Sparse Matrix Vector

Processing Formats

Sparse Matrix Vector Processing Formats

PROEFSCHRIFT

ter verkrijging van de graad van Doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op vrijdag 19 november om 10:30 uur

door

Pyrros Theofanis STATHIS

electrotechnisch ingenieur
geboren te Athene, Griekenland

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. S. Vassiliadis

Samenstelling van de promotiecommisie

Rector Magnificus:	voorzitter
Prof. dr. S. Vassiliadis	Technische Universiteit Delft, promotor
Prof. dr. N. Dimopoulos	University of Victoria
Prof. dr E. Deprettere	Leiden University
Prof. ir. G. L. Reijns	Technische Universiteit Delft
Prof. dr. Leonel de Sousa	Instituto Superior Technico
Dr. S. Cotofana	Technische Universiteit Delft
Dr. K. Bertels	Technische Universiteit Delft
Prof. dr. C. I. M. Beenakker	Technische Universiteit Delft, reservelid

ISBN 90-9018828-2

Cover Page by Faethon Stathis

Keywords: Vector Processor, Sparse Matrix, Storage Formats

Printed in the Netherlands

to the memory of my mother,
Andriëtte

Sparse Matrix Vector Processing Formats

Pyrros Theofanis Stathis

Abstract

In this dissertation we have identified vector processing shortcomings related to the efficient storing and processing of sparse matrices. To alleviate existent problems we propose two storage formats denoted as Block Based Compression Storage (BBCS) format and Hierarchical Sparse Matrix (HiSM) storage. Furthermore we propose vector architectural instruction set extensions and microarchitecture mechanisms to speed up frequently used sparse matrix operations using the proposed formats. Finally we identified the lack of benchmarks that cover both format and sparse matrix operations. We introduced a benchmark that covers both. To evaluate our proposal we developed a simulator based on SimpleScalar, extended so that it incorporates our proposed changes and established the following. Regarding storage space our proposed formats require 72% to 78% of the storage space needed for Compressed Row Storage (CRS) or the Jagged Diagonal (JD) storage, both commonly used sparse matrix storage formats. Regarding Sparse Matrix Vector Multiplication (SMVM) both BBCS and HiSM achieve a considerable performance speedup when compared to CRS and JD. More in particular, when performing the SMVM using the HiSM format and the newly proposed instructions we can achieve a speedup of 5.3 and 4.07 versus CRS and JD respectively. Additionally, the operation of element insertion using HiSM can be sped up by a factor of 2-400 depending on the sparsity of the matrix. Furthermore, we show that we can increase the performance of the transposition operation by a factor of 17.7 when compared to CRS.

Acknowledgments

First of all, I would like to thank Stamatis for giving me the opportunity to start my PhD in the Computer Engineering Laboratory. I am very grateful for his insistence, perseverance and commitment which greatly motivated me throughout the duration of my studies. I have very fond memories of our conversations which covered every conceivable subject, not always scientific.

I would also like to thank Sorin for his good advice, patience in correcting my papers and for the enjoyable philosophical discussions.

Many thanks go to my former roommates, Stephan, Casper, Dmitry and Elth, who have endured my presence and always laughed at my bad jokes.

Special thanks go to Bert, our system administrator, who was very helpful and always responded to my ever demanding requests.

To my colleagues Jeroen, Dan, Gabi, Said and Asad and the rest of the team, you have my thanks and appreciation for making the Computer Engineering department a fun place to be.

My stay in Delft would have be very boring without my friends who made this time unforgettable. My special thanks go to Sotiris, Ferdinand, Janna, Yiannis, Vitke, Aletta, Eleni, Herre, Ida and Nazma.

Last but not least, I am eternally grateful to Faethon, the best brother one could wish for, and to my parents for delivering me on this rock in space.

Contents

List of Figures

Chapter 1

Introduction

A large portion of scientific applications, for which expensive supercomputers are often bought, involve sparse matrices. Sparse matrices are a special flavor of matrices characterized by the proportionally small amount of non-zero elements they contain. Despite the fast pace of developments and innovation in computer architecture, performing sparse matrix operations on any type of computer today is still routinely expected to be an inefficient process that utilizes about 10-20% of peak performance [73] (See also Tables 1.1 and 1.2). The reason is that despite the regularity of operations involved, the sparsity patterns (i.e. distribution of non-zeros) within the matrix induce a data access irregularity that can be highly detrimental for the performance. In literature we can find numerous methods and schemes that attempt to increase the efficiency of sparse matrix operations. However, most efforts are focused on the software approach of the problem and only few have approached the issue from the hardware efficiency and performance point of view.

In this thesis we consider sparse matrices from the architectural and hardware design points of view. Our main focus is to provide hardware mechanisms that will improve sparse matrix computations. We present an approach that consists of an architectural extension to a vector processor and a sparse matrix storage format which were co-designed to alleviate the problems arising in sparse matrix computations. In this chapter we provide general information regarding the topic we investigate, the kinds of challenges we address and the organization of the discussion. More specifically in this chapter we discuss motivation, background information and open problems. The discussion is organized as follows: In Section 1.1 we will give some background information regarding vector processors, why we have chosen the vector processor paradigm for our proposal and discuss the shortcomings of vector processors.

Processor	Year	MHz	Peak Mop/s	SMVM Mop/s	Tuned Mop/s	Source
Intel i860	1987	5	23	5	8	[55]
DEC 21164	1994	500	1000	43	90	[37]
MIPS R8000	1994	76	300	-	39	[6]
HP PA-7200	1994	120	240	13	22	[77]
DEC Alpha 21164	1995	500	1000	43	58	[32]
Sun Ultra 1	1995	143	286	17	22	[66]
IBM PowerPC 604e	9/1995	190	190	20	25	[38]
IBM Power2	1996	66	266	40	100	[66]
MIPS R10000	1996	250	500	45	90	[65]
HP PA-8200	1996	240	480	6	8	[44]
IBM Power2	1997	160	640	56	140	[32]
IBM Power3	1997	375	1500	164	240	[74]
Intel Pentium II	9/1997	266	266	11	11	[10]
Sun Ultra 2i	1998	333	666	36	73	[74]
MIPS R12000	1998	300	600	94	109	[46]
DEC Alpha 21264a	1999	667	1334	160	254	[46]
Intel Pentium 4	2000	1500	3000	327	425	[74]
Hitachi SR-8000	2001	250	1000	45	145	[47]
IBM Power4	2001	1300	5200	595	805	[74]
Intel Itanium	2001	800	3200	120	345	[74]
Sun Ultra 3	2002	900	1800	53	108	[74]
Intel Itanium 2	2002	900	3600	295	1200	[74]

Table 1.1: Sparse Matrix Vector Multiplication (SMVM) performance on various scalar microprocessors [73]

Processor	Year	MHz	Peak Mflop/s	SMVM Mflop/s	Tuned Mflop/s	Source
Cray Y-MP	1988	-	333	127	127	[6]
Cray EL	1989	-	133	33	33	[6]
Cray C90	1992	240	960	120	234	[9]
Fujitsu VPP500	1994	100	1600	267	267	[6]
Cray SV1	1996	500	2000	125	125	[44]
NEC SX-4	1996	125	2000	600	675	[44]
Fujitsu VPP5000	1999	303	9600	1405	1881	[47]
NEC SX-5	2000	250	8000	1100	1200	[44]
Hitachi SR-8000	2001	250	1000	68	174	[47]
Japan Earth Simulator	2002	500	8000	1750	2375	[47]
NEC SX-6	2002	500	8000	620	620	[49]
UCB VIRAM	2003	200	1600	91	511	[49]

Table 1.2: Sparse Matrix Vector Multiplication (SMVM) performance on various vector processors [73]

Section 1.2 discusses related work on sparse matrices and associated operations and we give a problem definition dealt with in this dissertation. Finally, in Section 1.3 we outline the structure of this document.

1.1 Vector Processors

Our approach to tackle the problem of low efficiency on sparse matrix operations is based on extending vector processor architectures and memory formats. In the remainder of this section we will give a brief overview of vector architectures, some recent developments in vector architectures and give some reasons why we have chosen to pursue a vector processing approach for our proposed solution.

The main characterizing property of Vector Architectures is that they provide *vector instructions* to process arrays of data rather than processing one element at a time [33, 42]. This reduces the instruction control overhead and introduces a framework for optimizing the processor for array processing. Vector Processors (VPs) were introduced in the 70s with the appearance of the CDC Star-100 [34] and Texas Instruments Advanced Scientific Computer [76]. Both were memory-to-memory architectures, i.e., the vector instructions operated directly on the arrays in memory and streamed the result back to memory. The assumption behind such an approach was that the low locality of data would not create any need for caches or any form of intermediate storage. Additionally, the large data sets, often encountered in scientific computing, would amortize the array processing startup delays. Indeed, these machines could deliver a very high throughput when processing large dense arrays. However, poor scalar performance resulted in too much time spent on the non-vectorized part of the application.[1] Moreover, the long memory latencies tended to dominate the execution time since few applications are only processing very large arrays of data. Additionally the need to store intermediate results of more complex vector operations in memory further decreased flexibility and performance. Vector processors really gained acknowledgment by the introduction of CRAY-1 [54]. Cray-1 was the first register-to-register vector architecture, i.e., all vector instructions act on fast, limited size vector registers. The Vector Register (VR) file consists of a number of registers that can store arrays (vectors) of elements, typically 8-32 VRs of 32-256 elements. The introduction of VRs increased the flexibility of the processor and significantly reduced memory latency. Learning from the mistakes of the creators of the earliest vector machines, much attention was put on the performance of the scalar[2] process-

[1] This clearly demonstrates the generalization of Amdahl's law [1] that states that if a portion $0 < p < 1$ of the total execution time T of a program can be executed n times faster, then the total execution time becomes $T_{new} = T((1-p) + \frac{p}{n}$, i.e., unless p is high the speedup will have a small effect on the total speedup.

[2] A *vector* is a linearly ordered collection of data items. A *scalar*, in contrast, is a single item. The term *scalar* is also used here to mean *nonvector* [12]. Thus scalar section of the

ing section in CRAY-1. Subsequent vector architectures were largely based on this vector architecture paradigm. From this point on we will consider only register-to-register vector architectures.

In a register-to-register architecture the arrays are first loaded in the registers before they are operated upon. As was mentioned the VRs contain typically 32-256 each. This is called the *section size* (denoted s throughout this document) or *maximum vector length*. When an array which is to be processed is longer than the section size, then the array is sectioned in vectors of size s and each section is processed separately. The workings of a typical vector processor can be best illustrated by the use of an example. Consider we have allocated three arrays in memory, `A[]`, `B[]` and `C[]`, each having length 500 elements, and we want to perform the following operation: $\texttt{C}[\texttt{i}] = \texttt{A}[\texttt{i}] + \texttt{A} \times \texttt{B}[\texttt{i}]$ The vector code to achieve this is as follows (The example code that displayed here is similar to the vector code example in [12]):

```
(1)        LA    @A, R1
(2)        LA    @B, R2
(3)        LA    @C, R3
(4)        ADDI  R5, 500
(5)  LP:   SSVL  R5
(6)        LDV   R1, VR1
(7)        LDV   R2, VR2
(8)        MULVS VR1, VR1, R3
(9)        ADDV  VR3, VR2, VR1
(10)       STV   R3, VR3
(11)       BNE   R5, LP:
```

The first four instructions initialize the memory pointers in the registers. Instruction (5) Sets the vector length. In case the starting value of `R5` is larger than the section size s the vector length becomes equal to s and `R5` is set to the previous value minus s. when after some iteration the value of `R5` becomes smaller than s the vector length is set to `R5` and `R5` is set to zero. In the loop the instructions (6)-(7) perform the vector loads into the vector registers and instructions (8)-(9) perform the actual operations on the vectors. Subsequently, instruction (10) stores the result back to memory. Finally, instruction (11) repeats the loop until all 500 elements are processed.

Since their appearance, vector processors have traditionally been chosen for large scientific applications (supercomputing) due to their ability to process large amounts of data, both because they support vector operations but

vector processor is the section that does deal with vectors.

also because of the high memory bandwidth that many of the models offer. Since their introduction, the peak performance of vector processors has been steadily increasing from 160Mflop/s for CRAY-1 to 8Gflop/s for NEC SX-6. However, by the end of the 90s the area of supercomputing has seen a shift of the supercomputing field favorite architectures from vector processors toward parallel multiprocessors such as the CRAY T3D [41] which use of-the-shelf microprocessors. The main reason for this, we believe, is that vector processors today do not offer the flexibility for efficient execution of complex functions or operations on vectors.[3] Instead they are limited only to simple operations on large homogeneous data sets that can be accessed in a regular fashion. At the same time, microprocessors amortize the lack of a reduced instruction control overhead and fast vector pipelines by offering flexibility through out-of-order execution, caches, register renaming etc., which allow them to execute more complex functions which cannot be vectorized for current vector architectures. Here we have to note that a number of techniques to make vector processing flexible already exist. Such techniques are chaining, gather/scatter instructions and conditional execution (See also Chapter 3). However, these techniques haven't changed since the introduction of CRAY-1. Having observed that, we believe that in order to make vector processing more efficient we should make the processing of vectors within the processor more flexible. A number of other research efforts are pointing in that direction:

Espasa attributes the decline of vector processors as an architecture of choice for scientific applications to (a) the high price/performance of modern microprocessors which make use of (b) new techniques such as out of order execution, caches and instruction level parallelism while (c) the vector architecture paradigm has not seen many innovations since its introduction [27]. Subsequently, a number of well researched and currently used techniques, which are traditionally developed for superscalar processors, were successfully applied to vector processing. Such applied techniques include out-of-order execution [29], decoupling [28] and multithreading [30].

In [2], the author presents a single chip vector microprocessor with reconfigurable pipelines [4]. Additionally a vector flag processing model which enables speculative vectorization of "while" loops is described. Both aim to increase the flexibility of vector operations. Furthermore, in a related

[3] By *flexibility* we mean the ability of a vector processor to process arrays of data effectively with vector instructions when more complex operations are involved, instead of having to resort to scalar instructions because the complexity cannot be handled (efficiently) with vector instructions. Hence, we define flexibility to be the amount of operations on arrays that can be vectorized and this vectorization is yielding better performance than executing the same operation using scalar code given equivalent resources.

project [43], a vector processor is presented which is embedded in a DRAM memory to take advantage of the increased bandwidth such a configuration can offer. The advantages of vector computing could overcome the slower processing speed that can be attained by the integration of DRAM and logic on one chip.

In [15–19, 39] the authors describe a complex streamed instruction set where complex (in this case media related) vector instructions can act on long arrays of data which are streamed from and to memory. This is in essence a memory-to-memory architecture where the long media arrays amortize the startup latency costs. Furthermore, the implementation of pipelines that can perform complex operations ensure no additional latency for intermediate results that might need to be stored. We observe, especially by the last aforementioned approach, that we can alleviate the vector processing shortcomings, even of the memory-to-memory architectures if we can provide long vectors, or at least long enough to be able to hide the memory latency, and provide with instructions (and vector functional units, or pipelines) that support more complex instructions than those available now. This observation will form the basis of our approach for tackling the shortcomings of sparse matrix operations on vector processors.

1.2 Related Work and Problem Definition

Sparse matrices are (usually large) matrices that contain a relatively small amount of non-zero elements as opposed to *dense* matrices that can be considered to have non-zero elements on every position of the matrix. Since only a few zero elements in a matrix does not make it a sparse matrix, we need a more clear definition of a sparse matrix. We define a matrix as *sparse* when it is advantageous performance-wise to store and operate on the matrix in a way that takes in account only the non zero elements rather than all elements as is for dense matrices. Sparse matrices are involved in a wide variety of scientific applications such as Structural Engineering, Fluid Flow Modeling, Economic Modeling, Chemical Engineering etc. More specifically they arise from fields such as Finite Element Methods [36] and Partial Differential Equations. A example of a Sparse Matrix is depicted in Figure 1.1. Typically (see Figure 1.1), sparse matrices contain a low percentage of non-zeros which are fairly evenly distributed along the two dimensions, and usually very few non-zero elements per row.

As we mentioned it is advantageous to treat a sparse matrix taking into account the large amount of zeros it contains. The most common way to achieve

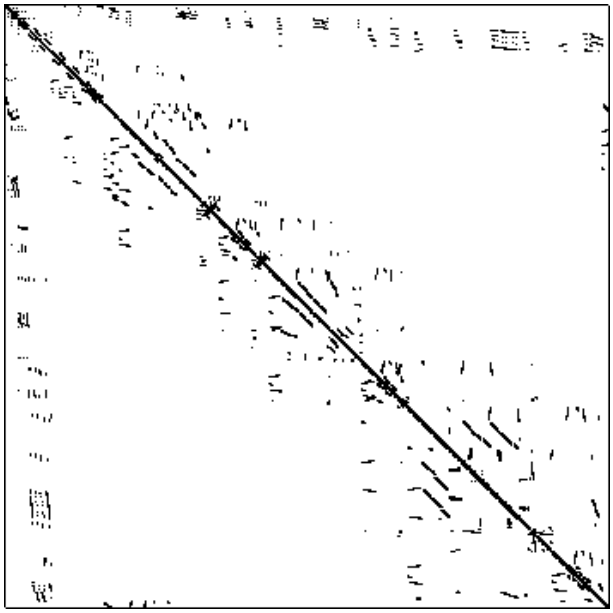


Figure 1.1: Example of Sparse Matrix. Name: dwt-2680, Dimensions: 2680×2680, 25026 non-zero elements, 9.3 non-zero elements per row, Source:[23]

this is to device special *sparse matrix storage formats* [7] whose main common characteristic is that the non-zero elements are explicitly stored whereas the zeros are implied and not stored.[4] Doing so (a) decreases the storage space and the bandwidth requirements for storing and accessing the matrix respectively and (b) allows avoiding the trivial operations on zero elements. However, this approach induces a higher complexity of the storage format when compared to the dense case. Consequently, such mechanisms may have a negative impact on performance (relative to the peak performance of a processor). Moreover, the functionality offered by current processor architectures does not dictate a straightforward way of constructing an efficient sparse matrix storage format. The result is that there are many formats to choose from [26, 59] (See also Chapter 2 for more information on formats). A number of these formats are *general type formats*, like the Compressed Row Storage (CRS) format, which make no assumption of the structure of distribution of non-zeros. Others are

[4]The separation of zeros and non-zero storage is not always strict. In some cases (See also Section 2.1.3, a storage format may store some zeros explicitly in order to maintain the regularity of the non-zero storage structure

optimized for specific types of matrices that mostly arise from a particular field. An example is Block Compressed Row Storage which is efficient for block matrices that typically arise from the discretization of partial differential equations. A number of storage formats, such as for instance the Jagged Diagonal (JD) Storage [56] (See Section 2.1.2), are designed for specific operations and/or specific architectures (in the JD case, designed for sparse matrix vector multiplication in vector processors).

We note that previously introduced storage schemes leave open several questions regarding performance and storage efficiencies. Current storage formats, described in Chapter 2 are either too specific for their application area and types of matrices (non-general) or suffer from several drawbacks, discussed in detail in Chapter 2, including the following:

- **Short vectors**: This problem is particularly evident in CRS but it is a more general phenomenon that is directly related to the fact that most sparse matrices only have a small number of non-zero elements in each row or column. By "small" we mean that when we construct a vector using the non-zero elements in a row and operate on it, the overhead cost is significant. The JD storage format successfully deals with this problem when the matrix has sufficiently large dimensions since we obtain vector of length on the order of the dimension of the matrix. However it does not deal with the other problems (see next bullets), moreover it is specially designed for sparse matrix vector multiplication and not efficient in other operations.

- **Indexed Accesses**: This problem is particular to JD and CRS, both general storage formats. The problem related to the fact that since no assumption is made of the sparsity pattern, the positional information of each non-zero element (e.g. the column position in CRS and JD) is stored explicitly. This results in indexed memory accesses whenever we need to access other elements using this positional information. This problem is mainly addressed by non-general storage formats such as BCRS and CDS that partly store data with implicit positional information.

- **Positional Information Storage Overhead**: This problem is related to the previously described problem. However, the focus here is on the overhead of storing the positional information. Formats like CRS and JD store the column position for each element (the row position is implied by the index we use to access the row) which requires a full word extra storage for each non-zero element. However, as we'll show later

in Sections 2.2.1 and 2.3, it is possible to device a format in such a way that the positional information storage is decreased without losing the general applicability of the format.

In this dissertation we will provide a framework consisting of new sparse matrix storage formats, organizational extensions and new instructions for vector processors that aim to alleviate the aforementioned problems that arise when using existing sparse matrix storage formats.

1.3 Framework

This section discusses the organization of the remainder of this dissertation which consists of the following chapters:

- In Chapter 2 we discuss a number of the most popular sparse matrix storage formats currently used in related applications. The emphasis lies on the general type formats, the Compressed Row Storage (CRS) and Jagged Diagonal (JD) Storage, formats that make no assumption of the sparsity structure of the matrix, since this type is also our target group. We proceed by indicating a number of shortcomings of the existing formats, such as the forming of short vectors, induced indexed memory accesses, and positional information overhead. Subsequently, we propose two new formats, the Block Based Compression Storage (BBCS) and the Hierarchical Sparse Matrix storage format (HiSM). We describe how BBCS and HiSM can alleviate the aforementioned shortcomings and offer a potentially increased performance. In the remainder of the chapter we provide with a quantitative comparison of the proposed formats with CRS and JD. Compared are the Storage Space and Vector Filling achieved by the formats.

- In Chapter 3 we propose a number of architectural extensions based on the vector processing paradigm that aim to provide support for the efficient execution of sparse matrix operations using the sparse matrix storage formats presented in Chapter 2. The proposal consists of a number of new functional units and vector instructions which can process data presented in the BBCS and HiSM format. The chapter starts by providing the general framework on which the extension is based. Subsequently, the BBCS and HiSM extensions are described. First, the vector load/store instructions that provide access to data stored in BBCS and HiSM formats and instructions to process the data, such as the `MIPA`

instruction that can perform the sparse matrix multiplication of a matrix block with a dense vector. We continue by presenting a number of functional units, such as the `MIPA` unit, that are designed to execute the aforementioned instructions.

- In Chapter 4 we present a set of benchmark algorithms and matrices, called the Delft Sparse Architecture Benchmark (D-SAB) Suite for evaluating the performance of novel architectures and methods when operating on sparse matrices. The focus is on providing a benchmark suite which is flexible and easy to port on (novel) systems, yet complete enough to expose the main difficulties which are encountered when dealing with sparse matrices. The chapter starts by describing the previous art in the field and we proceed with description of D-SAB that consists of 2 main parts: (a) a set of ten algorithms which are divided in five value related operations and five position related operations and (b) a set of representative matrices, carefully selected from a large existing sparse matrix collection, which capture the large diversity of sparsity patterns, sizes and statistical properties.

- In Chapter 5 we provide with some experimental results that show the performance gain that may be achieved when we utilize our proposed formats on an architecture extended with the architectural augmentations presented in Chapter 3. First we describe the simulation environment, a scalar processor simulator that has been extended to support basic vector processor functionality as well as the architectural augmentations which were described in Chapter 3. We show that by using the newly proposed formats in conjunction with an augmented architecture we can obtain significant performance gain over existing formats like CRS and JD with similar resources. More specifically, we show that for the operation of sparse matrix vector multiplication using the HiSM scheme we can obtain a speedup 5.3 and 4.07 when compared to CRS and JD respectively. Furthermore, we show a varying speedup of the element insertion operation of 2 to 400 times when compared to CRS. For the operation of matrix transposition we show an average speedup of 17.7 when compared with the CRS scheme.

- Chapter 6 presents the conclusions of the present thesis and outlines the main contributions of the conducted research. Finally, a number of directions for future work are described.

Chapter 2

Sparse Matrix Storage Formats

The need to device special sparse matrix storage formats arises from the existence of a substantial number of zero elements and the triviality of operation with zero elements. As the percentage of non-zero elements in a matrix decreases there is a turning point where it is no longer efficient to operate on them since most operations are trivial operations with zeros. It then becomes more efficient to pay an additional cost for handling a more complicated storage scheme in order to avoid the trivial operations with zeros. It is this turning point that defines a matrix as sparse. Note that in this paragraph we use the term "efficient operation" in a more general sense, including the storage space and bandwidth benefits that such special storage formats offer. The main common property of all sparse matrix storage formats is that they try to minimize the amount of zeros stored and at the same time provide an efficient way of accessing and operate on non-zero elements. It should be noted that combining the elimination of zeros with efficient operation is not a trivial task since, in contrast to dense matrices, current memory organizations and processor architectures don't dictate a straightforward way of doing zero elimination and efficient operations. Our proposed sparse matrix storage formats, discussed in this chapter are an integral part of our proposal for increasing the performance of sparse matrix operations on vector processors.

This Chapter is organized as follows: In the first Section we will discuss previous art, comprising of the most important and widely used sparse matrix formats where we will focus mainly on the General matrix formats, i.e., matrices that are not specifically tied to a particular type of application, since we are targeting general matrices in this thesis. The following Sections, 2.2.1 and 2.3, will consist of our proposed formats, the BBCS and HiSM formats respectively. The following Section, Section 2.4 discusses a number of quantitative

analysis of the existing and proposed formats and in Section 2.5 we give some conclusions.

2.1 Previous Art

In this section we describe the most commonly used sparse matrix storage formats as they are described in the literature [7, 26, 59]. We will mainly focus on two general formats, the Compressed Row Storage (CRS) in Section 2.1.1 and Jagged Diagonal (JD) in Section 2.1.2. As mentioned earlier, General Type Matrices are matrices that are not related to any particular type of application and therefore do not assume any particularities in the sparsity pattern that may be exploited in order to be more efficient when storing or operating on them. Other, non general or not so widely used formats will be briefly discussed in Section 2.1.3.

2.1.1 The Compressed Row Storage

The Compressed Row Storage (CRS) format for spares matrices is perhaps the most widely used format when no assumption is made about the sparsity structure of the matrix. Assuming we have an $M \times N$ sparse matrix $A = [a_{ij}]$, containing NZ non-zero elements, the CRS format is constructed as follows:

- First a 1-dimensional vector AN is constructed that contains all the values of the non-zero elements a_{ij} of matrix A, taken in a row-wise fashion from the matrix. Thus, AN $= [v_k]$ where $k = 1 \ldots \text{NZ}$, $v_k = a_{ij}$ where a_{ij} is the k^{th} element of A for which holds $a_{ij} \neq 0$ and $i = 1 \ldots N, j = 1 \ldots M$.

- Secondly, a 1-dimensional vector AJ $= [c_k]$ of length equal to the length of AN is constructed that contains the original column positions of the corresponding elements in AN. Thus, $c_k = j$ where $k = 1 \ldots \text{NZ}$, j is the column position of the k^{th} element of matrix A for which holds $a_{ij} \neq 0$ $a_{ij} \neq 0$ and $i = 1 \ldots N, j = 1 \ldots M$.

- Last, a vector AI $= [r_l]$ of length $M + 1$ ($l = 1 \ldots (M + 1)$) is constructed. $r_{l-1} - r_l$ denotes the length of row l of matrix A and $r_1 = 1$. Each element in vector AI is therefore a pointer to the first non-zero element of each row in vectors AN and AJ.

In Figure 2.1 is depicted an example on how an 8×8 matrix containing 14 non-zero elements can be stored in the CRS format.

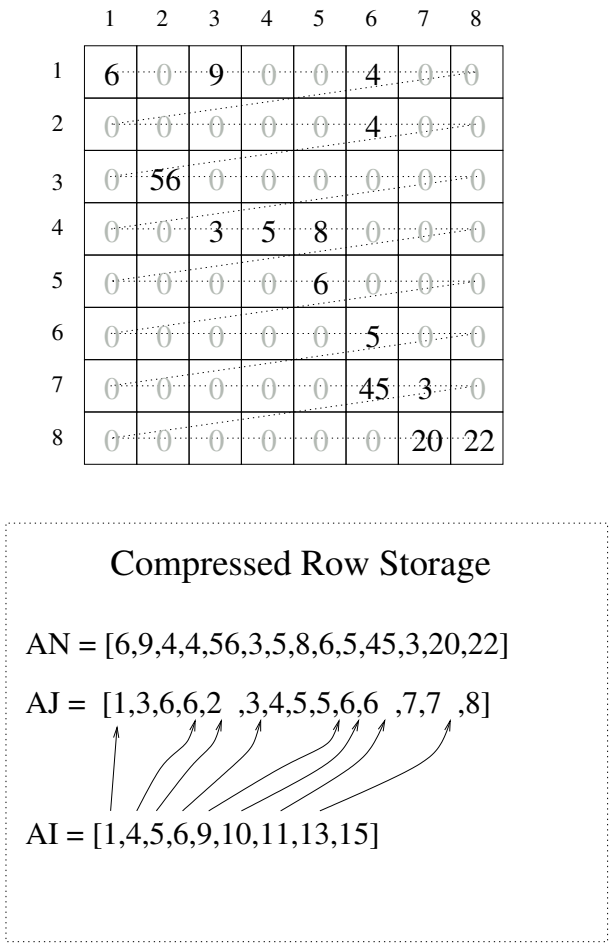


Figure 2.1: Compressed Row Storage Format

The main advantage of using this format is that it is rather intuitive and straightforward, and most toolkits support this format on most sparse matrix operations. However, since it is a general type matrix it does not take account of sparse matrix pattern particularities that can arise from specific types of applications, it does have several disadvantages:

- When accessing a matrix stored in CRS format one row at a time, we are constructing short vectors. This is related to the fact that most sparse matrices have only a small number of elements per row, irrespective of their size. This makes the overhead cost of starting a vector operation a significant part of the cost of operating on the matrix.

- When a non-zero element of the matrix is accessed, for many operations, such for instance a Sparse Matrix Vector Multiplication, this implies an indexed access in order to access a related element.

- For each single element we need to store its column position. This is directly related to the fact that no a priori assumption is made of the distribution of the non-zero elements within the matrix and any element can have an arbitrary position within the matrix. In contrast, some non-general formats do assume a certain distribution pattern and use this information and in doing so reduce the amount of positional information as we will see in Section 2.1.3.

Next to CRS, there exists a similar format, the Compressed Column Storage (CCS) (also called the Harwell-Boeing Sparse Matrix), which is constructed in exactly the same way as CRS but with the roles of rows and columns interchanged. This means that the sparse matrix is scanned column-wise to create AN, AI contains the corresponding row positions of the elements in AN and AJ contains the pointers to the first non-zero elements in each column. The main reason for using this format instead of CRS is that some programming languages, particularly FORTRAN, traditionally stores matrices column-wise rather that row-wise. Because of the similarities of these two formats we only focus on CRS on the remainder of this work.

2.1.2 The Jagged Diagonal Format

The Jagged Diagonal (JD) Format is a format that is specially tailored for sparse matrix vector multiplications. This operation often occupies the largest amount of computational time in sparse matrix applications (mainly due to iterative method solvers). This makes it an important operation that needs to be executed efficiently. The major problem that JD tackles is that of short vectors and is therefore particularly suited for vector processors and more generally SIMD machines. To store an $N \times M$ sparse matrix $A = [a_{ij}]$ with NZ non-zero elements in Jagged Diagonal format we proceed as follows:

- Step 1: All the non-zero elements are shifted left maintaining the order within each row, leaving the zeros to the right. This gives us a new matrix, which we will refer to as A_s with columns of decreasing number of non-zero elements per column. The number of non-empty columns will be equal to the number of non-zeros in the row with the maximum number of non-zeros. We will call this number NC. At the same time

we construct a new table, A_{cp} that contains the original column positions of the corresponding elements in A_s.

- Step 2: The rows of A_s and A_{cp} are permuted in such a way that we obtain a decreasing number of non-zero elements per row and we store the permutation order in a permutation vector (Perm).

- Step 3: The resulting 2 sets of NC vectors are then used to form two long vectors of length NZ starting from the longest column and ending with the smallest. The first vector, the Value (Val) vector, contains the values of the non-zero elements and the second, the Column positions vector (CP), contains the corresponding column positions. A third vector, the index (Ind) vector contains pointers to the beginning of the columns within V and C. The Val, CP, Ind and Perm together comprise the Jagged diagonal Storage.

In Figure 2.2 we depict the procedure of obtaining the JD storage by means of an example of an 8×8 sparse matrix. The main advantage that JD offers is that enables the operation on long vectors. The vectors operated upon will generally have the size in the order of N (the number of rows of the matrix). Hereby the problem of overhead cost of starting a vector operation is eliminated and this makes this method ideal for use in vector processors. However the problem of indexed memory accesses still remains due to the explicit positional information.

2.1.3 Other formats

In this subsection we describe two additional formats commonly used for sparse matrix storage. We first describe the Block Compressed Row Storage and the Compressed Diagonal Storage.

The Block Compressed Row Storage (BCRS): The BCRS format is non-general sparse matrix format which is used when the sparsity pattern of a sparse matrix is such that it comprises of dense sub-blocks whose density we can exploit by storing and treating the sub-blocks as small dense matrices (which might contain some zeros). These types of matrices usually are obtained with applications relating to the discretization of partial differential equations in which there are several degrees of freedom associated with a grid point [7].

This format is in essence an extension of the CRS format with the difference that the elements in AN vector are not the non-zero elements of the original matrix, as they are in the CRS format, but pointers to non-zero blocks

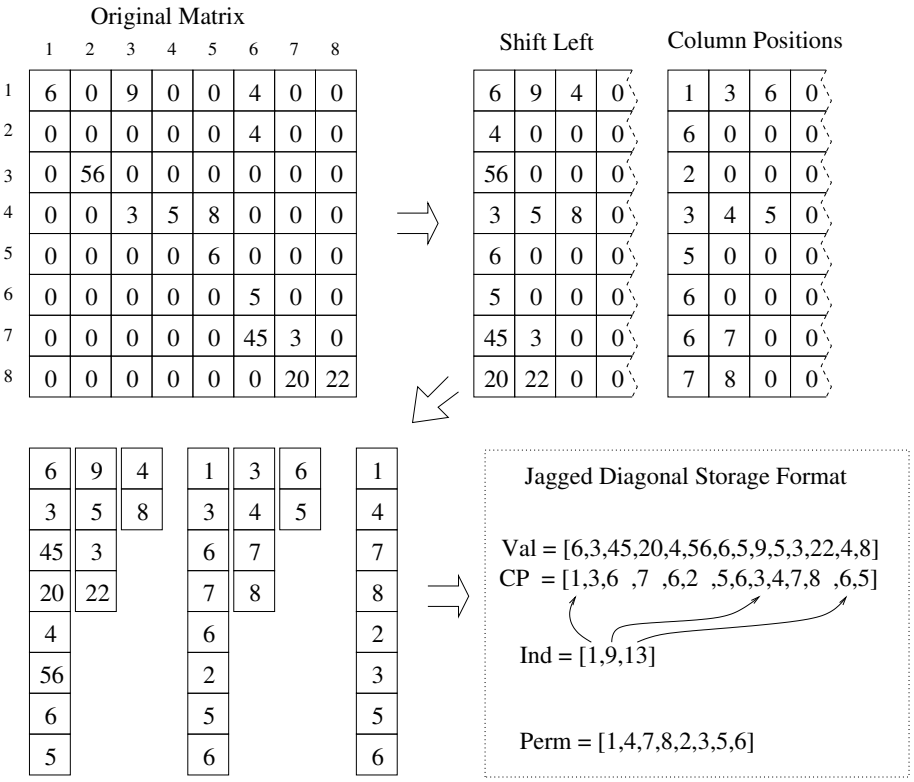


Figure 2.2: Jagged Diagonal Storage Format

which are stored row-wise in another array (Block array (BA)) that contains all the non-zero dense sub-blocks. This format can offer significant cost savings when the sub-blocks are large since we eliminate the need to explicitly store the positional information for each non-zero element. Furthermore, we can efficiently operate on the sub- blocks as we normally operate on dense matrices. However, apart from being non-general the BCRS has still a number of disadvantages: (a) the size of the dense sub-blocks often has to be precalculated in order to choose its optimal value and this induces an additional overhead for creating the matrix stored in this way, (b) the blocks tend to have low dimensions and this makes even the dense operations less efficient since the short rows of the dense sub-blocks result in short vector and thus high vector startup overhead.

In Figure 2.2 we depict the procedure of obtaining the JD storage by means of an example of an 8×8 sparse matrix. Figure 2.3 depicts the construction of the BCRS storage for an 8×8 sparse matrix where the block size is 2.

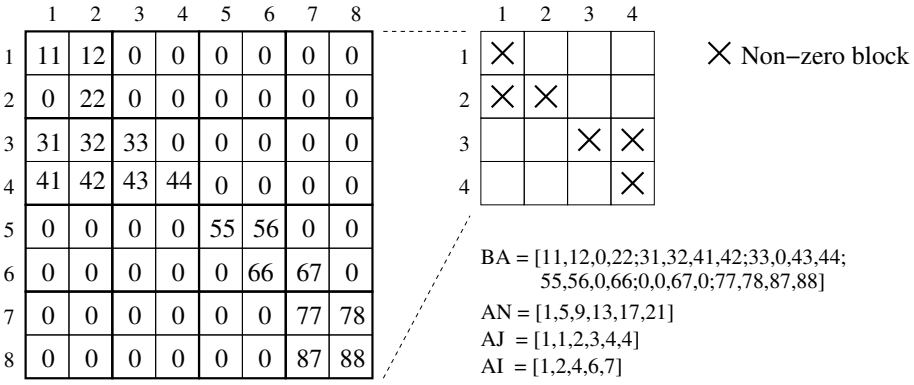


Figure 2.3: Block Compressed Row Storage Format

$$\begin{bmatrix} 11 & 12 & 0 & 0 & 0 & 0 \\ 21 & 22 & 23 & 0 & 0 & 0 \\ 0 & 32 & 33 & 34 & 0 & 0 \\ 0 & 0 & 0 & 44 & 0 & 0 \\ 0 & 0 & 0 & 54 & 55 & 56 \\ 0 & 0 & 0 & 0 & 65 & 66 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 12 & 23 & 34 & 0 & 56 \\ 11 & 22 & 33 & 44 & 55 & 66 \\ 21 & 32 & 0 & 54 & 65 & 0 \end{bmatrix}$$

Figure 2.4: Compressed Diagonal Format Example

The Compressed Diagonal Storage (CDS): Another non-general sparse matrix storage format that makes use of the particular sparsity pattern, in this case relating to finite element or finite difference discretization on a tensor product grid, is the CDS format. This format is used when the matrix is banded, that is, the non-zeros elements are within a diagonal band. In this case we are able to store only the diagonals in a regular form.

In Figure 2.4 we depict an example of how we can store only the three main diagonals of a banded matrix using the CDS method. Note that the expense of storing a few extra zeros is balanced by the regular, and thus more efficient, way that we can access and operate on the regularly stored diagonals. The CDS can give a considerable performance benefit if the matrix is banded. However, this format is highly unsuitable for general matrices since if we have a few rows that exceed the diagonal band, this will result in storing a large number of zero values.

2.2 Our Proposed Formats

In this section we describe a number of sparse matrix storage formats which emerged during our quest to find a format that alleviates the problems that

existing ones have. The current storage formats which were described in the previous section are either too specific for their application area and types of matrices (non-general) or suffer from several drawbacks that were mentioned earlier and are stated here for clarity:

- Short vectors: This problem is particularly evident in CRS but it is a more general phenomenon that is directly related to the fact that most sparse matrices only have a small number of non-zero elements in each row or column. By "small" we mean that when we construct a vector using the non-zero elements in a row and operate on it, the overhead cost is significant. The JD storage format successfully deals with this problem when the matrix has sufficiently large dimensions since we obtain vector of length on the order of the dimension of the matrix. However it does not deal with the other problems (see next bullets), moreover it is specially designed for sparse matrix vector multiplication and not efficient in other operations.

- Indexed Accesses: This problem is particular to JD and CRS, both general storage formats. The problem related to the fact that since no assumption is made of the sparsity pattern, the positional information of each non-zero element (e.g. the column position in CRS and JD) is stored explicitly. This results in indexed memory accesses whenever we need to access other elements using this positional information. This problem is mainly addressed by non-general storage formats such as BCRS and CDS that partly store data with implicit positional information.

- Positional Information Storage Overhead: This problem is related to the previously described problem. However, the focus here is on the overhead of storing the positional information. Formats like CRS and JD store the column position for each element (the row position is implied by the index we use to access the row) which requires a full word extra storage for each non-zero element. However, as we'll show later in Sections 2.2.1 and 2.3, it is possible to device a format in such a way that the positional information storage is decreased without losing the general applicability of the format.

In the following two sections we will describe two new formats and their variations that can alleviate the aforementioned problems that occur in the existing formats. We have to add here that the existing formats were developed having the existing architectures in mind. Our proposed formats are a result

of a hardware/software co-design, meaning that the formats were developed in parallel and work best if they are combined. However, this does not imply that the proposed formats require the associated architecture. The details of the architecture will be described in Chapter 3. Before proceeding to the actual description of the formats we will outline the main rationale that led us to those specific designs. As was mentioned earlier our goal was to alleviate the three main problems that were outlined above while trying to minimize the amount of fundamental changes in the vector architecture in order to support the format. We believe that the problems with operations on sparse matrices arise mainly due to two reasons:

- The irregularities of the sparsity patterns in combination with the current sparse matrix formats induce operations and memory accesses that are structured (no continuous or predictable access pattern) and do not have spacial locality. The result is that the working set of data cannot be kept in the processor, especially when the matrices have large dimensions.
- Even if the sparsity patterns cannot be parametrized easily[1], we have observed that the non-zeros elements are not evenly distributed over the matrix but are clustered.

We have based our proposed formats on these observations. The main idea behind our proposed formats is to use the clustering property to create smaller, manageable parts and operate on them separately rather than on the entire sparse matrix at once. By "manageable" we mean that the dimensions of the parts are such that the data can reside in processor memory (or registers). In our proposed formats we achieve this by partitioning the matrix in blocks whose dimensions are determined by the parameters of the processor. In the following Sections we describe in detail the process that led us to the specific sparse matrix storage formats and we provide a full description of the formats.

2.2.1 Blocked Based Compression Storage

The Blocked Based Compression Storage (BBCS) format is a result of studying the sparse matrix dense vector multiplication (SMVM) behavior on vector processors using existing formats. We have chosen this operation to assist us in the design of the format since it is the most ubiquitous and mostly used operation within the sparse matrix application field. Moreover it exhibits all the problems associated with sparse matrix operations that were mentioned earlier.

[1] except in the case of non-general matrices where this fact is taken in to consideration in constructing the storage format

Before proceeding to the description of the BBCS format we will briefly outline the problems that we encounter when we are performing the SMVM using the existing formats on vector processors.

Problem Analysis: The sparse matrix vector multiplication concerns the multiplication of a matrix by a dense vector. The definition of the multiplication of a matrix $\vec{A} = [a_{i,j}]_{i,j=0,1,\ldots,n-1}$ by a vector $\vec{b} = [b_i]_{i=0,1,\ldots,n-1}$ producing a vector $\vec{c} = [c_i]_{i=0,1,\ldots,n-1}$ is as follows:

$$\vec{c} = \vec{A}\vec{b}, \qquad c_i = \sum_{k=0}^{n-1} a_{i,k} b_k, \qquad i = 0, 1, \ldots, n-1 \tag{2.1}$$

Let now consider the execution of the multiplication in Equation (2.1) on a vector architecture. More in particular we assume a register type of organization, e.g., $IMB/370$ vector facility [12, 50], with the section size of s elements per register. When executed on such a VP the inner loop, i.e, the computation of $c_i = \sum_{k=0}^{n-1} a_{i,k} b_k$, can be vectorized. Ideally, if the section size s is large enough, i.e., $s \geq n$, one loop could be replaced with just one inner product instruction which multiplies the $\vec{A}_i$ vector, i.e., the i^{th} row of the matrix $\vec{A}$, with the $\vec{b}$ vector and produces as result the i^{th} element of the $\vec{c}$ vector. In practice the section size is usually smaller than n and the $\vec{A}_i$ and $\vec{b}$ vectors have to be split into segments of at most s element length to fit in vector registers. Consequently, the computation of c_i will be achieved with a number of vector instructions in the order of $\lceil \frac{n}{s} \rceil$. Although the speedup due to vectorization obviously depends on the section size value, due to issues like lower instruction fetch bandwidth, fast execution of loops, good use of the functional units, VPs perform quite well when $\vec{A}$ is dense. When operating on sparse matrices however VPs are not as effective because the lack of data regularity in sparse formats leads to performance degradation. Consider for instance that $\vec{A}$ is sparse and stored in the Compressed Row Storage (CRS) as described in Section 2.1.1 Under these assumptions the matrix-vector multiplication can be computed as follows:

```
for i = 0 to M   // M = Number of Rows
 c[i] = 0
 for k = AI[i] to AI[i+1]-1
  c[i] = c[i] + AN[k] * b[AJ[k]]
 end for
end for
```

In this case the vectorization of the inner loop is not straightforward any longer. One matrix row is still considered at a time but now the $Column\_Index$ set

should be used for selecting the appropriate values of the $\vec{b}$-vector. This means that when loading a section of the $\vec{b}$-vector first an index vector should be loaded from $Column\_Index$ and consequently a special Load Vector with Index has to be used to load the $\vec{b}$-vector section. To process this format each row of non-zero values forms a vector and it is accessed by using the index vector provided by the same format. As the probability that the number of non-zero elements in a matrix row is smaller than the VP's section size is rather high many of the vector instructions manipulate short vectors. Consequently, the processor resources are inefficiently used and the pipeline start-up times dominate the execution time. This makes the vectorization poor and constitutes a reason for VPs performance degradation. The other source for performance degradation is related to the use of the indexed load/store instructions which, in principle, can not be completed as efficient as a the standard load vector instruction.

The occurrence of short length vectors relates to the fact that vectorization is performed either on row or column direction but not on both. The number of $\vec{A}$ matrix elements within a vector register can be increased if more than one row (column) is loaded in a vector register at a time. Such vectorization schemes however are not possible assuming the VPs state of the art as such an approach introduces operation irregularity on the elements of the same vector register so that it is not possible to apply the "Single Operation Multiple Data" (SIMD) Principle.

When using the Jagged diagonal (JD) format for the SMVM we can eliminate the problem of short vectors. Consider a sparse matrix stored in the JD format as described in Section 2.1.2. Then the code for SMVM is as follows (for simplicity, we do not consider here the case where we need to permute the rows, e.g. when all the rows have an equal number of non-zero elements):

```
for k = 0 to M        // M = Number of Rows
  c[k] = 0            // initialize result
end for
for i = 0 to NC-1     // NC = Number of Columns in V
  for j = Ind[i] to Ind[i+1]
    c[j] = c[j] + Val[j] *b [CP[j]]
  end for
end for
```

Here we observe that if we vectorize the inner loop we will obtain larger vectors since the number of elements in the columns contained in CP are in the order of M (the number of rows in the original matrix), whereas the CRS provides vectors of size in the order of the average number of non-zero elements

per row. However, it is evident from the code that JD also suffers from indexed accesses in memory in order to access the elements of vector $\vec{b}$. Moreover, both JD and CRS suffer from the increased bandwidth requirements, imposed by the need to make use of the column position value for each non-zero element of the matrix.

With our proposed format, the Block Based Compression Format (BBCS), we try to tackle the problems associated with SMVM by providing a format that decreases the positional information overhead, alleviates the problem of short vectors, and reduces the amount of indexed accesses required when performing the SMVM. The storage formats can be utilized to its full potential when combined with an architectural extension for vector processors which will be briefly mentioned here and in full detail in Chapter 3

2.2.2 The Block Based Compression Storage (BBCS)

To obtain the BBCS format we proceed as follows: The $n \times n$ $\vec{A}$ matrix is partitioned in $\lceil \frac{n}{s} \rceil$ Vertical Blocks (VBs) $\vec{A^m}$, $m = 0, 1, \ldots, \lceil \frac{n}{s} \rceil - 1$, of at most s columns, where s is the VP section size. For each vertical block $\vec{A^m}$, all the $a_{i,j} \neq 0$, $sm \leq j < s(m+1)$, $i = 0, 1, \ldots, n-1$, are stored row-wise in increasing row number order. At most one block, the last block, can have less than s columns in the case that n is not a multiple of s. An example of such a partitioning is graphically depicted in Figure 2.5. In the discussion to follow we will assume, for the simplicity of notations, that n is divisible by s and all the vertical blocks span over s columns.

The rationale behind this partitioning is related to the fact that when computing the $\vec{A}\vec{b}$ product the matrix elements a_{ij} are used only once in the computation whereas the elements of $\vec{b}$ are used several times depending on the amount of non-zero entries in the $\vec{A}$ matrix in the corresponding column, as it can be observed in Equation (2.1). This implies that in order to increase performance it is advisable to maintain the $\vec{b}$ values within the execution unit which computes the sparse matrix-vector multiplication for reuse and only stream-in the a_{ij} values. As to each vertical block $\vec{A^m}$ corresponds an s element section of the $\vec{b}$-vector, $\vec{b^m} = [b_{ms}, b_{ms+1}, \ldots, b_{ms+s-1}]$ we can multiply each $\vec{A^m}$ block with its corresponding $\vec{b^m}$ section of the $\vec{b}$-vector without needing to reload any $\vec{b}$-vector element.

Each $\vec{A^m}$, $m = 0, 1, \ldots, \frac{n}{s} - 1$, is stored in the main memory as a sequence of 6-tuple entries. The fields of such a data entry are as follows:

1. **Value**: specifies the value of a non-zero $a_{i,j}$ matrix element if $ZR = 0$.

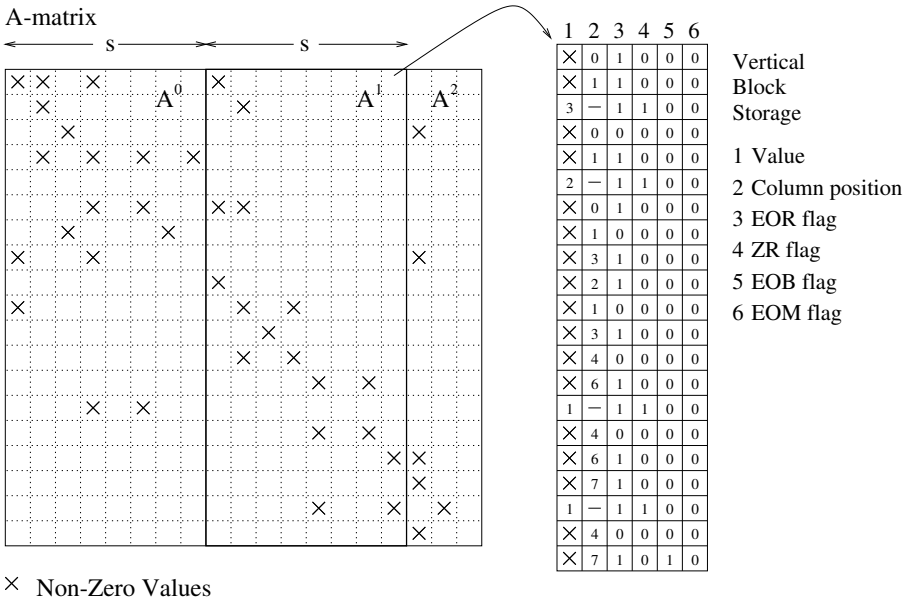


Figure 2.5: Block Based Compressed Storage Format

Otherwise it denotes the number of subsequent block rows[2] with no non-zero matrix elements.

2. **Column-Position (CP)**: specifies the matrix element column number within the block. Thus for a matrix element $a_{i,j}$ within the vertical block $\vec{A^m}$ it is computed as $j \bmod m$.

3. **End-of-Row Flag (EOR)**: is 1 when the current data entry describes the last non-zero element of the current block row and 0 otherwise.

4. **Zero-Row Flag (ZR)**: is 1 when the current block row contains no non-zero value and 0 otherwise. When this flag is set the $Value$ field denotes the number of subsequent block rows that have no non-zero values.

5. **End-of-Block flag (EOB)**: when 1 it indicates that the current matrix element is the last non-zero one within the VB.

6. **End-of-Matrix flag (EOM)**: is 1 only at the last entry of the last VB of the matrix.

[2]By block row we mean all the elements of a matrix row that fall within the boundaries of the current VB.

The entire $\vec{A}$ matrix is stored as a sequence of VBs and there is no need for an explicit numbering of the VBs.

When compared with other sparse matrix representation formats, our proposal requires a lower memory overhead and bandwidth since the index values associated with each $a_{i,j} \neq 0$ are restricted within the VB boundaries and can be represented only with $\log s$ bits instead of a word which can address the full address range. The flags can be explicitly 4-bit represented or 3-bit encoded and, depending on the value of s, they may be packed with the CP field on the same byte/word.

2.2.3 Block-wise Sparse Matrix Vector Multiplication Using the Block Based Compression Storage Format

Having described the BBCS storage format we proceed to describe how the SMVM is performed using the BBCS scheme. We assume a register vector architecture with section size s and the data organization described in Chapter 3. The second multiplication operand, the $\vec{b}$-vector, is assumed to be dense than no special data types or flags are required. The b_k, $k = 0, 1, \ldots, n-1$, values are sequentially stored and their position is implicit. The same applies for the result, i.e., the $\vec{c}$-vector. The $\vec{Ab}$ product can be computed as $\vec{c} = \sum_{m=0}^{\frac{n}{s}-1} \vec{A^m} \times \vec{b^m}$.

To vectorize each loop computing the $\vec{A^m} \times \vec{b^m}$ product and because generally speaking $\vec{A^m}$ can not fit in one vector register, we have to split each $\vec{A^m}$ into a number of subsequent VB-sections $\vec{A_i^m}$ each of them containing at most s elements. Under the assumption that each vertical block $\vec{A^m}$ is split into $\#s_m$ VB-sections $\vec{A_i^m}$ the $\vec{c}$-vector can be expressed as $\vec{c} = \sum_{m=0}^{\frac{n}{s}-1} \sum_{i=0}^{\#s_m} \vec{A_i^m} \times \vec{b^m}$. Consequently, $\vec{c}$ can be iteratively computed within $\frac{n}{s}$ loops as $\vec{c_m} = \vec{c_{m-1}} + \sum_{i=0}^{\#s_m} \vec{A_i^m} \times \vec{b^m}$, $m = 0, 1, \ldots, \frac{n}{s} - 1$, where $\vec{c_m}$ specifies the intermediate value of the result vector $\vec{c}$ after iteration m is completed and $\vec{c_{-1}} = \vec{0}$.

Assuming that $\vec{A_i^m} = [A_{i,0}^m, A_{i,1}^m, \ldots, A_{i,s-1}^m]$ and $\vec{b^m} = [b_0^m, b_1^m, \ldots, b_{s-1}^m]$ a standard vector multiplication computes the $\vec{A_i^m} \times \vec{b^m}$ inner product as being $c_i^m = \sum_{j=0}^{s-1} A_{i,j}^m b_j^m$ which is the correct result only if $\vec{A_i^m}$ contains just one row. As one VB-section $\vec{A_i^m}$ may span over $r_i \geq 1$ rows of $\vec{A^m}$ the $\vec{A_i^m} \times \vec{b^m}$ product should be an r_i element vector. In particular, if $\vec{A_i^m} = [\vec{A_i^{m,0}}, \vec{A_i^{m,1}}, \ldots, \vec{A_i^{m,r_i-1}}]$, with $\vec{A_i^{m,j}} = [A_{i,0}^{m,j}, A_{i,1}^{m,j}, \ldots, A_{i,\#r_j-1}^{m,j}]$, $j = 0, 1, \ldots, r_i - 1$, and $\#r_j$ being the number of elements within the row j,

$c_i^{\vec{m}} = A_i^{\vec{m}} \times b^{\vec{m}}$ has to be computed as follows:

$$c_i^{\vec{m}} = [c_{i,0}^m, c_{i,1}^m, \ldots, c_{i,r_i-1}^m], \qquad c_{i,j}^m = A_i^{\vec{m},j} \times b^{\vec{m}}, \quad j = 0, 1, \ldots, r_i - 1 \tag{2.2}$$

Consequently, to compute $c_i^{\vec{m}}$, r_i inner products, each of them involving $\#r_j$ elements, have to be evaluated. Moreover when executing the evaluation of $A_i^{\vec{m},j} \times b^{\vec{m}}$ inner product the "right" elements of $b^{\vec{m}}$ have to be selected. Therefore, $c_{i,j}^m$ is evaluated as follows:

$$\begin{aligned} c_{i,j}^m = & A_{i,0}^{m,j} \cdot b^m_{CP(A_{i,0}^{m,j})} + A_{i,1}^{m,j} \cdot b^m_{CP(A_{i,1}^{m,j})} + \ldots \\ & + A_{i,\#r_j-1}^{m,j} \cdot b^m_{CP(A_{i,\#r_j-1}^{m,j})}, \quad j = 0, 1, \ldots, r_i - 1 \end{aligned} \tag{2.3}$$

As each $c_{i,j}^m$ contributes to the $\vec{c}$ vector element in the same row position as $A_i^{\vec{m},j}$ and row position information is not explicitly memorized in the BBCS format bookkeeping related to index computation has to be performed. Moreover as $A_i^{\vec{m}}$ does not contain information about the row position of its first entry, hence not enough information for the calculation of the correct inner product positions is available, a Row Pointer Register (RPR) to memorize the starting row position for $A_i^{\vec{m}}$ is needed. The RPR register will be introduced and described in more detail in Chapter 3. The RPR is reset every time the processing of new $A^{\vec{m}}$ is initiated and updated by the index computation process.

To clarify the mechanism we present in Figure 2.6 an example. We assume that the section size is 8, the VB-section contains the last 8 entries of $\vec{A^1}$, $\vec{b^1}$ contains the second 8-element section of $\vec{b}$, and that an intermediate $\vec{c}$ has been already computed though we depict only the values of $\vec{c}$ that will be affected by the current step. First the inner product calculation and the index calculation are performed. For the inner product calculation only the a(i, j) elements with $ZR = 0$ are considered. Initially they are multiplied with the corresponding $\vec{b^1}$ elements, the CP field is used to select them, and some partial products are obtained. After that, all the partial products within the same row have to be accumulated to form the inner products. As the EOR flags delimit the accumulation boundaries they are used to configure the adders interconnection. The index calculation proceeds with the RPR value and whenever a entry with $ZR = 1$ is encountered RPR is increased with the number in the $Value$ field. When an EOR flag is encountered the RPR is assigned as index to the current inner product and then increased by one. Second the computed indexes are used to select the $\vec{c}$ elements to whom the computed inner products should be accumulated and the accumulation is performed.

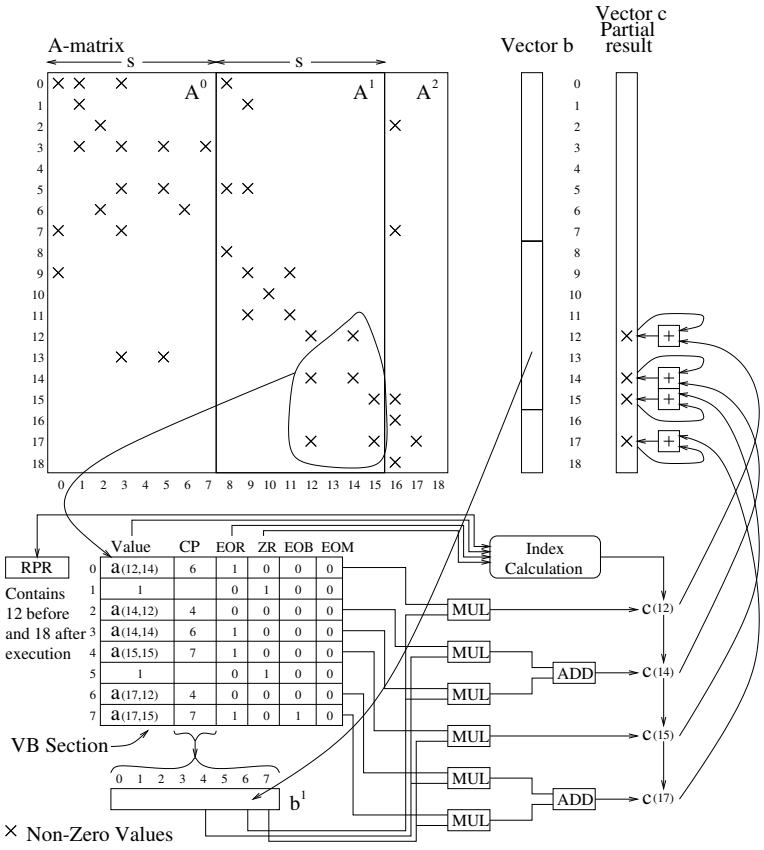


Figure 2.6: Sparse Matrix-Vector Multiplication Mechanism

To be able to execute the previously discussed block-wise SMVM we propose the extension of the VP instruction set with 2 new instructions: *Multiple Inner Product and Accumulate* ($MIPA$) and *LoaD Section* (LDS). We will give here a brief outline of the workings of the new instructions that are introduced. However, for a more detailed description please refer to Chapter 3.

$MIPA$ is meant to calculate the inner product $A_i^{\vec{m}} \times b^{\vec{m}}$. Furthermore, it also performs the accumulation of the $c_i^{\vec{m}}$ elements to the $\vec{c}$-vector values in the corresponding locations. The instruction format is $MIPA\ VR1$,$VR2$,$VR3$. The vector register $VR1$ contains $A_i^{\vec{m}}$, $VR2$ contains $b^{\vec{m}}$, and $VR3$ contains initially those elements of the $\vec{c}$-vector that correspond to the non-empty rows of $A_i^{\vec{m}}$ and after the instruction execution is completed the updated values of them.

LDS is meant to load an $A_i^{\vec{m}}$ VB-section from the main memory. The in-

struction format is LDS @A,$VR1$,$VR2$. @A is the address of the first element of the VB-section to be loaded. After an LDS instruction is completed $VR1$ contains all the non-zero elements of the VB-section starting at @A, $VR2$ contains the indexes of the VB-section rows with non-zero elements (to be used later on as an index vector to load and store a $\vec{c}$ section), and the Column Position Register (CPR), a special vector register, is updated with the CP and flag fields of the corresponding elements in $VR1$. To execute the LDS instruction the VP Load Unit has to include a mechanism to analyze the BBCS flags in order to compute the indexes and filter the entries with $ZR = 1$. In this way $VR1$ always contains only nonzero matrix elements and no trivial calculations will be performed.

Even though the previously discussed approach guarantees an efficient filling of the vector registers, it may suffer a performance degradation due to the use of indexed vector load/store instructions as such operations, depending on the implementation of the main memory and/or load unit, may create extra overhead. The total amount of indexed accesses is reduced since where the JD and CRS need to make as many indexed memory as the number of non-zeros in the matrix whereas using the BBCS scheme we only need to access the intermediate result elements. This number is equal to the number of non-zeros divided by the average number of non-zeros per VB-row which can vary from 1 to s, where s is the section size used to partition the matrix. However, the use of indexed load/stores can be avoided altogether if instead of partitioning the VBs in s-element VB-sections, they are divided in $\lceil \frac{n}{ks} \rceil$ ks-element hight *segments*[3] where $k \geq 1$. To support this new division the BBCS has to include an extra flag, the End of Segment (EOS) flag. This flag is 1 for the entries which describe the last element of a segment. Under this new assumption when a VB-section is loaded with the LDS instruction the loading stops after s nonzero elements or when encountering the EOS, EOB, or EOM flag. By restricting the VB-section's row span within a VB segment we guarantee that all the $A_i^m \vec{\times} b^m$s will contribute only to elements within a specific ks-element wide section of the vector $\vec{c}$. Consequently, the $\vec{c}$-vector can be manipulated with standard load/store instructions. This idea will be further developed in the Following Section where we describe the Hierarchical Sparse Matrix (HiSM) storage format.

Concluding, we proposed sparse matrix organization to alleviate some of the problems related to sparse matrix computation on vector processors, e.g., inefficient functional unit utilization due to short vector occurrence and in-

[3] All of them are $s \times ks$ segments except the ones on the right and/or bottom edge of the matrix which might be truncated if the dimension of the matrix n is not divisible by s.

creased memory overhead and bandwidth when performing sparse matrix vector multiplication on vector processors. First we introduced a Block Based Compressed Storage (BBCS) sparse matrix representation format which requires a memory overhead in the order of $\log s$ bits per nonzero matrix element, where s is the vector processor section size. Additionally, we described a Block-wise SMVM scheme to compute the product of an $n \times n$ sparse matrix with a dense vector in $\frac{n}{s}$ vectorizable loops. To support the vectorization of the Block-wise SMVM scheme two new vector instructions, *Multiple Inner Product and Accumulate* ($MIPA$) and *LoaD Section* (LDS) were proposed which will be fully described in Chapter 3

2.3 The Hierarchical Sparse Matrix Format

In this section we describe 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 an associated vector architecture extension is proposed in Chapter 3.

The HiSM format is designed taking into consideration the advantages and disadvantages that the BBCS format offers. The HiSM format has a more intuitive organization, retains the advantages of BBCS and alleviates its disadvantages. The BBCS has two main disadvantages:

- Indexed loads/stores. Performing the SMVM using the BBCS format reduces the amount of indexed memory accesses that need to be performed. However, since indexed accesses can highly influence the performance of sparse matrix operation we would prefer to fully alleviate the need for indexed accesses. We have already indicated in the previous Section how we can achieve this. This Section will describe in detail how this goal is reached.

- Flexibility. A problem related to sparse matrix formats is often the inflexibility of altering the contents of the matrix. This can occur in some operations (i.e. Gaussian elimination) where new non-zero elements are added to the matrix. With CRS, JD and BBCS, in order to maintain the correct organization of the storage format when a new element is added, a large part of the entire structure needs to be altered. This is usually dealt with by allowing some elbow room when adding new elements in order to avoid updating the structure for every new element. However,

itcan be a costly and complicated operation and therefore we aim to design a format that can offer higher flexibility than existing formats.

With the HiSM format we aim to alleviate the aforementioned disadvantages. This is mainly achieved in two ways: (a) we partition the matrix in such a way that each partition block has a limited amount of rows and columns in order to alleviate the indexed memory accesses and (b) we introduce a layered description of positional information in order to make the structure more flexible to changes. The details will be described in the following Section.

2.3.1 Hierarchical Sparse Matrix Format Description

With the introduction 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 [68], and a hierarchical way of storing a sparse matrix, discussed in [25].

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[4] 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 2.7 (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 2.7, the non-empty s^2-blocks form a similar sparsity pattern as the non-zero values

[4]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 [50].

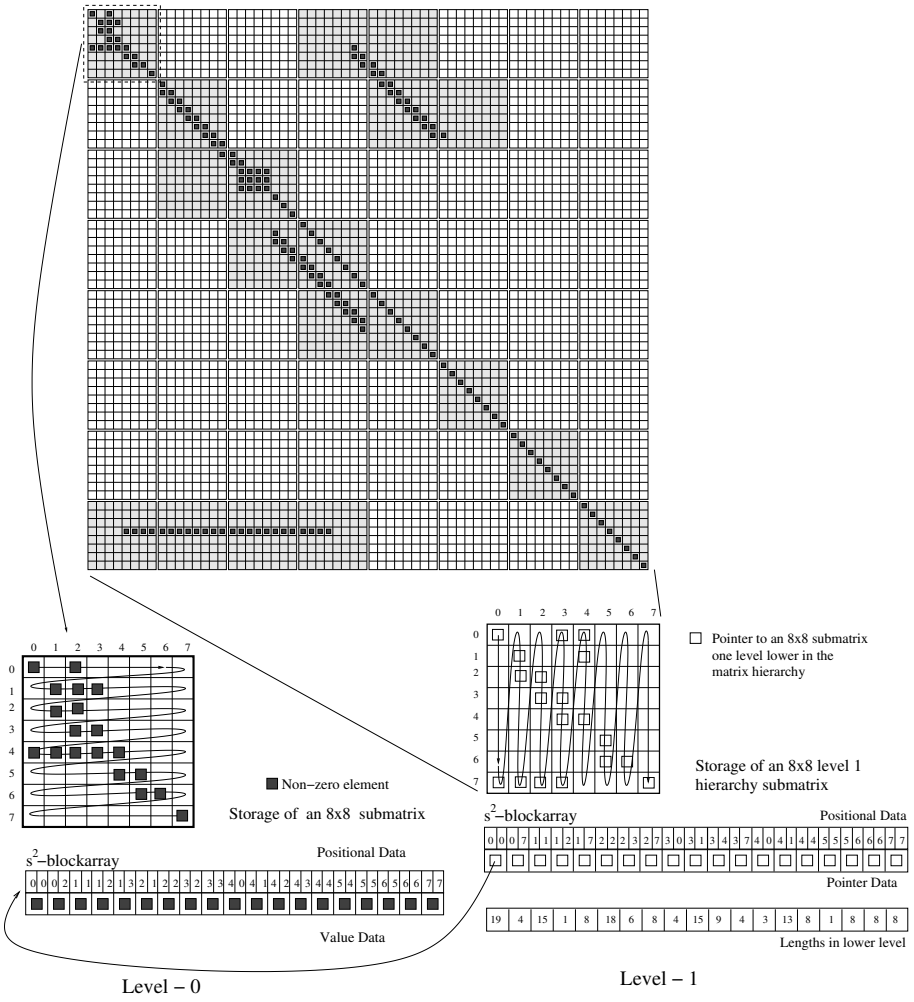


Figure 2.7: Example of the Hierarchical Sparse Matrix Storage Format

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 2.7 (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 2.7 we see that for level one an extra vector is stored in memory 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 2.7 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 we will discuss in Section 2.4 the HiSM format offers a storage reduction of about 27% 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 Chapter 4 we have measured that each s^2-blockarray contains on average $2.18 \times s$ non-zero elements 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 a 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 concern 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 func-

tionality 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 [67] how such a functional unit can be pipelined. Specifically, the MIPA functional unit described in Chapter 3 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.

2.4 A Comparison of Sparse Matrix Storage Formats

In this section we will provide a preliminary comparison of the presented formats. We will focus on storage efficiency of the formats in Section 2.4.1 and vector register filling in Section 2.4.2. The results presented here are indicative of the various aspects that contribute to the overall efficiency of the sparse matrix operations. However, for a performance comparison the reader can refer to Chapter 5.

2.4.1 Format Storage Efficiency

In Section 2.2 we identified the issue of storing the additional positional information to be one of the causes of inefficient operation on sparse matrices. We mentioned that we believe that this problem, mainly prevalent in general sparse matrix storage formats such as CRS and JD, can be alleviated using our proposed formats and described how the formats achieve this goal in Sections 2.2.1 and 2.3. The present section will provide quantitative analysis of the storage amount needed for each of the described sparse matrix formats.

Minimizing the amount of storage needed for the positional information is important since it simply reduces the amount of storage required, which can be substantial for large matrices. Even more important though is the reduction on the memory bandwidth requirements when we are processing a sparse matrix. Most operations, with the exception of some unary operations such as the multiplication of a matrix with a scalar[5], require the loading of the positional

[5]When multiplying a matrix with a scalar it suffices to multiply each nonzero element with the scalar value and storing it to its original position without requiring any knowledge about its position in the matrix

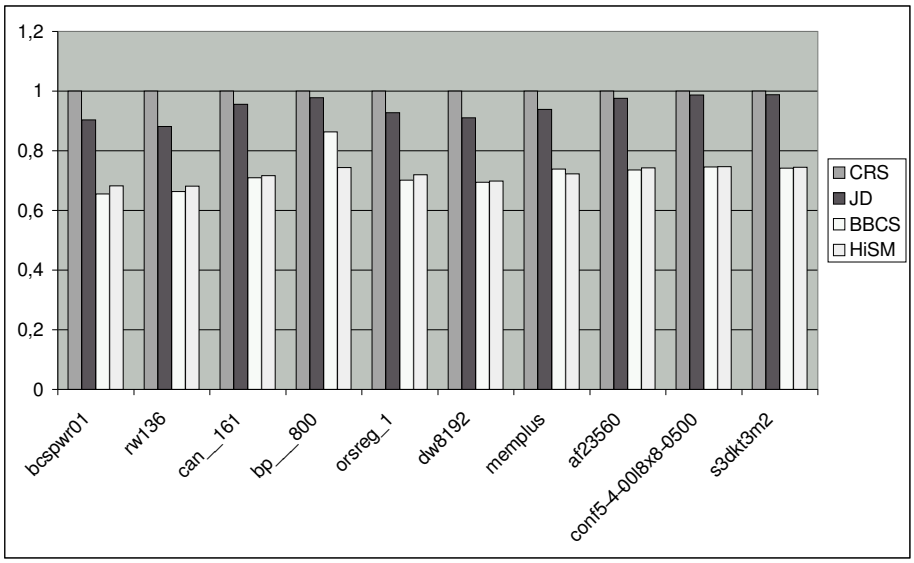


Figure 2.8: Storage Space for Increasing Number of Non-zeros in Matrix

information in order to carry out the operation. The positional information is usually used as indexing for operations on the non-zero elements and are not operated on themselves. Consider, for instance, the example of the multiplication of a sparse row from a sparse matrix with a dense vector using the CRS as described in Section 2.2.1. The non-zeros are loaded into the processor in one vector register. Subsequently the column positions of those elements are loaded in a second vector register. Those are then used as an index vector in order to load the corresponding values from the dense vector. The multiplication that will take place, and thus consuming the resources of the processor, involves only the non-zeros and the values from the dense vector. Consequently, we can conclude that reducing the amount of positional information reduces the memory bandwidth allowing better resource utilization especially in memory bound operations.

In Figures 2.8 to 2.10 we depict the storage space needed for each of the CRS, JD, BBCS and HiSM formats for various matrices. Depicted is the storage space needed per matrix versus the matrix name. The numbers have been normalized to the storage space needed for CRS due to the large scale differences in the sizes of the matrices which were used. The matrix suite used in this comparison is described in more detail in Chapter 4. Overall we can observe that the BBCS format needs 74% and 78% of CRS and JD formats respectively. The HiSM format achieves 72% and 76% when compared to CRS

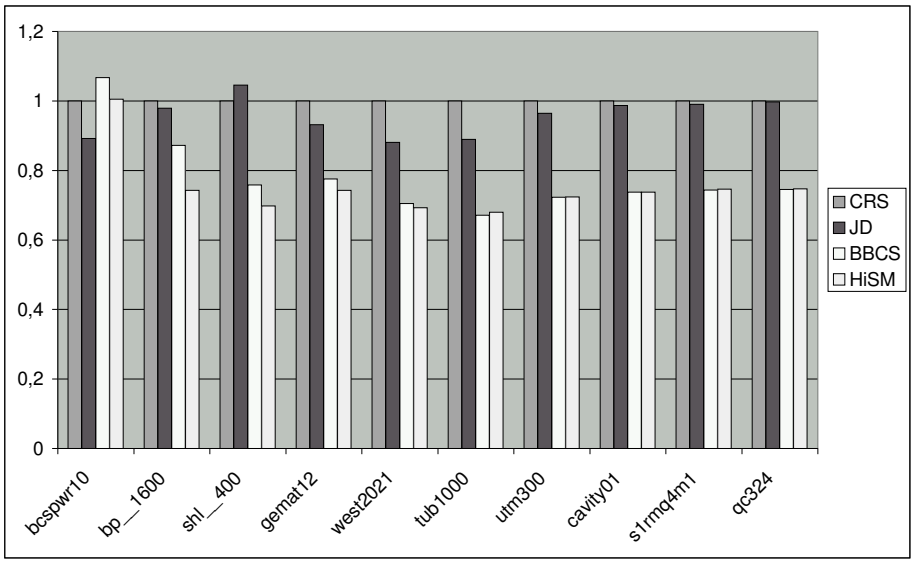


Figure 2.9: Storage Space for Increasing Matrix Locality

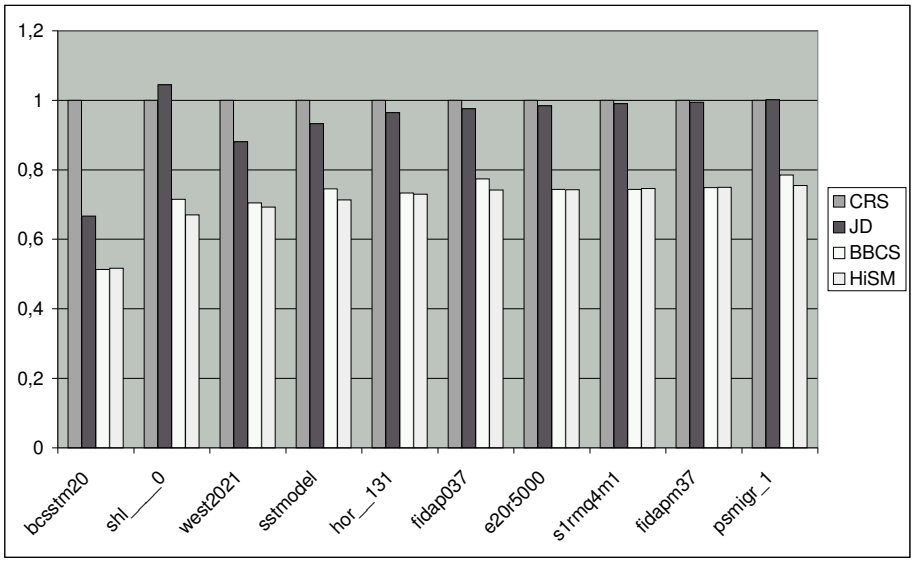


Figure 2.10: Storage Space for Increasing Average Number of Non-zeros per Row

and JD. The Matrix suit is divided in three sets according to important matrix characteristics: (a) The total number of non-zeros (Figure 2.8, (b) The local-

ity of non-zero distribution (Figure 2.9 and (c) the average non-zeros per row in the matrix (Figure 2.10. Looking at Figure 2.9 we can observe that very low locality, i.e. the non-zero elements are scattered almost randomly over the matrix, is the only case where CRS and JD marginally outperform BBCS and HiSM in storage space. The reason for this is as follows: In the case of the BBCS format the non-clustered distribution of the non-zero elements results in the use of relatively many entries with the NZ flag set. In the case of HiSM a random-like distribution of non-zeros means that the $s \times s$-blocks in level-1 will contain a relatively high number of non-zero $s \times s$-blocks. However, we have observed that matrices of very low spacial locality in the distribution of non-zeros are very rare. Moreover, as we shall demonstrate in Chapter 5 the other benefits of using the BBCS and HiSM still outweigh this drawback in overall performance.

2.4.2 Vector Register Filling Efficiency

In this section we provide a comparison of the vector register filling that is achieved using the various formats that were discussed previously. By Vector Register Filling (VRF) we mean the average number of elements that a vector register will contain while operating on a sparse matrix. A vector architecture that makes use of vector registers will typically load the elements to be operated upon in a vector register before execution. Given an overhead cost of starting a new vector instruction we can conclude that the vector register filling is an important aspect that can influence execution performance and therefore we strive to maximize the vector register filling. Generally, this mainly depends on the ability to load long arrays of data from memory. If the array which resides in memory is larger than the vector register size (also called the section size s) then the elements load in sections of s, in a process called strip mining. The larger the array the higher the VRF will be. In the case of sparse matrices, the organization of the format which is used to store the matrix influences the filling of the vector registers, depending on the availability of long chucks of data that can be accessed and operated on in a uniform manner. Consider for instance the case of CRS. Here the largest continuous data array that can be loaded into a vector register each time is equal to the number of elements in a row. the result is that a low number of non-zero elements per row will result in low vector register filling. In Figure 2.11 we depict how the partitioning of the sparse matrix and the organization of the format influence the formation of vectors of different lengths. The v value denotes the number of vectors which are needed to access the entire matrix. Clearly, the BBCS exhibits the longest vectors and CRS the lowest. However, Figure 2.11 is for

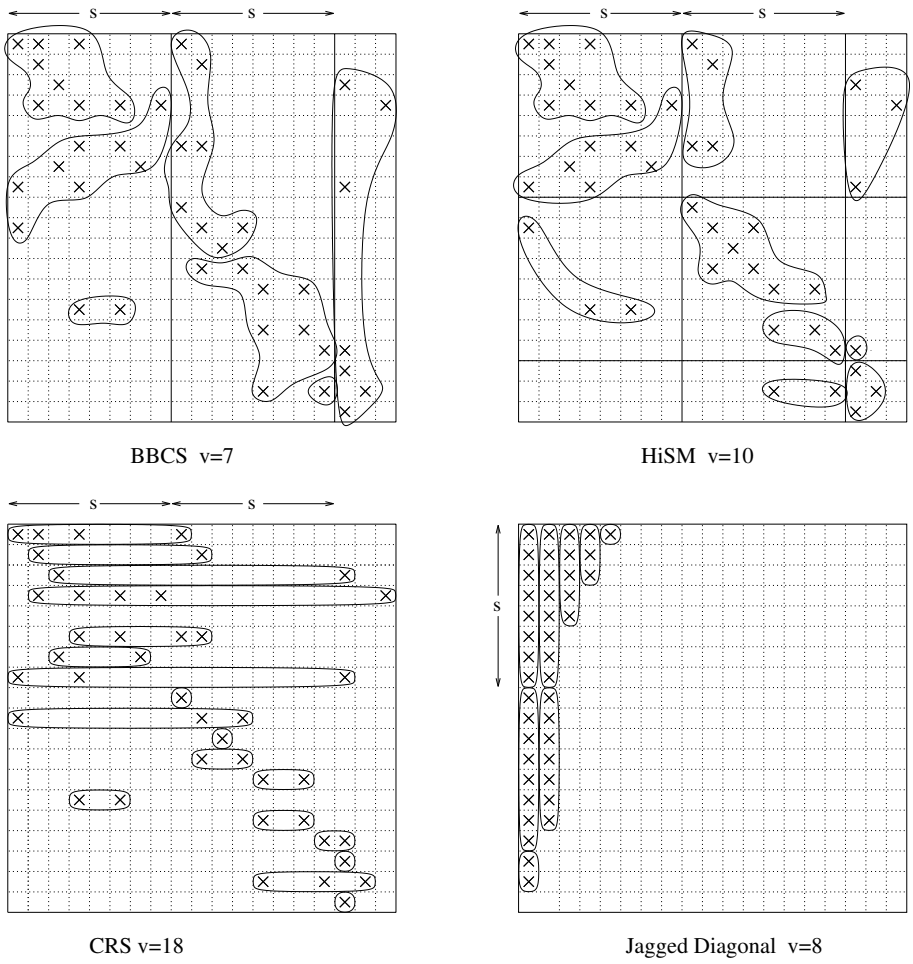


Figure 2.11: Vector Register Filling

illustration purposes and therefore we have conducted a quantification of the VRF achieved for CRS, JD, BBCS and HiSM based on real matrices (for a detailed description of the matrix suite please refer to Chapter 4. The results are depicted in Figures 2.12 to 2.15. First, in Figure 2.12 we depict the average VRF over all the matrices for varying section size of the vector processor. The vertical dimension denotes the vector filling achieved by each of the respective sparse matrix formats. For reference purposes we also depict the ideal (MAX) which is equal to the section size s. We can observe that unlike CRS, JD, BBCS and HiSM are achieving a high VRF and, more importantly, the VRF is scalable for increasing section sizes. JD achieves a high VRF as ex-

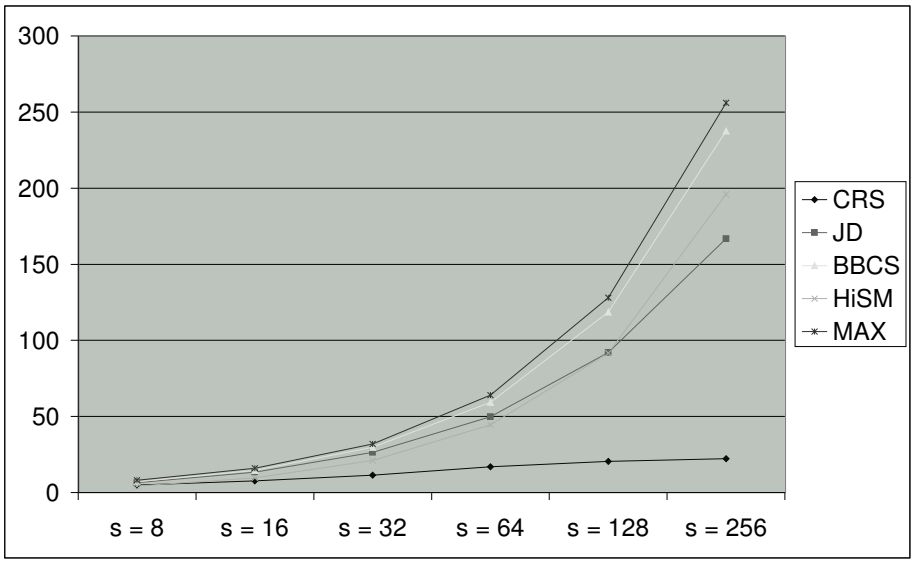


Figure 2.12: Vector Register Filling for Increasing Section Size

pected. As we described earlier, JD is specially designed to increase the VRF for sparse matrix vector multiplication by providing vectors as long as the vertical dimension of the matrix. The HiSM format achieves a VRF similar to JD and this is due to the fact that the matrix elements are mostly clustered, which results in non-zero $s \times s$-blocks that are sufficiently filled to provide enough non-zero elements to fill the vector registers. The BBCS achieves an even higher VRF simply because the blocks are larger and therefore contain a larger number of non-zero elements. In this graph we can also demonstrate the claim that CRS suffers from low VRF. For CRS, the achieved VRF is tightly related to the average number or Non-Zero Elements Per Row (NZPR) and this is clearly depicted in Figure 2.15 where we can observe the VRF rise for increasing NZPR. Furthermore, in Figure 2.14 we can observe the dependence of the HiSM VRF on the matrix locality. Low locality (higher "random" scattering of the non-zero elements) causes the $s \times s$-blocks to be generally poorly filled and therefore decreasing the possibility to form long vectors. However, as we mentioned earlier, these cases of very low spacial locality of non-zero distribution are very rare and generally the non-zeros are mostly clustered in such a degree that the $s \times s$-blocks are adequately filled to achieve a high VRF.

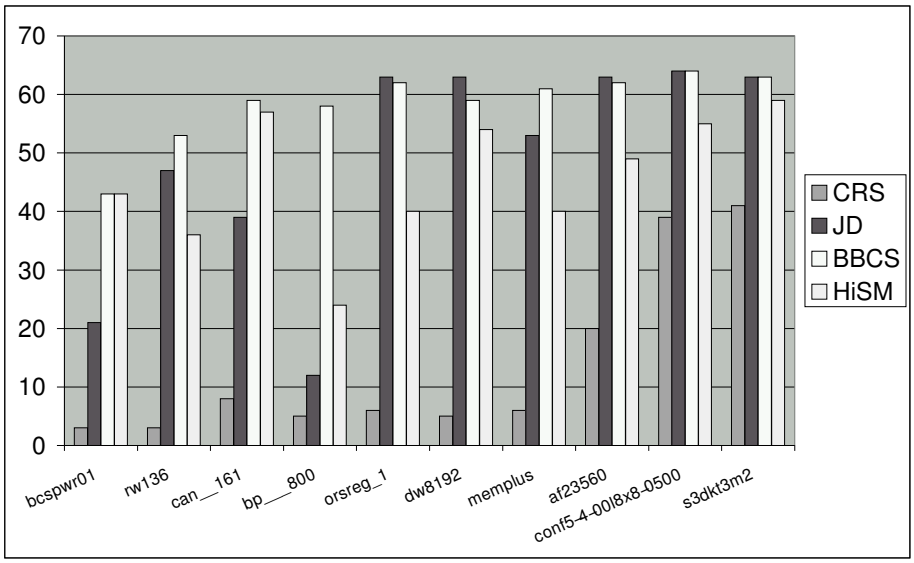


Figure 2.13: Vector Register Filling for Increasing Number of Non-zeros in Matrix (Section size = 64)

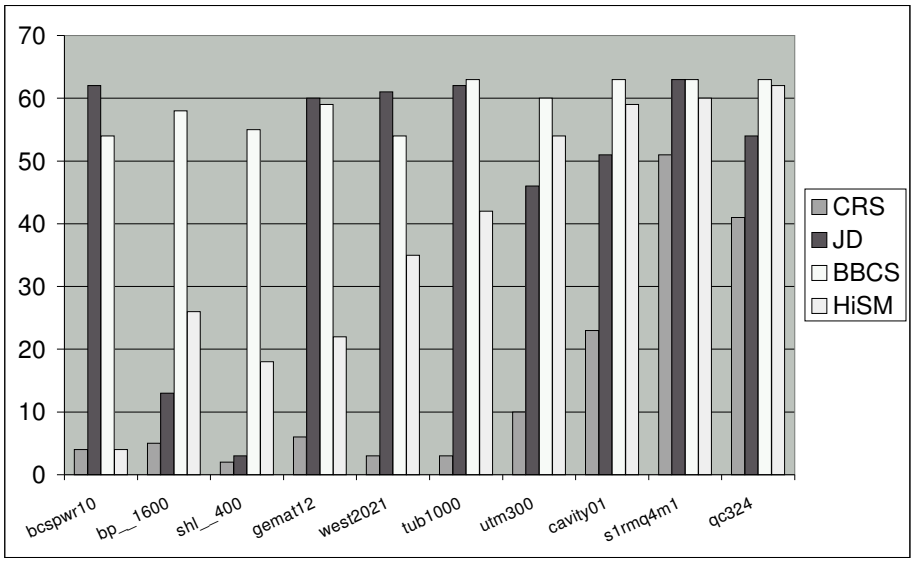


Figure 2.14: Vector Register Filling for Increasing Matrix Locality (Section size = 64)

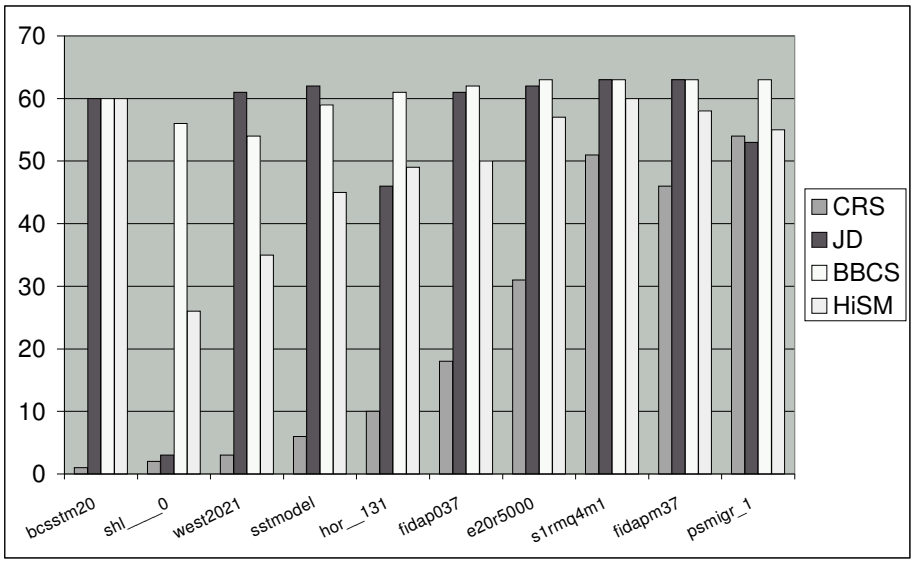


Figure 2.15: Vector Register Filling for Increasing Average Number of Nonzeros per Row (Section size = 64)

2.5 Conclusions

In this chapter we have introduced two new sparse matrix storage formats which constitute part of our proposed mechanism for increasing sparse operations performance on vector processors. First we described the existing sparse matrix storage formats with emphasis on Compressed Row Storage (CRS) and Jagged Diagonal (JD), both being general type and widely used formats. We have outlined their advantages and disadvantages. Subsequently, we have described our first proposed format, the Block Based Compression Storage (BBCS) that arose from our study of the sparse matrix vector multiplication and discussed the expected advantages over existing schemes. Following that, we presented our second proposed format, the Hierarchical Sparse Matrix (HiSM) compression format, which offers a number of improvements over BBCS in that it alleviates the use of Index loads altogether and has a more flexible structure. We completed the storage format discussion by providing a quantitative analysis of the existing and proposed formats. More in particular, we have compared the CRS, JD, BBCS and HiSM formats for required storage space and Vector Register Filling (VRF). We have shown that BBCS requires on average only 74% and 78% of the storage space needed for CRS and JD respectively. The corresponding numbers for HiSM are 72% and 76%

of the CRS and JD storage space respectively. Furthermore, we have shown that the BBCS format achieves the highest VRF and that the HiSM exhibits similar VRF as the JD format (which was designed for this purpose) despite the partitioning of the matrix in small parts.

The proposed formats however, were co-designed having a hardware support in mind and therefore, in order to utilize the full potential of BBCS and HiSM we need to use the vector architectural support. In the chapter to follow we present and describe the architectural support needed to accommodate our proposal.

Chapter 3

Vector Processor Extensions

In the previous chapter we described existing storage formats for sparse matrices and introduced our proposed sparse matrix storage formats, Block Based Compression Storage and Hierarchical Sparse Matrix Format, designed to alleviate a number of shortcomings of current solutions for efficiently performing sparse matrix operations on vector processor. We have provided evidence suggesting that our proposal has advantages in terms of storage space requirements when compared to commonly used sparse vector formats. We mentioned that in order to obtain the desired result we need to use these formats in conjunction with a vector processing unit that offers architectural and hardware support for the operation on these formats.

In this chapter we describe our approach for such a solution which consists of an architectural and organizational extension to a *Generic* vector processor. The remainder of this chapter is organized as follows: In the following section we describe what we mean by *generic* vector architecture, the framework on which we base our extension. Following, in Section 3.2.1 we describe our proposed architectural extension for supporting the BBCS format consisting of a set of new instructions, special registers and functional units. Similarly, in Section 3.3 we discuss architectural extensions associated with the HiSM format. Finally in Section 3.4 we draw a number of conclusions.

3.1 Basic Vector Architecture

In this section we describe the architecture and organization of a *Generic* Vector Processor (GVP). By describing the GVP we try to outline the *basic* functionalities offered by most available vector processors today and in the past. Designs that we have considered for the aforementioned *basic* functionalities

are processors such as the IBM system/370 Vector architecture [12, 50], the CRAY series [54], the VLX [33] etc.

The main distinguishing property of vector processors (VPs) when compared to scalar processors is the ability to operate on arrays (vectors) of data by using a single instruction rather than only one element at a time.[1]. Depending on the length of the vector that can be operated on by a single instruction, vector processors cab be divided in two main categories:

- *Memory-to-Memory* vector architectures. The very first vector architectures were of this type [20]. The vector instructions act on vectors which can have an unlimited size and reside in memory. These types of vector processors can have a very high theoretical peak performance. However, fetching a vector from memory induces high startup times which can be detrimental when the vectors are short or intermediate results need to be stored back in memory. For this reason this approach has been abandoned in favor of register-to-register vector architectures. Only recently we can observe a returned interest in memory-to-memory architectures [39] where long vectors are processed using complex instructions eliminating the need to store intermediate results to memory.

- *Register-to-Register* vector architectures were introduced with the Cray-1 [54] vector processor on which most subsequent vector processors were based. In a register-to-register architecture the vector instructions can only operate on limited sized vectors that reside in *Vector Registers*. Typically, a vector register will contain between 32 - 128 elements. By using vector registers the startup times for a vector instruction are reduced. Additionally, more flexibility is offered since intermediate results can be stored in vector registers and accessed fast. Furthermore, the execution can be further increased by the use of chaining. For the remainder of this thesis we will focus only on register-to-register vector architectures.

In the next few sections we will discuss in more detail the architecture and the internals of register-to-register vector processors.

3.1.1 Organization

In this section we describe the overall vector processor organization and some design techniques to some detail.

[1] Note that many modern microprocessors offer a vector processing functionality in the form of multimedia processing [31, 48, 51, 64]. However, the functionality is limited as it assumes only dense matrix processing and will not be considered further in this thesis.

Vector Registers: As we mentioned before, in a register-to-register vector architecture all the vector operations have as operands the architecturally visible Vector Registers (VR). Typically a vector processor will have a vector register file consisting of 8 to 32 vector registers, each register containing 32-256 elements of 32 or 64 bits. The size of the VRs is usually called maximum vector size or *section size*, which we will denote from now on as s. Since the size of an array of data which we want to operate upon can be less than s, vector processors usually employ a special register, the *Vector Length Register* (VLR), that is used to denote the number of elements in the VRs that will be affected by a vector instruction. When the size of the array we need to operate upon is larger than s a technique called *strip mining* is usually applied. Strip mining consists of sectioning the original array into sections equal to s and process each section until the array is consumed. An example will be shown in Section 3.1.2.

Vector Functional Units: In order to support the execution of vector instructions, VPs typically employ pipelined *functional units* (FUs). After issuing a vector instruction the operands are fetched one by one (or two by two etc.) from the VRs and fed to the FUs and the result is then stored back into a VR (or a regular register). Bear in mind that the actual implementation of the FU is an organizational issue which is architecturally hidden. As transistor resources increase we see many VPs that offer Functional Unit Parallelism (FUP), which means that the FU can process more than one operation per cycle. [35] The VLR determines how many elements will be processed by the FU.

Load/Store Unit (LSU) The LSU is a special functional unit that handles the vector memory accesses for instructions that fetch or store arrays to and from the VRs. The number of elements that are stored or fetched is determined from the VLR. Usually three types of accesses are supported by the LSU:

- *Stride-1 accesses*: The vector load instruction fetches contiguous elements from memory. In a vector store the elements of the vector are stored in adjacent positions in memory.
- *Stride-N accesses*: The vector load instruction fetches elements that are separated by N memory locations. Similarly, in a vector store adjacent elements in the VR are stored in memory locations separated by N locations.
- *Indexed Accesses*: A VR serves as an index that contains the memory locations of the elements to be fetched from memory and stored in the target VR. Similarly the index VR is used as an index to store the elements of a second vector into the desired memory locations.

The LSU serves as the interface with the main memory which is typically high throughput and organized in banks.
Chaining: To increase the performance of vector operations most VP employ a technique called chaining. Chaining is similar to result forwarding in Superscalar processors. In the case that the result of a vector operation is needed by a subsequent operation, the result is forwarded to the corresponding FU and at the same time stored in the register.

3.1.2 Vector Instructions

We consider as Vector Instructions (VIs), all the instructions additional to traditional scalar instructions that are related to the vector functionality of a vector architecture. In this section we will not present the full vector instruction set that comprise a vector ISA. Instead we will describe a number of representative instructions for each category and that will be used later in this thesis. The VIs can be distinguished in the following main categories:

- *Setting special registers*: These instructions are not operating on vector registers. However they effect the working of the remaining VIs.
  - *Set the VLR*: The VLR affects all VIs that read or write data into or from vector registers. This means that if the VLR is set to a value L, all subsequent accesses to the vector registers will only affect the first L elements of the register. In most Vector architectures the VLR is visible to the programmer and can be accessed as a regular general purpose register. For example `ADDI VLR, R0, 30` will set the VLR to 30. In the IBM System/370 vector architecture however the VLR is architecturally hidden. In this case the VLR can be set as follows: Consider we have the length of the complete array that is to be processed stored in general purpose register R1. The instruction `SSVL R1` (Subtract and Set Vector Length register) will set the VLR equal to min(R1,s) and update the $\texttt{R1} = \texttt{R1} - \texttt{VLR}$.
  - *Set the BV* The bit vector (BV) can usually be viewed as a regular general purpose register...
- *Memory Access*: These instructions are used to transfer arrays of data from memory to the vector registers and vice versa. The memory access VIs can be subdivided in categories according to their function as described in Section 3.1.1:

  - `LDV R1, VR1` Loads an array of `VLR` data elements into `VR1` from consecutive addresses in memory starting from location `R1`.
  - `STV R1, VR1` Stores `VLR` elements from `VR1` to consecutive memory addresses starting from location `R1`.
  - `SLDV R1, VR1, R2` Loads an array of `VLR` data elements into `VR1` from locations in memory with a stride if `R2` starting from location `R1`.
  - `SSTV R1, VR1, R2` Stores `VLR` elements from `VR1` to in memory with stride `R2` starting from address `R1`.
  - `LDVI VR1, VR2` Loads an array of `VLR` elements from memory locations defined by the corresponding elements in `VR2`.
  - `STVI VR1, VR2` Stores `VLR` elements from `VR1` to in memory locations defined by the corresponding elements in `VR2`.

  IBM System/370 vector architecture where the vector length is not visible to the programmer, the VIs `LDV`, `STV`, `SLDV` and `SSTV` automatically increase the value of the register that contains the address pointer `R1` by the appropriate amount (`VR1` $=$ `VR1` $+$ `VLR` $\times$ `STRIDE`), in order to point to the next section of data in the array in memory.

- *Vector - Vector*: These instructions have only Vector registers as operands. Those include element by element addition, multiplication, XOR etc. A few examples are listed below:
  - `ADDV VR1, VR2, VR3` Adds the `VLR` first elements of `VR2` and `VR3` element-wise and stores the resulting vector in `VR1`.
  - `MULV VR1, VR2, VR3` Multiplies the `VLR` first elements of `VR2` and `VR3` element-wise and stores the resulting vector in `VR1`.
- *Vector - Scalar* These VIs involve a Vector register and a general purpose register. Two examples are listed below:
  - `MULVS VR1, VR2, R1` Multiplies the elements of `VR2` with the scalar in `R1`. and stores the result in `VR1`.
  - `ACCV R1, VR1` Accumulates the elements in `VR1` and stores the result in `R1`.

Examples

To illustrate the presented brief functionality overview of the GVP we will give an example of vector code that performs the following operation:

C[1..1000] = A[1..1000] $*$ k + B[1..1000], where A, B and C are arrays of length 1000 and k is a scalar value:

```
(1)        LA    @A, R1
(2)        LA    @B, R2
(3)        LA    @C, R3
(4)        ADDI  R4, k
(5)        ADDI  R5, 1000
(6)  LP:   SSVL  R5
(7)        LDV   R1, VR1
(8)        LDV   R2, VR2
(9)        MULVS VR1, VR1, R3
(10)       ADDV  VR3, VR2, VR1
(11)       STV   R3, VR3
(12)       BNE   R5, LP:
```

The first five instructions (1)-(5)load the starting addresses of the three arrays into VRs 1-3, the scalar value and length of the arrays in general purpose registers VR4 and VR5 respectively. Instruction (5) sets the VLR to min(VLR,s) where s is the section size of the VP. Also the value of R5 is updated to reflect the length of the remaining portion of the arrays that will be processed at the next repetition of the loop. Subsequently, instructions, (7) and (8) load the array elements into the vector registers. The following two instructions perform the actual multiplication with the scalar value and the vector addition. Instruction STV stores the result into memory to form array C. The last instruction branches to repeat the loop LP: which is repeated until the remaining array length (R5) becomes zero. Note that the instructions LDV and STV automatically increase the value of R1,R2 and R3 by VLR address locations after each execution to ensure that the next section of the array to be fetched from or stored to memory will be from or to the subsequent section to that fetched or stored in the previous loop.

3.2 Proposed Architectural Extension

In this section we will discuss a number of architectural and organizational extensions to the GVP that we propose for supporting the BBCS and HiSM formats that were presented in Chapter 2. The extensions consist of additional Functional Units (FUs) and Vector Instructions (VIs). The new FUs are hardware units that can process arrays of data that represent a sparse matrix and

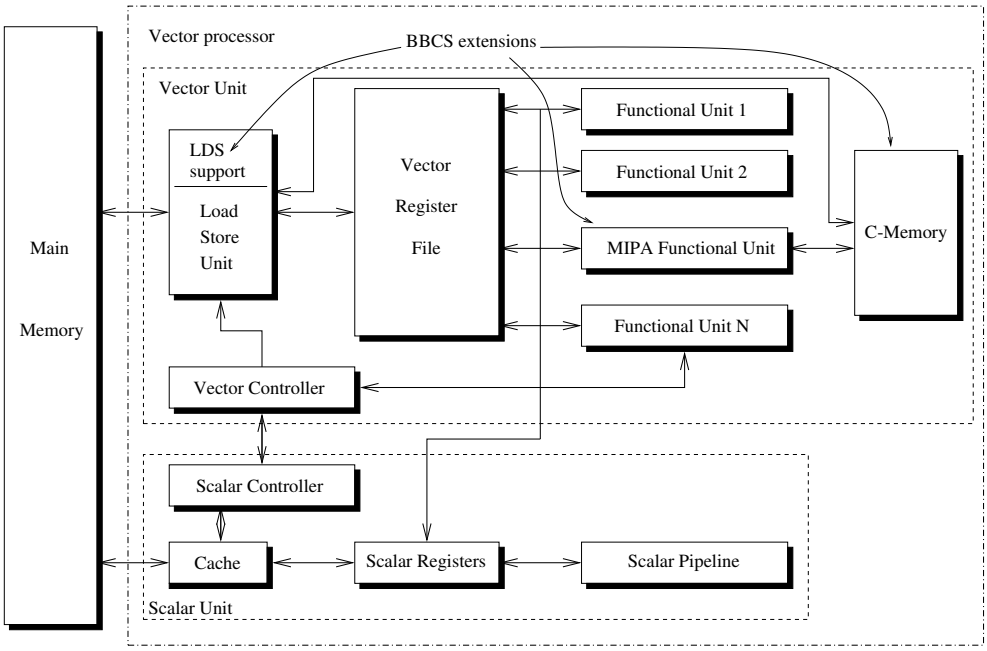


Figure 3.1: BBCS augmented Vector Processor Organization

are stored in the aforementioned formats. Additionally, a number of new VIs ensures that these functional units can be used as vector functional units and so conform to the vector processing paradigm that was described in Section 3.1. First, we will describe a number of extensions associated with the BBCS format in Section 3.2.1 and subsequently for the HiSM in Section 3.3.

3.2.1 Vector Architecture Extension For Block Based Compression Format

We describe here the extensions proposed in order to efficiently perform the Sparse matrix Vector Multiplication using the BBCS format that we have proposed in Chapter 2. The architectural extension consists of two new instructions, `LDS` (LoaD Section) and `MIPA` (Multiple Inner Product and Accumulate), a special vector register `CPR` that holds the positional information produced by the `LDS` instruction and to be used by `MIPA` instructions, and a scalar register `RPR` that holds the current row position in the currently processed VB. Furthermore two extra processor status flags that are updated by the `LDS` instruction, the EOM and EOB, have been added. Figure 3.1 depicts the main extensions to the vector processor organization. The *c-memory* module will be

described in Section 3.3.1

The `LDS` instruction format is `LDS @a,VR1,VR2`. It loads a sequence of BBCS entries from the memory address `@a` into the destination vector registers `VR1` and `VR2`. From the stream of BBCS data entries the non-zero elements are loaded in `VR1` and the index positions of the rows that contain at least one non-zero matrix element are loaded in `VR2`. `VR2` can serve later as an index vector to load $\vec{c}$ sections when performing SMVM. `LDS` will cease fetching BBCS entries on one of the following two conditions: either the vector register `VR1` has already been loaded with s elements, which is the maximum it can contain, or after loading a data entry with at least one of the flags that signal the end of a VB (EOB or EOM flag) set. In the latter case the corresponding EOB or EOM processor status flag are updated consequently. The `CPR` vector register is affected by `LDS` implicitly. After a `LDS` is completed the `CPR` contains the CP fields corresponding to the non-zero matrix elements that are loaded in `VR1`.

The `MIPA` instruction format is `MIPA VR1,VR2,VR3`. The `VR1` vector register is assumed to contain a sequence of non-zero matrix element values from a VB section previously loaded via an `LDS` instruction. The `CPR` vector register, that is used implicitly by `MIPA`, is also assumed to contain the column position information updated by the same `LDS` instruction. `VR2` is assumed to contain the s element wide corresponding section of the $\vec{b}$ vector. Before the execution `VR3` contains the values of the $\vec{c}$ vector elements located at the positions to be affected by the `MIPA` computation. These elements in `VR3` are selected from the vector $\vec{c}$ and loaded with the index vector created by the `LDS` instruction. Generally speaking `VR1` may contain one or more sequences of values that belong to the same matrix row. The end of such a sequence is marked by a set *EOR* flag in the corresponding `CPR` entry. For each of these sequences an inner product is computed with a vector formed by selecting values from `VR2` using the corresponding *CP* index values in the `CPR` register. To better clarify the functioning manner of the `MIPA` instruction we present in Figure 3.2 a visual representation of the `MIPA` operation. In the Figure it is assumed that the last 6 non-zero elements of the second VB, $\vec{A^1}$, are loaded in `VR1`, the corresponding $\vec{b}$ vector section in `VR2`, and the values of the vector $\vec{c}$ to be affected by the `MIPA` computation in `VR3`.

The MIPA Functional Unit

As discussed in the previous section, in order to support the execution of the SMVM on a vector processor using the BBCS format we have introduced an architectural extension. Here we describe a possible implementation of a functional unit dedicated to the execution of the `MIPA` instruction. The hard-

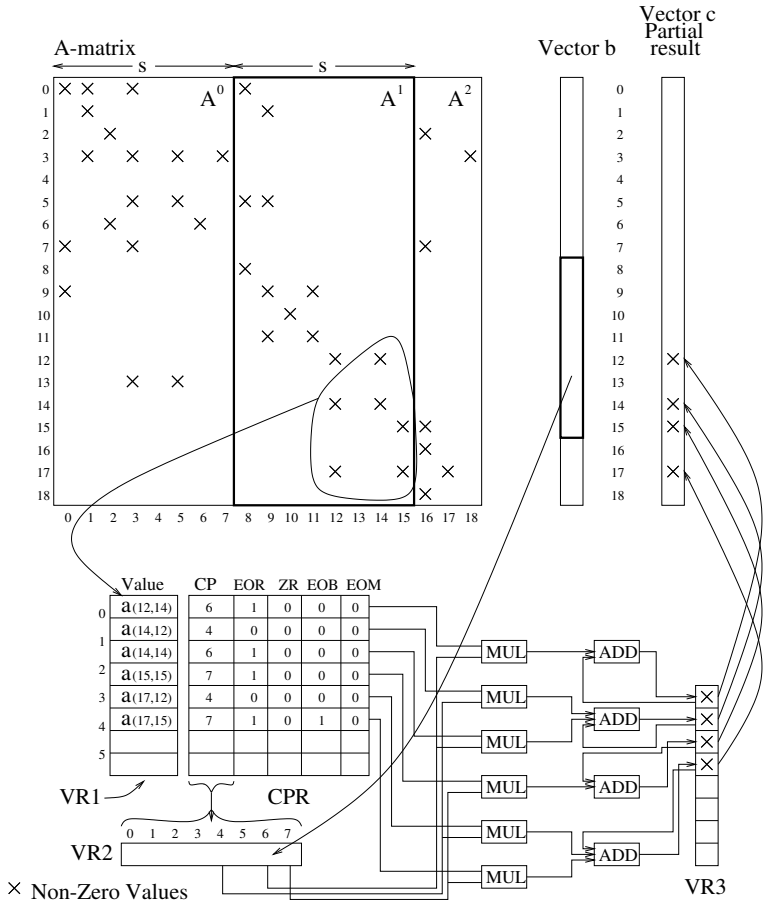


Figure 3.2: The functioning of the MIPA instruction

ware unit is meant to be one of the pipelined functional units of a vector processor as the one presented in Figure 3.1. A 3 stage pipelined implementation of the `MIPA` functional unit is graphically depicted in Figure 3.3.

The operation scenario for the MIPA unit can be described as follows: First, a number of elements in `VR1` vector register as well as their corresponding *CP*s from the `CPR` special vector register are stored in the *A_buffer*. The number of `VR1` elements that can be processed in each cycle is determined by the level of parallelism p within the implementation. In the implementation example in Figure 3.3 p is 4. Then the *CP* entry fields in the *A_buffer* are used to select the "right" elements from the `VR2` vector register. These two sets of data, i.e., the $Values$ in the *A_buffer* and the elements selected from `VR2`,

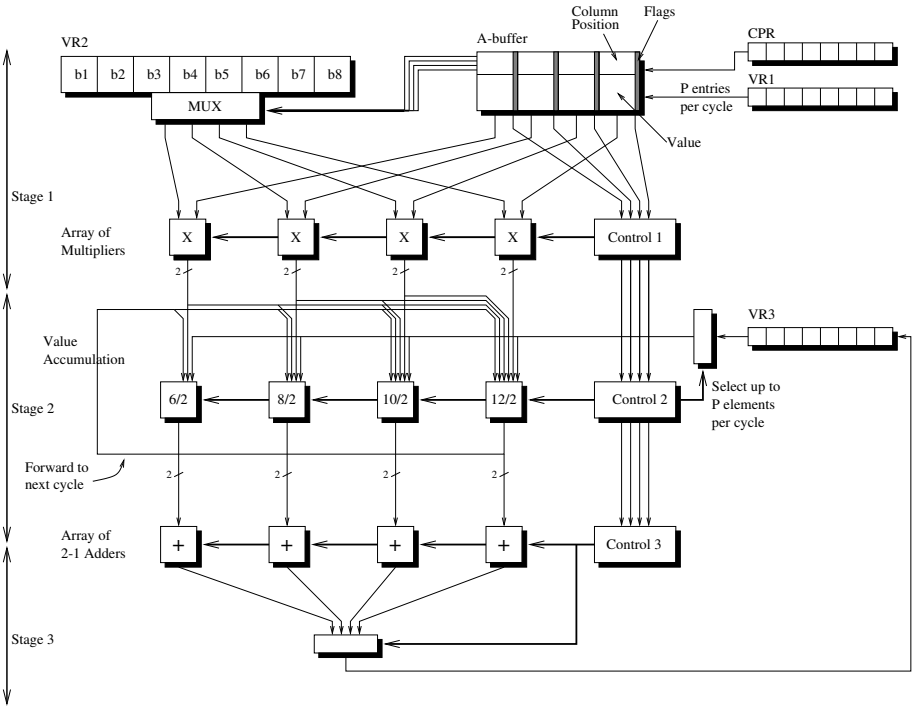


Figure 3.3: $MIPA$ Dedicated Functional Unit

are then pairwise multiplied with an array of p multipliers[2] in the first pipeline stage. As a result of the multiplications p new product results are forwarded to the next stage.

In the second stage the accumulation of the previously computed products is performed. On the basis of the EOR flag of each non-zero element that is contained in the `CPR` register, the $Control\ 2$ unit selects the results of the previous stage to be accumulated to form the final result element(s) and to be stored back in the vector register holding the partial $\vec{c}$ vector. In this accumulation step the current values in `VR3` are also included. This extra accumulation happens only when the last non-zero element of a matrix row is encountered which means that an end result should be produced. The bandwidth needed from `VR3` can vary from 0 to p elements/cycle. Note that when the element that is computed in the rightmost column of the value accumulation array is not the last element of its matrix row the accumulation cannot be completed

[2] Actually as they perform only a multiple-operand addition the multipliers do not contain the final adder stage.

in the current cycle and a partial accumulation result is forwarded to the next cycle at the same stage. Finally, in the third pipeline stage final $2\|1$ additions are performed and results are stored back in VR3.

3.2.2 BBCS transpose Sparse Matrix Vector Multiplication

In this section we present the transposed SMVM using the BBCS format on an AVA. To proceed we will first give a brief presentation of the direct SMVM scheme. For further details on the multiplication using BBCS see [68].

As was mentioned in the previous section, the SMVM for the BBCS scheme is supported by an architectural extension to a traditional vector architecture, which we refer to as an Augmented Vector Architecture (AVA). The extension includes two new instructions, a vector Functional Unit and a special vector register. The first instruction, LoaD Section (LDS), loads the matrix elements stored in BBCS format into the processor and the second instruction, Multiple Inner Product and Accumulate (MIPA), performs the multiplication using the MIPA functional unit. More specifically, suppose we want to multiply a sparse matrix A by a dense vector b. The matrix A is stored in the BBCS format as a sequence of Vertical Blocks of width s where s is the section size (or maximum vector register size). LDS loads a portion (up to s elements) of the VB's non-zero elements into a vector register VR1. At the same time the corresponding column positions of each element are loaded in the special vector register called the CPR (the Column Position Register). A Bit Vector (BV) is also created containing ones at the positions where the corresponding non-zero elements are the last in their rows. Finally, using the positional information implicitly contained in the BBCS format (EOB bits and ZR entries), LDS creates an index vector that contains the non-empty row positions in the Vertical block and therefore the positions of the vector elements that will result from the multiplication. After the execution of the LDS instruction, the MIPA instruction can be executed to calculate the inner products of the rows of the VB with their corresponding values of vector b using the MIPA functional unit. The data flow of the MIPA functional unit is depicted in Figure 3.4 (left). The vector code to perform the multiplication of one VB with the corresponding section of the b vector is as follows (some code details have been omitted for simplicity):

```
Standard Matrix vector Multiplication Code
for BBCS
  LV  VR2, b        ; load section of b in
                    ; vector register VR2
l LDS A, VR1, VR4   ; non-zero elements
```

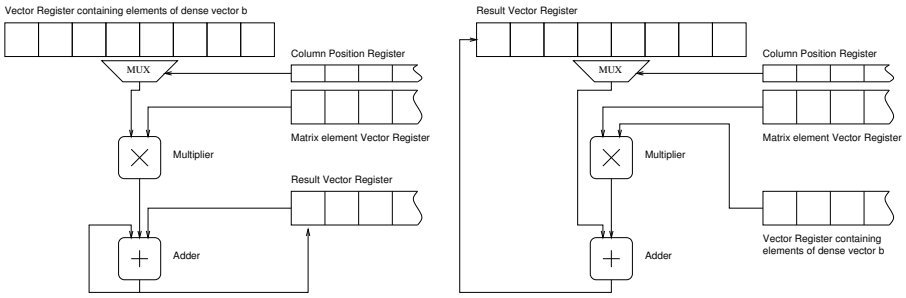


Figure 3.4: Functional difference of MIPA and MIPAT

```
                    ; are loaded in VR1 and
                    ; index vector  in VR4
  LVI VR3, c(VR4)   ; c elements in VR3 from
                    ; address c, VR4 is index
  MIPA VR3, VR1, VR2; multiple inner product
  SVI VR3, c(VR4)   ; store c elements VR2 to
                    ; address c, VR4 is index
Repeat l: for entire VB
```

Note that the `LDS` instruction sets the column position in the special vector register CPR. During execution of the `MIPA` instruction the CPR index values are used to select the correct values from the b vector residing in vector register `VR2` as indicated in Figure 3.4 (left).

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

```
Transpose Matrix Vector Multiplication Code
for BBCS
  SUB VR3, VR3, VR3  ; initialize result
                     ; VR3 = 0,0,0,...
l LDS A, VR1, VR4    ; non-zero elements
                     ; are loaded in VR1 and
                     ; index vector in VR4
  LVI VR2, b(VR4)    ; b elements in VR2 from
                     ; address b, VR4 is index
```

```
  MIPAT VR3, VR1, VR2; multiple inner product
                     ; transposed
Repeat l: for entire VB
  SV  VR3, c         ; store result VR3
                     ; starting from adress c
```

We observe that the access patterns involving vector b and c (the result) have been almost exchanged. As indicated above the working of the `MIPA` instruction has to be altered and therefore we have introduced a new instruction, `MIPAT` (`MIPA` Transposed), similar to `MIPA` to support the transposed SMVM. Due to the exchange of rows and columns, `MIPAT` has a different behavior than `MIPA` as depicted in Figure 3.4. However, despite the functionality difference, the `MIPA` and `MIPAT` can be executed using the same functional unit (FU) if the data flow of the FU is slightly reconfigured depending on the instruction which is executed. In Figure 3.4(right) the differing data flows required by the `MIPA` and `MIPAT` is depicted. Thus, we can implement support for the transposed SMVM on the AVA using the same storage format by only reconfiguring the data flow of the Functional unit of the AVA.

3.3 Architectural extensions for HiSM

We describe here a number of architectural extensions to the previously described GVP that were designed to provide vector functionality that supports sparse matrix operations using the HiSM Format. To support the SMVM using the HiSM format three instructions are introduced.

- LDB R1, VR1, VR2 loads s elements of the S^2-blockarray from memory, starting at position `R1`. The Non-zero elements (or pointers to nonzero blocks) of the s^2-block array are loaded in `VR1`. The position information is loaded in `VR2`. Each element in `VR2` will contain both the row and column position within the s^2-blockarray of the corresponding non-zero element in `VR1`. After issuing, the contents of `R1` will be updated to contain the starting address position of the next section of the s^2-blockarray to be processed.

- MIPAB VR4, VR1, VR3 performs the Block Multiple Inner Product and Accumulate operation. This operation consists of multiplying the non-zero elements of the s^2-block that reside in `VR1` with the corresponding elements in `VR3` (which contains a section of the multiplicand vector) and adding the results to the appropriate element in `VR4` (which contains the result vector of the operation). The positional information is

contained in VR2 which is not part of the operand list of the instruction but is implied.

- STB R1, VR1, VR2 ss the reverse operation of LDB. The contents of VR1 and VR2 are stored in memory using the HiSM format and starting from memory position defined by R1. Again, after issue the value contents of R1 will be updated to contain the starting address position of the next section of the s^2-blockarray to be processed.

3.3.1 SMVM using HiSM

To illustrate the working of the presented instructions we present here an example of how the SMVM for one s^2-block can be performed. Thus we, assume here that our matrix consists of only one $s \times s$-block on level-0. In the general case the SMVM using the HiSM is a bit more complicated and will be discussed in more detail in Chapter 5. We assume that the s^2-blockarray is stored in memory position @A, the multiplicand vector in memory position @B and the result vector will be stored in position @C. The length (or number of non-zero elements) of the s^2-blockarray is stored in R4. Also, we assume that VR4 contains only zeros before execution:

```
(1)      LA    @A, R1
(2)      LA    @B, R2
(3)      LA    @C, R3
(4)      LDV   R2, VR3
(5) LP:  SSVL  R4
(6)      LDB   R1, VR1, VR2
(7)      MIPAB VR4, VR1, VR3
(8)      BNE   R4, LP:
(9)      STB   R3, VR4
```

the first three instructions (1) - (3) load the starting addresses to the s^2-block, the multiplicand vector and the result vector respectively. The next instruction loads the multiplicand vector into register VR3. Note that this instruction is outside the loop since we only need to load the multiplicand vector once for all the elements of the s^2-blockarray. The four following instructions form the main loop that multiplies the s^2-block with the multiplicand vector. In each iteration the VLR is set (5) and the elements of the block are loaded by LDB (6). The actual multiplication of the section of the s^2-block, stored in VR1 and VR2, with the multiplicand vector, stored in VR3, is performed by MIPAB (7) and the result is stored in VR4. VR4 is continuously updated in each iteration.

After the last iteration the loop is exited and the final result (VR4) is stored into memory.

The MIPAB Pipeline: This section describes the working of the MIPAB functional unit. This is a variation of the MIPA unit which was described in the previous section but altered to allow the processing of the arrays stored in the HiSM format. As mentioned, the function to perform is the multiplication of the s^2-blockarray elements and the multiplicand vector to produce the result vector at the output. The unit can consume p s^2-blockarray elements each cycle where p is the parallelism of the unit. An instance of this unit with a parallelism p of four is depicted in Figure 3.5. The unit can be divided into 3 main parts: (a) the value selection, (b) the multiplication and (c) the addition. At the value selection the column information corresponding to each of the p s^2-blockarray elements is used to select the correct p values from the Multiplicand Vector (MR) . Note that the values from MR can be the same if the values in the input buffer belong to the same column. Subsequently, the resulting pairs of values are multiplied resulting into p products. Note that on each stage the row information of each of the partial results that propagate through the pipeline, their row position information is preserved. Following the multiplication, the elements that have the same row position plus the RV value at the same row position are added with each other to produce the final values at the end of the pipeline. There is however one complication which occurs when there are elements belonging to the same row but reside in different stages of the pipeline. This happens for instance when more than p elements belong to the same row. In this case the values resulting from the addition have to be fed back in the pipeline to be added again. To facilitate this we have added a buffer, called the Intermediate Buffer (IB), before the first stage of the addition pipeline. The IB holds several sets of 2 entries: The value of the element and its associated row position. An element is forwarded to the IB when another element with the same row position is either in the addition pipeline or already in the IB. In the second case the element is actually directly forwarded to the first stage of the addition, bypassing the IB. Due to the addition of the IB, the functionality of the first stage of the addition is slightly different than described earlier: In addition to the products from the multiplication stages also the elements residing in the IB are used when grouping the elements according to the row position for the addition. As can be observed in Figure 3.5, up to 7 values can simultaneously enter the addition pipeline and possibly added together (if on the same row). For this reason the addition occupies 3 pipeline stages.

Using this way of processing the data we can achieve a maximum throughput of the elements through the pipeline. The use of the IB for feeding back

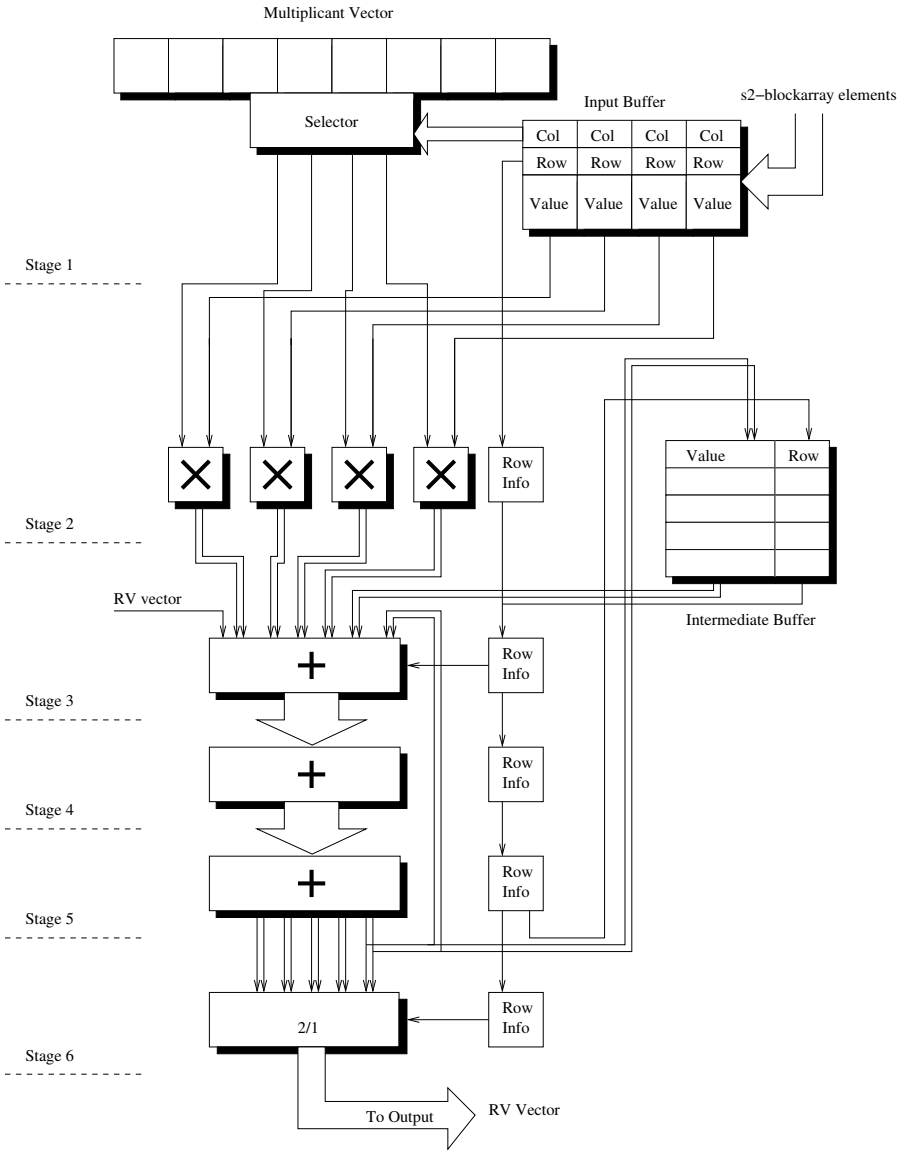


Figure 3.5: The Multiple Inner Product and Accumulate Block (MIPAB) vector functional unit

the elements to the addition pipeline does not inhibit the continuous stream of p elements per cycle that is produced at the multiplication part of the pipeline. This implies that the elements at the output will not appear in the order that

they enter the pipeline.

Timing Evaluation

In this section we will provide performance estimations of the proposed MIPAB mechanism. As we have discussed in the previous section, the processing of the s^2-blockarray elements depends on the distribution of the row elements within the loaded section of the s^2-blockarray that is processed by the MIPA unit. To understand the behavior of the MIPAB unit it is best to consider the two most extreme cases that can appear:

- All the elements in the loaded section of the s^2-blockarray belong to different rows. In this case the output of the MIPAB is a vector of length equal to the input vector.
- All the elements in the loaded section of the s^2-blockarray belong to the same row. In this case the output of the MIPAB is a single value.

In the first case the elements will not have to be fed back in the pipeline and therefore the number of cycles that the mechanism will need to complete the operation will be $\lceil \frac{s}{p} \rceil + w$ where s is the section size of the vector processor, p the parallelism of the functional unit and w the total number of stages of the functional unit pipeline. In the second case we have to note that all elements that pass through the pipeline have to be fed back to the addition pipeline to be added to the remaining elements. The behavior of this pipeline is similar to a regular vector accumulation unit. Therefore the number of cycles to complete the accumulation is given by $\lceil \frac{s}{p} \rceil + w + 7$. The seven extra cycles is the feedback penalty when using 3 stages for the addition pipeline. All other row configurations of the input vector will result in a completion time that lies between $\lceil \frac{s}{p} \rceil + w$ and $\lceil \frac{s}{p} \rceil + w + 7$.

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 3.6. 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 3.6 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 2.7 is transposed. We assume that a part of an s^2 block is stored in a

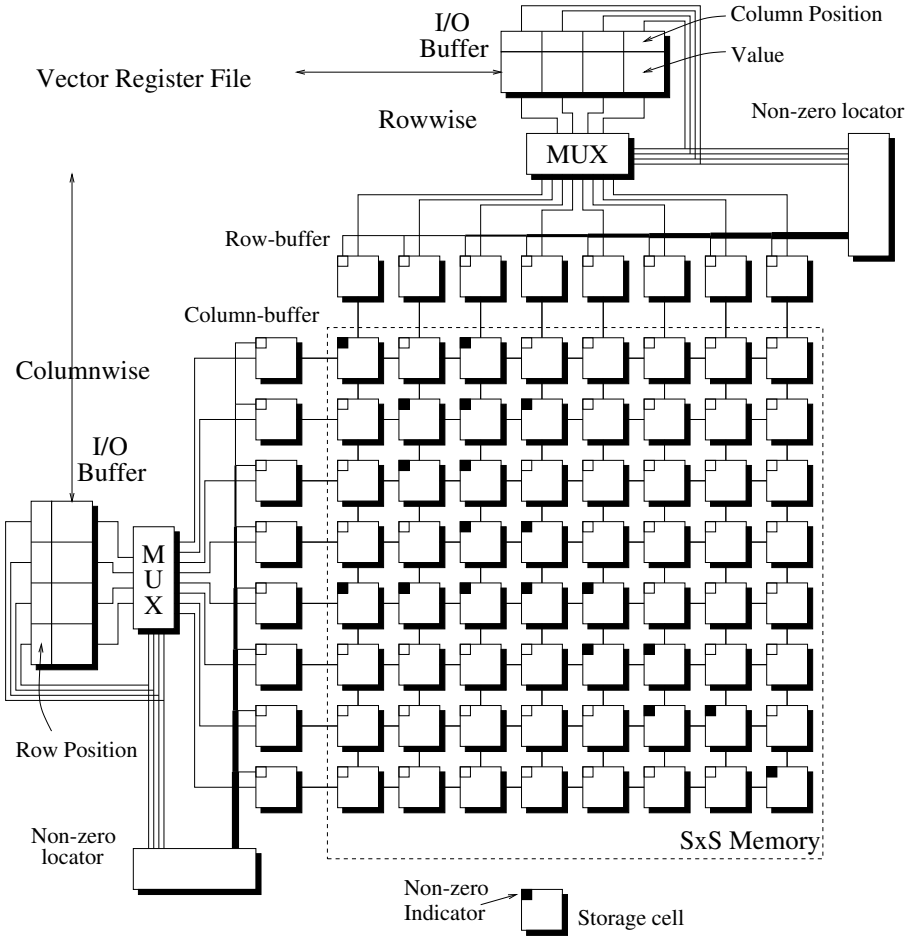


Figure 3.6: The Sparse matrix Transposition Mechanism (STM)

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 cycle and is referred to as the $s \times s$*-memory buffer bandwidth* B, which in the case of Figure 3.6 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 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 3.6). 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 3.7. 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 3.7) 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 3.8 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 3.8, the i-coordinates are as follows: $i = 10$, $i_0 = 2$, and $i_1 = 1$. Similarly, for the column coordinates we have $j = j_0 + j_1 s + j_2 s^2 + \ldots + i_q s^q$. 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

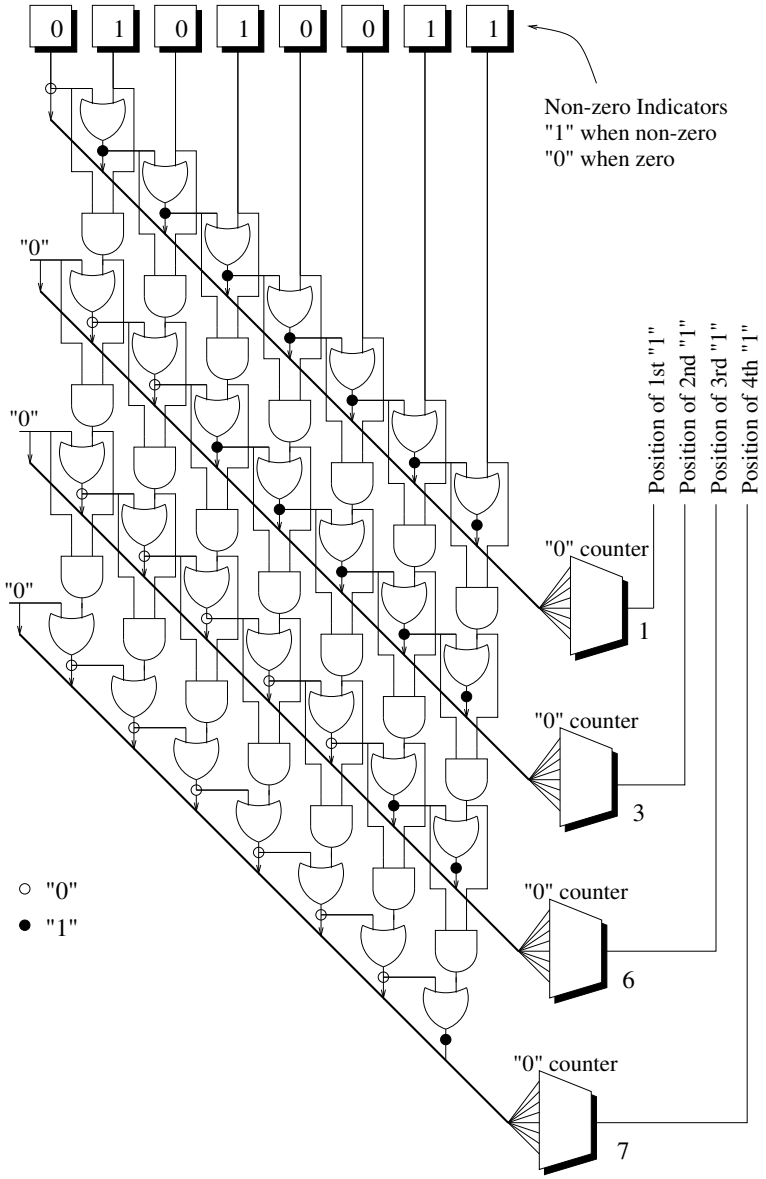


Figure 3.7: The Non-zero Locator

$\tilde{i_0} = j_0, \tilde{i_1} = j_1, \ldots, \tilde{i_q} = j_q$ and $\tilde{j_0} = i_0, \tilde{j_1} = i_1, \ldots, \tilde{j_q} = i_q$. Therefore,

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

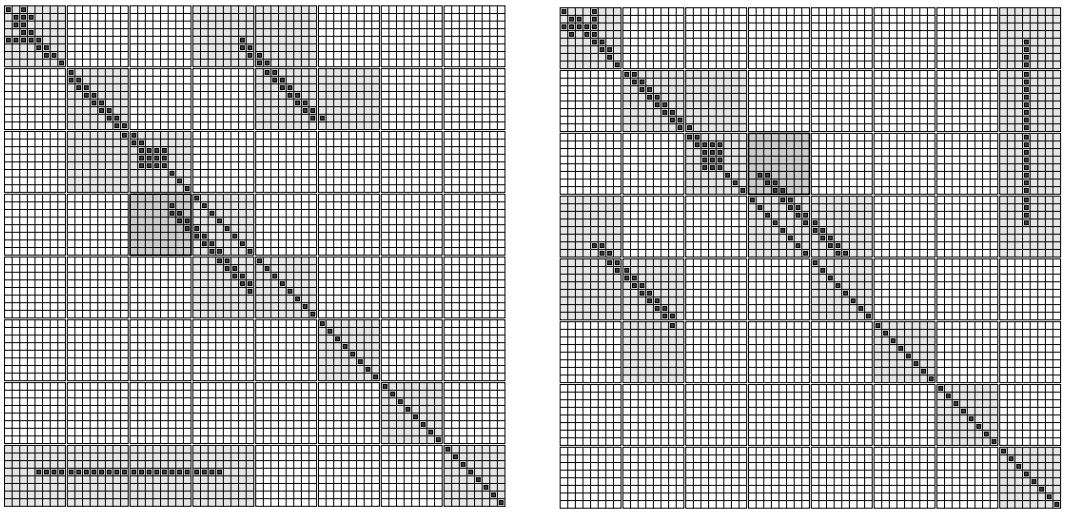


Figure 3.8: Hierarchical Matrix Transposition: Transposing each block at all hierarchies is equivalent to the transposition of the entire matrix

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

Transpose Mechanism Implementation Issues: In order to investigate the implementation details of the STM mechanism, we have created a VHDL model and successfully simulated the behavior of the STM. Subsequently, using the aforementioned model, we have synthesized an implementation of the mechanism on a Xilinx [79] 2v10000ff1517-6 FPGA device. As far as we could search in the literature we have not been able to obtain any other hardware transposition mechanism for sparse matrices and therefore we will not present here a comparison with another mechanism. Instead, we have focused our investigation on the effect of the section size s and the I/O-buffer depth B on the timing of the mechanism. To evaluate the timing we first had to determine the critical path of the circuit. After investigation of the separate stages we have concluded that the critical path is the non-zero locator circuit that we have described in detail in the previous Section. Therefore the latency of the non-zero locator circuit will determine the overall speed of the mechanism. In Table 3.1 the latency of the non-zero locator is depicted for varying values of s and B. It can be observed that the depth of the I/O-buffer has minimal effect on the non-zero locator latency whereas the section size mainly determines the timing of the STM. Additionally, in Figure 3.9 it can be observed that the latency is linear with the section size. The reason for this behavior can be traced

Bs	8	16	32	64
1	8.28 ns	16.49 ns	28.01 ns	49.24 ns
2	8.23 ns	16.49 ns	28.01 ns	49.24 ns
4	8.93 ns	16.68 ns	27.96 ns	49.20 ns
8	11.23 ns	19.24 ns	27.96 ns	49.20 ns

Table 3.1: non-zero locator latency for various s and B

in the design of the non-zero locator which mainly consists of cascaded ports with depth s.

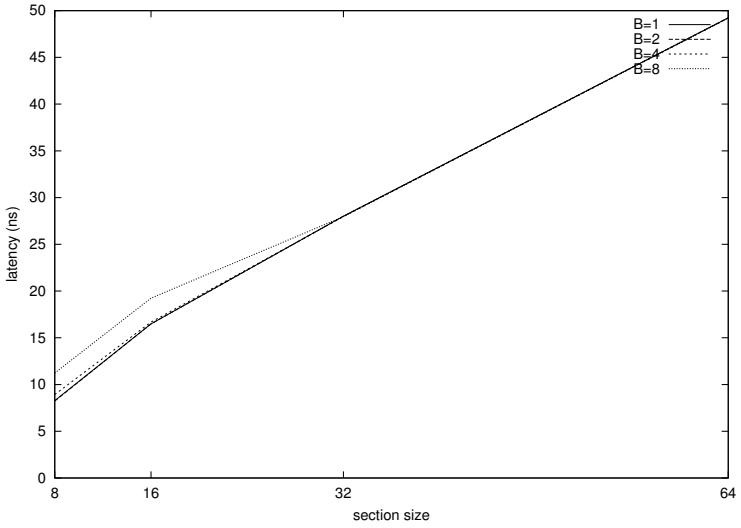


Figure 3.9: Stage Latency

3.4 Conclusions

In this chapter we have presented an architectural and organizational extension to the general vector processing paradigm in order to provide additional functionality to support the efficient operation on sparse matrices when using the BBCS and HiSM formats which were presented in the previous chapter. As a proof of concept, rather than providing a full architectural description which is beyond the scope of this thesis, we have presented architectural support for a number of key operations. First we have provided a brief description of

what we mean by the general vector processing paradigm on which our extension is based. Subsequently, we have described a number of new instructions that access data in memory which are stored in the described formats. Furthermore, we have presented instructions that enable the operation of Sparse Matrix Vector Multiplication both using the BBCS and HiSM schemes and have described the Functional Units which perform these operations in Hardware. Similarly, for HiSM we have provided a mechanism that can support performing the transposition of a sparse matrix.

Chapter 4

Sparse Matrix Benchmark

In the previous chapter we have described organization mechanisms and instruction set extensions to support our proposed formats. In this chapter we present a set of benchmark algorithms and matrices, called the Delft Sparse Architecture Benchmark (D-SAB) Suite for evaluating the performance of novel architectures and methods including our proposal when operating on sparse matrices. The focus is on providing a benchmark suite which is flexible and easy to port on (novel) systems, yet complete enough to expose the main difficulties which are encountered when dealing with sparse matrices. The novelty compared to previous benchmarks is that it is not limited by the need for a compiler thus making it usable for systems under development or systems that do not make use of a compiler, is more flexible in terms of the methods to be used for implementation and makes use of real-world matrices rather than synthetic ones. The D-SAB comprises of two parts: (1) the benchmark algorithms and (2) the sparse matrix set. The benchmark algorithms (operations) are categorized in (a) value related operations and (b) position related operations. The sparse matrix set is comprised by a careful selection of a number of representative matrices from a large existing sparse matrix collection that captures the large diversity of sparsity patterns, sizes and statistical properties.

The Chapter is organized as follows: in Section 4.1 we provide a general discussion regarding sparse matrix benchmarks including a summary of our proposed benchmark. Following that, Section 4.2 discusses previous work, motivation and goals that led us to the development of this benchmark suit. Subsequently the proposed benchmark is discussed in sections 4.3 and 4.5, where the operations and matrix collection are described respectively. Finally, in Section 4.6, some conclusions are drawn.

4.1 General Discussion

Dealing with sparse matrices has always been problematic in the scientific computing world. The reason for this, simply put, is that computers, and especially vector computers, are best in dealing with regularity. However, the arbitrarity of the sparsity pattern of a sparse matrix removes the ability to assume a regular structure of the data and therefore the various algorithms operating on sparse matrix become data dependent and complicated. This degrades the performance of an architecture in terms of resource utilization. A solution to this problem has been to develop different algorithms for doing the same operation depending on the type of the matrix. It has been observed that certain kind of problems give rise to certain kind of sparsity patterns. For those cases a regularity can be assumed that is particular for the specific type of matrix. Typical examples are certain kind of problem areas that will give rise to a sparse matrix that consists of only of a number of diagonals. In this case the sparse matrix can be stored as a collection of those diagonals and the operations can be very straightforward and regular. This approach is thus in essence a transformation of the problem to a regular one that can be dealt efficiently with a computer. It is working well and is often applied within specific disciplines. However, when we consider a more general approach to sparse computations such an approach gives rise to a plethora of different algorithms. Moreover, these are not always applicable and the assumed regularity is not always there. For these reasons there exist also methods (on architectural as well as algorithmic level, or combination of both) that do not assume any type of matrix and are general in their approach. Of course, for the reasons mentioned above, various problems arise in such general approaches and therefore we need a way to evaluate the efficiency of different architectures and more generally techniques in dealing with the problems that arise when sparse matrix computations are involved.

In a similar train of thought a number of proposals have already been made to date for evaluating architectures on sparse matrix operations. These efforts are listed in the next section. The main reason that we propose a new method to evaluate sparse matrix operations rather that use one of the existing ones is that these proposals are mainly targeted for evaluation of a fully functioning system as a whole and not as a tool for early evaluation of the potential of a novel approach. Furthermore, most are not flexible since the method of operation is predefined and the matrices are automatically generated rather than using matrices from actual applications. The proposed D-SAB [62] aims to alleviate the aforementioned shortcomings. The contributions of our proposed benchmark can be summarized as follows:

- We propose the Delft Sparse Architecture Benchmark (D-SAB), a benchmark suite comprising of a set of operations and a set of sparse matrices for the evaluation of novel architectures and techniques. By keeping the operations simple D-SAB is not dependent on the existence of a compiler on the benchmarked system.
- Although keeping the code simple, D-SAB maintains coverage and exposes the main difficulties that arise during sparse matrix processing. Moreover, the pseudo-code definition of the operations allows for a higher flexibility of the implementation.
- Unlike most other sparse benchmarks, D-SAB makes use of matrices from actual applications rather than utilizing automatically generated matrices.

The remainder of the chapter is organized as follows: In the next section, Section 4.2 we discuss previous work on the field, and give our motivation and goals for the development of D-SAB. Subsequently, in Section 4.3 we outline the most commonly used sparse matrix operations which aided us in selecting the operations for our benchmark suite which is described in Section 4.4. In Section 4.5 we describe the matrix collection that is an integral part the benchmark. Finally, in Section 4.6 we give some conclusions.

4.2 Previous Work, Motivation and Goals

To date several efforts have been made that address the problem of benchmarking the performance of various architectures on sparse matrix operations. Some focus exclusively on sparse matrices and others address then in a more general framework of scientific calculations which do also include sparse matrices. Some of the most important are listed below:

- The **Perfect Club** [8] is an acronym for PERFormance Evaluation by Cost-effective Transformations. It is a collection of 13 full applications taken from various fields in engineering and scientific computing and are written in Fortran. A number of these include code involving operations on sparse matrices, mainly linear iterative solvers. The main advantage of Perfect Club set of benchmark programs is that it offers real applications which are used in sparse matrix computations, rather than using only specific small kernels and thus reflects more accurately the real performance that can be expected for a particular architecture.

However, a number of these programs are originally made with a particular architecture in mind and therefore might be optimized in a way that does not benefit a different architecture.

- The **NAS Parallel Benchmarks** [5] are a set of 8 programs designed to help evaluate the performance of parallel supercomputers. The benchmarks, which are derived from computational fluid dynamics applications, consist of five kernels and three pseudo-applications and aim at providing a performance metric for both dense and sparse systems. Only one of the aforementioned kernels is sparse matrix related, the "CG Kernel: Solving an unstructured sparse linear system by the Conjugate Gradient method.". This kernel evaluates the performance of a system when solving sparse linear systems. The choice for CG Kernel has been made since it represents a class (the iterative sparse matrix solvers) of applications that are very frequently used and expose several of the problems that are encountered when operating on sparse matrices. It is interesting to note that the input matrix format is not specified and therefore the implementation details are omitted in the description of the algorithm. The input matrices are provided by a Fortran 77 routine that creates a fully random matrix.

- **SparseBench: a Sparse Iterative Benchmark** As we mentioned above, the solution of sparse matrix linear systems occupies a large deal of the sparse matrix applications and exposes a number of sparse matrix related problems. For this reason an extended Sparse Iterative Benchmark has been proposed by Dongarra et al. [22]. This benchmark uses common iterative methods, preconditioners, and storage schemes to evaluate machine performance on typical sparse operations. The benchmark components are: Conjugate Gradient and GMRES iterative methods, Jacobi and ILU preconditioners. Unlike the NAS benchmark, the algorithms are fully defined (although the possibility for small modifications is left open) in Fortran as well as the input matrix formats. The sparse matrix storage formats used are the (a) diagonal storage format and (b) the compressed row storage (CRS) format.

- **SPARK: A benchmark package for sparse computations** [58] (see also [60]) is a benchmark developed by Saad and Wijshoff to evaluate the behavior of various architectures on the field of sparse computations. The main rationale behind the SPARK approach for designing the benchmark is to try and capture the main kernels that expose the problems encountered in sparse computing. A number of different modules

are presented that capture different aspects of sparse computing: Module 1 includes very basic computational kernels such as dot products, sparse sums, linear combination and so on. All kernels include sparse vectors and mainly expose the performance degradation due to the indirect addressing of the sparse vectors. Module 2 includes the operations "Sparse matrix vector multiplication" and "Forward elimination" using various methods and matrix storage formats. The last module, Module 3, contains matrix manipulation kernels such as transformations to other storage formats, the transposition of a matrix and the extraction of parts of the matrix. The test matrices are automatically generated and are finite element and finite difference matrices.

All the aforementioned benchmarks (except the NAS benchmark) assume a fully functioning system including a compiler. Although this is a very useful approach that reflects real system settings and is easy to use, it cannot be used for architectures in an early stage of development that don't have a compiler or architectures that simply don't make use of a compiler. Furthermore, the benchmarks (except the NAS benchmark) define the storage method of the sparse matrices. Although the storage methods utilized are usually the most commonly utilized in the scientific world, this approach may fail to reveal the full potential of an architecture since the matrix format determines the way it will be accessed and processed. Therefore the flexibility of these benchmarks is limited and does not allow for a novel way of storing, accessing and operating on the sparse matrix. Additionally, the above benchmarks use matrices that are automatically generated. This is mainly done for reasons of memory efficiency. However, we believe that the parameters that are used to generate those matrices cannot capture the diversity of sparsity patterns that are observed in matrices obtained from actual applications.

Our proposed benchmark aims at removing the above shortcomings of the currently existing sparse matrix benchmarks while keeping their benefits. Our benchmark is partly inspired by the NAS and SPARK benchmarks regarding their design philosophy. The advantage of the NAS benchmark is that the algorithm of the benchmark is described in pseudo-code and the storage format is not defined. This allows for flexibility in the implementation of the algorithm. The advantage of the SPARK benchmark on the other hand is that it covers a wide range of aspects of sparse matrix operations thus offering a better covering of the possible sparse matrix operations.

As we have mentioned earlier the goal of this work is to define a benchmark for evaluating the performance of novel architectures and methods when operating on sparse matrices. Before proceeding to describe the benchmark

itself we will give an overview of the requirements we have set in designing this benchmark. These can be summarized as follows:

- **Deal with sparse related problems**. As mentioned before, the proposed benchmark is primarily aimed to evaluate an architecture's ability to cope with the problems that arise by the operation on sparse matrices. We have identified three main issues that have to be addressed in and exposed by our benchmark, namely:
  - *Indexed Accesses*. Due to the sparsity of the rows or columns in a sparse matrix it is very often needed to access the memory using indexed vectors, i.e. non regular rather than continuous or strided accesses. Memory systems are by design performing worse when not accessed in a regular way and therefore the way the benchmarked architecture or method deals with this problem is important.
  - *Short Vectors*. Irrespective of their dimension, most sparse matrices have a small number of non-zero elements per row where small is defined as the number of elements where it does not pay off to treat the elements as a vector due to startup penalties. This too can significantly influence the performance of an architecture or method operating on sparse matrices.
  - *Storage Requirements*. When storing a sparse matrix, the amount of data needed is the sum of the stored non-zero elements and the associated positional information. The latter overhead data can be a significant part of the total amount of data needed to be stored and therefore it is a factor that can significantly influence the number of accesses needed to access the matrix and thus the time needed for processing.
  - *Fill In*. A number of operations dealing with sparse matrices will alter the number or position of the non-zero elements of the matrix. The need to store these displaced or new elements will require a rearrangement or insertion of non-zero elements. Depending on the storage format this can be a tedious and time consuming task.
  - *Flexible Structure*. When manipulating (rather than operating on) sparse matrices it is often convenient that the sparse matrix format and the way it is handled offers the user an untroubled way of manipulating the matrix.
- **Simple Algorithms**. We require the benchmark algorithms to be as simple as possible. Since we are targeting novel architectures it is highly

probable that in the early stages of the architecture development there will be no compiler available that can map the code to the architecture. Therefore the algorithms should be relatively easy to implement in assembler, microcode, hardwired or any other low level means of implementation.

- **Real Applications**. While keeping the benchmark algorithms simple we strive to include algorithms in D-SAB that reflect the sequence of operations that are encountered in a real application. The reason for this is that sequences of operations can have a performance that differs from the sum of the performance of its parts since it is also influenced by dependencies.

Note that this benchmark, similarly to the existing ones, does not relate to any numerical aspect of operating on the sparse matrices such as convergence and rounding errors.

We will describe the benchmark in the following two sections. The first section concerns the algorithms and operations which are executed and timed. The second section comprises of the description of the sparse matrix set that serves as a test suite for the algorithms.

4.3 Sparse Matrix Operations

In this section we outline a number of the most commonly used sparse matrix operations. To construct the benchmark we need to construct a set of algorithms/operations that can cover the basic operations that make up most of the sparse matrix related applications. Rather than analyzing the wide gamma of applications that use sparse matrices (Finite Element Analysis, Fluid Flow, Structural Engineering etc.) we have chosen to examine the various toolkits and packages that offer libraries for operating on sparse matrices since the operations that they offer are mainly driven by the needs of larger applications. A brief description of the most noteworthy is presented below:

- **SPARSEKIT** [57] Extensive library on basic Sparse Matrix operations. Supports 16 sparse matrix storage formats; produces a Post-Script plot of non-zero pattern of a matrix. It does basic algebraic operations, ordering, computes statistics and performs some iterative methods.

- **The NIST Sparse BLAS** [14, 53] The package includes support for a basic toolkit, including matrix-multiply and triangular solve routines.

- **LASPACK** [61] A C package for solving large sparse systems of linear equations, especially those arising from discretizations of partial differential equations. Contains both classical and state-of-the-art algorithms such as CG-like methods for non-symmetric systems (CGN, GMRES, BiCG, QMR, CGS, and BiCGStab) and multilevel methods such as multi-grid and conjugate gradients preconditioned by multi-grid and BPX.

- **Sparselib++** [21] The Sparse BLAS Toolkit is used for efficient kernel mathematical operations (e.g. sparse matrix-vector multiply) and to enhance portability and performance across a wide range of computer architectures. Included in the package are various preconditioners commonly used in iterative solvers for linear systems of equations. The focus is on computational support for iterative methods.

After examining the aforementioned sparse packages/toolkits we have compiled a list of operations that are important when dealing with sparse matrices. In the following two sections we present the operations divided in two main categories, (a) Unary matrix operations and (b) Arithmetic operations. Along with each operation we mention the challenges and implications of the corresponding operation.

4.3.1 Unary Matrix Operations

This list contains mainly operations that have as their only operand the sparse matrix itself.

- *Submatrix Extraction*: Extract a rectangle or square matrix from a sparse matrix. Select a_{ij} from matrix $A[a_{ij}]$: $r_h < i < r_h$ and $c_h < j < c_h$, where r_h,r_l,c_h and c_l define the boundaries of the submatrix for upper-row, lower-row, upper-column and lower-column respectively.
  Implications: Unlike the elements of dense matrices, sparse matrix element coordinates are not implied by their position in storage. This means that the elements have to be searched and selected according to their positional information which is usually stored separately.

- *Filter Elements*: This operation filters out the elements of a matrix depending on the magnitude of the element, for instance filtering out the elements whose value is lower than a specific value.
  Implications: The creation of a new structure for storing the new matrix.

- *Transposition*: Create the transpose of matrix A, A^T. This operation is usually not used by itself but more in conjunction with another operation like for instance the sparse matrix multiplication with a vector: $A^T x$. However, the reason for this is often that it is a difficult operation since it can require a full rearrangement of the matrix storage structure. Current methods for operating on sparse matrices generally are optimized for accessing the matrix as regular A and not as the transposed A^T. A similar problem will be discussed later considering the operation $A^T x$.

- *Get Element*: Return the value of element a_{ij} of a sparse matrix $A[a_{ij}]$.
  Implications: The positional structure of the stored matrix needs to be searched in order to locate the element. In most storage methods, if the element does not exist in the stored matrix then the returned value is zero.

- *Get Diagonal*: Extract a diagonal of the sparse matrix: Select elements a_{ij} for $i = j + k$ where k is a constant denoting which diagonal is extracted, $k = 0$ represents the main diagonal.
  Implications: The implications for this operation are similar to the previous operation.

- *Extract Lower/Upper Triangular Matrix*: Extracts the elements a_{ij} of a matrix $A[a_{ij}]$ where $j \leq i$ for the Lower Triangular and $i \leq j$ for the Upper Triangular. the implications for these operations are similar to the *Submatrix Extraction* operation.

- *Row/Column Permutation*: Permute the rows or columns of a sparse matrix. These operations are equivalent to $B = PA$ for row permutation and $B = AQ$ for column permutation. Both P and Q are *permutation matrices*.
  Implications: The elements of the matrix will need to be reordered. Depending on the flexibility of the storage method used this can be a time consuming process.

- *Maximum Value per Row* Return a vector that contains the maximum value of each row of the matrix.
  Implications All the elements of the matrix need to be searched. (...)

- *Non-Zeros in AB* Calculate the number of non-zeros of the matrix-matrix multiplication $C = AB$. This is related to calculate the space needed to store the result of the multiplication without actually loading the values of the elements (only the positional information is needed)

and carrying out the actual computations. *Implications*: Are similar to the carrying out the matrix-matrix multiplication $C = AB$ which will be discussed further on.

- *Extract Given Row(s)*: Extract specific row(s) of the matrix.
  Implications: The complexity of the problem is similar to extracting one element from the matrix. However, depending on the sparse matrix storage format used, the process can be made faster than the sum of the time extract element separately.

- *Compute Norms*:
  Compute the norm $\|.\|_n = \sqrt[n]{|r_1|^n + |r_2|^n + \cdots + |r_k|^n}$ of the rows of a matrix where $r_1, r_2, \ldots, r_n$ represent the elements of the row. Usually three norms are used: $\|.\|_1 = |r_1| + |r_2| + \cdots + |r_k|$, $\|.\|_2 = \sqrt{|r_1|^2 + |r_2|^2 + \cdots + |r_k|^2}$ and $\|.\|_\infty = max(|r_1|, |r_2|, \cdots, |r_k|)$.
  Implications: We are primarily interested in the accessing of the row elements rather than the calculations themselves that are not specifically related to sparse matrix computations. Therefore the implications are similar to accessing one or more rows of a sparse matrix.

4.3.2 Arithmetic Operations

This list contains arithmetic operations on the sparse matrices.

- *Sparse Matrix-Dense vector Multiplication (SMVM)*: $y = Ax$ where x and y are dense vectors. Each k^{th} element of vector y, y_k is calculated as follows: $y_k = \Sigma_{i=0}^{n} a_{ik} x_i$. This is generally considered to be the most important operation in sparse matrix applications since it is the main part (in terms of consumed cycles) of most solver kernels.
  Implications: All the elements of the matrix have to be accessed in order to perform the computation.

- *Matrix-Matrix Addition* $C = A + B, c_{ij} = a_{ij} + b_{ij} \forall i, j$. *Implications*: Since in most sparse matrix storage formats the zero elements are not explicitly stored the storage space required is mainly defined by the number of non-zero elements. This implies that unless the sparsity patterns (the distribution of non-zeros over the matrix) of A and B are the same, the resulting matrix will have more non-zeros than either A or B and the size of the stored matrix C will not be known in advance. Special care should be taken to deal with this problem efficiently.

- *Matrix-Matrix Multiplication*: $C = AB$, $c_{ij} = \Sigma_{k=0}^{n} a_{ik} b_{kj}$ where n is the number of columns in A and the number of rows in B.
  Implications: The implications are similar to the matrix-matrix addition operation described earlier. Additionally, this operation exhibits a high level of data reuse which can be taken advantage of.

- *Lower Triangular System Solution*: Solves the equation $Lx = y$ for x, where L is a lower triangular matrix, $l_{ij} = 0, \forall (i,j) \in \{(i,j) | i < j\}$

- *Pivoting & Gauss Elimination*: This is not in essence a basic operation in the complexity sense and is the most complex of the ones described here. However it is an operation that is in the core of many (sparse) matrix operations since it is used for linear system solving and often in preconditioners for iterative linear system solving [7]. We will therefore cover this operation more extensively in the following paragraph.

The Gaussian Elimination is a standard direct method for solving linear equations of the form $Ax = b$ where b is a known vector. The purpose is to create an upper triangular matrix U from matrix A by subtracting multiple of rows from others in order to create zero entries at the lower triangular part. The procedure for Gaussian elimination is described as follows: Assume an $n \times n$ matrix $A = [a_{ij}]$ where a_{ij} is an elements of the matrix at row i and column$j$.

In order to create the first zero at position $(2,1)$ we start by multiplying the first row $(a_{11}, a_{12}, \ldots, a_{1n})$ by $-\frac{a_{21}}{a_{11}}$ and add the resiting vector from the second row $(a_{21}, a_{22}, \ldots, a_{2n})$ which creates a zero entry at position $(2,1)$. Consequently, the the first row is multiplied by $-\frac{a_{31}}{a_{11}}$ and add to the third row to create a zero at position $(3,1)$. The process is repeated for each row until all elements under a_{11}, are zero. After the elimination at the first column we proceed to the submatrix that is formed by removing the first row and column and repeat the previously described procedure.
To generalize, we use the k^{th} row to eliminate the element a_{ij} we proceed as follows: Multiply k^{th} row $(a_{k1}, a_{k2}, \ldots, a_{kn})$ by $-\frac{a_{ij}}{a_{kj}}$ and add to the i^{th} row, $(a_{i1}, a_{i2}, \ldots, a_{in})$.

For reasons of numerical stability a matrix is transformed before performing elimination by choosing the largest absolute element value in the matrix prior to the process of elimination in a column and moving this to the diagonal. This procedure is called **Pivoting**. A commonly used variation of pivoting is to choose the largest value in the column (rather than the whole matrix) that is to be eliminated. This is the version of the algorithm that we have chosen to use for our benchmark.
The complete procedure of the Gauss elimination with pivoting can be

described as follows:

for j = 0 to $n-2$ do
$\Rightarrow$ Find position (q) of max abs in column C_j
$\Rightarrow$ if $q <> j$ then exchange rows(j, q)
  for $i = j$ to $n-2$ do
    $R_{j+1} = R_{j+1} - a_{ij}a_{jj}R_j$
  end for
end for

C_k denotes the k^th row, $R_k = (a_{1k}, a_{2k}, \ldots, a_{nk})$, R_k denotes the k^th row, $R_k = (a_{k1}, a_{k2}, \ldots, a_{kn})$ and the $\Rightarrow$ signs indicate the part of the algorithm that is used for pivoting.

4.4 Benchmark Operations

In this section we describe the sparse matrix operations that we have chosen to form our benchmark operation suite. We have observed that although most packages offer an extensive set of functions, there is a plurality of functions performing the same operation, relating to the fact that these functions are implemented for various storage formats. However, in our benchmark the aim is to be independent of the storage format and thus we have chosen only the basic functions, one from each function class. After an initial analysis of the above packages we have chosen to divide the basic operations in two parts:

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.

2. *Position Related Operations* (PROs). These include operations for which the actual values of the elements are not important for the outcome, such as element searching, element insertion, matrix transposition, etc.

We have chosen 5 operations from each of the VROs and PROs that we believe represent the most basic operations of sparse matrix applications and satisfy the goals and requirements that we have set in the previous section. The benchmark operations for the VROs and PROs are listed in Tables 1 and 2 respectively.

Name	Operation	Description
1. Multiplication	$C = AB$	Multiplication of two sparse matrices. This operation has a high degree of value reuse and indicates how a method can deal with this fact.
2. Addition	$C = A + B$	Matrix addition exposes fill-in (i.e. the addition of extra nonzero elements in a sparse matrix)
3. SMVM	$y = Av$	Sparse Matrix - dense Vector Multiplication. This operation is one of the most important in sparse matrix computations in terms of execution time.
4. Gaussian Elimination, Pivoting	see text	Operations used in Direct Methods for linear system solving and the construction of preconditioners
5. (Bi)Conjugate Gradient	see Fig 4.4.1	(Bi)CG, 2 iterative solvers, typical sparse matrix applications. The main benchmark for most existing sparse matrix benchmarks

Table 4.1: Value Related benchmark operations

4.4.1 Value Related Operations

The Position Related Operations (VROs) are listed in Table 4.1. A brief description of operations follows. We begin by describing matrix to matrix multiplication and addition followed by sparse matrix vector multiplication, pivoting for Gaussian elimination, concluding with the (Bi)Conjugate Gradient. **Matrix-Matrix Multiplication**: The first operation, Multiplication of two matrices is defined as follows: Assume an $n \times k$ matrix A, a $k \times m$ matrix B and an $n \times m$ matrix C with corresponding elements a_{ij}, b_{ij} and c_{ij} for position (i, j) respectively. Then the product $C = AB$ is defined as

$$c_{ij} = \sum_{0}^{k-1} a_{ik} b_{kj} \forall i \in \{0, \ldots n-1\}, j \in \{0, \ldots m-1\} \qquad (4.1)$$

This operation has a high degree of reuse since each of the elements of A are accessed m times and the elements of B n times. Therefore, the memory bandwidth can be a bottleneck. For the dense case there exist methods to make clever use of the reused data and rearrange the accesses in such a way that the accesses to external memory are minimized. In the sparse case however the multiplication is more complicated; because of the various sparsity patterns of different matrices it is not known in advance which values are going to be reused. Moreover the exact number of non-zeros in the resulting matrix (C) is also not known.

Matrix Addition: The second operation, the Addition of two $n \times m$ A and B matrices is simply defined as the pairwise addition of each element in the same position in the matrices. Therefore $C = A + B$ is calculated as follows:

$$c_{ij} = a_{ij} + b_{ij} \forall i \in \{0, \dots n-1\}, j \in \{0, \dots m-1\} \tag{4.2}$$

This seemingly trivial operation when used with sparse matrices gives rise to the fill-in problem, which implies that the number of non-zero elements in the resulting matrix will be equal or larger that any of the two added matrices. The number of non-zeros will not be known without carrying out the addition or examining the sparsity pattern prior to the addition. Fill-in is a problem for two reasons:

1. It is not known how much memory will be needed to store the resulting matrix. The number of non-zeros in the resulting matrix can be anything between the number of non-zeros in any of the matrices in the case that the sparsity pattern of matrix A is equal with the sparsity pattern of matrix B, to the number of non-zeros of A and B taken together. The latter will occur in the case that $\forall(i,j), a_{ij} \neq 0 \Rightarrow b_{ij} = 0$ and $\forall(i,j), b_{ij} \neq 0 \Rightarrow a_{ij} = 0$.

2. In the case that $A = A + B$ is computed the new elements will have to be inserted to matrix A. Depending on the way the matrix storage structure is built, long arrays of elements may need to be shifted in order to insert new elements.

To add to the complexity, in both cases merely knowing the number of non-zeros of the resulting matrix does not guarantee (in most formats) what the memory requirements will be for storing the matrix since it is also dependent on the structure of the matrix. The amount of memory will be known only after the addition or an analysis of the sparsity patterns of the matrices to be added. Another approach is for the addition algorithm to provide for a means to dynamically allocate memory during addition or, a technique which is frequently

applied, to allow for some elbow room (e.g. considering the worst case scenario) when allocating memory, use an intermediate format, and subsequently rebuilding the matrix storage structure after the addition is performed.

Sparse Matrix dense Vector Multiplication (SMVM): As we mentioned before, SMVM is considered to be the most important operation in sparse matrix operations since it constitutes the core operation of iterative solvers which are the most commonly used tool in linear system solving when sparse matrices are involved. The Multiplication of an $n \times m$ matrix A with a vector b resulting in a vector c is defined as follows:

$$c = Ab \Leftrightarrow c_i = \sum_{j=0}^{m} a_{ij} b_j \tag{4.3}$$

The SMVM is an operation that exposes many of the previously mentioned problems that arise when operating with sparse matrices. Those problems are: short vectors, indexed accesses and Storage requirements as described in Section 4.2. For this benchmark operation the vector (b) to be multiplied with is a dense vector.

Bi-Conjugate Gradient (BiCG): Figure 4.4.1 depicts the code for the BiCG algorithm where x, p, z, q, $\tilde{p}$, $\tilde{z}$ and $\tilde{q}$ denote dense vectors and Greek letters denote scalars. The $\Rightarrow$ signs indicate the code lines of interest since they form the asymptotic execution of the code. We have included the BiCG code along the CG code because it includes SMVM with both the A and A^T. Performing both in the same code is considered troublesome and is avoided in practice in spite the fact that algorithmically it can offer advantages. However we believe that this precisely the reason to include it in our benchmark.

Note that for all the Value Related Operations of the benchmark that were just described the actual values of the matrix elements are not of any interest since we are not focusing on the numerical aspect of the operations but the handling of the data structures.

4.4.2 Position Related Operations (PROs)

The Position related operations in this benchmark are mainly used to evaluate the flexibility of the matrix storage structure and the ability of the architecture to handle this structure. When storing dense matrices the position of an element in the matrix is implied by the position in the memory. I.e. if an $n \times n$ dense matrix A is stored row-wise in memory starting at memory address $\&A$, the element a_{ij} at position (i, j) can be accessed by simply accessing memory address $\&A + ni + j$. However, in most sparse matrix storage formats this is

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

Figure 4.1: The Non-preconditioned Bi-Conjugate Gradient Iterative Algorithm.

not the case and a more complicated and more importantly, a data dependent algorithm is needed to access the correct element or part of the matrix since the positional information has to be accessed first. Furthermore, since the size of memory storage that is allocated in memory for the matrix is dependent on both the number of non-zeros as well as the structure of the matrix, any modification can influence the allocated memory size and require rebuilding the correct storage structure. These can therefore be rather time consuming operations and therefore they have been included in our benchmark.

- **Submatrix Extraction** The starting position of the matrix extract operation has been chosen to be $(5, 10)$ for two reasons: (a) the position should not be the origin $(0, 0)$ since this might give a benefit to certain matrix storage structures but were chosen to be small in order to be able to test small matrices (small dimensions). (b) Position $(5, 10)$ implies that resulting matrix will not be extracted symmetrically around the diagonal. Again the reason is not to give an advantage to storage structures that might benefit when the extracted matrix is symmetric around the di-

Name	Description
6. Sub-matrix Extraction	Create a new matrix by extracting a sub-matrix from matrix A. Start from position $(5, 10)$ Use sizes 10x10, 100x100, 1000x1000, 10000x10000 if the original matrix size permits to do so.
7. Transposition (A^T)	Create a new matrix that is the transpose of the original.
8. Get element from matrix	Return the time needed to access an element in the matrix averaged over 50 values randomly chosen over the whole matrix. At least 10 should return a non-zero value.
9. Extract Lower Triangular Part	Create a new matrix that comprises only of the elements a_{ij} of matrix A where $i \geq j$.
10. Insert or Modify Element	Modify a non-zero Element in the matrix or Insert an element in the matrix (modify a zero entry) .

Table 4.2: Position Related benchmark operations

agonal.

- **Transposition** The transposition $B = A^T$ of a matrix A is the exchange of the columns and with the rows of the matrix and vice versa. This implies that after the transposition, the matrix B will contain the value a_{ji} at position (i, j). This operation is not used very often in sparse matrix applications. However, the main reason for this is that it is a complicated operation and most sparse matrix formats are built in such a way that only one way of accessing the matrix (row-wise or column-wise) is efficient and only that one is used. See also the discussion in Section 4.4.1

- **Get Element from Matrix** Get the value of a random element in the matrix. The time to retrieve a value is averaged over 50 random ele-

ments. Since sparse matrices can exhibit a very high degree of sparsity, in order not to get only elements with value zero, we require that at least 10 of those elements must return a non-zero value. This in order not to favor a matrix where it is easier to determine whether a certain non-zero exists or vice-versa.

- **Extract Lower Triangular** Create a new matrix that comprises of the Lower Triangular part of the original. That is, the new matrix comprises only of the elements a_{ij} of matrix A where $i \geq j$.

- **Insert or Modify Element** This is in essence only one operation, the modification of an element. In the case the element in question is a non-zero, then in most sparse matrix formats this is a trivial operation (excepting locating the element). However, the case where the original element is a zero, we might have to modify large portions of the matrix to keep the format consistent. Note that this operation is not a pure VRO since it modifies the matrix elements, however we have chosen to include it here due to the fact that the interesting part of this operation is that it requires the modification of the sparse matrix structure and is therefore a good indicator of the flexibility of the matrix format.

All ten named benchmarks are to be executed using the benchmark matrices that are listed in the following section. Wherever a second matrix is needed (i.e. the B Matrix in Addition and Multiplication) we construct it as follows: The B matrix is the A matrix mirrored around the second diagonal, that is, the top right to bottom left diagonal. We have chosen to do so because the sparse matrix suits from which we have chosen our benchmark matrices do not offer pairs of matrices of the same dimensions. Therefore, to avoid the fill-in free addition of the matrix with itself we mirror the matrix at the diagonal where most of the matrices are not symmetric.

4.5 The Sparse Matrix Suit

The second part of the benchmark environment concerns a common reference of a set of matrices that the previously described algorithms are applied on. This section will describe this matrix set. First, existing matrix sets will be discussed, next the matrix selection procedure is described and finally our matrix set is presented.

The set of matrices that we have compiled for the benchmark have been selected from already existing sparse matrix collections. There are several

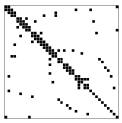
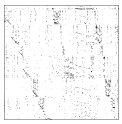
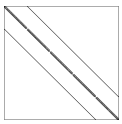
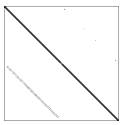
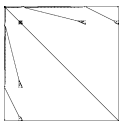
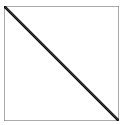
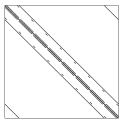
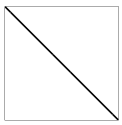
Sparsity Pattern	Name Dimension Non-zeros	Statistical Properties	Short Description
	bcspwr01 39x39 131	Locality: 102 Avrg nz per Row: 3 Largest Row: 6	BCSPWR01: Power network patterns Standard IEEE test power system – New England from set BCSPWR, from the Harwell-Boeing Collection
	rw136 136x136 479	Locality: 115 Avrg nz per Row: 3 Largest Row: 4	RW136: Markov Chain Transition Matrix generated by MVMRWK, from the NEP Collection
	can__161 161x161 1377	Locality: 268 Avrg nz per Row: 8 Largest Row: 17	CAN 161: Structures problems in aircraft design from set CANNES, from the Harwell-Boeing Collection
	bp___800 822x822 4534	Locality: 27 Avrg nz per Row: 5 Largest Row: 304	BP 800: Original Harwell sparse matrix test collection Simplex method basis matrix from set SMTAPE, from the Harwell-Boeing Collection
	orsreg_1 2205x2205 14133	Locality: 103 Avrg nz per Row: 6 Largest Row: 7	ORSREG 1: Oil reservoir simulation - generated problems oil reservoir simulation for 21x21x5 full grid from set OILGEN, from the Harwell-Boeing Collection
	dw8192 8192x8192 41746	Locality: 103 Avrg nz per Row: 5 Largest Row: 8	DW8192: Square Dielectric Waveguide from set DWAVE, from the NEP Collection
	memplus 17758x17758 126150	Locality: 91 Avrg nz per Row: 7 Largest Row: 574	MEMPLUS: Computer component design memory circuit from set HAMM, from the Independent Sets and Generators generated by MVMTLS
	af23560 23560x23560 484256	Locality: 189 Avrg nz per Row: 20 Largest Row: 21	AF23560: Transient Stability Analysis of a Navier-Stokes Solver from set AIRFOIL, from the NEP Collection
	conf5-4-00l8x8-0500 49152x49152 1916928	Locality: 248 Avrg nz per Row: 39 Largest Row: 39	CONF5.4-00L8X8-0500: Quantum Chromodynamics Quantum chromodynamics, b=5.4, kc=0.17865 from set QCD, from the Independent Sets and Generators generated by MVMTLS
	s3dkt3m2 90449x90449 3753461	Locality: 464 Avrg nz per Row: 41 Largest Row: 42	S3DKT3M2: Finite element analysis of cylindrical shells Cylindrical shell, uniform 150x100 triangular mesh, R/t=1000 from set CYLSHELL, from the Independent Sets and Generators generated by MVMTLS

Table 4.3: 10 Matrices sorted according to the Number of Non zeros

such collections that provide the opportunity to evaluate the performance of new sparse algorithms and methods [11, 24]. The various collections strive to cover a wide variety of different application areas since each problem type gives rise to sparse matrices with different properties. These properties include various matrix statistics related to the sparsity pattern and the actual values of the non-zeros. The latter include spectral portraits and various norms.

The benchmark matrices for the D-SAB suite were chosen from a wide variety of matrices that are available from the Matrix Market Collection [11]. 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.

The Matrix Market matrices represent various fields in scientific computing where sparse matrices are involved. These fields include Power network modeling, Structural Engineering, Finite element modeling, Finite-element structures problems in aircraft design, Quantum chromodynamics, Fluid dynamics, Chemical kinetics problems, Simulation of computer systems, Magnetohydrodynamics, Hydrodynamics, Chemical engineering plant modeling, Acoustic scattering, Oil reservoir modeling and so on. In order to reduce the number of matrices but at the same time retain the variety of the set intact we have chosen a few (around 3) representative matrices from each of the various sub-collections included in the Matrix Market Collection. The main criterion was the size of the matrix since the other non size related properties (i.e. sparsity pattern, number of non-zeros per row etc.) tend to be similar within a subcollection due to the similar nature of the problem. For most sub-collections a small, medium and large matrix were selected.

The above described selection process yielded 132 matrices. However, despite the variety of the covered applications, the 132 matrices exhibit a redundancy in terms of the various matrix properties. Considering this, we wanted to reduce the number of matrices to a smaller, more manageable number for the final set of matrices for the test matrix suit. This resulted in three sets of ten matrices that will be presented further on. Each of these three sets correspond to one of the three metrics (described in detail further on) which we have defined and were compiled as follows: For each metric the 132 matrices were sorted using that metric as the sorting criterion. Subsequently, from each of the three resulting sets of 132 matrices we chose ten matrices. For clarity this means that the first set comprises of ten matrices chosen from the 132 matrices when sorted by the first criterion, the second set comprises of ten matrices chosen from 132 when sorted with the second criterion and so on. For each set, the ten matrices were chosen to represent the complete range of the sorted matri-

ces by having a constant logarithmic distance in the corresponding metric. I.e. in the case where the metric is the matrix size, we have chosen each matrix to be approximately 3.5 times larger than the previous one with the largest being $3.5^9 = 78815$ times larger than the smallest. This has been preferred since we have observed that for all the criteria, the distribution of the matrices' value for the corresponding metric was logarithmic rather than linear.

We have chosen the three criteria of sorting the matrices to be such that they represent the variety of properties of the matrices and expose the cases where different sparse matrix problems arise. The three metrics that we have used are the following:

- *Matrix Size*. The metric is the number of non-zeros within the matrix. Note that this metric is not, and is not necessarily related to the dimensions on the matrix. The size of a matrix can significantly affect the performance of an operation. For example, if a matrix is small it may fit in the cache of the processor and an operation that requires repetitive accesses to that matrix can benefit from this. As a second example, the insertion of a new element in the data structure might be largely influenced by the size of the matrix depending on the storage method. In a more general sense the size of the matrix exposes issues related to storage, access and manipulation of the sparse matrix. The ten matrices which were chosen to represent the variety of matrix sizes are depicted in Table 4.3 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 metric is a value that gives an indication of the distribution of the non-zero elements in the matrix. A low value indicates that the elements are scattered uniformly around the matrix whereas a high value indicates that the elements are clustered. The locality metric is calculated as follows: First, each matrix is divided into blocks of 32×32. For each *non-empty block* the number of non-zeros within this block is divided by 32 to express the value in terms of the dimension of the block. The average over all non-empty blocks is the *locality* metric. For example, a locality value of 1.3 implies that the non-empty 32×32-blocks of that particular matrix contain on average $1.3 \times 32 = 41.6$ non-zero elements. The list of the ten matrices that were from the 132 matrices sorted by the locality metric is depicted in Table 4.4. The displayed locality value is multiplied with 100. 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

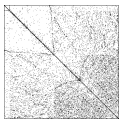
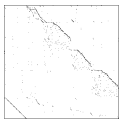
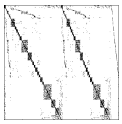
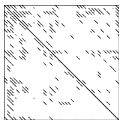
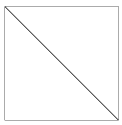
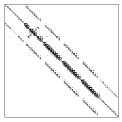
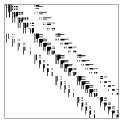
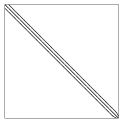
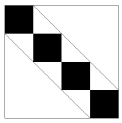
Sparsity Pattern	Name Dimension Non-zeros	Statistical Properties	Short Description
	bcspwr10 5300x5300 21842	Locality: 7 Avrg nz per Row: 4 Largest Row: 14	BCSPWR10: Power network patterns Western US power network – 5300 bus from set BCSPWR, from the Harwell-Boeing Collection
	bp__1600 822x822 4841	Locality: 26 Avrg nz per Row: 5 Largest Row: 304	BP 1600: Original Harwell sparse matrix test collection Simplex method basis matrix from set SMTAPE, from the Harwell-Boeing Collection
	shl__400 663x663 1712	Locality: 28 Avrg nz per Row: 2 Largest Row: 426	SHL 400: Original Harwell sparse matrix test collection Simplex method basis matrix from set SMTAPE, from the Harwell-Boeing Collection
	gemat12 4929x4929 33111	Locality: 37 Avrg nz per Row: 6 Largest Row: 44	GEMAT12: Optimal power flow problems Power flow in 2400 bus system in western US – basis after 100 iterations from set GEMAT, from the Harwell-Boeing Collection
	fs_760_3 760x760 5976	Locality: 69 Avrg nz per Row: 7 Largest Row: 21	FS 760 3: Chemical kinetics problems STRAT stratospheric ionization study – 3rd output time step from set FACSIMILE, from the Harwell-Boeing Collection
	tub1000 1000x1000 3996	Locality: 132 Avrg nz per Row: 3 Largest Row: 4	TUB1000: Tubular Reactor Model from set TUBULAR, from the NEP Collection
	utm300 300x300 3155	Locality: 219 Avrg nz per Row: 10 Largest Row: 33	UTM300: Tokamak Matrices from set TOKAMAK, from the SPARSKIT Collection
	cavity01 317x317 7327	Locality: 394 Avrg nz per Row: 23 Largest Row: 62	CAVITY01: Driven Cavity Problems from set DRIVCAV OLD, from the SPARSKIT Collection
	s1rmq4m1 5489x5489 281111	Locality: 745 Avrg nz per Row: 51 Largest Row: 54	S1RMQ4M1: Finite element analysis of cylindrical shells Cylindrical shell, uniform 30x30 quadrilateral mesh, stabilized MITC4 elements, R/t=10 from set CYLSHELL, from the Independent Sets and Generators generated by MVMTLS
	qc324 324x324 26730	Locality: 1285 Avrg nz per Row: 82 Largest Row: 83	QC324: Model of H2+ in an Electromagnetic Field from set H2PLUS, from the NEP Collection

Table 4.4: 10 Matrices sorted according to the Locality metric

large dense blocks. The average value for all matrices is 2.18. The locality metric is an indication of how spread the elements are in the matrix.

- The *average Non-Zeros Per Row (NZPR)* metric gives the average number of non-zero elements per row of the particular matrix. For symmetry reasons this metric has similar results as would the same metric for the columns of the matrix. Naturally not all matrices are symmetric, however, the irregularities tend to be equally distributed in both dimensions. The resulting list of the ten chosen matrices from the list of 132 that were sorted using this metric are depicted in Table 4.5. The metric varies from 1 (all rows have one element) for matrix *bcsstm20*, a matrix with only a diagonal and 172 for matrix *psmigr_1*. The average for all matrices is 15.9. This metric can influence the performance of many operations such as the sparse matrix vector multiplication. It is mainly important because it indicates the efficiency of the most widely used sparse matrix storage scheme, the Compressed Row Storage (CRS), and its associated algorithms for performing various operations on sparse matrices. One of the main disadvantages of this scheme, especially on vector processors and more general on data parallel processors, is that it performs poorly when the NZRP value is low.

Note that the use of the 32 for the size of the block for calculating the locality metric is arbitrary. We have found that any other value yields similar results in terms of locality and ordering of the matrices. The reason for this is that the sparsity patterns tend to be self-similar in different scales.

The Tables 4.3, 4.4 and 4.5 depict the Matrices that make up the matrix suit for the D-SAB benchmark. In the first column, for each matrix the sparsity pattern is depicted, that is, the distribution of the non-zero elements within the sparse matrix. In the second column the name, dimension and the umber of non-zero's for each matrix is depicted. In column four are depicted the locality (multiplied by 100), the average number of non-zeros for each row and finally the largest row that appears in the matrix. The largest row (LR) is included because it is a good indication for the regularity of a sparse matrix especially when this value is significantly higher than the average non zeros per row ($NZPR$). Moreover, the performance of certain algorithms, like the sparse matrix vector multiplication using the Jagged Diagonal storage, can be influenced by a high $\frac{LR}{NZPR}$ ratio.

For each metric we refer to the 3 first matrices as *small*, the following 4 matrices as *middle* and the last as *large*. The resulting sets are depicted in Tables 3, 4 and 5 along with detailed information about each matrix.

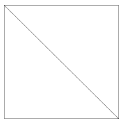
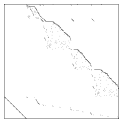
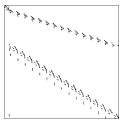
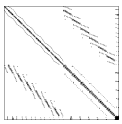
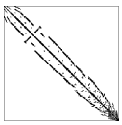
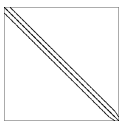
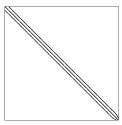
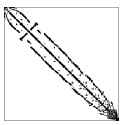
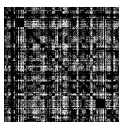
Sparsity Pattern	Name Dimension Non-zeros	Statistical Properties	Short Description
	bcsstm20 485x485 485	Locality: 94 Avrg nz per Row: 1 Largest Row: 1	BCSSTM20: BCS Structural Engineering Matrices (eigenvalue problems) Frame within a suspension bridge from set BCSSTRUC3, from the Harwell-Boeing Collection
	shl___0 663x663 1687	Locality: 42 Avrg nz per Row: 2 Largest Row: 422	SHL 0: Original Harwell sparse matrix test collection Simplex method basis matrix from set SMTAPE, from the Harwell-Boeing Collection
	west2021 2021x2021 7353	Locality: 69 Avrg nz per Row: 3 Largest Row: 12	WEST2021: Chemical engineering plant models Fifteen stage column section, all rigorous from set CHEMWEST, from the Harwell-Boeing Collection
	sstmodel 3345x3345 22749	Locality: 110 Avrg nz per Row: 6 Largest Row: 18	SSTMODEL: BCS Structural Engineering Matrices (linear equations) Elemental connectivity for the stiffness matrix of a 1960's design for a supersonic transport (Boeing 2707) from set BCSSTRUC6, from the Harwell-Boeing Collection
	hor__131 434x434 4710	Locality: 156 Avrg nz per Row: 10 Largest Row: 41	HOR 131: Flow network problem from set NNCENG, from the Harwell-Boeing Collection
	fidap037 3565x3565 67591	Locality: 162 Avrg nz per Row: 18 Largest Row: 85	FIDAP037: Matrices generated by the FIDAP Package from set FIDAP, from the SPARSKIT Collection
	e20r5000 4241x4241 131556	Locality: 392 Avrg nz per Row: 31 Largest Row: 62	E20R5000: Driven cavity driven cavity, 20x20 elements, Re=5000 from set DRIVCAV, from the SPARSKIT Collection
	s1rmq4m1 5489x5489 281111	Locality: 745 Avrg nz per Row: 51 Largest Row: 54	S1RMQ4M1: Finite element analysis of cylindrical shells Cylindrical shell, uniform 30x30 quadrilateral mesh, stabilized MITC4 elements, R/t=10 from set CYLSHELL, from the Independent Sets and Generators generated by MVMTLS
	fidapm37 9152x9152 765944	Locality: 413 Avrg nz per Row: 83 Largest Row: 255	FIDAPM37: Matrices generated by the FIDAP Package from set FIDAP, from the SPARSKIT Collection
	psmigr_1 3140x3140 543162	Locality: 174 Avrg nz per Row: 172 Largest Row: 2294	PSMIGR 1: Inter-county migration US inter-county migration 1965-1970. from set PSMIGR, from the Harwell-Boeing Collection

Table 4.5: 10 Matrices sorted according to the average number of non-zeros per row

4.6 Conclusions

In this chapter we introduced the Delft Sparse Architecture Benchmark (D-SAB) suite, a benchmark suite comprising of a set of operations and a set of sparse matrices for the evaluation of novel architectures and techniques. By keeping the operations simple D-SAB does not depend on the existence of a compiler meant to map the benchmark code on the benchmarked system. Although keeping the code simple D-SAB maintains coverage and exposes the main difficulties that arise during sparse matrix processing. Moreover, the pseudo-code definition of the operations allows for a higher flexibility for the way the operation is implemented. Unlike most other sparse benchmarks, D-SAB makes use of matrices from actual applications rather than utilizing synthetic matrices. To compile the set of operations we have divided the operations in Value Related Operations (VROs) and Position Related Operations (PROs), relating to weather the operation is mainly computational (VROs) or mainly concerns the manipulation of the matrix structure (PROs). The set of matrices chosen for D-SAB are real matrices (as opposed to produced by automatic sparse matrix generators) collected from a wide range of applications. The matrix suite comprises of 3 sets of matrices and have been chosen to reflect the large variety of matrix types and sizes that might be encountered in this application field.

Chapter 5

Experimental Results

In this chapter we present a number of experimental results in order to show the performance benefit that can be expected from using the HiSM and BBCS formats in conjunction with a vector processor augmented with the architectural extensions described in Chapter 3. The experimental results have been obtained through simulation on a Vector Processor Simulator (VPS) that supports the HiSM and BBCS related architectural extensions.

This chapter is organized as follows. In Section 5.1 we describe the simulation environment used to obtain the results. Subsequently in Section 5.2 we provide the results obtained for the Sparse matrix vector multiplication, the most widely used operation in the field of sparse matrix operations. Furthermore, in Section 5.3 we provide results for the operation of inserting an element in to a sparse matrix and in Section 5.4 we provide experimental results from the sparse matrix transposition. Finally in Section 5.5 we draw a number of conclusions.

5.1 The Simulation Environment

In order to be able to evaluate the performance benefits of the ideas proposed in Chapters 2 and 3 we have developed a simulation environment that enabled us to conduct a number of experiments and obtain quantitative results. This section describes the simulation environment which consists of a Vector Processor Simulator (VPS), described in the following section, and the input matrices, a sparse matrix suite, discussed in detail in Chapter 4 and briefly described here in Section 5.1.2.

5.1.1 The Vector Processor Simulator

We have developed a Vector Processor Simulator (VPS) implementing an architecture that is similar to a "traditional" vector processor like the DLXV as described in [33] or the IBM System/370 Vector Architecture [12] in that it supports "basic" vector processor functionality, that is, vector addition, vector multiplication, vector memory access (regular and indexed), etc.

The simulator is based on the SimpleScalar 2.0 simulator, part of the SimpleScalar Tool-Set, developed at the University of Wisconsin-Madison [13]. SimpleScalar is a cycle accurate processor simulator that simulates a modern superscalar processor with an architecture closely related to the MIPS architecture [40]. To support vector processor functionality several additions have been made to the main core of the SimpleScalar simulator. The overall organization of the VPS indicating the augmentations made to the SimpleScalar is graphically depicted in Figure 5.1. The main additions that have been made

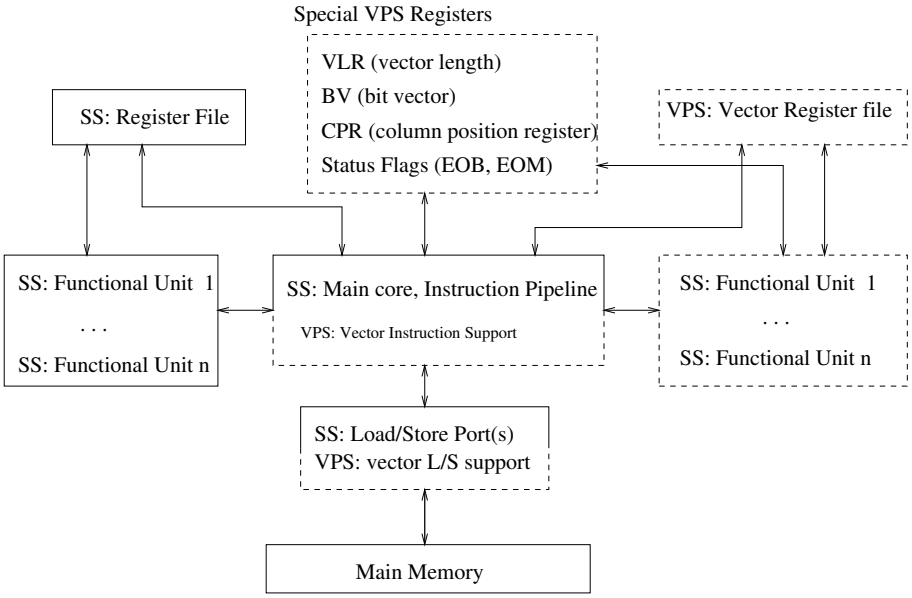


Figure 5.1: The Vector Processor Simulator

can be summarized as follows:

- The ability to fetch and execute vector instructions. To this end we have extended the SimpleScalar Instruction Set to include vector instructions

as described in Chapter 3. The extended instruction set includes "regular" vector instructions such as LV (Load vector), ADDV (Add two vectors). Additionally, the extended instruction set includes our proposed instructions to support the processing of our proposed formats such as LDS and MIPA. For a full description of the HiSM and BBCS support ISA extension please refer to Chapter 3.

- A vector register file consisting of 32 vector registers with variable section size (vector register size). That is to say, the section size can be set as a parameter at the start of the simulation but does not change after that.

- Implementation of Vector Functional Units (FUs). These include both the "regular" vector FUs for instructions such as LV and ADDV, as well as the FUs that we proposed in Chapter 3 for the support of the sparse matrix operations. Note that the vector FUs are completely separated from the scalar FUs. Therefore a scalar addition and a vector addition can execute concurrently. However, a number of scalar instructions related to the vector extension such as setting the vector length register SSVL (Subtract and Set Vector Length) are using the scalar FU.

- Load/Store Unit(LSU) implementation. To support vector memory accesses we have implemented the LSU which can handle LV (Load Vector), SV (Store Vector), LVI (Load Vector Indexed) and SVI (Store Vector Indexed). Furthermore the LSU handles the instructions related to loading and storing data in the BBCS and HiSM formats, such as LDS (LoaD Section), LDB (LoaD Block) and STB (Store Block). The details of the memory access model will be discussed below.

- In addition to the Vector Register File a number of special registers related to the execution of the vector instructions are implemented. These include the bit vector (BV) register and a vector length register (VLR), and the Column Position Register (CPR), a BBCS scheme specific vector register that is required for the correct functionality of the LDS and MIPA instructions. The CPR holds the column positions of the matrix elements loaded by the LDS instruction.

- Two status flags EOB and EOM that are set/unset by the LDS instruction and indicate whether the end of a block or the matrix respectively is reached. These influence the execution of control flow instructions BNEOB (Branch on Not End Of Block) and BNEOM (Branch on Not End Of Matrix).

- Vector accesses are not realized through the cache but instead are 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 implemented a Memory Unit (MU) for the support of the Vector Loads/Stores (LSU) which models a high bandwidth memory that can support the access of up to 4 32-bit words per cycle. For indexed accesses the memory can only provide one 32-bit word per cycle. The response to a vector load is also dependent on the bank memory modules. Each memory bank can deliver a word per cycle after a latency time of 20 cycles. 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. This memory behavior has been modeled similarly to the Torrent-0 vector architecture [3]. In our simulations we have used the number of banks to be 100.

The SimpleScalar provides the possibility to compile C and Fortran programs to execute in the simulator. However, all the code for the experiments was written by hand in vector assembly. A vectorizing compiler was not accessible to us and additionally an existing compiler would not offer support for the added, HiSM and BBCS specific vector instructions and creating or augmenting a vectorizing compiler was out of the scope of this work. Besides, we envision the HiSM and BBCS specific instructions to be used in code which is available through a library of functions to be used in conjunction with the formats rather than be automatically extracted from a high level language by a vectorizing compiler.

5.1.2 Input Matrices: The Sparse Matrix Suite

The matrices we use for our evaluation are the matrices described in Chapter 4. However, we will give here a brief description of the matrix suite used The *D-SAB* benchmark matrices were chosen from a wide variety of matrices that are available from the Matrix Market Collection [11]. 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 criteria resulted in three sets. From each of these sets ten matrices have been chosen with the equal steps (in logarithmic scale[1]) between their corresponding parameters. The result is a manageable but very diverse set of 30 benchmark matrices to be used in the simulations.

5.2 Performance Evaluation of Sparse Matrix Vector Multiplication

In this section we evaluate the benefits of utilizing the HiSM and BBCS storage methods on vector processors augmented to support HiSM and BBCS as described in Chapter 3. The evaluation will be split into two main parts. All evaluations are done by comparing with the most used methods for storing general sparse matrices, the CRS and JD which were described in Chapter 2. 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.

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

First we provide a brief description of the CRS, JD and BBCS formats. 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 row 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 array 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 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.

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. We will refer to the algorithms by the name of the storage method used.

5.2.1 The Sparse Matrix Vector Multiplication 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: To perform the SMVM using the CRS storage for the multiplicand matrix we proceed as follows:

for each row do
Load vector containing the non-zeros of a row → VV (Value Vector)
Load vector containing column positions → CPV (column positions)
Use CPV as index to load multiplicand vector → MV (multiplicand vector)
perform inner product of VV×MV
Store result

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.

JD SMVM. The algorithm to perform SMVM using **JD** storage format looks as follows:

for i=1 to k do
Load value vector → VV
Load corresponding column position vector → CPV
Use CPV as index to load the multiplicand vector → MV
Perform element-wise multiplication VV·MV → IR_i (ith Intermediate Result)
Load previous IR_{i-1} intermediate results vector
Perform vector addition $\text{IR}_i = \text{IR}_i + \text{IR}_{i-1}$
Store vector IR_i

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 from 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$_{col_pos}$ as is depicted in Figure 5.2. 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.

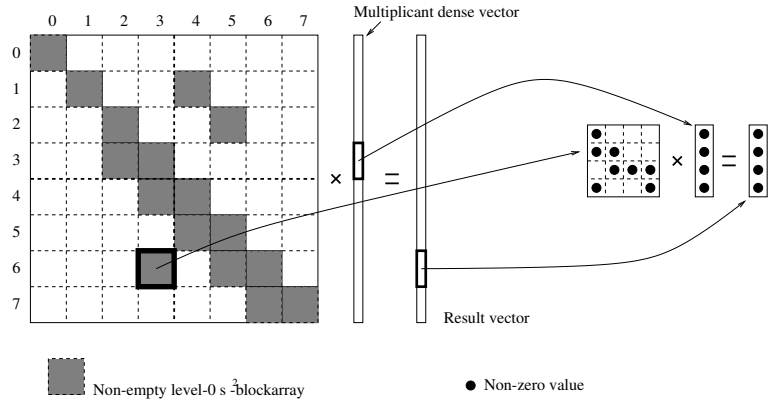


Figure 5.2: The BMIPA vector instruction multiplies an s^2-block with the corresponding multiplicand vector segment producing an intermediate result segment

The reason for this is that the multiplicand vector and result segments that are loaded for the BMIPA operation are treated as dense vectors. This is possible because in the test matrices that we have examined the non-zeros are not randomly scattered over the matrix but are localized and we skip all the empty s^2-blocks of the matrix. Inevitably some values will be loaded that will not be used by the BMIPA operation but does not balance the advantage of not performing indexed accesses.

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. This forms part of a case study and is done in order to investigate any further improvements that we can perform on the HiSM. As future research we think that it is interesting to investigate the $s \times s$-memory unit used for the transposition to be also used as the aforementioned vector cache. We will not go here further into the implementation details of this unit.

BBCS SMVM. The vector code that will serve as our benchmark for doing SMVM using the BBCS scheme to perform the multiplication $\vec{c} = \vec{A} \times \vec{b}$ is the following (Note that the capitals indicate vector instructions):

```
 add r12, r0, COLUMNSS       set the size of the b vector
NB:
 SSVL   r12                  Set the vector length to MAX(r12,section_size)
                             and substract from r12
 LV     vr19, 0(r10)         Load b in r19 (r10 points to the b vector)
 add    r10, r10, SEC_SIZE   Point to next chunck of data from b
                             corresponding to the next vertical block
NS:
 LDS    r9, vr15, vr16       Non-zero elements -> vr15, index -> vr16
                             (r9 contains the start address)
 LVI    vr18, r11, vr16      Load c in r18 using vector index r16
                             (r11 contains the start address of c)
 MIPA   vr18, vr15, vr19     Compute partial c = c + A*b for
                             the part of the vertical block loaded by LDS
 SVI    vr18, r11, vr16      Store c back to memory with the same index
 BNEOB  NS                   if not end of block do next section of A
                             within the vertical block
 BNEOM  NB                   if not end matrix do next vertical block
                             else multiplication finished
```

Notes: `SSVL` stands for "Set and Subtract Vector Length", `LV` stands for Load Vector, and `LVI, SVI` stand for Load/Store Vector with Index, using the second vector register as an index. The inner loop of the code performs the multiplication of a vertical block with the corresponding part of the $\vec{b}$ vector. The outer loop repeats the inner loop for all the vertical blocks. Note that the `LDS` instruction also updates the pointer (here `r9` to the matrix elements which are stored in BBCS format in memory. The main advantage of the BBCS method is a lower memory bandwidth overhead compared with CRS and JD as well as a reduced number of indexed memory accesses. However, it still suffers from the need to use indexed accesses.

5.2.2 Sparse Matrix Vector Multiplication Simulation Results

In this section we present the performance results from executing the various previously described algorithms on the vector processor simulator. All simulations have been run with a section size of 64, a typical vector processor section size. The Figures 5.3, 5.4 and 5.5 depict the performance results for each set of matrices obtained by the sorting criteria that were described in Subsection 4.5. 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 5.1 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 5.3 the matrices have been sorted according to the number

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 5.1: SMVM performance comparison averages for all matrices in the matrix suite and separately for small, medium and large sized matrices

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 5.1. The numbers depicted shown here are derived from Figure 5.3 grouped by size.

In Figure 5.4 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 performance. 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 5.5 the performance of SMVM is depicted for matrices with increasing locality, as discussed in the previous subsection. Here too we

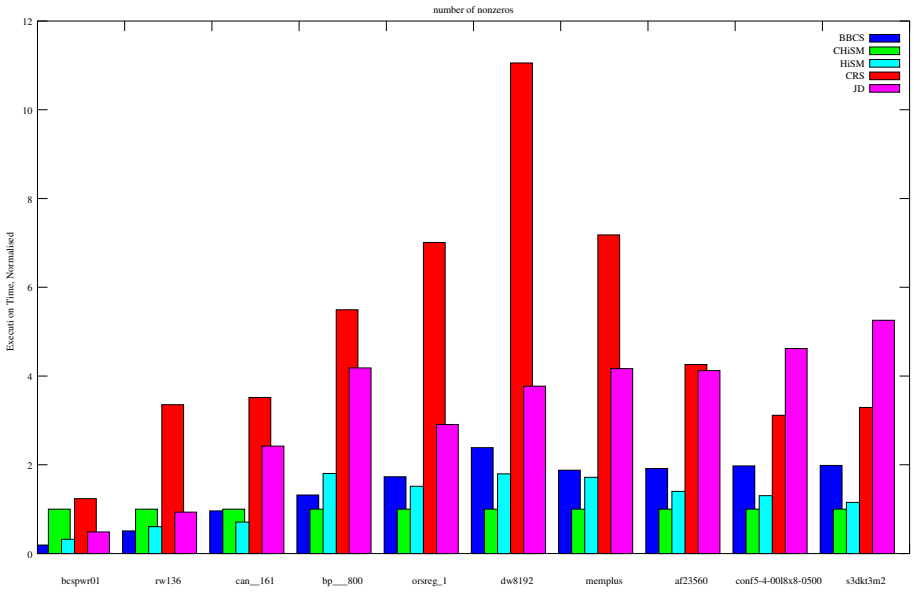


Figure 5.3: SMVM Performance results for increasing matrix size

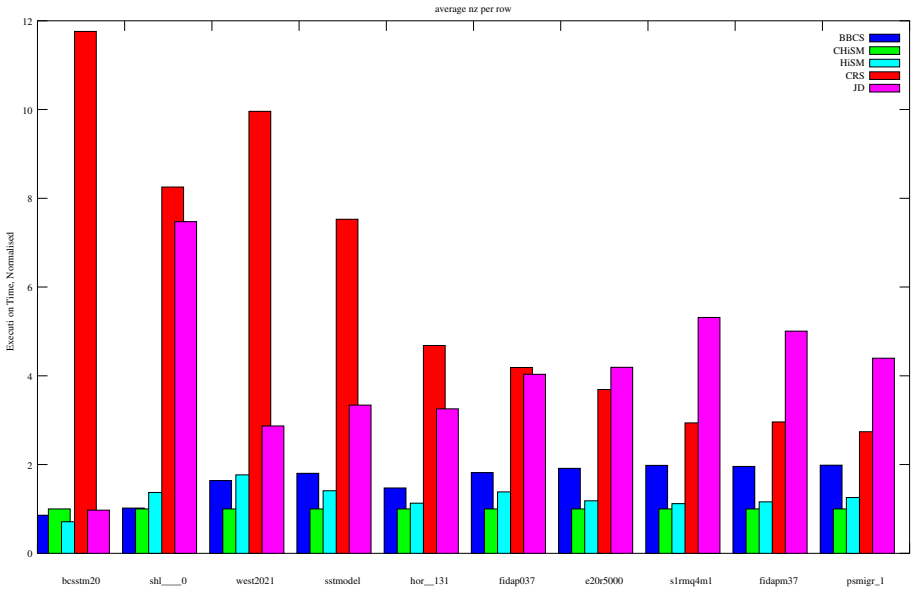


Figure 5.4: SMVM Performance results for increasing average non-zero elements per row

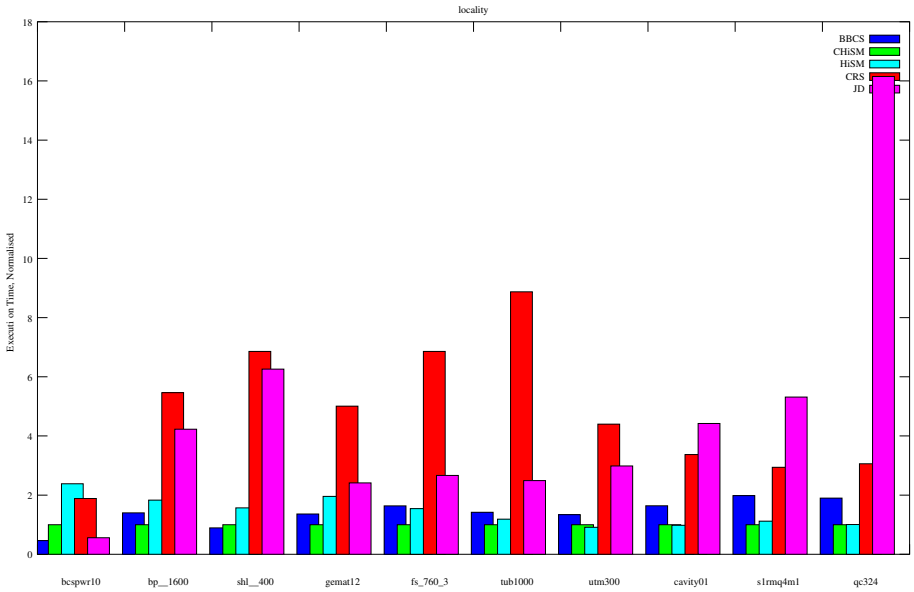


Figure 5.5: SMVM Performance results for increasing locality

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 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 are rare exceptions and therefore we can accept this disadvantage. Another point of interest in Figure 5.5 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 an 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.

5.3 Element Insertion Evaluation

In this section we illustrate the flexibility of the HiSM format to facilitate position related operations. We illustrate this by calculating the number of steps

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 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 4.5, 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 5.2, we give the number of steps (comparisons + displacements) needed to insert an element in the matrix for a number of 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.

5.4 Matrix Transposition Evaluation

To evaluate the performance of the proposed approach we have compared the performance of the proposed mechanism against performing the transposition

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 5.2: Hierarchical Sparse Matrix Storage Format speedup compared to BBCS, JD and CRS for Section size 64

on a vector processor by using the Compressed Row Storage (CRS), the most widely used sparse matrix compression storage for general types of sparse matrices.

The vector code for both HiSM and CRS have been hand-coded in assembly. The HiSM implementation employs recursion since it has to deal with a hierarchical data structure. The pseudo code for it, with a number of the actual implementation details being omitted, is presented in Figure 5.6. The transpose_block() procedure performs the transposition 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 transposition 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). Subsequently the reverse process is performed 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 [63]). More specifically, the code that performs the transposition of the s^2s block is depicted in Figure 5.7 and corresponds to lines 2-9 in Figure 5.6. 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 3.6) 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 subtracts that value 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

```
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 × s memory
5  od
6  for (all Block Sections (BS) in s × s memory) do
7      Load BS from sxs memory to Vector Register
8      Store BS to memory
9  od
10 if (LVL ≠ 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 × s memory
14   od
15   for (all Lengths Sections (LS) in s × 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
```

Figure 5.6: Transposition for HiSM.

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 3.3.1 via the row-wise 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 hierarchy) 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

```
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
```

Figure 5.7: Transposition for HiSM.

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 5.6. Lines 11-23 are executed 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 typically to about $2 - 5\%$ of the total matrix storage for $s = 64$.

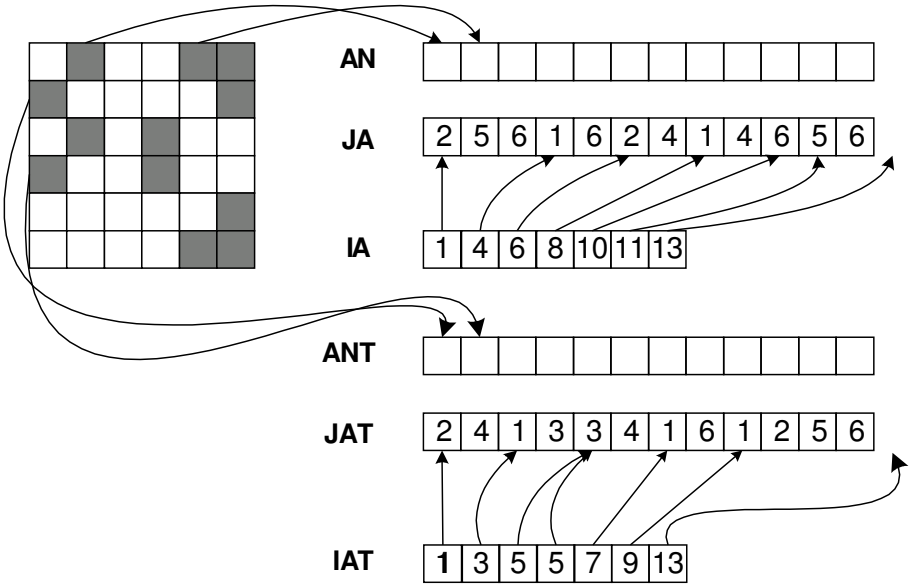


Figure 5.8: Compressed Row Storage (CRS) format

The CRS format is depicted graphically in Figure 5.8 and is briefly described as follows. The non-zero elements of the matrix are all stored in a long continuous 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 [52], a simplified pseudo-code which is depicted in Figure 5.9. Below, we sketch its main parts and how they have been vectorized.

```
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<=i} IAT[j]
4  for (each row i) do
5      iaa = IA(i);  iab = IA(i + 1) - 1;
6      for (jp = iaa;  jp <= 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
```

Figure 5.9: Transposition for CRS.

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 initialized 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 [75]. 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. [75]. 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 indexes 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 5.9.

5.4.1 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 extension 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:

$$BU = \frac{Z/C}{B} \tag{5.1}$$

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.

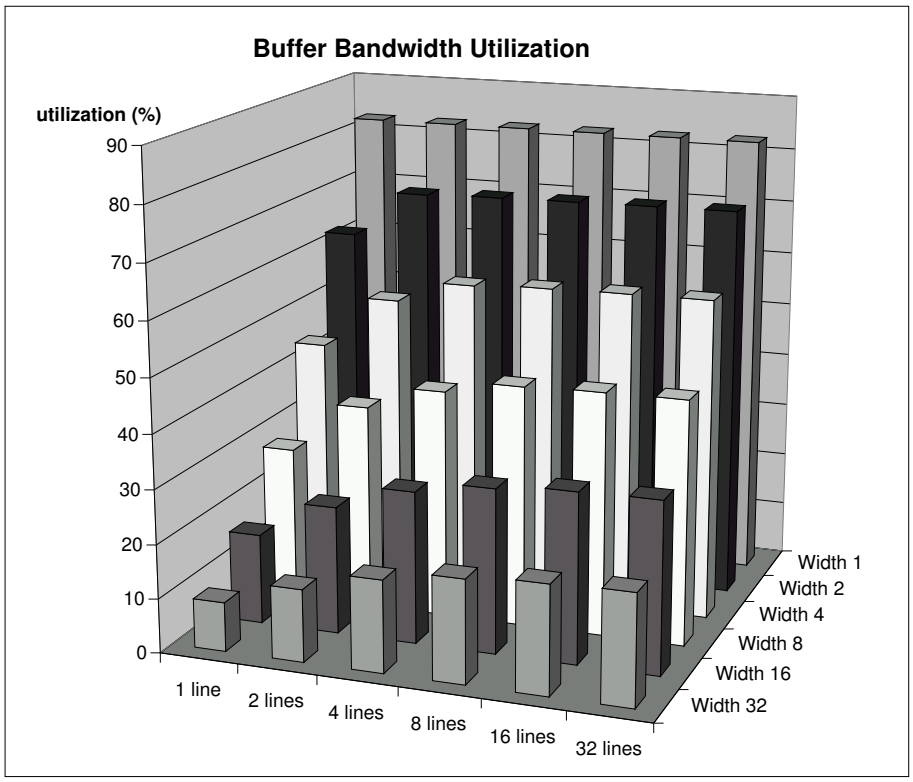


Figure 5.10: Matrix Transposition

The results have been averaged for the total of 30 benchmark matrices and are depicted in Figure 5.10. 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 accessible lines L the utilization increases. However, we remark that for a number of accessible lines $L > 4$ the 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 experiments.

5.4.2 Performance Results

In Figures 5.11, 5.12, and 5.13 we depict how performance of CRS and HiSM schemes depends on the the matrix locality, the number of non-zeroes per row,

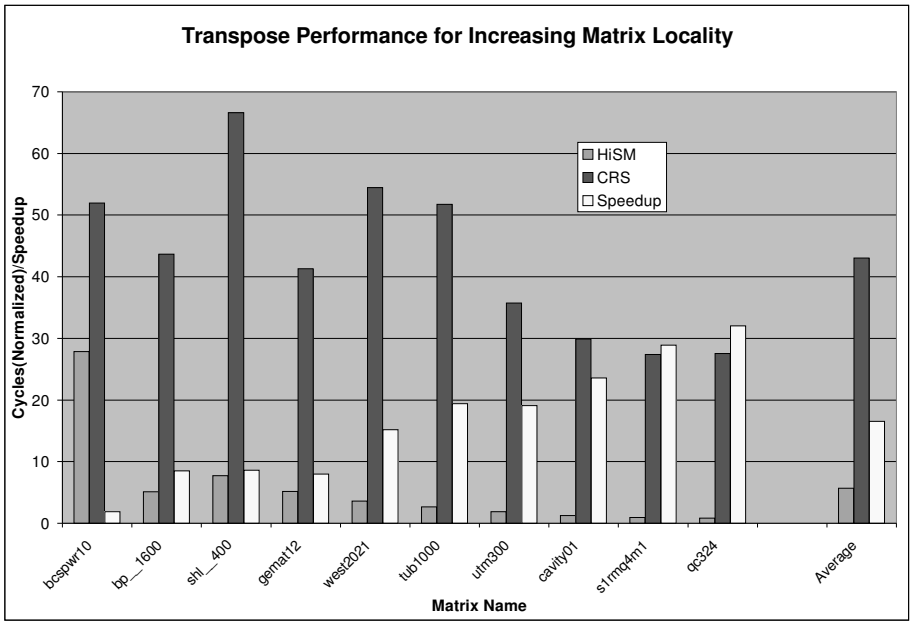


Figure 5.11: Performance w.r.t matrix locality

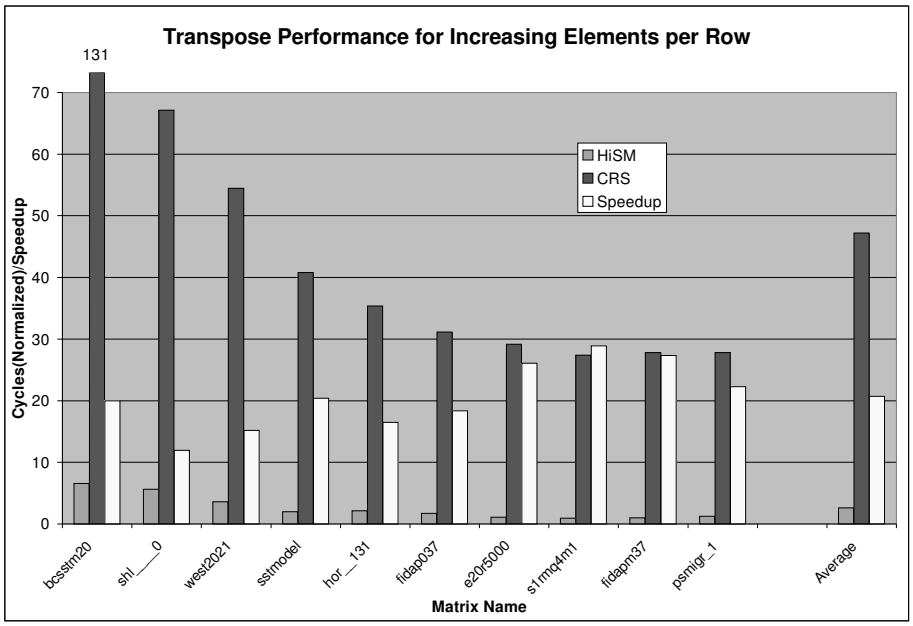


Figure 5.12: Performance w.r.t. average number of non-zeroes per row

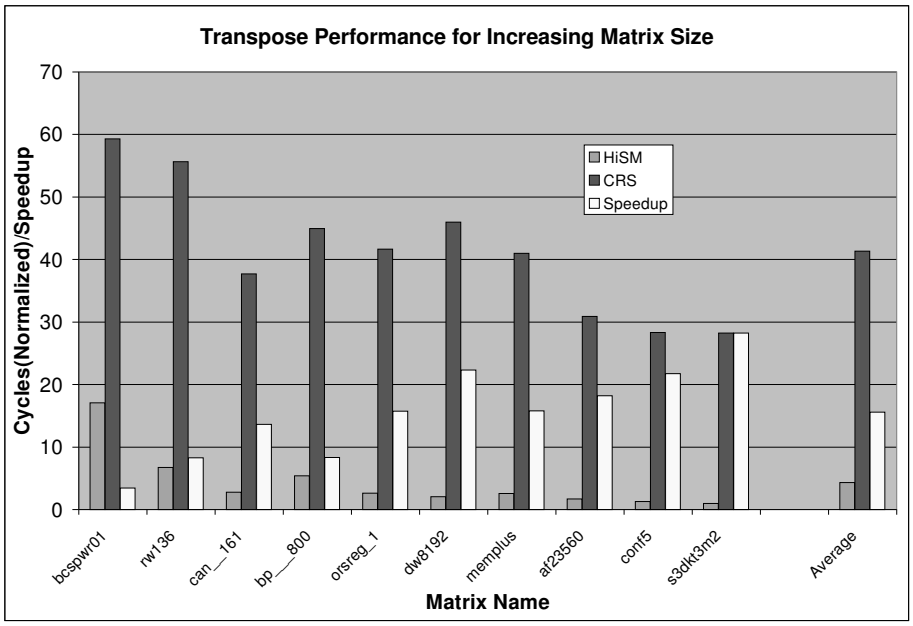


Figure 5.13: Performance w.r.t. matrix size

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 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 has been specifically 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*) increase, the performance of the CRS approach also increases, as shown in Fig-

ure 5.12. This effect is quite expectable. The performance behavior 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 explanation. 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 5.13 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 [62]), 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 5.13 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.

5.5 Conclusions

In this chapter we have discussed a number of experimental results that we have conducted in order to evaluate the potential performance benefit that we can expect using the BBCS or HiSM sparse matrix storage formats when used with a vector processor architecture augmented to support these formats. The results were obtained by simulating a number of benchmarks on real matrices. 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, re-

garding 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. Furthermore, concerning the sparse matrix transpose operation, we have calculated the optimal parameters for the transpose 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 nonzeroes within square blocks, and grows when this density increases.

Chapter 6

Conclusions

We have argued in this dissertation that existing formats for storing and operating on sparse matrices have shortcomings regarding performance and memory requirements. To improve vector processors we have introduced new memory formats, vector instruction set extension and proposed novel organizational techniques that, when implemented, will provide substantial improvements. Additionally, our investigations introduce a new benchmark that allows comparisons to be made on a set of sparse matrices.

In this chapter, we provide some concluding remarks, present the major contributions of our investigation, and present some possible future research directions. This chapter is organized as follows. In Section 6.1 we summarize the main conclusions of this dissertation. In Section 6.2 we list the major contributions described in this dissertation. Finally, in Section 6.3 we highlight some possible future research directions.

6.1 Summary

A brief outline of the achievements is as follows:

In Chapter 2 we have introduced two new sparse matrix storage formats which constitute part of our proposed mechanism for increasing sparse operations performance on vector processors. First we described the existing sparse matrix storage formats with emphasis on Compressed Row Storage (CRS) and Jagged Diagonal (JD), both being general type and widely used formats. We have outlined their advantages and disadvantages. Subsequently, we have described our first proposed format, the Block Based Compression Storage (BBCS) that arose from our study of the sparse matrix vector multiplication and discussed the expected advantages over existing schemes. Fol-

lowing that, we presented our second proposed format, the Hierarchical Sparse Matrix (HiSM) compression format, which offers a number of improvements over BBCS in that it alleviates the use of Index loads altogether and has a more flexible structure. We completed the storage format discussion by providing a quantitative analysis of the existing and proposed formats. More in particular, we have compared the CRS, JD, BBCS and HiSM formats for required storage space and Vector Register Filling (VRF). We have shown that BBCS requires on average only 74% and 78% of the storage space needed for CRS and JD respectively. The corresponding numbers for HiSM are 72% and 76% of the CRS and JD storage space respectively. Furthermore, we have shown that the BBCS format achieves the highest VRF and that the HiSM exhibits similar VRF as the JD format (which was designed for this purpose) despite the partitioning of the matrix in small parts.

In Chapter 3 we have presented an architectural and organizational extension to the general vector processing paradigm in order to provide additional functionality to support the efficient operation on sparse matrices when using the BBCS and HiSM formats which were presented in Chapter 2. As a proof of concept, rather than providing a full architectural description which is beyond the scope of this thesis, we have presented architectural support for a number of key operations. First we have provided a brief description of what we mean by the general vector processing paradigm on which our extension is based. Subsequently, we have described a number of new instructions that access data in memory which are stored in the described formats. Furthermore, we have presented instructions that enable the operation of Sparse Matrix Vector Multiplication both using the BBCS and HiSM schemes and have described the Functional Units which perform these operations in Hardware. Similarly, for HiSM we have provided a mechanism that can support performing the transposition of a sparse matrix.

In Chapter 4 we introduced the Delft Sparse Architecture Benchmark (D-SAB) suite, a benchmark suite comprising of a set of operations and a set of sparse matrices for the evaluation of novel architectures and techniques. By keeping the operations simple D-SAB does not depend on the existence of a compiler meant to map the benchmark code on the benchmarked system. Although keeping the code simple D-SAB maintains coverage and exposes the main difficulties that arise during sparse matrix processing. Moreover, the pseudo-code definition of the operations allows for a higher flexibility for the way the operation is implemented. Unlike most other sparse benchmarks, D-SAB makes use of matrices from actual applications rather than utilizing synthetic matrices. To

compile the set of operations we have divided the operations in Value Related Operations (VROs) and Position Related Operations (PROs), relating to the weather the operation is mainly computational (VROs) or mainly concerns the manipulation of the matrix structure (PROs). The set of matrices chosen for D-SAB are real matrices (as opposed to produced by automatic sparse matrix generators) collected from a wide range of applications. The matrix suite comprises of 3 sets of matrices and have been chosen to reflect the large variety of matrix types and sizes that might be encountered in this application field.

In Chapter 5 we have discussed a number of experimental results that we have conducted in order to evaluate the potential performance benefit that we can expect using the BBCS or HiSM sparse matrix storage formats when used with a vector processor architecture augmented to support these formats. The results were obtained by simulating a number of benchmarks on real matrices. 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. Furthermore, concerning the sparse matrix transpose operation, we have calculated the optimal parameters for the transpose 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 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.

6.2 Main Contributions

In this section we highlight the main contributions of our research that are described in this dissertation.

- We have identified a number of performance efficiency problems that arise during execution of sparse matrix operations on vector processors.

These problems relate mainly to the irregularity of the non-zero element distribution which negatively affects the streamlined processing that vector processors can offer. More in particular we have identified three main reasons of the decreased efficiency of sparse matrix operations on vector processors. (a) The small amount (relative to the section size of the vector processor) of non-zero elements that a sparse matrix typically contains per row induces the formation of small vectors when processing the matrix. This results in the inefficient use of the vector processor's functional units due to vector startup overhead. (b) The arbitrary positioning of the non-zero elements induces the need to use indexed memory access methods (using the positional information as indexes) for accessing data in memory. However, most memory systems, and especially vector memory systems, deliver their maximum throughput when the accessed data is sequential or (in most cases) when given a certain stride (constant distance between successive accesses). Therefore indexed accesses can severely affect the performance when operating on sparse matrices. (c) The need to store and load the positional information about the non-zero elements induces an additional strain on the bandwidth requirements, simply in order to load and store the matrix elements, thereby reducing the efficiency of operations, especially when the operations used are memory bound.

- We have proposed new formats, the Blocked Based Compression Storage (BBCS) and Hierarchical Sparse Matrix (HiSM) Format whose aim is to alleviate the aforementioned issues. The BBCS is a format that was developed having the sparse matrix vector multiplication in mind, an operation that dominates the execution time of many sparse matrix application kernels. BBCS achieves a near optimal vector register filling (and so tackling the problem of short vectors), significantly reduces the number of indexed accesses and has a reduced positional information overhead. The HiSM format, developed with a wider scope of applications, further reduces the amount of indexed accesses at the cost of a slight and negligible reduction of the resulting vector lengths when compared to the BBCS. Additionally, HiSM provides similar reduction of positional information as BBCS (around 25%). Furthermore, the structure of HiSM provides more flexibility compared to existing sparse matrix formats and BBCS with regard to position related operations such as transposition.

- We have proposed organizational extensions and new instructions. for vector processors to support the processing of the aforementioned for-

mats. Our proposal consists of a traditional vector processor augmented with additional functional units and related instructions that provide functionality to process data which are stored in the BBCS and HiSM formats. I.e., the extended vector Load/Store unit has the additional functionality of being able to directly access sections of a matrix stored in memory in BBCS or HiSM to or from the vector registers specified in related new memory access vector instructions. Likewise the new Multiple Inner Product and Accumulate (MIPA) instruction provides functionality for supporting the sparse matrix vector multiplication. This functionality is implemented in the MIPA Functional unit.

- We have proposed a sparse matrix benchmark, the Delft Sparse Architecture Benchmark (D-SAB). D-SAB, consisting of a set of algorithms and matrices, provides a benchmark suite for evaluating the performance of novel architectures and methods. D-SAB offers flexibility in allowing for new methods of storing and processing matrices to be developed and tested. This is in contrast to most currently existing sparse matrix benchmarks which assume a certain storage format and algorithm for processing those matrices. Furthermore, D-SAB provides with a set of real-world matrices selected to capture the large variation of types of matrices that can be encountered in actual applications.

6.3 Future Directions

In this dissertation we have provided a framework for significantly increasing the performance of sparse matrix operations on vector processors. However, there are still a number of interesting issues which could be addressed in the future as continuation of this work.

6.3.1 The Sparse Block Compressed Row Storage

For future research, we propose the investigation of a sparse matrix format which is a hybrid combination of the blocked structure of the Hierarchical Sparse Matrix Storage format (HiSM) and the Compressed Row Storage (CRS) which we call the Sparse Block Compressed Row Storage (SBCRS). The SBCRS format is similar to the Blocked Compressed Row Storage (BCRS, See Section 2.1.3) regarding its relation with CRS. The format is constructed as follows: The Sparse matrix is divided into $s \times s$ sized blocks which are stored in s^2-blockarrays identically as when constructing the HiSM format. However, in contrast to the HiSM format, the higher level of positional

description is not hierarchical but similar to the BCRS. Therefore, the SBCRS structure consists of two parts: (a) A CRS-like structure where the elements are pointers to non-empty s^2-blockarrays representing non-empty $s \times s$ sparse blocks in the matrix and (b) the collection of all s^2-blockarrays. In essence, this is a variation of the BCRS, where the blocks are not assumed to be dense and of arbitrary size, but rather are $s \times s$ sparse blocks. Although we might be loosing some benefits of the positional flexibility of the hierarchical storage, there are a number of advantages we can expect from the SBCRS approach. First, we can reuse the code that is already written for the BCRS regarding the CRS-like part of the SBCRS storage and only use the instruction extensions for the s^2-blockarrays. Secondly, the flat structure of the SBCRS could favor operations where we need to access full rows or columns since we will not need to traverse the hierarchical tree of HiSM. Furthermore, if it proves that in some cases the HiSM outperforms SBCRS and vice versa, the formats could be used interchangeably by only changing the higher level of description of the formats. This is possible since the underlining structure containing the actual non-zero blocks (contained in the s^2-blockarrays) is the same in both cases. Moreover, the higher level description of the formats occupies on average only around $1 - 5\%$ of the total storage space and therefore the conversion overhead can be expected to be low. In Figure 6.1 we depict an example of how a 40×40 sparse matrix can be stored using the SBCRS format where the section size $s = 8$. We observe that each individual $s \times s$ sub-block is stored in the same way it is stored when using the HiSM format. However, to continue the construction of the format we do not repeat the process hierarchically as we did in HiSM. Instead, we store the matrix at level 1 (which contains pointers to the non-zero $s \times s$-blocks in the lowest level) using the CRS method. In Figure 6.1 this corresponds to the 5×5 matrix on the lower right. We can observe that the arrays AI, AJ, and AN are precisely identical to the CRS format for the 5×5 matrix. However, in this case the elements of AN are pointers to the sparse blockarrays (ba1, ba2, ...). Therefore the full SBCRS format consists of array AI, AJ, AN and an array B that contains all the s^2-blockarrays.

To further illustrate the properties of the SBCRS format we will provide an example of sparse matrix vector multiplication (SMVM) using the SBCRS format. Assume that we need to multiply a 40×40 sparse matrix A stored in the SBCRS format with a dense vector b to produce the result vector c as is illustrated in Figure 6.2. In order to do the multiplication we proceed as if the matrix were stored using the CRS format. Therefore the pseudo-code for the SMVM will have the following form:

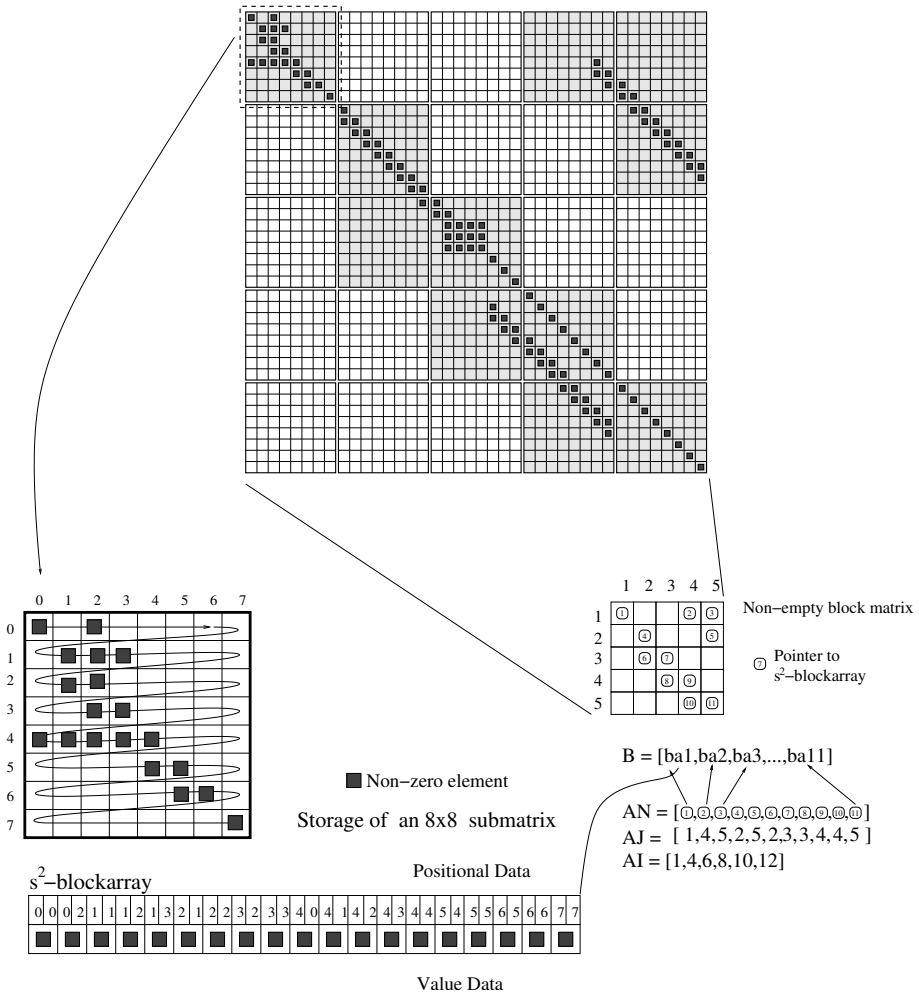


Figure 6.1: Example of the Sparse Block Compressed Row Storage (SBCRS) Format

```
for i=1..M              # M = Number of rows
 for j=AI[i]..AI[i+1]  # all elements in row
  MULT(AN[j],b[(AJ[j]-1)*s..AJ[j]*s],c[(i-1)*s..i*s])
 end for
end for
```

where `s` is the section size (8 in the example in Figure 6.2). In the above code we observe that the only difference with the normal CRS code for SMVM the difference is in the fact that instead of only updating one element of c

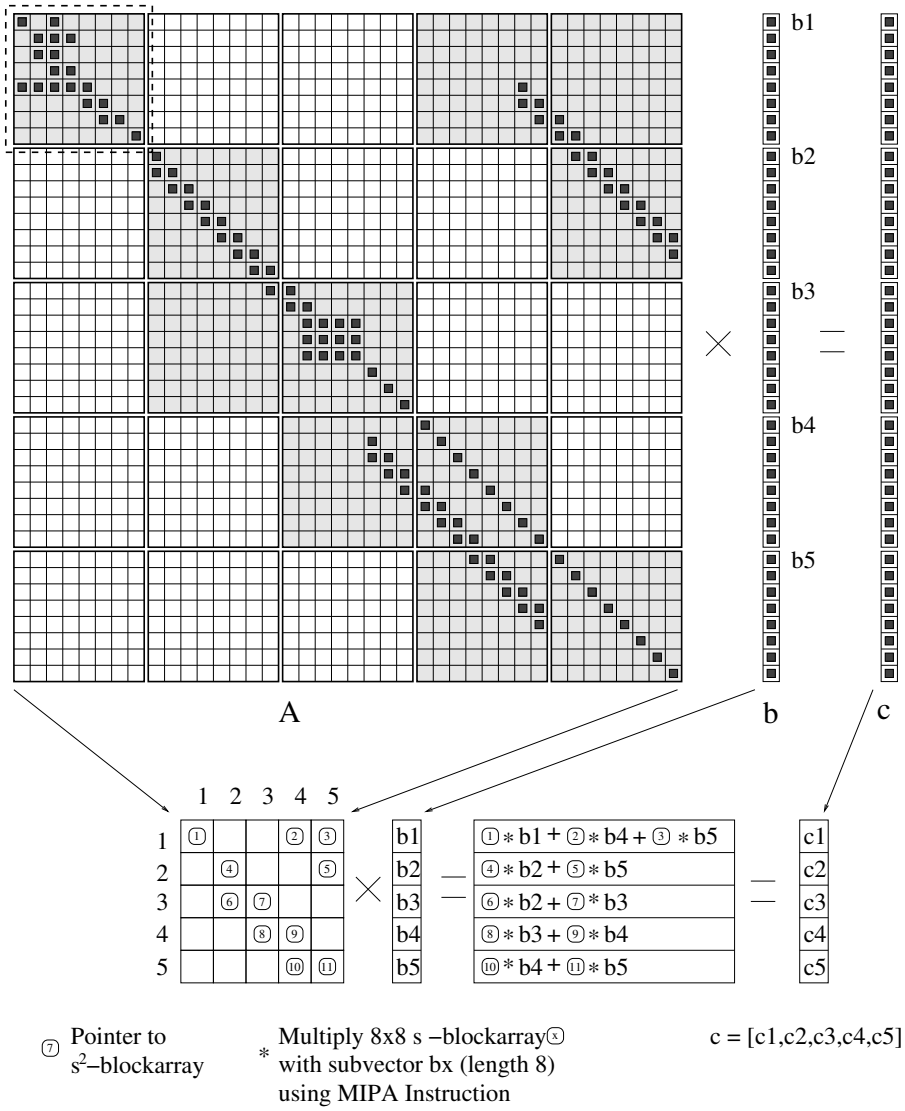


Figure 6.2: Example of the Sparse Block Compressed Row Storage (SBCRS) Format

with the code `c[i] += AN[j]*b[AJ[j]]` we update a sub-vector of c, `c[(i-1)*s..i*s]` of length s with the result of the multiplication of the sparse s^2-blockarray, represented by its pointer AN[j], with the corresponding sub-vector of the multiplicand vector b (length s). The multiplication, MULT(), can be implemented using the LoaD Block (LDB) and Block Multi-

ple Inner Product and Accumulate (BMIPA) instructions similarly to the level-0 multiplication when using the HiSM scheme for SMVM. The pseudo-code for this operation is as follows:

```
MULT(BA,b[(k..k+s],c[l..l+s])
 Load c[l..l+s] => VR1 # load intermediate result
 Load b[k..k+s] => VR2 # load multiplicand vector
 LDB AN[j] => VR3, VR4 # Load blockarray
                       # (VR3 values) (VR4 positions)
 MIPAB VR1, VR3, VR4   # multiply blockarray VR3
                       # (VR4 is implied)
                       # with VR2 and store in VR1
 Store c[l..l+s]       # Store result to memory
```

In Figure 6.2 we depict a graphical representation of the SMVM using the SBCRS format. The * denotes that the multiplication is performed using the BMIPA.
The disadvantages of using the SBCRS format may be the inheritance of the disadvantages of the CRS format which will occur when we are accessing the higher CRS-level of the SBCRS format. However, similarly to the HiSM format, when the section size s is large (around 64), the CRS-level positional description of the format will only occupy a small percentage of the total storage space needed for storing the entire matrix. Therefore, we believe that in the cases where it will be beneficial to use the SBCRS format the benefits will outweigh the disadvantages of using the CRS format.

6.3.2 Other Research Directions

Another promising future research direction can be the application of the ideas presented in this dissertation related to the HiSM scheme to other fields and applications. Such an application could be the sorting of items whose attributes by which they are sorted are unique. The HiSM format and associated extensions can be viewed as a scheme to efficiently process sorted trees. Consider an example where we have a 1-dimensional array of items consisting of the attribute a to be sorted and a pointer p to the other attributes of the item. This can be mapped to the HiSM s^2-blockarray as follows: The attribute s can be mapped to the positional information of the s^2-blockarray entry and the corresponding pointer p to the non-zero value or pointer. We can now process the array using the HiSM supporting architectural extensions which were described in Chapter 3. A field where this approach would be interesting is the manipulation of large database tables which are sorted according to a unique

key.

Furthermore, we expect that the set of instructions and architectural augmentations that were presented in Chapter 3 could be implemented on a reconfigurable extension to a general purpose processor such as is described in [78]. Therefore, we believe it is an interesting perspective to investigate the application of the presented ideas to such a flexible platform. This will enable us to implement the proposed schemes and test them and deal with the real world problems. Such a platform could be the MOLEN processor which is being developed at the Delft University of Technology [45, 69–72, 78]

Bibliography

[1] G. Amdahl. The validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference proceedings, Spring Joint Computing Conference*, volume 30, pages 483–485, 1967.

[2] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, Berkeley, CA94720, May 1998. technical report UCB/CSD-98-1014.

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

[4] K. Asanovic, B. Kingsbury, B. Irissou, J. Beck, and J. Wawrzynek. T0: A single-chip vector microprocessor with reconfigurable pipelines. In H. Grunbacher, editor, *Proceedings 22nd European Solid-State Circuits Conference (ESSCIRC'96)*, pages 344–347. Editions Frontieres, September 1996.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[6] D. H. Bailey, E. Barszcz, L. Dagum, and H. D. Simon. Nas parallel benchmark results. Technical report, NASA Ames Research Center, Mo ett Field, CA, USA, October 1994.

[7] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Philadalphia: Society for Industrial and Applied Mathematics., 1994.

[8] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT club benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[9] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computations on vector multiprocessors. Technical Report Technical Report CMU-CS-93-173, Department of Computer Science, Carnegie Mellon Unversity, August 1993.

[10] R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in java. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, March 1998.

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

[12] W. Buchholz. The IBM System/370 vector architecture. *IBM Systems Journal*, 25(1):51–62, 1986.

[13] 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.

[14] S. Carney. A revised proposal for a sparse blas toolkit, 1994.

[15] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff. Performance of the complex streamed instruction set on image processing kernels. In *Proc. 7th Int. Euro-Par Conference*, pages 678–686, August 2001.

[16] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff. Architectural support for 3d graphics in the complex streamed instruction set. *International Journal of Parallel and Distributed Systems and Networks*, pages 185–193, December 2002.

[17] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff. Performance scalability of multimedia instruction set extensions. In *Proc. Euro-Par 2002 Parallel processing*, pages 849–861, September 2002.

[18] D. Cheresiz, B.H.H. Juurlink, S. Vassiliadis, and H. A. G. Wijshoff. Implementation of a streaming execution unit. *Journal of Systems Architecture*, pages 599–617, vol 49, issues 12–15, December 2003.

[19] Dmitry Cheresiz. *Complex Streamed Media Processor Architecture*. PhD thesis, Universiteit Leiden, Leiden, The Netherlands, March 2003.

[20] Control Data Corporation. *CDC Cyber 200 Model 205 System Hardware Reference Manual*. Arden Hills, MN, 1981.

[21] J. Dongara, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in c++ for high performance architectures, 1994.

[22] J. J. Dongarra and H. A. Van der Vorst. Performance of various computers using standard linear equations software in a Fortran environment. *Supercomputer*, 9(5):17–30, September 1992.

[23] I. Duff, R. Grimes, and J. Lewis. Users guide for the harwell-boing sparse matrix collection. Technical report, October 1992.

[24] Iain S. Duff, R. G. Grimes, and J. G. Lewis. Users' guide for the Harwell-Boeing sparse matrix collection (Release I). Technical Report RAL 92-086, Chilton, Oxon, England, 1992.

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

[26] 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, September 1992. Mon, 26 Apr 99 20:19:27 GMT.

[27] R. Espasa. *Advanced Vector Architectures*. PhD thesis, Universitat Politcnica de Catalunya, 1997.

[28] R. Espasa and M. Valero. Decoupled vector architectures. In *Second Internationl Symposium on High-Performance Computer Architecture*, San Jose, CA, February 1996.

[29] R. Espasa, M. Valero, and J. E. Smith. Out-of-order vector architectures. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, Research Triangle Park, NC, December 1997.

[30] R. Espasa, M. Valero, and J. E. Smith. Simultaneous multithreaded vector architecture: Merging ilp and dlp for high performance. In *International Conference on High Performance Computing (HiPC)*, Bangalore, India, December 1997.

[31] Sam Fuller. Motorola's AltiVec technology. Technical Report ALTIVECWP/D, pub-MOTOROLA, pub-MOTOROLA:adr, 1998.

[32] R. Geus and S. Rollin. Towards a fast parallel sparse matrix-vector multiplication. In E. H. D'Hollander, J. R. Joubert, F. J. Peters, and H. Sips, editors, *Proceedings of the International Conference on Parallel Computing (ParCo)*, pages 308–315. Imperial College Press, 1999.

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

[34] R. G. Hintz and D. P. Tate. Control data star-100 processor design. In *Proc. Compcon 72*, pages 1–4, New York, 1972. IEEE Copmuter Society.

[35] NEC Corporation HNSX Supercomputers Inc. SX-5 series architecture, June 1998.

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

[37] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In volume 2073 of LNCS, editor, *Proceedings of the International Conference on Computational Science*, pages 127–136, San Francisco,CA, May 2001. Springer.

[38] E.-J. Im and K. A. Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *Proceedings of the International Conference on Computational Science, volume 2073 of LNCS*, pages 127–136, San Francisco, CA, May 2001. Springer.

[39] B.H.H. Juurlink, D. Cheresiz, S. Vassiliadis, and H. A. G. Wijshoff. Implementation and evaluation of the complex streamed instruction set. In *Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 73–82, September 2001.

[40] G. Kane. *MIPS RISC Architecture*. Prentice Hall, 1989.

[41] R. E. Kessler and J. L. Schwarzmeier. Cray T3D: a new dimension for Cray Research. In *1993 IEEE Compcon Spring (Feb 22–26 1993: San Francisco, CA, USA)*, pages 176–182, Piscataway, NJ, USA, 1993. IEEE. IEEE catalog number 93CH3251-6.

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

[43] C. Kozyrakis, J. Gebis, D. Martin, S. Williams, I. Mavroidis, S. Pope, D. Jones, D. Patterson, and K. Yelick. Vector iram: A media-oriented vector processor with embedded dram. In *12th Hot Chips Conference*, Palo Alto, CA, August 2000.

[44] U. Kster. Benchmarks: Sparse matrix vector multiplication. 2001.

[45] G.K. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis. The molen processor prototype. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, April 2004.

[46] J. Mellor-Crummey and J. Garvin. Optimizing sparse matrix vector multiply using unroll-and-jam. In *Proceedings of the Los Alamos Computer Science Institute Third Annual Symposium*, Santa Fe, NM, USA, October 2002.

[47] K. Minami and H. Okuda. Performance optimization of geofem on various computer architectures. Technical Report Technical Report GeoFEM 2001-006, Research Organization for Information Science and Technology (RIST), Tokyo, Japan, October 2001.

[48] S. Oberman, F. Weber, N. Juffa, and G. Favor. AMD 3DNow! technology and the K6-2 microprocessor. pages 245–254, 1998.

[49] L. Oliker, J. Carter, J. Shalf, D. Skinner, S. Ethier, R. Biswas, J. Djomehri, , and R. V. der Wijngaart. Evaluation of cache-based superscalar and cacheless vector architectures for scienti c computations. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, 2003.

[50] 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.

[51] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture — improving multimedia and communications application performance by 1.5 to 2 times. *j-IEEE-MICRO*, 16(4):42–50, August 1996.

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

[53] K. Remington and R. Pozo. Nist sparse blas: user's guide, 1996.

[54] Richard M. Russell. The CRAY-1 computer system. pages 63–72, January 1978.

[55] Y. Saad. Kyrlov subspace methods on supercomputers. *SIAM Journal on Scienti c and Statistical Computing*, 10(6):1200–1228, November 1989.

[56] Y. Saad. Numerical solution of large nonsymmetric eigenvalue problems. *Comp. Phys. Comm*, pages 71–90, 1989.

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

[58] Y. Saad. Wijshoff: Spark: A benchmark package for sparse computations, 1990.

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

[60] Y. Saad and Harry Wijshoff. A benchmark package for sparse matrix computations. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 500–509, St. Malo, France, 1988.

[61] Tomas Skalicky. *LASPack Reference Manual*, 1996.

[62] P. Stathis, S. Vassiliadis, and S. D. Cotofana. D-sab: Delft sparse architecture benchmark, http://ce.et.tudelft.nl/˜pyrrhos/d-sab/, 2003.

[63] P.T. Stathis, S. Vassiliadis, and S. D. Cotofana. A hierarchical sparse matrix storage format for vector processors. In *Proceedings of IPDPS 2003*, pages 61–61, April 2003.

[64] Sun Microsystems. UltraSPARC: The Visual Instruction Set (VIS): On chip support for new-media processing. Technical Report WPR-0004, pub-SUN, pub-SUN:adr, 1996.

[65] D. K. Tafti. Genidlest: A scalable parallel computational tool for simulating complex turbulent ows. In *Proceedings of the ASME International Mechanical Engineering Congress and Exposition*, New York, NY, USA, November 2001.

[66] S. Toledo. Improving memory-system performance of sparse matrix-vector multiplication. In *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientic Computing*, March 1997.

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

[68] S. Vassiliadis, S. Cotofana, and Pyrrhos 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.

[69] S. Vassiliadis, G. N. Gaydadjiev, K. Bertels, and E. Moscu Panainte. The molen programming paradigm. In *Proceedings of the Third International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 1–7, July 2003.

[70] S. Vassiliadis, S. Wong, and S. D. Cotofana. The molen $\rho\mu$-coded processor. In *in 11th International Conference on Field-Programmable Logic and Applications (FPL), Springer-Verlag Lecture Notes in Computer Science (LNCS) Vol. 2147*, pages 275–285, August 2001.

[71] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, and K. Bertels. Polymorphic processors: How to expose arbitrary hardware functionality to programmers. In *IEE FPGA Developer's Forum*, October 2003.

[72] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G.K. Kuzmanov, and E. Moscu Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, page to appear, November 2004.

[73] R. Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, University of California, Berkeley, December 2003.

[74] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of Supercomputing*, Baltimore, MD, USA, November 2002.

[75] 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*, 7:769–782, 1996.

[76] W. Watson. The ti-asc, a highly modular and flexible computer architecture. In *Proc AFIPS*, pages 221–228, 1972.

[77] J. B. White and P. Sadayappan. On improving the performance of sparse matrixvector multiplication. In *Proceedings of the International Conference on High- Performance C omputing*, 1997.

[78] S. Wong. *Microcoded Reconfigurable Embedded Processors*. PhD thesis, December 2002.

[79] XILINX. *DataSource CD-ROM*. XILINX, 2000.

List of Publications

1. P.T. Stathis, D. Cheresiz, S. Vassiliadis and B.H.H. Juurlink. Sparse Matrix Transpose Unit. In *18th International Parallel and Distributed Processing Symposium (IPDPS2004)*, (Santa Fe, NM, USA) April 2004

2. P.T. Stathis, S. Vassiliadis and S. D. Cotofana. D-SAB: A Sparse Matrix Benchmark Suite. In *Proceedings of 7th International Conference on Parallel Computing Technologies (PaCT 2003)*, pages 549–554, (Nizhni Novgorod, Russia), September 2003.

3. P.T. Stathis, S. Vassiliadis and S. D. Cotofana. A Hierarchical Sparse Matrix Storage Format for Vector Processors. In *Proceedings of 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, (Nice, France), April 2003.

4. S. D. Cotofana, P.T. Stathis and S. Vassiliadis. Direct and transposed sparse matrix-vector multiplication. In *Proceedings of the 2002 Euromicro conference on Massively-parallel computing systems, MPCS-2002*, pages 1–9, (Ischia, Italy), April 2002.

5. P.T. Stathis, S. D. Cotofana and S. Vassiliadis. Sparse Matrix Vector Multiplication Evaluation Using the BBCS scheme. In *Proc. of 8th Panhellenic Conference on Informatics*, pages 40-49, (Nicosia, Cyprus), November 2001.

6. S. Vassiliadis, S. D. Cotofana and P.T. Stathis. BBCS based sparse matrix-vector multiplication: initial evaluation. In *Proc. 16th IMACS World Congress on Scientific Computation, Applied Mathematics and Simulation*, pages 1–6, (Lausanne, Switzerland), August 2000.

7. S. Vassiliadis, S. D. Cotofana and P.T. Stathis. Block Based Compression Storage Expected Performance. In *Proc. 14th Int. Conf. on High*

Performance Computing Systems and Applications (HPCS 2000), pages 389-406, (Victoria, Canada), June 2000.

8. S. Vassiliadis, S. D. Cotofana and P.T. Stathis. Vector ISA Extension for Sprase Matrix Multiplication. In *Proceedings of EuroPar'99 Parallel Processing Symposium*, pages 708–715 September 1999.

Samenvatting

In dit proefscrift zijn een aantal tekortkomingen geidentificeerd die gerelateerd zijn aan de efficiente opslag en bewerking van ijle matrices. Om deze problemen te elimineren stellen wij twee opslagformaten voor, namelijk het *Block Based Compression Storage* (BBCS) formaat en het *Hierarchical Sparse Matrix* (HiSM) formaat. Verder, stellen wij vector architectuur instructie set extensies en microarchitectuur mechanismes voor om vaak gebruikte ijle matrix operaties sneller uit te voeren wanneer wij gebruik maken van de voorgestelde formaten. Tot slot hebben we de afwezigheid van benchmarks geidentificeerd die tegelijkertijd formaten en ijle matrix operaties bestrijken. Wij hebben een benchmark voorgesteld die beide bestrijkt. Om ons voorstel te evalueren hebben wij een simulator ontwikkeld gebaseerd op SimpleScalar, en deze uitgebreid zodat het onze voorgestelde veranderingen bevat. Wij hebben het volgende vastgesteld. Met betrekking tot opslag vereisen de voorgestelde formaten 72% tot 78% van de opslag ruimte die gebruikt wordt door de *Compressed Row Storage* (CRS) of *Jagged Diagonal* (JD), beide algemeen gebruikte ijle matrix opslag formaten. Wat betreft ijle matrix vector vermenigvuldiging stellen wij vast dat beide geintroduceerde formaten, nl. BBCS en HiSM, zorgen voor een aanzienlijke versnelling in vergelijking tot CRS en JD. Door het gebruik van HiSM en de voorgestelde nieuwe instructies wordt een gemiddelde versnelling van 5.3 en 4.07 ten opzichte van respectievelijk CRS en JD behaald. Verder kan de operatie voor het toevoegen van een element in een ijle matrix, gebruik makend van HiSM, met een factor van 2-400 versneld worden afhankelijk van de verdeling van de elementen in de matrix. Verder hebben we gedemonstreerd dat de transpositie operatie met een factor van 17.7 versneld kan worden in vergelijking tot CRS.

Curriculum Vitae

Pyrros Theofanis Stathis was born on the 23rd of May 1971 in Athens, Greece. After finishing his secondary education at the "Proto Lykeio Papagou, Athens", he studied at the Department of Electrical Engineering of the Delft University of Technology where he graduated in 1996 on the chaotic behavior of Neural Networks. In 1997 he started his PhD studies at the Computer Engineering group headed by Prof. Stamatis Vassiliadis. His research interests include Computer Architecture, Vector Processors, Neural Networks and Artificial Intelligence.