Computers and Electrical Engineering 47 (2015) 96–113

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

On kernel acceleration of electromagnetic solvers via hardware emulation

M. Tarek Ibn Ziad [a], Mohamed Hossam [a], Mohamad A. Masoud [a], Mohamed Nagy [a], Hesham A. Adel [a], Yousra Alkabani [a], M. Watheq El-Kharashi [a,*], Khaled Salah [b], Mohamed AbdelSalam [b]

[a] Computer and Systems Engineering Department, Faculty of Engineering, Ain Shams University, Cairo 11517, Egypt
[b] Mentor Graphics Egypt, Cairo 11361, Egypt

A R T I C L E I N F O

Article history:
Received 13 October 2014
Revised 6 August 2015
Accepted 6 August 2015

Keywords:
Electromagnetic simulations
Emulation in hardware
Finite element method
Gaussian Elimination method
Jacobi iterative method
Metamaterials

A B S T R A C T

Finding new techniques to accelerate electromagnetic (EM) simulations has become a necessity nowadays due to its frequent usage in industry. As they are mainly based on domain discretization, EM simulations require solving enormous systems of linear equations simultaneously. Available software-based solutions do not scale well with the increasing number of equations to be solved. As a result, hardware accelerators have been utilized to speed up the process. We introduce using hardware emulation as an efficient solution for EM simulation core solvers. Two different scalable architectures are implemented to accelerate the solver part of an EM simulator based on the Gaussian Elimination and the Jacobi iterative methods. Results show that the performance gap between presented solutions and software-based ones increases as the number of equations increases. For example, solving 2,002,000 equations using our Clustered Jacobi design in single floating-point precision achieved a speed-up of 100.88x and 35.24x over pure software implementations represented by MATLAB and the ALGLIB C++ package, respectively.

1. Introduction

Electromagnetic (EM) simulations continue to play an important role in modern industry. Using EM simulators is generally safer and more realistic than conducting experiments with a prototype of the final product. For instance, the increasing number of radar and communication systems in modern vehicles often results in mutual interference between these various systems. To avoid that, all of these systems' parameters along with the immediate environment must be taken into consideration in an EM simulation process to achieve precise results.

In general, the main target of an EM simulation process is to get an approximate solution to Maxwell's equations that satisfies given boundary conditions with a set of initial conditions. Many numerical techniques are used for solving Maxwell's equations, e.g., the finite difference method (FDM) [1], finite element method (FEM) [2], finite volume method (FVM) [3], and the boundary element method (BEM) [4]. FEM and FVM are widely used in engineering to model problems with complex geometries. FDM

* Corresponding author.
E-mail addresses: mohamed.tarek@eng.asu.edu.eg (M. Tarek Ibn Ziad), mohamed.hossam@eng.asu.edu.eg (M. Hossam), mohamad.masoud93@gmail.com (M.A. Masoud), eng.mohamednagyelewa@gmail.com (M. Nagy), eng.hesham.eldeeb@gmail.com (H.A. Adel), yousra.alkabani@eng.asu.edu.eg (Y. Alkabani), watheq.elkharashi@eng.asu.edu.eg (M.W. El-Kharashi), khaled_mohamed@mentor.com (K. Salah), mohamed_abdelsalam@mentor.com (M. AbdelSalam).

http://dx.doi.org/10.1016/j.compeleceng.2015.08.017
0045-7906/© 2015 Elsevier Ltd. All rights reserved.

is often regarded as the simplest method [5]. BEM is significantly less efficient than other methods due to its huge storage requirements and computational time, which tends to grow with the square of the problem size. In this work, we chose FEM to solve Maxwell's equations in the time domain for metamaterials [6]. We selected FEM rather than any other numerical method for its ability to model complex systems accurately.

Computations involved in the FEM, like other numerical methods, often consume too much time, affecting the final time-to-market value. Profiling shows that the most time-consuming part of the simulation process is the solver part [7], which is responsible for solving the resultant system of linear equations generated from the FEM. The total number of equations may reach thousands or millions of linear equations. Thus, software-based EM solvers are often too slow.

An alternative to pure software implementations is building specific computer systems using application-specific integrated circuits (ASICs) to accelerate the process. But, this approach is costly and requires a long development time. It also does not provide sufficient flexibility. Graphics processing units (GPUs) offer another approach. However, they are not suitable for problems that are data-intensive due to memory issues.

Field Programmable Gate Arrays (FPGAs) arose as a suitable solution for the EM simulation solver problem. They allow the execution of applications at near ASIC speeds, whilst circumventing the high cost of creating custom silicon [8]. Unfortunately, logic resources and memory constraints still represent a major problem that faces FPGA-based designs. In order to overcome this obstacle, there has been a recent trend towards using multi-FPGA systems to accommodate larger applications and to leverage multilevel parallelism. Examples for these multi-FPGA systems are Berkeley Emulation Engine 3 (BEE3), Cube, and Maxwell. The BEE3 is composed of modules with four tightly-coupled Virtex-5 FPGAs connected by a ring interconnection [9]. It was mainly designed for faster and larger computer architecture research. The Cube is a parallel FPGA cluster consisting of 512 Xilinx Spartan-3 FPGAs on 64 boards [10]. FPGAs in the Cube are connected in a chain to be suitable for pipeline and systolic architectures. Maxwell is a high-performance computer developed by the FPGA High Performance Computing Alliance (FHPCA) [11]. It has a total of 64 FPGAs on 32 blade servers, where each blade has an Intel Xeon CPU and two Xilinx Virtex-4 FPGAs. In this paper, we utilize another multi-FPGA system, which is the Veloce 1 hardware emulation platform from Mentor Graphics [12].

In general, the Mentor Graphics Veloce 1 emulators are fast, hardware-assisted verification systems, delivering comprehensive best-in-class emulation and acceleration platforms for SoC and embedded system verification. Each Veloce 1 emulator provides a significant increase in productivity for system-level verification because of their fast compiles, accurate modeling, productive debugging, and time-to-visibility features [13]. Furthermore, the Veloce 1 emulator family provides high-performance transaction-based acceleration, which delivers targetless acceleration with faster performance than other software solutions. Instead of using emulation in design verification only, in this paper we extend the emulator usage to be an efficient hardware accelerator for EM solvers calculations. To our knowledge, we are the first to do so.

To this end, the contributions of this paper could be summarized as follows.

1. Introduce an efficient emulation technique to accelerate the kernel of an EM simulator on a commercial hardware emulation platform from Mentor Graphics (Veloce 1) [12].
2. Validate our proposed solution on a time-domain problem of solving Maxwell's equations in metamaterials using FEM.
3. Illustrate in detail two different hardware-based approaches to solve the sparse system of linear equations, resulting from using the FEM. The first approach is based on the Gaussian Elimination method. The second one is based on the Jacobi iterative method.
4. Optimize architectures of the two proposed approaches to achieve the required performance that exceeds pure software-based solvers.
5. Show the resource utilization of implementing the optimized architectures and compare the obtained timing results with pure software implementations of the selected time-domain problem.

The rest of this paper is organized as follows. Section 2 gives a background on metamaterials and FEM. The mathematical background of our proposed approaches are introduced in Section 2. Section 3 presents some of prior work related to EM simulations, hardware acceleration for numerical methods, and the solution of sparse systems of linear equations using FPGAs. Section 4 provides an overview of our selected case study, which is used to validate proposed approaches. Internal details of various versions of our proposed solutions are presented in Sections 5 and 6, respectively. Experimental environment is described in Section 7. Obtained results and comparisons between software solutions and our optimized hardware solutions are presented in Section 8. Section 9 provides conclusion and future work.

2. Background

This section starts by presenting a brief overview of metamaterials and the governing equations used to model the wave propagation in them. It then introduces the basic steps of the FEM. Finally, it gives the essential background of Gaussian Elimination and Jacobi iterative methods.

2.1. Metamaterials

Since their first experimental demonstration in 2000 [14], metamaterials have witnessed a tremendous growth in interest in their study and potential applications in areas ranging from electronics, telecommunications, sensing, radar technology, to data storage. We can briefly describe a metamaterial as a metallic or semiconductor substance, whose properties depend on its

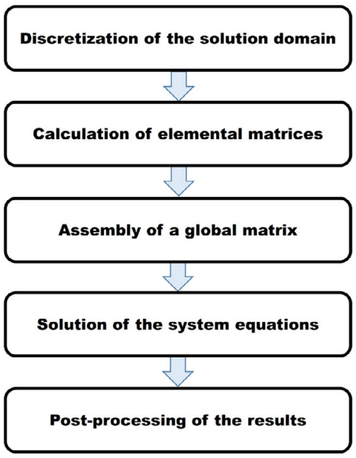

Discretization of the solution domain
Calculation of elemental matrices
Assembly of a global matrix
Solution of the system equations
Post-processing of the results

Fig. 1. FEM basic flowchart.

inter-atomic structure rather than on the composition of the atoms themselves. Certain metamaterials bend visible light rays in the opposite sense from traditional refractive media. Some metamaterials also exhibit that behavior at infrared (IR) wavelengths.

For metamaterials, the permittivity ϵ and the permeability μ are not just constants. That is due to the complicated interaction between the EM fields and the unit cell structures. As the scale of inhomogeneities in a metamaterial is much smaller than the wavelength of interest, the responses of the metamaterial to external fields can be homogenized and thus, are described using effective permittivity and effective permeability. The lossy Drude model is a popular model for metamaterial [15]. In the frequency domain, the model is described by:

$$\epsilon(\omega) = \epsilon_0\left(1 - \frac{\omega_{pe}^2}{\omega(\omega - j\Gamma_e)}\right) = \epsilon_0\epsilon_r \tag{1}$$

$$\mu(\omega) = \mu_0\left(1 - \frac{\omega_{pm}^2}{\omega(\omega - j\Gamma_m)}\right) = \mu_0\mu_r \tag{2}$$

where ω_{pe} and ω_{pm} are the electric and magnetic plasma frequencies, Γ_e and Γ_m are the electric and magnetic damping frequencies, ϵ_0 and ϵ_r are the vacuum and relative permittivity, μ_0 and μ_r are the vacuum and relative permeability, and ω is a general frequency.

2.2. Finite Element Method (FEM)

FEM is considered one of the most favorite numerical methods for solving Maxwell's equations in time domain. The basic procedure of using FEM is illustrated in Fig. 1. First, the solution domain is discretized into triangular elements or quadrilateral elements, which are the two most common forms of two-dimensional (2D) elements. Then, the element matrices and forces are formed. Afterwards, the system equations are assembled and solved. Equation solving is the most time-consuming part. Finally, results are post-processed to be presented in a suitable form for human interaction.

In FEM, the whole system is broken to many, but finite parts. So, the FEM can be very computationally intensive and available memory can be exhausted, especially when the number of grid points is large. The resulting system of linear equations may be solved either by direct methods, such as Gaussian Elimination and LU decomposition, or iterative methods, such as Jacobi, Gauss Seidel, and Conjugate Gradients.

Direct FEM solvers can provide the accurate solution for the system of linear equations with minimal round-off errors. However, they are computationally expensive in terms of both processing and memory requirements, especially in case of large matrices, since original zero entries will be filled-in during the elimination process. Alternatively, iterative methods are more efficient and more suitable for parallel computation. However, they provide lower accuracy. Therefore, higher accuracy can be obtained at the expense of the computation time and the risk of slow convergence rate. In this work, we propose two hardware emulation architectures based on the Gaussian Elimination method, as a direct method, and the Jacobi iterative method. The optimized versions of both architectures achieve better timing results than the best available software-based solutions, as will be shown in Section 8.

2.3. Gaussian Elimination method

Direct methods for solving systems of linear equations theoretically deliver exact solutions in arbitrary-precision arithmetics by a finite sequence of operations based on algebraic elimination. Gaussian Elimination, our first choice to be implemented as

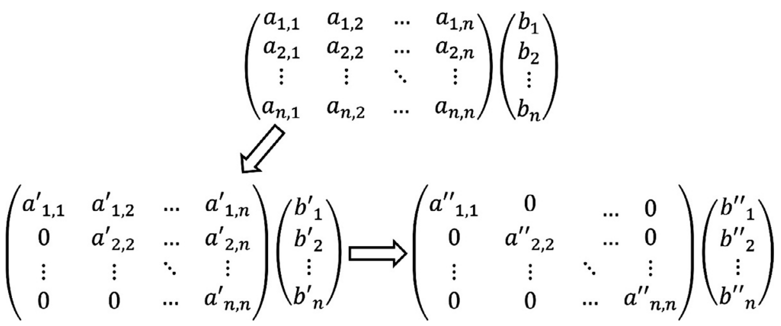


Fig. 2. Steps for the Gaussian Elimination method.

an emulation-based hardware solution, is one of the examples of direct methods. Fig. 2 summarizes the method steps. First, we use the matrix notation of linear equations with $n \times n$ coefficients matrix and $n \times 1$ right-hand-side (RHS) vector. Then, we use forward elimination to eliminate all the elements in the lower triangle of the matrix. Next, backward elimination is used to eliminate all the elements in the upper triangle of the matrix, leaving only the main diagonal. Finally, all RHS elements are divided by the corresponding elements in the main diagonal. This final operation is implemented on software to increase performance as floating-point division modules badly affect the design critical path and utilize much hardware resources.

2.4. Jacobi iterative method

As the resultant system of linear equations generated from FEM is often sparse [16], there is a need to find a suitable technique that can make use of this feature. There exists a variety of algorithms for solving sparse linear systems (SLS). However, selecting the best one depends on the matrix structure as well as different trade-offs, such as computational complexity, memory bottlenecks, convergence properties, and numerical precision. The parallel nature of the Jacobi iterative method, coupled with the massive capacity of emulator platforms, makes the Jacobi iterative method an ideal candidate for hardware acceleration.

Consider an SLS represented as $Ax = b$, where A is the coefficients matrix, x is the unknowns vector, and b is the RHS vector. Then, the i^{th} equation of the system can be represented as

$$\sum_{j=1}^{n} a_{i,j} x_j = b_i \tag{3}$$

where i and j are row and column indices, respectively. In order to solve for x_i iteratively, (3) can be rearranged in (4) to have a relation between ${x_i}^{(t+1)}$ and ${x_i}^{(t)}$, which represent the next and current iteration, respectively. t is the iteration number.

$${x_i}^{(t+1)} = \frac{b_i - \sum_{j=1, j\neq i}^{n} a_{i,j} {x_j}^{(t)}}{a_{i,i}} \tag{4}$$

Briefly, the Jacobi method starts with an initial guess of vector x, which may be assumed to be all zeros, and then solves each unknown x using (4). The obtained ${x_i}^{(t+1)}$ is then used as the current solution and the process iterates again. This process continues until it converges with a pre-defined accuracy. From (4), each x_i can be solved independently and thus the Jacobi method has inherent parallelism.

It is important to say that not all matrices can be solved using iterative methods. A matrix can be solved iteratively only if it is diagonally dominant. A matrix is said to be diagonally dominant if the magnitude of every diagonal entry is more than the sum of the magnitude of all the non-zero elements of the same row. Mathematically, a matrix is diagonally dominant if it satisfies (5) for all $i = 1, \ldots, n$. These conditions are satisfied in the coefficient matrix of the FEM [16].

$$|a_{i,i}| > \sum_{j=1, j\neq i}^{n} |a_{i,j}| \tag{5}$$

3. Related work

Here, we survey the related literature along three main lines of research: software-based EM simulators, hardware solutions for accelerating numerical techniques used in EM simulators, and general FPGA-based designs for solving systems of linear equations. First, we target the EM simulation process as it takes too long time specially in case of complex designs, and hence, it needs to be accelerated. Then, we survey different previous work that implemented hardware solutions to accelerate the numerical methods used in EM simulators. Finally, we discuss the strengths and weaknesses of prior work related to solving systems of linear equations either via direct methods or iterative ones.

In general, there are many work that handles EM simulations. For example, Ji and Hubing introduced the ElectroMagnetic Analysis Program (EMAP), as a numerical software package that employs both FEM and MoM to model EM problems [17]. They used the Complex Bi-Conjugate Gradient method to solve the final matrix equations resulting from the FEM. Unfortunately, this method is not suitable for hardware acceleration as it does not support full parallelism. Liebig et al. presented a free and open source EM solver that used the finite difference time-domain (FDTD) method [18]. Therefore, this solver inherited most of FDTD's advantages and disadvantages. The advantages include simple and robust numerical algorithm and good scalability of computing resources as a function of simulation volume size. The disadvantages are numerical dispersion and a stability constraint due to the finite difference (FD) approximation of Maxwell's equations.

Durbano and Ortiz introduced hardware-based implementations of the FDTD method [19]. They utilized a custom-designed accelerator board that supports up to 36 MB of DDR SRAM, 16 GB of DDR SDRAM, along with Xilinx Virtex-II 8000 FPGA. Taking advantage of this high-speed memory hierarchy, their solution achieves three times speed over a 30-node PC cluster. El-Kurdi et al. developed a deeply pipelined FPGA design for efficient sparse matrix-vector multiplication (SMVM) [20]. SMVM represents the core of many iterative numerical methods, such as the conjugate gradient (CG) method [21], to solve SLS, resulting from FEM formulations. In this paper, we overcame this point by implementing a simpler algorithm and splitting the main SMVM into small independent ones (using our clustering technique), as shown in Section 5.3.

A number of authors have proposed the idea of implementing efficient solutions for solving systems of linear equations on FPGAs. However, there are no work that can deal with very large systems, containing millions of equations, with acceptable accuracy. For direct solvers, the computation complexity is $O(n^3)$, where n is the number of rows of the coefficients matrix. Therefore, the overall performance is limited by computation. Zhuo and Prasanna implemented a direct solver that uses the LU factorization method [22]. They used a circular linear array of processing elements (PEs) in double precision to perform the calculations on a Virtex-II Pro XC2VP100. Johnson et al. introduced the design and prototype implementation of sparse direct LU decomposition on FPGA [23]. They compared their performance to a general purpose processor-based platform and achieved an order of magnitude speedup. Garcia et al. presented a scalable low cost architecture for the solution of linear equations based on the Gaussian Elimination method on FPGA [24]. Although they used single floating-point precision, which guaranteed a reasonable accuracy, their architecture failed to withstand large systems with hundreds of thousands of equations because of FPGA area limitations. Moussa et al. proposed the architecture of an optimized complex matrix inversion using Gauss–Jordan (GJ) Elimination on FPGA with single precision floating-point accuracy [25]. Although their optimized technique did only the critical arithmetic operations to get the needed values without performing all the arithmetic operations of the GJ Elimination algorithm, their reported results state that 10,316 slice LUTs and 5,430 slice registers were utilized to calculate the inverse of just a 16×16 matrix on a Xilinx Virtex 5 LX50T board.

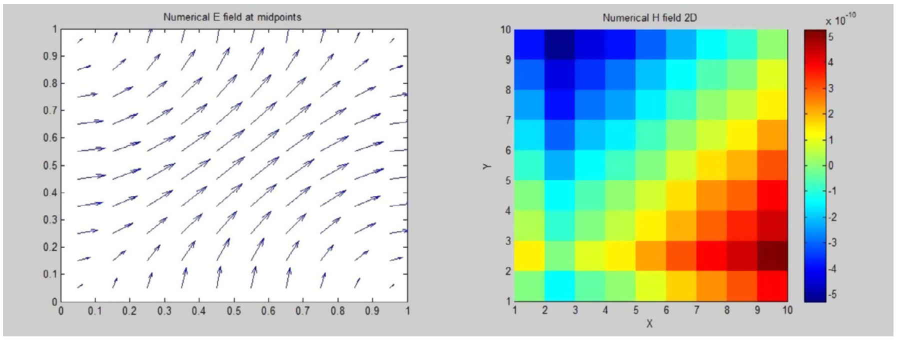
In the context of iterative solvers, Roldao and Constantinides introduced a parameterizable hardware implementation of the CG method [21] for solving dense systems of linear equations [26]. Their implementation is particularly suited for accelerating computations of multiple small-to-medium sized dense systems in parallel, using deep-pipelining. Morris and Prasanna introduced a double precision CG and a Jacobi iterative solver for sparse matrices using FPGA [27]. However, due to the limited bandwidth of on-chip memory, the matrix size is limited to 4,096 for the CG solver and 2,048 for the Jacobi iterative solver. In [28], Morris and Abed implemented a sparse matrix Jacobi iterative solver that ran on a contemporary high-performance heterogeneous computer (HPHC). They did not utilize hardware description language (HDL)-based FPGA kernel designs. They used a high-level language (HLL)-based design instead. Their solver, which was implemented using the Carte HLL-to-HDL compiler, could handle matrices up to an order $n = 8K$ only.

Pourhaj et al. presented a scalable hardware architecture for a Jacobi processor that represented the main component in a Jacobi solver [29]. Their architecture is independent of system size as multiple Jacobi processors can be cascaded to form a bigger Jacobi solver for processing a large linear system. However, every Jacobi processor must have read access to the memories of the RHS vector, coefficient matrix, and the current solution vector and write access to the new solution vector memory. Thus, at larger number of equations, the utilized logic resources will increase exponentially and the performance will degrade. For solving only eight equations, they utilized 15,522 slice LUTs on Xilinx XC5VLX50-FF1153 board. Our proposed designs are scalable and can solve millions of equations without affecting the overall performance.

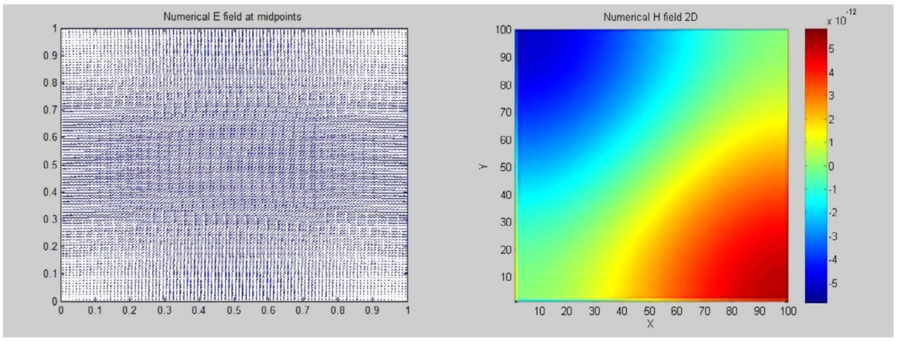
Greisen et al. investigated several solver techniques, discussed hardware trade-offs, and introduced FPGA architectures of Cholesky direct solver and BiCGSTAB iterative solver [30]. Although they outperformed software implementations, their iterative solver design was memory-bandwidth limited, as vectors of the full problem size need to be accessed in each iteration. We solved this issue in our clustered designs by dividing the coefficients matrix into independent clusters. Finally, we conclude that previous work is not scalable when dealing with large sparse systems containing hundreds of thousands of equations that result from EM simulations. As a result, the focus of our current work is to develop scalable hardware architectures that can solve large systems of linear equations of any size, up to the massive capacity of the emulator logic resources and memory bandwidth with suitable floating-point accuracy.

4. Electromagnetic solver

Here, we describe our selected case study, represented by an EM solver for solving Maxwell's equations in metamaterials using FEM. The solver was introduced by Li and Huang [31]. They presented a detailed discussion on coding the 2D edge element for solving Maxwell's equations for metamaterials using FEM. They covered the whole programming process including mesh generation, element matrices calculation, assembly process, and post-processing of numerical solutions. The full solution was based on MATLAB to make use of its facilities in solving equations and dealing with matrices efficiently. We follow their codes and build a MATLAB graphical user interface (GUI) program that takes inputs from the user and generates the final graphs of Electric



(a) 10x10 meshes.



(b) 100x100 meshes.

Fig. 3. Final orientation of numerical electric field (E) and magnitude of numerical magnetic field (H) for a unity element using different mesh sizes.

(E) and Magnetic (M) fields after performing FEM procedures. As the program simulates a 2D element, inputs include the x and y co-ordinates of the lower and higher edges of the element. Number of meshes and material properties are also determined by the user. The user is also capable of deciding whether to use the software-based solution or the hardware-based solution. Both solutions have the same functions for pre-processing and post-processing calculations performed on MATLAB. However, they differ in the solver part. The software-based solution depends on state-of-the-art software solvers, while the hardware-based solution is performed on the hardware platform, described in Section 7.

A sample output of the GUI program represented by the numerical E and H field graphs is shown in Fig. 3. Increasing the number of meshes in the x and y directions results in increasing the obtained accuracy. However, that causes a significant increase in the number of equations-to-solve.

5. Gaussian Elimination method hardware implementations

In this section, we discuss our proposed Gaussian Elimination hardware-based solvers in detail. We start with the *Basic Gaussian*, which is a simple design modeling the Gaussian Elimination method in a direct manner without any optimization. Then, we present the *Gaussian-using-sparse model*, which introduces using the concept of a sparse matrix in the design and therefore leads to smaller hardware and memory sizes with a reasonable performance. Finally, we introduce the *Clustered Gaussian*, which allows further parallelism through adding more ALUs that work in parallel, leading to the best timing and hardware utilization among our Gaussian Elimination designs.

5.1. Basic Gaussian

The design shown in Fig. 4 represents the basic implementation of the Gaussian Elimination method. It consists of three main modules; memory, ALU, and control unit. The memory is used for storing all the matrix elements, as well as the right hand side (RHS) vector, in the IEEE 754 floating-point representation with single precision. Each memory entry represents one row from the matrix, concatenated with the corresponding element from the RHS vector, so the word size of the memory equals $32 \times (n+1)$ bits, where n is the number of matrix columns.

The design uses a vectored ALU, which performs vector operations on the data received from memory. Operations are done vector-by-vector and not element-by-element, which is the main advantage of using hardware solutions as they allow a higher degree of parallelism compared to software-based solutions. Inputs to the ALU are two rows from the memory and address from

Algorithm 1 Our Basic Gaussian.

```
1:  procedure BASIC GAUSSIAN
2:  input: number of linear equations
3:      while index1 of current row < end of the memory do
4:          Load memory of current index1
5:          while index2 of current row ≤ end of the memory do
6:              Load memory of current index2
7:              Divide Mem[index2][0] by Mem[index1][0]
8:              Multiply output of step 7 by Mem[index1] of step 4
9:              Subtract output of step 8 from Mem[index2] of step 6
10:             Store output of step 9 in the memory of current index2
11:             Increment index2 by one
12:         end while
13:         Increment index1 by one
14:     end while
15:     while index1 of current row > start of the memory do
16:         Load memory of current index1
17:         while index2 of current row ≥ start of the memory do
18:             Load memory of current index2
19:             Divide Mem[index2][0] by Mem[index1][0]
20:             Multiply output of step 19 by Mem[index1] of step 16
21:             Subtract output of step 20 from Mem[index2] of step 18
22:             Store output of step 21 in the memory of current index2
23:             Decrement index2 by one
24:         end while
25:         Decrement index1 by one
26:     end while
27: end procedure
```

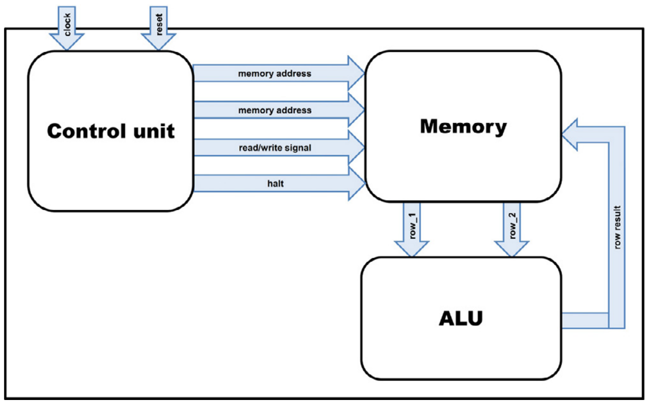


Fig. 4. Block diagram of our Basic Gaussian.

the control unit. There exists only one output which represents one row to be saved in the memory. The ALU operates on the two rows and uses the address from the control unit to know which element to be eliminated according to Gaussian Elimination operations. The ALU consists of:

1. Division module, which divides one element from the input row by an element from the other row.
2. Two multiplexers, which are used to choose elements from the two input rows to enter the division module according to the address from the control unit.
3. The vector multiplication module, which consists of a number of multiplication modules equal to the number of the elements of a row.
4. Vector subtraction module, which consists of a number of subtraction modules equal to the number of the elements of a row. Each module subtracts one output of the multiplication modules from the corresponding element in the second input row.

Finally, the control unit is used for synchronizing the ALU and the memory together, as well as controlling memory addresses and memory read and write operations. It also generates the halt signal, which is used to indicate that the equations have been solved and stops the system automatically. Algorithm 1 shows the operation procedures.

Algorithm 2 Our Gaussian-using-sparse Model.

```
 1: procedure GAUSSIAN-USING-SPARSE MODEL
 2: input: number of linear equations
 3:     while index of the current row $<$ end of diagonal memory do
 4:         Load register file of current index (addresses of the first and the second diagonal elements)
 5:         Load diagonal memory of the two addresses in step 4
 6:         Load elements memory of current index (to be eliminated)
 7:         Divide operand from step 6 by first operand[0] from step 5
 8:         Multiply output of step 7 by the operand of step 6
 9:         Subtract output of step 8 from the second operand[0] of step 5
10:         Multiply output of step 7 by first operand[1] of step 5
11:         Subtract output of step 10 from the second operand[1]
12:         Increment index by one
13:         Store output of steps 9 and 11 in diagonal memory of index
14:     end while
15:     while index of current row $>$ start of diagonal memory do
16:         Load register file of current index (addresses of the first and the second diagonal elements)
17:         Load diagonal memory of the two addresses in step 16
18:         Load elements memory of current index (to be eliminated)
19:         Divide operand from step 18 by first operand[0] from step 17
20:         Multiply output of step 19 by first operand[1] of step 17
21:         Subtract output of step 20 from the second operand[1] of step 17
22:         Decrement index by one
23:         Store output of step 21 in diagonal memory of index
24:     end while
25: end procedure
```

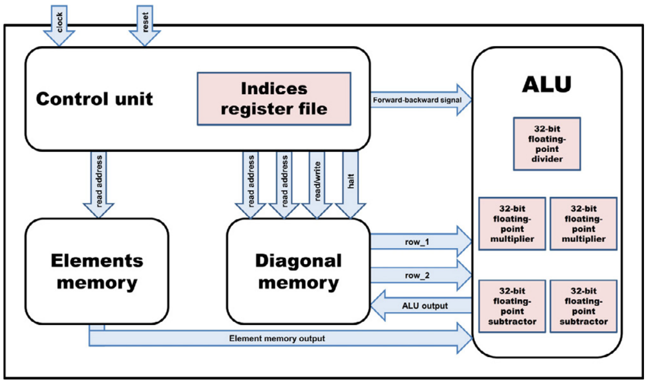


Fig. 5. Block diagram of our Gaussian-using-sparse.

The main disadvantages of this design is its high time complexity. A lot of time is wasted in eliminating zero elements. Moreover, a large amount of memory bytes are used to store the full matrix. This is addressed in the next design.

5.2. *Gaussian-using-sparse*

In this design, we tried to solve most of the problems of the previous design that are related to performance and resource utilization. This design stores only the non-zero elements and their locations (indices) in the matrix, which makes significant reduction in memory size compared to the previous design.

It was found that the coefficients matrix generated using FEM in our application has the following properties [31]:

1. All elements in the main diagonal of the matrix are non-zero elements.
2. Output matrix is symmetric.
3. Maximum number of non-zero elements in one row is three.

Based on these FEM matrix properties, the design is optimized to improve performance and hardware utilization. The modified design contains one ALU, control unit, and two types of memories, as shown in Fig. 5.

For the memories part of the design, the first one, diagonal memory, is used to store the diagonal elements in the matrix concatenated with the elements of the RHS elements. So, the word size of this memory is 64 bits; 32 bits for the diagonal

Algorithm 3 Our Clustered Gaussian.

```
 1: procedure CLUSTERED GAUSSIAN
 2: input: number of linear equations and number of clusters
 3:     while index of current row < end of diagonal memory do
 4:         Load diagonal memory of index and index + 1
 5:         Load elements memory of index (matrix elements to be eliminated)
 6:         Divide operands from step 5 by first operands[0] from step 4
 7:         Multiply output of step 6 by operands of step 5
 8:         Subtract output of step 7 from the second operands[0] of step 4
 9:         Multiply output of step 6 by first operands[1] of step 4
10:         Subtract output of step 9 from the second operands[1]
11:         Increment index by one
12:         Store output of steps 8 and 10 in diagonal memory at index
13:     end while
14:     while index of current row > start of diagonal memory do
15:         Load diagonal memory of index and index - 1
16:         Load elements memory of index (matrix elements to be eliminated)
17:         Divide operands from step 16 by first operands[0] from step 15
18:         Multiply output of step 17 by first operands[1] of step 15
19:         Subtract output of step 18 from the second operands[1] of step 15
20:         Decrement index by one
21:         Store output of step 19 in diagonal memory at current index
22:     end while
23: end procedure
```

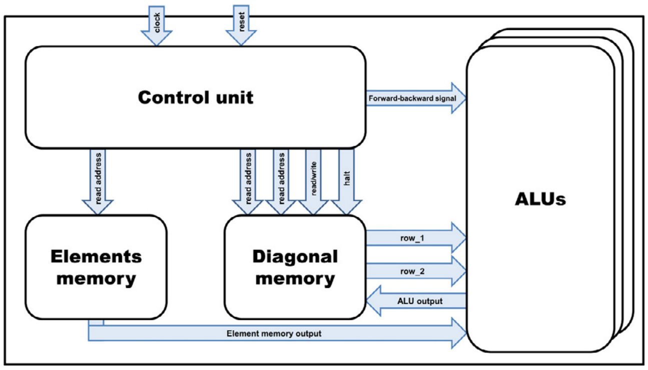


Fig. 6. Block diagram of our Clustered Gaussian.

elements and 32 bits for the RHS vector, while the memory depth equals the number of equations of the system. The elements memory is used to store the non-zero non-diagonal elements. It has a word size of 32 bits.

The ALU has a fixed number of submodules, taking advantage of the third property of the FEM matrix. The new ALU consists of one division module, two multiplication modules, and two subtraction modules. All modules use single floating-point precision.

The control unit is different from the previous one as it has a register file that stores the indices of each element in the elements memory, the size of this register file equals the length of elements memory and each entry consists of the element row index concatenated with the element column index. The main signal generated from the control unit is the forward-backward signal, which indicates the mode of the system, forward or backward elimination, in order to control ALU operations. It is also responsible for providing memories addresses along with the final halt signal to indicate the end of operations. Algorithm 2 shows the operation procedures.

Although this design improves the performance and memory size, there are some disadvantages regarding the hardware resources utilization as the hardware size is fixed regardless of number of equations solved due to the fixed size of the ALU, which is a waste of available resources.

5.3. Clustered Gaussian

This design depends mainly on the use of a specific feature in the coefficient matrix generated from our GUI program as a result of using FEM. We called it, Clustering, and it can be used to reduce dependencies between equations in the sparse linear system. This allows for higher levels of parallelism in implementation by using multiple ALUs in parallel. Thus, the main objective

Algorithm 4 Our Basic Jacobi.

```
 1: procedure BASIC JACOBI
 2: input: number of linear equations
 3:     while iteration number < the pre-defined number of iterations do
 4:         while index of current row < number of matrix rows do
 5:             Load current row (two elements) of the index memory
 6:             Iteration number decides which memory of the two result memories will be written and which one will be read
 7:             Send the elements loaded in step 5 to the read result memory
 8:             Load the elements from the read result memory to the ALU
 9:             Load one row from the non-diagonal memory to the ALU
10:             Multiply the elements from step 9 by the elements from step 8
11:             Add the results of the multiplication operations in step 10
12:             Load current row from the diagonal and RHS memory
13:             Save the loaded element in step 12 in a register inside the ALU
14:             Subtract the result of step 11 from the RHS element of step 13
15:             Divide the result of step 14 by the diagonal element
16:             Send the result of step 15 to the written result memory
17:             Save the output of step 15 inside the result memory
18:             Increment the current row number
19:         end while
20:         Increment the current iteration number
21:         Reset the current row number to 0
22:     end while
23: end procedure
```

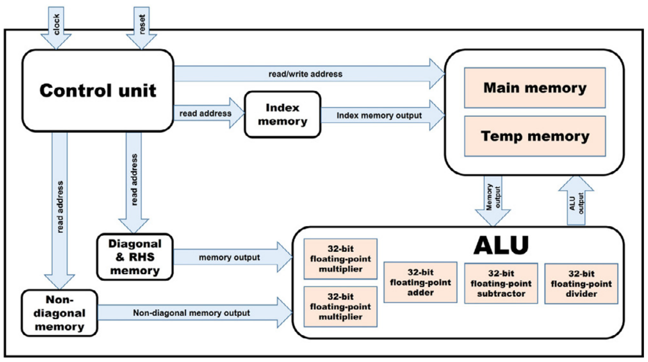


Fig. 7. Block diagram of our Basic Jacobi.

of proposing the Clustering concept is to separate equations into independent clusters, where each cluster has a fixed shape, so locations of the non-zero elements in each row are predictable.

Fig. 6 shows the block diagram for the *Clustered Gaussian* hardware design. The main difference between it and the *Gaussian-using-sparse model* design is the usage of multiple ALUs in parallel. The number of the ALUs used is equal to the number of clusters in the design. The memories' architectures remain the same, except for the memory depth and word size. Each entry in the memory stores the diagonal element and the RHS element, concatenated with those of other clusters. So, the memory word size is $64 \times$ number of clusters bits. The memory depth equals the number of equations per cluster.

The control unit has the same design as before, except that the register file has been removed due to the usage of clusters. The elements' indices are predictable. They are functions of the value of the counter, which is used to control the address of the elements memory or drives the halt and forward-backward signals. Algorithm 3 shows the operation steps.

This design achieves the best timing results among other Gaussian Elimination designs. It also allows for better resource utilization, as will be discussed in details in Section 8.

6. Jacobi iterative method hardware implementations

In this section, similar to what we did in the previous one, we first start with the hardware implementation of our *Basic Jacobi*, which is a simple design modeling the Jacobi iterative method with no optimization. Then, we present the *Pipelined Jacobi*, which is an improved version of the *Basic Jacobi* to reduce the execution time. Finally, we introduce the *Clustered Jacobi*, which uses the concept of Clustering, discussed before, to achieve maximum performance.

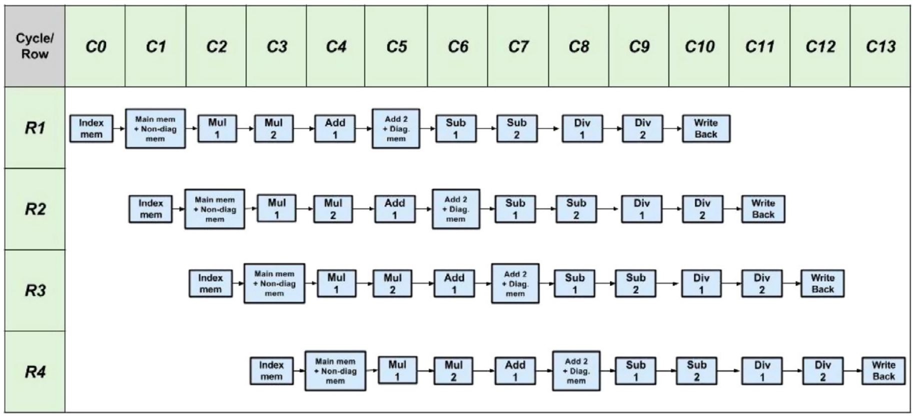


Fig. 8. Multi-clock-cycle diagram for our Pipelined Jacobi. *C* is the current clock cycle and *R* is the current row of the coefficients matrix. Each box represents the operation done on a specific row during a specific clock cycle.

6.1. Basic Jacobi

The architecture shown in Fig. 7 is the simplest design used for implementing the Jacobi iterative method without any optimization. This design consists of a control unit, ALU, and five memories; main memory, temporary memory, diagonal and RHS memory, non-diagonal memory, and index memory.

The main memory and its replica (temporary memory) contain the required solution vector, x. Initially, they are loaded with zeros. Each row of those two memories contains only one 32-bit floating-point element. In each iteration, one of these memories loads data, to be operated on, to the ALU, while the other memory stores the calculation results. This is to ensure that all data operated on is the data from the last iteration not the new data. After the iteration finishes, data is stored in one of the two memories depending on the current iteration number in order to be operated on during the next iteration until convergence is reached.

The diagonal and RHS memory, and the non-diagonal memory contain the diagonal and RHS elements, and non-diagonal elements of each row of the matrix, respectively. Each row of the diagonal and RHS memory, and the non-diagonal memory contains two 32-bit floating-point elements. The index memory stores the indices of the non-zero, non-diagonal elements. Those indices are then used to choose the corresponding elements in the result memories (main and temporary memories) to be multiplied by the non-diagonal elements. Memory depth of all memories is equal to the number of rows of the matrix.

The ALU is responsible for performing all arithmetic operations on data. It models the operations described by (4). Thus, it contains two floating-point multipliers, one floating-point adder, one floating-point subtractor, and one floating-point divider. All modules use IEEE 754 floating-point representation with single precision. Algorithm 4 shows the solver steps. The illustrated steps are repeated until results converge. Convergence here is represented by a pre-defined number of iterations given by the user.

Finally, the control unit consists of three counters. The first shows the current step, which is being executed in the design; either loading data from memory, executing arithmetic operations, or storing data in memory. The second counter represents the current row (being operated on). While the third counter represents the current iteration. However, the main problem with this ideal design is its low throughput compared to its resource utilization since only a single ALU is used. This issue is solved in the *Pipelined Jacobi*.

6.2. Pipelined Jacobi

This design is an improved version of *Basic Jacobi*. It uses the same main components as *Basic Jacobi*, which are a control unit, main memory, replica of main memory, diagonal and RHS elements' memory, non-diagonal elements' memory, index memory, and an ALU. The control unit is modified; each memory had to have its own counter, which generates an address that corresponds to the current element required for each ALU operation. The solver steps used here are exactly the same as the steps of the *Basic Jacobi* shown in Algorithm 4. The only difference is that *Pipelined Jacobi* does not wait until the current row is completely processed to start loading the next row, each new row is started immediately after the processing of the current row starts (after one clock cycle). We pipeline the ALU operations as well as the memory access.

Fig. 8 illustrates how the design works, where each column represents a clock cycle of execution and each row represents the operations done on each row of the matrix. The intersection between each row and column represents the operation done on that row of the matrix during that clock cycle. As every floating-point module used in this design needs two clock cycles in order to operate correctly, the notation 1 and 2 is used in the diagram after the name of every ALU operation.

Algorithm 5 Our Clustered Jacobi.

```
1:  procedure CLUSTERED JACOBI
2:  input: number of linear equations and number of rows per cluster
3:      while difference between current results and results of the previous iteration > the pre-defined accepted tolerance do
4:          while index of current row < number of matrix rows do
5:              Load one row from the main memory to the ALU
6:              Send the same row to the Result Convergence Check module
7:              Load one row from the non-diagonal memory to the ALU
8:              Multiply elements from step 5 by elements from step 7
9:              for each matrix row inside the cluster do
10:                 Add the results of multiplications from step 8
11:             end for
12:             Load a row from the RHS memory
13:             for each matrix row inside the cluster do
14:                 Subtract the result from step 8 from the output of step 12
15:             end for
16:             Load a row from the diagonal memory
17:             for each matrix row inside the cluster do
18:                 Multiply elements from step 14 by elements from step 16
19:             end for
20:             Compare the results of step 18 with the data saved in step 6 to check the tolerance
21:             Save the results from step 18 in the main memory (cluster)
22:             Increment the current row number
23:         end while
24:         Reset the current row number to 0
25:     end while
26: end procedure
```

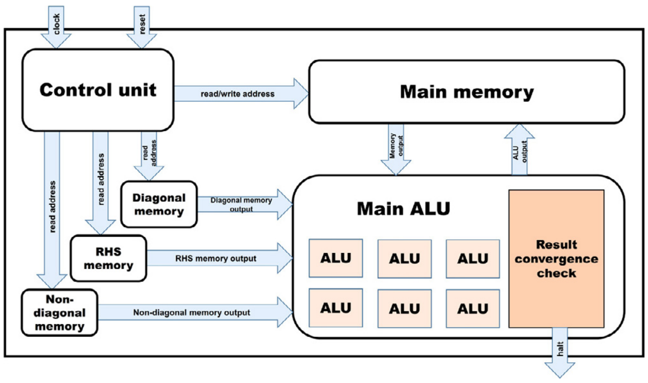


Fig. 9. Block diagram of our Clustered Jacobi.

Although pipelining greatly improves performance, we still need to wait for few extra clock cycles at the end of each iteration in order to allow the newly calculated data to be written since it might be needed by the first row during the next iteration, as implied by the Jacobi method. This pipeline data hazard is solved in the *Clustered Jacobi* design.

6.3. Clustered Jacobi

Fig. 9 illustrates the hardware architecture for *Clustered Jacobi* which is based on the Clustering technique, as the *Clustered Gaussian* design described before. It performs the same arithmetic operations of the Jacobi iterative method except that the final division operation is turned into a multiplication by the inverse of the diagonal element to minimize the design critical path. That is due to the remarkable difference in latency and required hardware resources between multiplication and division floating-point modules.

The memory architecture is slightly different here from the two previous Jacobi designs as each memory row now contains a whole cluster of data instead of just one row of the matrix. This makes up for memory constraints and allows memories to serve

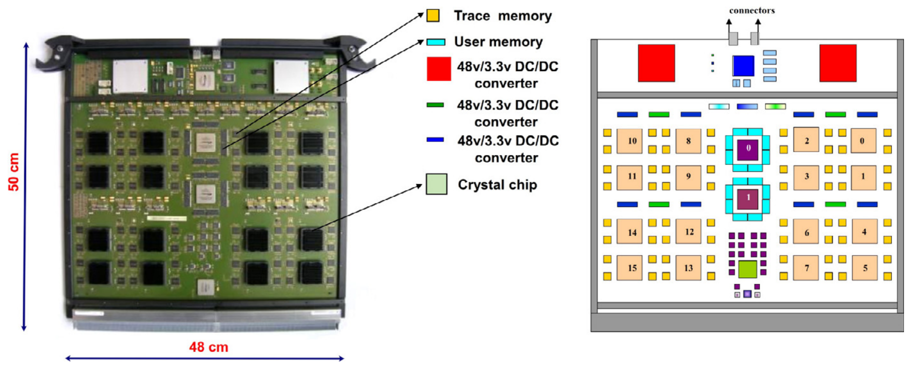


Fig. 10. Construction of Mentor Graphics Veloce 1 Advanced Verification Board (AVB).

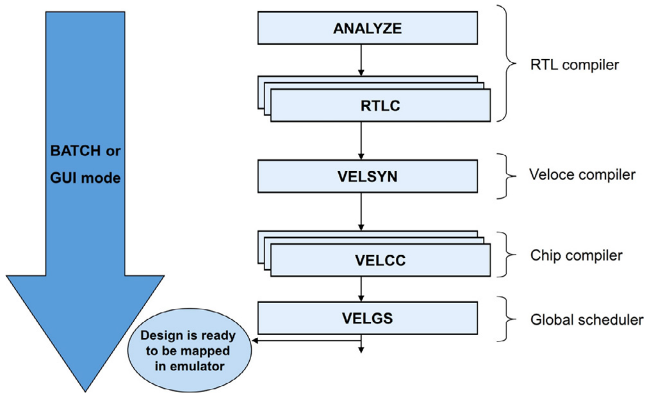


Fig. 11. Emulation flow overview.

more arithmetic cores simultaneously. There is no need for the old index memory as each memory row becomes self-contained and has its corresponding operands ready in other memories. Temporary result memory does not exist neither since matrix rows can be overwritten after being operated on without affecting other rows.

A new module, called result convergence check, is introduced in this design to define the number of iterations until reaching convergence based on a pre-defined accuracy. This generates a halt signal, which indicates the end of operations. Algorithm 5 shows the solver steps.

This design has the least execution time among the other proposed designs. It is a very flexible design that can be configured for a broad range of matrix sizes depending on the available hardware resources.

7. Experimental setup

Our proposed hardware architectures are modeled using Verilog. Actual tests are performed on Veloce 1 emulation platform from Mentor Graphics with a total capacity of eight advanced verification boards (AVBs) [13]. Fig. 10 gives a close look on the Veloce 1 AVB architecture. Each AVB offers 16 crystal chips and 512 MB of user memory. Crystal chips are the main programmable logic. They are Mentor Graphics-developed high-performance chips, specifically designed for emulation. Emulator crystal chips have a 96K configurable programmable block (CPB) and are capable of emulating up to 500K gates. So, the total available capacity on the used Veloce 1 emulator equals 128 crystal chips with 4 GB of memory. That provides enough resources of look up tables (LUTs) and flip-flops to handle a bigger design than any single FPGA that exist nowadays.

Fig. 11 highlights the main steps in the emulation design process. The Analyze step takes Verilog files as inputs and performs syntax checking. Register transfer level compiler (RTLC) is the primary RTL front-end for the Veloce 1 emulation system. It is responsible for generating a structural Verilog netlist of Veloce 1 primitives (LUT, tri-state, flip-flop, latch, and memory). Veloce 1 synthesizer or VELSYN performs partitioning and generating ASIC netlists for each crystal chip. After that, VELCC, which stands for Veloce 1 chip compiler, does placing and routing. Finally, VELGS or Veloce 1 global scheduler performs final timing analysis and generates timing information for resources access, memory and IO accesses, emulator events, and clocks.

Table 1
Timing results of our 32-bit floating-point Clustered Gaussian hardware compared to software-based solvers and speed-up.

Number of	Our Clustered Gaussian	MATLAB mldivide		C++ Eigen	
equations	Time (s)	Time (s)	Speed-up	Time (s)	Speed-up
420	0.000325	0.000204	00.63	0.000500	01.54
11,100	0.001891	0.004056	02.14	0.005000	02.64
19,800	0.002480	0.009725	03.92	0.010001	04.03
44,700	0.004271	0.025324	05.93	0.022001	05.15
60,900	0.004676	0.036769	07.86	0.032002	06.84
244,300	0.009715	0.216702	22.31	0.154883	15.94

Table 2
Timing results of our 32-bit floating-point Clustered Jacobi hardware compared to software-based solvers and speed-up.

Number of	Our Clustered Jacobi	MATLAB Jacobi		C++ ALGLIB	
equations	Time (s)	Time (s)	Speed-up	Time (s)	Speed-up
420	0.000319	0.000998	03.13	0.000350	01.10
4,900	0.001091	0.005438	04.98	0.002001	01.83
11,100	0.001721	0.011292	06.56	0.003001	01.74
44,700	0.003793	0.043967	11.59	0.012000	03.16
179,400	0.006167	0.215737	34.98	0.057003	09.24
2,002,000	0.024040	2.425221	100.88	0.847049	35.24

8. Experimental results

In this section, we evaluate the performance of our two optimized designs; *Clustered Gaussian* and *Clustered Jacobi*, described in Sections 5.3 and 6.3, respectively. The evaluation process includes collecting the timing results, calculating the speed-up over different software solutions, and showing the resource utilization on the physical emulation platform, described in Section 7. All the used test cases are generated from our MATLAB GUI program based on the EM solver, discussed in Section 4. Finally, we compare our two clustered architectures and mention the differences between them and other hardware-based solutions, mentioned in Section 3.

8.1. Speed-up calculations

The speed-up of our two emulation-based approaches is evaluated against various software solutions based on direct and iterative methods for solving systems of linear equations on a 2.00 GHz Core i7-2630QM CPU. Table 1 lists results obtained from comparing the 32-bit floating-point *Clustered Gaussian* against two software solutions; mldivide (\), the MATLAB special operator for solving linear systems of equations and Eigen, a C++ template library for linear algebra [32]. The mldivide operator is chosen as it uses direct methods for solving large systems of linear equations [33]. It also makes use of MATLAB optimizations for dealing with matrix operations to achieve higher performance. We also implemented the sparse Cholesky direct solver from Eigen as a C++ benchmark. Speed-up is computed by dividing the software runtime by our proposed hardware runtime. In order to be capable of calculating a near accurate processing time, the software was run for many iterations so that the data resides in the local cache and thus reduces any disk access time.

Table 2 illustrates the obtained results from comparing the 32-bit floating-point *Clustered Jacobi* hardware implementation against two software solutions for different test cases, as well. The first software benchmark is a standard Jacobi iterative method implementation using MATLAB [34]. The second is an iterative solver from ALGLIB [35], an open-source numerical analysis library that supports several programming languages, including C++. It was chosen due to its ease of implementation and ability to be compiled across multiple platforms. In our test cases, we set the pre-defined tolerance to 10^{-6}, in order to define the termination condition.

Fig. 12 shows a graphical representation of the speed-up results in Table 1, whereas Fig. 13 does the same but for the results in Table 2. Speed-up is plotted against the total number of solved equations for different test cases. It is clear that speed-up increases as the number of equations-to-solve increases. Thus, more speed-up can be obtained at larger numbers of equations.

8.2. Resource utilization

Tables 3 and 4 show the logic utilization, memory capacity, and operating frequency for our two proposed emulation-based approaches, *Clustered Gaussian*, and *Clustered Jacobi*, respectively. ALUs represent the basic design units. In *Clustered Gaussian* design, number of ALUs equals the number of clusters in the design, whereas in the *Clustered Jacobi* design, the number of ALUs equals the number of equations per cluster. That difference comes from the specific nature of each approach of being a direct or

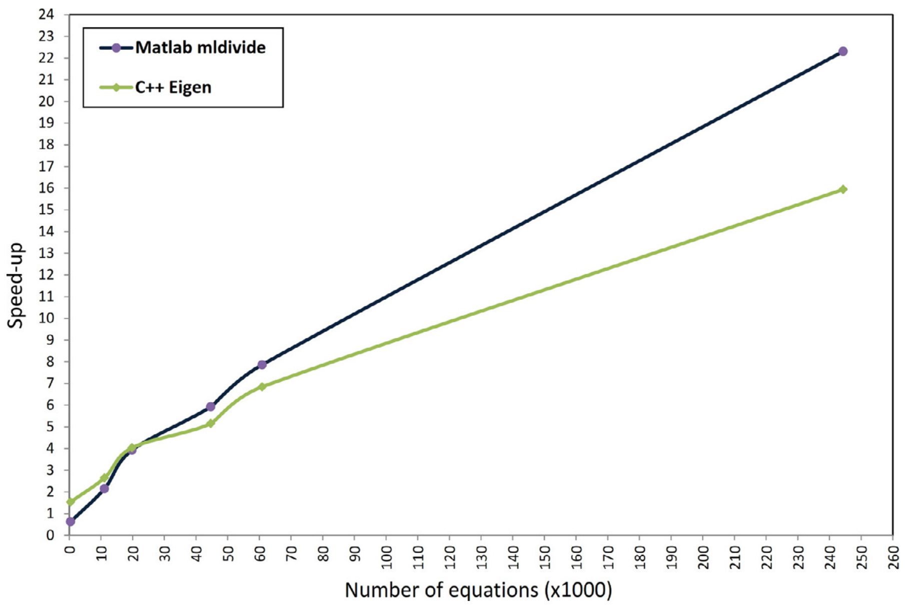


Fig. 12. Speed-up of our 32-bit floating-point Clustered Gaussian hardware over software implementations of MATLAB mldivide and C++ Eigen.

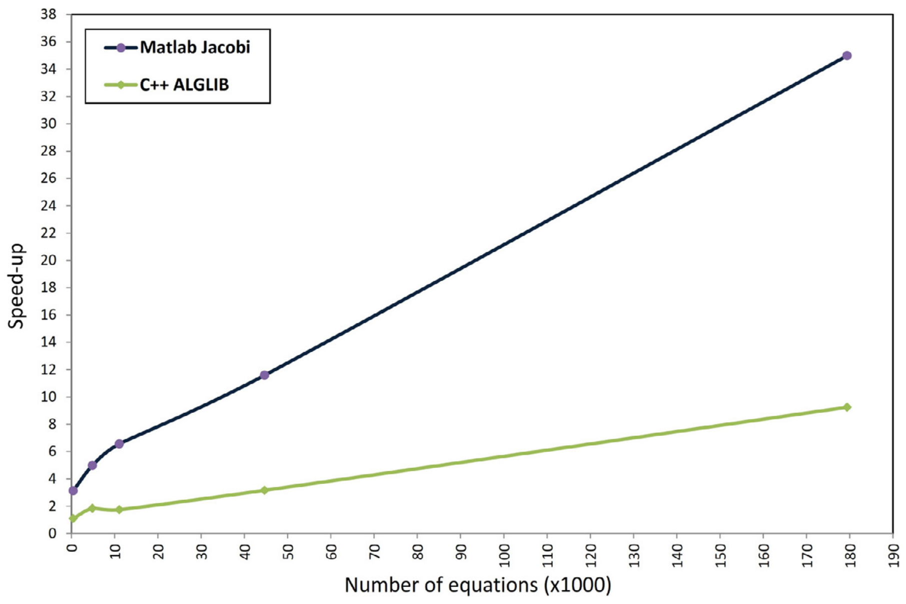


Fig. 13. Speed-up of our 32-bit floating-point Clustered Jacobi hardware over software implementations of MATLAB Jacobi and C++ ALGLIB.

Table 3
Resource utilization for our 32-bit floating-point Clustered Gaussian for different test cases with different number of equations.

	Clustered Gaussian test cases					
Number of equations	420	11,100	19,800	44,700	60,900	244,300
Number of ALUs	30	150	200	300	350	700
Max frequency (KHz)	406.5	387	396	347	370.4	358.4
Number of LUTs	195,472	977,195	1,302,918	1,954,331	2,280,075	4,560,087
Number of flip-flops	15,376	76,822	102,422	153,624	179,224	358,426
Memory bytes	5,760	230,400	307,200	921,600	1,075,200	4,300,800
Number of FPGAs in design	3	13	18	26	31	63

Table 4
Resource utilization for our 32-bit floating-point Clustered Jacobi for different test cases with different number of equations.

	Clustered Jacobi test cases					
Number of equations	420	4,900	11,100	44,700	179,400	2,002,000
Number of ALUs	14	49	74	149	299	999
Max frequency (KHz)	1666.7	1851	1754.4	1587.3	1754.4	1333.3
Number of LUTs	106,401	356,749	535,566	1,071,976	2,144,796	7,158,357
Number of flip-flops	18,752	57,684	85,490	168,896	335,702	1,115,220
Memory bytes	8,704	124,416	376,832	1,521,664	6,115,328	40,943,616
Number of FPGAs in design	2	5	8	15	29	98

Table 5
Timing and resources utilization comparison between our optimized emulation-based designs.

Number of	Our Clustered Gaussian		Our Clustered Jacobi	
equations	Time (s)	FPGAs/Memory	Time (s)	FPGAs/Memory
420	0.000325	03/005,760	0.000319	02/0,008,704
11,100	0.001891	13/230,400	0.001721	08/0,376,832
44,700	0.004271	26/921,600	0.003793	15/1,521,664

Table 6
Timing comparison between our optimized emulation-based designs and some of the designs in Section 3. The last two columns represent the maximum reported coefficients matrix size (number of equations) and the total execution time, respectively.

Design reference	Year	Algorithm	Matrix type	Precision	Hardware device	Maximum matrix size	Execution time
[22]	2006	LU factorization	dense	double	Virtex-II Pro XC2VP100	1,000	171.00 ms
[27]	2007	Conjugate Gradient	sparse	double	Reconfigurable computer	4,096	74.40 s
[27]	2007	Jacobi	sparse	double	Reconfigurable computer	2,048	12.30 s
[23]	2008	LU decomposition	sparse	-	Stratix[b] 1S25	19,285	-
[29]	2008	Jacobi	dense	double	Virtex[c] 5 XC5VLX50	8	5.00 μs/iteration
[24]	2012	Gaussian Elimination	dense	single	Virtex-5 XC5VLX330T	96	1.35 ms
[25]	2013	GJ Elimination	dense	single	Virtex 5 LX50T	16	9.00 μs
[28]	2013	Jacobi	sparse	single	HPHC[a]	8,000	25.00 s
[30]	2013	Cholesky	sparse	single	Stratix IV 530 GX	33,000	15.00 ms
[30]	2013	BiCGSTAB	sparse	single	Stratix IV 530 GX	33,000	60.00 μs/iteration
Ours	2015	Gaussian Elimination	sparse	single	Veloce 1 emulator	244,300	9.72 ms
Ours	2015	Jacobi	sparse	single	Veloce 1 emulator	2,002,000	24.04 ms

[a] High-performance heterogeneous computer.
[b] Stratix is an FPGA device from Altera.
[c] Virtex is an FPGA device from Xilinx.

an iterative solver. Number of LUTs and flip-flops are the major metrics for defining the resource utilization, whereas the number of FPGAs in a design only represents an easier way to compare between different test cases as the emulator total capacity is divided into small connected FPGAs, as described in Section 7.

8.3. Performance evaluation

Table 5 lists the timing results and the resource utilization (in number of used FPGAs and memory bytes) of our two optimized emulation-based approaches, *Clustered Gaussian* and *Clustered Jacobi*, respectively. Both designs run on the Veloce 1 emulator with the configuration described in Section 7 and results are reported.

The timing results show that almost the two approaches have the same performance with a small enhancement for the *Clustered Jacobi* in case of larger numbers of equations. Logic resources, represented by number of FPGAs in the design, are also better in case of using *Clustered Jacobi* as it uses less number of FPGAs than in the case of the *Clustered Gaussian* to handle the same number of equations, which utilizes memory better, unlike the *Clustered Jacobi*.

It is worth mentioning that direct comparisons with other hardware-based solutions, mentioned in Section 3, are not applicable here as the generated test cases used there are small, limited, and do not fit within the same specifications of our system of linear equations generated from the FEM process. However, we use Table 6 to indicate the superior aspects of our approaches compared to other work mentioned in Section 3. In [23], the floating-point accuracy was not mentioned. Although the implementation was tested using three different test cases with maximum number of equations that equals 19,285, no information was given about the total execution time. Only the final speed up over software benchmarks was introduced. In [29] and [30], the execution time per one iteration was given. It is obvious that no previous work has solved as high number of equations, within a reasonable execution time, as ours. To be fair, we should note that the massive emulator resources helped in achieving these results.

9. Conclusion and future work

This work tackled the EM simulation performance problem, which is one of the most widely studied problems in electrical engineering. The main target was to implement hardware solutions capable of minimizing the huge time requirements for simulating complex EM systems. We applied our solution on a real EM application by solving time-domain Maxwell's equations in metamaterials using FEM.

Two different solutions for solving systems of linear equations were implemented with their optimized versions and run on a physical emulation platform, Veloce 1, from Mentor Graphics. It is the first time to introduce emulation technology as an effective solution for EM solver acceleration and make use of the massive capacity of hardware emulators.

Compared to MATLAB, a powerful benchmark for matrix operations, on a 2.00 GHz Core i7-2630QM CPU, a speed-up of 22.31x was achieved for solving 244,300 equations using our *Clustered Gaussian* design, whereas a speed-up of 15.94x was achieved for solving the same number of equations using Eigen, a C++ template library for linear algebra, as a benchmark. Solving 2,002,000 equations using our *Clustered Jacobi* design on emulator achieved a speed-up of 100.88x over a standard MATLAB Jacobi implementation, whereas a speed-up of 35.24x was achieved in case of using an iterative solver from ALGLIB, a C++ open-source numerical analysis library. In addition to that, higher speed-up could be obtained using a bigger emulator as the design is fully parallelized and the emulation technology supports emulators with a capacity up to 128 AVBs, rather than the 8 AVBs emulator used in this work.

Future research ideas shall focus on optimizing the proposed design of *Clustered Jacobi* as it proved to be more efficient in solving large sparse system of linear equations. Furthermore, implementing hardware solutions for Maxwell's equations using methods other than FEM will be another useful extension.

References

[1] Smith GD. Numerical Solution of Partial Differential Equations: Finite Difference Methods. Oxford, UK: Oxford University Press; 1978.
[2] Strang G, Fix G. An Analysis of the Finite Element Method. Englewood Cliffs, NJ, USA: Prentice-Hall; 1973.
[3] LeVeque R. Finite Volume Methods for Hyperbolic Problems. Cambridge, UK: Cambridge University Press; 2002.
[4] Banerjee PK. Boundary Element Methods in Engineering. New York, NY, USA: McGraw-Hill; 1994.
[5] Saad A. Iterative Methods for Sparse Linear Systems. Philadelphia, PA, USA: SIAM; 2003.
[6] Veselago VG. The electrodynamics of substances with simultaneously negative values of epslon and mu. Sov Phys Uspekhi 1968;10:509–14.
[7] Taylor VE, Ranade A, Messerschmitt DG. SPAR: a new architecture for large finite element computations. IEEE Trans Comput 1995;44(4):531–45.
[8] Valderrama C, Jojczyk L, Possa PD, Gazzano JD. FPGA and ASIC convergence. In: Proceedings of the 7th Southern Conference on Programmable Logic (SPL). Cordoba, Argentina; 2011. p. 269–74.
[9] Davis JD, Thacker CP, Chang C. BEE3: Revitalizing computer architecture research. Technical Report. Microsoft Research Redmond, WA, USA; 2009.
[10] Mencer O, Tsoi KH, Craimer S, Todman T, Luk W, Wong MY, et al. Cube: A 512-FPGA cluster. In: Proceedings of the 5th Southern Conference on Programmable Logic (SPL). Sao Carlos, Brazil; 2009. p. 51–7.
[11] Baxter R, Booth S, Bull M, Cawood G, Perry J, Parsons M, et al. Maxwell - a 64 FPGA supercomputer. In: Proceedings of the 2nd NASA/ESA Conference on Adaptive Hardware and Systems (AHS). Edinburgh, UK; 2007. p. 287–94.
[12] Mentor Graphics Corporation. Veloce emulation platform. http://www.mentor.com/products/fv/emulation-systems/, Last accessed September 2015.
[13] Mentor Graphics Corporation. Veloce user's guide software version 2.1. 2012.
[14] Smith DR, Padilla WJ, Vier DC, Nasser SC, Schultz S. Composite medium with simultaneously negative permeability and permittivity. Phys Rev Lett 2000;84:4184–7.
[15] Ziolkowski RW. Wave propagation in media having negative permittivity and permeability. Phys Rev E 2001;64.
[16] Sadiku MNO. Numerical techniques in electromagnetics. second. Boca Raton, FL, USA: CRC Press; 2000.
[17] Ji Y, Hubing T. EMAP5: a 3D hybrid FEM/MOM code. Appl Comput Electromagn Soc 2000;15(1).
[18] Liebig T, Rennings A, Held S, Erni D. OpenEMS a free and open source equivalent-circuit (EC) FDTD simulation platform supporting cylindrical coordinates suitable for the analysis of traveling wave MRI applications. Int J Numer Model 2013;26(6):680–96.
[19] Durbano JP, Ortiz FE. FPGA-based acceleration of the 3D finite-difference time-domain method. In: Proceedings of 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM). Napa, CA, USA; 2004. p. 156–63.
[20] El-Kurdi Y, Giannacopoulos D, Gross WJ. Hardware acceleration for finite-element electromagnetics: efficient sparse matrix floating-point computations with FPGAs. IEEE Trans Magn 2007;43(4):1525–8.
[21] Shewchuk JR. An introduction to the conjugate gradient method without the agonizing pain. Technical Report. Pittsburgh, PA, USA; 1994.
[22] Zhuo L, Prasanna VK. High-performance and parameterized matrix factorization on FPGAs. In: Proceedings of International Conference on Field Programmable Logic and Applications (FPL). Madrid, Spain; 2006.
[23] Johnson J, Chagnon T, Vachranukunkiet P, Nagvajara P, Nwankpa C. Sparse lu decomposition using FPGA. In: Proceedings of International Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA). NTNU, Trondheim, Norway; 2008.
[24] Garcia JA, Llanos CH, Rincon MA, Jacobi RP. A fast and low cost architecture developed in FPGAs for solving systems of linear equations. In: Proceedings of 2012 IEEE Third Latin American Symposium on Circuits and Systems (LASCAS). Playa del Carmen, Mexico; 2012.
[25] Moussa S, Razik AMA, Dahmane AO, Hamam H. FPGA implementation of floating-point complex matrix inversion based on GAUSS-JORDAN Elimination. In: proceedings of the 26th Annual IEEE Canadian Conference on Electrical and Computer Engineering (CCECE). Regina, SK, Canada; 2013.
[26] Roldao A, Constantinides GA. A high throughput FPGA-based floating point conjugate gradient implementation for dense matrices. ACM Trans Reconfigurable Technol Syst 2010;3(1).
[27] Morris GR, Prasanna VK. Sparse matrix computations on reconfigurable hardware. Computer 2007;40(3):58–64.
[28] Morris GR, Abed KH. Mapping a Jacobi iterative solver onto a high-performance heterogeneous computer. IEEE Trans Parallel and Distrib Sys 2013;24(1):85–91.
[29] Pourhaj P, Teng D, Wahid K, Ko S-B. System size independent architecture for Jacobi processor. In: Proceedings of Canadian Conference on Electrical and Computer Engineering (CCECE). Niagara Falls, ON, Canada; 2008. p. 2033–6.
[30] Greisen P, Runo M, Guillet P, Heinzle S, Smolic A, Kaeslin H, et al. Evaluation and FPGA implementation of sparse linear solvers for video processing applications. IEEE Trans Circuits and Syst Video Technol 2013;23(8):1402–7.
[31] Li J, Huang Y. Time-domain finite element methods for Maxwell's equations in metamaterials. Springer Ser Comput Math; 2013.

[32] Guennebaud G., Jacob B. Eigen: A C++ template library for linear algebra. http://eigen.tuxfamily.org, Last accessed September 2015.
[33] Davis TA. Solving sparse linear systems; chap. 8. Philadelphia, PA, USA: SIAM; 2006. p. 135–44.
[34] Barrett R, Berry M, Chan TF, Demmel J, Donato J, Dongarra J, et al. Templates for the solution of linear systems: building blocks for iterative methods. Philadelphia, PA, USA: SIAM; 1994.
[35] Bochkanov S. ALGLIB: a cross-platform numerical analysis and data processing library. http://www.alglib.net, Last accessed September 2015.

M. Tarek Ibn Ziad received the B.Sc. degree (first class honors) in computer engineering from Ain Shams University, Cairo, Egypt, in 2014. He is currently an M.Sc. student and a teaching assistant in the Department of Computer and Systems Engineering, Ain Shams University, Cairo, Egypt. His research interests include hardware security and system architectures.

Mohamed Hossam received the B.Sc. degree (first class honors) in computer engineering from Ain Shams University, Cairo, Egypt, in 2014. He is currently an M.Sc. student and a teaching assistant in the Department of Computer and Systems Engineering, Ain Shams University, Cairo, Egypt. His research interests include digital design, security systems, and embedded systems.

Mohamad A. Masoud received the B.Sc. degree (first class honors) in computer engineering from Ain Shams University, Cairo, Egypt, in 2014. He is currently an M.Sc. student at the Department of Computer and Systems Engineering, Ain Shams University, Cairo, Egypt and works as a software engineer at Avelabs, Cairo, Egypt. He is interested in cryptography, computer architecture, and hardware acceleration.

Mohamed Nagy received the B.Sc. degree in computer engineering from Ain Shams University, Cairo, Egypt, in 2014.

Hesham A. Adel received the B.Sc. degree in computer engineering from Ain Shams University, Cairo, Egypt, in 2014. He is currently working as a software developer at SABIS Educational Network, Cairo, Egypt.

Yousra Alkabani received the Ph.D. degree in computer science from Rice University, Houston, TX, USA, in 2010, and the B.Sc. degree (first class honors) and the M.Sc. degree in computer engineering from Ain Shams University, Cairo, Egypt, in 2003 and 2006, respectively. She is currently an Assistant Professor in the Department of Computer and Systems Engineering, Ain Shams University.

M. Watheq El-Kharashi received the Ph.D. degree in computer engineering from the University of Victoria, Victoria, BC, Canada, in 2002, and the B.Sc. degree (first class honors) and the M.Sc. degree in computer engineering from Ain Shams University, Cairo, Egypt, in 1992 and 1996, respectively. He is a Professor in the Department of Computer and Systems Engineering, Ain Shams University.

Khaled Salah attended the School of Engineering, Department of Electronics and Communications at Ain Shams University, Cairo, Egypt, where he received his B.Sc. degree (first class honors) in 2003. He received the M.Sc. and the Ph.D. degrees in Electronics and Communications Engineering in 2008 and 2012, respectively. Currently, he is with the Emulation Division at Mentor Graphics Egypt.

Mohamed AbdelSalam received his B.Sc. and M.S. Degrees from Ain Shams University, Cairo, Egypt, and Doctor of Information Science and Technology from Osaka University, Osaka, Japan. He joined Mentor Graphics Egypt 1998–2002 working on software development of circuit simulation and IC layout tools and development of ModuleWare library in FPGA Advantage and again in 2008 in the Emulation Division.