A Scalable Linear Equation Solver FPGA using High-Level Synthesis

Haopeng Meng
The University of Tokyo
Tokyo, 112-0015
meng@kuroda.t.u-tokyo.ac.jp

Kazutoshi Wakabayashi
System Design Research Center
Tokyo, 112-0015
wakaba@dlab.t.u-tokyo.ac.jp

Tadahiro Kuroda
System Design Research Center
Tokyo, 112-0015
kuroda@dlab.t.u-tokyo.ac.jp

Abstract —— This paper mainly describes a scalable linear equation solver in FPGA based on Gauss-Jordan Elimination using high-level synthesis (HLS). A C++ generator is created in this work to obtain the HLS code for synthesis, which is able to balance area and performance of sovler by few parameters. Compared with the traditional RTL design, it has higher design efficiency. In the case of best performance, the solver has time complexity of o(N). Due to the high efficient in design, this scalable linear equation solver also could be used as IP in another design. The result is synthesized in NEC CyberWorkBench HLS, and RTL synthesis in Xilinx Vivado, ZYNQ UltraScale+, and ZCU104 Evaluation Kit at 200 MHz.

Key Words —— High-Level Synthesis, Linear Equation Solver, FPGA, C-Based Design

I. Introduction

In today's engineering applications, finding solution to a large set of simultaneous linear equations is required for a vast variety of problems, such as Finite Element Analysis, real-time circuit simulation, and digital signal/video processing. A linear equation problem can be rearranged into a matrix form, as shown in equation 1, where each equation becomes a row in the matrix.

In the literature, there are two common categories of methods for solving simultaneous linear equations: direct and iterative. Direct methods include LU (lower-upper triangular) decomposition, QR decomposition, and Cholesky decomposition which can typically be used for dense linear systems [8]. Iterative methods include Jacobi, Gauss-Seidel, and relaxation iterations, which are suitable for sparse linear systems[9]. This work focuses on dense linear systems and considers flexible scalable implementation of the solver in FPGA using high-level synthesis (HLS). This can significantly improve design efficiency and make it more convenient to use the linear equation solver as an IP in any other application.

The simplest direct method is to use the Gauss elimination algorithm. The Gauss elimination procedure updates the matrix continuously by applying a sequence of basic row operations to the lower portion of the matrix until the lower left-hand corner of the matrix becomes filled with zeros. This is also the algorithm of LU decomposition. It does not directly output the solution of the linear equations. A method improved from LU decomposition is also used in some FPGA design[2], called Gauss-Jordan Elimination. Unlike the LU method, this not only eliminates the lower triangular of the matrix, but also the upper triangular part. When only the diagonal of the coefficient matrix is left, the solution of the equation could be obtained.

II. Related Work

Linear equation solvers using matrix decomposition are implemented in different forms, such as software programming, systolic array FPGA[1], block LU FPGA[3][5], and GPU accelerator[4][6][7]. The high availability of CPU, the high performance of FPGA, and the efficient application of GPU to solve sparse matrices are reasons of their widespread adoption.

Linear equation solvers using CPU focus on cache hit. Solving linear equation problems requires reading and writing a large amount of data in a short time, so cache hit optimization of CPU software is very important. But no matter how you optimize, there will always be limitations, which are determined by the characteristics of the CPU. Performance bottlenecks will always occur when the problem size is too large.

Field programmable gate arrays (FPGAs) are one type of semiconductor IC device that consists of a large number of reconfigurable logic units, and many programmable input/output blocks and interconnects. Since the programmable logic arrays on FPGAs are massively parallel, they naturally allow for parallel processing of a large amount of data. So good FPGA designs will always achieve higher performance and energy efficiency, even if they typically operate at only 200-300MHz. Systolic array FPGA is therefore a great way to implement linear equation solvers. It has the highest degree of parallelism in theory. But the area is $o\left(N^2\right)$, which makes it difficult to implement in one FPGA for large problems. In addition, due to the use of traditional RTL design, the build is not flexible and works only for a fixed throughput, which often causes excess performance when demand is low.

This work uses HLS for a scalable design and a C++

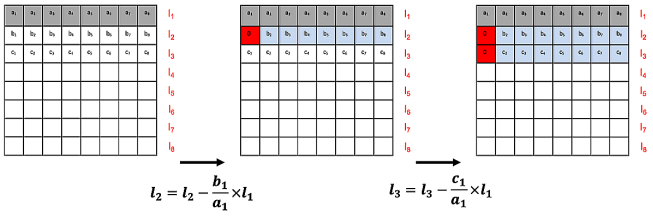

Fig. 1. Row Operation in Gauss Elimination, Making the first element in a row to 0, and update subsequent elements in sequence.

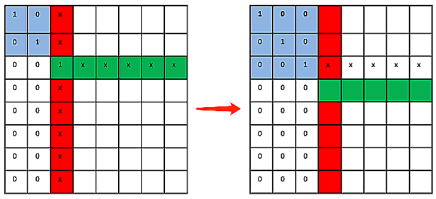

Fig. 2. Column Update in Gauss Elimination. In the same column, only the elements in the diagonal position are 1, and the other positions are 0.

generator to achieve auto-optimization in synthesis. In addition, it delivers a maximum performance of $o(N)$, , same as systolic array FPGA[1]. And compared with VHDL/Verilog, since fewer lines of the code and scalable design by generator is used, the design efficiency is significantly improved.

III. Linear Equation Solver Algorithm

A. Gauss Elimination

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \dots & & & & \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \dots \\ b_n \end{bmatrix} \quad (1)$$

The general system of linear equations is shown as equation 1. It is usually solved using Gauss Elimination. Figures 1-3 illustrate the procedure of Gauss Elimination, which is the foundation of the FPGA linear equation solver being proposed. The main tasks of the repetitive elimination in the Gauss Elimination procedure include three steps: row operation, column update, iteration.

B. Algorithm

Each step of Gauss Elimination is a loop executing a row operation, column update, and iteration as described in Figures 1, 2, and 3 respectively. The code for a triple loop is shown in Figure 4. When solving for a system of linear equations of size N, $o\left(N^3\right)$ time is required. Besides, because of data dependency in

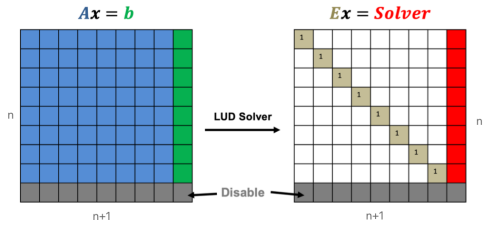

Fig. 3. Row Operation in Gauss Elimination, Iteration to the right in sequence, the left side of the matrix is the identity matrix. On the right is the result of linear equation.

iteration, the steps in Figure 3 cannot be executed in parallel. The row operation and column update can be accelerated by parallel computing. Although the number of operators can be increased to improve performance, it also requires more memory bandwidth, which needs to be balanced.

```
for(int i=0; i<M-1; i++){
        temp=Input[i][i];
/* Cyber unroll_times = 10  */
        for(int j=0; j<N; j++){
                Input[i][j]=Input[i][j]/temp;
        }
/* Cyber unroll_times = 10 */
        for(int k=0; k<M;k++){
/* Cyber unroll_times = 10 */
                for(int m=N-1; m>=0; m--){
                        if(k!=i){
                        Input[k][m]=Input[k][m]-Input[k][i]*Input[i][m];
                        }
                }
        }
        }
```

Fig. 4. Triple loop of Gauss Elimination codes.

IV. Scalable Architecture

The proposed process pipeline for the linear equation solver is shown in Figure 5. The unroll time m defines the degree of parallelism. When m is incresed, the solution time for same problem size is reduced. The pipeline stage number P defines the throughput of the solver. When the problem size and the degree of parallelism m are the same, increasing P improves the throughput of the solver, it used to solve multiple linear equation at the same time.

A. Unroll Time and Memory Acess

The adjustment of parallelism is achieved using loop unroll, as shown in Figure 6. When the unroll time is set to 4, the 4 elements in red are calculated at the same time. The unroll time is set in the HLS code. Here, the larger the unroll time is, the higher the parallelism and the more efficient the computation. When the unroll time is the same as the problem size, it is fully unrolled and the solver reaches its maximum performance. Its time complexity is o(n). However, when the unroll time is increased, higher memory bandwidth is required for calculator access. This limits the improvement of parallelism in large

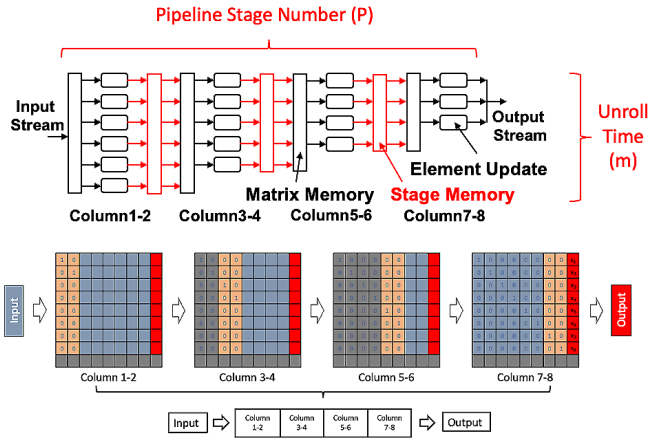

Fig. 5. Scalable pipeline architecture in proposed. P is the define of pipeline stage number. m is the degree of parallelism.

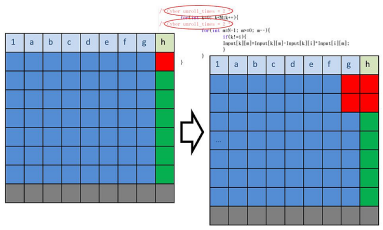

Fig. 6. Parallelize the parts without data dependency by setting the unroll time.

scale linear equation solvers. Therefore, o(n) is usually difficult to achieve in practical applications. Determining how to adjust unroll time m and pipeline stage number to obtain suitable performance is the core purpose of this work.

Figure 7 shows the memory access when unroll time is set to 4 in Figure 6. This allows simultaneous access to four elements in the array to meet the parallel computing requirements in Figure 6. Like the unroll time, the number of memory partitions cannot be increased indefinitely, and it should be set with reference to the unroll time.

B. *Process Pipeline and Throughput*

The effect of pipeline stage number P is shown in Figures 8 and 9. Figure 8 shows the process pipeline execution time of Figure 5. For a system of linear equations of size 8, only two columns are calculated per stage. When the pipeline stage number increases to 6 or 8, the architecture is shown in Figure 9. P is defined in the generator in section V.

C. *Area Optimization*

The amount of calculation in each stage of the solver is not the same. In order to reduce waste of resources, it is necessary to balance the amount of calculation

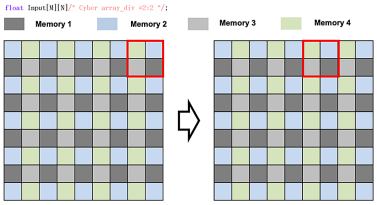

Fig. 7. By setting array div, the data is stored in multiple memories to get parallel access. The figure uses 4 memories and is able to update 4 elements at the same time.It divided memory into 4 parts by 2 times rows and 2 times columns, which is defined in ROM-DIV = Rows:Columns = 2:2.

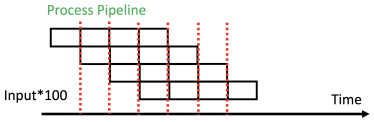

Fig. 8. The throughput of process pipeline in 4-stage pipeline.

in each stage or the number of calculation units to ensure that the delay of each stage is about the same. As shown in Figure 5, the part in grey of the identity matrix does not need data transfer and is not needed by the calculation in the next stage. The optimized structure is shown in Figure 10. Excess memory is removed here (Up).

Besides, since the number of columns is also reduced, the amount of computation for row updates is also gradually decreasing. As shown in Figure 10 (lower half), the total amount of computation for updates in columns 5-8 is less than for updates in columns 1-2. Therefore, 4 stages can be optimized to 3 stages. The throughput and latency are the same as before. This optimization is performed in the generator in section V and is done automatically when problem size N, unroll time M, and pipeline stage P are defined.

V. Generator for HLS

A. *C++ Generator for HLS Overview*

Unlike traditional HLS design, our proposal uses a C++ generator to generate HLS-specific code. This type of design has two advantages: more freedom to customize the scale of the solver and automatic optimization using a program.

Traditional HLS usually has special rules and cannot be directly synthesized using a general C++ program. When the construction is more complex, a large number of functions are usually required to describe it. We use a generator to generate these functions in batches and automatically add HLS-specific parameters, as shown in Figure 11. The example in proposed using different scale of function unit in each stage. When the problem size N, unroll time M, and pipeline stage number P are defined, the generator can output

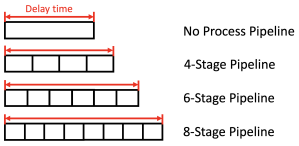

Fig. 9. Increasing the pipeline stage number (P) of solver will make more linear equation to be calculated at the same time. Due to the need for data transfer between stages, the delay time will increase slightly. In practical applications, delay time and throughput need to be balanced.

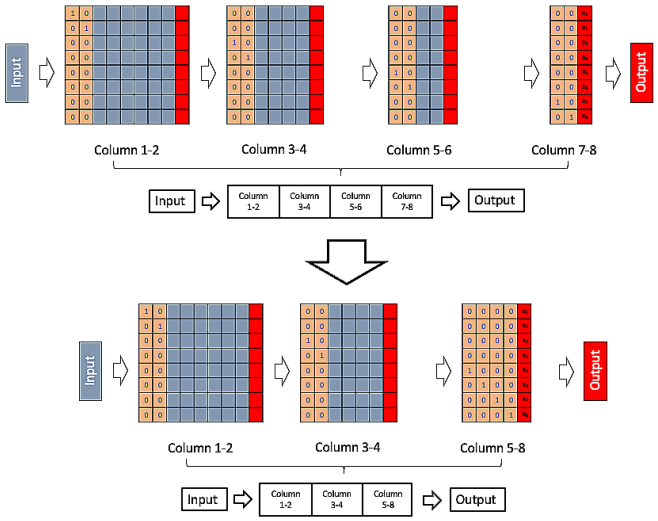

Fig. 10. Balance the amount of computation between stages to reduce the number of stage and area.

multiple designs for trade-off. This can significantly improve design efficiency and facilitate the use of linear equation solvers as IP in larger-scale designs for any application.

B. Area Optimization Procedure

In order to balance the amount of computation, you first need to know the total amount of computation when the problem size is N, and the amount of computation for each column update. As shown in Figure 12, when the k'th column is updated, the elements to be updated are all the elements on the right, calculated by the equation in Figure 12. The totalamount of calculation is shown in Equation 2.

For an example of size 8, the amount of calculation and total amount of calculation of column update can be obtained as shown in Equation 2. When the pipeline stage number is set to P, the amount of calculation is balanced in the following order:

- • Calculate the average amount of calculation per stage, which is obtained by P and C_{all}.
- • The amount of calculation is accumulated in turn, and when the average amount of calculation is exceeded, it is cut off for the current stage, and the rest is moved to the next stage.
- Summarized as segmentation

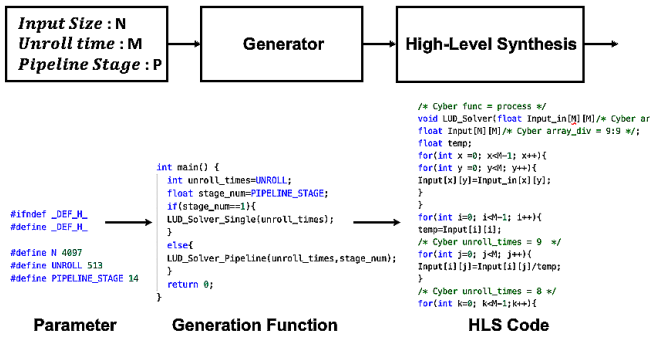

Fig. 11. C++ Generator for HLS

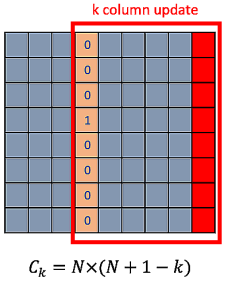

Fig. 12. When N is determined, the calculation amount of each column update is calculated as the formula.

$$C_{all} = \sum_{k=1}^{N} N(N+1-k) \tag{1}$$

$$N = 8,\ C = \{64, 56, 48, 40, 32, 24, 16, 8\},\ C_{all} = 288 \tag{2}$$

When the pipeline stage is set to 2 for size 8, the average amount of calculation is 144, which is Call/2 in equation 2. In the second step, the sum of the amount of calculation of the first two columns is 120, and that of the first three columns is 168. Therefore, it is cut off at the third column, and columns 1-2 and 3-8 are divided into two stages. It can be seen here that the division when P is 8 is invalid. The throughput in P=8 is limited by the amount of computation in the first column. In this step the delay time is affected due to the extra stage.

VI. Evaluation and Result

The result is synthesized in NEC CyberWorkBench HLS, and RTL synthesis in Xilinx Vivado, ZYNQ UltraScale+, and ZCU104 Evaluation Kit. The CPUs used in our comparison are the x86 CPU Xeon E-2146

12 cores @4.5Ghz (max is 4.5GHz) in single precision, and the ARM CPU Broadcom BCM2835 4 cores @1.5GHz in single precision. The HLS design is synthesized at 200MHz.

A. *Synthesis Result*

Table 1 shows the synthesis results for different problem sizes. Since the example is in a small scale, the unroll time m is set to N, same as the problem size, which is called full unroll. It achieves the highest performance for the same size, but it will cost a lot of area. When the size is increased beyond a certain point, full unroll will become impossible. To meet the requirements of the real applications, adjustment of unroll time will be necessary.

Table 1: The synthesis result in different problem size

Matrix Size	LUTs	reg	Block_M	DSP	I/O	Time/ns	Gflops
5	2881	3271	0	0	34/34	628	0.035
6	3061	3132	0	0	34/34	784	0.077
7	4875	4711	0	0	34/34	940	0.131
8	5135	4125	0	0	34/34	1096	0.198
9	9906	8635	0	0	34/34	1252	0.278
10	9564	6908	0	0	34/34	1408	0.371
11	10114	5479	0	0	34/34	1564	0.476
17	39895	21571	0	0	34/34	2452	1.406

Table 2 shows the synthesis results for different pipeline stage numbers. When problem size is the same, increasing pipeline stage number P will significantly improve performance at the expense of latency. Full unroll is also used in this example, which means the unroll time is the same as problem size.

Table 2: The synthesis in different pipeline stage

Size	Pipeline Stage	LUTs	reg	Block_M	Delay Time/ns	Time Per-In/ns	Gflops
5	2	5082	6177	800	688	456	0.048
5	4	8460	11540	2400	736	243	0.091
9	1	9906	8635	0	1252	1252	0.278
9	2	12460	8846	2592	1426	888	0.392
9	3	15354	13149	5184	1552	716	0.486
9	4	17514	17134	7776	1681	548	0.635

B. *Comparison*

Figure 13 shows the latency comparison with CPU and conventional systolic array FPGA. For the triple loop of Gauss Elimination, the CPU execution time is $o\left(N^3\right)$. For the systolic array FPGA, it is $o(N)$. Its area is fixed and performance may be an overkill when demand is low. And a large scale systolic array is difficult to design. In our proposed automatically generated HLS design, it is easy to balance between performance and area by changing the setup parameters (N,M,P).

As shown in Figure 14, the biggest advantage of our proposed solution is flexibility in design trade-off. Unlike CPUs, increasing (N,M,P) of our solver will

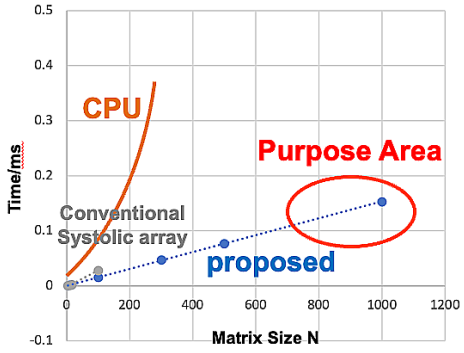

Fig. 13. Design purpose and latency comparison with cpu and systolic array.

definitely bring performance improvement and area increase. This certainty will facilitate decision-making in practical applications. In addition, it is more convenient to be able to use the linear equation solver as an IP to be a part of a very large-scale ASIC.

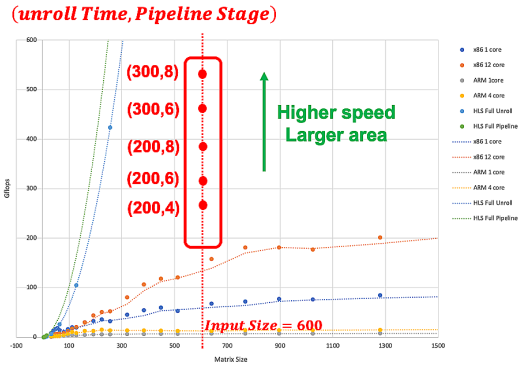

Fig. 14. Performance comparison with CPU. By changing problem size N, unroll time M and pipeline stage number P, any scale of solver could be synthesised for application requirement.

Table 3 shows the performance comparison with the systolic array FPGA from another paper. When the size is small, the structure of the systolic array becomes superior. When the size increases, the advantage of the proposed HLS design gradually appears. In larger scale designs or applications requiring flexible design, the proposed HLS design becomes convenient.

Table 3: Performance comparison with systolic array

	N=4	N=8	N=10	N=16	N=18	N=30	N=32	N=64	N=128
Proposed	0.035	0.278	0.476	1.41			6.16	25.68	104.82
Systolic Array[1]	2.4		6.5		13.11	19.11		37.88	no reported

Performance: GFlops Proposal/FPGA: Virtex-7 300Mhz

Acknowledgements

The test data was provided by NEC corporation, which is irreplaceable in this research. Without the help, this paper could not have been finished in time.

This paper is based on results obtained from a project,JPNP16007, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] **LIU Shu-yong, WU Yan-xia, ZHANG Bo-wei, ZHANG Guo-yin, DAI Kui, "Research of parallel hardware architecture for matrix triangularization decomposition based on reconfigurable computing system,"** ***Acta Electronica Sinica*****, Vol. 43, pp. 1642-1650, 2015.**

[2] **Zhenhua Jiang; Sayed Ata Raziei, "An efficient FPGA-based direct linear solver,"** ***IEEE National Aerospace and Electronics Conference (NAECON)*****, 2017.**

[3] **Manish Kumar Jaiswal; Nitin Chandrachoodan, "FPGA-Based High-Performance and Scalable Block LU Decomposition Architecture,"** ***IEEE Transactions on Computers*****, Vol. 61, pp. 60-72, 2012.**

[4] **Tingxing Dong; Azzam Haidar; Piotr Luszczek; James Austin Harris; Stanimire Tomov; Jack Dongarra, "LU Factorization of Small Matrices: Accelerating Batched DGETRF on the GPU,"** ***2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)***

[5] **Baoguang Fang; Shuqiang Chen; Xulong Wei, "Single-precision LU decomposition based on FPGA compared with CPU,"** ***2012 International Conference on Computational Problem-Solving (ICCP)***

[6] **M. J. Inman, A. Elsherbeni, C. Reddy, "CUDA Based LU Decomposition Solvers for CEM Applications,"** ***ACES JOURNAL*****, Vol. 25, 2010.**

[7] **G´eraud P. Krawezik, Gene Poole, "Accelerating the ANSYS Direct Sparse Solver with GPUs,"** ***Symposium on Application Accelerators in High-Performance Computing (SAAHPC)*****, 2009.**

[8] **Tingxing Dong; Azzam Haidar; Stanimire Tomov; Jack Dongarra, "A Fast Batched Cholesky Factorization on a GPU,"** ***2014 43rd International Conference on Parallel Processing***

[9] **Manashwi Tamuli; Shreyasee Debnath; Ashok Ray; Swanirbhar Majumdar, "Implementation of Jacobi iterative solver in verilog HDL,"** ***2016 2nd International Conference on Control, Instrumentation, Energy & Communication (CIEC)***