FPGA implementation of large-scale Matrix Inversion using single, double and custom floating-point precision

Janier Arias-García[1], Carlos H. Llanos[1], Mauricio Ayala-Rincón[2,3], Ricardo Pezzuol Jacobi[3]
Departments of [1]Mechanical Engineering, [2]Mathematics and [3]Computer Science
University of Brasilia
Brasília, D.F., Brazil, 70910-900
Email: {janier, llanos, ayala, jacobi}@unb.br

***Abstract*—This work presents an architecture to compute matrix inversions in a hardware reconfigurable FPGA using different floating-point representation precision: single, double and 40-bits. The architectural approach is divided into five principal parts, four modules and one unit, namely *Change Row Module*, *Pivo Module*, *Matrix Elimination Module*, *Normalization Module* and finally the *Gauss-Jordan Control-Circuit Unit*. This division allows the work with other smaller arithmetic units that are organized in order to maintain the accuracy of the results without the need to internally normalize and de-normalize the floating-point data. The implementation of the operations and the whole units take advantage of the resources available in the Virtex-5 FPGA. The error propagation and resource consumption of the implementation, specially the internal RAM memory blocks that are used, constitute improvements when compared with previous work of the authors and other more elaborated architectures whose implementations are significantly more complex than the current one and thus unsuitable for its application. The approach is validated by implementing benchmarks based on solutions in FPGA and software (e.g. Matlab) implemented previously.**

***Index Terms*—Matrix Inversion, Gauss-Jordan Elimination, Floating-Point Arithmetic**

I. Introduction

Matrices are the heart of different applications in several scientific fields. For instance, matrices are used to represent and solve electrical circuits, optimization, least squares problem, oscillatory systems, optics, image processing, stock market and quantum mechanics problems. A variety of algebraic techniques over matrices arise as adequate solution in different engineering problems but with one disadvantage: matrix algebra algorithmic implementations demand high computational costs. Because of this drawback, a major branch of numerical analysis is related to the development of efficient algorithms to perform matrix computations, which give rise to several open questions from both the theoretical and from the practical point of views.

The inversion is one of the most important and computationally expensive matrix operations for which several well-known numerical approaches are applied mostly in software platforms. Simple algorithmic methods, such as Gauss-Jordan (GJ) elimination, although of higher complexity ($O(n^3)$), are very important for developing practical architectural implementations. The hardware implementation of these algorithms is very interesting due to the fact that the von Neumann bottleneck can be avoided, mainly with regard to the writing/reading instructions to/from RAM memory. In the case of matrix operations that are implemented in hardware only write/read data operations need to be executed, without the complex decodification/execution steps related to the instruction executions, which are strongly related to the von Neumann model [1].

FPGAs are an increasingly used common platform to solve different computing problems that involve intensive matrix calculations necessary in several applications. Most of these applications involve algorithms that must deal with matrix data structures and their respective operations. Several matrix inversion algorithms have cubic complexity and, because of their algorithmic simplicity, specific reconfigurable hardware may provide an attractive practical solution. FPGAs are adequate platforms for these computationally intensive arithmetic calculations because they provide powerful computational architectural features: vast amount of programmable logic elements (LUTs), Block RAMs (BRAMs), embedded multipliers, shift register(SRLs), DSP blocks and Digital Clock Managers (DCMs) [1].

The major aspects to take into account when one works with matrix computations are those related to the fact that matrices are "ill-conditioned mathematical elements". Simple perturbations of the values of the matrix lead to a *large* modification in the results being computed. The meaning of *large* in this statement in fact depends on the application [2]. Because of this, the propagation of errors plays an important role in matrix computation and makes the adequate choice of the algorithms to be implemented in a hardware relevant for the successful and accurate solution of the problems under treatment.

In [1] the authors presented a hardware design for the GJ Elimination algorithm (using partial pivoting) that allows the user for setting the matrix range using Matlab VHDL automatic code generator. The architecture applies a special

978-1-4673-0186-2/12/$31.00 ©2012 IEEE

floating-point library (introduced in [3]) using single precision. Apart from performance and area results an experimental error analysis was achieved using Matlab results as a statistical estimator. The developed circuit involves the design of a *normalization module* at the end of the overall inversion process. An important aspect of this work was the internal memory RAM data access, which achieves read/write operations over words up to 1152 bit-width because of the implementations of the *Xilinx Block Memory LogiCore*. In this case the hardware implementation takes advantage of the parallelism capability of the selected FPGA device. For implementing this approach in the FPGA, following the standard stages of the algorithm (as described in Section III of this paper), an internal memory was used for storing the elements of the augmented matrix (that initially consists of the input matrix A and the identity matrix I), which represents a use of space of $2 \times N^2$.

In the current work the hardware implementation of matrix inversion (with partial pivoting) based on the GJ elimination algorithm presented in [1] has been improved for dealing with different floating point precision issues. In contrast to the approach in [1], where the augmented matrix is stored, in the current approach only the input matrix *A* is stored, reducing the use of space from $2 \times N^2$ to N^2 as proposed in [4]. Also, in the current work, the *Normalization Module* was implemented at the end of the *Forward Elimination Process*, in contrast to [1], where the *Normalization Module* was implemented at the end of the overall process with the disadvantage of the unsuitable use of memory resources. Additionally, the current approach pays special attention to the error propagation as long as the matrix size grows for maintaining the trade-off among the area cost, matrix range, performance and precision aspects. This is very important for defining the parameters of the floating-point representation, given that the user can apply these results for achieving a given target precision, saving FPGA hardware resources. Also, in [1] two pairs of floating-point units were used; namely, the *Multiplication Unit* and the *Addition Unit*. Each unit contains five multiplication and five addition floating-point operators, that are used to invert the augmented matrix $[AI]$, one pair of units operating over matrix *A* and the other one operating over matrix *I*. In contrast to the solution presented in [1], in the current approach an unique pair of floating-point units (with ten multiplication and ten addition operators for the *Multiplication* and the *Addition Units*, respectively) is dedicated to operate over the *A* matrix, that represents more resources working in a smaller matrix, which has a real positive impact in the circuit performance. As a final modification of the approach in [1], in this work, the floating point library developed by *Xilinx* has been used to perform arithmetic calculations.

The distinguishing features of the current approach are twofold:

- flexibility in the selection of bit-width which implies improvements on precision. The architecture parameterizes 32, 40 and 64 bit floating-point representation.
- Reduction of resource consumption allowing inversion in the space used by the input $N \times N$-matrix.

The results are supported by a detailed analysis of the error propagation, as long as the matrix-size grows, and its impact over the proposed architectural approach. The precision study has been developed using the Matlab results as an statistical estimator, which uses a 64 bit-width floating-point representation.

Section II presents a brief overview of related works covering FPGA implementation of matrix inversion; Section III presents the GJ algorithm and; Section IV, the proposed architecture. Results are given in Section V, before concluding and presenting future work in Section VI.

II. Related Works

Several articles reported matrix inversion implementations using different methods such as Cholesky, LU and Gauss-Jordan, [1], [5]–[8]. However, the vast majority of works report only fixed-point representation and architectures that cannot be extended to large matrices. Exceptions are the approaches presented in [1] and [9] where an architecture is proposed using floating-point representation with single and double precision, respectively. The architecture proposed in [9] uses external memory to meet the requirements. A drawback of that approach arises during the normalization process of the GJ method, which is executed in all components of each matrix row. This is due to the fact that such normalization is accomplished during the inversion process, which forces a multiplication for each line element also incrementing the error propagation inherent to the floating-point multiplications.

Also, in [9] a study in terms of precision issue (related to the overall matrix inversion process) was not presented. An analysis of the error propagation when the matrix size grows with single precision was presented in [1], which showed the importance of taking into account the propagation of the error when one works with ill-conditioned mathematical elements such as matrices.

The focus of the current work is on the design of a scalable architecture with small area requirements for matrices of different sizes with a performance at single, double and custom precision floating-point suitable for several applications. We have, therefore, chosen the GJ elimination method for which a detailed experimental study related to the precision problem was done. To the best of our knowledge there exists so far no similar research on matrix inversion showing an architecture capable of withstanding different matrix sizes as well as different floating-point precision and its error propagation analysis.

III. Gauss-Jordan Algorithm

GJ elimination is a method based on Gaussian elimination that puts zeros above and below each pivot giving in this way the inverse matrix. Let A be an $n \times n$ matrix, I the $n \times n$ identity matrix and X an $n \times n$ matrix of unknowns. The solution of the linear system $A \times X = I$ gives as result $X = A^{-1}$.

The GJ inversion method, firstly, applies Gaussian elimination to the augmented $n \times 2n$ matrix $\tilde{A} = [A\ I]$ giving as result

a matrix of the form $[U\ H]$ in which U is upper triangular. Secondly, $[U\ H]$ is transformed by applying elementary row operations into an augmented matrix of the form $[I\ K]$. Finally, extracting the right component of the augmented matrix one obtains K which is the matrix inverse: $A \times K = I$. The four main steps of the algorithm starting from row $i = 1$ and the augmented matrix $[A\ I]$, are presented below:

1) Partial pivoting: locate the pivot, that is the largest element in column i, considering the elements in rows i up to n. Then interchange row i with the row that contains the pivot.
2) Forward elimination: eliminate all the elements below the diagonal in column i, by subtracting each row below the pivot row by a multiple of the pivot row. Then, increment i and repeat from step 1 (if $i < n$), until the left matrix is upper triangular.
3) Back substitution: eliminate all the elements above the diagonal in column i, by subtracting each row above the pivot row by a multiple of the pivot row. Then, decrement i and repeat this step until all the elements above the diagonal are zero, till the left component becomes a diagonal matrix.
4) Normalization: divide each row i by the pivot, so that the diagonal element takes the value 1 and the remaining elements are scaled accordingly.Then left component is the identity matrix and the right becomes the inverse matrix.

IV. The GJ Matrix Inversion Reconfigurable Architecture

In order to represent the matrix components the IEEE-754 standard floating-point system with single, double and 40-bits precision was used. To accomplish this, we have used floating-point developed libraries developed by *Xilinx*. The internal RAM memory blocks available in the selected device (Virtex-5) are used for storing the components values of the matrix, having these as vectors that are extracted in a given order.

One important consideration must be taken into account in the development of this approach: *this architecture works with the hypothesis that all the given matrices are invertible*. For general purpose, this architecture is completely scalable because both the size of the matrix and the precision can be modified by changing only a few configuration parameters, allowing the inversion of matrices of different sizes, achieving the precision targeted by the users. Also, all the process start with a preloaded matrix stored in the internal memory RAM of the chosen device. This means that, an interface to send/receive the matrix is not used.

The general structure of the proposed architecture is depicted in Fig.1, which was presented in [1]. The hardware implementation of this architecture is divided in four different modules and one unit: *Pivot Finder Module*, *Change Row Module*, *Matrix Elimination Module*, *Normalization Module* and the *Gauss-Jordan control-circuit Unit*.

The flowchart *Matrix Elimination Module* is shown in Fig. 2. It can be observed that the same flowchart is proposed for

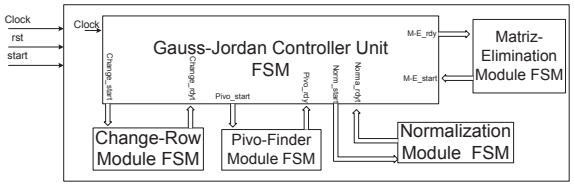

Fig. 1. General structure for the proposed architecture

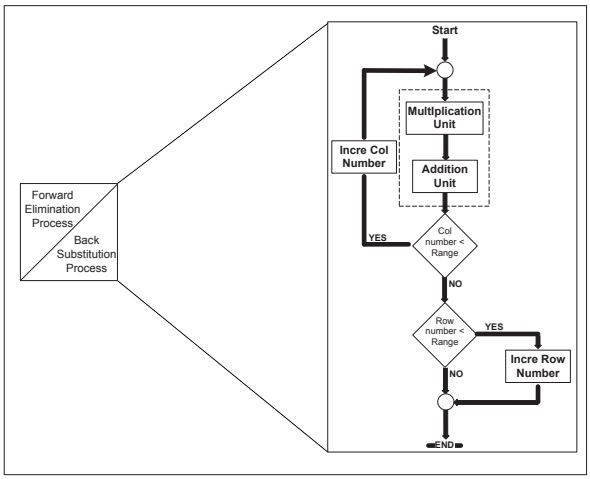

Fig. 2. Flowchart of the *Matrix Elimination Module*

both the *Forward Elimination Process* and the *Back Substitution Process* with the only difference related to the counters, which control the datapath through the different matrix components. In the current implementation, the *Forward Elimination Process* starts in the first position and ends in the $N - 1$ position. The *Back Substitution Process* starts in the second position and ends in the N position. These two processes are performed separately and in a sequential way (one begins after the other ends). The dotted square in the Fig. 2 represents the *Multiplication Unit* and the *Addition Unit*, respectively. These two units content ten *Multipliers Floating-Point Operators* and ten *Adder Floating-Point Operators* as shown in Fig.4.

The *Gauss-Jordan control-circuit Unit* has the task to control the overall access from/to the different modules as well as the different units (including also the writing/reading from/to the internal memory RAM). For the sake of the understanding, Fig.3 shows the *Control-Circuit Unit*, which comprises four modules (previously discussed) and the *Gauss-Jordan control-circuit Unit*. It can be observed the presence of the given matrix stored in the RAM internal memory of the selected FPGA device (*Virtex-5*). In [1], the architecture used two matrices of size $N \times N$, stored in the RAM internal memory, corresponding to the augmented matrix $[AI]$ of the Gauss-Jordan algorithm. Thus, a reduction in N^2 memory elements was achieved with this new configuration.

Also, the *Multiplication* and *Addition Units* had significant changes (see [1]), as shown in Fig.4. The intrinsic parallelism of the *FPGA* is again completely explored by these two units, because of the increasing of the number of arithmetic

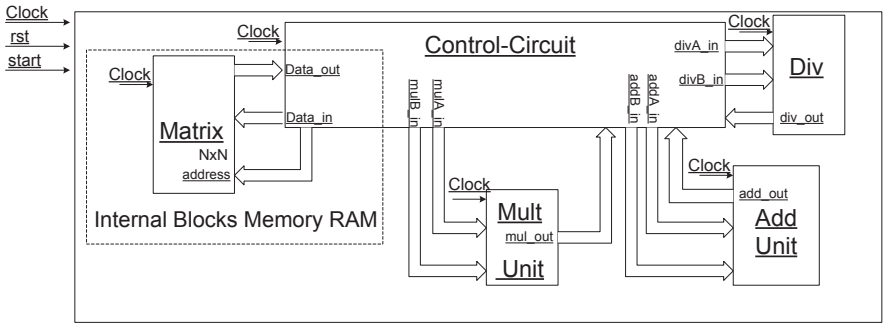

Fig. 3. The Control-Circuit Unit structure

operators now available for executing the GJ algorithm (from five *Multiplication* and *Adder floating-point operators* to ten *Multiplication* and *Adder floating-point operators*). The performance and the area optimization are important improvements of this new architecture, in order to achieve a large-scale matrix inversion implementation, keeping always in mind the behavioral of the error propagation.

The current approach uses the proposal presented in [4] that reduces by half the memory used resources in previous architectures and also reducing the number of the required floating-point units. To achieve this, the identity matrix used in the right side of the augmented matrix is not stored in the memory, because of the necessary information is contained in the left side of the augmented matrix. The filling behavior in the *Forward Elimination* and *Back Substitution Process* are well-known as well as the respective values related to the identity matrix (the right side of the augmented matrix). Due to the fact that these processes lead to triangular matrices, all the zero values (left side of the augmented matrix) are replaced by the values that would be stored in the right side. To accomplish the overall inversion process, the normalization process is performed with the *Forward Elimination*, just at the end of each loop. For instance, Fig. 5 depicts the *Forward Elimination Process* and The RAM Memory Organization, as explained above, only for one matrix column.

Taking advantage of this reduction and the parallelism achieved by the implementation of one row matrix stored in an address of the RAM memory as presented in [1], where the *Xilinx Block Memory LogiCore* was implemented to deal with the internal memory RAM of the device, the multiplication and addition units can be used to calculate only one side of the augmented matrix speeding up the process of matrix inversion, by exploring the intrinsic parallelism of the FPGA.

For dealing with different size, more that 36×36 (see [1]), the *Xilinx Block Memory LogiCore* was instantiated as necessary to work with the specific size.

V. Results

The HDL software used in this project was the Xilinx Integrated Software Environment (ISE) 10.1 and the selected FPGA device was a Virtex-5 XC5VLX330T. The mean error (ME) for single double and a custom precision (40-bits in this

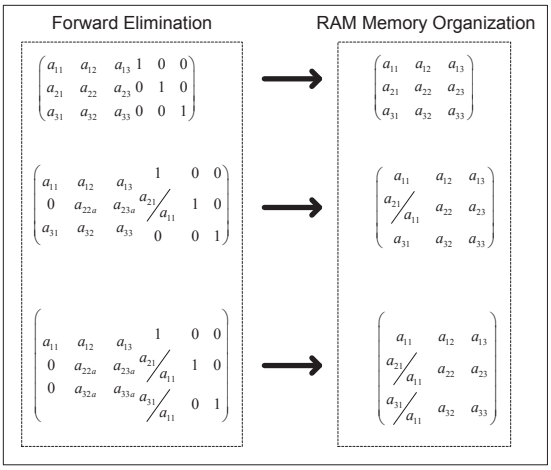

Fig. 5. *Forward Elimination Process* and Ram memory organization only for one matrix column

case, 10-bits for the exponent and 30-bits for the mantissa) is depicted in Fig. 6 for different square matrix sizes in the range from $n = 5$ to $n = 120$ using 1000 samples for each size n and using MatLab as a statistical comparator against our results. The data for each matrix size were generated using the Matlab command $A = double(rand(x, x))$; which defines a matrix with real numbers in double precision in the range [0,1].

The error propagation can be explained mainly by two factors:

1) When representing numbers with finite precision, not every number in the available range can be stored exactly. This is related to the finite precision in which the computers generally represent numbers. If a number cannot be represented exactly by the specified data type and scaling, a rounding method is used to cast the value to a representable number. Although precision decreases always in the rounding operation, the cost of the operation and the amount of bias that is introduced depend on the rounding method itself.
2) The number of necessary arithmetic operations to compute the inversion (namely multiplication, addition and

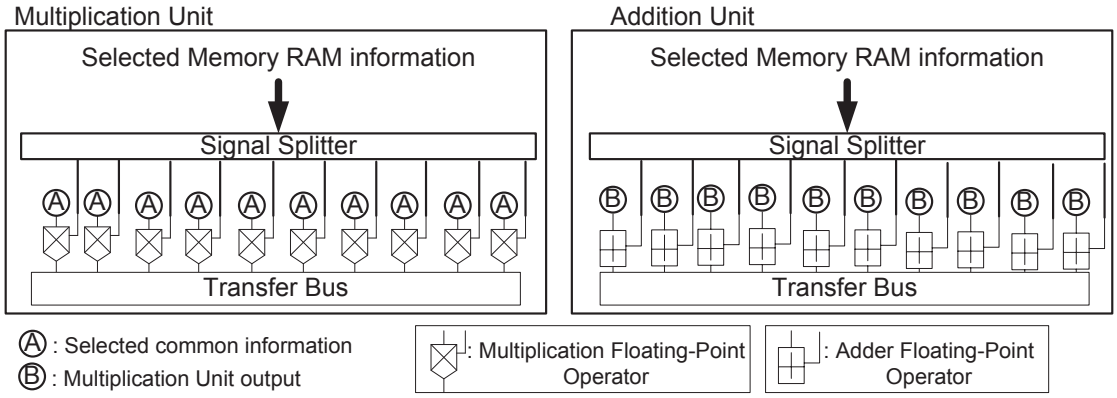

Fig. 4. Structure organization of the *Multiplication* and *Addition Units*

division) is a source of error and depends on the error produced by each arithmetic operator. Although, the error propagated by both addition and division is small, the fact that matrices are ill-conditioned mathematical objects, makes this propagation of error relevant for the analysis.

Fig. 6 also shows the importance of taking into account the necessary precision that the architecture must achieve (in the parameter specification of the desired architecture). This Fig. 6 shows which precision should be selected in order to invert a matrix of any size from 5 up to 120. The oscillations, as well as the tendency of the error to increase as long as the matrix size grows are due to the previous explained factors and, additionally, to the random method used to generate the input matrices. These results clearly shows the need to choose different precision according to the size of the matrices to be inverted, which is allowed in the proposed architecture.

In this context, it is important to stress that, as a consequence of the fact that any suitable implementation of matrix inversion should have the flexibility to allow the user to select the desired precision according to the input size of the matrices, the choice of an adequate FPGA must depend on a specific analysis in terms of the internal RAM resources as well as in terms of the number of DSP available.

Tables I, II and III summarize the resources occupied by each main modules and units design of the implementation after placing and routing.

TABLE I
SYNTHESIS RESULTS FOR *main modules*

Unit	LUTs	LUT-FF	Freq. [MHz]
Change Row Module	687	654	302.414
Pivo Finder Module	893	791	323.625
Normalization Module	1042	987	315.856
Matrix Elimination Module	1233	1146	304.257

This specific FPGA device has 192 DSP48E and 324 Block RAM Blocks distributed in memories of 36kb; thus, the resource occupation (see Tables I, II and III) show that this specific device can support matrices with large sizes and with different precisions, for achieving the matrix inversion operation for large-scale matrix.

TABLE II
SYNTHESIS RESULTS FOR THE *Multiplication Unit* FOR DIFFERENT PRECISION FORMAT

Precision	LUTs	LUT-FF	Freq. [MHz]	DSP48E
Single	656	331	260.484	30
Double	1178	598	211.433	130
40-bits	823	465	227.894	50

TABLE III
SYNTHESIS RESULTS FOR THE *Addition Unit* FOR DIFFERENT PRECISION FORMAT

Precision	LUTs	LUT-FF	Freq. [MHz]	DSP48E
Single	2379	2135	254.647	20
Double	4752	4293	200.4	30
40-bits	3058	2934	210.765	20

The total memory RAM Blocks occupied by the overall hardware implementation for the architecture depends on each precision. Thus, for the single, double and 40-bits precisions the respectively memory RAM Blocks used for storing a matrix of 120×120 are: 60, 120 and 120. Remember, that the selected FPGA device has 324 Blocks memory RAM organized as Blocks of 36kb and the *Xilinx Block Memory LogiCore* was implemented to use the internal memory RAM of the selected device.

VI. CONCLUSIONS AND FUTURE WORK

This paper presented an architecture for the computation of matrix inversion in a reconfigurable hardware based on FPGAs. The implementation allows selection of three different floating-point precisions: single, double and 40-bits, and it is divided in three main phases according to the GJ method: (a) Gaussian elimination (partial pivoting and forward elimination), (b) diagonalization (back substitution) and (c) normalization. These phases were developed by the five principal

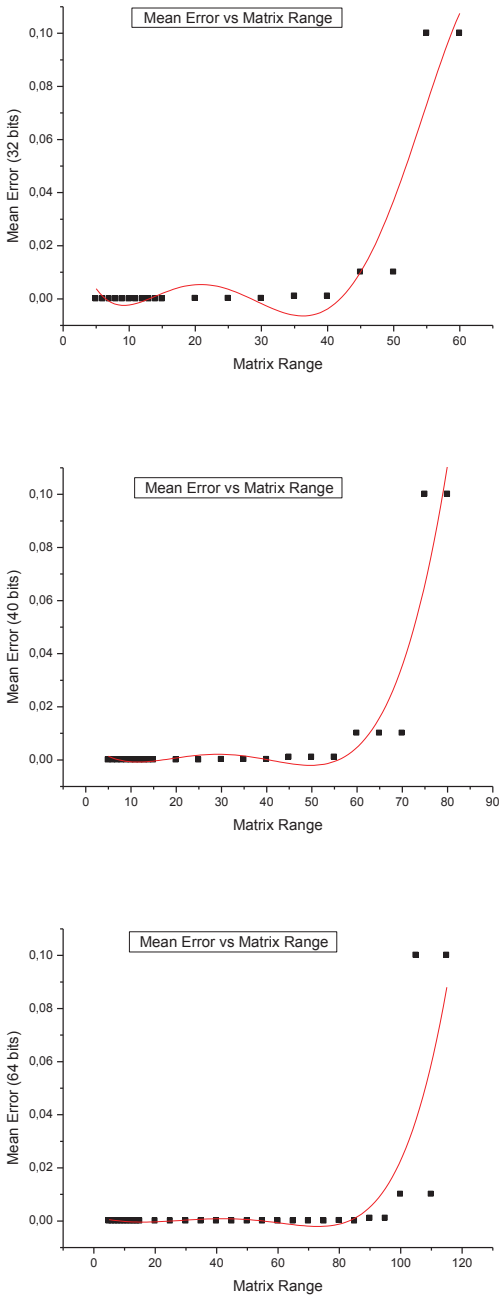

Fig. 6. Mean Error of the matrix inversion operation with different precision and with different matrix size. From top to bottom: The Mean Error using a). single precision. b). double precision. c). 40-bits precision.

architectural components: *Change Row Module*, *Pivot Module*, *Matrix Elimination Module*, *Normalization Module* and finally the *Gauss-Jordan Control-Circuit Unit*.

On the one side, an experimental analysis of the error propagation was developed using Matlab results as an statistical estimator. Results showed that the error tends to increase as long as the size of the matrix grows making evident that it is essential this analysis in order to estimate the propagation of error for practical and appropriate hardware implementations of matrix inversion operations.

On the other side, the FPGA resources consumption (namely, LUT, DSP-cells and frequency) showed that the Virtex5 used in the experiments can support matrices of large size due to the minimal amount of resources that were used for the architecture. This efficient use of resources allows for implementations on different portable systems.

In comparison with previous approaches, the current architecture significantly improves the consumption of resources and the ability to exploit in a more adequate capacity intrinsic parallelism of the FPGA, specially related to the internal RAM memory Blocks and the number of floating-point operators. In the future we will analyze the performance of this new proposed architecture against different platforms, GPUs and others FPGAs. Also, we will present implementations of the architecture in different engineering fields where it is needed such as robotics. Additionally, other architectures reusing some of the components of the current proposal can be developed being of particular interest the use of pipeline based arithmetic structures in order to speed up the performance of the arithmetic units.

VII. Acknowledgment

The authors would like to thanks CAPES Foundation and Xilinx University Program for the financial support of this work.

References

[1] J. Arias-Garcia, R. Pezzuol Jacobi, C. Llanos, and M. Ayala-Rincon, "A suitable fpga implementation of floating-point matrix inversion based on gauss-jordan elimination," in *Programmable Logic (SPL), 2011 VII Southern Conference on*, april 2011, pp. 263 –268.

[2] L. N. Trefethen and I. David Bau, *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[3] D. F. Sánchez, D. M. Muñoz, C. H. Llanos, and M. Ayala-Rincón, "Parameterizable floating-point library for arithmetic operations in fpgas," in *Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, ser. SBCCI '09. New York, NY, USA: ACM, 2009, pp. 40:1–40:6. [Online]. Available: http://doi.acm.org/10.1145/1601896.1601948

[4] G. de Matos and H. Neto, "Memory optimized architecture for efficient gauss-jordan matrix inversion," in *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, feb. 2007, pp. 33 –38.

[5] A. Burian, J. Takala, and M. Ylinen, "A fixed-point implementation of matrix inversion using cholesky decomposition," in *Micro-NanoMechatronics and Human Science, 2003 IEEE International Symposium on*, vol. 3, dec. 2003, pp. 1431 – 1434 Vol. 3.

[6] A. Happonen, O. Piirainen, and A. Burian, "Gsm channel estimator using a fixed-point matrix inversion algorithm," in *Signals, Circuits and Systems, 2005. ISSCS 2005. International Symposium on*, vol. 1, july 2005, pp. 119 – 122 Vol. 1.

[7] P. Salmela, A. Happonen, A. Burian, and J. Takala, "Several approaches to fixed-point implementation of matrix inversion," in *Signals, Circuits and Systems, 2005. ISSCS 2005. International Symposium on*, vol. 2, july 2005, pp. 497 – 500 Vol. 2.

[8] G. de Matos and H. Neto, "On reconfigurable architectures for efficient matrix inversion," in *Field Programmable Logic and Applications, 2006. FPL '06. International Conference on*, aug. 2006, pp. 1 –6.

[9] R. Duarte, H. Neto, and M. Vestias, "Double-precision gauss-jordan algorithm with partial pivoting on fpgas," in *Digital System Design, Architectures, Methods and Tools, 2009. DSD '09. 12th Euromicro Conference on*, aug. 2009, pp. 273 –280.