Reconfigurable Sparse/Dense Matrix-Vector Multiplier

Georgi Kuzmanov, Mottaqiallah Taouil
Computer Engineering Lab, EEMCS, Delft University of Technology, Mekelweg 4, Delft, The Netherlands
G.K.Kuzmanov@TUdelft.NL, M.Taouil@student.TUdelft.NL

***Abstract*—We propose an ANSI/IEEE-754 double precision floating-point matrix-vector multiplier. Its main feature is the capability to process efficiently both Dense Matrix-Vector Multiplications (DMVM) and Sparse Matrix-Vector Multiplications (SMVM). The design is composed of multiple processing elements (PE) and is optimized for FPGAs. We investigate theoretically the boundary conditions when the DMVM equals the SMVM performance with respect to the matrix sparsity. Thus, we can determine the most efficient processing mode configuration with respect to the input data sparsity. Furthermore, we evaluate our design both with simulations and on real hardware. We experimented on an Altix 450 machine using the SGI Reconfigurable Application Specific Computing (RASC) services, which couple Dual-Core Itanium-2 processors with Virtex-4 LX200 FPGAs. Our design has been routed and executed on the Altix 450 machine at 100 MHz. Experimental results suggest that only two PEs suffice to outperform the pure software SMVM execution. The performance improvement at the kernel level scales near linearly to the number of configured PEs both for the SMVM and DMVM. Compared to related work, the design does not indicate any performance degradation and performs equally or better than designs optimized either for SMVM or DMVM alone.**

I. Introduction

Many scientific applications involve computations with huge matrices of double precision floating-point data - both sparse and dense. When the number of zero elements is dominant in a matrix, the latter is considered sparse and it should be stored and processed in a specific way to avoid the huge number of trivial operations with zero elements. On the other hand, dense matrix operations require different design approaches, which are mainly focused on identifying and exploiting higher data-level parallelism and target efficient utilization of the memory bandwidth. Traditionally, in many complex scientific applications, the typical way to perform the computationally intensive sparse/dense matrix operations efficiently is to employ supercomputing power, achieved by numerous interconnected general purpose processors (GPP). The main drawback of this solution is its high hardware cost. An alternative approach to accelerate matrix computations relies on reconfigurable hardware, exploiting the huge processing parallelism the contemporary FPGA devices allow. The different design targets for sparse and dense matrix operations, however, traditionally impose separate design solutions for each of the two cases. Therefore, in literature, all reconfigurable proposals support either sparse or dense matrix operations, assuming that the target application data are always either sparse or dense. In reality, however, data often change their sparsity dynamically, resulting into performance degradation of the specific hardware. Therefore, a matrix processor that can easily switch between sparse and dense computational modes preserving high performance efficiency, would alleviate this data dependant speed degradation problem.

In this paper, we consider one of the most popular matrix operations - matrix-vector multiplication and, contrary to the tradition, we approach the design problem both from the dense and the sparse data prospective. As a result, we propose a design capable to process equally well both sparse and dense matrices given particular resource limitations such as bandwidth and silicon area. Our main contributions are:

- We merge an original sparse matrix-vector multiplier (SMVM) with dense matrix-vector multiplication (DMVM) in an original design based on a common double precision floating-point PE;
- We investigate theoretically the boundary conditions for performance efficient processing between sparse and dense matrix computational modes;
- The proposed design supports multiple operations, namely: Sparse Matrix by Dense Vector multiplication, Dense Matrix by Dense Vector multiplication, Dense Matrix by Dense Matrix multiplication;
- Experimental results on a RASC augmented SGI Altix 450 machine suggest no performance degradation compared to related DMVM work and higher sustained performance compared to pure software implementations.
- Our design achieves the highest sustained performance among related reconfigurable proposals for SMVM ranging between 70% and 99% of the peak performance.

The remainder of the paper is organized as follows: In Section II, the computational schemes we utilize for both DMVM and SMVM are introduced. Section III describes the proposed microarchitecture for the SMVM design, followed by the integrated design description that can compute both SMVM and DMVM. Furthermore, the theoretical analysis of the boundary condition for switching between sparse and dense matrix multiplications are presented in Section IV. Section V reports our experimental results and provides their analysis. Finally, Section VI concludes the paper.

II. Computational Scheme

Mathematically, the matrix vector product **c** = A**b**, with dimensions of A $M \times N$, **b** and **c** - $N \times 1$, is presented as:

$$c_i \quad = \quad \sum_{x=0}^{N-1} a_{i,x} \cdot b_x \qquad (1)$$

where $a_{i,x}$ is the i-th element in column x of matrix A, b_x is the x-th element of the input vector **b**, and c_i is the i-th element of vector **c**. For the DMVM in our design, we adopt the computational scheme from [1]. Equation (1) can be defined for the sparse matrix vector product identically. The difference is that the sparse matrix stores non-zeroes only. Different sparse matrix formats have been proposed to compress the storage space efficiently. We consider the most widely referenced and used one - the general Compressed Row Storage (CRS) format, illustrated in Figure 1.

Original matrix

$$\begin{bmatrix} 1 & 0 & 0 & 6 & 0 \\ 0 & 2 & 0 & 7 & 0 \\ 0 & 0 & 3 & 0 & 8 \\ 10 & 11 & 0 & 4 & 9 \\ 0 & 0 & 12 & 13 & 5 \end{bmatrix}$$

CRS-format

```
values = [ 1, 6, 2, 7, 3, 8, 10, 11, 4, 9, 12, 13, 5]
     col = [ 0, 3, 1, 3, 2, 4,   0,   1, 3, 4,   2,   3, 4]
row_ptr = [ 0, 2, 4, 6, 10, 13]
```

Fig. 1. The Compressed Row sparse matrix storage format.

For SMVM, we propose a computational scheme derived from the DMVM scheme. The proposed SMVM computational scheme is illustrated in Figure 2. Identically to DMVM,

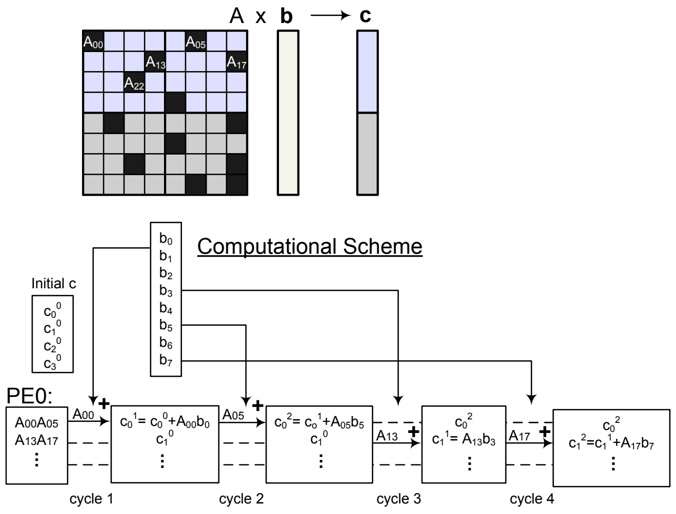

Fig. 2. Computational scheme of the SMVM product.

we assign several matrix rows to a PE. Ideally, we compute new temporary row results each cycle, by adding the product of the next non-zero value of the current row to the appropriate value of **b**. Due to the different row lengths, each executable row is processed immediately after a preceding row has been completed. This way, higher computational density and better resource utilization are achieved, leading to higher design performance. Practically, the latency to complete a floating point operation requires multiple cycles, which have to be considered when the scheme from Figure 2 is implemented.

III. Microarchitecture

The main unit of the Processing Element is an improved version of the multiply-add core component presented in [1]. An overview of the PE for the SMVM is depicted in Figure 3. The total latency of the multiply-add core is 11 cycles. Since

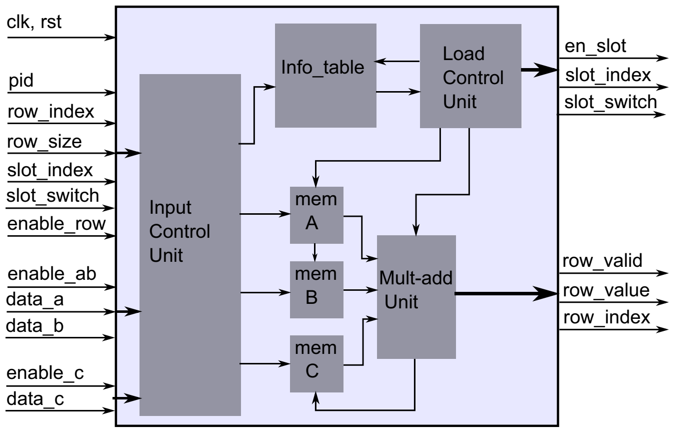

Fig. 3. Organization of the SMVM Processing Element

the adder latency in the reference core [2] is 8 cycles, 8 different rows are simultaneously active during a computation. To overlap the computations with the storage initialization of the memory, we create 16 memory slots, organized in pairs. We refer to a *slot* as to a *memory space reserved for a particular time period*. From each pair of slots, one is used for computations and the other - for communications. The functionality of each component, residing inside the PE, is briefly described below.

Input Control Unit: The Input Control Unit controls the inputs of the PE, its main tasks are as follows:

- Accept rows. Each row is stored in a free slot.
- Forward row related information, such as size and row index, in the Info_Table once the row is read in completely. Validate the slot, so that the row can be computed.
- Write the values of A, **b** and **c** into the correct indexes.

When all elements of the row are read in, the valid signal to the component Info Table is raised.

Info Table: The component Info_Table stores all information required to use the slots. For each slot, it stores: the corresponding **row size**, the processed **row index**, **current element index**, a **valid data** flag and a flag, that indicates whether a slot is **active** (used for computation) or is used for communication.

Load Control Unit: The load controller is based on one bit up counter that generates the addresses for the Info_Table unit. If a non-valid row has access to the multiply add unit, the Floating Point unit will be disabled. If a valid row is returned, a row address for the Memories A, B and C will be generated, based on the switch value for the current block, its address, and the value of *current_index*. The multiply-add unit will be enabled if the row stored in the slot that currently accesses the floating point unit contains valid data. If *current_index* equals the row size of the slot, additional controls signal the final row results. These control signals are passed together with *row_index* through the Floating Point unit to the output of the PE. When a row has finished its computations, the controller indicates that the slot has became free.

Memories A, B and C: Memory A is used to store the active row elements of matrix A, memory B - the elements of vector **b**, memory C - the initial values of vector **c**, temporary and final row results. Individual rows of the matrix are assigned to empty memory slots. The address signals connected

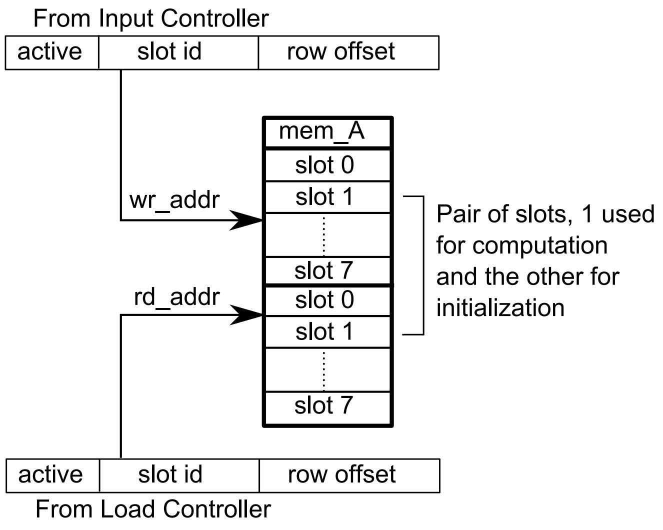

Fig. 4. Organization of memory A.

to memory A are depicted in Figure 4. The Input Controller stores elements at the correct addresses by combining the active signal, slot id and the row offset. The load controller generates the address to read the data and the output of the memory is forwarded directly to the Multiply-Add unit.

Memory A consist of $2 \times row\_length \times slots\_number$ entries, where number 2 represents one memory for buffering and another one for active computations for each slot and *slots_number* equals the latency of an adder, in our case it is 8. Memory B is organized in exactly the same way as memory A.

Memory C is responsible for storing the initial, temporary and final row results. The Multiply Add unit is reading temporary results from the memory address generated by the Load Control Unit and stores the new temporary result at the same location. Only one entry is required per slot to store initial or temporary row results. Two Dual Port memories, both 8 entries long, store the values for all the slots.

Table I reports the total hardware cost for each component of the PE. The total cost in the table is determined in terms of Xilinx Virtex 4 slices (used logic + routing) for the PE. It does not include 8 additional DSP48 blocks (18 x 18 bit multipliers) and the Block RAMS for memories A, B and C.

TABLE I
POST PLACE-AND-ROUTE RESULTS FOR THE COMPONENTS OF THE PE REPORTED BY THE ISE 10.1 TOOL SET FOR VIRTEX 4

Component	slices	frequency (MHZ)
Input Controller	95	265
Info Table	162	252
Load Controller	84	449
Multiply Add	1,414	159
Processing Element	2,140	156

All the memories inside the Info_Table that store row related information are generated using distributed reconfigurable logic.

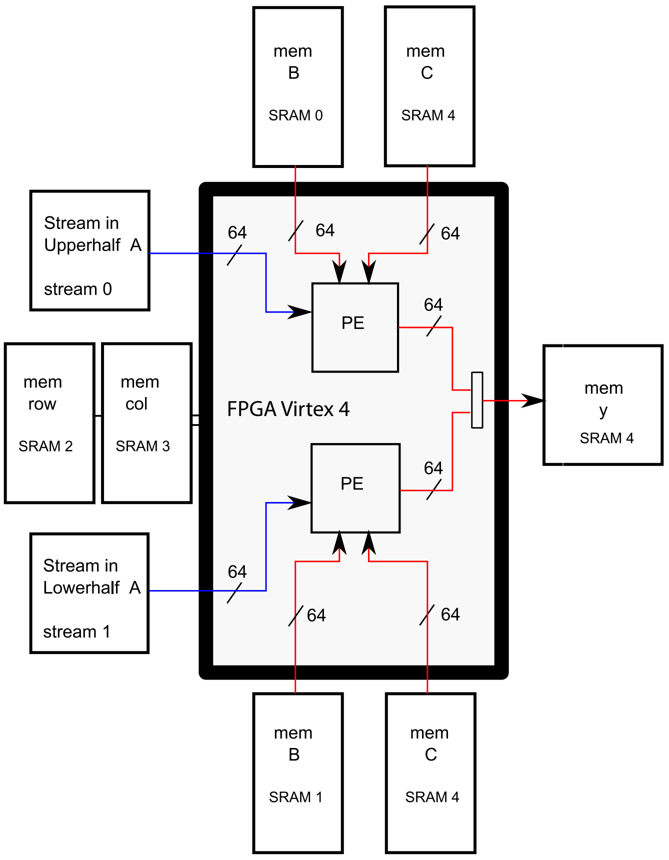

Fig. 5. Organization of the memories for the SGI RASC implementation.

Memory controller: Currently, we implemented relatively simple, unoptimized memory controllers that feed data from the memories to the PE compliant with the interface of the Input Control Unit. Figure 5 illustrates the mapping for 2 PEs of the arrays that store the matrix in the external FPGA memory banks of the SGI RASC-core. One SGI Altix 450 RASC-core contains as many as 5 SRAM banks (Off-Chip Memory) and 4 Input and Output DMA streams, which directly connect the CPU main memory with the FPGA.

Combined PE for Sparse and Dense Matrix Multiplication: The combined processing element is the PE for the SMVM, slightly modified with some control signals that multiplex addresses to the memories. The PE for the SMVM in Figure 3 is similar to the Dense Matrix multiplication (DMM) PE with the following differences:

- The DMM PE does not require the following components: The Input Control Unit, the Info_Table and the Load Control Unit. When a DMVM is performed these components stay idle. The area cost for these components is very small compared to the total PE cost as can be observed from Table I.
- Memory A is reused both by the DMVM and SMVM. Multiplexors select, which addresses gain memory access.
- Memory B is not available for the DMM PE, a register is used to store values of vector **b** instead.
- Similarly to memory A, a multiplexor is used to load the values from the correct addresses of memory C into the pipelines of the multiply-add core.

IV. Theoretical Performance Analysis

In this section, we analyze the proposed design theoretically through several mathematical formulas. Our main target is to identify the boundary conditions, which determine the configuration with the highest performance efficiency with respect to data sparsity. The performance is mainly limited by the bandwidth and by the available hardware, i.e., by the number of PEs. In the equations to follow, we denote performance with P [FLOPS], and the number of PE - with N_{PE}; the operation frequency is f_{op} [Hz]. Each PE contains one multiplier and one adder, i.e., it performs 2 floating-point operations per cycle. Therefore, the performance for the SMVM (P_{sparse}) is limited by N_{PE}:

$$P_{sparse} = 2N_{P_E} f_{op} \tag{2}$$

Let us denote the number of non-zero elements by N_z and the common dimension of the matrix and the vector by N. Then the total number of memory accesses can be approximated to $2.5N_z + 2.5N$ and thus the performance as a function of the limited bandwidth B can be determined by:

$$P_{sparse} = \frac{2N_z B}{\frac{5}{2}N_z + \frac{5}{2}N} = \frac{4N_z B}{5\left(N_z + N\right)} \tag{3}$$

Furthermore, the number of floating point operations for the sparse matrix are $2N_z$ and the execution time $t_{exec} = \frac{total_{operations}}{P}, \Rightarrow$

$$t_{exec,sparse} = \frac{5(N_z + N)}{2B} \tag{4}$$

The performance for the dense matrix vector product is limited by $2B$ and the number of operations required are $2N^2 \Rightarrow$

$$t_{exec,dense} = \frac{N^2}{B} \tag{5}$$

The sparsity of a matrix can be defined as:

$$\gamma = \frac{N_z}{N^2} \tag{6}$$

By equalizing the execution time for DMVM (4) and SMVM (5) and solving for N_z, we can substitute in (6):

$$\gamma' = 0.4 - \frac{1}{N} \tag{7}$$

The boundary condition separating the efficient DMVM from SMVM processing is determined by (7). If the matrix is sparser than γ', it is faster to perform SMVM, otherwise DMVM should perform better. Figure 6 depicts the above considerations graphically.

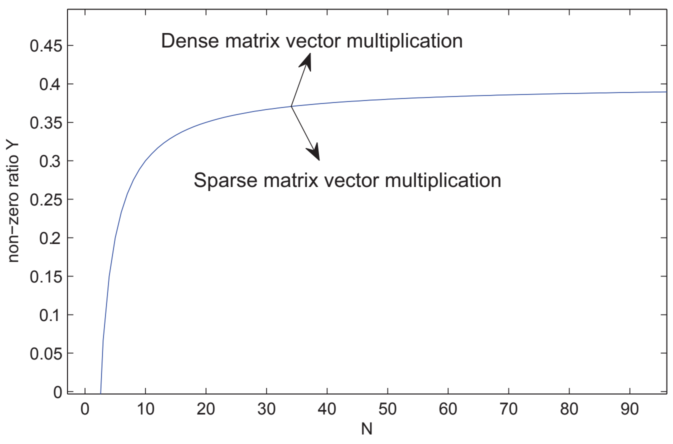

Fig. 6. Boundary condition to select between SMVM and DMVM.

V. Experimental Results

Our design has been developed using the Xilinx ISE 10.1.3 toolset and simulated in Modelsim 6.0d. The SMVM and DMVM VHDL codes have been targeted to RASC [3], installed in an Itanium-2 based SGI Altix 450 server. A number of matrices from various real-life engineering and scientific applications have been considered as benchmarks. The SMVM design is evaluated for the matrices shown in Table II, obtained from the University of Florida Sparse Matrix Collection [4]. For each matrix, Table II includes: the matrix/vector dimension N, the number of non-zero elements N_z, the sparsity γ, the minimal and the maximal non-zero entries per row.

Software and Hardware performance measurement: Figure 7 provides the performance results of our hardware design compared to software execution on Itanium-2 GPP. As a reference software we used OSKI [5], which is an optimized kernel for sparse matrix vector products. It includes optimizations like cache and register blocking. The OSKI tool has been compiled with the Intel 10.1.015 compiler with optimization level -O3. The Itanium-2 9130M is used for the software experiments and contains 16 kB L1 data cache, 256 kB L2 data cache and 4 MB L3 data cache. The machine operates at 1669 MHz, with the Front Side Bus (FSB) operating at 667 MHz. Figure 7 provides performance results for SMVM hardware with overdimensioned memory bandwidth. For this case, the performance is limited by the number of PEs and each array is stored in a separate memory. The charts suggest that the performance scales well with the number of PEs. The design is implemented in real hardware and the performance is measured for Custom Computing Units

TABLE II
PROPERTIES OF MATRICES USED TO EVALUATE OUR DESIGN

Number	Kind of matrix	Matrix	N	N_z	$\gamma[*10^{-4}]$	min/row	max/row
1	Power Network problem	gemat12	4929	33044	13.6	2	44
2	acoustics problem	k3plates	11107	378927	30.7	15	58
3	semiconductor device problem	wang3	26064	177168	2.6	4	7
4	optimization problem	jnlbrng1	40000	199200	1.2	4	5
5	Thermal problem	epb3	84617	463625	0.6	3	6
6	Optimization problem	cont-300	180895	988195	0.3	2	6

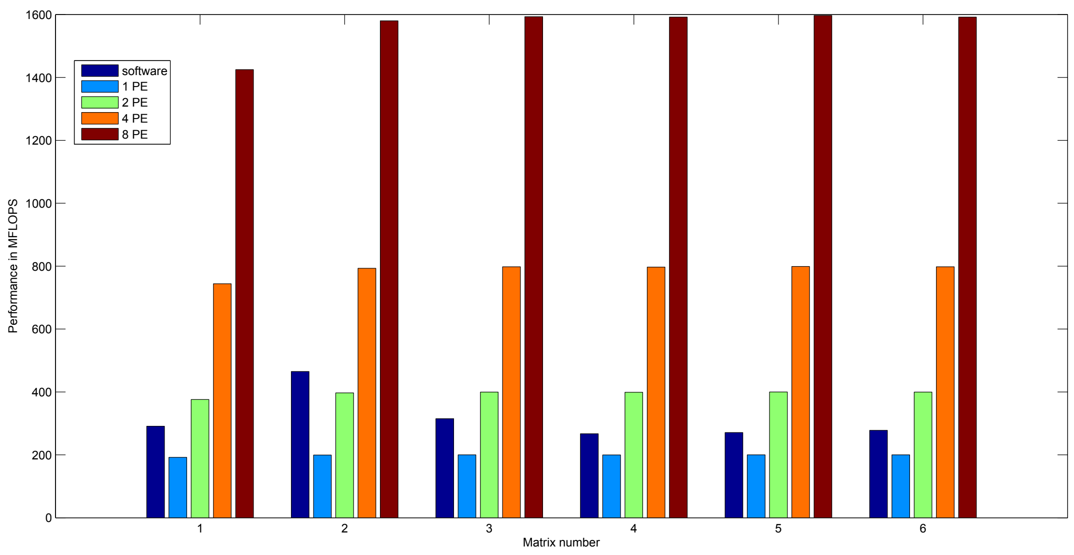

Fig. 7. Performance measured in software and hardware with overdimensioned bandwidth of 22.6 GB/s.

(CCUs) consisting of 1, 2, 4, and 8 PEs. Each PE is set to operate at 100 MHz and the peak performance per PE is 200 MFLOPS. From Figure 7, it can be observed that the proposed design is scalable and still reaches close to peak performance when the bandwidth is not a speed limiting factor.

Related Work: To our best knowledge, no related work exists on reconfigurable designs, capable to compute both dense and sparse matrix products in double precision floating-point arithmetic. In [1], [2], the starting point of our design, the dense matrix multiplications, are compared and analyzed to similar related works. In this section, we analyze and compare the performance of our sparse matrix vector product to the current available literature. Regarding related work on SMVM, Table III summarizes the peak performance, the actual performance in percentages of the peak performance, the required bandwidth, the area in terms of Virtex 4 slices and finally - the highest operating frequency.

In [6], the authors arrange PEs in a bidirectional ring to compute $\mathbf{y} = A^i\mathbf{b}$, with i being a natural number. The proposed design significantly saves I/O bandwidth due to local storage of the matrix and intermediate results. The local storage used for the matrix and for intermediate results is limited by the available On-Chip memory. In [7], a sparse matrix vector multiplication is performed as data from the matrix are converted to "pipelinable vertical nonzero stripes". The input vector is streamed in and the stripes are multiplied by the corresponding vector elements. The design performs well if the number of stripes is bounded, which is the case, for example, in finite element method (FEM) matrices.

The design in [8] splits the input matrices vertically, and computes for each vertical slice the temporary row results. Their presented hardware unit does not include the cost of these temporary results and the additions need to be performed on the CPU. The design is based on a tree, which sums up row results. Zeros need to be padded to this adder tree when the number of non-zeros within a row is not a multiple of the number of adders in the first stage. A special technique, called merging, is used to reduce this overhead. As the number of PE increases, the tree-based design requires complicated logic to reduce the hardware overhead and to increase performance. In Table IV, we selected randomly four matrices with different sizes from the 11 benchmarks in [8]. For each matrix, Table IV contains: its dimension N; the number of non-zeroes N_z; the sparsity γ; the sustained performance as a percentage of the peak performance for the real bandwidth of 22.6 GB/s; predicted performance for a limited bandwidth of 8 GB/s; and the performance of [8] for 8 GB/s. Both designs, ours and from [8], achieve equal maximal peak performance of 1600 MFLOPS at 8 GB/s, which is therefore assumed for the last two columns of Table IV. The figures suggest that our design achieves a better sustained performance, as percentage of the same peak performance, than [8] for all matrices considered.

TABLE III
PERFORMANCE RESULTS FOR THE SMVM OF RELATED WORK

Ref	Peak [MFLOPS]	Sustained [MLOPS] %	Sustained [MLOPS]	B [GB/s]	Area [slices]	f_{op} [MHz]
[6]	2240	33%-66%	739-148	N.A	N.A	140
[7]	1760	17%-86%	312-1520	8.0	**	110
[8]	2880	30%-75%	864-2160	14.4	*23856	165
[8]	1600	20%-79%	320-1264	8.0	*16613	165
This	1600	69%-98%	1104-1571	8.0	22700	100

* denotes area cost for reduction tree only.

** Area cost for this design equals 30% of logic resources and 40% internal RAM of Altera Stratix S80

TABLE IV
DESIGN COMPARISON TO [8]

matrix name	Application Area	N	N_z	γ	% P_{peak} [this]	% of P_{peak} [this]	% of P_{peak} [8]
				$[*10^{-4}]$	(B=22.6 GB/s)	($P_{PEAK} = 1.6\ GFLOPS$; B=8 GB/s)	
raefsky3	Fluid/structure	21200	148878	3	99	97	79
memplus	Circuit simulation	17758	99147	7	79	68	18
rdist1	Chemical processes	4134	94408	55	85	79	60
mcfe	Astrophysics	765	24382	417	78	74	59

The design in [9] does not assume any restrictions on the input format. The authors implement a reduction tree without the zero padding overhead which is easily scalable. In our design, we created slots which access the floating point core performance efficiently. With this approach, we obtain nearly peak performance for reasonably large matrices, without the need of a reduction tree. In [9], the autors compare their design to an Intel Pentium 4 machine and obtain relative speedups between 6 and 18.

All the related works, cited above, are based on simulations and analytical performance estimations. For our design, we provide actual results, measured from real hardware implementations and experiments. Moreover, we considered a high latency Off-Chip memory. The weakest point of our current design is the memory controller that feeds data into the PEs. In the future, we plan to design an efficient memory controller, similar to the one proposed in [9].

VI. CONCLUSION AND FUTURE WORK

We proposed a reconfigurable hardware design that computes both sparse and dense matrix vector multiplications performance efficiently. The design was based on multiple processing elements array, it was ANSI/IEEE-754 compliant and used double precision floating point arithmetic. The scalability and high performance efficiency of our proposal was evaluated experimentally on an Itanium-2 based SGI Altix 450 server, augmented with RASC blades. Experimental results suggested that our matrix-vector multiplier could provide higher sustained performance than related works. Moreover, we can safely conclude that the proposed approach, grounded on FPGA accelerated matrix vector multiplication, severely outperforms traditional GPP-based approaches. In the future, the design can be improved with more efficient memory controllers optimized towards higher bandwidth and performance.

ACKNOWLEDGMENTS

Special thanks to Yong Dou and Wouter van Ooijen for their earlier implementations of dense matrix multiplication, which have been used as starting designs for the implementation of the proposed SMVM. This work was partially supported by the Dutch Technology Foundation STW, applied science division of NWO, the Technology Program of the Dutch Ministry of Economic Affairs (project DCS.7533); and by the European Commission in the context of SARC IP #27648 (FP6) and hArtes IP #35143 (FP6).

REFERENCES

[1] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *FPGA*, H. Schmit and S. J. E. Wilton, Eds. ACM, 2005, pp. 86–95. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046204

[2] G. Kuzmanov and W. M. van Oijen, "Floating-point matrix multiplication in a polymorphic processor," in *In ICFPT International Conference Field-Programmable Technology*, 2007, pp. 249–252.

[3] S. G. Inc, "Reconfigurable application specific computer user's guide," 2008, http://techpubs.sgi.com/library/.

[4] T. A. Davis, "University of florida sparse matrix collection," *NA Digest*, vol. 92, 1994.

[5] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," *J. Phys.: Conf. Ser. 16*, pp. 521–530, 2005.

[6] M. DeLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *FPGA*, H. Schmit and S. J. E. Wilton, Eds. ACM, 2005, pp. 75–85. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046203

[7] Y. El-Kurdi, W. J. Gross, and D. Giannacopoulos, "Sparse matrix-vector multiplication for finite element method matrices on FPGAs," in *FCCM*. IEEE Computer Society, 2006, pp. 293–294. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/FCCM.2006.65

[8] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *FPGA*, H. Schmit and S. J. E. Wilton, Eds. ACM, 2005, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1046192.1046202

[9] J. Sun, G. D. Peterson, and O. O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," in *FCCM*, K. L. Pocek and D. A. Buell, Eds. IEEE Computer Society, 2007, pp. 349–352. [Online]. Available: http://dx.doi.org/10.1109/FCCM.2007.56