Reconfigurable Fixed Point Dense and Sparse Matrix-Vector Multiply/Add Unit

Humberto Calderón and Stamatis Vassiliadis
Computer Engineering Laboratory
Electrical Engineering Dept.,EEMCS,TU Delft, The Netherlands
email: {H.Calderon,S.Vassiliadis}@ewi.tudelft.nl
WWW home page: `http://ce.et.tudelft.nl`

Abstract

In this paper, we propose a reconfigurable hardware accelerator for fixed-point-matrix-vector-multiply/add operations, capable to work on dense and sparse matrices formats. The prototyped hardware unit accommodates 4 dense or sparse matrix inputs and performs computations in a space parallel design achieving 4 multiplications and up to 12 additions at 120 MHz over an xc2vp100-6 FPGA device, reaching a throughput of 1.9 GOPS. A total of 11 units can be integrated in the same FPGA chip, achieving a performance of 21 GOPS.

1. Introduction

It is well known that matrix-vector multiplication/addition is an important operation in both scientific and media applications. The main differences between the two environments is that media operates in short data formats (sub-words) and may extensively use fixed point rather than floating point arithmetic. There are numerous approaches for matrix (dense and sparse) hardware multiplication (see for example [10, 3, 7, 2]). All the approaches consider either sparse or dense computation, and mostly floating point operations [4, 2, 6]. In this paper we consider media like data formats and consider both sparse and dense computation performed by a single hardwired reconfigurable unit.

The remainder of this paper is organized as follows. Section 2, outlines the sparse matrix compression formats considered. Section 3, extensively describes the proposed unit. Section 4 describes the prototype results. The article is concluded in section 5 with some remarks and conclusions.

2 Sparse Matrix Compression Formats

We begin by presenting the Compressed Row Storage, the Block Based Compression Storage and the Hierarchical Sparse Matrix formats as background for our proposal. Those formats are described in Figure 1, which depicts pictorially a 32 x 32 matrix $\overrightarrow{A}$ with several nonzero elements represented in some cases by numbers and in general by an "**x**" as well as the dense vector $\overrightarrow{b}$ and the resulting vector $\overrightarrow{c}$. The division, shown in matrix $\overrightarrow{A}$, is used to best illustrate the sparse formats we consider.

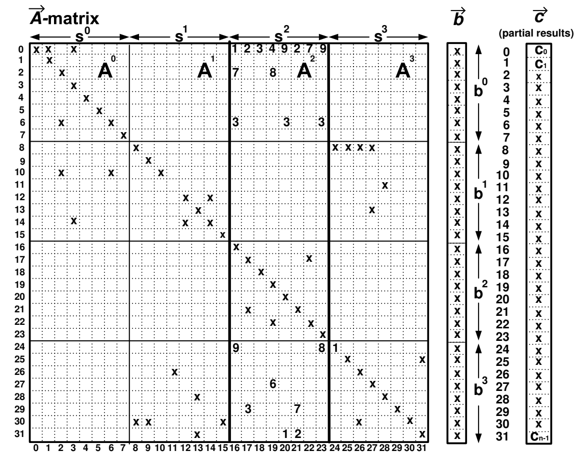

Figure 1. Sparse Matrix Representation

Compressed Row Storage (CRS) : The compression is achieved using a linear array A_N to store nonzero elements of matrix $\overrightarrow{A}$ in a row-wise way. A second linear array A_J, is used to store the column index for each nonzero element and a third one, A_I, holds the indices of the first element of a row in A_N [10]. An example of this encoding is presented in Figure 2.

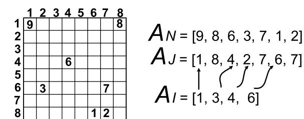

Figure 2. Compressed Row Storage Format

Block Based Compression Storage (BBCS) : In this compression format [7], the $n \times n$ $\overrightarrow{A}$ matrix is fractionated in $\lceil \frac{n}{s} \rceil$ Vertical Blocks (VBs), where s corresponds

to a processing section, as the 4 VBs presented in Figure 1. The format enables us to process $\overrightarrow{A^m}$ in sections for $m = 0, 1, 2, 3...\lceil \frac{n}{s} \rceil - 1$; in the same way the vector $\overrightarrow{b^m} = [b_{ms}, b_{ms+1}, ..., b_{ms+s-1}]$ can be fractionated where s remains the split section. Therefore, we are able to multiply each section A^m with the correspondent b^m section, segmenting in this way the processing. Figure 3 presents the BBCS format for nonzero values.

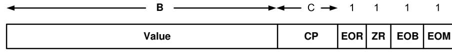

Figure 3. BBCS Format

As depicted in Figure 3, **Value** represents the B-bit nonzero element of $A_{i,j}$. **Column Position (CP)** is the column C-bit value of $A_{i,j}$ within a vertical block $\overrightarrow{A^m}$, for different blocks the value is j mod m. **End-of-Row-flag (EOR)** is asserted when the current nonzero value is the last element of the block, otherwise it is zero. **Zero-Row-flag (ZR)** is asserted when the current row has no nonzero value. **End-of-block-flag (EOB)** is true when the value is the last non zero value of the block. **End-of-Matrix-flag (EOM)** is asserted when the last nonzero element is found in the last section. Figure 4 offers an example of this format.

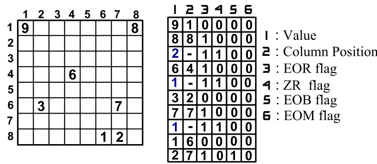

Figure 4. BBCS Format Example

Hierarchical Sparse Matrix format (HiSM): In this hierarchical format, a matrix $\overrightarrow{A}$ of $M \times N$ is partitioned in $\lceil \frac{M}{s} \rceil$ x $\lceil \frac{N}{s} \rceil$ squared sub-matrices. An example with 16 submatrices with $M = N = 32$, and section size $s = 8$ can be seen in Figure 1. The nonzero values in this format, as well as the positional information combined, are stored in a rowwise array in memory. The format uses two indexing levels; the first points to the squared blocks (Level 1), which contains nonzero elements and the second indexes the nonzero elements into the squared blocks (Level 0), using column and row representation [5]. A hierarchical format is illustrated in Figure 5.

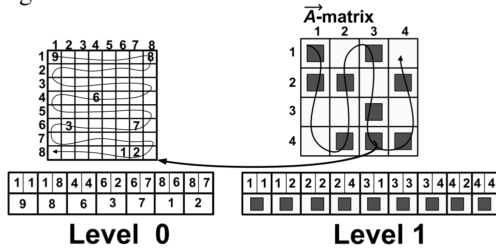

Figure 5. HiSM format example

Considering the three above-mentioned compression formats, it is appropriate to consider a common hardware to process those within the preprocessing stage. The CRS can be translated into a BBCS representation with the use of simple hardware which subtracts one from the values in the CRS A_I index, converting the row's pointer first element into a position associated with an EOR flag as used in BBCS format.[1] Furthermore, the low level indexing in the HiSM (Level 0) can be modified (HiSM-M), transforming into a squared section version of BBCS format, by adding an EOR flag to HiSM format. In summary we propose a hardware which works with the nonzero elements, a column pointer as well as an EOR flag for the processing of the above mentioned formats. Table 1 presents in short the aforementioned proposals.

Table 1. Comparison of Matrix-Vector Formats

Format	Value	Column Index	Row Index	EOR
Dense	$A_{(i,j)}$	$c_{(A)}$	$r_{(A)}$	EOR
CRS	$A_{(N)}$	$A_{(J)}$	-	extracted
BBCS	c_{A^S}	$c_{(A)}$		EOR
HiSM-M	$A^{s^2}_{(i,j)}$	$c_{(A^{s^2})}$	-	EOR

Therefore, a block A^m with $m = 0, 1, ..., \lceil \frac{n}{s} \rceil - 1$ of s columns where s is the processing section, the A^m are stored row-wise, in increasing row number order. Each A^m block will correspond to a section s of the b-vector, $\overrightarrow{b^m} = [b_{ms}, b_{ms+1}, ..., b_{ms+s-1}]$, and it is suitable to multiply each A^m with its corresponding $\overrightarrow{b^m}$ section of the b-vector without needing to reload any b-vector elements.

3 Dense-Sparse Matrix-Vector Multiply Unit

Dense matrix-vector multiplication (DMVM) and sparse matrix-vector multiplication (SMVM) are operations appropriate for parallel computing. Equation 1 suggests the possible concurrently computing of eight elements per cycle; the final result of the entire row is found through the addition of the partial calculated outcomes.

$$c_0 = \sum_{k=0}^{7} a_{0,k} b_k + \sum_{k=8}^{15} a_{0,k} b_k + ... + \sum_{k=n-8}^{n-1} a_{0,k} b_k \quad (1)$$

The hardware for the concurrently computing s entries, and particulary $s = 8$ (see equation 1) is designed. The hardware works on both dense and sparse matrix-vector multiply operations, processing nonzero elements of different rows and columns in a concurrent way. The proposed solution uses dynamic connecting channels to allocate the diverse data into four parallel execution units ($s = 4$) designed in a collapsed way [1], and optimized in terms of hardware use and delays. A simple control algorithm is used to enable the hardware unit to support $n \times s$ blocks. In this way block sizes like the ones described in Figure 1,

[1] Special attention has to be paid with the first element of the array.

as well as other cases[2], can be processed. The main tasks developed by a control allocation algorithm are:

1. Select one set of data (four data inputs) from a pool of two sets resident in local memory.

2. Route the correct dense vector multiplicand and a dense or sparse matrix multiplier in order to compute a multiplication into reduction trees.

3. Chose the necessary multi-operand units from a set of available resources for the addition of the 4 multiplication outcomes and a possible precalculated value. Depending on the incoming data, the algorithm has to allocate up to 4 multi-operand units.

4. Add the final partial value into the final (2/1) adders of a row when necessary, and update the registers for write back to memory

Memory Model: The memory model used to support the 4 stage pipeline proposed unit has the following register files that act like vector unit registers: **VR1:** 8 entry 16-bit $A^s_{(i,j)}$ elements register file. **CRC:** 8 entry 8-bit index information of the nonzero elements $A^s_{(i,j)}$ stored in VR1 register file. **VR2:** 8 entry 16-bit dense vector $\overrightarrow{b^s}$ register file. **VR3-O:** 8 entry 32-bit $\overrightarrow{c_i}$ write-back register file. **VR3-I:** 8 entries 32-bit $\overrightarrow{c_i}$ register file, used to load the partial results from memory.

3.1 The Pipeline Organization

The unit is segmented into the following stages:
1) Vector Read: In the first stage, the file registers are loaded from memory (i.e. embedded RAM), the file registers VR1, CRC and VR2. Figure 6 illustrates schematically the functionality.

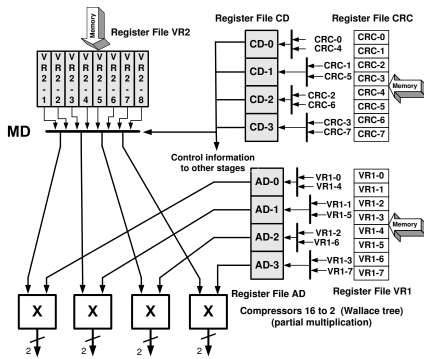

Figure 6. Vector Read and Multiply Reduction Trees

2) Multiply Reduction Trees: In the second stage the register files A-Buffer Data (AD) and Control Data (CD) are loaded. A toggle Flip-Flop (FF) controls which quadruple set of VR1 and CRC registers are routed with the eight 2-to-1 MUXs[3] to the registers AD and CD respectively, as depicted in Figure 6. Then the Column Positions (CPs) information held in CD registers controls the four 8-to-1 muxes (MD) to route the VR2 values. At this point, a partial multiply operation of the 4 matrix $\overrightarrow{A}$ elements and the corresponding dense vector $\overrightarrow{b}$ is carried out using 4 reduction trees as is shown in the aforementioned Figure 6, which does not specify the register-pipeline (R-Pipeline) that stores tree compressor results (SUMs and CARRYs). A Wallace type [8] tree without the final (2/1) addition is designed to compute the multiplication. With this organization we save the unnecessary repetition of the addition operation through the entire pipeline until the final stage, diminishing the processing delay.

3) Reduction Trees (Multiple-Addition): The third pipeline stage has four parallel reduction trees to compute the multiple-operand addition as shown in Figure 7.

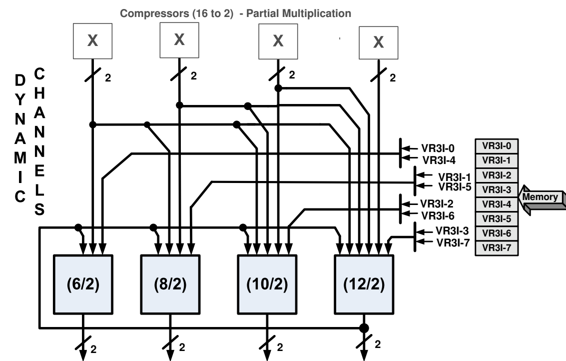

Figure 7. Reduction Trees (Multiple-Addition)

The multiple-operand addition hardware is intended to perform the following operations when necessary:

- The (6/2) compressor is used to carry out the partial addition of one multiply reduction tree outcome, the partial result of the computing row (12/2 outcome), and the partial result stored in VR3I.

- A reduction tree (8/2) receives two partial product outcomes, the feedback partial value and the corresponding VR3I.

- For adding three partial products, a feedback value and the VR3I data, a (10/2) reduction tree is used.

- The (12/2) reduction tree is expended to compute the partial addition of four incoming partial products, the

[2]Data can come from different rows, and in this proposal we present the processing of $2 \times s$ data.

[3]Multiplexors are represented by thick lines in the figure's.

Table 2. Resources Allocation Control Bits.

Term	W X Y Z	(6/2)	(8/2)	(10/2)	(12/2)	Compressor used.	$\overline{Z}$
0	0 0 0 0	000000	00000000	0000000000	FF0011111111	(12/2)	1
1	0 0 0 1	000000	00000000	0000000000	FF1111111111	(12/2)	0
2	0 0 1 0	000000	00000000	FF11111111	000000000011	(10/2),(12/2)	1
3	0 0 1 1	000000	00000000	FF11111111	001100000011	(10/2),(12/2)	0
4	0 1 0 0	000000	FF111111	0000000000	000000001111	(8/2),(12/2)	1
5	0 1 0 1	000000	FF111111	0000000000	001100001111	(8/2),(12/2)	0
6	0 1 1 0	000000	FF111111	0011000011	000000000011	(8/2),(10/2),(12/2)	1
7	0 1 1 1	000000	FF111111	0011000011	001100000011	(8/2),(10/2),(12/2)	0
8	1 0 0 0	FF1111	00000000	0000000000	000000111111	(6/2),(12/2)	1
9	1 0 0 1	FF1111	00000000	0000000000	001100111111	(6/2),(12/2)	0
10	1 0 1 0	FF1111	00000000	0011001111	000000000011	(6/2),(10/2),(12/2)	1
11	1 0 1 1	FF1111	00000000	0011001111	001100000011	(6/2),(10/2),(12/2)	0
12	1 1 0 0	FF1111	00111100	0000000000	000000001111	(6/2),(8/2),(12/2)	1
13	1 1 0 1	FF1111	00111100	0000000000	001100001111	(6/2),(8/2),(12/2)	0
14	1 1 1 0	FF1111	00110011	0011000011	000000000011	(6/2),(8/2),(10/2),(12/2)	1
15	1 1 1 1	FF1111	00110011	0011000011	001100000011	(6/2),(8/2),(10/2),(12/2)	0

VR3I value and feedback value for the processing of a row.

Resource allocation mechanism: Allocation of the multiply results into four parallel counters (6/2), (8/2), (10/2) and (12/2) is controlled by the end of row flags of the incoming set ($s = 4$). Table 2, show 4 variables: W, X, Y and Z, representing the EOR flags associated with the 4 nonzero $A_{(i,j)}$ elements, being W the first entry and Z the last entry of the set. The 16 different possible states are considered, as well as the reduction trees' control allocation bits, this information is a fundamental key for the entire unit operation. The final allocation strategy presented in the seventh column of table 2, is tuned to optimize the control complexity, reducing the logic control from 16 into 2 main states. Reduction is made upon consideration of the following 4 rules:

1. When processing an even state, Z = 0, (0, 2, 4, 6, 8, 10, 12 and 14 terms), it means that currently a row is still computing and that it will need to add its partial outcome to the rest of row values derived from the next quadruple. Partial computing is made by the (12/2) compressor. Therefore, the feedback data will always come from this multiple operands addition tree (see Figure 7). By this rule the feedback control is then simplified because we can rule out the rest of the resources as possible sources of a feedback data.

2. Allocation of resources follows the principle of using the biggest compressor when possible, taking always into account rule 1.

3. When possible, the nonzero element corresponding to W, X, Y and Z is distributed to compressors (6/2), (8/2), (10/2) and (12/2) respectively.

4. EOR flags enable VR3I values for proper accumulation of partial or final values.

Nomenclature used to represent allocation control bits in table 2, uses a '1' to enable the dynamic connecting channels[4] of data, and a '0' to disable the issuing of data into the reduction trees. Additionally, for any compressor, from left to right, we can determine the following meaning and behavior of the bits:

1. The first two bits are used to control the incoming result from (12/2) compression tree outcome. Using these bits we can compute n DMVM sets of 4 data with a supporting hardware of $s = 4$. Some entries have an 'F' symbol. We use symbol 'F' to show that the dynamic connecting channels of data will be either enabled or disabled to issue data into the reduction trees. This occurrence will depend on the Z value of the previous computed set. Therefore, when processing, e.g., the small matrix of section S^2, see Figure 1, partial result feedback is needed to obtain the row's final result, because we need to add the result of the first quadruple 1, 2, 3, 4 to quadruple 9, 2, 7, 9 values. Consequently, equations 2 and 3 controls the issue of the feedback data.

$$F4S_i = FBS4_i \cdot \overline{Z} \quad \forall\ 0 \leq i \leq 31 \tag{2}$$

$$F4C_i = FBC4_i \cdot \overline{Z} \quad \forall\ 0 \leq i \leq 31 \tag{3}$$

   where $FBS4$ and $FBC4$ are the SUM and CARRY outcomes of (12/2) compressor tree. Furthermore, these bits also control the feedback of a partial result when processing a sparse-matrix-vector-multiply operation.

2. The next two control bits, are used to control the issuing from VR3I when an EOR flag is found in any position of the incoming set of 4 inputs. The control functions for the SUM and CARRY values are described by equations 4 and 5. For example, Figure 1 shows that to compute row 24 in section A^3 we need to add to value '1' to the partial result of section A^2, completing

[4] Regards the path followed by a data coming from the multiply reduction tree outcome to the multiply-addition reduction tree.

thus (as suggested by equation 1) the computation of the row.

$$FVS_i = VR3I_i \cdot EOR \ \ \forall\, 0 \leq i \leq 31 \quad (4)$$
$$FVC_i = VR3I_i \cdot EOR \ \ \forall\, 0 \leq i \leq 31 \quad (5)$$

3. Finally, from left to right, the remainder of the bits controls the compressor outcomes' delivery. The next 2, 4, 6 and 8 bits enables or disables inputs to (6/2), (8/2), (10/2) and (12/2) compression trees respectively. The multiply reduction tree outcomes SUM and CARRY, are controlled in their issuing by the following equations:

$$S = SM_i \cdot SE \ \ \forall\, 0 \leq i \leq 31 \quad (6)$$
$$C = CM_i \cdot CE \ \ \forall\, 0 \leq i \leq 31 \quad (7)$$

where SM_i represents the SUM and CM_i the CARRY outcomes from the partial multiply unit, and SE and CE the control bits presented in table 2 for the issuing control. It is important to notice that a set of multiplexors is necessary to chose the correct state being processed. The input to those multiplexors comes from equations 2 to 7.

4) Final Addition and Result Update: The fourth stage reduces and finalizes computing of the multiple-addition trees' results. The 4 final adders outcomes are stored according to the column flags, and 4 de-multiplexors accomplish the task as schematized in Figure 8. The main memory should be updated from registers file VR3O.

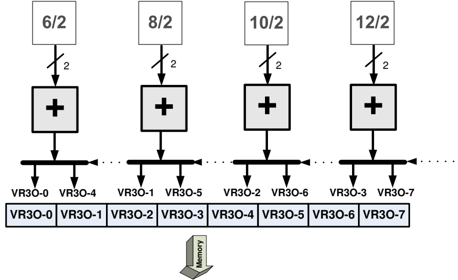

Figure 8. Final Addition

Scalability: Highly parallel algorithms can use several reconfigurable units, as the proposed above, to speed up the processing of dense and sparse matrix-vector-multiply operations. Such algorithms can schedule an inner loop instance as described by equation 1 to each unit. An improvement in performance can be achieved for DMVM, diminishing the size of the entire unit. This can be done by reconfiguring the pipeline, leaving the first stage as it is, employing a simple MUX instead of MD in the second stage, leaving only the (12/2) multiple-operand adder in the third stage, and finally, employing only one final (2/1) compressor instead of the four used in the whole unit.

4 Experimental Results

We have presented a general design intended to be used in ASICs as well as in reconfigurable technology. The proposed unit was described using VHDL, synthesized with the ISE 6.1i Xilinx environment [9], for a xc2vp100-6 FPGA device. The synthesis results indicate that our design utilizes 4255 slices, 1505 FFs and 6472 LUTs for the entire unit. Additionally, we synthesize each particular pipeline stage separately in order to find the contribution of each stage into the total number. We did this in respect of area and delay; those results are depicted in Table 3[5].

Table 3. Matrix-Vector Multiply/Add Unit.
(Pipelines stages: time delay - hardware use)

Xilinx XC2VP100	Partial Multiply	Multiple-Addition.	Final Add
Operand Width	16	32	32
Pipeline Stages	1	1	1
Area of Slices	1656	1595	332
LUT	3192	3174	640
Area in % design	37	35	7
Clock rate (-6) ns	8.3	6.0	6.3

From the above table, we conclude that the partial multiply stage is the bottleneck that limits frequency of the unit's operation. We notice that the partial multiplier unit without the final addition logic is equivalent in terms of time response to the built-in multipliers on the FPGA. Nevertheless, this stage, like others, can be pipelined in order to improve the unit's throughput. Concerning the hardware used and presented in table 3, the second stage consumes a 37% of the designed unit implementing the four multiply reduction trees and the routing logic. The third stage requires similar hardware to the previous one, a 35% is used implementing the four multiple-addition trees with their corresponding routing logic. The final 4 adders utilizes a 7% of hardware used, reaching those three stages a 79% of the total amount of hardware. Regarding the multiplexors used to route data into the second stage depicted in table 4, we should mention that the 1.3 ns introduced as an extra delay become a 16 % of the total stage delay.

Table 4. Routing Multiplexors - Second Stage.
(Routing hardware: - time delay and hardware use)

Xilinx XC2VP100	M-CD	M-AD	MD
Operand Width	8	16	16
Area of Slices	18	37	128
LUT	32	64	256
Clock rate (-6) ns	0.4	0.4	1.3

In reference to the area used to route the output of the 4 partial-multipliers, we found a maximum of 1.8 ns of extra

[5] For first stage see table 6. Independent synthesized pipeline units use more hardware when compared to the whole unit, due to synthesis introduces resource sharing.

Table 6. Matrix-Vector Multiply Unit.
(File Registers: time delay - hardware use)

Xilinx XC2VP100	VR1/VR2	CRC/Control	AD/CD	VR3I/VR30	R-Pipeline
Operand Width	16	16/8	8	32/32	64
Area of Slices	72/72	40/40	36/20	144/144	288
Flip-Flips	128/128	64/64	64/32	256/256	512
Area in % design	3.2	2.8	2.2	6.4	6.4
Clock rate (-6) ns	1.6	1.6	1.6	1.6	1.6

Control: refers to the control register that holds in the pipeline the Column and EOR information.

delay as is shown in table 5. The extra delay represents a 30% of the total stage delay. The information suggests a bigger use of resources compared with the previous stage. This occurs because of the operands size is increased due a partial multiply outcomes, it can be noted that we use SUM and CARRY representation in our pipeline.

Table 5. Allocation hardware -Third Stage.
(Dynamic channel hardware: time delay and hardware use)

Xilinx XC2VP100	G-6/2	G-8/2	G-10/2	G-12/2
Operand Width	32	32	32	32
Area of Slices	32	147	186	214
LUT	192	256	324	388
Clock rate (-6) ns	0.4	0.4	1.6	1.8

The remainder of used area, is employed to built the register files and pipeline registers as illustrated in table 6. Additionally, it was also synthesized the reduced unit presented in the sub-section scalability. Such reduced unit use 2507 slices and 3904 LUTs, representing 60% of the original unit. Also the frequency operation increments to 123 MHz. It is estimated that a 1/2 of the (3:2)counters used to construct the unit are shared by the scaled one. Therefore, the proposed unit can be considered as viable candidate block, suitable to work in a reconfigurable collapsed arithmetic unit.

5 Conclusions

We have presented a novel fixed point integer matrix-vector multiply unit suitable to work with sparse and dense matrices found in several media formats. The acceleration of the proposed unit is achieved using a new concurrent and scalable hardware capable of processing 4 matrix entries per cycle, carrying out 4 multiplications and up to 12 additions at 120 MHz using the 9% of the resources in a VIRTEX II PRO xc2vp100-6 FPGA device. We further presented pre-processing of different formats in order to convert the CRS and HiSM formats to an equivalent BBCS compression format, without incurring in significant area and time overhead. In all of the above representations we can extract the information of the last nonzero element of a row, and analyzing the data, we introduced an allocating algorithm which distributes efficiently the incoming data into parallel support hardware. A total of 11 units can be integrated in the same FPGA chip, achieving a performance of 21 GOPS. A 40% of hardware reduction is achieved using the reconfiguration capabilities of FPGA devices when operated as a scaled and modified unit.

References

[1] H. Calderón and S. Vassiliadis. Reconfigurable Multiple Operation Array. *Proceedings of the Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS05)*, pages 22–31, July 2005.

[2] M. deLorimeier and A. DeHon. Floating-Point Sparse Matrix-Vector Multiply for FPGAs. *ACM/SIGDA Thirteenth International Symposium on Field Programmable Gate Arrays (FPGA 2005)*, pages 75–85, February 2005.

[3] B. C. Lee, R. Vuduc, J. W. Demmel, and K. A. Yelick. Performance Models for Evaluation and Automatic Tuning of Symmetric Sparse Matrix-Vector Multiply. *Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 169–176, June 2004.

[4] E. Roesler and B. Nelson. Novel Optimizations for Hardware Floating-Point Units in a Modern FPGA Architecture. *In Proceedings of the 12th International Workshop on Field Programmable Logic and Application (FPL 2002)*, pages 637–646, August 2002.

[5] P. Stathis, S. Vassiliadis, and S. D. Cotofana. Hierarchical Sparse Matrix Storage Format for Vector Processors. *In Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, page 61a, April 2003.

[6] K. D. Underwood. FPGA vs. CPUs: Trends in Peak Floating-Point Performance. *In Proceedings of the ACM International Symposium on Field Programmable Gate Arrays (FPGA 2004)*, pages 171–180, February 2004.

[7] S. Vassiliadis, S. Cotofana, and P. Stathis. Block Based Compression Storage Expected Performance. *In Proceedings of the 14th International Conference on High Performance Computing Systems and Applications (HPC 2000)*, pages 389–406, June 2000.

[8] C. Wallace. A Suggestion for Fast Multiplier. *IEEE Transactions on Electronic Computers*, Vol. EC-13:, pages 14–17, February 1964.

[9] I. XILINX. The XILINX Software Manuals, XILINX 6.1i. *http://www.xilinx.com/support/sw_manuals/xilinx6/*, 2004.

[10] L. Zhuo and V. K. Prasanna. Sparse Matrix-Vector Multiplication on FPGAs. *ACM/SIGDA Thirteenth International Symposium on Field Programmable gate Arrays (FPGA 2005)*, pages 63–74, February 2005.