Sparse Matrix Storage Format

Fethulah Smailbegovic, Georgi N. Gaydadjiev, Stamatis Vassiliadis
Computer Engineering Laboratory,
Electrical Engineering Mathematics and Computer Science
Mekelweg 4, 2628CD Delft
TU Delft
fethulah@computer.org
{stamatis,georgi}@ce.et.tudelft.nl

Abstract— **Operations on Sparse Matrices are the key computational kernels in many scientific and engineering applications. They are characterized with poor substantiated performance. It is not uncommon for microprocessors to gain only 10-20% of their peak floating-point performance when doing sparse matrix computations even when special vector processors have been added as coprocessor facilities. In this paper we present new data format for sparse matrix storage. This format facilitates the continuous reuse of elements in the processing array. In comparison to other formats we achieve lower storage efficiency (only an extra bit per non-zero elements). A conjuncture of the proposed approach is that the hardware execution efficiency on sparse matrices can be improved.**

Keywords—**Sparse Matrix Formats, Operation Efficiency, Hardware**

I. Introduction

Many numerical problems in real life applications such as engineering, scientific computing and economics use huge matrices with very few non-zero elements, referred to as sparse matrices. As there is no reason to store and operate on a huge number of zeros, it is often necessary to modify the existing algorithms to take advantage of the sparse structure of the matrix. Sparse matrices can be easily compressed, yielding significant savings in memory usage. Several sparse matrix formats exist, like the Compressed Row Storage [9], the Jagged Diagonal Format [8] or Compressed Diagonal Storage format [2]. Each format takes advantage of a specific property of the sparse matrix, and therefore achieves different degree of space efficiency. The operations on sparse matrices are performed using their storage formats directly [1], [5], [6], [4]. This suggests that formats can influence the efficiency of the executed operations. The algorithms for least square problems or image reconstruction, for example, need to solve sparse linear systems using iterative methods involving large number of sparse matrix vector multiplications. Operations on Sparse Matrices tend to yield lower performance on general purpose processors due to sparse nature of data structure, which causes large number of cache misses and large number of memory accesses. As consequence, it leaves the processing elements not fully utilized.

Recently, Field Programmable Gate Arrays showed promising platform for implementation of operations on sparse matrices, like Sparse Matrix Vector Multiplication [1], [10]. Growing FPGA capacity in terms of on-chip memory, embedded elements, I/O pins and high floating point performance, make FPGA attractive platform for implementation.

Considering the recent achievements in implementation of operations on sparse matrices we propose a simple and efficient format, an extended Sparse Block Compressed Row Storage (SBCRSx) for sparse matrices. The format allows high degree of data reuse and therefore the format will enable us to achieve substantial performance gains over the recent implementations of operations on Sparse Matrices.

Therefore, the contribution of our work can be summarized as Simple Sparse Matrix format for efficient hardware implementation of arithmetic operations such as matrix by vector multiplication. The existing formats do not provide the hardware with essential information about the sparse matrix layout, since their primary focus is on space efficiency. We identify the need for formats to provide additional information on sparse matrix to the hardware. This is to enable efficient implementation of arithmetic operations on Sparse Matrices.

The remainder of the paper is organized as follows: In section II a brief background information on Sparse Matrix Formats is presented. Section III describes and gives evaluation of the proposed format. Finally, section IV concludes the paper.

II. Background

One widely used, space efficient sparse matrix representation is Compressed Row Storage [9] (see figure 1).

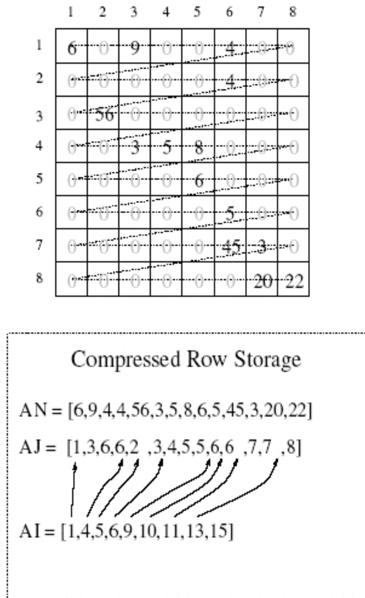

Fig. 1. Compressed Row Storage

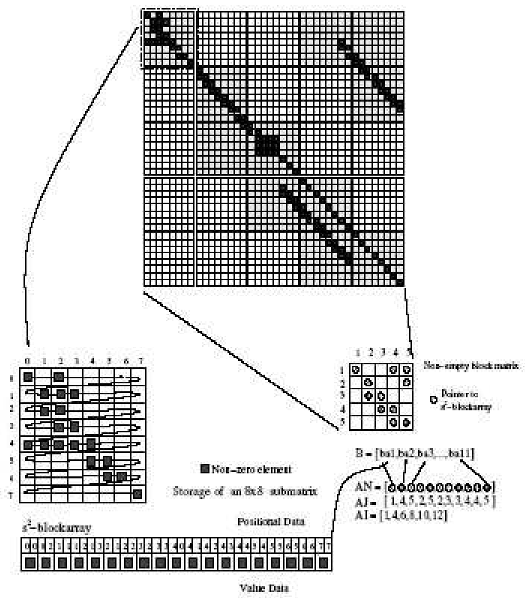

Fig. 2. Sparse Block Compressed Row Storage

The matrix is represented with three vectors: row AI, column AJ and matrix values AN vectors. An 1-dimensional vector AN is constructed that contains all the values of the non-zero elements taken in a row-wise fashion from the matrix. The next vector, 1-dimensional vector AJ of length equal to the length of AN is constructed that contains the original column positions of the corresponding elements in AN. Each element in vector AI is a pointer to the first non-zero element of each row in vectors AN and AJ. 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 focuses on matrices of general type, it does not takes into account particular sparse matrix patterns that may be inherent to specific application types.

A more efficient way to store the information is the Sparse Block Compressed Row Storage format [7]. Sparse Block Compressed Row Storage is presented in figure 2. The matrix is divided into blocks. The final output is the block compressed matrix where each element is a block from the original matrix. In each block we store every matrix element, with its value and position information using a single vector. On the block compressed matrix the standard compressed row storage scheme is performed. Thereafter, the vector with block coding is added where each block coding is represented.

The problem of implementation of operations on sparse matrices raises the following question: *What information a sparse matrix format should provide hardware with, to enable subsequent efficient implementation?* The current formats cope with the problems like, short vectors, indexed accesses, and positional information overhead, but essential problem is how to influence hardware to yield at the end execution of efficient operations. The difficulty with sparse matrix storage schemes is that they include additional overhead (to store indices as well as numerical values of nonzero matrix entries) than the simple arrays used for dense matrices[3], and arithmetic operations on the data stored in them usually cannot be performed as efficient either (due to indirect addressing of operands). To address the above questions, we have extended previously proposed Sparse Block Compressed Row Storage.

III. The proposed format and evaluation

In this section we propose an extension of the SPBCRS format described in II – the Extended Sparse Block Compressed Row Storage (SPBCRSx). Our goal is to enable as much as possible spatial reuse of on-chip resources. The existing formats store the positional and value information in three separate entities. Through indirect operands addressing we connect the value information with corresponding positional information. We believe that value and positional information must be seen as a unique entity and stored as a short linked list. Therefore, matrix storage scheme would consist of short linked lists as elements. Accessing one element of the matrix would mean accessing the value and positional information in subsequent manner. Also, these lists for every matrix element should connect between each other in case the elements

```
i – row index, j-column index
for each element of matrix NxN
  for i = 0 to N-1 loop
    for j = 0 to N-1 loop
      if value_matrix(i,j) is non zero then /* if non zero element found
              write the values and positi on information*/
        position (i,j) <= position_matrix(i,j);
        value (i,j) <= value_matrix (i,j);
        seen_nonzero_element <= seen_nonzero_element + 1;
        chain_bit_tmp(i,j) <= '1'; /*set temporary chain bit to 1*/
      else
          NULL; /* if zero element found do nothing*/
      end if;
    end loop;
      if seen_nonzero_element > 1 then /* when completed a row look if
                                          there were non -zero elements*/
        for j = 0 to N-1 loop
          chain_bit (i,j) <= chain_bit_tmp(i,j); /* if yes than commit */
        end loop;                  /* temporary chain bits to real chain bits */
      else
        NULL; /* otherwise do nothing*/
      end if;
  end loop;
end loop;
```

Fig. 3. Sparse Matrix to SPBCRSx transformation

are from the same row. By doing this, we create short data vectors, referred to as chunks, with their known beginning and ending address. Since these chunks have different lengths and only finite number of chunks can fit on the available on-chip resources, we augment the value and the positional information with another field (the chaining bit). The chaining bit tells the hardware if there are more elements of the same row to follow. In case not, the hardware can conclude the summation off the row and store the result in the outgoing vector. The algorithm for the SPBCRSx transformation is presented in figure 3. The algorithm would be executed runtime on the host processor. In the algorithm, we scan each element in the current row and search for the non-zero elements. If a non-zero element is found then the value and position data are stored into linked list for the matrix element. We count each non-zero element in the row. The chaining bit at this moment is temporary set to '1'. After we finish a row, we examine the counter of non-zero elements. If the counter is greater than 1, we then commit the temporary chain bits to real chain bits for every found non-zero matrix element. This procedure is repeated for each row in the sparse matrix. After we finish with the Extended Sparse Block Compressed Row Storage transformation, we divide the non-coded matrix into blocks of SxS, and take for each SxS block coded representation. This representation is now an element in

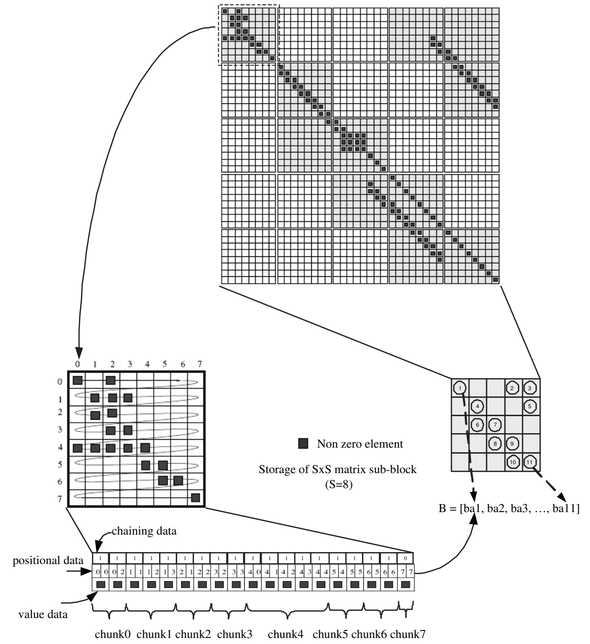

Fig. 4. The Extended Sparse Block Compressed Row Storage

the compressed matrix. How we store any sparse matrix in Extended Sparse Block Compressed Row Storage (SPBCRSx) is presented in the figure 4.

To clarify the discussion we consider the same matrix as in figure 1. As shown on figure 5, for each element

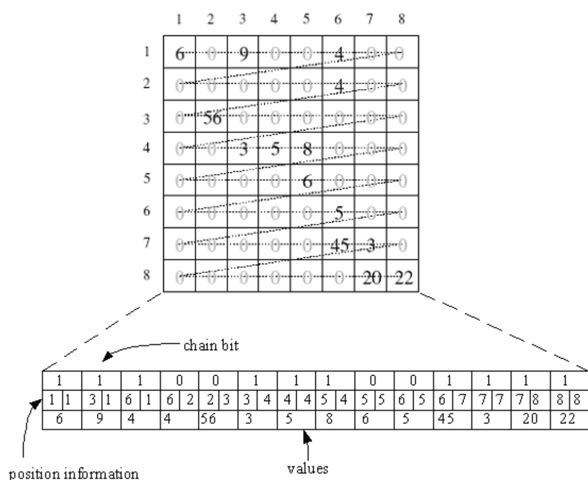

Fig. 5. Example of SPBCRSx

in the matrix a linked list is created. In this example ten elements are with chaining bits set to '1'. This information is important for the hardware to get the information about the matrix structure. The chaining information tells the hardware that there more than one non-zero elements are present in the matrix row. As a consequence, we have created for the rows, with more than one non-zero element, the chunks. For example, the elements with values 6, 9 and 4 build one chunk of data. The elements 3, 5 and 8 built another chunk of data. The chunks represent data group of the same row matrix elements and bundled with other chunks built a group with increased spatial data locality.

The proposed SPBCRSx format stores the sparse matrix into the same entities as SPBCRCS with an additional bit for having the value '1' for all but the last non-zero elements in a row. These extra bits allow for the hardware to detect the "end" of a row and avoid idle execution time when performing arithmetic operations. The above provides the capabilities to speed-up execution and facilitate the hardware unit design. The conjuncture suggested in the previous paragraph constitutes further research topic and its validity will be considered and tested for common sparse vector arithmetic operations such as matrix by vector multiplication.

IV. Conclusions

Operation on Sparse Matrices are important part in computational sciences. In this paper we proposed a new sparse matrix format and explained the expected benefits for the hardware execution units. We suggest that at expenses of an extra bit per value representation when compared to existing techniques that the design and speed-up of sparse matrix operations could be facilitated.

References

[1] M. deLorimier and A. DeHon. Floating-point sparse matrix-vector multiply for fpgas. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2005.

[2] J. Dongarra. Sparse matrix storage formats. In *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, SIAM*, 2000.

[3] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point fpga matrix multiplication. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2005.

[4] P. 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, November 2001.

[5] S. Vassiliadis, S. D. Cotofana, and P. Stathis. Vector isa extension for sprase matrix multiplication. In *Proceedings of EuroPar'99 Parallel Processing*, pages 708–715, September 1999.

[6] S. Vassiliadis, S. D. Cotofana, and P. 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, August 2000.

[7] S. Vassiliadis, S. D. Cotofana, and P. Stathis. Block based compression storage expected performance. In *Proc. 14th Int. Conf. on High Performance Computing Systems and Applications (HPCS 2000)*, June 2000.

[8] Y.Saad. Krylov subspace methods on supercomputers. In *SIAM Journal on Scientific and Statistical Computing*, 1989.

[9] J. D. Z. Bai, J. Dongarra, A. Ruhe, and H. van der Vorst. Templates for the solution of algebraic eigenvalue problems: A practical guide. In *Society for Industrial and Applied Mathematics*, 2000.

[10] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, New York, NY, USA, 2005. ACM Press.