Design Considerations of a Multiple Inner Product and Accumulate Vector Functional Unit

Pyrrhos Stathis Stamatis Vassiliadis Sorin Cotofana
Electrical Engineering Department,
Delft University of Technology,
Delft, The Netherlands
Email: {pyrrhos,stamatis,sorin}@dutepp0.et.tudelft.nl

Abstract— **A large number of scientific applications require computations that involve operations on sparse matrices. Due to irregularities induced by the diverse sparsity patterns, many operations on sparse matrices execute inefficiently on traditional scalar and vector architectures. In order to tackle this issue a scheme has been proposed [1] that alleviates the sparse matrix storage and computation overhead on vector processors. The scheme introduces a new sparse matrix storage format and utilizes the Multiple Inner Product and Accumulate (MIPA) vector pipelined functional unit to perform the sparse matrix vector multiplication, the function that constitutes the core operation of most sparse matrix applications. The implementation of the MIPA functional unit poses a number of challenges and its design considerations will be the focus of this work. The MIPA unit operates on a vector containing the nonzero elements of a sparse matrix with the associated positional information and a multiplicand vector in order to produce a vector containing a number of inner products. The vector of nonzero elements, as defined by the scheme in [1], represents a number of partial rows of the sparse matrix. Therefore, the processing of the vector produces multiple data streams within the MIPA unit that correspond to the multiplicity of rows it represents. This fact, combined with the necessity to feed back data over a multi-stage floating point adder to perform the accumulation, results in a scheduling problem. In this paper we propose a MIPA functional unit design that addresses this issue in an efficient manner. Simulation results on a sparse matrix benchmark suit suggest that by using our proposed scheduling scheme utilizing a multiple pipeline implementation of the functional unit we can achieve a near optimal resource utilization.**

Keywords— **vector processing, sparse matrices, matrix vector multiplication**

I. Introduction

In many scientific computing areas the manipulation of sparse matrices constitutes the kernel of the solvers. Due to the irregularities however of the matrix' sparsity patterns, i.e. the distribution of the non-zeros within the matrix, make many operations on sparse matrices execute inefficiently on traditional scalar and vector architectures. This problem has been tackled both software and hardware approaches. Most of the approaches are in software [2], [3], because they are less costly. However, research focused on hardware approaches [4], [5], [6], [7] indicates that much greater improvements can be obtained. In [6] the authors report a speedup of up to 3 times (depending on the sparsity pattern) when compared to the aforementioned JD method on a conventional vector processor when performing sparse matrix vector multiplication (SMVM) using a scheme that includes an Augmented Vector Architecture (AVA) and an associated sparse matrix storage scheme (BBCS). Furthermore, using a hierarchical version of the BBCS scheme called the Hierarchical Sparse Matrix (HiSM) scheme, SMVM performance improvements of on average 4-5 times have been achieved.

The problem addressed in this paper is that of the implementation of the vector Functional Unit that supports the execution of the SMVM using the aforementioned schemes. The Functional Unit is named the Multiple Inner Product and Accumulate (MIPA) Unit and was already described briefly in [8]. However, the unit was assumed to process integer rather than floating point values making the implementation rather straightforward. When floating point values are processed the implementation becomes more complicated due to the feedback required over more than one of the pipeline stages. In this paper we address this considerations and present an evaluation of the mechanism. The contributions of this paper can be summarized as follows:

- We propose and describe a pipelined mechanism for supporting the SMVM on vector processors using the BBCS and HiSM schemes. The unit is implemented as a vector functional unit and performs the Multiple Inner Product and Accumulate (MIPA) function.
- We evaluate and estimate the performance of the MIPA functional unit.

The remainder of the paper is organized as follows: In the next Section we provide with some background information on vector processors and the hierarchical sparse

matrix storage format and sparse matrix vector multiplication. In Section III we describe and evaluate the performance of the proposed mechanism and finally, in Section IV we draw some conclusions.

II. Background

This section provides some background information and assumptions made throughout the paper.

Before proceeding with the description MIPA functional unit we will first give a brief description of the *hierarchical storage format*, the sparse matrix format that we will assume for the remainder of our paper and which is a hierarchical variation of the aforementioned BBCS format: 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* of the vector architecture. Each of these $s \times s$ sub-matrices, which we will call s^2-blocks, is then stored separately in memory in the following way: All the non-zero values as well as the positional information combined are stored in a row-wise fashion in an array (s^2-blockarray) in memory. In Figure 1 (bottom left) we can observe how such a blockarray is formed containing both the position and value data from the top left s^2-block of an 64×64 sparse matrix. The section size is $s = 8$. Note that the positional data consists of the column and row position of the non-zero element with the s^2-block The s^2-blockarrays can contain up to s^2 non-zero elements and we will assume that an AVA can operate on these in the same way as described in [6].

These s^2-blockarrays that describe the non-empty s^2-blocks form the lowest (*zero*) level of the hierarchical structure of our format. As can be observed in Figure 1, the non-empty s^2-blocks form a similar sparsity pattern as the non-zero values within an s^2-block, Therefore, the next level of the hierarchy, level-1, is formed in exactly the same way as level *zero* with the difference that the values of non-zero elements are replaced by the pointers to the s^2-blockarrays in memory that describe non-empty s^2-blocks. This new array which contains the pointers to the lower level is stored in exactly the same fashion in memory (see Figure 1 (bottom right). Notice that at level-1 the pointers are stored in a column-wise fashion. In this way an access pattern is provided where the $s \times s^2$-element-wide columns are accessed row-wise. This is favorable for operations such as matrix-vector multiplication (refer to [1] for a more elaborate discussion). The next level, level-2, if there is one (in the example of Figure 1 there is none), is formed in the same way as level-1 with the pointers pointing at the s^2-blockarrays of level-1. Further, as in any hierarchical structure the higher levels are formed in the same way and we proceed until we have covered the entire matrix in $max(\lceil \log_s M \rceil, \lceil \log_s N \rceil)$ levels. 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. The formats are identical for all levels.

In order to perform the Sparse Matrix Vector Multiplication (SMVM) using the HiSM format we need to hierarchically traverse all the s^2-blockarrays and perform a local multiplication of the s^2-blocks at level-0 and the corresponding part of the Multiplicand Vector (MV) producing a partial Result Vector (RV). The described process is depicted in Figure 2 for $s = 4$. This multiplication is performed by the proposed functional Unit within the vector processor. Vector processors, such as the one depicted in Figure 3 are based on architectures that support the execution of *vector instructions*. On most current vector architectures [9], the vectors are copied from the main memory into *vector registers* within the processor before they are operated upon. Vector registers are arrays of scalar registers that hold (parts of) the vectors to be processed. Due to the fact that the vector register length can not be arbitrarily large, when operating on large vectors they have to be divided into smaller parts, a technique that is usually called *strip mining*, each of which cannot be larger than the maximum amount of elements a vector register can hold, i.e., the architecturally defined *section size* of the VP. In a VP the operations are carried out by (usually) pipelined *Func-*

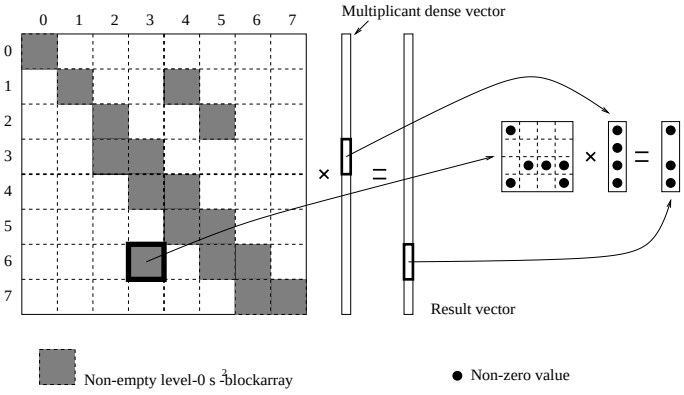

Fig. 2. Sparse Matrix Vector Multiplication (SMVM) using the Hierarchical Sparse Matrix (HiSM) storage format

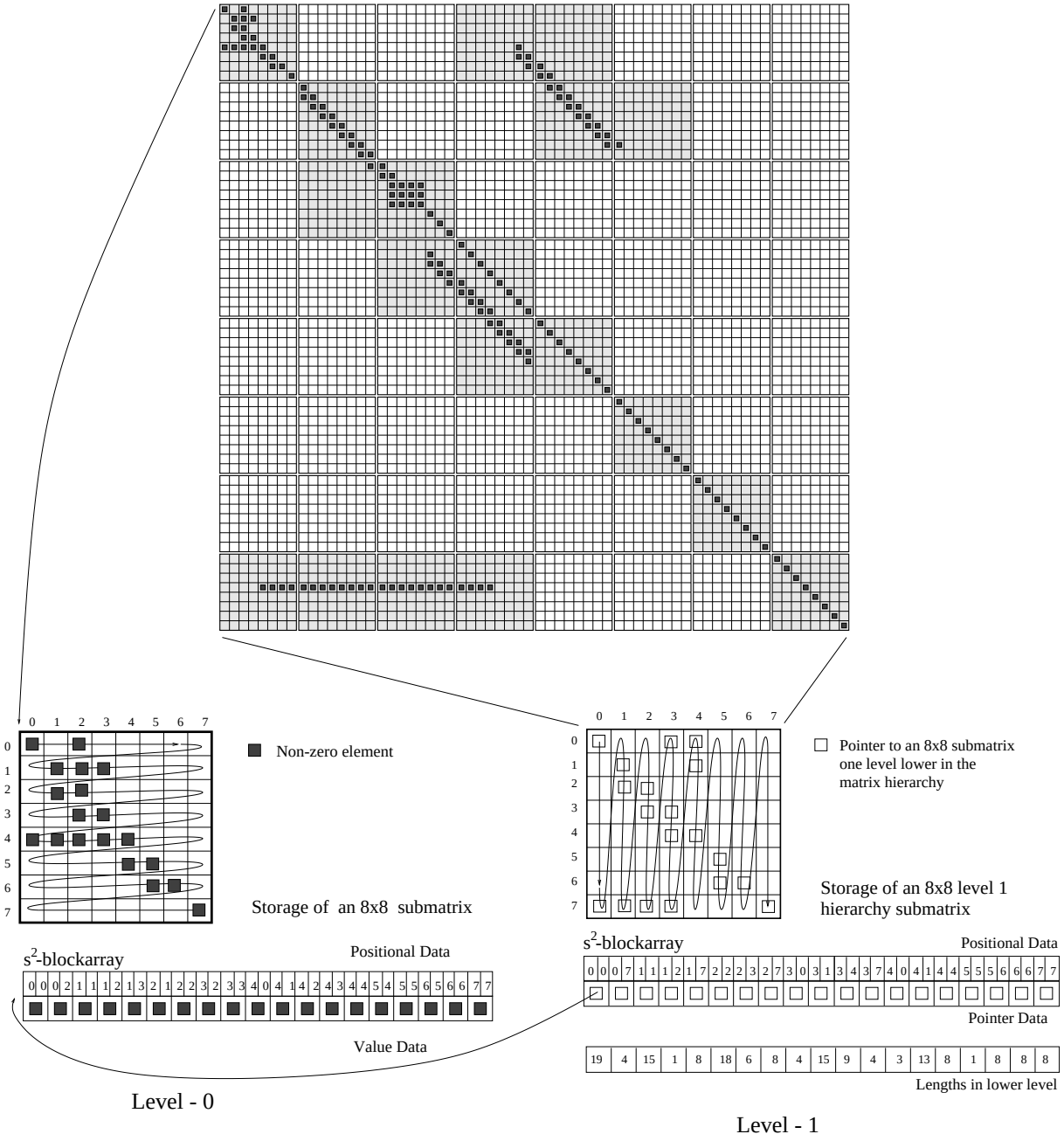

Fig. 1. Example of the Hierarchical Sparse Matrix Storage Format

tional Units (FU) that are able to fetch one or more new element per cycle from each of the source vector register(s) involved, operate on it/them, and return the result(s) to the result (vector) register.

To support the HiSM SMVM two new vector instructions have been introduced to augment the functionality of a traditional vector processor. First, the LoaD Section`LDS` instruction loads a section of the s^2-blockarray into two vector registers of the vector processor. The first vector register is filled with the non-zero values of the s^2-block. A second vector register is loaded with the corresponding column and row positions of those elements. The second instruction, the `MIPA` instruction performs the multiplication of the section loaded by the `LDS` by a dense vector which resides in a third register and produces the partial result which is stored in a forth vector register. The `MIPA` instruction is using the MIPA functional unit that will be described in the following Section.

III. The MIPA Pipeline

This section describes the workings of the MIPA functional unit. 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.

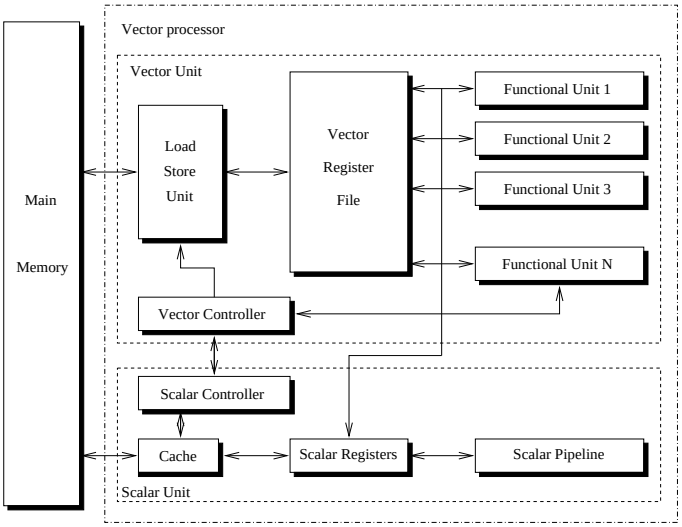

Fig. 3. Vector Architecture

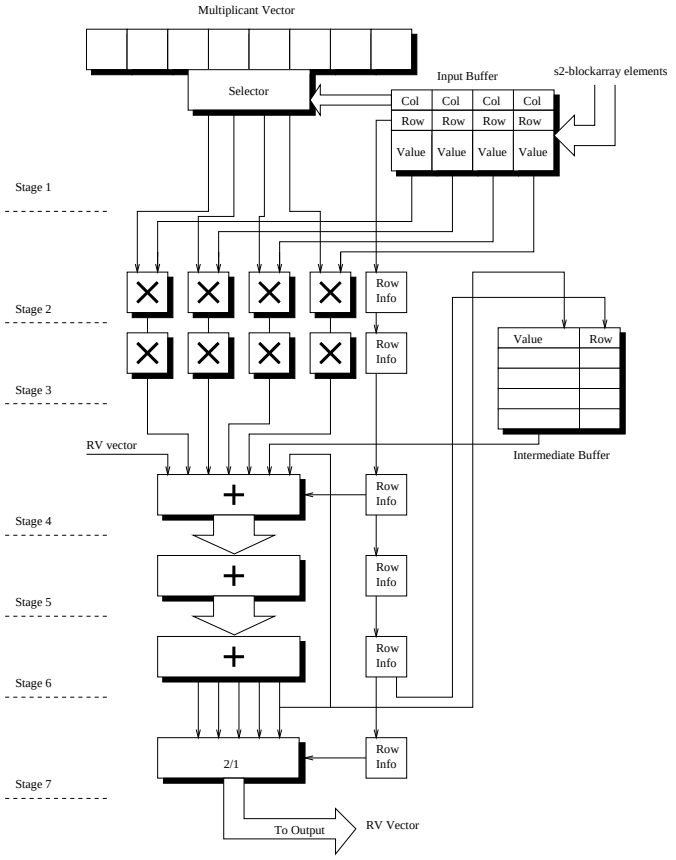

Fig. 4. The Multiple Inner Product and Accumulate (MIPA) vector functional unit

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 4. 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 4, 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 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.

A. Timing Evaluation

In this section we will provide performance estimations of the proposed MIPA 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 MIPA 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 MIPA 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 MIPA 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 $\lceil \frac{s}{p} \rceil + w + 7$.

IV. Conclusions

In this we have proposed and described a pipelined mechanism for supporting the SMVM on vector processors using the BBCS and HiSM schemes. The unit is implemented as a vector functional unit and performs the Multiple Inner Product and Accumulate (MIPA) function when invoking the `MIPA` vector instruction. Furthermore, we evaluated and estimated the performance of the MIPA functional unit and showed that the number of cycles to complete the `MIPA` instruction will lie between $\lceil \frac{s}{p} \rceil + w$ and $\lceil \frac{s}{p} \rceil + w + 7$ depending on the positional information of the input vector.

References

[1] S. Vassiliadis, S. Cotofana, and Pyrrhos Stathis, "Vector isa extension sprase matrix multiplication.," in *EuroPar'99 Parallel Processing*. 1999, Lecture Notes in Computer Science, No. 1685, pp. 708–715, Springer-Verlag.

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

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

[4] Hideharu Amano, Taisuke Boku, Tomohiro Kudoh, and Hideo Aiso, "$(SM)^2$-II: A new version of the sparse matrix solving machine," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Boston, Massachusetts, June 17–19, 1985, IEEE Computer Society TCA and ACM SIGARCH, pp. 100–107.

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

[6] Pyrrhos Stathis, Stamatis Vassiliadis, and Sorin Cotofana, "Sparse matrix vector multiplication evaluation using the bbcs scheme," To appear in 8th PCI, Nov 2001.

[7] A. Wolfe, M. Breternitz, Jr., C. Stephens, A. L. Ting, D. B. Kirk, R. P. Bianchini, Jr., and J. P. Shen, "The white dwarf: A high-performance application-specific processor," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, H. J. Siegel, Ed., Honolulu, Hawaii, May–June 1988, pp. 212–222, IEEE Computer Society Press.

[8] Stamatis Vassiliadis, Sorin Cotofana, and Pyrrhos Stathis, "Block based compression storage expected performance," in *Proceedings of HPCS2000, Victoria*, 2000, pp. 389–406.

[9] John L. Hennessy and David A. Patterson, *Computer Architecture A Quantative Approach*, Morgan Kaufman, San Mateo, California, 1990.