Memory Hardware Support for Sparse Computations*

Arnold J. Niessen Harry A.G. Wijshoff

High Performance Computing Division,
Department of Computer Science, Leiden University
P.O. Box 9512, 2300 RA Leiden, The Netherlands
+31 71 277037, fax: +31 71 276985
niessen@cs.leidenuniv.nl

*This work was supported in part by the Esprit Agency DG XIII under Grant No. APPARC 6634 BRA.

Contents

List of Tables

List of Figures

Abstract

Address computations and indirect, hence double, memory accesses in sparse matrix application software render sparse computations to be inefficient in general. In this paper we propose memory architectures that support the storage of sparse vectors and matrices. In a first design, called *vector storage*, a matrix is handled as an array of sparse vectors, stored as singly-linked lists. Deletion and insertion of a vector is done row- or column-wise only. In a second design, called *matrix storage*, a higher level of sophistication is achieved. A sparse matrix is stored as a bi-directionally threaded doubly-linked list of elements. This approach enables both row- and column-wise operations. Reading a row (column) can be done at the speed of one element (real value *and* indices) per memory cycle, while extracting or updating takes 2 memory cycles. Inserting an element can be done once every 2.5 memory cycles. A pipelined variant with 3-fold interleaved memory and write buffers yields higher efficiency, close to one sparse matrix element per memory cycle for all basic vector operations. In-memory operations also decrease the burden on processor, cache, and bus.

1 Introduction

Due to the importance of sparse matrix applications, research has been conducted to tackle problems that are inherent to software controlled sparse data storage: fill-in, data movement, need for memory management, indirect addressing, and hard-to-optimize (obscured) code.

In this paper we introduce some special-purpose memory architectures for the storage of sparse matrices. These memories are intended to be embedded in existing processor systems or future general purpose computing environments. We assume that a processor performs the following operations on a matrix: (1) *reading* a row (or column) of a matrix, (2) *extracting* (reading while deleting) a row (or column) of a matrix, and (3) *inserting* (part of) a row (or column), assuming the elements to be inserted are non-entries. In general, elements can not be dealt with individually, although the insertion of single non-entries is easy for most designs. Selective deletion of elements of a row which is being read is also possible, but not described in this paper.

All proposed architectures have some properties in common: they all use unordered storage for the vectors involved (motivated by [9]); the control is micro-coded for simplicity, whereas hard-wired control logic would give equal performance and be more logical; and all architectures are proposed without a cache. An additional proposal for a cache has been made in this paper, but no performance data is available yet about this part. Because this cache is capable of handling a matrix on a element-by-element, the main goal of the hardware architectures as described in this paper is to handle vectors efficiently, i.e., read-, extract-, or insert-operations of a row or column can be done at a fixed rate of 1 element per 1 up to 4 memory cycles.

This paper is organized as follows. The first two architectures (based on *vector storage* design) presented in Section 3 treat a matrix as an array of sparse vectors. Consequently, only row-wise (or only column-wise) operations can be dealt with. The main advantage of *vector storage* design is the relatively low overhead in terms of hardware and memory quantity. The architectures presented in Section 4 are based on *matrix storage* design and implement a matrix using a complicated storage scheme. This requires a more complex hardware design and increases the required amount of memory, but integrates both row-wise and column-wise read, extraction, and insertion.

The behaviour of the memory architecture for an example, sparse matrix vector multiply, is shown in section 5, and a comparison with Some possible ways for integration in an existing computing environment are discussed in section 6. A proposal for a cache, and the example of a saxpy operation, is discussed in Section 7. Finally, we state some conclusions in Section 8.

In Table 1 some of the parameters of the architectures and matrices involved are listed, as used in this paper. In order to explain the storage schemes used in this paper, we will use the following example sparse matrix A:

	meaning	magnitude
$\mathcal{N}$	max. order of sparse matrix A	$\mathcal{N} < 2^{16}$
$R(i)$	number of entries in row i	
$C(i)$	number of entries in column i	
ρ	av. number of entries per row	$\frac{1}{\mathcal{N}} \cdot \sum_{i=1}^{\mathcal{N}} R(i)$
M	number of elements per block	
m	size of memory system	2^{22} (4 M-word)
f	size of register file	4
r	size of real in bits (double precision float)	64
c	size of row/col. index in bits	16
p	size of pointer in bits	22

Table 1: System parameters and their assumed size.

$$A = \begin{pmatrix} 10.0 & 0.0 & 3.0 & 0.0 \\ 0.0 & 20.0 & 8.0 & 5.0 \\ 4.0 & 0.0 & 30.0 & 0.0 \\ 1.0 & 0.0 & 6.0 & 40.0 \end{pmatrix}$$

2 Related work

In the past, a number of special purpose machines has been proposed or built. In [18], a dedicated processor, called 'The White Dwarf', is described by Wolfe *et al.* for accelerating finite element analysis algorithms. Their system contains a wide memory organization, two floating point units and an ALU. This system has special support in memory for efficient link traversal. A special microcode compiler for a modified C subset supports the system.

In [17], Wing proposed a systolic array of processors with associative memory, connected in a ring. To solve a system of sparse linear equations, the columns are distributed over the processors in an interleaved manner.

In [1, 2], Amano *et al.* described a dedicated hierarchically built parallel machine with sophisticated shared memory that enables parallel conflict free reads. Although this machine is built to solve linear equations, it has a much broader range of use.

Others described architectural support for sparse computations. Banerjee *et al.* described a concept in [3] to enhance the memory usage of a SIMD machine for sparse computations. A special block of hardware reorders references to an array, distributed over parallel memories, such that per time step as many array elements as possible can be referenced without conflicts. This is described for both densely and sparsely stored arrays.

PSolve is a concurrent algorithm, described by Davis and Davidson in [8], for solving sparse systems of linear equations on parallel processors. In their paper, a so-called 'vector alignment unit' is proposed to achieve vector performance on a vector processor. This unit aligns two ordered sparse vectors for processing in an ALU, adding zeros to the input vectors if needed.

A similar alignment unit is described by Ibbett *et al.* in [12], together with the hardware support for the storage of ordered sparse vectors as a singly linked list of blocks. Variable-sized blocks are supported, as well as garbage-collection.

Hardware gather/scatter is in use on commercially available machines (e.g., the CRAY-2 and CRAY X-MP). The benefits of gather/scatter for sparse Gaussian elimination are discussed by Lewis and Simon in [14].

A VLSI design for sparse matrix vector multiply is presented in [6, 7] by Codenotti *et al.*. A regular and reprogrammable grid is used to store the sparse matrix. This grid transports the matrix entries to a ring of cells. These cells perform the multiplications and store the vector.

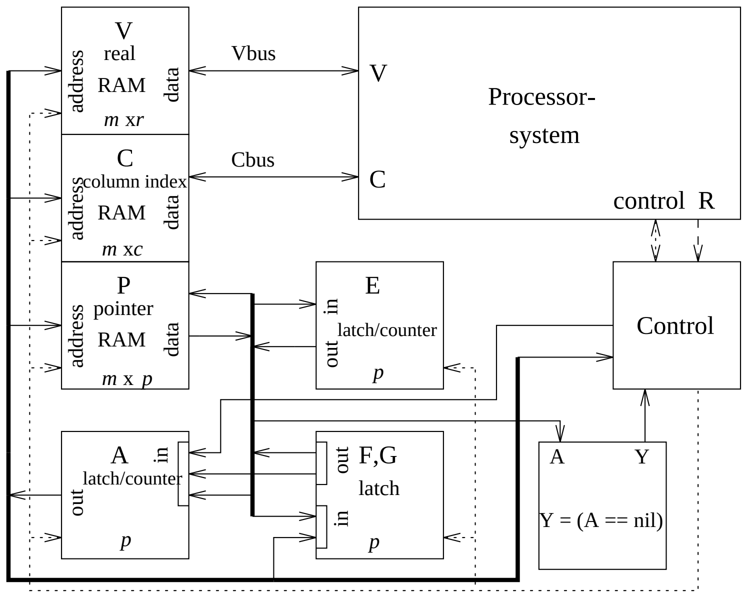

Figure 1: Architecture for element-wise storage of vectors

3 Vector storage designs

This section describes two variations on *vector storage* designs. In both designs matrices are stored as an array of vectors. In the first design, a vector consists of matrix elements in a linked list, in the second design a vector consists of a list of linked blocks of matrix elements.

3.1 Element-wise storage of vectors

Figure 1 depicts a simple hardware scheme to store singly-linked lists of elements. To illustrate the way this system works, Figure 2 shows a possible way how the example matrix A can be stored. The contents of field P at address A is denoted as M[A].P.

The pointer to the first unused memory place is stored in address 0. The pointers to the $\mathcal{N}$ rows are stored in memory locations $1..\mathcal{N}$. For simplicity, we decide not to include a vector element at the pointer's place. This simplifies the use of the system, but increases the memory demand. Simply by changing the microcode in the control unit, the system may operate with this element if desired. The other, empty, places in memory are linked together during initialization.

In Figure 1, dotted lines indicate control lines, straight lines are buses. The memory is partitioned in a number of blocks:

- V: This memory part stores the actual real values of elements in the

A	M[A].V	M[A].C	M[A].P	comment
0	-	-	12	empty list pointer
1	-	-	5	row pointer 1
2	-	-	7	row pointer 2
3	-	-	6	row pointer 3
4	-	-	10	row pointer 4
5	`3.0`	3	15	a_{13}
6	`4.0`	1	13	a_{31}
7	`8.0`	3	11	a_{23}
8	`5.0`	4	**nil**	a_{24}
9	`1.0`	1	16	a_{41}
10	`40.0`	4	9	a_{44}
11	`20.0`	2	8	a_{22}
12	-	-	14	empty
13	`30.0`	3	**nil**	a_{33}
14	-	-	17	empty
15	`10.0`	1	**nil**	a_{11}
16	`6.0`	3	**nil**	a_{43}
17	-	-	18	empty
18	-	-	19	empty
19	-	-	**nil**	empty

Figure 2: Contents of memory in *vector storage* design

matrix (also called *primary storage*).

- C: For every value stored in V, C contains the value of the column index (part of the so called *overhead storage).*
- P: The P field points to a next element. For addresses in use, P points to the next element in the row under consideration. For unused addresses, P points to a next unused address, hence linking all available free memory together. In both cases, P can have a special value **nil** to indicate the end of a list. We choose **nil** equal to 0 for reasons of simplicity.

Other parts in the design are:

Control: Control is the main control unit of the system. It is a simple and flexible micro-programmed unit, using well-known technology [11].

Processor-system: The interface to the host consists of 4 buses: (1) the V-bus for reals; (2) the C-bus for column indices; (3) the sync-bus for synchronization signals; and (4) the command-bus, used for the 'instructions' for the control-unit, containing information such as the number of the row to be handled, the order of the matrix $\mathcal{N}$, and the type of operation needed. This bus can be implemented in several ways: (4a) as an I/O mapped device, with an explicit address where instructions can be written to; (4b) directly connected to the processor hardware; or (4c) implicitly using the standard control signals. See Section 6 for more details.

Multiplexor/Data latch/Tri-state: These blocks consist of a two-stage block (multiplexor and data latch). Under control of *Control*, the data latches can be filled each memory-cycle with data on one of the buses, to be written into memory. The latches are provided with tri-state outputs to enable reads from memory. All multiplexors/latches function independently and simultaneously.

$\underline{A}$: address latch, in which the address of the next memory reference will be clocked. During list traversal, A is filled with the output of P. During initialization or clear-up phase, A can also be clocked with data from E, control or G. $\underline{E}$: E is used as an extra register to store the first address of the empty list. This information is normally stored in memory at address 0, but sometimes a backup is needed during initialization or during writes. E is also used during initialization as a counter. *F,G*: F and G are extra registers which normally hold the value of A during the previous cycle, because this value is needed during the clear-up phase.

We use pseudo-code to clarify the implementation of operations on the architecture. '=' denotes an assignment, and '==' a compare. Every labeled fragment of code is executed in one clock cycle. An atomic assignment is denoted like `[A,B] = [B,A]`, which means that the values of `A` and `B` are swapped.

In the sequel we will describe, in pseudo code, the following basic vector operations:
- *initialization* of the hardware
- *read* vector R
- *extract* vector R
- *insert* (part of) vector R (assuming the inserted elements of vector R were

non-entries)

Initialization

The pointers P of locations 1..$\mathcal{N}$ are initialized to **nil**, to create an empty matrix. The other addresses (0 and $\mathcal{N}$+1..m-1) are linked together starting at address 0 to form the list of free locations. We assume $m > N + 1$.

```
I1: A = 1
L1: while (A <= 𝒩) do [M[A].P, A] = [nil, A+1]
I2: [A, E] = [0, (𝒩+1) mod m]
L2: while (E > 0) do [M[A].P, A, E] = [E, E, (E+1) mod m]
E1: M[A].P = E
```

The choice we made for **nil**=0, leads to a number of optimizations. For example, some statements disappear. Since E counts modulo m, E equals **nil** after executing the last increment.

Vector read

Reading a vector is a very straightforward operation. Communication to, or from the processor-system is denoted using the 'variables' Vbus and Cbus (where R is the number of the requested row).

```
I1: A = R
I2: A = M[A].P
L1: while (A <> nil) do [Vbus, Cbus, A] = [M[A].V, M[A].C, M[A].P]
```

Vector extraction

The extraction of a vector involves, in addition to the vector read in the previous section, the rearrangement of the empty list and the initialization of the row pointer[1].

```
I1: A = 0
I2: [A, E] = [R, M[A].P]
I3: [A, G] = [M[A].P, M[A].P]
    if (A <> nil) then
L1:   while (M[A].P <> nil) do [Vbus, Cbus, A] = [M[A].V, M[A].C, M[A].P]
L2:   [A, M[A].P, Vbus, Cbus] = [0, E, M[A].V, M[A].C]
E1:   [A, M[A].P] = [R, G]
E2:   M[A].P = nil
    endif
```

Vector insertion

Insertion of a vector, or fill-in of one or more elements to an existing vector, is performed by adding the new elements to the list (at the beginning).

[1]Vector extraction can be done more elegant without an if-statement, at the cost of 1 extra clock cycle per vector.

```
I1: A = 0
I2: [A, E, G] = [M[A].P, M[A].P, 0]
L1: while ((insert more) and (A <> nil)) do
       [M[A].V, M[A].C, A, G] = [Vbus, Cbus, M[A].P, A]
    enddo
E1: if (G <> 0) then
       [A, F] = [0, A]
E2:    [A, M[A].P] = [R, F]
E3:    [A, F] = [G, M[A].P]
E4:    [A, M[A].P] = [R, F]
E5:    M[A].P = E
    endif
```

Vectors can be inserted one element per cycle, and the overhead per vector is 7 clock cycles. The insertion of a single element costs 8 cycles using this code. This can be decreased to 7 cycles without pipelining and 6 cycles with pipelining. Using a technique used in *matrix storage* design later in this paper, this could be reduced to 3 cycles for a single element by using different memory parts for the pointers in the empty list and pointers in vector lists.

3.2 Block-wise storage of vectors

A simple way to decrease the amount of memory needed per entry is to store the elements in consecutive chunks of M reals. A vector consists of a singly linked list of blocks of elements. Every block contains an extra bit per element which may be true to indicate the end of a row (EOR).

For clarification, Figure 3 depicts a possible storage pattern of A. Figure 4 shows the architecture for this variant.

So, at the cost of internal fragmentation and one extra bit per element, we can decrease the size of the P memory part by a factor M. We assume that the block size M is a power of two, and that. the first $\mathcal{N}$ blocks (apart from block 0), are used as the pointers to vectors, and store the first $(M-1)$ elements of these vectors. Register A is replaced by a counter and a smaller register.

Among the registers in Figure 4 there is a combination of two registers Ah and Al, which replace the former register A. Al is a modulo M counter which addresses one element within the (by Ah) specified block. Ah remains its value during the counting of Al, except when Al counts from M-1 to 0. If this happens, Ah is loaded with `M[Ah].P`, addressing the first element of the next block within a vector. The effective address of the Ah/Al combination can be obtained by concatenating their bit patterns `Ah##Al`, and is used as the address for the blocks V, C, and EOR (but not P).

Operation

We assume that the first $\mathcal{N}$ blocks (apart from block 0), are used as the pointers to vectors, and store the first $(M-1)$ elements of these vectors. This is not elegant, but one empty element is useful (for EOR detection), and a block of unused elements would be too much waste. The fact that the first block in a

A	M[A].V	M[A].C	M[A].EOR	M[A].P	comment
0	-	-	-		
1	-	-	-	18	empty list pointer
2	-	-	0		row 1 starts here
3	`3.0`	3	0	10	a_{13}
4	-	-	0		row 2 starts here
5	`8.0`	3	0	16	a_{23}
6	-	-	0		row 3 starts here
7	`4.0`	1	0	14	a_{31}
8	-	-	0		row 4 starts here
9	`40.0`	4	0	12	a_{44}
10	`10.0`	1	1		a_{11}
11	-	-	0	**nil**	empty slot
12	`1.0`	1	0		a_{41}
13	`6.0`	3	1	**nil**	a_{43}
14	`30.0`	3	1		a_{33}
15	-	-	0	**nil**	empty slot
16	`5.0`	4	0		a_{24}
17	`20.0`	2	1	**nil**	a_{22}
18	-	-	-		empty block
19	-	-	-	**nil**	

Figure 3: Contents of memory in block-wise *vector storage* design

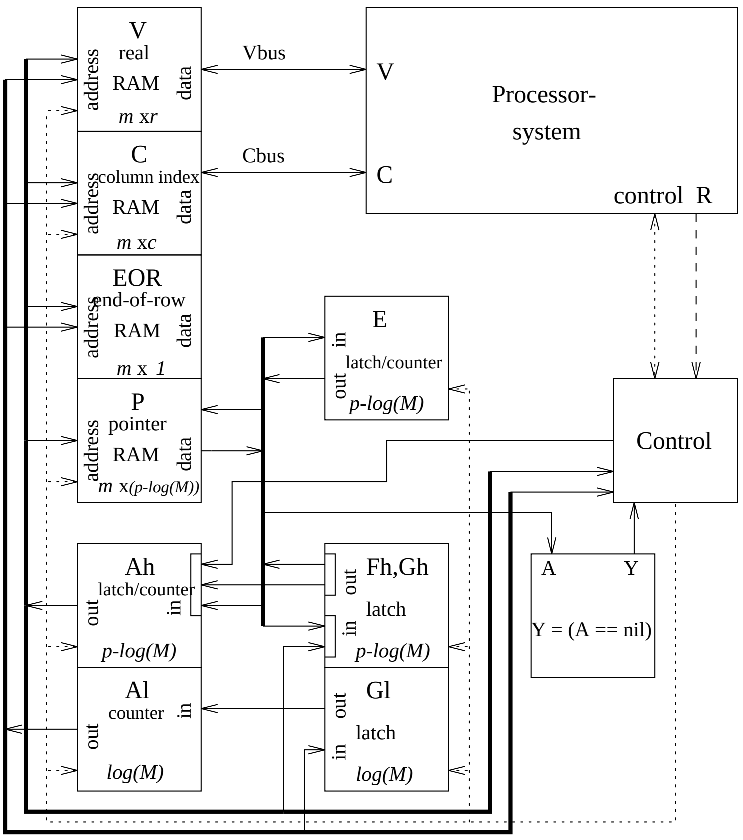

Figure 4: Architecture for block-wise storage of vectors

vector has a special status complicates the vector handling, but poses no real problems. (Again, by a change in microcode the variations in which the first block is either completely full or completely empty can be implemented).

We illustrate the behavior of the system with the basic operations, focusing on the loop only.

Initialization

The initialization is coded as:

```
I1: Ah = 1
L1: while (Ah <= 𝒩) do [M[Ah].P, M[Ah##0].EOR, Ah] = [nil, true, Ah+1]
I2: [Ah, E] = [0, (𝒩+1) mod (m/M)]
L2: while (E > 0) do [M[Ah].P, Ah, E] = [E, E, (E+1) mod (m/M)]
E1: M[Ah].P = E
```

Vector read

Reading a vector consists of counting within fields of the linked records:

```
I1: [Ah, Al] = [R, 0]
I2: [rdy,Al] = [M[Ah##Al].EOR, Al+1]
L1: while (not rdy) do
       [Vbus, Cbus, rdy] = [M[Ah##Al].V, M[Ah##Al].C, M[Ah##Al].EOR]
       if (Al == M - 1) then [Ah, Al] = [M[Ah].P, 0]
                        else Al = Al + 1
    enddo
```

The if-statement looks tricky, but it merely indicates the operation of the combination *Al* and *Ah*.

Vector extraction

During vector extraction all records but the first one of the row to be extracted are added to the list of empty records:

```
I1: [Ah, Al] = [R, 0]
I2: [rdy, Gh, Al] = [M[Ah##Al].EOR, M[Ah].P, Al+1]
L1: while (not rdy) do
      [Vbus, Cbus, rdy, Fh] = [M[Ah##Al].V, M[Ah##Al].C, M[Ah##Al].EOR, Ah]
      if (Al == M - 1) then [Ah, Al] = [M[Ah].P, 0]
                        else Al = Al + 1
    enddo
E1: if (Gh <> nil) then
      Ah = 0
E2:   [Ah, E] = [Fh, M[Ah].P]
E3:   [Ah, Al, M[Ah].P] = [R, 0, E]
E4:   [Ah, M[Ah##Al].EOR, M[Ah].P] = [0, true, nil]
E5:   M[Ah].P = Gh
    else
E1:   [Ah, Al] = [R, 0]
E2:   M[Ah##Al].EOR = true
    endif
```

Vector insertion

In contrast to the vector storage design, we demand that the vector which is to be inserted is empty.

```
I1: [Ah, Al, Gl] = [0,0,0,0]
I2: [E, Ah] = [M[Ah].P, R]
I3: [M[Ah].P, M[Ah##Al].EOR, Gh, Al] = [E, false, Ah, Al+1]
L1: while ((insert more) and (Ah <> nil)) do
      [M[Ah##Al].V, M[Ah##Al].C, M[Ah##Al].EOR, Gh, Gl] =
        [Vbus, Cbus, false, Ah, Al]
      if (Al == M - 1) then [Ah, Al] = [M[Ah].P, 0]
                        else Al = Al + 1
    enddo
E1: [Ah, Al] = [Gh, Gl]
E2: if (Ah == R) then
      [M[Ah##Al].EOR, M[Ah].P] = [true, nil];
    else
      [M[Ah##Al].EOR, E] = [true, M[Ah].P];
E3:   [M[Ah].P, Ah] = [nil, 0];
E4:   M[Ah].P = E;
    endif
```

3.3 Performance of vector storage design

In this section we compare the memory efficiency and the achievable bandwidth between traditional software and architectural solutions.

To store a matrix with $\mathcal{N} \cdot \rho$ entries using vector storage, we need $\mathcal{N}$ pointers of $r+c+p$ bits plus $\mathcal{N} \cdot \rho$ memory words of $r+c+p$ bits (Note that p bits would suffice, but the current design uses $r+c+p$ bits per pointer because the memory sizes of the V, C, and P parts are equal). The memory overhead compared to a frequently used standard software implementation, the row-wise storage scheme (see for example [9, 15, 19]), is illustrated in Figure 5. For the example size, and $\rho = 5$, the software vector storage is 38% more expensive in memory use than hardware row-wise storage. The hardware overhead in the vector storage

			example size	relative overhead
primary storage storage	SRW	$\mathcal{N}\rho r$	320 $\mathcal{N}$	
	HVS	$\mathcal{N}\rho r$	320 $\mathcal{N}$	
	HBS	$\mathcal{N}\rho r$	320 $\mathcal{N}$	
overhead storage p.el.	SRW	$\mathcal{N}\rho c$	80 $\mathcal{N}$	25 %
	HVS	$\mathcal{N}\rho(c+p)$	190 $\mathcal{N}$	59 %
	HBS	$\mathcal{N}\rho(c+\frac{p}{M})$	135 $\mathcal{N}$	42 %
overhead storage p.matrix	SRW	$\mathcal{N}2p$	44 $\mathcal{N}$	14 %
	HVS	$\mathcal{N}(r+c+p)$	102 $\mathcal{N}$	32 %
	HBS	$\mathcal{N}(r+c+\frac{p}{M})$	137 $\mathcal{N}$	43 %
total	SRW		444 $\mathcal{N}$	39 %
	HVS		612 $\mathcal{N}$	91 %
	HBS		592 $\mathcal{N}$	85 %

SRW = software row-wise
HVS = hardware vector storage
HBS = hardware block-wise vector storage

Figure 5: Memory efficiency of vector storage compared with row-wise storage (assuming $\rho = 5$).

design is small, in terms of transistor counts, but can be substantial in wiring and pin counts.

The pointer overhead can be decreased by block-wise storage, but internal fragmentation is introduced. The average memory use compared to row-wise storage in also depicted in Figure 5. The optimal value of M is approximately $\sqrt{\frac{p\cdot(2\rho+1)}{r+c}}$. Our example values give an optimal value for M of approximately 1.74. Only matrices which have a close to dense structure will benefit from a larger blocking factor M in the design.

The 'execution' time (in memory cycles) for the operations above are (for a vector with n elements): for initialization: m+2 clock cycles, for read vector: n+2 clock cycles, for extract vector: n+5 clock cycles, and for insert vector: n+7 clock cycles.

We compare these results to a software scheme performing the same operations on a row-wise storage. We assume a 64-bit memory system, which is the bottleneck in the system during these operations and count the memory accesses only. No caching is assumed. In this case the initialization of relevant data structures in software (both `LOW()` and `HIGH()` arrays) would cost approximately $2\mathcal{N}$ memory accesses. Reading or extracting a vector costs 2 memory accesses per element. All address computations can be executed at the register level, without introducing considerable overhead.
The (partial) insert of a vector costs 2 memory accesses per element, plus possible data movement and (expensive) garbage collection.

Hence, we gained a factor of 2 in execution speed. In addition, we removed the need for garbage collection, at the cost of a bus width increase factor of

1.59. (In a 32 bits architecture, this factor would have been 3.19, compared to a gain factor of 3 in execution speed).

The execution time for the block-wise vector storage is slightly better. The number of memory cycles remains the same. The hardware will operate at the same speed of 1 element per memory cycle. There is, however, one advantage: because the elements within a block fall always in the same page, the memory access time for all but the first element will profit from a faster access time if page mode DRAM is used[2]. The speed-up due to this effect is approximately $\frac{2 \cdot M}{M+1}$, compared to element-wise storage without any benefit from page-mode DRAM.

[2]In the element-wise system there was no support for page mode DRAM. The linked-list is initially a concatenation of all memory elements, thus profit of page-mode DRAM is expected. However, as computations proceed, the linked-list gets more and more scattered over memory and the advantage drops. Garbage collection might be used to improve this situation.

	V	R	C	P	W	E	N	S	comment
0	-	-	-	12	-	-	-	-	empty list pointer
1	-	-	-	-	15	5	15	9	row/col 1
2	-	-	-	-	8	7	11	11	row/col 2
3	-	-	-	-	13	6	16	5	row/col 3
4	-	-	-	-	16	10	8	10	row/col 4
5	`3.0`	1	3	-	1	15	3	7	a_{13}
6	`4.0`	3	1	-	3	13	9	15	a_{31}
7	`8.0`	2	3	-	2	11	5	13	a_{23}
8	`5.0`	2	4	-	11	2	10	4	a_{24}
9	`1.0`	4	1	-	10	16	1	6	a_{41}
10	`40.0`	4	4	-	4	9	4	8	a_{44}
11	`20.0`	2	2	-	7	8	2	2	a_{22}
12	-	-	-	14	-	-	-	-	empty
13	`30.0`	3	3	-	6	3	7	16	a_{33}
14	-	-	-	17	-	-	-	-	empty
15	`10.0`	1	1	-	5	1	6	1	a_{11}
16	`6.0`	4	3	-	9	4	13	3	a_{43}
17	-	-	-	18	-	-	-	-	empty
18	-	-	-	19	-	-	-	-	empty
19	-	-	-	0	-	-	-	-	empty

Figure 6: Contents of memory in *matrix storage* design

4 Matrix storage designs

The main disadvantage of the vector storage design is the inability to switch between row-wise and column-wise operation. The matrix storage design is proposed in order to combine the two different types of operation. To realize this, entries have to be linked in two directions, both column-wise and row-wise. Furthermore, if we extract a row, we need for every element in that row its two column-neighbors to update the involved column vector as well. The total data structure can be described as a bidirectionally doubly linked list. Singly linked bidirectionally linked lists have been used in production codes for static data structures, for example in SPICE [16]. Figures 6 and 7 illustrate the storage scheme for our example matrix.

The $\mathcal{N}$ row-pointers are stored at locations $1..\mathcal{N}$, using memory blocks W and E, initiating $\mathcal{N}$ circular doubly-linked lists. The $\mathcal{N}$ column-pointers are also mapped at addresses $1..\mathcal{N}$, but in memory blocks N and S.

4.1 Implementation of matrix storage

Figure 8 shows the elementary design. It is a straightforward extension of the vector storage design. Block C is duplicated as the block R and for every

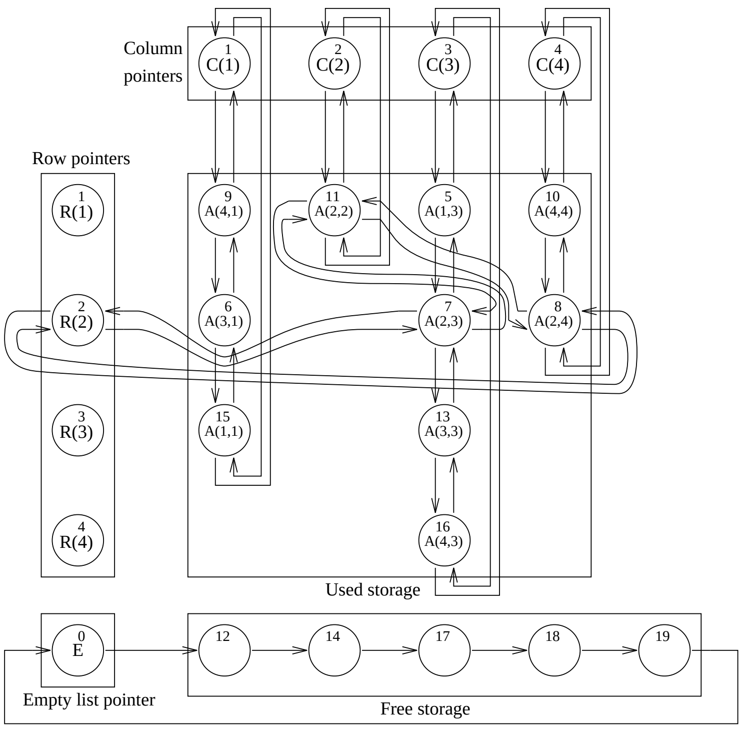

Figure 7: Storage of matrix A.

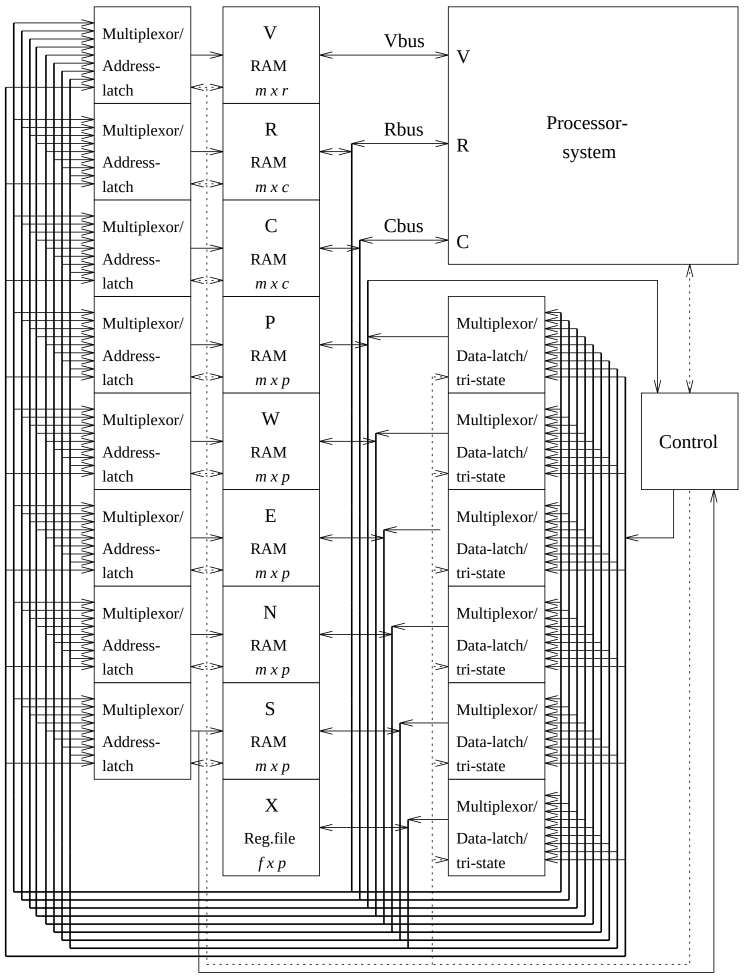

Figure 8: Architecture for element-wise matrix storage

element the row number is stored in R. Instead of the pointer P, we now use 4 pointers W, E, N, and S, for the pointers in four directions. Block P remains and is used to link free memory places together.

The following blocks have to be added for the matrix storage design:

Multiplexor/Address latch/Counter: These blocks consist of a two-stage block (multiplexor and address latch). Under control of *Control*, the address latches can be filled each memory-cycle with an address on one of the buses. The latches have also an increment capability. All multiplexors/latches function independently and simultaneously.

X: X is a small register file to store temporary values during the computation. We sometimes name the registers (e.g. `X[counter]`) to indicate their function. Some of these registers have also a count pin to increment the register contents.

The design in Figure 8 can be made more specific by looking at the usage. The number of address latches can not be decreased, because the memory parts perform possibly different operations on possibly different addresses. But most address-latches use only 3 of their inputlines; the multiplexors can be simplified using the exact knowledge which lines are used. This also reduces the size of the micro-instructions.

We will present example codes for the some basic operations on rows below. Initialization can be done in m memory cycles.

We use the following registers in the register file: `X[Prv]`, `X[Cur]`, and `X[Nxt]` for respectively the previous, current, and next element in the vector, and `X[Up]` and `X[Lo]` for respectively the upper and lower element (in the column vector). We will abbreviate these registers in the code fragments in this section as `Prv`, `Cur`, `Nxt`, `Lo`, and `Up`. Some of these registers are not actually used, as we will see in the sequel.

Initialization

For the first locations $1..\mathcal{N}$, the 4 pointers W, E, N, and S have to point to themselves, indicating empty circular doubly-linked lists. All other addresses have to be linked together to form a singly-linked list of unused locations.

```
I1: [AW,AE,AN,AS,DW,DA,DN,DS,cntr] = 1
L1: while (A <= 𝒩) do
      [AW,AE,AN,AS,DW,DE,DN,DS,cntr,M[AW].W,M[AE].E,M[AN].N,M[AS].S]=
         [cntr+1,cntr+1,cntr+1,cntr+1,cntr+1,cntr+1,cntr+1,cntr+1,cntr+1,
          DW,DE,DN,DS]
I2: [A, E] = [0, (𝒩+1) mod m]
L2: while (E > 0) do [M[A].P, A, E] = [E, E, (E+1) mod m]
E1: M[A].P = E
```

Vector read

A vector read is simply a linked-list traversal. Assume that `X[Cur]` (`Cur`) points to the first (next) element (in row Rbus) at the start of the first (or next) iteration. Before every iteration the condition (X[Cur] $\neq$ R) has to be checked. The next element can be read using the following code:

```
Cur = M[Rbus].E
while (Cur <> nil) do
  [Cur,Vbus,Cbus] = [M[Cur].E,M[Cur].V,M[Cur].C]
enddo
```

In the implemented micro-code the indirect addressing is removed by rewriting the code. For example, the code line `X[Cur] = M[X[Cur]].E` is replaced by `AE = X[Cur]` followed by `X[Cur] = M[AE].E`. The address and data latches of all memory blocks show up as variables in the code. The names are created by adding an *A* or a *D* before the memory block name. The latches AE, AV, AR, and AC have the same function as register X[Cur] before, and register X[Cur] can be removed from the code.

```
AE = Rbus
[AE,AV,AC] = [M[AE].E,M[AE].E,M[AE].E]
while (AE <> nil) do
  [AE,AV,AC,Vbus,Cbus] = [M[AE].E,M[AE].E,M[AE].E,M[AE].V,M[AE]C]
enddo
```

Vector extraction

Extraction of a vector involves the search for neighbours and the update of their pointers. This costs two cycles per element. Figure 7 depicts a possible storage of matrix A, and Figure 9 shows the change in the data structure after the extraction of row 2.

```
I1: Cur = R
I2: [Cur, Empty] = [M[Cur].E, M[0].P]
L1: if (Cur <> R) then
      while (Cur <> R) do
        [Up, Lo, Cur, Prv, Vbus, Rbus, Cbus] =
          [M[Cur].N, M[Cur].S, M[Cur].E, Cur, M[Cur].V, M[Cur].R, M[Cur].C]
L2:     [M[Up].S, M[Lo].N, M[Prv].P] = [Lo, Up, Cur]
      enddo
E1:   M[Prv].P = Empty
E2:   Empty = M[R].E
E3:   M[0].P = Empty
E4:   [M[R].E, M[R].W] = [R, R]
    endif
```

Vector insertion

During insertion of a row, every new element is inserted as the first element of the corresponding column. This costs 3 memory cycles per element. Figure 10 depicts the reinsertion of row 2 (in the order a_{22}, a_{23}, and a_{24}) after the previous delete of this row in Figure 9.
This can be coded as:

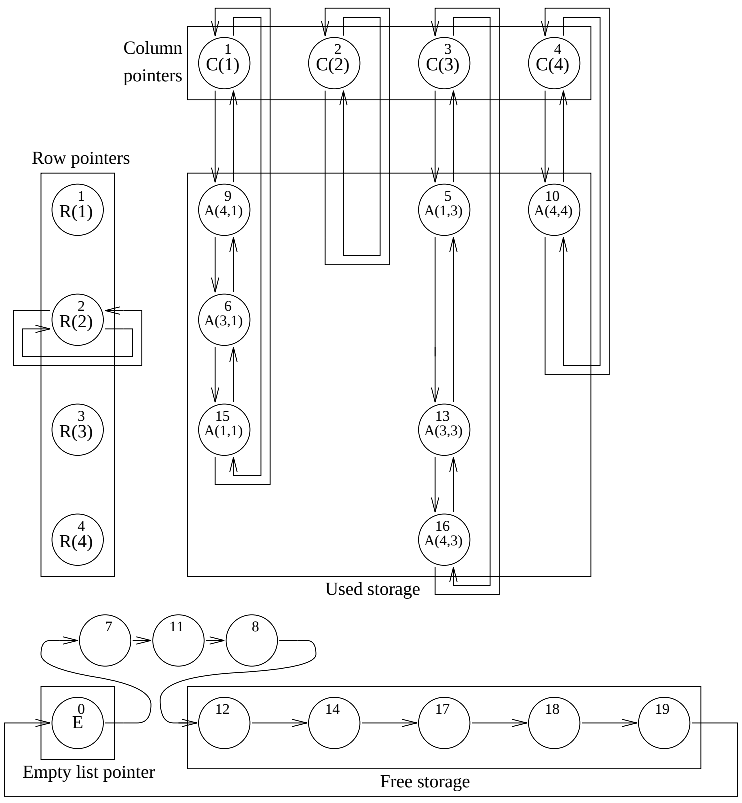

Figure 9: Storage of matrix A after row 2 has been deleted.

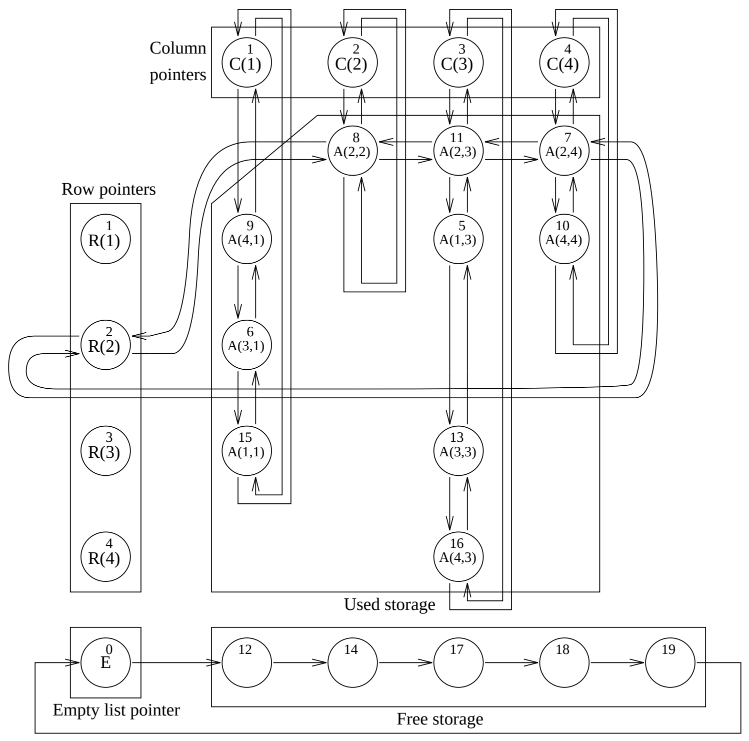

Figure 10: Storage of matrix A after row 2 has been deleted and reinserted.

```
I1: [Prv,Cur,Empty] = [M[Rbus].W,M[0].P,M[0].P]
L1: while ((insert more) and (Cur <> nil)) do
      Lo = M[Cbus].S
L2:   [M[Cbus].S,M[Lo].N,Nxt] = [Cur,Cur,M[Cur].P]
L3:   [M[Cur].N,M[Cur].S,M[Cur].E,M[Cur].W,M[Cur].R,M[Cur].C,M[Cur].V,Prv,Cur] =
        [Cbus,Lo,Nxt,Prv,Rbus,Cbus,Vbus,Cur,Nxt]
    endwhile
E1: if (insert more) out-of-memory-error
E2: [M[0].P,Cur] = [Cur,M[Rbus].W]
E3: M[Cur].E = Empty
E4: [M[Prv].E, M[Rbus].W] = [Rbus, Prv]
```

Rewriting this code to explicit register-form, a need for 4 memory cycles per iteration is suggested by the code. However, the first and last statement can be combined over the loop, improving throughput to 3 memory cycles per iteration.

The number of memory cycles per element can be reduced to 2.5 by reordering statements and alternating inserts via the S and N banks.

4.2 Pipelined storage of matrices using interleaved memory and write buffers

If we increase the available memory bandwidth by interleaving, the operations presented for the matrix storage design can be pipelined. Let b be the number of memory banks. Interleaving offers the possibility to create a special bank, bank 0, for the row and column pointers, and the V, R, C, and P blocks can be omitted. Banks 1 up to $b-1$ provide full functionality.

The microcode determines how interleaving can be exploited. Two of the many possibilities are: (1) Every bank contains locally linked lists and the elements of a vector are distributed over s (for $s < b$) separate linked lists, each in a different bank. The available bandwidth is increased by a factor s, but the number of row pointers in bank 0 is also increased by a factor s. (2) Per vector there is one linked list scattered over one or more banks. In the sequel, we will show how method (2) can be exploited.

Vector read

The only gain that can be exploited during a read is the possibility to read more than one vector at a time. However, some bank conflicts will occur. Nevertheless, if we prioritize the current vector read we might simultaneously start to read (prefetch) the next $b-2$ vectors. An upper bound for the speed-up is $b-1$.

Vector extraction

The code presented in Section 4.1 for vector extraction is

```
L1: [Up, Lo, Cur, Prv, Vbus, Rbus, Cbus] =
      [M[Cur].N, M[Cur].S, M[Cur].E, Cur, M[Cur].V, M[Cur].R, M[Cur].C]
L2: [M[Up].S, M[Lo].N, M[Prv].P] = [Lo, Up, Cur]
```

In the loop body the N and S banks are used twice, once for a read (each) and once for a write. Using multiple banks and a write buffer for conflicting memory accesses enables the extraction to continue at a rate of one element per memory cycle.

If the elements are distributed over at least 2 banks, the available memory bandwidth is sufficient provided the matrix elements are reasonably distributed over the banks. A full write buffer will stall the pipeline.

Vector insertion

The S bank has the highest memory load during vector insertion: 3 references per inserted element, as is illustrated by the following loop body:

```
L1: Lo = M[Cbus].S
L2: [M[Cbus].S,M[Lo].N,Nxt] = [Cur,Cur,M[Cur].P]
L3: [M[Cur].N,M[Cur].S,M[Cur].E,M[Cur].W,M[Cur].R,M[Cur].C,M[Cur].V,Prv,Cur] =
      [Cbus,Lo,Nxt,Prv,Rbus,Cbus,Vbus,Cur,Nxt]
```

To pipeline the insert, we require the existence of $b-1$ linked lists of unused memory locations, one per (full) bank. We assume that in step 1 the places of the $b-1$ heads of these $b-1$ lists are known. Immediately after the first read during the insertion of an element, it is known in which bank the 'south neighbor' of the element 'to be inserted' remains. By placing this element in a different bank, we obtain a conflict-free pipeline.

The pipeline stalls when it is impossible to find a free element (Cur) according to the rule stated above. Hence, it is important for all rows and columns to have a fair distribution of the elements over the memory banks. Increasing the number of banks helps to ease this problem but is expensive.

4.3 Performance of matrix storage designs

Figure 11 compares the memory demands of the matrix storage design with row-wise storage. This comparison is not fair, because a row-wise storage does not allow column-wise operations. A software scheme based on the same method of storage would consume about the same amount of memory.

Figure 12 presents an overview of the number of memory cycles per vector element for the presented implementations. A software scheme would cost considerably more due to the number of assignments which are parallelized in the matrix storage design. In practice, no comparable implementations are used.

[3]With page-mode DRAM: approximately $\frac{M+1}{2\cdot M}$ memory cycles.

			example size	relative overhead
primary storage	SRW	$\mathcal{N}\rho r$	320 $\mathcal{N}$	
	HMS	$\mathcal{N}\rho r$	320 $\mathcal{N}$	
	HPS	$\mathcal{N}\rho r$	320 $\mathcal{N}$	
overhead storage p.el.	SRW	$\mathcal{N}\rho c$	80 $\mathcal{N}$	25 %
	HMS	$\mathcal{N}\rho(5p+2c)$	710 $\mathcal{N}$	222 %
	HPS	$\mathcal{N}\rho(5p+2c)$	710 $\mathcal{N}$	222 %
overhead storage p.matrix	SRW	$\mathcal{N}2p$	44 $\mathcal{N}$	14 %
	HMS	$\mathcal{N}(r+5p+2c)$	206 $\mathcal{N}$	64 %
	HPS	$\mathcal{N}4p$	88 $\mathcal{N}$	28 %
total	SRW		444 $\mathcal{N}$	39 %
	HMS		1236 $\mathcal{N}$	286 %
	HPS		1118 $\mathcal{N}$	249 %

SRW = software row-wise
HMS = hardware matrix storage
HPS = hardware pipelined matrix storage

Figure 11: Memory efficiency of matrix storage compared with row-wise storage (assuming $\rho = 5$).

	vector storage		matrix storage		
	element-wise	block-wise[3]	standard	pipelined	
				worst-case	typ.
read	1	1	1	1	1
extract	1	1	2	2	1
insert	1	1	2.5	2	1

Figure 12: Memory cycles per operation for some hardware designs

	software
RD ALOW	$\mathcal{N}$
RD AHIGH	$\mathcal{N}$
RD AVAL	$\mathcal{N} \cdot \rho$
RD AIND	$\mathcal{N} \cdot \rho$
RD B	$\mathcal{N} \cdot \rho$
WR C	$\mathcal{N}$
total	$\mathcal{N} \cdot (3\rho + 2)$

Figure 13: Memory references for SpMxV using row-wise storage.

5 Example: Sparse matrix Vector Multiply

In this section we compare the performance of several designs for a frequently used sparse matrix application: sparse matrix vector multiply (SpMxV). The code for SpMxV based on row-wise storage is presented below:

```
do I=1,𝒩
  S = 0
  do JA = ALOW(I),AHIGH(I)
    S = S + AVAL(JA) * B(AIND(JA))
  enddo
  C[I]=S
enddo
```

The number of memory references for the arrays is presented in Figure 13. We consider only references to ALOW, AHIGH, AVAL, AIND and B. The total number of memory references is $\mathcal{N} \cdot (3\rho + 2)$. We assume no cache. This is an unrealistic approach, but so far no cache has been included in the matrix storage design.

Assuming we store the dense vector B at addresses $1..\mathcal{N}$, the standard matrix storage design needs $\mathcal{N} \cdot 2 \cdot \min(\rho, 1)$ references to obtain all elements of A and B. Indirect addressing is handled within the matrix storage design, using the flexible micro-code. The overhead per vector disappears for $\rho \geq 1$ because of the parallel reads in memory and the pipeline possibilities.

6 Implementation issues

By adding hardware it would be very simple to enable the processor to use the memory within the architecture as a standard RAM. Hence, all the memory installed can be used for normal purposes if no sparse computations are performed. The extra costs of this address logic are small. The width of the memory system should be a multiple of the processor word length.

The use of microcode yields great flexibility. For example, the free places in the V-bank at addresses 1..N can be used (1) for an independent dense vector (useful for sparse matrix vector multiply), or (2) can be used as a gathered sparse (i.e., dense) vector of one of the rows or columns of the matrix, or (3) can be used for the storage of the first element in a row (or column), or (4) can be used for the storage of the diagonal elements (especially useful for LU-decomposition). This decision is hardware-independent. It is also possible to maintain a separation between the L and U elements during LU-decomposition, either by consequently inserting the L elements at the east side and the U elements at the west side of the row pointer, or by the use of a double header array. Another hardware-independent choice is the way the interleaved memory is exploited. We proposed a linked-list across more banks, to enhance the insertion of elements. An alternative implementation uses several linked-lists, one per bank, to enhance the read of vectors.

To implement the support for more than one matrix at a time, some address arithmetic is needed. For every matrix, a base address is added to address pointers. Base addresses are found in a small table in the same memory.

In the sequel of this section we discuss a few possible ways to embed the memory architecture in an existing computing environment.

Library functions

A very efficient way to use the memory is by means of library functions. Such a library consists of two parts: downloadable micro-code for the control unit of the architecture and code for the host processor. The library can be highly optimized for a number of frequently used problems and support general operations for other problems.

Compiler supported code generation

In [4, 5] a sparse compiler is described, capable of transforming code that operates on dense matrices which are actually sparse, directly into corresponding sparse code. During this conversion, the compiler can identify those parts of the code that can exploit the proposed designs and target the resulting sparse code accordingly. A similar approach is taken in [18].

Virtual memory

If a large enough address space is available, the following approach is feasible. A subspace of 2^{2c} words ($2c$ address bits) is reserved for the sparse memory. The memory simulates a dense matrix, i.e., the code may contain direct references to dense, full matrices, which are in fact sparse. All operations are element-wise. Clearly, this is very inefficient in general. Every read or write to an entry takes on average $\frac{\rho}{2}$ memory cycles, reads or writes to non-entries consume on average ρ memory cycles. However, writes to an arbitrary element which is known to be a non-entry is possible in 3 memory cycles, and consecutive writes to known non-entries can be pipelined resulting in a rate of 1 insertion per 2.5 memory cycles (1 memory cycle for the interleaved design). The resulting efficiency of such a design might be discouraging, but fewer changes are needed in the code. A combination of the addressing method described here and the sparse compiler in the previous section may offer new possibilities.

Prefetching sparse vectors

Because of the gap between cpu speed and memory speed, it is advantageous to perform memory loads in advance without intervention of the processor (see for example also [10, 13]). If instructions are implemented to prefetch complete sparse vectors into a (second-level) cache, the processor may continue during the prefetch. The sparse vector can be stored in the cache in two ways: dense (row-wise storage) or sparse (by an automatic scatter). For a valid scatter operation the dense vector in cache has to be initialized to zero. The sparse vector should remain in cache until it is used, to profit from this approach.

7 Cache

So far, we did not implement a cache or cache support in the hardware design. This is needed for two reasons: to reduce the latency of memory accesses, and to enable element-wise operations, which are not supported by the DRAM. Although the work on cache is preliminary, we propose the following cache structure: a fully-associative cache to store the V values, which is not addressed by the addresses used in the DRAM, but which is associatively addressed by the row and column indices R and C. In addition, we need two bit-arrays of size $\mathcal{N}$. If a bit in this array is set, then we are sure that this row is completely in cache. If it is not set, it may or may not be completely in cache. Cache control takes care that any matrix element is either in cache or in DRAM, but never in both. This simplifies the use of the circuitry. The main disadvantage is the write-back obligation of unmodified data in cache which is no longer needed. The advantage is the faster retrieval of arbitrary elements left in DRAM, because the remaining vectors are - on average - of shorter length.

For every row (column), The associativity process can be used in two ways: (1) match on R and C, and produce a matching cache-line, or a cache miss; or (2) match on R *or* C, and produce the first matching cache-line for which the so-called 'used'-bit it false. Then, set the 'used'-bit for that line. The cache supports a reset of all the 'used'-bits of a row. After such a reset, the row elements can be read from cache one by one.

Replacement and placement policy

An LRU-based algorithm on vector-basis can be applied to replace vectors (and put them back in DRAM). We have a choice of either writing the vector in cache back into DRAM, or writing the vector back to DRAM on a element-by-element basis.

If a read leads to a miss in cache, and the bit-array corresponding to the row and column index of that element are both reset, the element may be in DRAM. Two approaches to handle this situation are possible: (1) a (partial) load of a row (or column) into cache, followed by a cache hit; or (2) the use of a special cache bypass to load the element from memory.

The same two approaches hold for vector operations: a vector read of a row, which is not known to be completely in cache, has to result in either (1) fetching the rest of the vector from DRAM to cache, setting the corresponding bit in the bit-array; or (2) reading the DRAM vector part (using the cache bypass) during or after the cached elements have been read.

Example: vector addition

Consider a (possibly repeated) saxpy operation $V = V + \alpha W$. First, V is read into cache if it is not already completely present as indicated by the bit-array. Since the elements are never duplicated, only the missing elements have to be read. Then, W is read element by element from either DRAM or cache. For every element of W, the corresponding element of V is searched for in cache. The values of W and V are sent to the processor, using 0 if no element of V could be found, and the returned value is inserted in the vector V. For $\alpha = 1$, the elements of W for which no corresponding element in V has been found have to been inserted into V.

Clearly, this approach works only if the cache is large enough to store the resulting vector V. If this is not the case, the use of a bit-array for the vector V may be useful. In this bit-array the bits are set if and only if the corresponding element of V is an entry in the data structure. On a cache miss, the corresponding bit indicates the necessity to perform an insert or a search/update in DRAM.

8 Conclusions

In this paper we presented two types of memory architectures, one for the storage of sparse vectors and one for the storage of sparse matrices. These memories are capable of vector read, extraction and insertion at a speed of 1 to 3 memory cycles per element. Reading can be done at the highest speed by all systems. The addition of write buffers and interleaving to the matrix storage improves the insertion and extraction performance, but improves the bandwidth during reads only if a parallel read can be exploited.

The main improvement of sparse computations is expected if the architectures in this paper are extended with a cache and specific sparse compiler support. Future research is necessary in this direction.

Acknowledgements The authors would like to express their thanks to Aart Bik and Olivier Temam for their helpful comments.

References

[1] Hideharu Amano, Takaichi Yoshida, and Hideo Aiso. $(SM)^2$: Sparse Matrix Solving Machine. In *Proceedings on Computer Architecture*, pages 213–220, 1983.

[2] Hideharu Amano, Takaichi Yoshida, Tomohiro Kudoh, and Hideo Aiso. $(SM)^2$-II : A new version of the Sparse Matrix Solving Machine. In *Proceedings on Computer Architecture*, pages 100–107, 1985.

[3] U. Banerjee, D. Gajski, and D. Kuck. Accessing sparse arrays in parallel memories. *Journal of VLSI and Computer Systems*, pages 69–100, 1983.

[4] Aart J.C. Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, pages 416–424, 1993.

[5] Aart J.C. Bik and Harry A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Lecture Notes in Computer Science, No. 768*, pages 57–75. Springer-Verlag, 1994.

[6] B. Codenotti, G. Lotti, and F. Romani. Area-time trade-offs for matrix-vector multiplication. *Journal of Parallel and Distributed Computing*, 8:52–59, 1990.

[7] B. Codenotti and F. Romani. A compact and modular VLSI design for the solution of general sparse linear systems. *Integration: The VLSI journal*, 5(1):77–86, March 1986.

[8] Timothy A. Davis and Edward S. Davidson. Pairwise reduction for the direct, parallel solution of sparse, unsymmetric sets of linear equations. *IEEE Transactions on Computers*, 37(12):1648–1654, December 1988.

[9] Iain S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1990.

[10] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *Proceedings on International Supercomputing*, pages 354–368, 1990.

[11] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[12] R.N. Ibbett, T.M. Hopkins, and K.I.M. McKinnon. Architectural mechanisms to support sparse vector processing. In *Proceedings on Computer Architecture*, pages 64–71, 1989.

[13] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. In *Proceedings on Computer Architecture*, pages 43–53, 1991.

[14] John G. Lewis and Horst D. Simon. The impact of hardware gather/scatter on sparse Gaussian elimination. *SIAM J. Sci. Stat. Comput.*, Volume 9:304–311, 1988.

[15] Sergio Pissanetsky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[16] A.L. Sangiovanni-Vincentelli. Circuit simulation. In *Computer Design Aids for VLSI Circuits*, pages 19–112, 1981.

[17] Omar Wing. A content-addressable systolic array for sparse matrix computations. *Journal of Parallel and Distributed Computing*, 2:170–181, 1985.

[18] A. Wolfe, M. Breternitz Jr., C. Stephens, A.L. Ting, D.B. Kirk, R.P. Bianchini Jr., and J.P. Shen. The White Dwarf: A high-performance application-specific processor. In *Proceedings on Computer Architecture*, pages 212–222, 1988.

[19] Zahari Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publishers, 1991.