2009 12th Euromicro Conference on Digital System Design / Architectures, Methods and Tools

Double-precision Gauss-Jordan Algorithm with Partial Pivoting on FPGAs

Rui Duarte
INESC-ID/IST/UTL
Technical University of Lisbon, Portugal
rduarte@inesc-id.pt

Horácio Neto
INESC-ID/IST/UTL
Technical University of Lisbon, Portugal
hcn@inesc-id.pt

Mário Véstias
INESC-ID/ISEL/IPL
Polytechnic Institute of Lisbon, Portugal
mvestias@deetc.isel.ipl.pt

***Abstract*—**This work presents an architecture to compute matrix inversions in a reconfigurable digital system, benefiting from embedded processing elements present in FPGAs, and using double precision floating point representation.

The main module of this system is the processing component for the Gauss-Jordan elimination. This component consists of other smaller arithmetic units, organized in pipeline. These units maintain the accuracy in the results without the need to internally normalize and de-normalize the floating-point data.

The implementation of the operations takes advantage of the embedded processing elements available in the Virtex-5 FPGA. This implementation shows performance and resource consumption improvements when compared with "traditional" cascaded implementations of the floating point operators. Benchmarks are done with solutions implemented previously in FPGA and software, such as Matlab and Scilab.

***Keywords*-Matrix inversion; Pivoting; Gauss-Jordan; Floating-point; FPGA;**

I. Introduction

Reconfigurable computing has already shown significant performance for a number of computationally intensive matrix computations. As today's devices already provide a large density of coarse-grained computation resources, such as embedded multipliers, scientific computing on FPGAs is becoming an increasingly attractive alternative.

Matrix inversion computations are found in most scientific computing problems, such as problem optimization, least squares, oscillatory systems, electric circuits, image processing, stock market and cryptanalysis. Most matrix inversion and/or factorization algorithms have cubic computation complexity and therefore specific reconfigurable hardware solutions may provide significant benefits to a wide range of applications.

The Gauss-Jordan elimination algorithm with partial pivoting has been selected for implementation and performance analysis because it is a direct method for matrix inversion, simple, efficient, parallelizable and handles any type of (dense) square matrix, i.e. it is not limited to sparse, band and/or symetric matrices. Furthermore, this method has a similar computation flow and the same computational complexity than other common factorization methods and therefore its performance analysis is representative of a large set of linear algebra computations. Particularly, the LU factorization technique, that is generally considered more convenient for solving systems of linear equations if multiple systems with the same left-hand side need to be solved, can be seen as a specific case of gaussian elimination [1].

In practice and for general matrices, the Gaussian elimination (and LU factorization) algorithm is considered numerically stable if partial pivoting, as described in section 2, is performed [1]. Gaussian elimination with partial pivoting has proven to be an extremely reliable algorithm in practice and therefore is used on most numerical software applications. Because pivoting may degrade performance and difficults parallelization some hardware implementations of matrix factorization simply do not implement pivoting.

A number of research studies has already shown that the performance of reconfigurable systems on matrix inversion and/or factorization operations is already competitive with that of general purpose computers. Traditionally, as for most successful reconfigurable computing applications, the proposed reconfigurable hardware architectures have, at first, relied on fixed-point arithmetic. [2], [3] are two examples of matrix inversion algorithms using fixed-point operators. Then, as the FPGA single floating-point performance started to overtake that of general CPUs, a number of matrix inversion and/or factorization have been proposed. [4], [5], [6], [7] are some examples of architectures proposed using single floating-point operators.

As indicated in [8], nowadays the performance of FPGAs for double precision floating point linear algebra functions is already competitive with that of general purpose processors. This trend is confirmed by the performance results shown in [9], [10] for linear algebra operations, including LU factorization (without pivoting).

In this paper, we propose a reconfigurable hardware architecture for double-precision floating-point matrix inversion computations with partial pivoting, and analyse its potential to achieve improved performance when compared to other implementations such as general purpose processors (GPPs).

The present work improves on previously published research in that the operators are double-precision floating-point, in that partial pivoting is included in order to assure numerical stability for general matrices, and in that the implementation takes full advantage of the coarse-grain computation elements available in today's FPGAs (in particular for the division).

978-0-7695-3782-5/09 $25.00 © 2009 IEEE
DOI 10.1109/DSD.2009.199

The architecture is fully scalable so that the performance of the system scales directly with the number of parallel datapath units of the architecture, as long as the pipeline(s) throughput can be sustained by the available system memory bandwidth.

The results obtained indicate that the architecture performance can surpass that of general purpose processors, using only a small percentage of a state-of-the-art FPGA device.

II. The Gauss-Jordan Algorithm

The Gauss-Jordan elimination algorithm computes the inverse of a square matrix by manipulating the given matrix using a number of elementary row operations.

The Gauss-Jordan method is an extension of the Gaussian elimination algorithm in that at each step the pivot element is forced to 1 and all elements above the pivot (and not only the elements below) are set to 0.

The method starts by defining the *augmented matrix*, which is a matrix of size $N \times 2N$ that consists of matrix A, on the left, and of the identity matrix I, on the right. Then the augmented matrix is transformed by a sequence of elementary row operations, until the left side becomes the identity matrix and the right side becomes the inverse of matrix A:

$$[\mathbf{AI}] \Longrightarrow [\mathbf{IA^{-1}}] \tag{1}$$

The algorithm performs as many iterations as the number of rows, or columns, for a given matrix. For each iteration i, the method performs partial pivoting considering column i, normalizes the row i and eliminates all the elements above and below the diagonal in column i.

In the partial pivoting step the algorithm locates the pivot, that is the element with the largest absolute value in the column i, considering the elements in rows i up to N, and then swaps row i with the row that contains the pivot. In most implementations swapping rows is done by changing the pointers for each row instead of moving data between rows [1].

In the normalization step, the elements in the pivot row are divided by the pivot, so that the diagonal element takes the value 1 and the remaining elements are scaled accordingly:

$$a_{n+1}(p,j) = \frac{a_n(p,j)}{a_n(p,p)} \tag{2}$$

where $a_n(p,p)$ is the pivot, $a_n(p,j)$ is the element in the pivot row, and $a_{n+1}(p,j)$ is the new value for $a_n(p,j)$, after normalization.

In the elimination step, row operations are performed in the remaining rows so that the elements in the same column as the pivot are eliminated. Thus, the elements in the remaining rows are updated according to equation:

$$a_{n+1}(i,j) = a_n(i,j) - \frac{a_n(i,p)\ a_n(p,j)}{a_n(p,p)} \tag{3}$$

where $a_n(i,p)$ is the element in the pivot column and in the row being processed and $a_n(i,j)$ is an element of the row being processed. $a_{n+1}(i,j)$ is the updated value for $a_n(i,j)$ after the row operation.

III. Reconfigurable Architecture

The proposed architecture considers that the matrix elements are represented using the IEEE-754 standard floating-point representation system [11] with double precision. The implementation uses an internal representation that guarantees full double-precision and only performs pre/post processing, normalization and conversion from/to floating-point when the data is read or written to memory. This fused datapath approach follows the concept described in [12] to significantly reduce the computation resources without affecting the precision of the results.

The datapath is pipelined, so that the system consumes and produces data in every clock cycle. The architecture is able to perform pivoting without having pipeline stalls. As shown below, the pivot search and computation is computed at the same time that the matrix elements are processed and therefore does not hinder the system performance.

The hardware arithmetic operators take full advantage of available embedded multipliers (DSP blocks) to improve both performance and resource consumption. Specifically, the computation of the reciprocal, for the pivot normalization operations, improves on previous works by using an iterative multiplication algorithm that maps efficiently to FPGA embedded multipliers.

For memory optimization, the architecture uses the storage structure proposed in [5], that permits overlapping the left and right matrices and therefore requires only half of the augmented matrix to be stored. By reducing by 50% the memory usage, twice as large matrices may be stored, and only half memory input/output accesses are required.

The architecture can be easily scaled to different matrix sizes without having to be re-designed.

A. Operation Scheduling

Figure 1 shows the sequence of operations for the proposed architecture. The figure shows each loop unrolled into a set of hierarchical sub-loops, from left to right.

When the processing starts, the registers are initialized. After that, execution enters the main iteration loop. This loop executes N iterations, one for each pivot. For each iteration, inside this loop, the pivot is searched, the rows are (eventually) exchanged, the pivot row is normalized and the elimination steps are performed for the remaining rows.

The following sequence of actions is executed:

1) Find the 1^{st} pivot;
2) Change rows and compute pivot's reciprocal;
3) Normalize pivot row and reduce every row of the matrix and finds the new pivot;

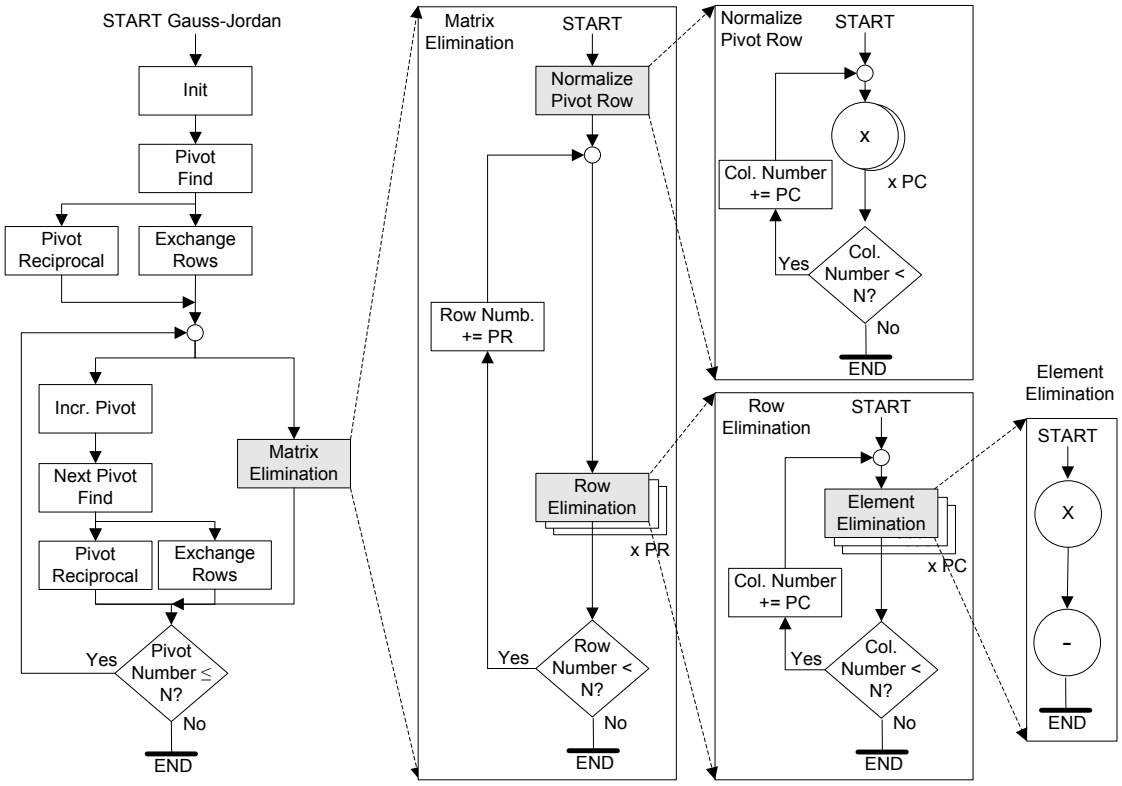

Figure 1. Sequencing graph for the proposed architecture.

4) If the column number is equal to pivot, then return to the pivot's column and increase row number;
5) If row number is equal to N, then go to first row and increment pivot number;
6) When pivot number is equal to N, then the algorithm has finished;
7) End of execution, no more actions performed.

In the pivot row normalization step each element is multiplied by the pivot's reciprocal, instead of directly dividing the element by the pivot, because multiplication uses less resources and takes less time than division. So, equation 2 is computed as

$$a_{n+1}(p,j) = a_n(p,j)\frac{1}{a_n(p,p)} \tag{4}$$

Then, the updated elements in the pivot row $a_{n+1}(p,j)$ are (re)used to process the elements in the remaining rows:

$$a_{n+1}(i,j) = a_n(i,j) - a_n(i,p)\ a_{n+1}(p,j) \tag{5}$$

As they are re-used for every row processing, the pivot reciprocal and the normalized pivot row are stored in caches to avoid fetching and storing them in (external) memory.

The parallel access to the matrix memory can be organized by rows, by columns or by matrix sub-blocks. If the matrix accesses are organized by sub-matrices, the partial pivoting pipelining can be seriously restricted [13], and therefore only organization by rows or by columns have been considered in this work.

Gauss-Jordan elimination operates differently if either configuration is used, because the normalized pivot row is used by the other columns in the elimination steps, and so it must be computed before. It is possible to process several columns at the same time, but they must be delayed by one column from the pivot column in order to work properly. Despite this, the performance is approximately the same for either configuration. For ease of implementation, the design proposed considers that the matrix accesses are organized by rows.

The search for the pivot, and the computation of its reciprocal are performed at the same time that the matrix elements are being processed. The datapth unit can therefore operate in pipeline without stalls if the elimination of the last row takes longer than the computation of the pivot reciprocal, such that

$$T_R < \frac{N}{P}(T_M + T_S), \tag{6}$$

where T_R, T_M, T_S are the clock cycles needed to compute reciprocation, multiplication and subtraction, respectively, and P is the number of parallel units available.

For typical values of T_R, T_M, T_S, the inequality (6) is true, even for small values of N. Note that, for example, if the time to compute the reciprocal is twice the time needed to compute the multiplication and the subtraction,

$$T_R \cong T_M + T_S \Rightarrow P < \frac{N}{2} \tag{7}$$

then the execution of the partial pivoting step is effectively hidden in the global algorithm schedule as long as the number of parallel units is lower than $\frac{N}{2}$.

Therefore, in practice, partial pivoting does not limit the system performance and, after the initial pivot selection and reciprocation, the datapath pipeline can receive and deliver new data in every clock cycle.

B. Data-Path

Figure 2 shows the data-path for the proposed architecture.

As shown, data input and output is done via a pair of registers. There are also registers to access each element in the matrix. The multiplexer between the input port and the pivot finder unit is used to switch between the memory contents, where the first pivot is found, and the output values of the Gauss-Jordan unit, where the new pivot candidates are computed.

After the pivot is found it is stored in a cache, so it can be used without external memory access. There is also a cache to store the pivot column values.

The row processing unit, shown inside the dashed box, is used to compute the pivot row normalization and the row elimination steps, depending on the configuration bit set by the control unit. To execute pivot row normalization the unit must compute only the multiplication, while for row elmination the unit computes a multiplication followed by a subtraction.

To guarantee that both operations have the same execution time to keep the pipeline stream continuous, one input of the subtractor is fed with the value 0 to execute only the multiplication or with the row element to eliminate in case of row elimination step. The routing of the correct data into the row processing unit is implemented with multiplexers at the input ports.

The dashed box indicates that this block can be replicated and operate concurrently. As shown below, the system throughput will scale directly with the number of parallel units, as long it can be sustained by the memory bandwidth.

An important point in the design of the floating-point dataflow has to do with the execution of each operation. When implemented independently of each other a sequence of processing steps must be followed: (1) the two operands are unpacked from floating-point representation, (2) the result is computed, (3) adjusted and rounded, and (4) finally data packed into floating-point representation. This sequence of floating-point processing steps are a major overhead because they are executed several (thousands) times.

An alternative is to consider only one pair of pre- and pos-processing steps for the complete datapath using an internal numbering system that maintains compatibility with the standard saving time and space. In this approach, error analysis is essential to estimate accuracy [14] and to guarantee that it is at least as accurate as if double precision floating-point representation was used with pre- and post-processing at all operations.

Figure 3 illustrates the simplified logical dataflow for the proposed Gauss-Jordan Elimination unit. At step 1 the reciprocal is calculated. Then at step 2 the dataflow executes a multiplication for pivot row normalization. Finally, the datapath eliminates the row.

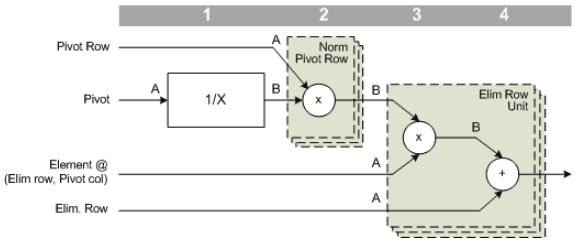

Figure 3. Simplified Gauss-Jordan Elimination Data-Flow.

Table I summarizes the number of bits required at the inputs and outputs of each unit of the datapath and their range of values. The number of bits for the integer part is separated from the decimal using a plus (+) sign.

Unit	Description	Signific. Bits (int. + dec.)	Range
1	reciprocator input	(1 + 52) 53 bits	[1, 2[
1	reciprocator output	(0 + 56) 56 bits	[$\frac{1}{2}$, 1[
2	multiplication 1 input A	(1 + 52) 53 bits	[1, 2[
2	multiplication 1 input B	(0 + 56) 56 bits	[$\frac{1}{2}$, 1[
2	multiplication 1 output	(1 + 56) 57 bits	[$\frac{1}{2}$, 2[
3	multiplication 2 input A	(1 + 52) 53 bits	[1, 2[
3	multiplication 2 input B	(1 + 56) 57 bits	[$\frac{1}{2}$, 2[
3	multiplication 2 output	(2 + 56) 58 bits	[$\frac{1}{2}$, 4[
4	subtractor input A	(1 + 52) 53 bits	[1, 2[
4	subtractor input B	(2 + 56) 58 bits	[$\frac{1}{2}$, 4[
4	subtractor output	(3 + 55) 58 bits	[-6, 6[

Table I
DATA-PATH INTERNAL REPRESENTATION.

In all cases, the number of bits indicated in the table are enough to guarantee that the post-processing determines the correct precision.

IV. PERFORMANCE ANALYSIS

The time to compute one full matrix inversion is given by:

$$T = T_{init} + T_{GJmatrix} + T_{flushOUT} \tag{8}$$

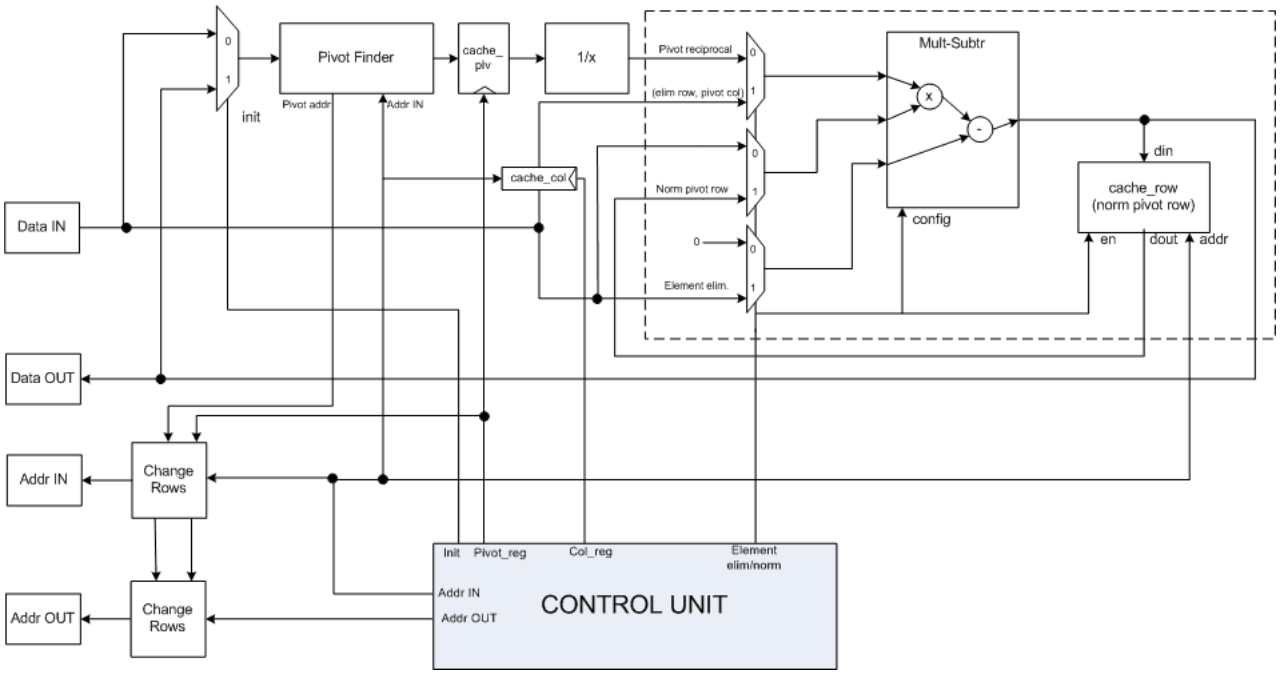

Figure 2. Data-path for the Gauss-Jordan processor.

where T_{init} is the time to start the row processing, $T_{GJmatrix}$ is the time to complete all the GJ row operations, and $T_{flushOUT}$ is the time to empty the pipeline.

To complete the GJ algorithm the N rows must be processed N times (one for each of the N pivots) and so

$$T_{GJmatrix} = N^2 \times T_{row}$$

where T_{row} is the time to process one row, or

$$T_{GJmatrix} = N^3 \times T_{MultSub}$$

where $T_{MultSub}$ is the time to process each element of the row, that is to compute one multiplication followed by one subtraction.

As referred, after the pipeline is full, the pivot selection and reciprocal computation are performed in parallel with the row processing, and so their execution time only adds to the overall computation time on the very first iteration. The time to initiate the first row processing is therefore the time for the floating-point pre-processing of the first element, T_{FPin}, plus the time to find the first pivot, T_{piv}, plus the time to compute the first pivot reciprocal, T_{recip}. The time to empty the pipeline is the time to compute the last row element operation, $T_{MultSub}$, plus the time to perform the floating-point post-processing, T_{FPout}.

$$T_{init} = T_{FPin} + T_{piv} + T_{recip} \quad (9)$$

$$T_{flushOUT} = T_{MultSub} + T_{FPout} \quad (10)$$

Equation (8) becomes then,

$$T = T_{FPin}+T_{piv}+T_{recip}+T_{GJmatrix}+T_{MultSub}+T_{FPout} \quad (11)$$

If P row processing units can work in parallel then

$$T = T_{FPin}+T_{piv}+T_{recip}+\frac{T_{GJmatrix}}{P}+T_{MultSub}+T_{FPout} \quad (12)$$

Considering that the datapath pipeline is designed to sustain a throughput of one operation per clock cycle (as long as memory bandwidth permits), given the clock frequency, the total computation times can be estimated from (12).

V. Arithmetic Units

This section briefly describes the implementation designs of the arithmetic units that process the significand values of the floating-point operators. These represent the most significant and complex part of the datapath. Signal and exponent processing are trivial, except for the subtractor unit where the exponent and signal affect the significant alignment and operation selection.

This implementation targeted Virtex-5 devices [15] and, as long as possible, all arithmetic units have been designed to take the most advantage of the embedded DSP blocks [16] available in these devices.

The floating point pre- and post-processing units used the designs presented in [17].

A. Subtractor

For subtraction, the operands significands must be aligned according to the difference of the two exponents. This

alignment is performed by right-shifting the significand of the operand with the lower exponent. This unit includes pre-shifting capability for both operands. The shifters are implemented as 6-level logarithmic barrel shifters grouped in 2 pipeline stages.

The operands are selectively added or subtracted, according to their signals, in order to compute their absolute difference so that the result significand can be directly provided as a sign-magnitude value. This conditional operator is implemented as 3 LUT-based adders/subtractors that perform the operations $a+b$, $a-b$ and $b-a$ in parallel and two $2-1$ multiplexers.

B. Multipliers

One multiplier unit is necessary for the Gauss-Jordan row processing Unit, and 6 multipliers of different sizes are necessary for the reciprocation unit.

The multiplier units do not require any pre-alignement of the operands significands and therefore have been directly implemented with fixed-multipliers (integer), with the appropriate operand and result widths, using the *Xilinx Core Generator* [15].

C. Gauss-Jordan Row Processing Unit

These functions are a composition of already defined elementary arithmetic units, more specifically the 58×58 multiplier and the subtractor. The pivot's reciprocal is computed outside this unit.

D. Reciprocation Unit

The pipelined reciprocation unit is based on the Goldschmidt's division algorithm using iterative multiplications. Previous work on this type of divider units has been presented in [18], [17].

This implementation was based on the design suggested in [19], with further operation simplifications that take advantage of the fact that the numerator of the divider is 1.

The design uses one BRAM to implement a $2^8 \times 8$ table-lookup initial approximation, followed by 4 unrolled iteration levels. The second and third levels require one pair of multipliers, while the first and the last level require only one.

The reciprocator increases representation by 3 least significant bits because of the division algorithm, Goldschmidt's convergence algorithm [18]. This algorithm was used because it suitable to use the coarse-grain arithmetic structures inside the FPGA and operate in pipeline.

VI. Results

The HDL software used in this project was Xilinx Integrated Software Environment (ISE) 10.1. The target FPGA was a Virtex-5 XC5VSX95T with 640 DSP48E blocks.

Table II summarizes the characteristics of the main datapath units after implementation, with timing values obtained after placing and routing.

Table III summarizes the resources occupied by the whole design and by the row processing unit (RPU), and show the percentage of the device occupied. RAM$\times$72 indicates the number of 72-bits memory positions available in the device BRAMs. Each 72-bits can store one floating-point element (in internal representation).

Unit	Total	%	RPU	%	Total device
LUT-FF	4047	7%	1074	7%	58800
LUTS	3924	7%	861	1%	58800
DSP48E	41	6%	12	2%	640
RAM$\times$72	variable				124928

Table III
Resources consumption.

The resources occupation shown in table III shows that this specific device can support up to 52 additional RPUs. Therefore, this is the maximum number of parallel units that can be included in one device of this type. Also, the total number of RAM$\times$72 positions, shows that a matrix of size 350×350 can be fully stored internally. (For a XC5VSX240T device these number increase to 87 RPUs and 510×510 matrices)

Compared to prior implementation this system doubles the operating frequency and uses less hardware.

For a system with several parallel units, an estimative of the resources can be obtained by multiplying the resources for a Gauss-Jordan unit by the number of parallel units. There is no penalty in latency and maximum operating frequency.

For comparison, the performance of software implementations running in SciLab [20] and MatLab [21] have been benchmarked. In both cases, the examples were executed in the same PC with a Pentium 4 processor at 3 GHz, 448 MB of RAM, running Microsoft Windows XP. Only one application was running during each test to avoid CPU utilization.

The following command line exemplifies the test for a 512×512 matrix:

```
x = 512; A = double(rand(x, x));
tic(); inv(A); toc()
```

Variable `x` is declared and initialized with the desired matrix size, 512 in the example. Variable A contains a random matrix with size $x \times x$. The `tic()` function starts the time counter, and the `toc()` function stops the counter and shows the elapsed time, in microseconds. In between the inversion of the matrix A is done with command `inv(A)`.

Table IV shows the results for several matrix sizes and the time taken for their computation, in seconds. These results are compared with the virtex-5 implemented system

Unit	Latency	FFs	LUTs	DSP48(E)	Freq. [MHz]
Subtractor	3	785	623	-	475
Multiplier	11	289	238	12	281
Reciprocation Unit	29	710	597	29	270
Row Processing Unit	14	1074	861	12	281
FP pre-processor	8	1032	1211	-	310
FP post-processor	12	297	349	-	320
Total	64	3736	3622	41	270

Table II
IMPLEMENTATION RESULTS FOR MAIN DATAPATH UNITS.

for $P = 1$ and $P = 4$ with a clock frequency of 250 MHz (achievable according to table III). As we can see, with a single processing unit the execution time of the hardware solution is worst, except for the first two dimensions (128 and 256). On the other side, with P = 4, we have better results. For example, for the biggest considered matrix (N = 5000), the time taken by the hardware solution is about 55% of the execution time of the SciLab solution.

To achieve the same execution time of SciLab for N = 5000, it is enough to use two row processing units in parallel (P = 2) and an operating frequency of 274 MHz. This is perfectly achievable with the actual solution, since the limiting unit is the reciprocator and this does not limit the frequency since its execution time is hidden by the row processing.

N	SciLab	MatLab	$\mathbf{P = 1}$	$\mathbf{P = 4}$
128	0.015	0.015	0.008	0.002
256	0.031	0.046	0.067	0.017
384	0.109	0.125	0.226	0.057
512	0.297	0.25	0.537	0.134
768	0.828	0.781	1.812	0.453
1024	1.859	1.813	4.295	1.074
1200	2.594	2.672	6.912	1.728
1400	4.062	4.250	10.976	2.744
1500	4.969	5.203	13.500	3.375
2000	11.125	12.438	32.000	8.000
2200	14.64	17.047	42.592	10.648
3000	35.047	44.141	108.000	27.000
5000	228.719	244.297	500.000	125.00

Table IV
TIME CONSUMED BY SOFTWARE IMPLEMENTATIONS *vs* PROPOSED ARCHITECTURE.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents a scalable pipelined architecture for computing matrix inversions in a reconfigurable system benefiting from embedded processing elements, using double precision floating point representation.

Results show that it is possible to obtain better results than those obtained with a state-of-the-art general purpose processor using only two units in parallel of a total of 87 parallel units supported by the largest device from the Virtex-5 family of FPGAs.

As stated above, the largest Virtex-5 FPGA have enough internal memory to store a matrix with a size up to 510×510. Therefore, in the future we intend to analyze the performance of the proposed architecture with the whole matrix stored in the internal memory of the FPGA.

REFERENCES

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996.

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

[3] F. Edman and V. Owall, "Implementation of a scalable matrix inversion architecture for triangular matrices," in *PIMRC 2003: 14th IEEE Proceedings on Personal, Indoor and Mobile Radio Communications*, vol. 3, Sept. 2003, pp. 2558–2562.

[4] G. Matos and H. Neto, "On reconfigurable architectures for efficient matrix inversion," in *FPL '06. International Conference on Field Programmable Logic and Applications*, Aug. 2006.

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

[6] K. Turkington, K. Masselos, G. Constantinides, and P. Leong, "Fpga based acceleration of the linpack benchmark: A high level code transformation approach," in *FPL '06: International Conference on Field Programmable Logic and Applications*, Aug. 2006.

[7] S. S. Demirsoy and M. Langhammer, "Cholesky decomposition using fused datapath synthesis," in *FPGA '09: Proceeding of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, 2009, pp. 241–244.

[8] K. Underwood and K. Hemmert, "Closing the gap: Cpu and fpga trends in sustainable floating-point blas performance," in *FCCM 2004: 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2004, pp. 219–228.

[9] Z. L. and V. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *Computers, IEEE Transactions on*, vol. 57, no. 8, pp. 1057–1071, Aug. 2008.

[10] L. Zhuo and V. Prasanna, "High-performance and parameterized matrix factorization on fpgas," in *FPL '06: International Conference on Field Programmable Logic and Applications*, Aug. 2006, pp. 1–6.

[11] American National Standards Institute/Institute of Electrical and Electronic Engineers, *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754-1985, 1985.

[12] M. Langhammer, "Floating point datapath synthesis for fpgas," in *FPL '08: International Conference on Field Programmable Logic and Applications*, September 2008, pp. 355–360.

[13] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed. Pearson. Addison-Wesley, 2003.

[14] D. E. Knuth, *The Art of Computer Programming Volumes 1-3 Boxed Set*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.

[15] Xilinx inc., Website, *http://www.xilinx.com/*.

[16] Xilinx, "Virtex-5 FPGA XtremeDSP Design Considerations User Guide - UG193 (v3.2)," Website, September 2008, *http://www.xilinx.com/bvdocs/userguides/ug193.pdf*.

[17] V. Silva, R. Duarte, M. Véstias, and H. Neto, "Multiplier-based double precision floating point divider according to the ieee-754 standard," in *Reconfigurable Computing: Architectures, Tools and Applications*, vol. 4943/2008. Springer Berlin / Heidelberg, 2008.

[18] R. Duarte, V. Silva, M. Véstias, and H. Neto, "Double precision floating point divider using iterative multiplications," in *REC '08: IV Jornadas de Sistemas Reconfiguráveis*, January 2008.

[19] B. Parhami, *Computer Arithmetic : Algorithms and Hardware Designs*. New-York: Oxford University Press, 2000.

[20] The French National Institue for Research in Computer Science and Control - INRIA, "Scilab," Website, *http://www.scilab.org/*.

[21] Mathworks, "Matlab," Website, *http://www.mathworks.com/*.