# On Hardware Solution of Dense Linear Systems via Gauss-Jordan Elimination

M. Tarek Ibn Ziad*, Yousra Alkabani*, and M. Watheq El-Kharashi†
*Computer and Systems Engineering Department, Ain Shams University, Cairo, Egypt
†Department of Electrical and Computer Engineering, University of Victoria, Victoria, Canada
Email: {mohamed.tarek, yousra.alkabani}@eng.asu.edu.eg, watheq@engr.uvic.ca

***Abstract*—Gauss-Jordan Elimination (GJE) is a popular method for solving systems of linear equations. Much work has been done to design high throughput, low cost, FPGA-based architectures for GJE. However, as the interest in energy efficient designs increases, power consumption becomes a prevalent metric that must be considered in any FPGA-based implementation. In this paper, we present a scalable architecture that can efficiently solve any generic system of linear equations using GJE with a single-precision floating-point accuracy and reasonable power and area overheads. Comparisons with two previous implementations show the efficiency of our design.**

**Keywords- dense linear systems, energy efficiency, floating-point, FPGA, Gauss-Jordan Elimination (GJE)**

## I. Introduction

Solving systems of linear equations has attracted scientists and researchers for a long time, as it represents the core operation in a wide variety of applications in fundamental sciences. For example, linear systems of equations arise in different physical problems, such as electromagnetic field calculations [1], [2].

There exist many different methods for solving systems of linear equations. Methods are divided into two main categories; direct methods and iterative ones. Direct methods operate on the matrix and solution vector until the solution can be computed. Iterative methods begin with an initial guess for the solution vector and refine it until convergence with sufficiently-small final errors [3]. Direct methods are typically used for dense matrices, which consist mainly of nonzero coefficients, as they would require large number of iterations and thus more memory accesses, if iterative methods are used. On the other hand, iterative methods are preferred for sparse matrices, which are matrices that have a lot of zero coefficients, to make use of this special structure in reducing the number of iterations.

One of the oldest, but still most widely used, direct methods for solving dense linear systems (DLS) is the Gauss-Jordan Elimination (GJE) method. It solves the system in a fixed number of steps and can handle coefficient matrices of any type, i.e., it is not limited to sparse, banded, or symmetric matrices. Moreover, GJE is used to compute matrix inversion when the given matrix is dense and unstructured [4]. Furthermore, other direct techniques like LU factorization can be seen as a specific case of GJE [5]. Although there already exist many implementations for the GJE on FPGAs [4], [6], [7], all of them only care about area and throughput. To the best of our knowledge, there are no energy-aware hardware implementations for solving DLS using the GJE algorithm.

978-1-4673-7788-1/15/$31.00 ©2015 IEEE

Nowadays, managing the system power consumption is not less important than meeting the performance specifications. It impacts cost and reliability of the entire system by reducing the supply requirements and cooling costs [8].

In this paper, we offer a generic hardware-based solution for any DLS, based on the GJE algorithm, designed with energy efficiency in mind. The key contributions of this paper include:

1) A pipelined architecture for solving DLS using the GJE algorithm with single-precision floating-point (FP) accuracy.
2) A portable and scalable design that can be downloaded on any FPGA from different vendors.
3) A detailed experimental analysis of the design logic utilization, time performance, and power consumption with a complete discussion about the design energy efficiency.
4) A performance comparison and evaluation of our proposed technique against similar hardware implementations with respect to time and area.

The remainder of this paper is organized as follows. Section II summarizes some of prior work. Section III provides an overview of the selected algorithm. Section IV details the hardware-based version of our GJE solver with all the optimization techniques. Experimental results and performance evaluation are described in Section V. Section VI concludes the work.

## II. Related Work

In this section, we survey some of prior work that targeted solving DLS on FPGAs using direct methods. We focus on the work that utilized the Gaussian Elimination (GE) algorithm and its extension, GJE algorithm. Then, we give a brief overview about other work related to energy efficient hardware implementations for matrix computations, as it is intensively used in DLS solvers.

There exist many work that offered hardware solutions for systems of linear equations. For example, Alonso and Lucio presented a parallel architecture for the solution of linear equations based on the Division Free Gaussian Elimination method with single and double FP representations [9]. Garcia *et al.* introduced a low cost single-precision architecture for the solution of linear equations based on the GE method [10]. Their implementation takes advantage of both the DSP blocks and the internal memory available in the Virtex-5 FPGA. Also, Garcia *et al.* in [11] introduced a hardware simulation flow for solving DLS with the GE method using Xilinx System

Generator Tool (XSG). They limited the usage of the FP division unit to execute only the reciprocal operation of a number, which enhances the overall performance. However, their architecture can handle limited number of equations with relatively high resource utilization.

On the other hand, many work implemented the GJE method, which could be seen as an extension for the GE method. Duarte *et al.* proposed an architecture for double-precision FP matrix inversion computations with partial pivoting [6]. They utilized their own implementation designs for the arithmetic units that process the FP values. Although their design is pipelined and can support several parallel units, their presented timing results are for system sizes up to 5000 equations, which cannot be synthesized on the used Virtex-5 device that is capable of handling up to 350 equations based on the shown resource consumption. In [4], Moussa *et al.* implemented a scalable design for computing matrix inversion using GJE with different FP precisions. They performed an experimental analysis for the error propagation using Matlab results as statistical estimators. However, no information are given about the required number of cycles or total execution time for a given matrix size. There are also many work that utilizes other direct methods for solving systems of equations, such as Cholesky [12] and LU factorization method [13], [14]. However, those implementations are out of the scope of this paper.

Unlike this paper, all the previous work mentioned above, did not offer any information about the overall power consumption of proposed designs. Benner *et al.* presented an evaluation of the impact of different software optimization techniques on the performance and energy efficiency of matrix inversion via GJE [15]. However, their investigation was limited to software implementations on general-purpose multicore processors only, which, unfortunately, consumes a large amount of power compared to FPGAs [15]. Recently, there exists few work, in the domain of FPGA-based computations, that take this criteria in mind. In [16], Matam *et al.* evaluated the energy efficiency of FP matrix multiplication (MM), which represents the core operation in many solvers. They estimated the peak energy efficiency of any MM implementation and compared their design against it. Energy-efficient FPGA implementations were also introduced for other kernels, such as signal processing [17], Fast Fourier Transform (FFT) [18], and Histogram Equalization [19]. To the best of authors knowledge, this work is the first to consider energy efficiency during designing a scalable hardware architecture for solving DLS.

## III. The Gauss-Jordan Elimination Algorithm

In this section, we provide an overview on the GJE algorithm used in this paper.

In general, the GJE method is an extension of the GE method in that, at each step the pivot element is forced to 1 and all elements above and below the pivot are set to 0. That is not like performing forward elimination, for the whole rows, followed by back substitution to obtain the solution vector in the GE method [20].

**Algorithm 1** The Gauss-Jordan Elimination Algorithm.

```
1:  procedure GJE(A, b)
2:      M = [A|b] /*Augmented matrix*/
3:      Pivot_row = 1
4:      for i = 1 to n do
5:          for j = i to n do /*Pivot location*/
6:              if |M(j,i)| > |M(i,i)| then
7:                  Pivot_row = j
8:              end if
9:          end for
10:         for j = i to n + 1 do /*Row exchange*/
11:             temp = M(i,j)
12:             M(i,j) = M(Pivot_row, j)
13:             M(Pivot_row, j) = temp
14:         end for
15:         Reciprocal = 1/M(i,i)
16:         for j = i to n+1 do /*Current row normalization*/
17:             M(i,j) = Reciprocal × M(i,j)
18:         end for
19:         for j = 1 to n do /*Row elimination*/
20:             if j ≠ i then
21:                 Pivot = M(j,i)
22:                 for k = i to n + 1 do
23:                     M(j,k) = M(j,k) − Pivot × M(i,k)
24:                 end for
25:             end if
26:         end for
27:     end for
28: end procedure
```

As shown in Algorithm 1, the GJE method starts by defining the augmented matrix, $M = [A|b]$, which is a matrix of size $n \times (n+1)$ that consists of coefficients matrix, $A$, on the left, and of the right hand side (RHS) vector, $b$, on the right. Then, $M$ is transformed by a sequence of elementary row operations, until the left side becomes the identity matrix, $I$, and the right side becomes the solution vector, $x$.

## IV. Hardware Implementation

In this section, we introduce the details of our proposed hardware design. First, we start describing the FP modules, which represent the building blocks of the architecture. Then, we describe the main modules in our design; memory, control unit, and ALU.

### A. Floating-point (FP) Modules

In our design, we utilize single-precision FP components, generated using FloPoCo, an open source generator of operators written in C++ [21]. This library takes operator specifications as an input and outputs synthesizable VHDL code, which can be easily downloaded to any FPGA board.

The FP format used in FloPoCo is identical to the one used in the IEEE-754 standard with the following difference. Zeroes, infinities, and Not a Number (NaN) are encoded as separate bits in FloPoCo, instead of being encoded as special exponent values in the IEEE-754 standard. Although,

this format helps saving a lot of decoding/encoding logic, it consumes two more bits when results have to be stored in memory. Thus, our data operand width would be 34 bits instead of 32 bits.

The used arithmetic cores are deeply pipelined in order to obtain the maximum performance. The total number of pipeline stages, needed for each module of our three FP modules, is given as follows. The subtraction module ($SUB$) is divided into 4 stages while the multiplication module ($MUL$) and the division module ($DIV$) are divided into 2 and 16 stages, respectively. These values are selected after various experiments to provide almost equal latencies for each pipeline stages, including memory access stages without the extra stages that would affect the overall performance.

### B. Main Design Modules

The architecture shown in Fig. 1 shows the three main components used in our design. The design is parameterized to cope with any size of the desired DLS. The two main parameters are the number of equations in the system, $n$, and the FP width, which equals 34 bits.

Here, we describe each module in detail along with all the implemented optimizations to get the highest performance of Algorithm 1.

Our memory unit is a three-port memory. It has two reading addresses associated with two output ports and one writing address associated with one input port. All ports are directly connected to the ALU to provide the rows of the augmented matrix, $M$, to operate on and store the result. Initially, the memory is loaded with $M$'s rows. So, each row of the memory contains the FP coefficients of the $A$ matrix concatenated with the corresponding RHS elements from $b$. Memory depth is equal to the dimension of $M$, which is $n$ equations. It is worth mentioning that the memory is provided with write/read enable signals. These control signals are only high while dealing with memory. Otherwise, the memory is disabled in order to reduce power consumption. It was proved that the power requirements of a block RAM is directly proportional to the amount of time it is enabled during [22]. Furthermore, our memory unit is built out of logic resources and does not depend on any specific RAM modules on FPGAs. That allows for higher portability of the design.

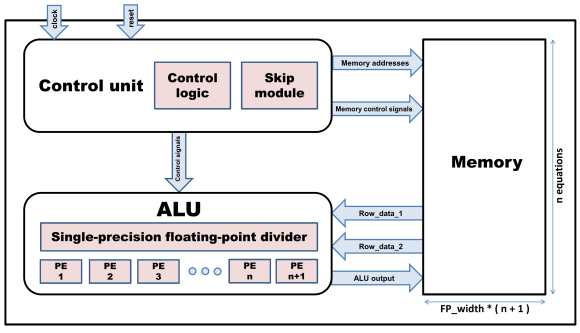


Fig. 1. Block diagram of main components for our GJE design.

ALU is the main processing component. It consists of only one single-precision FP divider and $n+1$ processing elements (PEs). For $n$ equations, we need $n$ PEs to deal with every coefficient and one additional PE for the RHS element. The architecture of each PE is shown in Fig. 2. It has only two FP modules, subtraction module ($SUB$), and multiplication module ($MUL$). All PEs are identical and are mainly used to perform the row elimination step in parallel. Although the normalization step mentioned in the main GJE algorithm requires the usage of $n+1$ dividers to normalize the whole row in one time, we use only one divider followed by $n+1$ multipliers. The divider is responsible only for computing the reciprocal of the pivot element. Then, the reciprocal is multiplied by all elements in the row that need to be eliminated. This modification is done to improve area and power efficiency as the FP divider modules require more area and consume more power than FP multipliers. The cost of doing so is adding two more cycles, for the FP multiplication operation, to the overall number of cycles.

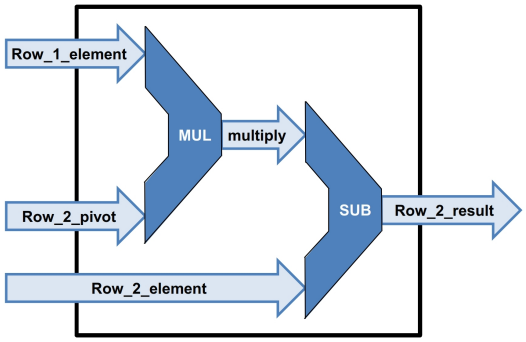


Fig. 2. Construction of one processing element (PE).

The control unit consists mainly of two submodules; the control logic, which is responsible for controlling the signals connected to memory and ALU, and the skip module. The control logic uses counters to determine the current addresses to be fetched from or written to the memory. It also generates control signals to handle the flow of data inside the ALU during each step of the four main steps, highlighted in Algorithm 1. As the elimination step could be done for each row without affecting other rows, there is no need to eliminate rows one after another. A read signal is issued at the start of the normalization step. When it finishes, and the normalized row is written to memory, a new row is mounted to the ALU in every clock cycle to make use of its pipelined structure. Hence, the total number of cycles needed for the row elimination step decreased from $k \times (n-1)$ to $(k-1)+(n-1)$, where $n$ is the number of rows and $k$ is the number of cycles used for memory accesses and PE operations.

The skip module is added to the control unit in order to enhance the performance of the main algorithm. At the beginning of the current row normalization step, it checks the pivot element after being read from memory. If the pivot element is already 1, the skip module generates a Skip_div signal to skip the division operation and go directly to the multiplication operations. This helps saving 14 cycles, the difference between

the 16 cycles utilized in the division operation and the 2 cycles needed for the check. In regular cases, this check is performed while the pivot element is being operated on by the division module, so no waste in clock cycles occurs.

Furthermore, the skip module is used to check for zeros in the first element of the eliminated row in the row elimination step. If this element is already 0, we do not need to complete this step with the $n+1$ multiplications, followed by $n+1$ subtractions. Finally, the control unit generates the halt signal when the calculations have been finished. This signal indicates that output data is written into memory and then stops all other modules to save power.

## V. Experiments and Analysis

This section describes the experimental environment for our design illustrated in Section IV. Then, it evaluates the design performance in terms of resource utilization, timing results, and power consumption. Finally, comparisons between our proposed design and other solutions mentioned in Section II are introduced to show the efficiency of our hardware-based solution.

### A. Experimental Setup

Our architecture modules were modeled using Verilog. The HDL software used here was the Xilinx Integrated Software Environment (ISE) 14.6 and the selected FPGA device was a Virtex-5 XC5VLX330T with a speed grade of $-2$. The data for our test cases were randomly generated using Matlab and converted to the FP notation before written in memory files. All test cases were successfully placed and routed on the selected FPGA.

### B. Resource Utilization

Table I shows the logic utilization and the maximum frequency of our proposed design using single-precision FP for different test cases. Logic utilization results are obtained from the place-and-route report, whereas the maximum frequency is obtained from the synthesis report. Each test case is defined using the number of equations that need to be solved. This number indicates the usage of $n+1$ PEs. The used FPGA provides a total number of 207,360 slice registers and the same number of slice LUTs. Moreover, 192 DSP48Es are available. The shown results indicate that the used board can support systems of equations of larger sizes due to the minimal amount of logic that were used for the architecture. It is worth mentioning that we did not optimize the code for a certain FPGA, while FPGA-specific optimizations might yield better area usage.

TABLE I. Hardware resource utilization for our single-precision FP design using different test cases.

| No. of equations | 5 | 10 | 15 | 20 | 25 |
|---|---|---|---|---|---|
| No. of slice registers | 3,979 | 6,313 | 9,197 | 11,737 | 14,262 |
| No. of slice LUTs | 6,480 | 11,022 | 17,312 | 23,494 | 32,773 |
| No. of DSP48Es | 24 | 44 | 64 | 84 | 104 |
| Maximum frequency | 198.230 | 179.880 | 149.897 | 141.606 | 140.183 |

As mentioned before in Section IV, our FP components are generated using FloPoCo. Although they operate on 34-bit operands, there would be no problem if the application, which is accelerated using our proposed solution, use only the IEEE-754 FP format. FloPoCo provides conversion operators from and to the IEEE-754 standard formats. These operators can be easily connected to the memory input and output ports with a low overhead. Table II shows the number of slice registers and LUTs used while implementing the FloPoCo conversion operators. The $Input\ converter$ module is used to convert a single-precision FP number from the IEEE-754 format (32 bits) to FloPoCo format (34 bits), whereas the $Output\ converter$ performs the reverse operation. The maximum combinational path delay is also given.

TABLE II. Resource utilization of FP FloPoCo converters.

| | Input converter | Output converter |
|---|---|---|
| No. of slice registers | 0 | 0 |
| No. of slice LUTS | 31 | 35 |
| No. of DSP48Es | 0 | 0 |
| Maximum frequency | 173 | 180 |

### C. Timing Results

Here, we first discuss the formula used to calculate the total number of cycles needed to complete the operations of Algorithm 1 on a DLS of any size. Then, timing results of our proposed design can be easily obtained through multiplying the calculated number of cycles by the maximum operating frequency.

In order to reach the final result represented by an identity matrix concatenated with the solution vector, we go through two time consuming steps from Algorithm 1. The first step is the current row normalization process that uses a division operation followed by $n+1$ multiplications. For normalizing one row, we need two clock cycles for memory access (read/write), 16 clock cycles for the $DIV$ operation, and two clock cycles for the $MUL$ operation, as stated in Subsection IV-A. Two more clock cycles are needed, as two pipelined stages are inserted between memory and FP modules and between FP $DIV$ and $MUL$ as FloPoCo pipelined operators do not directly buffer neither their inputs nor their outputs. Thus, it is the designer's responsibility to insert one more level of registers between two FloPoCo operators [21]. Complete normalization consumes $22 \times n$ clock cycles.

The second main step is the row elimination process. Eliminating one row consumes two cycles for memory accesses, two for $MUL$, four for $SUB$, and two for the inserted pipeline registers. As a result, we need $10 \times (n-1)$ clock cycles to complete the row elimination process for one row and that needs to be repeated $n$ times to cover the whole matrix. However, our pipeline fashion enables us of completing the row elimination process for one row in $(10-1)+(n-1)$ cycles only. So, the current implementation consumes $n \times (n+8)$ clock cycles for row elimination. Finally, the overall needed cycles are given by (1).

$$Number\ of\ needed\ cycles = n^2 + 30n \qquad (1)$$

Fig. 3 shows the timing results of our proposed design based on the total number of cycles provided in (1) and the maximum operating frequency. It should be noted that, results are obtained from the worst case scenarios when the skip module is not activated in order to show the efficiency of the design itself regardless of the input data.

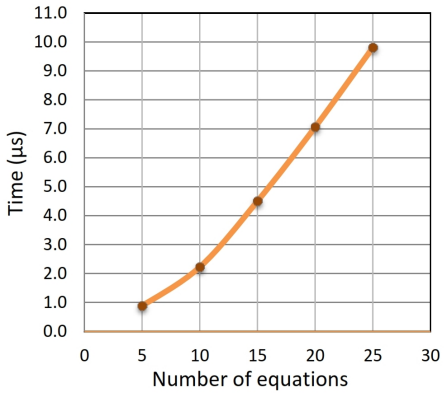


Fig. 3. Timing results of our GJE implementation using various test cases.

### D. Power Consumption and Energy Efficiency

The post implementation tool used in this work is the Xilinx XPower Analyzer. It provides the designers with an accurate view of the power consumption based on the exact resource utilization information extracted from the FPGA design implementation reports. Fig. 4 shows the power consumption of our design for different test cases. The VCD file (value change dump file) is used as input to the XPower Analyzer to produce accurate power dissipation estimation. The operating frequency of all the evaluated designs were set to 140 MHz. The total power is the sum of dynamic power and leakage power. Dynamic power includes the power dissipated in clocks, logic, signals, DSPs, and IOs. It can be seen from Fig. 4 that the power consumption is linearly proportional to the number of equations and processing elements. Also, the dynamic power is small compared to the leakage one.

To validate the energy efficiency of our implementation, we followed the same approach introduced in [16] by defining

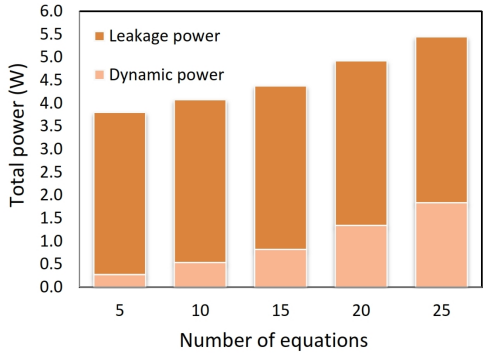


Fig. 4. Power consumption of our GJE implementation using various test cases.

energy efficiency as the number of operations per unit energy consumed. Energy consumed by the design equals time taken by the design multiplied by the power dissipation. The unit used is the number of floating-point operations per second per watt, which is equivalent to number of floating-point operations per Joule (FLOPs/Joule). In our case, timing and power results are obtained from Fig. 3 and Fig. 4, respectively. Number of operations are given by (2).

$$Number\ of\ operations = [1\ DIV + 2(n+1)\ MUL + (n+1)\ SUB] \times n = 3n^2 + 4n \quad (2)$$

Fig. 5 shows the energy efficiency of our single-precision GJE design for various configurations. The performance stabilizes as the number of equations, and thus number of PEs, increase since the total time and power increase linearly with the quadratic increase in number of operations in (2).

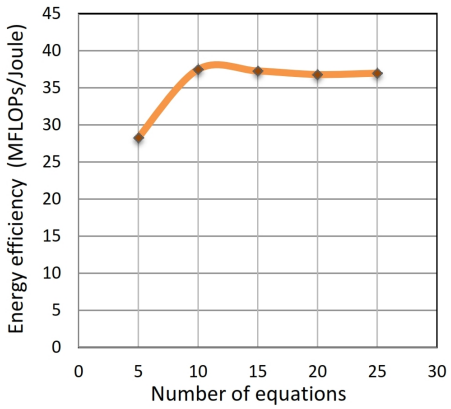


Fig. 5. Energy efficiency of our GJE implementation using various test cases.

### E. Performance Evaluation

Our results are compared with previous work. Table III shows the time used by the FPGA to solve systems of linear equations using a 100 MHz clock for our proposed design and the GE implementation introduced in [10]. We have estimated some of the performance numbers for the work in [10] from provided graphs as exact numbers are not given. Our design achieves a speed-up of 13.89x in case of 30 equations. Regarding the logic utilization, the authors in [10] gave the total resources used by the whole architecture for a linear system of 10 equations only. In this case, logic utilization is almost the same, while a speed-up of 20x is obtained.

TABLE III. TIMING COMPARISON WITH THE DESIGN IN [10].

| Number of equations | Execution time ($\mu s$) | | Speed-up |
|---|---|---|---|
| | Our design | GE's [10] | |
| 5 | 1.75 | 40 | 22.86 |
| 10 | 4.00 | 80 | 20.00 |
| 20 | 10.00 | 160 | 16.00 |
| 30 | 18.00 | 250 | 13.89 |

TABLE IV. RESOURCES AND TIMING COMPARISONS AGAINST THE DESIGN IN [4] USING VIRTEX-5 LX50T FPGA.

| Matrix size | Our design | | GJE design [4] | |
|---|---|---|---|---|
| | 4 | 8 | 4 | 8 |
| No. of slice registers | 3823 | 5369 | 3048 | 4321 |
| No. of slice LUTs | 5820 | 9184 | 4476 | 7396 |
| No. of DSP48Es | 20 | 36 | 10 | 10 |
| Number of cycles | 136 | 304 | 608 | 1204 |
| Maximum frequency (MHz) | 196.142 | 179.277 | 263.116 | 263.116 |
| Execution time ($\mu$s) | 0.693 | 1.696 | 2.311 | 4.576 |

Since the reported results in [4] use another version of the Virtex architecture, we have also synthesized our design on the same architecture, Xilinx Virtex-5 LX50T. Table IV shows the resource utilization and timing comparisons between the GJE design in [4] and ours. Although, the maximum frequency, reported in [4], is better than ours, the number of needed cycles in [4] scales exponentially, so the overall timing results are better in case of our design. It is worth mentioning that the work in [4] computes matrix inversion, not the solution of DLS like ours. However, we both use the same procedure of GJE. The only difference would be in the size of block RAMs used. Also, the architecture proposed in [4] uses complex multipliers and dividers, which increase the amount of resources used. Finally, direct comparisons against other energy-efficient implementations in Section II are not applicable as they neither implement the same algorithm nor target the solution of DLS. Other work, that actually does so, does not offer power analysis of presented implementations.

## VI. CONCLUSION

In this paper, we presented a single-precision FP architecture for solving generic DLS of equations using the GJE algorithm. Our architecture is designed with energy efficiency in mind. Using an open source library for generating FP modules gives our design a higher degree of portability. We implemented our design on a Virtex-5 FPGA and detailed experimental results were discussed to show time, area, and power costs. We also presented a quantitative comparison between the performance of our hardware prototype and some of the previous hardware solutions. The experimental results show that our design is more time-efficient with a reasonable area and power consumption. Directions for future work include investigating the trade-off between obtaining higher solution accuracy by using double and quadruple FP precision and maintaining area and energy efficiency.

## REFERENCES


[1] J. R. Poirier, P. Borderies, E. Gimonet, R. Mittra, and V. Varadarajan. Efficient solution of dense linear system of equations arising in the investigation of electromagnetic scattering by truncated periodic structures. In *IEEE Antennas and Propagation Society International Symposium Digest*, volume 1, pages 52–55, July 1997.

[2] M. Tarek Ibn Ziad, M. Hossam, M. A. Masoud, M. Nagy, H. A. Adel, Y. Alkabani, M. W. El-Kharashi, K. Salah, and M. AbdelSalam. Finite element emulation-based solver for electromagnetic computations. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1434–1437, Lisbon, Portugal, May 2015.

[3] W. Hager. *Applied Numerical Linear Algebra*. Prentice-Hall, Englewood Cliffs, NJ, USA, 1988.

[4] S. Moussa, A. M. A. Razik, A. O. Dahmane, and H. Hamam. FPGA implementation of floating-point complex matrix inversion based on GAUSS-JORDAN elimination. In *26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, Regina, Canada, May 2013.

[5] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, third edition, 1996.

[6] R. Duarte, H. Neto, and M. Vestias. Double-precision Gauss-Jordan algorithm with partial pivoting on FPGAs. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD)*, pages 273–280, Patras, Greece, August 2009.

[7] B. Zhang, G. GU, L. SUN, and Y. Wu. Floating-point FPGA Gaussian elimination in reconfigurable computing systems. *Chinese Journal of Electronics*, 20(1), 2011.

[8] Xilinx Inc. Power consumption in 65 nm FPGAs. A White Paper WP246 (v1.2), Available from: http://www.xilinx.com/support/documentation/white_papers/wp246.pdf. (accessed July 2015), Feb 2007.

[9] R. M. Alonso and D. T. Lucio. Parallel architecture for the solution of linear equation systems implemented in FPGA. In *Electronics, Robotics and Automotive Mechanics Conference (CERMA)*, pages 275–280, Cuernavaca, Mexico, September 2009.

[10] J. A. Garcia, C. H. Llanos, M. A. Rincon, and R. P. Jacobi. A fast and low cost architecture developed in FPGAs for solving systems of linear equations. In *2012 IEEE Third Latin American Symposium on Circuits and Systems (LASCAS)*, Playa del Carmen, Mexico, February 2012.

[11] J. A. Garcia, A. Braga, C. H. Llanos, M. A. Rincon, R. P. Jacobi, and A. Foltran. FPGA HIL simulation of a linear system block for strongly coupled system applications. In *IEEE International Conference on Industrial Technology (ICIT)*, pages 1017–1022, Cape Town, South Africa, February 2013.

[12] S. G. Haridas and S. G. Ziavras. FPGA implementation of a cholesky algorithm for a shared-memory multiprocessor architecture. *Parallel Algorithms Applications*, 19(4):211–226, 2004.

[13] W. Zhang, V. Betz, and J. Rose. Portable and scalable FPGA-based acceleration of a direct linear system solver. *ACM Transactions on Reconfigurable Technology and Systems*, 5(1), March 2012.

[14] G. Wu, Y. Dou, J. Sun, and G. D. Peterson. A high performance and memory efficient LU decomposer on FPGAs. *IEEE Transactions on Computers*, 61(3):366–378, March 2012.

[15] P. Benner, P. Ezzatti, E. Quintana-Ortí, and A. Remón. On the impact of optimization on the Time-Power-Energy balance of dense linear algebra factorizations. In *13th International Conference on Algorithms and Architectures for Parallel Processing*, pages 3–10, New York, NY, USA, 2013.

[16] K. K. Matam, H. Le, and V. K. Prasanna. Evaluating energy efficiency of floating point matrix multiplication on FPGAs. In *IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, MA, USA, September 2013.

[17] R. Chen and V. K. Prasanna. Energy-efficient architecture for stride permutation on streaming data. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2013.

[18] R. Chen, H. Le, and V. K. Prasanna. Energy efficient parameterized FFT architecture. In *23rd International Conference on Field Programmable Logic and Applications (FPL)*, Porto, Portugal, September 2013.

[19] A. Sanny, Y. E. Yang, and V. K. Prasanna. Energy-efficient histogram on FPGA. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2014.

[20] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. Numerical recepies in Fortran. *The Art of Scientific Computing*, 1992.

[21] F. D. Dinechin and B. Pasca. Designing custom arithmetic data paths with flopoco. *IEEE Design & Test of Computers*, 28(4):18–27, July 2011.

[22] Xilinx Inc. Virtex-5 FPGA system power design considerations. A White Paper WP285 (v1.0), Available from: http://www.xilinx.com/support/documentation/white_papers/wp285.pdf. (accessed July 2015), Feb 2008.