines 2-9) perform the actual tranposition of the s^2-block elements. This is done by first loading the elements plus the positional information into the processor and storing the values row-wise into the $s \times s$ memory (first *for* loop). Subsequnetly the reverse process is performed with the with the difference that the elements are loaded column-wise from the $s \times s$ memory in order for the elements to be arranged in the transposed order. This code is fully vectorized and makes use of new vector instructions that have been developed to support the HiSM format (see also [5]). More specifically, the code that performs the trasposition of the s^2s block is depicted in Figure 7 and corresponds to lines 2-9 in Figure 6. In the first two lines the Start address and the length of the s^2-blockarray are loaded into scalar registers. Subsequently, the *icm* instruction initializes the $s \times x$ memory by setting all the *non-zero indicator* (see Figure 3) to zero. On the next line the *ssvl* instruction sets the vector length register of the vector processor to vl = max(s,$r1$) where s is the section size of the processor and substracts that value

```
1  $transpose\_block(BAS, BAL, LVL)$
2  **for** (all Block Sections (BS) in memory) **do**
3      Load BS from main memory to Vector Register
4      Store BS row-wise in $s \times s$ memory
5  **od**
6  **for** (all Block Sections (BS) in $s \times s$ memory) **do**
7      Load BS from $sxs$ memory to Vector Register
8      Store BS to memory
9  **od**
10 **if** $(LVL \neq 0)$
11   **for** (all Lengths Sections (LS) in memory) **do**
12       Load LS from main memory to Vector Register
13       Store LS row-wise in $s \times s$ memory
14   **od**
15   **for** (all Lengths Sections (LS) in $s \times s$ memory) **do**
16       Load LS from $sxs$ memory to Vector Register
17       Store LS to memory
18   **od**
19   **for** (all pointers (PTR) in block) **do**
20       Load PTR
21       Load corresponding Length (LEN)
22       Call transpose_block(PTR,LEN,LVL-1)
23   **od**
24 **fi**
```

Fig. 6. Transposition for HiSM.

```
ld       r1, BSA        # Start Address
ld       r2, BSL        # Block Length
icm                     # Init sxs Memory
Loop1:
ssvl     r2             # Set vector length
v_ldb    r1, vr1, vr2   # Load block elements
v_stcr   vr1, vr2       # Store row-wise in sxs memory
bne      r2, Loop1      # repeat Loop 1
ld       r1, BSA        # Start Address
ld       r2, BSL        # Block Length
Loop2:
ssvl     R2             # Set vector length
v_ldcc   vr1, vr2       # Load column-wise from sxs memory
v_stb    r1, vr1, vr2   # Store block elements
bne      r2, Loop2      # repeat Loop 2
```

Fig. 7. Transposition for HiSM.

from $r1$. The vl parameter indicates to the subsequent vector instructions how many elements to process. Next, the v_ldb loads a section of length vl of the s^2-block array to the vector processor. The element values or pointers are stored in vector register $vr1$ and the corresponding row-column pairs in register $vr2$. The $vr1$ is updated automatically by thev_ldb instruction. The $vr1$ and $vr2$ registers are then stored row wise in the $s \times s$-memory by the use of the v_stcr. This instruction stores the elements at the corresponding positions in the $s \times s$-memory as described in Section III via the row-wies I/O buffer. During execution the corresponding *non-zero indicators* are also set. This process repeats until $r2 = 0$ and all the sections of the s^2-block have been processed. The second part of the code is the reverse process. The v_ldcc instruction loads a section of elements from the $s \times s$-memory and stores the values or pointers (depending on the level of the hierachy) to the vector register $vr1$ and the corresponding row-column pairs to the vector register $vr2$. The v_stb instruction then stores the $vr1$ and $vr2$ vectors into main memory in the HiSM format. Note that the same memory location and amount as the original is needed to store the transposed block and therefore no allocation of memory for the transposed is needed as is the case with CRS.

We remark that the vector instruction pairs (v_ldb, v_stcr) and (v_ldcc, v_stb) can be chained, i.e., the results of one instruction can be forwarded to the next. However, due to fact that the $s \times s$-memory has to be filled before it can be read back, the transposition unit can not be fully pipelined. Nevertheless, separately, the write and read phases can be pipelined in three stages. This means that 3 cycles are required for the last elements to enter the $s \times s$-memory: In the first stage (a) the elements enter the I/O buffer. In the second stage (b) the elements are scattered by the *non-zero locator* to their corresponding positions and set the *non-zero indicator* accordingly. In the last stage the elements are inserted in the corresponding row in the $s \times s$-memory. Similarly, 3 cycles are needed for the last results to be returned to the vector register.

Having described the main code for the transposition of the $s \times s$-block we continue here our description of Figure 6. Lines 11-23 are excuted only if the s^2-block is not at the lowest level, i.e., the elements contain pointers to lower level s^2-blocks that also need to be transposed. In lines 11-18 the vector associated with the current block and contains the lengths of the s^2-blocks one level below is treated in the same way as the elements in lines 2-9 in order to have the correct length corresponding to the already transposed element. Subsequently in lines 19-23 the function *transpose_block* is called again with parameters that correspond to the s^2-block one level below. This is repeated for all pointers of the current s^2-block. We remark here that the amount of overhead that is induced by the extra processing needed for the higher levels is small since the number of high level s^2-blocks amount typpically to about $2 - 5\%$ of the total matrix storage for $s = 64$.

The CRS format is depicted graphically in Figure 8 and

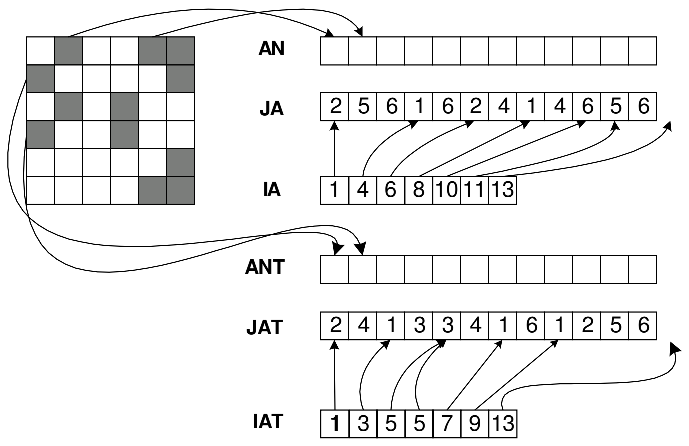


Fig. 8. Compressed Row Storage (CRS) format

```
1  for (each column $i$)
2        compute number of its non-zeroes, store it in $IAT[i]$.
3  Scan-add array $IAT$ : $IAT[i] = \sum_{j=1}^{j\leq i} IAT[j]$
4  for (each row $i$) do
5      $iaa = IA(i);\ iab = IA(i+1) - 1;$
6      for ($jp = iaa;\ jp \leq iab;\ ++jp$) do
7          $j = JA(jp) + 1;$
8          $k = IAT(j);$
9          $JAT(k) = i;$
10         $ANT(k) = AN(jp);$
11         $IAT(j) = k + 1;$
12     od
13 od
```

Fig. 9. Transposition for CRS.

is briefly described as follows. The non-zero elements of the matrix are all stored in a long continous vector (called the *Array of Non-zeros* (AN)) in a row-wise fashion. An equally long second array JA stores the column position of each of the non-zero elements in the AN (in the figure, columns and rows are numbered from 1, not from 0). Last, an *Index Array* (IA) of length M (the number of rows in the matrix) holds pointers to positions in the AN and JA that represent the first non-zero element in each row in the matrix.

The goal of a transposition algorithm for CRS is to build for the transposed matrix the array of non-zeroes, the column position array, and the index array, which are denoted as *ANT*, *JAT*, and *IAT*, respectively. For our experiments we have employed the standard CRS transposition algorithm described by S. Pissantetsky [10], a simplified pseudo-code of which is depicted in Figure 9. Below, we sketch its main parts and how they have been vectorized.

The first *for*-loop (lines 1–2) computes the number of non-zero elements in each column. Before this computation is started, elements of *IAT* are initilized to zeroes. This operation is easily vectorized, being translated into a sequence of vector stores. The computation of $IAT[i]$, i.e., the number of non-zeroes in column i, can be vectorized as follows: first, a mask vector $M_i[j]$ is generated, so that $M_i[j] = 1$ iff $JA[j] = i$. This can be done by means of vector compare operations. The required number of non-zeroes $IAT[i]$ is simply equal to the sum of all the M_i's elements. This accumulation can be vectorized, e.g., based on ideas presented in [11]. We remark, however, that because the matrix is sparse, the dominant part of M_i's elements will be zero and vector operations will be, therefore, inefficient. For this reason we have not vectorized this code for our experiments but translated it to the scalar instructions. These instructions are then executed by the baseline 4-way issue superscalar processor simulated by SimpleScalar.

Although the scan-add operation, which is performed on *IAT*, seems to be sequential at a first glance, it can be vectorized using, for example, the algorithm proposed by Wang et. al. [11]. The final part of the CRS transposition algorithm consists of two nested *for*-loops. The outer loop (starting at line 4) loads at each iteration i the interval of the column index vector *JA* corresponding to the i'th row. The variables iaa and iab denote the indices of the first and the last element in the interval, respectively. The inner loop processes the loaded interval element by element. The variable j computed in line 7 is the (column) index of the currently processed element increased by one. It means that the corresponding non-zero element belongs to the j'th *row* of the transposed matrix A^T. In line 8 the pointer k to the beginning of this row is computed. Since the element is in the i'th row of A, its column position in A^T is equal to i, and this value is filled for it at the corresponding position of *JAT* in line 9. In line 10 the value of this non-zero element is filled in *ANT*. Finally, in line 11, the row pointer for A^T is incremented so that it points to the next positions to be filled in *JAT* and *ANT*.

The pseudo-assembly code for the vectorized version of the body of the described loop nest, omitting the loop control instructions, is as follows.

```
v_ld          VR0, 4(&JA)           % 7
v_ld_idx      VR1, VR0, 4(&IAT)     % 8
v_setimm      VR2, i                % 9
v_st_idx      VR2, VR1, &JAT        % 9
v_ld          VR3, 4(&AN)           % 10
v_st_idx      VR3, VR1, &ANT        % 10
v_add_imm     VR1, 1                % 11
v_st_idx      VR1, 4(&IAT)          % 11
```

Here, the symbol % denotes comments and the number at the end of each line shows for each instruction the corresponding line in the algorithm depicted in Figure 9.

We have described the simulation environment, processor parameters, and the algorithms, The performance of the sparse matrix computations, however, also strongly depends on the input matrices.

B. Input Matrices: The Sparse Matrix Suite

We use the matrix set provided by the *Delft Sparse Architecture Benchmark (D-SAB)* suite [12] for our experiments. The *D-SAB* benchmark matrices were chosen from a wide variety of matrices that are available from the Matrix Market Collection [13]. 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 three 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.

Sorting of the selected 132 matrices with each of the three mentioned criterions resulted in three sets. From each of these sets ten matrices have been chosen with the equal steps (in logarithmic scale[2]) between their corresponding parameters. The result is a manageable but very diverse set of 30 benchmark matrices to be used in the simulations. For more detail on the selected matrices please refer to [12].

[2]The logarithmic scale was chosen because we observed that the distribution of paramters after sorting was logarithmic rather than linear

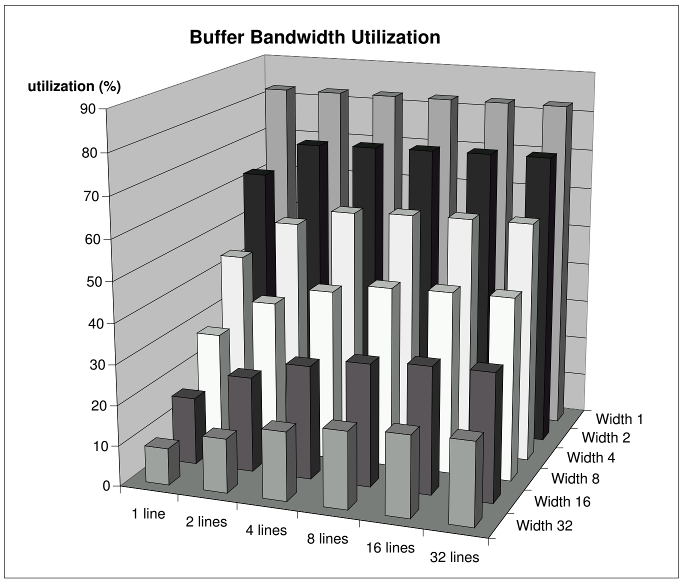


Fig. 10. Matrix Transposition

C. Buffer Bandwidth Utilization

Before presenting the performance results for CRS and HiSM implementations of transpose, we address the following issue. We remark that the I/O-buffer in the proposed mechanism can only contain elements that belong to the same row. Consequently, depending on the number of non-zero elements per row of an s^2-block, the buffer could be underutilized, especially for large buffer bandwidths. To avoid such an issue, we have additionally developed a version of the transpose mechanism for HiSM where multiple rows can be inserted at a time provided that these rows are consecutive. Since this extention is more costly in hardware, we have studied the impact of the number of accessible lines (i.e., rows/columns) L on the buffer bandwidth utilization, which is defined as the ratio of the achieved buffer bandwidth to the maximum buffer bandwidth:

$$BU = \frac{Z/C}{B} \qquad (1)$$

where Z is the number of non-zero entries in all the s^2-blocks, C is the execution time in cycles, and B is the bandwidth of the unit.

The results have been averaged for the total of 30 benchmark matrices and are depicted in Figure 10. The highest utilization is obtained for buffer bandwidth $B = 1$. The reason that the utilization is not 100% is that during transposition of each s^2-block a penalty of 6 cycles is payed, 3 cycles at the startup to and 3 at the end of block processing. Furthermore we observe that for increasing number of acceesible lines L the utilization increases. However, we remark that for a number of accessible lines $L > 4$ the

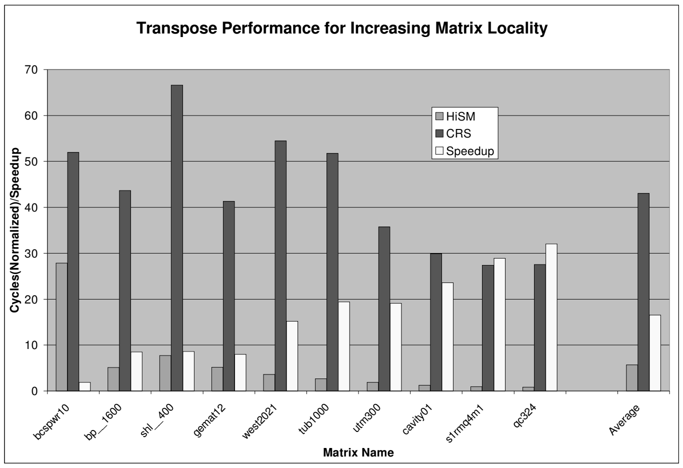


Fig. 11. Performance w.r.t matrix locality

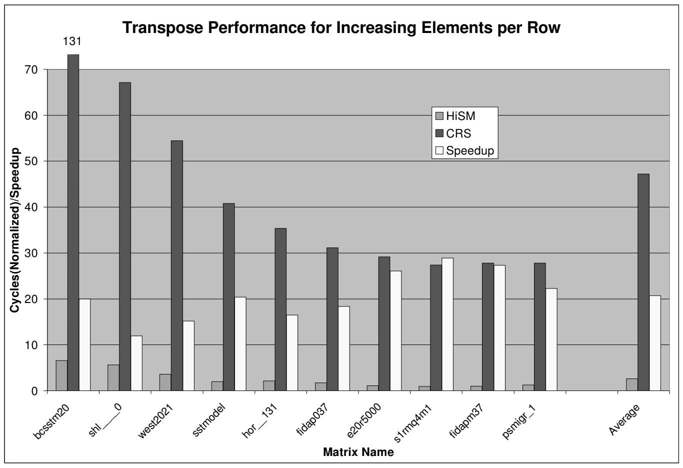


Fig. 12. Performance w.r.t. average number of nonzeroes per row

utilization does not increase significantly any more. Therefore, we have decided to use $L = 4$ as the number of accessible lines for the transposition mechanism and we will assume this value for further experimnets.

D. Performance Results

In Figures 11, 12, and 13 we depict how performance of CRS and HiSM schemes depends on the the matrix locality, the number of non-zeroes per row, and the size respectively. For each matrix, three values are presented. The first two bars show for each of HiSm and CRS the number of cycles needed to perform the transposition of a matrix normalized to the number of non-zero elements. This value is, essentially, the average number of cycles needed to process a non-zero element and, therefore, illustrates the efficiency of each algorithm. The third bar denotes the speedup achieved by HiSM with respect to CRS.

We observe that for all matrices HiSM consistently outperforms CRS. For the matrices from the first set (selected

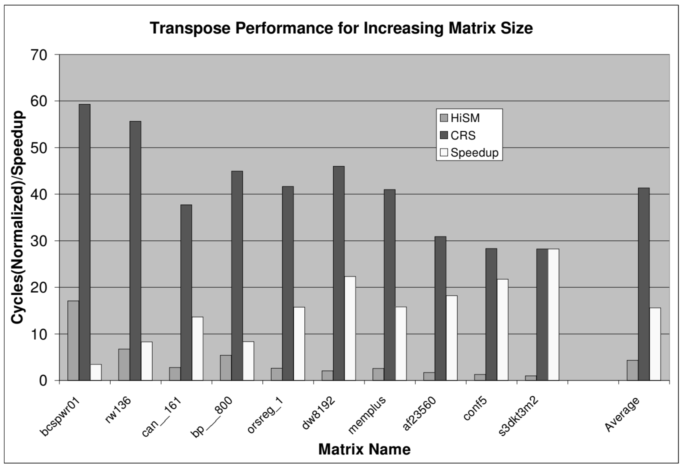


Fig. 13. Performance w.r.t. matrix size

according locality), the speedup is in the range from 1.8 to 32.0 with an average of 16.5. We remark that the speedup grows monotonically with the growth of the matrix locality. This is to be expected, since the locality represents the density of non-zeroes in a block. An increase in this density increases the average number of non-zeroes in a block and, consequently, the efficiency of the proposed HiSM transposition mechanism, which is specifically has been designed to operate on blocks. The CRS approach is row-oriented and, therefore, its performance does not show such a clear dependency on the locality.

On the other hand, when the average number of non-zeroes per row (*ANZ*) increaes, the performance of the CRS approach also increases, as shown in Figure 12. This effect is quite expectable. The performance behaviour for HiSM scheme depending on the ANZ value cannot be observed from the figure due to small absolute values. In fact, it is as follows. For the first several matrices (from *bcsstm20* to *s1rmq4m1*) it decreases almost monotonically, and after this remains constant. This effect, probably, has the following explanaition. In general, the sparsity pattern for the matrices in the second set is such that their locality correlates with *ANZ* and grows together with it, resulting in improved performance of HiSM, since that one increases with an increase in locality. For the matrices from the second set, which have been selected according to the *ANZ* value. the HiSM vs. CRS speedup ranges from 11.9 to 28.9 with an average of 20.0.

In Figure 13 we depict the transposition performance on the third set of matrices, which have been selected according to the matrix size. The performance of both CRS and HiSM does not show any particular dependence on the matrix size. After studying the other two parameters for the matrices in this set, the average number of non-zeroes *ANZ* and the locality (see [14]), we observed the following. The performance of each of both methods behaves consistently with the observations made above: the CRS performance increases together with *ANZ* value for a transposed matrix, while that of HiSM increases when the locality of the matrix is increased. In this sense, Figure 13 does not provide any new insights into CRS of HiSM and is presented here for completeness and consistency with the organization of the benchmark matrix collection. The speedup of HiSM vs. CRS for the matrices from the third set ranges from 3.4 to 28.2 with an average of 15.5. Finally, considering the whole collection of 30 matrices we observe that the speedup ranges from 1.8 to 32.0 with an average of 17.6.

V. Conclusions

In this paper we have proposed and described a novel mechanism for transposition of the sparse matrices stored according to the HiSM sparse matrix storage format. An example implementation of mechanism as functional unit for a vector processor is presented. We have calculated the optimal parametrs for the mechanism and compared the performance of the vector processor extended with the proposed functional unit on the transposition of HiSM-stored matrices with that of the standard vector processor performing the transposition of the matrix stored according to the popular CRS format. Using the proposed approach, the HiSM-based transposition algorithm exhibits the average speedup of 17.7 with when compared to the CRS-based algorithm. Finally, we observe that the performance of the HiSM-based transposition correlates with the matrix locality, which indicates the density of non-zeroes within square blocks, and grows when this density increases.

References

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

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

[3] 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*, Boston, Massachusetts, june 1985, IEEE Computer Society TCA and ACM SIGARCH, pp. 100–107.

[4] Valerie E. Taylor, Abhiram Ranade, and David G. Messerschitt, “SPAR: A New Architecture for Large Finite Element Computations,” *IEEE Transactions on Computers*, vol. 44, no. 4, pp. 531–545, April 1995.

[5] P.T. Stathis, S. Vassiliadis, and S. D. Cotofana, “A hierarchical sparse matrix storage format for vector processors,” in *Proceedings of IPDPS 2003*, April 2003, pp. 61–61.

[6] 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 *Proceedings of the 15th Annual International Symposium on Computer Architecture*, H. J. Siegel, Ed., Honolulu, Hawaii, may ”–” jun 1988, pp. 212–222, IEEE Computer Society Press.

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

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

[9] 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.

[10] S. Pissanetsky, *Sparse Matrix Technology*, Academic Press, 1984.

[11] H. Wang, A. Nicolau, S. Keung, and K.-Y. Siu, “Computing programs containing band linear recurrences on vector supercomputers.,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, pp. 769–782, 1996.

[12] P. Stathis, S. Vassiliadis, and S. Cotofana, “D-sab: A sparse matrix benchmark suite,” in *Proc. of 7th Int. Conf. on Parallel Computing Technologies (PaCT)*, 2003, pp. 549–554.

[13] R.F. Boisvert, R. Pozo, K. Remington, R. Barrett, and J.J. Dongarra, “The Matrix Market: A web resource for test matrix collections,” in *Quality of Numerical Software, Assessment and Enhancement*, Ronald F. Boisvert, Ed., London, 1997, pp. 125–137, Chapman & Hall.

[14] P. Stathis, Stamatis Vassiliadis, and Sorin Cotofana, “D-sab: Delft sparse architecture benchmark, http://ce.et.tudelft.nl/˜pyrrhos/d-sab/,” 2003.